CSCE 580: Artificial Intelligence
Codinw HW 1: Uninformed and Informed Search
Due: 1/28/2022 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 function. Your code must accept the exact arguments and return exactly what is specified in the documentation.

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

```
procedure BREADTH_FIRST_SEARCH(start_state)
    if start_state is a goal then
        return []
    end if

    Initialize OPEN (should be a FIFO queue)
    Initialize CLOSED (should be a set of states)
    Create node root with state start_state
    push root to OPEN
    CLOSED.add(root.state)
    while OPEN is not Empty do
        pop node n from the front of OPEN
        for child in expand(n) do
            if child.state is a goal then
                return get_soln(child)
            end if

            if child.state not in CLOSED then
                CLOSED.add(child.state)
                push child to OPEN
            end if
        end for
    end while
    return None
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

```
procedure ITERATIVE_DEEPENING_SEARCH(start_state)
    soln = None
    limit = 0
    while soln is None do
        soln = depth_limited_search(start_state, limit)
        limit += 1
    end while
    return soln
end procedure
```

Algorithm 3 Depth-Limited Search

```
procedure DEPTH_LIMITED_SEARCH(start_state, limit)
    Initialize OPEN (should be a LIFO queue)
    Create node root with state start_state
    push root to OPEN
    while OPEN is not Empty do
        pop node n from the back of OPEN
        if n.state is a goal then
            return get_soln(n)
        end if
        if (n.depth < limit) and (not is_cycle(n)) then
            for child in expand(n) do
                push child to OPEN
            end for
        end if
    end while
    return None
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

---

procedure BEST_FIRST_SEARCH(start_state,h,λ_g, λ_h)
    Initialize OPEN (should be a priority queue)
    Initialize CLOSED (should be a dictionary mapping states to path costs)
    Create node root with state start_state
    push root to OPEN with priority λ_g∗root.path_cost + λ_h∗h(root)
    CLOSED[root.state] = root.path_cost
    while OPEN is not Empty **do**
        pop highest priority node n from OPEN
        if n is a goal node **then**
            return get_soln(n)
        end if

        for child in expand(n) **do**
            if (child.state not in CLOSED) or (child.path_cost < CLOSED[child.state]) **then**
                CLOSED[child.state] = child.path_cost
                push child to OPEN with priority λ_g∗child.path_cost + λ_h∗h(child)
            end if
        end for
    end while
    return None
end procedure

What to Turn In

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