Assignment 2 - Artificial Intelligence

May 19, 2019

Student: Simon Jozsef-Gabriel

Group: 2.3 A

Speciality: CE

Contents

1 Problem 1 - The Water Jug Problem **3**
1.1 Problem Statement 3
1.2 Algorithms and Model Description 3
1.3 Experimental Data 5

2 Problem 2 - 15-Puzzle Problem **5**
2.1 Problem Statement 5
2.2 Algorithms and Model Description 5
2.3 Experimental Data 7

3 Conclusions **8**

4 References **9**

1 Problem 1 - The Water Jug Problem

1.1 Problem Statement

The Water Jug Problem: Given two jugs without any measuring markings on it, a 4-litre one and a 3-litre one and an infinite source of water, how can you get exactly 2-liters of water in the 4-liters jug? Based on the given Python framework, create a model representation for this problem and apply proper informed/uninformed search strategy.

1.2 Algorithms and Model Description

All the methods and the subclass are implemented in a file named *P1.py*, and used in *main.py*.

Firstly, I implemented a function which verifies if the resulting state is valid for the problem. The state is valid if the water you pour in one of the two jugs does not exceeds its total capacity and if, after some actions over it, does not goes under 0. The function looks like this:

function isValid($state$)
    $capacityJug1, totalCapacityJug1, capacityJug2, totalCapacityJug2 \leftarrow state$
    if $capacityJug1 > totalCapacityJug1$ **OR** $capacityJug1 < 0$ **AND** $capacityJug2 > totalCapacityJug2$ **OR** $capacityJug2 < 0$ **then**
        return $False$
    end if
    return $True$
end function

After this, I created a subclass of the problem class named ***WaterJugProblem*** which inherited the base class named ***Problem***. In this, I have extracted from the framework given to us at the laboratory the functions, and edited the last one, more exactly, the actions function. I took from this framework antoher two files(*utils.py* and *search.py*) where all the helping functions are implemented. In the actions function, I wrote all the cases for resolving the problem.

The array *cur_state* represents all the current states for all my variables, and is the array on which I worked. This array contains, on position 0, the current state of the variable *capacityJug1*, on position 1, for variable *totalCapacityJug1*, on 2, *capacityJug2*, and on position 3, *totalCapacityJug2*. First of all, I managed the case when the first jug is empty(equal to 0), and for this case, I filled the first jug till it's full. The second case that I managed was when the second jug is empty, and for this case I have 2 actions, the first one is filling the second jug with water from the first jug till it's full, and the second one is pouring all the remaining water in the second jug from the first jug. After that I managed the third case, when the second jug reaches its total capacity, and I gave the action to throw all the water inside it. The fourth and final case, is when the first jug

it's full, and in this case, I just pour the water from it in the second jug, till the second jug is full. This are all the cases I wrote, and after each one of them, I use the function *is_valid* to verify if the result state is good.

Here is the implementation for this class without the function extracted from the framework, and only with the function that I wrote:

```
class WaterJugProblem(Problem)
  function ACTIONS(self, cur_state)
      actions ← []
      self.visited_states.append(cur_state)
      if cur_state[0]=0 then
          new_state ← (cur_state[0]+cur_state[1], cur_state[1], cur_state[2], cur_state[3])
          if is_valid(new_state) then
              actions.append(new_state)
          end if
      end if
      if cur_state[2]=0 then
          new_state ← (cur_state[0]−(cur_state[3]−cur_state[2]), cur_state[1], cur_state[2]+
  (cur_state[3] − cur_state[2]), cur_state[3])
          if is_valid(new_state) then
              actions.append(new_state)
          end if
          new_state ← (cur_state[0] − cur_state[0], cur_state[1], cur_state[2] +
  cur_state[0], cur_state[3])
          if is_valid(new_state) then
              actions.append(new_state)
          end if
      end if
      if cur_state[2]=cur_state[3] then
          new_state ← (cur_state[0], cur_state[1], cur_state[2]−cur_state[3], cur_state[3])
          if is_valid(new_state) then
              actions.append(new_state)
          end if
      end if
      if cur_state[0]=cur_state[1] then
          new_state ← (cur_state[0]−(cur_state[3]−cur_state[2]), cur_state[1], cur_state[2]+
  (cur_state[3] − cur_state[2]), cur_state[3])
          if is_valid(new_state) then
              actions.append(new_state)
          end if
      end if
      return actions
  end function
end class
```

As you can see, I tried using an uninformed search strategy for this problem, similar to which we used at the laboratory for cannibals and missionaries, and the goat, cabbage and wolf problems. The formalization of this problem is: **State Space:** tuple(capacityJug1, totalCapacityJug1, capacityJug2, totalCapacityJug2) with 0 <capacityJug1 <totalCapacityJug1 and 0 <capacityJug2 <totalCapacityJug2; **Initial State:** (0, 4, 0, 3); **Actions:** we can fill up the jugs, pour water from one of the jugs into the other, throw away the water from the jugs, but some actions are illegal, like pouring water from the first jug into the second one if it's full; **Goal state:** (2, 4, 0, 3).

1.3 Experimental Data

I tried to generalize the algorithm as much as i could, but this algorithm gives the right answer to the problem. In the problem, we have two jugs, one with capacity of 4-liters(as the first jug) and one with the capacity of 3-liters(as the second jug).

For the algorithm I wrote, the initial state of this example will be (0 - liters of water in the first jug, 4 - the capacity of the first jug, 0 - liters of water in the second jug, 3 - the capacity of the second jug), and the goal state will be (2, 4, 0, 3).

With my algorithm, the path for reaching the goal state for the initial state will be as follows: **initial state** →(0, 4, 0, 3) →(4, 4, 0, 3) - fill up the first jug →(1, 4, 3, 3) - pour water from the first jug into the second one till it's full →(1, 4, 0, 3) - throw the water from the second jug →(0, 4, 1, 3) - pour the remaining water from the first jug into the second one →(4, 4, 1, 3) - fill up the first jug →(2, 4, 3, 3) - pour water from the first jug into the second one till it's full →(2, 4, 0, 3) - throw the water from the second jug →**goal state**.

2 Problem 2 - 15-Puzzle Problem

2.1 Problem Statement

Based on the 8-puzzle problem from the provided framework build a model representation of the 15-puzzle problem and create two heuristic functions (different from those presented at the laboratory) to solve the problem by applying A* search strategy.

2.2 Algorithms and Model Description

All the methods and the classes I had to implement are found in a file named *P2.py*, and are used in *main.py*.

I had to create two subclasses of the problem class in which to implement two heuristic functions in order to solve my problem. The first subclass named ***FifteenPuzzleMissManh*** which inherits the base class named ***FifteenPuzzle***

contains only one function h which has two parameters *self - the current object, node - the current node of the state* and calculates the heuristic h(n) =the number of misplaced tiles + Manhattan distance, which will return at the end of the function. I used this double heuristic function because I thought that combining two usual and successful heuristic functions will give me a more exact solution for my problem. In *main.py* I create one instance for this class which by calling the function *astar_search().solution()* will compute the correct A* search strategy using the returned values from the heuristic function I implemented.

Here is the implementation code written in pseudocode:

class FifteenPuzzleMissManh(FifteenPuzzle):
  function H($self$, $node$)
    $mMissManh \leftarrow sum(s \neq g + abs(s\ modulo\ 4 - g\ modulo\ 4) + abs(s/4 - g/4)\ for\ (s, g)\ in\ zip(node.state, self.goal))$
    return $mMissManh$
  end function
end class

As you can see here, in this algorithm, I call some function which are built-in functions, and this functions are sum - which returns the sum of a 'start' value plus an iterable of numbers - and zip - which returns a zip object whose .__next__() method returns a tuple where the i^{th} element comes from the i^{th} iterable argument. The two variables *s and g* are representing the current i^{th} state of the node(s) and the goal state of the i^{th} element(g). So, for this case, the *sum* function will return the sum between the value 1 or 0 given by the state $s \neq g$(if this is true will give the value 1, otherwise will give 0), and the value given by the second operation which calculates the Manhattan distance between two points p1(x_1,y_1) and p2(x_2,y_2) from the puzzle, and which has the formula $|x_1 - x_2| + |y_1 - y_2|$. In the function modulo of 4 calculates the x coordinates of the points and the division with 4 calculates the y coordinates.

The second subbclass named ***FifteenPuzzleEuclidean*** which inherits the same base class as the other class, ***FifteenPuzzle***, contains one function named h which has two arguments *self* as the current object and *node* as the current node of the state and calculates the heuristic h(n) =the Euclidean distance, which will return at the end of the function. I used the Euclidean distance heuristic function because I wanted to show that it can be successful and optimal even if is a L_2 distance, and is not as good as Manhattan distance. The formula of Euclidean distance between two points p1(x_1,y_1) and p2(x_2,y_2) is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. In *main.py* I create one instance for this class which by calling the function *astar_search().solution()* will compute the correct A* search strategy using the returned values from this heuristic function that I implemented.

Here is the implementation code written in pseudocode:

class FifteenPuzzleEuclidean(FifteenPuzzle):
  function H($self$, $node$)
    $mEucl \leftarrow sum(sqrt(s\ modulo\ 4 - g\ modulo\ 4)^2 + sqrt(s/4 - g/4)^2\ for\ (s, g)\ in\ zip(node.state, self.goal))$
    return $mEucl$
  end function
end class

In this heuristic algorithm implementation you can see that I have used the same built-in methods like those from the first subclass(*sum() and zip()*) in order to get the correct result for the problem. The two variables s and g represent the same thing as in the first subclass, s is the current i^{th} state of the node and g is the goal state of the i^{th} element. As for the algorithm, I implemented the sum of all Euclidean distances between the i^{th} element of the current state s and the i^{th} element of the goal state g, by applying the Euclidean distance formula, where the coordinates x are calculated with modulo of 4 and the coordinates y are calculated with division with 4.

For this problem, I tried to implement the heuristic functions which we needed for applying the A* search strategy, which are similar to the ones from the laboratory where it has been presented to us the 8-puzzle problem. The formalization of this problem is: **State Space:** tuple(initial, goal =(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0)); **Initial state:** we can have several initial states but the ones that I used are: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 0, 13, 14, 12, 11) and (1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 10, 15, 13, 14, 12, 11); **Actions:** we can go UP, RIGHT, DOWN and LEFT, depending on the position of the tile with the value 0, for example, if this tile is on the first row, it can't go UP; **Goal state:** (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0).

2.3 Experimental Data

For this problem I have two examples, one which applies the first heuristic function that I wrote(missplaced tiles + Manhattan distance), and the other one which applies the second heuristic function that I wrote(Euclidean distance).

First Example

The resulted path for the first heuristic function with the initial state (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 0, 13, 14, 12, 11) will be: ['DOWN', 'LEFT', 'UP', 'RIGHT', 'DOWN']. The solution path will compute like this: **initial state** →(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 0, 13, 14, 12, 11) →the algorithm will find that the best move for this case is 'DOWN', which is correct, so the tuple will become (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 11, 13, 14, 12, 0) →the next best move that the algorithm computes is 'LEFT' which is also correct, and the tuple will become (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 11, 13, 14, 0, 12) →the next best move that the

algorithm computes is 'UP' which is correct, and the tuple will become (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 11, 13, 14, 15, 12) →the next best move that the algorithm computes is 'RIGHT' which is correct, and the tuple will become (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 0, 13, 14, 15, 12) →the next best move that the algorithm computes is 'DOWN' which is also correct, and the tuple will become (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 11, 13, 14, 15, 0) →**goal state**.

Second Example

The resulted path for the second heuristic function with the initial state(1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 10, 15, 13, 14, 12, 11) will be: ['RIGHT', 'RIGHT', 'DOWN', 'LEFT', 'UP', 'RIGHT', 'DOWN']. The solution path will compute like this: **initial state** →(1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 10, 15, 13, 14, 12, 11) →the algorithm find that the best move to start is 'RIGHT' which is correct, and the tuple will change like this (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 15, 13, 14, 12, 11) →the algorithm will find that the best next move is 'RIGHT' which is correct, and the tuple will become (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 0, 13, 14, 12, 11) →the following moves are exactly the same as the example from above, and it is sure that will reach the goal state.

3 Conclusions

By working on this project I managed to understand some basics about searching in Artificial Intelligence, more exactly the uninformed and informed searching strategies, and also, I managed to understand heuristic functions, and how to apply different distance algorithms, like Manhattan distance and Euclidean distance.

In conclusion I think that the algorithms I wrote for both problems are correct and assures the user that it will reach the goal state and solve the problem by giving the right and optimal solutions.

4 References

1) https://stackoverflow.com/

2) https://en.wikibooks.org/wiki/LaTeX

3) https://tex.stackexchange.com

4) Artificial Intelligence courses and laboratories

5) https://xlinux.nist.gov/dads/HTML/euclidndstnc.html

6) https://xlinux.nist.gov/dads/HTML/manhattanDistance.html

7) https://www.sharelatex.com

8) http://ethesis.nitrkl.ac.in/5575/1/110CS0081-1.pdf

9) http://theory.stanford.edu/ amitp/GameProgramming/Heuristics.html