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 5, 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-Serversproblem, 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-Serversproblem. 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-Serversalgorithms 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-Serversproblem can be described by a metric space $M = (X, d)$, a number of servers $k > 1$, and an input sequence $\sigma = (r_1, r_2, r_3, ..., r_n)$, which each correspond 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 incurrs a cost of $c = d(x, r_i)$. The goal is to have ALG incurr 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 serverst that ALG services requests. Suppose that ALG moves a server from location x_1 to location x_2, 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 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 dis-

advantage due to this, real world applications often rely on the performance of these algorithms [1]. Applications range from disk access optimization, such as the two headed-disk problem **include citation**, to **include other examples, police/firetruck service?**.

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-Serversproblem. 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-Serversalgorithms, 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 if k-competitive has been shown for any online algorithm. 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 **make sure this is true, makes sense intuitively**.

Definition 2.2. Algorithm ALG has a direct analysis ratio of c if for every input σ **why do we need here?** 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. In some sense, it is doing for the MAX/MAX ratio what direct analysis does for the competitive ratio. 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, 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. This can be demonstrated by fig. **add figure**. Here, with a request sequence that alternates between point 0 and point 1, this algorithm will continue to incur a higher and higher cost by moving server 1 back and forth. It is clear that an optimal

agorithm would bring server 2 to service point 1, and incur a constant cost of 1 for any length request of this type [1].

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 **what reference?**.

4 Software implementation details and API

Write about the software implementation, threading, how the algorithms were implemented, etc.

5 Numerical Experiments

What experiments were conducted, and the results from the experiments...

6 Analysis

6.1 Analysis

Describe the results from the data that was collected, what is interesting, different results...

7 Conclusion

Write something here...

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.