Introduction to Artificial Intelligence HW#1

Name: Lee JaeJun

Student ID: 2021313030

Abstract

This project employs Uniform Cost Search and A* Search algorithms to solve a maze, visualized through PyMaze on a 20x20 grid generated via DFS backtracking. Initially, it visualizes each algorithm's search process, revealing exploration steps before highlighting the optimal path. The final visualization accentuates the chosen path with circles, displaying the associated costs. This concise showcase not only illustrates the algorithms' effectiveness but also emphasizes the cost considerations in maze solving, offering insights into algorithmic efficiency in navigational challenges.

Introduction

This report explores how I used two methods, Uniform Cost Search and A* Search, to find the best path through a maze. The maze is a 20x20 grid and moving up or down costs a little more (1.1) than moving side to side (0.9). I added these methods to PyMaze that works with mazes.

Uniform Cost Search looks for the cheapest path, thinking about how much each step costs. A* Search is a bit different because it tries to guess how far it is to the end, making it faster. It also uses the same step costs.

I’ll talk about how I put these methods into our program and compare them to see which one does better in finding a way through the maze. I'll look at how many steps they take and how good the paths they find are.

Definition

Uniform Cost Search (UCS): The algorithm that always finds the least cost path from a start node to a goal node, assuming that the cost of each step is non-negative, and all nodes are accessible.

A* Search: The algorithm that finds the most efficient path from a start node to a goal node by combining the cost to reach a node and an estimate of the cost to reach the goal from that node.

Manhattan distance: The metric that calculates the total distance between two points in a grid-based system by summing the absolute differences of their x and y coordinates.

Step: The movement from one cell or position to an adjacent cell, counted as a single action taken by the search algorithm in its process of navigating from the start to the goal within the maze.

Cost: The cumulative value associated with traversing from the start position to the goal, calculated based on predetermined values assigned to each type of movement.

Optimal path: The most cost-effective route from the starting point to the goal within the maze, considering the predefined costs of vertical and horizontal movements.

Optimality: The strategy guaranteed to find the lowest cost path to a goal state.

Methodology

Maze Generation

The maze used for testing was generated using a depth-first search backtracking algorithm on a 20x20 grid. Each cell in the grid represents a potential step for the algorithm. To ensure consistency and replicability, a fixed seed value was used during the maze generation process.

Cost Assignment

Each horizontal step in the maze was assigned a cost of 0.9, and each vertical step a cost of 1.1. The cost of step used by each algorithm are all the same.

Uniform Cost Search (UCS) Implementation

The UCS algorithm was implemented with a priority queue to manage the frontier of exploration. The queue prioritizes nodes by the cumulative cost of reaching them from the start node, ensuring the algorithm always expands the least costly path available at any step.

Heuristic Function

For all nodes, the Manhattan distance to the target was calculated based on the absolute difference in grid coordinates, which reflects the cost difference between vertical and horizontal.

A* Search Implementation

The A* Search algorithm was similarly implemented with a priority queue, with nodes prioritized by the sum of the cost to reach them and an estimated cost to reach the goal using heuristic function.

Testing and Evaluation

Both algorithms were tested on the same set of mazes to compare their efficiency, effectiveness, and cost of the optimal path. The primary metrics for evaluation were:

Visualization

To aid in the qualitative assessment of each algorithm's pathfinding capability, visualizations were generated showing the explored paths, search steps, and the optimal path highlighted. Additionally, the final visualization presents each path with the optimal path and its associated cost highlighted using circles, offering a comprehensive view of the algorithm's performance.

Implementation Details

examples/solve_a_star_search.py

```python
from __future__ import absolute_import
from src.maze_manager import MazeManager

if __name__ == "__main__":

    # Create the manager
    manager = MazeManager()

    # Add a 20x20 maze to the manager
    maze = manager.add_maze(20, 20)

    # Save mp4 file and png
    # manager.set_filename("a_star_search")

    # Solve the maze using the A* Search algorithm
    manager.solve_maze(maze.id, "a_star_search")

    # Show how the maze was solved
    manager.show_solution_animation(maze.id)

    # Display the maze with the solution overlaid
    manager.show_solution(maze.id)
```

This file runs A* star search. It creates a maze according to the conditions and shows the navigation process. The maze generation is set to "dfs_backtrack" by default, so it didn't specify it. You can uncomment the annotated "manager.set_filename("a_star_search")" part to save the executable action .mp4 and .png files.

examples/solve_uniform_cost_search.py

```
from __future__ import absolute_import
from src.maze_manager import MazeManager

if __name__ == "__main__":

    # Create the manager
    manager = MazeManager()

    # Add a 20x20 maze to the manager
    maze = manager.add_maze(20, 20)

    # Save mp4 file and png
    # manager.set_filename("uniform_cost_search")

    # Solve the maze using the A* Search algorithm
    manager.solve_maze(maze.id, "uniform_cost_search")

    # Show how the maze was solved
    manager.show_solution_animation(maze.id)

    # Display the maze with the solution overlaid
    manager.show_solution(maze.id)
```

This file runs uniform cost search algorithm with the same structure as solve_a_star_search.py

src/solver.py

This is a file implementing the Uniform Cost Search and A* Search algorithms, along with the necessary functions for their execution.

```
def reconstruct_path_and_calculate_cost(came_from, start, goal):
    current = goal  # Start from the goal and work back to the start
    path = []  # Initialize the path list
    total_cost = Decimal('0.0')  # Initialize total cost with a decimal value of 0
    while current != start:  # Loop until the start cell is reached
        path.append((current, True))  # Add the current cell to the path. True means that it belongs to the optimal path
        previous_node = came_from[current]  # Get the previous cell from the current cell

        # Determine movement cost based on the direction (vertical or horizontal)
        if current[0] == previous_node[0]:
            total_cost += Decimal('0.9')  # Horizontal movement cost
        else:
            total_cost += Decimal('1.1')  # Vertical movement cost

        current = previous_node  # Move to the next cell in the path towards the starts
    path.append((start, True))  # Finally, add the start cell to the path
    path.reverse()  # Reverse the path to start from the beginning
    return path, total_cost  # Return the constructed path and the total cost
```

This function reconstructs the optimal path from the goal to the start cell in a maze and calculates the total cost of this path. It iterates backward from the goal, using a `came_from`

dictionary to trace each step taken. The path is marked with each cell belonging to the optimal route, and movement costs are calculated based on direction, with different costs for horizontal and vertical moves. The result is a list of cells representing the optimal path and the total cost associated with traversing this path.

```
def uniform_cost_search(maze):
    came_from = {}  # Map each cell to its predecessor in the path
    path = []  # List to keep track of the path taken
    start = maze.entry_coor  # Starting cell coordinates
    goal = maze.exit_coor  # Goal cell coordinates

    # Initialize g_score (cost from start to a cell) for all cells to infinity
    g_score = {(x, y): float('inf') for x in range(maze.num_rows) for y in range(maze.num_cols)}
    g_score[start] = 0  # Cost from start to itself is zero

    pq = PriorityQueue()  # Initialize the priority queue. It consists of (cost, (x, y))
    pq.put((0, start))  # Add the start cell with a priority of 0

    while not pq.empty():
        current = pq.get()[1]  # Get the cell with the lowest cost from the queue
        path.append((current, False))  # Mark the current cell as visited (for path visualization)
        maze.grid[current[0]][current[1]].visited = True  # Mark the cell as visited in the maze grid

        # Check if the current cell is the goal
        if current == goal:
            break

        # Iterate through all neighbours of the current cell
        for neighbour in maze.find_neighbours(current[0], current[1]):
            # Skip if the neighbour has been visited or if there is a wall between current and neighbour
            if maze.grid[neighbour[0]][neighbour[1]].visited or
maze.grid[current[0]][current[1]].is_walls_between(maze.grid[neighbour[0]][neighbour[1]]):
                continue

            cost = 1.1 if neighbour[0] == current[0] else 0.9  # Determine the movement cost to the
neighbour
            tentative_g_score = g_score[current] + cost  # Calculate tentative cost from start to the
neighbour

            # If this path to neighbour is better than any previous one, record it
            if tentative_g_score < g_score[neighbour]:
                g_score[neighbour] = tentative_g_score
                came_from[neighbour] = current  # Record the path
                pq.put((g_score[neighbour], neighbour))  # Add the neighbour to the queue with its updated
cost

    # After exploring all paths, reconstruct the optimal path and calculate its cost
    found_path, found_cost = reconstruct_path_and_calculate_cost(came_from, start, goal)
    path.extend(found_path)  # Combine the explored path with the optimal path for visualization
    return [path, found_path, found_cost]
```

This function implements the Uniform Cost Search algorithm for finding the optimal path in a maze from a specified start cell to a goal cell. It initializes a priority queue to manage exploration based on the cumulative cost from the start cell, marking each visited cell for visualization.

As it explores, it records the path taken by keeping track of each cell's predecessor and the cost to reach it. For each current cell, it examines all accessible neighbors, skipping any that have already been visited or are blocked by walls. This process utilized functions already implemented in pymaze.

The function calculates a tentative cost for reaching each neighbor and updates the neighbor's cost and predecessor if this new path is cheaper. Once the goal is reached or all possible paths are explored, the function reconstructs the optimal path from the start to the goal using the came_from dictionary, calculates the total cost of this path. By Adding an optional path to the end of the path, you can go through the navigation process and finally visually check the optional path. And then it returns both the path taken and the optimal path with its associated cost.

```
# Calculate the heuristic as the Manhattan distance with different costs for horizontal and vertical moves
def heuristic(cell_1, cell_2):
    x_1, y_1 = cell_1
    x_2, y_2 = cell_2
    return (abs(x_1 - x_2) * 0.9 ) + (abs(y_1 - y_2) * 1.1)
```

This function calculate heuristic using Manhattan distance. It reflects the cost of vertical and horizontal, respectively.

```
def a_star_search(maze, h):
    came_from = {}  # Map each cell to its predecessor in the path
    path = []  # List to keep track of the path taken
    start = maze.entry_coor  # Starting cell coordinates
    goal = maze.exit_coor  # Goal cell coordinates

    # Initialize g_score (cost from start to a cell) and f_score (estimated total cost from start to goal through a cell) for all cells to infinity
    g_score = {(x, y): float('inf') for x in range(maze.num_rows) for y in range(maze.num_cols)}
    g_score[start] = 0  # Cost from start to itself is zero
    f_score = {(x, y): float('inf') for x in range(maze.num_rows) for y in range(maze.num_cols)}
    start_heuristic = h(start, goal)  # Calculate the heuristic cost from start to goal
    f_score[start] = start_heuristic  # Initialize the f_score of the start cell (0 + heuristic)

    pq = PriorityQueue()  # Initialize the priority queue (f_score, heuristicm, (x, y))
    pq.put((f_score[start], start_heuristic, start))  # Add the start cell with its f_score to the queue

    while not pq.empty():
        current = pq.get()[2]  # Get the cell with the lowest f_score from the queue
        path.append((current, False))  # Mark the current cell as visited (for path visualization)
        maze.grid[current[0]][current[1]].visited = True  # Mark the cell as visited in the maze grid

        # Check if the current cell is the goal
```

```
        if current == goal:
            break

        # Iterate through all neighbours of the current cell
        for neighbour in maze.find_neighbours(current[0], current[1]):
            # Skip if the neighbour has been visited or if there is a wall between current and neighbour
            if maze.grid[neighbour[0]][neighbour[1]].visited or
maze.grid[current[0]][current[1]].is_walls_between(maze.grid[neighbour[0]][neighbour[1]]):
                continue

            cost = 1.1 if neighbour[0] == current[0] else 0.9  # Determine the movement cost to the
neighbour
            tentative_g_score = g_score[current] + cost  # Calculate tentative cost from start to the
neighbour
            tentative_f_score = tentative_g_score + h(neighbour, goal)  # Add heuristic to get f  score for
the neighbour

            # If this path to neighbour is better than any previous one, record it
            if tentative_f_score < f_score[neighbour]:
                g_score[neighbour] = tentative_g_score
                f_score[neighbour] = tentative_f_score
                came_from[neighbour] = current
                if not any(neighbour == item[1] for item in pq.queue):
                    pq.put((f_score[neighbour], h(neighbour, goal), neighbour))  # Add it to the priority
queue with its f  score

    # After exploring all paths, reconstruct the optimal path and calculate its cost
    optimal_path, optimal_cost = reconstruct_path_and_calculate_cost(came_from, start, goal)
    path.extend(optimal_path)  # Combine the explored path with the optimal path for visualization
    return [path, optimal_path, optimal_cost]
```

This code implements the A* search algorithm to find the optimal path in a maze from a start cell to a goal cell. It uses a combination of actual travel cost from the start (`g_score`) and the sum of g_score and an estimated cost to the goal (`f_score`) to prioritize cell exploration. The algorithm initializes both `g_score` and `f_score` for all cells to infinity, except for the start cell, where `g_score` is 0 and `f_score` includes the heuristic estimate to the goal. It uses a priority queue to efficiently select the next cell to explore based on the lowest `f_score`. If the f_score is the same, the cell having smaller heuristic value has a higher priority.

As the algorithm explores the maze, it keeps track of each cell's predecessor to reconstruct the path once the goal is reached. It marks each visited cell for visualization and skips over cells that have already been visited or are blocked. For each neighbor of the current cell, it calculates tentative `g_score` and 'f_score' based on the movement cost to that neighbor and updates the neighbor's scores and predecessor if this new score is better.

After reaching the goal or exhausting all possible paths, the algorithm reconstructs the optimal path from the start to the goal using the `came_from` map and calculates the total cost of this path. It returns a comprehensive visualization of the explored path, the optimal path, and the total cost of the optimal path.

Modifications and Enhancements

src/maze.py

```
35          self.num_cols = num_cols
36          self.num_rows = num_rows
37          self.id = id
38          self.grid_size = num_rows*num_cols
39          self.entry_coor = self._pick_random_entry_exit(None)
40          self.exit_coor = self._pick_random_entry_exit(self.entry_coor)
41          self.generation_path = []
42          self.solution_path = None  # for saving optimal path
43          self.solution_cost = 0 # for saving optimal path cost
44          self.path = None  # for saving search process and optimal path
45          self.initial_grid = self.generate_grid()
46          self.grid = self.initial_grid
47          random.seed(1)
48          self.generate_maze(algorithm, (0, 0))
```

Line 42: Although the variable name was not changed, unlike the existing PyMaze code, it was set and used as a variable that only stores the optimal path.

Line 43: Added a member variable that stores the optimal path cost.

Line 44: Added a member variable that stores a list that contains both the navigation path and the optimal path.

Line 47: For the consistent start and exit positions, set as random.seed(1) in accordance with the conditions.

src/algorithm.py

```
7   random.seed(0)
```

Line 7: Set as random.seed(0) in accordance with the conditions.

src/maze_manager.py

```
3  from src.solver import uniform_cost_search
4  from src.solver import a_star_search
5  from src.solver import heuristic
```

Line 3: Import uniform_cost_search function to execute in the maze.

Line 4: Import uniform_cost_search function to execute in the maze.

Line 5: Import heuristic function that used for the parameter of the a_star_search function.

```
120            if method == "uniform_cost_search":
121                result = uniform_cost_search(maze)
122                maze.path = result[0]
123                maze.solution_path = result[1]
124                maze.solution_cost = result[2]
125            elif method == "a_star_search":
126                result = a_star_search(maze, heuristic)
127                maze.path = result[0]
128                maze.solution_path = result[1]
129                maze.solution_cost = result[2]
```

Delete preexisting classes for BFS, DFS, Bidirectional search because I don't use them in my assignment. And then connect my own solutions.

Line 120 ~ 124: If the parameter `method` is set to 'uniform_cost_search' in the `solve_maze` function, execute uniform cost search for the provided maze. The result will be a list consisting of the path, optimal path, and optimal path cost. Assign the path (result[0]) to `maze.path`, the optimal path (result[1]) to `maze.solution_path`, and the optimal path cost (result[2]) to `maze.solution_cost`.

Line 125 ~ 129: If the parameter `method` is set to 'a_star_search' in the `solve_maze` function, execute A* search for the provided maze. A* search uses a heuristic function that implements the Manhattan distance. The result will be a list consisting of the path, optimal path, and optimal path cost. Assign the path (result[0]) to `maze.path`, the optimal path (result[1]) to `maze.solution_path`, and the optimal path cost (result[2]) to `maze.solution_cost`.

src/maze_viz.py

```
29            self.step = 0
```

Line 29: In order to record only steps for search, a member variable was created separately, unlike the previous method of counting steps by the number of frames.

```
97    def show_maze_solution(self):
98        """Function that plots the solution to the maze. Also adds indication of entry and exit points."""
99
100       # Create the figure and style the axes
101       fig = self.configure_plot()
102
103       # Plot the walls onto the figure
104       self.plot_walls()
105
106       # Keeps track of how many circles have been drawn
107       circle_num = 0
108
109       self.ax.add_patch(plt.Circle(((self.maze.solution_path[0][0][1] + 0.5)*self.cell_size,
110                                      (self.maze.solution_path[0][0][0] + 0.5)*self.cell_size), 0.2*self.cell_size,
111                                     fc=(0, circle_num/(len(self.maze.solution_path)),
112                                         0), alpha=0.4))
113
114       for i in range(1, self.maze.solution_path.__len__()):
115           circle_num += 1
116           self.ax.add_patch(plt.Circle(((self.maze.solution_path[i][0][1] + 0.5)*self.cell_size,
117               (self.maze.solution_path[i][0][0] + 0.5)*self.cell_size), 0.2*self.cell_size,
118               fc = (0, circle_num/(len(self.maze.solution_path)), 0), alpha = 0.4))
119       self.ax.set_title("Cost: {}".format(self.maze.solution_cost), fontname="serif", fontsize=19)
120       # Display the plot to the user
121       plt.show()
122
123       # Handle any saving
124       if self.media_filename:
125           fig.savefig("{}{}.png".format(self.media_filename, "_solution"), frameon=None)
```

I eliminated backtracking visuals from uniform cost and A* search since they expand by distance, rendering step-by-step backtracking impractical. The Boolean now marks search steps (false) and the optimal path (true), removing the need for backtracking checks in circle rendering for simpler visualization.

Line 111, 118: When calculating the value of the variable fc, '2*len(list_of_backtracks)' was removed.

Line 119: Added the code for showing cost.

```
247   def animate_maze_solution(self):
248       """Function that animates the process of generating the a maze where path is a list
249       of coordinates indicating the path taken to carve out (break down walls) the maze."""
250
251       # Create the figure and style the axes
252       self.step = 0
253       fig = self.configure_plot()
254
255
256       # Adding indicator to see shere current search is happening.
257       indicator = plt.Rectangle((self.maze.path[0][0][0]*self.cell_size,
258                                  self.maze.path[0][0][1]*self.cell_size), self.cell_size, self.cell_size,
259                                 fc="purple", alpha=0.6)
260       self.ax.add_patch(indicator)
261
262       self.add_path()
```

Line 257 ~ 258: To illustrate the search process, `path` is used instead of `solution_path`.

```
263
264         def animate_squares(frame):
265             """Function to animate the solved path of the algorithm."""
266             if frame > 0:
267                 if self.maze.path[frame - 1][1]:  # Show Optimal Path
268                     self.squares["{},{}".format(self.maze.path[frame - 1][0][0],
269                                                 self.maze.path[frame - 1][0][1])].set_facecolor("orange")
270
271                 self.squares["{},{}".format(self.maze.path[frame - 1][0][0],
272                                             self.maze.path[frame - 1][0][1])].set_visible(True)
273                 self.squares["{},{}".format(self.maze.path[frame][0][0],
274                                             self.maze.path[frame][0][1])].set_visible(False)
275             return []
276
```

Line 267: Previously, if `path[frame-1][1]` was true, it was considered part of the backtracking process and represented in orange. This definition has been changed so that if the value is true, it signifies the optimal path. As a result, only the optimal path will be highlighted in orange at the end.

Line 268 ~ 274: To illustrate the search process, `path` is used instead of `solution_path`.

```
277         def animate_indicator(frame):
278             """Function to animate where the current search is happening."""
279             indicator.set_xy((self.maze.path[frame][0][1] * self.cell_size,
280                               self.maze.path[frame][0][0] * self.cell_size))
281             return []
282
283         def animate(frame):
284             """Function to supervise animation of all objects."""
285             animate_squares(frame)
286             animate_indicator(frame)
287             if(self.maze.path[frame][1] is False):
288                 self.step = frame + 1
289             self.ax.set_title("Search Step: {}".format(self.step), fontname = "serif", fontsize = 19)
290             return []
291
292         anim = animation.FuncAnimation(fig, animate, frames=self.maze.path.__len__(),
293                                        interval=100, blit=True, repeat=False)
294
295         # Handle any saving
296
297         if self.media_filename:
298             print("Saving solution animation. This may take a minute....")
299             mpeg_writer = animation.FFMpegWriter(fps=24, bitrate=1000,
300                                                  codec="libx264", extra_args=["-pix_fmt", "yuv420p"])
301             anim.save("{}{}{}x{}.mp4".format(self.media_filename, "_solution_", self.maze.num_rows,
302                                              self.maze.num_cols), writer=mpeg_writer)
303
304         # Display the animation to the user
305         plt.show()
```

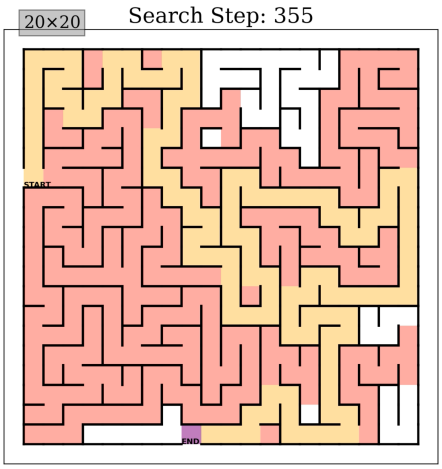
Line 279, 280, 287: To illustrate the search process, `path` is used instead of `solution_path`.

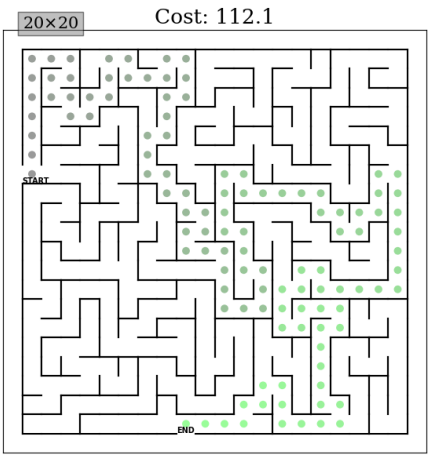
Line 287 ~ 289: I excluded the process of displaying the optimal path at the end from the step count.

Line 297 ~ 302: To capture the process of coloring the path in the video, I moved the code above `plt.show()`.

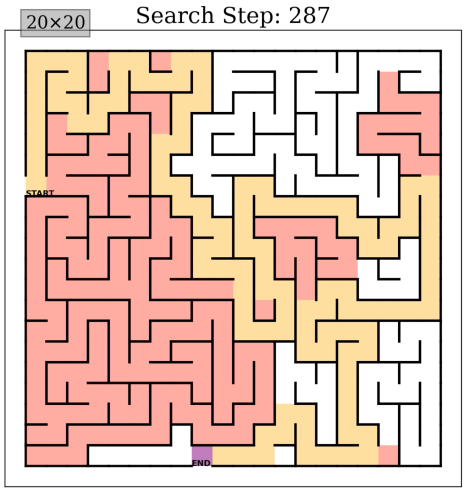
Results and Discussion

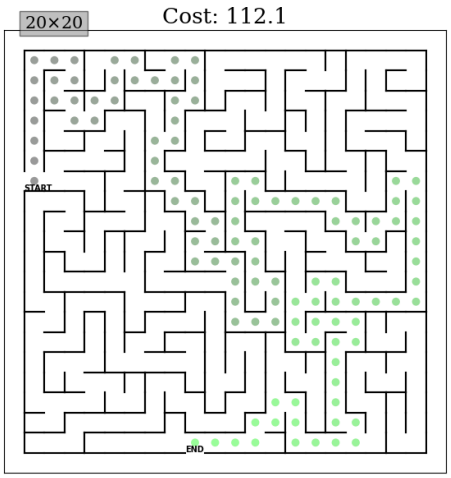
Uniform cost search





A* search





Uniform Cost Search and A* Search are both pathfinding algorithms with their unique strengths and weaknesses, particularly noticeable when comparing their performance in terms of steps taken and the optimality of the solution.

Uniform Cost Search, taking 355 steps to reach the goal, guarantees an optimal path by exploring all possible paths in order of their cumulative cost. This approach, while straightforward and reliable, can be inefficient in larger search spaces due to its non-discriminatory exploration of paths, potentially leading to high memory usage.

On the other hand, A* Search, with 287 steps, demonstrates greater efficiency by leveraging heuristic information to guide its search towards the goal, thus reducing the number of explored nodes.

Both algorithms ultimately achieve the same cost of 112.1, indicating their ability to find optimal solutions. However, A*'s dependency on the quality of the heuristic function means its performance can vary significantly; a well-designed heuristic function leads to fewer steps and more efficient pathfinding, highlighting A*'s advantage in scenarios where heuristic guidance is applicable.

In essence, while Uniform Cost Search offers a simpler, brute-force approach to finding the least cost path, A* Search optimizes the search process through heuristics, making it generally more suited for complex pathfinding tasks where efficiency is important.

References

- CS 188 Spring 2024. (2024). *Introduction to Artificial Intelligence*. University of California, Berkeley.

- Russell, S. J., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.