Algorithmics SAT - Friendship Network Part 3

Garv Shah

2023-08-25

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 algorithmand 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

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.

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 = 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
```

Dijkstra's

```
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
```

Fetch Dijkstra's (Cached)

```
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
```

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
```

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 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.

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. Below is the new Pythonic implementation:

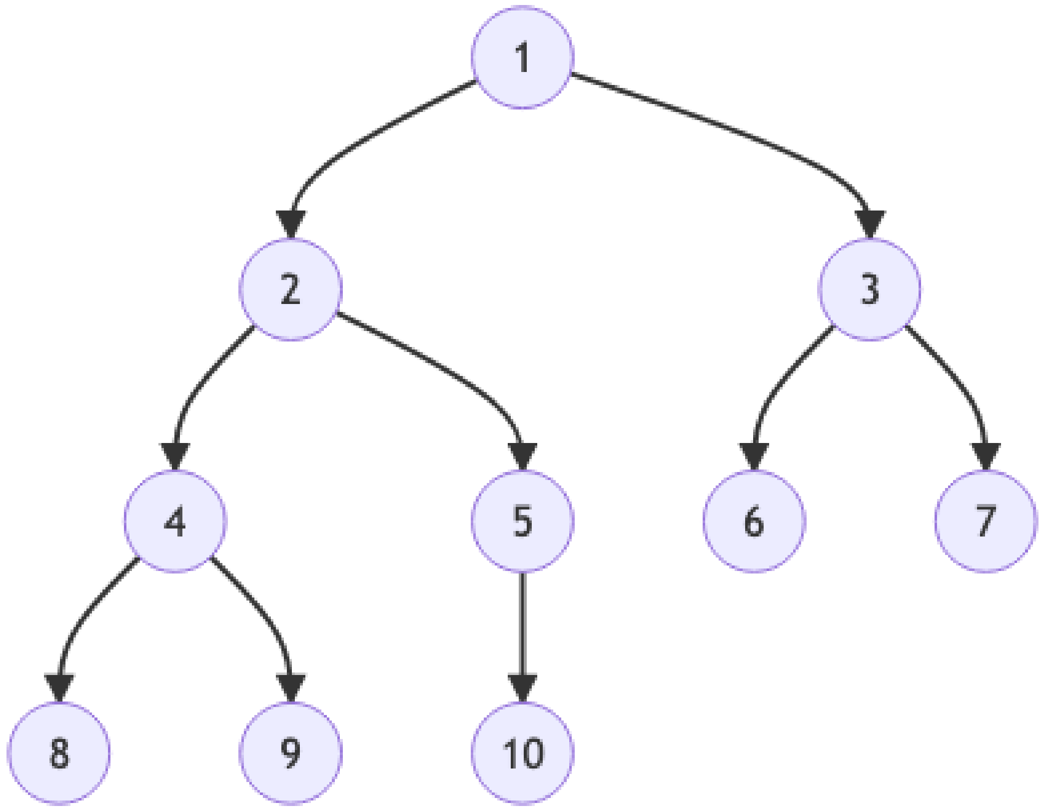
```python
def dijkstra(start, current_time):
    """
    Dijkstra's Shortest Path Algorithm.
    :param start: start node    :type start: str
    :param current_time: the current time when Dijkstra's is being called    :type current_time: dt.datetime
    :return: The distance dictionary and the predecessor dictionary.
    :rtype: dict
    """
    # 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 = []

```

```
    size = len(g.nodes())

    for node in list(g.nodes()):
        if node == start:
            unexplored.append((0, node))
        else:
            unexplored.append((float('inf'), node))

    hq.heapify(unexplored)

    distance[start] = 0

    while size > 0:
        min_node = hq.heappop(unexplored)[1]
        size = size - 1

        for neighbour in g.neighbors(min_node):
            current_dist = distance[min_node] + dist(min_node, neighbour,
                                                    current_time +
                                                      dt.timedelta(minutes=distance[min_node]))
            # a shorter path has been found to the neighbour -> relax value            if
                current_dist < distance[neighbour]:
                distance[neighbour] = current_dist
                hq.heappush(unexplored, (current_dist, neighbour))
                predecessor[neighbour] = min_node

    return {'distances': distance, 'predecessors': predecessor}
```

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.



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.

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

```python
def fetch_hk(start, end, visit):
    name = f"{start}-{end}{visit}@{current_time}"

    global cached_hk
    if name not in cached_hk:
        cached_hk[name] = held_karp(start, end, visit, current_time)

    return cached_hk[name]
```

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

Visit Set Size	Initial Algorithm (s)	Improved Dijkstra's (s)	Improved Dist (s)	Improved Held-Karp (s)
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.

Modified Exact Algorithm Pseudocode

Below is the final pseudocode for the exact algorithm, based on Held-Karp.

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 = 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
```

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 = 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
```

Fetch Held-Karp (Cached)

```
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
```

Dijkstra's

```
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
```

Fetch Dijkstra's (Cached)

```
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
```

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

    return min(distances)
end function
```

Practicalities of an Exact Algorithm

Tractability

Approximate/Heuristic Algorithms

Christofides' Algorithm*

Pairwise Exchange

Simulated Annealing

Ant Colony Optimisation*

Final Solution

Include brief explanation of how it works as conclusion.

Comparison of Solutions

Design Features

Coherence

Fitness for Problem

Efficiency

Time Complexities

Constraints

Similarities and Differences

Tractability & Implications