Assignment 1 – Problem solving and search

Preliminaries

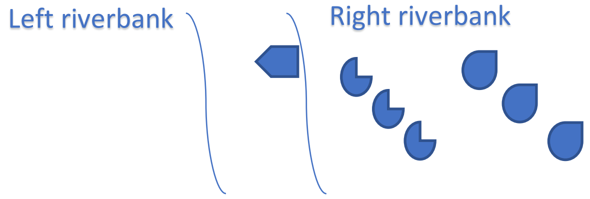
The programming language for this assignment is Python 3. The assignment should be submitted through ilearn2 no later than the 12th of September at 23.59 (CET).

Uninformed search

In the file "PFAI_Assignment_1.zip" found in the ilearn2 course page, there are three files: 'run_assignment_1.py', 'missionaries_and_cannibals.py', and 'node_and_search.py'. Where the first file contains code to set up a search problem, in this case the *missionaries and cannibal problem (m&c)*, and then code that calls the search algorithm with the defined problem. Here the initial state is the following:

`[[0, 0], 'r', [3, 3]]` and the goal state is: `[[3, 3], 'l', [0, 0]]`

The missionaries and cannibal problem are defined as follows, three cannibals and three missionaries want to cross a river, at their disposal they have one boat. The boat carries at most 2 persons (be it missionaries or cannibals), but can at minimum be operated but only one person. One crucial constraint is that *no missionary is allowed to be **outnumbered** by cannibals on either riverbank.*



In the figure above the initial setting of the problem is described, where the different shapes decipher missionaries and cannibals on the right river bank, and a boat in the river. The goal state is when all person has crossed the river and the boat in on the left river bank, the state is hence defined by a list of three parts: Left riverbank state, boat position and Right riverbank state, e.g.,

`[[#miss, #cann], boat pos, [#miss, #cann]]`

The problem is not fully defined and **your first assignment is to complete the description of movement**, in the class constructor for the m&c-problem all possible actions are defined, so you can easily figure out what actions are not implemented.

In the file 'node_and_search.py' a breath first search algorithm is implemented in a more general search class. There are also a class defining nodes for search problems. Please investigate the code and figure out how it works. Once you have completed the problem description, try to solve the problem using the *run_assignment_1.py* code. If it does not work, please check all your newly created movement definitions.

Even if your code works, and give you solutions, it still has room or further improvements:

1. **Add a check in the search algorithm for detection of already visited states**, if a state has already been visited it should not be put in the frontier of the search.
2. Add a method **pretty_print_solution** (in Node class), that prints out the solution, i.e., actions needed to go from start to goal. It should include the possibility of a *verbose* flag to toggle printout of actions and states or just to print out the actions. Note, this is simple to implement using a recursive method. Examples usage:

```
mc = MissionariesAndCannibals(init_state, goal_state)
sa = SearchAlgorithm(mc)
solution = sa.bfs()
solution.pretty_print_solution()
```

```
 action:  mc
 …
 action:  m
 action:  mc
```

Examples usage using (verbose=True):

```
mc = MissionariesAndCannibals(init_state, goal_state)
sa = SearchAlgorithm(mc)
solution = sa.bfs()
solution.pretty_print_solution(verbose=True)
```

```
----------------------------
 #miss on left bank:  0
 #cann on left bank:  0
            boat is:  r
#miss on right bank:  3
#cann on right bank:  3
----------------------------
action:  mc
----------------------------
…
----------------------------
 #miss on left bank:  3
 #cann on left bank:  2
            boat is:  l
#miss on right bank:  0
#cann on right bank:  1
----------------------------
action:  m
----------------------------
 #miss on left bank:  2
 #cann on left bank:  2
            boat is:  r
#miss on right bank:  1
#cann on right bank:  1
----------------------------
action:  mc
----------------------------
 #miss on left bank:  3
 #cann on left bank:  3
            boat is:  l
#miss on right bank:  0
#cann on right bank:  0
 ----------------------------
```

3. Add a method **statistics**, that informs the user about (for a solution): *depth, search cost (number of nodes generated), cost for solution (typically the number of nodes in path from root node to goal), cpu time consumed* and finally the *effective branching factor (use the approximation $N^{(1/d)}$)* where N = Total number of nodes and, d = depth. Here it is recommended that you use the *process_time* function from *time* module, to compute the cpu time. Note also that search cost is best stored as a static class variable. A report should look similar to this:

```
solution = sa.bfs(statistics=True)
solution.pretty_print_solution()

----------------------------------------
Elapsed time (s): 2.421875
Solution found at depth:  11
Number of nodes explored:  24279
Cost of solution:  11
Estimated effective branching factor:  2.504132732032241
----------------------------------------
```

Your next assignment is to implement depth first search (DFS). Do this by creating a new method under the *SearchAlgorithm* class. Here a good idea is to use a copy of the breadth first search (BFS) code as a basis.

1. Compare both DFS and BFS (use the statistics function), save the results in a **text document,** named *Assignment 1 – Report + Names of Group members*
2. If you *remove the check for already visited* states in DFS and run the problem, what happen then? Write your answer in the text document.
3. Compare BFS with and without *already visited states check*. Write the statistics in the text document.

Your next assignment is to implement iterative deepening search (IDS). Do this by creating a new method under the *SearchAlgorithm* class. As you know this algorithm is easy to construct when using *depth limited search* as one component. Add a parameter to your DFS algorithm, that if set, limits the algorithms depth.

1. Run the IDS algorithm and get the statistics from it, add the results to your text document.

Informed search

Now it is time to look at informed search algorithms. You'll start out with a new (harder) problem to tackle. This time you need to define the 8-puzzle, do this in a new file (eight_puzzle.py) and define a class named **EightPuzzle**. Tips for modelling this problem is to use a list of list, to denote the positions in the puzzle. Hence the initial state and goal state look like: `[[7,2,4],[5,'e',6],[8,3,1]]` respectively `[['e',1,2],[3,4,5],[6,7,8]]`. Where 'e' denotes the empty position in the puzzle. Moves are easy to describe by moving the 'e' and swapping values in the direction of movement, i.e., if from initial setting we move the 'e' down we need to swap places for 'e' and 3. Use the m&c problem description as an inspiration for your problem description and also note that methods such as *move, pretty_print* and *check_goal* needs to be implemented.

For informed search to work, *heuristics* are needed. Fortunately for the 8-puzzle there are two easy to implement heuristics, *Number of tiles out of place* (h_1) and *Manhattan distance* (h_2). **Implement both of the heuristic methods**, named (h_1) and (h_2).

Now it is time to **implement Greedy search** (name the method **greedy_search**), here you need to associate the heuristic cost with the position in the queue. *PriorityQueue* sorts entries by the lowest cost, exactly wat we want, so use this in conjunction with DFS to create this search algorithm.
As our node definition do not implement methods for it to be *comparable*, you'll need to either (implement the necessary methods) or ignore this and wrap our nodes in another class, *PrioritizedItem*, see: https://docs.python.org/3/library/queue.html#simplequeue-objects for more details. In the latter case add:

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

(15, <node_and_search.Node object at 0x0000017AF003F0D0>)
(15, <node_and_search.Node object at 0x0000017AF003A9A0>)
same priority goes to next and its not comparable . if string compares etc

to your code and add node to the frontier like:

```
frontier.put(PrioritizedItem(child.state.h_1(), child.start))
```

Compare the **outcome of the different heuristics using greedy search** save the statistics in your text document.

Finally, modify greedy_search algorithm to A*, i.e., update the value for sorting nodes, to be estimated_cost = previous_cost + heuristic_value. Name the method **a_star.** Compare the **outcome of the different heuristics using A*** save the statistics in your text document.

Finally, try the uninformed search methods on the eight_puzzle problem, **save their performance measure (statistics)** in your text document. If they fail to find a solution within 5 minutes, abort the experiment, and assign it *time_out* in your text document.

Also, note that not all setups of the eight_puzzle is solvable, see more about the topic here:
https://math.stackexchange.com/questions/293527/how-to-check-if-a-8-puzzle-is-solvable

Summary of assignment

Your groups assignment should contain the code that you have produced:
'missionaries_and_cannibals.py' - completed according to assignment.
'Eight_puzzle.py' - working problem description for the eight puzzle.
'run_assignment_1.py' - containing all your experimental calls, in the same order as they are requested in this document. Note that you should comment out (use ''' spanning rows ''' or #line) all but the first call in your final hand in.
'node_and_search.py' – should contain, bdf, dfs, ids, greedy_search, a_star. Also contain updates for statistical measurement, nodes, etc.
text document – Notes from your experiments, in the same order as they are requested in this document.
Add all files above into a zip file with the last names of both group members, like
'LastName1_LastName2_ass1.zip'