CSE 421 Winter 2025
Lecture 7: Greedy Part 2

Nathan Brunelle

http://www.cs.uw.edu/421

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

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

- Consider an arbitrary other PB&J sandwich. Show that every ingredient I use increases the deliciousness by at least as much as the other sandwich's ingredient.

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.

- Show that the maximum deliciousness of a PB&J sandwich is 9.5/10, then show that my sandwich has a deliciousness score of 9.5.

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

- Consider an arbitrary other PB&J sandwich. Show that, for each ingredient, swapping it out with my choice won't decrease the deliciousness.

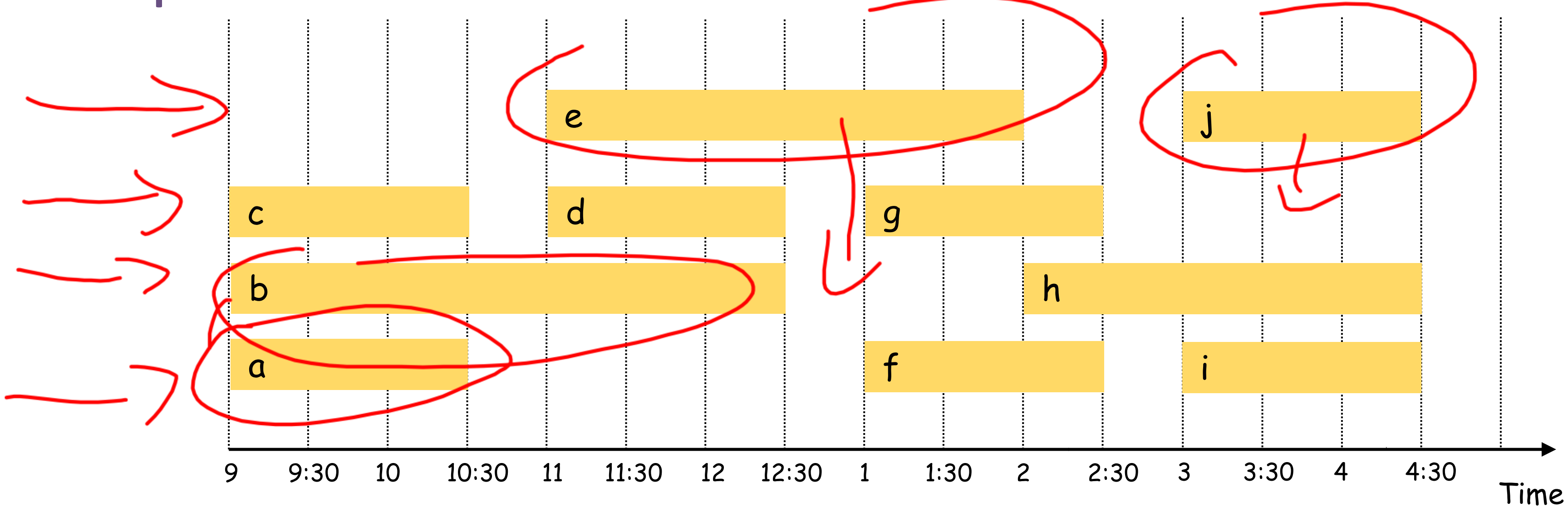
Scheduling All Intervals: Interval Partitioning

Interval Partitioning:

- Lecture j starts at s_j and finishes at f_j.

Goal: find minimum number of rooms to schedule all lectures so that no two occur at the same time in the same room.

Example: This schedule uses 4 rooms to schedule 10 lectures.



Can you do better?

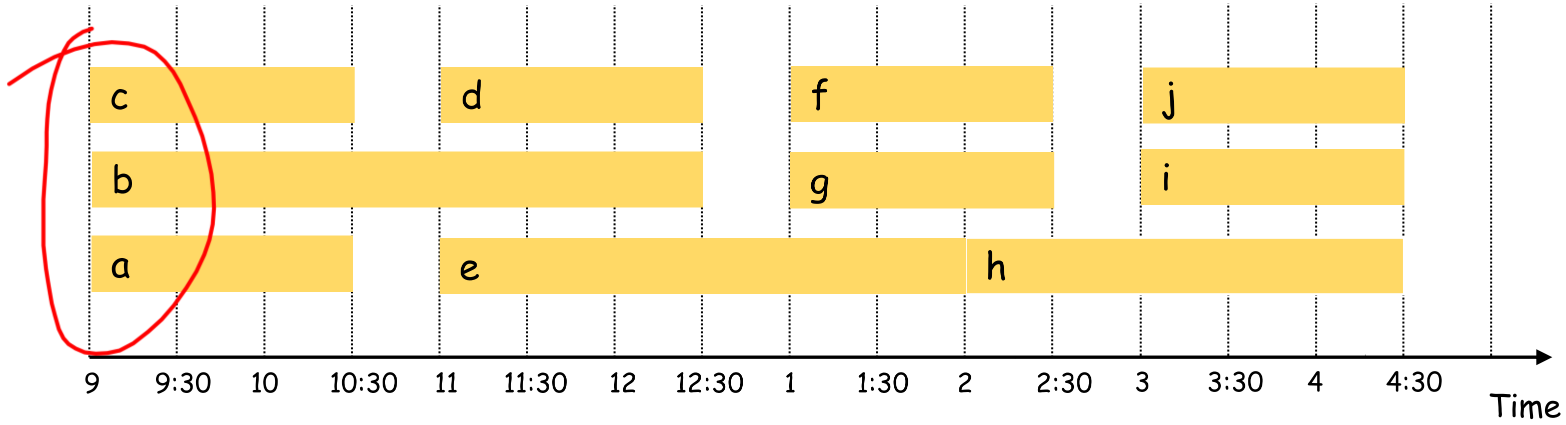
Scheduling All Intervals: Interval Partitioning

Interval Partitioning:

- Lecture j starts at s_j and finishes at f_j.

Goal: find minimum number of rooms to schedule all lectures so that no two occur at the same time in the same room.

Example: This schedule uses only 3 rooms.



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

- Consider an arbitrary other PB&J sandwich. Show that every ingredient I use increases the deliciousness by at least as much as the other sandwich's ingredient.

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.

- Show that the maximum deliciousness of a PB&J sandwich is 9.5/10, then show that my sandwich has a deliciousness score of 9.5.

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

- Consider an arbitrary other PB&J sandwich. Show that, for each ingredient, swapping it out with my choice won't decrease the deliciousness.

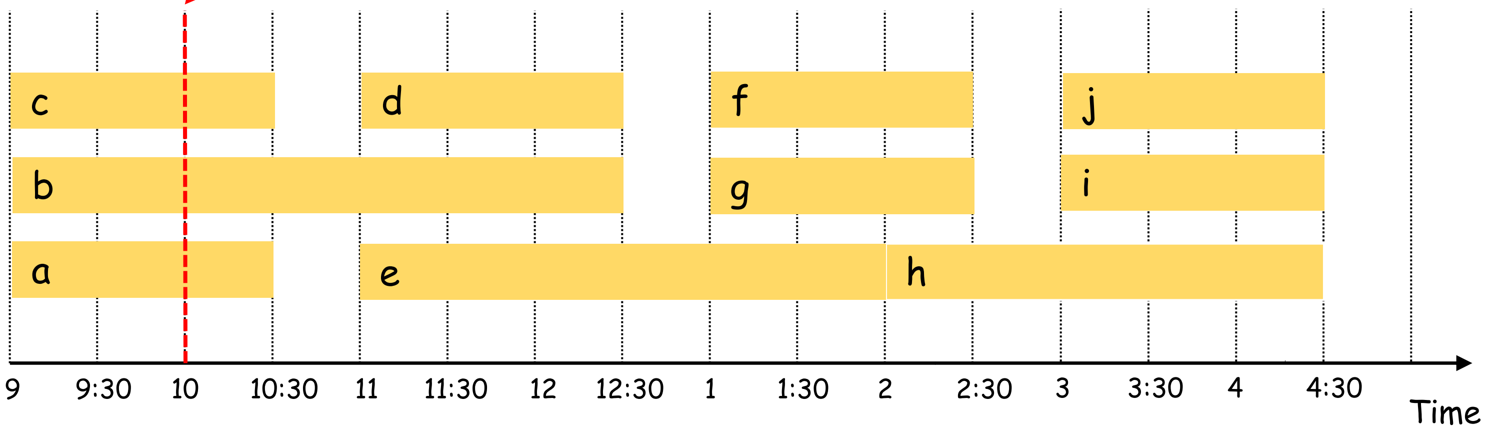
Scheduling All Intervals: Interval Partitioning

Defn: The depth of a set of open intervals is the maximum number that contain any given time.

Key observation: # of rooms needed $\geq$ depth.

Example: This schedule uses only **3** rooms. Since depth $\geq$ **3** this is optimal.

depth $\geq$ **3**



A simple greedy algorithm

Sort requests in increasing order of start times $(s_1, f_1), \ldots, (s_n, f_n)$

$last_1 = 0$ // finish time of last request currently scheduled in room 1

for $i = 1$ to n {

$j = 1$

while (request i not scheduled) {

if $s_i \geq last_j$ then

schedule request i in room j

$last_j = f_i$

$j = j + 1$

if $last_j$ undefined then $last_j = 0$

}

}

Look for the first room where the request will fit, opening a new room if all the others used so far are full.

Interval Partitioning: Greedy Analysis

Observation: Greedy algorithm never schedules two incompatible lectures in the same room

- Only schedules request $\boldsymbol{i}$ in room $\boldsymbol{j}$ if $\boldsymbol{s_i \geq last_j}$

Theorem: Greedy algorithm is optimal.

Proof:

Let $\boldsymbol{d}$ = number of rooms that the greedy algorithm allocates.

- Room $\boldsymbol{d}$ is allocated because we needed to schedule a request, say $\boldsymbol{j}$, that is incompatible with some request in each of the other $\boldsymbol{d-1}$ rooms.
- Since we sorted by start time, these incompatibilities are caused by requests that start no later than $\boldsymbol{s_j}$ and finish after $\boldsymbol{s_j}$.

So… we have $\boldsymbol{d}$ requests overlapping at time $\boldsymbol{s_j} + \varepsilon$ for some (maybe tiny) $\varepsilon > \boldsymbol{0}$.

Key observation $\Rightarrow$ all schedules use $\geq \boldsymbol{d}$ rooms.

A simple greedy algorithm

Runtime analysis

Sort requests in increasing order of start times $(s_1, f_1), \ldots, (s_n, f_n)$

$O(n \log n)$

$last_1 = 0$ // finish time of last request currently scheduled in room 1

```
for i = 1 to n {
  j = 1
  while (request i not scheduled) {
      if s_i ≥ last_j then
            schedule request i in room j
            last_j = f_i
      j = j + 1
      if last_j undefined then last_j = 0
  }
}
```

Might need to try all d rooms to schedule a request

$O(n\,d)$

d might be as big as n

Worst case $\Theta(n^2)$

A more efficient implementation: Priority queue

$O(n \log n)$

Sort requests in increasing order of start times $(s_1, f_1), \dots, (s_n, f_n)$

d = 1
schedule request 1 in room 1
$last_1$ = f_1
$O(1)$
insert 1 into priority queue Q with key = $last_1$
for i = 2 to n {
  j = findmin(Q)
  if $s_i \geq last_j$ then {
    schedule request i in room j
    $last_j$ = f_i
    increasekey(j,Q) to $last_j$ }
  else {
    d = $d + 1$
    schedule request i in room d
    $last_d$ = f_i
    insert d into priority queue Q with key = $last_d$ }
}

$O(n \log d)$

$O(\log d)$

$\Theta(n \log n)$ total

$O(\log d)$

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

- Consider an arbitrary other PB&J sandwich. Show that every ingredient I use increases the deliciousness by at least as much as the other sandwich's ingredient.

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.

- Show that the maximum deliciousness of a PB&J sandwich is 9.5/10, then show that my sandwich has a deliciousness score of 9.5.

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

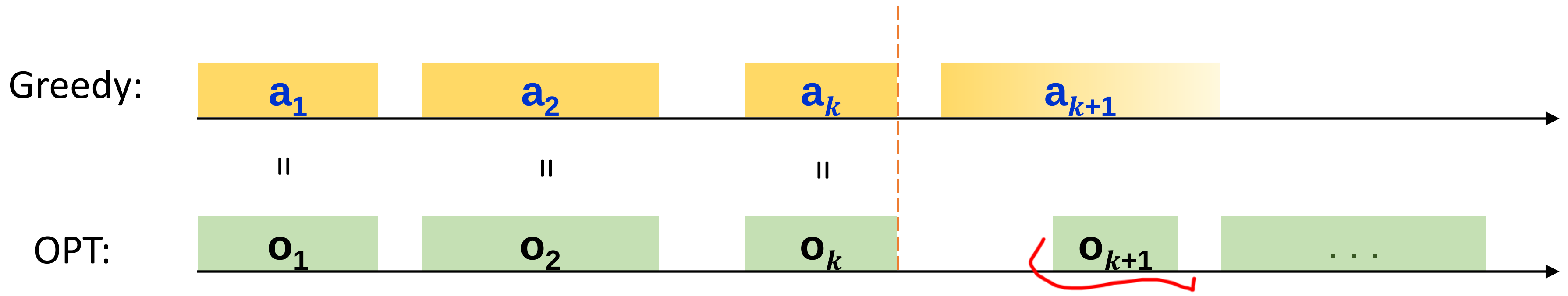
- Consider an arbitrary other PB&J sandwich. Show that, for each ingredient, swapping it out with my choice won't decrease the deliciousness.

Interval Scheduling: Analysis (Exchange form)

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

Proof:

- Let $a_1, a_2, \ldots, a_t$ denote set of jobs selected by greedy algorithm.
- Let $o_1, o_2, \ldots, o_s$ denote set of jobs in an alternative optimal solution with $a_1 = o_1, a_2 = o_2, \ldots, a_k = o_k$ (i.e. the solutions match for the first k intervals).
- We will show that exchanging out o_{k+1} in favor of a_{k+1} is also a valid schedule
  - If o_{k+1} exists then a_{k+1} must exist, since o_{k+1} is an example of an interval compatible with all of $a_1, \ldots, a_k$.

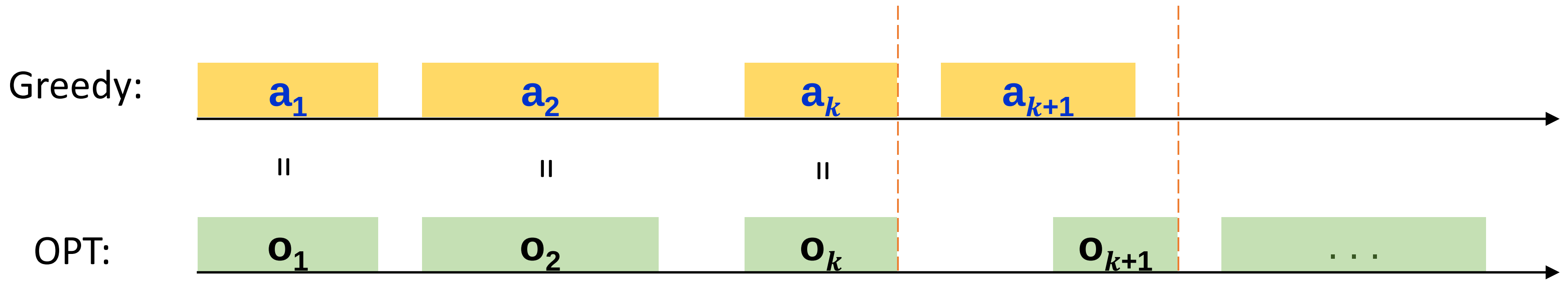


Interval Scheduling: Analysis (Exchange form)

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

Proof:

- Let $a_1, a_2, \dots, a_t$ denote set of jobs selected by greedy algorithm.
- Let $o_1, o_2, \dots, o_s$ denote set of jobs in an alternative optimal solution with $a_1 = o_1, a_2 = o_2, \dots, a_k = o_k$ (i.e. the solutions match for the first k intervals).
- We will show that exchanging out o_{k+1} in favor of a_{k+1} is also a valid schedule
  - If o_{k+1} exists then a_{k+1} must exist, since o_{k+1} is an example of an interval compatible with all of $a_1, \dots, a_k$.
  - If $a_{k+1} \neq o_{k+1}$ then a_{k+1} the finish time of a_{k+1} is less than or equal to that of o_{k+1}

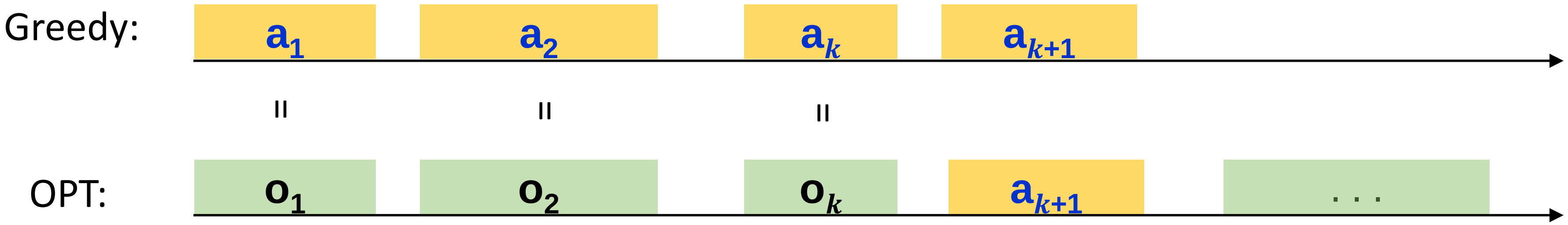


Interval Scheduling: Analysis (Exchange form)

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

Proof:

- Let $a_1, a_2, \dots, a_t$ denote set of jobs selected by greedy algorithm.
- Let $o_1, o_2, \dots, o_s$ denote set of jobs in an alternative optimal solution with $a_1 = o_1, a_2 = o_2, \dots, a_k = o_k$ (i.e. the solutions match for the first k intervals).
- We will show that exchanging out o_{k+1} in favor of a_{k+1} is also a valid schedule
    - If o_{k+1} exists then a_{k+1} must exist, since o_{k+1} is an example of an interval compatible with all of $a_1, \dots, a_k$.
    - If $a_{k+1} \neq o_{k+1}$ then a_{k+1} the finish time of a_{k+1} is less than or equal to that of o_{k+1}
    - This means a_{k+1} is also compatible with o_{k+2}, so $o_1, \dots, a_{k+1}, \dots, o_s$ is a solution that matches greedy for the first $k+1$ intervals.



Scheduling to Minimize Lateness

Scheduling to minimize lateness:

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

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

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

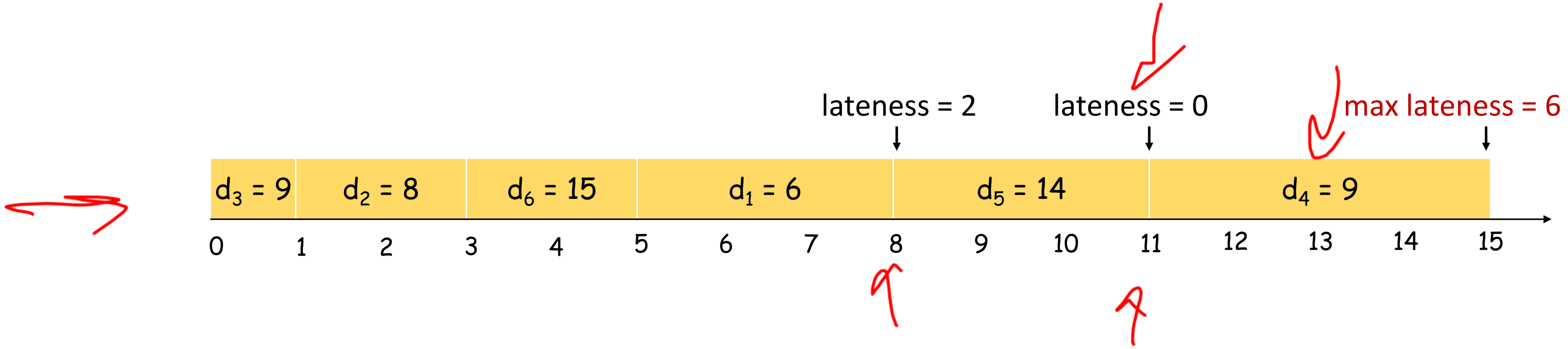
Goal: Find a schedule for **all** requests (values of $\boldsymbol{s_i}$ and $\boldsymbol{f_i}$ for each request $\boldsymbol{i}$) to minimize the maximum lateness, $\boldsymbol{L}$.

Scheduling to Minimizing Lateness

6

- Example:

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



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 $\boldsymbol{d_j}$.

Greedy Algorithm: Earliest Deadline First

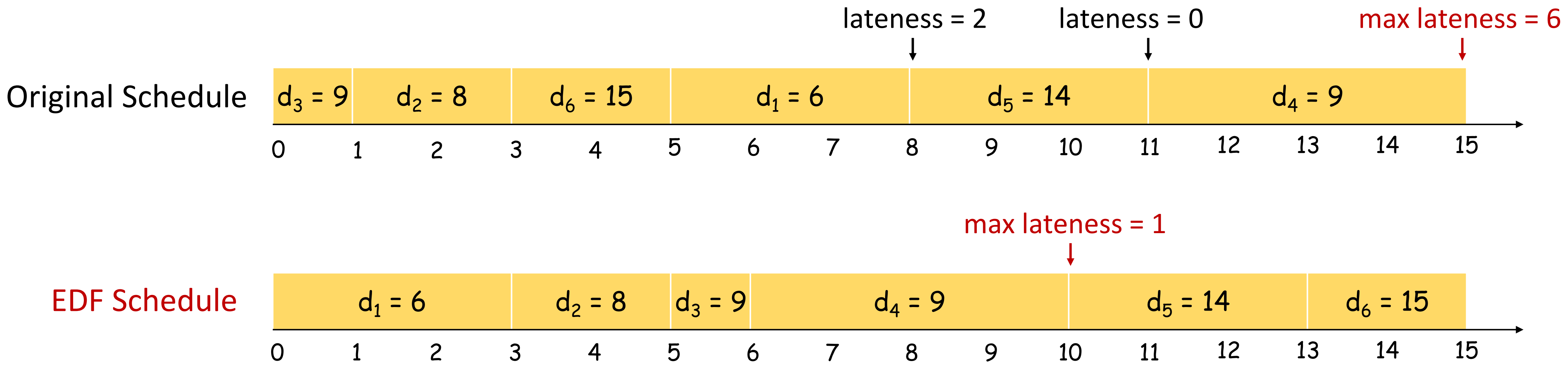
Consider requests in increasing order of deadlines

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

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

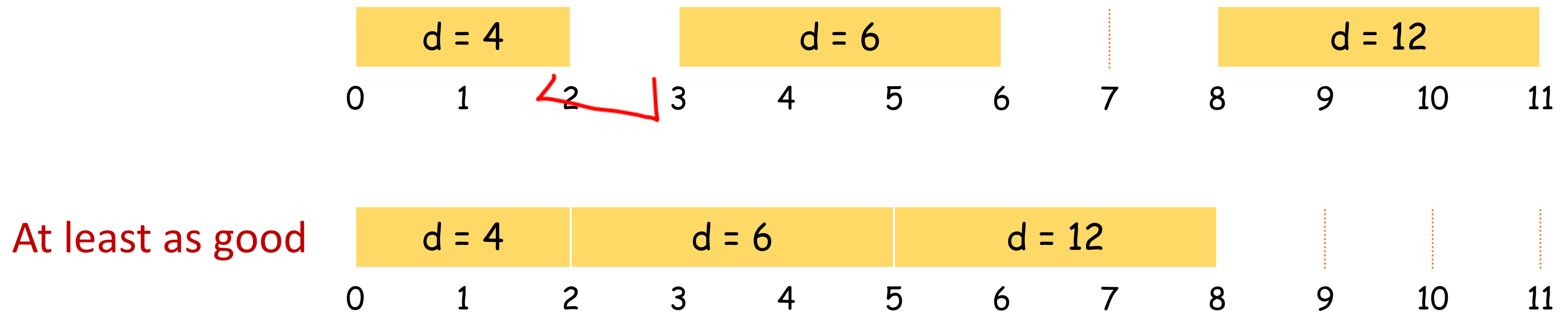
Show that if there is another schedule **O** (think optimal schedule) then we can gradually change **O** so that…

- 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

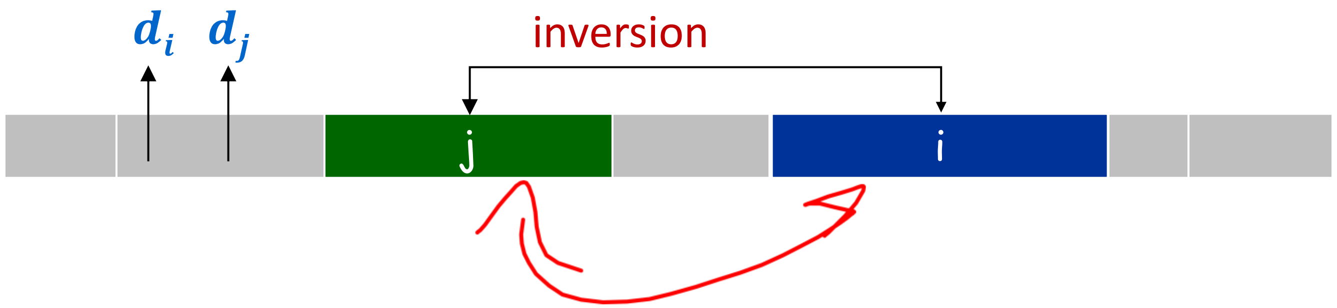
Observation: There exists an optimal schedule with no idle time



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.



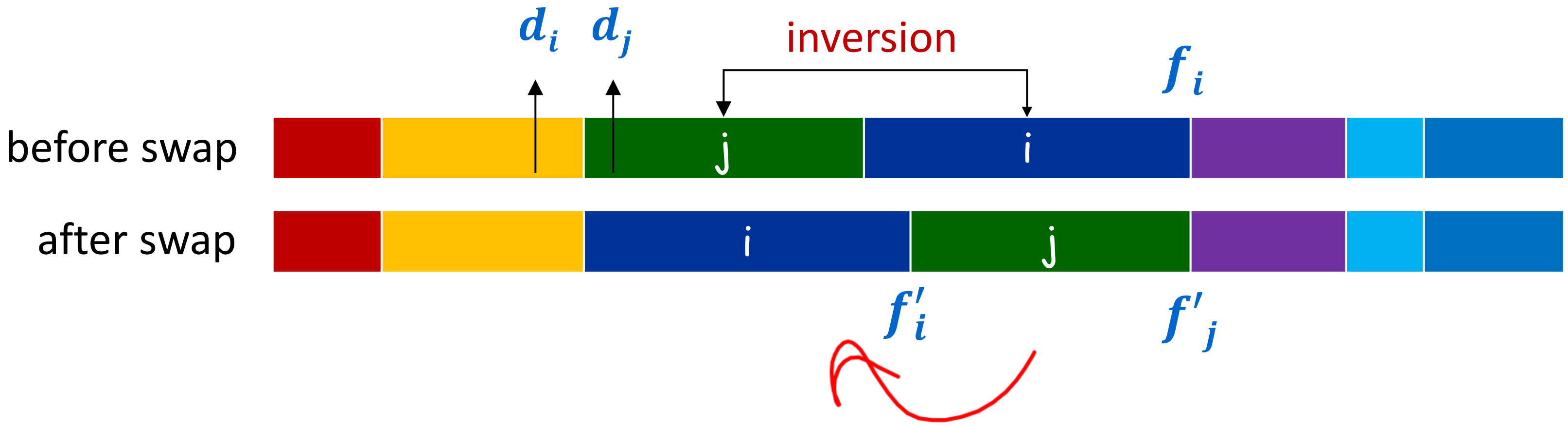
Observation: Greedy EDF schedule has no inversions.

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.

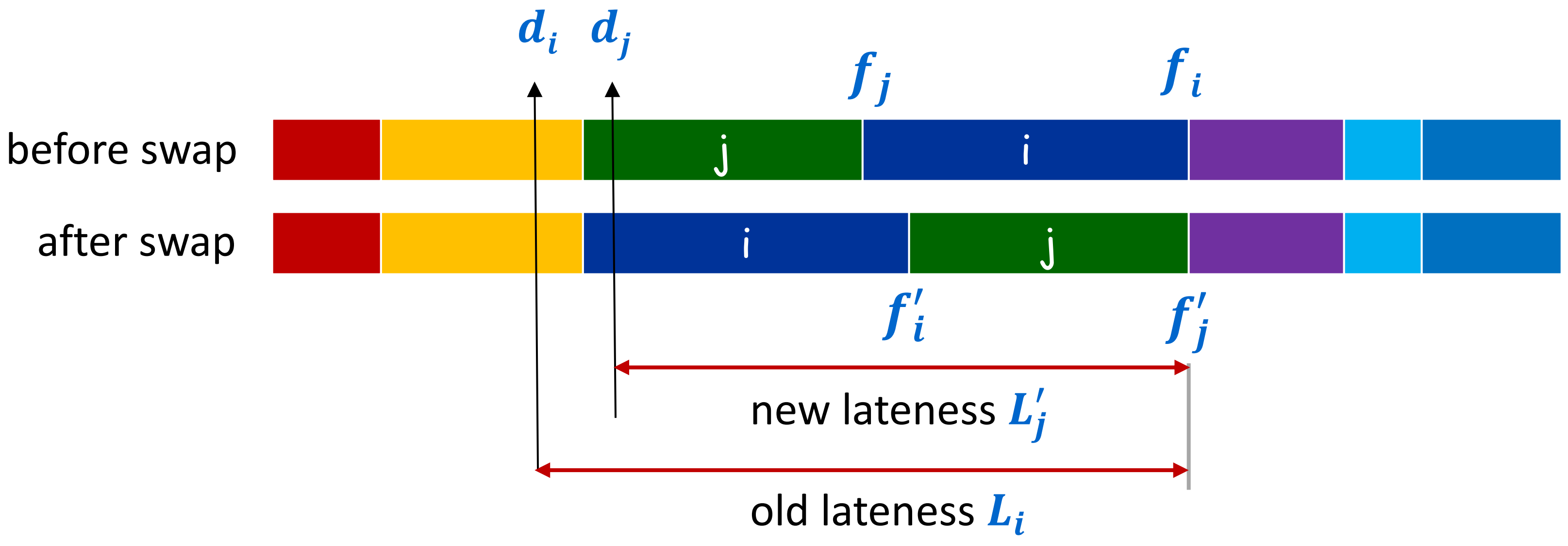


Claim: Swapping two adjacent, inverted jobs

- reduces the # of inversions by $\mathbf{1}$
- 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 **O** with no idle time

If **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 $\mathbf{1}$

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

- at most $\boldsymbol{n(n-1)/2}$ 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 $\boldsymbol{d}$.

- 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.

$\Rightarrow$ 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.

Single-source shortest paths

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

Find: (length of) shortest paths from $\boldsymbol{s}$ to each vertex in $\boldsymbol{G}$

A Greedy Algorithm

Dijkstra's Algorithm:

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

Dijkstra's Algorithm

Dijkstra(G,w,s)

S = $\{s\}$

$d[s]$ = 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 = $S \cup \{v\}$

$d[v]$ = $d[u] + w(e)$

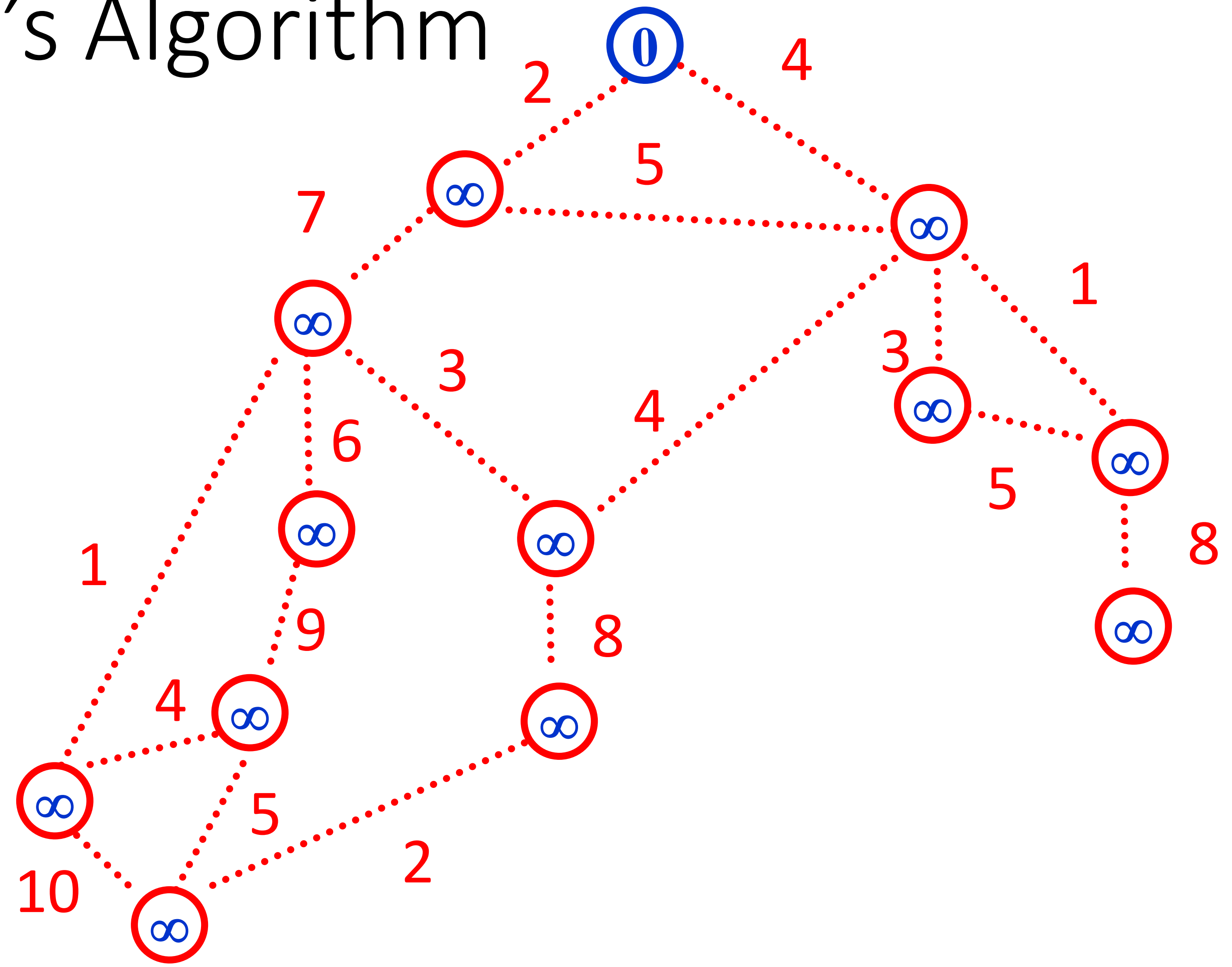
$pred[v]$=u

}

*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

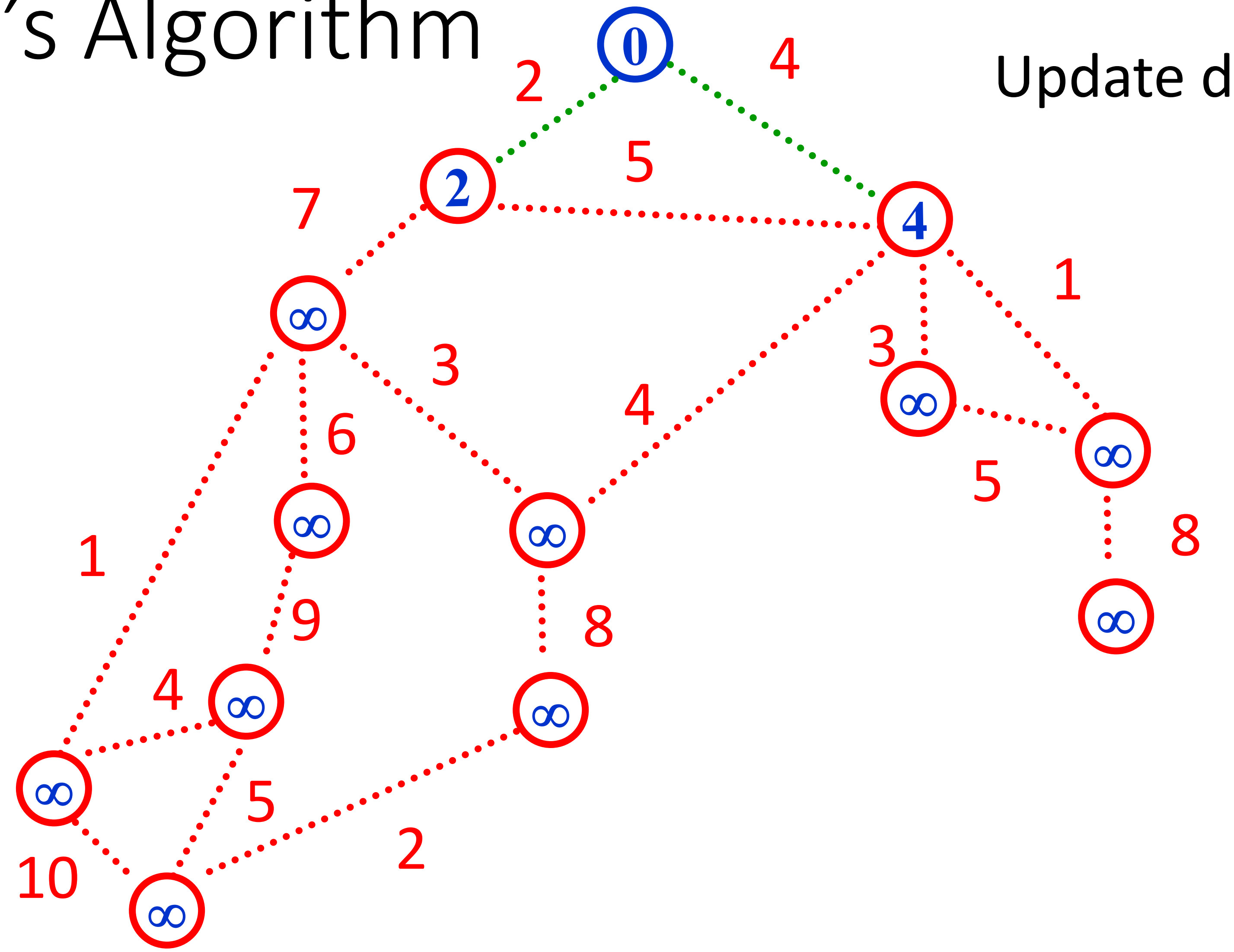
Dijkstra's Algorithm

Add to S



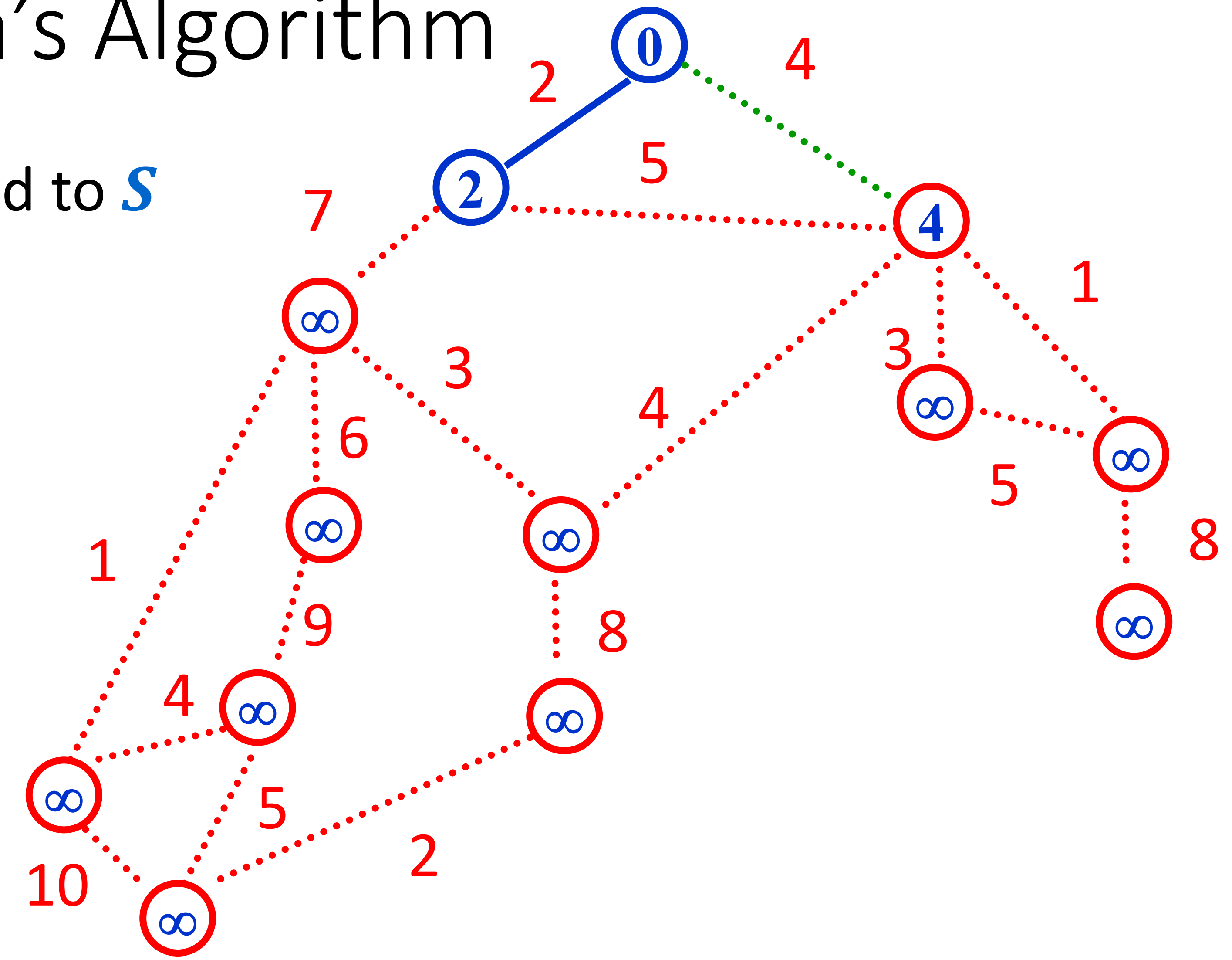
Dijkstra's Algorithm

Update distances



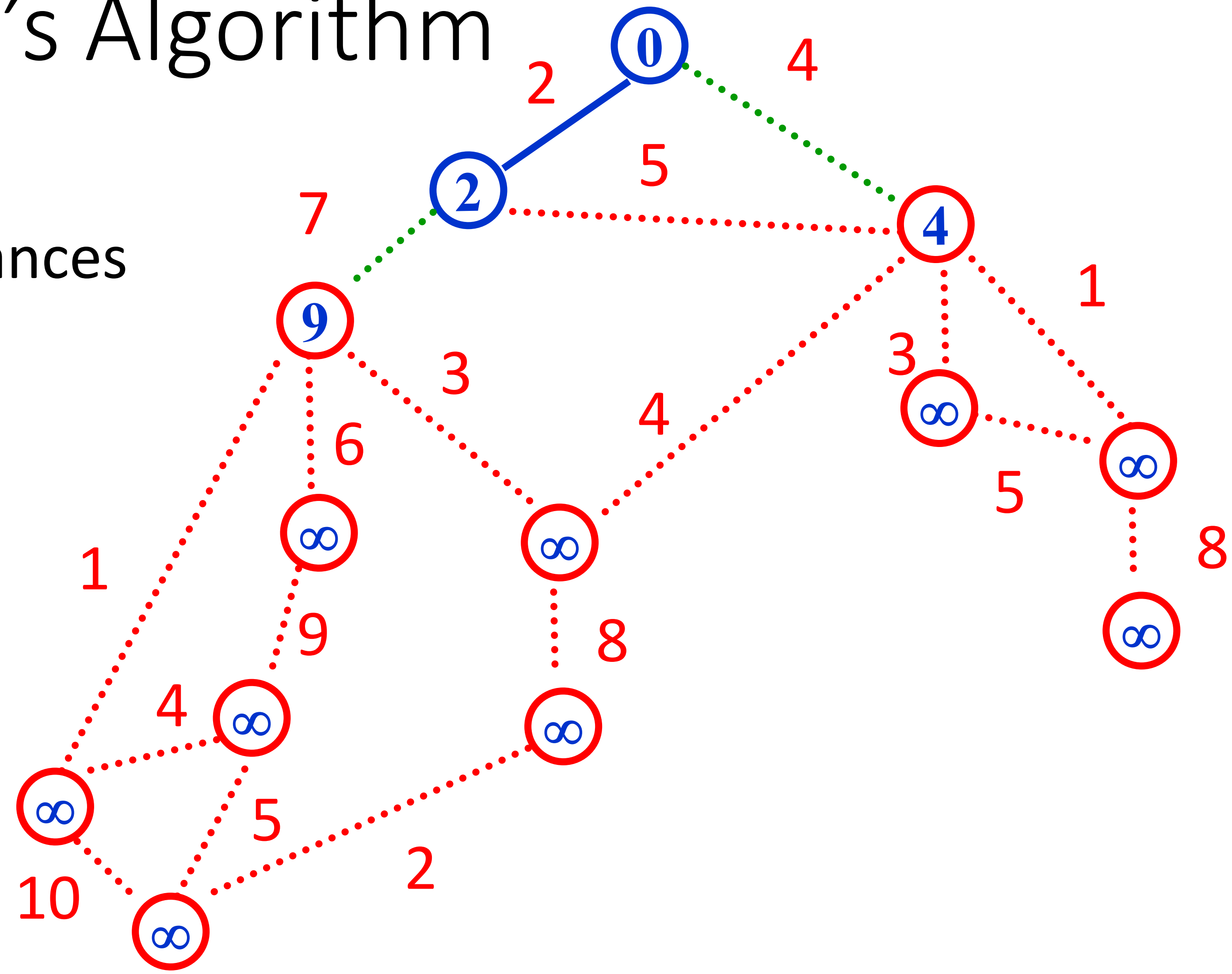
Dijkstra’s Algorithm

Add to ***S***

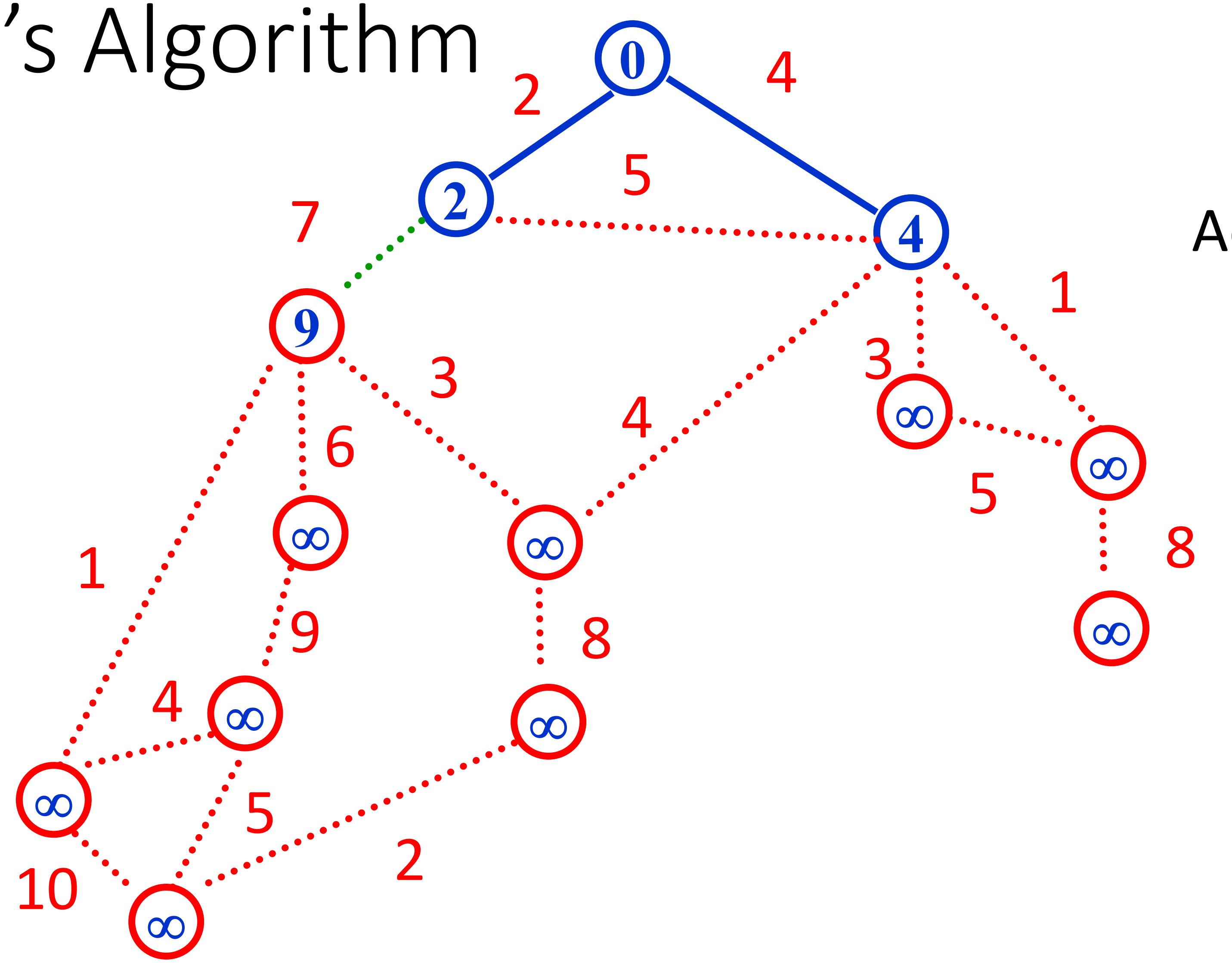


Dijkstra’s Algorithm

Update distances

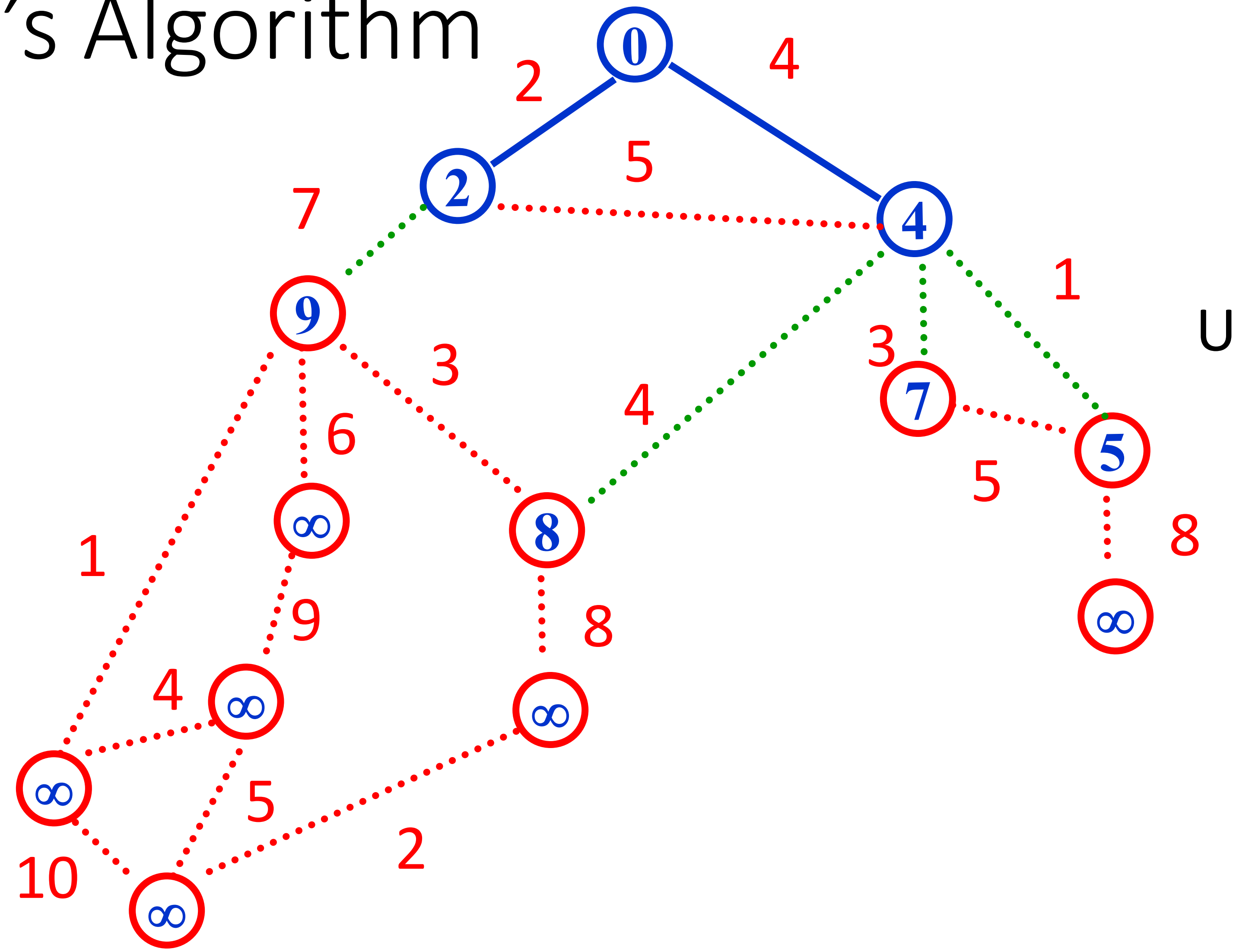


Dijkstra’s Algorithm



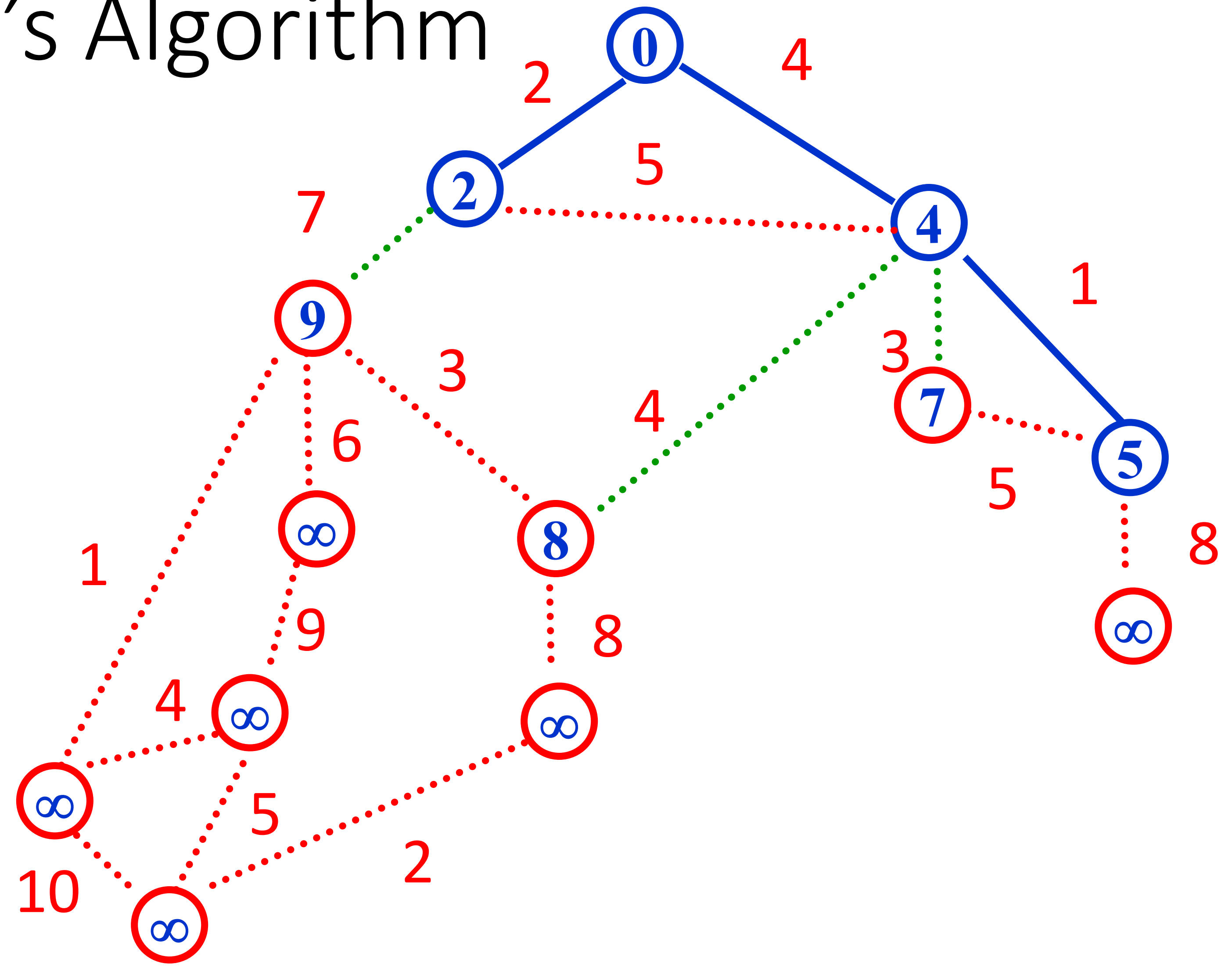
Add to S

Dijkstra’s Algorithm



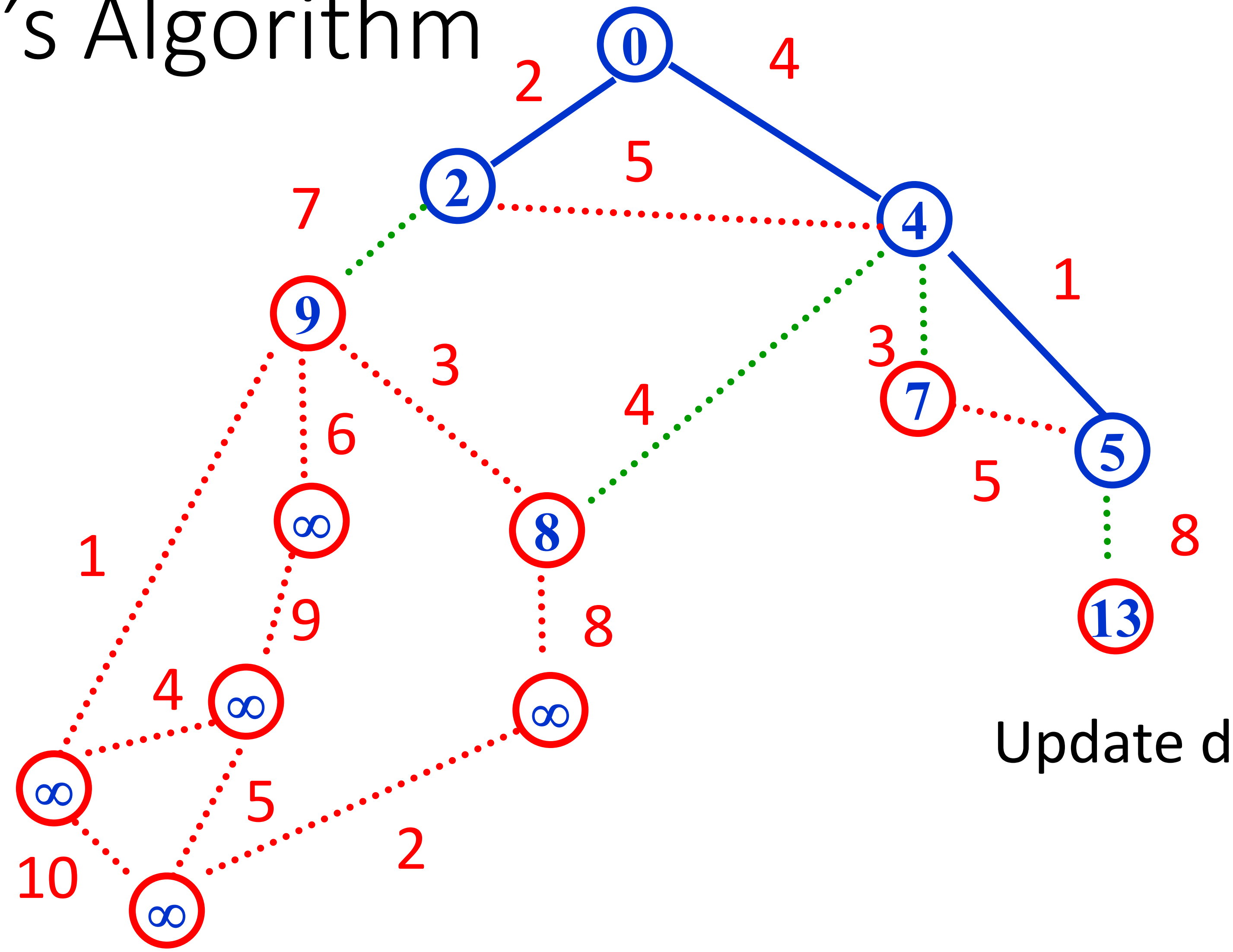
Update distances

Dijkstra’s Algorithm



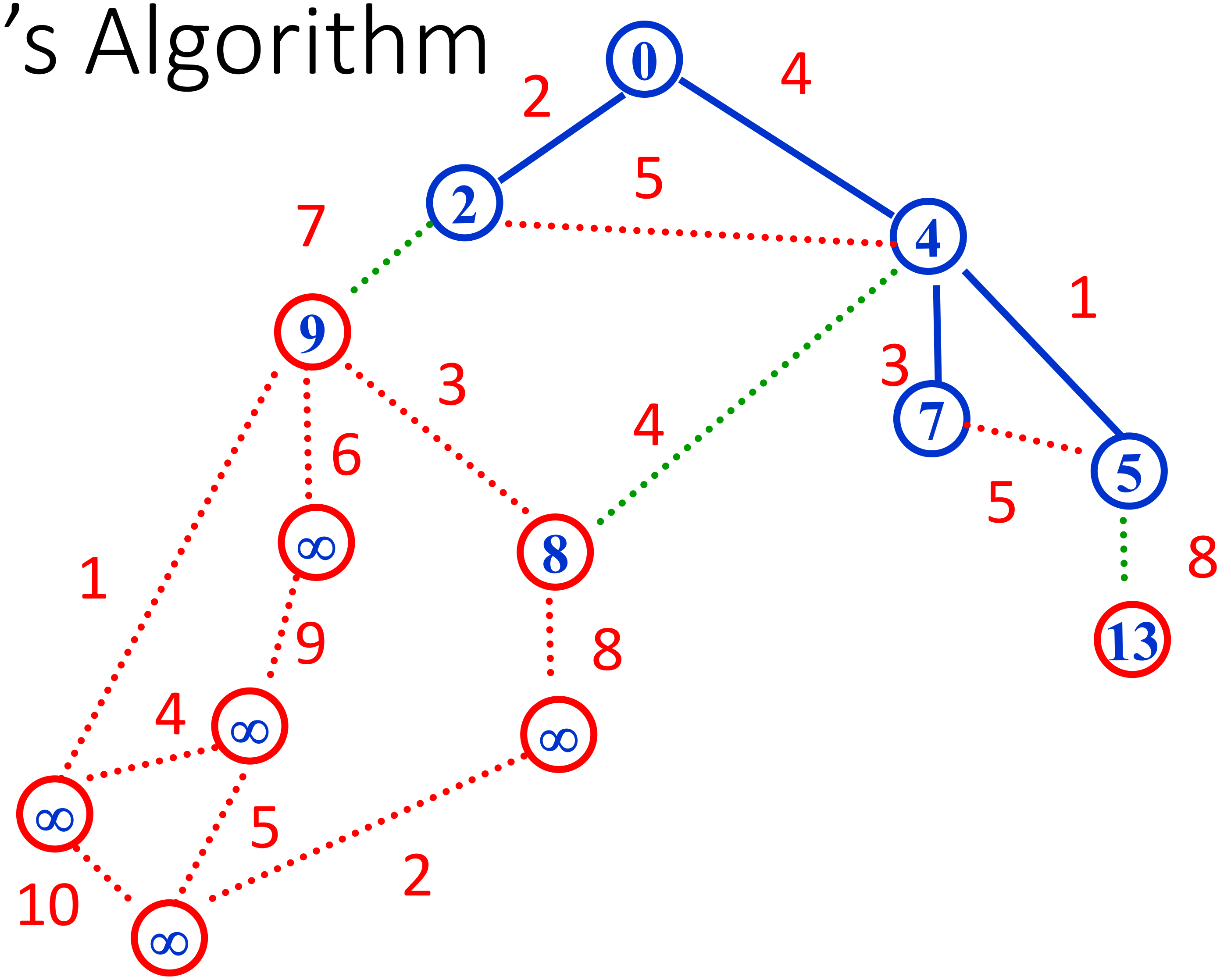
Add to S

Dijkstra's Algorithm



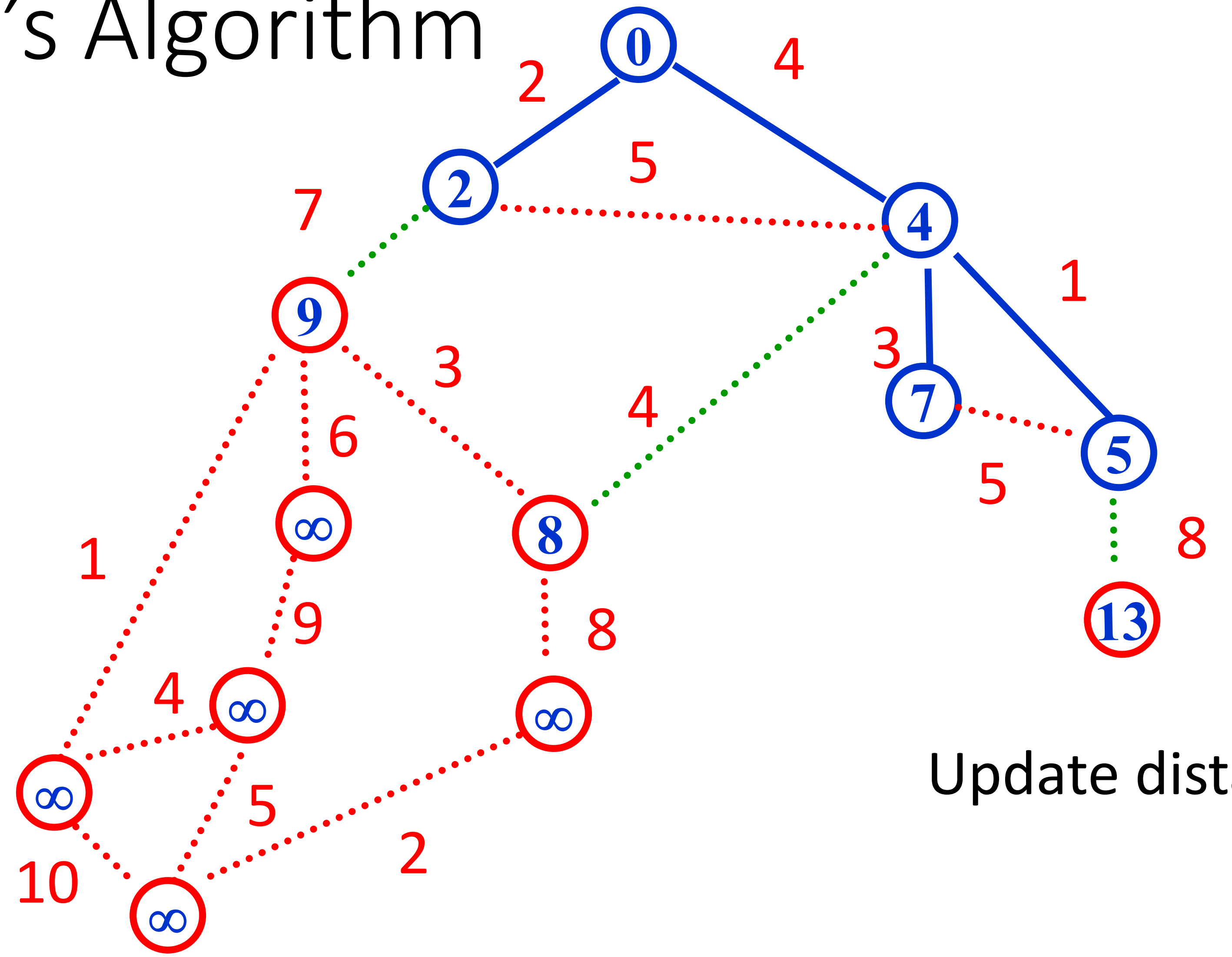
Update distances

Dijkstra's Algorithm



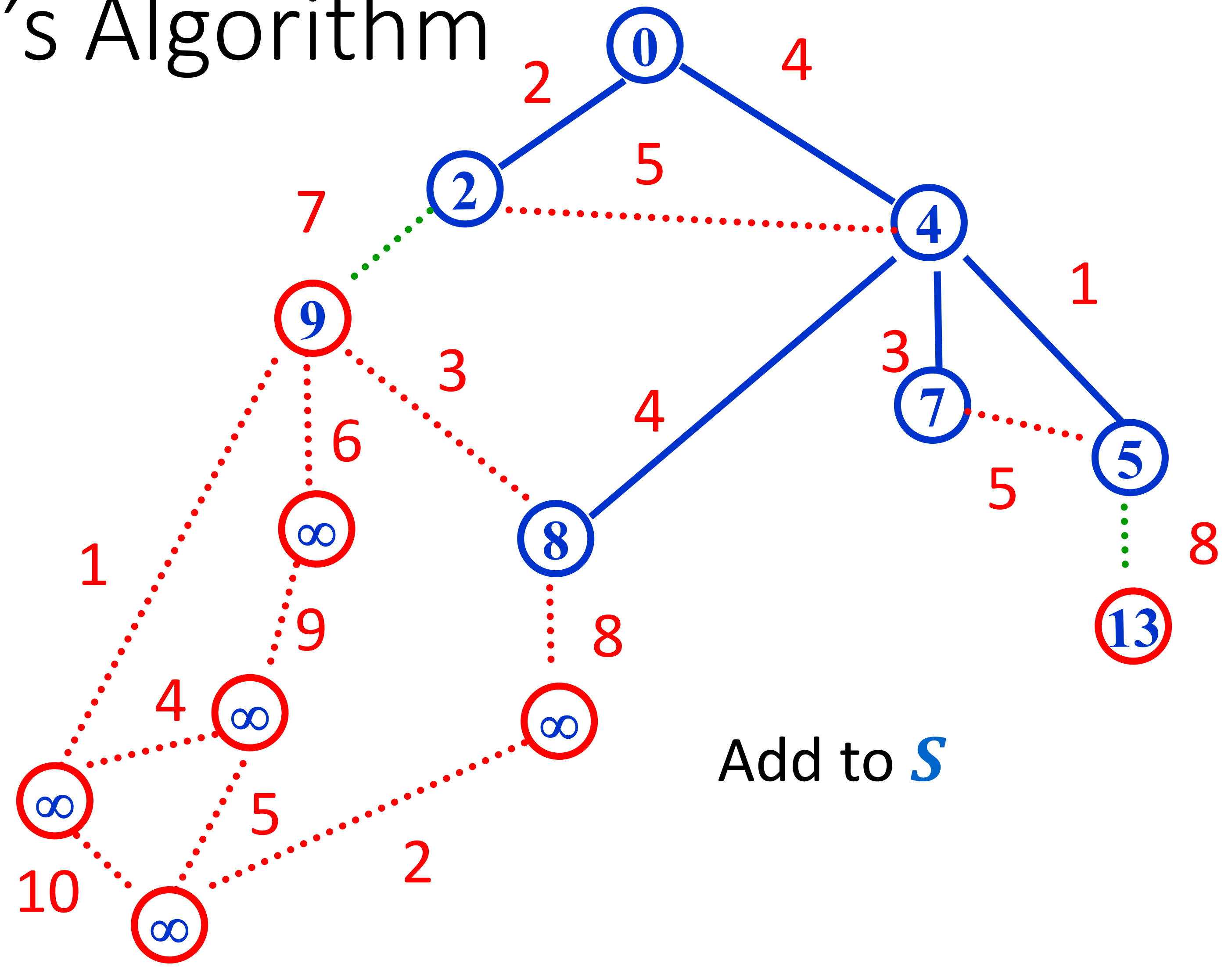
Add to **S**

Dijkstra's Algorithm



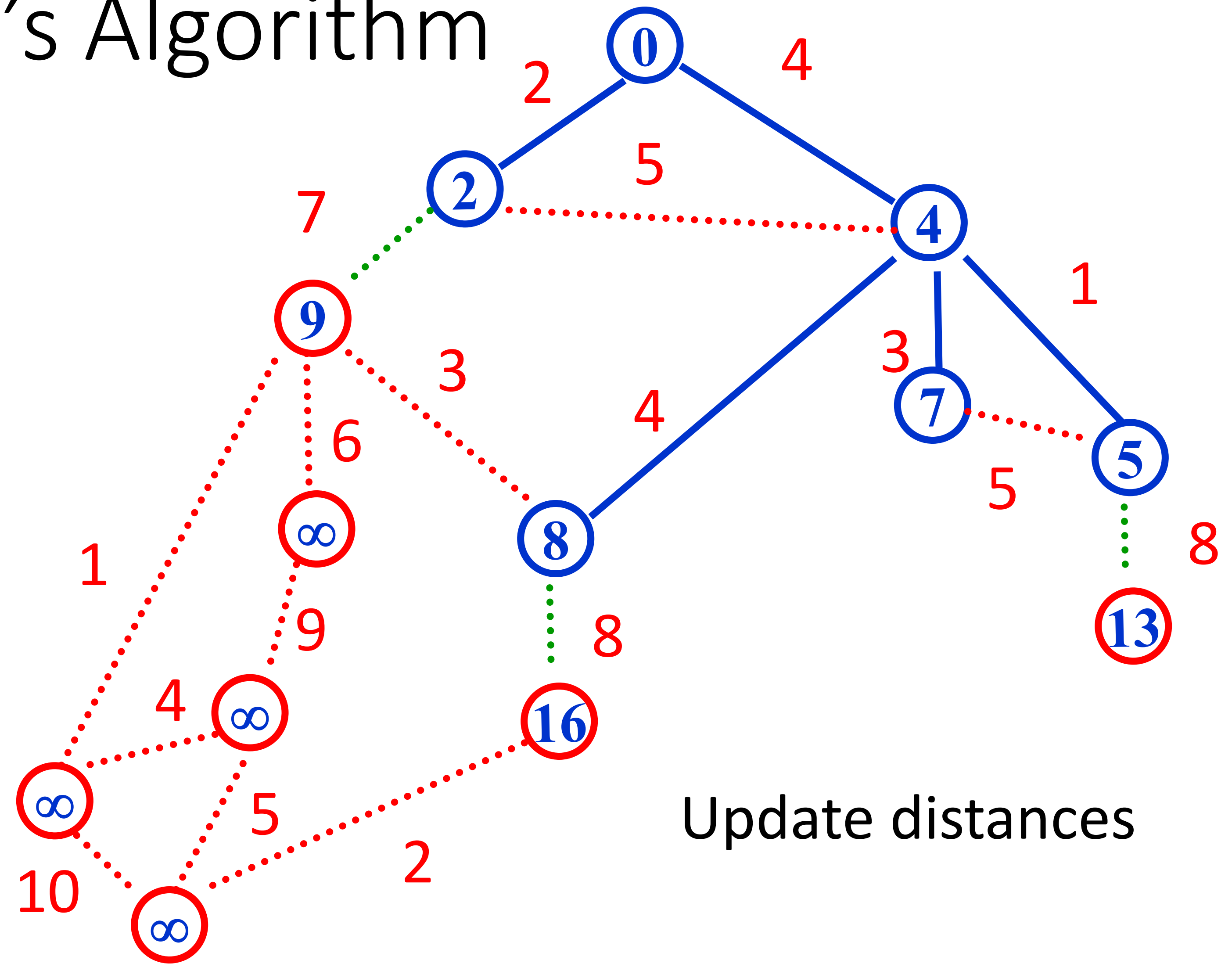
Update distances

Dijkstra's Algorithm



Add to S

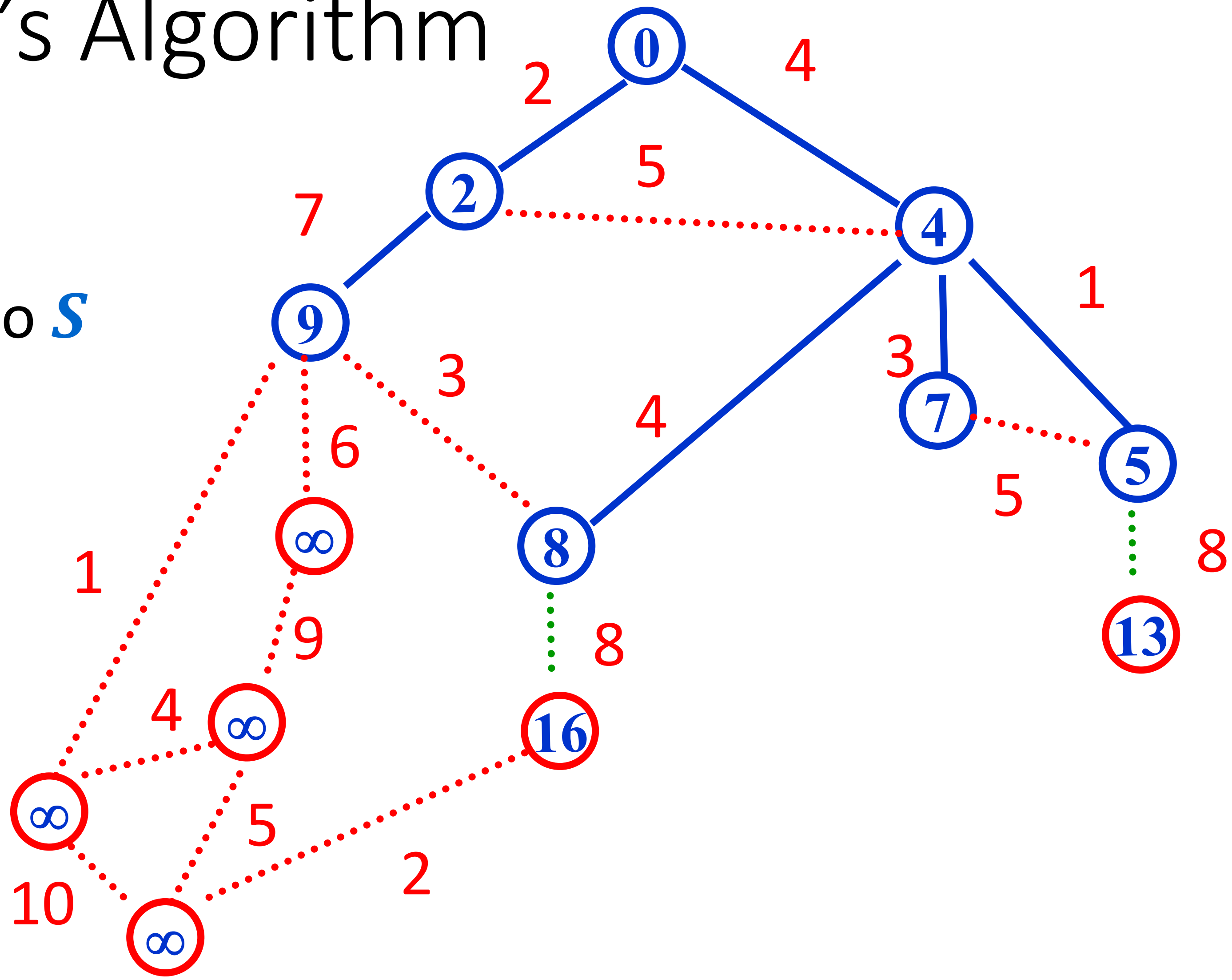
Dijkstra’s Algorithm



Update distances

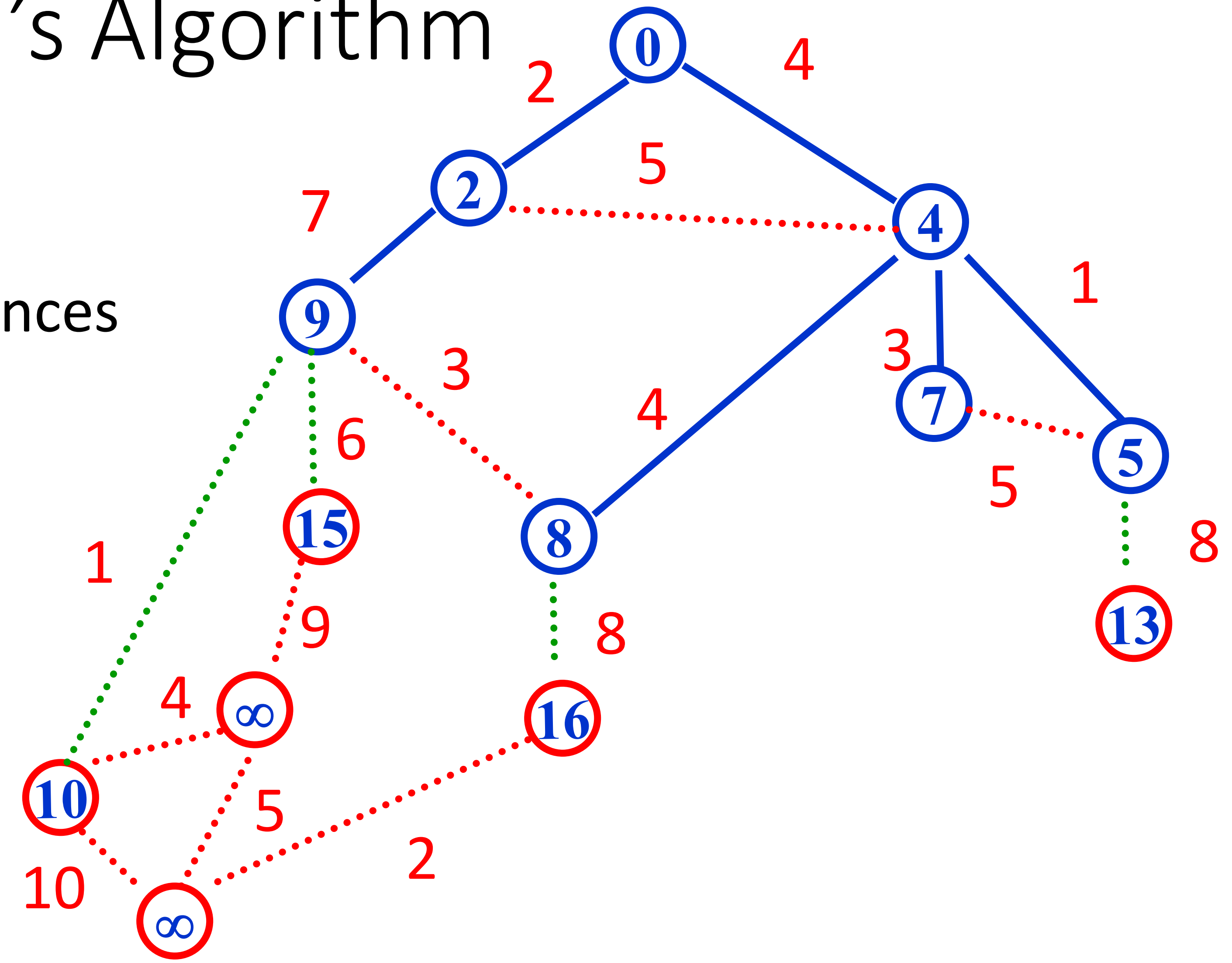
Dijkstra’s Algorithm

Add to S

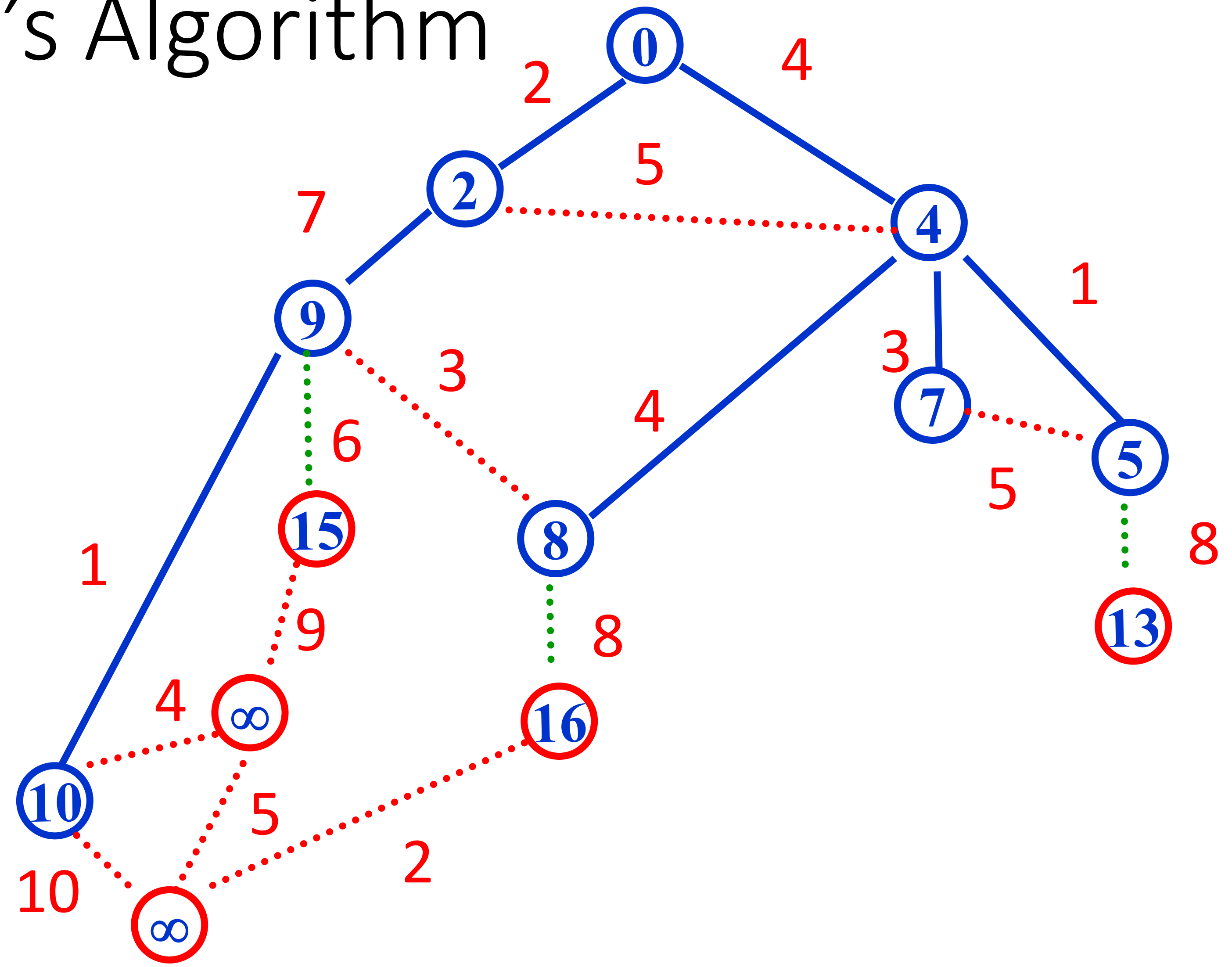


Dijkstra’s Algorithm

Update distances

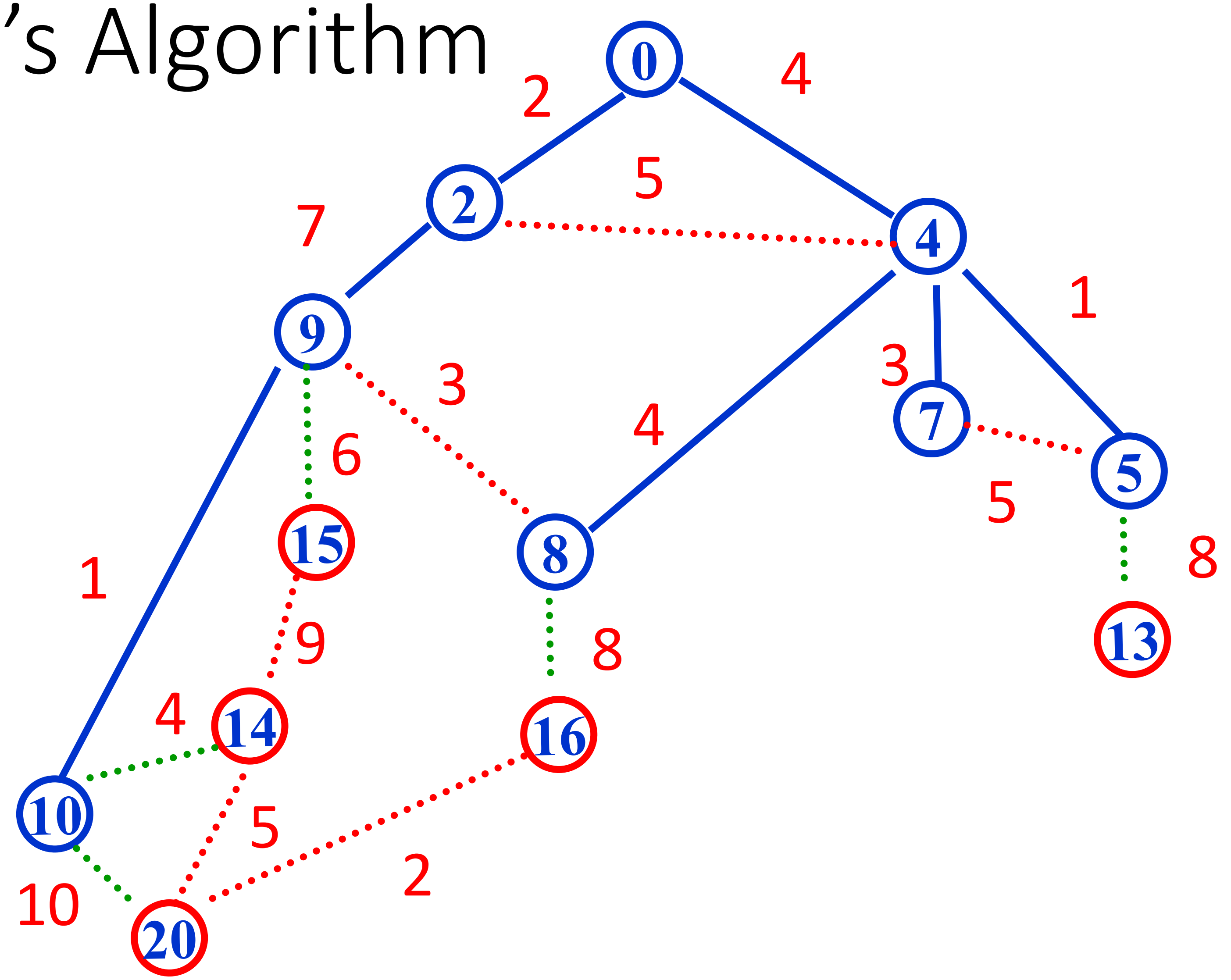


Dijkstra's Algorithm



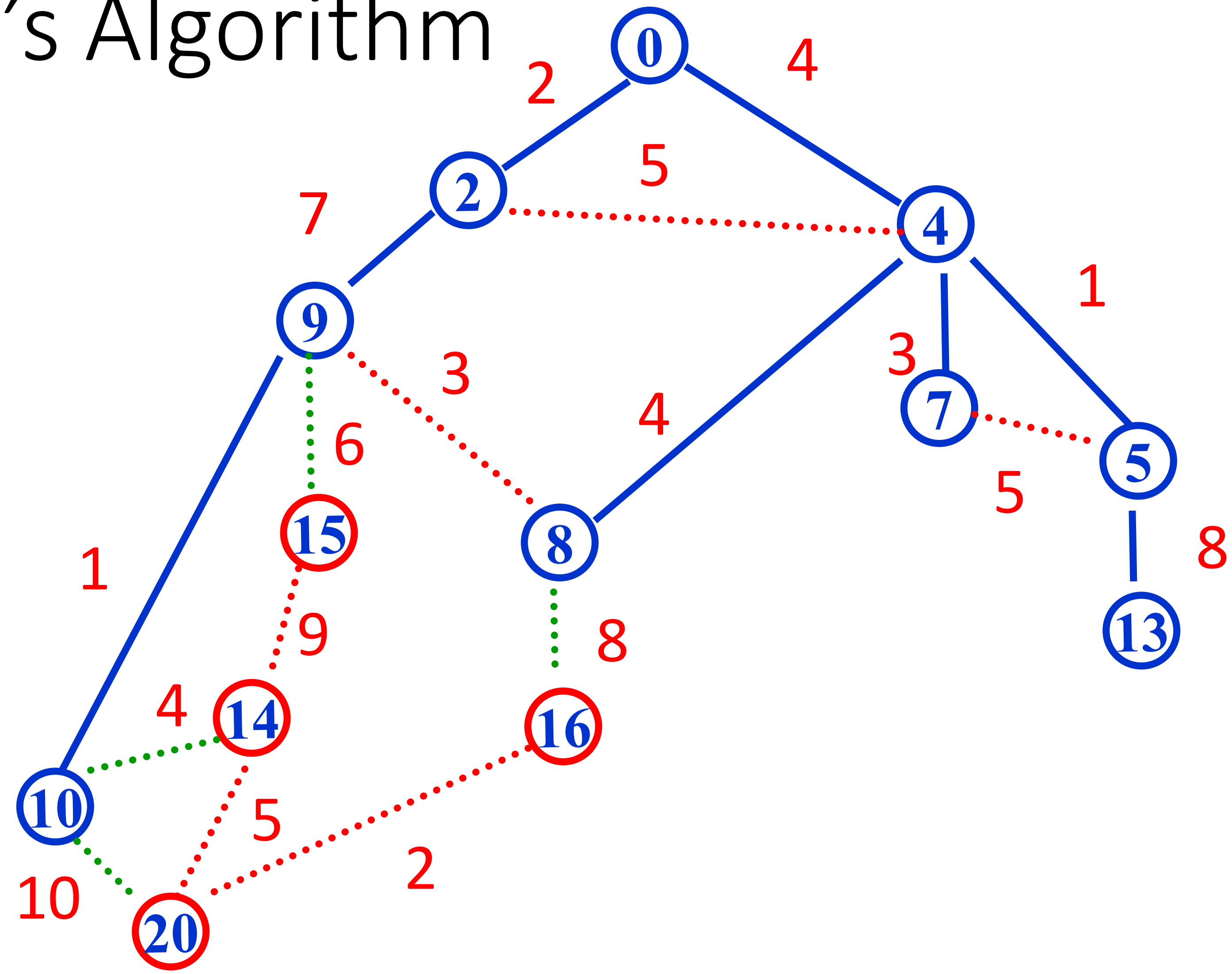
Add to ***S***

Dijkstra’s Algorithm



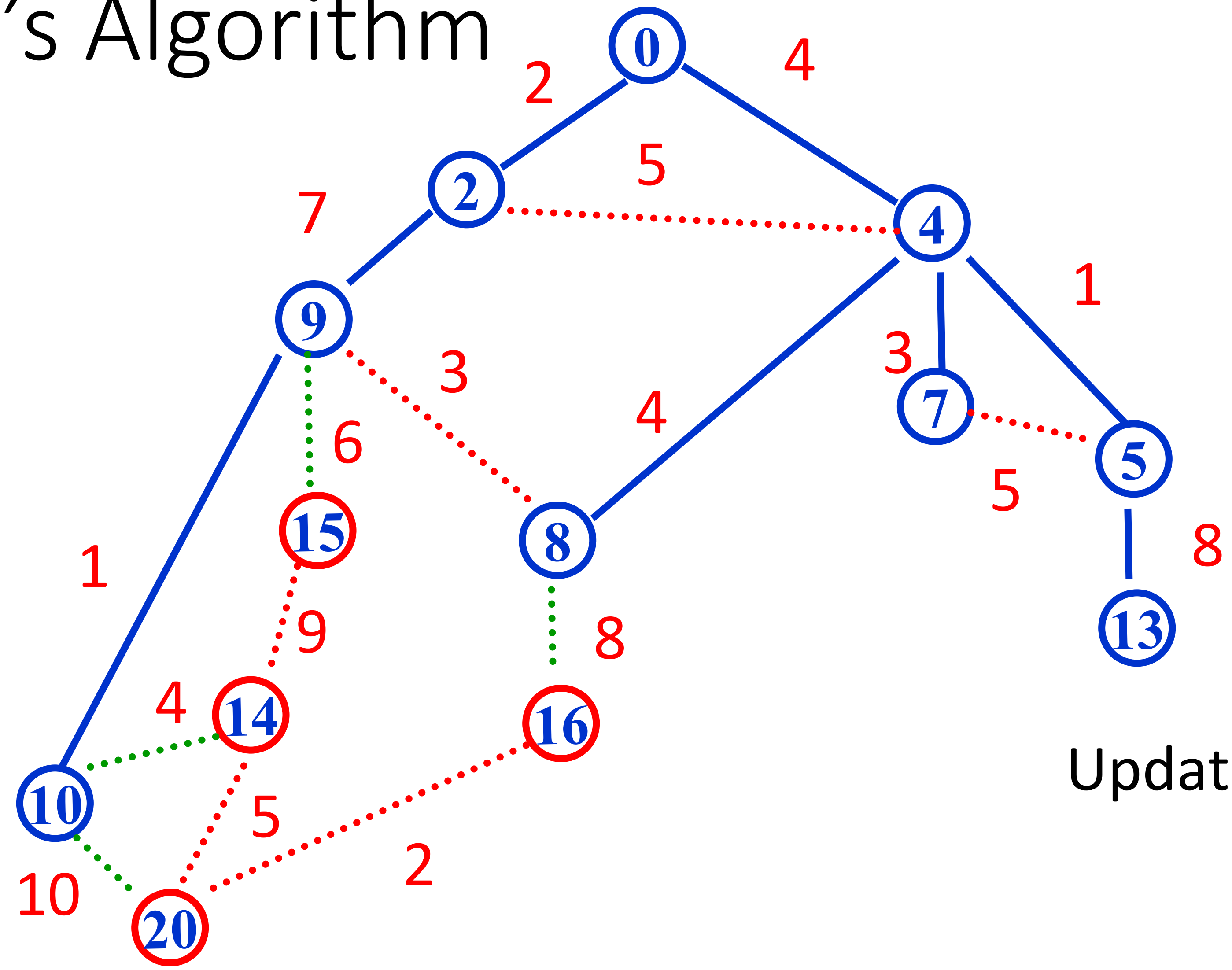
Update distances

Dijkstra’s Algorithm



Add to ***S***

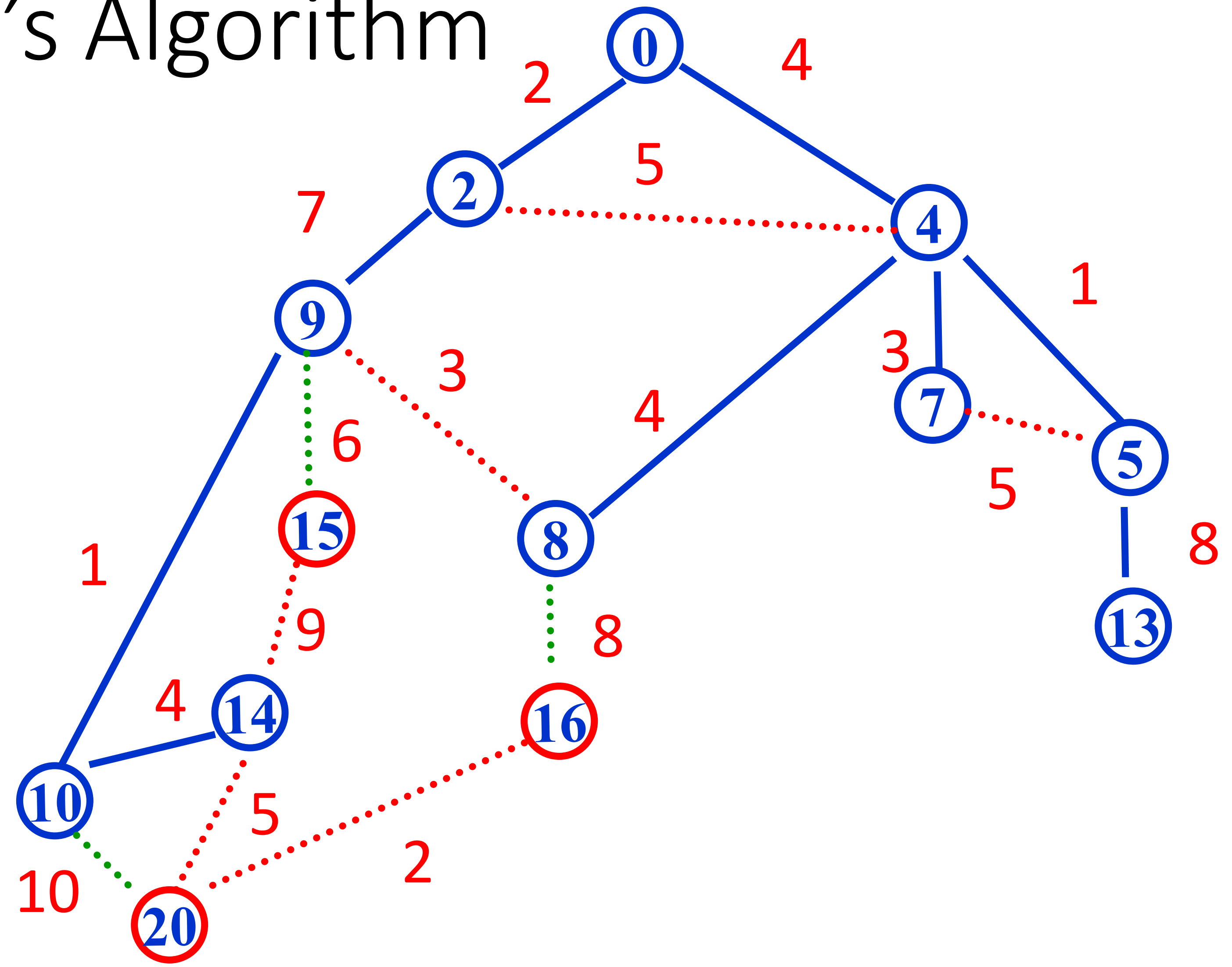
Dijkstra’s Algorithm



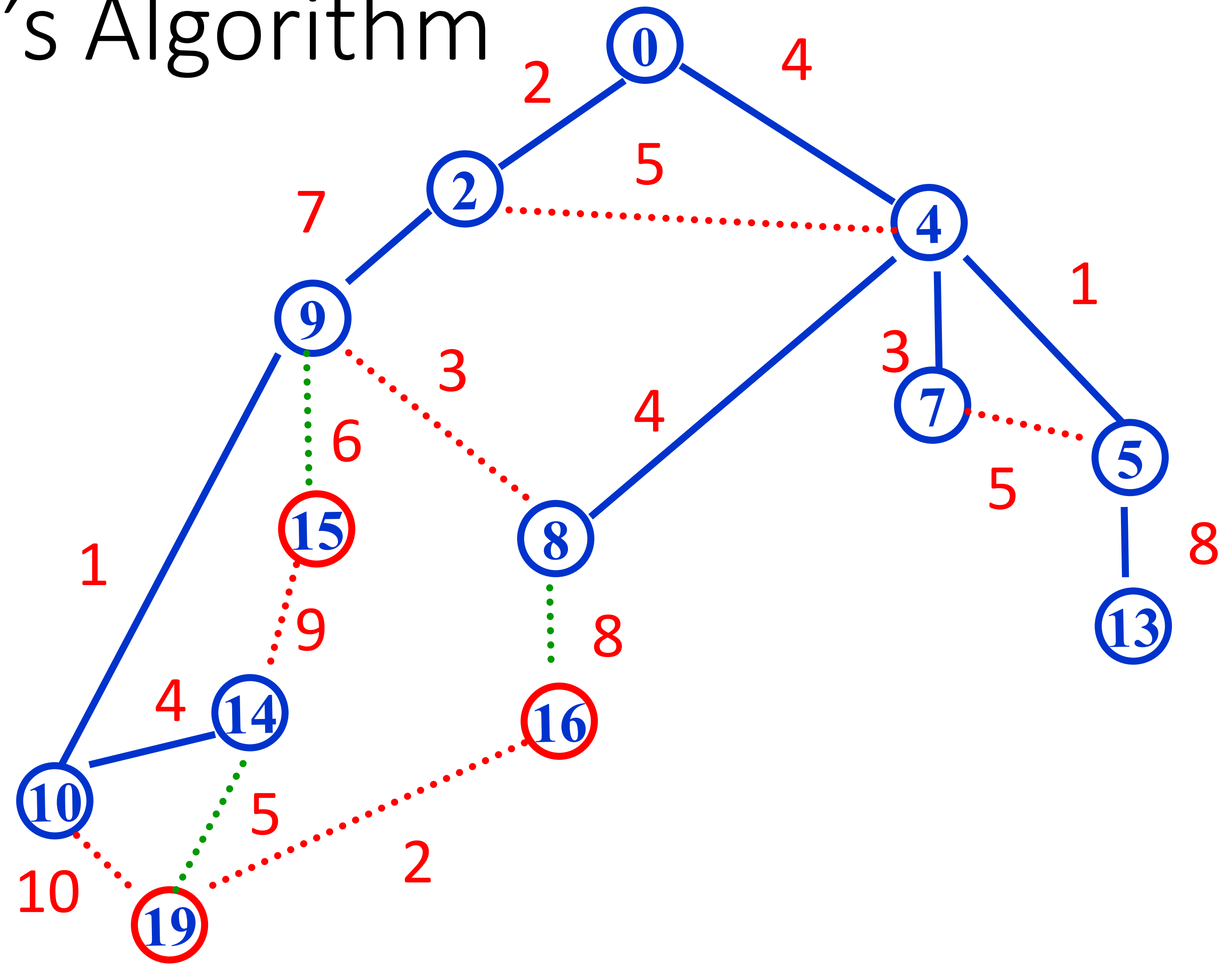
Update distances

Dijkstra's Algorithm

Add to ***S***



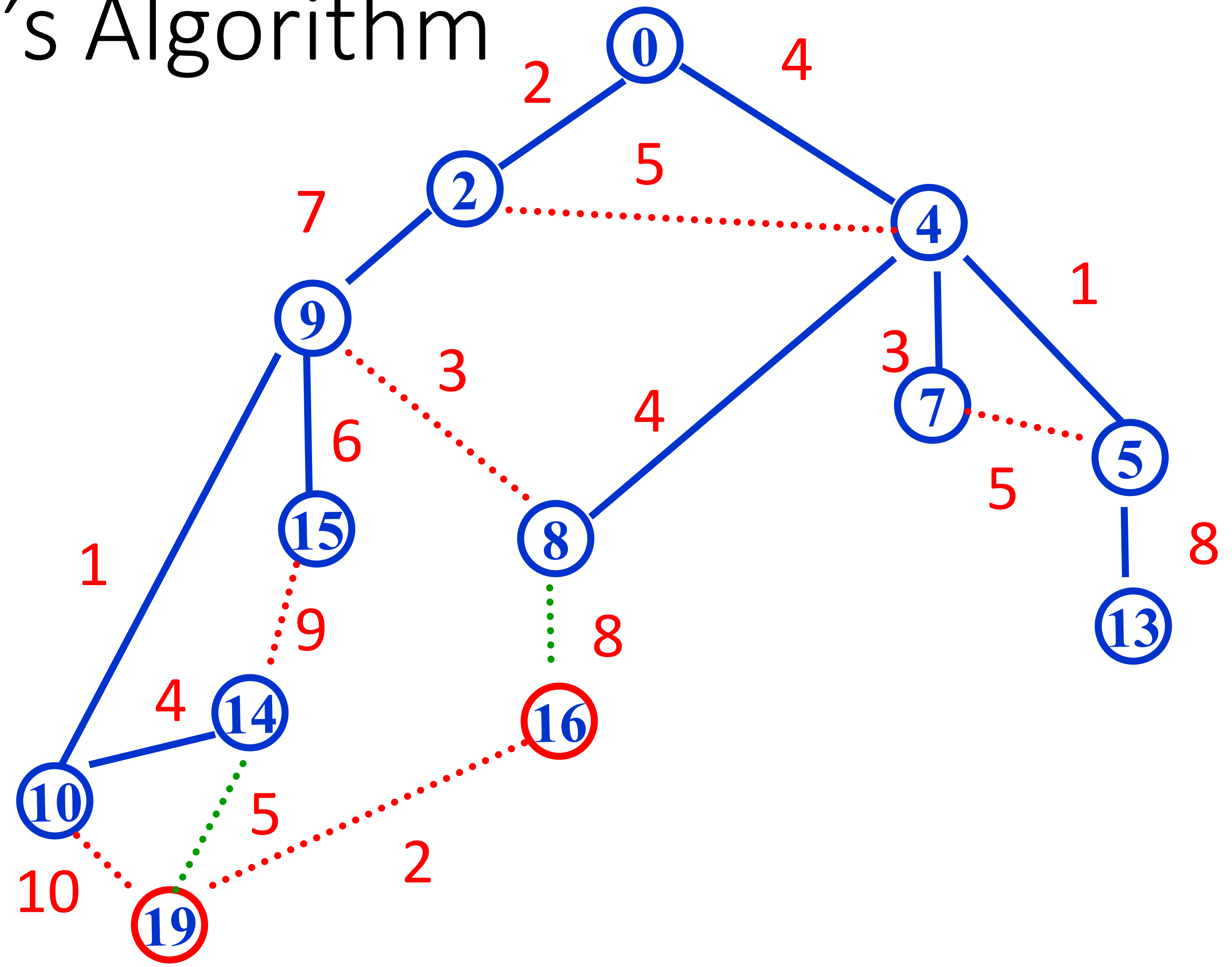
Dijkstra's Algorithm



Update distances

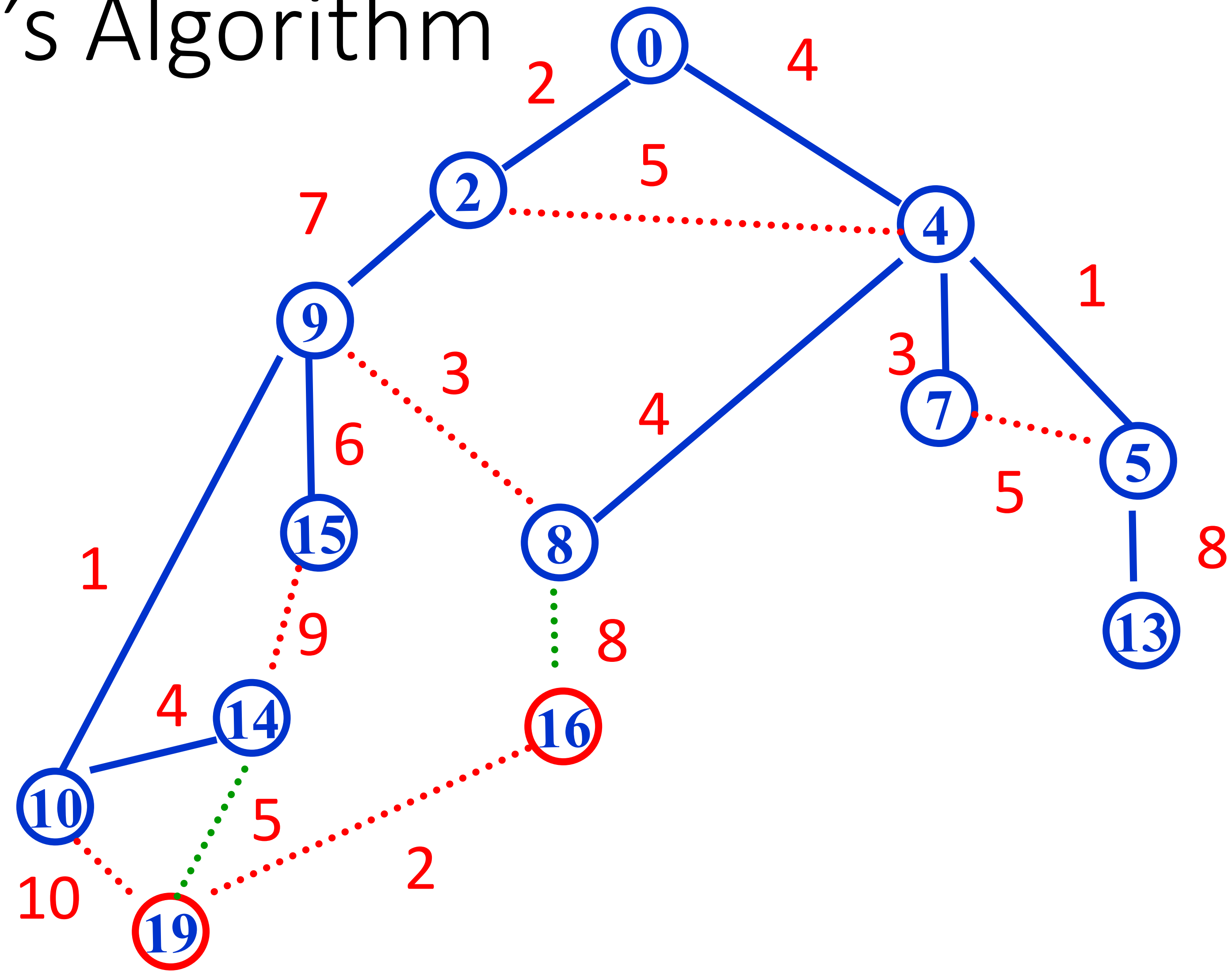
Dijkstra's Algorithm

Add to ***S***

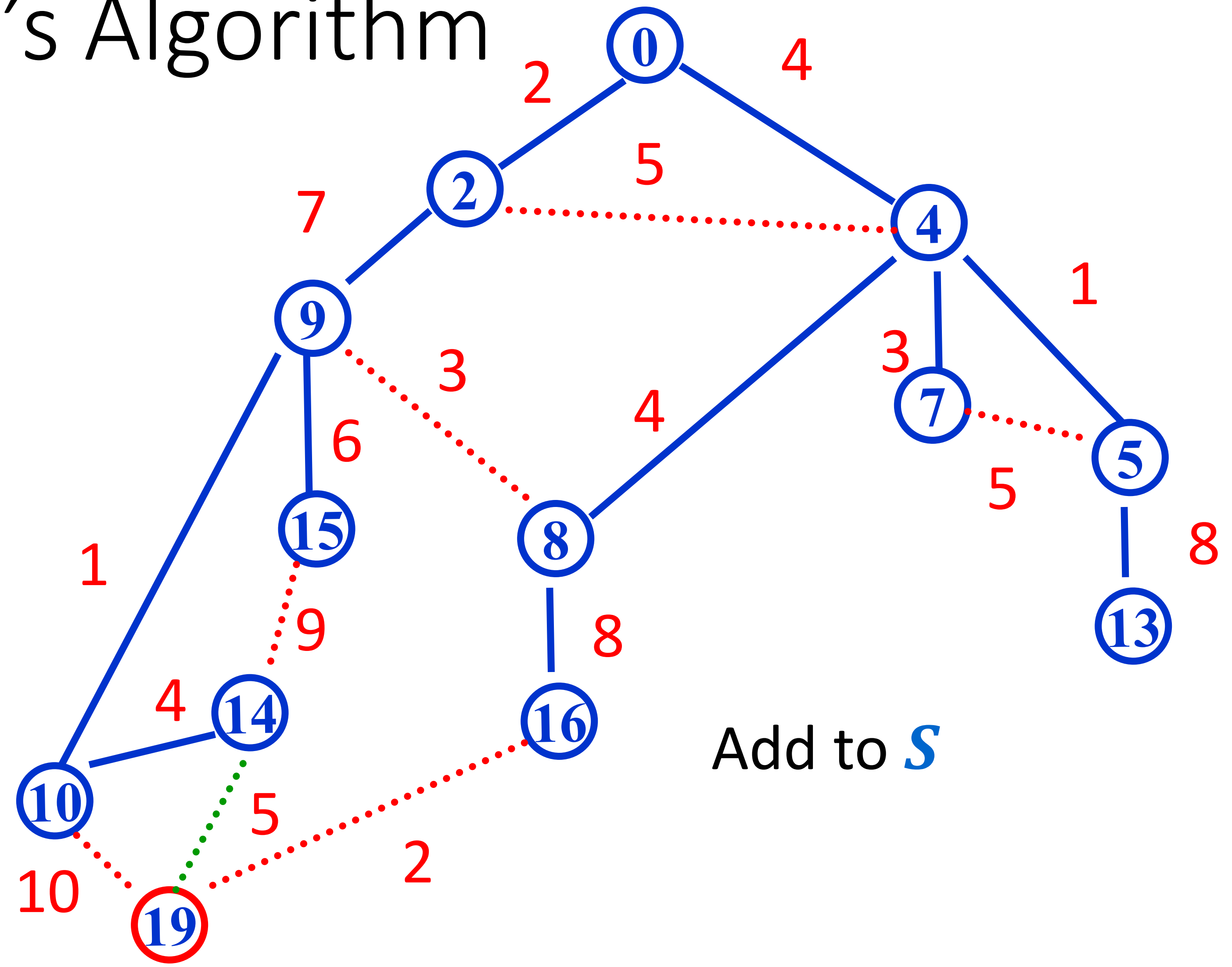


Dijkstra's Algorithm

Update distances

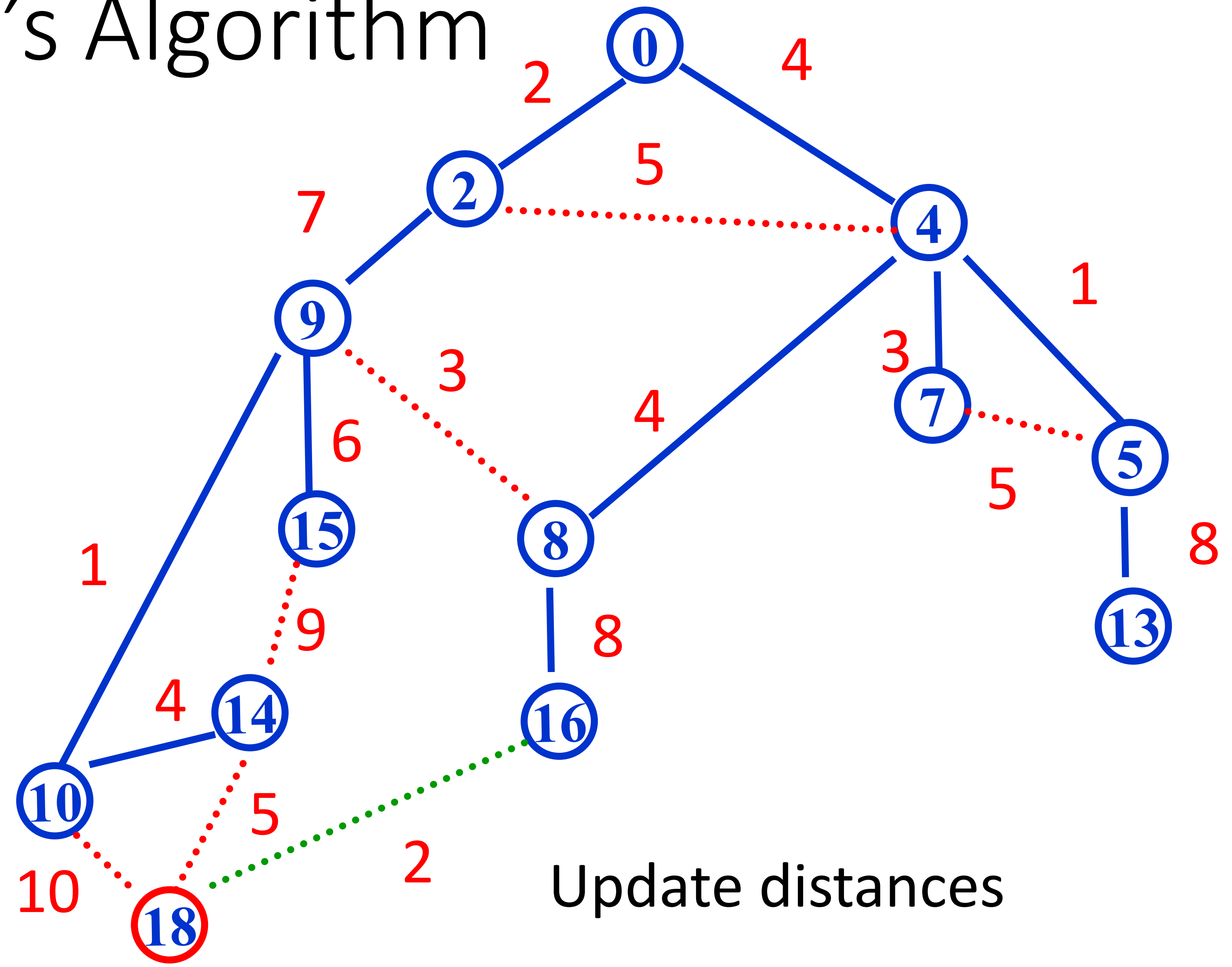


Dijkstra's Algorithm



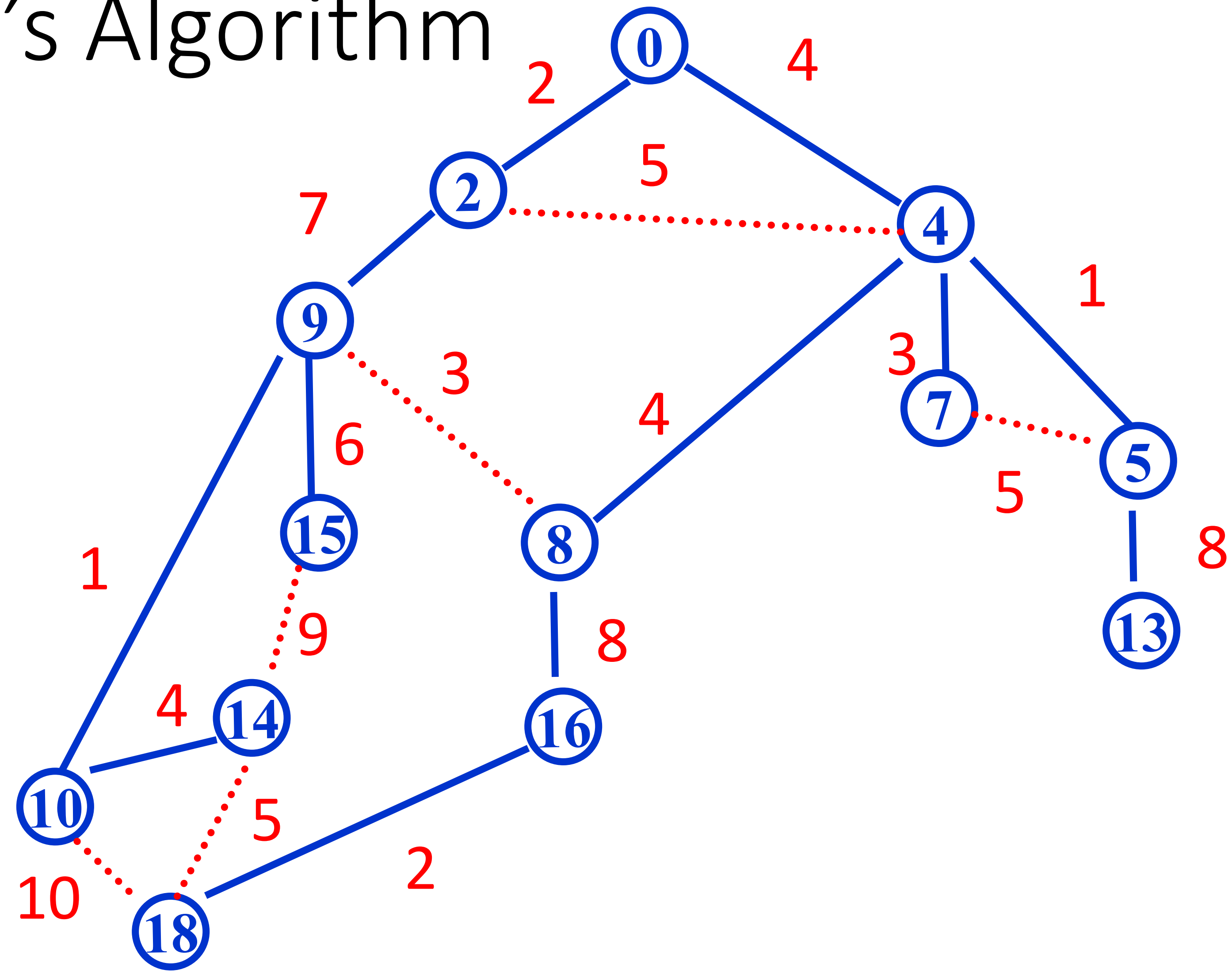
Add to ***S***

Dijkstra’s Algorithm



Update distances

Dijkstra's Algorithm

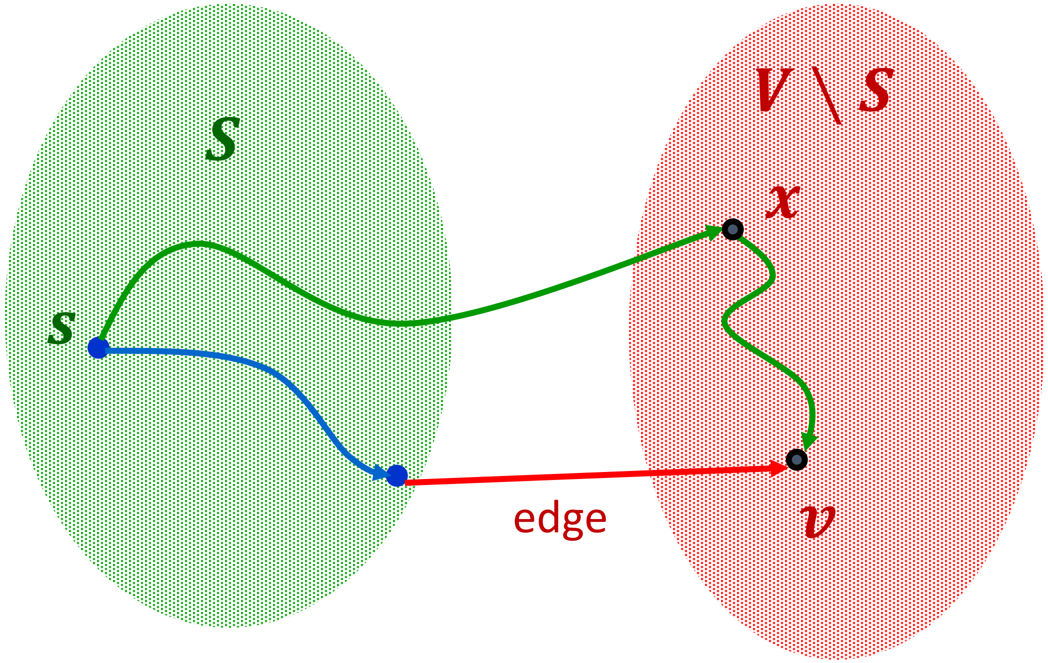


Add to ***S***

Dijkstra's Algorithm Correctness

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

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



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

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

$\boldsymbol{x} \rightarrow \boldsymbol{v}$ path length $\geq \mathbf{0}$

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

Therefore adding $\boldsymbol{v}$ to $\boldsymbol{S}$ maintains that all distances inside $\boldsymbol{S}$ are correct

Dijkstra's Algorithm

- Algorithm also produces a tree of shortest paths to $\boldsymbol{v}$ following the inverse of $\boldsymbol{pred}$ links
  - From $\boldsymbol{v}$ follow its ancestors in the tree back to $\boldsymbol{s}$ reversing edges along the path

- If all you care about is the shortest path from $\boldsymbol{s}$ to $\boldsymbol{v}$ simply stop the algorithm when $\boldsymbol{v}$ is added to $\boldsymbol{S}$

Dijkstra's Algorithm

Dijkstra($\boldsymbol{G}$,$\boldsymbol{w}$,s)

$\boldsymbol{S}$ = $\{\boldsymbol{s}\}$

$\boldsymbol{d}[\boldsymbol{s}]$ = $\mathbf{0}$

while $\boldsymbol{S}\neq\boldsymbol{V}$ {

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

$\boldsymbol{S}$ = $\boldsymbol{S}$ ∪ $\{\boldsymbol{v}\}$

$\boldsymbol{d}[\boldsymbol{v}]$ =$\boldsymbol{d}[\boldsymbol{u}] + \boldsymbol{w}(\boldsymbol{e})$

$\boldsymbol{pred}[\boldsymbol{v}]$=$\boldsymbol{u}$

}

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

Implementing Dijkstra’s Algorithm

Need to

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

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

Dijkstra's Algorithm with Priority Queues

- For each vertex $\boldsymbol{v}$ not in tree maintain cost $\boldsymbol{d'}[\boldsymbol{v}]$ of current cheapest path through tree to $\boldsymbol{v}$
  - Store $\boldsymbol{v}$ in priority queue with key = length of this path
- Operations:
  - $\boldsymbol{n-1}$ insertions (each vertex added once)
  - $\boldsymbol{n-1}$ 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
  - $< \boldsymbol{m}$ decrease-keys (each edge updates one vertex)

Dijskstra’s Algorithm with Priority Queues

Priority queue implementations

- 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 $m = \Theta(n^2)$

- d-Heap $(d = m/n)$
  - m insert, decrease-key $O(\log_{m/n} n)$
  - $n - 1$ delete-min $O((m/n)\log_{m/n} n)$
  - total $O(m \log_{m/n} n)$

Better for all values of m