Voyage into the Unknown

Yu-Ju Ku (YK544) Jason Luo (JL2362) Vincent Taylor (VT152)

September 2021

1 Properties of Repeated Search

Question 1. Why does re-planning only occur when blocks are discovered on the current path?

Assuming optimality of the search algorithm used (which holds for A^*), at any moment, the remaining path left to be travelled by the agent represents a shortest path from the agent's current position to the goal (based on the agent's knowledge of the gridworld). When the agent's knowledge is updated to introduce more blocks, this new information can only increase the length of the presumed shortest path, because more obstacles will make pathfinding more difficult (formally, the set of valid paths in the updated gridworld is a subset of the set of valid paths in the original gridworld).

Thus, the agent can do no better than its current path, *provided the blocks introduced do not obstruct this path* (which would invalidate the path and force a re-plan). Re-planning with the new information will return either the same path or some other path of equal length, making it a waste of effort.

Question 2. Will the agent ever get stuck in a solvable maze?

Assuming the search algorithm used to plan is complete (which holds for A^*), the agent will not get stuck in a solvable maze. This is to say that in a solvable maze, (1) the agent will always find a possible path during each (re-)planning phase, and (2) the agent will not get stuck in an infinite cycle of re-planning.

During each planning/re-planning phase, there exists a path from the agent's current location to the start by virtue of the agent being able to move between the two. Because the maze is solvable, there also exists a path from the start cell to the end. Combining these two, there is always a path from the agent's current location to the end, so completeness ensures (1).

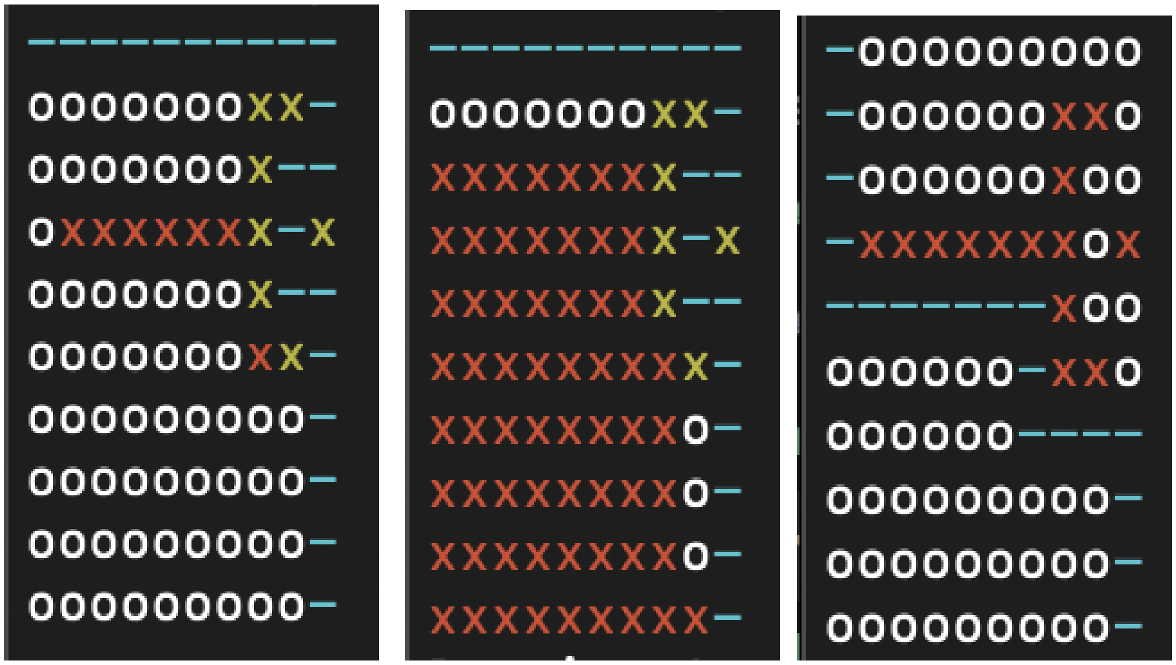
Also, each time the agent re-plans, it adds at least one obstacle to its knowledge of the environment (namely, the obstacle on the current path that forced the re-plan). Suppose there are n obstacles in the maze (n must be finite). Then in the worst case, the agent will re-plan n times, after which the agent will have seen all the obstacles and hence have full knowledge of the environment. At this point, the agent will find a path which is guaranteed to be valid in the complete maze, meaning it will not have to re-plan again. Thus the agent will re-plan at most finitely many times, proving (2).

Question 3. Once the agent reaches the target, consider re-solving the now discovered gridworld for the shortest path (eliminating any backtracking that may have occurred). Will this be an optimal path in the complete gridworld?

In general, no. Consider the following example in Figure 1. Barriers are marked with 'x', free spaces are marked with 'o', and the path the agent takes is marked with '–'.

Repeated search is run using A^*, equipped with the Manhattan distance as a heuristic. The agent first attempts travelling to the right, and as a result gets stuck in a winding hallway, as shown in Figure 1a; the resulting path length is 20.

Figure 1b depicts the shortest route through the gridworld that the agent has discovered. In addition to the barriers that the agent observed (marked in yellow), any cells that never entered the agent's field of view during the first pass are also considered blocked (so that the agent never routes through cells that it doesn't know for certain is free). Since the agent didn't actually do any backtracking on the first pass, the shortest path on the discovered gridworld is the same as the path found on the first pass, with a length of 20.



```
----------
OOOOOOOXX-
OOOOOOOX--
OXXXXXXX-X
OOOOOOOX--
OOOOOOOXX-
OOOOOOOOO-
OOOOOOOOO-
OOOOOOOOO-
OOOOOOOOO-
```

(a) with no initial knowledge

```
----------
OOOOOOOXX-
XXXXXXXX--
XXXXXXXX-X
XXXXXXXX--
XXXXXXXXX-
XXXXXXXXO-
XXXXXXXXO-
XXXXXXXXO-
XXXXXXXXX-
```

(b) on discovered gridworld

```
-OOOOOOOOO
-OOOOOOXXO
-OOOOOOXOO
-XXXXXXXOX
-------XOO
OOOOOO-XXO
OOOOOO----
OOOOOOOOO-
OOOOOOOOO-
OOOOOOOOO-
```

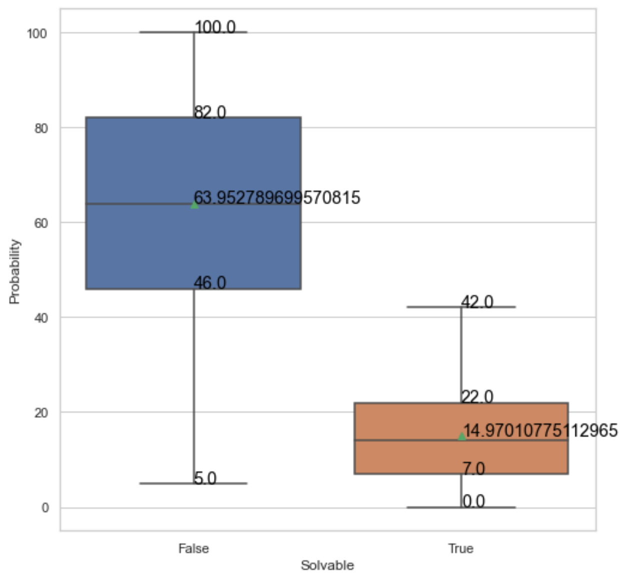
(c) on complete gridworld

Figure 1: Searching with zero, partial, and complete information

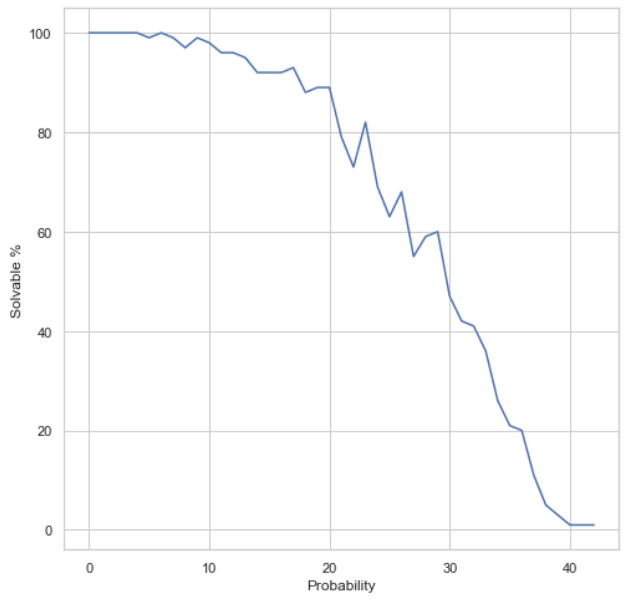
However, because the agent doesn't attempt to explore areas it hasn't seen before on the second pass, it misses the actual shortest path, which has length 18 and lies below the starting cell in an undiscovered region of the gridworld. One of the many optimal paths is shown in Figure 1c.

2 Analysis of Regular A^*

Question 4. Examining gridworld density vs. solvability.



(a) Box plot



(b) Line plot

Figure 2: Plotting density vs. solvability

For each probability percentage p between 0 and 100, 100 random gridworlds were generated and tested for solvability. Figure 2 shows a box plot indicating the distribution of solvable versus unsolvable grids from this data, as well as a line plot displaying the percentage of solvable grids (dependent on p). From Figure 2b we can see that the threshold probability lies at around $p_0 = 30\%$.

Question 5. Quality of different heuristics.

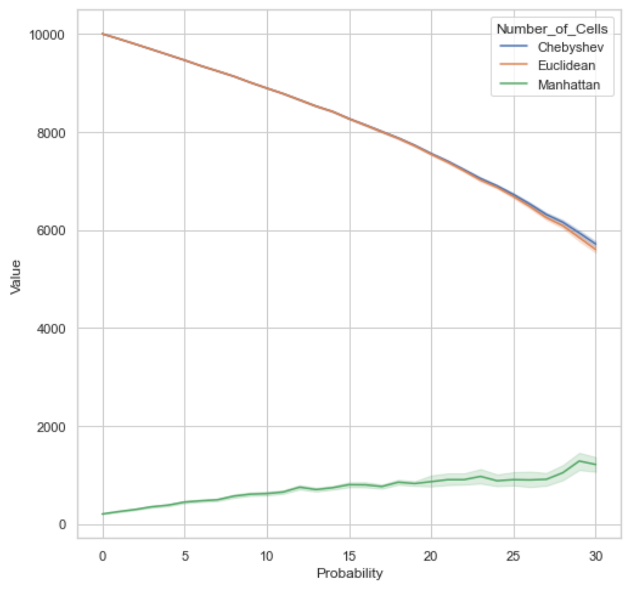
We tested each heuristic (Euclidean, Manhattan, and Chebyshev distance) on the same sample of solvable gridworlds, and gathered data on the runtime, and the number of nodes processed/expanded for each heuristic.

Mathematically, since all the heuristics considered are consistent, it is known that A^* will expand all the nodes with cost $f(x) < C^*$ (where C^* is the length of the optimal path). Thus, the less a heuristic underestimates the true cost to a goal, then the less likely nodes will have an f-cost lower than C^*, and in turn the less total nodes A^* will have to expand. Via simple manipulations, we can also show that for any $a, b > 0$, we have

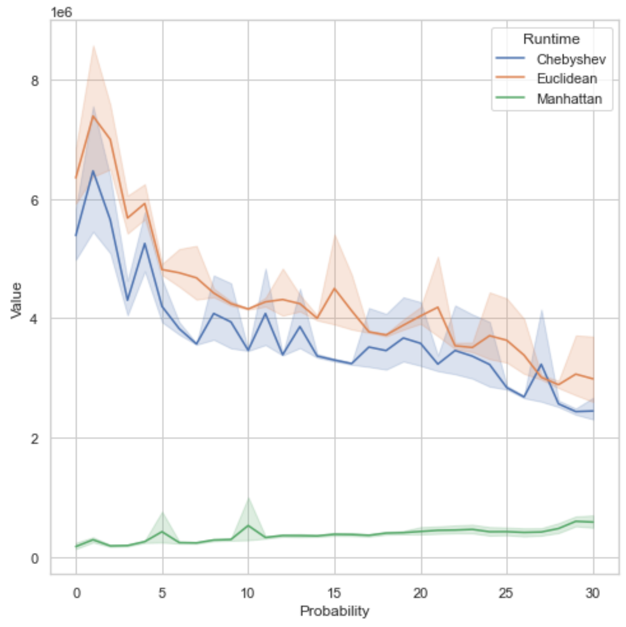
$$\underbrace{\max(a,b)}_{\text{Chebyshev distance}} < \underbrace{\sqrt{a^2+b^2}}_{\text{Euclidean distance}} < \underbrace{a+b}_{\text{Manhattan distance}}$$

We see that the Manhattan metric is always a better estimate than the Euclidean metric, which is in turn always a better estimate than the Chebyshev. Thus the Manhattan metric should perform uniformly better than the Euclidean, and likewise for the Euclidean metric vs. the Chebyshev.

Our expectations are confirmed in Figure 3; the Manhattan metric significantly outperforms both the other metrics in both runtime and number of cells processed. Note that although the Euclidean metric always expands slightly less nodes than the Chebyshev metric, the Chebyshev metric generally takes less time to run. This implies that running A^* with the Euclidean metric causes the f-cost of nodes in the fringe to update more frequently, which incurs an overhead because the node's position in the fringe then must be



(a) Number of Cells processed



(b) Runtime

Figure 3: Measuring efficacy of different heuristics

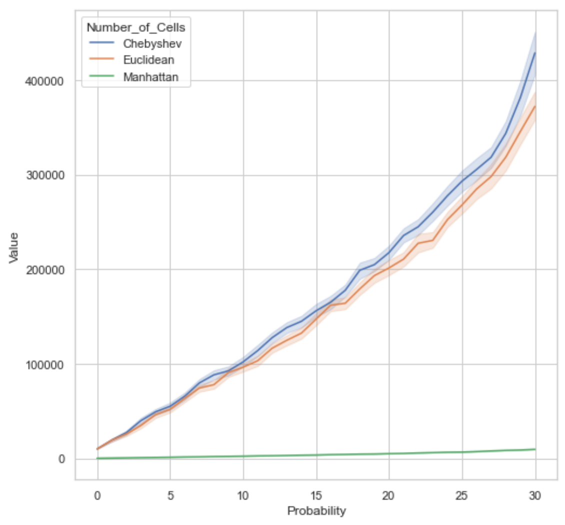
updated. (In our implementation, updates to the fringe are performed simply by removing and re-inserting the relevant node, which is an $O(n)$ operation.)

It is also interesting to observe that while the Manhattan metric has a upward trend in both graphs as density increases, both the Chebyshev and Euclidean metrics start with a high runtime/cell count, which actually *decreases* as density increases. We believe this is because the Chebyshev and Euclidean metrics underestimate the cost so much that the A^* algorithm has BFS-like behaviour (with uniform edge-costs, BFS can be considered as an extreme form of A^* where the heuristic is $h(x) = 0$, i.e. the cost is underestimated as much as possible). As a result, when using these heuristics we have a large fringe and expand a large number of nodes. Then as the density increases, there are less and less unblocked nodes, so the size of the fringe is naturally limited. On the other hand, the Manhattan metric is virtually a perfect heuristic at lower densities, i.e. it gives almost the exact cost from a given node to the goal; in this way, BFS-like behaviour is avoided with the Manhattan metric and A^* is able to close in on the goal relatively quickly.

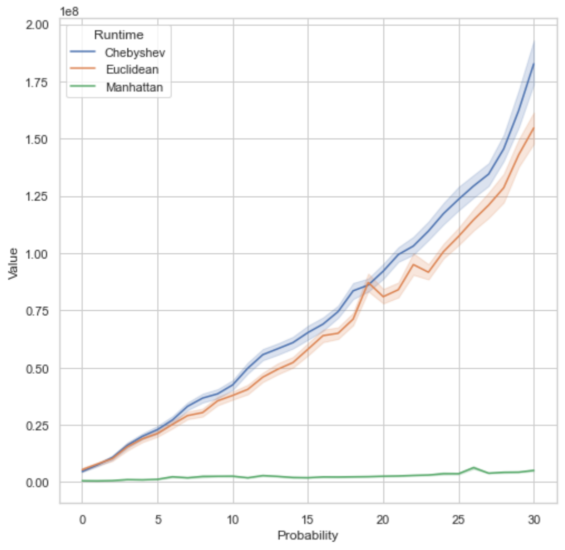
Aside. For completeness, we also ran Repeated A^* (i.e., running the search with zero initial knowledge instead of complete knowledge) with the different heuristics and gathered the same data. This data is shown in Figure 4.

We see that in general, the Manhattan metric still outperforms the Euclidean and Chebyshev metrics in both runtime and number of cells processed. However, Figure 4c indicates that the Manhattan metric yields a longer path length than the other metrics when used in Repeated A^*. In other words, the Manhattan metric is more likely to lead the agent into dead ends, from which it has to backtrack, adding to the distance travelled. We believe this is because the Manhattan metric encourages the agent to explore along the boundaries of the gridworld, while both the Euclidean and Chebyshev metric encourage the agent to explore along the diagonal connecting the start and goal cells, and this difference is what causes the amount of backtracking needed to vary.

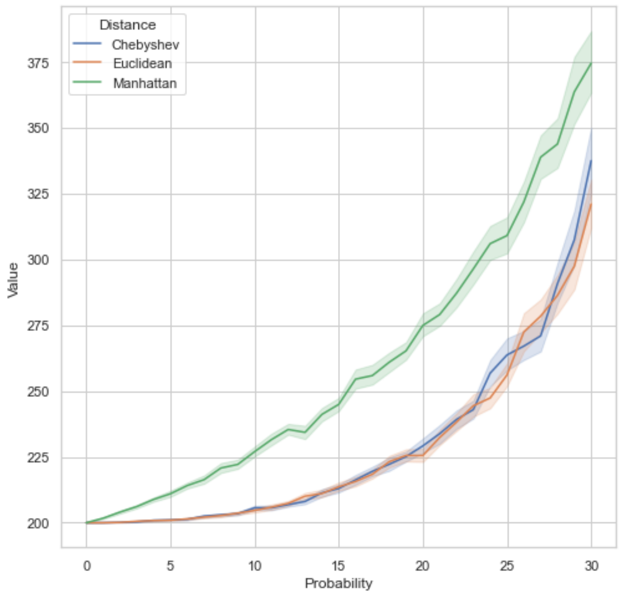
Moving forward, we use the Manhattan metric to conduct our tests, favoring its superior performance.



(a) Number of Cells processed



(b) Runtime



(c) Path Length

Figure 4: Measuring efficacy of different heuristics with Repeated A^*

3 Analysis of Repeated A^*

Question 6. Using an agent that can see sideways.

We ran 100 iterations of Repeated A^* at each density value from 0 to 33, and generated the data in Figure 5.

All of the graphs follow upward trends as density increases. The reason for each upward trend is different however, as explained below.

- **Trajectory Length (Figure 5a):**

  As density increases, two things occur. First, the optimal path in the complete grid lengthens (on average), since the path-finding problem becomes more difficult. Second, the agent is more likely to run into obstacles, forcing a re-plan and potential backtracking. Both of these factors contribute to a longer trajectory length as density increases.

- **Ratio of Path Length with Repeated A^* and Path Length in Discovered Gridworld (Figure 5b):**

  As mentioned earlier, a higher density corresponds to more potential backtracking. This backtracking is eliminated when solving the discovered gridworld, so the ratio is expected to increase as density increases (i.e., the path on the discovered gridworld becomes shorter w.r.t. the path returned by Repeated A^*).

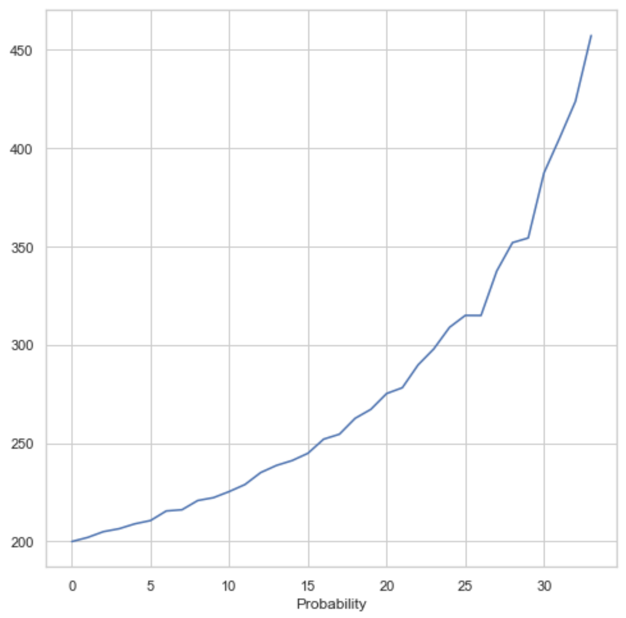
- **Ratio of Path Length in Discovered Gridworld and Path Length in Complete Gridworld (Figure 5c):**

  This upward trend went against our expectations. As density increases, we surmised a higher amount of re-planning and backtracking would mean the agent gets to discover more of the environment, which in turn implies the discovered gridworld becomes closer and closer to the complete gridworld. Thus, we expected there to be some turning point where the ratio would start decreasing again and approach 1. However, this did not occur. It seems the increasing trajectory length (as discussed in the first item) outweighs any benefits the increased re-planning might have offered.

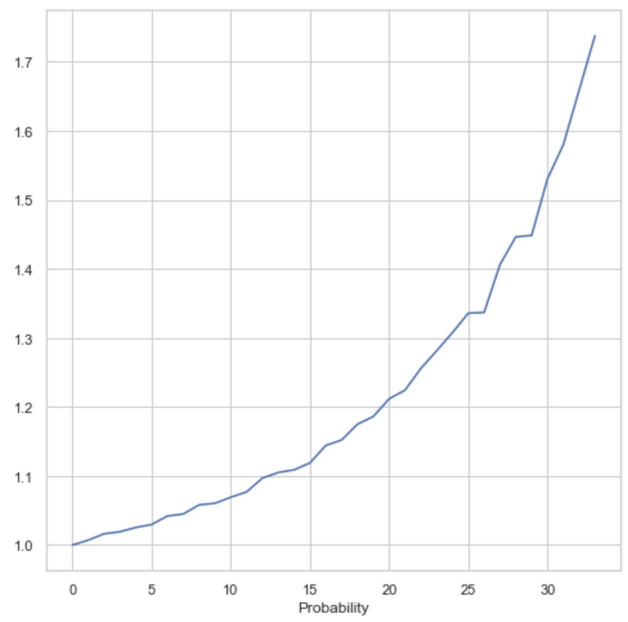
- **Number of Cells Processed (Figure 5d:)** As mentioned earlier, a higher density means more re-planning. Each re-planning step requires cells that might have been expanded in previous planning steps to be expanded yet again, contributing substantially to the total count of cells processed. Hence the upward trend is expected.

Question 7. Using an agent that cannot see (i.e., that can only observe by bumping into obstacles).

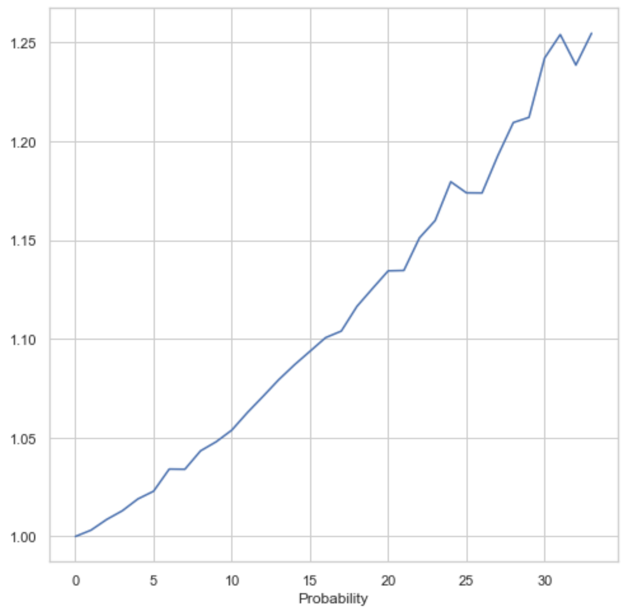
Figure 6 contains the same data as Figure 5, except using an agent with no f.o.v. this time. The same upward trends were observed as in the previous part, as expected. Also, the ratio of path length in the discovered gridworld vs. the complete gridworld, as well as the number of cells processed, were both noticeably higher in the no f.o.v. case than in the with f.o.v. case. This aligns with the fact that with no f.o.v., less discovery is done by the agent, meaning that the discovered gridworld is missing more information compared to the complete gridworld, and also that more re-planning steps have to be taken. Surprisingly, running the agent with no f.o.v. did not incur a noticeable cost in the trajectory length of the agent (see Figure 5a vs. 6a).



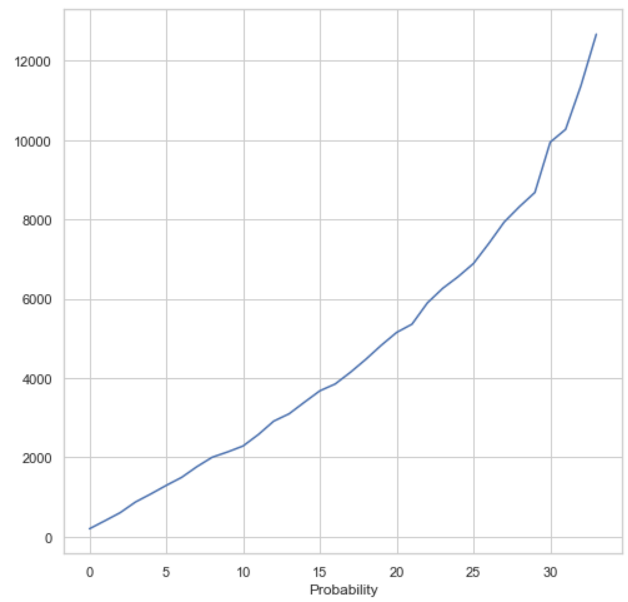
(a) Trajectory Length



(b) Path Length / Path Length in Discovered Gridworld

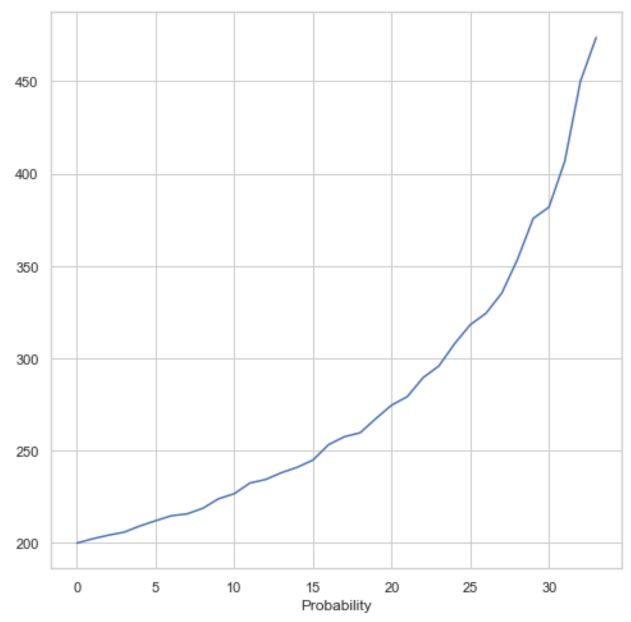


(c) Path Length in Discovered / Path Length in Full Gridworld

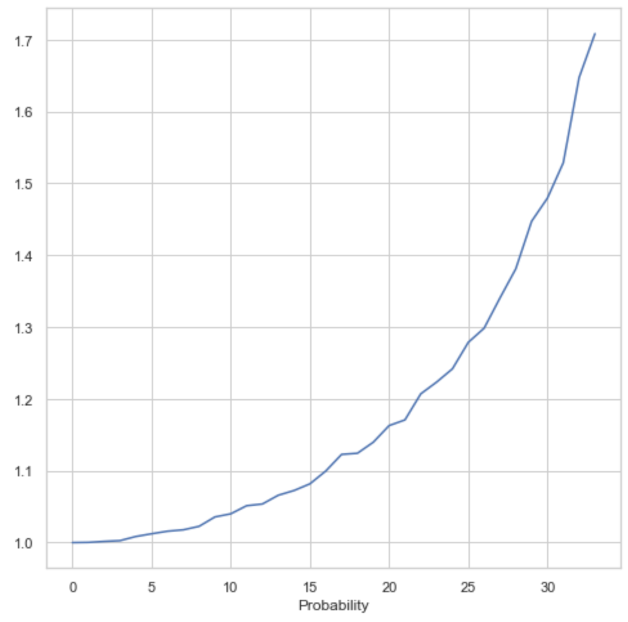


(d) Number of Cells Processed

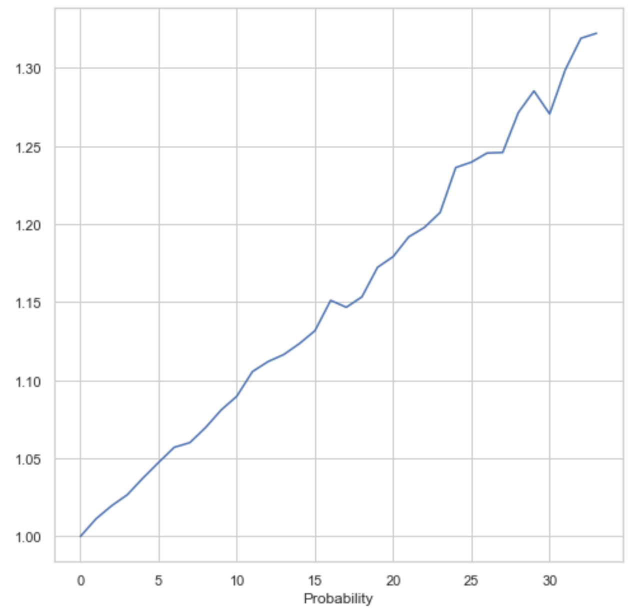
Figure 5: Data for Repeated A^* with f.o.v.



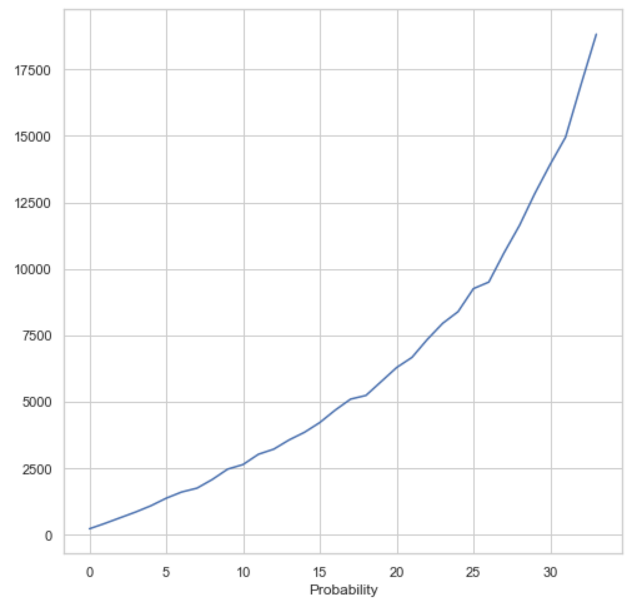
(a) Trajectory Length



(b) Path Length / Path Length in Discovered Gridworld



(c) Path Length in Discovered / Path Length in Full Gridworld

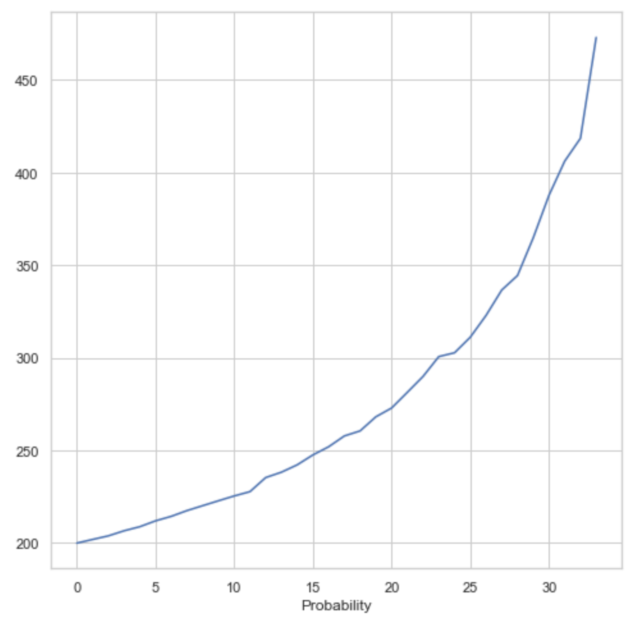


(d) Number of Cells Processed

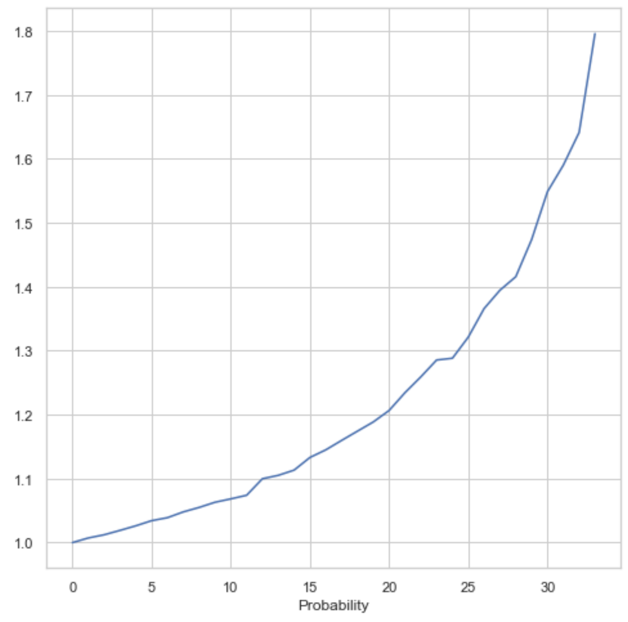
Figure 6: Data for Repeated A^* with no f.o.v.

3.1 Extra Credit: Repeated BFS

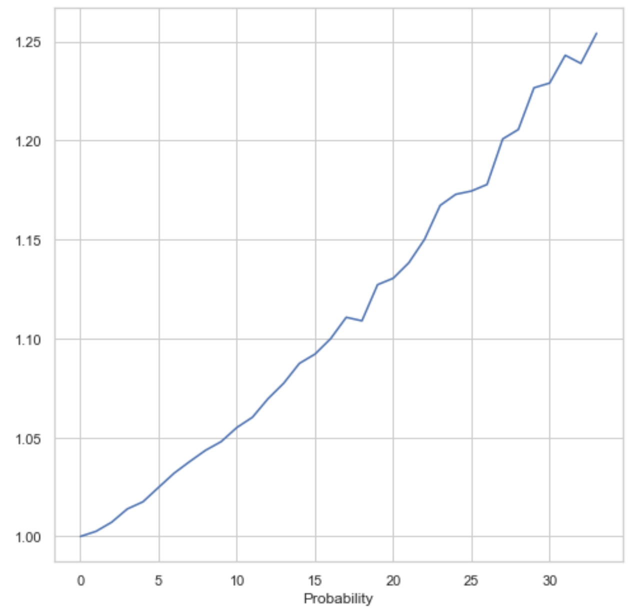
The same tests as in Questions 6 and 7 were run, using BFS as the driving search algorithm. The results are in Figure 7 (for Question 6) and Figure 8 (for Question 7). Following the discussion in Question 5, since in this situation (uniform edge-costs) BFS is equivalent to A^* with a zero-cost heuristic, we expect Repeated BFS to perform much worse in terms of runtime and number of cells expanded. This is supported by Figures 7d and 7d, where the number of cells expanded is magnitudes higher than what we saw with Repeated A^*. There were not any major differences in the other metrics (path length, ratios of path lengths etc.). We conclude that there is no benefits to using Repeated BFS over Repeated A^*.



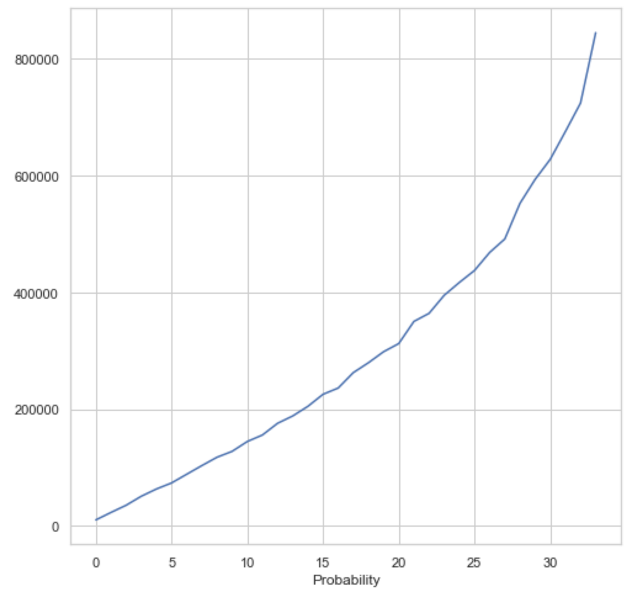
(a) Trajectory Length



(b) Path Length / Path Length in Discovered Gridworld

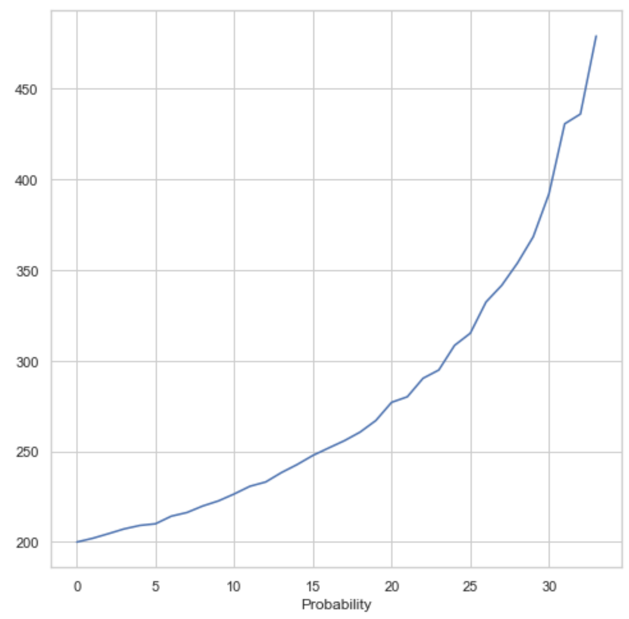


(c) Path Length in Discovered / Path Length in Full Gridworld

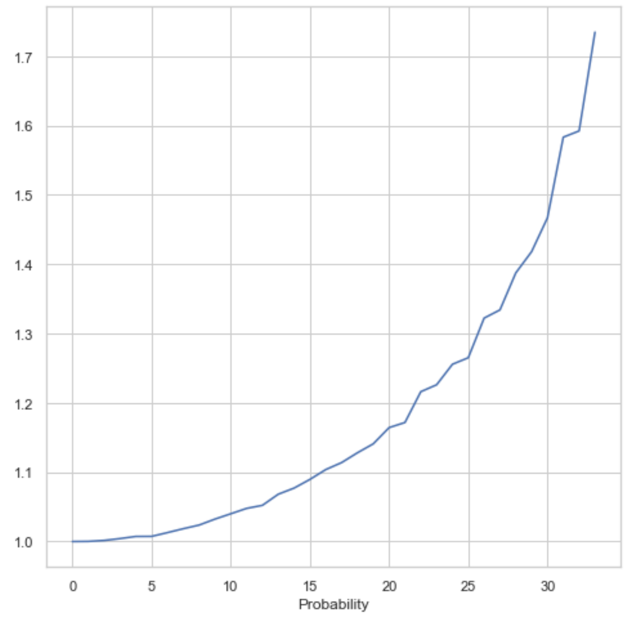


(d) Number of Cells Processed

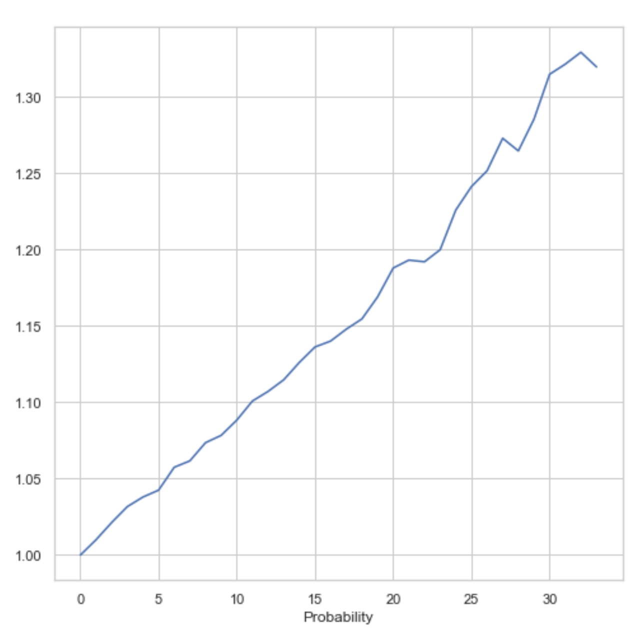
Figure 7: Data for Repeated BFS with f.o.v.



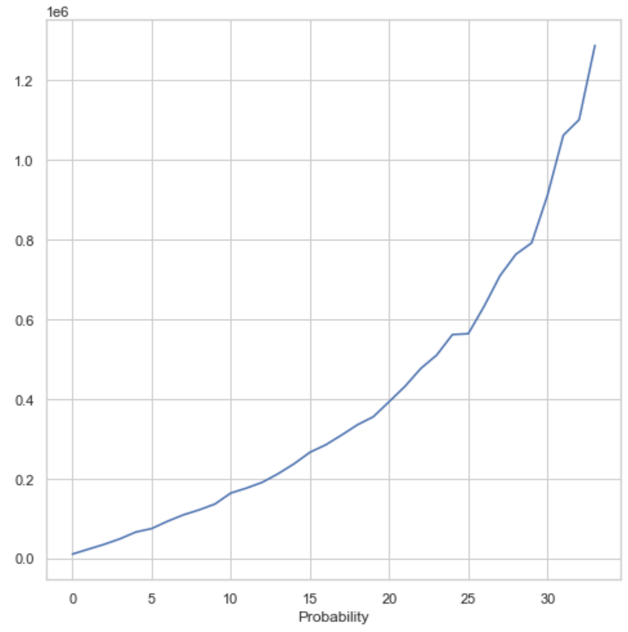
(a) Trajectory Length



(b) Path Length / Path Length in Discovered Gridworld



(c) Path Length in Discovered / Path Length in Full Gridworld



(d) Number of Cells Processed

Figure 8: Data for Repeated BFS with no f.o.v.

4 Improvements and Variants

Question 8. Implementation of Backtracking

Overall as we will see below for a grid of DIM 101 the optimal backtracking distance is 1; however in select cases a distance of approx. 10 percent of dim is better for run-time.

- **Relationship Between Trajectory Path and Backtracking Distance (Figure 9a:)** As expressed by the figure. At low probabilities (where a majority of mazes are solvable) longer back tracking increases trajectory length. This is expected because with relatively few obstacles when an obstacle is hit it will most likely be isolated. This leads the unnecessary backtracking and thus our planned trajectories grow at larger as the back-tacking step size grows. Specifically when backtracking equals 51 every obstacle will either restart the search entirely or bring the algorithm back to the halfway point leading to a large redundancy of steps. Therefore distance and trajectory are heavily correlated

- **Relationship Between Number of Cells Processed and Backtracking Distance (Figure 9b:)** The graph shows a tight grouping as for all steps at low probabilities. This is to be expected as there will be few if any recalculations. As the probability of a blocked space increases the grouping spreads a bit more. As we see the number of cells processed is only heavily affected with larger back tracking steps. This is mainly due to the backtracking covering retracing over open cells. The highest divergence is seen with backtracking distance equaling 51 steps this is because a majority of the grid will be blank space, but one obstacle will cause the algorithm to search at most half the grid again. Therefore distance and number of cells processed are correlated, but as the density of obstacles increase the number of cells processed will increase. From looking at the figure one is able to conclude that distances greater than 36 are only beneficial at lower probabilities due to lack of recalculations

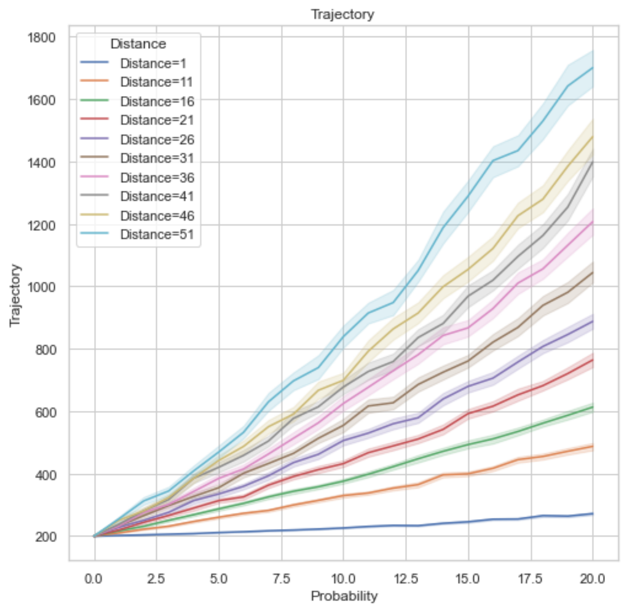
- **Relationship Between Run-time and Backtracking Distance (Figure 9c:)** This graph poses interesting questions and correlations. As more obstacles are introduced there will be an increase of recalculations. Each recalculation is effectively done at most n-1 times again (where n is back tracking distance) this means that 51 step back tracking will result in at most 50 recalculated steps in the worst case. As more obstacles become present our redundant calculations pile up increasing run-time. Interesting to note however, that at times there are points where higher back tracking distances outperform lower backtracking distances and this comes from that fact that the robots movement and "bumping" also factor into run-time. so as the obstacles grow there are instances where backtracking further are beneficial (i.e. in a blocked hallway or corner) . Ultimately the distance of 1 and 11 for backtracking provide the overall best performance. Where a distance of 11 beats a distance of 1 is in the case of a hallway or corner blockage.

Question 9. Using weighted heuristics.

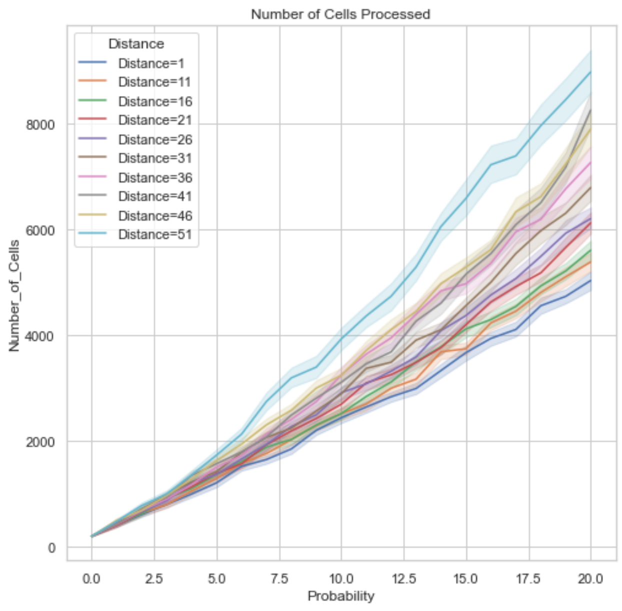
By ranking nodes via the modified f-cost $f(x) = g(x) + \epsilon h(x)$ (where $\epsilon \geq 1$), we can cut down on the runtime while incurring a small cost to the path length. We ran the regular A^* on complete gridworlds with different weights while holding the density constant, and collected various data which is displayed in Figure 10. As can be seen in the figure, these tests were done twice, at the two different density levels $p = 15\%$ and $p = 25\%$.

The data at both density levels follow the same trends, so aside from the additional costs associated with a higher density, the behaviour of weighted heuristics does not seem to vary between density values. We see that as the weight increases from 1 to 2, there is a sharp decrease in the runtime and the number of cells processed, which is accompanied by an equally sharp increase in the length of the path found. After that, the curves level off, and increasing the weight past 3 does not have any impact on any of the values. The optimal value for the weight appears to be just before 2, e.g. $\epsilon = 1.8$. This weight value allows the benefits of faster performance to be reaped, while only occurring an additional cost length of 20-25 (for **dim** $= 101$).

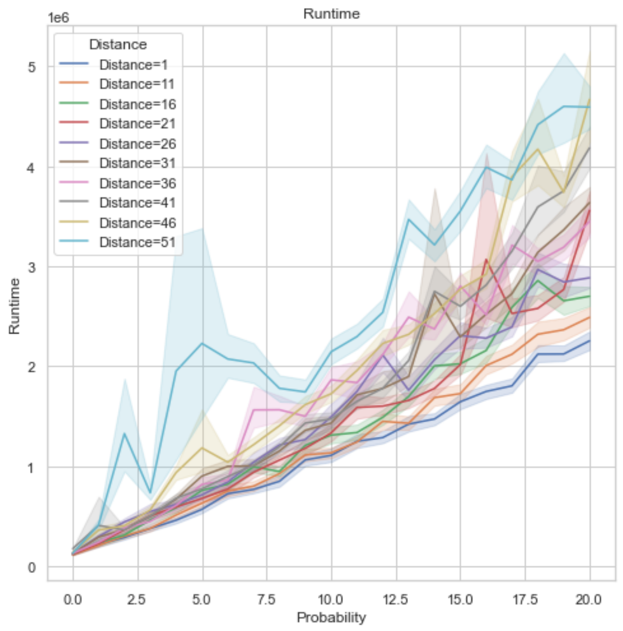
We also collected the same data using weight heuristics with Repeated A^* (and with no initial knowledge of the gridworld). The results of this can be seen in Figure 11. Unlike with regular A^*, there don't appear to be any benefits (or downsides) to using weighted heuristics with Repeated A^*.



(a) Trajectory Length vs Backtracking Distance

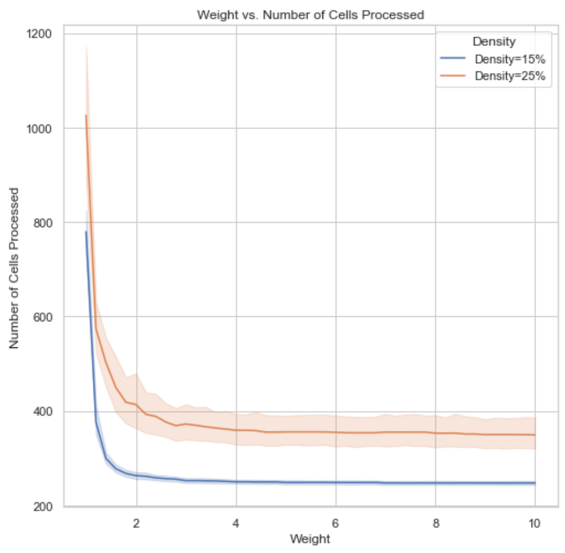


(b) Number Of Cells Processed Vs Backtracking Distance

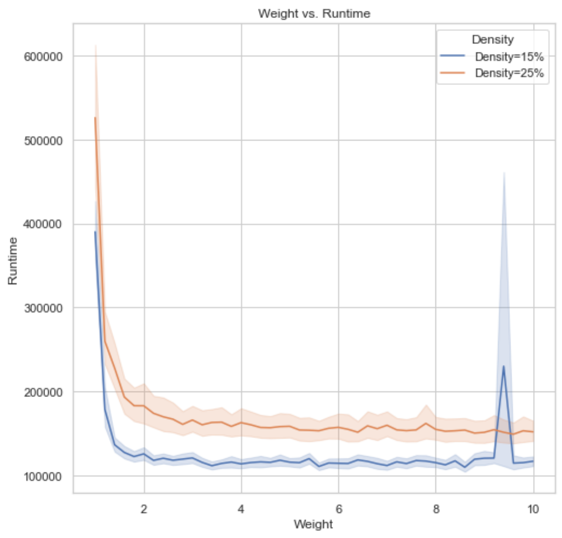


(c) Process Time Vs Backtracking Distance

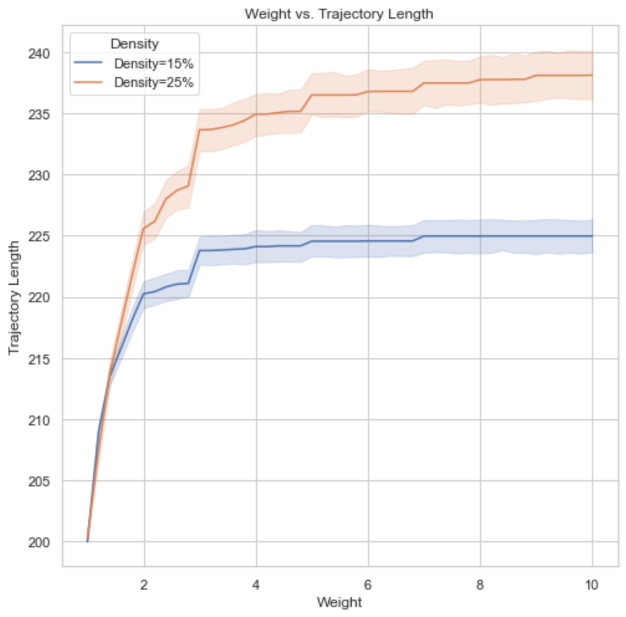
Figure 9: Measuring Viability and Efficiency of Repeated A^* with Backtracking



(a) Number of Cells processed

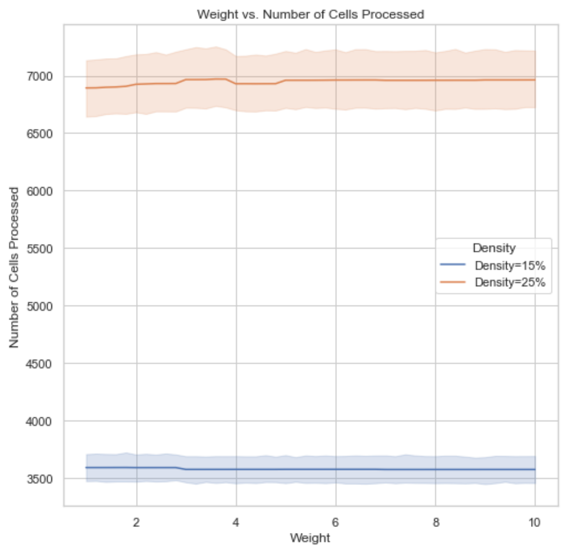


(b) Runtime

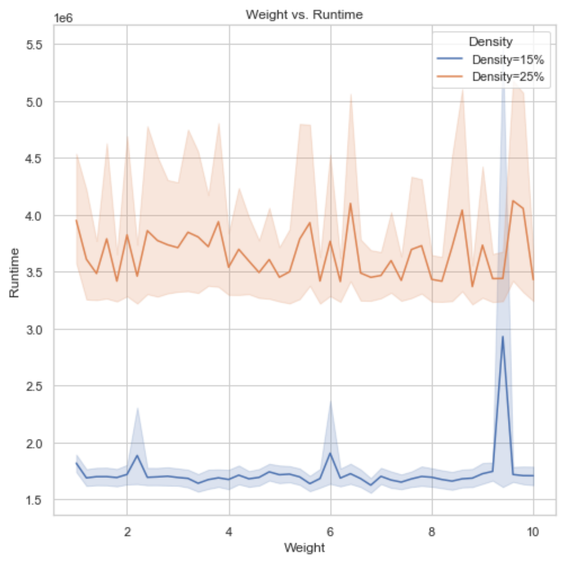


(c) Path Length

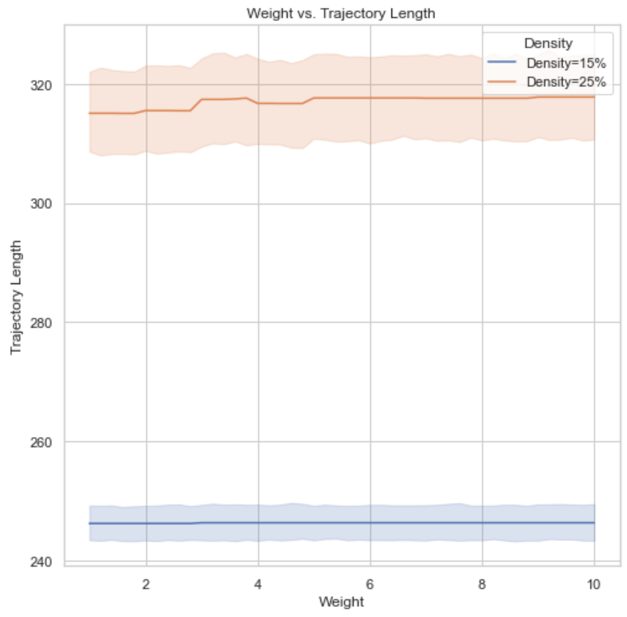
Figure 10: Measuring efficacy of weighted heuristics



(a) Number of Cells processed



(b) Runtime



(c) Path Length

Figure 11: Measuring efficacy of weighted heuristics on Repeated A^*

A Code for the Main Functions

Full Java Docs https://pietropaolov.github.io/A-Search-520/
A* Search Algorithm

```java

@Override
public GridWorldInfo search(Tuple<Integer, Integer> start, Tuple<Integer, Integer> end,
Grid grid, Predicate<GridCell> isBlocked) {
    GridCell startCell = grid.getCell(start);
    GridCell endCell = grid.getCell(end);
    if (startCell == endCell || startCell == null || endCell == null)
    return null; // Checks invalid cells

    // create fringe and process start cell
    // Fringe is a priority queue with a custom comparator
    PriorityQueue<GridCell> fringe =
    new PriorityQueue<>(new GridCellComparator()); // Priority Queue for reference
    startCell.setCost(0);
    startCell.setHeuristicCost(heuristic.apply(start, end));
    startCell.setPrev(null);
    fringe.add(startCell);

    // begin processing cells
    GridCell currentCell;
    double previousCost;
    HashSet<GridCell> discoveredCells = new HashSet<>();
    discoveredCells.add(startCell);
    int numberOfCellsProcessed = 0;
    while (!fringe.isEmpty()) {
        numberOfCellsProcessed++; //Cell Processed Counter
        currentCell = fringe.poll();// Get First in Queue
        previousCost = currentCell.getCost(); //Previous cost of the cell
        if (currentCell.equals(endCell)) { //check if end then start return obj creation
            // goal found, reconstruct path
            LinkedList<Tuple<Integer, Integer>> path = new LinkedList<>();
            while(currentCell.getPrev() != null) { // while we have not reached the start
    cell...
                path.push(currentCell.getLocation());
                currentCell = currentCell.getPrev();
            }
            return new GridWorldInfo(previousCost, numberOfCellsProcessed, path);
        }

        // else, generate the children of currentCell
        ArrayList<Tuple<Integer, Integer>> directions = new ArrayList<>(4);
        directions.add(new Tuple<>(currentCell.getX() + 1, currentCell.getY())); // right
        directions.add(new Tuple<>(currentCell.getX() - 1, currentCell.getY())); // left
        directions.add(new Tuple<>(currentCell.getX(), currentCell.getY() - 1)); // up
        directions.add(new Tuple<>(currentCell.getX(), currentCell.getY() + 1)); // down

        // process each child
        for(Tuple<Integer, Integer> direction : directions) {
            GridCell child = grid.getCell(direction);
            if(child == null || isBlocked.test(child))
                continue; // check that cell is valid
            if(!discoveredCells.contains(child)) {
            // first time processing initialize and insert into fringe
                child.setHeuristicCost(heuristic.apply(child.getLocation(), end));
                child.setCost(previousCost + 1);
                child.setPrev(currentCell);
                fringe.add(child);
                discoveredCells.add(child);
            } else if(previousCost + 1 < child.getCost()) {
            // already en-queued check priority for updates
```

```java
                boolean check = fringe.remove(child);
                child.setCost(previousCost + 1);
                child.setPrev(currentCell);
                fringe.add(child);
            }
        }
    }

// path not found
 return new GridWorldInfo(Double.NaN, numberOfCellsProcessed, null);

 //Custom Comparator for Priority Queue
 class GridCellComparator implements Comparator<GridCell> {

        @Override
        public int compare(GridCell o1, GridCell o2) {
            Double cost1 = o1.getCost() + o1.getHeuristicCost();
            Double cost2 = o2.getCost() + o2.getHeuristicCost();
            if(Double.compare(cost1, cost2) == 0) { // prefer higher g-cost over higher h-
    cost
                return Double.compare(o1.getHeuristicCost(), o2.getHeuristicCost());
            } else {
                return Double.compare(cost1, cost2);
            }
        }
    }
```

Breadth-First Search Algorithm

```java
 public static GridWorldInfo BFS(Tuple<Integer, Integer> start, Tuple<Integer, Integer> end,
     Grid grid, Predicate<GridCell> isBlocked) {
        GridCell startCell = grid.getCell(start);
        GridCell endCell = grid.getCell(end);
        if (startCell == endCell || startCell == null || endCell == null) return null;

        // create fringe and process start cell
        Queue<GridCell> fringe = new LinkedList<>();
        startCell.setCost(0);
        startCell.setPrev(null);
        fringe.add(startCell);

        // begin processing cells
        GridCell currentCell;
        double previousCost;
        HashSet<GridCell> discoveredCells = new HashSet<>();
        discoveredCells.add(startCell);
        int numberOfCellsProcessed = 0;
        while (!fringe.isEmpty()) {
            numberOfCellsProcessed++;
            currentCell = fringe.poll();
            previousCost = currentCell.getCost();
            if (currentCell.equals(endCell)) {
                // goal found, reconstruct path
                LinkedList<Tuple<Integer, Integer>> path = new LinkedList<>();
                while(currentCell.getPrev() != null) { // while we have not reached the
    start cell...
                    path.push(currentCell.getLocation());
                    currentCell = currentCell.getPrev();
                }
                return new GridWorldInfo(previousCost, numberOfCellsProcessed, path);
            }

            // else, generate the children of currentCell
            ArrayList<Tuple<Integer, Integer>> directions = new ArrayList<>(4);
            directions.add(new Tuple<>(currentCell.getX() + 1, currentCell.getY())); //
    right
```

```java
            directions.add(new Tuple<>(currentCell.getX() - 1, currentCell.getY())); // left
            directions.add(new Tuple<>(currentCell.getX(), currentCell.getY() - 1)); // up
            directions.add(new Tuple<>(currentCell.getX(), currentCell.getY() + 1)); // down

            // process each child
            for(Tuple<Integer, Integer> direction : directions) {
                GridCell child = grid.getCell(direction);
                if(child == null || isBlocked.test(child) || discoveredCells.contains(child)
    ) continue; // check that cell is valid and undiscovered
                child.setCost(previousCost + 1);
                child.setPrev(currentCell);
                fringe.add(child);
                discoveredCells.add(child);
            }
        }

        // path not found
        return new GridWorldInfo(Double.NaN, numberOfCellsProcessed, null);
    }
```

Main Running Object/Functionality

```java
    // attempt to follow a path, updating known obstacles along the way
    // stops prematurely if it bumps into an obstacle
    // returns the number of steps succesfully moved
    private int runPath(List<Tuple<Integer, Integer>> path, int backtrackDistance) {
        int numStepsTaken = 0;
        for(Tuple<Integer, Integer> position : path) {
            if(canSeeSideways) { // update obstacles based on fov
                ArrayList<Tuple<Integer, Integer>> directions = new ArrayList<>(4);
                directions.add(new Tuple<>(current.f1 + 1, current.f2)); // right
                directions.add(new Tuple<>(current.f1 - 1, current.f2)); // left
                directions.add(new Tuple<>(current.f1, current.f2 - 1)); // up
                directions.add(new Tuple<>(current.f1, current.f2 + 1)); // down

                for(Tuple<Integer, Integer> direction : directions) {
                    GridCell cell = grid.getCell(direction);
                    if(cell != null) {
                        if(cell.isBlocked()) addObstacle(cell);
                        else addFreeSpace(cell);
                    }
                }
            }

            GridCell nextCell = grid.getCell(position);
            if(nextCell.isBlocked()) { // if bump into an obstacle, stop
                addObstacle(nextCell);
                break;
            } else {
                numStepsTaken++;
                addFreeSpace(nextCell);
                move(position);
                if(numStepsTaken % backtrackDistance == 0){
                    setRestartPoint(position);
                }
            }
        }
        return numStepsTaken;
    }

    //Main Run Function
    //run through path until goal recalulating and moving a predefined number of backtrack
    spaces
    //compile all statistics from previous attempted path runs/calculations
     public GridWorldInfo run(int backtrackDistance) {
        GridWorldInfo gridWorldInfoGlobal = new GridWorldInfo(0, 0, new ArrayList<>());
        // loop while robot has not reached the destination
        while(!getLocation().f1.equals(getGoal().f1) || !getLocation().f2.equals(getGoal().f2)) {
            // find path
            GridWorldInfo result = getSearchAlgo().search(getLocation(), getGoal(), getGrid(), getKnownObstacles()::contains);

            // if no path found, exit with failure
            if(result == null || result.getPath() == null) {
                gridWorldInfoGlobal.setPath(null);
                gridWorldInfoGlobal.setTrajectoryLength(Double.NaN);
                return gridWorldInfoGlobal;
            }
            // attempt to travel down returned path, and update statistics
            int stepsTaken = runPath(result.getPath(), backtrackDistance );
            gridWorldInfoGlobal.addTrajectoryLength(stepsTaken);
            gridWorldInfoGlobal.addCellsProcessed(result.getNumberOfCellsProcessed());
            gridWorldInfoGlobal.getPath().addAll(result.getPath().subList(0, stepsTaken));
        }

        return gridWorldInfoGlobal;
    }
```

Utility Object Tuple

```java
public class Tuple<X, Y> {
        public final X f1;
        public final Y f2;
        public Tuple(X f1, Y f2) {
            this.f1 = f1;
            this.f2 = f2;
        }


    /**
     * Tests whether two tuples are equal by using the stored objects .equals method
     *
     * @param obj Tuple to be compared
     * @return true if both objects in the tuple are equal
     */
    @Override
    public boolean equals(Object obj) {
        Tuple<X,Y> tuple1 = (Tuple<X,Y> ) obj;
        return (tuple1.f1.equals(this.f1) && tuple1.f2.equals(this.f2));
    }

    @Override
    public String toString() {
        return "<" + f1 +
                "," + f2 +
                '>';
    }
}
```

Utility Object For Data

```java
public class GridWorldInfo {
    double probability;
    double trajectoryLength;
    double trajectoryLengthDiscovered; // running A* on the discovered gridworld
    double trajectoryLengthComplete; // running A* on the complete gridworld
    int numberOfCellsProcessed;
    long runtime;
    double weight; // weight of the heuristic used (EC)
    double backtrackSteps; // how many steps backwards are taken (EC)

    List<Tuple<Integer, Integer>> path; // path does not include start cell

    public GridWorldInfo(double trajectoryLength, int numberOfCellsProcessed, List<Tuple<
    Integer, Integer>> path) {
        this.trajectoryLength = trajectoryLength;
        this.numberOfCellsProcessed = numberOfCellsProcessed;
        this.path = path;

        // set default values for other data
        this.trajectoryLengthDiscovered = -1;
        this.trajectoryLengthComplete = -1;
        this.runtime = -1;
        this.weight = 1;
        this.backtrackSteps = 0;
    }
}
```