CSCE 580: Artificial Intelligence
Project 1: Uninformed and Informed Search
Due: 1/28/2021 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. For each of the classes you are asked to implement, see the `__init__` function to see the objects you have available to you. Please do not change their names as they are necessary for the visualization.

Keep in mind that, in the instructions, "frontier" is the same as "OPEN" and "reached" is the same as CLOSED in the lecture slides.

For a visualization of working implementations, see Blackboard under ProjectExamples/Project1/.

When debugging, use this code to set a breakpoint. It will be your best friend.

```
import pdb
pdb.set_trace()
```

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".

Basic Search Functions

These function will called from all of your other implementations.

Expand (10 pts)

Implement the `expand_node` function. This function 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, depth)`.

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

Use `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.

Get Solution (10 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 root node, `node.parent is None` and `node.parent_action is None`.

Breadth-First Search (20 pts)

Implement the `step` function in the `BreadthFirstSearch` class. This is outlined in the red box in Figure 1. See the `__init__` function to see the class variables.

Use `env.is_terminal(state)` to get whether or not a state is a goal state.

To check your implementation, run:
`python run_proj1.py --map maps/map1.txt --method breadth_first`

Use the `--wait` argument to set how many seconds the display waits between iterations.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

Figure 1: Breadth-first search. Implement the red box.

Iterative Deepening Search

For iterative deepening search, you will have to implement a function that checks for cycles and then implement depth-limited search. Your implementation of depth-limited search will be called in iterative deepening search, which has already been implemented.

To check your implementation, run:
`python run_proj1.py --map maps/map1.txt --method itr_deep --wait 0.01`

Is Cycle (5 pts)

Implement the `is_cycle` function. 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. This function will be used in depth-limited search.

Depth-Limited Search (15 pts)

Implement the `step` function in the `DepthLimitedSearch` class. This is outlined in the red box in Figure 2. See the `__init__` function to see the class variables.

function DEPTH-LIMITED-SEARCH(*problem*, ℓ) **returns** a node or *failure* or *cutoff*
  frontier ← a LIFO queue (stack) with NODE(*problem*.INITIAL) as an element
  result ← *failure*
  while not IS-EMPTY(*frontier*) **do**
    node ← POP(*frontier*)
    if *problem*.IS-GOAL(*node*.STATE) **then return** *node*
    if DEPTH(*node*) $>$ ℓ **then** Typo in the book. $>$ should be changed to $\geq$
      result ← *cutoff* Just return None here
    else if not IS-CYCLE(*node*) **do**
      for each *child* **in** EXPAND(*problem*, *node*) **do**
        add *child* to *frontier*
  return *result*

Figure 2: Depth-limited search. Implement the red box.

Best-First Search

For this section, you will implement 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.

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.

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 each of the methods use:
Uniform Cost Search
```
python run_proj1.py --map maps/map1.txt --method best_first --weight_g 1.0 --weight_h 0.0
```

Greedy Best-First Search
```
python run_proj1.py --map maps/map1.txt --method best_first --weight_g 0.0 --weight_h 1.0
```

A* Search Search
```
python run_proj1.py --map maps/map1.txt --method best_first --weight_g 1.0 --weight_h 1.0
```

Weighted A* Search Search
```
python run_proj1.py --map maps/map1.txt --method best_first --weight_g 1.0 --weight_h 8.0
python run_proj1.py --map maps/map1.txt --method best_first --weight_g 1.0 --weight_h 16.0
```

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

Get Heuristic (5 pts)

Implement the `get_heuristic` function. This function returns the heuristic of a node n, $h(n)$ where $h(n)$ 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.

Get Cost (5 pts)

Implement the `get_cost` function. The function returns the cost of a node n, $f(n)$ where
$f(n) = \lambda_g g(n) + \lambda_h h(n)$

Best-First Search (30 pts)

Implement the `step` function in the `BestFirstSearch` class. This is outlined in the red box in Figure 3. See the `__init__` function to see the class variables. See the Python documentation on `heappush` and `heappop` from `heapq`.

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

Figure 3: Best-first search. Implement the red box.

What to Turn In

Turn in your implementation of `proj_code/proj.py` to Blackboard.