COMP3711: Design and Analysis of Algorithms

Tutorial 8

HKUST

Question 2

Let $T = (V, E)$ be a tree and $e = (u, v) \in E$.

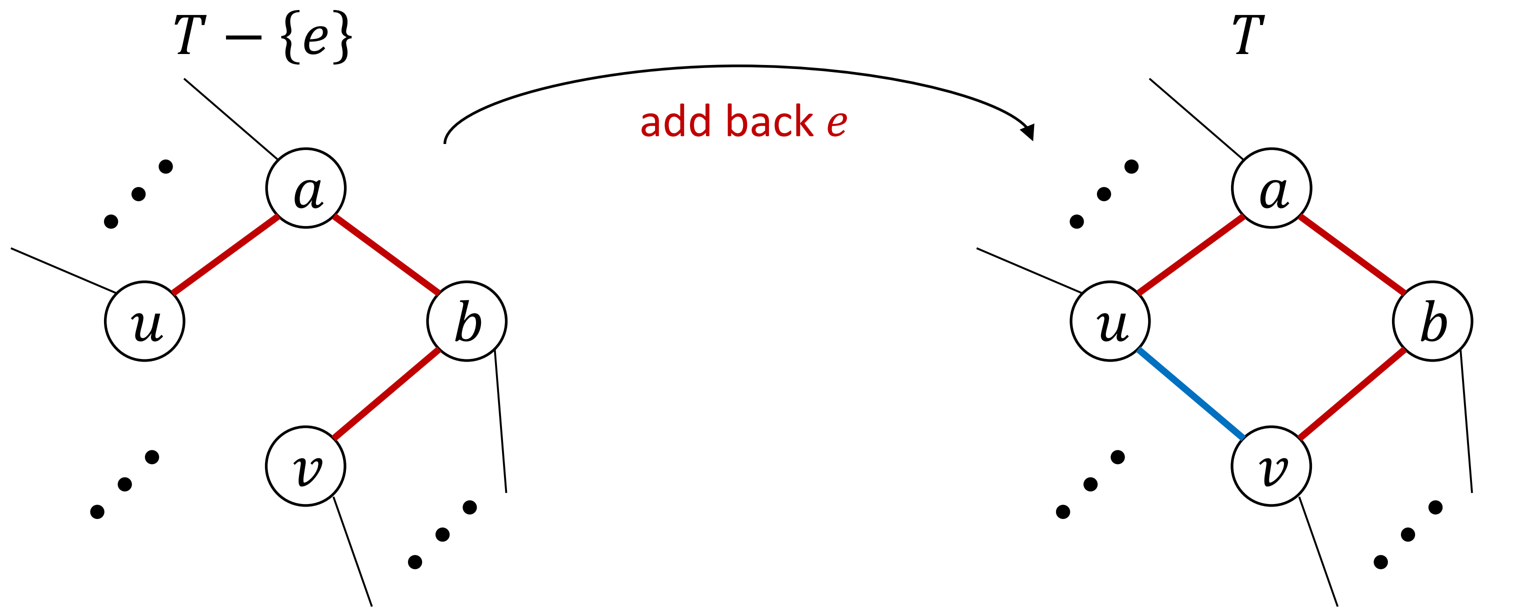
Show that removing e from T leaves a graph with exactly two connected components, with one component containing u and the other containing v.

Note: This fact will be (implicitly) essential in the later design of optimal algorithms for constructing Minimal Spanning Trees

We prove the statement by showing that,
after removing $e = (u, v)$,
the graph that remains will have only two connected components,
C_u, C_v which, respectively, contain u, v

Solution 2

First note that u and v cannot be in the same connected component in $T - \{e\}$. This is because if u, v were connected by some path P in $T - \{e\}$ then P and e together would create a cycle in T, contradicting the fact that T is a tree.



A connected component containing u, v.
The path $P = u \rightarrow a \rightarrow b \rightarrow v$

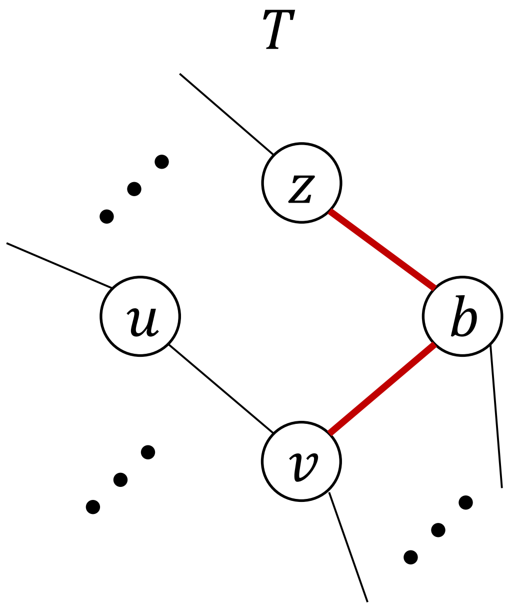
$P \cup e = u \rightarrow a \rightarrow b \rightarrow v \rightarrow u$ forms a cycle in T.
Contradiction!

Solution 2

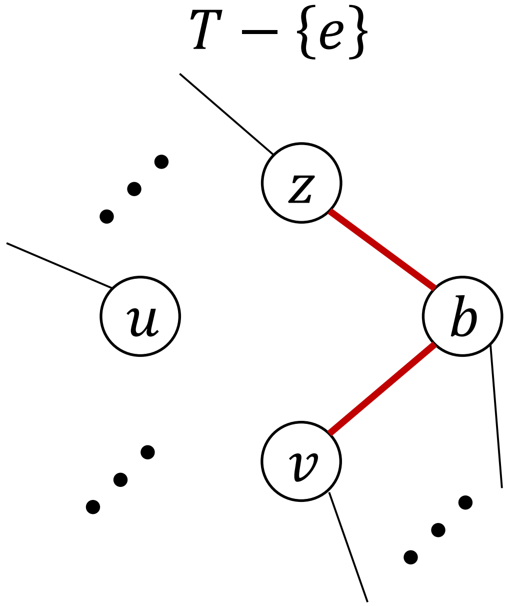
Now let C_u, C_v be the connected components containing u, v. Let $z \neq u, v$ be *any* other vertex in E. We will show that either $z \in C_u$ or $z \in C_v$. This would prove the original statement.

Since T is connected it must contain a simple path P from z to v.

If $e \notin P$ then every edge in P is in $T - \{e\}$ so $z \in C_v$.



$P = z \to b \to v$



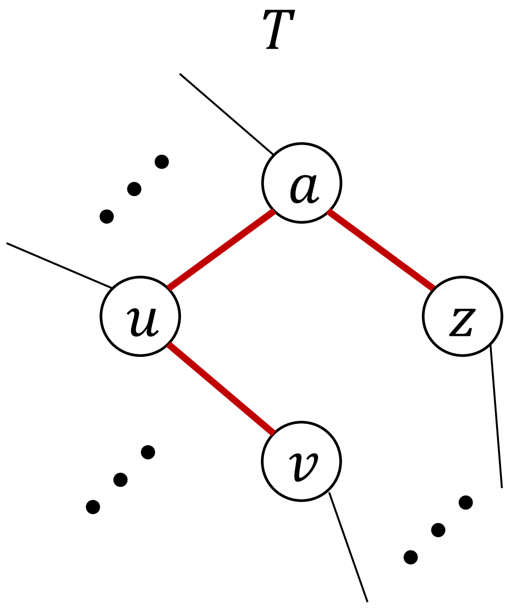
Every edge in P is in $T - \{e\}$

Solution 2

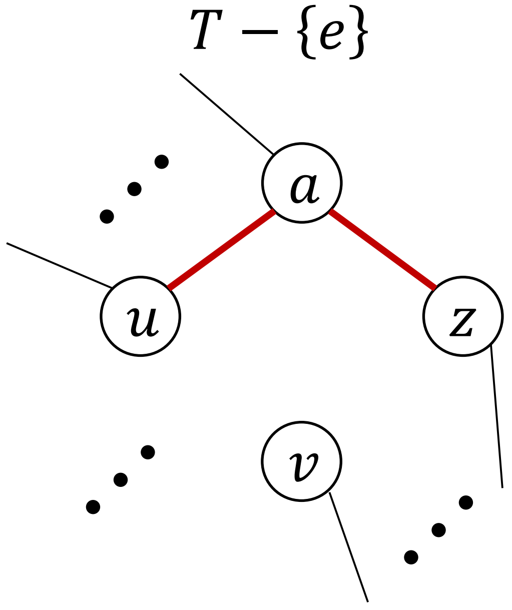
Now let C_u, C_v be the connected components containing u, v. Let $z \neq u, v$ be *any* other vertex in E. We will show that either $z \in C_u$ or $z \in C_v$. This would prove the original statement.

Since T is connected it must contain a simple path P from z to v.

If $e \in P$ (because P is simple), then e must be the *last* edge on P, so $P - \{e\}$ is a path in $T - \{e\}$ from z to u, and $z \in C_u$.



$P = z \to a \to u \to v$



$P - \{e\}$ is a path from z to u

Solution 2

Let C_u, C_v be the connected components containing u, v.

Let $z \neq u, v$ be *any* other vertex in E.

Since T is connected, it must contain a simple path P from z to v.

We just showed that

- If $e \notin P$ then $z \in C_v$.
- If $e \in P$ then $z \in C_u$.

This proves the original statement.

Let $T = (V, E)$ be a tree and $e = (u, v) \in E$.

Show that removing e from T leaves a graph with exactly two connected components, with one component containing u and the other containing v.

Question 3

Let $G = (V, E)$ be a weighted graph with non-negative distinct edge weights. In class we showed that T is the *unique* MST of G.

Now replace every weight $w(u, v)$ with its square $(w(u, v))^2$.

(a) Is T still a MST of G with the new weights?
Either prove that it is or give a counterexample.

(b) Next consider a shortest path $u \rightarrow v$ in the original graph.
Is this path still a shortest path with the new weights?
Either prove that it is or give a counterexample.

Solution 3a

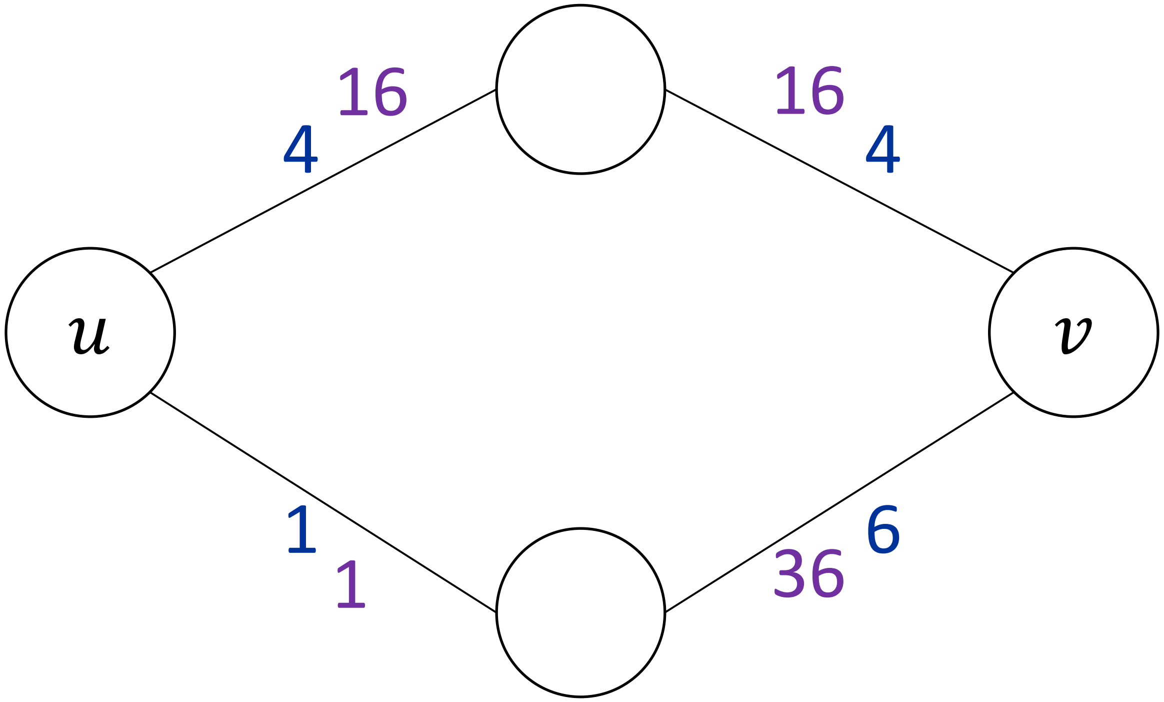
Yes, the MST remains unchanged when the weights are changing from w to w^2. There are many proofs of this statements. Here's one

Run Kruskal's algorithm on the edges.
The first step of Kruskal's algorithm is to sort the edges by weight.
Squaring the weights of the edges does not change their order so Kruskal's algorithms on the new weights will start with the same ordering as with the old weights.

After the sorting step, Kruskal's algorithm never looks at the weights again; it just looks at which edges connect to which vertices.
=> the second part of the algorithm will perform exactly the same operations for both the original and squared edge weights.
=> the algorithm will output the same set of edges, e.g., the same MST, for both the original and squared edge weights.

Solution 3b

No, after squaring the weights the old shortest path might not remain a shortest path in the new graph.
In the graph below the shortest path from u to v for original blue weights goes through the bottom but after squaring the weights it goes through the top.



Question 4

Let G be a connected undirected graph with distinct weights on the edges, and let e be an edge of G.

Suppose e is the largest-weight edge in some cycle of G.

Show that e can not be in the MST of G.

Recall that if the edge weights are distinct then we proved in class that the MST is unique. So the statement

*Show that e can not be in **the** MST of G.*

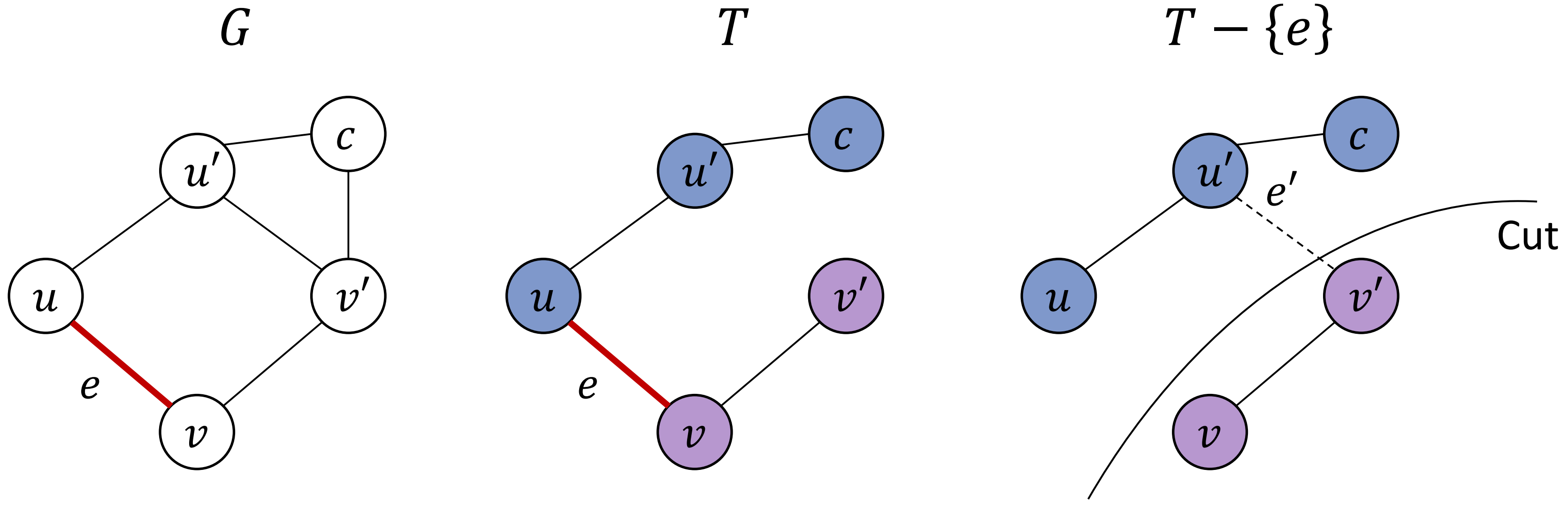
is well defined.

Solution 4

Let T be the MST of G and suppose T contains $e = (u, v)$.
Removing e from T breaks it into two connected components:
S & $V - S$, with $u \in S$ & $v \in V - S$.

Consider the cycle C that has e as its largest-weight edge.
C starts from u, goes to v, and takes another path to go back to u.

This path must cross this cut somewhere via an edge $e' = (u', v')$ with $u' \in S$ and $v' \in V - S$.
e was the largest weight edge on the cycle => $w(e') < w(e)$.



Solution 4

Let T be the MST of G and suppose T contains $e = (u, v)$.
Removing e from T breaks it into two connected components:
S & $V - S$, with $u \in S$ & $v \in V - S$.

Consider the cycle C that has e as its largest-weight edge.
C starts from u, goes to v, and takes another path to go back to u.

This path must cross this cut somewhere via an edge $e' = (u', v')$
with $u' \in S$ and $v' \in V - S$.
e was the largest weight edge on the cycle => $w(e') < w(e)$.

Now add e' to $T - \{e\}$. This connects the two components
=> $T' = (T - \{e\}) \cup \{e'\}$ is also a spanning tree T'.

T' is T with e removed and e' added so

$$cost(T') = cost(T) - w(e) + w(e') < cost(T)$$

This contradicts the fact that T was a MST.
$\Rightarrow$ e cannot be the largest edge on some cycle.

Question 5

It is not difficult to see that if *e* is a minimum weight edge in *G* then *e* is **always** an edge in *some* Minimum Spanning Tree for *G.*

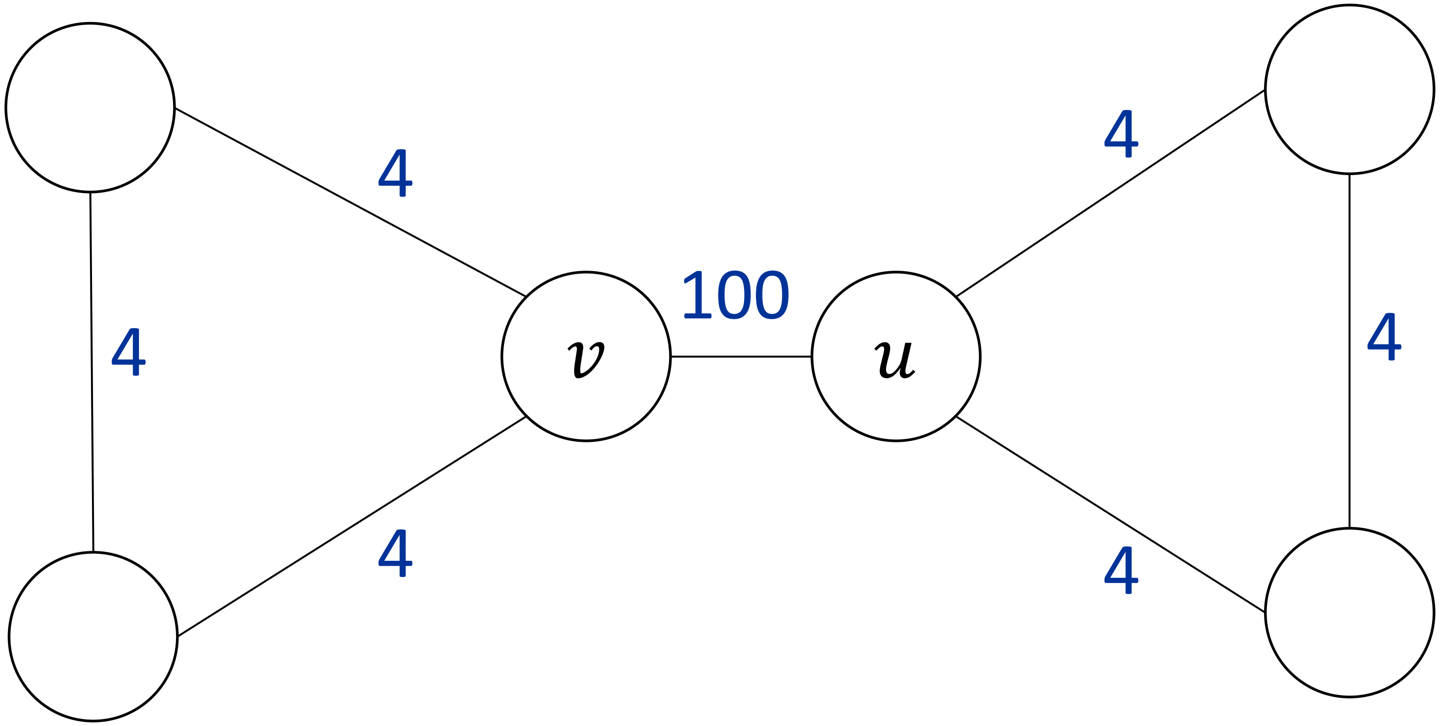
Prove that if *e* is a maximum weight edge, the corresponding statement (which would be that *e* **never** belongs to a MST*)* is not correct. In fact

(a) It is possible that *e* *does* belong to a MST of *G.*

(b) But, it is also possible that *e* does not belong to any MST for *G.*

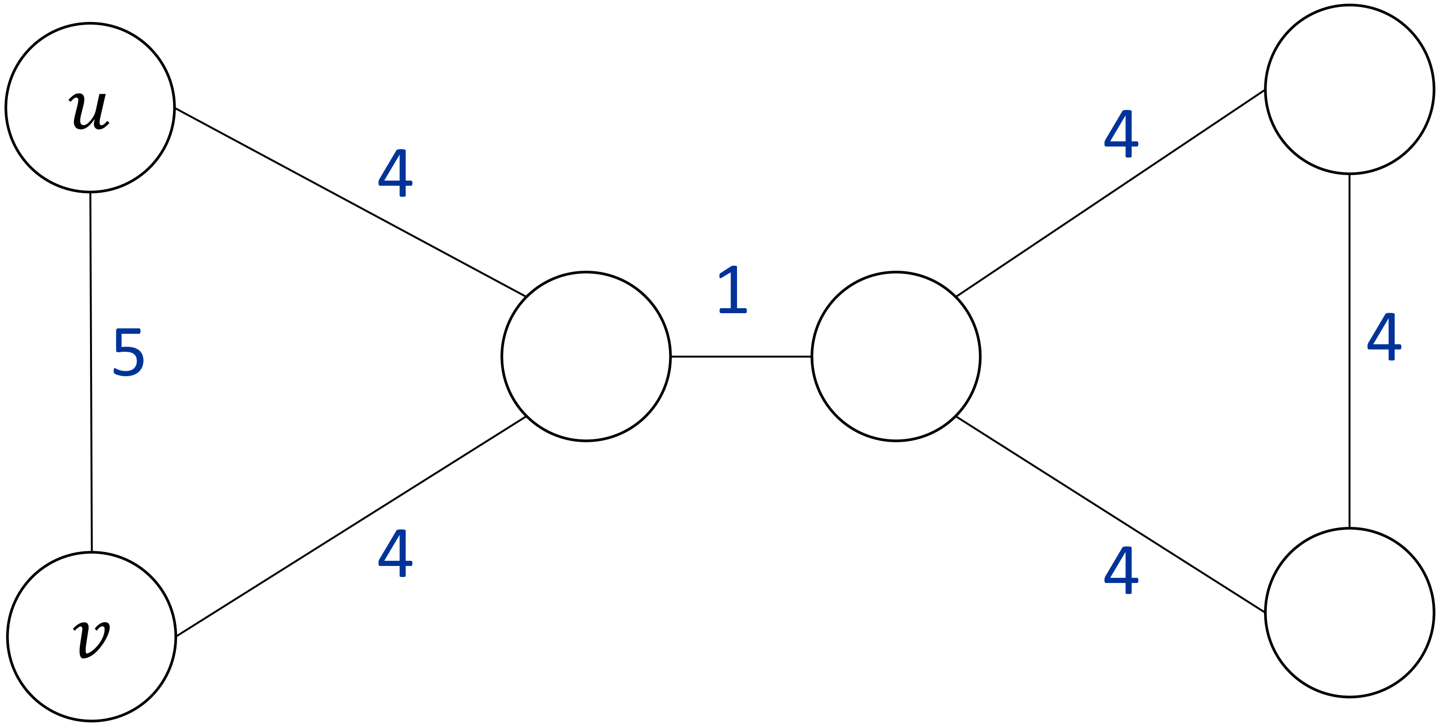
Solution 5

a) If the removal of (u, v) makes G disconnected, then (u, v) is in the MST regardless of its weight.



Solution 5

b) There are many cases when the heaviest edge is NOT part of any MST.



Question 6

Let G be a connected undirected graph in which all edges have weight either 1 or 2.

Give an $O(|V| + |E|)$ time algorithm to compute a minimum spanning tree of G.

Justify the running time of your algorithm.

Note: You may either present a new algorithm or just show how to modify an algorithm taught in class.

Solution 6

G is a connected undirected graph in which all edges have weight either 1 or 2. We want an $O(|V| + |E|)$ algorithm to compute the MST of G.

One approach is to run Prim's algorithm, except taking advantage of the fact that all edges are 1 or 2 to implement the priority queue more efficiently.

Recall that Prim's algorithm was $O(|E| \log |V|)$ because it made

- $O(|V|)$ calls to Heap Insert => $O(|V| \log |V|)$
- $O(|V|)$ calls to Extract-Min => $O(|V| \log |V|)$
- $O(|E|)$ calls to Decrease-Key => $O(|E| \log |V|)$

We will now see, how, IN THIS SPECIAL CASE, we can implement every heap operation in $O(1)$ time, which will lead to an $O(|V| + |E|) = O(|E|)$ alg.

Important: We are NOT changing the MST algorithm.
We are only changing the data structure it calls.

Solution 6

G is a connected undirected graph in which all edges have weight either 1 or 2. We want an $O(|V| + |E|)$ algorithm to compute the MST of G.

We are going to create a priority queue (min-heap) data structure that uses only $O(|1|)$ time per operation when all values are 0, 1 or ∞.

(∞ is needed because the heap keys in Prim's algorithm can take the value of an edge or the value ∞).

The priority Queue will just be a set of 3 doubly linked lists:

- List 0 contains all items with key value ∞
- List 1 contains all items with key value 1
- List 2 contains all items with key value 2

Heap Insert:

All nodes get inserted into List 0 because all nodes have key value ∞ at start. Inserting a node into a list uses $O(1)$ time.

=> Heap Insert uses $O(1)$ time

Solution 6

G is a connected undirected graph in which all edges have weight either 1 or 2. We want an $O(|V| + |E|)$ algorithm to compute the MST of G.

We are going to create a priority queue (min-heap) data structure that uses only $O(1)$ time per operation when all values are 0, 1 or ∞.
(∞ is needed because the heap keys in Prim's algorithm can take the value of an edge or the value ∞).

The priority Queue will just be a set of 3 doubly linked lists:

- List 0 contains all items with key value ∞
- List 1 contains all items with key value 1
- List 2 contains all items with key value 2

Extract Min:

If List 1 is not empty, just pull off the first value in List 1. $O(1)$
 Otherwise, if List 2 is not empty, just pull the first value off in List 2. $O(1)$
  Otherwise, pull the first item off List 0 (this actually won't happen in Prim's) $O(1)$

=> Extract Min takes $O(1)$ time

Solution 6

G is a connected undirected graph in which all edges have weight either 1 or 2. We want an $O(|V| + |E|)$ algorithm to compute the MST of G.

We are going to create a priority queue (min-heap) data structure that uses only $O(1)$ time per operation when all values are 0, 1 or ∞.
(∞ is needed because the heap keys in Prim's algorithm can take the value of an edge or the value ∞).

The priority Queue will just be a set of 3 doubly linked lists:

- List 0 contains all items with key value ∞
- List 1 contains all items with key value 1
- List 2 contains all items with key value 2

Decrease Key: Will only move an item from list 0 to lists 1,2, or list 2 to list 1
(i) remove item from its current list in $O(1)$ time (because list is doubly linked).
(ii) Insert item into the front of the appropriate new list in $O(1)$ time.
This can be done in $O(1)$ time because there are only two such possible lists.

=> Decrease Key takes $O(1)$ time

Solution 6

G is a connected undirected graph in which all edges have weight either 1 or 2. We want an $O(|V| + |E|)$ algorithm to compute the MST of G.

Our approach is to run Prim's algorithm, except taking advantage of the fact that all edges are 1 or 2 to implement the priority queue more efficiently. Have just seen that all operations can be implemented in $O(1)$ time.

Recall that Prim’s algorithm uses
– $O(|V|)$ calls to Heap Insert => $O(|V|)$
– $O(|V|)$ calls to Extract-Min => $O(|V|)$
– $O(|E|)$ calls to Decrease-Key => $O(|E|)$

Thus, for this special case in which all edges have cost 1 or 2, Prim’s algorithm will run in $O(|V| + |E|) = O(|E|)$ time.

Solution 6 – Wrap up

This idea of not changing the **algorithm** but changing the **data structure** the algorithm calls to take advantage of special input, is a common one in algorithmics.

It is a very standard way of speeding up running times in real life