# Exercises: Traverse a Graph; Escape from Labyrinth

This document defines the **in-class exercises** assignments for the ["Data Structures" course @ Software University](https://softuni.bg/trainings/1147/Data-Structures-June-2015).

# Part I – Traverse a Graph to Find Its Connected Components

The first part of this lab aims to implement the **DFS algorithm** (Depth-First-Search) to **traverse a graph** and find its **connected components** (nodes connected to each other either directly, or through other nodes). The graph nodes are numbered from 0 to n-1. The graph comes from the console in the following format:

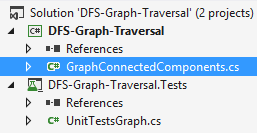
* First line: number of lines n
* Next n lines: list of child nodes for the nodes 0 … n-1 (separated by a space)

Print the connected components in the same format as in the examples below:

|  |  |  |
| --- | --- | --- |
| **Input** | **Graph** | **Output** |
| 9  0->3 6  1->3 4 5 6  2->8  3->0 1 5  4->1 6  5->1 3  6->0 1 4  2 |  | Connected component: 6 4 5 1 3 0  Connected component: 8 2  Connected component: 7 |
| 1  0 |  | Connected component: 0 |
| 0 | (empty graph) | Connected component: |
| 8  2 6  1  4  3  1 |  | Connected component: 0  Connected component: 2 6 1  Connected component: 4 3  Connected component: 5  Connected component: 7 |
| 4  1 2 3  0 1 2 3 3  0 1 3  0 1 1 2 |  | Connected component: 3 2 1 0 |

## Graph Traversal – Project Skeleton

You are given a **Visual Studio project skeleton** (unfinished project) holding the unfinished class GraphConnectedComponents and **unit tests** for its functionality. The project holds the following assets:

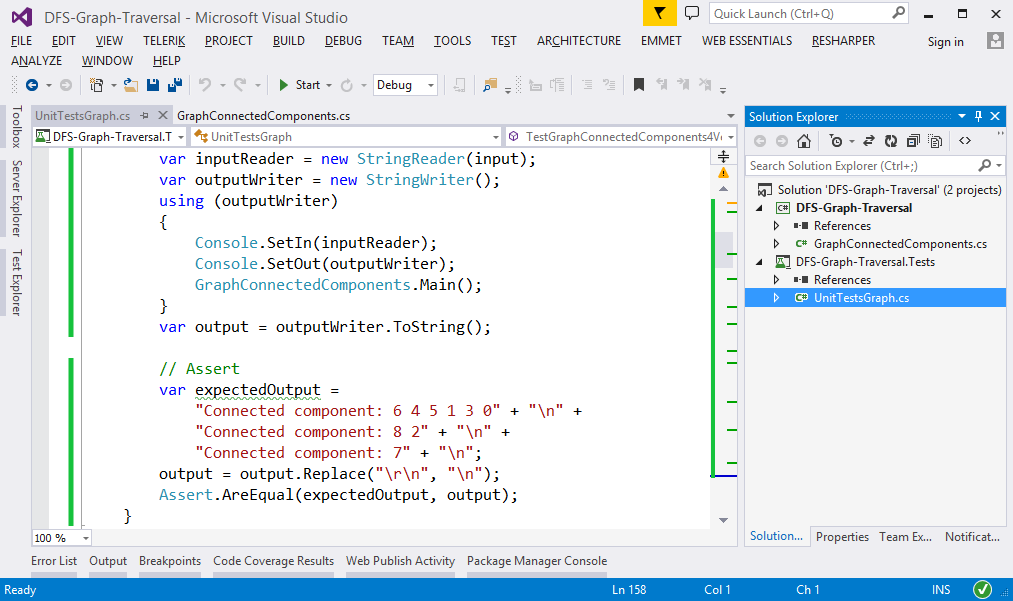


The project skeleton opens correctly in **Visual Studio 2013** but can be open in other Visual Studio versions as well and also can run in **SharpDevelop** and **Xamarin Studio**.

The unfinished GraphConnectedComponents class stays in the file GraphConnectedComponents.cs:

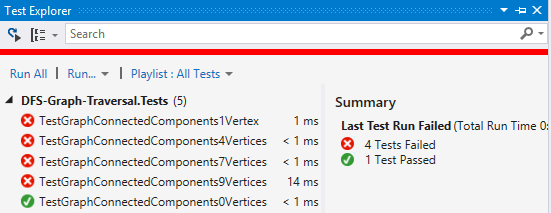
|  |
| --- |
| GraphConnectedComponents.cs |
| public class GraphConnectedComponents  {  public static void Main()  {  // **TODO: implement me**  }  } |

The project comes with **unit tests** covering the functionality of the GraphConnectedComponents class:



## Run the Unit Tests to Ensure They Initially Fail

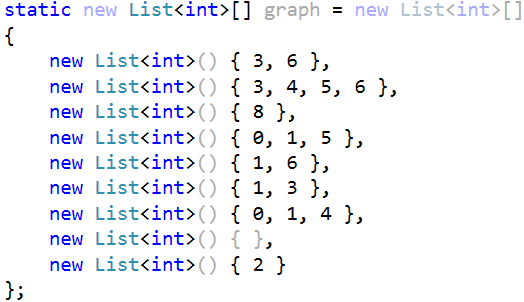
**Run the unit tests** from the DFS-Graph-Traversal.Tests project. Open the "**Test Explorer**" window (Menu 🡪 Test 🡪 Windows 🡪 Test Explorer) and run all tests. The expected behavior is that all tests should fail:



This is quite normal. We have unit tests, but the code covered by these tests is missing. Let's write it.

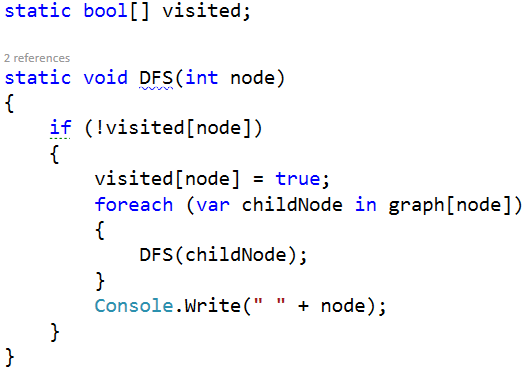
## Define a Sample Graph

The first step is to define a sample graph. It will be used to test the code during the development:



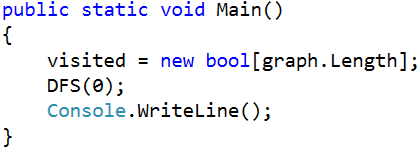
## Implement the DFS Algorithm

The next step is to implement the **DFS** (Depth-First-Search) algorithm to traverse recursively all connected nodes reachable from specified start node:

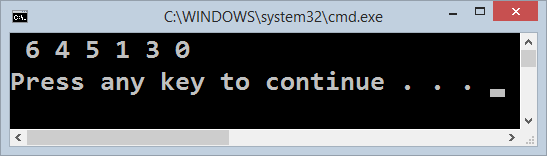


## Test the DFS Algorithm

Now, test whether the DFS algorithm implementation. Invoke it starting from node 0. It should print the connected component, holding the node 0:

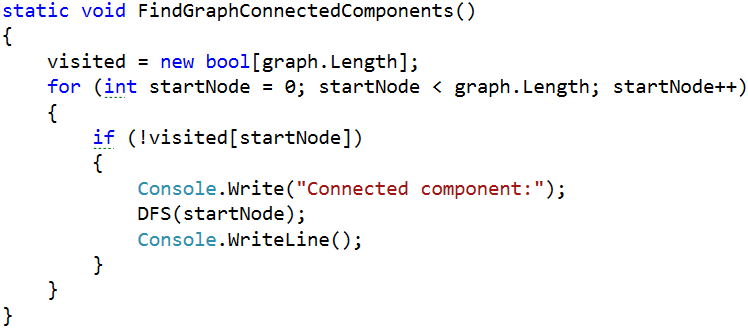


Now run the code above. It should find the first connected component in the graph, holding the node 0:

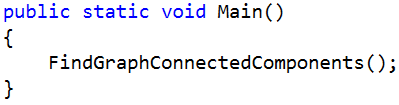


## Find All Connected Components

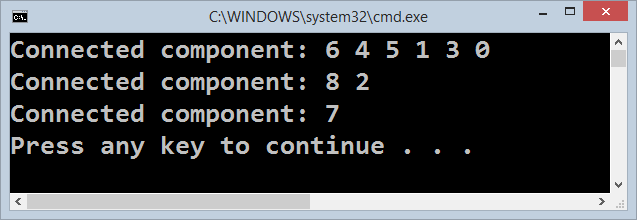
Now, we have DFS algorithm implemented, which finds the connected component holding all nodes reachable from given starting node. This is good, but we want to find all connected components. We can just run the DFS algorithm many times from each node (which was not visited already):



Now let's test the above code. Just call it from the main method:

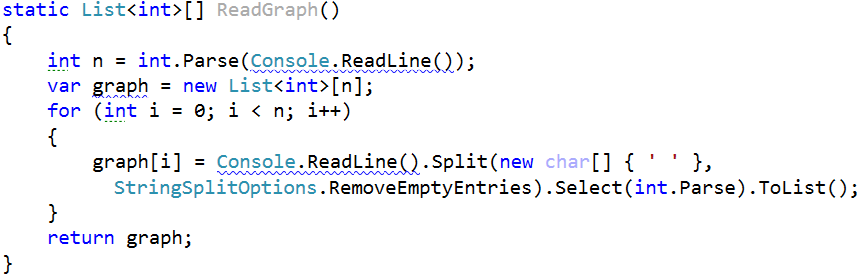


The output is as expected. It prints all connected components in the graph:

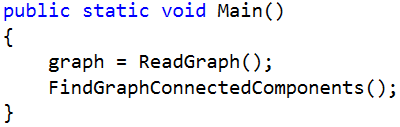


## Read the Input Data from the Console

Usually, when we solve problems, we work on hard-coded sample data (in our case the graph is hard-coded) and we write the code step by step, test it continuously and finally, when the code is ready and it works well, we change the hard-coded input data with a logic that reads it. Let's implement the data entry logic (read graph from the console):



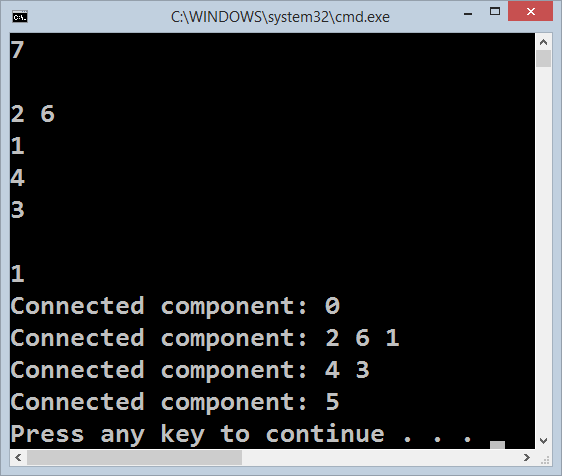
Modify the main method to read the graph from the console instead using the hard-coded graph:



Now test the program. Run it ([Ctrl] + [F5]). Enter a sample graph data and check the output:

|  |  |  |
| --- | --- | --- |
| **Input** | **Graph** | **Expected Output** |
| 8  2 6  1  4  3  1 |  | Connected component: 0  Connected component: 2 6 1  Connected component: 4 3  Connected component: 5  Connected component: 7 |

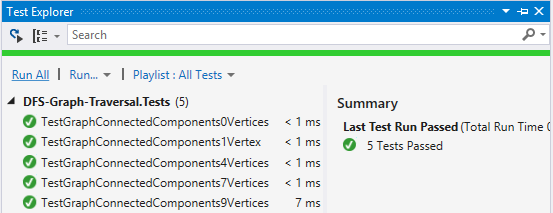
Seems like it runs correctly:



We are ready for the unit tests.

## Run the Unit Tests

Seems like we solved the graph problem. Let's run the unit tests that come with the program skeleton:

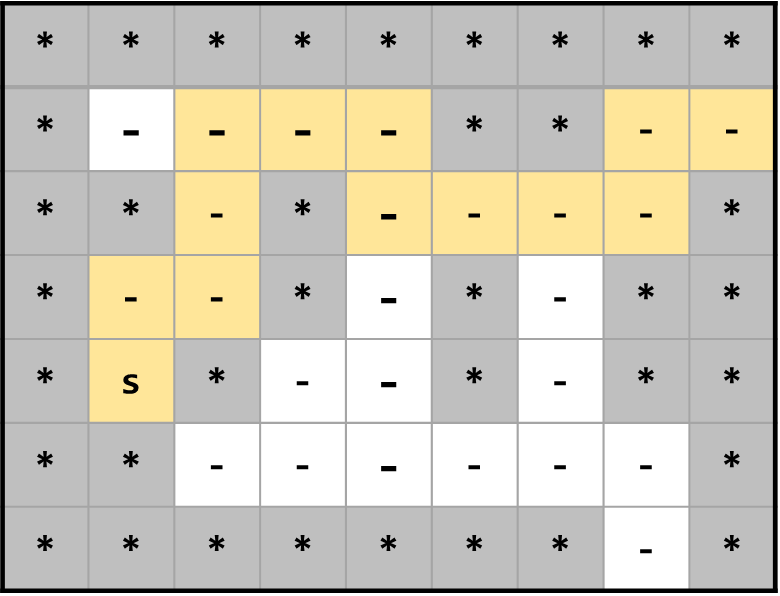


Congratulations! You have implemented the DFS algorithm to find all connected components in a graph.

# Part II – Find the Nearest Exit from a Labyrinth

The second part of this lab aims to implement the **Breadth-First-Search (BFS) algorithm** to find the nearest possible exit from a labyrinth. We are given a labyrinth. We start from a cell denoted by 's'. We can move **left**, **right**, **up** and **down**, through empty cells '-'. We cannot pass through walls '\*'. An exit is found when a cell on a labyrinth side is reached.

For **example**, consider the labyrinth below. It has size **9 x 7**. We start from cell {1, 4}, denoted by 's'. Тhe nearest exit is at the right side, the cell {8, 1}. The path to the nearest exit consists of **12** moves: URUURRDRRRUR (where 'U' means up, 'R' means right, 'D' means down and 'L' means left). There are two exits and several other paths to these exits, but the path URUURRDRRRUR is the shortest.



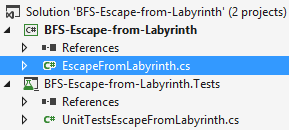
The input comes from the console. The first line holds the labyrinth width. The second line holds the labyrinth height. The next height lines hold the labyrinth cells – characters '\*' (wall), '-' (empty cell) or 's' (start cell).

Examples:

|  |  |  |
| --- | --- | --- |
| **Input** | **Labyrinth** | **Output** |
| 9  7  \*\*\*\*\*\*\*\*\*  \*----\*\*--  \*\*-\*----\*  \*--\*-\*-\*\*  \*s\*--\*-\*\*  \*\*------\*  \*\*\*\*\*\*\*-\* |  | Shortest exit: URUURRDRRRUR |
| 9  7  \*\*\*\*\*\*\*\*\*  \*----\*\*-\*  \*\*-\*----\*  \*--\*-\*-\*\*  \*s\*--\*-\*\*  \*\*------\*  \*\*\*\*\*\*\*-\* |  | Shortest exit: URUURRDRRDDDRD |
| 4  3  \*\*\*\*  \*-s\*  \*\*\*\* |  | No exit! |
| 4  2  \*\*\*\*  \*\*\*s |  | Start is at the exit. |
| 2  2  \*\*  \*\* |  | No exit! |

## Escape from Labyrinth – Project Skeleton

You are given a **Visual Studio project skeleton** (unfinished project) holding the unfinished class EscapeFromLabyrinth and **unit tests** for its functionality. The project holds the following assets:

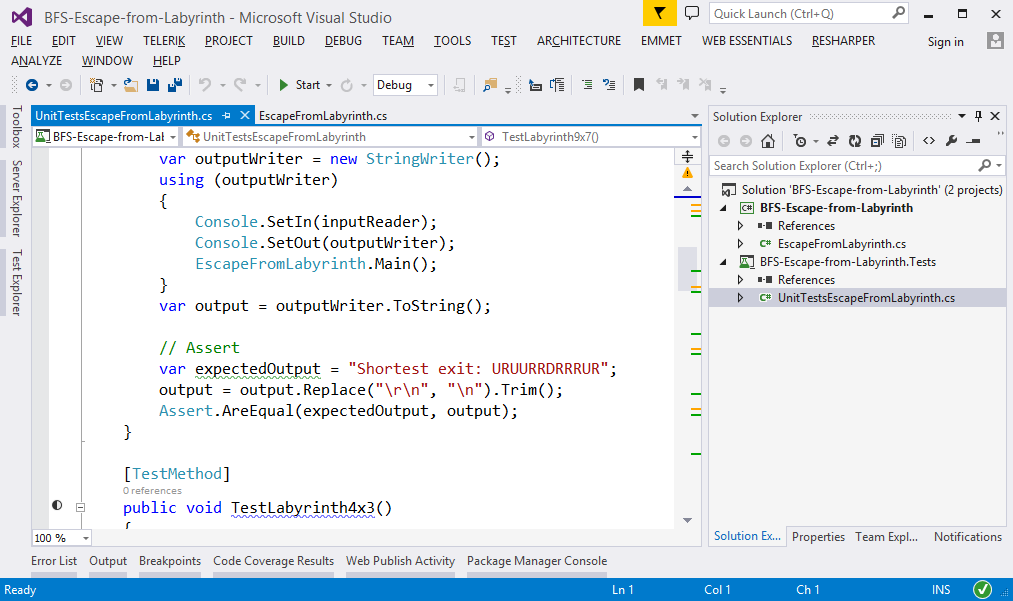


The project skeleton opens correctly in **Visual Studio 2013** but can be open in other Visual Studio versions as well and also can run in **SharpDevelop** and **Xamarin Studio**.

The unfinished EscapeFromLabyrinth class stays in the file EscapeFromLabyrinth.cs:

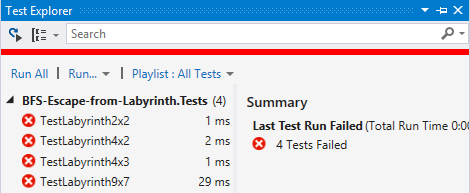
|  |
| --- |
| EscapeFromLabyrinth.cs |
| public class EscapeFromLabyrinth  {  public static void Main()  {  // **TODO: implement me**  }  } |

The project comes with **unit tests** covering the functionality of the EscapeFromLabyrinth class:



## Run the Unit Tests to Ensure They Initially Fail

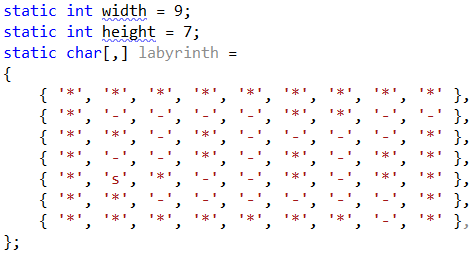
**Run the unit tests** from the BFS-Escape-from-Labyrinth.Tests project. Open the "**Test Explorer**" window (Menu 🡪 Test 🡪 Windows 🡪 Test Explorer) and run all tests. The expected behavior is that all tests should fail:



This is quite normal. We have unit tests, but the code covered by these tests is missing. Let's write it.

## Define a Sample Labyrinth

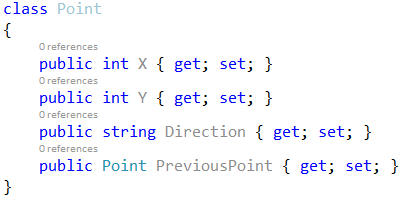
The first step is to define a sample labyrinth. It will be used to test the code during the development:



This sample data will be used to test the code we write instead of entering the labyrinth each time we run the program.

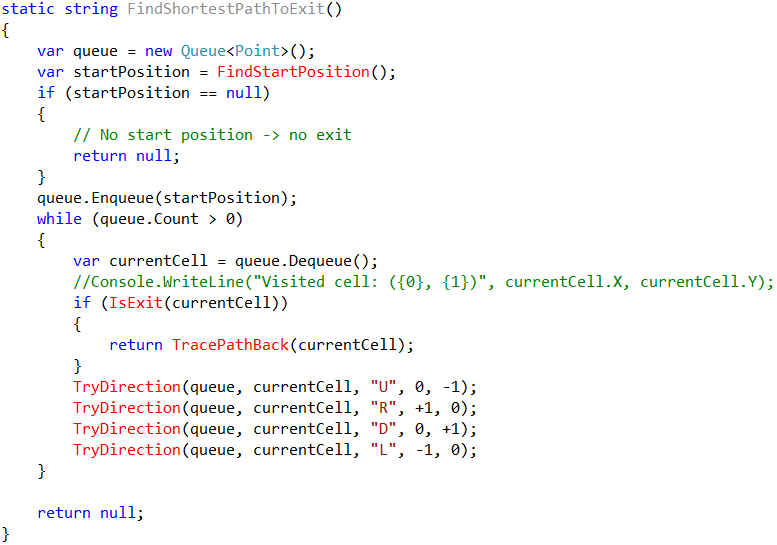
## Define the Class Point

We will define the class Point to hold a cell in the labyrinth (x and y coordinates). It will also hold the **direction** of move (Left / Right / Up / Down) used to come to this cell, as well as the previous cell. In fact, the class Point is a **linked list** that holds a cell in the labyrinth along with a link to the previous cell:



## Implement the BFS Algorithm

The next step is to implement the **BFS** (Breadth-First-Search) algorithm to traverse the labyrinth starting from a specified cell:



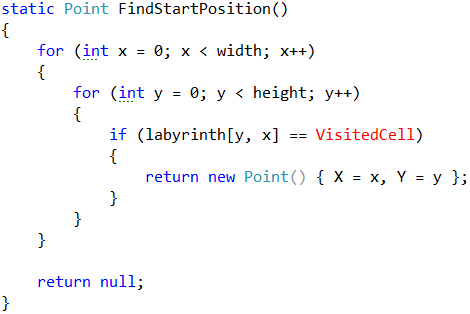
This is **classical implementation of BFS**. It first puts in the queue the start cell. Then, while the queue is not empty, the BFS algorithm takes the next cell from the queue and puts its all unvisited neighbors (left, right, up and left). If, at some moment, an exit is reached (a cell at some of the labyrinth sides), the algorithm returns the path found.

The above code has several missing pieces: finding the start position, checking if a cell is an exit, adding a neighbor cell to the queue, and printing the path found (a sequence of cells).

## Find the Start Cell

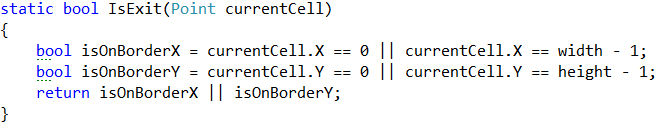
Finding the start position (cell) is trivial. Just scan the labyrinth and find the 's' cell in it:





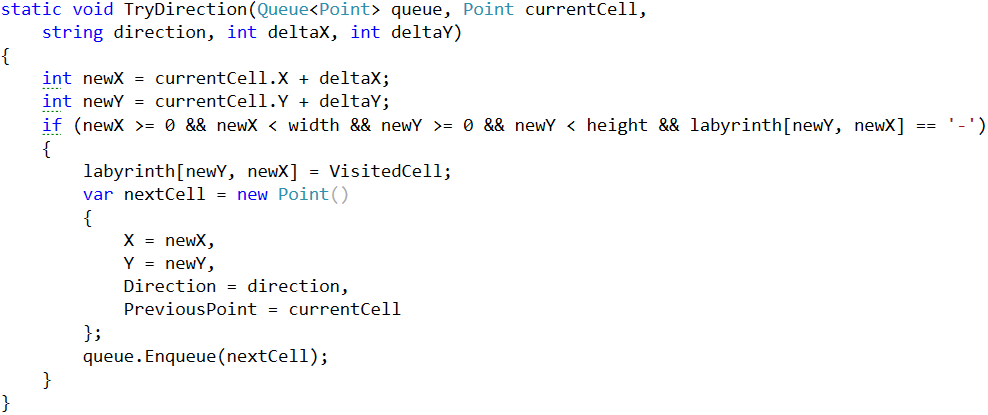
## Check If a Cell is at the Exit

Checking whether a cell is at the exit from the labyrinth is simple. We just check whether the cell is at the left, right, top or bottom sides:



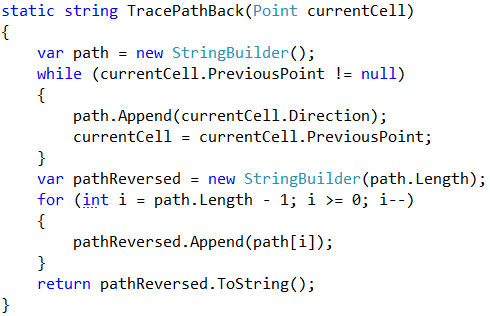
## Try the Neighbor Cell in Given Direction

Now, write the code to try to visit the neighbor cell in given direction. The method takes an existing cell (e.g. {3, 5}), a direction (e.g. right {+1, 0}). It checks whether the cell in the specified direction exists and is empty '-'. Then, the cell is changed to "not empty" and is appended in the queue. To preserve the path to this cell, it remembers the previous cell (point) and move direction. See the code below:



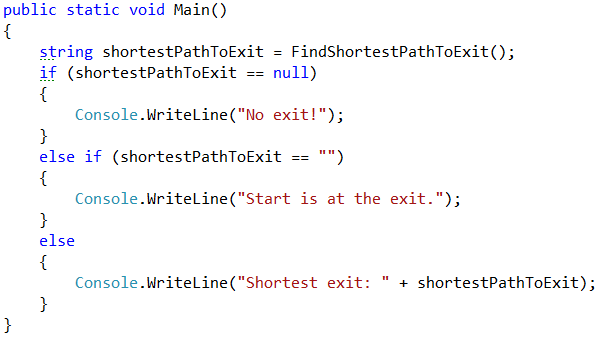
## Recover the Path from the Exit to the Start

In case an exit is found, we need to trace back the path from the exit to the start. To recover the path, we start from the exit, then go to the previous cell (in the linked list we build in the BFS algorithm), then to the previous, etc. until we reach the start cell. Finally, we need to reverse the back, because it is reconstructed from the end to the start:



## Test the BFS Algorithm

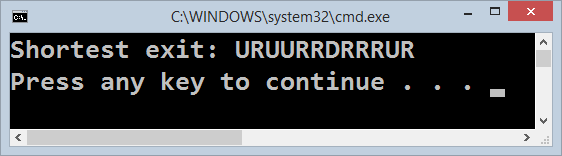
Now, test whether the BFS algorithm implementation for finding the exit from a labyrinth:



The method FindShortestPathToExit() returns a value that has three cases:

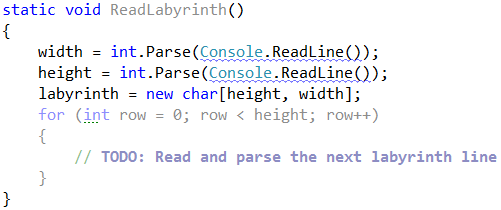
* null 🡪 exit not found
* "" 🡪 the path is empty 🡪 the start is at the exit
* non-empty string 🡪 the path is returned as sequence of moves

So, let's test the code. Run it ([Ctrl] + [F5]):



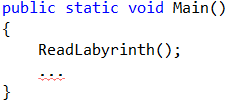
## Read the Input Data from the Console

Usually, when we solve problems, we work on hard-coded sample data (in our case the labyrinth is hard-coded) and we write the code step by step, test it continuously and finally, when the code is ready and it works well, we change the hard-coded input data with a logic that reads it. Let's implement the data entry logic (read the labyrinth from the console):



The code above is unfinished. You need to write it.

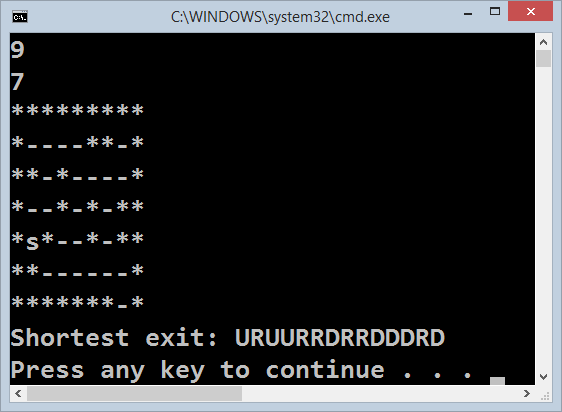
Modify the main method to read the labyrinth from the console instead of using the hard-coded labyrinth:



Now test the program. Run it ([Ctrl] + [F5]). Enter a sample graph data and check the output:

|  |  |  |
| --- | --- | --- |
| **Input** | **Labyrinth** | **Output** |
| 9  7  \*\*\*\*\*\*\*\*\*  \*----\*\*-\*  \*\*-\*----\*  \*--\*-\*-\*\*  \*s\*--\*-\*\*  \*\*------\*  \*\*\*\*\*\*\*-\* |  | Shortest exit: URUURRDRRDDDRD |

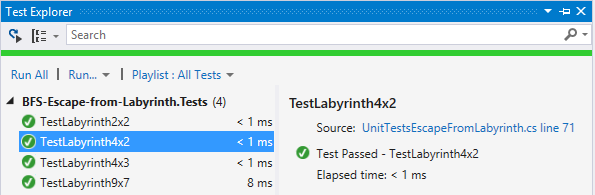
Seems like it runs correctly:



We are ready for the unit tests.

## Run the Unit Tests

Seems like we solved the labyrinth-escaping problem. Let's run the unit tests that come with the program skeleton:



All tests pass. Experienced developers will tell: "***hold the line green to keep the code clean***".

Congratulations! You have implemented the BFS-based escape from labyrinth algorithm.