AI project 1

a) Crazy intersection

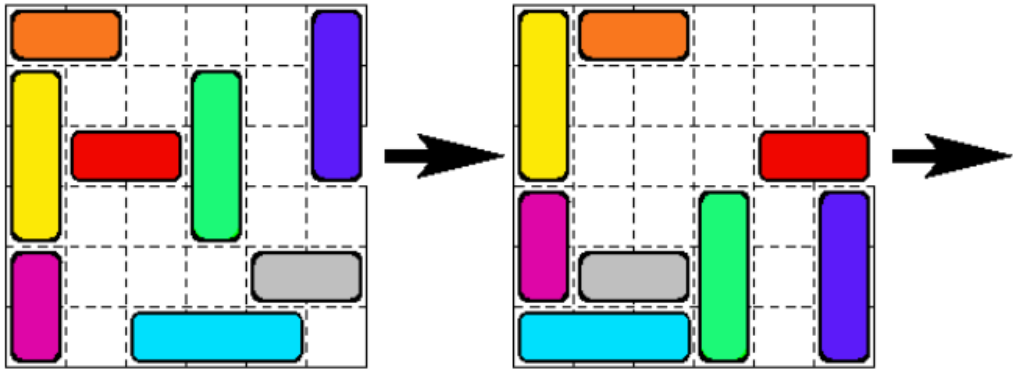
Márk Bartalos

Seminars with**: Ing. Martin Komák, PhD.**
Seminars on**: Monday 9:00-10:50**

Introduction

As task I had to create an algorithm that will find a solution for the crazy intersection problem. The algorithm had to use Breath for search (BFS) and Dept first search methods (DFS) to find the solution. Each car had only one orientation, be it VERTICAL or HORIZONTAL.

When a correct solution was found the algorithm had to generate a series of moves that led to the cars being in the correct configuration.



Used data structures

Car class

A car class represents a car in the map, with its own position (x,y), orientation, color and Id. Each car has a unique id which is used to differentiate between them during search operations.

Car
id: int position: Position(x,y) color: string orientation: Orientation(H | V)

On movement the cars position is updated accordingly. The orientation of the car dictates the possible directions the car can move. If the orientation is horizontal than the car can only move left or right, if the orientation is vertical, the car can move only up and down. As the game rules defined. Color and orientation are only set at the creation of the car and are not modified through the algorithm.

Map class

Map class represents the configuration of the cars on the map. The configuration is saved in a 2D array which is the size of the map. Each car in the map is marked with its Id. The empty fields are marked with null. The carList property contains reference to all the cars on the map.

If a car is moved it is moved on the map (meaning in the values array) and the car itself is moved, by changing its position.

Map
values: string[][] width: int height: int carList: List<Car>

On every move a new instance of the map is created and returned.

```
var newMap = this.DeepClone();
```

Also every map contains a GetHash() method that return the hash of the current configuration of the cars, by hashing the “values” array.

CarDecisionNode

One of the most important data structures for this algorithm. Using this node, we well create the solution tree for the problem.

CarDecisionNode
direction: int depth: int car: Car mapConfiguration: Map children: List<CardDecisionNode> parent: CardDecisionNode

The direction property signifies the last move that got the mapConfiguration to this configuration. The children are the list of child nodes. The parent is the parent of the node, this is set automatically for each child when added.

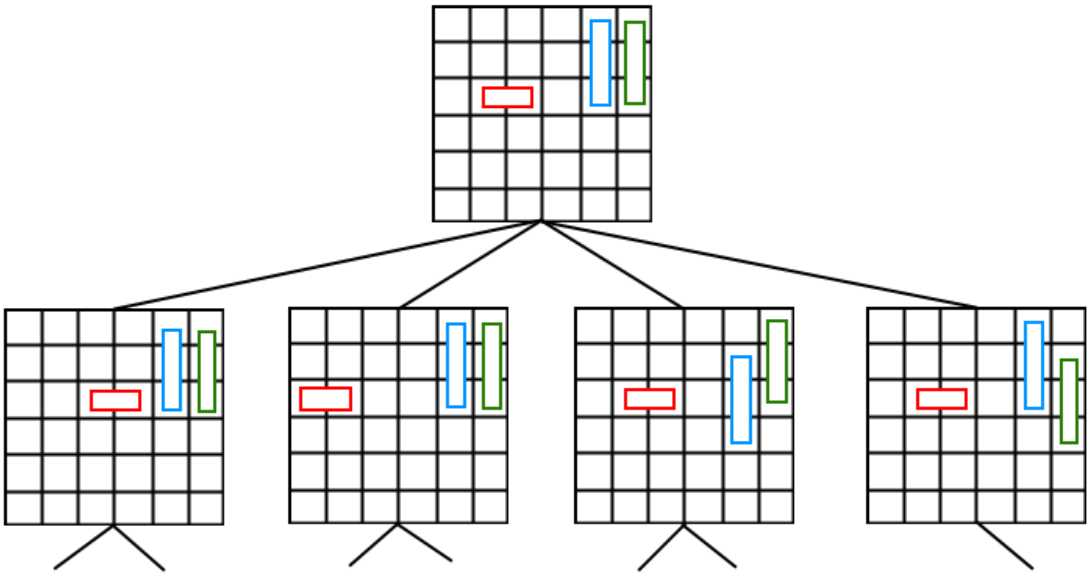
Main part of the algorithm

Breath first search method (BFS)

Breath first search method is the method of first generating all the child nodes and only then recursing to the newly generated nodes. For this solution a Queue (First in first out) data structure was used, which was built in the .Net framework. With this method the generation of the nodes was the following:

- Start with the start configuration
- Try to move each car either in positive direction or negative direction

    - If it could be moved add the moved configuration as children to the current node and add to the queue, **check if red car is in the finish position**
    - Else go to the next car
- If all cars have been checked then get the first node from the queue, set it as the current node and go back to step 2. Try this until the correct configuration is found or all possibilities has been tried.



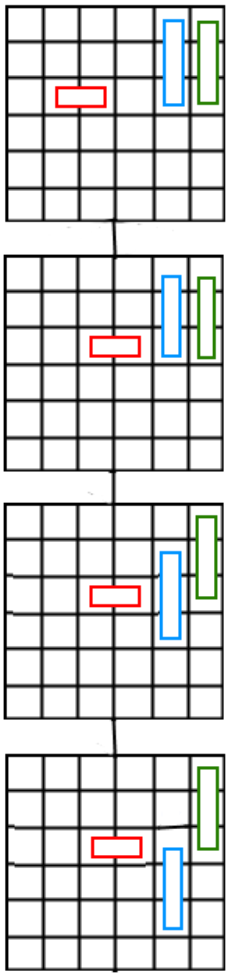
If the correct solution is found the algorithm will print out the steps required for solving the problem.

Depth first search method (DFS)

The depth first search is in opposition to BFS based on only selecting one node and going deeper until we find the correct solution. If the correct solution isn't found in the current branch we go back to the parent and try to find the solution from there.

Generation of the nodes:

- Start with the start configuration
- Try to move a car in either direction
    - If the car has been moved, **check if red car is in the finish position,** if not add the moved configuration as children to the current node and go back to step 2. and set the current node to this node.
        - Else go to the next car
    - If no car could not be moved than set the current node to the current nodes parent and go to step 2.



Here no additional data structures were used as in case of the BFS method, the only requirement was to be able to go back to the parent of the node. Same with the BFS method the algorithm will print out the required steps to solve the problem.

Generating the steps for the solution

The steps for the correct solution are generated by traversing back to the root node from the solution node. The BFS and DFS methods both return either a node which contains the correct solution or they return null. If the result is null than there is no possible solution to the given problem. If we get a node that is not null, then we can trace back the steps through going through node's ancestors. Each node contains the car that was moved, the direction (and the car also contains its orientation) so that way we get a list of steps from the solution back to the original position. Then the steps are summarized and reversed, all done by using a Stack data structure and creating the steps as Step objects, which contains the step direction, the color of the car and the actual step count. Every time the same step (with the same car) comes as the previous, the step counter is incremented.

Testing the solutions

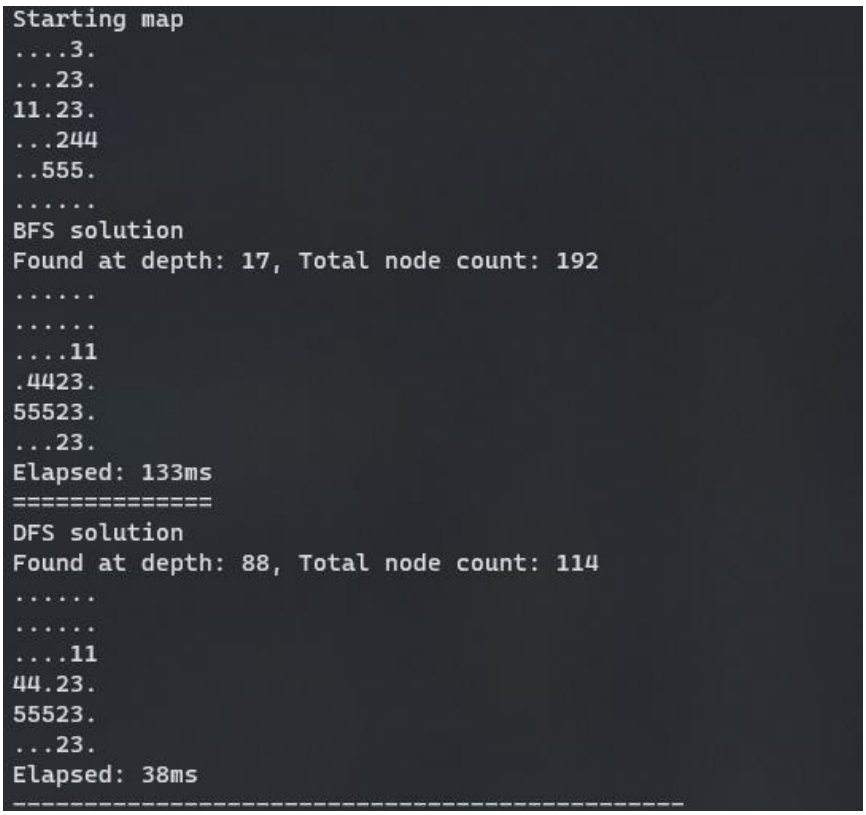
For testing the solutions, I used two methods. I measured the depth of the three and kept count of how many nodes are generated, and the other is that I also measured how long it takes to solve each problem with each method. Test were created for maps in the format given in the instructions of the problem.

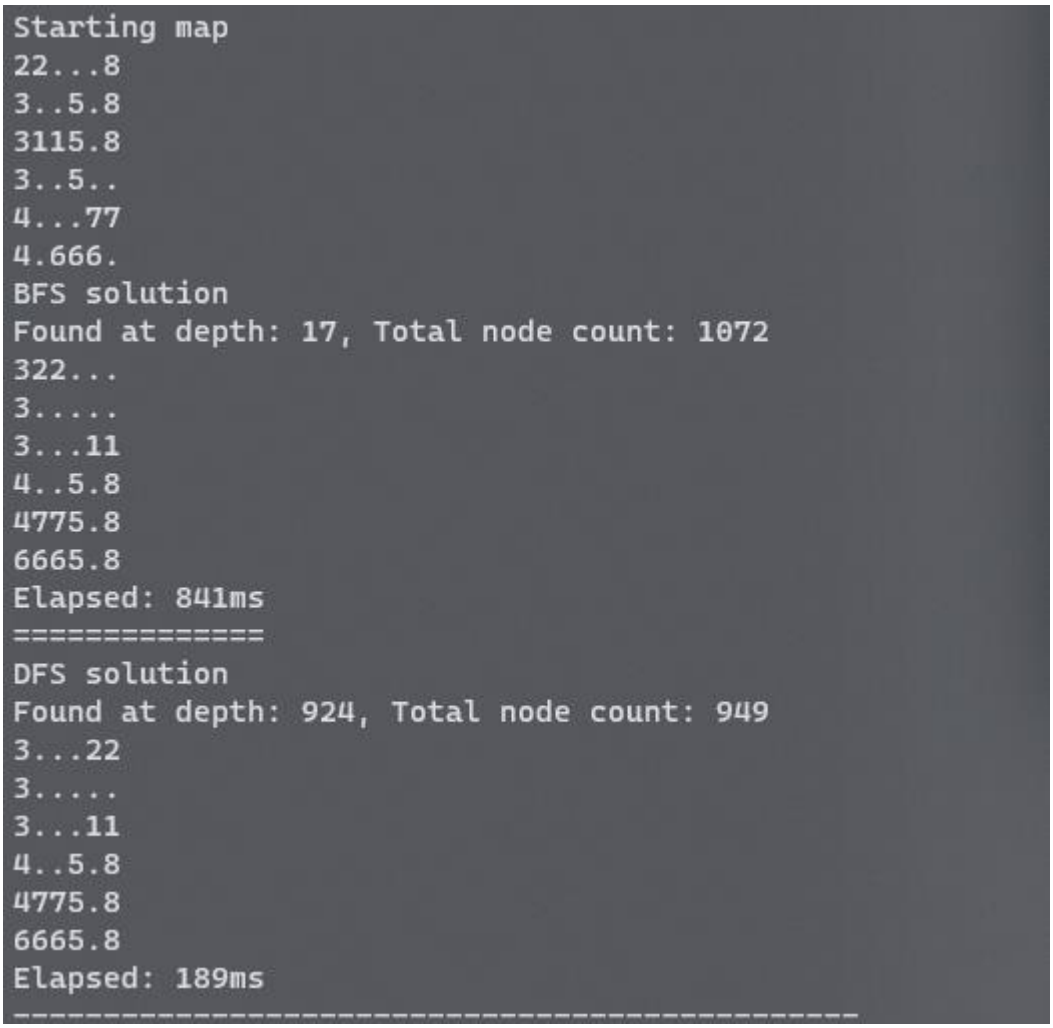
For example:

```
((cervene 2 3 2 h)(oranzove 2 1 1 h)(zlte 3 2 1 v)(fialove 2 5 1 v)(zelene 3 2 4
v)(svetlomodre 3 6 3 h)(sive 2 5 5 h)(tmavomodre 3 1 6 v))
```

The result of the algorithm was:

```
VPRAVO(oranzove 1)
HORE(zlte 1)
HORE(fialove 1)
VLAVO(svetlomodre 2)
VLAVO(sive 3)
DOLE(zelene 2)
VPRAVO(cervene 2)
DOLE(tmavomodre 3)
VPRAVO(cervene 1)
```





Results

As you can see on the images the correct map has been found in both cases and printed to the screen. After each map there is a time, it took in ms (milliseconds). At the top of each solution, you can also see the depth of the tree and how many nodes were generated in total. The depth also means how many steps it takes to reach the solution***. If we compare BFS and DFS in both cases the BFS took much longer, the node count was also slightly higher but the number of steps to solve the problem is much smaller. BFS is much more optimal if we need to find the shortest path, and DFS is the more appropriate option if we want to compute the result faster and the step count doesn't matter much.***

Map	Method	Depth	Total nodes	Time (ms)
Configuration 1	BFS	17	192	133
Configuration 1	DFS	88	114	38
Configuration 2	BFS	17	1072	841
Configuration 2	DFS	924	949	189

Optimalizations used

Step counting

Both the DFS and the BFS methods return a solution tree consisting of single steps only. I optimized them using a Step object and a Stack, which groups same steps that are after each other together.

Possible optimalization for DFS

The DFS method most of the time generates much longer step tree than the BFS method. That is due to DFS always first tries to go deeper until a correct solution is found. That is why some unnecessary steps are included in the solution. These possibly could be reduced by taking the starting configuration and reducing then unnecessary steps to reach the goal.