> How can a tourist best spend their day out?

> ⓘ **Info**
>
> I've been finding it hard to plan trips with my friends, especially when everybody lives all over the city and we would all like to travel together. This SAT project aims to model the Victorian public transport network and its proximity to friends' houses, factoring in data about each individual to find the most efficient and effective traversals and pathways for us travelling to locations around Victoria.

I will start and end my day at my house, picking up all my friends along the way. The algorithm will find the quickest route to go to all my friends' houses, go to our desired location(s), and drop them all off before I go back to my own house. It will then return to me the traversal path, the time taken, and my cost for transport throughout the day.

Information to Consider

The following is key information to consider when modelling the real life problem. This will be done by representing the problem with an undirected network/graph, as all public transport methods go both ways, just at different times depending on the transport method.

Node Representation

Nodes represent key landmarks such as train stations, bus stops or a tourist attraction.

Edge Representation

Edges represent a route (train, bus, tram, walking, etc) from one location to another

Weight Representation

The edge weights will represent:

- the time taken to travel from one house to the other
- the financial cost of the route, with buses being more expensive than trains, which are more expensive than walking, etc.
  These can be interchanged to prioritise the certain attribute, such as time or money being of higher importance in the algorithm.

Additional Information Modelled Outside Graph

The following would be modelled as dictionaries:

- The arrival time/timetable of buses and trains
- The cost of changing lines
- Attributes of each friend, such as name, home, the time they wake up, the amount of time they take to get ready, and who is friends with whom or to what degree.
- Proximity to all friends' houses (by walking), which would be a dictionary for each node separately.
  This information could be used to add further complications to make the model reflect real life more closely, such as different friends being ready earlier than others or requiring a certain number of "close friends" (by threshold) to be within the travel party at all times.

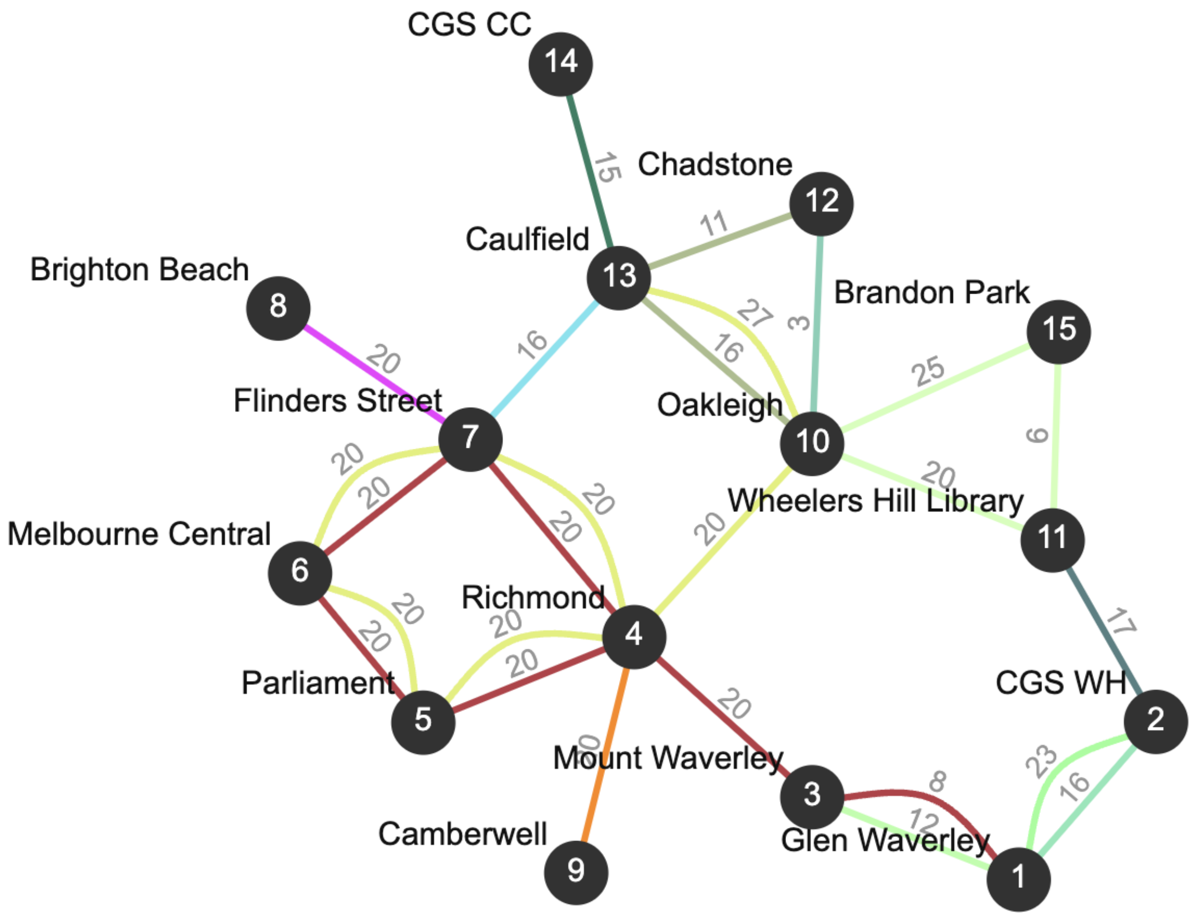
Abstract Data Types

I have selected a number of stations, bus stops and locations which I feel are relevant to my friend group.

Property	Stored as	Notes
Key Landmarks	Node	

Property	Stored as	Notes
Landmark Name	Node Attribute	
Route	Edge	
Route Name	Edge Attribute	
Transport Method/Line	Edge Colour	
Time or Cost	Edge Weight	These can be interchanged to prioritise different aspects. Distance is more relevant than time, but cost may be important as well.
Time/Cost of Changing Lines	Node attribute "interchange_cost" & "interchange_time"	
Train and Bus Timetable	Dictionary: Dict<String: Array<Dict<String: Int or String>>>	Keys would be each line (bus or train), and the values would be arrays of dictionaries with what node they are at, arrival times and departure times.
Attributes of Each Friend	Dictionary: Dict<String: Dynamic>	This will be a json style nested dictionary that has various attributes about each friend, such as waking up time, other close friends and other relevant information
Proximity to Friends' Houses	Node Attribute: Dict<String: Float>	Proximity of all houses as an attribute for each node, which has keys as friends' names and values as the distance or time to their house

Possible Graph



Signatures

Function Name	Signature
addLandmark	[name, interchange_cost, friend_proximity] -> node
addRoute	[start_node, end_node, travel_method, time, cost] -> edge
findShortestPath	[start_node, end_node] -> integer, array
addFriend	[name, wake_time, close_friends] -> dictionary

Algorithm Selection

While simplifying my problem, I found that starting and ending my day at my house while picking up all my friends along the way is simply an applied version of finding the shortest hamiltonian circuit. In other words, the shortest cost circuit that will visit every node.

While researching into how to solve this, I found that this was a classic example of the travelling salesman problem, which turns out to be an NP-hard problem. This means that there currently exists no exact solution to the problem in polynomial time, and the best I can currently do is the Held–Karp algorithm, which has a time complexity of $O(n^2 2^n)$ which is not ideal at all in terms of efficiency, but will have to be sufficient for the use cases of this project.

Held-Karp algorithm

The Held-Karp algorithm is a method for finding the exact shortest hamiltonian circuit in the exponential time complexity of $O(n^2 2^n)$, which is much better than if we to brute force it, which would have a complexity of $O(n!)$.

It works by utilising the fact the following principle.

Let $A =$ starting vertex
Let $B =$ ending vertex
Let $S = \{P, Q, R\}$ or any other vertices to be visited along the way.
Let $C \in S$

We $\therefore$ know that $\text{Cost}_{\text{min}}\ A \rightarrow B \text{ whilst visiting all nodes in S}$ = $\min(\text{Cost } A \rightarrow C \text{ visiting everything else in S} + d_{CB})$.
Put more simply, we can find the smallest cost hamiltonian path by gradually building larger and larger subpaths from the minimum cost to the next node in S, using dynamic programming to combine the subpaths to form the larger hamiltonian path.

This logic leads to the following pseudocode:

```
function held_karp (
        start: node,
        end: node,
        visit: set<node>
):
    if visit.size = 0:
        return dist(start, end)
    else:
        min = infinity
        For node C in set S:
            sub_path = held_carp(start, C, (set \ C))
            cost = sub_path + dist(C, end)
            if cost < min:
                min = cost
        return min
end function
```

After being implemented in Python (with a slight modification to return the path as well), this pseudocode looks like this:

```python
def held_karp(start, end, visit):
    if type(visit) is not set:
        print("Error: visit must be a set of nodes")
        return {'cost': float('inf'), 'path': None}
    if len(visit) == 0:
        return {'cost': dist(start, end), 'path': [start, end]}
    else:
        minimum = {'cost': float('inf')}
        for rand_node in visit:
            sub_path = held_karp(start, rand_node, visit.difference({rand_node}))
            cost = dist(rand_node, end) + sub_path['cost']
            if cost < minimum['cost']:
                minimum = {'cost': cost, 'path': sub_path['path'] + [end]}
        return minimum
```

The problem with this implementation is that it currently only works with connected graphs, where the distance between any two given nodes will not be infinity.
This becomes clear if we try and find the cost of going from Oakleigh to Melbourne Central while visiting Caulfield along the way. The pseudocode would choose Caulfield as the value for C, as it is the only node in the set. The issue is at line `12`, as the algorithm would try and get the distance between Caulfield and Melbourne Central, but as there is no edge between these two nodes, it will return ∞.

This can be solved by using Dijkstra's Algorithm, instead of the `dist` function, which will instead find the shortest path (and $\therefore$ distance) between any two given nodes.

After this modification, our hybrid algorithm works great!

```
Let's say I have 5 friends, they live closest to the
following nodes: Caulfield, Mount Waverley, Glen
Waverley, Melbourne Central and Chadstone

The following would be the fastest path to go from my
house (Brandon Park) to all my friends' and back:

{'cost': 182, 'path': ['Brandon Park', 'Wheelers Hill
Library', 'CGS WH', 'Glen Waverley', 'Mount Waverley',
'Richmond', 'Parliament', 'Melbourne Central', 'Flinders
Street', 'Caulfield', 'Chadstone', 'Oakleigh', 'Brandon
Park']}
```

Dijkstra's Algorithm

Dijkstra's Algorithm is a method for finding the shortest path between any two given nodes in a weighted graph, given that the weights are non-negative.
If some of the weights were negative, the Bellman-Ford Algorithm could

also be used to find the shortest path between two vertices, but as this is not the case for our model (a method of transport cannot take you negative time to get somewhere), Dijkstra's Algorithm is preferred for simplicity.

Dijkstra's Algorithm is a greedy algorithm, which actually finds the distance between a node and every other node on the graph. It does this based on the notion that if there were a shorter path than any sub-path, it would replace that sub-path to make the whole path shorter. More simply, shortest paths must be composed of shortest paths, which allows Dijkstra's to be greedy, always selecting the shortest path from "visited" nodes, using the principle of relaxation to gradually replace estimates with more accurate values.

Dijkstra's Algorithm follows the logic outlined by the following pseudocode:

```
function dijkstras (
    start: node,
    end: node,
    graph: graph
):
    // Set all node distance to infinity
    for node in graph:
        distance[node] = infinity
        predecessor[node] = null
        unexplored_list.add(node)

    distance[start] = 0

    while unexplored_list is not empty:
        min_node = unexplored node with min cost
        unexplored_list.remove(min_node)

```

```
            for each neighbour of min_node:
                    current_dist = distance[min_node] + dist(min_node, neighbour)
                    // a shorter path has been found to the neighbour ∴ relax value
                    if current_dist < distance[neighbour]:
                            distance[neighbour] = current_dist
                            predecessor[neighbour] = min_node

    return distance[end]
end function
```

After being implemented in Python (with a slight modification to return the path as well), the pseudocode looks like this:

```python
def dijkstra(start, end):
    # set all nodes to infinity with no predecessor
    distance = {node: float('inf') for node in g.nodes()}
    predecessor = {node: None for node in g.nodes()}
    unexplored = list(g.nodes())

    distance[start] = 0

    while len(unexplored) > 0:
        min_node = min(unexplored, key=lambda node: distance[node])
        unexplored.remove(min_node)

        for neighbour in g.neighbors(min_node):
            current_dist = distance[min_node] + dist(min_node, neighbour)
            # a shorter path has been found to the
neighbour ∴ relax value
            if current_dist < distance[neighbour]:
                distance[neighbour] = current_dist
                predecessor[neighbour] = min_node

    # reconstructs the path
    path = [end]
    while path[0] != start:
        path.insert(0, predecessor[path[0]])

    return {'cost': distance[end], 'path': path}
```