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.

You can also specify a single testcase only (e.g. testcases01.json in problem 1) by typing:

```
python autograder.py -q 1/test1.json
```

To debug your code through the autograder, you should disable the timeout functionality. This can be done via the `debug` flag as follow:

```
python autograder.py -d -q 1/test1.json
```

Or you could set a time scale to increase or decrease your time limit. For example, to half your time limits, type:

```
python autograder.py -t 0.5 -q 1/test1.json
```

Note: You machine may be faster or slower than the grading device. To automatically detect your machine's speed, the autograder will run `speed_test.py` to measure your machine relative speed, then it will scale the time limits automatically. The speed test result is automatically stored in `time_config.json` to avoid running the speed test every time you run the autograder. If you want to re-calculate your machine's speed, you can do so by either running `speed_test.py`, or deleting `time_config.json` followed by running the autograder.

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.

Problem Definitions

There are three problems defined in this problem set:

1. **Parking**: where you have to route the cars to their parking slots. The details are explained in problem 1.
2. **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.
3. **Sokoban**: where the environment is a 2D grid where the player `'@'` has to push the crates `'$'` so that each crate is on a goal tile `'.'`. The player cannot stand in a wall tile `'#'` or with a crate `'$'`, but they can stand on empty tiles `'.'` or goal tiles `'.'`. If the player stands on a goal tile, it will look like this `'+'`. If a crate is on a goal tile, it will look like this `'*'`. Only one crate can exist on a single tile, and the player cannot more than one crate. In addition, crates cannot be pushed into walls. The problem definition is implemented in `sokoban.py` and the problem instances are included in the `levels` folder.

You can play a graph routing or a Sokoban game by running:

```
# For playing a sokoban (e.g. level1.txt)
python play_sokoban.py levels\level1.txt

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

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

```
python play_sokoban.py levels\level1.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 Sokoban 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 `sokoban_heuristic.py`.
- `strong` to use the `strong_heuristic` which you should implement in `sokoban_heuristic.py` for problem 6.

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

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

Important Notes

The autograder will track the calls to `problem.get_actions` to check the traversal order and compare with the expected output. Therefore, **ONLY CALL** `problem.get_actions` **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 `Depth First Search` 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. The action cost depends on the rank of the employee whose car is moved by the action. The rank is ordered alphabetically where `A` is the top ranking manager, and `Z` is the least ranking employee. The action cost increases linearly with rank where moving `A` will cost `26`, moving `B` will cost `25`, and so on till you reach `Z` whose action costs `1`. In addition, your colleagues hate it when another car passes over their parking slot. So if any action moves a car into another employee's parking slot, the action cost

goes up by 100. So if A moves into another employee's slot, the action cost is 26 (the cost of moving A) + 100 (the cost annoying the other employee) = 126.
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 Sokoban

The requirement for this problem is to design and implement a consistent heuristic function for Sokoban. The implementation should be written in `strong_heuristic` which is in the file `sokoban_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.

Delivery

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. You also must compress the solved python files and the **`student_info.json`** file together in a **zip** archive so that the autograder can associate your solution files with the correct **`student_info.json`** file. The failure to abide with the these requirements will lead to a zero since your submission will not be graded.

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

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

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.