Artificial Intelligence:

Assignment 3 - Technical Report

Albert Adamson

Department of Computer Science. Saint Joseph's University

CSC 362: AI

Dr. Forouraghi

October 21, 2025

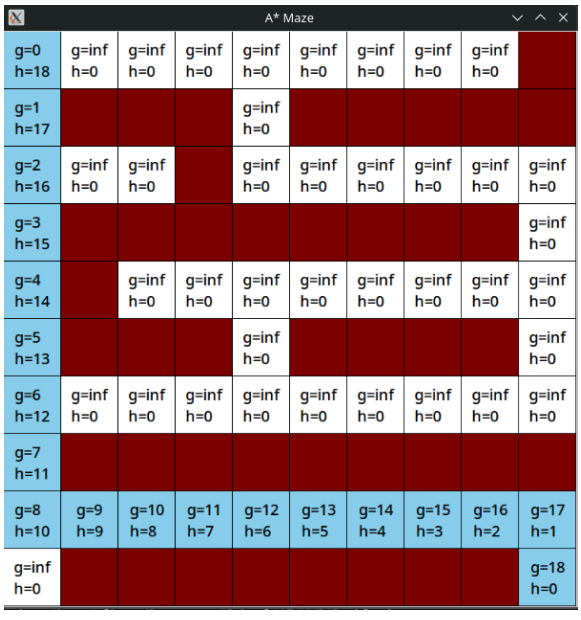
Experiment 1: Greedy Search vs A*

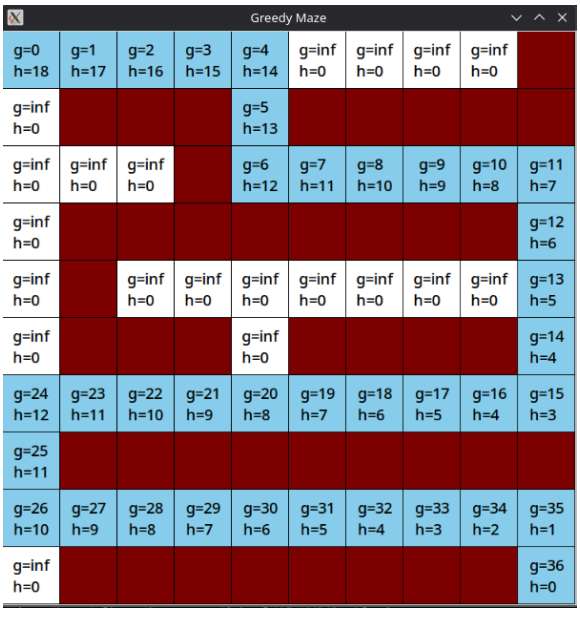
The purpose of this experiment is to see the difference between a greedy search algorithm and a regular A* search algorithm. When it comes to the code of them, the main algorithms are very similar, since they both use priority queues, but there is one main difference. In the regular A* algorithm, the evaluation function is f(n) = g(n) + h(n) where g(n) is the total cost to reach that node and h(n) is a heuristic function that is used to estimate how close the node is to the goal. A greedy function takes the A* evaluation function, and assumes that the estimate will always pick the best next step. In other words f(n) = h(n). In python the implementation looks something like this:

```
# Greedy Evaluation Function: f(n) = h(n)
if self.isGreedy:
    self.cells[new_pos[0]][new_pos[1]].f = self.cells[new_pos[0]][new_pos[1]].h
# A* Evaluation Function: f(n) = g(n) + h(n)
else:
    self.cells[new_pos[0]][new_pos[1]].f = new_g + self.cells[new_pos[0]][new_pos[1]].h
```

Note: This is the main modification that was made to implement the greedy function. Additionally, the MazeGame class had an isGreedy variable added to choose between the two.

When it comes to the result of the two, we can easily see the difference between them:





On left, we can see that the A* algorithm successfully picked the shortest path, but on the right, we can see that the path that the greedy algorithm picked happened to be twice as long. The reason the difference is so drastic is due to some changes I made in the actual maze. In the seventh row, I made it so that the only way to get to the goal was through the left most side. I made this choice because in the original maze, the A* function followed the same path as the Greedy function and reached the goal via the rightmost side of the seventh row. In the current maze, the A* algorithm recognizes that the best path is going straight down since it is also keeping track of the path count, while the Greedy best search just continues to pick the best move at the current instance.

Experiment 2:

Experiment 2 introduced 3 new variables into the algorithm presented in experiment 1. Firstly, the heuristic function was changed from the manhattan distance to being the euclidean distance. Secondly, the agent is allowed to make diagonal moves (NE, NW, SE, SW). Lastly, the moves are made randomly and not in any specific order. The first change changes the heuristic function from measuring the number of N, E, S, and W movements to measuring the distance between the two objects if they were to have a straight line drawn between them. While this may seem like an insignificant change, it changes a lot since it extends to diagonal movements. This leads into the next change, which allows the agent to move on diagonals, so rather than moving down and right taking up two movements, the agent can move diagonally to the bottom-right. As a result, this can decrease the total path cost compared to Experiment 1. The last change makes it so that the agent chooses the directions to move in a random order, however this last change doesn't have

much effect on the results of this experiment since the euclidean distance is weighted much more between diagonal movements. The above variables were introduced with the following python code:

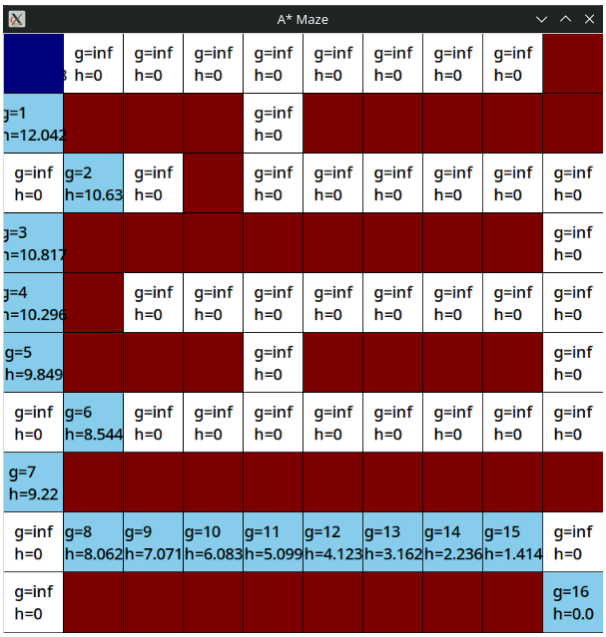
```
#############################################################
#### Euclidean distance
#############################################################
def heuristic(self, pos):
    return round(sqrt((pos[0] - self.goal_pos[0])**2 + (pos[1] - self.goal_pos[1])**2), 3)
```

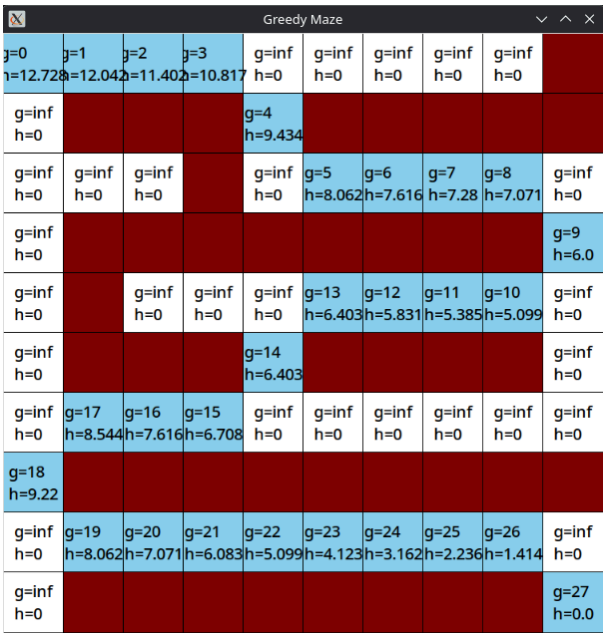
```
#### Agent goes E, W, N, S, NE, NW, SE, and SW whenever possible
choices = [(0, 1), (0, -1), (1, 0), (1, 1), (1, -1), (-1, 0), (-1, 1), (-1, -1)]
while len(choices) != 0:
    # Pick a random element then remove it from the list (List is recreated every outer iteration)
    direction = choices[random.randint(0, len(choices) - 1)]
    choices.remove(direction)

    # Update the new position based on the new direction
    dx, dy = direction[0], direction[1]
    new_pos = (current_pos[0] + dx, current_pos[1] + dy)
```

The first snip-bit is of the change in the heuristic function where it essentially calculates the hypotenuse of the triangle generated by a given point to the end goal, rounding it to 3 decimal points. The second snip-bit shows both the diagonal movements and the randomness. Firstly, the diagonal movements were added to the choices array and can be found in this set {(1,1),(1,-1),(-1,1),(-1,-1)}. The randomness comes from the next lines. I first check that the choices array is not empty, since the random algorithm removes elements that were already used. I then pick a random element then remove it from the choices array. Each iteration of the while loop slowly removes the choices the agent can make until it has taken every direction.

As a result of these changes, this is the new outputs:





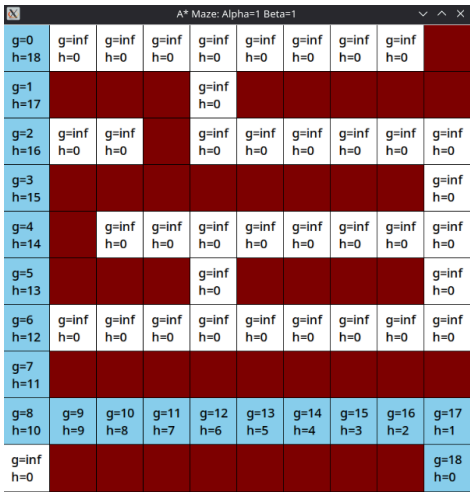
By adding the extra diagonal diagonal movements, the A* algorithm was able to decrease the total path cost from 18 to 16, and similarly the Greedy Best-First algorithm was able to decrease its cost from 36 to 27 (A difference of 9 nodes). Interestingly, to note the difference between the two path costs went from 18 to 11, but regardless A* was still able to find the best path. However, looking at the second to last node, it seems like the heuristic has a value of 1.414, which is greater than the cost to move to the final node. This means that the node is no longer admissible since we overestimated the heuristic, but regardless, in this specific case we still were able to find the optimal path. Also, the randomness didn't seem to have an effect on the results of the maze. This is likely due to the fact that the new heuristic is heavily weighted on the diagonal and the fact that the algorithm uses floating point numbers so it makes it harder for two nodes to have the same value. As a result of these changes, the algorithms' path costs were able to be decreased, however, at the expense of the admissibility.

Experiment 3:

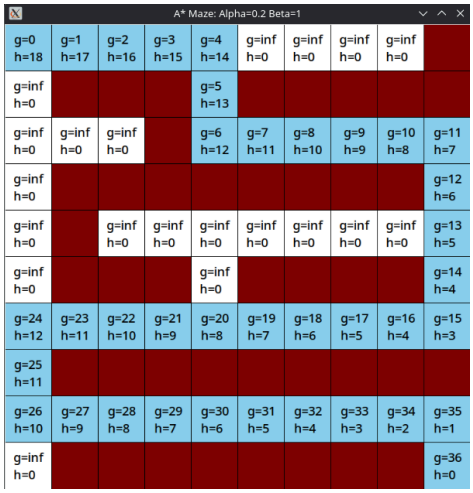
Experiment 3 reverts the all of the changes to back the original A* algorithm, so this means the heuristic is the original manhattan distance, there is no greedy function, and no randomness. However,the main difference is that the g(n) and the h(n) in the evaluation function are now weighted. This means that g(n) and h(n) are multiplied by specific values to increase/decrease their importance in the evaluation function. The purpose of this experiment is to show how leaning towards a more greedy algorithm can either improve/change the results of the original A* algorithm. The main change to the code can be found here:

```
# Update the evaluation function for the cell n: f(n) = alpha * g(n) + beta * h(n)
self.cells[new_pos[0]][new_pos[1]].f = (self.alpha * new_g) + (self.beta * self.cells[new_pos[0]][new_pos[1]].h)
self.cells[new_pos[0]][new_pos[1]].parent = current_cell
```

As a baseline, the algorithm was ran with alpha and beta both as 1, similar to the original A*.



Now let's decrease the alpha until we get a different result than the original, which happens to be for alpha = 0.2 and beta = 1.



The results here mimic the results that we would expect if we were to run a Greedy Best-first algorithm. For each trial of the experiment, the two expected outcomes were either the first output or the second. However, for more indepth information, I added a tracker which counted the number of nodes which had a parent (How many were visited by the algorithm) which presented some interesting results. This is the code which counted the nodes:

```
#
#   Calculate Number of Nodes Visited
#
def calculateNodes(self):
    count = 1 # Start doesn't have a parent
    for row in self.cells:
        for cell in row:
            if cell.parent != None:
                count += 1

    print(f"Visited: {count} nodes")
```

The results of the trial:

Alpha Value	Beta Value	Number of Nodes	Solution
1	1	51	A
1	0.75	52	A
1	0.5	56	A
1	0.2	57	A
1	0	57	A
0.75	1	52	A
0.5	1	55	A
0.2	1	52	B
0	1	52	B

Note1: Solution A comes from the solution for the A algorithm and Solution B comes from the solution for the Greedy Best-First algorithm.

**Note2: Any values >= 0 could have been chosen for Alpha and Beta, but I stuck with values between 0 and 1 since it made it more clear to me that Alpha was less important than Beta and vice versa.

From the results of the table, it seems like two different trends start to appear. Firstly, when the beta value is decreased it results in the number of nodes evaluated to increase. This is likely because at lower beta values less emphasis is placed on the heuristic function and we start shifting more towards Dijkstra's shortest path algorithm. This algorithm tries to evaluate every possible option that the nodes which would result in significantly more memory usage compared to the original algorithm. Decreasing the alpha value causes more interesting results. At first, the values seem to increase (Going from 52-55), but all of a sudden they go back down to 52, when the new solution is chosen. This is likely due to one of the properties of greedy best-first search algorithm picking the locally best projected path, since it isn't constantly bouncing back and forth between different cells like Dijkstra's algorithm is. However, the best weights are happen to be the regular A* algorithm as it combines the strengths of both algorithms, producing the best solution most efficiently.

Concluding Remarks:

After all three experiments, it becomes clear how different modifications to the A* algorithm affects the results in efficiency and optimal path finding. In the first experiment, the A* algorithm was able to consistently find the shortest path, while the Greedy Best-First, which relied solely on the heuristic, resulted in solutions that were longer. Experiment 2, took things up a notch and introduced the euclidean distance heuristic, diagonal movement, and random movement patterns. While the random movement didn't affect much, the diagonal movement changed a lot, as the path cost was able to be significantly decreased for both the A* algorithm and the Greedy Best-First algorithm. Interestingly, it should that the A* was still able to find the

shortest path despite having its admissibility condition violated by the euclidean distance heuristic.

Finally, the last experiment showed how weighting the g(n) and h(n) of the A* algorithm influenced the behavior of the algorithm. For instance, lowering the alpha value pushed the algorithm towards more of a Greedy Best-First algorithm, while conversely lowering beta's value made it behave more like Dijkstra's algorithm. For this maze in particular, the results showed that the regular A* algorithm (Alpha = 1, Beta = 1) resulted in the best performance. All in all, these experiments demonstrated how small changes made to the A* algorithm can greatly change the optimality and efficiency of the informed search algorithm.