

CSE 421

Introduction to Algorithms

Lecture 6: More Greedy Algorithms

Last time: Greedy Algorithms

Hard to define exactly but can give general properties

- Solution is built in small steps
- Decisions on how to build the solution are made to **maximize some criterion without looking to the future**
 - Want the 'best' current partial solution as if the current step were the last step

no undoing part decisions!

May be more than one greedy algorithm using different criteria to solve a given problem

- Not obvious which criteria will actually work

Greedy Analysis Strategies

Greedy algorithm stays ahead: Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's

- Example: Interval Scheduling analysis

Structural: Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

- Example: Interval Partitioning analysis

Exchange argument: Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

Scheduling to Minimize Lateness

Maximum

Scheduling to minimize lateness:

- Single resource as in interval scheduling but, instead of start and finish times, request i has
 - Time requirement t_i which must be scheduled in a contiguous block
 - Target deadline d_i by which time the request would like to be finished
- Overall start time s for all jobs

Schedule all requests

Requests are scheduled by the algorithm into time intervals $[s_i, f_i]$ s.t. $t_i = f_i - s_i$

- Lateness of schedule for request i is
 - If $f_i > d_i$ then request i is late by $L_i = f_i - d_i$; otherwise its lateness $L_i = 0$
- Maximum lateness $L = \max_i L_i$

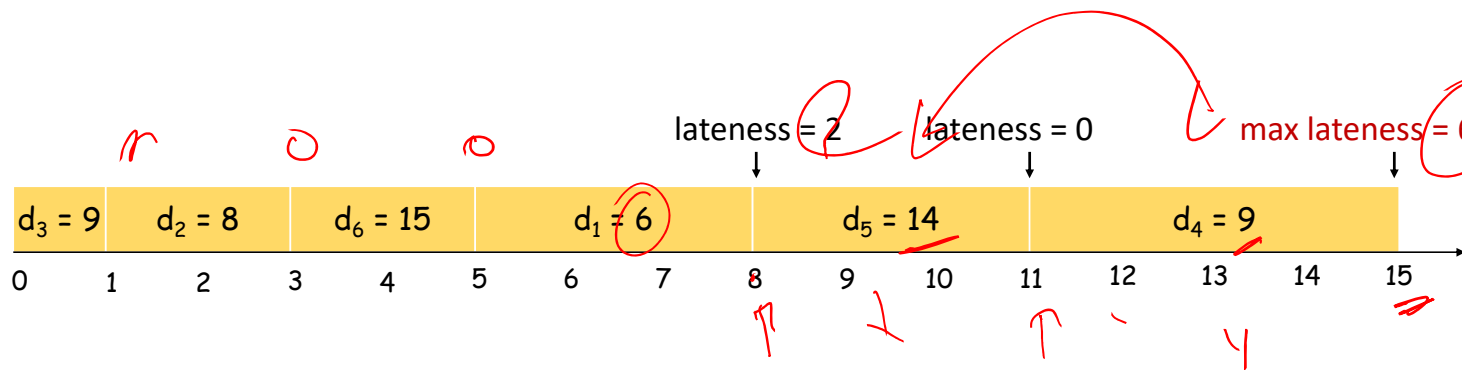
Goal: Find a schedule for all requests (values of s_i and f_i for each request i) to minimize the maximum lateness, L .

Scheduling to Minimizing Lateness

- Example:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15

time of death



*After swap
lateness 1*

*After swap
lateness 3*

Minimizing Lateness: Greedy Algorithms

Greedy template: Consider jobs in some order.

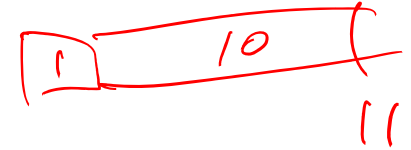
[Shortest processing time first] Consider jobs in ascending order of processing time t_j .

[Earliest deadline first] Consider jobs in ascending order of deadline d_j .

[Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

Minimizing Lateness: Greedy Algorithms

Greedy template: Consider jobs in some order.



[Shortest processing time first] Consider jobs in ascending order of processing time t_j .

	1	2
t_j	1	10
d_j	100	10

counterexample

Will schedule 1 (length 1) before 2 (length 10).
 2 can only be scheduled at time 1
 1 will finish at time 11 > 10. Lateness 1.
 Lateness 0 possible if 1 goes last.

[Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

	1	2
t_j	1	10
d_j	2	10

counterexample

Will schedule 2 (slack 0) before 1 (slack 1).
 1 can only be scheduled at time 10
 1 will finish at time 11 > 10. Lateness 9.
 Lateness 1 possible if 1 goes first.

Minimizing Lateness: Greedy Algorithms

Greedy template: Consider jobs in some order.

[Earliest deadline first] Consider jobs in ascending order of deadline d_j .

Greedy Algorithm: Earliest Deadline First

Consider requests in increasing order of deadlines

Schedule the request with the earliest deadline as soon as the resource is available

Minimizing Lateness: Greedy EDF Algorithm

- Greedy Earliest Deadline First (EDF).

end of last scheduled job

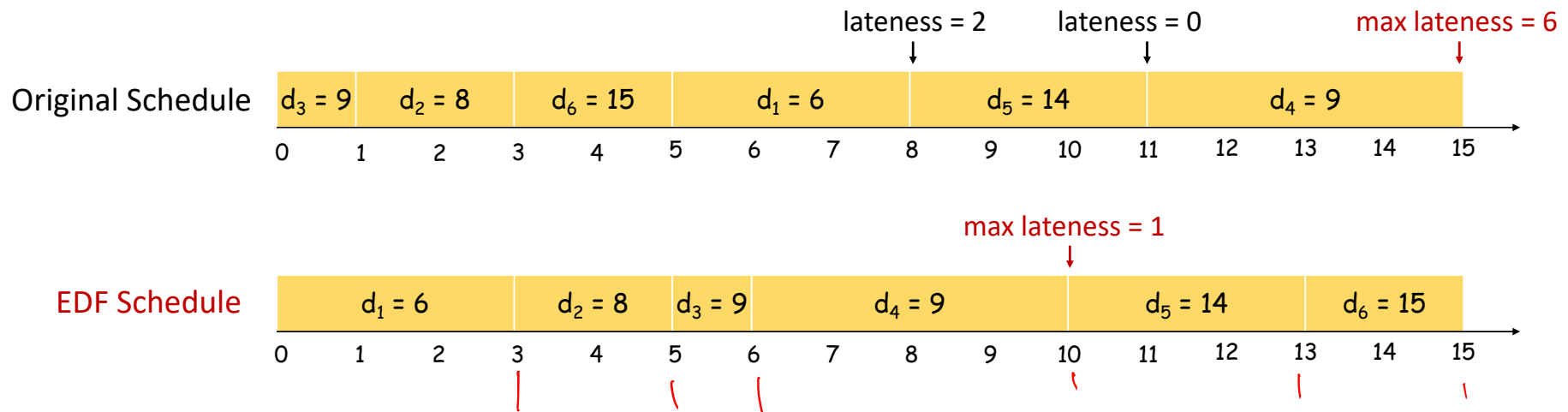
```
Sort deadlines in increasing order ( $d_1 \leq d_2 \leq \dots \leq d_n$ )  
 $f \leftarrow s$   
for  $i \leftarrow 1$  to  $n$  {  
     $s_i \leftarrow f$   
     $f_i \leftarrow s_i + t_i$   
     $f \leftarrow f_i$   
}
```

~~More on Monday!~~

Scheduling to Minimizing Lateness

- Example:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Proof for Greedy EDF Algorithm: Exchange Argument

Show that if there is another schedule O (think optimal schedule) then we can gradually change O so that...

- at each step the maximum lateness in O never gets worse
- it eventually becomes the same cost as A

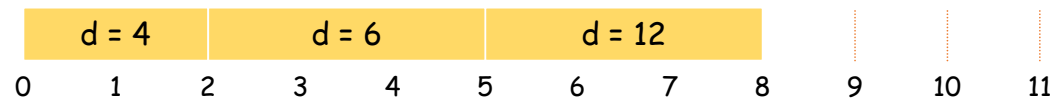
This means that A is at least as good as O , so A is also optimal!

Minimizing Lateness: No Idle Time

Observation: There exists an optimal schedule with no **idle time**



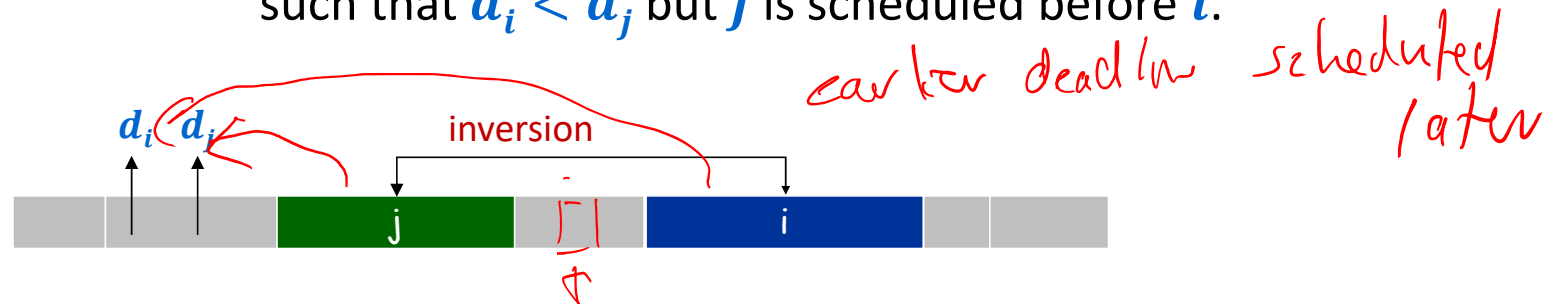
At least as good



Observation: The greedy EDF schedule has no idle time.

Minimizing Lateness: Inversions

Defn: An **inversion** in schedule S is a pair of jobs i and j such that $d_i < d_j$ but j is scheduled before i .



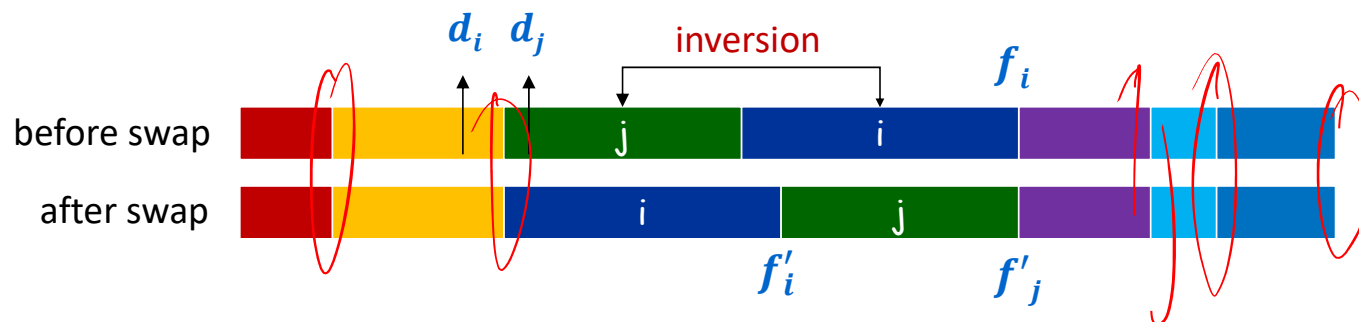
Observation: Greedy EDF schedule has no inversions.

Observation: If schedule S (with no idle time) has an inversion it has two adjacent jobs that are inverted

- Any job in between would be inverted w.r.t. one of the two ends

Minimizing Lateness: Inversions

Defn: An **inversion** in schedule S is a pair of jobs i and j such that $d_i < d_j$ but j is scheduled before i .

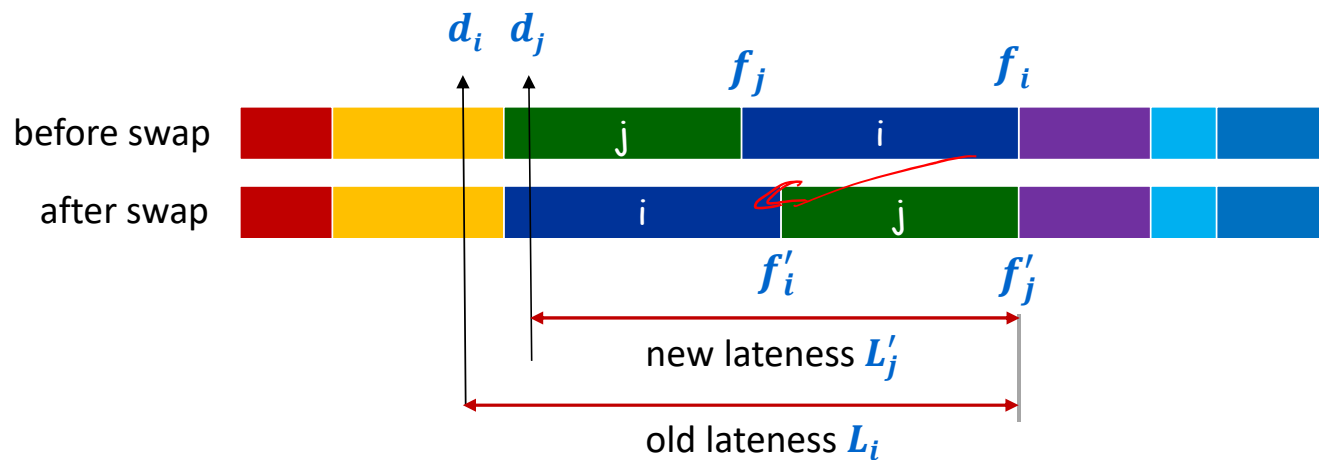


Claim: Swapping two adjacent, inverted jobs

- reduces the # of inversions by **1** ✓
- does not increase the max lateness.

Minimizing Lateness: Inversions

Defn: An **inversion** in schedule S is a pair of jobs i and j such that $d_i < d_j$ but j is scheduled before i .



Claim: Maximum lateness does not increase

■

Optimal schedules and inversions

Claim: There is an optimal schedule with no idle time and no inversions

Proof:

By previous argument there is an optimal schedule \mathcal{O} with no idle time

If \mathcal{O} has an inversion then it has an **adjacent** pair of requests in its schedule that are inverted and can be swapped without increasing lateness

... we just need to show one more claim that eventually this swapping stops

Optimal schedules and inversions

Claim: Eventually these swaps will produce an optimal schedule with no inversions.

Proof:

Each swap decreases the # of inversions by **1**

There are a bounded # of inversions possible in the worst case

- at most $n(n - 1)/2$ but we only care that this is finite.

The # of inversions can't be negative so this must stop. ■

Idleness and Inversions are the only issue

Claim: All schedules with no inversions and no idle time have the same maximum lateness.

Proof:

Schedules can differ only in how they order requests with equal deadlines

Consider all requests having some common deadline d .

- Maximum lateness of these jobs is based only on finish time of the last one ... and the set of these requests occupies the same time segment in both schedules.

⇒ The last of these requests finishes at the same time in any such schedule. ■

Earliest Deadline First is optimal

We know that

- There is an optimal schedule with no idle time or inversions
- All schedules with no idle time or inversions have the same maximum lateness
- EDF produces a schedule with no idle time or inversions

So ...

- EDF produces an optimal schedule ■

Greedy Analysis Strategies

Greedy algorithm stays ahead: Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's

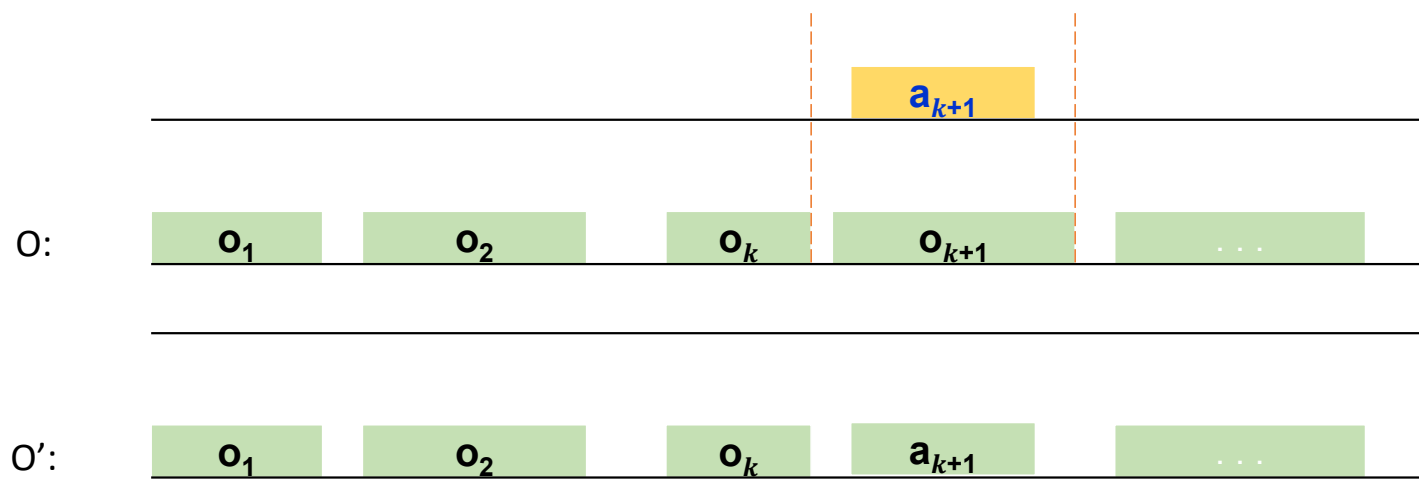
Structural: Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

Exchange argument: Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

Interval Scheduling: Exchange Argument

Theorem: Greedy (by-finish-time) algorithm produces an optimal solution

Proof: Consider the first request (in the sequence order) chosen by that other algorithm O that doesn't include a compatible request with the earliest finish time. Replace that request by the one with the earliest finish time.



Repeat at most n times to get greedy schedule. # of requests is at least as good. ■

Single-source shortest paths

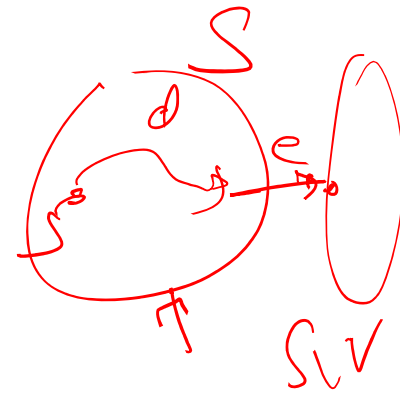
Given: an (un)directed graph $G = (V, E)$ with each edge e having a non-negative weight $w(e)$ and a vertex s

Find: (length of) shortest paths from s to each vertex in G

A Greedy Algorithm

Dijkstra's Algorithm:

- Maintain a set S of vertices whose shortest paths are known
 - initially $S = \{s\}$
- Maintaining current best lengths of paths that *only go through* S to each of the vertices in G
 - path-lengths to elements of S will be right, to $V \setminus S$ they might not be right
- Repeatedly add vertex v to S that has the shortest path-length of any vertex in $V \setminus S$
 - update path lengths based on new paths through v



Dijkstra's Algorithm

Dijkstra(G, w, s)

$S \leftarrow \{s\}$

$d[s] \leftarrow 0$

while $S \neq V$ {

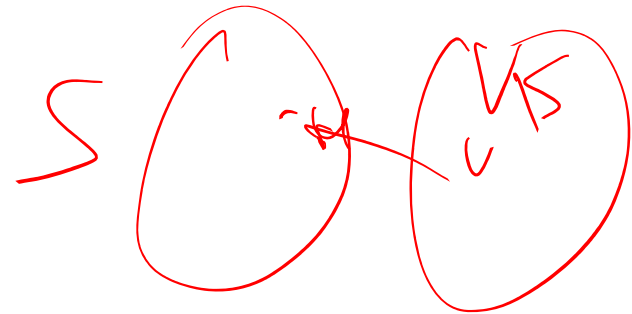
among all edges $e = (u, v)$ s.t. $v \notin S$ and $u \in S$ select* one with the minimum value of $d[u] + w(e)$

$S \leftarrow S \cup \{v\}$

$d[v] \leftarrow d[u] + w(e)$

$pred[v] \leftarrow u$

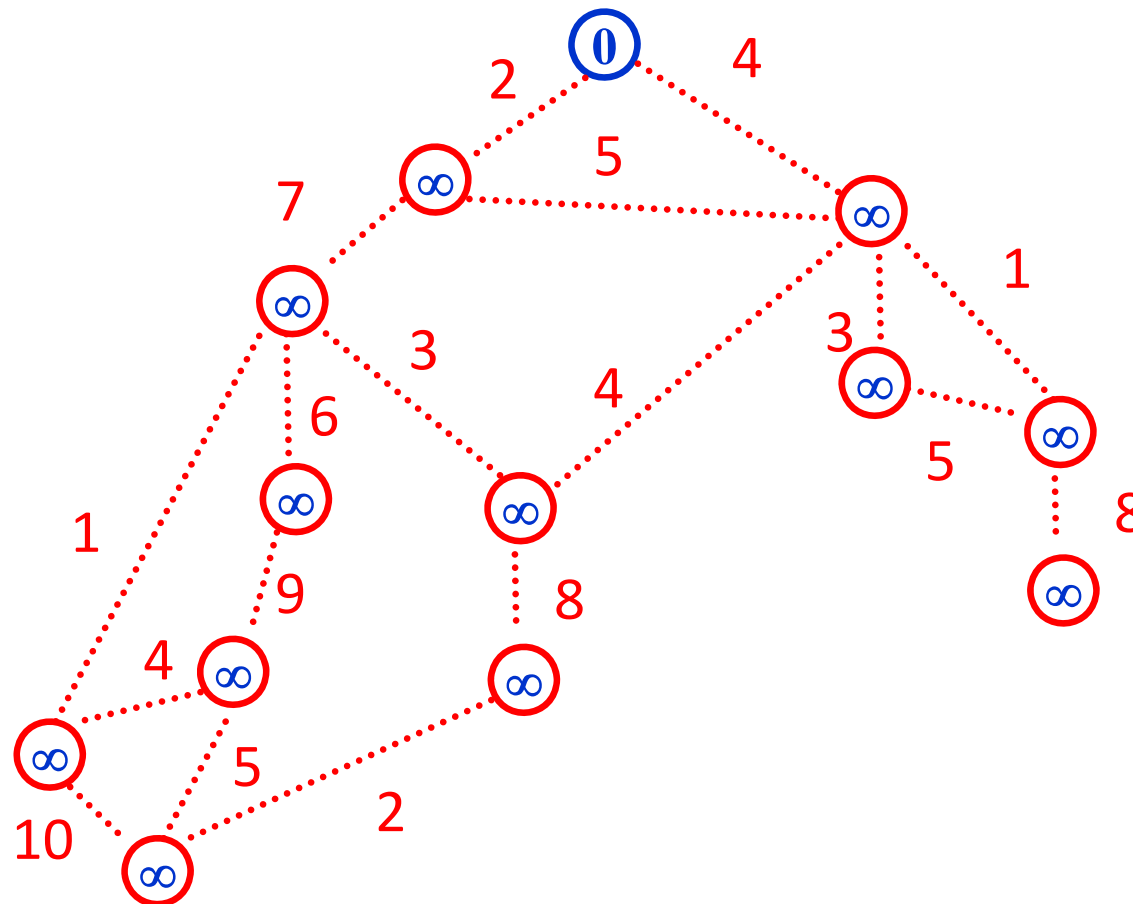
}



*For each $v \notin S$ maintain $d'[v]$ = minimum value of $d[u] + w(e)$
over all vertices $u \in S$ s.t. $e = (u, v)$ is in G

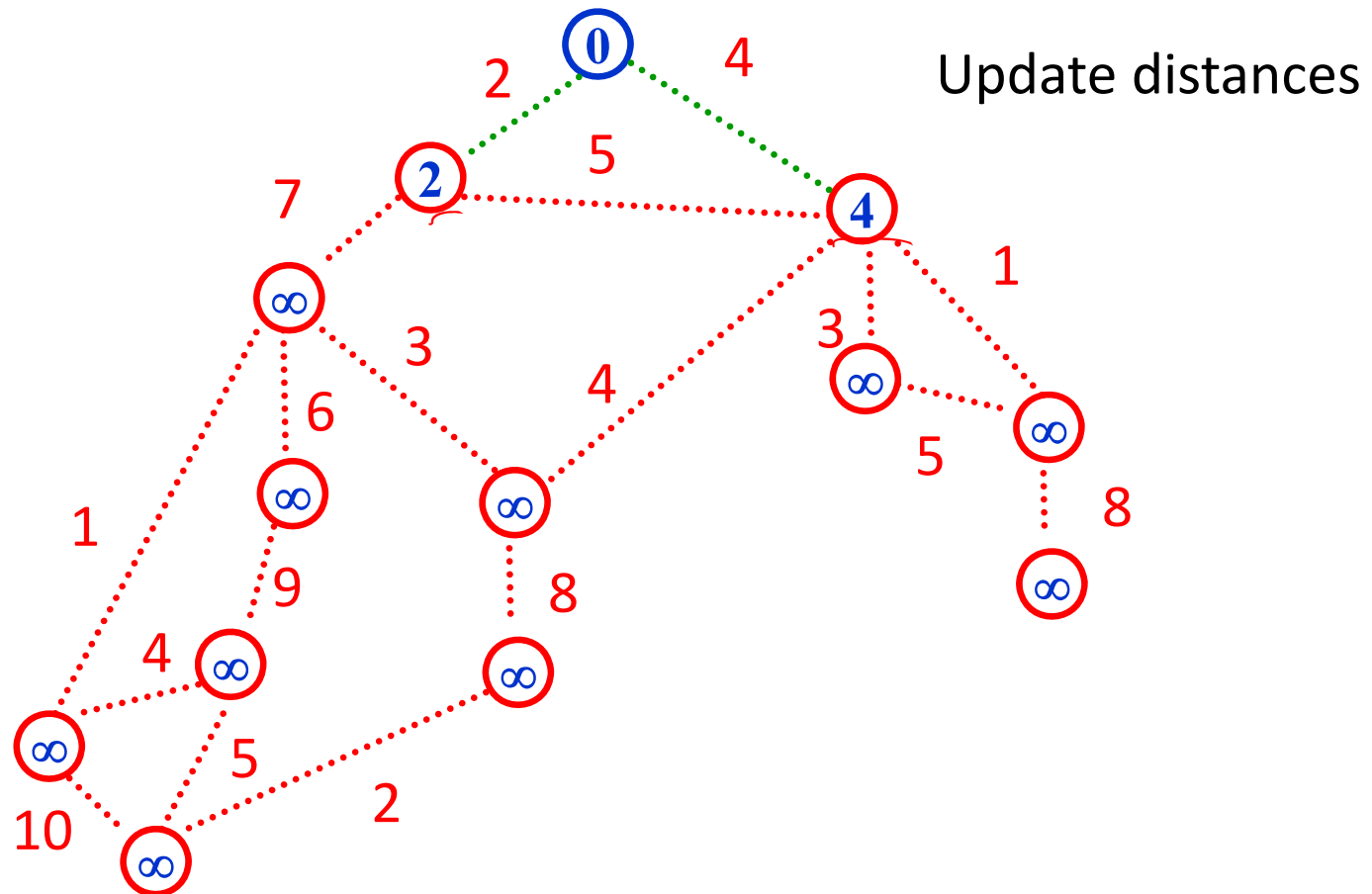
~~Handwritten scribble~~

Dijkstra's Algorithm

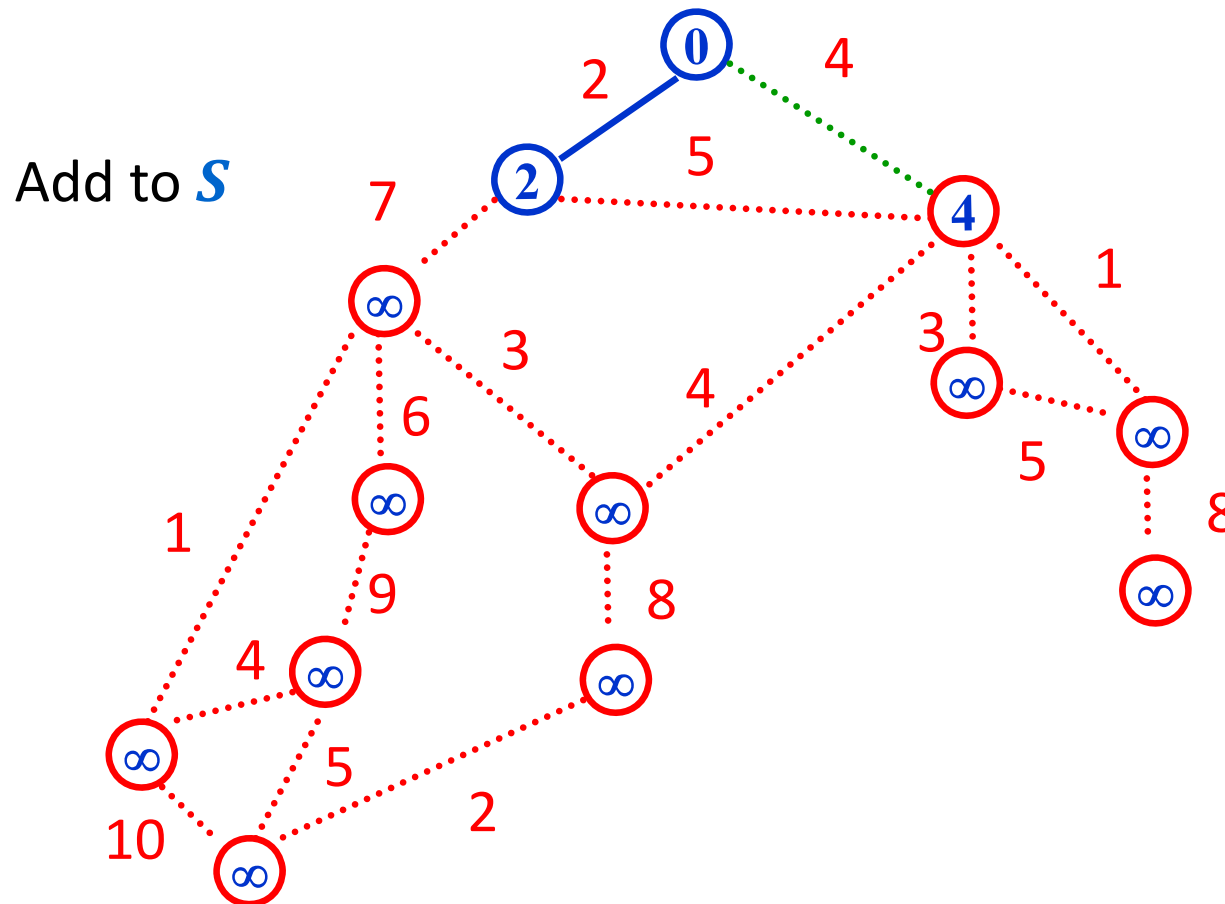


Add to S

Dijkstra's Algorithm

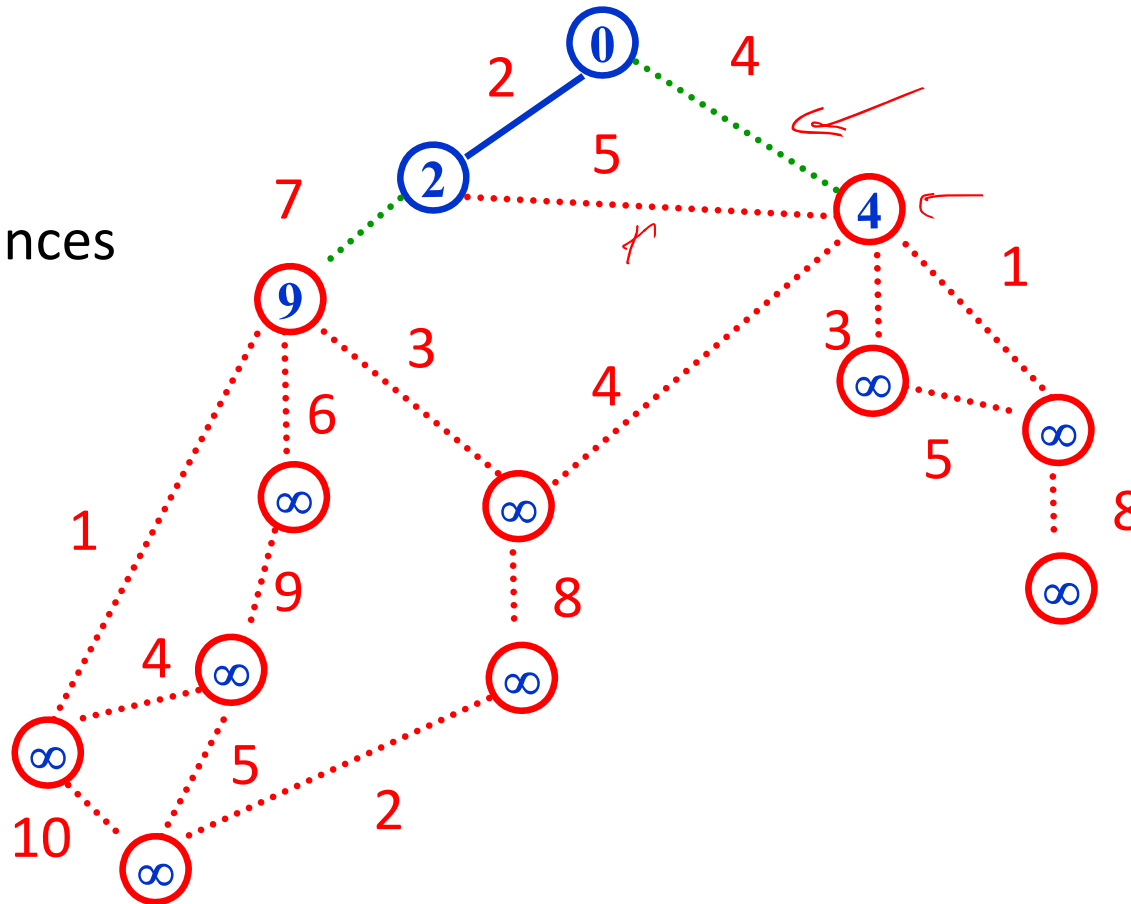


Dijkstra's Algorithm

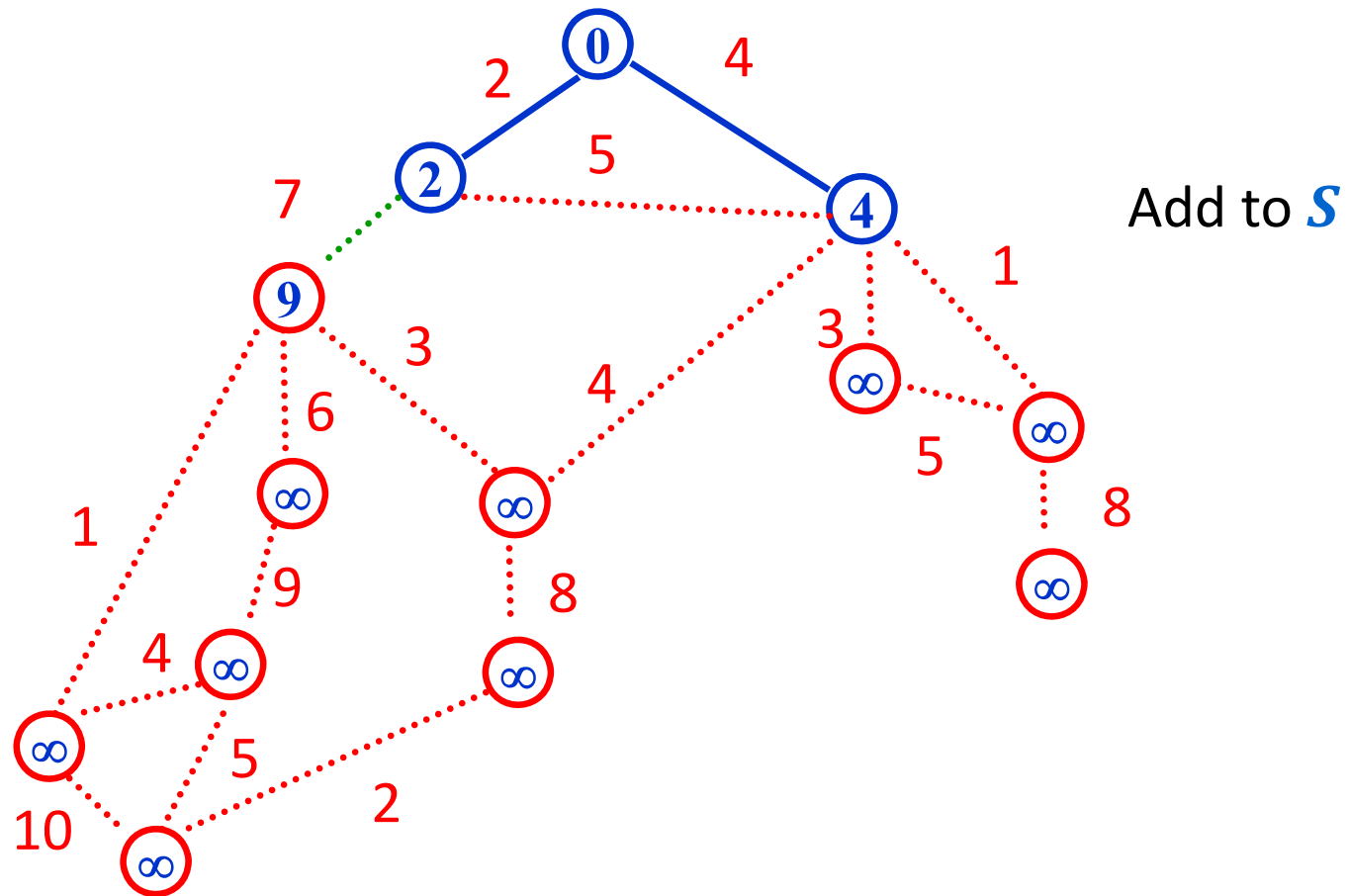


Dijkstra's Algorithm

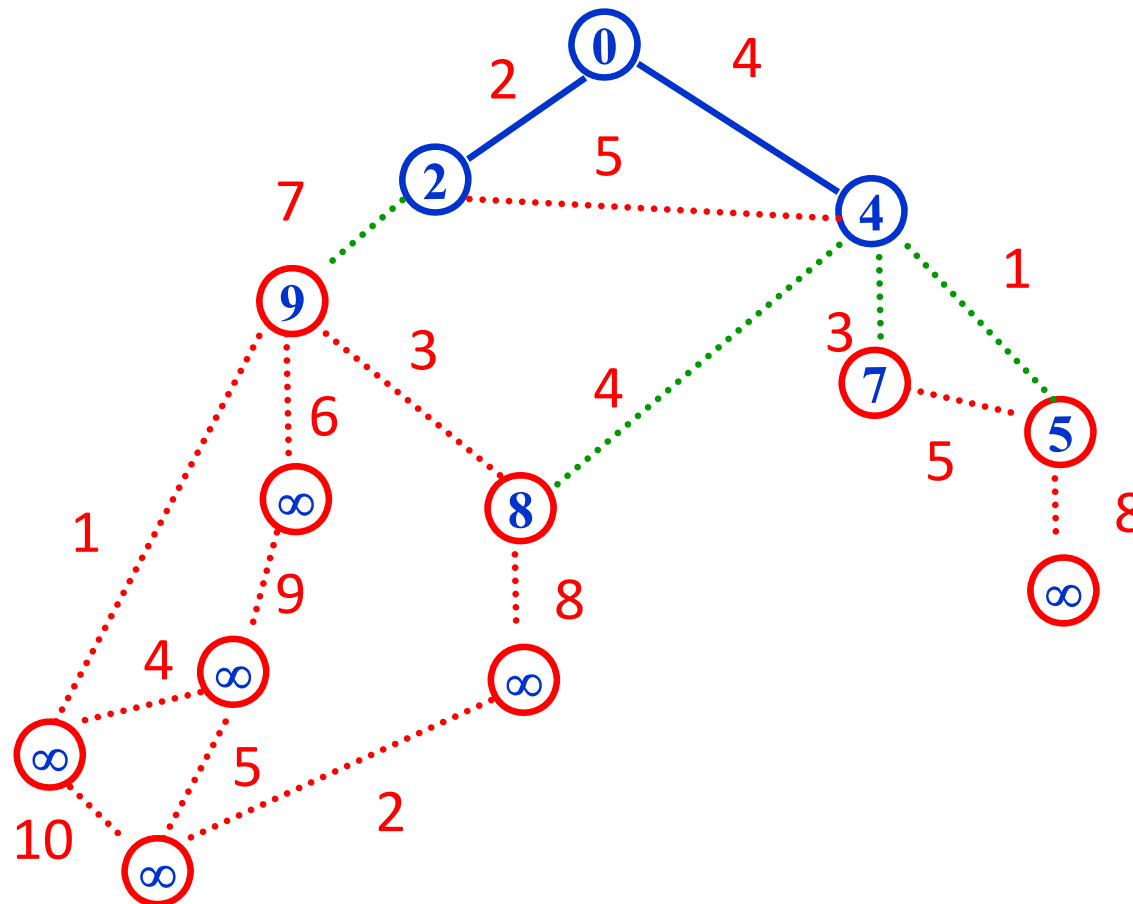
Update distances



Dijkstra's Algorithm

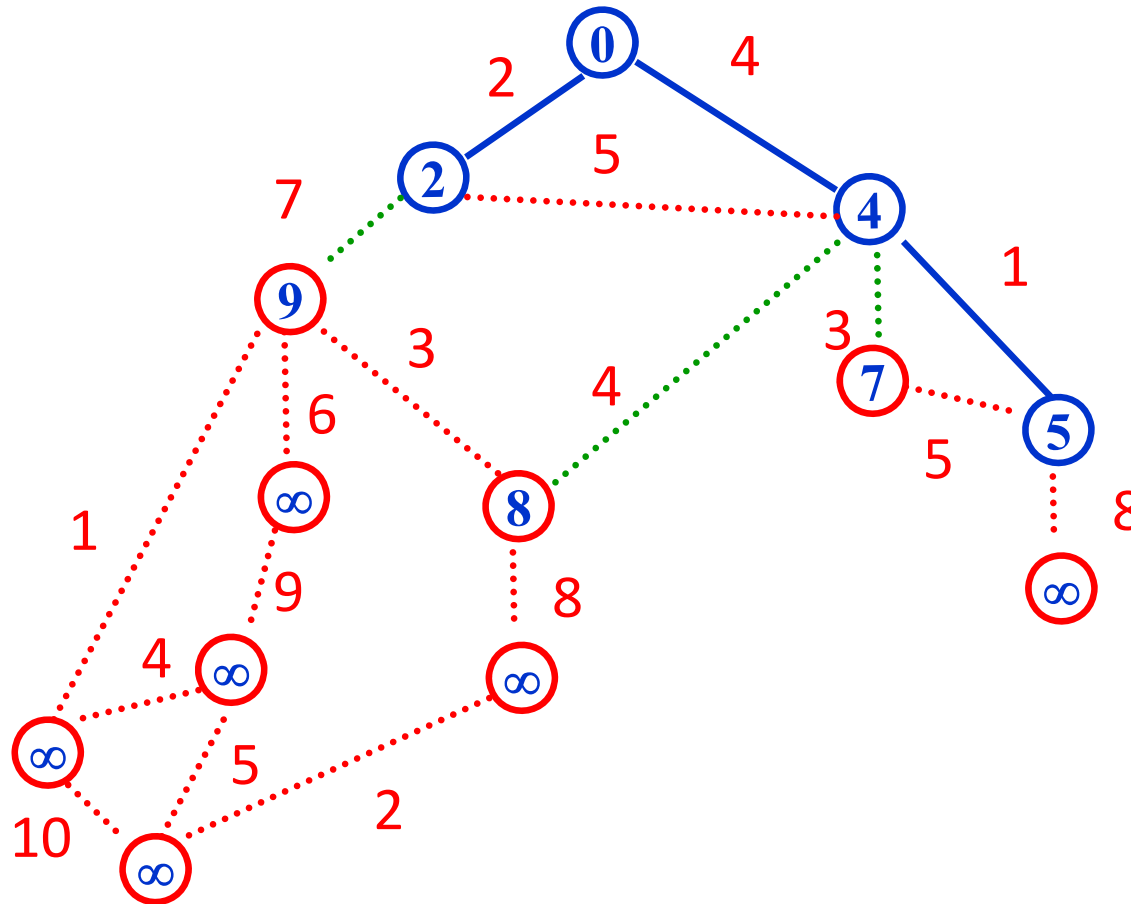


Dijkstra's Algorithm



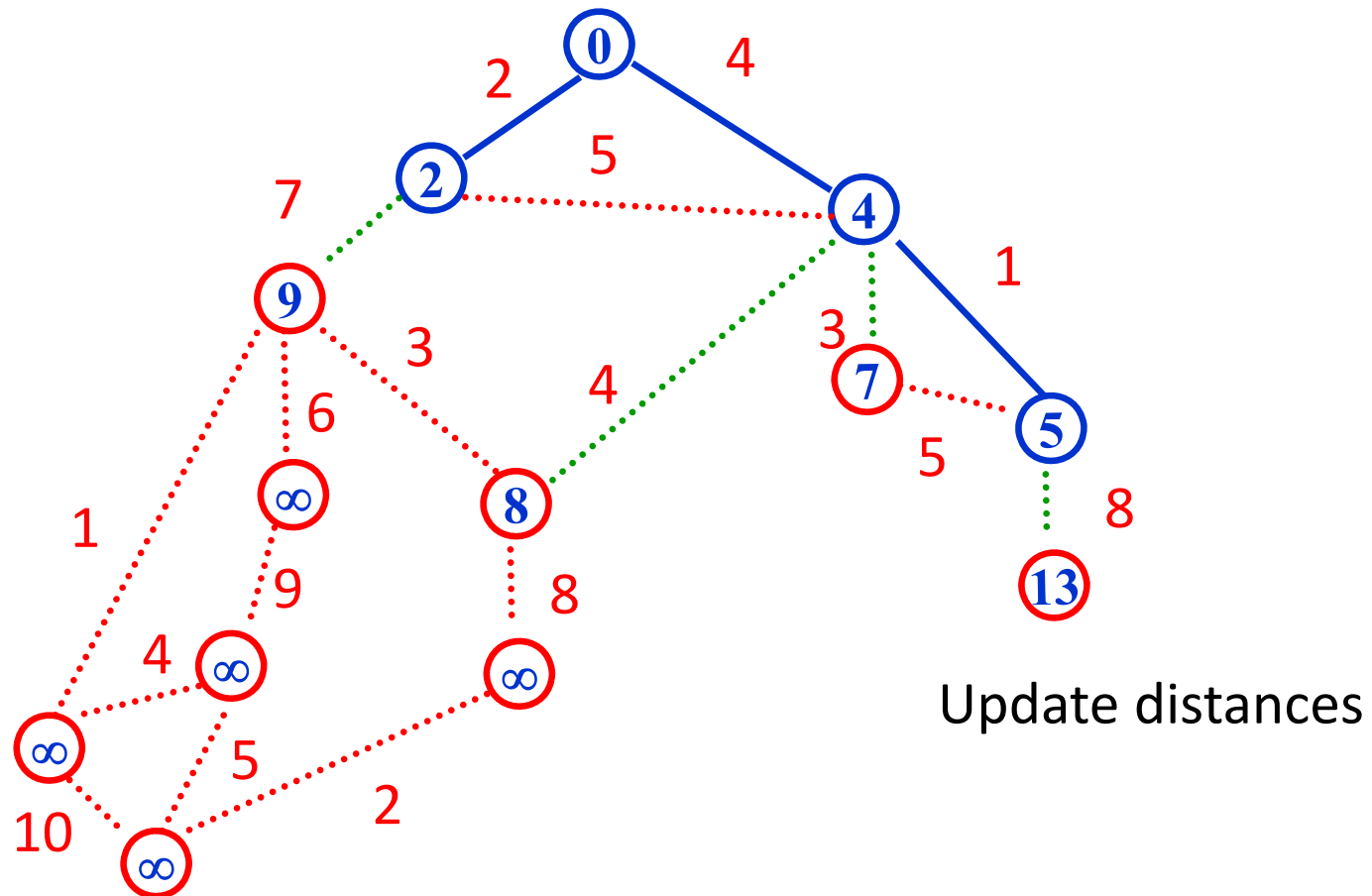
Update distances

Dijkstra's Algorithm

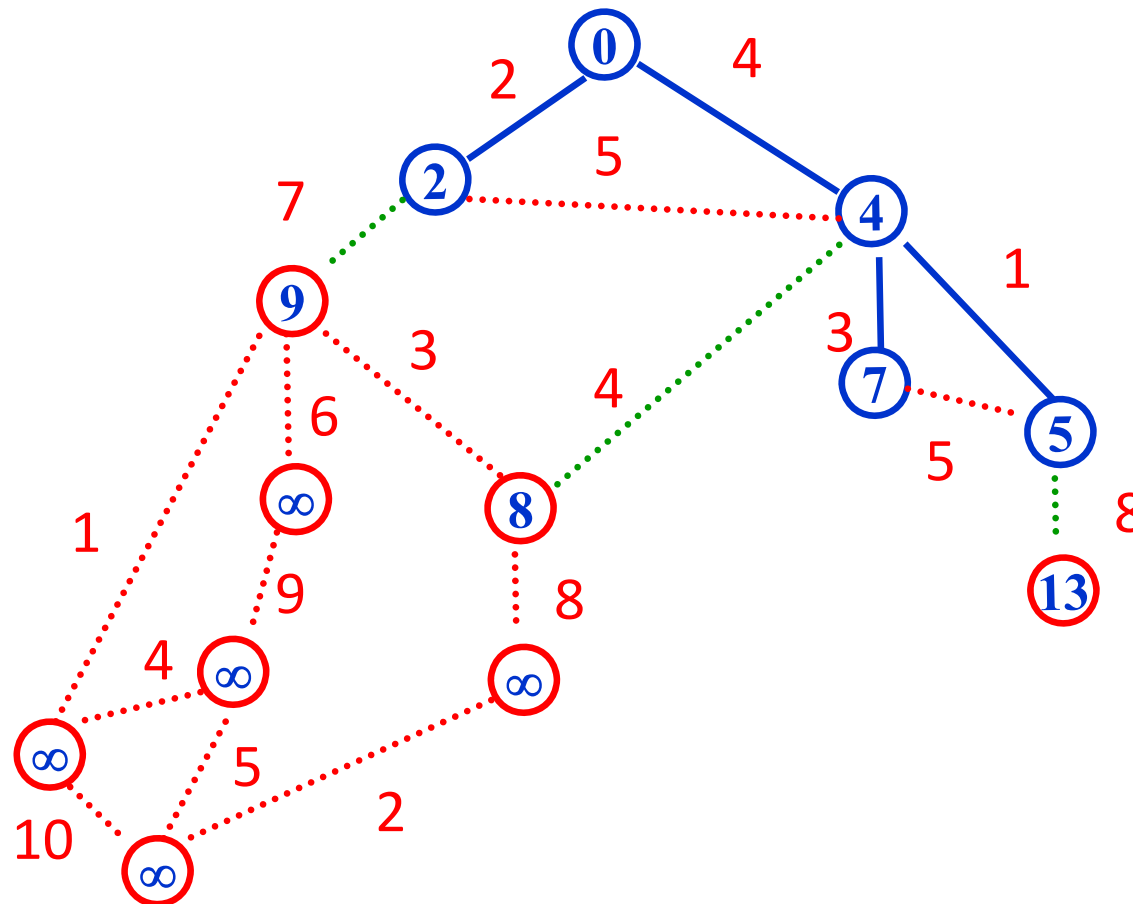


Add to S

Dijkstra's Algorithm

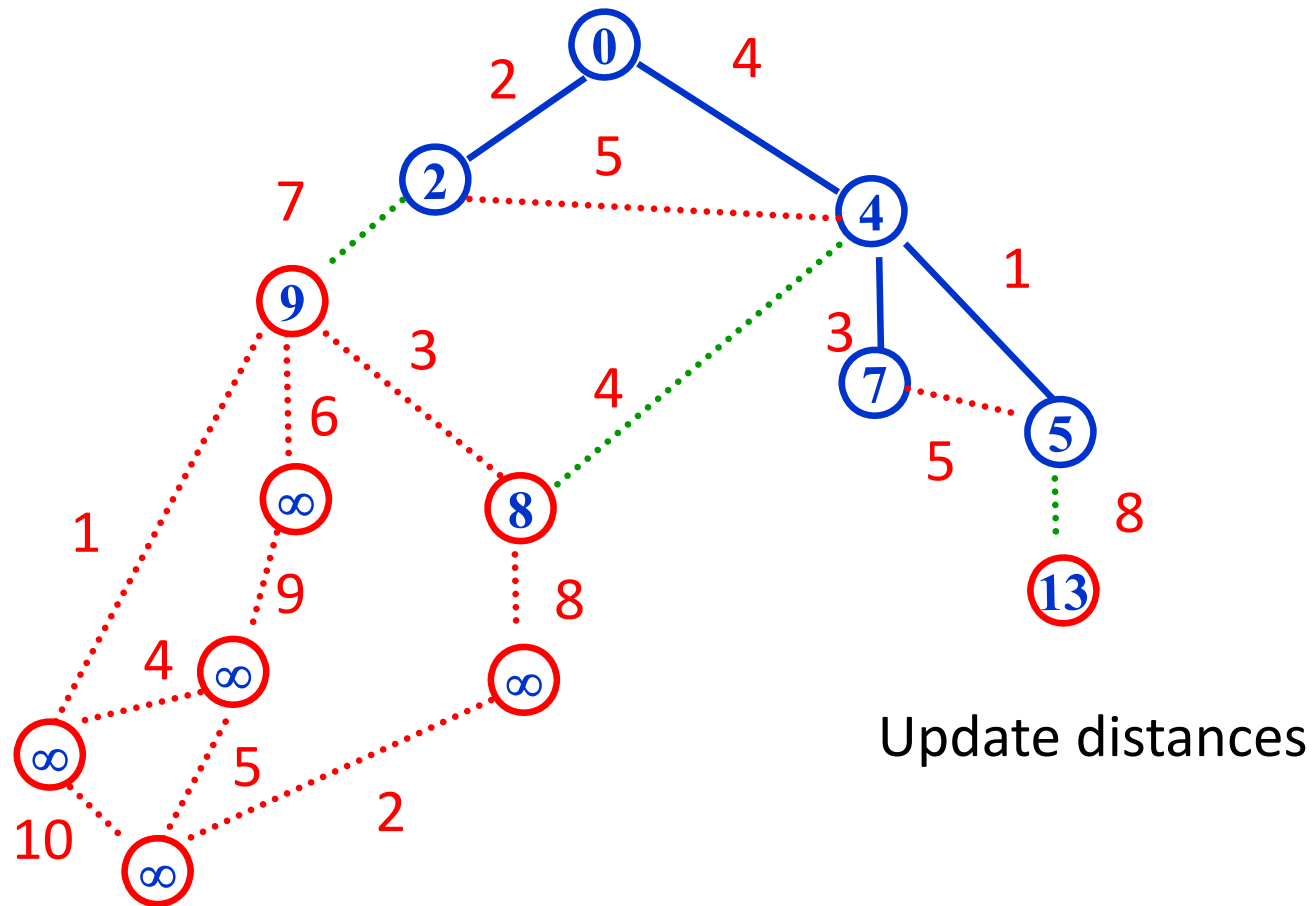


Dijkstra's Algorithm

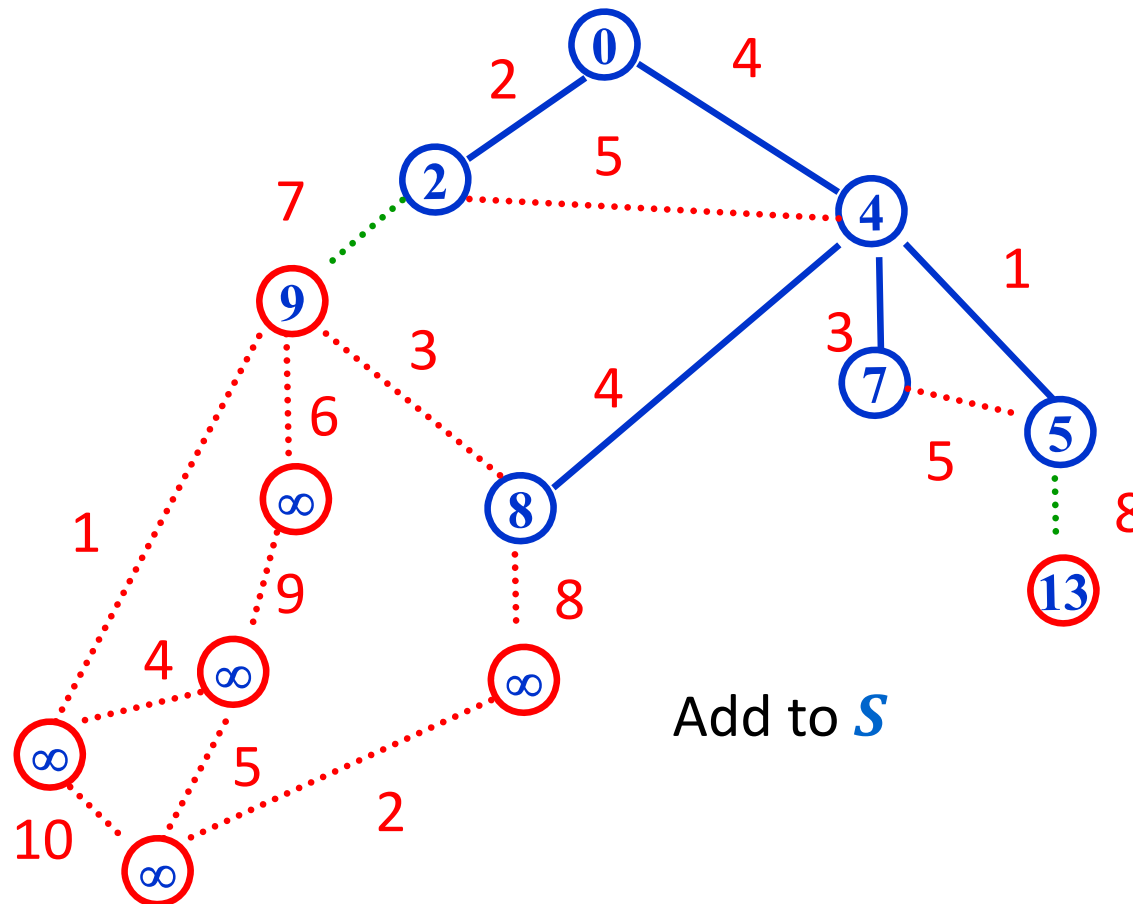


Add to **S**

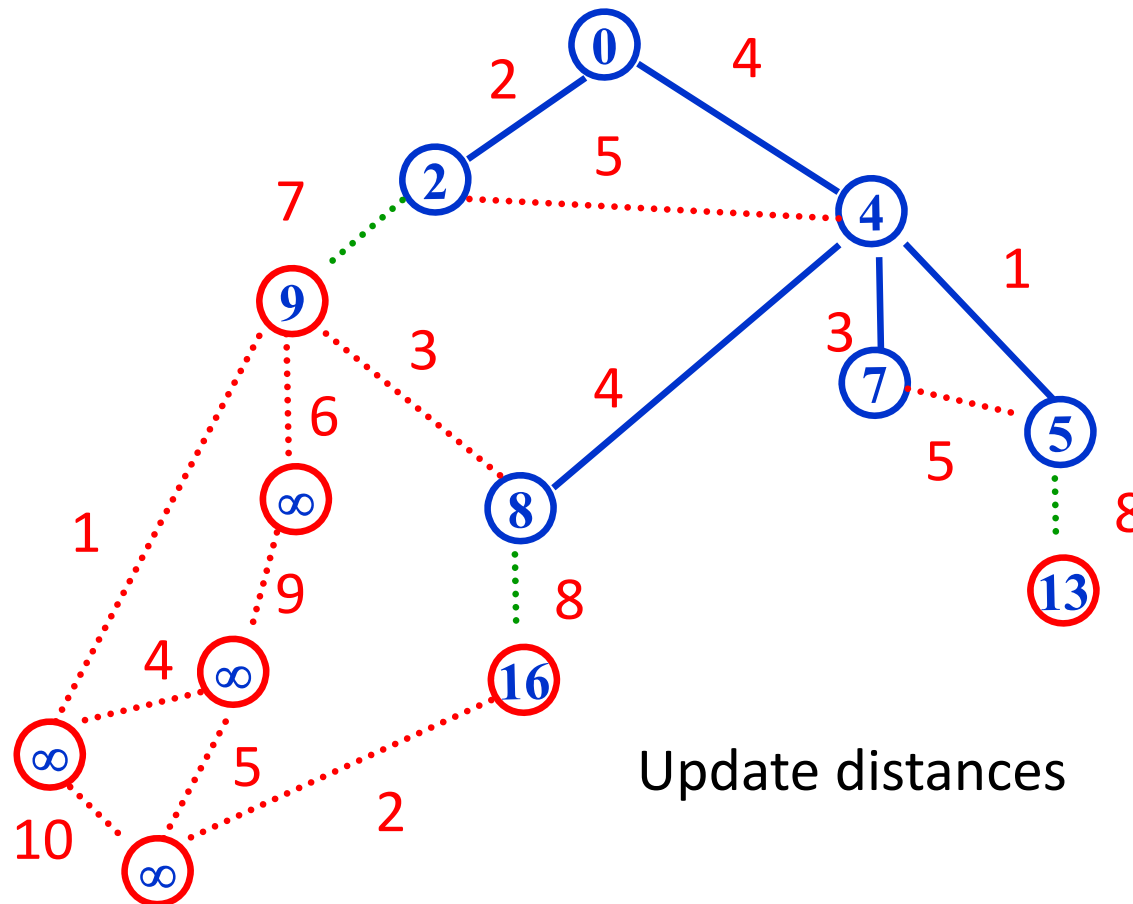
Dijkstra's Algorithm



Dijkstra's Algorithm

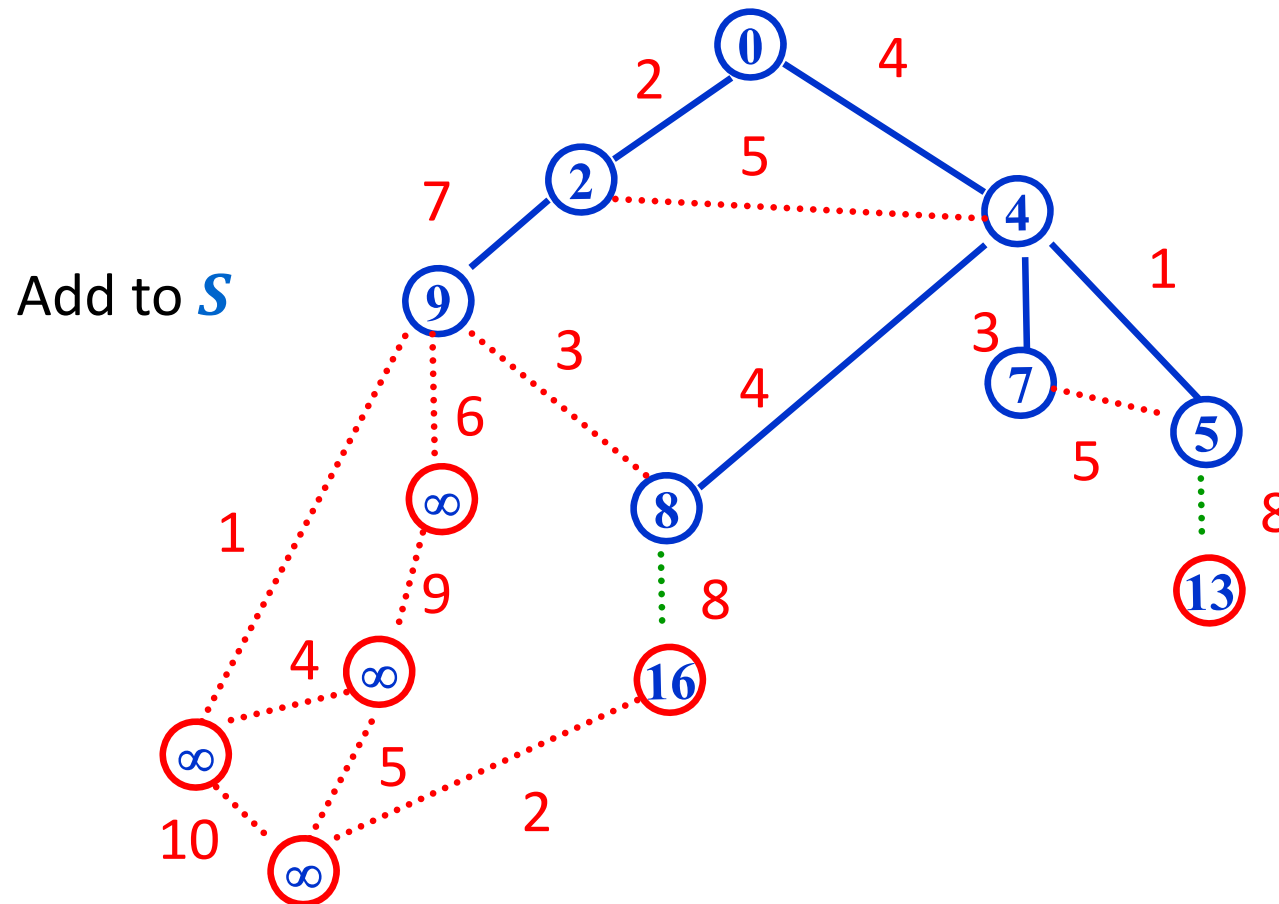


Dijkstra's Algorithm

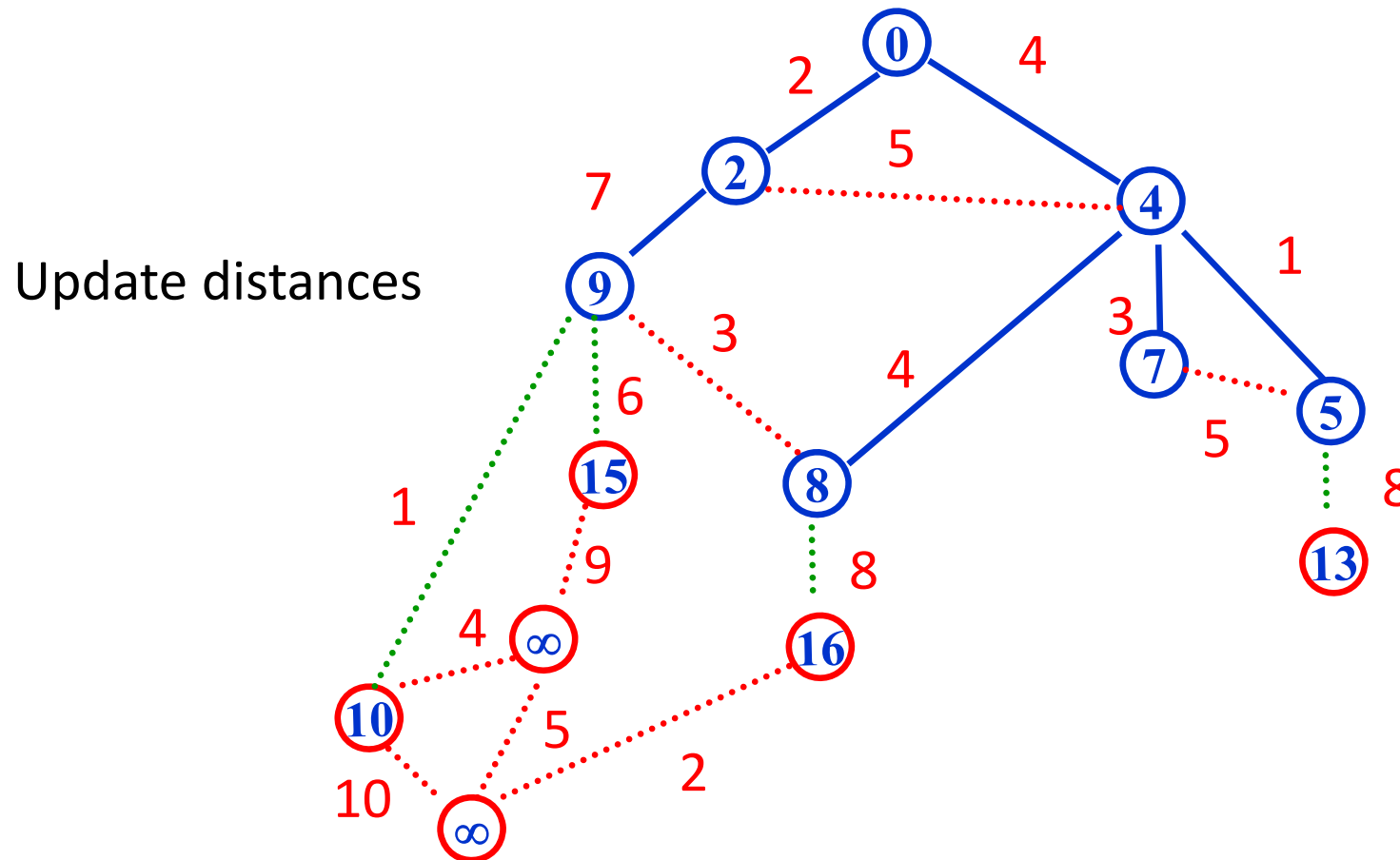


Update distances

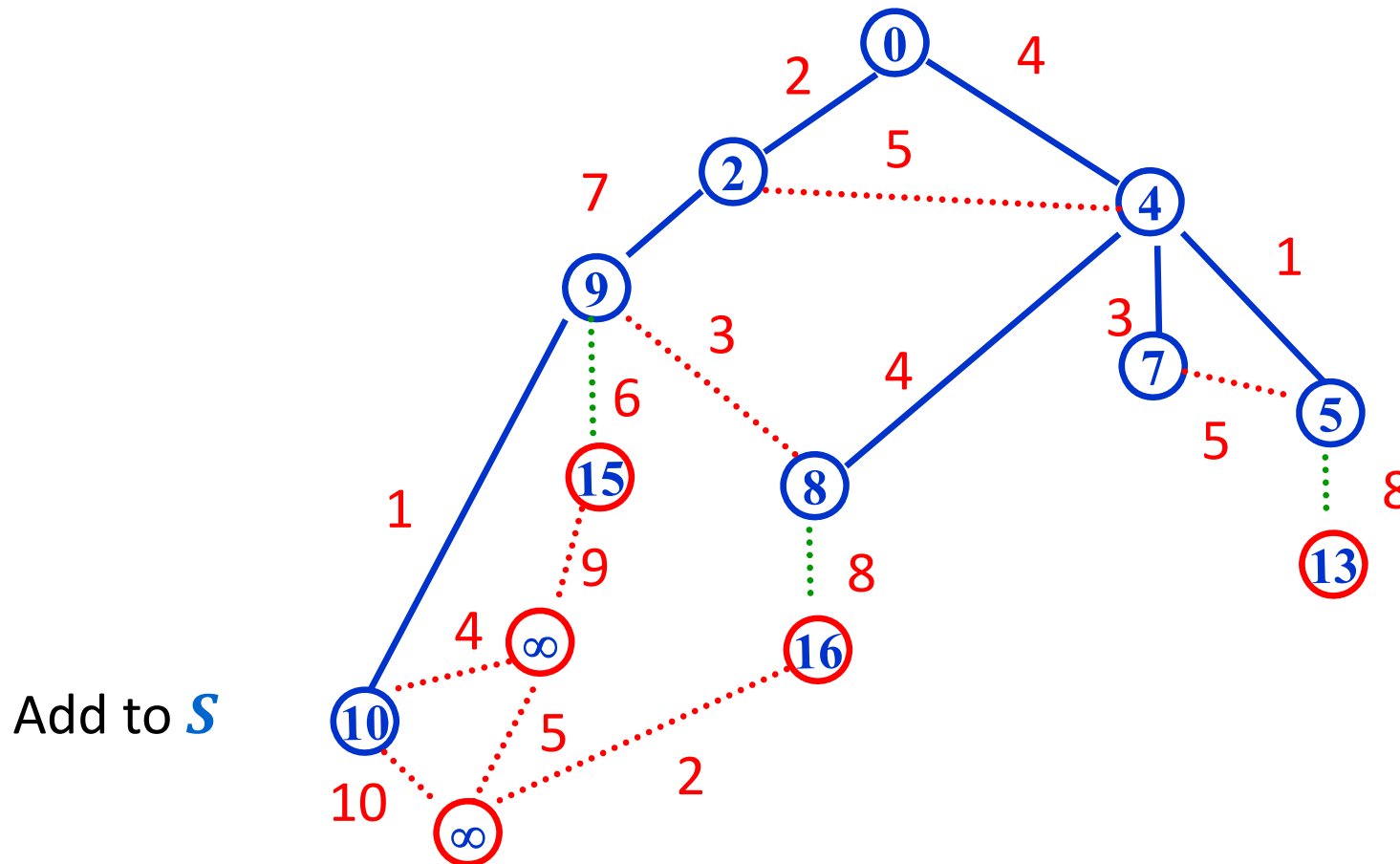
Dijkstra's Algorithm



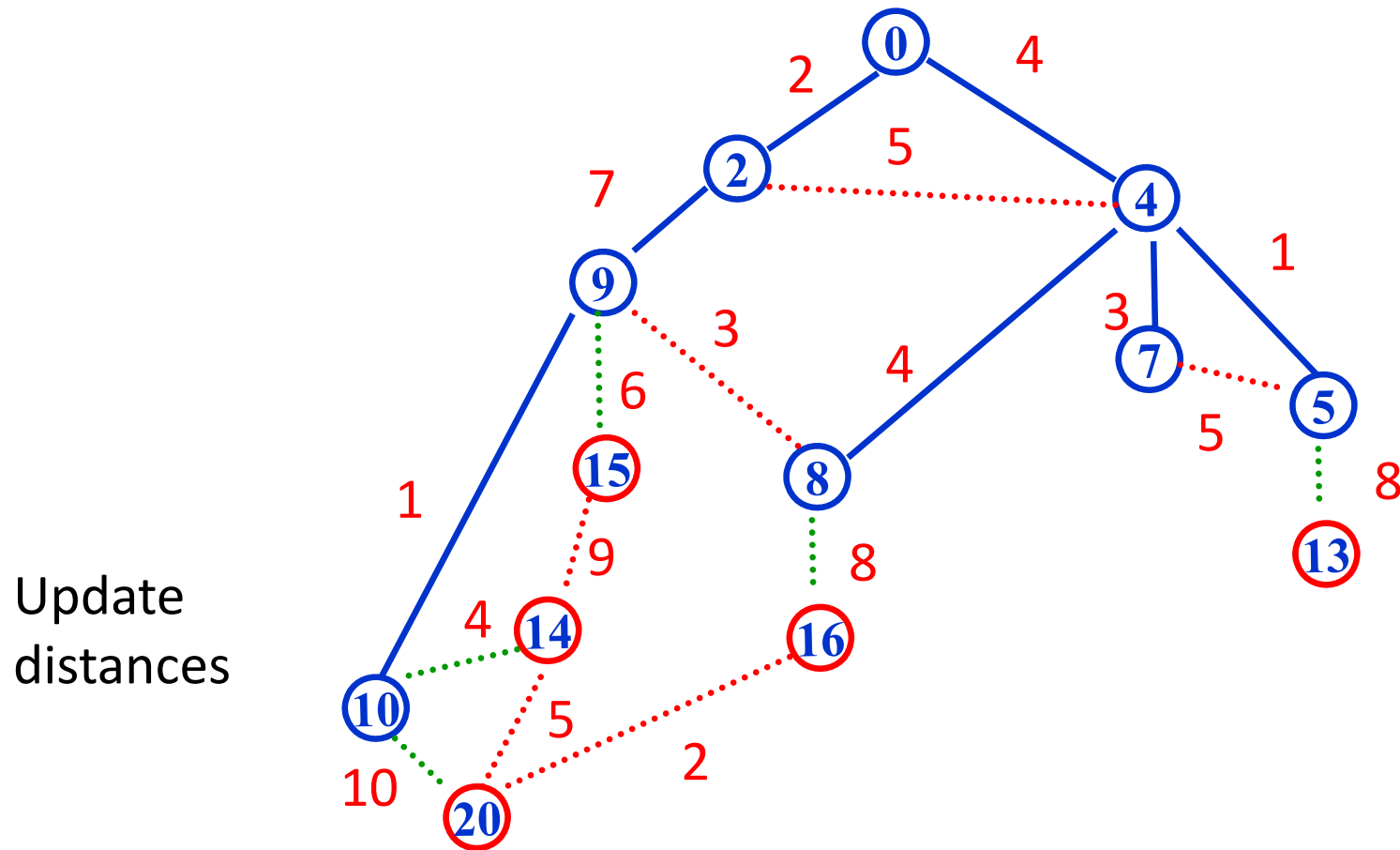
Dijkstra's Algorithm



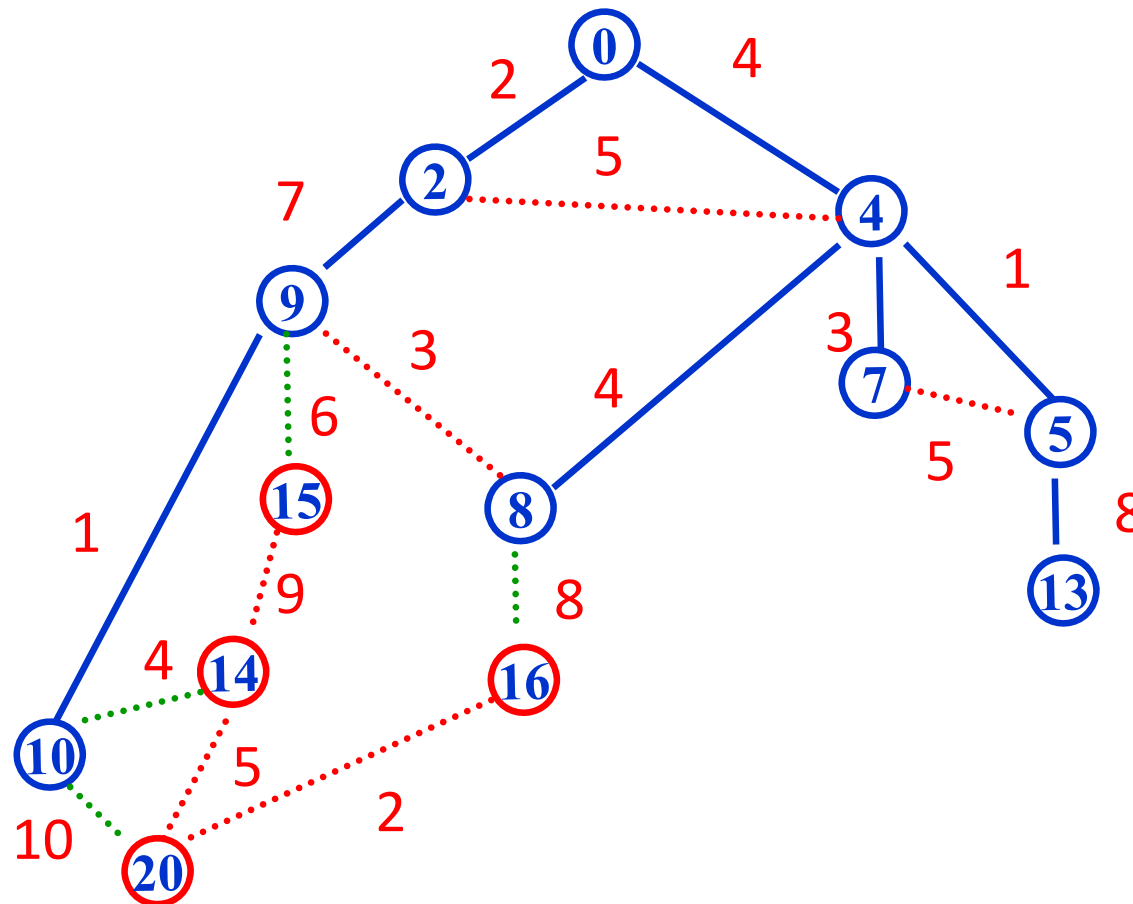
Dijkstra's Algorithm



Dijkstra's Algorithm

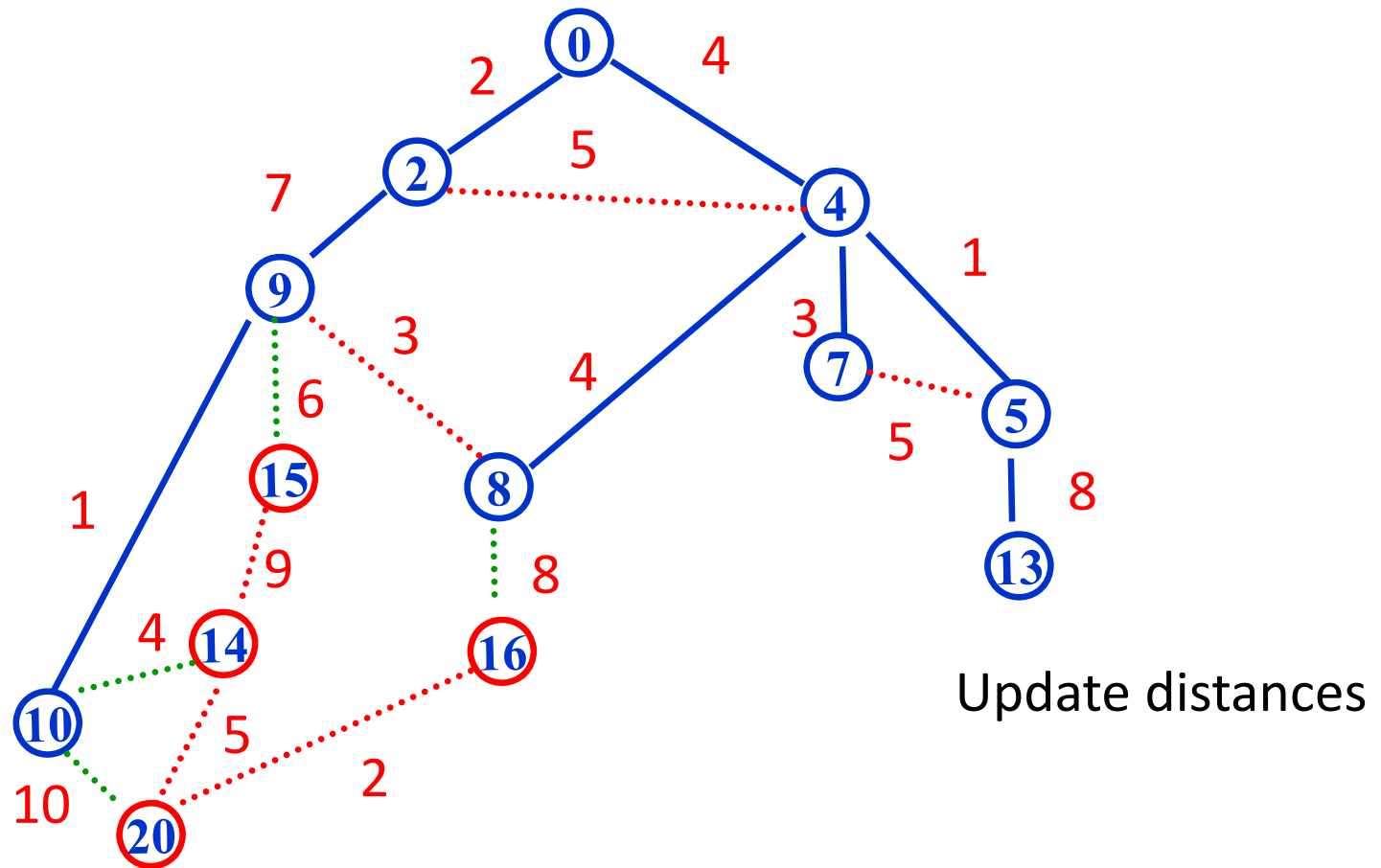


Dijkstra's Algorithm

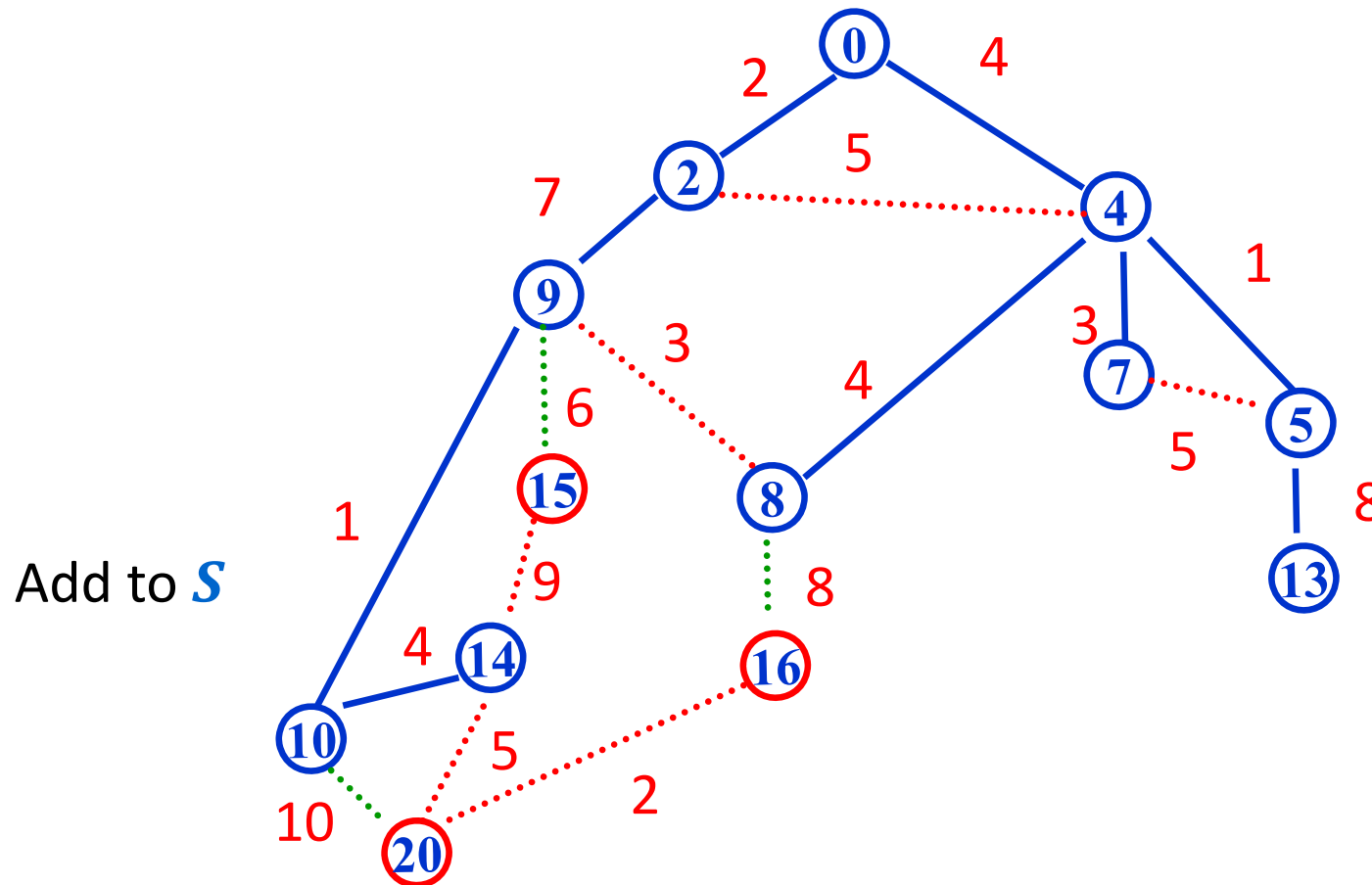


Add to S

Dijkstra's Algorithm

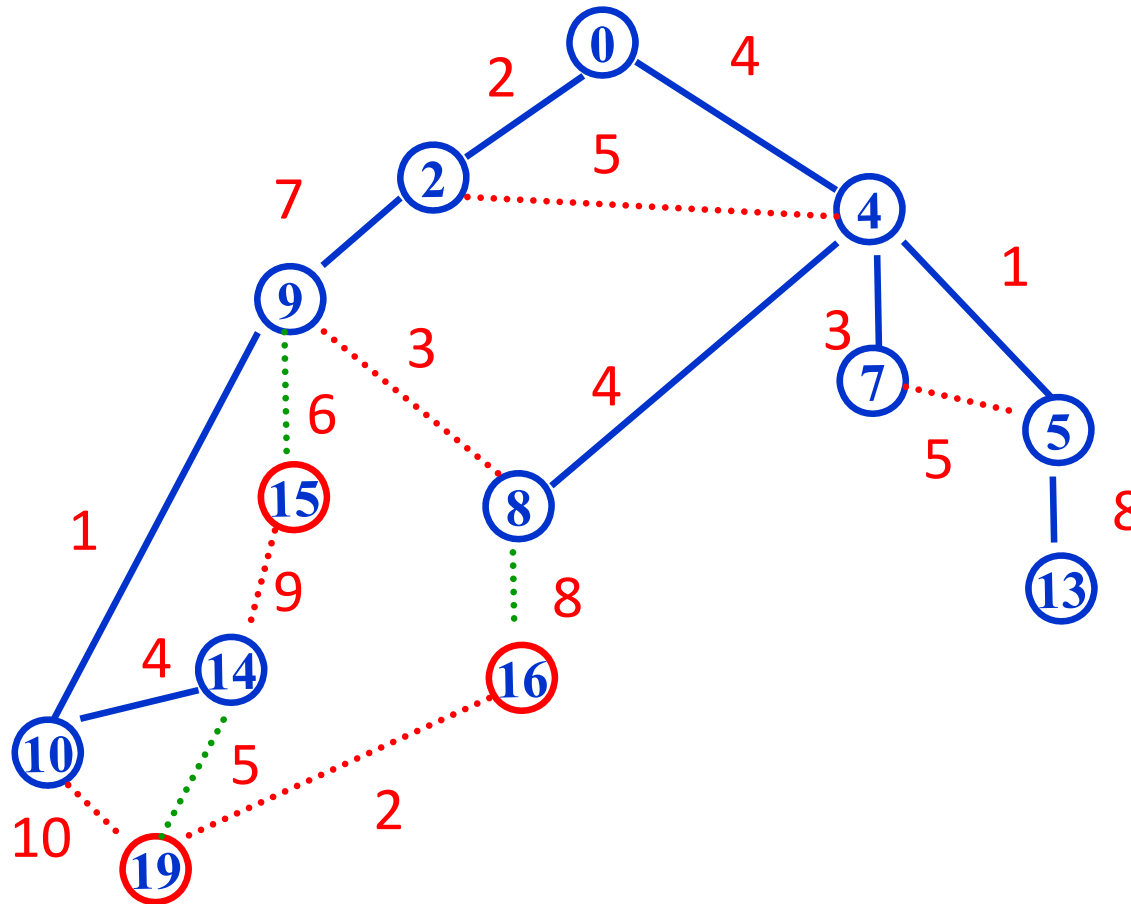


Dijkstra's Algorithm



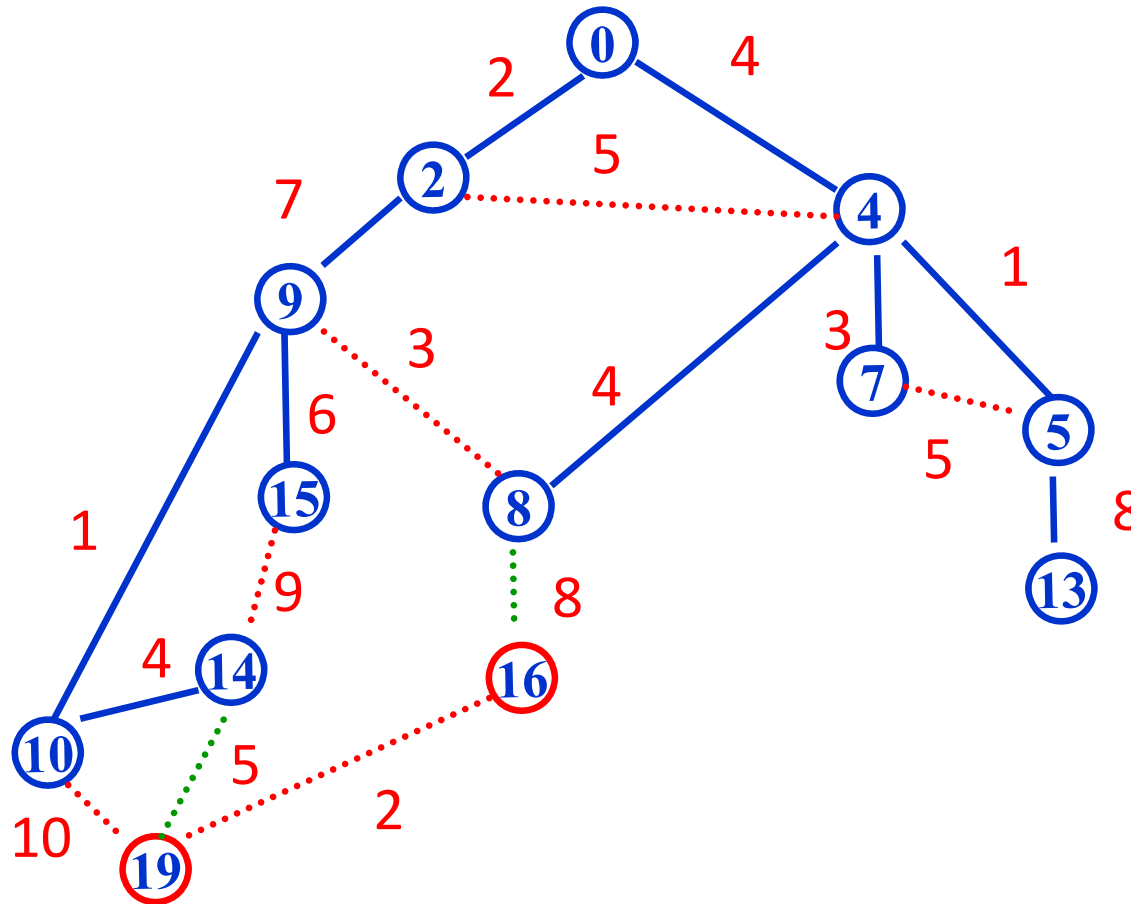
Dijkstra's Algorithm

Update
distances



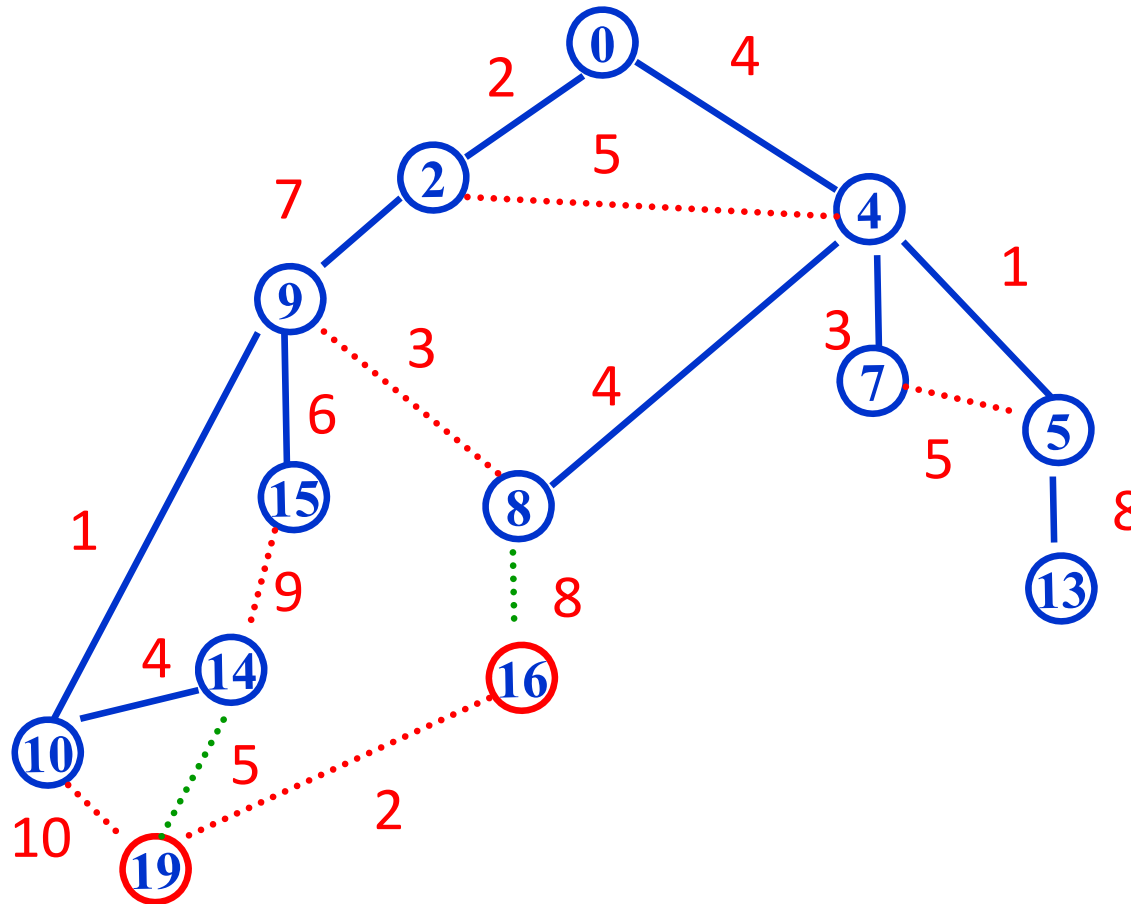
Dijkstra's Algorithm

Add to S

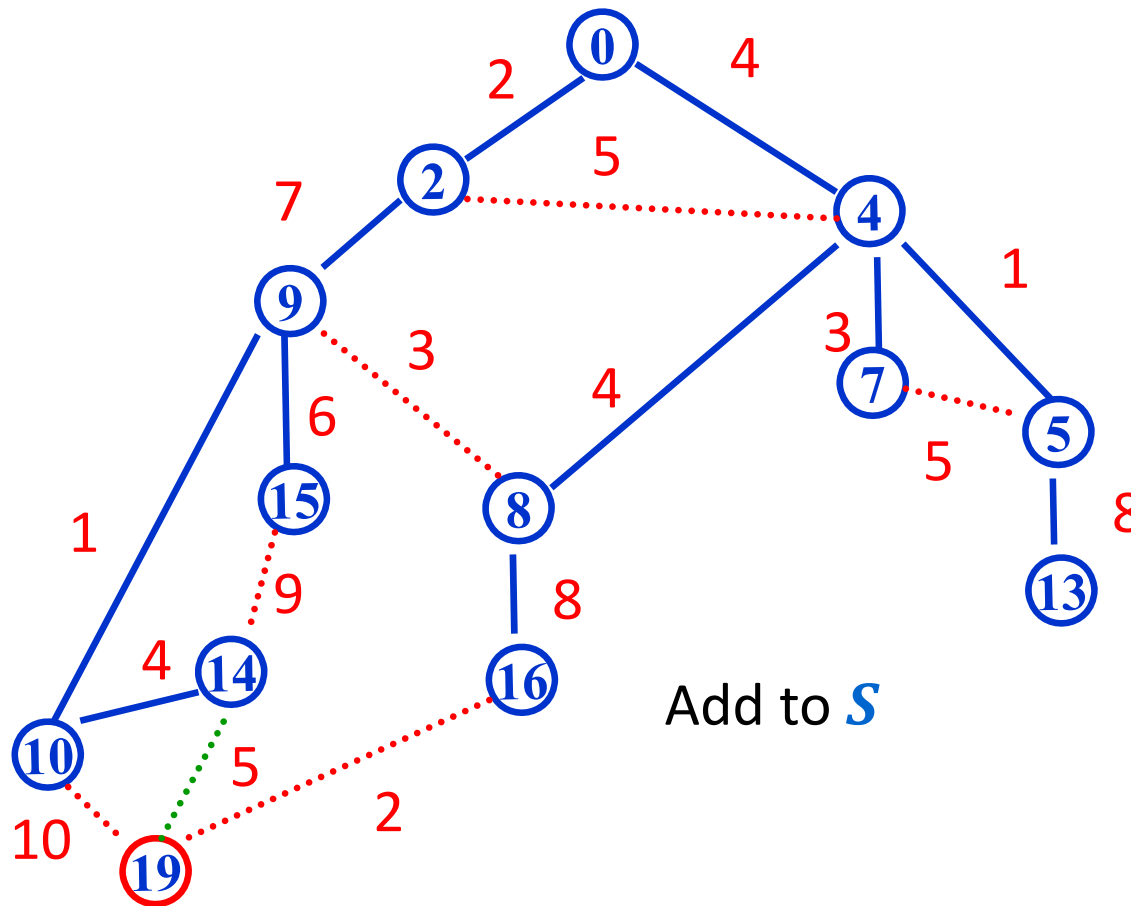


Dijkstra's Algorithm

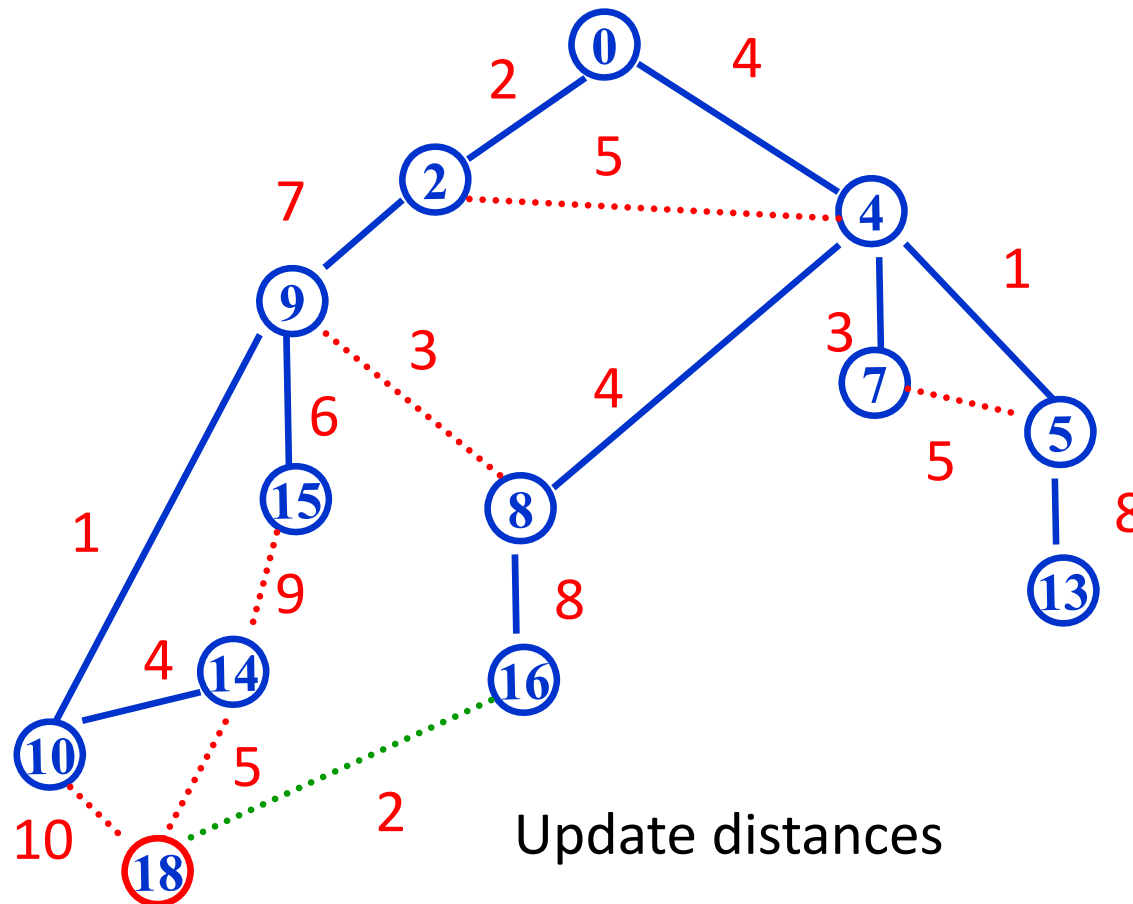
Update
distances



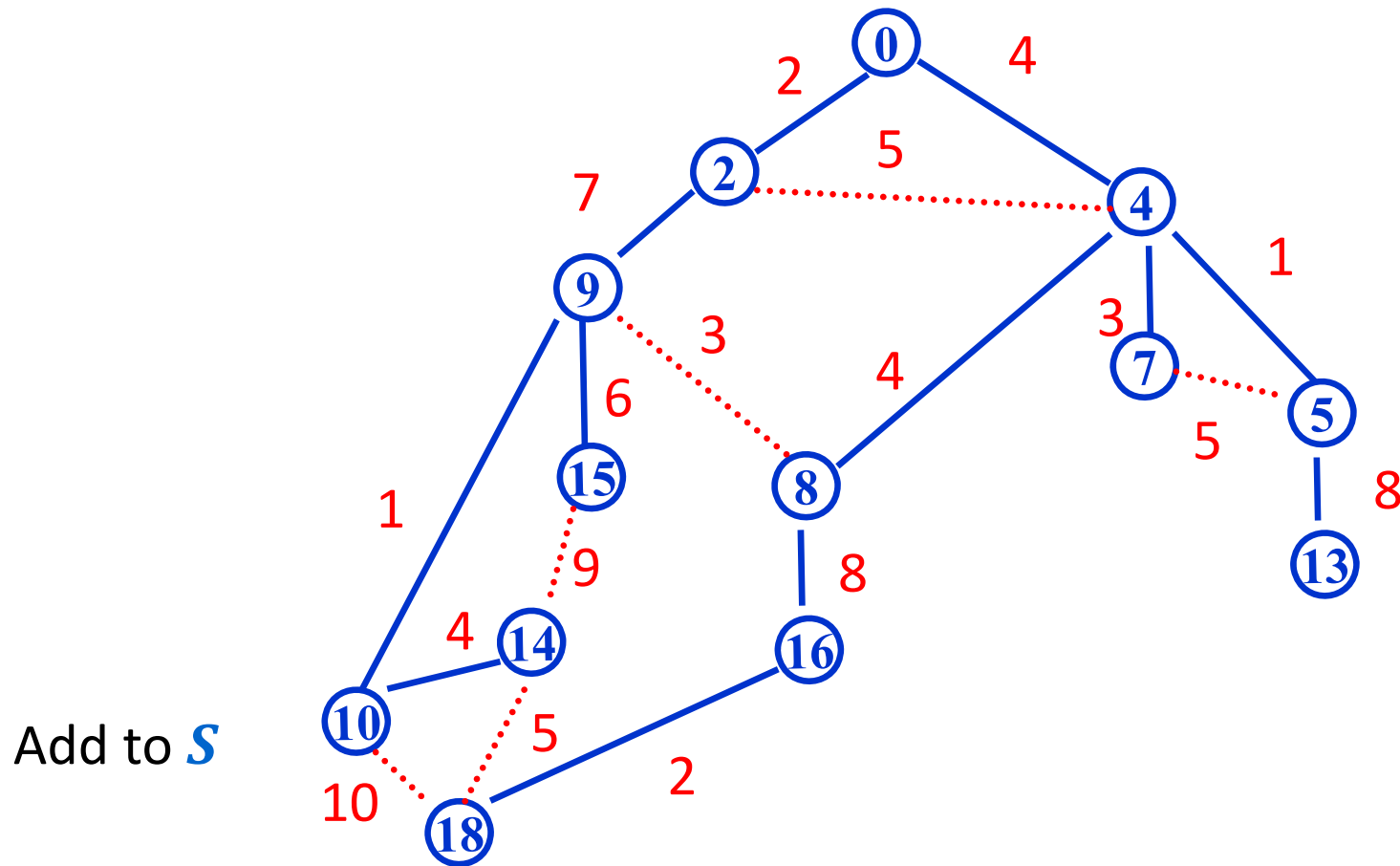
Dijkstra's Algorithm



Dijkstra's Algorithm



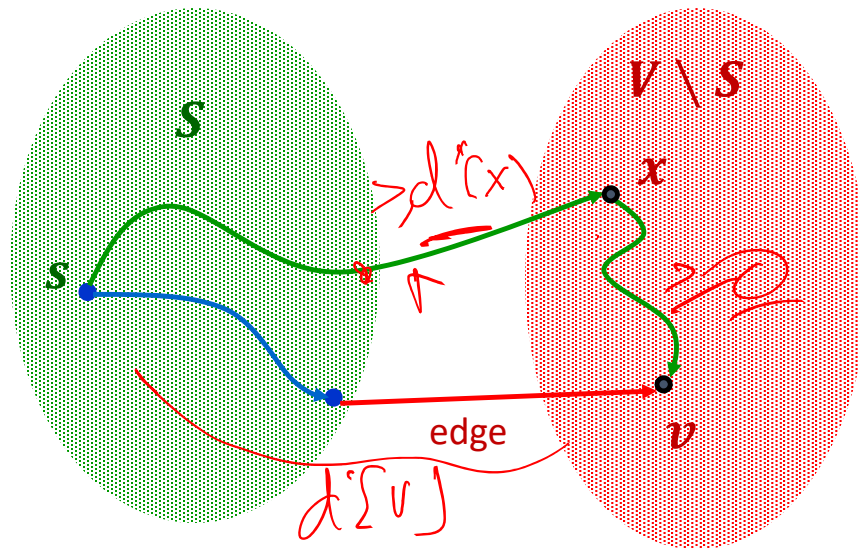
Dijkstra's Algorithm



Dijkstra's Algorithm Correctness

Suppose that all distances to vertices in S are correct and v has smallest current value $d'[v]$ in $V \setminus S$

$\Rightarrow d'[v] =$ length of shortest path from s to v with only last edge leaving S



Suppose some other path P to v .
Let $x = 1^{\text{st}}$ vertex on this path not in S

Since v was smallest, $d'[v] \leq d'[x]$

$x-v$ path length ≥ 0

\Rightarrow length of P is at least $d'[v]$

Therefore adding v to S maintains that all distances inside S are correct

Dijkstra's Algorithm

- Algorithm also produces a **tree** of shortest paths to v following the inverse of *pred* links
 - From v follow its ancestors in the tree back to s reversing edges along the path
- If all you care about is the shortest path from s to v simply stop the algorithm when v is added to S

Dijkstra's Algorithm

Dijkstra(G, w, s)

$S \leftarrow \{s\}$

$d[s] \leftarrow 0$

while $S \neq V$ {

among all edges $e = (u, v)$ s.t. $v \notin S$ and $u \in S$ select* one with the minimum value of $d[u] + w(e)$

$S \leftarrow S \cup \{v\}$

$d[v] \leftarrow d[u] + w(e)$

$pred[v] \leftarrow u$

}

*For each $v \notin S$ maintain $d'[v]$ = minimum value of $d[u] + w(e)$
over all vertices $u \in S$ s.t. $e = (u, v)$ is in G

Implementing Dijkstra's Algorithm

Need to

- keep current distance values $d'[\cdot]$ for nodes in $V \setminus S$
- find minimum current distance value $d'[v]$
- reduce distances in $d'[\cdot]$ when vertex v moved to S

Data Structure Review

Priority Queue:

- Elements each with an associated **key**
- Operations
 - **Insert**
 - **Find-min**
 - Return the element with the smallest key
 - **Delete-min**
 - Return the element with the smallest key and delete it from the data structure
 - **Decrease-key**
 - Decrease the key value of some element

Implementations

- Arrays: $O(n)$ time find/delete-min, $O(1)$ time insert/decrease-key
- Heaps: $O(\log n)$ time insert/decrease-key/delete-min, $O(1)$ time find-min

Dijkstra's Algorithm with Priority Queues

- For each vertex v not in tree maintain cost $d'[v]$ of current cheapest path through tree to v
 - Store v in priority queue with key = length of this path
- Operations:
 - $n - 1$ insertions (each vertex added once)
 - $n - 1$ delete-mins (each vertex deleted once)
 - pick the vertex of smallest key, remove it from the priority queue and add its edge to the graph
 - $< m$ decrease-keys (each edge updates one vertex)

Dijkstra's Algorithm with Priority Queues

Priority queue implementations

Use
4-heaps

- Array
 - insert $O(1)$, delete-min $O(n)$, decrease-key $O(1)$
 - total $O(n + n^2 + m) = O(n^2)$

- Heap
 - insert, delete-min, decrease-key all $O(\log n)$
 - total $O(m \log n)$

Worse if $m = \Theta(n^2)$



Better for all values of m

- d -Heap ($d = \lceil m/n \rceil$)
 - insert, decrease-key $O(\log_{m/n} n)$
 - delete-min $O((m/n) \log_{m/n} n)$
 - total $O(m \log_{m/n} n)$