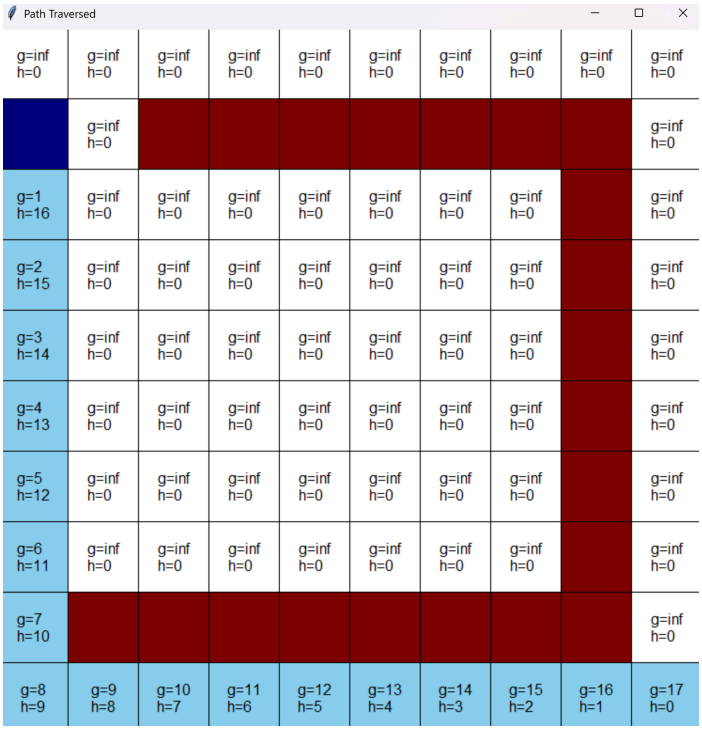
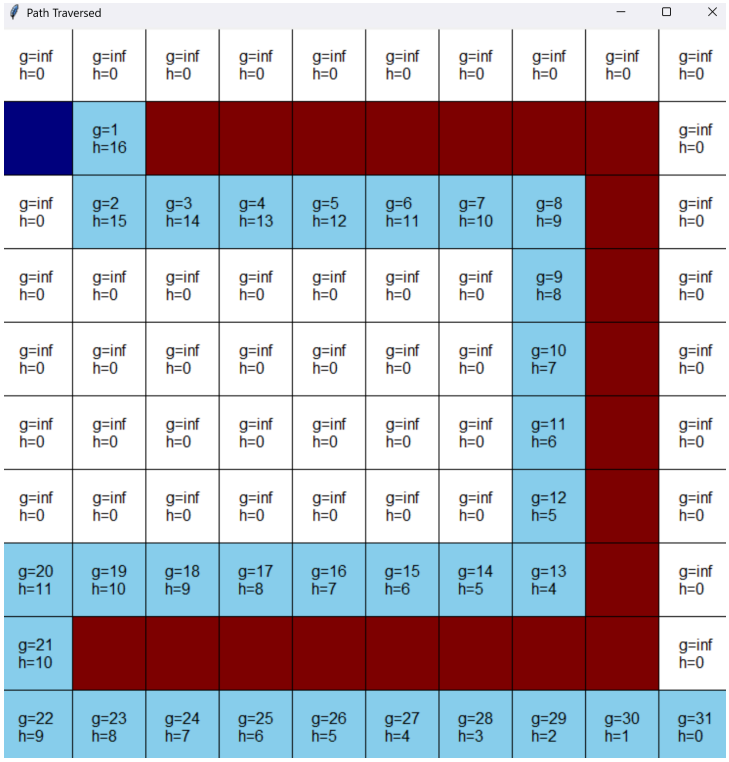
Problem 1:





The figure on the left shows the path traversed by the A* algorithm, which balances the g(n) and h(n) components of f(n). This balance allows A* to find an optimal path since it accounts for both the cost incurred and the estimated cost to reach the goal. The figure on the right shows the path traversed by the Greedy Best-First algorithm, where f(n) = h(n). This algorithm always pursues the locally best option because it only considers the estimated future cost to the goal. This is why the path immediately starts to move towards the bottom right corner before hitting walls and realizing that it must go around to reach the goal. A*, in contrast, picks the optimal path because of its ability to always think about the cost that has already been incurred. This contrast between A* and Greedy Best-First algorithms explains why A* is optimal with an admissible heuristic, while the Greedy Best-First is not.

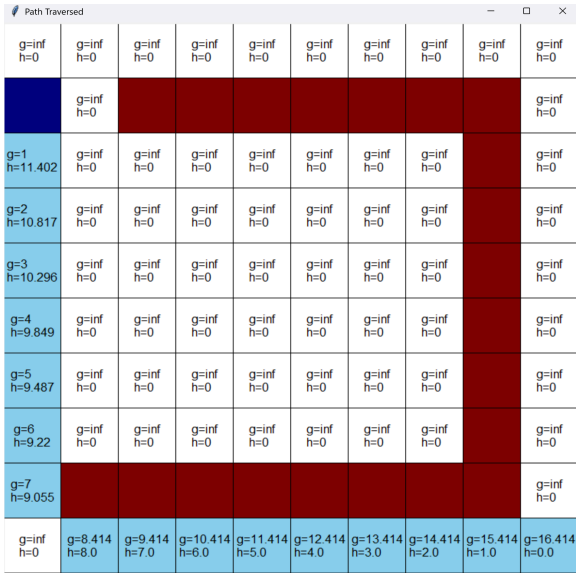
```
def __init__(self, root, maze):

    ### Set the search algorithm here: "A_star" or "Greedy"
    self.algorithm = "Greedy" # Change as needed
```

```
93      ############################################################
94      #### Manhattan distance
95      ############################################################
96      def heuristic(self, pos):
97          return (abs(pos[0] - self.goal_pos[0]) + abs(pos[1] - self.goal_pos[1]))
```

```
# Calculate the new heuristic and evaluation function values
new_h = self.heuristic(new_pos)
if self.algorithm == "A_star":
    new_f = new_g + new_h # if A* search -- f(n) = g(n) + h(n) -- since h(n) is fixed for each cell, this is the same as comparing just g(n) values
else:
    new_f = new_h # if Greedy Best-First search -- f(n) = h(n); no g(n) component
```

Problem 2:



In this example, I replaced the Manhattan Distance heuristic with Euclidean Distance and allowed diagonal moves in addition to the standard four directions (N, S, E, W). Since each diagonal step has a cost of sqrt(2), the heuristic remains admissible because it never overestimates the true cost of reaching the goal. With these changes, the Greedy Best-First search produces a more diagonal-looking path. This is because the Euclidean Distance heuristic naturally pulls the agent toward the straight-line direction of the goal. A* still balances the path cost already incurred with the heuristic estimate, which means it continues to guarantee the optimal path while making use of diagonals when efficient.

```
41  class MazeGame:
42      def __init__(self, root, maze):
43
44          ### Set the search algorithm here: "A_star" or "Greedy"
45          self.algorithm = "Greedy" # Change as needed
```

```
93      ############################################################
94      #### Euclidean distance
95      ############################################################
96      def heuristic(self, pos):
97          return  ((pos[0] - self.goal_pos[0]) ** 2 + (pos[1] - self.goal_pos[1]) ** 2) ** 0.5
```

```
121             #### The agent is allowed to make diagonal moves (i.e., NE, NW, SE, SW) in addition to the usual N, S, E, and W moves.
122             #### The moves are made randomly and not in any specific order.
123             moves = [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (1, -1), (-1, 1), (-1, -1)]
124             random.shuffle(moves) # this randomizes the order in which the moves are considered
125
126             for dx, dy in moves:
127                 new_pos = (current_pos[0] + dx, current_pos[1] + dy)
128
129                 if 0 <= new_pos[0] < self.rows and 0 <= new_pos[1] < self.cols and not self.cells[new_pos[0]][new_pos[1]].is_wall:
130
131                     ### Ensuring that the costs of moving to a new position are set correctly, so that the heuristic remains admissible.
132                     if dx == 0 or dy == 0:
133                         new_g = current_cell.g + 1 # The cost of moving to a new position is 1 unit if moving N, S, E, or W
134                     else:
135                         new_g = current_cell.g + (2**0.5) # The cost of moving diagonal (NE, NW, SE, SW) is sqrt(2) units
136
137
138                     # Calculate the new heuristic and evaluation function values
139                     new_h = self.heuristic(new_pos)
140                     if self.algorithm == "A_star":
141                         new_f = new_g + new_h # if A* search -- f(n) = g(n) + h(n) -- since h(n) is fixed for each cell, this is the same as comparing just g(n) values
142                     else:
143                         new_f = new_h # if Greedy Best-First search -- f(n) = h(n); no g(n) component
```

```
172             # Redraw cell with updated g() and h() values
173             self.canvas.create_rectangle(y * self.cell_size, x * self.cell_size, (y + 1) * self.cell_size, (x + 1) * self.cell_size, fill='skyblue')
174             # Round the values of g() and h() to 3 decimal places for better readability
175             text = f'g={round(self.cells[x][y].g, 3)}\nh={round(self.cells[x][y].h, 3)}'
176             self.canvas.create_text((y + 0.5) * self.cell_size, (x + 0.5) * self.cell_size, font=("Purisa", 12), text=text)
```

Problem 3a:

- Used Manhattan Distance; No diagonal moves allowed; All moves are a cost of 1; f(n) = Alpha*g(n) + Beta*(h(n)

Figure	Alpha	Beta	Observed Behaviors
1	1.0	1.0	This is the A* search, as Alpha and Beta are perfectly balanced. The traversed path is the same as the A* path from Problem 1. 61 nodes expanded.
2	1.0	0.5	The traversed path mirrors that of the A* search. 76 nodes expanded. This algorithm has less of a heuristic component, so it explores more. This path is still optimal.
3	1.0	0.0	The traversed path also mirrors that of the A* search. 78 nodes expanded. Since this does include the heuristic, the search space is not limited at all. This makes it the same as Dijkstra's algorithm/UCS search, which explains why it expands the largest number of nodes. This path is still optimal.
4	0.5	1.0	This is a greedier search algorithm due to Beta being twice as large as Alpha. It makes sense that 56 nodes expanded because it is more greedy than a balanced approach. This would not be an optimal path.
5	0.0	1.0	This is the Greedy Best-First search, as the past costs (g-values) are not considered. The path pursues the next node by looking for the minimal h-value, which is why this path "hugs" the wall throughout its path. Only 50 nodes were expanded because of how greedy the search algorithm is. This would not be an optimal path.

Figures 1, 2, and 3:

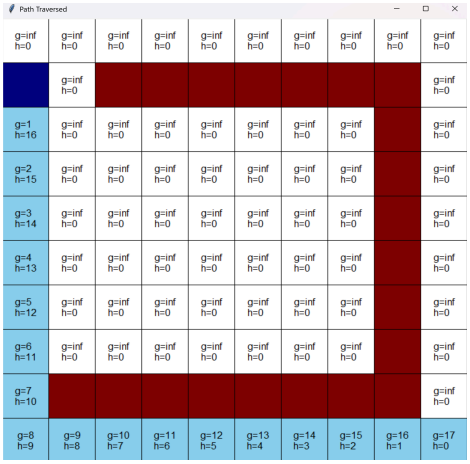
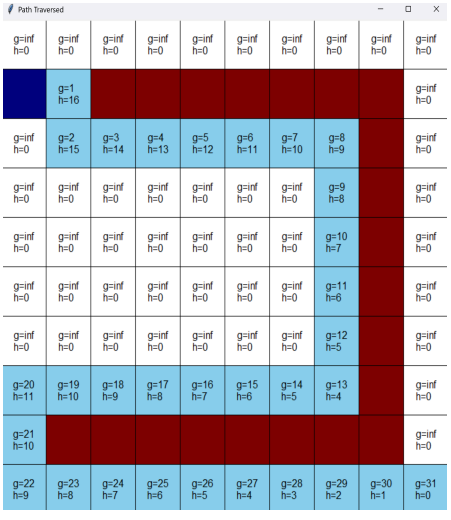


Figure 4:

Figure 5:



When alpha becomes larger than Beta (heavy weight on g(n)), the search behaves more like Dijkstra's algorithm. It stays optimal, but explores more nodes in the process. The extreme in this case is when Beta=0, which is a purely Dijkstra's algorithm. When Beta becomes larger than Alpha, the search gets more goal-directed. This means it expands fewer nodes but risks taking sub-optimal paths. The extreme in this case is when Alpha=0, which is a purely Greedy Best-First algorithm. When Alpha and Beta are balanced, this balance gives an optimal path with fewer expansions than Dijkstra's.

Problem 3b:

- Changing values of Beta

Figure	Alpha	Beta	Observed Behaviors
1	1.0	0.0	The traversed path mirrors that of the A* search. 78 nodes expanded. Since this does include the heuristic, the search space is not limited at all. This makes it the same as Dijkstra's algorithm/UCS search, which explains why it expands the largest number of nodes. This path is still optimal.
2	1.0	0.5	The traversed path mirrors that of the A* search. 76 nodes expanded. This algorithm has less of a heuristic component, so it explores more. This path is still optimal.
3	1.0	1.0	This is the A* search, as Alpha and Beta are perfectly balanced. The traversed path is the same as the A* path from Problem 1. 61 nodes expanded.
4	1.0	2.0	This is a greedier search due to Beta being larger than Alpha. 56 nodes expanded, since the algorithm leans more on the heuristic. This would not be an optimal path.
5	1.0	3.0	The greedy influence increases further. 58 nodes expanded, as the search continues to prioritize heuristic guidance. This would not be an optimal path.
6	1.0	7.0	The search behaves very similarly to the Greedy Best-First search, strongly favoring heuristic guidance. 51 nodes expanded. This would not be an optimal path.
7	1.0	15.0	This search is nearly identical to Greedy Best_first search, as the heuristic weight dominates. 51 nodes expanded. This would not be an optimal path.

Figure 1, 2, and 3:

Figure 4:

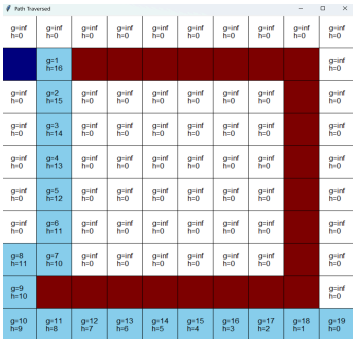


Figure 5:

Figure 6:

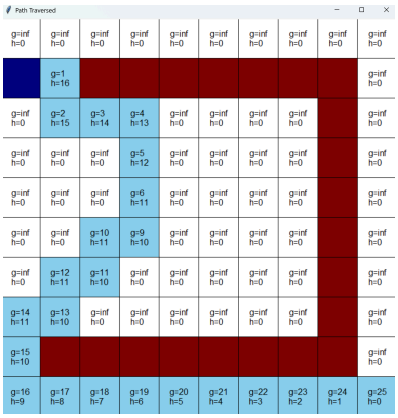
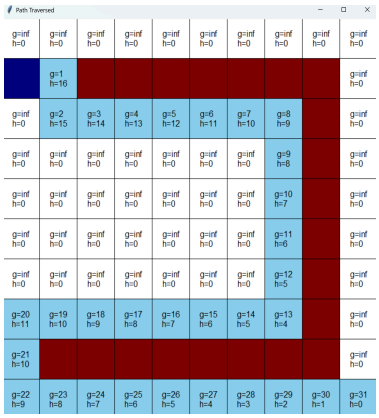


Figure 7:



As Beta increases relative to Alpha, the algorithm becomes more greedy. This causes it to expand fewer nodes but also makes it lose optimality. When Alpha and Beta are balanced (A*), the search is efficient and optimal. At the extremes, the search either mirrors Dijkstra's (when Beta =0) or Greedy Best-First search (when Beta is **much** greater than Alpha). This shows the trade-off between completeness, optimality, and efficiency.

Code Snippets from Problem 3:

```
35  ######################################################
36  # A maze is a grid of size rows X cols
37  ######################################################
38  class MazeGame:
39      def __init__(self, root, maze):
40
41          # Set the weights for g() and h() in the evaluation function f(n) = alpha * g(n) + beta * h(n)
42          self.alpha = 1.0 # Change as needed
43          self.beta = 15.0  # Change as needed
```

```
59          #### Start state's initial values for f(n) = g(n) + h(n)
60          # --- We use alpha and beta to weight g() and h() respectively
61          self.cells[self.agent_pos[0]][self.agent_pos[1]].g = 0
62          self.cells[self.agent_pos[0]][self.agent_pos[1]].h = self.heuristic(self.agent_pos)
63          self.cells[self.agent_pos[0]][self.agent_pos[1]].f = self.alpha * 0 + self.beta * self.heuristic(self.agent_pos)
```

```
92          ######################################################################
93          #### Manhattan distance
94          ######################################################################
95          def heuristic(self, pos):
96              return (abs(pos[0] - self.goal_pos[0]) + abs(pos[1] - self.goal_pos[1]))
```

```
129                 #### Stop if goal is reached
130                 if current_pos == self.goal_pos:
131                     self.reconstruct_path()
132                     # Printing information about the search for better analysis
133                     print(f"Alpha: {self.alpha}, Beta: {self.beta}")
134                     print(f"Number of expanded nodes: {expanded}")
135                     break
```

```
137             #### Agent goes E, W, N, and S, whenever possible
138             for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
139                 new_pos = (current_pos[0] + dx, current_pos[1] + dy)
140
141                 if 0 <= new_pos[0] < self.rows and 0 <= new_pos[1] < self.cols and not self.cells[new_pos[0]][new_pos[1]].is_wall:
142
143                     #### The cost of moving to a new position is 1 unit
144                     new_g = current_cell.g + 1
145
146                     # Calculate the new heuristic and evaluation function values
147                     new_h = self.heuristic(new_pos)
148
149                     # Use alpha and beta to weight g() and h() respectively
150                     new_f = self.alpha * new_g + self.beta * new_h
```