TECHNICAL UNIVERSITY OF CRETE

SCHOOL OF ELECTRICAL & COMPUTER ENGINEERING

Information & Networks Laboratory

‘Content Caching in Cellular Networks: An Algorithmic Comparison’

Michail Motos

Thesis Committee:

Professor Michael Paterakis, Supervisor

Professor Athanasios Liavas

Assoc. Professor Antonios Deligiannakis

A thesis submitted to the Technical University of Crete in partial fulfillment of the requirements for the degree of Diploma in Electrical and Computer Engineering

September 2019

Acknowledgements

First and foremost, I would like to express my deep appreciation towards my supervisor, Professor Michail Paterakis who gave me the opportunity to study a topic that I found really interesting, always pointing me to the right direction with his assistance and research throughout the duration of my thesis. I would also like to also thank the two members of the committee, Prof. Athanasios Liavas and Assoc. Prof. Antonios Deligiannakis for taking the time and effort to examine my thesis.

I cannot neglect to thank my family and close friends who were beside me all those years. I wish to thank my mother, Marina Marinaki and my father, Ioannis Motos, for giving me the chance to pursue a degree away from home. I want to thank them for their unconditional support and unceasing trust they have shown throughout all those years. Without them, I would never be able to complete my studies. I'd also like to thank my friends Costas, Dimitris and Michalis, we were always there for each other, and experienced both great and bad moments.

Finally, I would like to thank my lyceum math teacher, George, for pushing and guiding me towards a degree that I ended up loving.

Abstract

The ever-increasing content demand along with an increase in content size has begged the question to explore efficient caching algorithms in order to address the need for lag free content delivery and reduced costs and backhaul load. First, we consider the caching placement policy as a 0-1 Knapsack problem and then proceed to introduce the system model. Next, we comprehensively formulate the 0-1 Knapsack problem, and describe two algorithmic solutions, a simple and fast greedy algorithm, which is not optimal, and an optimal dynamic programming one. We also describe in detail the simulation model and the pertinent distributions we ran our simulations with. The results indicate that no matter the type of the weight distribution (weight here corresponds to the length of the content items) or its characteristics, the greedy algorithm manages to closely match the results of the dynamic programming one, while at the same time offering greatly reduced runtime complexity. On this basis, the simple and fast greedy algorithm considered is a very good choice as a caching policy, since content provision services currently offer large catalogues, where dynamic programming exhibits exorbitant running times.

Contents

List of Figures

Chapter 1

Introduction

Network technological advances in recent years have sparked interest in caching content offered by provision services. The ever-increasing demand for content in today's networks, combined with the constant growth of content size consumed by the users, demands the use of efficient and effective algorithms in order to maximize the cache hit rate. At the same time, content providers drive consumption using recommendation solutions that aim to satisfy the user as much as possible, by recommending appealing content. This comes in contrast with a typical caching scenario, since individual recommendations may differ from cached content that aims to satisfy the maximum demand aggregated over all users. This work focuses on the caching solutions required in such systems, without taking into account the fusion with recommendation systems.

According to Netflix, over 80% [1] of views come from algorithmic recommendations, while on YouTube 30% [2] of overall views come from related videos. Moreover, according to CISCO [3], globally, IP video traffic will be 82 percent of all IP traffic (both business and consumer) by 2022, up from 75 percent in 2017. According to the same source, Content Delivery Networks (CDNs) will carry 72 percent of Internet traffic by 2022 up from 56 percent in 2017. In 3G and 4G LTE networks caching has been shown to be able to reduce mobile traffic by one third to two thirds. These statistics alone, along with the oncoming arrival of 5G networks, demand a closer look on the caches required to satisfy the explosivity in network traffic demands. The usage of caches in Networks has lots of benefits, both for the end user and the provider. The user enjoys Quality of Experience (QoE) improvements, while the provider gains reduced network load.

In its simplest form, the problem to determine which content will be cached or not, can be formulated as a Knapsack Problem. The name Knapsack problem is derived from research done by mathematician Tobias Dantzig, referring on the problem of which are the most important items that should be packed without overloading the luggage. Dating as far back as 1897, earliest works about this problem date back more than a century.

The purpose of this work is to provide sufficient data in order to assess and compare algorithmic solutions to the Knapsack problem. The idea is that through this work we will be able to determine which algorithmic solutions are sufficient as caching protocols, both in terms of solution optimality and complexity.

1.1 Introduction to Knapsack Problem

The Knapsack problem is an NP combinatorial optimization problem. There are many variations of this problem, with most of them differentiating a certain aspect of the problem, such as the number of Knapsacks, the number of items or the number of objectives. In this particular case, we are going to formulate and examine the most common variation of such a problem, called the 0-1 Knapsack problem. Our goal is to simulate a network caching scenario, aiming to cache content that maximizes satisfaction of user demand. This is achieved by using various algorithms in order to maximize the cache hit ratio. Therefore, a need arises to determine which content from the catalogue will be stored in the cache, given a maximum cache size. We assume a catalogue with n contents, and a probability p_j for each item $j \in \{1, \dots, n\}$ of our catalogue to get picked. The probability distribution assumes values in [0,1], so that $\sum_{j=1}^{n} p_j = 1$. We assume W to be the maximum cache size, which will be a fraction of the overall size of the catalogue. Each item $j \in n$ has weight equal to w_j. Using the data described above as input, we calculate the Cache Hit Ratio (CHR) achieved by various algorithms and compare them. The abovementioned tests are done using various distribution types as input, to cover a large number of use cases and provide the corresponding results.

1.2 Introduction to the system model & pertinent distributions

In order to thoroughly examine the problem, we need to conduct exhaustive tests with various types of distributions and cache sizes as input. We assume that the popularity of each of the n items follows a Zipf-like distribution [4], with (s,V) being the distribution input parameters.

More specifically, V denotes the total number of items, which in our case is equal to n, while s defines the degree of skew. Every item j, $j \in \{1, \dots, n\}$ has a probability given by $p_j = \frac{c}{x^{1-s}}, where\ c = 1/\sum_{j=1}^{n} 1/x^{1-s}$ is a normalization constant. For s = 1 the distribution is uniform with no skew, while for s = 0 the distribution is highly skewed. For our testing purposes, we use a skew factor s equal to 0.2. We chose a Zipf-like distribution to simulate the item popularities, since web requests have been shown to be distributed according to such a model which also has been widely used in related works [9], [10].

To model the item length (or item weight) we used various types of distributions in order to get conclusive results about our caching scenario. The first distribution we used was a discrete uniform distribution. The item weights were distributed uniformly between a minimum and a maximum value a and b, respectively. The minimum value was set equal to 1(minimum length of an item) while the maximum value varied, in order to exhaustively test on a wide range of mean and variance distribution values. The second weight distribution we used was a geometric distribution.

We also performed simulation tests using the Pareto distribution. This distribution was originally created to describe the distribution of wealth in a society. It is a heavily skewed distribution, characterized by a heavy tail, defined by a shape factor α. However, compared to the distributions that were previously used, Pareto distribution is continuous. This presents a problem, since 0/1 Knapsack problem can only be solved using dynamic programming if the weight distribution is composed of integers. Therefore, we have converted the continuous Pareto distribution to a discrete one, for the purposes of these simulation tests. This is achieved by rounding the floating-point numbers to the closest integer. The pareto distribution's unique characteristic is that its variance does not converge for $a \leq 2$, while the mean value converges for $a > 1$. This distribution simulates an extreme scenario where a few items have very large lengths (or weights) while most of the items have small lengths.

Last but not least, in order to thoroughly examine the problem, we also need to perform our tests on a wide array of cache sizes. Firstly, we calculated the mean value of our item length distribution. As cache sizes for each test, we used a percentage of the mean value of the weight distribution multiplied by the number of items. Specifically, we performed tests using typical cache sizes of 1%, 2%, 5%, 10% and 20% of the average size of the n items.

1.3 Related Work

The Knapsack problem has been the subject of research for many centuries. In recent research [14], a new algorithm is proposed that reduces the running time from O(Tn), where T is total weight and n is the number of items, to O(TD), where D is the number of distinct weights. The algorithm in [14] implies a bound of $O(nM^2)$, without any dependence on V, or $O(nV^2)$, without any dependence on M, compared to the previously possible runtime of O(nMV), bounded by both M and V, where M is the maximum weight and V is the maximum value of any item. For the unbounded Knapsack problem, an additional algorithm running in time $O(M^2)$ or $O(V^2)$ is provided. Both their proposals match recent conditional lower bounds shown for the Knapsack problem. The bounded Knapsack algorithm essentially partitions the items into D sets according to their weights and solves the Knapsack problem in each set of the partition for every possible capacity up to T. This is done efficiently in O(T) time as all items in each set have the same weight and thus Knapsack can be greedily solved in those instances. Then, the overall solution is obtained by performing (max,+)-convolutions among them. Similarly, the algorithm for the unbounded Knapsack also uses (max,+) convolutions.

As far as network caching is concerned, it has been the subject of many studies in order to provide further insight and solutions to it. In [4], a novel cache management policy is introduced which is a combination of Least Frequently Used and Least Recently Used cache replacement policies. The simulations prove that it has a positive effect on hit ratio, while significantly reducing the fraction of user requests with delayed starts and the required CPU overhead. Additionally, the paper further introduces a collaborative environment of proxy servers that act in a decentralized way, meaning there is no centralized coordinator to organize the cached contents. This collaborative environment consists of a hierarchical tree topology of proxies which significantly improves the performance of the previously examined simple topology of non-collaborative proxies introduced in the first part of the work.

A lot of research lately is focused on the implementation of 5G network caching. In [11], caching techniques on such a network are explored. The authors first discuss how content is cached on current mobile networks and then proceed to explore caching techniques on 5G networks. They argue that caching in 3G and 4G LTE networks has been proven to be able to reduce mobile traffic by one third to two thirds. They propose various techniques, including evolved packet core network caching and radio access network caching. They found that both techniques can significantly reduce user-perceived latency as well as the transmission of redundant traffic over the network. A smoothening effect was also found on traffic spikes as well as a balancing of the backhaul traffic over a long period of time. Lastly, a content centric caching scenario is also explored in [11].

In [13], the authors claim that small cells heterogeneous architectures such as femtocells and WIFI off-loading can handle video traffic to nomadic users by short range links to the nearest small cell access points. As the density of the small cells increases, a system bottleneck is presented on the system backhaul. Therefore, they propose that small cells with low rate backhaul and high storage capacity cache popular video files. They show that optimum file assignment is NP-hard and provide a greedy strategy that can be used in order to approximate a solution to this problem.

Further research in [12] proposes a proactive caching scenario. To alleviate backhaul congestion, a proactive caching of files during off peak demands based on file popularity and correlations among users and files patterns is proposed. They also propose getting advantage of device to device (D2D) communications and social networks to proactively cache strategic contents and disseminate them to their social ties. These improvements show gains on backhaul savings that can be improved by increasing the storage capability at the network edge.

Apart from the typical caching scenarios, some research lately is focused on the balance between cached videos and individually tailored recommendation systems which the authors claim that can be applied in future 5G networks. For example, in [5] a new caching model is proposed. We are introduced to the “soft cache hit” which occurs if a user’s requested content is not in the local cache, but the user can be (partially) satisfied by a related content that is cached. The idea is that in case of a cache miss, we can satisfy the user with highly related substitute content that may provide similar satisfaction as the initially requested content. The paper argues that this can be activated during periods of predicted congestion or for selected users, in order to avoid expensive remote access.

1.4 Thesis Goal and Contribution

As we previously mentioned, the main goal of this work is the comparison between algorithms that can be used to decide caching policies. Our main goal was to compare the results of a simple and fast greedy algorithm, with the optimal solution that can be achieved using dynamic programming. The idea was to find how efficient and how close to the optimal solution can the above greedy solution be, for various kinds of different input distributions and cache sizes. This was done to provide insight on which algorithms are to be used in realistic caching scenarios.

1.5 Thesis Outline

In Chapter 2, we present and formulate the Knapsack problem. We also present the types of Knapsack problems and their various differences. In section 2.2 we analytically discuss dynamic programming, the algorithm used to solve the Knapsack problem as well as provide an example of a simple Knapsack problem to further demonstrate our point. We also discuss the space and time complexity of such an algorithm. In section 2.3 we present the greedy algorithm and provide a detailed explanation of the algorithmic steps. Furthermore, we provide an example in order to further facilitate the comprehension of this method. Last but not least, we analyze the space and time complexity of the algorithm.

In Chapter 3, we present the simulation model and characteristics of pertinent distributions. In section 3.2 we comprehensively discuss the popularity Zipf distribution, its characteristics, and the way it was constructed. In section 3.3, we present the weight distributions used, namely geometric, discrete uniform and discrete pareto. We discuss their theoretical characteristics, provide sample characteristics and assess their accuracy. Finally, we explain the method used to construct the distributions.

In Chapter 4, we present the results of our work. More specifically, in section 4.2 we discuss the simulation results for the geometric weight distributions. Similarly, in section 4.3 we discuss the simulation results for the uniform weight distributions and finally, in section 4.4 we discuss the simulation results for the pareto weight distributions. In section 4.5 we compare our results and draw conclusions. Finally, in section 4.6 we present ideas on how we can expand this work in the future.

Chapter 2

0-1 Knapsack Problem

2.1 Mathematical Formulation

The 0-1 Knapsack problem can be defined as follows: Given a set of n items, j = 1, …, n, each characterized by a weight w_j and a probability p_j to get picked by the user, we must select a subset of these items in order to maximize the cumulative probability, without surpassing the maximum weight capacity W. We define W as the maximum weight capacity, which refers to maximum content length that can be cached.

Given the above, the problem can be formulated as follows:

$$max\ z = \sum_{j=1}^{n} p_j x_j$$

subject to:

$$\sum_{j=1}^{n} w_j x_j \leq W$$

where x_j is a binary variable that defines whether item j is part of the solution. If it belongs in the Knapsack, then x_j is 1, otherwise it is equal to 0. Therefore, we define:

$$x_j = \begin{cases} 0, & if\ j\ belongs\ in\ the\ knapsack \\ 1, & otherwise \end{cases}$$

We also assume that the following condition holds:

$$\sum_{j=1}^{n} w_j > W$$

Therefore, not all items can fit in the Knapsack. We further assume that w_j, W are positive integers. On the other hand, the probability distribution assumes values in [0,1], so that $\sum_{j=1}^{n} p_j = 1$. Therefore, p_j is a floating-point number.

We can also formulate a Knapsack problem using min instead of max by substituting utility for cost. However, in this work we will only study maximization scenarios, since our goal is to evaluate the caching protocol performance.

0-1 Knapsack problem can be split into two variations. First, the unbounded Knapsack problem (UKP), which places no limit on the number of copies of each item. Second, the bounded Knapsack problem (BKP), the one we are interested in, which places the restriction that there can be only one copy of each item in the Knapsack.

2.2 Dynamic Programming Algorithm

Richard Bellman pioneered dynamic programming in 1950s, creating an optimal method to be used in multistage decision problems. According to the man himself, he chose the name dynamic because it sounded impressive. Interestingly enough, "programming" refers to the tabulation of intermediary results and not in computer programming. The basis of dynamic programming is to essentially break a complex problem into smaller, easier to solve sub-problems. Then, after solving the sub-problems recursively, beginning from the smallest ones first, and storing their result using memorization, we use the sub problems solutions in order to solve the more complex ones. The idea is that if we store the sub problems solutions, we need only calculate each solution once, meaning that these solutions can be reused to solve more complex problems.

We can summarize that a problem has to have two main properties in order to be solved using dynamic programming. The first property is called optimal substructure. A problem has this property when the optimal solution contains optimal solutions of its subproblems. The second property is overlapping subproblems, which entails that the solutions of the subproblems we memorize will be needed in the solving of higher up problems, in order to construct the solution.

2.2.1 Algorithm & mathematical formulation

The algorithm can be broken down as follows:

First of all, we construct a 2-dimensional array of size (n+1) x (W+1). Each (j,w) cell of the array represents a subproblem and will contain the optimal solution up to that point. After we successfully calculate every cell, the optimal solution to the problem can be found in the cell (n,W), which will be the bottom rightmost cell of our array.

Secondly, for every w in our array with j = 0, we set the array cells (0, w) = 0. Then, for every j in our array with w = 0, we set the array cells (j, 0) = 0. Therefore, our array should look as follows:

array(j,w)	w = 0	w = 1	…	…	w = W
j = 1	0	0	0	0	0
j = 2	0				
…	0				
j = n	0				

After we have successfully defined the initial values, we can start populating the Knapsack. We will calculate the remaining cell values in order to solve the problem. The order in which we will populate the array is line by line, from left to right. Therefore, the next step is for every item j and for every weight to calculate the cell weight using the following formula:

If the weight of item j, w_j, is greater than w, then the value of the cell (j,w) is equal to the value of the cell (j-1,w). This means that if the current item does not fit into the Knapsack yet, we use the previously calculated values, which were the optimal solutions up to that point.

On the other hand, if the weight of item j, w_j, is less than or equal to w, then the value of the cell (j,w) is equal to $max\{\, knapsack(j-1,w), p_j + knapsack(j-1, w-w_j)\}$. This means that we have two choices, we either place the item in the Knapsack or not. If we do not pick the current item as part of the solution, we pick the previously calculated optimal since it is better compared to the current item we are currently trying to fit. If we pick the current item as part of the solution, we calculate the new probability as our current item probability summed to the optimal utility we have previously calculated for the remaining w – w_j weight.

After we successfully evaluate every cell of our array, the optimal solution after having considered for every item if it should be placed in the Knapsack or not, is the (n,W) cell of our array. This returns the maximum utility we gain through our solution; however, it does not contain the items that are part of the optimal solution.

Taking into consideration the steps above, the dynamic programming algorithm for 0-1 Knapsack can be mathematically formulated as follows:

$$knapsack(j,w) = \begin{cases} 0, & if\ j = 0\ or\ w_j = 0 \\ knapsack(j-1,w), & if\ w_j > w \\ max\{\, knapsack(j-1,w), p_j + knapsack(j-1, w-w_j)\}, & otherwise \end{cases}$$

The algorithm described above possesses the two properties we first discussed that a problem must meet in order to be solvable by dynamic programming. First of all, it possesses the optimal substructure property, since the final optimal solution contains optimally solved subproblems. Last, but not least, we are constantly reusing memorized solutions of subproblems in our recursions. Therefore, it also possesses the overlapping subproblem property.

2.2.2 Example

Below, we are presenting a simple example to facilitate the understanding of the algorithm we described above. Let's suppose we want to place the following videos in a network cache, with a maximum weight limit equal to 3:

Item	1	2	3
Probability	0.1	0.2	0.7
Weight	3	2	3

First, we are going to construct the Knapsack array, filling it with zeros, as described above.

	w = 0	w = 1	w = 2	w = 3
j = 0	0	0	0	0
Item 1(j = 1)	0			
Item 2(j = 2)	0			
Item 3(j = 3)	0			

Afterwards, we start filling the array from left to right, line by line. Therefore:

j = 1:

Array (1,1): Weight of item 1 is greater than current weight limit (w = 1), therefore this is case two of the mathematical formulation. So, $knapsack(1,1) = knapsack(j-1, w) = knapsack(0,1) = 0$.

Array (1,2): The same applies for Array (1,2), since the weight limit is less than the item weight.

Array (1,3): However, for cell (1,3), the item fits in the Knapsack, so this is case 3 of our mathematical formulation. For that reason:

$$\begin{aligned} knapsack(1,3) &= max\{\, knapsack(j-1, w), p_j + knapsack(j-1, w-w_j)\} \\ &= max\{0\,, 0.1 + 0\} = 0.1 \end{aligned}$$

j = 2:

Array (2,1): Weight of item 2 is equal to 2, greater than the weight limit. Consequently, $knapsack(2,1) = knapsack(j-1, w) = knapsack(1,1) = 0$.

Array (2,2): Weight of item is equal to the weight limit. For that reason,

$$knapsack(2,2) = max\{ knapsack(j-1, w), p_j + knapsack(j-1, w-w_j)\}$$
$$= max\{0 , 0.2 + 0\} = 0.2$$

Array (2,3): Weight of item is less than the weight limit. Therefore,

$$knapsack(2,3) = max\{ knapsack(j-1, w), p_j + knapsack(j-1, w-w_j)\}$$
$$= max\{0.1 , 0.2 + 0\} = 0.2$$

j = 3:

Array (3,1): Weight of item is greater than the weight limit, ergo

$knapsack(3,1) = knapsack(j-1, w) = knapsack(2,1) = 0$

Array (3,2): As is the case above, weight of item is greater than the weight limit, consequently: $knapsack(3,2) = knapsack(j-1, w) = knapsack(2,2) = 0.2$

Array (3,3): Weight of item is equal to weight limit, therefore:

$$knapsack(3,3) = max\{ knapsack(j-1, w), p_j + knapsack(j-1, w-w_j)\}$$
$$= max\{0.2 , 0.7 + 0\} = 0.7$$

Finally, after we have calculated every cell value, the array is:

	w = 0	w = 1	w = 2	w = 3
j = 0	0	0	0	0
Item 1(j = 1)	0	0	0	0.1
Item 2(j = 2)	0	0	0.2	0.2

Item 3(j = 3)	0	0	0.2	0.7

The solution of the Knapsack problem is located in the bottom rightmost cell and in this case, it is equal to 0.7.

2.2.3 Recovering Items Contained in the Knapsack

Following the algorithmic steps described above, we calculate the value of the optimal solution, but not the actual items contained in it. In order to recover the items that are actually contained in the Knapsack we follow the method described below:

After successfully calculating all the values of our array and solved the Knapsack, we can develop a backtracking algorithm that constructs the solution. We start from the bottom rightmost cell, in this case (j = 3, w = 3), which is the result of the Knapsack. We compare it to the cell directly above. If the values are the same, then the item of this row is not included in the Knapsack. On the other hand, if the value changes, as in this case, the item is contained in the Knapsack. If an item is included in the Knapsack, we subtract its weight from the current weight and go vertically up one row and left according to its weight. We repeat this method until the weight reaches zero or until we reach the starting row.

	w = 0	w = 1	w = 2	w = 3
j = 0	0	0	0	0
Item 1(j = 1)	0	0	0	0.1
Item 2(j = 2)	0	0	0.2	0.2
Item 3(j = 3)	0	0	0.2	0.7

In our specific case, we compare 0.7 to 0.2. Since the values are different item 3 is included in the Knapsack. In order to find the next item, we go upwards one line and left, subtracting the weight of item 3. Since the weight of item 3 is 3, the next cell is (j = 2, w = 0). The remaining weight is equal to 0, therefore no other item fits the Knapsack and the optimal solution is item 3 with value 0.7.

2.2.4 Complexity

On the surface, the dynamic programming algorithm described above seems like it has a polynomial O(n*W) time complexity, since we construct an array of size n x W, with each cell having an O (1) cost to compute. However, taking a closer look, this is not entirely the case. The database size input n is polynomial in the length of the input of the problem, since n is the length of the array of the problem. On the other hand, the Knapsack input capacity W is just a number. The input size in this case is not W, but the number of bits that represent this number, which are equal to $\log(W)$ and not W itself. Therefore, the complexity is in reality accurately described as $O(n * 2^{logW})$. Hence, the algorithm runs in pseudo-polynomial time, ergo its running time is polynomial in the numeric value of the input, but exponential in the length of the input.

This can be made easier to understand with an example: Let's assume we have a cache capacity W = 2. To represent this number, we only need 2 bits. If we double the cache capacity to 4, we need 3 bits in order to represent this number in binary. So, the bits to represent to number increased by 1, but the complexity doubled, since it increased from $O(n * 2^2)$ to $O(n * 2^3)$. As a result, time complexity is exponential with respect to Knapsack capacity, leading to rapidly increasing runtimes for large cache sizes.

The space complexity is equal to O(n*W), since we construct an array of n * W entries. However, we can reduce this complexity, by making a simple observation looking at the solution algorithm. When solving the problem, in order to compute the cells for the next line, we only need the values of the current line cells and not of the entire array. Therefore, we reduce the space complexity to O(W), drastically decreasing memory requirements and space optimizing our solution.

2.3 Greedy Algorithm

When attempting to solve a problem using a greedy algorithm, we construct the problem solution in stages. In each stage, we pick the solution that looks best and by combining all sub-solutions, we construct the final one. This means that we make a locally optimum choice (the choice that is best at the moment) in the hope that this choice will lead us close to a global optimum solution, or rather the global optimum itself. Greedy algorithms usually don't lead to optimal solutions, but when they do, they are most likely the most efficient algorithms available, due to their simplicity.

The first step of the algorithm is to define a greedy heuristic in order to rank the items. In this case, we calculate the probability to pick an item over item weight.

$$heuristic = \frac{p_j}{w_j}$$

The next step is to sort the calculated ratios of our item database, from highest to lowest. Last but not least, we fill the Knapsack trying to fit as many items as possible, picking the ones that provide the highest ratio first. This goes on until the remaining unused weight is 0 or until we exhaust the list of n items. If an item does not fit in the Knapsack, we continue onto the next item with the next highest heuristic ratio.

2.3.2 Example

Below, we are going to show a simple example to further demonstrate the algorithm we described above. Let's suppose we want to place the following videos in a network cache, with a maximum weight limit equal to 5:

Item	1	2	3
Probability	0.1	0.2	0.7
Weight	3	2	3

We first calculate the Probability/Weight heuristic of each item, which yields the following results:

Item	1	2	3
Probability/Weight	0.033333333	0.1	0.233333333

Using the above results as input, we sort them by the heuristic ratio and try to fill the Knapsack, respecting the imposed weight limit. It is evident that we are going to pick the 3^{rd} item first, which will yield a probability of 0.7 and reduce the weight limit from 5 to 2. Continuing the execution of our algorithm, we check whether the 2^{nd} item fits in the Knapsack, since it has the 2^{nd} highest ratio. The item successfully fits into the Knapsack and we now have a cumulative probability of 0.9, consisting of the 2^{nd} and 3^{rd} item. However, after subtracting the weights of these items from the weight limit, we are left with zero remaining empty space in our cache. Therefore, we terminate the execution of our algorithm and conclude that the greedy algorithm solution yields a probability of 0.9, consisting of the 3^{rd} and 2^{nd} item. In this case, the solution found by the greedy algorithm is also the optimal one. This claim can be validated if we run the dynamic algorithm described above in this case.

2.3.3 Complexity

From the algorithm described above, we can conclude that the algorithm is split into two parts. The first part is the calculation and sorting of the heuristic ratios while the second part is the placement of the items in the Knapsack. In order to calculate and sort the heuristic ratios, we will use the most efficient sorting algorithm, which is Merge Sort. Merge Sort has an average, worst- and best-case complexity of O(nlogn), while the placement of the items in the Knapsack has O(n) complexity. Therefore, we can conclude that the overall time complexity of the algorithm is O(nlogn), which is a lot more efficient that the dynamic programming time complexity equivalent. The space complexity of the algorithm is O(n), since we need only create an array of size n in order to store the heuristic ratios.

Chapter 3

Simulation Model Characteristics

3.1 Introduction

Our aim in this and in the next chapter, is to provide extensive and convincing data, in order to justify our simulator. First, we are going into detail about the distributions we used and the way they were simulated. Afterwards, we are going to present the characteristics of each simulated distribution and compare them to the corresponding expected theoretical ones in order to verify that the simulations of the distributions were accurate.

3.2 Popularity Distribution

As we have already stated in our introductory chapter, the utility distribution we are going to use is a Zipf-like distribution, which is supported by experimental evidence about web caching [9], [10]. Our Zipf distribution is generated using two inputs, s and V, which represent the degree of skew and total number of items, respectively. A skew factor equal to 1 results in a uniform distribution with no skew, while a skew factor equal to 0 in a highly skewed distribution instead. For our testing purposes, we use a Zipf-like distribution with a skew equal to 0.2. Additional tests were also performed using higher and lower skew values in order to observe how the cache hit ratio (CHR) changes depending on the skew of our Zipf distribution. Below we are going to provide a graph of such a distribution:

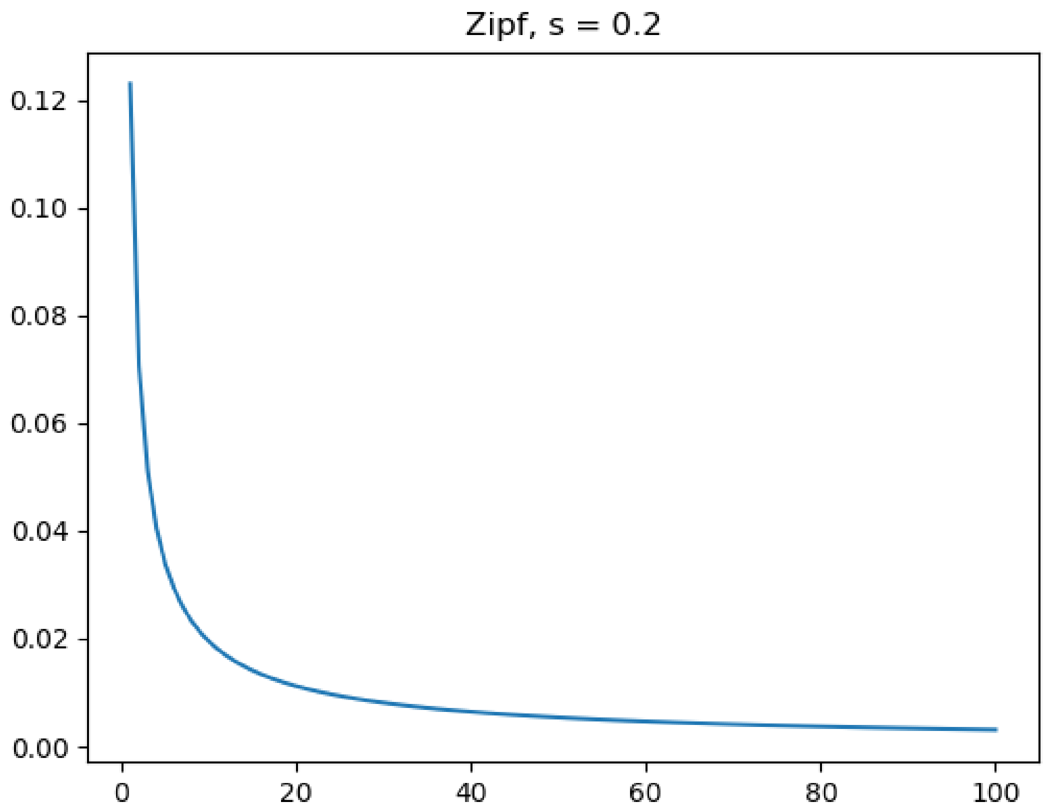

Figure 1 Zipf distribution example, s = 0.2

3.3 Weight Distributions

3.3.1 Geometric Distribution

The first distribution used in our experiments as an item length distribution was the geometric distribution. We simulate this distribution using the Inverse Transform Sampling technique. Given the Cumulative Distribution Function (CDF) of a distribution we can easily generate random variates, when the CDF is of such simple form that its inverse can be explicitly computed analytically. The first step is to compute the CDF of the desired random variable. In this case, the CDF of the geometric distribution is equal to $F(X) = 1 - (1 - p)^x$. Afterwards, we need to solve the equation F(X) = R, for X in terms of R where R is a random number in [0, 1]. For the geometric distribution, the result is $\lceil \ln(1-R)/\ln(1-p) \rceil$, where p is the success probability. This is our random variate generator for the geometric distribution. In order to compute the desired random variates, first we generate uniform random numbers in [0, 1] $R_1, \ldots, R_n$ and compute $X_j = F^{-1}(R_j)$. The resulting distribution is a sampled geometric distribution. The method results in the following formula:

$$x = \left\lceil \frac{\ln(1-R)}{\ln(1-p)} \right\rceil$$

In order to calculate the simulated distribution's characteristics, we calculate the sample mean and sample variance of the distribution and then proceed to compare them to the corresponding theoretical values, a way to confirm the validity of the simulated weight distribution. The formulas to calculate the sample mean and sample variance respectively are:

$$\mu = \frac{\sum_i x_i}{n}$$

$$\sigma^2 = \frac{\sum_i (x_i - \mu)^2}{n-1}$$

While the formulas used in order to calculate the expected mean and variance of the geometric distribution are respectively:

$$\mu = \frac{1}{p}$$

$$\sigma^2 = 1 - \frac{p}{p^2}$$

In order to verify that the geometric distribution is correctly simulated using the Inverse Transform Sampling method, we compare the expected mean and variance to the sample mean and variance values. We generated a distribution sample with size equal to 100.000.000 values and compared the sample mean and sample variance to the corresponding theoretical values, for various values of the parameter p:

p	Theoretical mean	Sample mean	Theoretical Variance	Sample Variance	Theoretical std
0.005	200	200.0255	39800	39818.9925	199.499
0.01	100	100.0121	9900	9900.0094	99.498
0.1	10	9.9983	90	89.9608	9.486
0.5	2	2	2	2.0004	1.414
0.8	1.25	1.2499	0.31	0.3122	0.556

Figure 2 Characteristics of the Geometric distributions

As we can see from the results in the above table, the theoretical expected and variance values are very close to the simulated ones, therefore our geometric distribution simulation is considered accurate.

3.3.2 Discrete Uniform Distribution

Due to the nature of the dynamic programming Knapsack algorithm, our weight distributions that characterize the item length, need to be discrete. The second type of such a distribution we considered, is a discrete uniform distribution, which distributes the item length uniformly between a minimum value and a maximum value.

A discrete uniform distribution, has a mean value equal to $\frac{k+1}{2}$, where k is the maximum value. The minimum value is always set to 1. In order to verify the accuracy of the simulated distribution, we need to compare the theoretical mean and variance to the corresponding sampled values. We simulated discrete uniform distributions that had the same mean values as the geometric distributions we used, in order to compare the obtained results. Each discrete uniform distribution below has been sampled 100.000.000 times, confirming that our discrete uniform distributions are simulated correctly:

Discrete Uniform	Theoretical Mean	Sample Mean	Theoretical Variance	Sample Variance	Theoretical std
[1,3]	2	2	0.66	0.66	0.812
[1,19]	10	10	30	30.001	5.47
[1,199]	100	99.99	3300	3300.4	57.44
[1,399]	200	199.99	13266.66	13266.72	115.18

Figure 3 Characteristics of the Uniform distributions

3.3.3 Discrete Pareto Distribution

The last distribution we used in our simulation results, was the Pareto distribution. We used this distribution because of its unique characteristic, namely that its variance does not converge for $a \leq 2$, while its mean value converges for $a > 1$. For the purposes of our simulations, we used the following a values: 1.01, 1.16 and 1.9. Since Pareto is a continuous distribution, in order to discretize it, we generated continuous Pareto distribution variates and then converted the floating-point values to integers. Below, we present the comparison table between sample mean and variance and the corresponding theoretical values.

In order to generate the distribution via inverse transform sampling, we firstly calculate the random variate of the distribution, which can be shown to be equal to $\lfloor 1/e^{lnR/a} \rfloor$, and then compute the variate function using uniform random numbers in [0, 1], R, as input.

Regarding the characteristics of the discrete pareto distribution, the mean value of the distribution can be shown to be equal to:

$$\mu = \sum_{x=1}^{\infty} \frac{1}{x^{\alpha}} \in \mathbb{R}$$

While variance can be shown to be equal to:

$$\sigma^2 = \sum_{x=1}^{\infty} \left[x^2 \left(\frac{1}{x^a} - \frac{1}{(x+1)^a} \right) \right] - \mu^2 = +\infty, 1 < \alpha < 2$$

The clarification for the statements above can be found in the appendix of this work.

# of Samples	Theoretical mean	Sample Mean	Theoretical Variance	Sample Variance
1000	100.5	10.3	∞	6208
5000	100.5	8.4	∞	3141
10000	100.5	9.1	∞	6140
25000	100.5	17.6	∞	1840110
50000	100.5	14	∞	941288
100000	100.5	12.8	∞	514279
1.000.000	100.5	11.5	∞	305533
5.000.000	100.5	13.6	∞	3135092
10.000.000	100.5	26.7	∞	813301707
20.000.000	100.5	23.8	∞	613004380
30.000.000	100.5	21.8	∞	429571444
40.000.000	100.5	19.9	∞	324628106
50.000.000	100.5	19.5	∞	271783321
60.000.000	100.5	21.2	∞	354476453
70.000.000	100.5	20.8	∞	30361269
80.000.000	100.5	21.1	∞	313951646
90.000.000	100.5	21.2	∞	305031987
100.000.000	100.5	21.1	∞	291861713

Figure 4 Characteristics of a Pareto distribution with a shape factor a = 1.01

# of Samples	Theoretical mean	Sample mean	Theoretical Variance	Sample Variance
1000	6.8387	19.8	∞	263198
5000	6.8387	8.1	∞	53466
10000	6.8387	7.1	∞	29849
25000	6.8387	31.3	∞	15683668
50000	6.8387	19.4	∞	7869944
100000	6.8387	12.5	∞	3937156
1.000.000	6.8387	6.9	∞	494689
5.000.000	6.8387	6.3	∞	175383
10.000.000	6.8387	6.8	∞	1558444
20.000.000	6.8387	6.5	∞	830229
30.000.000	6.8387	6.6	∞	812135
40.000.000	6.8387	6.6	∞	723635
50.000.000	6.8387	6.6	∞	598502
60.000.000	6.8387	6.8	∞	1675810
70.000.000	6.8387	6.8	∞	1611945
80.000.000	6.8387	6.8	∞	1425318
90.000.000	6.8387	6.7	∞	1277182
100.000.000	6.8387	6.8	∞	2383606

Figure 5 Characteristics of a Pareto distribution with a shape factor a = 1.16

# of Samples	Theoretical mean	Sample mean	Theoretical Variance	Sample Variance
1000	1.7497	2	∞	6.3
5000	1.7497	2.08	∞	32.4
10000	1.7497	2.05	∞	20.1
25000	1.7497	2.05	∞	17.1
50000	1.7497	2.05	∞	13.2
100000	1.7497	2.05	∞	11.5
1.000.000	1.7497	2.05	∞	13.4
5.000.000	1.7497	2.05	∞	58.5
10.000.000	1.7497	2.05	∞	41.7
20.000.000	1.7497	2.05	∞	30.6
30.000.000	1.7497	2.05	∞	35.3
40.000.000	1.7497	2.05	∞	32.1
50.000.000	1.7497	2.05	∞	36.1
60.000.000	1.7497	2.05	∞	35.3
70.000.000	1.7497	2.05	∞	34.5
80.000.000	1.7497	2.05	∞	32.5
90.000.000	1.7497	2.05	∞	31.5
100.000.000	1.7497	2.05	∞	31

Figure 6 Characteristics of a Pareto distribution with a shape factor a = 1.9

From the results in the above tables, we conclude that the accuracy of the Pareto simulated variates varies depending on the value of the shape factor of the distribution. Using Inverse Transform Sampling yields accurate results for a shape factor equal to 1.16, but the sample mean in the case of shape factor equal to 1.01 is way off the theoretical result. The same was also found to be true when simulating a continuous Pareto distribution with shape factor $\alpha = 1.01$ with the theoretical mean being equal to 101 and the sample mean being equal to 21.1. In the case when the shape factor is equal to 1.9, we notice a slight difference between sample mean and the theoretical mean results. However, in this case, the difference is not as great as when $\alpha = 1.01$.

Chapter 4

Simulation Results

4.1 Introduction

In this chapter, we present representative results of our simulations. In sections 4.2 and 4.3, we will showcase our results using a geometric and a discrete uniform distribution, respectively. Additionally, in section 4.4, we are going to present our results using the discrete Pareto distribution. Finally, for each type of distribution used, we are going to comment on the results and extract a conclusion about the performance of each caching algorithm.

4.2 Simulation Results for the Geometric Weight Distributions

A geometric distribution is characterized by its parameter p, which is the probability of success after each trial. For each p value, we simulated scenarios with database size equal to 1.000, 5.000, 10.000, 25.000 and 50.000 items.

We performed our tests using typical cache sizes equal to 1%, 2%, 5%, 10% & 20% of the average size of the n items in the database. For example, in our first simulation test, which used a n value equal to 1.000 and a success probability equal to 0.8 (which means that average item size is equal to 1.25), the cache size was calculated as follows:

$$cache\ size = percentage * average\ item\ size * n = percentage * 1.25 * 1000$$

In the tables below, the first column calculates the cache size as indicated by the formula above. The second and third column contain the greedy algorithm Cache Hit Ratio (CHR) and the average runtime in order to calculate the CHR respectively. The fourth and fifth columns contain the CHR and the average runtime of the dynamic programming algorithm. In order to make our results more conclusive, we run each simulation for each cache size 100 times, which is indicated in the sixth column. Therefore, we create 100 different distributions with the same values of the parameters n and p and calculate the CHR and average runtime of each algorithm. The last column, Total Time, indicates the total runtime of our simulation, including both the greedy and the dynamic programming algorithms. Simulations were run on a desktop computer equipped with a Ryzen 2600 3.9 GHz processor and 16GB DDR4 RAM. Last but not least, because the CHR results of the greedy and dynamic programming algorithm are similar, the decimal places that are different are highlighted in the table below.

The first p value we considered was equal to 0.8, which yielded the following results:

P	Theoretical mean	Sample mean	Theoretical variance	Sample Variance	Standard Deviation	Std./Mean
0.8	1.25	1.247	0.31	0.3112	0.55	0.44

Figure 7 Geometric distribution characteristics, p = 0.8

Cache Size	Greedy CHR	Greedy Avg. Runtime	Dynamic Pr. CHR	Dynamic Pr. Avg. Runtime	Total Time
$n = 1000$					
1% = 13	0.240762394024	0.0036s	0.2407**97417918**	0.0387s	4.6s
2% = 25	0.312432074743	0.0041s	0.3124**46653036**	0.0413s	4.97s
5% = 63	0.432839408563	0.005s	0.432839**522647**	0.0439s	5.27s
10% = 125	0.537771432596	0.0048s	0.537771**539726**	0.0427s	5.26s
20% = 250	0.660289487758	0.0043s	0.660289487758	0.0435s	5.17s
$n = 5000$					
1% = 63	0.291427568694	0.0255s	0.29142**8490676**	0.2099s	25.63s
2% = 125	0.362087986664	0.0262s	0.36208**8243534**	0.2307s	27.9s
5% = 313	0.473183079348	0.0253s	0.473183**109061**	0.2326s	27.85s
10% = 625	0.571624669434	0.0256s	0.5716246**84046**	0.2481s	29.3s
20% = 1250	0.685367152627	0.0234s	0.685367152**75**	0.2638s	30.51s
$n = 10000$					
1% = 125	0.307452839677	0.0495s	0.30745**3008726**	0.4391s	53s
2% = 250	0.377226762762	0.047s	0.377226**815851**	0.4734s	56.31s
5% = 630	0.485949126937	0.0492s	0.485949**141376**	0.5096s	59.88s
10% = 1300	0.582390777311	0.0444s	0.5823907**80421**	0.549s	63.31s
20% = 2600	0.693057353545	0.0414s	0.69305735**6098**	0.6193s	69.53s
$n = 25000$					
1% = 313	0.325941859492	0.1323s	0.325941**962778**	1.24s	148.2s
2% = 625	0.39380375708	0.1322s	0.3938037**62199**	1.37s	161.3s
5% = 1563	0.499570238087	0.125s	0.49957023**9647**	1.6s	183.1s
10% = 3125	0.593401859997	0.115s	0.5934018**60828**	1.8s	201.3s
20% = 6250	0.701286584023	0.1016s	0.70128658**428**	2.15s	233.4s
$n = 50000$					
1% = 625	0.336861311778	0.284s	0.3368613**33218**	2.82s	333.67s
2% = 1250	0.403802003101	0.2773s	0.40380200**662**	3.18s	367.37s

5% = 3125	0.507873027655	0.255s	0.507873028**018**	3.93s	438.9s
10% = 6250	0.600225561931	0.2176s	0.6002255619**95**	4.73s	512.18s
20% = 12500	0.706315431195	0.1886s	0.706315431**203**	6.1s	645.5s

Figure 8 Simulation Results for the geometric distribution, p = 0.8

P	Theoretical mean	Sample mean	Theoretical variance	Sample Variance	Standard Deviation	Sd./Mean
0.5	2	1.999	2	2.001	1.414	0.707

Figure 9 Geometric distribution characteristics, p = 0.5

Cache Size	Greedy CHR	Greedy Avg. Runtime	Dynamic Pr. CHR	Dynamic Pr. Avg. Runtime	Total Time
$n = \mathbf{1000}$					
1% = 20	0.251734667856	0.0033s	0.2517**63309412**	0.041s	4.94s
2% = 40	0.330490093859	0.0036s	0.3305**1003111**	0.0418s	4.96s
5% = 100	0.45345412494	0.005s	0.4534**61104583**	0.0412s	5.1s
10% = 200	0.562718152705	0.0037s	0.56271**9335424**	0.0448s	5.33s
20% = 400	0.688400741944	0.0043s	0.688401**305986**	0.0447s	5.37s
$n = \mathbf{5000}$					
1% = 100	0.305110132808	0.0254s	0.30511**289275**	0.2184s	26.47s
2% = 200	0.378284754177	0.0254s	0.37828**5683654**	0.2339s	27.96s
5% = 500	0.492537473172	0.0249s	0.492537**698264**	0.2427s	28.71s
10% = 1000	0.59434706407	0.0246s	0.594347**138388**	0.2615s	30.53s
20% = 2000	0.711467157913	0.0228s	0.711467**175155**	0.2935s	33.4s
$n = \mathbf{10000}$					
1% = 200	0.32166659209	0.0562s	0.32166**7689268**	0.4775s	57.57s
2% = 400	0.393836867252	0.0544s	0.393837**286686**	0.4955s	59.2s
5% = 1000	0.505631914279	0.0527s	0.5056319**65242**	0.5484s	64s
10% = 2000	0.604880213203	0.049s	0.60488023**5625**	0.6219s	70.66s
20% = 4000	0.718936135417	0.046s	0.7189361**39735**	0.7117s	79s
$n = \mathbf{25000}$					
1% = 500	0.339327433278	0.1401s	0.339327**827849**	1.2753s	152.27s
2% = 1000	0.409392441309	0.1405s	0.4093924**56198**	1.4528s	170s
5% = 2500	0.518343378428	0.1277s	0.5183433**87604**	1.8s	202.5s
10% = 5000	0.615139582178	0.1159s	0.61513958**4592**	2.118s	232s
20% = 10000	0.726326145848	0.1022s	0.7263261**46239**	2.6757s	284.76s
$n = \mathbf{50000}$					
1% = 1000	0.350397161792	0.3141s	0.350397**218827**	2.9857s	352.3s
2% = 2000	0.419315235384	0.2975s	0.4193152**4747**	3.6356s	415.2s
5% = 5000	0.526420812645	0.2787s	0.5264208**15555**	4.771s	524.1s
10% = 10000	0.621517703236	0.233s	0.621517703**927**	6s	645.72s
20% = 20000	0.730798690394	0.1915s	0.730798690**571**	8.1467s	847s

Figure 10 Simulation Results for the geometric distribution, p = 0.8

P	Theoretical mean	Sample mean	Theoretical variance	Sample Variance	Standard Deviation	Sd./Mean
0.1	10	10.049	90	90.733	9.48	0.948

Figure 11 Geometric distribution characteristics, p = 0.1

Cache Size	Greedy CHR	Greedy Avg. Runtime	Dynamic Pr. CHR	Dynamic Pr. Avg. Runtime	Total Time
$n = 1000$					
1% = 100	0.286421873914	0.005s	0.286**724775139**	0.044s	5.38s
2% = 200	0.36925492424	0.0057s	0.369**318798024**	0.0472s	5.73s
5% = 500	0.501073633927	0.0053s	0.501**107261928**	0.0494s	5.96s
10% = 1000	0.617845234087	0.0046s	0.617**923006199**	0.0537s	6.19s
20% = 2000	0.742570116683	0.0043s	0.742**639005058**	0.0599s	6.83s
$n = 5000$					
1% = 500	0.338976661219	0.0288s	0.3389**83649688**	0.2611s	31.1s
2% = 1000	0.417719562285	0.0275s	0.4177**21133715**	0.2903s	33.8s
5% = 2500	0.540412293443	0.0268s	0.540412**78977**	0.3572s	40.2s
10% = 5000	0.647367365631	0.0244s	0.6473**79142418**	0.4233s	46.4s
20% = 10000	0.762527885619	0.0212s	0.7625**36448972**	0.5318s	56.6s
$n = 10000$					
1% = 1000	0.354733627793	0.0637s	0.35473**5385478**	0.6091s	71.8s
2% = 2000	0.431484226879	0.0595s	0.43148**5268257**	0.747s	85s
5% = 5000	0.551264147762	0.0531s	0.551264**237961**	0.9537s	104.6s
10% = 10000	0.656177657346	0.0462s	0.6561**83729408**	1.2s	128.4s
20% = 20000	0.768535345244	0.0395s	0.7685**40630817**	1.6424s	170.81s
$n = 25000$					
1% = 2500	0.371710017953	0.1776s	0.371710**206339**	2.14s	242.6s
2% = 5000	0.446827797487	0.1674s	0.446827**891015**	2.74s	301.2s
5% = 12500	0.563468522409	0.1444s	0.56346854**094**	4.2s	440.9s
10% = 25000	0.66523191966	0.1138s	0.66523**3543674**	5.9s	608.2s
20% = 50000	0.774465281468	0.0938s	0.77446**6507894**	8.7s	882.9s
$n = 50000$					
1% = 5000	0.382528397614	0.4019s	0.382528**49171**	6s	668.4s
2% = 10000	0.456449160034	0.3833s	0.456449**176919**	9.2s	978.5s
5% = 25000	0.571056521843	0.3229s	0.5710565**26824**	18.4s	1894.4s
10% = 50000	0.67113154641	0.2377s	0.67113**2642082**	39s	3944.1s
20% = 100000	0.778493581297	0.1791s	0.77849**4283456**	54.8s	5513.9s

Figure 12 Simulation Results for the geometric distribution, p = 0.1

P	Theoretical mean	Sample mean	Theoretical variance	Sample Variance	Standard Deviation	Sd./Mean
0.01	100	100.447	9900	9975.4	99.5	0.995

Figure 13 Geometric distribution characteristics, p = 0.01

Cache Size	Greedy CHR	Greedy Avg. Runtime	Dynamic Pr. CHR	Dynamic Pr. Avg. Runtime	Total Time
$\boldsymbol{n = 1000}$					
1% = 1000	0.310036195099	0.0051s	0.310**69219168**	0.05s	6.7s
2% = 2000	0.395868857358	0.0052s	0.39**6060696283**	0.07s	8s
5% = 5000	0.525148506956	0.0048s	0.525**2435869**	0.09s	10.1s
10% = 10000	0.635846093289	0.0042s	0.635**947089828**	0.12s	13.2s
20% = 20000	0.75496770997	0.0038s	0.75**503502281**	0.1629s	17s
$\boldsymbol{n = 5000}$					
1% = 5000	0.359244534771	0.0335s	0.3592**64120474**	0.6s	66.4s
2% = 10000	0.439178466843	0.0317s	0.4391**96961772**	0.91s	96.7s
5% = 25000	0.560599875214	0.027s	0.560**617320984**	1.73s	177.2s
10% = 50000	0.663529923421	0.02s	0.6635**47487435**	3.9s	392.7s
20% = 10^5	0.774026578914	0.0144s	0.774**039157445**	6s	601s
$\boldsymbol{n = 10000}$					
1% = 10000	0.373718286094	0.0768s	0.3737**2848586**	2.1s	224.4s
2% = 20000	0.451987708091	0.0783s	0.4519**95684592**	3.6s	373.6s
5% = 50000	0.570695929598	0.0612s	0.570**703893505**	11.6s	1170.1s
10% = 100000	0.671320792679	0.0425s	0.6713**28495563**	28.9s	2897.1s
20% = 200000	0.778884564338	0.0334s	0.7788**9063817**	43.6s	4363.5s
$\boldsymbol{n = 25000}$					
1% = 25000	0.390915755531	0.2298s	0.39091**986456**	13.9s	1424.4s
2% = 50000	0.466935759252	0.2145s	0.46693**8752506**	38.3s	3860.9s
5% = 125000	0.582148747452	0.1624s	0.5821**52084204**	153.4s	15366.9s
10% = 250000	0.679893644681	0.1302s	0.6798**96300691**	268.7s	26894.4s
20% = 500000	0.784607823051	0.102s	0.7846**10329353**	383.8s	38397.6s
$\boldsymbol{n = 50000}$					
1% = 50000	0.40078486637	0.5331s	0.40078**6604214**	122.7s	12349.3s
2% = 100000	0.47581389408	0.4687s	0.47581**5346588**	300.4s	30110.9s
5% = 250000	0.589265367338	0.3811s	0.5892**66762514**	742s	74264s
10% = 500000	0.685460908869	0.315s	0.6854**62542428**	1213s	121348.6s
20% = 10^6	0.788342543067	0.2351s	0.7883**43675868**	1695.4s	169578.9s

Figure 14 Simulation Results for the geometric distribution, p = 0.01

P	Theoretical mean	Sample mean	Theoretical variance	Sample Variance	Standard Deviation	Sd./Mean
0.005	200	200.05	39800	39809.3	199.5	0.9975

Figure 15 Geometric distribution characteristics, p = 0.005

Cache Size	Greedy CHR	Greedy Avg. Runtime	Dynamic Pr. CHR	Dynamic Pr. Avg. Runtime	Total Time
$n = \mathbf{1000}$					
1% = 2000	0.316045517326	0.005s	0.316**376494778**	0.0777s	8.9s
2% = 4000	0.399568852119	0.0053s	0.399**710937234**	0.0975s	10.7s
5% = 10000	0.527257314079	0.0042s	0.527**383255581**	0.148s	15.6s
10% = 20000	0.636524522962	0.0038s	0.636**608155348**	0.1987s	20.6s
20% = 40000	0.754948266877	0.0035s	0.75**5030817885**	0.2808s	28.6s
$n = \mathbf{5000}$					
1% = 10000	0.361031043694	0.0351s	0.3610**54495031**	1.021s	108s
2% = 20000	0.440138850961	0.0331s	0.4401**57960864**	1.8796s	193.3s
5% = 50000	0.560673879303	0.0271s	0.5606**89802209**	6s	611.93s
10% = 100000	0.6636071204	0.0195s	0.6636**22051259**	14.7s	1479.64s
20% = 200000	0.773956337942	0.0153s	0.77396**734939**	22s	2199.5s
$n = \mathbf{10000}$					
1% = 20000	0.376234682526	0.0821s	0.3762**45416542**	4.3s	443.3s
2% = 40000	0.453791396883	0.0748s	0.4537**99970831**	11.6s	1173.6s
5% = 100000	0.571738199353	0.0594s	0.5717**45381972**	43.8s	4391.2s
10% = 200000	0.671998609391	0.0456s	0.67**2006457533**	81.4s	8150.6s
20% = 400000	0.779490224408	0.0366s	0.7794**95821275**	116.6s	11669.3s
$n = \mathbf{25000}$					
1% = 20000	0.392550854618	0.254s	0.39255**464038**	50.4s	5077.8s
2% = 40000	0.468387260837	0.2178s	0.4683**90158542**	154.2s	15458.3s
5% = 100000	0.583062633142	0.1827s	0.58306**5569533**	371.2s	37149.2s
10% = 200000	0.680482554553	0.141s	0.68048**4579301**	606.5s	60674.7s
20% = 400000	0.784964630293	0.1116s	0.78496**6629108**	847.3s	84751.8s
$n = \mathbf{50000}$					
1% = 100000	0.402266979772	0.5484s	0.40226**8493046**	360.4s	36115.3s
2% = 40000	0.477116071219	0.4925s	0.47711**7589971**	766s	76668.5s
5% = 100000	0.590213173913	0.3906s	0.59021**464342**	1616.3s	161688.3s
10% = 200000	0.686168070975	0.3094s	0.68616**9299989**	2553.6s	255406.9s
20% = 400000	0.788895159815	0.2411s	0.78889**6041158**	3521s	352131.3s

Figure 16 Simulation Results for the geometric distribution, p = 0.005

Below, we are providing a few graphs in order to further demonstrate the comparisons of the algorithmic results:

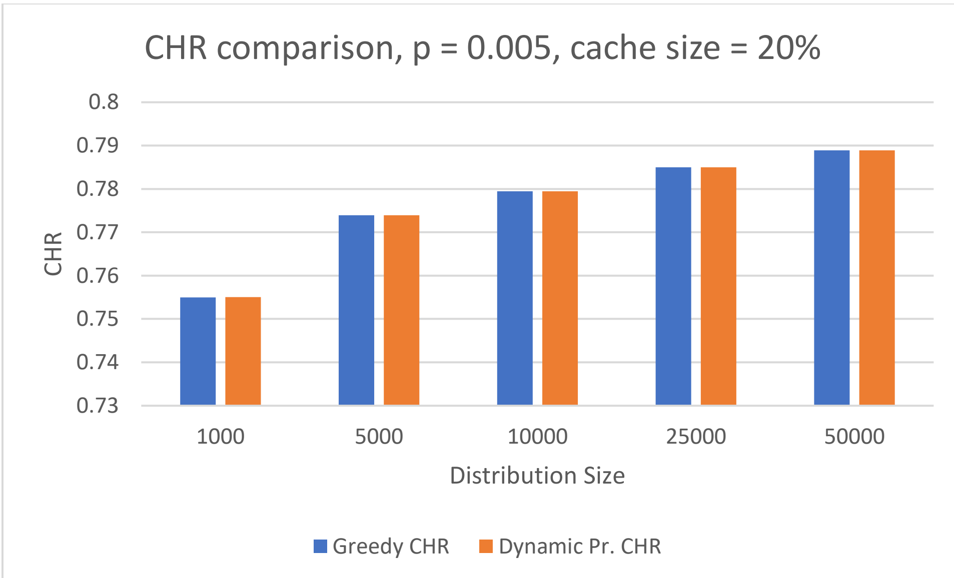

Figure 17 CHR comparison chart, p = 0.005, cache size = 20%

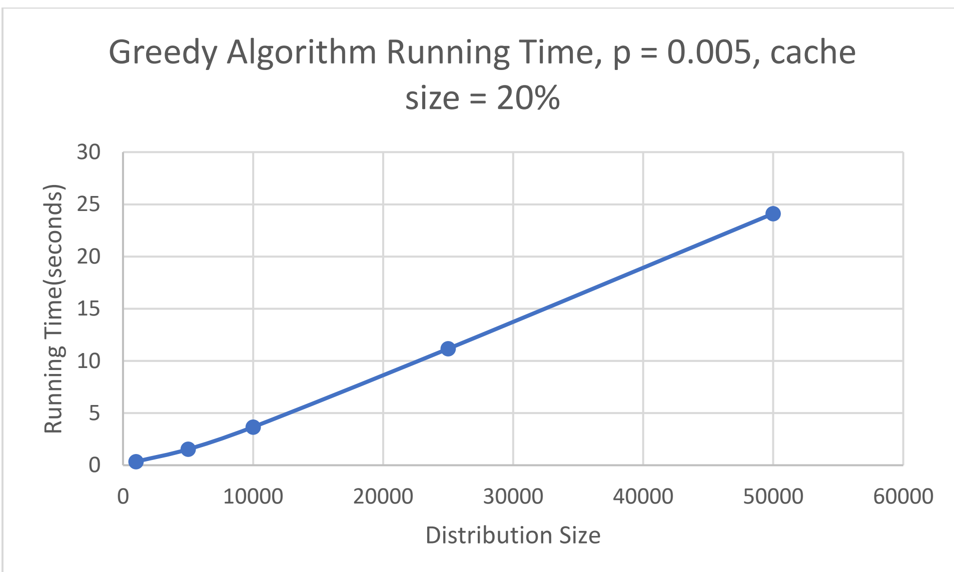

Figure 18 Greedy Algorithm Running Time, p = 0.005, cache size = 20%

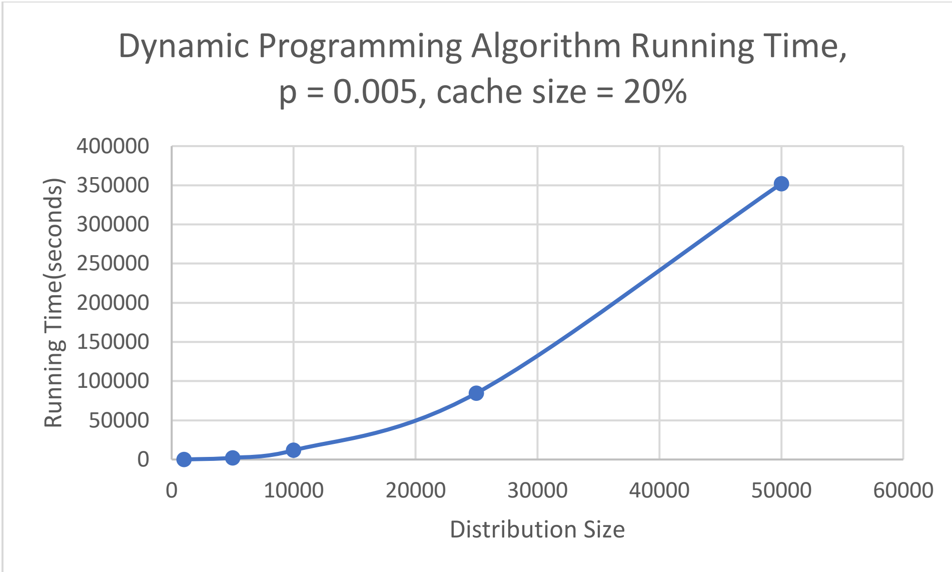

Figure 19 Dynamic Programming Algorithm Running Time, p = 0.005, cache size = 20%

The graphs provided show how comparable are the solutions to the problem produced by the greedy and the dynamic programming algorithms. No matter what the value of the success probability p or of the cache size, the Cache Hit Ratios are really very similar. Due to lack of space we do not provide the corresponding graphs for every case.

4.3 Simulation Results for the Uniform Weight Distributions

As in the case of the geometric distribution simulations, we performed our simulations by creating distributions for number of items in the database equal to 1.000, 5.000, 10.000 and 50.000. We decided to simulate discrete uniform distributions that have the same mean value as the simulated geometric distributions we used in the previous tests, in order to facilitate comparisons. Since our discrete uniform distributions take values over the integers in [1, k], an equivalent in mean value discrete uniform distribution to the geometric distribution with $p = 0.8$ could not be constructed (since the mean value of the latter is equal 1.25). Our simulations yield the following results:

Discrete Uniform	Theoretical Mean	Sample Mean	Theoretical Variance	Sample Variance	Theoretical std
[1,3]	2	2	0.66	0.66	0.812

Figure 20 Uniform distribution characteristics, μ = 2

Cache Size	Greedy CHR	Greedy Avg. Runtime	Dynamic Pr. CHR	Dynamic Pr. Avg. Runtime	Total Time
$\mathbf{n = 1000}$					
1% = 20	0.24102630909833067	0.001s	0.2410**9038429824353**	0.0088s	1.08s
2% = 40	0.3200262773569456	0.0011s	0.3200**394568537226**	0.0088s	1.06s
5% = 100	0.4371007296380347	0.001s	0.43710**856767366224**	0.0089s	1.08s
10% = 200	0.545805804977664	0.0013s	0.54580**67778543679**	0.0093s	1.14s
20% = 400	0.6694800223228726	0.0013s	0.669480**3875372334**	0.0097s	1.19s
$\mathbf{n = 5000}$					
1% = 100	0.29453210781815176	0.0059s	0.29453**73589625834**	0.0438s	5.4s
2% = 200	0.3671908425610158	0.0058s	0.36719**19494692401**	0.0454s	5.59s
5% = 500	0.4786799526305358	0.006s	0.4786**801210378908**	0.0481s	5.85s
10% = 1000	0.5780476684625024	0.0064s	0.578047**7223748153**	0.0541s	6.48s
20% = 2000	0.6929558340039277	0.0061s	0.6929558**477000074**	0.0632s	7.41s
$\mathbf{n = 10000}$					
1% = 200	0.3110986274471151	0.0121s	0.31109**9478374455**	0.0918s	11.2s
2% = 400	0.38152684919796576	0.0121s	0.38152**709431215137**	0.0959s	11.68s
5% = 1000	0.4914677158058989	0.0123s	0.4914677**941642017**	0.1076s	12.86s
10% = 2000	0.5888248966054511	0.0124s	0.588824**9208519906**	0.1262s	14.71s
20% = 4000	0.700558002587276	0.0134s	0.70055800**6984076**	0.1629s	18.52s
$\mathbf{n = 25000}$					
1% = 500	0.32939320703380637	0.0305s	0.329393**3412150134**	0.2483s	30.12s
2% = 1000	0.397857530300534	0.0297s	0.3978575**6643298514**	0.2763s	32.85s
5% = 2500	0.5049727285512485	0.0308s	0.5049727**371268602**	0.3521s	40.5s
10% = 5000	0.5995880440572602	0.0314s	0.59958804**56929981**	0.4492s	50.2s
20% = 10000	0.7085372230345132	0.033s	0.708537223**6891522**	0.6637s	71.89s
$\mathbf{n = 50000}$					
1% = 1000	0.3409203014210793	0.0594s	0.3409203**476149061**	0.5412s	64.4s
2% = 2000	0.4082256735585127	0.0602s	0.4082256**8726742026**	0.629s	73.2s
5% = 5000	0.5131761608178532	0.0614s	0.51317616**50235311**	0.8777s	98.25s
10% = 10000	0.6063998977178301	0.0628s	0.60639989**86234563**	1.2753s	138s
20% = 20000	0.7134830470530147	0.0673s	0.713483047**2979712**	2.084s	219.36s

Figure 21 Simulation Results for the uniform distribution, μ = 2

Discrete Uniform	Theoretical Mean	Sample Mean	Theoretical Variance	Sample Variance	Theoretical std
[1,19]	10	10	30	30.001	5.47

Figure 22 Uniform distribution characteristics, μ = 10

Cache Size	Greedy CHR	Greedy Avg. Runtime	Dynamic Pr. CHR	Dynamic Pr. Avg. Runtime	Total Time
$\mathbf{n = 1000}$					
1% = 100	0.26091759041070206	0.0012s	0.26**11456215945535**	0.0089s	1.09s
2% = 200	0.3421349299969754	0.0012s	0.342**21153976118274**	0.0093s	1.13s
5% = 500	0.46698441076682906	0.0013s	0.46**70098400781651**	0.0098s	1.19s
10% = 1000	0.5769390060874067	0.0013s	0.57**70032889870675**	0.0109s	1.31s
20% = 2000	0.7001525746124423	0.0014s	0.700**2311319964234**	0.0129s	1.52s
$\mathbf{n = 5000}$					
1% = 500	0.31394314746401125	0.0063s	0.3139**5432101255905**	0.0478s	5.84s
2% = 1000	0.38902148949221393	0.0065s	0.38902**48055714948**	0.0533s	6.41s
5% = 2500	0.5050144560964988	0.0065s	0.50501**50206233714**	0.0687s	7.95s
10% = 5000	0.6076314474285424	0.0067s	0.6076**405990045831**	0.0892s	10.02s
20% = 10000	0.7208183237169304	0.0072s	0.7208**342275425417**	0.1309s	14.22s
$\mathbf{n = 10000}$					
1% = 1000	0.329221892556377	0.0129s	0.32922**488391126525**	0.1063s	12.79s
2% = 2000	0.4039697255818268	0.0129s	0.40397**0495339026**	0.124s	14.56s
5% = 5000	0.5168754326963855	0.0136s	0.51687**58585852**	0.1753s	19.75s
10% = 10000	0.6169331211908016	0.0137s	0.61693**87484491304**	0.2558s	27.8s
20% = 20000	0.7278180338201672	0.0149s	0.7278**230358758981**	0.4138s	43.74s
$\mathbf{n = 25000}$					
1% = 2500	0.3474985098149124	0.0338s	0.347498**9644591816**	0.3409s	39.65s
2% = 5000	0.4191639051804622	0.0338s	0.41916**40262335971**	0.4404s	49.6s
5% = 12500	0.5301464223758339	0.035s	0.5301464**516771932**	0.7475s	80.43s
10% = 25000	0.6274865949960773	0.0357s	0.62748**86665619082**	1.2411s	129.87s
20% = 50000	0.7352787475997838	0.0372s	0.7352**808621340504**	2.1617s	222s
$\mathbf{n = 50000}$					
1% = 5000	0.35847826645979725	0.0774s	0.358478**36347661227**	0.9205s	104.16s
2% = 10000	0.4288878479545991	0.0772s	0.4288878**904318677**	1.3478s	146.81s
5% = 25000	0.5377765884556378	0.0794s	0.5377765**954884164**	2.6249s	274.74s
10% = 50000	0.6338597116986141	0.0781s	0.6338**608478493999**	4.5742s	469.62s
20% = 100000	0.7396455452288798	0.0779s	0.7396**467602100542**	9s	918.34s

Figure 23 Simulation Results for the uniform distribution, μ = 10

Discrete Uniform	Theoretical Mean	Sample Mean	Theoretical Variance	Sample Variance	Theoretical std
[1,199]	100	99.99	3300	3300.4	57.44

Figure 24 Uniform distribution characteristics, μ = 100

Cache Size	Greedy CHR	Greedy Avg. Runtime	Dynamic Pr. CHR	Dynamic Pr. Avg. Runtime	Total Time
$n = \mathbf{1000}$					
1% = 1000	0.2749320512370247	0.0012s	0.27**53347778125061**	0.0133s	1.55s
2% = 2000	0.3554067756603906	0.0017s	0.355**569754611764**	0.0153s	1.79s
5% = 5000	0.4783788238759184	0.0018s	0.478**49749787913154**	0.0217s	2.45s
10% = 10000	0.5873285447440102	0.0026s	0.587**4340082291298**	0.0314s	3.458s
20% = 20000	0.7050160053380158	0.0019s	0.705**1204579134466**	0.0558s	5.92s
$n = \mathbf{5000}$					
1% = 5000	0.3257928767185998	0.0086s	0.325**8207041781456**	0.1139s	12.77s
2% = 10000	0.3995370488824803	0.0085s	0.3995**609699687581**	0.1639s	17.7s
5% = 25000	0.5155730664939486	0.008s	0.5155**904128638928**	0.3125s	32.55s
10% = 50000	0.6171644680910414	0.009s	0.6171**825379968866**	0.5672s	58s
20% = 10^5	0.7276040567172244	0.0091s	0.7276**215762402001**	1.27s	128.9s
$n = \mathbf{10000}$					
1% = 10000	0.34074524157503716	0.018s	0.3407**5699849180296**	0.3486s	37.63s
2% = 20000	0.41495079112306066	0.0174s	0.41495**979327259463**	0.561s	58.91s
5% = 50000	0.5280807040556075	0.0174s	0.52808**94807225802**	1.1525s	118s
10% = 100000	0.6256302643034065	0.016s	0.62563**79857861328**	2.2s	223.53s
20% = 200000	0.7336999560762795	0.0149s	0.733**7089095482373**	15.4s	1551.28s
$n = \mathbf{25000}$					
1% = 25000	0.3577316866352443	0.049s	0.35773**55194487892**	1.42s	149.85s
2% = 50000	0.4301338242068312	0.0458s	0.43013**75625366861**	2.55s	262.69s
5% = 125000	0.5398319392849712	0.0426s	0.53983**51036234341**	7s	707.3s
10% = 250000	0.6353338862898696	0.0486s	0.63533**77081213497**	72.2s	7233.6s
20% = 500000	0.7409987503814766	0.0471s	0.74**10021004420195**	138.1s	13823.5s
$n = \mathbf{50000}$					
1% = 50000	0.36840909126559246	0.1165s	0.3684**107035157749**	5.9s	606.52s
2% = 100000	0.4394898561185189	0.1083s	0.4394**915658671026**	12.3s	1247.14s
5% = 250000	0.5476758730219858	0.0919s	0.54767**75433975969**	150.3s	15052.6s
10% = 500000	0.6416053944203438	0.0908s	0.64160**72484593879**	291.1s	29124s
20% = 10^6	0.7454271700991775	0.0882s	0.74542**887978638**	506.9s	50701s

Figure 25 Simulation Results for the uniform distribution, μ = 100

Discrete Uniform	Theoretical Mean	Sample Mean	Theoretical Variance	Sample Variance	Theoretical std
[1,399]	200	199.99	13266.66	13266.72	115.18

Figure 26 Uniform distribution characteristics, μ = 200

Cache Size	Greedy CHR	Greedy Avg. Runtime	Dynamic Pr. CHR	Dynamic Pr. Avg. Runtime	Total Time
$n = \mathbf{1000}$					
1% = 2000	0.27273193620931874	0.0011s	0.27**31153059414385**	0.013s	1.5s
2% = 4000	0.3533850886970898	0.0012s	0.353**55202947697423**	0.0162s	1.8s
5% = 10000	0.47736979721833833	0.0013s	0.477**49222150127835**	0.0263s	2.8s
10% = 20000	0.5860038236083996	0.0012s	0.586**1013939974378**	0.0424s	4.45s
20% = 40000	0.7067401384626534	0.0014s	0.706**8387564196338**	0.0732s	7.5s
$n = \mathbf{5000}$					
1% = 10000	0.3254934189959115	0.0071s	0.325**5213453043621**	0.1312s	14.3s
2% = 20000	0.3999268992597758	0.0069s	0.3999**482960089667**	0.2148s	22.6s
5% = 50000	0.516122507543533	0.0072s	0.5161**407120138196**	0.46s	47.17s
10% = 100000	0.6161959651341544	0.0071s	0.616**2145325065613**	0.98s	99.1s
20% = 200000	0.7277793752292939	0.0073s	0.7277**972884556977**	7.4s	740s
$n = \mathbf{10000}$					
1% = 20000	0.34066330865176603	0.0154s	0.3406**7388471802224**	0.4497s	47.4s
2% = 40000	0.41500783260905777	0.0159s	0.4150**186960674145**	0.8s	83.3s
5% = 100000	0.5279371219844604	0.0159s	0.5279**472611006457**	2.1s	214.6s
10% = 200000	0.626602301258445	0.0154s	0.6266**118259239991**	20.4s	2041.6s
20% = 400000	0.7341350390474953	0.0153s	0.7341**433665631812**	37.8s	3782s
$n = \mathbf{25000}$					
1% = 50000	0.3584394522900653	0.05s	0.3584**428675913742**	2.6s	266.2s
2% = 100000	0.4311759133446538	0.0495s	0.43117**99190637143**	5.7s	578.2s
5% = 250000	0.5407289094068061	0.0438s	0.5407**328396504016**	77s	7713.7s
10% = 500000	0.6359236339957312	0.0445s	0.63592**70540729496**	146.1s	14618s
20% = 1000000	0.7412750937872725	0.044s	0.74127**83763849043**	256.2s	25634.8s
$n = \mathbf{50000}$					
1% = 100000	0.3697491200965831	0.1083s	0.3697**506237648362**	11.8s	1195.2s
2% = 200000	0.44017233134141515	0.1032s	0.44017**39890008854**	141.7s	14188.1s
5% = 500000	0.548358755878752	0.105s	0.5483**604307390268**	360.7s	36088.3s
10% = 1000000	0.6420276129168839	0.0991s	0.64202**93402588412**	642.7s	64291.5s
20% = 2000000	0.7458391908323426	0.0928s	0.7458**410947076672**	1111s	111122.5s

Figure 27 Simulation Results for the uniform distribution, μ = 200

Below, we provide a few graphs in order to facilitate the comparison of algorithmic results:

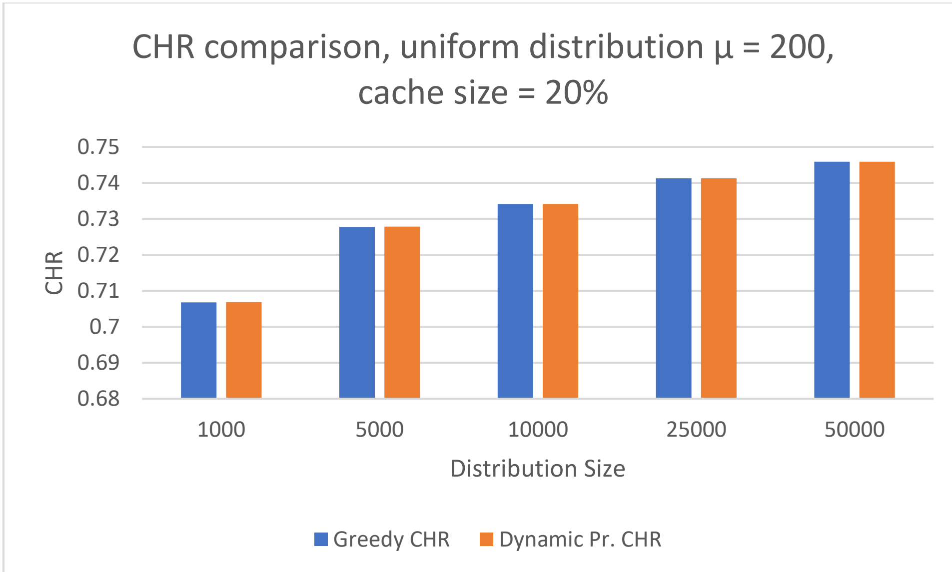

Figure 28 CHR comparison, uniform distribution, µ = 200, cache size = 20%

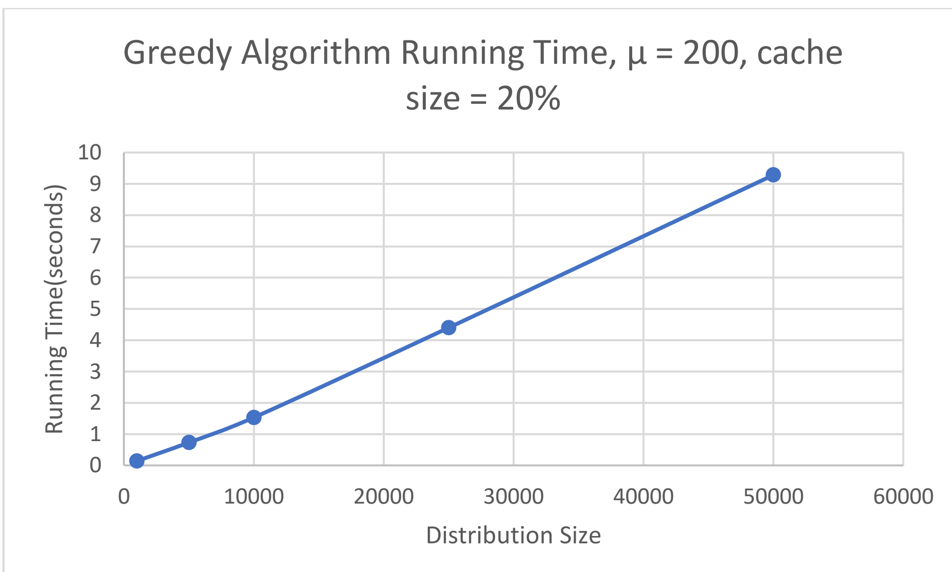

Figure 29 Greedy Algorithm Running Time, µ = 200, cache size = 20%

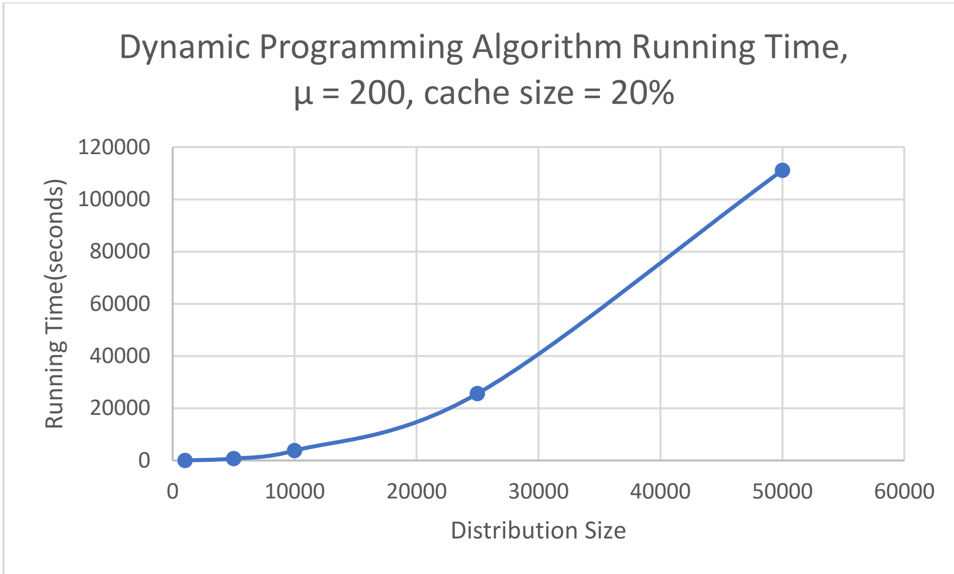

Figure 30 Dynamic Programming Algorithm Running Time, μ = 200, cache size = 20%

Once again, the results in the tables and graphs provided show that no matter what the mean value of the discrete uniform distribution, the two algorithms achieve almost identical results, although the dynamic programming algorithm consistently achieves a higher CHR. We observe that the CHR achieved in the case of the geometric distribution is slightly higher compared to the CHR achieved in the case of the discrete uniform distribution for the same mean item size value.

4.4 Simulation Results for the Pareto Weight Distributions

The last type of distribution in our simulations was the Pareto distribution. The simulations below were performed for values of the parameter α equal to 1.01, 1.16 and 1.9.

Discrete Pareto α = 1.01	Theoretical mean	Theoretical Variance
	100.5	∞

Figure 31 Pareto distribution characteristics, α = 1.01

Cache Size	Greedy CHR	Greedy Avg. Runtime	Dynamic Pr. CHR	Dynamic Pr. Avg. Runtime	Total Time
$n = \mathbf{1000}$					
1% = 1005	0.8027539582307185	0.0015s	0.802**834822117092**	0.0113s	1.37s
2% = 2010	0.9278757387102305	0.0017s	0.927**9778075686953**	0.0133s	1.6s
5% = 5025	0.9960620662048659	0.0016s	0.996**2106148024086**	0.0193s	2.19s
10% = 10050	0.9995776174991394	0.0017s	0.9995**915339552571**	0.0272s	2.99s
20% = 20100	0.9999587676078486	0.0017s	0.9999**610286761822**	0.045s	4.77s
$n = \mathbf{5000}$					
1% = 5025	0.8168957992376061	0.0085s	0.816**9099728997747**	0.1s	11.8s
2% = 10050	0.9335853518219801	0.009s	0.933**6012545391214**	0.153s	16.7s
5% = 25125	0.996899636095483	0.009s	0.996**9170412946581**	0.275s	28.93s
10% = 50250	0.9998529305691313	0.0089s	0.9998**67183984285**	0.46s	48.31s
20% = 100500	0.9999933118171757	0.0088s	0.9999933118171757	1s	109.5s
$n = \mathbf{10000}$					
1% = 10050	0.8221480877712947	0.0175s	0.8221**560188597806**	0.324s	35.2s
2% = 20100	0.935421349524826	0.0185s	0.93542**96602200808**	0.518s	54.7s
5% = 50250	0.9971570982210274	0.0187s	0.9971**683445454967**	1s	104.7s
10% = 100.500	0.9999617711155069	0.0186s	0.99996**40154421715**	2s	203.1s
20% = 201.000	0.9999917655357682	0.0174s	0.999991765535768**7**	15.7s	1577s
$n = \mathbf{25000}$					
1% = 25125	0.8266011443208049	0.055s	0.82660**45051854362**	1.5s	167.3s
2% = 50250	0.9371200578323073	0.0576s	0.93712**32247989423**	2.7s	286.2s
5% = 125.625	0.9972527066373362	0.0539s	0.99725**62139876144**	9.5s	958.6s
10% = 251.250	0.9999668002584663	0.0497s	0.99996**8412053203**	66.2s	6628.6s
20% = 502.500	0.9999987633467515	0.0483s	0.99999876334675**21**	127.2s	12731.7s
$n = \mathbf{50000}$					
1% = 50.250	0.829617359143741	0.1151s	0.82961**8980042332**	5.6s	586.5s
2% = 100.500	0.9383076656672802	0.1133s	0.93830**92416308966**	11.8s	1197.8s
5% = 251.250	0.9972734480180254	0.1159s	0.99727**55691562528**	157.7s	15790.8s
10% = 502.500	0.9999777511319453	0.1064s	0.99997**88874627056**	301.1s	30131.8s
20% = 1.005.000	0.9999985188849817	0.0982s	0.9999985**568643794**	529.6s	52975.9s

Figure 32 Simulation Results for the pareto distribution, a = 1.01

Discrete Pareto $\alpha = 1.16$	Theoretical mean	Theoretical Variance
	6.8	∞

Figure 33 Pareto distribution characteristics, a = 1.16

Cache Size	Greedy CHR	Greedy Avg. Runtime	Dynamic Pr. CHR	Dynamic Pr. Avg. Runtime	Total Time
$n = \mathbf{1000}$					
1% = 68	0.3706629149684002	0.0012s	0.3706**927468283927**	0.0086s	1s
2% = 136	0.47365679667759414	0.0012s	0.4736**607057330222**	0.0089s	1.1s
5% = 340	0.6234148985257825	0.0014s	0.62341**60416353149**	0.0095s	1.18s
10% = 680	0.757701068980383	0.0015s	0.757701**2949059058**	0.0107s	1.32s
20% = 1360	0.8958763756762245	0.0016s	0.895**9587874939909**	0.0121s	1.47s
$n = \mathbf{5000}$					
1% = 340	0.42024791169856246	0.006s	0.42024**97394262826**	0.0462s	5.71s
2% = 680	0.5102307303489586	0.0062s	0.51023**1722521159**	0.0514s	6.27s
5% = 1700	0.6507728420273282	0.0068s	0.650772**9140647343**	0.0616s	7.3s
10% = 3400	0.7763851569496993	0.0074s	0.776385**2269100104**	0.0808s	9.3s
20% = 6800	0.9045508619729311	0.008s	0.9045**621638144117**	0.1143s	12.7s
$n = \mathbf{10000}$					
1% = 680	0.43365996963610426	0.0125s	0.4336**604771134958**	0.1048s	12.7s
2% = 1360	0.5211105275011263	0.0127s	0.521110**6975437642**	0.1167s	13.94s
5% = 3400	0.6590199493802732	0.0132s	0.6590199**943562695**	0.1606s	18.37s
10% = 6800	0.7812834123402022	0.0145s	0.7812834**241009018**	0.2358s	26s
20% = 13600	0.9076135434529462	0.0162s	0.90761**95290948028**	0.358s	38.4s
$n = \mathbf{25000}$					
1% = 1700	0.4471979418013976	0.0333s	0.44719**811336532883**	0.3174s	37.6s
2% = 3400	0.5340451335471473	0.0343s	0.534045**2387920108**	0.4028s	46.2s
5% = 8500	0.6686727166024822	0.0358s	0.66867271**7580867**	0.6525s	71.38s
10% = 17000	0.7871352427635588	0.0394s	0.787135242**9643467**	1.04s	110.47s
20% = 34000	0.9099156437718376	0.0429s	0.90991**77594742598**	1.8s	186s
$n = \mathbf{50000}$					
1% = 3400	0.4569749602700892	0.0696s	0.4569749**638161411**	0.8068s	92.65s
2% = 6800	0.5414721499572308	0.0709s	0.5414721**549085404**	1.12s	124.8s
5% = 17000	0.6738432084211345	0.073s	0.673843208**7292746**	2s	214.4s
10% = 34000	0.7906990840489415	0.0804s	0.790699084**6871709**	3.5s	365.3s
20% = 68000	0.9115080122416644	0.086s	0.911508**9143523177**	6.7s	690.8s

Figure 34 Simulation Results for the pareto distribution, a = 1.16

Discrete Pareto α = 1.9	Theoretical mean	Theoretical Variance
	1.75	∞

Figure 35 Pareto distribution characteristics, a = 1.9

Cache Size	Greedy CHR	Greedy Avg. Runtime	Dynamic Pr. CHR	Dynamic Pr. Avg. Runtime	Total Time
$n = \mathbf{1000}$					
1% = 18	0.26774226490203257	0.0011s	0.267**81132861611817**	0.0085s	1s
2% = 35	0.34646402892080586	0.0012s	0.3464**807466466814**	0.0085s	1.07s
5% = 88	0.47006287649212675	0.0012s	0.470**1227150683427**	0.0087s	1.09s
10% = 175	0.5811607002506596	0.0012s	0.58116**12562924355**	0.0091s	1.13s
20% = 350	0.7093314343223075	0.0013s	0.709331**8664782611**	0.0096s	1.18s
$n = \mathbf{5000}$					
1% = 88	0.31522873813773755	0.006s	0.3152**5408850463643**	0.0432s	5.43s
2% = 175	0.3898576963836037	0.006s	0.38985**81060795756**	0.0447s	5.55s
5% = 438	0.5081965936468305	0.006s	0.50819**76366440218**	0.049s	6s
10% = 875	0.6117342467541662	0.0062s	0.611734**3052880002**	0.0544s	6.55s
20% = 1750	0.7305590726090769	0.0067s	0.730559**1050579285**	0.0664s	7.8s
$n = \mathbf{10000}$					
1% = 175	0.3319256014681526	0.0118s	0.33192**672822616237**	0.0896s	11.1s
2% = 350	0.40526020993843337	0.0118s	0.405260**3703263784**	0.0935s	11.51s
5% = 875	0.519932893914531	0.0124s	0.519932**919890781**	0.109s	13.15s
10% = 1750	0.6208746818342036	0.0126s	0.62087468**83859625**	0.1327s	15.52s
20% = 3500	0.7375986516386441	0.0131s	0.73759865**37952106**	0.1708s	19.38s
$n = \mathbf{25000}$					
1% = 438	0.3501089693828819	0.0301s	0.35010**907455789814**	0.249s	30.44s
2% = 875	0.42125401421525255	0.0306s	0.4212540**819824545**	0.273s	32.8s
5% = 2188	0.532592438342475	0.031s	0.5325924**443772836**	0.35s	40.95s
10% = 4375	0.6309837026234897	0.0326s	0.63098370**3292129**	0.4629s	52s
20% = 8750	0.7447482789004307	0.0339s	0.74474827**96048712**	0.71s	77s
$n = \mathbf{50000}$					
1% = 875	0.36033925742149925	0.0628s	0.3603**410937305476**	0.563s	67.7s
2% = 1750	0.4304328615594892	0.0631s	0.430432**9000213182**	0.6712s	78.4s
5% = 4375	0.5402120692490817	0.065s	0.54021206**9151798**	0.929s	104.4s
10% = 8750	0.637136989568242	0.0658s	0.63713698**9079954**	1.4s	151s
20% = 17500	0.7489528070058118	0.0695s	0.74895280**71099921**	2.2s	238.6s

Figure 36 Simulation Results for the pareto distribution, a = 1.9

Below, we provide a few graphs in order to further facilitate the comparison of the algorithmic results:

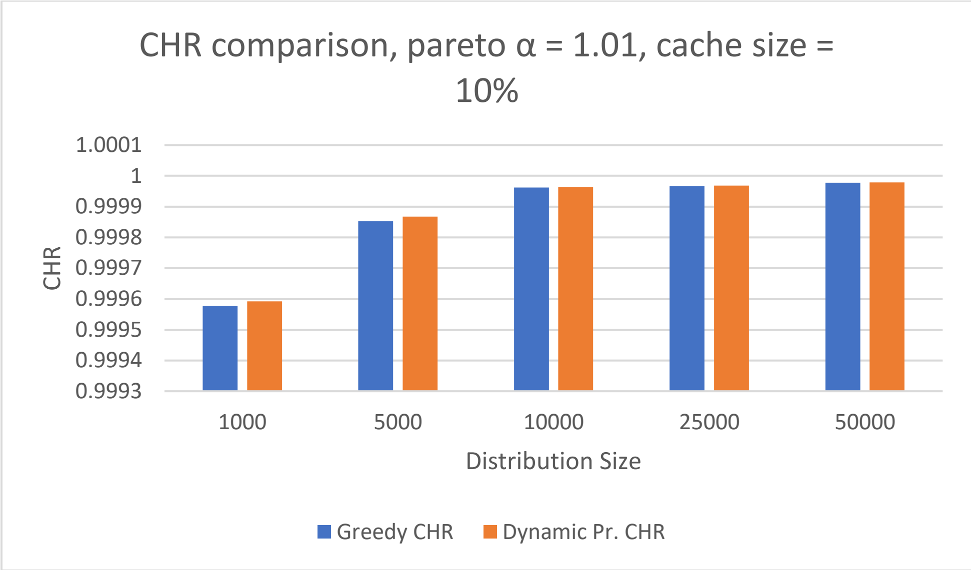

Figure 37 CHR comparison, pareto α = 1.01, cache size = 10%

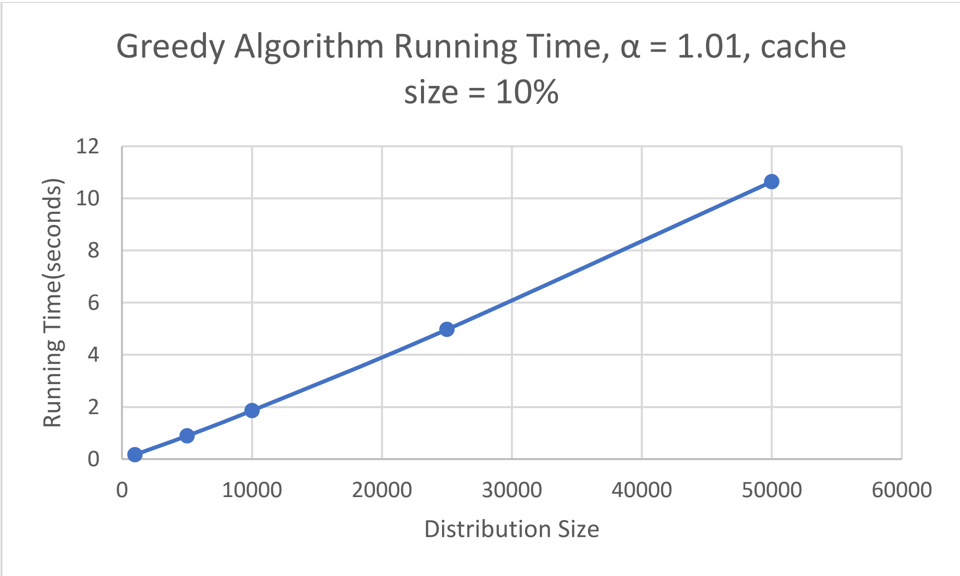

Figure 38 Greedy Algorithm Time, α = 1.01, cache size = 10%

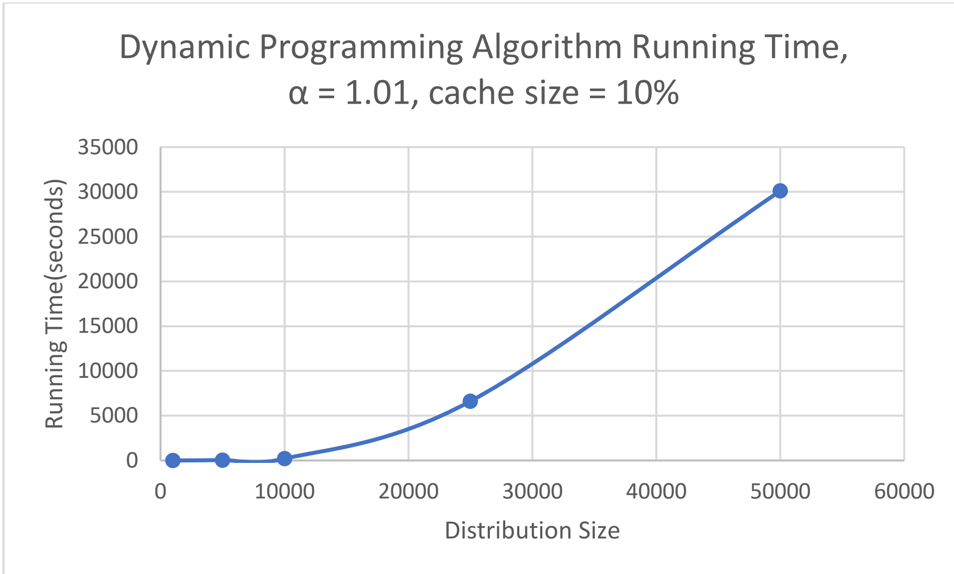

Figure 39 Dynamic Programming Time, α = 1.01, cache size = 10%

Even in the extreme cases of a Pareto distribution, we still notice no noticeable difference between the performance of the two algorithms, in terms of CHR. The only discernible difference is the time complexity of the two algorithms.

4.5 Conclusions

We have conclusively tested the two algorithms using Geometric, Discrete Uniform and Pareto weight distributions. All the tests presented were done using a Zipf popularity probability distribution with a shape factor s equal to 0.2. Additional tests were performed for each weight distribution using different Zipf shape factors. These shape factors were s = 0 corresponding to a highly skewed distribution and s = 0.5 corresponding to a less skewed distribution. However, the results we obtained were similar to the results already presented, therefore they were not included in the Thesis. As expected, the greedy algorithm, since it is not an optimal solution, does not exactly achieve the optimal results of the dynamic programming algorithm. However, from the presented results we can clearly conclude that the greedy algorithm consistently achieves almost identical results in terms of CHR, that only differ in some decimal places. These results are true no matter the kind of the weight distribution of our data or the skew of our Zipf popularity distribution.

Studying our results, we also observe that if the weight distribution is a Pareto one, the CHR achieved is extremely high, especially for large cache sizes. In many cases, we even manage to achieve Cache Hit Ratios close to 100%. Moreover, comparing the CHR results for the geometric and for the discrete uniform distributions for the same mean item size value, we conclude that the results are almost identical, with the results for the geometric distribution case being slightly higher.

In terms of average runtime there is a huge difference between the two algorithms, which is highlighted as the number of items n increases. As the number of items n increases, we confirm that the dynamic programming algorithm does not have a linear time complexity, but rather exhibits its pseudo polynomial complexity. All things considered, we can confidently conclude that the greedy algorithm can achieve an almost optimal solution in all cases examined while providing a vastly superior time complexity. Therefore, in a caching scenario where costs and time become increasingly important, while there is an ever-increasing number of items, it can be of great benefit to use the specific simple and fast greedy approach rather than solving the problem using dynamic programming. The dynamic programming solution is proven to only be suitable for usage in cases where the number of items in the database is very small.

4.6 Ideas for Future Work

An interesting idea would be to predict content popularity by examining the evolution patterns of content popularity on provisional services. Therefore, the work in this Thesis could be expanded by modifying the greedy heuristic to take into account popularity estimates for new content. Viral content rises in popularity really fast, where it either remains popular for a very long time, or it declines in popularity equally fast.

This idea could be implemented using a Machine Learning approach, by automatically leveraging the vast amount of data provisional services have, in order to obtain the capability to recognize the popularity patterns of content. A similarity supervised learning approach could be applied. Similarity learning uses a similarity function to measure how similar new objects are. This could be used in order to classify the popularity patterns of new content and therefore manage to successfully predict future viral content.

Finally, we could also expand this work to take into account the recommendation system as well. This would in turn imply a need to change our heuristic, since in such case we have to jointly solve a complex problem. Related work on this subject classifies the joint caching and recommendations problem as a generalization of the 0-1 Knapsack Problem, meaning it is NP-hard. Therefore, in the future, we could propose a heuristic algorithm that jointly takes into account both caching and recommendations.

Appendix

Below, we are going to provide further clarification on how we constructed the random variate of the discrete pareto distribution. Given the Cumulative Distribution Function (CDF) of the pareto distribution, we compute its inverse. In this case, the CDF of the pareto distribution is equal to:

$$F_X(x) = P[X \leq x] = \begin{cases} 1 - \left(\frac{x_m}{x}\right)^a & , x \geq x_m \\ 0, otherwise \end{cases}$$

Where x_m is the minimum value. In our case we assume $x_m = 1$. We need to solve the equation F(X) = R, for X in terms of R where R is a random number in [0,1].

$$1 - \left(\frac{1}{x}\right)^a = R \Rightarrow 1 - R = \left(\frac{1}{x}\right)^a \Rightarrow \ln(1 - R) = a * ln\left(\frac{1}{x}\right) \Rightarrow \ln\left(\frac{1}{x}\right) = \frac{\ln(R)}{a} \Rightarrow e^{\ln\left(\frac{1}{x}\right)}$$
$$= e^{\frac{\ln(R)}{a}} \Rightarrow 1/x = e^{\ln(R)/a} \Rightarrow x = \frac{1}{e^{\frac{\ln(R)}{a}}}, x > 1$$

In order to discretize it, we use the floor function. Therefore, the random variate generator is equal to:

$$x = \lfloor 1/e^{lnR/a} \rfloor$$

Calculating the mean and variance of the discrete pareto distribution:

First, we calculate the survival function of the pareto distribution, which is equal to:

$$S(x) = P[X \geq x] = P[X > x] = 1 - P[X \leq x] = \begin{cases} \frac{1}{x^a}, x \geq 1 \\ 1, x < 1 \end{cases}$$

The Probability Mass Function (PMF) of the discretized pareto distribution is equal to [15]:

$$P[X = x] = S(x) - S(x + 1)$$

- $x + 1 < 1 \Leftrightarrow x < 0$:

$$P[X = x] = S(x) - S(x + 1) = 1 - 1 = 0$$

- $x = 0$

$$P[X = x] = S(0) - S(1) = 1 - \frac{1}{1^a} = 0$$

- $x > 0$

$$P[X = x] = S(x) - S(x+1) = \frac{1}{x^a} - \frac{1}{(x+1)^a}$$

We now calculate the rth moment of the discretized pareto:

$$E(x^r) = \sum_{x=1}^{\infty} x^r P[X = x] = \sum_{x=1}^{\infty} x^r (\frac{1}{x^a} - \frac{1}{(x+1)^a})$$

We make the following observation:

$$S_1 = \sum_{x=1}^{1} x^r (\frac{1}{x^a} - \frac{1}{(x+1)^a}) = 1^r \left(\frac{1}{1^a} - \frac{1}{2^a}\right) = (1^r - 0^r)\frac{1}{1^a} - 1^r \frac{1}{2^a}$$

$$S_2 = 1^r \left(\frac{1}{1^a} - \frac{1}{2^a}\right) + 2^r \left(\frac{1}{2^a} - \frac{1}{3^a}\right) = (1^r - 0^r)\frac{1}{1^a} + (2^r - 1^r)\frac{1}{2^a} - 2^r \frac{1}{3^a}$$

$$\begin{aligned} S_3 &= 1^r \left(\frac{1}{1^a} - \frac{1}{2^a}\right) + 2^r \left(\frac{1}{2^a} - \frac{1}{3^a}\right) + 3^r \left(\frac{1}{3^a} - \frac{1}{4^a}\right) \\ &= (1^r - 0^r)\frac{1}{1^a} + (2^r - 1^r)\frac{1}{2^a} + (3^r - 2^r)\frac{1}{3^a} - 3^r \frac{1}{4^a} \end{aligned}$$

where S_i is the partial sum of the first i terms of the summation $\sum_{x=1}^{\infty} x^r P[X = x]$.

Therefore:

$$S_n = \sum_{x=1}^{n} \left[\frac{x^r - (x-1)^r}{x^a}\right] - \frac{n^r}{(n+1)^a}$$

The mean value of the distribution is the first moment, so we set $r = 1$, then

$$S_n = \sum_{x=1}^{n} \left[\frac{1}{x^a}\right] - \frac{n}{(n+1)^a}$$

So, the mean value is equal to:

$$E(x) = \lim_{n\to\infty} S_n$$

The first part of S_n: $\sum_{x=1}^{n} \left[\frac{1}{x^a}\right]$ is a hyperharmonic series, which converges if $\alpha > 1$, as is the case in this work (α takes three possible values in our work, 1.01, 1.16 and 1.9).

Regarding the second part of S_n: $\frac{n}{(n+1)^a}$:

$$0 \leq \frac{n}{(n+1)^a} \leq \frac{n+1}{(n+1)^a} = \frac{1}{(n+1)^{a-1}}$$

and the term on the r.h.s. above converges to 0 if $\alpha > 1$,as n tends to infinity $\frac{1}{(n+1)^{a-1}} \to 0$.

So, we come to the conclusion that $\lim_{n\to\infty} \frac{n}{(n+1)^a} = 0$

The first moment is therefore equal to:

$$E(x) = \lim_{n\to\infty} S_n = \lim_{n\to\infty} \left(\sum_{x=1}^{n} \left[\frac{1}{x^a}\right] - \frac{n}{(n+1)^a}\right) = \lim_{n\to\infty} \sum_{x=1}^{n} \left[\frac{1}{x^a}\right] - \lim_{n\to\infty} \frac{n}{(n+1)^a} = \sum_{x=1}^{\infty} \left[\frac{1}{x^a}\right]$$

In conclusion:

$$\mu = E(x) = \sum_{x=1}^{n} \left[\frac{1}{x^a}\right] \in R$$

In order to calculate the variance, we need to calculate the 2nd moment.

If we assume r = α, then:

$$S_n = \sum_{x=1}^{n} \left[\frac{x^a - (x-1)^a}{x^a}\right] - \frac{n^a}{(n+1)^a}, a \in N^*$$

$$\lim_{n\to\infty}\frac{n^r}{(n+1)^a}=\lim_{n\to\infty}\frac{n^r}{(n+1)^r}=\lim_{n\to\infty}\left(\frac{n}{n+1}\right)^r=1^r=1$$

So:

$$E(x^r)=\lim_{n\to\infty}S_n=\sum_{x=1}^{\infty}\left[\frac{x^r-(x-1)^r}{x^r}\right]-1$$

So, all that is left is to calculate $\sum_{x=1}^{\infty}\left[\frac{x^r-(x-1)^r}{x^r}\right]$.

We are going to prove that $x^r-(x-1)^r\geq x^{r-1}$, when $r\geq 1$ and $x\geq 1$:

$$x^r-(x-1)^r\geq x^{r-1}\overset{x^r}{\Leftrightarrow}1-\left(\frac{x-1}{x}\right)^r\geq\frac{1}{x}\Leftrightarrow\frac{x-1}{x}\geq\left(\frac{x-1}{x}\right)^r\xleftrightarrow{0<\frac{x-1}{x}<1}1\leq r$$

Therefore, our statement is true.

Using the above statement:

$$E(x^r)\ \geq\sum_{x=1}^{\infty}\left[\frac{x^{r-1}}{x^r}\right]-1=\sum_{x=1}^{\infty}\left[\frac{1}{x}\right]-1=\ +\infty$$

The sum $\sum_{x=1}^{\infty}\left[\frac{1}{x}\right]$ is a harmonic series, therefore it diverges. So, we managed to show that $E(x^r)=E(x^a)=+\infty$.

Now in order to calculate the 2nd moment, we are going to use what we proved above.

$$E(x^2)=\sum_{x=1}^{\infty}x^2\left(\frac{1}{x^a}-\frac{1}{(x+1)^a}\right)$$

We know that $x^2\geq x^a$, since $x\geq 1\ and\ a<2$.

Therefore,

$$E(x^2)\ \geq\sum_{x=1}^{\infty}x^a\left(\frac{1}{x^a}-\frac{1}{(x+1)^a}\right)=E(x^a)=+\infty$$

Finally, in order to calculate the variance:

$\sigma^2 = E(x^2) - E(x)^2$, which is equal to infinite minus something that converges. Which means the result is $+\infty$.

References

[1] C. A. Gomez-Uribe and N. Hunt, “The Netflix recommender system: Algorithms, business value, and innovation,” ACM Trans. on Management Information Systems, pp. 6(4):13:1–13:19, 2016.

[2] R. Zhou, S. Khemmarat, and L. Gao, “The impact of YouTube recommendation system on video views,” in Proc. ACM IMC, Melbourne, Australia, November 2010, pp. 404–410.

[3] Cisco Visual Networking Index: Forecast and Trends, 2017–2022.

[4] A. Satsiou and M. Paterakis “Impact of Frequency-Based Cache Management Policies on the Performance of Segment Based Video Caching Proxies”. In: Mitrou N., Kontovasilis K., Rouskas G.N., Iliadis I., Merakos L. (eds) Networking 2004. NETWORKING 2004. Lecture Notes in Computer Science, vol 3042. Springer, Berlin, Heidelberg

[5] P. Sermpezis, T. Giannakas, T. Spyropoulos, L. Vigneri, “Soft Cache Hits: Improving Performance through Recommendation and Delivery of Related Content” in IEEE Journal on Selected Areas in Communications (Volume: 36, Issue: 6, June 2018).

[6] D. Liu, C. Yang (2018) “A Learning-based Approach to Joint Content Caching and Recommendation at Base Stations”. 1-7. 10.1109/GLOBECOM.2018.8647827

[7] E. Chatzieleftheriou, M. Karaliopoulos, I. Koutsopoulos (2018) “Jointly Optimizing Content Caching and Recommendations in Small Cell Networks”. IEEE Transactions on Mobile Computing. PP. 1-1. 10.1109/TMC.2018.2831690.

[8] T. Giannakas, P. Sermpezis, T. Spyropoulos, (2018) “Show me the Cache: Optimizing Cache-Friendly Recommendations for Sequential Content Access”. in Proc. of IEEE WoWMoM 2018.

[9] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, “Web caching and zipf-like distributions: Evidence and implications,” in Proc. IEEE INFOCOM ’99, New York, NY, USA, March 1999, pp. 126–134.

[10] P. Gill, M. F. Arlitt, Z. Li, and A. Mahanti, "Youtube traffic characterization: A view from the edge," in Proc. 7th ACM SIGCOMM Internet Measurement Conference (IMC), San Diego, California, USA, October 2007, pp. 15–28.

[11] X. Wang, M. Chen, T. Taleb, A. Ksentini, V. C. M. Leung "Cache in the Air: Exploiting Content Caching and Delivery Techniques for 5G Systems" in IEEE Communications Magazine (Volume: 52 , Issue: 2 , February 2014)

[12] E. Bastug, M. Bennis, M. Debbah "Living on the Edge: The Role of Proactive Caching in 5G Wireless Networks" in IEEE Communications Magazine (Volume: 52 , Issue: 8 , Aug. 2014)

[13] K. Shanmugam, N. Golrezaei, A. Dimakis, A. Molisch, G. Caire "FemtoCaching: Wireless Content Delivery through Distributed Caching Helpers" in IEEE Transactions on Information Theory (Volume: 59 , Issue: 12 , Dec. 2013)

[14] Kyriakos Axiotis, Christos Tzamos, "Capacitated Dynamic Programming: Faster Knapsack and Graph Algorithms" in 46th International Colloquium on Automata, Languages and Programming, 2019

[15] Krishna, Hare & Pundir, Pramendra. (2009) "Discrete Burr and discrete Pareto distributions" Statistical Methodology. 6. 177-188. 10.1016/j.stamet.2008.07.001.