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

The general problem of planning trips with friends can be made more specific by considering scenarios for hangouts. In this particular scenario, my friends have decided that we want to travel in one big travel party and I will start and end my day at my house, picking up all my friends along the way. This form of hangout is quite common with my friends, where we pick up people along the way to get to a final destination. 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

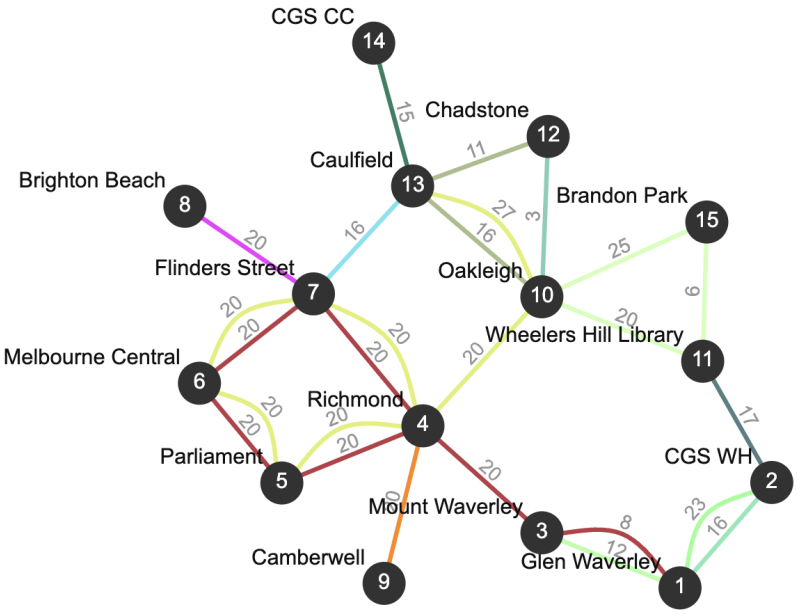

Figure 1: Possible Graph

Final Graph

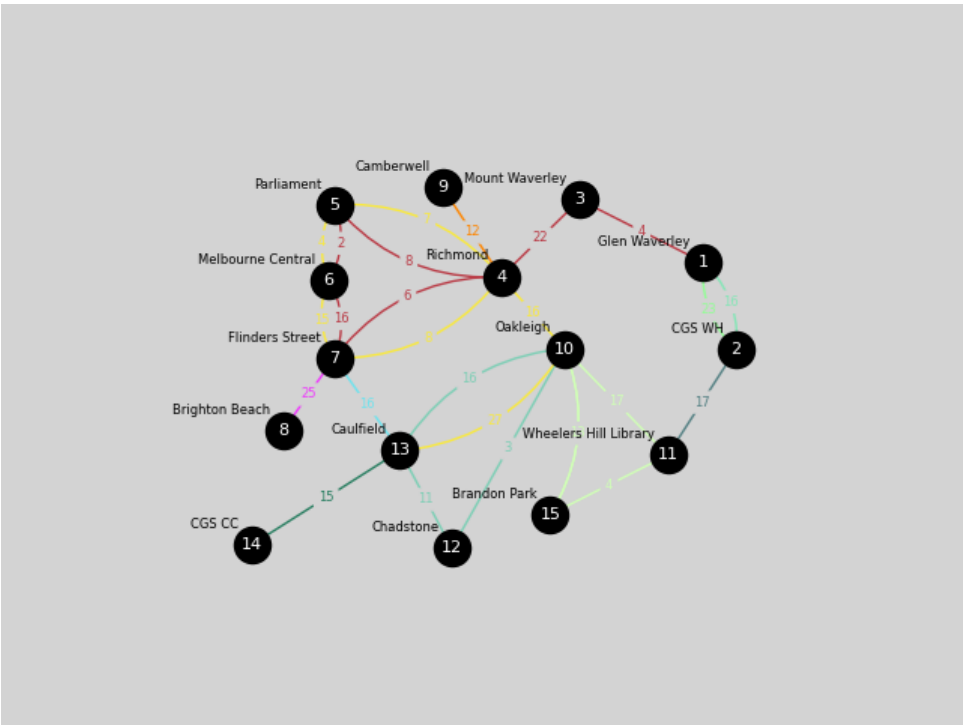

Figure 2: Final Graph

Signatures

Function Name	Signature
add_landmark	[name, timetable, latlong_coordinates] -> node
add_route	[start_node, end_node, travel_method, time, line?] -> edge
add_line	[colour, zone, timetable] -> dictionary
add_friend	[name, latlong_coordinates] -> dictionary
setup_graph	[landmarks, routes, friends, timetable] -> graph
latlong_distance	[coord1, coord2] -> floating point number
calculate_nodes	[friend_data, node_data] -> dictionary<string, node or float>
calculate_prices	[line_data, hamiltonian_path, concession, holiday] -> float
dist	[start, end, current_time] -> float
fetch_djk	[start, end, graph, current_time] -> dictionary with cost and path
dijkstras	[start, end, graph, current_time] -> cost and path
held_karp	[start, end, visit, current_time] -> cost and path

Function signatures can also be found within the `main.py` Python file as comments within the code

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 $O(n!)$.

The Travelling Salesman problem does not allow us to be greedy, because for us to choose the best choice at any moment, we have to be able to discard all other solutions. TSP is too complex for this, as going down any node may lead to a shorter solution later on. Because of this, solving for the TSP has to use the decrease and conquer principle to make our problem smaller piece by piece, which can be done by recursion or using dynamic programming if the results of operations are saved.

Held-Karp works by utilising the the following information.

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

We $\therefore$ know that the minimum cost of going from A to B while visiting all nodes in the set S can be split up into the following two parts: - Going from A to C (a random node in S) while visiting all nodes in the set S besides C - Going from C to B directly Essentially, this goes through the set S and makes any node C the last node, giving us the same problem with a smaller set. This then allows us to identify that the problem is recursive, as the larger path can be split up into smaller and smaller sub-paths by the above logic, until we reach a base case of S having length 0, where we can then just calculate the direct distance.

To reiterate more formally: $\text{Cost}_{\min} A \to B$whilst visiting all nodes in S = min(Cost$A \to C$visiting everything else in S+ d_{CB}). As such, 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>
):
    // base case: if no visit set then we can just return direct distance
    if visit.size = 0:
        return dist(start, end)
    else:
        min = infinity
        // find the minimum subpath
```

```
        For node C in set S:
            // uses property described above to split larger path into smaller subpath, and
                solves recursively
            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 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!

```
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)

    // starting distance has to be 0
    distance[start] = 0

    // while more to still explore
    while unexplored_list is not empty:
        min_node = unexplored node with min cost
        unexplored_list.remove(min_node)

        // go through every neighbour and relax
        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)
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`

```
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 argument by mathematical induction demonstrates the correctness of this modification, which seems to work well when tested within the algorithm.

Implementing Current Time in Held-Karp Factoring in the current time into Held-Karp follows the same recursive nature as the algorithm itself. First we can change the base case to work with the new Dijkstra's Algorithm outlined above:

```
1 if visit.size = 0:
2     djk = dijkstras(start, end, current_time)
3     return djk['cost']
```

Now that our base case is returning a cost with the current time factored in, we need to make the sub path on line 11 of the original algorithm also factor in the current time. The current time when the sub_path is created will always be the current time at the start node, which we defined as the time inputed into Held-Karp at initialisation. As such, the line is changed to the following:

```
1 sub_path = held_carp(start, C, (set \ C), current_time)
```

Finally, the only other change needs to be made on line 12. Previously, we replaced the `dist` function here with `dijkstras` to solve the Infinite Distance Problem, but Dijkstra's also requires the input of time. As the starting node here is C, or the randomly selected node, the current time for this function call would have to be the time when we are at C. This can simply be found by treating the distance of `sub_path` as minutes which are added to the current time, as the `sub_path` ends at the same random node C. As such, line 12 can be changed to the following:

```
1 djk = dijkstras(C, end, current_time + toMinutes(sub_path['cost']))
2 cost = sub_path['cost'] + djk['cost']
```

This leaves us with the a sound implementation of Held-Karp factoring in time, demonstrated by the following pseudocode:

```
1  function held_karp (
2      start: node,
3      end: node,
4      visit: set<node>,
5      current_time: datetime
6  ):
7      if visit.size = 0:
8          djk = dijkstras(start, end, current_time)
9          return djk['cost']
10     else:
11         min = infinity
12         For node C in set S:
13             sub_path = held_carp(start, C, (set \ C), current_time)
14             djk = dijkstras(C, end, current_time + toMinutes(sub_path['cost']))
15             cost = sub_path['cost'] + djk['cost']
16             if cost < min:
17                 min = cost
18         return min
19 end function
```

This works because of a similar principle to the informal argument for the modified Dijkstra's correctness: it works for the base case (because Dijkstra's works), and it also must work for the $k+1$ case, because the time being inputed into the functions is always the time at the starting nodes. It then $\therefore$ works for all cases, which seems to also be true when used in practice.

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:
3 {'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']}
4
5 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

n (no. nodes)	t (execution time in seconds, 4dp)	y (line of best fit, 4dp)
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:

```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}
```

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

n (no. nodes)	t (execution time in seconds, 4dp)	y (line of best fit, 4dp)
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.

Justification of Solution

Throughout this report, each individual algorithm has been challenged and justified for it's suitability and effectiveness at solving their individual problems. To evaluate the overall suitability of the combined algorithms, we can refer back to our original problem:

> I've been finding it hard to plan hangouts with my friends, and I want a solution that will plan a trip using the Victorian public transport network so that can find the quickest route to pick up all of my friends and we can all come back to my house.

In reality, this is a relatively niche use case, as most friends *could* just travel on their own, but given that I want to pick up all my friends along the way, this solution its suitability and fitness for purpose well.

Below is the output of the solution when I (`Garv`, with a concession card) leave my house at `8:30am` , on a Saturday:

```
I have 18 friends and they live closest to the following 7 nodes:
Grace lives 3.317km from Caulfield
Sophie lives 10.094km from Camberwell
Zimo lives 1.046km from CGS WH
Emma lives 2.317km from Wheelers Hill Library
Sabrina lives 1.036km from CGS WH
Audrey lives 6.993km from CGS WH
Eric lives 2.592km from Glen Waverley
Isabella lives 2.048km from CGS WH
Josh lives 0.657km from CGS WH
Molly lives 7.56km from Wheelers Hill Library
Avery lives 6.313km from Mount Waverley
Sammy lives 3.409km from Brandon Park
Natsuki lives 6.419km from CGS WH
Liam lives 0.808km from Mount Waverley
Nick lives 1.37km from Glen Waverley
Will lives 6.405km from Wheelers Hill Library
Bella lives 0.716km from Wheelers Hill Library
You (Garv) live 0.432km from Brandon Park

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)

The trip would cost you $3.35 and would take you 266.17 minutes, taking the following route:
From Brandon Park (Garv, Sammy) to Wheelers Hill Library (Emma, Molly, Will, Bella) to CGS WH
    (Zimo, Sabrina, Audrey, Isabella, Josh, Natsuki) to Glen Waverley (Eric, Nick) to Mount
```

```
    Waverley (Avery, Liam) to Richmond to Flinders Street to Caulfield (Grace) to Flinders
    Street to Richmond to Camberwell (Sophie) to Richmond to Oakleigh and back to Brandon Park.

It took 0.8578 seconds to run.
```

The correctness of this being the quickest route was presented as informal arguments via mathematical induction throughout the report, relying on modifications to the Held-Karp Algorithm to model features of the real world scenario and provide us with an answer to our problem. As can be seen above, the solution suitably provides the fastest route, which friends will be picked up at which nodes, the time it would take for the traversal to occur and the overall cost of the trip. This satisfactorily answers the initial problem and is fit for the purpose of planning real life trips that would involve picking up all my friends to visit my house.

Final Code

The final Python implementation of the code can be found here on Trinket. Below is the final main thread in structured pseudocode that invokes all the modules described throughout the report.

```
function main(
    friends: dictionary,
    landmarks: dictionary,
    routes: dictionary,
    timetable: dictionary
):
    // global variable declarations
    concession: bool = Ask the user "Do you posses a concession card?"
    holiday: bool = Ask the user "Is today a weekend or a holiday?"
    user_name: string = Ask the user to select a friend from friends dictionary
    selected_time = Ask the user what time they are leaving

    cached_djk: dictionary = empty dictionary
    edge_lookup_matrix: matrix = |V| x |V| matrix that stores a list of edges in each entry

    // get distance of all friends from landmarks
    friend_distances: dictionary = calculate_nodes(friends, landmarks)
    visit_set: set = set of all closest nodes from friend_distances
    people_at_nodes: dictionary = all friends sorted into keys of which nodes they are closest
        to, from visit_set

    home: string = closest node of user_name

    print all friends, where they live closest to and how far away

    print out friends that would take more than 20 minutes to walk (average human walking
        speed is 5.1 km/h)

    hamiltonian_path = held_karp(home, home, visit_set, selected_time)

    print how much the trip would cost and how long it would take

    print the path of the hamiltonian_path
end function
```

Algorithmics SAT - Friendship Network Part 2

Garv Shah

2023-07-28

Abstract

'How can a tourist best spend their day out?'. The first part of this SAT project aimed to model the Victorian public transport network and its proximity to friends' houses in order to construct an algorithm. In Part 2 we will now consider the time complexity of said algorithms and analyse their impact on real life use-cases.

Contents

This section of the Algorithmics SAT focuses on a time complexity analysis of the solution in order to establish the efficiency of the algorithm and feasibility in the real world.

Throughout the analysis, note the following variables are used as shorthand:

Let F = number of friends

Let L = number of landmarks

Let R = number of routes

Time Complexity Analysis

Expected Time Complexity

As explained in Part 1 of the SAT, the algorithm in essence boils down to an applied version of the Held–Karp algorithm, which has an optimal worst case time complexity of $O(n^2 2^n)$. Hence, it would make sense for our combination of Held-Karp and Dijkstra's to result in a time complexity slightly larger.

Call Tree

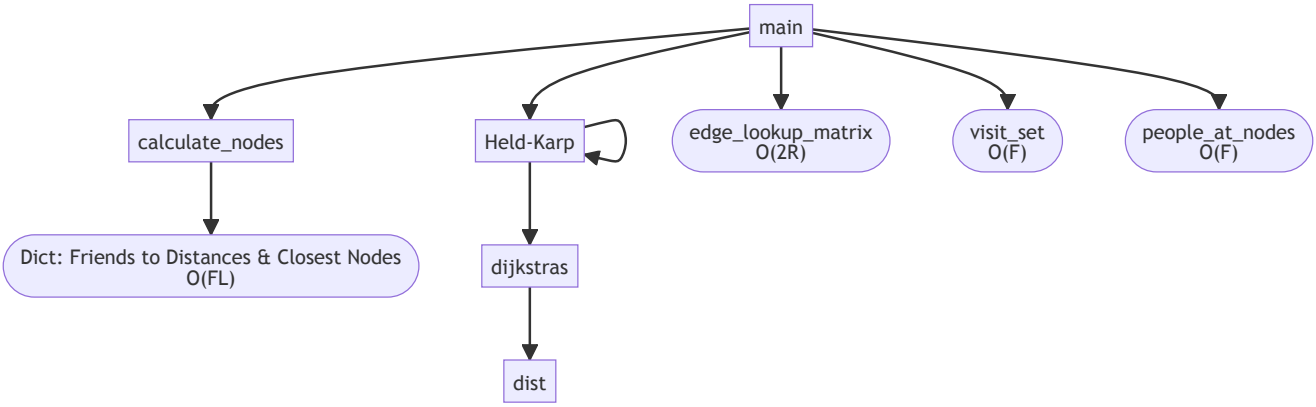

Figure 1: Call Tree

As we can see, the main function calls a few distinct processes [1]:

1. First it creates the edge lookup matrix, which is abstracted in the pseudocode. This Big O time is derived from the Pythonic implementation of the lookup matrix as follows [2]:

```
edge_lookup_matrix = {frozenset({edge['from'], edge['to']}): [] for edge in edges}
for edge in edges:
    edge_lookup_matrix[frozenset({edge['from'], edge['to']})].append(edge)
```

   Evidently, this loops over each edge in `edges` twice, resulting in a linear time complexity of $O(2R)$

2. It then calls `calculate_nodes` with an input of both `friends` and `landmarks`, the output of which is used to create our `visit_set`. This Big O time is derived from the fact that `calculate_nodes` is simply a nested for-loop, iterating over each friend and every landmark, resulting in a worst case time complexity of $O(F \times L)$.

3. It now uses the output of `calculate_nodes` (stored as `friend_distances`) to create a set of nodes we need to visit, which is abstracted in the pseudocode. This Big O time is derived from the Pythonic implementation of the set as follows:

```
visit_set = set(val['closest_node'] for key, val in friend_distances.items())
```

   Evidently, this loops over each friend once, resulting in a linear time complexity of $O(F)$

4. Similar to the above implementation, the `main` function now creates `people_at_nodes` to create a dictionary of nodes and which people are closest to that node, with a similar $O(F)$ as above.

5. Various other print statements are called, all with $O(F)$ time to display information about each friend.

6. Finally, after all this prep is done, `held_karp` is called to find the shortest hamiltonian path of the graph.

As we can see from this process and the call tree above, there are 3 main elements that contribute to the time complexity of our algorithm besides `held_karp`:

1. `calculate_nodes` which contributes $F \times L$ to our time.

2. Calculating the `edge_lookup_matrix`, which contributes $2R$ to our time complexity but simply turns into R when considering the asymptotic complexity.

[1] This analysis is done assuming that the time complexity of accessing a dictionary, list or array element is $O(1)$, as these basic pseudocode elements are generally done in constant time.

[2] Due to the nature of functional abstraction, the implementation of creating the `edge_lookup_matrix` is not specified in the pseudocode. Although it is referred to as a lookup matrix of size $|V| \times |V|$ which would have a quadratic time complexity, the pseudocode has actually been implemented as a dictionary in $O(2R)$ time, which is a bit more efficient. Nonetheless, even if it was changed to $O(L^2)$, it would make minimal difference to the final asymptotic time complexity.

3. Calculating the `visit_set`, `people_at_nodes` and two other print calls. This contributes $4F$ where 4 accounts for these 4 processes but could be any other arbitrary constant, as this simply turns into F when considering the asymptotic time complexity.

If we let the time complexity of `held_karp` be represented by $HK(n)$ where n denotes the calculated size of the `visit_set`, our current time complexity of the `main` function can be represented as $O(HK(n) + FL + R + F)$.

Held-Karp Time Complexity

Figuring out the time complexity of the other processes in our algorithm was relatively easy; we can simply look at their pseudocode implementation (or what they would be if they are abstracted) and look at the general number of operations. Held-Karp on the other hand is a bit harder as it is a recursive algorithm, making direct analysis a bit more troublesome. To begin, we can try to represent the modified Held-Karp algorithm as a recurrence relation to aid in mathematical analysis.

To recap, Held-Karp[3] works by utilising the fact that every subpath of a path of minimum distance is itself of minimum distance. This means that we can reduce the length of S by one each time by finding the minimum distance/path between C and B while running Held-Karp again on the set S without C, but as C as the new value for B.

As stated in part 1, this logic can be represented recursively as the following:

Let $\text{Cost}_{A\to B,S}$ = The minimum cost of a cycle free path from A to B that visits all the vertices of S.

Let $d_{A,B}$ = The minimum cost of travelling from A to B, as outputted by Dijkstra's.

$\therefore \text{Cost}_{A\to B,S} = \min(\text{Cost}_{A\to C,S-\{B\}} + d_{CB})$

We can then turn this into a recurrence relation for Big O, where n is the size of the set S and $d(n)$ is the cost function, which in our case is Dijkstra's:

$$T_n = \begin{cases} n(T_{n-1} + d(n)) & n > 0 \\ d(n) & n = 0 \end{cases}$$

Now that we have a recurrence relation for Held-Karp in terms of the cost of running Dijkstra's, the next logical step is to find the number of operations required to run Dijkstra's every time (which would be in the worst case scenario where none of our previous calculations are reused).

Dijkstra's Time Complexity

We can analyse Dijkstra's step by step by viewing all the elements of the pseudocode and evaluating them separately and then add them up together at the end:

1. We can see that initial loop runs for every node, or L times, as each node represents a landmark.
2. In the main while loop, we iterate over every node in the graph, making the while loop run L times as well.
3. To find the `min_node`, the pseudocode iterates over every single node in the `unexplored_list`. As this list decreases by one each time, the total cost of finding the `min_node` can be represented as $L + (L-1) + (L-2) + \cdots + 1 + 0$. This resembles the triangular numbers, and hence we can also represent the total `min_node` cost as $\frac{L(L+1)}{2}$.
4. The nested for loop inside the while loop is a bit trickier as it covers all neighbours of the current `min_node`. As we have established that every single node in the graph will be the `min_node` at some point, we can use the graph below as an example for how many times this loop would occur. Over here, we can see that A has 2 neighbours, B has 2 neighbours, C has 1 neighbour and D has 1 neighbour. This makes it evident that the

[3]The following variables will be used as shorthand throughout the analysis.
Let A = starting vertex
Let B = ending vertex
Let $S = \{P, Q, R\}$ or any other vertices to be visited along the way.
Let n = the length of the visit set S.
Let $C \in S$ (random node in S), and to clarify: $C \neq A, B$ as S does not include them

amount of times this inner for loop will run is actually just the sum of the degrees of the graph, and by the handshaking lemma, this is simply equal to twice the number of edges in the graph. Hence, the total amount of times this loop will run is $2R$.

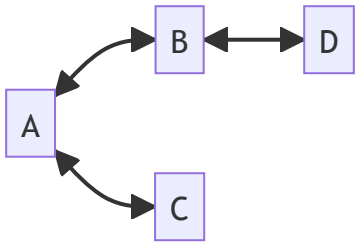

Figure 2: Sample Graph

5. Finally, inside this for loop, we call the `dist` function. As is evident from the pseudocode, this function uses the `edge_lookup_matrix` and goes over the edges between two nodes. In most practical cases, this will simply be one or two edges if multiple bus or train lines go across the same nodes. The `soonest_time_at_node` function is also an abstraction the next available bus/train time given any time at a particular node, which can possibly be implemented into a dictionary to be done in constant time. Due to these two factors, when looking at the asymptotic behaviour, this can be simplified to $O(1)$.

Now that we have considered all parts of our implementation of Dijkstra's, we can combine it to get a single cost function: $d(n) = L + L\left(\frac{L(L+1)}{2} + 2R\right) = 2LR + \frac{1}{2}L^3 + \frac{1}{2}L^2 + L$. Considering the behaviour of this function asymptotically, we can see that it would have a time complexity of $O(2LR + L^3)$, which is far from ideal and can be improved significantly (Dijkstra's can supposedly be done in $O(L + R\log L)$ with a min-priority queue).

Modified Held-Karp Time Complexity

Now that we have an established cost function, we can attempt to evaluate T_n in terms of $d(n)$. To reiterate:

$$T_n = \begin{cases} n(T_{n-1} + d(n)) & n > 0 \\ d(n) & n = 0 \end{cases}$$

$$d(n) = 2LR + \frac{1}{2}L^3 + \frac{1}{2}L^2 + L$$

Keeping this in terms of $d(n)$, we can create a table to see how this recurrence relation gets bigger as n increases.

n	T_n
0	$d(n)$
1	$2d(n)$
2	$6d(n)$
3	$21d(n)$
4	$88d(n)$
5	$445d(n)$

The working for this table is shown below, but you can easily keep going to follow the pattern for higher values of n:

$n = 0$: $T_n = d(n)$

$n = 1$: $T_n = 1(T_0 + d(n)) = 2d(n)$

$n = 2$: $T_n = 2(T_1 + d(n)) = 6d(n)$

$n = 3$: $T_n = 3(T_2 + d(n)) = 21d(n)$

$n = 4$: $T_n = 4(T_3 + d(n)) = 88d(n)$

$n = 5$: $T_n = 5(T_4 + d(n)) = 445d(n)$

Recurrence Relation

Just looking at the coefficients for a second, we have the following recurrence relation:

$$T_n = n(T_{n-1} + 1), T_0 = 1$$

It is easy to see that this recurrence relation implies that the running time for the algorithm is factorial. After all, the recurrence relation for $n!$ is $T_n = n(T_{n-1}), T_0 = 1$.

Attempting to Find an Explicit Formula

Now clearly it is of interest to solve this recurrence relation and find a non-recursive formula, and here we run into a bit of an issue. If the relation were a linear recurrence with constant coefficients or a typical divide-and conquer recurrence, it would likely be solvable by well-known methods such as telescoping or the Master Theorem, but this is not the case.

Theorem 1 While trying to find a way to solve this recurrence relation, I arrived at the conjecture that $T_n = n! + \sum_{i=0}^{n-1} \frac{n!}{i!}$, so let us try to prove it.

> For $n \in \mathbb{N}$, the number of operations used to solve an n-sized visit set TSP by the above algorithm (ignoring the cost function) satisfied the formula: $T_n = n! + \sum_{i=0}^{n-1} \frac{n!}{i!}$.

First let us work with the RHS to rearrange it a bit into a more convenient form: RHS

$= n! + \sum_{i=0}^{n-1} \frac{n!}{i!}$

$= n! + \frac{n!}{0!} + \frac{n!}{1!} + \frac{n!}{2!} + \cdots + \frac{n!}{(n-2)!} + \frac{n!}{(n-1)!}$

$= n! \times (1 + \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{(n-2)!} + \frac{1}{(n-1)!})$

Base Case When $n = 0$, the base case of the recurrence relation says that $T_0 = 1$. The above formula matches that with $T_0 = 0! \times (1 + 0) = 1$, $\therefore$ base case is true.

Induction Step Pick an arbitrary $k \in \mathbb{N}$. Assume that the theorem holds for any TSP with a visit set of size k. Thus, it is assumed that $T_k = k! \times (1 + \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{(k-2)!} + \frac{1}{(k-1)!})$.

Proof by induction requires showing the following:

$T_{k+1} = (k+1)! \times (1 + \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{(k-1)!} + \frac{1}{k!})$.

Next, we can combine the recurrence above with the induction hypothesis as follows:

LHS

$= T_{k+1}$

$= T_k(k+1) + (k+1)$ (from recurrence relation)

$= [k! \times (1 + \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{(k-2)!} + \frac{1}{(k-1)!})](k+1) + (k+1)$

$= (k+1)! \times (1 + \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{(k-2)!} + \frac{1}{(k-1)!}) + (k+1)$

$= (k+1)! \times (1 + \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{(k-2)!} + \frac{1}{(k-1)!}) + (k+1) \times \frac{(k+1)!}{(k+1)!}$

$= (k+1)! \times \left(1 + \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{(k-2)!} + \frac{1}{(k-1)!} + \frac{k+1}{(k+1)!}\right)$

$= (k+1)! \times \left(1 + \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{(k-2)!} + \frac{1}{(k-1)!} + \frac{1}{k!}\right)$

$= RHS$

Thus $T_n = n! + \sum_{i=0}^{n-1} \frac{n!}{i!}$ by the principle of mathematical induction.

Theorem 2 Looking all over the web for this, the only place I could find any reference to this sequence is here, which provides us with the relation $T_n = n! + \lfloor e \times n! \rfloor - 1$ for the coefficients. This can be rearranged to $T_n = \lfloor n!(e+1) - 1 \rfloor$, but just to be sure that this works for every case, we should probably prove it too.

> For $n \in \mathbb{Z}^+$, the number of operations used to solve an n-sized visit set TSP by the above algorithm (ignoring the cost function) satisfied the formula: $T_n = \lfloor n!(e+1) - 1 \rfloor$.

Case 1 This is the case where $n = 1$. As seen above, $T_1 = 2$ and the proposed formula predicts that $T_1 = \lfloor 1!(e+1) - 1 \rfloor = \lfloor e + 1 - 1 \rfloor = \lfloor e \rfloor = 2$. Thus, the base case holds.

Case 2 This is the case where $n > 1$. Because of the floor function, if it can be shown that the following difference is small enough, it will probably be possible to prove that this case works as well.

$$\text{Let } r_n = n!(e+1) - 1 - T_n$$

Lemma 1

> When $n > 1$, the following must be true: $r_n = \frac{1}{n+1} + \frac{1}{(n+1)(n+2)} + \frac{1}{(n+1)(n+2)(n+3)} + \cdots$

This sum looks like it might be related to the power series for e^x at $x = 1$. We already know the power series for e^x, a proof for which can be found here:

$$e^x = \frac{1}{0!} + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

It therefore follows that:

$$e = e^1 = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots$$

Since we know that $T_n = n! \times (1 + \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{(n-2)!} + \frac{1}{(n-1)!})$ from the first theorem, we can sub both the power series for e and this fact into our definition of r_n:

r_n

$= n!(e+1) - 1 - T_n$ (by definition)

$= n!(1 + \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots) - 1 - T_n$ (power series for e)

$= n!(1 + \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots) - 1 - n! \times (1 + \frac{1}{0!} + \frac{1}{1!} + \cdots + \frac{1}{(n-2)!} + \frac{1}{(n-1)!})$

$= n! \times (\frac{1}{n!} + \frac{1}{(n+1)!} + \frac{1}{(n+2)!} + \cdots) - 1$

$= (1 + \frac{1}{n+1} + \frac{1}{(n+1)(n+2)} + \cdots) - 1$

$= \frac{1}{n+1} + \frac{1}{(n+1)(n+2)} + \frac{1}{(n+1)(n+2)(n+3)} + \cdots$

$\therefore$ The lemma is true.

Lemma 2

> When $n > 1$, it is true that $r_n < \frac{1}{n+1} + \frac{1}{(n+1)^2} + \frac{1}{(n+1)^3} + \cdots = \frac{1}{n}$

This is easily proven using the first lemma:

r_n

$= \frac{1}{n+1} + \frac{1}{(n+1)(n+2)} + \frac{1}{(n+1)(n+2)(n+3)} + \cdots$ (Lemma 1)

$< \frac{1}{n+1} + \frac{1}{(n+1)(n+1)} + \frac{1}{(n+1)(n+1)(n+1)} + \cdots$

$= \frac{1}{n+1} + \frac{1}{(n+1)^2} + \frac{1}{(n+1)^3} + \cdots$

This upper bound above is in the form of an infinite geometric series with ratio $\frac{1}{n+1}$, so the usual formula of $S_\infty = \frac{a}{1-r}$ can be used: $r_n < \frac{\frac{1}{n+1}}{1-\frac{1}{n+1}} = \frac{1}{n}$.

$\therefore$ The lemma is true.

Lemma 3

If $n > 1$, $0 < r_n < 1$ must hold true.

From Lemma 1, it is clear that r_n is positive $\therefore 0 < r_n$.

Then, by Lemma 2, the following must hold: $r_n < \frac{1}{n} \leq \frac{1}{2} < 1$. $\therefore r_n < 1$.

$\therefore$ The lemma is true.

Conclusion Thus, the proof for this theorem is complete for the case $n > 1$:

By the definition of r_n, it must be true that $T_n + r_n = n!(e+1) - 1$. Since the recurrence relation set up T_n as integer and $0 < r_n < 1$ by Lemma 3, it must hold that $\lfloor n!(e+1) - 1 \rfloor = \lfloor T_n + r_n \rfloor = T_n$.

Time Complexity

Now that we have proved this works for the coefficients of the cost function, we have the formula of $T(n) = d(n)\lfloor n!(e+1) - 1 \rfloor$.The floor function here is just to deal with the difference of r_n so that we can get an integer output. Subbing in our known time complexity of $d(n)$, we get a final Big O of $O(\lfloor n!(e+1) \rfloor (2LR + L^3))$ for the original implementation of our modified Held-Karp with no caching of its own Dijkstra's outputs. Note that it should already have been obvious that the running time for this algorithm would be in factorial time from the recurrence relation itself, even before finding an explicit formula.

We have already verified that this is correct given that the recurrence relation is correct, but we can also do so by general intuition . If we look back at Part 1, we can get the time taken to run the unoptimised modified Held-Karp on our data with different n values. $(2LR + L^3)$ should be a constant for any particular predefined graph, meaning that if our Big O time complexity is correct then execution time $\propto \lfloor n!(e+1) \rfloor$[4].

n	$\frac{\text{execution time}}{\lfloor n!(e+1) \rfloor}$
5	3×10^{-5}
6	4×10^{-5}
7	3×10^{-5}
8	3×10^{-5}
9	3×10^{-5}

As we can see, this proportionality is fairly constant, so it would probably be safe to assume that the worst-case time complexity for the unoptimised modified Held-Karp algorithm would be $O(\lfloor n!(e+1) \rfloor (2LR + L^3))$, or at least something pretty close to it.

Optimised Modified Held-Karp Time Complexity

As was established in part 1, this factorial time complexity is not nearly sufficient enough for real world applications. Not only is it simply worse than brute forcing it, it makes it so calculating the Hamiltonian path with just my own friend group takes a ludicrous amount of time.

One optimisation that was made in Part 1 was the caching of Dijkstra's outputs, so that once Dijkstra's is called from one starting node, all subsequent calls to Dijkstra's will be done in $O(1)$ time. This means that the full Dijkstra's algorithm will only be called a maximum of once for every node in the graph, and then all subsequent calls will just

[4]Note that $n < 5$ would be rather unreliable due to the decimal inaccuracy of my recorded execution times (4dp)

use the cache. Since the time complexity for our Dijkstra's implementation is currently $O(2LR + L^3)$, we can simply multiply this by the amount of nodes (L) to get the worst case scenario for how long Dijkstra's takes.

This transforms our time complexity of $O(\lfloor n!(e+1) \rfloor (2LR + L^3))$ into $O(\lfloor n!(e+1) \rfloor + L(2LR + L^3))$, which doesn't *look* like that much of a difference, but it means that when looking at the asymptotic time as $n \to \infty$, we can remove the whole second term as it becomes a constant if we are not considering increasing the amount of landmarks and routes, which is much better than multiplying by this value instead.

As $n \to \infty$, not only does the 2nd term become negligible as explained above, but the floor function also doesn't make a difference because it is simply for making the output an integer number of operations. As such, it is safe to conclude that the implemented algorithm runs in factorial time for an increasing size of the `visit_set`.

In conclusion, the final algorithm from part one has a time complexity of $O(\lfloor n!(e+1) \rfloor + L(2LR + L^3))$, which means that the algorithm runs in factorial time.

Consequences of Time Complexity

As detailed in the previous section, the final time complexity of the algorithm so far is $O(\lfloor n!(e+1) \rfloor + L(2LR + L^3))$. This isn't very ideal, because simply brute forcing it would likely lead to a better worst case time complexity than the current algorithm.

Let's quickly take the example of the time complexities of our two algorithms, the one with cached Dijkstra's values and the one without. The graph/input data detailed in Part 1 has 15 landmarks, 26 routes and a `visit_set` of size 7. For these values, the unoptimised algorithm would take 77,864,700 time units and the algorithm with Dijkstra's caching would take 81,065 time units. This is over 960 times faster in the worst case scenario, but as shown in part 1, about 31 times faster in the average case. Below is a discussion on the real world consequences of this time performance difference, as well as how practical this algorithm is for real world use cases.

Revisiting Problem Requirements

This algorithm was made to solve the general problem of planning trips with friends, but more specifically the scenario where my friends decided that we want to travel in one big travel party and I am to start and end my day at my house, picking up all my friends along the way. In other words, this algorithm is designed for the real world use case of finding the shortest circuit that picks up all my friends as we travel.

Let us consider some requirements for this real world use case. By my own general estimates, most people would only have about 5 to 10 close friends that they would travel like this with. Similarly, most people live relatively close to their friends, so the case of 15 landmarks (or train stations/buses) and 26 routes (or train/bus lines) is realistic. As shown in Part 1, below is the real world performance as $n \in [0, 12]$ and $L = 15, R = 26$.

n (size of `visit_set`)	t (execution time in seconds, 4dp)
0	0.0001
1	0.0001
2	0.0001
3	0.0001
4	0.0001
5	0.0005
6	0.0060
7	0.0287
8	0.2148
9	1.6055
10	17.4555
11	171.6719
12	1750.1065

Presuming most people's friends live somewhat close to each other, even in the case where we have 10 close friends that we want to hang out with, most of them probably share "pickup points" which reduces the size of the `visit_set`.

For example, the current input data has 18 friends but a visit set of size 7! This means that in almost every case $n < 10$, and if people were using this in a mapping application like Google Maps for example to have certain pickup points along the way, this would most likely be fine, returning a result in a couple seconds at worst.

The problems start arising when this problem is scaled up more. As the algorithm is in factorial time, it scales rather terribly and has minimal improvements over brute force, if any improvements at all. The algorithm more generally is a solution for TSP with a graph that is not necessarily complete, and this can be applied to a lot more real life applications than just houses of friends. For example, if the person starting the trip was a truck driver for a logistics company rather than me, and the pickup points were necessary delivery points rather than the closest meeting points for friends, we would have a completely different scale in which the algorithm would perform very poorly. Not only would these pickup points be across a *much* larger distance, meaning the value of R will likely be much higher, but there are potentially many more pickup/dropoff points in a day than the previous scenario, causing both L and n to be greatly larger. Simply put, a factorial time complexity of $O(\lfloor n!(e+1) \rfloor + L(2LR + L^3))$ just does not scale very well for many other practical use cases besides the one explored, and even then, if the party of friends was sufficiently large, the algorithm would crawl to a halt. Looking at the example above, with just 12 pickup points the algorithm ground to a staggering half an hour of required time when tested on my machine.

Due to the fact that most users are not willing to wait more than a couple seconds for a result, the practical input sizes are $n < 10$, $L \leq 15$, $R \leq 30$. These values are taken from the input values that produced the table above while considering the time complexity of the algorithm. This is not a very big scope of possible use cases, and therefore optimisations are most definitely needed. Although this algorithm as of now is suitable to the problem's requirements, it very quickly falls apart for a "power user" or anyone else that has a different use-case in mind. Another possible alternative is using "approximate" solutions that have a better time complexity which may not provide the *most* optimal solution, but will most definitely scale better for a variety of use cases.

To conclude, this algorithm's time complexity directly influences how practically it can be used in the real world to solve the problem it is intended to solve. Users of a program as such would expect a result within seconds at most, and the practical input sizes are therefore restricted to those described above.

Appendix

Possible Optimisations

It is also worth quickly noting the possible optimisations the findings of the report above lead to.

1. The current implementation of Dijkstra's is far from optimal: the current algorithm has a cubic time complexity but with a a min priority queue this can supposedly be reduced to $O(L + R \log L)$.
2. The abstraction of `soonest_time_at_node` can be implemented as a dictionary that is accessed in constant time but is currently implemented as two for loops that makes the `dist` function more complex than necessary.
3. The biggest optimisation needed is the caching of the Held-Karp outputs, meaning that subpaths are calculated once only, and all subsequent subpaths will be read in $O(1)$ time (basically dynamic programming by definition). This should probably help the factorial time complexity, though it might be hindered by the fact that a different starting time means that the whole subpath is different which decreases how effective this optimisation is.
4. Finally, it may be worth considering approximate solutions. This being said, the scope of the problem to solve does *just* fit into the practical input sizes that the algorithm allows, but definitely limits its usefulness and real world use cases. In many times, the *best* solution is not needed, just a relatively good one.

Algorithm Pseudocode

The following is the final pseudocode reiterated from Part 1, namely for convenience while analysing, since multiple modifications were made to the initial pseudocode.

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

Main Function

```
function main(
    friends: dictionary,
    landmarks: dictionary,
    routes: dictionary,
    timetable: dictionary
):
    // global variable declarations
    concession: bool = Ask the user "Do you posses a concession card?"
    holiday: bool = Ask the user "Is today a weekend or a holiday?"
    user_name: string = Ask the user to select a friend from friends dictionary
    selected_time = Ask the user what time they are leaving

    cached_djk: dictionary = empty dictionary
    edge_lookup_matrix: matrix = |V| x |V| matrix that stores a list of edges in each entry

    // get distance of all friends from landmarks
    friend_distances: dictionary = calculate_nodes(friends, landmarks)
    visit_set: set = set of all closest nodes from friend_distances
    people_at_nodes: dictionary = all friends sorted into keys of which nodes they are closest
        to, from visit_set

    home: string = closest node of user_name

    print all friends, where they live closest to and how far away

    print out friends that would take more than 20 minutes to walk (average human walking
        speed is 5.1 km/h)

    hamiltonian_path = held_karp(home, home, visit_set, selected_time)

    print how much the trip would cost and how long it would take

    print the path of the hamiltonian_path
end function
```

Calculate Nodes

```
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
```

Held-Karp

```
function held_karp (
    start: node,
    end: node,
    visit: set<node>,
    current_time: datetime
):
    if visit.size = 0:
        djk = dijkstras(start, end, current_time)
        return djk['cost']
    else:
        min = infinity
        For node C in set S:
            sub_path = held_karp(start, C, (set \ C), current_time)
            djk = dijkstras(C, end, current_time + toMinutes(sub_path['cost']))
            cost = sub_path['cost'] + djk['cost']
            if cost < min:
                min = cost
        return min
end function
```

Dijkstra's

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

    // starting distance has to be 0
    distance[start] = 0

    // while more to still explore
    while unexplored_list is not empty:
        min_node = unexplored node with min cost
        unexplored_list.remove(min_node)

        // go through every neighbour and relax
        for each neighbour of min_node:
            current_dist = distance[min_node] + dist(min_node, neighbour, current_time +
                to_minutes(distance[min_node]))
            // 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
```

Distance Function

```
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)
end function
```

Algorithmics SAT - Friendship Network Part 3

Garv Shah

25/08/2023

Abstract

'How can a tourist best spend their day out?'. The first part of this SAT project aimed to model the Victorian public transport network and its proximity to friends' houses in order to construct an algorithm and the second part considered the time complexity of said algorithms and analysed their impact on real life use-cases. In Part 3, we will finally design an improved algorithm for the original problem using more advanced algorithm design techniques

Contents

This section of the Algorithmics SAT focuses improving the original data model and algorithm to solve the original problem more efficiently and effectively.

Throughout the analysis, note the following variables are used as shorthand:

Let F = number of friends

Let L = number of landmarks

Let R = number of routes

Suggested Improvements

From Part 2, there were various possible optimisations that became evident from the time complexity analysis. These read as follows:

1. The current implementation of Dijkstra's is far from optimal: the current algorithm has a cubic time complexity but with a min priority queue this can supposedly be reduced to $O(L + R \log L)$.
2. The abstraction of `soonest_time_at_node` can be implemented as a dictionary that is accessed in constant time but is currently implemented as two for loops that makes the `dist` function more complex than necessary.
3. The biggest optimisation needed is the caching of the Held-Karp outputs, meaning that subpaths are calculated once only, and all subsequent subpaths will be read in $O(1)$ time (basically dynamic programming by definition). This should probably help the factorial time complexity, though it might be hindered by the fact that a different starting time means that the whole sub-path is different which decreases how effective this optimisation is.
4. Finally, it may be worth considering approximate solutions. This being said, the scope of the problem to solve does *just* fit into the practical input sizes that the algorithm allows, but definitely limits its usefulness and real world use cases. In many times, the *best* solution is not needed, just a relatively good one.

The first three can be implemented and compared relatively easily, so they will be the focus of this section.

Improving Dijkstra's Implementation

As stated above, the current implementation of Dijkstra's is naïve because each iteration of the while loop requires a scan over all edges to find the one with the minimum distance, but the relatively small change of using a heap as a min priority queue allows us to find the edge with minimum distance faster. In terms of the pseudocode, this just means turning `unexplored_list` into a min priority queue, where the priority is based on the distance to the node.

Note that even though the `unexplored_list` simply appears as a priority queue in the pseudocode, for this change to be beneficial the priority queue data structure must itself be implemented efficiently, using something like a heap.

See the modified version of Dijkstra's for the pseudocode.

Heaps In most implementations (such as the Python implementation we will be testing with), the inner workings of how a min priority queue works will be abstracted and hence doesn't *need* to be worried about. Nonetheless, it is worth exploring how they are actually implemented, a popular method being min heaps!

A heap is a special tree-based data structure in which the tree is a complete binary tree. In other words, each node has exactly two children and every level will be completely filled, except possibly the deepest level. In a min heap, the parent nodes are always smaller than their children, meaning that the root node is the very smallest element.

Interestingly, since there are no gaps in the tree, the heap can actually be stored simply as an array with additional logic for adding and removing from the priority queue.

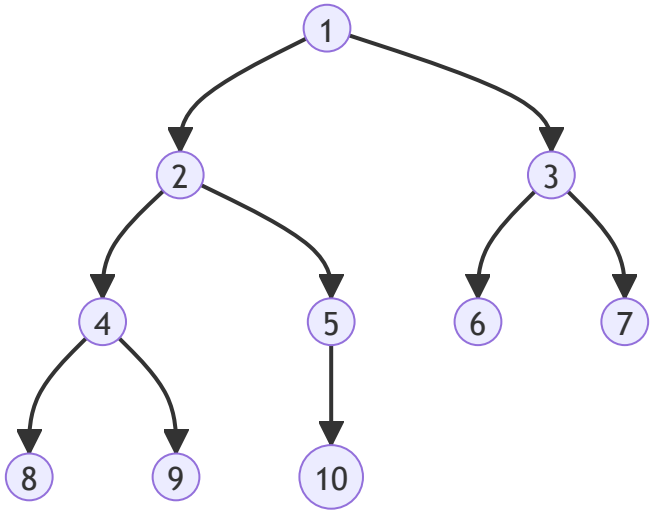

Figure 1: Complete Binary Tree

Insertion When inserting an element, it goes in the next empty spot looking top to bottom, left to right. If that's not where the element should actually go, we can "bubble it up" until it is, meaning that we can swap that element with its parent node repeatedly until it has gone up the tree enough to be in the correct position. Since it is a binary tree, we can do this in $O(\log n)$ time.

Deletion Since we would want to remove the smallest node, this would of course be the root node. Removing the root node would create an empty spot, so when we remove the root, we instead fill that with the last element added. Similar to above, since this element might not be in the right spot, we take that element and "bubble it down" until it is, this time swapping with the smaller of the two children repeatedly. Similar to above, we can do this in $O(\log n)$ time.

Improvement

Visit Set Size	Initial Algorithm (s)	Improved Dijkstra's (s)
8	1.4038	1.2842
9	3.9718	3.9315

All times are the average of 10 trials. Evidently, the improvement is slight, if any improvement at all.

Improving Distance Function

To find the `soonest_time_at_node`, the original Pythonic implementation was using a nested for loop to find when the next train/bus would arrive. This is thoroughly inefficient, namely due to the amount of times that the `dist` function is called, meaning that there would be a lot of overlap. This *could* be improved using dynamic programming, but since there is a fixed amount of time in a day (24 hours), it doesn't actually take that long to precompute this waiting time and store it along with the rest of our data. The pseudocode for this function is below:

```
time_data = dictionary of dictionaries

```

```
for line in line_data:
    for start_node in line_data[line]['timetable']:
        for current_time in every minute of a day:
            // calculate next time at node
            for arrival_time at start_node:
                if arrival_time >= current_time and is first:
                    next_time = arrival_time

            wait_time = next_time - current_time
            add wait_time to time_date
```

This produces a rather large dictionary of wait times, but the change to $O(1)$ time complexity pays off, even if space complexity is sacrificed.

Improvement

Visit Set Size	Initial Algorithm (s)	Improved Dijkstra's (s)	Improved Dist (s)
8	1.4038	1.2842	0.2746
9	3.9718	3.9315	2.2123
10	27.8881		24.4954

All times are the average of 10 trials and improvements are cumulative. The improvement seems quite large for smaller visit set sizes, but evidently this does not influence the Big O much as $\lim n \to \infty$.

Improving Held-Karp Implementation

Maybe the biggest flaw in the initial algorithm is that Held-Karp did not use dynamic programming. Due to the way Held-Karp works (explained previously), there are many overlapping problems and without the caching of these outputs, they will be calculated repeatedly unnecessarily. Since this main function is what contributes to the majority of the time complexity, improving it should make the algorithm scale better.

As we did with Dijkstra's in Part 1, caching can be done with an intermediary function, `fetch_hk`, which only runs `held_karp` if the value hasn't already been stored.

The pseudocode for this process is relatively simple and can be found below.

Improvement

Visit Set Size	Initial Algorithm (s)	Improved Dijkstra's (s)	Improved Dist (s)	Improved Held-Karp (s)
8	1.4038	1.2842	0.2746	0.0264
9	3.9718	3.9315	2.2123	0.0579
10	27.8881		24.4954	0.1460
11				0.2339
12				0.5172
13				1.2122
14				2.8075

All times are the average of 10 trials and improvements are cumulative. The improvement from this change is much better than the previous changes, likely changing our Big O time from factorial to

exponential, as seen by the roughly doubling running times. This can be verified by creating a line of best fit from the data above, which works out to be $t(n) \approx a^{n-b}$ where $a = 2.29792$ and $b = 12.7609$. This has an R^2 value of 0.9996, which provides us with a relatively high confidence that the new algorithm has $\Theta(2^n)$. According to this line of best fit, $n = 20$ would take about 7 minutes and 53 seconds, while $n = 30$ would take almost 3 weeks.

It is worth noting that although this does improve the time complexity by a large factor, the cache also takes up a lot of space, making the space complexity worse. This tradeoff is quite good in most cases since modern devices have plenty of memory and storage, but in the case that space complexity is a constraint, this may be an unideal optimisation.

Practicalities of an Exact Algorithm

Though the algorithm has seen a dramatic improvement from factorial time to likely exponential time, it still maintains a lot of the issues that the previous version possessed. Namely, because exponential time still does not scale very well, the practical input size for n is still very limited, changing from about $n \leq 9$ to $n \leq 14$.

As stated in Part 2, this is mostly sufficient for the specific use case of the problem outlined in most cases since the amount of friends people would hang out with in this fashion is intrinsically small, as it only applies itself to close friends. Because of this, even if someone does have a large amount of close friends, it is unlikely that the visit set that gets computed is larger than 14 (the current input data has 18 friends but a visit set of size 7). As such, for the practical cases of this specific problem, the exact algorithm is sufficient, and also works for adjacent scenarios such as mapping applications (Google Maps, etc.) wanting to have certain pickup points along the way.

The algorithm begins to become impractical once the problem is scaled up more as a general solution for the TSP. For example, if a truck driver for a logistics company wanted an optimal route given a list of pickup points, this would very quickly surpass the practical limit of $n \leq 14$, and the graph would be much larger as well. In wider applications like this, using an exact algorithm is simply not useful, and we would rather want paths that have a "small enough" cost but have a feasible time complexity. This is where we get into the realm of approximate algorithms.

Tractability

It is important to note that the problem that was initially described can simply be generalised as the Travelling Salesman Problem, which is famously NP-Hard meaning that there is no known polynomial time solution for the problem.

Due to the fact that our final exact algorithm implementation had its execution time double every time n was increased by 1, it is safe to assume that the algorithm runs in exponential time at best, meaning that it is still considerably intractable for large inputs due to the exponential growth.

From this, it is clear that the problem does not become tractable based on the above implementation, and it will be hard to make an exact algorithm that is much faster. This is why approximate algorithms are worth considering, namely those that have performance guarantees of worst cases that are within a certain factor of the minimal cost solution. They provide a trade-off between speed and optimality, and while they make the problem more tractable than exact algorithms, they do not make it completely tractable due to their approximate nature and how they do not always produce the optimal solution.

Approximate/Heuristic Algorithms

The general idea of most approximation algorithms is we can start with an initial candidate solution and then keep making changes to see if we can get better. The initial candidate solution need not be good, but it would certainly help produce results closer to the global optimum after a certain amount of iterations.

One of the most intuitive ideas to generate an initial candidate solution would be to visit the closest node in the visit set from any given node, and this can more formally be described as the Nearest Neighbour Heuristic.

Nearest Neighbour Heuristic

The Nearest Neighbour (NN) algorithm is a greedy (and somewhat naïve) approach where the closest unvisited city is selected as the next destination. This method produces a reasonably short route, but usually not the optimal one. The informal steps of this approximate algorithm are listed below:

1. Mark every vertex as unvisited.
2. Set the starting vertex as the current vertex **u**, marking it as visited.
3. Find the shortest outgoing edge from **u** to an unvisited vertex **v**.
4. Set **v** as the current vertex **u** and mark it as visited.
5. If all vertices have been visited, terminate, if not, go to step 3.

This is a very simple algorithm, but as is the case with most greedy approaches, it can quite easily miss shorter routes. For this specific use case step 3 may cause a few issues in terms of time complexity, as unlike the normal TSP, our graph is not complete. This means that at this step, we would need to run Dijkstra's at every single node in the graph and then sort them to find the shortest path, which is inefficient.

To make this slightly faster we *could* simply choose the first unvisited node in the visit set to go to, but that would still require Dijkstra's to run at every node to find a path, meaning that only the time spent sorting would be saved (which is minimal since Dijkstra's will already have them sorted from the min heap). The problem with this approach is that it will produce a less optimal solution, causing the algorithm to have to spend a longer amount of time improving the solution in the simulated annealing phase. This means that it is a bit of a tradeoff, and for now the shortest node will be chosen.

To avoid using Dijkstra's at all, it is worth considering candidate solutions based on the MST, such as those created by Christofides' Algorithm, which may turn out to be faster. This can be further considered to optimise the algorithm, but for simplicity's sake, the NN Heuristic will be continued with.

Below is the pseudocode to generate an initial candidate solution. Note that in this pseudocode, `fetch_djk` only has the input of the starting node and visit set and returns the path to the closest node in the visit set, so it is a slightly modified version of the `fetch_djk` outlined above.

```
// creates a canditate solution using the NN Heuristic
function canditate_solution (
    start: node,
    end: node,
    visit: set of nodes,
    current_time: datetime,
):
    path = [start]
    current_vertex = start
    cost = 0

    while len(visit) != 0:
        closest_node = fetch_djk(current_vertex, visit, current_time)
        path.add(closest_node)
        cost += closest_node.cost
        visit.remove(closest_node)
        current_vertex = closest_node

    // go back to the end node
    closest_node = fetch_djk(current_vertex, end, current_time)
    path.add(closest_node)
    cost += closest_node.cost

    return {'path': path, 'cost': cost}
end function
```

Pairwise Exchange

Once we have an initial candidate solution that has a reasonable cost for the traversal, a natural question to ask is "how can we make it better?" More specifically, it is worth considering how we could make modifications to generate a better solution.

One way to do this is random swapping, where we randomly pick two cities in the current tour order, and swap them. The goal is to see if these random swaps will ever create a lower cost tour, and if they do, we can accept the new solution. This is a form of the Hill Climbing heuristic, where we keep moving around the sample space to see if we can improve our solution at all.

A slightly more sophisticated technique than randomly swapping the nodes is a method called Pairwise Exchange or 2-opt. The main idea is that we can select any two edges and reconfigure them in the only other way possible with the hopes that this may result in a lower cost tour.

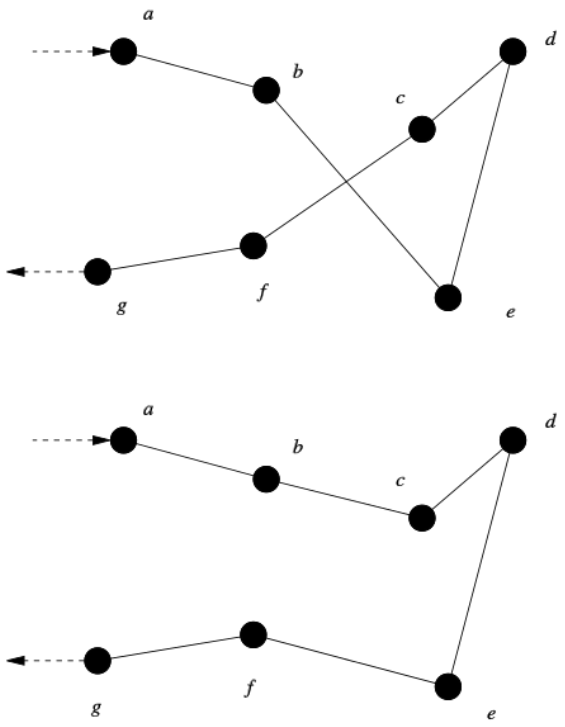

Figure 2: Demonstration of the 2-opt Technique

For example, in the diagram above, it can be seen that the pairs $b - e$ and $c - f$ cross over each other, so the edges can be swapped so that they do not.

More simply, when imagined as a one dimensional array, this could be viewed as the following transformation where we simply reversed the order of the path $e \leftrightarrow d \leftrightarrow c$:

1. $a \leftrightarrow \mathbf{b} \leftrightarrow \mathbf{e} \leftrightarrow d \leftrightarrow \mathbf{c} \leftrightarrow \mathbf{f} \leftrightarrow g$
2. $a \leftrightarrow \mathbf{b} \leftrightarrow \mathbf{c} \leftrightarrow d \leftrightarrow \mathbf{e} \leftrightarrow \mathbf{f} \leftrightarrow g$

In essence, this "untangles" our candidate solution and can go through all the possible edge combinations much faster than simply randomly switching nodes (which has a much lower chance of being any better).

It is worth noting that the 2-opt technique (where 2 edges are selected and reconfigured) can actually

be extended to any number of edges, known as k-opt for k edges. It might be worth working with a larger amount of edges (3-opt for example), but for simplicity's sake, 2-opt will the one continued with.

The above notion of reversing the order of a certain path can be expanded upon to develop our pseudocode. The informal steps for this process are listed below:

Let u and v be the first vertices of the edges that are to be swapped. Let tour be an array of vertices that defines our candidate path. 1. Add all vertices up to and including u in order. 2. Add all vertices after u up to and including v in reverse order. 3. Add all vertices after v in order.

In the example above, u would have been b and v would have been c.

This basic logic can be combined with the Hill Climbing Heuristic to provide a simple way to improve the initial candidate solution. Here, the `calculate_cost` function would simply add up the cost of traversing the graph in the input order, using Dijkstra's at every vertex.

```
function pairwise_swap (
    u: integer,
    v: integer,
    path: path of nodes
):
    new_tour = []

    for i in [0, u]:
        new_tour.add(path[i])
    for i in [v, u):
        new_tour.add(path[i])
    for i in (v, len(path)]:
        new_tour.add(path[i])

    return new_tour
end function

function calculate_cost (
    path: path of nodes,
    current_time: datetime
):
    cost = 0
    time = current_time

    for i from 0 to len(path) - 1:
        djk = fetch_djk(path[i], path[i + 1], current_time)
        cost += djk['cost']
        time += djk['cost'] number of minutes

    return cost
end function

function hill_climbing (
    candidate: path of nodes,
    current_time: datetime,
    fail_count: int = 0
):
    if fail_count < 200:
```

```
        cost = calculate_cost(candidate, current_time)
        u = random number from 1 to len(candidate) - 1 inclusive
        v = random number from u to len(candidate) - 1 inclusive

        new_tour = pairwise_swap(u, v, candidate)
        new_cost = calculate_cost(new_tour, current_time)

        if new_cost <= cost:
            // new cost is better/equal -> accept
            return hill_climbing(new_tour, current_time, 0)
        else:
            // new cost is worse -> go again
            return hill_climbing(candidate, current_time, fail_count + 1)
    else:
        return candidate
end function
```

Note that the above range of u and v values has been chosen to prevent them from referring to the start or end of the tour, since in our particular use case we would like to force the tour to start and end at particular locations

Simulated Annealing

One of the problems with the above solution is that it will quite easily get stuck on a local minimum. Demonstrated by the graph below, the Hill Climbing Heuristic is blind to anything besides its local vicinity. As such, there may be an overall better solution, but not one that can be achieved by constantly improving the current candidate solution. In other words, sometimes things have to get worse before they get better, especially for the TSP.

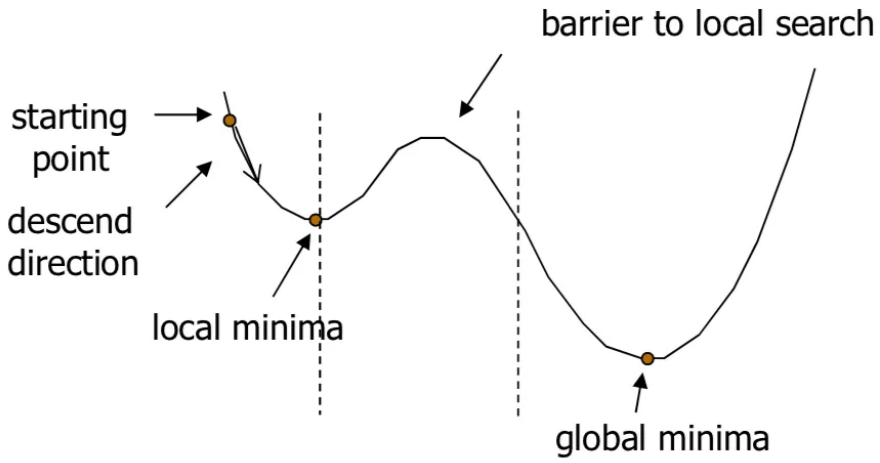

Figure 3: Example of the Limitations of Hill Climbing

Currently, once the Hill Climbing algorithm is implemented in Python, it produces a somewhat suboptimal result. It is hardcoded to terminate after it has had 200 consecutive iterations that have

seen no improvement. Sometimes, it can terminate on a relatively good result, but in other cases it gets stuck on much more sub-par candidates. This can be demonstrated by the two paths bellow, both of which the Hill Climbing algorithm terminated on.

```
The cost has been improved from 234.0 to 227.0
['Brandon Park', 'Oakleigh', 'Wheelers Hill Library', 'CGS WH', 'Chadstone',
    'Caulfield', 'Flinders Street', 'Camberwell', 'Parliament', 'Melbourne
    Central', 'Brighton Beach', 'Richmond', 'Mount Waverley', 'Glen Waverley',
    'Brandon Park']
```

```
The cost has been improved from 277.0 to 270.0
['Brandon Park', 'CGS WH', 'Glen Waverley', 'Mount Waverley', 'Camberwell',
    'Chadstone', 'Caulfield', 'Brighton Beach', 'Flinders Street', 'Melbourne
    Central', 'Parliament', 'Richmond', 'Oakleigh', 'Wheelers Hill Library',
    'Brandon Park']
```

Simulated Annealing is a concept that builds off of this idea of possibly selecting a worse solution to hopefully get to the global optimum. Namely, it tries to explore as much of the search space as possible at the start (by being more likely to select worse candidates) and then gradually reduces this chance so that it can converge on a better solution.

The logic behind this is quite similar to Hill Climbing:

1. Start with a candidate solution, from a previous algorithm or just a random tour.
2. Modify this candidate by trying to apply some tour improvements, in this case 2-opt.
3. Decide whether to accept the new solution or stay with the old one.

The key difference here is step 3. In both algorithms, if the new tour's cost is lower than the previous one, we will always accept it. If the cost is more than the current solution, with some probability, we will actually accept the higher cost solution but this probability will decrease over time.

How this probability is determined is mostly based on a parameter called the "Temperature" T. At the start we will initialise this to a high value, and a higher temperature means we are more likely to select a worse solution. Any $T \in [0, 1]$ will work, but we want to gradually reduce our temperature over time, so that it can influence some probability function.

There are usually three main types of temperature reduction functions, where α is the factor by which the temperature is scaled after n iterations:

1. Linear Reduction Rule: $T = T - \alpha$
2. Geometric Reduction Rule: $T = T \times \alpha$
3. Slow-Decrease Rule[1]: $T = \frac{T}{1+\beta T}$

Each of these reduction rules decreases the temperature at a different rate, so they may be better for different use cases. For now, we will settle upon the Geometric Reduction Rule (as it is the most common).

Starting at the initial temperature, the algorithm will loop through n iterations and then decrease the temperature according to the selected temperature reduction function at the end of every iteration. This loop will stop once the terminating condition is reached, generally some low cutoff temperature where we have determined an acceptable amount of the search space has been explored.

[1]This rule is not often used, but β is a different constant that we'll get to later.

Finally, within each iteration, we will use our temperature, the old cost and the new cost to determine whether we accept the new solution or not. This follows the formula below where $\Delta c =$ new cost $-$ old cost:

$$P = \begin{cases} 1 & \Delta c \leq 0 \\ e^{-\beta \Delta c / T} & 0 < \Delta c \end{cases}$$

To demonstrate, if the new cost is less than or equal to the old cost, the new cost will always be accepted. If on the other hand the new cost is greater, then we *might* pick it based on the formula shown above. This equation is inspired by the formula for the energy released by metal particles as they cool down from thermodynamics: $P(\Delta E) = e^{-\frac{\Delta E}{k * t}}$. This process is known as annealing, hence the name of the algorithm! Borrowing this equation from physics turns out to be quite elegant, giving us a probability distribution known as the Boltzman distribution.

It is worth noting the different parameters that can be tuned, and the effectiveness of the algorithm depends on the choice of these parameters:

1. β - Normalising Constant The choice of this constant is dependent on the expected variation in the performance measure over the search space, If the chosen value of β is higher, the probability of accepting a solution is supposedly also higher in later iterations. In our use case, we can simply play around with this number and see if it changes anything!
2. T_0 - Initial Temperature This is simply the temperature we start with, and should be relatively close to one so that we accept a lot of new solutions at the start. For now, we will set $T_0 = 0.98$.
3. α - Temperature Scaling Factor As explained above, depending on the temperature reduction function chosen, α will reduce it at a different rate. Low α values restrict the search space faster, so we can choose $\alpha = 0.85$ for now.

The number of iterations before the temperature is updated can also be played around with, for now this will be set to 5. Also, the cutoff terminating temperature can also be set to allow the algorithm to search for longer.

The above should demonstrate the main weakness of simulated annealing: there are a lot of tunable parameters that vastly influence the performance of the algorithm. If our input data is very sparse, the algorithm may perform much worse for certain use cases. Nonetheless, it is most definitely an improvement over the Hill Climbing algorithm as it does not increase time complexity or space complexity, but it does provide a more accurate output.

Below is the pseudocode that summarises the above discussion:

```
function acceptance_probability (
    old_cost: number,
    new_cost: number,
    beta: number,
    temp: number
):
    c = new_cost - old_cost

    if c <= 0:
        return 1
    else:
        return e**((-beta * c)/temp)
end function

```

```
function simulated_annealing (
    candidate: path of nodes,
    current_time: datetime,
):
    // parameters to fiddle with
    temp = 0.98
    min_temp = 0.00001
    temp_change = 5
    beta = 1.2
    alpha = 0.85

    old_cost = calculate_cost(candidate, current_time)

    while temp > min_temp:
        for n from 1 to temp_change:
            u = random number from 1 to len(candidate) - 1 inclusive
            v = random number from u to len(candidate) - 1 inclusive

            new_tour = pairwise_swap(u, v, candidate)
            new_cost = calculate_cost(new_tour, current_time)

            ap = acceptance_probability(old_cost, new_cost, beta, temp)

            if ap > random float from 0 to 1:
                candidate = new_tour
                old_cost = new_cost

        temp *= alpha

    return candidate
end function
```

Normalising Function Something that may have become apparent when viewing the above examples is how the paths generated by this approximate solution are somehow much shorter than those generated by Held-Karp. This is due to the fact that the implementation of Held-Karp is not restricted to only visiting each node once, whereas the approximate algorithms are. Due to this, we get some interesting behaviour that needs to be accounted for.

```
['Brandon Park', 'Oakleigh', 'CGS WH', 'Wheelers Hill Library', 'Caulfield',
    'Flinders Street', 'Melbourne Central', 'Parliament', 'Glen Waverley',
    'Chadstone', 'Brighton Beach', 'Camberwell', 'Mount Waverley', 'Richmond',
    'Brandon Park']
```

The above is a path generated by the Hill Climbing algorithm. The issue to note is that it advises the user to go from Glen Waverley to Chadstone, but there is no edge between them for this to happen. Since the algorithms have been using Dijkstra's to go to any other node, it has in essence been treating our tour as a complete graph, even though it is not. As such, the edges in between these locations need to be added in again.

This is quite simple to do, and is similar to the `calculate_cost`, except the paths are added instead of the costs.

```
function normalise_path (
```

```
    path: path of nodes,
    current_time: datetime
):
    return_path = []
    time = current_time

    for i from 0 to len(path) - 1:
        djk = fetch_djk(path[i], path[i + 1], current_time)
        time += djk['cost'] number of minutes
        // this is to prevent the last and first item double up
        return_path += everything in djk['path'] except last item

    return_route.add(last item in route)

    return cost
end function
```

Final Solution

The problem these algorithms were set out to solve is a specific application of the TSP: how could the shortest closed walk be found that picks up all my friends as we travel around the city?

The initial approach to solve this problem used the concepts of dynamic programming to recursively split up the larger problem into smaller overlapping subproblems. Unfortunately, because the number of subpaths increases exponentially as the size of the visit set increased, it was demonstrated that even though an exact algorithm may provide an optimal solution, intractable problems like the TSP may require a better time complexity in a trade-off for accuracy.

The approaches for the approximate solutions have followed two main phases: - Generate a possible candidate solution. - Improve the candidate using some optimisation algorithm.

The Nearest Neighbour heuristic was used to generate the initial candidate, simply travelling to the closest node remaining in the visit set until a closed walk has been achieved. This was then later improved upon by processing this candidate through both the Hill Climbing and Simulated Annealing algorithms.

In regard to the performance of Simulated Annealing (SA) vs Hill Climbing (HC), it seems that the output of the former is heavily dependent on the parameters set. Whereas HC produced results in a relatively large range, SA could be tuned to consistently provide the same "good" results every time or if the parameters were not optimal, a completely rubbish result every time.

For example, with $T_0 = 0.98, \beta = 4, \alpha = 0.9$ and the 5 iterations before updating the temperature, SA consistently produced a hamiltonian path that would take 254 minutes to traverse. HC was more inconsistent, outputting 274 initially, 281 next and struck gold with the last try with 237. Surprisingly though, the difference between Hill Climbing and Simulated Annealing doesn't seem to be vast for this particular input graph, and SA can simply be viewed as a more tunable and adjustable version of HC to be able to produce a more consistent result.

When this was changed to simply be the visit set that the friends reside at, the output for both HC and SA was as follows:

```
1 Final candidate cost is 143.0
2 Final candidate path is ['Brandon Park', 'Wheelers Hill Library', 'CGS WH', 'Glen
    Waverley', 'Mount Waverley', 'Richmond', 'Camberwell', 'Richmond', 'Flinders
    Street', 'Caulfield', 'Oakleigh', 'Brandon Park']
```

Nonetheless, neither of them are able to find the true optimal path that Held-Karp creates:

```
1 Final candidate cost is 130.0
2 Final candidate path is ['Brandon Park', 'Wheelers Hill Library', 'CGS WH', 'Glen
    Waverley', 'Mount Waverley', 'Richmond', 'Flinders Street', 'Caulfield',
    'Oakleigh', 'Richmond', 'Camberwell', 'Richmond', 'Oakleigh', 'Brandon Park']
```

This could simply be because 2-opt does not provide the required permutations to be able to reach the optimal path, but still demonstrates the required tradeoff between approximate solutions and exact algorithms, a tradeoff of time vs accuracy.

Comparison of Solutions

Design Features As discussed above, Held-Karp (the exact algorithm) used the principle of dynamic programming to split the larger problem into instances of the similar overlapping subproblems that can be solved recursively. By utilising the fact that every subpath of a path of minimum distance is itself of minimum distance, we can recursively reduce the size of the visit set by one and solve for the

smaller cases. In this case, due to the TSP's intractability, this only decreases the time complexity from factorial to exponential, saving time by ensuring that subpaths are not re-calculated.

On the other hand, the combination of algorithms that produce the approximate solutions operate based off a variety of design principles.

The initial candidate solution generated by the NN Heuristic uses a greedy design pattern to find a possible path. This design pattern does not work with many problems (including the TSP) because sometimes things have to get worse for an overall better result.

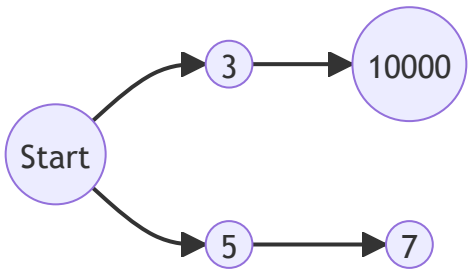

Figure 4: Demonstration of Why Greedy Algorithms Fail

Demonstrated above, the greedy design feature would select "3" as it is the best option visible at the time, but will end up selecting a far worse solution that could easily be avoided with some intuition for what comes afterwards.

Nonetheless, the greedy design pattern in the NN heuristic generally produces a somewhat viable candidate, that is then improved upon by certain Generate and Test algorithms.

One such algorithm is Hill Climbing, which refers to a type of local search optimisation technique that provides an iterative way to make incremental changes to a candidate and proceed if an improvement has been found.

Simulated Annealing expands upon this idea by using a probabilistic technique to decide if we accept an incremental change or not. Both these local search algorithms allow for an exploration of adjacent solutions that help find an improved solution in a tractable way.

The difference between the two approaches and their design patterns lies between the intended output. The dynamic programming approach guarantees a correct output, but since the requirements are slightly different for the approximate algorithms, a wider range of design techniques are available (such as using random probability or the Generate and Test pattern) that can get us closer to a better solution, even if it produces a non-deterministic non-optimal result.

Coherence Overall, Held-Karp is far more of a consistent and logical solution. Since the exact algorithm is inherently deterministic, it is always guaranteed to produce the same optimal result consistently. In contrast, the NN algorithm's performance can vary widely depending on the arrangement of nodes and both the optimisation algorithms use probability to pick u and v values. Simulated annealing is also non-deterministic (∴ probabilistic), meaning that it is nowhere near as consistent as Held-Karp. That being said, Simulated Annealing does seem to converge consistently on the same or similar local optima based on its input parameters, so we can render it more coherent than Hill Climbing but much less so than Held-Karp.

The influence of this difference in consistency between the two approaches on the real world applications is key to deciding which approach is better. Exact algorithms would be preferred in scenarios where

predictability and repeatability are crucial. For example, in scientific research studies on geographical data that is static, the superior coherence of Held-Karp would mean that the study is repeatable and verifiable by peers. On the other hand, the lower consistency of SA and HC are not necessarily disadvantageous in real world applications, because they can provide more flexibility and adaptability. Instead of providing only one solution, they provide many good candidates that the user can consider between. This flexibility would be ideal for larger operations such as a logistics company, where the clients and pickup points are very actively changing, and alternative routes need to be provided in case the algorithm does not account for real world disturbances such as road closures.

Fitness for Problem In terms of fitness for the problem, it would be safe to say that the exact algorithm would be preferred for the initial problem described. Even though Held-Karp would have a larger space complexity (due to all the subpaths that need to be stored), a typical user's phone will have plenty of storage such that space should not be too much of an issue. The inefficient time complexity of the algorithm mostly relates to how it scales to larger visit set sizes, anything below $n = 14$ is barely noticeable to the typical user. Since most people will not be intending to travel in this fashion with such a large number of friends, it would likely be preferred to use the exact algorithm as it provides the optimal solution. This being said, Held-Karp is somewhat inflexible, especially when it comes to frequently changing data. As it only provides one path and one path only, it could be a bit of an issue when it does not account for certain data such as a bus replacement (very common around Victoria). As such, it might be best to use a combination of both in an application, defaulting to the modified Held-Karp but switching over to the approximate algorithms once $n > 13$ or more solutions are requested.

Efficiency & Time Complexity As established above, the improved Held-Karp algorithm maintains an exponential time complexity, similar enough to $O(2^n)$ that we can use this simplified version to come to more clear conclusions.

Going through the pseudocode for the approximate algorithms, the algorithm to find a candidate solution is run first. In this case, this would be the Nearest Neighbour heuristic, which runs the following code for every node in the visit set (of size n)

```
closest_node = fetch_djk(current_vertex, visit, current_time)
path.add(closest_node)
cost += closest_node.cost
visit.remove(closest_node)
current_vertex = closest_node
```

Since it runs Dijkstra's at every node, our time complexity for NN will just be $n\times$Dijkstra's Time Complexity. If we presume that the above optimisations for Dijkstra were effective then this would be at $O(L+R\log L)$ (the generally accepted time complexity for Dijkstra's using min heaps), but even if this was not the case, we would have a time complexity of $O(L^2)$. This provides an NN time complexity of $O(n \times L^2)$.

In terms of Hill Climbing, during every iteration i, the algorithm runs the following pseudocode:

```
cost = calculate_cost(candidate, current_time)
u = random number from 1 to len(candidate) - 1 inclusive
v = random number from u to len(candidate) - 1 inclusive

new_tour = pairwise_swap(u, v, candidate)
new_cost = calculate_cost(new_tour, current_time)

if new_cost <= cost:
    // new cost is better/equal -> accept
    Go again with the new tour
```

```
else:
    // new cost is worse -> go again
    Go again with the same tour
```

First, the cost of the candidate is evaluated. This requires us to run Dijkstra's on each node in the visit set again, but since the output of Dijkstra's is cached, this would actually only take $O(n)$ time. Next, a pairwise swap is done, which adds every node in the visit set to a new array in a differing order which is also in $O(n)$ time. Finally, the cost is calculated again, leaving us with a final total of $O(3n)$. Overall, this means that this process is done in linear time for i iterations, leaving a final time complexity of $O(i \times n)$.

Simulated Annealing has the exact same time complexity as Hill Climbing because the only major difference is if a candidate solution is accepted or not and this is done in $O(1)$ time because the time complexity of selecting a random number is $O(1)$.

This leaves us with a final time complexity of $O(n \times L^2 + i \times n) = O(n(L^2 + i))$.

Tractability & Implications

As discussed above, the time complexity for the exact algorithm is effectively $O(2^n)$ and the time complexity for the approximate algorithms is $O(n(L^2 + i))$ where n is the size of the visit set, L is the number of landmarks in the graph overall and i is the amount of times that the optimisation algorithm will iterate. i will typically be a constant and can therefore be ignored and for the same input graph (this assumption was made for the simplification of Held-Karp too) L^2 will be constant as well.

In effect, this means that for the same input graph, the time complexities we are looking at are $O(2^n)$ vs $O(n)$ as the visit set size increases by a constant factor. The vast difference between these two time complexities shows how easily approximate solutions can be derived in polynomial time, which helps make this problem more tractable. Namely, this demonstrates that the problem of finding a solution to the TSP within a set factor of the optimal solution is a tractable one, even if finding the *actual* optimal solution is not.

This has many implications for the real world applications of the broader version of this problem. Though the discussion above concluded that the exact algorithm would be superior for the initial specified problem, the tradeoff of lower accuracy for an improved time complexity can be beneficial to many use-cases. Below is a list of applications that would be better suited to either type of algorithm:

Exact Algorithm:

- An exact algorithm would be well suited to static non-changing data where time is not much of a concern but the best solution is required. In a scenario where large freeways need to be built to visit a few key cities, the geographical data remains mostly static since the overall terrain does not change suddenly, but an inefficient solution could cost millions. Similarly, in wartime where tunnels and bunker networks need to be built that connect everyone to a few key locations, a few extra kilometres could result in hundreds of lost lives. In cases like this, provided that the number of key locations is sufficiently small, users would likely not mind waiting for a more optimal output.

Approximate Algorithm:

- As discussed previously, an approximate algorithm would be very well suited to logistics/trucking companies that have to move a lot of shipments and goods across the country fast. The nature of real world companies means that clients would appear and disappear on a daily basis, and there are always new locations to be delivered to or picked up from. Since the input graph is dynamically changing, an exact solution would be very quickly out of date and an $O(n)$ time

complexity would be preferred over the intractable $O(2^n)$ complexity since the amount of pickup points would simply be so large.

- An approximate algorithm would be well suited to data routing, specifically peer to peer networks that want to connect a large group of people. For example, a P2P video conferencing call would need to find a sufficiently small closed walk to ensure that the call has minimal delay. Since the input data for this case would be constantly changing (people leaving and joining with variable bandwidths), it would need to be run very often, and an intractable solution would not suffice.

This being said, most applications would be better suited to a combination of both. With a small number of nodes in the visit set, the intractability of finding an exact solution is not much of an issue, as the speeds are virtually instant anyway, but anything above about 15 to 20 nodes will render the computational time to be prohibitive. As such, for most real world applications, it makes more sense to use a combination of both the algorithms and switch over once the input size has exceeded the practical time constraints a layman user would expect. Such is the case with the initial solution, as described above.

Appendix

Initial Pseudocode

The following is the final pseudocode reiterated from the previous 2 parts, namely for convenience while analysing, since multiple modifications were made to the initial pseudocode. A Python implementation of this pseudocode can be found here.

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

```
function main(
    friends: dictionary,
    landmarks: dictionary,
    routes: dictionary,
    timetable: dictionary
):
    // global variable declarations
    concession: bool = Ask the user "Do you posses a concession card?"
    holiday: bool = Ask the user "Is today a weekend or a holiday?"
    user_name: string = Ask the user to select a friend from friends dictionary
    selected_time = Ask the user what time they are leaving

    cached_djk: dictionary = empty dictionary
    edge_lookup_matrix: matrix = |V| x |V| matrix that stores a list of edges in
        each entry

    // get distance of all friends from landmarks
    friend_distances: dictionary = calculate_nodes(friends, landmarks)
    visit_set: set = set of all closest nodes from friend_distances
    people_at_nodes: dictionary = all friends sorted into keys of which nodes they
        are closest to, from visit_set

    home: string = closest node of user_name

    print all friends, where they live closest to and how far away

    print out friends that would take more than 20 minutes to walk (average human
        walking speed is 5.1 km/h)

    hamiltonian_path = held_karp(home, home, visit_set, selected_time)

    print how much the trip would cost and how long it would take

    print the path of the hamiltonian_path
end function
```

```
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
```

```
function held_karp (
    start: node,
    end: node,
    visit: set<node>,
    current_time: datetime
):
    if visit.size = 0:
        djk = fetch_djk(start, end, current_time)
        return djk['cost']
    else:
        min = infinity
        For node C in set S:
            sub_path = held_karp(start, C, (set \ C), current_time)
            djk = fetch_djk(C, end, current_time + toMinutes(sub_path['cost']))
            cost = sub_path['cost'] + djk['cost']
            if cost < min:
                min = cost
        return min
end function
```

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

```

```
    // starting distance has to be 0
    distance[start] = 0

    // while more to still explore
    while unexplored_list is not empty:
        min_node = unexplored node with min cost
        unexplored_list.remove(min_node)

        // go through every neighbour and relax
        for each neighbour of min_node:
            current_dist = distance[min_node] + dist(min_node, neighbour,
                current_time + to_minutes(distance[min_node]))
            // 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 {
        'distances': distance,
        'predecessors': predecessor,
    }
end function
```

```
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, current_time)

    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
```

```
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)
end function
```

Modified Exact Algorithm Pseudocode

Below is the final pseudocode for the exact algorithm, based on Held-Karp. A Python implementation of the following pseudocode can be found here.

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

```
function main(
    friends: dictionary,
    landmarks: dictionary,
    routes: dictionary,
    timetable: dictionary
):
    // global variable declarations
    concession: bool = Ask the user "Do you posses a concession card?"
    holiday: bool = Ask the user "Is today a weekend or a holiday?"
    user_name: string = Ask the user to select a friend from friends dictionary
    selected_time = Ask the user what time they are leaving

    cached_djk: dictionary = empty dictionary
    edge_lookup_matrix: matrix = |V| x |V| matrix that stores a list of edges in
        each entry

    // get distance of all friends from landmarks
    friend_distances: dictionary = calculate_nodes(friends, landmarks)
    visit_set: set = set of all closest nodes from friend_distances
    people_at_nodes: dictionary = all friends sorted into keys of which nodes they
        are closest to, from visit_set

    home: string = closest node of user_name

    print all friends, where they live closest to and how far away

    print out friends that would take more than 20 minutes to walk (average human
        walking speed is 5.1 km/h)

    hamiltonian_path = fetch_hk(home, home, visit_set, selected_time)

    print how much the trip would cost and how long it would take

    print the path of the hamiltonian_path
end function
```

```
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
```

```
function held_karp (
    start: node,
    end: node,
    visit: set<node>,
    current_time: datetime
):
    if visit.size = 0:
        djk = fetch_djk(start, end, current_time)
        return djk['cost']
    else:
        min = infinity
        For node C in set S:
            sub_path = fetch_hk(start, C, (set \ C), current_time)
            djk = fetch_djk(C, end, current_time + toMinutes(sub_path['cost']))
            cost = sub_path['cost'] + djk['cost']
            if cost < min:
                min = cost
        return min
end function
```

```
cached_hk = dictionary of list -> dict

function fetch_hk (
    start: node,
    end: node,
    visit: set of nodes,
    current_time: datetime,
):
    // unique identifier
    name = start + '-' + end + visit set + '@' + current_time
    if cached_hk[name] does not exists:
        cached_hk[name] = held_karp(start, end, visit, current_time)
    return cached_hk[name]
```

```
end function
```

```
function dijkstras (
    start: node,
    current_time: datetime
):
    unexplored = empty min priority queue of nodes based on distance

    // Set all node distance to infinity
    for node in graph:
        distance[node] = infinity
        predecessor[node] = null
        unexplored.add(node)

    // starting distance has to be 0
    distance[start] = 0

    // while more to still explore
    while unexplored is not empty:
        min_node = unexplored.minimum_node()
        unexplored.remove(min_node)

        // go through every neighbour and relax
        for each neighbour of min_node:
            current_dist = distance[min_node] + dist(min_node, neighbour,
                current_time + to_minutes(distance[min_node]))
            // 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 {
        'distances': distance,
        'predecessors': predecessor,
    }
end function
```

```
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, current_time)
```

```

    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
```

```
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:
        wait_time = wait time from data (precomputed)
        distances.add(edge.weight + wait_time)

    return min(distances)
end function
```

Approximate Algorithm Pseudocode

Below is the final pseudocode for the approximate algorithm, using Simulated Annealing. A Python implementation of the following pseudocode can be found here.

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

```
function main(
    friends: dictionary,
    landmarks: dictionary,
    routes: dictionary,
    timetable: dictionary
):
    // global variable declarations
    concession: bool = Ask the user "Do you posses a concession card?"
    holiday: bool = Ask the user "Is today a weekend or a holiday?"
    user_name: string = Ask the user to select a friend from friends dictionary
    selected_time = Ask the user what time they are leaving

    cached_djk: dictionary = empty dictionary
    edge_lookup_matrix: matrix = |V| x |V| matrix that stores a list of edges in
        each entry

    // get distance of all friends from landmarks
    friend_distances: dictionary = calculate_nodes(friends, landmarks)
    visit_set: set = set of all closest nodes from friend_distances
    people_at_nodes: dictionary = all friends sorted into keys of which nodes they
        are closest to, from visit_set

    home: string = closest node of user_name

    print all friends, where they live closest to and how far away

    print out friends that would take more than 20 minutes to walk (average human
        walking speed is 5.1 km/h)

    candidate = candidate_solution(home, home, visit_set, selected_time)
    hamiltonian_path = simulated_annealing(candidate['path'], selected_time)
    // or hill_climbing(candidate['path'], selected_time)

    hamiltonian_path['path'] = normalise_path(hamiltonian_path['path'],
        selected_time)

    print how much the trip would cost and how long it would take

    print the path of the hamiltonian_path
end function
```

```
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
```

```
// creates a canditate solution using the NN Heuristic
function canditate_solution (
    start: node,
    end: node,
    visit: set of nodes,
    current_time: datetime,
):
    path = [start]
    current_vertex = start
    cost = 0

    while len(visit) != 0:
        closest_node = fetch_djk(current_vertex, visit, current_time)
        path.add(closest_node)
        cost += closest_node.cost
        visit.remove(closest_node)
        current_vertex = closest_node

    // go back to the end node
    closest_node = fetch_djk(current_vertex, end, current_time)
    path.add(closest_node)
    cost += closest_node.cost

    return {'path': path, 'cost': cost}
end function
```

```
function pairwise_swap (
    u: integer,
```

```
    v: integer,
    path: path of nodes
):
    new_tour = []

    for i in [0, u]:
        new_tour.add(path[i])
    for i in [v, u):
        new_tour.add(path[i])
    for i in (v, len(path)]:
        new_tour.add(path[i])

    return new_tour
end function
```

```
function calculate_cost (
    path: path of nodes,
    current_time: datetime
):
    cost = 0
    time = current_time

    for i from 0 to len(path) - 1:
        djk = fetch_djk(path[i], path[i + 1], current_time)
        cost += djk['cost']
        time += djk['cost'] number of minutes

    return cost
end function
```

```
function hill_climbing (
    candidate: path of nodes,
    current_time: datetime,
    fail_count: int = 0
):
    if fail_count < 200:
        cost = calculate_cost(candidate, current_time)
        u = random number from 1 to len(candidate) - 1 inclusive
        v = random number from u to len(candidate) - 1 inclusive

        new_tour = pairwise_swap(u, v, candidate)
        new_cost = calculate_cost(new_tour, current_time)

        if new_cost <= cost:
            // new cost is better/equal -> accept
            return hill_climbing(new_tour, current_time, 0)
        else:
            // new cost is worse -> go again
```

```
            return hill_climbing(candidate, current_time, fail_count + 1)
    else:
        return candidate
end function
```

```
function simulated_annealing (
    candidate: path of nodes,
    current_time: datetime,
):
    // parameters to fiddle with
    temp = 0.98
    min_temp = 0.00001
    temp_change = 5
    beta = 1.2
    alpha = 0.85

    old_cost = calculate_cost(candidate, current_time)

    while temp > min_temp:
        for n from 1 to temp_change:
            u = random number from 1 to len(candidate) - 1 inclusive
            v = random number from u to len(candidate) - 1 inclusive

            new_tour = pairwise_swap(u, v, candidate)
            new_cost = calculate_cost(new_tour, current_time)

            ap = acceptance_probability(old_cost, new_cost, beta, temp)

            if ap > random float from 0 to 1:
                candidate = new_tour
                old_cost = new_cost

        temp *= alpha

    return candidate
end function
```

```
function acceptance_probability (
    old_cost: number,
    new_cost: number,
    beta: number,
    temp: number
):
    c = new_cost - old_cost

    if c <= 0:
        return 1
    else:
```

```
        return e**((-beta * c)/temp)
end function
```

```
function dijkstras (
    start: node,
    current_time: datetime
):
    unexplored = empty min priority queue of nodes based on distance

    // Set all node distance to infinity
    for node in graph:
        distance[node] = infinity
        predecessor[node] = null
        unexplored.add(node)

    // starting distance has to be 0
    distance[start] = 0

    // while more to still explore
    while unexplored is not empty:
        min_node = unexplored.minimum_node()
        unexplored.remove(min_node)

        // go through every neighbour and relax
        for each neighbour of min_node:
            current_dist = distance[min_node] + dist(min_node, neighbour,
                current_time + to_minutes(distance[min_node]))
            // 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 {
        'distances': distance,
        'predecessors': predecessor,
    }
end function
```

```
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, current_time)

    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
```

```
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:
        wait_time = wait time from data (precomputed)
        distances.add(edge.weight + wait_time)

    return min(distances)
end function
```

```
function normalise_path (
    path: path of nodes,
    current_time: datetime
):
    return_path = []
    time = current_time

    for i from 0 to len(path) - 1:
        djk = fetch_djk(path[i], path[i + 1], current_time)
        time += djk['cost'] number of minutes
        // this is to prevent the last and first item double up
```

```
12          return_path += everything in djk['path'] except last item
13
14      return_route.add(last item in route)
15
16      return cost
17  end function
```