COMP3702/7702 Artificial Intelligence
Semester 2, 2020
Tutorial 3 - Sample Solutions

Exercises 3.1 and 3.2

Tactics for approaching Exercises 3.1 and 3.2

Skim Ex 3.1 and Ex 3.2, and try to map what is required for designing (modeling) agent and environment interaction:

- What is the agent?
- What is the environment?
- What is the agent's goal?
- What are its costs?

Recall by model, we mean the state and actions spaces and transition function (or world dynamics) as the "calculus" of autonomous decision making. Refer to Tutorial 1.

Hint: For both questions, structure your solution code into 3 modules, i.e. `experiment`, `agent` and `environment`. Follow a top-down approach.

Experiment module

Begin developing the experiment module by considering the following:

- Identify what kinds of information that we should log in order to diagnose problems, improve performance, and answer the questions.
- Based on this, design the logging mechanism.

0.1 Environment codes

This module will model the world. Try to reuse as much code as possible as you answer Ex 3.1 and then Ex 3.2. Most aspects of the worlds are identical, the main difference being the presence of hard obstacles in Ex 3.1 and costly state transitions in Ex 3.2. One module can be written to handle either case.

Within the module, think about how you want to structure the interaction with the agent, and use this to define different classes. Two reasonable classes might be a `GridWorld` class and a separate `GridWorldState` class. The `GridWorld` class should provide a parser for the concrete problem you are trying to solve, and provide a `step` function that encodes the dynamics of this environment and a function that returns a list of neighbouring states and the transition costs to move to them. You may also want to have a method to hold heuristics here, although that could also be placed inside your search agent module.

The `GridWorldState` class then provides a place to put information and methods relevant to each specific state, e.g. its parents in the search tree.

Search agent module

Recall the common ingredients of search agents

- data structure (container, e.g. a queue or stack)
- goal state checker
- a way to obtain the neighbors (successors) of the currently explored (expanded) state

You could either build separate code for each algorithm, or use inheritance to develop one module with an abstract search agent class that is reused for each algorithm. In the example code, the former approach is used, so that each algorithm is clearly defined in a separate file.

In Ex 3.1, re-use BFS and DFS from Tutorial 2 and build IDDFS on top of DFS. Some resources to assist you:

- https://eddmann.com/posts/using-iterative-deepening-depth-first-search-in-python/
- https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/
- Iterative Deepening Depth First Search, Douglas Fisher
  - https://www.youtube.com/watch?v=7QcoJjSVT38
  - https://www.youtube.com/watch?v=5MpT0EcOIyM

In Ex 3.2, first build UCS, re-using BFS from Tutorial 2 or Ex 3.1. Then extend UCS to build A*:

- think of some admissible heuristics

Example code

The following is merely sample code that prioritizes pedagogical benefits over code efficiency, advanced programming techniques, advanced language features, etc. (Please report any bugs to the teaching team, thank you.)

How to use

- Download this gist's zip here
- Extract everything insize that zip to `your_directory`
- Type `python3 your_directory/question_3_1.py`
- Type `python3 your_directory/question_3_2.py`
- Select any sub-question by modifying the main question files, namely question_3_1.py, and question_3_2.py

Each of the modules are:

- Experiment codes:
  - question_3_1.py

  - question_3_2.py
- Environment codes:
  - gridworld.py
- (Search) Agent codes:
  - breath_first_search.py
  - depth_first_search.py
  - iterative_deepening_search.py
  - astar_search.py

The module gridworld.py contains the following classes:

- Class `GridWorld` provides
  - `step(self, state, action)` encodes the dynamics of this environment.
    - This is used for, eg getting the neighbors of a state.
  - `get_neighborlist(self, state)` identifies the neighbors of any given `state`.
    - It "simulates" the interaction by calling the `step()` function.
  - `estimate_cost_to_go(self, state, heuristic_mode)` provides some heuristics for informed search algorithms.
    - Recall that a heuristic is essentially an estimate of the cost-to-go from a state.
    - Some may prefer putting this in the agent module.
  - `__init__(self, cfg)`, which is the class' constructor, handles
    - setup based on the configuration `cfg`
    - some sanity checks for a valid enviroment instance.
- Class `GridWorldState` provides
  - state equality check: `__eq__(self, other)`
  - state prioritization for, eg Priority Queue: `__lt__(self, other)`

Exercises 3.3

Reuse your 8-puzzle code from last week. Use the same tactics as above.

Possible heuristics to use include:

- Hamming distance (i.e. number of tiles that are different between the current state and the goal state)
- Manhattan distance of each tile to its goal position
- The number of inversions

All these heuristics would be admissible, but they vary in how efficiently we can find the solution.

See sample solution code on Blackboard.

Exercises 3.4

a) Admissibility

We know that h_1 is an admissible heuristic. That is to say, it does not overestimate the true cost to the goal. From the question, we know that $h_1(s)$ has values between 0.1 and 2.0.

$h_2(s) = h_1(s) + 5$: We can't guarantee that $h_2(s)$ is admissible because $h_2(s)$ is larger than $h_1(s)$. Because of this, we cannot say for sure whether or not $h_2(s)$ overestimates the true cost to the goal, and it is not admissible.

$h_3(s) = 2h_1$: This heuristic will always be larger than $h_1(s)$. Thus, for the same reason as $h_2(s)$, we cannot guarantee its admissibility.

$h_4(s) = \cos(h_1(s)\pi)$: We know that: $-1 \leq \cos(x) \leq 1$ and that $0.1 \leq h_1(s) \leq 2.0$. This means that there may be some state s, such that $\cos(h_1(s)\pi) > h_1(s)$. For example when $h_1(s) = 0.1$, $h_4(s) = \cos(0.1\pi) = 0.99998 > 0.1$. For this reason, we cannot guarantee the admissibility of $h_4(s)$.

$h_5(s) = h_1(s) * |\cos(h_1(s)\pi)|$: This heuristic is equivalent to: $h_1(s)|h_4(s)|$. Since $h_1(s)$ is being scaled by the absolute value of the output of $h_4(s)$ (i.e. a value between 0 and 1), we can guarantee that $h_5(s) \leq h_1(s)$. Since we know that $h_1(s)$ is admissible and thus never overestimates the true cost to the goal, we can guarantee the same for $h_5(s)$.

We can also compare these heuristics graphically. A plot of the 5 heuristics with respect to $h_1(s)$ is shown in Figure 1. We can see that only $h_5(s)$ is always $\leq h_1(s)$.

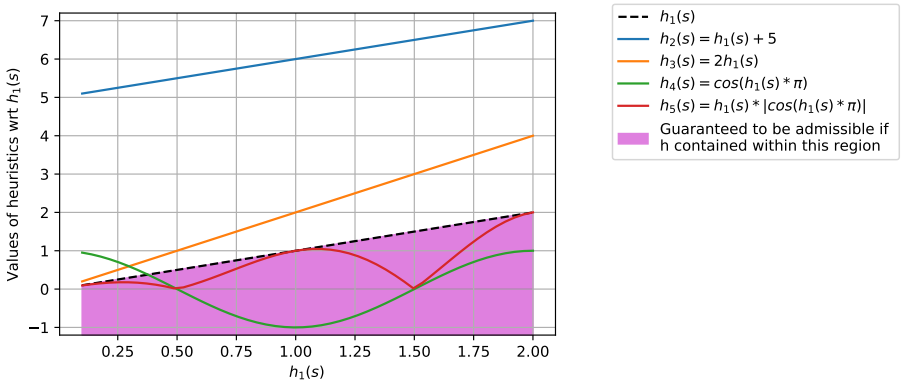

Figure 1: Comparing the 5 heuristics

b) Optimality

Despite $h_2(s)$ not being admissible, we can guarantee that it generates an optimal path.

When choosing between two possible nodes to expand in A*, the priority queue uses the quantity: $f(s) = g(s) + h(s)$. Whichever node has the lowest f value will be chosen next. Thus if $h_1(s)$ is an admissible heuristic (such that it leads to an optimal solution), replacing it with $h_1(s) + 5$ in this formula will never change the order of the queued nodes. This is because all the nodes get shifted by the same, constant amount, meaning it will still choose nodes in the same way. This only happens with 'constant' shifts.

A similar argument can not be used for $h_3(s)$. The shift here is not constant but is multiplicative. This can create cases where the h term in $f = g+h$ becomes too prominent and changes which nodes will be chosen. This is shown in the following example:

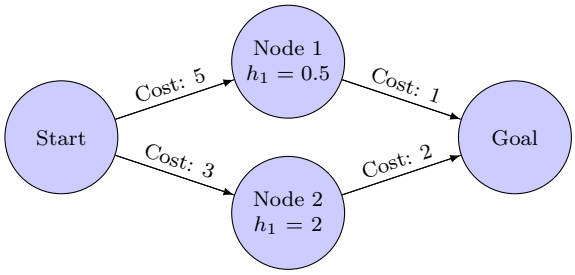

Figure 2: Example graph

This is a very simple graph where there are only two choices from the start node. Both Nodes 1 and 2 have been assigned an example h_1 value. In this case when the agent adds Nodes 1 and 2 to the priority queue, their scores will be:

$$\text{Node 1:} f(s_1) = 5 + 0.5 = 5.5$$
$$\text{Node 2:} f(s_2) = 3 + 2 = 5$$

And it will correctly choose Node 2 as the path to expand, followed by expanding the goal node. If the h values are all multiplied by 2 however (as in h_3), the result would be:

$$\text{Node 1: } f(s_1) = 5 + 1 = 6$$
$$\text{Node 2: } f(s_2) = 3 + 4 = 7$$

And Node 1 would be chosen incorrectly, followed by expanding the goal node on a sub-optimal path.

For $h_4(s) = \cos(h_1(s)\pi)$, the values can change fairly drastically, and this heuristic is no longer admissible. To check this, consider the following: $h_1(s_1) = 1/4$, $h_1(s_2) = 1/2$, $h_1(s_3) = 3/4$. In this case, $h_4(s_1) = \sqrt{2}/2$, $h_4(s_2) = 0$, $h_4(s_3) = -\sqrt{2}/2$.

This means that the order in which nodes are expanded from the frontier of the priority queue can differ between an A* search using $h_4(s)$ and $h_1(s)$. Because $h_4(s)$ is also a

non-admissible heuristic, we cannot guarantee that using $h_4(s)$ will generate an optimal path.

h_5 is always equal to or less than the admissible heuristic h_1 because it is basically just h_1 being scaled down by a number that is always positive and less than or equal to 1. This means it is also an admissible heuristic and will generate an optimal solution.