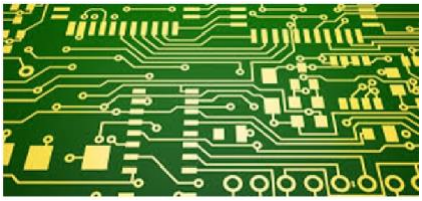
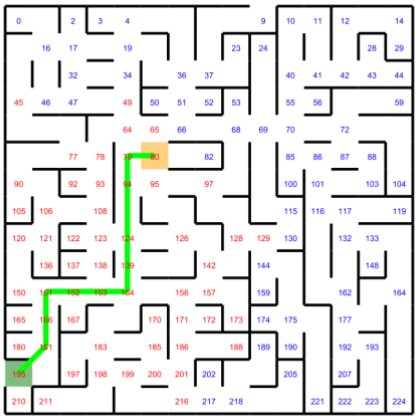
Algorithms and Data Structures

Escape the Maze with help of Graph Searches



Assignment-5

Version: 22.1, November 30th 2022



Introduction

In this assignment you will explore implementations of graph search methods that enable you to escape from our most challenging mazes in minimum time. You will implement three algorithms and compare results:

1. Depth-First Search
2. Breadth-First Search
3. Dijkstra's Shortest Path

These search methods play a key role in the software that optimize the wire routing of printed circuit boards (PCB). But

before you consider to already apply for a high paid job at https://www.unimicron.com/, possibly, you better first escape from our mazes…

Input data.

At https://en.wikipedia.org/wiki/Maze_generation_algorithm you find a variety of algorithms to generate challenging mazes with varying properties. In the Maze class of the provide starter project we already have implemented the "**Randomized Prim**" generator for you. This algorithm generates a spanning tree across all cells within the boundaries of a rectangular grid. Consequently, there will be exactly one path between any two cells in your maze and if you then open one wall at the boundary, there will be exactly one escape route starting from any other internal location.

But a spanning tree has no cycles and that is not enough of a challenge for you. So, the Maze class also implements a '**removeRandomWalls**' method which removes at random some additional walls in the maze (without leaving behind isolated corners of cells that are not connected to any wall anymore). In the picture above you find a 15x15 randomized Prim maze with 25 additional walls removed. Also, all walls around the orange starting cell have been removed. It should not be too difficult to find 5 or more cycles in that maze?

Visual representation

The Maze class of the starter project also provides a **print**() method and an **svgDrawMap()** method to create a visual representation of your maze. The print() method shows a character based console output of the lay-out of the maze. The svgDrawMap() method generates an .svg file in the target/classes folder with the lay-out of the maze, the start and exit point, the path found by one of your search methods and the identification numbers of all vertices that were found by the graph's neighbour detection algorithm. The implementation of the **getNeighbours**() neighbour vertex detection algorithm is also provided in the starter project. Cells in the maze that have exactly two walls are excluded from the set of vertices in the graph, because you can only just pass through these cells without any option for making a smart decision while searching… These cells do not show an id number in the picture.

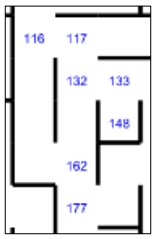
The text colour of the cell numbers in the picture indicates whether the vertex was visited by the search algorithm (=red) or not visited at all (=blue). You track this information for analysis purposes only, to be able to demonstrate efficiency of each of the algorithms and confirm that the aspired search strategy has been followed. In the production version of your solution, you would not bother to track this information when it is not used to find the solution path.

When paths are plotted in the map, for convenience, we draw straight lines between the vertices of the graph. Consequently, in our maps the solution path will not always follow the cell trajectories between two vertices, exactly.

Generic class AbstractGraph<V>

We expect you to provide a clean object-oriented solution in which the search algorithms are coded within a generic, abstract class **AbstractGraph<V>**. This class is leverages a Graph abstraction of the maze and should not need to know about other implementation details in the **Maze** class.

- The **V**(ertex) type parameter of this class represents a vertex of the graph which corresponds to a cell in your maze. We will identify maze cells by Integers, so you find that our Maze class extends AbstractGraph<Integer>. You may assume that the V type has a proper implementation of equals() and hashCode().

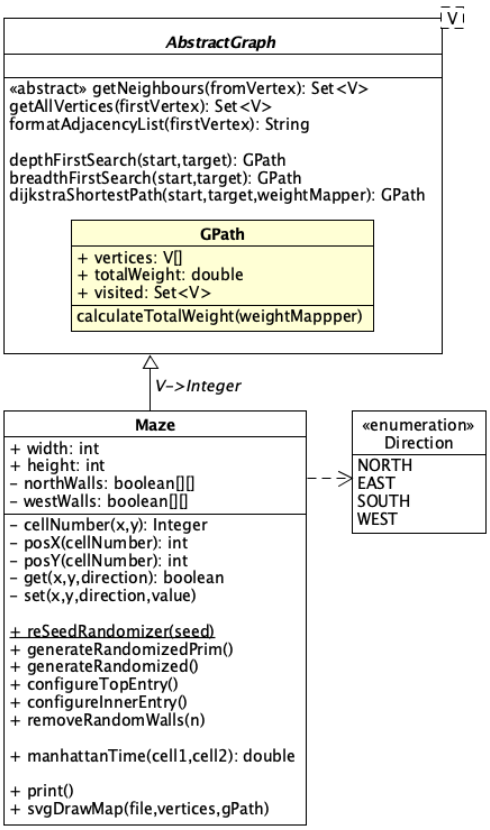


- The abstract method **getNeighbours**(fromVertex) is expected to return a Set of all vertices that are direct neighbours of the given fromVertex in the graph. It is not always the case that neighbouring vertices are also neighbouring cells in the maze. In the picture at the left you find that vertex 162 has three neighbours 116, 132 and 177 in the graph, whilst only cell 162 and 177 are truly next to each other in the maze. The Maze implementation of getNeighbours() will skip maze cells that only provide a direct passthrough and thereby no extra opportunity for a decision by your search algorithms.

You are expected to complete the implementation of five methods in the AbstractGraph class:

- **Set<V> getAllVertices(V firstVertex)** builds a set of all vertices in the graph that can be directly or indirectly found starting from the given firstVertex. If the graph is connected, this will deliver all existing vertices.
- **String formatAdjacencyList(V firstVertex)** uses StringBuilder to format the adjacency list of the graph according to the format:
  vertex1: [neighbour11,neighbour12,…]
  vertex2: [neighbour21,neighbour22,…]
  …
  Use a pre-order traversal of a spanning tree of the graph starting with firstVertex as the root, and using the getNeighbours() method to retrieve the roots of the child subtrees.
- **GPath depthFirstSearch(startVertex, targetVertex)** tries to find a path in the graph between the given start vertex and target vertex using a recursive depth-first search algorithm. If no such path can be found, it shall return null.
- **GPath breadthFirstSearch(startVertex, targetVertex)** tries to find a path in the graph between the given start vertex and target vertex using a breadth-first search algorithm. If no such path can be found, it shall return null.
- **GPath dijkstraShortestPath(startVertex, targetVertex, weightMapper)** tries to find a path in the graph between the given start vertex and target vertex using the Dijkstra Shortest Path algorithm. If no such path can be found, it shall return null. The weightMapper is a BiFunction<V,V,Double> which calculates the cost (distance or time) of traversing from one vertex in the graph to a neighbour. We use the function **Maze::manhattanTime** for this, which calculates a cost proportional to the manhattan distance between two cells, starting from a fixed cost of 1.0 which models some thinking time at each decision point in the maze.

The inner class **AbstractGraph.GPath** captures all relevant information about the outcome of a search. It represents the sequence of traversed vertices from the start vertex towards the target vertex and the total weight (sum of manhattanTime-s) of all connections in between. Only Dijkstra uses this weight in its search, for depth-first search and breadth-first search it can be calculated afterwards with the method GPath.reCalculateTotalWeight.
In addition, GPath tracks all vertices that have been visited during the search, such that we can easily visualise and compare the search characteristics of different algorithms.



Here, you find an outline UML class diagram of the classes in the starter project. (Some implementation details are omitted.)

In the unit tests you find another example of the topology of Europe being modelled by AbstractGraph<Country>. There every Country tracks its neighbours explicitly and the lengths of its borders are used as a measure of the distance between two countries. That independent example could guide you how to develop a tidy implementation in AbstractGraph.

Requirements.

Please complete the implementation of AbstractGraph<V> without breaking its abstraction or encapsulation. You find **// TODO** comments indicating where code is missing. The other classes are complete and should not require any change. You are not allowed to change the signature of any public method. You may change signatures of private methods and classes and add more public and private methods and attributes as you find appropriate.

Below is a summary of your tasks:

- R1. Download and unpack the starter project into your working folder and commit and you're your project to GIT before your first change of code.
- R2. Complete the implementation of AbstractGraph.getAllVertices.
  That enables the complete picture in your target/classes/*.svg output files.
- R3. Complete the implementation of formatAdjacencyList, practicing pre-order traversal.
- R4. Complete the implementation of AbstractGraph.depthFirstSearch such that it calculates a viable path, stops when a path is found and also populates the visitedVertices in the gPath. Verify your results from the console statistics and in the .svg picture.
- R5. Complete the implementation of AbstractGraph.breadthFirstSearch such that it calculates a viable path, stops when a path is found and also populates the visitedVertices in the gPath. Verify your results from the console statistics and in the .svg picture.
- R6. Complete the implementation of AbstractGraph. dijkstraShortestPath such that it calculates a shortest path (with minimum total weight), stops when such path is found and also populates the visitedVertices in the gPath. Verify your results from the console statistics and in the .svg picture.

R7. Provide a class with extra unit tests that verify relevant numbers of the output of the PrimMazeEscapeMain program (some of which you can verify against the sample output of Appendix B).
Also provide extra unit tests for scenario's that were impacted by defects in your initial development work but were not easy traceable from the available unit tests.

R8. Deliver tidy code:
a) with appropriate naming conventions and in-line comments.
b) with proper encapsulation, code reuse and low cyclomatic and cognitive complexity.
c) calculating the same intermediate result within the context of a method only once.
d) with acceptable CPU-time complexity and memory footprint (without easy avoidable waste).

R9. Provide a report including
a) an explanation of which search algorithm has calculated the route in each of the figures 1 – 6 in Appendix A, below.
b) Explanation and justification of seven of your most relevant code snippets in your solution.
c) Full console output of the PrimMazeEscapeMain program, and justification of the differences between your output and the reference console output that is provided in Appendix B below.
d) Full console output of the result of HugePrimMazeEscapeMain, running your algorithms on a maze of 5000 x 5000. If that doesn't succeed, try lower values of WIDTH until you find a solution on your computer within 5 minutes of calculation time, approximately.
e) References to external sources that you have consulted, if any.

Grading

For a sufficient grade, you shall have attempted all of above requirements, deliver all green unit tests and reproduce all results from the console output of appendix B except those which are due to valid implementation variations or due to at most two coding defects.

Appendix A.

Here you find six pictures of the result of a path search on a 30x30 maze. Three different search algorithms ‘depthFirstSearch’, ‘breadthFirstSearch’ and ‘dijkstraShortestPath’ with ‘Manhattan Time’ have been tried. Each search algorithm has been tried in two directions: ‘Escape’ from the inner orange cell towards the green exit, and ‘Treasure Search’ from the green entry point toward the orange treasure target cell. The found routes are depicted by a lime-green line. Visited cells have been noted with a red number. The blue numbers indicate vertices in the graph that have not been visited by the search algorithm. Cells without a number are pass-through cells that are not included as decision points by the algorithms. The cell numbers in the pictures are not readable, but that is not relevant: only their colours matter.

Indicate which picture belongs to what algorithm and provide your rationale of that assessment:

Algorithm	**Figure**	**Rationale**
Depth-First Escape (orange->green)		
Depth-First Treasure Search (green->orange)		
Breadth-First Escape (orange->green)		
Breadth-First Treasure Search (green->orange)		
Dijkstra Escape (orange->green)		
Dijkstra Treasure Search (green->orange)		

(This is a bit of a tricky question: with this information, two of the six searches cannot be distinguished.)

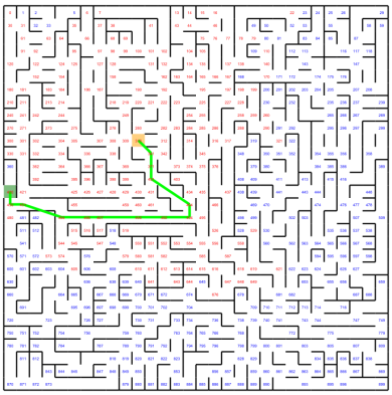


Figure 1

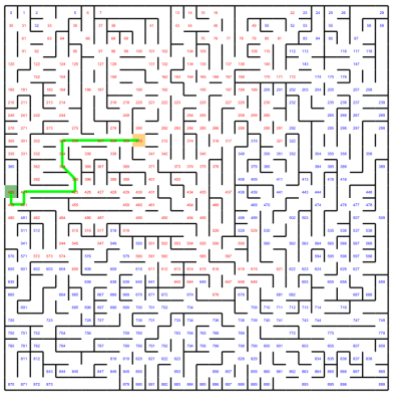


Figure 6

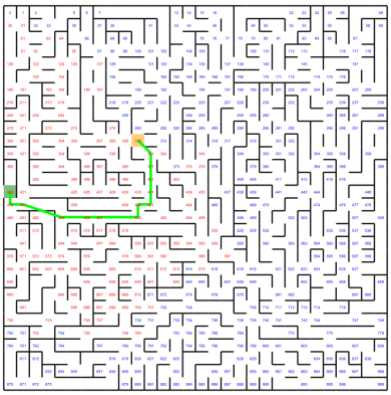


Figure 2

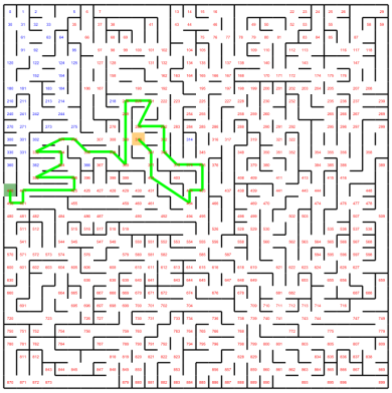


Figure 5

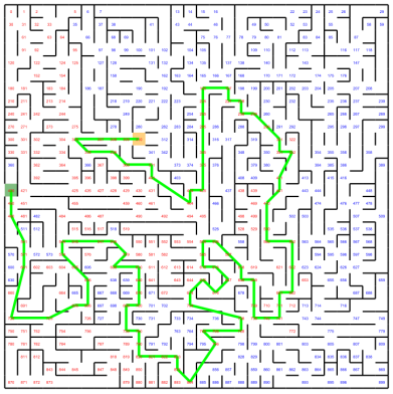


Figure 4

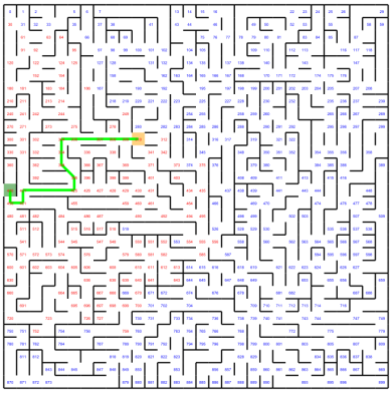


Figure 3

Appendix B.

Below you find console output of the example routes found by the PrimMazeEscapeMain program in the starter project after a slight variation in the implementation of getNeighbours().
The outcome of your implementation will differ in some of the numbers where the result is sensitive to the order in which neighbour vertices are explored. Identify and explain the specific sensitivities in your report.

```
Welcome to the HvA Maze Escape

Created 100x100 Randomized-Prim-Maze(20221203) with 250 walls removed
Maze-Graph contains 5565 connected vertices in 10000 cells

Results from 'Depth First Search' in 100x100 maze from vertex '3333' to '2000':
Depth First Search: Weight=1284,00 Length=461 visited=4974 (3333, 3434, 3633, 3634,
3934, 3935, 3936, 4036, 4136, 4236, ..., 2305, 2405, 2403, 2402, 2500, 2300, 2302,
2102, 2101, 2000)
Depth First Search return: Weight=668,00 Length=249 visited=5013 (2000, 2101, 2102,
2302, 2403, 2405, 2305, 2306, 2307, 2308, ..., 4236, 4136, 4036, 3936, 3935, 3934,
3634, 3633, 3434, 3333)

Results from 'Breadth First Search' in 100x100 maze from vertex '3333' to '2000':
Breadth First Search: Weight=119,00 Length=38 visited=1741 (3333, 3236, 3136, 3032,
2934, 2734, 2736, 2634, 2632, 2630, ..., 2411, 2509, 2508, 2308, 2307, 2306, 2205,
2102, 2101, 2000)
Breadth First Search return: Weight=119,00 Length=38 visited=475 (2000, 2101, 2102,
2205, 2306, 2307, 2308, 2508, 2509, 2411, ..., 2630, 2632, 2634, 2736, 2734, 2934,
3032, 3136, 3236, 3333)

Results from 'Dijkstra Shortest Path' in 100x100 maze from vertex '3333' to '2000':
Dijkstra Shortest Path: Weight=113,00 Length=40 visited=2162 (3333, 3332, 3231, 3131,
3031, 3032, 2934, 2734, 2736, 2634, ..., 2111, 2210, 2209, 2308, 2307, 2306, 2205,
2102, 2101, 2000)
Dijkstra Shortest Path return: Weight=113,00 Length=40 visited=632 (2000, 2101, 2102,
2205, 2306, 2307, 2308, 2209, 2210, 2111, ..., 2634, 2736, 2734, 2934, 3032, 3031,
3131, 3231, 3332, 3333)


    Process finished with exit code 0
```