EECS 660: FUNDAMENTALS OF COMPUTER ALGORITHMS

MODULE III: GREEDY ALGORITHMS, PART I

DISCLAIMER

ACKNOWLEDGEMENT

- Many of the figures, unless otherwise stated, come from ***Algorithm Design*** 4th edition, by Jon Kleinberg and Eva Tardos.

GREEDY ALGORITHMS

- Recall that an algorithm tries to solve a computational problem, i.e., to achieve some mathematical objective.

- One common way (and perhaps the most intuitive way) to design algorithm is, at each stage, try to get closer to the mathematical objective as much as possible.
  - an "myopic" approach
  - in some cases it does solve the problem correctly
  - in some cases it does NOT
    - while it does not, it could give a useful approximation of the optimal solution

GREEDY ALGORITHM

- its design is usually intuitive and simple
- it is often efficient, and its time complexity is easy to analyze
- however, it needs to be proved that the greedy algorithm, which may stuck in local optimal, can or cannot achieve the global optimal

GREEDY ALGORITHM

- Consider the **Knapsack Problem** (imagine an explorer enters a treasure cave with a small, limited capacity knapsack, and he tries to take the most valuable treasures with him):

- You are given a total capacity C, a set of items I, each $i \in I$ is assigned with a weight w_i and a value v_i. The problem seeks to pack a subset of items $I' \subseteq I$, such that:

$$\sum_{i \in I'} w_i \leq C$$

and that the total value

$$\sum_{i \in I'} v_i$$

is maximized.

GREEDY ALGORITHM

- Considering the following instance of the problem.

$$C = 12$$
$$w_1 = 8, v_1 = 16$$
$$w_2 = 6, v_2 = 13$$
$$w_3 = 6, v_3 = 10$$
$$w_4 = 5, v_4 = 7$$

GREEDY ALGORITHM

- In this problem, our math objective is to maximize the sum of value $\sum_{i \in I'} v_i$, subject to the constraint $\sum_{i \in I'} w_i \leq C$.
- According to the greedy design principal, we could try keep adding the most valuable item to the knapsack until it is full.
- With this strategy, we will starting picking item 1. And since no more knapsack space is available for the rest for the items, we end up with a total value of 16.

GREEDY ALGORITHM

- We could try to be a bit smarter, by considering the value per unit weight of the items.

$$w_1 = 8, v_1 = 16, \frac{v_1}{w_1} = 2$$
$$w_2 = 6, v_2 = 13, \frac{v_2}{w_2} = 2.2$$
$$w_3 = 6, v_3 = 10, \frac{v_3}{w_3} = 1.7$$
$$w_4 = 5, v_4 = 7, \frac{v_4}{w_4} = 1.4$$

GREEDY ALGORITHM

- We will still start with picking item 2 as it has the highest value per unit weight.
- The next item with the highest value per unit weight is item 1; but we cannot pick it since that will violate the knapsack capacity constraint.
- We then turn to item 3.

- The knapsack is then full, we will terminate the algorithm and end up with a total value of 23. Which is indeed the optimal solution to this instance.

GREEDY ALGORITHM

- **Although designed under the same design principal, some algorithms are better than the other algorithms (at least for some instances).**

- It is important to consider the characteristics of the problem, and design a proper algorithm for solving it.

GREEDY ALGORITHM

- Unfortunately, the value per unit weight algorithm is still unable to solve the problem.

- Consider the following instance:

$$w_1 = 8, v_1 = 16, \frac{v_1}{w_1} = 2$$
$$w_2 = 6, v_2 = 10, \frac{v_2}{w_2} = 1.7$$
$$w_3 = 6, v_3 = 10, \frac{v_3}{w_3} = 1.7$$
$$w_4 = 5, v_4 = 7, \frac{v_4}{w_4} = 1.4$$

GREEDY ALGORITHM

- However, we see that if we pick item 2 and 3, we could end up with a total value of 18, which is larger than what we have achieved using greedy algorithm design.
- It could stuck in local optimum (picking item 1) and cannot jump out (with no mechanism of removing selected items) to achieve global optimum.
- As a result, it is very important to note that: greedy algorithms cannot solve all problems.
  - but it does solve some problems
  - and they are of more interests

THE SHORTEST PATH PROBLEM

- Given a graph $G = \{V, E\}$, and the cost/length for each edge (denote the length of edge e as l_e and $l_e \geq 0$ for all $e \in E$), find the shortest path from vertex u or v whose total length is minimized among all paths from u to v. (We will discuss the problem allowing for negative-weighted edges later the class, if time permits.)
- The single-source version: given a source vertex s, find the shortest paths from s to all other vertices in V. Solving the single-source version directly solves the source-target version.
- Denote the length of a path P as $d(P) = \sum_{e \in P} l_e$; for the single-source path, we also denote the path from s to v as P_v and its length $d(P_v)$ whenever s is clear in the context. Define the shortest distance between s and v as $d(v)$ whenever s is clear in the context. Note that $d(P_v) \geq d(v)$ and the equality only holds when P_v is the shortest path.

- You might have learnt it in EECS 560: Data Structures.

THE SHORTEST PATH PROBLEM

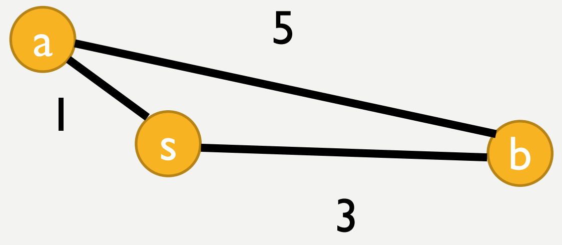
- Let's consider the single-source version.

- Recall that our math objective is to find a path P_v such that $d(P_v)$ is minimized among all possible paths between s and v.

- It is intuitive to model the path finding process as walking from the source s to v.
  - at each step, we walk from one node to another

THE SHORTEST PATH PROBLEM

- It is intuitive to model the path finding process as walking from the source s to v.
  - at each step, we walk from one node to another
  - as a result, we could always pick the shortest edge that is available at the current stage
  - however, we could get stuck in a long path, if the long path begins with a short edge



THE SHORTEST PATH PROBLEM

- So, it is necessary to consider reaching v from different paths. While we don't know which path is the correct one, we should try all possibilities.

- Note that for each step, we jump from one node to another. Obviously we do not want to jump back to some nodes that have already been visited, since it is a waste of effort and does not help in the shortest path setting. So, we assume we shall visited a new node at each step.

- Under the greedy design schema, we shall visit a new node with as little effort (total path length from the source s) as possible. Hopefully for any node v, it is also reached with the minimum effort.

THE SHORTEST PATH PROBLEM

- To find the node that we can reach with the minimum effort, we will keep track of all nodes we have visited (say S), and the corresponding effort that we have spent in reaching them.

- Hence, the new vertex we can reach with the minimum effort is:

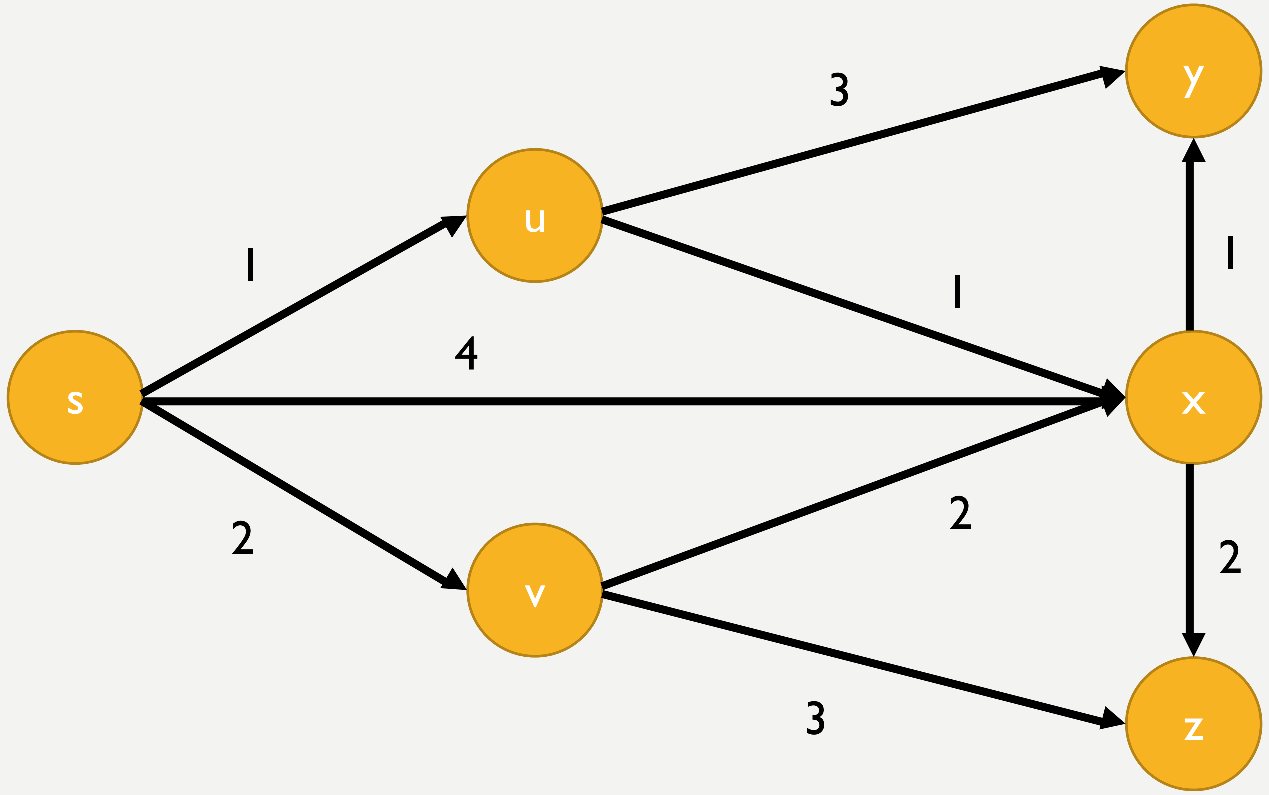
$$\underset{v, e=(u,v): u \in S}{\operatorname{argmin}} \left(d(u) + l_e\right)$$

- We can keep walking through the graph to continuously seek the new node we can reach with the minimum effort, and terminate until we finish exploring the entire graph. This is called the **Dijkstra's algorithm**.

THE SHORTEST PATH PROBLEM

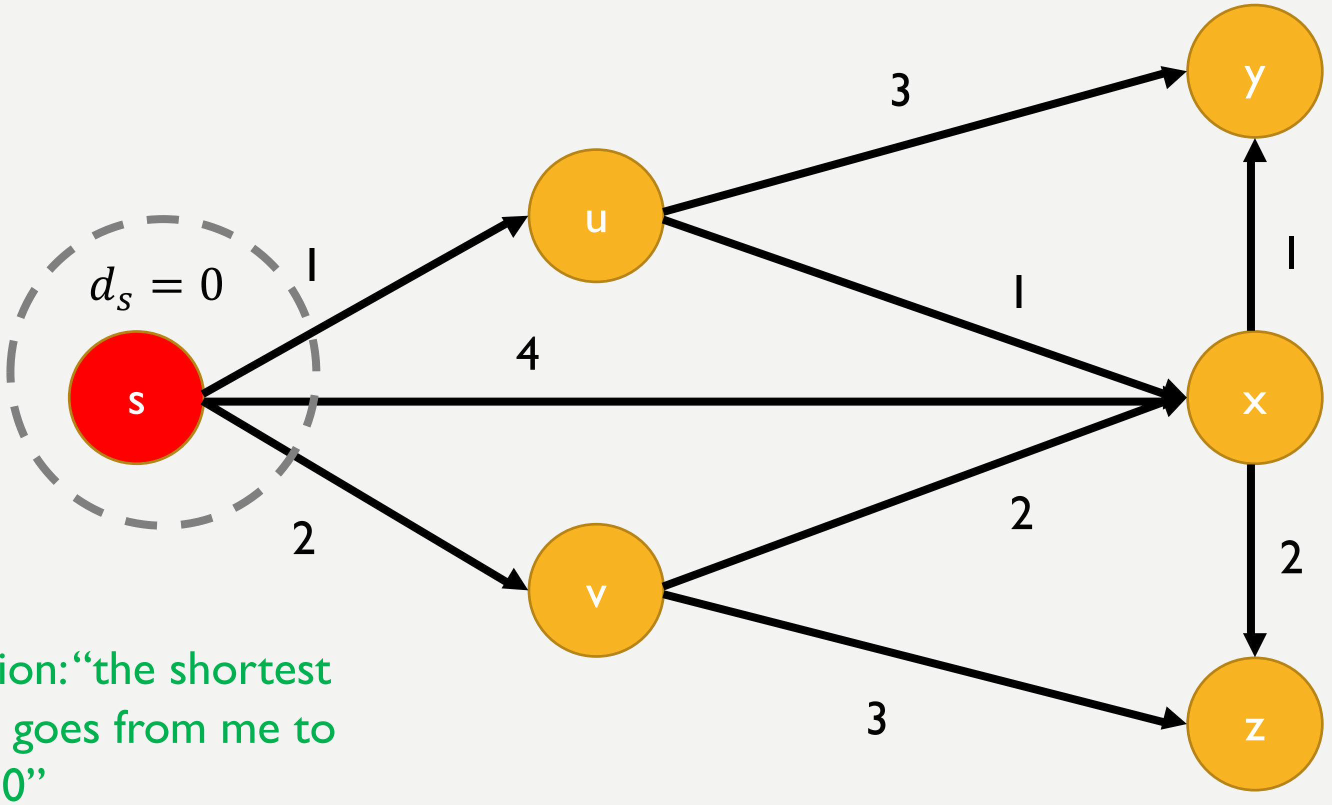
```
Dijkstra's algorithm ($G$)
Initialize $S = \{s\}$ and $d(s) = 0$;
While $S \neq V$
        Select a node $v \notin S$ with at least one edge from $S$ for which
                $d(v) = \min_{e=(u,v):u \in S} (d(u) + l_e)$;
        Add $v$ to $S$, record $d(v)$;
Endwhile
```

THE SHORTEST PATH PROBLEM



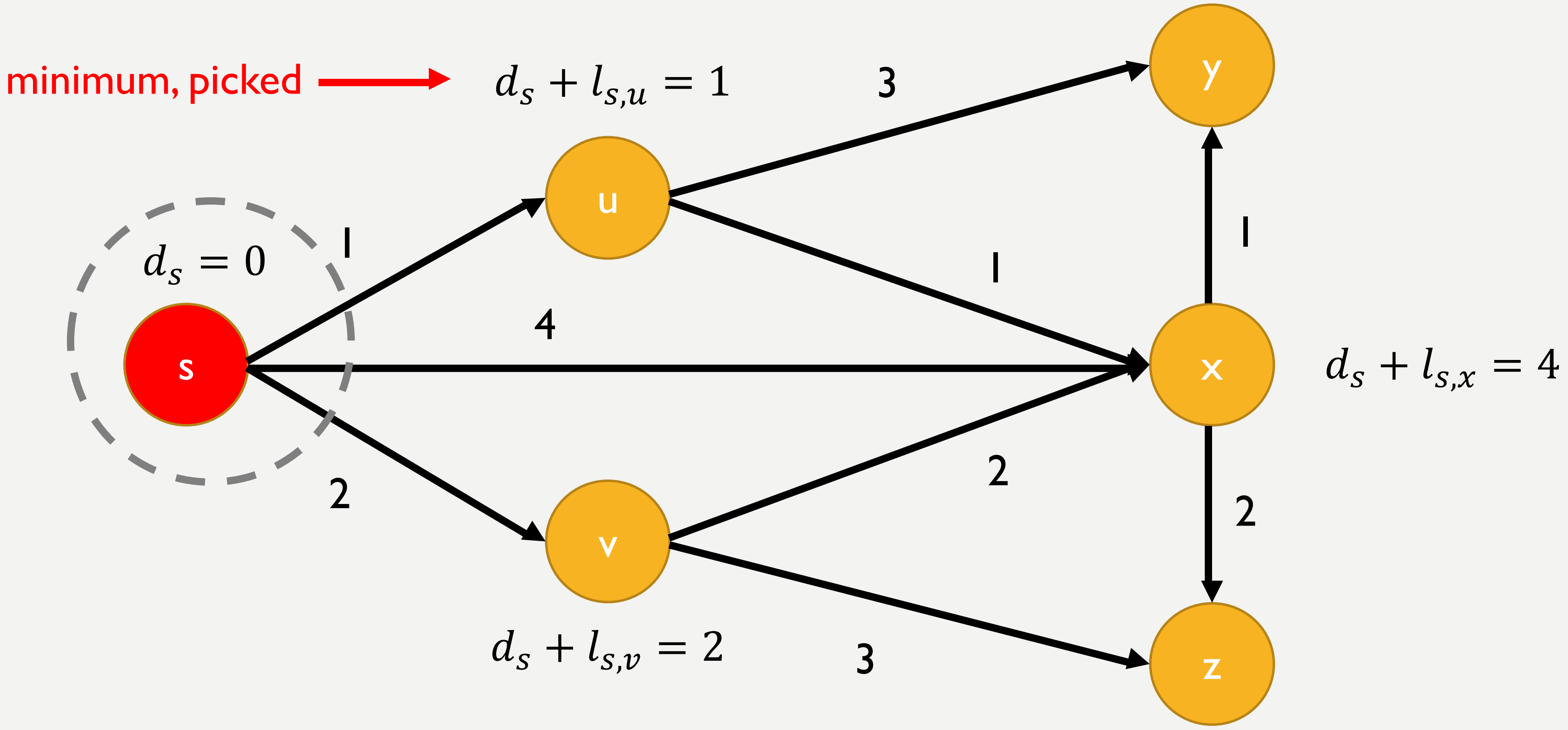
Consider the following example with no negative edge (we will see in a minute why we cannot handle graphs with negative edges)

THE SHORTEST PATH PROBLEM

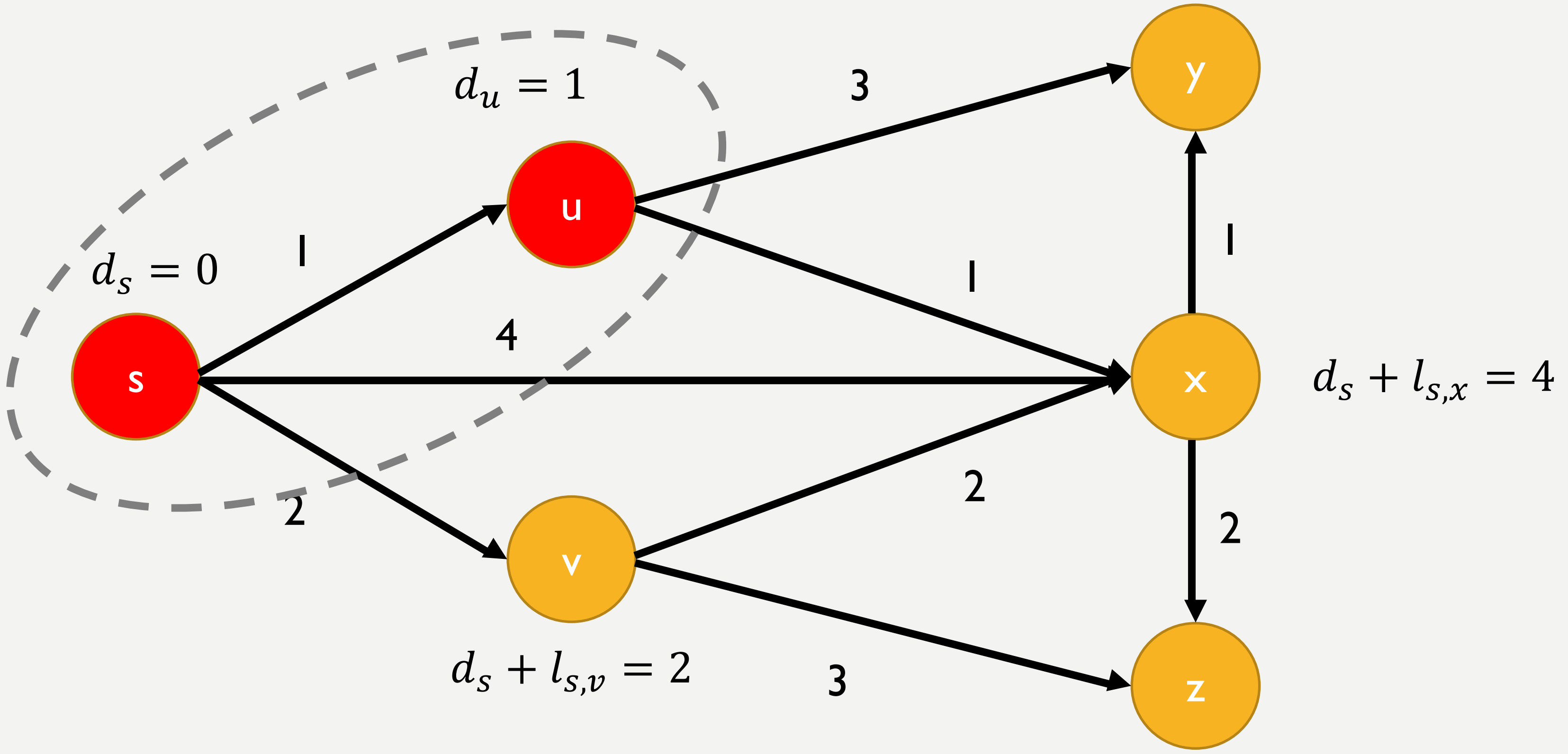


Initialization: "the shortest path that goes from me to myself is 0"

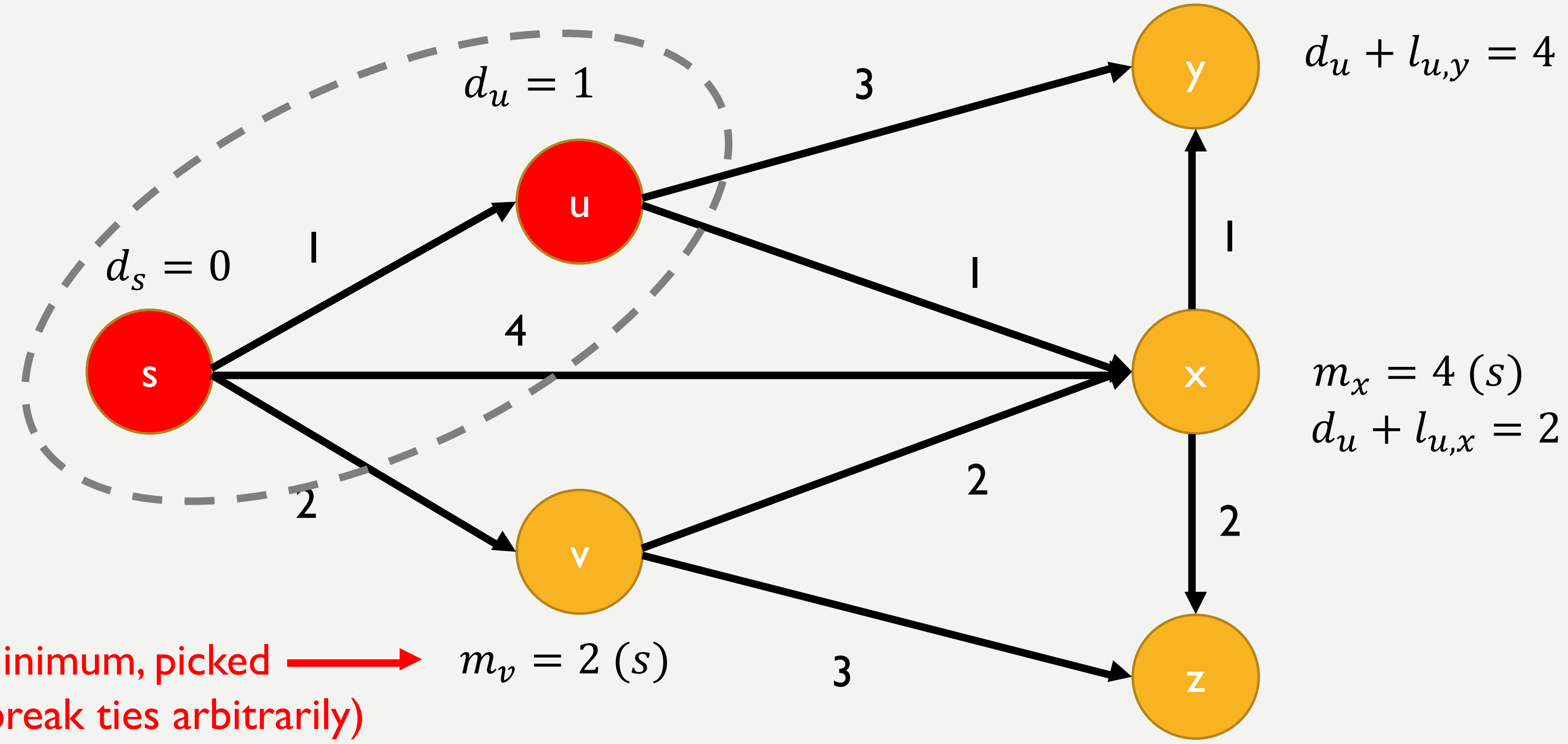
THE SHORTEST PATH PROBLEM



THE SHORTEST PATH PROBLEM

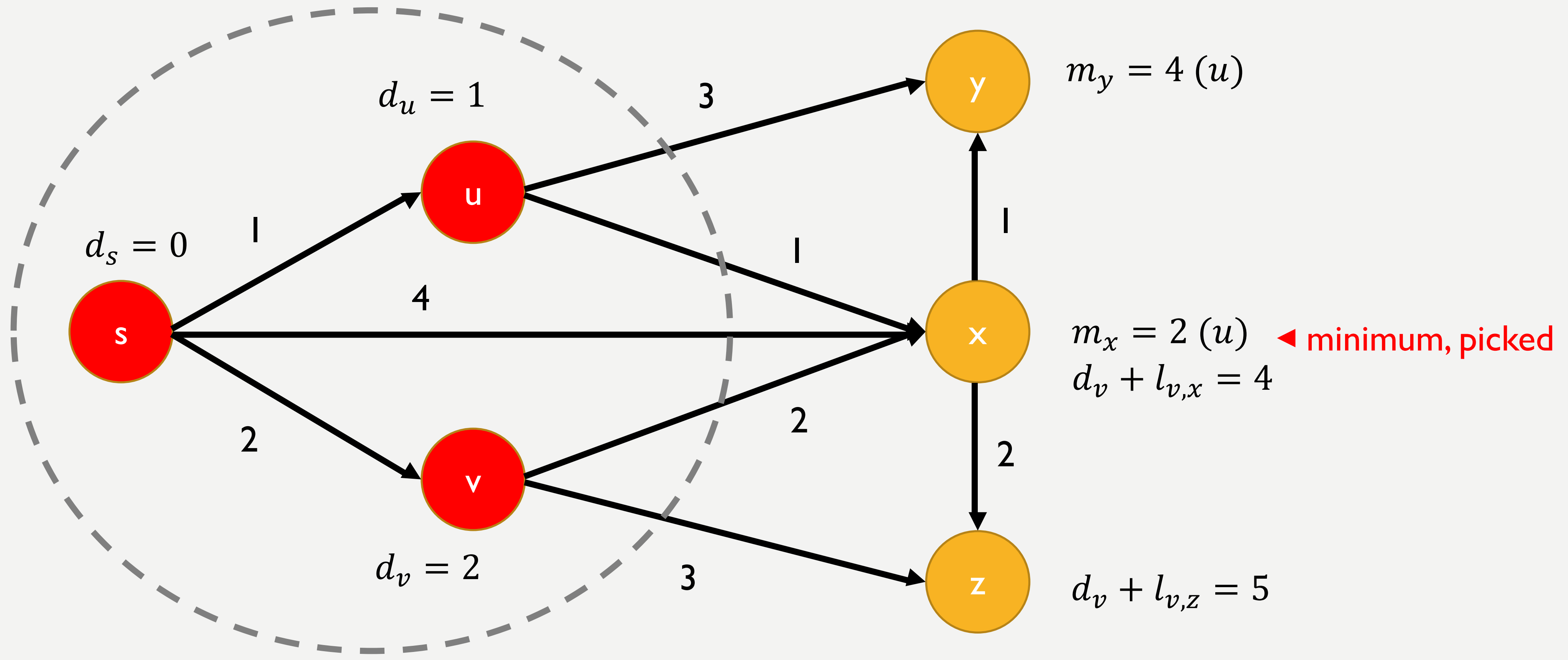


THE SHORTEST PATH PROBLEM

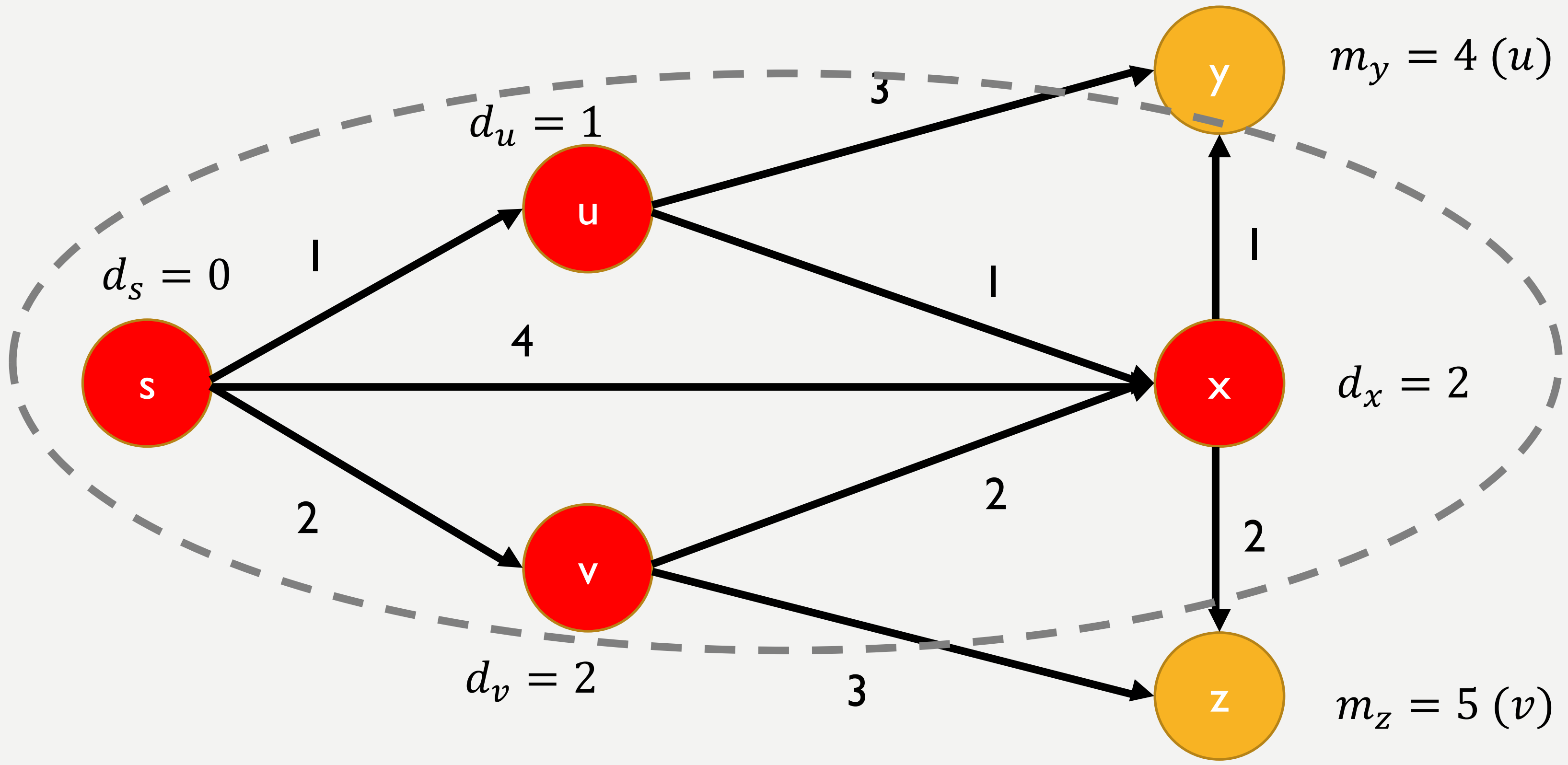


minimum, picked
(break ties arbitrarily)

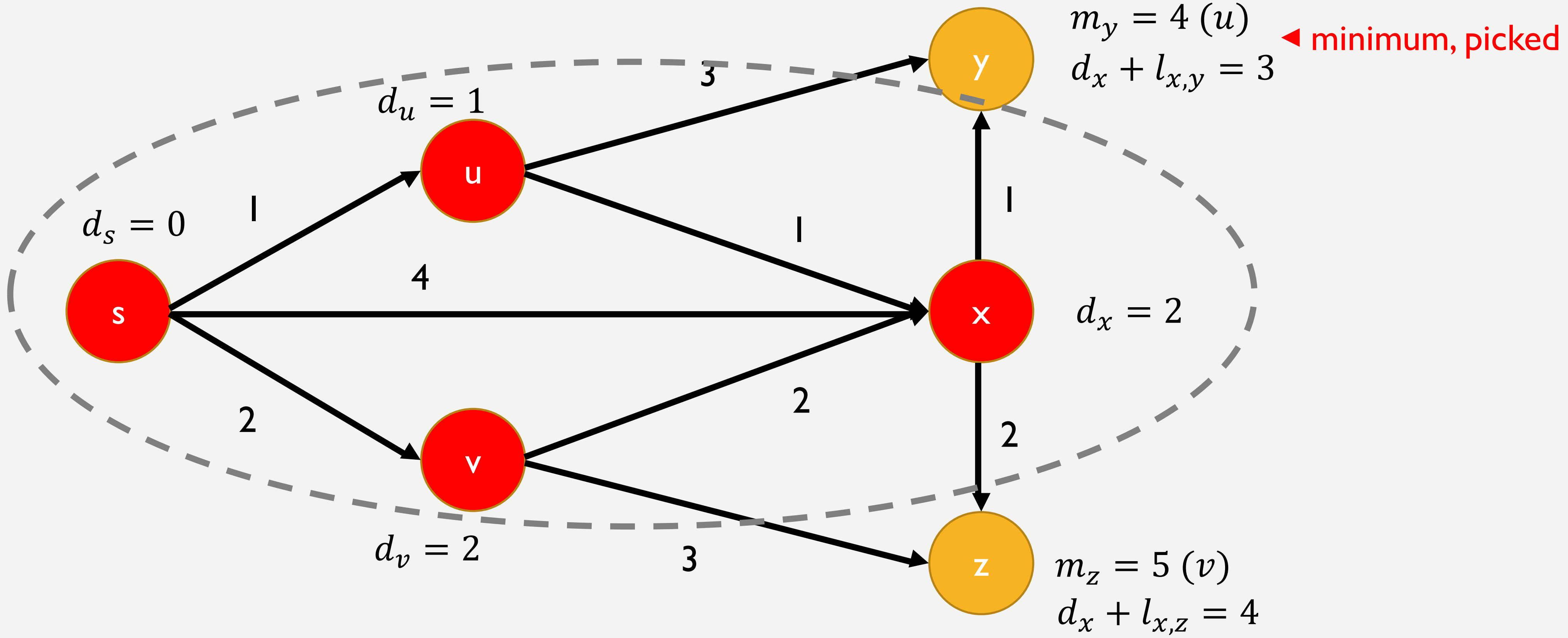
THE SHORTEST PATH PROBLEM



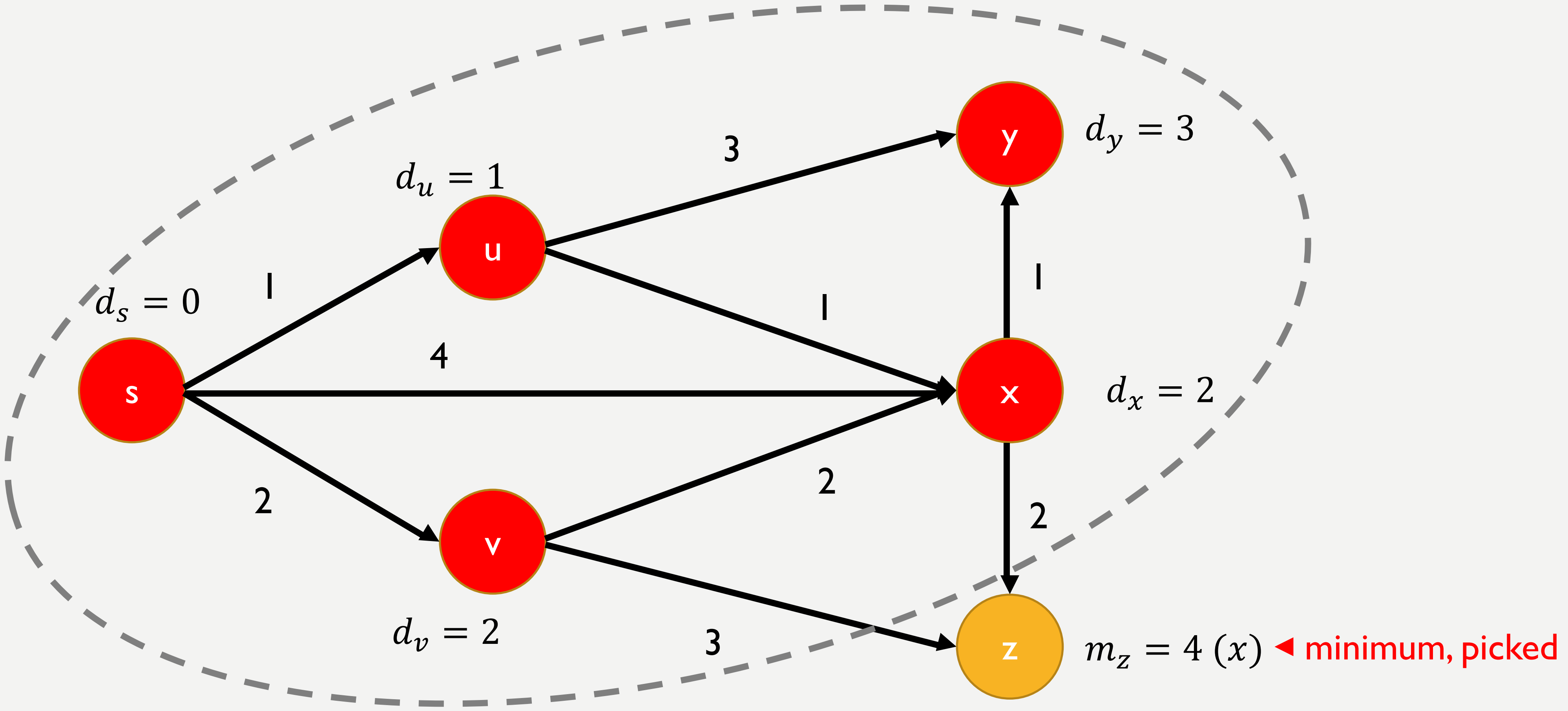
THE SHORTEST PATH PROBLEM



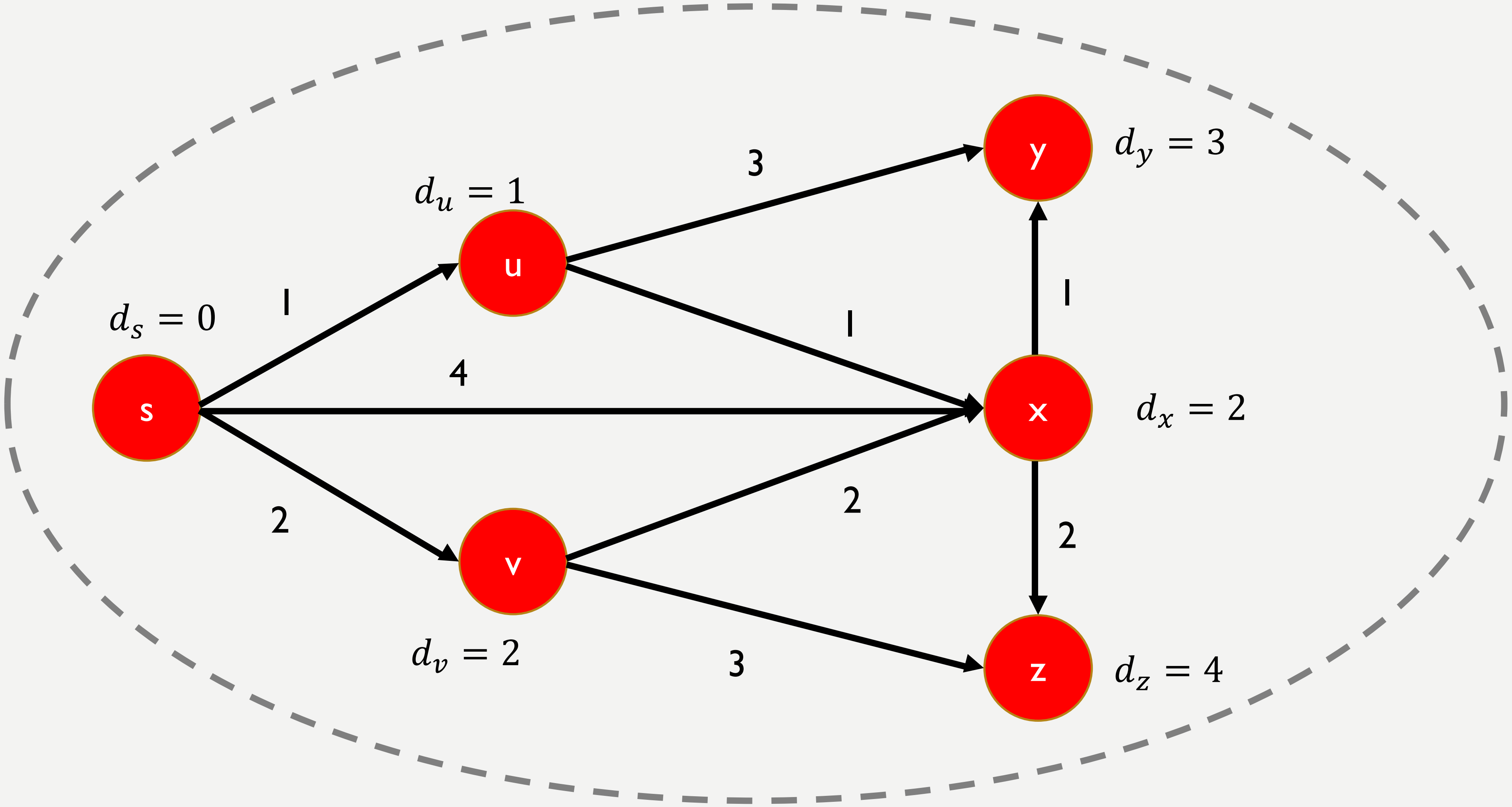
THE SHORTEST PATH PROBLEM



THE SHORTEST PATH PROBLEM



THE SHORTEST PATH PROBLEM



DONE!

THE SHORTEST PATH PROBLEM

- The algorithm seems to work on the previous example.
- However, we cannot prove the correctness of the algorithm only by example, we need more rigorous proof.
- According to our selection criteria $\underset{v,e=(u,v):u\in S}{\operatorname{argmin}} (d(u) + l_e)$, is the quantity $d(u) + l_e$ the smallest among the lengths of all paths between s and v?

THE SHORTEST PATH PROBLEM

- Proposition: Given a graph $G(V, E, L)$ with no negative edge, for any node v selected by the criteria $\underset{v,e=(u,v):u\in S}{\operatorname{argmin}} (d(u) + l_e)$, the quantity $d(u) + l_e$ corresponds to an $s - v$ path, which is the shortest $s - v$ path.

- Proof: We will use math induction for the proof.

  (Base case): The statement is clearly true when $v = s$, if we set $d(s) = 0$ and $l_{i,i} = 0$ for arbitrary node i. That is, the shortest path from s to s is null and its length is 0, and it is true per definition.

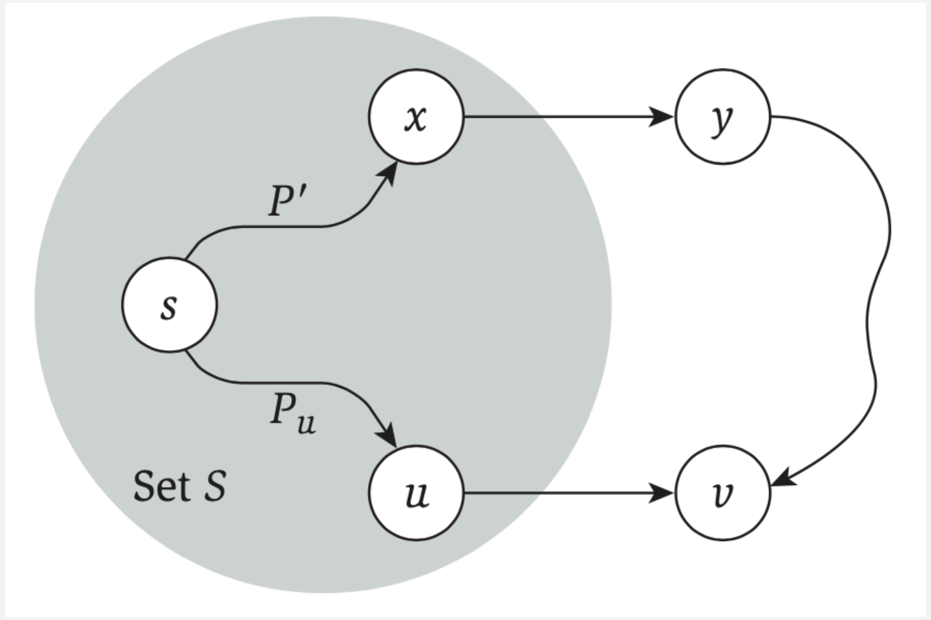
  (Induction): Let S be a set of vertices that have been visited. We assume for any node $u \in S$ we have $d(u)$ being the shortest path length between s and u, and the corresponding shortest path is known. Note that initially $S = \{s\}$ and the hypothesis is true.

THE SHORTEST PATH PROBLEM

- Proof (Dijkstra's algorithm continued):

  (Induction cont.): Per the algorithm, we will pick the node v, where $\underset{v,e=(u,v):u\in S}{\operatorname{argmin}} (d(u) + l_e)$.

  We compare the greedy path P_u with a length of $d(u) + l_e$ and another arbitrary $s - v$ path P' (generalization of a generic particular). Since the path P' connects $s \in S$ and $v \in V - S$, P' must contain an edge $e_{x,y}$ where $x \in S$ and $y \in V - S$. See the figure on the right.

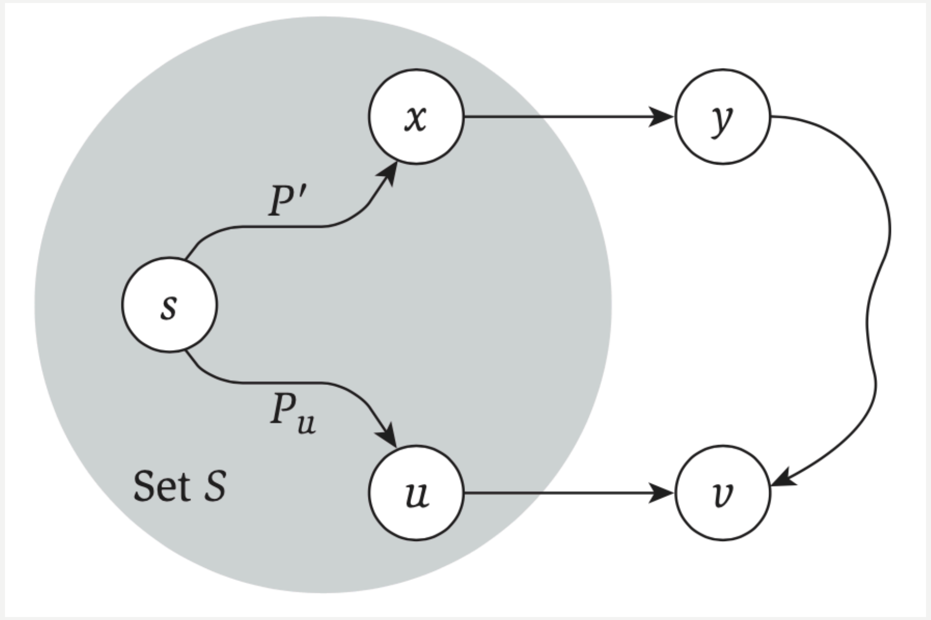


Algorithm Design, by Kleinberg and Tardos

THE SHORTEST PATH PROBLEM

- Proof (Dijkstra's algorithm continued):

  (Induction cont.): Note that it is possible that $x = u$ or $y = v$, but the equalities do not both hold at the same time because it would directly lead to $d(u) + l_e$ correspond to the shortest path (what we need to prove). Now, per the selection criterion, we have $d_u + l_{u,v} \leq d_x + l_{x,y}$. Also note that the graph contains no negative-weighted edge, hence $d(P_{y,v}) \geq 0$. Adding the term to the right-hand-side (RHS) we have $d_u + l_{u,v} \leq d_x + l_{x,y} + d(P_{y,v})$. Or, $d(P_u) \leq d(P')$.



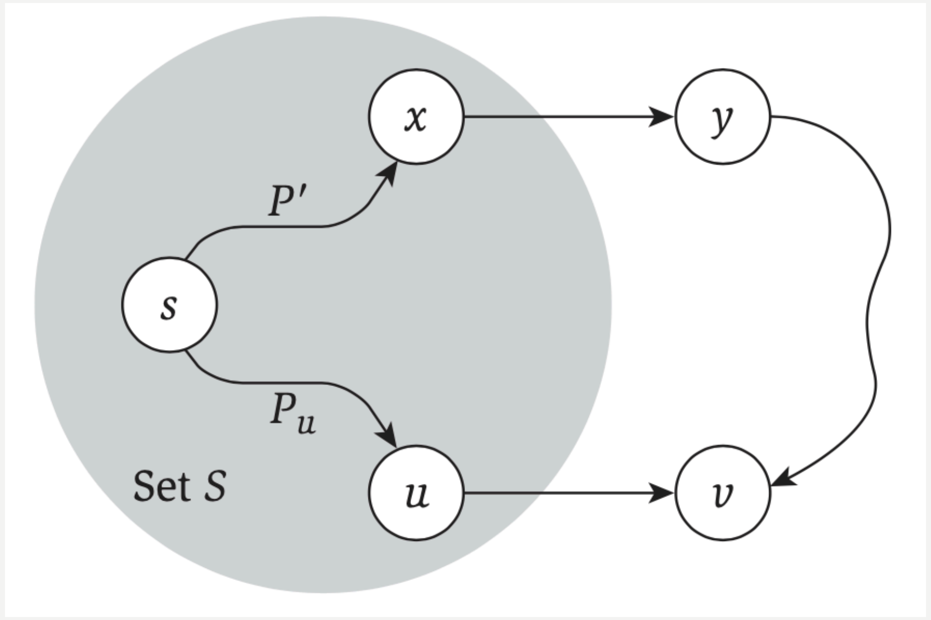
Algorithm Design, by Kleinberg and Tardos

THE SHORTEST PATH PROBLEM

- Proof (Dijkstra's algorithm continued):

  (Induction cont.): Since P' is an arbitrary $s-v$ path, it indicates that P_u is shorter than or has the same length as any valid $s-v$ path. Note that P_u can be constructed by appending the edge $e_{u,v}$ to the path corresponding to $d(u)$, which exists per induction hypothesis.

  (Conclusion): Therefore, for the node v selected by the criteria $\underset{v,e=(u,v):u\in S}{\operatorname{argmin}} (d(u)+l_e)$, the shortest $s-v$ path has a length of $d(u)+l_e$.



Algorithm Design, by Kleinberg and Tardos

THE SHORTEST PATH PROBLEM

- With the proven proposition, it is trivial to prove that Dijkstra's algorithm is correct for both the source-target version and the single-source version.

- We have proved the correctness of the Dijkstra's algorithm under the math induction framework. To complete the induction step, we used direct prove with the generalization of generic particular technique. (It is common to combine multiple prove techniques.)

- **To prove the correctness of greedy algorithms, it is common to compare the greedy solution with arbitrary alternative solutions (or a hypothetical non-greedy optimal solution), and show that the greedy solution works better in each step.** (We will see this over and over again in this module.)

THE SHORTEST PATH PROBLEM

- The next step is to analyze the time complexity of the Dijkstra's algorithm.
- Recall the algorithm:

```
Initialize $S = \{s\}$ and $d(s) = 0$;                // $O(1)$ time
While $S \neq V$                                        // how many iterations do we have
    Select a node $v \notin S$ with at least one edge from $S$ for which
        $d(v) = \min_{e=(u,v):u \in S} (d(u) + l_e)$;   // how do we effectively select the node
    Add $v$ to $S$, record $d(v)$;                     // $O(1)$ time
Endwhile
```

THE SHORTEST PATH PROBLEM

- How many iterations do we have?
  - Since at each iteration, we will add one node to the set S, and we only $|V|$ nodes, hence we have $O(|V|)$ iterations

- How do we effectively select the node?
  - note the algorithm only needs to extract the minimum quantity each time, which can be achieved using the data structure **priority queue**
  - at each iteration, a new node v is added to S. Hence, we need to compute all quantities relating to v and add those to the priority queue.
  - on average, there are k (the average degree) edges associated with the node v; and insertion to a priority queue takes $O(\log|V|)$. (Because each of such quantity is associated with a unvisited node.)
  - therefore, we need $O(k \log|V|)$ time to select the node

THE SHORTEST PATH PROBLEM

- Taken together, the overall time complexity of the Dijkstra’s algorithm is $O(|V|) * O(1 + k \log |V|)$.

- Note that we have $k|V| = O(|E|)$, the time complexity thus becomes $O(|V| + |E| \log |V|)$.

THE MINIMUM SPANNING TREE PROBLEM

- Given a graph $G = (V, E, W)$ where G is connected and $l_e \geq 0$ for all $e \in E$, we would like to select a subset of edges $T \subseteq E$ such that all vertices can be connected through T and that the total cost of $\sum_{e \in T} l_e$ is minimized.

- The math objective of the problem is to minimize the sum of the weights of the edges contained in the subset T.

THE MINIMUM SPANNING TREE PROBLEM

- Note that in the problem formulation, the subset of edges T is not defined as a tree, but we will show later that T actually corresponds to a tree, or it contains an equal-weighted tree as its subset.
  - since we assume all edge weights are non-negative, removing edges will not increase the total cost (the math objective we wish to minimize)
  - so, if T is not a tree, then per definition it contains some cycles.
  - we can break the cycles by removing some edges; the operation keeps the solution intact (not increasing the weight nor leaving some nodes unconnected)
  - hence, even T is not a tree, we can still reduce it to a tree

THE MINIMUM SPANNING TREE PROBLEM

- Example: Imagine that you represent a communication network using a graph. The weight of the edges is the cost of transmitting information between the two sites. Now, you want to broadcast some information to all sites in the network with minimized total cost.

- Example: Imagine that you are comparing a set of species by measure their similarity by either their morphological appearance or DNA sequences. You can represent the results by a graph, with nodes representing the species and the edge weights representing their similarity. Since the nature is smart and always select the most efficient way to create one from the other, finding the MST from the graph can give hints to the evolution among the set of species.

THE MINIMUM SPANNING TREE PROBLEM

- We are going to design an algorithm under the greedy schema.

- Note that our goal is to connect the entire graph while minimizing the total weight.
  - imagine the very-first step: we select an edge and it connects two nodes.
  - obviously we should select the edge with the smallest weight
  - this immediately leads to a simple algorithm: sort the edges by ascending weight, and select one-by-one
  - a more sophisticated consideration notices that, if the selected edge connects two nodes that are already connected, we can discard the edge to reduce the total weight

THE MINIMUM SPANNING TREE PROBLEM

```
Kruskal's algorithm ($G, W$):
Sort edges weights so that $w_1 \leq w_2 \leq \ldots \leq w_m$;
$T \leftarrow \emptyset$;
foreach ($u \in V$) make a set containing singleton $u$;
for $i$ = 1 to $m$
        $(u, v) = e_i$;
        if ($u$ and $v$ are in different sets)
                $T \leftarrow T \cup \{e_i\}$
                merge the sets containing $u$ and $v$
        endif
endfor
return $T$
```

THE MINIMUM SPANNING TREE PROBLEM

- To prove the correctness of the Kruskal's algorithm, we will first prove that every edge selected by the greedy algorithm is correct; that is, each selected edge is a part of an MST (minimum spanning tree) of the graph.

- Lemma: Assume that all edge costs are distinct (will be relaxed later). Let S be any subset of vertices that is neither empty nor equal to all of V, and let edge $e = (v, w)$ be the minimum-cost edge with one end in S and the other end in $V - S$. Then every minimum spanning tree contains the edge e.

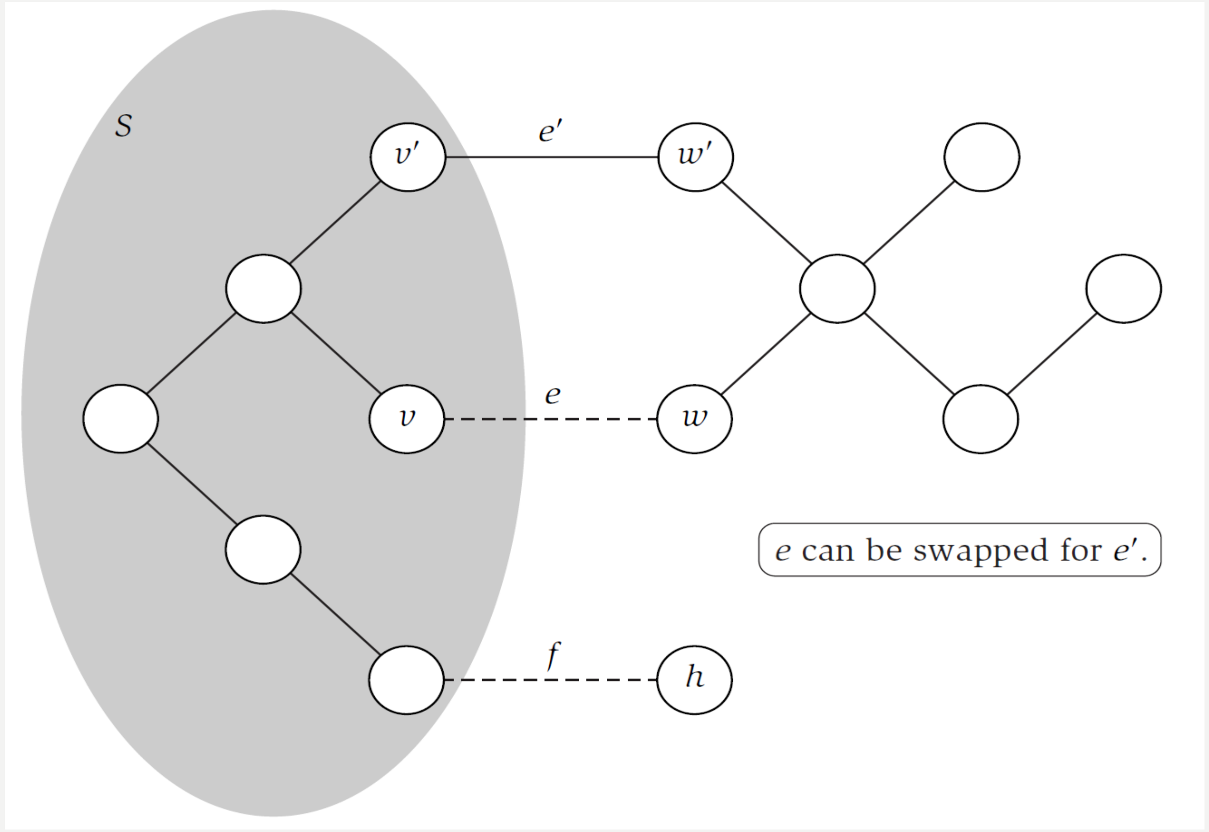
THE MINIMUM SPANNING TREE PROBLEM

- To prove the lemma, we will try to argue that every edge selected by the greedy algorithm is at least as good as any other edge.

- The high-level idea, independent of the MST problem, is to compare the solution provided by the greedy algorithm with arbitrary other solutions, and show that the greedy algorithm is at least as good as the the others. The technique is called "**exchange argument**", which is another common schema seen in proving the correctness of greedy algorithms.

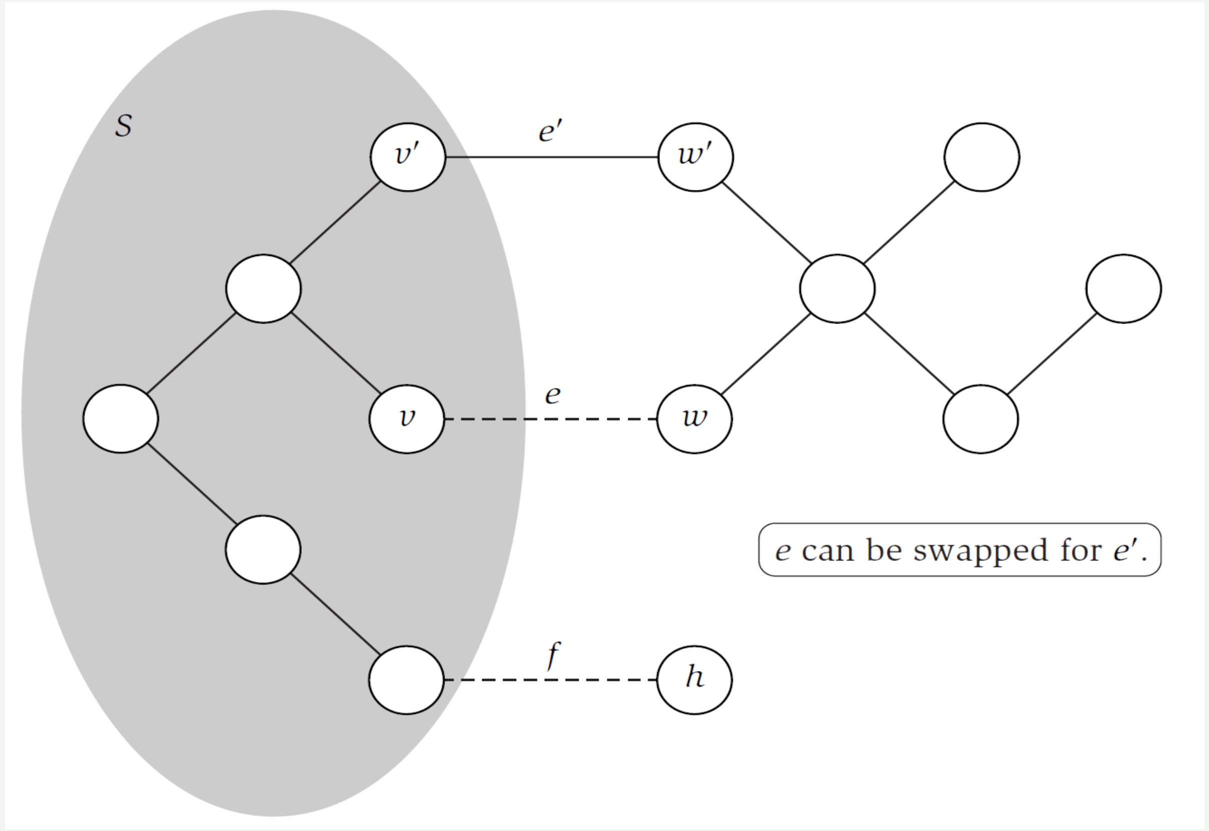
THE MINIMUM SPANNING TREE PROBLEM

- Proof: Consider the scenario shown in the right figure, where the greedy algorithm picked edge $e = (v, w)$, where $v \in S$ and $w \in V - S$. We assume that the MST of the graph is T and it does not contain e. (We use prove by contradiction.) Per the definition of MST, all nodes must be connected through it, including v and w. Since $e \notin T$, there must exist another path in T that connects v, w. Since $v \in S$ and $w \in V - S$, the path must contain an edge $e' = (v', w')$, where $v' \in S$ and $w' \in V - S$.



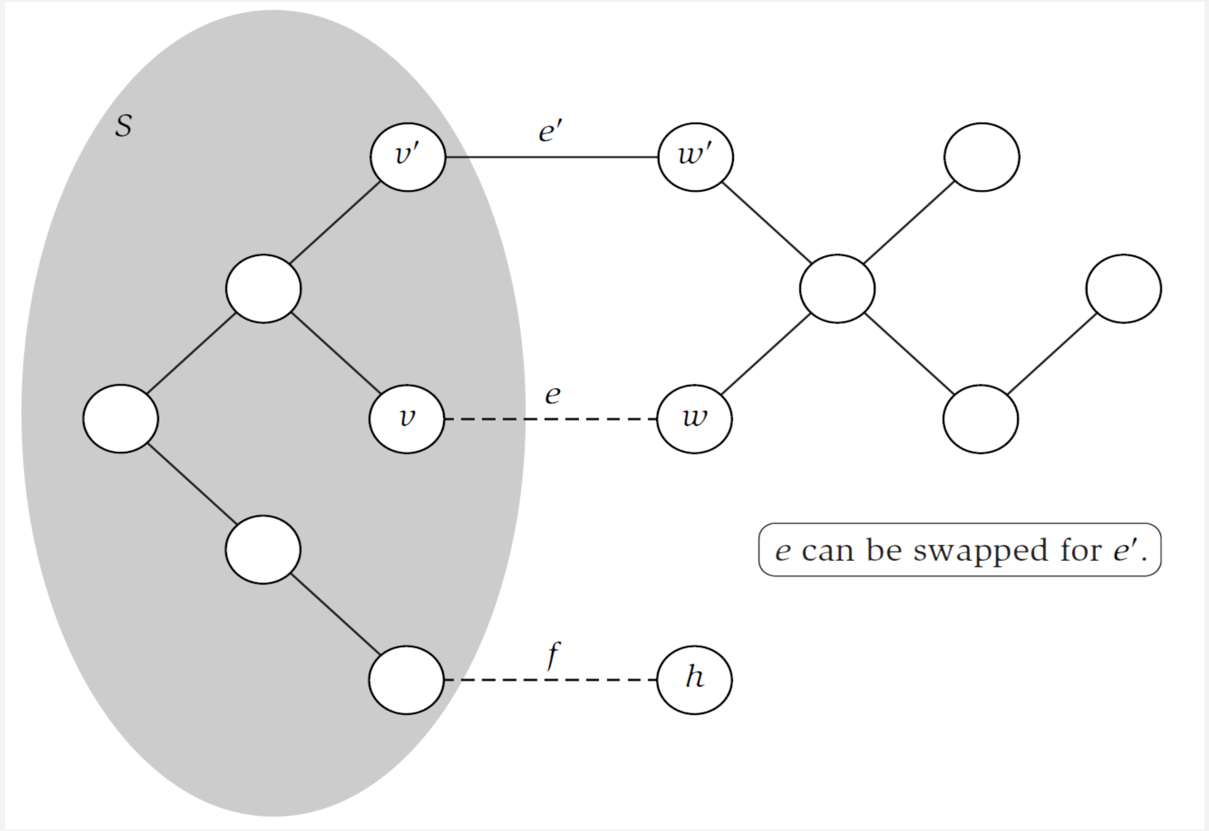
THE MINIMUM SPANNING TREE PROBLEM

- Proof: (Now we use the exchange argument.) Note that $e' \in T$. Let $T' = T - \{e'\} + \{e\}$ (replace e' with e in the MST). Clearly, $l_e < l_{e'}$ per the greedy selection criterion (and also because we assumed that all edges have distinct weights). It is then clear that $w(T') < w(T)$. We also argue that all nodes in G remained connected, since all connections in T that need to pass e' can now be redirected through e.



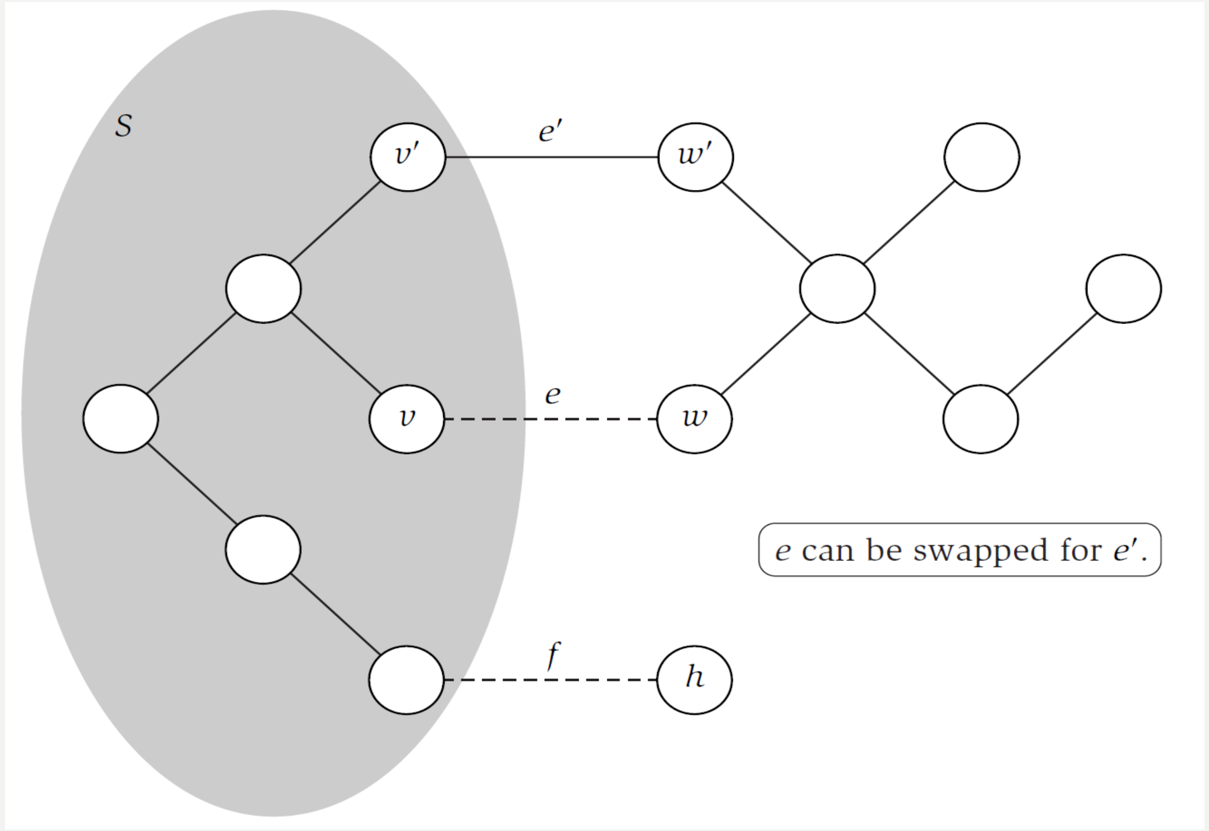
THE MINIMUM SPANNING TREE PROBLEM

- Proof: Specifically, let $P \subseteq T$ be the path that connects v, w. It follows that v, v' are connected and w, w' are connected. So, it is feasible to connect v', w' in T via $P_{v'-v,e,w-w'}$ in T'. (Discuss bidirectional cases for directed graphs.)



THE MINIMUM SPANNING TREE PROBLEM

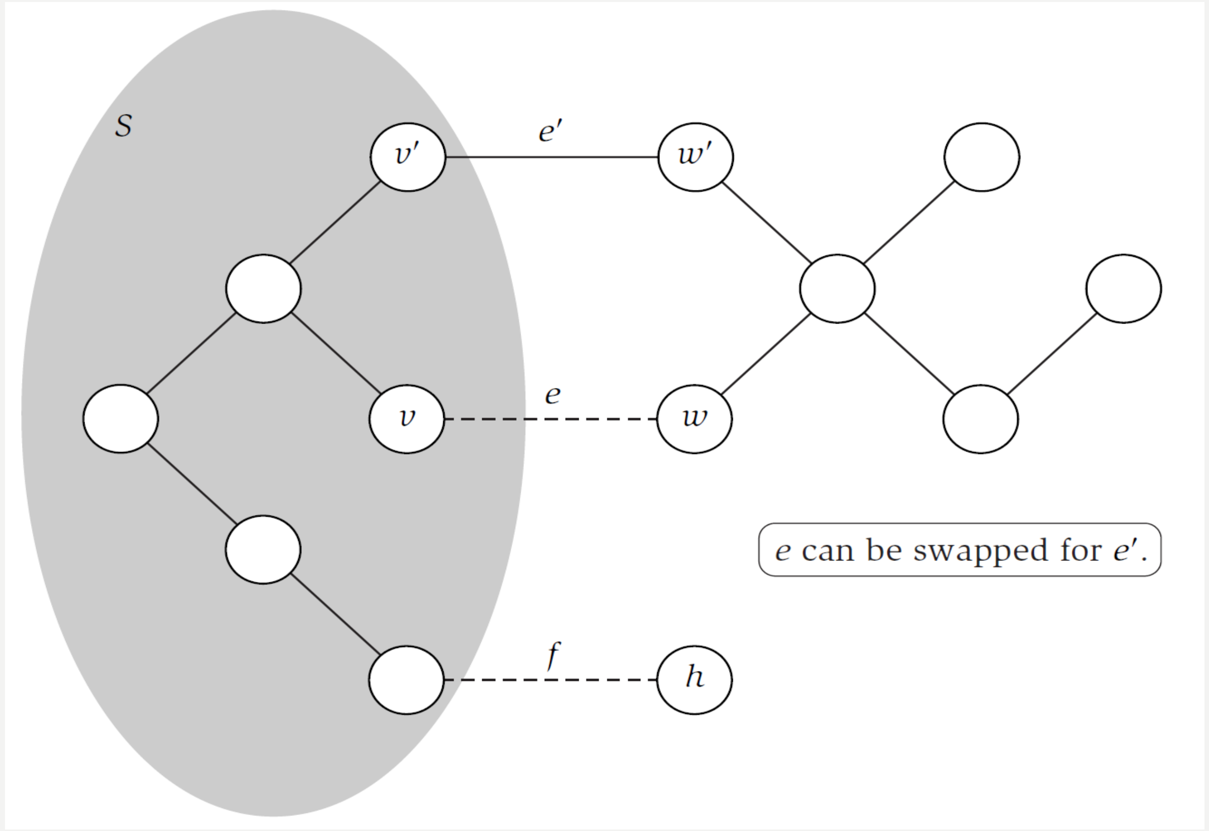
- Proof: We also claim that T' contains no cycle. Note that T is an MST, and it contains no cycle per definition. It indicates that P is the unique path that connects v, w. After removing $e' \in P$, no path can connect v, w any more. After adding e, we can only create a unique connection between v, w. Since e is the only added edge in T', the other parts of T' remained the same as T and contain no cycle either. Taken together, T' contains no cycle.



THE MINIMUM SPANNING TREE PROBLEM

- Proof: To summarize, we have shown that (1) $w(T') < w(T)$, (2) T' connects all nodes in G, and (3) T' contains no cycle. In this case, T' is a spanning tree that has a lower cost than T, which contradicts with the assumption that T is an MST. Therefore, any MST for G must contain e.

- The lemma is also called **cut property**, and is important for future studies of flow algorithms and duality (EECS 764).



THE MINIMUM SPANNING TREE PROBLEM

- Proposition: Kruskal's algorithm produces an MST of G.
- Proof: According to the proven cut property, each added edge by the Kruskal's algorithm must be contained in an MST of G. We also claim that when the algorithm terminates, we have added sufficient amount of edges to form a spanning tree. This is true because the algorithm only terminates when all nodes are connected.

THE MINIMUM SPANNING TREE PROBLEM

- Now, we need to relax the constraint (recall the assumed condition of the cut property) and allow equally-weighted.

- We are going to prove that, the spanning tree produced by the Kruskal’s algorithm remains an MST on graphs with equally-weighted edges.

THE MINIMUM SPANNING TREE PROBLEM

- Proposition: Kruskal's algorithm is correct on graphs containing equally-weighted edges.
- Proof: In a special case where all edges have the same weight, any spanning tree will be an MST. Since Kruskal's algorithm identifies a spanning tree, it also identifies an MST.

On the other hand, we have at least two edges that have different weights. Let the minimum difference between any two differently-weighted edge be d. We can make an alternative graph G', which has the identical node and edge set as G, but with its edge-weight ties broken by adding positive, arbitrarily small perturbations e. (For example, consider four edges with weights 1.9, 2, 2, 2. Then $d = 0.1$. We can set $e = 0.01$, and break the ties of the equally-weighted edges by reassigning their weights into 2 $(+0e)$, 2.01 $(+1e)$, 2.02 $(+2e)$). Note that we have at most $|V|$ equally-weighted edges, then we can set $e < d/|V|^2$ to satisfy the condition.

THE MINIMUM SPANNING TREE PROBLEM

- Proof (cont.): We then run the Kruskal's algorithm on G' and denote the result as T'. Since G' contains no equally-weighted edge, T' is an MST of G'. Also denote the true MST of G be T (which we don't know yet). Denote the weight of T' evaluated on the graph G be $w_G(T')$. (That is, the total weight after removing all the added perturbations.) There are three cases to discuss. First, when $w_G(T') < w_G(T)$. This is impossible since it contradicts with the assumption that T is the MST of G. Second, when $w_G(T') = w_G(T)$. Then T' is also an MST of G, so Kruskal's algorithm is correct.

THE MINIMUM SPANNING TREE PROBLEM

- Proof (cont.): Third, when $w_G(T') > w_G(T)$. In this case, $w_G(T') - w_G(T) < d$. Otherwise ($w_G(T') - w_G(T) \geq d$), we will have $w_{G'}(T) - w_G(T) < |V|^2 e < d \leq w_G(T') - w_G(T)$. Simplify the inequation we get $w_{G'}(T) < w_G(T')$. Also note that $w_G(T') \leq w_{G'}(T')$, since we only add positive perturbations to form G'. Taken together, we have $w_{G'}(T) < w_G(T') \leq w_{G'}(T')$. It indicates that T will be an MST of G', contradicting with the fact that T' is an MST of G'. (We take the just-proved fact that $w_G(T') - w_G(T) < d$.) Since d is defined as the smallest difference between any two differently-weighted edges, T' and T are identical.

THE MINIMUM SPANNING TREE PROBLEM

- Proof (cont.): In all possible cases, T' and T are identical. Hence, the Kruskal's algorithm correctly identifies an MST on graphs containing equally-weighted edges. (And what we need to do is to simply break ties arbitrarily when sorting edge weights.)

THE MINIMUM SPANNING TREE PROBLEM

- Time complexity analysis

```
Kruskal's algorithm ($G, W$):
Sort edges weights so that $w_1 \le w_2 \le \ldots \le w_m$;      // $O(|E|log|E|)$ time
$T \leftarrow \emptyset$;                                          // $O(1)$ time
foreach ($u \in V$) make a set containing singleton $u$;          // $O(|V|)$ time
for $i$ = 1 to $m$                                                  // $O(|E|)$ iterations
    $(u, v) = e_i$;                                                 // $O(1)$ time
    if ($u$ and $v$ are in different sets)                          // how to determine efficiently?
        $T \leftarrow T \cup \{e_i\}$                               // $O(1)$ time
        merge the sets containing $u$ and $v$                       // how to merge efficiently?
    endif
endfor
return $T$
```

THE MINIMUM SPANNING TREE PROBLEM

- The bottleneck of the time complexity analysis becomes:
  - how do we determine whether two vertices are in the same set
  - how do we merge two sets

- The two operations can be performed effectively using the **disjoint set (or the union-find)** data structure.
  - you should have already learnt it in EECS 560 Data Structures
  - determine whether two vertices are in the same set: $O(1)$ time
  - determine whether two vertices are in the same set: $O(\log |V|)$ time

THE MINIMUM SPANNING TREE PROBLEM

- When using array, say A, we will use $A[i]$ to store the ID of the correspondent for element i.
  - find(x): this is obviously $O(1)$; we just need to return $A[x]$
  - union(x, y): we will merge the two disjoint sets through relabeling; we will first retrieve the labels for the elements, i.e., $A[x]$ and $A[y]$, and relabel $A[y]$ to $A[x]$ if $A[x]$ has fewer members (or vice versa). We can prove that on average it takes $O(\log n)$ time.

1	1	1	3	3	5	5	5

items 0, 1, 2 are in set 1; items 3, 4 are in set 3; and items 5, 6, 7 are in set 5

THE MINIMUM SPANNING TREE PROBLEM

- Taken together, the time complexity is $O(|V|) + O(|E| \log|E|) + O(|E| \log|V|)$
  - If we assume the graph is very sparce where $|E| = O(1)$, the algorithm runs in $O(|V|)$
  - Otherwise, the algorithm runs in $O(|E| \log|V|)$

- In summary, the algorithm runs in $O(|V|) + O(|E| \log|V|)$.

THE MINIMUM SPANNING TREE PROBLEM

- Another greedy design idea to the MST problem:

- We can view the MST finding process as a "tree-growing" process. Similar to the Dijkstra's algorithm, we walk through the graph step-by-step.
  - we can also keep the set S, which contains the nodes that have been visited
  - at each step, we will try to reach a node that is not in S, through an edge with the cheapest cost (here is where we apply the greedy schema)
  - this is called **Prim's algorithm**

THE MINIMUM SPANNING TREE PROBLEM

Prim's algorithm (G, W):

```
$S \leftarrow \{s\}; T \leftarrow \emptyset$;
while $S \neq |V|$
        Select a node $v \notin S$ with at least one edge $e_c$ from $S$ for which
                $w(e_c) = \min_{e=(u,v):u \in S} (w(e))$;
        Add $v$ to $S$; Add $e_c$ to $T$;
endfor
return $T$
```

THE MINIMUM SPANNING TREE PROBLEM

- We need to prove that the Prim's algorithm is correct.

- A high-level, informal proof: (Note, it doesn't mean that you do not need to present formal proof in the exams if related questions were asked.)
  - note that per the algorithm, we maintain a set S that contains all nodes that have been visited
  - also, per the algorithm, we know that the edge we selected should have the minimum cost among all edges that span S and $V - S$
  - note that this is exactly the situation for the cut property; and we know that the edge selected by Prim's algorithm must be contained in an MST
  - the algorithm terminates when all vertices are included in S, hence the collection of edges is complete to form a spanning tree

THE MINIMUM SPANNING TREE PROBLEM

- Time complexity analysis:

```
Prim's algorithm ($(G, W)$):
$S \leftarrow \{s\}; T \leftarrow \emptyset$;                                  // $O(1)$ time
while $S \neq |V|$                                                            // how many iterations?
        Select a node $v \notin S$ with at least one edge $e_c$ from $S$ for which
                $w(e_c) = \min_{e=(u,v):u \in S}(w(e))$;                      // how to select effectively?
        Add $v$ to $S$; Add $e_c$ to $T$;                                      // $O(1)$ time
endfor
return $T$
```

THE MINIMUM SPANNING TREE PROBLEM

- The key to analyze the efficiency of Prim's algorithm is how to select the minimum-weighted edge.

- This can be done using Priority Queue, in a similar way as the Dijkstra's algorithm
  - Note that the stored values are different
  - Prim's algorithm only needs to store the edge weight, not the sum of the existing minimum distance and the edge weight

- So, Prim's algorithm has the same time complexity as the Dijkstra's algorithm: $O(|V| + |E| \log |V|)$.

THE MINIMUM SPANNING TREE PROBLEM

- **It is important to note that: there could be multiple ways to solve a problem, as long as you can prove they are correct (and efficient).**
  - Kruskal's algorithm
  - Prim's algorithm

- Interpretability (can you explain the idea easily to your peers?) and implementational difficulty (is it easy to program?) should be considered when choosing between algorithms.