An Analysis of K-Server algorithm performance

Stefan Caldararu[1] and Marc Renault[2]

[1] *Undergraduate Student with Department of Computer Science, UW-Madison*
[2] *Professor in the Department of Computer Science, UW-Madison*

May 12, 2023

Department of Computer Science
University of Wisconsin – Madison, USA

Abstract

Write something here...

Keywords: add keywords...

Contents

1 Introduction

1.1 Outline

This paper considers the K-Servers problem, and the practical development of algorithms within a C++ environment. Additionally, we both provide a literature review, and multiple forms of analysis of a variety of algorithms. In sec. 1.2, we provide a problem description for the K-Servers problem. In section 2 a brief literature overview is provided, and then sec. 2.2 describes different performance metrics that can be used when looking at these algorithms. In section 3, multiple popular K-Servers algorithms are described, and a brief list of benefits and drawbacks are provided. We then describe the software implementation for the algorithms in sec. 4, and some analysis of numerical experiments in sec. 5. Finally, we provide some analysis of results in sec. 6, and propose future research thrusts in sec. 7.

1.2 Problem Description

An instance of the K-Servers problem can be described by a metric space $M = (X, d)$, a number of servers $k > 1$, and an input sequence $\sigma = (r_1, r_2, r_3, ..., r_n)$, where each r_i corresponds to a point in the metric space. Each of the k servers is assigned an initial starting location within the metric space (generally this is assumed to be the first k requests of the input sequence). Following this, The sequence of reqeusts is processed one at a time. When a request comes in, a given algorithm ALG must decide on one of the servers to service the request. It must then move said server from it's current location x to the request r_i. This incurs a cost of $c = d(x, r_i)$. The goal is to have ALG incur the smallest possible cost while servicing all of the requests in the sequence [1].

There are two major distinctions to be made between different classes of algorithms. The first is the classification of a "lazy" algorithm - one that only moves a server in order to service the current request. Non-lazy algorithms will process a request, and then potentially also move other servers preemptively in order to prepare for future requests. It is important to note that for any non-lazy algorithm that performs well, there is a parallel lazy algorithm that performs just as well, if not better. We can describe this algorithm as follows: suppose we have our non-lazy algorithm, ALG. We have the algorithm $LAZY$ service requests with the same servers that ALG services requests. Suppose that ALG moves a server from location x_1 to location x_2 preemptively, and then later services request r_i with this server. $LAZY$ will be servicing r_i with the same server, except it will be moving directly from x_1 instead of x_2. By the triangle inequality, $d(x_1, r_i) \leq d(x_1, x_2) + d(x_2, r_i)$. By applying this principle throughout the request sequence, we will see that the cost of $LAZY$ will be as good if not better than that of ALG. This shows us that we can create a lazy algorithm from a non-lazy algorithm by maintaining "ghost" locations for servers in relation to how the non-lazy algorithm would use them. Then, we can determine which server the non-lazy algorithm would use, and then service that location with the correct server [1].

The second major distinction is between "online" and "offline" algorithms. An offline algorithm receives the entire request sequence at once, and so as a result is able to make decisions on what server to use for the current request based off of future requests. In contrast, an online algorithm receives the request one at a time, and so is only able to make decisions based off of past requests, the current server configuration, and the current request. While online algorithms are at a severe

disadvantage due to this, real world applications often rely on the performance of these algorithms. Applications range from disk access optimization, such as the two headed-disk problem [1], to police or firetruck servicing.

2 Background

2.1 Literature Overview

As preliminary research for this project, a literature review was conducted to gauge the current state of research into the K-Servers problem. First, initial readings included information on online algorithms and competitive analysis proofs, in order to get a solid background on the subject. Future readings progressed to common K-Servers algorithms, and their performance relative to the competitive ratio [1]. Following readings included information about other analysis methods [2, 3], which are described in sec. 2.2. Finally, a practical implementation for the Work Function Algorithm was analyzed [4], and algorithm implementations began, as described in sec. 3 and sec. 4.

2.2 Analysis Methods

Competitive Analysis

The most popular method for analyzing the performance of online algorithms for a problem is competitive analysis. Here, we assume a "malicious adversary", who is attempting to make our algorithm ALG perform as poorly as possible in relation to the performance of the optimal algorithm OPT. The malicious adversary is allowed to come up with any finite input, and our competitive ratio is determined by this. If for any finite input, we are able to guaruntee that ALG has a cost within a certain ratio c of OPT (allowing for a constant additive factor), then we say that our algorithm is c-competitive [1]. So, if we define $ALG(\sigma)$ as the cost ALG incurs while processing request σ, then we have the following definition:

Definition 2.1. Algorithm ALG is said to be ***c*-competitive** if for every finite request sequence σ , $ALG(\sigma) \leq c \cdot OPT(\sigma) + \alpha$ for some constant α.

Competitive analysis is in some sense similar to "worst case" analysis, where we try to see how poorly an algorithm will ever perform. This may not always be as useful in practice, as there are algorithms that perform very well on most inputs, but given specific inputs may not be competitive at all. That is, given some value c, a finite length input can be found such that the algorithm doesn't satisfy the above definition. Additionally, a lower bound of k has been shown for any online algorithm's competitive ratio. This means that if we are looking at the 3-Servers problem, the best competitive ratio an online algorithm can achieve is 3 [1].

Direct Analysis

Direct analysis is a similar techinque to competitive analysis, in that we directly compare the performance of ALG to the performance of OPT on a request. Rather than looking across all request sequences of any finite length, we determine a specified length for our request sequence. Then, we look directly at the ratio of ALG's to OPT's performance for each input, and take the largest such ratio. It is important to note that as the length of our input approaches infinity, our direct analysis ratio will aproach the competitive ratio of our algorithm.

Definition 2.2. Algorithm ALG has a direct analysis ratio of c if for every input σ of length n, $ALG(\sigma) \leq c \cdot OPT(\sigma)$.

Max/Max Ratio

For the Max/Max ratio, we are comparing the worst case of each algorithm to each other. In practice, this makes sense to do on finite sets of input sequences. Here, we will take the highest cost of ALG and the highest cost of OPT, and this ratio will be our performance metric [2].

Bijective Analysis

The bijective ratio is similar to using direct analysis, except we allow for a bijection between the two data sets. So, we look at the input space of all inputs of length n, denoted I_n. If there exists a bijetion $\pi : I_n \to I_n$ such that $ALG(\sigma) \leq c \cdot OPT(\pi(\sigma))$, then we say that the bijective ratio between ALG and OPT is c [3]. Therefore, we obtain def. 2.3.

Definition 2.3. For a given input space I_n, we say that algorithm A has bijective ratio c with respect to algorithm B if there exists a bijection $\pi : I_n \to I_n$ such that $A(\sigma) \leq c \cdot B(\pi(\sigma))$, $\forall \sigma \in I_n$.

It is important to note that this definition doesn't only compare the ratio of an algorithm to the optimal, but can also be used to compare between two different online algorithms. This can be used to prove an online algorithms optimality [3], and allows for some interesting test-case results found in sec. 6.

3 Algorithm Description

Random Algorithm

The random algorithm $RAND$ is the most basic of all of the online algorithms, and has little practical use. If the request is not currently covered by a server, then $RAND$ randomly selects one of it's servers, and moves the selected server to the service point. This algorithm is mostly just used as a baseline, as we would hope that none of the other algorithms will perform worse than random.

Greedy Algorithm

The greedy algorithm $GREEDY$ is also a computationally inexpensive online algorithm, but which has many practical uses. This algorithm checks the distance that each server would have to travel to get to the service point, and selects the server which would incur the smallest cost. While this algorithm has very good performance for the majority of practical application inputs, it is surprisingly not a competitive algorithm [1]. $GREEDY$ is one of the primary focuses of this study, as it is often looked over due to it's non-competitiveness, but still performs very well in practice.

Optimal Algorithm

Any optimal offline algorithm OPT is simply defined as an algorithm that will have the smallest possible cost for any request sequence. Computationally, the fastest implementations leverage a

reduction to a Min Cost Max Flow problem. This reduction can be further studied in [4]. This still ends up being a good bit more computationally expensive than the previous algorithms, but is needed to be used as a metric to compare algorithms against, as most algorithms are compared to the optimal when determining their strenghts.

Work-Function Algorithm

The Work-Function Algorithm *WFA* is considered to be one of the most promising algorithms in terms of the competitive ratio, as it has been proven to be $(2k-1)$-competitive, and is believed to be k-competitive. Given a certain starting server configuration, current configuration, and previous input request sequence, *WFA* determines which server an optimal algorithm would move, while also ensuring that $k-1$ of the servers end up in the current server configuration. This final server is then used to service the current request. A similar reduction to the OPT computation can be used to find this server [4]. This means that the *WFA* must compute a Min Cost Max Flow for each request, making it much more expensive than OPT, and all of the other algorithms. Additionally, it is also worth noting that this is not a finite memory algorithm, as the WFA must remember all of the previous requests [2].

Double Coverage Algorithm

The Double Coverage algorithm DC is a k-competitive algorithm on the line. If the request is to the left of all of our current server locations, then we just move the left most server to service the request. If the server is to the right of all of the current locations, we use the rightmost server. Otherwise, the request will be between two servers. In this case, we move the two closest servers towards the reqeust at the same rate, until one of the servers reaches the request location. A proof of DC's competitiveness can be found in [1]. It is also important to note that DC is not a lazy algorithm, as it does move more than one server at a time. Additionally, generalizations of this algorithm can be used on metric spaces other than the line, such as trees.

k-Centers Algorithm

The k-Centers algorithm divides the metric space into k sections, and assigns each server a section of the space to operate in. Implementations of this algorithm can either have the server return to the center of it's operating space after servicing the request in a non-lazy fashion, or just remember the bounds of each operating space, and service each request with the appropriate server [3].

4 Software implementation details and API

This program is implemented in C++, and is available publicly at https://github.com/StefanCaldararu/KServers. There is a main program defined in the "csv parser" file, which is intended to take a csv file as input, and will output a corresponding csv file. Additionally, there is an implementation of this parser that is able to run multiple inputs in parallel, and is designed to run with 16 independant threads. The input file should be formatted with the first line specifying which algorithms should be run. There will either be a 1, or a 0 corresponding to whether or not the algorithm is being run for this input. The following input line is a single integer, describing how many input metric spaces there will be. The following is repeated for each metric space. First, the size of the metric space, n

is defined. Following that, n lines of length n are written, corresponding to the cost to get from the row index to the column index. Following this, the number of inputs is specified, and the number of servers for this problem is specified on seperate lines. Finall, inputs are written on individual lines. The output file is formatted such that the input is printed, and then each algorithms output is printed for each input.

There are a couple of notable class definitions in this application that help the algorithms run. First, there is a metric space class, which holds an adjacency matrix of cost between nodes. This also has information about the server configuration, number of servers, and has methods allowing the main program to move the servers around and edit the metric space. There is additionally an Algorithm super class, which each algorithm derives the majority of its methods from. Finally, the algorithms *OPT* and *WFA* use the class "Mcfp" in order to compute their Min Cost Max Flow problems. This allows them to generate network flow graphs, compute their costs, and for *WFA*, determine which server to move.

Finally, there is a set of programs designed to generate input files. These files can generate inputs of lines or circles, with any number of nodes and servers. There is additionally a file designed to take the output files, and process the data in te different forms of analysis. Currently, data is generated for comparison between *GREEDY* and *OPT*, *WFA* and *OPT*, and between *WFA* and *GREEDY*, using Bijective analysis, the Max/Max ratio, and Direct analysis.

5 Numerical Experiments

In this section, we will present some of the numerical results from tests that were run. These are entirely experiment based, and there are no proof based results yet. In section 6, we will describe some of the interesting qualities of the results presented. Proof of these results will be a part of the future works.

Data on the bijective ratios between *OPT*, *GREEDY*, and *WFA* are provided, as these were found to be the more intersting results. Table 1 shows which tests were run, and assigns each test an input number. Table 2 shows the results of the bijective analysis for each input number. Tests were run on a line metric space, and circle metric space. For all tests, points are distributed with uniform distance.

6 Analysis

While data was also collected for bijective ratios between the other algorithms, as well as Max/Max ratios and direct analysis ratios, the data presented is the most notable. *GREEDY* has already been show to be non-competitive, and competitive ratios for *DC* and *KS* have already been proven. For *WFA*, a competitive ratio of $2k-1$ has been proven, and no input with competitive ratio worse than k has been found. This makes the Direct Analysis rather uninteresting. Additionally, the Max/Max ratio on small inputs such as these doesn't seem to provide much insight into patterns, or the true performance of the algorithms.

Input Number	Space Type	Space Size	Servers	Input Length	Number of Inputs
1	Line	6	2	8	ALL
2	Line	8	2	7	ALL
3	Line	8	3	7	ALL
4	Line	10	3	8	ALL
5	Line	10	3	20	10,000,000
6	Line	10	3	100	1,000,000
7	Circle	6	2	8	ALL
8	Circle	8	3	8	ALL
9	Circle	10	2	6	ALL
10	Circle	10	3	6	ALL
11	Circle	20	3	6	ALL

Table 1: Tests

Input Number	GRE/OPT Bij.	WFA/OPT Bij.	WFA/GRE Bij.
1	3/2	3/2	4/3
2	3/2	1.55	4/3
3	3/2	1.6	4/3
4	3/2	1.59	4/3
5	1.319	1.308	1.03
6	1.447	1.486	1.2
7	3/2	3/2	7/6
8	3/2	3/2	4/3
9	3/2	3/2	7/6
10	3/2	3/2	4/3
11	3/2	3/2	4/3

Table 2: Results

The first thing to note is that running these algorithms on a small subset of random inputs of a certain length seems to provide little to no insight into the algorithms true performace. Both *GREEDY* and *WFA* perform well - if not optimally - on a large subset of the inputs. As our input lengths increase, the number of possible inputs increases exponentially. To demonstrate the phenomena we observe, suppose for example that *GREEDY* and *WFA* perform optimally on 90% of inputs, and we are able to run 1,000,000,000 inputs of any length on a metric space with 10 points in reasonable time (this is not true, as our computation time increase drastically with regard to input length due to the WFA). So, wecan run all inputs of length 9 within this time. If we want to run inputs of length 10, we are only able to run 10% of the inputs. If we want to run inputs of length 12, we can only run 0.1% of the inputs! The probability that all of these inputs will land within the 90% where the algorithms perform optimally increases exponentially as our input length increases, making our bijective ratio approach 1.

Therefore, we can only start to draw interesting conclusions from the input parameters where we can run any possible input sequence.

The final interesting result is with respect to the comparison between *WFA* and *GREEDY*. The bijection in the other direction is not shown, as it is always 1 (meaning that *GREEDY* always performs as well if not better than *WFA*, bijectively). The fact that the bijective ratio shown is not 1 suggests some interesting results. If both of these bijections could be proven, then this would show that it is better to use *GREEDY* rather than *WFA* (a computationally much more expensive algorithm), across a uniform distribution of inputs. For every input where *GREEDY* performs poorly, there would be an input where *WFA* performs just as poorly if not worse.

7 Conclusion

In this work, we analyze common algorithms and performance metrics for the K-Servers problem, and provide a practical application designed to analyze the algorithms described. Future work will consist of further simulation of larger inputs, and improvement upon program efficiency. The two largest things holding back the current application are computation power and memory usage. A program designed to function with finite memory for large input sets will be implemented. Additionally, GPU parallelism will be leveraged to provide further computational improvements. The algorithms provided will hopefully be able to run as batch jobs on a super computer, allowing for analysis of longer inputs. Finally, theoretical proofs for the observations described in sec. 6 will be explored. This project will continue into an Honors Thesis, and results will be presented at the end of the 2024 Spring semester.

References

[1] A. Borodin and R. El-Yaniv, *Online computation and competitive analysis.* Cambridge University Press, 1998.

[2] S. Ben-David and A. Borodin, "A new measure for the study of on-line algorithms," *Algorithmica*, vol. 11, pp. 73–91, 2005.

[3] S. Angelopoulos, M. P. Renault, and P. Schweitzer, "Stochastic dominance and the bijective ratio of online algorithms," 2016.

[4] T. Rudec, A. Baumgartner, and R. Manger, "A fast work function algorithm for solving the k-server problem," *Central European Journal of Operations Research*, vol. 21, pp. 1–19, 01 2009.