THE UNIVERSITY OF
SYDNEY

Assignment 4

COMP2123: Data Structures and Algorithms
510415022

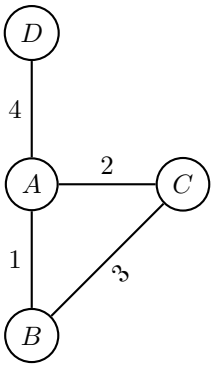
Problem 1. (20 points)

ModifiedPrim is an algorithm which claims to either:

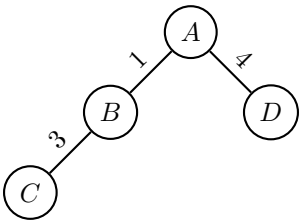
1. produce a binary spanning tree T from a given weighted graph G and a specified node r, or
2. correctly return infeasible when it is not possible to produce a binary spanning tree from the given node

We show here that the algorithm is incorrect because it returns infeasible in a situation where it is very much possible to construct a binary spanning tree.

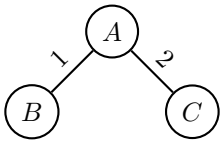
Consider the following graph G



and let the starting node be A. It is possible to construct a binary spanning tree from this graph as such:



However, the modifiedPrim's algorithm will (in a greedy fashion) choose to append nodes B and C as the children of A first and then return infeasible as D cannot be successfully added to the tree without violating the binary constraint. See the figure below for the state of the tree upon returning infeasible.



Hence the algorithm is incorrect.

Problem 2. (40 points)

a) It is helpful to first formalise the question into some concrete requirements instead of the colourful problem statement given in the assignment.

Let's suppose that the bus timetables are given in a series of m `line` objects. Each object, denoted here by L, is very simple and contains the bus stops on that line, the bus timetable, and the flat fee for that bus line.

Components of a `line` object

- a list of ℓ bus stops: $S = [s_1, s_2, \ldots, s_\ell]$
- a $k \times \ell$ matrix T containing the bus timetable such that T_{ij} represents the time the j-th instance bus arrives at the i-th stop

$$T = \begin{bmatrix} s_1 I_1 & s_1 I_2 & \ldots & s_1 I_k \\ s_2 I_1 & s_2 I_2 & \ldots & s_2 I_k \\ \vdots & \vdots & \ddots & \vdots \\ s_\ell I_1 & s_\ell I_2 & \ldots & s_\ell I_k \end{bmatrix} \tag{1}$$

- a fee variable F which stores the flat fee

Alongside the global departure and arrival time t and t' and the origin and destination stops a and b, assume we are given some collection of these objects as input, such as

$$\texttt{bus\_lines} = [L_1, L_2, \ldots, L_m] \tag{2}$$

Some further assumptions (as per the notification or from Ed):

- There are no "express" buses i.e. all instances of a certain bus line travel to all ℓ stops.
- You need to pay for the full fare of every bus you ride e.g. if you get off a L_1 bus, travel on an L_2 bus and get back on an L_1 bus, you would need to pay $2 \times F_1 + 1 \times F_2$.
- All bus lines go in one direction i.e. there are no "loops."

The high-level idea of our algorithm is to first create a graph of the bus network from the input bus lines list, and then to run Dijkstra's shortest path algorithm on the graph starting at a. We can then find the cheapest route between nodes a and b. Let's examine each of these steps in further detail.

I. **Create a directed graph G of the entire bus network**
We need to first convert the series of `line` objects into a directed graph before we can run Dijkstra's algorithm on it. To do this, we will iterate over each `line` L and then add each bus stop $s \in S_L$ to the graph if it is not already present. One directed edge

will be added between nodes s_i and s_{i+1} for each bus line which travels that route, and the edge will contain as attributes the line L and all the departure times from s_i to s_{i+1} as an array (i.e. the i-th row of T_L). The edge weight will be the flat fee F_L for that bus line. Some pseudocode for this has been provided below:

```
1:  function CREATE-BUS-NETWORK(bus_lines, m, ℓ)
2:      G ← empty directed graph
3:      for i in range(m) do
4:          L_i = bus_lines[i]
5:          for j in range(ℓ) do
6:              s_j = L_i.bus_stops[j]
7:              if s_j ∉ G then
8:                  G.add_vertex(s_j)
9:              if j ≠ 0 then
10:                 s_{j-1} = L_i.bus_stops[j − 1]
11:                 e ← new directed edge (s_{j-1}, s_j)
12:                 e ← edge weight F_{L_i}
13:                 e ← edge attributes L_i, T[j − 1]
14:                 G.add_directed_edge(e, s_{j-1}, s_j)
```

II. **Run a modified Dijkstra's algorithm on G**

We further split this step up for ease of explanation.

i. Modify Dijkstra's algorithm to work with directed graphs:
As we are working with a directed graph, of course we need to update Dijkstra's algorithm to work with directed edges. This has already been studied in Tutorial 8 Problem 1. Please refer to the tutorial sheet for more details.

ii. Modify Dijkstra's algorithm to take into account the flat fee of bus lines:
Dijkstra's algorithm is a shortest path algorithm. That is, it considers edge weights to be "distances" and outputs the minimum distance from an input node u to every other node v in the graph. If does this by checking for each u and v whether the current estimated distance to v, $d[v]$, can be lowered by considering a new path (u, v). Our algorithm will utilise the same principle, albeit checking for cost c instead of distance.

$$c[u] + w(u, v) < c[v] \ \forall \ v \in V \tag{3}$$

However we must also accommodate for the flat fees of bus lines. To do this, alongside the true cost $c[v]$ of reaching the node v from u, the algorithm will also store and maintain $L[v]$ which is the last bus line required to reach v from u. Then, updating the definition of $w(u, v)$:

$$w(u, v) = \begin{cases} 0, & \text{if } L[v] = L[u] \\ w(u, v), & \text{otherwise} \end{cases} \tag{4}$$

That is, if the bus line taken from u to v is the same as the bus line taken to travel

to u, the weight will count as 0 at each iteration of Dijkstra's algorithm. In essence, we are ensuring that Dijkstra's algorithm prioritises remaining on the same bus line throughout as much of the journey as possible in order to obtain the cheapest route possible.

iii. Modify Dijkstra's algorithm to take into account the global arrival time t':
We do not just wish to find the cheapest path between input nodes a and b. We are also required to ensure that we arrive at stop b before the global arrival time t'. Hence, at each iteration of Dijkstra's algorithm, we also check if the arrival time at node v is greater than t'. If it is, then obviously the path the algorithm is currently taking is infeasible. We remove the edge (u, v) that the algorithm chose to travel to v and run Dijkstra's algorithm again from u.

iv. Consider the earliest departure time from each node:
When departing a node u on the path (u, v), the algorithm will look through the array attributed to the edge and find the earliest departure time which is greater than the local arrival time to u.

In summary, when the algorithm reaches node u at time t, it will

i. decide which path is the cheapest (II.ii.)

ii. check the array on the cheapest path for the earliest departure time $> t$ (II.iv.)

iii. check if the arrival time to node v is greater than the global arrival time t' (II.iii.)

If it is, the edge (u, v) will be removed and the algorithm will be run again from u.

b) The correctness of the algorithm can be argued through considering whether the algorithm backtracks or not:

(a) **The algorithm does not backtrack at all**
This case implies that the algorithm began at a, checked for the earliest and cheapest path possible from each node between a and b, checked the time constraint, and arrived at b in one go, without ever overshooting the global arrival time. This case will clearly produce the most correct and cheapest travel plan from a to b simply due to it following the conditions we described above.

(b) **The algorithm backtracks ≥ 1 times**
Consider the moment the algorithm decides to backtrack due to the time constraint not being fulfilled. Suppose it backtracks from a node v to a node u; then, the greatness of the algorithm lies in the fact that the path until u which the algorithm chose is *still the cheapest and earliest valid path from a to u*. Once we remove the edge (u, v) we do not need to restart the algorithm from the start node a, rather, we can simply restart from u.

Once you restart the algorithm from u, in an almost induction-like scenario the algorithm's correctness is once again based on these two cases: either the algorithm never backtracks between u and b and therefore the entire path from a to b is correct, or it backtracks once more.

Unless a and b were never connected to begin with, or if there is no legitimate solution within the time bounds t and t', there must eventually be some moment that the algorithm does not backtrack from a node u to b. The path (a, u, b) is then guaranteed to be the cheapest route between a and b due to the discussion above.

c) The time complexity of this algorithm is not polynomial on $mk\ell$ however. In it's worst case, the algorithm needs to iterate over *all possible routes* between a and b before finding the ideal path. For example, consider the contrived situation of 4 bus lines ($m = 4$) each sharing the same 6 bus stops ($\ell = 6$). Suppose L_1 has fee F_1, L_2 has cost F_2, etc, such that $F_4 > F_3 > F_2 > F_1$. If the only possible solution for the route was to take L_4 from a to b directly, our algorithm would check *all possible routes* before checking this solution last. That is, it would check $4^6 - 1 = m^\ell - 1$ different routes before landing on the solution. This is definitely not polynomial time.

Problem 3. (40 points)

a) Assume the collection of n shapes is provided as input to the algorithm as an array, e.g.

$$shapes = [s_1, s_2, s_3, \ldots, s_n] \tag{5}$$

The high level overview of the algorithm is to first convert the collection of shapes into a directed weighted graph, and then run Dijkstra's algorithm on the graph starting from shape A to find the minimum cost traversal from A to B (if it exists).

I. **Converting the collection into a graph**
For each shape in the collection, iterate over all other shapes and run the overlap() function for each. If the function returns true, and the two shapes are not both squares, add the two shapes s_i and s_j to the graph as nodes. Alternatively, if one of the shapes is already in the graph, only add the other shape and the 2 new edges; we only want one unique node per shape. Run the area() function for s_j and attach the result as a weight for the directed edge (s_i, s_j). Similarly, run the area() function for s_i and attach the result as a weight for the directed edge (s_j, s_i). That is, each pair of nodes has two directed edges connecting them, with each edge weighted as the area of the shape at the head of the edge. We are left with a graph of nodes which can all be successfully traversed (i.e. the graph will never contain two connected squares in a row).

II. **Check if the shapes A and B both exist in the graph**
Traverse over the array of nodes in the graph V and check for shapes A and B. If even one of them is missing (meaning it was infeasible to get to that shape in the first place), return -1 as the minimum cost.

III. **Run a modified Dijkstra's algorithm on the graph**
Starting at the input shape A, run the directed edge implementation of Dijkstra's algorithm on the graph. The algorithm will take into account the areas of the shapes as edge weights. If $D[B]$ is still ∞ at the end of the algorithm, this means that the graph is disconnected. Return -1 in this case. Otherwise, the lowest cost path between the two shapes will simply be given by $D[B]$.

b) There is very little doubt that the second part of the algorithm is correct: we know from the lectures and tutorials that given a directed graph with edge weights, Dijkstra's algorithm will be able to compute the shortest path from an input node r to every other node in r's connected component. Instead, the main element of correctness to prove is that our graph is constructed properly.

Firstly, we do not add paths which cannot be traversed given the constraint of the problem. Because we add two shapes at a time, we can check if both elements in the pair are squares and disregard these paths. In this way, Dijkstra's never has to perform any checks, and *every* Dijkstra's shortest path solution is a valid one.

Secondly, we need to show that the edges and edge weights are constructed correctly. When Dijkstra's algorithm performs edge relaxation, it is checking whether the current estimated distance to the node v can be improved by considering the new edge (u, v). In other words, the edge (u, v) is responsible for adding "weight" to the node v. Using this, we have ensured to place the area of each shape v on the edges pointing inward to v, as area in our problem is analogous to weight. Hence, the cost of each traversal from shape A to B is guaranteed to consider the areas of all the shapes $v_1, v_2, \ldots, v_n$ in between A and B.

In this way, the algorithm we have constructed is correct, and should return either -1 for infeasible traversals or the cheapest cost traversal between two shapes A and B.

c) The time complexity of our algorithm is dominated by the conversion of the input array of shapes into a graph. This step requires a nested loop to check whether any other shapes overlap with each shape. As the `overlap()` function itself is constant time, this entire procedure takes $O(n^2)$ time. The rest of the algorithm is quite simple:

- getting the area for each shape will take $O(n)$ for n shapes.
- the construction of the graph using an adjacency-list representation will take $O(n + m)$ time as we have to iterate over each edge and vertex at least once
- checking the existence of A and B in the vertices array will take $O(n)$ time
- running Dijkstra's with a binary heap will take $O((n + m) \log n)$ time

Regardless, the running time is still dominated by the construction of the graph and is $O(n^2)$.