Problem Set 1: Search

The goal of this problem set is to implement and get familiar with search algorithms.

To run the autograder, type the following command in the terminal:

```
python autograder.py
```

If you wish to run a certain problem only (e.g. problem 1), type:

```
python autograder.py -q 1
```

where 1 is the number of the problem you wish to run.

Instructions

In the attached python files, you will find locations marked with:

```
#TODO: ADD YOUR CODE HERE
utils.NotImplemented()
```

Remove the `utils.NotImplemented()` call and write your solution to the problem. **DO NOT MODIFY ANY OTHER CODE**; The grading of the assignment will be automated and any code written outside the assigned locations will not be included during the grading process.

IMPORTANT: Starting from this problem set, you must document your code (explain the algorithm you are implementing in your own words within the code) to get the full grade. Undocumented code will be penalized.

For this assignment, you should submit the following files only:

- `search.py`
- `dungeon_heuristic.py`

Put the 2 files in a compressed zip file named `solution.zip` which you should submit to Blackboard.

Problem Definitions

There are two problems defined in this problem set:

1. **Graph Routing**: where the environment is a graph and the task is to travel through the nodes via edges to reach the goal node. The problem definition is implemented in `graph.py` and the problem instances are included in the `graphs` folder.

2. **Dungeon Scavenging**: where the environment is a 2D grid where the player `'@'` has to collect all the coins `'$'` and reach the exit `'E'`. The player cannot stand in a wall tile `'#'` so they have a find a path that consists of empty tiles `'.'`. The problem definition is implemented in `dungeon.py` and the problem instances are included in the `dungeons` folder.

You can play a graph routing or a dungeon scavenging game by running:

```
# For playing a dungeon (e.g. dungeon1.txt)
python play_dungeon.py dungeons\dungeon1.txt

# For playing a graph (e.g. graph1.json)
python play_graph.py graphs\graph1.json
```

You can also let an search agent play the game in your place (e.g. a Breadth First Search Agent) as follow:

```
python play_dungeon.py dungeons\dungeon1.txt -a bfs
python play_graph.py graphs\graph1.json -a bfs
```

The agent search options are:

- `bfs` for Breadth First Search
- `dfs` for Depth First Search
- `ucs` for Uniform Cost Search
- `astar` for A* Search
- `gbfs` for Greedy Best First Search

If you are running the dungeon game with an informed search algorithm, you can select the heuristic via the `-hf` option which can be:

- `zero` where `h(s) = 0`
- `weak` to use the `weak_heuristic` implemented in `dungeon_heuristic.py`.
- `strong` to use the `strong_heuristic` which you should implement in `dungeon_heuristic.py` for problem 6.

You can also use the `--checks` to enable checking for heuristic consistency.

To get detailed help messages, run `play_dungeon.py` and `play_graph.py` with the `-h` flag.

Important Notes

The autograder will track the calls to `problem.is_goal` to check the traversal order and compare with the expected output. Therefore, **ONLY CALL** `problem.is_goal` **when a node is retrieved from the frontier** in all algorithms. During expansion, make sure to loop over the actions in same order as returned by `problem.get_actions`. The expected results in the test cases cover two possibilities only: the actions are processed either from first to last or from last to first (to take into consideration whether depth first search is implemented via a stack or via recursion).

Problem 1: Breadth First Search

Inside `search.py`, modify the function `BreadthFirstSearch` to implement Breadth First Search (graph version). The return value is a list of actions that define the path from the initial state to a goal state. If no solution is found return `None`.

Problem 2: Depth First Search

Inside `search.py`, modify the function `DepthFirstSearch` to implement Depth First Search (graph version). The return value is a list of actions that define the path from the initial state to a goal state. If no solution is found return `None`.

Problem 3: Uniform Cost Search

Inside `search.py`, modify the function `UniformCostSearch` to implement Uniform Cost Search. The return value is a list of actions that define the path from the initial state to a goal state. If no solution is found return `None`. When there are multiple states at the queue's front with the same `g(n)`, pick the state that was enqueued first (first in first out).

Hint: Python builtin modules already contain algorithms for Priority Queues which include:

- queue.PriorityQueue
- heapq

Problem 4: A* Search

Inside `search.py`, modify the function `AStarSearch` to implement A* search. The return value is a list of actions that define the path from the initial state to a goal state. If no solution is found return `None`. When there are multiple states at the queue's front with the same `h(n)+g(n)`, pick the state that was enqueued first (first in first out).

Problem 5: Greedy Best First Search

Inside `search.py`, modify the function `BestFirstSearch` to implement Greedy Best First search. The return value is a list of actions that define the path from the initial state to a goal state. If no solution is found return `None`. When there are multiple states at the queue's front with the same `h(n)`, pick the state that was enqueued first (first in first out).

HINT: Greedy Best First Search is similar to A* search except that the priority of node expansion is determined by the heuristic `h(n)` alone.

Problem 6: Heuristic for the Dungeon Scavenging Game

The requirement for this problem is to design and implement a consistent heuristic function for the Dungeon Scavenging Game. The implementation should be written in `strong_heuristic` which is in the file `dungeon_heuristic.py`.

The grade will be decided based on how many nodes are expanded by the search algorithm. The less the expanded nodes, the higher the grade. However, make sure that the code is not slow. To be safe, try to write

the heuristic function such that the autograder takes much less than the assigned time limit.

In case your computer has a different speed compared to mine (which I will use for testing the submissions), you can use the following information as a reference: `On my computer, A* search with the weak heuristic takes ~1 seconds on dungeon3.txt with the heuristic consistency checks enabled`. You can measure the run time for this operation on computer by running:

```
python play_dungeon.py dungeons\dungeon3.txt -a astar -hf weak --checks
```

If your computer is too slow, you can increase the time limit in the testcases files in the folder `q6`.

Delivery

The delivery deadline is `November 15th 2021 23:59`. It should be delivered on **Blackboard**. This is an individual assignment. The delivered code should be solely written by the student who delivered it. Any evidence of plagiarism will lead to receiving **zero** points.