CSCE 580: Artificial Intelligence
Codinw HW 1: Uninformed and Informed Search
Due: 9/5/2022 at 11:59pm

In this homework, you will implement breadth-first search, iterative-deepening search, and a best-first search algorithm that encompasses uniform cost search, greedy best-first search, A* search, and weighted A* search.

Keep in mind that "frontier" is the same as "OPEN" and "reached" is the same as CLOSED.

When debugging, use this code to set a breakpoint.
```
import pdb
pdb.set_trace()
```

Your code must run in order to receive credit.

Do not change the signature of the functions. Your code must accept the exact arguments and return exactly what is specified in the documentation. You can use helper functions as long as they are contained in the same python file.

Installation

We will be using the same `conda` environment as in Homework 0.

The entire GitHub repository should be downloaded again as changes were made to other files. You can download it with the green "Code" button and click "Download ZIP".

Helper Classes and Functions

`Node`: the Node class used in the search algorithms

`get_next_state_and_transition_cost(env, state, action)`: gets the resulting state and transition cost of taking the given action in the given state.

`visualize_bfs(viz, closed_set, queue, wait)`: to visualize breadth-first search or best-first search while it is running (can be used for debugging).

`visualize_dfs(viz, popped_node, lifo)`: to visualize depth-first search while it is running (can be used for debugging). `popped_node` is the node that was just removed from the LIFO.

`env.is_terminal(state)`: returns True if the state is a goal state and False otherwise.

`env.get_actions(state)`: gets the actions (a list of integers) available in that state

1   Breadth-First Search (30 pts)

Implement the `breadth_first_search` function. This pseudo-code is shown in Algorithm 1.

To check your implementation, run:
`python run_assignment_1.py --map maps/map1.txt --method breadth_first`

Algorithm 1 Breadth-First Search

1: **procedure** BREADTH_FIRST_SEARCH(start_state)
2:     **if** start_state is a goal **then**
3:         return []
4:     **end if**
5:
6:     Initialize OPEN (should be a FIFO queue)
7:     Initialize CLOSED (should be a set of states)
8:     Create node root with state start_state
9:     push root to OPEN
10:    CLOSED.add(root.state)
11:    **while** OPEN is not Empty **do**
12:        pop node n from the front of OPEN
13:        **for** child in expand(n) **do**
14:            **if** child.state is a goal **then**
15:                return get_soln(child)
16:            **end if**
17:
18:            **if** child.state not in CLOSED **then**
19:                CLOSED.add(child.state)
20:                push child to OPEN
21:            **end if**
22:        **end for**
23:    **end while**
24:    return None
25: **end procedure**

---

2 Iterative Deepening Search (30 pts)

Implement `iterative_deepening_search` and `depth_limited_search`. The pseudocode for iterative deepening search is shown in Algorithm 2 and the pseudocode for depth-limited search is shown in Algorithm 3.

For depth-limited search, you will have to implement a function that checks for cycles. This function should return True if any of the ancestors of the given node have the same state and False otherwise. You can compare states with the `==` operator.

To check your implementation, run:
`python run_assignment_1.py --map maps/map1.txt --method itr_deep`

Algorithm 2 Iterative Deepening Search
___
1: **procedure** ITERATIVE_DEEPENING_SEARCH(start_state)
2:     soln = None
3:     limit = 0
4:     **while** soln is None **do**
5:         soln = depth_limited_search(start_state, limit)
6:         limit += 1
7:     **end while**
8:     return soln
9: **end procedure**
___

Algorithm 3 Depth-Limited Search
___
1: **procedure** DEPTH_LIMITED_SEARCH(start_state, limit)
2:     Initialize OPEN (should be a LIFO queue)
3:     Create node root with state start_state
4:     push root to OPEN
5:     **while** OPEN is not Empty **do**
6:         pop node n from the back of OPEN
7:         **if** n.state is a goal **then**
8:             return get_soln(n)
9:         **end if**
10:         **if** (n.depth < limit) and (not is_cycle(n)) **then**
11:             **for** child in expand(n) **do**
12:                 push child to OPEN
13:             **end for**
14:         **end if**
15:     **end while**
16:     return None
17: **end procedure**
___

3   Best-First Search (40 pts)

For this section, you will implement `best_first_search` which encompasses uniform cost search, greedy best-first search, A* search, and weighted A* search. You will find that only a small modification sets these algorithms apart. The pseudocode is shown in Algorithm 4.

At each iteration, a node is popped from OPEN that has the highest priority. The priority is determined by the cost f where the node with the lowest cost has the highest priority. The cost f is computed as $f(n) = \lambda_g g(n) + \lambda_h h(n)$. $g(n)$ is the cost of getting to the start node to node n, $h(n)$ is the heuristic, which is the estimated cost of getting from node n to the goal node and λ_g and λ_h are weights.

For the heuristic value $h(n)$ of a node n is the Manhattan distance to the goal. The Manhattan distance between an object located at (x_1, y_1) and an object located at (x_2, y_2) is:
$|x_1 - x_2| + |y_1 - y_2|$
Use `state.agent_idx` and `state.goal_idx` in your solution.

To implement a priority queue, see the Python documentation on `heappush` and `heappop` from `heapq`.

Uniform cost search is obtained by setting $\lambda_g = 1$ and $\lambda_h = 0$.
Greedy best-first search is obtained by setting $\lambda_g = 0$ and $\lambda_h = 1$.
A* search is obtained by setting $\lambda_g = 1$ and $\lambda_h = 1$.
Weighted A* search is obtained by varying either λ_g or λ_h.

To run best first search do:

`python run_assignment_1.py --map maps/map1.txt --method best_first`

To run each of the methods use add the following switches:

Uniform Cost Search

`--weight_g 1.0 --weight_h 0.0`

Greedy Best-First Search

`--weight_g 0.0 --weight_h 1.0`

A* Search Search

`--weight_g 1.0 --weight_h 1.0`

Weighted A* Search Search

`--weight_g 1.0 --weight_h 8.0`
`--weight_g 1.0 --weight_h 16.0`

where `--g_weight` is λ_g and `--h_weight` λ_h

Algorithm 4 Best-First Search

1: **procedure** BEST_FIRST_SEARCH(start_state,h,λ_g, λ_h)
2:     Initialize OPEN (should be a priority queue)
3:     Initialize CLOSED (should be a dictionary mapping states to path costs)
4:     Create node root with state start_state
5:     push root to OPEN with priority $\lambda_g*\text{root.path\_cost} + \lambda_h*\text{h(root)}$
6:     CLOSED[root.state] = root.path_cost
7:     **while** OPEN is not Empty **do**
8:         pop highest priority node n from OPEN
9:         **if** n is a goal node **then**
10:             return get_soln(n)
11:         **end if**
12:
13:         **for** child in expand(n) **do**
14:             **if** (child.state not in CLOSED) or (child.path_cost < CLOSED[child.state]) **then**
15:                 CLOSED[child.state] = child.path_cost
16:                 push child to OPEN with priority $\lambda_g*\text{child.path\_cost} + \lambda_h*\text{h(child)}$
17:             **end if**
18:         **end for**
19:     **end while**
20:     return None
21: **end procedure**

---

4 Extra Credit (10 pts)

Implement a best-first search heuristic for the 8-puzzle. For credit, with your heuristic, A* search should find a shortest path in 10 seconds or less. The length of a shortest path for the particular instance given is 22.

To access the tiles of a 8-puzzle state, use `state.tiles` to obtain the numpy array. You can reshape the array into the 3x3 representation using `state.tiles.reshape((3, 3))`.

To run A* search with the 8-puzzle do:
`python run_assignment_1.py --method best_first --weight_g 1.0 --weight_h 1.0 --env puzzle8`

You are welcome to look up ideas (not involving code) for heuristics, however, you **must** cite any ideas in your submission and you **CANNOT** copy any code whatsoever.

Your code must work for both the AI Farm and the 8-puzzle without the user having to modify it! Remember, only the heuristic function needs to change. You can automatically select a different heuristic function based on the type of environment using `type(state) is FarmState` and `type(state) is NPuzzleState`.

What to Turn In

Turn in your implementation of `coding_hw/coding_hw1.py` to Blackboard.