CS 3600: Pacman Graph Search Project Tips

Osama Sakhi
Siyan Li
Satwik Mekala

Georgia Institute of Technology

Contents

1 Notation:

Here we'll define the terms used within the project and how they may relate to concepts from the course.

1.1 `state`

In this project, *state* is simply a `tuple` of information encoding everything needed to solve the problem at hand. For the first 4 questions, the *state* is simply of the form (x, y), Euclidean coordinates on the grid. Later on in Q5 you'll be refining this notion of state.

1.2 `successor`

A *successor* is simply the *child node* of some *parent node*. To reach a successor, you take an *action* from some *parent node*, which may have some *cost* (depending on the problem you are solving). For this project, the successors of a state can be obtained using the `getSuccessors()` function. Each successor contains three elements: (1) the current state, being either a position or the name of the node, (2) the action taken from the previous state to reach the successor, (3) the cost of this action, if applicable.

1.3 `path`

A path is simply a sequence of actions taken from some start state S to the goal state G.

1.4 `goal`

The *goal*, G, is a state that you want to reach. Your graph search functions will all lead you to reaching this *state*. Remember: once you reach this state, you have no reason to continue searching for successors.

2 Generic Graph Search Algorithm

2.1 Psuedocode

Algorithm 1 Generic Graph Search Algorithm

Input: *startState*, the state at which we begin our search.
Output: *path*, the sequence of actions to get from *startState* to a `goal state`

1: **// Initialize variables**
2: $openList \Leftarrow dataStructure()$ ▷ dataStructure ∈ {`Stack, Queue, PriorityQueue`}
3: $closedList \Leftarrow set()$ ▷ Only explored nodes will go in here
4: $metadata \Leftarrow createMetadata(startState, nil)$ ▷ Track actions/costs. See Section 2.3.
5: $current \Leftarrow (startState, metadata)$ ▷ Define *current* as the state plus metadata
6:
7: **// Search until goal state found.**
8: **while** *current.state* is not a `goal state` **do** ▷ Continue until goal reached
9: $p \Leftarrow current.state$ ▷ Define p as the state we're interested in, the *parent*
10: **if** p is not in `closedList` **then** ▷ If p is unexplored, explore it
11: `closedList`.$add(p)$ ▷ Mark p as explored.
12: **for** $s \in successors(p)$ **do** ▷ Retrieve all successors for the *parent*
13: $s.metadata \Leftarrow createMetadata(s, p)$ ▷ Create metadata using both p and s
14: $openList.add((s.state, s.metadata))$ ▷ Add the successor onto the open list
15: $current \Leftarrow openList.next()$ ▷ Fetch a new *current* from the `openList`
16:
17: **// Reconstruct** *path* **from** *startState* **to goal state**
18: $path \Leftarrow reconstructPath()$ ▷ Reconstruct path to goal. See Section 2.4.
19: **return** *path*

2.2 Data Structures

As we can see above in Line 2 of Algorithm 1, we have a choice of data structures to use. The choice of data structure will determine which algorithm we end up implementing:

- `Stack` → Depth-First Search**(DFS)**
- `Queue` → Breadth-First Search**(BFS)**
- `Priority Queue` → Uniform Cost Search**(UCS)** and A^*

2.3 Metadata

Beyond simply choosing the data structure, we may also need to determine slightly different `metadata` to implement each algorithm. The metadata we'll pass around will accomplish 2 things:

1. Help keep track of the path taken to reach the `goal`

2. Allow the priority of a state to factor in properly for UCS and A^*

The `metadata` is problem-dependent, so we won't go into the specifics just yet. The most important thing to know is how to pass the metadata along with the state. In the case of Python, this ends up being pretty easy, as we can put everything we want to know about a state into a `tuple`, and store that tuple in the `openList`. We can also assign it priority, in the case of Priority Queues.

2.4 Reconstructing Paths

Like we mentioned in Section 2.3, you'll need to decide what to put in metadata. This will also be dependent on how you go about keeping track of the path taken to reach the `goal state`. There are 2 general approaches we suggest you explore. Both get you full credit, but it's up to you to decide which one you want to try implementing:

- **Parent Hashmap:** With this approach, we need to always be aware of what action was taken to reach the successor, and we need to add this into the metadata we store in Line 14. When we begin exploring a state (i.e. Line 12), we need to mark in a hash map (`dict` in Python) the action taken to reach this *state*.

  Then, once we reach the `goal state`, we need some way to use the hash map to back-track our way through the actions, taken from each parent until we reach the start node, *startState*.

- **Full-Length Sequence of Actions:** Like the *Parent Hashmap* approach, we need to pay attention to the action taken to reach a successor (e.g Line 12). Unlike that approach, however, we can be very memory inefficient, and simply keep a full-length list of actions taken to reach whichever state we are currently evaluating. So for each successor, the *metadata* will contain an entire sequence of actions to reach the *parent*, and a new action will be added to the end of that sequence to signify the *action* that takes you from p to s (see Line 13).

  Once you complete the graph search, you'll already have the full sequence of actions to reach the goal G, in your *current.metadata*.

In both cases above, we've omitted some implementation details for you to determine as you complete the project.

2.5 Determining Priority for A^* and `UCS`

Now, let's think about how we can use what we know from Sections 2.2 and 2.3 to help us out.

We know that both algorithms have some notion of priority. This *priority* value will need to be passed around as `metadata` for you to access as you pop items off of your data structure. We'll leave it up to you to determine how to implement that part. Let's shift our focus to the differences between A^* and `UCS`'s priority functions.

Now, let's define 2 functions, g, and h. Both take as input a state s, and both return a real number.

- $g(s)$ tells us the cost of reaching the state s, meaning of the ways that we've found to get to s so far, the cheapest of them is of cost $g(s)$
- $h(s)$ *approximates* the remaining cost of reaching a `goal` state, starting *from* the current state s.

Now that we understand these two functions, let's look at what the priority values for each of the two algorithms should be:

- `UCS:` $priority(s) = g(s)$
  In other words, **ONLY** the true cost of reaching state s is factored into the priority.
- `A*:` $priority(s) = g(s) + h(s)$
  In other words, both the true cost of reaching s and the approximate cost of reaching the `goal` from s are factored into the priority.

  IMPORTANT: Although priority of a node for A^* being $priority(s) = g(s) + h(s)$, this does NOT mean that this priority value should be used by successors, s', to compute their own costs, $g(s')$. The $g(s')$ values must be determined solely through the $g(s)$ values, with no contributions from $h(s)$.

3 Redefining State

For Q5, we'll be refining the notion of state used for the project. From Section 1.1, we know that for Q1-Q4, we had been using Euclidean coordinates alone to express state. Now, we need something more sophisticated to handle this problem.

You'll notice that in the instructions, we're asked to implement `CornersProblem`, which is a Python class that captures some of the utilities you've already used by this point: `getStartState()`, `getSuccessors()`, and `isGoalState()`.

Now, you're tasked with coming up with a representation that encodes when you're already been to a certain corner of the map. Think about it this way: what is bare minimum amount of information you need to know that you've reached the goal state for this type of problem? Probably that you've *marked* a corner as *seen* or *visited*, right? Think of ways you can do that using `tuples` in Python.

4 Heuristics

A heuristic is a *rule of thumb* or a *hint* that informs us how good the current state is. Typically, the states closer to the goal states are better than the states further away from the

goal state. In other words, a heuristic function gives an estimate to the goal state from the current state.

Heuristics provide the search algorithm with an intuition of how to focus the search to a specific part of the state space. Search algorithms that utilize heuristics are called *informed search* algorithms. The informed search algorithm we implement for this project, A*, is designed to beat `UCS`, but it needs a good enough heuristic to do so. We will explain two key characteristics good heuristic in the following sections.

4.1 Admissibility

A heuristic is admissible if and only if it **does not overestimate** the true cost of reaching the goal. Formally, we can define it as the following:

- Let $h(s)$ be the *guess* of the true cost of reaching the `goal` from state s
- Let $h^*(s)$ be the true cost of reaching the `goal` from state s

Our heuristic, h, is admissible if and only if:

$$h(s) \leq h^*(s)$$

for every possible state s.

4.1.1 Why is Overestimating Bad?

Suppose you're in a group project, and you have a couple different roles to choose from, with each role having its own set of tasks. These tasks are successive, meaning the first task needs to be completed before the second can be started. Being the busy tech student you are, you decide you want to go with the role that has the least amount of overall work for you to do. Unfortunately, you don't actually know how long each task takes, so for any task t, you only have *some idea* of how long that task and how long all remaining tasks for that role will take. This means you have a heuristic $h(t)$ that tells you how much work is left for you to do for the role, if you're about to work on task t.

Now, let's say you decide to just go with your gut and pick the role that has a first task with the lowest $h(t)$, because that would mean you have the least work to do.

What happens if your hunch is totally wrong? If you estimate that you'll need 2 days for the first task t_0 for some role, when in reality it takes 2 hours, you're likely going to pick up a role that gives you more work in the end, just because you made a bad guess.

Alternatively, if your guesses for the first few tasks, $t_0, t_1, ...t_{M-1}$, for some role were correct, but your guess for just some intermediate task, t_M was a complete overestimate, your error is going to propagate backwards, so every ancestor of t_M will be affected. This means you'll have an incorrect $h(t_{M-1}), h(t_{M-2}), \ldots, h(t_2, h(t_1)$, even though your actual guess for those tasks in isolation was correct.

This is analogous to Pacman exploring the wrong part of the grid because his heuristic tells him that the other parts of the grid are much further from his goal.

4.2 Consistency

A heuristic, $h(s)$, is consistent if the heuristic value decreases from a parent state p to a child state c, where c is closer to the `goal` than p is, and that this decrease is less than or equal to the cost of going from p to c. Formally, we define it as:

$$h(p) - h(c) \leq cost(p, c)$$

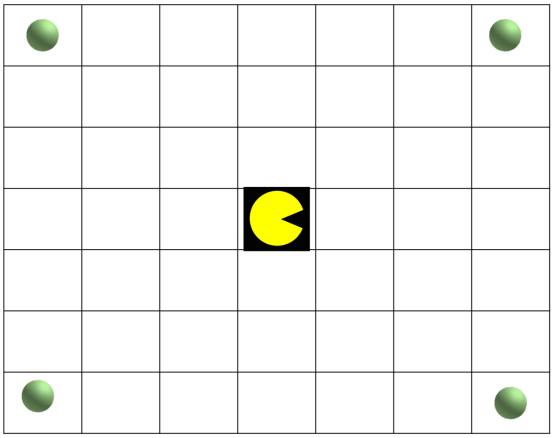
with the additional condition that the heuristic value is 0 once you've reached the `goal`, i.e:

$$h(goal) = 0$$

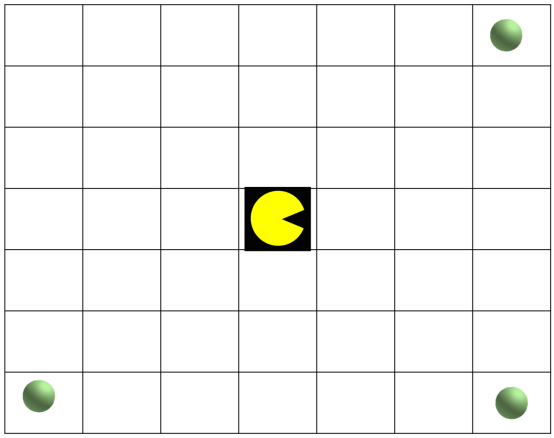
4.3 Corners Heuristic

Now, let's think about how to approach this problem: we're able to see Pacman at any point on the grid, and we can always determine the exact corner locations he hasn't gone to yet. How can we use this to make a good heuristic?

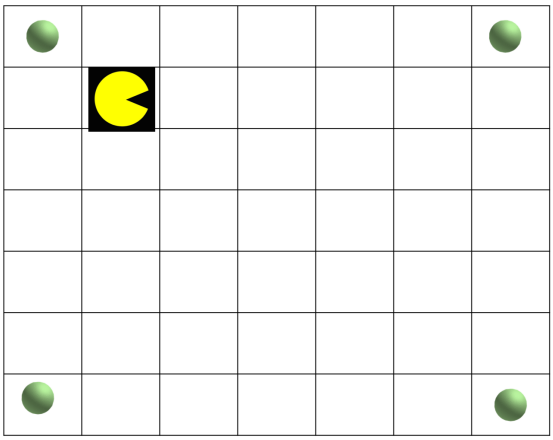
Let's start by thinking about some sstate Pacman could be in, and imagining what the heuristic values should look like for those states.



(a) Pacman in the middle of the grid. We'll call this state a.



(b) Pacman in the middle of the grid, but with one food pellet already eaten. We'll call this state b.



(c) Pacman is just 2 steps away from the food pellet. We'll call this state c.

(d) Pacman is has eaten the nearest food pellet, so all the remaining ones are distant from him. Let's call this state d.

Figure 1: Here we see Pacman in the a couple of different states. Let's analyze what our heuristic values should be.

Now let's think about what we're looking at. Let's start with Figures 1c and 1d. Suppose our heuristic was the Manhattan distance to the single pellet nearest to Pacman. We'd get a value of $h(c) = 2$ and $h(d) = 6$. This sounds wrong since we know that at state c, Pacman

has more work left than he does in state d, meaning we expect $h(d) < h(c)$.

Now, let's think about if we did the opposite, what if our heuristic gave us the distance to the furthest pellet from Pacman, would that help us? Looking at Figures 1a and 1c, we see that our heuristic values would then be $h(a) = 6$ and $h(c) = 10$. That also sounds off, since in state c we're just 2 steps away from the closest food pellet, whereas in state a we're 6 whole steps away.

By now, you're probably thinking there's no good way to make a heuristic for this problem without incorporating information about the remaining pellets, and you'd be totally right. So let's start thinking of ideas for a heuristic that incorporates all of the pellets.

Would averaging the the distances of the pellets get us further? Look at Figures 1a and 1b and try to justify your answer.

4.4 Food Heuristic

Once you've created the heuristic necessary for Q6, you'll probably be a bit stumped as to why it does not also solve Q7 directly. This is totally understandable, because at first glance, they seem like the same problem: Pacman has many pellets to reach, and they're scattered all over the map, right?

The reason why your solution to Q6 will most likely prove to be ineffective for Q7, is because Q6 has a very special problem structure: the food pellets are conveniently placed at the corners, meaning it's often the case that you're equidistant from many food pellets. This allows you to use a simple, greedy heuristic, to solve for the optimal paths.

Q7, on the other hand, is another beast entirely. The problem no longer has any special structure that yields it to being solvable exactly, even without walls factored in. In fact, to compute the true cost from Pacman to every other food pellet falls, you would need infinite computational resources, as the problem grows in complexity with each additional pellet. In other words, for n pellets, you would have a runtime of $O(n!)$ to determine the exact cost of the best path that goes through all of them.

All that to say, **computing a heuristic that captures the true cost is simply not possible** given the time constraints. So what can we do instead? Well, we can still find novel ways to factor in the many pairwise distances between pellets.