Russian-Armenian University
Artificial Intelligence
Spring 2022

Homework Assignment 1

Due Date: Saturday, March 26 by 23:59 electronically

Introduction

This programming assignment investigates the implementation of some basic search strategies in a generic search library. You should ensure that the code of your solution is well documented.

A Description of the Code Archive

Before starting your work on this programming assignment, download the code archive attached to it and import its contents into an Integrated Development Environment. (Use of an IDE is not mandatory; however, you are likely to find working on this programming assignment significantly easier if you use an IDE, such as IDLE or PyCharm).

The code in the archive is organized in two modules. **You should familiarize yourself thoroughly with the code in these modules, and make sure you understand how it works, before you start working on these tasks.** This will save you a lot of time later. A brief description of the classes and functions in the two relevant modules is given below.

The *search* Module

- The *BreadthFirstTreeSearch* class implements the breadth first tree search algorithm. Its *find_solution()* static method is given an initial state, and it returns a node that contains a solution, or *None* if no solution can be found.
- The *Node* class represents a search node. It stores a reference to the parent node; a reference to the current state; and a reference to the action that, when applied to the state stored in the parent node, produces the state stored in this node. For the root node, parent and action are set to *None*.
- The *Printing* class sets up the framework for solution printing. It contains the *print_solution()* class method that, given a solution node, prints the path to the solution by calling two (abstract) static methods *print_action()* and *print_state()* to print each *Action* and *State* in this path, respectively.
- The *Action*, *GoalTest* and *State* classes specify how the *BreadthFirstTreeSearch* class interacts with the particular problem to be solved.

The *npuzzle* Module

The *npuzzle* module describes the n-puzzle problem and provides suitable child classes of the *Action*, *GoalTest* and *State* parent classes.

- The *Tiles* class extends the *State* class and represents a configuration of tiles. The *width* member specifies the width of the puzzle; thus, each puzzle consists of tiles arranged in a $width \times width$ grid. The layout of the tiles is stored in the *_tiles* array, where $\_tiles[i \times width + j]$ determines the number of the tile located in row i and column j of the grid; the empty tile is identified by the tile number 0. Finally, *empty_tile_row* and *empty_tile_col* contain the row and the column of the empty tile, respectively; although these values can be obtained from the tiles array, they are stored explicitly for convenience.
- The *Movement* class extends the *Action* class and enumerates the four directions in which the empty tile can be moved.
- The *TilesGoalTest* class extends the *GoalTest* class and checks whether all tiles are in the correct positions.
- The *NPuzzlePrinting* class extends the *Printing* class by overriding the static methods *print_action()* and *print_state()* to print a movement and a tile configuration, respectively.
- The *BFTS_demo* function demonstrates the breadth-first tree search (this is the meaning of the *BFTS* prefix). When started, the *BFTS_demo()* function creates an initial tile configuration, invokes the search process, and prints a solution (if one was found).

Task 1 (12 points): Encapsulate the Notion of a Frontier

Whether a search algorithm is depth- or breadth-first depends on the way in which the frontier is managed during the search process. Your task is to encapsulate this behaviour in a separate parent class and provide the child classes of this class that implement depth-first and breadth-first frontiers. Doing this will allow you to implement tree and graph search in a generic way, without hard-coding the frontier behaviour into the search algorithm.

To this end, add to the *search* module a class called *Frontier* that contains empty methods, i.e. with empty bodies, providing the following functionality.

- It should be possible to add a node to the frontier.
- It should be possible to clear the contents of a frontier.
- It should be possible to test whether the frontier is empty.
- If the frontier is not empty, it should be possible to remove and return a node from the frontier.

Furthermore, create two child classes of the *Frontier* class called *DepthFirstFrontier* and *BreadthFirstFrontier*, each implementing the required behaviour.

Task 2 (4 points): State Equality

In order to implement graph search, you will need a way to check if states are equal to each other; classes extending the *State* class are required to support the notion of equality.

In Python, this can be achieved by implementing the following two methods:

- *a.__eq__(b)*
- *a.__hash__()*

Add these two empty methods to the *State* class. Furthermore, implement these methods in the *Tiles* class in the appropriate way.

The general principles for implementing a proprietary notion of equality in Python are available in Python textbooks and online [1].

Task 3 (14 points): Encapsulate Search Algorithms

Add to the *search* module a class called *Search* that encapsulates the notion of a search algorithm. The class should provide an empty method that, given an initial state and a goal test, returns a node containing a solution or *None* if no solution can be found.

Provide two child classes of the *Search* class called *TreeSearch* and *GraphSearch*, each overriding the inherited method for finding a solution appropriately. The constructors of both classes should take an instance of the *Frontier* class, which will parameterize the search algorithm with the appropriate frontier behaviour. In this way it should be possible to obtain the four combinations of depth-first vs. breadth-first search and graph vs. tree search without any code duplication; for example, to obtain depth-first graph search, one should instantiate the *GraphSearch* class parameterized with an instance of the *DepthFirstFrontier* class. A detailed pseudocode for the general graph search algorithm is provided here for your convenience.

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

[1] e.g. https://hynek.me/articles/hashes-and-equality/

Please note that breadth-first search is typically implemented with modified versions of tree search and graph search to make use of *early goal tests*[2]. Provide two more child classes of the *Search* class called *BreadthFirstTreeSearch* and *BreadthFirstGraphSearch*. The constructors of these classes should take no arguments but create *BreadthFirstFrontier* instances to be used during the search.

To provide information about the performance of each kind of search, extend the *Search* class with an empty method that returns the number of nodes generated during the last search, extend the *Frontier* class with an empty method that returns the maximum number of nodes stored on the frontier since the frontier was created, and implement these methods in the relevant child classes.

Task 4 (12 points): Implement Iterative Deepening

Provide a child class of the *Search* class that implements iterative deepening; the inheriting class should be called *IterativeDeepeningTreeSearch*. The constructor of the class should take no arguments (this is because iterative deepening always performs depth-first search, so it is not necessary to parameterize it with a frontier). Instead of using recursion to implement depth-limited search, you should use the *DepthFirstFrontier* and a non-recursive algorithm. In order to cut off the search at a particular depth, you will need to extend the *Node* class with an integer member called *depth* which will keep track of the node's depth; this member should be set in the constructor of the *Node* class.

Task 5 (8 points): Compare Efficiency

For the n-puzzle game, implement code that will execute the following types of search:

- breadth-first tree search (both optimised and with general tree search),
- breadth-first graph search (both optimised and with general graph search),
- depth-first tree search,
- depth-first graph search, and
- iterative deepening tree search.

In addition to running the search and printing the results, in each case your code should print

- the total number of nodes generated during the search, and
- the maximum number of nodes stored on the frontier at any point during the search.

Discuss in a few sentences the results obtained by running the above-mentioned search classes on the n-puzzle game.

[2]You can refer to our lecture slides for detailed pseudocode of the breadth-first tree/graph search

Task 6 (15 points): Implement Best-First Search

Add to the *search* module a class called *NodeFunction*. This class should contain a single empty method that is given a node and that produces an integer value. Child classes of this class will provide the values that are denoted as $f(n)$ and $h(n)$ in the lectures.

In best-first search, nodes are sorted on the frontier according to a *node value*; in lectures, this value is denoted as $f(n)$. To this end, extend the *Node* class with a member that keeps track of the node's value. This member should be set when adding a node to the best-first frontier, as described next.

Provide a child class of the *Frontier* class called *BestFirstFrontier*. The constructor of this class should accept an instance of the *NodeFunction* type that will be used to compute $f(n)$. When a node is added to a best-first frontier, the frontier should invoke the evaluation function on the node to determine the node's value, and it should store the value in the node. (In this way, the node's value is cached and does not need to be recomputed over and over.) When removing a node from a best-first frontier, a node with the lowest value should be returned; if there are several such nodes, then one of them should be selected arbitrarily. A simple way to implement this behaviour is to keep the nodes on the frontier sorted by their value; you can use Python's *heapq* class to obtain the required behaviour.

Task 7 (10 points): Implement the A* Node Function

Extend the *Action* class with an empty method that returns the action's cost; in lectures, this is denoted as $c(n', a, n)$. You should assume that the cost is a non-negative integer. Implement this method in the n-puzzle problem.

Extend the *Node* class with a member that keeps track of the cost of the path from the root node; in lectures, this is denoted as $g(n)$. This value should be set in the constructor of the *Node* class. For a root node n, set $g(n) = 0$; and for a node n with parent n', set $g(n) = g(n') + c(n', a, n)$.

The A* node value is given by $f(n) = g(n) + h(n)$, where $h(n)$ is a heuristic function that estimates the cost of reaching a goal from node n. Provide a child class of the *NodeFunction* called *AStarFunction*. The constructor of *AStarFunction* should accept an instance of *NodeFunction* called *heuristicFunction*; the latter should provide the *AStarFunction* with $h(n)$. When an instance of *AStarFunction* is asked to provide the value for a node n, it should invoke *heuristicFunction* in order to obtain the value of $h(n)$, and it should then return $g(n) + h(n)$.

Task 8 (3 points): Implement the UCS Node Function

Provide a trivial child class of the *NodeFunction* class called *UCSFunction* that can provide the value for a node n to be its path-cost.

Task 9 (10 points): Implement a Heuristic for the n-puzzle

Provide two child classes of the *NodeFunction* class that, given a node n in a search tree for the n-puzzle problem, provide an admissible and consistent heuristic function for that node.

Task 10 (12 points): Compare Efficiency

For the n-puzzle game, implement code that will execute the following types of search:

- Uniform-cost tree search,
- Uniform-cost graph search,
- Greedy tree search (variants with both heuristics),
- Greedy graph search (variants with both heuristics),
- A* tree search (variants with both heuristics), and
- A* graph search (variants with both heuristics).

Your implementation should print the total number of nodes generated during the search and the maximum number of nodes stored on the frontier at any point during the search. Compare and discuss these values for uniform-cost search, greedy search, A* search and the values obtained for uninformed search strategies from Task 5.