Flight Plan with Dijkstra's

Tulio Contramaestre
Undergrad Student, Computer Science
University of Central Florida
Orlando, FL, USA
tcontramaestre@knights.ucf.edu

Matthew Daley
Undergrad Student, Computer Science
University of Central Florida
Orlando, FL, USA
matthew_daley@knights.ucf.edu

Christian Vincent
Undergrad Student, Computer Science
University of Central Florida
Orlando, FL, USA
christianvincent@knights.ucf.edu

Jackie Lin
Undergrad Student, Computer Science
University of Central Florida
Orlando, FL, USA
jackie01128@knights.ucf.edu

***Abstract*—Dijkstra's algorithm is a path-finding algorithm that takes in a given weighted map of nodes and finds the most efficient path from one node to another given a set of weights and values. The programming language Java will be used for testing and optimization for this problem given its popularity and powerful multi-threading capabilities. Even though it might be considered slower than other options, it has the ability to be easily reproduced and used in other cases and problems with ease. The goal of this project would be to implement a version of Dijkstra's that takes advantage of concurrency using Java to find the most optimal path to any given node when provided with a large map of nodes and improve efficiency drastically for any sized map.**

***Index Terms*—Java, Concurrent, Dijkstra's**

I. Introduction

With the ongoing progression of multiprocessor systems within the computer industry. Multiprocessor programming has been an impactful field within modern computer systems and a rather unfamiliar subject to many in modern Computer Science. For those just beginning to learn how to program, it can seem like a confusing and difficult pathway in their journey through Computer Science. However, the benefits of utilizing concurrent threads running in parallel with one another show just how necessary and vital multithreading can be at solving a given problem. In this project, we will focus on the implementation of multithreading for a heavy-duty problem based on a real-world scenario. This topic being the estimated wait time based on air travel between stations.

The language that we will be utilizing to meet the requirements of the assignment and demonstrate a clear understanding of the topic is Java. Known for its abilities as an object-oriented language, Java possesses thousands of in built classes designed for accomplishing rudimentary solutions in a simple and easy-to-understand manner. Taking advantage of Java's inbuilt thread-related classes, we will construct and design a given real life topic that uses multiple threads running in parallel with one another to derive some solution.

A. Problem Statement

You are given a flight route map of a country consisting of N cities and M undirected flight routes. Each city has an airport and each airport can work as a layover. The airport will be in two states, Loading and Running. In the loading state, luggage is loaded into the planes. In the running state, planes will leave the airport for the next city. All the airports will switch their states from Loading to Running and vice versa after every T minutes. You can cross a city if its airport state is running. Initially, all the airports are in a running state. At an airport, if its state is loading, you have to wait for it to switch its state to running. The time taken to travel through any flight route is C minutes. Find the lexicographically smallest path which will take the minimum amount of time (in minutes) required to move from city X to city Y. It is guaranteed that the given flight route map will be connected. Graph won't contain multiple edges and self-loops. A self-loop is an edge that connects a vertex to itself.

Input: Input.txt - The first line contains one integer, n. The next n lines contain the weighted directional matrix of all of the possible paths in the graph. If a path has a weight of "-1" then there is no path from the first node to the second. Finally, the last line contains n space-separated integers corresponding to the respective wait times of each airport.

Output: Console - A map of the total time to get from each node to every other node, updated in real-time.

II. Motivation

In our research for this project, we were able to discover that the notable Flight plan problem recognized by the many few in CS had no implementation of multithreading. In addition, Java had a unique implementation and handling of multi-threading. To learn more about Java's implementation of threads and how efficiently these threads interact with the program, we decided that it would be worth investigating the behavior of threads under the well-known and highly used language Java. After finding a problem that fits our criteria, it surprised us when we found little to no usage of multi-threading despite the question's multiple requests. Seeing the potential for a

more complex and optimal problem, we thought that this would make for a great opportunity to code a program that can be more efficient with multiprocessor programming. On top of it being more efficient it would have the capability of solving larger and more dense maps, while still maintaining an efficient and fast run time.

III. Related Work

As College of Engineering and Computer Science students from the University of Central Florida, Dijkstra's algorithm was a shared part of our curriculum leading up to this class. We have each worked with and manipulated this algorithm in our academic studies. It is our hope that by using this well-known algorithm we can better enhance the performance of our code under multiple concurrent threads. Using the different methods of parallel and distributed processing from class. This project will seek to enforce our skill set in both multiprocessing and Dijkstra's algorithm.

IV. Implementation

The following is the breakdown of the tasks we completed on this project:

- Implement Dijkstras's algorithm in Java to solve the initial Flight plan problem
- Implement and incorporate multiple threads into the initial algorithm in Java
- Create a testing environment to extract performance results
- Measure performance
- Test a variety of scenarios by running tests with many uniquely made test cases to test all possible scenarios

A. Plan for Implementation

PLAN FOR IMPLEMENTATION - og desgin

B. Anticipated & Encountered Challenges

ANTICIPATED AND ENCOUNTERED CHALLENGES

1) The problem that we were referencing had inputs that weren't compatible with multi-threading solutions. To address this we created our own inputs and increased the problem's level of difficulty. Now, each terminal has its own load and unload schedule because each airport is different and runs on a different schedule.
2) Another problem we are going to face is the possibility that our time controller (that updates the wait time in live time) might not update before the threads progress through. This would cause all planes to have the maximum possible wait time for every terminal.
3) Dijkstra's algorithm was designed and meant for replicable travel results that don't change and don't look into the future. Due to each airport being on its own schedule, we have to alter Dijkstra's algorithm to account for these future wait times and it has to be accurate to the combined current wait and travel times of the flights before it.
4) Have to figure out a way to get a shared array between threads with no locks for both wait times as well as updating travel times for flight plan

C. More Challenges & Significant Changes to Design

A significant change in the design of our project was whether or not these airport nodes should be bidirectional. In other words, should these planes be allowed to travel from one destination to another and then back again. We decided to go against this idea and only allow for one-way travel between airports. This is because we believed that having bidirectional travel could further complicate our plan of action in such a way that would interfere with the goals this project set out to accomplish. Furthermore, the scenario is that we are creating a map to give some general idea of the total time it would take to reach certain destinations. We thought that it did not make much sense for someone to get on a plane to travel to one location only to head back to where they started and do it all over again.

D. Synchronization

Going into this project, the synchronization of concurrent threads would have to be one of our top priorities. Every thread needs to be able to update specific matrices according to the information provided to them. If these threads are not synchronized in their actions it would become almost impossible to dictate the wait times between different nodes. We were able to facilitate the synchronization between concurrent active threads using read and write locks, as well as, atomic integers. We also utilized synchronization among threads to help ensure that the work these threads would engage in would be evenly distributed. If one thread was already working on Dijkstra's algorithm to help calculate the distance between destinations, having another thread also work on those same destinations would only waste time and resources.

V. Testing

By nature of our problem's design, every execution time hypothetically takes the exact same amount of time every time. With that, we decided measuring the time from resetting the threads to thread completion would be the metric we used to compare the two methods.

A. Variation #1 - multiple threads taking turns with Dijkstra's

GOING TO HAVE TO COME UP WITH SOMETHING Come up with a better name for these

B. Variation #2 - multi-threaded Dijkstra's algorithm

MORE CHALLENGES

C. Testing Environment

TESTING ENVIRONMENT

VI. Evaluation

EVALUATION

A. Conclusion

CONCLUSION

REFERENCES

Please number citations consecutively within brackets [1]. The sentence punctuation follows the bracket [2]. Refer simply to the reference number, as in [3]—do not use "Ref. [3]" or "reference [3]" except at the beginning of a sentence: "Reference [3] was the first ..."

Number footnotes separately in superscripts. Place the actual footnote at the bottom of the column in which it was cited. Do not put footnotes in the abstract or reference list. Use letters for table footnotes.

Unless there are six authors or more give all authors' names; do not use "et al.". Papers that have not been published, even if they have been submitted for publication, should be cited as "unpublished" [4]. Papers that have been accepted for publication should be cited as "in press" [5]. Capitalize only the first word in a paper title, except for proper nouns and element symbols.

For papers published in translation journals, please give the English citation first, followed by the original foreign-language citation [6].

REFERENCES

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

APPENDIX

INCLUDE GRAPHS AND CHARTS HERE - Comparing v1 to v2