

CSC111 Project Report: The Bus Stop Creator

Andy Wang, Varun Pillai, Ling Ai, Daniel Liu

September 23, 2021

Problem Description and Research Question

Cars and traffic have contributed a lot to global warming and the emission of greenhouse gasses. In 2014, cars and trucks made up about one-fifth of all emissions in the US, spewing out 24 gallons of carbon dioxide for every gallon of gas (Union of Concerned Scientists, 2014). According to the Center for Climate and Energy Solutions, cars and trucks make up half of all carbon dioxide emissions in the US (C2ES).

Luckily for us, there are many solutions to combat all of these greenhouse gasses. One could choose to ride a bike, or walk instead of driving a car. One can carpool with others instead, or use electric cars that don't require gasoline. One very popular solution is to opt for public transport, like subways and public buses. According to the C2ES, if one's commute is a 20-mile round trip, then switching to public transportation can lower one's carbon footprint by 4800 pounds per year (C2ES). An appropriate implementation of the public transportation may reduce pollution, but also possibly reduce noise and congestion caused by individual traffic.

Clearly, bus transportation is an important thing. It is imperative for cities and neighbourhoods to have some sort of effective and efficient public transport, so that the public transportation is a good alternative enough for residents to voluntarily decide to use public transportation. That is why our computational question is: **given the layout of a city and number of buses, where do we place bus stops, and how do we route each bus to make the most efficient transport system possible?** And by most efficient, we want to create a transportation system such that each bus completes its route in the quickest and most eco-friendly way, while also covering as much of the city's residents who may use public transportation as possible.

Computational Overview

Graphs

We used graphs for this project since there is a natural parallel to be drawn between road networks and the graph abstract data type. We implemented classes such as `_Place` to represent places such as buildings, with subclasses such as `_BusStop` and `_Intersection` to represent bus stops and street intersections respectively. All these places described act as vertices within the `City` class, which is an implementation of the graph abstract data type. The edges of `City` class are represented by streets used to connect bus stops, intersection, buildings, etc. Due to the overall complexity of our project, there are a lot of graph related methods for the `City` class. Generally speaking, we have methods that add and remove places from the city, add and remove streets between places, and access and format private instance attributes.

Pygame

The program reports the results of the various computations in the form of an interactive `pygame` window. Using the `run_visualization` function in `main.py`, a user can create their own city by adding/deleting places, intersections, and streets. By the click of a button (the controls are detailed later in this report), the program will use KMeans clustering and Dijkstra's algorithm to not only place bus stops, but also route them in an optimized way, and display them to the screen. The actual program itself is implemented with an event loop which continuously runs until the program is closed by the user. Whenever the user interacts with the program, some code in an if statement will be executed which may alter the state of the program. These changes are then displayed by various calls to the `draw` method from the `Drawable` abstract class. To do this, we made both `_Place` and `_City` subclasses of the `Drawable` abstract class.

File I/O

Our program does not use datasets, but still uses `.txt` files to load and save cities. Each city has a map file, which encodes data about its places, intersections, and streets, and a bus stop file, which encodes data about its bus stops, bus routes, and "inertia".

Each line in a map file encodes either a place, an intersection, or a street. For example:

```
place 111 137
intersection 250 250
place 700 420
111 137 250 250
250 250 700 420
```

This encodes a city with a place with coordinates (111, 137), an intersection at (250, 250), another place at (700, 420). The last two lines encode streets, one being a street connecting the place at (111, 137) with the intersection at (250, 250), and the other one being a street connecting the intersection at (250, 250) with the place at (700, 420).

Likewise, each line in a bus file encodes either a bus stop, a bus route, or inertia. For example:

```
inertia -1
bus_stop 572 372
bus_stop 324 363
bus_stop 420 420
572 372 324 363 111 137 420 420 572 372
```

This encodes a bus stop with coordinates (572, 372), another at (324, 363), and another at (420, 420). The last line encodes a bus route starting at the first bus stop, then goes to the second, then goes to the *place* at coordinates (111, 137) (that place must exist in the corresponding map file), then goes to the third bus stop.

The first line represents the "inertia" of all the bus stops. Inertia is calculated as the sum of the squares of the distance each place is from that place's nearest bus stop, so essentially it's the "variance" of all the bus stops, which means the lower the inertia, the more satisfied the people are. An inertia of -1 means inertia has not yet been calculated.

Of course, it's not practical to have an inertia of zero - it would not be very cost efficient to have a bus stop at every single place in the city, so our algorithm aims to keep inertia low while having a realistic number of bus stops. This is helpful for the KMeans algorithm used by our program.

Path Finding Algorithms

This program also includes two path finding algorithms which find the shortest path between two places in the city. The first algorithm we implemented is called **Dijkstra's Algorithm**. It is a path finding algorithm which finds the shortest path between two vertices on a positively weighted graph (Alby et al). Since the weights on our graph are the distances between two adjacent vertices, it is the perfect algorithm for our needs.

In our implementation of Dijkstra, we started by initializing two sets to keep track of vertices, aptly named **visited** and **unvisited**. We also have a dictionary called **distances** where the keys are the unvisited places and the values are their respective distances from the starting node. Finally, we have a dictionary called **predecessor** where the keys are places and the values are previous places that connect to the key in a path. To actually find the shortest path, we have a while loop that iterates as long as the target destination has not been visited yet. With each iteration of the loop, we find the place with the shortest distance and set that as our current vertex of interest. However, if the current vertex's distance from the start is `float("inf")` then we know that there cannot possibly exist a path between the start vertex and the target vertex so we exit early.

If the distance is not infinity, we loop through every neighbour of the current vertex and compare the distance they are from the start to the distance a path which passes through our current vertex to our neighbour would take. If our path is shorter, then we change `distances[neighbour]` to be equal to the distance of this new shorter path. We also add our current vertex as the predecessor to our neighbour in this path. This cycle continues until we reach

the end in which case we have found the shortest path, or we exit early due to finding no connected path.

The second algorithm we implemented is called **A*** (pronounced 'A star') is a path finding algorithm which also finds the shortest path between two vertices. It can be seen as an extension to Dijkstra's algorithm since it follows the same steps except when calculating the 'distance'/cost between adjacent vertices. The only difference is that it uses the value of a heuristic function to determine which nodes are better instead of traversing over every node (Patel, 2014). For A* to find the shortest path, the heuristic must not overestimate the remaining distance to the end. With a city graph, there is no particular rule to how streets and places must be placed (compared say to a grid-based graph). As a such, basic heuristic functions like Euclidean distance or Manhattan distance may overestimate the remaining distance (Patel, 2020). There are certainly custom heuristic functions out there that provide far better estimates but they are way beyond the scope of this project and course. Since the heuristics we use may overestimate the remaining distance, it cannot guarantee the shortest path. However, in theory it is much quicker than Dijkstra since it doesn't explore unnecessary vertices.

Bus Stop Placement

As the name suggests, the main focus of our project is determining where to place bus stop given some distribution of places. To accomplish this, we used **K-Means Clustering** which is an algorithm that finds central points which similar data points cluster around (Piech, 2012). Since K-Means Clustering is most commonly implemented as a machine learning algorithm, we used an external library called **scikit-learn** for their K-Means implementation. Their K-Means implementation called `sklearn.cluster.KMeans` has many different arguments we can enter to customize how the central points or centroids are generated (Pedregosa et al, 2011a). However, we stuck with a basic configuration of `init='k-means++'` since we just want to know where the bus stops will be placed.

We also found that `sklearn.cluster.KMeans` did not accept regular Python lists as input, so we had to import **pandas** for their DataFrame data structure. After converting our set of tuples (set of coordinates) into an appropriate DataFrame, we ran the K-Means algorithm on our data (pandas, 2021). Once we got the DataFrame of bus stops from the K-Means algorithm, we then converted it into a list of tuples for use later. With our newly calculated bus stop coordinates, we then project (using vectors) them onto the closest street. This is to ensure the bus stop is placed on an existing street. We then connect the endpoints of the original street to the projected bus stop and remove the old street between the two endpoints. If the generated bus stop is on top of a place, it replaces that place. This is repeated for all the generated bus stops.

You might be thinking: "how many bus stops do we choose?" and we have an answer for that. We can decide the amount of bus stops by using inertia. As said, inertia is "the sum of the squares of the distance each place is from that place's nearest bus stop".

For every possible value of k (the k value indicates the amount of bus stops), `sklearn.cluster.Kmeans.fitpredict()` will find a configuration (in terms of clusters and corresponding centroids) that indicates the smallest possible inertia. For every increase of the k value, the smallest possible inertia will decrease by a certain amount; let's call this the variation. The general idea is that if for an increase of the k value, the variation of inertia is very small (in other words if we build one more bus stop, the inertia does not change very much), then this increase of k value will not be worth it (as at this point building one more station would cost money, but the trade off in terms of increasing consumer convenience of walking to station is not large). So we have to find a point where there is a large change (decrease) in variation of inertia, or aka the local max of change of variation of inertia. In mathematical terms, we want to find the local max of the 2nd derivative of the graph plotting inertia against k . The k value at that point is the optimal amount of bus stops to place on the map (Alade, 2018).

Also, there's an interesting detail about the `sklearn.cluster.Kmeans.fitpredict()` function: it predicts the absolute centroids corresponding to each cluster of places: these centroids may not lie on the streets. Thus, the computational plan is to project the centroids to the nearest spot on the nearest street. In that case, the inertia will be changed from the one calculated automatically by `sklearn.cluster.Kmeans`, and thus we also recalculated the inertia based on the projected centroids.

Bus Stop Routing

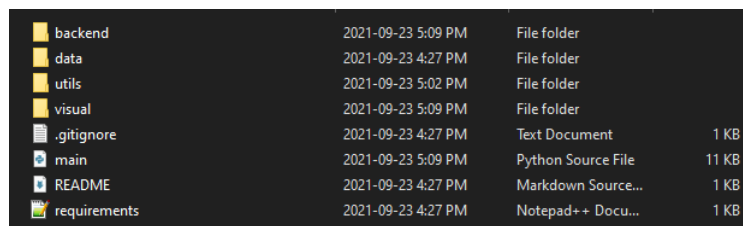
For the bus routes generation, we considered many variables that would construct a model, but finally decided to implement the most important variable of them all: consumer demand. Specifically, we decided to quantify the demand of consumers as passenger flow; say, if there are 250 person who wants to travel from place (vertex) A to place (vertex) B, then the path flow from A to B is 250. Notice that there is a different path flow from B to A. Every pair of vertex in the map will have such an attribute added to them, and thus the new class `PlacePair` is created; we could also call this the demand pair as it shows the demand of consumers in a pair of vertexes.

Since this project does not use datasets, functions are implemented to generate realistic model of demands of a city; this is why we have the new class `ModelCity`. In order to generate a realistic model city (with demands), the population density for each place is also needed, thus we have a new class: `ComplicatedPlace`. The reason the population density of places is needed is because the passenger flow between a high density place and low density place is special; namely, it is assumed that this flow is partly proportional to the high density place's population (because large cities and city centers attract more people to come). Every modelled city also assumes another thing: the flow from vertex A to vertex B should be about the same to the flow from vertex B to vertex A on average; people who go to work have to return home, people who went out to hike/play has to come back, these boomerang flow should make up the majority of the total flow from vertex A to B.

In order to find the optimal route, an algorithm is used that prioritizes large demands. This means that the initial bus routes recorded are first generated using the `PlacePairs` that have the highest average path flow. Then we go down the list of `PlacePairs` (demand pairs) with lower average path flow (demand), generate bus routes based on these `PlacePairs` and record them, and stop when all bus stops are already included in recorded bus routes. Then, we merge the bus routes, prioritising the initial bus routes (we merge bus routes corresponding to `PlacePair` with less path flow INTO the initial bus routes, which indicate `PlacePairs` with the highest path flows). The merged set of bus routes are the final bus routes generated.

Instructions for Running the Program

First create a folder for this program. Inside this folder should be four sub folders named `visual`, `backend`, `data`, and `utils`. Place `drawing.py` inside `visual`. Place `city.py`, `place.py`, and `route.py` inside `backend`. Place the `.txt` files inside `data`. Finally place `main.py` in the outer folder. The file format should look something like this:

A screenshot of a file explorer window showing the project structure. It lists folders and files with their creation dates and sizes.

backend	2021-09-23 5:09 PM	File folder	
data	2021-09-23 4:27 PM	File folder	
utils	2021-09-23 5:02 PM	File folder	
visual	2021-09-23 5:09 PM	File folder	
.gitignore	2021-09-23 4:27 PM	Text Document	1 KB
main	2021-09-23 5:09 PM	Python Source File	11 KB
README	2021-09-23 4:27 PM	Markdown Source...	1 KB
requirements	2021-09-23 4:27 PM	Notepad++ Docu...	1 KB

To run this program, simply run `main.py` in the python console. If there are valid `.txt` files inside the data folder, the program will automatically create the city for you. Otherwise you will start with an empty green canvas to play around on.

The user can interact with the city visualization to perform the following changes by pressing the following keys on the `pygame` window:

1. **Left mouse:** Click to place a normal place at the position of the cursor (red square)
2. **I + Left mouse:** Click to place a street intersection at the position of the cursor (grey circle)
3. **LSHIFT + Left mouse:** Click two places (either street intersections and/or normal places) to connect them with a road (grey line)
4. **CTRL + Left mouse:** Click to delete the street or place that the cursor is currently on
5. **S + Left mouse:** Click two places to see the shortest path between them using the Dijkstra's Algorithm. (pink line) **Note:** the two places must be connected by street(s).

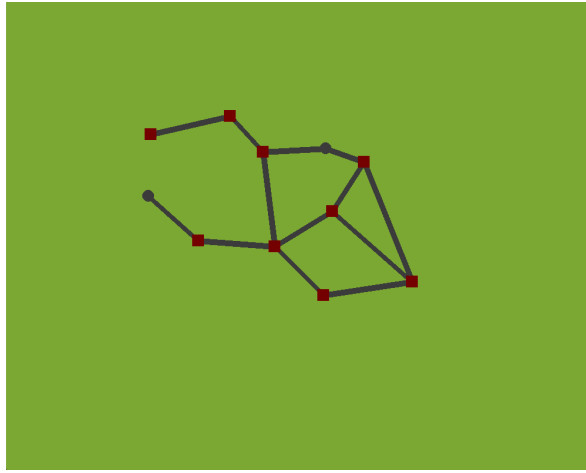


Figure 1: The `pygame` window visualizing the preset city when `run_visualization` is called

6. **D + Left mouse:** Click two places to see the shortest path between them using the A* Algorithm. (pink line) **Note:** the two places must be connected by street(s).
7. **B + 1:** Press to generate and place down bus stops (yellow squares)
8. **B + 2:** Press to generate and place down bus routes (the bus routes overlap sometimes, save and see all routes in `bus_save.txt`)
9. **CTRL + S:** Save the current city layout to a txt file
10. **Q:** Quit the program

Changes to the Project Plan

While we did keep to our original plan, the scope of our project has grown quite a bit. Instead of letting the user choose the number of bus stops they want to place, the program automatically selects the optimal number of bus stops for the city. We then made the route selection for bus stop far more advanced than anything we initially thought of. It now automatically selects bus stops for the route since without the need for user input. We also added the option for users to see the shortest path between two places in the city and have that path highlighted. Finally, we changed the bus stop placement from the closest place to the closest projection to make things slightly more realistic. Despite all of our new additions, there were a couple things we did not do from our proposal. We could not make any preset cities based off of real cities since it was too ambitious and time consuming. We also did not run any bus stop simulations as there was not enough time to formulate a working idea.

Discussion

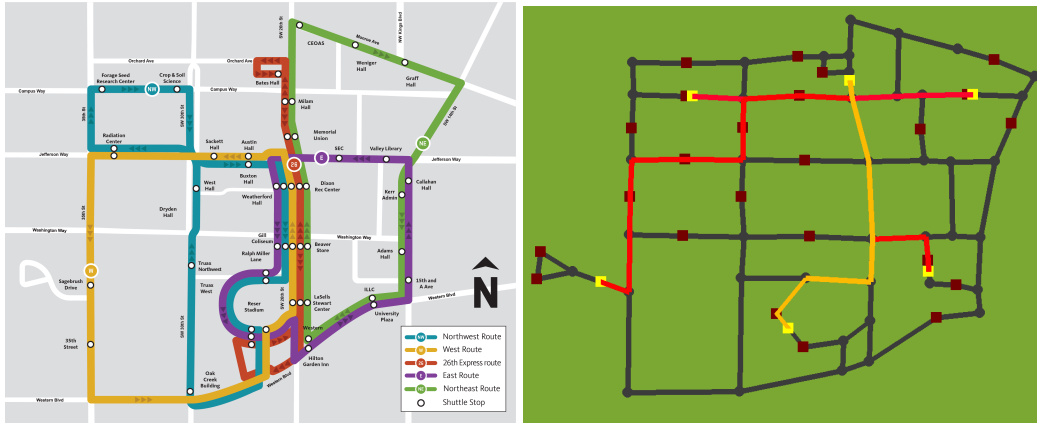
There are many things to discuss about this project, including how we set up the framework of the code, the algorithms we decided to use, and the algorithms we designed. First of all, the first model we used was that we represented real life distance and space as coordinates. This is realistically achievable by using Mercator projection, in which the functions are actually implemented in `utility_functions.py`. However, 2d coordinates cannot perfectly represent a curved surface and there are bound to be some distortions if we translate the corresponding coordinates to latitude and longitude. This distortion is acceptable when the map is small.

After creating our own city layouts using the `pygame` window, we were able to visualize the answers to our research questions. Using K-means Clustering, we were able to locate where to place the bus stops, given a simple city layout of buildings, streets and street intersections. As for the number of bus stops we planned to generate given a layout, we used the statistics tool: inertia, the sum of the squares of the distance each place is from that place's nearest bus stop. Using Dijkstra's Algorithm and A* Algorithm, we were able to generate the shortest path between two bus stops, and also by controlling variables such as consumer demand of buses, we were able to create a more

realistic and efficient bus routing system. However, there are some problems with how we auto generated the bus stops via the k means algorithm. The k means is highly efficient in predicting a centroid that is close (to the point that it is optimal) to every point in its cluster; however, we projected the centroid to a nearby street, which means that the solution (position of bus stop) might not be optimal in the end. We could instead use other algorithms related to geometry that finds the intersection of the radius (radius of effect) of each demand point (place), but that involves other assumptions and complications.

Obviously, the model we have created was a very simplified version of how real world bus stops work. It does not take into account many factors which influence real world bus routing such as differences between bus (for example, bus speed), frequency of bus employment, operational cost of maintaining bus in action, fixed cost like buying bus, capacity of buses, real life traffic situation and rush hours, etc. A model that includes these variables are too complicated to be computed by us as we not only lack knowledge in computation but also knowledge optimization. The model that we designed only ever considered consumer demand, but in real life, costs of bus system is just as important; in many cases, models of public transportation aims to minimize [cost + passenger travel time].

When we tried to recreate real street layouts and generated bus stops, we noticed that our program's bus stops weren't very accurate. The routes in real life showed more variability than those of the program. This is most likely due to the many factors in the previous paragraph that we did not consider in our simplified program.



As previously stated in this report, our program has both Dijkstra's Algorithm and A* for finding the shortest path. Our implementation of Dijkstra guarantees the shortest path but it will visit every vertex up to that point. In contrast, A* uses a heuristic to guide it towards the target vertex which reduces unnecessary visits. Unfortunately due to inaccuracies in our heuristic, A* does not always find the absolute shortest route; although in testing, A* tend to find paths that are decently close, if not equal to, the absolute shortest path found by Dijkstra's Algorithm. Perhaps in the future we could investigate more into custom heuristic functions so that A* always find the shortest path between two places in the city. That way we can effectively utilize the speed and efficiency of A*.

As for now, our model is far too straightforward to be able to realistically simulate bus routes, or efficiently model a city's bus system. In the future, our program could introduce more complexities such as traffic density, scheduling of buses, operational/fixed costs, bus capacities, prioritized bus-stops indicated by number of consumers using them, and more. The scope of mathematics required for a such a complex algorithm is beyond the scope of which we have learnt, so tackling this problem will require an in depth knowledge into theories of optimization and modelling in computer science. We could also introduce complicated street routes by allowing u-turns, roundabouts, expressways and other forms of road objects to become a part of the program and help simulate a more realistic bus routing system to accomplish our program's goals, as well as using real life latitude longitude coordinate system.

References

- Alby, T., et al. (n.d.) *Dijkstra's Shortest Path Algorithm* Brilliant. <https://brilliant.org/wiki/dijkstras-short-path-finder/>
- Alade, T. (2018). *Tutorial: How to determine the optimal number of clusters for k-means clustering*. Cambridge Spark. <https://blog.cambridgespark.com/how-to-determine-the-optimal-number-of-clusters-for-k-means-clustering-14f27070048f>
- Center for Climate and Energy Solutions. (n.d.). *Reducing your transportation footprint*. Center for Climate and Energy Solutions. <https://www.c2es.org/content/reducing-your-transportation-footprint/>
- pandas.(2021). *pandas.DataFrame*. pandas. <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>
- Pedregosa., et al. (2011). *sklearn.cluster.KMeans*. Scikit-learn: Machine Learning in Python. <https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans>
- Patel, A. (2014). *Introduction to the A* Algorithm*. Red Blob Games. <https://www.redblobgames.com/pathfinding/a-star/introduction.html>
- Patel, A. (2020). *Heuristics*. Red Blob Games. <https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>
- Piech, C. (2012). *K Means*. Stanford CS221. <https://stanford.edu/~cpiech/cs221/handouts/kmeans.html>
- Union of Concerned Scientists. (2014, July 18). *Car emissions and global warming*. Union of Concerned Scientists. <https://www.ucsusa.org/resources/car-emissions-global-warming>