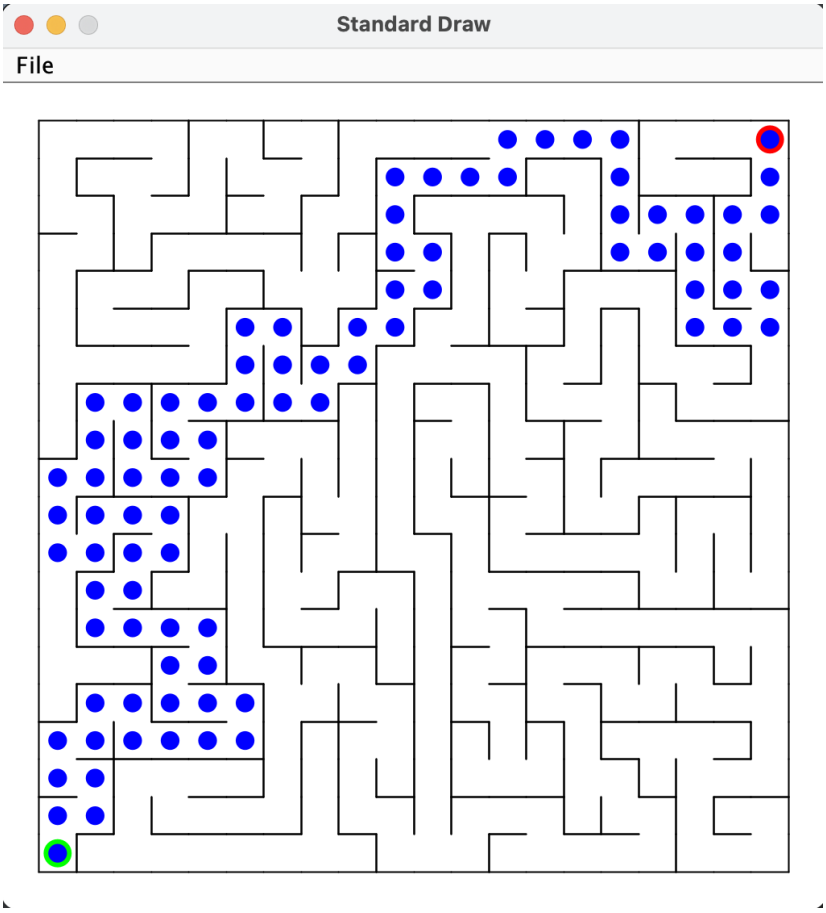
Parallelization of the A Star Search Algorithm



Ulysses Lin

Peter Loyd

Anh Tran

Introduction

A* search is a popular path-search algorithm, valued due to its completeness and optimality. Given a bounded or unbounded weighted graph and a admissible and consistent heuristic to find the target vertex, it will find the optimal path to the target. Its major drawback is space efficiency; it needs to hold the set of all areas it has visited or has seen. Its efficiency is also dependent on the choice of a good heuristic: the method for determining the cost to reach the target goal. A* path-searching is often the natural choice to make when looking for an intelligent search algorithm.

Due to its popularity and adaptability, A* search continues to provide fertile ground for researchers and practitioners. However, the memory issues and guided path of the algorithm tends to resist parallelization. A* uses $O(b^d)$ space for a singly threaded algorithm, where b is the branching factor and d is depth. The question of how to share that memory and how to access it is critical. Allowing each thread to independently hold memory can massively increase space complexity and require complicated message passing in order to identify when threads are doing redundant work. But maintaining a single silo of memory can result in race conditions, requiring the use of high latency synchronized operations.

Our goals in this research were first to implement a sequential single-threaded version of the algorithm, then convert that directly into a naïve multi-threaded solution, and finally look for other ways to improve on the algorithm.

Setup

To maintain an intuitive understanding of the choices made by our algorithms, we used a graph visualization tool made by Princeton University's Robert Sedgewick and Kevin Wayne. The class representing our maze was then based in part on their structure, in that it consists of a series of parallel 2d grids representing the walls surrounding a particular cell. However, to better follow the typical way graph search algorithms are written we then populated a grid full of nodes, with the maze effectively reporting edge locations to calling classes. We found the setup of the sequential version of A* relatively simple, however we found the maze creation and adaptation unexpectedly hard.

Java was chosen as our language to take advantage of its rich variety of concurrent structures. Since we implemented on a grid we decided to use Manhattan distance to the target as our heuristic. The computer used for metrics was running Windows 10 on an Intel i5-6600K CPU @ 3.50GHz, with 256KB L1 cache, 1.0MB L2, 6.0MB L3, and 16GB RAM. The i5-6600K has 4 cores.

Parallelization 1: Concurrent Priority Queue

Concept

In our first parallelization methodology we were aiming to use shared concurrent priority queues for our `frontier` nodes list and our `visited` nodes list. This would allow us to split up the processing of nodes in the frontier nodes priority queue by thread. The generation of nodes for the frontier node's priority queue is dynamic, so there will be times when threads have nothing to do but block and times when they will operate without constraint. If blocked, they will then restart when a node is added to the queue. The processing logic of each node needs to be synchronized to prevent any race conditions, and this is the main drawback of this method of parallelization: thread progress is a function of what percentage of calculations are synchronized. Another issue to consider is the "end-state", what is to happen once the target is found. Threads will process indefinitely unless a safe method of ending the progress is implemented.

Implementation

The idea is to have one `frontier` node list and one `visited` list implemented as a concurrent priority blocking queue. This is shared amongst all the threads. The main thread processes the start node and puts it into the frontier list and threads are created to process the frontier list from that point. Since we use the frontier.take() method it will block until a node is available. We then check to see if the current node is the target. If the node is not the target it is processed. First, we find its neighbors and calculate its new weight from the current node. If the neighbor is not in the visited or frontier priority queue, then we calculate the new f value. To prevent race conditions when adding this new g (new weight) and f value to the neighboring node we do this in a synchronized method `processGValue`. We set the parent of the neighbor as the current node and add the neighboring node to the frontier list. If the neighbor is in the frontier or visited list, we check to see if the new g value (new weight) is less than the neighbor's g value if so, this means that a better path is found so we calculate the new f value and using the same synchronous `processGValue` to process the neighbor node. If it has been visited before, remove it from the visited list and put it into the frontier list because now it has a different f value and needs to be processed again. Finally, we add the current node to the visited list.

A known complexity in parallelizing A* search is termination detection. A sequential search will terminate the algorithm naturally upon finding the target, and with an acceptable heuristic the path that results is guaranteed to be the optimal path. With parallel threads we cannot assume such a simple termination of the process. As a simple example, as threads process nodes it is possible for multiple threads to find their node is connected to the target. In many sequential A* implementations the first step upon finding the target is to set its internal data such that it can be used to trace the optimal path as a linked list. In parallel this results in race conditions as both threads attempt to claim themselves as the predecessor in the path. This situation was solved with a synchronized method that confirms optimality; however, this increases the latency dramatically.

A more difficult problem to account for is what to do when a path has been found to the target. There will be cases in which only a single approach to the target is possible, and there will be cases in which the optimal path might be discovered much later than the first path to the target. Our approach to this was to create a variable initialized to a maximum value. On finding the target this is set to the f score of the target and becomes a filter for all subsequent behavior: threads acquiring a node with a higher f score than the target will simply drop irrelevant nodes.

Results/Issues

We tested the speedup of our parallelization using concurrent priority queue by running the method without animation and printouts and timing the duration it takes to generate an optimal pathway. There are three different dimensions tested 20 by 20, 200 by 200, and 500 by 500 mazes. For each dimension we tested for 1, 2, 4, and 8 threads. Each test consists of running the parallelized A*Star search algorithm 500 times, except for those on the 500x maze, which was run 50 times. The result is averaged as time per maze solution.

Based on our result listed under Figure A.1 we do not see a speedup with the parallelization using a concurrent priority queue. It is worth noting though that as the maze dimension increases the speedup improves at 500 x 500 maze and 8 threads the speed up is 1.092. This indicates that at smaller dimensions the thread communication overhead is substantial. Unfortunately, the processing time for a 1000 x 1000 maze is far too long for us to realistically run the test and see if there is significant improvement, but the existing data suggests so. Given more time we would like to test out our parallelization with a bigger maze and have the capability to animate it.

Analysis

Ideally, without considering thread contention, our time complexity would be **O(b^ (d/p))** for our parallelization instead of **O(b ^ d)** because we were splitting the number of Nodes to examine in given timeframe by P number of threads. However, our implementation requires several synchronized methods as well as concurrent data structures, all of which contribute to latency. From our testing we can confirm that as soon as there is a possibility of thread contention, performance drops dramatically (Col 1 -> Col 2). With small maze sizes the sequential version dominates, which may reflect overhead of thread creation (Row 1 -> Row 2). With more than 2 threads performance increases as maze size increases. We theorize this could be due to an increase in I/O usage to access memory allowing threads more time to perform synchronized operations. Our largest maze size paired with the most threads (twice as many as we have processors) saw the most promising result.

We can characterize this as inconsistent to poor performance, with some possibility that larger maze sizes would show more rewards. Note that better results possible by increasing the cost to calculate the heuristic, which might be relevant in some applications but was considered beyond the scope of this project.

Maze Dimension	**1 Thread**	**2 Threads**	**4 Threads**	**8 Threads**
20 x 20	*0.332*	*0.301*	*0.136*	*0.080*
200 x 200	*1.355*	*0.903*	*0.777*	*0.658*
500 x 500	*0.873*	*0.609*	*0.849*	*1.092*

Figure A.1 Speedup result of parallel priority queue. Sequential queue / parallel. For dimensions 20 and 200 the comparison was run 500 times, for the 500x maze dimensions the thread was run 50 times.

Parallelization 2: Bi-directional A Star Search

Concept

Another methodology we considered and implemented was applying A Star search, but with multiple threads (in our case, two) originating from the two ends of the maze. Each thread would indeed have its respective start and target nodes at opposite corners of the maze, for example with thread 0 starting in the SW and targeting the NE, while thread 1 would start in the NE and target the SW. The ideal would be to have the threads concurrently utilize A Star, using the same heuristic, gradually moving to their respective targets, and meeting each other somewhere in the middle of the maze. In our research online and after running numerous tests, we found that the shortest path chosen by thread 0 would in fact be the same path as thread 1, but simply reversed in direction. This was a comforting notion initially, as it could then be assumed that the two threads would definitely meet in the middle. We discovered there was an occasional but important exception to this assumption, which will be discussed later when analyzing the results of bi-directional search.

Implementation

The general strategy was to take the sequential A Star algorithm and apply it to two threads instead of just one. This meant have two sets of `visited` and `frontier` `PriorityQueues`, and starting a `Runnable` for each thread (we used `HalfPath` as the `Runnable` implementation). Understanding under what conditions we should declare the threads as having met was a struggle. If we defined “meeting” as thread 0 checking if its current `Node` was the same as thread 1’s current `Node`, the two would theoretically report their paths as the *same* path; our A Star code reports back the target/final Node and iterates through the

parent `Nodes` of the ancestors – if both threads report the same `Node`, their ancestors are the same and thus the same path (undesirable). To avoid that, we compare each thread's current `Node's` *neighboring* `Nodes` and seeing if the other thread has seen the `Node` in any of its previous/current frontiers. `Node` thus has a seen field for this purpose. Any time a frontier meets another thread's frontier, we know the paths have met. After meeting, the current thread interrupts the other, reporting back the current `Node` and the neighbor `Node` for the other thread. We need to ensure both threads have finished to display results, so we do `join()` in the driver file.

Results/Issues

We can see the results of different sizes of mazes below. Note that when running the driver code, the display animation is *not* displaying the pathfinding in real time. The code solves the bi-directional problem, reports some results to the console, and then animates thread 0's then thread 1's paths.

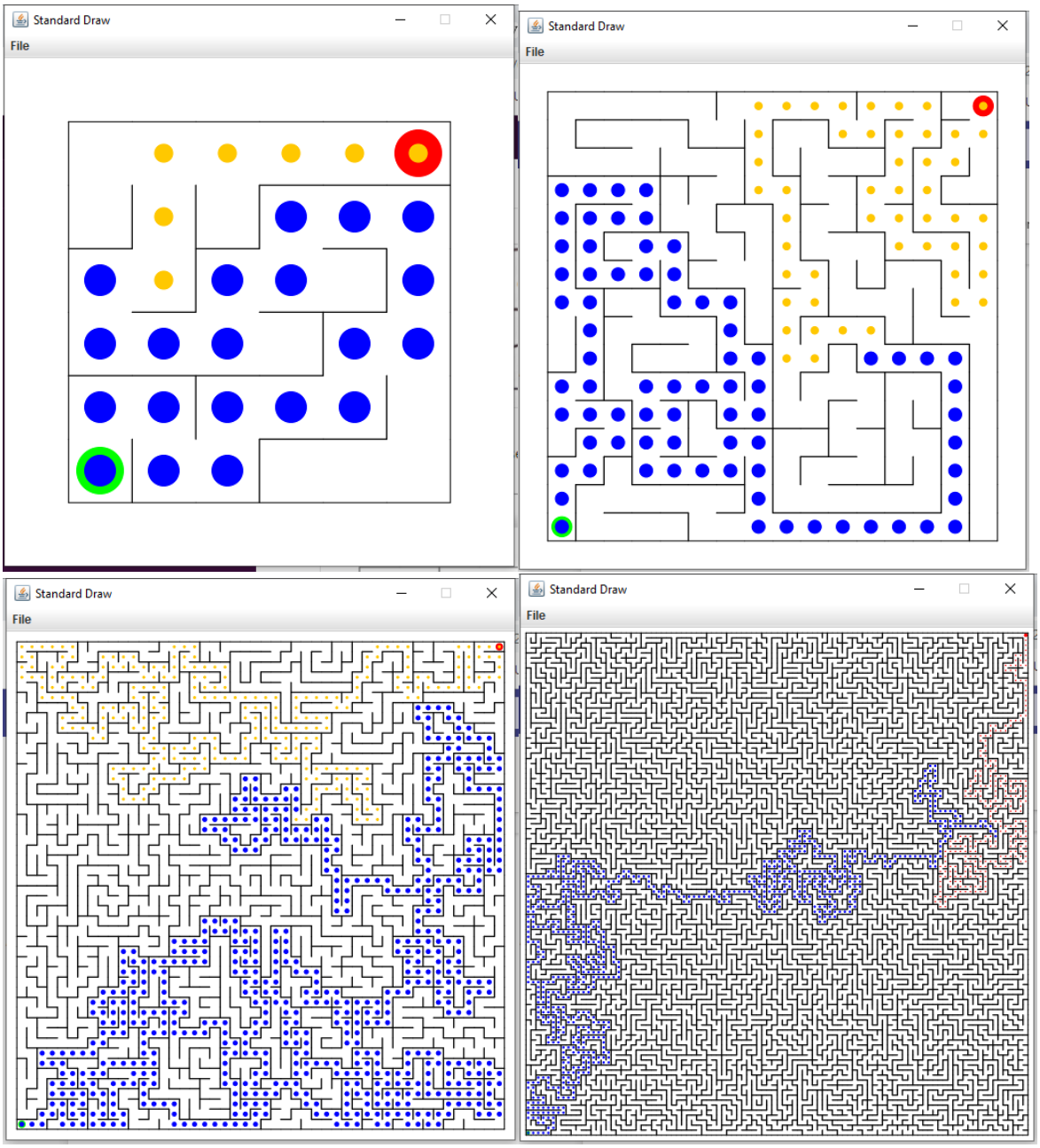


Figure B.1: 6x6, 16x16, 48x48, and 100x100 mazes. Thread 0 in blue and thread 1 in yellow/red.

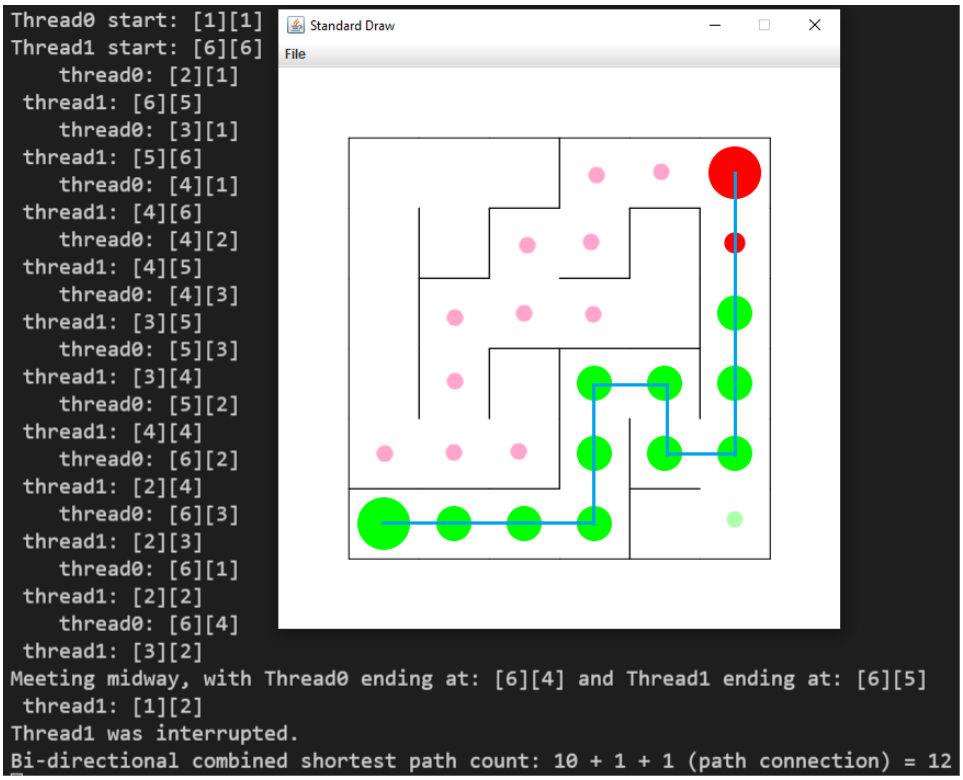


Figure B.2: Both threads do in fact share the work of exploration among about the same number of Nodes, but the red (thread 1) struggles more and only contributes 1 step to the final shortest path (blue line). Pink dots show Nodes explored by the red that did not ultimately go anywhere, compared with the single light green dot for the other thread.

Note that in these examples and typically, the blue (thread 0) path is longer than the yellow (thread 1) path. This may have been due to the maze often creating less forks in the road or less dead ends in the ways that thread 0 goes. This notion is supported by the fact that in Maze.java maze creation originates at the SW corner (start) and forks are created later. Looking into console logs stating the grid coordinates that both threads explore, there does not seem to be bugs in pathfinding. Thread 1 usually appears to have a shorter final path, and this is due to thread 1 exploring paths that are ultimately dead ends and needing to double back (see Figure B.2).

Something we found that was an occasional exception was the threads failing to meet in the middle. As seen below.

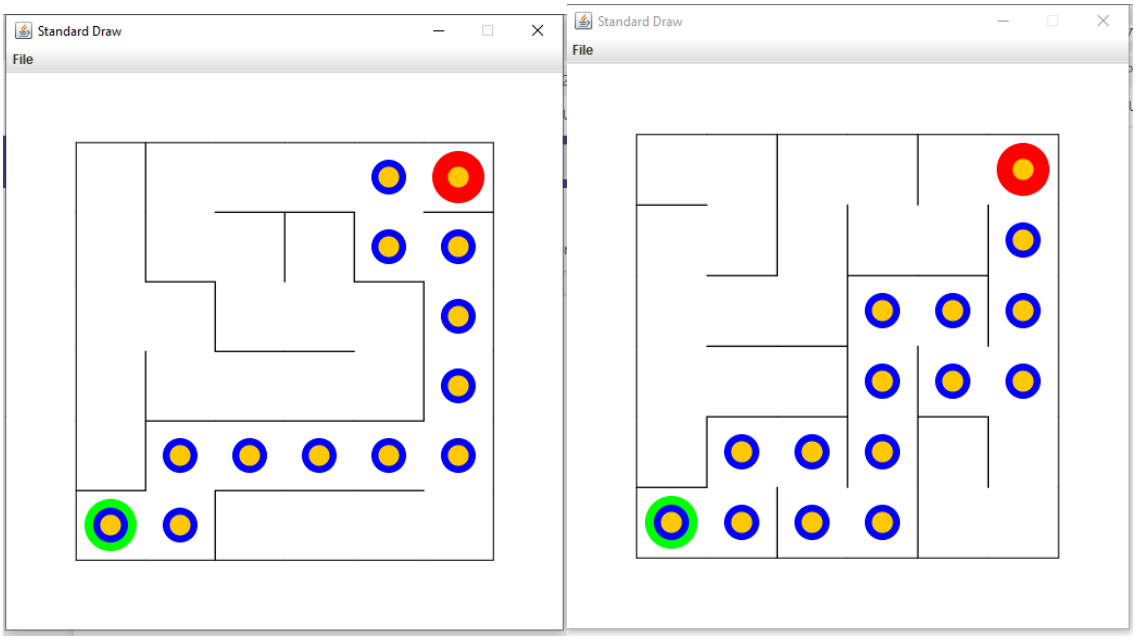


Figure B.3: Threads failing to meet in the middle.

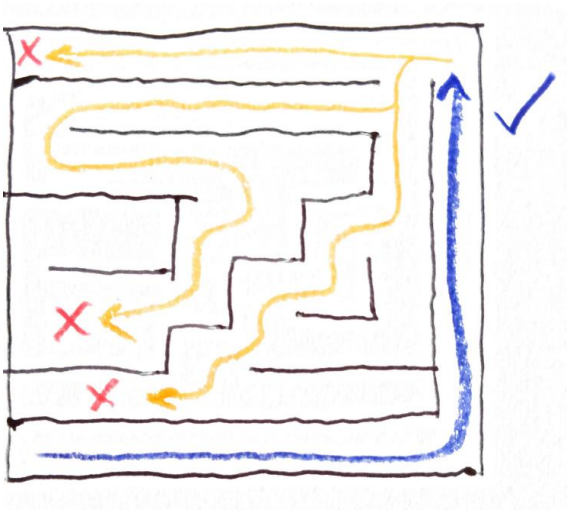


Figure B.4: A theoretical depiction of the yellow path encountering reasonable, though ultimately frustrating and convoluted paths, while the blue path is straightforward and finishes the entire maze first.

Though this looks unusual and undesirable, we believe **this is reflective of the nature of A Star and bi-directional pathfinding**. A Star uses a heuristic to find a reasonably probably path toward a target, but sometimes this means going down a path that ultimately leads to a disappointing dead-end. Because say thread 0 is concurrently searching for a path, it may end up traveling across the maze and finding a full shortest path to the other corner before thread 1 has time to "correct" its path by reversing course and trying another branch of the maze. A Star bi-directional search is not guaranteed to meet in the middle as both threads are independently searching for the shortest path and one thread may take a significantly more convoluted path

than the other and thus give the luckier thread enough time to just complete the maze in full (instead of just partially). In Figure B.3 we can see how the yellow path (thread 1) would take a heuristically reasonable path toward the left, eventually hitting dead ends, while the blue (thread 0) is not presented with forks in the road at the start and has a much more defined path forward and finishes first. Figure B.4 demonstrates how forks in the road for one thread and an easy path for the other thread can lead to the situation described.

Analysis

In latency tests, bi-directional appears to be more than twice as fast as sequential search. This seems an exaggeration of the expected speed-up using two threads coming from two ends. According to online sources, we can expect speeds of **O(b ^ (n/2))** for bidirectional instead of **O(b ^ n)**, since we are theoretically splitting the number of Nodes to examine in a given timeframe by half [1]. The math does not appear to be in line with our timing results. That being said, logically the availability of two threads *exploring independently* promises significant potential speedup. Consider the examples above where thread 0 goes its full path length while thread 1 meanders – had we only sequentially explored using thread 1's path, we would have hit dead ends and had a convoluted, time-consuming journey to end up with the same final shortest path as thread 0, which faced less barriers and thus explored much faster. Having two threads explore independently accommodates one of them taking an "unlucky" journey, and we just take the path of the thread that finished first. In the typical scenario though, we would see the Nodes explored theoretically split evenly between the two threads, while the steps of the combined path contributed by a given thread would really depend on the luckiness of the pathfinding. Bi-directional search offers less overhead as two threads are spun up then after the maze is solved, terminated; there is no frequent creation or termination of threads.

Bi-directional Speedup:

Maze Dimension	**2 Threads**
20 x 20	*0.315*
200 x 200	*4.594*
500 x 500	*2.865*

Figure A.2 Speedup result of Bi-directional A*

Conclusion

Based on our implementation and latency testing the parallelization of A Star Search using a concurrent priority queue was not very successful as it did not produce any speed up. Bi-

directional parallelization on the other hand produced very successful speed up. We are surprised by this success as it goes beyond our initial assumption. If time permits, we would like to explore different methodologies that can improve the parallelization of the priority queues. One such example in literature is the use of a lock free priority queue. This is achieved via implementation of a skip list. Or perhaps another good example is the use of one priority queue per thread followed by some form of message passing between threads.

Overall, we were able to dive deeply into the A Star Search method and develop two parallelization methods for it in which one turned out to be successful. This was a growing experience and if given more time we would have explored more into other methodologies and fix our strategy for the shared concurrent priority queue.

Reference

Belwariar, Rachit 2022, accessed 4 March 2022, <https://www.geeksforgeeks.org/a-search-algorithm/>.

[1]Sawlani, Sneha, "Explaining the Performance of Bidirectional Dijkstra and A* on Road Networks" (2017). Electronic Theses and Dissertations. 1303.

Sedgewick, Robert and Wayne, Kevin 2017, accessed 4 March 2022, <https://algs4.cs.princeton.edu/41graph/Maze.java.html>.

Sedgewick, Robert and Wayne, Kevin 2022, accessed 4 March 2022, <https://introcs.cs.princeton.edu/java/stdlib/StdDraw.java.html>.

Weinstock, Ariana, and Rachel Holladay. "Parallel A* Graph Search." (2017).