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 for the notable flight plan problem.

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. We will then compare the performance between the multithreaded solution to it's initial version (which in our case is the Dijkstra Algorithm) to determine if there were any improvements when solving the flight plan problem. In addition, we'll also point out the challenges and techniques we came across throughout the experiment.

A. Problem Statement

You are given a flight route map of a country consisting of N cities. Each city has an airport and each airport can work as a layover. The airport will be in a state of waiting until it is scheduled to take off. In the waiting state, luggage is loaded into the planes and will not take off under any circumstances until the wait time has ended. In the running state, planes will leave the airport for the next city. The time it takes for an airport to depart is based off its wait time which is given to us in a text file. You can go to another city if its airport is waiting and has not left yet. Initially, all the airports are in a waiting state. 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 every city X to every city Y. It is guaranteed that the given flight route map will be connected. The graph won't contain any 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 similar to the ones in Fig. 1. 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 T 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 many 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 also 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. There are many implementations that are available explaining the use and theory behind Dijkstra's algorithm and for the purpose of understanding this algorithm and its overall concept we will be using a website GeeksForGeeks [3] to better help us during this assignment. Another piece of work that relates to the concepts found in our project is the original design for the Flight plan problem [2]. This problem does not exactly match the criteria of what we aim to accomplish, but instead provides a baseline template for us to use moving forward. It is our hope that by using this well-known algorithm and problem that we can better enhance the performance of our code under multiple concurrent threads. Using the different methods of parallel and distributed processing from class, our knowledge of locks and how they behave with concurrently running threads, 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 Dijkstra'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
- Create a way to generate different inputs of varying sizes and complexity
- Measure performance by running a variety of tests with many uniquely made test cases to test all possible scenarios

A. Plan for Implementation

The plan going into this project is to have a Time controller that will update an array holding the wait times of every node. To do this a master wait time array would be needed so that once a station is done waiting and has reached a count of -1, the code can look reference the saved copy of the original input array and reset the wait time values all over again. This is so that we can ensure that the code is able to handle multiple iterations as well as different wait times per station since different planes can leave at different times.

It is also important that our code ensure that no thread has the ability to access the shared wait time array before or while it is being updated. If this were to happen, the times shared amongst all threads would be incorrect and the results of running Dijkstras's algorithm would not be accurate. With each thread being independent of one another, sharing the same wait time array, and atomic integer counters, the program should check to see if the time has been updated and then get a number to use as input in our shared travel time matrix.

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 where it previously had one static 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 a longer than accurate wait time for some terminals, leading to an inaccurate final matrix of travel times. This mutual exclusion will have to be done without utilizing a concurrent lock to avoid risking delaying the time iteration.
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 a shared matrix for updating travel times in the 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 exact time it would take to reach certain destinations. We thought 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.

Another significant change to our design and methodology would have to be how we managed and controlled our time with our time controller. Our original idea was to manage a travel time array within a completely separate thread called the time controller. All this thread would do is decrement each value in the travel time array every predetermined amount of time, resetting the value to a master copy of the travel time array once the value reached -1. We have decided that another implementation would be more beneficial and efficient while ensuring that no two threads reference a different time while running Dijkstra's algorithm. The change that was conducted was instead of maintaining an entire array of data that would need to utilize a read/write lock combination, we would create an application-wide current time. This current time would be represented by a volatile int. The time controller would constantly increment over time which would then cause the wait time value to decrement over time through the use of a formula that calculates the time using future times and distance in each of the threads running Dijkstra's algorithm. With this change, we could no longer use the original method for finding future wait times and instead, we had to use the one as follows:

D. Future Wait Times

A standard implementation of Dijkstra's algorithm neither includes any wait times between any two nodes nor a way to see what the wait time will be at the time the algorithm arrives at each node. With that said, we had to create our own way to both track wait times respective to each node as well as a way to know how far into the future to find its wait time. To do this, we had to break Dijkstra's algorithm down to its fundamental steps: find the next minimum current travel time then check the rest of the nodes that haven't been picked yet and see if going through that node is faster. In our case, all we had to modify was that second step. If we know some node will pass through some other node, all we have to do is add some future wait time to the comparisons when looking for faster paths. The equation we came up with is as follows.

$$T - ((C + F)\%T)$$

Where
T = turnTime[node]; node is an input
C = currentTime; the application-wide volatile int
F = futureTime; due to the nature of Dijkstra's is represented by time[node]

This equation works by subtracting the amount of relevant time passed from the wait time where the relevant time is determined by adding C to F to get the time in the future relative to the current time and modding that by T. This quantity is then subtracted from T to get the future wait time of that node with reference to the current time.

E. 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 synchronized objects, atomic and volatile integers, and specific threads dedicated to the management of shared resources. To assist in the synchronization of wait time amongst all threads and ensure that these threads do not interfere with one another, we created a time controller thread that would manage the time values across all nodes. This is important because if a nodes wait time is not reset or updated, the threads running Dijkstra's algorithm can not correctly assign values to the matrix. However, one of the most important parts of our synchronization process, had to be use of the volatile int. Not only did this help save us considerable time in developing the time controller but it also gave us a reasonable way of simulating the current time of the overall project. Before using the volatile int, the current time of the project would sometimes not match what was intended and would either be delayed not updated at all.

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. Multi-threaded implementation of Dijkstra's algorithm

For our multi-threaded implementation of Dijkstra's algorithm, we decided to primarily use an atomic counter to be able to assign each thread to a specific starting terminal or node. This allowed for each thread to be able to find the shortest path from its assigned node to the other nodes using Dijkstra's shortest path algorithm. This would guarantee mutual exclusion and allow for the threads and our implementation to run without the need of locks. It ended up working similarly to a divide and conquer approach.

It worked as such, firstly a graph would be randomly generated using the algorithm we created. After the graph is generated each thread would be assigned a value using an atomic integer that increments after each thread has grabbed their number or node in this case. Each thread would then perform it's operations on the give node using the Dijkstra's method we implemented and from there it would add its result to the final graph that contains all the distances from one node to another.

This first variation of the multi-threaded implementation was using a time controller that would decrement the time per station since each station had a independent wait time on top of the distance that is being travelled to it. This was causing some issues when testing since there were problems were the

time would decrement before the distance was calculated and would lead to problems with the final time travelled. To help solve this we implemented a separate thread that would control the time and created a formula that would calculate the future times using the value from the time controller thread.

B. Controllers

For our method of solution we had one main controller we called the Time Controller. This controller kept track of the current world time. It handles the increment of the time clock that we have in place for predicting the travel time along with handling how the station wait times are lowered. Originally the time controller lowered the wait time of each terminal or node, however this caused some issues as stated earlier. Therefore changes had to be made onto how we operated the time controller and dealt with managing time. A separate thread being allocate to increment time instead of decrement the station wait time was what we decided was best for us to use. It fixed mutual exclusion and synchronization issues along with helping prevent race conditions that could happen between our threads. It also guaranteed a safe and efficient way to predict the future time and total time that would be calculated from finding the shortest path using Dijkstra's algorithm.

C. Testing Environment

While collecting data for our solution, with the utilization of Visual Studio as our IDE, we have tested several machines over different operating systems. Originally, we started collecting data on a Windows 11 machine with an AMD Ryzen 7 3800X 8-Core Processor clocked at 4.20 GHz and 16 gigabytes of RAM. This computer however produced inconsistent results, running our algorithm too fast to record any meaningful data at smaller inputs. To address this issue, we switched operating systems and the chipset to Apple Silicon, specifically an 8-core M1 MacBook Pro with 8 gigabytes of unified memory. This machine produced meaningful results and actual data we could compare.

VI. Evaluation

A. Results

After performing intensive research on multiple graphs with varying sizes and with varying amounts of threads we concluded that multiple threads do improve performance for this task.

Testing was done on graphs of size 10, 50, 100, 500 and 1000 airplanes. As shown by the line graph Fig 2, a single threaded method of performing Dijkstra's performed optimally up until around 100 airport after which it drastically slowed down compared to 2 threads 4 threads and especially 8 threads. We can also see from Fig 2 that the time increased exponentially as more terminals were added to the matrix. However 8 threads always maintained a consistent tread and only slightly decreased in performance has we increased the amount of terminals that were being calculated by the algorithm. This helps to show the benefits of multi threading Dijkstra's.

If we look at the data more in detail Figures 3, 4 and 5 we can actually see that a single threaded implementation of the algorithm performs drastically better than an 8 threaded implementation of the algorithm on a small group of data such as 10 terminals. However as we begin to increase the data pool the opposite begins to be true we are able to see that the single threaded operations start to slow drastically while multi threaded operations improve the more data is being handled. In fact as we can see in Fig 5, the relationship between 1 thread and 8 threads flip with 8 threads now being drastically faster than a single thread solution.

As proven by our testing we can see that a multi threaded solution does benefit the performance of Dijkstra's especially when dealing with large amounts of data. Even though a single threaded operation performed a lot better than an 8 threaded method when dealing with smaller amounts of data such as 10 terminals 8 threads completely outperforms it in the 1000 terminal test.

B. Conclusion

Following the implementation of the Dijkstra's algorithm in our program, we were able to utilize multiple threads running concurrently with each other to achieve our desired result. Our program has shown a significant reduction in run time and overall better performance than compared to the standard Dijkstra algorithm approach that only uses one thread. This basis is backed by our performance testing throughout our java structured program that was experimented under multiple trials to test a variety of different possible scenarios. This has shown numerous cases of faster run time for our multi threaded program execution for almost all the given cases when compared to the single thread implementation of Dijkstra's. We were able to observe from the data that as the number of airports added to the system increased, so did the effectiveness of having multiple threads running Dijkstra's as seen in the evaluation when the airport count reached and exceeded 50. As such, our research has clearly shown the significant impact of multi thread coding towards high request problems such as the flight plan problem. It can also be noted that the usage of multi thread programming is quite rare for our following problem, as it is more commonly approached with only one thread using Dijkstra's algorithm. By demonstrating this, it is our hope that others may seek to improve or enhance more commonly used problems in Computer Science like we have to better help their own understanding of the material.

C. Future improvements

Some future improvements that could be made to the current implementation of our multi threading algorithm would be to have a more improved version Dijkstra's. Currently we are using a very simple implementation of Dijkstra's algorithm that was only slightly modified to fit the needs of a multi threaded implementation. Therefore researching and further development on Dijkstra's would drastically improve performance and run time along with possibly improving data and

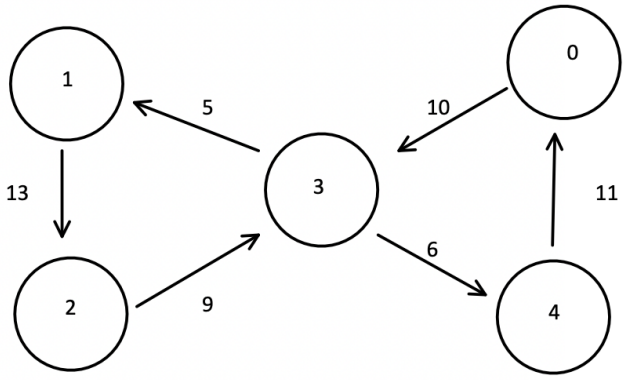

Fig. 1. Sample input

thread usage since currently we are using a thread for time management.

Another improvement that can be made would be the usage of a more improved method of storing data and how the shortest paths are stored in the matrix. This needs further development due to after testing we were achieving size errors when trying to run the data on larger inputs such as 40,000 airports. The input text files generated when creating these large test cases would, in some cases, be holding gigabytes worth of information concerning the construction of the given scenario. It may be worth looking into a program that can compress data or rearrange it in such a way that the file size becomes considerably smaller and still usable. However, for now we believe going that far may be beyond the scope of the assignment and are choosing to stick with a normal text file.

REFERENCES

[1] He, Mengqing, "Parallelizing Dijkstra's Algorithm" (2021). Culminating Projects in Computer Science and Information Technology. 35.

[2] A. Sharma and S. Gupta, "The Flight Plan: Practice Problems", HackerEarth. [Online]. Available: https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/practice-problems/algorithm/traffic-light-2-ee27ba45/.

[3] GeeksforGeeks, "Find shortest paths from source to all vertices using Dijkstra's algorithm", GeeksforGeeks, 28-Mar-2023. [Online]. Available: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/.

APPENDIX

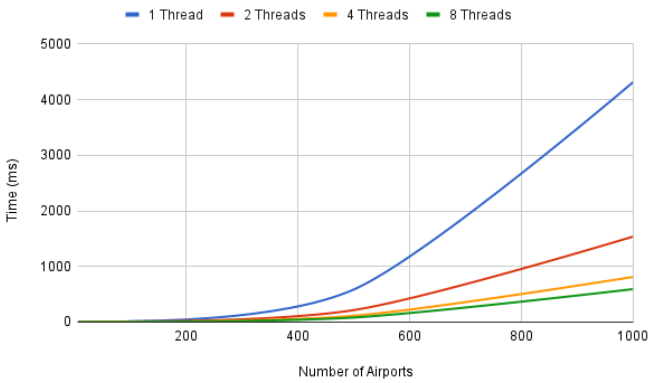

Fig. 2. Line Graph

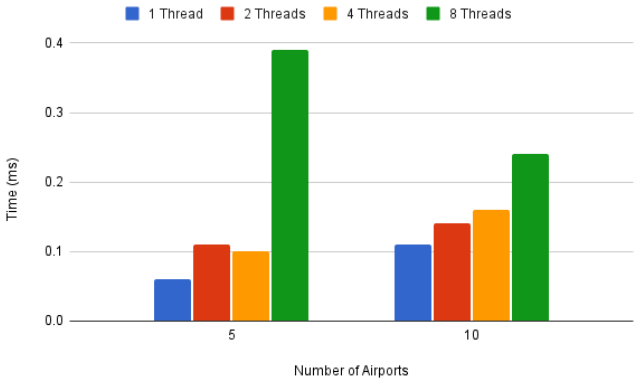

Fig. 3. Bar Graph

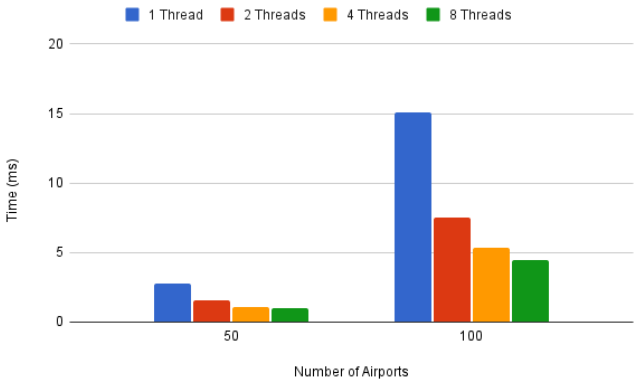

Fig. 4. Bar Graph 2

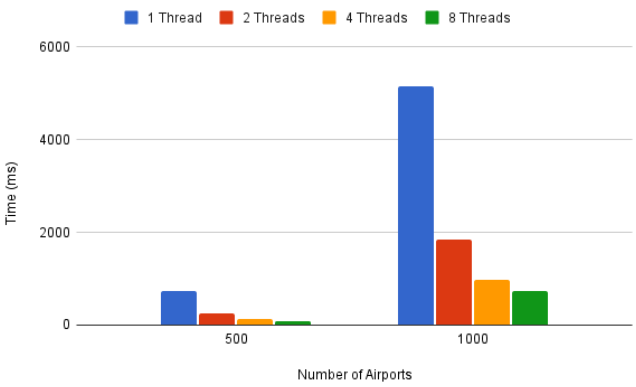

Fig. 5. Bar Graph 3