**Solution 1.**

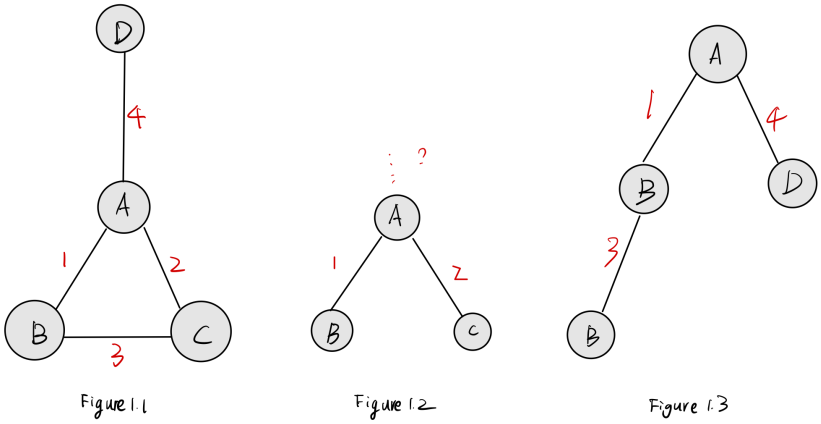


Figure 1: Counterexample for MODIFIEDPRIM

A counter example is provided as above.

Figure 1.1 is the graph to be traversed.

By the provided algorithm, all the vertices in the tree $T$ will be full in the order of edge weight: edge$(A, B)$, edge$(A, C)$ (as indicated in line 4 for the set $S$). We then end up with Figure 1.2. The vertex $D$ is thus not considered and the algorithm returns **infeasible**.

However, we can see clearly that a MST is indeed feasible, as in Figure 1.3. Thus the algorithm is not correct in that it returns **infeasible** when in fact an MST is feasible.

**Solution 2.**

**(a)**

**High-level understanding**

We do three iterations. **The first iteration** takes $O(ml)$ which traverses through every single bus stop to create a graph $G$. In this graph $G$, every bus line $B_i$ is a vertex and edges represent the connection between two bus lines. **The second iteration** takes $O(m^2)$ time. It produces every possible option respresented by an array $[B_1, B_2, \cdots, B_n]$ meaning boarding $B_1$ then changing into $B_2$ and so on. The travel option array is added as a vertex attribute to the corresponding vertex. It also calculates the bus fare incurred by it successively and add it as a vertex attribute. **The final iteration**, which takes $O(m^3)$ time, considers all the travel options produced by the second iteration which ends at the destination stop $b$ and sort them by the bus fare. Then it iterates from the cheapest travelling option to the more expensive ones. For each option, it checks if it satisfies the time constraints. If it does, the algorithm stops immediately and export that option in the correct format.

**Detailed description**

***get-graph()*** To convert it into a graph, we traverse from $a$ to find a set $S_0$ of all the bus lines $a$ is on. Then for each bus line $B_i$ that we just found than go through $a$, we traverse through all the possible bus stops in that bus line. If that bus stop is also on some other bus lines other than the ones in $S_0$, then we now have an edge between the current bus line that the new bus line. We then go on and traverse through all the bus stops in the new bus line and repeat the process.

***get-all-options()*** We first add all the bus lines $B_i$ that go through $a$ to a set $A$, and the ones go through $b$ to a set $B$. The set of vertices $A$ serves as a set containing all vertices that have been visited. We have an array stored as an attribute of the vertex initialized with all vertices in $A$, e.g., $[B_1], [B_2], \ldots, [B_n]$. Then for each vertex (e.g. $B_k$) (bus line) in $G$ but not in $A$ that is connected to any vertex in $A$, we attach that to the corresponding array ($[B_2, B_k]$ where we can transfer from $B_2$ to $B_k$). We also add them to the set $A$ as visited vertices.

Eventually, we do this for every vertex that is yet to be visited $v \in G, v \notin A$. Suppose $v$ is adjacent to $B_i \in \{B_1, B_2, \ldots, B_n\}$, we first add it to the visited vertices set $A$. Then, we look at all the arrays stored in $B_i$, e.g., $[B_1, B_4, B_2, B_i]$. All these arrays represent a path from $a$ to the bus line represented by the last vertex If it is not the same as any of the elements in the corresponding array (i.e. adding this bus line causes a loop), we add it to the array and store it in the new vertex $v$. We also calculate the bus fare for this renewed array option by adding the bus fare of $v$ to the corresponding attribute of the previous vertex $B_i$. After adding, we add the bus fare for $B_k$ on top of the price attribute of the previous vertex, e.g., $B_2$, and add the updated price to $B_k$.

***get-cheapest-option-in-time()*** We first iterate through all the bus options array and cheerypick the one ending with $b$ destination stop. We first sort all the options by the bus fare. Then we iterate through all the options from the ″cheapest to the most expensive ones. For each option, we check if it satisfies the time constraints. If it does, the algorithm stops immediately and export that option in the correct format.

How to check for time constraints? For each bus line, we have another time-pair array aside from the travel options array. We always add the the stop-time

pair of the bus stop to the time-pair array where we change bus line into the *travel plan array*. We go from the first bus line $B_1$. We check if there is any buses with a departure time $t$ later than $t$. Then we follow along to the intersecting bus stop to the second bus line $B_2$ and changes buses. We check if the time of getting off is earlier than the earliest available departure time of $B_2$ from that stop. We repeat the same process for every bus line $B_i$ afterwards and always get on the next available bus to change into. Ultimately, we reach $b$ and check if the arrival time is still earlier than $t'$. If so, then this travelling option is still valid and we stop and output the time-pair array.

**(b)** First, we prove the correctness of *get-graph()*.

**Claim 2.1.** *All instances where we can change from bus $B_1$ to $B_2$ is matched to exactly one edge between the corresponding vertices (bus lines).*

*Proof.* The *get-graph()* function builds a graph where each vertex represents a bus line, and an edge exists between two vertices if there is a bus stop that is common between the corresponding bus lines.

We consider two bus lines, $B_1$ and $B_2$. We need to show that an edge exists between $B_1$ and $B_2$ in the graph if and only if there is a bus stop common to both $B_1$ and $B_2$.

Let's consider a bus stop $s$ which is common to both $B_1$ and $B_2$. When the *get-graph()* function traverses through all the bus stops in $B_1$, it will reach $s$ and find that $s$ is also on $B_2$. Thus, it will add an edge between $B_1$ and $B_2$ in the graph. This proves the "if" part of our claim.

For the "only if" part, let's assume that an edge exists between $B_1$ and $B_2$ in the graph. This edge would have been added by the *get-graph()* function only when it found a bus stop that is common to both $B_1$ and $B_2$. Thus, there must exist such a bus stop.

Therefore, we can conclude that all instances where we can change from bus $B_1$ to $B_2$ are matched to exactly one edge between the corresponding vertices (bus lines) in the graph. This validates the correctness of the *get-graph()* function. □

**Claim 2.2.** *The get-all-options() function generates all possible sequences (without a loop) of bus lines from a to b and correctly calculates the total fare for each sequence.*

*Proof.* We first prove that it generates all possible sequences (without a loop) of bus lines from $a$ to $b$. We prove by induction.

**Base case** We have only one bus line $B_1$ in the set $A$. Then we have $B_1$ in the array $[B_1]$.

**Inductive hypothesis** For all the visited vertices in the set $A$, the algorithm has generated all possible sequences (without a loop) of bus lines from $a$ up to that visited vertex.

**Inductive step** For the current vertex $A_i$ in $A$ that have been visited, we look at all the adjacent vertices $v$ of $A_i$. For each adjacent $v$ that is not in the travel plan array which can incur a loop, we add it to that array. That means we have **exhausted** all our options for the next step for that $A_i$. Thus, we have generated all possible sequences (without a loop) of bus lines from $a$ up to that visited vertex.

We then also prove by induction that the algorithm correctly calculates the total fare for each sequence.

**Base case** For the case where we only have one bus line $B_1$ in the set $A$, the total fare is the fare of $B_1$. Thus the correctness is proven for this case.

**Inductive hypothesis** For each travel plan sequence, our algorithm correctly calculates the total fare.

**Inductive step** We calculate the bus fare for this renewed array option by adding the bus fare of $v$ to the corresponding attribute of the previous vertex $B_i$. Since that is the only difference between the previous bus fare and the current bus fare, our algorithm is correct. □

**Claim 2.3.** *The get-cheapest-option-in-time() function returns the cheapest travel plan that satisfies the time constraints.*

*Proof.* The correctness lies in whether or not we can correctly decide if it satisfy the time constraints.

We prove by induction.

**Base case** For a single bus line $B_1$ in the travel option array, we because our algorithm make sure the the departure time of $B_1$ from $a$ is later than $t$, it is correct.

**Inductive hypothesis** Suppose that for $i$-th element in the array, the time constraints are satisfied.

**Inductive step** We only need to prove that for the new element, it departs later than the existing bus time and ends before the $t'$ closing time. Because our algorithm checks that, the overall correctness is proven. □

**(c)** *get-graph()* takes $O(ml)$ time. Since it goes through all possible bus stops and check if the intersecting bus lines have been added previously.

*get-all-options()* takes $O(m^2)$ time. Let us suppose that we have $a_0$ elements in the original set $A$ and $a_n$ elements in the original set $B$. Then we have $a_1$ vertices adjacent to any of the vertices in $A$ and we add them to $A$. And $a_2$ vertices for the updated $A$ and so on. Then we have $a_0a_1a_2 \dots a_n \leq (a_0 + a_1 + a_2 + \cdots + a_n)^2 = m^2$.

This inequality is a special case of the Cauchy-Schwarz Inequality, which is a very fundamental and powerful inequality in mathematics. Here's a proof using Cauchy-Schwarz. Firstly, let's understand what the Cauchy-Schwarz Inequality states: For any sequences of real numbers $x_1, x_2, \dots, x_n$ and $y_1, y_2, \dots, y_n$, the inequality is as follows:

$$(x_1y_1 + x_2y_2 + \cdots + x_ny_n)^2 \leq \left(x_1^2 + x_2^2 + \cdots + x_n^2\right)\left(y_1^2 + y_2^2 + \cdots + y_n^2\right)$$

Now, let's apply this to prove the inequality you mentioned. Let $x_i = a_i$ and $y_i = 1$ for $i = 1, 2, \dots, n$. Then, the left-hand side of the Cauchy-Schwarz inequality becomes

$$(a_1 * 1 + a_2 * 1 + \cdots + a_n * 1)^2 = (a_1 + a_2 + \cdots + a_n)^2$$

and the right-hand side becomes

$$\left(a_1^2 + a_2^2 + \cdots + a_n^2\right)\left(1^2 + 1^2 + \cdots + 1^2\right) = \left(a_1^2 + a_2^2 + \cdots + a_n^2\right) * n$$

Since the product $a_1a_2a_3 \dots a_n$ is one of the terms in the sum $a_1^2 + a_2^2 + \cdots + a_n^2$ it follows that $\left(a_1^2 + a_2^2 + \cdots + a_n^2\right) * n \geq a_1a_2a_3 \dots a_n$. Therefore, we have shown that

$$(a_1 + a_2 + \cdots + a_n)^2 \geq a_1a_2a_3 \dots a_n$$

So, the inequality holds as per the Cauchy-Schwarz inequality. Please note, however, that this is a very specific case of the inequality where all $a_i$ are non-negative. The Cauchy-Schwarz inequality holds more generally for any sequences of real numbers.

*get-cheapest-option-in-time()* takes $O(m^3)$ time. Because we have proven the number of all travel options is $O(m^2)$. For the worse case scenario where only the most expensive travel plan is within the given time constraints, we have to iterate through all $O(m^2)$ plan in which every plan can only involve $O(m)$ vertices since no loop is allowed. Then it will be $O(m^2(m)) = O(m^3)$ time.

**In conclusion,** the total time complexity is $O(m^3 + ml)$ time.

**Solution 3.**

**(a)**

**High-level understanding**

We will first abstract a graph from the given $n$ shapes. Every shape is a vertex in the graph. For any pair of shapes, we will examine if they (a) have an overlap and (b) have different number of sides. Only when these two conditions are met do we add this add an edge.

Then we adapt the Dijkstra's algorithm to find the shortest path from $A$ to $B$. The difference is that we put weight on the vertices but not on the edges. But the spirit remains.

**Detailed description**

***get-graph()*** To do this, first we abstract a shape into a vertex with the following quality: (1) number of sides, (2) area, (3) a list of vertices that it is connected to.

Then we iterate through all the shapes and find the edges that satisfy the condition that (a) they have an overlap and (b) they have different number of sides.

The upshot is that we have a graph that have all the possible vertices and possible edges.

***Dijkstra-traverse()*** To traverse it, we slightly modify the Dijkstra's algorithm.

From vertex $A$ in the remaining priority queue, we look at all the adjacent vertex $K$ we first assign the value of $\text{Area}(A)$ to the $\text{edge}(A, K)$. After updating the edge, we continue with the Dijkstra's algorithm as usual.

After calling *get-graph()* first and *Dijkstra-traverse()* later, we have obtained the shortest path from $A$ to all other vertices. And $B$ is one of them, and thus the correctness is proven.

**(b)** The correctness is proven as the Dijkstra's algorithm is correct as in the lectures.

**Claim 3.1.** *The two filtering conditions in get-graph() is justified.*

*Proof.* It is as our algorithm uses it to ensure that all the edges in the graph are valid. □

**Claim 3.2.** *The modified Dijkstra's algorithm is correct.*

*Proof.* Because we only have weights associated with vertices, the burden of proof lies in whether or not the the weight assigning process is justified. In this case, we need to show that the weight for each vertex after updating shows the minimum cost to get to that vertex.

We prove by induction.

**Base case** We are still at vertex $A$ and the weight is the area of $A$ itself.

**Inductive hypothesis** We have visited a set $S$ of vertices and the weight attribute of the vertices in them have been updated correctly to reflect the costs to travel from vertex $A$ to that visited vertex.

**Inductive step** To visit a new vertex $K$, for an edge between $S_i$ and $K$ where $S_i \in S$, we set the weight of $\text{edge}(S_i, K)$ to $\text{Area}(K)$.

After that, when we are performing the *decrease-key()* operation, we notice that when we add the weight of $\text{edge}(S_i, K)$ to the key of $S_i$, we have successfully obtain the value to get from $A$ to $K$ via $S_i$ since the value of $\text{Area}(K)$ is the only difference between the current weight of paths and the previous one. □

**In conclusion,** the two claims are proven and the rest is the same as the standard Dijkstra's algorithm as proven in the lectures.

**(c)** Building a graph from the shapes take $O(n^2)$ time and using the modified Dijkstra's algorithm takes $O(n^2)$ time. In total, $O(n^2)$ time.

For building the graph *get-graph()*, we have $n$ shapes in total. For each shape $K$, we look at all other $n-1$ shapes and conduct two $O(1)$ operations to examine if (a) they have an overlap and (b) they have different number of sides. That is $O(n)$ time for each shape, which yields $O(n^2)$ time for all $n$ shapes.

For the modified Dijkstra's algorithm *Dijkstra-traverse()*, we have $n$ vertices and at most $n(n-1)/2$ edges. We do one more assiging value operation which takes $O(1)$ time. As proved in the lectures, using the Fibonacci heap, the algorithm takes $O(m + n \log n) < O(m + n^2)$ time. In our case, number of edges $m = O(n^2)$ because $m \leq n(n-1)/2$, so the algorithm takes $O(n^2)$ time.

Combining the two parts, we have $O(n^2)$ time in total.