Course Assignment - Artificial Intelligence

June 8, 2019

Student: Simon Jozsef-Gabriel

Group: 2.3 A

Speciality: CE

Contents

1 Problem Statement

For choosing the problem we had to compute a hash function which calculates the number of the problem according to our family name and the first given name like this: 1 + (the sum of ASCII codes of the names modulo 4). My hash function calculates this number according to the next algorithm:

```
function HASH($name$) returns a number
    return $1 + (sum(ord(c)\ for\ c\ in\ name)\ modulo\ 4)$
end function
```

This algorithm takes every letter from the family name and first given name denoted by the string variable *name* and after calculating his ASCII code, it sums up with the other letters. After the sum is computed, it takes the modulo 4 from it and in the end, at the final value, adds 1.

An example in which you can see how it computes the problem value it is given below for my names:

EX: name = SIMON JOZSEF → calls the algorithm hash(SIMON JOZSEF) → ord(c) means the ASCII code of the character c, so for the characters from my name it will be like this: ord(S) = 83, ord(I) = 73, ord(M) = 77, ord(O) = 79, ord(N) = 78, ord( ) = 32 (this does not influence the outcome), ord(J) = 74, ord(O) = 79, ord(Z) = 90, ord(S) = 83, ord(E) = 69, ord(F) = 70 → after this, sum(ord(c)) will be equal with 83 + 73 + 77 + 79 + 78 + 32 + 74 + 79 + 90 + 83 + 69 + 70 = 887 → the final value, 887 modulo 4 = 3 at which we add 1 so, the final result, and the problem chosed for me is → 3 + 1 = 4.

The statement of the homework problem chosed for me is: n vehicles occupy squares (1, 1) through (n, 1) (i.e., the bottom row) of a nxn grid. The vehicles must be moved to the top row but in reverse order; so the vehicle i that starts in (i, 1) must end up in (n-i+1, n). On each time step, every one of the n vehicles can move one square up, down, left or right, or stay put; but if a vehicle stays put, one other adjacent vehicle(but no more than one) can hop over it. Two vehicles cannot occupy the same square.

Firstly I have to say that instead of using a matrix with i,j positions, I gave to each square from the matrix a number from 0 to the size of the grid raised to 2 power like this: 1,1 → 0, 1,2 → 1 and so on, till you reach n,n → the size of the grid raised to 2 power. According to the idea the formulation for this search problem is:
State space: tuple (0,..., i*gridSize,..., (n-1)*gridSize), so the state space will be equal with n.
Initial state: the position of each vehicle *i* according to what i said above, so it will be equal with the multiplication of the index of the state, *i*, and the size of the grid *(i*gridSize)*, which represents the first column. For example, for *gridSize* = 4 the initial state will be a tuple (0, 4, 8, 12).
Actions: according to my algorithm, the vehicle can move RIGHT, LEFT, UP

or DOWN, or stay put (PUT), depending on it's position on the grid and the other vehicles position. So, for example, for the vehicle with the number 0 (i = 0), which is situated in the position 0 on the grid (the first position 1,1), the vehicle cannot go up and left, because it would be outside of the grid, and neither down, because another vehicle is there.
Goal state: the positions where each of the *i* vehicles have to reach are the top row in reverse order, so that means, that for each *(i,1)* vehicle, the final destination will be *(n-i+1,n)*, which according to my algorithm will be *(gridSize^2 - initial(i) - 1)*. For the example from the initial state, with *gridSize* = 4, the goal state will be the tuple (15, 11, 7, 3).
Path cost: each action has a cost of 1.

To solve this problem I used the A* search algorithm and calculated the heuristic function with the Manhattan distance with the formula: $|x_1 - x_2| + |y_1 - y_2|$.

2 Pseudocode of Algorithms

The algorithms I used for this problem are integrated in several functions from a class named ***VehicleRoads*** which inherits the class from the file *search.py*, ***Problem***. The pseudocode for these functions are presented below:

```
function __INIT__($self, gridSize, initial, goal$) initializes the problem with the size of the grid, the initial state and the goal state
    persistent: $self.stateIndex$, index used to acces an element from the state
    $self.gridSize \leftarrow gridSize$
    $self.initial \leftarrow initial$
    $self.goal \leftarrow goal$
    $self.stateIndex \leftarrow -1$
    PROBLEM.__INIT__($self, initial, goal$)
end function

function ACTIONS($self, state$) returns the possible actions for each state
    persistent: $possibleActions$, a list with possible actions
    $positionVehicle$, the position of the i^th vehicle
    $positionVehicle1$, the position of the i^th vehicle used to verify if the current vehicle can do an action
    if $self.stateIndex = self.gridSize - 1$ then
        $self.stateIndex \leftarrow 0$
    else
        $self.stateIndex \leftarrow self.stateIndex + 1$
    end if
    $possibleActions \leftarrow list['UP'+str(self.stateIndex),' DOWN'+str(self.stateIndex),' LEFT'+ str(self.stateIndex),' RIGHT'+str(self.stateIndex),' PUT'+str(self.stateIndex)]$
    $positionVehicle \leftarrow state[self.stateIndex]$
    if $positionVehicle < self.gridSize$ then
        if $'UP' + str(self.stateIndex)\ in\ possibleActions$ then
            $possibleActions.remove('UP' + str(self.stateIndex))$
        end if
    end if
    if $positionVehicle \geq self.gridSize * (self.gridSize - 1)$ then
        if $'DOWN' + str(self.stateIndex)\ in\ possibleActions$ then
            $possibleActions.remove('DOWN' + str(self.stateIndex))$
        end if
    end if
    if $positionVehicle modulo self.gridSize = 0$ then
        if $'LEFT' + str(self.stateIndex)\ in\ possibleActions$ then
            $possibleActions.remove('LEFT' + str(self.stateIndex))$
        end if
    end if
    if $positionVehicle modulo self.gridSize = self.gridSize - 1$ then
```

```
        **if** $'RIGHT' + str(self.stateIndex)$ *in possibleActions* **then**
            $possibleActions.remove('RIGHT' + str(self.stateIndex))$
        **end if**
    **end if**
    $i \leftarrow 0$
    **while** $i < self.gridSize$ **do**
        $positionVehicle1 \leftarrow state[i]$
        **if** $positionVehicle - self.gridSize = positionVehicle1$ **then**
            **if** $'UP' + str(self.stateIndex)$ *in possibleActions* **then**
                $possibleActions.remove('UP' + str(self.stateIndex))$
            **end if**
        **end if**
        **if** $positionVehicle + self.gridSize = positionVehicle1$ **then**
            **if** $'DOWN' + str(self.stateIndex)$ *in possibleActions* **then**
                $possibleActions.remove('DOWN' + str(self.stateIndex))$
            **end if**
        **end if**
        **if** $positionVehicle - 1 = positionVehicle1$ **then**
            **if** $'LEFT' + str(self.stateIndex)$ *in possibleActions* **then**
                $possibleActions.remove('LEFT' + str(self.stateIndex))$
            **end if**
        **end if**
        **if** $positionVehicle + 1 = positionVehicle1$ **then**
            **if** $'RIGHT' + str(self.stateIndex)$ *in possibleActions* **then**
                $possibleActions.remove('RIGHT' + str(self.stateIndex))$
            **end if**
        **end if**
        $i \leftarrow i + 1$
    **end while**
    **return** possibleActions
**end function**

**function** RESULT($self, state, action$) **returns** the new state calculated according to the possible actions
    **persistent:** $positionVehicle$, the position of the ith vehicle
    $newState$, the new state that is going to be calculated
    $delta$, used to calculate the new state according to the action
    $positionVehicle \leftarrow state[self.stateIndex]$
    $newState \leftarrow list(state)$
    $delta \leftarrow ['UP'+str(self.stateIndex) : -self.gridSize,\ 'DOWN'+str(self.stateIndex) : +self.gridSize,\ 'LEFT'+str(self.stateIndex) : -1,\ 'RIGHT'+str(self.stateIndex) : +1,\ 'PUT' + str(self.stateIndex) : +0]$
    $newState[self.stateIndex] \leftarrow positionVehicle + delta[action]$
    **return** $tuple(newState)$
**end function**
```

```
**function** H($self, node$) **returns** the sum of all Manhattan distances so is the
function that calculates the heuristic for the search problem
    $manhattanDistance \leftarrow sum(abs(s-g)*2\ for\ (s, g)\ in\ zip(node.state, self.goal))$
    **return** $manhattanDistance$
**end function**
```

For generating the input data, I created a new file called *InputDataGenerator.py* that has functions that calculates the size of the grid randomly and also calculates the initial and the goal states according to some rules. The pseudocode for these functions is presented below:

```
**function** GENERATEGRIDSIZE **returns** a rondom integer between 1 and 10
    **return** random.randint(1, 5)
**end function**

**function** GENERATEINITIALSTATE($gridSize$) **returns** a tuple that represent
the initial state for the problem
    $initialState \leftarrow tuple()$
    $i \leftarrow 0$
    **while** $i < gridSize$ **do**
        $initialState \leftarrow initialState + tuple([gridSize * i])$
        $i \leftarrow i + 1$
    **end while**
    **return** $initialState$
**end function**

**function** GENERATEGOALSTATE($gridSize, initialState$) **returns** a tuple that
represent the goal state for the problem
    $goalState \leftarrow tuple()$
    $i \leftarrow 0$
    **while** $i < gridSize$ **do**
        $goalState \leftarrow goalState + tuple([gridSize^2 - initialState[i] - 1])$
        $i \leftarrow i + 1$
    **end while**
    **return** $goalState$
**end function**
```

3 Application Outline

Here, I will present the application design of my code, starting with how I chosed to represent the input and the output data. For the input data, I implemented all the functions in *InputDataGenerator.py*. The functions are:
—**def** generateGridSize();
—**def** generateInitialState(gridSize);
—**def** generateGoalState(gridSize, initialState).

The first function from this file, *generateGridSize()*, which does not have any arguments, just returns a random integer number between 1 and 5. The problem with my code is that if the variable *gridSize* is greater than 5, the algorithm takes too much time to compute the result, and therefore, the result is not printed and I think it enters in an infinite cycle. I don't know if this problem is because of my personal computer or not, but I thought that this detail is important to be written in this tehnical report. And that is the reason why the number must be between 1 and 5.

The second function, *generateInitialState(gridSize)*, which has one argument, *gridSize* which represents the size of the grid randomly given by the first function. Here, I declare a variable named *initialState* as a tuple. After that, I add for each vehicle, its start position(i.e., its initial position, first column) after the rule I denoted in the 1 - Problem Statement - meaning that for each vehicle starting from (i,1) I calculate the position with the formula *gridSize*i*, with $i \in \{0, 1, ..., gridSize - 1\}$. At the end, the function returns the variable *initialState* as computed before.

The last function from this file, and that generates the last input data, is the function named *generateGoalState(gridSize, initialState)* which has two arguments, *gridSize* which represents the size of the grid randomly given by the first function and *initialState* used to calculate the goal state, because the goal state is related with the initial state according to my code. Here, I declare the variable *goalState* as a tuple and I compute for each vehicle the goal state. The position starting from (i,1) - *gridSize*i* - has to reach the top row but in reverse order, so, the vehicles finish position has to be (n-i+1,n), which according to my algorithm described at section 1 - Problem Statement - is the equivalent of the position $gridSize^2$*-initialState[i]-1*, with $i \in \{0, 1, ..., gridSize - 1\}$. At the end, the function returns the tuple denoted by the variable *goalState* computed with the rule above.

The output data is generated by the function *astar_search* from the file *search.py* from the framework given to us. This function calls another function named *best_first_graph_search* which finally will compute the path generated for our problem, so the output. With the method *solution()* from the class *Node* from the same file, I will print the path generated according to my algorithm. The output data is a list of actions: [UPi, DOWNi, RIGHTi, LEFTi, PUTi], where i is the index of the state of the ith vehicle, and which I used to help myself know which of the vehicles made the move.

All the functions I used for my algorithm and for generating an output data(a path list), were implemented in the file named *P4_VehicleRoads.py*.All these functions are part of a class *VehicleRoads*, which is a subclass of *Problem* class from the file *search.py*. The functions are:
—**def** __init__(self, gridSize, initial, goal);
—**def** __repr__(self, initial, goal);
—**def** actions(self, state);
—**def** result(self, state, action);
—**def** goal_state(self, state);
—**def** h(self, node).

The function *__init__(self, gridSize, initial, goal)* is the constructor used to initialize the problem with all the values I need. It has three arguments, so it will initialize and define three variables, more exactly *gridSize* which is the size of the grid, *initial* which is the initial state for my problem, generated using the function described above, at the input and output data presentation, and *goal* which represents the goal state for my problem generated using the function described at the same subsection.

The function __repr__(self, initial, goal) is just a function that helps me represent the states in a more ordered way. This helped me at the debug part of my code.

The next function, *actions(self, state)*, it only has one argument, because *self* is used to access the variable from inside the class. The only argument, *state* represents the state of my problem, used to perform the actions needed and also to access the position of a vehicle. This argument changes after each move of a vehicle, in order to change all the positions, and also the states, of the vehicles. This function returns a list, *possibleActions*, which is a list containing the moves a vehicle can do according to some rules, like: 2 vehicle cannot share the same square, so if a vehicle is above another one, this vehicle cannot go down, so the down actions will be removed from the list with possible actions. Another rule is that a vehicle cannot go outside of the grid, so when it is placed on the first row, it can't go up, so the actions up will be removed from the list with possible actions, for example. This function will be used for each vehicle.

The function that returns the change an action is making over the state of a vehicle is the function *result(self, state, action)*, which has two arguments, *state*, which represents the state of my problem, and *action* which is the current action performed by the vehicle and which will take every element of the list returned from the function above(*actions* function) and will calculate the result of that action over the state, and saves it in a new list called *newState* which represents the state changed after performing an action. This list will be returned at the end of the function as a tuple in order to perform the next actions on it if it's required. For calculating the change an action makes over the state, I introduced a variable named *delta* which is holding the result for each action: if the action will be 'UP' this will substract the value of the variable *gridSize* from the current state of the vehicle, if the action will be 'DOWN' it adds this value to the current state, if the action is 'RIGHT' it adds 1 to the current state, if the actions is 'LEFT' substracts 1, and if the actions is 'PUT' it does nothing, because the vehicle is not moving.

The function that will stop the algorithm is *goal_state(self, state)* which returns True if the state is equal with the goal state, or False if is not. So this function detects when all the vehicles reached the position required.

The last function represents the implementation of the heuristic function I used to solve this problem, *h(self, node)*. It has one argument which is the state of one vehicle represented like a node. The heuristic function I used is the Manhattan distance because this was the most suitable heuristic I found for this problem, and I thought that it will give me the fastest and correct answer for the problem. This function calculates in the variable *manhattanDistance* the sum of all distances from the current state to the goal state according to this formula: $abs(s-g)*2$, where s and g are the current state denoted by *node.state* and the goal of the current state denoted by *self.goal*.

4 Experimental Data

Test 1
The **grid size** is 2x2;
The **initial state** is: (0, 2);
The **goal state** is: (3, 1);
Path: ['RIGHT0', 'UP1', 'DOWN0', 'RIGHT1'], where the actions that end with 0 represent the moves of vehicle with the number 0, and those that end with 1 the moves of vehicle 1;
Execution Time: 0.0.

Test 2
The **grid size** is 3x3;
The **initial state** is: (0, 3, 6);
The **goal state** is: (8, 5, 2);
Path: ['RIGHT0', 'RIGHT1', 'UP2', 'RIGHT0', 'DOWN1', 'UP2', 'DOWN0', 'UP1', 'RIGHT2', 'DOWN0', 'RIGHT1', 'RIGHT2'], where the actions that end with 0 represent the moves of vehicle with the number 0, those that end with 1 the moves of vehicle 1, and those with 2 the moves of the vehicle 2;
Execution Time: 0.000997304916381836.

Test 3
The **grid size** is 4x4;
The **initial state** is: (0, 4, 8, 12);
The **goal state** is: (15, 11, 7, 3);
Path: ['RIGHT0', 'RIGHT1', 'RIGHT2', 'UP3', 'RIGHT0', 'RIGHT1', 'UP2', 'UP3', 'RIGHT0', 'DOWN1', 'DOWN2', 'DOWN0', 'RIGHT1', 'LEFT2', 'RIGHT3', 'LEFT1', 'LEFT3', 'DOWN0', 'DOWN1', 'RIGHT3', 'LEFT1', 'LEFT3', 'DOWN0', 'UP1', 'UP3', 'UP2', 'RIGHT2', 'RIGHT3', 'RIGHT1', 'RIGHT2', 'RIGHT3', 'LEFT0', 'RIGHT3', 'RIGHT0', 'RIGHT1', 'RIGHT2'], where the actions that end with 0 represent the moves of vehicle with the number 0, those that end with 1 the moves of vehicle 1, those with 2 the moves of the vehicle 2, and those with 3 the moves of the vehicle 3;
Execution Time: 0.008975028991699219.

Test 4
The **grid size** is 5x5;
The **initial state** is: (0, 5, 10, 15, 20);
The **goal state** is: (24, 19, 14, 9, 4);
Path: ['RIGHT0', 'RIGHT1', 'RIGHT2', 'UP3', 'UP4', 'RIGHT0', 'RIGHT1', 'RIGHT2', 'RIGHT3', 'RIGHT0', 'RIGHT1', 'RIGHT2', 'UP3', 'RIGHT1', 'RIGHT2', 'RIGHT3', 'UP4', 'RIGHT0', 'LEFT2', 'RIGHT3', 'UP4', 'LEFT0', 'DOWN1', 'LEFT2', 'RIGHT3', 'RIGHT4', 'DOWN0', 'DOWN1', 'LEFT2', 'LEFT4', 'DOWN0', 'LEFT1', 'DOWN2', 'RIGHT4', 'LEFT1', 'DOWN0', 'DOWN1', 'LEFT2', 'LEFT3', 'LEFT4', 'DOWN0', 'UP1', 'RIGHT2', 'LEFT3', 'RIGHT1', 'UP2', 'RIGHT1',

'RIGHT2', 'LEFT3', 'RIGHT2', 'RIGHT3', 'RIGHT4', 'RIGHT2', 'RIGHT3', 'UP4', 'RIGHT3', 'RIGHT4', 'LEFT0', 'RIGHT4', 'RIGHT0', 'LEFT1', 'RIGHT4', 'RIGHT0', 'RIGHT1'], where the actions that end with 0 represent the moves of vehicle with the number 0, those that end with 1 the moves of vehicle 1, those with 2 the moves of the vehicle 2, those with 3 the moves of the vehicle 3, and those with 4 the moves of the vehicle 4;
Execution Time: 0.2812478542327881.

Because of the reason I gave details for in section 3 - Application Outline - regarding the *gridSize* variable which represents the size of the grid, the last test(test 4) is the last value for which my algorithm give an answer. For testing if this paths resulted are correct, I drew the grid on an A4 paper and started to go with the path resulted until I reached the final position.

5 Conclusions

Working at this assignment helped me gain more experience in artificial intelligence search algorithms and also it helped me understand how it works. I also managed to make me study more the Python programming language and gain more codding ability experience. I chosed Python as a programming language for this problem, because the framework given to us was in Python, and that helped me a lot, and because I thought that in this language will be the easiest way to solve the problem, after working at the last 2 laboratories with it.

About the problem, I can say that was a complicated one and it made me some problems, but I am happy that afterall I managed to solve it, even if not completely. From my point of view, the algorithm I implemented, as well as, the heuristic I used for this search problem, are correct and both compute the right answer, but it needs more work to finally make it work perfectly.

6 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/manhattanDistance.html

7) https://www.sharelatex.com

8) Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach.* Prentice Hall, 3rd Edition, 2010.

9) https://www.cs.unc.edu/ lazebnik/fall10/lec05_search.pdf