Location Searching and Road Path Planning for Autonomous Vehicles With Improved Intersection Considerations

by

Yushi Guan

Supervisor: Professor Angela Schoellig

April 2018

Abstract

Autonomous vehicles are becoming more popular and advanced these days. In order to reach Level 3 (Conditional Automation) of autonomous driving, a vehicle must plan its path and monitor road conditions. High level road path planning is an essential component for any autonomous vehicle to plan its route from one location to another. In this report, we proposed a new location searching algorithm based on preprocessed on-disk hash table, which we found to have better performance compared to traditional SQL query methods. In addition, we demonstrated the difficulties to assign street intersections with any cost when representing road networks with traditional graphs. We proposed a new representation, and an associated Road Path Planning with Improved Intersections Considerations Algorithm that take into account the cost incurred when crossing through and turning at intersections. Compared to other methods that attempt to address vehicle turning cost, our representation does not require preprocessing of the map database. Using this new representation, the turning cost can be dynamically allocated, and a turn-by-turn instruction that is easy to execute for an autonomous vehicle can be generated. Evaluated using Greater Toronto Area road network data, the algorithm reduces the number of turns in the proposed path by half, without significantly increasing the total distance of the path.

Acknowledgement

This thesis project is conducted as part of the aUToronto Autodrive Project. I would like to thank Professor Angela Schoellig for supervising my thesis. I would also like to thank my teammates for their support through this challenging project.

Contents

List of Figures

List of Tables

1. Introduction

The aUToronto Autodrive Team is participating in the AutoDrive Challenge organized by SAE International [1]. The goal is to build a Level 3 (Conditional Automation) autonomous vehicle over the course of three years [2]. As defined by SAE International, Level 3 Conditional Automation requires the automated driving system to take care of all aspects of the driving task in a dynamic environment under normal operating conditions [2]. High-level road path planning is an essential component to enable autonomous vehicles to operate in real road networks by itself.

To motivate the design and implementation of a road path planning framework, SAE International organized a Mapping Challenge as part of the AutoDrive Challenge that spans over three years [3]. In year one, participating teams should demonstrate location searching and displaying ability in North America map database. Types of location include Point of Interest (POI) Name and ID, Street Addresses, GPS Coordinates and Street Intersections. In year two, teams should showcase road path planning algorithm that provides a path from one location on the map to another. In year three, teams will enable the communication between the mapping framework and the vehicle, providing turn-by-turn instructions for the vehicle.

For efficient location search in a map database, a lot of algorithms and data structures has been proposed. Popular methods and techniques include: SQL database, where storage and query of information have been highly standardized and work across many systems [4]; Distributed Systems and Cloud Computing, where location search queries will be simultaneously handled by different systems to speed up the process [5]. However, for the SAE AutoDrive Challenge, the vehicle is not connected to the internet during its operation [3]. The vehicle needs to consult map information stored on the computers in the vehicle, and quickly give location information without the use of Cloud Computing or powerful servers located somewhere else. SQL databases, using its underlying B+ Tree structure, do not scale well when there is a huge number of data [4]. To speed up the location search process, we propose a hash table data

structure that enables information search in O(1) [6]. Details of preprocessing and performance analysis will be presented in this report.

We observed that road network path planning may not generate an optimal path for autonomous vehicles when traditional graph representations are used. Traditional graph representations that involve vertices with unique identifiers and weighted edges have a lot of applications in computer science and robotics research. For example, graphs can be used to represent networks of communication and flow of computation in computer science [7]. Robotics navigation performance can also be enhanced with the adaptation of topological graph and path planning [8]. However, these traditional graph representations that do not associate weights with vertices have a lot of problems generalizing to road networks. A lot of applications naively treat street intersections as graph vertices and streets as weighted edges [9]. However, autonomous vehicles incur a lot of cost at street intersections. These costs include time spent waiting for traffic signal change, potential hazards and risks associated with crossing and turning at intersections. In addition, traditional graph algorithms usually generate solutions that are purely based on an ordering of the vertices. For example, topological sort generates an ordering of the vertices such that for every direct edge from vertex A to vertex B, A comes before B in the ordering [10]. Dijkstra's and A* algorithm produce the shortest path based on the ordering of the vertices [11] [12]. When combining these algorithms with road graph that treats street intersections as vertices, the results got are not executable for autonomous vehicles. The vehicles will be informed with the order of intersections, but it does not have complete information to navigate from one intersection to another. In this report, we propose a different representation that provides turn-by-turn instruction for vehicles to operate on a real road network, and dynamically penalize the use of intersections and turns to ensure safer operation of the vehicle.

2. Literature Review

2.1 Survey of Database Search Techniques

The North America map data provided by SAE AutoDrive Challenge contains about 10^9 feature points [3]. In year one, the team is required to demonstrate software framework that can search for a geographical location using Street Intersection Names, Street Address, GPS Coordinate, or Point of Interest ID and Name. The program should also be able to zoom in and zoom out on the map after the location has been found. Due to the large amount of data, constant or near constant search time is required to fulfill the requirements. In the following section, an overview is provided for two constant and near constant search complexity data structures: B+ Tree and Hash Table [4] [5].

2.1.1 B+ Tree:

B+ tree is heavily used by SQL databases to store data on disk persistently [4]. Each node in B+ tree may have a number of children, up to its branching factor b. The primary difference between a B+ tree and a B Tree is that each node in a B+ tree only contains a key instead of a key-value pair. Values in the B+ tree are only located at the bottom of the tree where the leaves of the tree are found. On each level, each node is also linked to its adjacent nodes, in addition to its parent and child. These particular properties make B+ tree highly efficient for information retrieval in a block-oriented storage context such as hard drives. Therefore, it has been widely utilized in file system and SQL database designs. A very large branching factor b can be utilized to reduce the height of a B+ Tree, reducing the time required for querying when there is a very large amount of data such as the North America map data. The time complexity of retrieving a record is $O(\log_b n)$, where b is the branching factor and n is the total number of data points. The time complexity of retrieving a range of records, which is required when querying for street intersection locations, is $O(\log_b n + k)$, where k is the average number of street segment objects in the database under the same street name [4]. For the Mapping

Challenge in particular, PostgreSQL has been a standardized SQL database that works seamlessly with the open source GIS mapping applications. The team can benefit from the ease of maintenance and relatively good performance by using a PostgreSQL application.

2.1.2 Hash Table:

Hash table is a data structure that maps keys to values [6]. A hash table involves a hash function, an array of buckets that stores the key-value pairs, and a collision resolution mechanism. A hash function is a function that quickly computes an integer hash value given a key. During storage, the bucket with the index of the integer hash value will be used to store the key-value pair. During a search, the bucket will be instead consulted to retrieve the value associated with the key. The objective of a hash function is to minimize the number of potential collisions, i.e. the potential of two different keys having the same hash value, this is not possible if the size of the array of buckets is small compared to the total number of key-value terms. Therefore, hash tables usually need to maintain a relatively low load factor (number of key-value pair to the size of bucket array ratio), through dynamical resizing: expand the size of the array of buckets in case of more key-value pairs being added. In the case of hash value collisions, colliding key-value pairs can be stored in a linked list with its head attached to one of the buckets in the array. This particular method is named chaining. Another mechanism is open addressing, which helps to reduce storage space but performs poorly at a high load factor. The biggest advantage of hash table is an constant O(1) lookup time when there are minimal key collisions. This is often achievable in practice through the choice of a reasonable hash function and optimal load factor. When searching for information among the entire North America database, this constant lookup time can be much faster than SQL database which has logarithmic lookup time. Although traditionally hash table is usually stored in computer memories, on-disk versions of the data structure exist such as the Python Shelve module [6]. If an on-disk hash table is used, the performance of a query is reduced by the ratio of memory speed and disk speed, but the time required for loading the hash table is essentially eliminated.

2.2 Survey of Ambiguous Search Algorithm

Modern search engines have been very successful at detecting spelling errors. For example, suggesting the word ‘spelling’ given an input ‘speling’. However, this success is backed by powerful servers and exhaustive learnings through tokens online [13]. In the following section, an overview of Google’s approach is provided. We will also take an overview of the idea behind a lightweight open source project spellmaster [14].

2.2.1 The Google Approach

Google has developed an ene-to-end system spellchecking system that does not require any manually annotated training data [13]. Google uses a large (> 1 billion) sample of web pages to build an error model and an n-gram language model. Using a small secondary news texts with artificially inserted misspellings, a classifier is trained to detect potential misspellings in the n-gram language model, creating a reference set of high-confidence correct language model.

2.2.2 The open source project spellmaster approach

The open source project spellmaster uses a reference dictionary that contains correctly spelled words, and assume no correctly spelled words exist beyond the words included in the dictionary [14]. When a word suggestion is given, a set of its variants are produced based on common spelling errors, which includes: insertion of an extra letter, deletion of a letter, substitution of a letter and swap of letters.

2.3 Survey of Path Planning Techniques

In the following section, an overview of typical graph representations, and graph representations for road networks will be provided. In addition, a survey of some existing path planning algorithms will also be given.

2.3.1 Graph Representations:

Definition **(weighted) undirected graph**. An (weighted) undirected graph is an ordered pair $G = (V, E)$ comprising a set of Vertices V and a set of Edges E [15]. A graph contains N vertices, and M edges. Each edge $E_{m,m\in[0,M)}$ is associated with an unordered two element subset of Vertices (v_1, v_2). In a weighted graph, each edge $e_{m,m\in[0,M)}$ is also associated with a weight $W(e_{m,m\in[0,M)})$, which typically represents the cost of getting to one vertex in the subset from another vertex in the subset.

Definition **(weighted) directed graph**. A (weighted) directed graph is an ordered pair $G = (V, E)$. Each edge $e_{m,m\in[0,M)}$ is associated with an ordered two element subset of Vertices (v_1, v_2) [16]. In a weighted directed graph, each edge $e_{m,m\in[0,M)}$ is also associated with a weight $W(e_{m,m\in[0,M)})$, which typically represents the cost of getting to the second vertex in the subset from the first vertex in the subset.

Definition **(weighted) symmetric directed graph**: A (weighted) symmetric directed graph is an ordered pair $G = (V, E)$, a particular case of (weighted) directed graph [17]. Each edge is associated with an ordered two element subset of Vertices (v_1, v_2). For each edge $e_{m,m\in[0,M)}$, there must exist a complementary edge $\bar{e}_{\bar{m},\bar{m}\in[0,M)}$ that is associated with a corresponding two element subset of Vertices (v_2, v_1). In a weighted symmetric directed graph, each edge is associated with a weight $W(e_{m,m\in[0,M)})$, where $W(e_{m,m\in[0,M)}) = W(e_{\bar{m},\bar{m}\in[0,M)})$.

Definition **weighted simple undirected road network graph**: A weighted simple undirected road network graph is an ordered pair $G = (V, E)$ comprising a set of Vertices V and a set of Edges E. Each vertex $v_{n,n\in[0,N)}$ represents a street intersection, or the end of a road. Each edge $e_{m,m\in[0,M)}$ represents a street segment between two vertices in a road network. Each edge is associated with a weight $W(e_{m,m\in[0,M)})$, which typically represents the cost of getting to one vertex from another.

Definition **weighted symmetric directed graph**: A weighted symmetric directed graph is an ordered pair $G = (V, E)$ comprising a set of Vertices V and a set of Edges E. Each vertex $v_{n,n\in[0,N)}$ represents a street intersection, or the end of a road. Each edge $e_{m,m\in[0,M)}$ represents a street segment from one vertex to another (v_1, v_2), and may have a complementary edge $\bar{e}_{\bar{m},\bar{m}\in[0,M)}$ which represents the complementary street segment between the vertex pair (v_2, v_1).

2.3.2 Path Planning Algorithms:

A lot of path planning algorithms and their modified versions exist up-to-date. These algorithms typically vary in their time and space complexity during query time and preprocessing, as well as the complexity of implementation. In this section, an overview of the algorithms is provided.

Here, we present the typical definition of a path based on the ordering of the vertices:

Definition **path**: A path in a graph $G = (V, E)$ is a sequences of vertices $P = < v_1, v_2, \dots, v_k >$, where $v_i \in V$, such that v_i is adjacent to v_{i+1} for $1 \leq i < k \leq N$. The path P is an ordered array of size k [18]. In a undirected graph, v_i is adjacent to v_{i+1} if and only if there exists an edge $e_{m,m\in[0,M)}$ that is associated with the unordered 2-element set (v_i, v_{i+1}). In a directed graph, v_i is adjacent to v_{i+1} if and only if there exists an edge $e_{m,m\in[0,M)}$ that is associated with the ordered 2-element set (v_i, v_{i+1}).

Definition of shortest path based on an ordering of the vertices is:

Definition **shortest path**: A shortest path from a source vertex $\boldsymbol{S}$ to a destination vertex $\boldsymbol{D}$ in a graph $G = (V, E)$ is a sequences of vertices $P = < S, v_2, v_3, \ldots\ v_{k-1}, D >$, where $v_i \in V$, such that v_i is adjacent to v_{i+1} for $2 \leq i < k - 1 \leq N$, S is adjacent to v_2, and v_{k-1} is adjacent to D [18]. Adjacent has the same definition as in definition 6. Given a weighted function $W: (V, V) \rightarrow R$. A shortest path P is one that minimizes the cost function $\sum_{i=1}^{k} W(v_i, v_{i+1})$, where $S = v_1$, $D = v_k$ and v_i is adjacent to v_{i+1}.

2.3.2.1 Dijkstra's Algorithm:

Dijkstra's algorithm can generate a shortest path tree from a single source vertex S [11]. In the shortest path tree, the shortest path from S to any other vertex $v_{n,n\in[0,N)}$ is represented by the branch of the tree for which $v_{n,n\in[0,N)}$ is lying on.

To generate the shortest path tree, the algorithm performs the following steps:

Initialization [11]: mark the distance of all vertices to be infinity, and their status as unvisited (other possible states are visited and fully explored), set the source S as the current vertex.

Iterative Steps [11]:

1. Explore each adjacent vertex of the current vertex and mark each adjacent vertex as visited, unless the adjacent vertex has been marked as fully explored. Calculate the tentative distance of each adjacent vertex from source S via the current vertex, i.e. if the distance from source vertex S to the current vertex is X, and the distance from the current vertex to the adjacent vertex is Y, the adjacent vertex's tentative distance is X + Y. If the adjacent vertex has been previously marked with a distance greater than X + Y, then change its distance to X + Y, and set the current vertex as the parent of the adjacent vertex.

2. Mark the current vertex as fully explored and then select a visited vertex with the smallest tentative distance. Mark the selected vertex as the current vertex and repeat step 1. The shortest path tree is generated once there is no visited vertex available.

Using min-priority queues, the worst case run-time of Dijkstra's Algorithm is $O(M + N \log N)$, where M is the number of edges, and N is the number of vertices.
In addition, Dijkstra's Algorithm also allows early termination if there is a specific destination vertex $D \in V$. The algorithm will terminate once the destination D has been marked as fully explored. Dijkstra's Algorithm guarantees optimality (the discovery of the shortest path if there is a path from source S to destination D exists).

2.3.2.2 Bidirectional Dijkstra's Algorithm:

A Bidirectional Dijkstra's Algorithm explores the shortest path by starting two Dijkstra's searches in parallel: one from the source vertex S and another one from the destination vertex D [19]. The algorithm terminates when the two searches found a mutual fully explored vertex M. The full path can be reconstructed by combining the path from S to the M and D to M.

This parallel search technique greatly reduces run-time because the number of edges and vertices that are being searched will grow exponentially as the frontier of the search expands in Dijkstra's Algorithm [19]. When early termination is used in Dijkstra's Algorithm and Bidirectional Dijkstra's Algorithm, the worst-case time complexity $O(M + N \log N)$ is not applicable if the distance from S to D is much smaller than the largest distance between two arbitrary vertices in the graph. The average runtime can be represented by $O(b^d)$ where b is the average branching factor of the graph, and d is the number of levels (the length of the path) that need to be explored in the graph. In a road network that is made up of 4-way intersections (crossroads), the average branching factor m will be 4, and d will represent the number of intersections between S and D. With the adaptation of the Bidirectional Dijkstra's Algorithm, the run-time will be reduced to $O(2b^{d/2})$. The multiplier 2 comes from the 2 instances of parallel search, where the divisor 2 in the exponent of m is the result of a reduced level of

searches when the searches are started from both source and destination. In a road network mainly made up of 4-way intersections, a source and destination that is about 10 intersections apart may see a run-time reduction on the order of 105 when the bidirectional search is used [19].

2.3.2.3 A* Algorithm:

A* algorithm works similarly as the Dijkstra's Algorithm, in which the graph is being explored starting from the source vertex S. However, in A* search algorithm, the vertex with the smallest total estimated cost to reach the goal is selected to be explored first [12]. This gives A* algorithm a greedy nature: vertices that appear to be on the shortest path to the destination will be explored first. A* Algorithm is usually faster than Dijkstra's Algorithm in practical situations, but its worst-case time complexity remains the same.

The estimated total cost is the sum of the current tentative distance plus a heuristic function $H: (V,\ V) \rightarrow R$. $H(v_{current},\ D)$ estimates the distance between the current vertex and the destination. A* algorithm will guarantee the optimality of the solution if the heuristic $H(v_{current},\ D)$ selected is admissible, meaning the heuristic will never overestimate the actual cost of the path. In road network graph analysis, straight line distance between the current vertex and destination is a handy admissible heuristic. The distance can be easily calculated using geographical locations (latitude and longitude) of the vertices. A bidirectional A* Algorithm also exists, working similarly to the Bidirectional Dijkstra's Algorithm.

2.3.2.4 Contract Hierarchy Algorithm

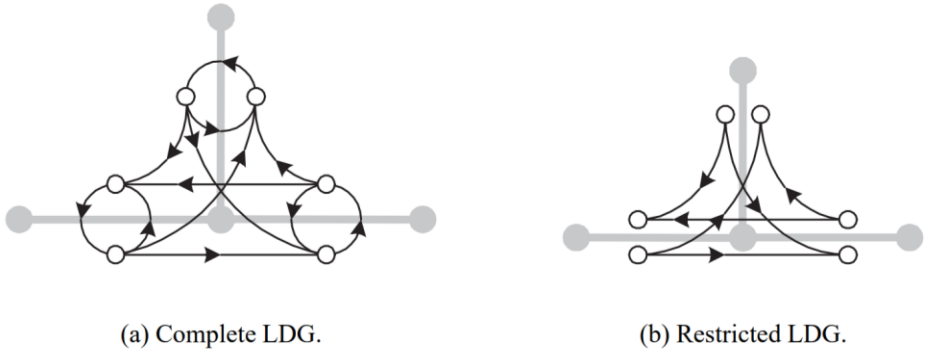
The Contract Hierarchy Algorithm enables query of a path on the order of $O(H \log_H N)$, where H is the number of hierarchies it has constructed after preprocessing of the map [9]. The preprocessing is also very fast as it is strictly linear: (N+M). It produces a guaranteed optimal path by avoiding arbitrary division of the map. With a query time approximately on the order of $O(H \log_H N)$, it can query a USA map with 10^7 nodes using about 10^3 $O(1)$ operations. Briefly, the Contract Hierarchy algorithm has two parts:

1. **Hierarchy Construction**: A higher layer highway abstraction Ei contains an edge $(v_i, v_{i+1}) \in E_{i-1}$ if it belongs to some canonical shortest path $P = < s, \dots, v_i, v_{i+1}, \dots t >$ for any $s,t \in Ei-1$ and $v_i \notin N_H(s)$, $v_{i+1} \notin N_H(t)$ where $N_H(s)$ is the set of H nearest neighbors of s.

2**. Query**: Use bi-directional Dijkstra's search on each abstraction layer and combine the result

Intuitively, the algorithm prohibits slow edges (local roads) from being added to a higher abstraction layer because it cannot be on the shortest path for two nodes (s and t) sufficiently far away from it.

2.3.2.5 Linear Dual Graph for Turn Cost Representation



(a) Complete LDG. (b) Restricted LDG.

Figure 1. Complete Linear Dual Graph and Restricted Linear Dual Graph Representation

Modeling Costs of Turns in Route Planning is an algorithm proposed by researchers at Technical University Vienna [20]. The idea of this algorithm is for each intersection in the road network, create two additional vertices for each edge (street) connected to the intersection. Therefore, by properly connecting the vertices with directed edges and without creating loops, a Restricted Linear Dual Graph of the original graph can be created. Turn cost can be associated with these newly created edges. Using Dijkstra's Algorithm on the Restricted LDG, a path that is optimized for reducing the number of turns can be created.

The main drawback of this algorithm is extensive preprocessing of the map data is required. In following sections, we will propose an algorithm penalizes turns on the fly without preprocessing required.

3. Methods and Implementation

3.1 Location Searching Software Framework Implementation

In order to complete the requirements for year one Mapping Challenge. I and my teammates have explored several methods in parallel. Our most successful implementations have been using SQL Database and on-disk Hash Table. The implementation details will be presented below.

3.1.1 SQL Database implementation

Location searching using SQL Database has been mostly implemented by my teammate Zane Huang. The software framework used to search for the locations is PostgreSQL 10.3 for Ubuntu 16.04. During preprocessing, the extension 'postgis' in PostgreSQL 10.3 is used to load map information into the database [21]. Standard SQL queries have been used to search for Street Addresses, Point of Interest ID and Name, as well as Street Intersections.

3.1.2 On-disk Hash Table Implementation

Among all of the year one search types, street intersection search has been the most difficult one. All other search types usually have a direct key-value pair. The following table summarizes how the geographical location is associated with other information [22].

Object ID	Point of Interest ID	Point of Interest Name	Street Number	Street Name	Geographical Location
4598	32495	ABC Restaurant	78	John St.	(-123.345, 45.56)
8945	13244	CDE Hospital	34	King RD	(-67.892, 56.34)

Table 1. Example Representation of Feature Point in North America map database

As shown in Table 1, the feature point data is nicely structured in the database. Information can be easily stored in a relational database such as SQL and later retrieved using SQL queries. Although storing the information in an on-disk Hash Table will be more efficient at search time, there is a lot more preprocessing required, and the software framework will be less systematic than an SQL implementation. Therefore, the on-disk Hash Table is used to tackle street intersection search, where significant theoretical improvement exists. In the rest of this section, detailed preprocessing and implementation steps are presented.

3.1.2.1 Street Object Data Structure in Map Database Provided by SAE International

In order to correctly identify street intersections, the structure of how street information is originally presented in the database needs to be studied [22].

Object ID	Street Name	Street Section ID	Street Coordinates
101	YONGE ST	5647	[(-89.987, 46.238), (-89.988, 46.239)]
105	YONGE ST	3985	[(-89.234, 45.345), (-89.234, 45.236), … ***(-89.235, 45.359)***]
…	…	…	…
215	YONGE ST	2394	[(-89.678, 47.543), (-89.578, 47.545)]
783	COLLEGE ST	5849	[***(-89.235, 45.359)***, (-89.230, 45.359)]

Table 2. Street Object Data Structure in North America map database

Deducing street intersection locations directly from database entries are non-trivial tasks. Unlike finding an entry related to a single feature point, street intersection search involves querying entries related to both streets and compare their Street Coordinates to find if the streets intersect with each other. In Table 2., the bolded street coordinates match for “YONGE ST” and “COLLEGE ST”, as a result, it can be deduced that “YONGE ST” and “COLLEGE ST” is intersecting at (-89.235, 45.359).

3.1.2.2 Design of Hash Table for O(1) Time Complexity Street Intersection Search

The final Hash Table that allows O(1) street intersection search has a two-level hash table design. In the level-1 hash table, the keys are street names, and the values are level-2 hash tables. In the level-2 hash table, the keys are the name of the streets. The values are the geographical coordinate that represents the intersecting location of the level-1 hash table key street and the level-2 hash table key street. A graphical representation of the final data structure is shown below:

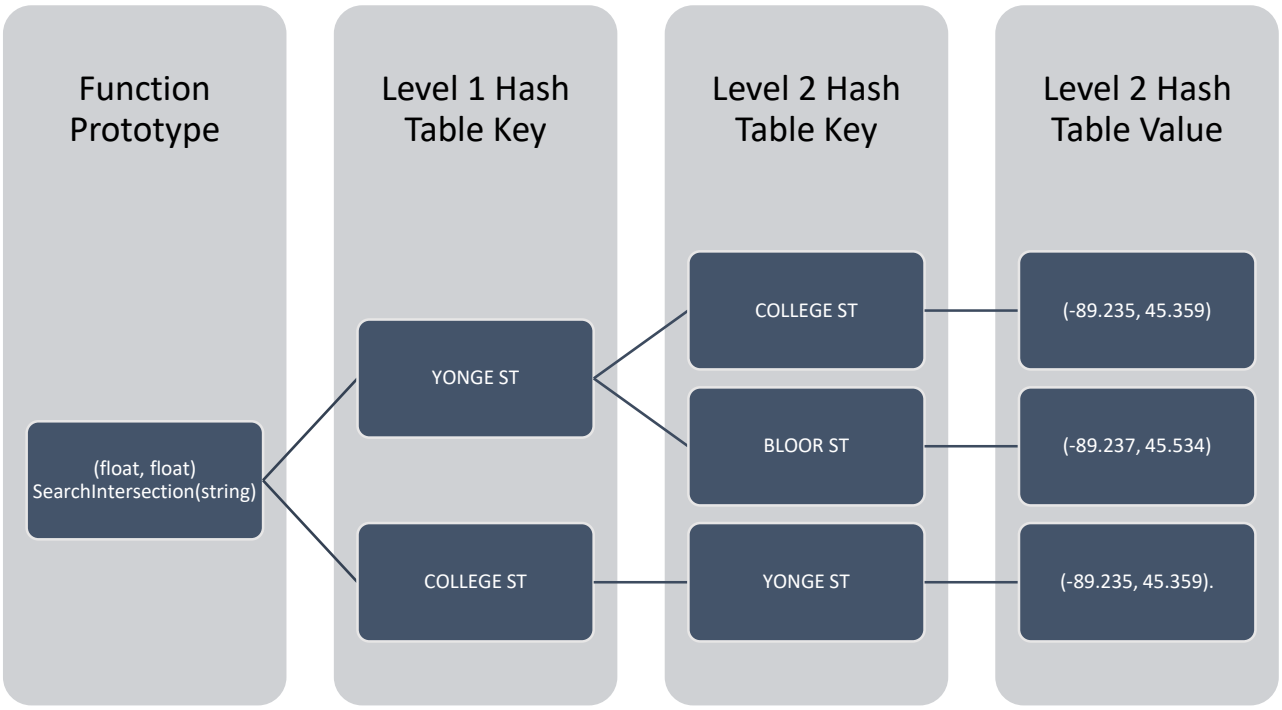


Figure 2. Structure of 2-Level Hash Table for O(1) search of Street Intersections

Given a search term, for example, “YONGE ST and COLLEGE ST”, “YONGE ST” will be used as the first key, to retrieve the level-2 hash table that contains all the streets that intersect with it. Using “COLLEGE ST” as the second key, the intersection of “YONGE ST” and “COLLEGE ST” will be retrieved, which is (-89.235, 45.359). Both of the searches have O(1) time complexity, therefore the entire search operation has O(1) time complexity [6].

3.1.2.3 Preprocessing of Data for Creation of Final Hash Table, Data Structure Perspective

Several steps are required to efficiently preprocess the North America map database to create the final hash table structure described in section 3.1.2.2.

Step 1: Create a temporary mapping between street intersection geographic coordinates and names of the streets intersecting at the location

This mapping is essentially a temporary hash table where the keys are the geographic coordinates, and the values are the name of the streets intersecting at the location. All the street segment entries in the map database are iterated through. The start and end coordinates of the street segment are extracted. Both the start and end coordinates will be added to the hash table as keys if non-exist. The corresponding values are array-like structures that contain the name of the streets at this location.

This preprocessing step is relatively computationally efficient. Each entry in the original map data only needs to be accessed once, and only the start and end street coordinates need to be retrieved to be used as the keys of the temporary hash table. Accessing the key location and adding the new street name to the list are also O(1) operations. Therefore, the overall time complexity of this step is O(n) [6].

Step 2: Merge the temporary mapping created for each map segment

The North American street information does not come from a single large file. There are 177 files representing different types of Highway and Street information in 44 regions (some regions may not have certain types of highway or street) [22]. In step 1, each of the files has been processed separately. Merging of the files is necessary because some street intersections may be missed if the files are not combined with each other. For example, geographical coordinate GC1 may exist in both temporary hash table A and B (from step 1).

	Key	Value
Hash Table A	GC1	[H ST, X ST]
Hash Table B	GC1	[K RD, X ST]
Merged Hash Table	GC1	[H ST, X ST, K RD]

Table 3. Data Entries in Individual Hash Tables and Merged Hash Table

Without merging hash table A and B together, the fact that at GC1, H ST and K RD are intersecting cannot be directly retrieved. The situation that one street intersection exists in two files occurs when: the street intersection is an intersection between different street types, or the intersection is at the border between two regions.

During the preprocessing, it is observed that only 0.71% of street intersections exist in more than one file. The street name list is usually less than 4 because 5- or 6- way intersections hardly exists. Therefore, the actual merging of the street name list does not have a big impact on the overall time complexity. During the creation of the merged mapping, each key in the temporary mapping will be checked for existence in the merged mapping through an O(1) hash table search operation. If the key exists, the corresponding street name list will be updated, otherwise, a new entry will be added to the merged mapping. Since searching and storing in a hash table are both O(1) operations, the overall time complexity of this step is O(n).

Step 3: Create the final hash table structure described in section 3.1.2.2.

Using the geographical coordinate to street name hash table generated in step 2, the final on-disk hash table that maps a street name to its intersecting streets can be created. Iterate through hash table from step 2, for example, if one of the entry has the following structure:

Key: GC1; Value: [H ST, X ST, K RD]

“H ST” is intersecting with both “X ST” and “K RD” at GC1. Each street is intersecting with both of the rest. As a result, we can create or add to the following entries in the final hash table:

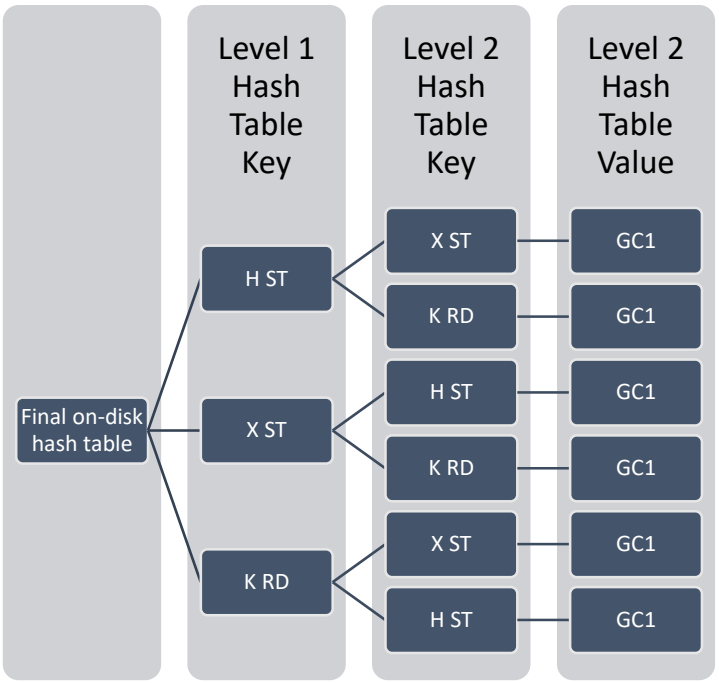


Figure 3. Structure of Final Hash Table in a Network with 1 Intersection and 3 Streets Intersecting

Creation of the final hash table has average time complexity $O(n \times b!)$, where n is the number of street segments and b is the average number of streets at each intersection. In actual road networks, most of the intersections are 4-way or 3-way intersections, higher way intersections hardly exist. It is reasonable to treat $O(n \times 4!) = O(24n) = O(n)$ as the average time complexity.

When a search term is given, for example “X ST and H ST”, the intersection’s geographical coordinate can be found first using “X ST” as the key for the Level 1 hash table and retrieve the corresponding Level 2 hash table, and then use “H ST” as the key to retrieve the geographical coordinate from the Level 2 hash table. The search term “H ST and X ST” will generate the same result.

Summary: from a time complexity perspective, each of the step has $O(n)$ time complexity, and each of the steps can be separated. Therefore, the overall time complexity for preprocessing is $O(n)$. Space wise, each step creates a constant amount of duplicate information compared to the input of the step. The input to each step can be deleted after successful creation of the output data. Therefore, the space complexity is also $O(n)$

The alternative naïve preprocessing procedure that is not efficient:

Alternatively, one may attempt to create the final hash table directly by iterating through the data entries in the original database. However, this approach will result in $O(n^2)$ time complexity because for every new street segment inserted, it needs to be compared to all existing entries to check if they meet at the same street intersection.

3.1.2.4 Preprocessing of Data for Creation of Final Hash Table, Software Perspective

In the following section, the software frameworks used to create and store the hash tables will be reviewed. Some of the speed-up techniques such as multiprocessing and memory conservation will be presented.

Software Frameworks:

QGIS 2.18

To access street information stored in the North America map database, QGIS 2.18 APIs that are written in Python 2.7.14 is used [23]. At the time of this project, QGIS 2.18 has been the most recent stable release version. Python 2.7.14 is the corresponding version Supported. In particular, APIs from the QGIS VectorLayer Class is heavily used in step 1 of the preprocessing to access street names and street coordinates of the street segments.

Python Dictionary for Python 2.7.14

The Python dictionary is the built-in hash table implementation in Python [24]. It has all the components of a hash table implementation: a hash function, bucket of arrays for storage and mechanism for collisions. The built-in Python hash function has been reported to be very efficient at avoiding key collisions. The built-in hash function works with other Python built-in immutable objects. In this project, keys of the hash tables have been selected to be standard Python built-in objects. Geographical coordinates are represented as a tuple of floats. Street names are represented as strings.

Python Pickle Module for Python 2.7.14

The development laptop used has 8GB of memory, which is not enough to handle all the information required for preprocessing at once. Python Pickle Module is a module for storing Python objects in files [25]. Python objects can be saved in pickle files in their binary form, allowing fast storage and data restoration. Theoretically, if the computer used has enough memory, all the preprocessing can be done without saving any information in files. However, due to memory constraints, simultaneously keeping both North America map database files that contain street information and the created temporary geographical coordinate to street name hash tables in memory is not possible. The result of preprocessing in step 1 is saved in Python Pickle files. By storing the processed information in files, information from unprocessed map database files are being loaded into the memory, while memory used by processed database files and already created hash tables can be freed. The processing can continue without using the hard drive as memory swap space.

Python Shelve Module for Python 2.7.14

Python Shelve is a module that saves Python dictionaries on-disk [26]. Values of the dictionary entries can also be accessed using the key without loading the entire dictionary into memory. Python shelve module has been used twice in the creation of the final on-disk hash table. It is used to

i. Temporarily store the results of step 2

   Unlike results of step 1, where the temporary hash table of corresponding to each file can be stored separately. Result of step 2, the merged hash table, needs to be stored as one piece of information. Storing it as one hash table is technically not possible given only 8GB of memory. Therefore, the merged result is saved directly on disk by calling the sync() function from the Shelve module, which basically flushes data in memory onto disk and allows data in memory to be freed when memory is no longer available.

ii. Permanently store the final on-disk hash table

   Information stored in the result of step 2 is accessed using the Shelve module read methods to be loaded into memory. After processing as described in step 3. The final hash table is stored on-disk as a Shelve object.

Python Multiprocessing Module for Python 2.7.14

The Python Multiprocessing Module is used to speed up step 1 of preprocessing [27]. Processing information in North America map files after loading them is a CPU-bound task. By default, each Python process runs on a single CPU core. Most modern CPUs, including the one used in this project, have multi-cores. The pool() function from the multiprocessing module is used to ensure there is the same number of processes running as the number of CPU-cores. As a result, all CPU-cores are in full usage and expensive context switches (which switches CPUs onto a different process).

3.2 Ambiguous Search Algorithm Implementation

In order to improve the success rate of street intersection search in North America map database, we implemented an ambiguous search algorithm that is optimized using the reference street names in the database.

3.2.1 Street Name Cleaning

First, the street names are preprocessed so that they become more standard without losing critical information. The cleaning includes:

i. Change street names to their upper case
ii. Remove periods '.' in the street names

3.2.2 Street Name Bank

During preprocessing of the hash table for street intersection search, a reference street name bank that includes all street names is created. Street name and street type are not separated. For example, "YONGE ST" is recorded as one street name, instead of being recorded separately as "YONGE" and "ST".

3.2.3 Standard Ambiguous Search Transformations

We define a set of standard transformations of an input search word adapted from the spellmaster open source project. Instead of using the full alphabet [a-zA-Z], the union of [A-Z] and [-] (dash "-", and space " ") is used as the set for our transformation letters (trans-letters). The list of standard transformations is below:

i. Insertion: insertion of a trans-letter between each of two letters in the input
ii. Deletion: deletion of a letter from the input
iii. Substitution: substitution of a letter in the input with a trans-letter
iv. Swap: swap of adjacent letters in the input

Variant Type	Examples
Insertion	AHWY ONE, H WY ONE
Deletion	HY ONE, HWYONE
Substitution	AWY ONE, A-Y ONE
Swap	WHY ONE, HW YONE
Abbreviation	HIGHWAY ONE
Number Swap	HWY 1

Table 4. Examples of Results Using Standard Transformations on Input "HWY ONE"

3.2.4 Street Name Special Ambiguous Search Transformations

Street names can often have different close formats that have similar meanings. For example, "YONGE ST" can be written as "YONGE STREET" without substantial changes to its meaning. In addition, numbers written in their numerical and English form are often both valid. Therefore, there are two additional variations for street names:

i. Abbreviation Substitute: Substitute an abbreviation with its full form, or vice versa
ii. Number Substitute: Substitute a numerical number with its English form, or vice versa

There are two reasons for which the street names are not cleaned to using only one form (such as only non-abbreviation and numerical form):

i. To preserve as much meaning as the original form if possible. For example, some highways may be named H-201, and they are rarely expressed in their English form H-TWO-HUNDRED-AND-ONE
ii. Some street names may contain standard street type tag in their name. For example, "View" is a standard street type that has an abbreviation "VW". If a street is named "Oceanview St.", the name becomes unrecognizable if it is renamed to "OceanVW St.".

Variant Type	Examples
Abbreviation Substitute	HIGHWAY ONE
Number Substitute	HWY 1

Table 5. Examples of Results Using Street Name Special Transformations on Input "HWY ONE"

3.2.5 Double Variation

In order to improve the chance of successfully finding the word in the database, a set of double variation versions of the original input will be created and looked up if none of the words in the single variation set has been found in the database. For example, single variations of "HWY ONE" include ["HIGHWAY ONE", "HWY 1", "AWY ONE", …], then double variations of "HWY ONE" include ["HIGHWAY OEN", "HIGHWAY 1", "WAY ONE", …]. Calculation and lookup of double variations are very computationally inefficient, but is a handy brutal force method to increase the chance of successfully finding the intersection.

3.3. Path Planning with Improved Intersection Considerations Implementation

In the following section, analysis on drawbacks of using traditional road network graph representations will be provided first. We will then propose an alternative representation, and a corresponding algorithm that calculates the lowest cost path with this representation. We will also present how can the algorithm be used to apply turn restrictions and improve the safety of the vehicle.

3.3.1 Disadvantages of Traditional Road Graph Network Representations

A traditional road network graph representation treats street intersections as vertices and streets as edges. The representation has difficulties taking into account the following differences and costs that occur for vehicles on a road network [20]:

(1) Different Shortest Path Based on Vehicle's Driving Direction

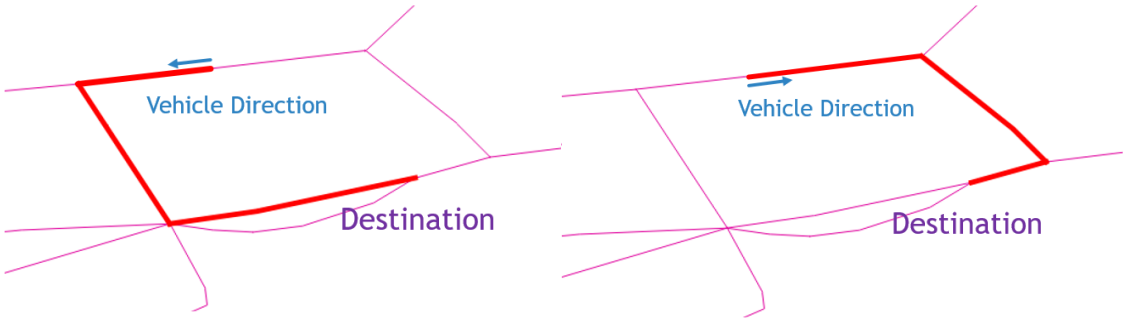


Figure 4. Different Actual Shortest Path on Road Network Based on Vehicle Direction

As shown in Figure 4, the vehicle may be at the same location, and is trying to reach the same destination. Depending on the vehicle's driving direction, it will have completely different shortest paths to the destination. However, the vehicle's driving direction cannot be modeled in a weighted undirected graph. As a result, the shortest path based on distance, which do not represent one that has the true lowest cost for the vehicle may be generated.

(2) Cost Associated with Waiting for Traffic Lights at Intersections

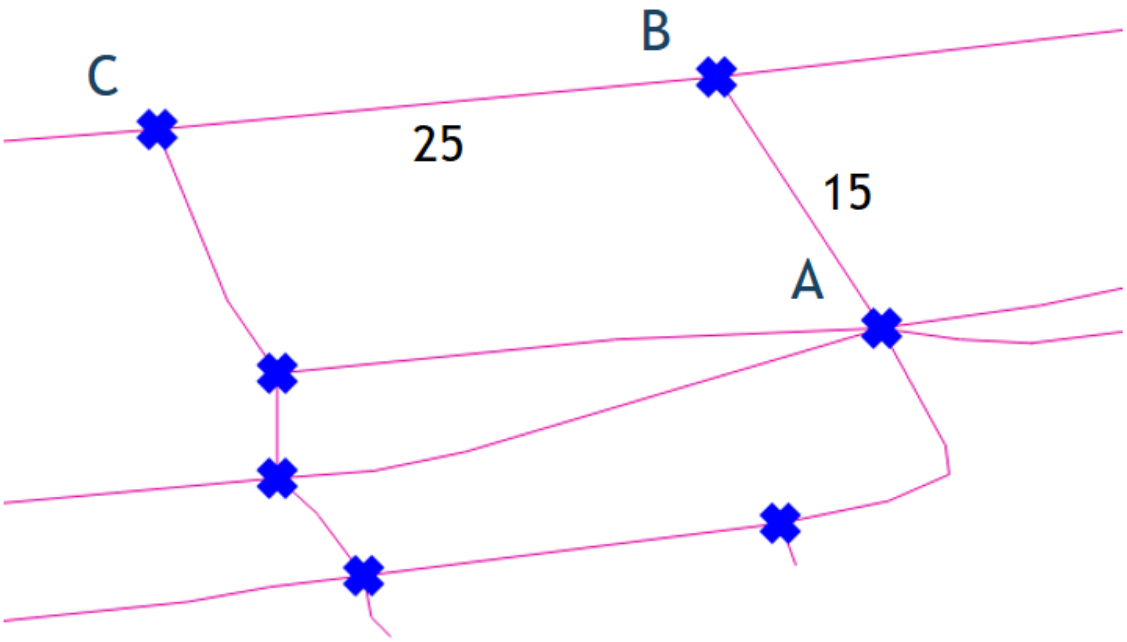


Figure 5. Road Graph Network with Street Intersection as Vertices and Streets as Weighted Edges

Most of the time, the absolute distance of intersection may be negligible compared to the length of the streets. For example, based on distance only, the cost of vertex A to vertex C may be calculated as 15 + 25 = 40. However, vehicles spend a significant amount of time waiting for traffic signals in an urban environment [20]In areas that do not contain traffic signals, vehicles still need to slow down in order to avoid potential oncoming or crossing traffics.

If the cost of the path is defined as the expected amount of time a vehicle needs to spend on the path from source S to destination D, a typical road network representation that does not associate cost with vertices is not suitable for minimizing total travel time cost.

(3) Cost and Potential Hazard Associated with Turns at Intersections

Turning at intersections are costly for vehicles on the road, it has been reported that UPS trucks (almost) never turn left [28]. This has become an important strategy for the company to reduce collisions and cost. Vehicles turning left at intersections may need to spend longer time waiting

for Traffic Signal change. The driver may not be absolutely certain if there are oncoming traffics on the opposite side of the road as the view might be obscured by vehicles turning left from the other side. Pedestrians that are crossing streets also need to be watched as they have a higher priority of the road. These problems are also valid for autonomous vehicles. If an autonomous vehicle obeys traffic laws, yield to oncoming traffics that are going straight at the intersection, and is also more conservative (not moving during yellow traffic signals), the vehicle may have problems ever performing a left turn at intersections without dedicated left turn signals.

Using the road network in Figure 5 again, if a left turn has a time cost of 5, and a right turn has a time cost of 2, the cost from A->C becomes 45, and the cost from C->A becomes 42. The cost of 5 or 2 cannot be added to any single edge. The argument is equally valid if the cost is based on potential hazards of the path from source to destination.

(4) Obeying One Way Roads and Turn Signals in Real Road Networks

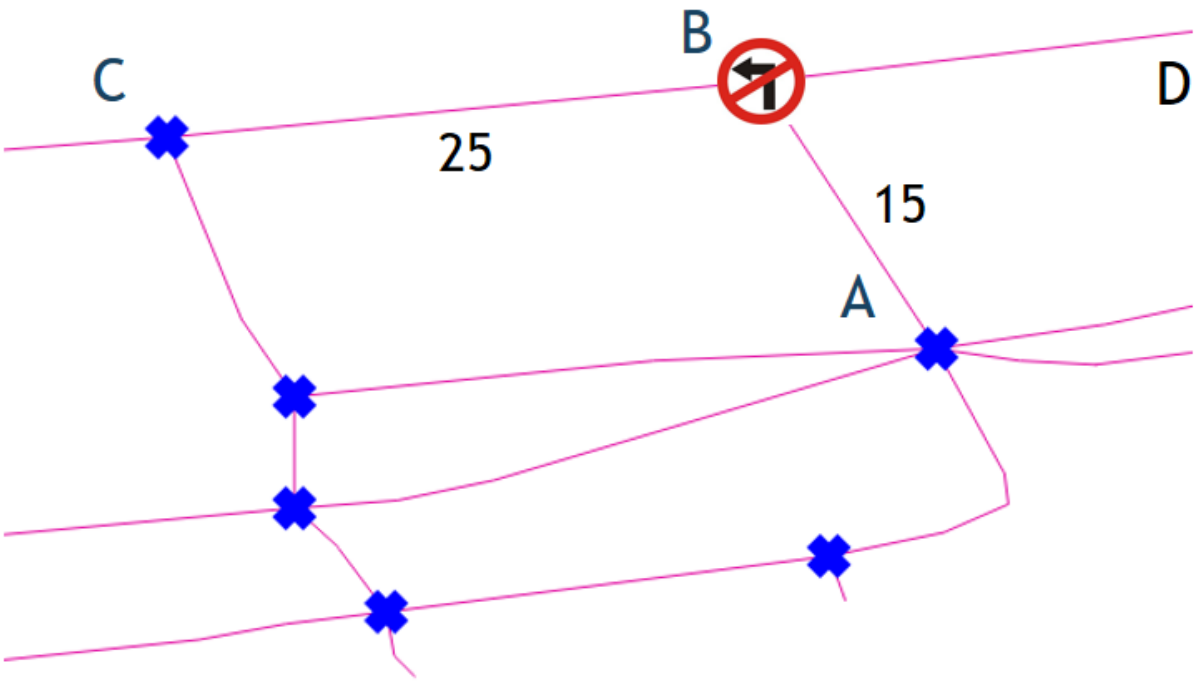


Figure 6. Road Network With Turn Restrictions

One way roads can be relatively easy to deal with if the road network is represented using weighted directed graph. For example, if B->C is a one-way road, in the graph representation, there should be a directed edge from B to C, and there shouldn't exist a directed edge from C to B. However, modeling turn restrictions is particularly difficult. If B->C is a drivable path, driving straight from D to C is allowed, but turning left at intersection B is prohibited, it is hard to write concrete rules to prohibit a path that requires left turn at intersection B.

One rule might be driving the path A->B->C is not allowed. However, in most path planning algorithms, such as Dijkstra's Algorithm and A* Algorithm, a vertex will be fully explored when it has the lowest current cost among all vertices that have not been explored yet. At the time of attempting to fully explore vertex B, if its current parent is assigned as A, we have already lost the possibility to get to B from D and use D->B->C as part of the shortest path.

3.3.2 Objectives

In the following section, we will propose a combination of Graph Representation and Path Planning Algorithm that will produce the lowest cost path that has the following properties:

(1) Models vehicle's driving direction
(2) Dynamic cost is allocated at intersections based on turn types and turn angles
(3) Turn restrictions and one way rules are adhered
(4) Optionally force the vehicle to enter road network by taking certain turns
(5) Provide the vehicle with an executable turn-by-turn instruction

3.3.3 Graph Representation

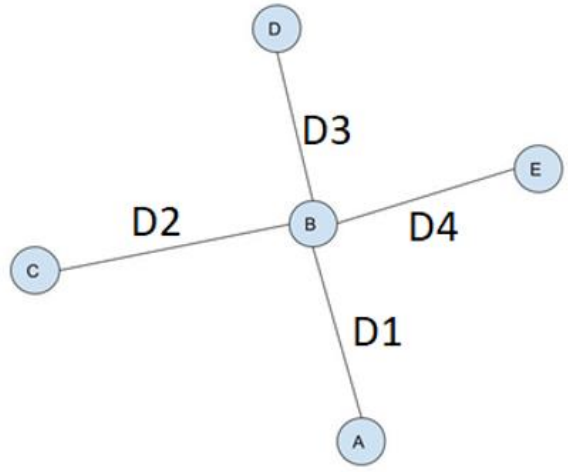


Figure 7. Road Graph Representation with Weighted Undirected Graph [15]

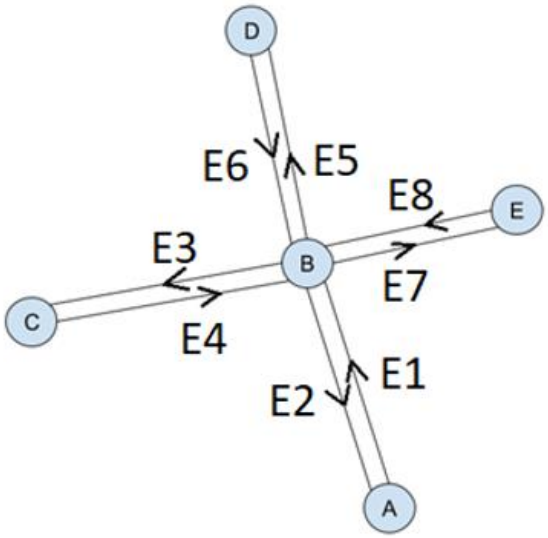


Figure 8. Road Graph Representation with Weighted Symmetric Directed Graph [17]

First, a weighted directed symmetric graph (defined in section 2.3.1) is created using the original undirected graph. Each vertex represents an intersection or a road end in the road network. Each edge represents one direction of a drivable path from one vertex to another. This representation requires preprocessing of the original graph. However, algorithms that

produce the shortest path satisfying properties in section 3.3.2 can also be created without preprocessing of the graph. Its implementation is not too different from the version with the preprocessed graph, the changes will be addressed in section 3.3.4.7.

Formally, the procedure for preprocessing the graph is:

Procedure Graph Preprocessing: Given a weighted undirected road network graph $G = (V, E)$, where each edge $e_{m,m\in[0,M)} \in E$ is associated with an unordered two element subset of Vertices $(v_{n,n\in[0,N)}, v_{n+1,n+1\in[0,N)})$, and has a weight $W(e_{m,m\in[0,M)})$. For each $e_{m,m\in[0,M)} \in E$, replace $e_{m,m\in[0,M)}$ with its symmetric directed duals $e_{2m,2m\in[0,2M)}$ and $e_{2m+1,2m+1\in[0,2M)}$. The edge $e_{2m,2m\in[0,2M)}$ is associated with an ordered two element subset of vertices $(v_{n,n\in[0,N)}, v_{n+1,n+1\in[0,N)})$, and $e_{2m+1,2m+1\in[0,2M)}$ is associated with an ordered two element subset of vertices $(v_{n+1,n+1\in[0,N)}, v_{n,n\in[0,N)})$. Weight of each of the directed dual is same as the original edge, i.e. $W(e_{m,m\in[0,M)}) = W(e_{2m,2m\in[0,2M)}) = W(e_{2m+1,2m+1\in[0,2M)})$.

3.3.4 Algorithm

3.3.4.1 Pseudo Code Comparison with Dijkstra's Algorithm

This algorithm is adapted from Dijkstra's Algorithm with a min-priority queue implementation [11]. As such, a side-to-side comparison between Dijkstra's Algorithm and Path Planning with Improved Intersection Considerations Algorithm is presented, with their differences underlined and italicized.

Dijkstra's Algorithm	Path Planning with Improved Intersections Considerations Algorithm
1 function Dijkstra(Graph, source):	1 function IntersectionImproved(Graph, source):
2 dist[source] ← 0	2 dist[source] ← 0
3	3
4 create min priority queue Q	4 create vertex set Q
5	5
6 for each vertex v in Graph:	6 for each vertex v in Graph:
7 if v ≠ source	7 if v ≠ source
8 dist[***v***] ← INFINITY	8 dist[(***v, from_edge, turn_type***)] ← INFINITY
9 prev[***v***] ← UNDEFINED	9 prev[(***v, from_edge, turn_type***)] ← UNDEFINED
10 ***explored[v] ← FALSE***	10 ***explored[v, from_edge] ← FALSE***
11 Q.add_with_priority(***v, dist[v]***)	11 Q.add_with_priority(***v, dist[v], from_edge, turn_type***)
12	12
13 while Q is not empty:	13 while Q is not empty:
14 ***u ← Q.extract_min()***	14 ***u, dist[u], u_from_edge, turn_type ← Q.extract_min()***
15 ***explored[u] ← TRUE***	15 ***explored[u, u_from_edge] ← TRUE***
16 for ***each neighbor v*** of u if !explored[v]:	16 for ***each neighbor v and edge v_from_edge*** of u if !explored[v,from_edge]:
17 ***alt ← dist[u] + weight(u, v)***	17 ***alt ← dist[u,u_from_edge] + length(u,v) + turn_cost(v_from_edge, u_from_edge)***
18 if alt < dist[v]	18 if alt < dist[v, v_from_edge, turn_type]
19 dist[***v***] ← alt	19 dist[(***v, from_edge, turn_type***)] ← alt
20 prev[***v***] ← u	20 prev[(***v, from_edge, turn_type***)] ← u
21 Q.decrease_priority(***v, alt***)	21 Q.decrease_priority(***v, dist[v], from_edge, turn_type, alt***)
22	22
23 return dist, prev	23 return dist, prev

Table 6. Pseudo Code Comparison Between Dijkstra's and Intersections Considerations Algorithm

Underlined and italicized parts are the differences between Dijkstra's Algorithm and the Intersection Improved Algorithm. The highlighted portions of Intersection Improved Algorithm are not the most accurate representation of the underlying data structure used to track the information. A detailed discussion of the underlying data structure will be presented in section 3.3.5. Nevertheless, the pseudo code is a good representation of the main ideas of the algorithm.

The key difference between Intersection Improved Algorithm and the original Dijkstra's Algorithm is the information tracked. In Dijkstra's Algorithm, information about edges are used but quickly discarded. When operating a Dijkstra's Algorithm on either a directed or undirected graph, only information related to vertices are being tracked. For example, in the "dist" array as shown in the Dijkstra's Algorithm pseudo code, distance is only calculated and tracked based on the distance between the vertices. Once the distance is calculated, only the parent vertex is kept track in "prev" array. Information about the incoming edge is thrown away. At the same time, the relationship between edges (the need of left turn or right turn), as well as any cost that might incur at vertices (intersections) is completely ignored. As a result, from a Dijkstra's Algorithm, the shortest path can be only generated based on the distance between vertices, and the shortest path only keeps track the vertices (intersections) on the path, which is hard to deduce a turn-by-turn driving instruction from.

In contrary, Path Planning with Improved Intersection Considerations Algorithm keeps track of more information. The basic information unit in the algorithm is a vertex-edge pair.

Definition **Vertex-Edge Pair**: Given a weighted symmetric directed graph $G = (V, E)$, a vertex-edge pair is defined as $(v_{n,n\in[0,N)} \in V, e_{2m,2m\in[0,2M)} \in E)$, where $e_{2m,2m\in[0,2M)}$ is associated with an ordered vertex set $(v_{n-1,n-1\in[0,N)}, v_{n,n\in[0,N)})$, N is the total number of vertices and M is the total number of directed edges. The vertex $v_{n,n\in[0,N)}$ is the outgoing vertex of the edge $e_{2m,2m\in[0,2M)}$.

3.3.4.2 Algorithm in Detailed Steps

In order to improve the clarity of the algorithm, turn_type will not be presented as part of the definitions. VE-pair stands for vertex-edge pair. SVE-pair stands for source vertex-edge pair.

(1) Initialization

Procedure **Path Planning with Improved Intersection Considerations Initialization**: Given a weighted symmetric directed graph $G = (V, E)$, to plan the lowest cost path from vertex $S \in V$ to vertex $D \in V$, initialize by:

For every vertex-edge pair $(v_{n,n\in[0,N)} \in V, e_{m,m\in[0,2M)} \in E)$, where $e_{m,m\in[0,2M)}$ is associated with an ordered vertex set $(v_{n-1,n-1\in[0,N)}, v_{n,n\in[0,N)})$, and $v_{n,n\in[0,N)}$ is the outgoing vertex of edge $e_{m,m\in[0,2M)}$, assign:

distance from SVE-pair to source vertex-edge pair: $D\big((S, e_{NULL}), (S, e_{NULL})\big) = 0$

parent of SVE-pair (S, e_{NULL}) to (S, e_{NULL}): $P((S, e_{NULL})) = (S, e_{NULL})$

distance from SVE-pair (S, e_{NULL}) to (v_n, e_m) for all $e_{m,m\in[0,2M)} \in E$:

$$D\big((S, e_{NULL}), (v_n, e_m)\big) = +Inf$$

parent of other VE-pair (v_n, e_m) for all $e_{m,m\in[0,2M)} \in E$: $P\big((v_n, e_m)\big) = (v_{NULL}, e_{NULL})$

Min Priority Queue: $Q\{[\big(D\big((S, e_{NULL}), (S, e_{NULL})\big),\ (S, e_{NULL})\big]\}$

exploration Status for all of VE-pair: EXPLORED$\{(S, e_{NULL})\}$ = False

Distances to all other vertex-edge pairs are marked as Infinity (Line 8). Through software optimization discussed in section 3.3.5, one does not have to go through all edges leading to each vertex. Correspondingly, the parent of each vertex-edge pair is another vertex-edge pair, which is marked as undefined for all vertex-edge pairs except for the parent vertex (Line 9). Upon initialization, all turn_types will be NULL, and will only be updated during calculation of the shortest path.

(2) Cost of Path Calculation

Definition **Path**: In a weighted symmetric directed graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$, a path from vertex-edge pair (v_0, e_0) to another vertex-edge pair (v_k, e_k) exists if there is a sequence:

$$P = < (v_0, e_0) , (v_1, e_1), (v_2, e_2), \dots (v_k, e_k) >$$

Such that e_i is associated with the an ordered two element subset of vertices (v_{i-1}, e_i) for $i \in [1, k)$, and e_0 is associated with (v_{any}, v_0).

Definition **Path Cost**: In a weighted symmetric directed graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$, cost of path from vertex-edge pair (v_0, e_0) to another vertex-edge pair (v_k, e_k) is:

$$D(P) = \sum_{i=1}^{k-1} W(v_i, v_{i+1}) + \sum_{i=1}^{k-1} TurnCost(e_i, e_{i+1})$$

For a given path : $P = < (v_0, e_0) , (v_1, e_1), (v_2, e_2), \dots (v_k, e_k) >$

Where: $W(v_i, v_{i+1})$ is weight of the edge associated with the ordered set (v_i, v_{i+1})

$TurnCost(e_i, e_{i+1})$ is the cost incurred transitioning from e_i to e_{i+1}.

A different cost of path is the core of this algorithm. The algorithm not only considers cost incurred on edges (streets), but it also considers cost incurred at vertices (intersections). As shown in Line 17 of the pseudo code, a temporary cost to the vertex-edge pair (v, e_{to_v}) is calculated using the distance from source S to (u, e_{to_u}), distance of $u \rightarrow v$ (which is same as $W(e_{to\_v})$), as well as the turn cost based on relationships between e_{to_u} and e_{to_v}. Turn cost should be higher if the turn is more difficult to execute for a vehicle. Therefore, a U-Turn will have the highest cost, followed by a Left-Turn and a Right-Turn. Turn cost can also be penalized by the size of the angle: the sharper the angle, the higher the cost.

(3) Update Distance array, Parent Array and Min Priority Queue

Definition **Distance Update Condition**:

An update condition is True if, in a weighted symmetric directed graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$, for a vertex-edge pair (v_k, e_k), there exists a path from source vertex-edge pair $P_{new} = <$ $(S, e_{NULL}), \dots (v_k, e_k) >$ for which $D(P_{new}) < D(P_{old})$

Where $P_{old} = < (S, e_{NULL}), \dots (v_k, e_k) >$ is the previously discovered shortest path.

Procedure **Distance Update**: In a weighted symmetric directed graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$, for a vertex-edge pair (v_k, e_k), upon discovery of a new path $P_{new} = < (S, e_{NULL}), \dots, (v_{k-1}, e_{k-1}), (v_k, e_k) >$ and "Distance Update Condition" is True, update the following for (v_k, e_k):

Parent of VE-pair (v_k, e_k): $P\big((v_k, e_k)\big) = (v_{k-1}, e_{k-1})$

Distance from SVE-pair (S, e_{NULL}) to (v_k, e_k): $D\big((S, e_{NULL}), (v_k, e_k)\big) = D(P_{new})$

Min Priority Queue: $Q.update\_priority(D(P_{new}), (v_k, e_k))$

The min priority queue is still ordered based on the cost from source S to the vertex-edge pair, which is in this algorithm based on a combination of cost of the edges and cost of the turns. Distances to vertex-edge pairs, as well as the parent of vertex-edge pairs, will be recorded. During calculation of turn_cost, the turn type needs to be identified. The turn type calculated will be recorded as turn_type in the main function. This information is not required for finding the shortest path based on edge and turn cost, but is required for generating turn-by-turn driving instruction for the vehicle.

(4) Exploration of New Vertex

Procedure Exploration of New Vertex: A vertex (v_k, e_k) will be explored if Min Priority Queue $Q.top = (v_k, e_k)$. During exploration, do:

1. For all edges e_{k+i} that is associated with an ordered vertex set (v_k, v_{k+i}), if $EXPLORED(v_{k+i}, e_{k+i})$ is False, and (v_{k+i}, e_{k+i}) has Update Condition True, then Update (v_{k+i}, e_{k+i}).
2. $EXPLORED(v_k, e_k) =$ True
3. $Q.remove\_top()$

3.3.4.3 Calculation of Turn Cost

Cost of Turn Type:

In order to calculation turn cost, the relationship between e_{to_u} and e_{to_v} has to be identified.

Definition CCW Angle Between Two Edges. For a pair of edges $e_{to\_u} \in E$ and $e_{to\_v} \in E$, where e_{to_v} is the directed edge from vertex u to vertex v, and e_{to_u} is the directed edge from another arbitrary vertex t to vertex u. The counterclockwise angle between e_{to_u} and e_{to_v} is given by:

$$CCW_{(e_{to\_u}, e_{to\_v})} = arctan2(v_y - u_y, v_x - u_x) - arctan2(t_y - u_y, t_x - u_x)$$

where the function $arctan2(y, x)$ gives the polar angle of the point (x, y).

If $CCW_{(e_{to\_u}, e_{to\_v})} < 0$, then $CCW_{(e_{to\_u}, e_{to\_v})} = CCW_{(e_{to\_u}, e_{to\_v})} + 360°$,
so that $CCW_{(e_{to\_u}, e_{to\_v})} \in [0°, 360°)$.

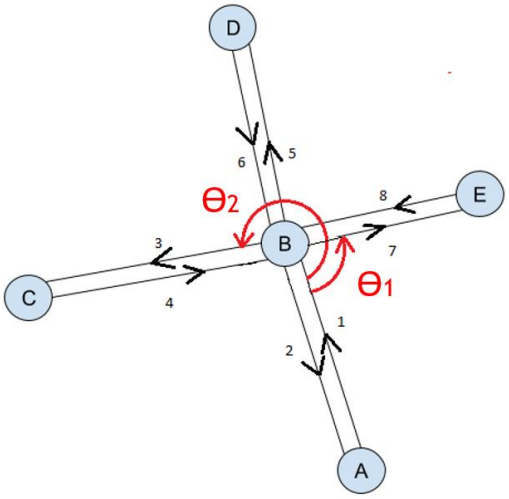


Figure 9. Polar Angle Between Edges in Graph

For example, in Figure 9, $CCW(e_1, e_7) = \theta_1 = 90°$, and $CCW(e_1, e_3) = \theta_2 = 270°$ angle. $CCW(e_1, e_2)$ is defined as the counterclockwise angle from edge e_1 to edge e_2. In the case of non-straight streets (edges), the street is represented as a list of geographical coordinates. The geographical coordinate that is the closest to the intersection (but not at the intersection) will be chosen to calculate the angles.

The relationship between turn type and value of CCW angle is shown in the following table:

Turn Type	Example Criteria 1	Example Criteria 2	Example Cost
Right	$\theta \in [10°, 170°)$	$\theta \in (0°, 160°)$	2
Straight	$\theta \in [170°, 190°)$	$\theta \in [160°, 200°)$	0
Left	$\theta \in [190°, 340°)$	$\theta \in [200°, 340°)$	5
U-Turn	$\theta \in [0°, 10°) \cup [340°, 360°)$	$\theta \in [340°, 360°)$	10

Table 7. Turn Type Along with Example Criteria and Cost

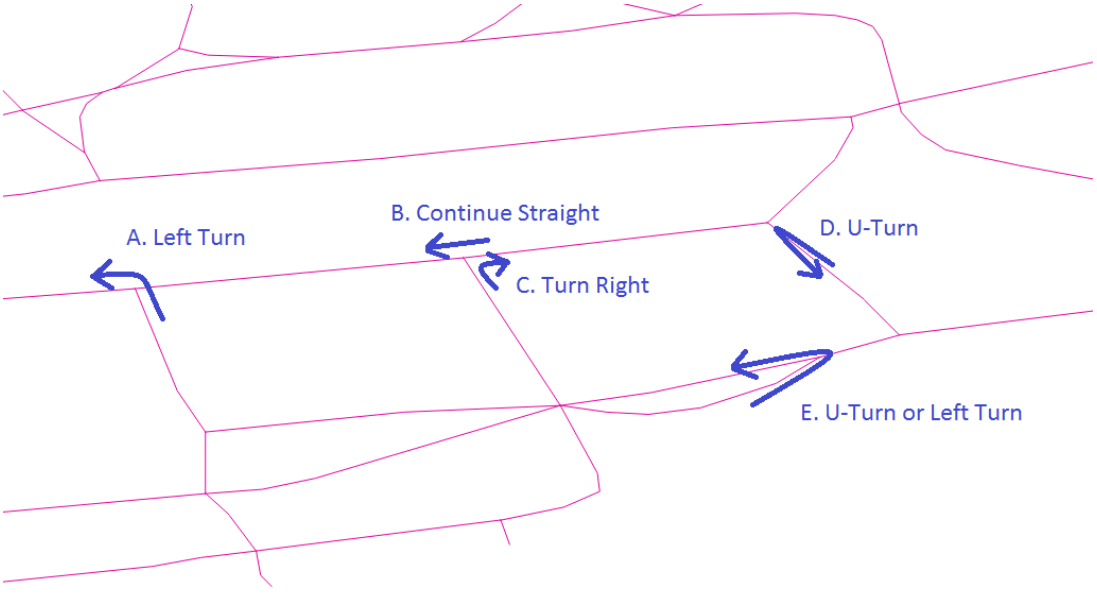


Figure 10. Example Turn Classification on a Real Road Network

There are several variabilities with the Turn type and their associated cost. First, the range of the crossing intersections straight can be variable. In “Example Criteria 2”, intersection crosses with up to $20°$ angle is categorized as crossing straight, where in “Example Criteria 1”, only $10°$ variation from heading straight is allowed. In addition, the definition of U-Turn can vary. On North American roads, it might be suitable to define a super sharp left turn as a U-Turn. However, in a road system where vehicles drive on the left side of the road, it might be more appropriate to define a super sharp right turn as a U-Turn. The cost associated with each type of turn can also be variable, so that penalizing certain behaviors can be easy. For example, on a road network that is relatively chaotic and does not have dedicated left turn signals, left turns should be penalized. On a road network that involves a lot of bicycles, right turns should be penalized, as right-turning vehicles need to yield the right-of-way to cyclists going straight and they may potentially hard to detect.

Cost of Angle Acuteness:

Angle cost should be following the rule that the smaller the angle, the higher the cost. The angle $\varphi \in [0°, 180°)$ between two edges e_1 and e_2, $\varphi(e_1, e_2)$, can be calculated from $CCW(e_1, e_2)$ using the following formula:

$$\varphi(e_1, e_2) = abs(180° - CCW(e_1, e_2))$$

A cost associated with acuteness of the angle can be given by:

$$\varphi_{cost}(e_1, e_2) = \frac{180°}{\varphi(e_1, e_2)} - 1$$

Where $\varphi(e_1, e_2)$ is expressed in degrees.

Dynamical Cost Based on Turn Type and Angle Acuteness:

The magnitude of turn type penalization and angle acuteness penalization can be dynamically adjusted:

$$total\_turn\_cost = \alpha \times turn\_type\_cost(e_1, e_2) + \beta \times \varphi_{cost}(e_1, e_2)$$

Turn Type for Turn-by-Turn Driving Instructions:

Turn type is found during calculation of the turn cost. This information can be tracked in the parent array in the main function. When a turn-by-turn driving instruction is needed, turn instructions can be found using the turn type stored.

3.3.4.4 Path Planning with Improved Intersection Considerations Lowest Cost Path Format

Through the tracking of extra information, a final path based on either on-path vertices or edges, or a combination of both can be proposed.

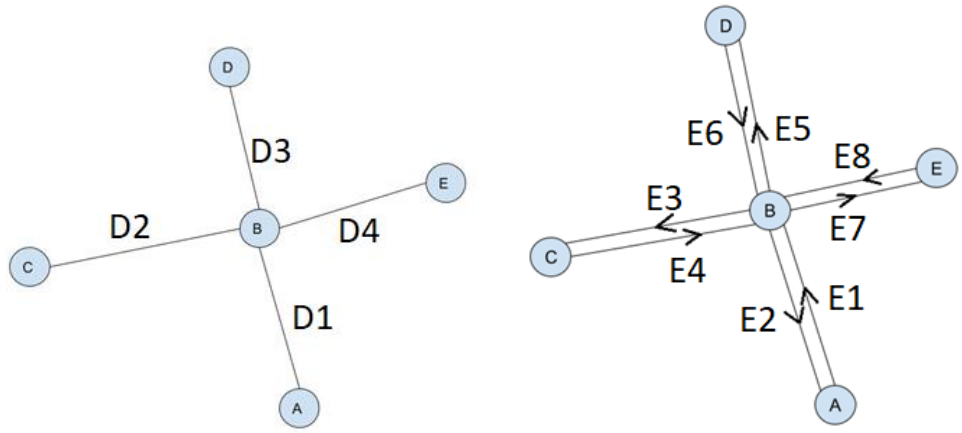


Figure 11. Weighted Undirected Graph and Weighted Symmetric Directed Graph

	Dijkstra's Algorithm	Road Graph with Intersection Considerations
Cost from A -> C	W(D1) + W(D2)	W(E1) + TurnCost(E1->E3) + W(E3)
Final Path from A -> C	[A, B, C]	[(A, NULL, NULL), (B, from E1, NULL), (C, from E3, left turn)]
Turn by Turn instruction based on Final Path	A -?-> B -?-> C	A Continue on E1 (to reach B) Turn left onto E3 (to reach C) Your destination has arrived

Table 8. Comparison with Dijkstra's Algorithm on Cost, Planned Path and Turn-by-Turn Instruction

Table 8 compares the information retrieved from Dijkstra's Algorithm and Road Graph with Intersection Considerations. Both attempting to find a path from Intersection A to Intersection C, Dijkstra's Algorithm gives a solution (Final Path) based on vertices: [A, B, C]. Road Graph with Intersection Considerations give a list of three element set information, from which one can deduce the vertices on the path: [A, B, C], all the edges on the path (with optional turn instructions): [E1, (turn left), E2].

3.3.4.5 Applying Turn Restrictions and One Way Road Restrictions

Turn restrictions can be easily recorded using edges. Using graph in Figure 11 as an example, given the following constraints:

(1) At intersection B, turning left from A into C it not allowed,
turning right from A into E is not allowed

(2) At intersection B, going straight from E to C is not allowed

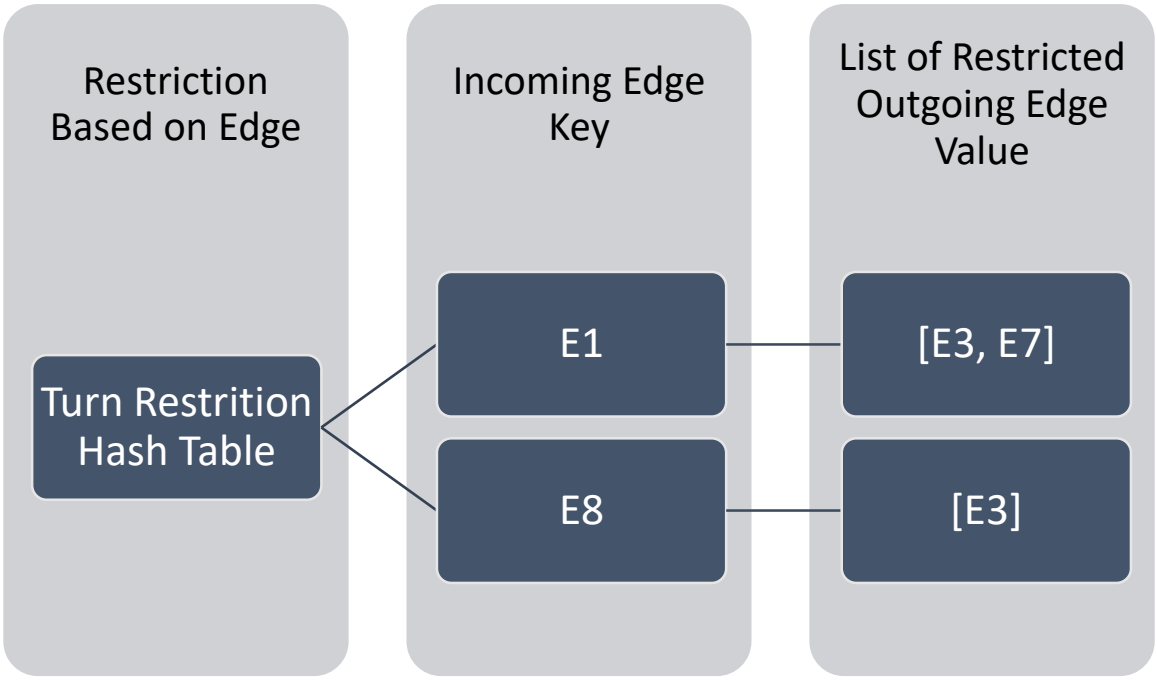


Figure 12. Hash Table Structure, Turn Restriction Based on Edges

Turn restrictions can be easily stored in a hash table, the lookup time for any incoming edge key is $O(1)$. Searching through the restricted outgoing edge is $O(k)$, where k is the size of the list. However, since 5-way or above intersections hardly exist, in practice, k is often less than 3, keeping the actual lookup time low.

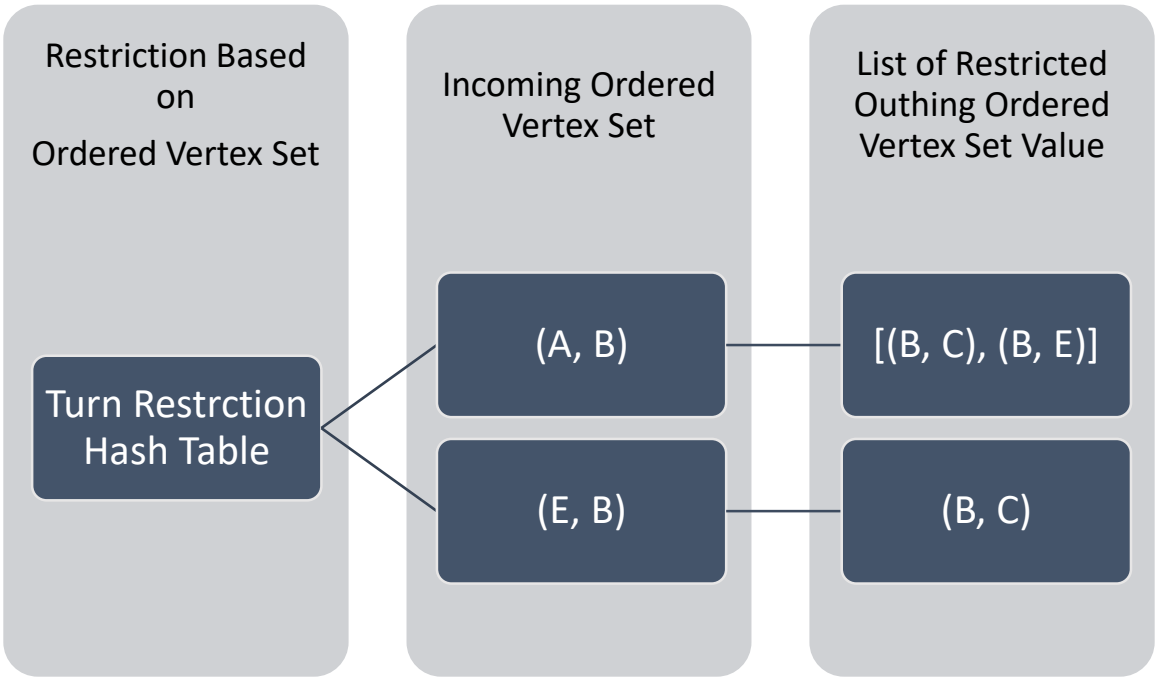


Figure 13. Hash Table Structure, Turn Restriction Based on Ordered Vertex Set Dual

This dual form is useful if one does not want to preprocess the map. Details of processing on the fly will be discussed in section 3.3.4.7. This turn restriction representation has same time complexity as the previous representation.

One Way Road Restrictions:

One way restrictions can be modeled in two ways: deletion of a directed edge or a set of turn restriction that forbids the use of the directed edge.

Using map in Figure 11 as an example, if traveling from intersection D to B is one way only (E5 is banned), the corresponding turn restrictions are:

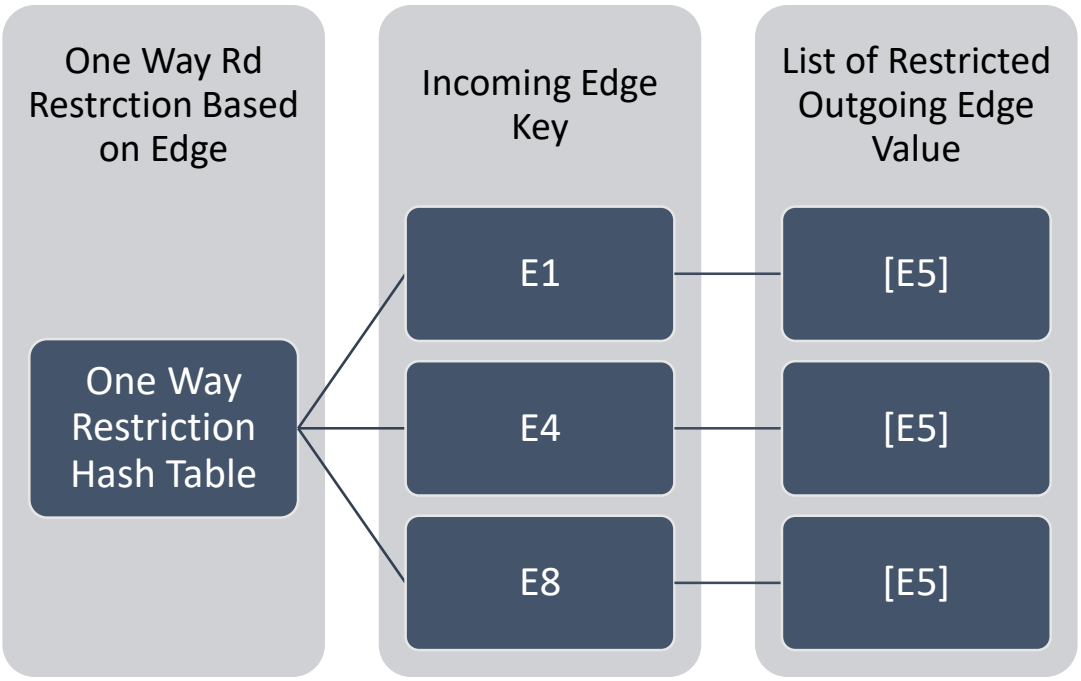


Figure 14. Hash Table Structure, One Way Road Restriction

3.3.4.6 Safe Driving Consideration: Force the Vehicle to Turn Right onto the Road Network, and Turn Right into the Destination

Vehicles often drive onto the road network from garages or parking spots that are not marked as part of the road network. When driving onto the road network, it is often safer and easier to drive right onto the road network as only one side of the crossing traffic needs to be watched. In order to force the vehicle to turn right onto the road network, how to move the vehicle onto the road network will be defined first.

Procedure Off-Graph Start: Given a weighted symmetric directed graph $G = (V, E)$ and a vehicle starting geographical location GC_{start}, if $GC_{start} \neq v_m.GC()$ for all $v_m \in V$, then create a new graph $G' = (V', E')$

Where $V \cup (v_{start}, v_{edge}) = V'$

$$E \cup \left(e_{start}, (v_{edge}, v_i), (v_{edge}, v_{i+1})\right) = E',$$

such that $v_{start}.GC() = GC_{start}$

$e_{start} = (v_{start}, v_{edge})$

$v_{edge}.GC()$ is the closes point to GC_{start} for all point on any edges.

For a vehicle to depart off-road, two new vertices and three new edges need to be added to the graph. If the vehicle is forced to take a direction at the beginning, the restriction can be simply added to the restriction hash table in section 3.3.4.4.

3.3.4.7 Road Graph with Improved Intersection without Preprocessing of the Graph

In section 3.3.3, we defined how to create a weighted directed symmetric graph from the original weighted undirected graph. Algorithms in the section 3.3.3 are based on this graph representation. However, it is equally valid run the Road Graph with Intersection Considerations algorithm on the original graph. The key differences are:

(1) Replace Directed Edge with Ordered-Vertex-Set

Definition Ordered-vertex-set: Given a directed edge e from vertex v_1 to v_2, edge e is associated with the ordered-vertex-set (v_1, v_2).

(2) Replace Vertex-Edge Pair (VE-pair) with Vertex-Ordered-Vertex-Set Pair (VO-Pair)

Definition Vertex-Ordered-vertex-set Pair: Given a weighted undirected graph $G = (V, E)$, a vertex-edge pair is defined as $(v_{n,n\in[0,N)} \in V, ordered(v_{n-1,n-1\in[0,N)} \in V, v_{n,n\in[0,N)} \in V))$, where $e_{m,m\in[0,M)} \in E$ is associated with the unordered version of the vertex set $(v_{n-1,n-1\in[0,N)} \in V, v_{n,n\in[0,N)} \in V)$. The vertex $v_{n,n\in[0,N)}$ is the outgoing vertex of the vertex set $(v_{n-1,n-1\in[0,N)}, v_{n,n\in[0,N)})$.

Using these two replacements, one can perform rest of the algorithm on-the-fly using ordered-vertex-sets instead of directed edges, and using VO-Pairs instead of VE-pairs. The algorithm will be syntactically more difficult to present but will not require extra computation.

3.3.5 Software Optimizations

In the algorithm initialization procedure presented in section 3.3.4.2, the steps for initializing the distance and parent arrays are:

1. Initialize distance from SVE-pair (S, e_{NULL}) to (v_n, e_m) for all $e_{m,m\in[0,2M)} \in E$: $D\big((S, e_{NULL}), (v_n, e_m)\big) = +Inf$
2. Initialize parent of VE-pair (v_n, e_m) for all $e_{m,m\in[0,2M)} \in E$: $P\big((v_n, e_m)\big) = (v_{NULL}, e_{NULL})$

Initializing the distance to all other VE-pair to infinity, and parent of all other VE-pairs to be NULL requires going through all edges and identifying all VE-pairs, which is not necessary. Instead, this information can be updated on-the-fly as new VE-pairs are discovered.

For Road Graph with Improved Intersection Algorithm, parent and distance are 2d arrays. Upon initialization, only the primary dimension based on the number of vertices in the graph is created. The secondary dimension that is based on the number of VE-pairs associated with each vertex will be dynamically created on the fly. This can be accomplished by using a Python List of List structure, where the inner List is mutable, allowing extra information to be recorded if needed [24].

Distance:

	$v_1 \in V$	$v_2 \in V$	…	$v_i \in V$	…	$v_n \in V$
$e_{i1} \in E$	0	$\boldsymbol{D(P < (v_1, e_{11}),}$ $\boldsymbol{(v_2, e_{21}) >)}$		$\boldsymbol{D(P < (v_1, e_{11}),}$ $\boldsymbol{(v_i, e_{i1}) >)}$		$\underline{\boldsymbol{D(P < (v_1, e_{11}),}}$ $\underline{\boldsymbol{(v_i, e_{i1}), (v_n, e_{n1}) >)}}$
$e_{i2} \in E$		$\underline{\boldsymbol{D(P < (v_1, e_{11}),}}$ $\underline{\boldsymbol{(v_i, e_{i1}), (v_2, e_{22}) >)}}$				
…						
$e_{ij} \in E$						
…						

Table 9. 2D Array Represent of Distance

Parent:

	$v_1 \in V$	$v_2 \in V$	…	$v_i \in V$	…	$v_n \in V$
$e_{i1} \in E$	(v_1, e_{11}), $turn_{null}$	$(\boldsymbol{v_1, e_{11}})$, $\boldsymbol{turn(e_{11}, e_{21})}$		$(\boldsymbol{v_1, e_{11}})$, $\boldsymbol{turn(e_{11}, e_{i1})}$		$\underline{\boldsymbol{(v_i, e_{i1})}}$, $\underline{\boldsymbol{turn(e_{i1}, e_{n1})}}$
$e_{i2} \in E$		$\underline{\boldsymbol{(v_i, e_{i1})}}$, $\underline{\boldsymbol{turn(e_{i1}, e_{22})}}$				
…						
$e_{ij} \in E$						
…						

Table 10. 2D Array Representation of Parent, including Turn Types

An Example with Two Iterations of the Algorithm:

Assume v_1 is the source vertex, and its neighbors are v_2 and v_i, at initialization, distance and parent for entry (v_1, e_{11}) is updated, where $e_{i1} = NULL$.

In the first iteration of the algorithm, distances and parents of a VE-pair associated with v_2, and a VE-pair associated with v_i can be updated, shown in bold in Table 9 and 10.

At second iteration, assume v_i has the lowest cost, the neighbor of v_i: v_2 and vn are both updated. For v_2, another VE-pair that has not been discovered before will be added into the 2d array. The updates are shown in bold and underline in Table 9 and 10.

Path to any of the discovered VE-pair can be deduced easily by going through the Parent 2d array. For example, in order to reach (v_2, e_{22}), one starts from (v_2, e_{22}), observe its parent is (v_i, e_{i1}), which in turn has parent (v_1, e_{11}), i.e. the source VE-pair.

Reverse this observation, the path can be constructed,

$$P = < (v_1, e_{11}), (v_i, e_{i1}), (v_2, e_{22}) >$$

and the turn instructions can also be included

$$P_{turn-by-turn} = < (v_1, e_{11}), turn(e_{11}, e_{i1}), (v_i, e_{i1}), turn(e_{i1}, e_{22}), (v_2, e_{22}) >$$

3.3.6 Heuristic Optimization

Heuristics can be used to optimize the average performance of the algorithm [12]. In the min priority queue Q, instead of ordering the vertices using actual cost from source vertex S, the vertices can be ordered using a total estimated cost from source to destination, which is defined below

Definition **Estimated Total Path Cost**: In a weighted symmetric directed graph $G = (V, E)$, the total estimated cost of path from source vertex-edge pair (S, e_{NULL}) to one of the destination vertex-edge pair $\left(D \in V, e_{m,m\in[0,2M)} \in E\right)$ via a vertex-edge pair (v_k, e_k) is:

$$D(P) = \sum_{i=1}^{k-1} W(v_i, v_{i+1}) + \sum_{i=1}^{k-1} TurnCost(e_i, e_{i+1}) + H((v_k, e_k), (D, e_m))$$

For a given actual path : $P = < (v_0, e_0) , (v_1, e_1), (v_2, e_2), \dots (v_k, e_k) >$

Where: $(v_0, e_0) = (S, e_{NULL})$

$H((v_k, e_k), (D, e_m))$ estimates the cost from (v_k, e_k) to (D, e_m)

$W(v_i, v_{i+1})$ is weight of the edge associated with the ordered set (v_i, v_{i+1})

$TurnCost(e_i, e_{i+1})$ is the cost incurred transitioning from e_i to e_{i+1}.

This heuristic, if admissible, will guarantee the correctness of the solution if it always underestimates the cost of the actual path. An admissible heuristic is one that always underestimates the actual cost from one VE-pair to another. In our algorithm, a heuristic using the straight line distance between two vertices are always admissible, as the cost to get from one VE-pair to another will at least be the straight line distance.

4 Result

The following result is generated on a laptop with AMD 18-4500M APU without using the integrated card for acceleration. The memory on board are 2x4096MB DDR3 800MHz memories. Hard drive has 1TB capacity with 5400RPM. The system is not nearly competitive with other high-performance devices released in 2018. Using the latest devices should result in a significant performance improvement.

4.1 Location Searching Performance Evaluation

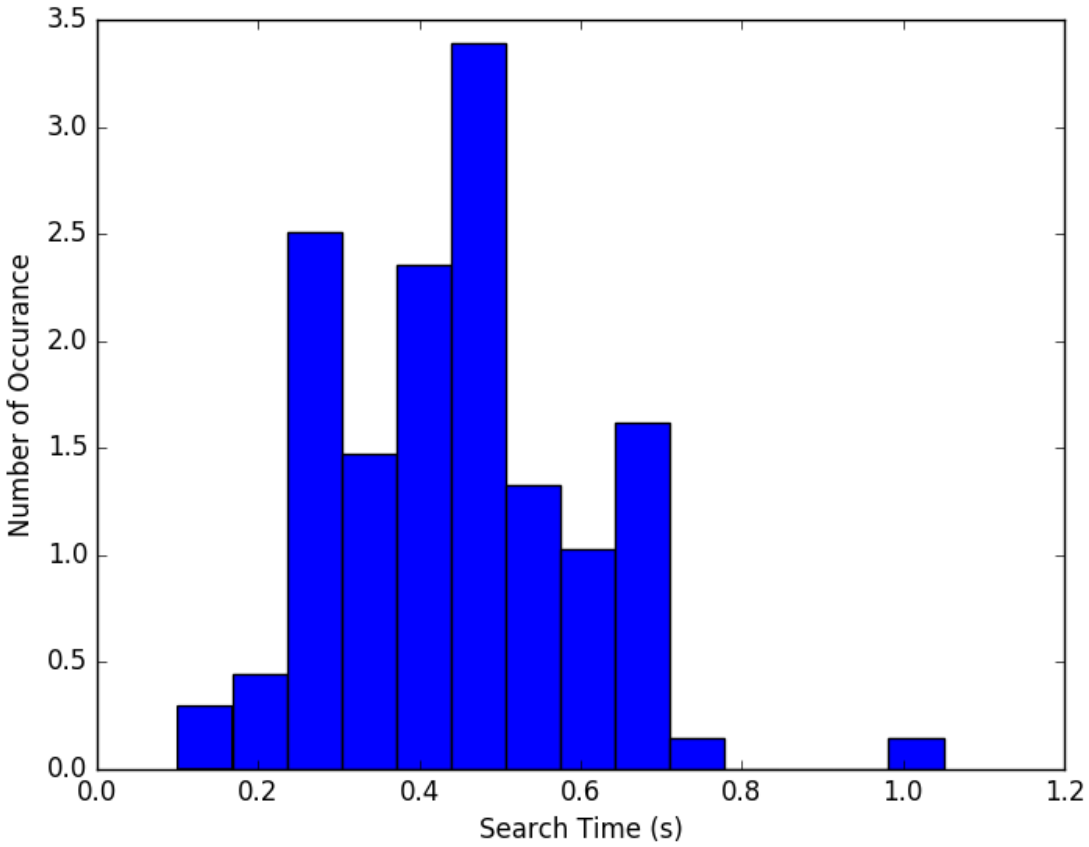


Figure 15. Intersection Location Search Time for 100 Random Intersections

To evaluate the performance of searching of intersections using the on-disk hash table, one hundred existing intersections has been selected. To clear the potential cache of data in

memory, the device has been rebooted before performing the test. It can be seen that searching for an intersection in the entire North American database has been very fast through a clever design of the data structure. The average time of search is around 0.5s.

Since the PostgreSQL implementation is not done on the device, a head to head comparison of performance is not very possible. However, on a faster device, the PostgreSQL implementation searches for a street intersection usually in between 4 and 10 seconds, which is significantly slower than the on-disk hash table implementation.

4.2 Path Planning with Improved Intersection Considerations Performance Evaluation

In the following section, we will look into the difference between several paths proposed by Dijkstra's and Road Graph with Intersection Considerations. We will also present their difference in runtime on a large amount of random starting and ending locations.

4.2.1 Comparison of Path Planned

The following paths are planned from coordinate (110.74374W, 62.40692N), to (110.70546W, 62.3656N), on map data provided by SAE International.

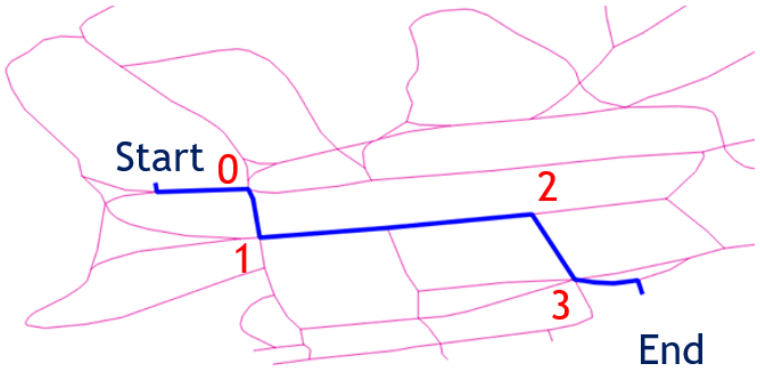


Figure 16. Path Planned from Start to End using Dijkstra's Algorithm

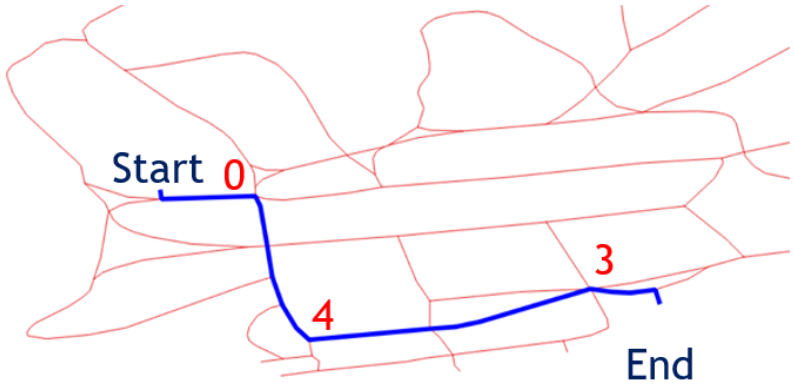


Figure 17. Path Planned from Start to End using Road Path using Road Graph with Intersection Considerations

As seen from the figures, Dijkstra's algorithm will produce a path that involves 4 close to $90°$ turns: two right turns and two left turns. Road Graph with Intersection Considerations algorithm chooses a route with a slightly longer total distance, but only includes two close to $90°$ turns: right turn at intersection 0, and left turn at intersection 4. The 'turn' at intersection 3 has been classified as cross straight in the algorithm.

4.2.2 Safe Driving Consideration: Path Planned with Right Turn onto Road Network

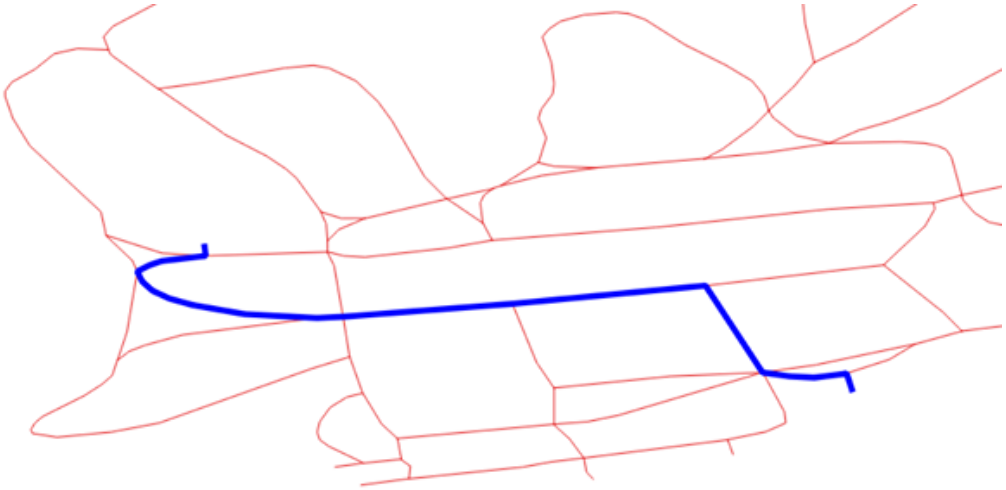
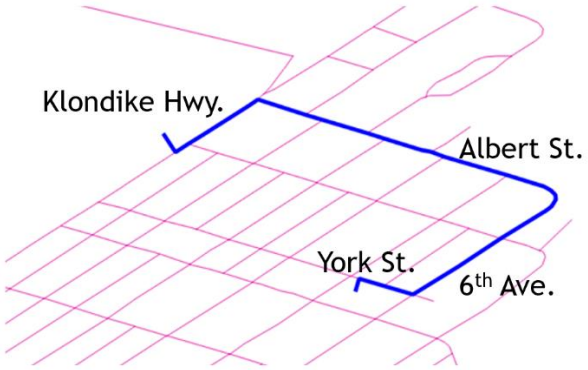


Figure 18. Vehicle forced to turn right onto the road network, and forced to turn right into the destination

Another improvement is shown in the above figure, using Road Graph with Intersection Considerations Algorithm, the vehicle can be easily forced to turn right onto the road network at the beginning. Compare Figure 18 and Figure 17, it can be seen that the vehicle is taking a much safer route by turning right onto the road. And it is only performing $90°$ turns at 3 intersections, which is still less in the number of turns compared to the path proposed by Dijkstra's Algorithm.

4.2.3 Turn-by-Turn Instruction



Turn **Left** onto Klondike Hwy
Turn **Right** onto Albert St.
Slide **Right** onto Albert St.
Continue **Straight** onto 6th Ave
Turn **Right** onto York St.
Turn **LEFT** onto YOUR DESTINATION

Figure 19. Turn-by-Turn Instruction Auto Generated from Algorithm

As shown in Figure 19, the Vehicle is forced to turn left onto Klondike Hwy, and turn left into the destination. The turn-by-turn instruction is naturally generated using information about turn types that have previously computed. After proper merging of the turn instructions (such as removing unnecessary continue straight when there is no street name change), a concise executable turn-by-turn instruction can be provided to the autonomous vehicle.

4.2.3 Algorithm Speed and Benefit Comparison on Real Road Networks

One hundred random starting and ending geographical coordinates have been selected in the Greater Toronto Area (GTA) [29]. On the map database provided by SAE International, four algorithms have been running to generate the shortest path between these locations. There was no requirement on the starting turn type. There was no angle penalization. A left turn is penalized the same as a 40-meter regular street (with speed limit 40km/h). Each right turn has the same cost as a 15-meter regular street. Each U-turn is equivalent to a 100-meter regular street in terms of cost. A straight crossing of intersections is not penalized. The result is shown in the graph below:

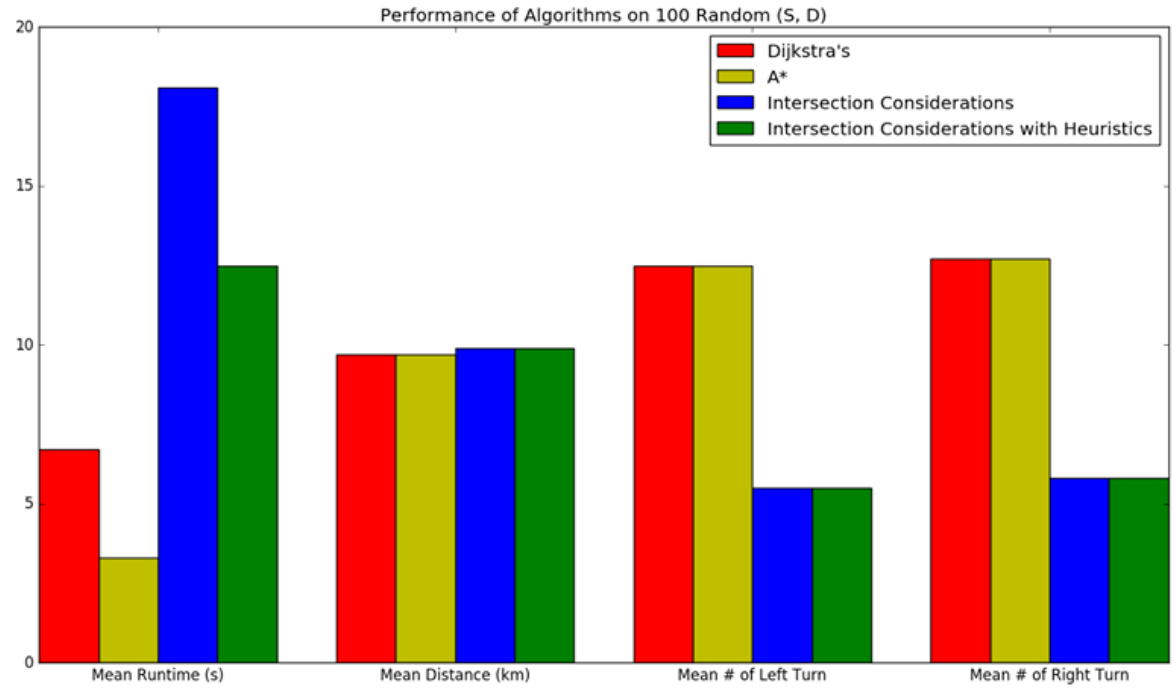


Figure 20. Comparison of Run Time and Properties of Proposed Shortest Path for Different Algorithms

Note that the number of left turns is computed offline: computed after Dijkstra's Algorithm and A* Algorithm has proposed its shortest path. Therefore, counting of the number of left turns does not have an impact on the runtime of Dijkstra's Algorithm and A* Algorithm.

Road Graph with Intersection Considerations Algorithms takes longer to run than Dijkstra's Algorithm. The conclusion is the same between the heuristic version of the two algorithms. The advantage of Intersection Considerations is a significantly lower amount of turns in the proposed final path. On average, there are only half of the turns compared to Dijkstra's Algorithm. The total distance of the path is very similar, with Intersection Considerations Algorithm proposing slightly longer paths.

As expected, the heuristic versions and non-heuristic versions for both types of algorithms are generating the same result. Which means heuristic only helped improving efficiency, without compromising optimality of the solution. A* almost reduced the runtime of Dijkstra's Algorithm by half, which is a much more significant improvement than using heuristics on Road Graph with Intersection Considerations Algorithm. The reason is Intersection Considerations is using

the same heuristic (straight line distance). However, the actual cost in Intersection Considerations is larger than that in Dijkstra's due to the extra turn costs. As a result, there is a larger gap between the straight line cost estimation and the actual cost, resulting in a less efficient heuristics algorithm.

5 Summary

In this thesis, a fast street intersection search using hash-table technique has been proposed and implemented. On average, it takes only 0.5 seconds to search for a street intersection in the entire North America map database. Its performance has been compared with searching using traditional SQL database queries, where an improvement on the order of 10^2 has been seen. A search blur algorithm that helps improving the chance of retrieving search results has been proposed and implemented. For road path planning, the Road Graph with Improved Intersection Considerations Algorithm has been proposed to incorporate cost incurred at intersections in real road networks without the need for preprocessing of the map. The algorithm can automatically generate executable turn-by-turn instructions. Using real Toronto map data, it is shown that the algorithm is successful at reducing the number of turns required in the proposed path by about half, without significantly increasing the total distance of the path or the computation time required.

References

[1] "AutoDrive Challenge," SAE International, [Online]. Available: https://www.sae.org/attend/student-events/autodrive-challenge/. [Accessed 09 04 2018].

[2] SAE International, "Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems," 16 01 2014. [Online]. Available: https://saemobilus.sae.org/content/j3016_201401. [Accessed 09 04 2018].

[3] "SAE AutoDrive Challenge Competition Guide," SAE International, 2017.

[4] H. F. K. S. S. Avi Silberschatz, Database System Concepts, McGraw-Hill, 2010.

[5] J. D. G. F. Kai Hwang, Distributed and Cloud Computing, 2011.

[6] M. T. Goodrich, Data Structures and Algorithms in Python, 2016.

[7] P. V. Dooren, "Graph Theory and Applications," 2009. [Online]. Available: http://www.hamilton.ie/ollie/Downloads/Graph.pdf. [Accessed 09 04 2017].

[8] F. Blöchliger, "Topomap: Topological Mapping and Navigation Based on Visual SLAM Maps".

[9] P. S. D. S. D. D. Robert Geisberger, "Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks," *C.C. McGeoch (Ed.): WEA 2008, LNCS 5038,* pp. 3-9-333, 2008.

[10] "Topological Sort," [Online]. Available: http://www.cs.utoronto.ca/~tabrown/csc263/2014W/week9.pdf. [Accessed 09 04 2017].

[11] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," in *25th Annual Symposium on Foundations of Computer Science. IEEE*, 1984.

[12] P. E. N. N. J. R. B. Hart, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics,* p. 100–107, 1968.

[13] B. H. G. Y. C. G. E. Casey Whitelaw, "Using the Web for Language Independent Spellchecking and," 2009. [Online]. Available: http://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/36180.pdf. [Accessed 09 04 2018].

[14] P. Norvig, "Write a Spelling Corrector," 08 2016. [Online]. Available: https://norvig.com/spell-correct.html. [Accessed 09 04 2018].

[15] Princeton Computer Science, "Undirected Graphs," [Online]. Available: https://www.cs.princeton.edu/~rs/AlgsDS07/11UndirectedGraphs.pdf. [Accessed 11 04 2018].

[16] Princeton Computer Science, "Directed Graphs," [Online]. Available: https://www.cs.princeton.edu/~rs/AlgsDS07/13DirectedGraphs.pdf. [Accessed 09 04 2018].

[17] Stanford Computer Science, "Graphs and Relations," [Online]. Available: https://web.stanford.edu/class/archive/cs/cs103/cs103.1132/lectures/05/Small05.pdf. [Accessed 09 04 2018].

[18] NIST, "shortest path," [Online]. Available: https://xlinux.nist.gov/dads/HTML/shortestpath.html. [Accessed 09 04 2018].

[19] S. Sawlani, "Explaining the Performance of Bidirectional," [Online]. Available: https://digitalcommons.du.edu/cgi/viewcontent.cgi?article=2303&context=etd. [Accessed 09 04 2018].

[20] A. G. Stephan Winter, "Modeling Costs of Turns in Route Planning," [Online]. Available: https://pdfs.semanticscholar.org/a735/e1cd1b724932b1844d5b21038d287967a59f.pdf. [Accessed 09 04 2018].

[21] "PostgreSQL 10.3 Released!," PostgreSQL, [Online]. Available: https://www.postgresql.org/. [Accessed 09 04 2018].

[22] SAE International, "Mapping Challenge Map Database".

[23] "QGIS A Free and Open Source Geographic Information System," QGIS, [Online]. Available: https://www.qgis.org/en/site/. [Accessed 09 04 2018].

[24] "Python Data Structures," Python, [Online]. Available: https://docs.python.org/2/tutorial/datastructures.html. [Accessed 09 04 2018].

[25] "11.1. pickle — Python object serialization," Python, [Online]. Available: https://docs.python.org/2/library/pickle.html. [Accessed 09 04 2017].

[26] "11.4. shelve — Python object persistence," Python, [Online]. Available: https://docs.python.org/2/library/shelve.html. [Accessed 09 04 2018].

[27] "16.6. multiprocessing — Process-based “threading” interface," Python, [Online]. Available: https://docs.python.org/2/library/multiprocessing.html. [Accessed 09 04 2018].

[28] "Why UPS trucks (almost) never turn left," CNN, 23 02 2017. [Online]. Available: https://www.cnn.com/2017/02/16/world/ups-trucks-no-left-turns/index.html. [Accessed 09 04 2018].

[29] "TORONTO NEIGHBOURHOOD GUIDE," [Online]. Available: http://www.torontoneighbourhoods.net/suburbs. [Accessed 09 04 2017].