University of Essex

BEng Computers with Electronics

CE301 Capstone Project

A tool for teaching graph algorithms

Author	Supervisor	Second assessor
Michal Borowski 1904535	Dr Alexandros Voudouris	Dr Cunjin Luo

30th April 2021

Acknowledgements

Abstract

The goal of this project is to create a program which will assist with teaching graph algorithms such as those taught in "CE204 Data Structures and Algorithms" module. It allows the user to select 1 of 5 available algorithms (Prim's, Kruskal's, Dijkstra's, BFS, DFS), and observe a step-by-step execution of the selected algorithm and key data used by it. It offers multiple pre-defined graphs selectable by the user as well as graph creating/editing facilities. The graphical display of algorithms implementation is provided at different levels of abstraction, using graph colouring and various sub-windows aiming to make algorithms' working principle as intuitive as possible, such windows include:

- step title/description window
- python-based pseudocode window
- edge/vertex collection windows (e.g. priority queue, stack)
- key data window ("V table")

When learning new concepts, working through practical examples "by hand on paper" is invaluable. While it can be easily done for algebraic equations, it is not as straightforward when learning about graph algorithms. Using this tool solves that problem, it makes graph creation quick and convenient, it improves learning experience by near-optimal representation of algorithm states and helps to consolidate understanding by using custom graphs.

Phrases used interchangeably

Phrases	Description
Vertex, node	A part of a graph.
Weight, cost, distance	Value assigned to an edge.
Source node, starting node	For example, the first expanded node in Dijkstra algorithm.
Neighbouring nodes, adjacent nodes	Nodes within distance of a single edge.

Table 1. Phrases used interchangeably.

Table of Contents

1. Existing solutions

There are many tools with similar functionality (Table 2). Most of them were created as a part of academic projects, and in many cases such projects are not actively maintained. Tools closely fitting the profile of this project, and the ones I found the most impressive are located in the upper area of the table.

Name	Characteristics
VisuAlgo (Active since 2011)	VisuAlgo seems like the most advanced publicly available tool of this kind. It is web-based, supports vast range of algorithms, including different implementations of the same algorithms. It allows to play/pause and step through algorithms. It gives very neat insight into algorithms state, including concise pseudo-code with highlighting of the current instruction, graph colouring, dynamic edge/node labels, animations, and precise textual description for every step. It is actively maintained since 2011 and had around 24 contributors since beginning (it was a part of multiple final year projects). Available at: https://visualgo.net/
TRAKLA2 (2009)	It is a teaching system where the teacher can track progress of individual students. It has exercises/visualisations of different types of algorithms (including but not limited to graph algorithms). It has pseudocode, dynamic graphs and various data representation items. These data representation items are the most advanced from all the other programs from this table. It is web-based, although the source code is available for download, it is not hosted on the internet as a service [6].
Algorithm Visualizer (Active since 2016)	A web-based algorithm simulator with interactive scripting interface used for graph creation and algorithms processing. It has great range of algorithms, nice "markdown" based descriptions, a log window with some information about the last algorithm step. In some cases, it displays the state of underlying data structures. It appears to be popular (has over 33k stars on GitHub) and it is open source with community-driven contributions. Available at: https://algorithm-visualizer.org
K. Mocinecová and W. Steingartner software (2020)	It has a neat interface with algorithm play/stop and step-by-step execution. It builds upon previous work in this area, including Graph Online, and Algorithm Visualizer (attempting to combine positives of previous tools). The insight into algorithm state is really precise in the breadth-first search (BFS) example shown in the original paper, however the tool itself seems to not be publicly available, therefore it is hard to compare against this project [7].
GraphTea (Active since 2015)	It has rich graph editing utilities, large range of algorithms and provides some insight into algorithms state. It is one of the few active projects of this kind [8].
Visage (2009)	Aside of graph editing facilities and the ability to assign algorithms to them, it contains a set of "teaching units". These are interactive descriptions of graph theory concepts. It appears that it can simulate algorithms step-by-step, during processing it displays pseudocode (with the current instruction highlighted), it also colours the graph and changes graph labels dynamically [9].

JAVENGA (2009)	It is one of the few tools from this table that displays the detailed information about algorithms internal state (e.g. the state of underlying data structures). Additionally, it has an option to step-back an algorithm processing which is also not common [10].
Sketchmate (2012)	A graph creator with algorithms simulation capability, and textual explanation of each step [11].
EVEGA (2001)	It stands for "Educational Visualisation Environment for Graph Algorithms". It supports GUI and built-in auto generators for graph creation. It allows to run algorithms automatically with controllable speed, also allowing step-by-step execution. The insight into algorithm state involves textual description, graph colouring and dynamic graph labels, the download link included in its publication is no longer active [12].
DIDAGRAPH (1998)	It has GUI and command interface for graph creation. It provides some insight into algorithms state and displays graph-oriented pseudocode [13].
Graphynx (2014)	Mobile solution, with some insight into algorithms state, it's not available anymore [14].
Graph Online (Active since 2015)	A web-based graph editor, supporting big range of algorithms but no insight into their state during processing [15]. Available at: https://graphonline.ru/en/
D3 Graph Theory (Active since 2017)	A web-based learning resource focused more on basic concepts of graph theory rather than algorithms [16]. Available at: https://d3gt.com/
Swan (2006)	It has a very distinct feature from other tools from this table, it can analyse C/C++ source code to find data structures and then represent them (which includes graphs). It can also step through algorithms and colour the graph during algorithm processing, however no textual explanation or deeper insight of algorithms state is displayed. It is not actively maintained anymore [17].
GraphShop (2011)	It has GUI and command interface (ECMAScript) used for graph creation. Using the scripting interface, the user can run custom algorithms on created graphs. It seems to not have step-by-step insight. It is not maintained anymore and its' binaries are unavailable [18].
Rin'G (2010)	A basic graph creator with limited insight into algorithms state during processing [19].
Rocs IDE (Active since 2014)	A graph creator focused more on final results rather than insight into algorithms state during processing, its' official hosting website is down [20].

Table 2. Tools with similar functionality.

1.1 Top features among existing tools

In this section I would like to outline the most useful or impressive features I found in similar programs and reflect on how my program deals with the same problems. I based the "usefulness and impressiveness" on the insight these features give into algorithm state (and how easy to interpret their representation is).

1.1.1 Visualisation of data structures (TRAKLA2)

These visualisations include arrays, hash tables, heaps, trees, even a call stack, allowing for graphical representation of recursive function calls. What is particularly good about these visualisations is the fact that they get updated as an algorithm progresses. In my program I implemented the "V table" item displaying the state of key data structures, as well as different types of node/edge collection graphic items that are very much

Figure 1. Visualisation of a hash table by TRAKLA2 tool.

like TRAKLA2 visualisations. In my program, each change to a data structure is highlighted by using orange colour.

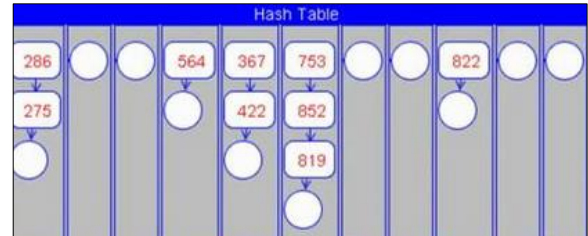Only few other programs from Table 2 attempt to output the state of data structures, these are: VisuAlgo (in textual from), JAVENGA (in textual form), Algorithm Visualizer (in visual form, but to limited extent).

1.1.2 Pseudocode with highlighting the last instruction (VisuAlgo)

VisuAlgo is one of the few programs introducing such feature, and it does it in concise and clear way.

It is worth to mention that Algorithm Visualizer goes a step ahead in this field. Although it shows the more extensive code, the code it shows is fully functional (it is not merely pseudocode).

Figure 2. Pseudocode with highlighting the last instruction by VisuAlgo tool, as used for Kruskal's algorithm.

My program displays a python-based pseudocode, it indicates groups of instructions that were executed (during the last algorithm step) by surrounding them with a rectangle.

1.1.3 Comprehensive algorithm state classification and output

From my observations, the tool created by K. Mocinecová and W. Steingartner (published in 2020, but not given a name) exhaustively classifies and displays the key information about algorithm state [7]. The breadth-first search (BFS) example presented in their publication displays 6 different categories of information as shown in Figure 3.

In my program I attempted to dissect the state of algorithms in similar fashion, mainly by using distinct colours directly on the graph and dynamic highlighting for affected nodes and edges (on the graph itself, and also on other sub-windows like "V table" and "collection graphics item"). Such colours/markers are labelled in the "Legend" sub-window. This and remaining features are further examined in the "Description of the system" section.
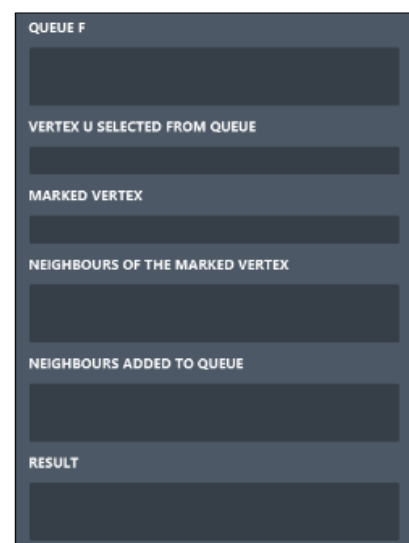

Figure 3. Types of information displayed for BFS algorithm in the tool created by K. Mocinecová and W. Steingartner (2020) [7].

1.2 Overall comparison with existing tools

The scope of this project is much smaller in comparison to programs from the top of Table 2 (that were typically created by groups of academics/professionals over several years). These programs tend to support more algorithms and have extensive interfaces (often web-based). However, I would risk a statement that what my program attempts to do it does well. The insight it gives into algorithm state is comprehensive and relatively easy to understand. There is not a single tool from Table 2 which would implement all the "top features" identified in the previous section, my program attempts to do so for the 5 algorithms it supports. In the next section, I further investigate into what makes it stand-out among similar tools by comparing it to GraphTea.

1.3 Comparison with GraphTea processing Dijkstra algorithm

In GraphTea program the user has to memorize and pay attention to relatively more changes during algorithm processing in order to keep track of what is going on. In my program more changes are evident by looking at the graph itself or by reading the description window, for example during Dijkstra algorithm processing, my program:

- uses blue colour for "considered" edges and uses orange highlighting to indicate the "considered" edges added during the last step (in case of GraphTea there's no indication which edges are those)
- uses brackets to display total distance from the starting node on the graph itself (in case of GraphTea the "currently computed distance" label is displayed, as shown near the top of Figure 4, however it does not indicate which path results in the "currently computed distance")
- has more detailed descriptions, often including reasoning behind operations
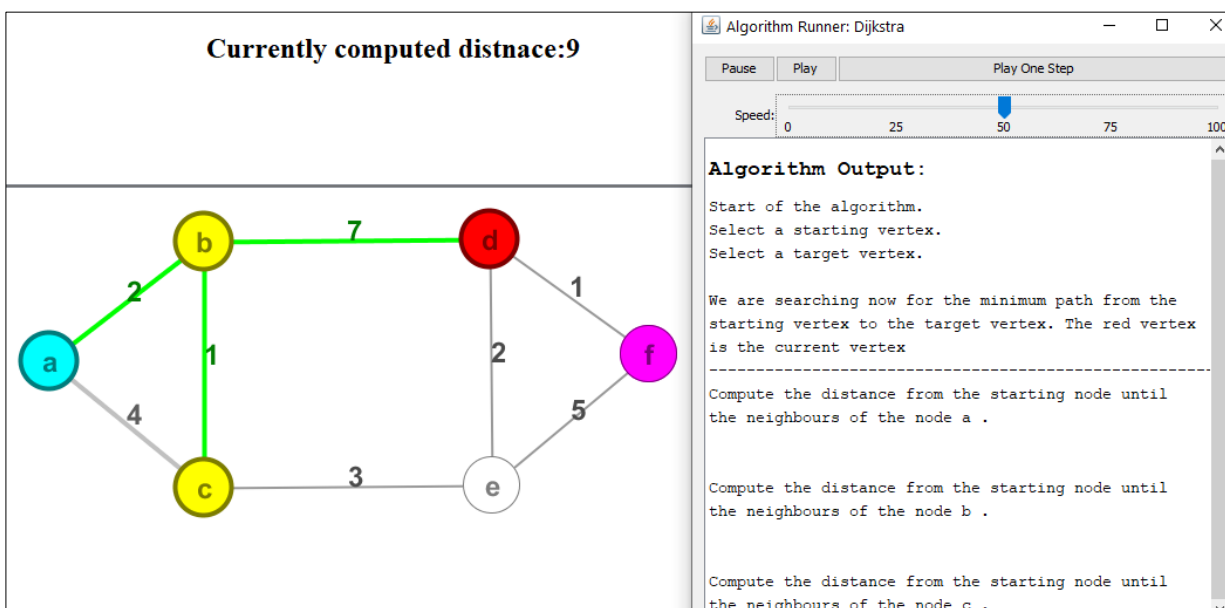
Figure 4. Screenshot from the GraphTea program interface processing Dijkstra algorithm.

Furthermore, the GraphTea and most of similar programs do not display dynamic graphical representation of underlying data structures, a feature that is very helpful to understand not only the working principle but also how to implement algorithms by manipulating data (only TRAKLA2 really implements it).

In the "Dijkstra algorithm implementation" section on page 11 I presented how my program processes the same graph example as the one from Figure 4 (above).

4

2. Introduction

This project is software based. As suggested in the Project Handbook [21], the final report gives an overview of the software, while the more detailed technical documentation is hosted at GitLab repository in the form of markdown (".md") files [22]. For each noteworthy feature, the documentation includes a demonstrative animation, a description, and a code snippet (in most cases). Aside of description of program features, the documentation includes implementation details, project management, and testing report. In the following chapters I will describe the functionality of the program, how I created it, and why it is useful.

2.1 Why is it useful?

According to the creators of educational DIDAGRAPH software [13], the algorithmic aspect of graph theory is known for being a topic that computer science students find difficult to grasp. That is partially due to intrinsic difficulty of the topic itself, and partially due to lack of adequate teaching tools. The program developed for this project is addressing that issue.

Typical algorithms teaching/learning methods involve familiarizing with:

- the working principle
- how to implement the working principle (how to manipulate data, possibly in efficient way)

Both of these may involve:

- verbal description (either spoken or written)
- practical example being shown

	Description	Example
Principle	✔	✘
Implementation	✔	✘

Table 3. Content of CE204 lecture slides.

	Description	Example
Principle	✔	✔
Implementation	✘	✘

Table 4. Content of video about MST by Abdul Bari.

If we looked closely at lectures about graph algorithms and analysed their content, we could produce a table showing what the lecture included. It could be found that the content is usually not exhaustive, which is the case with CE204 lecture slides [23] (Table 3) and the video about MST algorithms by Abdul Bari [24] (Table 4). As mentioned in the abstract of this project, a great potential lies in the use of practical examples. Interpretation of both aforementioned learning materials could be improved by using a tool which would present key data changes, teaching both: principle and implementation using practical examples.

That is exactly what this program offers. In a way, it uses programming to introduce dynamic, interactive implementations of beautiful fact sheets from the "Algorithms in a Nutshell" book [25] (e.g. Figure 5). These include all information needed to understand the "nuts and bolts" of a graph algorithm implementation:

- working principle
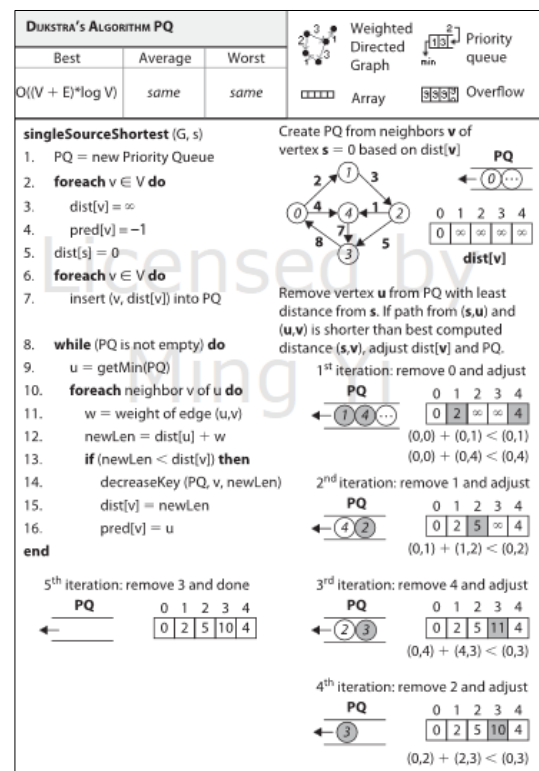- underlying data changes
- visualisation

Figure 5. Example of an algorithm fact sheet from "Algorithms in a Nutshell" book (Licensed by Ming Yi).

3. Description of the system

3.1 Graph creation

The graph can be created and edited by interacting with it directly using mouse and keyboard (see "Program controls" section on page 17 for details).

When designing the user interface and interactive features, a lot of my attention was directed on making the program convenient in use. One of such features is the "Select graph" window (Figure 6), providing graph saving, loading, removing, and selecting facilities. In-built graph manager like this is more convenient in use when compared to strictly file-based saving/loading.
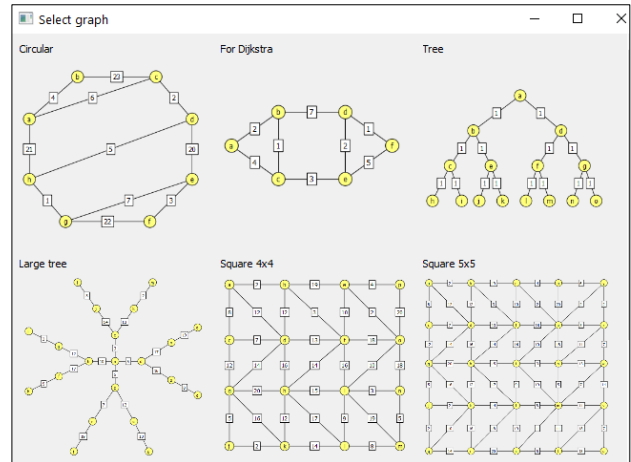
The algorithm selection window has an in-built "Use selected graphs" checkbox. Using this checkbox, the

Figure 6."Select graph" window.

user can select an algorithm from the combo-box and a graph suitable for teaching of specific algorithm will appear automatically. For example, Figure 7 shows the graphs suggested for BFS, DFS (left image) and Dijkstra (right image) algorithms. The suggested graphs were chosen based on their suitability to teach specific algorithms.
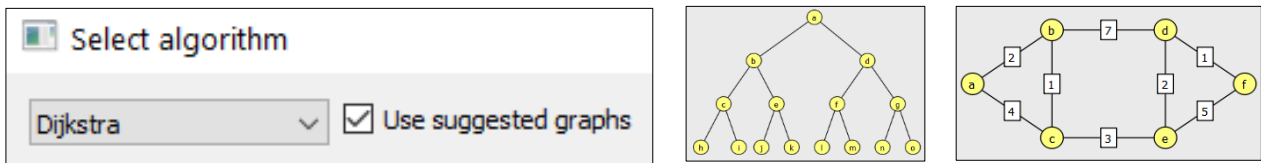

Figure 7. Algorithm selection facility with an option to use suggested graphs. On the right we can see graphs suggested for BFS, DFS (left) and Dijkstra (right) algorithms.

3.2 Common characteristics of implemented algorithms

Processing of each algorithm in the program is done on step-by step basis (triggered by the "right arrow" key), it involves displaying the similar set of sub-windows and graphical items:

- the **graph** itself (which gets coloured depending on the chosen algorithm)
- **legend** graphics item (describing what each colour means for the chosen algorithm)
- **pseudocode** graphics item (displaying python-based pseudocode and highlighting instructions executed during the previous step of the chosen algorithm)
- **V table** graphics item (displaying real-time key data used by the chosen algorithm)
- **collection** graphics item (e.g. priority queue, queue, stack; it is used to visualise how some algorithms prioritize certain nodes/edges before others)

Each algorithm has its' own short description in algorithm selection window. These descriptions usually include practical examples of using algorithms to solve real world problems. In case of MST algorithms (Prim's, Kruskal's), the example states that MST can be used to minimize the length of wires/pipes connecting all households in an area (Figure 8).

Prims algorithm finds minimum cost spanning tree (MST). It traverses the graph focusing on nodes. It selects a random one first, and then adds nodes based on closest distance to any previously added node.

Nodes could be treated as buildings/households within neighborhood, edge weights could be treated as distances between these households, in such scenario MST can be used to find minimum length of hydraulic pipes, or electronic wires, to supply all households with water or electricity using minimum total length of wires/pipes.

Figure 8. Screenshot of example algorithm descriptions from the algorithm selection window.

3.3 Kruskal's algorithm implementation

Kruskal's algorithm finds the minimum spanning tree (MST) by looping over every edge of a graph in ascending order in terms of their weights. It checks if adding particular edge would form a cycle, if that is the case then the edge is ignored (not added to MST). In the process of building MST, effectively it forms multiple **subgraphs** (Figure 9). In the program I created, subgraphs have different colours to easily distinguish them.

In case of Kruskal's algorithm, the "legend graphics item" is relatively simple (Figure 10).
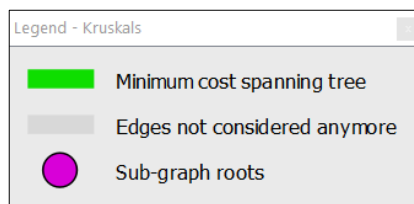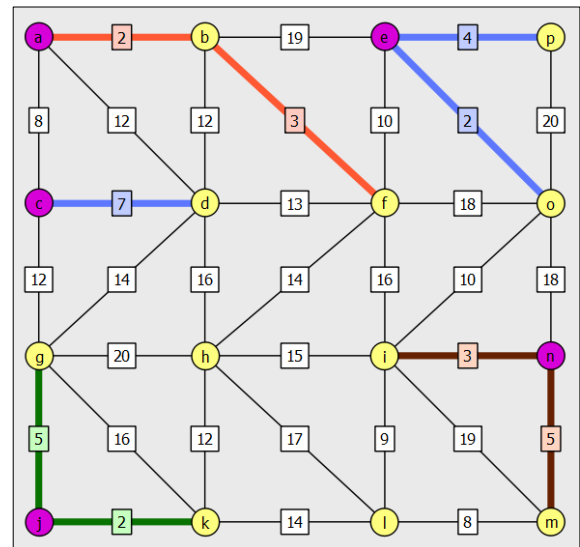

Figure 9. A coloured graph during Kruskal's algorithm processing. Different colours are used to signify separate subgraphs that formed up to this point.

Figure 10. Legend graphics item, used for Kruskal's algorithm.

Whenever subgraphs are merged (by inclusion of an edge that happens to connect 2 subgraphs), the new graph is painted with a single colour (Figure 11).

Checking if inclusion of an edge would create a cycle involves comparing root nodes of 2 subgraphs from 2 vertices of that edge. For that reason, root nodes have distinct colour (**purple**) from the rest of nodes. As shown in Figure 11, merging 2 subgraphs turns 1 of the roots into a regular node (node "n").

It is a time-consuming process to manually inspect the graph looking for the new edge with the lowest weight that was not processed yet. That is why I implemented a "priority queue graphics item" (Figure 12), it presents in what order the edges will be processed.
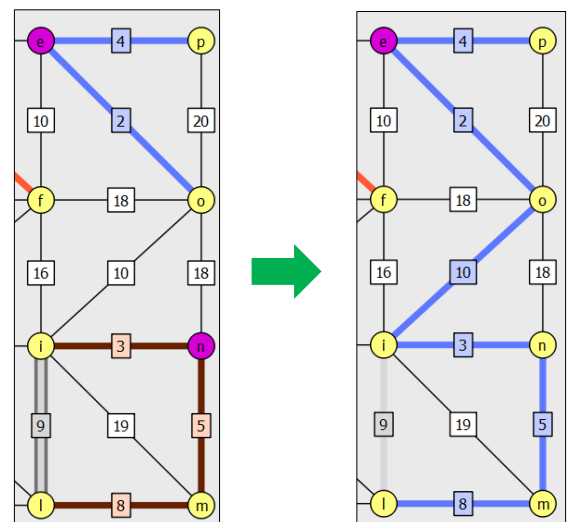

Figure 11. Kruskal's algorithm subgraphs retaining a single colour following inclusion of the "o-i" edge (in the middle of the graph).
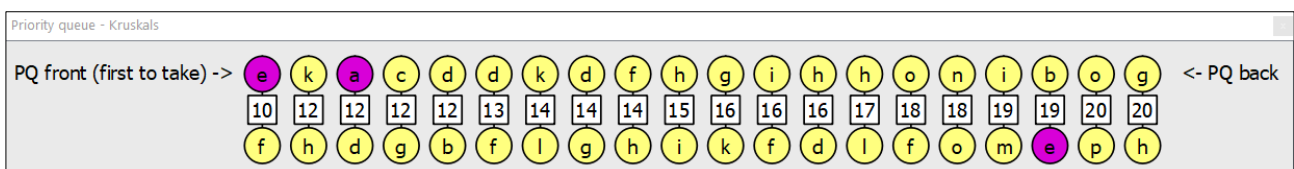

Figure 12. Priority queue graphics item presents what edges will be processed next by Kruskal's algorithm.

The pseudocode graphics item (Figure 13) contains python-based instructions that may help users to not only understand the principle of the algorithms working method, but also how to implement it, so it could be done automatically by the computer.

This is especially useful when combined with an insight into key data changes performed by the algorithm, which is available in the "V table" (table of vertices) graphics item (Figure 14). In case of Kruskal's algorithm, the "V table" contains an array of tuples where the 1st value is rank of a subgraph root node, and the 2nd value is a predecessor node (either the root of a subgraph, or a node leading to the root of a subgraph).

During the processing of an algorithm, descriptions and labels of each step are displayed.

Figure 15 and Figure 16 present labels, descriptions and illustration that differ, in one case an edge could be included to MST, in another case, an edge would create a cycle, therefore it could not be added to MST (grey colour is used to indicate that). In both cases the underlying reason is explained. That is helpful because it allows the user to discover what happens, and most importantly why it happens (so the user improves understanding and gains intuition).

V table - Kruskals

V[a] = (2, a)
V[b] = (0, a)
V[c] = (1, a)
V[d] = (0, c)
V[e] = (2, e)
V[f] = (0, a)
V[g] = (0, j)
V[h] = (0, h)
V[i] = (0, n)
V[j] = (1, j)
V[k] = (0, j)
V[l] = (0, n)
V[m] = (0, n)
V[n] = (1, e)
V[o] = (0, e)
V[p] = (0, e)

Figure 14. V table - Kruskal's.

Pseudocode - Kruskals

```python
def findSet(n):
    """ Finds sub-graph root node. """
    while n != pred[n]:
        n = pred[n]
    return n

def union(root_1, root_2):
    """ See "union by rank" for better efficiency
    due to faster "FindSet" lookup. """
    pred[root_1] = root_2

#initialization
pred = {}
for n in nodes:
    pred[n] = n   #each node becomes a sub-graph with 1 node
PQ = PriorityQueue()
for e in edges:
    PQ.enqueue(e)

MCST = []

# main work
for MCST.size() < node_count - 1:
    edge = PQ.dequeue()
    root_1 = findSet(edge.node_1)
    root_2 = findSet(edge.node_2)
    if root_1 != root_2:
        MCST.append(edge)
        union(root_1, root_2)  # merge 2 sets into 1
    else:
        # ignore this edge because it would create a cycle
        pass
```

Figure 13. Pseudocode graphics item - Kruskal's.

Edge 'j - k' (2) was added.
Edge 'a - b' (2) was added.
Edge 'n - i' (3) was added.
Edge 'b - f' (3) was added.
Edge 'p - e' (4) was added.
Edge 'm - n' (5) was added.
Edge 'g - j' (5) was added.
Edge 'c - d' (7) was added.
Edge 'l - m' (8) was added.

Edge 'l - m' (weight=8) was added to minimum cost spanning tree. It was chosen becaues it had the lowest weight of all remaining edges.

Using findSet method on both nodes and comparing the result allowed to verify that adding this edge wouldn't result in a cycle, because the root of node 'l' (root='l' itself) and 'm' (root='**n**') differs.

Joining of 2 graphs was performed by simply overwriting root node of 'l'.

Figure 15. Label, description, and corresponding subgraph illustration following a successful edge inclusion (because it didn't form a cycle). The edge "l-m" (weight 8) just got included in the brown subgraph.

Edge 'n - i' (3) was added.
Edge 'b - f' (3) was added.
Edge 'p - e' (4) was added.
Edge 'm - n' (5) was added.
Edge 'g - j' (5) was added.
Edge 'c - d' (7) was added.
Edge 'l - m' (8) was added.
Edge 'a - c' (8) was added.

Edge 'l - i' (weight=9) was dropped. That is because adding it would create a cycle. Node 'l' and node 'i' happen to have the same root ('**n**').

Figure 16. Label, description, and corresponding subgraph illustration following failed edge inclusion (because it would form a cycle). It can be noticed how edge "i-l"(weight 9) is marked with grey colour because this edge could not be added to the subgraph.

3.4 Prim's algorithm implementation

Prim's algorithm also finds MST, however unlike Kruskal's, it operates mainly on vertices, rather than edges. It builds the MST by "expanding" the lowest cost neighbour-node (such nodes are marked using **blue** colour in the program).

Prim's algorithm does not divide the graph into subgraphs, so the program colours all MST nodes **green** from the beginning.
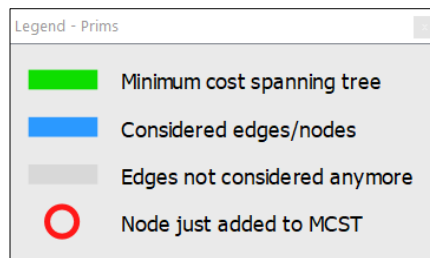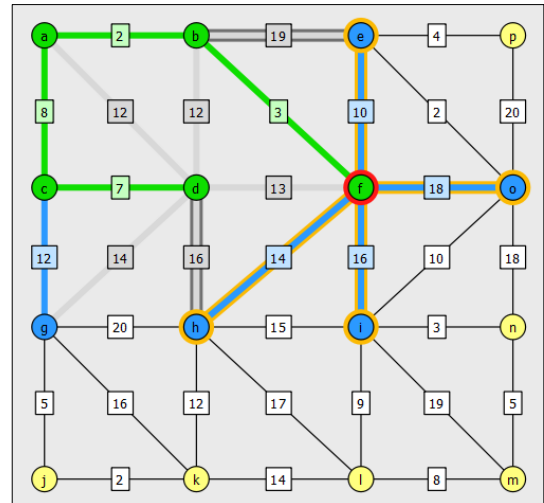

Figure 17. A coloured graph during Prim's algorithm processing.

Figure 18. Legend graphics item, used for Prim's algorithm.

If we look closely at Figure 17 we can see that the **node F** was recently added to MST, and "expanded". Meaning that distances/costs to all of its' surrounding nodes were registered (and will be considered when deciding which node to add next). **Orange** and **dark grey** colours signify these changes by temporarily surrounding edges and nodes affected by "expansion". Prim's algorithm repetitively repeats 2 major steps:

- Inclusion of a new node to MST. Based on the lowest cost among considered nodes (Figure 19).
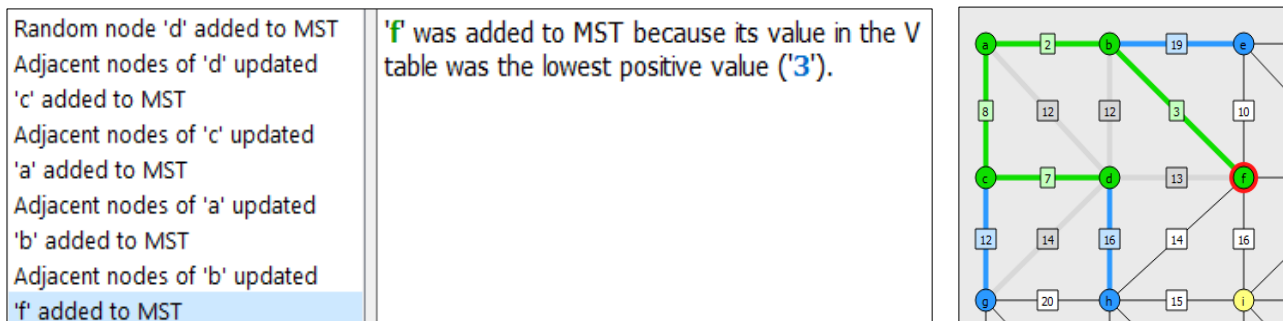- Expansion of the newly included node (Figure 20).

Figure 19. Label, description, and the corresponding graph illustration following an inclusion of a node to MST in Prim's algorithm.
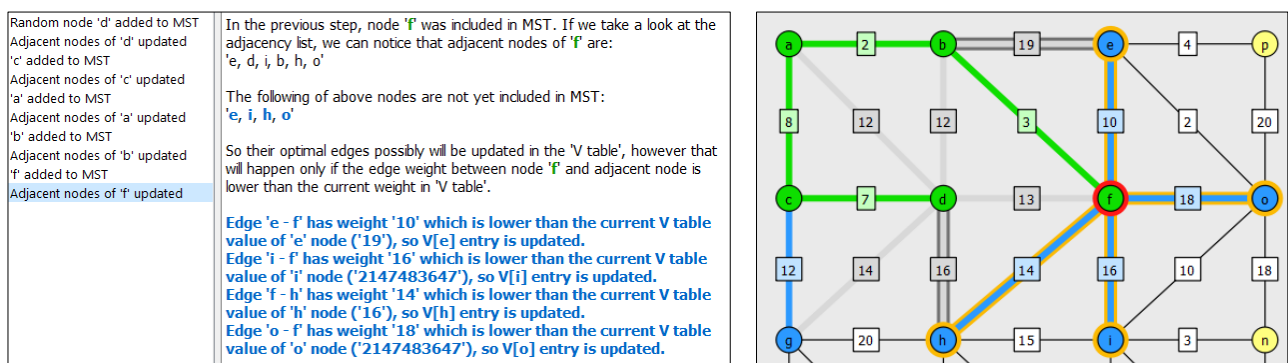

Figure 20. Label, description, and the corresponding graph illustration following an "expansion" of a node in Prim's algorithm (registering edge weights adjacent to the newly included node).

The working principle of Prim's is more straightforward when compared with Kruskal's which can be seen in its' pseudocode and V table items. A node with the lowest value in V table gets added to MST. I used "← min" label to dynamically indicate which value is currently the lowest.

If we take a look at Figure 21, which depicts the state of V table corresponding to the graph illustration from Figure 17, we can notice that the next node that will be added to MST is node "e". It has the lowest positive value in V table, which can be verified by looking at the graph, where the blue edge with the lowest weight is "f-e" (it has weight of 10).

Figure 21. V table - Prim's.

Figure 22. Pseudocode graphics item - Prim's.

It is worth to notice that in the first step of Prim's algorithm a random node can be chosen. That is how the program is implemented, it selects a random node from the graph. However, for the convenience of the user, before the first step is performed, it is possible to select specific node by clicking on it, this way we can force the algorithm to start from this specific node. It can be useful if the user wanted to make some teaching point that requires starting from specific node. The selected node is indicated by the thickened node boundary (e.g. node "d" in Figure 23).
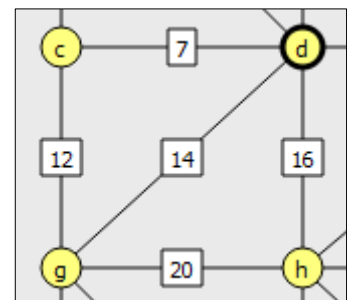

Figure 23. Selection of starting node ("d") - Prim's.

Both algorithms, Kruskal's and Prim's in the end produce the **green** coloured MST (Figure 24).
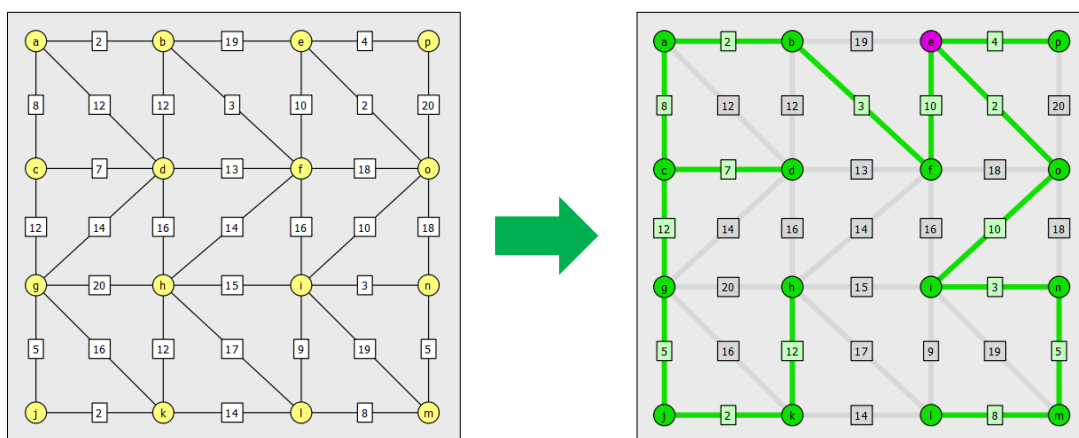

Figure 24. Graph at the start and end of MST algorithms (Kruskal's and Prim's).

10

3.5 Dijkstra algorithm implementation

Dijkstra algorithm is a single source shortest path algorithm, it finds optimal routes to each vertex in a graph starting from the same vertex (single source).

In Figure 25, node "a" is the source, which is also shown in the legend graphics item. Blue numerical values in brackets are total distances from the source node (e.g. " 7 (9) " means that distance between "a" and "d" is 9)
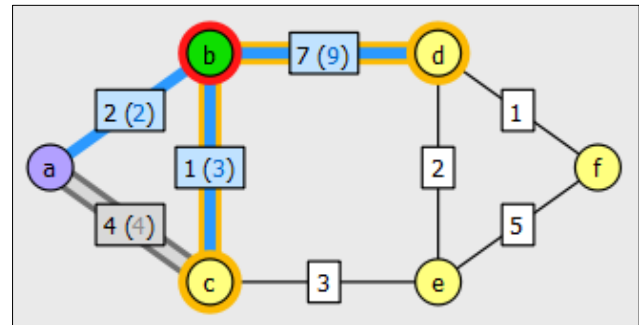

Figure 25. A coloured graph during Dijkstra algorithm processing.
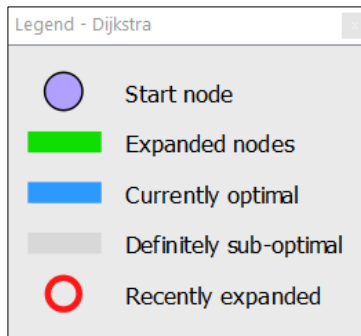

Figure 26. Legend graphics item - Dijkstra.

Dijkstra algorithm working principle is similar to Prim's, the difference is that Dijkstra algorithm determines which node to expand basing on accumulated costs (all the way from the source node), instead of individual edge cost.

Because of that, it happens to update the shortest distances of nodes that their distances calculated by the algorithm, that procedure is also known as "relaxation" [26]. The graph from Figure 25 is intentionally designed to present it.

Despite relatively low size of the graph, "relaxation" occurs 3 times altogether when using "a" as the source node. This is neatly represented in appearance of the graph in the program, as shown in Figure 27. Table 5 lists the changes that occurred by "relaxing" 3 different nodes.

Node	Before	After
c	4	3
d	9	8
f	11	9

Table 5. Nodes upon which relaxation was done. In other words, new (shorter) paths were discovered to them.
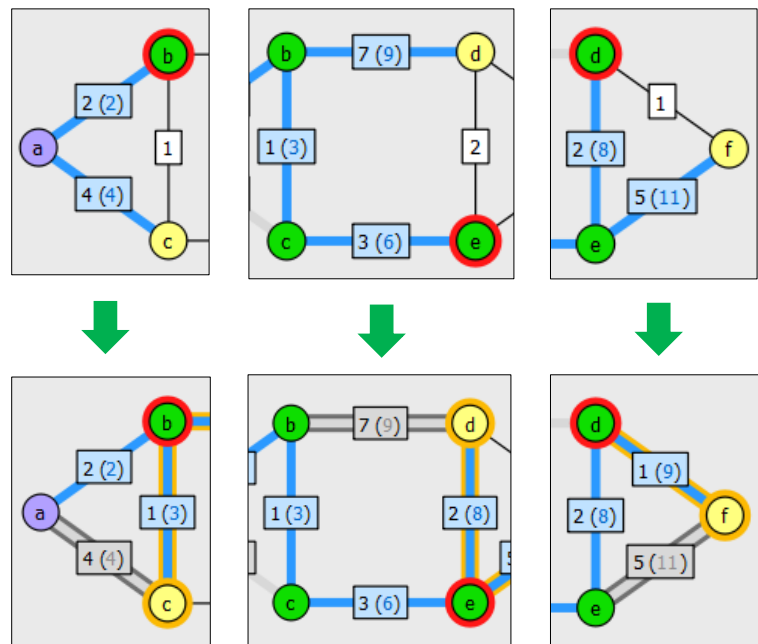
Figure 27. Three "relaxations" (nodes: c, d, f) occurring during different steps of Dijkstra algorithm processing (grey edges become sub-optimal routes).

Efficient implementation of Dijkstra algorithm makes use of priority queue data structure. In this case the queue contains nodes with their total distance from the source node, instead of edges (as it is for the Kruskal's algorithm). The state of priority queue from Figure 28 indicates that node "c" will be expanded next.

Figure 28. Priority queue graphics item - Dijkstra. It contains nodes and their total distances to the source nodes (the state of the queue corresponds to the graph presented in Figure 25).

The working principle of Dijkstra algorithm is presented in the pseudocode window. This, accompanied with the V table window (displaying the underlying key data changes) and visual graph effects, creates near-optimal learning resource to grasp the general principle of Dijkstra algorithm, and also to understand how it can be implemented in code (to work as an automated solution).

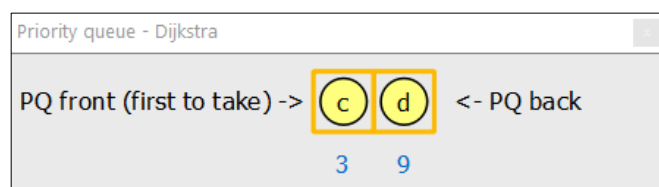In both, priority queue, and V table, entries of which values were modified during the last step get marked using orange rectangles.

Figure 29. V table - Dijkstra.

Figure 30. Pseudocode graphics item - Dijkstra.

Depending on the progress of algorithm, four different labels and descriptions may be displayed.

Figure 31. Starting node selection step label and description - Dijkstra.

Figure 32. Expansion of a node step label and description - Dijsktra.

Figure 33. Selection of a node step label and description - Dijsktra.

Figure 34. The final step of Dijkstra algorithm.

Keeping the colouring convention consistent, the final result of Dijkstra algorithm has a green colour (Figure 35). Green colour is not used from the beginning (as it is for Prim's) because in case of Dijkstra, the paths considered optimal may change as the algorithm is progressing.
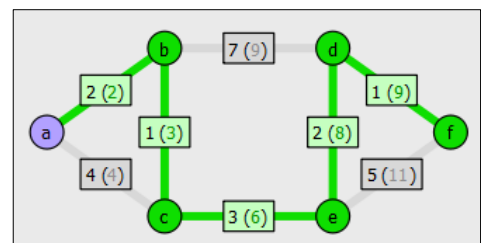
Figure 35. The end result of Dijkstra algorithm.

12

3.6 Breadth-first search (BFS) implementation

Breadth-first search can be used to find a node, or to assign a distance to each node from the selected source node. It works very much like a variant of Dijkstra that pretends all edges have weights equal to 1. When deciding which node to "expand", BFS prioritizes nodes seen the earliest, that is why the queue data structure is appropriate for its' implementation.

Figure 36. A coloured graph during breadth-first search.

Figure 38. Legend graphics item - BFS.
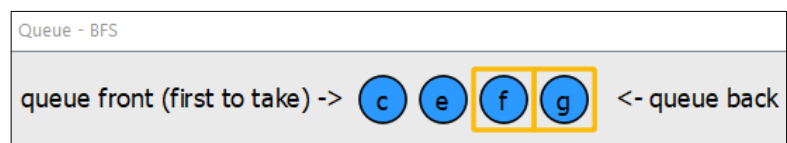
Figure 37. Queue graphics item, displaying which nodes will be expanded next.

When a node is expanded, all its' **neighbours** (that were not seen by the search yet) get assigned 2 values:

- distance from the source node (in terms of node count)
- reference to the parent node

In Figure 36 the node "**d**" got expanded, resulting in "**f**" and "**g**" being added to the queue, as shown in Figure 37. The example graph used above is a tree (it does not contain cycles), therefore all edges are used for backtracking.

A different example is shown in Figure 39 (on the right). In that example, the graph contains cycles, therefore some edges are not used for backtracking. These edges are coloured with grey colour. Just as it was in the previous example, green rectangles (with numbers inside) indicate the distance from the source node ("**h**" in this case).

Unlike Prim's, Kruskal's, or Dijkstra algorithms, the BFS algorithm ignores edge weights (if there are any) and defines distance in terms of node count from the source. For that reason, the original edge weights get hidden when BFS algorithm is selected.
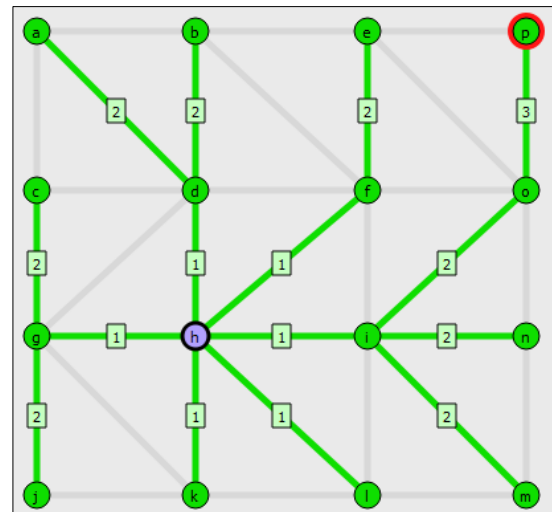
Figure 39. Graph (containing cycles) resulting from the completed BFS. Grey edges are not used for backtracking.

The pseudocode and V table windows provide insight into algorithm from the perspective of the implementation. Each V table entry contains 2 previously mentioned values that get overwritten as their parent node gets expanded. For example, the state of V table from Figure 40 (which corresponds to the state of graph from Figure 36), indicates that nodes "f" and "g" are 2 edges apart from the source node ("a"), and their common parent node is "d".

V table - BFS

V[a] = (0, -)
V[b] = (1, a)
V[c] = (2, b)
V[d] = (1, a)
V[e] = (2, b)
V[f] = (2, d)
V[g] = (2, d)
V[h] = (2147483647, -)
V[i] = (2147483647, -)
V[j] = (2147483647, -)
V[k] = (2147483647, -)
V[l] = (2147483647, -)
V[m] = (2147483647, -)
V[n] = (2147483647, -)
V[o] = (2147483647, -)

Figure 40. V table - BFS.

Pseudocode - BFS

```
for v in all_nodes:
    pred[v] = -1 # keeps track of preceding node
    dist[v] = INT_MAX # keeps track of depth
    state[v] = UNKNOWN

Q = queue()
Q.enqueue(start_node)

while not Q.is_empty():
    u = Q.dequeue() # node to expand
    state[u] = EXPANDED
    for v in u.neighbors(): # surrounding nodes
        if state[v] == UNKNOWN:
            dist[v] = dist[u] + 1
            pred[v] = u
            state[v] = SEEN # avoids adding same node twice
            Q.enqueue(v)# add to the end of queue
```

Figure 41. Pseudocode graphics item - BFS.

Depending on the last processed step, the label and description areas are appended with one of the following 3 possible entries.

Start node 'a' selected	Start node (a) was selected. The algorithm will search through all the nodes, prioritizing the
Adjacent nodes of 'a' updated	closest nodes first.

Figure 42. Label and description following the initial step of BFS algorithm.

Start node 'a' selected	'b' was selected to be expanded. That is because it was marked as 'seen' the earliest from
Adjacent nodes of 'a' updated	all 'seen' nodes.
Node 'b' selected	

Figure 43. Label and description following selection of a new node to expand. The node gets "dequeued" from the queue.

Start node 'a' selected	Adjacent nodes of selected node 'd' were updated. In other words, the node 'd' was
Adjacent nodes of 'a' updated	expanded. All neighbors of node 'd' that were not expanded yet, were marked as 'seen'.
Node 'b' selected	
Adjacent nodes of 'b' updated	These nodes were:
Node 'd' selected	f, g
Adjacent nodes of 'd' updated	

Figure 44. Label and description following updating entries of neighbouring nodes that were not expanded yet..

14

3.7   Depth-first search (DFS) implementation

The DFS algorithm is very similar to BFS, but instead of expanding all the closest nodes first, it expands nodes as it "sees" them. It can be implemented in different ways. One of them involves using recursion. Another way is almost identical to BFS implementation with a stack data structure instead of queue, and that way is what I implemented in the program.

Figure 45 illustrates an example graph looks like when processing depth-first search algorithm. Representation of DFS in the program is very similar to BFS. The only differences are:

- graph colouring order, and the different final result which comes with it (presented in Figure 48)
- different variant of the "collection graphics item" (stack instead of queue, see Figure 47)
- minor changes in the pseudocode (Figure 46)
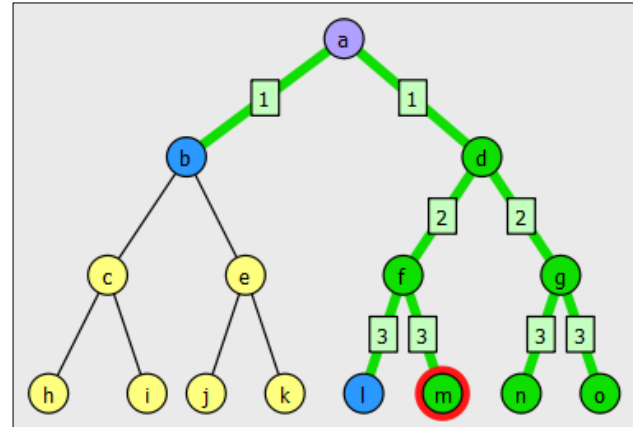
Figure 45. A coloured graph during depth-first search.
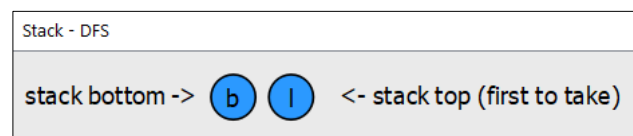
Figure 47. Stack graphics item, unlike queue, the next node to be expanded is located on the right (top of the stack).
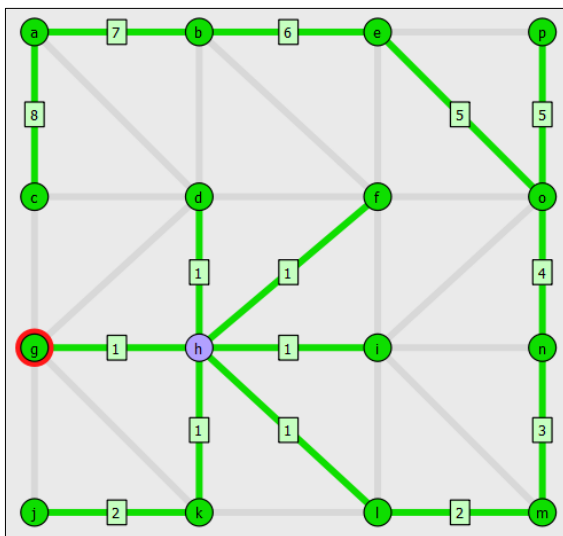
Figure 48. Final result of DFS applied on a graph that is not a tree (has cycles).

As shown on above image, the final result of DFS is different from BFS when applied on a graph that is not a tree (a graph that has at least one cycle). In this case, one node was found as distance of 8, whereas BFS farthest node was found at distance of 3 in the same setup.


```
Pseudocode - DFS

for v in all_nodes:
        pred[v] = -1 # keeps track of preceding node
        dist[v] = INT_MAX # keeps track of depth
        state[v] = UNKNOWN

S = stack()
S.push(start_node)

while not S.is_empty():
        u = S.pop() # node to expand
        state[u] = EXPANDED
        for v in u.neighbors(): # surrounding nodes
                if state[v] == UNKNOWN:
                        dist[v] = dist[u] + 1
                        pred[v] = u
                        state[v] = SEEN # avoids adding same node twice
                        S.push(v)# equivalent to appending to list
```
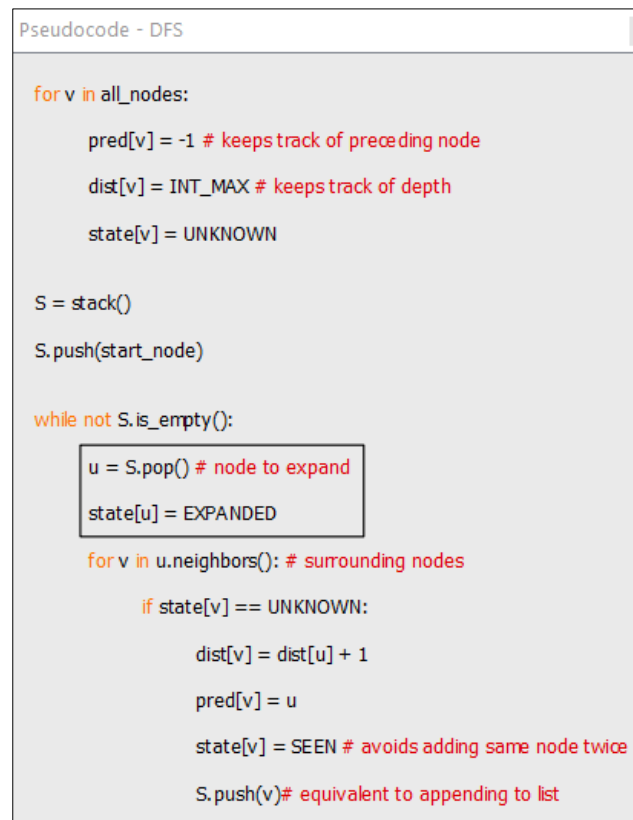
Figure 46. Pseudocode graphics item - DFS.

3.8   User interface

Demonstrative animations of user interface features are available at GitLab repository [22]. Initially, positions and sizes of graphical items (e.g. pseudocode, V table) were hardcoded in the program. The program displays large volume of information, so as I added more features (that took more space), the space management gradually started becoming a problem. For that reason, at some point, I decided to address that problem by introducing resizable and movable sub-windows ("docks"), each containing an independent graphical item (see Figure 49) that scales upon resizing.
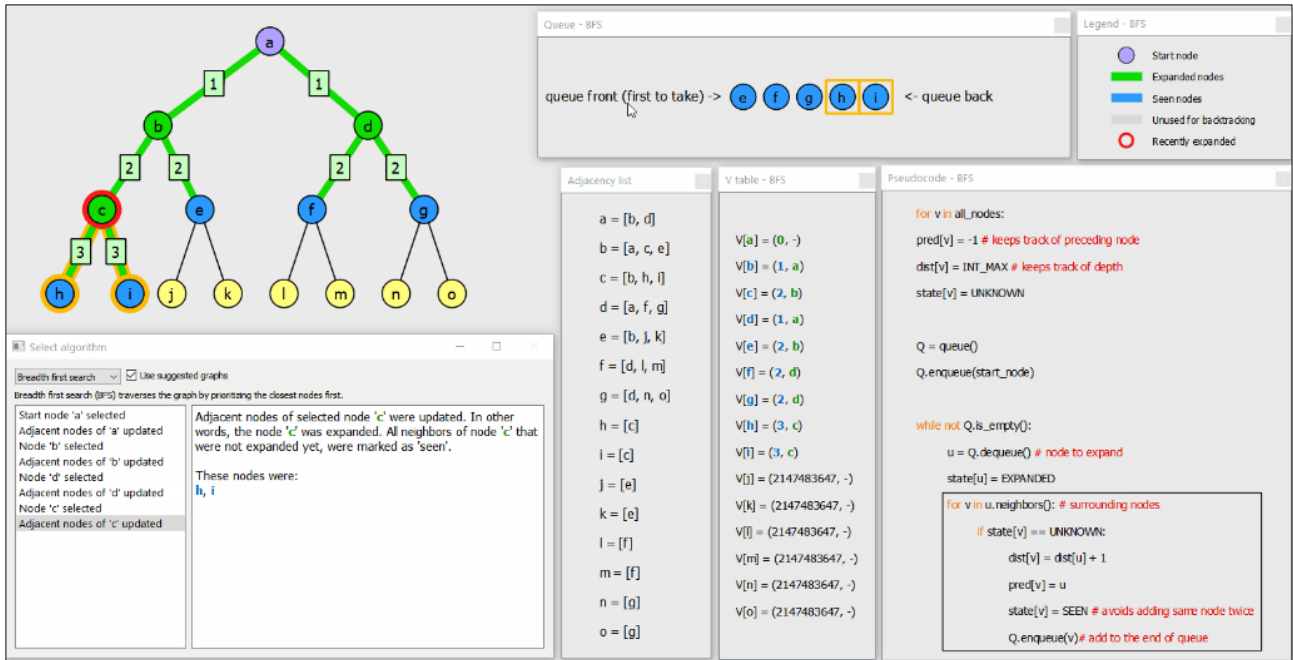
Figure 49. Sub-windows ("docks") based user interface.

Other user interface features introduced for convenient use include:

▪ Zooming (using mouse wheel) and easy movement of the whole graph or individual nodes. Individual nodes get aligned with certain granularity to allow aesthetic positioning, ("Ctrl" key can be used for fine positioning when needed).
▪ The graph is preserved after restarting the program (including positioning and zoom level).
▪ The graph is kept in-sight when resizing the main window.
▪ Placing cursor near an edge makes that edge temporarily thicker (for easy selection, it is illustrated in Figure 50).
▪ Node letters (labels) are alphabetically incremented, letters are reused in alphabetical order after removing them.
▪ Edge weights can be edited by clicking directly on them.

Figure 50. Selectable edge area when under cursor (marked with blue).

▪ Entering new edge weights preserves previous value if completely invalid value is entered. If the user accidentally enters a letter or symbol within the numeric value, then only the numeric value stays.

It is rare to find any of features listed above in existing solutions (that were outlined on page 1).

3.9   Program controls

Action	Control
Step through algorithm	Right-arrow key
Select node/edge	Left click
Deselect node/edge	Esc key
Node movement	Left click and drag (Ctrl key may be kept down to disable auto-alignment)
Edit edge weight	Left click (on weight rectangle)
Adding a node	Shift + left click on blank space (source node must be previously selected)
Adding an edge	Shift + left click on another node (source node must be previously selected)
Remove a node Remove an edge	Delete key (node/edge must previously be selected)
Saving a graph	Right click on a graph (in the main window)
Loading a graph	Left click on a graph (in the "Select graph" window)
Removing a saved graph	Right click on a graph (in the "Select graph" window)
Zoom	Mouse wheel (over the main window)
Reset algorithm	Esc key

Table 6. Program controls table.

4. Implementation details

4.1 Technology that was used

The program was written in C++ using Qt framework. Qt has a lot of convenient utilities, handy types for data structures, and a powerful GUI designer called "Qt Creator". I used the following software versions:

- Qt Creator 4.13.1 IDE
- Qt 5.15.0 framework
- MinGW 8.1.0 64-bit compiler

The Project Handbook [21] suggests to explicitly state what technical advancements were created by me entirely from new. Therefore, I find it appropriate to first reflect on what exact utilities Qt framework brings.

Qt offers the following 3 key classes to organize custom 2D graphics:

Class	Description
QGraphicsView	**View** acts as a host widget and a window for a scene, allowing for "pan and crop" kind of operations (and much more) of the scene (if someone is familiar with video editing)
QGraphicsScene	**Scene** acts as a coordinate and transformation system for items, items can be added and removed from a scene. A scene can be assigned to a graphics **view**.
QGraphicsItem	I**tems** are like "sticky notes" on the scene.

Table 7. Three key Qt classes responsible for organizing 2D graphics.

Additionally, it has many built-in subclasses of QGraphicsItem to represent various objects.

Class	Description
QGraphicsTextItem	Drawing text (that can be coloured and styled with HTML).
QGraphicsSimpleTextItem	Drawing text (less styling options, but more efficient).
QGraphicsLineItem	Drawing lines.
QGraphicsRectItem	Drawing rectangles (including rounded rectangles).
QGraphicsEllipseItem	Drawing circles/ellipses.
QGraphicsPixmapItem	Drawing images (I used it in graph selection widget).
QGraphicsPolygonItem	Drawing polygons.
QGraphicsPathItem	Drawing custom shaped items that may consist of mixture of any items above (any item can be converted to path itself and added to QGraphicsPathItem).

Table 8. Sub-classes of QGraphicsItem provided by Qt to represent various objects.

These classes are convenient, and I used many of them in this project, but in most cases I sub-classed the "QGraphicsItem" class itself and overridden 2 key virtual methods provided by it ("boundingRect" and "paint"). By specifying my own definition of paint method, I was able to create custom effects for graphs, such as node highlighting (e.g. because it was recently expanded in some algorithm), or varying brightness of item under the mouse cursor (to make it look responsive).

Using the "2D graphics toolkit" described above, the Qt Creator graphical designer, occasional google/stackoverflow search, and resources outlined in the "Acknowledgements" section of this report, I created every feature mentioned in "Description of the system" (page 6) and "User interface" (page 16)

sections from scratch in general. Below I present a list of code parts that were either copied, or based on code written by other people:

I directly copied the **qt_graphicsItem_shapeFromPath** function (located in functions.cpp of my project) from the qgraphicsitem.cpp [27] which is a part of Qt source code. This function allowed me to generate a thick line shape. I overrode the virtual "shape" method of the Edge class, and returned **qt_graphicsItem_shapeFromPath** function from it, this way the Edge responds to mouse cursor that is in 10-pixel proximity (cursor does not have be directly over the thin line, which would be inconvenient).

I used an answer from "Round to nearest multiple of a number" stackoverflow topic [28] in the **getClosestPoint** function (in functions.cpp file of my project). I modified it to handle negative numbers.

Node, Edge and GraphicsView classes were initially part of the "Elastic Nodes" example [29] provided by Qt, however I modified them beyond recognition.

Table 9. Parts of the code that were either copied or based on code written by other people.

4.2 Design patterns

It is evident from previous sections of this document that a lot of functionality is recurring among different algorithms. Thanks to object-oriented design, I was able to reuse a lot of code to implement common behaviour and still maintain high level of flexibility (in terms of behaviour of specific algorithms). The diagram below shows inheritance of QGraphicsItem as used in this project.
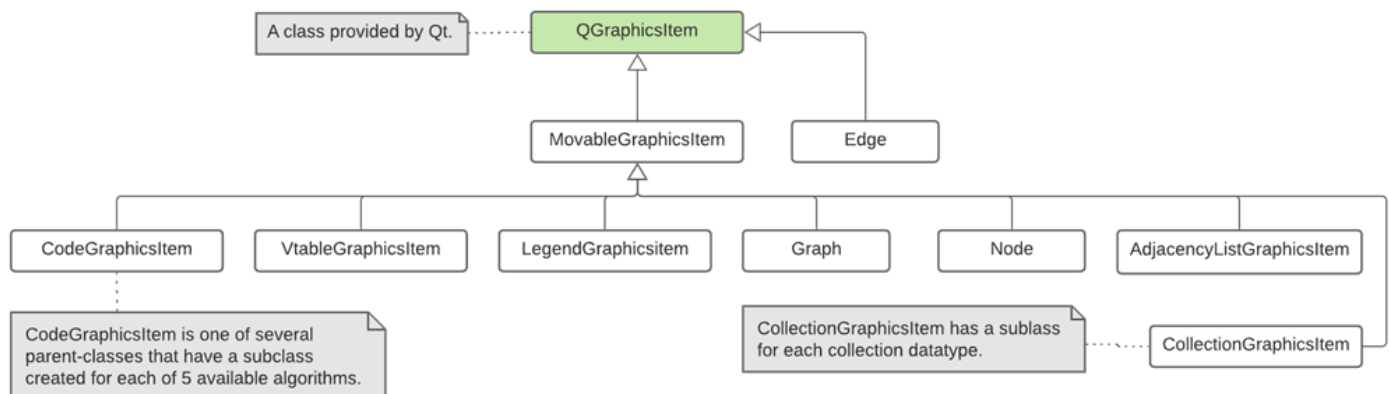
The CollectionGraphicsItem currently has 3 subclasses used to implement 3 datatypes (PriorityQueue, Queue, Stack) which can be seen in the diagram below.

19

As mentioned in one of the grey notes, the CodeGraphicsItem is one of parent-classes that have a subclass created for each of 5 available algorithms. In the following diagrams I will present what are these classes and what reusable functionality they bring.

Abstract class implementing common behaviour. To avoid duplication. Handling painting and geometry of all sub-classes using:

```
virtual void paint(...) override;
virtual QRectF boundingRect() const override;
```

And forcing sub-classes to implement:

```
getTextListItemsForPreviousState(int state) = 0;
```

CodeGraphicsItem

- PrimsCodeGraphicsItem
- KruskalsCodeGraphicsItem
- DijkstraCodeGraphicsItem
- BfsCodeGraphicsItem
- DfsCodeGraphicsItem
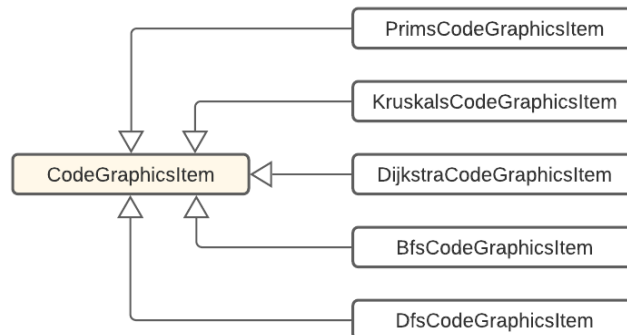
Having separate Code item for each class and having explicit object for each line of code (or group of lines) allows to implement highly customized behaviour for specifc algorithm and specific part of code.

Abstract class implementing common behaviour and forcing sub-classes to have the same structure, containing methods like:

```
virtual void onAlgorithmStart() = 0;
virtual void onAlgorithmStep() = 0;
virtual void onAlgorithmEnd() = 0;
virtual void onAlgorithmEvent() = 0;
```

And handling mouse/key events for all subclasses using:

```
virtual void mousePressEvent(...) override;
virtual void mouseReleaseEvent(...) override;
virtual void keyPressEvent(...) override;
```

AbstractScene

- PrimScene
- KruskalScene
- DijkstraScene
- BfsScene
- DfsScene
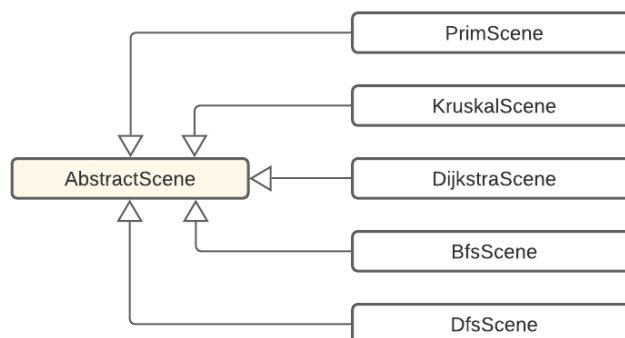
Derived classes implementing custom behaviour.

For example:
- Dijkstra is using priority queue graphics item for nodes.

- Kruskals is using priority queue graphics item for edges.

Abstract class implementing common behaviour and forcing sub-classes to have the same structure, containing methods like:

```
virtual void reset();
virtual void step() = 0;
virtual QString name() = 0;
virtual QString brief() = 0;
virtual QString currentStepLabel() = 0;
virtual QString currentStepDescription() = 0;
virtual bool finished() = 0;
```
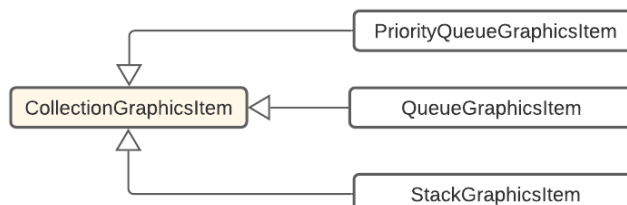
ControlledAlgorithm

- ControlledPrimsAlgorithm
- ControlledKruskalsAlgorithm
- ControlledDijkstraAlgorithm
- ControlledBfsAlgorithm
- ControlledDfsAlgorithm
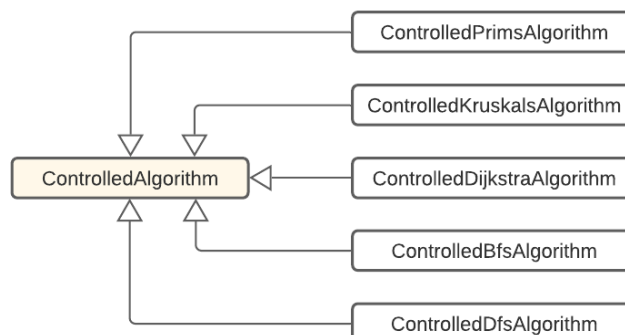
Derived classes implementing custom behaviour used to compute the results and state of algorithms.

These classes contain the logic for algorithms.

Instead of mixing algorithm logic with graphical representation I separated implementation of these into separate classes/files, as shown in the diagram below.

Has   **Each algorithm**   Has

Has

Logic

In order to present inner workings of algorithms, the algorithms themseleves need to be implemented in the program.

This is done in "**ControlledAlgorithm**" class and its subclasses for each algorithm.

Scene

Scene in Qt is a set of graphical items and their coordinates, it can handle user input.

This is done in "**AbstractScene**" and its subclasses for each algorithm.

Own set of **graphical items** spawned by the scene.

For example:
- code item
- legend item
- priority queue item
- v table item

The use of such design patterns was not accidental, I have spent a lot of time thinking on how to design the program in a way to avoid difficult to manage "spaghetti code", avoid code duplication, but at the same time make it flexible enough to easily implement custom behaviour of each algorithm. That was a challenge, it led to many issues when I started development of 2nd algorithm, but in the end I think that careful design saved me a lot of time overall despite being a time-consuming process itself. That is because after implementing a robust infrastructure for 2 algorithms, adding the remaining 3 algorithms was relatively straightforward.

4.3 File structure

This project source code contains around 10 thousand lines of code written in 91 different "cpp" and header files. From the beginning, I tried to manage the file structure to allow convenient navigation and to avoid "getting lost" in my own creation. In Figure 51 I used different colours to mark files located in separate folders. These files tend to contain recurring design patterns (subclassing a common parent class in order to avoid code duplication and force consistency).


```
adjacency_list_graphics_item.cpp          code_items/kruskals_code_graphics_item.h        graph/graph.h                                scenes/dijkstra_scene.h
adjacency_list_graphics_item.h            code_items/prims_code_graphics_item.cpp         graph/node.cpp                               scenes/kruskal_scene.cpp
algorithms/controlled_algorithm.cpp       code_items/prims_code_graphics_item.h           graph/node.h                                 scenes/kruskal_scene.h
algorithms/controlled_algorithm.h         collection_items/collection_graphics_item.cpp   graph/preset_graphs.cpp                      scenes/prim_scene.cpp
algorithms/controlled_bfs_algorithm.cpp   collection_items/collection_graphics_item.h     graph/preset_graphs.h                        scenes/prim_scene.h
algorithms/controlled_bfs_algorithm.h     collection_items/priority_queue_graphics_item.cpp  graphics_view.cpp                         select_algorithm_widget.cpp
algorithms/controlled_dfs_algorithm.cpp   collection_items/priority_queue_graphics_item.h graphics_view.h                              select_algorithm_widget.h
algorithms/controlled_dfs_algorithm.h     collection_items/queue_graphics_item.cpp         legend_graphics_item.cpp                    select_graph_widget.cpp
algorithms/controlled_dijkstra_algorithm.cpp  collection_items/queue_graphics_item.h       legend_graphics_item.h                      select_graph_widget.h
algorithms/controlled_dijkstra_algorithm.h    collection_items/stack_graphics_item.cpp     line_description_graphics_item.cpp          shortest_path_graphics_item.cpp
algorithms/controlled_kruskals_algorithm.cpp  collection_items/stack_graphics_item.h       line_description_graphics_item.h            shortest_path_graphics_item.h
algorithms/controlled_kruskals_algorithm.h    colors.h                                     main.cpp                                    step_description_graphics_item.cpp
algorithms/controlled_prims_algorithm.cpp     dock.cpp                                      mainwindow.cpp                              step_description_graphics_item.h
algorithms/controlled_prims_algorithm.h       dock.h                                       mainwindow.h                                tutorial.cpp
code_items/bfs_code_graphics_item.cpp         functions.cpp                                movable_graphics_item.cpp                   tutorial.h
code_items/bfs_code_graphics_item.h           functions.h                                  movable_graphics_item.h                     typedefs.h
code_items/code_graphics_item.cpp             graph/custom_graphs.cpp                      scenes/abstract_scene.cpp                   vtable_entry_graphics_item.cpp
code_items/code_graphics_item.h               graph/custom_graphs.h                        scenes/abstract_scene.h                     vtable_entry_graphics_item.h
code_items/dfs_code_graphics_item.cpp         graph/edge.cpp                               scenes/bfs_scene.cpp                        vtable_entry_hover_text_item.cpp
code_items/dfs_code_graphics_item.h           graph/edge.h                                 scenes/bfs_scene.h                         vtable_entry_hover_text_item.h
code_items/dijkstra_code_graphics_item.cpp    graph/edge_text_item.cpp                     scenes/dfs_scene.cpp                       vtable_graphics_item.cpp
code_items/dijkstra_code_graphics_item.h      graph/edge_text_item.h                       scenes/dfs_scene.h                         vtable_graphics_item.h
code_items/kruskals_code_graphics_item.cpp    graph/graph.cpp                              scenes/dijkstra_scene.cpp
```

Figure 51. File structure used in this project. Coloured rectangles signify recurring design patterns and files containing those (typically such files were placed in their own folders).

5. Project Planning

Initially I was planning to complete a computer-vision based project titled "Recognizing galaxy types", however I realized that "ambitiously" selecting a project based on unfamiliar subject may have bad consequences like:

- not being able to stay on top of other modules work
- producing a low-quality project despite putting a lot of effort into it

Selecting wrong project from the start was a major mistake, but other than that, my planning was careful and accurate, I never felt behind with the project or uncertain of what to do next.

Throughout the project I tried to get familiar with project resources/guidelines as soon as they got released (e.g. viewing all project proposals, paying attention to lectures, reading moodle resources such as project handbook or marking schemes). Thanks to that I prioritized producing deliverables required for the closest upcoming deadline.

As presented in the "Project introduction" document [30] submitted at the end of the challenge week, my plan was defined from the beginning of the project and consisted of the following key points:

- **Background reading** [31] [32] [33]. Without having substantial understanding of the topic, it would be likely to make early implementation mistakes that would be costly in later stages of the project.
- **Reading Qt documentation** [34]. Understanding algorithms and having a vision of program design are essential for this project, but knowing how to use programming language/framework to implement such design is also important. It would be a good idea to spend some time trying to play with the development environment and implement some functionality required by the program (e.g. drawing circles and lines to represent graph vertices and edges, implementing text-input fields for edge-weights).
- **Finding relevant Qt/C++ examples** [35]. Qt is a powerful framework with many ways of accomplishing the same task. So it may be a good idea to take a look at official examples provided in Qt Creator, looking for graph drawing, graph modification examples, even if such example won't be included in this project.
- **Implementing a facility to create and modify a graph** [36]. This should be done before attempting to implement an algorithm.
- **Implementing a single algorithm** [37]. This will allow to verify whether previous steps gave clear idea on how to implement the key features of the program. Upon transition to implement 2nd algorithm I could do some refactoring to promote clean code (e.g. by inheriting from a common abstract algorithm class which could provide some common interface/members for all of algorithms, to avoid code duplication).
- **Implementing all graph algorithms taught in CE204 module.** Implementing multiple algorithms will require a facility to view and change the current algorithm.
- **Implementing additional features (if there's enough time), like:**
  - saving and loading a graph from a file
  - in-built documentation explaining terminology and concepts related to graph theory
  - ability to view different types of implementation of particular algorithm
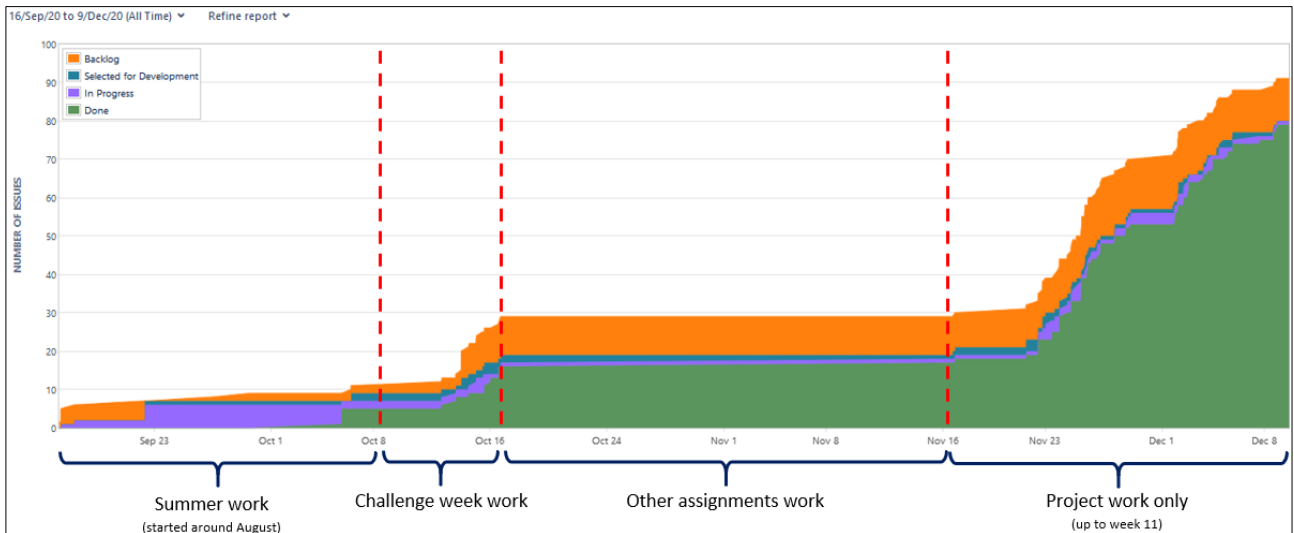 - implementing algorithms not covered in CE204

5.1 Development stages

Figure 52. Project development stages.

As described in the "project_management.md" document [38] , the work I undertaken could be split into several stages:

- Summer work
- Challenge week work
- Completing assignments of other modules
- Intense work on the project between 16th November and interim interview
- Work after interim interview

In the following subsections I will try to summarize what each stage involved.

5.1.1 Summer work

During this stage I accomplished the following objectives (evidenced mainly on Jira):

- Familiarizing with graph theory [32]
- Identifying and understanding algorithms [31]
- Recognizing and evaluating methods of teaching them [33]
- Reading Qt documentation [34]
- Finding relevant Qt examples [35]
- I undertook investigation to verify the use case of MST as a near-optimal route.
  I was intrigued by the claims that MST can be used to find near-optimal route from the article called "An Application of Minimum Spanning Trees to Travel Planning" (2010) by L. Fitina et al. [38]
  I wrote a code to verify accuracy of few statements from that article and presented my findings on page 14 of Project introduction document [30]. My findings are presented in Table 10.

Start node	Optimal path	Opt. path length	MST path	MST path length	MST overhead
MAD	MAD - POM - LAE - HKN - RAB - BUA - KAV - MAS	3824	MAD - POM - LAE - HKN - RAB - BUA - KAV - MAS	3824	0%
KAV	KAV - MAS - POM - MAD - LAE - HKN - RAB - BUA	4147	KAV - MAS - RAB - BUA - HKN - LAE - POM - MAD	4318	4%
MAS	MAS - KAV - MAD - POM - LAE - HKN - RAB - BUA	3668	MAS - KAV - RAB - BUA - HKN - LAE - POM - MAD	3824	4%
HKN	HKN - LAE - MAD - POM - MAS - KAV - RAB - BUA	4152	HKN - KAV - MAS - RAB - BUA - LAE - POM - MAD	5153	24%
POM	POM - MAD - LAE - HKN - RAB - BUA - KAV - MAS	4275	POM - MAD - LAE - HKN - RAB - BUA - KAV - MAS	4391	2%
RAB	RAB - BUA - HKN - LAE - POM - MAD - KAV - MAS	4142	RAB - KAV - MAS - BUA - HKN - LAE - POM - MAD	4738	14%
LAE	LAE - HKN - RAB - BUA - KAV - MAS - POM - MAD	4131	LAE - MAD - POM - HKN - RAB - BUA - KAV - MAS	4850	17%
BUA	BUA - RAB - HKN - LAE - POM - MAD - KAV - MAS	3668	BUA - KAV - MAS - RAB - HKN - LAE - POM - MAD	4264	16%

Table 10. Results of investigation regarding the use of MST as an optimal path for the travelling salesman problem. In the publication by L Fitina et al. (2010) the starting node "MAD" was chosen. It can be noticed that it is the only node that resulted in optimal path. All remaining nodes would result in overhead if they were chosen, my goal was to measure the exact value of that overhead.

5.1.2 Challenge week work

During the challenge week I accomplished the following tasks:

- Formulated project objectives [39]
- Documented summer coding [40]
- Added facility to create and modify a graph [41]
- Implemented a single algorithm [42]
- Created extensive Project introduction document [30]

At the end of the challenge week I created 0.1.0 release on Jira which contains all issues completed up to the challenge week:

A301079-1 Project introduction document
A301079-2 Identify and understand algorithms
A301079-3 Recognize and evaluate methods of teaching algorithms
A301079-4 Read Qt documentation
A301079-5 Find relevant Qt examples
A301079-6 Familiarize with graph theory
A301079-7 Formulate project objectives
A301079-10 Facility to create and modify a graph
A301079-11 Implement a single algorithm.
A301079-12 Document summer coding
A301079-13 Moving vertices isn't perfect
A301079-14 Complete "Project management" section
A301079-23 Write conditional and dynamic step descriptions for Prims algorithm
A301079-25 Refactor creating "Step log" description
A301079-27 Disable docking

Figure 53. Jira issues included in the 0.1.0 release.

5.1.3   Completing assignments of other modules

Capstone project has no limit in terms of progress that can be made. So I decided to strategically postpone further work on it until I complete (or almost complete) all available assignments of other modules and afterwards focus entirely on the project. This way I could utilize all remaining time without fear of not giving other modules enough attention. This period lasted around a month, giving me another full month of project work before the interim interview.

5.1.4   Intense work on the project between 16th November and interim interview

During this time, I spent most of my free time working on the project and implemented most of the program functionality, including:

- algorithm selection facility
- 4 new algorithms
- custom graph saving/loading
- selectable preset graphs

I slightly transitioned away from carefully considering every code addition (which I did during early design stage) into adding sheer number of features. With this approach I managed to complete the great majority of program before the interim interview. The biggest challenge during that time was transitioning from single algorithm implementation into implementation of multiple algorithms, which resulted in various bugs and stability issues that I had to resolve.

On 04/12/2020, the project supervision was passed from Dr Michael Gardner to Dr Alexandros Voudouris.

5.1.5   Work after interim interview

During this time I considered the development to be finished and made only minor changes to the code (fixing small bugs/typos). I focused on producing documentation and deliverables (abstract, poster, final report), and the coursework of other modules. This way I avoided the risk of overengineering the program at expense of other parts of the coursework.

5.2   Testing

Testing involved:

- ad-hoc tests of program behaviour/functionality at different stages
- the use of debugger
- attempting to run the program on a PC without any prerequisites installed (e.g. no Qt libraries present on the system)
- getting feedback about the program from people unfamiliar with graph theory, a single CE204 Data Structures and Algorithms student and the project supervisor

More details about testing and justification of chosen testing methods is available in testing_report.md file on GitLab [43].

5.3   Risks management

Figure 54 presents my considerations (saved on Jira) regarding risks that could negatively affect the project.

In terms of motivation I managed to stay focused by creating task issues on Kanban board which helped me to remember what to do, and also motivated me to set them as "done" as soon as possible.

After implementing the first algorithm (Prims), I realized that repeating the same work for all 4 remaining ones may take a very long time. Keeping in mind how too much attention to details can be slowing down overall project progress ("dwelling on details" issue), I decided to implement "barebones" of all remaining 4 algorithms and then gradually improve each of them using remaining time. Using this approach, I was not stressed about time limits, worked more comfortably and nearly completed the program at the stage when MVP was expected (interim interview deliverable).

Completing the program relatively early was my goal because in the spring term I had more modules.

Stability of the program was also a risk, especially that C++ is a language without garbage collection, meaning that the memory has to be managed explicitly by the programmer which makes it error prone. For that reason, it was not uncommon for the program to crash during development, however I managed to solve all of such issues by effectively using debugger which in most cases pointed to the source of the problem (e.g. displaying "null" or "dangling" pointer value with a corresponding list of previously executed instructions and function calls).
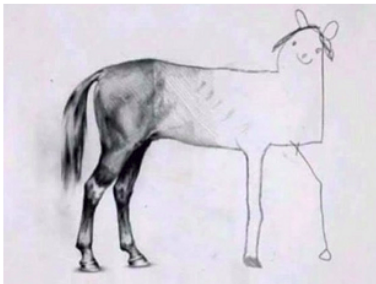

Figure 54. Risk-related issues from Jira.

6. Use of project management tools (GitLab, Jira)

It is evident from previous sections that I used GitLab and Jira project management tools. Being aware of their importance in the marking process, I would like to further examine what these tools are and how exactly I used them for this project.

6.1   What GitLab is

All that is needed to create software is a text editor and a compiler. If humans were perfect and were able to plan/implement software in impeccable way straight from the beginning, then this kind of development environment could be sufficient. However, that is not the case, plans often turn out to be wrong, mistakes are made, changes and objectives are forgotten. Having a tool that would keep track of these and allow management would enhance productivity. That is what GitLab can do. It is a web-based platform for production of software, allowing to store files, commit changes, view, or revert them later, create different project branches and possibly merge them later to implement specific features (and much more). A great benefit of using such tool is how it helps with collaborative development. For example, in case of CE301 project, it allows the project supervisor and module coordinators to check on progress.

6.2   How I used GitLab

I downloaded "Git Bash" software and cloned the GitLab repository (previously created by the CSEE department) to my PC. This way I was able to make changes on files locally and upload them all at once. During development, after completing work on some feature (or after making significant progress), I used the following commands to keep my GitLab repository up to date:

- git add *relevant_file_or_files*
- git commit -m "message describing changes"
- git push

I tried to use meaningful descriptions for commit changes (e.g. Figure 55), so I could easily find specific changes done in the past.

GitLab supports "markdown" files (files with ".md" extension). Such files are text files with simple syntax used for styling. It is a common practice to create repository title page by placing "README.md" file in its' root directory [44]. That is what I did, my "README.md" file includes:

Figure 55. Example GitLab commits.

- description
- installation instructions
- links to documentation (categorized depending on deliverable, to be easily found by the supervisor and second assessor)
- preview
- author name and student ID

I think markdown files are great way to produce documentation. None of Microsoft Office products I know of are able to directly incorporate animations within documents. That, plus an ability to nicely format code snippets and insert hyperlinks to different parts of a markdown file makes it superior to other alternatives in my eyes. For that reason, all documentation I created for this project was in the form of markdown files:

- features_documentation.md [45]
- implementation_report.md [46]
- project_management.md [47]
- program_controls.md [48]
- testing_report.md [43]

Each of these files has a list of contents at the top with hyperlinks to each section. This way, a reader can click on section name and easily navigate to any part of the document.

Technical documentation is presented as a list of animations with descriptions and code snippets for each noteworthy feature. Figure 58 displays one of such features presented in the "features_documentation.md" file. The animation shows what keys are typed, that is why "Shift" label happens to be displayed on the screenshot.

Documented features are grouped into few sections:

- User interface features
- Graph features
- Algorithm features

On top of these, the "features_documentation.md" file includes animations of each algorithm being processed.
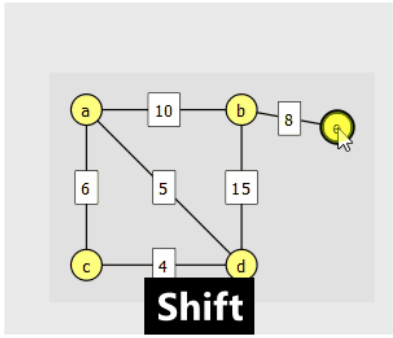
I paid attention to readability of documentation files and resized images using html syntax supported by markdown on GitLab. In some cases, I additionally enclosed the "img" tag with "kbd" tag, which is a nice trick to surround an image with a gray border.

```
<kbd><img src="url" width=400 /></kbd>
```

Another useful GitLab feature is ability to create "tags" and "releases". This allows to mark specific points in development history and keep a copy of selected files. I used it to create the "MVP" tag and the corresponding release. This way it could be examined by anyone in the future if needed.

An alternative of creating release for purposes mentioned above, would be to create "permalinks" (to keep the state of a file from specific point in

Figure 58. Example animation, description and a part of the code snippet used to document management of vertices and

Figure 57. "MVP" GitLab release I created.

history). However, this would be not practical because a permalink can be created for 1 file at a time. That is why tags and releases are very useful.

Adding a "LICENSE" file in the root directory is also a common practice [49]. GitLab software recognizes it and displays the type of license on the front page of repository (Figure 59). I would be happy if people used my code in their projects, regardless whether these projects are commercial or not. That is why I chose to use MIT license, which in simple words states: "use it for any purposes, as long as you accredit the author", it also mentions that I will not be liable for any potential damage caused by using this software. Although Qt itself is licensed with LGPL, it does not interfere with me using MIT license as stated in "stackoverflow.com" topic available at this link [50].

Figure 59. License type is displayed on the front page of repository, along with the "README" file reference.

It is likely that the only people who accessed my GitLab repository were me, the supervisor, the second assessor and possibly module coordinators. For that reason, I wanted to clearly indicate which files are created for which deliverable, so I created the following folders:

- challenge week
- interim interview
- poster and abstract
- final report

The source code is kept inside "src" folder, which is another common practice when working with git. At the bottom of Figure 60, there is "release.zip" file, it contains a deployed version of the program, ready to be unpacked and executed on any PC with modern Windows operating system (which is described in the "Installation" section of "README.md" file).
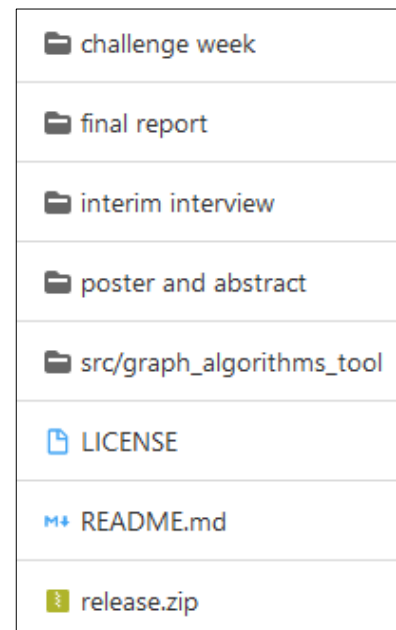

Figure 60. Top-level repository files..

6.3 What Jira is

Jira is also a web-based tool, it allows to organise work (e.g. distribute tasks among a group of people). It can be used to facilitate agile methodology, for which it provides Scrum and Kanban boards. On Jira, project workload is divided into units called "issues". These can have various types, as shown in Figure 61. Issues of "Epic" type could be treated as major milestones. "Stories" are objectives from the perspective of different parties (e.g. users or specific type of users, developers). "Tasks" are pieces of work to be completed that typically do not fit into other categories. "Bugs" usually are relating to software faults that need to be addressed. "Risks" may refer to any aspects threatening the success of a project or the quality of software.
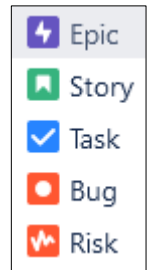
Figure 61. Jira issue types.

6.4 How I used Jira

At the beginning of the project, a Kanban board was created on my Jira account. It contains 4 columns called:

- backlog
- selected for development
- in progress
- done

I found the Kanban board helpful at keeping track of ideas and prioritizing them when needed (e.g. to produce some deliverable before deadline). Issues created by me, precisely reflected the development stage of the program. In most cases, when setting the status of a completed task or fixed bug issues, I attached a link to relevant GitLab commits, sometimes I also added an image of the implemented feature or a code part responsible for a bug (as shown in Figure 63).

Figure 62. Quantity and distribution of Jira issues I created.

Figure 63. Comment added to Jira issue following successful resolution of a bug [29].

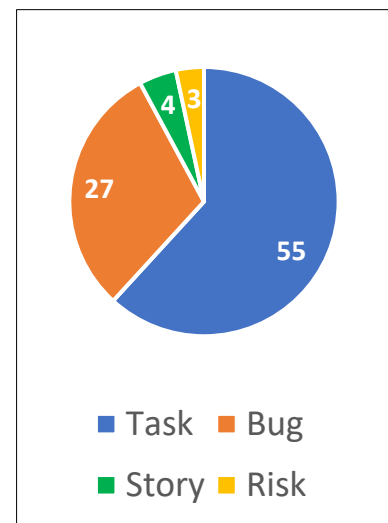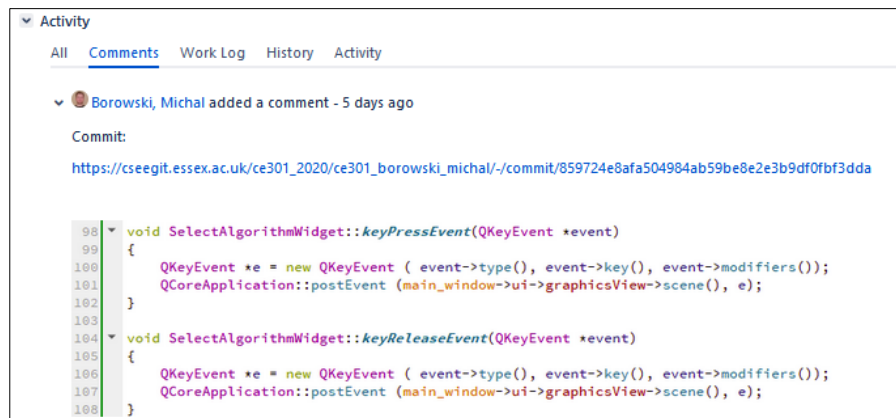| ☑ A301079-87 Graph to json	☑ A301079-61 Modify VtableGraphicsItem	☑ A301079-51 Simplify code item colouring
☑ A301079-86 Learn how size policy works in Qt	☑ A301079-60 Recursive function depth indicator for DFS	☑ A301079-48 Create abstract code graphics item
☑ A301079-81 Export graph	☑ A301079-59 Checkbox - "use suggested graphs"	☑ A301079-47 Step log store/restore geometry
☑ A301079-79 Collection graphics item to store edges (not only nodes)	☑ A301079-58 BFS code graphics item	☑ A301079-45 Dijkstra algorithm code graphics item
☑ A301079-78 Independent geometry of different algorithm docks	☑ A301079-57 BFS step log descriptions	☑ A301079-40 Ability to start algorithm from the begining
☑ A301079-73 Abstract common behaviour from QueueGraphicsItem	☑ A301079-56 Hide edge weights for BFS	☑ A301079-38 Dijkstra algorithm dock with shortest paths to nodes
☑ A301079-71 Kruskals - Differently coloured sub-graphs	☑ A301079-55 Kruskal's algorithm	☑ A301079-37 Dijkstra algorithm graph colouring
☑ A301079-70 Tidy docks code	☑ A301079-54 Depth first search (DFS) algorithm	☑ A301079-36 Dijkstra algorithm logic
☑ A301079-66 Dijkstra - add total weight to edge labels as it progresses	☑ A301079-53 Breadth first search (BFS) algorithm	☑ A301079-35 Dijkstra algorithm
☑ A301079-65 Preset graphs	☑ A301079-52 Node collection graphics item	

Figure 64. Example Jira task issues. [51].

🟥 A301079-50 Prims code item misbehaves with current state (it expects previous state)	🟥 A301079-90 Node letters not reused after deletion
🟥 A301079-46 Prims step log showing "0" as lowest positive value	🟥 A301079-89 Exported graphs not being centered
🟥 A301079-43 Code graphics item does not change code part highlighting	🟥 A301079-82 Graph changing issues
🟥 A301079-42 Pageup/pagedown moves graph out of sight	🟥 A301079-80 Docks/step-log not passing key presses
🟥 A301079-39 Items resized in docks after setting algorithm back to Prims	🟥 A301079-77 PriorityQueueGraphics item needs to remove retired nodes
🟥 A301079-34 Blocked graph editing after changing algorithm	🟥 A301079-76 Node collection position is moving/skipping
🟥 A301079-33 Zoomed-in "code graphics item"	🟥 A301079-72 V table not holding constant boundingRect size
🟥 A301079-32 Changing algorithm (creating new scene) moves the viewport to 0,0 coordinates	🟥 A301079-69 Crash if step log window is closed and algorithm tries to proceed
🟥 A301079-31 Docks don't preserve size when changing algorithm	🟥 A301079-67 Dijkstra - suboptimal edges showing incorrect total distance
🟥 A301079-30 Closing main window doesn't terminate program if step log window is open	🟥 A301079-62 Deleting node/edge breaks Vtable entry destructor

Figure 65. Example Jira bug issues [52].

7. Conclusions

The program created for this project satisfies objectives initially set during the challenge week (listed in "Project Introduction" document [30]). It gives comprehensive insight into algorithms internal state from different perspectives, attempting to be as intuitive as possible. In these terms, it exceeds capabilities of most publicly available tools. It supports the "ideal scenario" requirement of the project proposal [53] it originates from by allowing to create custom graphs using graphical user interface. In fact, the program extends that further by introducing a visual graph manager, where the user can select and load previously saved graphs. Drawing graphs (and representations of algorithm states) manually is a time-consuming process [18]. It can be accelerated and simplified by using this tool, especially that it has the visual graph manager aside of manual graph editing facilities.

The table below details the implemented functionality by reflecting on the initial requirements. Orange fields indicate features introduced despite not being planned from the beginning.

Ability to create and edit a graph	
Add, remove, move vertices	✔
Remove vertices and add new ones between existing vertices	✔
Set weight of edges (mouse selection, keyboard input)	✔
Move the whole graph	✔
Implementation of all graph algorithms taught in CE204	
Prim's (MST)	✔
Kruskal's (MST)	✔
Dijkstra (single source shortest path)	✔
Breadth-first search	✔
Depth-first search	✔
Ability to observe changes of the key data structures of algorithms	
Displayed "V table"	✔
Graph colouring	✔
Descriptions/labels of graph colours	✔
Dynamic edge labels (for Dijkstra, BFS, DFS)	✔
Pseudocode with the last step instructions being highlighted	✔
Textual description	✔
Visual representation of collection data types (e.g. priority queue, stack)	✔
Adjustable user interface	
Components scaling	✔
Ability to move components into multiple screens independently	✔
Preservation of components size/position when program restarts	✔
Additional features	
Implementation of algorithms not covered in CE204 module	✘
In-built documentation explaining terminology and concepts related to graph theory	✘
Different implementations of the same algorithm	✘
Visual graph manager	✔
Ability to select previously executed step label and read its' description	✔

Table 11. Comparison of what functionality was implemented against what functionality was initially planned. Yellow fields indicate features that were implemented despite not being planned in the first place.

I believe this program can realistically improve the quality of teaching graph algorithms it supports. It could be used by the instructor/teacher himself. It could also be used as a standalone learning tool for students who would like to practice on their own by simulating algorithms using custom graphs. This could help to consolidate their understanding.

According to the authors of "EVEGA" software [12], there is a trend in Computer Science education to make learning methods more interactive. Development of programs such as this one contributes to that trend. This allows students to be more independent with their learning and get a greater sense of accomplishment as a result. Furthermore, modernisation of teaching methods (by introducing interactive visualisation software) may lead to increased interest of students as suggested by K. Mocinecova and W. Stengartner (authors of the novel, unnamed tool published in 2020) [7].

The developed program implements all graph algorithms taught in the "CE204 Data Structures and Algorithms" module which makes it especially suitable for use in that specific module.

7.1   Possible improvements

- ability to substitute variables for values inside pseudocode window (or lookup by placing cursor over a variable)
- variable granularity of algorithm steps (e.g. by introduction of "sub-steps" or "small steps")
- importing and exporting graphs to files
- ability to use directional graphs
- better user interface (e.g. storing a graph by clicking on an icon or menu option, instead of right clicking on the graph itself which is not intuitive)
- higher range of algorithms (e.g. Floyd-Warshall, Travelling Salesman problem solutions)
- in-built descriptions of graph theory concepts
- ability to play/pause an algorithm with controllable speed
- ability to step back the algorithm
- ability to undo/redo graph edition events

8. References

[1]   moodle.essex.ac.uk, "CE301 Project Choices Database (Academic staff proposals)," 2020. [Online].
        Available:
        https://moodle.essex.ac.uk/mod/data/view.php?d=490&mode=list&perpage=30&search=&sort=0&
        order=ASC&advanced=0&filter=1&advanced=1&f_3666=graph&f_3667=&u_ln=&f_3671=&f_3672=.
        [Accessed 05 10 2020].

[2]   A. Bari, "3.6 Dijkstra Algorithm - Single Source Shortest Path - Greedy Method," 9 2 2018. [Online].
        Available: https://www.youtube.com/watch?v=XB4MIexjvY0. [Accessed 04 02 2021].

[3]   P. Strakhov, W. Wysota and L. Haas, "Game Programming Using Qt 5," 04 2018. [Online].

[4]   thenewboston, "C++ GUI with Qt playlist," 2014. [Online]. Available:
        https://www.youtube.com/playlist?list=PLD0D54219E5F2544D. [Accessed 04 02 2020].

[5]   ProgrammingKnowledge, "Qt C++ GUI Tutorial For Beginners," 2020. [Online]. Available:
        https://www.youtube.com/playlist?list=PLS1QulWo1RIZiBcTr5urECberTITj7gjA. [Accessed 04 02
        2021].

[6]   cse.tkk.fi, "TRAKLA2," 2009. [Online]. Available:
        http://www.cse.tkk.fi/en/research/TRAKLA2/download2.shtml. [Accessed 13 04 2021].

[7]   K. Mocinecová and W. Steingartner, "Software Support for Visualizing of the Graph Algorithms in a
        Novel Approach in Educating of Young IT Experts," 4 2020. [Online]. Available:
        http://ipsitransactions.org/journals/papers/tir/2020jul/p3.pdf. [Accessed 13 04 2021].

[8]   graphtheorysoftware.com, "GraphTea website," 2015. [Online]. Available:
        http://www.graphtheorysoftware.com/download. [Accessed 05 04 2021].

[9]   A. Fest and U. Kortenkamp, "Teaching graph algorithms with Visage," 2009. [Online]. Available:
        http://tmcs.math.unideb.hu/load_doc.php?p=151&t=doc. [Accessed 13 04 2021].

[10]   B. Thanasis, "Static and Dynamic Visualization ofNetwork Algorithms (JAVENGA)," 2009. [Online].
        Available: https://www.iti.gr/iti/files/document/seminars/JAVENGA.pdf. [Accessed 11 04 2021].

[11]   K. S. V. Hornweder, "Sketchmate: A Computer-Aided Sketching and Simulation Tool for Teaching
        Graph Algorithms Teaching Graph Algorithms," 08 2012. [Online]. Available:
        https://trace.tennessee.edu/cgi/viewcontent.cgi?article=2588&context=utk_graddiss. [Accessed 11
        04 2021].

[12]   S. Khuri and K. Holzapfel, "EVEGA: An Educational Visualization Environment for Graph Algorithms,"
        ACM SIGCSE Bull., pp. 101-104, 2001.

[13]   V. Dagdilelis and M. Satratzemi, "DIDAGRAPH: software for teaching graph theory algorithms," *ACM
        SIGCSE Bulletin,* pp. 64-68, 1998.

[14]   alternativeto.net, "Graphynx profile," 2020. [Online]. Available:
        https://alternativeto.net/software/graphynx/about/. [Accessed 06 04 2021].

[15]   graphonline.ru, "Graph Online web-based tool," 2021. [Online]. Available:
        https://graphonline.ru/en/. [Accessed 06 04 2021].

[16]   d3gt.com, "D3 Graph Theory web-based tool," 2018. [Online]. Available: https://d3gt.com.
        [Accessed 06 04 2021].

[17]   C. Shaffer and J. Yang, "Swan - A Data Structure Visualisation System," 2006. [Online]. Available:
        https://research.cs.vt.edu/AVresearch/Swan/. [Accessed 11 04 2021].

[18]   A. Andersen, "Graphshop github repository," 2011. [Online]. Available:
        https://github.com/stringoftheseus/graphshop/. [Accessed 05 04 2021].

[19]   softpedia.com, "Rin-G program official page," 21 07 2010. [Online]. Available:
        https://www.softpedia.com/get/Multimedia/Graphic/Graphic-Others/Rin-G.shtml. [Accessed 05 04
        2021].

[20]   KDE, "Rocs github repository," 2021. [Online]. Available: https://github.com/KDE/rocs. [Accessed 05
        04 2021].

[21]   moodle.essex.ac.uk, "Project Handbook - Chapter 12 - Final Report," 2020. [Online]. Available:
        https://moodle.essex.ac.uk/mod/book/view.php?id=622901&chapterid=11747. [Accessed 08 04
        2021].

[22]   GitLab, "Repository - ce301_borowski_michal," 2021. [Online]. Available:
        https://cseegit.essex.ac.uk/ce301_2020/ce301_borowski_michal. [Accessed 05 04 2021].

[23]   M. Sanderson, "CE204 Data Structures and Algorithms - Lecture slides (part 7)," 18 02 2020.
        [Online]. Available:
        https://github.com/michalmonday/files/blob/master/ce301%20Capstone%20project/part7%20-
        %20Graphs%20-%20MCSTs%20-%20Shortest%20Paths.pdf. [Accessed 11 02 2021].

[24]   A. Bari, "3.5 Prims and Kruskals Algorithms - Greedy Method," 09 02 2018. [Online]. Available:
        https://www.youtube.com/watch?v=4ZlRH0eK-qQ. [Accessed 11 02 2021].

[25]   H. George, P. Gary and S. Stanley, Algorithms in a nutshell - A Desktop Quick Reference, O'Reilly,
        2009.

[26]   stackoverflow.com, "Relaxation of an edge in Dijkstra's algorithm," 08 10 2012. [Online]. Available:
        https://stackoverflow.com/questions/12782431/relaxation-of-an-edge-in-dijkstras-algorithm.
        [Accessed 03 04 2021].

[27]   code.woboq.org, "qgraphicsitem.cpp file containing the qt_graphicsItem_shapeFromPath function,"
        2016. [Online]. Available:
        https://code.woboq.org/qt5/qtbase/src/widgets/graphicsview/qgraphicsitem.cpp.html. [Accessed
        09 04 2021].

[28]   stackoverflow.com, "Round to nearest multiple of a number," 15 4 2015. [Online]. Available:
        https://stackoverflow.com/questions/29557459/round-to-nearest-multiple-of-a-number. [Accessed
        09 04 2021].

[29]   doc.qt.io, "Elastic Nodes example," 2020. [Online]. Available: https://doc.qt.io/qt-5/qtwidgets-
        graphicsview-elasticnodes-example.html. [Accessed 09 04 2021].

[30] M. Borowski, "Project introduction document - Gitlab," 12 10 2020. [Online]. Available: https://cseegit.essex.ac.uk/ce301_2020/ce301_borowski_michal/-/blob/master/challenge%20week/Project%20introduction.docx. [Accessed 04 02 2021].

[31] Jira, "Identify and understand algorithms," 16 09 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-2. [Accessed 17 09 2020].

[32] Jira, "Familiarize with graph theory," 17 09 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-6. [Accessed 17 09 2020].

[33] Jira, "Recognize and evaluate methods of teaching algorithms," 16 09 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-3. [Accessed 17 09 2020].

[34] Jira, "Read Qt documentation," 16 09 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-4. [Accessed 06 10 2020].

[35] Jira, "Find relevant Qt examples," 16 09 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-5. [Accessed 06 10 2020].

[36] Jira, "Facility to create and modify a graph," 06 10 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-10. [Accessed 06 10 2020].

[37] Jira, "Implement a single algorithm," 06 10 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-11. [Accessed 06 10 2020].

[38] L. Fitina, I. John, U. Vanessa, M. Nathaline and G. Elizabeth, "An Application of Minimum Spanning Trees to Travel Planning," *Contemporary PNG Studies: DWU Journal,* vol. 12, 05 2010.

[39] Jira, "Formulate project objectives," 16 10 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-7. [Accessed 27 04 2021].

[40] Jira, "Document summer coding," 16 10 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-12. [Accessed 27 04 2021].

[41] Jira, "Facility to create and modify a graph," 16 10 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-10. [Accessed 27 04 2021].

[42] Jira, "Implement a single algorithm," 16 10 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-11. [Accessed 27 04 2021].

[43] GitLab, "testing_report.md," 2021. [Online]. Available: https://cseegit.essex.ac.uk/ce301_2020/ce301_borowski_michal/-/blob/master/final%20report/testing_report.md. [Accessed 28 04 2021].

[44] G. docs, "About READMEs," 2021. [Online]. Available: https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/about-readmes. [Accessed 08 04 2021].

[45] GitLab, "features_documentation.md," 2021. [Online]. Available: https://cseegit.essex.ac.uk/ce301_2020/ce301_borowski_michal/-/blob/master/final%20report/features_documentation.md. [Accessed 28 04 2021].

[46] GitLab, "implementation_report.md," 2021. [Online]. Available: https://cseegit.essex.ac.uk/ce301_2020/ce301_borowski_michal/-/blob/master/final%20report/implementation_report.md. [Accessed 28 04 2021].

[47] GitLab, "project_management.md," 2021. [Online]. Available: https://cseegit.essex.ac.uk/ce301_2020/ce301_borowski_michal/-/blob/master/final report/project_management.md. [Accessed 28 04 2021].

[48] GitLab, "program_controls.md," 2021. [Online]. Available: https://cseegit.essex.ac.uk/ce301_2020/ce301_borowski_michal/-/blob/master/final report/program_controls.md. [Accessed 28 04 2021].

[49] G. docs, "Licensing a repository," 2021. [Online]. Available: https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/licensing-a-repository. [Accessed 08 04 2021].

[50] stackoverflow.com, "Is it OK to use the MIT license for a project that uses Qt?," 29 12 2014. [Online]. Available: https://softwareengineering.stackexchange.com/questions/267582/is-it-ok-to-use-the-mit-license-for-a-project-that-uses-qt. [Accessed 10 04 2021].

[51] Jira, "Jira task issues," 2021. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-52?jql=project%20%3D%20A301079%20AND%20issuetype%20%3D%20Task%20AND%20status%20%3D%20Done%20AND%20created%20%3E%3D%202020-11-16%20AND%20created%20%3C%3D%202020-12-18. [Accessed 07 04 2021].

[52] Jira, "Jira bug issues," 2021. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-91?jql=project%20%3D%20A301079%20AND%20issuetype%20%3D%20Bug%20AND%20status%20%3D%20Done%20AND%20created%20%3E%3D%202020-11-16%20AND%20created%20%3C%3D%202020-12-18. [Accessed 07 04 2021].

[53] M. Sanderson, "CE301 Project Choices Database (Academic staff proposals)," 2020. [Online]. Available: https://moodle.essex.ac.uk/mod/data/view.php?d=490&mode=list&perpage=30&search=&sort=0&order=ASC&advanced=0&filter=1&advanced=1&f_3666=graph&f_3667=&u_ln=&f_3671=&f_3672=. [Accessed 17 04 2021].

[54] Jira, "Jira bug issue - Docks/step-log not passing key presses," 03 12 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-80. [Accessed 07 04 2021].