Optimization of Heuristic Search using Recursive Algorithm Selection and Reinforcement Learning

Vasileios C. Vasilikos

Department of Electronic and Computer Engineering

Technical University of Crete

A thesis submitted in partial fulfillment for the Diploma degree of

Electronic and Computer Engineering

Chania, July 2009

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

Βελτιστοποίηση Ευριστικής Αναζήτησης μέσω Αναδρομικής Επιλογής Αλγορίθμου και Ενισχυτικής Μάθησης

Βασίλειος Χ. Βασιλικός

Τμήμα Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών

Χανιά, Ιούλιος 2009

This work is dedicated to my parents, who made my choices come true by supporting at me every step of the way.

Acknowledgements

This work took almost an entire year to complete, and a lot of things happened in my life during that time, things that made working on this thesis harder than I expected. I would like to take some time here and thank the people who made completion of this work possible despite of all that happened.

I'd like to thank all my colleagues at TU Crete, who made these years in Chania what is possibly the best years in my life so far. Their help during my studies was invaluable to me, and I will always be grateful for it.

I would also like to thank my Greek colleagues at TU Delft for their understanding and support during my first year there, who made working on my thesis at the same time as my Master's degree a lot easier.

Of course, it goes without saying that tremendous help was provided by my supervisor, professor Lagoudakis. From the undergraduate courses all the way to the diploma thesis, he was always open to new ideas, providing his knowledge and experience to make what we had in mind possible.

Most importantly, I would like to thank my family, my parents and sister, for their help, support, guidance and love that made my life and my choices clear, easier, and most importantly: possible.

Finally, I would like to thank the person that made these past years truly the best years of my life; my girlfriend Valia. Thank you for bearing with me during my good times and my bad times, I don't know if I could have made it without you!

Abstract

The traditional approach to computational problem solving is to use one of the available algorithms to obtain solutions for all given instances of a problem. However, typically not all instances are the same, nor a single algorithm performs best on all instances. This thesis investigates a more sophisticated approach to problem solving, called Recursive Algorithm Selection, whereby several algorithms for a problem (including some recursive ones) are available to an agent who makes an informed decision on which algorithm to select for handling each sub-instance of a problem at each recursive call made while solving an instance. Reinforcement learning methods are used for learning decision policies that optimize any given performance criterion (time, memory, or a combination thereof) from actual execution and profiling experience. This thesis focuses on the well-known problem of state-space heuristic search and combines the A* and RBFS algorithms to yield a hybrid search algorithm, whose decision policy is learned using the Least-Squares Policy Iteration (LSPI) algorithm. Our benchmark problem domain involves shortest path finding problems in a real-world dataset encoding the entire street network of the District of Columbia (DC), USA. The derived hybrid algorithm exhibits better performance results than the individual algorithms in the majority of cases according to a variety of performance criteria balancing time and memory. It is noted that the proposed methodology is generic, can be applied to a variety of other problems, and requires no prior knowledge about the individual algorithms used or the properties of the underlying problem instances being solved.

Περίληψη

Ο παραδοσιακός τρόπος επίλυσης υπολογιστικών προβλημάτων είναι η χρήση ενός εκ των διαθέσιμων αλγορίθμων προκειμένου να βρεθούν λύσεις για όλα τα δεδομένα στιγμιότυπα κάποιου προβλήματος. Ωστόσο, συνήθως δεν είναι όλα τα στιγμιότυπα ίδια, ούτε ένας συγκεκριμένος αλγόριθμος αποδίδει βέλτιστα για όλα τα στιγμιότυπα. Αυτή η διπλωματική εργασία ερευνά μια πιο εκλεπτισμένη προσέγγιση του προβλήματος, γνωστή ως Αναδρομική Επιλογή Αλγορίθμων (Recursive Algorithm Selection), όπου διάφοροι αλγόριθμοι για κάποιο πρόβλημα (κάποιοι εξ αυτών αναδρομικοί) είναι διαθέσιμοι σε έναν πράκτορα ο οποίος αποφασίζει ποιον αλγόριθμο θα επιλέξει για να χειριστεί το κάθε υποστιγμιότυπο ενός προβλήματος σε κάθε αναδρομική κλήση που γίνεται κατά την επίλυση ενός στιγμιοτύπου. Για τη μάθηση πολιτικών λήψης αποφάσεων που βελτιστοποιούν οποιοδήποτε δεδομένο κριτήριο απόδοσης (χρόνος, μνήμη, ή συνδυασμός τους) χρησιμοποιούνται μέθοδοι ενισχυτικής μάθησης (Reinforcement Learning)που κάνουν χρήση εμπειρικής πληροφορίας από πραγματικές εκτελέσεις και μετρήσεις. Η συγκεκριμένη εργασία εστιάζει στο γνωστό πρόβλημα της ευριστικής αναζήτησης σε χώρο καταστάσεων (state-space search)και συνδυάζει τους αλγορίθμους A* και RBFS για να παράγει έναν υβριδικό αλγόριθμο αναζήτησης, όπου η πολιτική λήψης αποφάσεων μαθαίνεται χρησιμοποιώντας τον αλγόριθμο Least-Squares Policy Iteration (LSPI). Το πεδίο προβλημάτων στο οποίο δουλέψαμε περιλαμβάνει προβλήματα εύρεσης συντομότερου μονοπατιού σε μια βάση δεδομένων με πραγματικά στοιχεία που κωδικοποιεί όλο το οδικό δίκτυο της Πολιτείας District of Columbia (DC) , ΗΠΑ. Ο τελικός υβριδικός αλγόριθμος παρουσιάζει καλύτερη απόδοση σε σχέση με τους μεμονωμένους αλγορίθμους στην πλειονότητα των περιπτώσεων σύμφωνα με μια πληθώρα κριτηρίων απόδοσης που σταθμίζουν χρόνο και μνήμη. Σημειώνεται ότι η προτεινόμενη μεθοδολογία είναι γενική, μπορεί να εφαρμοστεί σε πλήθος προβλημάτων, και δεν απαιτείται πρότερη γνώση σχετική με τους αρχικούς αλγορίθμους που χρησιμοποιούνται ή τις ιδιότητες των στιγμιοτύπων του προβλήματος που επιλύονται.

Contents

List of Figures

Chapter 1

Introduction

Problem solving is a central and well defined concept of Computer Science in general. After formulating a computational problem, an algorithm is used to solve the problem. Algorithms are formal and methodic approaches to a certain problem that returns a solution within finite (and reasonable) time and space. A problem may be (and is often) approached by more than one algorithms. Much research has been invested into analyzing the properties of algorithms and their efficiency in problem solving, in order to be able to understand the impact of choosing a particular algorithm.

1.1 Problem Solving today

So far the field of problem solving has been dominated by single algorithm approaches, leaving the choice of the appropriate algorithm to the developer. It is the developer's responsibility to study the problem, analyze requirements, comply with platform constraints and choose the algorithm that best fits these constraints while fulfilling the necessary requirements. Unfortunately, the choices available always come with trade-offs. Each choice allows for advantages in some areas, while having inherent disadvantages in others. These trade-offs, at the algorithmic level, can be execution speed versus memory consumption, parallelization versus synchronization overhead, solution accuracy versus overall complexity, etc. Making the right choice is what an engineer is expected to do, after meticulous analysis of the problem, the choices, and the available resources. Since the problem itself and the resources available are usually out of the engineer's control, the choices available are critical to their job and the final outcome of their work.

While there has been significant research in the development of new algorithms that try to compensate for the disadvantages of existing algorithms as technology evolves, research has shifted focus to implementation optimizations as well as hardware optimizations, that allow for further compensation of algorithm-inherent disadvantages. It is often the case that despite the variety of available choices, improvements in hardware as well as in the process of bringing an algorithm from the theoretical level to the actual implementation that will be used, the trade-offs that are inherent to the algorithm that was chosen do not fulfill the intentions in a satisfactory way.

1.2 A real world example

To illustrate this point, consider a possible real-world scenario, where a developer is called upon to choose an algorithm to use in a GPS unit that will display a path from point A to point B. GPS units do not have the memory capacity or computational power of a modern PC, but they need to produce results fast and accurately, given the circumstances in which they are used. The lead developer would have to choose the algorithm that is best for fast results, possibly increasing memory requirements, which would lead to either a need for more memory in the design of the device or perhaps the reduction of memory consumption on other parts of the application, possibly at the expense of features that would otherwise improve the usability of the device. Extra cost would be induced to spend more work hours to optimize other levels of the development process due to lack of options at the algorithmic level.

There has been some research to aid the developer in choosing the algorithm that will end up solving the problem out of a given set of available algorithms. All these solutions however make use of a single algorithm solving the problem from beginning to end, which means that all inherent advantages and disadvantages as well will be carried on in the final implementation.

1.3 What this work is about

What we propose in this thesis is in the direction of dynamic algorithm selection, using different algorithms on different sub-instances of the original problem, in a way that ends up making use of the best aspects of each available algorithm, by applying to each sub-instance the most appropriate algorithm to yield the minimum possible cost. In other words, we interleave available algorithms during the problem solving process. Naturally, the way that this dynamic selection works has to be generic, and not specific to a particular problem or to a given set of algorithms, otherwise the re-usability value of this work would be minimal, if any.

We focused on Tree-Search algorithms, designing and implementing a framework, where the available algorithms can be re-defined, new algorithms added, and to some extend the entire problem class re-defined as well, without inducing extended extra work. The system is designed in a generic fashion, such that the developer can define the problem, add algorithms to the selection set, train the decision policy and eventually get a hybrid algorithm that will allow for improved performance given the cost definition that was provided. Once the system has been setup for a specific problem, it can be trained and produce different hybrid algorithms rapidly, in case the cost definition needs to change.

The way hybrid algorithms are created is by allowing for a choice during the recursive step of each algorithm. Instead of calling itself, a recursive algorithm calls a decision making function that has been trained, and is responsible for choosing the specific algorithm to use on the next recursive call. That way, at each recursive call we can have a different algorithm, and the solution will not be produced by any single algorithm alone, but by a the combination of the available algorithms.

One of the most important aspects of this dynamic selection process is of course the decision making procedure, as it is this part that eventually produces the specific hybrid algorithm, and it is this part that will make the difference between a hybrid algorithm and the individual ones. This is where Reinforcement Learning (RL) comes in, to train this decision making procedure so that the choices it makes eventually produce favorable results. We use the Least-Squares Policy Iteration (LSPI) algorithm [1], which is an off-line Reinforcement Learning algorithm, meaning it works independently of the problem-solving itself. The input for this algorithm is a set of samples that represent different choices and their observed impact. The LSPI algorithm processes this input and returns a policy for making decisions based on the definition of cost/gain that the developer gives (penalty/reward function).

1.4 How it is done

Our implementation used two algorithms for Tree-Search: A* and RBFS. They are both informed-search algorithms, meaning that they both need a heuristic function. The choice of these two specific algorithms illustrates our original point; A* is an algorithm that performs very well when speed is the main concern, but has a great memory consumption, and RBFS is an algorithm that performs very well when memory is important, but is significantly slower. As described above, developers are rarely faced with a trade-off choice that one of the opposing factors can be neglected. In our case, there will rarely be a Tree-Search problem case where speed is irrelevant and only memory matters, or vice versa. Usually both are a concern, so even though there are choices available that perform very well for speed or memory, there is no choice available that performs well for both speed and memory.

We attempted to achieve a hybrid version of these two algorithms using the method described above, and measured the performance of the hybrid algorithm produced against the best of the two original algorithms, for different definitions of cost based on memory and speed.

The algorithms were applied to a shortest-path finding domain. In particular, we consider several instances of the problem involving different paths, encoding a map of the complete road network of Washington, D.C., in a weighted undirected graph format. Each node represents an actual point on the map, with real latitude (φ) and longitude (λ) coordinates from TIGER/Line data files provided to the public by the US government [2]. A simple Straight-Line-Distance heuristic was used for both algorithms. The Straight-Line-Distance heuristic calculates distance in meters from φ/λ coordinates based on approximations that compensate for earth's oblate spheroid shape.

An initial run was made on this set of instances with random algorithm choices at each recursive step to collect training samples. This sample set was fed as the input to LSPI, which in turn produced a decision policy, effectively creating the desired hybrid algorithm. Different hybrid algorithms were created for different definitions of the penalty function, all using the same sample set. This means that only one training run was performed. Subsequently, the hybrid algorithms were benchmarked and compared against the original algorithms according to the definition of cost used to create each hybrid algorithm, both on the original training instances, but also on new unknown instances.

1.5 Overall picture

Overall, the results are very encouraging, as the hybrid versions, on average, out-performed the original algorithms, with varying amounts of gain depending on the definition of cost. This will hopefully provide a springboard for further research in this field.

1.6 Thesis outline

This thesis is organized as follows: In Chapter 2 we will provide the necessary theoretical background, which includes Tree-Search, heuristics, the algorithms used (A* and RBFS), Reinforcement Learning and LSPI in particular. In Chapter 3 we provide a more detailed problem statement and an overview of the related work in the area. In Chapter 4 we describe our approach to Recursive Algorithm Selection in Tree-Search in detail. In Chapter 5 we present the most important aspects of the implementation of our approach. In Chapter 6 we provide the results of our work, and finally in Chapter 7 we discuss the significance of our results, suggest future work directions and draw some conclusions.

Chapter 2

Background

2.1 Searching

2.1.1 The search tree

In the field of Artificial Intelligence, it is quite common for the solution of a problem to be a sequence of successive states, from some initial state to a goal state where the problem has been solved. In algorithmic terms, the procedure of finding such a sequence or path can be modeled by searching a tree, where each node represents a state, and its successor nodes represent the states to which an action on that node will lead to. Traversing this tree is the equivalent of taking actions in the world to which these states belong. Eventually what is sought is the sequence of actions that needs to be taken in order to reach a goal state from the initial state. A Tree-Search algorithm is an algorithm that traverses a tree using a certain strategy, looking for a goal state, and returns the sequence of actions that led to that state when it finds it.

Obviously, what discriminates problems from each other, other than the description of states themselves, is the set of specific rules that apply to the succession of states. This is what is known as the successor function or node expansion function. When we expand a node, we are applying the rule set that describes our problem to a specific state that is represented by that node. The effect is a set of new nodes that represent different states that would be the result of all the possible actions that can be taken in the current node/state.

The set of nodes that have been generated but have not yet been expanded is what is called the fringe. As such, each node in the fringe is a leaf node of the traversed part of the tree. The afore-mentioned strategy that a Tree-Search algorithm uses is the strategy that picks a node from the fringe to expand. This is called the search strategy.

2.1.2 Tree-Search algorithms

Figure 2.1 shows the general Tree-Search algorithm. In general, a search algorithm is characterized by the following features:

- Completeness: Is the algorithm guaranteed to find a solution if there is one?
- Optimality: Is the algorithm guaranteed to find an optimal solution?
- Time complexity: How much time does it take to find a solution?
- Memory complexity: How much memory is needed to find a solution?

When talking about Tree-Search algorithms in particular, complexity, time and memory depend on the tree's branching factor b, the depth of the shallowest goal node d, and the maximum length of any path in the state space m.

2.1.3 Heuristics

There are two main types of search algorithms: Uninformed or blind search algorithms and informed or heuristic search algorithms [3]. The difference between them is the absence or presence respectively of additional information about the problem other than the problem definition itself. A blind search algorithm can only generate states and distinguish goal states from non-goal states. A heuristic search can use the additional information that it has to distinguish between states that are more likely to lead to a goal state than others. In this work we focused on heuristic search, so blind search will not be discussed further.

Heuristic search makes use of an evaluation function $f(n)$, where n is a node. This function returns a value for that node, and based on that value and the search strategy, the algorithm decides which node it will expand next. The general approach is expanding the node with the best evaluation. It is important to stress the fact that this is an evaluation and not actual measurement of exactly how

```
GeneralTreeSearch.
Input: Problem, fringe. Returns: Solution/Failure
  $fringe \leftarrow insert(makeNode(initialState[problem]), fringe)$
  **loop**
    **if** $fringe.empty = true$ **then**
      **return** $failure$
    **end if**
    $node \leftarrow selectNode(fringe)$ //Search Strategy
    **if** $node.isGoal = true$ **then**
      **return** $success(node)$
    **end if**
    $fringe \leftarrow insertAll(expand(node, problem), fringe)$
  **end loop**
```

Figure 2.1: The general Tree-Search algorithm

much value a certain node holds; if the exact value was known, there would not be a need for searching and the path to the solution would simply be following the best node each time. But, since only evaluation is possible, there is no guarantee that when a node receives a favorable result from evaluation that this node will necessarily be a good choice.

Part of the evaluation function is the heuristic function. It is this function that conveys most the additional (external) information that is available to the algorithm. The search strategy dictates how the algorithm will make use of that additional information when making a decision. A heuristic function $h(n)$ works in the same way the evaluation function works. It takes as input a state or node and returns a value for that node. That value is an informed guess on the actual value of that node. For example, when searching for shortest paths, a hypothetical heuristic might return a value that is an informed guess on the actual distance between the input node and the goal. The quality of the heuristic is just how accurate this informed guess is; the more accurate a heuristic function is, the better the algorithm will perform, since it will lead the search mostly towards the goal rather than away from it.

Heuristics are categorized depending on their behavior. An admissible heuristic is a heuristic that never overestimates cost (or underestimates gain). This means that by definition, admissible heuristics are always optimistic. If a heuristic's return value consistently increases or decreases for successive states as input, then this heuristic is called consistent. Consistent heuristics are always admissible, however not all admissible heuristics are consistent [3].

As mentioned in the introductory part of this thesis, the two algorithms that we used were A* and RBFS. They are both heuristic, complete, optimal search algorithms. We will describe both algorithms in this section in order to demonstrate the point that was made earlier, namely the fact that A* has a high memory complexity compared to RBFS, but much better time complexity, and vice versa.

2.1.4 A*

The most widely known form of best-first search is called A* search [4]. It evaluates nodes by combining $g(n)$, the actual cost to reach the node, and $h(n)$, the estimated cost to get from the node to the goal:

$$f(n) = g(n) + h(n)$$

Since $g(n)$ gives the path cost from the start node to node n, and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have:

$$f(n) = \text{estimated cost of the cheapest solution through node } n$$

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of $g(n) + h(n)$. It turns out this strategy is more than just reasonable: provided that the heuristic $h(n)$ is admissible, A* is both complete and optimal. Since $g(n)$ is the exact cost to reach n and $h(n)$ is admissible, $f(n)$ will never overestimate the true cost of a solution that goes through n.

It is worth noting that A* is also optimally efficient. This means that it expands no more nodes than the exact amount of nodes it needs in order to ensure optimality. Any algorithm that expands less nodes runs the risk of missing the optimal solution. Any algorithm that expands more nodes, is producing excess sub-trees that are irrelevant to the search.

That A* search is complete, optimal, and optimally efficient among all such algorithms is rather satisfying. Unfortunately, it does not mean that A* is the answer to all our searching needs. The catch is that, for most problems, the number of nodes within the goal contour search space is still exponential in the length of the solution.

Because it keeps all generated nodes in memory, A* usually runs out of space long before it runs out of time. For this reason, A* is not practical for many large-scale problems. Recently developed algorithms have overcome the space problem without sacrificing optimality or completeness, at the cost of execution time.

Figure 2.2 shows a basic implementation of the A* algorithm. The evaluate function used is the one described above.

```
A*. Input:node, fringe. Returns: Solution
  if $node.isGoal = true$ then
    return $success(node)$
  end if
  $fringe \leftarrow insertAll(expand(node), fringe)$
  $best \leftarrow \infty$
  for each $n$ in $fringe$ do
    if $evaluate(n) < best$ then
      $best \leftarrow n$
    end if
  end for
  A*$(best, fringe)$
```

Figure 2.2: The A* algorithm

The following sequence of figures shows an example of how A* would solve a problem on a hypothetical tree.

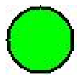


Figure 2.3: A* Example: Step 1

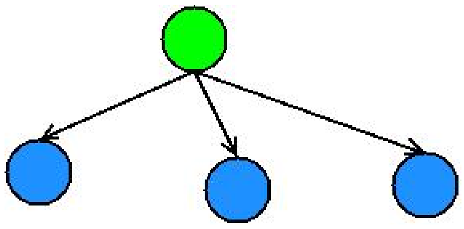


Figure 2.4: A* Example: Step 2

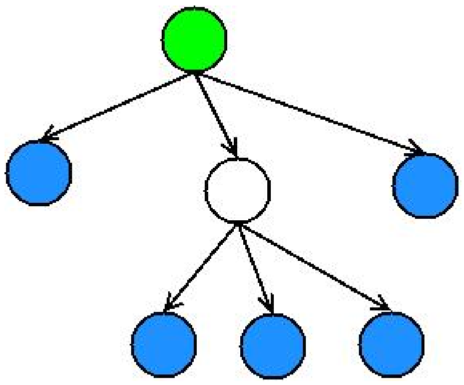


Figure 2.5: A* Example: Step 3

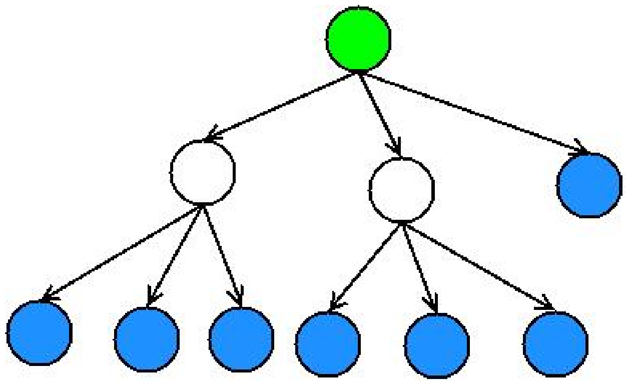


Figure 2.6: A* Example: Step 4

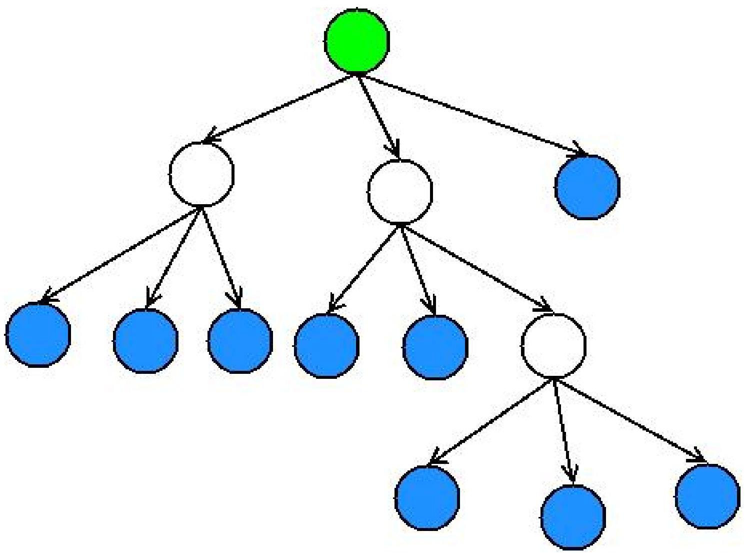


Figure 2.7: A* Example: Step 5

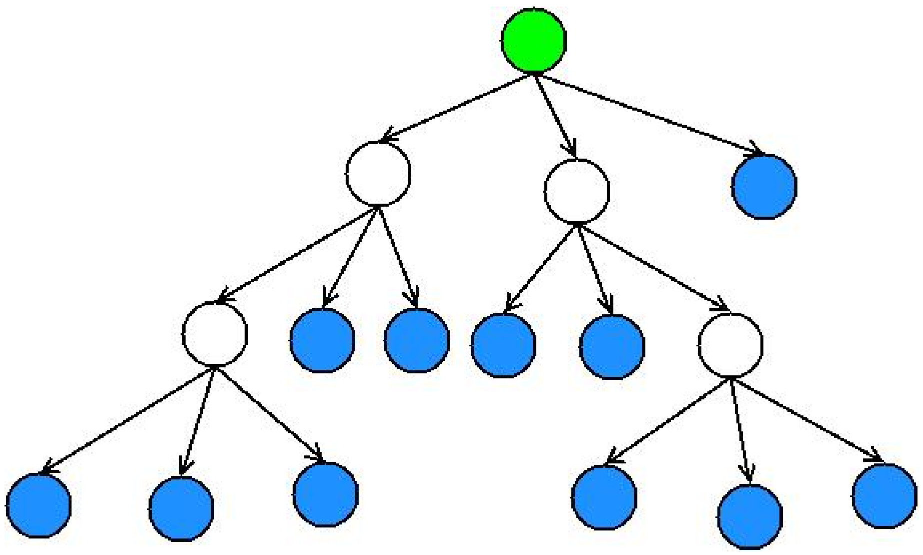


Figure 2.8: A* Example: Step 6

2.1.5 RBFS

Recursive Best First Search (RBFS) [5] is a simple recursive algorithm that attempts to mimic the behavior of A* but using only linear space. The algorithm is shown in Figure 2.10. Its structure is similar to that of a recursive depth-first search, but rather than continuing indefinitely down the path, it keeps track of

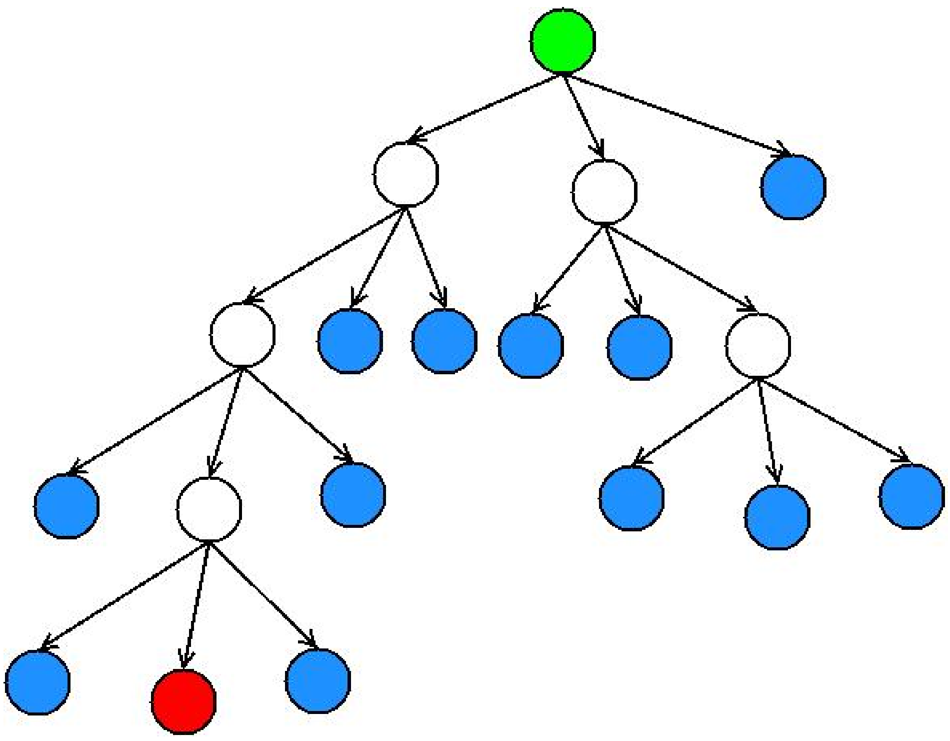


Figure 2.9: A* Example: Step 7

the f-value of the best alternative path from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the f-value of each node along the path with the best f-value of its children. In this way, RBFS remembers the f-value of the best leaf in the forgotten subtree and can therefore decide whether it's worth reexpanding the subtree at some later time.

It is this changing of opinion that can occur during RBFS searching that induces a great cost in execution time. Each time RBFS changes opinion about the optimal path, it needs to backtrack and collapse the current sub-tree it is looking at, and re-expand the alternative sub-tree.

Like A*, RBFS is an optimal algorithm if the heuristic function $h(n)$ is admissible. Its space complexity is linear in the depth of the deepest optimal solution, but its time complexity is rather difficult to characterize: it depends both on the accuracy of the heuristic function and on how often the best path changes as nodes are expanded. RBFS is subject to the potentially exponential increase

```
RBFS.
Input:node, fringe, fLimit.
Returns: Solution or Failure and new fLimit
  if $node.isGoal = true$ then
    return $success(node)$
  end if
  $successors \leftarrow expand(node, fringe)$
  for each $s$ in $successors$ do
    $f[s] \leftarrow max(g(s) + h(s), f[node])$
  end for
  loop
    $best \leftarrow minNode(successors, f)$
    if $best < fLimit$ then
      return $failure, f[best]$
    end if
    $alternative \leftarrow secondLowestNode(successors, f)$
    $result, f[best] \leftarrow RBFS(best, fringe, min(fLimit, alternative))$
    if $result! = failure$ then
      return $result$
    end if
  end loop
```

Figure 2.10: The RBFS algorithm

in complexity associated with searching on graphs, because it cannot check for repeated states other than those on the current path. Thus, it may explore the same state many times.

The following sequence of figures shows an example of how A* would solve a problem on a hypothetical tree.

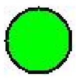


Figure 2.11: RBFS Example: Step 1

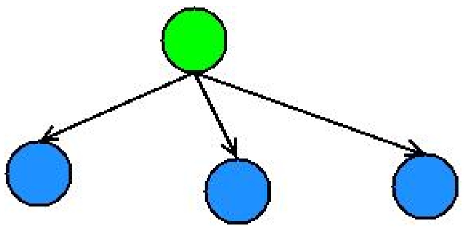


Figure 2.12: RBFS Example: Step 2

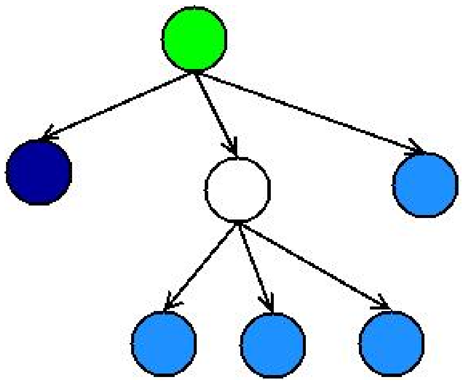


Figure 2.13: RBFS Example: Step 3

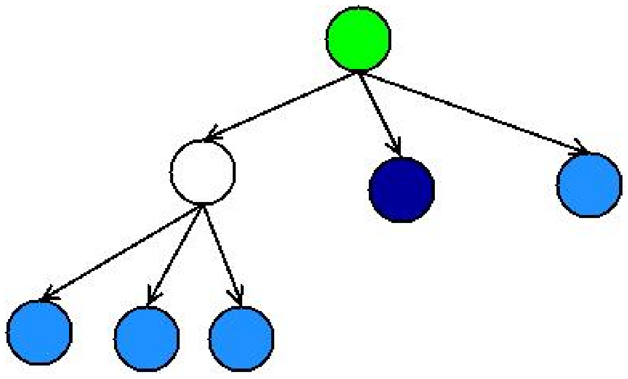


Figure 2.14: RBFS Example: Step 4

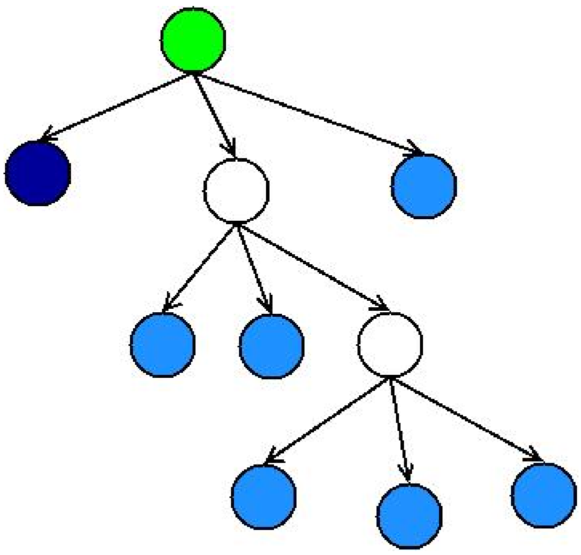


Figure 2.15: RBFS Example: Step 5

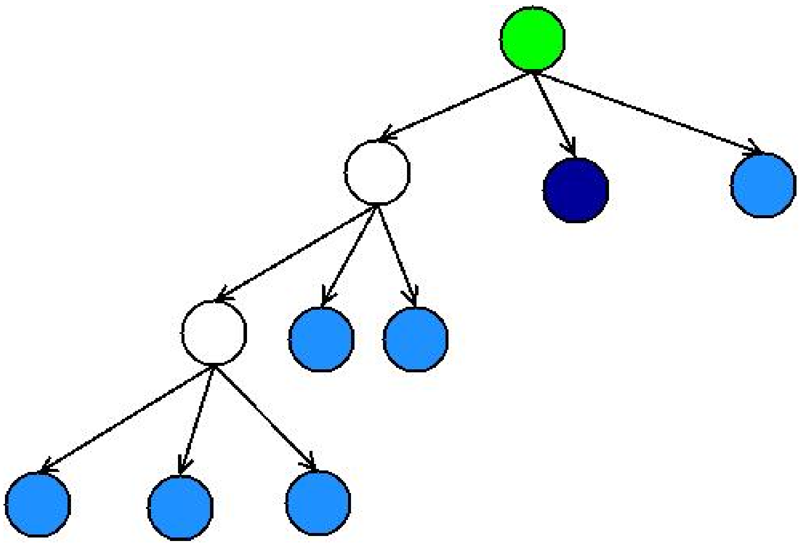


Figure 2.16: RBFS Example: Step 6

2.2 Learning

2.2.1 Markov Decision Process

A Markov Decision Process (MDP) is a discrete-time mathematical modeling framework for decision making, particularly useful when the outcome of a pro-

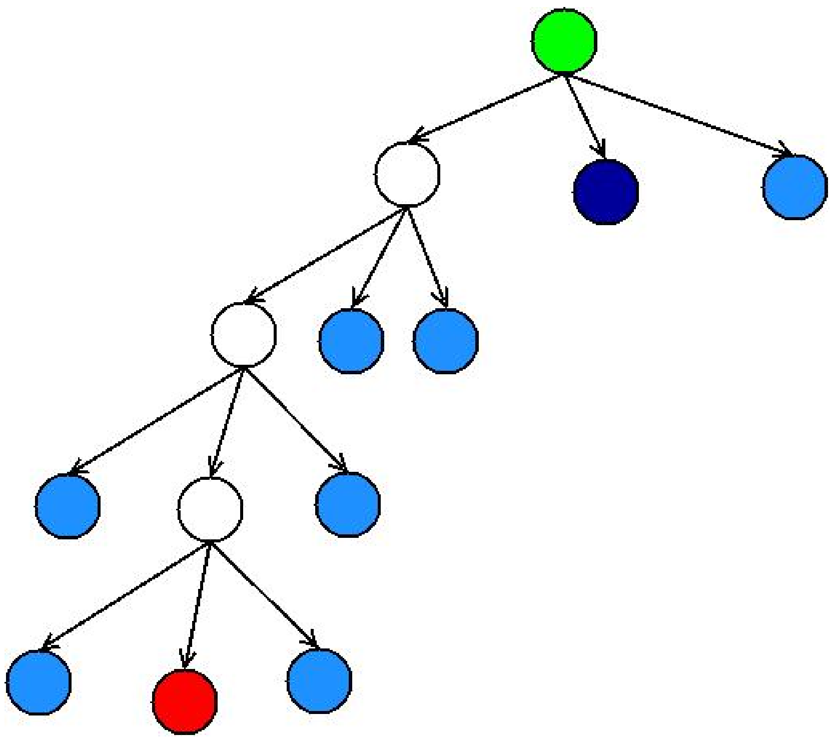


Figure 2.17: RBFS Example: Step 7

cess is in part a result of the agents actions and in part random. They have found extensive use in areas such as economics, control, manufacturing, and Reinforcement Learning.

An MDP can be described as a 4-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where:

- $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ is the (finite) state space of the process. The state s is a description of the status of the process at a given time.

- $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ is the (finite) action space of the process. The set of actions are the possible choices an agent has at a particular time.

- $\mathcal{P}$ is a Markovian transition model, where $\mathcal{P}(s, a, s')$ is the probability of making a transition to state s' when taking action a in state s. A Markovian transition model, means that the probability of making a transition to state s' when taking action a in state s depends only on s and a and not on the history of the process.

- $\mathcal{R}$ is the reward function (scalar real number) of the process. It is Markovian as well and can be the immediate or the expected immediate reward (for stochastic rewards) at each time step. The expected reward for a state-action pair (s, a), is defined as:

$$\mathcal{R}(s,a) = \sum_{s' \in S} \mathcal{P}(s,a,s')\mathcal{R}(s,a,s')$$

An MDP is often augmented to include γ and $\mathcal{D}$ as $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma, \mathcal{D})$, where:

- $\gamma \in (0,1]$ is the discount factor. When γ=1 a reward retains its full value independently of when it is received. As γ becomes smaller, the importance of rewards in the future is diminished exponentially by γ^t.

- $\mathcal{D}$ is the initial state distribution. It describes the probability that each state in $\mathcal{S}$ will be the initial state. On some problems most states have a zero probability, while few states (possibly only one) are candidates for being an initial state.

The optimization objective in an MDP is the maximization (or minimization depending on the problem) of the expected total discounted reward, which is defined as:

$$E_{s \sim D; a_t \sim ?; s_t \sim P} \left(\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s \right)$$

2.2.2 Policies

A policy π is a mapping from states to actions. It defines the responce (which may be deterministic or stochastic) of an agent in the environment for any state and it is sufficient to completely determine its behavior. In that sense,$\pi(s)$ is the action chosen by the agent following policy π.

An optimal policy π^* also known as an "undominated optimal policy" is a policy that yields the highest expected utility. That is, it maximizes the expected total discounted reward under all conditions (over the entire state space). For every MDP there is at least one such policy although it may not be unique (multiple policies can be undominated; hence yielding equal expected total discounted reward through different actions).

The state-action value function $\mathcal{Q}_\pi(s,a)$ for a policy π is defined over all possible combinations of states and actions and indicates the expected, discounted, total reward when taking action a in state s and following policy π thereafter:

$$\mathcal{Q}_\pi(s,a) = E_{a_t \sim \pi; s_t \sim P}\left(\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a\right)$$

2.2.3 Reinforcement Learning

Reinforcement Learning is learning in an environment by interaction [6, 7, 3]. It is usually assumed that the agent knows nothing about how the environment works (has no model of the underlying MDP) or what the results of its actions are. In addition, the environment can be stochastic, yielding different outcomes for the same situation. In contrast to supervised learning there is no teacher to provide examples of correct or bad behavior. It is very similar to unsupervised learning, except that one of the percepts is "hardwired" to be recognized as a reward.

The goal of an agent in such a setting is to learn from the consequences of its actions, in order to maximize the total reward over time. Rewards are in most cases discounted, in the sense that a reward early in time is "more valuable" than a reward later. This is done mainly in order to give an incentive to the agent to move quickly towards a solution, rather than wasting time in areas of the state space where there is no large negative reward.

Two related problems fall within Reinforcement Learning: Prediction and Control. In prediction problems, the goal is to learn to predict the total reward for a given policy, whereas in control the agent tries to maximize the total reward by finding a good policy. These two problems are often seen together, when a prediction algorithm evaluates a policy and a control algorithm subsequently tries to improve it.

The learning setting is what characterizes the problem as a Reinforcement Learning problem. Any method that can successfully reach a solution, is considered as a Reinforcement Learning method. This means that very diverse algorithms coming from different backgrounds can be used; and that is indeed the case. Most of the approaches can be distinguished into Model-Based learning and

Model-Free learning. In Model Based learning, the agent uses its experiences in order to learn a model of the process and then find a good decision policy through planning. Model Free learning on the other hand tries to learn a policy directly without the help of a model. Both approaches have their strengths and drawbacks (guarantee of convergence, speed of convergence, ability to plan ahead, use of resources).

2.2.4 Q Learning

Q-learning [8] is a model-free, off-policy, Reinforcement Learning control algorithm that can be used in an online or off-line setting. It uses samples of the form (s, a, r, s') to estimate the state-action value function of an optimal policy. The simple temporal difference update equation for Q-learning is:

$$Q(a, s) = Q(a, s) + \alpha(R(s) + \gamma \max_{a'} Q(a', s') - Q(a, s))$$

Essentially, it is an incremental version of dynamic programming techniques, which imposes limited computational demands at every step, with the drawback that it usually takes a large number of steps to converge. It learns an action-value representation, allowing the agent using it to act optimally (under certain conditions) in domains satisfying the Markov property. Figure 2.18 shows the Q learning algorithm.

2.2.5 LSPI

Least-Squares Policy Iteration (LSPI) [1] is a relatively new, model-free, approximate policy iteration Reinforcement Learning algorithm for control. It is an off-line, off-policy, batch training method that exhibits good sample efficiency and offers stability of approximation. LSPI has met great success in the last few years, applied in domains with continuous or discrete states and discrete actions. LSPI iteratively learns the weighted least-squares fixed-point approximation of the state-action value function of a sequence of improving policies π, by solving the (kxk) linear system

$$Aw^{\pi} = b$$

```
Q LEARNING.
Input: $D, \gamma, Q_0, \alpha_0, \sigma, \pi$. Learns $\hat{Q}^*$ from samples
  // $D$: Source of samples $(s, a, r, s')$
  // $\gamma$: Discount factor
  // $Q_0$: Initial value function
  // $\alpha_0$: Initial learning rate
  // $\sigma$: Learning rate schedule
  // $\pi$: Exploration policy
  $\tilde{Q} \leftarrow Q_0; \alpha \leftarrow \alpha_0; t \leftarrow 0$
  for each $(s, a, r, s')$ in D do
    $\tilde{Q}(s,a) \leftarrow \tilde{Q}(s,a) + \alpha\Big(r + \gamma \max_{a' \in A} \tilde{Q}(s', a') - \tilde{Q}(s,a)\Big)$
    $\alpha \leftarrow \sigma(\alpha, \alpha_0, t)$
    $t \leftarrow t + 1$
  end for
  return $\tilde{Q}$
```

Figure 2.18: The Q Learning algorithm

where w^π are the weights of k linearly independent basis functions. Once all weights have been calculated, each action-state pair is mapped to a Q value, as such:

$$Q(s,a) = \sum_{i=0}^{k} w_i \phi_i(s,a)$$

and the improved action choice in each state s is given by

$$\pi'(s) = \arg\max_{a'' \in \mathcal{A}} Q(s,a)$$

Figure 2.19 summarizes the LSPI algorithm.

```
LSPI.
Learns a policy from samples.
  Input: samples $D$, basis $\phi$, discount factor $\gamma$, error $\epsilon$
  Output: weights $w$ of learned value function
  Initialize $w' \leftarrow \mathbf{0}$
  repeat
    $w \leftarrow w'$, $\mathbf{A} \leftarrow \mathbf{0}$, $b \leftarrow \mathbf{0}$
    for each $(s, a, r, s')$ in $D$ do
      $a' = \arg\max_{a'' \in \mathcal{A}} w^\top \phi(s', a'')$
      $\mathbf{A} \leftarrow \mathbf{A} + \phi(s, a)\Big(\phi(s, a) - \gamma\phi(s', a')\Big)^\top$
      $b \leftarrow b + \phi(s, a)r$
    end for
    $w' \leftarrow \mathbf{A}^{-1}b$
  until $(\|w - w'\| < \epsilon)$
```

Figure 2.19: The LSPI algorithm

Chapter 3

Problem Statement

3.1 Algorithm Selection: A meta-problem

As described in the introductory section, the main issue this work is trying to address is that of algorithm selection. Selecting the right algorithm for a problem is a choice that depends on more things than just the theoretical expectations for time and memory complexity of each algorithm. Details and specifications of the hardware are important, as well as the expected distribution of input from a statistical point of view. Inner details of an algorithm are also important, in combination with the above. This makes algorithm selection a complex problem in itself, a sort of meta-problem. As such, it deserves the same amount of research and experimentation as any regular computational problem. Solutions should be formulated in a generic, methodic way, so that they can be applied with minimal modification to every instance of this meta-problem.

A per-instance solution from scratch for any problem is usually not cost effective, especially when there has already been solutions proposed that apply to the general nature of that problem. This is why there are generic algorithms that apply to general formulations of problems, such as sorting algorithms, search algorithms, constraint satisfaction algorithms, etc. Since we have established the procedure of selecting the correct algorithm as a problem, it would be rational to try to create solutions for that problem that apply in the general case rather than a per-instance method, where each instance is treated like a completely new problem.

This directly implies that the solution will not be the same for each instance, which is why we will refer to this as dynamic algorithm selection. This means that a decision will be made according to the specifics of the instance, but will be made so based on a set of general rules that will apply, rules that have been defined in order to make the best decision based on the specifics.

3.2 Recursive Algorithm Selection

Recursive algorithms work exactly by breaking down their input problem into a sub-problem until they reach a base problem of much smaller complexity, and gradually reconstruct the solution by composing the final solution out of the solutions of all the base sub-problems that the original problem was broken down to. Essentially, each recursive call is equivalent to the original one, only applied to a smaller problem. So, the algorithm selection problem appears at each recursive step, when we consider recursive algorithms.

In this work, we take this one step further. Why treat algorithm selection as a problem whose solution should apply to an entire instance? Why not consider that a sub-instance of a specific instance could, potentially, be solved better by another algorithm than the one that seemingly seems the better choice for an instance overall?

To put it plainly, why should an algorithm be chosen to solve an entire instance, even if the choice has been made in a methodic, regulated way? Problems usually can be broken down into sub-problems, and it can be the case that certain sub-problems can be better solved by another algorithm than the one that was chosen to solve the entire problem. This is what this thesis attempted to investigate and implement.

3.3 Recursive Algorithm Selection in Tree-Search

We consider the problem of algorithm selection for Tree-Search algorithms, which are recursive algorithms. During Tree-Search, an algorithm begins at the root of a tree and traverses through the tree in search of a solution. This traversal is recursive. Like we described above, recursive algorithms break down the original problem into sub-problems. In Tree-Search, the original tree is broken down into sub-trees during each recursive step. We treat each sub-problem and the algorithms associated with it as a new algorithm selection problem that needs to be treated based on its specific characteristics, rather than treating the entire search tree as one problem and selecting an entire algorithm to solve it.

The problem we attempt to solve is finding a decision policy that would choose an algorithm at each recursive step of the Tree-Search procedure. Since there has not been any related work in the field of Tree-Search, we also had to solve the problem of modeling Tree-Search in such a way that solving the algorithm selection problem at each recursive step is possible to begin with.

3.4 Related Work

Treating algorithm selection as a problem in itself is not a completely new idea. It was first stated formally by J.R. Rice in 1976 [9] as a computational problem. The Algorithm Portfolio was introduced as a paradigm to treating algorithm selection as a formal computational problem [10].

Since algorithm selection has been established as a computational problem, there have been a few approaches as to how it should be solved. Those include studying the instance to be solved and selecting a proper algorithm for it [11], running multiple algorithms from the portfolio in parallel and terminating as soon as one solution is obtained by the fastest algorithm, [12] and runtime switching of algorithms [13].

The work of Lagoudakis, Littman, and Parr [14, 15, 16] is the one closest to ours, since they consider learning decision policies for Recursive Algorithm Selection. In particular, they have shown that efficient hybrid algorithms can be obtained for the problems of sorting, order-statistic selection, and branching rule selection in the DPLL procedure for satisfiability.

Chapter 4

The Proposed Approach

We propose a recursive algorithm selection system which refines its decision making policy through the LSPI Reinforcement Learning algorithm in order to make the choices that will allow for optimum performance.

As described in the Problem Statement section of this document, we treat each recursive step of a Tree-Search as a new instance of the algorithm selection problem. We use Reinforcement Learning and the LSPI algorithm in specific to solve each instance of this problem.

The implementation, however, and the design do not restrict themselves to these problem classes and domains; They are general and with little to no modification can support different problem classes and domains. We use Tree-Search and shortest-path as a specific running example, a way to showcase the features.

We implemented a basic Tree-Search framework that would support any Tree-Search algorithm. As described in the background section, the general Tree-Search algorithm makes use of a fringe, needs a goal test, and an expand function. Since we are performing heuristic search, a heuristic function was also provided. We then proceeded to add two different algorithms to this framework. The framework was designed and implemented in such a way that it can support any number of different algorithms attempting to solve the problem at any given time, so that interleaving of the available algorithms is possible with as little modification to the original algorithms needed as possible. No knowledge of the internal workings of the algorithms is embedded in the framework, to keep the generic nature of the framework. Once algorithms have been added to the framework and their interleaving made possible, a training dataset is collected, by making random algorithm choices to solve a few problems. LSPI is used to process this dataset and find a policy that minimizes the penalty function. Once this policy has been found by LSPI, it is implemented into the original framework, and the hybrid algorithm is benchmarked against the original algorithms using the penalty function.

4.1 The problem used

The problem we decided to use for our Tree-Search algorithms was a simple shortest-path problem. We used a graph generated from real world data, with nodes in the graph representing points on the map. The points are all within the state of Washington D.C., with geographical coordinates provided by TIGER/Line. The edges of the graph are also provided by TIGER/Line and are weighted and undirected. They represent the distance in meters between two nodes, and an edge exists between two nodes when these two nodes are directly connected on the map with a street, a road, a highway etc. An instance is created when a start and finish node is selected from that graph. The solution asked for is one of the shortest paths from the start node to the finish node.

4.2 The Framework

We designed and implemented a framework that will support problem definitions of any kind and different Tree-Search algorithms that can be applied on the problem defined. The framework also supports more than one heuristic. Based on the general Tree-Search algorithm described in section 2.1.2, all the necessary structures and methods are available for all Tree-Search algorithms, and with the presence of heuristics, any heuristic search algorithm can be implemented on the framework with everything provided for.

4.2.1 Class outline

The main concept behind allowing multiple algorithms to run on the same problem instance at the same time is to separate data and data structures from the algorithms themselves; the search tree is not part of the algorithm, and the same goes for the other structures and methods described in the general Tree-Search algorithm, like the fringe, the expand function and the goal test function. These are all part of the problem definition and modeling/representation. So they can be separated from the algorithm, and once separated, they can be used by any number of algorithms. So the framework consists of an N-ary Tree class, with all the necessary methods to traverse it, modify it, without destroying it when the algorithm has finished. It also includes an initially empty set of algorithms, and an initially empty set of heuristics. Algorithms and heuristics can be added by a simple instruction. The framework includes of course the necessary data to represent the problem - in our specific example, an adjacency matrix that represents the original Washington DC graph. Finally, it also includes necessary functions and data for benchmarking and logging.

4.2.2 The Fringe

The fringe of the search tree is not duplicated, it is simply pointed to. All nodes in the fringe are linked to each other, in a double linked list that can be traversed forwards or backwards. The current recursive step is always called on a node that is on the fringe when the recursive step begins. All internal tree methods are designed to comply with these constraints: Maintaining the fringe, making sure that recursion begins at a node on the fringe each time etc.

4.2.3 The Expand Function

The expand function is part of the problem definition and not of the tree, as described in section 2.1.1. In our case, the shortest path problem, expanding a tree node means generating all tree nodes that represent adjacent graph nodes to the graph node that is represented by the tree node we are expanding. Adjacent nodes can be found by looking through the adjacency matrix that represents the graph.

4.2.4 The Heuristic

As described in sections 2.1.5 and 2.1.4, both RBFS and A* are optimal when the heuristic used is admissible. In our shortest-path problem, an obvious choice for an admissible heuristic is the straight line distance between two nodes. Since the smallest possible distance between two points in any dimensional space is the straight line distance, it will always be smaller or equal to the actual distance of the two points.

Since we have geographical coordinates available, calculating distance in meters is not straightforward. Latitude (φ) and longitude (λ) coordinates are coordinates on the spheroid plane of earth, not a flat Cartesian plane. Hence, some conversion is needed that will approximate the spheroid plane into a flat plane. The formula used to calculate the distance d in meters between 2 points (a, b) with φ and λ coordinates is:

$$d = 60 * 1.1515 * 1.609344 \quad * \quad 1000 * \frac{180}{\pi} * \\ \arccos\big(\sin(\frac{\pi}{180}\phi_a)\sin(\frac{\pi}{180}\phi_b) \quad + \quad \cos(\frac{\pi}{180}\phi_a)\cos(\frac{\pi}{180}\phi_b)\cos(\frac{\pi}{180}\lambda_a - \lambda_b)\big)$$

For distances in the order of a few kilometers, this approximation does not compromise accuracy when we need distances expressed in meters with no decimal digits.

4.2.5 The Algorithms

The algorithms are as described in sections 2.1.5 and 2.1.4. There are some minor changes to allow for improved performance, such as not generating the parent node when expanding a node, and completely removing a node from the tree when it does not generate any nodes if expanded (ie, a dead-end node). RBFS uses Quick Sort to sort the successors of the current node.

4.3 Interleaving algorithms

Interleaving different algorithms is no simple task. Each of the two algorithms is designed with certain assumptions made. In the case of recursive algorithms, the main assumption is that the same algorithm is making all the recursive calls, and that the arguments passed on from recursive step to recursive step follow the internal dictations of the algorithm. Certain modifications need to be made in order to keep the algorithms consistent even if the algorithm making the recursive calls is not always the same one.

4.3.1 Recursive Interference

This still leaves the question of how the actual interleaving works. The idea was first introduced in [16]. We intercept every recursive call with a call to an internal function of the framework. This function will decide which algorithm will be called on the next recursive step. The sequence of decisions made by this function will eventually form the hybrid algorithm, as it will not always call the same algorithm over and over, but rather make a different decision each time based on a certain policy.

4.3.2 Scope and Internal Knowledge

Since we are trying to keep the framework generic, we do not want to incorporate any knowledge of the internal workings of the algorithms in the framework itself. This is why we need to separate the problem from the algorithm in terms of scope, as described above. Everything that an algorithm needs that is outside of the general algorithm definition needs to be added.

4.3.3 An Example

Figure 4.1 shows the hybrid creation algorithm, ie the decision making function that intercepts recursive calls.

Using this function, we present the following set of figures that demonstrate how the example used in sections 2.1.4 and 2.1.5 would be handled by our framework.

4.4 The Learning Process

As explained above, we intercept all recursive calls and replace them with a function that decides which algorithm will be responsible for the next recursive

```
HYBRID CREATION. Input: AbstractArg.
Called at each recursive step and decides next algorithm
  // $w$: The set of weights provided by LSPI
  // $\phi$: The base functions
  for each $a$ in $portfolio$ do
    $q[a] \leftarrow \sum_{i=0}^{\Phi} w(a)_i \phi(s)_i$
  end for
  $best \leftarrow min(q)$
  $AbstractReturn \leftarrow portfolio[best](AbstractArg)$ //Recursive call.
  //portfolio[best] is a function pointer.
  return $AbstractReturn$
```

Figure 4.1: The Hybrid creation algorithm

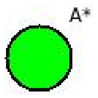


Figure 4.2: Hybrid Example: Step 1

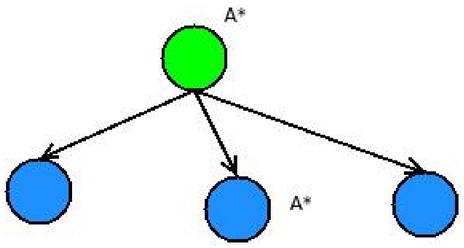


Figure 4.3: Hybrid Example: Step 2

step. This decision is made based on a policy that we provide. This policy comes from the learning process, which takes place offline.

In practice, what takes place during the decision making is that this decision making function calculates two functions: The Q function for A* and RBFS. Since our problem, the shortest-path route, is a cost minimization problem, the algorithm that gets chosen is the one that has the lowest Q function value.

In order to calculate the Q function (see section 2.2.5 for the definition of Q

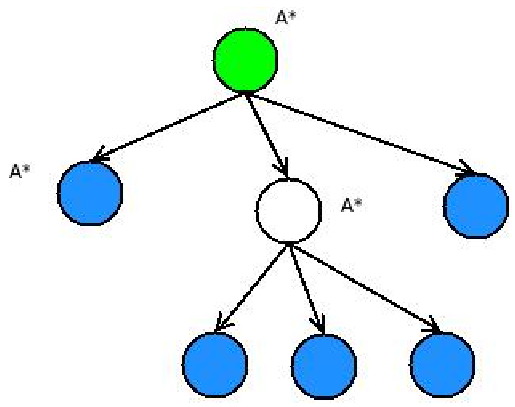


Figure 4.4: Hybrid Example: Step 3

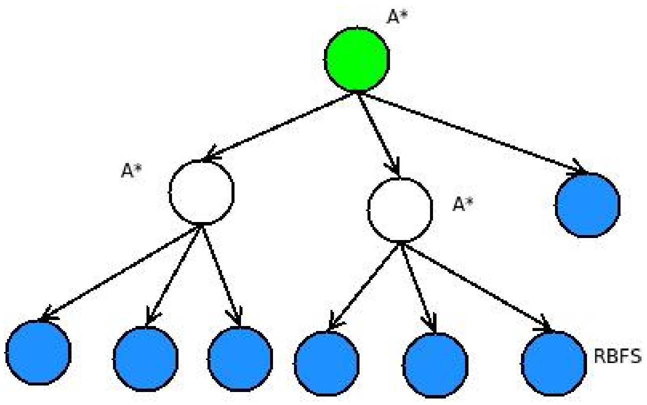


Figure 4.5: Hybrid Example: Step 4

function) a data set needs to be collected. A data set that will log decisions and their impact. By impact we mean their penalty function value. We define the penalty function as a linear combination of time and memory.

Once this data set has been collected, LSPI works on converging to a policy that will minimize the penalty function. This policy is a set of weights for the different base functions, for each of the actions available. These weights are coded in the decision making function to calculate Q values for all actions and make the decision.

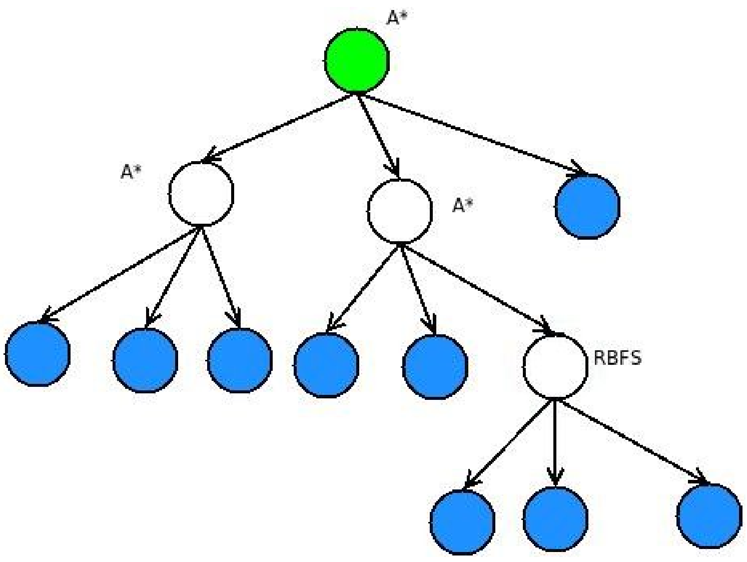


Figure 4.6: Hybrid Example: Step 5

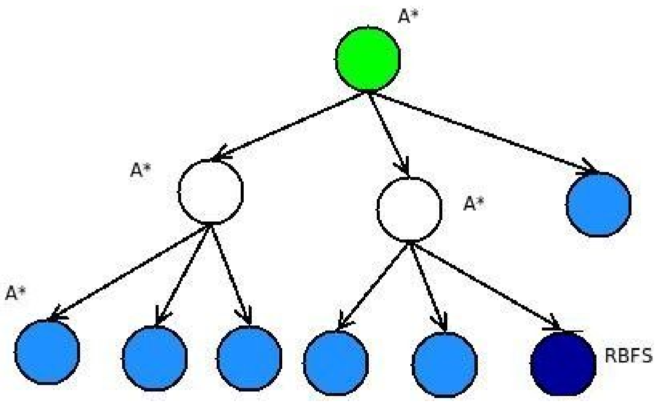


Figure 4.7: Hybrid Example: Step 6

4.4.1 The Penalty Function

As stated above, we need a measure for how well or how bad certain choices are. We define cost as a linear combination of time and memory. Time is measured in microseconds and memory in bytes. To make combination possible, we need to scale memory and time into comparable magnitudes. By comparing time values and memory values in the data set, the ratio that we ended up using was 1μς$\sim$ 5 bytes.

That said, there is still the question of just how much importance does time

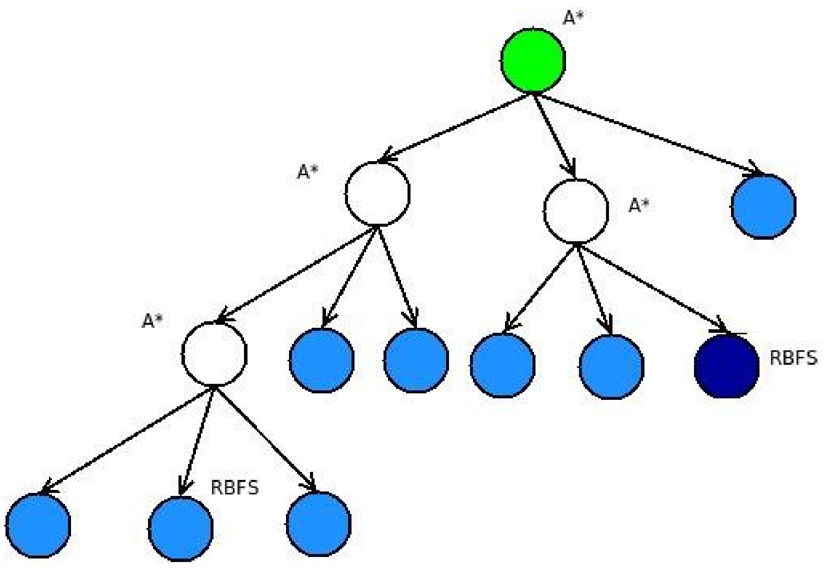


Figure 4.8: Hybrid Example: Step 7

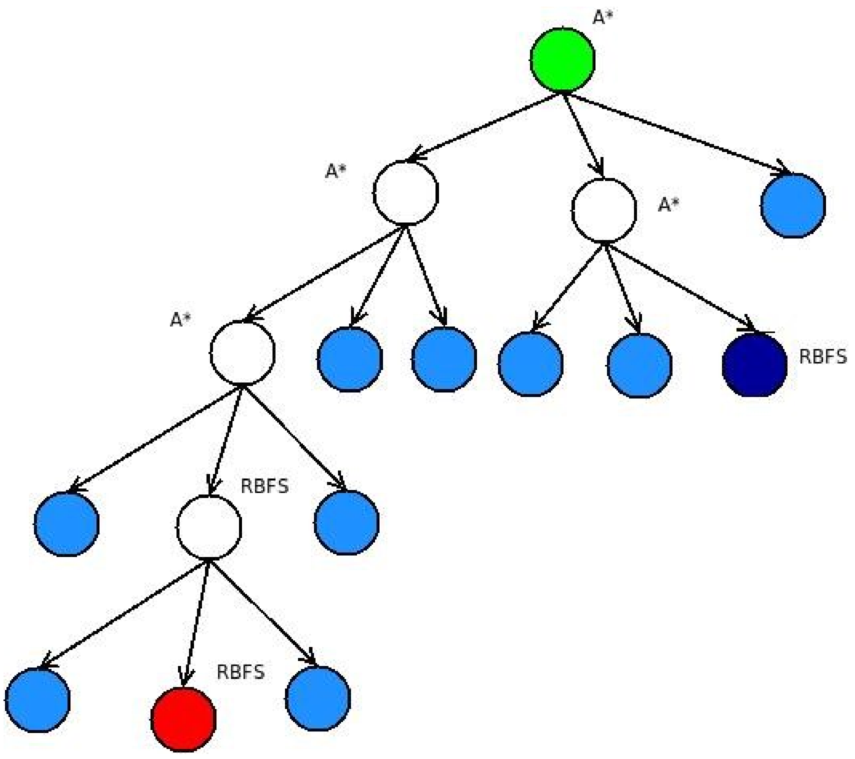


Figure 4.9: Hybrid Example: Step 8

have against memory. This question has no real answer; it is different each time. This is why we decided to test a lot of different combinations of time and memory contribution. Starting from 0% time - 100% memory and ending up to 100% time and 0% memory in steps of 10%, we defined 11 different penalty functions:

$$\begin{aligned}
p_1 &= 0.0T + 1.0M \\
p_2 &= 0.1T + 0.9M \\
p_3 &= 0.2T + 0.8M \\
p_4 &= 0.3T + 0.7M \\
p_5 &= 0.4T + 0.6M \\
p_6 &= 0.5T + 0.5M \\
p_7 &= 0.6T + 0.4M \\
p_8 &= 0.7T + 0.3M \\
p_9 &= 0.8T + 0.2M \\
p_{10} &= 0.9T + 0.1M \\
p_{11} &= 1.0T + 0.0M
\end{aligned}$$

Naturally, for each different definition of the penalty function, a different policy is applied.

4.4.2 Base Functions

The base functions we defined for LSPI were derived from "geometrical" characteristics of the tree. We define the following set of base functions on a given node:

1. Depth
2. Amount of child nodes
3. Cost of cheapest child node
4. Cost of most expensive child node
5. Average cost of child nodes

These values should give a picture of the geometrical characteristics of the tree around that node. We decided to use these characteristics because the performance of the algorithms greatly depends on geometrical characteristics of the tree, and with these base functions we should get a good distinction of when to use which algorithm.

4.4.3 Logging

Since we have no policy to decide actions prior to collecting the initial training data set, we started with a completely random policy, uniformly choosing from all available actions, randomly at each recursive step. The application logged the action taken, the values of the base functions at that point, the time and memory separately, and then the values of the base functions after the action was taken, and a true/false value in case the state that resulted was a goal state. This took place at each recursive step, from start to finish of an instance. This was repeated for 100 different instances, collecting a total of about 5000 samples.

Logging time and memory separately allowed us to execute a single sample collecting run. The parsing of the data set by LSPI could create all 11 different penalty functions from the same data set.

4.4.4 LSPI

We defined a new domain for LSPI, named SEARCH. As described in the LSPI documentation, we had to define a new set of base functions, which we described above, and parse the data set to create a sample set for each definition of the penalty function.

Once the data set has been parsed, a sample set is created. LSPI takes this sample set and begins iterating in an attempt to converge to an optimal policy based on the penalty function.

The LSPI learn function was called with a discount factor of 1.0, an epsilon value of 0.001, and a very high limit of iterations in order to ensure convergence, but this was not needed as LSPI converged at 4 or 5 iterations each time.

4.4.5 Policy implementation

Implementing the policy LSPI provides is as easy as typing some fixed values in the original code. LSPI returns a set of weights that correspond to the set of base functions. This set of weights was in turn coded by hand in the original application. It is this set of weights that defines exactly how much of each algorithm will be part of the final hybrid algorithm, since these weights affect the final value of the Q functions of each action.

Since we have 11 different definitions of the penalty function, naturally there are 11 different policies that needed to be implemented separately.

4.5 Benchmarking

After successfully implementing a policy, the algorithm that was created was tested against the original algorithms to see if the policy that we found was worth the extra cost or not.

4.5.1 Comparison Validity

We are comparing algorithms to one another. Since we provide 11 different definitions for the penalty function, a comparison between the three algorithms (A*, RBFS, Hybrid) needs to be made separately for each of the 11 definitions of the penalty function. Comparing performance of one algorithm to another with a different definition of the penalty function is completely pointless and bears no actual meaning. The hybrid version was compared against the best of the two original algorithms at each different penalty function definition.

4.5.2 Performance Definition

Since we are trying to compare two algorithms that are competing against each other by excelling on the opposing ends of the time/memory trade-off, it is natural that performance needs to be defined on these two terms. This is why the penalty function described above is a linear combination of time and memory. Beginning from favoring one end and gradually shifting to the other means that performance is defined differently each time, favoring the term that has the biggest contribution in the linear combination.

Chapter 5

Implementation Details

5.1 Language of Choice and Reasoning

Since we decided to go with a more abstract and generic design for the framework in order to facilitate re-definition of some of its aspects and algorithm interleaving, object oriented languages were sort of mandatory. There was also a need for a more direct memory access and system access as well for benchmarking; therefore scripting languages and Java were not a good choice. This led to C++ as the language of choice for this implementation.

LSPI however is implemented in MATLAB, but since it is an offline learning algorithm, it did not affect the choice for the main application language. LSPI could even be run on an entirely different system than the one that the application was running on.

5.2 Operating System, Hardware Specifications and their impact

The entire application was written with as much platform independency in mind as possible, but benchmarking needs shifted implementation to be Linux only. The Eclipse IDE [17] was used which makes use of the gcc compiler for C++ [18], and the Linux libraries for measuring real execution time were used for benchmarking, and the Debian-based distribution Ubuntu in specific.

Linux also provides a very versatile environment for manipulating various aspects of the application that are after the compilation. Since we have a lot of recursive calls, it is sometimes the case that the call stack allocated by default from the operating system is just not enough. Stack size and other aspects of the application runtime are easily manipulated in Linux using the terminal command "ulimits".

The system on which the application was benchmarked is a 64-bit system, which allowed for larger integers and longs, possibly needed when calculating the Q() functions. The processor is dual core with a core frequency of 2GHz on both cores.

5.3 Debugging and Profiling

The debugger used was gdb [19] from within the Eclipse environment and sometimes the DDD front-end for gdb [20] due to the complexity of debugging large trees and other data structures.

Since memory management is vital to the application's performance and benchmarking, there could be no leaks, mis-allocations or corrupt deallocations. This made the need for profiling very important. The Valgrind profiler [21] was used for that purpose, and made proofing the application against bad memory management a lot easier.

5.4 Code structure, memory management, function pointers

Following the typical object oriented approach, each class is defined in a separate pair of .cpp and .h files. Header files contain the declaration and interface of a class while the source files contain their implementation. The code is arranged in such a way that the developer only needs to write the main.cpp file, in which they need to implement the algorithms that they need to add as well as the heuristics if any. After including the main header of the framework, all the necessary functions to add the algorithms to the framework and begin working on the problem are available in a intuitive way that allows for easy use of the classes.

As described above, memory management is vital to the purpose of this work, so great care and work has been put into it. Every chunk of memory that is allocated by the framework is deallocated exactly when needed, ensuring data consistency. The algorithms that were implemented for this work were also implemented with memory management being a high priority.

Finally, it is worth noticing that the entire process of adding algorithms to a set might sound simple, but it is not something that comes naturally in programming languages. This is achieved by using function pointers, and using an abstract class for passing arguments and returning results from algorithms. The developer needs to extend these abstract classes to fit his own needs. This is so because adding an unknown number of algorithms to a class can only be achieved by using an array of function pointers, which implies that their prototype needs to be the same. Heuristics are also added in the same way.

Chapter 6

Results

6.1 The benchmarking process

The first step is selecting the instances on which the algorithms will be benchmarked. We tried two different sets of instances, each one having 100 instances of a shortest path problem from the Washington D.C. graph. The first instance set was the same set of paths that created the training data set (”Training instance set”). Figure 6.1 shows these paths on the graph. After that, each algorithm (A*, RBFS and Hybrid) was called to solve these 100 instances, measuring performance in terms of time and memory. The same was repeated for the second instance set, which was composed of 100 different instances to the ones in the original set, for which no learning had been applied (”Testing instance set”). Figure 6.2 shows the paths from the second instance set on the graph.

6.2 Comparisons

As we have 11 different penalty functions, we benchmarked the 3 algorithms in all 11 definitions of the penalty function, for each of the two instance sets. What follows is the comparison charts for each of the two instance sets. For each penalty function we provide both the absolute penalty function charts over 100 instances, and the percentage gain over 100 instances.

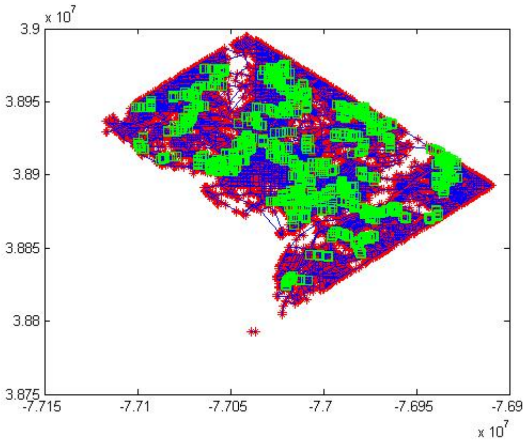


Figure 6.1: Training instance set paths on the Washington DC graph

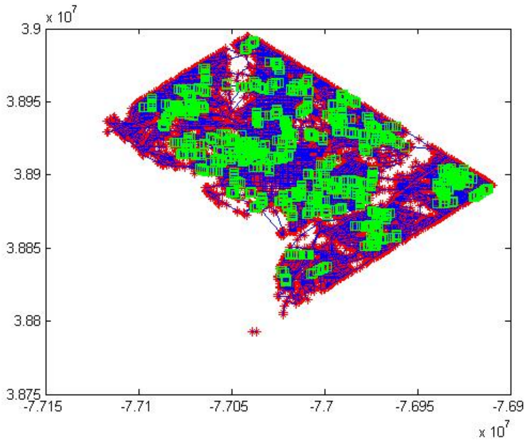


Figure 6.2: Testing instance set paths on the Washington DC graph

6.2.1 Training Instance Set

Here we present the charts for the training instance set, the one from which we extracted the training samples. The charts are presented per penalty function.

Figure 6.3 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.4 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_1 = 0.0T + 1.0M$

Figure 6.5 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.6 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_2 = 0.1T + 0.9M$

Figure 6.7 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.8 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_3 = 0.2T + 0.8M$

Figure 6.9 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.10 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_4 = 0.3T + 0.7M$

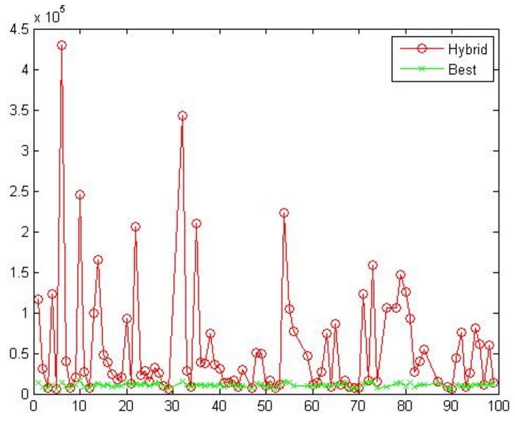


Figure 6.3: Absolute cost, 0% time - 100% memory, training instance set

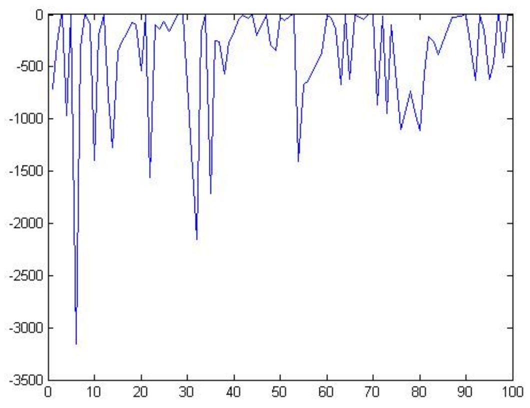


Figure 6.4: Percentage gain, 0% time - 100% memory, training instance set

$$p_1 = 0.0T + 1.0M$$

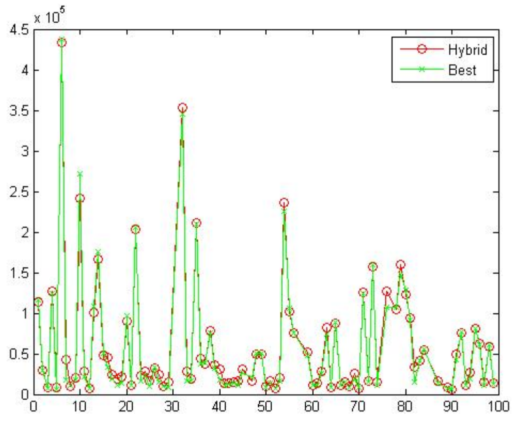


Figure 6.5: Absolute cost, 10% time - 90% memory, training instance set

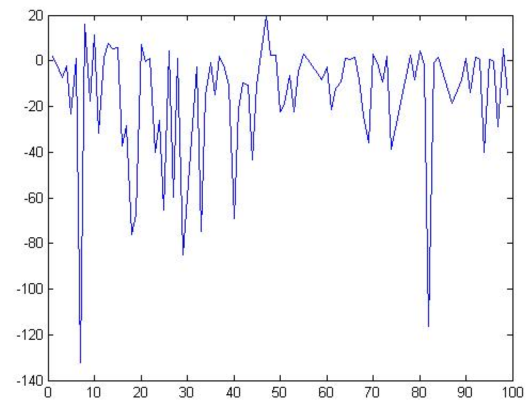


Figure 6.6: Percentage gain, 10% time - 90% memory, training instance set

$$p_2 = 0.1T + 0.9M$$

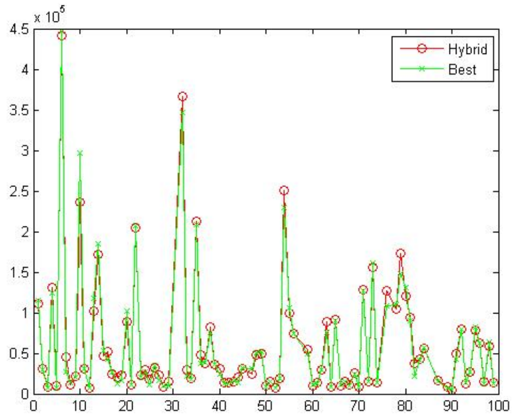


Figure 6.7: Absolute cost, 20% time - 80% memory, training instance set

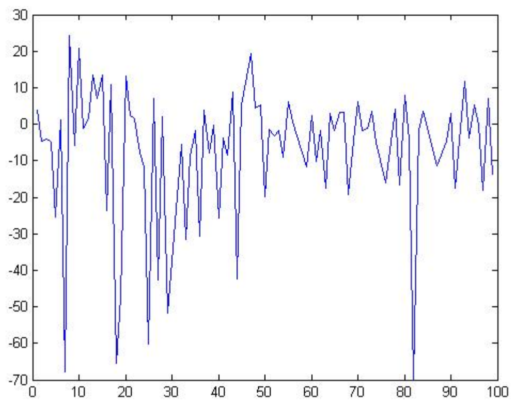


Figure 6.8: Percentage gain, 20% time - 80% memory, training instance set

$$p_3 = 0.2T + 0.8M$$

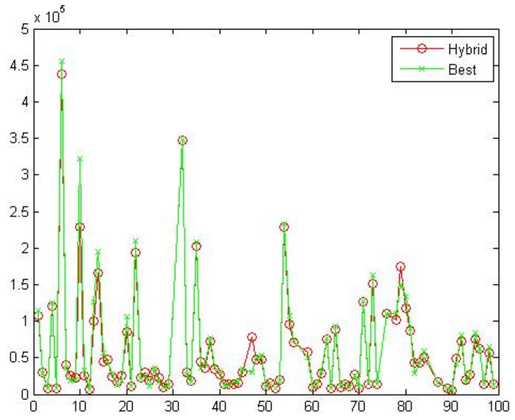


Figure 6.9: Absolute cost, 30% time - 70% memory, training instance set

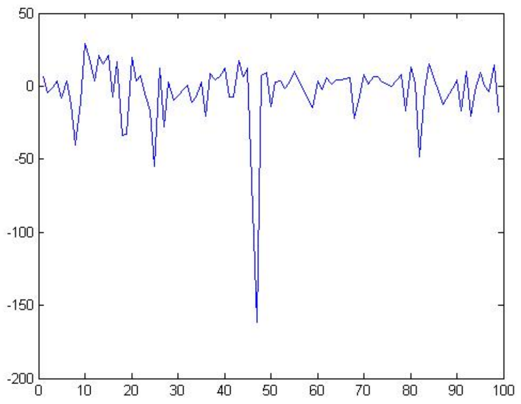


Figure 6.10: Percentage gain, 30% time - 70% memory, training instance set

$$p_4 = 0.3T + 0.7M$$

Figure 6.11 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.12 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_5 = 0.4T + 0.6M$

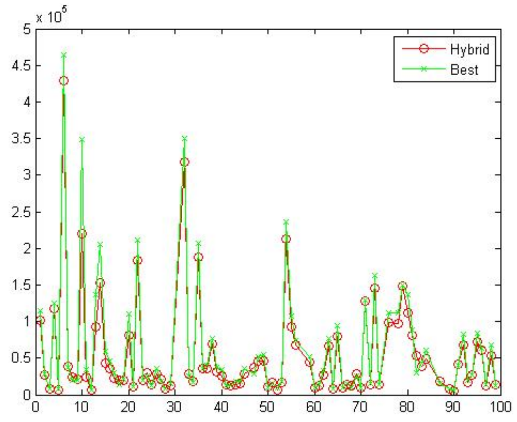


Figure 6.11: Absolute cost, 40% time - 60% memory, training instance set

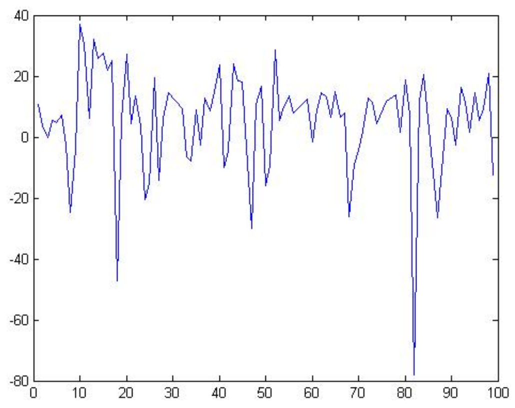


Figure 6.12: Percentage gain, 40% time - 60% memory, training instance set

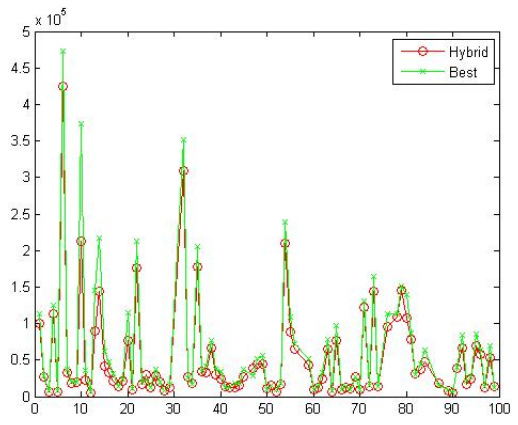
$$p_5 = 0.4T + 0.6M$$

Figure 6.13 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.14 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_6 = 0.5T + 0.5M$

Figure 6.15 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.16 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_7 = 0.6T + 0.6M$

Figure 6.17 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.18 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_8 = 0.7T + 0.3M$

Figure 6.19 shows the absolute cost of the hybrid algorithm against the best



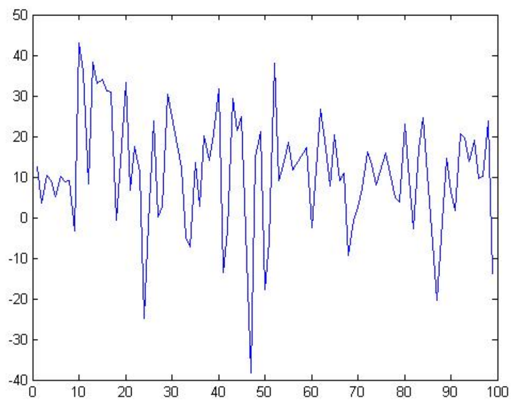
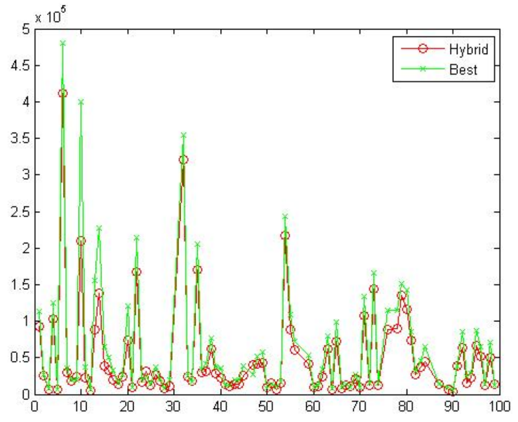


Figure 6.13: Absolute cost, 50% time - 50% memory, training instance set

Figure 6.14: Percentage gain, 50% time - 50% memory, training instance set

$$p_6 = 0.5T + 0.5M$$



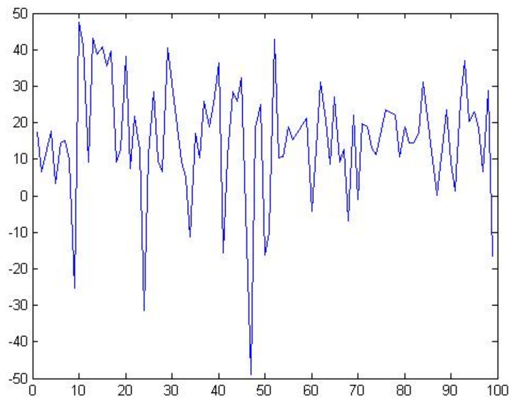


Figure 6.15: Absolute cost, 60% time - 40% memory, training instance set

Figure 6.16: Percentage gain, 60% time - 40% memory, training instance set

$$p_7 = 0.6T + 0.4M$$

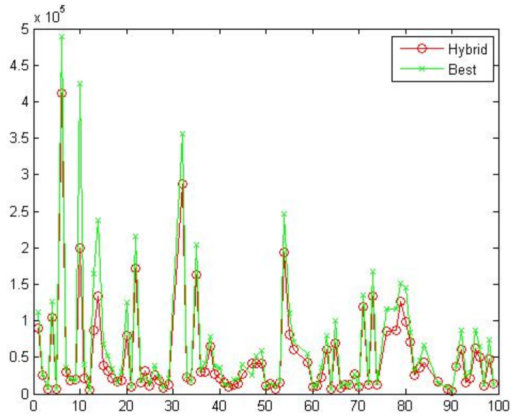


Figure 6.17: Absolute cost, 70% time - 30% memory, training instance set

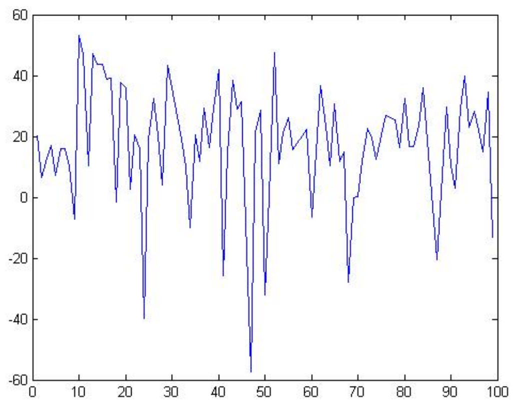


Figure 6.18: Percentage gain, 70% time - 30% memory, training instance set

$$p_8 = 0.7T + 0.3M$$

of the original algorithms, over 100 instances. Figure 6.20 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_9 = 0.8T + 0.2M$

Figure 6.21 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.22 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_{10} = 0.9T + 0.1M$

Figure 6.23 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.24 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_{11} = 1.0T + 0.0M$

6.2.2 Testing Instance Set

Here we present the charts for the testing instance set, containing instances different from the ones that were used to collect the training samples. The charts are presented per penalty function.

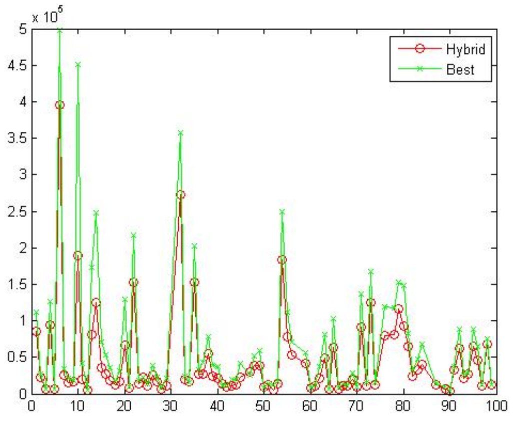


Figure 6.19: Absolute cost, 80% time - 20% memory, training instance set

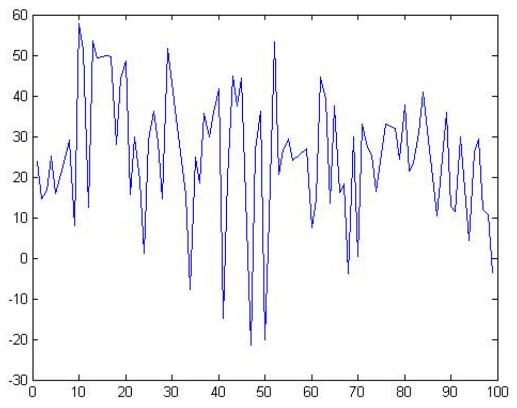


Figure 6.20: Percentage gain, 80% time - 20% memory, training instance set

$$p_9 = 0.8T + 0.2M$$

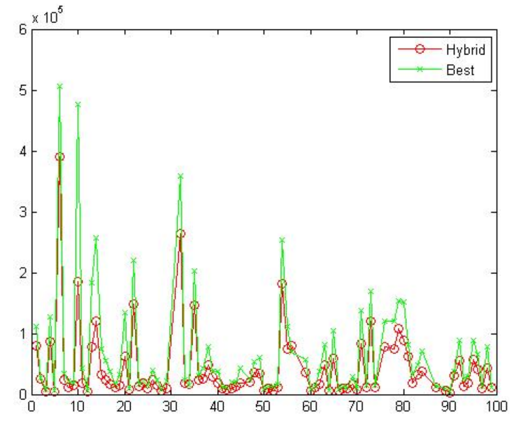


Figure 6.21: Absolute cost, 90% time - 10% memory, training instance set

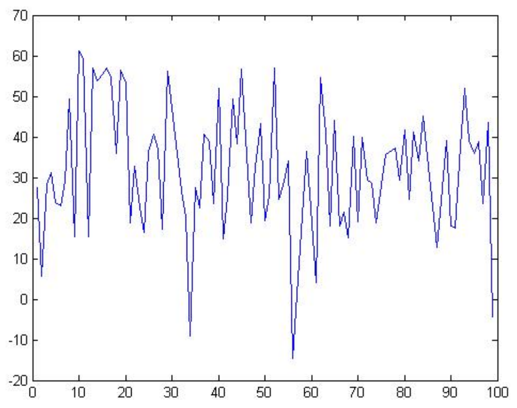


Figure 6.22: Percentage gain, 90% time - 10% memory, training instance set

$$p_{10} = 0.9T + 0.1M$$

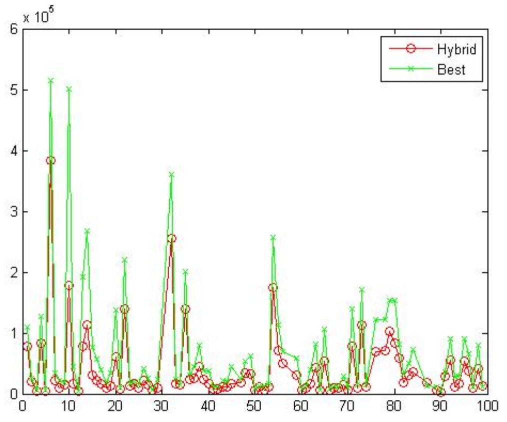


Figure 6.23: Absolute cost, 100% time - 0% memory, training instance set

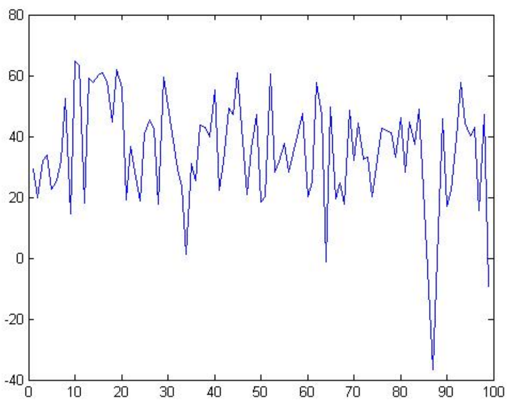


Figure 6.24: Percentage gain, 100% time - 0% memory, training instance set

$$p_{11} = 1.0T + 0.0M$$

Figure 6.25 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.26 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_1 = 1.0T + 0.0M$

Figure 6.27 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.28 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_2 = 0.1T + 0.9M$

Figure 6.29 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.30 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_3 = 0.2T + 0.8M$

Figure 6.31 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.32 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_4 = 0.3T + 0.7M$

Figure 6.33 shows the absolute cost of the hybrid algorithm against the best

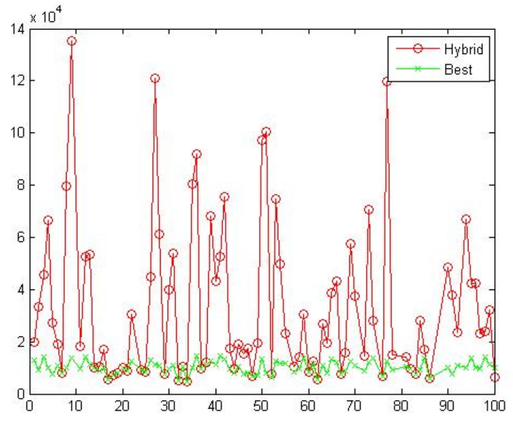


Figure 6.25: Absolute cost, 0% time - 100% memory, testing instance set

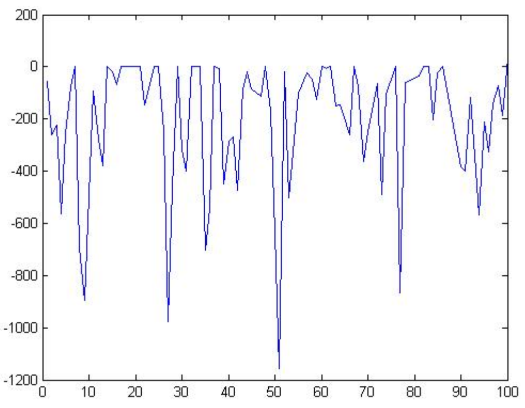


Figure 6.26: Percentage gain, 0% time - 100% memory, testing instance set

$$p_1 = 0.0T + 1.0M$$

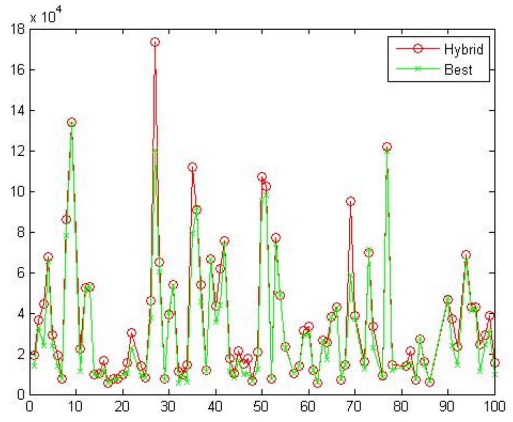


Figure 6.27: Absolute cost, 10% time - 90% memory, testing instance set

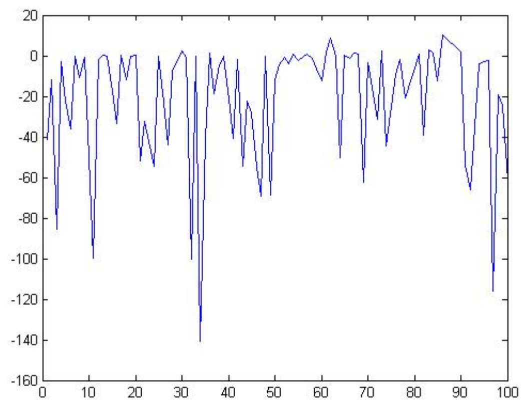


Figure 6.28: Percentage gain, 10% time - 90% memory, testing instance set

$$p_2 = 0.1T + 0.9M$$

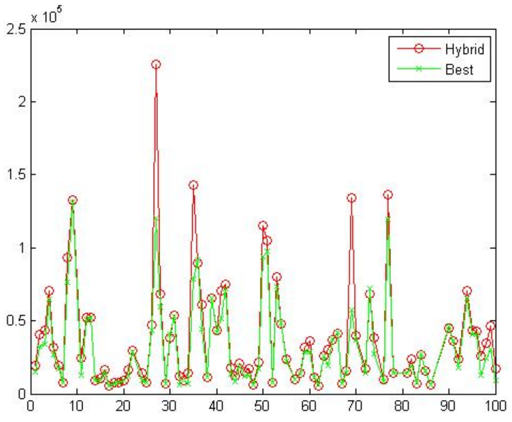


Figure 6.29: Absolute cost, 20% time - 80% memory, testing instance set

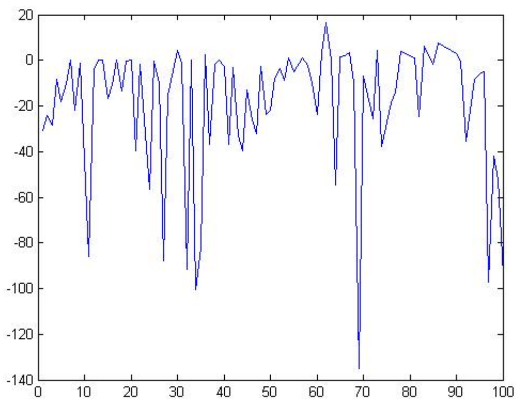


Figure 6.30: Percentage gain, 20% time - 80% memory, testing instance set

$$p_3 = 0.2T + 0.8M$$

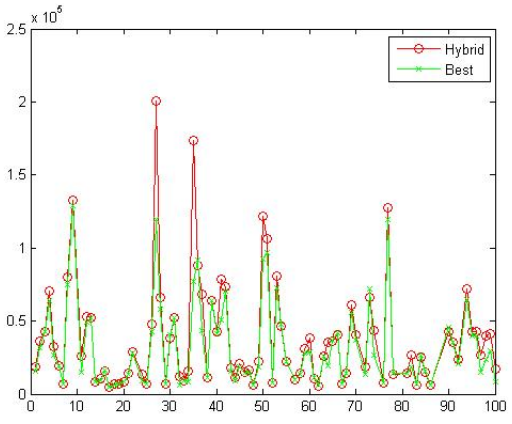


Figure 6.31: Absolute cost, 30% time - 70% memory, testing instance set

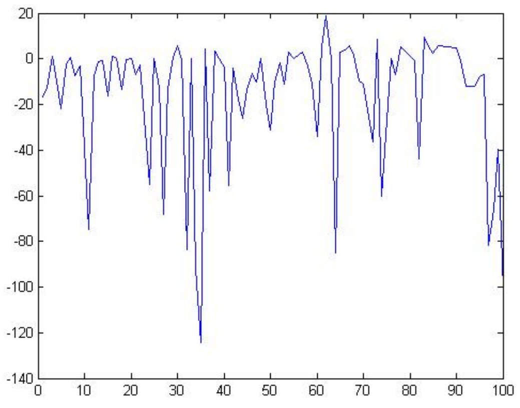


Figure 6.32: Percentage gain, 30% time - 70% memory, testing instance set

$$p_4 = 0.3T + 0.7M$$

of the original algorithms, over 100 instances. Figure 6.34 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_5 = 0.4T + 0.6M$

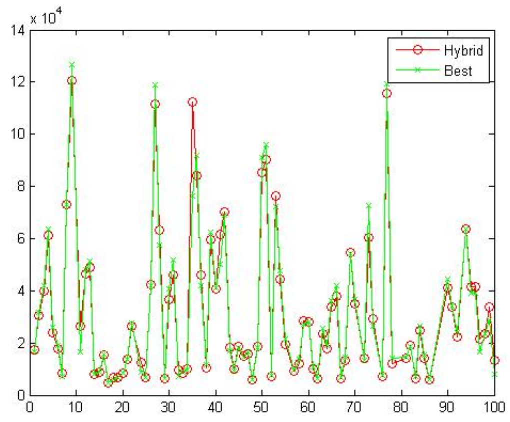


Figure 6.33: Absolute cost, 40% time - 60% memory, testing instance set

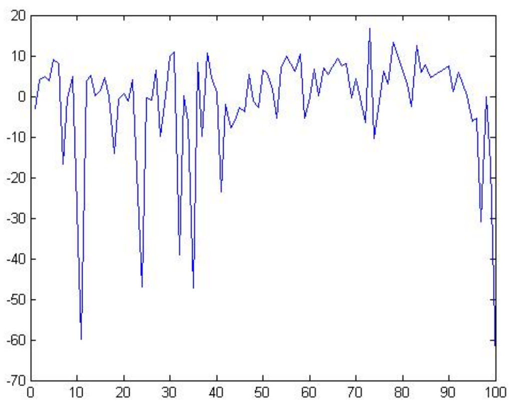


Figure 6.34: Percentage gain, 40% time - 60% memory, testing instance set

$$p_5 = 0.4T + 0.6M$$

Figure 6.35 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.36 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_6 = 0.5T + 0.5M$

Figure 6.37 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.38 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_7 = 0.6T + 0.4M$

Figure 6.39 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.40 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_8 = 0.7T + 0.3M$

Figure 6.41 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.42 shows the percentage

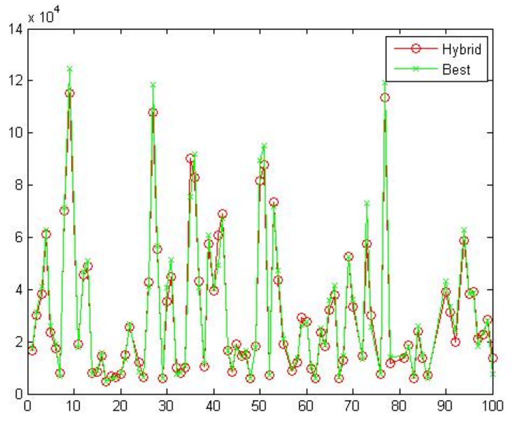


Figure 6.35: Absolute cost, 50% time - 50% memory, testing instance set

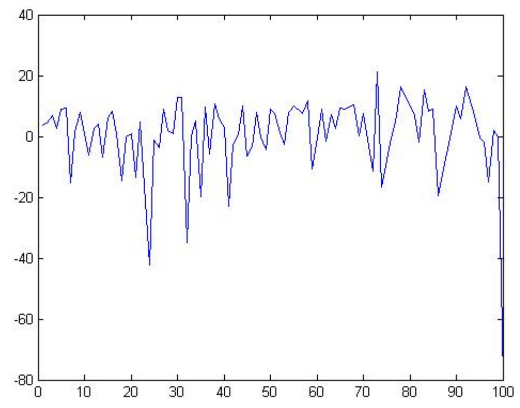


Figure 6.36: Percentage gain, 50% time - 50% memory, testing instance set

$$p_6 = 0.5T + 0.5M$$

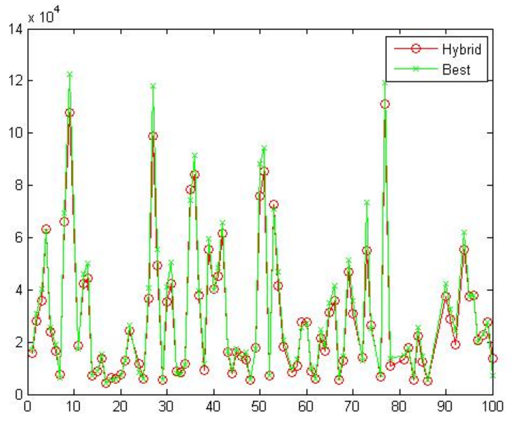


Figure 6.37: Absolute cost, 60% time - 40% memory, testing instance set

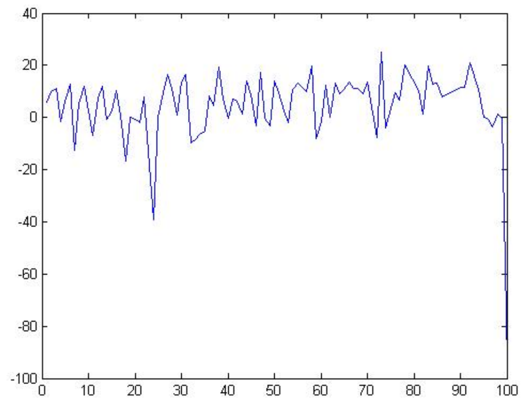


Figure 6.38: Percentage gain, 60% time - 40% memory, testing instance set

$$p_7 = 0.6T + 0.4M$$

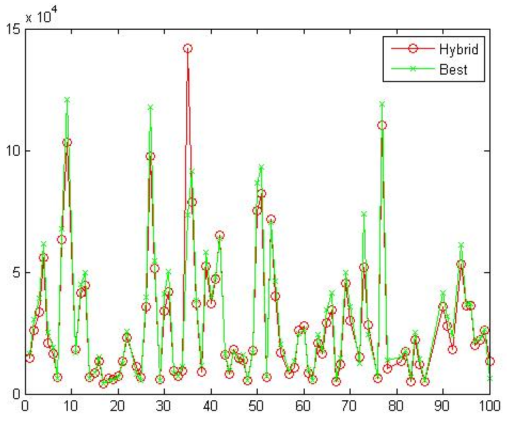


Figure 6.39: Absolute cost, 70% time - 30% memory, testing instance set

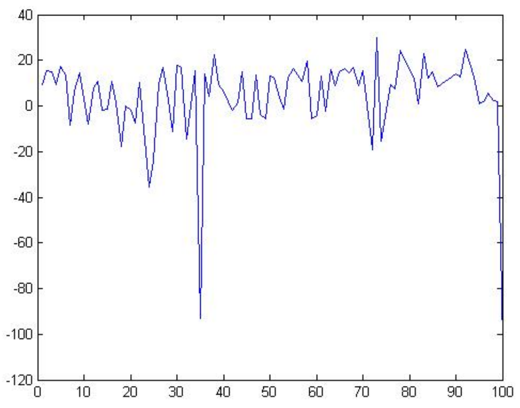


Figure 6.40: Percentage gain, 70% time - 30% memory, testing instance set

$$p_8 = 0.7T + 0.3M$$

gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_9 = 0.8T + 0.2M$

Figure 6.43 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.44 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_{10} = 0.9T + 0.1M$

Figure 6.45 shows the absolute cost of the hybrid algorithm against the best of the original algorithms, over 100 instances. Figure 6.46 shows the percentage gain of the hybrid against the best of the original algorithms on the same 100 instances. The penalty function for these figures is $p_{11} = 1.0T + 0.0M$

6.3 Overall Figures

It is of course more easy to see exactly what was achieved in the following charts that present the results concentrated, averaging the performance of each of the 11 tests over the 100 instances.

Figure 6.47 shows the absolute average performance in the training instance

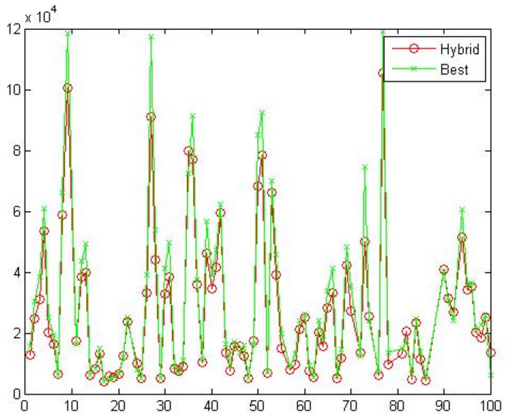


Figure 6.41: Absolute cost, 80% time - 20% memory, testing instance set

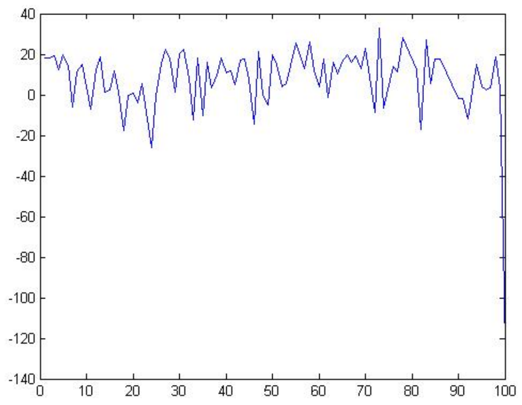


Figure 6.42: Percentage gain, 80% time - 20% memory, testing instance set

$$p_9 = 0.8T + 0.2M$$

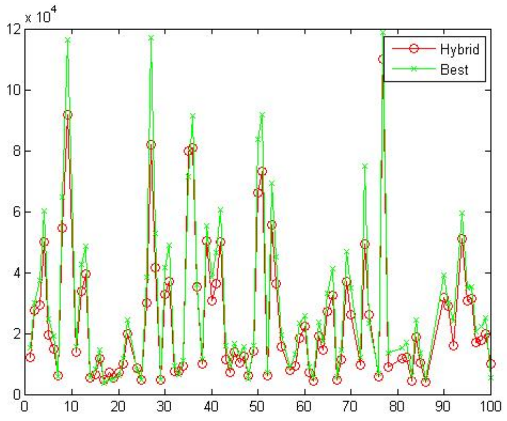


Figure 6.43: Absolute cost, 90% time - 10% memory, testing instance set

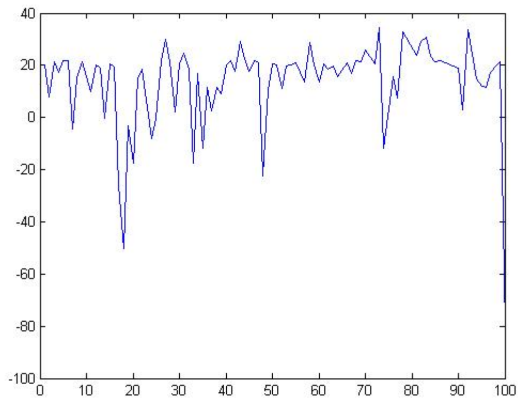


Figure 6.44: Percentage gain, 90% time - 10% memory, testing instance set

$$p_{10} = 0.9T + 0.1M$$

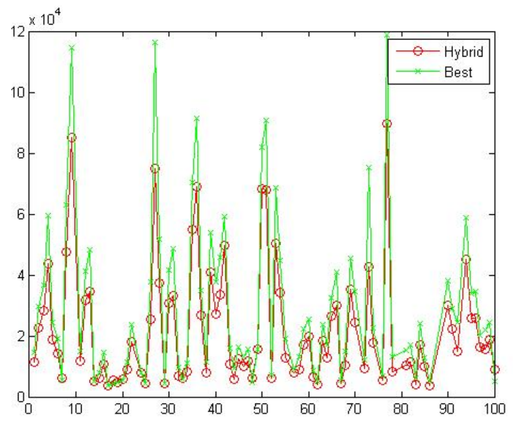


Figure 6.45: Absolute cost, 100% time - 0% memory, testing instance set

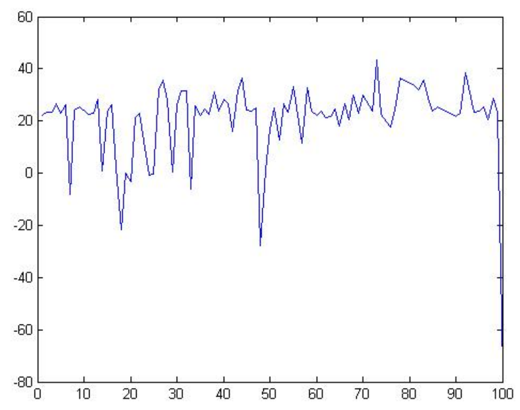


Figure 6.46: Percentage gain, 100% time - 0% memory, testing instance set

$$p_{11} = 1.0T + 0.0M$$

set, on all 11 penalty function definitions, and Figure 6.48 shows the percentage gain that was achieved on all 11 penalty function definitions.

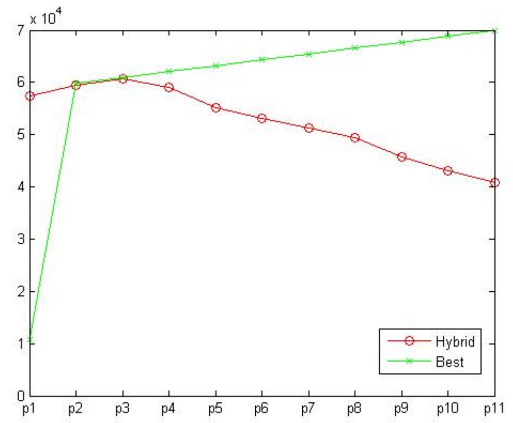


Figure 6.47: training instance set, average absolute cost

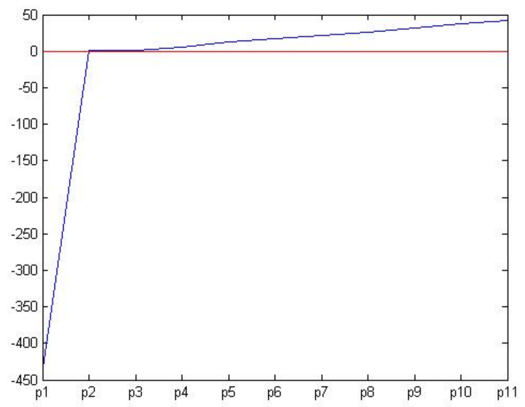


Figure 6.48: training instance set, average percentage gain

The next figure, Figure 6.49 shows the percentage gain again but zoomed on the last 10 penalty functions to better show the details of the curve.

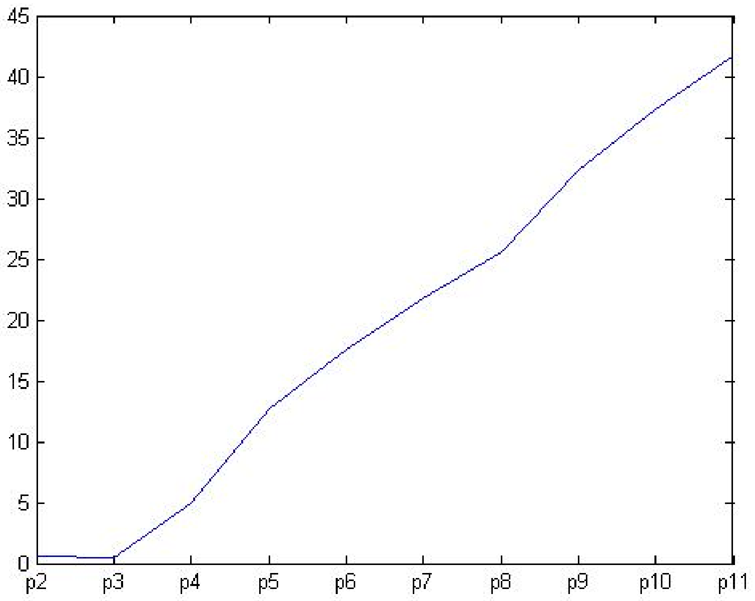


Figure 6.49: Training instance set, average percentage gain detail

These next two figures, Figure 6.50 and Figure 6.51 show the same concentrated average performance results but this time on the testing instance set. They show absolute penalty function values and percentage gain respectively.

The next figure, Figure 6.52 shows the percentage gain again but zoomed on the last 10 penalty functions to better show the details of the curve.

One last set of figures that is worth presenting, is the amount of each of the two algorithms that is included in the final hybrid algorithm, as a function of the penalty function definition. Figure 6.53 shows that as time is more important, A*'s contribution in the final hybrid grows and RBFS's becomes smaller. As memory becomes more important, A*'s contribution becomes smaller and RBFS's becomes larger. This was calculated based on the samples that were collected during the training phase. Figures 6.54 and 6.55 show the actual composition of the hybrid algorithm during the process of solving the instances.

This is exactly what we expected since it shows (apart from the fact that this is indeed a hybrid algorithm) that the learning process has correctly identified the

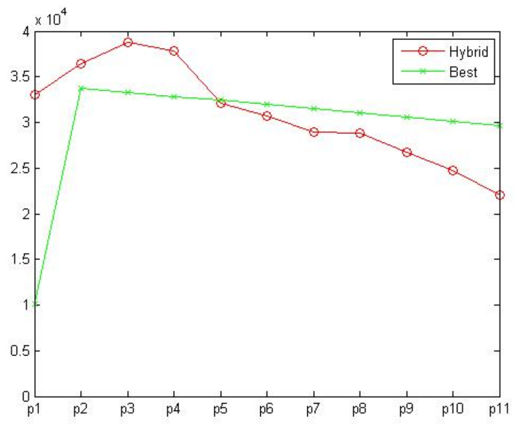


Figure 6.50: testing instance set, average absolute cost

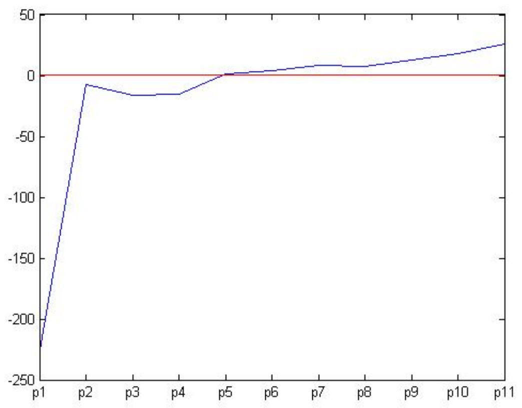


Figure 6.51: testing instance set, average percentage gain

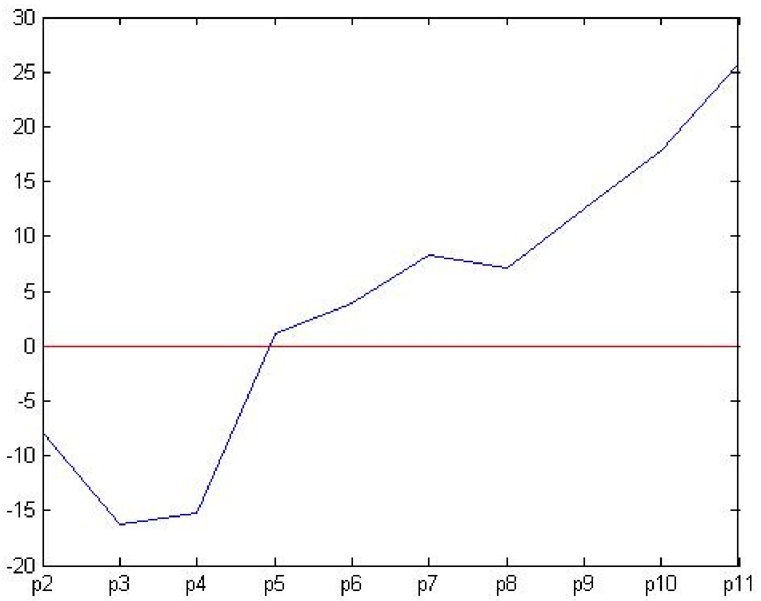


Figure 6.52: Testing instance set, average percentage gain detail

characteristics of each algorithm, performance-wise, and has successfully applied this new knowledge into the algorithm selection policy.

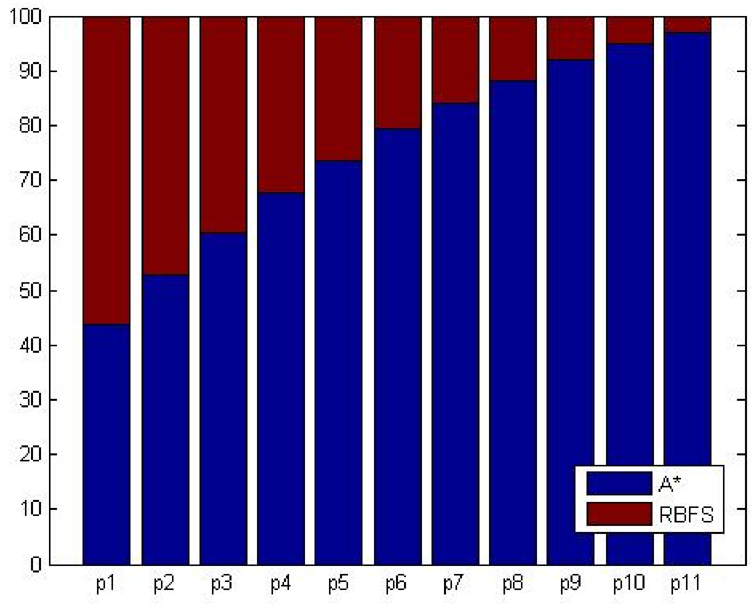
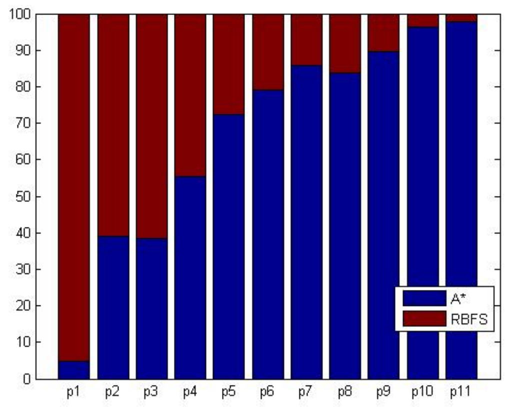


Figure 6.53: Hybrid algorithm composition over the Time-Memory spectrum



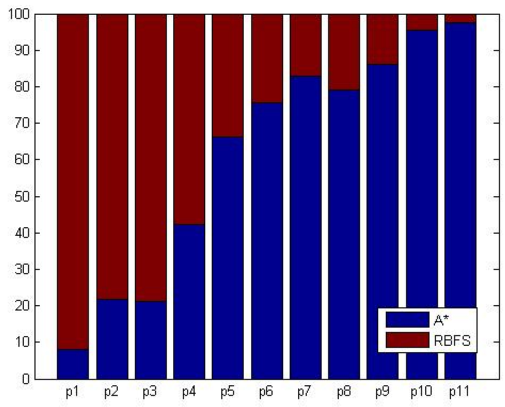


Figure 6.54: Training instance set, hybrid composition

Figure 6.55: Testing instance set, hybrid composition

Chapter 7

Discussion, Conclusions and Future Work

7.1 Importance of results and implications

We have shown that the hybrid algorithm created with Recursive Algorithm Selection and Reinforcement Learning can and indeed does out-perform the original algorithms from which it was created.

What is noteworthy is that the problem is solved not by one algorithm appropriately chosen, but rather a mix of the original algorithms. We see that as the definition of the penalty function changes, so does the composition of the hybrid algorithm, to shift focus in respect to the definition of the penalty function.

Also important is the fact that using a single sample collecting run we managed to produce as many policies as we liked. We chose to uniformly spread them across all of the memory-time spectrum, but there is no limitation as to how many redefinitions of the penalty function there can be. What is more, we managed to achieve positive results on completely different instances than the ones used for collecting the sample data. This is very important and implies that possibly, with a few well spread sample collecting runs, a much larger area of instances can be covered and still yield positive results.

However, there is still the issue of actual gain. We benchmarked our hybrid algorithm against the original algorithms, all within the same framework. However small, some performance overhead is still induced by the framework, when compared to a stand-alone implementation of the original algorithms. It is possible that the framework overhead can cancel out the performance gain that is achieved by the framework. But since this overhead is a one-time cost, it will be divided over the number of problems it manages to achieve some gain in, lessening its impact even more.

The same can be said for the off-line learning cost that LSPI introduces; Since it is a one-time cost, over a large number of uses its cost will become minimal and will be overcome by the average gain that can be achieved on each of the uses.

7.2 Future work

There are a few aspects of this work that can be extended, improved, or redone. In terms of experimentation, there is still a lot to be tested in terms of performance. Different problems, different algorithms, or even on the same problem can be tested, with different maps, different heuristics, different systems.

In terms of implementation, the way the application works is still a bit primitive. There is a lot of room for work in terms of interfacing with the end user. Slipstreaming the entire process of optimizing problem solving is probably the next step in this work.

7.3 Conclusions

Obviously the results were produced in sort of laboratory conditions, and not in real world conditions. However, we believe that as this work presents a relatively new approach in problem solving strategies, since the results are positive, it should provide some incentive for further research in this direction. As hardware optimization techniques reach a saturation point, and compile-time or run-time optimizations can only go so far, we believe that research should also return focus to algorithmic optimizations.

Having achieved a positive result in this research, we have proven that optimizations can still be achieved without necessarily upgrading to better hardware or a better compiler, or even operating system. Hopefully this will provide the necessary motive for more research in this area.

References

[1] Michail G. Lagoudakis and Ronald Parr, “Least-squares policy iteration,” *Journal of Machine Learning Research*, vol. 4, pp. 1107–1149, 2003. 3, 21

[2] TIGER/Line, “US Road Network Data.” `http://www.dis.uniroma1.it/~challenge9/data/tiger/`. 4

[3] Stuart Russel and Peter Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2 ed., 2003. 8, 10, 20

[4] P.E. Hart, N.J. Nilsson and Raphael B., “A formal basis for the heuristic determination of minimum cost paths,” *IEEE Transaction on Systems Science and Cybernetics*, vol. SSC-4(2), pp. 100–107, 1968. 10

[5] R.E. Korf, “Real-time heuristic search,” *Artificial Intelligence*, vol. 42(3), pp. 189–212, 1990. 13

[6] Leslie Pack Kaelbling, Michael Littman and Andrew Moore, “Reinforcement learning: A survey,” *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996. 20

[7] Richard Sutton and Andrew Barto, *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts: The MIT Press, 1998. 20

[8] Christopher J. C. H. Watkins, *Learning from Delayed Rewards*. PhD thesis, Cambridge University, 1989. 21

[9] J. R. Rice, “The algorithm selection problem,” *Advances in Computers*, vol. 15, pp. 65–118, 1976. 27

[10] B. A. Huberman, R.M. Lukose and T. Hogg, “An economic approach to hard computational problems,” *Science*, vol. 275, pp. 51–54, 1996. 27

[11] L. Xu, F. Hutter, H. H. Hoos and K. Leyton-Brown, “SATzilla-07: The Design and Analysis of an Algorithm Portfolio for SAT,” in *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP-07)*, pp. 712–727, 2007. 27

[12] Carla P. Gomes, Bart Selman, Nuno Crato and Henry Kautz, “Heavy-tailed phenomena in satisfiability and constraint satisfaction problems,” *J. Autom. Reason.*, vol. 24(1-2), pp. 67–100, 2000. 27

[13] C. P. Gomes and B. Selman, “Algorithm portfolios,” *Artificial Intelligence*, vol. 126(1-2), pp. 43–62, 2001. 27

[14] M. G. Lagoudakis and M. L. Littman, “Learning to select branching rules in the DPLL procedure for satisfiability,” *LICS/SAT*, vol. 9, pp. 344–359, 2001. 27

[15] M. G. Lagoudakis, M. L. Littman, and R. Parr, “Selecting the right algorithm,” in *Proceedings of the 2001 AAAI Fall Symposium Series: Using Uncertainty within Computation* (C. Gomes and T. Walsh, eds.), (Cape Cod, MA), November 2001. 27

[16] Lagoudakis M. and Littman M., “Algorithm Selection using Reinforcement Learning,” in *Proceedings of the 17th International Conference on Machine Learning (ICML-00)*, (Stanford, CA, USA), pp. 511–518, 2000. 27, 33

[17] Eclipse Foundation, “Eclipse IDE.” `http://www.eclipse.org/`. 43

[18] GNU Project, “GNU Compiler Collection.” `http://gcc.gnu.org/`. 43

[19] GNU Project, “GNU Debugger.” `http://www.gnu.org/software/gdb/`. 44

[20] GNU Project, “Data Display Debugger.” `http://www.gnu.org/software/ddd/`. 44

[21] The Valgrind Developers, “Valgrind.” `http://valgrind.org/`. 44