Master Thesis

“Pattern Recognition and Machine Learning Applications for Embedded Systems”

Technical University of Crete
Department of Microprocessors and Hardware

Panos Kalodimas (Author)
Yannis Papaeystathiou (Supervisor)
Chania (Greece), April 2016

CHAPTERS

FIGURES

TABLES

DIAGRAMS

1. Master Thesis Outline

The first part of the master thesis was the part of studying the preparing for the implementation of the TSM Algorithm. In this stage except of studying over the X. Zhu and D. Ramanan “Face detection, pose estimation and landmark localization in the wild” paper [1], other related paper had to be studied too in order to understand and analyze the algorithms structure and methodology. One of these paper is the one from which the TSM algorithms come from, the "Object Detection with Discriminatively Trained Part-Based Models" paper [2]. As the master thesis demanded general knowledge around the computer vision and pattern recognition, extra studying over these Computer Science fields had to done. For example, the book “Computer Vision: A modern Approach” [24] by David Forsyth and Jean Ponce was used for that purpose. The difficulties on this stage was the fact that, without any background knowledge, in a short time period a lot of new Computer Science Fields had to be learned and combined in order to understand a state of the art algorithm. We have to refer that this master was done in the Microprocessors and Hardware department of TUC, a department not specialized in Computer Vision and Machine Learning fields.

The second stage of the master thesis working was the implementation of the TSM algorithm using the C\C++ programming language. The algorithm was offered by the creators in Matlab script using some parts write in C++ as the Matlab tool did not offer implementation for every procedure. These procedures where the HOG, Convolution, Resize, Reduce and DT procedures. Although these procedures where already implemented in C++ the designer had implemented them using the Matlab array memory format. This means that this implementation was reading the array data column by column instead of line by line as the C array memory format does. For that reason this procedures had to be rewritten and debugged.

1	6	11	16	21
2	7	12	17	22
3	8	13	18	23
4	9	14	19	24
5	10	15	20	25

Figure 1 - Matlab Arrays Memory Format

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Figure 2 - C Arrays Memory Format

One of the greatest difficulties in the implementation of the TSM algorithm was the debugging procedure. Millions of data had to be tested in order to be sure that the procedures implementation had no errors. The solution to that problem was the usage of Matlab tool. Every procedure we implemented we called through the Matlab tool and we receive the return data

inside the Matlab environment. Every part of the algorithm we implemented used inside the creators implementation and the returned data were compared with the data the creator's implementation return. These processing was much easier and faster than doing it in C\C++. Although this solution helped us a lot make us save a lot of time, it cost us a considerable amount of time on creating special libraries for formatting the data from the Matlab array memory format to C and vice versa. For the debugging procedure we had to create a full set library functions for converting all the data structures the TSM algorithm needed from the Matlab format to C and the opposite. Despite the effort of creating these libraries, the advantage we got worth the trouble.

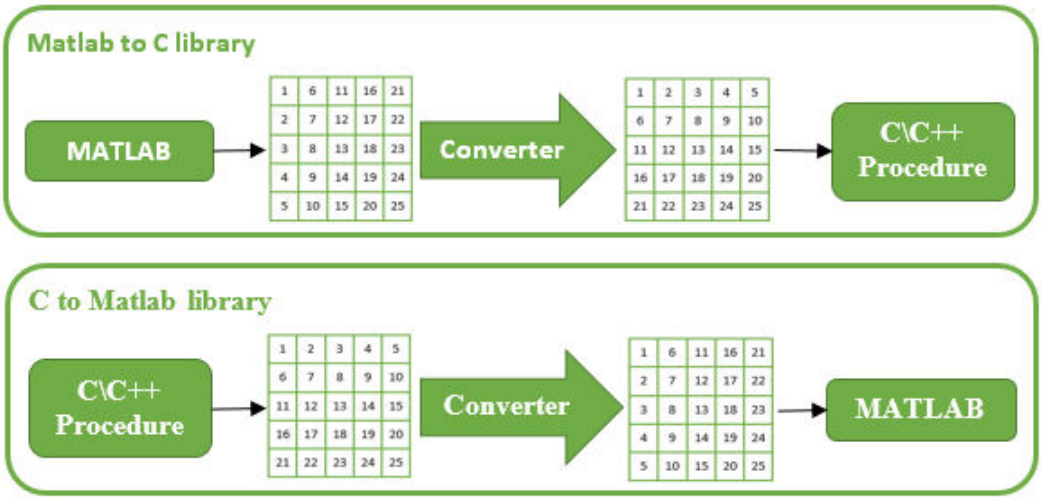


Figure 3 - Mat2C Library Diagram

Another part of the implementation was the creation of different kind of versions of the algorithm. This part was not difficult but as long as the parallelism was used the time consumption of testing all these versions was extended. Every different parallelization technique had to be tested for all the versions to see its effect on the TSM algorithm execution time. Sometimes we had to take decisions in order to reject some versions because the number of versions would increase in an out of scope number.

The Research part of this thesis was also a great time consumer. As happens in the research field there were techniques designed, that in the end were rejected as they did not offer any useful results and they are not mentioned in this thesis despite the fact that a lot of time was spent to be implemented and tested. During the research period a lot of time was also needed for testing the results of the implemented patches in order to see their effect in the TSM algorithm performance. We had to let the algorithm run for hours to get these results, as the sample images used for testing were 205 images of multiple, usually large, sizes. We needed about 10 hours for a single test. Also a lot of Matlab scripts had to be written in order to make automatic the procedure of data analysis. The uncertainty of the research was a difficult but on the other hand constructive part of this thesis working time.

At last the writing stage of this thesis was also a great time consumer. The main reason for this delay was the fact that this thesis had a huge amount of data analysis. For every graph presented in this thesis lots of data had to be processed. Hundreds of Excel files were used in order to process these data and create useful graphs by them. We also had to create our own profiler inside the TSM algorithm implementation code in order to derive the data needed for these analysis. Multiple Matlab scripts had to be written in order to profile the algorithms memory consumption and regularize the data in order to be graphically presented. This thesis contains 129 diagrams, 182 tables and 94 figures the majority of which are custom made. All those diagrams and tables shown in this thesis caused us a lot of effort and time but they are a necessary part of it we could not omit.

2. Master Thesis Abstract

In this thesis a new implementation of the “Face Detection, Pose Estimation, and Landmark Localization in the Wild” [1] algorithm by Xiangxin Zhu and Deva Ramanan is represented. This implementation was firstly designed for being used by embedded systems but finally it can also be used by large multiprocessors systems. This is because the modern embedded systems tend to be similar to what we used to call multiprocessor systems years ago. Because of the huge needs of the market in the area of embedded systems (smart-phone, tablets and more) the latest embedded system are in the category of small multiprocessor systems using from 2 to 4 and even more cores in their central processing unit.

Our implementation of the “Face Detection, Pose Estimation, and Landmark Localization in the Wild” algorithm was implemented in basic C\C++ as there is no usage of any external C\C++ library in the core of the algorithm. This gives the algorithm the ability to be used in both Windows and UNIX systems with no further changes. It also allows further improvements and alteration as it is easily readable for those who would like to use it for custom application. Our implementation gives the ability of customizing the functionality of the algorithm through a set of settings and parameters that can easily be modified.

As this implementation is designed for usage in embedded systems the need of reducing memory consumption and processing speedup was encounter. For that reason a number of customizations were made in contrast to the original implementation of its creators. There were also produced a set of techniques that some may pull down the algorithm’s performance but in contrast they offer extra speedup and memory saving. These techniques may be very useful for custom application.

Despite any further speedup the main problem of making the face detection task a great time consumer is the fact that the image size in the one that makes it a long time processing. Large images compel the system to create large image pyramids in order to search them for face detection. In addition the larger the top image is the more time is needed to be processed. The main solution on this problem is proposed is the scaling of the original image to a smaller size in order to reduce the number of data needed to be processed. This solution makes the systems faster but they lose part of their performance as scaling an image to a smaller size makes small size faces to be unable for detection. Our implementation offers a method that scans the image pyramid faster for face detections in order to avoid detection processing in pyramid levels that seems to be empty of faces. This can be a very effective method for video application where empty faces frames can be faster processed and rejected.

3. Related Work

As far as we knew, no previous work was introduced jointly addressing the tasks of face detection, landmark localization and pose estimation until the June of 2012 when X. Zhu and D. Ramanan proposed the “Face detection, pose estimation and landmark localization in the wild” [1] work. This work was supposed to be the state-of-the-art that time and was used as a baseline for further research leading to the presentation of more proposals for systems trying to make the face detection process a much faster and efficient. To succeed this, new models was used except of discriminant parts models like neural networks. The neural networks are considered to be the more efficient and fast models that can detect faces and estimate pose. We are not going to mention all of them but only the most recent like [3], [4], [5], [6] and [7]. The most similar work to [1] is the [8], [9] and [10].

Our work does not try to present a new face detection or object detection method but to make the Discriminant Part Models and Tree Structural Model systems faster and less memory consumption ones. For this reason the only related work that can be referred is the [25] that implements the same algorithm. The reason of choosing this algorithm is because except of face detection and pose estimation it also offer landmark localization of the 68 or 39 (depends on the viewing angle) human face landmarks. Another task it also implements is the face detection of faces in the range of over 60 degrees viewing angle. Many algorithms have been deployed since then, like [26], [27], [28], [29], [30], [31] and [32] but most of them do not offer all these tasks the same way. Many of them do not offer landmark localization at all or they detect few of them, the most significant for the face detection (ex. Eyes). The need of the landmarks localization demand the convolution procedure of at least 68 cascade windows of the image features space that is a very heavy procedure. Others does not offer pose estimation at all while the most of them that does, only offer pose estimation in the range of 60 degrees. Only the [25] does offer the complete set of tasks and it’s the one to compare with.

As far as we know, there are also many other freeware algorithms offered in the web but none of the uses the TSM method meaning that all of the have a lack of tasks. They usually offer face detection or/and pose estimation but not the 68 landmark localization or face detection in more centered faces as referred in the previous paragraph. Some of these algorithms are [3], [4], [5], [6], [7], [8] and [10].

4. TSM Algorithm Simple Description

The “Face Detection, Pose Estimation, and Landmark Localization in the Wild” [1] algorithm was created by Xiangxin Zhu and Deva Ramanan from the University of California, department of Computer Science on 2012. On this algorithm XiangXin Zhu and Deva Ramanan presented a unified model for face detection, pose estimation and landmark localization in the real world. It is a model based on a mixture of trees with a shared pool of parts, which represent facial landmarks, and used to capture topological changes due to viewpoint.

The creators claimed for achieving reliable estimates of head pose and facial landmarks, particularly in unconstrained “in the wild” images. They presented a single model that simultaneously advanced the state of the art for all three. It is a novel but simple approach to encoding elastic deformation and three-dimensional structure using mixture of trees with a share pool of parts. They define a “part” at each facial landmark and use global mixtures to model topological changes due to viewpoint. Different mixtures are authorized to share part templates which allow the model a large number of views with low complexity.

They presented an extensive evaluation of their model for face detection, pose estimation and landmark localization. They compared to the state-of-the-art from both the academic community and commercial systems such as Google Picasa and face.com. In terms of face detection, their model substantially outperforms Viola-Jones and is on par with the commercial systems above. In terms of pose and landmark estimation, their results dominate even commercial systems. Their results are particularly impressive since their model is trained with hundreds of faces while commercial systems use up to billions of examples.

No previous work had jointly addressed the task of face detection, pose estimation, and landmark estimation until then. Their system is also trained discriminatively, but with much less training data, particularly when compared to commercial systems.

4.1. Face Detection Based on Parts Based Detection

The “Face Detection, Pose Estimation, and Landmark Localization in the Wild” [1] algorithm is based on the “Object Detection with Discriminatively Trained Part Based Models” [2] by Pedro F. Felzenswalb, Ross B. Girshick, David McAllester and Deva Ramanan. This algorithm is an object detection system based on mixtures of multi-scale deformable part models. In the Tree Structural Model (TSM) algorithm the mixtures are one scale deformable part models.

The Deformable Parts Based Detector (DPBD) algorithm, it tries to detect specific parts of an object within an image using trained filters. After the object detection the usage of the mixtures of trees is taking place. The algorithm checks the locality of the detected parts and the location

correspondence between those detected parts to make a conclusion if they are bringing forward the object we are looking for or they are just dispread parts within the image. As the filters used for object detection are all the same size, different size objects are detected in different scales of the image that is why it is based on mixtures of multi-scaled deformable part models.

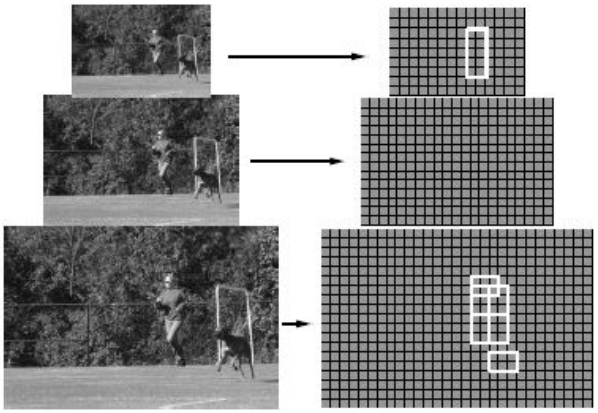


Figure 4 - DPBD Algorithm Root and Child Parts Detection

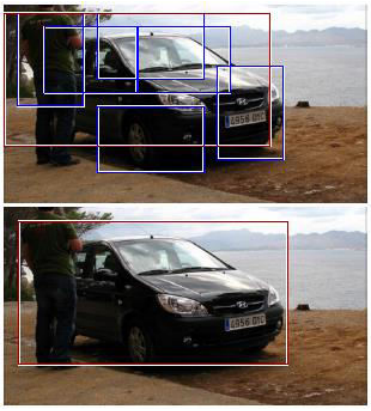


Figure 5 - DPBD Algorithm Root and Child Parts Locality

The DPBD algorithm uses a root filter to detect the object is looking for and a set of multiple filters to detect specific parts inside the object the root filter detects. The combination of those results gives the final approval of the correctness of the detection (Figure 4 and Figure 5). The set of filters used for the parts detection needs different scales of the image as these parts are obviously smaller that the main object. For example if a car is the object the algorithm is looking for, the wheals, the lights and other parts of it are all smaller than the car's shape itself, that is why the system is multi-scaled, as the algorithm has to search inside lower scales of the image to detect these parts.

In the Figure 6 below the full diagram of the DPBD algorithm is shown. The algorithm uses two features maps of the image with resolution ratio of two. The small feature map is used for applying the root filter and the second one for the child parts filters. Adding the filter's responses of all the parts gives the final results of the detection procedure.

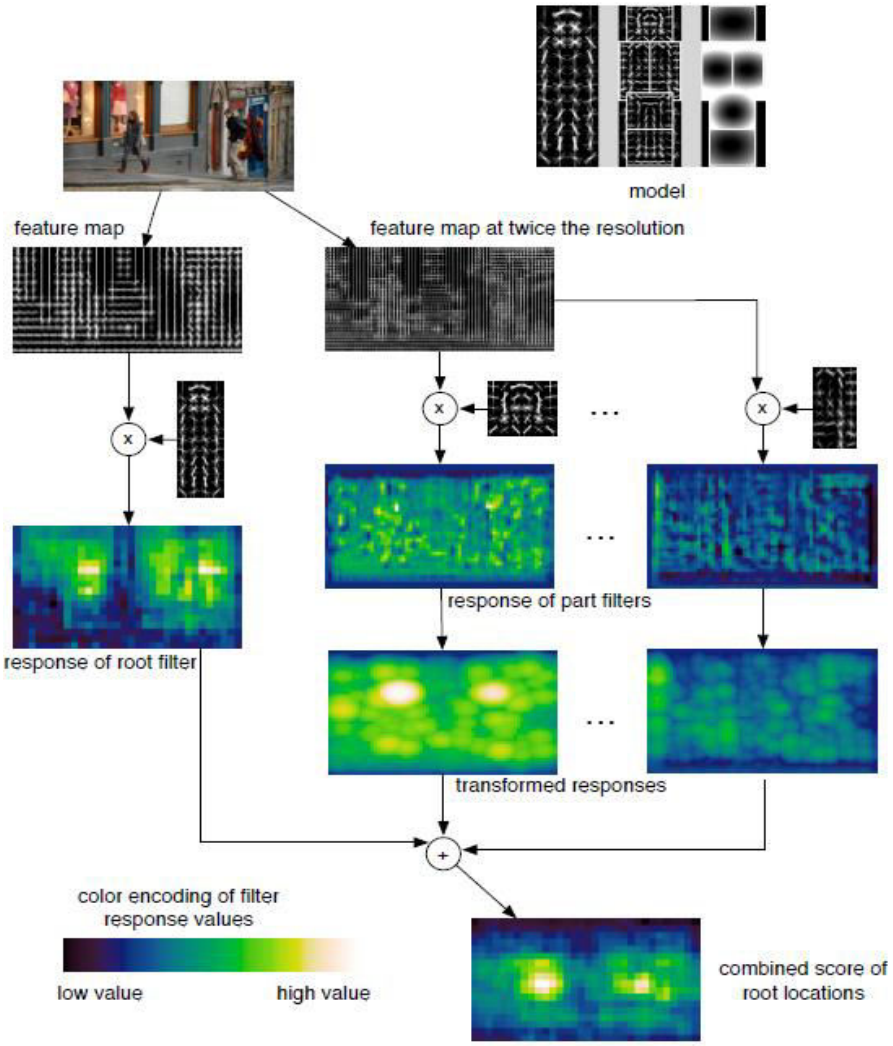


Figure 6 - Deformable Parts Based Detection Algorithm Execution Flow Diagram

4.2. TSM Face Detection Algorithm

On the "Face Detection, Pose Estimation, and Landmark Localization in the Wild" [1] algorithm there is a small but important difference. This algorithm does not use a main root filter for the detection of human face but only the combination of a set of parts (Figure 7). This small difference gives us a good flexibility during the implementation. The algorithm is only trying to detect specific parts of the human face and checks the location correspondence to figure out if they fit to the face template it is trained.

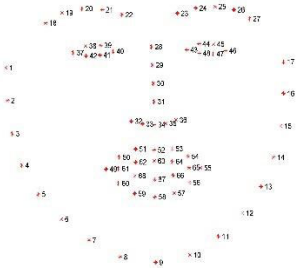


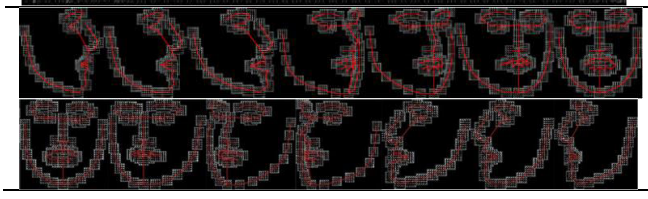
Figure 7 - Human Face 68 Landmarks

For the pose estimation, our algorithm uses 13 different pose model trees each of which represents a different point of viewing a human face by the step of 15 degrees viewing angle. The one achieving the best score is the one recognized.

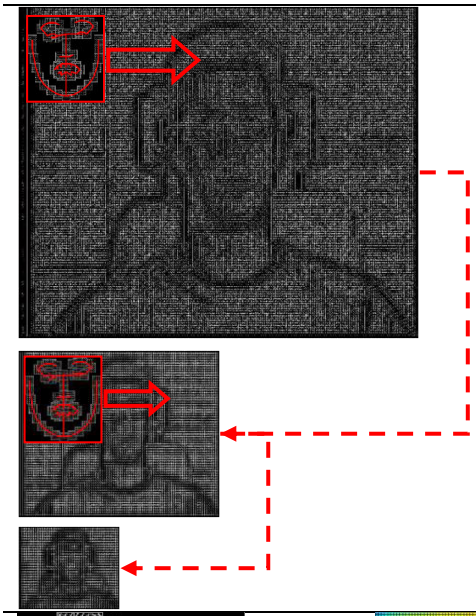
As described before, the filters used for detecting face landmarks are one size so in order to detect different sized faces within the image the algorithm has to apply the detection procedure over a series of image's scaled copies. The detection procedure does not use simple images but the HOG descriptors of them. The series of the HOG images of the scaled copies of the original image is called the features pyramid of the image and it is described in detail in chapter 5.5. On all these HOG images the algorithm applies the detection procedure for all the different pose model trees. At the end of this procedure the algorithm selects the top detection as the most accurate. This is a simple abstract of the way the algorithm works. In the next chapter (Chapter 5) a more detailed description is presented with deeper analysis on every stage of the algorithms detection procedure.

The Image

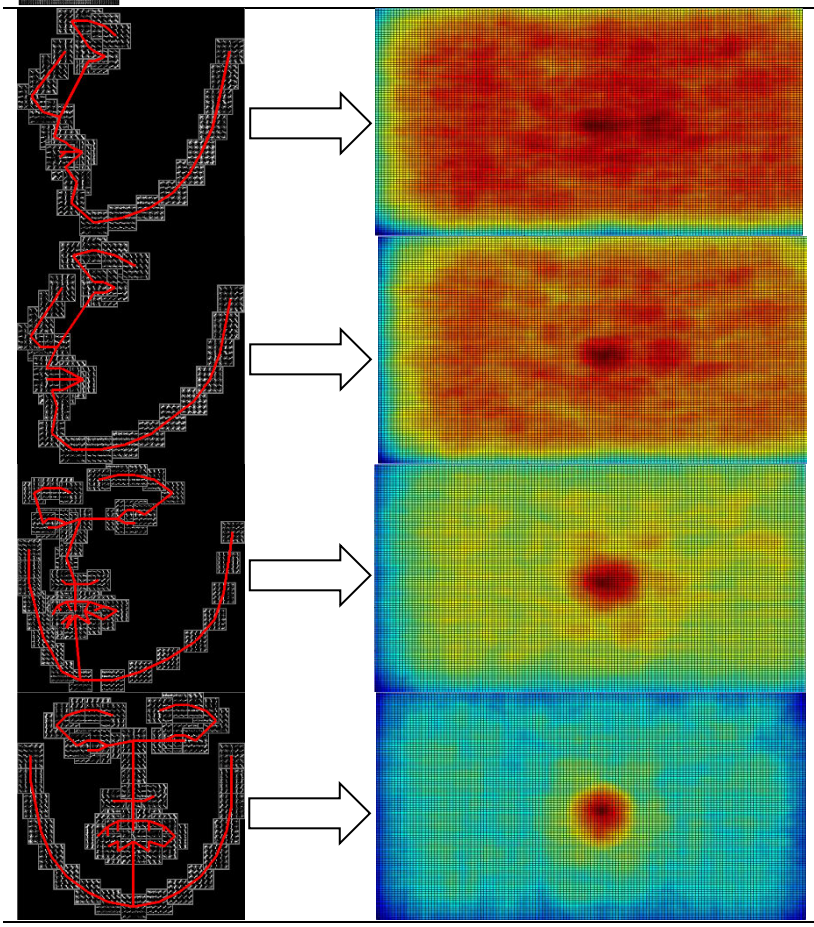
Create Feature Pyramid



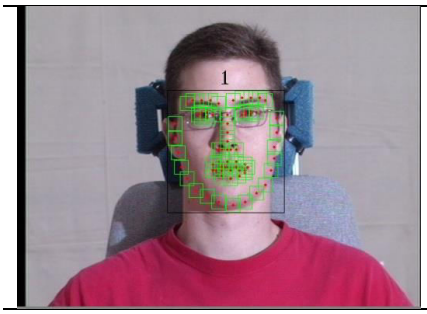
Pose Model Trees



Detect each Pose within the Image



Compare Results



Select the highest scores

Figure 8 - TSM Algorithm Execution Flow

5. TSM Algorithm Procedures Description

The TSM algorithm it was used in this thesis uses some well-known and widely used procedures of the Computer Visio science field. In this chapter a short description on them is appose as they might not be already known by the reader.

5.1. TSM Face Detection Algorithm

In the previous chapter the way the "Face Detection, Pose Estimation, and Landmark Localization in the Wild" algorithm works was described in a few words. In this chapter a detail description of the algorithm is referred.

The detection process consists of five sequential procedures.

- **Feature Pyramid:** Having an image for processing, the algorithm firstly creates its image pyramid. By the image pyramid the algorithm gets the feature pyramid of the image by applying a HOG procedure.

- **Convolution Stage (Filter Responses):** The next step is to convolve all the filters used for detecting facial landmarks with every level of the features pyramid. This means that using the Model of total 99 filters and having a feature pyramid of 20 levels, this step is a procedure of 1980 convolution procedures and the production of 1980 different results stored in 20 lists of 99 elements. This is a very heavy procedure. The result of a convolution between a filter and a pyramid level is called the «Response of the filter».

- **Distance Transformation (DT Scores):** This procedure is the processing of the convolutions result in order the algorithm to decide whether there is useful information at the results. It is a procedure where the results of the parts over the features or in other words the landmarks over the image have to be partial combined in order to produce a face contour. For that purpose the algorithm is using a tree model where information about the position of each part according to its parental part exists. This process is achieved by applying multiple distance transformations and additions between the parts filter responses. All this processing is ends up to a results array called the «Score» of the procedure. This array data reveal the existence of any detections.

- **Find & Backtrack (Result Cache):** As soon as the distance transformation stage finishes, the algorithm checks the final result for high-scored values. High scored values means face detection. By the time that high score values exists inside the score table the algorithm starts a process called «Backtrack» were the position of the landmarks within the image is estimated. The results of the Backtrack procedure are the results returned by the algorithm

with information about the position of every landmark. All results are saved in a results table called «Results Cache».

- **Non-Maximum Supreme (NMS):** At the end of the detection process, the algorithm has to make a selection between the detection results as many detections does not mean multiple faces within the image but also multiple detections of the same face.

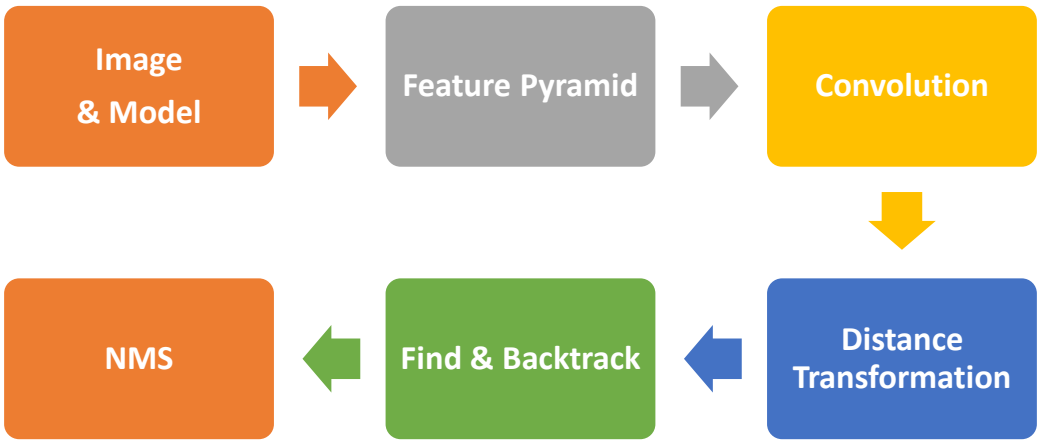


Figure 9 - TSM Algorithm Procedures Sequel

A more detailed description of each phase of the detection process is represented in the following subchapters.

5.2. Model

The Tree structural model is used in the TSM algorithm for face recognition contains a variety of data and parameters used during the recognition and estimations procedures. We will describe the most important as it is necessary for understanding how the algorithm works.

On a human face there are a lot of landmarks that can be used for face recognition as shown in Figure 10. Every landmark of this kind is called a part. A human face inside an image can be appeared through a variety of points of view depending on the angle the head of the face's owner had the moment the image was captured. This indicates that lots of the parts of the human face can probably not be visible on some points of view. Many parts of a human face can also look different when seen from different points of view. This point also indicates that a standard set of parts cannot be used for face detection. For that reason the algorithm's model contains a set of different parts for every 15 degrees of viewing angle starting for -90 degrees to +90 degrees for total 13 different pose angles. This method gives as also the pose estimation.

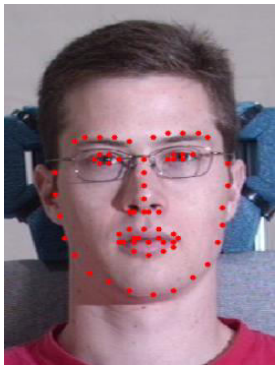


Figure 10 - Human Face Landmarks

Every set of parts used for detecting faces in a specific point of view is called a component. As the angle distance of every component with its vicinal is only 15 degrees some parts may appear tiny defacements, so we can use the same part (landmark) to more than one components. Another characteristic of the human face is its proportion. This proportion produces a similarity between the mirrored components. As a result the majority of the parts that are used by a component can also be used by its mirror component.

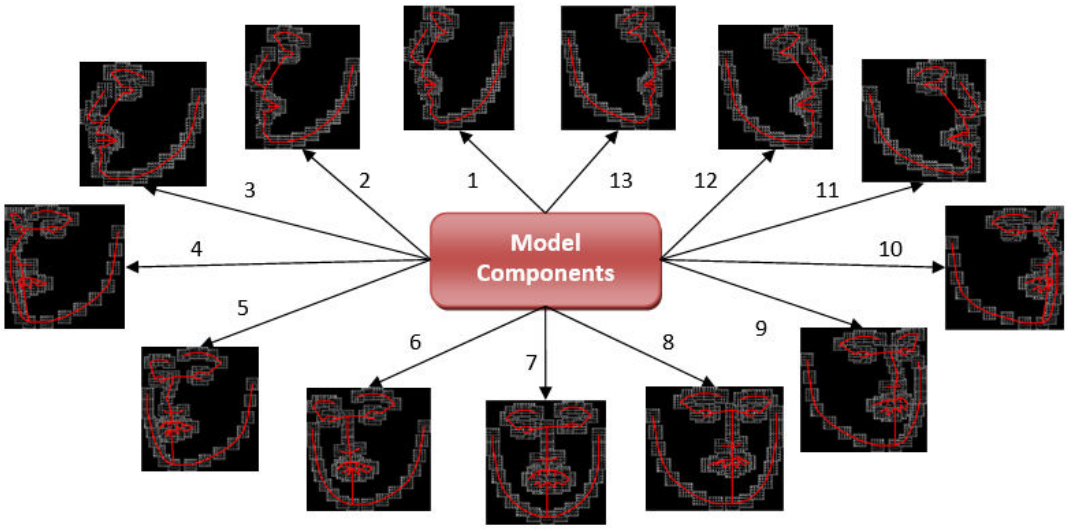


Figure 11 - TSM 13 Components

The above remarks conclude to a model that can use only a few amount of parts for a total of 13 components. The creators offer two Models for face detection. One using only 99 filters and one using 146. The second one appears to be more accurate as long as the detection results but the first one is faster. On this thesis we are mainly focused on the 99 filters model as we care more for a fast implementation running on embedded systems. Despite that no substantial difference exist between those two models and all the important information concerning the algorithm are referred for both models.

In both 99 and 146 filters models the median component (centered pose of 0 degrees angle) uses 68 parts for its recognition. All the components used for recognizing faces at most of 45 degrees viewing angle use the same amount of parts when the rest ones use only 39 parts. This means that a fusion of 710 to 99 (and 146) parts is achieved by using the same parts on multiple components. This is a very important achievement for the time performance of the algorithm as is explained later on chapter 6.

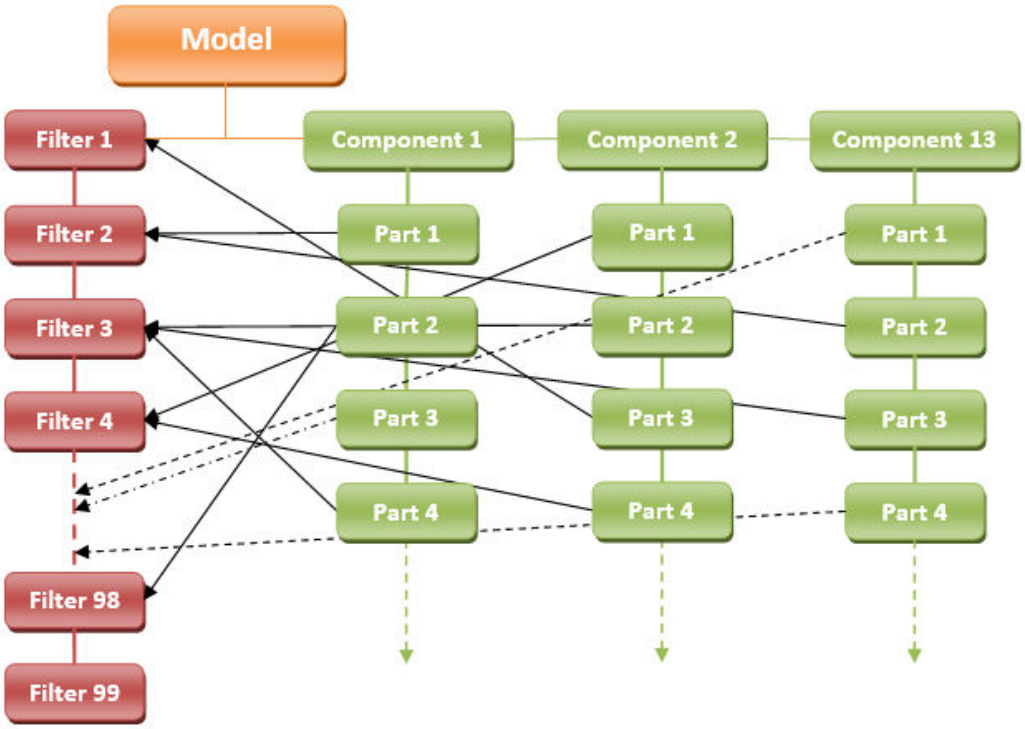


Figure 12 - TSM Parts and Filters Connection Structure

In both models (99 and 146 filters) the middle components (4-10) representing faces of -45 to 45 degrees viewing angle use the same filters for landmark detection. The position between them is the criteria for individualizing them. On the other hand the filters used by the edge components (1-3 and 11-13) representing -90 to -60 and 60 to 90 degrees are not always the same. On the 146 filters model the left and the right edge components use their own set of 39 filters. This is how the number 146 comes from (Table 1). On the other hand on the 99 filters model only the half of the edge components part's filters are unique while the rest are borrowed by the parts of the middle components as also shown in Table 1.

Table 1 - TSM Components Mutual Parts

Filters	Components												
	1	2	3	4	5	6	7	8	9	10	11	12	13
99	16/23	16/23	16/23	68	68	68	68	68	68	68	15/24	15/24	15/24
146	39	39	39	68	68	68	68	68	68	68	39	39	39

Every part of the model is associated with a three dimensional filter that is used in the detection process in order the landmark that the part represents inside the image to be discovered.

Every component uses an amount of parts. These parts are connected in a tree style hierarchy. The reason of doing that is because the position of each part according to the rest ones inside

the image produces the conclusion of a face existence. The tree model of the component 7 is shown in the Figure 13 below.

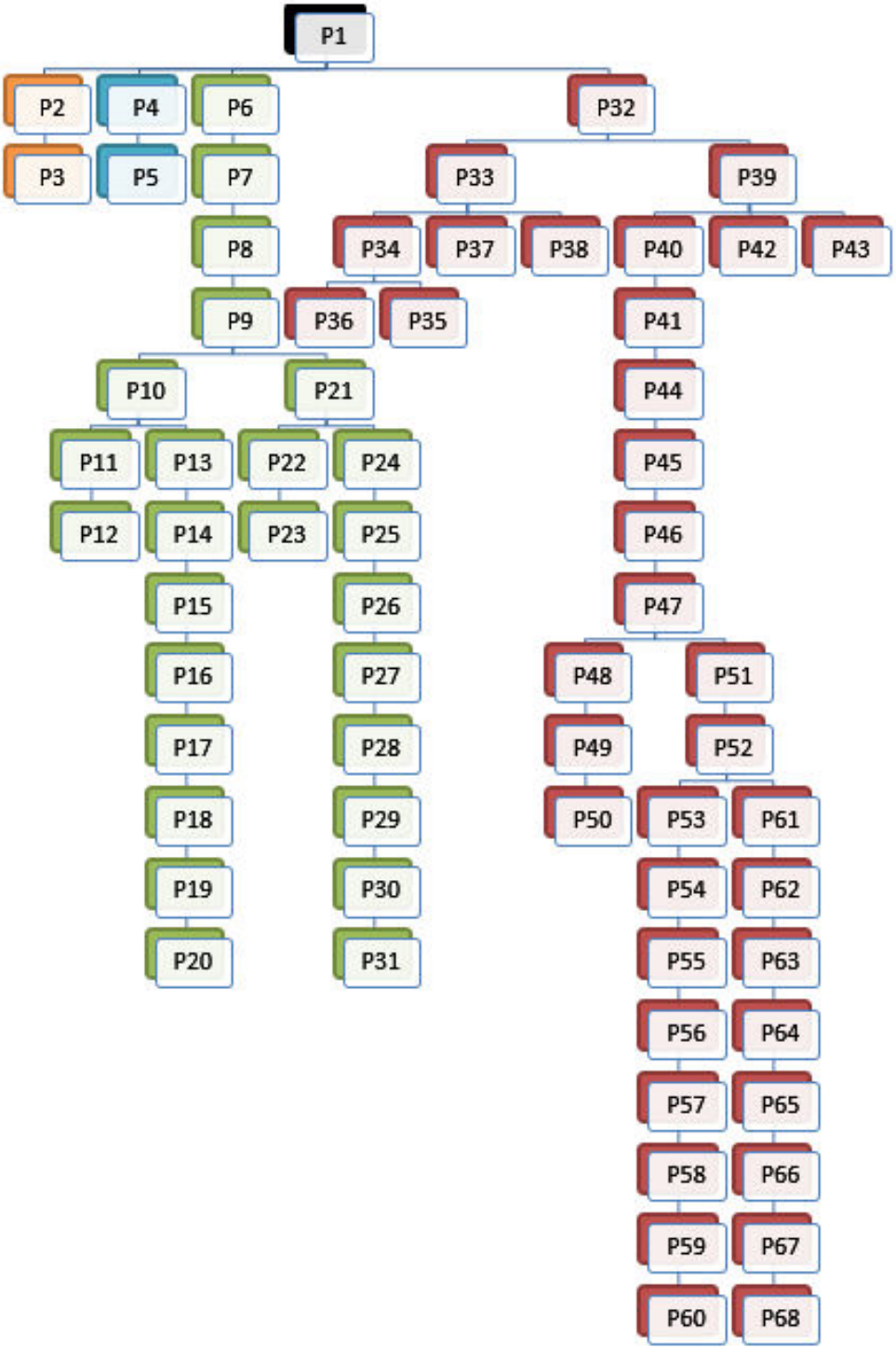


Figure 13 - TSM Component 7 Parts Tree Structure

5.3. Image Pyramid

An image pyramid is a collection of multi-scaled representations of an image. The parameter «Levels» of an image pyramid is the number of scaled images in the pyramid and the «Interval» one is referring to the number of levels exist in the pyramid between two images with scale ratio of 2. In the Figure 14 below an image pyramid of 12 levels and interval parameter set to 4 is presented. For further reading use [11].

Figure 14 - Image Pyramid Example

5.4. HOG

The Histogram of Oriented Gradients is feature descriptors used in image processing for object detection. There are more than one feature descriptors in computer vision but this one is considered to be the most accurate and suitable for human detection as described by Navneet Dalal and Bill Triggs in 2005 [12] and that's why it is used as a part of TSM algorithm. In the Figure 15 below a visual representation of the HOG descriptors of two images is shown.

Figure 15 - Histogram of Oriented Gradients Descriptors Example

The idea behind the HOG descriptors method is that the shape and the characteristics of the objects within an image can be described through the intensity of oriented gradients and edge directions. The way for doing that is by dividing the image into small boxes of pixels called cells and calculate the histograms of gradients direction or edge orientation within each cell. The combination of these histograms represents the descriptor.

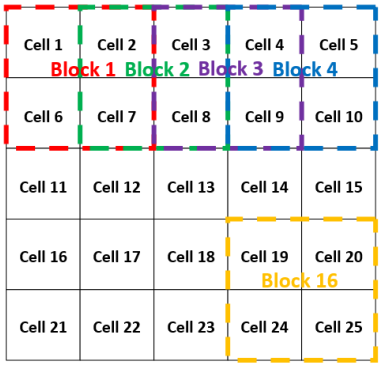


Figure 16 - HOG Cells and Blocks

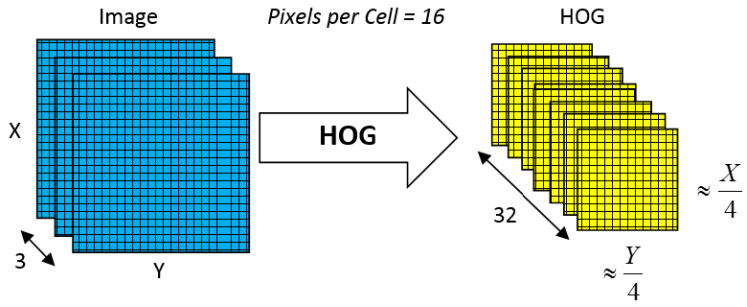


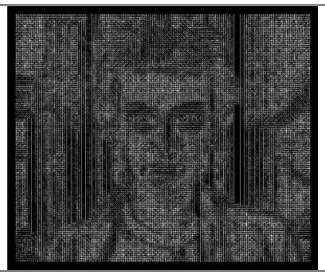
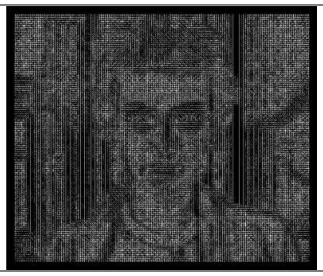
Figure 17 - TSM Algorithm HOG Procedure Data

The improvement of the descriptor can be achieved using normalization methods against illumination differences and shadowing. This normalization is applied separately on groups of cells called blocks and not in the whole image at once for better accuracy (ex. Shadows). Using cells and blocks, the HOG descriptor method keeps a good tolerance against geometric and illumination transformations and that's a good. Transformations affect more when using large regions of pixels within a cell.

In the TSM algorithm the HOG stage gets a 3 levels (colors) $Width * Height$ array and returns a 32 levels $\frac{Width * Height}{Pixels - Per - Cell}$ one. This array is the «Features image» of this image

5.5. Feature Pyramid

The first thing the TSM algorithm does is creating a feature pyramid of the image. A feature pyramid is similar to an image pyramid but instead of scaled patterns of the image it uses scaled patterns of the histogram of oriented gradients of the image. The creation of a feature pyramid demands the existence of the image pyramid as its more accurate to scale the image first and the get its HOG than create its HOG and scale it afterwards. The last option does not produce the desirable results as deferent scales of an image produce different kind of HOGs as is shown in Figure **18**.

	Original Image	**HOG from Image Pyramid**	**HOG from Scale**
$1/1$			

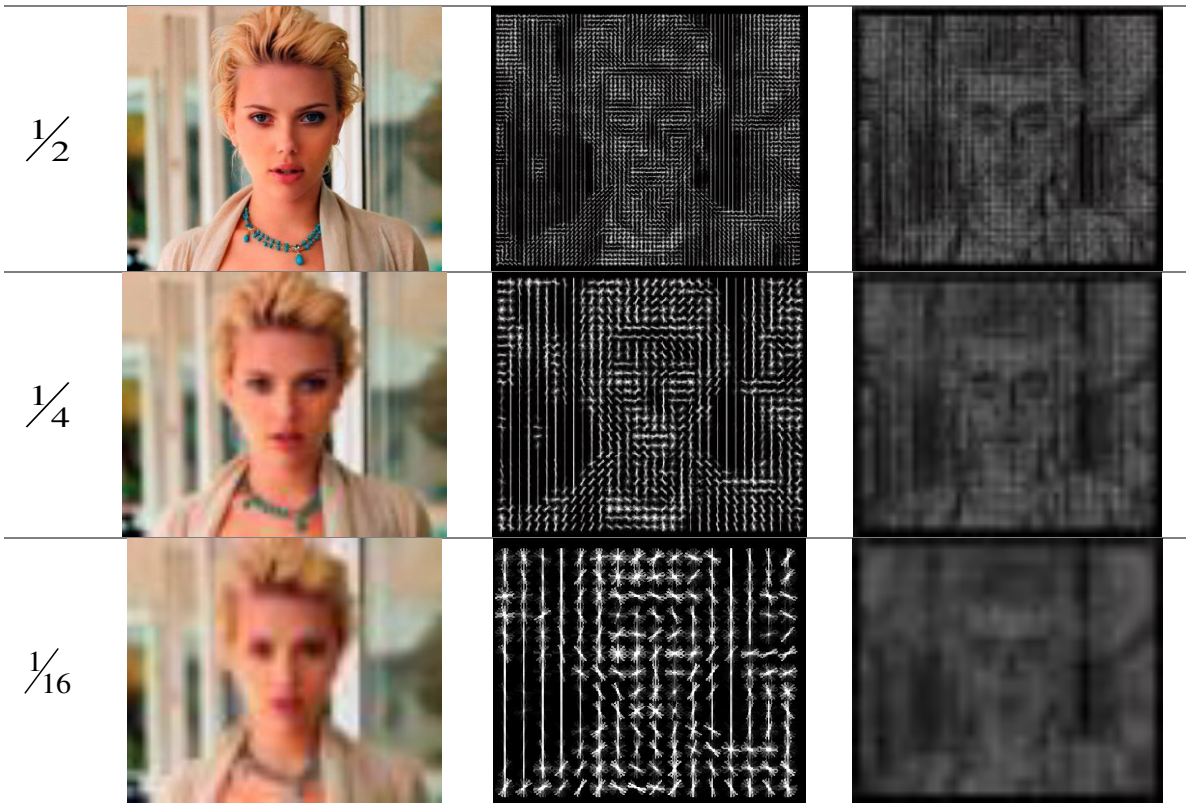


Figure 18 - Features Pyramid from Image Pyramid vs Scaled HOG Images

In the Figure 18 above in the third column is presented the HOG images coming from the images at the second column. The first column images are getting blurred as moving downwards because they are smaller size than the top one in the scale noted at the first column. On the last column the HOG images comes from the top HOG image at the same column scaled by the scale factor at the corresponding first column. It is clearly visible that the HOG images at the third column are much more accurate than the ones at the forth column. This is why the features pyramid comes from the image pyramid and not by scaling the HOG images. As is obvious the features pyramid of the face detector algorithm is comes as the third column of the Figure **18**.

There are three parameters in the features pyramid that have to be explained

- **Interval**: The Interval parameter defines the number of levels exists between two levels with scale ratio of two, as explained in chapter 5.3. This parameter defines a measurement of the density of the pyramid. A low density pyramid can cause the escape of detections as our model detects faces of a specific size. The higher the density is the more accurate the algorithm is. In addition to accuracy the higher the density is the more hardware resources are needed to execute the algorithm and the detection process last more time. The creators of the algorithm have define this parameter value to 5 as the most efficient.

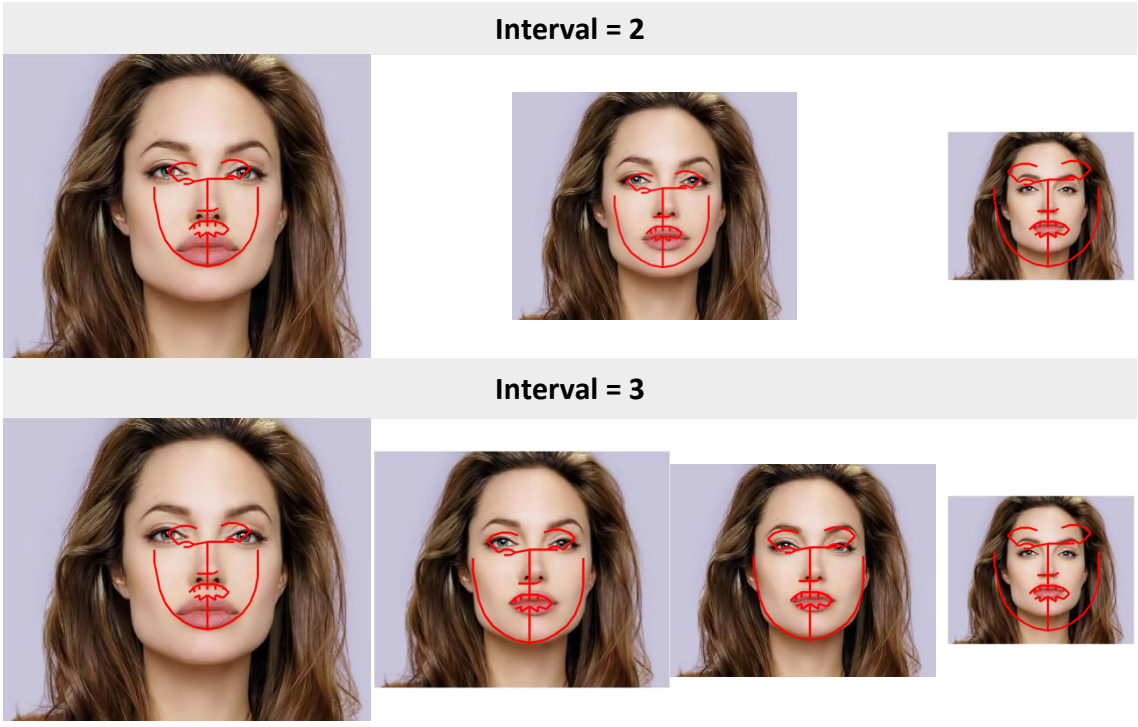


Figure 19 - TSM Algorithm Interval Parameter Impact

In the Figure 19 above the features pyramid at the top is using an interval parameter of 2 in contrast to the bottom one using an Interval parameter of 3. As shown by the red lines over the images the most accurate detection is succeeded in the third level of the right features pyramid. The left pyramid fails to have such an accurate detection and it might probably miss the detection.

- **MinLevel**: This parameter defines the minimum level of the image pyramid that will be used for detection. As the model detects faces of a specific size, the minimum this value is, the smaller is the size of the faces within the image that can be detected. The maximum is the MinLevel parameter value is the greater the size of the faces within the image must be.

- **MaxLevel**: The MaxLevel parameter defines the length of the image pyramid and it affects the maximum size of a human face within the image that can be detected. If the MaxLevel parameter is low value then large faces within the image may not be detected. In contrast to the MinLevel parameter, this parameter affect much less the algorithm execution time and memory resources needed as in the end of the feature pyramid the images' size tend to be smaller in addition to the beginning.

Table 2 - TSM Features Pyramid Parameters Defaults	
Interval	5
MinLevel	1
sBin	4
MaxLevel	$1+\left\lfloor \frac{\log\left(\frac{\min(image.size)}{5*sbin}\right)}{\log\left(2^{\frac{1}{\text{interval}}}\right)} \right\rfloor$

- **Sbin**: This parameter represents the number of pixels each side of the HOG cell tile uses. The value of this parameter affects the size of the features image the HOG process produces as described is chapter 5.4. As referred in this chapter the HOG process produces features images smaller than the original ones at a scale factor of the Sbin parameter value. This means that the features pyramid levels are all Sbin times smaller than the respective ones on the respective image pyramid.

Table 3 - TSM Algorithm Features Pyramid per Image Size			
Image Size	Levels	Max Level Size	Min Level Size
320x240	18	86x66x32	13x11x32
640x480	23	326x326x32	13x11x32
800x600	25	406x306x32	13x11x32
1024x768	27	518x518x32	13x11x32
1280x960	28	646x646x32	13x11x32

For building the features pyramid the algorithm creators used two procedures. The first one resizes the image according a scale factor and the second one creates an image half the input image. That's because as explained in chapter 5.3 all the images in the pyramid with level distance equal to the Interval parameter have scale ratio equal to two.

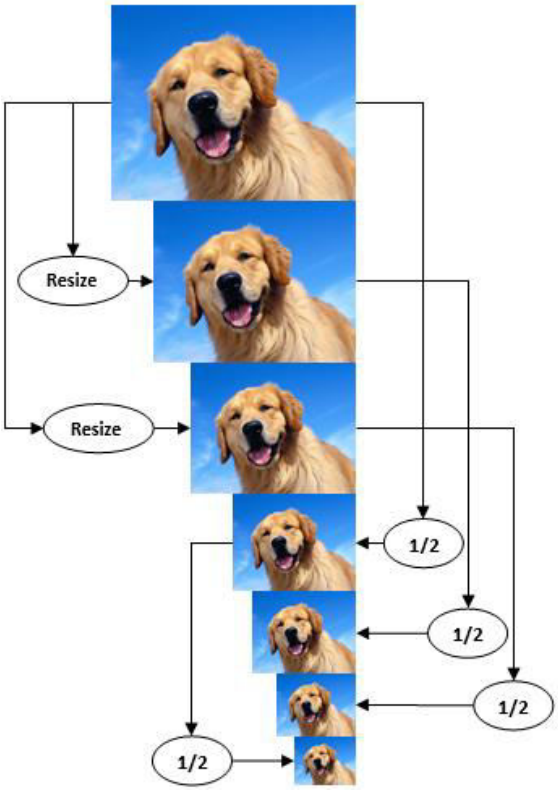


Figure 20 - TSM Algorithm Image Pyramid Creation Execution Flow

5.6. Convolution

The convolution process is a well-known one in the area of image processing. It is the procedure of applying a filter over an image. In the TSM algorithm the convolution process is used for part detection over the features images. As mentioned in chapter 5.2, the algorithm's models contains a set of either 99 or 146 filters. Each filter is used for a human's face landmark detection. By convolving each filter to the image features map, high score pixels appears in the place where the landmark exists.

In the convolution process, the image is a HOG descriptors image, a 3D flexible array and the filter data is also a 3D array in the stable size of 5x5x32. The result of the convolution is in contrast a flexible 2D array as shown in Figure 21 below.

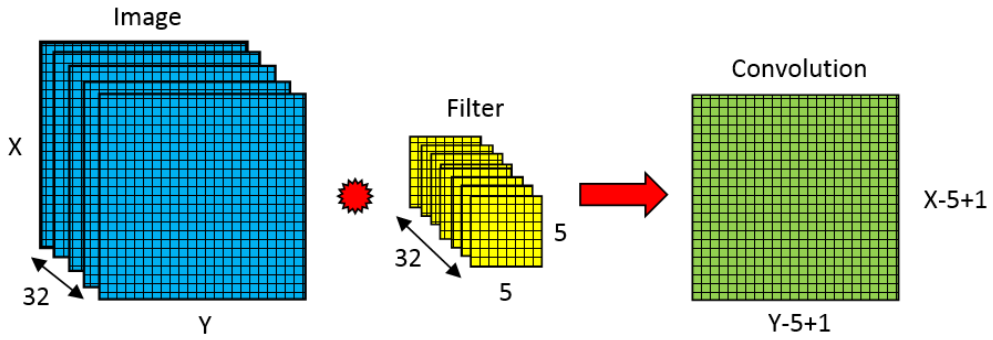


Figure 21 - TSM Algorithm Convolution Procedure Data

In the TSM algorithm the convolution process is repeated for all filters for every level of the features pyramid. In the creators implementation at Matlab the pyramid reaches the 23 levels for a 640x480 pixels image. This means that 2277 (23*99) convolution processes occur during the algorithm execution. This is the most memory and CPU consumption stage of the algorithm although it is the less complicated. In Table 4 the number of convolution procedures occur in the face detection one according to the input image size.

Table 4 - Convolution Procedure Calls per Image Size

Image Size	Levels	99 filters Model	146 filters Model
320x240	18	1,782	2,628
640x480	23	2,277	3,358
800x600	25	2,475	3,650
1024x768	27	2,673	3,942
1280x960	28	2,772	4,088

In the Figure 22 below a visualization of the convolution results is shown.

Image

Lips

Nose

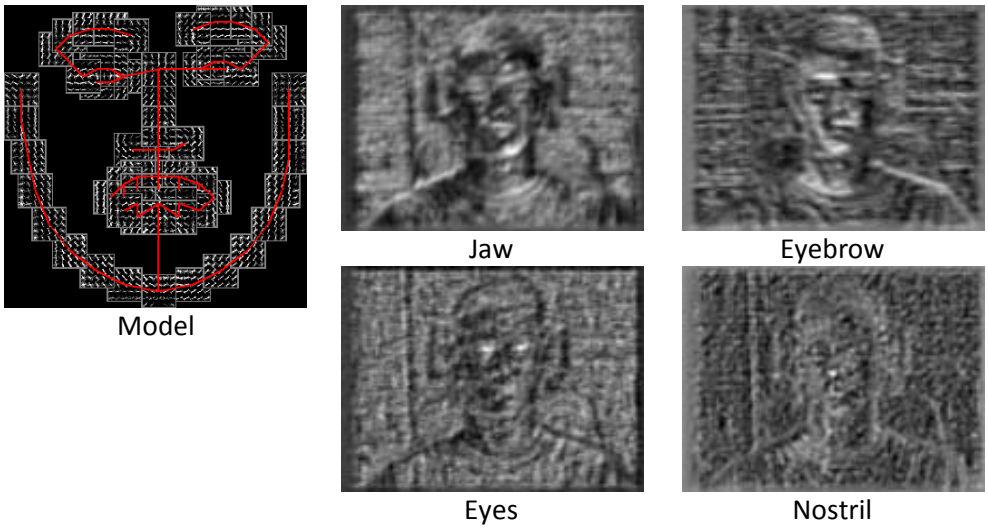


Figure 22 - TSM Algorithm Convolution Results Examples (Visualized)

By the convolution process a series of results arrays comes. These arrays are called as «Filters Responses» and consists one of the basic data structures of the algorithm as they allocate a great amount of memory. For every convolution process a filter response array comes. At the end of the convolution process the total number of arrays produced by the convolution process is equal to the number of the levels of the features pyramid multiplied with the number of filters used by the model. The total amount is the same shown in Table 4.

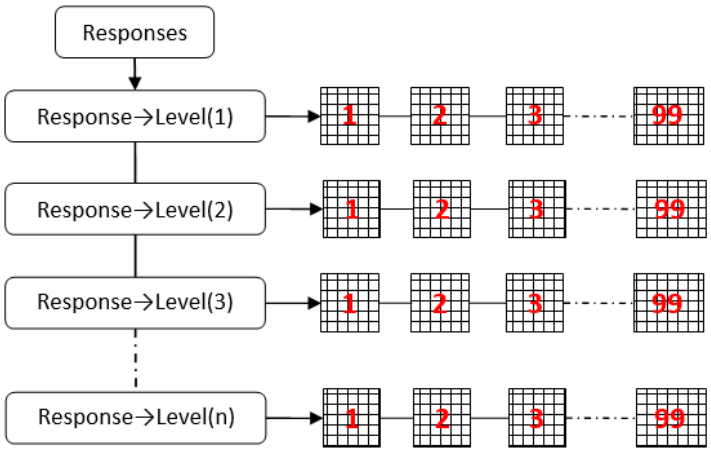


Figure 23 - TSM Algorithm Filters Responses Data Structure

5.7. Distance Transformation

Distance transformation is a method used in computer vision, image processing and pattern recognition for comparison of binary images, especially when these images are results of feature detection. The distance transformation technique specifies the distance from each pixel to the nearest non-zero pixel.

On a binary feature image the distance transformation produces an image map where all non-feature pixel have a value corresponding to its distance to the nearest featured pixels. It's a representation of the features cost to each pixel.

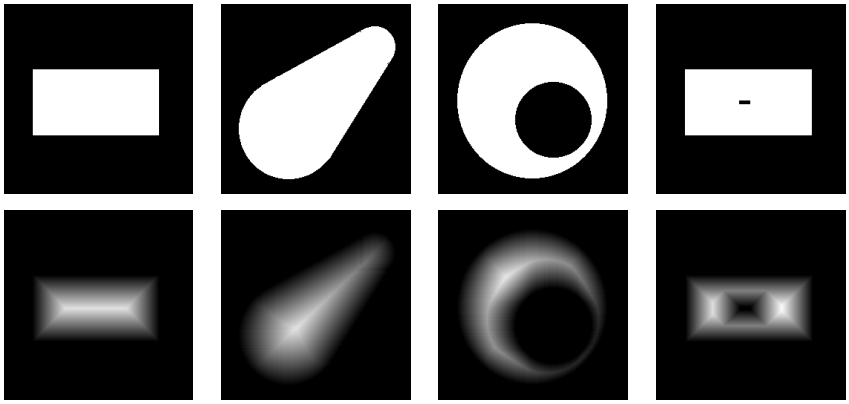


Figure 24 - Distance Transformation Examples

In our algorithm the implementation of distance transformation is used is the Pedro F. Felzenszwalb and Daniel P. Huttenlocher [13] one as it is one of the fastest. The distance transformation stage does not contain just an execution of a distance transformation process but a sequential execution of the process for every part of the model tree. The algorithm climbs the tree from the leaves to the root adding each parts' score to its parent's one just after it applies the distance transformation process as shown in Figure 25.

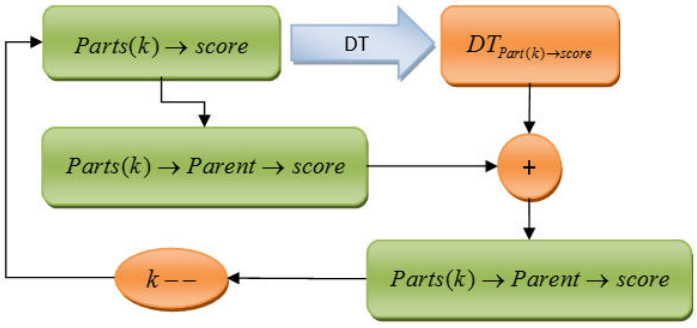


Figure 25 - TSM Algorithm Distance Transformation Procedures

In a simple trial of visualizing this process a summary of it is shown in Figure 26 and Figure 27. In Figure 26 a summary of this process applied on the model tree of component 13 is represented and an extendible representation of its last branch (68 to 61 leaf) in Figure 27.

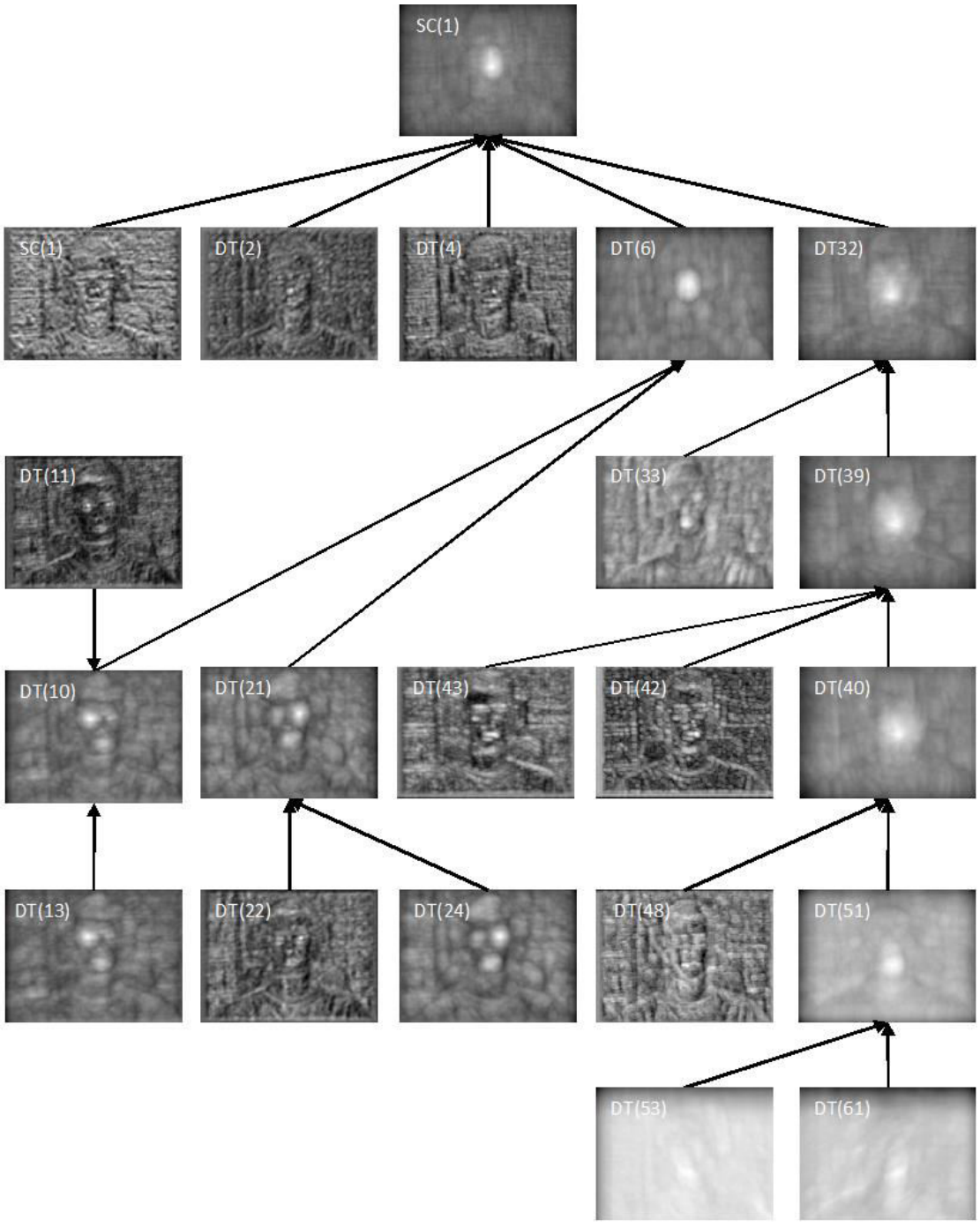


Figure 26 - TSM Algorithm DT Results of Component 7 Tree Example (Visualized)

In the Figure 26 above is visible that after applying the distance transformation process multiple times at last the final image comes of this procedure is an image with high-score pixels (white pixels) in the place where the human faces exists.

In the Figure 27 below a detailed representation of how the distance transformation procedure works on the detection process. Using the filters responses produced by the convolution

procedure the algorithm applies the distance transformation process on it and add the parental filter response according to the pose's model tree.

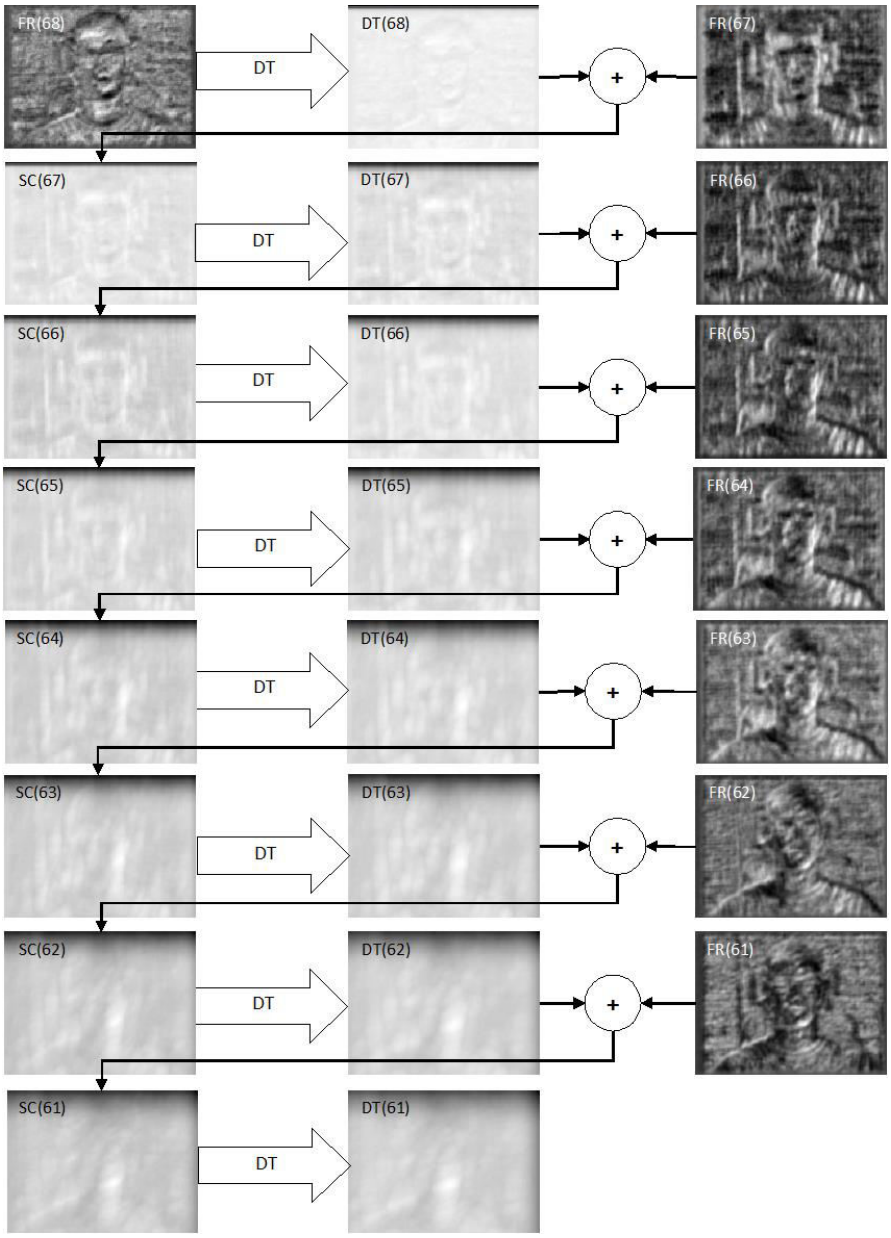


Figure 27 - TSM Algorithm DT Results of Component 7 Tree Leafs 61-68 Example (Visualized)

The result arrays form the distance transformation process are called as «DT Scores» and are those data that are passed in the next stage, the Backtrack stage (Chapter 5.8 and 5.9), for further processing. These arrays are two for every part of the pose tree, except from the root one, plus one with the whole tree score. The tree score array is the one where the detection is discovered while the others are used by the Backtrack procedure for the landmark localization one. The number of DT Score arrays produced in the TSM algorithm is large as shown in the

Table 5 below and it is independent by the number of the filters the TSM algorithm model is using (99 or 146 filters).

Table 5 - TSM Algorithm DT Scores Arrays per Image Size		
Image Size	**Levels**	**DT Scores**
320x240	18	25,092
640x480	23	32,062
800x600	25	34,850
1024x768	27	37,638
1280x960	28	39,032

5.8. Find

At the end of the sequential distance transformation procedure the Find procedure is returning the coordinates of the high-scored pixels within the image. It just makes a selection of the scores values that is considered to be detection results. The Threshold parameter that defines the limit over which a pixel value is considered a detection is set by the creators in the value of -0.65.

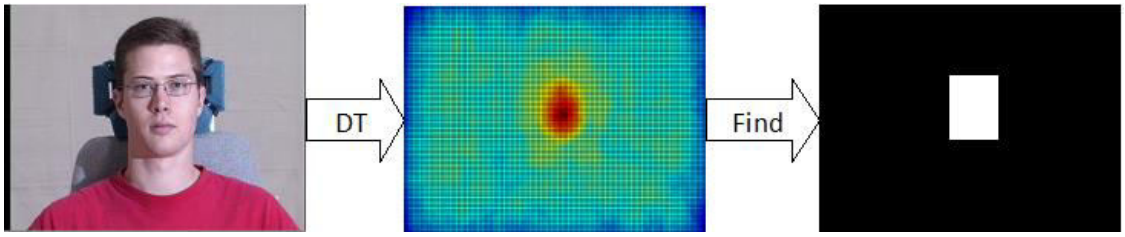


Figure 28 - TSM Algorithm Find Procedure Results

By observing the algorithm results during the profiling process (more details in chapter 6.2), we noticed that the find procedure discovers high-score values not only at the place of an existing human face but in different occasions. These occasions are,

- **One face, multiple poses detection**: When a human face exists within an image during the detect process the majority of the poses trees produce high-score values. Small viewing angles differences at the pose trees is sensible to create similar results.

- **One face, multiple scale detection**: When an image illustrating a human face is used for creating an image pyramid it is sensible that the models would detect the same face in multiple nearby levels of the features pyramid. As larger is the interval parameter, explained in chapter 5.5, of the features pyramid more the levels where the same face is detected would be.

- **One face, multiple high-scores**: As is visible in Figure 28 above, after the distance transformation process the results around the highest score have similar values close to the highest one. The threshold used for selecting the highest value cannot be accurate as different images creates different high-scores. The threshold value comes after several tests using several different input images. As a result it is impossible for the algorithm to use a Threshold parameter value that would select only one high-score value after the DT process.

More details about the find process results are presented in chapter 6.13.1.

5.9. Backtrack

Backtrack procedure is the part of the algorithm that makes on the landmark localization. Even if there was no interest in landmark localization, this stage would be needed for localizing the face detection. The Backtrack procedure is a resources cheap process and is only executed when detections come up. What is necessary to be mentioned is that the Backtrack procedure produces a landmark estimation set for every high-score pixel the find procedure discovers. This means that a series of landmark positioning sets candidates comes from the Backtrack procedure. The final selection of the most accurate sets comes from the NMS procedure based on each candidate's high-score value and its position within the image that is explained in chapter 5.10.

All the Backtrack procedure results (Candidate detections) are stored in a Results Cache array. This array size is set to 10,000 results cells by the creators. Every time this array is full the NMS procedure (Chapter 5.10) is called in order to free array cells from inaccurate and duplicated detections.

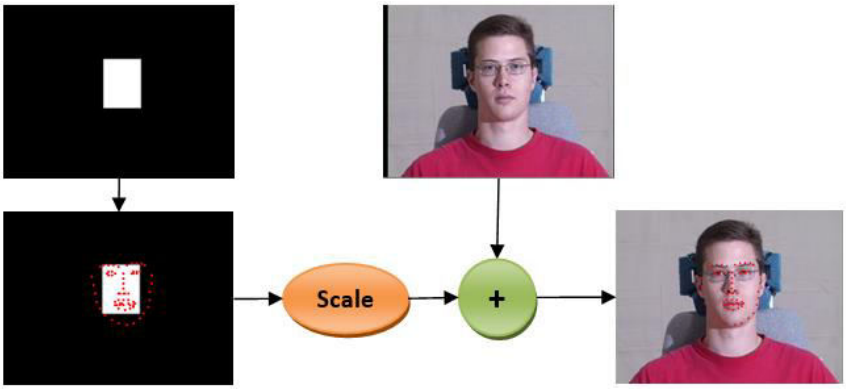


Figure 29 - TSM Algorithm Backtrack Procedure Results

The Backtrack stage results are temporary saved in a data structure called «Results Cache». This data structure has a user defined size and its default one is 10,000 set by the algorithm creators. If this data structure is fully filled with detection results the NMS process is called in order to release data by rejecting the fake results.

5.10. Non-Maximum Suppression (NMS)

Non-maximum suppression (NMS) [14] process is used for selecting high-scoring detections and skipping the ones that are significantly covered by previously selected detections. As described in chapter 5.9 the TSM algorithm produce many detection results while trying to detect a face within an image. As it is obvious poses that are near the same area of viewing angles produce scores with low contrast. For this reason the algorithm has to find out which detections refer to the same face within the image and which ones to different faces as an image can contain more faces. Detections that refer to the same face would have the same locality with low overlapping differences. The NMS method detects these overlaps and keeps only the highest score detection, rejecting the rest. This method also makes clear the pose estimation.

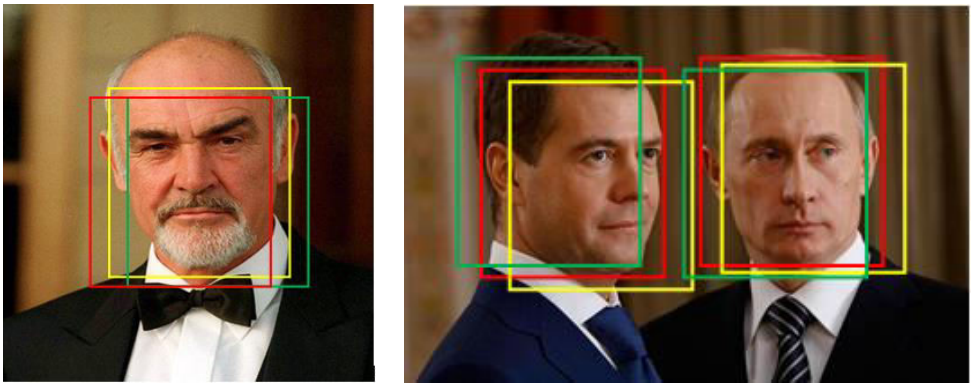
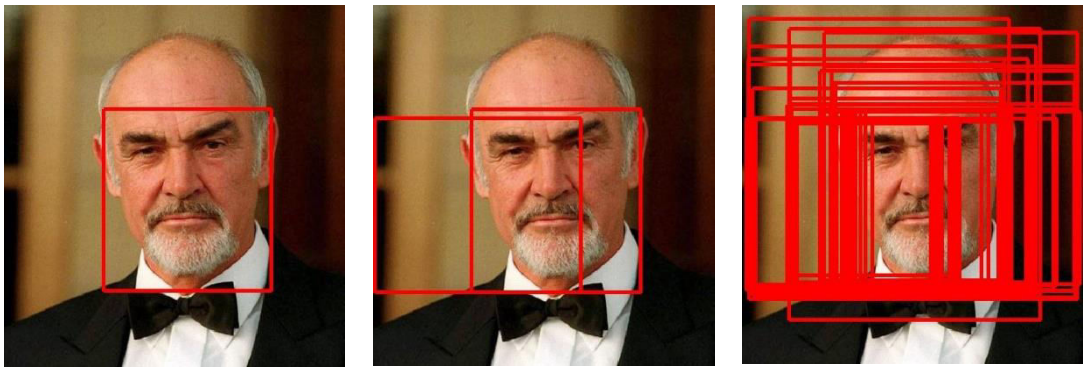
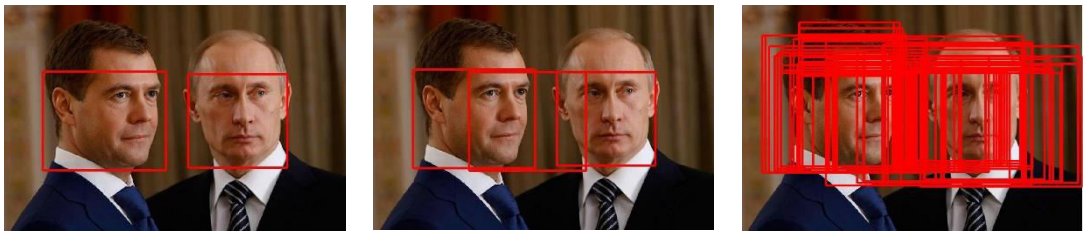


Figure 30 - TSM Algorithm One Face Multiple Detections Example

There is a parameter on this process called «Overlap». This parameter defines the percentage of one detection box area that overlap another one in order those two detection boxes to be considered as overlapping boxes. Two overlapping boxes refer to the same face. The score that follows each one is the parameter that creates the dominated one. The lower score boxes are discarded. The default value of the Overlap parameter is set to 0.3. This value must be also a product of multiple tests by the creators. Experiments in some different values come up with faulty results as shown in Figure 31.





0.3 0.7 1

Figure 31 - TSM Algorithm Overlap Parameter Impact

6. TSM Algorithm Implementation

In this chapter an implementation analysis of the TSM algorithm will be quoted. On the implementation architecture we divide the algorithm in three separate modules (Figure 32) according to their role and their dependencies. These three modules are the “Input”, the “Output” and the “Face Detector” one.

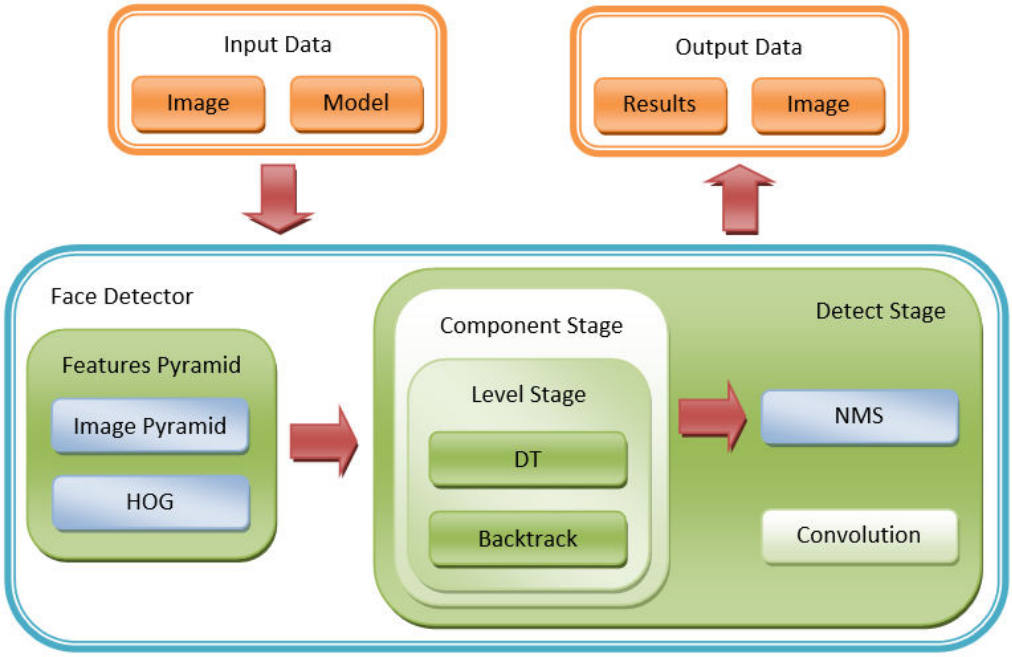


Figure 32 - TSM Algorithm Implementation Modules

The inputs module is where the input data of the algorithm come up. The algorithm gets two basic inputs, a 3D array structure containing the image data and the model data structure. The image array has to be a three channel array, one for each color. In our implementation we used the OpenCV [27] libraries in order to read image files and decode them in array data structures. We used the OpenCV library as it provides a variety of functions for reading image files, it is very popular to the computer vision society and it is free licensed. For the model data structure we used the XML data format almost for the same reasons. To read XML data format files we used the open source library rapidXML [33]. This stage is fully independent as it can be easy replaced by any custom module using other methods for providing the face detector algorithm with the input data it needs in the format we described above.

The output module is the one that gets the results from the face detector TSM algorithm and converts it in the format the user desires. In our implementation we offer three output types, projection in the computer screen (for PCs), exporting in image format file (JPEG) and in XML format file containing the algorithm's results data. For those three types we used the OpenCV and the rapidXML libraries as in the Inputs module. This module as the previous one is also fully independent and can be easily replaces by any custom implementation that a user can create.

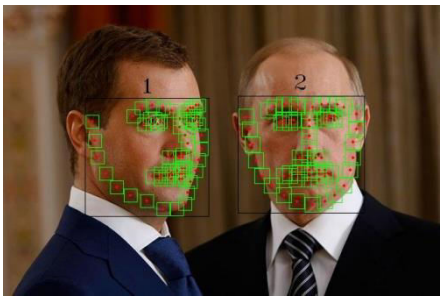


Figure 33 - TSM Algorithm Output Image

At last the "Face Detector" module is the one where the face detection process takes place. The "Face Detector" module consists of seven different stages. This stages are,

1. The Features Pyramid stage produces the pyramid of image descriptors (HOG). This stage was described in chapter 5.5 and it was separated from the rest stages as an independent stage because the next ones have to wait for its outputs in order to start their execution. None stage can start running if at least one features image is produced. It is a preparations stage that creates the data needed for the recognition process to start. The convolution stage needs it and it has to wait for it. Extensive description of this stage exists in chapter 6.6.

2. The Detect stage represents the main detection process and it is the algorithms real body. The process followed inside this stage is what makes the algorithm so special that the creators claim it as state-of-art algorithm. This stage contains all the rest stages of the algorithm.

3. The Components stage is the one where the detection procedure of a specific component takes place. In this stage, having a component as an input, the algorithm tries to detect it within all the levels of the Features Pyramid. This stage is executed one time for every component of the model.

4. The Level stage contains all the procedures needed to detect one component in one level feature image. This stage is executed once for every level of the Features Pyramid for every component of the algorithm's model.

5. The Convolution stage is the one where the convolution procedure takes place. The convolution process is described in chapter 5.6. The convolution stage is a very simple in complexity but with a heavy data processing one. It is better described by detail in chapter 6.9.

6. The Distance Transformation stage is using the distance transformation algorithm for creating detection results, as described in chapter 5.7. It represents the algorithms main detection process as it produces the data where the detection comes from. Detailed description of its implementation exists in chapter 6.11.

7. The Backtrack stage is the one where the landmark estimation takes place. It is a small but complex stage where the output data come from. It is the second pure representative of the detection algorithm. The Backtrack stage implementation is described in chapter 6.13.

In the next subchapters we represent the implementation architecture as provided by the creators in combined Matlab and C++ scripts. We firstly created a similar implementation in pure C\C++ script in order to profile the algorithm and check our implementation correctness. In the chapters following we exhibit a set of improvements we applied in our implementation in order to make it faster and less memory consuming.

6.1. Original Edition

The first version (version 1.1) of our implementation was a complete conversion of the creator's edition in Matlab to C++ in order to check the correctness of our implementation and be able to profile it and watch its attitude during its execution. The flow diagram of this implementation is shown in Figure 34.

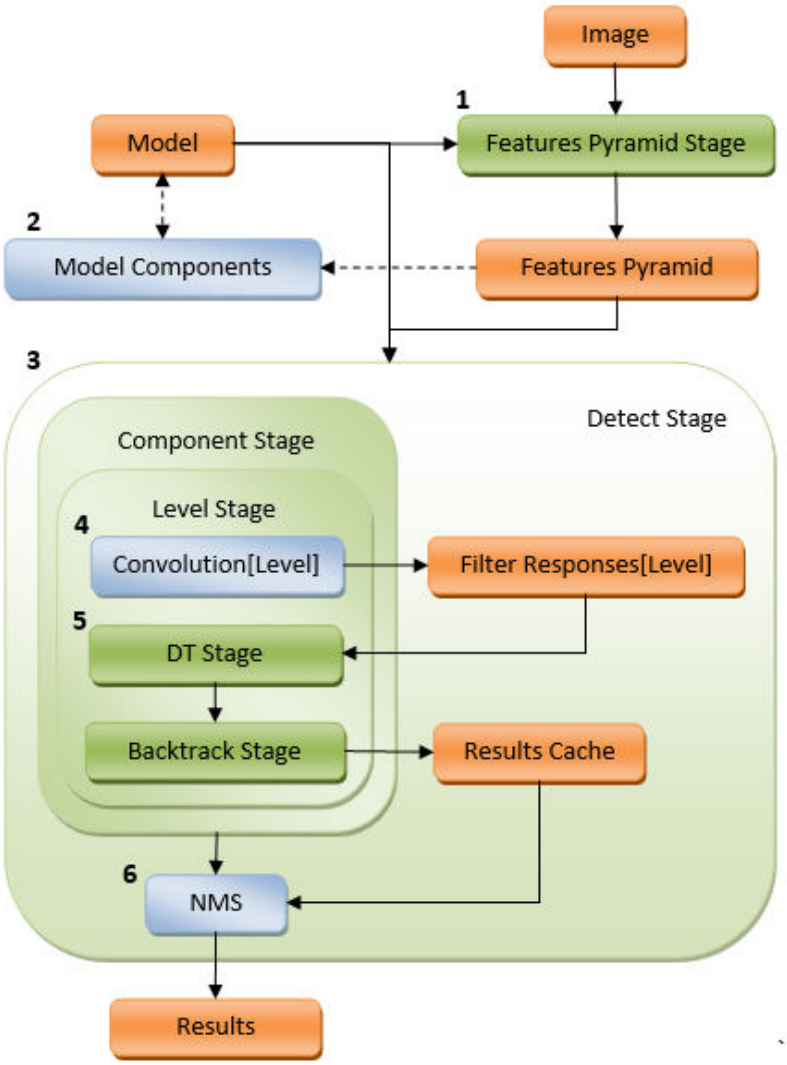


Figure 34 - TSM v1.1 Algorithm Implementation Diagram

As the number indexes indicates the algorithm flow follows the according flow,

1. **Features Pyramid Stage**: Having an image and a model available, the algorithm firstly produces the Features pyramid as explained in chapter 5.5.

2. **Model Components**: After the Features Pyramid is available the algorithm uses information about the pyramid's scales and arrays' sizes in order to update and calculate some model's parameters.

3. **Detect Stage**: After having the Features Pyramid calculated and the necessary data updated inside the model the detection process is ready to begin. The creators' edition begins the detection process trying to detect every component through the levels of the Features Pyramid. As seen in the graph two nested loops are used for this procedure separated as two different stages.

4. **Convolution Stage**: At the Level stage, where the algorithms tries to detect a component through all the levels of the Features Pyramid, the algorithm checks if the Filters' Responses are calculated for each level of the Features Pyramid. If they are not, then it call the Convolution stage to calculate them. This happens because in multi-scaled models some parts of the component may use Filter Response of other levels of the Features Pyramid. That is why are called multi-scaled models.

5. **Distance Transformation Stage**: At this moment the actual detection process starts for a specific level and component. The Filters Responses are necessary for this procedure. After the DT stage the Backtrack one follows and the detection results are stored in the Results Cache data structure.

6. **NMS**: At the end of the Detect stage when all the components have completed the detection procedure through all levels of the Features Pyramid the NMS procedure has to be applied in order to collect the right detection as explained in chapter 5.10.

The Feature Pyramid stage is the first process of the algorithms execution flow. This stage is using three main procedures implemented in C++ by the creators as shown in Figure 35.

1. **Resize**: The Resize procedure is the one that reduces the size of an image in a custom scale factor that gets as an argument. The scale factor value can be between 1 and 0.

2. **Reduce**: The Reduce procedure creates images in the half size of the source ones that gets as arguments. This procedure replaces the Resize one when the scale factor is 0.5 because it is a much faster one.

3. **HOG**: The HOG procedure converts an image into its Histogram of Oriented Gradients descriptors.

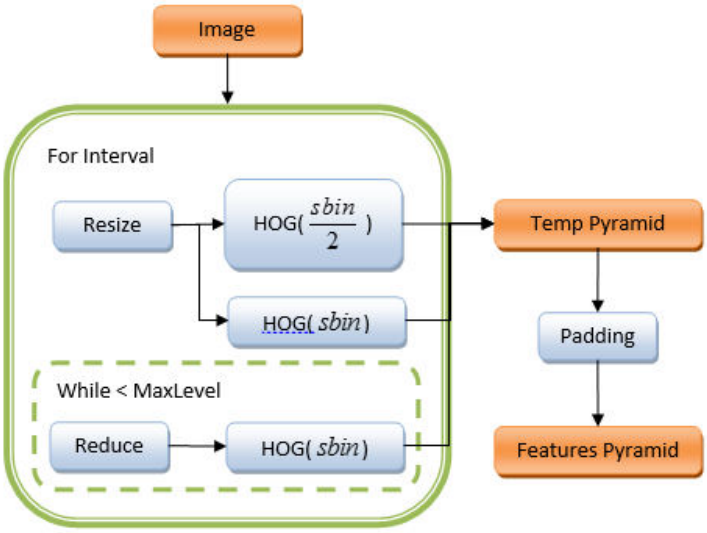


Figure 35 - TSM v1.1 Algorithm FP Stage Implementation Diagram

As seen in the Figure 35, the TSM algorithm uses the resized images to create half scaled ones with the Reduce procedure. By these images it gets the corresponding HOG images. After the algorithm completes the features pyramid (Temp pyramid), then it begins a padding procedure so the HOG images data arrays can be convolved later in the Convolution stage without any loss of information.

6.2. Profiler

Profiling the Face Detection TSM algorithm is not as simple as it may seem. This is because some parts of the algorithm are either image size or detection independent and some of them both.

In the profiling process there are four types of dependencies in the different parts of the algorithm.

- **Image size dependencies**: The image size dependencies come from the size of the image that is being processed by a part of the algorithm.

- **Pyramid dependencies**: This kind of dependencies come from the number of levels the features pyramid has. If the input image size is large, the number of features images come out the features pyramid process would be larger than a smaller size image.

- **Detection dependencies**: In chapter 5.7 the detection process is described of how the DT stage produces high-score values in the score array when face detection exists. By the results of the DT stage the execution of the Backtrack stage is depending as it is processing the detection results. If no detection results exists the Backtrack stage has no job to do. This is a detection dependency.

- **Model dependencies**: The models proposed for the TSM algorithm affect its performance as they contain different number of filters. Each filter is convolved with the features images of the pyramid and this is time and memory consuming procedure.

In the Table 6 below the dependencies table is presented showing the different stages and process dependencies as long as the time profiling of the algorithm.

Table 6 - TSM Algorithm Time Dependencies

Procedure	Size	Pyra	Detect	Model	Description
Features Pyramid Stage	Yes	Yes	No	No	See Resize, Reduce and HOG procedure
Resize Procedure	Yes	No	No	No	Procedure calls are the same as interval parameter (Pyramid) (Chapter 5.5) Larger image means more execution time (Size)
Reduce Procedure	Yes	Yes	No	No	Larger pyramid means more procedure calls (Pyramid) (Chapter 5.5) Larger image means more execution time (Size)
HOG Procedure	Yes	Yes	No	No	Larger pyramid means more procedure calls (Pyramid) Larger image means more execution time (Size)
Detect Stage	Yes	Yes	Yes	Yes	See Convolution, DT and Backtrack stage. See NMS procedure
Conv. Stage	Yes	Yes	No	Yes	See Convolution procedure
Convolution Procedure	Yes	Yes	No	Yes	Larger image means more execution time (Size) Larger pyramid means more procedure calls (Pyramid) More filters means more procedure calls (Model)
Component Stage	Yes	Yes	Yes	No	See DT and Backtrack stages
Level Stage	Yes	Yes	Yes	No	See DT and Backtrack stages
DT Stage	Yes	Yes	No	No	See DT procedure
DT Procedure	Yes	Yes	No	No	Larger image means more execution time (Size) Larger pyramid means more procedure calls (Pyramid)
Backtrack Stage	Yes	Yes	Yes	No	See Find and Backtrack procedures
Find Procedure	Yes	Yes	No	No	Larger image means more execution time (Size)

					Larger pyramid means more procedure calls (Pyramid)
Backtrack Procedure	No	No	Yes	No	More high-score values detected more execution time. More detections means more data to process (Chapter 5.9) More detections means more procedure calls (Chapter 5.9)
NMS Procedure	No	No	Yes	No	More high values detected cause easier the results cache to full meaning more procedure calls (Chapter 5.10aaaaaa) More high values means more execution time

As the execution time of the algorithm may varies due to hardware resources and the operating system workload, the time profiling of the algorithm is presented in percentages according to its total execution time. In chapter 10 a set of measurements for different hardware resources is appose.

As long as the time profiling of the algorithm, the profiling process had to be done using a variety of image's sizes that would also produce high detection results. This way all these three profiling dependencies are calculated inside the profiling process. In our profiling process we used images of the following sizes shown in Table 7.

Table 7 - TSM Algorithm Profiling Images

Sample Images	Pixels	Pixels (Mpx)	FP Levels	Max Faces
320x240	76,800	0.1 Mpx	18	8
640x480	307,200	0.3 Mpx	23	31
800x600	480,000	0.5 Mpx	25	48
1024x768	786,432	0.8 Mpx	27	79
1280x960	1,228,800	1.2 Mpx	28	123
1600x1200	1,920,000	1.9 Mpx	30	192

In the Table 8 below the dependencies of the algorithm parts as long as their memory impact are shown.

Table 8 - TSM Algorithm Memory Dependencies

Procedure	Size	Pyra	Detect	Model	Description
Features	Yes	Yes	No	No	See Resize, Reduce and HOG procedure

Pyramid Stage					
Resize Procedure	Yes	No	No	No	Larger images produce larger scaled images (Size)
Reduce Procedure	Yes	Yes	No	No	Larger images produce larger reduced images (Size) Larger pyramid means more Reduce procedure calls (Pyramid)
HOG Procedure	Yes	Yes	No	No	Larger images produce larger HOG images (Size) Larger pyramid means more HOG procedure calls (Pyramid)
Detect Stage	Yes	Yes	Yes	Yes	See Convolution, DT and Backtrack stage. See NMS procedure
Convolution Stage	Yes	Yes	No	Yes	See Convolution procedure
Convolution Procedure	Yes	Yes	No	Yes	Larger images produce larger filters responses images (Size) Larger pyramid means more procedure calls (Pyramid) More filters means more procedure calls (Model)
Component Stage	Yes	Yes	Yes	No	See DT and Backtrack stages
Level Stage	Yes	Yes	Yes	No	See DT and Backtrack stages
DT Stage	Yes	Yes	No	No	See DT procedure
DT Procedure	Yes	Yes	No	No	Larger images produce larger DT images. Larger pyramid means more DT procedure calls
Backtrack Stage	Yes	Yes	Yes	No	See Find and Backtrack procedures
Find Procedure	No	Yes	Yes	No	Larger pyramid means more Find procedure calls (Pyramid dependence) More high-score values detected more find results (Detections dependence) (Chapter 5.8)
Backtrack Procedure	No	No	Yes	No	More high-score values detected produce more backtrack results. More detections means more Backtrack procedure calls
NMS Procedure	No	No	Yes	No	More high-score values detected produce detection results and results cache filling. More results cache fillings mean more NMS procedure calls

At last these dependencies affect the memory needed for the basic TSM algorithm data structures used for the detection procedure. In the Table 9 above this dependencies are presented.

Table 9 - TSM Algorithm Data Dependencies					
Procedure	**Size**	**Pyra**	**Detect**	**Model**	**Description**
Features Pyramid	Yes	Yes	No	No	Larger images produce larger sub-scaled images and features images (Size) Larger images produce greater levels features pyramid (Pyramid)
Filters Responses	Yes	No	No	Yes	Larger features images produce larger Filters Responses (Size) More filters produce more Filter Responses (Model)
DT Scores	Yes	Yes	No	No	Larger images produce larger DT Scores (Size) Larger pyramid produce more DT scores arrays (Pyramid)
Results Cache	No	No	Yes	No	More detections produce more detection results

As long as the memory profiling process a virtual profiler was created in order to produce the maximum memory consumption results assuming the worst case scenarios. For maximum memory consumption profiling, the profiler reacts as the detection process is achieving full detection results on all levels of the features pyramid on every pose tree. This way there is no case that can escape. This is the worst case of maximum memory consumption. This scenario is impossible to happen in real world but is accurate to predict the possible maximum memory consumption as it is used for different sized images and assuming the worst detection dependencies scenario. The first three dependencies (except Model) are calculated for the worst case by the profiler.

On the other hand in total memory consumption profiling the virtual profiler assumes that the image is fully filled with faces but this faces cannot produce full detection results in every component at all levels as this scenario is out of sense and it would produce memory profiling results that would be misguided. Using different sizes images is the easy way to beat the image size and pyramid levels dependencies, but as long as the detection ones using the maximum consumption profiling scenarios it produces huge amounts of memory consumption that leads to misunderstandings and it is far away from the real life results.

The only stage that is actually detection dependent is the Backtrack one. By this stage is also depended the NMS procedure calls. The Backtrack stage is the one that checks inside the score array, which comes from the Distance transformation stage, for high-score values and matches these values with the corresponding model tree landmarks for landmark and pose estimation.

High-score values means face detection. If the Find procedure does not find high-score values the rest of the Backtrack stage is not executed. The whole Backtrack stage is difficult to be profiled as it is fully depended by the detection results. The Find procedure though is the only part of the backtrack stage that is always executed.

When profiling the algorithm for maximum memory consumption we assume that the Backtrack stage is getting the maximum high-scores values from the Distance Transformation stage. It is like getting an image full of high-scored values. On the other hand when we profile the algorithm for total memory consumption this strategy gives as a huge amount of memory consumption that is very far away from the real life results and it would lead to incorrect conclusions. For that reason a series of tests were made in order to create a memory profiling model that could create the most secure and close to real life profiling.

By testing the algorithm in different scales of faces, it was discovered that it is able to detect faces larger than 100 pixels high when using the 99 filters model and larger than 50 pixels when the 146 filters one. With a width of the same size in pixels, a face's area is about 1000 and 250 pixels. This way it is easy to predict the maximum number of faces can be presented within an image according to its size using the functions *(1*) and *(2*).

$$Faces_{\max 99} = \left\lfloor \frac{image.width \times image.height}{10000} \right\rfloor \quad (1)$$

$$Faces_{\max 146} = \left\lfloor \frac{image.width \times image.height}{2500} \right\rfloor \quad (2)$$

In the Distance Transformation stage not all face detections produce the same high-score values. A clear image of a face inside a laboratory environment produces more high-score values than a face within an into-the-wild environment. In addition a face of zero degrees angle produces more high-score values than another one with more degrees angle. Knowing that a face's area is 1000 pixels within the image and also knowing that a feature image has about 16 times less pixels than its original (4 times smaller) it is sensible that the maximum high-score values that a face can produce in the Distance Transformation stage is about 625 values. This gives also a maximum face approximate function, the function *(3*).

$$Faces_{\max} = \left\lfloor \frac{HOG.width \times HOG.height}{600} \right\rfloor \quad (3)$$

Another parameter that takes matter in the prediction of the Backtrack stage attitude according to the memory profiling is in how many levels a face within an image can create high-score values. Again, faces with angles and into-the-wild images produce fewer high-score values than faces with zero degrees angle and captured in laboratory environment.

By testing the algorithm using both detectable images from laboratories and into the wild images a close prediction to the real maximum memory consumption can be exclaimed. The results showed that a face can be detected about at the 12% of the features pyramid levels succeeding an average of about 80 high-score values per component at each level. These results are shown in Table 10 below.

Table 10 - Find Procedure Profiling Results						
	$\frac{Levels_{with-High-Scores}}{Levels_{Features\_Pyramid}}\%$			$\frac{Pixels_{with-High-Score}}{Find_{with-High-Score}}$		
Samples	**Max**	**Average**	**Min**	**Max**	**Average**	**Min**
	99 Filters Model					
Top 10%	18.9	14.3	12.1	611	169	1
Top 20%	28.6	17.2	9.18	611	128	1
Top 50%	28.6	14.8	7.37	611	103	1
All (100%)	28.6	11.6	0.31	611	79	1
	146 Filters Model					
Top 10%	21.5	16.1	12.9	343	116	1
Top 25%	24.0	16.8	10.7	343	91	1
Top 50%	24.0	15.7	7.21	343	70	1
All (100%)	24.0	11.8	0.32	343	53	1

As seen in the Table 10 above the two Models offered for the algorithm creates much different results. It is obvious that using more filters the algorithm is more accurate at its detection producing less high-scores for the same or even better results. This is because every filter used in the 146 filters Model is better trained and more accurate on detecting human face landmarks. As also seen in the Table 10 the clearest the images are more concentrated are the high-score values inside the Feature Pyramid levels. In the results table, the maximum number of high-score values reached by an image is 611 values, almost the same with the theoretical value calculated in the previous paragraph. This shows that the number of high-score values a face produces is much smaller than it real size in pixels.

According to these measurements two basic functions were created in order to predict the number of high-value pixels result after the DT stage procedure. For creating these two functions and for prediction safety reasons the top 50% of the samples were used. These two functions are,

99 Filters Model

$$Levels_{High-Scored} = Round\left(0.15 \cdot levels\right) \quad (4)$$

$$HighScores_{Total} = 100 \times Components \times Levels_{High-Scored} \quad (5)$$

146 Filters Model

$$Levels_{High-Scored} = Round\left(0.15 \cdot levels\right) \quad (6)$$

$$HighScores_{Total} = 70 \times Components \times Levels_{High-Scored} \quad (7)$$

The first function ((4) & (6)) gives the levels of the features pyramid that high-scored values appears cause of the faces within the image. The levels start counting always from the top in order to profile the Backtrack stage with the hardest amount of data even if it is image size independent. The second function ((5) & (7)) calculates the total number of high-score values detected by the find function in the whole algorithms execution. In Table 11 various cases results are presented for the 99 Filters Model.

Table 11 - High-Score Pixels Profiler Results

Image Size		320x240	640x480	800x600	1024x768	1280x960
$Levels_{Features\_Pyramid}$		18	23	25	27	28
$Levels_{High-Scores}$		3	4	4	5	5
High-Scored Pixels	Faces	High-Score Pixels				
	1	3,900	5,200	5,200	6,500	6,500
	2	7,800	10,400	10,400	13,000	13,000
	3	11,700	15,600	15,600	19,500	19,500
	4	15,600	20,800	20,800	26,000	26,000
	5	19,500	26,000	26,000	32,500	32,500
	6	23,400	31,200	31,200	39,000	39,000

One of the parameters affecting the maximum memory consumption of the algorithms is the Results cache memory. This data structure keeps the data returned from the Backtrack procedure in addition to some more information until the NMS process select the correct ones. This cache memory is defined to 10,000 detection results by the creators but it is easily changeable. For that reason the results cache memory is not included in the max memory consumption profiling as it affects the distribution statistics. In the table below the results cache max memory usage is shown according to its size.

Table 12 - Results Cache Sizes

Cache Size (Detections)	Max Memory (Mbytes)
10,000	11,24
8,000	8,99
6,000	6,74
4,000	4,50
2,000	2,25

6.3. *Original Edition Profiling*

After the implementation of the TSM algorithm's version 1.1 a profiling process took place in order to watch the algorithms' attitude during its execution. In the profiling process we watch only the «Face Detector» module as the rest (Inputs, Outputs) are customize according to every specific application and the main function.

6.3.1. Time Profile

In Table 13 the percentage of CPU holding time of each stage and procedure for different image sizes is shown. A graphic representation of these results is presented in Diagram 1. As it is apparent the main CPU time consumer is the Convolution stage. The second procedure that keeps the CPU busy is the Distance Transformation one.

Table 13 - TSM v1.1 Execution Time Distribution (%)						
	320x240	**640x480**	**800x600**	**1024x768**	**1200x960**	**Average**
Levels	18	23	25	27	28	
FP Stage	3.86	4.47	4.50	5.02	4.66	4.50
Conv. Stage	64.3	65.9	66.3	66.3	66.8	65.9
DT Stage	31.2	29.2	28.9	28.4	28.4	29.2
Backtrack Stage	0.35	0.38	0.30	0.21	0.14	0.28
Others	0.30	0.09	0.08	0.06	0.04	0.11

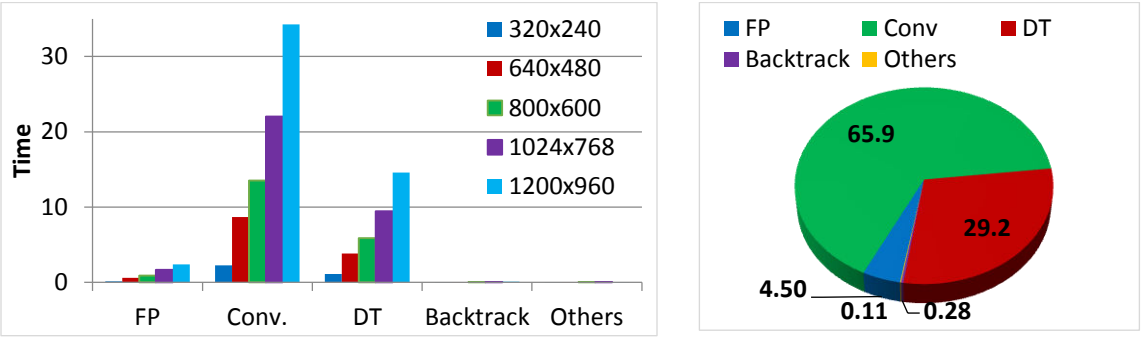


Diagram 1 - TSM v1.1 Algorithm Execution Time Distribution per Stage

As is shown in Diagram 1 the execution time of each stage is almost stable in ratio to the algorithms total execution time despite to the processed image size.

In Diagram 2 the algorithms' execution timeline for a 640x480 size image is shown. What is conspicuous is that all the convolution processing takes place at the first run of the Component stage executed for the first component of the model. This is a useful note concerning the algorithm's execution flow for further improvements exposed in the following chapters.

Diagram 2 - TSM v1.1 Algorithm Execution Timeline

In the Diagram 3 below the time consumption incremental trend is shown. As seen the Convolution, DT and FP stages execution time is normally increased as the image size does. One the other hand the Backtrack stage has reversal trend. This is because the Backtrack stage consists from image size independent parts. As referred in chapter 5.9 the Backtrack stage is mainly detection dependent and that is why it is not following the same trend as the rest stages of the algorithm that are mainly image size dependent.

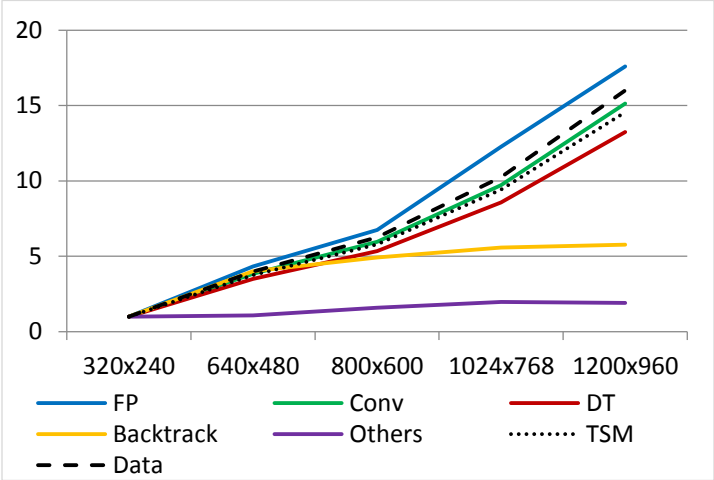


Diagram 3 - TSM v1.1 Stages Execution Time Growth Trend per Image

6.3.2. Memory

A second type of profiling applied in the algorithm is the memory one. The memory consumption of the algorithm cannot be profiled accurate as the number of detection within the image affects extensively the memory consumption. For that reason we used for that process a memory profiling simulator that takes as parameters the worst cases of memory consumption so that the maximum memory consumption can be accurate forecasted as mentioned in chapter 6.2.

By profiling the algorithm memory usage in a variety of different size images we got measurements about the memory usage of the algorithm for the features pyramid's arrays, the filters responses' arrays and the whole amount of memory it request from the operating system. All these measurements are shown in Table 14 and in Diagram 4.

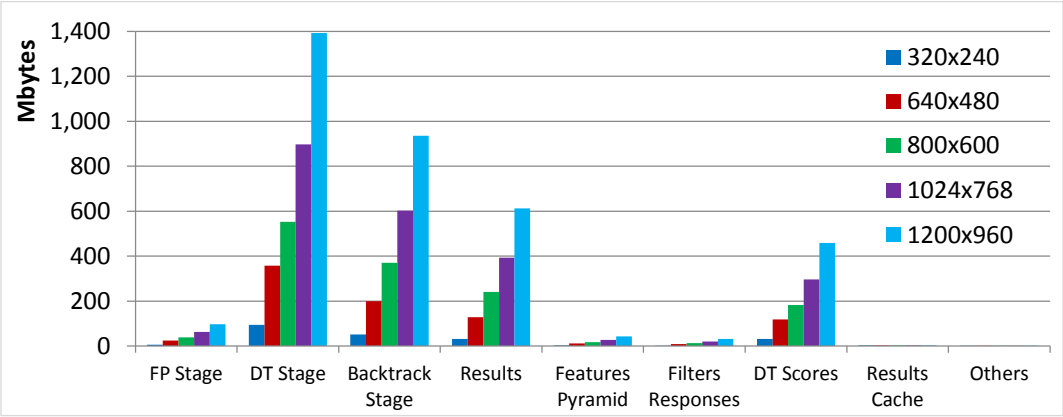


Diagram 4 - TSM v1.1 Memory Consumption Distribution

Table 14 - TSM v1.1 Memory Consumption Distribution (%)							
	Image Size	**320x240**	**640x480**	**800x600**	**1024x768**	**1200x960**	**Average**
	Total Usage	1.8 Gb	6.8 Gb	11.3 Gb	18.4 Gb	28.6 Gb	
Stages	**FP stage**	2.71	2.86	2.69	2.71	2.73	2.74
	Conv. stage	0.00	0.00	0.00	0.00	0.00	0.00
	DT stage	42.2	42.0	39.0	39.0	38.9	40.2
	Back. stage	23.4	23.5	26.1	26.1	26.2	25.1
Data	**F. Pyramid**	1.34	1.29	1.20	1.19	1.18	1.24
	F. Responses	0.98	0.97	0.91	0.90	0.90	0.93
	DT Scores	13.9	13.8	12.9	12.8	12.8	13.3
	Results	14.1	15.1	17.0	17.1	17.1	16.1
	Others	1.39	0.38	0.24	0.16	0.11	0.46

As seen in the graph the DT stage is the most memory consumer of the algorithm creating suspicious for memory leakages and possibilities of memory usage improvements. The second greater memory consumer of the algorithm is the Backtrack stage with the detection results in the third position. As seen the data structures needed for the detection (Features Pyramid, Filters Responses, DT Scores, Results Cache) use a small amount of memory in relation to the whole algorithm memory consumption. The DT Scores arrays are those that use the most memory unlike the rest ones.

In the Diagram 5 below the incremental trend according to the image process size is presented. As seen all the stages memory consumption is normally increased as the image size does. The only stage that stay still is the Convolution stage that uses zero temporary memory for its procedure and the Results Cache memory that is a stable, image size, independent data structure.

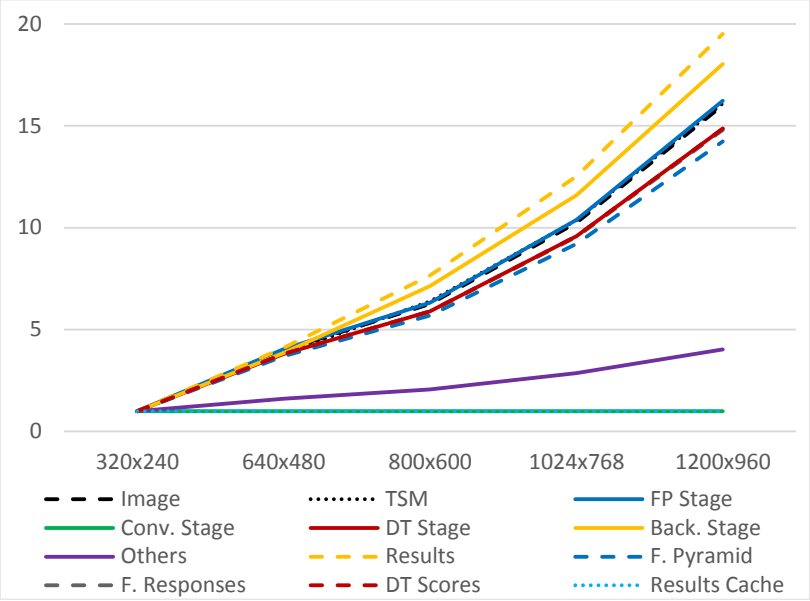


Diagram 5 - TSM v1.1 Memory Consumption Growth Trend per Image

6.3.3. Max Memory

A third type of profiling is the maximum memory one. This is a very important measurement as it reveals the maximum memory needed for the algorithm to be able to be executed in the hardware. Unlikely, the total memory consumption profiler that is used only for checking the algorithm attitude during its execution, the max memory profiler is critically used for checking the hardware resources needed for the algorithm to be executed. In Table 15 below the distribution of the maximum memory consumption of the algorithm is shown. In this table the Results Cache memory is not contained as it is volatile and user determined as explained in chapter 6.2. Despite that, in the Results Cache table line, the incremental caused to the maximum memory when the default size Result Cache is used is filled. The table's contents are graphically shown in the Diagram 6.

Table 15 - TSM v1.1 Max Memory Consumption Distribution (%)						
Image Size	**320x240**	**640x480**	**800x600**	**1024x768**	**1200x960**	**Average**
Pyramid Levels	18	23	25	27	28	
Max Usage	70 Mb	287 Mb	409 Mb	664 Mb	1,030 Mb	
FP Stage	0.00	0.00	0.00	0.00	0.00	0.00
Conv. Stage	0.00	0.00	0.00	0.00	0.00	0.00
DT Stage	0.00	0.00	0.00	0.00	0.00	0.00
Backtrack Stage	31.6	30.0	31.1	31.9	32.4	31.4
Features Pyramid	33.9	30.7	31.4	31.9	32.2	32.0
Filters Responses	24.8	23.1	23.8	24.3	24.6	24.1
DT Scores	7.81	7.43	7.68	7.88	8.00	7.76

Others	1.88	0.94	0.91	0.84	0.80	1.08
Results Cache (default)	+32.0	+7.82	+5.49	+3.39	+2.18	+10.2

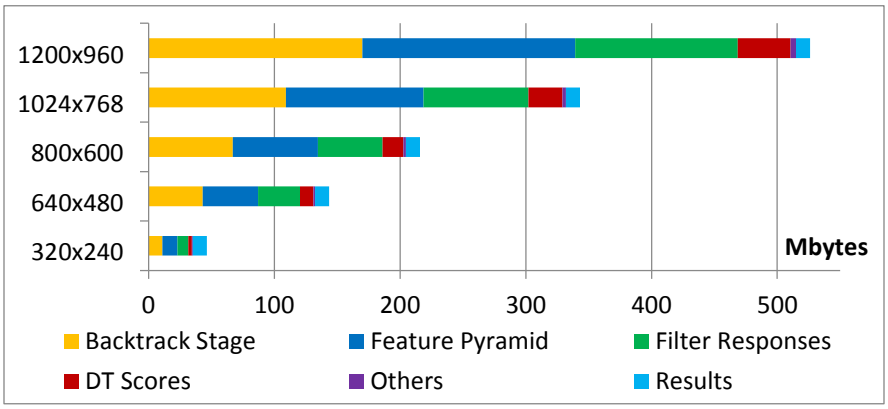


Diagram 6 - TSM v1.1 Maximum Memory Distribution per Image

Looking at the maximum memory distribution graph (Diagram 6) it is visible that the greatest parts of the maximum memory consumption are hold by the Backtrack stage temporary memory, the features pyramid data structure and the filters responses one. What is very important is that almost the one third of the maximum memory consists of temporary memory unlike the rest memory that consists of useful data structures. Another point is that the Results Cache data structure affects the maximum memory consumption of the algorithm more when the image size is used in getting smaller. As shown in the Table 15, above the increment on the maximum memory consumption of the algorithm when the default Result cache size is used reaches the 32% on a 320x240 image while this increment is only 2.2% for a 1200x960 one. This makes sensible that the Results Cache size should dynamically change according to the size of the processing image.

In the Diagram 7 below the maximum memory distribution is incremental trend is presented. All the participants of the maximum memory consumption are increasing normally as the image size is increasing except of the Result cache that remains stable independent the image size.

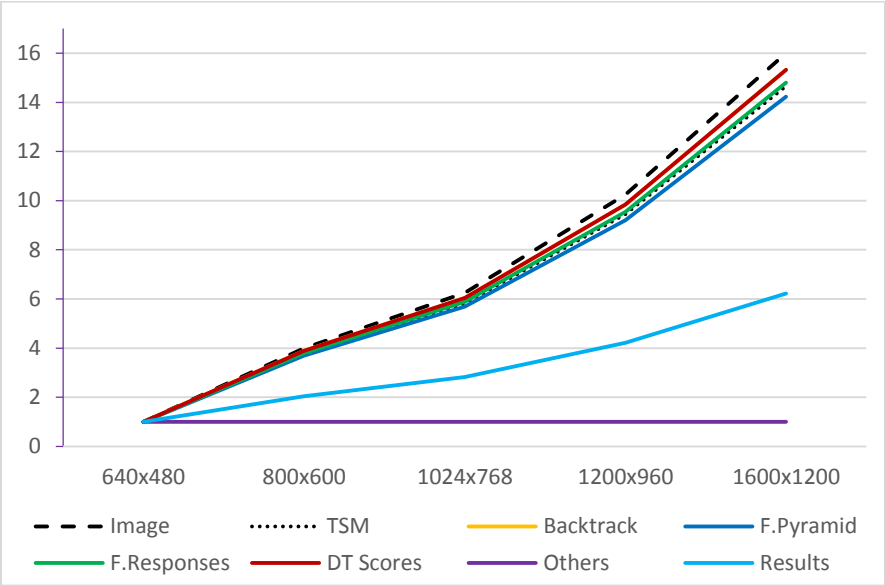


Diagram 7 - TSM v1.1 Maximum Memory Consumption Trend per Image

In the Diagram 8 a detailed memory profile of the algorithm is presented. In dark vertical line the Components loop is defined as described is chapter 6.1 (Figure 34). What is suspicious for memory leakage is the fact that the Features Pyramid data structure seems to consume more memory than the Filters Responses ones. For every level of the features Pyramid the algorithm uses an XxYx32 image and for its response to all filter a (X-4)x(Y-4)x99 array. This means that the Features Responses should use more memory than the features images. The memory profile graph above betrays a series of parental remains of the Parts Based Detector algorithm explained in next subchapter (chapter 6.4).

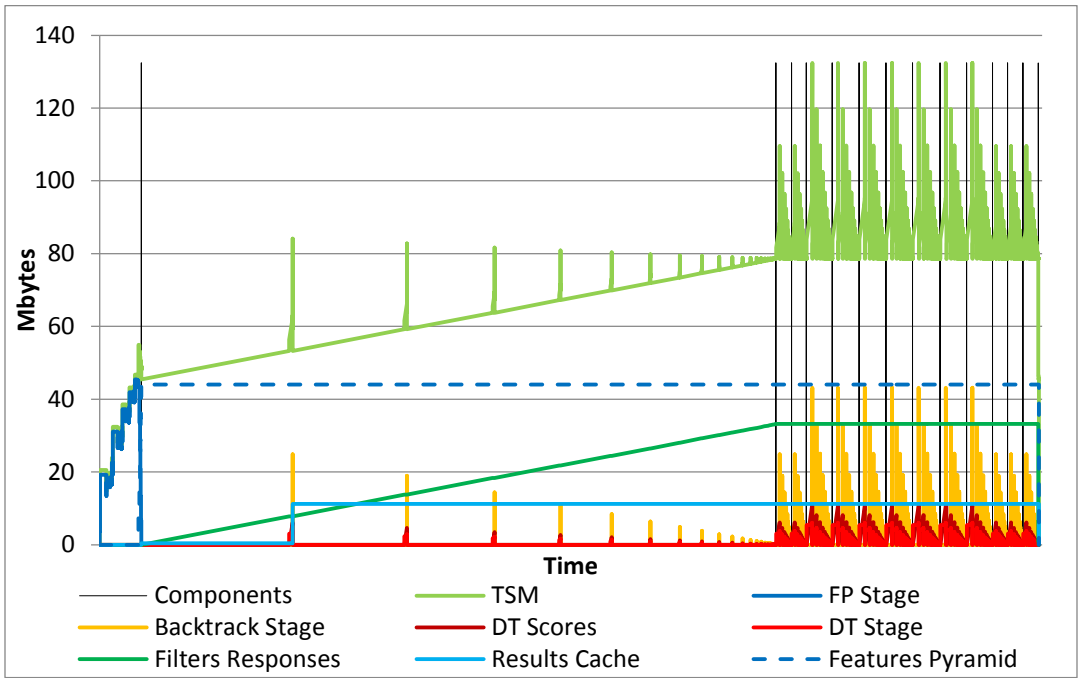


Diagram 8 - TSM v1.1 Algorithm Memory Profile

6.4. DPBD Algorithm Remains

Before starting analyzing the algorithm from the top to the bottom and proceed to changes in details, a series of small but crucial changes had to be made as they affect the whole algorithms execution and it would be better to referred before the in deep analysis.

6.4.1. Removing the Model Components Process

In the index 2 of the Figure 34 (Chapter 6.1) the algorithm uses some information produced in the features pyramid and updates some of the parameters of the model. This effect of the feature pyramid over the model comes from the multi-scale models of the DPBM algorithm. On the TSM algorithm's one-scale model this affect is disappeared and the "Model Component" procedure can take place before the features pyramid process and even omitted. As the TSM face detection model's parameters are independent from the features pyramid's information the "Model Component" procedure can take place once and its effect over it can be saved permanently in the model data structure file.

As long as its contribution to time and memory saving, this change has no impact as it is a very fast and memory costless procedure. The removal of this process is not a crucial one and can be let as is, although for informational reasons it had to be referred. It is important though to refer that removing this procedure from the algorithm creates the need of creating a new model data structure with the data that the model components procedure calculates inside.

6.4.2. Convolution Process

In the index 4 of the Figure 34 (Chapter 6.1), the algorithm calls the convolution procedure to calculate the filters responses over a specific level of the feature pyramid. These responses are saved and used for all the parts asking for their response to this specific level of the feature pyramid. At this point one more remain of the DPBM algorithm exists. The model uses a scale parameter for every part because of the multi-scale type of the DPBM algorithm where each parts of the model may needs different level response. On the TSM face detector algorithm all the parts of the model use the same level responses. This difference allow as to change the location inside the algorithm where the convolution process can take place and use less memory at its execution as shown in Figure 36. The effect of a change like this is described in chapter 6.17 as it changes the whole algorithms' execution flow.

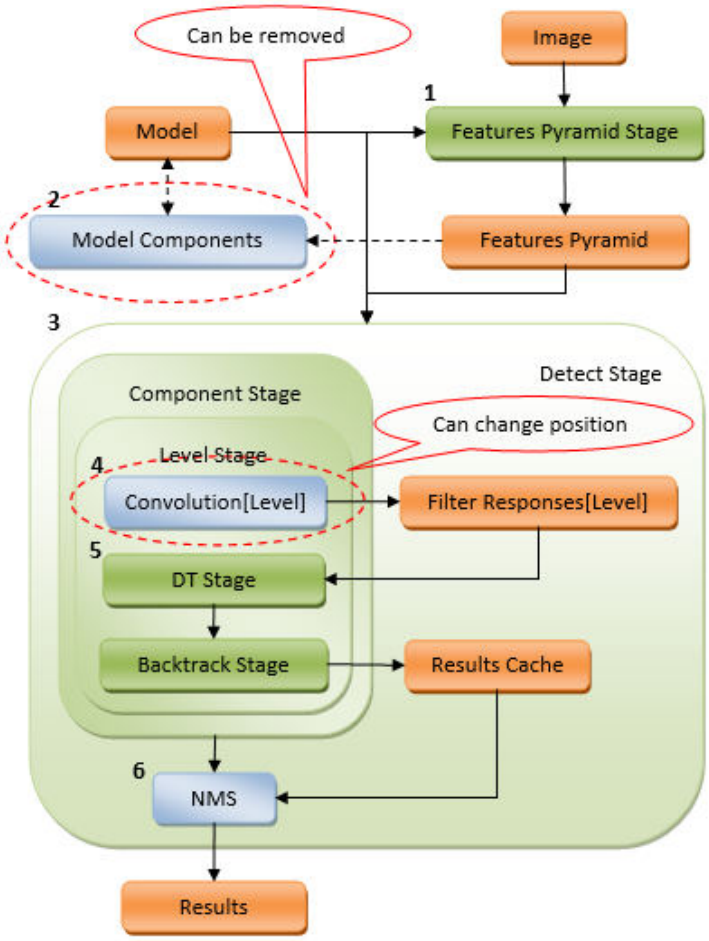


Figure 36 - TSM v1.2 Algorithm Execution Flow Changes

6.4.3. Root Filter Interval Set

As seen in Figure 35 in chapter 6.1 the memory allocated for the features pyramid is larger than the one that used for the Filter Responses. This reveals another remain of the parental DPBD algorithm. As shown in Figure 37, inside the red circle the algorithm creates a series of features images using the half sbin parameter value. This action creates an interval set at the top of the features pyramid that is two times smaller than its original image as referred in chapter 5.5. The reason the creators proceed at this implementation is probably because they need features images twice larger than the ones the parts need for all levels for the root part of every model as referred in chapter 4.1. In the TSM face detection algorithm the root part is similar to all the others and these interval set is actually not ever used. This is the reason why the features pyramid structure uses more memory, as its top interval set is never used in the convolution process and does not create filters responses. As a result this interval set can be removed from the features pyramid.

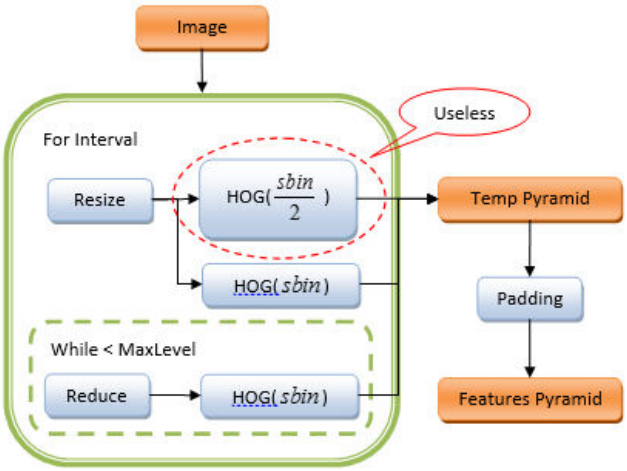


Figure 37 - Features Pyramid Stage Changes (TSM v1.2)

By removing this top interval set from the Features Pyramid the results are got in the algorithm are those shown in Table 16 below. As seen in this table the Features Pyramid stage time consumption is reduced at the half of it. This is because the interval set of features images removed is the top one which means the greatest images set. As also seen in the same table the algorithm's maximum memory consumption is significantly reduced by just removing an interval set of features images. This is an indication of how great impact has the images size in the maximum memory consumption of the algorithm.

Table 16 - Features Pyramid Extra Interval Set Removal Effect (TSM v1.1) (%)

	320x240	640x480	800x600	1024x768	1280x960	Average
Levels	18	23	25	27	28	
Time						
TSM	-2.01	-2.21	-2.23	-2.67	-2.06	-2.23
FP Stage	-51.4	-50.7	-48.8	-54.2	-49.2	-50.8
Memory Usage						
TSM	-2.30	-2.36	-2.20	-2.22	-2.23	2.26
FP Stage	-49.5	-49.4	-49.4	-49.4	-49.4	-49.4

Max Memory						
TSM	-24.2	-30.2	-24.3	-24.3	-24.3	-25.5
Features Pyramid	-71.5	-73.1	-73.4	-73.7	-74.0	-73.1

6.4.4. Double to Float

At last another global change in the algorithm is the conversion of it in order to use float data types instead of the double ones. This small conversion reduces all the memory consumption to its half as the float data type is using 4 bytes instead of 8 ones. The algorithm's accuracy in not influenced at all and its execution time is reduced as shown in Table 17 below.

Table 17 - TSM v1.1 Double to Float Effect (%)						
	320x240	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
Time	-6.13	-7.58	-7.69	-7.08	-7.62	-7.22
Memory	-50.0	-50.0	-50.0	-50.0	-50.0	-50.0
Max Memory	-50.0	-50.0	-50.0	-50.0	-50.0	-50.0

A similar attempt of conversion the algorithm to run using normalized integer values was tried. The results were negative in this attempt as the algorithm lost a bit of its accuracy especially during the landmark localization and the time consumption was worst compared to the float version due to the continuous normalizations needed.

By using the float data type instead of the double one and removing the extra features pyramid interval set the algorithm is now consider as an extended version of the original in order to individualize it from the original version 1.1. This version is called the 1.2 version of the algorithm. The differences are not much and not important but from this point every comparison with the primary version would be a reference to the version 1.2.

6.5. TSM Original Version 1.2

After removing the remains of the DPBM algorithm from the creators' edition (1.1) as referred in the previous chapter 6.4, the algorithm moves to the new 1.2 version. For this version is required to present the new profiling tables and graphs as they are going to be used as comparison data for the changes that will referred in the following chapters. In the Table 18 below the time table is presented.

Table 18 - TSM v1.2 Execution Time Distribution (%)						
	320x240	**640x480**	**800x600**	**1024x768**	**1200x960**	**Average**
Levels	18	23	25	27	28	
FP Stage	1.91	2.25	2.36	2.37	2.42	2.26

Conv. Stage	65.6	67.4	67.8	68.1	68.1	67.4
DT Stage	31.9	29.9	29.5	29.3	29.3	30.0
Backtrack Stage	0.37	0.39	0.31	0.22	0.15	0.29
Others	0.30	0.09	0.08	0.06	0.04	0.11

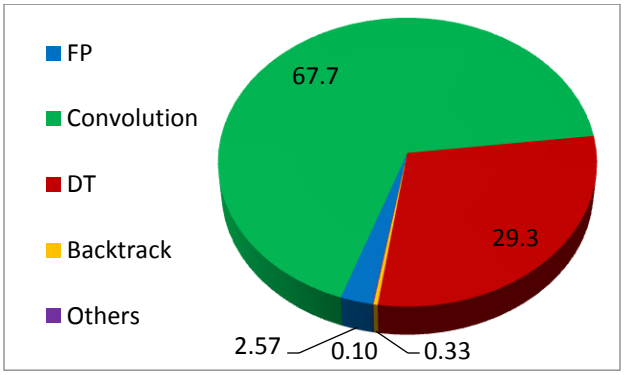


Diagram 9 - TSM v1.2 Execution Time Distribution per Stage

As far as the memory consumption, the change of using double type data to float ones reduced the algorithm memory usage in the physical memory to the half but this is not a real change to the algorithms' structure. On the other hand removing the first interval set of the Features Pyramid has reduced the size of the Features Pyramid data structure and the algorithm maximum memory consumption despite the data type used (float, double). The new memory consumption tables are presented below.

Table 19 - TSM v1.2 Memory Consumption Distribution (%)							
		320x240	**640x480**	**800x600**	**1024x768**	**1200x960**	**Average**
	Total Usage	0.86 Gb	3.3 Gb	5.5 Gb	9.0 Gb	14.0 Gb	
	vs 1.1	-2.30	-2.36	-2.20	-2.22	-2.23	-2.26
Stages	**FP stage**	1.40	1.48	1.39	1.40	1.41	1.42
	Conv. stage	0.00	0.00	0.00	0.00	0.00	0.00
	DT stage	43.2	43.0	39.9	39.9	39.8	41.2
	Back. stage	23.9	24.1	26.7	26.7	26.8	25.6
	Results	14.4	15.5	17.3	17.4	17.5	16.4
Data	**F. Pyramid**	0.39	0.36	0.33	0.32	0.31	0.34
	F. Responses	1.01	1.00	0.93	0.93	0.92	0.96
	DT Scores	14.2	14.2	13.2	13.1	13.1	13.6
	Results Cache	1.30	0.34	0.20	0.12	0.08	0.41
	Others	0.13	0.06	0.04	0.04	0.03	0.06

As far as the maximum memory consumption of the algorithm that is a more critical indicator affecting the algorithm execution ability over the hardware resources, the new maximum memory tables are below,

Table 20 - TSM v1.2 Max Memory Consumption Distribution (%)						
	320x240	**640x480**	**800x600**	**1024x768**	**1200x960**	**Average**
Pyramid Levels	23	25	27	28	30	
Max Usage	27 Mb	100 Mb	155 Mb	251 Mb	390 Mb	
	-24.2	-24.3	-24.3	-24.3	-24.3	-24.3
FP Stage	0.00	0.00	0.00	0.00	0.00	0
Conv. Stage	0.00	0.00	0.00	0.00	0.00	0
DT Stage	0.00	0.00	0.00	0.00	0.00	0
Backtrack Stage	41.7	38.7	40.4	41.7	42.5	41.0
Features Pyramid	12.7	11.84	11.64	11.45	11.30	11.79
	-24.2	-22.4	-24.3	-24.3	-24.3	-23.9
Filters Responses	32.8	29.8	30.9	31.7	32.2	31.5
DT Scores	10.3	9.6	10.0	10.3	10.5	10.1
Others	2.48	1.35	1.21	1.11	1.06	1.44
Results Cache (default)	42.2	10.1	6.8	4.3	2.8	13.2

As seen in the Table 20 above, the maximum memory consumption of the algorithm is totally affected by the removal of the DPMD algorithm remains. The maximum memory consumption is reduced about 24% and the Features Pyramid data structure is now participating at the 12% of the total maximum memory instead of the 32% at the original version (1.1).

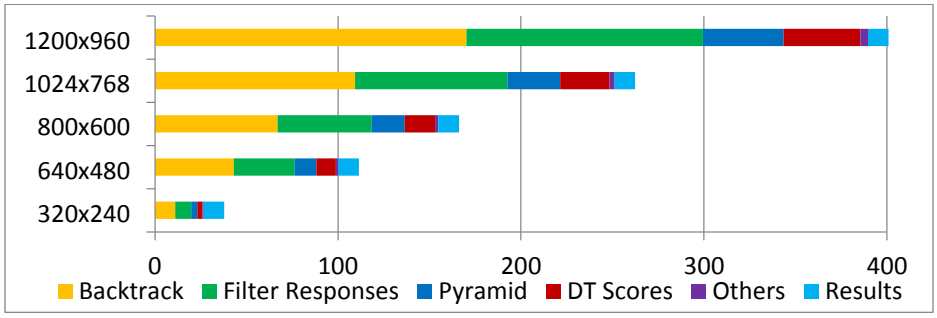


Diagram 10 - TSM v1.2 Max Memory Distribution per Image

At last memory profile diagram (Diagram 11) of the version 1.2 of the algorithm is presented below.

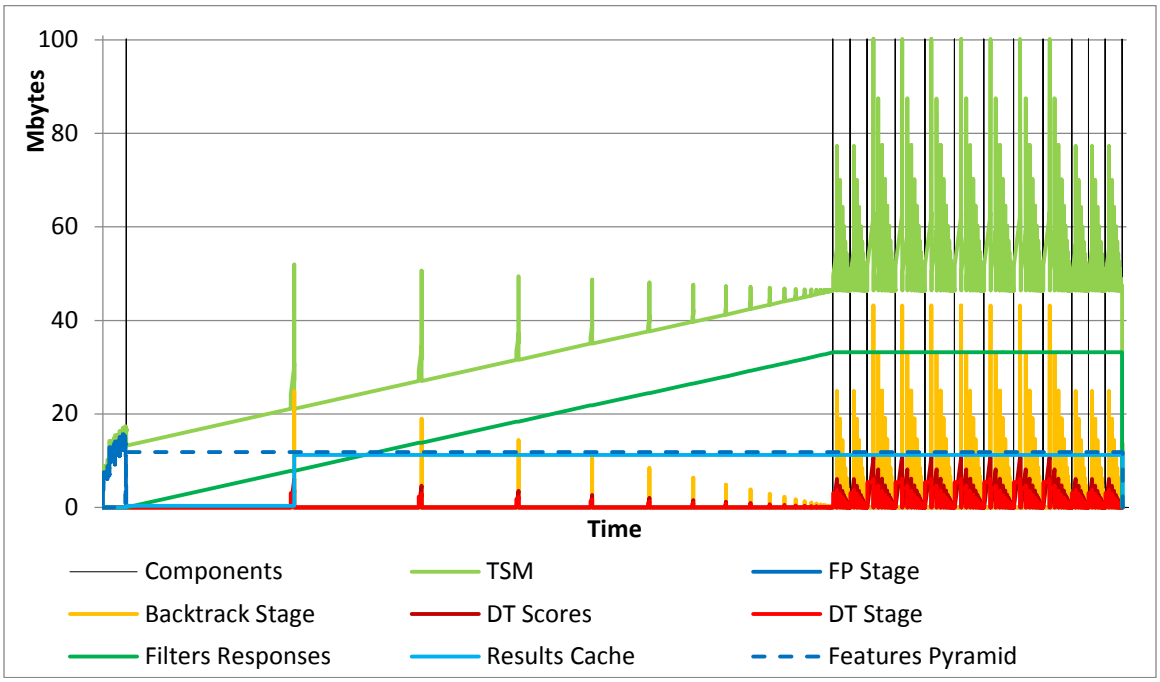


Diagram 11 - TSM v1.2 Algorithm Memory Profile

6.6. *Features Pyramid Stage*

The Features Pyramid stage is the stage of the algorithm where the image pyramid is created and afterwards the features one. This stage is a short one but it is very critical as it is the first one executed by the algorithm. Its results are the input to the Detect stage and required to the detection process to start. In the Table 21 on the right the Features Pyramid stage characteristics are presented.

Table 21 - FP Stage to TSM (%)

Image	Time	Memory	Max
320x240	1,91	1,40	0
640x480	2,25	1,48	0
800x600	2,36	1,39	0
1024x768	2,37	1,40	0
1280x960	2,42	1,41	0
Average	2,26	1,42	0

The Features Pyramid stage consists by three main procedures, the Resize, the Reduce and the HOG one. The Resize and the Reduce one are those who create the image pyramid and scale the images in certain scales. The Resize procedure scales an image at any custom scale while the Reduce one scales images at their half size. The difference of these two procedures is the execution time they need to be completed. The HOG procedure is the one that creates the histogram of oriented gradients descriptors of an image. This procedure creates the Features Pyramid data structure and the actual output of the whole Features Pyramid stage. In the Table 22 below the execution time distribution is presented.

Table 22 - Features Pyramid Stage Execution Time Distribution (v1.1) (%)						
Procedure	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
Resize	13.6	18.2	22.3	23.9	23.9	20.4
Reduce	10.0	10.8	11.1	10.5	11.1	10.7
HOG	72.5	66.9	62.9	61.7	61.4	65.1
Others	3.87	4.17	3.75	3.85	3.64	3.86

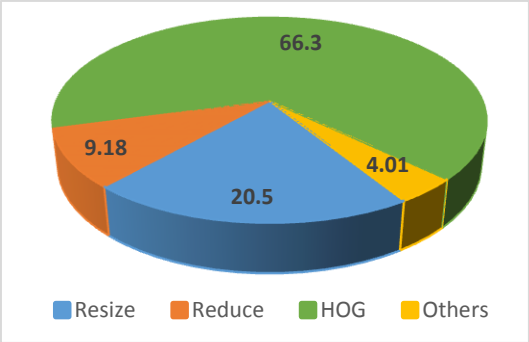


Diagram 12 - FP Stage Execution Time Distribution per Procedure (v1.1) (%)

As seen in Diagram 12 above the main time consumer of the Features Pyramid stage is the HOG procedure holding a little more than the 66% of the whole stage execution time. In chapter 6.6.2 the HOG procedure is explained extended.

As far as the memory consumption inside the Features Pyramid stage the distribution between the stage's procedures is shown in the Table 23.

Table 23 - Features Pyramid Stage Memory Consumption Distribution (v1.1) (%)						
Procedure	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
FP Stage	12 Mb	49 Mb	77 Mb	126 Mb	197 Mb	
Resize	22.3	21.7	21.6	21.6	21.5	21.7
Reduce	15.3	15.4	15.4	15.4	15.4	15.4
HOG	12.4	12.3	12.3	12.2	12.2	12.3
Others	50.0	50.6	50.7	50.8	50.9	50.6
Features Pyramid	+27.9	+24.1	+23.4	+22.8	+22.3	+24.1

The Table 23 shows that the main consumer of the Features Pyramid stage's memory is the temporary one and not the memory consumed inside its main procedures. The reason for this is the temporary image and features pyramids that are created as shown in the Figure 35 (chapter 6.1). This is also visible in the Diagram 13 below where the memory profiling of the stage is presented. On the last line of this table the Features Pyramid output size is presented in ratio with the stage's memory consumption.

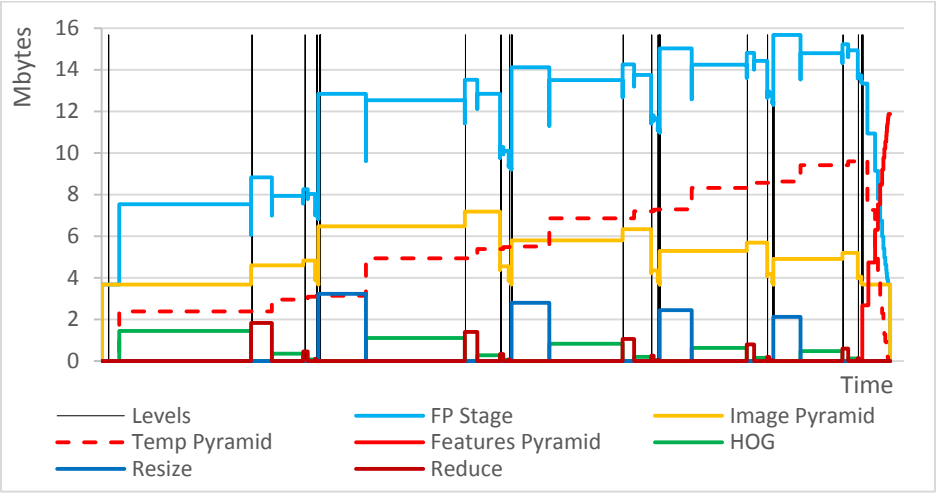


Diagram 13 - Features Pyramid Stage Memory Profile (v1.1)

In the Diagram 13 above the memory consumption profile of the Features Pyramid stage is shown. As seen the temporary features pyramid and the image pyramid are the main consumers. As seen in the beginning of the graph the image pyramid is using an image at the original size for the first level and holds this image until the end as input to the resize procedures. On the other hand, all the rest images of the image pyramid are used for a while and then they are released. The temporary features pyramid is filled with HOG images and at the end is released while the final features Pyramid is created when padding the HOG images.

At the next chapters the Resize, Reduce and HOG procedures are analyzed and memory and time improvements are presented.

6.6.1. Resize

The resize procedure is the one for scaling an image to any custom size. In our implementation is the one that resize the image at the scale of the first interval set of the image pyramid. After that the reduce procedure creates the rest levels of the pyramid. Both the Resize and the Reduce procedures implementation were provided by the algorithm creators in C\C++ script.

The Reduce procedure is the one that takes an image and returns a copy of it in the half size. In our implementation this procedure takes the scaled images of the first interval set and creates copies half of those images for the next sets of intervals as shown in Figure 20 (Chapter 5.5). That is because the reduce procedure is much faster than the Resize one as the scale factor is already known (0.5) and its implementation is customized for it. In Table 24 the amount of time the reduce procedure needs in addition to the resize one is shown.

Table 24 - Reduce to Resize Procedures Comparison (%)						
Image	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
Time	75.1%	73.5%	70.6%	66.2%	66.3%	70.3%
Memory	98.2%	98.6%	98.8%	99.1%	99.3%	98.8%

In the Table 25 below a memory comparison between the Resize and the Reduce procedure is presented. As seen the reduce procedure has a little better memory consumption profile.

Table 25 - Resize & Reduce Procedures Memory Profile			
Input	**Resize**	$X \times Y \times 3$	
	Reduce	$X \times Y \times 3$	
Output	**Resize**	$(X \cdot scale) \times (Y \cdot scale) \times 3$	33%
	Reduce	$(X \cdot scale) \times (Y \cdot scale) \times 3$	33,5%
Temporary	**Resize**	$(X \cdot scale) \times Y \times 3 + (X + Y) \cdot 3 + ((X \cdot scale) + (Y \cdot scale)) \cdot 6$	67%
	Reduce	$(X \cdot scale) \times Y \times 3$	66,5%
Max	**Resize**	$(X \cdot scale) \times Y \times 3 + (X) \cdot 3 + (Y \cdot scale) \cdot 6$	99,6%
	Reduce	$(X \cdot scale) \times (Y \cdot scale) \times 3 + (X \cdot scale) \times Y \times 3$	100%

In Diagram 14 below the ratio of memory consumption and execution time needed by each procedure is shown according to the size of the image they use. As seen, both procedures react the same way to image size increments.

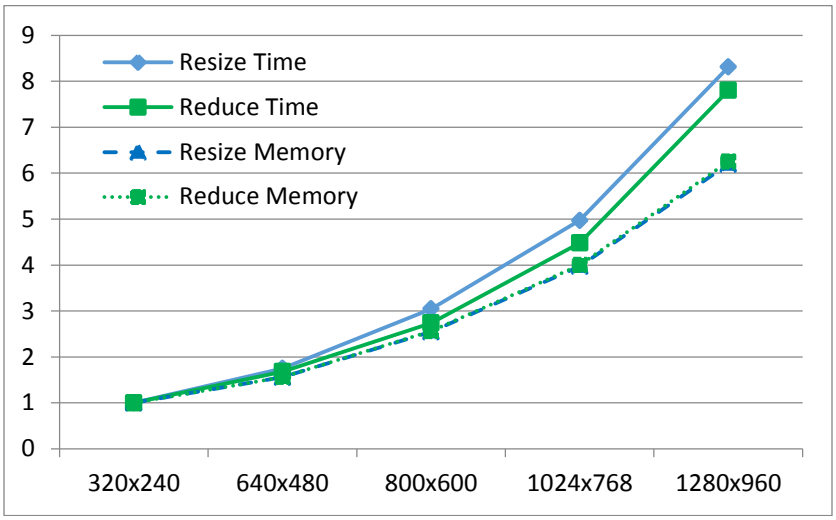


Diagram 14 - Resize and Reduce Procedure Growth Trend per Image

6.6.2. HOG

The HOG procedure is the one that creates the Histogram of Oriented gradients descriptor described in chapter 5.4. This procedure is the greatest time consumer of the Features Pyramid stage as shown in Diagram 12 and Table 22 (Chapter 6.6). In Table 26 below the memory profile of the HOG procedure is presented.

Table 26 - HOG Procedure Memory Profile		
Input	$X \times Y \times 3$	
Output	$\left(X/4+6\right)\times\left(Y/4+6\right)\times 32$	61%
Temporary	$\left(X/4+2\right)\times\left(Y/4+2\right)\times 19$	38%
Max	$\left(\left(X/4+6\right)\times\left(Y/4+6\right)\times 32\right)+\left(\left(X/4+2\right)\times\left(Y/4+2\right)\times 19\right)$	100%

As it is sensible the larger an image is the more memory is needed for the HOG procedure. As shown in Diagram 15 the ratio of memory consumption between different levels of the features pyramid is exponential both in temporary and the results memory which are increasing as the image sizes increases. The perpendicular thin red line in this graph shows how greater is the memory needed for the first interval set of the features pyramid in addition to the rest levels.

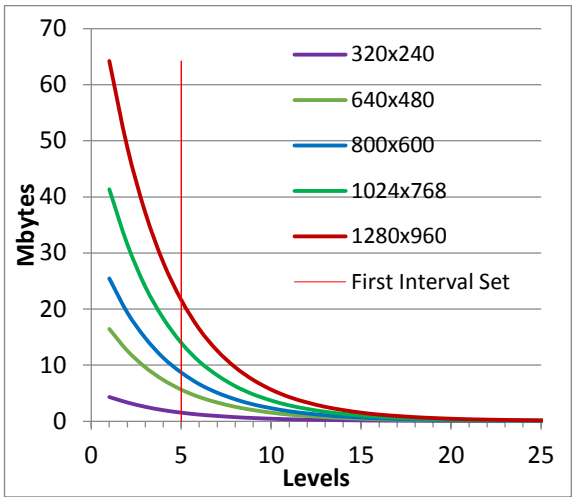


Diagram 15 - HOG Procedure Max Memory per Level

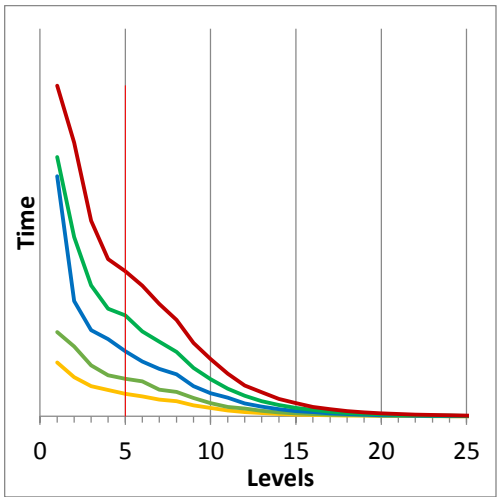


Diagram 16 - HOG Procedure Time Consumption per Level

At the Diagram 16, the time consumption that each level needs at different size images is shown. The HOG procedure has the same attitude at time consumption as in the memory one. Again the red thin line in the graph divides the time consumption needed for the first interval set.

As seen in Figure 35 (Chapter 6.1) the Features Pyramid is not created directly by the results of this procedure but the arrays are padded first. This happens in order to have an accurate convolution process later. The padding procedure costs in the Features Pyramid stage a small amount of time and temporary memory. These costs can be avoided if the padding procedure could be done inside the HOG procedure saving its results in previously padded arrays. This technique produces a new flow diagram of the Features Pyramid stage as shown in Figure 38.

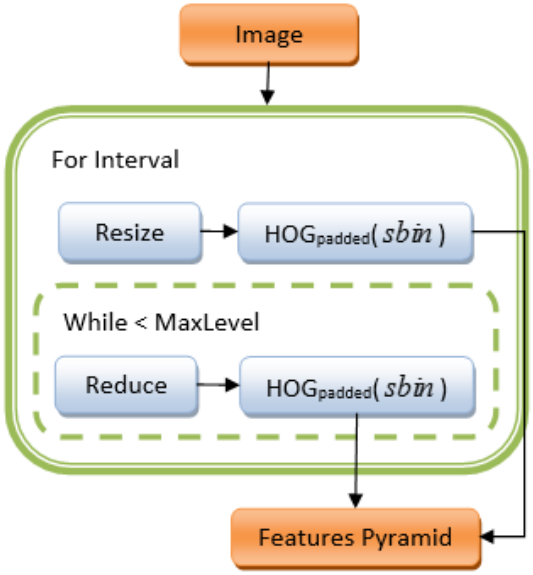


Figure 38 - Features Pyramid Stage Execution Flow (v1.3)

In order to nest the padding procedure inside the HOG one a series of changes inside the procedure in the way the memory pointers are used was made. The time cost of this change is closed to zero and it could not be able to be measured in action, it is only theoretically understandable, although the total time consumption of the Features Pyramid stage was reduced. The results of this improvement in the Features Pyramid stage are shown in Table 27 (Chapter 6.6.3).

6.6.3. Features Pyramid Stage v1.3

At the chapter 6.6.2 a new version of the HOG procedure was presented. This version creates already padded HOG images changing the Features Pyramid stage flow diagram as shown in the Figure 38. This change in addition to implementation changes inside the stages procedures caused changes to the stage's time and memory tables as shown below. In Table 27 the effect of these changes on the execution time of stage is presented.

Table 27 - Features Pyramid Stage Execution Time Distribution (v1.3) (%)						
Procedures	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
v1.2	-3.46	-3.25	-4.25	-2.27	-3.82	-3.41
Resize	16.2	20.2	22.1	21.7	22.4	20.5
Reduce	11.0	10.4	10.1	10.0	10.2	10.3
HOG	72.3	68.4	66.8	67.2	66.6	68.3
Others	0.53	1.03	0.93	1.00	0.89	0.88

By the Table 27 data it is visible that the changes inside the Features Pyramid stage and its procedures reduced the execution time of it for about 3.5%. This reduction is actually caused

because of the removal of the HOG images padding procedure in the end of the stage as it is visible in this table.

As far as the memory consumption of the Features Pyramid stage the Table 28 shows the effect of the changes.

Table 28 - Features Pyramid Stage Memory Consumption Distribution (v1.3) (%)						
Procedures	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
FP Stage	-13.2	-16.8	-17.6	-18.2	-18.7	-16.9
Resize	25.6	26.1	26.2	26.4	26.4	26.2
Reduce	17.6	18.5	18.7	18.8	18.9	18.5
HOG	20.9	18.1	17.6	17.1	16.7	18.1
Others	35.9	37.2	37.5	37.7	37.9	37.2
Features Pyramid	32.2	29.0	28.4	27.8	27.4	29.0

The removal of the temporary features pyramid is the main reason of the reduction of the memory consumption of the Features Pyramid stage for about 17%. This was the effect of the new HOG procedure implementation that creates already padded HOG images. In the Diagram 17 the new memory profiling graph is presented.

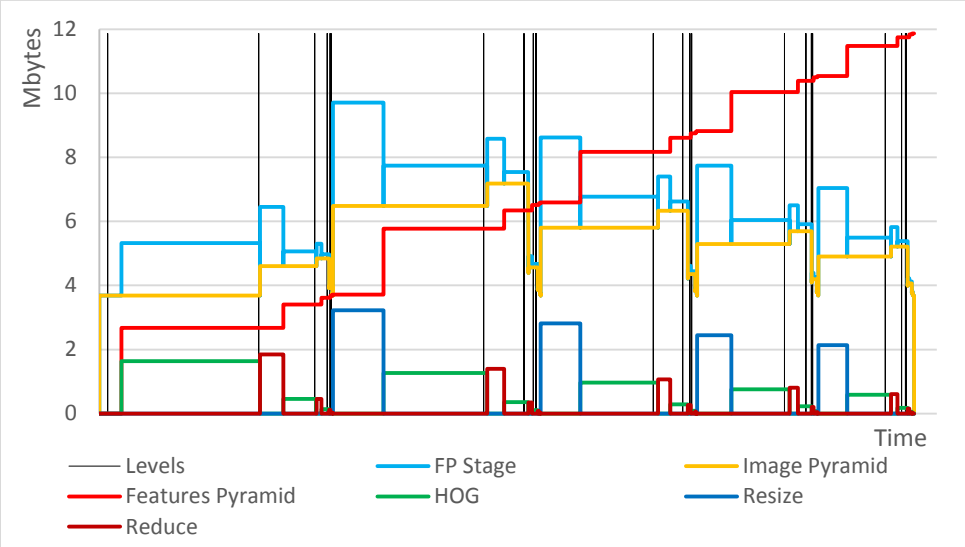


Diagram 17 - Features Pyramid Stage Memory Profile (v1.3)

As seen in the Diagram 17 above the Features Pyramid output is creating during the execution of the Features Pyramid stage. The main memory consumers are the image pyramid that is used temporary and the output data of the stage, the Features Pyramid.

The features Pyramid stage does not participate at the maximum memory consumption limit the algorithm reaches. For this reason the memory consumption reduction is not an important achievement on this version (see version 3.x, Chapters 6.19 and 6.20). On the other hand, the speedup of the stage's execution time is actually the important change achieved. Even if the Features Pyramid stage is a short one, is very important to make shorter as the detection procedure needs its output results in order to begin, as mentioned in the first paragraph of this chapter.

6.7. Features Pyramid

The Features Pyramid data structure is a global one created inside the features pyramid stage and used at the detection process. It handles the HOG images of the image pyramid as described in chapter 5.5. Its life time starts at the end of the features pyramid stage and finish at the end of the detect stage as shown in Diagram 8 (Chapter 6.5). What is worth to focus on is the fact that it holds a noticeable amount of memory at the maximum memory consumption index.

Table 1 - Features Pyramid Max Memory

Image	FP/TSM	Levels	Memory
320x240	12.7%	18	3.4 Mb
640x480	11.8%	23	11.9 Mb
800x600	11.6%	25	18.0 Mb
1024x768	11.4%	27	28.8 Mb
1280x960	11.3%	28	44.0 Mb

By using the new version of the HOG procedure, described in chapter 6.6.2, the HOG images come of, are already padded and immediately registered in the Features Pyramid data structure as shown in the Figure 38. So, actually the Features Pyramid data structure is created during the Features Pyramid stage and is released during the Detect stage. The features images are not useful by the time the Filter Responses are calculated and they are immediately released after that. Even though this data structure participates on the maximum memory consumption indicator. As shown in the Table 20 (Chapter 6.5) the Feature Pyramid holds about the 12% of the algorithms data structures memory.

6.8. Image Pyramid

One sensible question would probably be why the algorithm creates the features pyramid of the image and not the simple image one transferring the HOG procedure inside the detection procedure just before the convolution stage as shown in the Figure 39 below.

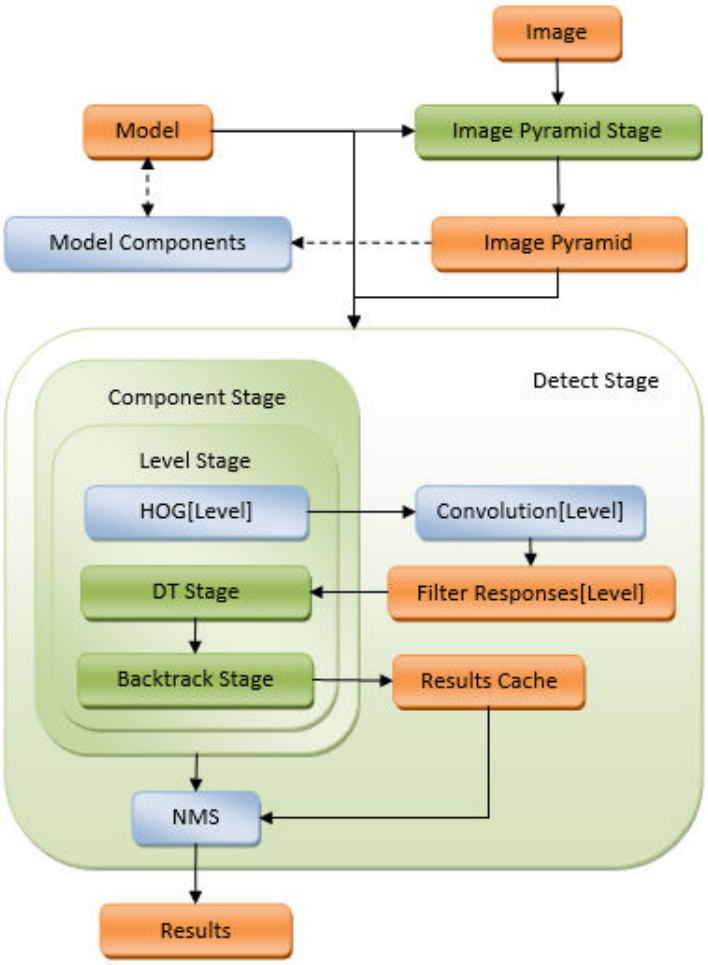


Figure 39 - Image Pyramid in TSM Algorithm

As far as the time consumption this change would not offer any serious benefit, as in a single core CPU the results are the same. On the other hand as far as the memory consumption, the features pyramid needs less memory than the image one. As shown in the Diagram 18 below the size of the image pyramid is larger than the features one for the majority of image sizes until the pyramids coming from an image sized 304x228 pixels and lower. These are very small images where the algorithm anyway consumes very low memory.

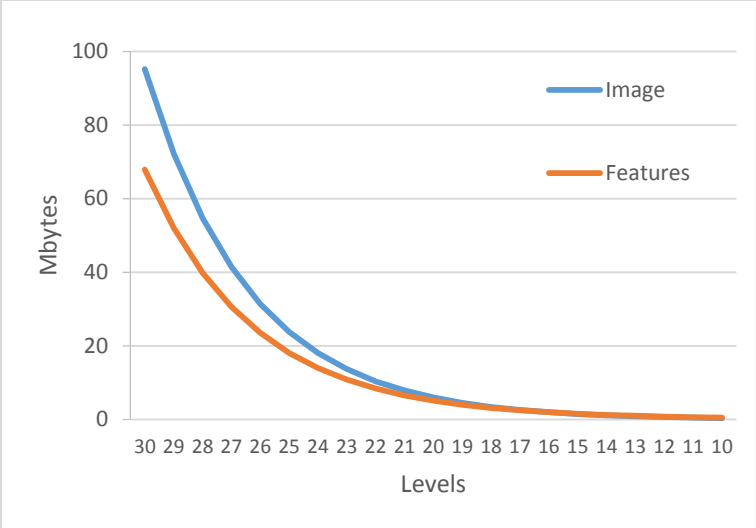


Diagram 18 - Image vs Features Pyramid Memory Consumption

Table 29 - Image vs Features Pyramid						
Levels	28	26	24	22	20	18
Image Size	1213x909	919x689	697x523	528x396	400x300	304x228
Image Pyramid	54.7 Mb	31.4 Mb	18.0 Mb	10.4 Mb	5.9 Mb	2.0 Mb
Features Pyramid	39.8 Mb	23.5 Mb	14.0 Mb	8.4 Mb	5.1 Mb	1.9 Mb
Features/Image	72.9%	74.9%	77.5%	81.0%	85.6%	99.9%

In the Table 29 above the Image and the Features Pyramids sizes are shown. At the last line the ratio between them is also shown. In the next chapters various versions of the algorithm are presented. In some of them the features pyramid data structure does not participate at all in the maximum memory consumption formation of the algorithm and gives the ability of choosing the image pyramid instead of the features one.

6.9. Convolution

The convolution stage is implemented by the convolution procedure which was implemented in C++ by the creators and it is the most important procedure of the algorithm as uses the most resources of the hardware and any small improvement on it can cause large improvement to the whole algorithm execution. As shown in Table 30 the convolution process uses almost the 68% of the complete algorithms execution time. This means that it is very important to find ways to decrease this procedure execution time. In the following graph below (Diagram 19) in the thick lines the time needed for the convolution process according to the features pyramid levels is shown.

Table 30 - Convolution to TSM (%)		
Images	**Time**	**Mem**
640x480	65.6	0
800x600	67.4	0
1024x768	67.8	0
1280x960	68.1	0
1600x1200	68.1	0
Average	67.4	0

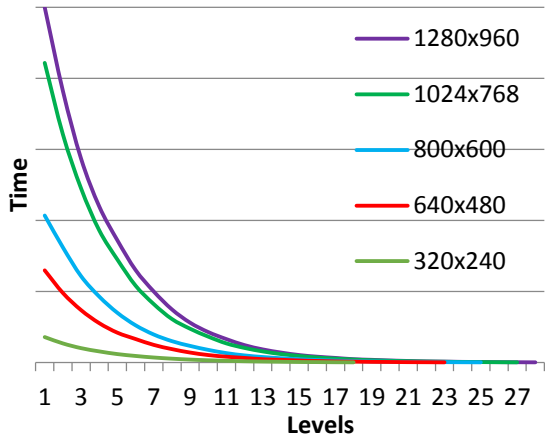


Diagram 19 - Convolution Procedure Time Consumption per Level

In chapter 5.6 the convolution process is described. By this description and by looking at the convolution procedure memory table (Table 31) it is clear that the convolution procedure is a very simple one with a very heavy work to execute. It is actually a many data to a simple process procedure and that is why no great improvements can be applied to it. By looking at the memory table (Table 31) it is easy understandable that the convolution procedure has no space for memory saving improvements.

Table 31 - Convolution Procedure Memory Profile		
Input	$X \times Y \times 32 + 5 \times 5 \times 32$	
Output	$X \times Y$	100%
Temporary	0	0%
Max	$X \times Y \times 32 + 5 \times 5 \times 32 + X \times Y$	100%

In the creators' implementation, the convolution procedure design was used for flexible filters size. By customizing the convolution procedure design for using only $5 \times 5 \times 32$ sized filters we got the Table 32 results. By executing the convolution process at once and for all the Features Pyramid levels, not any extra speedup was succeeded but actually a tiny latency. This might be caused by memory bandwidth overflows.

Table 32 - Convolution Procedure Time Improvements (v1.3) (%)						
Image Size	**320x240**	**640x480**	**800x600**	**1200x768**	**1280x960**	**Average**
Customized for 5x5x32	-13.3	-12.8	-12.9	-12.9	-13.2	-13.0
+ all levels at once	-13.1	-12.6	-12.7	-12.8	-13.0	-12.8

The reduction of the execution time needed for the convolution procedure by almost 13% is a very important change as the convolution procedure holds the 67% of the whole algorithm's

execution time. As shown in the Table 32 above the average of 13% of reducing the Convolution stage execution time, an about 8.7% reduction is succeeded in the whole algorithm execution time. This is a great result!

6.10. Filters Responses

The Filters Responses is a set of arrays used for holding the results of the convolution process between the filters used for landmark detection and the HOG images of the features pyramid data structure. These arrays' data come from the Convolution Stage. What makes this data structure worth to refer is the great amount of memory used that affects the algorithms maximum memory consumption as shown in Table 33.

Table 33 - Filters Responses to TSM Max Memory

Image	FR/TSM (%)	Levels	Memory
320x240	32.8	18	8.7 Mb
640x480	33.1	23	33.2 Mb
800x600	33.1	25	51.3 Mb
1024x768	33.2	27	83.3 Mb
1280x960	33.2	28	129.2 Mb

In the Diagram 20 below the memory consumption of every level at different sizes of images is presented. As is visible the top levels of the Features Pyramid creates high memory size Filters Responses. Reducing the MinLevel parameter of the Features Pyramid, it would cause great reduce on the algorithm memory consumption (see Chapter 9.4).

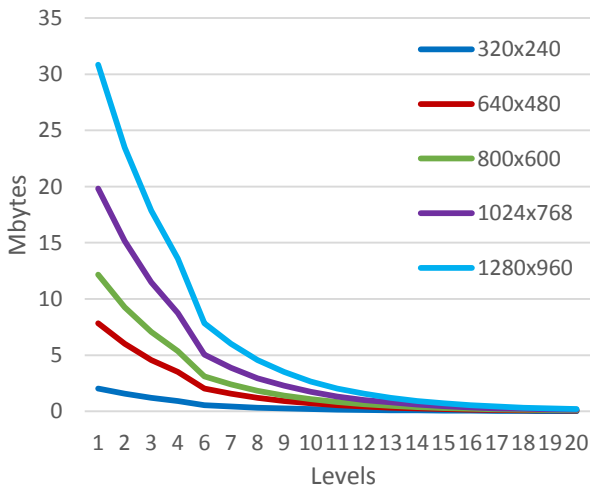


Diagram 20 - Filters Responses Memory Consumption per Level

6.11. Distance Transformation Stage

The Distance transformation (DT) stage is the one explained in chapter 5.7. This stage is used by the algorithm for every pose map tree, (a component) at every Features Pyramid level. It creates a copy of the filter response of every part of the component and it starts a series of DT processes and matrix additions as explained in chapter 5.7 (Figure 25). The Distance transformation stage main part consists by the DT procedure which applies an extended version of the Distance Transformation process (Chapter 5.7) created by the algorithm's creator, implemented in C++.

Table 34 - DT Stage to TSM (%)			
Images	**Time**	**Mem**	**Max**
320x240	31.9	37.7	0
640x480	29.9	37.6	0
800x600	29.5	35.2	0
1024x768	29.3	35.2	0
1280x960	29.3	35.2	0
Average	30.0	36.2	0

In the Table 35 and Table 36 the Distance Transformation time and memory usage distribution is shown. As is clearly visible the main time and memory consumer of the Distance Transformation stage is the DT procedure. What is also visible from the same tables is that the percentage of memory and time usage that the DT stage holds remains almost the same independently the used image's size. At this point a detailed analysis of the DT procedure will be quoted in chapter 6.11.1.

Table 35 - DT Stage Execution Time Distribution (v1.1) (%)						
Image	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
DT proc	92.8	92.1	92.1	91.9	91.8	92.1
Others	7.19	7.95	7.95	8.07	8.20	7.87

Table 36 - DT Stage Memory Consumption Distribution (v1.1) (%)						
Image	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
DT proc	66.7	66.7	66.7	66.7	66.7	66.7
Others	33.3	33.3	33.3	33.3	33.3	33.3

6.11.1. Distance Transformation

In the Face Detection TSM algorithm the creators create an extended the Pedro F. Felzenszwalb and Daniel P. Huttenlocher [13] implementation in C\C++. The Distance transformation procedure as shown in Table 35 holds about the 92% of the DT stage execution time and consumes the 27.5% (Table 37) of the temporary memory the stage uses.

Table 37 - DT Procedure to TSM (%)						
Image Size	**320x240**	**640x480**	**800x600**	**1024x768**	**1200x960**	**Average**
Time	29.6%	27.5%	27.1%	26.9%	26.9%	27.6%
Memory	28.8%	28.7%	26.6%	26.6%	26.6%	27.5%
DT proc Calls	16,031	17,425	18,819	19,516	20,910	18,886

The distance transformation procedure implementation given by the creators is almost like the pseudo-code in Table 38. This implementation uses a lot of temporary memory and creates a great amount of system memory allocations calls. This fact in addition to the number of times this procedure is called (Table 37) during the detection process produces a huge amount of memory consumption and system memory allocation calls as shown in Table 37.

Table 38 - DT Procedure Original Version Implementation (v1.1)	
For y=1; y=Image→height; y++	
Temp = DT-1D(Image→line(y))	*Apply Distance Transformation to every line*
For x=1; x=Image→width; x++	
dt = DT-1D(Temp→row(x))	*Apply Distance Transformation to every row*

In contrast to this version a new one was created in order to reduce the memory consumption and memory allocation system calls. To achieve that a unique temporary memory buffer was created and used for all the instances of 1D transformation function. This way we reduce the system calls for a great amount. In order to extent this version of the distance transformation procedure to multiprocessing computing an instance of this buffer is created for every thread that may execute the 1D-DT function. The pseudo code of this version of distance transformation procedure is shown in Table 39 below. The reduction of memory allocation calls bring also an execution time improvement as is also shown in the same table.

Table 39 - DT Procedure New Version Implementation (v1.3)	
tmp = Array[max(x,y), getMaxThreads()]	*Allocate temporary memory*
For y=1; y=Image→height; y++	
Temp = DT-1D(Image→line(y), tmp[0, currentThread())	*Apply DT to every line*
For x=1; x=Image→width; x++	
dt = DT-1D(Temp→row(x) , Temp[0, currentThread())	*Apply DT to every line*

Table 40 - DT Procedure Memory Profile (v1.1 & v1.3)				
Versions	**Version 1.1**		**Version 1.3**	
Inputs	$X \times Y$		$X \times Y$	
Outputs	$3 \cdot (X \times Y)$	42.9%	$3 \cdot (X \times Y)$	59.8%
Temp	$4 \cdot (X \times Y)$	57.1%	$2 \cdot (X \times Y) + \max(X, Y)$	40.2%
Max	$4 \cdot (X \times Y) + \max(X, Y)$	57.3%	$4 \cdot (X \times Y) + \max(X, Y)$	79.9%

Table 41 - DT Procedure Versions Memory Profile Comparison (1.1 vs 1.3)				
	Original v1.1	**New v1.3**	**Profit**	**Ratio**
Temp Memory	$4 \cdot (X \times Y)$	$2 \cdot (X \times Y) + \max(X, Y)$	$\approx 2 \cdot (X \times Y)$	*≈2*
Memory Allocation	$2 + 2 \cdot (X + Y)$	*4*	$2 \cdot (X + Y)$	$\approx 0.5 \cdot (X + Y + 1)$

As seen in the Table 41 above, the new version of the DT procedure consumes almost two times less temporary memory and keeps the number of the memory allocation calls stable, independent of the image size.

Table 42 - DT Procedure Versions Comparison						
	Images	**320x240**	**640x480**	**800x600**	**1024x768**	**1200x960**
	Levels	18	23	25	27	28
Memory	**v1.1**	251.3 Mb	955.9 Mb	1,475 Mb	2,395 Mb	3,712 Mb
	v1.3	129.1 Mb	485.0 Mb	746.4 Mb	1,209 Mb	1,870 Mb
	v1.3 / v1.1	51.5 %	50.7 %	50.6 %	50.5 &	50.4 %
Allocations	**v1.1**	1,556,320	3,132,520	3,923,460	5,015,440	6,242,320
	v1.3	51,120	65,320	71,000	76,680	79,520
	v1.3 / v1.1	3.28 %	2.09 %	1.81 %	1.53 %	1.27 %
Memory Calls	**v1.1**	40	76	94	119	149
	v1.3	632	1,856	2,628	3,941	5,880
Time	**v1.3 / v1.1**	82.0 %	88.8 %	91.3 %	92.6 %	93.9 %

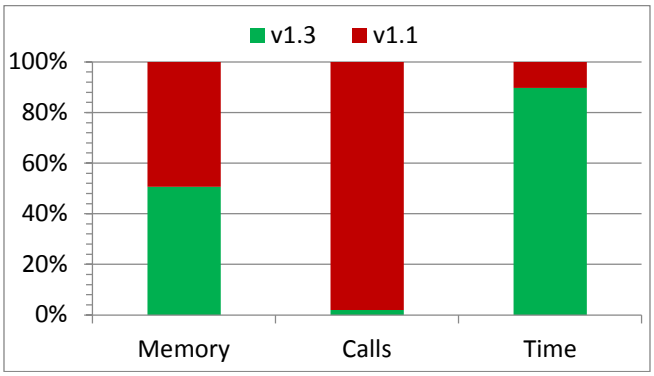


Diagram 21 - DT Procedure Versions Resources Consumption (v1.1 & v1.3)

As seen in Table 42 and visualized in the Diagram 21, the greatest advantage of the new version of the DT procedure is the huge reduction of memory allocation calls. This reduction is also responsible for the small reduction on the DT procedure execution time. In addition a great reduction at the temporary memory is also achieved reducing the memory needed at the half amount. As seen in the Diagram 22, the reduction of the memory allocation calls has greater impact in the algorithm execution time when small size images are used while its impact is less in larger images. This is caused because in small images the ratio between the allocation calls and the size of memory used is larger than in the large ones.

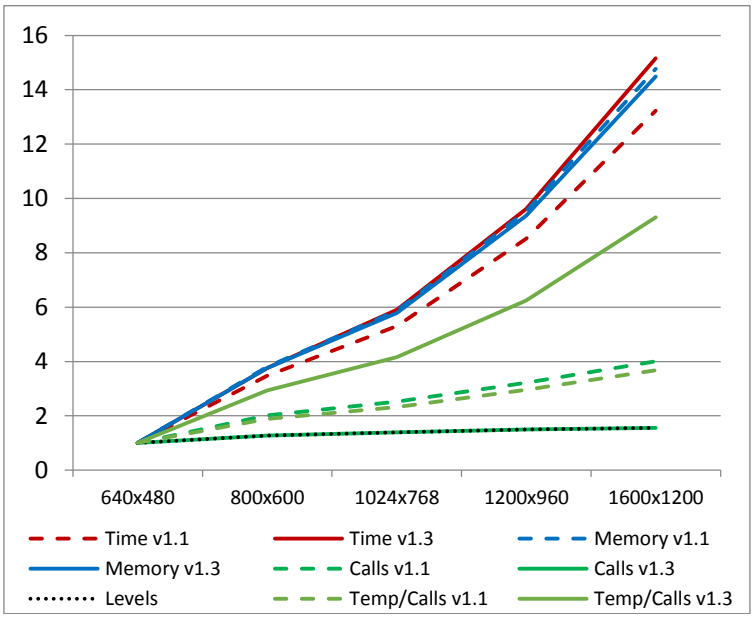


Diagram 22 - DT Versions Growth Trend per Image (v1.1 & v1.3)

In Diagram 22 above the effect of the image size used in the algorithm to the execution time and the memory needed by the DT procedure is shown. As seen, the memory allocation calls are image size independent in the new version. The small increment in the diagram is only caused by the enlargement of the features pyramid levels as shown in the black dotted line. The lines

gradients also show that the new version is the similarly affected by the image size as the old one as far as the memory consumption and the execution time needed.

6.11.2. DT Stage v1.3

In addition to the new version of the DT procedure, one final improvement one the DT stage that has to do with the temporary usage of the parts filters responses come from the convolution stage applied. As shown in Figure 40, each part's filter response gets the distance transformation process applied on it and the result is added to its parental part filter response. This way the filter responses have to be copied to temporary arrays in order to retain their data as they are used from multiple parts of different components.

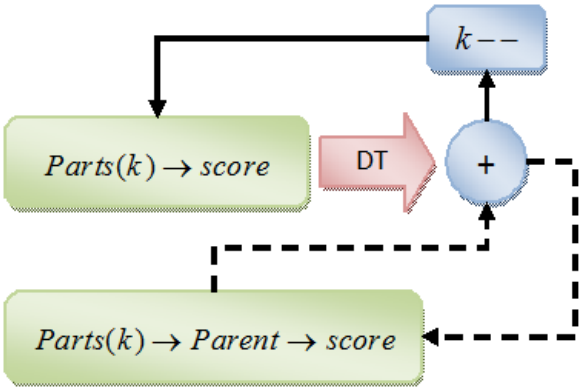


Figure 40 - DT Stage Execution Flow (v1.1)

The implementation of the DT stage is shown in Table 43 as a pseudo-code implementation. The memory allocations and the memory consumption of the original version of every execution of the DT stage are shown in Table 45 below.

Table 43 - DT Stage Original Implementation (v1.1)

```
For  k=1;  k=Parts→length;  k++
   Parts(k) →score = Copy( Responces(
Parts(k)→filterID ) )                          Copy Array Data

For  k=Parts→length;  k=2;  k--
   Child = Parts(k)
   Parent = Parts( Child→parent )
   dt = DT( Child→score )                      Apply DT to Filter Response
   Parent→score += dt                          Add DT Score to Parent FR
```

Trying to create a less memory consuming version of DT stage finally we end up in a new version as shown in the pseudo-code below (Table 44). This version takes advantage of the fact that the majority of parent-child relationships inside the parts of a model's tree are sequential. This expression means that having a part with id *N* its parent id is *N-1*. This is what we call a sequential relationship. In Figure 26 (Chapter 5.7) a series of sequential relationships existences are shown. On our new implementation of DT stage a single array's memory is used for a whole sequential relationship and only when this continuity breaks a new array memory block is allocated. This way all the processes read their data from this array and save their results again on it. The filters responses data are used as read only arrays and no need of coping them exists.

Table 44 - DT Stage New Implementation (v1.3)

```
For  k=Parts→length;  k=2;  k--
   Child = Parts(k)
   Parent = Parts( Child→parent )
   If( !Child→score )
      Child →score = Copy( Responses( Child→filterID ) )        Points to F. Response

   Child→score = DT( Child→score)                               Apply DT
   If( !Parent→score ) Parent→score = Responces(
Parent→filterID )                                               Points to F.Response

   Child→score += Parent→score                                  Add Parent F.Response to
                                                                DT Score
   Parent→score = Child→score                                   Parent Points to Arrays
                                                                Sum
```

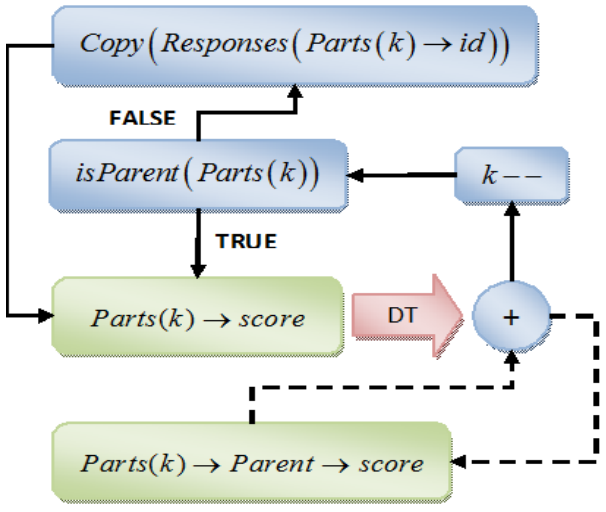


Figure 41 - DT Stage Execution Flow (v1.3)

Finally a comparison between the new (v1.3) and the original (v1.1) implementation is shown in Table 45. It is clear that the new version of the DT stage implementation the memory allocations and consumption is much less than the original. The profits of this change is shown in the last column.

Table 45 - DT Stage Versions Comparison (1.1 vs 1.3)			
	Average v1.1	**Average v1.3**	**v1.1/v1.3**
DT Procedure Calls	54	54	1
Memory Allocations	164	10	≈16.4
Memory Consumption	$164 \cdot (X \times Y)$	$10 \cdot (X \times Y)$	≈16.4
Array Additions	54	54	1

At last the final improvement of the DT stage memory consumption and execution time by applying both the DT procedure and the DT stage new implementation versions can be seen in Table 46. As seen both in time and memory consumption the new version achieves a great improvement not as far as the DT stage but also for the TSM algorithm as the DT stage is the second more important stage of the algorithm.

Table 46 - DT Stage Consumption Improvement (v1.3) (%)						
	320x240	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
Time	-20.0	-14.9	-12.8	-11.7	-10.6	-14.0
Temporary Memory	-64.2	-64.4	-64.4	-64.5	-64.5	-64.4

These improvements in both time and memory inside the DT stage are very important as the DT stage is the second most important as far as the detection procedure and most consuming stage as far as the hardware resources needed for the algorithm.

6.12. DT Scores Data Structure

The DT scores data structure is the output data of the DT stage. This data is a series of tables containing information about the parts filters responses' results when the DT procedure is applied to them. The algorithm keeps two table for every part of each component. This tables are used by the Backtrack procedure in order to make the landmark estimation. For every component the DT stage except of these parts scores, it return a table containing the whole component score as described in chapter 5.7. This table reveals if there is a face detection within the image and is used by the Find procedure.

Table 47 - DT Scores Memory Profile (%)			
Image	**Max**	**Mem**	**Mem (Mb)**
320x240	10.3	14.2	123.5 Mb
640x480	10.6	14.2	470.8 Mb
800x600	10.7	13.2	728.9 Mb
1024x768	10.8	13.1	1.183 Mb
1280x960	10.8	13.1	1.835 Mb
Average	10.6	13.6	

The DT scores data are one of the shareholder of the algorithm maximum memory consumption holding the 10.6% of it. The memory consumption of this data structure is increasing as the image size increase as shown in the Diagram 23. What is significant is the fact that the DT Scores constitutes the 13.6% of the whole memory usage of the algorithm much larger than the Features Pyramid and the Filters Responses data structures.

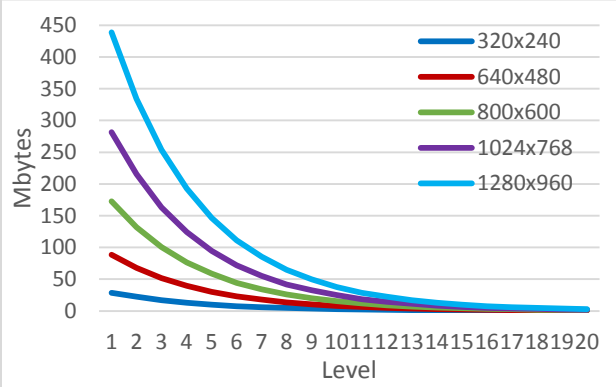


Diagram 23 - DT Scores Memory Consumption per Image

6.13. Backtrack Stage

The Backtrack stage is the one that handles the possible face detection and identifies the landmarks. The first job the Backtrack stage has to do is to check the DT stage scores array for high-score values. This is the Find procedure job. If no high-score values are detected the Backtrack stage is over. On the other hand when high-score values are detected the Backtrack procedure is the one that makes the landmark estimation according to the position of the high-score values and the scale of the corresponding feature image. Finally the results of the Backtrack procedure in combination with the Find procedure ones are filling the results cache. Whenever the result cache is fully filled the NMS procedure is applied to select the correct results, but this process is explained in chapter X. The Backtrack stage flow diagram is shown in Figure 42 below.

Table 48 - Backtrack Stage to TSM (%)

Images	Time	Mem	Max
320x240	0.37	24.0	41.7
640x480	0.39	24.1	43.1
800x600	0.31	26.7	43.3
1024x768	0.22	26.7	43.5
1280x960	0.15	26.8	43.7
Average	0.29	25.7	43.1

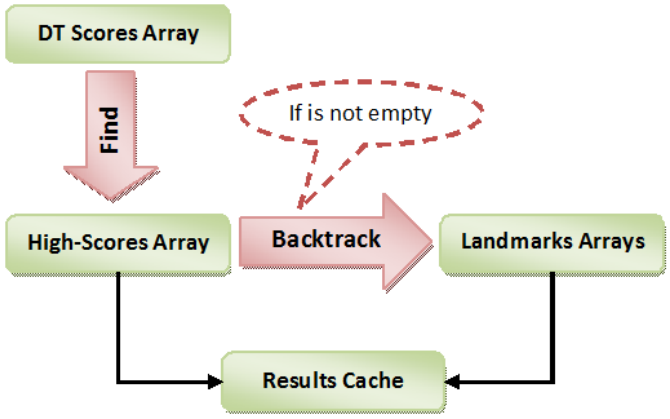


Figure 42 - Backtrack Stage Execution Flow Diagram

As long as the time consumption, the Backtrack stage uses a tiny percentage less than 0.5% of the algorithms execution time and for that reason very few attention is given to that part of the stage. In contrast this stage consumes about the 25% of the algorithms memory consumption and holds the 43% of the maximum one and that is why more attention to memory consumption improvements is given. In Table 50 the memory consumption distribution of the Backtrack stage is shown.

Table 49 - Backtrack Stage Execution Time Distribution (v1.1) (%)						
	320x240	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
Find	7.73	4.70	5.10	6.26	8.34	6.43
Backtrack	62.4	52.8	53.9	55.0	52.4	55.3
Others	29.9	42.5	41.0	38.7	39.3	38.3

As referred in chapter 6.2 the Backtrack stage is in a way independent from the image size. This stage's attitude during the algorithm execution is fully dependent by the number of the detection occur. This means that the algorithm may make the minimum usage of this stage if no faces are detected and either the maximum when plenty of faces are detected. This is why it very difficult to profile it. In our profiling process we assume that the Backtrack process makes full detection at every level and component when we are looking for the maximum memory consumption. On the other hand when profiling for total memory consumption we assume that the image is full of faces and we use the profiling settings explained is chapter 6.2. Under these cases the memory profile table of the Backtrack stage is shown in Table 48.

Table 50 - Backtrack Stage Memory Consumption Distribution (v1.1) (%)						
	320x240	640x480	800x600	1024x768	1280x960	Average
Find	0.60	0.60	0.60	0.60	0.60	0.60
Backtrack	33.7	33.7	33.7	33.7	33.7	33.7
Others	65.7	65.7	65.7	65.7	65.7	65.7

At this point an extensive analysis of the two basic procedures of the Backtrack stage is given and after that a small improvement as far as the memory consumption is presented.

6.13.1. Find

The find procedure is the one that checks the Distance Transformation stage scores array for high-score values and returns a vector of indexes to the corresponding pixels. The find procedure is the only part of the Backtrack stage that is executed for every component at every level of the features pyramid. The time needed for this procedure is closed to zero, as it is a very simple and fast procedure. As far as the memory consumption it is described in the Table 51 below.

Table 51 - Find Procedure Memory Profile		
Input	$X \times Y$	
Output	$2 \times \left\lfloor \frac{P_{High-Score}}{Buffer\_Size} \right\rfloor \times Buffer\_Size$	100%
Temporary	0	0%
Max	$2 \times \left\lfloor \frac{X \times Y}{Buffer\_Size} \right\rfloor \times Buffer\_Size$	100%

In the Table 51 above the memory consumption is directly depended by two parameters. The first parameter, $P_{High\text{-}Scores}$, is the number of high-scores detected inside the DT stage results and is unpredictable. The only prediction can be made is that it cannot be larger than the DT scores array size. In the chapter 6.2, statistics about the high-score values produced in the DT stage according to the model used and faces exist within the image.

On the other hand, the *Buffer_Size* parameter is the size of a buffer used in order to minimize the memory consumption of this procedure and avoid the temporary memory used to save the find process results. If this buffer is completely filled, another block of memory of the same size is allocated. This type of implementation of the find procedure makes it image size independent as the memory usage is only affected by the number of detections.

The decision for the default memory buffer size was the result of profiling the find procedure using series of images both in laboratory and into-the-wild environment. The Diagram 24 shows

the probability density of the profiling results. On this graph the high-score values discovered by the Find procedure every time it was called are shown. This results come from the same tests that produced the data tables in chapter 6.2, where the Find procedure profiling is presented.

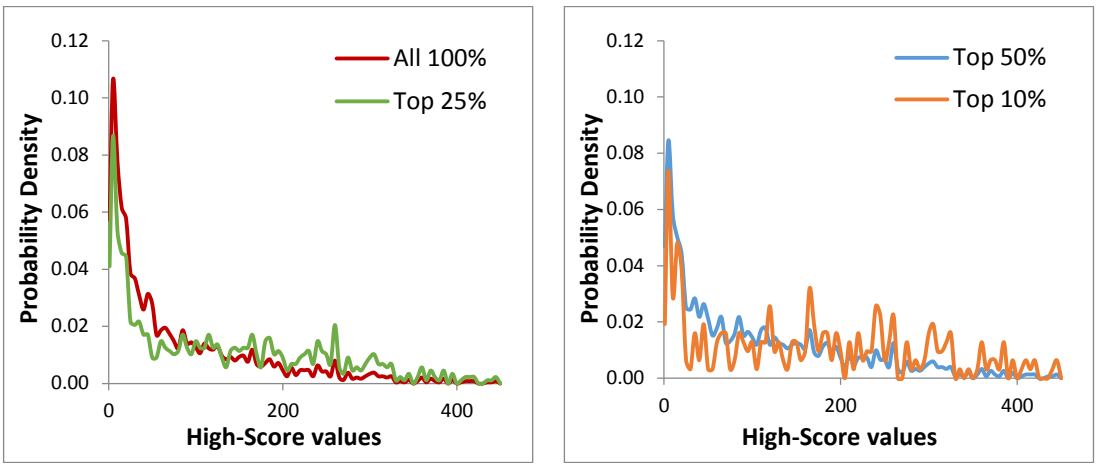


Diagram 24 - Find Procedure High-Score Values Probability Density

In this diagram it is visible that the probability density curve has global maximum close to the lower values. As seen in the Table 52 more than 50% of the results are entered in the area between 1 and 50. This means that as the buffer size increases the wasted memory will also increasing. On the other hand when the buffer size increases the buffer reallocation calls will be decreased. By analyzing the Diagram 24 data, the following tables' results return. In the Table 53 the buffer reallocations per useful find procedure are presented, while in Table 54 the size of memory wasted.

Table 52 - High-Scores per Find	
< 10	21.4 %
< 20	33.8 %
< 50	54.0 %
< 100	70.7 %
< 200	89.3 %

Table 53 - Find Buffer Reallocations per Find										
Buffer	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
Top10	17.8	9.0	6.5	4.8	3.9	3.2	3.1	2.4	2.3	2.3
Top25	13.5	7.0	4.5	3.6	2.9	2.8	2.2	2.1	2.1	2.0
Top50	11.0	5.4	3.7	2.9	2.7	2.1	2.0	2.0	1.9	1.8
All	8.1	4.3	3.3	2.5	2.4	1.8	1.8	1.7	1.6	1.3

Table 54 - Find Buffer Unused Memory per Find (Bytes)										
Buffer	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
Top10	37	49	101	92	115	88	187	84	158	230
Top25	30	48	27	67	65	167	93	168	231	291
Top50	26	20	32	46	134	89	150	214	265	315

All	7	28	82	90	160	125	177	229	275	201

What is interesting in these results is the size of useless memory consumed by the find procedure and the number of system reallocation call according to the buffer size. The ideal size of the Find Buffer would produce the minimum system reallocation calls and the minimum wasted memory. In the Table 55 below the results of Reallocation Calls x Wasted Memory are shown. The desirable buffer size is when the results are lower.

Table 55 - Find Buffer Reallocations x Unused Memory Indicator											
Buffer	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**	**Proposal**
Top10	166	112	164	111	113	**70**	144	**50**	91	130	80.60
Top25	103	84	**30**	60	**47**	117	**50**	89	119	146	70.50
Top50	70	**27**	**30**	**33**	92	**46**	76	105	125	143	60.40
All	14	**31**	68	**57**	96	**58**	78	98	113	65	60.40

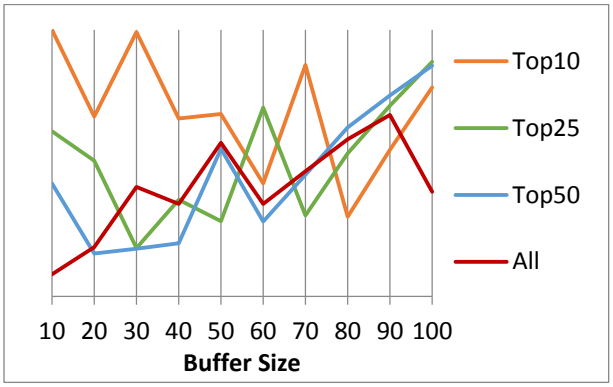


Diagram 25 - Find Buffer Calls x Unused Memory Graph

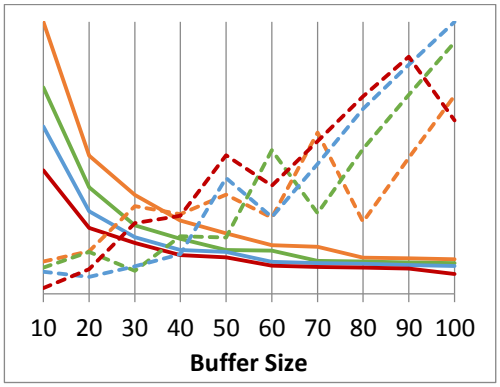


Diagram 26 - Find Buffer Calls And Unused Memory Graph

As seen in the Diagram 25, according to the data of Table 55, the ideal size of the find buffer is 60 memory blocks. This size produces the lowest Reallocation Calls x Wasted Memory results for most samples sets. Using this size for the Find buffer memory the time and memory consumption of the Find procedure is as shown in the Table 56 below. In this table is obvious that this procedure is really a tiny procedure inside the whole detection procedure.

Table 56 - Find to Backtrack Stage (%)						
	320x240	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
Time	9.67	5.50	6.24	7.86	10.63	7.98
Memory	0.60	0.60	0.60	0.60	0.60	0.60

6.13.2. Backtrack

The Backtrack procedure is the one that calculates the landmarks localization after a face is detected. The output data of this procedure are used in the results cache and are part of the algorithms final output results. The Backtrack procedure is using a series of correlation between the high-score values detected in order to correlate them with the corresponding parts (landmarks). This is a simple procedure with a simple complexity running every time the find one detects high-score values. No important improvements were made in procedure that worth to be reported. In fact this procedure uses less than the 0.15% of the algorithms execution time. In Table 57 the memory profiling of this procedure is presented. As seen this procedure as the whole Backtrack stage is image size independent. Its execution time is decreasing as the image size in increasing because all the other size dependent parts of the algorithm are increasing and it stays stable.

Table 57 - Backtrack Procedure Memory Profile		
Input	$2 \times P_{High-Score}$	
Output	$4 \times P_{High-Score} \times Parts$	66%
Temporary	$3 \cdot P_{High-Score} + 2 \cdot Parts \cdot P_{High-Score}$	34%
Max	$3 \cdot P_{High-Score} + 6 \cdot Parts \cdot P_{High-Score}$	100%

Table 58 - Backtrack Procedure to Backtrack Stage (%)						
	320x240	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
Time	62.4	52.8	53.9	55.0	52.4	55.3
Memory	33.7	33.7	33.7	33.7	33.7	33.7

The Backtrack procedure results in addition to the Find ones are used at the end of the Backtrack stage to fill the face detection algorithms results for face detection. At the creators design the Backtrack output data are a bit processed and copied at the results cache data structure. In our implementation the Backtrack procedure return its results in a ready to use from the results cache form. This way the Backtrack stage gains time and saves memory.

6.13.3. Backtrack Stage v1.3

In chapters 6.13.1 and 6.13.2 the implementation of the Find and Backtrack procedures was described. This two procedures where implemented in Matlab script by the creators so no further improvements can be made as they are designed by the beginning at the maximum memory saving mode could be achieved. At Table 50 the “Other” line represent a data copy process that transfers data from the Backtrack output results to the results cache with a small processing. At this point a small modification inside the Backtrack procedure could skip this copy

and processing procedure, as done in the HOG procedure at Features Pyramid stage in chapter 6.6. By doing this modification, changing the data structure and using multiple pointers, the memory saving succeeded is shown in Table 59.

Table 59 - Backtrack Stage Version Comparison (1.3 vs 1.1) (%)						
	320x240	640x480	800x600	1024x768	1280x960	Average
Time	-20.0	-20.8	-24.8	-27.4	-28.3	-24.2
Memory	-65.7	-65.7	-65.7	-65.7	-65.7	-65.7
Max Memory	-24.4	-24.4	-24.4	-24.4	-24.4	-24.4

As seen in the Table 59 the memory consumption reduction is the same size as the "Other" line memory consumption in Table 50. This happens because by the time the Backtrack procedure returns its results in the format the results cache needs, there is no need for extra processing and no need of temporary memory for that processing. This also gains speedup in the Backtrack stage and the most important is that it cause a total reduction of about 10% of the algorithm's maximum memory.

6.14. Results Cache

The Results Cache is a data structure where the detection data are saved. The default Result Cache size can hold up to 10,000 detection results. This means that the Results cache data structure can carry this data by the first detection moment until the end of the algorithms execution where the NMS procedure selects the correct detections as described in chapter 5.10. This amount of memory affects the maximum memory consumption of the algorithm. In Table 60 the increment of the maximum memory consumption that a full Results Cache can cause is shown in the second column. At the third column the total memory used by the algorithm is shown. As is visible about 29.9% of the total memory consumption is allocated for saving detection results.

Table 60 - Results Cache to TSM Memory (%)			
Image	Max	Mem	Mem (Mb)
320x240	+42.2	27.9	137 Mb
640x480	+11.2	28.2	526 Mb
800x600	+7.26	31.0	972 Mb
1024x768	+4.47	31.1	1,581 Mb
1280x960	+2.89	31.1	2,458 Mb
Average	+13.6	29.9	

When the Result Cache cannot hold more data the NMS procedure applies, in order to clear the Result Cache from the useless detection as described in chapter 5.10. These amounts of data removed from the Results Cache are considered to be temporary results memory. According to the profiling rules set in chapter 6.2 the maximum Results Cache temporary memory is shown in Table 60 ("Mem" column). The Results cache temporary memory is Results cache size independent as at the end only a few detection are forwarded as detection results equal to the number of faces detected.

Table 61 - Results Cache Max Memory Participation (%)					
Size	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**
10,000	+42.2	+11.2	+7.26	+4.47	+2.89
8,000	+33.8	+8.97	+5.81	+3.58	+2.31
6,000	+25.3	+6.73	+4.35	+2.68	+1.73
4,000	+16.9	+4.49	+2.90	+1.79	+1.15
2,000	+8.44	+2.24	+1.45	+0.89	+0.58

The effect of the Results Cache to the global algorithms maximum memory consumption is affected by the size of the Results cache. In the Table 61 above the participation of the Results cache in the algorithms maximum memory consumption according to its size is presented. As seen in this table the larger the image is, less the maximum memory consumption is affected. That is because the results cache memory consumption is the same independently the image size, in addition to the rest parts of the algorithm.

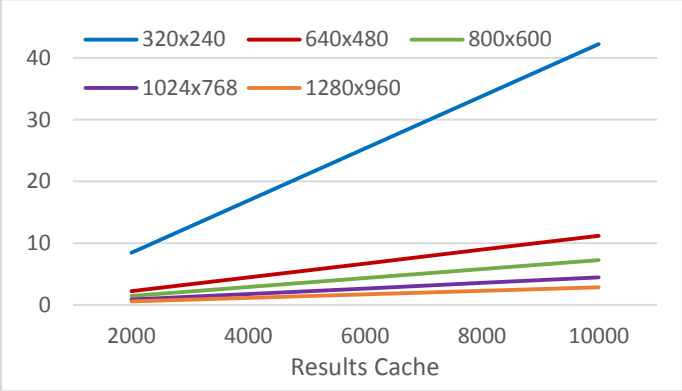


Diagram 27 - Results Cache Participation in TSM Max Memory per Image

The temporary memory consumption values are profiled for the worst case scenarios using perfect images full filled with human faces. The Table 62 below introduces cases closer to real life.

Table 62 - Results Cache Real Temporary Memory (10,000) (%)					
Faces	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**
1	0.90	0.31	0.19	0.14	0.09
2	1.79	0.63	0.37	0.29	0.19
3	2.69	0.94	0.56	0.43	0.28
4	3.58	1.25	0.75	0.57	0.37
5	4.48	1.56	0.93	0.72	0.46

6.15. Non-Maximum Suppression (NMS)

The NMS procedure is executed at least once in the detection process but it is also called whenever the algorithm wants to clear its results cache memory. In the implementation of the algorithm a cache memory is used for keeping the list of the results of each component and level detection process. Depending on the memory resources of our hardware the size of it can vary. Small sized cache can cause the call of this procedure in order to make a selection of the useful results as explained in chapter 5.10. A large sized cache is using memory resources that might be needed and affect the algorithms max memory consumption. The size of the results cache can vary depending on the application used and the available hardware resources. More details about the usage of the results cache memory is referred in chapter 6.14. In this chapter we focus on the usage of the NMS procedure. In Table 63 the NMS procedure profile is shown

Table 63 - NMS Procedure Memory Profile		
Input	$Cache\_Size \cdot \left(9 + 4 \cdot AVG\left(Find_{Results}\right) \cdot AVG\left(Parts\right)\right)$ $AVG\left(Find_{Top50}\right) \approx 100 \Rightarrow 21,850 \times Cache\_Size$	
Output	$-\left(Cache\_Size - Faces\right) \times \left(4 \times AVG\left(Find_{Results}\right) \times AVG\left(Parts\right)\right)$ $AVG\left(Find_{Top50}\right) \approx 100 \Rightarrow -21,846 \times Cache\_Size$	0%
Temporary	$12 \times Cache\_Size$	100%
Max	$12 \times Cache\_Size$	100%

The NMS procedure as the Backtrack stage does, is image size independent. It is only affected by the results cache contents. If the results cache is full, depending on its size it needs more time to execute. Using the creators default cache size at the size of 10,000 the NMS execution of a full one is about 0.1% to 0.03% of the algorithms execution time. Even if the NMS procedure is called multiple times the effect to the whole algorithm execution is tiny. The same fact comes with the memory consumption of the NSM procedure as seen in Table 64. These mean that the NMS procedure is a costless one in addition to the Max memory limitation that can cost when it is emptying the results cache memory or keeping the results memory cache in smaller size.

Table 64 - NMS Consumption (Results Cache = 10,000) (%)					
Calls	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**
Time					
1	0.16	0.10	0.07	0.05	0.03
2	0.32	0.20	0.15	0.10	0.07
3	0.48	0.30	0.22	0.15	0.10
4	0.64	0.40	0.29	0.20	0.14
5	0.80	0.49	0.37	0.24	0.17

Memory					
1	0.06	0.01	0.01	0.01	0.00
2	0.11	0.03	0.02	0.01	0.01
3	0.17	0.04	0.03	0.02	0.01
4	0.22	0.06	0.03	0.02	0.01
5	0.28	0.07	0.04	0.03	0.02

In the Diagram 28 below the number of the NMS procedure calls are presented according to the result cache memory size and the samples group used. The Top10 samples groups (Chapter 6.2) is shown using dotted lines, the Top50 the dashed lines and the continuous is for all. As is visible when the results cache size is large (as the default) the NMS procedure is called few times even when the number of detection results is great.

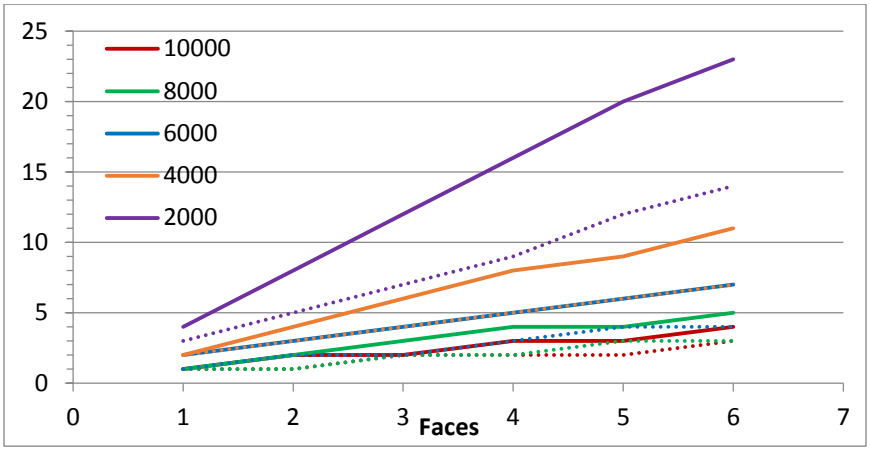


Diagram 28 - NMS Procedure Calls per Results Cache Size

6.16. TSM Face Detector v1.3

After applying all those changes inside the stages described in the previous chapters the algorithms execution time is affected as shown in Table 65 below. In this table the impact of the changes inside each stage is also shown. At this stage, using all these changes inside every stage the algorithm is reached to an extended new version, version 1.3.

Table 65 - TSM v1.3 Execution Time Comparison (Compared to v1.2) (%)						
	320x240	**640x480**	**800x600**	**1024x768**	**1200x960**	**Average**
Levels	18	23	25	27	28	
TSM	-15.1	-13.2	-12.6	-12.3	-12.2	-13.1
F. Pyramid	-0.07	-0.07	-0.10	-0.05	-0.09	-0.08
Convolution	-8.75	-8.64	-8.74	-8.81	-8.98	-8.78

DT	-6.38	-4.46	-3.76	-3.43	-3.11	-4.23
Backtrack	-0.07	-0.08	-0.08	-0.06	-0.04	-0.07

In the table is clear that the reduction of the algorithms execution time is the reduction of the Convolution stage one. The 8.8% of the total 13% is caused by the convolution stage. The second main participant at this reduction is the Distance Transformation stage with correspondence at about 4.2%. These results are very sensible as this two stages hold the main parts of the algorithm's execution time as presented in Diagram 9and Table 18 (Chapter 6.5).

In the Diagram 29 and Diagram 30 below compared to the corresponding Diagram 9 (Chapter 6.5) makes it clear that despite this changes, the algorithms time distribution has not actually change. As seen in both graphs the Convolution stage still stays at the top of the time consumption pyramid using almost the 67.5% of the algorithms total time. It is clear that the algorithms execution time is dependent by the amount of the processed data used in the Convolution stage primary, and in the DT stage secondary. As described in chapters 6.9 and 6.11 the best effort for accelerating the convolution and distance transformation process was given. For further improvement of this procedures other techniques have to be used (ex. Chapter 8 - multithreading) that are explained in following chapters.

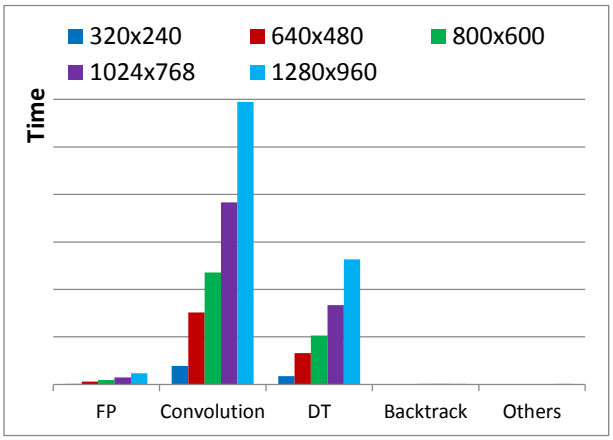


Diagram 29 - TSM v1.3 Execution Time Distribution

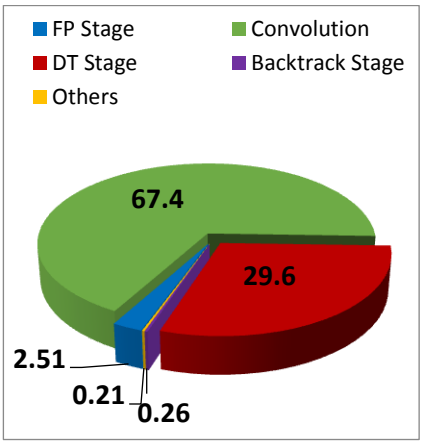


Diagram 30 - TSM v1.3 Execution Time Distribution per Stage

In the Table 66 below the new memory distribution of the algorithm is shown. The same distribution table is also graphically displayed in Diagram 31 below it. As seen the memory distribution ratios has change a bit as a result of the changes inside the DT and the Backtrack stages. The effect of those changes are shown in the Table 66. As seen in this table the greatest memory consumer is the temporary results. This is because in the profile process the scenario of a full faces image is used as explained in chapter 6.2. On the other hand the DT stage and its output data, the DT scores, are still the main memory consumers. Despite that, the DT stage memory reduction achieved, caused about 26.5% memory reduction to the whole algorithm.

Table 66 - TSM v1.3 Memory Consumption Distribution (Comparisons to v1.2) (%)							
		320x240	640x480	800x600	1024x768	1200x960	Average
	TSM	0.49 Gb	1.87 Gb	3.13 Gb	5.08 Gb	7.90 Gb	
		-43.6	-43.8	-43.5	-43.5	-43.5	-43.6
Stages	F. Pyramid	2.16	2.19	2.03	2.03	2.03	2.09
		-0.18	-0.25	-0.24	-0.26	-0.26	-0.24
	Convolution	0.00	0.00	0.00	0.00	0.00	0.00
	DT	27.4	27.2	25.1	25.1	25.1	26.0
		-27.7	-27.7	-25.7	-25.7	-25.7	-26.5
	Backtrack	14.6	14.7	16.2	16.3	16.3	15.6
		-15.7	-15.8	-17.5	-17.6	-17.6	-16.8
Data	F. Pyramid	0.69	0.64	0.58	0.57	0.56	0.61
	F. Responses	1.78	1.78	1.64	1.64	1.64	1.69
	DT Scores	25.2	25.2	23.3	23.3	23.2	24.0
	Results	25.6	27.6	30.7	30.9	31.0	29.1
	Others	0.23	0.10	0.08	0.06	0.06	0.11

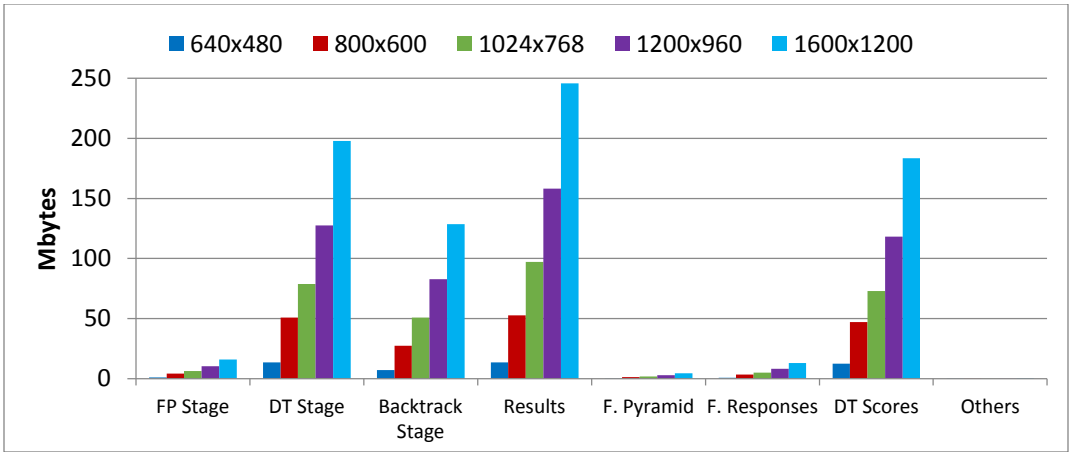


Diagram 31 - TSM v1.3 Memory Consumption Distribution

What is worth to mention is that most of the memory consumption shown in Table 66 and Diagram 31 is used for useful data that cannot be avoid. For example a part of the 2% of the memory used in the Features Pyramid stage is used for the images of the image pyramid. This data are used temporary but they cannot be avoided as they are necessary for the procedure. The results memory that consumes about 29% of the memory is used for saving the detections results that are also useful and important data. Only the Backtrack and DT stage memory usage of 41.5% is actually real temporary memory.

In addition, the maximum memory consumption distribution is formed as shown in the Table 67 below. By the Table 67 data it is obvious that the maximum memory consumption factor is critically constitute by the algorithm's critical data structure as the DT scores, the filters responses and the results cache. The Backtrack stage temporary memory is the only temporary memory that participates the maximum memory consumption.

Table 67 - TSM v1.3 Maximum Memory Consumption (Comparisons to v1.2) (%)						
	320x240	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
TSM v1.3	20.5 Mb	77.8 Mb	121 Mb	196 Mb	304 Mb	
	-22.3	-22.2	-22.0	-21.9	-21.9	-22.1
FP Stage	0	0	0	0	0	0
Conv. Stage	0	0	0	0	0	0
DT Stage	0	0	0	0	0	0
Backtrack Stage	40.9	41.9	42.1	42.2	42.3	41.9
	-10.2	-10.5	-10.5	-10.6	-10.6	-10.5
F. Pyramid	0	0	0	0	0	0
	-12.7	-11.8	-11.6	-11.4	-11.3	-11.8
F. Responses	42.5	42.6	42.6	42.5	42.5	42.5
DT Scores	13.4	13.7	13.8	13.8	13.8	13.7
Others	3.22	1.74	1.55	1.43	1.35	1.86
Results Cache	+54.7	+14.4	+9.33	+5.74	+3.70	+17.6

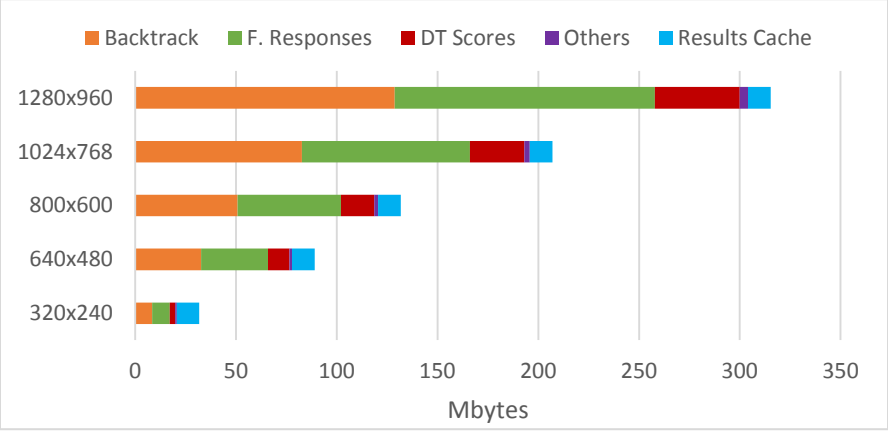


Diagram 32 - TSM v1.3 Maximum Memory Consumption Distribution per Image

As seen in the Table 67 the maximum memory consumption of the algorithm is decreased about 22%. This is basically because in this version of the algorithm the features pyramid images are released every time the filters responses are calculated. Despite the great decrease of the algorithms total memory consumption the maximum memory one was actually reduced at least as only the Backtrack stage part of it succeed a real decrement. The great reduction of the

memory used in the DT stage is outshined by the memory needed in the Backtrack one as shown in the Diagram 33.

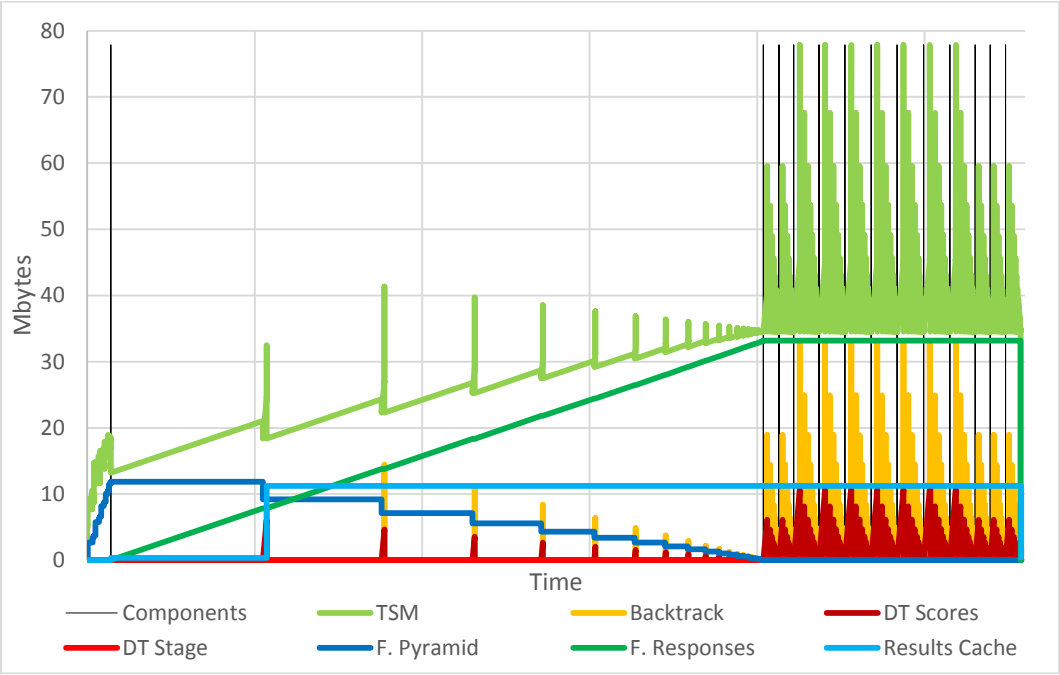


Diagram 33 - TSM Algorithm v1.3 Memory Profile

The maximum memory consumption has reached at an end on this version. A new version (version 2.x) of the algorithm is presented in the next chapter (chapter 6.17) that is customized for further decrease of the maximum memory consumption. The differences of the next versions relatively to the version 1.x of the algorithm is that by the version 2.x and above the algorithm execution flow and its architecture is changed and customized losing its parental relation with the Parts Based Detection algorithm.

6.17. TSM Face Detector v2.1

In this chapter a new version of the TSM algorithm is presented. This version is called version 2.1. The reason that it is dissociated by the version 1.x is because in this version the algorithm is customized to the face detection procedure disconnected by its parental algorithm, the DPBD algorithm. This separation gives also the ability of changing the algorithms execution flow. In the Figure 34 (Chapter 6.3) the execution flow of the Detect stage of the original version of the TSM algorithm is shown. At the Figure 43 below version 2.x Detect stage execution flow is shown. This execution flow comes from taking advantage of the one scale model used in the face detection TSM algorithm in contrast to the multi scale models used in the DPBD one.

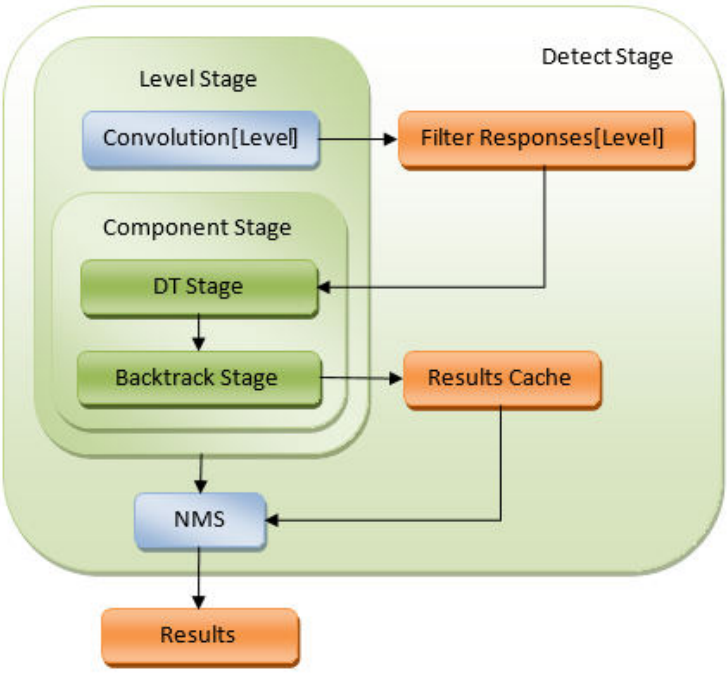


Figure 43 - TSM Algorithm v2.1 Detect Stage Execution Flow

This new version of the Detect stage (Figure 43) of the algorithm does not reduce at all the temporary memory consumption of the algorithm, what it improves is the management of the maximum memory consumption. On this version of the detect stage inverts the levels and components stage so it can release the Filters Responses after the end of every level stage execution. The reduction of the maximum memory consumption relatively to the original version (1.2) is shown in Table 68.

Table 68 - TSM v2.1 Maximum Memory Consumption (Compared to v1.2) (%)						
	320x240	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
TSM	16.3 Mb	60.8 Mb	93.8 Mb	152 Mb	236 Mb	
	-38.9	-39.4	-39.4	-39.5	-39.5	-39.3
FP Stage	0	0	0	0	0	0
Conv. Stage	0	0	0	0	0	0
DT Stage	0	0	0	0	0	0
Backtrack Stage	51.7	53.7	54.1	54.4	54.6	53.7
	-10.2	-10.5	-10.5	-10.6	-10.6	-10.5
F. Pyramid	16.4	15.1	14.8	14.5	14.3	15.0
	-2.73	-2.67	-2.66	-2.64	-2.63	-2.67
F. Responses	12.4	12.9	13.0	13.0	13.1	12.9
	-25.2	-25.3	-25.3	-25.3	-25.2	-25.3
DT Scores	16.9	17.6	17.7	17.8	17.8	17.5

Others	2.65	0.71	0.46	0.28	0.18	0.86
	-0.87	-0.92	-0.93	-0.94	-0.95	-0.92
Results Cache	+69.1	+18.5	+12.0	+7.4	+4.8	+22.4

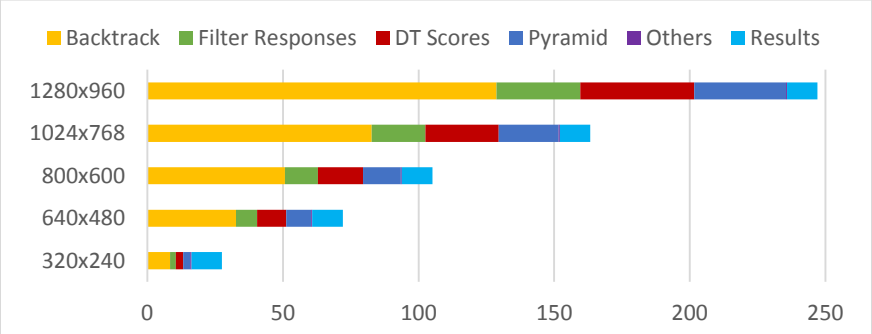


Diagram 34 - TSM v2.1 Maximum Memory Consumption Distribution per Image

As is visible in the Table 68 the Features Pyramid data structure participates in the maximum memory distribution but this is necessary in order to achieve the Filters Responses data structure reduction. In version 1.3 the Features Pyramid data structure does not join the data structures participating the maximum memory consumption but the Filters Responses data structure is fully included. On the other hand in the version 2.1 the Features Pyramid data structure is included almost completed but the Filters Responses one is included only by its first level reducing the total maximum memory for 25%.

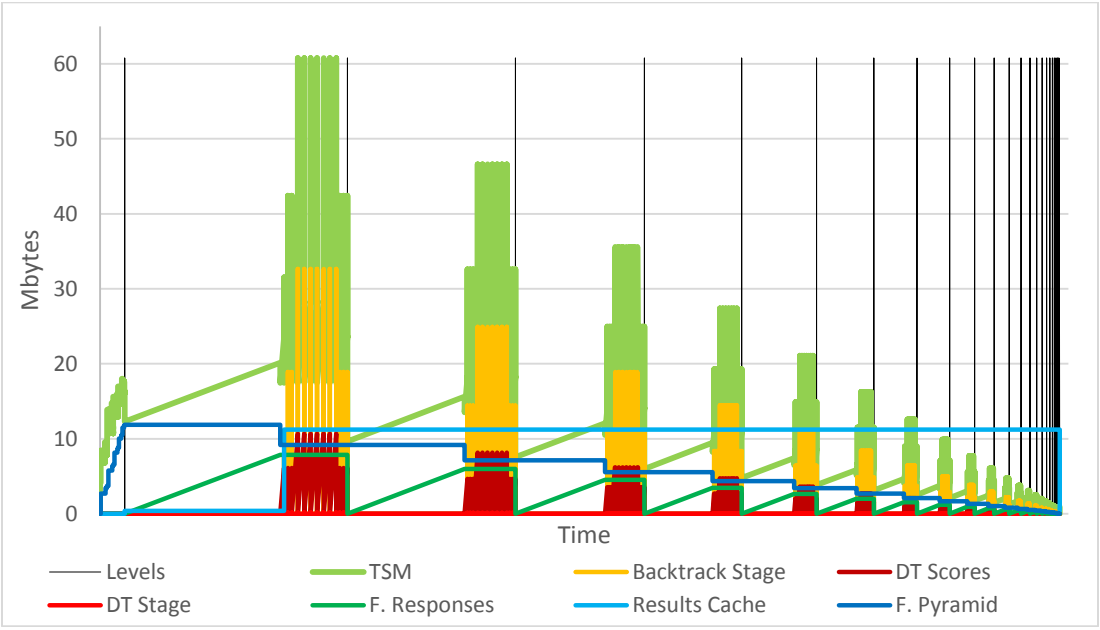


Diagram 35 - TSM Algorithm v2.1 Memory Profile

As seen in the Diagram 35, where the maximum memory consumption profiler is presented, the maximum memory consumption is reached at the point of the first level (bigger size image). At this point it is visible that the biggest size HOG image Filters Responses are added to the rest of the Features Pyramid HOG images waiting for the convolution procedure in the following levels detection procedure. This fact produced the idea of the version 2.2 of the detect stage referred in the next chapter (Chapter 6.17).

Except of the maximum memory consumption factor, the version 2.1 changes have also impact to the algorithms execution time. This impact is tiny and shown in Table 69.

Table 69 - TSM v2.1 Execution Time Comparison (%)						
	320x240	**640x480**	**800x600**	**1024x768**	**1200x960**	**Average**
Levels	18	23	25	27	28	
Vs TSM v1.2	-15.7	-14.4	-13.7	-13.4	-13.3	-14.1
Vs TSM v1.3	-0.61	-1.35	-1.27	-1.22	-1.25	-1.14

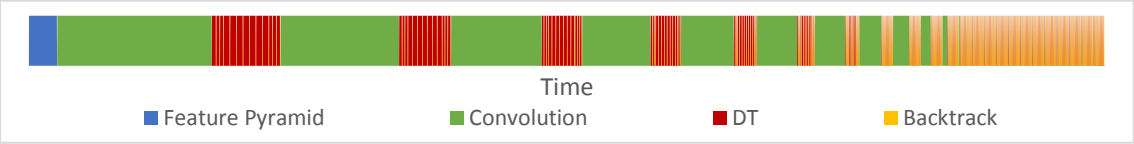


Diagram 36 - TSM v2.1 Algorithm Timeline Profile

At last in the algorithms timeline profile in the Diagram 36 it is visible the new execution flow and how the convolution procedure takes place just in the beginning of every level stage.

6.18. TSM Face Detector v2.2

The TSM algorithm's version 2.2 is almost the same with the 2.1 one with only one change, the order the features pyramid levels are forwarded to the Level stage. On the version 2.1 the algorithm starts the detection procedure from the top to the last level of the features pyramid. As shown in the Diagram 34 (Chapter 6.17), the features pyramid data structure participates at the maximum memory consumption of the algorithm. The time that the top level of the features pyramid enters the level stage and its detection procedure begins, the features pyramid data structure is full of the rest level's features images. This way the most memory consuming level (the top) reaches its maximum memory consumption while the features pyramid is full of features images. This can be changed if the order that the algorithm forwards the levels of the features pyramid change. If the algorithm begins the detection procedure from the last to the top level, the features pyramid data structure will be empty when the top levels detection procedure begins. This way the version 2.2 of the TSM algorithm is created.

The timeline profile of this version of the algorithm is shown in the Diagram 37. It is visible that the algorithm's image pyramid creation takes place in the beginning as also that the diverse level detection described before.

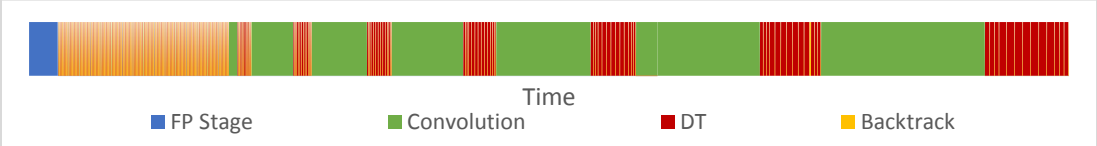


Diagram 37 - TSM v2.2 Algorithm Timeline Profile

At the next graph (Diagram 38) the memory profiling of the algorithm is shown. In this graph is also visible that the as the features pyramid stage is empting from the features images data the base on which the level stage begins is lower.

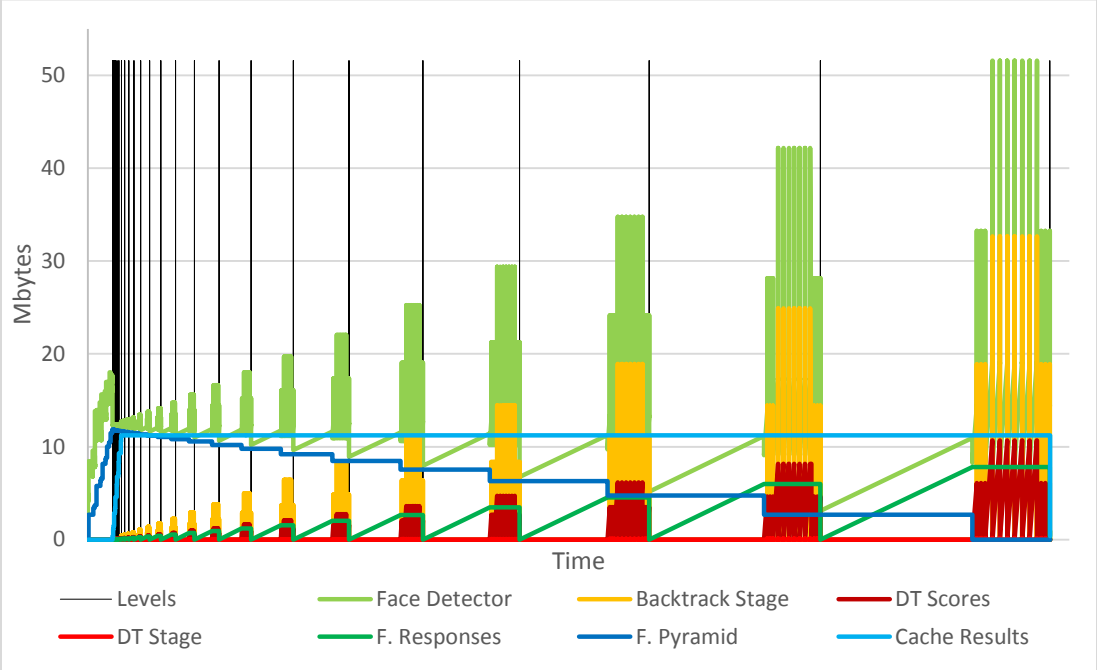


Diagram 38 - TSM v2.2 Algorithm Memory Profile

Observing the algorithms maximum memory profiling at the Diagram 38, it reveals that the maximum memory consumption of the algorithm is reached during the greatest size image detection as in the previous version (2.1) of the algorithm. As seen in the same graph at the time of the maximum memory consumption point the features pyramid are fully released.

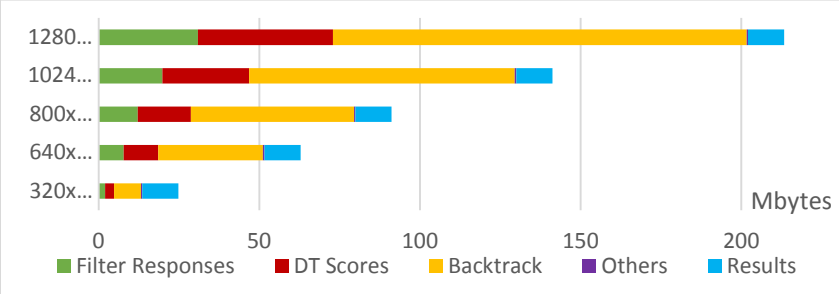


Diagram 39 - TSM v2.2 Maximum Memory Consumption Distribution per Image

At the Diagram 39 above the maximum memory consumption distribution is presented using the data of the Table 70 below.

Table 70 - TSM v2.2 Maximum Memory Consumption (Compared to v1.2) (%)						
	320x240	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
TSM v2.2	13.6 Mb	51.6 Mb	79.9 Mb	130 Mb	202 Mb	
	-49.0	-48.5	-48.4	-48.3	-48.1	-48.5
FP Stage	0	0	0	0	0	0
Conv. Stage	0	0	0	0	0	0
DT Stage	0	0	0	0	0	0
Backtrack Stage	61.8	63.3	63.5	63.6	63.7	63.2
	-10.2	-10.5	-10.5	-10.6	-10.6	-10.5
F. Pyramid	0	0	0	0	0	0
	-12.7	-11.8	-11.6	-11.4	-11.3	-11.8
F. Responses	14.8	15.2	15.2	15.3	15.3	15.1
	-25.2	-25.3	-25.3	-25.3	-25.2	-25.3
DT Scores	20.2	20.7	20.8	20.8	20.8	20.7
Others	3.17	0.84	0.54	0.33	0.21	1.02
	-0.87	-0.92	-0.93	-0.94	-0.95	-0.92
Results Cache	+82.7	+21.8	+14.07	+8.65	+5.56	+26.6

The time results of the 2.2 version of the algorithm are presented in Table 71. This table's data is clearly sensible as the change of the order may cause a tiny speedup cause of better memory management but there is not any important change that could affect the time consumption of the algorithm.

Table 71 - TSM v2.2 Execution Time Comparison (%)						
	320x240	**640x480**	**800x600**	**1024x768**	**1200x960**	**Average**
Levels	18	23	25	27	28	
Vs v1.2	-15.7	-14.5	-14.0	-13.5	-13.3	-14.2

Vs v1.3	-0.68	-1.45	-1.62	-1.34	-1.31	-1.28
Vs v2.1	-0.07	-0.10	-0.36	-0.13	-0.06	-0.14

6.19. TSM Face Detector v3.1

The TSM algorithm version 3.1 is very similar to the 2.x ones. The main change is the unification of the Features Pyramid stage with the Detect one and the order in which the features pyramid levels are pushed to the Level stage. In the version 2.1 the levels were pushed ascending. In the version 2.2 they were pushed descending. In this version they are pushed as soon as an image in the image pyramid is created as shown in the Figure 44 below.

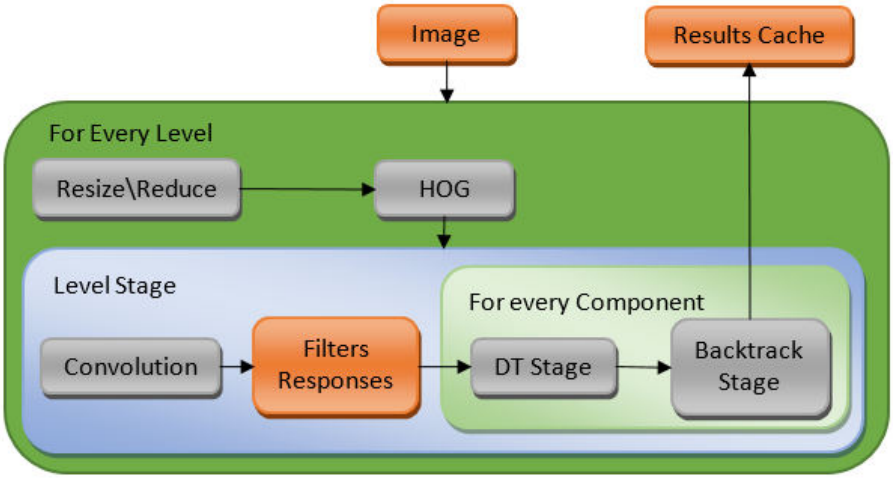


Figure 44 - TSM Algorithm v3.1 Execution Flow Diagram

What is obvious is that as the scaled images are pushed to the Level stage for detection there is no need for the features pyramid data structure to exist. The HOG procedure takes place just before the Convolution one and the features images are released just after. What is although needed is the temporary image pyramid to hold the scaled images longer than in the other versions. As described in the chapter 5.5, the features pyramid stage uses the scaled images as inputs in the Reduce procedure to create half copies of them. So the algorithm in this version cannot release scaled images from the image pyramid as long as it has not create their next interval ones. A tactic can be used here is the algorithm to create the next interval scaled image immediately in order to be able to release the ones used in the Level stage. This way the image pyramid can hold smaller sized images in order to reduce the maximum memory consumption that appears during the Level stage. Unfortunately, the maximum memory consumption of the TSM algorithm appears during the detection procedure of the first level of the pyramid where the unscaled resource image is used that is needed not only for the next interval scaled image but also for the rest scaled images of the first interval set of images in the pyramid. This means

that there is no way to avoid it as shown in Figure 44 above. By applying this execution flow the time results coming of are the following shown in Table 72 below.

Table 72 - TSM v3.1 Execution Time Comparison (%)						
	320x240	640x480	800x600	1024x768	1200x960	Average
Levels	18	23	25	27	28	
v1.3	-15.1	-13.2	-12.6	-12.3	-12.2	-13.1
v2.1	-15.7	-14.4	-13.7	-13.4	-13.3	-14.1
v2.2	-15.7	-14.2	-13.5	-13.5	-13.3	-14.1
v3.1	-22.7	-17.2	-16.0	-14.3	-14.0	-16.8

The version 3.1, as the Table 72 shows, is faster than the rest versions. This is probably caused by the memory cashing of the data used. The results of the scaling processing are probably saved in the cache memory and stay there as they are used immediately by the HOG procedure. As soon as the HOG processing is finished, its results data are used in the convolution procedure and the filters responses coming from this are used in the level detection stage for face detection. This sequential usage of data benefits the cashing process inside the CPU cache memory.

On the other hand the non-sequential order the pyramid levels are pushed to the detection procedure creates other problems that are not visible until this chapter but in the next chapters (ex. Chapter 9). Another problem also is that this version has definitely lost its relation with its parental algorithm and cannot be used at all for multi-scaled model in contrast to the other versions that can with only small changes.

The timeline profile of the version 3.1 of the algorithm is shown in the Diagram 40. It is visible that the sequence of the levels send for detection is not ascending but they follow the sequence of the Features Pyramid stage loop.

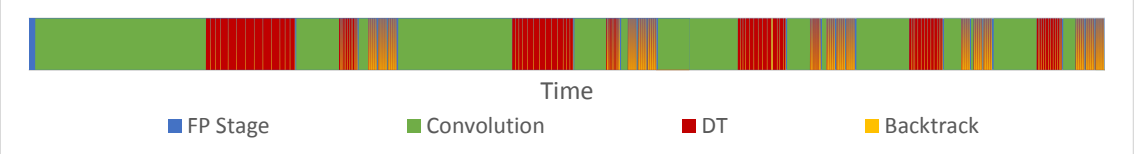


Diagram 40 - TSM Algorithm v3.1 Timeline Profile

At the next graph (Diagram 41) the memory profiling of the algorithm is shown.

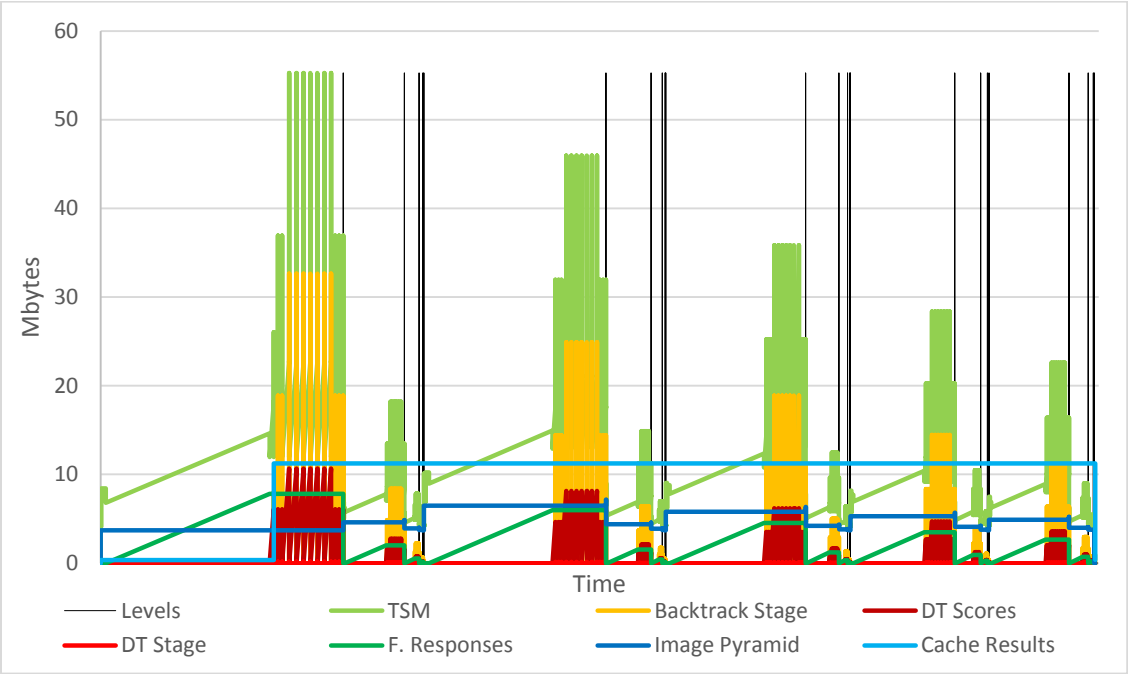


Diagram 41 - TSM Algorithm v3.1 Memory Profile

The Diagram 41, reveals that in the 3.1 version of the algorithm a new participant in the formation of the maximum memory appears. This participant is an image pyramid level. In the chapter 6.8 it was explained how larger is the image pyramid compared to the features one. In this version is inevitable the usage of the image pyramid instead of the features one and the cost of this change is paid in memory consumption. As referred in a previous paragraph this fact cannot be avoided as the image used in the first level of the pyramid is used as source not only for the Reduce procedure (next interval level) but also for the Resize one (All levels of the first interval set).

Table 73 - TSM v3.1 Maximum Memory Distribution (Compared to v1.2) (%)						
	320x240	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
TSM v2.3	14.5 Mb	55.3 Mb	85.7 Mb	139 Mb	217 Mb	
	-45.5	-44.9	-44.7	-44.5	-44.3	-44.8
FP Stage	0	0	0	0	0	0
Conv. Stage	0	0	0	0	0	0
DT Stage	0	0	0	0	0	0
Backtrack Stage	57.9	59.1	59.2	59.3	59.4	59.0
	-10.2	-10.5	-10.5	-10.6	-10.6	-10.5
Image Pyramid	6.35	6.67	6.72	6.77	6.80	6.66
	-12.7	-11.8	-11.6	-11.4	-11.3	-11.8
F. Responses	13.9	14.2	14.2	14.2	14.2	14.1

	-25.2	-25.3	-25.3	-25.3	-25.2	-25.3
DT Scores	18.9	19.3	19.4	19.4	19.4	19.3
Others	2.97	0.78	0.50	0.31	0.20	0.95
	-0.87	-0.92	-0.93	-0.94	-0.95	-0.92
Results Cache	+77.5	+20.3	+13.1	+8.06	+5.18	+24.8

At the Diagram 42Diagram 41 below it is visible the participation of the image pyramid at the formation of the maximum memory consumption value of the algorithm as just explained in the paragraph above.

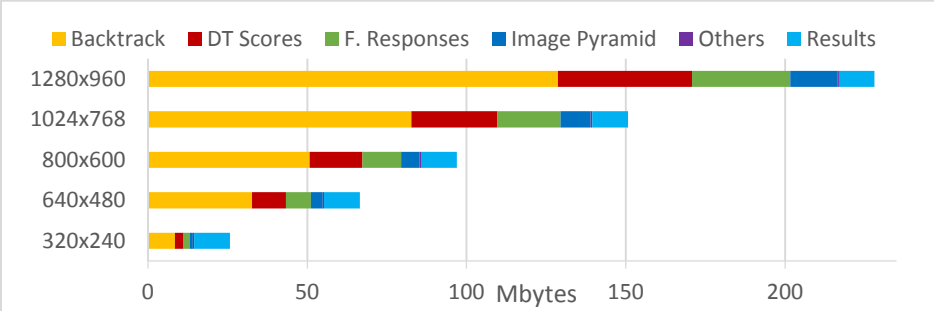


Diagram 42 - TSM v3.1 Maximum Memory Distribution per Image

This version has two basic disadvantages. The First and most significant is the inconsecutive order of forwarding the levels of the pyramid to the detection procedure. The second one, less significant or even not significant is the usage of Image pyramid that is more memory costly. Both these disadvantages are exceeded in the next subversion of the version 3.x of the TSM algorithm, presented in the next chapter (Chapter 6.20).

6.20. TSM Face Detector v3.2

In the version 3.1 of the TSM algorithm two main disadvantages are referred. The most significant disadvantage is the fact that the algorithm in this version is passing the pyramid levels to the detect stage in an inconsecutive series. This execution flow is repaired in this version so that the detection procedure can be applied in the pyramid's levels ascending to their size starting from the top level. To achieve this change the execution flow of the features pyramid stage, as presented in the chapter 5.5, changes and the algorithm calculates the levels of the image pyramid in a sequential way. This way demand to the algorithm to hold in the memory at least a set of interval of the image pyramid in order to be able to use it for the next one as shown in the Figure 45 below.

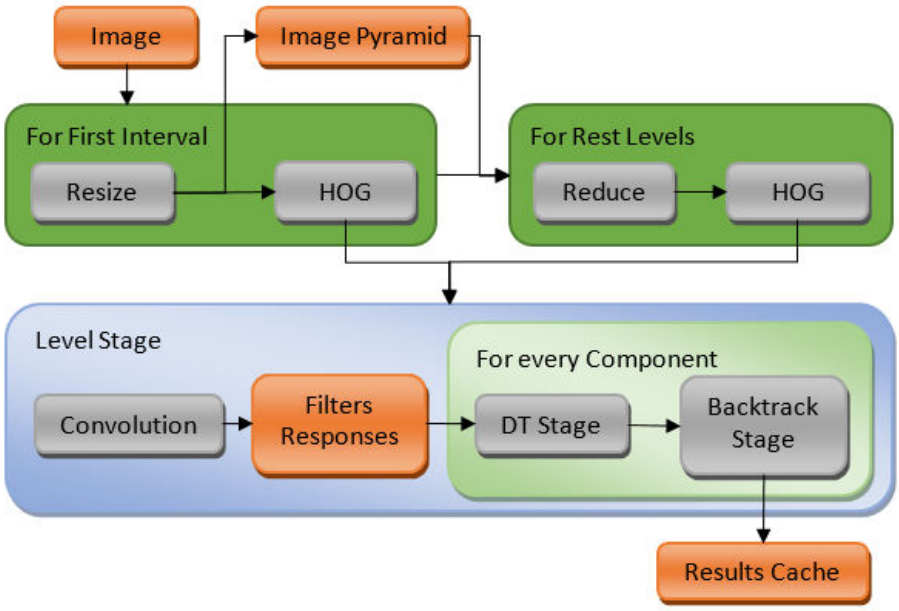


Figure 45 - TSM Algorithm v3.2 Execution Flow Diagram

The second disadvantage of the 3.1 version of the algorithm is the fact that the first level of the image pyramid joins the parts of data forming the algorithm maximum memory consumption. This level is the largest one coming from the source image and it is needed for the calculation of the rest scaled images in the first interval set of the image pyramid. In the chapter 6.6.1 a version of the Resize procedure using 8 bit images instead of 32 bit ones was introduced. This version can be used in the version 3.2 in customized to get an 8 bit image as input and return a 32 bit one as output. This way the source image can be used as an 8 bit image using only the 25% of the memory reducing the maximum memory consumption. In the Table 74 below the maximum memory consumption of this version is shown.

Table 74 - TSM v3.2 Maximum Memory Distribution (Compared to v1.2) (%)						
	320x240	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
TSM v3.2	13.8 Mb	52.5 Mb	81.3 Mb	132 Mb	206 Mb	
	-48.1	-47.6	-47.5	-47.3	-47.2	-47.5
FP Stage	0	0	0	0	0	0
Conv. Stage	0	0	0	0	0	0
DT Stage	0	0	0	0	0	0
Backtrack Stage	60.8	62.2	62.4	62.5	62.6	62.1
	-10.2	-10.5	-10.5	-10.6	-10.6	-10.5
Image Pyramid	1.67	1.76	1.77	1.78	1.79	1.75
	-11.9	-10.9	-10.7	-10.5	-10.4	-10.9
F. Responses	14.6	14.9	15.0	15.0	15.0	14.9

	-25.2	-25.3	-25.3	-25.3	-25.2	-25.3
DT Scores	19.9	20.3	20.4	20.4	20.4	20.3
Others	3.12	0.82	0.53	0.33	0.21	1.00
	-0.87	-0.92	-0.93	-0.94	-0.95	-0.92
Results Cache	+81.3	+21.4	+13.8	+8.49	+5.46	+26.1

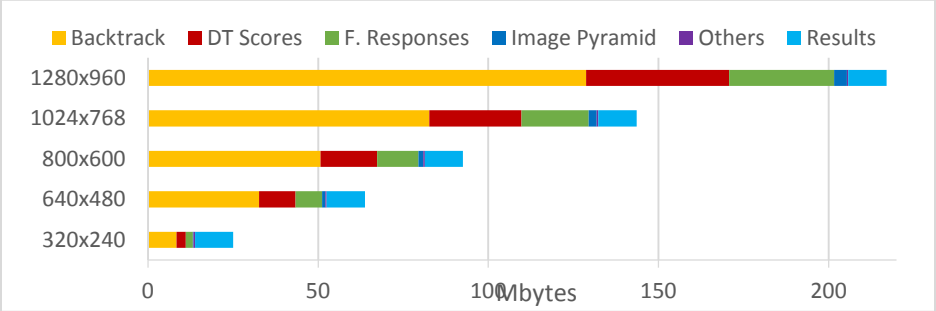


Diagram 43 - TSM v3.2 Maximum Memory Distribution per Image

As seen in the Diagram 43 above as the memory consumption of the TSM algorithm is reducing the most significant part of it is the results cache and the Backtrack stage temporary memory which is affected also by the detection results. This makes it clear that the number of detection within the image is significantly affecting its maximum memory consumption. In chapter 7.2 a patch that changes this attitude is presented. In the Diagram 44 below the memory profile of the version 3.2 of the TSM algorithm is presented.

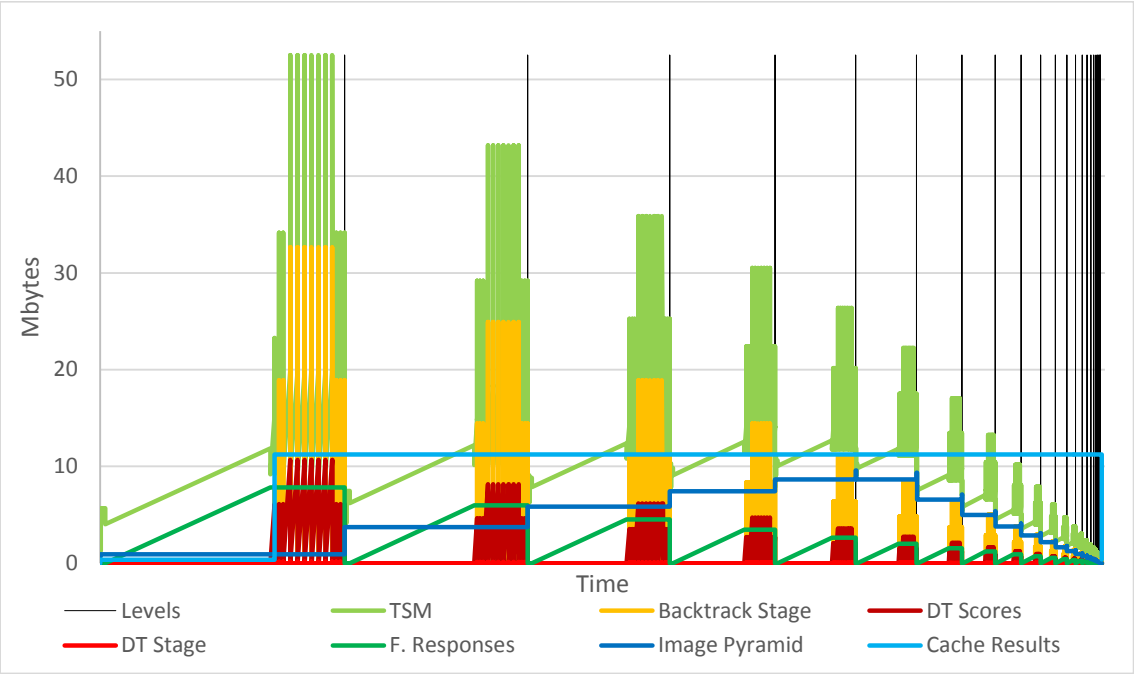


Diagram 44 - TSM Algorithm v3.2 Memory Profile

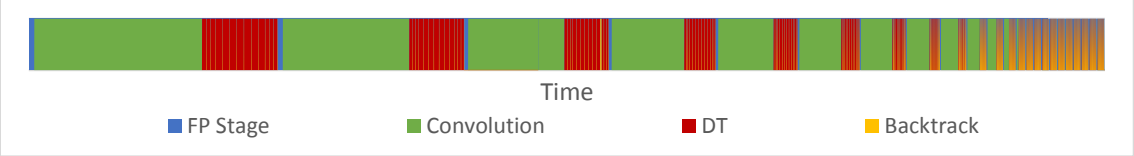


Diagram 45 - TSM Algorithm v3.2 Timeline Profile

As seen in the Diagram 44 and Diagram 45 the sequential flow of the pyramid's levels passing to the Detect stage is recovered. It is also visible in the Diagram 44 that the first level of the image pyramid consumes much lower memory from the next one even if its size is larger. That's because it is saved in the 8 bit format.

As far as the execution time needed for this version, as it is sensible, it has not changed relatively to the version 3.1 as shown in the Table 75 below.

Table 75 - TSM v3.2 Execution Time Comparison (%)						
	320x240	**640x480**	**800x600**	**1024x768**	**1200x960**	**Average**
Levels	18	23	25	27	28	
v1.2	-23,0	-17,2	-16,3	-14,4	-14,2	-17,0
v1.3	-9,22	-4,66	-4,17	-2,44	-2,25	-4,55
v2.1	-8,66	-3,36	-2,94	-1,23	-1,02	-3,44

v2.2	-8,60	-3,52	-3,16	-1,11	-0,96	-3,47
v3.1	-0,28	-0,09	-0,36	-0,13	-0,20	-0,21

6.21. TSM Face Detector All Versions

At this last chapter of Chapter 6 a quick summary about the different versions of the algorithm are appose. Firstly the main change between the versions 1.x and 2.x has to do with the execution flow of the algorithm. As seen in Figure 46 in version 1.x the execution flow of the Detect stage is using two nested loops. The outer loop is the one iterating between the different components (pose trees) and the inner one with the different levels of the features pyramid. The convolution process takes place inside the components loop. As the filters responses data are used by all components every calculation of filters responses occur at the first iteration of the component loop is required until the last iteration. This way the filters response data are calculated for all levels at the first iteration of the component loop as shown in the timeline Diagram 2 and used until the end of the component loop as shown in Diagram 8 in chapter 6.3.

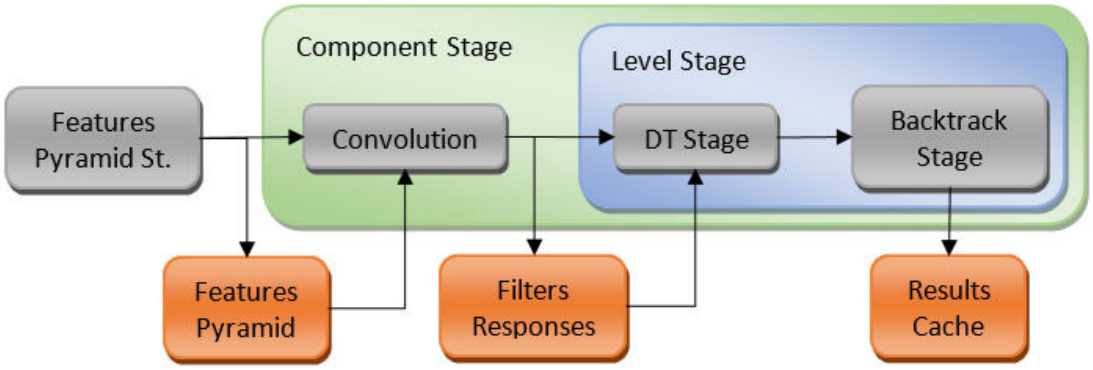


Figure 46 - TSM Algorithm v1.x Diagram

On the other hand in the execution flow of the version 2.x the two nested loops change sides. The levels loop becomes the outer loop and the components one the inner. The convolution process takes place inside the outer loop, the levels loop and calculates the filters responses of each level as shown in Figure 47. This way, as the face detector uses one scale models, after the end of the components loop the corresponding level's filters responses are note needed any more and can be released. This is visible also in the timeline profile of the version 2.1 in Diagram 36s (Chapter 6.17).

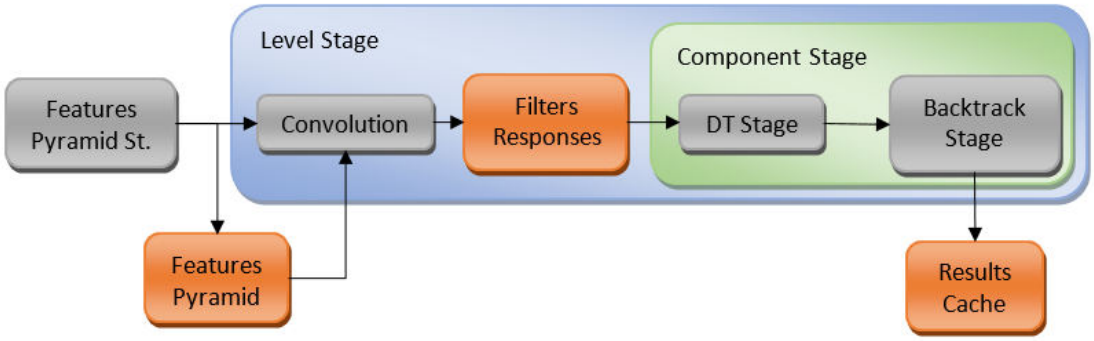


Figure 47 - TSM Algorithm v2.x Diagram

The algorithm versions 3.x are actually use the same execution flow with very small differences compared to the 2.x versions. This difference is that they merge the Features Pyramid stage with the Detect one and they detection procedure begins immediately when a features image is created as shown in the Figure 48 below.

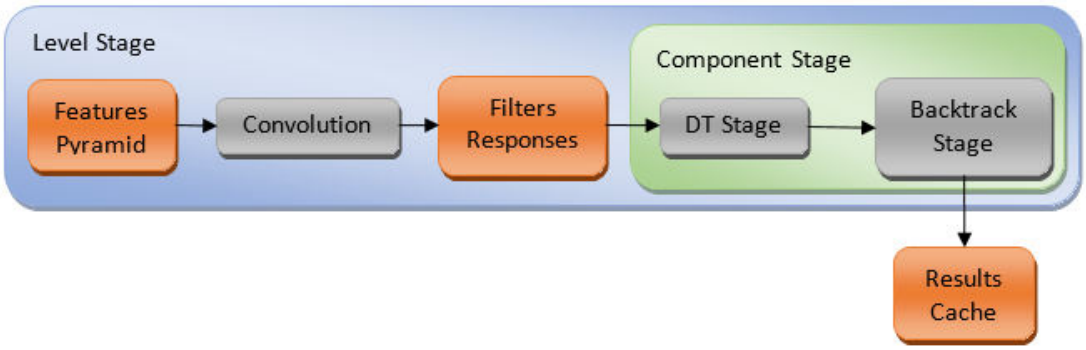


Figure 48 - TSM Algorithm v3.x Diagram

The comparison between all the version of the TSM algorithm can be appose as a summary of the algorithm version history. At the Diagram 46 below the time execution comparison is presented. As is visible the version 2.x is at least faster than the version 1.3. The greatest speed up improvements was achieved from the transition of the version 1.2 to 1.3. In the Table 76 the time execution ratio are shown.

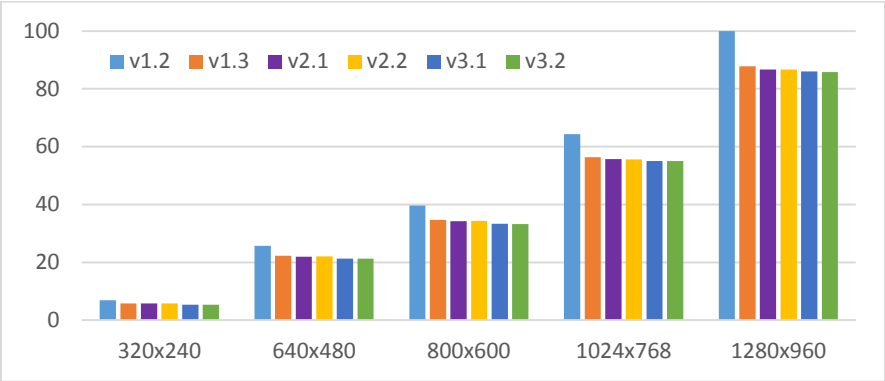


Diagram 46 - TSM Algorithm Execution Time Versions Comparison

Table 76 - TSM Algorithm All Versions Execution Time Comparison (%)						
Version	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
v1.2	100	100	100	100	100	100
v1.3	84.9	86.8	87.4	87.7	87.8	86.9
v2.1	84.3	85.6	86.3	86.6	86.7	85.9
v2.2	84.3	85.8	86.5	86.5	86.7	85.9
v3.1	77.3	82.8	84.0	85.7	86.0	83.2
v3.2	77.0	82.8	83.7	85.6	85.8	83.0

As far as the memory consumption the Diagram 47 below shows the differences between each version of the algorithm. As happened with the time execution the same happens as far as the memory consumption. The greatest improvement achieved at the version 1.3 of the algorithm. As shown in the Diagram 47 and in the Table 77 data, the memory consumption of the algorithm is the same in all the versions greater than the 1.3. It is obvious that in this version the memory consumption improvements have reached to ceil.

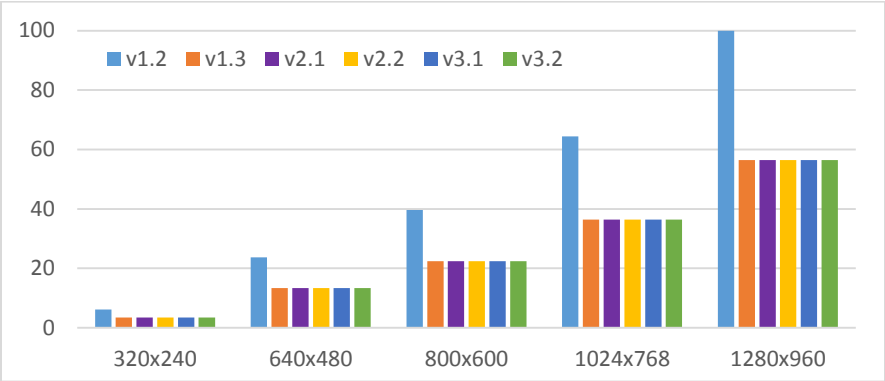


Diagram 47 - TSM Algorithm All Versions Memory Consumption Comparison

Table 77 - TSM Algorithm All Versions Memory Comparison (%)						
Version	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
v1.2	100	100	100	100	100	100
v1.3	56.4	56.2	56.5	56.5	56.5	56.4
v2.1	56.4	56.2	56.5	56.5	56.5	56.4
v2.2	56.4	56.2	56.5	56.5	56.5	56.4
v3.1	56.4	56.2	56.5	56.5	56.5	56.4
v3.2	56.4	56.2	56.5	56.5	56.5	56.4

At last, as long as the maximum memory consumption of the algorithm, the comparison graph (Diagram 48) and table (Table 78) have different indications. As seen the maximum memory consumption is finally reduced at less than its half for all 2.x versions of the algorithm reaching the minimum of 51% relatively to the original version 1.2.

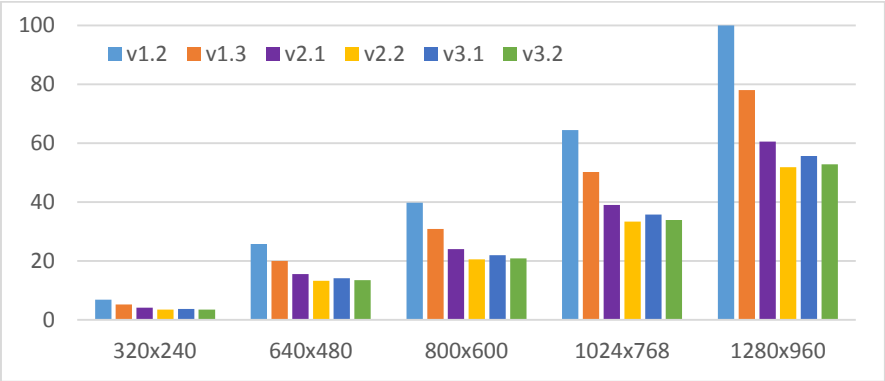


Diagram 48 - TSM Algorithm All Versions Maximum Memory Consumption Comparison

Table 78 - TSM Algorithm All Versions Maximum Memory Consumption Comparison (%)						
Version	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
v1.2	100	100	100	100	100	100
v1.3	77.1	77.7	77.8	78.0	78.1	77.7
v2.1	61.1	60.6	60.6	60.5	60.5	60.7
v2.2	51.0	51.5	51.6	51.7	51.9	51.5
v3.1	54.5	55.1	55.3	55.5	55.7	55.2
v3.2	51.9	52.4	52.5	52.7	52.8	52.5

Until this chapter many changes have been made inside the algorithms stages and procedures and the execution flow of the algorithm has been modified. The results of these changes have offer a reduction to the execution time by 17%, to the memory consumption by 43.6% and by 48.5% to the maximum memory consumption. Especially as far as the memory consumption the improvement is very significant. As shown in Table 79 below, the algorithm, using 1280x960 size images needs less than 512Mbytes of RAM to be executed instead of the 1Gbyte needed in the original version 1.1. This makes the algorithm available to be used in embedded system with low hardware resources!

Table 79 - TSM Algorithm All Versions Max Memory Requirements (Mbytes)					
Version	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**
v1.1 (double)	70	265	409	664	1030
v1.1	35	132	205	332	515
v1.2	27	100	155	251	390
v1.3	21	78	121	196	304
v2.1	16	61	94	152	236
v2.2	14	52	80	130	202
v3.1	15	55	86	139	217
v3.2	14	53	81	132	206

Comparing the algorithm's versions presented, two of them seems to be completed. The version 3.2 is the fastest one but the version 2.2 is the most memory saving. According to the Table 76 and Table 78, the difference between these two versions is small both in time and memory consumption. Although, the execution time is preferred instead of the maximum memory consumption as the last one difference does not seems to be critical at all in contrast to the execution time one, so the final version can be consider the 3.2.

All these changes are implementation changes that do not affect the algorithms creators design and accuracy. In the next chapters more modifications are appose that either change the creators design adding new techniques or affect the algorithm's accuracy.

7. TSM System Default Patches

In the chapter 9 some patches for the TSM algorithm are presented trying to make the algorithm a faster one. These patches though reduce the algorithm reliability and detection efficiency. For that reason these patches are called alternative patches. In this chapter two special patches are presented as they contribute to the memory and execution time improvement without affecting at all the algorithm detection performance. For that reason these two patches are called default patches and they are included in all the x.x.2 versions of the TSM algorithm.

7.1. Short Pyramid

In chapter 6.2 it was mentioned that the Face Detector algorithm is designed to detect faces in the size of 100 pixels high (50 pixels on 146 filters model). The image pyramid is used in order to detect larger faces by scaling the image and match the faces on that size. Any faces smaller than the 100 pixels high are not able to be detected.

The algorithm, as explained in chapter 4.2, uses histograms of oriented gradient in order to detect the existing faces by using its model filters. As explained in chapter 5.4, a HOG image is about four times smaller than the original it comes from. This means that a features canvas containing a face must be larger than 35 pixels.

In chapter 5.5 the mathematic type calculating the number of the image pyramid levels is referred. This type is the one shown in function (8) below and its results are shown in the Table 80. As seen in this table the last 11 levels of the image pyramid created have image height less than 100 pixels while the last six less than 50. This make it sensible that even if human faces are contained within these images, the algorithm is not able to detect them.

$$Levels_{pyramid} 1 + floor\left(\frac{\log\left(\frac{\min\left(Width_{image}, \text{Height}_{image} \right)}{5 \cdot sbin} \right)}{\log\left(2^{1/\text{int}erval} \right)} \right) \qquad (8)$$

Table 80 - Features Pyramid Level Images High Size													
Image Size	**Levels**	**Bottom 12 levels**											
320x240	18	105	91	80	69	60	53	46	40	35	30	27	23
640x480	23	105	91	80	69	60	53	46	40	35	30	27	23
800x600	25	99	87	75	66	57	50	44	38	33	29	25	22

1024x768	27	96	84	73	64	56	48	42	37	32	28	24	21
1280x960	28	105	91	80	69	60	53	46	40	35	30	27	23

By testing the algorithm without using these levels of the image pyramid in the detection process there was no affect in the detection results. By this conclusion there is no need of using this levels of the image pyramid that is having a small effect on the algorithms execution as it is explained in later.

According to the conclusions of the previous paragraph the levels of the images of the Table 80 should be as shown in the Table 81. As seen the number of levels is almost reduced to the half. These numbers comes from a new mathematic type that calculates the image pyramid levels until they get to a size not smaller than the limit of 100 pixels height. This mathematic type is the one shown in function (9). The $Height_{min}$ parameter is the minimum size of a detectable face which is 100 pixels for the 99 filters model, as explained before.

Table 81 - Short Pyramid Levels

Image Size	Model	
	99	**146**
320x240	7	12
640x480	12	17
800x600	13	18
1024x768	15	20
1280x960	17	22

$$Levels_{pyarmid} = \left\lfloor \log_2\left(\frac{Height_{image}}{Height_{\min}}\right) \cdot \text{interval} \right\rfloor \qquad (9)$$

By applying this change in the algorithm there is an impact to its execution time. By reducing the number of levels in the image and also the features pyramid the whole algorithm is affected. First of all, the Image Pyramid stage is speeded up as the less levels the image pyramid has the less scaled images have to be produced and of course less features images have to be created. The creation of HOG descriptors is a costly procedure as far as the time and memory usage. To continue, as the image pyramid is shorter, there are less features images to apply the model's filters that means less call of the convolution procedure. The convolution process is the most time consumer procedure as explained in chapter 6. At last, shorter feature pyramid means less levels for face detection (DT stage, Backtrack stage). This small change causes a wide impact to the whole algorithm.

On the other hand this change cause the reduction of the Image pyramid levels by rejecting its bottom levels. This means that this levels corresponds to the smallest features images. Small images consume few amount of time for their execution in the most parts of the algorithm as for example explained in chapter 6.9 where the convolution process is described. Even if the half of the image pyramid is rejected, the execution time cost that is saved by this change is much smaller than if one of the top levels of the image pyramid was rejected. All these claims are visible in Table 82 and in the Diagram 49.

Table 82 - Short Pyramid Patch Time Effect on TSM (%)						
	320x240	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
v2.2	-16.3	-4.45	-3.42	-1.94	-1.18	-5.45
v3.2	-14.2	-3.53	-2.64	-1.80	-0.91	-4.61

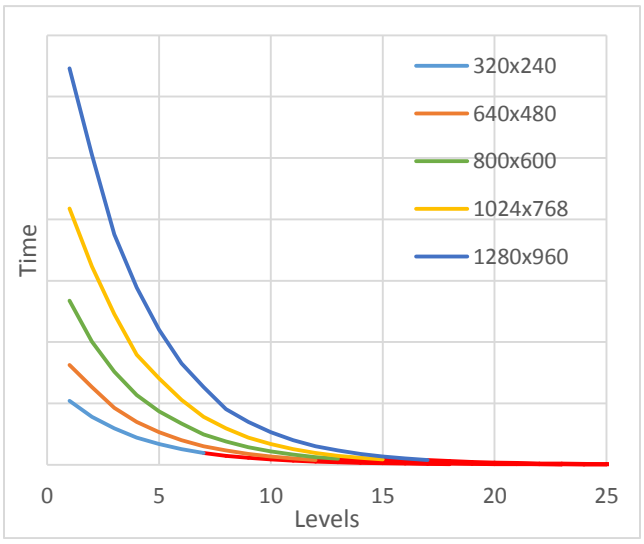


Diagram 49 - TSM Algorithm Execution Time per Level

As seen in the Table 82, even if the pyramid is reduced in its half the time reduction is not more than 4.5%. As the image size is getting larger the rejection of the highest levels of the pyramid seems to be insignificant. This is because as the image is getting larger the lower levels of the pyramid tend to be larger and this affects the execution time of the most stages of the algorithm exponentially as referred in the corresponding chapters (ex. Convolution, Chapter 6.9).

As far as the impact of this change to the memory consumption, the reduction is not so important. The temporary memory consumption is sensible reduced as the features pyramid levels are reduced. In addition, the maximum memory consumption is not expected to be reduced as it is clearly depended by the top level of the pyramid and its detection procedure (see Chapter 6.18 and 6.20). This patch in the algorithm's design is actually a time saving one and no changes relative to memory consumption are applied. In the next chapter (chapter 9.1) the "Find v2.0" patch is a memory consumption improvement one.

At last after removing all these levels of the pyramid, the relations of the functions *(4)* and *(6)* (Chapter 6.3) have to change, as they do not longer represent the real number of levels of the feature pyramid that contain high-score values. Testing the algorithm in the image sample referred in chapter 6.3, the new results are as presented in the Table 83 below. As sensible the high-score values per find data are still the same.

Table 83 - $Levels_{with\text{-}High\text{-}Scores}$ / $Levels_{Features\_Pyramid}$ (%)						
	99 Filters Model			146 Filter Models		
Samples	Max	Average	Min	Max	Average	Min
Top 10%	29.0	23.0	18.9	29.9	20.1	15.1
Top 20%	55.0	29.4	14.2	31.2	21.5	13.6
Top 50%	55.0	25.0	13.0	31.2	20.2	8.88
All (100%)	55.0	19.6	0.48	31.2	15.0	0.43

From the data contained in Table 83 the functions *(4)* and *(6)* (Chapter 6.3) has to be converted to the functions (10) and (11) as shown below.

99 Filters Model

$$Levels_{High-Scored} = Round\left(0.25 \cdot levels\right) \qquad (10)$$

146 Filters Model

$$Levels_{High-Scored} = Round\left(0.20 \cdot levels\right) \qquad (11)$$

By applying the Pyramid patch to the TSM algorithm the versions using it are changed to x.x.1. For example when this patch is used with the version 3.2 this version is now called the 3.2.1 one. This is useful when more patches and versions are applied or combined to one or more versions.

7.2. Find v2.0

In chapter 6.13 the Backtrack stage is described. The execution flow of the Backtrack stage starts form the Find procedure that discovers the high-scored values come from the DT stage and forwards them to the Backtrack procedure where the last one makes the landmark estimation. At last the NMS procedure is the one that selects the correct ones by rejecting the overlapping ones. The Backtrack procedure is the most time and memory consuming one in the Backtrack stage and unfortunately increases the algorithms maximum memory consumption at a notable amount. The reason that the Backtrack procedure is using this great amount of resources is that it uses a complex way to estimate the landmarks and a lot of memory to store the results. In addition to this the algorithm needs a large amount of results cache memory to store this great amount of results coming from the Backtrack stage. The memory needed for storing the results in the results cache memory is also increasing the maximum memory consumption of the algorithm.

As described in chapter 6.12 a face within an image produce a number of high-score values during the DT stage. From all those values only one is the top and it's the one used as the real detection result. The rest ones are considered as overlapping results. Overlapping results are produced around the top high-score value in the same level DT scores result image and in the

near levels images. In chapter 7 it is described that, after testing the algorithm along a series of testing images, the result was that a detected face produce high-score values at the 20% of the features pyramid levels with a mean number of high-scores of 80 pixels per Find procedure executed. From all those high-score values only one is the top that results to the final detection.

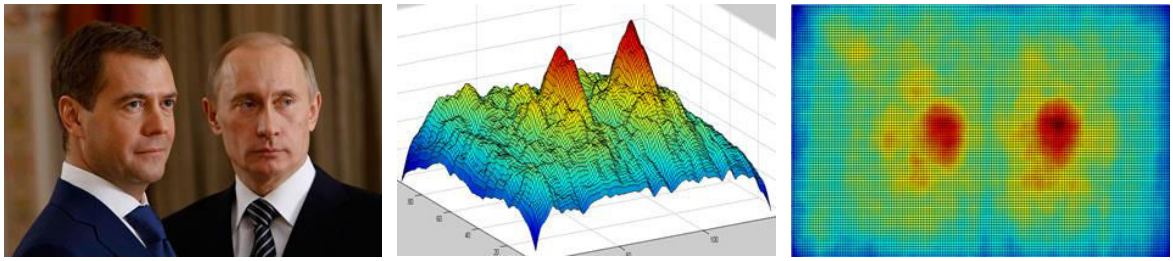


Figure 49 - Image DT Scores Array Example (Find Input)

For selecting the top high-score value that return the real face detection the algorithm is using the NMS procedure as described in chapter 5.10. The NMS procedure is used every time the Results cache is full in order to release space and at the end of the detection procedure in order to select the real results.

All these problems can be distinguish using a technique that rejects the overlapping results before they are forwarded to the Backtrack procedure. This way the workload of the Backtrack procedure can be greatly decreased in addition to the memory consumption reduction. Additionally, the results cache memory can be also abridged. The technique we propose for that purpose is a new implementation (version 2.0) of the Find procedure that would discover only the highest value of a high-value pixels neighborhood as shown in the Figure 50.

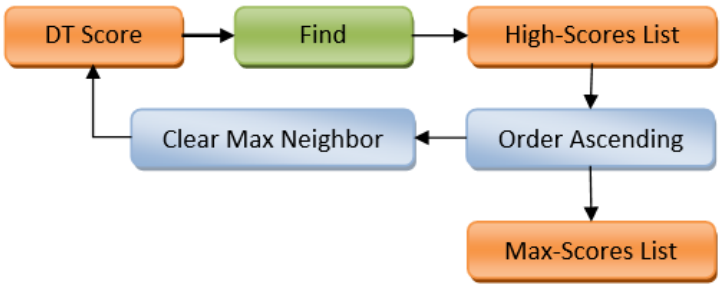


Figure 50 - Find v2.0 Procedure Diagram

As shown in the Find v2.0 procedure execution graph (Figure 50), the v1.0 Find procedure is used to discover high-score values. If High-score values are discovered the patch saves the highest one in a list and removes it and its neighbors from the DT scores table. Afterwards it calls again the v1.0 Find procedure and repeats the same procedure. The reason of repeating this procedure is because when there are more than one faces inside the image, more high-

scores values neighborhoods would exist. Graphically the impact of the Find procedure patch over the DT scores table is shown in Figure 51 below.

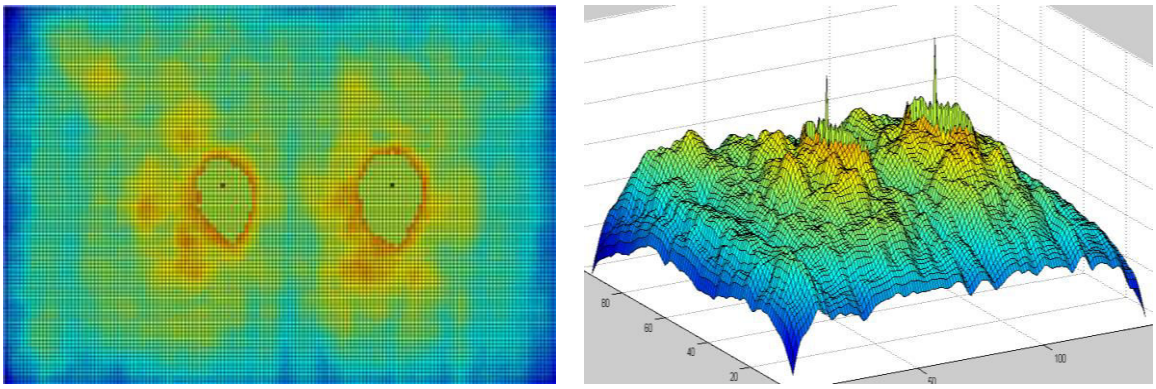


Figure 51 - Find v2.0 Results on the DT Score Array Example

The advantage of this Find procedure version is that it mostly discovers only one high-score value for every face detection. This is very important as only substantial high-score values are passed to the Backtrack one. This change create new conditions around the Backtrack stage that repeal the statistic results of the chapter 6.2.

In chapter 6.2 a presentation of the statistics results as far as the number of high-score values produced during the detection process is presented. This data come from the usage of the Find v1.0 procedure. By using the new version a new set of data comes on. Using the same image and applying the same experimental process the results using the new version of Find procedure are presented in the Table 84 below.

Table 84 - Find v2.0 $Pixels_{with\text{-}High\text{-}Scores}$ / ($Levels_{with\text{-}High\text{-}Scores}$ x Components)									
	v2.0			v1.0			Profit		
Samples	Max	Average	Min	Max	Average	Min	Max	Average	Min
	99 Filters Model								
Top 10%	5	1.6	1	611	169	1	-99.2	-99.1	0
Top 20%	5	1.5	1	611	128	1	-99.2	-98.8	0
Top 50%	5	1.4	1	611	103	1	-99.2	-98.6	0
All (100%)	5	1.3	1	611	79	1	-99.2	-98.4	0
	146 Filters Model								
Top 10%	5	1.2	1	343	116	1	-98.5	-99.0	0
Top 25%	5	1.1	1	343	91	1	-98.5	-98.8	0
Top 50%	5	1.1	1	343	70	1	-98.5	-98.4	0
All (100%)	5	1.1	1	343	53	1	-98.5	-97.9	0

In the Table 84 above the great effect of the version 2.0 of the Find procedure is shown. As seen in the right columns the average number of High-Score values is reduced for more than 98% compared to the 1.0 one. This is great decrement with many impacts on the whole Backtrack stage as presented in the next paragraphs.

In chapter 6.2 the number of levels with high-score values is estimated after testing the algorithm. By changing the Find procedure this number changed also at the top10, top20 and top50 samples (Table 85). On the other hand it's the same when all the samples are used. This difference is caused because, by the time the Find v2.0 procedure is used, every face within an image creates only one high-score value. In our sample images only one face appears within it so the sensible result would be one High-score value per Find procedure or none. What is shown in Table 84 above is that the testing results show even five high-score values to appear. This is because sometimes the DT score of a component might create multiple neighborhoods of high-score values in the area where face exists. This phenomenon appears usually at the levels close to the right one where the face is not yet in the right size (about 100 pixels high) to be detected.

Table 85 - Find v2.0 $Levels_{with\text{-}High\text{-}Scores}$ / $Levels_{Features\_Pyramid}$ (%)						
	Original Version			New Version		
Samples	Max	Average	Samples	Max	Average	Samples
	99 Filters Model					
Top 10%	18.9	14.3	12.1	23.4	11.8	0.3
Top 20%	28.6	17.2	9.2	23.4	12.7	0.3
Top 50%	28.6	14.8	7.4	28.6	13.7	0.3
All (100%)	28.6	11.6	0.3	28.6	11.6	0.3
	146 Filters Model					
Top 10%	21.5	15.8	6.1	21.5	16.1	12.9
Top 25%	24.0	14.5	3.7	24.0	16.8	10.7
Top 50%	24.0	14.8	3.7	24.0	15.7	7.2
All (100%)	24.0	11.8	0.3	24.0	11.8	0.3

Another great impact of this new version of the Find procedure is over the output results of the whole Backtrack stage. As the number of high-scored values is reduced to more than 98% the number of the Backtrack stage results is also reduced at the same percentage. This change implies the need of reducing the results cache memory. As referred in Chapter X the default result's cache memory is 10,000 detection results. Using the new Find procedure this number is sensible to be reduced by 98% less, to the size of 200. This change means that the maximum memory consumption can be reduced by a remarkable amount of memory.

On the Table 86 below the impact of the Find v2.0 procedure as far as the execution time of the algorithm and its stages is shown. As seen in this table the execution time consumption of the

whole Backtrack stage is greatly reduced due to the Backtrack procedure time consumption reduction despite the Find one increase. Although the impact of this reduction is tiny on the whole Face Detection algorithm's execution time as the Backtrack stage consumes only about the 0.25% of the algorithm execution time.

Table 86 - Find v2.0 Execution Time Impact on TSM v3.2.1 (%)						
	320x240	640x480	800x600	1024x768	1280x960	Average
Find	+77.3	+93.7	+92.9	+117	+103	+96.9
Backtrack	-96.7	-95.4	-94.7	-94.0	-92.5	-94.6
Back. Stage	-84.3	-87.3	-84.8	-78.5	-72.0	-81.4
NMS	-99.4	-93.9	-99.6	-96.0	-94.6	-96.7
TSM	-0.42	-0.24	-0.18	-0.11	-0.07	-0.20

On the other hand the impact of the new version (v2.0) of the Find procedure as far as the memory consumption is much larger than the time one. In the Table 87 below the temporary memory consumption reduction is shown. As seen in this table the Find procedure temporary memory consumption is increased but this incremental caused a huge reduction in the temporary memory consumption of the Backtrack and NMS procedure as also the Backtrack stage and the temporary results. This is a great achievement as the Backtrack stage and the temporary results data constitute a large part of the total temporary memory consumption. As shown in the Table 87 the TSM algorithm temporary memory consumption is actually reduced about 45% by the usage of the new version of the Find procedure.

Table 87 - Find v2.0 Impact on TSM v3.2.1 Memory Consumption (%)						
	320x240	640x480	800x600	1024x768	1280x960	Average
Find	+50.0	+50.0	+50.0	+50.0	+50.0	+50.0
Backtrack	-99.8	-99.7	-99.7	-99.7	-99.7	-99.7
Back. Stage	-97.1	-97.1	-97.1	-97.0	-97.0	-97.1
NMS	-98.0	-98.0	-98.0	-98.0	-98.0	-98.0
Results	-99.7	-99.7	-99.7	-99.7	-99.7	-99.7
TSM v3.2.1	-39.7	-43.8	-43.7	-47.7	-47.5	-44.5

As far as the maximum memory consumption, as seen in **Table 88** below, is reduced by about 61.5% as a result of the Backtrack stage 99% reduction. The Backtrack stage was one of the main participants at the maximum memory consumption formation and limiting its memory consumption the whole algorithms maximum memory consumption is affected. In addition to that the Results cache size reduction and the precocious rejection of the overlapping detection cause also a reduction of the impact of it over the maximum memory consumption. As seen in the table, the analogous of the default size result cache can cause about 1% incremental on the algorithm's maximum memory consumption.

Table 88 - Find v2.0 Maximum Memory Consumption Impact on TSM v3.2.1 (%)						
	320x240	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
TSM v3.2.1	5.47 Mb	20.1 Mb	31.0 Mb	50.2 Mb	78.0 Mb	
	-60.4	-61.7	-61.9	-62.0	-62.1	-61.6
Conv. Stage	0	0	0	0	0	0
DT Stage	0	0	0	0	0	0
Backtrack Stage	0.90	1.18	1.19	1.20	1.20	1.13
	-99.4	-99.3	-99.3	-99.3	-99.3	-99.3
F. Responses	36.8	39.0	39.3	39.5	39.6	38.8
DT Score	50.2	53.1	53.5	53.8	54.0	52.9
Image Pyramid	4.2	4.6	4.6	4.7	4.7	4.6
Others	7.88	2.15	1.39	0.86	0.55	2.57
Results Cache (200)	2.32	1.12	0.73	0.45	0.29	0.98
	-98.9	-98.0	-98.0	-98.0	-98.0	-98.2

As is visible in the Diagram 50 below the main modulators of the algorithm's maximum memory are the Filters Responses and the DT scores. These two data structure hold about the 91.5% of the algorithm's maximum memory consumption (the 96% on the 2.2.2 version!). These two data structure are very critical and their data cannot be reduced. The version 2.0 of the Find procedure has managed to reach the maximum memory consumption to a floor with no ability for further significant reduction.

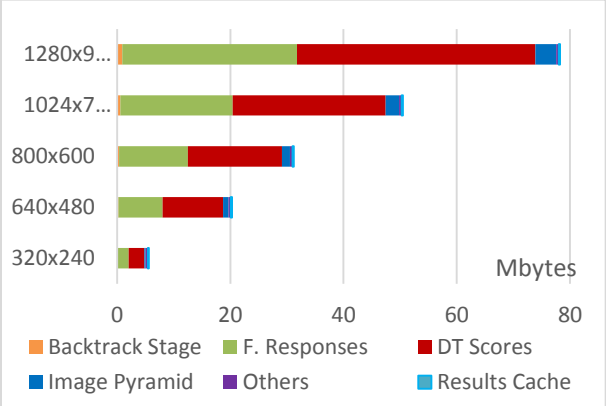


Diagram 50 - TSM v3.2.2 Maximum Memory Consumption per Image

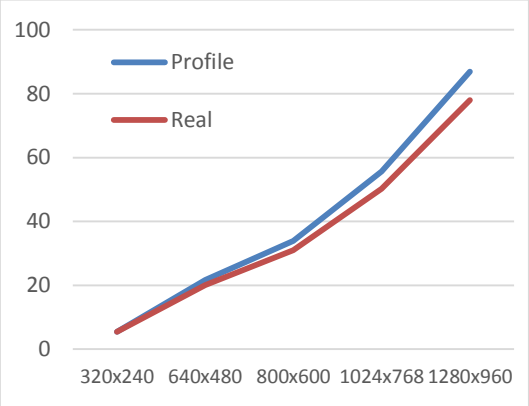


Diagram 51 - TSM v3.2.2 Maximum Memory Profiling

By the time the maximum memory consumption of the TSM algorithm is formed almost at all by predictable known parameters it can be easily predicted with a simple function as the function (12). This function calculates in a very simple mode the maximum memory consumption using only two parameters, the image width and height. The results of the function (12) compared to the real ones are presented in Diagram 52.

$$Mem_{Max} = Heigth(Image) \cdot Width(Image) \cdot \left(3 + \frac{Model.filters + 136}{16}\right) \quad (12)$$

In the Diagram 52 below a graphical view of the TSM algorithm memory profile is shown. As seen, the Results Cache and the Backtrack stage lines are now almost at the bottom of the diagram.

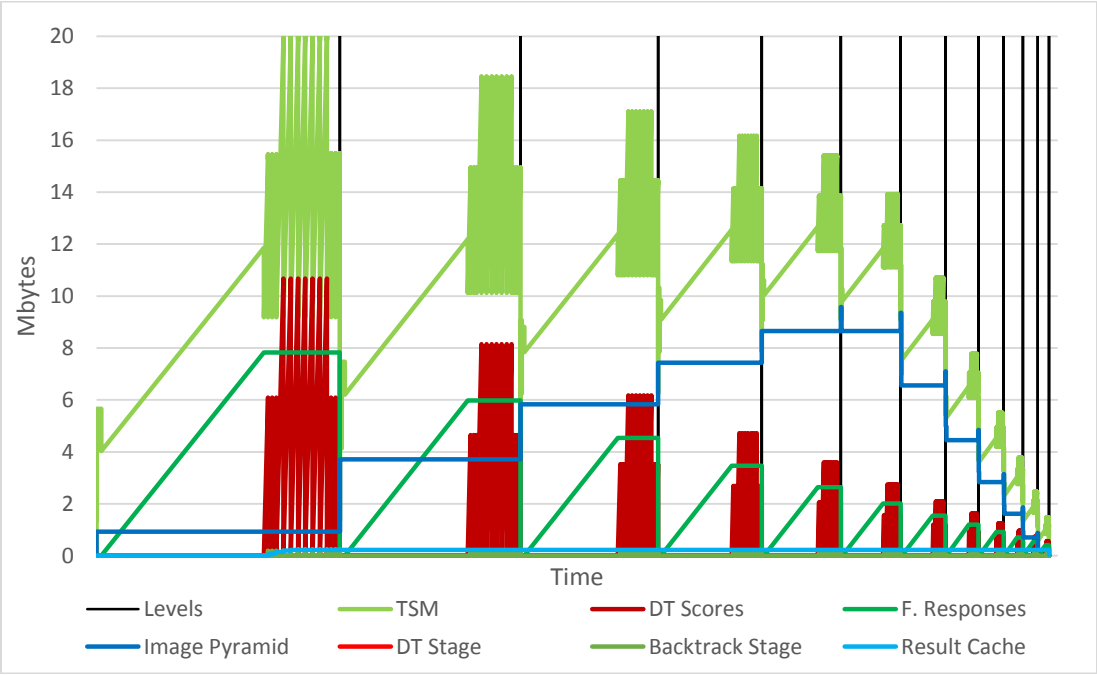


Diagram 52 - TSM Algorithm v3.2.2 Memory Profile

By applying the Find v2.0 procedure to the version 2.2.1 of the algorithm its maximum memory consumption was also reduced. In the **Table 89** below its new maximum memory consumption is presented. As seen in the "Mbytes" lines both versions need less than 100 Megabytes of memory to run. In the "VS 1.2" lines it is visible that after all the changes applied to the TSM algorithm until these two versions are created, the maximum memory consumption of the algorithm is reduced about 80%. This is a great reduction. This makes also the algorithm able to run in very low resources hardware!

Table 89 - TSM Basic Versions Maximum Memory Consumption						
		320x240	**640x480**	**800x600**	**1024x768**	**1280x960**
v2.2.2	Mbytes	5.25 Mb	19.2 Mb	29.5 Mb	47.9 Mb	74.3 Mb
	Vs 1.2	-80.3%	-80.9%	-80.9%	-80.9%	-80.9%
v3.2.2	Mbytes	5.47 Mb	20.1 Mb	31.0 Mb	50.2 Mb	78.0 Mb
	Vs 1.2	-79.5%	-80.0%	-80.0%	-80.0%	-80.0%

The Find patch importance is very high because it totally released the algorithm from a very high memory consumption. In the single thread versions this might not look so significant but as is presented in later chapters (chapter 8) where parallelized versions are introduced, the absence of this patch would probably cause a lot of problems.

8. Multi-Threading Implementation

In this chapter the conversion of the TSM algorithm implementation of a single thread one to a multi-threading one using the OMP library [34]. Every stage and procedure is tested using multiple CPU cores and the best combination and distribution of cores are finally used to succeed the best execution time speed and memory consumption. To this task the modern hardware boards' available resources are considered. The versions presented in chapter 6 are tested in order to discover the most efficient one when multithreading technology is used and at last one more version of the algorithm is presented in chapter 8.9.3, that is designed totally for multiple cores CPU.

8.1. Features Pyramid

The Features Pyramid stage consumes a small part of the whole algorithms execution time but it precedes the Detect one. This means that while the Features Pyramid stage is executed the Detect one waits for it and all the hardware resources are available to be used. This fact allows the use of any of the hardware resources in order to speed it up and abridge the Detect stage execution. For that reason the usage of the OMP technology is applied to the Features Pyramid stage in order to take advantage of it. The OMP technology is applied to the three main procedures of this stage, the Resize, the Reduce and the HOG one and also at the whole stage. Of course, as the number of CPUs in the hardware is limited there are two types of using the multiprocessors technology. Either focus it on one process at a time or share it around multiple procedures. Both tactics were tested.

8.1.1.1st Tactic

The first tactic is the one where the multiprocessors technology is focused on every procedure in order to speed it up individually as shown in Figure 52 below. The parallelization efficiency of every procedure contained in the Features Pyramid stage is explained in previous chapters.

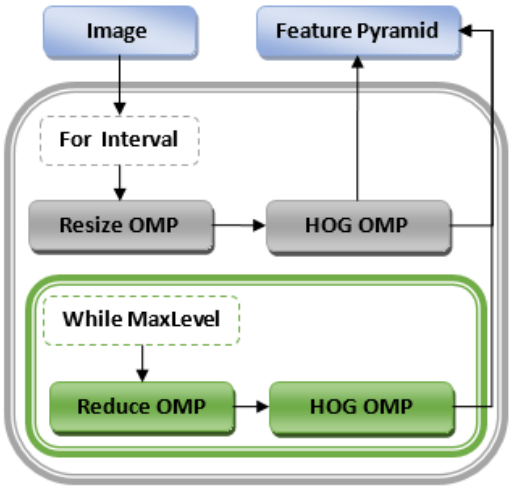


Figure 52 - Features Pyramid Stage OMP Diagram - 1st Tactic

By applying this tactic in the Features Pyramid stage the following results come on as shown in Table 90.

Table 90 - FP Stage OMP Execution Time - 1st Tactic (%)						
CPUs	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Speedup**
2	55.9	61.1	58.5	64.8	62.0	x1.7
3	42.7	46.4	43.1	46.7	43.2	x2.3
4	38.4	38.6	36.7	38.8	36.6	x2.6
5	32.7	36.2	33.2	36.8	34.2	x2.9
6	35.6	34.0	34.4	35.8	33.1	x2.9
7	41.1	33.5	35.9	33.5	30.1	x2.9
8	43.7	33.3	35.8	31.1	29.7	x2.9

The two graphs below shows the time consumption of the Features Pyramid stage according to the CPU cores used (Diagram 53) and its speedup efficiency (Diagram 54). In the Diagram 53 is visible that the speedup of this tactic is image size independent and that the speedup is gained by the use of more cores is reducing. This is also visible by the Diagram 54 where the efficiency of the number of CPU cores used is decreasing as more cores are used. According to the Diagram 53 the usage of 3-4 CPU cores added make the Feature Pyramid stage much faster when any additional cores does not offer any significant acceleration of the procedure.

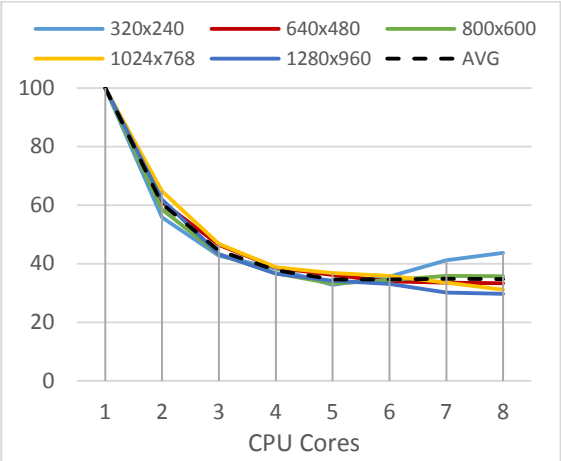


Diagram 53 - FP Stage OMP Execution Time (1st Tactic)

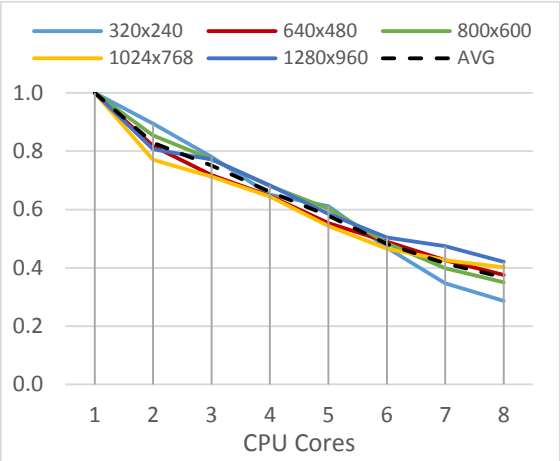


Diagram 54 - FP Stage OMP Execution Time Efficiency (1st Tactic)

As far as the memory consumption of this tactic is actually almost the same when one core only is used. This is because the memory consumption of the parallelized versions of the Resize, Reduce and HOG procedures is insignificant.

8.1.2.2nd tactic

The second parallelization tactic is the one where multiple procedures where shared in multiple processors cores as shown in Figure 53 below.

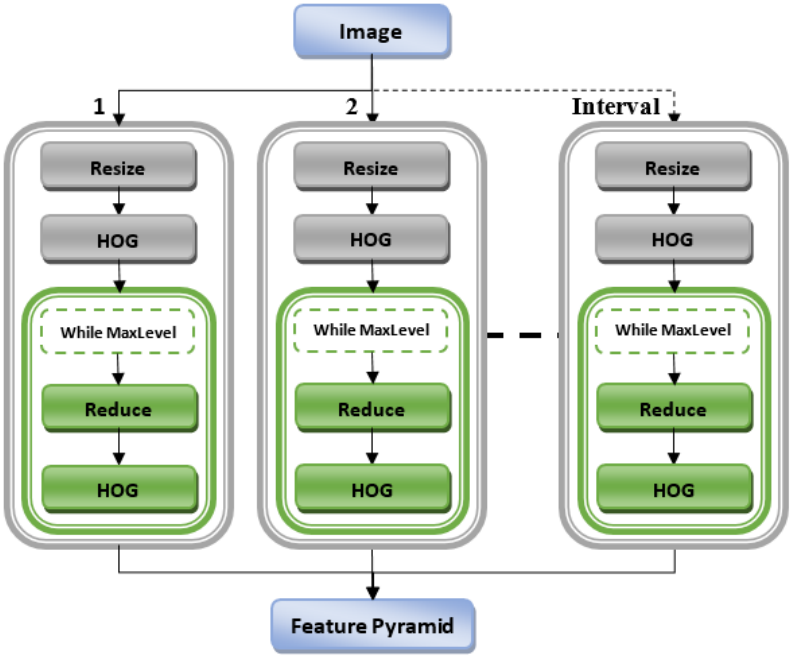


Figure 53 - Features Pyramid Stage OMP Diagram - 2nd Tactic

Using this tactic the results for the whole Features Pyramid stage is improved according to the single version of the algorithm. According to the Table 91 the most efficient results comes when the hardware can support a parallelization of five CPU cores. Using more cores does not offer better results and that is because the number of five cores is equal to the value of the interval parameter. Any other CPU cores more than the five cores are not used by this tactic and stays idle. This is not a disadvantage for this tactic because the idle cores can probably be used in nested parallelization as explained later.

Table 91 - FP Stage OMP Execution Time - 2nd Tactic (%)						
CPUs	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Speedup**
2	77.4	76.2	76.3	74.5	72.0	x1.3
3	60.4	55.7	53.5	54.0	51.8	x1.8
4	60.7	56.1	53.8	53.9	52.4	x1.8
5	34.0	30.6	29.9	31.2	29.5	x3.2
6	34.4	30.2	30.0	31.3	29.8	x3.2
7	32.4	30.5	30.0	30.3	30.1	x3.3
8	32.8	32.4	29.9	30.6	29.0	x3.2

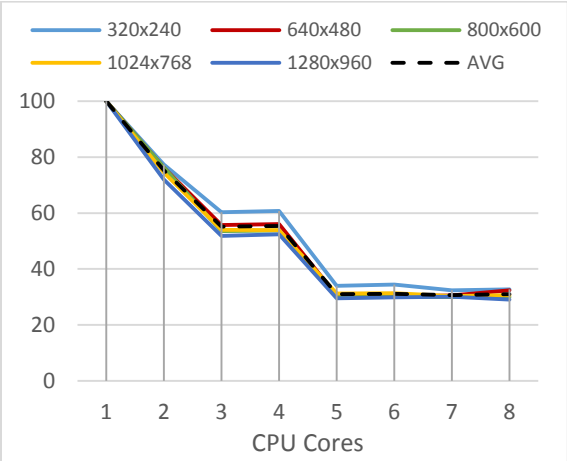


Diagram 55 - FP Stage OMP Execution Time (2^{nd} Tactic)

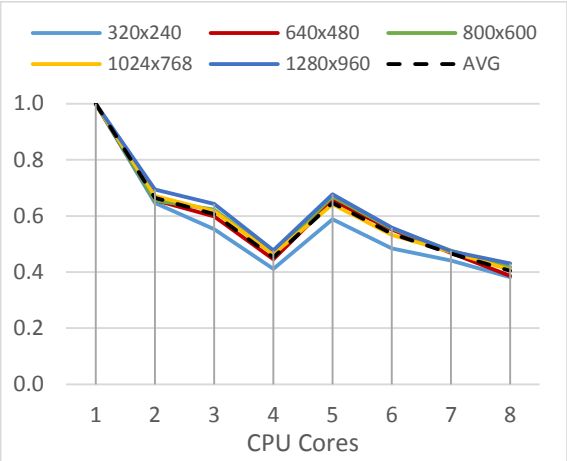


Diagram 56 - FP Stage OMP Execution Time Efficiency (2^{nd} Tactic)

In the Diagram 55 above it is visible that the usage of more than five CPU cores is useless as explained in the previous paragraph. What seems strange is the path that the time curves when four CPU cores are used. At this point is important to be explained that on this tactic the time speedup is achieved by the reduction of the features pyramid stage loop iterations. On the single core version this loop iterates for «interval» times. When used two cores the number of loop iterations are the half, etc. The value of the «Interval» variable is 5 on the algorithm, so when used three and four cores of the CPU the number of iterations of the stage's loop is on both cases two! This is why the Features Pyramid stage does not gain any speedup.

As far as the memory consumption of this tactic it is obvious that it is not the same as in the tactic 1. While multiple thread execute different procedures inside the Features Pyramid stage multiple data are calculated and created simultaneously. Looking at the Figure 53 above it is obvious that a simultaneous creation of the image pyramid and the features pyramid would allocate a great amount of memory that would also increase the stages maximum memory consumption that could affect the whole algorithm's execution time. The Features pyramid stage maximum memory consumption is reached while the algorithm is in the first Reduce procedure as by that time the first level of the image pyramid is in use and cannot be released and the first level of the Features Pyramid is already produced (function (13)). This amount of memory can be multiplied by the number of the CPU cores used estimating the differences in the image sizes are used (function (14)).

$$FPstage_{\max} = I.Pyramid[0] + F.Pyramid[0] + I.Pyramid[\text{int}\, erval] + 0.5 \cdot I.Pyramid[0] \quad (13)$$

$$FPstage_{\max} = \sum_{t=cores}^{t=0} \frac{2}{3} I.Pyramid[t] + F.Pyramid[t] + I.Pyramid[t + \text{int}\, erval] \quad (14)$$

In the function (13) expression the $0.5 \cdot I.Pyramid[0]$ parameter, represents the Reduce procedure temporary memory. By these functions results the Table 92 presents the amount of memory needed for the Features Pyramid stage depending on the number of CPU cores used.

Table 92 - FP Stage OMP 2nd Tactic Max Memory (Mbytes)						
CPUs	320x240	640x480	800x600	1024x768	1280x960	+%
1	2.3	9.1	14	23	36	
2	4.1	16	25	41	63	+76%
3	5.5	21	33	54	84	+134%
4	6.6	25	39	64	100	+178%
5	7.4	29	44	72	112	+212%
TSM	5.3	19	30	48	74	

As seen in the Table 92 the features pyramid stage maximum memory consumption is increasing the same percentage for all image sizes. This comes from the stable ratio between the image and features pyramid levels sizes. In the Features Pyramid stage all the data and procedures all image size dependent and this creates this stable ratio. While the features pyramid stage maximum memory consumption is extremely increasing when using multiple CPU cores, it seems to affect the whole algorithms maximum memory consumption (TSM line). This means that the second's tactic memory consumption increment should be considerable before used.

By comparing these two tactics it is obvious that the most suitable is the 1st one as it is the fastest one, more efficient and it does not affect the algorithm maximum memory consumption. The 2nd parallelization tactic is only faster when more than 1 CPU is used in the hardware resource and the speedup it gains is just a little better than the one the 1st tactic offers. As seen in the diagrams below the 1st parallelization tactic is succeeding the best results both in execution time and the efficiency on 1 CPU hardware resource.

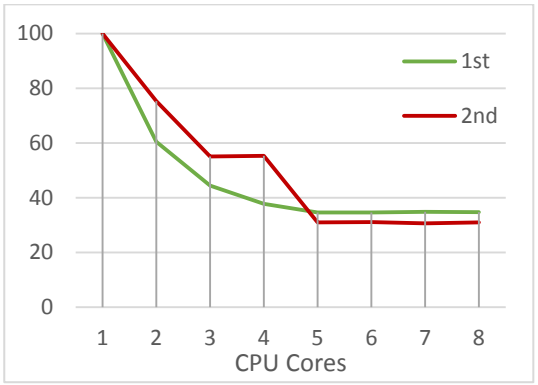


Diagram 57 - FP Stage OMP Execution Time (All Tactics)

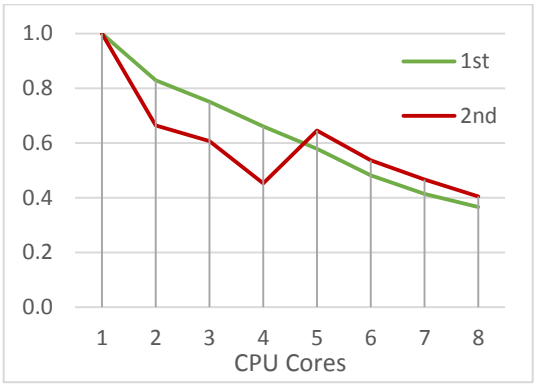


Diagram 58 - FP Stage OMP Execution Time Efficiency (All Tactics)

8.2. Resize

The Resize procedure uses less than 0.70% of the whole algorithm execution. Despite that it is a tiny part of the algorithm it is a part of the Features Pyramid stage that precede the Detect one that is the main time consumer. The fact that it precedes makes it desirable to speed up this process in order to abridge the detect stage execution. On the other hand at the features pyramid stage all the hardware resources are available

In the Table 93 below the Resize procedure's time consumption is not stably decreasing for all image sizes. As is visible also in the Diagram 59 below, the Resize procedure is reducing its execution time rapidly until the fourth CPU core and by that time it starts an unstable reaction to the CPU cores added. This instability is not unique for all image sizes but follows different attitude in each of them. This fact makes the Resize procedure unsafe and unreliable for used for more than 4 CPU cores.

Table 93 - Resize Procedure OMP Execution Time (%)						
CPUs	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Speedup**
2	73.4	87.9	75.9	87.2	80.8	x1.2
3	65.7	80.9	64.5	70.4	60.4	x1.5
4	68.5	69.1	55.3	57.0	51.7	x1.7
5	55.1	70.9	53.1	59.8	50.9	x1.7
6	51.9	70.1	56.0	60.4	51.2	x1.7
7	52.3	67.3	65.6	57.9	45.6	x1.8
8	56.6	68.1	64.6	51.6	44.8	x1.8

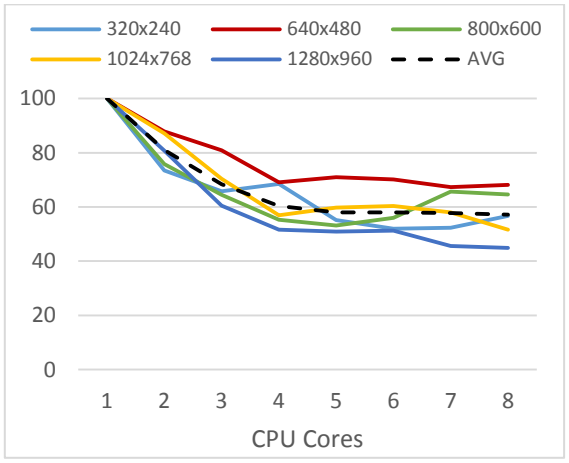


Diagram 59 - Resize Procedure OMP Execution Time

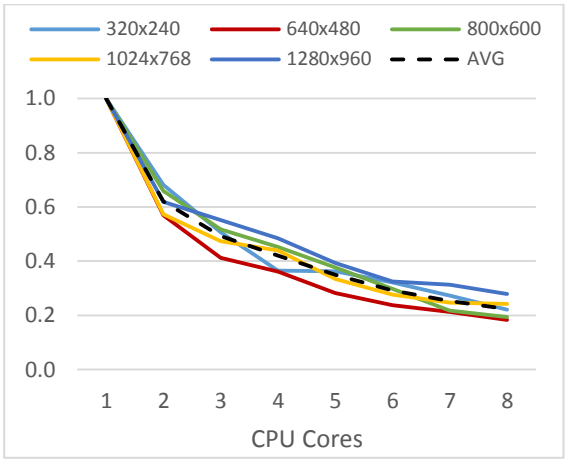


Diagram 60 - Resize Procedure OMP Execution Time Efficiency

In the Diagram 60 above the CPU cores usage efficiency is represented. As is visible the Resize procedure parallelization efficiency is not very good. It seems that using many cores on that procedure is actually speeds up its execution time but this speedup is not proportional to the cores sacrificed on it.

As shown in the diagrams the best number of CPU cores to be used for this procedure is up to three cores. The decision as long as the number of cores to be offered for this procedure is complicate because there may be multiple ways of doing that according to the global strategy used for the Feature Pyramid stage.

8.3. Reduce

The Reduce procedure is open to parallelism using multithreading (OMP) as it contains very simple loops that can handle parallelism. Although looking the whole algorithm, Reduce procedure takes place in a very small part of it so that it would be preferable to spend resources to more significant stages of the algorithm. What is very important though is that the Reduce procedure is a part of the features pyramid module that is necessary for the detection to start. For this reason is important to accelerate this stage's process in order to shorten the detection process beginning. We have all the hardware resources available while the detection stage is disabled. In the Table 94 below the time consumption results after testing the Reduce procedure using a multicore CPU is shown.

Table 94 - Reduce Procedure OMP Execution Time (%)

CPUs	320x240	640x480	800x600	1024x768	1280x960	Speedup
2	63.9	65.0	70.0	68.1	67.4	x1.5
3	35.9	36.8	38.3	38.4	37.0	x2.7
4	36.4	36.9	38.0	37.2	36.6	x2.7
5	36.0	36.9	36.9	37.6	36.5	x2.7
6	36.6	36.7	39.9	37.6	36.3	x2.7
7	36.2	36.3	39.4	37.6	36.3	x2.7
8	36.5	36.8	36.9	38.0	37.0	x2.7

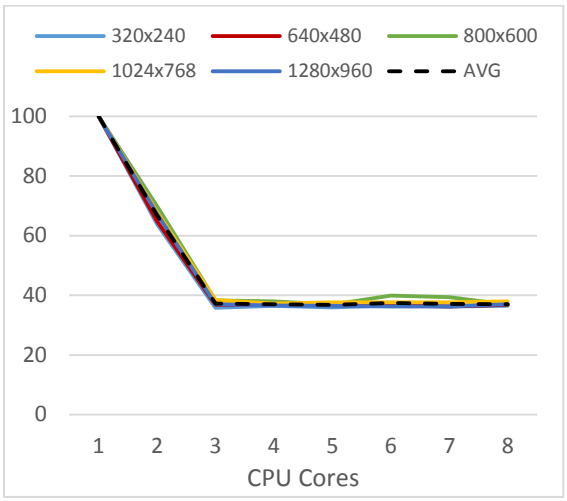


Diagram 61 - Resize Procedure OMP Execution Time

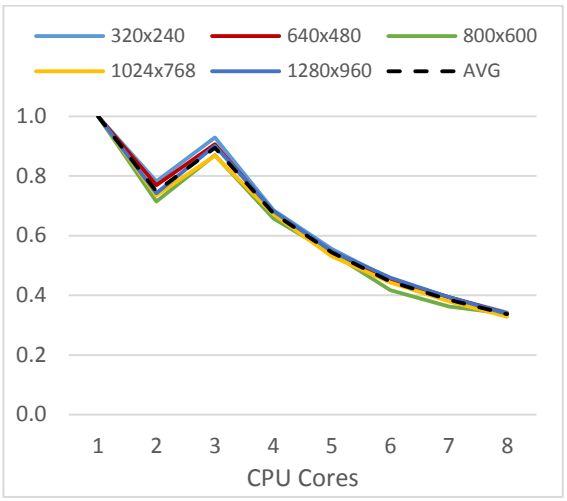


Diagram 62 - Resize Procedure OMP Execution Time Efficiency

As seen in the Diagram 61 above the Reduce procedure's execution time is greatly reduced until the usage of the third CPU core. By the fourth one and upper no more speedup appears. This is also visible in the Diagram 62 where the CPU cores usage efficiency is stably decreased when using more than three CPU cores. As also seen the Reduce procedure time speedup is accurate similar corresponding to the image size. As seen in the Diagram 61 and Diagram 62 the best number of CPU cores to be used is about two or three cores. It is worth to remind that the Reduce procedure is a small part of the Features Pyramid and is more complicated how the CPU cores are about to be shared as other procedure may need the more.

8.4. HOG

The HOG procedure is the one that creates the Histogram of Oriented gradients descriptor described in chapter 5.4. This procedure is open to parallelism using multithreading (OMP) as it contains loops that can handle parallelism. Looking the whole algorithm, the HOG procedure is the third most time consuming part of it even if it hold only a small percentage of the whole algorithm execution time. It is very significant to reduce its execution time as the feature images it creates are the input data to the detection procedure and to be accurate to the Convolution stage which is the greatest time consumer of the TSM algorithm. The execution time of the HOG procedure when parallelism is applied on it is shown in the Table 95 below.

Table 95 - HOG Procedure OMP Execution Time (%)						
CPUs	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Speedup**
2	56.2	55.2	52.7	53.1	52.7	x1.9
3	37.5	37.0	36.0	35.5	37.8	x2.7
4	30.3	28.6	27.5	28.4	29.3	x3.5

5	24.0	24.1	23.7	25.0	23.9	x4.1
6	21.7	21.4	20.4	20.9	22.3	x4.7
7	19.0	18.2	17.8	18.2	18.2	x5.5
8	17.4	16.9	15.1	15.6	16.0	x6.2

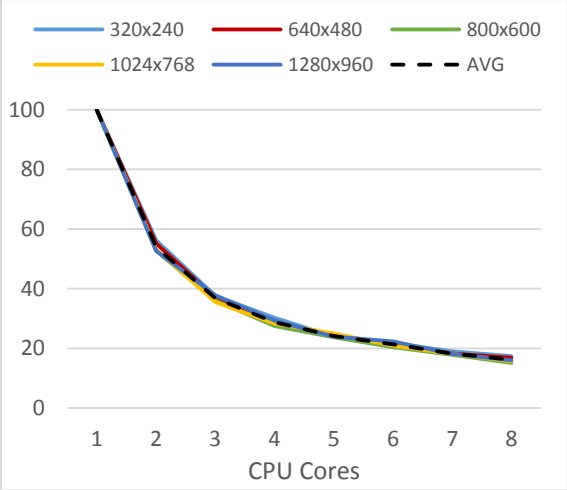


Diagram 63 - HOG Procedure OMP Execution Time

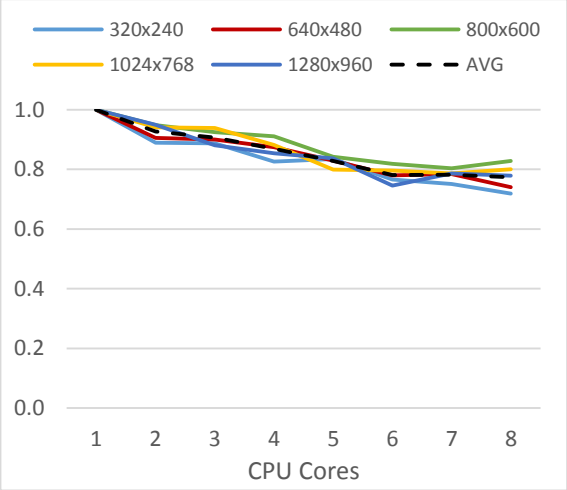


Diagram 64 - HOG Procedure OMP Execution Time Efficiency

As seen in the Table 95 and also in Diagram 64 the HOG procedure is more efficient to the parallelism. The more CPU cores used the more the execution time is decreased. There is no limit to the number of CPU cores used. The HOG procedure is the most time consuming procedure in the Features Pyramid stage and probably is better if the majority of the CPU cores are going to be available for this procedure when the v2.x is used.

As seen in the Diagram 64 the larger the image is the more efficient is the usage of multicore CPUs. It is also visible that the efficiency of using more CPU cores in the HOG procedure is stable and pleasantly good as even when using eight CPU cores, the efficiency does not fall under the 75%.

8.5. Convolution

The convolution procedure is the most important one of the algorithm as uses the most resources of the hardware and any small improvement on it can cause large improvement to the whole algorithm execution. It has a very low complexity and handles a lot of data processing. Parallelism can cause much more acceleration on it not only by multithreading usage but also with other techniques like GPU usage.

On the multiprocessor technology the highest performance parallelism is achieved when the parallelization is applied over different filters and not inside the convolution process of a filter with the features image as shown in the Figure 54 below.

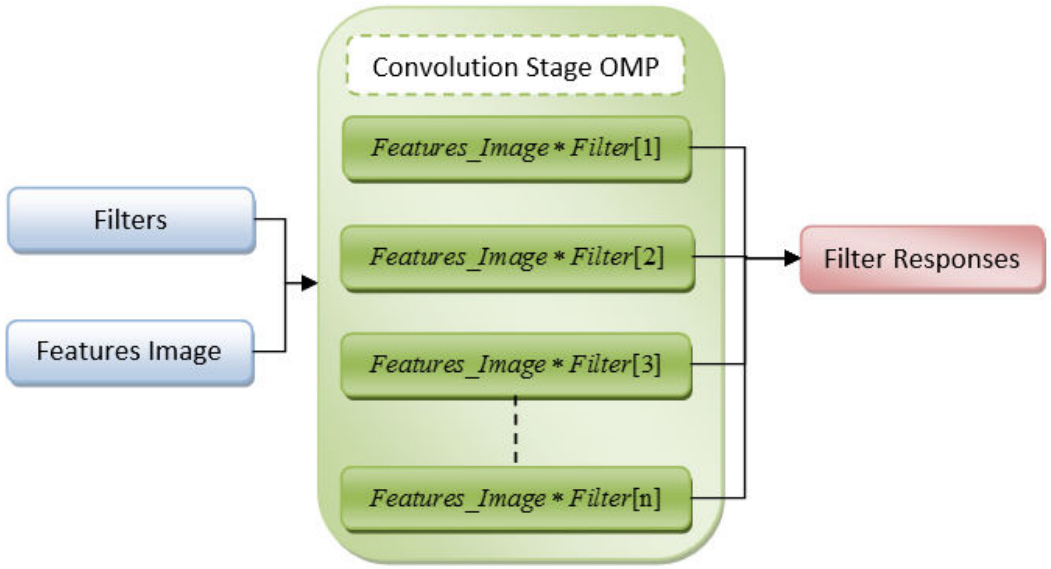


Figure 54 - Convolution Procedure OMP Diagram

By applying this parallelization in the convolution stage the results as long as the execution time are shown in Table 96 below.

Table 96 - Convolution Procedure OMP Execution Time (%)						
CPUs	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Speedup**
2	50.6	50.5	50.5	50.5	50.5	x2.0
3	33.5	33.4	33.4	33.4	33.4	x3.0
4	25.4	25.3	25.3	25.3	25.3	x4.0
5	20.3	20.3	20.2	20.2	20.2	x4.9
6	17.2	17.2	17.2	17.2	17.2	x5.8
7	15.2	15.2	15.2	15.2	15.2	x6.6
8	13.2	13.2	13.2	13.2	13.1	x7.6

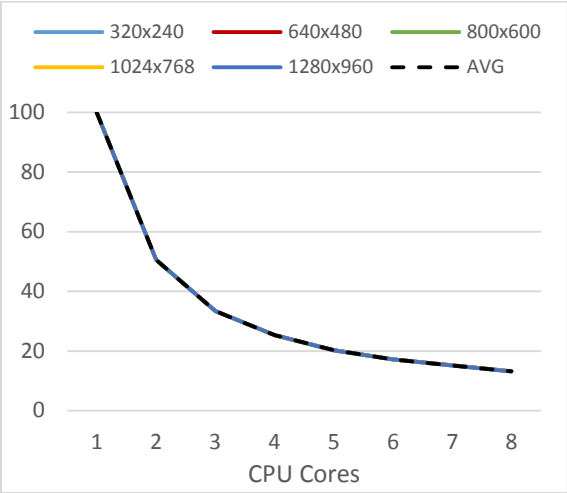


Diagram 65 - Convolution Procedure OMP Execution Time

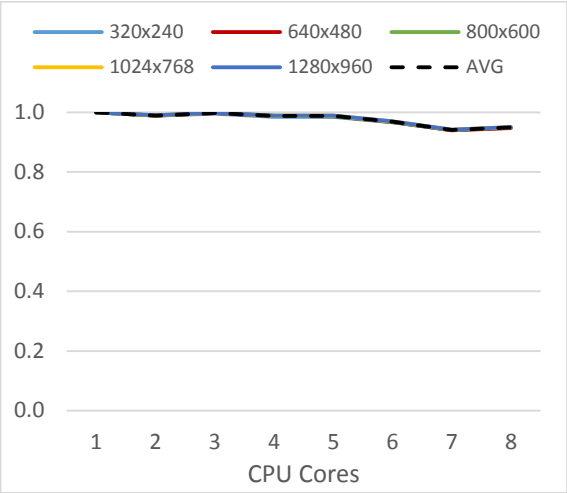


Diagram 66 - Convolution Procedure OMP Execution Time Efficiency

As seen in the diagrams and also in the Table 96, the multiprocessors parallelization technique creates great results as long as the Convolution stage. Just the usage of a 2nd CPU core achieved a speedup twice the time needed when using a simple core CPU. As shown in the last column in the Table 96 every CPU core added in the parallelization process gives the same size speedup. This is a very pleasant fact as the Convolution process is the one that needed most a time execution decrement. It is also very pleasant the fact that the use of every CPU core offers very efficient speedup with the efficiency index to be always over the 95%.

The convolution procedure is the one that probably deserves the most the bound of the hardware resources. It is very important to focus all the CPU cores at this stage as this parallelization tactic returns the highest results.

8.6. Distance Transformation

The Distance transformation stage cannot be parallelized as this process is sequential as explained in chapter 5.7. The parallelization technique can although used inside the Distance Transformation procedure that handles the main part of this stage. The Distance transformation procedure owns the 96.5% of this stage and about the 30% of the whole algorithm so it is very useful if a parallelized technique could reduce its execution time. By testing the OMP technology in the DT procedure the results are as shown in Table 97 below.

Table 97 - Distance Transformation Procedure OMP Execution Time (%)						
CPUs	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Speedup**
2	84.7	70.1	65.6	63.0	62.3	x1.5
3	67.7	51.6	49.1	45.8	44.2	x2.0

4	69.5	46.7	41.9	39.5	36.3	x2.2
5	70.3	42.7	37.7	34.1	31.2	x2.5
6	71.0	42.0	38.0	33.4	28.3	x2.6
7	80.6	43.3	37.5	32.6	26.7	x2.6
8	87.7	44.3	38.2	30.3	29.4	x2.5

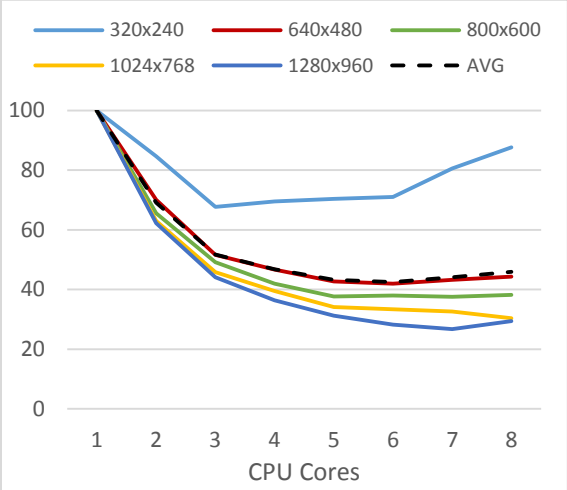


Diagram 67 - DT Procedure OMP Execution Time

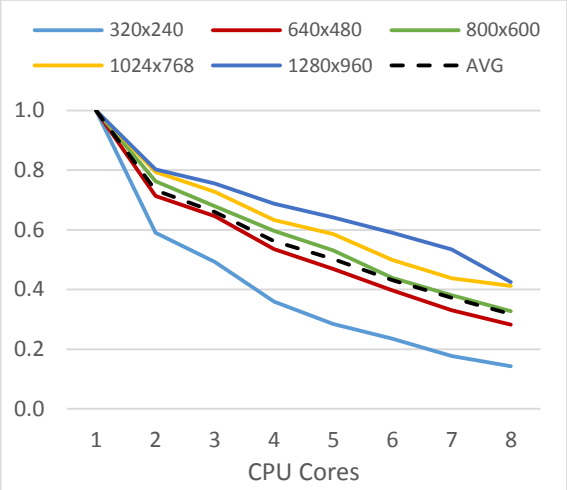


Diagram 68 - DT Procedure OMP Execution Time Efficiency

As seen in the Table 97 above and in the Diagram 67, the Distance Transformation procedure is actually gains speedup until the usage of the third core of the CPU. After the third CPU core the speedup is affected by the size of the image. In addition the efficiency graph shows that the DT procedure parallelization is not very efficient as its efficiency is stably reducing. If the hardware resources are available maybe the usage of six CPU cores would be useful but on the other hand by observing the Diagram 67 the usage of 3 cores might be the best choice. What is also visible, especially in Diagram 68, is that the image size affects the parallelization efficiency. It seems that as larger is the image, more CPU cores can be used efficiently.

8.7. Backtrack Stage

The Backtrack stage consists of two basic procedures, the Find one and the Backtrack one. The Backtrack procedure cannot be parallelized as its processing is sequential and sequence depended. This is not a problem as far as the Find v2.0 patch is used that extremely reduced its execution time. On the other hand after the usage of the Find v2.0 patch the Find procedure increased its execution time and it is the main time consumer of the Backtrack stage. The Find procedure can be parallelized but is contains a great part of critical procedures. Anyway, the Backtrack stage holds only the 0.05% of algorithms execution time. As far as the Find procedure, the speed up gain using the multicore processors is shown in Table 98 below.

Table 98 - Find Procedure OMP Execution Time (v2.0) (%)						
CPUs	320x240	640x480	800x600	1024x768	1280x960	Speedup
2	82.1	64.5	63.6	50.5	50.2	x1.7
3	65.0	57.2	56.5	48.5	51.1	x1.8
4	76.4	65.7	59.8	56.1	58.1	x1.6
5	75.4	63.1	61.2	58.9	60.4	x1.6
6	67.5	67.2	63.9	58.6	59.9	x1.6
7	67.5	74.5	61.3	62.0	57.3	x1.6
8	81.4	70.1	57.4	61.9	56.5	x1.6

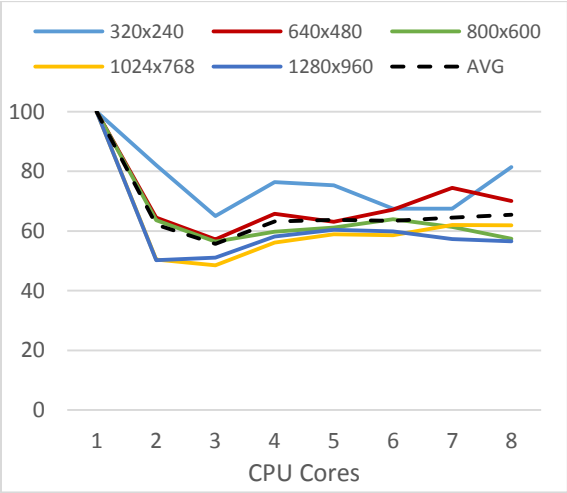


Diagram 69 - Find v2.0 Procedure OMP Execution Time

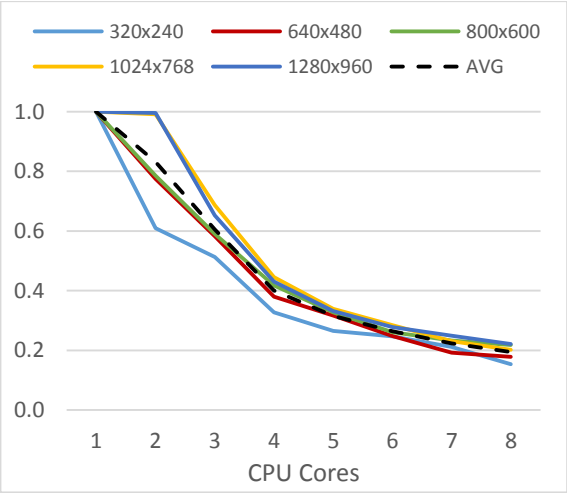


Diagram 70 - Find v2.0 Procedure OMP Execution Time Efficiency

The Table 98 and also the two diagrams (Diagram 69, Diagram 70) above makes it clear that the usage of parallelization techniques are efficient in the Find procedure when using only 2 CPU cores. Probably the existence of critical data are the reason of the negative efficiency and unstable attitude when more CPU cores are used. It is also visible in the Diagram 70 that the Find procedure parallelization is more efficient when used for large images in contrast to the small ones.

The Find procedure and the whole Backtrack stage are holds such a small part of the algorithm's execution time that either using parallelization or not would probably affect the algorithm's execution time in an unnoticeable level.

8.8. Level Stage

The Level detection stage is the main stage of the TSM algorithm. It is the stage where the convolution stage results are processed and the detection results come from. It contains the Distance Transformation procedure repeated by multiple times for every component of the

algorithm's model. It is the most complex part of the whole algorithm. In this chapter a series of tests over this part is presented in order to discover the most effective parallelization tactic.

8.8.1.1st Tactic

The first parallelization tactic uses the parallelization of the Distance Transformation and Find procedure inside this stage, as the Backtrack one cannot be parallelized. The results of this tactic is presented in the Table 99 below,

Table 99 - Level Stage OMP 1st Tactic Execution Time (%)						
CPUs	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Speedup**
2	90.2	76.0	71.2	68.2	68.0	x1.4
3	75.1	59.2	56.9	53.1	51.6	x1.7
4	76.5	54.0	49.1	46.6	43.6	x1.9
5	78.0	50.7	45.7	42.1	39.1	x2.1
6	78.3	49.9	45.9	40.7	36.6	x2.1
7	87.7	50.5	45.7	40.5	34.6	x2.1
8	96.7	51.9	46.0	37.4	37.1	x2.1

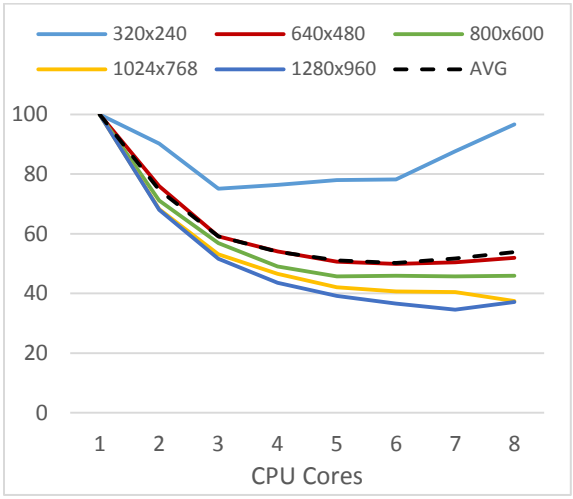


Diagram 71 - Level Stage OMP Execution Time (1st Tactic)

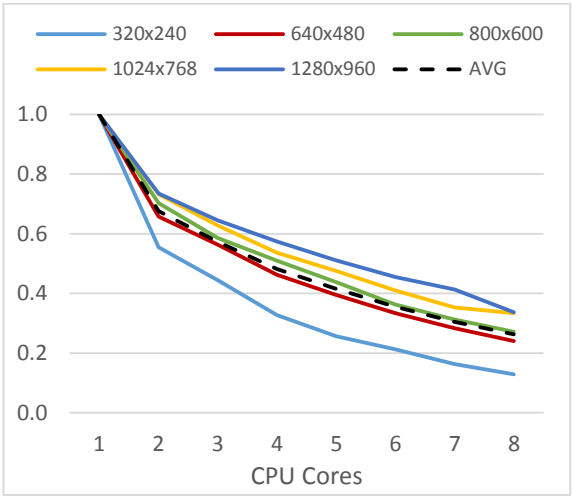


Diagram 72 - Level Stage OMP Execution Time Efficiency (1st Tactic)

As seen in the Table 99 and Diagram 71 the Levels stage reaction to this tactic is not linear. The levels stage is speeding up at the usage of the first CPU cores but at the end it seems to lose its acceleration. This is sensible as it follows the attitude of the DT stage (Chapter 8.6) which is the main stage contained by the levels stage. This attitude reacts negatively to this parallelization tactic efficiency as shown in the Diagram 72 above. As it is visible this tactic's efficiency is reducing continuously as the number of CPU cores is increased.

Another characteristic of the first parallelization tactic is that it is not so image size independent. As seen in the graphs the smaller the size of the image is the less efficient is this tactic. This is very obvious with the smallest tested image of 320x240 pixels where the speedup is gained is very low and it tends to become lower as the CPU cores are increasing.

As far as the memory consumption of this tactic is not actually affected as the DT and Find procedure parallelization does not consumes any significant amount of memory.

8.8.2.2nd Tactic

As done in the Feature Pyramid stage where a loop procedure exists, the second parallelization tactic is a tactic that separates the loop in different CPU cores (Figure 55). In practice every CPU cores undertakes a component stage execution of the level. The number of components is enough (13) to bind all available cores. The negative affect of this tactic is that it consumes much more memory than the single core one. The increase of the memory consumption of this stage when using this tactic is shown in the Table 100 below.

In the second parallelization tactic a new execution flow diagram is applied as shown in the Figure 55 below where the multiple threads of the CPU are distributed to every component stage procedure. This way multiple components detection procedure can run in parallel.

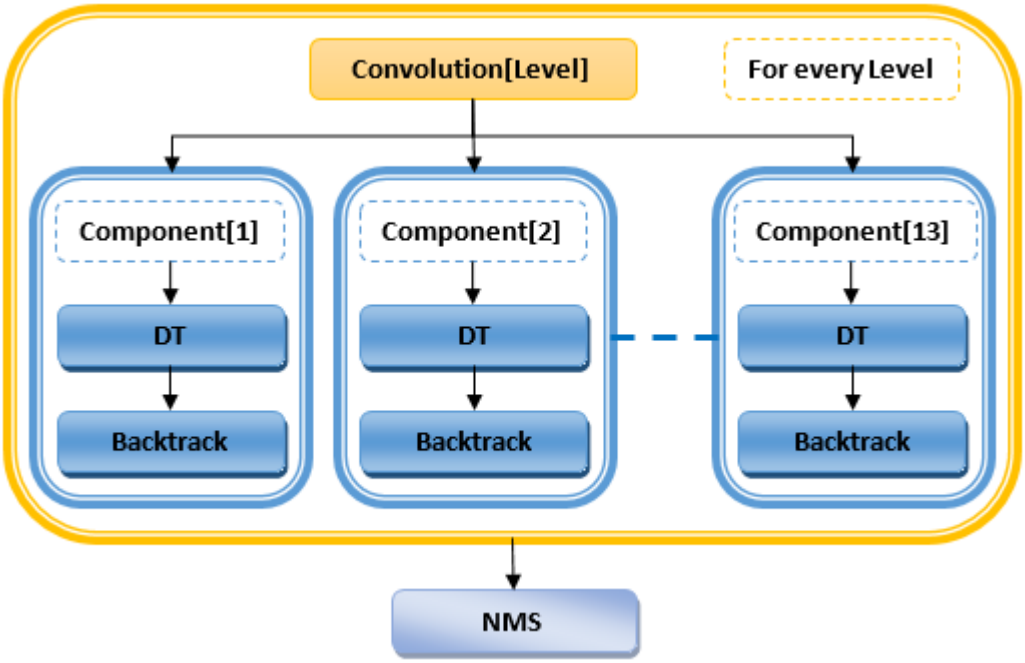


Figure 55 - Level Stage OMP 2nd Tactic Diagram

In the Table 100 below the effect of the 2nd parallelization tactic is shown as far as the time consumption of the Level stage. As seen in this table the Level stage gains as great speedup, up

to five times, when used with more than five CPU cores. It is also visible that the Level stage is gaining a very efficient speedup even from the usage of the first extra CPU core in contrast to the 1st tactic.

Table 100 - Level Stage OMP 2nd Tactic Execution Time (%)						
CPUs	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Speedup**
2	53.4	52.1	51.7	53.7	51.5	x1.9
3	38.7	37.1	36.5	36.9	36.5	x2.7
4	28.9	28.2	27.9	28.1	28.0	x3.5
5	28.0	27.8	27.8	28.1	27.3	x3.6
6	19.4	19.3	18.4	19.4	18.7	x5.3
7	20.5	19.7	19.0	19.5	19.0	x5.1
8	20.4	19.4	18.9	19.4	19.1	x5.1

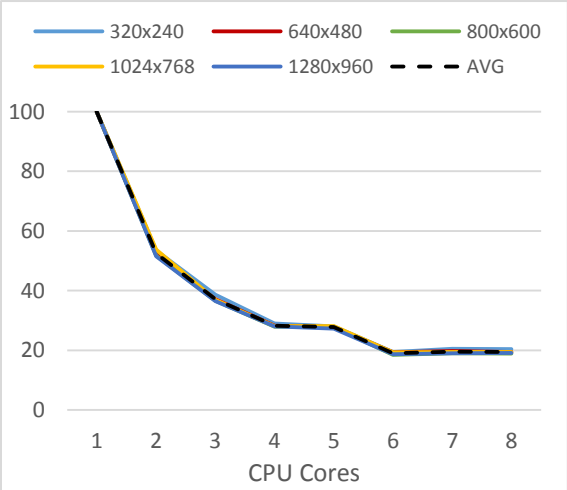


Diagram 73 - Level Stage OMP Execution Time (2nd Tactic)

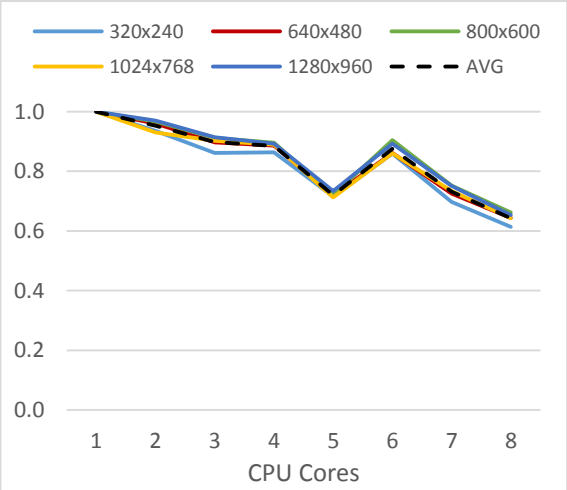


Diagram 74 - Level Stage OMP Execution Time Efficiency (2nd Tactic)

The results of this tactic as shown in the Table 100 and Diagram 73 is that the execution time of the level stage is reducing rapidly until the fourth CPU core. By the fifth core and the usage of the second CPU of the testing hardware the stages execution time stops reducing significantly. What is extremely positive is that the usage of extra CPU cores in this tactic produces a very good efficiency that always stays over 60%.

Except of the time effect, the 2nd parallelization tactic increases also the memory consumption. As referred in chapter 6.20 the maximum memory consumption of the TSM algorithm is reached at the Level stage when applied in the top image pyramid level. The maximum memory consumption at this point can be predicted using the function (15). In this function the algorithm's maximum memory is the sum of the Filters Responses, plus the DT Scores produced

in the DT stage (the same for every component), plus the Backtrack stage temporary memory, all during the execution of the Level stage for the features pyramid top level. At last the memory consumed by the Model and other data, that is a stable size, is added.

When more CPU cores are used what is parallelized is actually the Component stage. The Filters Responses are the same for all threads. What is private is the DT and Backtrack stages that are executed multiple times in the different CPU cores. This means that the maximum memory consumption can be predicted using the expression of function (16).

$$FD_{\max} = F.\text{Responses}[0] + \text{DT}.\text{Scores}[0] + \text{Backtrack}[0] + \text{Others} \quad (15)$$

$$FD_{\max} = F.\text{Responses}[0] + \text{Others} + Cores \times \left(\text{DT}.\text{Scores}[0] + \text{Backtrack}[0]\right) \quad (16)$$

Using the function (16) the memory consumption of this stage is multiplied by the number of cores used as presented in Table 101 below.

Table 101 - TSM v3.2.2 Level Stage OMP 2nd Tactic Max Memory (Mbytes)						
CPUs	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
1 core	5.47	20.1	31.0	50.2	78.0	
2 cores	8.27	31.0	47.9	77.9	121	+54%
3 cores	11.1	41.9	64.9	106	164	+108%
4 cores	13.9	52.8	81.8	133	207	+162%
5 cores	16.6	63.7	98.8	161	250	+216%
6 cores	19.4	74.6	116	188	293	+270%
7 cores	22.2	85.6	133	216	336	+324%
8 cores	25.0	96.5	150	244	379	+378%

As seen in the Table 101 the usage of a full eight CPU cores parallelization can cause up to 380% memory consumption incremental. It is obvious by the data of the Table 101 that this parallelization technique has a heavy memory consumption cost. On the other hand, considering the relationship between CPU cores and RAM memory that is usually offered in the hardware market, this increment in the maximum memory consumption is not prohibitive. It would be unusual an eight cores CPU hardware with less than one Gigabyte of RAM memory!

At this point is very important to refer to the Find v2.0 patch described in chapter 7.2. At this chapter the memory reduction that was gained using this patch was appose. This reduction is proved very significant on this 2nd parallelization tactic as it keeps low the memory consumption incremental. In the Table 102 below the maximum memory consumption of this tactic without using the Find v2.0 patch is presented in order to be understandable the benefits this patch offered.

Table 102 - TSM v3.2.2 Level Stage OMP 2nd Tactic Max Memory (Mbytes)						
CPUs	320x240	640x480	800x600	1024x768	1280x960	Average
1 core	13.8	52.5	81.3	132	206	
2 cores	25.0	95.8	149	242	377	+82.4%
3 cores	36.1	139	216	352	547	+165%
4 cores	47.3	182	283	461	718	+247%
5 cores	58.4	226	351	571	889	+329%
6 cores	69.5	269	418	681	1,060	+412%
7 cores	80.7	312	485	791	1,231	+494%
8 cores	91.8	356	552	900	1,401	+577%

As seen in the Table 102 above, the maximum memory consumption of the algorithm when using the 2nd parallelization tactic would be much larger creating questions about the ability of using it at any hardware. As seen in the 8 cores line the maximum memory consumption reaches even more than one gigabyte of memory.

8.8.3.3rd Tactic

At last, a combination of these two tactics is tested. This 3rd tactic used the parallelized version of the DT and Find procedure and on the same time shares component detection on several CPU cores. On the Table 103 the results of this tactic when used until two CPU cores for the 2nd tactic while the rest cores are shared to the 1st tactic. In the next table, Table 104, the same result when the 2nd tactic uses until four CPU cores.

Table 103 - Level Stage OMP 3rd Tactic Execution Time (%) (Tactic 2 = 2 cores)						
CPUs	320x240	640x480	800x600	1024x768	1280x960	Speedup
2	54.9	52.3	52.1	52.5	52.5	x1.9
3	58.6	51.4	47.9	45.9	44.5	x2.0
4	53.0	50.0	46.6	41.8	40.6	x2.2
5	58.8	45.9	42.0	36.5	34.0	x2.4
6	59.6	49.1	43.9	40.6	36.5	x2.2
7	67.4	52.2	45.1	38.5	33.4	x2.2
8	81.6	61.5	52.8	44.7	37.7	x1.9

Table 104 - Level Stage OMP 3rd Tactic Execution Time (%) (Tactic 2 = 4 cores)						
CPUs	320x240	640x480	800x600	1024x768	1280x960	Speedup
2	54.5	52.4	52.4	52.9	52.1	x1.9
3	37.7	36.1	36.5	37.1	35.9	x2.7

4	28.8	27.7	27.3	27.1	26.7	x3.6
5	28.7	28.7	27.8	27.9	27.9	x3.5
6	31.8	29.5	28.3	29.3	28.0	x3.4
7	49.0	40.8	35.0	31.6	28.4	x2.8
8	28.8	25.5	23.8	23.1	22.2	x4.1

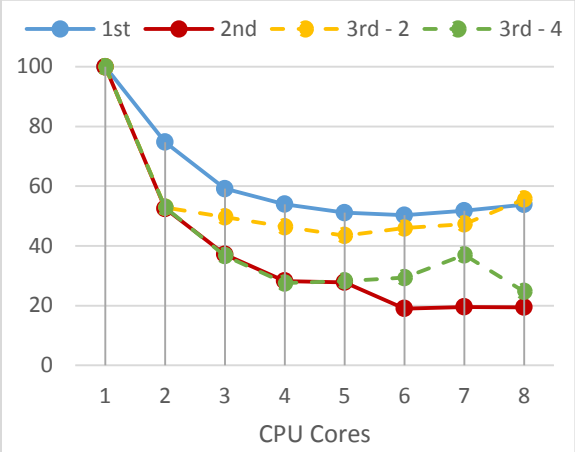


Diagram 75 - Level Stage OMP Execution Time (All Tactics)

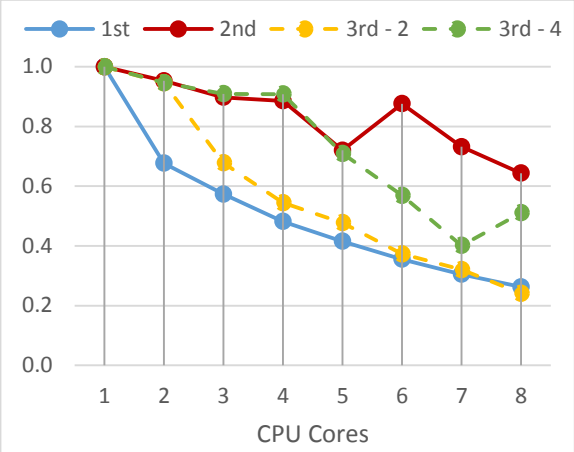


Diagram 76 - Level Stage OMP Execution Time Efficiency (All Tactic)

As seen in these two graphs, according to the Table 99, Table 100, Table 103 and Table 104, the 3rd parallelization tactic is not gaining any special speedup by the time parallelization is applied in the DT and Find stage. When it is using only two CPU cores for the component stage parallelization it seems that there is no worth using it, as until the 3rd CPU core, it is having the same result as using the 2nd tactic on its own. When using 4 cores for the component stage parallelization the results also does not seems to be better than using the 2nd parallelization tactic on its own with maximum of four CPU cores.

The 3rd parallelization tactic does not seems at all to produce any beneficial result as shown in Diagram 75 above. As shown in the Diagram 76, the 2nd tactic seems to be more efficient than the others, consuming memory that does not seems to produce memory issues. Even if the Find v2.0 patch is not in use the 2nd parallelization tactic would also be preferable as even when used with only 2 or three CPU cores succeeds better results in contrast to the 1st tactic even when the last one uses all the eight CPU cores.

8.9. TSM Algorithm

After examining the effect of parallelism in most of the stages and procedures of the algorithm, in this chapter a comparison of every stage and procedure according to its efficiency is appose. In the Table 105 below the efficiency of every stage of the algorithm that is referred in previous chapters.

Table 105 - TSM Procedures & Stage OMP Efficiency								
Stage	Memory Charge	CPU Cores						
		2	3	4	5	6	7	8
Resize	No	0.62	0.49	0.42	0.35	0.29	0.25	0.22
Reduce	No	0.75	0.90	0.68	0.54	0.45	0.38	0.34
HOG	No	**0.93**	**0.91**	**0.87**	**0.83**	0.78	0.78	0.77
FP Stage	Yes	0.66	0.60	0.45	0.64	0.53	0.46	0.40
	No	**0.83**	0.75	0.66	0.58	0.48	0.41	0.37
Convolution	No	**0.99**	**1.00**	**0.99**	**0.99**	**0.97**	**0.94**	**0.95**
DT	No	0.73	0.66	0.56	0.50	0.43	0.37	0.32
Find	No	**0.83**	0.61	0.40	0.32	0.26	0.22	0.19
Level Stage	Yes	**0.95**	**0.90**	**0.89**	0.72	**0.88**	0.73	0.64
	No	0.68	0.57	0.48	0.41	0.35	0.31	0.26

The Table 105 is a parallelization map giving useful information of how the parallelization affects the algorithms parts and proposing the parts that the CPU cores have to be focused. It is also shows when the parallelization affects the memory consumption of the TSM algorithm warning for memory issues. Using this table, two parallelization tactics are presented for the whole algorithm. The 1st tactic is using the most time efficient tactics of the algorithms' parts while the 2nd one is using the most memory consumption efficient tactics of the DT and Features Pyramid stages. In the next subchapters the impact of those two tactics over the two latest versions of the algorithm is appose.

8.9.1. TSM Algorithm v2.2

In the Table 105 (Chapter 8.9) the green bolded lines show the parallelized procedure with the higher efficiency that can be used in order to achieve the highest speedup. By using the most efficient tactics of the algorithms in version 2.2.2 the results are as shown in the Table 106 below.

Table 106 - TSM v2.2.2 OMP Execution Time (Time Efficient Version) (%)						
CPUs	320x240	640x480	800x600	1024x768	1280x960	Speedup
2	51.5	51.2	50.9	51.5	51.5	x1.9
3	35.6	34.9	34.8	34.8	34.4	x2.9
4	27.2	26.7	26.5	26.6	26.3	x3.8
5	23.6	23.1	22.9	23.0	22.9	x4.3
6	18.9	18.4	18.3	18.3	18.0	x5.4
7	17.2	17.1	16.9	17.0	16.8	x5.9
8	15.7	15.8	15.5	15.7	15.6	x6.4

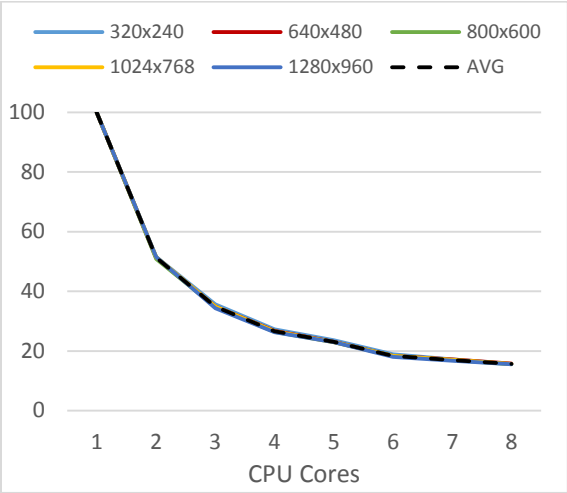


Diagram 77 - TSM v2.2.2 OMP Execution Time (Time Efficient)

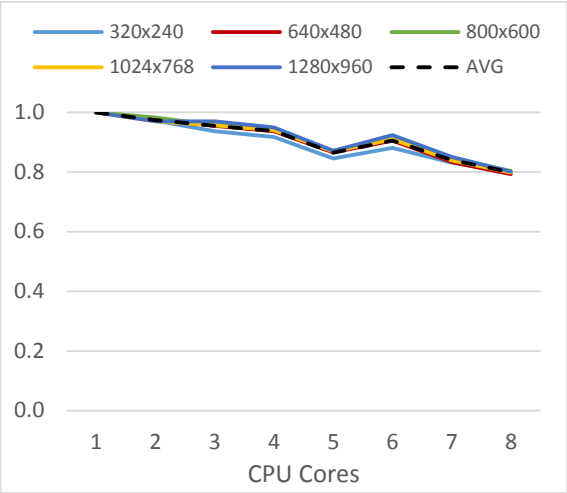


Diagram 78 - TSM v2.2.2 OMP Execution Time Efficiency (Time Efficient)

In the Diagram 77 the Table 106 data are figured. As seen in the graph the 1st parallelization tactic of the 2.2.2 version of the TSM algorithm is image size independent. It is very pleasant that using eight CPU cores produce a speedup of 6.4 times faster. As shown in the Diagram 78 the algorithms parallelization efficiency is always higher than 80% and more than 95% for the first four cores used. This is a very positive result!

As far as the memory consumption of the algorithm when using this tactic the maximum memory consumption of the algorithms is presented in the Table 107 below.

Table 107 - TSM v2.2.2 OMP Max Memory Consumption (Mbytes)					
CPUs	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**
1	5,3	19	30	48	74
2	8,1	30	46	76	117
3	11	41	63	103	160
4	14	52	80	131	203
5	16	63	97	158	246
6	19	74	114	186	289
7	22	85	131	214	332
8	25	96	148	241	375

As seen in the Table 107 above the maximum memory consumption needed for the algorithm, even when eight CPU cores are used, seems not to be prohibited according to the usual hardware designs in the market. It is almost unusual to have a hardware with more than 4 CPU cores and less than 2 gigabytes of memory. In addition it is also unusual to have an 8 cores CPU

(2 CPUs) with less than 4 gigabyte memory. Even the embedded systems are usually designed with 0.5 or 1 gigabytes of memory and 2 or 4 cores CPU.

On the other hand the image processing algorithms usually use small sized images at the size of 640x480 (0.3 megapixels). As seen in the Table 107 the algorithm consumes less than 100 megabytes of memory for images of this size. This observations show that this tactic can be used for any hardware design.

Nevertheless the fact that the 1st tactic hardware requirements is suitable for the majority of the embedded systems in the market a second parallelization tactic is appose as a CPU cores independent version. This tactic uses the parallelized versions on every procedure that keeps the maximum memory consumption stable. This tactic time results are shown in the Table 108 below.

Table 108 - TSM v2.2.2 OMP Execution Time (Memory Efficient Version) (%)						
CPUs	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Speedup**
2	62.1	57.4	56.3	56.6	52.5	x1.8
3	46.2	40.9	39.8	39.8	38.4	x2.4
4	40.8	32.9	31.8	31.5	32.5	x3.0
5	37.5	28.4	28.6	27.2	26.0	x3.4
6	36.9	25.5	24.9	24.3	25.0	x3.8
7	35.0	23.6	22.9	21.8	21.2	x4.2
8	34.7	23.2	21.9	20.4	19.5	x4.4

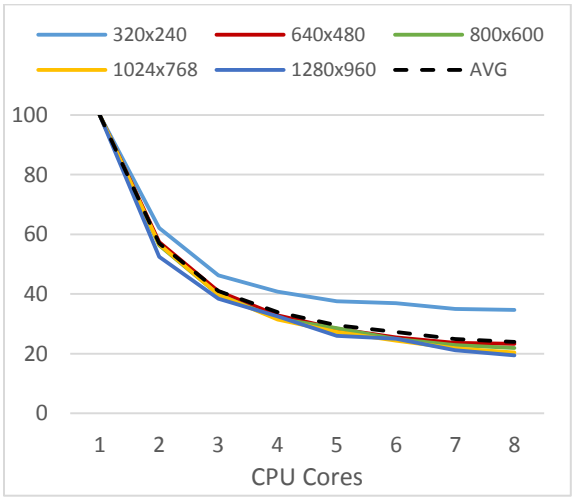


Diagram 79 - TSM v2.2.2 OMP Execution Time (Memory Efficient)

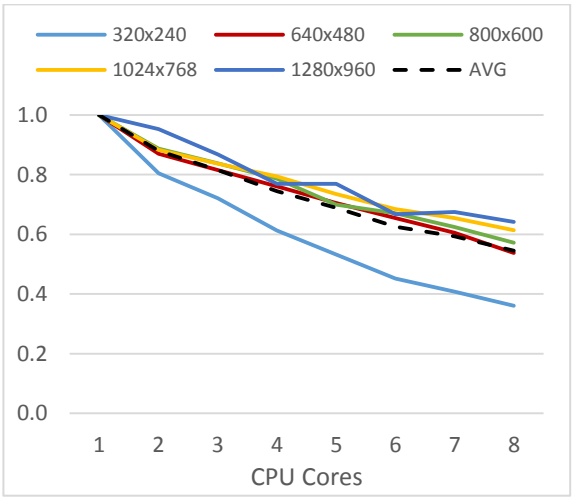


Diagram 80 - TSM v2.2.2 OMP Execution Time Efficiency (Memory Efficient)

As seen in the Table 108 the algorithms speedup is increasing a bit as the image size is greater. On the Diagram 80 is also visible a fall of the algorithms efficiency as the image size is getting smaller. These tables show that the 2nd parallelization tactic can be efficiently used for large sized images as this tactic holds the maximum memory consumption low and also its execution time is approaching the execution time of the 1st tactic as the image size is getting larger.

The 2nd tactic maximum memory consumption is the same with the single core algorithms implementation shown in the Table 107 above corresponding line.

8.9.2. TSM Algorithm v3.2

Applying the 1st parallelization tactic in the version 3.2 of the algorithm (Chapter 6.20), the execution time impact is as shown in the Table 109 below. As seen in this table the 3.2 version of the algorithm succeeds up to 6.4 times. It is very positive that even from the usage of a second CPU core the execution time is decreased to its half.

Table 109 - TSM v3.2.2 OMP Execution Time (Time Efficient Version) (%)						
CPUs	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Speedup**
2	51.6	51.3	51.2	51.9	51.1	x1.9
3	35.3	34.9	34.7	34.8	34.7	x2.9
4	26.7	26.6	26.5	26.5	26.6	x3.8
5	22.8	23.1	23.1	23.1	23.0	x4.3
6	18.4	18.5	18.2	18.3	18.2	x5.5
7	17.3	17.2	17.0	17.0	17.0	x5.8
8	16.3	15.8	15.6	15.7	15.7	x6.3

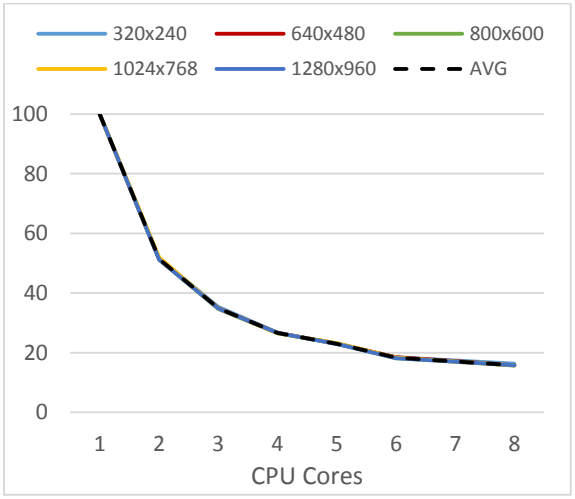


Diagram 81 - TSM v3.2.2 OMP Execution Time (Time Efficient)

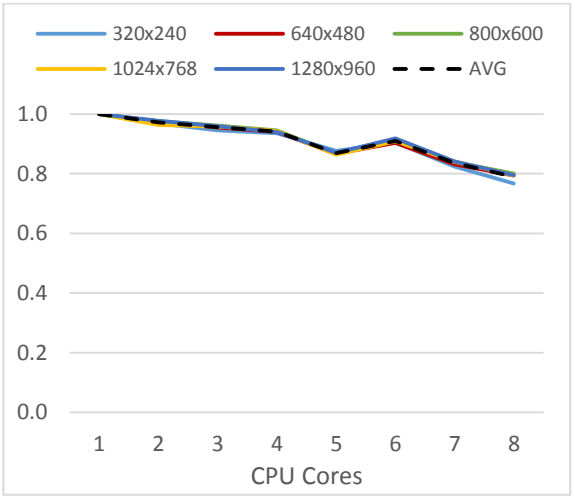


Diagram 82 - TSM v3.2.2 OMP Execution Time Efficiency (Time Efficient)

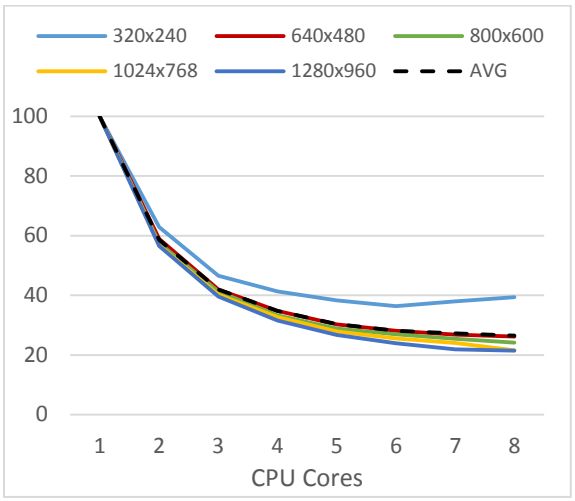
In the Diagram 81 and Diagram 82, it is extremely obvious that the 1st parallelization tactic is totally image size independent. This is also visible by the data of the Table 109. The algorithm is speeding up during all the extra CPU cores used and its efficiency is always very high over the 80% holding it over the 95% for the four primal cores (1 CPU). The impact of this tactic is very positive.

In the Table 110, the impact of the 1st parallelization tactic is shown. As seen in this table even when using more than one CPU and large sized images the algorithm's maximum memory does not exceed the 400 megabytes. Although, as referred in previous chapters, the combination of CPU cores and RAM memory appears in the hardware market does not make the usage of this version of the algorithm prohibited.

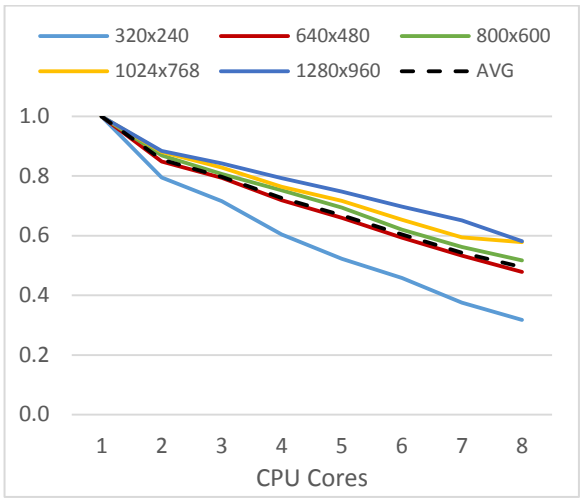
Table 110 - TSM v3.2.2 OMP Max Memory Consumption (Mbytes)						
CPUs	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
1	5,5	20	31	50	78	
2	8,3	31	48	78	121	+54.0 %
3	11	42	65	106	164	+108 %
4	14	53	82	133	207	+162 %
5	17	64	99	161	250	+216 %
6	19	75	116	188	293	+270 %
7	22	86	133	216	336	+324 %
8	25	96	150	244	379	+378 %

The 1st parallelization tactics is the fastest one using a sensible amount of memory available at the majority of the hardware designs in the market. Although it is important to present the impact of the 2nd parallelization tactic on this version of the algorithm. This impact in the algorithm's execution time is shown in the Table 111.

Table 111 - TSM v3.2.2 OMP Execution Time (Memory Efficient Version) (%)						
CPUs	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Speedup**
2	62.9	58.9	57.5	56.8	56.6	x1.7
3	46.6	42.0	41.3	40.3	39.6	x2.4
4	41.4	34.8	33.3	32.7	31.5	x2.9
5	38.3	30.3	28.8	27.9	26.8	x3.3
6	36.3	28.1	26.9	25.5	23.9	x3.6
7	38.0	26.8	25.4	24.0	21.9	x3.8
8	39.4	26.1	24.1	21.6	21.5	x4.0



TSM v3.2.2 OMP Execution Time (Memory Efficient)



TSM v3.2.2 OMP Execution Time Efficiency (Memory Efficient)

In contrast to the 1st parallelization tactic, the 2nd one of the 3.2.2 version of the algorithm does not react the same way to all image sizes. As seen this tactic is more efficient to the larger images. As seen in the Diagram 82 the algorithms efficiency is decreasing linearly as the number of CPU cores is increasing falling under the 60% at the last 2 cores. This is the main difference between this and the 1st tactic which keeps its efficiency high for all usable CPU cores.

The 2nd tactic maximum memory consumption is the same with the single core algorithms implementation shown in the Table 110 above corresponding line.

8.9.3. TSM Algorithm v4.1

On this chapter a last version of the algorithm is presented, designed for multi CPU systems using more than 1 CPUs. As presented in chapter 8.9, many of the algorithms stages and procedures shows reduction of their parallelization efficiency while the CPU cores used are increasing. In this version the maximum efficiency of all procedures is tried to be succeeded. In the Diagram 83 below the efficiency of all procedures is presented according to the CPU cores used. As seen in this graph, in the most procedures the maximum efficiency is when using less cores and only the HOG and the Convolution procedures keep their efficiency high despite the usage of multiple CPU cores.

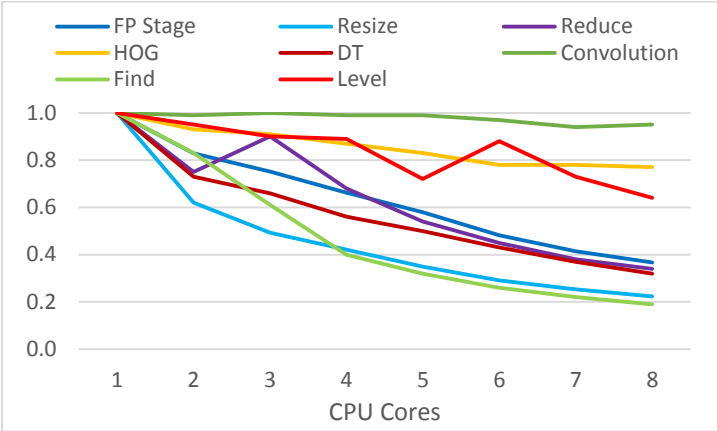


Diagram 83 - TSM OMP Procedures Efficiency per CPU Core

According to the Diagram 83 above, the HOG and the Convolution procedures are the most efficient and the most stable, all the rest are either inefficient or unstable. This version of the TSM algorithm (v4.1) is based on focusing on these two procedures offering the majority of the hardware resources to them. The idea of this last version is based on splitting the algorithm is two sections trying to get the maximum efficiency of each of them. As seen in the Figure 56 below the first section is an extension of the Features Pyramid stage containing the HOG and the Convolution procedures that are the most efficient ones and the second one the Level stage that is less efficient and its maximum efficiency is reached when using low number of cores. These two section have to share the hardware resources (CPU cores) in a way that would offer the maximum efficiency to the algorithm.

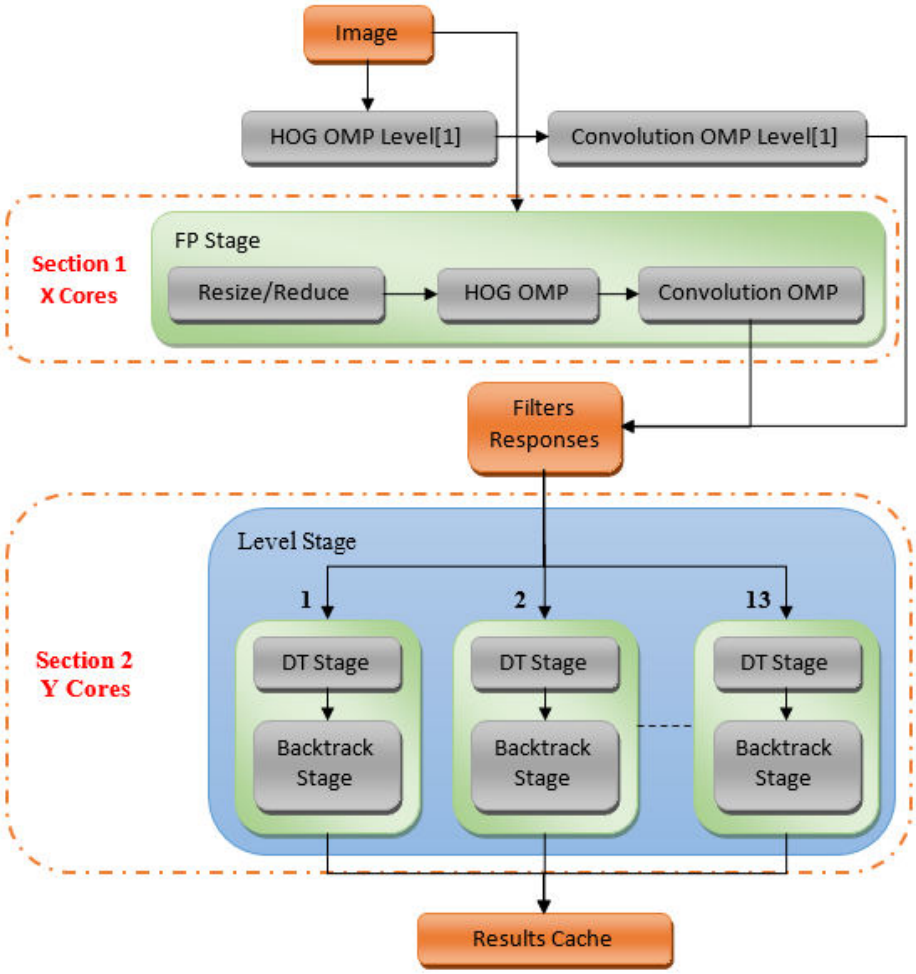


Figure 56 - TSM Algorithm v4.1 Execution Flow Diagram

In the Figure 56 above a complicated execution flow graph is presented. At the beginning, the algorithm uses all the available CPU cores in order to calculate the HOG descriptors and the Filter Responses of the first level of the Image Pyramid. No resize procedure is needed for the top level of the Features pyramid as it uses the image on its original size. This level is also the one with the greatest size image so the most time consuming level. The HOG and the Convolution procedures on the other hand are the most efficient ones worth to the usage of the full CPU resources. After the first level Filters Responses are calculated the algorithm separates the available CPU cores in the two sections. The first section is calculating the features images and the Filters Responses of every level while and the second one executes the level stage looking for detections. The Level stage section needs the Filters Responses data structure as input and this is the reason why the algorithm calculates the first level Filters Responses outside the sections section. If the algorithm do that inside the first section the second would be idle, waiting the first one to finish the first level calculations in order to use it. In this design the

second stage starts immediately its execution as its input data are already calculated while the first section calculates the next level's Filters Responses.

The idea behind this execution design is the usage of as many as it can CPU cores to the most efficient and also most time consuming procedure. This procedure is the Convolution one. The HOG procedure is added to the same section cause of its sequential relationship with the Convolution one and mainly because of its very high efficiency in parallelism. On the other hand, while most of the CPU cores are allocated to the section one, less cores are busy to the section two where other procedures are executing that need less execution time and are less efficient. This technique is hiding in a way the time consumption of the procedures executed in the section two behind the consumption time of the section one.

Another though that motivated this version is to limit the number of CPU cores used in the Level stage parallelization in order to reduce the maximum memory consumption it consumes. Using the CPU cores at the Convolution procedure that creates zero temporary memory the algorithm can succeed high execution time efficiency with less memory consumption. At the same time the time the algorithm can execute the Section two without the need of allocating the great amount of memory that the each Component stage needs.

Between those two sections there is unfortunately a dependency. This dependency is that the output data of the section one (Filters Responses) are the input of the section two (Level Stage). It is obvious that the section two has to wait for a while the section one to finish some of its calculations. The ideal usage of this version would be a balanced share of the CPU cores between those two sections so that when the first section finishes the calculations of a levels Filters Responses, the second section would start the components detection on the same level.

There are two moments where the section two waits the section one. The first time is when the section two waits the section one to calculate the first level Filters Responses. The second moment is when the first section finishes its procedure and the whole algorithm has to wait the section two to finish. As referred above the section two cannot finish before the section one as it uses its outputs as an input. This means that the section one will always finish a levels component detection time earlier than the section two. As far as this type of «waiting», the solution is to start the detection from the top to the bottom of the Image Pyramid so that the last components detection of section two would use the smallest image Filters Responses which is the less time consuming level. This solution transfer the «waiting» problem to beginning where the section two has to wait for the first calculated Filters Responses. The solution to this problem is as designed in this version and figured in Figure 56, where the algorithm uses all its available CPU cores in order to calculate the first level's Filters Responses. This way the first levels HOG and Convolution procedure occurs outside the sections section and no cores have to set idle.

Using the data of the efficiency and execution time tables, a performance function was created for the algorithm version 3.1 in order to predict the algorithm performance according to the CPU cores used in each section. This function is depending on two more function as shown below.

$$x2 = x - x1 \quad (17)$$

$$F_{t.x} = F_t * \mathrm{F}_{ef.x} \quad (18)$$

$$FD_{t.x} = 0.25 \cdot \left(HOG_{t.\mathrm{x}} + Conv_{t.x}\right) + MAX\left(0.75 \cdot \left(HOG_{t.\mathrm{x}1} + Conv_{t.\mathrm{x}1}\right) + IPstage_{t.\mathrm{x}1}, Comp_{t.\mathrm{x}2}\right) \quad (19)$$

In function (17), the *x2* is the number of cores used in section 2 and *x1* are the cores used in the section 1. X is the number of CPU cores available by the hardware.

In function (18), $F_{t.x}$ is the execution time percentage needed by the procedure *F* during the algorithms execution, when F_t is the execution time percentage needed by the procedure when no parallelism techniques are used and $F_{ef.x}$ the efficiency of the procedure *F* when parallelism techniques are used with *x* number of CPU cores.

At last the function (19) is the algorithms version 3.2 performance function according to the CPU cores used in each section. As seen in the first part of the function the performance of the HOG and Convolution procedures when executed for the first level of the Image Pyramid are multiplied by the number of 0.25. This is because the first level of the Image Pyramid holds the 25% of the whole Image Pyramid data as explained in chapter 6.10. At the second part of the function (19) the execution time of each section is calculated and the greater is kept. What is not calculated, because it is very hard to be predicted in contrast to its significance, is the delay that the section 1 can cause to section 2. This delay is insignificant as it affects the function result only when both sections needs almost the same time and also because it has a very small value as it concerns the last level of the Image pyramid that is the smallest one and the detection procedure is extremely fast.

Using function (19) with the data of the execution time and efficiency tables (the average efficiency as far as the image size) the results come of are shown in Table 112 below,

Table 112 - TSM v4.1.2 Execution Time Simulation

Available Cores	Section 2 cores							v3.2 Sim	v3.2 Real
	1	2	3	4	5	6	7		
8	33.4	18.6	**13.8**	15.7	19.9	28.8	54.1	15.6	15.8
7	33.7	19.0	**16.0**	20.2	29.1	54.4		17.0	17.1
6	34.1	**19.3**	20.6	29.4	54.8			18.2	18.3

5	34.6	**21.1**	29.9	55.3				23.0	23.0
4	35.5	**30.8**	56.1					26.5	26.6
3	**36.8**	57.5						34.9	34.9
2	**60.4**							51.5	51.4

As shown in the Table 112 above the simulated results show that this version (4.1.2) is going to be faster than the version 3.2.2 when using more than one CPU hardware resources but slower for single CPU hardware. The simulated results also shows that this design of the algorithm works better when 2 or 3 CPU cores are offered to the Section 2. Using this data, the version 4.1.2 was tested in real world and its results are shown in the following tables.

Table 113 - TSM v4.1.2 Execution Time

Cores	Section 2 cores 320x240				v3.2		Section 2 cores 640x320				v3.2	
	1	2	3	4	Time	Mem	1	2	3	4	Time	Mem
8	23.9	19.8	18.3	21.6	16.0	39.4	20.6	16.5	14.4	17.4	15.6	26.1
7	23.1	19.5	21.6	24.5	17.1	38.0	20.5	16.7	17.2	21.3	17.0	26.8
6	26.0	20.5	26.0	32.5	18.1	36.3	20.8	18.1	21.3	29.2	18.2	28.1
5	24.9	23.7	32.6	59.3	22.4	38.3	22.1	21.2	29.6	54.5	22.8	30.3
4	28.0	34.0	60.6		26.3	41.4	26.1	30.6	55.4		26.3	34.8
3	35.7	62.5			34.7	46.6	33.2	56.9			34.5	42.0
2	66.7				50.7	62.9	60.2				50.7	58.9
Cores	Section 2 cores 800x600				v3.2		Section 2 cores 1024x768				v3.2	
	1	2	3	4	Time	Mem	1	2	3	4	Time	Mem
8	20.6	15.9	14.3	16.6	15.5	24.1	20.1	16.3	13.8	16.6	15.5	21.6
7	21.0	16.5	16.8	21.0	16.9	25.4	20.2	16.2	16.7	20.6	16.8	24.0
6	21.2	17.9	21.1	29.4	18.1	26.9	20.4	17.7	20.8	29.2	18.1	25.5
5	22.6	21.1	29.5	54.7	22.9	28.8	22.1	20.9	29.4	54.7	22.8	27.9
4	26.4	30.5	55.6		26.3	33.3	25.7	30.4	55.4		26.1	32.7
3	33.8	57.0			34.5	41.3	33.7	56.8			34.3	40.3
2	60.2				50.9	57.5	59.8				51.2	56.8
Cores	Section 2 cores 1280x960				v3.2		Section 2 cores 1600x1200				v3.2	
	1	2	3	1	Time	Mem	1	2	3	1	Time	Mem
8	20.4	15.6	14.3	16.6	15.8	21.5	20.2	15.3	14.4	16.6	15.6	19.5
7	20.4	16.4	16.8	20.8	17.0	21.9	20.3	16.9	16.7	20.6	16.9	20.9
6	20.2	18.3	20.9	29.5	18.2	23.9	20.5	18.6	20.9	29.6	18.2	23.2
5	22.4	21.1	29.9	55.4	23.1	26.8	22.8	21.2	29.9	55.6	22.9	26.1
4	26.5	30.6	56.2		26.7	31.5	27.0	30.7	56.4		26.2	31.2

3	34.5	57.5			34.9	39.6		34.7	57.7			34.5	38.7
2	60.4				51.3	56.6		60.6				51.2	52.3

As seen in the Table 113 the version 4.1.2 of the TSM algorithm is faster than any other version especially when used with 2 CPUs. The speedup of the 4.1.2 version is not so significant but combined with the maximum memory advantages it offers it could replace the version 3.2.2. As seen in these tables the version 4.1.2 it is much faster than the memory efficient edition («Mem» column) of the 3.2.2 version. This means that this version is ideal when used with multi-CPUs hardware and large images.

In version 4.1 the TSM algorithm is split in two parallel section where each of them has its own memory consumption. The section 1 is creating Features images and the Filters Responses. The Features images data are locally created and released just after the calculation of corresponding Filter Responses but the Filter Responses are released by the section 2 after the level detection is completed. The section 2 uses the Filter Responses created in section 1 while the rest data it uses are locally created and released. The detection results are created inside the section 2 but they are calculated as global memory consumption because their size is dependent by the size of the Results Cache data structure. They also have a very small size after the Find v2.0 patch (Chapter 7.2).

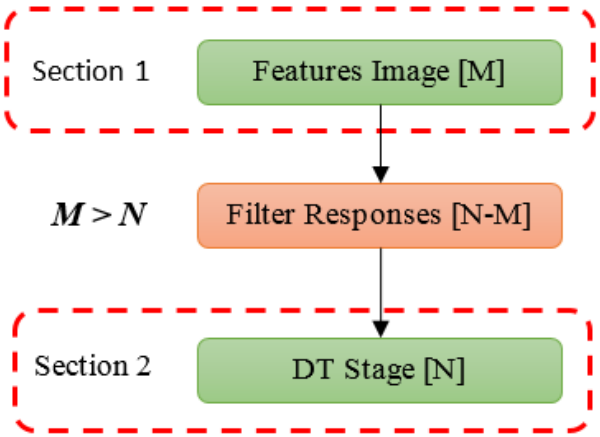


Figure 57 - TSM v4.1 Maximum Memory Sections Diagram

As shown in Figure 57 above the maximum memory consumption is actually formed by the level every section is and the distance between the two sections level. As greater this distance is so larger is the memory consumption. It is obvious that maximum memory consumption of both sections is reached when they execute its procedures at the top level of image pyramid, actually at their first run. On the other hand the Filter Responses of each level is always greater than the HOG image of the same level. This means that the Filters Responses that are created by the section 1 are overlapping the section's memory. When the section 2 finishes its processing on a

level then the Filters Responses of this level is released. So as it is sensible the maximum memory consumption of the algorithms is reached when the section 2 is on the first level and the section 1 on the last one. This way the all levels Filter Responses are hold in the memory increasing the algorithms maximum memory consumption like in the Figure 58.

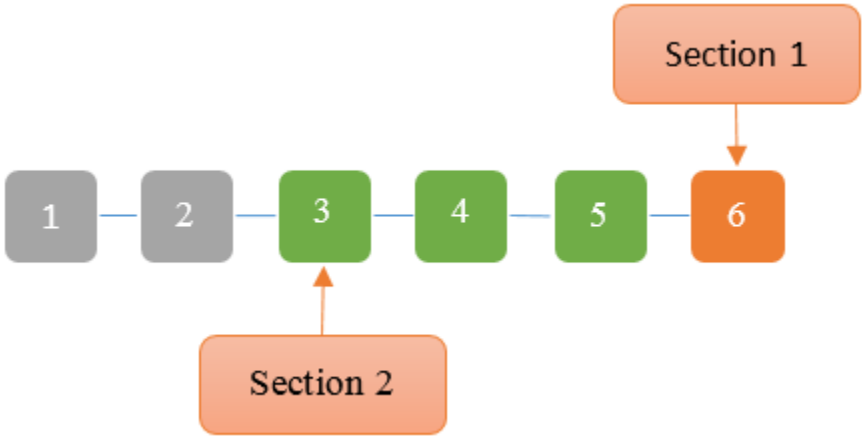


Figure 58 - TSM v4.1 Filters Responses Section Usage Diagram

To reduce the maximum memory consumption of this version the distance between the execution levels of every section has to be limited to the minimum. This can be achieved by obligating the section 1 to wait for the section 2 on a specific maximum distance. If the section 2 is faster or equal to the section 1 this problem does not appears. On the other hand if the section 1 is faster, then this problem is getting larger. This means that the best distance limit that would not affect the algorithm execution time is 1 meaning that when section 2 is processing the level *N* the section is processing the level *N+1*. This way the maximum memory consumption is equal to the following function result,

$$M_{\max} = M_{\text{Scection 2[1]}} + \mathbf{M}_{\text{Filter\_Responses[1-2]}} + \mathbf{M}_{\text{Features\_Image[2]}} \quad (20)$$

$$M_{\max} = M_{\text{Scection 2[1]}} + \mathbf{M}_{\text{Filter\_Responses[1-3]}} + \mathbf{M}_{\text{Features\_Image[3]}} \quad (21)$$

On the other hand in the real world execution of the TSM algorithm such an ideal synchronization between these two sections cannot be achieved. This means that the Filters Responses of N+1 Level must be already calculated when the Section 2 finishes the processing of the level N. This means that when the Section 2 is finishing the level N processing the Section 1 has to start processing the level N+2 in order to avoid at any chance that there is no possibility of Section 2 to wait for Section 1. The conclusion is that the distance limit that should be set in order to avoid section waiting and at the same time minimum maximum memory consumption is 2. The results using this options are as shown in Table 114 below,

Table 114 - TSM v4.1 Maximum Memory Consumption Comparison							
Version	**Cores**	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
v3.2.2	**1**	5.60 Mb	20.3 Mb	31.2 Mb	50.5 Mb	78.2 Mb	
	8	+347%	+375%	+379%	+383%	+385%	+374%
V2.2.2	**8**	+345%	+370%	+375%	+378%	+380%	+370%
V4.1.2	**8**	+151%	+162%	+163%	+165%	+165%	+161%
		14.0 Mb	53.2 Mb	82.2 Mb	134 Mb	208 Mb	

As seen in the Table 114 above, the version 4.1.2 of the algorithm is using almost the half memory of the rest parallelized versions and less than three times the memory a single core version uses.

Table 115 – TSM v4.1.2 vs v3.2.2	
Image	3200x2400
Time	89.8%
Memory	30.8%
Memory v3.2	2,427 Mb
Memory v4.1	749 Mb

To summarize, the version 4.1.2 of the algorithm is succeeding execution times similar to the rest parallelized version with a tiny, insignificant speedup. On the other hand the memory consumption of this version is about the half of the rest parallelized version. As seen in the Table 115, the TSM algorithm consumption is not large according to the available memory a multi-core hardware design usually dispose. The conclusion is that this version could probably be very useful when used for large size images where the memory consumption and the execution time are really high and small percentage differences can be noticeable sizes in real world. For example in a 3200x2400 pixels image the version 4.1.2 is 10.2% faster than the version 3.2.2.

8.9.4. TSM Algorithm Versions Comparison

After presenting the three version (v2.2.2, v3.2.2, v4.1.2) of the TSM algorithm using OMP parallelization technology a last survey has to be done. From these three version the most efficient is the 3.2.2 version.

Table 116 - TSM OMP Versions Execution Time Comparison (%)								
Version	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
V3.2.2 (Mem)	51.0	34.6	26.3	22.8	18.1	16.9	15.7	51.0
V3.2.2 (Time)	58.5	41.9	34.7	30.4	28.1	27.2	26.6	58.5
V4.1.2	61.5	34.2	26.6	21.6	18.5	17.1	15.0	61.5

The version 4.1.2 is a special version that has to be customized carefully according to the hardware resources offered in order to warranty its performance. As referred in chapter 8.9.3, this version is suitable for multiprocessors systems processing large size images, either wise the

profit it can offer is few in contrast to its instability of performance if it is not correctly customized.

Table 117 - TSM OMP Versions Max Memory Comparison (%)								
Version	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
V3.2.2 (Mem)	0	0	0	0	0	0	0	0
V3.2.2 (Time)	0	+54	+108	+162	+216	+270	+324	+378
V4.1.2	+118	+54	+54	+54	+108	+108	+108	+161

On the other hand the version 3.2.2 is stably faster than the version 2.2.2 and its memory consumption is insignificantly higher. It offer good performance without any further customization at any kind of hardware resource. Both these version can be used with the memory efficient editions described in chapters 8.9.1 and 8.9.2, but in our opinion there is no reason for doing that as the time efficient editions consume affordable memory related to the modern embedded systems capabilities. The usage of the Find v2.0 patch has a major role on that.

As every procedure of the TSM algorithm has a different parallelization efficiency the execution time distribution is different according to the CPU cores used at the parallelized versions. The execution time distribution of the version 3.2.2 of the TSM algorithm is shown in the Table 118 below.

Table 118 - TSM v3.2.2 Execution Time Distribution (%)								
CPU Cores	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Resize	0.95	1.43	1.63	1.96	2.28	2.86	3.13	3.52
	+0	+0.49	+0.69	+1.01	+1.33	+1.92	+2.19	+2.57
HOG	1.72	1.72	1.74	1.80	1.75	2.30	2.27	2.44
	+0	+0	+0.02	+0.08	+0.03	+0.58	+0.55	+0.72
Convolution	66.1	65.0	63.3	63.0	58.2	62.2	58.7	55.4
	+0	-1.16	-2.82	-3.17	-7.96	-3.92	-7.39	-10.7
Level Stage	31.2	31.8	33.2	33.1	37.6	32.4	35.6	38.2
	+0	+0.64	+2.01	+1.94	+6.43	+1.22	+4.40	+7.10

As seen in this table the Convolution procedure tends to participate less as the CPU cores number increases. This is very sensible as this procedure has very high efficiency at the parallelization technology and its execution time tends to reduce more than the rest procedures. That is why the other procedures tend to increase its participation. As seen in the same table the Resize (and Reduce) procedure and the Level stage participation in the algorithm execution time increased much more as these two participants efficiency is not as high as the two others when used too many CPU cores.

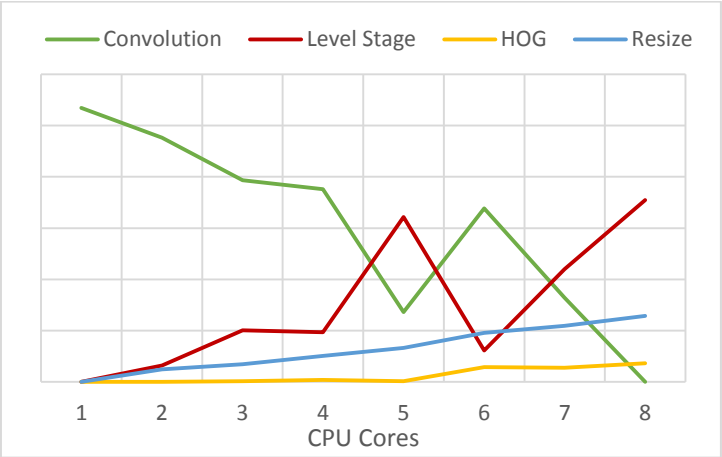


Diagram 84 - TSM Algorithm v3.2.2 OMP Execution Time Distribution Impact

9. TSM System Alternative Patches

In this chapter some patches over the TSM algorithm design are presented. These patches aim to improve the algorithm memory consumption or speed up the detection procedure or both. All of these improvements affect the algorithm detection performance making the algorithm less reliable or detection efficient but much faster. The goal is the ratio of speedup to the detection performance fall to be the greatest it can be achieved.

By studying the relations between the procedures execution time and the data structures (Filter Responses, Features Images, etc) of different levels of Features Pyramids, a stable ratio between them is discovered strictly connected with the image size. Using 5 as the value of the Interval parameter (default and proposed by the creators) the execution time needed for a levels' filters responses, for example, to be calculated is given by the function (22) . The same ratio occurs also when refer to memory consumption on the Filters Responses data structure, as shows the function (23).

Another ratio between the Features Pyramid levels is the ratio between a level and its following. Function (24) and (25) shows the relation between a level and the rest pyramid including it. The Functions (26) and (27) express the relation between a level and the following ones.

$$Time\left(level\right) = 0.75 \times Time(level - 1) \quad (22)$$

$$Mem\left(level\right) = 0.75 \times Mem(level - 1) \quad (23)$$

$$Time(level) = 0.25 \times \sum_{l=last}^{l=level} Time(l) \quad (24)$$

$$\mathrm{Mem}(level) = 0.25 \times \sum_{l=last}^{l=level} \mathrm{Mem}(l) \quad (25)$$

$$Time(level) = 0.33 \times \sum_{l=last}^{l=level+1} Time(l) \quad (26)$$

$$\mathrm{Mem}(level) = 0.33 \times \sum_{l=last}^{l=level+1} \mathrm{Mem}(l) \quad (27)$$

Using the functions **(22)** to (27) it is easy to predict the memory and execution time speedup can be achieved when the MinLevel parameter value changes. If for example the first level of the Features Pyramid is skipped the benefit that would be gained is a speedup of 25% at the Convolution stage. A 25% speedup would also cause the same speedup at the execution time of the whole algorithm as skipping a level in the Feature Pyramid, means also skipping a level in the Convolution and the detection procedures.

This conclusion is the basic idea behind most of the next alternative patches (Chapter 9.3 to 9.8) that try to speed up the algorithm by skipping the execution of some of the most time consuming procedures (Convolution, DT Stage). Skipping the convolution procedure in some levels of the features pyramid would cause a significant execution time saving, especially if these levels are from the top ones. The same applies to the Level stage execution where the detection procedures is applied.

9.1. NMS Limit

The NMS procedure is the one that selects the best detection results within a multiple set of values appear in the area of a detected human face, as described in chapter 5.10. In the owners implementation the NMS procedure sorts the detected results ascending and starting from the highest results checks the rest ones for overlaps, rejecting the overlapping ones. One detail in this implementation is that the NMS procedure after sorting the detections results it automatically rejects the lowest values without processing. The number of the detections results that are rejected is defined by the "NMS limit" parameter which default value is the 70% of the default Results Cache size. By examining the algorithms results setting this parameter value to zero, the algorithm accuracy change. The effect of this technique is double. Firstly, the majority of the detections in that area of lowest 70% are usually faulty detection, so this rejection protects the algorithm results from fake detection that decrease its reliability. On the other hand within this area sometimes correct detections exist, that unfortunately are rejected, decreasing the algorithm detection efficiency (Diagram 85). This sacrifice of correct detection against the fake ones is probably decided cause of the greater ratio of faulty against correct results in this area.

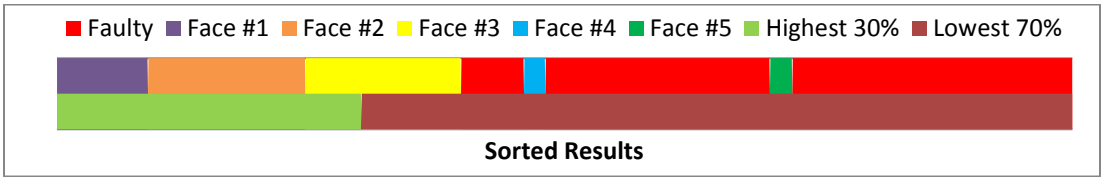


Diagram 85 - Results Cache NMS Limit Parameter Example

In the Table 119 below the results of testing the algorithm using a set of 205 images used by the creators of the algorithm also for testing, are shown. At this table, the detections of the algorithm according to the threshold variable value is presented when using the 99 filters model.

Table 119 - NMS Limit Results using 99 Filters Model								
Threshold	**-0.70**	**-0.65**	**-0.60**	**-0.55**	**-0.50**	**-0.45**	**-0.40**	**-0.35**
NMS Limit	**70%**							
Detected	89.1	87.8	86.1	82.9	82.5	82.1	80.8	-

Missed	10.9	12.2	13.9	17.1	17.5	17.9	19.2	-
Fake	21.8	15.3	10.6	8.27	4.93	3.27	1.82	-
Reliability	71.4	75.8	78.1	77.1	79.1	**79.8**	79.6	-
Fake/Real	15.0	10.9	4.25	1.84	1.57	1.08	0.82	-
NMS Limit	**0%**							
Detected	93.2	91.2	90.4	88.2	85.5	84.8	83.1	80.3
Missed	6.84	8.76	9.62	11.8	14.5	15.2	16.9	19.7
Fake	47.9	36.7	23.9	16.4	9.50	6.37	3.95	2.59
Reliability	50.2	59.6	70.4	75.2	78.4	80.2	**80.4**	78.7

As seen in the Table 119, the default threshold variable value produce totally different results on the algorithm when the "NMS Limit" parameter is set to 70%. As seen, the number of face detections is slightly increased when on the other hand the number of fake detections is increased by more than two times. The ratio between the fake and the real faces that the "NMS Limit" parameter rejects, changes according to the "Threshold" parameter value. The lower this parameter is the larger is this ratio. As seen in this table "Fake/Real" line, the ratio between fake and real faces rejected starts from 15 when the "Threshold" parameter value is -0.70 while it is less than one for "Threshold" parameter values greater than -0.45. As seen in this table the reliability of the algorithm with the "NMS Limit" parameter disabled is better as long as the "Fake/Real" face ratio is greater than 1.5. When the "Threshold" parameter value is greater than -0.50 the version of the algorithm without the use of the "NMS Limit" parameters succeeds better reliability.

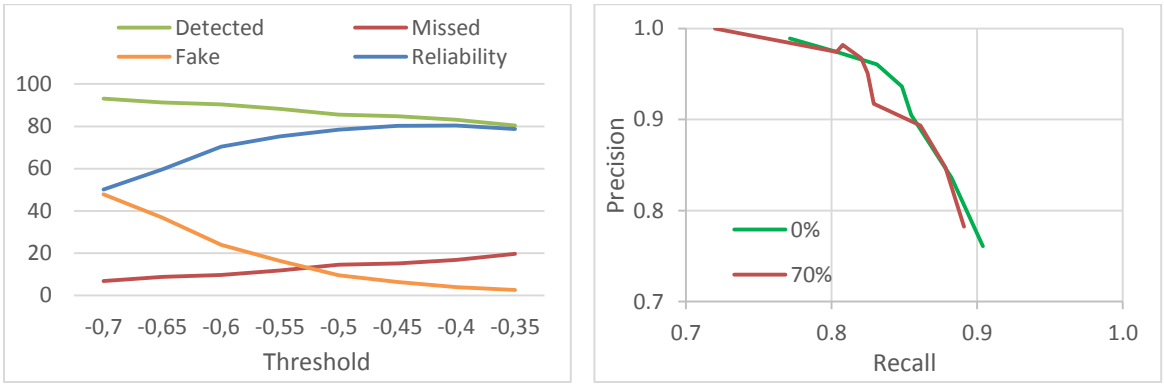


Diagram 86 - TSM Algorithm Performance with NMS Limit Disabled (99 Filters Model)

In the Diagram 86 above as is visible the Threshold parameter value is inversely analogous to the number of face detections. As it is increasing, the number of face detections (real or faulty) is decreasing. On the other hand as the Threshold variable is increasing the number of missed detections is increasing. The "Reliability" indicator is the one that reveals the best ratio between successful detections, missed detections and fake ones. The function that gives this indicator is the (28) below

$$\text{Re}\,liability = \frac{D_{correct}}{D_{missed} + D_{fake} + D_{correct}} \quad (28)$$

$$\text{Pr}\,ecision = \frac{D_{correct}}{D_{fake} + D_{correct}} \quad (29)$$

$$\text{Recall} = \frac{D_{correct}}{D_{missed} + D_{correct}} \quad (30)$$

As far as the 146 filters Model the results of the same testing procedure are shown in the Table 120 below.

Table 120 - NMS Limit Results using 146 Filters Model							
Threshold	**-0.70**	**-0.65**	**-0.60**	**-0.55**	**-0.50**	**-0.45**	**-0.40**
NMS Limit	**70%**						
Detected	88.5	86.3	84.2	82.5	78.8	77.4	-
Missed	11.5	13.7	15.8	17.5	21.2	22.6	-
Fake	10.8	7.13	5.06	3.26	2.12	1.36	-
Reliability	79.9	**81.0**	80.6	80.2	77.5	76.5	-
Fake/Real	2.48	1.45	0.79	0.62	0.33	0.33	-
NMS Limit	**0%**						
Detected	93.4	90.6	88.2	85.3	81.4	79.3	76.5
Missed	6.62	9.40	11.8	14.7	18.6	20.7	23.5
Fake	19.7	12.4	8.02	5.00	3.05	2.11	1.65
Reliability	76.0	80.3	**81.9**	81.6	79.4	77.9	75.5

In the Table 120 above the impact of disabling the “NMS Limit” parameter in the NMS procedure is much lower than in the 99 filters model. This is because the 146 filters model is more accurate as it uses more and better trained filters for the landmark detection. This is also depicted in the relation between the fake and the real face detections inside the 70% of results rejected by the “NMS Limit” parameter. This relation is 2.5 fake detections for every correct one when in the 99 filters model this relation is 15 when the “Threshold” parameter value is -0.70. What is interesting is that the “NMS Limit” parameter has negative impact on the algorithm results when values lower than -0.65 on the “Threshold” parameter are used.

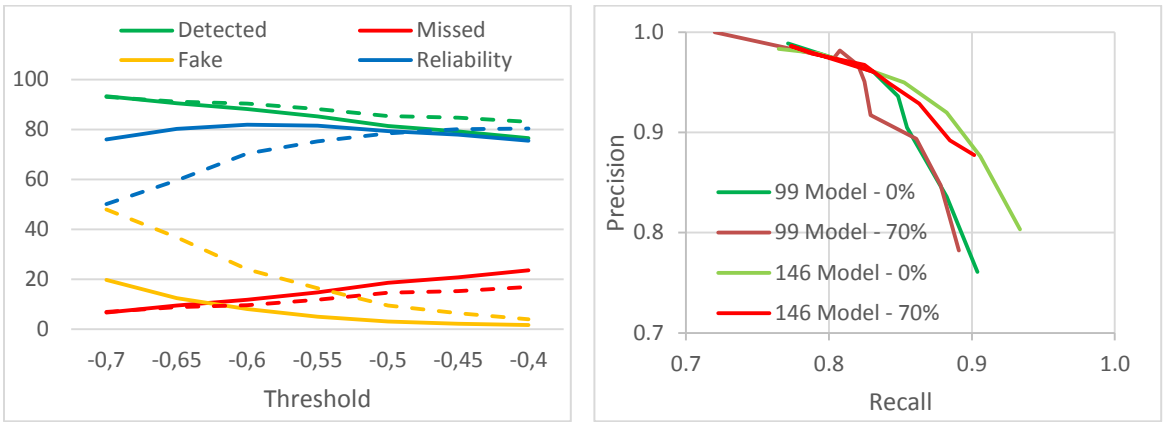


Diagram 87 - TSM Algorithm Performance with NMS Limit Disabled (Both Models)

At the Diagram 87 above the effect of different values of the Threshold parameter is shown for both models. This effect is the same as described in the corresponding paragraph for the 99 filters Model. It is sensible that when the Threshold parameter is reducing the number of real and also fake face detection to increase and the opposite when increasing. The only difference is that the 146 filters model is more accurate creating much less fake detections. On the other hand the detection efficiency is better for the 99 filters model as it is more abstract. The interesting point of those two models is that as the "Threshold" parameter value is increasing their reliability is converging. This is very encouraging as the 99 filters model is much faster than the 146 filters one and it can succeed pretty good reliability and detection efficiency than can make it more preferable.

As referred in this chapter's paragraphs, the "NMS Limit" parameter has a positive impact in the algorithm's results when the "Threshold" parameter value is low at both Models. The disabling of the "NMS Limit" parameter is not a wise decision if it is not replaced by another method that would be able to increase the algorithms reliability by rejecting the fake faces. This method is appose in the next chapter, chapter 9.2.

9.2. Dynamic Threshold

A static value on the Threshold parameter might not be always efficient. Sometimes in a sharp image the faces within it can produce many high-score values much higher than the Threshold parameter value. In addition fake faces may be detected with high-score values much lower than the real faces ones but still over the Threshold parameter limit. In our implementation a new proposal to this problem is presented using a dynamic Threshold value. By examining the fake results values a ratio between the correct detection and the fake one was discovered. Usually the better the sharpness of an image is the easier for an image processing algorithm to have accurate results. A common technique for making an image processing algorithm more independent of this parameter is the normalization method. A similar technique is the one we

create for that purpose. First of all, the NMS procedure does not reject the lowest detection results using the "NMS Limit" parameter but in addition the results are compared among themselves. The results that their values are less than the Ratio parameter value of the highest one are rejected.

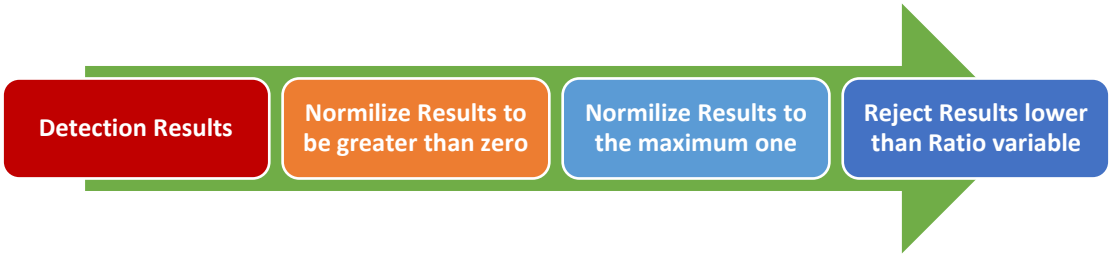


Figure 59 - Dynamic Threshold Patch Execution Flow Diagram

On the Table 121 below the results of the Face Detection TSM algorithm using different values of the Ratio and Threshold parameters for the 99 filters Model is shown. As seen in the table the most critical parameter is the reliability of the algorithm. As greater it is the more reliable the algorithm is. A second important parameter is the number of detected faces as the more face detection the algorithm achieves the more efficient it is. The desirable result is the algorithm to detect as more faces it can with the maximum percentage of reliability.

Table 121 - Dynamic Threshold Patch Results with 99 Filters Model

Threshold	Ratio	Detected	Missed	Fake	Reliability	Precision	Recall
-0.70	**Original**	**89.1**	**10.9**	**21.8**	**71.4**	**0.78**	**0.89**
	0%	93.2	6.84	47.9	50.2	0.52	0.93
	5%	91.2	8.76	33.7	62.3	0.66	0.91
	10%	**90.6**	**9.40**	**20.5**	**73.5**	**0.80**	**0.91**
	15%	87.8	12.2	13.7	77.1	0.86	0.88
	20%	86.1	13.9	7.99	80.1	0.92	0.86
	25%	83.8	16.2	4.39	**80.7**	0.96	0.84
	30%	81.4	18.6	1.55	80.4	0.98	0.81
	35%	79.3	20.7	1.07	78.6	0.99	0.79
-0.65	**Original**	**87.8**	**12.2**	**15.3**	**75.8**	**0.85**	**0.88**
	0%	91.2	8.76	36.7	59.6	0.63	0.91
	5%	90.8	9.19	22.4	71.9	0.78	0.91
	10%	**88.2**	**11.8**	**14.7**	**76.6**	**0.85**	**0.88**
	15%	86.3	13.7	8.18	80.2	0.92	0.86
	20%	84.2	15.8	5.29	80.4	0.95	0.84
	25%	81.8	18.2	1.54	**80.8**	0.98	0.82
	30%	79.5	20.5	1.06	78.8	0.99	0.79

-0.60	**Original**	**86.1**	**13.9**	**10.6**	**78.1**	**0.89**	**0.86**
	0%	90.4	9.62	23.9	70.4	0.76	0.90
	5%	88.0	12.0	14.3	76.7	0.86	0.88
	10%	**85.7**	**14.3**	**8.86**	**79.1**	**0.91**	**0.86**
	15%	84.8	15.2	5.48	80.9	0.95	0.85
	20%	82.5	17.5	1.78	**81.3**	0.98	0.82
	25%	79.9	20.1	1.06	79.2	0.99	0.80
-0.55	**Original**	**82.9**	**17.1**	**8.27**	**77.1**	**0.92**	**0.83**
	0%	88.2	11.8	16.4	75.2	0.84	0.88
	5%	**85.9**	**14.1**	**9.26**	**79.0**	**0.91**	**0.86**
	10%	**85.3**	**14.7**	**6.12**	**80.8**	**0.94**	**0.85**
	15%	**82.9**	**17.1**	**2.51**	**81.2**	**0.97**	**0.83**
	20%	80.6	19.4	1.05	79.9	0.99	0.81
-0.50	**Original**	**82.5**	**17.5**	**4.93**	**79.1**	**0.95**	**0.82**
	0%	85.5	14.5	9.50	78.4	0.90	0.85
	5%	**84.8**	**15.2**	**6.59**	**80.0**	**0.93**	**0.85**
	10%	**82.9**	**17.1**	**3.48**	**80.5**	**0.97**	**0.83**
	15%	80.3	19.7	1.05	79.7	0.99	0.80
-0.45	**Original**	**82.1**	**17.9**	**3.27**	**79.8**	**0.97**	**0.82**
	0%	**84.8**	**15.2**	**6.37**	**80.2**	**0.94**	**0.85**
	5%	**82.9**	**17.1**	**3.96**	**80.2**	**0.96**	**0.83**
	10%	81.2	18.8	1.55	80.2	0.98	0.81
	15%	78.8	21.2	0.81	78.3	0.99	0.79
-0.40	**Original**	**80.8**	**19.2**	**1.82**	**79.6**	**0.98**	**0.81**
	0%	**83.1**	**16.9**	**3.95**	**80.4**	**0.96**	**0.83**
	5%	**81.4**	**18.6**	**2.06**	**80.0**	**0.98**	**0.81**
	10%	78.8	21.2	0.81	78.3	0.99	0.79
	15%	77.1	22.9	0.82	76.6	0.99	0.77

As seen in the Table 121 the Ratio variable improves the algorithm's performance at about 1.5% as far as its Maximum Reliability and 4.1% its maximum successful detections. At every value of the Threshold parameter the Dynamic Threshold patch increases the reliability and efficiency indexes about 1-2%. These increments on the TSM algorithm performance indexes is not very significant as the numbers reveal but they show that the Dynamic Threshold patch is an successful substitute of the "NMS Limit" parameter.

As the Threshold parameter value is increasing the Ratio techniques does not seems to offer any positive results, but on the other hand when low values are set to Threshold parameter, a small value of the Ratio one offers a much better performance to the algorithm results. The

conclusion is that a Ratio of 0.05 to 0.15 can be usefully used when low Threshold parameter value is used aiming on high face detections rates, as shown in Diagram 88. As seen in the Diagram 88, the results of the algorithm when the Ratio parameter is used is better in all indexes. In the Diagram 88 the impact of the Ratio technique in the results when used with -0.65 (continuous) and -0.60 (dashed) Threshold variable values is shown.

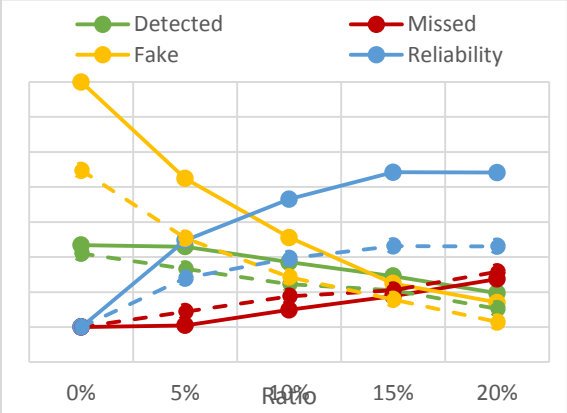


Diagram 88 - Dynamic Threshold Patch Impact on Threshold Low Values (99 Filters Model)

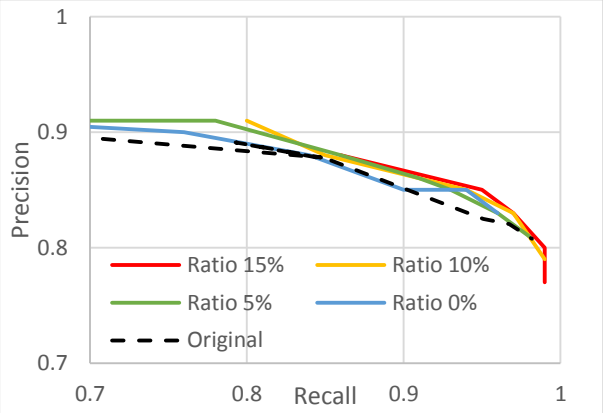
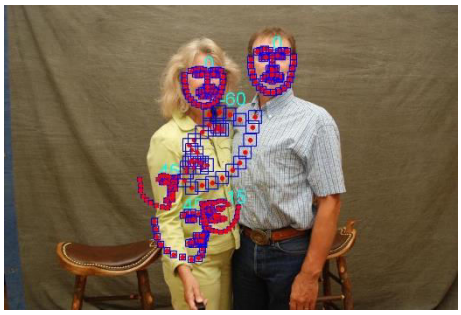
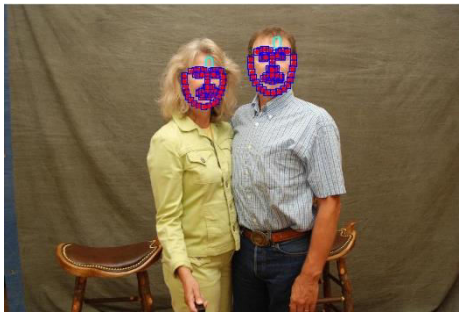


Diagram 89 - Dynamic Threshold Patch Performance Impact (99 Filters Models)

As seen in Diagram 88, the results reliability is strongly increased when the “Ratio” parameter is set to 5% while the face detection rate is not actually reduced. As described above this phenomenon is smaller when used with -0.60 Threshold parameter value than with -0.65.



Threshold = -0.65



Threshold = -0.65, Ratio = 0.15

Figure 60 - Dynamic Threshold Patch Performance Examples

As described above the usage of the Ratio parameter is a useful technique giving the algorithm a portion of stability as it rejects the fake face detection even when the Threshold parameter value is lower than it should. As seen in the Table 121 a Ratio value of 5-15% gives always a satisfactory result.

As far as the 146 filters Model the results applying the Ratio parameter is shown in the Table 122 below.

Table 122 - Dynamic Threshold Patch Results with 146 Filters Model							
Threshold	**Ratio**	**Detected**	**Missed**	**Fake**	**Reliability**	**Precision**	**Recall**
	Original	**88.5**	**11.5**	**10.8**	**79.9**	**0.89**	**0.88**
	0%	93.4	6.62	19.7	76.0	0.80	0.93
	5%	**92.3**	**7.69**	**12.0**	**82.0**	**0.88**	**0.92**
-0.70	**10%**	**89.5**	**10.5**	**8.11**	**83.0**	**0.92**	**0.90**
	15%	87.6	12.4	5.96	83.0	0.94	0.88
	20%	85.7	14.3	4.75	82.2	0.95	0.86
	25%	82.1	17.9	3.52	79.7	0.96	0.82
	Original	**86.3**	**13.7**	**7.13**	**81.0**	**0.93**	**0.86**
	0%	90.6	9.40	12.4	80.3	0.88	0.91
-0.65	**5%**	**89.1**	**10.9**	**7.95**	**82.7**	**0.92**	**0.89**
	10%	**87.4**	**12.6**	**5.76**	**83.0**	**0.94**	**0.87**
	15%	85.3	14.7	4.77	81.8	0.95	0.85
	Original	**84.2**	**15.8**	**5.06**	**80.6**	**0.95**	**0.84**
	0%	**88.2**	**11.8**	**8.02**	**81.9**	**0.92**	**0.88**
-0.60	**5%**	**86.3**	**13.7**	**5.16**	**82.4**	**0.95**	**0.86**
	10%	**84.6**	**15.4**	**4.35**	**81.5**	**0.96**	**0.85**
	15%	81.6	18.4	3.05	79.6	0.97	0.82
-0.55	**Original**	**82.5**	**17.5**	**3.26**	**80.2**	**0.97**	**0.82**

	0%	**85.3**	**14.7**	**5.00**	**81.6**	**0.95**	**0.85**
	5%	**83.1**	**16.9**	**3.47**	**80.7**	**0.97**	**0.83**
	10%	81.2	18.8	2.31	79.7	0.98	0.81
-0.50	**Original**	**78.8**	**21.2**	**2.12**	**77.5**	**0.98**	**0.79**
	0%	**81.4**	**18.6**	**3.05**	**79.4**	**0.97**	**0.81**
	5%	**80.8**	**19.2**	**2.33**	**79.2**	**0.98**	**0.81**
	10%	**78.8**	**21.2**	**1.60**	**77.8**	**0.98**	**0.79**

By the application of the Ratio parameter, the detection efficiency and the reliability of the algorithm is increased as seen in the Table 122 overcoming the results of the original version. This is very important as by reducing the Threshold parameter value to the -0.70 the algorithm face detection efficiency is increasing and by using the Ration parameter value to a 10 percentage the algorithm results are better in all indexes. The same thing is observed when the threshold variable is set to -0.65 and the ratio one to 10%.

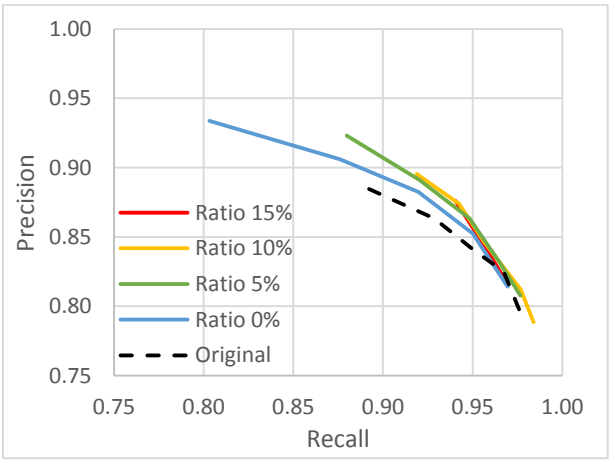


Diagram 90 - Dynamic Threshold Patch Performance Impact (146 Filters Model)

As a conclusion the disabling of the “NMS Limit” parameter of NMS procedure revealed a number of valid face detections but also a larger number of invalid ones. The number of fake face detections can be reduced by increasing the value of the “Threshold” variable but this change cause also a small decrement of the algorithms detection efficiency. At last the Dynamic Threshold patch using the Ratio parameter is the one that can replace the “NMS Limit” one in the NMS procedure. The Ratio parameter is more efficient and also fairer as it is not dependent by the number of detections but by the detected faces sharpness. The results tables are the proof.

9.3. Interval

The greatest consumer of execution time of the algorithm is the Convolution stage (Diagram 30, Chapter 6.20) where the Filter Responses are produced. The Convolution stage duration is depended on two parameters. The first parameter is the number of filters used by the model parts and the second one is the number of levels the features pyramid has. The larger the feature pyramid is the more time is needed for the convolution procedure. The length of the features pyramid is also affecting the number of the Level stage calls that uses the Distance Transformation stage which is the second greatest time consumer (Diagram 30, Chapter 6.20). The Convolution and the DT stage consumes about the 96% of the whole algorithm execution.

In chapter 7, the Short Pyramid patch reduced the number of levels in an important amount, as explained. Although the number of levels reduced is closely to its half, the levels of the features pyramid that was removed was the latest. The latest levels of the feature pyramid have actually the smallest sizes and that was the reason that even if the length of the features pyramid was reduced about to its half, the reduction of the execution time of the algorithm was reduced for only about 4%. It would be a very pleasant if there was a way of reducing the number of levels of the features pyramid removing levels from the top.

Removing levels from the top of the Feature pyramid would remove the ability from the algorithm of detecting small faces within the images. If for example the first level of the feature pyramid is removed then the ability of the algorithm to detect faces in the size of 100 pixels high within the image would be greatly reduced. If the algorithm is used for an application than does not tries to detect very small face within large images then it would not be a problem but this is the subject of chapter 9.4.

In this chapter, a method of reducing the number of the top levels of the features pyramid of the algorithm aiming on reducing its execution time is appose. In the chapter 5.5 the “Interval” parameter was introduced. This parameter determines the number of scaled images, inside the features pyramid, between two images with scale ratio of two. In this chapter the impact of the reduction of this parameter value is going to be examined.

The TSM algorithm creators set the “Interval” parameter value to five as the default value. Changing this parameter value to four would change the whole features pyramid images scale and this is why the impact of this change cannot be calculated using the execution time of the algorithm when used with other “Interval” parameter values. In the Table 123 the number of the levels of the features pyramid according to the Interval parameter is

Table 123 - FP Levels per Interval

Image Size	Interval		
	5	**4**	**3**
320x240	7	6	4
640x480	12	10	7
800x600	13	11	8
1024x768	15	12	9
1280x960	17	14	10

presented. What is very important on this change is not only the execution time gain but its impact on the algorithms detection efficiency and its reliability.

The algorithms execution time when the "interval" parameter is reduced to four is shown in the Table 124 when this change is applied to the version 3.2.2. As seen in this table, the execution time gained is noticeable, about 20%. If the Interval parameter is further reduced to three the execution time is almost twice reduced to 37.8%. These reductions is very positive but they have an impact on the algorithm's performance as shown in the next table (Table 125).

Table 124 - TSM v3.2.2 Interval Patch Execution Time (%)						
Interval	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
4	-28.3	-19.8	-18.5	-18.7	-17.7	-20.6
3	-45.4	-37.0	-35.9	-35.7	-35.2	-37.8

Table 125 - TSM Algorithm Interval Patch Performance (%)						
Threshold	**-0.65**	**-0.60**	**-0.55**	**-0.50**	**-0.45**	**-0.40**
Interval	**5**					
Detected	91.2	90.4	88.2	85.5	84.8	83.1
Missed	8.76	9.62	11.8	14.5	15.2	16.9
Fake	36.7	23.9	16.4	9.50	6.37	3.95
Reliability	59.6	70.4	75.2	78.4	80.2	**80.4**
Interval	**4**					
Detected	86.3	84.0	82.3	80.1	77.4	75.0
Missed	13.7	16.0	17.7	19.9	22.6	25.0
Fake	27.9	17.3	11.5	7.18	3.21	1.13
Reliability	64.7	71.5	74.3	**75.5**	75.4	74.4
Interval	**3**					
Detected	72.0	69.9	66.9	65.4	62.8	61.1
Missed	28.0	30.1	33.1	34.6	37.2	38.9
Fake	19.0	10.7	5.44	3.16	1.01	0.35
Reliability	61.6	**64.5**	64.4	64.0	62.4	61.0

In the Table 125 above the detection efficiency and the reliability results are not as positive as the execution time gain. As seen the reduction of the Interval parameter to four causes 5 to 10 percent reduction of the algorithms detection efficiency while its reliability is also low. The results are even worst when the interval variable is set to three where the algorithms detection efficiency and reliability is getting lower than 70%.

The detection efficiency and the reliability of the algorithms seems to reduce a lot when the interval parameter value is change and is reducing. On the other hand the execution time

speedup gained is significant reaching the 20%. Reducing the Interval parameter value seems to be a risk as it makes the algorithm less reliable and efficient and it does not seems to worth it. This technique reduces the algorithm efficiency so much that it would not be advisable to be used in combined to other patches presented in this thesis. On the other hand, other patches that do not affect significantly the algorithms efficiency can be combined offering similar execution time speedup without making the algorithm unreliable.

9.4. Canvas

As referred in the previous chapters, the features pyramid levels consumes time according to their image size. Using the “Interval” parameter set to five, the time needed for the first level of the features pyramid is about the 25% of the time needed for all the features pyramid levels. The time needed for the next level is about 19% etc. To sum up, the first interval set of levels (1 to interval) needs about the 75% of the whole features pyramid levels. All these leads to the conclusion that if the algorithm skipped even one level from the top of the features pyramid this would significantly reduce the algorithm’s execution time.

In this chapter one method for speeding up the algorithm is presented sacrificing a part of its reliability and detection efficiency, but controllable. In this method two new parameters are imported in the algorithm implementation that gives the opportunity of sacrificing the ability of the algorithm to detect very small or very large faces within the image but gaining time consumption.

These two parameters are the “Min Face” and the “Max Face” defining the minimum and the maximum face size according to the image’s size that the algorithm would try to detect. This way the levels used for detecting faces larger or smaller than this percentages would be skipped by the algorithm.

The algorithm detects large faces in the latest levels of the image pyramid. This means that when the “Max Face” parameter is reducing the algorithm skips levels ascending starting from the last level. As far as the time consumption profit of this change will not be great as the latest a level is in the features pyramid, the less execution time needs. This way when the “Max Face” parameter is reduced the execution time saved would be too few in contrast to the reliability that it may lose. For that reason the “Max Face” parameter should be reduced only if the algorithm is used in applications that do not try to detect faces conceiving large part of the image.

Figure 61 - Faces Size Within the Image Examples

As far as the small faces, these are detected in the top levels of the features pyramid so in the largest features images. In contrast to the "Max Face" parameter, the "Min Face" one is much more significant as far as the saving of execution time. Even if the algorithm skips one level from the top, its execution time is reduced by 25%. This is a very important reduction. So, if the algorithm is used for detecting large faces within images, then it would be very helpful if the "Min face" parameter would be increased in order some of the top levels of the features pyramid would be skipped and the algorithm could gain a significant speed up.

In previous chapters has been referred that the algorithm can detect faces with the minimum size of 100 pixels. This makes it obvious that the "Min Face" parameter does not always has a real effect to the algorithm. As the Table 126 shows the minimum face size that the algorithm can detect in specific image sizes is larger as the image size is

Table 126 - TSM Minimum Detectable Face (%)

Image Size	99 Model	146 Model
320x240	41.7	20.8
640x480	20.8	10.4
800x600	16.7	8.33
1024x768	13.0	6.51
1280x960	10.4	5.21

smaller. As seen in this table when the image height is 240 pixels the algorithm can detect faces larger than the 41.7% of the image height. This concludes that if the "Min Face" parameter is set to 25% it would have no impact to the algorithms execution. In the Table 126 below the faces size that can be detected by the features pyramid's levels are shown.

The execution time saving for every level of the pyramid skipped using these parameters are shown in the Table 127 below. As seen in this table, setting the "Min Face" parameter value to 30% can gain a speedup up to 68.4% for a very large image (1028x960) and at least 25% for a small one (640x480). This is a very important speedup that can be easily used when the image classification is known. If for example the algorithm is used for images with a close capture of faces then the top levels of the features pyramid are useless, while if the face capture is from large distance, then the bottom levels are the ones that are useless.

Table 127 - Max/MinFace Parameters Execution Time Profit (%)								
	Detectable Face Size per Level (%)					Profit Approach		
Level	320x240	640x480	800x600	1024x768	1280x960	Level	MaxFace (Bot→Top)	MinFace (Top→Bot)
1	45.8	24.1	19.9	15.9	13.1	25.0	74.4	-0%
2	54.2	30.7	26.3	21.7	18.3	18.8	55.7	-25.0
3	62.5	37.3	32.7	27.5	23.6	14.1	41.6	-43.8
4	70.8	43.9	39.1	33.3	28.9	10.5	31.1	-57.8
5	79.2	50.5	45.5	39.1	34.1	7.91	23.2	-68.4
6	87.5	57.1	51.9	44.9	39.4	5.93	17.2	-76.3
7	95.8	63.7	58.3	50.7	44.7	4.45	12.8	-82.2
8		70.3	64.7	56.5	49.9	3.34	9.45	-86.7
9		76.9	71.2	62.3	55.2	2.50	6.94	-90.0
10		83.5	77.6	68.1	60.5	1.88	5.07	-92.5
11		90.1	84.0	73.9	65.7	1.41	3.66	-94.4
12		96.7	90.4	79.7	71.0	1.06	2.60	-95.8
13			96.8	85.5	76.3	0.79	1.81	-96.8
14				91.3	81.6	0.59	1.22	-97.6
15				97.1	86.8	0.45	0.77	-98.2
16					92.1	0.33	0.44	-98.7
17					97.4	0.25	-0%	-99.0
Range	±4.2	±3.3	±3.2	±2.9	±2.6			

In the Figure 61 the size of multiple faces according to the image size are shown. As seen in these images a face must be too zoomed in to the camera to take place in a large part within the images as happens in the bottom left image in Figure 61. The most common distance can make a face holding the 10% to 50% of the image height and that makes it difficult to raise the "Min

Level" parameter value because it raises the possibility of affecting the algorithms detection efficiency. What might worth a try is to increase the value of the "Min Level" parameter at such a small quantity that the algorithm could at least skip the top level of the features pyramid and gain a 25% time speedup.

On the other hand the "Max Face" parameters as seen in these images can easily be reduced for a significant amount as it is very unusual for a face to be captured at such a close zoom that it can conceive larger than 70% or at most 80%. The bad news is that the levels of the features pyramid that could be skipped by this parameter are the smallest ones and the speedup the algorithm can gain very insignificant. Although, it seems that a "Max Face" parameter value equal to 80% is very possible to have a tiny, insignificant impact to the algorithm's detection efficiency.

The heartening fact is that as the pyramid size is decreasing, all the levels of the pyramid execution time participation is increasing. As seen in the Table 128 below the last level of the pyramid is using the 5.78% of the whole pyramid detection time on a 7 levels while in a 17 levels pyramid only 0.33%. This means that a MaxFace Parameter value set to 80% can offer about 13% reduction on the execution time of a 320x240 image and about 5% on a 640x480 image when the corresponding reduction is less than 2% on a 1280x960 image. On the other hand the top levels participation is always very significant and skipping them can be proved very useful in large size images as mentioned before.

Table 128 - Max/MinFace Execution Time Profit per Image Size

	320x240		640x480		800x600		1024x768		1280x960	
Level	**Face**	**Profit**	**Face**	**Profit**	**Face**	**Profit**	**Face**	**Profit**	**Face**	**Profit**
1	45.8	27.4	24.1	24.5	19.9	24.4	15.9	24.2	13.1	24.1
2	54.2	21.3	30.7	18.9	26.3	18.6	21.7	18.5	18.3	18.3
3	62.5	16.3	37.3	14.3	32.7	14.2	27.5	14.0	23.6	13.9
4	70.8	12.4	43.9	11.0	39.1	10.8	33.3	10.7	28.9	10.6
5	79.2	9.56	50.5	8.28	45.5	8.29	39.1	8.14	34.1	8.06
6	87.5	7.24	57.1	6.31	51.9	6.25	44.9	6.16	39.4	6.11
7	95.8	5.78	63.7	4.91	58.3	4.82	50.7	4.74	44.7	4.70
8			70.3	3.75	64.7	3.66	56.5	3.59	49.9	3.57
9			76.9	2.87	71.2	2.81	62.3	2.78	55.2	2.74
10			83.5	2.20	77.6	2.20	68.1	2.13	60.5	2.06
11			90.1	1.67	84.0	1.66	73.9	1.60	65.7	1.57
12			96.7	1.33	90.4	1.28	79.7	1.23	71.0	1.23
13					96.8	0.99	85.5	0.96	76.3	0.94
14							91.3	0.74	81.6	0.71
15							97.1	0.57	86.8	0.55

16									92.1	0.42
17									97.4	0.33
Range	±4.2		±3.3		±3.2		±2.9		±2.6	

9.5. 68 Filters Model

One of the major advantages of this TSM algorithm as referred in the Related Work chapter (Chapter 3) is its ability to detect faces until the viewing angle of ±90 degrees. Many related algorithms used for face detection are trained to detect centered faces. As described in chapter 4, the algorithm is using parts based mixtures of trees to detect faces and estimate their pose. For the detection and pose estimation of faces in the area of ±60 to ±90 degrees viewing angle the algorithm is using six pose trees which use 78 filters (146 filters model). These 78 filters are compressed to 39 ones on the 99 filters model. The existence of these extra pose trees and filters cost to the TSM algorithm extra time and memory consumption as they extend the execution time of the Convolution and Level stage and also enlarge the Filters Responses arrays list which is one of the main participants on the maximum memory consumption formation.

As far as the memory consumption of the TSM algorithm, the impact of removing the edge pose trees is as shown in Table 129 below. The Filters Responses data memory is reduced to its 68/99 as it is sensible. The impact of this reduction ends to a reduction of about 12% to the total maximum memory consumption.

As far as the execution time consumption, the impact of the 68 filters model to the TSM algorithm is reaching the -31% as the Table 129 shows. This is a significant reduction as the algorithm needs the two thirds of the time needed in the 99 filters model when the 68 one is used.

Table 129 - TSM v3.2.2 68 Filters Model Performance (Compared to 99 Model)						
	320x240	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
Memory	4.84 Mb	17.6 Mb	27.2 Mb	44.0 Mb	68.3 Mb	
	-11.5%	-12.2%	-12.3%	-12.4%	-12.4%	-12.2%
Time	-31.2%	-31.3%	-31.1%	-31.2%	-30.9%	-31.1%

The disadvantage if using the 68 filters model is the fact that the algorithm is now able to detect faces of ±45o viewing angle. The performance of the algorithm is not actually change as far as the 68 filters poses. Any difference in the reliability and detection efficiency is shown in the Table 130 below is caused by the removal of the 39 filters poses from the aggregation of the testing results. As seen in the results table below the algorithm results in faces of ±45o viewing angle are even better than the 99 filters model. Its performance fall starts when it is asked to

detect faces in greater viewing angles where the detection efficiency is reducing, luring its reliability as the fake detection results are stable.

Table 130 - TSM 68 Filters Model Results								
Threshold	**-0.70**	**-0.65**	**-0.60**	**-0.55**	**-0.50**	**-0.45**	**-0.40**	**-0.35**
	-45° to +45°							
Detected	98.2	94.6	90.4	88.3	85.8	82.6	79.5	76.9
Missed	1.81	5.44	9.59	11.7	14.2	17.4	20.5	23.1
Fake	29.8	22.0	15.1	9.31	4.89	3.04	1.29	0.00
Reliability	69.3	74.6	77.9	81.0	**82.1**	80.6	78.7	76.9
	-60° to +60°							
Detected	91.1	87.7	83.9	82.0	79.6	76.7	73.8	71.4
Missed	8.89	12.3	16.1	18.0	20.4	23.3	26.2	28.6
Fake	29.8	22.0	15.1	9.31	4.89	3.04	1.29	0.00
Reliability	65.7	70.3	73.0	75.6	**76.4**	74.9	73.1	71.4
	-75° to +75°							
Detected	86.5	83.3	79.7	77.9	75.6	72.8	70.1	67.8
Missed	13.5	16.7	20.3	22.1	24.4	27.2	29.9	32.2
Fake	29.8	22.0	15.1	9.31	4.89	3.04	1.29	0.00
Reliability	63.3	67.5	69.8	72.1	**72.7**	71.2	69.5	67.8
	-90° to +90°							
Detected	81.0	78.0	74.6	72.9	70.7	68.2	65.6	63.5
Missed	19.0	22.0	25.4	27.1	29.3	31.8	34.4	36.5
Fake	29.8	22.0	15.1	9.31	4.89	3.04	1.29	0.00
Reliability	60.3	63.9	65.8	67.8	**68.2**	66.7	65.0	63.5

9.6. Detection Components

As realized already by the algorithm characteristics, the main execution time consumers are the Convolution and the Component detection processing. In this chapter a technique that can reduce the time consumption of the Component stage.

During the component detection procedure the nearby components produce similar high-score values as referred in 5.1. This means that if the component 7 (0° viewing angle) is removed by the model a corresponding face can continue be detected by the components next to it (6 and 8, ±15°). As the components diverge from the same component the detection results are reducing as shown in the Diagram 91 below.

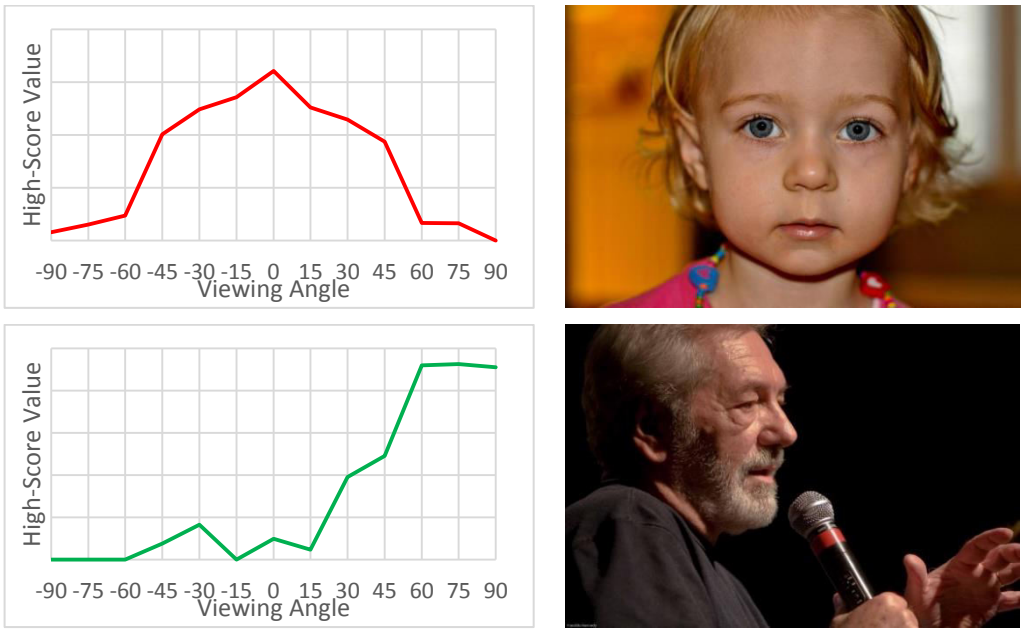
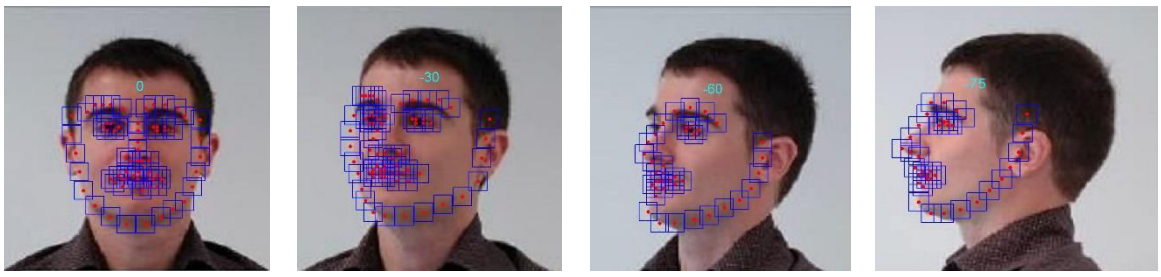


Diagram 91 - Components High-Score Results Example

In the 99 filters model the seven components in the middle refers to the face poses from -45 to 45 degrees and they all use the same filters. The distance between the landmarks is the criteria for the pose estimation. On the other hand the six components at the edges, refer to the -90 to -60 and 60 to 90 degrees are using a complex of filters, half of which are also used in the middle components. This is why in Diagram 91 above the face detection of a face on 0 degrees creates such lower scores on the edge components while in the middle ones the scores are close. In the Diagram 91 the detection scores of a face in the angle of 75 degrees is presented. Only the components close to the left edge components of the middle components succeed a detection but even though their scores are much lower than the ones of the right edge components. This is caused due to the common filters used by both 68 and 39 parts components.



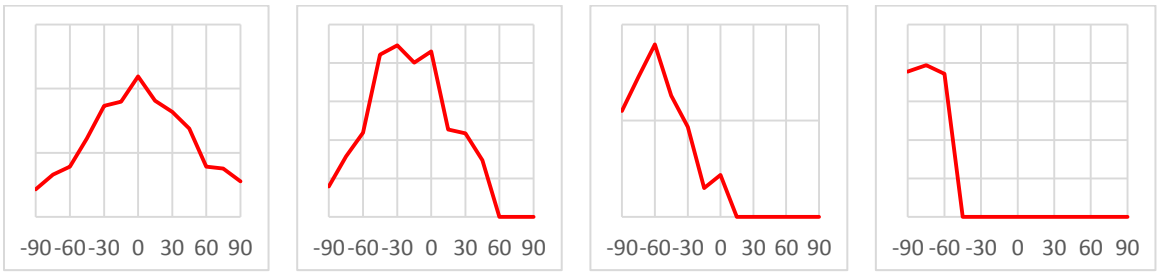


Diagram 92 - Components High-Score Results per Viewing Angle Example

In the Diagram 92 above some examples of components scores in various angles faces are shown.

The existence of 13 components increases the efficiency of the face detection algorithm but it actually aims on the pose estimation and not to a general face detection. All those properties of the algorithm gives the idea of splitting the algorithm in two similar but with different scope sections. The first section aims on the face detection and the second one on the pose estimation. If the first section detects a face within the image of a pyramid level then the second section is executed on the same image. It works as the Backtrack stage where the Backtrack procedure applies only when high-score values are detected by the find procedure.

The Face Detection section is using for the detection procedure one or a few more components instead of using all of them. These called the detection components. The Threshold parameter value on this section is a little lower in order to be more efficient in the face detection procedure and the detection components can detect faces belonging to other viewing angles far away from them. If in this procedure no faces are detected then the Pose Estimation section is not executed. On the other hand when a face is detected the rest components are used on the Pose Estimation section which is executed in order to achieve an accurate detection with pose estimation. The Threshold parameter value on the Pose Estimation section is as usual. The benefit of this patch is that it reduced the times the component stage execution is executed as it overtakes empty images faster than the original version.

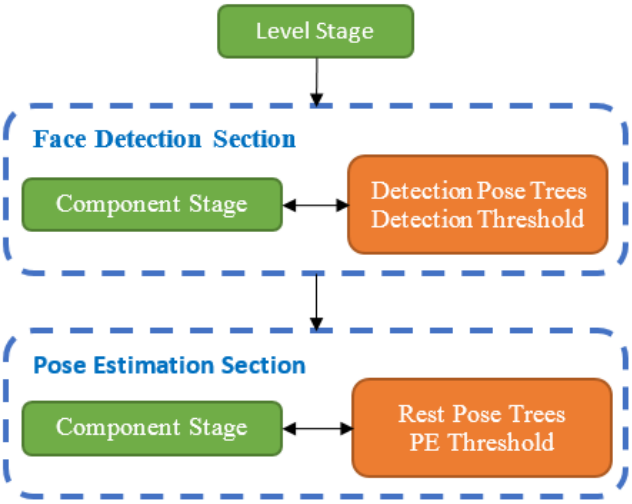


Figure 62 - Detection Components Patch Execution Flow Diagram

The ideal component for the face detection section seems to be the component 7 that represents the zero degree angle of a human face. It's the ideal as it can create high-score values in the detection procedure for both sides of the human face as it is symmetric. As the aim of the face detection procedure is the detection of faces in all thirteen poses, the threshold parameter value of this section must be lower than in the pose estimation one. As shown in the Diagram 91 and Diagram 92 the seventh component creates low scores on the faces belong to the edge components and this is a very important reason to reduce the threshold as this angles faces will not be detectable. On the other hand when low threshold is used on the face detection section, the more fake faces will be detected activating the pose estimation one. This would not create faulty detections as the pose estimation section uses the most efficient threshold value (Chapter 9.2) but it would cancel the advantage of the patch as it would treat to empty face levels as they contain faces.

Table 131 - DC Patch Face Detection Section Results (DC Set 7) (%)						
Threshold	**-0.75**	**-0.70**	**-0.65**	**-0.60**	**-0.55**	**-0.50**
Detected	86.5	84.2	80.6	78.2	75.0	72.2
Missed	13.5	15.8	19.4	21.8	25.0	27.8
Fake	35.0	25.2	17.5	10.1	6.90	3.98
Reliability	59.0	65.6	68.8	**71.9**	71.1	70.1

As seen in the Table 131 above, the parameter Threshold value of -0.60 is the most reliable. On the other hand its detection rating is not as high as in the lower values. At this point the reliability is not as significant as the pose estimation section would reject the fake faces from the results. The only way that this patch can affect the algorithms results is if faces are missed. Even if more fake faces are detected at the face detection section, they will be skipped during the pose estimation one where the usual Threshold parameter value will be used. So at this point of

the face detection section what is significant is faces not to be missed. If fake faces are detected the punishment would be useless calls of the pose estimation section than would cost execution time. So the -0.75 to -0.60 might be the most efficient Threshold parameter value for the face detection section. In the Table 132 below, the results using different values of this variable are shown.

Table 132 - DC Patch Results (DC Set 7) (%)									
DC Threshold	**-0.75**			**-0.70**			**-0.65**		
Threshold	**-0.50**	**-0.45**	**-0.40**	**-0.50**	**-0.45**	**-0.40**	**-0.50**	**-0.45**	**-0.40**
Detected	84.6	84.0	82.1	84.6	84.0	81.6	83.3	82.7	80.8
Missed	15.4	16.0	17.9	15.4	16.0	18.4	16.7	17.3	19.2
Fake	9.59	6.43	4.00	9.59	6.43	4.02	9.09	6.30	4.06
Reliability	77.6	79.4	79.3	77.6	**79.4**	78.9	76.9	78.3	78.1
FD Threshold	**-0.60**			**-0.55**			**-0.50**		
Threshold	**-0.50**	**-0.45**	**-0.40**	**-0.50**	**-0.45**	**-0.40**	**-0.50**	**-0.45**	**-0.40**
Detected	83.1	82.5	80.6	82.3	81.6	79.5	81.6	81.0	78.8
Missed	16.9	17.5	19.4	17.7	18.4	20.5	18.4	19.0	21.2
Fake	9.11	6.08	4.07	8.98	6.14	4.12	9.05	6.19	4.16
Reliability	76.7	78.3	77.9	76.1	77.5	76.9	75.5	76.9	76.2

As presented in the Table 132 above the patch results are very close to the ones without it (Chapter 9.2). All the indexes values have change at least. What is not shown in this table is what kind of faces have been missed. In the Table 138 the detection analysis of all components is shown, revealing that the missed face detections are coming from the edge components. As seen the -90 and 90 degrees face detections were decreased about 20% while the middle components seems not to be affected.

The patch's efficiency can be improved if more components are used as detection components in the face detection section. The ideal components would be some of the edge components as they create high scores in the face angles where the component seven does not. This is because they use some different filters. By testing the patch using the components seven, three and eleven (-60 ^{o}, 0 ^{o}, 60^{o}) the following results come of, as shown in Table 133.

Table 133 - DC Patch Face Detection Section Results (DC Set 7-3-11) (%)						
Threshold	**-0.65**	**-0.60**	**-0.55**	**-0.50**	**-0.45**	**-0.40**
Detected	89.1	87.8	85.0	83.1	82.9	81.0
Missed	10.9	12.2	15.0	16.9	17.1	19.0
Fake	21.5	12.2	7.87	4.89	3.24	1.81

Reliability	71.6	78.3	79.3	79.7	80.7	79.8

In the Table 133 the threshold used for testing is greater than when the patch used only the component seven. This is because the components three and eleven are used for the detection of faces in the angles of 60 to 90 (and -60 to -90) degrees instead of the component seven which is used for the detection of the rest centered viewing angles. As seen in the table the usage of a threshold of -0.45 is actually the most efficient, the same as the whole algorithms without it. At the usage of the same Detection threshold parameter value as seen in both Table 132 and Table 133, the Detection section has much better reliability when using three components instead of only one. The full algorithms detection results using these three components as detection ones are shown in the Table 134 below.

Table 134 - DC Patch Results (DC Set 7-3-11) (%)						
DC Threshold	-0.60			-0.55		
Threshold	-0.50	-0.45	-0.40	-0.50	-0.45	-0.40
Detected	85.5	84.8	82.9	85.5	84.8	82.7
Missed	14.5	15.2	17.1	14.5	15.2	17.3
Fake	9.30	6.15	3.96	9.50	6.15	3.97
Reliability	78.6	80.4	80.2	78.4	80.4	80.0
FD Threshold	-0.50			-0.45		
Threshold	-0.50	-0.45	-0.40	-0.50	-0.45	-0.40
Detected	85.5	84.8	82.3	84.2	84.2	81.8
Missed	14.5	15.2	17.7	15.8	15.8	18.2
Fake	9.30	6.15	3.99	6.19	6.19	4.01
Reliability	78.6	**80.4**	79.5	79.8	79.8	79.1

Looking at the results on the algorithms testing using this patch with those three components, it is visible that the algorithms reliability is not reduced at all. At the reliability line it appears that when the algorithm is used with this patch and detection threshold value larger than -0.50, the algorithm detection efficiency is not affected. What is actually important is that the algorithm's performance is not affected negatively and in contrast to the one component usage it is still effective in angles close to 90 degrees.

At last, one more benefit of this patch is that the algorithm can avoid the calculation of some of the Filter Responses that are not used in the face detection procedure. The algorithm can calculate only the Filter Responses used by the detection components. If the face detection section makes a detection, then the rest Filters Responses have to be calculated for use in the pose estimation section. If no detections occur, the algorithm can skip these Filters Responses

calculations. This is very important as the Filters Responses calculations are the main time consumer of the algorithm.

When using the 99 filters model the usage of the components seven, three and eleven as detection ones on the patch is enough to fill all the Filters Responses tables. This is a considerable reason of using only one detection component. This would not reduce the algorithm efficiency if the algorithm is used for applications interested in faces with not great viewing angles. In applications like these this patch can be combined with the 68 filters model (Chapter 9.5).

Another idea is to replace the components three and eleven with the four and ten ones. This means that the patch would use the same 68 filters for all the three detection components when used on the 99 filters model. Testing this version of the DC patch gets the following results of the Table 135 below.

Table 135 - DC Patch Face Detection Section Results (DC Set 7-4-10) (%)						
Threshold	**-0.70**	**-0.65**	**-0.60**	**-0.55**	**-0.50**	**-0.45**
Detected	92.5	90.8	89.7	87.6	85.0	84.4
Missed	7.48	9.19	10.3	12.4	15.0	15.6
Fake	48.2	36.7	23.8	16.5	9.55	6.40
Reliability	49.7	59.5	70.1	74.7	78.0	79.8

As did with the other two cases the algorithm was tested in different combinations between the Face Detection section and Pose estimation one threshold. The results are shown in the Table 136 below.

Table 136 - DC Patch Results (DC Set 7-4-10) (%)									
DC Threshold	**-0.75**			**-0.70**			**-0.65**		
Threshold	**-0.50**	**-0.45**	**-0.40**	**-0.50**	**-0.45**	**-0.40**	**-0.50**	**-0.45**	**-0.40**
Detected	85.0	84.4	82.5	85.0	84.4	82.5	85.0	84.4	82.5
Missed	15.0	15.6	17.5	15.0	15.6	17.5	15.0	15.6	17.5
Fake	9.55	6.40	3.98	9.55	6.40	3.98	9.55	6.40	3.98
Reliability	78.0	79.8	79.8	78.0	79.8	79.8	78.0	**79.8**	79.8
FD Threshold	**-0.60**			**-0.55**			**-0.50**		
Threshold	**-0.50**	**-0.45**	**-0.40**	**-0.50**	**-0.45**	**-0.40**	**-0.50**	**-0.45**	**-0.40**
Detected	84.6	84.0	82.1	83.8	82.9	81.2	83.8	82.9	81.0
Missed	15.4	16.0	17.9	16.2	17.1	18.8	16.2	17.1	19.0
Fake	9.38	6.21	4.00	9.26	6.28	4.04	9.26	6.28	4.05
Reliability	77.8	79.6	79.3	77.2	78.5	78.5	77.2	78.5	78.3

Finally a comparison table (Table 137) is used in order to see and compare the differences of using one and three detection components and which of them. As seen in this table the usage of the three 68 filters detection components (7, 4, 10) does not provide any crucial benefit in contrast of using only one. It only offers a small increment in the detection efficiency and the reliability but they can be matched if a lower FD Threshold parameter value is used to the one detection component method. Actually when using about -0.10 lower FD Threshold parameter value the one component detection method succeeds almost the same results with the 3 components one (7, 4, 10). The 99 filters detection components (7, 3, 11) usage on the other hand offers much better result, close to the ones the algorithm succeeds without the patch.

Table 137 - DC Patch Results Comparison (Threshold = -0.45) (%)									
D. Components	**C-7**						**ALL**		
FD Threshold	**-0.70**	**-0.65**	**-0.60**	**-0.55**	**-0.50**	**-0.45**	**-0.50**	**-0.45**	**-0.40**
Detected	84.0	82.7	82.5	81.6	81.0	79.5	85.5	84.8	83.1
Missed	16.0	17.3	17.5	18.4	19.0	20.5	14.5	15.2	16.9
Fake	6.43	6.30	6.08	6.14	6.19	6.06	9.50	6.37	3.95
Reliability	**79.4**	78.3	78.3	77.5	76.9	75.6	78.4	80.2	80.4
D. Components	**C-7-4-10**						**ALL**		
FD Threshold	**-0.70**	**-0.65**	**-0.60**	**-0.55**	**-0.50**	**-0.45**	**-0.50**	**-0.45**	**-0.40**
Detected	84.4	84.4	84.0	82.9	82.9	82.1	85.5	84.8	83.1
Missed	15.6	15.6	16.0	17.1	17.1	17.9	14.5	15.2	16.9
Fake	6.40	6.40	6.21	6.28	6.28	6.11	9.50	6.37	3.95
Reliability	79.8	**79.8**	79.6	78.5	78.5	77.9	78.4	80.2	80.4
D. Components	**C-7-3-11**						**ALL**		
FD Threshold	**-0.70**	**-0.65**	**-0.60**	**-0.55**	**-0.50**	**-0.45**	**-0.50**	**-0.45**	**-0.40**
Detected	84.8	84.8	84.8	84.8	84.8	84.2	85.5	84.8	83.1
Missed	15.2	15.2	15.2	15.2	15.2	15.8	14.5	15.2	16.9
Fake	6.37	6.37	6.15	6.15	6.15	6.19	9.50	6.37	3.95
Reliability	80.2	80.2	80.4	80.4	**80.4**	79.8	78.4	80.2	80.4

One important difference between the 68 filters detection components (7, 7-4-10) and the 99 one is that the first's reliability is linear. The 68 filters components reliability is increasing as the face detection section threshold parameter value is decreasing. This is sensible as this way the face detection section makes more detections and calling for the final detection the pose estimation section. This creates better results but destroys the reason of using the detection section. If the detection section detects faces every time it is executed, then there is no execution time profit as the pose estimation section is also always executed. On the other hand the 99 filters detection components reliability is not linear making the detection section more efficient. As seen in the Table 137 data these detection components combination has its best

reliability at the detection section threshold parameter value at -0.50 succeeding reliability better than the algorithm without this patch. This is a very important fact as these detection components can make the detection section very profitable, skipping the pose estimation one many times as explained before.

One important comparison is between the two different sets of detection components that use the 68 filters. When using only the component seven the algorithms reliability is decreased about 1%, a difference that can be omitted if the Threshold parameter value is decreased as referred in the previous paragraph. The main difference between these two detection components sets is that when one detection component is used the algorithm is less efficient on great viewing angles as shown in the Table 138 below. As seen in this table when one detection component is used the algorithm is having great loss in the pose angles less than -60^{O} and more than 60^{O}, almost the twice more than the three detection components set. Again the reduction of the FD Threshold parameter value can fix this problem. As far as the pose estimation, the algorithm accuracy is affected less than 1% on the 68 filters detection components models and almost 0% to the 99 one.

Table 138 - DC Patch Missed Detections Viewing Angle Classification (%)									
D. Components	C-7-3-11			C-7-4-10			C-7		
FD Threshold	-60<	-60<&<60	<60	-60<	-60<&<60	<60	-60<	-60<&<60	<60
-0.70	0	0	0	-5.26	0.32	0	-10.5	0.95	-4.17
-0.65	0	0	0	-5.26	0.32	0	-17.5	0.32	-4.17
-0.60	-1.75	0.32	0	-8.77	0.63	-4.17	-17.5	0.32	-8.33
-0.55	-1.75	0.32	0	-14.0	0.95	-16.7	-19.3	0.32	-20.8
-0.50	-1.75	0.32	0	-14.0	0.95	-16.7	-22.8	0.32	-25.0

Comparing the DC patch results with the ones that the algorithm succeeds without it, it is visible and at the same time sensible that the face detections efficiency is decreased a bit but in a very tiny amount. Using this patch, the algorithm can sacrifice a very small amount of its face detection efficiency and maybe a little bit of its reliability gaining execution time. Especially when the 68 filters detection components are used the algorithm except of skipping the components stage of many model components it can also skip the calculation of 31 filters responses saving a lot of execution time.

The amount of execution time that the algorithm can save using these detection component is not able to be defined as it is detections dependent. The only thing can be done is to predict it using some of the algorithms characteristics according to previous tests mad

In chapter 6.2, a profiling of the Find procedure was appose using the default threshold variable value of -0.65. After the NMS procedure changes (Chapter 9.1) application on the algorithm, the most suitable threshold variable value is set to -0.45. When using the 99 filters detection

components the most efficient value of the detection threshold parameter is the -0.50 while for the 68 filters ones the minimum is the best. For that reason a profiling for multiple different values of the Threshold variable was done. By this testing, the data we need is the number of levels with high-score values a face produce to its corresponding component. This is a different pointer relatedly to the Levels with high-score values used in chapter 7. In this patch the maximum number of levels with high-score values that a face produce at every component is needed. This component is probably the one that will be detected as the correct pose of it. This way, according to the threshold parameter value, the number of levels in which the patch will execute the pose estimation section can be estimated. The profiling results are shown in Table 139 below.

Table 139 - DC Patch Max($Levels_{High\text{-}Scores}$[Component]) %			
Threshold	**Max**	**Average**	**Min**
-0.75	92.3	44.1	11.8
-0.70	92.3	42.6	11.8
-0.65	84.6	39.9	6.25
-0.60	71.4	37.1	6.25
-0.55	69.2	36.1	6.25
-0.50	69.2	34.1	6.25
-0.45	61.5	32.7	6.25

Another parameter for predicting the execution time profit of this patch is the time needed for the Component and Convolution stages. As referred in chapter 8.5 the relation between two sequential levels execution time is given by the function (31) below. As referred in the previous paragraph, the number of levels with high-score values produced by a detection is equal to the expression (32) where the $AVG_{\max-LHS}(Threshold)$ is the average column of Table 140. In the function (32) the rounding is down in order to get safer results.

$$Time(L) \approx 0.25 \cdot 0.76^{L} \cdot Time(All) \qquad (31)$$

$$MaxL_{High-Scores}(Threshold) = \left\| AVG_{Max-LHS}(Threshold) \times length(F.Pyramid) \right\| \qquad (32)$$

Using the function (31) above the execution time graph of every level execution time is as shown in Diagram 93. In the Table 140 also is shown the value of $MaxL_{High-Scores}(Threshold)$ on different image sizes and different Threshold parameter values.

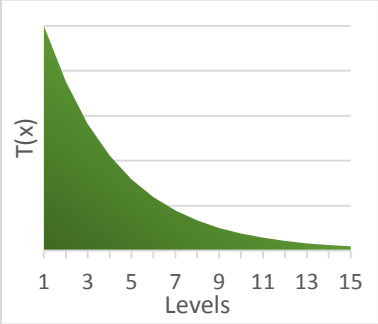


Diagram 93 - Function (31) Diagram

Table 140 - DC Patch MaxL$_{High\text{-}Scores}$(Threshold)							
Threshold	**-0.75**	**-0.70**	**-0.65**	**-0.60**	**-0.55**	**-0.50**	**-0.45**
320x240	3	3	3	3	3	2	2
640x480	5	5	5	4	4	4	4
800x600	6	6	5	5	5	4	4
1024x768	7	6	6	6	5	5	5
1280x960	7	7	7	6	6	6	6

Combining the function (31) and (32) a prediction functions that tries to calculate the execution time saved in different situations is created, the function (33) below, that calculates the execution time of the detect stage when using this patch.

$$T_{\text{Level}}^{'} = T_{\text{Level}} - \frac{13 - C_{\det ect}}{13} \times \sum_{x=L[\text{no\_face}]} T_{Level}(x) \qquad (33)$$

$$T_{Conv}^{'} = T_{Conv} - \frac{31}{99} \times \sum_{x=L[\text{no\_face}]} T_{Conv}(x) \qquad (34)$$

Another advantage of this patch appears when using the 68 filters detection components on the detection section. If only these filters are used then there is no reason for the Convolution stage to calculate the rest filters responses (31 filters). If a detection is discovered then it will be executed again to complete the calculations of the rest 31 filters responses. This gives an extra saving of execution time that might worth the usage of 68 filters detection components. For this case the function predicting the execution time needed for the Convolutions stage is the function (34).

It is very complicate to make predictions of any case scenario of detections. It is obvious that when an image is full of faces with different scales then detections would occur through the whole features pyramid levels. On the other hand even if more than one faces exist within the image but these faces are the same scale (for example, a family photo) then all the detections would probably appear in the same levels of the features pyramid like only one existing. In order to present the advantages of this patch, this scenario is going to be used as it is a possible to real life images.

Figure 63 - Multiple Faces, Same Scale Image Example

Figure 64 - Multiple Faces, Multiple Scales Image Example

In the Table 141 below the execution time profit of no detections in the detection procedure according to the features pyramid level is shown at the last line. By this table, adding the profits, it is easy to predict some way what would final profit be for different detections scenarios. In our main scenario, described in the next paragraph, the total profit would be equal to the sum of the levels with no detections. As is visible in the last line the 68 filters components detection model can reduce the algorithm's execution time to its half when no detection occur while the 99 filters one only for a quarter.

Table 141 - DC Patch Execution Time Profit per Level (%)					
Levels	**Level Stage**			**+ Convolution Stage**	
	C-7-3-11	**C-7-4-10**	**C-7**	**C-7-4-10**	**C-7**
1	-5.96	-5.35	-6.79	-10.6	-12.1
2	-4.53	-4.07	-5.16	-8.07	-9.17
3	-3.44	-3.09	-3.92	-6.14	-6.97
4	-2.62	-2.35	-2.98	-4.66	-5.29
5	-1.99	-1.79	-2.26	-3.54	-4.02
6	-1.51	-1.36	-1.72	-2.69	-3.06
7	-1.15	-1.03	-1.31	-2.05	-2.32
8	-0.87	-0.78	-0.99	-1.56	-1.77
9	-0.66	-0.60	-0.76	-1.18	-1.34
10	-0.50	-0.45	-0.57	-0.90	-1.02
11	-0.38	-0.34	-0.44	-0.68	-0.78
12	-0.29	-0.26	-0.33	-0.52	-0.59
13	-0.22	-0.20	-0.25	-0.39	-0.45
14	-0.17	-0.15	-0.19	-0.30	-0.34
15	-0.13	-0.11	-0.15	-0.23	-0.26
16	-0.10	-0.09	-0.11	-0.17	-0.20

17	-0.07	-0.07	-0.08	-0.13	-0.15
All	**-24.6**	**-22.1**	**-28.0**	**-43.8**	**-49.8**

The Table 141 above presents the execution time profit when no detection occur within an image. This is very important if the TSM algorithm is used in video application where the empty frames detection procedure can be completed much faster.

On the other hand, in real life applications it is more probable one or more detections, real or fake, to appear in the detection procedure. In the next tables the profiling of the DC patch is presented according to the profiling scenario where it is supposed that one or more, same scale, faces are detected, including a number of fake faces (Detection Noise).

Table 142 - DC Patch Execution Time Reduction per Face Size (DC Set 7) (%)

Faces Size	Threshold	320x240	640x480	800x600	1024x768	1280x960	Average
50%	**-0.70**	**-21.2**	**-30.6**	**-33.5**	**-34.8**	**-35.3**	**-31.1**
	-0.65	-22.7	-33.1	-36.0	-37.4	-38.1	-33.4
	-0.60	-24.5	-35.7	-38.5	-40.0	-40.9	-35.9
	-0.55	-25.6	-36.8	-39.5	-41.1	-42.1	-37.0
40%	**-0.70**	-	-24.0	-28.9	-31.4	-32.8	-29.3
	-0.65	-	-27.0	-31.7	-34.3	-35.7	-32.2
	-0.60	-	-30.1	-34.6	-37.2	-38.8	-35.2
	-0.55	-	-31.3	-35.7	-38.4	-40.0	-36.4
30%	**-0.70**	-	-9.93	-19.0	-24.2	-27.3	-20.1
	-0.65	-	-13.9	-22.5	-27.6	-30.7	-23.7
	-0.60	-	-18.1	-26.1	-31.0	-34.2	-27.4
	-0.55	-	-19.6	-27.5	-32.4	-35.6	-28.8
20%	**-0.70**	-	-	-12.3	-4.77	-11.8	-9.60
	-0.65	-	-	-14.8	-8.55	-16.4	-13.2
	-0.60	-	-	-17.5	-13.6	-21.2	-17.4
	-0.55	-	-	-18.5	-15.5	-23.0	-19.0

Table 143 - DC Patch Execution Time Reduction per Face Size (DC Set 7-4-10) (%)

Faces Size	Threshold	320x240	640x480	800x600	1024x768	1280x960	Average
50%	**-0.65**	**-18.7**	**-26.9**	**-29.2**	**-30.1**	**-30.3**	**-27.0**
	-0.60	-20.4	-29.3	-31.6	-32.6	-33.0	-29.3
	-0.55	-21.5	-30.7	-33.0	-34.1	-34.7	-30.8
	-0.50	-23.4	-32.5	-34.8	-36.0	-36.8	-32.7

40%	**-0.65**	**-**	**-21.5**	**-25.4**	**-27.3**	**-28.2**	**-25.6**
	-0.60	-	-24.4	-28.1	-30.0	-31.1	-28.4
	-0.55	-	-25.9	-29.6	-31.7	-32.8	-30.0
	-0.50	-	-28.0	-31.6	-33.7	-35.1	-32.1
30%	**-0.65**	**-**	**-10.0**	**-17.3**	**-21.4**	**-23.8**	**-18.1**
	-0.60	-	-13.8	-20.6	-24.6	-27.0	-21.5
	-0.55	-	-15.6	-22.3	-26.4	-28.9	-23.3
	-0.50	-	-18.4	-24.8	-28.8	-31.4	-25.9
20%	**-0.65**	**-**	**-**	**-10.6**	**-4.69**	**-11.2**	**-8.82**
	-0.60	-	-	-13.0	-9.31	-15.6	-12.6
	-0.55	-	-	-14.5	-11.5	-17.8	-14.6
	-0.50	-	-	-16.4	-15.0	-21.0	-17.5

Table 144 - DC Patch Execution Time Reduction per Face Size (DC Set 7-3-11) (%)

Faces Size	Threshold	320x240	640x480	800x600	1024x768	1280x960	Average
50%	**-0.50**	**-13.4**	**-18.7**	**-20.0**	**-20.8**	**-21.3**	**-18.9**
	-0.45	-14.0	-19.2	-20.4	-21.2	-21.7	-19.3
40%	**-0.50**	**-**	**-16.2**	**-18.3**	**-19.6**	**-20.4**	**-18.6**
	-0.45	-	-16.7	-18.7	-20.0	-20.8	-19.1
30%	**-0.50**	**-**	**-10.8**	**-14.5**	**-16.8**	**-18.3**	**-15.1**
	-0.45	-	-11.6	-15.1	-17.4	-18.8	-15.7
20%	**-0.50**	**-**	**-**	**-9.75**	**-9.01**	**-12.5**	**-10.4**
	-0.45	-	-	-10.2	-9.93	-13.3	-11.1

As is visible in the Table 142, Table 143 and Table 144, the execution time profit of this patch is larger as the faces size is increasing. This is sensible because as the larger a detected face is the larger (to the bottom) the level detected is. Then the levels of the pyramid with detections are the smallest one and the largest are empty making the DC patch much more useful. The same phenomenon applies to the image size. The larger an image is the more profitable the DC patch is. This is because the larger an image is the larger is the pyramid created, making the corresponding to the face scale levels to move toward to the bottom ones. As small irregularity on this is cause on very small face size where the upper part of the detection range to the levels of the pyramid is expanding beyond the pyramid as explained in chapter 6.2. In some images very small faces cannot also be detected as described in chapter 9.4.

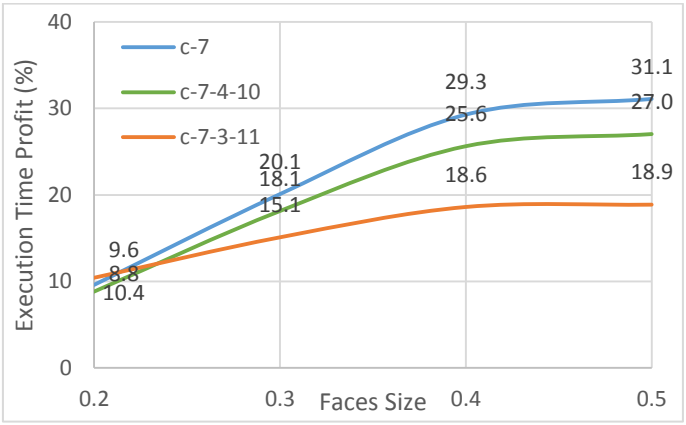


Diagram 94 - Detection Components Sets Execution Time Profit per Face Size

To summarize, a 10% to 30% of execution time can be saved when using the DC patch according to the detection component set is used. This means that this patch can be a bit profitable to the TSM algorithm execution time. This patch is a root patch for the next patches that are presented in the next chapters and give the algorithm the ability of extra time saving.

9.7. Fast Pose Estimation

In this chapter a new approach on the execution flow of the pose estimation section of the DC patch is presented. This approach aims on the same accuracy with less execution time needed. The pose estimation section consists by the component stage where the DT and Backtrack ones are contained. Inside this stage the algorithm applies the detection procedure for every pose tree in order to decide which one is the correct. The highest score is the parameter that defines the correctness. This patch is actually the next step of DC one and it can be considered as its extension or a later version.

In this patch a new way of estimating the right pose is introduced. The algorithm does not apply the detection procedure to all pose trees but makes decisions about the pose trees to be used in the component stage using information from the already used pose trees results. In the Figure 65 on the right, the execution flow of this patch is shown. At the beginning of every Level stage the algorithm executes the Component stage for the detection components used by the DC patch. This patch adds a new data structure, the "Faces". This data structure holds information about every face detected in the component stage. This data structure is explained in chapter 9.7.1.

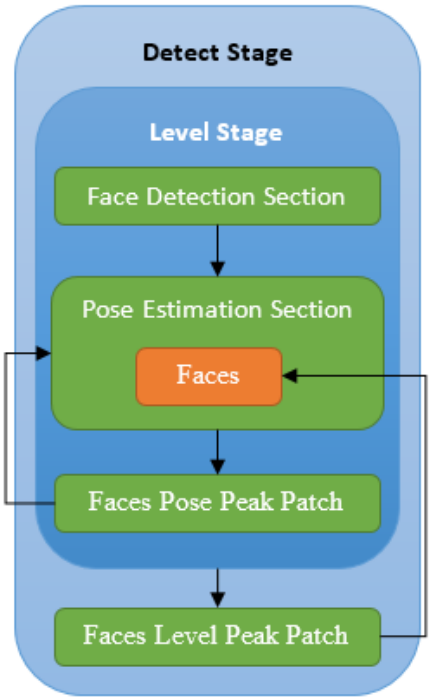


Figure 65 - Fast Pose Estimation Patch Execution Flow Diagram

After the detection components has completed the face detection section, the "Pose Peak Detection" patch use the information stored in the "Faces" data structure list to decide which pose tree the pose estimation section should execute. Decisions are made after every component stage execution as the Faces data structures information are updated. The way this patch makes decisions is explained in chapter 9.7.2.

When the Level stage is completed, then the algorithms uses again the information of the Faces data structures through the "Face Peak Detection" patch to decide which faces have completed their pose estimation procedure. The FPD patch uses the detection results from multiple levels in order to take these decisions as is described in chapter 9.7.2.

The basic idea behind the FPE Patch is the application of the PPD one. The FPD patch is not mandatory but as it is explained in the chapter 9.7.3 is does not affect at all the algorithms detection performance and for this it is greatly recommended. The execution time profit these patched offer is presented in the corresponding chapters.

9.7.1. Face Data Structure

The Face structure (Table 145) is a data structure that can easily replace the Results Cache one. When a component stage is completed every high-score value of the DT score results is consider a detected face. Using the Find v2.0 procedure these results end to less than two high-score values per face detection. The algorithm then adds a Face data structure in the Faces array storing the high-score value of this detection. If this high-score value is the highest this Face has succeeded then it stores its detection result to the box variable. Every time a high-score value is

discovered, the algorithm check if it overlaps an already added Face as exactly the NMS procedure does. If an overlap occurs then the high-score value is rejected. Even if it is rejected, the algorithm keeps the highest high-score values of every component at every level to the Scores array of this Face like keeping a high-score values log file. These information are used by the FPD (Chapter 9.7.2) and LPD (Chapter 9.7.3) patches. This processing may delay a little bit the component stage to be completed but it rejects the need of the NMS procedure at the end of the detection stage. It is like the NMS procedure to be applied for every new detection.

Table 145 - Face Data Structure	
Box	Detection results. The Backtrack stage output.
Scores[13, length(FP)]	Array holding the highest high-score value for every component stage executed.
Completed	Flag used when the detection procedure of this face is completed

The Face data structure array can replace the Results Cache data structure as it contains the significant information the last one holds. The Results Cache according to the algorithm profiling presented in chapter 6.14, contains the amount of data as shown in the Table 146 below for a detection. The Scores array size is smaller than the Result data structure when the pyramid length is less than 21 levels. It is almost impossible to exceed twice the size of the Result data structure. As is visible in this table the Results Cache structure holds much more data per detection than the Face one that makes the last one more efficient.

Table 146 - Face vs Results Cache Data Structures Size per Detection	
Data Structure	**Size**
Face	$\mathrm{Re}sult + lenght(Components) \times length(F.P) \approx 2 \cdot \mathrm{Re}sult$
Results Cache	$\mathrm{Re}sult \cdot \left(length(Components) \times length(F.P) \times Levels_{High-Scores} \times AVG\left(Pixels_{High-Scores}\right)\right)$ $\approx 3.4 \cdot length(F.P) \cdot \mathrm{Re}sult$

As far as the time consumption needed for the Face data structure to check for overlaps, it is totally insignificant compared to the whole algorithm execution time and it replaces the time needed for the NMS procedure that is not needed any more. After the Find procedure 2.0 version the execution time and the memory needed for the NMS procedure and the detections results were reduced so much that any further reduction seems totally insignificant compared to the whole algorithm consumptions.

9.7.2. Pose Peak Detection

The Pose Peak Detection patch is the one that compares a Face highest-score values across the component stages executed and decides if its pose estimation procedure is completed for the corresponding level or which pose tree should be used for the next component stage needed. As

shown in graphs X in chapter X, at the DC patch, the highest-score value of every pose tree create the highest-scores curve for every face as the ones shown in the Diagram 95 below.

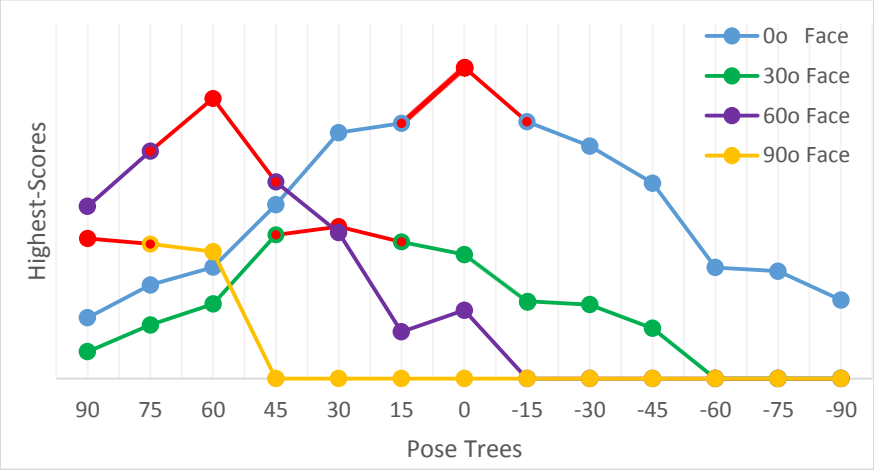


Diagram 95 - Level Highest-Scores Curves Peaks Example

Every curve draws a peak at the position of a pose tree which is consider to be the correct pose estimation of the detection. This peak is the one that the PPD patch is trying to detect. Using the maximum high-score (highest-score) of the pose trees components stage results the PPD patch searches the gradient that leads to that peak. The execution flow this patch follows is as the Figure 66 shows.

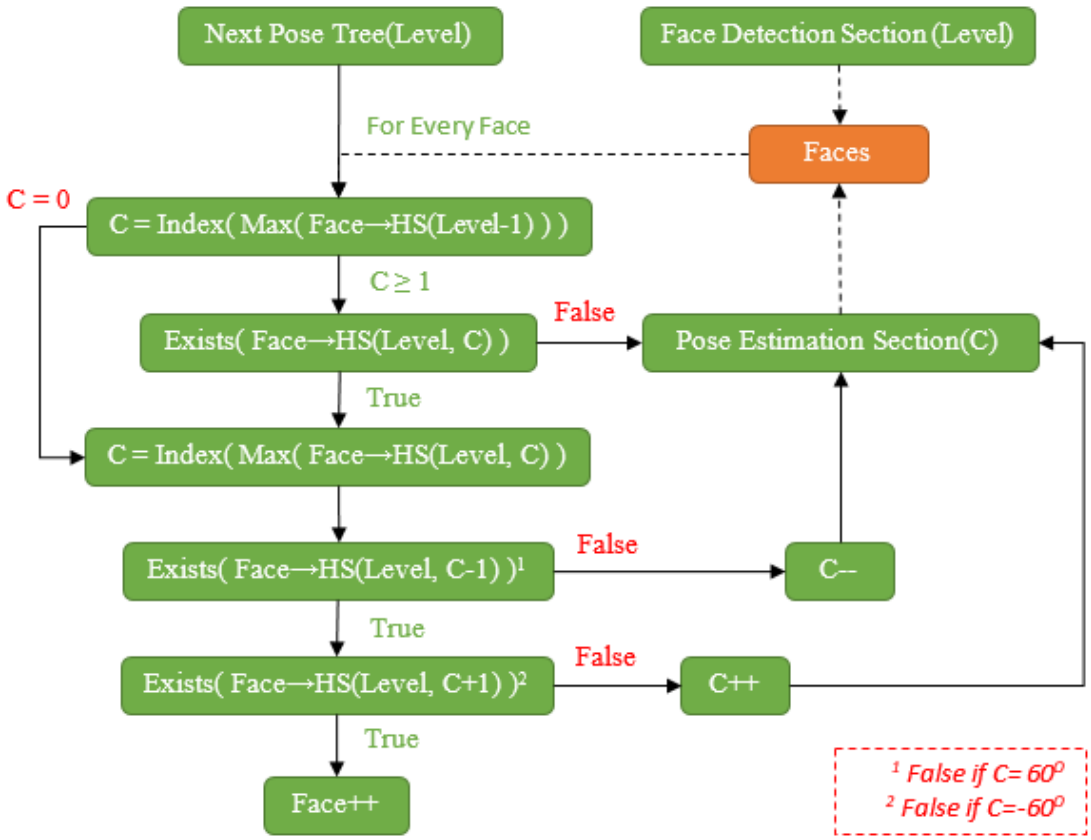


Figure 66 - Pose Peak Detection Patch Execution Flow Diagram

The route that the PPD patch will follow until it discovers the pose peak is defined by the detection components (DC) used in the face detection section of the DC patch. If the 99 filters DC are used then the poses trees tree that is going to be followed is as shown in Figure 67. Otherwise for the 68 filters DC sets the poses trees tree is the ones on the Figure 68 and Figure 69. When more than two faces are detected within the image the second one uses components stage results of the pose trees executed for the first one and if it is not enough, it can continue from the closest node to the leaf it belongs.

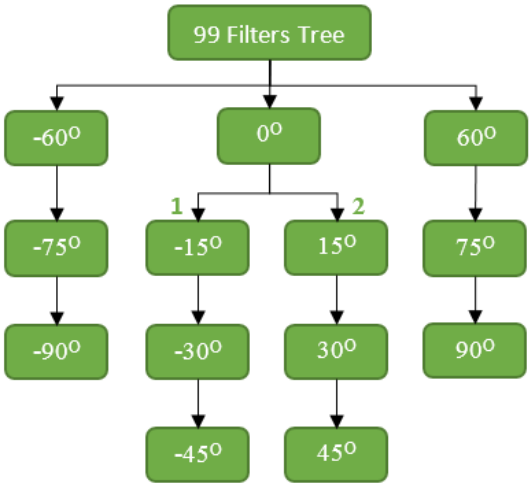


Figure 67 - Detection Components PPD Tree for 99 Filters 3 DC

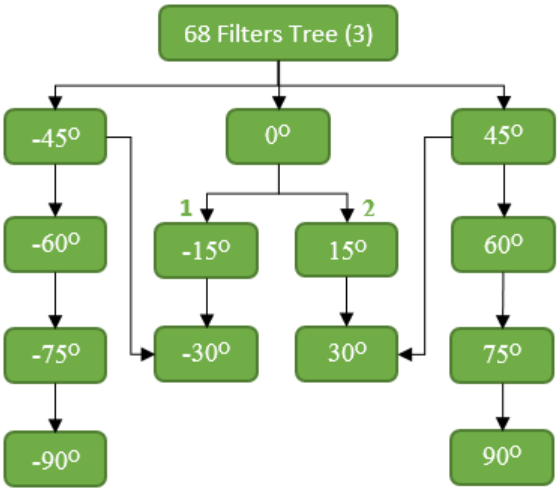


Figure 68 - Detection Components PPD Tree for 68 Filters 3 DC

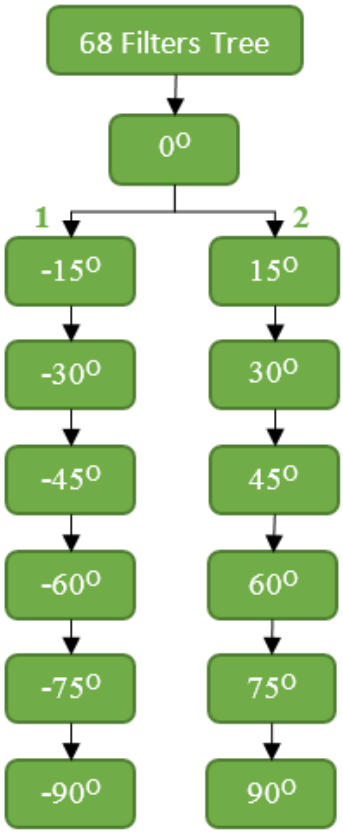


Figure 69 - Detection Components PPD Tree for 68 Filters 1 DC

According to these three trees the Table 147 below shows the number of pose trees that have to be used in the component stage in order the patch can estimate a face pose correctly. As is visible in the "AVG" column there is no significant difference between these three DC sets of the face detection section as far as the average components need to be executed until the patch makes a decision. The main difference between these three sets is their variance. As seen the three DC set variance is less than 1, meaning that the number of executions of the component stage will be always close their average value. As seen in the Table 147, the minimum number of the component stage executions is 4 while the maximum is 7 for the 99 filters DC set. On the other hand the 68 filters one DC set succeeds a minimum number of component stage executions to 3 while the maximum is 8. That is why its variance is about 3.

Table 147 - PPD Patch Components Stage Execution Times per Pose													
Poses	**-90**	**-75**	**-60**	**-45**	**-30**	**-15**	**0**	**15**	**30**	**45**	**60**	**75**	**90**
DC-7	7	7	6	5	4	3	3	4	5	6	7	8	8
DC-7-4-10	6	6	5	5	5	5	5	6	5	5	5	6	6
DC-7-3-11	5	5	4	6	6	5	5	6	7	7	4	5	5

	VAR(All)	**AVG(All)**	**AVG(-45°≤ & ≤+45°)**	**AVG(-60°≥ & ≥+60°)**
DC-7	2.85	5.62	4.29	7.17
DC-7-4-10	0.24	5.38	5.14	5.67
DC-7-3-11	0.85	5.38	6.00	4.67

As seen in the Table 147 above, the 99 filters DC set is faster when the detected faces belongs to the 39 filters components (±(60°-90°)) while the 68 filters ones are faster on the 68 filters components. What is also important for choosing one of these sets is their efficiency at face detection section and their time consumption profit as referred in the DC patch (chapter 9.6). Probably this would be the main criteria for using each one.

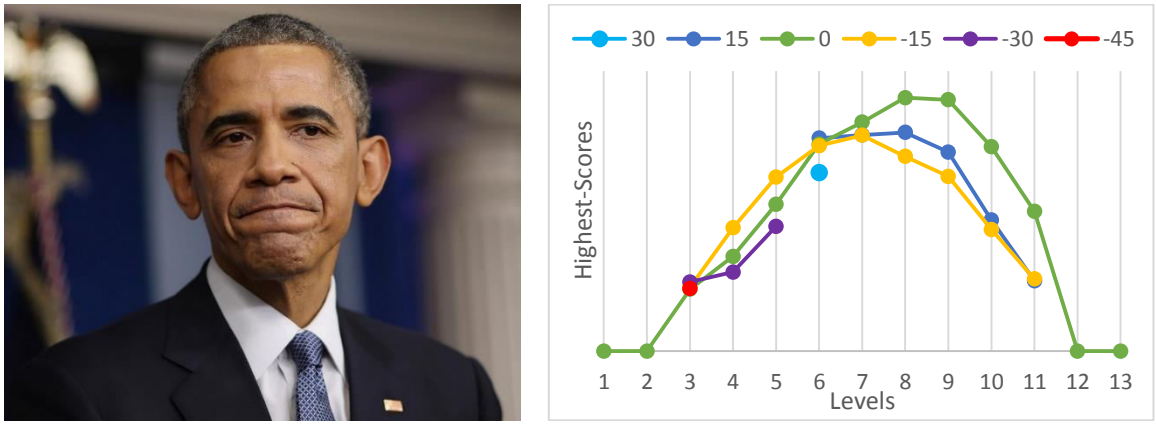


Diagram 96 - Face Pose Peak Patch Example

In the Diagram 96 above the Component stage executions are shown for the image to its left using the 68 filters one DC set at the face detection section of DC patch. As is visible the algorithm executed the component stage only for the pose models 4 to 9 (-45° to 30°). At the top levels of the features pyramid, where the face is not clear yet, the results of the detection procedure lead to the -15° pose tree, but as the feature pyramid level is reaching the appropriate scale the pose estimation approaches the correct pose tree (pose 0°). As seen, the number of pose trees used at every level is maximum at four.

By testing the PPD patch the following results came from (Table 148). As far as the 99 filters DC set, the results are very positive as the algorithms performance seems not to have been affected at all compared to the results of the Table 134 (Chapter 9.6) and are almost similar to the results

of the algorithm without using any patch. Looking at the results when the 68 filters three DC sets what is observed is a small drop of the algorithm performance, about 1-2% on its reliability and detection efficiency indexes compared to the algorithm version without any patch and about 1% compared to the DC patch. On the other hand, the 68 filters one DC set performance is much lower than the other two sets. It is obvious that this set's low detection efficiency in the great viewing angles drops its total performance, although it can be useful for centered faces detection applications.

Table 148 - PPD Patch Results Comparison (Threshold = -0.45) (%)									
DC Set	**DC-7**						**ALL**		
FD Threshold	**-0.75**	**-0.70**	**-0.65**	**-0.60**	**-0.55**	**-0.50**	**-0.50**	**-0.45**	**-0.40**
Detected	78.8	77.8	76.9	75.6	75.0	73.3	85.5	84.8	83.1
Missed	21.2	22.2	23.1	24.4	25.0	26.7	14.5	15.2	16.9
Fake	5.38	5.70	5.26	5.09	5.39	4.72	9.50	6.37	3.95
Reliability	**75.5**	74.3	73.8	72.7	71.9	70.7	78.4	80.2	80.4
DC Set	**DC-7-4-10**						**ALL**		
FD Threshold	**-0.75**	**-0.70**	**-0.65**	**-0.60**	**-0.55**	**-0.50**	**-0.50**	**-0.45**	**-0.40**
Detected	82.9	82.7	81.8	81.6	80.6	79.7	85.5	84.8	83.1
Missed	17.1	17.3	18.2	18.4	19.4	20.3	14.5	15.2	16.9
Fake	6.05	6.07	6.36	5.68	5.99	6.05	9.50	6.37	3.95
Reliability	**78.7**	78.5	77.5	77.8	76.6	75.8	78.4	80.2	80.4
DC Set	**DC-7-3-11**						**ALL**		
FD Threshold	**-0.70**	**-0.65**	**-0.60**	**-0.55**	**-0.50**	**-0.45**	**-0.50**	**-0.45**	**-0.40**
Detected	-	84.6	84.6	84.2	84.0	81.6	85.5	84.8	83.1
Missed	-	15.4	15.4	15.8	16.0	18.4	14.5	15.2	16.9
Fake	-	5.71	5.71	5.74	5.30	2.30	9.50	6.37	3.95
Reliability	-	80.5	**80.5**	80.1	80.2	80.1	78.4	80.2	80.4

What is significant is the fact that the 68 filters one DC set needs a very low face detection threshold parameter value to succeed functional performance in contrast to the DC patch. This is because in the DC patch a detection of one face was enough to activate all components to be used in the pose estimation section. On the PPD patch this is not happening. On this patch only the components needed for the pose estimation of this detected face are used. This reveals the weakness of the 68 filters one DC set to respond to the detection efficiency the TSM algorithm has to offer. The 68 filters one DC would probably be useful if used with the 68 filters Model presented in chapter 9.5.

At last, compared to the Table 134 (Chapter 9.6) is deduced that the effect of PPE patch is tiny to the algorithm detection efficiency and reliability as far as the face detection procedure. In addition comparing the pose estimation results of the algorithm with and without this patch the difference is about 1% as presented in Table 149. These results makes it obvious that this patch is safe enough to be used with the TSM algorithm as it can offer a reduction on its execution time without sacrificing any significant amount of its performance.

Table 149 - FPE Patch Pose Estimation (%)		
Threshold	**FPE Patch**	**No Patch**
-0.65	81.4	82.3
-0.60	81.7	82.6
-0.55	82.0	83.4
-0.50	82.9	83.9
-0.45	83.1	84.0
-0.40	83.2	84.2

As far as the execution time consumption profit using this patch, it is fully detection dependent. There is a huge variety of occasions that may occur so only the basic scenarios introduced in chapter 9.6 are going to be presented. In this patch another parameter affecting its performance is the viewing angle of the detected faces. If there are faces looking to all directions within the image, then it will be no execution time profit as all the pose trees will be necessary for the right pose estimations. If there is one face only within a level image or multiple looking at similar direction, then this patch will be proved useful. All these scenarios are presented in the Table 150 below. What is interesting in this patch is the fact that it also affects the fake detections detection procedure the same way it affects the real ones. This means that the execution time saving does not totally comes from the real faces detection procedure but also by the fake ones.

Table 150 - PPD Patch Execution Time Reduction per DC Set (1 Face Scenario) (%)							
Faces Size	**Threshold**	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
	C-7-4-10						
50%	**-0.70**	-33.9	-36.5	-37.4	-37.7	-37.7	-36.7
	-0.60	-34.8	-37.9	-38.7	-39.1	-39.2	-38.0
40%	**-0.70**	-	-34.5	-36.0	-36.6	-37.0	-36.0
	-0.60	-	-36.2	-37.5	-38.2	-38.6	-37.6
30%	**-0.70**	-	-30.3	-33.0	-34.4	-35.3	-33.2
	-0.60	-	-32.6	-34.9	-36.3	-37.2	-35.3
20%	**-0.70**	-	-	-30.9	-28.5	-30.6	-30.0
	-0.60	-	-	-32.3	-31.0	-33.2	-32.2
	C-7-3-11						
50%	**-0.60**	-20.6	-22.1	-22.5	-22.7	-22.8	-22.1
	-0.50	-21.0	-22.4	-22.8	-23.0	-23.2	-22.5
40%	**-0.60**	-	-21.3	-21.9	-22.3	-22.5	-22.0

	-0.50	-	-21.7	-22.3	-22.7	-22.9	-22.4
30%	**-0.60**	-	-19.7	-20.8	-21.4	-21.9	-20.9
	-0.50	-	-20.3	-21.3	-21.9	-22.3	-21.4
20%	**-0.60**	-	-	-19.6	-19.1	-20.1	-19.6
	-0.50	-	-	20.0	-19.8	-20.7	-20.2

At this scenario (Table 150), the algorithm, when using the 68 filters DC set (DC-7-4-10), does not calculate all of the edge 39 pose trees Filters Responses but just the half of them (19 or 20) according to the face direction. This gives an extra execution time saving. The same thing also applies on the next table (Table 151) where the execution time profit when multiple faces exist within an image having the same scale and looking at the same direction covering the viewing angle of 0 to +90 or 0 to -90 degrees. Of course, the 10% to 15% of the 68 filters DC set is not a product of these Filters Responses skipped but also the 31 Filters Responses skipped on the detection empty levels.

Table 151 - PPD Patch Execution Time Reduction per DC Set (0°→±90° Scenario) (%)

Faces Size	Threshold	320x240	640x480	800x600	1024x768	1280x960	Average
	DC-7-4-10						
50%	**-0.70**	-30.6	-34.7	-36.0	-36.6	-36.8	-34.9
	-0.60	-32.1	-37.0	-38.2	-38.9	-39.3	-37.1
40%	**-0.70**	-	-31.8	-34.0	-35.1	-35.7	-34.1
	-0.60	-	-34.5	-36.5	-37.6	-38.3	-36.7
30%	**-0.70**	-	-25.6	-29.6	-31.9	-33.3	-30.1
	-0.60	-	-29.2	-32.7	-34.9	-36.3	-33.3
20%	**-0.70**	-	-	-26.6	-23.3	-26.4	-25.5
	-0.60	-	-	-28.9	-27.3	-30.6	-28.9
	DC-7-3-11						
50%	**-0.60**	-18.0	-20.8	-21.4	-21.8	-22.0	-20.8
	-0.50	-18.7	-21.3	-22.0	-22.4	-22.7	-21.4
40%	**-0.60**	-	-19.4	-20.5	-21.1	-21.5	-20.6
	-0.50	-	-20.1	-21.1	-21.8	-22.2	-21.3
30%	**-0.60**	-	-16.4	-18.4	-19.6	-20.4	-18.7
	-0.50	-	-17.4	-19.2	-20.4	-21.2	-19.5
20%	**-0.60**	-	-	-16.2	-15.3	-17.2	-16.2
	-0.50	-	-	-16.9	-16.5	-18.3	-17.2

At the next last table (Table 152) the results of the 68 filters one DC set are also presented. This is because this table scenario produces the same results with the 68 filters Model when the 68

filters DC sets (C-7, C-7-4-10) are used. This way a comparison between these two sets can be done.

Table 152 - PPD Patch Execution Time Reduction per DC Set (-45°→+45° Scenario) (%)							
Faces Size	**Threshold**	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
	DC-7						
50%	**-0.70**	-38.6	-41.9	-43.0	-43.4	-43.6	-42.1
	-0.60	-39.8	-43.8	-44.8	-45.3	-45.6	-43.8
40%	**-0.70**	-	-39.6	-41.3	-42.2	-42.7	-41.5
	-0.60	-	-41.8	-43.3	-44.3	-44.9	-43.6
30%	**-0.70**	-	-34.5	-37.8	-39.6	-40.8	-38.2
	-0.60	-	-37.5	-40.3	-42.1	-43.2	-40.8
20%	**-0.70**	-	-	-35.4	-32.7	-35.2	-34.4
	-0.60	-	-	-37.2	-35.9	-38.6	-37.2
	DC-7-4-10						
50%	**-0.70**	-36.0	-38.3	-39.0	-39.3	-39.4	-38.4
	-0.60	-36.8	-39.5	-40.2	-40.5	-40.7	-39.6
40%	**-0.70**	-	-36.7	-37.9	-38.5	-38.8	-38.0
	-0.60	-	-38.2	-39.2	-39.9	-40.2	-39.4
30%	**-0.70**	-	-33.4	-35.5	-36.8	-37.5	-35.8
	-0.60	-	-35.3	-37.2	-38.4	-39.1	-37.5
20%	**-0.70**	-	-	-33.9	-32.1	-33.8	-33.3
	-0.60	-	-	-35.1	-34.2	-36.0	-35.1
	DC-7-3-11						
50%	**-0.60**	-15.4	-19.4	-20.4	-20.9	-21.2	-19.5
	-0.50	-16.4	-20.2	-21.2	-21.8	-22.1	-20.3
40%	**-0.60**	-	-17.4	-19.0	-19.9	-20.5	-19.2
	-0.50	-	-18.4	-19.9	-20.8	-21.4	-20.1
30%	**-0.60**	-	-13.1	-15.9	-17.7	-18.8	-16.4
	-0.50	-	-14.5	-17.1	-18.8	-20.0	-17.6
20%	**-0.60**	-	-	-12.8	-11.5	-14.2	-12.8
	-0.50	-	-	-13.7	-13.2	-15.8	-14.2

As seen in the Table 152 above, when centered faces exist within an image the 99 filters DC set appears very lower execution time profit as, except of calculating the 31 Filters Responses of the edge pose trees, it also uses two 39 parts pose trees on the face detection section of the DC patch that is useless.

It is also significant to be referred that the execution time profit difference between the two 68 filters DC sets is actually insignificant. The usage of one DC instead of three does not actually offers any noticeable execution time saving and probably does not worth the usage in contrast to the performance impact it has.

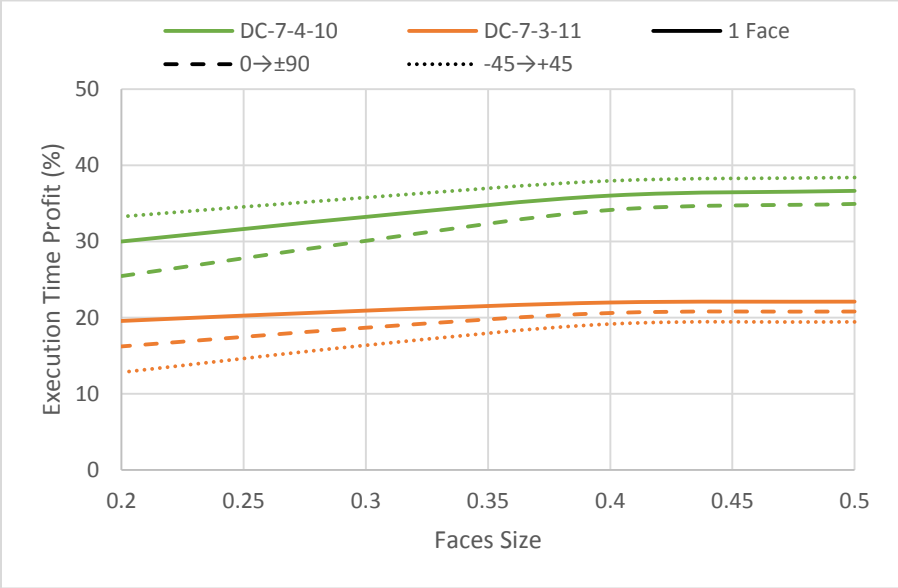


Diagram 97 - Pose Peak Detection Patch DC Sets Execution Time Profit

As seen in the Diagram 97 above the 68 filters DC Set is always faster than the 99 filters one. That's because of the Filters Responses that are not calculated in the detection empty levels. The fake detections useless processing does not seems to be able to reduce this performance. As far as the 68 filters DC set, it is obvious that it performs better on centered faces while the 99 one prefers the more devious ones. Although for both DC sets the faces size makes the differences more intensive as it is getting smaller, leading its detections to the top levels of the pyramid.

9.7.3. Level Peak Detection

In this chapter the "Levels Peak Detection" Patch is described. As referred in chapter 6.2, every face within an image produces a large number of high-score values both on nearby components and levels. This means that when a face detection is located in a level, then the same face would be detected also in previous and next levels. The LPD patch tries to discover this depictions and terminate the pose estimation procedure for this face.

As happens with the neighbor components (PPD Patch, Chapter 9.7.2), the same happens with the high-score values of the neighbor levels. In the Diagram 98 (left) below the high-score values of components 5 to 9 across the features pyramid levels of the Diagram 96 (Chapter 9.7.2), are shown. As seen in this graph all the components highest-score curves across the levels are

creating a peak highlighted with red color in the same graph. The LPD patch is trying to locate this peak and terminate the pose estimation search for this face. After applying the LPD patch, the same image produces the Diagram 98 (Right). As seen in this graph, as soon as the highest peak across the components and levels is discovered the algorithm stops searching for the face's pose estimation as it considers this procedure completed.

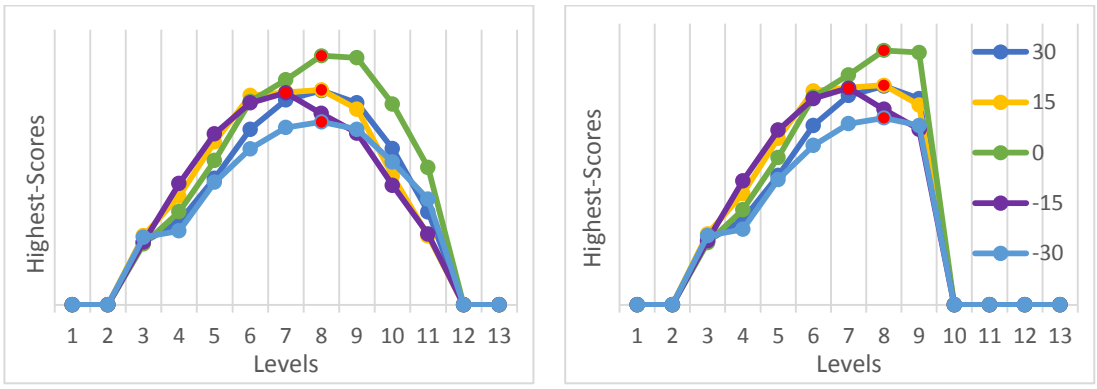


Diagram 98 - Level Peak Detection Patch Example

The combination of the PPD and the LPD patches gives the completed Fast Pose Estimation Patch. In the Diagram 99 below the final highest-score results of the Diagram 96 (Chapter 9.7.2) when the FPE patch is completely used is shown. What is gained is not only the less pose trees component stage executions but also less component stage executions per pose model.

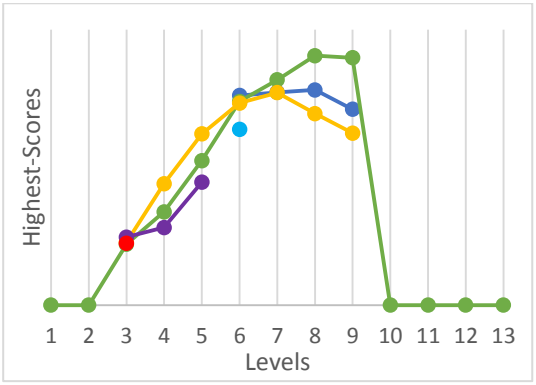


Diagram 99 - Fast Pose Estimation Patch Example for TSM v3.2.2

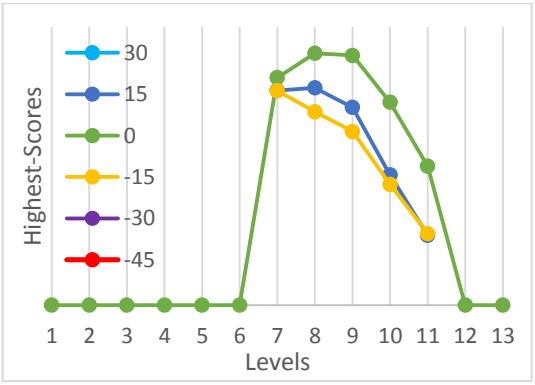


Diagram 100 - Fast Pose Estimation Patch Example for TSM v2.2.2

As seen in the Diagram 99, the algorithm skips the execution of the component stage for the -15^{O} and 15^{O} pose trees for two levels. These two levels are the ones with the smallest features images so the execution time saved is the least. In Chapter 6.18, the version 2.2.x of the algorithm was presented. In this version the algorithm executes the Level stage descending, starting from the bottom level. The Diagram 99 at this version would look like more with the

Diagram 100 at its right. At this version the execution time saving would be much larger than the 3.2.x versions as the skipped levels would correspond to the largest features images.

The LPD patch combined with the PPD one make up the Fast Pose Estimation patch. After testing these two patches together the results of the algorithm are as follow in the Table 153 below. As seen, the algorithm performance results did not changed at all by the application of the LPD patch. The same applies for all the rest face detection threshold parameter values. This patch as expected does not affect at all the algorithm efficiency and reliability and this is very pleasant.

Table 153 - Pose & Level Peak Detection Patches Results (FD Threshold = -0.65) (%)							
DT Set	**DC-7-3-11**						
Threshold	**-0.65**	**-0.60**	**-0.55**	**-0.50**	**-0.45**	**-0.40**	**-0.35**
Detected	89.5	88.7	86.8	84.8	84.6	82.7	80.1
Missed	10.5	11.3	13.2	15.2	15.4	17.3	19.9
Fake	28.4	17.7	12.5	7.89	5.71	3.49	2.60
Reliability	66.1	74.5	77.2	79.1	80.5	80.3	78.5
	Without LPD Patch (only the PPD)						
Detected	89.5	88.7	86.8	84.8	84.6	82.7	80.1
Missed	10.5	11.3	13.2	15.2	15.4	17.3	19.9
Fake	28.4	17.7	12.5	7.89	5.71	3.49	2.60
Reliability	66.1	74.5	77.2	79.1	80.5	80.3	78.5

As far as the execution time reduction this patch can succeed is detection dependent as the PPD one. In some scenarios this profit is not significant as it is too small but although it is important to be presented as these results are the final results of the Fast Pose Estimation Patch. At the following results the assumption that the level highest-scores curves are symmetric or with a small slope to the bottom levels (negative round) in order not overestimated results to be presented. The maximum $Levels_{with\text{-}high\text{-}score}$ values that are produced by a detection are shown in the Table 140 (Chapter 9.6). The detection procedure is applied to all of them without the LPD patch. The LPD patch configures the amount of levels that the detection procedure is applied as shown in the Table 154 below. The new maximum $Levels_{with\text{-}high\text{-}score}$ values are shown in the Table 155. As seen in this table, the LPD patch affects mainly the larger images as these images are the ones with larger $Levels_{with\text{-}high\text{-}score}$. A peak needs at least 3 levels to be created and four to start offering a profit so as it is sensible this patches efficiency is increasing as the image size also does.

Table 154 - LPD Patch Detection Procedure Levels							
Threshold	**-0.75**	**-0.70**	**-0.65**	**-0.60**	**-0.55**	**-0.50**	**-0.45**
320x240	3	3	3	3	3	3	3
640x480	4	4	4	4	4	4	4
800x600	5	5	4	4	4	4	4
1024x768	5	5	5	5	4	4	4
1280x960	5	5	5	5	5	5	5

$$\mathbf{LPD\text{-}MaxL_{High\text{-}Scores}} = \left\lfloor \frac{\mathbf{MaxL_{High\text{-}Scores}} + 1}{2} \right\rfloor + 1 \qquad (35)$$

Table 155 - LPD Patch $MaxL_{High\text{-}Scores}$	
No Patch	**LPD Patch**
3	3
4	4
5	4
6	5
7	5
8	6
9	6
10	7

The function (35) above is the one that calculates the $Levels_{with\text{-}high\text{-}score}$ of a detection when the LPD patch is used in the algorithm.

In the Table 156 below the execution time profit of the FPE algorithm is presented as it is conformed after the usage of both PPD and LPD patches for the same scenarios as in chapter 9.6. As happens to the PPD patch, the same way the LPD one is affecting the fake detections. The fake detections have the same characteristics with the real ones so LPD patch reduces the detection procedure of the fake faces also.

Table 156 - LPD Patch Execution Time Reduction per DC Set (1 Face Scenario) (%)							
Faces Size	**Threshold**	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
	DC-7-4-10						
50%	**-0.70**	-34.3	-37.7	-38.6	-38.9	-39.1	-37.7
	-0.60	-35.1	-38.7	-39.5	-39.9	-40.1	-38.7
40%	**-0.70**	-	-35.9	-37.3	-38.0	-38.4	-37.4
	-0.60	-	-37.2	-38.4	-39.2	-39.6	-38.6
30%	**-0.70**	-	-32.1	-34.6	-36.1	-37.0	-35.0
	-0.60	-	-33.9	-36.1	-37.5	-38.3	-36.5
20%	**-0.70**	-	-	-33.6	-31.0	-32.8	-32.5
	-0.60	-	-	-34.3	-32.8	-34.9	-34.0
	DC-7-3-11						
50%	**-0.60**	-20.7	-22.4	-22.7	-22.9	-23.1	-22.3
	-0.50	-21.0	-22.6	-22.9	-23.2	-23.3	-22.6
40%	**-0.60**	-	-21.7	-22.2	-22.6	-22.8	-22.3
	-0.50	-	-22.0	-22.5	-22.9	-23.1	-22.6
30%	**-0.60**	-	-20.2	-21.2	-21.8	-22.3	-21.4

	-0.50	-	-20.6	-21.6	-22.2	-22.6	-21.7
20%	**-0.60**	-	-	-20.4	-19.8	-20.7	-20.3
	-0.50	-	-	-20.5	-20.3	-21.2	-20.7

The results of the LPD patch in the result Table 156, Table 157 and Table 158 shows that this patch has greater impact on the large images than in the small ones. This is sensible as large image detections create large detection range over the levels giving the LPD patch the space to operate. Low Threshold parameter values also enables the LPD patch as this parameter affects the detection range either when it is for real detections or when it is for the fake ones that are increasing. The impact of this patch is increasing as the detected faces size is reducing because the faces detection level is getting closer to the top one where the impact of skipping even one level is getting greater.

Table 157 - LPD Patch Execution Time Reduction per DC Set (0°→±90° Scenario) (%)							
Faces Size	**Threshold**	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
	DC-7-4-10						
50%	**-0.70**	-31.0	-36.2	-37.4	-38.1	-38.4	-36.2
	-0.60	-32.2	-37.8	-39.0	-39.7	-40.1	-37.7
40%	**-0.70**	-	-33.6	-35.6	-36.8	-37.4	-35.9
	-0.60	-	-35.5	-37.4	-38.6	-39.3	-37.7
30%	**-0.70**	-	-28.1	-31.8	-34.0	-35.3	-32.3
	-0.60	-	-30.7	-34.1	-36.2	-37.5	-34.6
20%	**-0.70**	-	-	-30.3	-26.6	-29.3	-28.7
	-0.60	-	-	-31.3	-29.4	-32.5	-31.1
	DC-7-3-11						
50%	**-0.60**	-18.1	-21.2	-21.9	-22.3	-22.5	-21.2
	-0.50	-18.7	-21.6	-22.2	-22.7	-22.9	-21.6
40%	**-0.60**	-	-19.9	-21.0	-21.6	-22.0	-21.2
	-0.50	-	-20.5	-21.4	-22.1	-22.5	-21.6
30%	**-0.60**	-	-17.2	-19.1	-20.3	-21.0	-19.4
	-0.50	-	-18.0	-19.7	-20.9	-21.6	-20.0
20%	**-0.60**	-	-	-17.6	-16.5	-18.2	-17.4
	-0.50	-	-	-17.8	-17.4	-19.0	-18.1

Table 158 - LPD Patch Execution Time Reduction per DC Set (-45°→+45° Scenario) (%)							
Faces Size	**Threshold**	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
	DC-7-4-10						
50%	**-0.70**	-36.3	-39.1	-39.8	-40.1	-40.3	-39.1
	-0.60	-36.9	-39.9	-40.6	-41.0	-41.2	-39.9

40%	**-0.70**	-	-37.7	-38.8	-39.4	-39.7	-38.9
	-0.60	-	-38.7	-39.7	-40.4	-40.7	-39.9
30%	**-0.70**	-	-34.7	-36.7	-37.9	-38.6	-37.0
	-0.60	-	-36.1	-37.9	-39.1	-39.8	-38.2
20%	**-0.70**	-	-	-35.9	-33.9	-35.4	-35.0
	-0.60	-	-	-36.5	-35.4	-37.1	-36.3
	DC-7-3-11						
50%	**-0.60**	-15.5	-20.0	-21.0	-21.6	-21.9	-20.0
	-0.50	-16.4	-20.6	-21.5	-22.1	-22.5	-20.6
40%	**-0.60**	-	-18.2	-19.7	-20.7	-21.2	-19.9
	-0.50	-	-18.9	-20.4	-21.3	-21.9	-20.6
30%	**-0.60**	-	-14.3	-17.0	-18.7	-19.8	-17.4
	-0.50	-	-15.3	-17.9	-19.5	-20.6	-18.3
20%	**-0.60**	-	-	-14.8	-13.2	-15.7	-14.6
	-0.50	-	-	-15.2	-14.5	-16.9	-15.5

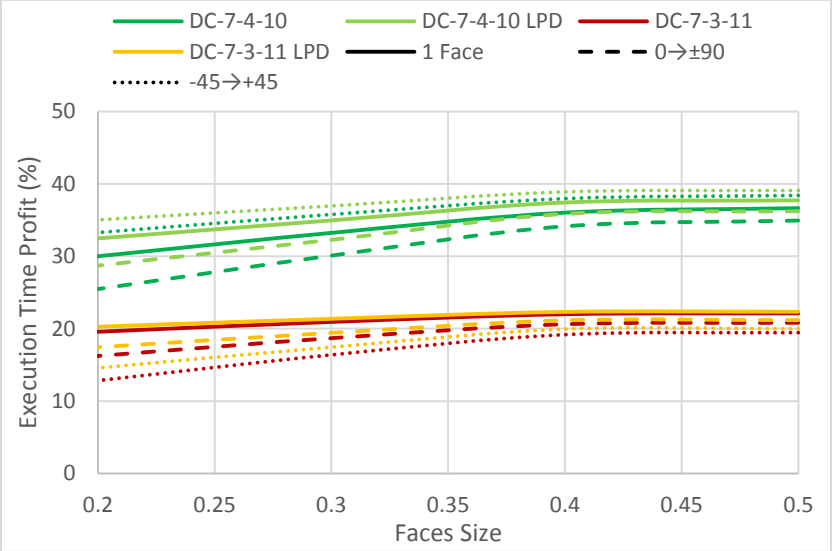


Diagram 101 - Level Peak Detection Patch DC Sets Execution Time Profit

Looking at the Diagram 101 above, what is obvious is that the LPD patch is just saving only a small amount of execution time. This amount is not over 3% and it could be considered insignificant for using this patch in the TSM algorithm. A reason for this small execution time profit is the fact that the levels skipped by the LPD patch is the smallest ones in the detection range of a detection (real or fake). In chapter X the version 2.2.2 is presented. In this version the algorithm forwards the pyramid levels to the detection procedure starting from the bottom to the top, exactly the opposite way the version 3.2.2 does (the main version). In the v2.2.2 of the algorithm the LPD patch would skip the largest levels of the detection range instead of the small

ones. This means that in this version the impact of the LPD patch would be greater than the v3.2.2. In the following tables the results of the LPD patch using the version 2.2.2 of the algorithm are presented.

Table 159 - LPD Patch Execution Time Reduction per DC Set (1 Face Scenario) (v2.2.2) (%)							
Faces Size	**Threshold**	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
	DC-7-4-10						
50%	**-0.70**	-34.4	-38.7	-39.4	-39.9	-40.0	-38.5
	-0.60	-35.1	-39.3	-40.0	-40.5	-40.7	-39.1
40%	**-0.70**	-	-37.4	-38.6	-39.3	-39.7	-38.7
	-0.60	-	-38.0	-39.2	-39.9	-40.3	-39.4
30%	**-0.70**	-	-34.5	-36.7	-38.1	-38.8	-37.0
	-0.60	-	-35.3	-37.4	-38.8	-39.5	-37.8
20%	**-0.70**	-	-	-32.0	-34.6	-36.5	-34.4
	-0.60	-	-	-33.0	-35.5	-37.3	-35.3
	DC-7-3-11						
50%	**-0.60**	-20.7	-22.6	-22.9	-23.1	-23.2	-22.5
	-0.50	-21.0	-22.8	-23.1	-23.3	-23.4	-22.7
40%	**-0.60**	-	-22.0	-22.5	-22.9	-23.1	-22.6
	-0.50	-	-22.2	-22.7	-23.1	-23.3	-22.8
30%	**-0.60**	-	-20.8	-21.7	-22.4	-22.7	-21.9
	-0.50	-	-21.0	-21.9	-22.6	-22.9	-22.1
20%	**-0.60**	-	-	-19.8	-20.9	-21.7	-20.8
	-0.50	-	-	-20.0	-21.1	-22.0	-21.1

Table 160 - LPD Patch Execution Time Reduction per DC Set (0°→±90° Scenario) (v2.2.2) (%)							
Faces Size	**Threshold**	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
	DC-7-4-10						
50%	**-0.70**	-31.0	-37.5	-38.6	-39.2	-39.5	-37.2
	-0.60	-32.2	-38.5	-39.6	-40.3	-40.7	-38.3
40%	**-0.70**	-	-35.6	-37.3	-38.4	-39.0	-37.6
	-0.60	-	-36.7	-38.4	-39.5	-40.2	-38.7
30%	**-0.70**	-	-31.4	-34.6	-36.6	-37.8	-35.1
	-0.60	-	-32.7	-35.8	-37.8	-39.0	-36.3
20%	**-0.70**	-	-	-27.8	-31.6	-34.4	-31.3
	-0.60	-	-	-29.4	-33.1	-35.8	-32.8
	DC-7-3-11						
50%	**-0.60**	-18.1	-21.6	-22.2	-22.6	-22.8	-21.5
	-0.50	-18.7	-21.9	-22.5	-22.9	-23.1	-21.8

40%	**-0.60**	-	-20.6	-21.6	-22.2	-22.5	-21.7
	-0.50	-	-20.9	-21.8	-22.5	-22.9	-22.0
30%	**-0.60**	-	-18.4	-20.1	-21.2	-21.9	-20.4
	-0.50	-	-18.8	-20.4	-21.6	-22.2	-20.7
20%	**-0.60**	-	-	-16.5	-18.6	-20.1	-18.4
	-0.50	-	-	-17.0	-19.0	-20.5	-18.8

Table 161 - LPD Patch Execution Time Reduction per DC Set (-45°→45° Scenario) (v2.2.2) (%)							
Faces Size	**Threshold**	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
	DC-7-4-10						
50%	**-0.70**	-36.3	-39.8	-40.4	-40.7	-40.9	-39.6
	-0.60	-36.9	-40.3	-40.9	-41.3	-41.5	-40.2
40%	**-0.70**	-	-38.7	-39.7	-40.3	-40.6	-39.8
	-0.60	-	-39.4	-40.3	-40.9	-41.2	-40.4
30%	**-0.70**	-	-36.5	-38.2	-39.3	-39.9	-38.5
	-0.60	-	-37.2	-38.9	-40.0	40.6	-39.2
20%	**-0.70**	-	-	-34.5	-36.6	-38.1	-36.4
	-0.60	-	-	-35.4	-37.4	-38.9	-37.2
	DC-7-3-11						
50%	**-0.60**	-15.5	-20.6	-21.5	-22.1	-22.4	-20.4
	-0.50	-16.4	-21.0	-21.9	-22.5	-22.8	-20.9
40%	**-0.60**	-	-19.1	-20.5	-21.4	-22.0	-20.8
	-0.50	-	-19.6	-20.9	-21.9	-22.4	-21.2
30%	**-0.60**	-	-15.9	-18.4	-20.1	-21.0	-18.9
	-0.50	-	-16.5	-18.9	-20.6	-21.5	-19.4
20%	**-0.60**	-	-	-13.2	-16.2	-18.4	-16.0
	-0.50	-	-	-13.9	-16.8	-19.0	-16.6

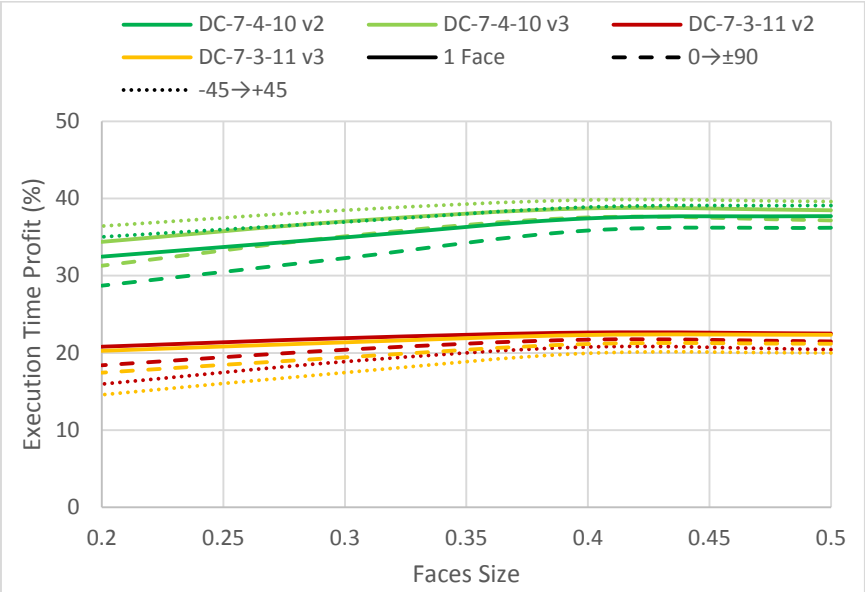


Diagram 102 - Level Peak Detection Patch DC Sets Execution Time Profit (TSM v2.2.2)

The Diagram 102 shows the results of the results tables above. As is seen in this graph the extra execution time saving came from the usage of the version 2.2.2 of the TSM algorithm is tiny in order this version to be considered for replacing the 3.2.2 one as the main version of the algorithm.

The conclusion for this patch is that it offers an insignificant execution time profit to the TSM algorithm but it also does not cost anything in its detection performance. So, it is subjective if it is worth to use or not. Either ways it does not cost anything to the detection performance and there is no crucial reason for not using it. Many small execution time savings can produce a larger one like the Short Pyramid patch (Chapter 7).

9.8. Pyramid Fast Pass

In this chapter an extra patch for gaining execution time on the TSM algorithm is appose. This patch was inspired by the DC patch and the reduction of the interval parameter value. As introduced in the chapter 9.3 the reduction of the Interval parameter causes significant reduction of the algorithms detection performance. On this chapter a new technique is introduced that succeeds an important speedup without a significant reducing the algorithm detection performance.

The “Pyramid Fast Pass” patch is using the DC patch with an extended procedure. In the PFP patch if the algorithm detects a face within the image in the face detection section then it forwards the specific level to the pose estimation section. On this patch the algorithm does not pass to the face detection section all the levels sequential but with a step of two levels. Starting

from the second level of the pyramid if the face detection section does not make any detection the next level passed will be the after the next level. On the other hand if the face detection section makes a detection then the next one will be passed to the same section as shown in the Figure 70 below.

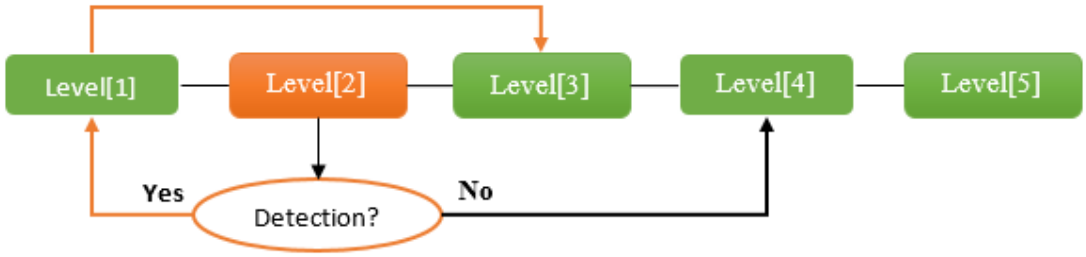


Figure 70 - Pyramid Fast Pass Patch Execution Flow Diagram

The idea behind this patch is that if a face exists within the image then it will produce high-score values in more than one levels. If the algorithm checks for detections within the levels with a step of two then it will detect this face. After it detects it then it will check the levels near it for a more accurate detection. This can lead to the detection of more faces if they are in the same scale or close it. By starting from the second level of the pyramid, the algorithm has the chance to skip the greatest image size level meaning a reduction of the execution time by about 25%. This is e very good deal.

At this point the usage of the Level Peak Detection patch would be very useful. According to the previous paragraph the algorithm has to check for detections all the levels where the detection range spreads plus two extra levels where the algorithm will not find anything and it will enable the double stepping again. This can be avoided if the LPD patch is used because as the algorithm makes a detection it can continue passing the next levels with a step of one until it detects the detection level peak curve. Looking at the level curve, the LPD patch can decide if the algorithm will have to continue passing the next levels or it will have to go back and apply the detection section to the last skipped level. The LPD patch would also decide when it is the time to increase the level step back to two as shown in Figure 71.

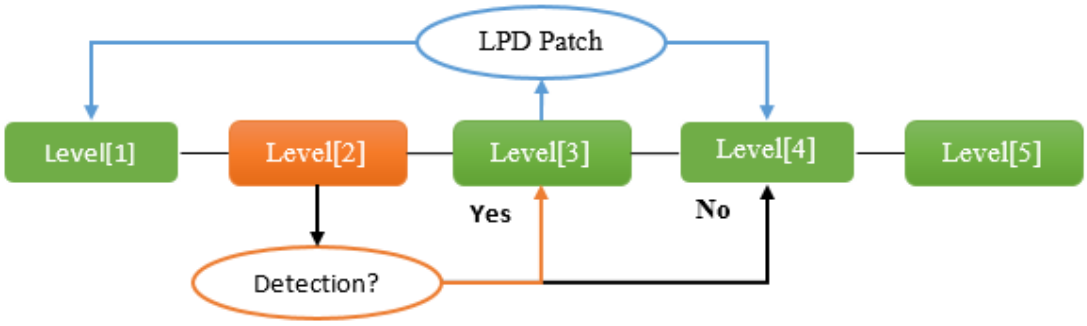


Figure 71 - Pyramid Fast Pass & LPD Patch Execution Flow Diagram

In the Table 85 in chapter 7.2, the average number of levels with high-scores appear in the features pyramid when a face exists within the image is shown. As seen in this table it is very important the face detection section threshold parameter value to create high-score values to more than one level on average so that this patch will not bypass detections. As seen in the values of Table 162 not all image sizes are able to create high-score values in enough levels when a face exists within them, making this patch able to be applied without causing serious detection skips. As is visible the small size image of 320x240 is just on the limit so the image size should be considerable for the usage of this patch.

Table 162 - PFP Patch |$Levels_{High\text{-}Scores}$|Results per Threshold							
Threshold	**-0.75**	**-0.70**	**-0.65**	**-0.60**	**-0.55**	**-0.50**	**-0.45**
320X240	3	2	2	2	2	2	2
640x480	5	5	4	4	4	4	3
800x600	5	5	5	4	4	4	4
1024x768	6	6	5	5	5	5	4
1280x960	7	7	6	6	6	5	5

By testing the face detection section using the Half Pyramid patch the following results come of, as shown in Table 163 below.

Table 163 - Pyramid Fast Pass Patch Face Detection Section Results (%)							
Threshold	**-0.75**	**-0.70**	**-0.65**	**-0.60**	**-0.55**	**-0.50**	**-0.45**
DC Set	**DC-7-4-10**						
Detected	86.8	84.2	80.8	78.4	76.9	73.9	70.5
Missed	13.2	15.8	19.2	21.6	23.1	26.1	29.5
Fake	37.6	30.8	22.1	13.6	8.40	4.95	2.08
Reliability	56.9	61.3	65.7	69.8	71.9	71.2	69.5
DC Set	**DC-7-3-11**						
Detected	-	-	85.0	83.3	81.6	79.7	78.0

Missed	-	-	15.0	16.7	18.4	20.3	22.0
Fake	-	-	16.4	6.70	3.78	2.61	1.08
Reliability	-	-	72.9	78.6	79.1	78.0	77.3

As seen in the Table 163, the detection efficiency of the Detection section is reducing as the Threshold parameter value is increasing. It is very obvious that the 99 filters DC set is much more efficient than the 68 filters one. The 99 filters DC set reaches its maximum reliability without losing an important part of its detection efficiency when the 68 filters DC set appears a great loss. In the next tables (Table 164 and Table 165) the impact of all these results in the final detection results of the algorithm are shown.

Table 164 - Pyramid Fast Pass & LPD Patch Results (DC Set 7-4-10) (%)						
FD Threshold	**-0.70**			**-0.65**		
Threshold	**-0.50**	**-0.45**	**-0.40**	**-0.50**	**-0.45**	**-0.40**
Detected	84.6	84.2	82.3	84.2	83.8	82.1
Missed	15.4	15.8	17.7	15.8	16.2	17.9
Fake	7.91	5.74	3.51	7.51	5.31	3.27
Reliability	78.9	**80.1**	79.9	78.8	80.0	79.8
FD Threshold	**-0.60**			**-0.55**		
Threshold	**-0.50**	**-0.45**	**-0.40**	**-0.50**	**-0.45**	**-0.40**
Detected	82.3	81.8	80.3	79.9	79.3	78.2
Missed	17.7	18.2	19.7	20.1	20.7	21.8
Fake	5.41	3.77	2.59	5.32	3.64	2.66
Reliability	78.6	79.3	78.7	76.5	77.0	76.6

Table 165 - Pyramid Fast Pass & LPD Patch Results (DC Set 7-3-11) (%)						
FD Threshold	**-0.70**			**-0.65**		
Threshold	**-0.50**	**-0.45**	**-0.40**	**-0.50**	**-0.45**	**-0.40**
Detected	85.5	84.8	83.1	85.0	84.4	82.7
Missed	14.5	15.2	16.9	15.0	15.6	17.3
Fake	9.50	6.37	3.95	9.34	6.40	3.97
Reliability	78.4	80.2	**80.4**	78.2	79.8	80.0
FD Threshold	**-0.60**			**-0.55**		
Threshold	**-0.50**	**-0.45**	**-0.40**	**-0.50**	**-0.45**	**-0.40**
Detected	83.1	82.7	81.4	81.6	81.0	79.5
Missed	16.9	17.3	18.6	18.4	19.0	20.5
Fake	5.58	3.73	2.56	5.45	3.32	2.36
Reliability	79.2	80.1	79.7	78.0	78.8	78.0

As is visible in the Table 165, as smaller is the Detection section threshold variable the better is the detection results. As is sensible, decreasing the detection section threshold variable can produce the same results as without using the PFP patch. If this reduction is applied the speedup of the algorithm will be reduced. In the 99 filters DC set the results are much better than those in the 68 filters one. This means that a greater detection threshold value can be used. The choice of the DC set and the detection threshold parameter value is a difficult one as every possible combination offer different pros and cons. It is a matter of the goals are set to the algorithm. If the reliability is the greatest factor the algorithm set up would be different than when the execution time is more important.

As far as the execution time that can be saved using the PFP patch the Table 166 below can show the profit of every level skipped to the whole TSM algorithm execution time. As seen in this table, with the bold text, the profit on the execution time is summed at the last line of the table. When the detection section uses the 99 filters DC set that is more accurate, the algorithm reduces its execution time by 69.4% on empty faces images. In the same case the algorithm reduces its execution time by 77.6% for the 68 filters DC set.

Table 166 - Pyramid Fast Pass Patch Execution Time Profit (No Face) (%)			
Levels	**Both**	**DC Set**	
		C7-3-11	**C7-4-10**
1	**-25.0**	-5.96	-10.6
2	-19.0	**-4.53**	**-8.07**
3	**-14.4**	-3.44	-6.14
4	-11.0	**-2.62**	**-4.66**
5	**-8.34**	-1.99	-3.54
6	-6.34	**-1.51**	**-2.69**
7	**-4.82**	-1.15	-2.05
8	-3.66	**-0.87**	**-1.56**
9	**-2.78**	-0.66	-1.18
10	-2.11	**-0.50**	**-0.90**
11	**-1.61**	-0.38	-0.68
12	-1.22	**-0.29**	**-0.52**
13	**-0.93**	-0.22	-0.39
14	-0.71	**-0.17**	**-0.30**
15	**-0.54**	-0.13	-0.23
16	-0.41	**-0.10**	**-0.17**
17	**-0.31**	-0.07	-0.13

The advantage of this patch is extremely good as the algorithm finish the detection procedure very quickly when the image does not contain any face. This would be very useful on video

applications where the useless frames would be skipped fast until a useful arrives. On the other hand when there are faces within the image, this patch acts differently as it is detection dependent. According to the number and the scale of the faces, the algorithm would react differently. The worst case scenario is when an image contains many faces in many scales as it would produce high-score values in all the levels of the features pyramid. A more common case is an image to contain one or more faces in the same scale like portraits or team photos. This is an average scenario where this patch can act in many ways as far as the time profit it succeeds.

As referred in the chapter 9.6, the same scenarios are used in this patch. The results on these scenarios are shown in Table 167 below.

Table 167 - Pyramid Fast Pass & LPD Patch Execution Time Reduction per DC Set (%)							
Faces Size	**Threshold**	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
	DC-7-4-10						
50%	**-0.70**	-32.9	-50.2	-54.8	-57.7	-59.6	-51.0
	-0.60	-35.6	-54.1	-58.0	-60.5	-62.2	-54.1
40%	**-0.70**	-	-41.7	-48.8	-53.4	-56.4	-50.1
	-0.60	-	-46.8	-52.8	-56.8	-59.5	-54.0
30%	**-0.70**	-	-23.5	-36.1	-44.2	-49.4	-38.3
	-0.60	-	-30.9	-41.8	-49.0	-53.7	-43.8
20%	**-0.70**	-	-	-31.1	-19.8	-29.8	-26.9
	-0.60	-	-	-32.9	-26.7	-37.1	-32.2
	DC-7-3-11						
50%	**-0.70**	-31.0	-47.4	-51.4	-53.9	-55.6	-47.9
	-0.60	-31.5	-47.8	-51.2	-53.5	-54.9	-47.8
40%	**-0.70**	-	-40.1	-46.2	-50.2	-52.8	-47.3
	-0.60	-	-41.4	-46.8	-50.3	-52.6	-47.8
30%	**-0.70**	-	-24.3	-35.2	-42.2	-46.8	-37.1
	-0.60	-	-27.7	-37.2	-43.5	-47.5	-39.0
20%	**-0.70**	-	-	-30.9	-21.0	-29.7	-27.2
	-0.60	-	-	-29.5	-24.2	-33.2	-28.9

The results on the Table 167 above are very optimistic. As seen in this table the algorithm can reduce its execution time about its half. These are very useful results. What is changed in this patch is the speedup relation between the 99 and 68 filters DC sets. In contrast to the DC the execution time profit differences are reduced as the number of levels where the filters responses are calculated is also reduced. This means that if the 99 filters DC set is used the execution time loss will not be much while the algorithms reliability and detection efficiency will be also increased for a little.

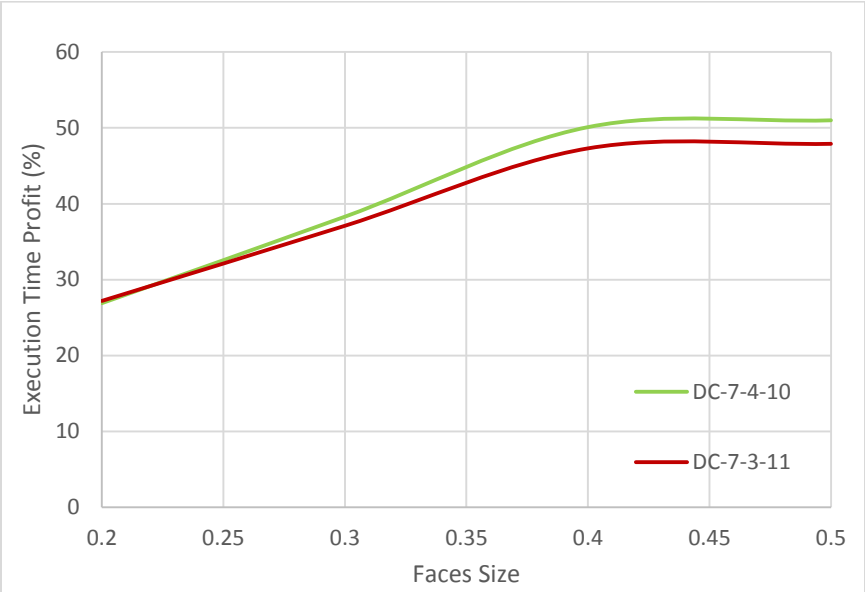


Diagram 103 – PFP & LPD Patch DC Sets Execution Time Profit

Except of the LPD patch that does not affect the algorithm detection performance, the PFP patch can also be combined with the PPD one. The PPD patch has its impact on the algorithm detection performance and for that reason it is important to test these patch together in order to known what this combination impact on the detection performance would be. By doing this the results are as the Table 168 and Table 169 shows.

Table 168 - Pyramid Fast Pass & PPD Patch Results (DC Set 7-4-10) (%)						
FD Threshold	**-0.75**			**-0.70**		
Threshold	**-0.50**	**-0.45**	**-0.40**	**-0.50**	**-0.45**	**-0.40**
Detected	83.1	82.5	81.0	82.5	82.3	80.6
Missed	16.9	17.5	19.0	17.5	17.7	19.4
Fake	9.53	6.08	3.81	9.39	5.87	3.83
Reliability	76.4	78.3	78.5	76.0	78.3	78.1
FD Threshold	**-0.65**			**-0.60**		
Threshold	**-0.50**	**-0.45**	**-0.40**	**-0.50**	**-0.45**	**-0.40**
Detected	73.9	73.1	71.2	73.9	72.9	70.9
Missed	26.1	26.9	28.8	26.1	27.1	29.1
Fake	5.21	2.84	1.77	4.95	2.85	1.78
Reliability	71.0	71.5	70.3	71.2	71.3	70.0

Table 169 - Pyramid Fast Pass & PPD Patch Results (DC Set 7-3-11) (%)		
FD Threshold	**-0.70**	**-0.65**

Threshold	-0.50	-0.45	-0.40	-0.50	-0.45	-0.40
Detected	85.0	84.8	82.9	84.0	83.5	82.1
Missed	15.0	15.2	17.1	16.0	16.5	17.9
Fake	7.87	5.25	3.48	7.31	5.10	3.27
Reliability	79.3	81.0	80.5	78.8	80.0	79.8
FD Threshold	-0.60			-0.55		
Threshold	-0.50	-0.45	-0.40	-0.50	-0.45	-0.40
Detected	82.9	82.5	81.2	82.5	82.5	81.0
Missed	17.1	17.5	18.8	17.5	17.5	19.0
Fake	6.73	4.93	3.06	6.54	4.93	2.32
Reliability	78.2	79.1	79.2	78.0	79.1	79.5

As is seen in these tables the 99 filters DC set keeps having good performance with only a small reduction in its reliability which is a very good result. On the other hand when the PFP and PPD patches are combined with the 68 filters DC the performance is greatly reduced if the FD Threshold parameter value is low. These means that these two patches can be efficiently combined giving time performance speedup with only a small reduction in the detection performance.

As far as the execution time saving from the usage of the PPD patch in combination with the PFP one the results are as shown in the Table 170, Table 171 and Table 172 below according to the scenario described in chapter 9.6.

Table 170 - PFP & PPD Patch Execution Time Reduction per DC Set (1 Face) (%)							
Faces Size	Threshold	320x240	640x480	800x600	1024x768	1280x960	Average
	DC-7-4-10						
50%	**-0.70**	-47.6	-58.0	-60.8	-62.6	-63.7	-58.5
	-0.60	-49.3	-60.4	-62.7	-64.3	-65.3	-60.4
40%	**-0.70**	-	-53.0	-57.2	-60.0	-61.8	-58.0
	-0.60	-	-56.0	-59.6	-62.1	-63.6	-60.3
30%	**-0.70**	-	-42.0	-49.6	-54.4	-57.6	-50.9
	-0.60	-	-46.4	-53.0	-57.3	-60.1	-54.2
20%	**-0.70**	-	-	-46.6	-39.7	-45.7	-44.0
	-0.60	-	-	-47.6	-43.9	-50.2	-47.2
	DC-7-3-11						
50%	**-0.70**	-39.4	-51.2	-54.0	-55.8	-57.0	-51.5
	-0.60	-39.8	-51.5	-53.9	-55.5	-56.5	-51.4
40%	**-0.70**	-	-45.9	-50.3	-53.2	-55.0	-51.1

	-0.60	-	-46.9	-50.7	-53.2	-54.9	-51.4
30%	**-0.70**	-	-34.7	-42.4	-47.5	-50.7	-43.8
	-0.60	-	-37.1	-43.9	-48.4	-51.2	-45.1
20%	**-0.70**	-	-	-39.4	-32.3	-38.5	-36.7
	-0.60	-	-	-38.4	-34.6	-41.0	-38.0

Table 171 - PFP & PPD Patch Execution Time Reduction per DC Set (0°→±90°) (%)							
Faces Size	**Threshold**	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
	DC-7-4-10						
50%	**-0.70**	-44.1	-56.2	-59.4	-61.4	-62.7	-56.8
	-0.60	-46.0	-58.9	-61.6	-63.4	-64.5	-58.9
40%	**-0.70**	-	-50.3	-55.2	-58.4	-60.5	-56.1
	-0.60	-	-53.8	-58.0	-60.8	-62.7	-58.8
30%	**-0.70**	-	-37.6	-46.4	-52.0	-55.7	-47.9
	-0.60	-	-42.8	-50.4	-55.4	-58.6	-51.8
20%	**-0.70**	-	-	-42.9	-35.0	-42.0	-40.0
	-0.60	-	-	-44.2	-39.8	-47.1	-43.7
	DC-7-3-11						
50%	**-0.70**	-36.8	-50.0	-53.2	-55.2	-56.6	-50.4
	-0.60	-37.2	-50.3	-53.1	-54.9	-56.0	-50.3
40%	**-0.70**	-	-44.1	-49.1	-52.3	-54.3	-49.9
	-0.60	-	-45.2	-49.5	-52.3	-54.2	-50.3
30%	**-0.70**	-	-31.5	-40.2	-45.8	-49.5	-41.7
	-0.60	-	-34.2	-41.8	-46.9	-50.1	-43.2
20%	**-0.70**	-	-	-36.7	-28.8	-35.8	-33.8
	-0.60	-	-	-35.6	-31.3	-38.6	-35.2

Table 172 - PFP & PPD Patch Execution Time Reduction per DC Set (-45°→+45°) (%)							
Faces Size	**Threshold**	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
	DC-7-4-10						
50%	**-0.70**	-43.7	-56.0	-59.2	-61.3	-62.6	-56.5
	-0.60	-45.6	-58.7	-61.5	-63.3	-64.5	-58.7
40%	**-0.70**	-	-50.0	-55.0	-58.2	-60.3	-55.9
	-0.60	-	-53.5	-57.8	-60.7	-62.5	-58.6
30%	**-0.70**	-	-37.1	-46.0	-51.7	-55.4	-47.6
	-0.60	-	-42.3	-50.0	-55.1	-58.4	-51.5
20%	**-0.70**	-	-	-42.5	-34.4	-41.5	-39.5
	-0.60	-	-	-43.7	-39.3	-46.7	-43.3

	DC-7-3-11						
50%	**-0.70**	-34.2	-48.9	-52.4	-54.7	-56.1	-49.2
	-0.60	-34.6	-49.2	-52.2	-54.2	-55.5	-49.2
40%	**-0.70**	-	-42.3	-47.8	-51.3	-53.6	-48.8
	-0.60	-	-43.5	-48.3	-51.4	-53.4	-49.2
30%	**-0.70**	-	-28.3	-38.0	-44.2	-48.3	-39.7
	-0.60	-	-31.3	-39.8	-45.3	-48.9	-41.3
20%	**-0.70**	-	-	-34.1	-25.3	-33.1	-30.9
	-0.60	-	-	-32.9	-28.1	-36.2	-32.4

As seen in the execution time saving tables above the usage of the PPD patch in combination to the PFP one offer an execution time profit of about 5-10% which is more significant in small size images and less in the larger ones. As it is sensible the most significant part of the algorithm is the Convolution stage. Every time the filters responses calculation is skipped the execution time benefits are increasing significantly.

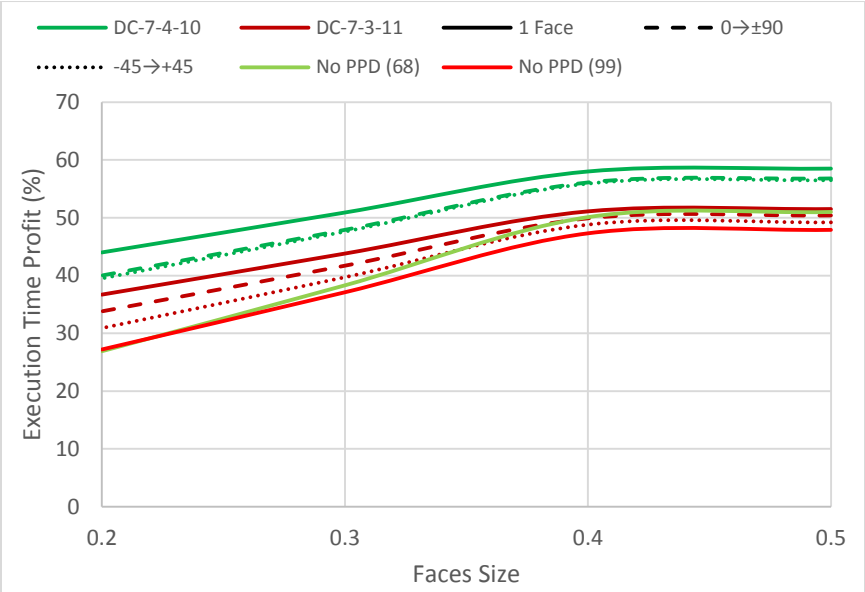


Diagram 104 – PFP & PPD Patch DC Sets Execution Time Profit

10. Related Comparison

In this chapter the comparison of this thesis implementation with the [25] one provided by the creators of the TSM system is presented. There are also other related algorithms as referred in chapter 3, but only a short description and tasks support comparison is appose in the next subchapters as related and no similar systems cannot easily compared. There are also no freeware implementation on C\C++ to many of the related systems as [3], [4], [6], [7], [8], [9] and [10]. On the other hand there are some freeware systems free to use in the web but they do not implement the same tasks in order to be compared as far as the time performance with this thesis and [25] implementation.

Table 173 – Tests Hardware Specifications			
	System 1	**System 2**	**System 3**
CPU Model	Intel Core i7-4600U @2.70GHz	Intel Core 2 Duo T8100 @2.10GHz	Dual Core ARM Cortex-A9 @866MHz
CPU Cores	4	2	2
RAM Memory	8 GB	4 GB	512 MB
Operating System	VM Ubuntu 15.01 (no GUI)	Ubuntu 14.04 (no GUI)	Ubuntu 12.04 (no GUI)
v3.2.2/Creators[25]	-56.3 %	-57.2 %	-63.4 %

As far as the [25] implementation of Hang Su, the open source code provided had to be customized as it uses some extra methods for making the detection process faster like scaling the input image to a small size one. This method makes the face detection process faster but it avoids the detection of small faces as described in chapter 9.4. This thesis implementation does not reduce the input image size and it was sensible that only same procedure systems can be compared. Studding the Hand Su C\C++ code of his implementation what is noted is that it is very similar to the TSM v1.2 described in chapter 6.5.

Testing these two implementation in the same hardware resources as the ones shown in Table 173 the following results came as the ones presented in Table 174.

Table 174 - TSM v3.2.2 vs Creators Execution Time (%)							
System	**CPU Cores**	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**	**Average**
1	1	-50.9	-38.6	-32.5	-31.7	-29.4	-36.6
	2	-61.6	-46.1	-42.1	-41.4	-39.6	-46.2
	3	-61.6	-49.9	-50.4	-47.4	-47.6	-51.4
	4	-65.2	-53.2	-55.5	-54.1	-53.4	-56.3
2	2	-76.5	-54.4	-52.7	-52.5	-49.6	-57.2

3	2	-76.6	-56.7	-56.9	Out of memory	Out of memory	-63.4

In Table 174 it is visible that the TSM v3.2.2 algorithm implementation is getting faster as the number of CPU cores in the hardware is increasing. This is because the original version implemented by Hang Su is using the multithreading technique only in the convolution procedure and nowhere else. This makes us assume that the memory consumption of this implementation is similar to the memory consumption of the versions 1.2 or 1.3. On the other hand the absence of parallelization in the rest parts of the algorithm (ex. DT stage) makes it getting slower as the parallelization resources increase compared to the version 3.2.2.

As referred in chapter 3 there are algorithms designed since the [1] published that some of them claim to have better detection performance and others to be faster. There are also some freeware libraries offering face detection implementing some of them. In the next subchapters the differences between these systems and the [1] is appose.

10.1. Freeware Libraries

Some of the related systems to this thesis are offered freely in the web ready to be used by anyone. The following subchapters present some of them and describe the differences between these ones and the implementation of this thesis and [25] that are based on [1] face detection method.

10.1.1. OpenCV

The OpenCV [27] library is the most famous and most used one. It uses the face detection method proposed by Viola and Jones in 2001 [17] and it is the most famous face detection algorithm. This algorithm is very fast but it only supports face detection without pose estimation and landmark localization. It is also efficient in frontal face detection. Although it is very famous it lacks on detection performance. Despite that it was the state-of-the-art algorithm of face detection task for many years.

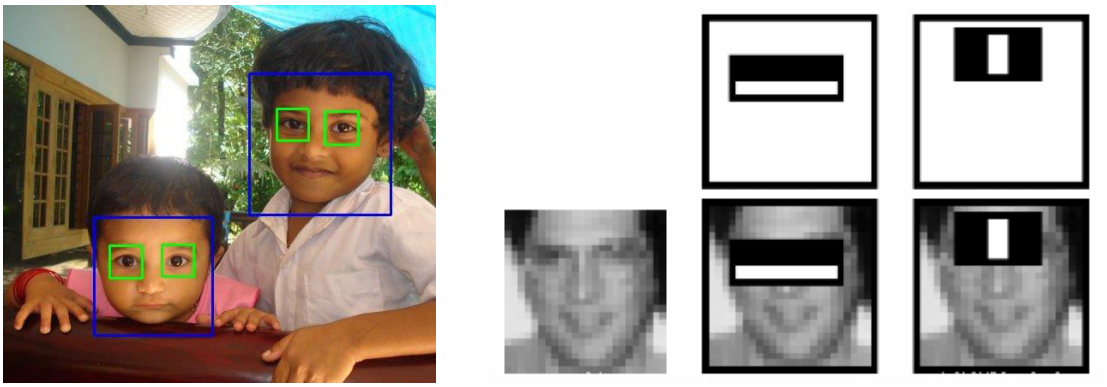


Figure 72 – OpenCV Face Detection Example

10.1.2. Dlib C++ Library

The Dlib [26] library is a C++ and Python library offering a variety of C++ libraries for multiple purposes, one of them is the image processing and the face detection. The Dlib library offers two different choices of face detection, the single face detection and the face detection with landmark localization.

The single face detection system offer frontal face detection only (-45^{o} to +45^{o}) using the object detection method of [2]. It does not offer though pose estimation. The extra landmark localization library is used after the face detection procedure using the data returned by the face detection task and uses the [15] only in the area of the image where the face is detected. This is a very good method for fast landmark localization. The difference with the [1] system is that the last one offers pose estimation and face detection on a greater range of viewing angles (-90^{o} to +90^{o}).

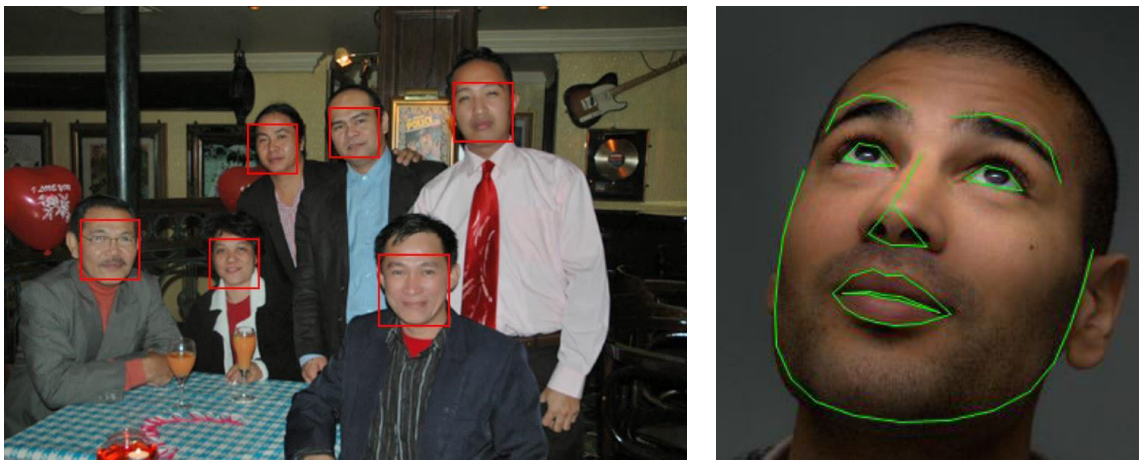


Figure 73 – Dlib Face Detection and Landmark Localization Example

10.1.3. Face SDK

The Face SDK [30] library is a library for face detection, recognition and verification. This library is not referred to any known face detection algorithm and it only supports face detection. It detects the eyes, nose and mouth centers and by them it results to a face detection. No pose estimation and landmark localization is supported. The method used for the face detection makes it obvious that probable can support only frontal face detection.

Figure 74 – Face SDK Face Detection Example

10.1.4. Flandmark

This Flandmark [31] library uses the OpenCV library for face detection and by the returned data searches the area of the detected faces for seven critical landmarks. These landmarks are the edges of the eyes, the tip of the nose and the edges of the mouth. No other landmark localization is supported and also does not support pose estimation. The fact that the OpenCV library is used for face detection concludes that only frontal face detection is supported.

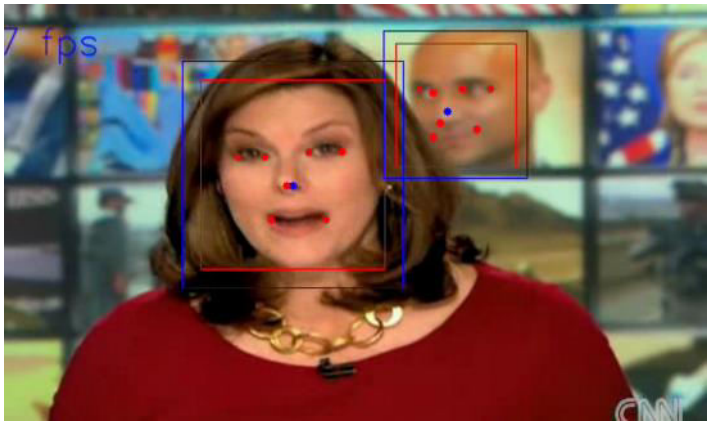


Figure 75 – Flandmark Face Detection Example

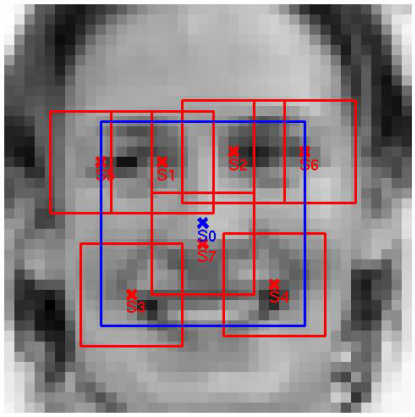


Figure 76 – Flandmark Landmarks Localization

10.1.5. Semantic Vision Technologies

The Semantic Visions technologies [32] library is based on the [5]. This system is based on the detection of 15 critical landmarks that result to a face detection. Although it provides face parts localization like the eyes, the nose and the mouth, no more landmark localization is offered.

Besides it does not support pose estimation and the face detection is seems to be limited in frontal faces only.

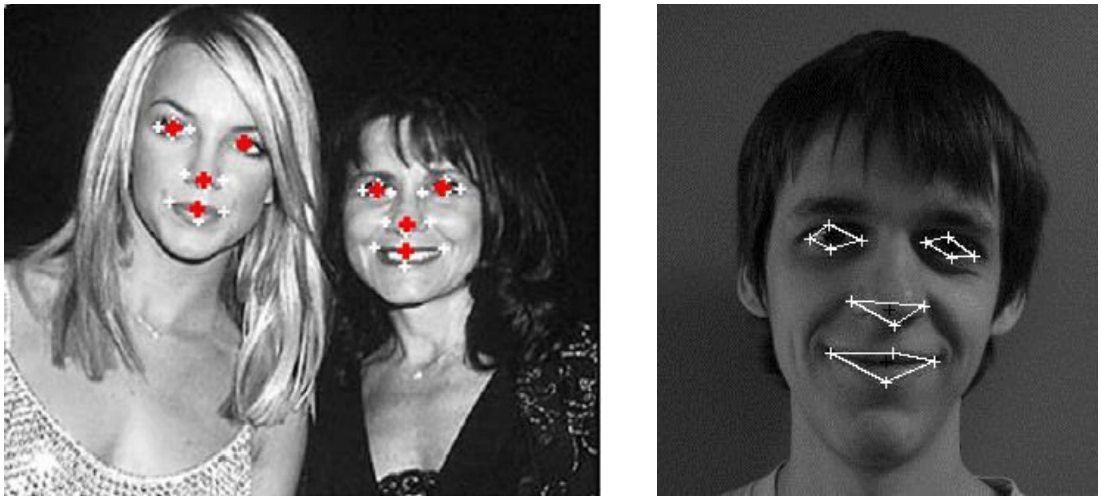


Figure 77 – Semantic Vision Technologies Face Detection and Landmark Localization Example

10.1.6. FDLib

The FDLib library [29] is based on the [16] algorithm publication. This system supports the face detection task without any of the pose estimation and landmark localization ones. It was published in 2005 and its one of the oldest face detection algorithms using neural networks for doing that. It is an old dated system and it would be unfair to be compared with modern systems.

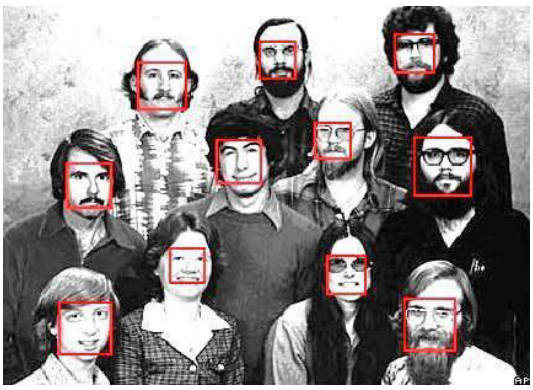


Figure 78 – FDLib Face Detection Example

10.2. Latest Systems

Except of the freeware libraries there are also other face detection systems as the ones referred in chapter 3. These systems does not offer their implementation freely for usage. In this subchapter a short reference on each one is made and their differences against the [1] are appose.

10.2.1. Face Detection and Pose Estimation Based on Evaluating Facial Feature Selection

This system [3] offers a face detection method based on Haar-like features as exactly Viola and Jones [17] algorithm does. What is different is that its features are more efficient. It firstly detects the face parts (eyes, nose and mouth) and this concludes to the face detection. The

authors claim that this algorithm performs better than the [1]. Despite that this systems does not offer pose estimation and landmark localization and is also efficient only on frontal face detection.

Figure 79 – Publication [3] Face Detection Example

10.2.2. Head Pose Estimation Based On Detecting Facial Features

This systems [4] is a sequel of [3]. It does not claim to succeed as good results as the [3] but it also offers the task of pose estimation. It uses the same Haar-like features as the [3] and it combines the face parts (eyes, nose and mouth) detected by the cascade windows in order to estimate the pose of the face. What it does not support is the 68 face landmark localization and also it is efficient in frontal face detection.

10.2.3. Discrete area filters in accurate detection of faces and facial features

This system [5] is offered as a freeware library to be used by anyone and it is described in chapter 10.1.5.

10.2.4. Real-time High Performance Deformable Model for Face Detection in the Wild

The creators of this system [6] claim to have design a much faster and efficient algorithm compared to [1]. As far as the face detection speed they claim to reach real-time performance. The differences are the fact that this systems does not localize all the 68 landmarks of the human face but only some of them in order to complete the face detection task. In addition it is only able to classify the pose estimation in 9 classes of viewing angle when [1] uses 13. Its main advantage is the fact that it is very fast and efficient as the authors claim but it still lacks on the pose estimation accuracy and the full 68 landmark localization.

Figure 80 – Publication [6] Face Detection Example

10.2.5. Multi-view Face Detection Using Deep Convolutional Neural Networks

This system [7] is one of the latest state-of-the-art ones using convolutional neural networks that are considered to be the best method for face detection right now. This system succeeds better detection performance compared to [1], detecting faces in the full viewing angle (-90^{o} to +90^{o}). What is missing from this system is the pose estimation and the landmark localization that the [1] system supports.

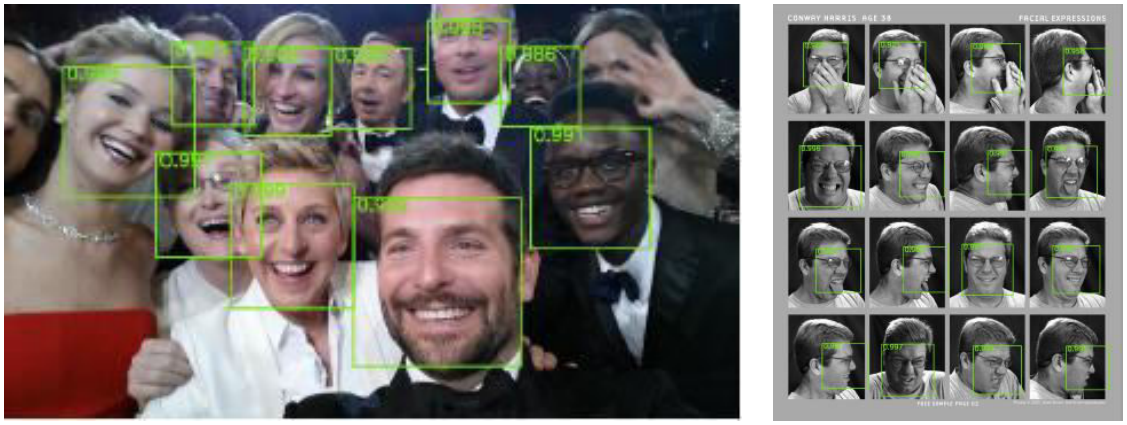


Figure 81 – Publication [7] Face Detection Example

10.2.6. Face and Landmark Detection by Using Cascade of Classifiers

This systems [8] is using face parts detection (eyes and mouth) in order to result to a complete face detection. The authors claim to succeed better results than the [1]. This systems though

does not support pose estimation neither landmark localization. It only supports eyes and mouth localization and it works better for frontal face detection.

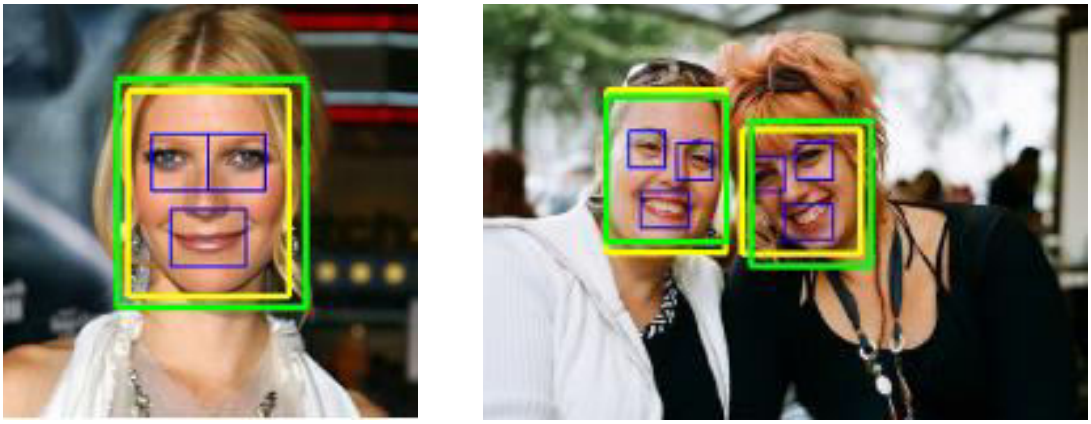


Figure 82 – Publication [8] Face Detection Example

10.2.7. Extensive Facial Landmark Localization with Coarse-to-fine Convolutional Network Cascade

This system [9] is the most related to the [1]. It is using the same method for face detection but what is different is the fact that it splits the face detection task from the landmark localization one. It firstly uses the jaws landmarks for face detection and then it localizes the rest landmarks of the human face. This method provides it a better landmark localization than [1] and the whole process is faster as the most landmarks detection is applied in the face detected area. On the other hand this system only supports frontal face detection and does not supports pose estimation. This last task could be easily implemented as long as the landmark localization task exists.

Figure 83 – Publication [9] Face Detection Example

10.2.8. Face detection by structural models

This system [10] is also very similar to [1]. It uses the method, locating landmarks for the face detection process. In contrast to [1] it uses less landmarks than the global 68 human face ones that means that the process should sensibly be faster. The authors claim to succeed better detection performance than the [1] but their system does not support pose estimation. It is also efficient only on frontal face detection.

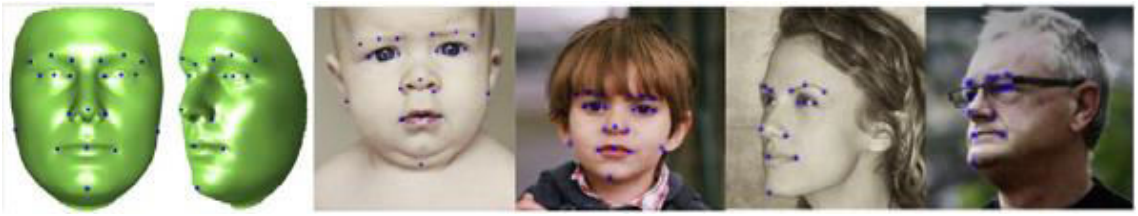


Figure 84 – Publication [10] Face Detection Example

11. Future Work

There are to areas where this thesis system can be extended in the future. The first area is the one of face detection, pose estimation and landmark localization and the second one is the area of object detection.

As far as the face detection procedure TSM system can be separated in two sections. The first section could be the face detection one while the second the pose estimation and landmark localization one. In the face detection section the usage of less landmarks can be applied as the most systems ([5], [6], [7], [8] and [10]) after [1] do and as exactly the [9] does. This way the face detection procedure would be a much faster procedure. By the time the face detection procedure is completed then the pose estimation and landmark localization ones can be applied in the detected face area within the image (like [9]). By doing so the size of data have to be processed would be much less than in the whole image as the [1] does. At this section a validation of the face detection result can also be applied increasing the algorithms reliability. This execution flow would reduce in a large scale the execution time needed for the algorithm to complete the whole procedure.

As far as the pose estimation task the system can be extended in a way that not only the yaw angle to be estimated but also the roll and the pitch angles. This task can be achieved not only by using shape models but also with relative models between the main critical face landmarks as [4] does.

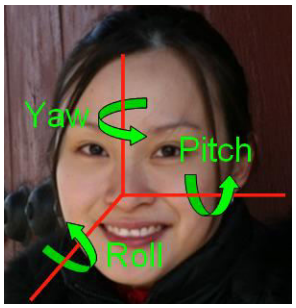


Figure 85 – Complete Pose Estimation

As far as the object detection area, this system can be easily transformed to an object detection system using tree structural models. As this thesis system implements the [1] that is based on the object detection system [2], this system can also be used as an object detection one. The only changes have to be done is to change the way the TSM system handles its memory consumption in order to hold the Filters Responses arrays when multi-scale TSM is used.

There are also many other ways that can extend this TSM system that are left in the readers creativity!

12. Annex A – TSM Execution Times

In this chapter the exact execution time of the TSM v3.2.2 system in seconds is presented in order the ability of using it in applications. As presented in Table 175 the TSM algorithm can complete the detection procedure in less than one second in the majority of the non-embedded systems for small sized images (320x240). For larger images the system need more than one second of time.

Table 175 – TSM v3.2.2 Execution Time in Seconds				
CPU Model	Intel Xeon E5430 @2.66GHz (x2)	Intel Core i7 4600U @2.70GHz	Intel Core 2 Duo T8100 @2.10GHz	Dual Core ARM Cortex-A9 @866MHz
CPU Cores	8	4	2	2
RAM Memory	12 GB	8 GB	4 GB	512 MB
Operating System	Ubuntu Server (no GUI)	VM Ubuntu 15.01 (no GUI)	Ubuntu 14.04 (no GUI)	Ubuntu 12.04 (no GUI)
Image Size	**Execution Time (sec)**			
320x240	0.287	0.590	0.747	7.325
640x480	1.282	2.699	5.379	52.84
800x600	2.028	3.957	8.525	83.65
1024x768	3.421	6.528	14.25	143.0
1280x960	5.390	10.20	22.56	220.4

At the next table (Table 176) the exact execution time of the TSM algorithm in seconds is presented using the hardware system of the 2nd column of Table 175. This is a virtual machine so that the number of CPU cores can be customized and the execution time needed for the algorithm can be tested using different number of CPU cores every time. The data of Table 175 and Table 176 are the ones that create the statistical data of the Table 174.

Table 176 - TSM v3.2.2 and Creators Execution Time (sec)						
CPU Cores	**TSM**	**320x240**	**640x480**	**800x600**	**1024x768**	**1280x960**
1	Original	2.20	7.96	12.0	19.9	29.7
	v3.2.2	1.08	4.89	8.10	13.6	21.0
2	Original	1.85	6.06	8.93	14.7	21.9
	v3.2.2	0.71	3.27	5.17	8.60	13.2
3	Original	1.72	5.88	8.96	14.2	21.8
	v3.2.2	0.66	2.95	4.45	7.46	11.4
4	Original	1.70	5.76	8.89	14.2	21.9
	v3.2.2	0.59	2.70	3.96	6.53	10.2

13. Bibliography

[1] X. Zhu, D. Ramanan. "Face detection, pose estimation and landmark localization in the wild" Computer Vision and Pattern Recognition (CVPR) Providence, June 2012.

[2] Pedro F. Felzenszwalb, Ross B. Girshick, David McAllester and Deva Ramanan, “Object Detection with Discriminatively Trained Part Based Models”, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 32, No. 9, Sep. 2010

[3] Hiyam Hatem, Zou Beiji, Raed Majeed, Mohammed Lutf and Jumana Waleed, “Face Detection and Pose Estimation Based on Evaluating Facial Feature Selection” International Journal of Hybrid Information Technology, Vol. 8, No. 2, 2015

[4] Hiyam Hatem, Zou Beiji, Raed Majeed, Jumana Waleed and Mohammed Lutf “Head Pose Estimation Based On Detecting Facial Features” International Journal of Multimedia and Ubiquitous Engineering, Vol. 10, No. 3, 2015

[5] Jacek Naruniec, “Discrete area filters in accurate detection of faces and facial features” Image and Vision Computing 32, 2014

[6] Junjie Yan, Xucong Zhang, Zhen Lei and Stan Z. Li “Real-time High Performance Deformable Model for Face Detection in the Wild” Chinese Academy of Sciences

[7] Sachin Sudhakar Farfade, Mohammad Saberian and Li-Jia Li “Multi-view Face Detection Using Deep Convolutional Neural Networks”, Yahoo 2015

[8] Hakan Cevikalp, Bill Triggs and Vojtech Franc “Face and Landmark Detection by Using Cascade of Classifiers”, Automatic Face and Gesture Recognition (FG), 2013

[9] Erjin Zhou, Haoqiang Fan, Zhimin Cao, Yuning Jiang and Qi Yin “Extensive Facial Landmark Localization with Coarse-to-fine Convolutional Network Cascade” International Conference of Computer Visio (ICCV), 2013

[10] Junjie Yan, Xuzong Zhang, Zhen Lei and Stan Z. Li “Face detection by structural models”, Image and Vision Computing 32, 2014

[11] E. H. Adelson, C. H. Anderson, J. R. Bergen, P. J. Burt and J. M. Ogden, “Pyramid methods in image processing”, RCA Engineer, 1984

[12] Navneet Dalal and Bill Triggs, “Histograms of Oriented Gradients for Human Detection”, Computer Vision and Pattern Recognition, (CVPR) 2005

[13] P. Felzenszwalb and D. Huttenlocher, “Distance Transforms of Sampled Functions”, Theory of Computing, Vol. 8, No. 19, September 2012

[14] Rasmus Rothe , Matthieu Guillaumin , and Luc Van Gool, “Non-Maximum Suppression for Object Detection by Passing Messages between Windows”, Pattern Recognition, (ICPR) 2006

[15] Vahid Kazemi and Josephine Sullivan, “One Millisecond Face Alignment with an Ensemble of Regression Trees”, (CVPR) 2014

[16] W Kienzle, G BakIr, M Franz, and B Schölkopf, “Face Detection: Efficient and Rank Deficient”, July 2005

[17] Paul Viola and Michael Jones, “Rapid Object Detection using a Boosted Cascade of Simple Features”, (CVPR) 2001

14. Web Sources

[18] http://www.ece.tuc.gr/4516.html (Technical University of Crete, Electronics Laboratory)

[19] http://www.ece.tuc.gr/4515.html (Technical University of Crete, Electronic Circuits and Renewable Energy Sources Laborartoty)

[20] http://www.ece.tuc.gr/4514.html (Technical University of Crete, Microprocessors and Hardware Laboratory)

[21] http://www.ece.tuc.gr/4512.html (Technical University of Crete, Intelligence Systems Laboratory)

[22] http://cordis.europa.eu/project/rcn/97141_en.html (SAFEMETAL)

[23] http://www.tsi.gr/?page_id=498&lang=en (EXEHON)

[24] http://luthuli.cs.uiuc.edu/~daf/book/book.html (Computer Vision: A Modern Approach)

[25] https://people.cs.umass.edu/~hsu/ (Hang Su TSM Algorithm Implementation)

[26] http://blog.dlib.net/2014/02/dlib-186-released-make-your-own-object.html (Dlib)

[27] http://opencv.org/ (OpenCV)

[28] http://chenlab.ece.cornell.edu/projects/FaceTracking/ (Advanced Multimedia Processing Lab)

[29] http://people.kyb.tuebingen.mpg.de/kienzle/facedemo/facedemo.htm (fdlib)

[30] http://facesdk.eu/main_en (FaceSDK)

[31] http://cmp.felk.cvut.cz/~uricamic/flandmark/ (Flandmark)

[32] http://www.semanticvisiontech.com/ (Semantic Vision Technologies)

[33] http://rapidxml.sourceforge.net/ (RapidXML)

[34] http://openmp.org/ (OMP)