COMS 4995 Parallel Functional Programming, Fall 2024
Final project proposal

Project name: maze-solver
Project members: Mohsin Rizvi (UNI: mkr2151)

For my project, I would like to write a maze solving program that works using a parallelized A* (or A-star) algorithm. A* is a graph algorithm that takes in a weighted graph as input, as well as a source node and a target node, and finds the shortest path from the source node to the target node. The algorithm uses a heuristic function, which provides an estimate of how “far apart” a given node is from the target node. A* is commonly used in video games to compute paths between in-game locations.

A* algorithm

The algorithm works using two structures, each containing nodes. Each node n stores the following:

- X and Y coordinates
- Number of steps from the start node
- h(n), where h is the heuristic function and n is the node
- A reference to the previous node in the path from the start node

The first structure is the **open list**, which is a priority queue of nodes to process, sorted in ascending order by the sum of their heuristic value and distance from the start node. The **closed list** contains nodes that have already been processed.

The algorithm first adds the start node to the open list, then continuously extracts nodes from the open list. For each extracted node n, it first checks if n is a goal node. If so, n’s previous node information is used to rebuild the path from the start node to the goal node. If n is not a goal node, n is moved from the open list to the closed list. All neighbors of n not in the closed list or the open list then have their fields populated and are added to the open list. This repeats until a goal node is found or the open list is empty, in which case no path exists.

Using A* for maze solving

For my project, I would write a parallel implementation of A* that uses the algorithm to solve mazes. As input, my program would take the path to a “maze file” which contains an ASCII representation of a maze. A maze is a 2-dimensional grid of tiles, where each tile can be one of the following:

- ”s“ denotes a start tile that may be part of a maze solution
- “g” denotes a goal tile that may be part of a maze solution
- “.“ denotes an empty tile that may be part of a maze solution
- “#” denotes a wall tile that may **not** be part of a maze solution

A maze must contain exactly one start tile and one goal tile. Paths cannot go diagonally between tiles. For example, the following 4x4 grid is a valid maze:

```
s..#
.#.#
#...
g.#.
```

The program would read in the maze file, construct a graph between non-wall tiles, and then run the parallelized A* algorithm on the graph. The output from the program would be a printed path from the start tile to the goal tile, or a message indicating that no solution path exists. For my A* heuristic function, I will use the distance between a given tile and the goal tile, computed using the X and Y positions of tiles in the maze.

Parallelizing the algorithm

To parallelize the algorithm, I will have the program take in as input a maximum number of CPU cores to use. Then, using the par monad, have each CPU process the next node from the priority queue, and return either a solution or a list of nodes to add to the open list.

I believe this approach will lead to a faster solution, as processing nodes will be done in parallel. Nevertheless, there is the risk of situations where something like the following happens, leading to unnecessary work:

- Core 1 takes the first node (n1) from the open list.
- Core 2 takes the new first node (n2) from the open list, which was previously the second node.
- Core 1 finishes its work on n1, and returns a new node (n3) that is then added to the open list. n3 has a higher priority than n2, but n2 was already removed from the open list.

In this case, doing a serial A* would have processed n3 before n2, since n3 has higher priority. Nevertheless, n3 would still be eventually processed in this case; we would just spend a bit of extra time processing n2 (possibly unnecessarily). There is also a risk that n2 terminates first, adds its neighbors to the open list, and then those are searched before n3, which may result in a less-than-optimal resulting path if n3's path is shorter than the one resulting from n2.

Test data

To come up with test data, I will use the following:

- Websites that generate tile-based mazes, such as https://znuznu.github.io/daedal/. Note that I will need to add a start and goal node to the mazes, as well as translate the generated mazes into a maze file. Websites that generate non-tile based mazes (i.e., mazes where walls are simply sides of a tile as opposed to tiles themselves) cannot be used.
- Mazes hand-drawn by myself.

Additional features

Additional features that I may implement if I have time is support for multiple goal tiles, where a path from the start tile to any goal tile is a correct solution, support for paths containing diagonal steps, and support for mazes with no valid solution (in which case a message indicating such will be printed).