Evaluation of Euclidean Shortest Path, Voronoi Diagram and Line segment intersection using MASS, Spark and MapReduce

Richard Ng

Term report of work done as CSS595

Master of Science in Computer Science & Software Engineering

University of Washington, Bothell

Spring 2022

Project Committee:

Munehiro Fukuda, Ph.D, Committee Chair

Kelvin Sung, Ph.D Committee Member

Geethapriya Thamilarasu, Ph.D, Committee Member

1. Introduction

The purpose of this project is to develop,implement and evaluate efficient algorithms to solve geometric programming problems using Multi-Agent Spatial Simulation (MASS), Spark and MapReduce. In this project, the main focus is to solve the Euclidean Shortest Path in parallel using Spark, MapReduce and MASS. Unlike MASS, Spark and MapReduce had proven their advantages in processing big data in a data-streaming fashion. MASS, as a relatively new parallel-computing library, provides a friendly interface to utilize threads and processes to solve computational problems in parallel to maximize the execution performance. It provides a different computational framework that dispatches agents into a structured dataset and finds its shape and attributes in their emergent, collective group behavior. Compared to Spark and MapReduce, which are text/plain data analysis oriented, MASS has outstanding performance on analyzing data structures with agents. This advantage becomes more beneficial as data formats are diverse nowadays. More statistical support is needed to prove its efficiency to attract potential stakeholders. In this project, the comparison of these three libraries will be evaluated based on the various geometric problems.

2. Progress

In the last quarter, I have implemented trapezoidal decomposition using Spark. The method is based on Divide and Conquer along with dynamic global variables to help to compute the transformation from point data to trapezoid data.

In this quarter, the MapReduce version implementation is done using the similar approach from Spark. For the MASS version, a different approach is implemented to achieve the goal of ESP, agent propagation. The details are given in the up-coming section.

3. Method

3.1 Overview of trapezoid decomposition using Slab decomposition

Assuming there is a wrapped area that contains multiple obstacles. An example of the area is illustrated in Figure 1. The goal is to find the shortest path from the S location to the D destination. It is not that difficult to find the solution with a small area and few obstacles just by observation. However, if the area is hundred times larger and the obstacles are hundred times more, it becomes a complicated problem. It is known that the ESP is one of the solutions to find the shortest path among many other possible paths. The ESP approach works when there is data given. For example, the point data and their neighbor relationship. In our case, those data are missing. To build the data by only the obstacles information, trapezoid decomposition can achieve it.

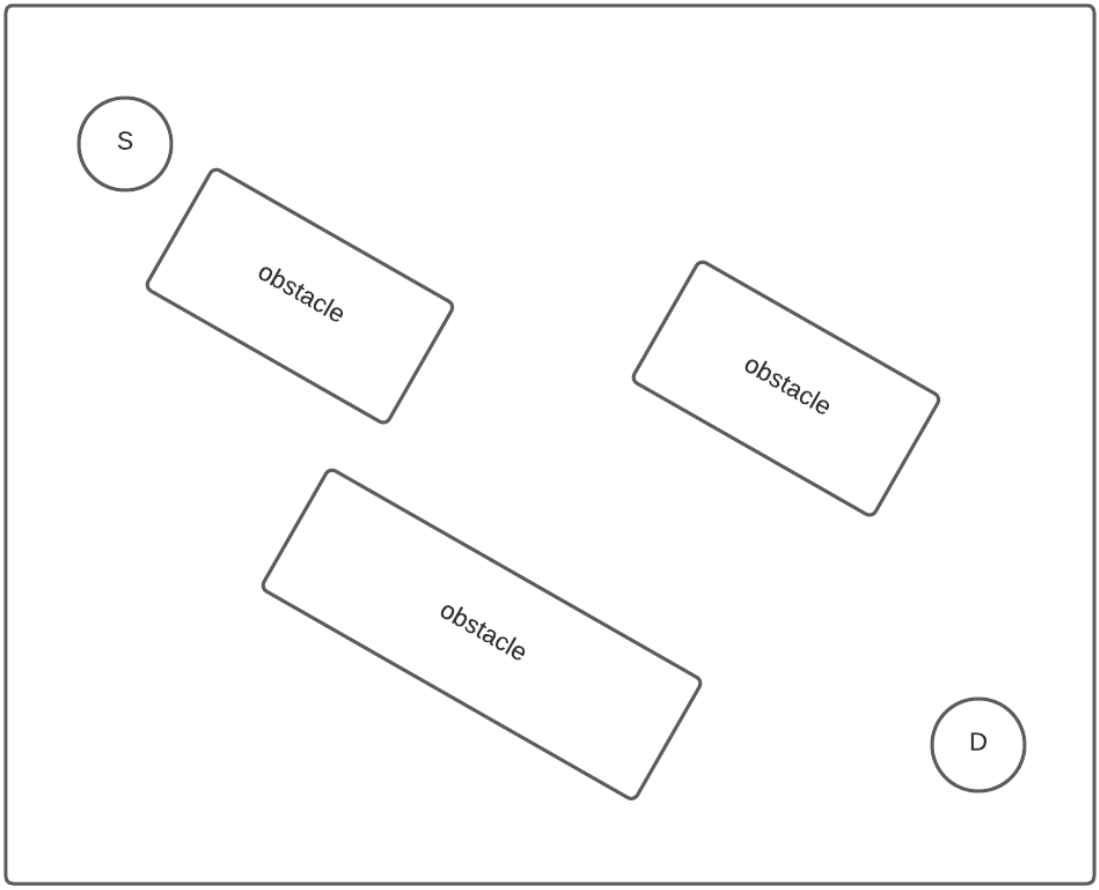

Figure 1: A area that contains obstacles

One of the trapezoidal decomposition approaches is slab decomposition. It places vertical lines on every vertex of the obstacles. The result is shown in Figure 2. The area is then divided by several slabs by these vertical lines. In each slab, the two vertical lines from the right side and the left side serve as its boundary. The other line segments are from the obstacles that are overlapped with the slab boundary. Figure 3 shows one of the slabs from the decomposition. By observation, each slab contains an even number of the line segments. The next step is to sort the line segments in ascending order by its Y-axis. Then, use an ordered pair of the line segments and the boundary to form a trapezoid. The final product is shown in Figure 4.

For building the neighbor relationship of the trapezoid map, there are two methods. The more straightforward one is to iterate all the trapezoids and check if any two of the trapezoids share the same edge. If there is, they have a neighbor relationship. The other method is to iterate each slab and pick the two nearby slabs, then use a two

pointers approach to check if the trapezoids share the same edge. The second approach seems to be much faster than the first one as the number of slabs and the number of trapezoids insides the slab are always less than the total number of the trapezoids in the area.

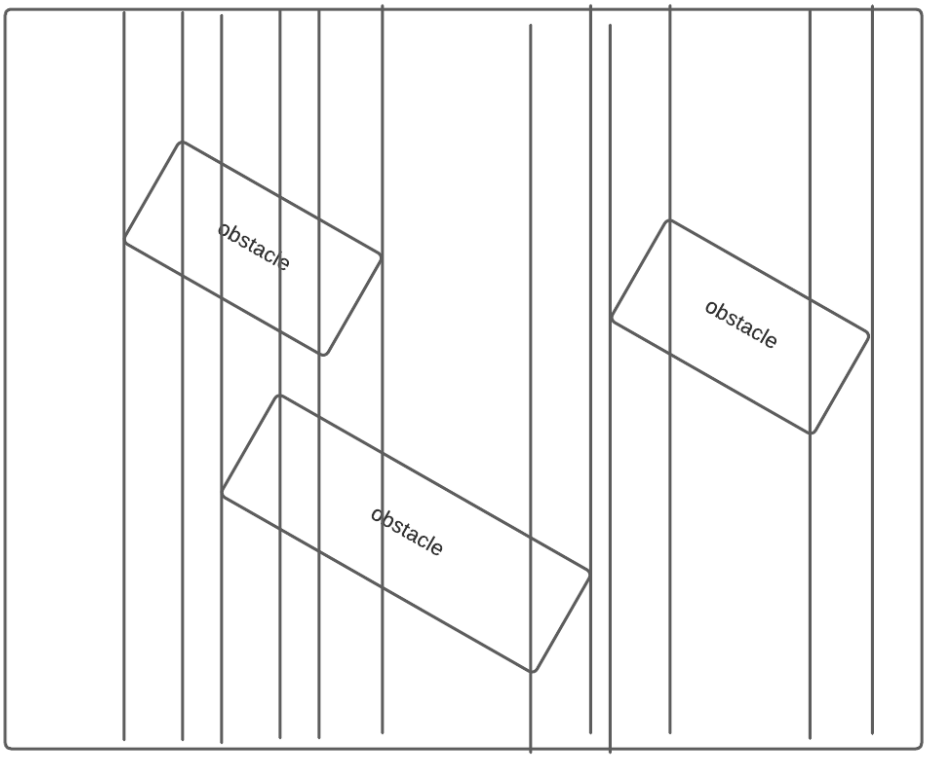

Figure 2: A updated area after the first step of the Slab decomposition

Figure 3: A slab

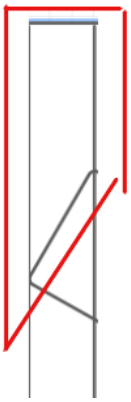

Figure 4: A trapezoid formed from the slab

3.2 Divide and Conquer concept in Slab decomposition

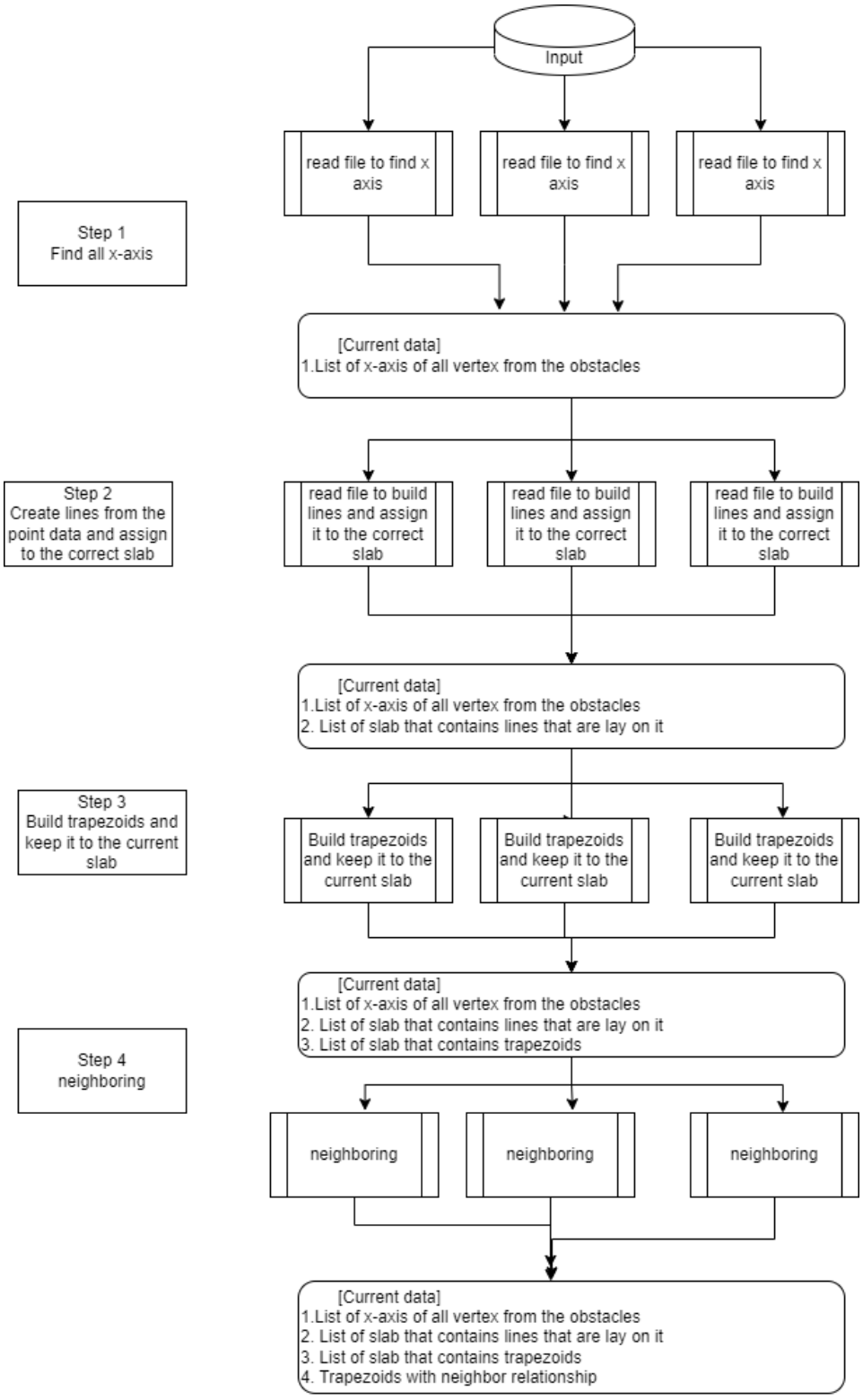

Figure 5: Divide and Conquer

The efficiency of slab decomposition can be improved by parallelizing partially using the divide and conquer approach with Spark and MapReduce. Figure 5 illustrates the major steps to build the trapezoid map from the point data file.

Step 1: Read the file in parallel, get the x-axis from the point data. Sort the x-axis in ascending order. Output the x-axis list and use it as the slab boundary.

Step 2: Read the file again. Build the line segments data from the point data.
Check if the lines reside in any slab boundary. Add the lines to the corresponding slabs if they reside in the boundary. Output the slab with lines set.

Step 3: Transform the slab with lines set into trapezoids. The approach is mentioned in section 3.1.

Step 4: Build the neighbor relationship among all the trapezoids The approach is mentioned in section 3.1.

Spark and Mapreduce share the same concept as above with little different techniques to handle the shared variables.

3.3 Trapezoidal decomposition (Spark)

The complete flow chart is shown in figure 6. The flow starts from reading the file in parallel with the built-in method from Spark. It then transforms the data point into line segments. During the transformation, all the unique X-axis are stored into an accumulator for future uses. While having all the line segments ready, the next step is to assign them into their corresponding stripes. By having the X-axis stored as a global variable, the method can now fetch the X-axis data to build the range of a stripe. Line segments that reside in between the boundary will be added to that stripe. As the stripe class includes the left X-axis and the right X-axis, the boundary, and all the line segments residing in between it, we can build a trapezoid using these data. To build a trapezoid using the above data, we sort the line segment in ascending order by its Y-axis, and pick the first two. Using these two line segments and the left and the right boundary, we are able to figure out the four corners of a trapezoid. Building four lines using these four corners will get a proper trapezoid. There is one more global variable that stores a copy of all trapezoids. The copy is used to establish the neighbor relationship for all the trapezoids at the last step.

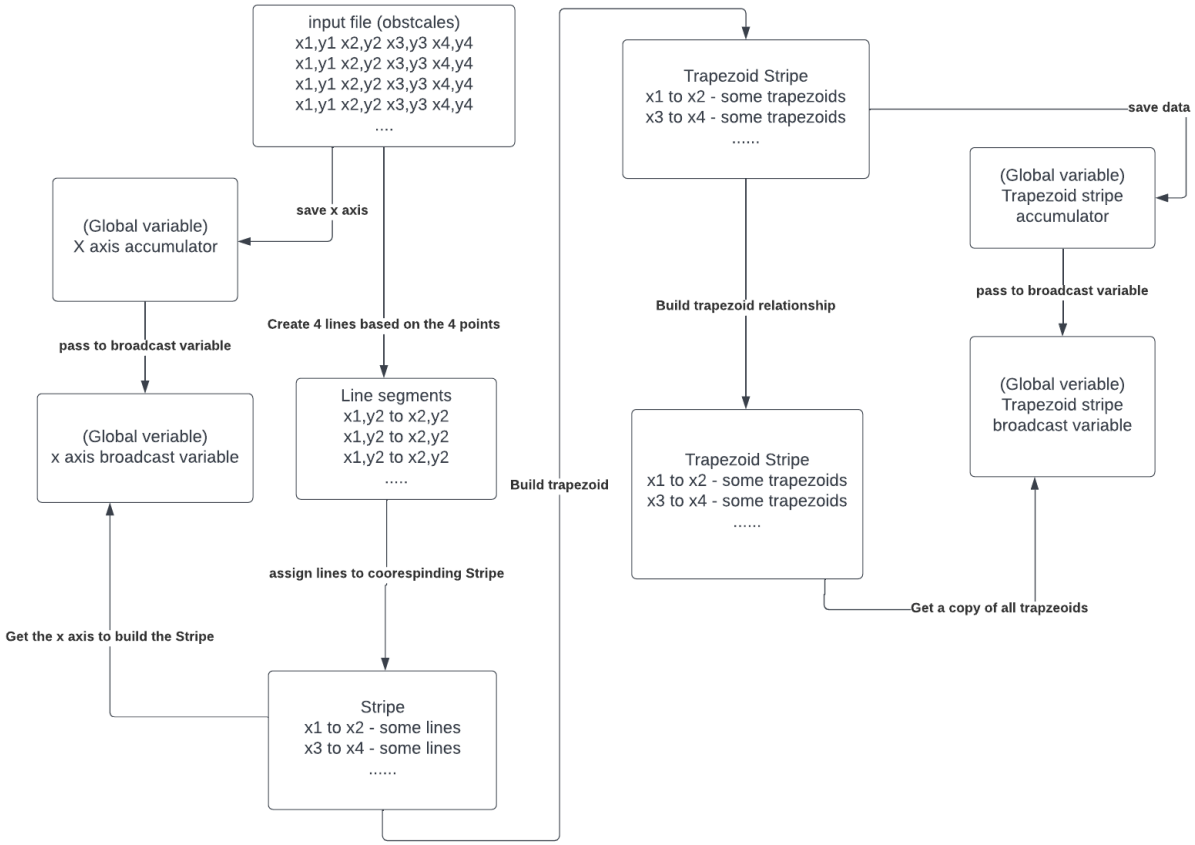

Figure 6: Flow chart of the slab decomposition (Spark version)

3.4 Spark global variables

For the needs of reusing post-processing data from former RDD, global variables are used to achieve the goal. In Spark, there are two different global variables - accumulator and broadcast variable. Accumulator is a shared variable across all computing nodes and it is write-only. On the other hand, a broadcast variable is a shared variable that is read-only across all computing nodes. The main purpose of it is to distribute a large dataset from the driver to the executors efficiently. In the algorithm I used, there is a need for dynamic global variables. The global variables cannot be determined before some RDDs are processed. Therefore, the accumulator has to wait for the certain RDDs to process, and add the related data to it. After that, the driver program will be able to broadcast this data to all other executors for the further calculation. The logic is shown in figure 7.

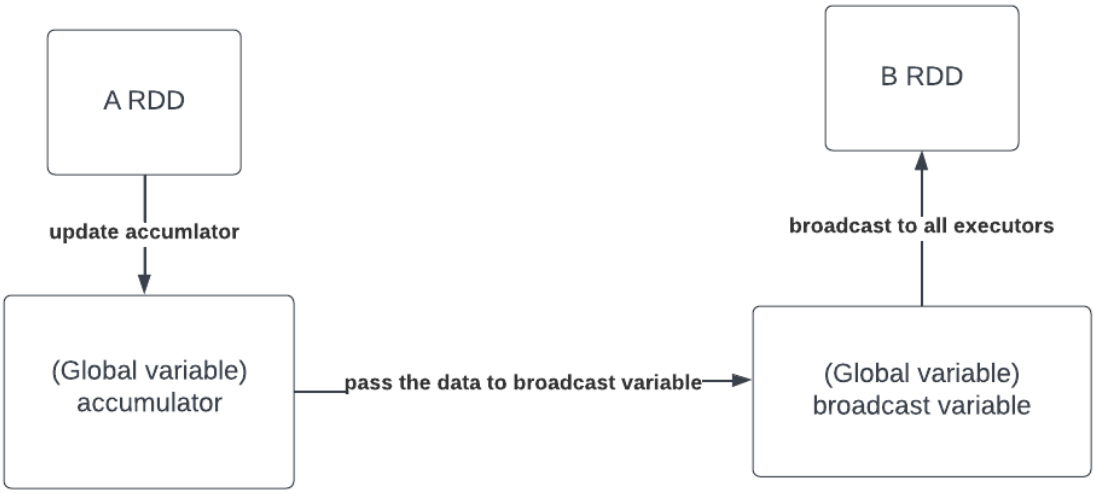

Figure 7: Utilize accumulators and broadcast variables as dynamic global variables

3.5 MapReduce global variables

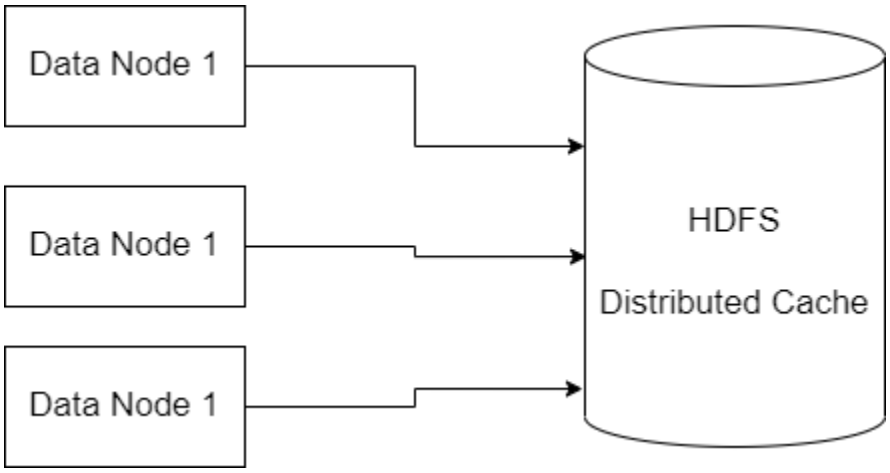

Figure 7: Distributed cache in HDFS

In HDFS, there are two methods to fetch shared variables.

1. Store the shared files into HDFS, fetch the file each time before mapper starts to work.
2. Store the shared files into a distributed cache, and assign them to each data node. The model is shown in Figure 7.

The first technique directly fetches the file when a data node needs the file. This process wastes too much network bandwidth since each communication between the data node and the driver consumes some bandwidth.

The second one sends a copy of the file to all the data nodes. The file is then marked as a local file. Therefore, when reading the file, it does not consume any bandwidth.

3.6 MASS Euclidean Shortest Path with agent propagation

Instead of working on trapezoidal decomposition above, MASS can directly find the shortest path from the source to the destination by propagating agents. Here are the steps.

Step 1:
Initialize a 2-D array as the area, mark the obstacles in the array. Figure 8 is an example of places with some obstacles. For the sake of simple illustration, it is assumed the obstacles are in square shape and are marked in red color. To calculate the obstacle area, I refer to an explanation about point location inside a polygon that is made by “Paul Bourke”. It is said that if there is a ray emanating horizontally to the right from the test point, if the number of intersections with the polygon is even, then the point is outside the polygon, vice versa. The demonstration is shown in Figure 9. Using this theory, the obstacles polygon can be marked correctly on the 2-D array;

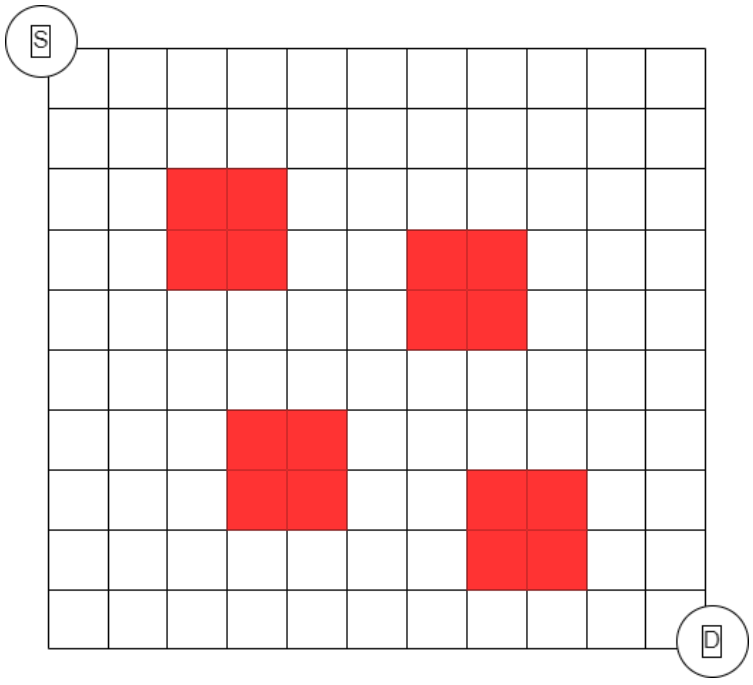

Figure 8: Places initialization

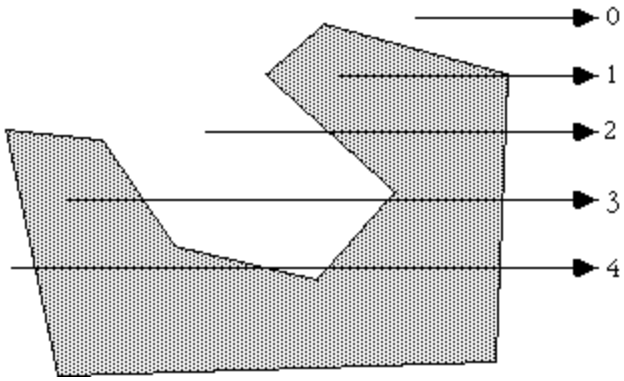

Figure 9: A demonstration of point lies on the polygon from “Paul Bourke”

Step 2:
Place the starting agent onto the starting point

Step 3:
Start propagating the agent.

There are two way of propagation: Von Neumann pattern and Moore pattern

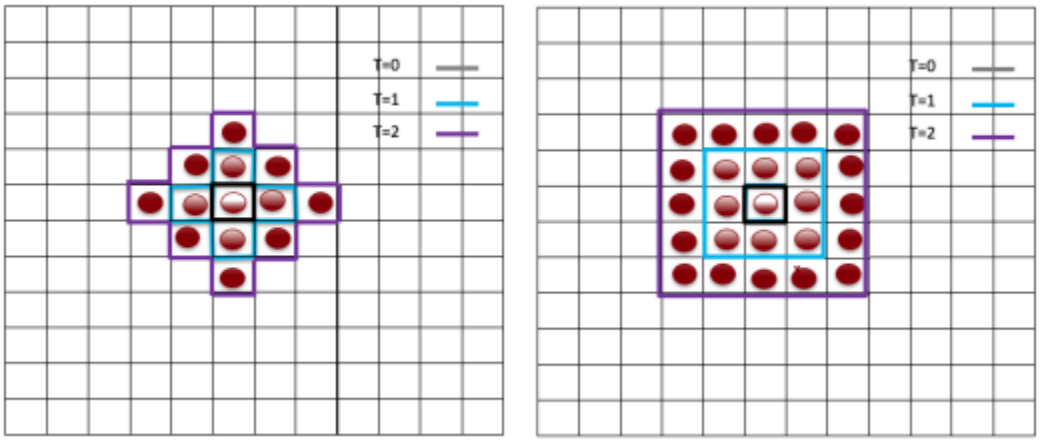

Figure 10: Von Neumann and Moore agent migration at time intervals 0, 1, and 2. [4]

Both patterns work as eventually they will visit almost all the places.

Instead of only propagating using either of the above patterns, when agents hit a vertex of an obstacle, it propagates along the edges to the remaining vertex from the same obstacle. The concept is shown in Figure 11. When the agent (green) hits the left vertex of an obstacle, it spawns three other agents (blue) on the remaining vertex. After that, the new spawn agents continue to propagate. This method can avoid some corner cases

and greatly increase the performance of the program as it is faster to get to the destination

.

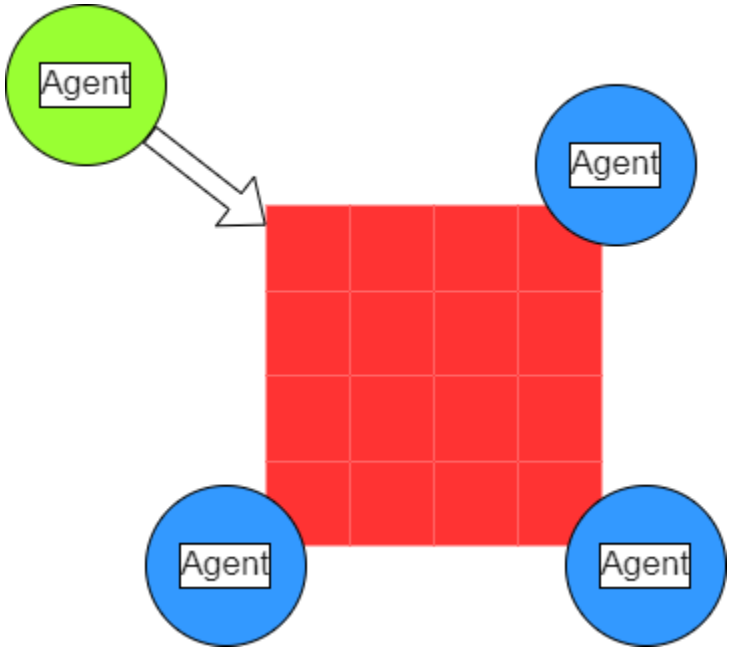

Figure 11: Agent propagates to another vertex on an obstacle.

Step 4:
Agent termination condition

As agent propagation progress is pretty heavy, minimizing the redundant agents is important. Here are termination conditions to decrease the number of agents.

1. Out of boundary
2. The place is located inside a obstacle
3. The place is visited before
4. The place has more than one agent, terminate the duplication

With the above conditions, the program asks the agents to terminate itself, and hence, reduce the pressure.

4. Conclusion

The Spark and MapReducer version of trapezoidal decomposition is just a part of Euclidean Shortest Path. They produce a trapezoidal map data for ESP. With this data, ESP can be solved using BFS, A* and other algorithms. There is an ESP parallelization project done by previous student Satin. Combining her work to this project can build a complete ESP in parallelization.

For the MASS implementation, since it does not need to do the trapezoidal decomposition process, it is likely to perform better than the Spark version and the MapReducer.

Reference

[1] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-77974-2.

[2] F. Li and R. Klette, *Euclidean Shortest Paths*. London: Springer London, 2011. doi: 10.1007/978-1-4471-2256-2.

[3] “Seidel’s Trapezoidal partitioning Algorithm.pdf.”

[4] Saranya G, “Agent Based Parallelization of Computational Geometry Algorithms”

[5] Patrick W, Andy Konwinski, Holden Karau and Matei Zaharia, *Learning Spark: Lightning-Fast Big Data Analysis*