Final Project Assignment
Evaluation of Route Logistics in Search Problems

Curso Artificial Intelligence
Tanner O'Rourke
Spring 2019

Introduction

In this project, I worked with different search algorithms in a simulated graphical environment that had a delivery boy delivering pizzas from a restaurant to customers on a grid. Starting his route from his base, he had to do this in an optimal by utilizing Depth First, Breadth First, Limited Depth First Search, and A* Heuristic Search algorithms to complete his task as fast as possible. The problem is, however, he works in many different cities, that all have different characteristics to their layout. I first implemented a Manhattan Distance heuristic, pertinent actions, results from those actions, and cost functions. After this, our aim in this evaluation was to propose different situations where some of these algorithms may work better than others. After analyzing maps such as checkerboards, mazes and basic routes, we discovered how some algorithms are best suited for other cases.

Part 1

Explanation of Problem Representation

For part 1 (and part 2), I defined the state of the pizza boy to hold:

- x-position on map
- y-position on map
- # of pizzas in bag
- # of pizzas delivered

I represented this with a 4-variable tuple as *state = (x-cord, y-cord, # pizzas in bag, # pizzas delivered).*
Permittable actions must be one's that alter the state, therefore the actions allowed were:

- 'Load' a pizza into the bag
- 'Unload' a pizza from the bag
- And move in any x or y coordinate direction (North, South, East, West)

The result of these actions would either change the x-coordinate, y-coordinate position, # of pizzas in the bag, or # of pizzas delivered.

Explanation of Implemented Functions

Take note that:
State variable *s* = *(x-cord, y-cord, # pizzas in bag, # pizzas delivered)* - tuple of current state
References to *self.[var]* are global gameProblem class variables initialized at runtime.

• *actions(s):*

This function receives a state *s* and returns a list of applicable actions in *s*. As described above, we can 'Load', 'Unload' or move 'North', 'East', West', or 'South'. However, at certain coordinates some of these aren't allowed. In my implementation, I initialize only the four movement directions into a list. I then use a list of *if* statements to evaluate the following, then return.

> • If our state is at a restaurant or customer, we ADD 'Load' or 'Unload' respectively to actions (Keeping in mind that we must have room to load pizzas, or pizzas to deliver).
> • If moving in a direction will either a) move the delivery boy off the map, otherwise he's at the edge, or b) move the delivery boy into a blocked tile such as a building, we REMOVE that direction from actions.

• *result(s1,a):*

This function receives a state *s1* and action *a* and returns a state *s2*, which is the result of applying *a* to *s1*. Like the *actions(s)* function, we must account for all 6 actions, 'Load', 'Unload', and move 'North', 'East', West', and 'South'. To do this I copy the *state* into a *nextstate* variable, then apply the below actions conditionally with *if* statements to update *nextstate*, then we return *nextstate*.

> • IF we 'Load' a pizza, we add 1 to the # of pizzas in our bag in the state variable.
> • IF we 'Unload' a pizza, we both subtract 1 pizza from the # in the bag and add 1 pizza to the # delivered in the state variable.
> • Moving in a specific direction will add or subtract 1 from either our x-coordinate or y-coordinate state variable. For example, if the action is 'North', we subtract 1 from the y-coordinate variable.

• *is goal(s):*

returns *True* if the given state *s* equals the *self.GOAL* variable initialized in setup, otherwise it returns *False.*

• *cost(s1,a,s2):*

As defined in the writeup, for part 1 all actions have unit cost. Therefore, this function can return 1.

• *heuristic(s):*

This function receives a state *s* and returns the heuristic value of s; that is, the estimated cost of reaching the goal state from s. For the A* heuristic function, I implemented the Manhattan Distance, because this heuristic works well when using a x-y coordinate grid to finding distances to the goal (just like the city of Manhattan!). In part 1, because the 1 client can make only 2 orders at once and the delivery boy can hold at most 2, only one restaurant trip has to be made. This means the delivery boy can simply find the shortest path to the customer through the (one) restaurant, and then the shortest path back home. Developed in the code, this yields one of 3 occurrences when calculating the heuristic value of *s*.

- *Delivery boy is searching for pizzas*
    - Heuristic evaluates the minimum Manhattan Distance from delivery boy to *closest* pizza on the map (goal)
- *Pizzas in bag, delivering*
    - Heuristic evaluates the Manhattan Distance from the delivery boy to customer (goal)
- *Pizzas delivered, returning home*
    - Heuristic evaluates the Manhattan Distance home

• *setup():*

The *setup* function configures our initial state, goal state, and the search algorithm to use. To use each search algorithm, you comment out all the *algorithm* vars except for the one you want to use. The *self.GOAL* class variable will always be the same as the *self.INITIAL_STATE* class variable, except the # delivered are respectively the # we want to deliver and 0 to initialize. The amount delivered in the goal state must be changed to run different amounts of deliveries in tests.

I also use the pre-initialized *self.POSITIONS*, *self.AGENT_START* and *self.CONFIG* to update other global class variables such as *self.CUSTOMERS* and *self.MAXBAGS* so that we can use them throughout other methods and so that they can be update only in the config file.

• *printState (s):*

This function prints the 4 variables in the state tuple *s* next to their definitions.

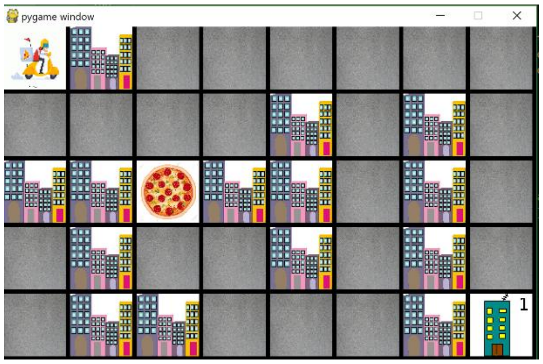
• *getPendingRequests (s):*

This function returns, if the state *s* is at a delivery location, the difference in the # of deliveries to be made in the *self.GOA*L class variable (initialized in setup) minus the # of deliveries made in the state class variable *s*. Otherwise, it returns None.

Experimental Evaluation and Comparison

The three scenarios I developed were a Medium sized map with random buildings, a large maze, and a small map with obvious paths. I thought it best to change multiple things between scenarios, because this specific problem representation is fairly straightforward (find the pizza, load, deliver, return). For each problem there is no terrain, only buildings, restaurant and customer, however I moved the pizza to different places. Each Scenario shows the total solution length, solution cost, visited nodes, iterations, and max fringe size, and an algorithmic comparison. I compare the scenarios and their results against each other after.

Scenario 1 • Small sized grid with obvious path



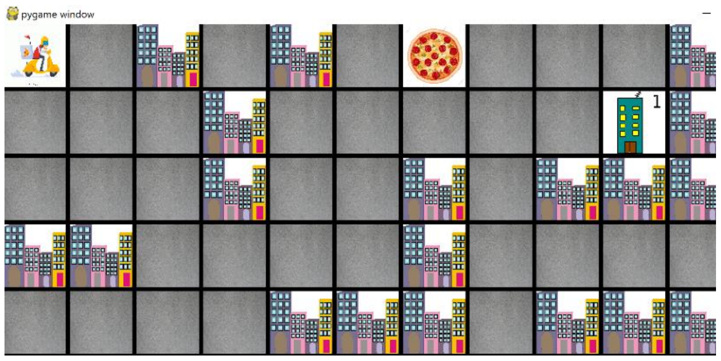
Breadth First Search
# of expanded nodes = 121
Solution Cost = 32
Memory Usage = 10

Limited Depth First Search
# of expanded nodes = 704
Solution Cost = 42
Memory Usage = 45

A* Search (an example of a call to any of these algorithms is included in the given code).
# of expanded nodes = 129
Solution Cost = 32
Memory Usage = 10

For this scenario I started the delivery boy in the top left, customer in the bottom right, and the pizza close by. The obvious path here would be to grab the pizza, then go back up, over and down. Limited Depth Search worked by far the worst (Depth First Search didn't work at all due to memory errors). A* and Breadth First Search were both able to find *the* optimal solution of 32 while visiting roughly the same number of nodes.

Scenario 2 • Maze-based grid



Breadth First Search
# of expanded nodes = 346
Solution Cost = 34
Memory Usage = 24

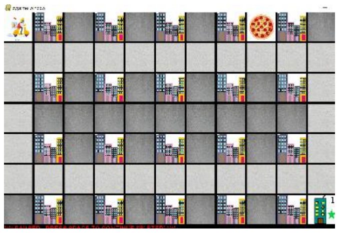
Depth First Search
# of expanded nodes = 84
Solution Cost = 36
Memory Usage = 24

Limited Depth First Search
# of expanded nodes = 84
Solution Cost = 36
Memory Usage = 24

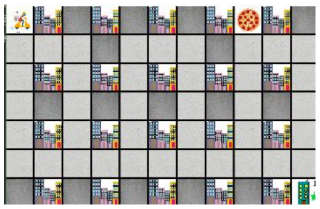
A* Search (an example of a call to any of these algorithms is included in the given code).
# of expanded nodes = 210
Solution Cost = 34
Memory Usage = 28

This scenario was extremely difficult to get for some algorithms to run without crashing (more in personal comments). However, all algorithms had roughly about the same solution cost and memory usage. However, A* Search and Breadth First Search had many expanded nodes. This is most likely because there are many different similar paths that have about the same cost (look at figure above on first route to the restaurant). This causes a lot of nodes to be checked, in contrast to Depth First Search, which can find an optimal solution on one of its first iterations.

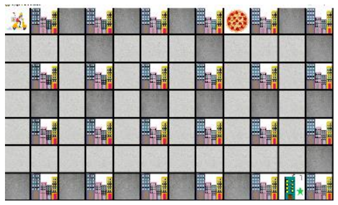
Scenario 3 • Checkerboard Grid



Breadth First Search
# of expanded nodes = 456
Solution Cost = 38
Memory Usage = 28

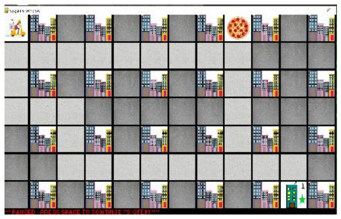


Depth First Search
# of expanded nodes = 112
Solution Cost = 66
Memory Usage = 31



Limited Depth First Search
# of expanded nodes = 112
Solution Cost = 66
Memory Usage = 31

A* Search (an example of a call to any of these algorithms is included in the given code).
# of expanded nodes = 305
Solution Cost = 38
Memory Usage = 27



This, I thought was the most interesting scenario. Both of the Depth First Search algorithms expanded very few nodes in comparison to A* Search and Breadth First Search. However, there solution cost was almost twice as much and used slightly more memory. Why? As you can see in the traversals above, the depth first algorithms snaked through the middle, while the other two stayed on one path. I believe this is because it is much more optimal in this case to stay on one tree of nodes, which Breadth First Search does primarily in comparison to Depth First Search algorithms and the A* Heuristic function.

Conclusions

The first conclusion I came to was that Breadth First Search and A* Heuristic algorithms are better ‘route’ algorithms when there are *a lot* of possibilities, and Depth First Search are better when there are *fewer* possible routes. As you can see from Scenario 1 when the route was very straightforward, both Depth First Search algorithms had a lower solution cost and number of expanded nodes. however, in the checkerboard grid, these two algorithms were *much* slower. This is inherently from how Depth and Breadth First Search algorithms work. DFS will, given some time, find the optimal route in a checkerboard style grid quickly by ‘snaking’ down specific paths, however, will take a long time. However, BFS will expand on many routes to find many sub-optimal solutions.

BFS and A* seemed to work almost identically in these problems, as well as the limited DFS and DFS. In the real world, this poses the question of whether you want an algorithm that can quickly find sub-optimal solutions or take a while to find an optimal solution. Both of these groups of algorithms work in different ways to accomplish either one of these tasks in different situations.

Personal comments

I had a few large overarching problems with this project. Firstly, I found that although we had to implement printState, I never had to use it and debugging elsewhere was easier. Secondly, after I ran the checkerboard grid, limited depth first search stopped working. After multiple hours of looking through code and consulting other teammates, I found out a fix by adding a parameter to the result in the gameSearch.py file. Lastly, I had a lot of problems getting Breadth First Search to work. After I restarted my computer, it would allow me to run a larger grid only a few times, before it started crashing. I think this is due to memory overload from having so many node trees being searched at once.