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

Project Report

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

Performed by:
Łukasz Kwiecień

1. Topic

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

2. 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.

3. Input parameters of the program

User can provide the input data in two ways:

a) Draw by himself the maze on the 40x20 grid using provided tools in the GUI.
b) 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 predecessors
- 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.

4. 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:

```
int[] neighbours = GetNeighbours(visits, current);
for (int i = 0; i < 4; i++)
{
    int index = neighbours[i];
    if (index != -1)
    {
        int dist = distances[current] + 1;
        if (dist < distances[index])
        {
            distances[index] = dist;
            previous[index] = current;
        }
    }
}
```

ASM:

```
check_loop: ; check if the distance from the current element to its neighbours is smaller than their current distances, if yes - update the distance and set current cell as previous

    inc r12d ; increment the counter
    cmp r12d, 4 ; check if all neighbours were check
    je main_loop ; if yes come back to the main loop and find new current cell

    pextrd r14d, xmm0, 0 ; extract the first element from the xmm0 register to r14d register
    psrldq xmm0, 4 ; shift xmm0 right by 4 bytes - second element becomes first

    cmp r14d, 800 ; check if the index of the neighbour is valid - if yes it means that neighbour exist and can be checked
    jae skip ; if not skip this one

    mov r13d, visited ; move the pointer to the visited array to r13d register
    cmp DWORD PTR DWORD PTR[r13d + r14d*4], 1 ; check if the current neighbour was visited
    je skip ; if yes skip this one

    mov r13d, distances ; move the pointer to the array of distances to r13d register
    mov r11d, [current] ; move the index of the current cell to r11d register

    mov eax,  DWORD PTR[r13d + r11d*4] ; store the distance to the current cell in the eax
    add eax, 1 ; distance to check = distances[current] + 1;


    mov r11d, DWORD PTR[r13d + r14d*4]  ; move the distance to the current neighbour to the r11d register

    cmp eax, r11d   ; compare if the from the current cell to the neighbour is smaller than neighbour's current distance
    jae skip ; if not skip

    ;if yes change the distance and previous element

    mov DWORD PTR[r13d + r14d*4], eax ; update the distance to the neigbhour - distance from current cell is smaller
    mov r13d, [current] ; move the index of the current element to r13d register
    mov r11d, previous ; move the pointer to the array with previous cells to the r11d register
    mov DWORD PTR[r11d + r14d*4], r13d      ; set the current cell as the previous cell on the path to the neighbour
skip:
    jmp check_loop ; proceed to the next neighbour
```

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).

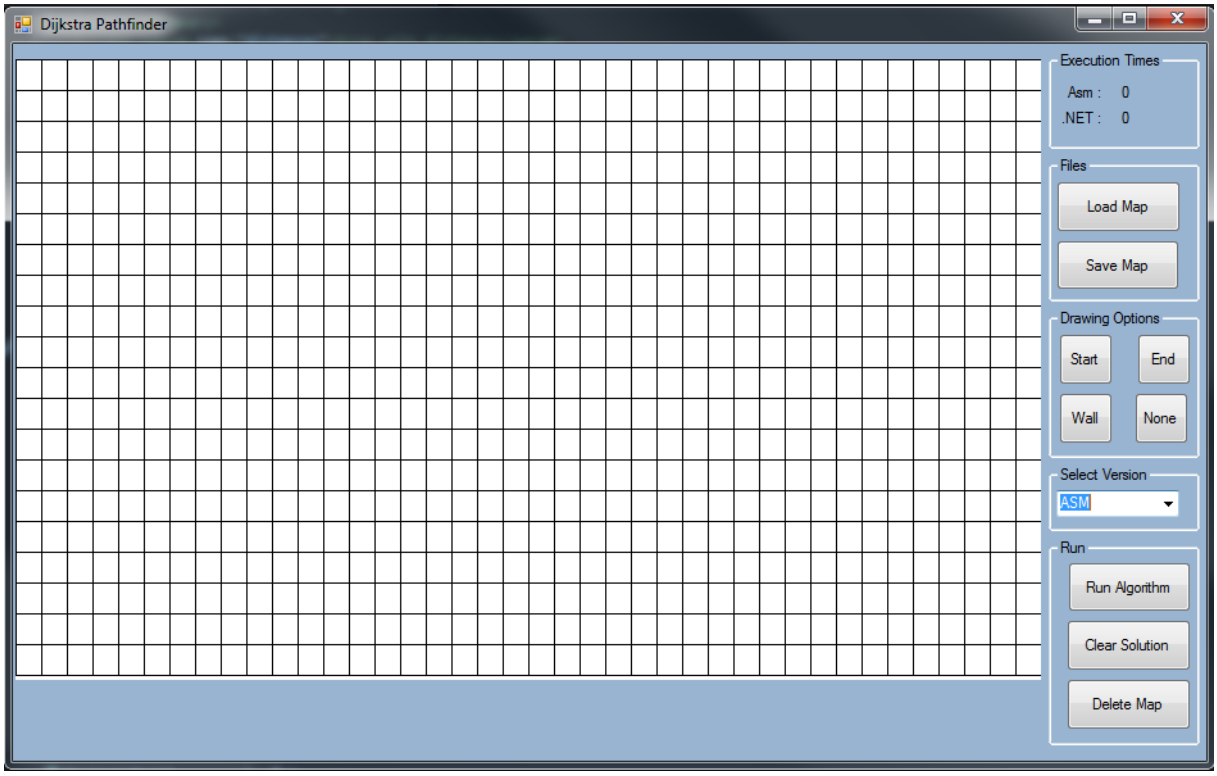
The next step is to check if the value of the neighbour (index of the element) is valid (in a range between 0-799), if it is invalid jump to skip and proceed to the next neighbour.

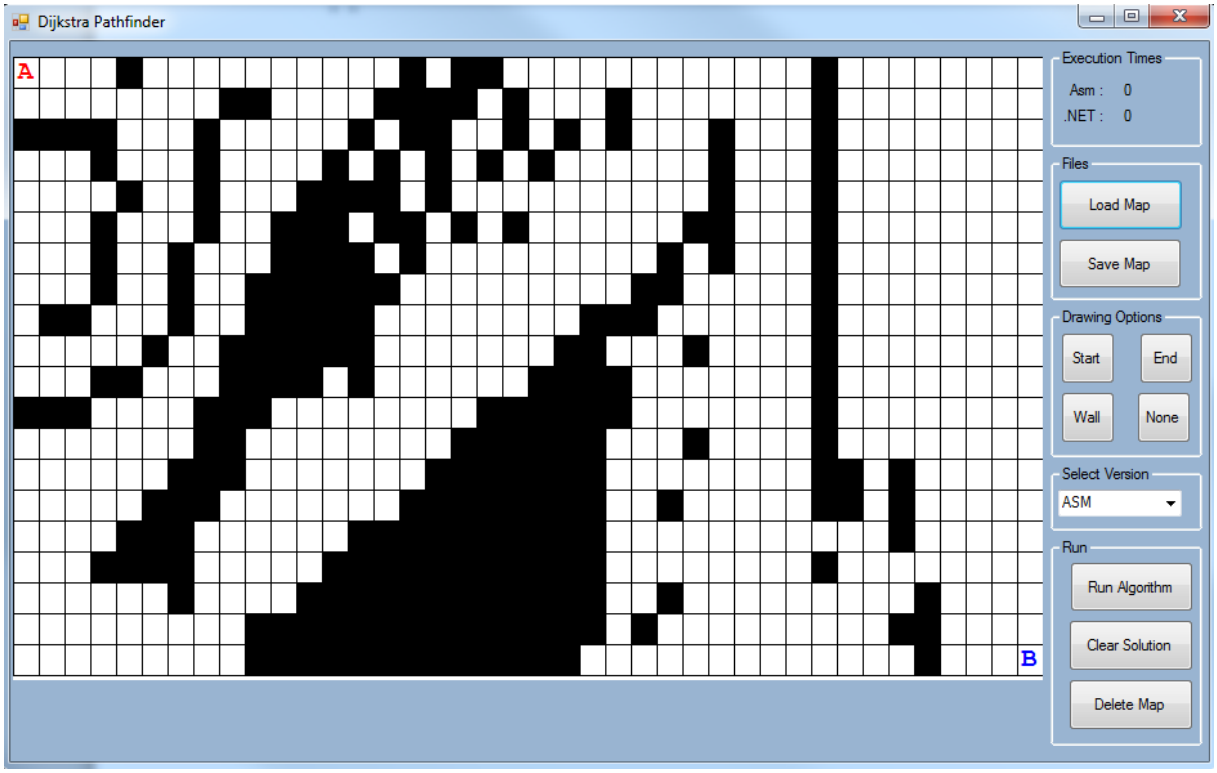
If the neighbour is valid we have to check that the neighbour was already visited. We can only consider unvisited neighbours of the current element.

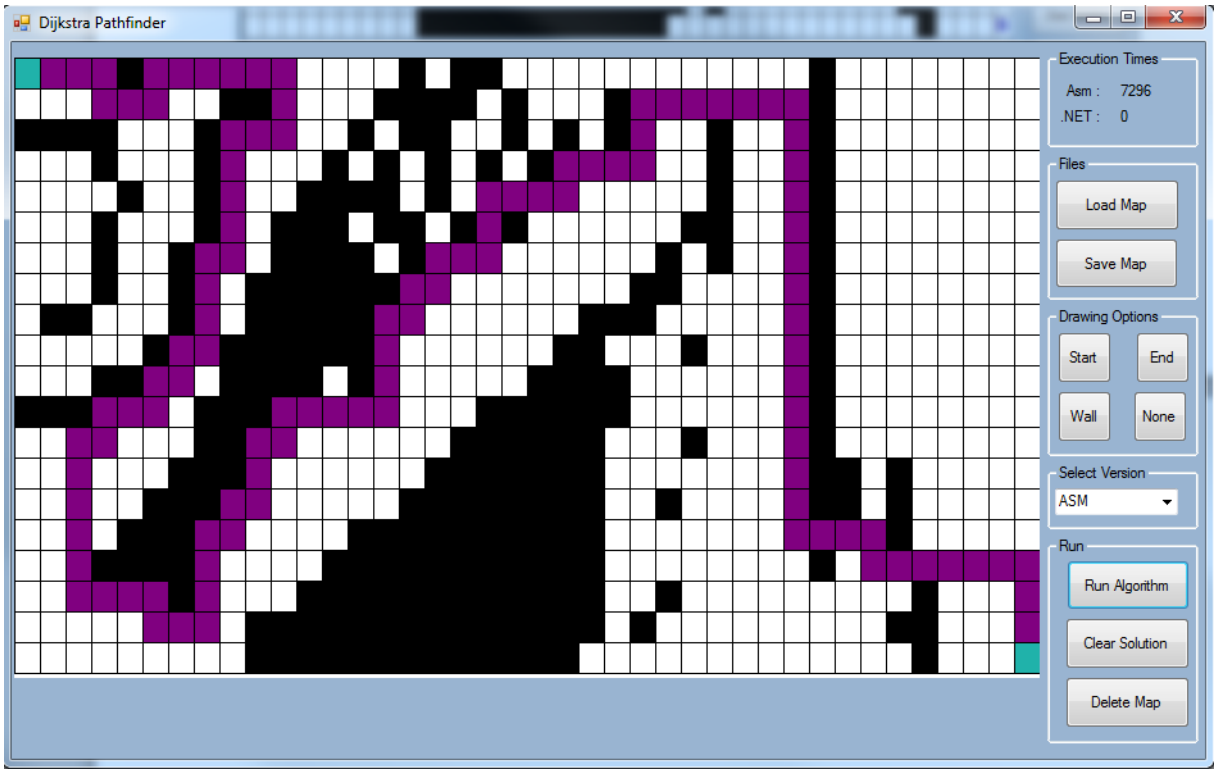
Then in lines 17-21 the “new” distance to check is calculated by getting from the array of distances value for the current element and incrementing it by one (distance between every cell in the maze is equal to 1). After the calculations the distance is stored in the eax register.

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.

5. Appearance of the User Interface



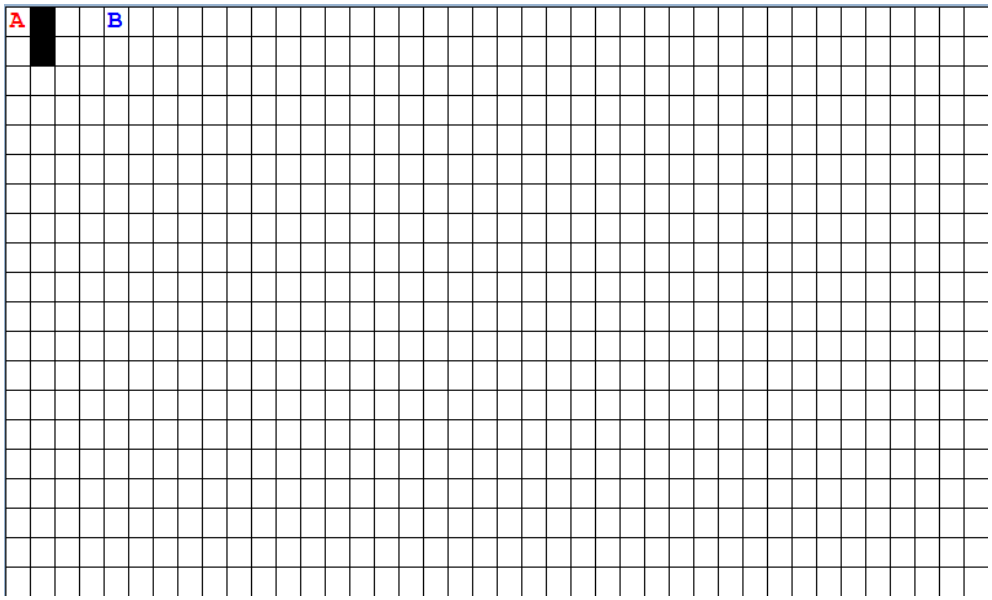



Dijkstra Pathfinder
Execution Times
Asm : 7296
.NET : 0
Files
Load Map
Save Map
Drawing Options
Start
End
Wall
None
Select Version
ASM
Run
Run Algorithm
Clear Solution
Delete Map

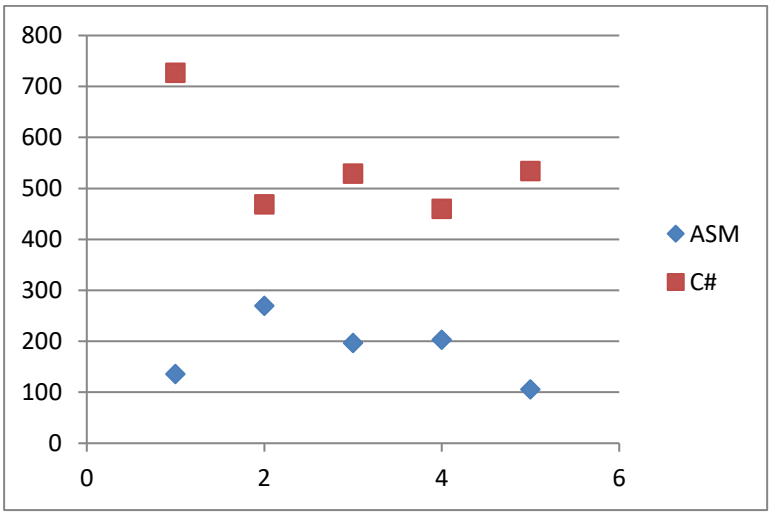
6. Speed of the execution

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 CPU ticks.

Input 1:

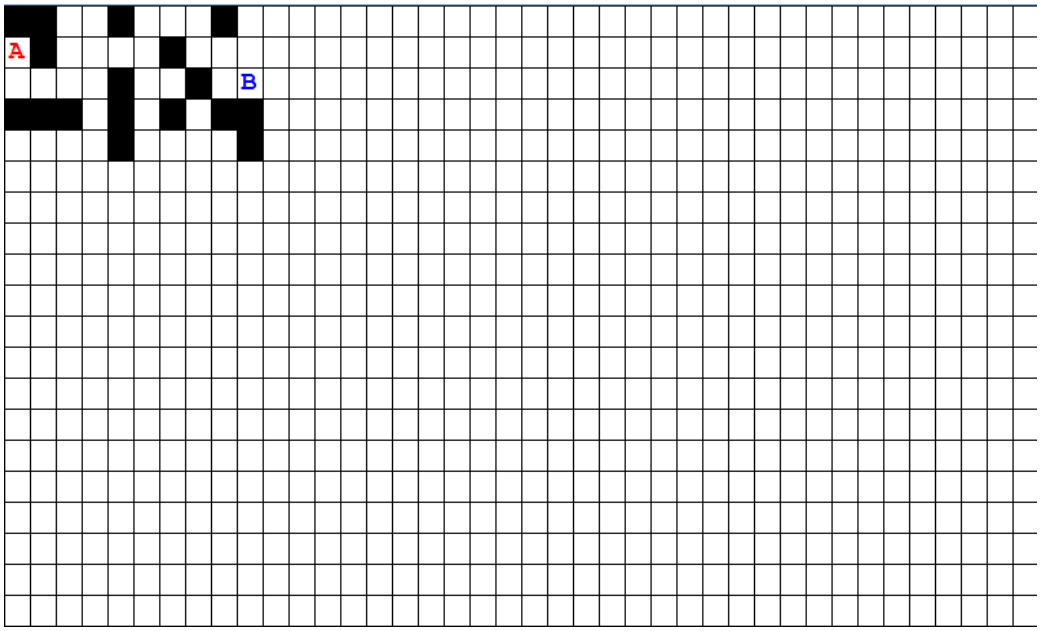


Speed of the execution (CPU ticks) – input 1

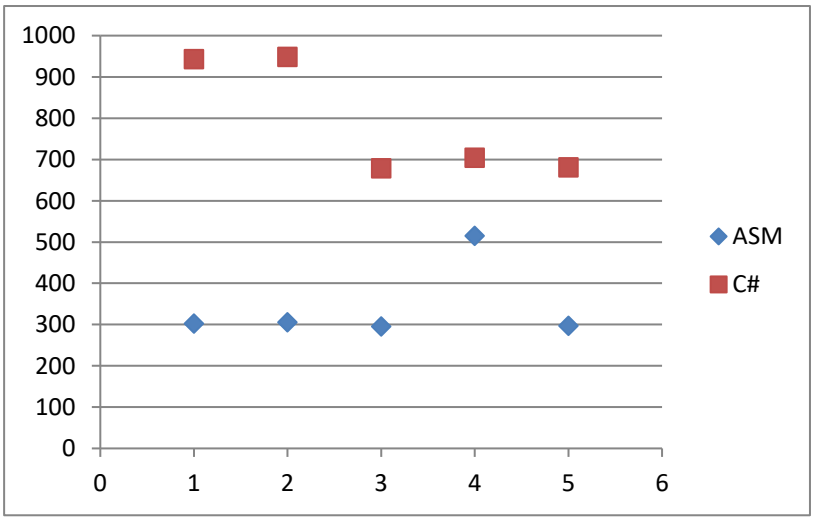


Attempt	ASM	C#
1	136	727
2	270	469
3	197	529
4	203	460
5	106	534
Average	182.4	543.8

Input 2:

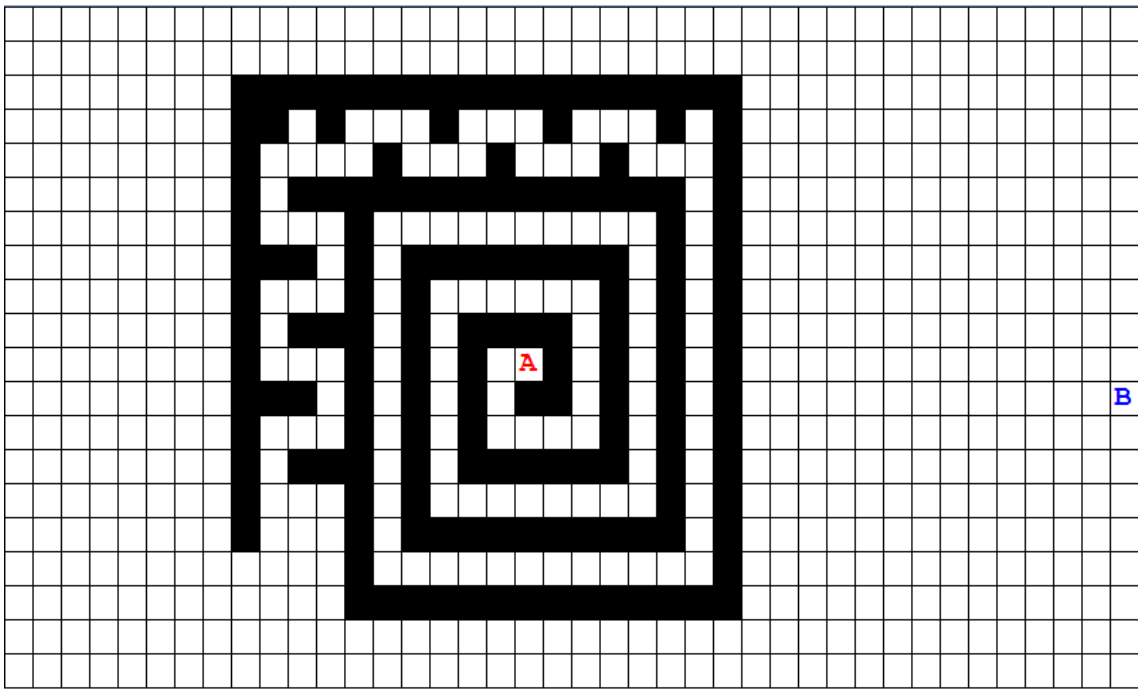


Speed of the execution (CPU ticks) – input 2

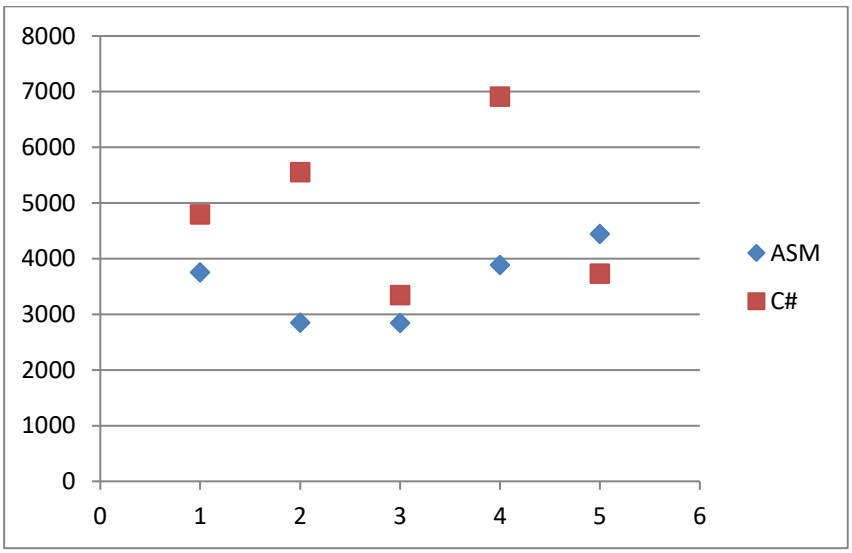


Attempt	ASM	C#
1	303	943
2	306	949
3	296	679
4	515	704
5	297	681
Average	343.4	791.2

Input 3:

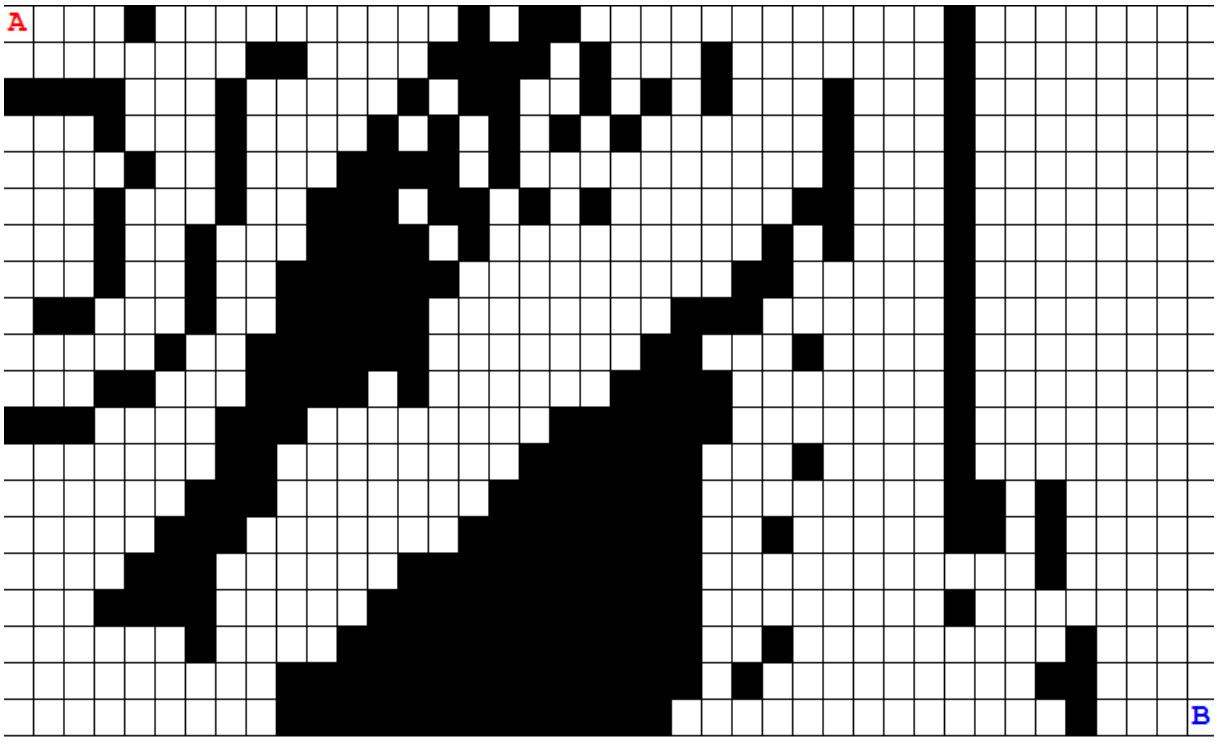


Speed of the execution (CPU ticks) – input 3

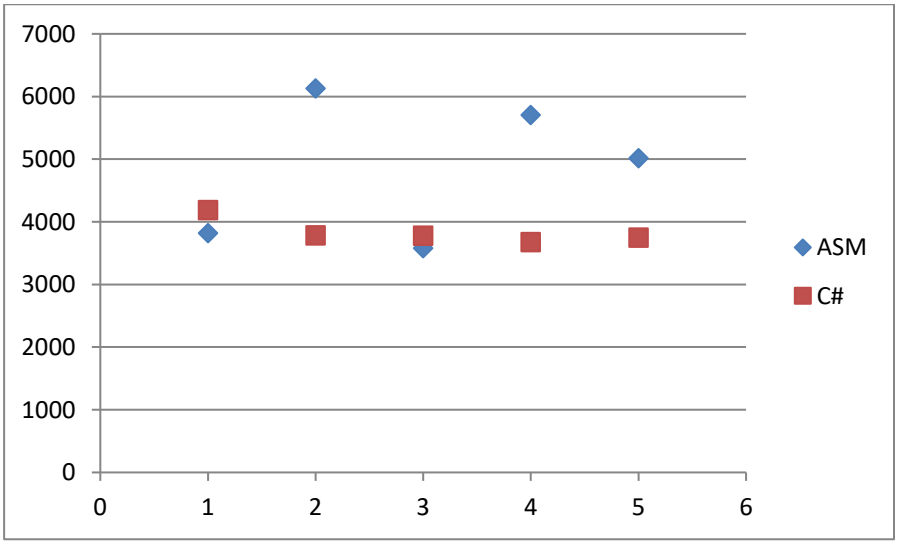


Attempt	ASM	C#
1	3752	4795
2	2848	5550
3	2847	3345
4	3883	6908
5	4441	3732
Average	3554.2	4866

Input 4:

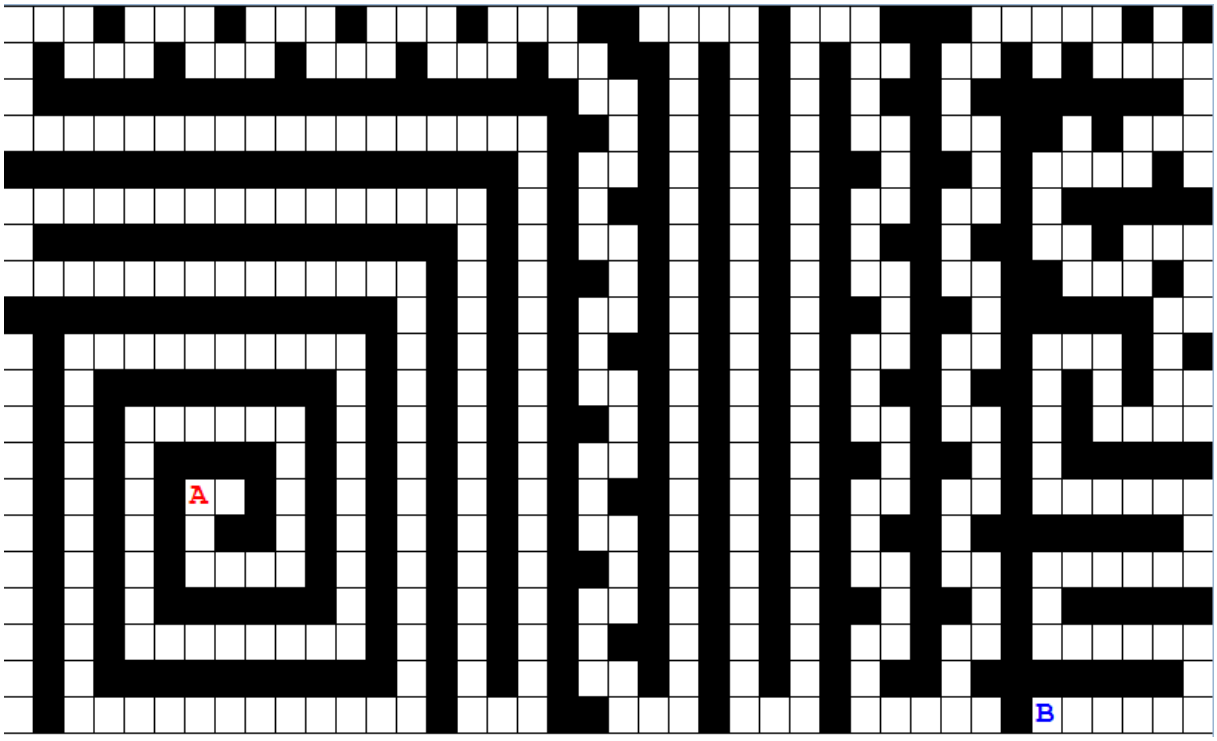


Speed of the execution (CPU ticks) – input 4

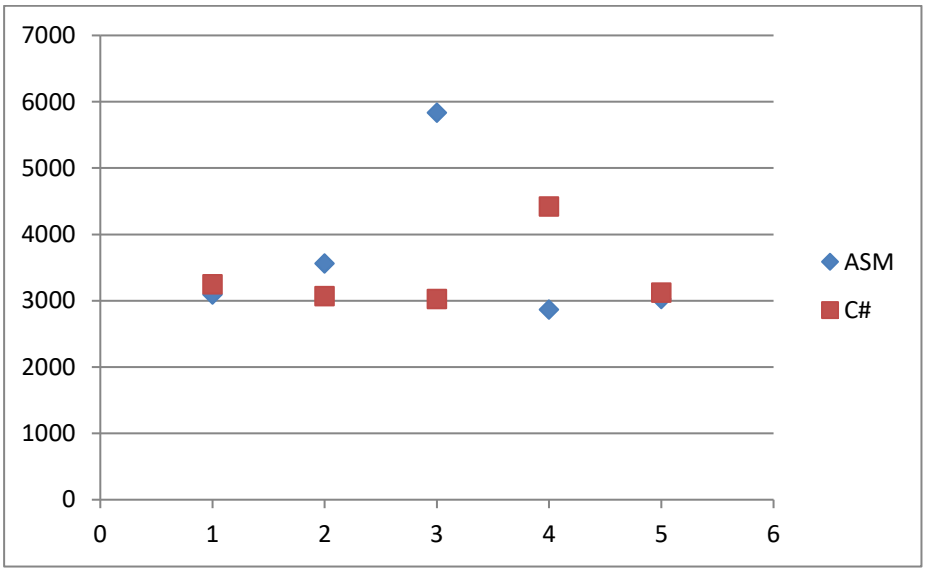


Attempt	ASM	C#
1	3824	4189
2	6130	3784
3	3583	3782
4	5708	3677
5	5015	3751
Average	4852	3836.6

Input 5:

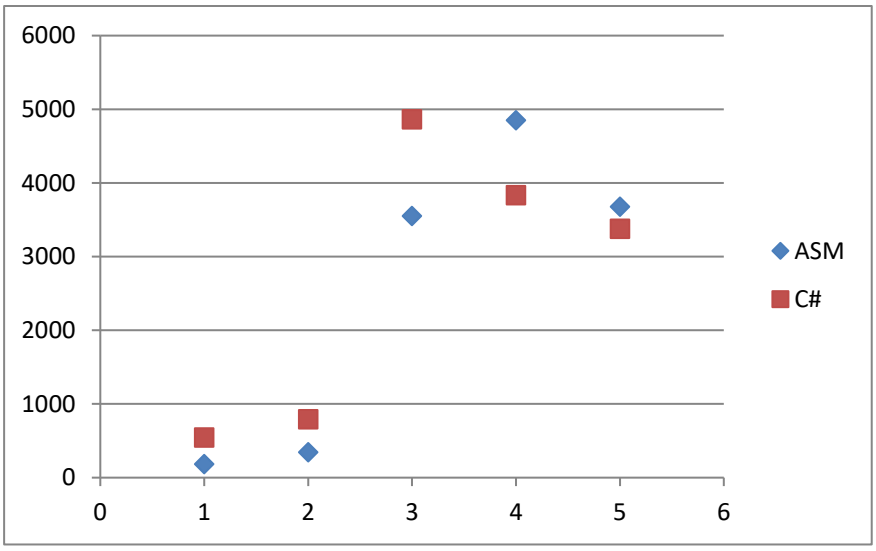


Speed of the execution (CPU ticks) – input 5



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

7. Description of testing and debugging of the program

Program was tested using various inputs with different levels of complexity – from txt to ones drawn manually in the GUI. Both C# and ASM versions of the libraries were tested and debugged during the development process.

Asm procedure was debugged several times during the implementation process. I was writing the algorithm step by step so I debugged the procedure line by line after each part, checking that every register contained the correct values.

To check that the algorithm finds the correct path I prepared inputs where the shortest path was very simple and easy to verify (e.g. straight line).

8. Conclusions

The C# part of the project was not so complicated. Both the GUI and the C# implementation of the algorithm were fairly easy to implement. My first version of the C# implementation used custom Objects but then I decided to rewrite it to use the simplest structures and types possible to make the ASM implementation easier. I wrote the ASM version following the C# code step by step, so that both versions would perform in a similar way.

The hardest part of the project was to parallelize the ASM version of the algorithm due to the serial nature of the algorithm. I ended up using the SIMD instructions in the initialization of arrays, because the entire arrays are filled with the same initial value, I could parallelize that part of the algorithm. I tried to parallelize the get_minimum macro, but the parallelized version would have more operations than the serial one (only unvisited cells can be considered) so I chose not to implement it as it would slow down my algorithm execution time.

Charts and tables in point 6 give us interesting conclusions. As we can see, when the level of complexity is not so big (inputs 1-3) ASM version is way faster than the C# one. However when the maze is more complex, like the one from the input number 5, C# implementation is faster.

Moreover there is another interesting relation. When the input data is not so complex the ASM version execution times do not differ much, the values are similar to each other when the C# ones are very different from each other. When the input data is more complex we have the opposite situation – C# values seem to be more “stable”.

Based on presented data we can come to the conclusion that the ASM version is better for not complicated mazes when the C# one is the better choice when the maze is complex.