Search Programming Assignment

Design of the Search Algorithms for Part 1

In Part 1, the object was to find the shortest path from a given start state to a goal state. To do so, there were four main search algorithms to implement to compare the differences in their implementations and their results.

The four were: Breadth-first search (BFS), Depth-first search (DFS), Uniform-cost search (UCS), and A* search.

Starting with BFS, the code implements BFS in the traditional way by utilizing a queue so that the first items put in are the first ones to come out. This allows the search algorithm to search the grid wide and then move downward. This allows BFS to be a complete search algorithm.

```
# Initialize FIFO data structure with the start state
queue = [(start, [])]   # position, path
while queue:
    # Pull next
    current_pos, current_path = queue.pop(0)
```

From there, it will continue until there are elements in the queue. As it goes through the queue, it checks the current position and the path to it, and expands to its neighbors. If the neighbor hasn't been visited yet, it will add it to the queue to be explored and to the path.

```
# Get valid neighbors of the current position
neighbors = maze.getNeighbors(current_pos[0], current_pos[1])
for neighbor in neighbors:
    new_path = current_path + [current_pos]
```

For DFS, the implementation is almost identical, but instead of using a queue, it will use a LIFO data structure (a stack) so that the search algorithm will go downward instead of outwards. This means that DFS is not a complete search algorithm and isn't guaranteed to find the optimal path.

Moving to UCS, this is similar to BFS as it is a complete algorithm that is guaranteed to find an optimal solution. This algorithm is different from the other two because it keeps track of the cost to reach the goal states from the starting state.

It iterates through the list and looks for the state with the lowest cost to expand. Once on the current state, it will check that state's neighbors and add to their cost. Then, it will go back up and check the costs of all of the states on the ucs_list. The state with the lowest cost will be expanded and the process will continue until it finds the goal state.

```
# Sort ucs_list data structure to find lowest cost
for i in range(len(ucs_list)):
    cost = ucs_list[i][0]
    if cost < min_cost:
        min_cost = cost
        min_index = i
min_state = ucs_list.pop(min_index)  # Pull lowest cost state
cost, current_pos, current_path = min_state
```

The last search algorithm implemented for finding a single dot is A* search. This search method is often complete and cost-optimal because as long as the state space has a solution or is finite. For this situation, it will be complete and cost-optimal, and should be the most efficient search algorithm.

A* search works using a heuristic, which for this assignment will always be calculated using the manhattan distance from the position to the objectives.

```
def manhattan_distance(point1, point2):
    """
    Calculates the Manhattan distance between two points.
      @param point1: First point: (x1, y1).
    @param point2: Second point: (x2, y2).
      @return distance: The Manhattan distance between the two points.
    """
    return abs(point1[0] - point2[0]) + abs(point1[1] - point2[1])
```

A* search, similar to UCS, will keep track of cost, but the cost (fn) is calculated as a combination of the distance from position to objectives (gn), but also the calculated value from the heuristic function (hn). As it iterates through astar_list, it will take the state with the lowest estimated total cost and check its neighbors and continue that process. At each new state it explores, it will calculate the costs and assemble the path that way.

```
# Get valid neighbors of the current position
neighbors = maze.getNeighbors(current_pos[0], current_pos[1])
for neighbor in neighbors:
    new_path = current_path + [current_pos]

    gn = len(new_path)  # g-cost is the number of steps taken
    hn = manhattan_distance(neighbor, objectives[0]) # h-cost is heuristic
cost
    fn = gn + hn    # f-cost is the cost
```

Design of A* for Part 2

For this part, the objective was to find the shortest path through the maze, starting from P and touching all four corners. As explained in the problem, the most optimal path may not be one that starts from the objective closest to the starting point, so there were a lot of iterations necessary to get the true optimal path.

The logic is very similar to Part 1, but the difference is in the values contained in astar_list. Here, astar_list contains five parameters instead of 3:

- hn: heuristic cost
- gn: cost from start state to current state
- fn: total estimated cost
- current_pos: current position in the maze
- collected: set of indices representing the corners collected
- path: current path taken to reach the current state

The heuristic here is also manhattan distance, but it instead calculates the manhattan distance to each of the corners and picks the best option based on the results.

```
# Define the heuristic (manhattan distance, but to each corner)
def heuristic(state):
    # Calculate maximum Manhattan distance to all unvisited corners
    max_distance = 0
    for obj in objectives:
        found = False
        for i in range(len(objectives)):
           if state[0] == obj and i not in state[1]:
            distance = abs(state[0][0] - obj[0]) + abs(state[0][1] - obj[1])
                max_distance = max(max_distance, distance)
                found = True
                break
        if found:
            break
      return max_distance
```

For this part, since there are multiple objectives, it keeps track of which objectives have been “collected” and tracks which sets have been “explored” so that it can know which states to add to the frontier.

```
# Check if all corners have been visited
if len(collected) == len(objectives):
    return path
# Check if current state has been explored
if (current_pos, tuple(sorted(collected))) in explored:
    continue
else:
    explored.add((current_pos, tuple(sorted(collected))))  # Mark
as explored
```

Design of A* for Part 3

In the final part, the objective was to find the shortest path through a maze while hitting multiple dots. For this problem, the method was changed dramatically as the previous implementations would be too inefficient to find the optimal path through so many objectives.

For example, the previous implementations of A* would not have a predetermined first objective. However in this case, the first objective is determined as the closest one to make the search much more efficient. Along with that, instead of calculating a collection of manhattan distances and comparing them, this implementation uses just the manhattan distance of the closest neighbor to significantly reduce the computation and amount of states to visit.

```
start = maze.getStart()
objectives = maze.getObjectives()
objectives_visited = []  # To keep track of visited objectives
path = []  # To store the final path
while objectives:
    nearest_goal = objectives[0]  # Pre-determine the starting objective for
efficiency
    min_distance = manhattan_distance(start, objectives[0])  # Initial
Manhattan distance
```

As a result of these decisions, the solution is not optimal, and will not provide the actual shortest path through the objectives, but it is very close for how fast it works.

It uses the same logic as the previous two A* implementations, but one of the newer things is the “path_track”. As a new state is added to the astar_list that is being iterated over, it is also added to this tree, which allows it to remember the path taken. When the search reaches its goal, it knows how it got there because it can go through the track from the goal state to the start state.

Results

Part 1: Results for bigMaze

	BFS	DFS	UCS	A* (astar)
Path Length	210	210	210	210
States Explored	619	426	619	549
Total Time (s)	0.002711058	0.001968861	0.006149769	0.002539157

Part 2: Results for bigCorner

	A* (astar_corner)	A* (astar_multi)
Path Length	162	208
States Explored	7983	655
Total Time (s)	0.15518522262573242	0.0015799999237060547

Part 3: Results for bigSearch

	A* (astar_multi)
Path Length	615
States Explored	588
Total Time (s)	0.005825996398925781

Part 3: Results for tinySearch

	A* (astar_multi)	A* (astar_corner)
Path Length	42	27
States Explored	33	5072
Total Time (s)	0.00012803077697753906	0.13444960117340088

Analysis

Part 1

I chose a simple admissible heuristic (manhattan distance) to reduce the computation time and cause A* performing worse than it should. However, for such a simple problem, this added computation time meant that A* did not perform significantly better than BFS or UCS.

The surprise was that DFS, which is the only non-complete search algorithm tested, performed better than the others and still provided the true optimal solution. This is likely because of the way this chart is laid out. DFS searches deep, and the solution was deep in the corner, making it a favorable layout for DFS.

However, it is clear after testing on multiple maze confirmations that A* always found the optimal solution in less time and fewer explored states than the other search methods.

Part 2

The corners search algorithm will provide the true shortest path, but that means it is very slow. For example, it struggles to finish the openCorner map, and cannot finish the openSearch from Part 3. Compared against astar_multi from Part 3, it is about 100x slower and visits 12x more states.

This method is not efficient. However, it finds a shorter path than astar_multi, and proves that it consistently finds the shortest path over the much faster method. At the same time, it will always go through more states than the faster method which still delivers reasonable results.

Part 3

For this method, I kept my heuristic simple because I was worried about performance more than everything else. For example, I continued using manhattan distance, which had low computation time over other admissible heuristics I could have used which may have been better at estimating the true cost of the path.

Along with that, I chose to predetermine the starting state. This helped a lot with computation time, as shown in the results, where astar_corners visits 150x more states and takes 1000x longer to solve the same tinySearch problem. This means that the approach is very efficient and still provides reasonable results. However, in the same table, it shows that although it is so much faster it provides an answer that is 55% higher than the actual optimal path. This means it provides the wrong answer at the expense of speed.

Conclusion

It is clear from these results that A* is a very versatile search algorithm, and that a search algorithm doesn't need to provide the exact most optimal results when the state space is too large. It is better to be efficient than 100% accurate for most cases, as most applications, especially robotics, can deal with a small percentage of error for faster results.