BFS: Breadth-First Search

1 Introduction

In this session you are required to implement and analyse BFS (Breadth-First Search), as presented in section 22.2 of Introduction to Algorithms by Th. Cormen et al.

2 The structure of the boiler plate code

You are already provided with several source files:

- `main.cpp` - the main source file, which makes the calls to the implemented functions and provides the visualization code
- `bfs.h` - contains definitions of data structures and functions used in the project
- `bfs.cpp` - will contain the implementations of the required algorithms
- `grid.txt` - the maze which represents the graph for the demo
- `Profiler.h` - the library used for algorithm evaluation and chart generation

!!! You should only make changes to `bfs.cpp`.

For a user-friendly visualization, `main.cpp` displays an ASCII-like interface, in which the maze (i.e. the graph) is displayed (black cells are free, white cells represent walls).

2.1 Project setup for Windows, with Visual Studio

Make a new Project in Visual Studio. Make sure to check the *"Empty Project"* checkbox. Then, copy all the files provided in the project folder.

Then, in Visual Studio, add the two files `.h` to the *"Header Files"* section of your project, and the two `.cpp` files to the *"Source Files"* section (right click on corresponding project section → *"Add"* → *"Existing Item"*, see Figure 1.

2.2 Project setup for Linux and Mac

You may edit the source files using your editor of choice. The project comes with a `Makefile`, so it is enough to run `make` in a terminal to compile it and generate the executable. The resulting `.exe` will be called `main`, and can be run in the terminal by executing `./main`.

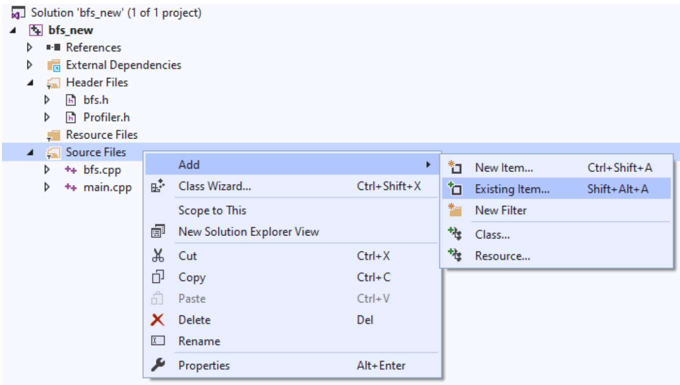

Figure 1: The *"Solution Explorer"* window in Visual Studio

3 Running the program

When you run the program, it will display the maze, similarly to what you can observe in Figure 2.

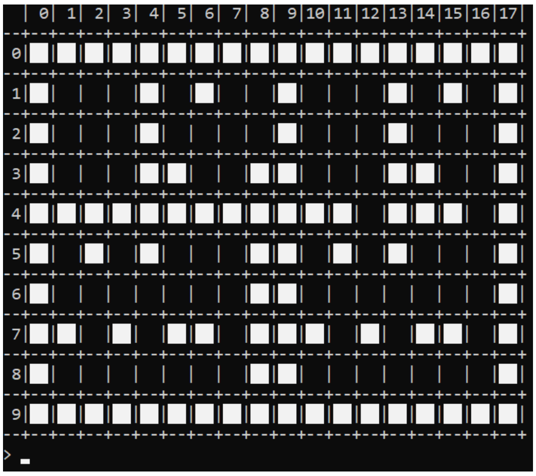

Figure 2: Interface of the program

The user may input one of the following commands:

- `exit`
  program termination
- `clear`
  clear the previous information from the grid
- `neighb <row> <col>`
  display the neighbors of cell on row `<row>` and column `<col>`.
- `bfs <row> <col>`
  execute the BFS traversal, starting from the cell on row `<row>` and column `<col>`.

- bfs_step <row> <col>
  same as bfs, but the result is displayed step by step, depending on the distance from the source node
- bfs_tree <row> <col>
  same as bfs, but it will also display the output tree under the grid
- path <row1> <col1> <row2> <col2>
  displays the shortest path between (<row1> <col1>) and (<row2> <col2>)
- perf
  generates the charts for the algorithm evaluation

3.1 Example: command neighb

If you run:
neighb 2 3 you should get the output displayed in Figure 3.

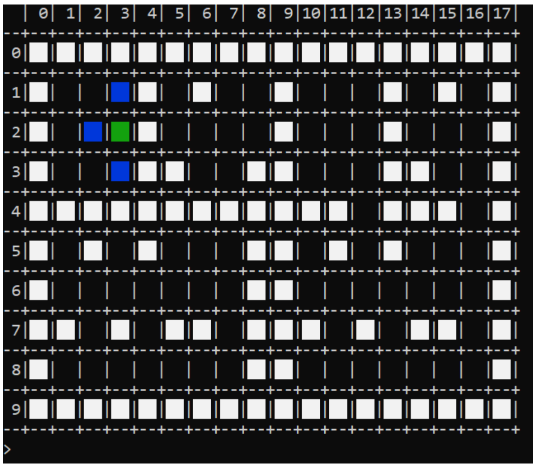

Figure 3: The result of running command neighb 2 3

The starting cell will be coloured green, with its neighbors coloured blue.

Since get_neighbors() is not yet implemented (you will be required to do the implementation), you will not get this answer if you run the command. Once you implement the function, you may use this command to check the correctness of your implementation. Each cell can have at most 4 neighbors (up, down, left, right); cells outside the grid or wall cells must not appear as neighbors.

3.2 Example: commands bfs and bfs_step

Upon running the command:
bfs 6 3 the program should output the result shown in Figure 4.

The source cell will be coloured green, and the other cells traversed will be coloured blue. Each blue cell will have an arrow on it, indicating the direction of the parent in the BFS tree.

Currently, the bfs() function is not implemented. Once implemented, use this command to check the correctness of your implementation.

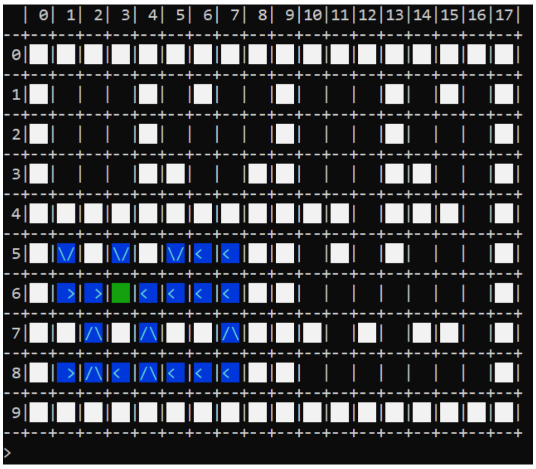

Figure 4: The result of running command `bfs 6 3`

3.3 Example: command `bfs_tree`

By running:
`bfs 2 6` you should get the image in Figure 5.

The root of the tree is the source node, i.e. `(2, 6)`. The children of this node in the tree are: `(2, 5)`, `(2, 7)` and `(3, 6)` (the order might differ, according to the implementation).

3.4 Example: command `path`

By running:
`path 5 10 3 15` you should obtain the image in Figure 6.

The source cell will be coloured green, the destination cell red, and the rest of the cells on the path - blue. Each blue cell will also have an arrow indicating the direction of the traversal.

Currently, `shortest_path()` and `bfs()` are not implemented, so you will not see this result when running the `path` command. Once the functions implemented, use this command to check the correctness of your implementations.

3.5 Employed data structures

The file `bfs.h` contains definitions of data structures used in the framework.

The `Grid` structure models a grid, having `rows` lines and `cols` columns, the grid elements are stored in the matrix `mat`. An empty cell has value 0 in the matrix, and a wall cell will have value 1.

The `Point` structure models a point (i.e. a grid cell) the `row` and `col` fields representing the location, i.e. the row and the column of the point in the grid.

The `Node` structure models a node in the graph, and it contains:

- `position` having type `Point` represents the cell corresponding to the graph node.
- `adjSize` - the number of neighbors of the node

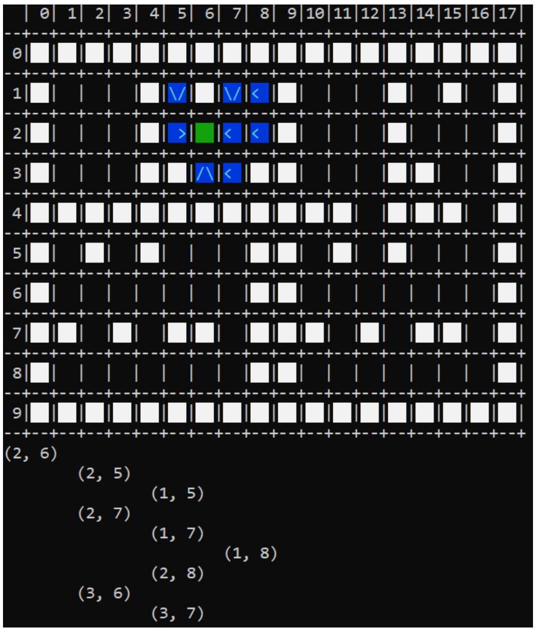

Figure 5: The result of running command `bfs_tree 2 6`

- `adj` - the neighbor array, containing `adjSize` neighbours
- `color` - the color of the node; initially, all nodes are colored `COLOR_WHITE`, i.e. the color has value 0
- `dist` - the distance from the source node, in BFS
- `parent` - pointer to the parent node, in the BFS tree

The `Graph` structure models the graph; it has as members the number of nodes, `nrNodes` and the array `v` containing pointers to the neighbor arrays of each node.

4 Requirements

4.1 Determine the neighbors of a cell (2p)

In `bfs.cpp`, tyou have to complete the `get_neighbors()` function which receives a pointer to a `Grid` structure, a point `p` of type `Point` and an array `neighb` of points, which the function will fill with the neighbors of point `p`. The function returns the number of neighbors filled in the `neighb` array.

A point on the grid can have maximum 4 neighbors (up, down, left, right). Not all neighbors are necessarily valid: some may end up outside the grid (negative, or out of bounds coordinates) or inside a wall.Therefore, after computing the position of a potential neighbor, you should check that it is situated inside the grid, on a free value cell (the value in the matrix of the corresponding cell should be 0).

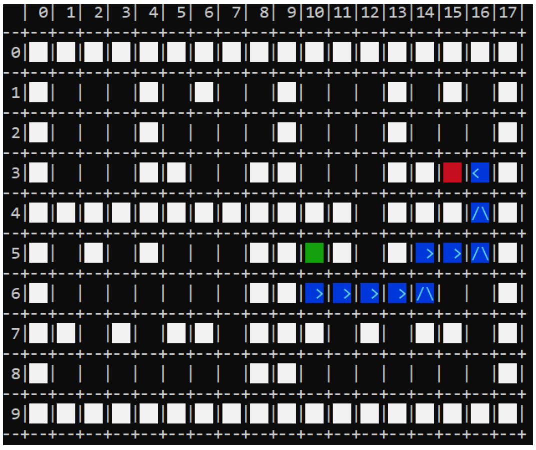

Figure 6: The result of running command `path 5 10 3 15`

The valid neighbors will be added to the array `neighb`. It is guaranteed that the array has at most 4 elements, so you may not exceed this capacity. Because the number of neighbors could be less than 4, you should also return this information.

4.2 BFS algorithm implementation (3p)

In `bfs.cpp`, you have to complete the function `bfs()` which receive as arguments a pointer to a structure of type `Graph` and the source node `s` of type `Node*`. The function will implement BFS, as specified in the algorithm from the book (see section 22.2 from Cormen).

Initially, the nodes of the graph are colored white, i.e. `COLOR_WHITE`, and the `dist` and `parent` fields are initialized with 0 and `NULL`, respectively. At the end of the traversal, all nodes that can be reached from the cource node are colored `COLOR_BLACK`, the distance `dist` has as value the number of steps from the source node to that node, and the `parent` pointer should idicate the parent in the BFS tree.

4.3 Pretty printing the BFS tree (2p)

In `bfs.cpp`, t you have to complete the implementation for `print_bfs_tree()` which receives as parameter a pointer to a `Graph` structure on which the BFS algorithm has already been run, so the node colors and the parent information is already set.

In the function, you already have the construction of the parent array `p`, in which the nodes colored black in the BFS traversal will be numbered from 0 to n. Also, it builds the `repr` array, which contains the coordinates of each node (in the grid).

To display this tree, you have to adapt the code from the multiway trees assignment.

4.4 Evaluate the performance of BFS (3p)

The `performance()` function evaluates the BFS algorithm, by varying, in turn, the number of edges, then the number of nodes (and always keeping the other as constant). For each value , you have to implement the generation of a random, connected, graph, having a given number of nodes and edges, respectively.

Inside the `bfs()` function, you will have to actually count the operations performed, using the optional argument `op`. Because this parameter is optional, sometimes `bfs()` will be called by the framework with this parameter set to `NULL`. Consequently, whenever counting an operation, you must first check that `op` is a valid pointer, i.e.:

```
if(op != NULL) op->count();
```

4.5 Bonus: Determine the shortest path (0.5p)

In `bfs.cpp`, you have to fill in the code for the `shortest_path()` function, which receives as argument a pointer to a `Graph` structure, a source node and a destination node `start` and `end` of type `Node*`, and a `path` array, y - where the result will be stored, i.e. the nodes on the path, in order. The function returns the number of nodes stored in `path`.

To determine the shortest path between two nodes, you should use the already implemented BFS, and reconstruct the path from the parent array computed by BFS.

The `path` array has a capacity of 1000 elements (upon the call). The function should return the number of elements that it contains - i.e. the path length - or -1 in case there is no path from `start` to `end`.

4.6 Bonus: Where can a knight end up on the board? (0.5p)

Using this framework, show that a knight starting from the up-left corner can end up in any position of an empty chess board. Give examples of empty chessboards that contains positions unreachable by a knight.