Moving vehicles

Group CEN2.2A, IInd *year*, 2nd *semester*, *Computers*

Frăsineanu Claudia Andreea

June 7, 2021

1 Problem statement

Let us assume that m vehicles are located in squares (1, 1) through (m, 1) (the bottom row) of an $m \times m$ squared parking. The vehicles must be moved to the top row, but arranged in reverse order; so vehicle i starting from $(i, 1)$ must end up in $(m - i + 1, m)$. On each time step, each of the m vehicles is restricted to move only one square up, down, left, or right, or keep current position (i.e. does not move); but if a vehicle does not move, one other adjacent vehicle (but not more than one) can hop over it. Two vehicles cannot occupy the same square.

a. Write a detailed formulation for this search problem.

b. Identify a suitable search algorithm for this task and explain your choice.

2 Approach

First and foremost we need to determine the bounds between which our solution can vary. To find these limits, we will take into consideration the worst and best cases.

First, we take the case where we have a single car in the parking lot. Because it cannot jump over any other cars, then the distance of it, from a point $(i, 1)$ to $(m - i, m)$ would be the same as the *Manhattan distance*. This would give us the upper bound.

The second case will give us the lower bound, and is the ideal scenario, in which a car will be able to jump over all the cars until it reaches its place. Because the car actually moves over two blocks every time, it means that it will reach the destination in half the usual time, thus in *half of the Manhattan distance*.

Now that we have determined the bounds, we can safely assume that an optimal heuristic for our search algorithm would be given by the Manhattan distance.

Since the given problem is a dynamic one, it becomes clear that we will need to keep track of all the cars and decide which ones to move, and into which direction. Thus the algorithm will be divided into two parts: the *states* each car can be in, and a proper *search algorithm* that will take into consideration both the states and our heuristic function.

2.1 States

Each of the cars can have one of the following states: *move down*, *move up*, *move right*, *move left*, *jump down*, *jump up*, *jump right*, *jump left*, *remain in the current position*.

The key elements regarding the states are the conditions used to ensure that the cars will not move out of bounds. At each step, we will check the conditions for each move, and, if the action can be completed, then it is added to the list of actions.

2.2 Heuristic

As I mentioned above, an optimal heuristic for this problem would be the one given by the *Manhattan distance*. Thus, the function implementing this will be:

$$h = \sum |currentState - goalState| * 2$$

2.3 Search function

To ensure that the most optimal algorithm is chosen, we can test for a simple case each of the search algorithms. We will take into consideration three major factors: the *total cost of the path*, because we need to provide one of the most cost efficient solutions, the *total considered actions*, since they have a crucial impact on the overall performance of the algorithm, and the *runtime*.

An important thing to note is the fact that the way the initial and goal state is organized, greatly affects the performance of the algorithms. There are cases where certain search algorithms are not working because they are returning an *empty* path and thus are not able to call any other functions.

In order to better emphasize these differences, I conducted tests for the two possible layouts. I took into consideration four important search algorithms: *Breadth First Search* on a graph, *Depth First Search* on a graph, *Best First Search* and *A* Search*. For each of these I ran two tests, to see how their performance changes with the size of the parking lot.

Without further due, in the two tables below can be seen the results of the tests conducted if the input and output state looked like this:

$$InitialState = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{bmatrix}, GoalState = \begin{bmatrix} 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Size = 3	Path Cost	Possible Actions	Runtime
Breadth First	11	995	0.008072
Depth First	-	-	-
Best First	21	389	0.00715
A*	11	782	0.01463

Size = 4	Path Cost	Possible Actions	Runtime
Breadth First	17	130063	56.80981
Depth First	161	889	0.01267
Best First	64	8407	0.41881
A*	-	-	-

We can observe from these that there are significant discrepancies between the behaviours of the search algorithms. On one side, we have *Breadth First Search*, which gives an optimal result, but it takes a lot of time to compile, and on the other side we have *Best First Search*, which gives a non-optimal solution, but within a reasonable runtime. *Depth First Search* and *A** cannot be taken into consideration because there are cases for which either of them will not work properly.

Now we can proceed to the second layout, where the two states will be :

$$InitialState = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, GoalState = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 4 & 3 & 2 & 1 \end{bmatrix}$$

Size = 3	Path Cost	Possible Actions	Runtime
Breadth First	8	246	0.00101
Depth First	-	-	-
Best First	-	-	-
A*	12	141	0.01213

Size = 4	Path Cost	Possible Actions	Runtime
Breadth First	-	-	-
Depth First	-	-	-
Best First	64	8407	0.41881
A*	19	28744	16.64201

From this data we can gather that there are two search algorithms that have given optimal results: *Breadth First Search*, for the first type of input, and *A* Search*, for the second type of input. But, because *Breadth First Search* takes too much time to compute the results, I will chose to implement *A* Search*, in order to be able to provide more test results.

2.4 A* Search

A* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes

$$f(n) = g(n) + h(n)$$

where :

- n is the next node on the path,
- $g(n)$ is the cost of the path from the start node to n,
- $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal.

A* terminates when the path it chooses to extend is a path from start to goal or if there are no paths eligible to be extended.

```
function A*-SEARCH(problem, function) returns a solution node or failure
    node ← NODE(STATE = problem. INITIAL)
    frontier ← a priority queue ordered by f, with node as an element
    explored ← a set containing all the visited nodes
    if problem. GOAL-TEST (node. STATE) then return node
    while not IS-EMPTY (frontier) do
            node ← POP (frontier)
            if problem. GOAL-STATE (node. STATE) then return node
            explored ← ADD (node. STATE)
            for each child in EXPAND (problem, node) do
                    if child. STATE is not in explored and child is not in frontier then
                        frontier ← ADD (child)
                    elif child is in frontier then
                        incumbent ← frontier[child]
                        if f(child) < f(incumbent) then
                          delete frontier[incumbent]
                           frontier ← ADD (child)
```

Figure 1: Pseudocode for A* Search

3 Experimental Data

For this problem, the input consists of only one number: the *size of the parking lot*. There are two major limitations of this number: it has to be *positive*, and the maximum value cannot exceed 7, as the tests take too much time to finish and it becomes impossible to obtain the results.

$$number = \{randomNumber | randomNumber \in [1, 7]\}$$

The matrices used to describe the initial and goal state will be generated within the class constructor, in accordance with the *number*.

4 Projection of experimental application

The following section and figures will give further details concerning the structure and organization of the programs. The most important algorithms are separated into their own module, to facilitate the eventual modifications, and, overall to keep the code clean. There is also a separate module for the random generator and two folders where all the input and output files are kept.

4.1 High level application design

The hierarchy of the application is fairly simple. At the bottom we have the *utils* module, where are kept some auxiliary functions needed for solving the problem. Above it is the *search* module, inside which lay a few search algorithms, along with the class *Problem*. The module *movingVehicles* contains the actual solution of the problem, with all the data, states and heuristic. At the top we have the *main* module, where the functions are called and where the actual reading, writing and measuring of runtime happens.

The hierarchy of the project folder is shown in the figure below:

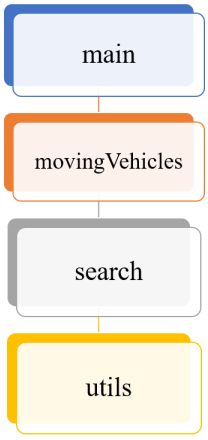


Figure 2: Hierarchy

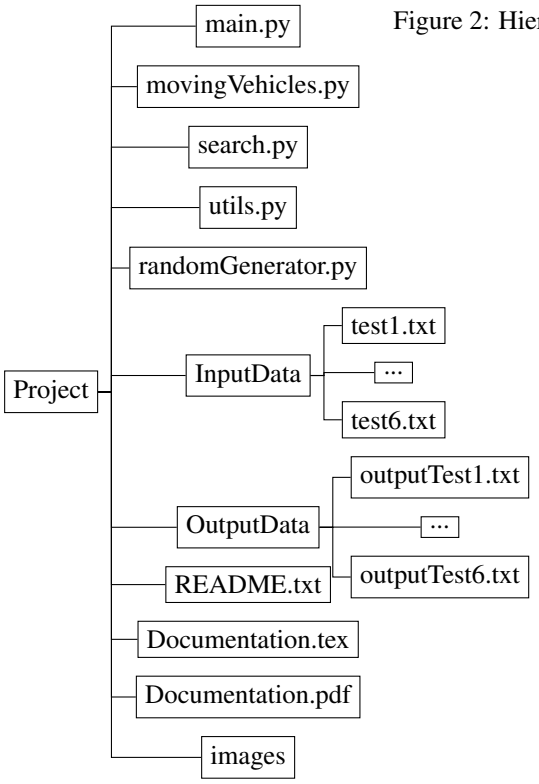


Figure 3: Project hierarchy

Below we have a more in-depth description of each folder and module.

I *Project\InputData*
Contains all the input files needed to generate test cases for the problem.

- *testi.txt*→ all the tests generated randomly, used to research the time execution and compare the different implementations ; $i = \overline{1,6}$

II *Project\OutputData*
Contains all the output files with the results needed to formulate the conclusions for the problem.

- *outputTesti.txt*→ the output resulted from each test, with all the necessary data; $i = \overline{1,6}$

III *Project\main.py*
It is the "main" of the python implementation,holding all the calls of the functions necessary for successfully compiling the program.

IV *Project\movingVehicles.py*
Holds the implementation of the problem, extending the class *Problem* and completing its functions.

V *Project\search.py*
Holds the implementation of the problem, extending the class *Problem* and completing its functions. In here we define the states and the heuristic function.

VI *Project\utils.py*
Holds the implementation of the problem, extending the class *Problem* and completing its functions. In here we define the states and the heuristic function.

4.2 Implemented functions

Due to the fact that the structure and hierarchy of the whole project has been taken into consideration, the following subsections will offer further insight regarding each function, variable, as well as the input and output data.

I In *main.py*

main()

Its purpose is to serve as a sort of "front-end", dealing only with the initialization of the variables, calls of the functions and measuring the execution time. It also serves as a link both between the subprograms and the input and output files.

II In *movingVehicles.py*

class *MovingVehicles (Problem)*

This class is an extension of the class *Problem* found in *search.py* module, and it contains the essential functions needed to solve our search problem and their implementations.

(a) *__init __(*self*, parkingSize)*
This is the constructor of the class, and it takes care of all the variables that need to be initialized. Its only parameter is:

- *parkingSize* → the size of the parking lot. It is needed in order to create the matrices used to define the initial and goal state.

(b) *findFirstEmptySpot(*self*, state)*
It searches for the next car that can be moved and returns its state. The only variable is:

- *state* → after searching for an empty space, we return the state of the car object at the new position. In order to do that, we need to keep track of all the states.

(c) *actions(*self*, state)*
This is the implementation of the function found in the class *Problem*, in *search.py*. Inside this function are defined all the possible actions a car can make at a given moment. We start with a list of *possible actions*, where we add the "default" action, which is *Stay*. After that we check if we have reached the goal node, and if true, we return the list.
If we have not reached the goal node, we begin to check if the conditions for each possible move is met, i.e. we ensure that if the car completes the move it does not go outside of bounds. As mentioned in section 2.1, all the possible moves an object can make at a time are:

actions = { Stay, MoveUp, MoveDow, MoveLeft, MoveRight, JumpUp, JumpDown, JumpLeft, JumpRight }

If the car satisfies the conditions and can move into one of these directions, then the *action* is added to the list of *possible actions*. After going through all possibilities, the variable *self.consideredActions* is increased. This variable serves as to show the user how many lists of actions are taken into consideration, compared with what are the ones that are part of the final answer.

- *state* → due to the fact that it is a dynamic problem, we need a way to keep track of all the states of the cars in order to determine their possible actions. Thus, the variable containing these states needs to be updated and passed around.

(d) *result(*self*, state, action)*
Like the above function, this is also the implementation of the function with the same name from the class *Problem*. Its main scope is to return the state

that results from executing the given action in the given state. Hence, the two parameters are:

- *state* → contains a list of the states in which the objects can be found.
- *action* → the action from which we decide the outcome

(e) *heuristic(*self*, node)*
At each step we compute the heuristic for the given node and return its value.

- *node* → the current node from which we want to determine the heuristic.

III In *search.py*

class *Problem*

This is the abstract class upon which I based the class *MovingVehicles*. It contains all the necessary functions for this problem. The most notable functions, which have not been yet discussed are:

(a) *goal _test(*self*, state)*
This acts as a *bool* function, returning true only if the state is the same as the goal state.

- *state* → the state we want to check

(b) *path _cost(*self*, c, state1, action, state2)*
There was no need for me to modify this function, as it returns the proper result. In this particular case, the last three function parameters are not needed.

- *c* → the cost of the path so far.

class *Node*

This class is the structure of a single node inside a graph. A single node is characterized by the following parameters:

* *state* → the current state of the node
* *parent* → the parent of the current node
* *action* → the action that can be performed by the node
* *path_cost* → the total cost of the path
* *depth* → if the current node has a parent, then its depth is equal to the depth of the parent node plus one.

An important thing to node regarding this class is the fact that I have change the way the path is printed.

(a) *__repr __(*self*)*
This function returns a printable representational string of the given object. Thus, instead of printing the matrix as a list, it can now be printed as an actual matrix. This was done in order to facilitate the visual parsing of the solution.

IV *breadth _first _graph _search(problem)*
This, along with a few more functions, have been left inside the module in case we want to test the problem using different types of search algorithms.

This particular function searches the deepest nodes in the search tree first. Unlike *Breadth First Search* on a tree, this function does not get trapped in loops. Its performances can be seen in 2.3. This method brings the most efficient path in terms of cost, but it performs badly in terms of runtime, which is why I did not choose it as my search function.

V *depth _first _graph _search(problem)*
Just like the algorithm above, this one too, does not get caught in loops. It goes through the deepest nodes in the graph first, and then makes its way upwards. As seen in section 2.3, this algorithm performed badly on all the cases, so it cannot be used for this problem.

VI *best _first _graph _search(problem, f)*
This function is called by the *A* Search*, the only difference being in the use of functions. Because *A** uses a more complex system, it yields better results.

VII *astar _search(problem, f)*
This is the final search algorithm left in the module, and is also the one I choose for the problem, due to the complexity of its function *f*. Further details regarding both the algorithm and the pseudocode can be found in section 2.4.

4.3 Input data

To generate our input data we need only one number: the size of the parking lot, m. Because we will always now what the initial and goal states will look like, we can generate them inside the class constructor. Thus, the input read from file will be:

$$Input = \{m \quad | \quad m = \text{size of the parking lot}\}$$

4.4 Output data

The output is generated by the implemented algorithm, and placed in text file. I first chose to print the matrices corresponding to each step, so we can picture the way the cars are moving in the parking lot. Immediately after that, it will be printed the actions the cars take throughout the algorithm in order to reach their destination. The next two lines will contain the total cost of the path, and the number of all the actions the program considers during runtime. I chose to include this to show how complex the search becomes after just a few increases in the size of the matrix. The last line contains

the execution time of the program, which was needed, like the previous variable, to show how the algorithm behaves on different test cases.

The structure of the output will be:

$$Output = \begin{cases} ml, & ml = \text{list of matrices} \\ l, & l = \text{list of actions} \\ cost, & cost = \text{total cost of the algorithm} \\ actions, & actions = \text{actions taken into consideration} \\ time, & t = \text{execution time} \end{cases}$$

It should be noted that the list of matrices can only be printed in the console, because its print function is dependent on the *__repr__()* function. In order to make it write the data in a file, it would be needed to pass that file as parameter and that would change the functions unnecessarily.

5 Results

As we said before, this is a dynamic problem, meaning we have to take into consideration the state of each of the *m* cars. Due to its complex nature it is difficult to find an algorithm that will yield both the lowest cost for the path and the most efficient runtime. The main focus was upon finding a solution that could best approximate the final result, within a reasonable runtime.

Through trial and error, I have narrowed the best search algorithms to the four major ones: *Breadth First Search*, *Depth First Search*, *Best First Search* and *A* Search*. After running test and analyzing the results that can be found in section 2.3. Thus, as I mentioned previously, the algorithm capable of offering a middle ground between the two aspects was *A* Search*. It is true that it did not offer the best solutions, and that there is room for improvement. Unfortunately, as will be seen from the graph of the execution time, even for small numbers all the algorithms took a long time to complete and so my choices were limited.

In the image below there is an example of the console output for a case where the parking size is equal to three.

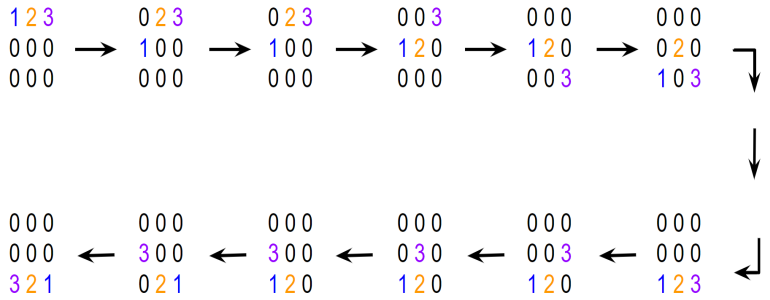


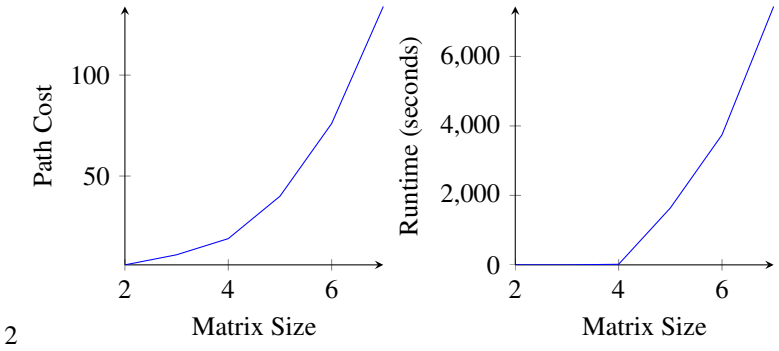
Figure 4: Example

From this example we can deduce that the algorithm is providing a near optimal solution. We can also see they way it takes into consideration all the other solutions and the states of the other cars. In addition to this, the function which determines the total cost of the path is correct and returns the proper value.

The tables below contain the actual data gathered after running test on the algorithm.

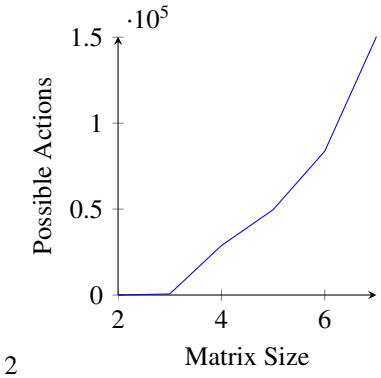
A* Search	Path Cost	Possible Actions	Runtime
m = 2	6	17	0.000009543
m = 3	11	629	0.00901937
m = 4	19	28774	16.64201235
m = 5	40	49635	1637.28562839
m = 6	76	83713	3743.27394039
m = 7	134	150342	7438.82732398

Using this data the following three graphs have been generated:



2

Figure 5: Path Cost and Runtime



2

6 Conclusions

The assignment I was handed proved to be challenging on a few aspects. At first it was a bit difficult to adapt my problem to the material provided in laboratory. After that I had to make sure that I had the right formulas when checking whether a car will be out of the matrix' bounds.

But the most notable I think was the fact that I could not provide the output for non-trivial numbers. Because the search became so complex due to the many actions it had to take into consideration, the number of test had to be limited to numbers smaller than ten. Yet, there is important information to gather and many conclusions to be taken even if the information is scarce.

All in all, although challenging, I learned a lot, both by researching and implementing different algorithms to see which would suit my problem better. I made significant progress and I am pleased with the fact that I managed to find an almost optimal solution to this problem.

References

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

[2] http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html, *Heuristics*, (accessed at May 29th 2021).

[3] https://en.wikipedia.org/wiki/A*_search_algorithm, *A* Search*, (accessed at May 29th 2021).

[4] https://isaaccomputerscience.org/concepts/dsa_search_a_star, *A* Search Algorithm*, (accessed at May 30th 2021).

[5] *problem _solving _framework.zip*, uploaded on Google Classroom *Artificial Intelligence D27CEL427*, on May 5th 2021

[6] https://www.overleaf.com/learn/latex/Tables, *Tables in LaTeX*, (accessed at June 4th 2021).

[7] https://www.overleaf.com/learn/latex/TikZ_package, *Graphics in LaTeX*, (accessed at June 4th 2021).