**A tool for teaching graph algorithms**

**CE301 Capstone Project introduction**

Contents

[Preface 3](#_Toc53704630)

[Project Objectives 3](#_Toc53704631)

[Proposed functional requirements: 3](#_Toc53704632)

[Proposed non-functional requirements: 4](#_Toc53704633)

[Implementation plan: 4](#_Toc53704634)

[Risks: 5](#_Toc53704635)

[Background Reading 6](#_Toc53704636)

[Familiarized myself with graph theory 6](#_Toc53704637)

[Understood different types of algorithms, and evaluated different methods of teaching them 7](#_Toc53704638)

[Undertook investigation to verify use case of MST as a near-optimal route 14](#_Toc53704639)

[Found out what Qt framework components and examples could be helpful for implementation 15](#_Toc53704640)

[Technical Achievements 16](#_Toc53704641)

[Project Management 16](#_Toc53704642)

[References 19](#_Toc53704643)

[Appendix A 22](#_Toc53704644)

# Preface

I used different colours to highlight GitLab and Jira references.

# Project Objectives

Clear formulation of the Project Objectives, Implementation plan, risks, context. Specific deliverables to be achieved by the oral interview in Week 11 must be agreed with your supervisor. Suitable evidence must be uploaded in Jira/Gitlab. The product may be hardware (e.g. electronic device), software ( e.g. game, app, website, database), or the use of software packages to deliver a product ( e.g. virtual network with security, device simulation). It is important that you describe the goals in a manner that gives confidence to their being achieved. All code for a project and the technical documentation must be held in the CSEE GitLab repository. We will provide some examples of how to develop the technical documentation within GitLab. This will be explained once the Autumn term has started.

The idea of this project is derived from “A tool for teaching graph algorithms” project proposal from project choices database on moodle [1]. The original description of the project is presented below:

*“The aim of this project is to implement a tool to assist with the teaching of graph algorithms such as those taught in CE204.*

*It should provide a graphical display of step-by-step implementation of the algorithms with accompanying descriptive text. This should be offered at various levels of abstraction (e.g. for Kruskal's algorithm a version showing the connection sets and a more abstract version that uses the concept of not forming a cycle). It should be able to work on arbitrary graphs (i.e. the descriptions should not be based on a particular graph).*

*Ideally there should also be a graphical user interface that allows the user to create graphs by adding nodes at points on the display and creating edges to connect them, allowing the user's chosen graph to be used to demonstrate the algorithm.”*

A program like this may help to consolidate understanding when using other teaching methods. Being able to view graphical representation of algorithm states may improve learning by allowing to tinker with and observe custom graphs.

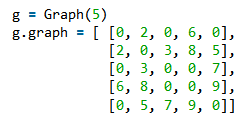
### Proposed functional requirements:

* Ability to create and edit a graph:
  + create, delete and move vertices
  + delete existing edges and create new edges between existing vertices
  + set weight of edges (select using mouse, input using keyboard)
  + move the whole graph
* Implementation of all graph algorithms taught in CE204, which includes:
  + Prim’s (minimum spanning tree)
  + Kruskal’s (minimum spanning tree)
  + Dijkstra (single source shortest path)
  + Breadth-first search
  + Depth-first search
* Ability to observe changes of the key data structures of algorithms. For example, for Prim’s algorithms it could be:
  + “V table” being displayed
  + Changed colour of edges included in MST
  + Changed colour of edges included in MST during last iteration
  + Changed colour of edges that are considered to be added next
  + Information about which colours mean what
  + Pseudocode with the last instruction being highlighted
* Adjustable user interface:
  + components scaling
  + ability to move components into multiple screens independently
  + save components size and position when program exists and load previously saved size and position when program is launched

Optimally the program would allow to view different implementations of the same algorithm and display time-complexity of each (but that is not a priority, the priority is to have a least 1 implementation of each CE204 algorithm).

One of the objectives when teaching algorithms is to minimize ambiguity and decrease risk of making false assumptions when looking at learning materials. For example let’s take a look at the “Prism and Kruskals Algorithms - Greedy Method” explanation by Abdul Bari in his [video](https://www.youtube.com/watch?v=4ZlRH0eK-qQ) [2]. He’s using numbers as labels for vertices. There’s a risk associated with teaching it this way because it increases probability that a new learner will confuse labels of vertices with weights of edges. If alphabetical names of vertices (e.g. “a”, “b”, “c”, etc.) were used instead, then it would be clear at first sight what is meant by numbers/letters.

It makes the difference especially when a graph is expressed using adjacency matrix (like in the [example](https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/) contributed to “geeksforgeeks” website by Divyanshu Mehta [3]).

If someone is not already familiar with what adjacency matrix is, then it may take a while to understand what is being supplied to “g.graph”. It would be more intuitive if vertices were labelled using letters instead of numbers.

On the other hand, using numbers allows to simplify implementation of algorithm, because vertices numbers can be used as index.

### Proposed non-functional requirements:

* The program should run on a standard PC.
* Implementation should be done using Qt/C++.
* The program should run smoothly and not crash.

### Implementation plan:

* **Complete and document background reading** [4][5][6]. 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.
* **Read Qt documentation** [7]. 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).
* **Find relevant Qt/C++ examples** [8]. 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.
* **Implement facility to create and modify a graph** [9]. This should be done before attempting to implement an algorithm.
* **Implement a single algorithm** [10]. 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).
* **Implement all graph algorithms taught in CE204 module.** Implementing multiple algorithms will require a facility to view and change the current algorithm.
* **Implement 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

### Risks:

* **Lack of motivation** [11]. It may be the case that the whole term or even whole year will be taught online. In the past, going to physical lectures automatically directed my focus on studying. Currently, sitting constantly in the same room, and knowing that it doesn't matter at which point I'll watch the online lecture recording, doesn't help with being focused. This situation makes it easier to get distracted by social media or computer games. This could be addressed by for example making a daily routine of writing task list to accomplish during the day.
* **Interference with coursework of other modules**. This project is easily extendable, if I gave it priority over other modules, I could find myself in a situation where I have insufficient time to complete other modules coursework. I created “Adjust workload (depending on other modules)” task issue on Jira [12] with the following description:

*“My main objective is keep on top of deliverables of all modules. At this point I know the following:*

* *challenge week deliverable is my current priority*
* *in 2nd term I'll have 3 other modules instead of 2 (so I have to do relatively more work on this project during 1st term)*
* *I'll have to submit important and extensive CE303 assignment (worth 60% of module) in week 9 (I'll focus on it straight after the challenge week)*

*I'm still waiting for CE314 (NLE) module moodle page to be published. Probably it would be a good idea to finish CE314 as soon as possible. That's because this project has no limit in terms of progress. If I finish all other coursework first, then I can direct all focus on this project without stress.”*

# Background Reading

You need to demonstrate that you have read and understood a range of background literature from a range of sources including to a great extent peer reviewed articles. A 3-5 page summary of the background reading with appropriate referencing (IEEE[1]) should be uploaded to Jira as evidence and must be summarised in your presentation.

Below I present what objectives I accomplished with background reading.

Familiarized myself with graph theory [5]

I found out that a graph is a collection of vertices (sometimes referred to as “nodes”) and edges (also referred to as “relationships” or “connections”). A graph can be represented in different ways as presented in Table 1. [13] [14]

| **Graphical** | **Adjacency list** | **Adjacency matrix** |
| --- | --- | --- |
|  | A = [E]  B = [E]  C = [E]  D = [E]  E = [A, B, C, D] | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | A | B | C | D | E | | A |  | - | - | - | 1 | | B | - |  | - | - | 1 | | C | - | - |  | - | 1 | | D | - | - | - |  | 1 | | E | 1 | 1 | 1 | 1 |  | |

Table . Graph representation methods (using undirected non-weighted graph as an example)

|  |  |  |
| --- | --- | --- |
|  | **Unidirectional graph** | **Directional graph** |
| **Not weighted** |  |  |
| **Weighted** |  |  |

Table . Different types of graphs.

A graph can be unidirectional (where edges do not point into any direction) or directional (where edges point towards single or both vertices).Adjacency matrix is a data structure well suited to represent directed graphs.

A graph can also be weighted, where each edge holds a value, typically representing some kind of distance or cost.

Table 2 shows examples of different graphs, Table 3 shows what kind of real-world objects can be represented by different graphs.

|  | **Unidirectional graph** | **Directional graph** |
| --- | --- | --- |
| **Not weighted** | Users friendships on facebook | Users friend requests on facebook |
| **Weighted** | Roadmap between cities | Single city roadmap  (with one-way roads) |

Table . What kind of real world objects can be represented by different graphs.

Wanting to see how a graph could be implemented I took a look at several implementations, including:

* Graph.java file from Guava (which is set of Google’s core Java libraries) [15]
* “Core graph operations” UML diagram from “Algorithms in a Nutshell” book (p. 141) [14]
* Qt “Elastic Nodes” example [16]

Understood different types of algorithms, and evaluated different methods of teaching them [4] [6]

I researched graph algorithms using books, articles and videos, below I present my findings which I’m planning to use as a lookup when designing and implementing the program.

**Minimum cost spanning tree (MST) finding (e.g. Prim’s, Kruskal’s).**

These algorithms operate on undirected weighted graphs. Their purpose is to find a way to “connect” all vertices by using edges with the lowest values. They’re sometimes called “greedy”, “greedy” strategy is about making an optimal choice at each small step. In real world, MST could be used to minimize total length of hydraulic pipes, or wires required to connect all households within an area. Some literature also suggests that minimum cost spanning tree could be used when we want to visit all locations at least once, as shown in Figure 1 and Figure 2.

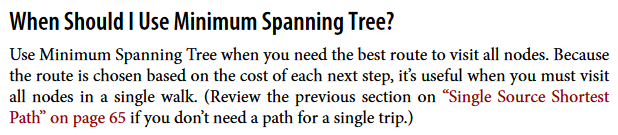


Figure . Use case from page 71 of “Graph Algorithms Practical Examples in Apache Spark & Neo4j” by Mark Needham and Amy Hodler [17]

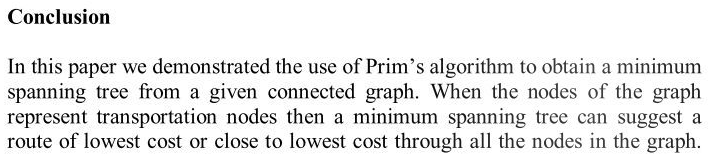


Figure . Conclusion from page 10 of “An Application of Minimum Spanning Trees to Travel Planning” by L. Fitina et al. [18]

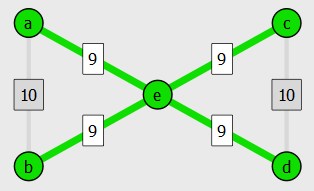
MST often can suggest the lowest cost route or close to the lowest cost, but it also can suggest route relatively far from the lowest cost in unfortunate circumstances as shown in Figure 4 and Figure 3.

Figure . Route not based on MST (47 total distance). Starting with “e”.

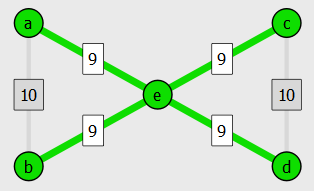


Figure . Route based on MST (63 total distance). Starting with “e”.

The problem with using MST as an optimal route to visit all nodes is that it forces returning to previously visited nodes. In 2D space, visiting previous nodes instead of using non-MST direct edges will always result in higher distance due to triangle inequality, so it could be improved by for example deleting the second occurrence of any non-leaf node as suggested by James Aspnes [19].

MST guarantees to be within a factor of 2 of the optimal path. For example, if MST has total weight of 20, then optimal path between all vertices can’t be lower than 10.

A better approach for “visiting all places by walking” problem (typically known as Travelling Salesman Problem) would be to use Christofides approximation algorithm which guarantees to result in path within a factor of 1.5 of the optimal path (and ensures that each node is visited only once) [20].

On page 14 I further analysed how effective MST is when used as optimal route.

**MST - Prim’s algorithm**

Prim’s algorithm builds MST using the following methodology (expressed in pseudocode):

|  |
| --- |
| select random vertex  **while** MST **is not** complete:  add selected vertex to MST  select new vertex //(one that isn’t part of MST and is currently the closest to any MST vertex) |

|  |
| --- |
|  |

Figure . Prim's algorithm example.

Table 4 shows comparison of Prims algorithm implementations. CE204 implementation uses the same data structure (“V” table) for 2 different purposes:

* it keeps track of current optimal weights/vertices
* it indicates whether vertex is included in MST (by having its weight set to 0)

This approach is efficient in terms of space but may be confusing if both purposes are not explained. An alternative implementation is using priority queue which stores vertices not yet included in MST, it is less confusing because each data object has a single purpose:

* “key” array - store current optimal weights
* “pred” array - store vertex currently having optimal weight
* priority queue - store vertices not yet included in MST (and allow easy retrieval of the smallest value)

| **Comparison of Prim’s implementations** | | | |
| --- | --- | --- | --- |
| **Source** | CE204 Data Structures and Algorithms lecture slides [13] | “Algorithms in a Nutshell” book [14] | “MST using Heap” journal article [21] |
| **Data types used** | Does not use priority queue.  Uses “V” table data structure (holding both: weights, vertices). | Uses priority queue implemented with binary heap.  Uses 2 separate arrays called “key” and “pred” (in place of “V” table). | Uses priority queue implemented with Fibonacci heap.  Uses 2 separate arrays called “C” and “E” (corresponding to “key” and “pred”). |
| **Time complexity** | O(V^2) | O((V + E) \* log n) | O(E + V log V) |
| **When to use** | Dense graphs | Sparse graphs | Sparse graphs |

Table . Prims algorithm - comparison of implementations. (V = number of vertices, E = number of edges)

The “When to use” information in Table 4 originates from lecture slides of “cs226” course of Princeton University [22] which presents the following comparison:

|  |  |
| --- | --- |
| **Graph type** | **Binary heap implementation performance** |
| Dense (2k vertices, 1m edges) | 2-3 times slower |
| Sparse (100k vertices, 1m edges) | 500 times faster |
| Sparse (1m vertices, 2m edges) | 10000 times faster |

Table . Prims algorithm - binary heap implementation, comparison of performance.

**MST - Kruskal’s algorithm**

Kruskal’s algorithm takes different approach than Prim’s, it builds multiple sub-graphs and joins them to form MST unlike Prim’s which builds MST using single, gradually expanded sub-graph. It appends edges into subgraphs starting from the edge with lowest weight, for that reason it requires to sort all edges by their weight at the beginning. It also requires facility to check whether cycle would be formed if particular edge was added (to avoid creation of cycle), which typically is accomplished by using “find” function. If such cycle is not found, then the connection can be made, merging 2 sets together using “union” function.

|  |
| --- |
|  |

Figure . Kruskal's algorithm example.

Kruskal’s algorithm is dependent on the number of edges, its time complexity is O(E log E) or O(E log V) depending on the implementation of disjoint-set data structure [23]. In general, it is more suitable for sparse graphs in comparison with Prim’s algorithm which can run in O(V^2), so independent of the number of edges.

**Single source shortest path (SSP) finding (e.g. Djikstra, Bellmand-Ford)**

Single source shortest path algorithms are useful at finding optimal routes to each vertex in a graph starting from the same vertex (single source). In real world it can be useful for someone who would like to visit several locations around his hometown, but had to return to hometown after visiting each destination.

**SSP - Dijkstra algorithm**

Dijkstra algorithm is similar to Prims algorithm, but Prims algorithm updates adjacent nodes of the newly included vertex, whereas Dijkstra algorithm updates adjacent nodes (performs “relaxation”) surrounding vertex with the lowest distance (that wasn’t selected yet).

The operation of Dijkstra algorithm could be described in the following python-based pseudocode:

|  |
| --- |
| // source vertex is supplied (path to source vertex is 0, so it is selected as “v” first)  set\_paths\_to\_all\_vertices\_to\_max\_int() // except source vertex  **for** \_ **in** vertices: // “\_” is ignored  v = vertex\_with\_smallest\_path\_that\_was\_not\_selected\_yet()  **for** adjacent\_v **in** v:  relax(adjacent\_v) // relaxation = updating of minimum path distance  mark\_v\_as\_selected() // so it won’t be selected as “v” anymore |

Example from Figure 7 shows the key behaviour of Dijkstra algorithm, where A is the source. Green colour indicates which vertices were already selected. When a vertex gets selected, its surrounding neighbours (that were not yet selected) get updated with potentially shorter paths (in other words, relaxation is performed). After each set of relaxations, a blue vertex with the lowest path gets selected (and turns green).

|  |
| --- |
| End result |

Figure . Dijkstra algorithm example.

The time complexity of Dijkstra algorithm is O((V+E) log V) or O(V^2 + E) depending on implementation. These implementations are also similar to Prims algorithm implementations.

| **Comparison of Dijkstra implementations** | | |
| --- | --- | --- |
| **Source** | “Algorithms in a Nutshell” book [14] | |
| **Data types used** | Uses “visited” bool array. | Uses priority queue instead of bool array. |
| **Time complexity** | O(V^2 + E) | O((V+E) log V) |
| **When to use** | Dense graphs | Sparse graphs |

Table . Dijkstra algorithm - comparison of implementations. (V = number of vertices, E = number of edges)

The drawback of using Dijkstra algorithm is that the graph cannot contain negative edges. If the graph contains negative edges then Bellman-Ford algorithm can be used instead.

**SSP - Bellman-Ford algorithm**

This algorithm is an alternative to Dijkstra algorithm which can be used when graph contains negative edges. It is worth to notice that the cannot have a cycle with cumulative weight that is negative.

While Dijkstra algorithm is focused mainly on vertices in the first place, the Bellman-Ford is mainly focused on edges. It needs a list of edges to work on, its method of operation is relatively simple as shown in pseudocode below:

|  |
| --- |
| // source vertex is supplied (path to source vertex is 0, so it is selected as “v” first)  set\_paths\_to\_all\_vertices\_to\_max\_int() // except source vertex  **for** \_ **in** range(number\_of\_vertices - 1):  **for** edge **in** all\_edges:  relax(edge.destination) // relaxation = updating of minimum path distance |

Bellman-Ford algorithm has time complexity of O(V\*E), using it on a complete graph will make it slow, its time complexity in such case would be equivalent to O(V^3).

**All pairs shortest path finding (e.g. Floyd-Warshall)**

This type of algorithm is equivalent to running single-source shortest path for every vertex within a graph, however its performance is much better.

It can be useful for finding optimal path to visit all vertices at least once in a graph that is either complete or incomplete. Which is a problem very similar to classical Travelling Salesman Problem (TSP), however in TSP vertices must be visited only once (not “at least once”). Thanks to that characteristic, Floyd-Warshall algorithm, together with “bruteforce” method (checking every possibility), can be used to determine optimal path even if the graph is a tree. On page 14, I presented results of using this methodology to verify how good MST is as a route to visit every location by walking.

**Searching algorithms (depth-first, breadth-first)**

Searching algorithms (also called traversal algorithms) allow finding a specific vertex (in such case the algorithm could be terminated once that vertex is found), they also allow assigning order value to nodes (which could be helpful for the Travelling Salesman Problem approximation algorithm based on minimum spanning tree [19]).

Both algorithms, BFS and DFS require marking previously visited vertices, in both cases a simple Boolean array would be sufficient, however it may be useful for teaching purposes to store more states than just “visited”. In “Algorithms in a Nutshell” pseudocodes, this is addressed by using “color” array which stores colour for each vertex using the following:

* white (not visited)
* grey (visited)
* black (BFS - all adjacent vertices visited, DFS - all deeper vertices visited)

**Breadth First Search (BFS)**

|  |
| --- |
| queue.append(starting\_vertex)  **while** **not** queue.isEmpty():  v = queue.pop\_head()  **for** adjacent\_v **in** v:  **if** adjacent\_v **not** **in** queue:  queue.append(adjacent\_v) |

Figure . Breadth First Search pseudocode.

BFS algorithm visits all adjacent vertices of a single vertex first (before moving deeper into the graph). Figure 9 presents that behaviour in 2nd step where A-D visit is made. BFS can be implemented using queue data structure in straightforward way shown in pseudocode from Figure 8.

|  |
| --- |
|  |

Figure . Breadth First Search example (A is the starting vertex). 2nd step would be different in DFS.

**Depth First Search (DFS)**

DFS algorithm gives priority to deeper edges before exploring all adjacent ones of the starting vertex.

|  |
| --- |
|  |

Figure . Depth First Search example (A is the starting vertex). 2nd step would be different in BFS.

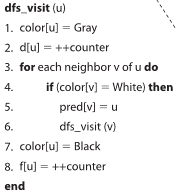
DFS can be solved using recursion, which is not intuitive, and often hard to visualize. For that reason, using “color” array can be helpful when teaching this algorithm. Figure 11 shows example of how it could be implemented. The purpose of gray colour is to show that the recursive function was called, but not yet returned, because black colour is used when that happens. Figure 12 shows example visualisation using such code.

Figure . DFS visit function example from "Algorithms in a Nutshell" pseudocode (p. 150).

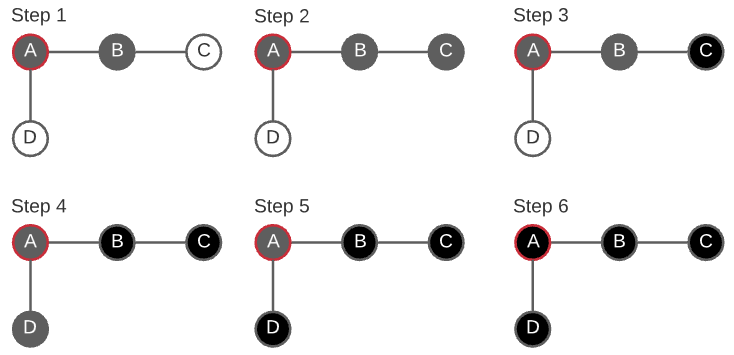


Figure . Example of how colours can be used to give insight into behaviour of recursive "dfs\_visit" function. A is the starting node. 3rd step behaviour (where the deepest node “C” turns black) indicates that “dfs\_visit(C)” returned.

## Undertook investigation to verify use case of MST as a near-optimal route

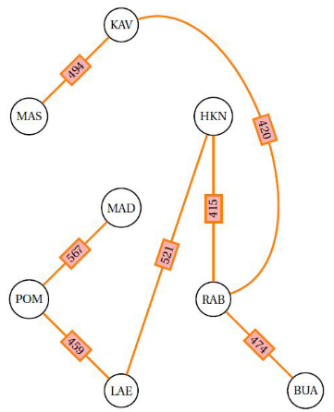
I was intrigued by the claims that MST can be used to find near-optimal route from the article called “An Application of Minimum Spanning Trees to Travel Planning” by L Fitina et a. [18]. The article presented the graph shown in Figure 14.

Figure . MST from Figure 8-g of "An Application of Minimum Spanning Trees to Travel Planning" (page 10)

Figure . Graph from Figure 7 of "An Application of Minimum Spanning Trees to Travel Planning" (page 8)

The article states:

*“Fig 8-g represents the minimum spanning tree of the connected graph in Figure 7. This is the subtree with minimum weight. What this means is that if one wants to visit all the nodes in Figure 7, this is the route that one should take in order to minimize cost. In this case, the total cost is* ***K3350****.”*

It isn’t possible to visit all nodes within 3350 distance by following MST due to the fact that at least 1 node would have to be revisited. This is acknowledged by the authors in the next paragraph (but for some reason the additional distance wasn’t added to 3350):

*“This is not as simple as it looks. In order to take full advantage of this tree, one has to visit at least one of the nodes more than once. For example a possible route is MAD, POM, LAE, HKN, RAB, BUA, RAB, KAV, MAS. Here, a person starts at MAD, then POM, etc until he/she reaches RAB. From RAB he/she takes the trip to BUA, then back again to RAB and thence to HKN, KAV and MAS. Thus RAB is visited twice.* *However, even with the above scenario, one still spends less than any other route in the graph, in which every node is visited at least once* ***(as one can easily verify)****.”*

I wanted to verify if the last sentence is true and realised that it’s not simple. I wrote a program which finds n optimal routes through graph starting from specific node (using all possible combinations) and compares the optimal distance with the distance achieved by following MST (which also differs depending on starting node). Program code is available in Appendix A. I found out that the starting node selected by authors of that article (node called “MAD”) was the only starting node where using MST resulted in optimal path.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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 . Investigation results. (“MST overhead” is additional distance covered if MST was used instead of optimal path).

Found out what Qt framework components and examples could be helpful for implementation [24] [25]

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. I chose to use Qt framework and C++ language to implement the program.

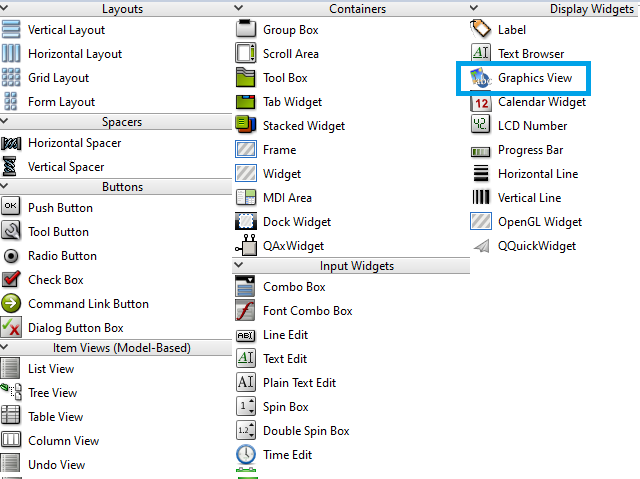
Qt offers great range of GUI widgets (e.g. Figure 15). The one of most interest for this project is “Graphics View”. As described in the “Game Programming Using Qt 5” book [26], it is a facility to create and display graphical items (from simple rectangles to more complex, structured, dynamic items).

Figure . Range of available Qt widgets. (QGraphicsView is marked with blue colour).

**Implementing graphs**

Optimally, I would like to construct structured graphical items for graphs, which would consist of smaller “child” items such as vertices and edges. The program should allow to move both:

* the graph itself
* each vertex within a graph

Trying to understand how this can be accomplished I’ve read QGraphicsItem class documentation [27] which presents example of how to create custom subclass, in my case that subclass will be called “Graph”. I could also subclass it to create “Node” and “Edge” classes. Fortunately, QGraphicsItem offers an in-built way of creating structured items, if we supply one item in the constructor of another item, the supplied item will become a parent and the newly created item will become a child. When parent gets deleted, all the child items get automatically deleted too, it is convenient because otherwise I would have to implement some kind of collection member variable to keep track and manage child items, doing it “Qt way” will result in cleaner code.

Regarding the movement of the parent item and child items, it isn’t as straightforward as it could be. It is possible to use the in-built Qt-way of moving graphical objects by setting their “ItemIsMovable” flag, however it misbehaves when both, parent, and child items are movable [28]. For that reason, it may be a better idea to avoid using the in-built way, and override the following QGraphicsItem methods instead:

* mousePressEvent - set bool member variable to “true”
* mouseMoveEvent - move the item if bool member variable is “true”
* mouseReleaseEvent - set bool member variable to “false”

**Relevant examples**

Qt is a powerful framework with many ways of accomplishing the same task. Reading official examples is likely to help with design and improve code readability. I found the following examples useful:

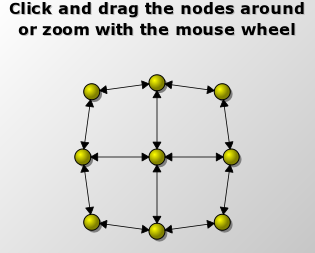
**Elastic nodes example (Figure 16)**

Figure . Elastic nodes example. [12]

Mainly it shows:

* how a graph could be created (including “Node” and “Edge” classes)
* math behind calculating arrow heads shape and position
* how “timerEvent” method can be overridden to create animation

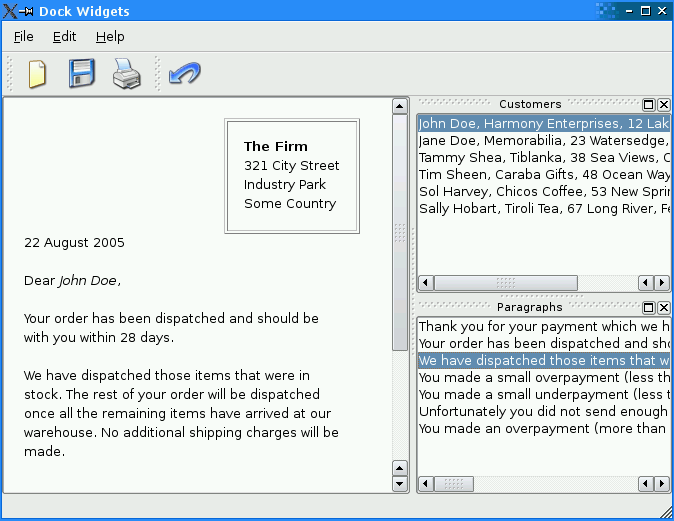
**Dock widgets example (Figure 17)**

Figure . Dock widgets example. [27]

It shows how to accommodate multiple windows (docks) within single application that can float as separate windows, but also can be “docked” in different parts of the main window. This example may be very helpful for making adjustable user interface for this program. This program will display a lot of different information about graphs, and states of algorithms. Therefore, space efficiency is a big concern. Using QDockWidget class will allow to adjust position and visibility of different GUI components without the need to hard-code them.

# Technical Achievements

Technical achievements during challenge week- highlighted in your presentation with suitable evidence uploaded in Jira/Gitlab. Evidence could be software, videos, snapshots, a short written summary of achievements based on your own individual project/work done.

The summary of technical achievements was uploaded on GitLab because it allows using animations (which is not possible directly in Word document). Link:

<https://cseegit.essex.ac.uk/ce301_2020/ce301_borowski_michal/-/blob/master/challenge%20week/features_documentation.md>

It includes description, images and animations of features, it also includes some key parts of code responsible for them.

# Project Management

A record of summer and challenge week activity on Jira, demonstrating a clear strategy for the use of Jira, including the use of issues, releases, and reports.

As shown in the “Project objectives” section, my plan regarding implementation of this project was clearly defined [7] [8] [9] [10]. Being aware of the importance of CE301 module I put a lot of time and effort into background reading which included reading relevant parts of several books, journal articles, lecture notes and viewing electronic resources. I attempted to maximise chances of this project being successful by:

* trying to implement some of features I knew will be required (e.g. graph creation)
* considering risks and trying to find ways to address these risks [11] [12]
* making frequent backups of the written software
* practical and realistic approach towards using Jira (not using features of Jira that could unnecessarily complicate workflow, for example in tasks issues related to challenge week, I’ve set the “Due date” to reasonable dates like 14th, 15th, 16th of October, for other issues, that are not necessarily challenge week related, I didn’t set any “Due date”)

Until 12th October zoom lecture I was not sure if it is completely alright to start with some coding during the summer, for that reason, before the start of the new academic year I used Kanban board only to set general tasks regarding background reading, rather than for setting features-related tasks. From the start of the challenge week:

* Gitlab commits reflect the software changes, example commits:
  + [Fixed positioning bug when moving graphical items](https://cseegit.essex.ac.uk/ce301_2020/ce301_borowski_michal/-/commit/386f56c682da218d1f2d228e42891e87f9de13b7)
  + [Added step log](https://cseegit.essex.ac.uk/ce301_2020/ce301_borowski_michal/-/commit/5b5e67cc9e69228fa6afe07a3ae4b6bd43d6323e)
* Jira issues contain links to relevant Gitlab commits, example issues:
  + [Moving vertices isn’t perfect](https://cseejira.essex.ac.uk/browse/A301079-13) (bug)
  + [Refactor creating “Step log” description](https://cseejira.essex.ac.uk/browse/A301079-25) (task)
  + [Write conditional and dynamic step descriptions for Prims algorithm](https://cseejira.essex.ac.uk/browse/A301079-23) (task)

The cumulative diagram below (generated using Jira reports) shows the distribution and management of Jira issues created so far.

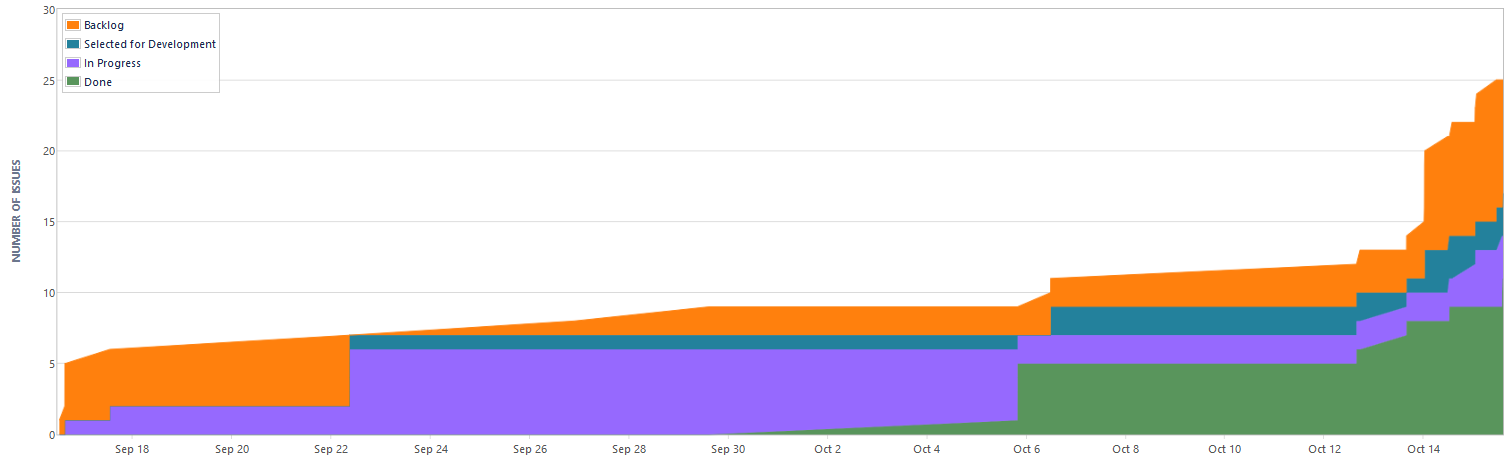


Figure . Cumulative flow diagram.

On the next page I included all the issues created on Jira so far.



Figure . All issues created on Jira so far (as of 15/10/2020)

Documentation hosted on Gitlab includes a lot of images and animations. I decided to use a separate repository to host these files. Having such independent repository will allow to:

* **decrease the size of original repository** (so it can be cloned much faster, there’s no point for each user to download 100MB of png or gif files used solely for documentation)
* **keep image references alive** even if the project is moved to GitHub in future (after the end of the academic year, this way I will be able to present this project to potential employers)
* **make less commits** (so the actual progress is easier to track by looking at commit names, having 15 commits where single image is added would disrupt that to some extent)

# 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, “Youtube video - "Prims and Kruskals Algorithms - Greedy Method",” 08 02 2018. [Online]. Available: https://www.youtube.com/watch?v=4ZlRH0eK-qQ. [Accessed 14 10 2020]. |
| [3] | D. Mehta, “geeksforgeeks - python implementation of Prim's algorithm,” 27 07 2020. [Online]. Available: https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/. [Accessed 14 10 2020]. |
| [4] | Jira, “Identify and understand algorithms,” 16 09 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-2. [Accessed 17 09 2020]. |
| [5] | Jira, “Familiarize with graph theory,” 17 09 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-6. [Accessed 17 09 2020]. |
| [6] | 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]. |
| [7] | Jira, “Read Qt documentation,” 16 09 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-4. [Accessed 06 10 2020]. |
| [8] | Jira, “Find relevant Qt examples,” 16 09 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-5. [Accessed 06 10 2020]. |
| [9] | 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]. |
| [10] | Jira, “Implement a single algorithm,” 06 10 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-11. [Accessed 06 10 2020]. |
| [11] | Jira, “Risk issue - Lack of motivation,” 1 10 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-9. [Accessed 06 10 2020]. |
| [12] | Jira, “Adjust workload (depending on other modules),” [Online]. Available: https://cseejira.essex.ac.uk/projects/A301079/issues/A301079-8. [Accessed 06 10 2020]. |
| [13] | M. Sanderson, “CE204 Data Structures and Algorithms (part 7, graphs, MCST, shortest path),” 18 02 2020. [Online]. Available: https://github.com/michalmonday/files/raw/master/ce301%20Capstone%20project/part7%20-%20Graphs%20-%20MCSTs%20-%20Shortest%20Paths.pdf. [Accessed 17 09 2020]. |
| [14] | H. George, P. Gary and S. Stanley, Algorithms in a nutshell - A Desktop Quick Reference, O'Reilly, 2009. |
| [15] | github.com, “Google - Guava (Graph implementation),” [Online]. Available: https://github.com/google/guava/blob/master/guava/src/com/google/common/graph/Graph.java. [Accessed 06 10 2020]. |
| [16] | doc.qt.io/, “Elastic nodes example,” 2020. [Online]. Available: https://doc.qt.io/qt-5/qtwidgets-graphicsview-elasticnodes-example.html. [Accessed 20 09 2020]. |
| [17] | A. H. Mark Needham, Graph Algorithms - Practical Examples in Apache Spark & Neo4j, O'Reilly, 2019. |
| [18] | 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. |
| [19] | J. Aspnes, “Approximation Algorithms,” 2012. [Online]. Available: https://www.cs.yale.edu/homes/aspnes/pinewiki/ApproximationAlgorithms.html. [Accessed 18 09 2020]. |
| [20] | S. Burille, “Traveling Salesman Problem - Christofides problem,” 16 05 2016. [Online]. Available: http://math.stmarys-ca.edu/wp-content/uploads/2017/07/Sophia-Burille.pdf. [Accessed 21 09 2020]. |
| [21] | C. Maumita, S. Rahul and M. Ruchi, “Minimum Spanning Tree using Heap,” *International Journal of Scientific & Engineering Research,* vol. 8, no. 5, pp. 1755-1761, 2017. |
| [22] | cs.priceton.edu, “cs226 - Minimum Spanning Tree (lecture slides),” [Online]. Available: https://www.cs.princeton.edu/courses/archive/spr02/cs226/lectures/mst-4up.pdf. [Accessed 23 09 2020]. |
| [23] | C. Thomas, L. Charles, R. Ronald and S. Clifford, “Introduction to Algorithms, Second Edition,” The MIT Press, 2001, pp. 483-484. |
| [24] | Jira, “Read Qt documentation,” 16 09 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-4. [Accessed 17 09 2020]. |
| [25] | Jira, “Find relevant Qt examples,” 16 09 2020. [Online]. Available: https://cseejira.essex.ac.uk/browse/A301079-5. [Accessed 17 09 2020]. |
| [26] | S. Pavel, W. Witold and H. Lorenz, Game Programming Using Qt 5 - Beginner's guide (second edition), Packt, 2018. |
| [27] | doc.qt.io, “QGraphicsItem class documentation,” 2020. [Online]. Available: https://doc.qt.io/qt-5/qgraphicsitem.html. [Accessed 20 09 2020]. |
| [28] | qtcentre.org, “Forum thread - Moving child QGraphicsItem,” 27 02 2011. [Online]. Available: https://www.qtcentre.org/threads/39152-Moving-child-QGraphicsItem. [Accessed 20 09 2020]. |
| [29] | doc.qt.io, “Dock widgets example,” 2020. [Online]. Available: https://doc.qt.io/qt-5/qtwidgets-mainwindows-dockwidgets-example.html. [Accessed 20 09 2020]. |

# Appendix A

**MST as a near-optimal path verification python3 code**

The code below was used to produce results of investigation presented on page 14. It compared distance covered by following MST only, with distance covered by following optimal path through all vertices.

from itertools import permutations  
  
def floyd\_warshall(graph):  
 *''' This function was copied from:  
 https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/ '''* V = len(graph)  
 dist = [\*map(lambda i : [\*map(lambda j : j , i)], graph)]  
 for k in range(V):   
 for i in range(V):   
 for j in range(V):   
 dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])   
 return dist  
  
def get\_path\_length(path, optimal\_distances):  
 *''' optimal\_distances is the result of Floyd Warshall algorithm. '''* length = 0  
 current\_node = path[0]  
 for next\_node in path[1:]:  
 length += optimal\_distances[current\_node][next\_node]  
 current\_node = next\_node  
 return length  
  
def get\_n\_best\_paths\_starting\_from(n, starting\_node, nodes, optimal\_distances):  
 *''' It checks all possible permutations. '''* nodes\_to\_visit = [\*range(len(nodes))]  
 lengths = [999999]  
 paths = [ [] ]  
 for path in permutations(nodes\_to\_visit, len(nodes\_to\_visit)):  
 if nodes[path[0]] != starting\_node:  
 continue  
 length = get\_path\_length(path, optimal\_distances)   
 if length < lengths[-1]:  
 for i, L in enumerate(lengths):  
 if length < L:  
 lengths.insert(i, length)  
 paths.insert(i, path)  
 if len(paths) > n:  
 del paths[-1]  
 del lengths[-1]  
 break  
 return lengths, paths  
  
INF = 99999  
nodes = ['KAV', 'MAS', 'HKN', 'MAD', 'POM', 'RAB', 'LAE', 'BUA']  
  
# data from Figure 7 of: https://www.dwu.ac.pg/en/images/All\_Attachements/Research%20Journals/vol\_12/2010-V12-1\_Fitina\_et\_al\_spanning\_trees\_for\_travel\_planning.pdf  
graph = [  
 # KAV, MAS, HKN, MAD, POM, RAB, LAE, BUA  
 [ 0, 494, 600, 738, INF, 420, INF, INF], # KAV  
 [ 494, 0, INF, INF, 766, INF, INF, INF], # MAS  
 [ 600, INF, 0,1054, 603, 415, 521, INF], # HKN  
 [ 738, INF,1054, 0, 567, 901, 910, INF], # MAD  
 [ INF, 766, 603, 567, 0, 758, 459, 922], # POM  
 [ 420, INF, 415, 901, 758, 0, 677, 474], # RAB   
 [ INF, INF, 521, 910, 459, 677, 0, INF], # LAE  
 [ INF, INF, INF, INF, 922, 474, INF, 0] # BUA  
 ]  
  
  
mst = [  
 # KAV, MAS, HKN, MAD, POM, RAB, LAE, BUA  
 [ 0, 494, INF, INF, INF, 420, INF, INF], # KAV  
 [ 494, 0, INF, INF, INF, INF, INF, INF], # MAS  
 [ INF, INF, 0, INF, INF, 415, 521, INF], # HKN  
 [ INF, INF, INF, 0, 567, INF, INF, INF], # MAD  
 [ INF, INF, INF, 567, 0, INF, 459, INF], # POM  
 [ 420, INF, 415, INF, INF, 0, INF, 474], # RAB   
 [ INF, INF, 521, INF, 459, INF, 0, INF], # LAE  
 [ INF, INF, INF, INF, INF, 474, INF, 0] # BUA  
 ]  
  
optimal\_distances = floyd\_warshall(graph)  
mst\_distances = floyd\_warshall(mst)  
  
for starting\_node in nodes:  
 # get total distance of mst (including travel back)  
 mst\_lengths, mst\_paths = get\_n\_best\_paths\_starting\_from(1, starting\_node, nodes, mst\_distances)  
 mst\_length = mst\_lengths[0]  
 mst\_path = mst\_paths[0]  
  
 print(f'MST starting from {starting\_node} results in the following distances:')  
 print(mst\_length, ' - '.join(nodes[i] for i in mst\_path))  
 print()  
  
 lengths, paths = get\_n\_best\_paths\_starting\_from(10, starting\_node, nodes, optimal\_distances)  
  
 print(f'MST is {int(mst\_length/lengths[0]\*100)}% of optimal path.')  
 print()  
  
 print('Top 10 distances:')  
   
 for length, path in zip(lengths, paths):  
 print(length, ' - '.join(nodes[i] for i in path))   
 print()  
 print()  
 print()  
  
#for node, distances in zip(nodes, optimal\_distances):  
# print (node, distances)