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. Imagine that you are in a discussion, and that your documentation are answers to all the questions that you could be asked about your implementation (e.g. why choose something as a data structure, what purpose does conditions on `if`, `while` and `for` blocks serve, etc.).

IMPORTANT: For this assignment, you can only use the **Built-in Python Modules**. Do not use external libraries such as `Numpy`, `Scipy`, etc. You can check if a module is builtin or not by looking up The Python Standard Library page.

IMPORTANT: You must fill the **`student_info.json`** file since it will be used to identify you as the owner of this work. The most important field is the **id** which will be used by the automatic grader to identify you.

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

- `student_info.json`
- `parking.py`
- `search.py`

- `dungeon_heuristic.py`

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

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: Problem Implementation

As a software engineer in the world's best car company, you and your colleagues take too much pride in your personal cars. Each employee gets a dedicated parking slot, where only their car can be parked. One day, a saboteur scrambles your cars around, and every car was left somewhere in the passage ways. All the employees left their offices, and are trying to return their car to its parking slot, but without planning, the process is taking forever. To save the company's precious time and money, you propose building an AI agent that would find the optimal sequence of actions to solve the problem. But, first you have to formulate and implement the problem, so you collect the following notes:

1. In each step, only one car can move, and it can only move to one of the four adjacent cells (Left, Right, Down, Up) if it is vacant (is not a wall and does not contain another car).
2. Each car has one (and only one) dedicated parking slot.
3. Each action costs `1`. However, your colleagues hate it when another car passes over their parking slot. So any action that moves a car into another car's parking slot, the action cost goes up by `100` to become `101` instead of `1`.
4. The goal is to have each car into its own parking slot.

The following map shows an example of a parking slot:

```
#########
#1A...B0#
####.####
#########
```

There are two cars (`A` and `B`) where the parking slot of car `A` is denoted by `0`, and the parking slot of car `B` is denoted by `1`. In general, each car will denoted by an letter, and each parking slot will be denoted by the index of its car's letter in the english alphabet (`A -> 0`, `B -> 1`, `C -> 2` and so on). Walls are denoted by `#`. One solution to this map is to move car `A` using the sequence `[RIGHT, RIGHT, DOWN]` so that it gets out of car `B`'s way, then car `B` would move `LEFT` 5 times till it reaches its parking slot `1`, and finally, car `A` would move `[UP, RIGHT, RIGHT, RIGHT]` till it reaches its parking slot `0`. The solution would require `12` actions where each action costs `1` (no action causes a car to enter another car's parking slot). If car `A` had moved into the parking slot `1` (by going left), such an action would cost `101` (`1` for the move and `100` for angering the owner of car `B`).

Inside `parking.py`, complete the class `ParkingProblem` so that it works as described above and so that it passes its testcases.

Problem 2: 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 3: 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 4: 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 5: 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 6: 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 7: 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.

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, running speed_test.py takes ~17 seconds`. You can measure the run time for this operation on computer by running:

```
python speed_test.py
```

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

Delivery

The delivery deadline is `November 7th 2022 23:59`. It should be delivered on **Google Classroom**. 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.