Graph Traversal Algorithms for Transit Optimization

Ben Kronberger

Urbanization is a growing trend in modern society, and in order to accommodate this increasing population cities invest large amounts of time, energy, and effort into bolstering their public services. One of the most relevant and useful services for the urban population is public transit. Planning of public transit is key in making sure it is effective and useful. This planning takes into account relevant locations for stations on routes, the distances between these stations, and how their locations can positively impact the surrounding neighborhoods and general populace. A relevant problem arises from this, that being planning routes between points using as much existing infrastructure as possible. This problem can be abstracted to one of search algorithm utilization, as an effective method of determining paths between points becomes a problem of graph searching using weighted edges. In this investigation, different methods will be used to find routes in a node graph. These methods include Dijkstra's algorithm for shortest paths, a bidirectional implementation of Dijkstra's algorithm, A* search, and the Bellman-Ford algorithm.

1 Problem Background and Literature Review

In their paper *Genetic Algorithm for Scheduling Routes in Public Transport* de los Angeles Saez Blasquez et al discuss a genetic algorithm for optimizing distance and carbon emissions in a simulated metro environment [BGGMEP14]. They then compare this algorithm with Dijkstra's shortest paths algorithm in order to establish whether the proposed genetic algorithm is more efficient. The paper first covers the definition of the graph representation that is used for the algorithms, denoting edges and vertices as paths and stops. Next it defines both Dijkstra's algorithm and genetic algorithms in order to later discuss their pertinence of these concepts to their application. Finally the genetic algorithm is proposed and defined by selection, crossover, and mutation stages. In their proposition, a main facet is the reduction of graph size in each search by concatenating non-junction nodes between the origin and destination. The summary of their findings shows that their genetic algorithm requires less computational effort and outperforms the two classical methods (Dijkstra's and standard genetic algorithm).

A Benders Decomposition Algorithm for Demand-Driven Metro Scheduling by Schettini et al proposes a new approach to metro scheduling [SJM22]. They discuss scheduling trains individually instead of between terminals in fixed timetables. This is accomplished by using a Benders-based branch-and-cut algorithm on a path-based formulation of the problem. This approach simplifies complicating variables in a complex problem, making the subproblem significantly easier to solve. Once this subproblem is determined, the decomposition iterates by projecting this relaxed "sub-solution" onto the space of the remaining variables, converging to optimality. The paper then goes on to discuss the problem space and representation of their variables, and defines lemmas upon which they construct the guiding equations for their algorithm. In their conclusion, they observe that their algorithm decreased the maximum number of boarded passengers, the variance of the occupancy of trains, and the distance traveled by the trains.

Hasan et al in *A Genetic Algorithm-Based Optimal Train Schedule and Route Selection Model* evaluate several strategies for solving schedule and route optimization and eventually select a genetic approach [MSM+21]. In their evaluation they investigate Tabu search, Shifting Bottleneck algorithms, and their proposed genetic algorithm. After being tested with different historical data sets, they conclude that their genetic approach measures with far higher accuracy than the other approaches considered.

In their conference paper *An integrated and optimal scheduling of a public transport system in metro Manila using genetic algorithm* Escolano et al focus their efforts to develop a genetic algorithm that concerns itself with dispatching and scheduling of vehicles in their model in order to quell passenger demand and congestion [EDF14]. The optimization shown lies in decreasing the transfer time of passengers at transfer nodes while satisfying the constraints of traffic demand, departure time, and headway. They then discuss the mathematics being used to model the system and how they operate under different constraints, and then their implementation in C++. They observe their method to be quite efficient, and cite numbers of boarding/departing passengers and vehicle headway as major factors in getting these results.

In their paper *Train re-scheduling with genetic algorithms and artificial neural networks for single-track railways* Dundar and Sahin investigate methods of rescheduling timetables for trains in response to various issues and confounding variables [D13]. In doing so, they also discuss the building of a genetic algorithm for the purpose of short-track route optimization. They compare their genetic algorithm against a neural network trained with actual conflict resolution data from Turkish State Railways, and found that the genetic algorithm outperformed the neural network for small sized problems.

Modeling and Solving the Train Timetabling Problem from Caprara et al proposes a graph theoretic formulation for the problem of timetable scheduling using a directed multigraph [CFT02]. This allows them to derive a linear programming model that utilizes Lagrangian relaxation in its method of solving. They then go on to discuss more aspects of their modeling of the situation, including definitions for nodes and arcs to aid solving. This is all applied to a heuristic algorithm for solving of the overall problem.

Public transport route planning: Modified Dijkstra's algorithm by Bozyigit et al discusses the application of Dijkstra's algorithm for shortest paths to the problem of metro route scheduling, noting that it does not consider factors such as number of transfers and walking distances [BAN17]. They then impose a penalty system in order to account for these, and apply the modified algorithm to a real-world public transport network and compare the results to that of the standard Dijkstra approach.

Lewis discusses modified versions of Dijkstra's algorithm in *Algorithms for Finding Shortest Paths in Networks with Vertex Transfer Penalties* [Lew20]. These algorithms are discussed in relation to graphs of different sizes, outline procedures for imposing penalties upon vertices. These penalties can be used to simulate things like transfer time at a station in a public transport problem. These algorithms are then tested on various graphs and evaluated against the standard Dijkstra algorithm with positive results in specific cases, highlighting constraints and considerations of the larger problem.

Mitra et al investigates several algorithms for graph search and routing in *A Survey on Routing Algorithms for Efficient and Optimised Railway Scheduling* [MMP14]. Among discussing and comparing algorithms, the setup of the experiment and comparison thereof is particularly helpful for the purposes of this project.

In summary of the referenced materials, there exists a complicated problem of graph searching and optimization for public transport networks. The formulation of the graph model and subsequent factors to consider are outlined well as it relates to this study by de los Angeles Saez Blasquez et al, Caprara et al, and Lewis [BGGMEP14] [CFT02] [Lew20]. Variations on Dijkstra's algorithm are proposed by de los Angeles Saez Blasquez et al, Bozyigit et al, and Lewis [BGGMEP14] [BAN17] [Lew20]. Genetic algorithms are proposed by Hasan et al, Escolano et al, and Dundar and Sahin [MSM+21] [EDF14] [D13]. Other propositions for solving this problem are Benders Decompositon from Schettini, Tabu search and Shifting Bottleneck algorithms from Hasan et al, and a linear programming based heurisitc algorithm from Caprara et al [SJM22] [MSM+21] [CFT02].

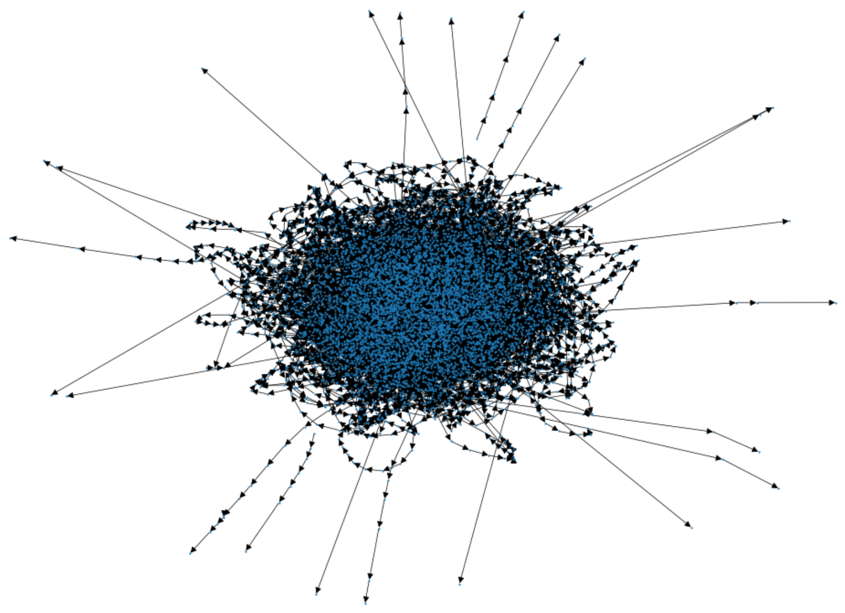
2 Problem Approach

2.1 Representation

The foundation of this problem is a node graph. A node graph is a collection of nodes, with edges assigned that connect them. In this problem, a node is defined as a transit stop location with a name, latitude, and longitude. The data provided by MetroTransit includes other characteristics of each transit stop, but for the purposes and scope of this investigation only name and location are considered. In addition, edges are defined in the graph by distances between each stop. Given the data, each edge is one that connects two stops on a line, and in this representation each line is aggregated into the larger node graph so that every stop is connected to every other stop by way of intermediate stops between them. This is to allow for any possible route between any two stops to be searched for and to eliminate orphaned stops due to characteristics of real-world lines and metro routes. The node graph used also makes no consideration for stops that are solely intended for the light rail trains (LRT), busses, or bus rapid transit (BRT) modes of transportation. This combination of routes and stops creates a generally connected graph with no barriers between stops other than physical distance and predetermined bidirectional connections. These constraints allow the problem to more accurately align with the goals of investigating optimal routes, rather than being limited by the necessity for new infrastructure to be built in the real world. Edge weights are then applied so that the algorithms may more accurately evaluate optimal routes, and these weights are calculated from the real-world distance between routes.

2.2 Graph Data

The data used to construct this graph is open source and provided directly by MetroTransit, offering a dataset that models the actual 8679 transit stops across every line and route in Minneapolis. This data was selected in order to model a real world problem in a real world location. This data is provided in the General Transit Feed Specification (GTFS) format, which is an open source project that transit authorities are able to participate in to allow for independent development using real world datasets. in this specification are many aspects that go far beyond the scope of this examination, and the data subsets used are those that name the stops and their locations. This data is then compiled into a Networkx graph using Peartree and Pandas. Peartree was used to convert the GTFS feed into Pandas data frames which were passed to Networkx. Using these tools the feed data was translated into a list of stops and a list of edges, and these were combined into a graph representing accurately every transit stop in Minneapolis and their edges.



2.3 Solving the Problem

For the purposes of this project, the problem is formulated to investigate graph search algorithms as they relate to a node graph. The solution to this problem will be the algorithm that performs best across a series of experiments, with the results of these experiments viewed in aggregate. Considerations will be made for performance on paths of varying lengths, so as to investigate applications to paths in a short space of investigation as well as those in a larger space.

3 Experimentation and Results

3.1 Experiment Design

The code for this investigation is written in a fashion that allows each algorithm to be run on a series on inputs, being evaluated each time on the same inputs. The code constructs a node graph using Networkx and selects two random points. These random points are then passed to each algorithm in order, and their performance is evaluated. This process is embedded in a loop to determine performance across a varied set of inputs, so that particular characteristics of the graph are not deterrent to the performance of each algorithm. In each iteration of the loop, the input stops are chosen at random from the stop list. The performance and path length are recorded in a list at the end of each loop. This list is then averaged at the end of execution, so that performances over the full body of evaluations may be compared with one another. The path lengths are recorded so that optimality may be investigated, as if every algorithm truly finds the optimal paths then they will all agree on the length of said optimal path. These metrics are then recorded in a results table so that they may be easily compared with one another.

3.2 Algorithms

The algorithms used in this investigation are all of the same utility, namely finding optimal routes based on edge weights and distances within the graph itself.

3.2.1 Dijkstra's Algorithm

Dijkstra's shortest paths algorithm is the single-source shortest paths algorithm, and it utilizes the weight of each edge between nodes to evaluate the path being constructed. [Lew20] This algorithm is greedy in execution, as it overestimates distances between each node and the starting node, then visits each neighbor and records the shortest subpath to the neighbors. By iteratively finding optimal subpaths, the overall shortest path is constructed. The algorithm stores the unvisited neighbors of the source node S in a priority queue Q, then pops out the minimum weight neighbor in a process called relaxing the edge. This process runs in O(V log V + E log V), where V is the set of vertices and E is the set of edges. If each of the vertices in the graph have an outvalence of m, and the edge weight from the source vertex s to the target vertex t is n, then Dijkstra's algorithm would expect to relax edges on the order of m^n.[BAN17]

3.2.2 Dijkstra's Algorithm (Bidirectional)

The bidirectional implementation of Dijkstra's algorithm maintains the mechanics of the standard version, but instead utilizes a different stop condition in tandem with a split execution to cut down on the total number of neighbor nodes that are relaxed in each iteration. Dijkstra's standard algorithm begins at a source node and continues until a target node is reached, but in the case of the bidirectional version a search is begun from both the source node and target node. These searches then continue in opposite directions until they meet each other in the middle. With this edit to the algorithm, the expected number of edges to be relaxed is reduced to the order of $2m^{n/2}$.[BAN17]

3.2.3 A*

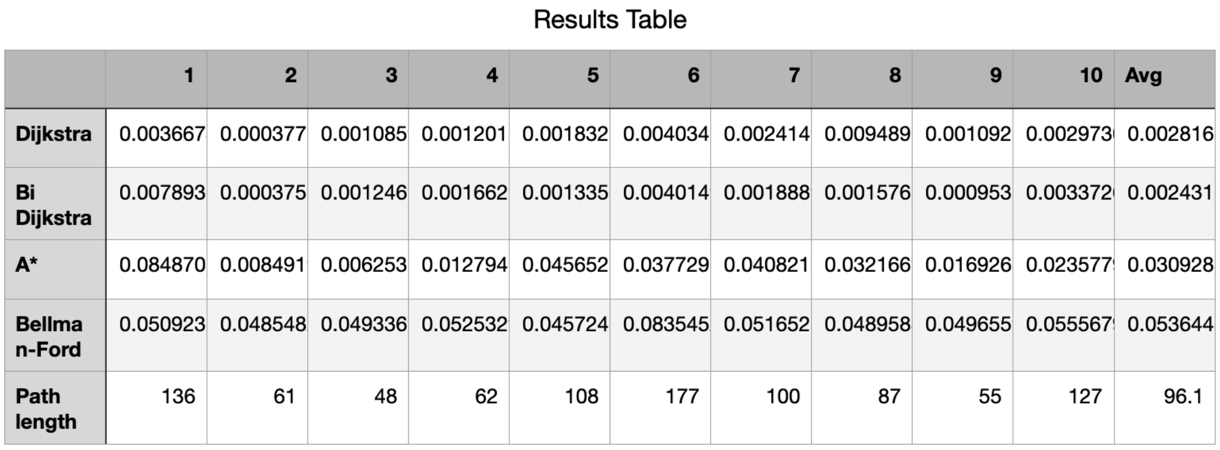
The A* algorithm is an informed search algorithm that uses a heuristic function to expand edges one at a time from the start node to the end node. [Dor] It aims to find a path with the lowest overall weight according to its heuristic, and maintains a tree of paths between the nodes in order to evaluate each path possible. Once the end node is reached, it returns the path found of least weight from this tree. A* often uses a priority queue to determine which edge to expand first, with the nodes being ordered according to the function f(n) = g(n) + h(n), where g(n) is the cost of the path from the start to that node, and h(n) being the heuristic function that estimates the cost of the path from that node to the goal. The worst case runtime of A* in a setting with unbounded memory is exponential in the depth of solution path and branching factor of successors per state, or $O(b^d)$.

3.2.4 Bellman-Ford Algorithm

The Bellman-Ford algorithm also utilizes edge relaxation, but in contrast to Dijkstra's algorithm that greedily selects the closest unvisited neighbor, this algorithm relaxes all of the neighboring edges. It then repeats this process and calculates distances with every iteration until every node in the graph has been visited and has correct distances calculated for each edge. This algorithm runs in $O(V * E)$ time, with V being the number of vertices and E the number of edges. [Kla]

3.3 Results

The results of this experiment are reported by code in each iteration of the main loop, and summarized in the results table. The times shown for each algorithm are reported in seconds, and the path lengths are the number of nodes in each path. The rightmost column shows the average time to complete the search for each algorithm, and the average path length for this run of the experiment. The columns are labelled by the run of the evaluation for which they're reporting times, and the rows are labelled by which algorithm is being evaluated.



Results Table

	1	2	3	4	5	6	7	8	9	10	Avg
Dijkstra	0.003667	0.000377	0.001085	0.001201	0.001832	0.004034	0.002414	0.009489	0.001092	0.002973	0.002816
Bi Dijkstra	0.007893	0.000375	0.001246	0.001662	0.001335	0.004014	0.001888	0.001576	0.000953	0.003372	0.002431
A*	0.084870	0.008491	0.006253	0.012794	0.045652	0.037729	0.040821	0.032166	0.016926	0.023577	0.030928
Bellman-Ford	0.050923	0.048548	0.049336	0.052532	0.045724	0.083545	0.051652	0.048958	0.049655	0.055567	0.053644
Path length	136	61	48	62	108	177	100	87	55	127	96.1

Times shown in seconds

4 Analysis

The best performing algorithm for this dataset was the bidirectional Dijkstra's algorithm, outperforming the other algorithms applied by significant margins aside from Dijkstra's algorithm itself. The standard variation of Dijkstra's algorithm was at each iteration very close to the bidirectional version in terms of time took to complete, with their average execution times being approximately 0.0004 seconds apart. The next most efficient algorithm was A*, with an average time of approximately 0.03 seconds to complete the search. The slowest algorithm to complete was the Bellman-Ford algorithm with an average execution time of approximately 0.05 seconds. There emerge two major sides of the resulting data, that being Dijkstra's algorithm and its bidirectional variant being separated from A* and Bellman-Ford by approximately .028 seconds on average.

An interesting trend emerges when viewing the results outside of the averages, however, as A* and Bellman-Ford trade places in terms of completion order for paths of different lengths. The shorter portion of paths (¡100 nodes) saw A* outperforming Bellman-Ford, occasionally by orders of magnitude. In contrast, for longer paths (¿100 nodes) the execution times for the two slower algorithms were much closer.

As expected from the average results, the difference between the two versions of Dijkstra's algorithm in terms of execution time is incredibly small, often times within .0001 seconds of each other.

These results are intriguing for a variety of reasons, most notably the performance of A*. One of the lesser performing algorithms in this experiment, A* is frequently heralded as a leader in speed and efficiency in terms of search algorithms, so to see it place 3rd overall feels in contradiction to this legacy. This performance is most likely due to the heuristic function used, which while admissible could perhaps have been replaced with one that yielded better results. For this experiment the heuristic used was the linear distance between the points, but there exists room for a more advantageous heuristic to have been developed. Another interesting facet was that Dijkstra's algorithm was so effective, as it is often introduced as a more basic search algorithm and there have been proposed many versions that report better performance. An unsurprising result is that of the Bellman-Ford algorithm, as relaxing every edge in a graph this large and dense is expensive in both time and space.

5 Conclusions

Public transit route planning is a perfect application of graph search theory and algorithms, representing a real-world situation that can be abstracted directly to a node graph. Planning these routes is of course a more nuanced practice in actuality, as many other factors go into planning where a route will provide service to a city, such as the surrounding neighborhood, intersections with other transit lines, and projected amount of utility for locations that may not directly path-optimal. With this in mind, a simplified version of this problem can be viewed directly by the node graph constructed in this experiment. This graph can then be used for comparing search algorithms, and as shown with varying results.

These results were rather direct, and the performance on such a large graph is a testament to how well the algorithms in question are conceptualized and developed. For this experiment, Dijkstra's algorithm utilizing bidirectional search was the most effective one applied to the problem with the fastest average searches. The least effective algorithm in this case was the Bellman-Ford algorithm with the longest searches on average. Between these were the standard Dijkstra algorithm and A* search.

5.1 Considerations for future experiments

Going forward, there are several areas where this experiment could be improved. Firstly, more algorithms could be implemented and tested. This project ended up removing the genetic algorithm originally conceived due to technical issues with the implementation of the node graph. To add to this, the initial research suggests large improvements in performance over Dijkstra's algorithm, so having a genetic algorithm present may yield even better results. Another aspect that could be improved for a later version is the implementation of A* with a better heuristic, in the hopes of seeing better performance overall. This experiment could also be applied to a larger variety of node graphs of varying sizes, to explore farther the limits of what can be achieved by these algorithms. To improve the problem itself, there could be more aspects added to each algorithm to take into account different types of edges that better model real-world circumstances, or more aspects of nodes themselves such as the type of stop they represent or how many other routes use that particular stop.

References

[BAN17] Alican Bozyigit, Gazihan Alankuş, and Efendi Nasibov. Public transport route planning: Modified dijkstra's algorithm. pages 502–505, 10 2017.

[BGGMEP14] Maria Angeles Sáez Blázquez, Sebastián García-Galán, José Enrique Munoz-Expósito, and Rocío Pérez Prado. *Genetic Algorithm for Scheduling Routes in Public Transport*, volume 233. Springer, Heidelberg, 2014.

[CFT02] Alberto Caprara, Matteo Fischetti, and Paolo Toth. Modeling and solving the train timetabling problem. *Operations Research*, 50(5):851–861, 2002.

[Dor] Alan Dorin. Search algorithms - a*.

[D13] Selim Dundar and İsmail Sahin. Train re-scheduling with genetic algorithms and artificial neural networks for single-track railways. *Transportation Research Part C: Emerging Technologies*, 27:1–15, 2013. Selected papers from the Seventh Triennial Symposium on Transportation Analysis (TRISTAN VII).

[EDF14] Cyrill O. Escolano, Elmer P. Dadios, and Alexis M. Fillone. An integrated and optimal scheduling of a public transport system in metro manila using genetic algorithm. In *2014 International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment and Management (HNICEM)*, pages 1–6, 2014.

[Kla] Andreas Klappenecker. The bellman-ford algorithm.

[Lew20] Rhyd Lewis. Algorithms for finding shortest paths in networks with vertex transfer penalties. *Algorithms*, 13(11), 2020.

[MMP14] Karan Mitra, Nishikant Mokashi, and Prasad Patil. A survey on routing algorithms for efficient and optimized railway scheduling. *Indian Journal of Applied Research*, 4(4), 2014.

[MSM$^+$21] Zahid Hasan M., Hossain S., Mehadi Hassan M., Chakma M., and Uddin M.S. A genetic algorithm-based optimal train schedule and route selection model. *Proceedings of International Joint Conference on Advances in Computational Intelligence*, 2021.

[Sen20] Raj Sengo. Bellman-ford single source path algorithm on gpu using cuda. 2020.

[SJM22] Tommaso Schettini, Ola Jabali, and Federico Malucelli. A benders decomposition algorithm for demand-driven metro scheduling. *Computers I& Operations Research*, 138:105598, 2022.