Algorithmics SAT - Friendship Network

Garv Shah

2022-06-02

Abstract

'How can a tourist best spend their day out?' 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.

Contents

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	
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

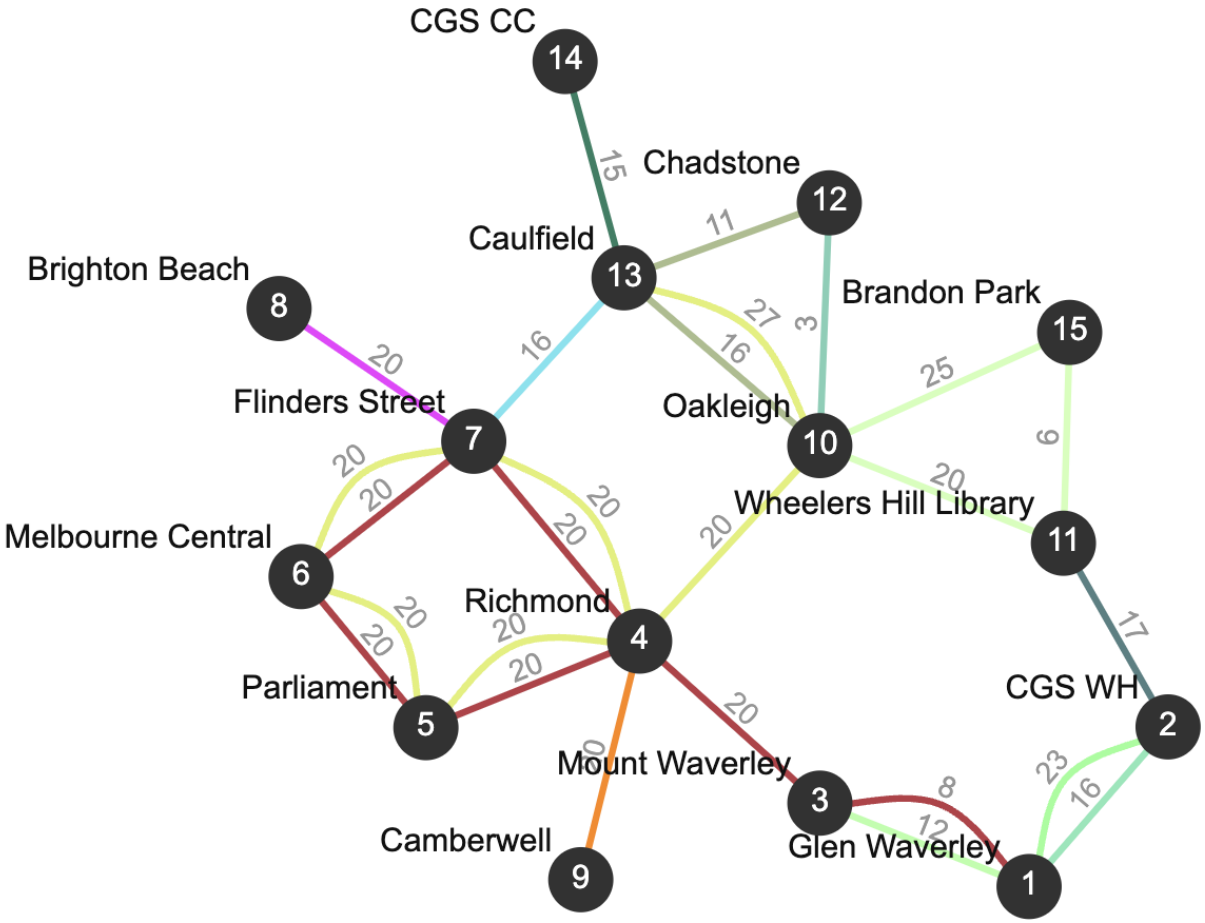

Figure 1: Possible Graph

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 $\text{Cost}_{\min} A \to B$whilst visiting all nodes in S = min(Cost$A \to C$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:

```
1. function held_karp (
2.  start: node,
3.  end: node,
4.  visit: set<node>
5. ):
6.  if visit.size = 0:
7.      return dist(start, end)
8.  else:
9.      min = infinity
10.         For node C in set S:
11.             sub_path = held_carp(start, C, (set \ C))
12.             cost = sub_path + dist(C, end)
13.             if cost < min:
14.                 min = cost
15.         return min
16. 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 Infinite Distance Problem The problem with this implementation is that it currently only works with complete 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 ∴ distance) between any two given nodes. (the justification of this specific algorithm selection is evaluated and challenged here)

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:

```
1. function dijkstras (
2.  start: node,
3.  end: node,
4.  graph: graph
5. ):
6.  // Set all node distance to infinity
7.  for node in graph:
8.      distance[node] = infinity
9.      predecessor[node] = null
10.         unexplored_list.add(node)
11. 
12.     distance[start] = 0
13. 
14.     while unexplored_list is not empty:
15.         min_node = unexplored node with min cost
16.         unexplored_list.remove(min_node)
17. 
18.         for each neighbour of min_node:
19.             current_dist = distance[min_node] + dist(min_node, neighbour)
20.             // a shorter path has been found to the neighbour -> relax value
21.             if current_dist < distance[neighbour]:
22.                 distance[neighbour] = current_dist
23.                 predecessor[neighbour] = min_node
24. 
25.     return distance[end]
26. 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}
```

Dijkstra's Algorithm vs Floyd Warshall's Shortest Path Algorithm

The problem that using Dijkstra's was attempting to solve was that Held-Karp treats the distance between two unconnected vertices as ∞, as demonstrated here.

There are 3 main shortest path algorithms covered in Unit 3: 1. Dijkstra's Algorithm: - Shortest path from **one** node to all nodes - Negative edges **not** allowed - Returns **both** path and cost 2. Bellman-Ford Algorithm: - Shortest path from **one** node to all nodes - Negative edges **allowed** - Returns **both** path and cost 3. Floyd-Warshall's Shortest Path Algorithm: - Shortest path between **all** pairs of vertices - Negative edges **allowed** - Returns **only** cost

As we can see, to be able to output the traversal path, we need both the cost and the path, so Floyd-Warshall's was initially discarded because it did not do so, even if it meant that the less desirable solution of running Dijkstra's from every source node had to be used, calculating the shortest path to every other node each time.

The most optimal solution would be an algorithm that returns both the cost and the traversal order of the shortest path between *all* pairs of vertices, as this operation is carried out many times by Held-Karp. Implementing Floyd-Warshall's Shortest Path with the modification of a predecessor matrix (similar to Bellman-Ford and Dijkstra's) was attempted, but this requires additional recursive computation to reconstruct the path, making it not ideal in terms of efficiency.

An alternative solution, Johnson's Algorithm, is one that gives us the exact output we want: the shortest path and cost between all vertex pairs. The algorithm works by first running Bellman-Ford to account for negative edge weights (not a problem for this SAT) and then runs Dijkstra's from every source node to construct a matrix and paths for each. Surprisingly, this algorithm is comparable to the efficiency of running just normal Floyd-Warshall's, and can even be faster in some cases.

As such, the only modification that needs to be made is that instead of calling Dijkstra's *every* time a vertex pair distance and path is needed, the whole distance matrix can be constructed at once, so subsequent calls only take $O(1)$ time instead. This can be achieved using dynamic programming, by caching the output of Dijkstra's whenever it is invoked, so we are only running the algorithm as many times as we need to.

Optimisations

The optimisations below were created after the following base case:

```
Let's say I have 9 friends, they live closest to the following nodes: {'Mount Waverley',
    'Melbourne Central', 'Chadstone', 'CGS WH', 'Parliament', 'Wheelers Hill Library',
    'Flinders Street', 'Brighton Beach', 'Camberwell'}
The following would be the fastest path to go from my house (Brandon Park) to all my friends'
    and back:
{'cost': 262, 'path': ['Brandon Park', 'Wheelers Hill Library', 'CGS WH', 'Glen Waverley',
    'Mount Waverley', 'Richmond', 'Camberwell', 'Richmond', 'Parliament', 'Melbourne Central',
    'Flinders Street', 'Brighton Beach', 'Flinders Street', 'Caulfield', 'Chadstone',
    'Oakleigh', 'Brandon Park']}

It took 47.3621 seconds to run.
```

As seen, running the above Held-Karp + Dijkstra's combination took about 50 seconds to calculate the minimal cost path for 9 nodes. The following is a table for $n$vs$t$, with an approximate line of best fit of $y \approx a \times b^x$ where $a = 8.1017 \times 10^{-8}$ and $b = 9.3505$:

n (no. nodes)	t (execution time in seconds, 4dp)	y (line of best fit, 4dp)
0	0.0001	0.0000
1	0.0002	0.0000
2	0.0002	0.0000
3	0.0016	0.0001
4	0.0083	0.0006
5	0.0132	0.0058
6	0.1090	0.0541
7	0.5674	0.5063
8	4.7193	4.7343
9	44.2688	44.2680

Anything above 7 nodes takes far too long, and calculating the entire hamiltonian circuit would take 5 weeks 1 day 14 hours 56 mins and 39 secs based on the line of best fit, so the following optimisations have been utilised.

Caching Dijkstra's Output

When replacing the `dist` function with Dijkstra's Algorithm, a certain time compromise was made. `dist` has a time complexity of $O(1)$, simply fetching the distance from the distance matrix, but Dijkstra's Algorithm is relatively slower at $O(E \log V)$ where E is the number of edges and V the number of vertices. For our sample graph above, with $E = 27$ and $V = 15$, $O(E \log V) \approx 31.75$. This makes using Dijkstra's roughly 31 times slower than `dist`as it is called every time.

To avoid this, we can cache the results of Dijkstra's Algorithm to avoid running the same calculation multiple times. This can be done with the following pseudocode:

```
1. cached_djk = dictionary of node -> dict
2.
3. function fetch_djk (
4.  start: node,
5.  end: node,
6. ):
7.  if cached_djk[start] does not exists:
8.      cached_djk[start] = dijkstras(start)
9.
10.     djk = cached_djk[start]
11.     # reconstructs the path
12.     path = [end] as queue
```

```
13. while path.back != start:
14.     path.enqueue(djk['predecessors'][path.back])
15.
16. return {
17.     'distance': djk['distances'][end],
18.     'path': path
19. }
20. end function
```

In this case, `dijkstras` would need to be modified to return the `distance` and `predecessor` rather than just `distance[end]`.

After being implemented in Python, `cached_djk` resembles the following:

```python
def fetch_djk(start, end):
    if start not in cached_djk:
        cached_djk[start] = dijkstra(start)

    djk = cached_djk[start]
    # reconstructs the path
    path = [end]
    while path[0] != start:
        path.insert(0, djk['predecessors'][path[0]])

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

Performance Improvement As expected by the theoretical time savings calculated above, this optimisation makes Held-Karp roughly 31 times faster. The base case from above, which took 44 - 47 seconds before the optimisation now only takes about 1.25 seconds.

```
Let's say I have 9 friends, they live closest to the following nodes: {'Parliament',
    'Melbourne Central', 'Chadstone', 'Camberwell', 'Flinders Street', 'Brighton Beach',
    'Mount Waverley', 'CGS WH', 'Wheelers Hill Library'}
The following would be the fastest path to go from my house (Brandon Park) to all my friends'
    and back:
{'cost': 262, 'path': ['Brandon Park', 'Wheelers Hill Library', 'CGS WH', 'Glen Waverley',
    'Mount Waverley', 'Richmond', 'Camberwell', 'Richmond', 'Parliament', 'Melbourne Central',
    'Flinders Street', 'Brighton Beach', 'Flinders Street', 'Caulfield', 'Chadstone',
    'Oakleigh', 'Brandon Park']}

It took 1.2799 seconds to run.
```

The $n$vs$t$ table now looks like this, with an approximate line of best fit of $y \approx a \times b^x$ where $a = 1.4002 \times 10^{-9}$ and $b = 10.1876$:

n (no. nodes)	t (execution time in seconds, 4dp)	y (line of best fit, 4dp)
0	0.0001	0.0000
1	0.0001	0.0000
2	0.0001	0.0000
3	0.0001	0.0000
4	0.0001	0.0000
5	0.0005	0.0002
6	0.0060	0.0016
7	0.0287	0.0159
8	0.2148	0.1625
9	1.6055	1.6551
10	17.4555	16.8620

n (no. nodes)	t (execution time in seconds, 4dp)	y (line of best fit, 4dp)
11	171.6719	171.7832
12	1750.1065	1750.0590

We can see that this line of best fit is relatively accurate, and if we extend it to run for 14 nodes (our hamiltonian circuit), it would take a total of about 2 days 2 hours 27 mins and 14 secs to compute it all.