|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Assembler Programming Languages - Project | | | | | | |
| Academic year | Project meetings on | Mode of studies | Field of studies | Supervisor | Group | Section |
| 2020/2021 | Thursday | SSI | Informatics | PCz | 1 | 2 |
| 10:30-12:00 |

Project Report

Topic: Finding the shortest path in a maze using Dijkstra algorithm.

|  |
| --- |
| Performed by:  Łukasz Kwiecień |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

# Topic

The topic of the project is the use of the Dijkstra algorithm to find the shortest path in a maze.

# Assumptions

The user selects the source and destination points on the grid, it is also possible to add walls to create a maze. Then he clicks the start button, which calls functions from the DLL library (ASM or C# implementation) with the implementation of the Dijkstra algorithm, after the algorithm is executed, the path is visualized on the grid by changing the colors of the appropriate cells.

# Input parameters of the program

User can provide the input data in two ways:

1. Draw by himself the maze on the 40x20 grid using provided tools in the GUI.
2. Load the prepared maze (max size 40x20) from the txt file.

Maze will be saved in a custom Grid object which contains an array of Cell objects. Cell object contains x and y coordinates and the type of the cell.

There are 7 types of cells: invalid, solid (wall), empty, A (start), B (end), path and visited. For visualization purposes every type has its own color.

Grid object is only used to visualize the solution, both ASM and .NET implementations of the algorithms take one dimensional arrays of integers as parameters. When the solution is to be visualized, there is converter which takes the index of the cell from the one dimensional array and returns the corresponding cell from the grid. Then the type of the cell from the grid can be changed, and the path can be visualized.

Input parameters of the algorithm implementations:

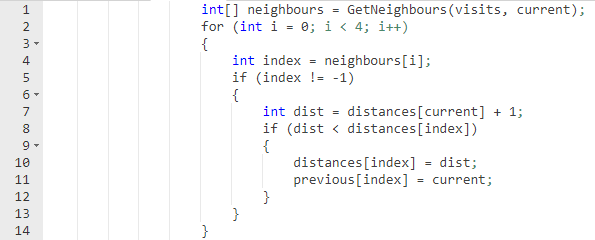
* Int [800] distances – an array with distances
* Bool [800] visited – an array with information if the cell was visited
* Int [800] previous – an array with the indexes of the previous cell in the path to each cell
* Int source – index of the source cell – cell of type A in the grid
* Int destination – index of the destination cell – cell of type B in the grid
* Int len – length of the array – 800 because the grid size is 40x20

The arrays are empty (except the visited array, cells which are walls in the maze are marked as visited before calling the algorithm because the information about the type is obtained from the grid object) and they are filled with values by the algorithm. Then the shortest path can be obtained from the previous cells array. We take the destination cell and proceed to its predecessor until we reach the source.

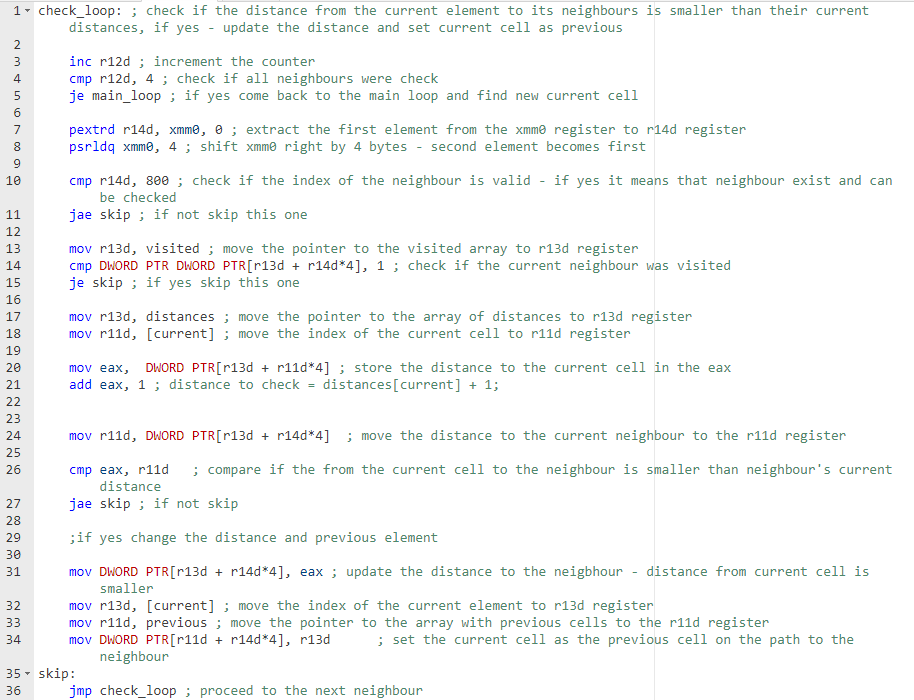
# Description of the selected piece of assembly language DLL

I want to show the part of the code responsible for checking if the distance to the neighbour through the current node is smaller than the current distance assigned to the neighbour.

C# version of the code:



ASM:



This part of the code contains two parts: the check\_loop and skip.

The loop as I mentioned before is responsible for checking the distances.

Variables and registers used in the code:

|  |  |
| --- | --- |
| visited | Contains pointer to the “visited” array |
| distances | Contains pointer to the “distances” array |
| current | Contains index of the current element |
| previous | Contains pointer to the “previous” array |
| r11d | Used to store the index of the current element |
| r12d | Counter |
| r13d | Used to access the arrays |
| r14d | Index of the currently considered neighbour |
| xmm0 | Holds indexes of 4 neigbhours |
| eax | Used to calculate and store the “new” distance |

In lines 3,4 and 5 the counter is incremented and checked that the loop has been run 4 times.

After that the neighbour is extracted from the xmm0 register (line 7) and the xmm0 is shifted to the right in order to extract the next neighbour during the next iteration (line 8).

The next step is to check if the value of the neighbour (index of the element) is valid (in a range between 0-799), if it is not valid jump to skip and proceed to the next neighbour.

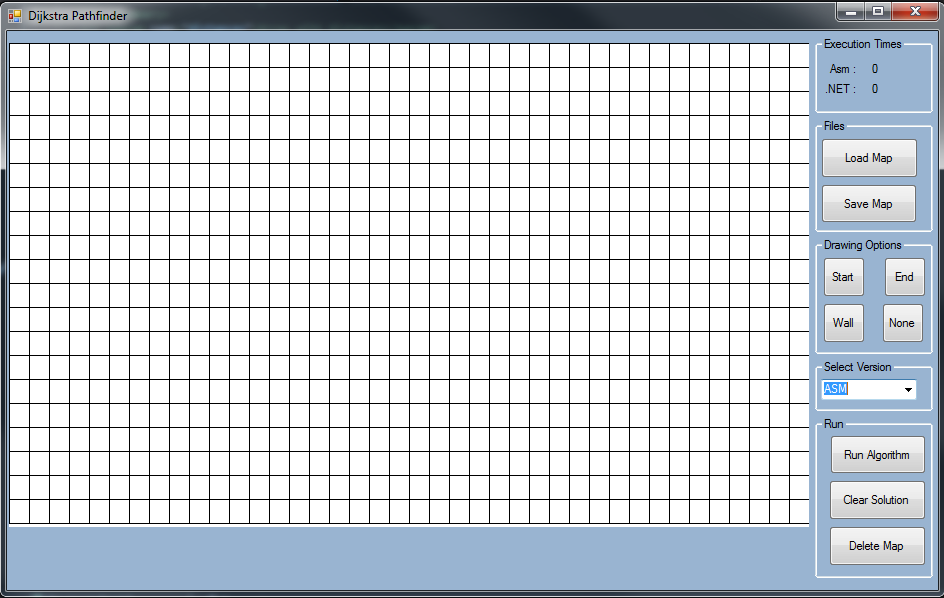
If the neighbour is valid we have to check if the neighbour was visited already. We can only consider unvisited neighbours of the current element.

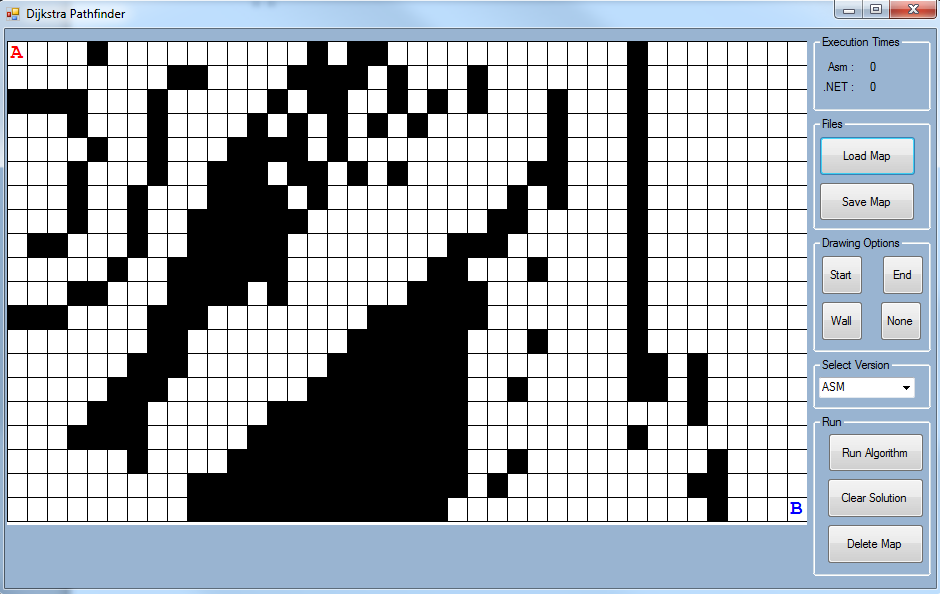
Then in lines 17-21 the “new” distance to check is calculated by getting from the array of distances value for the current element and incrementing it by one (distance between every cell in the maze is equal to 1). After the calculations the distance is stored in the eax register.

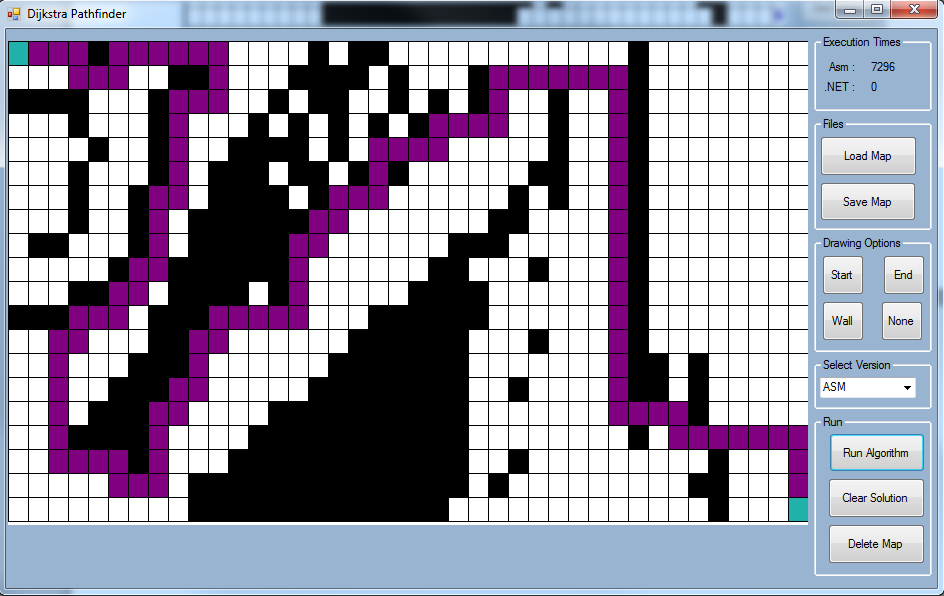
The current distance to the neighbour is stored in r11d register (line 24) and compared with the distance stored in eax register. If the new distance is smaller we have to update the current distance to the neighbour and its predecessor.

The update is performed in lines 31-34.

# Appearance of the User Interface



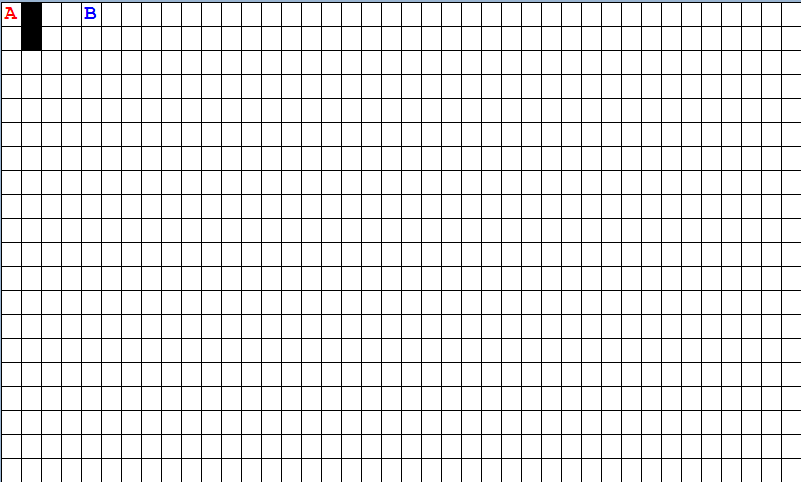




# Speed of the execution

For measurements I used 6 inputs (all of them can be found in the inputs folder) with different levels of complexity. Time of execution was measured in processor’s ticks.

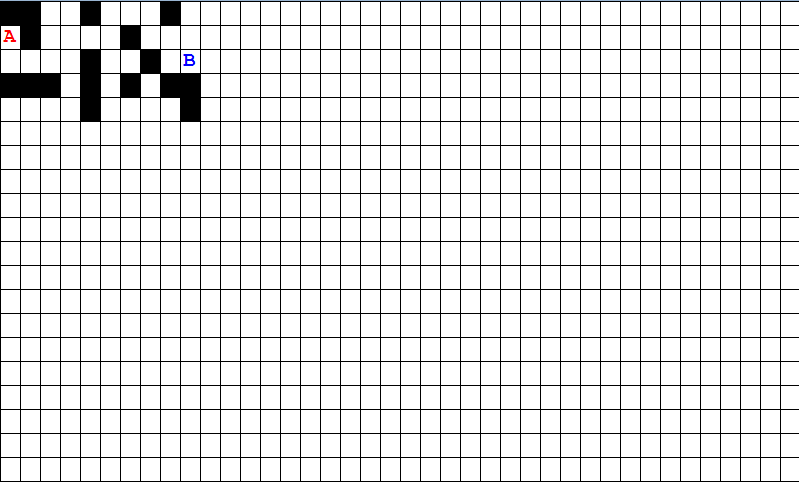
**Input 1:**



Speed of the execution (CPU ticks) – input 1

|  |  |  |
| --- | --- | --- |
| Attempt | ASM | C# |
| 1 | 136 | 727 |
| 2 | 270 | 469 |
| 3 | 197 | 529 |
| 4 | 203 | 460 |
| 5 | 106 | 534 |
| Average | 182.4 | 543.8 |

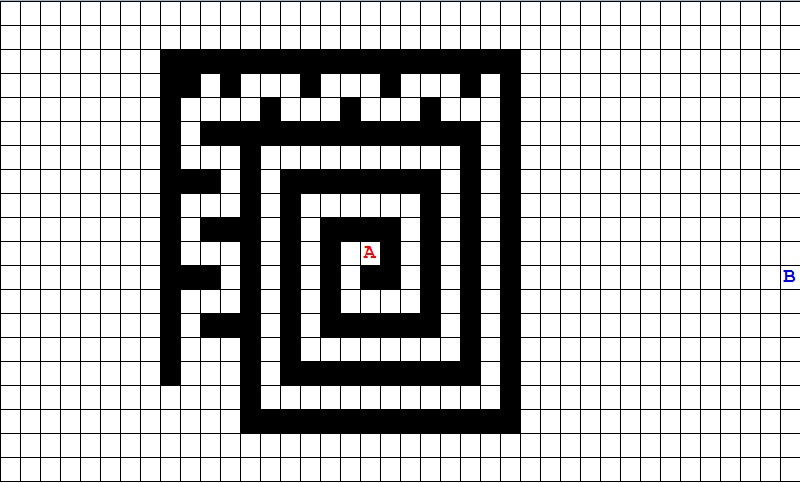
**Input 2:**



Speed of the execution (CPU ticks) – input 2

|  |  |  |
| --- | --- | --- |
| Attempt | ASM | C# |
| 1 | 303 | 943 |
| 2 | 306 | 949 |
| 3 | 296 | 679 |
| 4 | 515 | 704 |
| 5 | 297 | 681 |
| Average | 343.4 | 791.2 |

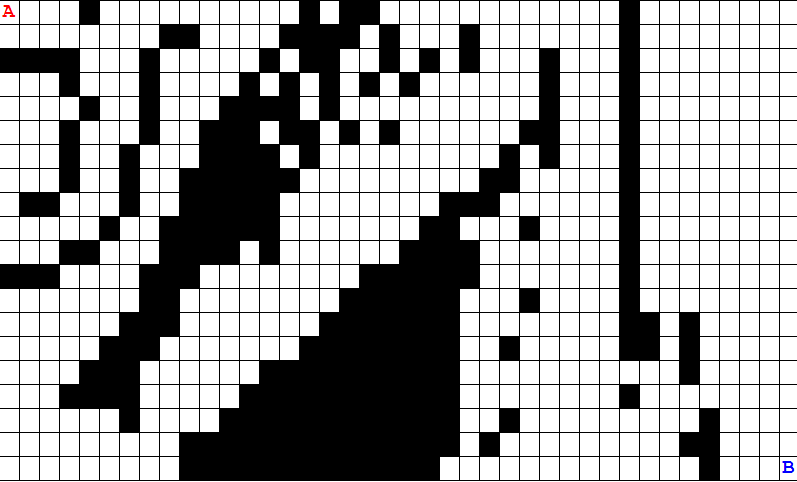
**Input 3:**



Speed of the execution (CPU ticks) – input 3

|  |  |  |
| --- | --- | --- |
| Attempt | ASM | C# |
| 1 | 3752 | 4795 |
| 2 | 2848 | 5550 |
| 3 | 2847 | 3345 |
| 4 | 3883 | 6908 |
| 5 | 4441 | 3732 |
| Average | 3554.2 | 4866 |

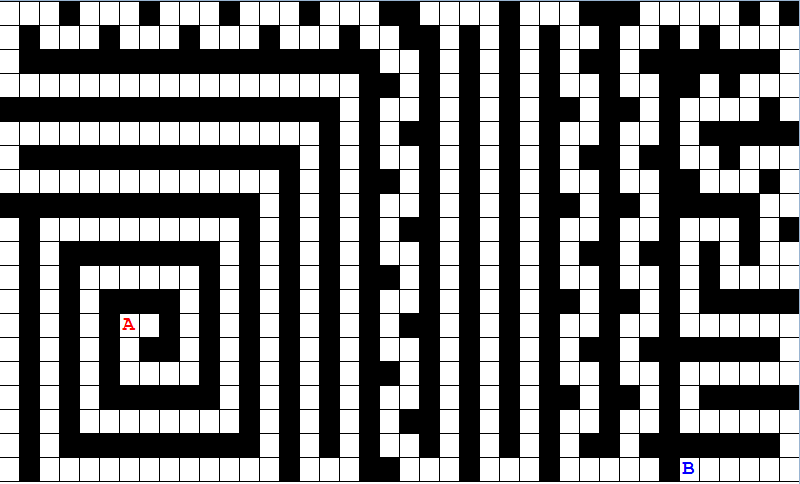
**Input 4:**



Speed of the execution (CPU ticks) – input 4

|  |  |  |
| --- | --- | --- |
| Attempt | ASM | C# |
| 1 | 3824 | 4189 |
| 2 | 6130 | 3784 |
| 3 | 3583 | 3782 |
| 4 | 5708 | 3677 |
| 5 | 5015 | 3751 |
| Average | 4852 | 3836.6 |

**Input 5:**



Speed of the execution (CPU ticks) – input 5

|  |  |  |
| --- | --- | --- |
| Attempt | ASM | C# |
| 1 | 3093 | 3249 |
| 2 | 3564 | 3072 |
| 3 | 5837 | 3030 |
| 4 | 2867 | 4423 |
| 5 | 3028 | 3124 |
| Average | 3677.8 | 3379.6 |

**Averages for every input:**

|  |  |  |
| --- | --- | --- |
| Input | ASM avg | C# avg |
| 1 | 182.4 | 543.8 |
| 2 | 343.4 | 791.2 |
| 3 | 3554.2 | 4866 |
| 4 | 4852 | 3836.6 |
| 5 | 3677.8 | 3379.6 |

# Description of testing and debugging of the program

# Conclusions