CSCE 790-002: Deep Reinforcement Learning and Search
Homework 4
Due: 12/1/2020

Installation

We will be using the same `conda` environment as in Homework 3.

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".

A* Search

For this homework, you will implement A* search and, with only a one-line modification, you will also implement weighted A* search, greedy best-first search, and uniform cost search. The A* search algorithm is outlined in Algorithm 1.

For a visualization of a working implementation, see Blackboard under Course Content/Homework/Homework 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 estimated cost of getting from node n to the goal node and λ_g and λ_h are weights.

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.
Greedy best-first search is obtained by setting $\lambda_g = 0$ and $\lambda_h = 1$.
Uniform cost search is obtained by setting $\lambda_g = 1$ and $\lambda_h = 0$.

To run each of the methods use:

A* Search
```
python run_astar.py --map maps/map1.txt --g_weight 1.0 --h_weight 1.0
```

Weighted A* Search
```
python run_astar.py --map maps/map1.txt --g_weight 0.1 --h_weight 1.0
python run_astar.py --map maps/map1.txt --g_weight 0.5 --h_weight 1.0
```

Greedy Best-First Search
```
python run_astar.py --map maps/map1.txt --g_weight 0.0 --h_weight 1.0
```

Uniform Cost Search
```
python run_astar.py --map maps/map1.txt --g_weight 1.0 --h_weight 0.0
```

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

Algorithm 1 A* Search

procedure A* SEARCH($s_0, h_\theta, \lambda_g, \lambda_h$)
    Initialize OPEN, CLOSED ▷ priority queue and set of expanded nodes
    $n_g = None$ ▷ have not yet found the node associated with the goal state
    Create node n_0 with state s_0
    push n_0 to OPEN
    while n_g is None **do**
        if OPEN is Empty **then**
            return failure
        end if
        pop highest priority node n from OPEN and put it in CLOSED
        if n is a goal node **then**
            $n_g = n$
        end if
        $n_{children}$ = expand_node($n.state, a$)
        $n_{children}$ = remove_nodes_in_open_or_closed($n_{children}$)
        $costs$ = compute_costs($n_{children}, h_\theta, \lambda_g, \lambda_h$)
        push nodes $n\_children$ to OPEN
    end while
    return get_soln(n_g)
end procedure

The `Node` class has the following properties:
`state`: the state associated with the node
`path_cost`: the cost to go from the start state to the current state
`parent_action`: the action taken from the parent node to get to the current node
`parent`: the parent node

Expand Nodes (35 pts)

Implement the `expand_node` method. This method will take a parent node and return a list of its child nodes. These child nodes are the result of taking every possible action from the state associated with the parent node (`parent_node.state`).

Instantiating a node is done with `Node(state, path_cost, action, parent_node)`.

Use `self.env.get_actions()` to get all possible actions.

Use `self.get_next_state_and_transition_cost(node.state, action)` to get the resulting state and transition cost of taking the given action in the state associated with the given node. You can get the correct `path_cost` of the child node by adding the `path_cost` of the parent node to the transition cost.

Compute Costs (35 pts)

Implement the `compute_cost` function.
The `heuristic_fn` takes a list of states as an input and returns a `np.array` of the computed heuristics.
The `compute_cost` function should return the costs (a list of floats) where the cost of a node n is computed as $f(n) = \lambda_g g(n) + \lambda_h h(n)$

Get Solution (30 pts)

Implement the `get_soln` function.
This function should return the sequence of actions needed to get to the given node from the start node. The return type should be a list of integers.

Use `node.parent` and `node.parent_action`. Keep in mind that, for the start node, `node.parent is None` and `node.parent_action is None`.

What to Turn In

Turn in your implementation of `assignments_code/assignment4.py`.