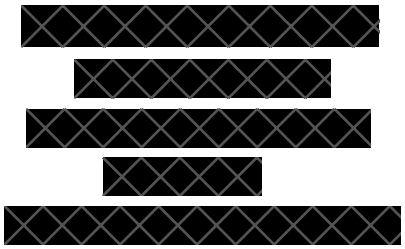
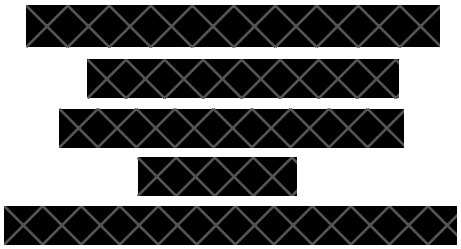
Flighting Agenda Using Single-Source Shortest Path
Final Project Report





Abstract

In this paper, we will be presenting our final progress for the Single Source Shortest Path problem with the inclusion of a flighting agenda. We explain what our accomplishments have been and how we faced different problems and worked as a team to solve those issues. This paper also includes pseudocodes, diagrams, and figures to support our topic and its implementation.

Research Problem & Project Description

Our research problem was to understand what was the single-source shortest path problem and why it is one of the very important problems in computer science. Along with understanding this problem, we wanted to implement a flighting agenda using the parallel Dijkstra's algorithm in order to find the best flights from a source to multiple destinations.

The single shortest path problem was designed to determine the shortest distances from one vertice to another. These paths are known as edges. Each edge has a weight (which can be deciphered as a distance or price cost). There have been multiple algorithms built in order to solve this problem but the one that has been used to implement the flighting agenda along with the single shortest path problem in this paper is parallel Dijkstra's algorithm. It is a must that the edges must be non-negative.

Background

Relaxation: Relaxation is used in Shortest Path Algorithms. Consider an undirected graph where the distance of each vertex 'v' is determined by d[v] and the cost of each edge between vertex 'u' and vertex 'v' is c(u,v). If the value (d[u]+c(u,v)) is less than d[v], d[v] is updated to d[u].

Dijkstra's Algorithm: This algorithm uses relaxation techniques in order to calculate the shortest distances/costs between the vertices in a graph which can be used to implement many applications. At first, the distance to all vertices from a single source vertice is marked to be infinity, and as the algorithm progresses, these distances are replaced by the relevant value of the shortest path to the destination.

Initially, we have to perform relaxation on the vertices connected to the source vertex and push the source vertex into the shortest distance vertex set. Then we select a node 'u' with the least distance value d[u], push the node u into the shortest distance vertex set, and perform relaxation on the vertices connected to the node u. This technique is continuously applied to the graph until all nodes are pushed into the set of shortest distances.

Flighting agenda: This extra functionality is added to our project which allows users to find the shortest costs for their flights from a source to multiple destinations. Each vertice is representing a source or a destination whereas each edge is representing the path from said source and destination. Each edge has a weight that in normal terms is referred to as price cost. Parallel Dijkstra's is performed on this graph to give the fastest arrival time of a flight at the destination location.

Related Work

We were unable to find other small-scale projects that implemented the flighting agenda using the Parallel Dijkstra's Algorithm but we did compare our algorithm with projects that were just about the Dijkstra's algorithm. Ultimately, we found out that the time complexity for our algorithm was similar to the ones we compared them to. At first, we did not expect our algorithm to be efficient but as we kept working on it and implemented the parallel dijkstra's algorithm, we were able to achieve a better time complexity.

As we noticed with other projects, over time the developers realize that the project is more complex than they had thought before, however, they also find out that the algorithm they build is better than the previous iterations. Similarly, once we implemented the non-parallel Dijkstra's algorithm, we realized the parallel was much more complicated, but we did create much better algorithms in later iterations. At first, our approach was to just use the research paper to find our solution but later on, we changed our method and scoured the internet for better resources, which ultimately did prove to be a better idea as building this algorithm became much easier than before.

Implementation

Below is the pseudocode for our implementation of the parallel Dijkstra's algorithm. This does not include the actual C++ code with the flighting agenda. That code is included in the submission folder.

Pseudo Code for Parallel Dijkstra's Algorithm:

The pseudo code for Parallel Dijkstra's Algorithm is as follows:

```
Function Parallel_Dijkstra's(l[u][v], int source):

int d[]=Arrays.fill(INT_MAX,N);

d[source]=0;

set<integer>visited=new set();

Divide the adjacency matrix into P parts.

While(!visited)

{

//In each process.

    while(visited.size()<N)

    {

            int u =(!visited)min(x->d[x]);

            visited.add(u);

            for (v->vertex adjacent to u)

            {

                    if(d[v]>d[u]+l[u][v])

                    {

                            d[v]=d[u]+l[u][v];

                    }

            }

    Local minimum= d[v];

MPI_Alltoall ( local minimum );

}

Global minimum=min{local minimum};

d[v]=visited.

// Here v is the vertex from which the global minimum is found.

}
```

The same Dijkstra's Algorithm is followed in each process. Once the local minimums are found they are broadcasted to all the processes. The global minimum is found and the vertex is marked visited.

Challenges & Solutions

At first, we read the research paper, which was too wordy, more than what we thought was necessary. Although it was helpful for us, at times we found information which was redundant and not really of any use. However, even while facing this problem, we did most of our research online and found some good sources to help us come up with this algorithm and implement the flighting agenda.

One of the most important parts of this project was to use the MPI library in order to be able to make the algorithm run parallely on multiple processors. At first, we were struggling with finding resources for this library and how it is supposed to be installed on one's computer, and how it could be run using Visual Studio but ultimately we found some YouTube videos that helped us solve this problem. The instructions on how to run this algorithm are in the attached ReadMe file.

Using the MPI directory

MPI is a short form for Message Passing Interface which is used for parallel programming. This directory can be used in multiple programming languages and it allows to run an algorithm using multiple processors/computers. This works by allowing multiple processors to exchange information amongst themselves and let the program run much more efficiently than if not used.

It was crucial to use the MPI directory in order to be able to run our algorithm parallely on multiple processors with one goal - to achieve a better time complexity. To use this directory, there are a few steps. Firstly, we need to install MinGW and Visual Studio. After that, we need to configure MinGW into Visual Studio. It is a must to have the desktop development with the C++ tool installed. We also have to include necessary directories and then also add MinGW to

the environment variables. Once all that is done, run the debugger and test if it is working. After that, run the command “mpiexec -n 5 project_mpi.exe” to run the algorithm and get the output.

Usage of Google Cloud

We were able to run our algorithm on the Google cloud in order to make people run this algorithm if they are not able to install the required libraries and use visual studio to run this algorithm on their computers. To run it on the Google cloud, create a new project, and then create a new Virtual machine instance named as “e2-standard-4”. Once that is done, install the mpi library on the virtual machine by using the command “apt-get install lam4-dev”. Also, run the command “sudo apt-get install lam-runtime” and “lamboot”. Create a new CPP file and paste the provided code from the zip file into the new CPP file. Once that is done, compile the code by using the command “mpicxx -o source project.c“. After that, run the code by typing in “mpirun -np 5 --oversubscribe ./source” and get the output. Below is a screenshot of this algorithm being run on the Google cloud.

```
Time taken by function: 4057052 microseconds
bhavyala4@instance-1:~/cpp$ mpirun -np 5 --oversubscribe ./source
The available sources are:
1. Doha
2. New York
3. Dubai
4. London
5. Madrid
6. Istanbul
7. Karachi
8. Los Angeles
9. Houston
10. Mumbai
Enter the source: Doha
The shortest distance from the source Doha to all the other destinations are:
Doha 0
New York 16
Dubai 5
London 12
Madrid 15
Istanbul 7
Karachi 5
Los Angeles 19
Houston 19
Mumbai 4

The paths are:
New York:Doha->New York

Dubai:Doha->Dubai

London:Doha->London

Madrid:Doha->Mumbai->Madrid

Istanbul:Doha->Istanbul

Karachi:Doha->Karachi

Los Angeles:Doha->Los Angeles

Houston:Doha->Dubai->Houston

Mumbai:Doha->Mumbai

Time taken by function: 4757812 microseconds
Time taken by function: 4771286 microseconds
Time taken by function: 4761583 microseconds
Time taken by function: 4775785 microseconds
Time taken by function: 4768264 microseconds
bhavyala4@instance-1:~/cpp$
```

Figures/Diagrams

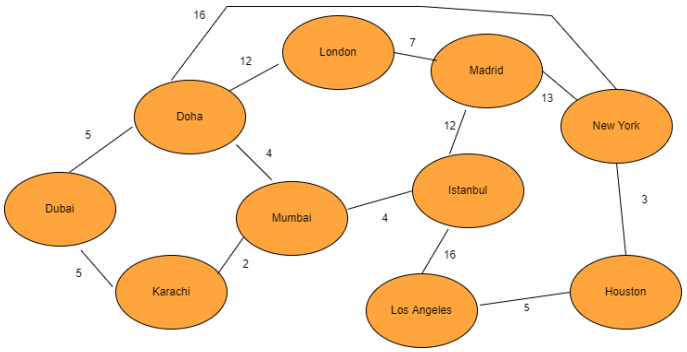


Fig 1: This figure is an example of the dataset we can use on this algorithm to output the best flights from a source to a destination. This kind of data is supposed to be inputted as a Matrix in the actual code.

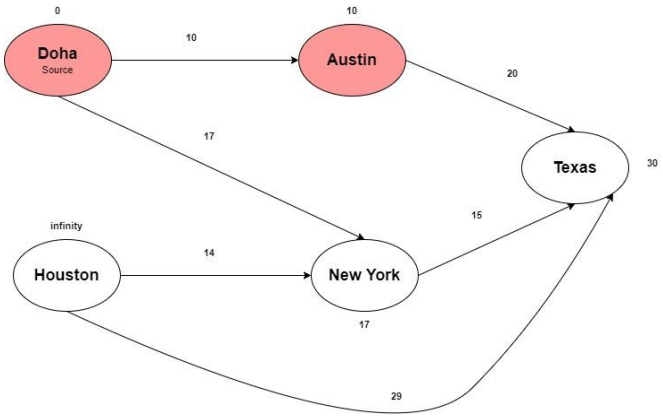


Fig2: Another example data for the flighting agenda.

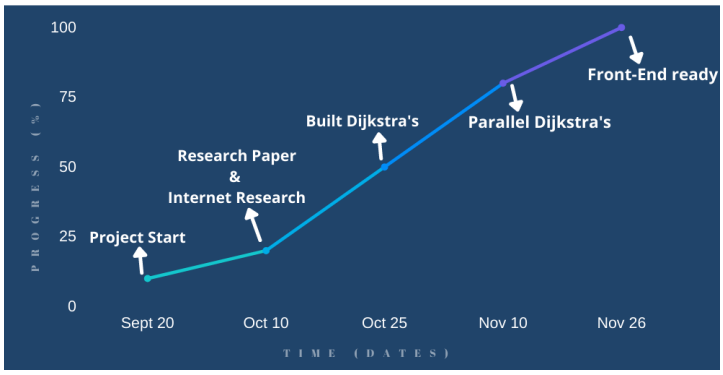


Fig 3: This figure shows the progress of this project and how the time was spent working on this project.

Comparison to Related Works

As mentioned above, we were able to compare our parallel dijkstra’s algorithm to the non-parallel algorithm and we proved that the time complexity for our algorithm was better than that of a non-parallel algorithm.

For each process, the run time required to find the local minimums is $O(V^2/P)$. The run time required to find the global minimum is O(Vlog(P)). Therefore the total run time complexity is $O(V^2/P+Vlog(P))$. This is much better than the performance of normal Dijkstra's Algorithm which is $O(V^2)$. Below are the figures that show how the time complexity changes with multiple processors.

```
The available sources are:
1. Doha
2. New York
3. Dubai
4. London
5. Madrid
6. Istanbul
7. Karachi
8. Los Angeles
9. Houston
10. Mumbai
Enter the source: Doha
The shortest distance from the source Doha to all the other destinations are:
Doha 0
New York 16
Dubai 5
London 12
Madrid 15
Istanbul 7
Karachi 5
Los Angeles 19
Houston 19
Mumbai 4

The paths are:
New York:Doha->New York

Dubai:Doha->Dubai

London:Doha->London

Madrid:Doha->Mumbai->Madrid

Istanbul:Doha->Istanbul

Karachi:Doha->Karachi

Los Angeles:Doha->Los Angeles

Houston:Doha->Dubai->Houston

Mumbai:Doha->Mumbai

Time taken by function: 31180111 microseconds
```

Fig 4: Time complexity of non-parallel Dijkstra's algorithm when run on a windows core i-7 computer with 16GB ram.

```
C:\Windows\System32\cmd.exe
Los Angeles 19
Houston 19
Mumbai 4

The paths are:
New York:Doha->New York

Dubai:Doha->Dubai

London:Doha->London

Madrid:Doha->Mumbai->Madrid

Istanbul:Doha->Istanbul

Karachi:Doha->Karachi

Los Angeles:Doha->Los Angeles

Houston:Doha->Dubai->Houston

Mumbai:Doha->Mumbai

Time taken by function: 2569538 microseconds
Time taken by function: 2657427 microseconds
Time taken by function: 2622680 microseconds
Time taken by function: 2599929 microseconds
Time taken by function: 2686275 microseconds
```

Fig 5: Time complexity of parallel Dijkstra's algorithm when run on a windows core i-7 computer with 16GB on 5 processors.

Evaluation

Gladly, we were able to successfully implement the parallel Dijkstra's algorithm with the inclusion of a flighting agenda. The main highlight of our algorithm was the flighting agenda as we were able to show a real-life usage of the Dijkstra's algorithm. And by using the MPI library, we could solve this algorithm by running it parallelly.

Conclusion

In conclusion, we are proud of the work we have done. The most important thing was that we understood how and why the Dijkstra's algorithm was created. We are certainly glad that we were able to run this algorithm parallelly on multiple processors to achieve a better time complexity.

References

[1] GeeksforGeeks, 2021. Dijkstra's shortest path algorithm | Greedy Algo-7 Geeks For Geeks.[online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>[Accessed 7 May 2021].

[2] GeeksforGeeks, 2020. Applications of Dijkstra's Shortest Path Algorithm- GeeksforGeeks. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/applications-of-dijkstras-shortest-path-algorithm/> [Accessed 5 May 2021].

[3] Saeed Maleki, Donald Nguyen (2016). DSMR: A Parallel Algorithm for Single-Source Shortest Path Problem. : Department of Computer Science, the University of Illinois at Urbana-Champaign: Microsoft Research. DOI: https://iss.oden.utexas.edu/Publications/Papers/ICS2016.pdf

[4] Github. Parallel MPI Dijkstra's. [online] https://github.com/Lehmannhen/MPI-Dijkstra