NTNU

Norwegian University of
Science and Technology

Geospatial Routing Engine in MySQL

Authors
Erik Kaasbøll Haugen
Georg Vilhelm Seip

Bachelor of Engineering in Computer Science
20 ECTS
Department of Computer Science
Norwegian University of Science and Technology,

20.05.2022

Supervisor Tore Mallaug

Summary of Graduate Project

Title:	**Geospatial Routing Engine in MySQL**
Date:	20.05.2022
Authors:	Erik Kaasbøll Haugen Georg Vilhelm Seip
Supervisor:	Tore Mallaug
Employer:	Oracle
Contact Person:	Norvald Ryeng
Keywords:	BSE, IDI, SQL, MySQL, ST_SHORTEST_DIR_PATH, routing engine
Pages:	40
Attachments:	1
Availability:	Open

Abstract:

Development of a geospatial routing engine and integration into MySQL to produce a user friendly SQL based method of deriving the shortest path between two nodes in a graph. Previously, this was possible in MySQL through use of recursive user defined functions. A method which is both lacking in ergonomics and functionality, but also generally underperforming. The implementation devised in this paper supports two modes. Either node geometry is declared by the user, in which case an appropriate heuristic will be used, or no heuristic is used. During performance testing it was indicated that this paper's method is roughly three and a half time faster than the old method. However, that was only the case when not applying heuristics. Given the increased data complexity and demand of spatial heuristics, using heuristics made the process three times slower compared to not using heuristics. This surprisingly poor performance was ultimately blamed on the ephemeral nature of graphs in dynamic query-based graph construction.

Preface

One of the main reasons for picking this specific assignment was wanting to create something that could potentially be used on a daily basis by a large amounts of people. The thought of creating something which would never be used again was not appealing, and getting to contribute to a project as big as MySQL was quite motivating. A chance to also get to work closely with a big player in the world of technology was no less preferential.

Furthermore, this afforded a chance to work with the programming language C++, a coveted language with which familiarity already existed. Previously, a course had been taken, videlicet, 'IDATT2101 Algoritmer og datastrukturer', where knowledge on different algorithms for path searching had been acquired. This course can be considered a source of inspiration, so getting to work with this type of project was quite motivating.

We would like to thank Torje Digernes and Norvald Ryeng at Oracle for their help in guiding us through their massive codebase and setting us up with the necessary equipment to complete our task. We would also like to extend our thanks to our supervisor Tore Mallaug for his advice and guidance.

Technical

The purpose of this bachelor thesis is to extend the MYSQL server with geospatial routing functionality in SQL and make it available as an SQL function. This will make it possible for users to write a query to find the shortest route from point A to point B. For example a user can load information about how to navigate hallways and stairways at a university campus and use the routing engine to find out how to get from their current location to their next lecture at the auditorium.

The project must be written in C++ as this is the language used in the MySQL source code. The use of Boost Graph library is possible, although an own standalone implementation is preferred. As the project is to be used in a database setting, it is crucial the the function is robust and air tight. To achieve this, a large amount of tests must be written an passed.

All code will be contributed under the Oracle Contributor Agreement(OCA).

Contents

1 Definitions

1.1 From code

An **edge** is a route between two points in a graph.

```
struct Edge {
  int id;
  // node id
  int from, to;
  // weight
  double cost;
};
```

A **point/node** is the origin or destination of an edge in a graph. Throughout this thesis they will be referred to as nodes.

```
struct Node {
  // sum of edge.cost along path
  double cost = INFINITY;
  // cost_heu = real_cost + heuristic
  double cost_heu = INFINITY;
  // used to retrace path
  const Edge* path = nullptr;
};
```

A **graph** is a set of edges, here in the form of a hash map adjacency list.

```
std::unordered_multimap<int , const Edge> graph;
```

A **path** is a contiguous set of edges between two nodes.

```
std::vector<Edge> path;
```

1.2 Abbreviations

- **SQL** Structured Query Language
- **MTR** MySQL Test Run
  MySQL's in-house integration testing framework
- **STL** Standard Template Library
  Common algorithms and data types for standardized C++ programming
- **JSON** JavaScript Object Notation
  Generic data type
- **NTNU** Norwegian University of Science and Technology
- **ISO** International Organization for Standardization
- **IEC** International Electrotechnical Commission
- **CPU** Central Processing Unit
- **GIS** Geographic Information System

- **THD** Thread Descriptor
  MySQL connection object
- **GDB** GNU Debugger
- **GNU** GNU's Not Unix (recursive)
- **BFS** Breadth First Search
- **CLI** Command Line Interface
- **SRID** Spatial Reference Identifier
  Id to uniquely identify a coordinate system
- **YACC** Yet Another Compiler-Compiler

Se section 3 (Theory) for more in depth definitions.

2 Introduction

Relational database management systems are the workhorses of modern day data persistence. They allow easy maintenance and administration of relational data. Which is data structured to relate to and identify other data. Throughout the years they have become one of the most popular types of database systems among organizations across the whole world. They provide a dependable system for storing and retrieving huge amounts of data while offering a consolidation of ease of implementation and system performance. Between the many such database systems, according to a survey on stack overflow, the one used the most is MySQL [Kamaruzzaman, 2021].

Expansion of MySQL to further improve database usability and searchability is always endeavoured. To this end, it became noticeable how lacking support for such properties accost a certain data structure, videlicet, graphs. Improvements therein have therefore been made the subject of this thesis. More explicitly:

> Would a tailored implementation to derive a graph's shortest path with SQL help streamline graph database searching in MySQL?

Deriving the shortest path through data stored in a MySQL database was not impossible before. Two obvious solution already existed; the whole graph could be extracted and the path derived externally, or a recursive SQL function could be defined to calculate the path internally. However, the former solution demands disconcerting amounts of broadband, and CPU cycles if transferred data is compressed. Whilst the latter is complex and unintuitive, not to mention reliant on user defined SQL functions, which are less integrated and thus expected to perform worse than integrated SQL functions. For these reasons, it is believed that work done on the thesis of this paper could be relevant to improving MySQL.

3 Theory

Theoretical deliberation within this chapter serves as a general foundation for the less exoteric segments of this paper. Notably, section 5 (Result) and section 6 (Discussion) will assume prior knowledge, which should not exceed this chapter.

3.1 Dijkstra's algorithm

3.1.1 Core principles

Dijkstra's shortest path algorithm was deviced by Edsger W. Dijkstra in 1956 [Richards, 2019]. It finds the shortest path through any graph from a given point to any other. Something, realized through a surprisingly straightforward expansionist exploration of nodes beginning at the path's origin. How this works, and some less obvious caveats of implementing such an algorithm will be this part's focus.

First and foremost, how does Dijkstra's algorithm work? In light of the BFS algorithm it is based on, Dijkstra's algorithm is reassuringly similar. It diverges slightly from BFS in how it chooses its succeeding nodes and by lodging more complex bookkeeping. The first step is marking the origin node as *found*. After this the main process begins. It checks all edges leading out from all found nodes whose edges have yet to be investigated, beginning with the node whose path from the origin is currently shortest. If any of these edges lead to an unexplored node, the edge is remembered as the final edge in the hitherto shortest path to said note. However, if the node was already explored the algorithm can only do so if this edge is part of a shorter path than the previously found path. This process is then repeated until the destination node's edges themselves are being considered, at which point the shortest path will be known [Dijkstra, 1959]. The process may appear surprisingly short and simple, but important details of actually implementing it were purposefully left out.

When implementing Dijkstra's algorithm, the devil is in the detail. As indicated, Dijkstra's algorithm needs a way to remember and order its found nodes by which is closest to the origin. Suboptimally, a simple array could be used for this task. Which would give it a worst case time complexity of $O(n^2)$, as it struggles to reorder the found nodes whilst new ones are constantly added or modified. Fortunately, this can be improved. By using a heap the complexity is reduced to only $O((n + E) * log(n))$, drastically improving the performance [Helge Hafting, 2014]. Here n is the number of nodes and E is the number of edges. Heaps are properly explained in subsection 3.5.3 (Heaps).

3.1.2 A-star & heuristics

Dijkstra's algorithm is, as mentioned, reminiscent of BFS, but where the BFS algorithm expands its search layer by layer going out from the center, Dijkstra's algorithm maintains a more constant radial exploration. This is a side effect of the Dijkstra's algorithm constantly moving slightly further down the shortest possible path it can find from the origin. Such a circular search pattern means that by the time the destination has been discovered all shorter routes from the origin to any other point in the graph will have been evaluated. Most off

putting, this means that the algorithm will happily waddle of in the completely wrong direction. A preferable version of Dijkstra's algorithm should surely explore more vehemently towards the destination, but how?

For this to happen, a way of determining whether the algorithm's navigation *appears* reasonable is imperative. Underwhelmingly, the officially recognized *modus operandi* is to guess the distance between nodes and the destination, and then first explore those that look closer. This approach is formally known as the A-star algorithm, and the guessed distance is called a heuristic. More specifically, the A-star algorithm is identical to Dijkstra's algorithm apart from how it sorts its node min heap. One possible function to sort by is found in the equation below.

$$f(n) = g(n) + h(n) \tag{3.1}$$

Here g is the distance so far and h is the heuristic [Duchon et al., 2014]. Many varieties of function f(n) are in use, for example this other function found in the equation below.

$$f(n) = g(n) + h(n) \cdot c \tag{3.2}$$

Here c is a coefficient deciding how much the heuristic should matter. The last function worth mentioning can again be found in the equation below.

$$f(n) = h(n) \tag{3.3}$$

This last equation causes a super greedy form of A-star, which will travel in as straight a line as possible to the destination. The last function is computationally the most efficient, for most graphs, but gives no guarantee of finding the optimal path. Instead, the first function is most often utilized. It strikes a good balance between accuracy and speed, and so long as the heuristic guesses correctly or undershoots the actual distance it will give the shortest path. Mathematically, this means maintaining the relation below.

$$h(n) \leq h^*(n) \tag{3.4}$$

In this case the bigger h(n), is the faster the algorithm will run, but, as mentioned, if it ever surpasses the actual remaining distance, $h^*(n)$, it could return a suboptimal path [Hart et al., 1968].

One question remains, how does the heuristic find the remaining distance? More than one valid answer exists, but this section will be limited to spatial heuristics. Spatial heuristic are the most obvious solution, and often the simplest. With Cartesian coordinates, it is as simple as using the formula for euclidean distance, as found below.

$$h(n) = \sqrt{\sum_{i=0}^{dim(n)} (n_i - d_i)^2} \tag{3.5}$$

Where n is the current node, d is the destination node, and n_i and d_i are corresponding dimensional magnitudes for dimension i. Some implementations drop the square root for lower execution time, but within this thesis it is deemed too sacrificial in terms of precision. By keeping the above equation, all euclidean graphs with defined costs no less than the real spatial distance spanned by the corresponding edge will ensure a correct answer from the A-star algorithm when using Equation 3.1. This characteristic is deemed beneficial, and can be expanded to cover all other coordinate systems so long as an accurate heuristic distance can be calculated.

One serious downside to spatial heuristics is linked to their spatial complexity. Every node needs a representation of its geometry, and depending on the range and precision wanted, this can cause crippling demands on system memory. Which is especially critical for sparse graphs, meaning graphs with few edges relative to number of nodes. A circumstance where memory usage could easily double. Moreover, even dense graphs could have problems. Even though memory usage is unlikely to increase equally dramatically in relative terms, for massive data sets, any increased memory usage could surpass available system memory. If so, it would be advisable not to use any spatial heuristic, and instead drop geometric data, which also remains true for most sparse graphs [Zeng and Curch, 2008].

3.2 Structured Query Language

Structured Query Language, also called SQL, is a programming language for interacting with relational databases, such as MySQL. It was invented in the 1970s and has become highly standardized [Loshin, 2022].

A simple example could look like the code segment below.

```
SELECT abstract FROM papers WHERE subject LIKE '%MySQL%';
```

This query will search data from the table *papers*, as defined by *FROM*. It will only pick the papers with subjects that contain the phrase *MySQL*, as indicated by *WHERE*. Finally, it will return the abstract of these papers as decided by *SELECT*.

Advantageously, all data searching and partition is done on the server side, which saves broadband and leaves the client to worry about other work. The language is also fairly legible to everyday people, though more complex queries can get strikingly messy.

3.3 MySQL Test Run

MySQl Test Run, or MTR, is a perl script that runs individual test cases stored in a structured test directory. MTR can be run using the generalized shell script below.

```
shell> [dir]/mtr [options] [test_name] ...
```

The *test_name* argument represents the test to be run. Multiple tests can be run simultaneously by adding more *test_name* arguments. If no specific *test_name* is provided, MySQL Test Run will run all test files in the test directory. Each test file has a corresponding result file stored in a result directory. The result file has to be named identically to its equivalent test file and contains the expected output of the test. When the test file is run, its output is compared with the contents of the result file. The test will only pass if no differences are detected [Oracle, 2018]. Lastly, *dir* is simply the file in which MTR is kept.

3.4 Compiling with Yet Another Compiler-Compiler

Yet Another Compiler-Compiler, more commonly known as YACC, is a parser generator used to easily define compiler syntax. For example, the YACC code below defines a very simple calculator.

```
%{
#include <stdio.h>
#include <stdlib.h>
extern int yylex();
```

```
void yyerror(char* msg);
}%
%union { float f; }
%token <f> NUM
%type <f> A P
%%
S : A        {printf("%f\n", $1); /* final action */}
  ;
A : A '+' P { $$ /* res */ = $1 /* A */ + $3 /* P */;}
  | A '-' P { $$ = $1 - $3; }
  | P        { $$ = $1; }
  ;
P : P '*' NUM { $$ = $1 * $3; }
  | NUM        { $$ = $1; }
  ;
%%
void yyerror(char* msg){
    fprintf(stderr, "%s\n", msg);
    exit(1);
}
int main(){
    yyparse();
    return 0;
}
```

For YACC to understand what to parse the Lex code below is also needed to first of all define the token *NUM*.

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "y.tab.h" // from yacc -d [yacc.y file]
}%
%%
[0-9]+(\.[0-9]+)?    { yylval.f=atof(yytext);return NUM;}
[-+*]                { return yytext[0]; /* grab -+* */ }
[ ]                  { ; /* ignore spaces */ }
%%
```

When compiled the calculator will first combine anything by multiplication which is separated by a '*', before adding or subtracting the blocks that remain separated by '+' or '-' respectively, until one number remains. This final number will then be printed by *printf*. In other words, this short snipped enforces order of operations for addition, subtraction, and multiplication. Lex defines the tokens/words of the parser whilst YACC defines the syntax and grammar.

A clear advantage with YACC is how easily it integrates with C and by extension C++. This is due to all actions, outside of parsing, being defined in C, as seen above, or C++.

3.5 Data types

3.5.1 JavaScript Object Notation

JavaScript Object Notation, or JSON, is an open standard data exchange format. With JSON, a person could be represented as seen in the code segment below.

```
{
    "name" : "Arthur",
    "age" : 42,
    "friends" : [ "Martin", "Ford" ]
}
```

Here, *name* is a string, *age* is an integer, and *friends* is an array of strings. As is evident, this format is very readable to humans. Furthermore, JSON support is well integrated into JavaScript, the scripting language of the web. For these reason JSON has become a hallmark of user friendly web technologies, and is widely used therein [Safris, 2022].

3.5.2 Hash tables

Hash tables are data structures commonly used for their constant lookup time, regardless of how many elements they contain. Each element has one key and a corresponding value. When *hashed*, the key defines the location of the value. To store an edge value with a corresponding edge id as they key, a simple hash function could look like the one below.

```
unsigned int hash(int key) {
    return key % hash_map_capacity;
}
```

In which case, persisting edges with ids 1, 5, 7, 2, and 10, in that order, would result in the memory layout seen below. *This assumes that hash map capacity is eight. Key value pairs are represented as '[key:value]', and '->' indicates a reference to the next element in a singly linked list.*

```
0: null
1: -> [1:Edge]
2: -> [2:Edge] -> [10:Edge]
3: null
4: null
5: -> [5:Edge]
6: null
7: -> [7:Edge]
```

Admittedly, extracting the edge with id 10, would take slightly longer than for any other edge. Collisions are therefore highly unfavoured. Utilizing STL implementations, such as *unordered_map* or *unordered_multimap*, ensures properly tested methods to minimize collisions. For example, keys that share common factors with the capacity are more likely to collide. Therefore, forcing the hash table capacity to always be a prime number will lessen the likelihood of collisions [Corman, 2020].

3.5.3 Heaps

A heap is both a memory segment and a data structure, but this section will only cover the data structure. It is worth noting that for this section, and this section only, nodes will refer to a number in a heap tree with relative position data.

In simple terms, a heap is a binary tree, meaning a tree with at most 2 sub-nodes per node, where all nodes are ordered by size. In a min heap all nodes are smaller than their sub-nodes, or greater in a max heap. All nodes are stored contiguously in an array, and a potential min heap could look like the representation below.

```
  Tree:          |        Array:
    3            |
   / \           |    [3, 4, 9, 9, 5]
  4   9          |
 /\              |
9  5             |
```

To maintain this structure most efficiently after mutation, some useful tricks have been invented. For this example min heaps will still be used. When inserting a node it will be placed in an unused spot furthest to the left on the highest unfilled row. Afterwards, the node will be continually moved up the tree by swapping positions with to node above so long as the node above is greater. Extracting the smallest node is slightly more demanding. First, the top node will be extracted, this is the result. Then the top node will be replaced by the node on the lowest row furthest to the right. Afterwards, it will move down the tree by continually swapping positions with any smaller sub-node, always preferring the left sub-node, until no sub-nodes are left or both sub-nodes are greater.

Advantages of this data structure lays first and foremost in being able to cheaply keep a changing set of numbers *sorted* with minimal overhead. Or more correctly, to maintain an ascending or descending order when sequentially going through the set, number by number.

3.6 Enhancement projects

An enhancement project takes an existing system and adds new capabilities to it. Enhancement projects may also involve adding new reports, correcting defects, and modifying functionality to comply with revised business needs or rules. This must not be confused with a maintenance project, as the terms are used interchangeably, which fixes bugs and improves on already existing functionality within a project [Anneliese Mayrhauser and Lang, 1998]. One significant difference enchantment and maintenance projects have from more traditional development projects, which develop a project from scratch, is the significantly increased time spent analyzing old code. Code analysis consists of trying to understand code. This is traditionally done through methods such as reading and stepping through the source code, asking people who have familiarity with the code, as well as trying and failing [Xia, 2018].

3.7 Agile development

Agile development is a software development method that revolves around self-organizing teams, evolutionary development, adaptive planning, early delivery, and continual improvement. It is based on the principles in the agile manifesto released in 2001 [Beck et al., 2001]. The core principles of agile include:

- Collaboration
- Open communication
- Trust and Independence
- Efficiency
- Continuous delivery

The goal of agile development is to make a product which satisfies the customer while

removing unnecessary overhead and focusing on the product.

4 Choice of Method and Technology

This chapter will explore how the project took advantage of different tools and methods to complete development in the most sensible and efficient manner. A general description of the applied *modus operandi* will also be paraded, but a more in depth look of the process, as it unfolded, can be found in section 6.3 (Process).

4.1 Choice of technology

Technologies used during the project, their advantages or just why they were necessary, will be explored and hopefully justified in this section.

4.1.1 Git

Version control systems are especially useful and completely unavoidable in any serious coding project, but why exactly git?

Git is an exceedingly popular Version Control System used to track file changes, typically in software development projects. It is used here, quite simply, because it was the system already in use by MySQL. That is not to say that it would not be chosen anyway. Git is so immensely popular and familiar that using it was all but unavoidable.

A separate repository, forked from MySQL's open source project on GitHub, was the project's beginning. This was primarily necessary, because write access to the main repository requires registration in Oracle's contributor registry. Something which was not put in place before the end of this project.

4.1.2 GDB

GDB, also known as the GNU debugger, is a portable debugger that runs on most UNIX-like systems. What does it do, and why was it chosen?

GDB allows setting breakpoints throughout the code and investigate its behavior line by line. Furthermore, it allows dynamically viewing how memory is mutated during program execution, a crucial tool when debugging [Free Software Foundation, 2022]. Most importantly, it is lightweight, quick to boot, and straightforward to use. It quickly became a vital tool for this project's development, especially when debugging.

GDB was recommenced by Oracle, primarily since it enjoys great integrated support in many of their testing systems, such as MTR. Familiarity through NTNU was basic, but easy set-up and an intuitive CLI made GDB hard to dislike.

4.1.3 MySQL's test framework

Oracle has created their own testing framework built into MySQL. It takes SQL queries to be tested as input and compares the given result with an expected result. The testing is separated into two different files where the queries are stored in test files, and the results are stored in result files. The test will pass only if the given output from the test file is exactly the same as the expected content in the corresponding result file.

This testing system was relatively easy to get into, especially for those that know a bit of SQL. With the previous knowledge that had been gained from studies at NTNU, the testing wasn't too hard to get into with help from already written tests. Not only is it a requirement from Oracle, but also a powerful tool [Oracle, 2022a].

4.1.4 Google unit tests

Unit testing was implemented using MySQL's g-unit-test testing framework, which relies on the Test Anything Protocol (TAP) and Google Test (unit-testing). Unit tests were employed to isolate the algorithms and test them in order to determine it they work as intended. To run TAP with C++ MySQL uses their own conversion library [Oracle, 2016]; other than a little CMake magic in some CMakeLists.txt files, it was fairly straightforward to use. Other unit testing frameworks were available, but following previously set standards, in this case TAP, is typically more optimal.

4.2 Choice of method

4.2.1 Agile development

The concept of agile development was familiar from previous studies. Its practice includes discovering requirements and solutions through communication with the end-user and the collaborative effort of self-organizing and cross-functional teams. Although agile development was meant for larger groups, the principles could also be applied to smaller teams.

Working software

The project's focus has been to deliver a working product with as many of the specified features as possible. Working software is a critical element in the agile manifesto. It was therefore important to get a minimum viable product up and running as soon as possible, before continuing to add features.

Individuals and cooperation

Good communication and motivated developers are an absolute necessity to have significant development. Being open and sharing knowledge has been a priority for this project. The threshold to ask for guidance has been low, and help has always been fast and willing. Any conflict that has surfaced during the duration of the problem was quickly handled in an efficient and civilised manner. For example, discussing details around code implementation was a frequent event.

Customer communication

As the project will be implemented directly into the MySQL codebase, tight cooperation with the product owner was preferable. Frequent meetings have therefore been vital to keeping everyone updated and on the right path. Meetings were also a good platform for asking questions about the codebase and uncertainties within the project.

Reflection

The ability to sit down and reflect on how to improve the effectiveness of the workday has been a part of the process. Although not omnipresent, it has helped streamline the work process and benefited productivity to some extent. For example, it was concluded that levels of communication within the team regarding what team members were doing was lacking in

the beginning. This caused some confusion, but was improved after only a few initial hiccups.

Key points from agile development

- **Frequent delivery**
  Delivering stable software at a regular rate is a key element in agile development. During this project, it has been prioritized first to make a core minimalistic version of the software and then build upon it in minor incremental *releases*. Working like this made it a lot easier to find bugs that appeared throughout the project. In addition, it gave an overview of which parts were completed and which were still under development. This reduced the risk of fatal mistakes going undiscovered until the very end of the project. Another benefit came in the form of motivational boosts from rapidly finishing new features, allowing for more flexibility. Changing the project's direction and adapting the successive iterations became easier when seeing and using the software.
- **Integrated testing**
  In agile development, it is essential to test early and often. This should be done throughout the project's life cycle. Agile does not have a rigid testing phase and developers are much more heavily engaged in constant testing. It has been a core element throughout this project as it was vital that each feature worked issueless before moving on to the next.
- **Frequent customer contact**
  A good relationship with the customer has helped guide the end result in their desired direction. Giving them frequent updates on progress and results has helped to keep them satisfied and push the project forward. Access to their experts on the source code has also been a great help when stumbling upon difficulties.
- **Team building**
  Functioning well as a team lead to a more stress-free and productive workday. In addition, spending time together outside of work on leisure activities has been a rewarding activity both on collaboration and for the team in general.

4.2.2 Pair-programming

In parts of the project process, pair-programming has been utilized. With the complexity of some parts of the code combined with the severe intricacy and entanglement of the MySQL source code, it was sometimes quite beneficial for an extra set of eyes to best complete the task. This also helped to gain a better understanding of the code for both parties which was crucial for the further development of the code. It made it easier to make the correct decisions as both programmers were involved and invested in a problem where one programmer might struggle. The quality and correctness of pair-programmed code was also a source of improvement. Pair-programming was avoided for trivial and less demanding tasks where two people would be a waste of time. For tasks such as this, the faster pace of solo-programming and dynamic schedules for the team was more beneficial. This was largely taken advantage of during the more critical parts of the project, such as setting up the project structure within the repository and weaving the routing engine into MySQL's source code.

4.3 Structure and architecture

4.3.1 Google code style

The Google C++ style was used as it is the one used by Oracle and this project had to comply with that [Google, 2022].

4.4 Work roles and distribution

There has been no official role distribution for this project as it would not be beneficial or purposeful to appoint a leader or something similar for a group consisting of two people. Both members have contributed in all areas from planning, problem solving, implementation and documentation.

When the need for pair-programming was unnecessary, task were divided by the team members based on knowledge and interest for the given tasks. Both parties got to work with interesting jobs and collaboration was strong throughout the whole process. As all the tasks were quite similar in category, there were never any need for team members to specialize in particular fields.

5 Result

Chapter 5, videlicet, *Result*, prevails, not in exploring nuance, for which section 6 (Discussion) exists, but rather in reporting the exact state of the product.

5.1 Geospatial routing engine

5.1.1 Dijkstra's algorithm

Dijkstra's algorithm was implemented as an independent functor, only including standard C++ libraries. It has two public methods, its constructor, Dijkstra::Dijkstra, and main method, Dijkstra::operator(). They take the following arguments:

```
Dijkstra::Dijkstra(
  edge_map, heuristic, allocate
);
std::vector<Edge> Dijkstra::operator()(
  start_node_ID, end_node_ID,
  total_cost, popped_points, stop
);
```

Dijkstra

- **edge_map** (std::unordered_multimap<int, const Edge>)
  A hash map containing all the edges in the graph with associated keys equal to their source node, here being the node id Edge.from.
- **heuristic** (std::function<double(int node_ID)>)
  Heuristic function which should return expected distance to the end node from given node id. **Defaults to** null heuristic if no parameter is given, which means that the functor will be equivalent to pure Dijkstra.
- **allocate** (std::function<void*(size_t n)>)
  External memory allocation callback function to, for example, keep track of memory usage. Must return a pointer to a valid memory address of at least n bytes of contiguous memory, this memory will never be deallocated by Dijkstra. Called whenever more memory is needed, and **defaults to** managing its own memory if no paramater is given.

operator()

- **start_node_ID** (int)
  Id of node where the path must begin
- **end_node_ID** (int)
  Id of node where the path must end
- **total_cost** (double*)
  Pointer for returning the total cost of traversing the resulted path (unchanged if no path exists). **Defaults to** nullptr if no parameter is given, meaning that total cost will not be returned.

- **popped_points** (int*)
  Pointer for returning number of points visited by the algorithm whilst searching for the end node (unchanged if no path exists). **Defaults to** nullptr if no parameter is given, meaning that number of visited nodes will not be returned.
- **stop** (std::function<bool()>)
  Callback function to externally stop execution if for example to much time has passed. Execution will stop if it returns true, and it is called once every time a new node is visited. **Defaults to** always returning false if no parameter is given, meaning that execution won't stop unless path was found or no path exists.

5.1.2 Unit testing

Currently, the Dijkstra functor's unit test is a CMake style test named gis_dijkstra-t. Testing occurs both with an euclidean distance based heuristic and without any heuristic on two handcrafted graphs. Everything from path and path cost to number of nodes visited is tested and verified to coincide with expected results.

The euclidean heuristic test was done on the data set seen below. First, spatial data can be seen here.

```
typedef std::pair<double, double> Node;
Node nodes[] {
    {  0,  0 }, // A 0
    {  2,  1 }, // B 1
    { -1, -1 }, // C 2
    {  2, -2 }, // D 3
    {  1,  3 }, // E 4
    {  4,  3 }, // F 5
    {  4,  1 }, // G 6
    {  3, -1 }, // H 7
    {  6,  2 }, // I 8
    { -2,  1 }, // J 9
    { -3, -2 }, // K 10
    { -1,  3 }  // L 11
};
```

And finally the edges can be seen here.

```
Edge edges[] {
    // Edge{ id, from, to, cost }
    Edge{  0,  0,  2,  1.5  }, // A 0 -> C 2
    Edge{  1,  0,  3,  2.9  }, // A 0 -> D 3
    Edge{  2,  0,  5,  5.0  }, // A 0 -> F 5
    Edge{  3,  3,  1,  3.0  }, // D 3 -> B 1
    Edge{  4,  2, 11,  2.0  }, // C 2 -> L 11
    Edge{  5,  2, 10,  2.4  }, // C 2 -> K 10
    Edge{  6,  2,  9,  2.4  }, // C 2 -> J 9
    Edge{  7,  9,  4,  3.65 }, // J 9 -> E 4
    Edge{  8,  4,  5,  4.0  }, // E 4 -> F 5
    Edge{  9,  5,  8,  2.4  }, // F 5 -> I 8
    Edge{ 10,  5,  6,  2.0  }, // F 5 -> G 6
    Edge{ 11,  0,  7,  3.2  }, // A 0 -> H 7
    Edge{ 12,  7,  6,  2.3  }, // H 7 -> G 6
    Edge{ 13,  8,  6,  2.3  }, // I 8 -> G 6
```

```
    Edge{ 14,  1,  6,  2.0  }  // B 1 -> G 6
};
```

For better clarity, a visual representation of the graph can be found in the below figure (Figure 1). ¨

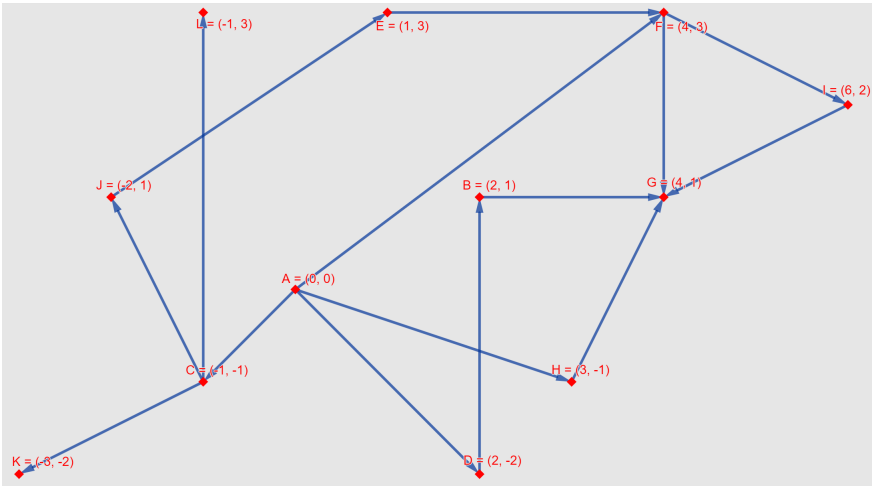


Figure 1: Euclidean heuristic test graph

When going from node A (#0) to G (#6), it was expected that the shortest path would go through edge 11 and then 12, meaning nodes A (#0), H (#7), and then G (#6). After this had been verified by the test, edge 12 was removed, and a new shortest path was tested before again removing an edge from that path. Ultimately, three paths were tested, and the expected results are those seen in the code comments below.

```
// [no edges have been removed]
// A 0 -> H 7 -> G 6         : 5.5m (test 1)

// [edge 12 has been removed]
// A 0 -> F 5 -> G 6         : 7.0m (test 2)

// [edges 12 and 10 have been removed]
// A 0 -> D 3 -> B 1 -> G 6 : 7.9m (test 3)
```

5.2 SQL code execution

5.2.1 Compiler syntax

SQL syntax was defined in Yacc to mimic any other aggregation function; parameters are separated by commas and surrounded by parentheses. With a total of 7 parameters the syntax is as follows.

```
ST_SHORTEST_DIR_PATH (
    /* edge */
    ID, from_ID, to_ID, cost, to_geom,
    /* path */
    begin_ID, end_ID
);
```

Edge parameters

- **ID** (INT)
  Edge id, primarily used to identify edges in the resulting path.
- **from_ID** (INT)
  Node id, defines an edge's origin.
- **to_ID** (INT)
  Node id, defines an edge's destination.
- **cost** (DOUBLE)
  Edge cost/weight, defines the cost of traversing an edge.
- **to_geom** (POINT)
  Node geometry, defines the geometry of a destination node in the form of a SQL Point. Can be null, but must either be null for all rows or none. *It is worth noting that this parameter only serves to speed up the routing engine by using an appropriate spatial heuristic based on the defined coordinate system, but it can also lead to incorrect results see section 6.1 (Geospatial routing engine) for more.*

Path parameters

- **begin_ID** (INT)
  Path start, defines the origin of the path to be found.
- **end_ID** (INT)
  Path end, defines the destination of the path to be found.

5.2.2 Returned result

Function output is formatted in JSON. A possible stylized result could look like the segment below.

```
{
    "cost": 42.0,
    "path": [
        {"id": 3, "cost": 31.4 },
        {"id": 9, "cost": 10.6 }
    ],
    "visited_nodes": 7
}
```

JSON tokens

- **cost** (DOUBLE)
  Path cost, sum of all edges' costs along the *path*.
- **path** (JSON_ARRAY)
  Path, array of edges representing one possible shortest path from *begin_ID* to *end_ID*. Each edge is defined by its id, but also contain its cost.
- **visited_nodes** (INT)
  Number of nodes visited by routing engine. Gives a rough estimate of the efficiency of the used routing engine heuristic. *Not to be confused with number of nodes along path, which always equals the size of the path array plus one (no loops).*

5.2.3 Exception handling

Potential errors, their respective causes, and names, can be found in the list below.

- **No path exists** (ER_NO_PATH_FOUND)
  Caused by no set of edges existing so that a contiguous path can begin at *begin_ID* and end at *end_ID*.
- **Ambiguous edge id** (ER_DUPLICATE_EDGE_ID)
  Caused by incorrectly defining multiple edges with the same id.
- **Cost less than or equal to zero** (ER_NEGATIVE_OR_ZERO_EDGE_COST)
  Caused by setting *cost* to zero or less.
- **Edge loop** (ER_EDGE_LOOP)
  Caused by edge originating from the same node as it ends.
- **Path begins where it ends** (ER_START_AND_END_NODE_CONFLICT)
  Caused by *begin_ID* equalling *end_ID*.
- **Inconsistent null geometry** (ER_INCONSISTENT_GEOMETRY_NULLNESS)
  Caused by one or more *to_geom* rows being null whilst one or more *to_geom* rows are not. All *to_geom* rows must be null or none of them.
- **Ambiguous coordinate system** (ER_GIS_DIFFERENT_SRIDS_AGGREGATION)
  Caused by *to_geom* rows having different spatial reference identifiers (SRID) and thus pertaining to separate coordinate systems.
- **Node geometry not a point** (ER_GIS_WRONG_GEOM_TYPE)
  Caused by any *to_gem* row not being of type POINT (gis::Geometry_type::kPoint).
- **Ambiguous node geometry** (ER_GEOMETRY_REDEFINED)
  Caused by incorrectly defining the same node geometry twice with differing values.
- **Cost not a real number** (ER_WRONG_ARGUMENTS)
  Caused by cost neither being a real number nor being convertible to a real number. Non real numbers will be converted automatically if possible.
- **Id not an integer** (ER_WRONG_ARGUMENTS)
  Caused by *ID*, *from_ID*, *to_ID*, *begin_ID*, and/or *end_ID* not being integers or convertible to integers. Non integers, for these parameters, will be converted automatically if possible.
- **Ambiguous path origin and/or destination** (ER_WRONG_ARGUMENTS)
  Caused by *begin_ID* or *end_ID* not being equal for all rows.
- **Invalid use of null** (ER_WRONG_ARGUMENTS)
  Caused by incorrectly setting *ID*, *from_ID*, *to_ID*, *cost*, *begin_ID*, or *end_ID* to null. Only *to_geom* can be null.

5.2.4 Integration testing

Integration testing is done with the entire system, meaning a whole instance of a SQL server paired against a SQL client. As alluded in the theory chapter, this is done through MTR. The test file is named st_shortest_dir_path.test and expected result is found in st_shortest_dir_path.result. Testing is done in three stages.

Stage one tests the whole implementation without geometry data, and thus without heuristics. Multiple paths are checked and verified to ensure that the function is consistently correct. *GROUP_BY* on edge type is also tested to ensure proper memory management. The table and a reduced representation of values used in one of the tests in this stage can be found below.

```
/* Table */
CREATE TABLE edges (
    id INT PRIMARY KEY,
    from_id INT,
    to_id INT,
    cost DOUBLE,
    type ENUM(
        "car",
        "bike",
        "pedestrian"
    )
);
/* Data */
INSERT INTO edges VALUES
( 10, 0, 1,   2.0,  "car"        ),
( 11, 1, 2,   3.0,  "car"        ),
( 12, 0, 2,   8.0,  "car"        ),

( 20, 0, 1,  20.0,  "bike"       ),
( 21, 1, 2,  25.0,  "bike"       ),
( 22, 0, 2,  40.0,  "bike"       ),

( 30, 0, 1,  80.0,  "pedestrian" ),
( 31, 1, 2, 100.0,  "pedestrian" ),
( 32, 0, 2, 200.0,  "pedestrian" ),
( 33, 1, 3,  40.0,  "pedestrian" ),
( 34, 3, 2,  20.0,  "pedestrian" );
```

In essence, this test consisted of the automated integration test query, and its expected result, seen below.

```
/* Query */
SELECT ST_SHORTEST_DIR_PATH (
    id, from_id, to_id, cost, NULL, 0, 2
) FROM edges GROUP BY type;
/* Expected result */
{ /* car */
    "cost": 5.0,
    "path": [
        {"id": 10, "cost": 2.0},
        {"id": 11, "cost": 3.0}
    ],
    "visited_nodes": 2
}
{ /* bike */
    "cost": 40.0,
    "path": [
        {"id": 22, "cost": 40.0}
    ],
    "visited_nodes": 2
}
{ /* pedestrian */
    "cost": 140.0,
    "path": [
```

```
            {"id": 30, "cost": 80.0},
            {"id": 33, "cost": 40.0},
            {"id": 34, "cost": 20.0}
        ],
        "visited_nodes": 3
}
```

This is also a demonstration of a valid use case, namely, deducing what type of transportation is optimal for a given trip.

Stage two tests the whole implementation with geometry data, and thus with a relevant heuristic. Currently it only uses the euclidean distance, as expected with SRID zero. A much larger data set than in the previous stage was used, also with added geometry points. New tables and an exemplary cutout of this data set can be seen below.

```
/* Tables */
CREATE TABLE nodes (
    id INT PRIMARY KEY,
    node POINT NOT NULL SRID 0
);
CREATE TABLE edges (
    id INT PRIMARY KEY,
    from_id INT,
    to_id INT,
    cost DOUBLE,
    CONSTRAINT node_ref
        FOREIGN KEY (to_id)
        REFERENCES nodes(id);
);
/* Data (reduced) */
INSERT INTO nodes VALUES
( 0, POINT(  0,  0 ) ),
( 1, POINT(  2,  1 ) ),
( 2, POINT( -1, -1 ) ),
( 3, POINT(  2, -2 ) ),
( 4, POINT(  1,  3 ) ),
( 5, POINT(  4,  3 ) ),
( 6, POINT(  4,  1 ) ),
( 7, POINT(  3, -1 ) );

INSERT INTO edges VALUES
( 0, 0, 1,  3.0 ),
( 1, 0, 2,  5.0 ),
( 2, 0, 3,  8.0 ),
( 3, 1, 4,  1.0 ),
( 4, 4, 3,  2.0 ),
( 5, 3, 5,  6.0 ),
( 6, 2, 5, 12.0 ),
( 7, 4, 5, 10.0 );
```

One edge in the found path is removed before finding a new path between the same nodes, just as in subsection 5.1.2 (Unit testing). This is analogous to redirecting traffic because of a traffic jam or a road being temporarily closed. A shortened version of this test can be seen below.

```
/* Query */
SELECT ST_SHORTEST_DIR_PATH (
    edges.id, edges.from_id, edges.to_id,
    edges.cost, nodes.node, 0, 5
) FROM edges
JOIN nodes ON nodes.id = edges.to_id;
/* Expected result */
{
    "cost": 12.0,
    "path": [
        {"id": 0, "cost": 3.0},
        {"id": 3, "cost": 1.0},
        {"id": 4, "cost": 2.0},
        {"id": 5, "cost": 6.0}
    ],
    "visited_nodes": 7
}
/* Disabling edge #3 */
UPDATE edges SET cost = 1000.0 where id = 3;
/* Expected result after re-executing the query above */
{
    "cost": 14.0,
    "path": [
        {"id": 2, "cost": 8.0},
        {"id": 5, "cost": 6.0}
    ],
    "visited_nodes": 5
}
```

It is worth noting that all above paths are identical to those found without any heuristic. The only difference is a lower *visited_nodes* number.

Stage three tests the input validation/Exception handling. This was done in order to validate the function's sturdiness. Everything that could go wrong was tested to make sure the correct errors were thrown at the correct time. In the example below, it is verified that the proper error is raised when creating two edges with identical ids. *Irrelevant code segments have been redacted and replaced by "..." for clarity*.

```
/* Data */
INSERT INTO edges VALUES
( 0,  ... ),
( 0,  ... )
;
/* Query */
SELECT ST_SHORTEST_DIR_PATH (
    id, ...
) FROM edges;
/* Expected result */
ERROR HY000: Error on st_shortest_dir_path,
edge id 0 used more than once. Ids must be unique
```

5.3 Performance

All performance tests were done manually, and show the general *fitness* of the solution.

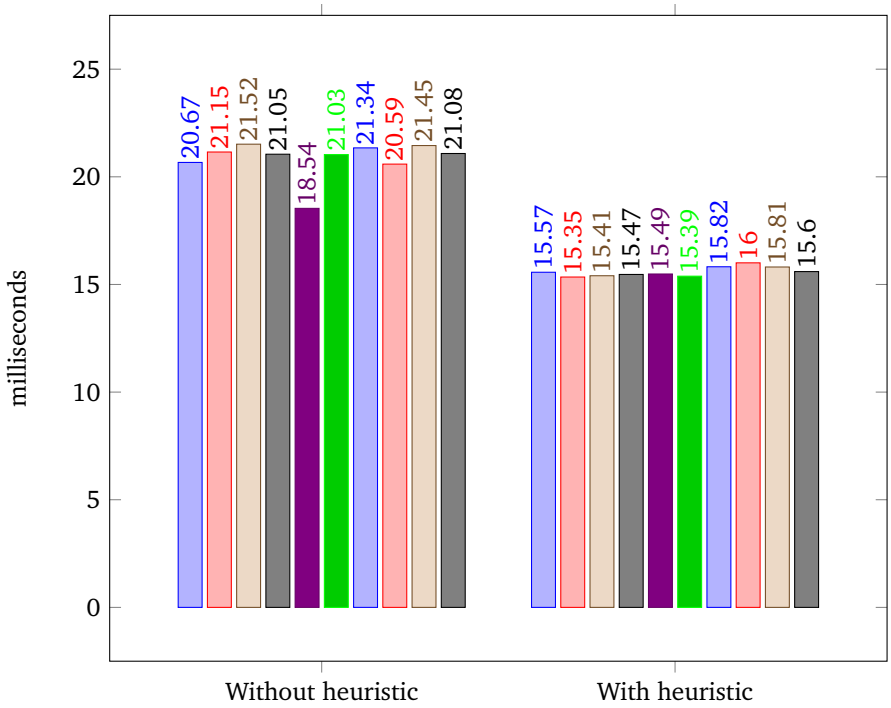
5.3.1 Routing engine

These test were only done on the routing engine, not the entire system. Two tests were done. The first would ensure correctness of paths found, whilst the other was meant to prove the temporal advantage of heuristics.

To initially test the correctness of the routing engine, a reduced data set of northern Europe, based on Open Street's map database, was used. It contained a little over 6 million nodes and 14 million edges. Some data had been pruned and reduced to fit inside available system memory, and no heuristics were utilized. Finding the shortest path from Oslo to Berlin took an average of 3 858 milliseconds over one hundred test runs. Comparison with a similar algorithm, from the C++ boost library, confirmed that this and other tested paths were exactly as expected.

When testing heuristics, two different versions were compared. In both cases, the graph seen in Figure 1 was used, with the data declared above the referenced figure. One used no heuristic, and the other used a version of Equation 3.1 with h equal to Equation 3.5 (Euclidean distance). After running ten tests on both, where each test calculated the shortest path from node A to G ten thousand times each time, results were gathered. Using a heuristic required on average 15.59 milliseconds, whilst not using a heuristic landed on 20.84 milliseconds. The heuristic did, on the other hand, require 34.8% more system memory for node data.

A summation of measured time usage is found in the graph below. As mentioned, every measurement is roughly scaled by a factor of ten thousand.



5.3.2 Entire system

Tests mentioned here were applied on the entire system. Three test were performed in total. All tests used the same graph seen in Figure 1, which was also used to compare heuristics in subsection 5.3.1 (Routing engine). The statistics found here are therefore somewhat comparable to those found above, although not directly.

All tests used the same tables, which are identical to those used in stage two of the integration test, as seen in subsection 5.2.4 (Integration testing). However, each test had its own SQL query. For path finding without any heuristic, the SQL query below was used.

```
SELECT ST_SHORTEST_DIR_PATH(
    id, from_id, to_id, cost,    /* Edge */
    NULL,                        /* Node */
    0, 6                         /* Path */
)
FROM edges;
```

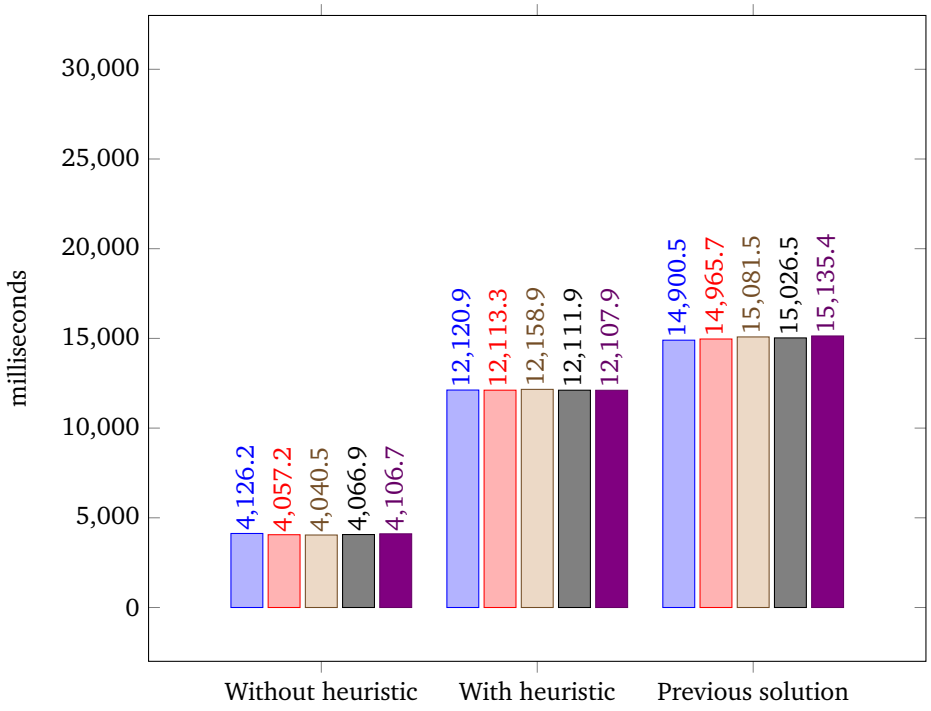
For path finding with a euclidean heuristic, the SQL query below was used. *To avoid measuring the time it takes to join tables together, the actual JOIN was done before the test*.

```
SELECT ST_SHORTEST_DIR_PATH(
    edges.id, edges.from_id, edges.to_id, edges.cost,
    nodes.node,
    0, 6
)
FROM edges JOIN nodes ON nodes.id = edges.to_id;
```

To compare this paper's solution to a solution that existed before, the SQL query below was used. *This method is an adaptation of mr. sticky bit's method from Stack Overflow, and it does not support heuristics [Stack Overflow, 2019].*

```
WITH RECURSIVE
cte AS
(
SELECT p.destination,
       concat(p.source, '->', p.destination) path,
       p.cost length
       FROM path p
       WHERE p.source = 0
UNION ALL
SELECT p.destination,
       concat(c.path, '->', p.destination) path,
       c.length + 1 length
       FROM cte c
            INNER JOIN paths p
                       ON p.source = c.destination
       WHERE c.destination <> 6
)
SELECT c.path
       FROM cte c
       WHERE c.destination = 6
       ORDER BY c.length
       LIMIT 1;
```

These SQL queries were run ten thousand times to again find the shortest path from node A to G. This was repeated five times for each query and resulted in the timestamp statistics found below.



5.4 Call stack

To conclude the chapter, a quick overview of how all the aforementioned parts fit together will be provided here. This simplified model is probably better understood in the light of section 6 (Discussion).

A SQL call to ST_SHORTEST_DIR_PATH would go directly to YACC. YACC defined the SQL syntax and called Item_sum_shortest_dir_path as seen below. *Less important parts have been replaced by '...' for clarity*.

```
ST_SHORTEST_DIR_PATH_SYM '(' in_sum_expr ',' ... ')' {
    ...
    $$ = NEW_PTN
        Item_sum_shortest_dir_path (
            @$, args, ...
        );
}
```

Item_sum_shortest_dir_path was at the core of the implementation, and began with the aggregation process, which includes the C++ code below. *Less important parts have been replaced by '...' for clarity*.

```
bool Item_sum_shortest_dir_path::add() {
    ...
    // get data
    // (args contain SQL parameter data
    // provided by YACC)
    id =        args[0]->val_int();
    from_id =   args[1]->val_int();
    to_id =     args[2]->val_int();
    cost =      args[3]->val_real();
```

```
    add_geom(   args[4], to_id, ...);
    ...
    // input validation e.g.
    if (cost <= 0){
        my_error(...);
        return true;
    }
    ...
    // persist edge
    Edge edge = Edge{ id, from_id, to_id, cost };
    m_edge_map.insert(std::pair(from_id, edge));
    ...
    // no error
    return false;
}
```

After all data is gathered the routing engine *Dijkstra* is called as seen below. *Less important parts have been replaced by '...' for clarity*.

```
bool Item_sum_shortest_dir_path::val_json(
    Json_wrapper *wr
) {
    // THD represents a client connection
    const THD *thd = base_query_block->parent_lex->thd;
    // stops the routing engine if e.g. no more time
    static std::function stop_dijkstra =
        [&thd]() -> bool {
            return thd->is_error() ||
                   thd->is_fatal_error() ||
                   thd->is_killed();
        };
    // trivial heuristic for SQL call without geometry
    std::function heuristic = [](const int&) -> double {
        return 0.0;
    };

    if (!m_point_map.empty()){
        ...
        const gis::Geometry *end_geom =
            m_point_map.at(m_end_node).get();
        // spatial heuristic
        heuristic =  [this, end_geom]
            (const int& node) -> double {
            static gis::Distance dst(NAN, NAN);
            return dst(
                end_geom,
                this->m_point_map.at(node).get()
            );
        };
    }
    ...
    Dijkstra dijkstra(&m_edge_map, heuristic,
        [...](const size_t n) -> void* {
            // my_malloc registers memory use
```

```
            void* p = my_malloc(..., n, ...);
            ...
            return p;
        }
    );
    std::vector<Edge> path = dijkstra(
        m_begin_node,
        m_end_node,
        &cost, &popped_points,
        stop_dijkstra
    );
    ...
    // jsonifying path and storing it in param wr
    // (this part is best understood by reading
    // about aggregation in chapter 'Discussion')
    ...
    return false;
}
```

After all this, the result can finally be returned in SQL. *For more details on jsonifying and so forth see* https://github.com/VilhelmSeip/mysql-server *(8.0/sql/Item_sum_shortest_dir_path.cc, 8.0/sql/Dijkstras_functor.cc, and 8.0/sql/sql_yacc.yy) or section 6 (Discussion).*

In short, the call stack looks like this: SQL → YACC → C++ (Aggregator → Routing engine).

6 Discussion

Here, within the chapter *Discussion*, exists a deep dive into the inner workings of the implemented solution. It is meant as an elaboration on section 5 (Result) which, as formerly stated, only covers the more user relevant surface of the product in absolute terms.

6.1 Geospatial routing engine

6.1.1 Dijkstra's algorithm

As in most routing engines a derivation of Dijkstra's algorithm made its way into this implementation. However, some noteworthy peculiarities arose as a byproduct of the very specific requirements living inside this twenty-seven year old codebase.

First and foremost, the used implementation of Dijkstra's algorithm incorporates an external heuristic function, allowing use of any spatial heuristic commonly used in the A-star variation of Dijkstra's algorithm. Due to the always coveted low execution time and mandated support for graphs without any defined geometry the solution contains two separate heuristics; the first and simplest merely being a so called null-heuristic, always returning zero and thus converging the whole algorithm to a pure version of Dijkstra's algorithm. The second heuristic uses MySQL's OpenGIS based geometry library to dynamically guess the distance from any given node to the end node. As seen in subsection 3.1.2 (A-star & heuristics), an advantage of the former is the inherent guarantee of always correctly finding the shortest path as advertised. Sadly this requires more CPU-cycles as it allegorically stumbles, without direction, through the darkness. For the latter heuristic a much more conservative CPU demand is expected for most graphs, especially if based on real world geography, as it will actively zero in on the end node. However, the downside here is that the found path can only be guaranteed to be the shortest if the geometry and cost inputs compute into a positive heuristic, meaning that spatial distance found by the heuristic must be less than or equal to the defined cost. Thus upholding the relation in Equation 3.4. Which is entirely in the hands of the user, but since the user is expected to be a computer engineer, in most cases, this is not going to be an issue.

Furthermore, support for external memory allocation was integrated so MySQL's Malloc_allocator could track memory usage without causing any direct dependency on MySQL's codebase. A separation of state and church, or in this case, a separation of geospatial routing engine and SQL code execution, though not critical, was conveyed as preferable, especially given the unit-testing framework used. Therefore resulting in this optional memory allocation callback parameter formerly named *allocate*.

6.1.2 Unit testing

Testing is always important. Though, integration tests are often considered more indicative of correctness, unit tests are much more precise and efficient. Properties which can easily streamline the debugging processes.

As seen in subsection 5.1.2 (Unit testing), the data sets used in automatic unit testing

were all very small. They were handcrafted to contain relevant anomalies for testing desired behavior. However, a proper stress test, meaning a test with a massive data set, was never programmatically introduced. Although it was originally planned, Oracle did not want large tests in their normal testing stack. Sufficient time to develop a separate system for stress testing was therefore never a priority, but larger tests were done manually as seen in subsection 5.3.1 (Routing engine performance).

As expected, the test done on Figure 1 was also able to show how an appropriate heuristic could reduce number of visited nodes as theorised in subsection 3.1.2 (A-star & heuristics). Partially because, using a heuristic allowed skipping dead ends such as node K, but mostly from avoiding detours.

6.2 SQL code execution

6.2.1 Compiler syntax

As stated, the syntax itself was fairly undemanding, but some notable steps in its evolution are worth mentioning.

Originally, only six parameters were defined, to wit, all but *to_geom* as seen in subsection 5.2.1 (Compiler syntax). These were chosen at Oracle's behest, but were rather clear-cut as they contain the necessary start and end of the path, as well as the indispensable node origin, node target, and edge cost, to define the graph's edges. Edge id, on the other hand, was the only parameter which was not entirely unavoidable. However, in order for the resulting JSON path to be less verbose, and more importantly for the returned edges to refer directly to existing data structures. This parameter was favoured. Alternatively, the whole edge could be printed for every instance in the JSON path. Though perhaps somewhat trivial, this would also result in ambiguity regarding identical edges.

Continuing, a predilection for spatial heuristics surfaced, forcing further function parameter additions. Inspiration was initially drawn from an ISO/IEC standard supplied by Oracle, which turns out to not be freely available for public use at the moment. Nonetheless, the standard advocated for the edge nodes to be defined, not by their numeric ids, but their spatial positions. Howbeit, since all points in SQL were defined by floating point numbers, it was eventually decided that integer ids were a safer bet. Two geometry parameters are admittedly cleaner than two id parameters as well as two geometry parameters, but upon realizing that only one geometry parameter was needed with the latter approach for directed paths, the extra parameter was deemed acceptable. Serendipitously, it happened that the origin node's position could be ignored for all edges. The rationale being that spatial data is only relevant for nodes sorted in the routing engine's heap. This only includes the origin and nodes accessible from the origin, meaning, nodes that are the destination of any edge accessible from the origin, making the position of all edge origins superfluous.

Lastly, regarding choice of heuristics, three options were considered. An eighth parameter could be used as coefficient c in Equation 3.2. For graphs with poorly balanced costs, this could ensure correctness by setting c between zero and one. It also introduces more intuitive user control over when to elevate speed over precision. Furthermore, situations where the quality of the path is irrelevant were perceivable, for example wanting to check if any path exists at all. In which case the user could set c to infinity, causing an implementation of Equation 3.3 to be used for super greedy searching. Another approach considered allow-

ing the user to independently device their own heuristic. Truly making it the most modular strategy, but also the most demanding of future users. It would, like the first solution, require an eighth parameter. However, unlike the first solution, it would also likely introduce myriads of potential edge cases, not to mention typical concerns surrounding remote execution. Ultimately, a third option was implemented. As explained in subsection 6.1.1 (Dijkstra's algorithm), heuristic are dynamically selected to fit provided geometries. In a future iteration, it is believed that all three methods could symbiotically exist in the same solution.

6.2.2 Aggregation

Cumulative congregation of edges, here known as aggregation, serves to form a data set graph. Simply put, it is the process of gathering and packing user input into a graph data structure. This graph is then reduced by the routing engine into a cost minimization optimized path through said graph, which is ultimately returned as the SQL function's result. It is consequently imperative that the aggregation mutates the given edge data into an optimized graph data structure for Dijkstraesque algorithms in the most efficient manner.

Commonly, the favoured data structure is achieved by storing the graph in a contiguous array of edges with indices corresponding to the node id of the edge's origin. Unfortunately, due to the unpredictable nature of user input and all ids being user defined, this would likely be spatially inefficient, and could consume all available system memory with only one data point. Solving this issue required hash tables; an approach which quickly branched into two possible solutions. The former was to simply stores all edges in a so called unordered_multimap, as defined per STL, which is a standard hash table with shared key support. These edges are unaltered and the hash table is used directly in the geospatial routing engine. One major downside of this solution is the repeated hash searches required within the routing engine, which in isolation claimed the largest share of computation time therein. The latter solution offers a workaround to this problem. By converting all input ids into a cohesive range of integers incrementing from zero, hash tables could be replaced by arrays within the routing engine itself. Not only avoiding hash searches entirely, but also ensuring more CPU cache hits from a generally denser data structure. For most geospatial routing applications this would undoubtedly be optimal. However, given the ephemeral complexion of a database query, wherein each graph is necessarily reconstructed for each call to the SQL function, it became more and more evident that the true time complexity laid in constructing the graph, and not in using it. This can be seen in section 5.3 (Performance) where the integration test takes much longer than the corresponding unit test when working on the same graph. In reality, this is not conclusive proof, but it was sufficiently persuasive at the time. The former solution was therefore preferred, as the latter would mean twice as many hashes during construction, to translate node ids, and a much worse spatial and temporal complexity from both constructing and keeping a hash table conversion ledger.

Regarding the memory allocation and usage within the aggregation itself, there are some key notes. Throughout the aggregation all useful data is accumulated onto the heap. Because of the shear size expected coupled with the established aggregation function structure, stack allocations would neither work nor be advisable. An obvious issue with iterative heap allocation is the immense cost of heap allocations over stack allocations. A common optimization revolves around requesting larger chunks of memory with longer intervals between them, thus requiring fewer heap allocations overall. Effectively, this is done through allocating all edges onto a so called Memory_arena owned by a single threaded connection instance encap-

sulated in a THD object. This Memory_arena has the added advantage of also being automatically deallocated after its respective query connection has been executed, further reducing number of deallocations when freeing memory. Other than the stack, the Memory_arena is the most efficient place to keep memory. After this assertion, it falls natural to mention the one place where neither the stack nor Memory_arenas were applicable, scilicet, when converting the result into JSON.

When parsing data into JSON, MySQL used a so called *Json_wrapper*; the wrapper handled the actual conversion, but required input in the form of so called *Json_dom* objects. Therefore, before wrapping raw binary data into JSON, an intermediary conversion into *Json_dom* was required for non string like data. *Json_dom* in itself can be stack allocated and effortlessly handled by the wrapper. However, *Json_dom* objects alone would result in meaningless unlabeled data. A predicament which is illustrated below.

```
    Labeled JSON:
{
    "label" : "corresponding_data",
    "grouped_elements" : [
        "element",
        {
            "other_label" : 2.718,
            "max_size" : 255
        }
    ]
}
    Unlabeled JSON:
{
    "corresponding_data",
    "element",
    2.718,
    255
}
```

Labeling demands pairing labels and *Json_dom* objects into a collective data structure, here named *Json_object*. Moreover, JSON arrays, to define the path, would also need grouping together into a so called *Json_array*. Both *Json_object* and *Json_array* are adaptations of *Json_container*, and *Json_container* demands full ownership of every *Json_dom* it is provided, so it can automatically deallocate them when the *Json_container* itself is deallocated. A functionality which is understandably desirable, but it has the unfortunate side effect of requiring all *Json_dom* elements to be allocated separately on the heap. Otherwise, a segmentation fault would occur as the *Json_container* attempts to free data from a memory address without any previously registered allocation. Ultimately, this results in an exuberant amount of separate allocation calls when converting the result to JSON. Something which is not strictly necessary, but can not be fixed without intrusive alterations to the pre-existing codebase.

Finally, pruning could help reduce memory usage and computation time, but how could it be done without loss of important data? Discarding duplicate edges and more expensive edges, where a cheaper edge is known to exist, was initially considered. Though it would reduce usage of memory, and possibly time, for very dense graphs, it became clear that most road network graphs were too sparse. Thus, the pruning process became too expensive for a large selection of expected inputs. This was unacceptable. A cheaper way of selecting what

to prune was needed. The simplest solution may have been discarding all edges with nodes too far away from from the path's beginning or end. Continuing with this idea, nodes would either always be undefined or only defined for an edge's destination node. When undefined, no pruning could be done, but even with geometry data, the problem remains that important data still could be lost. For example, a land path from Yemen to Djibouti could see its routes through Egypt's Suez canal pruned for being too much of a detour. Which would remove all possible paths. Pruning should therefore be optional, but given time restraints this was never properly implemented.

6.2.3 Exception handling

Few concepts in computer engineering induce quite as much fear as the leviathan that user input is. A primary concern on Oracle's side was allowing unorthodox input which might later be disallowed. To this end, strict limitations were put on said input. All limitations are listed in subsection 5.2.3 (Exception handling), and this section will elaborate the rationale behind some of them.

First among other, why not allow ambiguous edge ids? Computation wise this ambiguity would be of no consequence. However, it would make the resulting output less ideal, and possibly uninterpretable.

Secondly, why not allow costs less than or equal to zero? Explaining the negative costs part is simple; Dijkstra's algorithm does not work with negative costs, in part because negative loops could make the path infinite. Zero costs, on the other hand, are not algorithmically problematic. So why are they not allowed? Primarily, this was done in concord with Oracles preference of strict input validation. Although not technically wrong, zero cost edges are an inefficient waste of memory, and in part time. All nodes connected by at least one zero cost edge could be replaced by a single node without changing the geometry of the graph. Admittedly, zero cost edges could be a useful feature for quickly merging nodes in a large dataset without extensive mutations. This limitation might therefore be removed in the future.

Thirdly, on the topic of not allowing ambiguous geometry, edge ids, path origin, path destination, or SRIDs. All of these have the same rationale; they can only have one value, so assigning multiple will never make sense within the realm of classical computing. Geometries can be ambiguous since every node arriving at the same destination will define the same node, if nodes have geometries. Ambiguity for edge ids and SRIDs is fairly self explanatory, and path origin and destination can be ambiguous since they are defined in every row along the aggregation.

6.3 Process

6.3.1 Unplanned divergence

Any large project is bound to have certain deviations from its scheduled plans. To highlight these deviations, dissimilarities between the planned schedule and the reported hours will serve as the determining data. *These are found in the external appendix, which is not formally part of this paper*. For coherence this summation will be chronological relative to the planned schedule. Further detail, if not sufficiently specified here, regarding what was done on specific dates can be found in the reported hours or the weekly summaries. *These, as well as the preliminary project plan and the poster, are all found in the external appendix*.

Initially, a preliminary project plan and initial setup for the project was scheduled between the 21st of January and the 8th of February. Pre-planning of what to produce and how, as seen in the preliminary project plan, should have been concluded on the 25th of January. Instead, it was technically not finished before the 18th of February. However, that was due to some alterations done after feedback on that date. Work had originally been concluded the month before as expected. Despite a late beginning, as Oracle was unable to supply equipment before the 8th of February, the initial project setup was finished more or less as scheduled on that day. It is worth bringing up that the project officially began in the beginning of January, but given other scholarly obligations, work could not realistically start before the latter half of January.

Implementation of the geospatial routing engine was planned to take place between the 8th and 24th of February. Although an independent geospatial routing engine was not originally mandated, it was coveted. It may seem frivolous to direct efforts into a non compulsory component so early. However, after being advised to wait for Oracle to provide sufficiently sophisticated computing devices for building the MySQL codebase, it was deemed reasonable to start on an engine from scratch whilst waiting. Work began on the 1th of February and the engine was properly implemented and tested by the 18th. On the other hand, critical optimizations were implemented as late as the 23rd of March and some features as late as the 16th of April. As it would have been impossible to foresee these necessities before then, they are not considered to be major or unexpected deviations.

Continuing to the heart of the matter, namely, SQL code execution, a working period was planned from the 24th of February to the 25th of March. Auspiciously, this process began somewhat earlier, on the 21st of February, but quickly dissolved into a long slog. Pre-existing knowledge of SQL was wanting, and code compilation was not an area of expertise. Not to mention the thousands of files, often written in the preceding century, which had to be sifted for relevance. Just understanding the product as it was before adding to it was by far the most frustrating and demanding part, and was, in part, never fully realized. None the less, a breakthrough in comprehension was achieved around week 10 (7th to 14th of March), and productivity soared into a fully functional implementation on the 16th of March. Further development continued all the way into the 16th of April, and the last alteration took place on the 29th of April. Sadly, this ate large chunks of time previously withheld for documentation.

Halfway through the project, preparation and presentation of a project poster was obligatory. This was planned to be handled between the 25th and 28th of March. In the end, the poster was created in a single day, on the 27th, instead of two. It was then presented on the 28th as planned. The poster carries no relevant information beyond this paper.

Testing and optimization was originally planned from the 28th of March to the 11th of April. Optimizations were done in this period, but also sporadically throughout the entire project. Integration testing showed a similar pattern, but unit testing of the routing engine happened more exclusively in this stage as planned. Unit testing began early on the 26th of March and was finished on the 6th of April. These tests included heuristics, which had not yet been implemented in the SQL code execution, and ultimately inspired the last push to get heuristics fully implemented in MySQL by the end of the 7th of April. Not fully implementing heuristics was considered around this time, since input validation was considered more pressing, but time was eventually found for both.

Documentation of the project was optimistically planned between the 11th of April and

the 10th of May. Since the project would not be over before the 20th of May, this allowed some leeway for when things would inevitably take longer than expected. Code documentation naturally took place during code creation, but more in depth documentation began vaguely on the 10th of April, and properly on the 19th. It persisted into the 16th of May, and final touches were done on the 19th, but panic was avoided, given the anticipation of time overshooting.

Conclusively, it is worth mentioning the asynchronous nature of the planned progression. Primarily, this was a byproduct of the small development team, as it seemed challenging to split efforts without excessive overhead from maintaining code coherence. With a greater grasp of the chores involved, it would be easier to allow concurrent development of multiple components. In reality, different parts were indeed developed simultaneously, but not too often. In retrospect, spending more exploratory time in the early stages to more explicitly define concurrent lines of work would have been beneficial.

6.3.2 Other impediments

Beyond any unexpected aberrations from the initial schedule, other issues were inevitably encountered. These were closer to the bones and wires of the project, and this part will therefore be slightly more technical.

When defining SQL syntax multiple problems arose. Syntax was defined in YACC, as is common for compilers. Thousands of lines of YACC code was already in use by MySQL. Little documentation was found, so to understand what did what, aid was requested from Oracle. However, nobody at the office knew the inner workings of MySQL's YACC code, which, according to them, was handled primarily by colleges in Russia. Given current geopolitical conditions, this posed a problem, as Oracle had withdrawn business from that region [Oracle, 2022b]. Some extra trial and error was therefore necessary. Moreover, the lacking of similar functions with more than one or two parameters to use as a template was also suboptimal. For reasons not entirely understood, declaring functions with multiple parameters required many little details to be added in multiple parts of the codebase. Luckily, little YACC code had to be written and only one Lex symbol was added, videlicet, the name of the SQL function.

It has been mentioned more than once, but the complexity of getting accustomed to MySQL's codebase can not be overemphasised. Initially, development came to a halt because of YACC, but terminology related to SQL *items* and aggregation was far from obvious. Logic related to handling all kinds of edge case features and limitations was littered throughout all relevant example code snippets. This is not to say that there was anything wrong with the source code. Based on previous experience, MySQL's codebase is exceptionally ordered and concise. However, being thrown into such a large ecosystem is always a challenge. Not to mention, how everything from memory allocation to error handling had their own unique rules.

6.3.3 Collaboration with Oracle

Collaborating with Oracle was more than pleasant, as they were very accommodating. How collaboration has strengthened the product and made the process more commodious will primarily be the focus in this subsection.

Workspace is notoriously hard to come by at NTNU during final bachelor assignments.

Having access to Oracles offices was therefore much appreciated. Access to information and assistance was plentifully at the office, and discussing implementation and requirements was never an issue.

Frequent meetings with Oracle has also been taken advantage of for the duration of this project. These meetings have mostly taken place online with a few being physical. This has helped to push the project in Oracle's desired direction as well as straightened out uncertainties along the way. *Meeting summaries can be found in the external appendix*.

Finally, Oracle initiated an internal review process of the product. Although not yet finished, it has helped point out certain shortcomings. Most noticeably it caused the routing engine to no longer return references to provided edges, but instead return direct copies. This did demand more system memory, as the references were 64 bit pointers and each copy took 192 bits, but it also improved performance with less dereferencing. Furthermore, copies were also a simpler memory model, and thus less error prone.

6.4 Performance

Nuances of the reported statistics in section 5.3 (Performance) will be explored in this chapter.

6.4.1 Routing engine

Performance statistics from exclusively testing the routing engine were just as expected. Using a euclidean heuristic took on average 74.8% as long as not using any heuristic. This coalesces with theory found in subsection 3.1.2 (A-star & heuristics).

6.4.2 Entire system

As seen in subsection 5.3.2, some of the results were as expected while other results were a bit more surprising. This section aids in better understanding these results, and deviations.

The result from the test without heuristics compared to the previous implementation is as expected. None of these tests take advantage of heuristics, they accomplish the same goal, and are therefore easy to compare. As seen, the implementation devised in this paper is roughly three and a half times faster than the previous solution. This is not only a huge improvement in performance, but it also offers a lot more functionality. Not only is a more complex output given along with the possibility to make use of the GROUP_BY statement, but it also offers much needed simplicity for the end user.

Regarding the most discombobulating result, videlicet, the difference in performance with heuristics compared to without. In most cases, using heuristics would outperform not using them; that was not the case here. However, when testing exclusively on the routing engine with the same graph, heuristics did in fact improve computation time. Logically, this implies what was long suspected, that aggregation was and is much slower than actually computing the shortest path. At the same time, the extra work involved in gathering nodes during aggregation was not expected to make the whole process three times slower. For this reason it seemed reasonable to assume that this was a case of poor implementation, rather than an inherent limitation to the aggregation system. To discern what caused this blip, all error checking, input validation, and even heuristics involved were removed. The only thing now that was different from not using a heuristic was having to extract nodes from user input and store them in a hash map. Surprisingly, this did not improve run time one bit. However, after removing node geometry type deduction, it ran about five seconds faster, still about twice as

slow as when not using any heuristic. Ultimately, it turned out that hash table persistence of nodes and type deduction of node geometries each caused way more computation complexity than any heuristic could make up for. A possible solution would be to store destination node geometry data in the same hash table as edges. However, this would cause more data redundancy whilst not ensuring node data consistency. Not to mention how node geometry type deduction still makes the whole process prohibitively slow.

7 Social impact

What good is any project or paper if does not contribute to society? Social impact, good and bad, will be the subject of this chapter.

7.1 Streamlined database searching

Making graph database searching easier, at least when looking for shortest paths, is this project's main contribution. How is this done and what does it mean for society?

Graph database searching is made easier first of all through using an established language such as SQL. A simple SQL function like ST_SHORTEST_DIR_PATH is both user friendly and instantly integratable into the also easy to use SQL language. This makes it very approachable without too much experience in computer science or engineering.

This can help streamline data analysis. SQL is a salient tool in any data analyst's belt. With extended access to path finding algorithms their jobs become less demanding. Although this solution is, first and foremost, developed to be a geospatial routing engine, there is no reason why it can not be used on more general graphs. It could help find that path of least resistance in an electric circuit, the path of least congestion in an information network, or the path of fewest nodes in a network of proxies, to name a few. On the other hand, if more systems rely on this system because it is easier it is likely to result in less efficient applications overall, which means more wasted electricity. A more tailor made system, with better graph caching, would therefore often be more eco-conscious. This is especially a concern for immutable graphs.

7.2 Open source

MySQL is an open source relational database management system. Open source means that the computer software is released under a special license where the owner grants users the right to use, study and change its source code to anyone and for any purpose. Other types of software can only be altered by the owner of the license. Therefore, if someone other than that wishes to make any changes, they would need permission. By adding the function, ST_SHORTEST_DIR_PATH, to the MySQL repository, anyone can make use of it.

There are several benefits to open source, making it preferable in many situations. First, it provides a larger community of developers, lowering the chance of mistakes and errors. More developers also increases the chance that they work on software they like, which can give a better result and happier developers. Secondly, it gives more people access to useful knowledge which would otherwise be reserved to a small minority [Open Source, 2022].

8 Conclusion

Conclusively, this chapter will discern to what degree the introductory goal has been achieved, as well as provide a summation of remaining/future work.

8.1 Final product

As elaborated in section 7.1 (Streamlined database searching) and section 6 (Discussion), the implemented system offers users the benefit of easy to use functional and intuitive SQL syntax. It also contains responsive error handling, as seen in subsection 5.2.3 (Exception handling). Ultimately, this makes the product approachable and graphs easier to handle in MySQL.

Compared to previous methods, the solution found here is, as discussed in section 6.4 (Performance), over than three times faster, on the tested graphs. However, this is only the case when not using heuristics. Heuristics do improve computation time in the routing engine, but due to more complex aggregation, using them ultimately causes worse performance in the final system. This is still faster than the previous solution, but surprisingly underperforming. None the less, this is faster than before, and especially so when not using heuristics. It has therefore improved the performance of graph searching in MySQL.

8.2 Future agenda

Given time constraints, some features and improvements were naturally cut out. For later development it is believed that some of the suggestions mentioned in this segment could help lift the project further.

As it is vital for any important system to be robust and reliable, developing more tests is advisable. Prudently constructing grander data sets with a larger variety of geometric anomalies is therefore a logical next step.

Giving the user more control over applied heuristics would also be purposeful for further development. This was discussed in detail in subsection 6.2.1 (Compiler syntax). In short, support for user defined heuristic coefficients and/or user defined heuristic functions could be added.

Inexpensive data pruning could also be integrated to speed up the whole system. This was proposed in subsection 6.2.2 (Aggregation). In short, it revolves around ignoring edges that are for some reason deemed irrelevant. Naturally, this should be more considerate to reducing aggregation time, rather than simplifying path finding, as the former is a larger performance liability.

Finally, simplifying and improving heuristics is probably the most pressing issue. Multiple possible improvements have been proposed, but one sticks out as more promising. Graph cashing, to avoid constant graph reconstruction, would eliminate the most expensive part of the method. However, this is expected to be a difficult task, and it would still require reconstructing the graph each time it changes drastically.

Bibliography

[Anneliese Mayrhauser and Lang, 1998] Anneliese Mayrhauser, A. M. V. and Lang, S. (1998). Program Comprehension And Enhancement Of Software. https://www.researchgate.net/publication/2332938_Program_Comprehension_And_Enhancement_Of_Software. Last accessed 27 April 2022.

[Beck et al., 2001] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). Manifesto for Agile Software Development. https://agilemanifesto.org/. Last accessed 14 May 2022.

[Corman, 2020] Corman, M. (2020). Why Should the Length of Your Hash Table Be a Prime Number? https://medium.com/swlh/why-should-the-length-of-your-hash-table-be-a-prime-number-760ec65a75d1. Last accessed 14 May 2022.

[Dijkstra, 1959] Dijkstra, E. W. (1959). A Note on Two Problems in Connexion with Graphs. https://ir.cwi.nl/pub/9256/9256D.pdf. Last accessed 25 April 2022.

[Duchon et al., 2014] Duchon, F., Babinec, A., Kajan, M., Beno, P., Florek, M., Fico, T., and Jurisica, L. (2014). Path planning with modified A star algorithm for a mobile robot. https://core.ac.uk/download/pdf/82238411.pdf. Last accessed 22 April 2022.

[Free Software Foundation, 2022] Free Software Foundation (2022). GDB: The GNU Project Debugger. https://www.sourceware.org/gdb/. Last accessed 27 April 2022.

[Google, 2022] Google (2022). Google C++ style guide. https://google.github.io/styleguide/cppguide.html. Last accessed 26 April 2022.

[Hart et al., 1968] Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. https://ieeexplore.ieee.org/document/4082128. Last accessed 24 April 2022.

[Helge Hafting, 2014] Helge Hafting, M. L. (2014). *Algoritmer og Datastrukturer*, volume 1. Gyldendal Akademisk. Page 283.

[Kamaruzzaman, 2021] Kamaruzzaman, M. (2021). Top 10 Databases to Use in 2021. https://towardsdatascience.com/top-10-databases-to-use-in-2021-d7e6a85402ba. Last accessed 29 April 2022.

[Loshin, 2022] Loshin, P. (2022). Structured Query Language (SQL). https://www.techtarget.com/searchdatamanagement/definition/SQL. Last accessed 13 May 2022.

[Open Source, 2022] Open Source (2022). Open Source. https://opensource.com/resources/what-open-source. Last accessed 9 May 2022.

[Oracle, 2016] Oracle (2016). Creating and Executing Unit Tests. https://dev.mysql.com/doc/dev/mysql-server/latest/PAGE_UNIT_TESTS.html. Last accessed 26 April 2022.

[Oracle, 2018] Oracle (2018). MySQL Test Run. https://dev.mysql.com/doc/dev/mysql-server/latest/PAGE_MYSQL_TEST_RUN_PL.html. Last accessed 11 May 2022.

[Oracle, 2022a] Oracle (2022a). The MySQL Test Framework. https://dev.mysql.com/doc/dev/mysql-server/latest/PAGE_MYSQL_TEST_RUN.html. Last accessed 26 April 2022.

[Oracle, 2022b] Oracle (2022b). Update for Russian and Belarusian companies, subsidiaries, and partners. https://www.oracle.com/corporate/conflict-in-ukraine/russia/. Last accessed 10 May 2022.

[Richards, 2019] Richards, H. (2019). EDSGER WYBE DIJKSTRA. https://amturing.acm.org/award_winners/dijkstra_1053701.cfm. Last accessed 25 April 2022.

[Safris, 2022] Safris, S. (2022). A Deep Look at JSON vs. XML, Part 1: The History of Each Standard. https://www.toptal.com/web/json-vs-xml-part-1. Last accessed 10 May 2022.

[Stack Overflow, 2019] Stack Overflow (2019). MySQL Shortest Path between two nodes. https://stackoverflow.com/questions/59506079/mysql-shortest-path-between-two-nodes. Last accessed 13 May 2022.

[Xia, 2018] Xia, X. (2018). Measuring Program Comprehension: A Large-Scale Field Study with Professionals. https://ieeexplore.ieee.org/abstract/document/7997917. Last accessed 27 April 2022.

[Zeng and Curch, 2008] Zeng, W. and Curch, R. (2008). Finding shortest paths on real road networks: the case for A*. https://www.tandfonline.com/doi/abs/10.1080/13658810801949850. Last accessed 25 April 2022.

A Appendix

Requirements specification

Description:

This wl implements st_shortest_dir_path(id, from_id, to_id, cost, to_point, start_node, end_node), which will find the shortest path between the nodes start_node and end_node usinge the A star variation of dijkstra's algorithm. The function outputs the answer in JSON format containing the total path cost, number of visited nodes, and an array of edges representing the path. GROUP BY type can be used to group the answer in different categories e.g bike or car.

It takes 7 parameters,
id INT NOT NULL
- *The ID of and edge*

from_id INT NOT NULL
- *The ID of the starting node for this edge*

to_id INT NOT NULL
- *The ID of the ending node for this edge*

cost DOUBLE NOT NULL
- *The cost of this edge*

to_point POINT
- *The geometry data of node with id = to_id*

start_node INT NOT NULL
- *The start node from which you want to find the shortest path*

end_noe INT NOT NULL
- *The end node to which you want to find the shortest path*

It returns a JSON object with values:
cost DOUBLE
path ARRAY
- *The path found. Consists of edge objects. Each edge object contains:*
  id INT
    - *The id of the edge*
  cost DOUBLE
    - *The cost of the edge*

visited_nodes INT
- *The number of nodes visited by the routing engine*

Example:

```
CREATE TABLE edges(
id INT PRIMARY KEY,
from_id INT,
to_id INT,
cost DOUBLE,
type VARCHAR(50)
);
INSERT INTO edges VALUES
(10, 0, 1, 2.0, "car"),
(11, 1, 2, 3.0, "car"),
(12, 0, 2, 8.0, "car"),
(20, 0, 1, 20.0, "bike"),
(21, 1, 2, 25.0, "bike"),
(22, 0, 2, 40.0, "bike");

SELECT ST_SHORTEST_DIR_PATH(id, from_id, to_id, cost, NULL, 0, 2) FROM edges
GROUP BY type;
# Expected result:
{"cost": 40.0, "path": [{"id": 22, "cost": 40.0}]}
{"cost": 5.0, "path": [{"id": 10, "cost": 2.0}, {"id": 11, "cost": 3.0}]}
```

Requirements:

F-1 - The function must raise an ER_WRONG_ARGUMENTS exception if argument id, from_id, to_id, cost , to_point, start_node or end_node is NULL. In other words, only to_point can be NULL.

F-2 - If any no path exists between provided start_node and end_node exception ER_NO_PATH_FOUND must be raised.

F-3 - The function must support GROUP BY.

F-4 - st_shortest_dir_path shall give the result in JSON-format. This result contains total path cost (double), the path (array of edges*), and number of nodes visited by the routing engine (int). A potential stylized result could look like the JSON below.

```
{
   "cost": 42.7,
   "path": [
      {"id": 0, "cost": 20.0 },
      {"id": 5, "cost": 22.7 }
   ],
   "visited_nodes": 5
}
```

Visited_nodes must not be confused with number of nodes along path,which always equals the size of the path array plus one.
**an edge must consist of an id (int) and its traversal cost (double).*

F-5.1 - if all to_point cells are NULL the function will calculate the shortest path without any heuristic.

F-5.2 - if all to_point cells are valid points the function will calculate the shortest path using the distance between these points and the destination as a heuristic.

F-5-3 - if geometry data is given for some edges, but not all, an ER_INCONSISTENT_GEOMETRY_NULLNESS will be raised.

F-6 - if cost is less than or equal to 0 for any row, exception ER_NEGATIVE_OR_ZERO_EDGE_COST must be raised.

F-7 - if multiple rows have the same edge id, exception ER_DUPLICATE_EDGE_ID must be raised.

F-8 - If to_point is not a point one or more rows, exception ER_GIS_WRONG_GEOM_TYPE must be raised.

F-9 - if from_id and to_id are equal for any edge, exception ER_EDGE_LOOP must be raised.

F-10 - if different to_point cells have different SRIDs exception ER_GIS_DIFFERENT_SRIDS_AGGREGATION must be raised.

F-11 - if id from_id, to_id, start_node or end_node are null or not intergers in any row exception ER_WRONG_ARGUMENTS must be raised.

F-12 - if cost is null or not a real number in any row expcetion ER_WRONG_ARGUMENTS must be raised.

F-13 - if start_node or end_node are not identical for all rows, exception ER_WRONG_ARGUMENTS must be raised.

High level architecture:

I-1:

No new files

I-2:

No new commands

I-3:

No new tools