CSC111: Project Phase 2
TTC Route Planner for Toronto, Ontario

Anna Cho, Charles Wong, Grace Tian, Raymond Li

April 17, 2021

Problem Description and Project Goal

In an expansive metropolitan area such as Toronto, it can be difficult for transit users to navigate the bewildering labyrinth of bus, streetcar, and subway paths. Travelling from one destination to another in the least amount of time poses complications for many new and existing users, and subway maps are often insufficient when attempting to discern the most efficient route.

There are 75 TTC subway stations that are currently divided into four lines (Yonge-University, Bloor-Danforth, Scarborough, and Sheppard) ("TTC Future System Map", n.d.), as well as a streetcar network composed of ten routes ("Streetcars", n.d.). Additionally, the TTC had 159 bus routes in 2018, with an average daily ridership of 1.58 million on weekdays ("Operating Statistics", n.d.). In terms of ridership, the TTC is the third most heavily-used transit system in North America ("List of North American rapid transit systems by ridership", n.d.), which is why we decided to create an efficient transit planner for millions of potential users.

We chose this as our project because it can be useful for many people taking the TTC—particularly commuters or students like ourselves—and the transit system is especially suited for a graph data structure. Moreover, we were able to seamlessly integrate relevant computations in order to search for the most efficient path between two sets of coordinates (both of which will be inputted by the user using a point-and-click map).

Although the incorporation of transit schedules (hours of operation for each type of vehicle), temporary construction dates, and future transit lines would likely be outside the scope of this project, our project will still be functional for regular hours of use. The transit destination planner that we plan to build will make use of multiple routes (for all types of vehicles within the TTC) as well as arrival times, in order to generate the shortest path between two points based on the schedule of the given day of the week.

Project Goal: How do we create a system that uses TTC routes and arrival times in order to compute the quickest path between two points?

Dataset Description

We plan on using the TTC Routes and Schedules dataset from the City of Toronto's Open Data Portal ("TTC Routes and Schedules", n.d.), which provides data in a relational database format, following the General Transit Feed Specification ("GTFS Static Overview", n.d.).

The dataset consists of eight separate text files: agency, calendar, calendar_dates, routes, shapes, stop_times, stops, and trips. For this project, the most relevant files are as follows: stops, routes, and stop_times. Samples of these files are included below.

`stops.txt` *(in a csv format)*
This file includes stop IDs, stop names, latitude/longitude data, etc.

stop_id	stop_code	stop_name	stop_desc	stop_lat	stop_lon	zone_id	...
557	11989	EXHIBITION LOOP		43.635967	-79.416408		
558	13900	MARKHAM RD AT PASSMORE AVE NORTH SIDE		43.831141	-79.250611		
560	11331	MARTIN GROVE RD AT FINCH AVE WEST NORTH SIDE		43.737241	-79.591714		
⋮							

`trips.txt` *(in a csv format)*
This file includes route IDs, trip IDs, etc.

route_id	service_id	trip_id	trip_headsign	trip_short_name	direction_id	...
62459	1	41575454	EAST - 10 VAN HORNE towards VICTORIA PARK		0	
62459	1	41575456	WEST - 10 VAN HORNE towards DON MILLS STATION		1	
62459	1	41575476	WEST - 10 VAN HORNE towards DON MILLS STATION		1	
⋮						

`stop_times.txt` *(in a csv format)*
This file includes trip IDs, arrival times, departure times, etc.

trip_id	arrival_time	departure_time	stop_id	stop_sequence	stop_headsign	...
41636131	16:57:42	16:57:42	6298	19		
41636132	16:57:40	16:57:40	14163	1		
41636132	16:59:42	16:59:42	5633	2		
⋮						

We will represent the TTC routes as a directed graph, with the stops (encompassing bus, subway, and streetcar stops) as the vertices of the graph. The edges of the graph will represent routes connecting one stop to another.

Computational Overview

1. **Data Transformation, Filtering, and Aggregation**

   (a) The first step in our project was to transform the dataset (a collection of text files) into a format that was efficient and more easily usable by other parts of the program. We used SQLite through the Python library `sqlite3` in order to create a data interface, which can be seen in `data_interface.py`. Although we could have modelled our data transformation step after the methods shown in class — using .csv files and containing the necessary information in the graph — it would be a time-consuming process for larger datasets, such as the one we chose to use.

   (b) We used this data interface to load and transform data into a SQLite database file: the function `init_db` initializes the database into a file called `transit.db`, and utilizes basic SQL commands to create several different tables (e.g. `calendar`, `routes`, `shapes`, etc.) One important table is the `edges` tables (within the function `_compute_distances`), which is based off the `stop_times.txt` GTFS static file, and allows for faster lookup of edge information.

   (c) All in all, our data interface creates a plethora of endpoints and functions that package SQL queries into usable formats. To illustrate, we have a method called `get_stops` within the class `TransitQuery`, which functions to return all of the stops as a set. Likewise, we implemented the ability to get route information given a route ID, through the method `get_route_info` (also within the class `TransitQuery`). Another notable endpoint that we included provides the physical coordinates between two stops (using the `shapes.txt` GTFS file), which was necessary for our eventual visualization.

2. **Building Graphs From *TTC Routes and Schedules***

   (a) Our project uses data related to transit (e.g. stop IDs, arrival times, etc.) to represent the domain of transportation, and graphs play an integral role in this data representation because edges naturally occur between the vertices — the stops — of such a graph. We chose to build a graph from the *TTC Routes and Schedules* dataset because the most logical representation would involve one vertex for each stop of the transit system (whether it be a streetcar, bus, or subway stop) with edges denoting a connection between two given stops.

   (b) We used the same graph representation from class and Assignment 3. The `_Vertex` class in `graph.py` represents one stop of the transit system, and stores the stop ID of the vertex, the location of the vertex using a latitude-longitude tuple, and neighbours. We imported data from the graph using the `data_interface.py` file, as described above.

   (c) The `Graph` class in `graph.py` generates a directed graph (in the case that a pair of given stops does not have a two-way connection). Each edge is weighted in order to provide necessary information for the pathfinding algorithm, which will be described in detail in the following section. The `get_weight` method uses data from the `get_edge_data` method of `data_interface.py`, and stores the following information in each edge: `trip_id`, `time_dep` (the time of departure for the next vehicle that travels between the two stops), `time_arr` (the time of arrival), and `weight` (the travel time between the two stops).

3. **Computations and Algorithms**

   (a) The next step in our project was to generate the optimal path from one vertex — one stop — to another vertex. These paths were optimized for average schedule times in specific time blocks, for example morning rush hour, noon, evening rush hour, and midnight. Depending on the two stops chosen, there could be multiple paths, and our goal was to determine which path was the shortest (not in terms of the number of vertices traversed, but in terms of the time it takes to travel between each pair of vertices).

   (b) We found that the optimal way to complete this was through the use of a greedy algorithm: Dijkstra's algorithm, or the A* pathfinding algorithm ('A* search algorithm', n.d.). We settled on the A* pathfinding algorithm because it runs rapidly, helps to find the best first search, and includes the involvement of a heuristic ('A* Search', n.d.). The A* algorithm factors in the edge weights of the graph, as well as an implementer-specified heuristic in order to determine which nodes to pursue as a possible path. Our choice for a heuristic was simply the distance from a given node to the final destination, which would allow the A* algorithm to estimate the minimized cost (distance) between vertices.

   (c) In summary, A* takes in a starting vertex as well as a given vertex n, where the cost is denoted by sum of all edges from the starting point to n, and heuristic(n) is the estimated remaining distance from n to the final vertex. The optimal path contains the vertices for which these properties/functions are minimized.

   (d) We implemented the algorithm ourselves in the `pathfinding.py` file, using several functions from `util.py`. The function `a_star` follows the description of the A* algorithm we detailed above, and takes the time and day into account when determining the cheapest path. `a_star` calls the `h` heuristic function (which in turn calls the `distance` function from `util.py`) and `construct_path`. Our heuristic function `h` works by calculating the great-circle distance — the orthodromic distance between two points on the earth's surface — between the current node and the goal node, using the haversine formula ('Great-circle distance', n.d.). On the other hand, `construct_path` returns a reversed path that was constructed using `path_bin`.

(e) We have another function named `find_route`, which uses a start location, end location, and a time block, and calls the `a_star` function in order to compute the fastest route.

(f) In certain cases, the travel time between two stops may be faster if the user walks. In order to account for walking times, we computed the closest stop to the user using the haversine formula, and considered a set of start stops within 100 metres of the closest stop. After repeating the process for the end location, we were able to find a more realistic path (as opposed to directly using the closest stop). We took transfers into account by searching for stops within 50 metres of each node in A*, in order to consider switching to another bus (or switching between busses and subways).

4. **Visual Representation and Interactive Features**

   (a) For our visual representation, we decided to expand upon the Pygame library and use it to a greater extent than we did in class. In short, the program displays a map with waypoints where the user can select their start and end destination, and then display the optimal path for the user on the map. The map will be interactive: the user can select waypoints, scroll, and zoom in/out. To report the results of our computations, the output path will be displayed using a map format as well.

   (b) We first retrieved images of maps from OpenStreetMap, which we used as a base map. The class `image` in `image.py` stores the information of each of these images, and contains methods to translate between Pygame coordinates and latitude/longitude. Another file called `waypoint.py` contains the class `_Waypoint`, which we used to organize the start and stop points of a path. Finally, the class path in `path.py` stores information concerning each stop and trip that it traverses within the path.

   (c) We expanded upon our knowledge of Pygame by allowing it to function as the user interface: the user can scroll, zoom in, zoom out, and choose a day and time to receive a transit path output. The file `pygui.py` contains Pygame events (e.g. a `PygButton` class that calls an `on_click` method), a method to display text, a method to draw buttons, and more). Another way we used the library in a new way was by creating a dropdown menu (using the `PygDropdown` class), which allows the user to select a day of the week.

   (d) Once the path is generated between the two points, it is displayed on the map using Pygame. We included a sidebar for additional information about each trip contained within the path.

   Note: We added `'Too many function args (E1121)'` to the 'disabled' section of PyTA, as it was a necessary requirement for our GUI to function.

Instructions for Obtaining Data Sets and Running the Program

1. We have shared a .zip file (containing our datasets and images) with `csc111-2021-01@cs.toronto.edu`, that should be extracted into the project directory. For reference, the link is as follows: `https://bit.ly/3wZwNtW`

2. Running main.py will create the database file, generate the graph (this will take ∼30 seconds!), and open the map visualization.

3. The program can be interacted with using `pygame` window: you can scroll, pick a day/time, select two points on the map, and click 'Get Route'.

   Note: Long paths may take 10+ minutes to generate. For a quicker runtime, consider selecting start and end locations that are relatively close to each other (or near major roads/transit lines).
   We have provided two images with example outputs on **Page 7** (Intel i7 10th-generation / base clock speed 2.6GHz / 6 cores).

4. If you wish to get `logging` debug messages, uncomment the line `logging.basicConfig(level=logging.DEBUG)`, or change the configuration to another level (such as the INFO level).

Changes Between the Proposal and Final Submission

1. We initially stated that we would use the tkinter library to create a graphical user interface (GUI) for displaying our interactive map, with Pillow (PIL) to help display images. However, our TA mentioned that it may be easier to use Pygame for the visual and interactive component of our program. We settled on using Pygame for our final map representation because it was easier to use given the limited amount of time we had, yet it was sufficient for our purposes.

2. We originally planned on transforming our data from text files to .csv files. However, we realized that the process would be smoother and far more efficient if we used `sqlite3`, thus we changed our final project to include SQL (after inquiring on CampusWire first). This sped up the running time of our program, as we had a large amount of transit information that could be used more rapidly through the inclusion of databases.

3. We encountered an issue when we attempted to efficiently find distinct routes, as we had to compare various permutations of start/end stops around the user's clicks. We resolved this problem by utilizing the `multiprocessing` module, which allowed us to parallelize our A* algorithm and simultaneously run multiple permutations of A* ("multiprocessing — Process-based parallelism", n.d.). This module enables our program to spawn new child processes, which execute the A* algorithm on different combinations of start/end stops. The child processes then transmit information back to the parent process using a `multiprocessing.Queue` (thus notifying the user of the search's progress), then they automatically self-terminate.

4. The last change between our project proposal and our final submission was the way we handled our data dynamically, based on the time and day inputted by the user. We initially planned on leaving out the finer details of transit schedules (e.g. hours of operation for each vehicle), but we decided that dealing with times and dates was still within the manageable scope of the project. Moreover, changing our data and outputs depending on the day of the week would also render our project a great deal more practical for the user.

Discussion

Our initial project goal is as follows: *How do we create a system that uses TTC routes and arrival times in order to compute the quickest path between two points?*

We were able to successfully create the system that we envisioned, following the computational plan that we included in our proposal. By and large, the A* algorithm and Pygame visualization was successful, and the edges of the directed graph (that was built using our data interface) were reliable when it came to pathfinding. The results of our computational exploration were effective when it came to finding the most efficient path, which was instrumental when it came to answering our project question. Put succinctly, we used TTC routes and arrival times to generate vertices and weighted edges, which were then traversed by a greedy algorithm in order to compute the quickest path.

One new consideration that we did not mention in the project proposal was the inclusion of walking times. In certain cases, we realized that the time taken for a transit user to travel from one stop to another (through a subway/bus/streetcar) was, in fact, shorter than the time it took for them to walk. Although this situation did not happen for every pair of start and end destinations, we recognized that this feature would paint a more accurate representation of the shortest path between two points, and it would realistically be of greater use to the user of our program.

We encountered several limitations and obstacles with our datasets, algorithms, and libraries throughout the course of this project. One such example was occurred during our data transformation step, as the given text file data had an unusual format that we did not expect to encounter. In particular, the `stop_times` file contained a data column titled `stop_sequence`, which denotes the ordering of rows in the database. This specific column proved to be a mild hurdle because it slowed down our queries, and we were thus forced to transform the data into a more usable format. In essence, our challenge was in creating a data format that was not overly complex (so as to not have an excessive running time), yet useful enough to generate a comprehensive overview of the information that mattered.

Another obstacle that we encountered was choosing how to represent our edge weights for the graph, and what information we wanted to store within the edge. Since we had a number of stops and edges, it was necessary to strike a balance between memory usage and simple queries for information. Additionally, we debated on the value of the edge weight itself: we considered making each weight the distance between two stops, or perhaps the time between two stops. We eventually settled on creating dynamic edges through database queries (since the program is dependent on the time of day, as well as the day of the week), with edge weights computed using a great-circle distance formula.

Our final challenge occurred when we were working on the visualization of our program, as we hoped to display our path results to the user on a map. We ended up resolving this obstacle by piecing together a number of map images (using OpenStreetMap) in order to create a full map with decent resolution. We did not have the time to figure out both tkinter and Pillow (as we originally intended on doing as per our proposal), thus we used our base map in conjunction with Pygame in order to display the most efficient path.

There are a number of possible steps that we could implement for further exploration in the future:

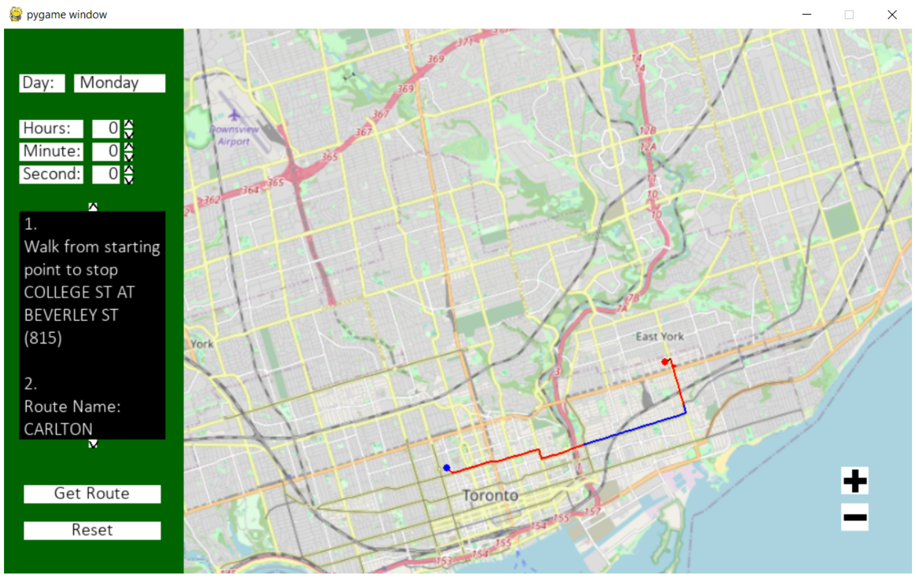
1. During our project, we attempted to include a secondary dataset containing information about historic and future subway and streetcar delays. The user would then be able to pick a date in the near future (or in the past) and receive the ideal route for that *specific* day, given the number of delays and closures (temporary construction dates) that the TTC has. This would have changed our graph slightly, as certain timings would change (and closed stops would have to be ignored in our set of vertices). Although we did not have the time to implement this, it is a possible future step for our project.

2. In a similar vein, we could include plans for future transit lines, by factoring in those stops as our vertices. However, this next step would be of lesser priority given the amount of time it takes for Toronto to decide and act upon transit changes.

3. It is also very possible for us to expand the geographic area that our program could cover. Given a dataset for another city with an expansive transit system (e.g. New York, Tokyo, or any other major metropolitan area), we could expand our program to cover the transit paths of those cities as well.

4. Our program runs quite slowly, and there are multiple methods through which we could optimize the A* algorithm. One such method is to incorporate 'interchange' stops, where we would store intermediary paths between such stops (Delling et al. 2). Once a user starts a query, we would merely need to find paths to these interchange stops, thereby speeding up our algorithm.

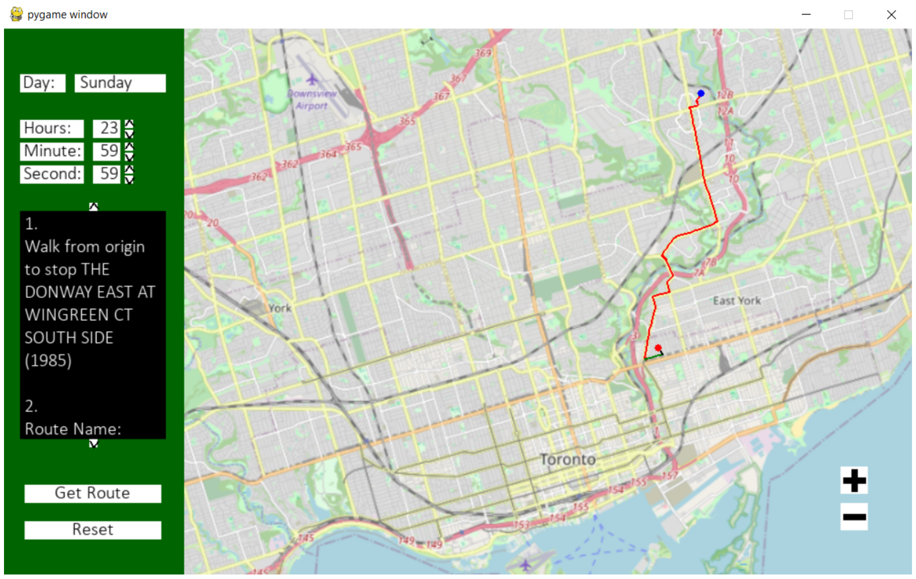
In summary, we were able to implement the majority of the changes we wished to include, and our computations and visualizations resulted in a path that was in line with our project goal. Although there are still a number of features that could ameliorate the quality of our project, the base functionality is sufficient, and our program fulfills the expectations that we had laid out at the outset.



1.
Walk from starting point to stop COLLEGE ST AT BEVERLEY ST (815)

2.
Route Name: CARLTON

Figure 1: Example 1 (2-3 min. runtime)



1.
Walk from origin to stop THE DONWAY EAST AT WINGREEN CT SOUTH SIDE (1985)

2.
Route Name:

Figure 2: Example 2 (3-4 min. runtime)

References

A Search.* (n.d.). Brilliant. https://brilliant.org/wiki/a-star-search/

A Search Algorithm.* (2002, October 7). Wikipedia. Retrieved April 15, 2021, from https://en.wikipedia.org/wiki/A*_search_algorithm

Delling, Daniel, et al. *Engineering Route Planning Algorithms*.* Universitat Karlsruhe (TH), i11www.iti.kit.edu/extra/publications/dssw-erpa-09.pdf. Accessed 17 Apr. 2021.

Great-circle distance. (2003, November 26). Wikipedia. Retrieved April 15, 2021, from https://en.wikipedia.org/wiki/Great-circle_distance

GTFS Static Overview. (n.d.). Google Developers. https://developers.google.com/transit/gtfs

List of North American rapid transit systems by ridership. (2008, March 26). Wikipedia. Retrieved March 13, 2021, from https://en.wikipedia.org/wiki/List_of_North_American_rapid_transit_systems_by_ridership

multiprocessing — Process-based parallelism. (n.d.). Python Software Foundation. https://docs.python.org/3/library/multiprocessing.html

Operating Statistics. (n.d.). Toronto Transit Commission. https://www.ttc.ca/About_the_TTC/Operating_Statistics/index.jsp

Reference. (n.d.). Google Developers. https://developers.google.com/transit/gtfs/reference

TTC future system map. (n.d.). Toronto Transit Commission. https://www.ttc.ca/Spadina/About_the_Project/TTC_System_Map.jsp

TTC Routes and Schedules. (n.d.). City of Toronto. https://open.toronto.ca/dataset/ttc-routes-and-schedules/

TTC streetcars. (n.d.). Toronto Transit Commission. https://www.ttc.ca/Routes/Streetcars.jsp