Greedy Algorithms

What makes a greedy algorithm?

- Feasible
  - Has to satisfy the problem's constraints
- Locally Optimal
  - The greedy part
  - Has to make the best local choice among all feasible choices available on that step
    - If this local choice results in a global optimum then the problem has optimal substructure
- Irrevocable
  - Once a choice is made it can't be un-done on subsequent steps of the algorithm

- Simple examples:
  - Playing chess by making best move without lookahead
  - Giving fewest number of coins as change

- Simple and appealing, but don't always give the best solution

An Activity Selection Problem (Conference Scheduling Problem)

- **Input: A set of activities S = {a_1,…, a_n}**
- Each activity has start time and a finish time
  - a_i=(s_i, f_i)
- Two activities are compatible if and only if their interval does not overlap
- **Output: a maximum-size subset of mutually compatible activities**

The Activity Selection Problem

- Here are a set of start and finish times

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

The Activity Selection Problem

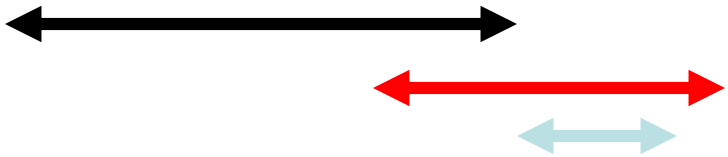
- Here are a set of start and finish times

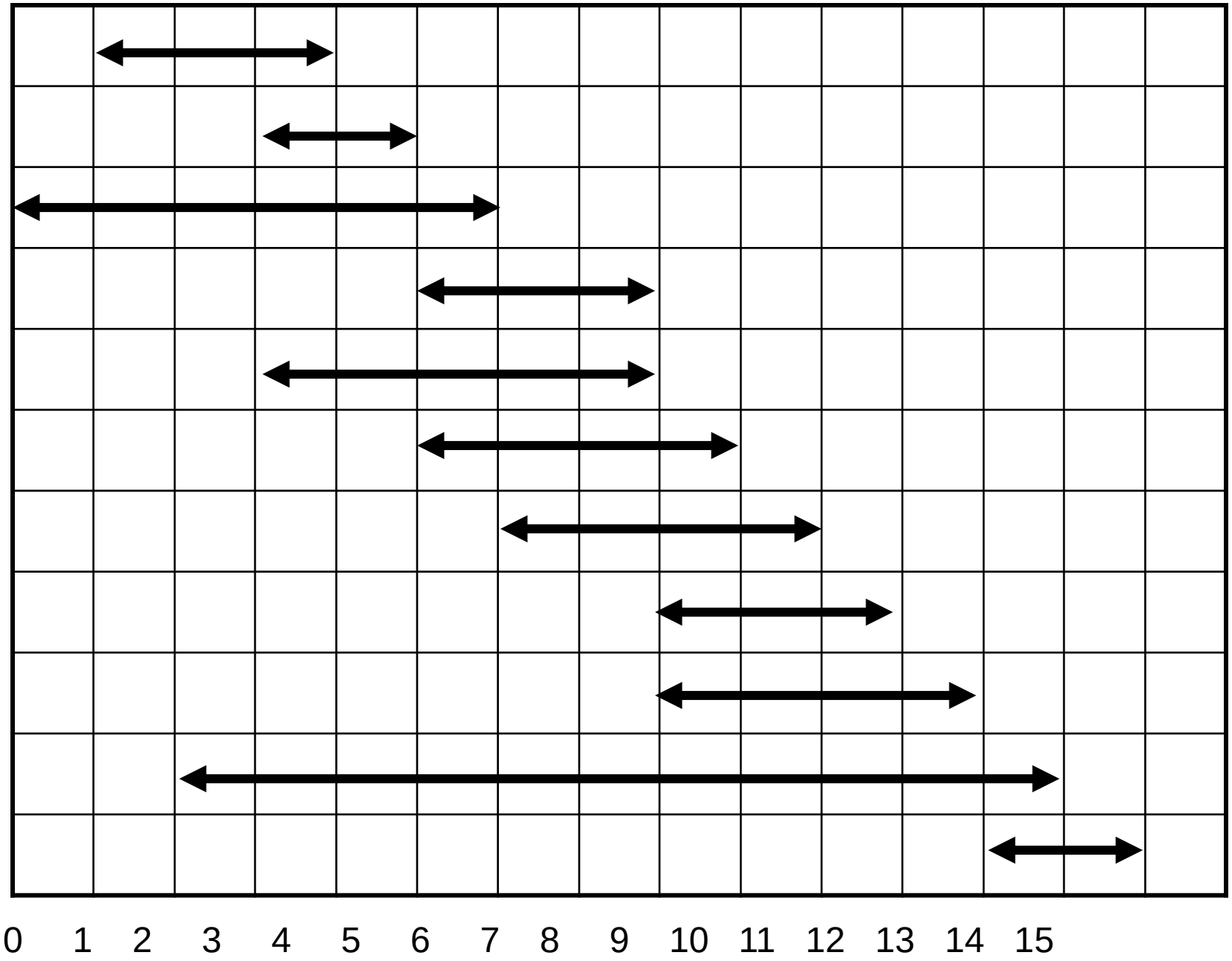
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

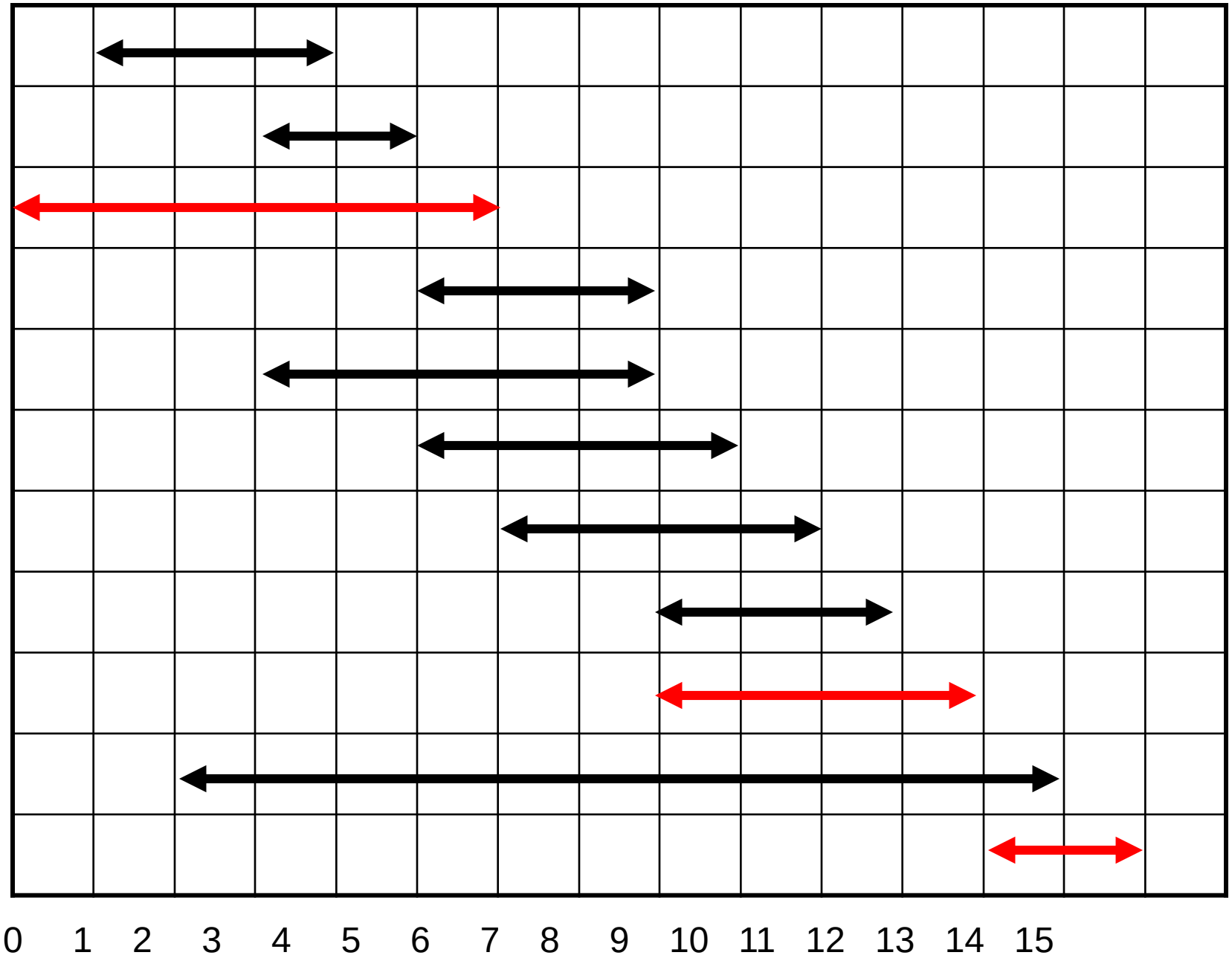
- What is the maximum number of activities that can be completed?
  - $\{a_3, a_9, a_{11}\}$ can be completed
  - But so can $\{a_1, a_4, a_8, a_{11}\}$ which is a larger set
  - But it is not unique, consider $\{a_2, a_4, a_9, a_{11}\}$

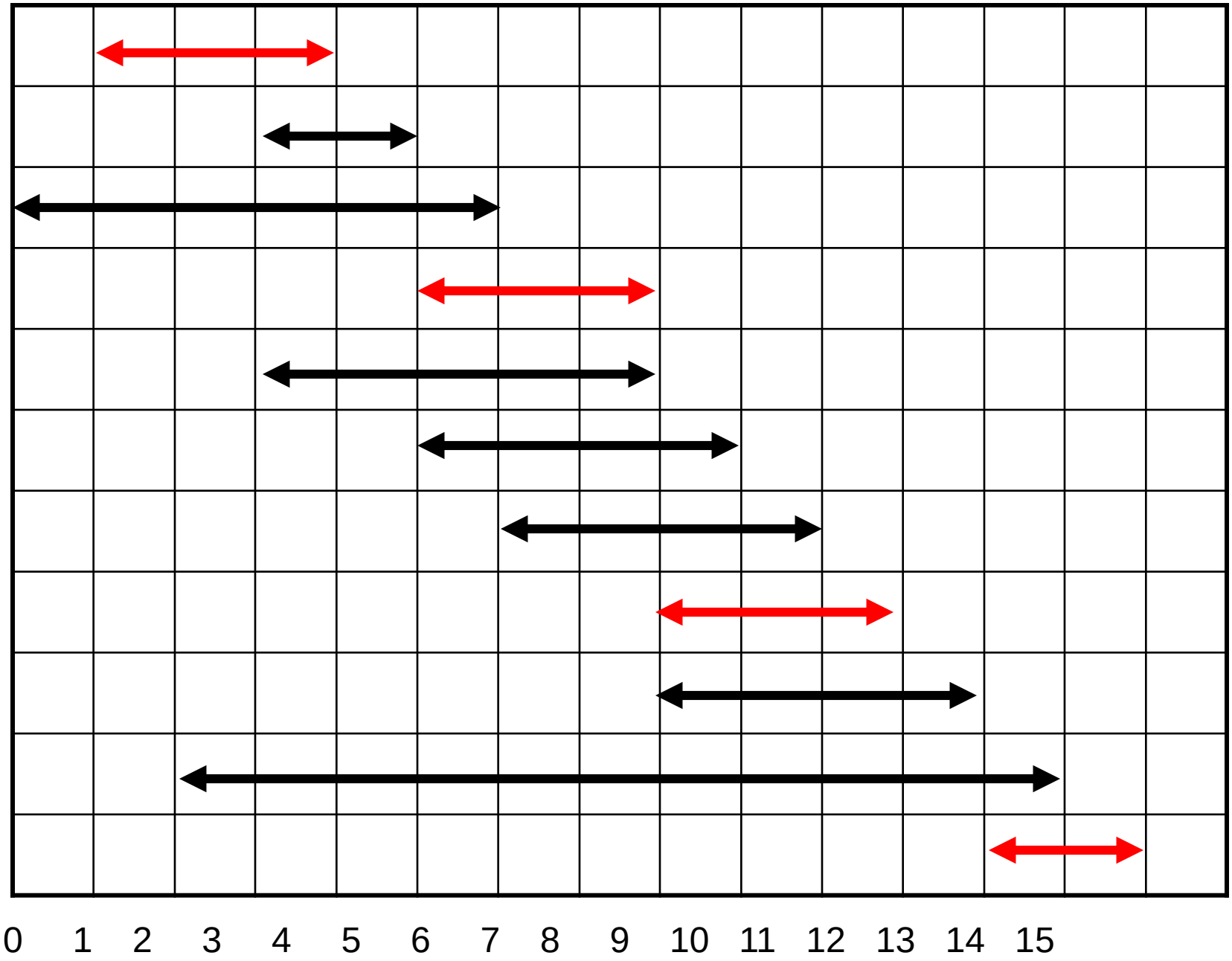
Interval Representation

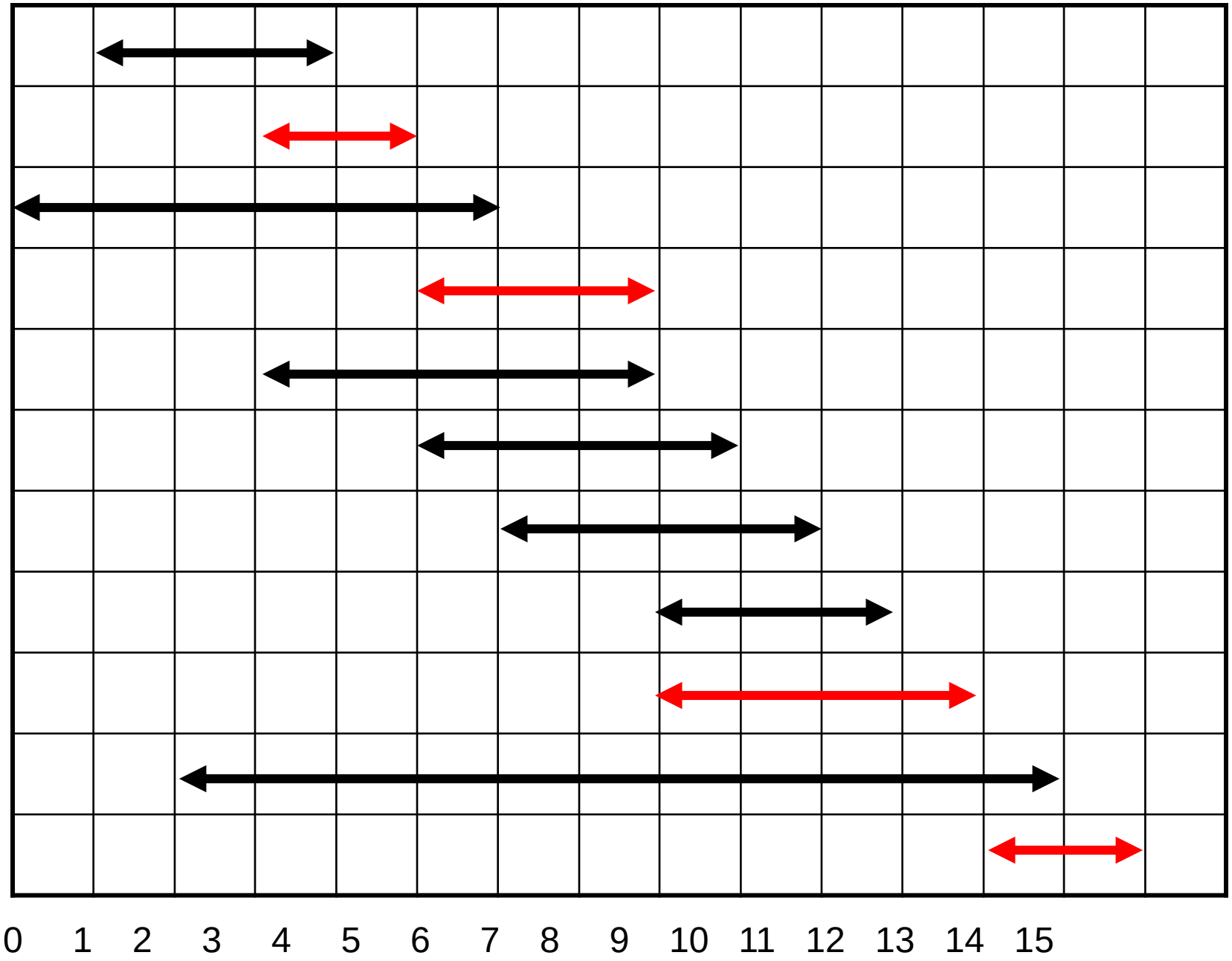
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14




0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

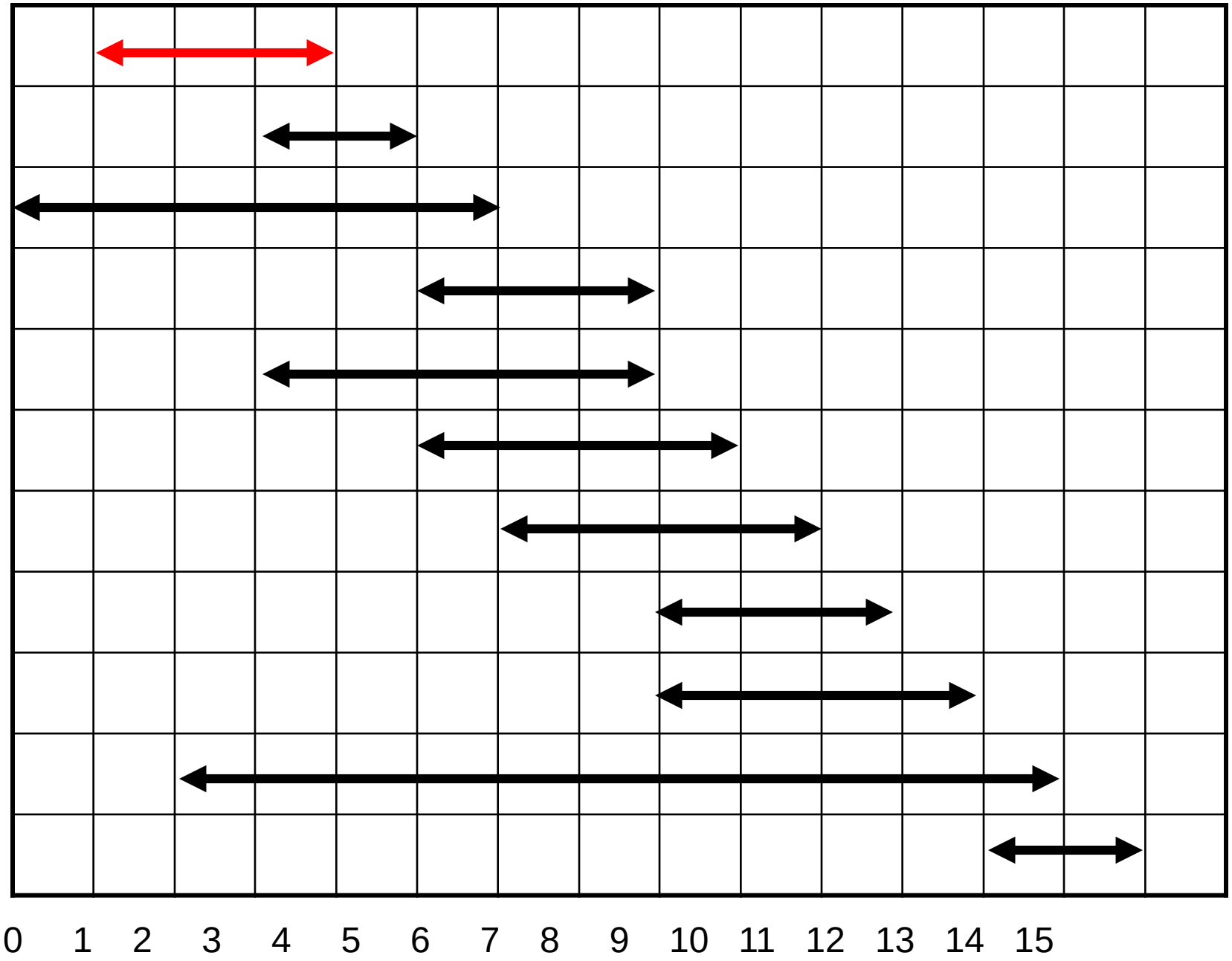

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

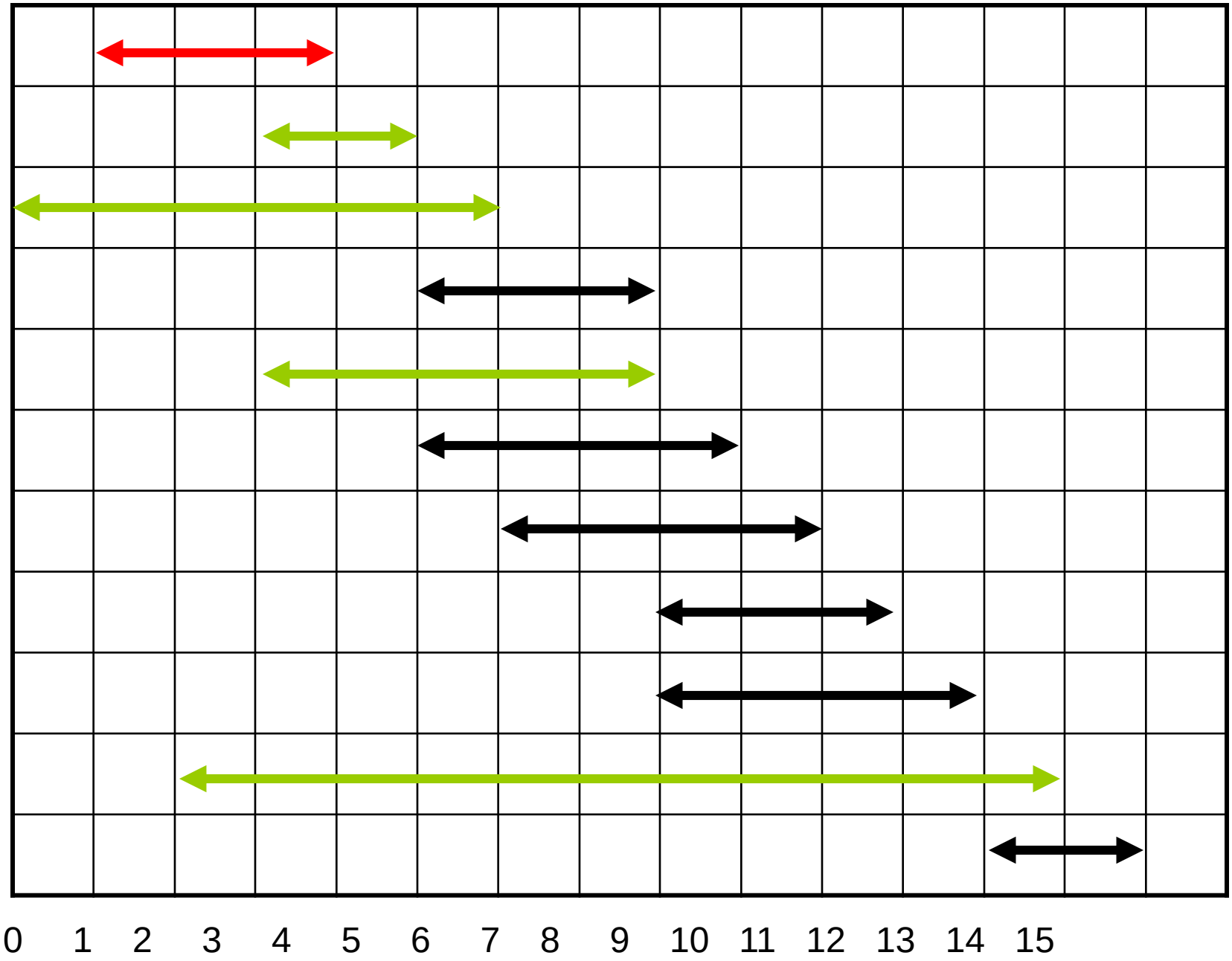

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

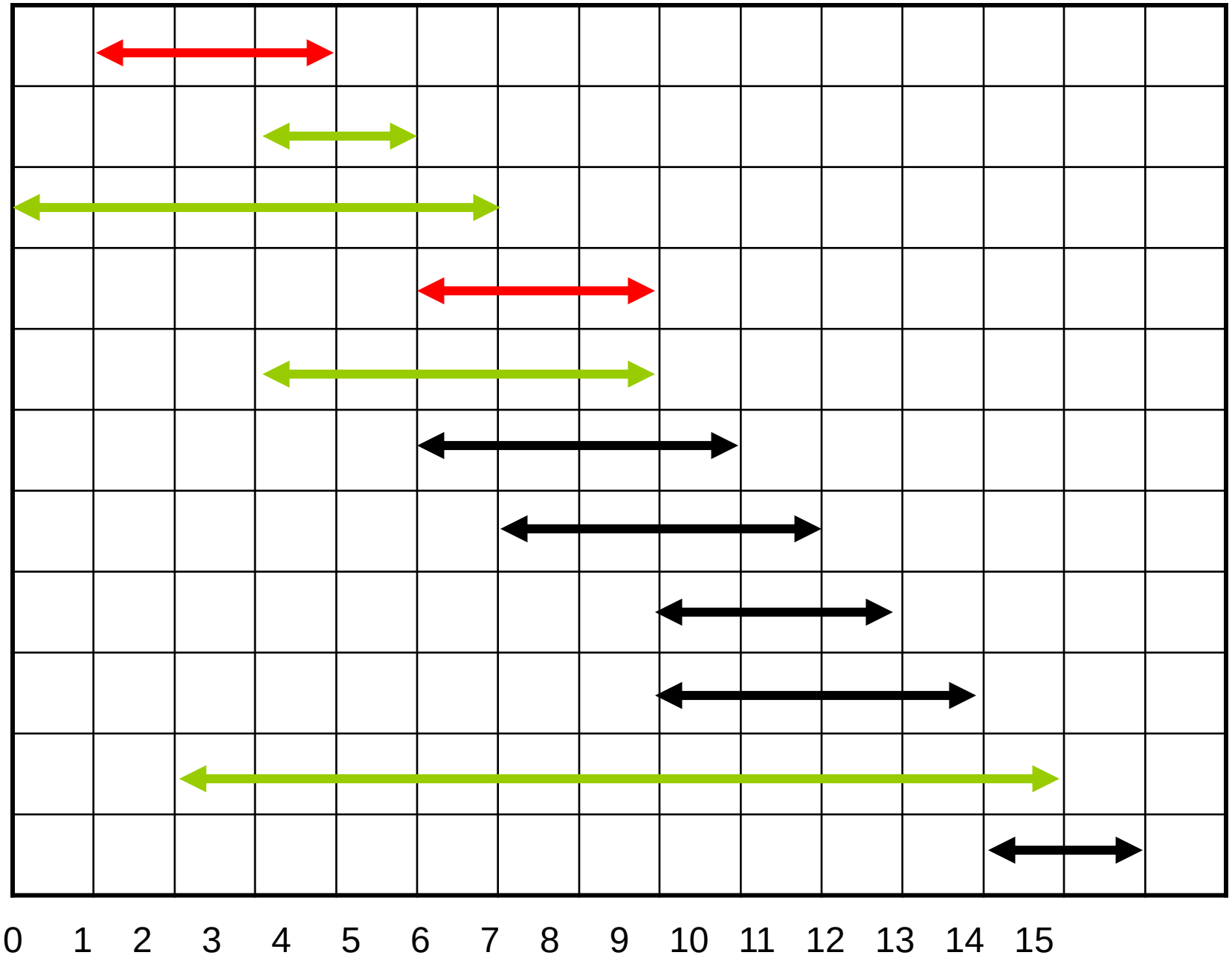

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

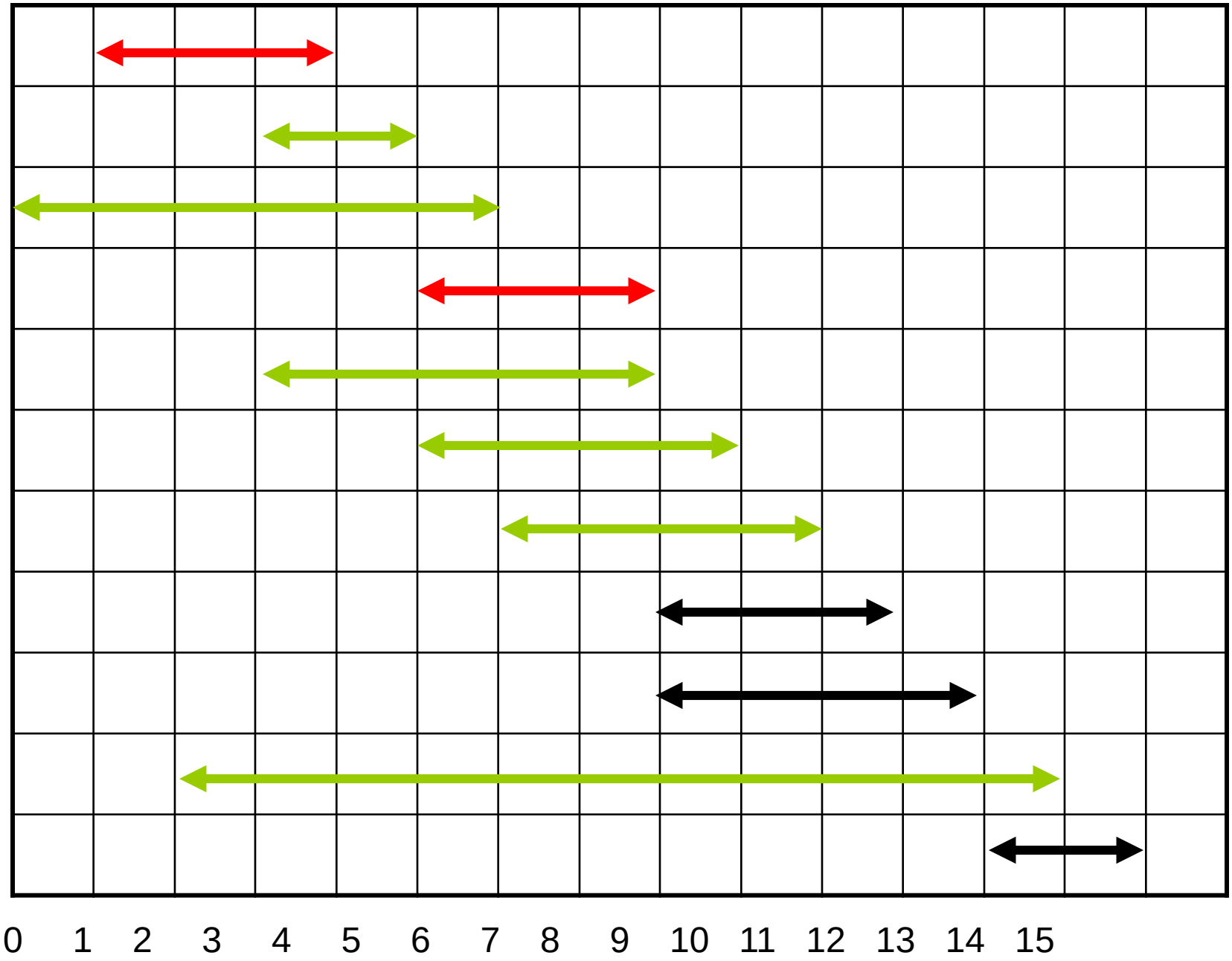
Early Finish Greedy

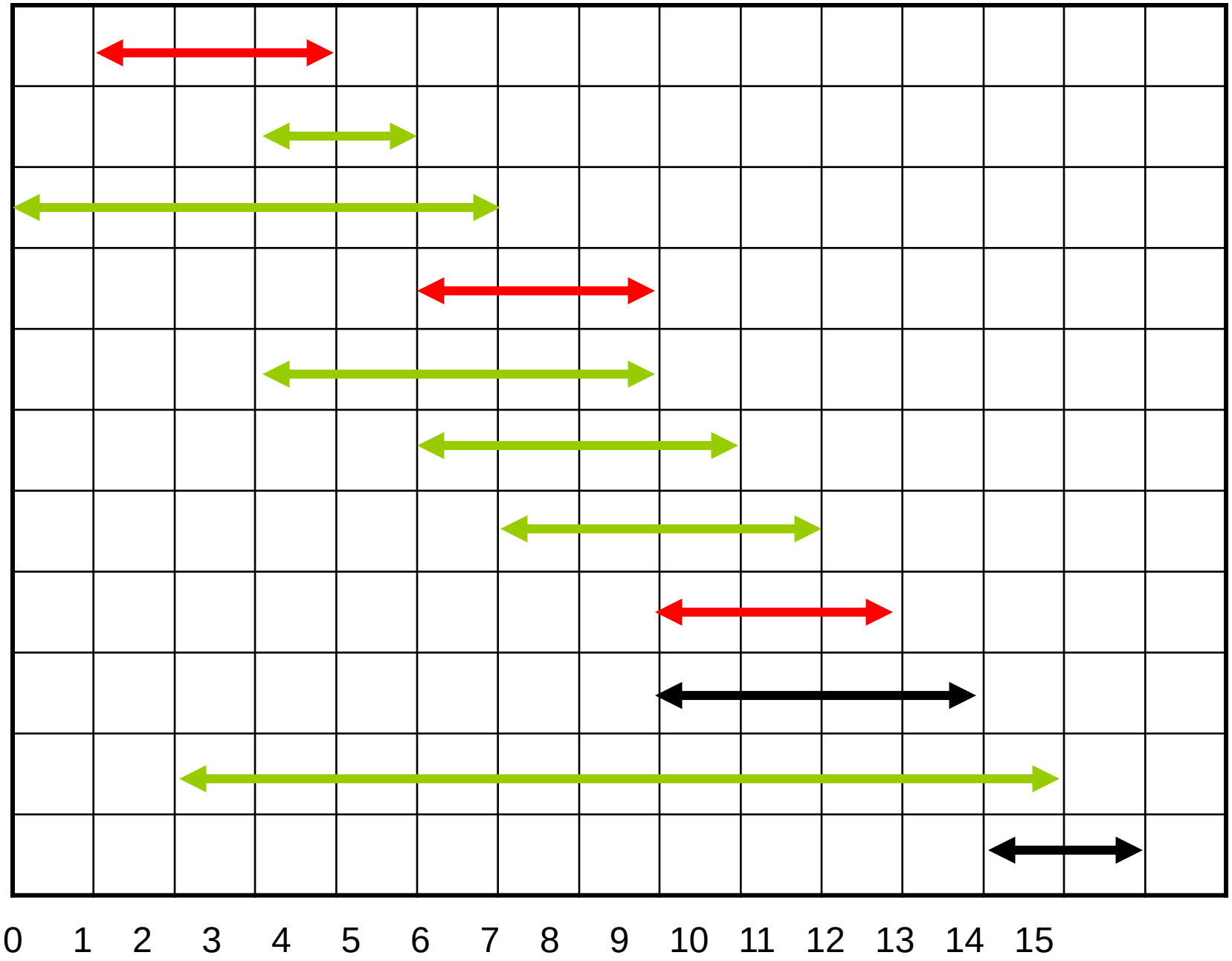
- Select the activity with the earliest finish
- Eliminate the activities that could not be scheduled
- Repeat!

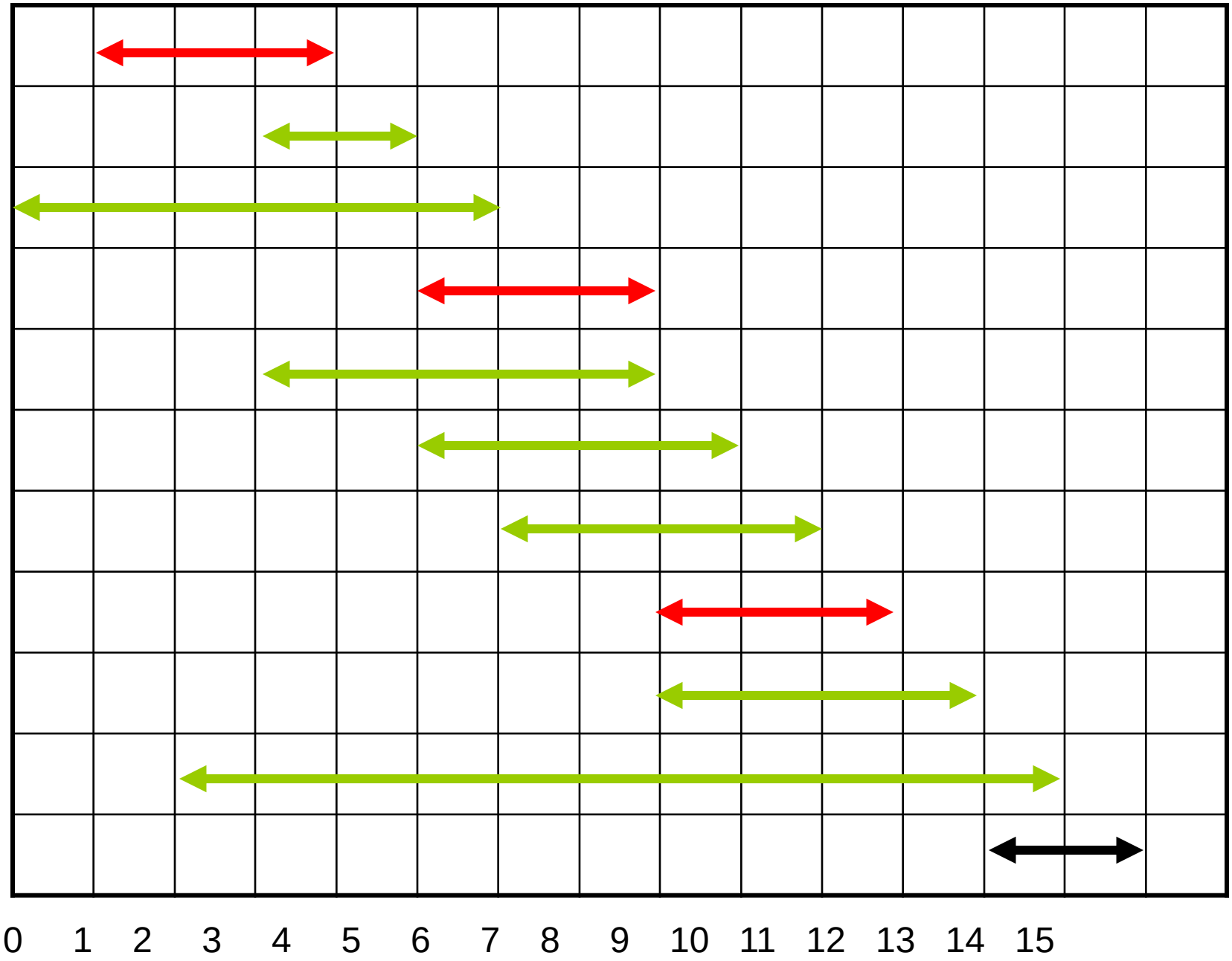

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

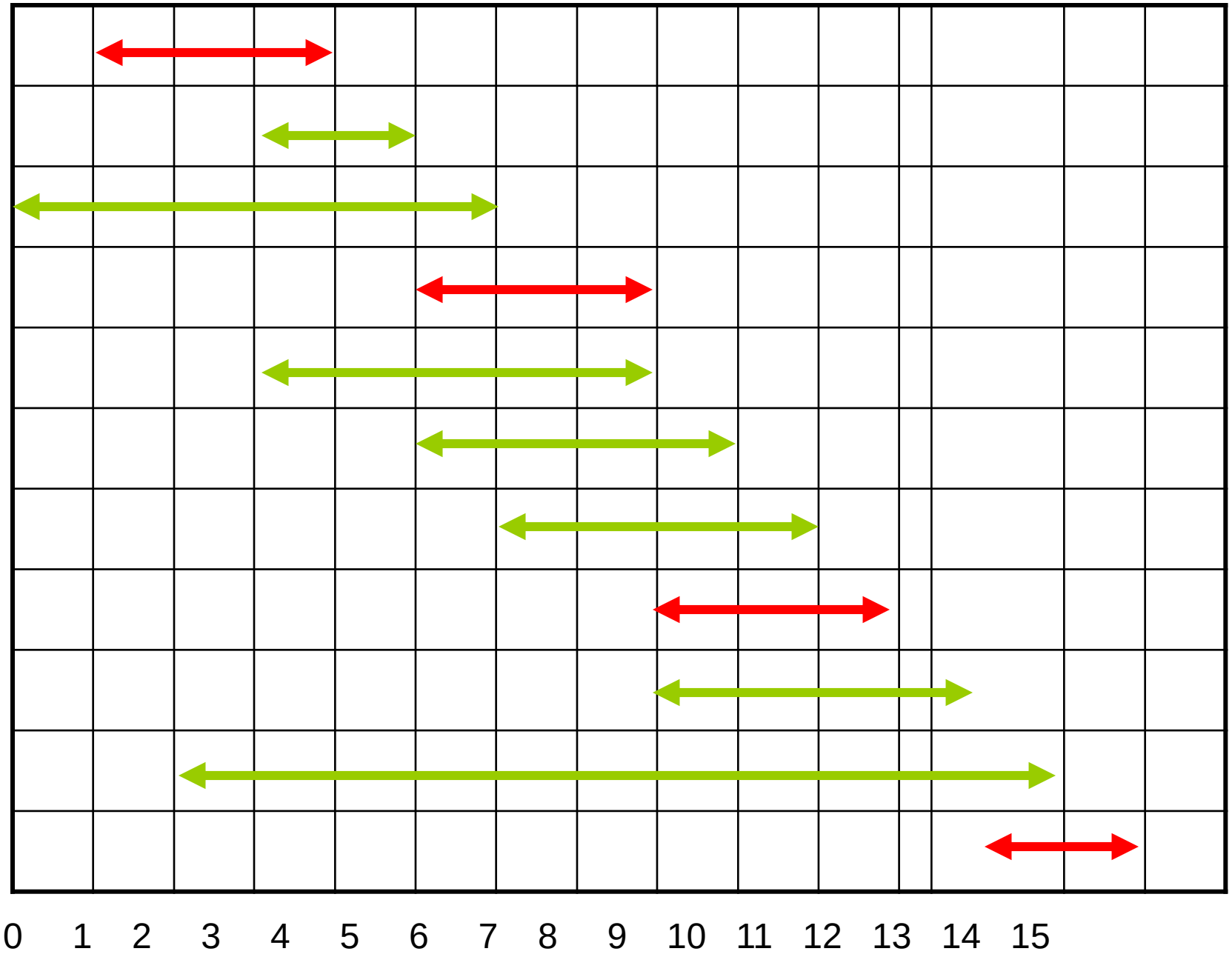

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Assuming activities are sorted by finish time

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1   n ← length[s]
2   A ← {a_1}
3   i ← 1
4   for m ← 2 to n
5         do if s_m ≥ f_i
6               then A ← A ∪ {a_m}
7                    i ← m
8   return A
```

Disjoint Sets

- Some applications involve grouping n distinct elements into a collection of **disjoint sets**.
- Important operations in this case are to construct a set, to find which set a given element belongs to, and to unite two sets.

Definitions

- A **disjoint-set data structure** maintains a collection $S = \{ S_1, S_2, \cdots, S_n \}$ of disjoint dynamic sets.
- Each set is identified by a **representative**, which is some member of the set.
- The disjoint sets might form a **partition** of a universe set U.

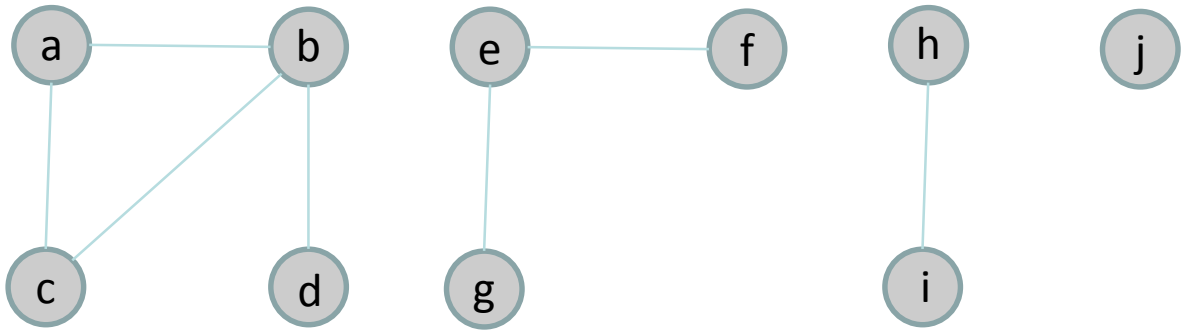
Definitions (cont'd)

- The disjoint-set data structure supports the following operations:
  - MAKE-SET(x): It creates a new set whose only member (and thus representative) is pointed to by x. Since the sets are disjoint, we require that x not already be in any of the existing sets.
  - UNION(x, y): It unites the dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets. One of the S_x and S_y give its name to the new set and the other set is "destroyed" by removing it from the collection S. The two sets are assumed to be disjoint prior to the operation. The representative of the resulting set is some member of $S_x \cup S_y$ (usually the representative of the set that gave its name to the union).
  - FIND-SET(x) returns a pointer to the representative of the unique set containing x.

Determining the Connected Components of an Undirected Graph

- One of the many applications of disjoint-set data structures is **determining the connected components of an undirected graph**.
- The implementation based on disjoint-sets that we will present here is appropriate when the edges of the graph are not static e.g., when **edges are added dynamically** and we need to maintain the connected components as each edge is added.

Example Graph



Computing the Connected Components of an Undirected Graph

- The following procedure CONNECTED-COMPONENTS uses the disjoint-set operations to compute the connected components of a graph.

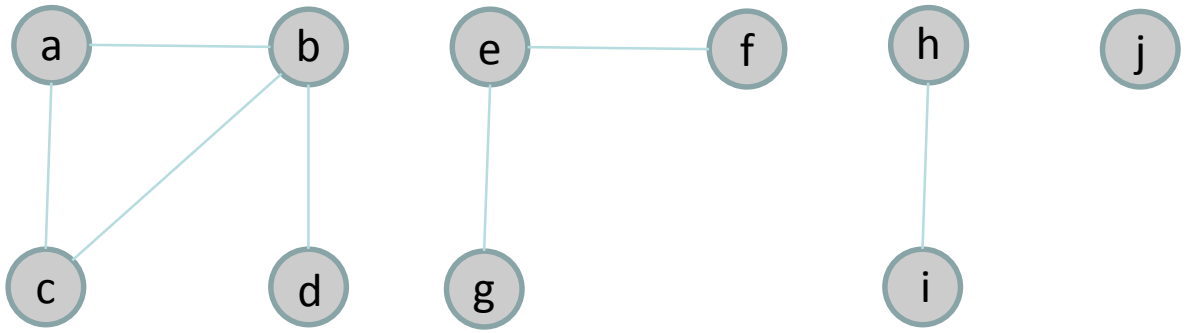
CONNECTED-COMPONENTS(G)
  for each vertex $v \in V[G]$
    do MAKE-SET(v)
  for each edge $(u, v) \in E[G]$
    do if FIND-SET(u)≠FIND-SET(v)
      then UNION(u, v)

Computing the Connected Components (cont'd)

- Once CONNECTED-COMPONENTS has been run as a preprocessing step, the procedure SAME-COMPONENT given below answers queries about whether two vertices are in the same connected component.

SAME-COMPONENT(u, v)
  if FIND-SET(u)=FIND-SET(v)
    then return TRUE
    else return FALSE

Example Graph



The Collection of Disjoint Sets After Each Edge is Processed

Edge processed										
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

Minimum Spanning Trees

- Another application of the disjoint set operations that we will see is Kruskal's algorithm for computing the minimum spanning tree (MST) of a graph.

Spanning Tree

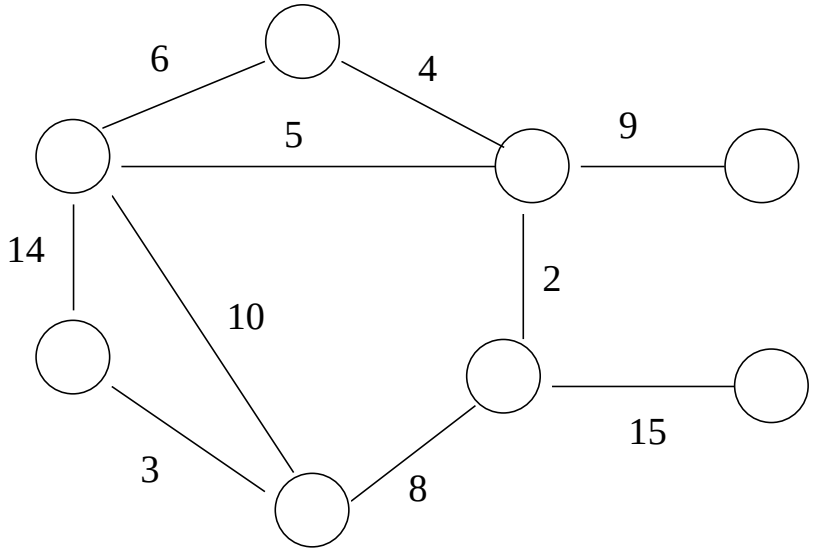
- Definition
  - A spanning tree of a graph G is a tree (acyclic) that connects all the vertices of G once
    - i.e. the tree “spans” every vertex in G
  - A Minimum Spanning Tree (MST) is a spanning tree on a weighted graph that has the minimum total weight

$$w(T) = \sum_{u,v \in T} w(u,v) \text{ such that w(T) is minimum}$$

Where might this be useful? Can also be used to approximate some NP-Complete problems

Sample MST

- Which links to make this a MST?



Optimal substructure: A subtree of the MST must in turn be a MST of the nodes that it spans.

MST Claim

- Claim: Say that M is a MST
  - If we remove any edge (u,v) from M then this results in two trees, T1 and T2.
  - T1 is a MST of its subgraph while T2 is a MST of its subgraph.
  - Then the MST of the entire graph is T1 + T2 + the smallest edge between T1 and T2
  - If some other edge was used, we wouldn’t have the minimum spanning tree overall

Greedy Algorithm

- We can use a greedy algorithm to find the MST.
  - Two common algorithms
    - Kruskal
    - Prim

Kruskal's MST Algorithm

- Idea: Greedily construct the MST
  - Go through the list of edges and make a forest that is a MST
  - At each vertex, sort the edges
  - Edges with smallest weights examined and possibly added to MST before edges with higher weights
  - Edges added must be "safe edges" that do not ruin the tree property.

Kruskal's Algorithm

Kruskal(G,w) ; Graph G, with weights w

A $\leftarrow$ {} ; Our MST starts empty

for each vertex $v \in V[G]$ do Make-Set(v) ; Make each vertex a set

Sort edges of E by increasing weight

for each edge $(u, v) \in E$ in order

; Find-Set returns a representative (first vertex) in the set

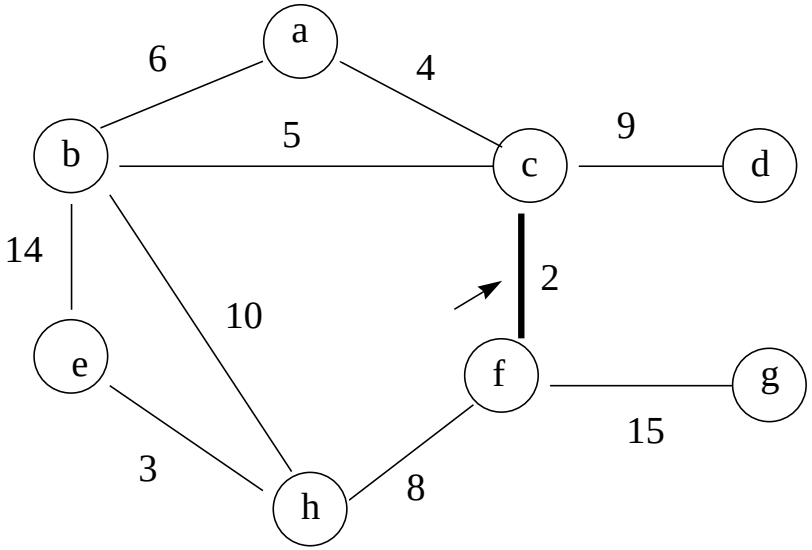
do if Find-Set(u) $\neq$ Find-Set(v)

then A $\leftarrow$ A $\cup \{(u, v)\}$

Union(u,v) ; Combines two trees

return A

Kruskal's Example



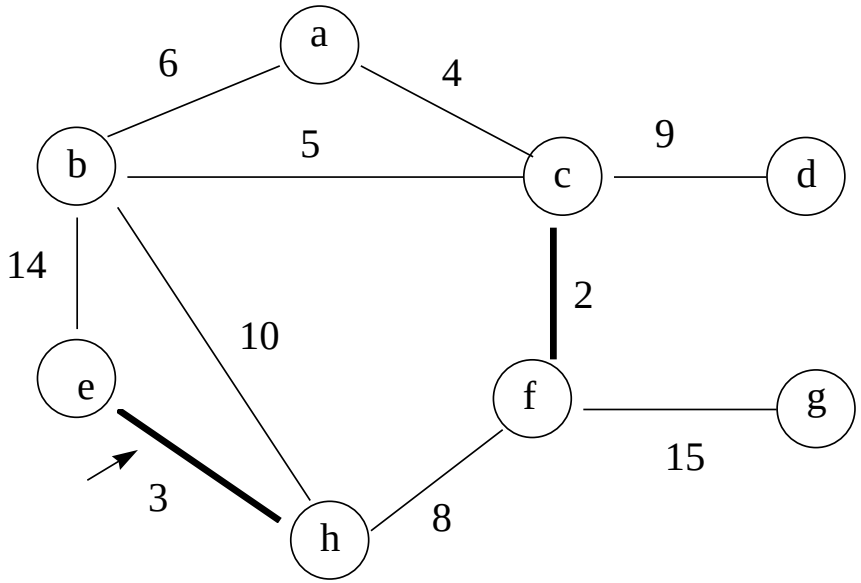
- A={ }, Make each element its own set. {a} {b} {c} {d} {e} {f} {g} {h}
- Sort edges.
- Look at smallest edge first: {c} and {f} not in same set, add it to A, union together.
- Now get {a} {b} {c f} {d} {e} {g} {h}

Kruskal Example

Keep going, checking next smallest edge.

Had: {a} {b} {c f} {d} {e} {g} {h}

{e} ≠ {h}, add edge.



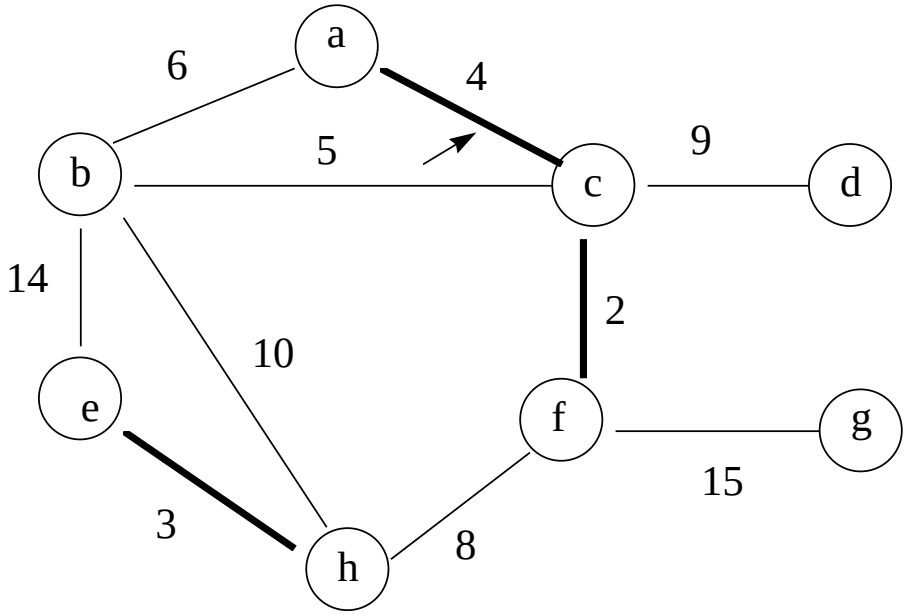
Now get {a} {b} {c f} {d} {e h} {g}

Kruskal Example

Keep going, checking next smallest edge.

Had: {a} {b} {c f} {d} {e h} {g}

{a} ≠ {c f}, add edge.



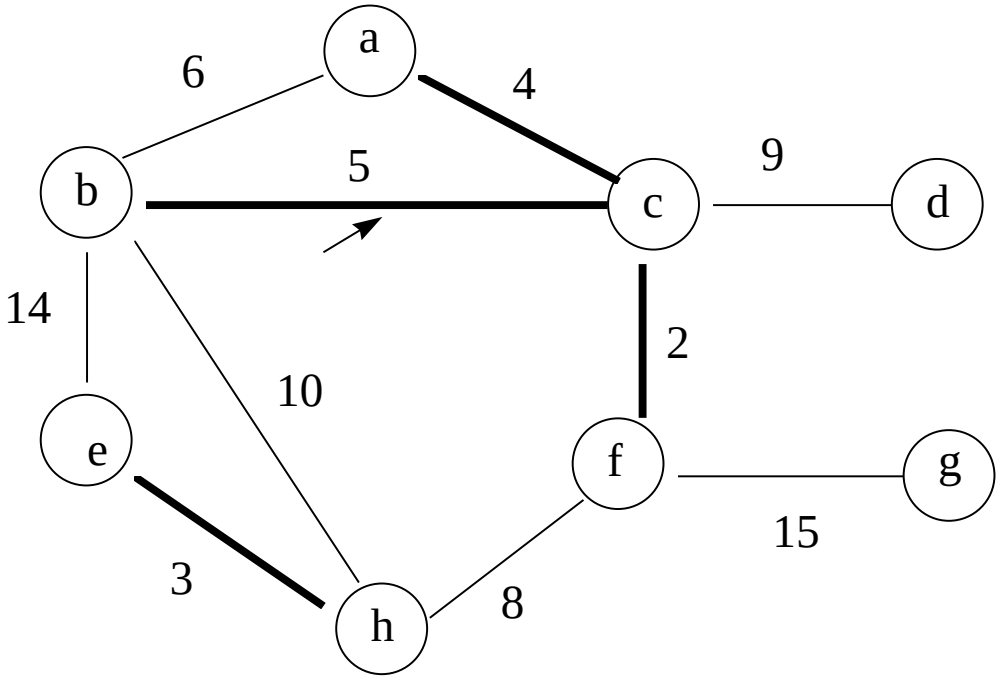
Now get {b} {a c f} {d} {e h} {g}

Kruskal's Example

Keep going, checking next smallest edge.

Had {b} {a c f} {d} {e h} {g}

{b} ≠ {a c f}, add edge.



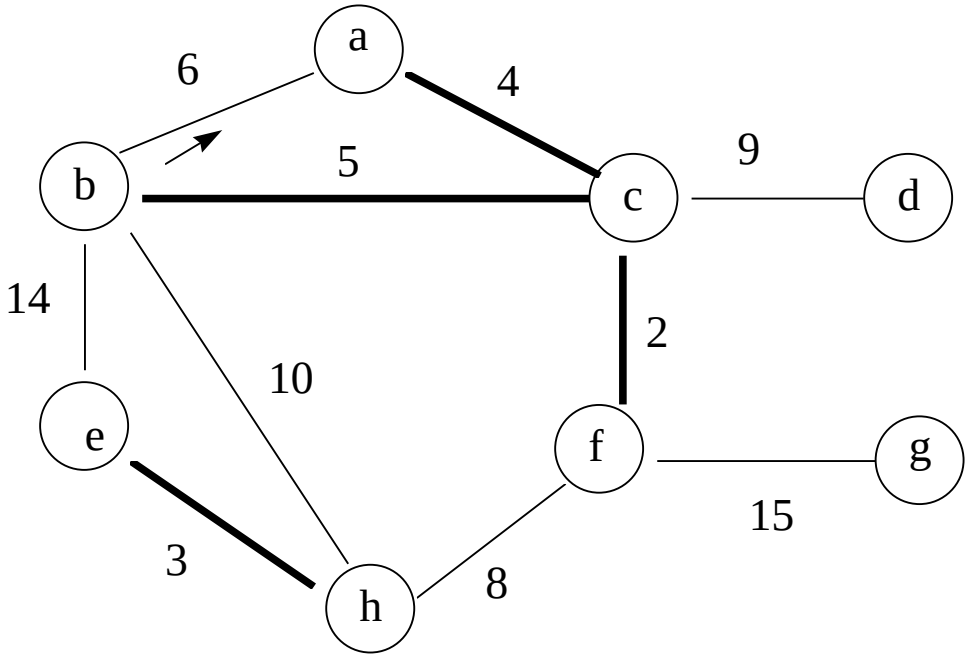
Now get {a b c f} {d} {e h} {g}

Kruskal's Example

Keep going, checking next smallest edge.

Had {a b c f} {d} {e h} {g}

{a b c f} = {a b c f}, dont add it!

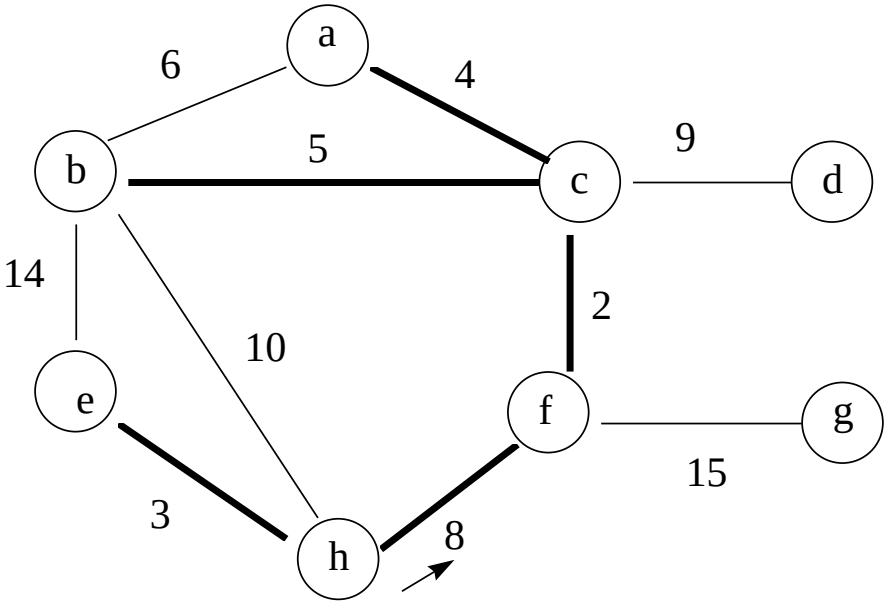


Kruskal's Example

Keep going, checking next smallest edge.

Had {a b c f} {d} {e h} {g}

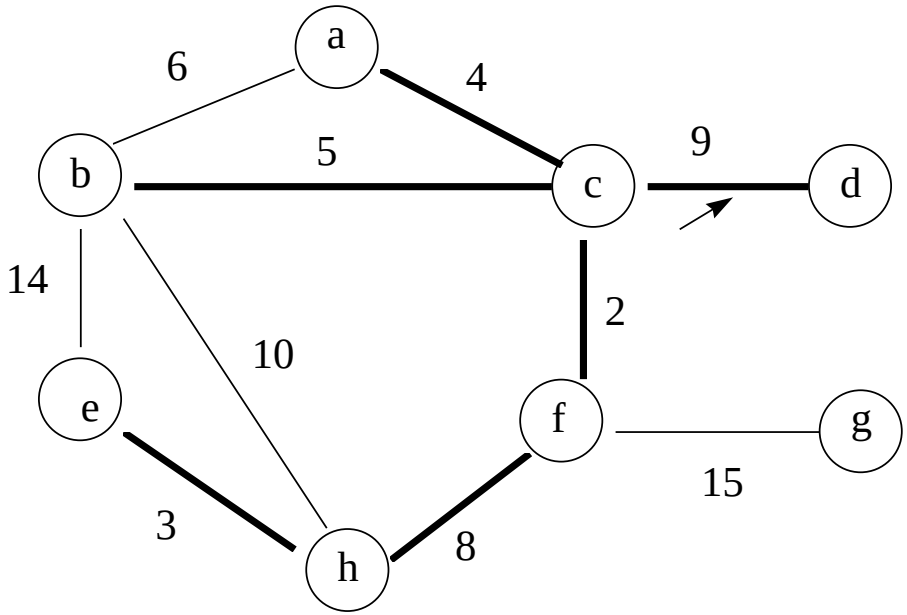
{a b c f} = {e h}, add it.



Now get {a b c f e h} {d}{g}

Kruskal's Example

Keep going, checking next smallest edge.
Had {a b c f e h} {d}{g}
{d} ≠ {a b c e f h}, add it.



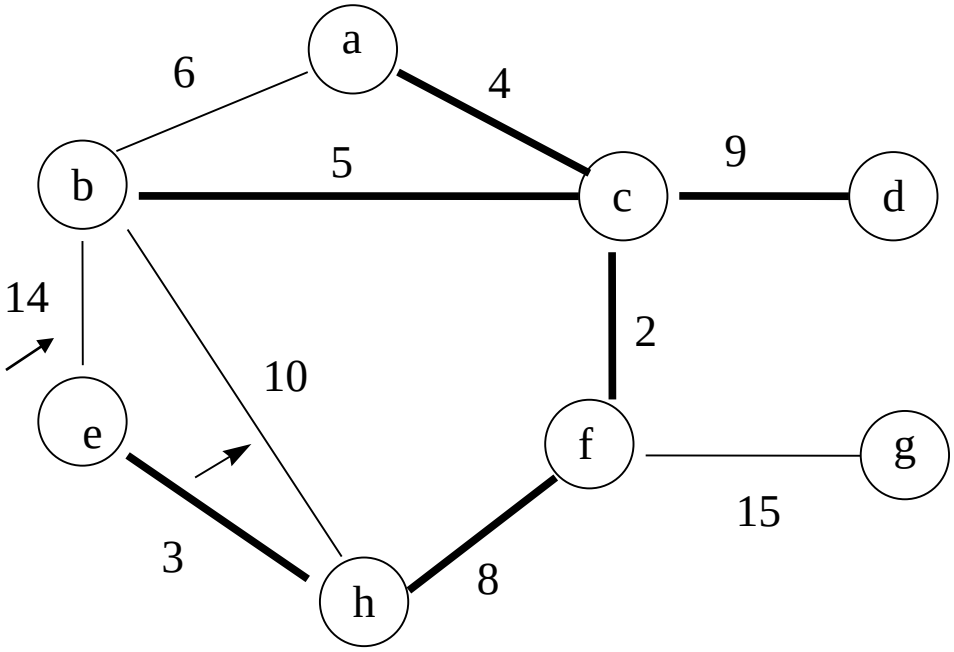
Now get {a b c d e f h} {g}

Kruskal's Example

Keep going, check next two smallest edges.

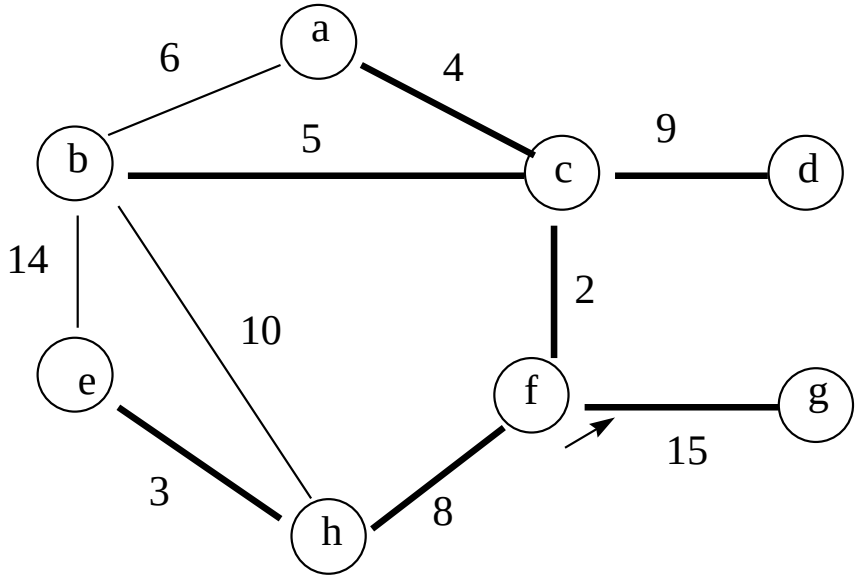
Had {a b c d e f h} {g}

{a b c d e f h} = {a b c d e f h}, don't add it.



Kruskal's Example

Do add the last one:
Had {a b c d e f h} {g}



Runtime of Kruskal's Algo

- Runtime depends upon time to union set, find set, make set
- Simple set implementation: number each vertex and use an array
  - Use an array

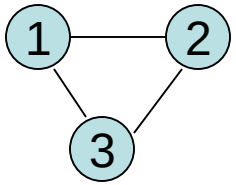
    member[] : member[i] is a number j such that the ith vertex is a member of the jth set.
  - Example

    member[1,4,1,2,2]

    indicates the sets S1={1,3}, S2={4,5} and S4={2};

    i.e. position in the array gives the set number. Idea similar to counting sort, up to number of edge members.

Set Operations



- Given the Member array

- Make-Set(v)
  member[v] = v

member = [1,2,3] ; {1} {2} {3}

Make-Set runs in constant running time for a single set.

- Find-Set(v)
  Return member[v]

find-set(2) = 2

Find-Set runs in constant time.

- Union(u,v)
  for i=1 to n
  do if member[i] = u then member[i]=v

Union(2,3)
member = [1,3,3] ; {1} {2 3}

Scan through the member array and update old members to be the new set.
Running time O(n), length of member array.

Overall Runtime

```
Kruskal(G,w)                ; Graph G, with weights w
    A ← {}                  ; Our MST starts empty
    for each vertex v ∈ V[G] do Make-Set(v)   ; Make each vertex a set
    Sort edges of E by increasing weight
    for each edge (u,v) ∈ E in order
        ;  Find-Set returns a representative (first vertex) in the set
        do if Find-Set(u) ≠ Find-Set(v)
            then A ← A ∪ {(u,v)}
                Union(u,v)                   ; Combines two trees
    return A
```

O(V)

O(ElgE) – using heapsort

O(E)

O(1)

O(V)

Total runtime: O(V)+O(ElgE)+O(E*(1+V)) = O(E*V)

Book describes a version using disjoint sets that runs in O(E*lgE) time

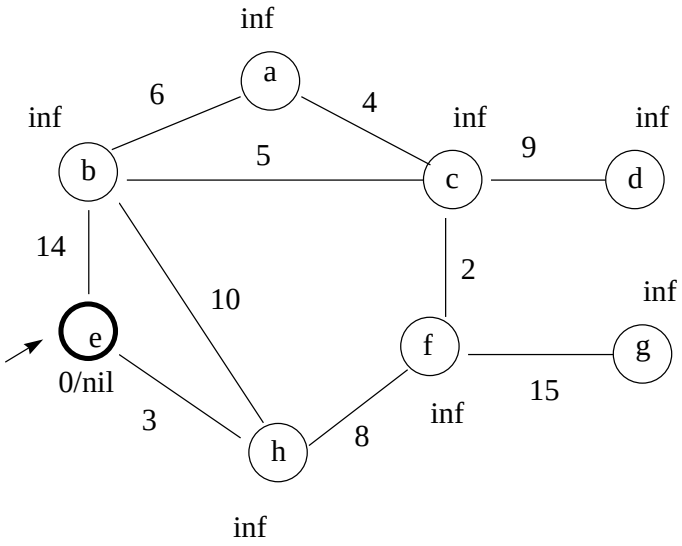
Prim's MST Algorithm

- Also greedy, like Kruskal's
- Will find a MST but may differ from Prim's if multiple MST's are possible

```
MST-Prim(G,w,r)                          ; Graph G, weights w, root r
        Q ← V[G]
        for each vertex u ∈ Q do key[u] ← ∞      ; infinite "distance"
        key[r] ← 0
        P[r] ← NIL
        while Q<>NIL do
                u ← Extract-Min(Q)               ; remove closest node
                ; Update children of u so they have a parent and a min key val
                ; the key is the weight between node and parent
                for each v ∈ Adj[u] do
                        if v ∈ Q & w(u,v)<key[v] then
                                P[v] ← u
                                key[v] ← w(u,v)
```

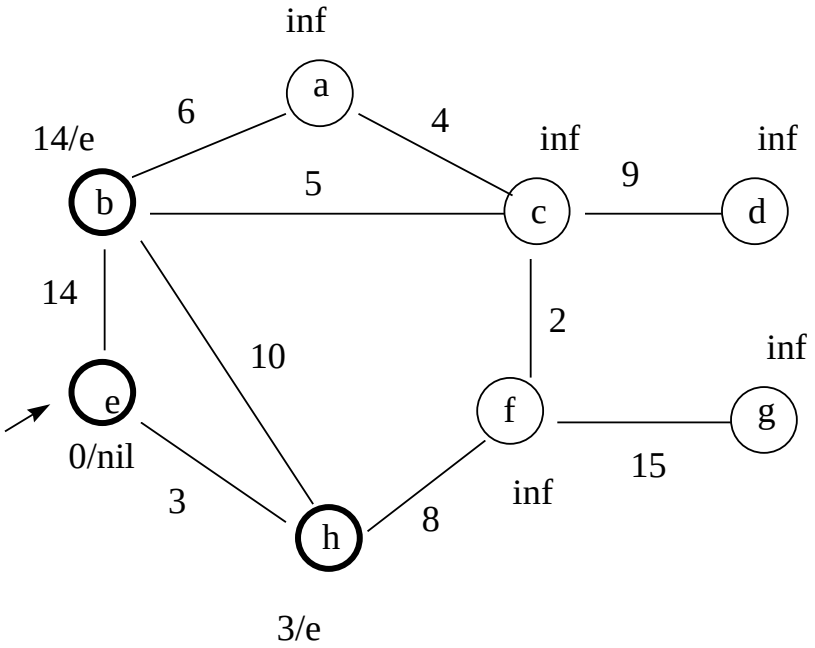
Prim's Example

Example: Graph given earlier.
Q={ (e,0) (a, ∞) (b, ∞) (c, ∞) (d, ∞) (f, ∞) (g, ∞) (h, ∞) }



Extract min, vertex e. Update neighbor if in Q and weight < key.

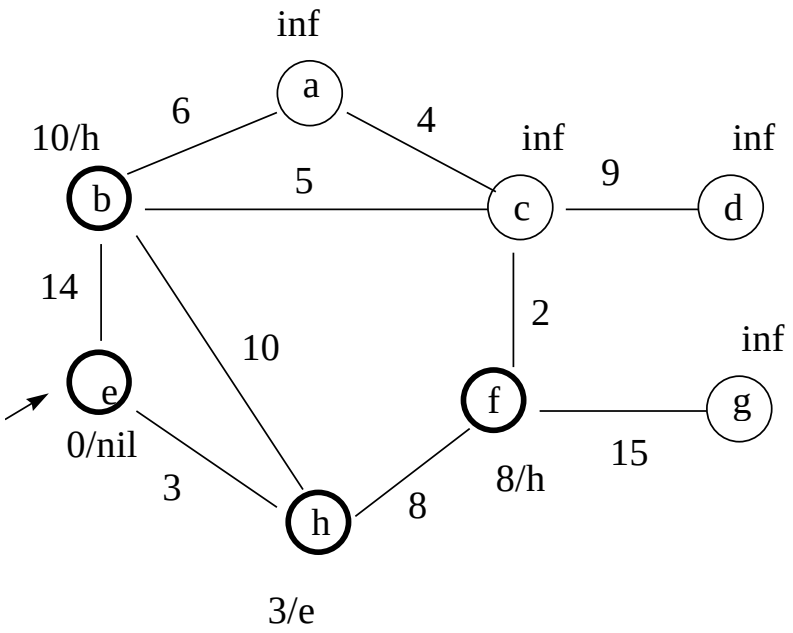
Prim's Example



Q={ (a, ∞) (b,14) (c, ∞) (d, ∞) (f, ∞) (g, ∞) (h,3) }

Extract min, vertex h. Update neighbor if in Q and weight < key

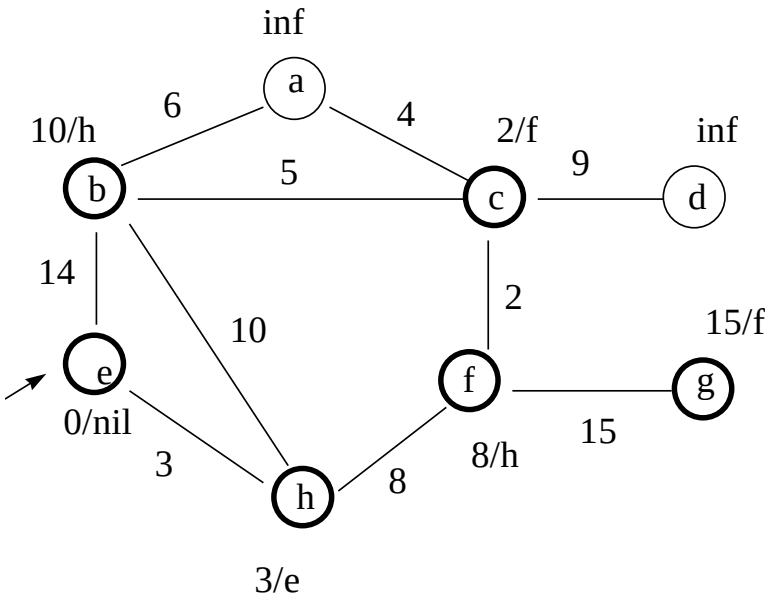
Prim's Algorithm



Q={ (a, ∞) (b,10) (c, ∞) (d, ∞) (f,8) (g, ∞) }

Extract min, vertex f. Update neighbor if in Q and weight < key

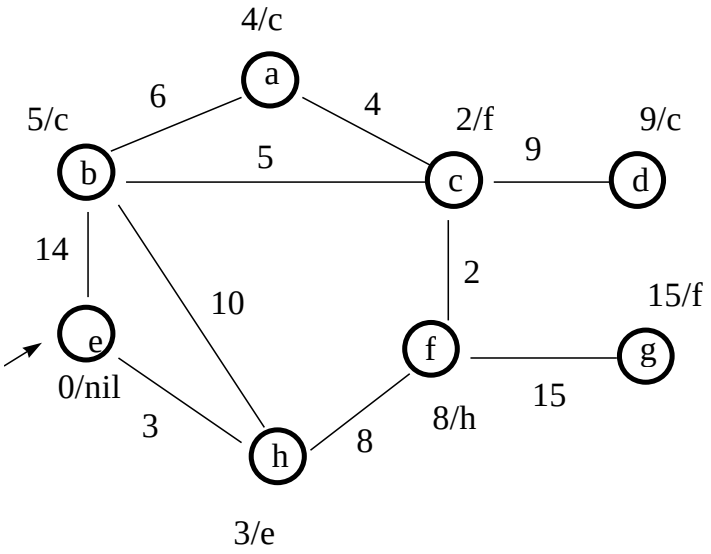
Prim's Algorithm



Q={ (a, ∞) (b,10) (c, 2) (d, ∞) (g,15) }

Extract min, vertex c. Update neighbor if in Q and weight < key

Prim's Algorithm



Q={ (a,4) (b,5) (d,9) (g,15) }

Extract min, vertex a. No keys are smaller than edges from a (4>2 on edge ac, 6>5 on edge ab) so nothing done.

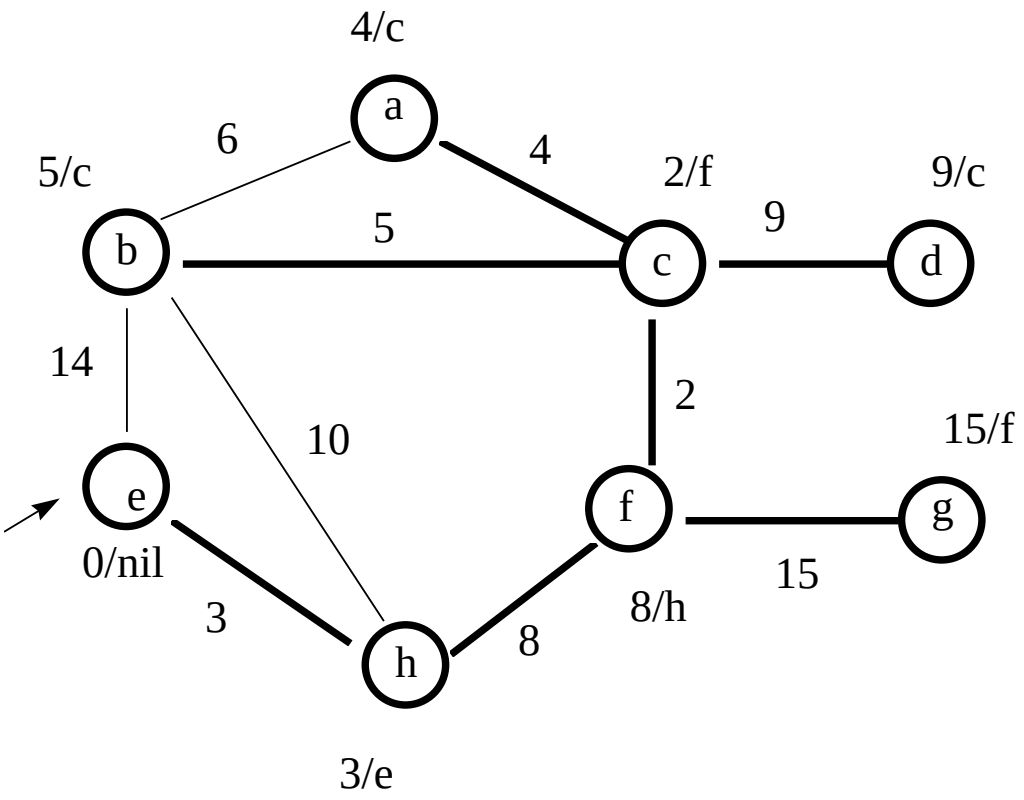
Q={ (b,5) (d,9) (g,15) }

Extract min, vertex b.

Same case, no keys are smaller than edges, so nothing is done.
Same for extracting d and g, and we are done.

Prim's Algorithm

Get spanning tree by connecting nodes with their parents:



Runtime for Prim's Algorithm

```
MST-Prim(G,w,r)                          ; Graph G, weights w, root r
        Q ← V[G]
        for each vertex u ∈ Q do key[u] ← ∞      ; infinite "distance"
        key[r] ← 0
        P[r] ← NIL
        while Q<>NIL do
                u ← Extract-Min(Q)               ; remove closest node
                ; Update children of u so they have a parent and a min key val
                ; the key is the weight between node and parent
                for each v ∈ Adj[u] do
                        if v ∈ Q & w(u,v)<key[v] then
                                P[v] ← u
                                key[v] ← w(u,v)
```

O(V) if using a heap

O(lgV) if using a heap

O(V)

O(E) over entire while(Q<>NIL) loop

O(lgV) to update if using a heap!

The inner loop takes O(E lg V) for the heap update inside the O(E) loop.
This is over all executions, so it is not multiplied by O(V) for the while loop
(this is included in the O(E) runtime through all edges.

The Extract-Min requires O(V lg V) time.
O(lg V) for the Extract-Min and O(V) for the while loop.

Total runtime is then O(V lg V) + O(E lg V) which is O(E lg V)
in a connected graph
(a connected graph will always have at least V-1 edges).

Prim's Algorithm – Linear Array for Q

- What if we use a simple linear array for the queue instead of a heap?
  - Use the index as the vertex number
  - Contents of array as the distance value
  - E.g.

Val[10 5 8 3 …]
Par[6 4 2 7 …]

Says that vertex 1 has key = 10, vertex 2 has key = 5, etc.
Use special value for infinity or if vertex removed from the queue
Says that vertex 1 has parent node 6, vertex 2 has parent node 4, etc.

Building Queue: O(n) time to create arrays
Extract min: O(n) time to scan through the array
Update key: O(1) time

Runtime for Prim's Algorithm with Queue as Array

```
MST-Prim(G,w,r)                          ; Graph G, weights w, root r
        Q ← V[G]
        for each vertex u ∈ Q do key[u] ← ∞      ; infinite "distance"
        key[r] ← 0
        P[r] ← NIL
        while Q<>NIL do
                u ← Extract-Min(Q)               ; remove closest node
                ; Update children of u so they have a parent and a min key val
                ; the key is the weight between node and parent
                for each v ∈ Adj[u] do
                        if v ∈ Q & w(u,v)<key[v] then
                                P[v] ← u
                                key[v] ← w(u,v)
```

O(V) to initialize array

O(V) to search array

O(V)

O(E) over entire while(Q<>NIL) loop

O(1) direct access via array index

The inner loop takes O(E) over all iterations of the outer loop.
It is not multiplied by O(V) for the while loop.

The Extract-Min requires O(V) time.
This is $O(V^2)$ over the while loop.

Total runtime is then $O(V^2) + O(E)$ which is $O(V^2)$

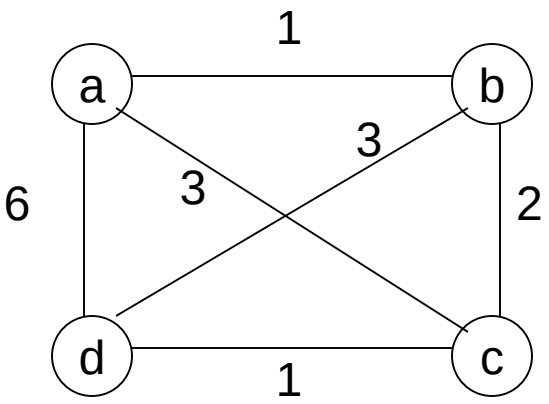
Using a heap our runtime was O(E lg V). **Which is worse?**
Which is worse for a fully connected graph?

Approximations for Hard Problems

- Greedy algorithms are commonly used to find approximations for NP-Complete problems
- Use a heuristic to drive the greedy selection
  - Heuristic: A common-sense rule that approximates moves toward the optimal solution
- If our problem is to minimize a function f where
  - f(s*) is the value of the exact solution; global minimum
  - $f(s_a)$ is the value of our approximate solution
  - Then we want to minimize the ratio:
    - r(sa) = $f(s_a)$ / f(s*) such that this approaches 1
- Opposite if maximizing a function (f(s*)/f(sa) maximizing the objective function)
- The approximation called is called c approximate if r(sa) ≤ c
- The smallest c that holds for all instances of the problem is called the performance ratio

Example: Traveling Salesman Problem

- Cheap greedy solution to the TSP:
  - Choose an arbitrary city as the start
  - Visit the nearest unvisited city; repeat until all cities have been visited
  - Return to the starting city
- Example graph:



Starting at a: a->b->c->d->a
Total = 10

Optimal: 8 a->b->d->c->a

$r(s_a) = 10/8 = 1.25$

Is this a good approach? What if a->d = 999999?

Greedy TSP

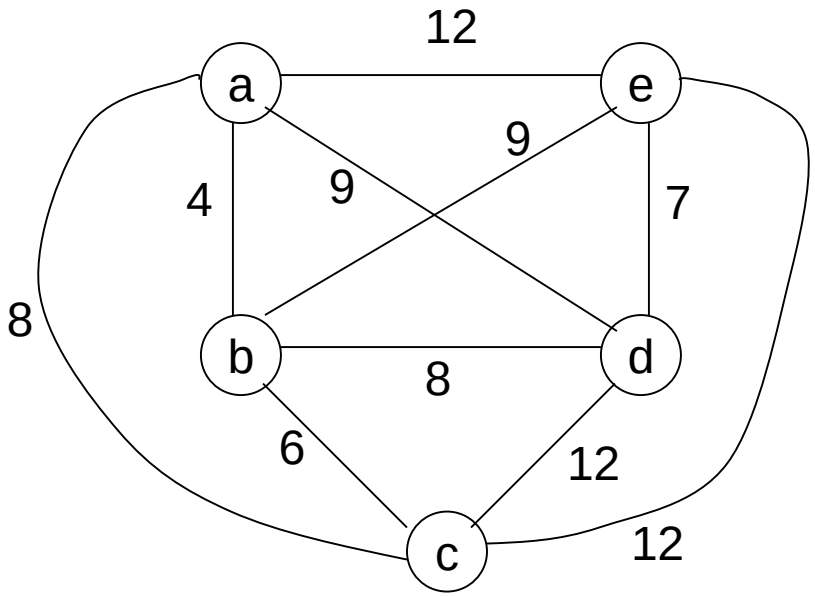
- Our greedy approach is not so bad if the graph adheres to Euclidean geometry
  - Triangle inequality
    - $d[i,j] \leq d[i,k] + d[k,j]$ for any triple cities i,j,k
  - Symmetry
    - $d[i,j] = d[j,i]$ for any pair of cities i,j
- In our previous example, we couldn't have a one-way edge to a city of 999999 where all the other edges are smaller (if a city is far away, forced to visit it some way)
- It has been proven for Euclidean instances the nearest neighbor algorithm:
  - $f(s_a) / f(s^*) \leq (\lg n + 1) / 2$ $n \geq 2$ cities

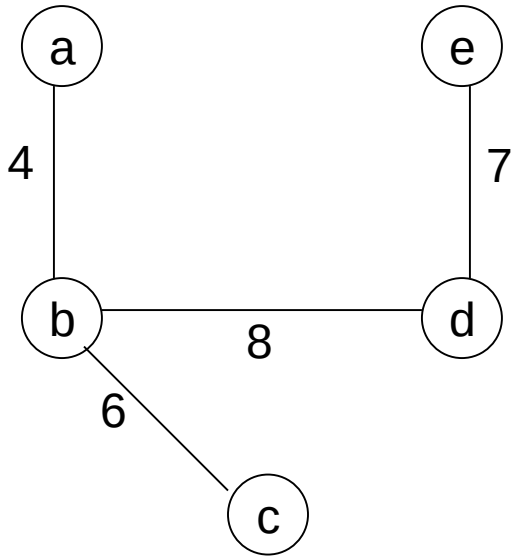
Minimum Spanning Tree Approximation

- We can use a MST to get a better approximation to the TSP problem
- This is called a twice-around-the-tree algorithm
- We construct a MST and “fix” it up so that it makes a valid tour
  - Construct a MST of the graph corresponding to the TSP problem
  - Starting at an arbitrary vertex, perform a DFS walk around the MST recording the vertices passed by
  - Scan the list of vertices from the previous step and eliminate all repeat occurrences except the starting one. The vertices remaining will form a Hamiltonian circuit that is the output of the algorithm.

MST Approximation to TSP

- Example graph:





MST: ab, bc, bd, de

Walk: a, b, c, b, d, e, d, b, a → a, b, c, d, e, a

MST Approximation

- Runtime: polynomial (Kruskal/Prim)
- Claim:
  - $f(s_a) < 2f(s^*)$
  - Length of the approximation solution at most twice the length of the optimal
- Since removing any edge from s* yields a spanning tree T of weight w(T) that must be ≥ w(T*), the weight of the graph's MST, we have:
  - $f(s^*) > w(T) \geq w(T^*)$
  - $2f(s^*) > 2w(T^*)$
- The walk of the MST tree we used to generate the approximate solution traversed the MST at most twice, so:
  - $2w(T^*) > f(s_a)$
- Giving:
  - $2f(s^*) > 2w(T^*) > f(s_a)$
  - $2f(s^*) > f(s_a)$

Knapsack Problem

- One wants to pack *n* items in a luggage
  - The *i*th item is worth v_i dollars and weighs w_i pounds
  - Maximize the value but cannot exceed *W* pounds
  - v_i , w_i, *W* are integers
- 0-1 knapsack → each item is taken or not taken
- Fractional knapsack → fractions of items can be taken
- Both exhibit the optimal-substructure property
  - 0-1: If item *j* is removed from an optimal packing, the remaining packing is an optimal packing with weight at most $W-w_j$
  - Fractional: If *w pounds* of item *j* is removed from an optimal packing, the remaining packing is an optimal packing with weight at most *W-w* that can be taken from other *n-1* items plus $w_j - w$ of item *j*

Greedy Algorithm for Fractional Knapsack problem

- Fractional knapsack can be solvable by the greedy strategy
  - Compute the value per pound v_i/w_i for each item
  - Obeying a greedy strategy, take as much as possible of the item with the greatest value per pound.
  - If the supply of that item is exhausted and there is still more room, take as much as possible of the item with the next value per pound, and so forth until there is no more room
  - O(n lg n) (we need to sort the items by value per pound)

O-1 knapsack is harder!

- 0-1 knapsack cannot be solved by the greedy strategy
  - Unable to fill the knapsack to capacity, and the empty space lowers the effective value per pound of the packing
  - We must compare the solution to the sub-problem in which the item is included with the solution to the sub-problem in which the item is excluded before we can make the choice
- Dynamic Programming

Fractional knapsack

- N=5, W=100

w	10	20	30	40	50
v	20	30	66	40	60
v/w	2.0	1.5	2.2	1.0	1.2

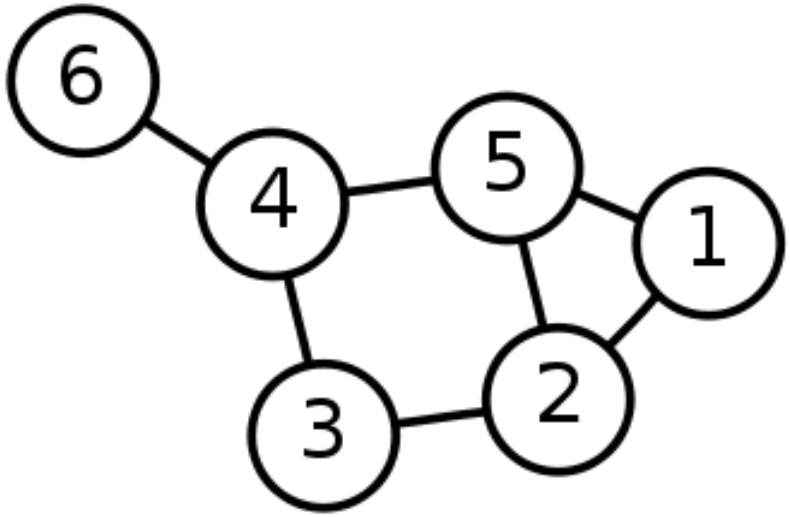
Fractional knapsack

- N=5, W=100

Select	xi					Value
Max vi	0	0	1	0.5	1	146
Min wi	1	1	1	1	0	156
Max vi/wi	1	1	1	0	0.8	164

Single-Source Shortest Path Problem

Single-Source Shortest Path Problem - The problem of finding shortest paths from a source vertex *v* to all other vertices in the graph.



Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Approach: Greedy

Input: Weighted graph G={E,V} and source vertex $v \in V$, such that all edge weights are nonnegative

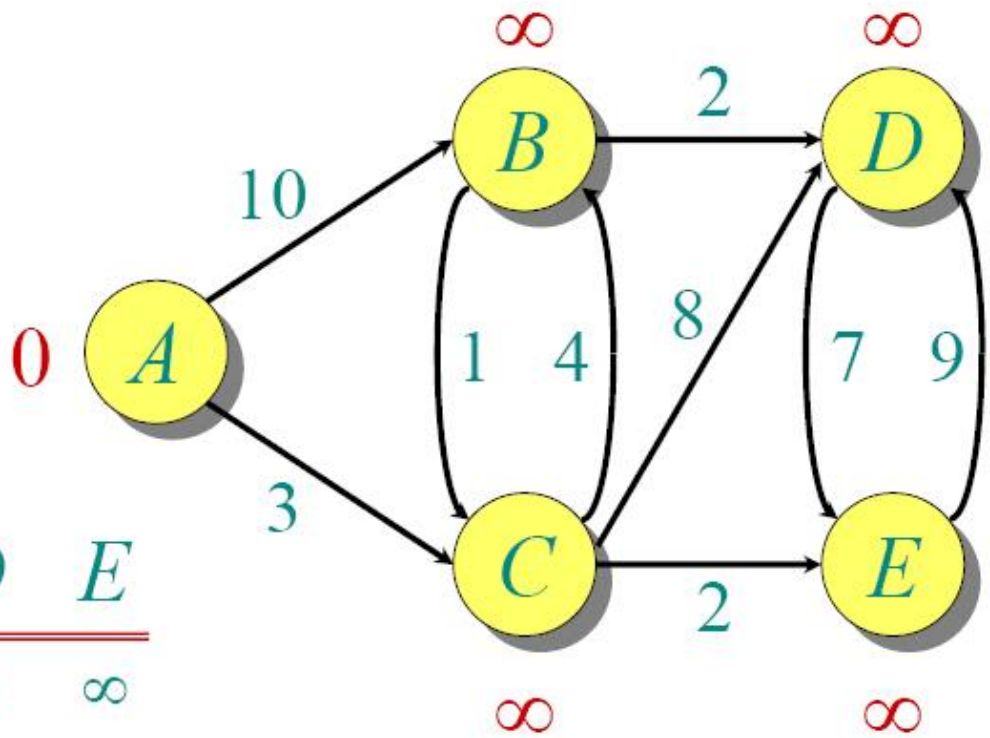
Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

Dijkstra's algorithm - Pseudocode

```
dist[s] ←o                          (distance to source vertex is zero)
for  all v ∈ V–{s}
       do  dist[v] ←∞                (set all other distances to infinity)
S←∅                                 (S, the set of visited vertices is initially empty)
Q←V                                 (Q, the queue initially contains all
vertices)
while Q ≠∅                          (while the queue is not empty)
do   u ← mindistance(Q,dist)        (select the element of Q with the min.
distance)
     S←S∪{u}                        (add u to list of visited vertices)
      for all v ∈ neighbors[u]
            do  if   dist[v] > dist[u] + w(u, v)          (if new shortest path found)
                  then      d[v] ←d[u] + w(u, v)      (set new value of shortest path)
                        (if desired, add traceback code)
return dist
```

Dijkstra Animated Example

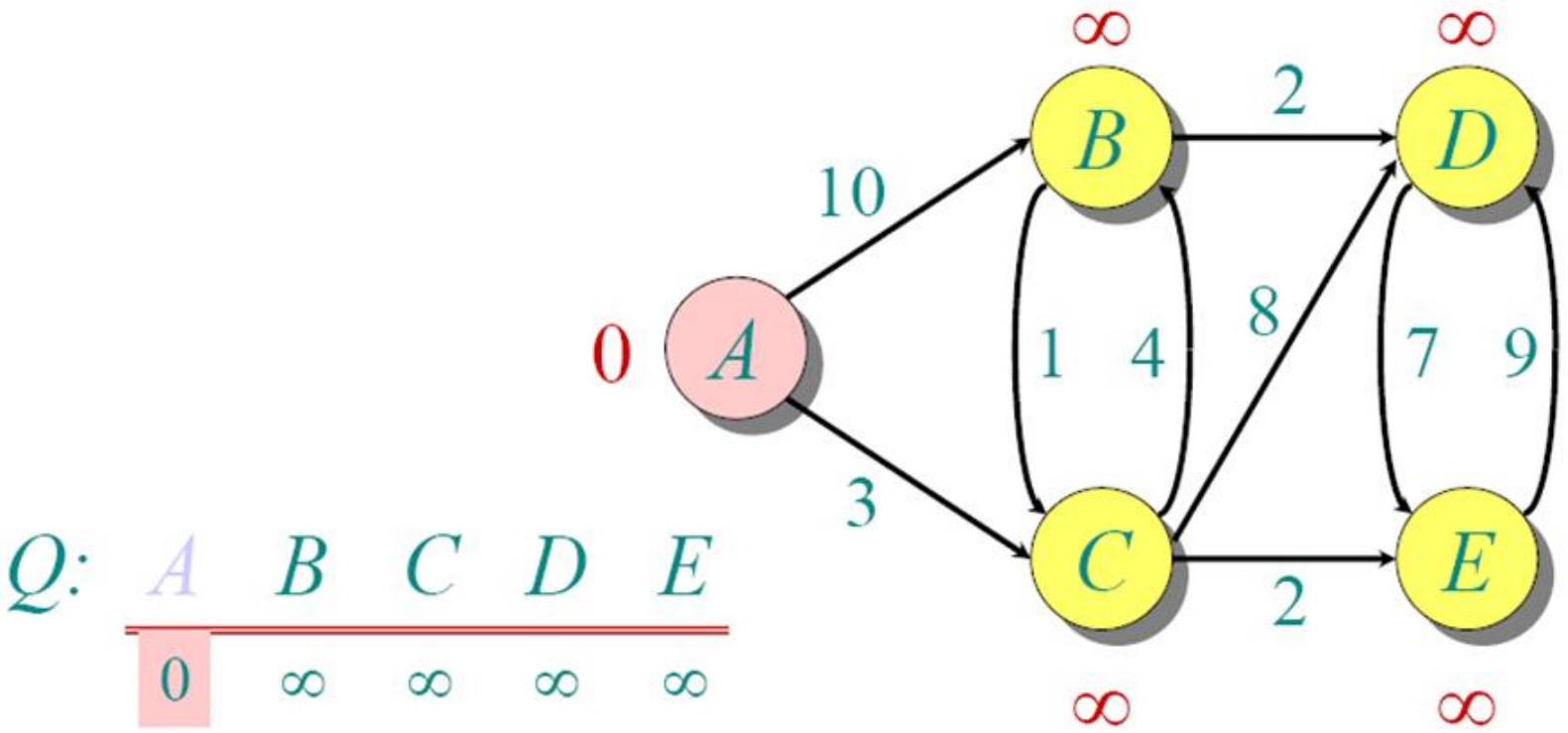
Initialize:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

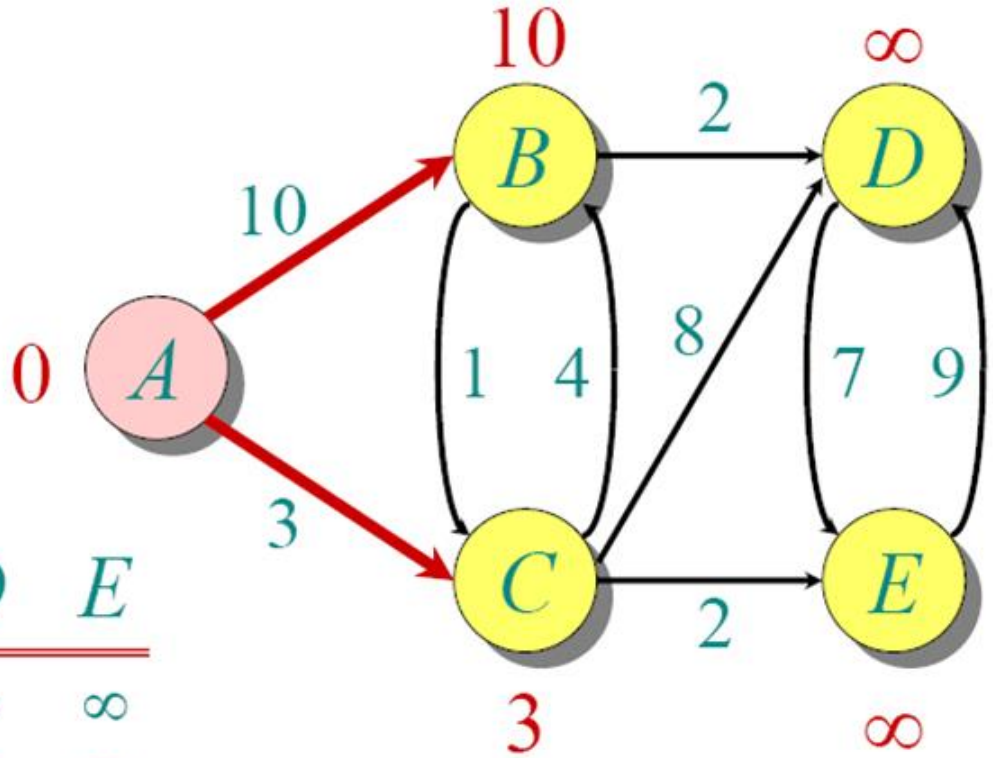
S: {}

Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

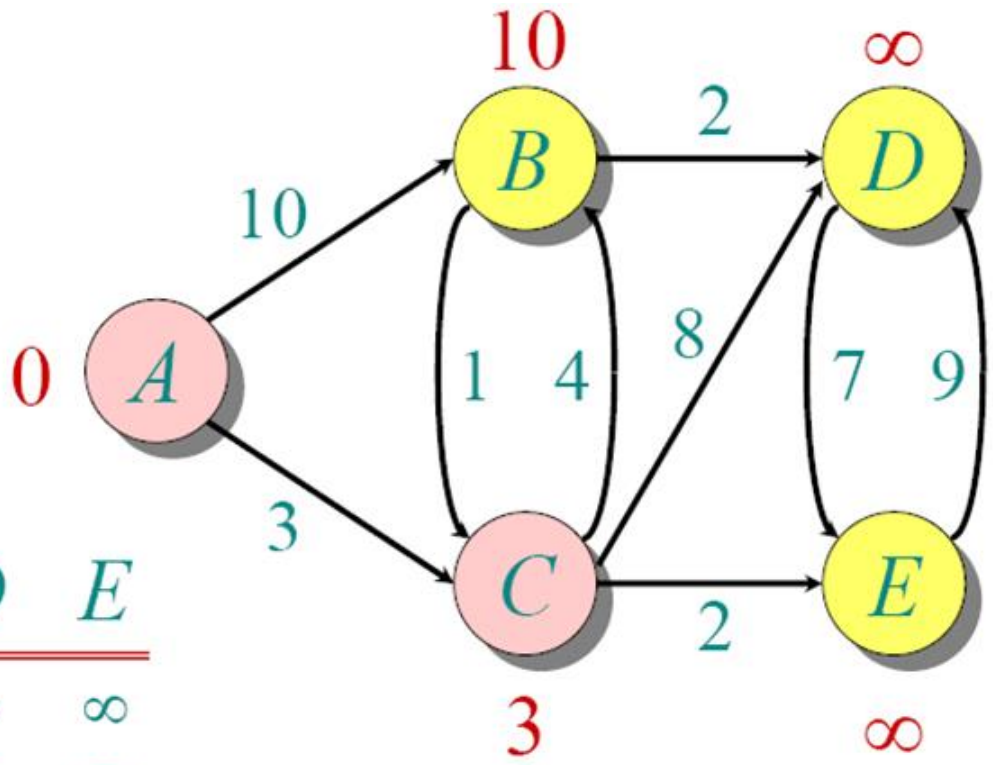
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

$S: \{ A \}$

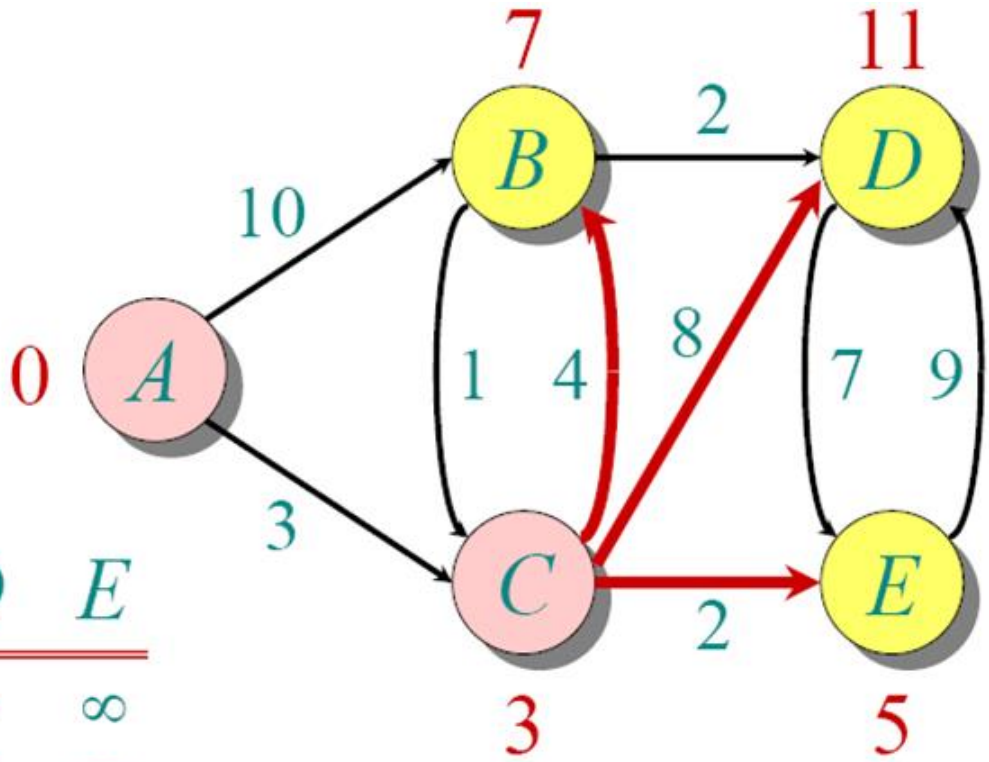
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

$S: \{ A, C \}$

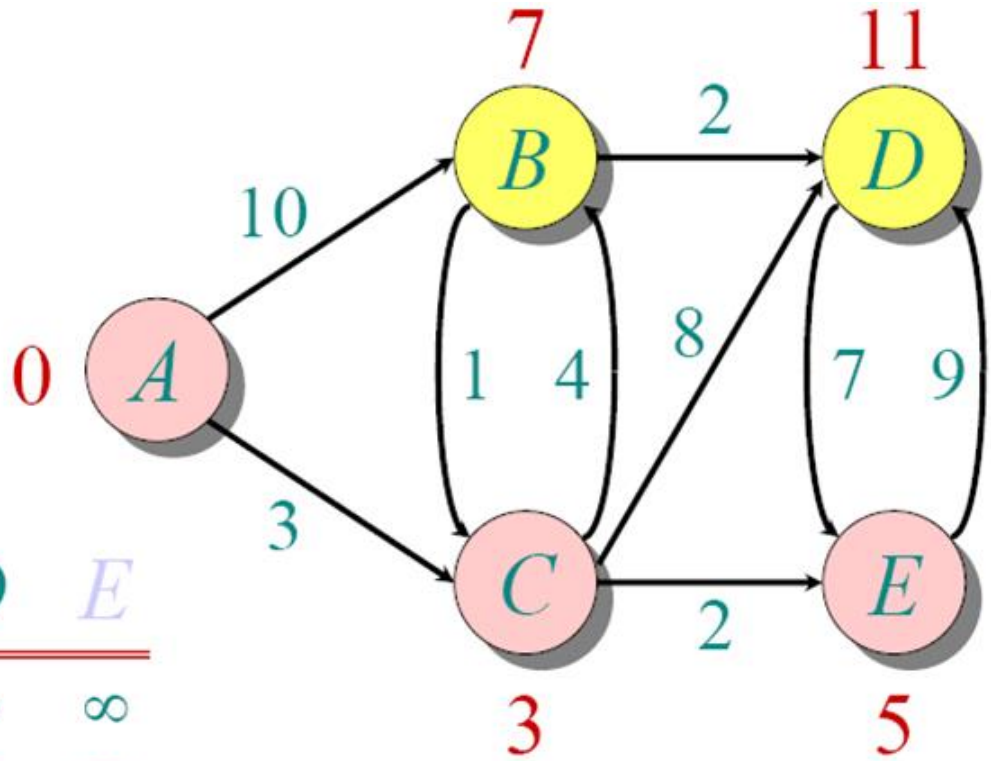
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

$S: \{ A, C \}$

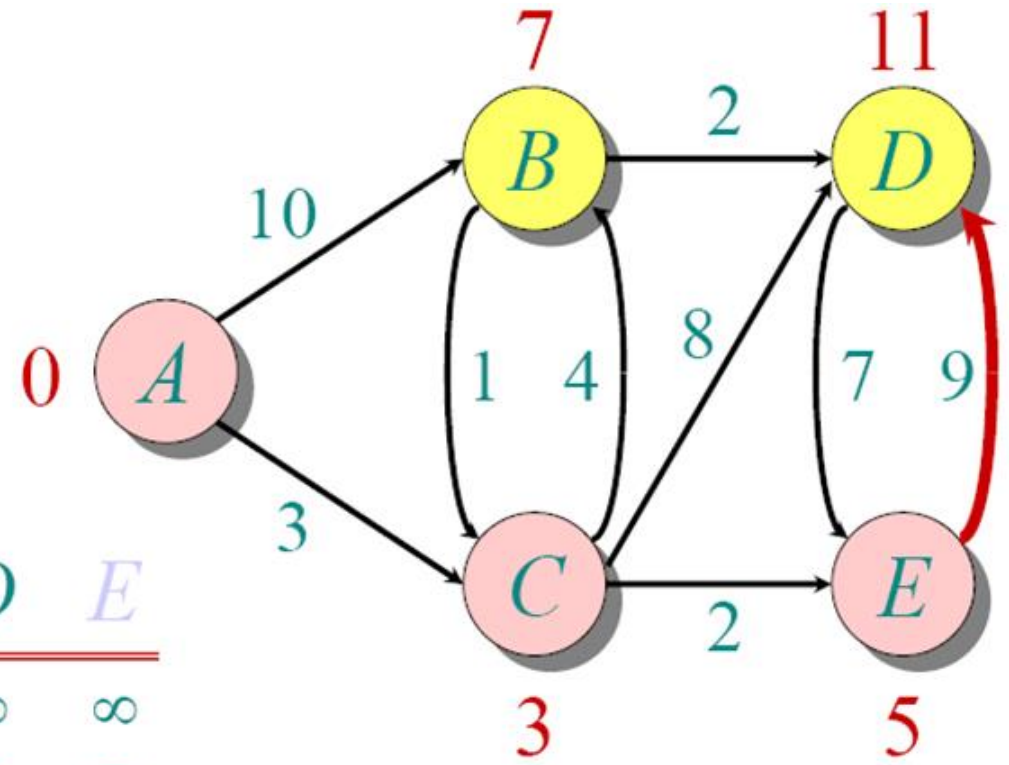
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

$S: \{ A, C, E \}$

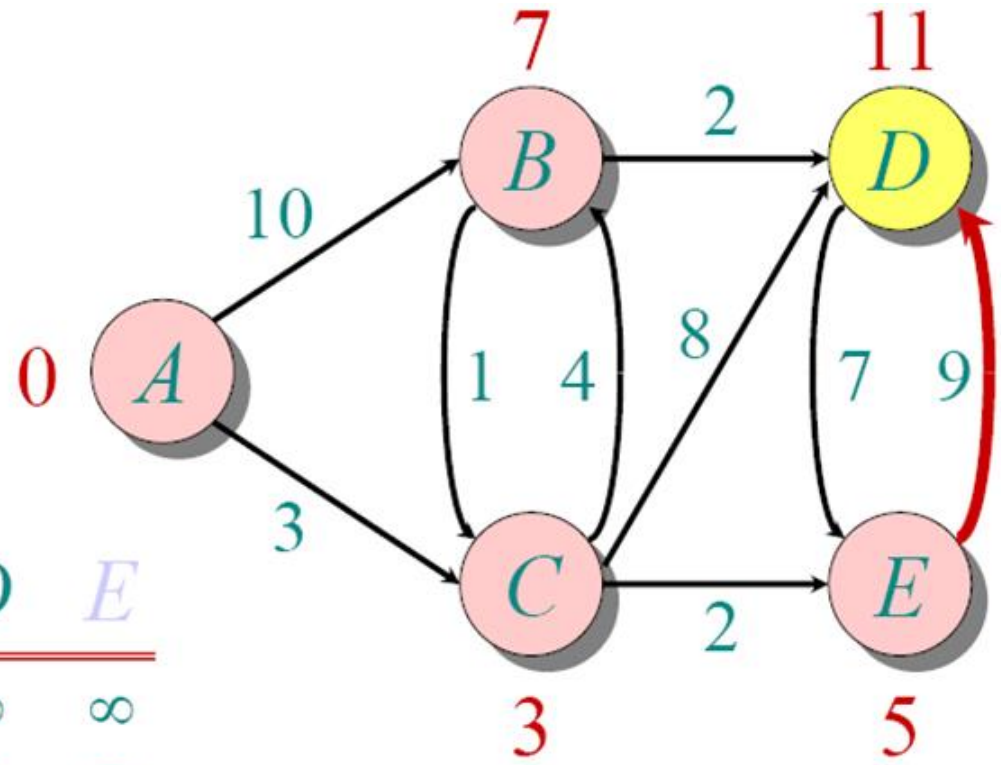
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

$S: \{ A, C, E \}$

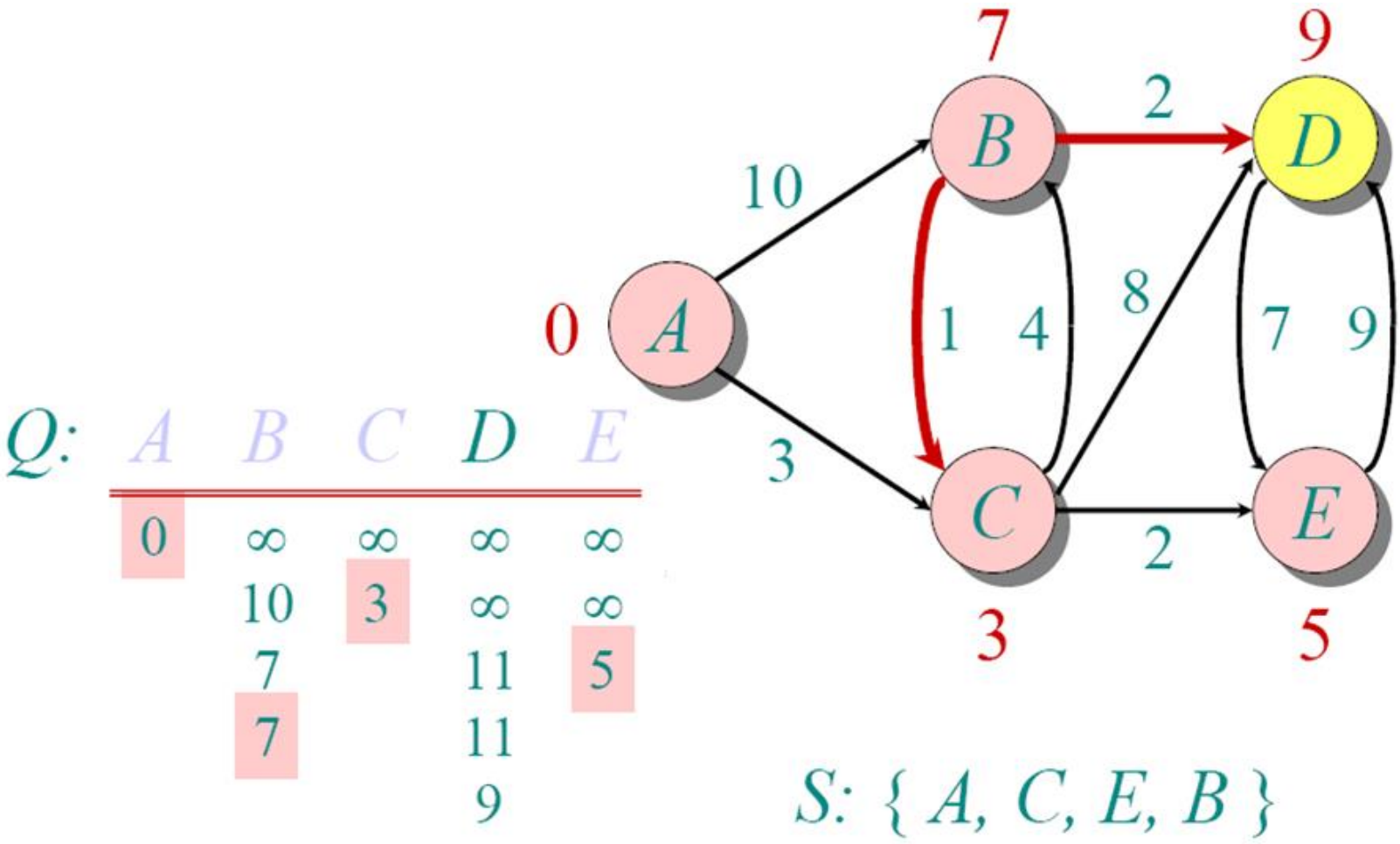
Dijkstra Animated Example



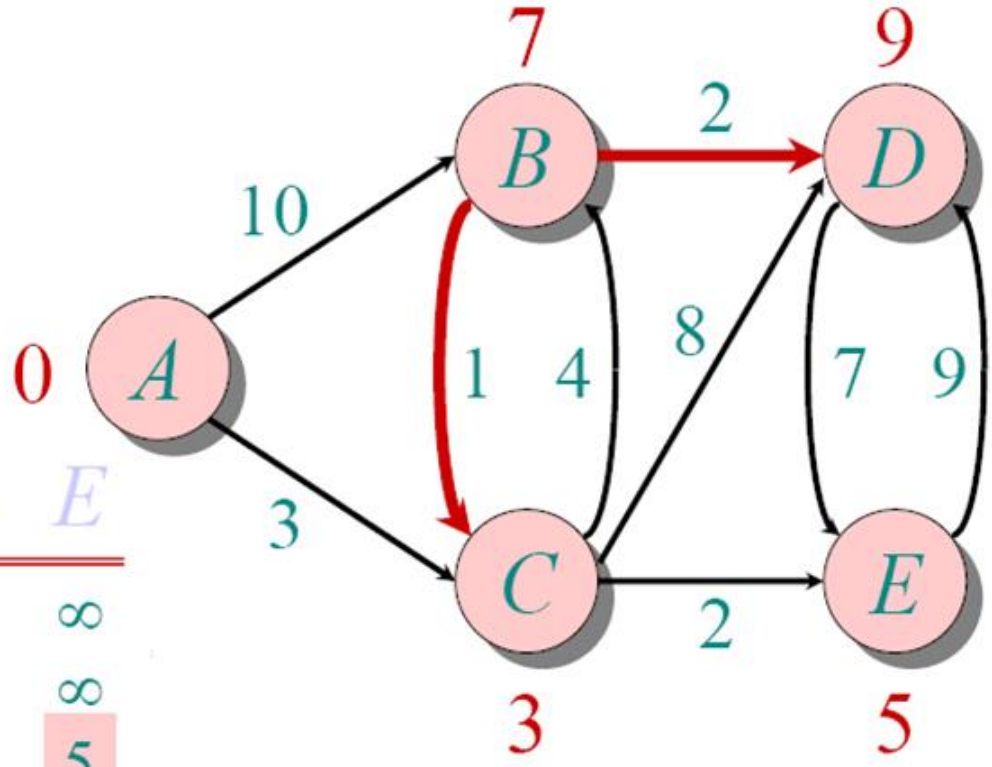
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

Dijkstra Animated Example



Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B, D }

Implementations and Running Times

The simplest implementation is to store vertices in an array or linked list. This will produce a running time of

$O(|V|^2 + |E|)$

For sparse graphs, or graphs with very few edges and many nodes, it can be implemented more efficiently storing the graph in an adjacency list using a binary heap or priority queue. This will produce a running time of

$O((|E|+|V|) \log |V|)$

Dijkstra's Algorithm - Why It Works

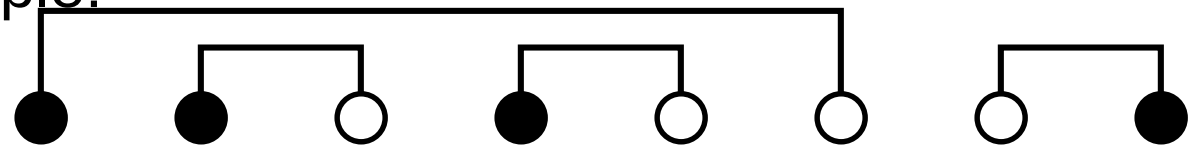
- To understand how it works, we'll go over the previous example again. However, we need two mathematical results first:

- **Lemma 1**: Triangle inequality
  If $\delta(u,v)$ is the shortest path length between u and v,
  $\delta(u,v) \leq \delta(u,x) + \delta(x,v)$
- **Lemma 2**:
  The subpath of any shortest path is itself a shortest path.

- The key is to understand why we can claim that anytime we put a new vertex in S, we can say that we already know the shortest path to it.
- Now, back to the example…

Connecting wires

- There are n white dots and n black dots, equally spaced, in a line
- You want to connect each white dot with some one black dot, with a minimum total length of "wire"
- Example:



- Total wire length above is 1 + 1 + 1 + 5 = 8
- Do you see a greedy algorithm for doing this?
- Does the algorithm guarantee an optimal solution?
  - Can you prove it?
  - Can you find a counterexample?

Connecting wires

- A Wire Connections game board consists of a list of n white dots and n blacks dots, interleaved in an arbitrary order. For example:

W W B W B B

- In the game, players take turns connecting a white dot to a black dot. When the game is over, every white dot on the board is connected to a black dot.

- There are several different ways a game can proceed. For our example board, see three immediately:

- 2-3, 4-5, 1-6
- 1-3, 2-5, 4-6
- 1-6, 2-5, 3-4

Connecting wires

- Note that moves are always written with the position of the leftmost dot first, giving smaller-larger pairs. This means that we can connect the dots in W-B order or B-W order.

- Moves are scored based on the distance between the connected dots. The distance between adjacent dots is 1 "hop". So the move 2-3 scores one point, while 2-5 scores three.

- After the board is completely wired, the winner is the player with the lowest total number of points. For this reason, game boards usually have an even n, so that each player makes the same number of moves.

Connecting wires

- **Minimizing Wire Connections**
- How might we attack this using brute-force?
- On a given board, there are n! possible wirings. We could iterate through them all, scoring each, and choose the best move from the best one. This algorithm is O(n!), or O(nn)

Connecting wires

- Given the adversarial nature of the game, we need a way to score board quickly, seeing what future moves we could be forced to make. Here is a greedy approach: Scan the line, making as many minimal connections on each pass. Continue until all dots are connected.

- How well does the greedy approach work on this board?

- W W B B W W B B
- It generates 2-3, 4-5, 6-7, 1-8. The total length of the connections is 1+1+1+7 = 10, and we would win the game, two to eight.

- Our opponent might choose a different second move, though, say, 1-4. That would leave the board as

- W W B B W W B B
- Then we could play 6-7, force our opponent to play 5-8, and win two points to six. Notice that the total length of this wiring is only 8