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

Information to Consider . . . 1
Node Representation . . . 1
Edge Representation . . . 1
Weight Representation . . . 2
Additional Information Modelled Outside Graph . . . 2
Abstract Data Types . . . 2
Possible Graph . . . 3
Signatures . . . 3
Algorithm Selection . . . 3
Node Selection Algorithm . . . 4
Fare Cost Calculation Algorithm . . . 6
Held-Karp Algorithm . . . 7
Dijkstra's Algorithm . . . 9
Considering Train/Bus Arrival Times & Switching Lines . . . 10
Dijkstra's Algorithm vs Floyd Warshall's Shortest Path Algorithm . . . 11
Optimisations . . . 11
Caching Dijkstra's Output . . . 12

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 pick up all my friends, 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

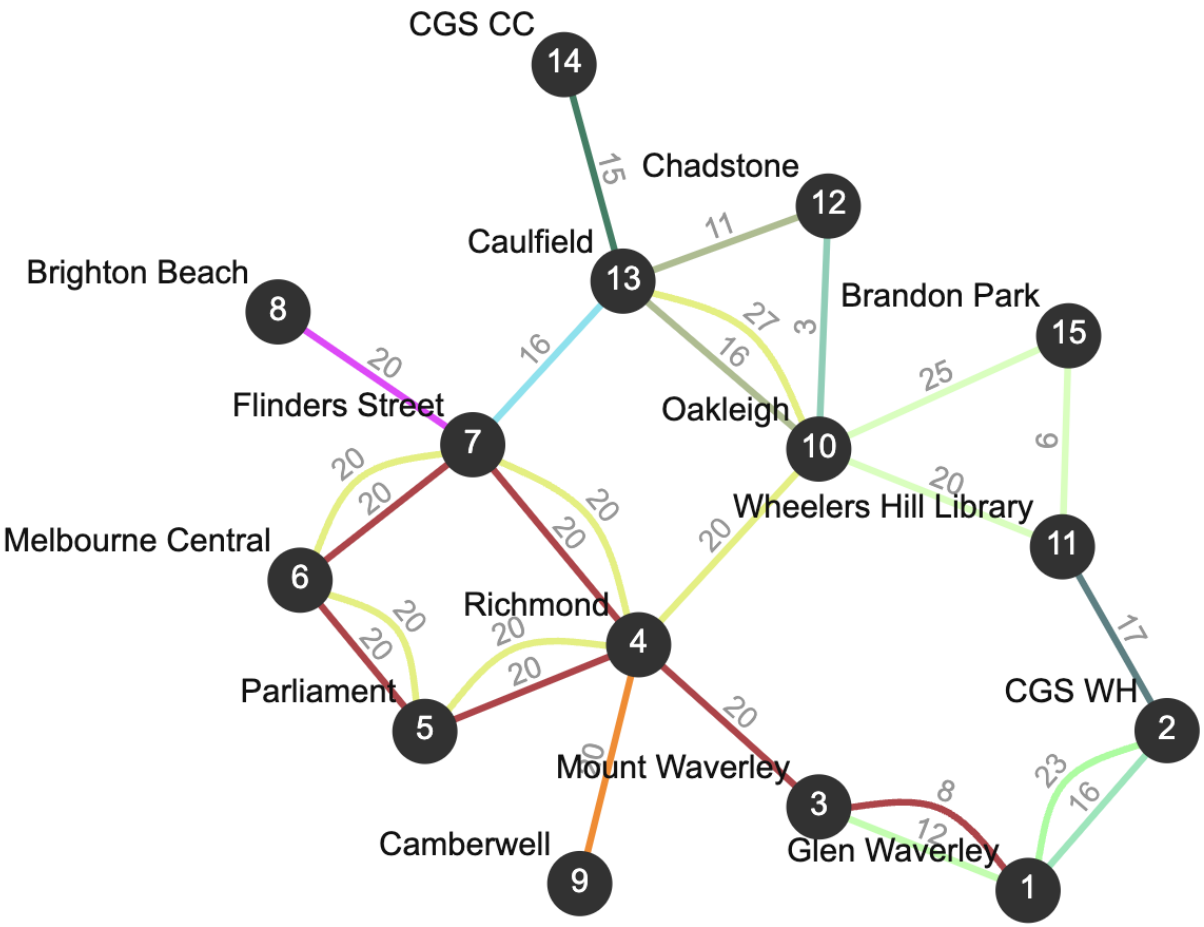

Figure 1: Possible Graph

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 that is needed to be visited to pick up my friends.

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.

Node Selection Algorithm

Before we can find the shortest circuit that visits a set of nodes, we need to know what nodes to visit in the first place! Each node, which is part of the public transport network, can be assigned latitude and longitude coordinates, and these can be compared with the coordinates of each of my friends' houses to determine the shortest distance they would need to walk to reach a transport hub that is represented as a node on our graph.

The process of finding the nodes can then ∴ be represented as the following informal steps: 1. Get the latitude and longitude coordinates of all transport hubs and friends' houses. 2. Loop over all friends and transport hubs, comparing the distance of each to find the closest transport hub to each friend. 3. Finally store each friends' closest transport hub and distance into their respective dictionary entries.

The question still remains though: how can we find the distance between two lat/long coordinates? The answer is the haversine formula!

The Haversine Formula The haversine formula determines the distance between two points on a sphere given their latitude and longitude coordinates. Using the distance formula $\sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$ may be sufficient in terms of finding the closest transport hub, but the distances it provides only work on a flat cartesian plane, not spheres like the earth, distances which could be used for later computation such as time taken to walk to the transport hubs.

The haversine formula can be rearranged given that the Earth's radius is 6371km to give us the following equation (with d representing the distance between two locations):

$\Delta lat = lat_1 - lat_2 \ \Delta long = long1 - long_2 \ R = 6371$

$a = \sin^2(\frac{\Delta lat}{2}) + \cos(lat_1)\cos(lat_2)\sin^2(\frac{\Delta long}{2}) \ c = 2\,\text{atan2}(\sqrt{a}, \sqrt{1-a}) \ d = R \times c$

It *is* somewhat long on not the cleanest formula, but it should be more than sufficient in our code.

Pseudocode Finally we can use the informal steps above to construct the following pseudocode:

```
distance_dict: dictionary = {}

function calculate_nodes (
    friend_data: dictionary,
    node_data: dictionary
):
    for friend in friend_data:
        home: tuple = friend['home']
        // initial min vals that will be set to smallest iterated distance
        min: float = infinity
        min_node: node = null

        for node in node_data:
            location: tuple = node['coordinates']
            // find real life distance (functional abstraction)
            distance: float = latlong_distance(home, location)
            if distance < min:
                min = distance
                min_node = node

        distance_dict[friend]['min_node'] = min_node
        distance_dict[friend]['distance'] = min
end function
```

This combines the haversine formula and simple iteration to find the minimum distance node for each and stores it into a dictionary. When translated to Python, the above code looks like this:

```python
def lat_long_distance(coord1, coord2):
    # assign lat/long from coords
    lat1 = coord1[0]
    long1 = coord1[1]
    lat2 = coord2[0]
    long2 = coord2[1]

    # radius of earth
    r = 6371

    # equation definitions from haversine formula
    phi_1 = math.radians(lat1)
    phi_2 = math.radians(lat2)

    delta_phi = math.radians(lat2 - lat1)
    delta_lambda = math.radians(long2 - long1)

    a = math.sin(delta_phi / 2.0) ** 2 + math.cos(phi_1) * math.cos(phi_2) *
        math.sin(delta_lambda / 2.0) ** 2

    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))

    # distance in kilometers
    d = r * c

    return d


def calculate_nodes(friend_data, node_data):
    distance_dict = {}
    for friend in friend_data:
        friend_home = friend_data[friend]['home']
        # initial min vals that will be set to smallest iterated distance
        min_dist = float('inf')
        closest_node = None

        for node in node_data:
            location = node_data[node]
            distance = lat_long_distance(friend_home, location)
            if distance < min_dist:
                min_dist = distance
                closest_node = node

        distance_dict[friend] = {}
        distance_dict[friend]['closest_node'] = closest_node
        distance_dict[friend]['distance'] = min_dist
    return distance_dict
```

The output of this code on our data set is as follows:

```
{
    'Garv': {'min_node': 'Brandon Park', 'distance': 0.4320651871428905},
    'Grace': {'min_node': 'Caulfield', 'distance': 3.317303898425856},
    'Sophie': {'min_node': 'Camberwell', 'distance': 10.093829041341555},
    'Zimo': {'min_node': 'CGS WH', 'distance': 1.0463628559819804},
```

```
    'Emma': {'min_node': 'Wheelers Hill Library', 'distance': 2.316823113596007},
    'Sabrina': {'min_node': 'CGS WH', 'distance': 1.0361159593717744},
    'Audrey': {'min_node': 'CGS WH', 'distance': 6.99331705920331},
    'Eric': {'min_node': 'Glen Waverley', 'distance': 2.591823985420863},
    'Isabella': {'min_node': 'CGS WH', 'distance': 2.048436485663766},
    'Josh': {'min_node': 'CGS WH', 'distance': 0.656799522332077},
    'Molly': {'min_node': 'Wheelers Hill Library', 'distance': 7.559508844793643},
    'Avery': {'min_node': 'Mount Waverley', 'distance': 6.312529532145972},
    'Sammy': {'min_node': 'Brandon Park', 'distance': 3.408577759087159},
    'Natsuki': {'min_node': 'CGS WH', 'distance': 6.419493747390275},
    'Liam': {'min_node': 'Mount Waverley', 'distance': 0.8078481833574709},
    'Nick': {'min_node': 'Glen Waverley', 'distance': 1.3699143560496139},
    'Will': {'min_node': 'Wheelers Hill Library', 'distance': 6.404888550878483},
    'Bella': {'min_node': 'Wheelers Hill Library', 'distance': 0.7161158445537555}
}
```

If it takes any of my friends' more than 20 minutes to walk to their transport location, I'd probably want a little warning advising me to consider adding closer transport hubs, because that seems like an awfully long time to walk! This can be done by considering the average human walking speed of 5.1km/h. Dividing their distance to transport hubs by this constant should give a good approximation of walking time. This gives the following list of friends that it would be too long for, and we can consider expanding our graph for better results:

```
Warning! These 11 friends have to walk more than 20 minutes in order to get to their transport
    hub. Possibly consider adding hubs closer to their houses:  Grace (39.03), Sophie
    (118.75), Emma (27.26), Audrey (82.27), Eric (30.49), Isabella (24.1), Molly (88.94),
    Avery (74.27), Sammy (40.1), Natsuki (75.52) and Will (75.35)
```

Evaluation of Solution The solution above works alright for short distances, but slightly breaks apart the further you have to go. This is because humans in the real world have to walk across set designated pathways that the algorithm is not aware of, which is simply calculating the direct distance, which could be walking directly through houses or shopping centres. As such, the distances and times taken for walking are very much approximations in this model that could be further refined by a path finding algorithm that has an awareness of roads and pathways, but as that is an immense amount of data, this approximation will have to suffice for the purposes of this SAT.

Fare Cost Calculation Algorithm

As well as the time taken to pick up all my friends, it would be useful for the algorithm to tell me how much the trip costs in ride fairs. PTV uses a "zoning system" that charges different for the zones you are in. It also charges a set rate for under 2 hours of travel, and a seperate "daily rate" for any more than that:

2 hour	Zone 1 + 2	Zone 2
Full Fare	$4.60	$3.10
Concession	$2.30	$1.55

Daily	Zone 1 + 2	Zone 2
Full Fare	$9.20	$6.20
Concession	$4.60	$3.10

There are also caps on public holidays and weekends set to $6.70 for full-fare users and $3.35 to concession users. Zone 0 can be used to denote the free zone as well, or transport methods such as walking or cycling that have no associated cost.

This can be setup into the following conditional statements in pseudocode to calculate fare prices:

```
function calculate_prices (
    line_data: dictionary,
    hamiltonian_path: dictionary,
    concession: boolean,
    holiday: boolean
):
    zones: set = {}
    // add all traversed zones into a set to see which zones were visited
    for node in hamiltonian_path['path']:
        zones.add(line_data[node['line']]['zone'])

    money = 0

    // if it took us 2 hours or less
    if hamiltonian_path['time'] <= 120:
        // 2 hour bracket
        if zones has 1 and 2:
            if concession:
                money = 2.30
            else:
                money = 4.60
        else if zones has 2:
            // just zone 2
            if concession:
                money = 1.55
            else:
                money = 3.10
    else:
        // daily fare bracket
        if zones has 1 and 2:
            if concession:
                money = 4.60
            else:
                money = 9.20
        else if zones has 2:
            // just zone 2
            if concession:
                money = 3.10
            else:
                money = 6.20

    // if it is a weekend or a holiday
    if holiday:
        if concession and money > 3.35:
            money = 3.35
        else if money > 6.70:
            money = 6.70

    return money
end function
```

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:

```
1  def held_karp(start, end, visit):
2      if type(visit) is not set:
3          print("Error: visit must be a set of nodes")
4          return {'cost': float('inf'), 'path': None}
5      if len(visit) == 0:
6          return {'cost': dist(start, end), 'path': [start, end]}
7      else:
8          minimum = {'cost': float('inf')}
9          for rand_node in visit:
10             sub_path = held_karp(start, rand_node, visit.difference({rand_node}))
11             cost = dist(rand_node, end) + sub_path['cost']
12             if cost < minimum['cost']:
13                 minimum = {'cost': cost, 'path': sub_path['path'] + [end]}
14         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 $\therefore$ 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!

```
1  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}
```

Considering Train/Bus Arrival Times & Switching Lines

Evidently, trains do not leave immediately when you get to the station, and neither do buses. The algorithm thus far assumes no waiting time during transit, and as anyone who has used public transport would know, this is not realistic. As such, the arrival time of trains and buses needs to be considered. This also has the added benefit of factoring in the time it takes to switch lines, as this time is lost waiting for another train or bus.

All the algorithms above eventually call the `dist` function to get the direct distance between two nodes, which in and of itself is an abstraction of a distance matrix. By taking the input of the current time, the `dist` function can consider how long one must wait for a bus/train to arrive at the node, and modify the edge weights according, returning a larger cost for edges that require long wait times.

The following `dist` function takes the above into consideration:

```
function dist (
    start: node,
    end: node,
    current_time: datetime
):
    // if the start and end node are the same, it takes no time to get there
    if start = end:
        return 0
    else if edges = null:
        // if no edge exists between nodes
        return infinity

    edges = edge_lookup_matrix[start][end]
    distances = []

    // go over each possible edge between nodes (multiple possible)
    for edge in edges:
        line = edge.line
        // next time bus/train will be at node (functional abstraction)
        next_time = soonest_time_at_node(timetable, line, start, current_time)
        wait_time = next_time - current_time
        distances.add(edge.weight + wait_time)

    return min(distances)
```

```
25 end function
```

After implementing this function, an additional problem is introduced: how can the algorithms that are dependant on `dist` be aware of the current time?

Implementing Current Time in Dijkstra's The process for keeping track of the current time for Dijkstra's is relatively simple: it will just be the given time of day inputed into Dijkstra's + n amount of minutes, where n is the distance to the `min_node`. As such line 19 from the pseudocode above simply needs to be changed to the following, along with a new input of `current_time`

```
1 current_dist = distance[min_node] + dist(min_node, neighbour, current_time +
      to_minutes(distance[min_node]))
```

This works because distance in our algorithm is analogous to minutes, and since the `dist` function returns the correct distance initially and stores it into the distance array, subsequent calls will be using the correct distance from `distance[min_node]` along with the correct distance from the `dist` function. This informal proof by mathematical induction shows the correctness of this modification, which seems to work well when tested within the algorithm.

Implementing Current Time in Held-Karp

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:

```
1 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'}
2 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:

```
cached_djk = dictionary of node -> dict

function fetch_djk (
    start: node,
    end: node,
):
    if cached_djk[start] does not exists:
        cached_djk[start] = dijkstras(start)

    djk = cached_djk[start]
    # reconstructs the path
    path = [end] as queue
    while path.back != start:
        path.enqueue(djk['predecessors'][path.back])

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

```
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}
```

Update: Caching After Timetable Considerations The above pseudocode for `fetch_djk` breaks once considerations of train/bus arrival times are added, because for example, the time it takes to travel from Glen Waverley to Melbourne Central at 7am is not necessarily the same as the same trip at 9pm. Above, the `cached_djk` dictionary only takes the starting node into consideration, so the pseudocode has to be modified to the following to us an 'id' like system for the paths.

```
cached_djk = dictionary of node -> dict

function fetch_djk (
    start: node,
    end: node,
    current_time: datetime,
):
    name = start + '@' + current_time

    if cached_djk[name] does not exists:
        cached_djk[name] = dijkstras(start)

    djk = cached_djk[name]
    # reconstructs the path
    path = [end] as queue
    while path.back != start:
        path.enqueue(djk['predecessors'][path.back])

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

As such we can have a more specific key in our dictionary. This does have the disadvantage of having less reusable paths (running at 7 nodes was about 4 times slower than below), but at least the result isn't nondeterministic!

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
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.