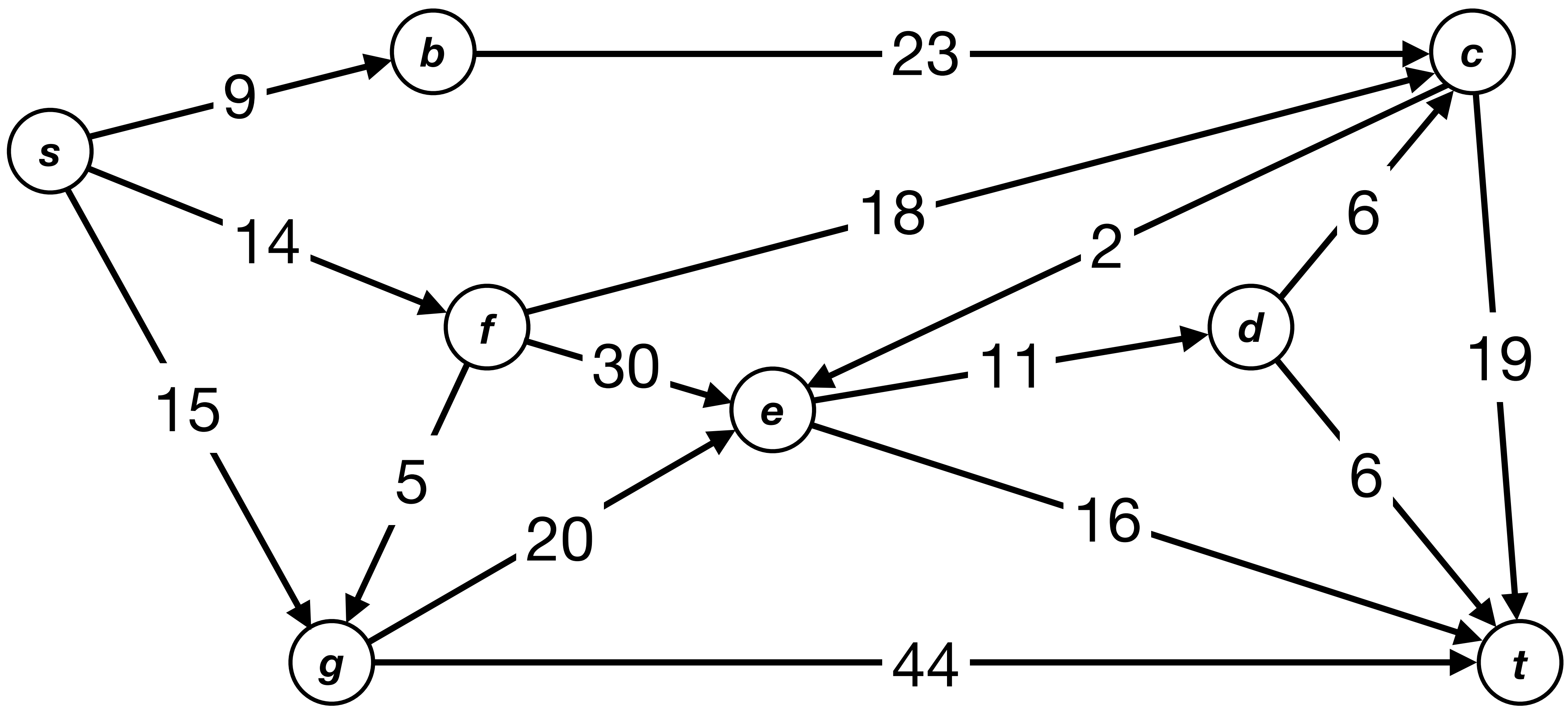
COMP3251
Lecture 9: Dijkstra Algorithm
(Chapter 4.3, 4.4, and 4.5)

Edges with Lengths

We now consider weighted graph, i.e., every edge *(u,v)* is associated with a length *L(u,v)*.

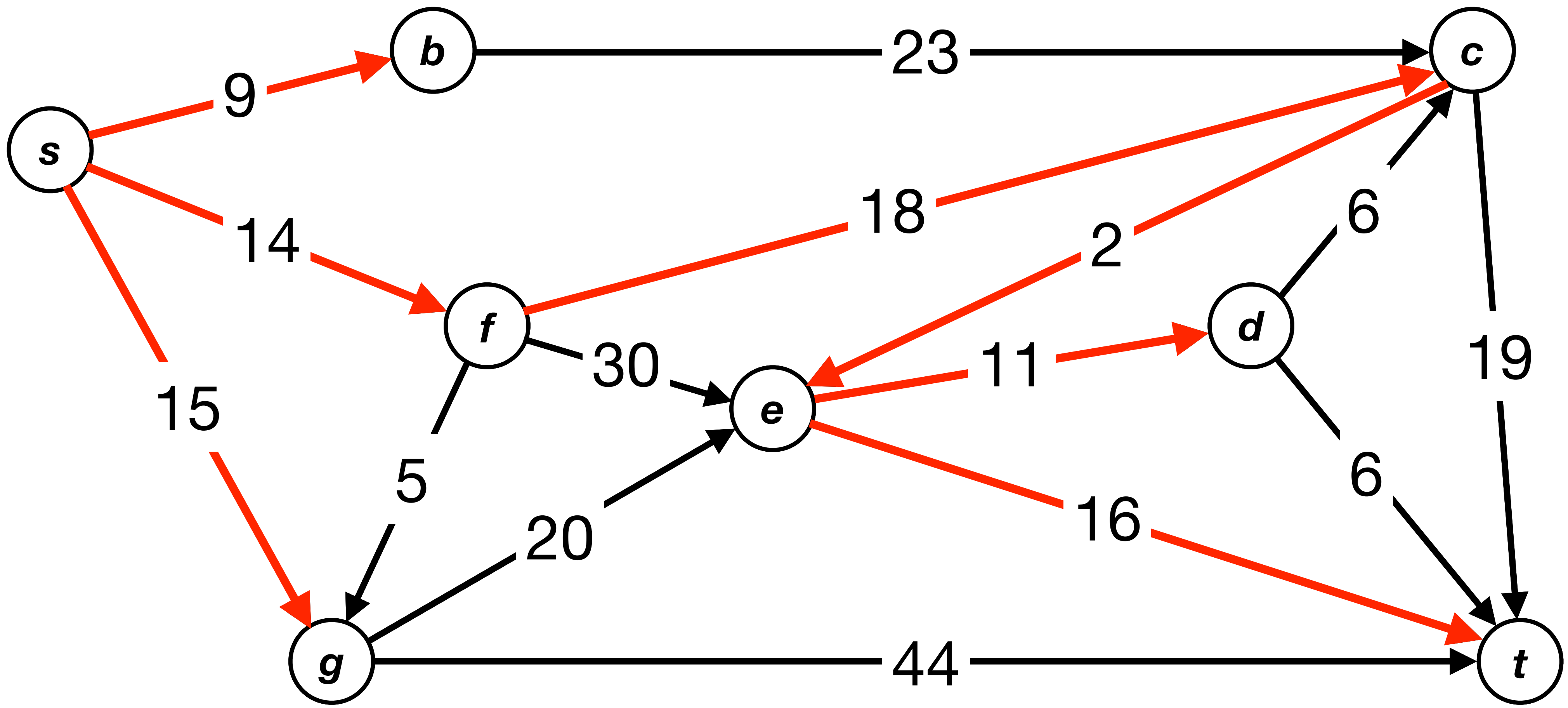


Single-Source Shortest Paths Problem (for Weighted Graphs)

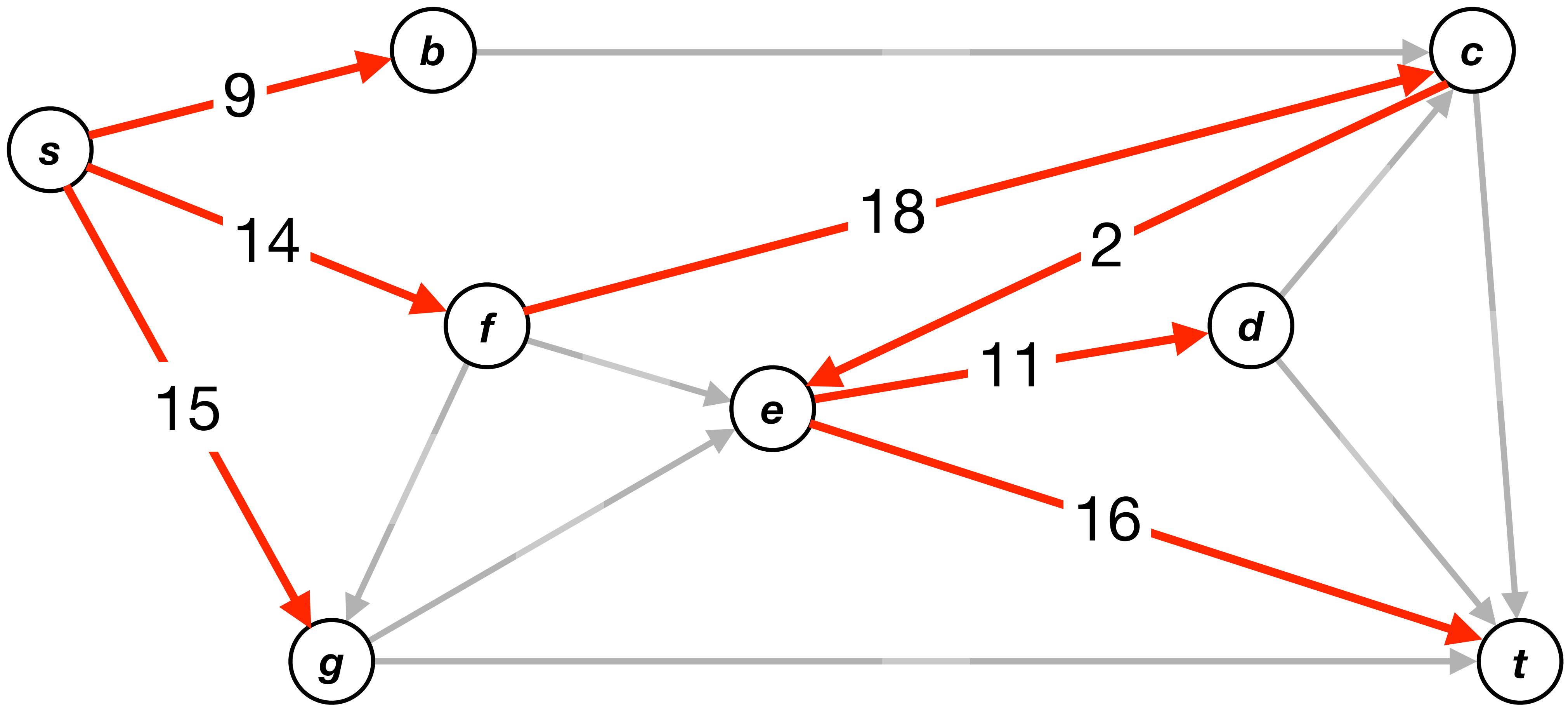
Given a weighted directed graph and a source vertex $s \in V$, find, for every vertex $v \in V$, the length of the shortest path from the source s to v.

Definition: length of a path P is equal to the sum of the length of the edges on P.

An Example



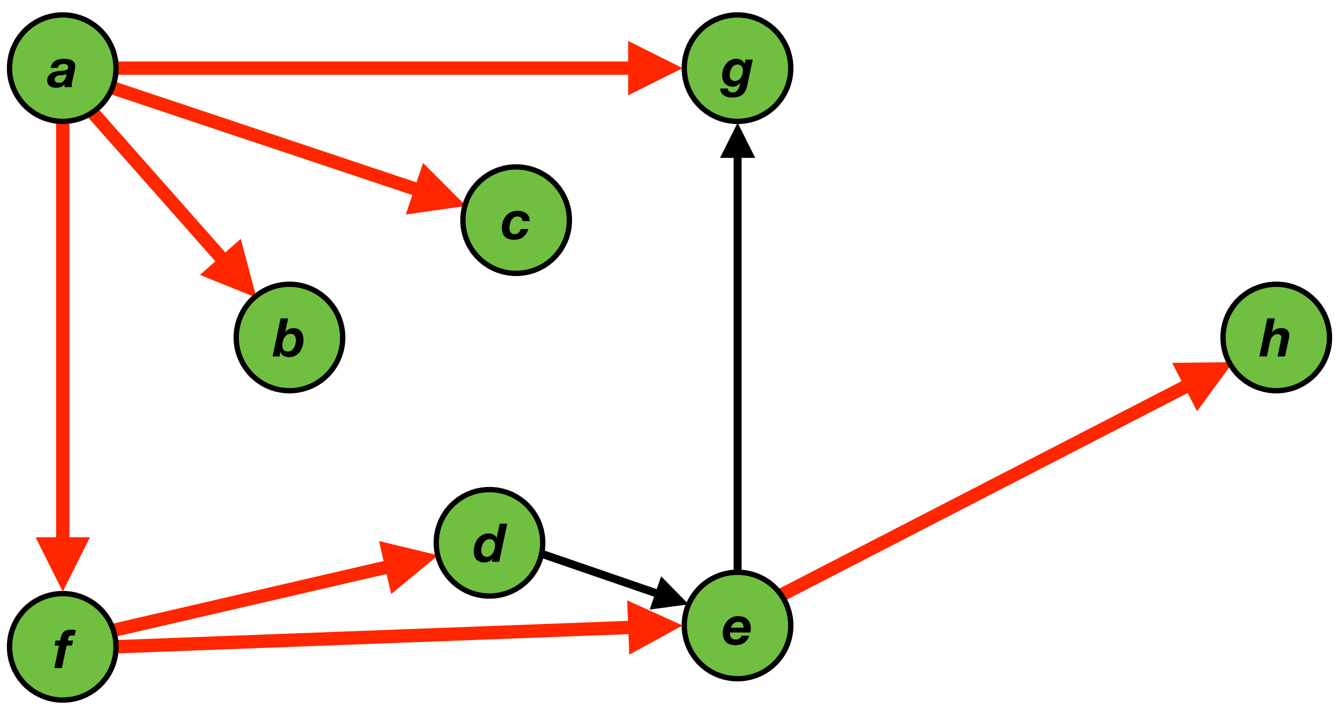
An Example



Note that this is a tree, and for any vertex *v*, the tree path from *s* to *v* is the shortest path *s* to *v*. Such a tree is called the Shortest Path Tree (SPT).

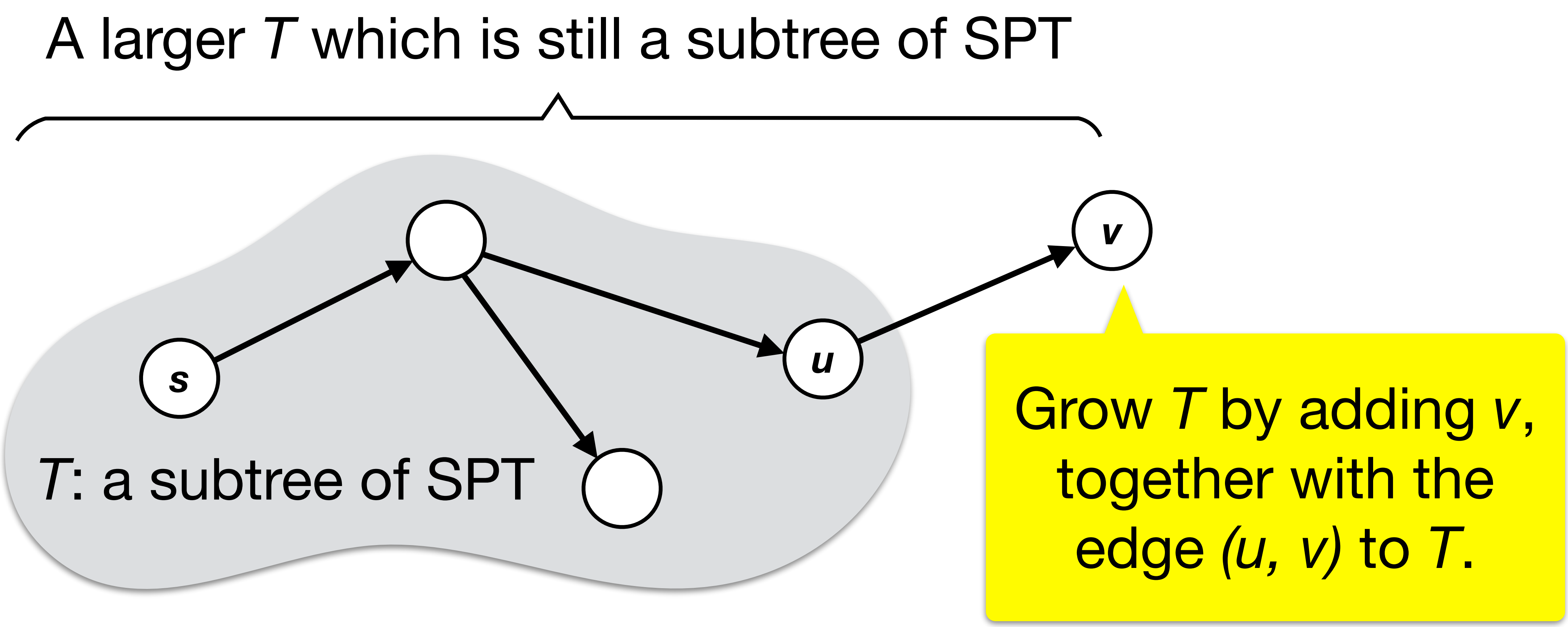
The BFS tree is a Shortest Path Tree

- If all edges have length 1, then BFS solves the shortest path problem and the BFS tree coincides with the SPT tree.
- An alternative view of BFS:
  - Starting from the smallest subtree of BFS tree that contains only the starting vertex *s*.
  - Add vertices and the corresponding tree edges one by one in ascending order of their distance from *s*.



General Idea

- Starting from the smallest subtree of SPT that contains only s.
- Iteratively attached a "correct" edge to the subtree such that the larger tree is still a subtree of SPT.
- When we include all vertices in the subtree, it is the SPT.

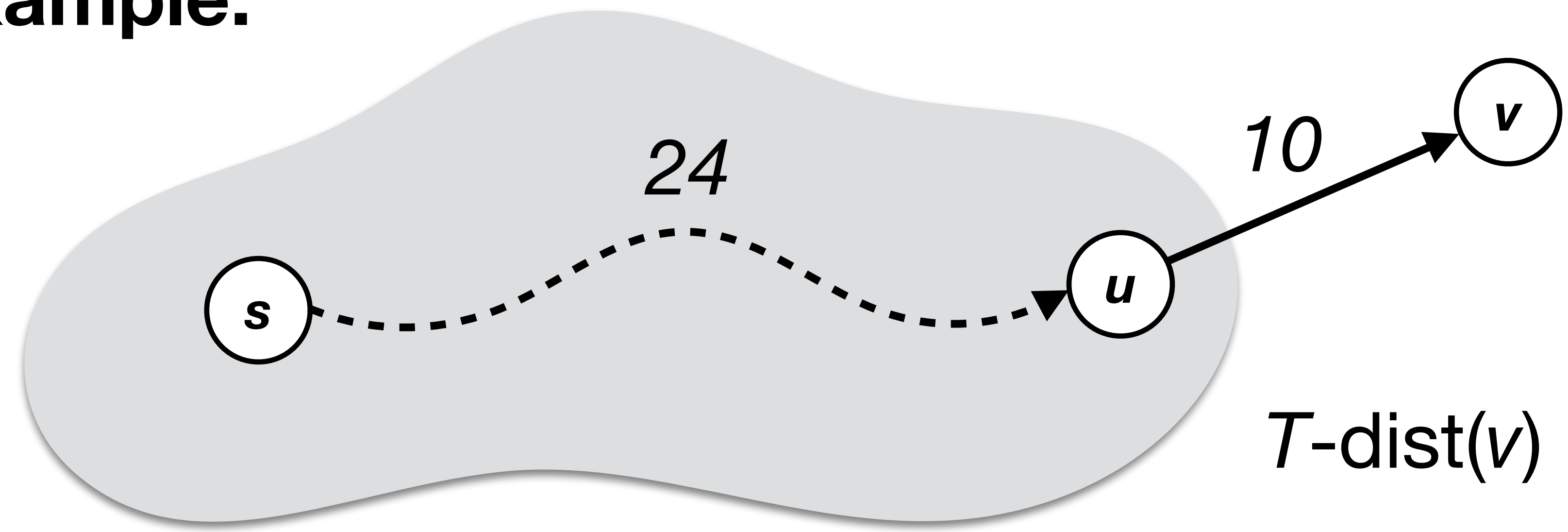


What is the correct vertex/edge?

Let T be a subtree of the SPT. Here are a few **definitions**:

- A T-path from s to v is a path from s to v that goes through only vertices in T.
- The T-dist(v) is the length of shortest T-path from s to v.
- Note that for any vertex v, T-dist(v) ≥ dist(v), the length of the shortest path from s to v.
- For every $v \in T$, T-dist(u) = dist(u).

Example:



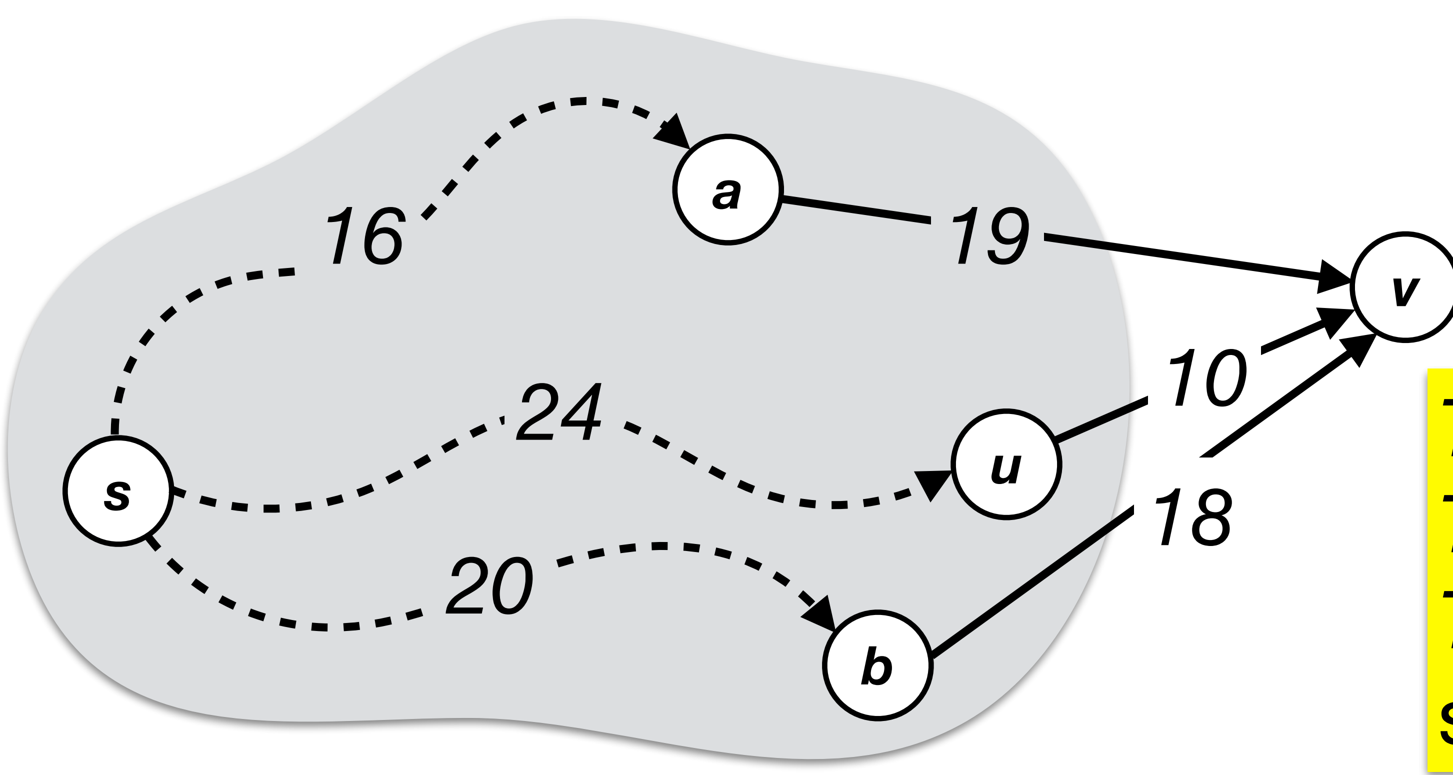
T-dist(v) = $24 + 10 = 34$

What is the correct vertex/edge?

Observation: A shortest T-path from s to v, must be consists of a shortest T-path from s to some vertex $u \in T$ and edge (u, v).

- To determine T-dist(v), we check all vertices $u \in T$ such that $(u, v) \in E$, and the corresponding shortest T-path from s to u.
- Then, determine their total lengths and return the minimum.

Example:



T-dist(a) + $L(a,v) = 35$,
T-dist(u) + $L(u,v) = 34$,
T-dist(b) + $L(b,v) = 38$,
so T-dist(v) = 34.

The Algorithm (Informal)

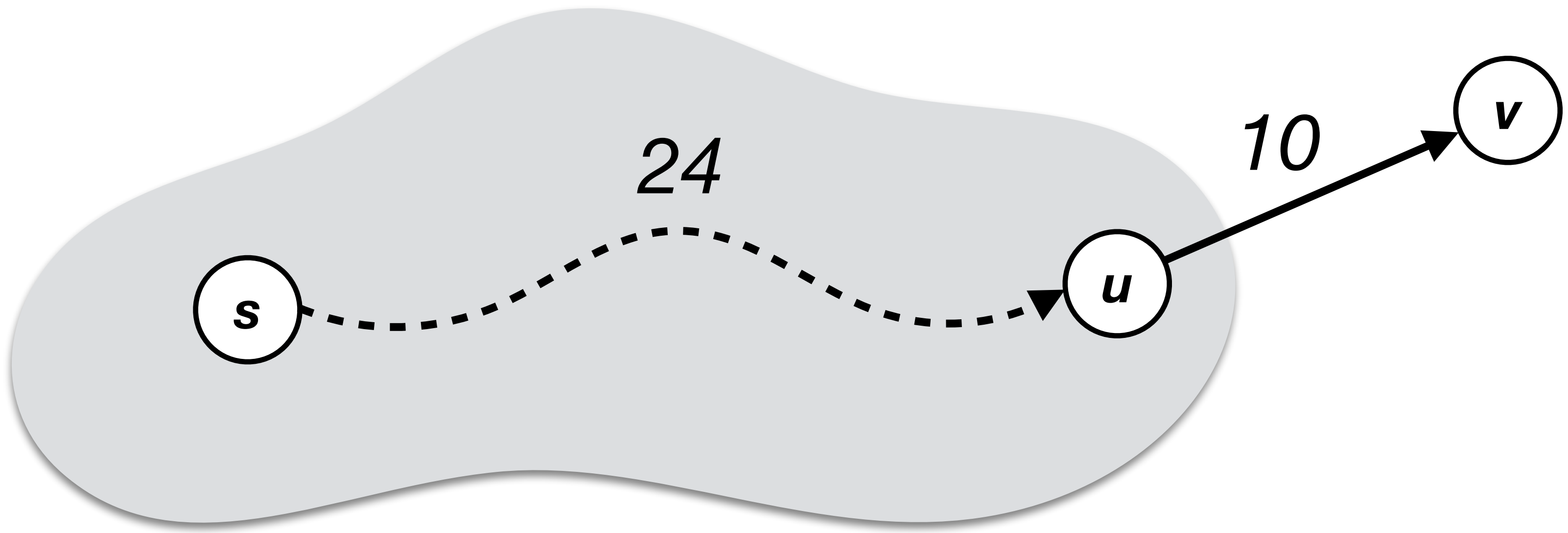
Starting with $T = \{ s \}$ and repeatedly do the following to enlarge T until T includes all vertices:

- Find the vertex $v \notin T$ with the smallest T-dist(v), add it to T;
- Update the T-dist(x) for all vertices $x \notin T$.

What next?

- Why is the above algorithm correct?
- What is its running time?

Correctness



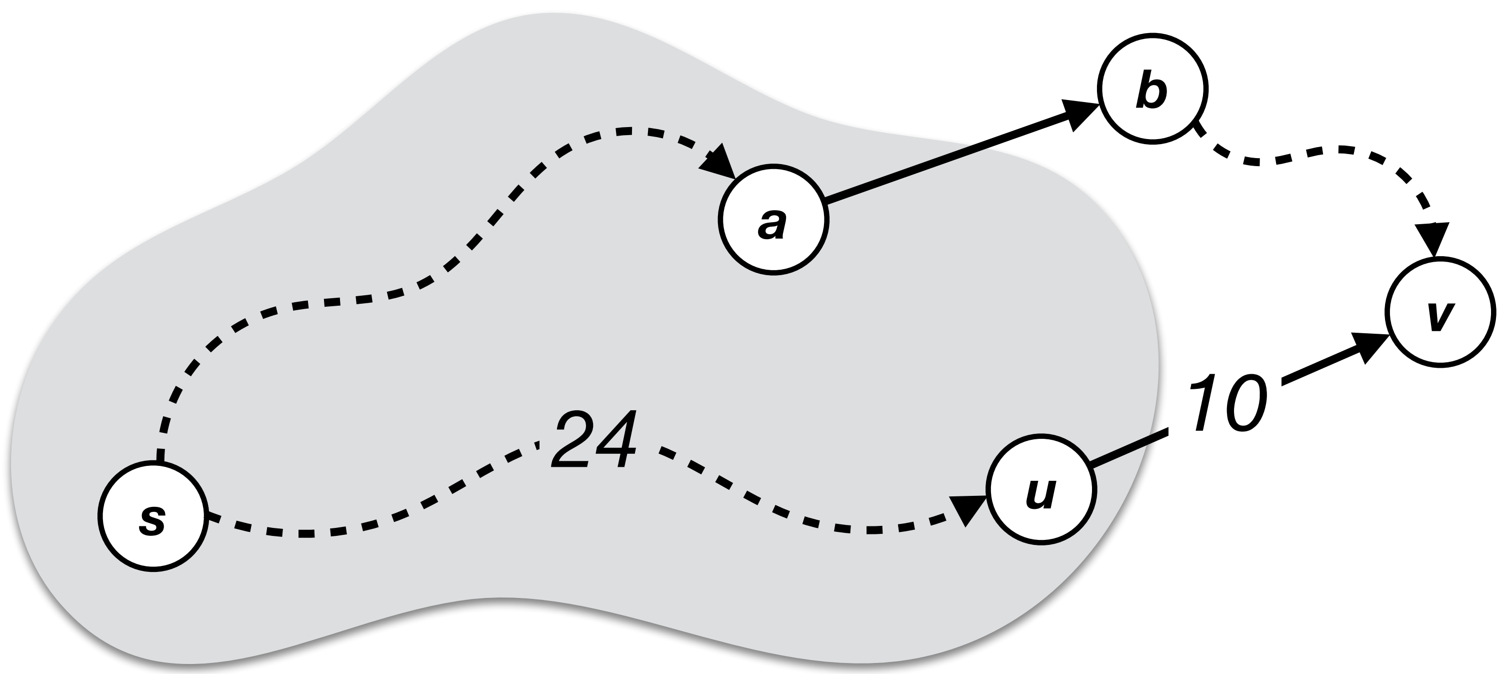
Suppose:

- T is a subtree of SPT.
- v has the smallest T-dist among vertices that are not in T, the the path leaves T through a vertex $u \in T$.

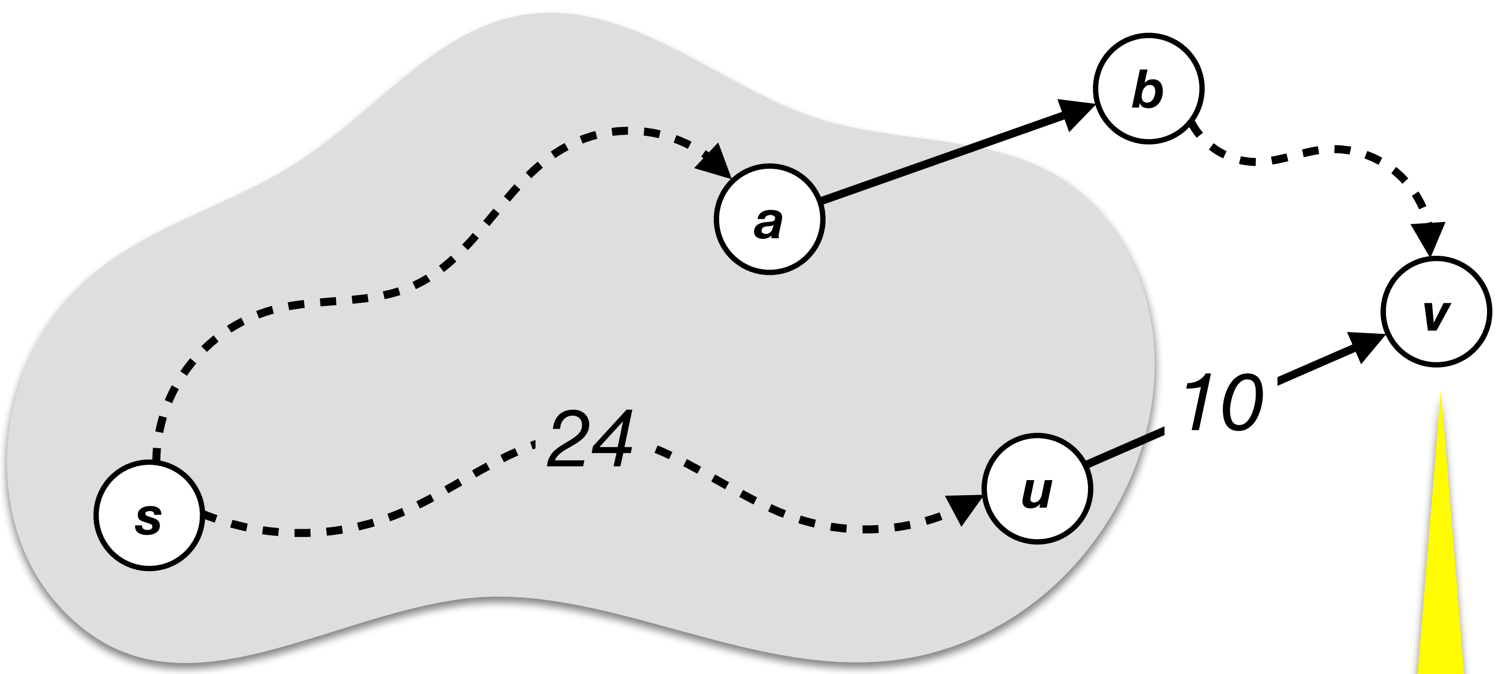
Want to show:

- This path from s to v is indeed the shortest path.

Correctness (cont'd)



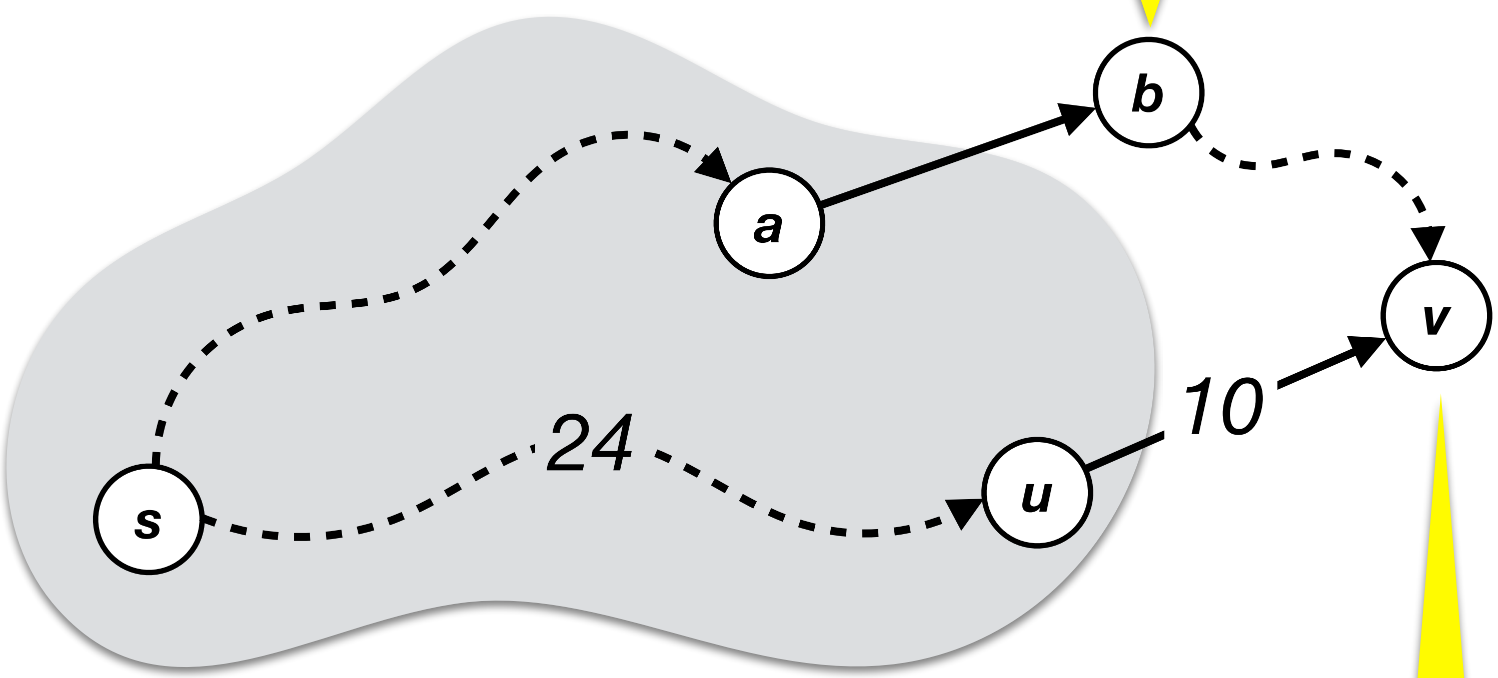
Correctness (cont'd)



By our choice of v, for all $x \notin T$, T-dist(v) $\leq$ T-dist(x)

Correctness (cont'd)

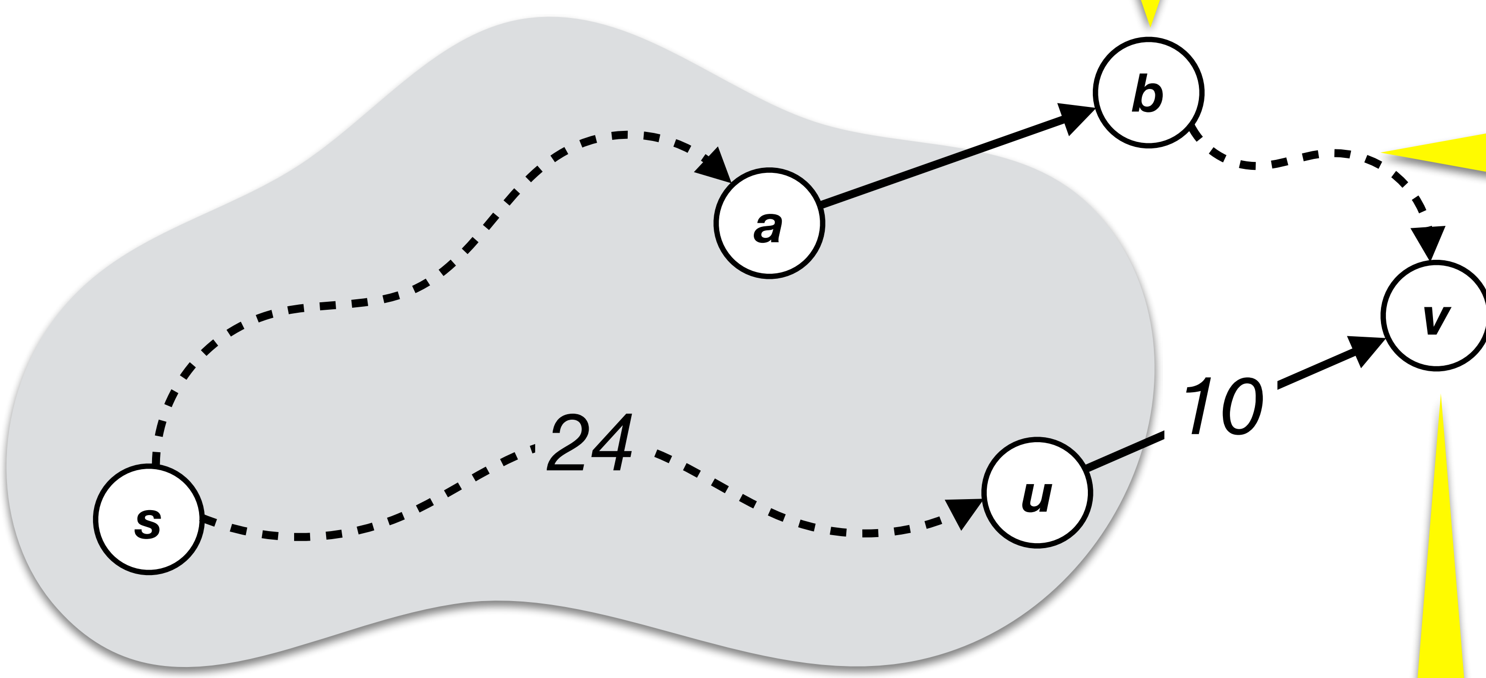
Since going through u is the shortest T-path from s to t, any alternative path from s to v must go through some vertex $b \notin T$, and T-dist(b) $\geq$ T-dist(v).



By our choice of v, for all $x \notin T$, T-dist(v) $\leq$ T-dist(x)

Correctness (cont'd)

Since going through u is the shortest T-path from s to t, any alternative path from s to v must go through some vertex $b \notin T$, and T-dist(b) ≥ T-dist(v).

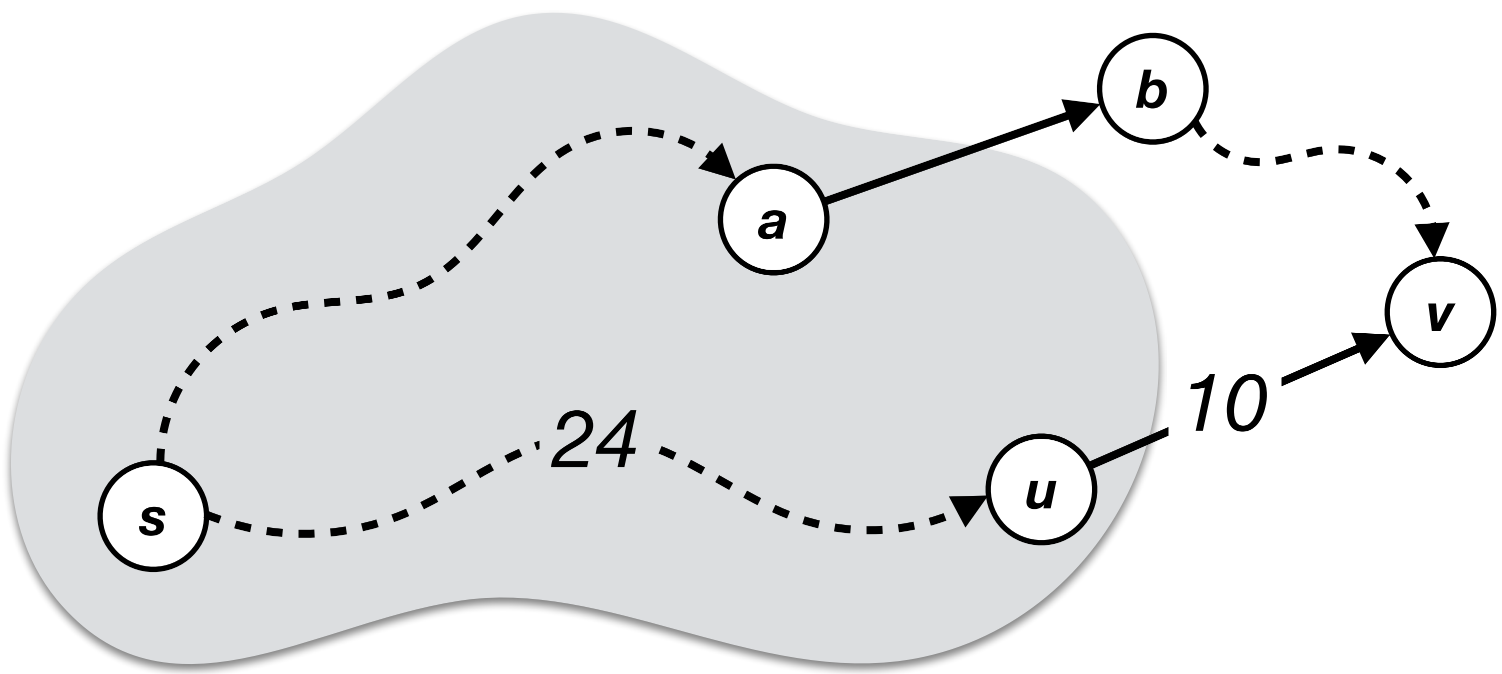


The length of this part from b to v is positive. (all edges have positive length.)

By our choice of v, for all $x \notin T$, T-dist(v) ≤ T-dist(x)

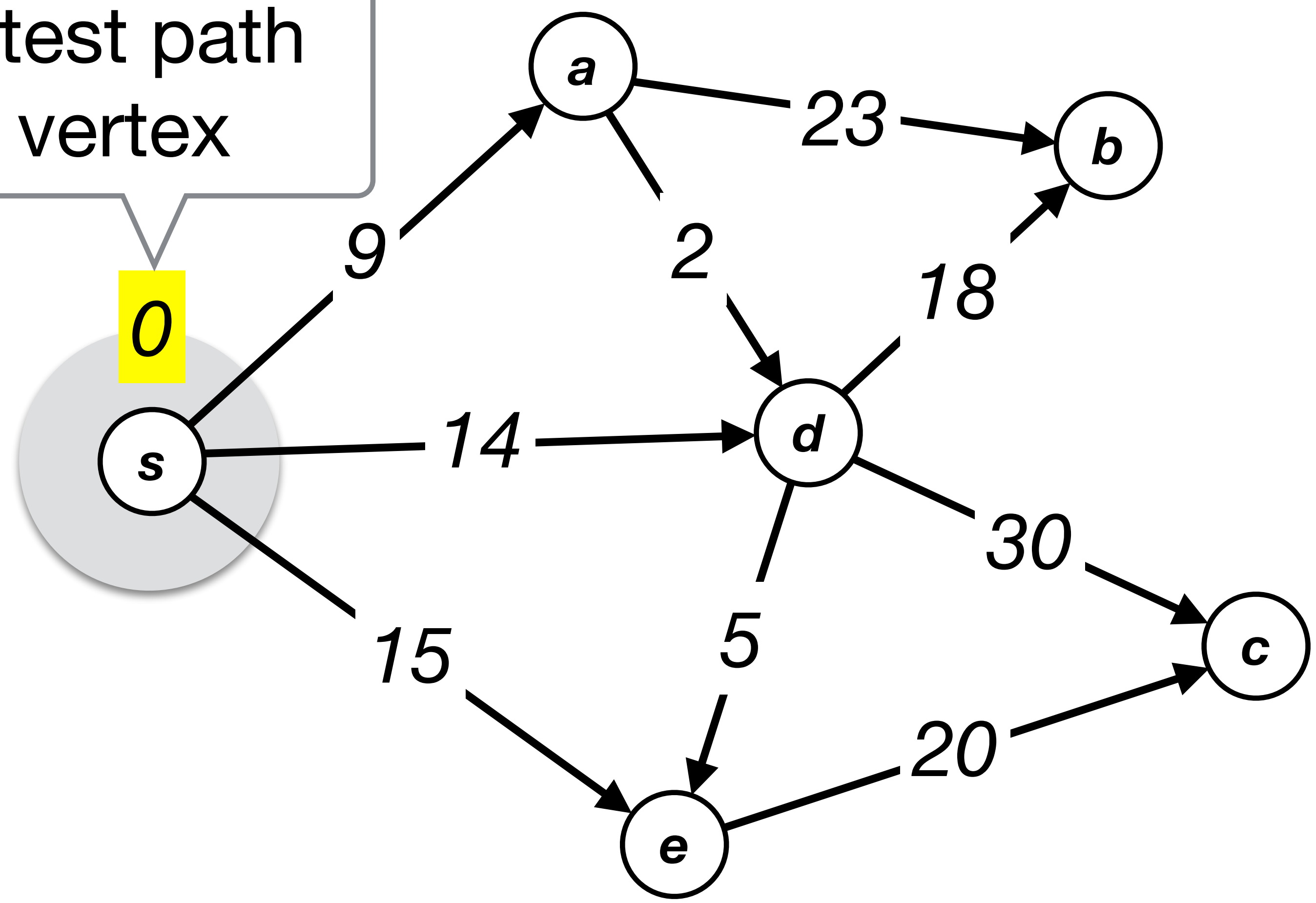
Correctness (cont'd)

Conclusion: The length of any path *P* from s to v is no smaller than *T*-dist(*v*) because length(*P*) ≥ *T*-dist(*b*) ≥ *T*-dist(v). Hence, the *T*-path from *s* to *v* through *u* is the shortest path.



Sample Run

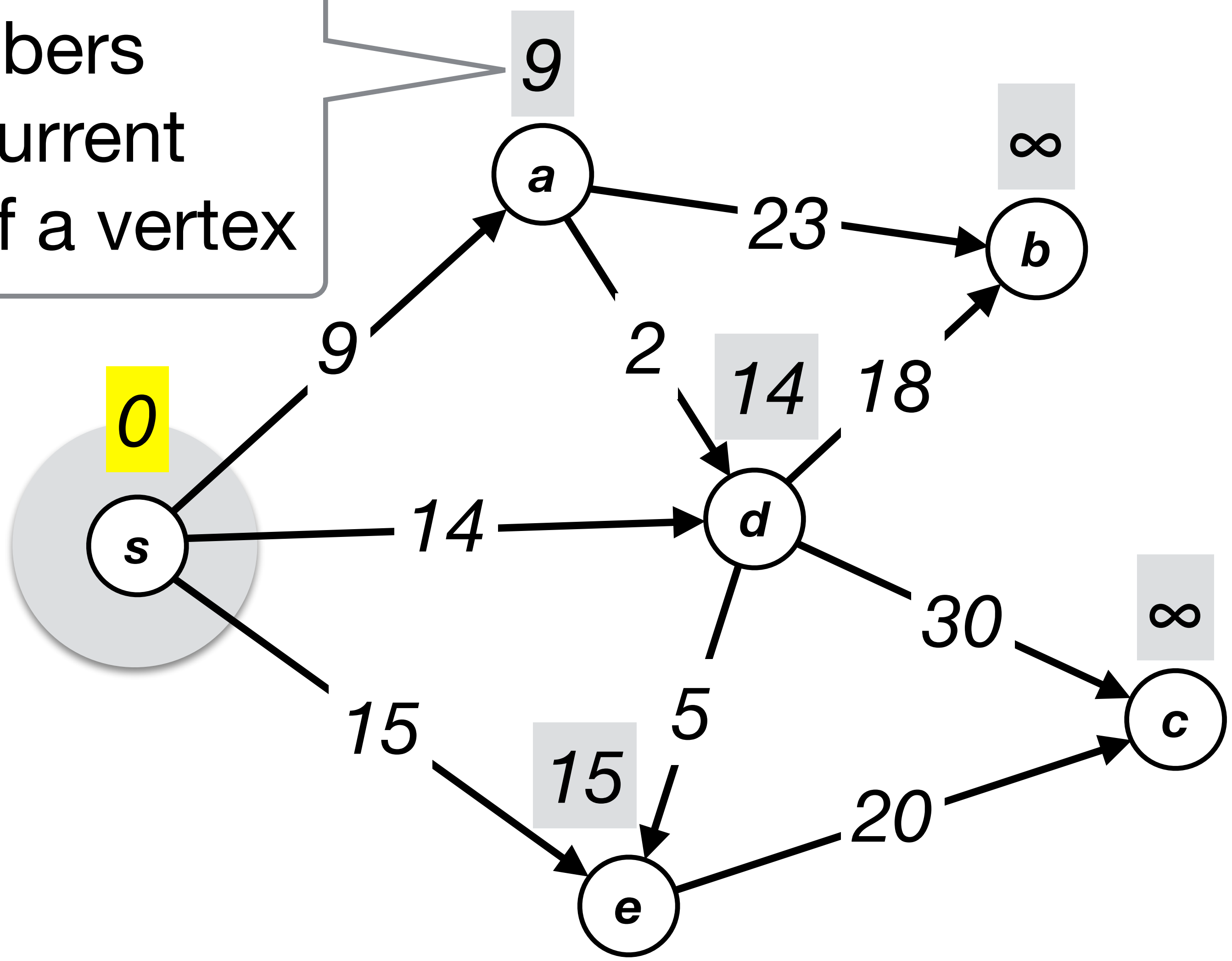
Highlighted numbers denote the length of of the shortest path from *s* to *a* vertex



$T = \{ s \}$

Sample Run

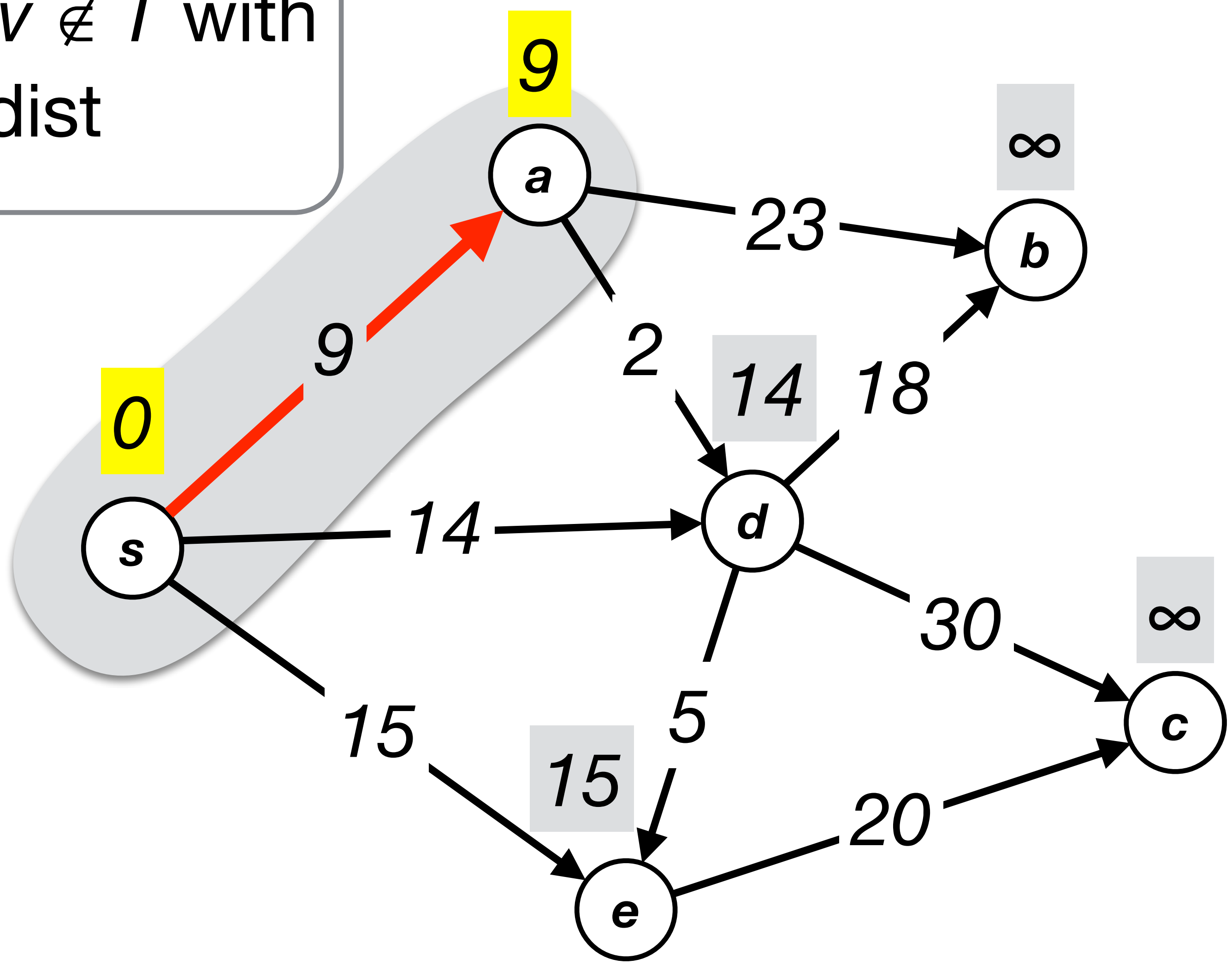
Shaded numbers denote the current *T*-distance of a vertex



$T = \{ s \}$

Sample Run

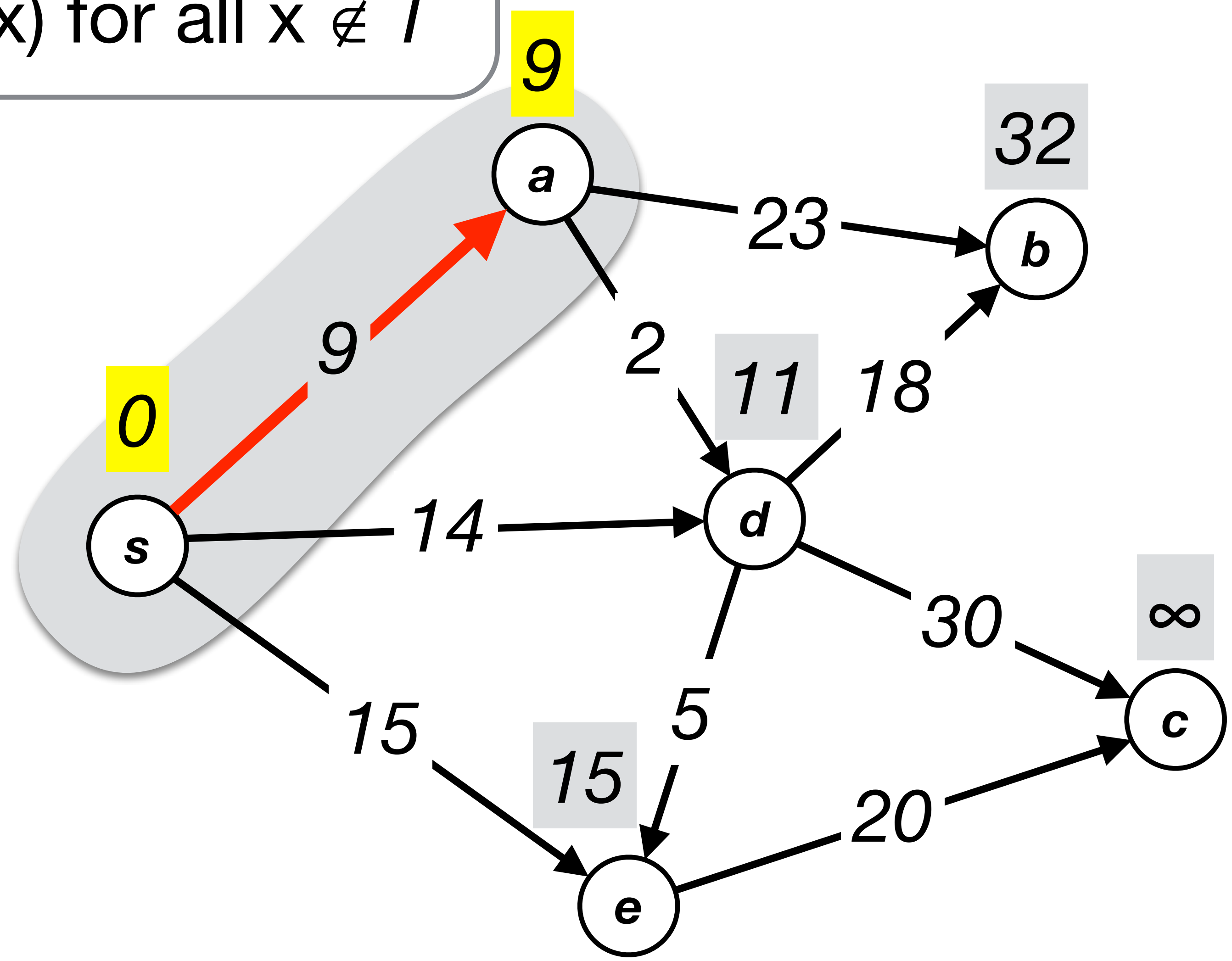
Add the vertex $v \notin T$ with the smallest T-dist



$T = \{ s, a \}$

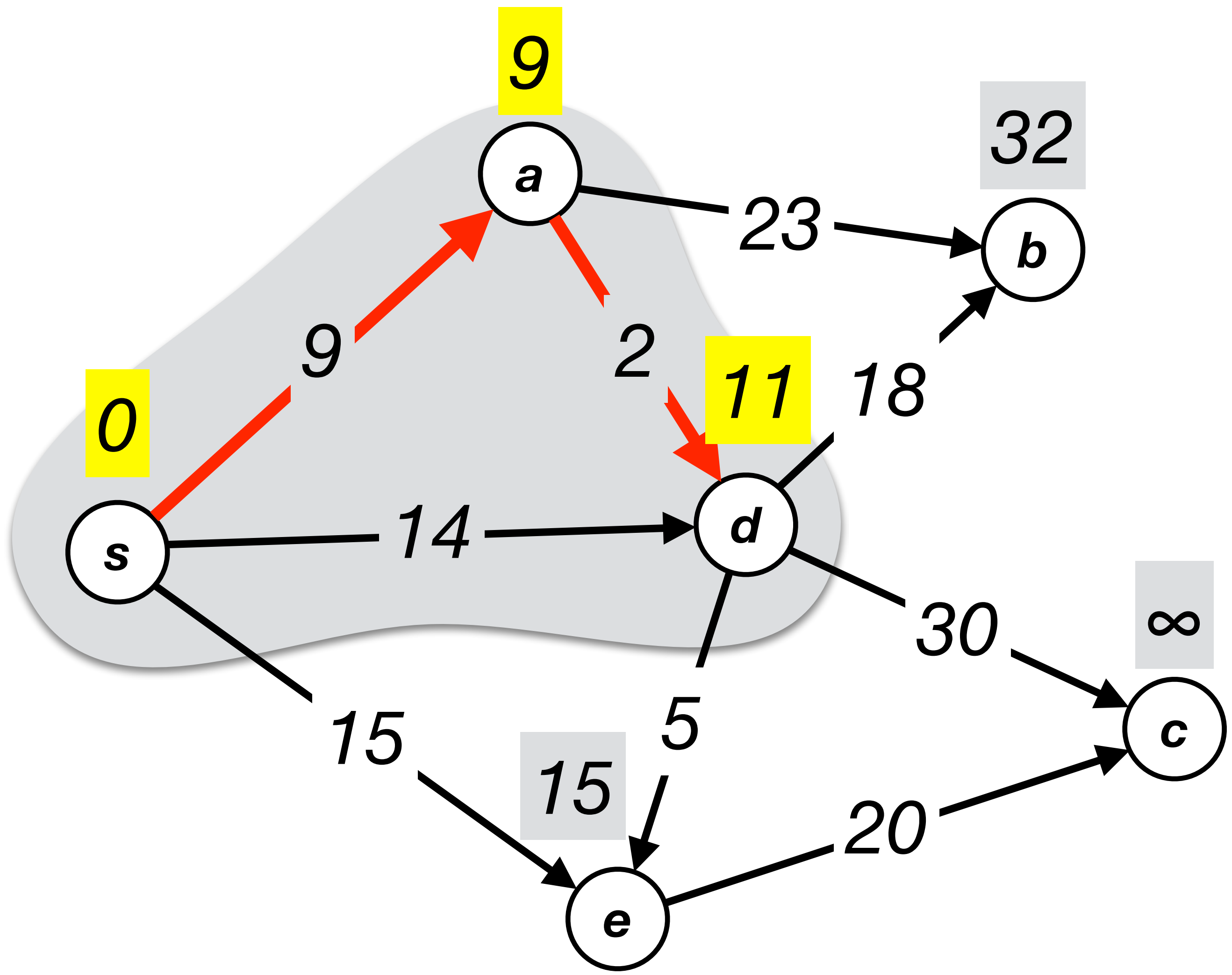
Sample Run

Update T-dist(x) for all x $\notin T$



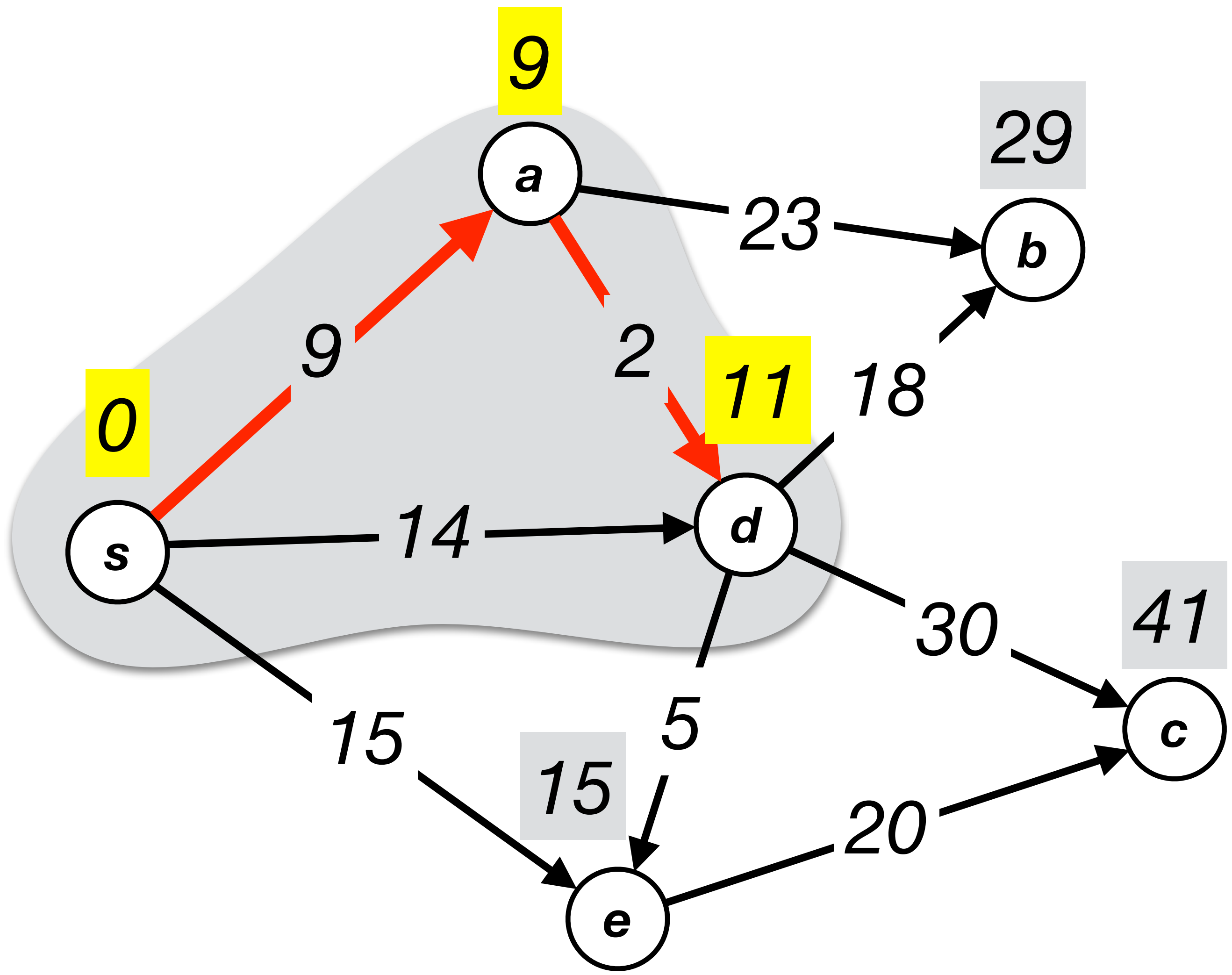
$T = \{ s, a \}$

Sample Run



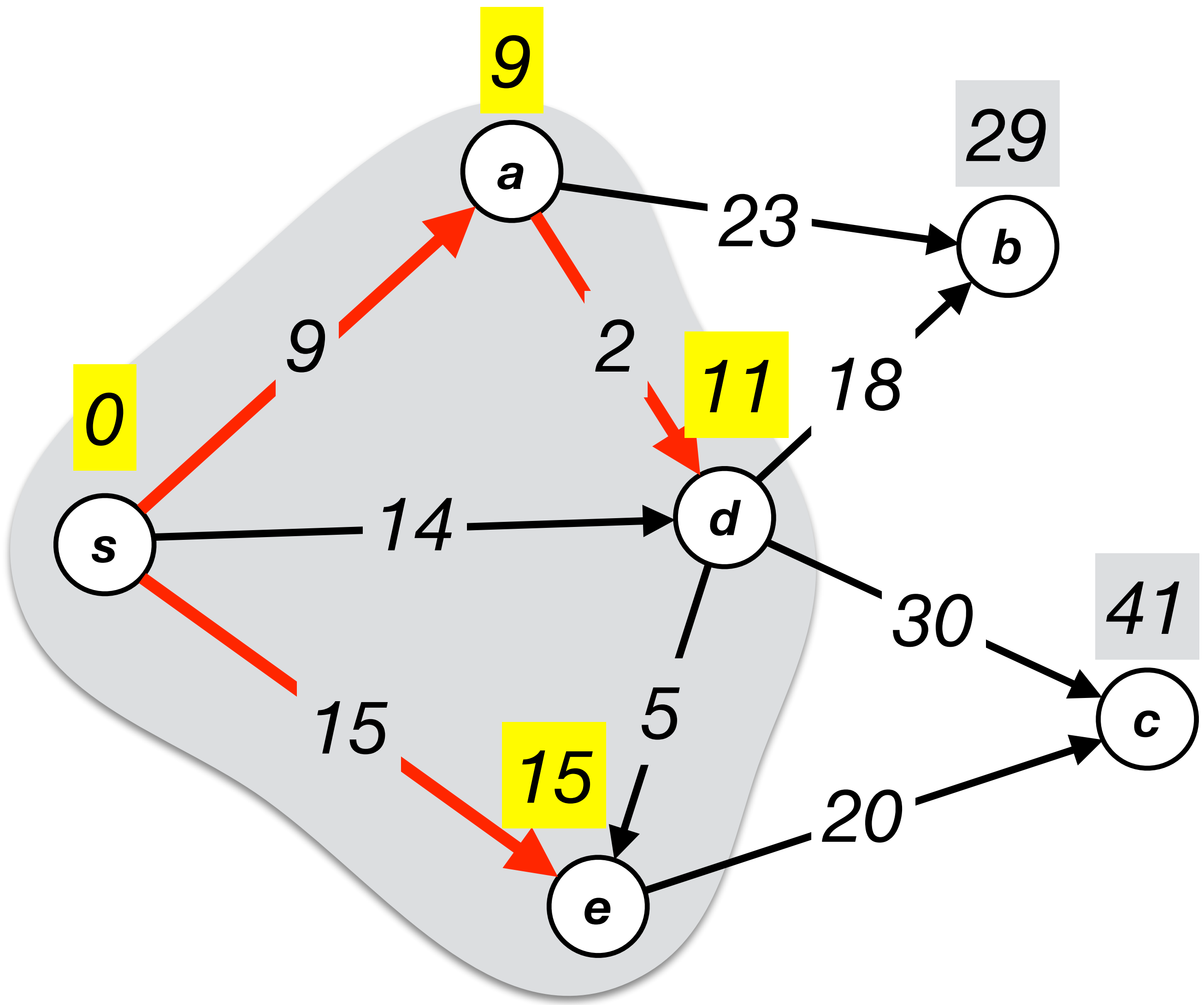
$T = \{\, s, a, d \,\}$

Sample Run



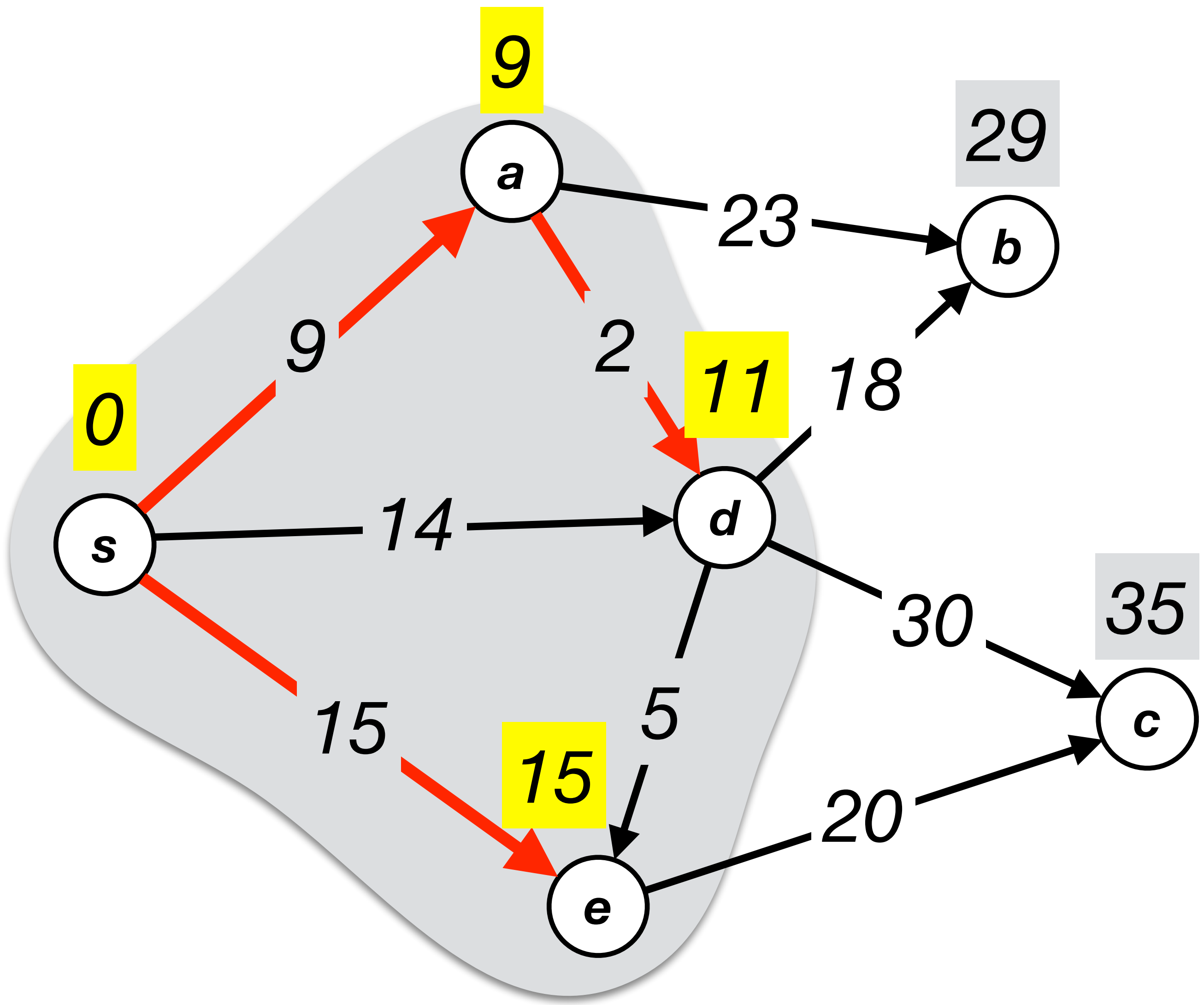
$T = \{ s, a, d \}$

Sample Run



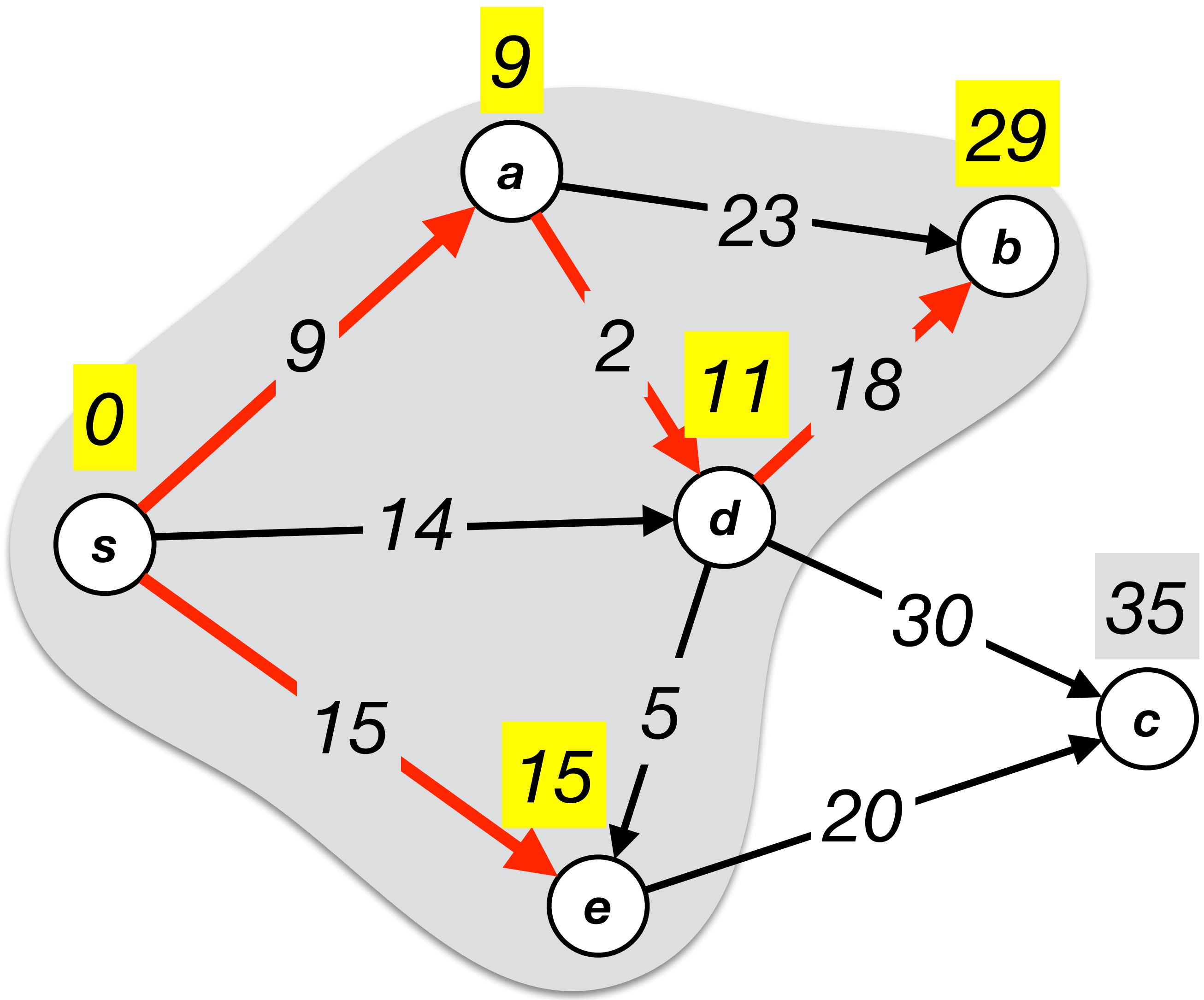
$T = \{ s, a, d, e \}$

Sample Run



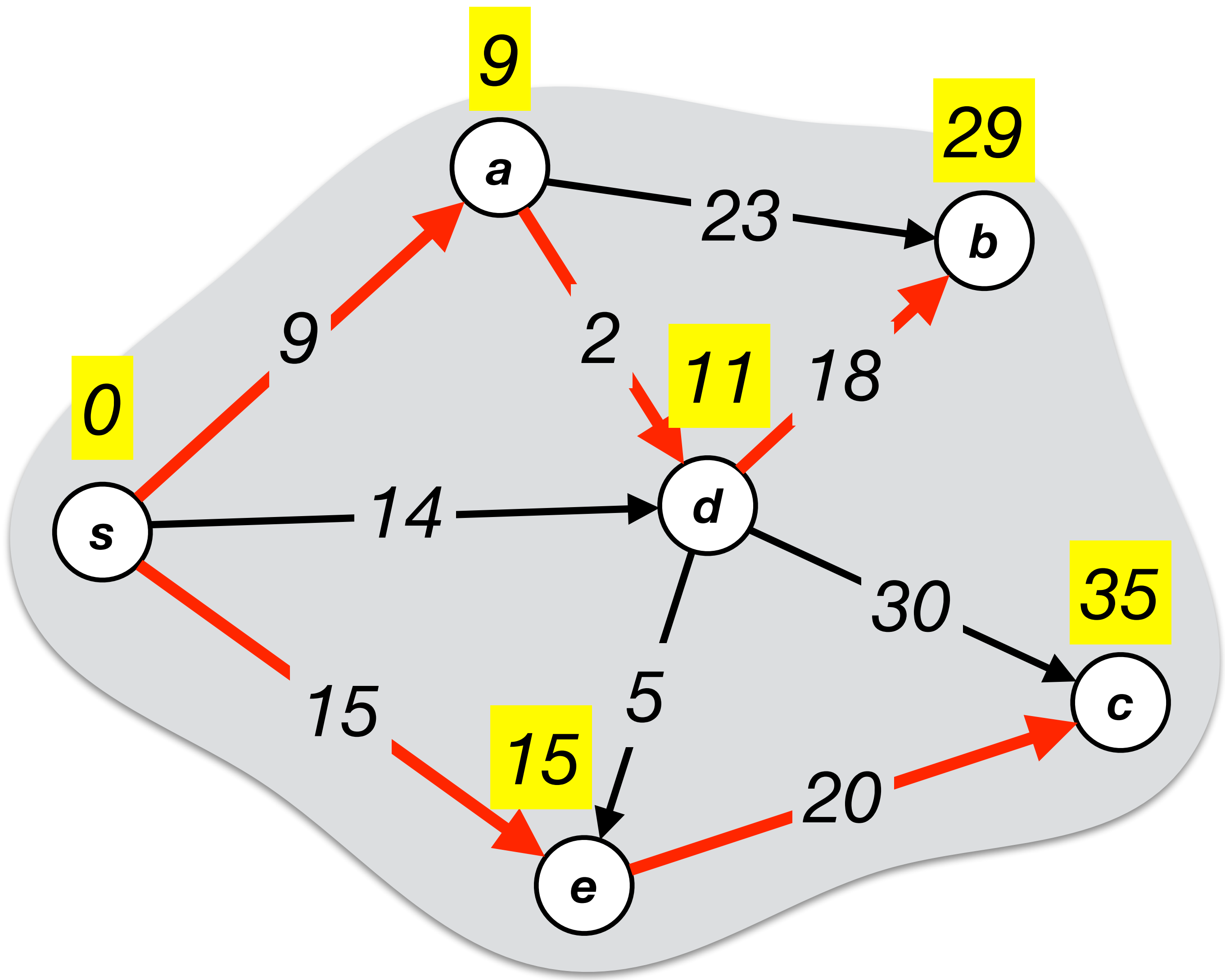
$T = \{ s, a, d, e \}$

Sample Run



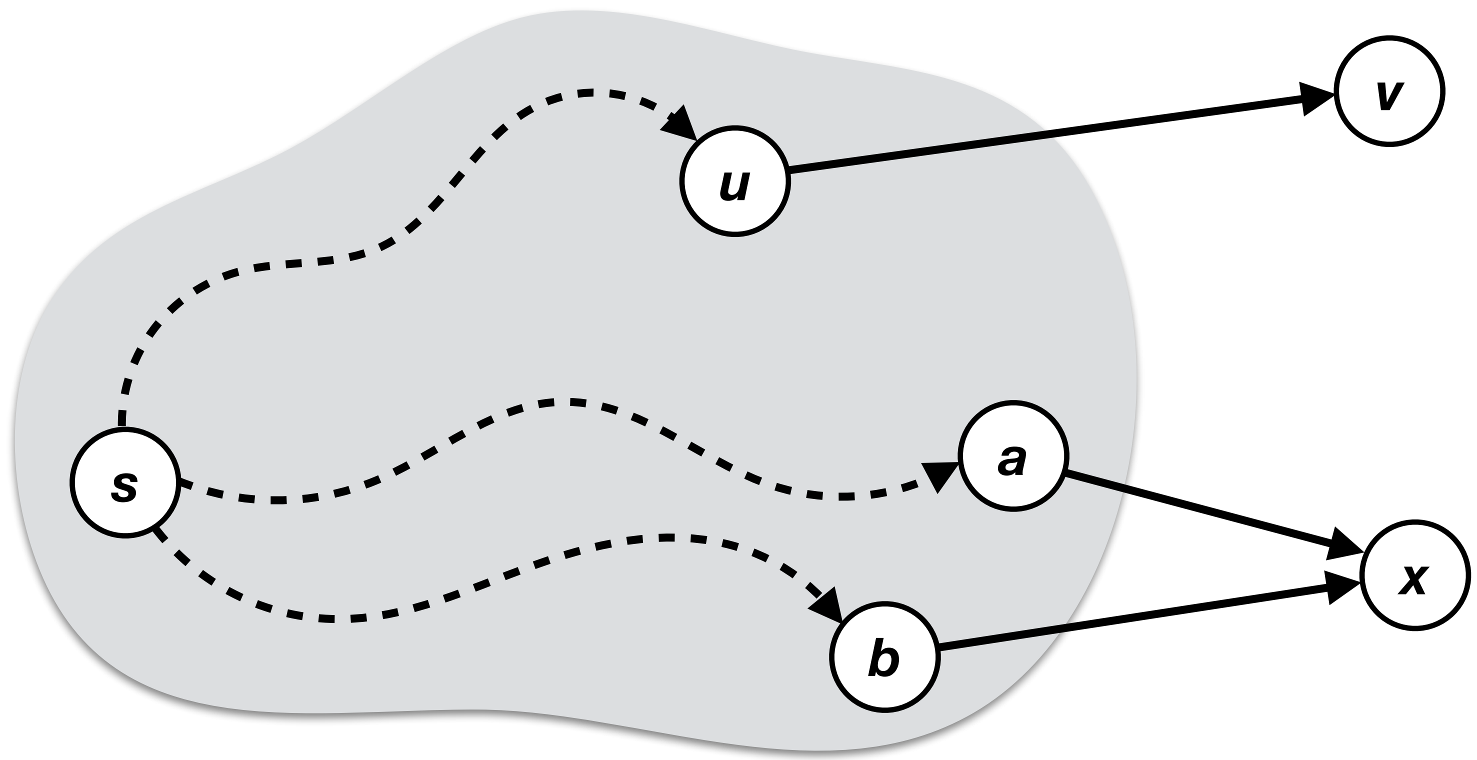
$T = \{ s, a, d, e, b \}$

Sample Run



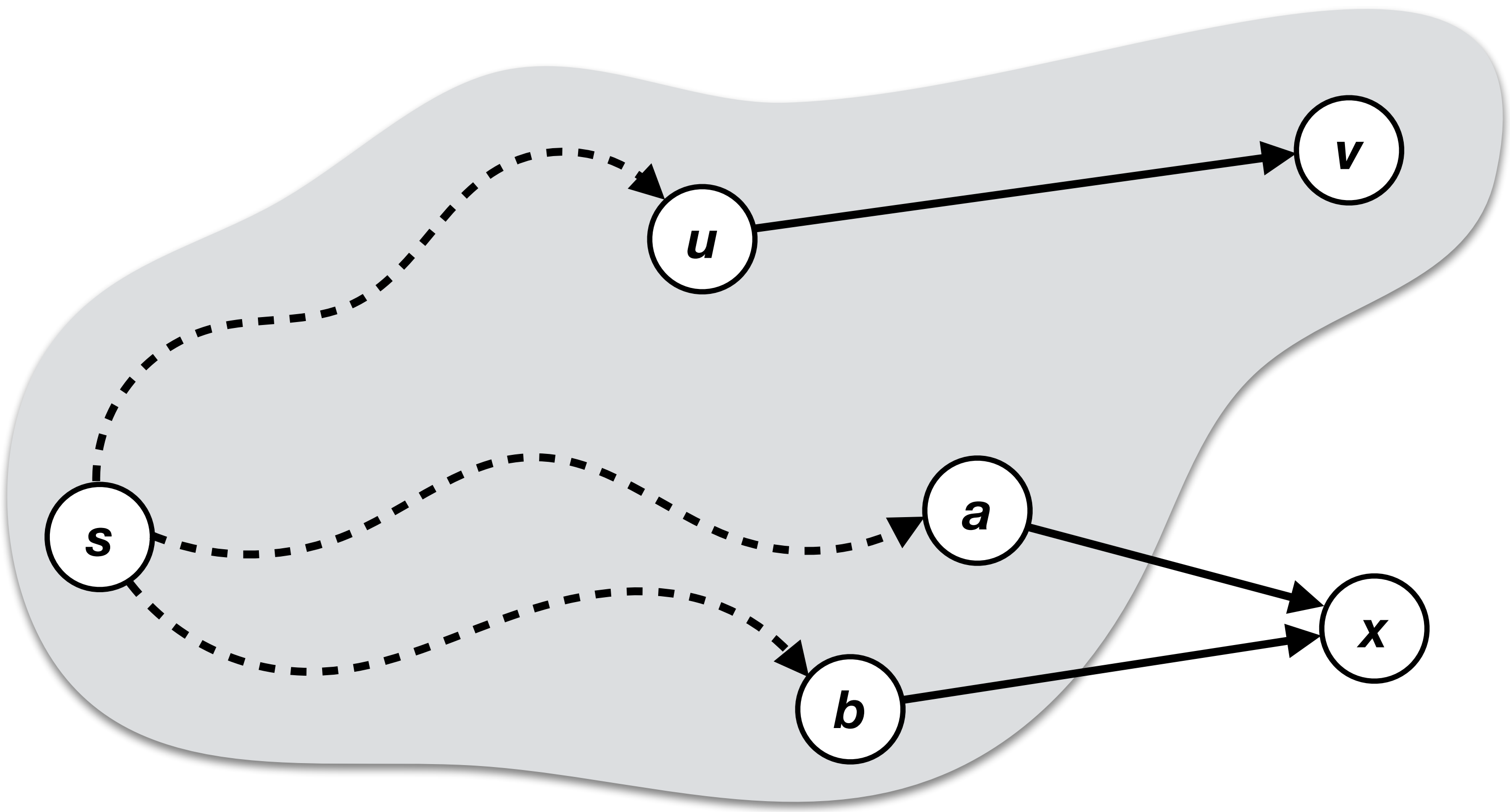
$T = \{ s, a, d, e, b, c \}$

How to update *T*-dist(*x*)?



- Recall T-dist(x) = min { T-dist(a) + $L(a, x)$, T-dist(b) + $L(b, x)$ }
- Suppose the algorithm adds a new vertex v to T.
  - Let T' denote the new tree.
  - Let T'-dist denote the updated tree-distance.

How to update T-dist(x)?

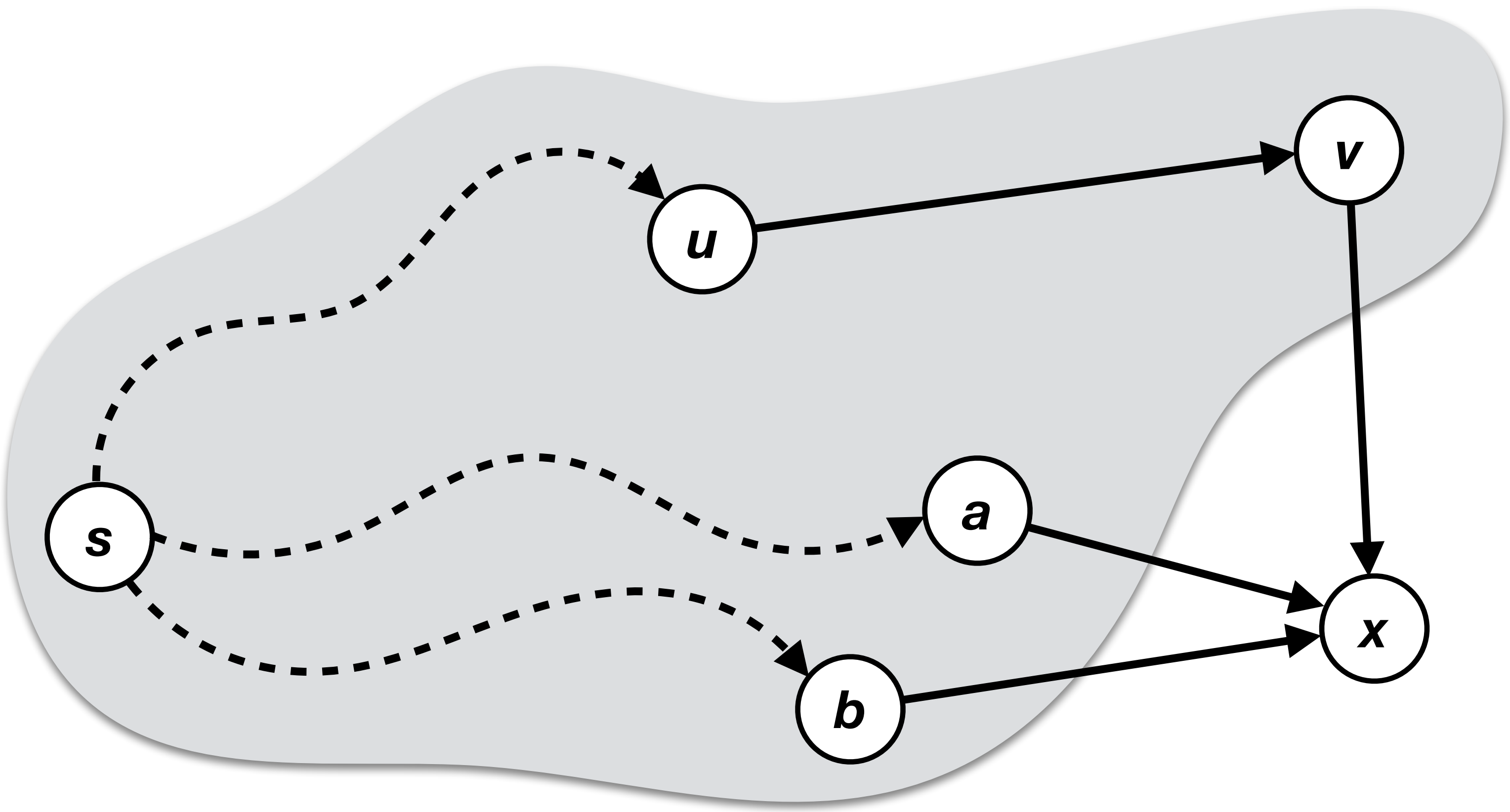


- If there is no edge (v, x), then we have

  T'-dist(x) = min { T'-dist(a) + $L(a, x)$, T'-dist(b) + $L(b, x)$ }

- Adding v to T does not change the tree-distance of a and b: T'-dist(a) = T-dist(a), T'-dist(b) = T-dist(b). Hence, no change.

How to update *T*-dist(*x*)?

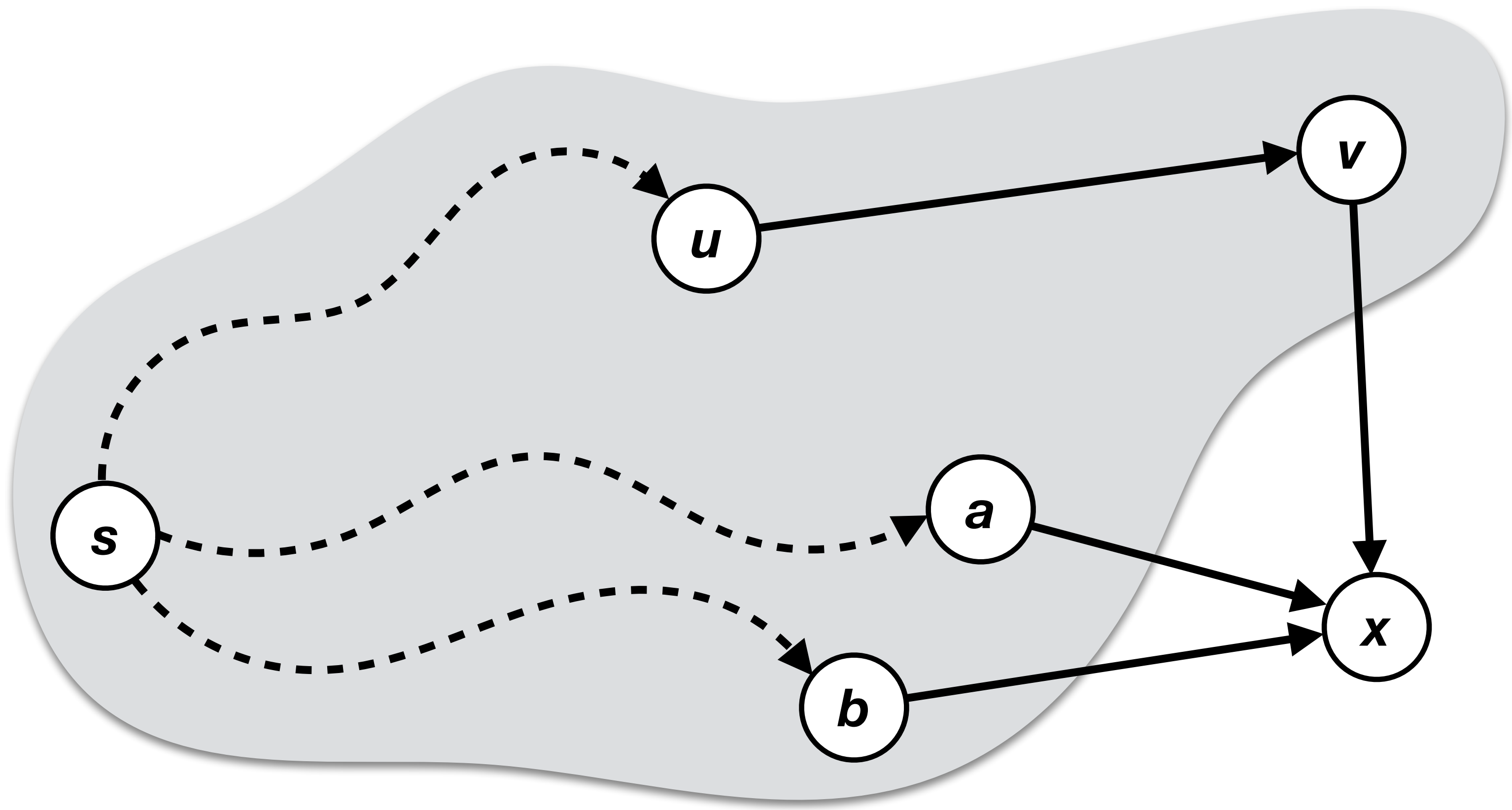


- If there is edge *(u, x)*, then *T'*-dist(x) is equal to

  min { *T'*-dist(*a*) + *L(a, x)* , *T'*-dist(*b*) + *L(b, x)* , *T'*-dist(*v*) + *L(v, x)* }
- Note that *T*-dist and *T'*-dist are the same for *v*, *a*, *b*. Thus,

  T'-dist(x) = min { *T*-dist(*x*) , *T*-dist(*v*) + *L(v, x)* }

How to update T-dist(x)?



Conclusion: After adding v to enlarge T, we only need to

- scan the adjacent list of v in order to update T-dist;
- for each vertex x in the adjacent list, update its tree-distance as the minimal of the old tree-distance and T-dist(v) + $L(v, x)$.

The Algorithm (DPV 4.1.1)

1) **initialize** dist(s) = 0, and dist(x) = ∞ for other vertices x;
2) $V' = \{\ \}$;
3) **while** $V' \neq V$ **do**
4) pick the node $v \notin V'$ with the smallest dist(.)
5) add v to V'
6) **for** all edges $(v, x) \in E$ **:**
7) **if** dist(x) > dist(v) + $L(v, x)$ **:** update dist(x) = dist(v) + $L(v, x)$.

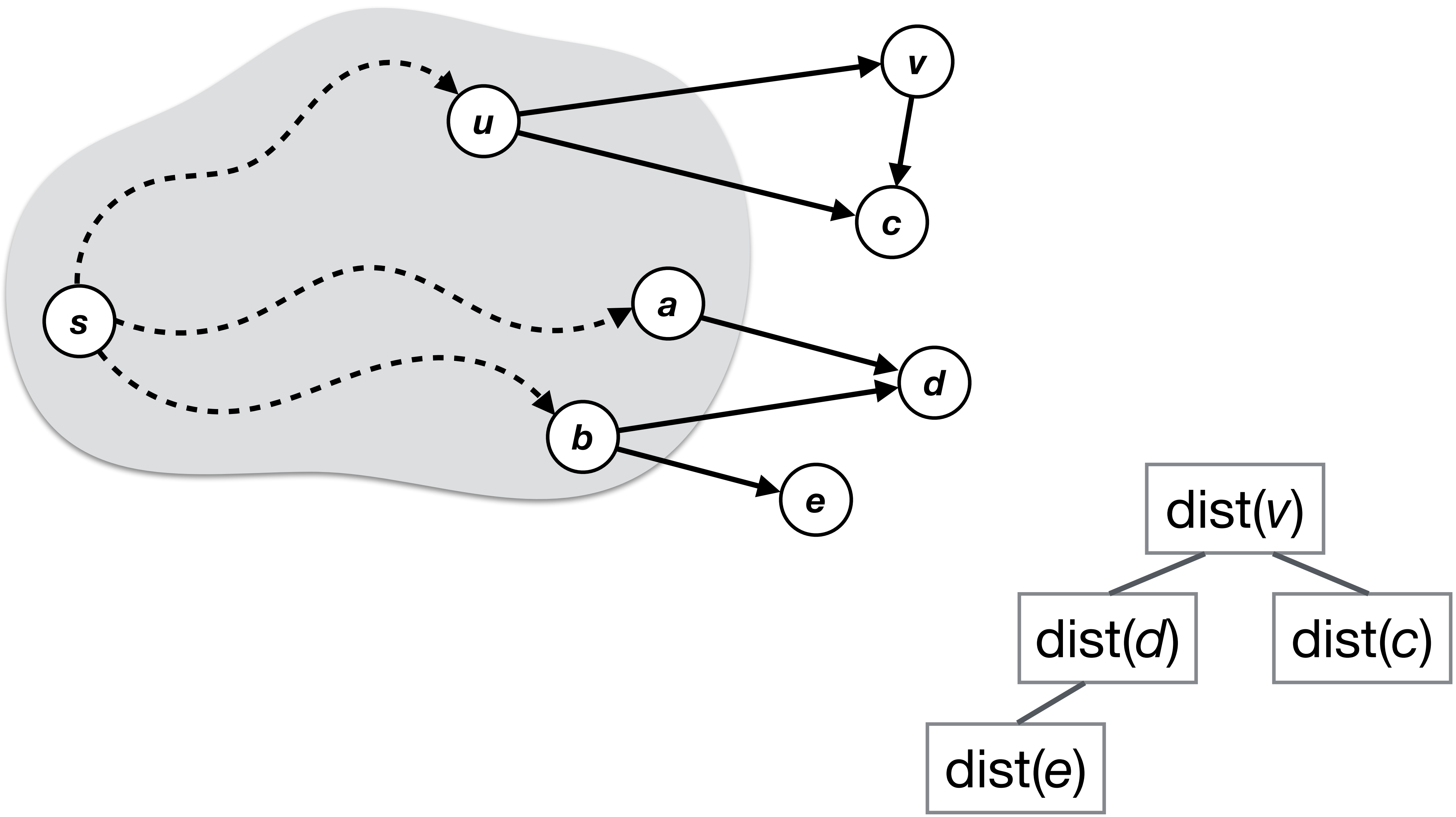
Running time:

- Steps 1, 2, 5 take $O(|V|)$ time. Steps 6, 7 takes $O(|E|)$ time.
- If we implement dist(.) as a simple array, step 4 takes $O(|V|)$ time to find the smallest dist(.), and since we execute step 4 $O(|V|)$ times, total time is $O(|V|^2)$.

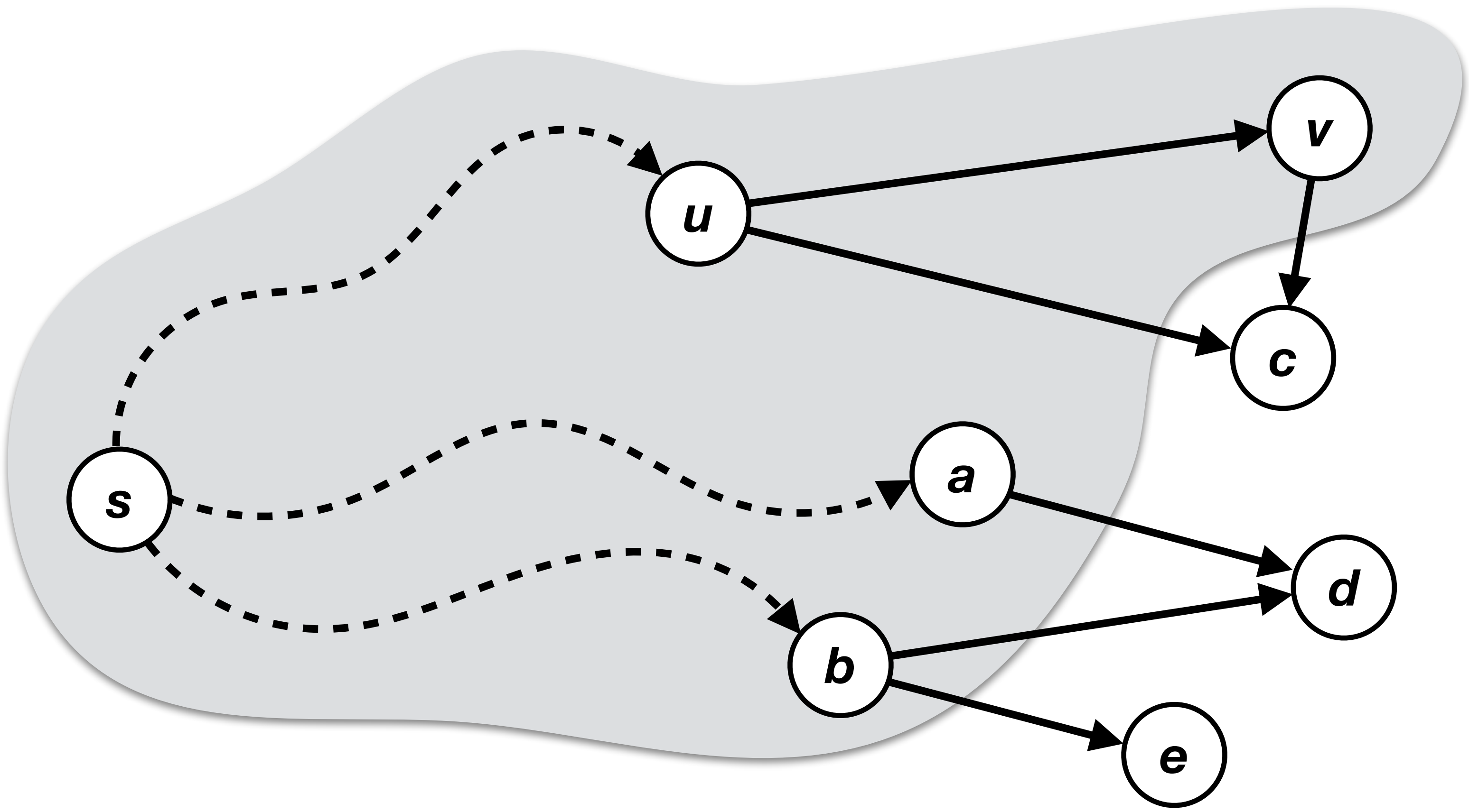
Priority Queue Implementation

- It is usually more efficient to store dist(.) in a priority queue, e.g., implemented through binary min heap.
- Recap of basic properties of binary min heap:
  - Building an empty heap (**build-heap**): *O(1)*;
  - Finding the minimal number (**find-min**): *O(1)*;
  - Deleting a the minimal number (**delete-min**): *O(log n)*.
  - Inserting a new number (**insert**): *O(log n)*;

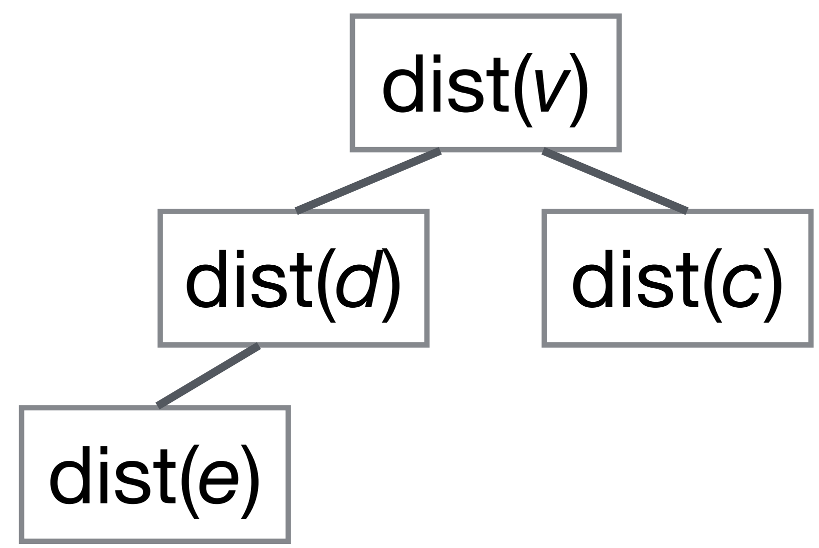
Priority Queue Implementation



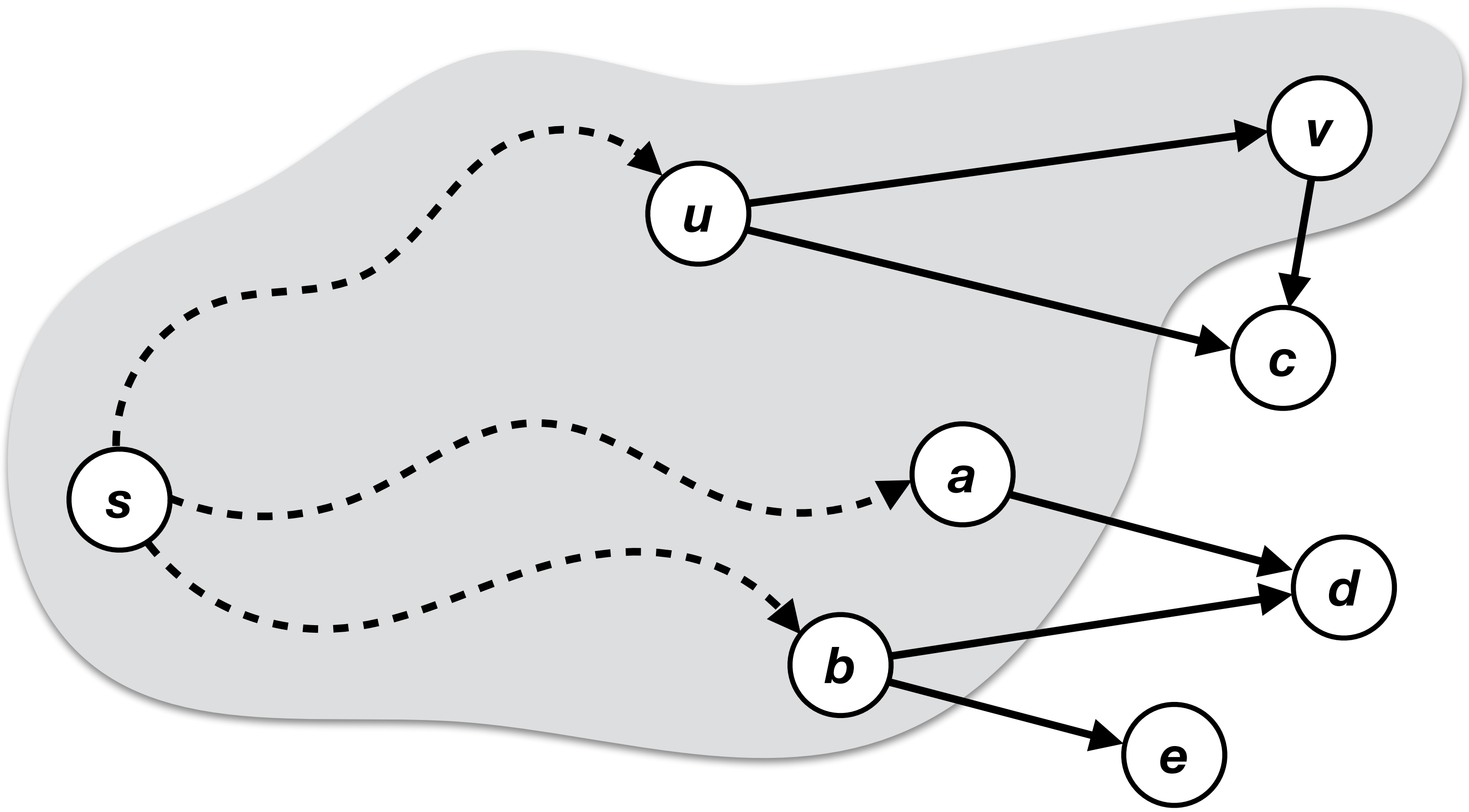
Priority Queue Implementation



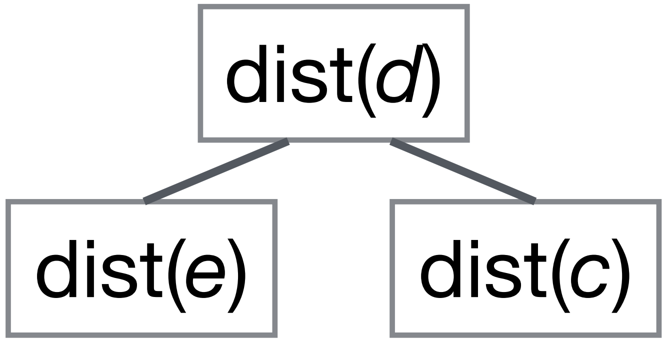
find-min



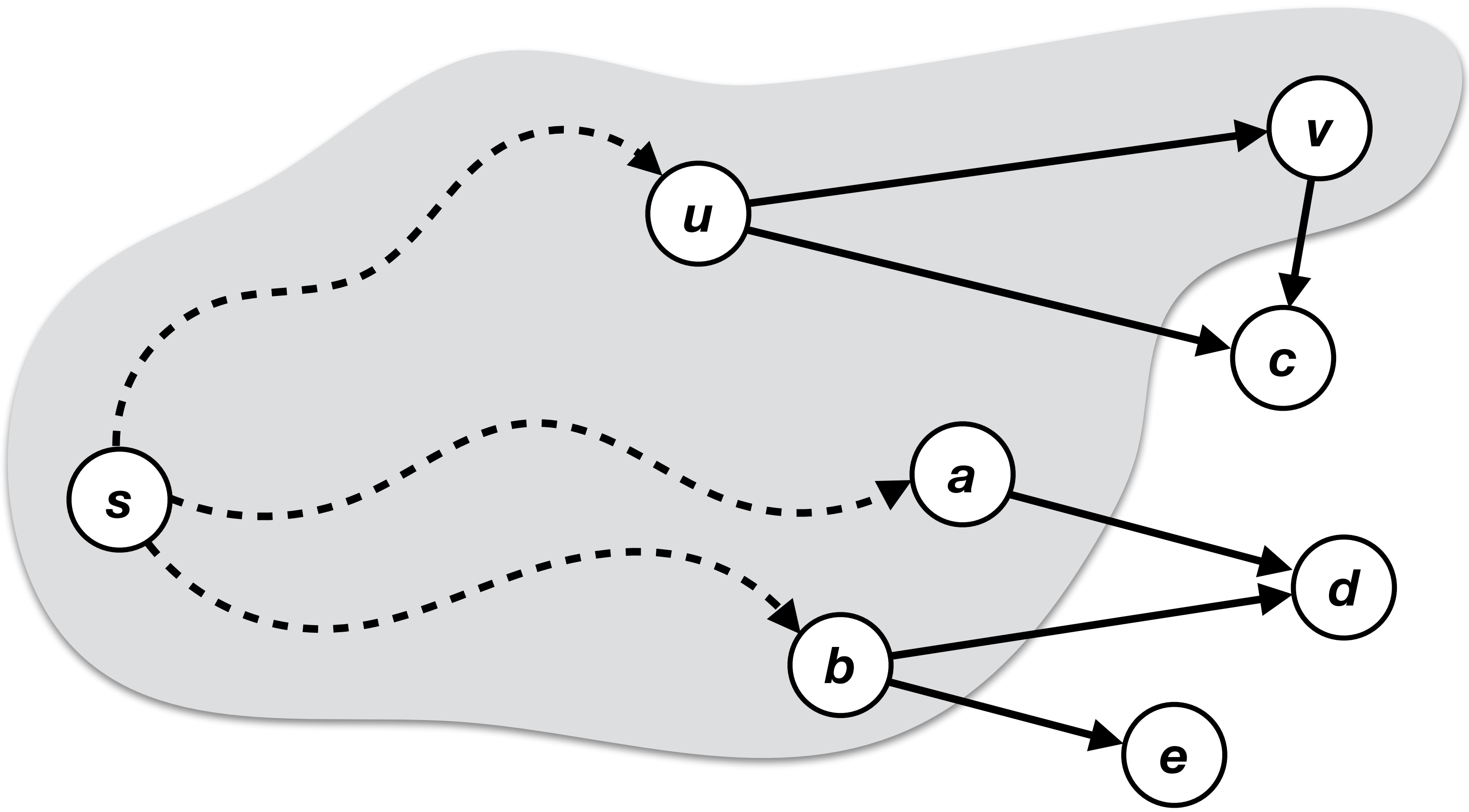
Priority Queue Implementation

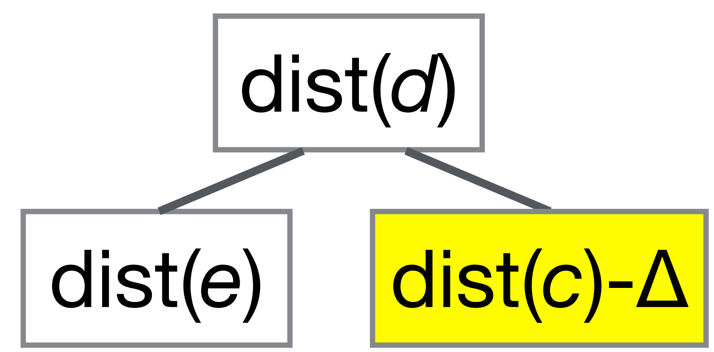


delete(v)



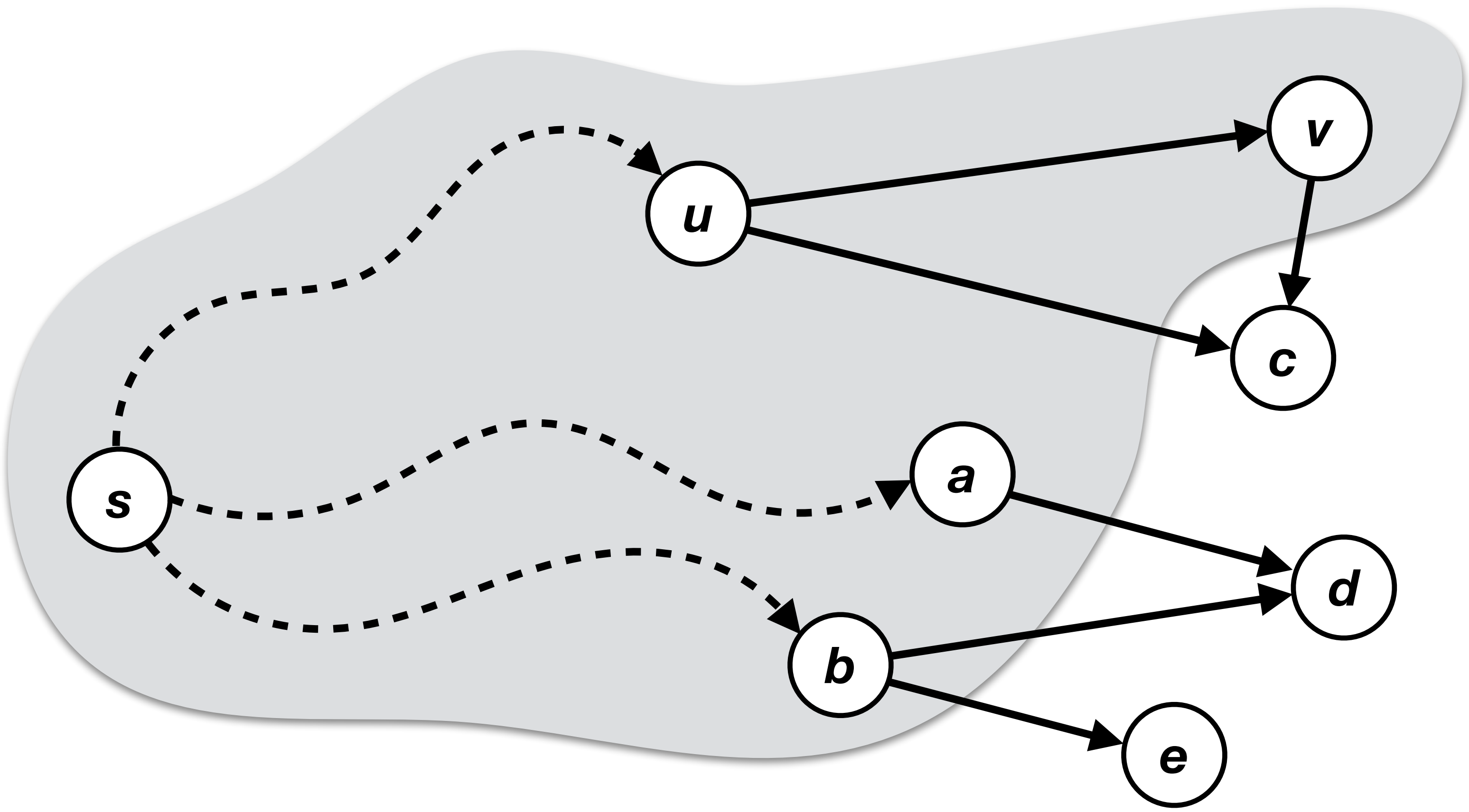
Priority Queue Implementation

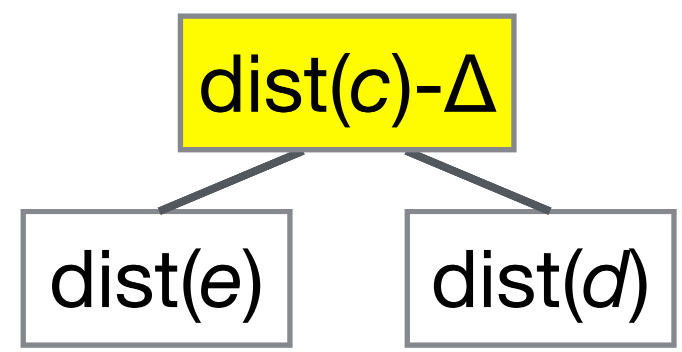




Update dist[x] for every x s.t. there is edge (v, x); for example, dist[c] may be decrease by some $\Delta > 0$.

Priority Queue Implementation



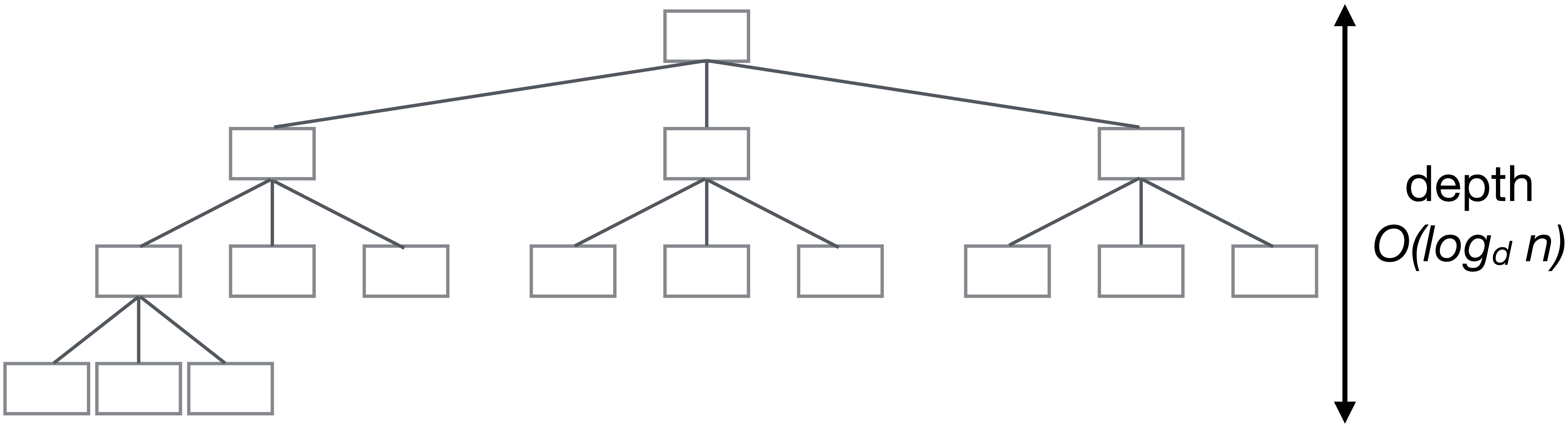


Remember, we may need to shift up the new value in order to restore heap order.

Running time for this implementation

- Initially, we set dist(s) = 0, and dist(u) = ∞ for all other vertices u. Then, we insert all n vertices to a heap: $O(|V| \log |V|)$ time.
- We need to delete the minimum value from the heap n times to add n vertices to the SPT: $O(|V| \log |V|)$ time.
- After adding each vertex to the SPT, we need to scan its adjacency list and for each vertex x in the list, we may need to update dist(x) and shift up to restore heap order.
  - The hight of the tree is $\log |V|$, so we need to shift up at most $\log |V|$ times per vertex in the adjacency list.
  - Total size of the adjacency lists of all vertices is $O(|E|)$.
  - In total, this takes $O(|E| \log |V|)$ time.
- In sum, the running time is $O((|E| + |V|) \log |V|)$.

Advance topic: *d*-heap implementation



	Heap	3-Heap	*d*-Heap
build-heap	$O(1)$	$O(1)$	$O(1)$
insert	$O(\log n)$	$O(\log_3 n)$	$O(\log_d n)$
find-min	$O(1)$	$O(1)$	$O(1)$
delete-min	$O(\log n)$	$O(3 \log_3 n)$	$O(d \log_d n)$

Advance topic: d-heap implementation

- Initially, we set dist(s) = 0, and dist(u) = ∞ for all other vertices u. Then, we insert all n vertices to a heap: $O(|V|\ log_d\ |V|)$ time.
- We need to delete the minimum value from the heap n times to add n vertices to the SPT: $O(|V|\ d\ log_d\ |V|)$ time.
- After adding each vertex to the SPT, we need to scan its adjacency list and for each vertex x in the list, we may need to update dist(x) and shift up to restore heap order.
  - The hight of the tree is $log_d\ |V|$, so we need to shift up at most $log\ |V|$ times per vertex in the adjacency list.
  - Total size of the adjacency lists of all vertices is $O(|E|)$.
  - In total, this takes $O(|E|\ log_d\ |V|)$ time.
- Total $O(\ (|E| + d\ |V|)\ log_d\ |V|)$ time.
- Choosing $d = |E|/|V|$, the running time is $O(\ |E|\ log_{|E|/|V|}\ |V|\ )$.