John Nickle

Maze Heuristics

This project entailed using a variety of heuristic searches in order to solve any randomized maze. Five heuristic searches were implemented, all of which solved the maze with varying degree of success.

One heuristic simply uses Manhattan distance. The Manhattan distance is calculated using only horizontal or vertical directions on a grid; diagonals are not considered. Since the mazeExplorer only moves in horizontal or vertical directions to solve the maze, the difference of the estimated distances from any parent node to its successor can never exceed 1. This makes Manhattan distance monotonic. The Manhattan distance heuristic was quite efficient at solving mazes. Here we have the data of the hardest mazes that it attempted to solve in under two minutes using my computer with a clock speed of 4.0GHz:

Trial #	Perfection	# Treasures	Maze Size	# Nodes	Depth	b*
Trial 1	0%	15	50x50	NA	NA	NA
Trial 2	50%	15	50x50	703433	53	1.3
Trial 3	100%	15	75x75	857844…	5220	1.0
Trial 4	0%	10	100x100	541731…	410	1.03
Trial 5	100%	10	150x100	1084559…	9066	1.0

Interestingly, when creating the “hardest” maze, the perfection and amount of treasure change the difficulty the most. The hardest combination involves a high number of treasures and a very imperfect maze. This is most likely due to the fact that solving an imperfect maze produces more successor nodes, increasing the time to solve the maze while also having to find treasure scattered around the map.

Next we have the ManhattanTreasure heuristic. This heuristic also happens to be monotonic. It uses the Manhattan distance of the MazeExplorer to the goal and subtracts the

distance of the furthest treasure from the exit. This effectively gives a Manhattan distance minus a constant. Since we already know that the Manhattan distance is monotonic, simply subtracting a constant will allow it to remain monotonic. Here is the data gathered from the hardest maze experiment:

Trial #	Perfection	# Treasures	Maze Size	# Nodes	Depth	b*
Trial 1	0%	10	50x50	NA	NA	NA
Trial 2	50%	15	50x50	NA	NA	NA
Trial 3	100%	15	75x75	888137…	3605	1.0
Trial 4	0%	10	100x100	NA	NA	NA
Trial 5	100%	10	150x100	882503…	9582	1.0

With ManhattanTreasure, there seems to be a large difference in the ability to complete the maze. Over half of the trials had to be halted due to never completing.

Next we will examine MazeSweeper. MazeSweeper gets the Manhattan distance of the closest treasure to the MazeExplorer and adds the Manhattan distance to the goal. Since each calculated distance is monotonic, adding them together is also monotonic. Here is the data from the hardest maze experiment:

Trial #	Perfection	# Treasures	Maze Size	# Nodes	Depth	b*
Trial 1	0%	10	50x50	957618…	213	1.06
Trial 2	50%	15	50x50	NA	NA	NA
Trial 3	100%	15	75x75	NA	NA	NA
Trial 4	0%	10	100x100	NA	NA	NA
Trial 5	100%	10	150x100	1107381…	8497	1.0

MazeSweeper did very poor as well. This could be due to the gratuitous amount of computation the heuristic must do in order to calculate the closest treasure each time.

Next we will look at two non-monotonic heuristics used to solve the maze. The first we will examine is GreedyTreasure. This heuristic has the MazeExplorer seek out the closest

treasure until they are all collected, and then uses the Manhattan distance to solve the rest of the maze. This heuristic is definitely non-monotonic, as each time a treasure is collected, the next returned distance could be any number. This breaks the monotonicity, as, by definition, the difference between h_1 and h_2 must be < or = 1. Here is the data for GreedyTreasure:

Trial #	Perfection	# Treasures	Maze Size	# Nodes	Depth	b*
Trial 1	0%	10	50x50	590325…	195	1.06
Trial 2	50%	15	50x50	NA	NA	NA
Trial 3	100%	15	75x75	186707…	1933	1.0
Trial 4	0%	10	100x100	619177…	357	1.0
Trial 5	100%	10	150x100	252276…	7489	1.0

GreedyTreasure did ok compared to the other monotonic heuristics, only failing one of the trials.

Finally, the last non-monotonic heuristic is TreasureDistance. What this heuristic does is it takes into account the furthest non-collected treasure away from the goal and uses that as its estimate. This heuristic is non-monotonic because as treasure is collected, the furthest treasure away could be much closer or even 0. This breaks the rule of monotonicity.

Here is the data for TreasureDistance:

Trial #	Perfection	# Treasures	Maze Size	# Nodes	Depth	b*
Trial 1	0%	10	50x50	370495	211	1.06
Trial 2	50%	15	50x50	NA	NA	NA
Trial 3	100%	15	75x75	912455…	4763	1.0
Trial 4	0%	10	100x100	777741	391	1.03
Trial 5	100%	10	150x100	520593…	10255	1.0

Again, the non-monotonic heuristic appears to be better able to solve any random maze with varying difficulty.

The results of these experiments were not what I expected. I fully expected the monotonic heuristics to outperform the non-monotonic ones in every way. However, the non-monotonic heuristics outperformed the monotonic ones in every way, with GreedyTreasure even being more

efficient overall. Interestingly, the simple Manhattan heuristic appeared to be the most efficient in the trials that it completed. It seems simple, less informed calculations outperform the more complex, but better informed heuristics. This makes sense when thinking about the amount of calculations needed to perform one search adds up compared to one calculation.

Proofs and Code

Manhattan:

```
//Monotonic
public class Manhattan implements BestFirstHeuristic<MazeExplorer>{
    public int getDistance(MazeExplorer node, MazeExplorer goal) {
            int manhattanDistance = goal.getLocation().getManhattanDist(node.getLocation());
            return manhattanDistance;
            }
}
```

ManhattanTreasure:

```
//Monotonic
public class ManhattanTreasure implements BestFirstHeuristic<MazeExplorer> {
   public int getDistance(MazeExplorer node, MazeExplorer goal) {
            int manhattanDistance = node.getLocation().getManhattanDist(goal.getLocation());
            int furthest = 0;

            node.getAllTreasure().removeAll(node.getTreasureFound());

            for(int i = 0; i < node.getAllTreasure().size()-1; i ++){
                    furthest = Math.max(node.getAllTreasure().get(i).getManhattanDist(goal.getLocation()),
node.getAllTreasure().get(i+1).getManhattanDist(goal.getLocation()));
            }
            return manhattanDistance + furthest;
   }
}
```

MazeSweeper:

```
//Monotonic
public class MazeSweeper implements BestFirstHeuristic<MazeExplorer> {
            public int getDistance(MazeExplorer node, MazeExplorer goal) {
            int closest = 0;

            node.getAllTreasure().removeAll(node.getTreasureFound());

            for(int i = 0; i < node.getAllTreasure().size()-1; i ++){
                    closest = Math.min(node.getAllTreasure().get(i).getManhattanDist(node.getLocation()),
node.getAllTreasure().get(i+1).getManhattanDist(node.getLocation()));
            }

            int manhattan = node.getLocation().getManhattanDist(goal.getLocation());


            return closest + manhattan;
   }

}
}
```

GreedyTreasure:

```
//Non-Monotonic
public class GreedyTreasure implements BestFirstHeuristic<MazeExplorer>  {
        public int getDistance(MazeExplorer node, MazeExplorer goal) {
                int closest = 0;

                node.getAllTreasure().removeAll(node.getTreasureFound());

                if(node.getAllTreasure().size() > 0){
                        for(int i = 0; i < node.getAllTreasure().size()-1; i ++){
                                closest =
Math.min(node.getAllTreasure().get(i).getManhattanDist(node.getLocation()),
node.getAllTreasure().get(i+1).getManhattanDist(node.getLocation()));
                        }
                }
                return closest;
        }
}
```

TreasureDistance:

```
//Non-Monotonic
public class TreasureDistance implements BestFirstHeuristic<MazeExplorer> {

   public int getDistance(MazeExplorer node, MazeExplorer goal) {
        int furthest = 0;

        node.getAllTreasure().removeAll(node.getTreasureFound());

        for(int i = 0; i < node.getAllTreasure().size()-1; i ++){
                furthest = Math.max(node.getAllTreasure().get(i).getManhattanDist(goal.getLocation()),
node.getAllTreasure().get(i+1).getManhattanDist(node.getLocation()));
        }
        return (furthest);
   }
}
```

Proofs:

Between Manhattan and ManhattanTreasure, they are equal when simply solving the maze, as they both would just use Manhattan distance. However, ManhattanTreasure becomes better informed as treasure is added. This is due to the fact that a simple Manhattan distance to the goal ignores the fact that there are treasures to collect, and adding the furthest uncollected treasure to the distance gives a better representation of how much further the node actually needs to go.

Between Manhattan and MazeSweeper, they are also equal when simply solving the maze. Again, MazeSweeper is more informed in that it takes into account that there are treasures to be collected when giving an estimated distance to the goal.

Between ManhattanTreasure and MazeSweeper, ManhattanTreasure is better informed. Getting the furthest distance necessary to travel is much more informed than using the next closest treasure. MazeSweeper is simply too optimistic in its estimate, while ManhattanTreasure gives a much closer estimate to the actual distance left to travel.