Analysis of Algorithms
CECS 528

Topic 7.1. Greedy algorithm

Greedy algorithms

GA are characterized by the following two properties:

- the algorithm works in stages, and during each stage a selection is made which is locally optimal;
- the sum totality of all the locally optimal choices produces a globally optimal solution.

A greedy method does not always lead to a globally optimal solution.

They are referred to as will refer to it as a heuristic, or a greedy heuristic.

These are algorithms that provide a "short cut" to a solution, but not necessarily to an optimal solution.

The term "greedy algorithm" is used for a greedy method that always produces a correct/optimal solution.

Problems

Some problems that can be solved using a greedy algorithm:

- Minimum Spanning Tree: finding a spanning tree for a graph whose edges have minimal weight.
- Fractional Knapsack: selecting what goods to load into a shipping container.
- Activity Selection: finding a maximum set of activities (each with start and end times) which can be hosted in an auditorium.
- Huffman Coding: finding a code for a set of words which minimizes the expected code-length.
- Optimal List Storage.

Example 1

Optimal List Storage: given a set of items $I_1, \ldots, I_n$ where item I_i

1. has a length l_i

2. a probability of access p_i

find an ordering of the items in a list so that, upon scanning the list from front to back for a randomly chosen item, the expected traversal length is minimized.

Example 2

Unit Task Scheduling with Deadlines: given a set of tasks $t_1, \ldots,$ t_n, where task t_i

1. takes unit time to complete

2. has a deadline $d_i > 0$ associated with it

3. a profit p_i associated with it only if t_i is completed before d_i

select a subset of the tasks to complete so as to maximize the total profit.

Example 3.

Distance between graph vertices: Given two graph vertices *u* and *v* of a graph G, find the minimum-cost path from *u* to *v*.

Example 4

Minimum Spanning Trees

Consider a problem in which roads are to be built that connect all four cities a, b, c, and d to one another. In other words, after the roads are built, it will be possible to drive from any one city to another. The costs of building a road between any two cities are given.

Find a set of roads of minimum cost that will connect the cities.

Activity selection

Given a set of *activities A* of length *n*

$A = \langle a_1, a_2, ..., a_n \rangle$

with *starting times*

$S = \langle s_1, s_2, ..., s_n \rangle$

and *finishing times*

$F = \langle f_1, f_2, ..., f_n \rangle$

such that $0 \leq s_i < f_i < \infty$, we define two activities a_i and a_j to be *compatible* if $f_i \leq s_j$ **or** $f_j \leq s_i$

i.e. one activity ends before the other begins so they do not overlap.

Task

Find a *maximal* set of compatible activities, e.g. scheduling the most activities in a lecture hall.

Note that we want to find the maximum *number* of activities, **not** necessarily the maximum *use* of the resource.

Dynamic Programming Solution

Step 1: Characterize optimality

Without loss of generality, we will assume that the activities, a, are sorted in non-decreasing order of finishing times, i.e. $f_1 \leq f_2 \leq ... \leq f_n$.

Define the set S_{ij}

$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$

as the subset of activities that can occur between the completion of a_i (f_i) and the start of a_j (s_j).

Step 1 – Note

Note that $S_{ij} = \emptyset$ for $i \geq j$ since otherwise $f_i \leq s_j < f_j \Rightarrow f_i < f_j$ which is a contradiction for $i \geq j$ by the assumption that the activities are in sorted order.

Furthermore let A_{ij} be the *maximal* set of activities for S_{ij}. Using a "cut-and-paste" argument, if A_{ij} contains activity a_k then we can write

$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$

where A_{ik} and A_{kj} must also be optimal (otherwise if we could find subsets with more activities that were still compatible with a_k then it would contradict the assumption that A_{ij} was optimal).

Step 2

Step 2: Define the recursive solution (top-down)

Let $c[i,j] = |A_{ij}|$, then

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{\substack{i<k<j \\ a_k \in S_{ij}}} (c[i,k] + 1 + c[k,j]) & \text{if } S_{ij} \neq \emptyset \end{cases}$$

i.e. compute $c[i,j]$ for each $k = i+1, \ldots, j-1$ and select the max.

Step 3

Step 3: Compute the maximal set size (bottom-up)

Construct an *n*x*n* table which can be done in polynomial time since clearly for each *c*[*i*,*j*] we will examine no more than *n* subproblems giving an upper bound on the worst case of $O(n^3)$.

BUT WE DON'T NEED TO DO ALL THAT WORK! Instead at each step we could simply select (*greedily*) the activity that finishes first and is compatible with the previous activities. Intuitively this choice leaves the most time for other future activities.

Greedy Algorithm Solution

To use the greedy approach, we must *prove* that the greedy choice produces an optimal solution (although not necessarily the *only* solution).

Consider any non-empty subproblem S_{ij} with activity a_m having the earliest finishing time, i.e.

$$f_{min} = \min\{f_k : a_k \in S_{ij}\}$$

then the following two conditions must hold

1. a_m is used in an optimal subset of S_{ij}
2. $S_{im} = \emptyset$ leaving S_{mj} as the only subproblem

meaning that the greedy solution produces an optimal solution.

Proof

Let A_{ij} be an optimal solution for S_{ij} and a_k be the first activity in A_{ij}

$\rightarrow$ If $a_k = a_m$ then the condition holds.

$\rightarrow$ If $a_k \neq a_m$ then construct $A_{ij}' = A_{ij} - \{a_k\} \cup \{a_m\}$.

Since $f_m \leq f_k \Rightarrow A_{ij}'$ is still optimal.

If S_{im} is non-empty $\Rightarrow a_k$ with

$f_i \leq s_k < f_k \leq s_m < f_m \Rightarrow f_k < f_m$ which contradicts the assumption that f_m is the minimum finishing time. Thus $S_{im} = \emptyset$.

Thus instead of having 2 subproblems each with n-j-1 choices per problem, we have reduced it to 1 subproblem with 1 choice.

Algorithm

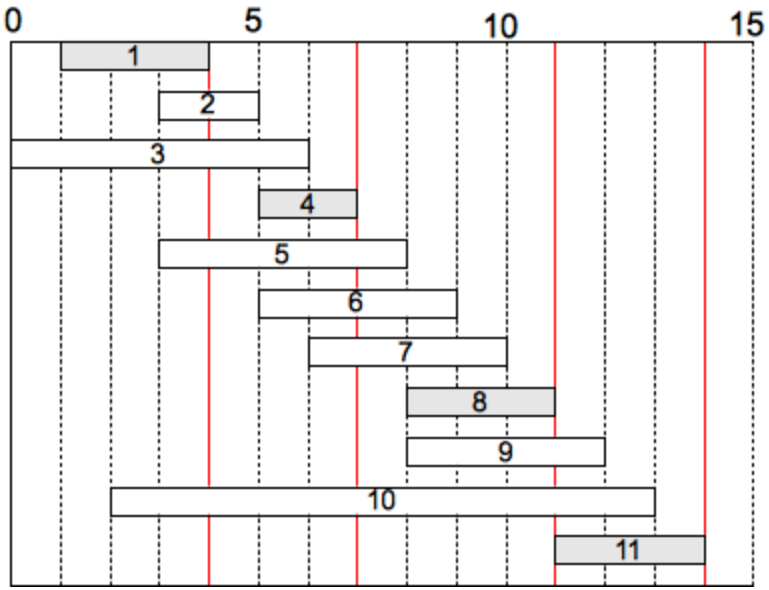
Always start by choosing the first activity (since it finishes first), then repeatedly choose the next compatible activity until none remain.

The algorithm can be implemented either recursively or iteratively in O(n) time (assuming the activities are sorted by finishing times) since each activity is examined only once.

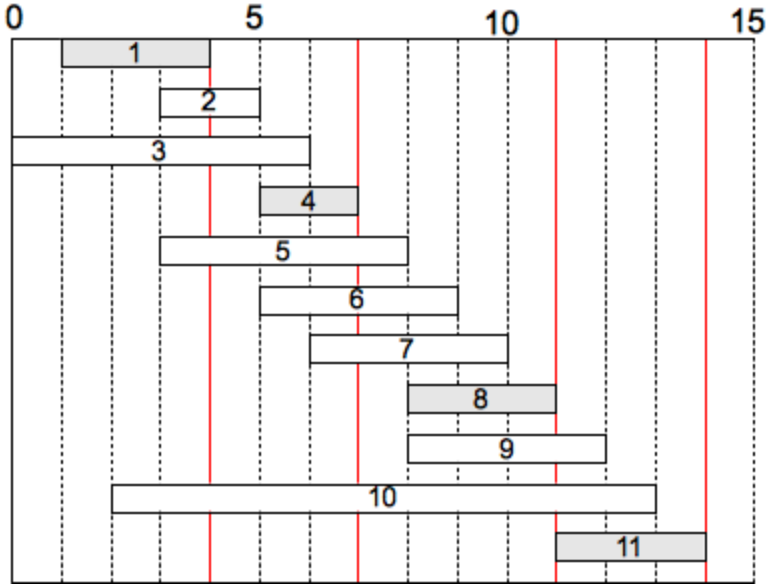
Example

Consider the following set of activities represented graphically in non-decreasing order of finishing 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	9	9	10	11	12	14	16



Example



Using the greedy strategy an optimal solution is $\{1, 4, 8, 11\}$.

Note another optimal solution not produced by the greedy strategy is $\{2, 4, 8, 11\}$.

Recursive greedy algorithm

The procedure RECURSIVE-ACTIVITY-SELECTOR (R-A-S) takes the start and finish times of the activities, represented as arrays s and f, the index k that defines the subproblem S_k it is to solve, and the size n of the original problem.

It returns a maximum-size set of mutually compatible activities in S_k.

In order to start, we add the fictitious activity a_0 with $f_0 = 0$, so that subproblem S_0 is the entire set of activities S.

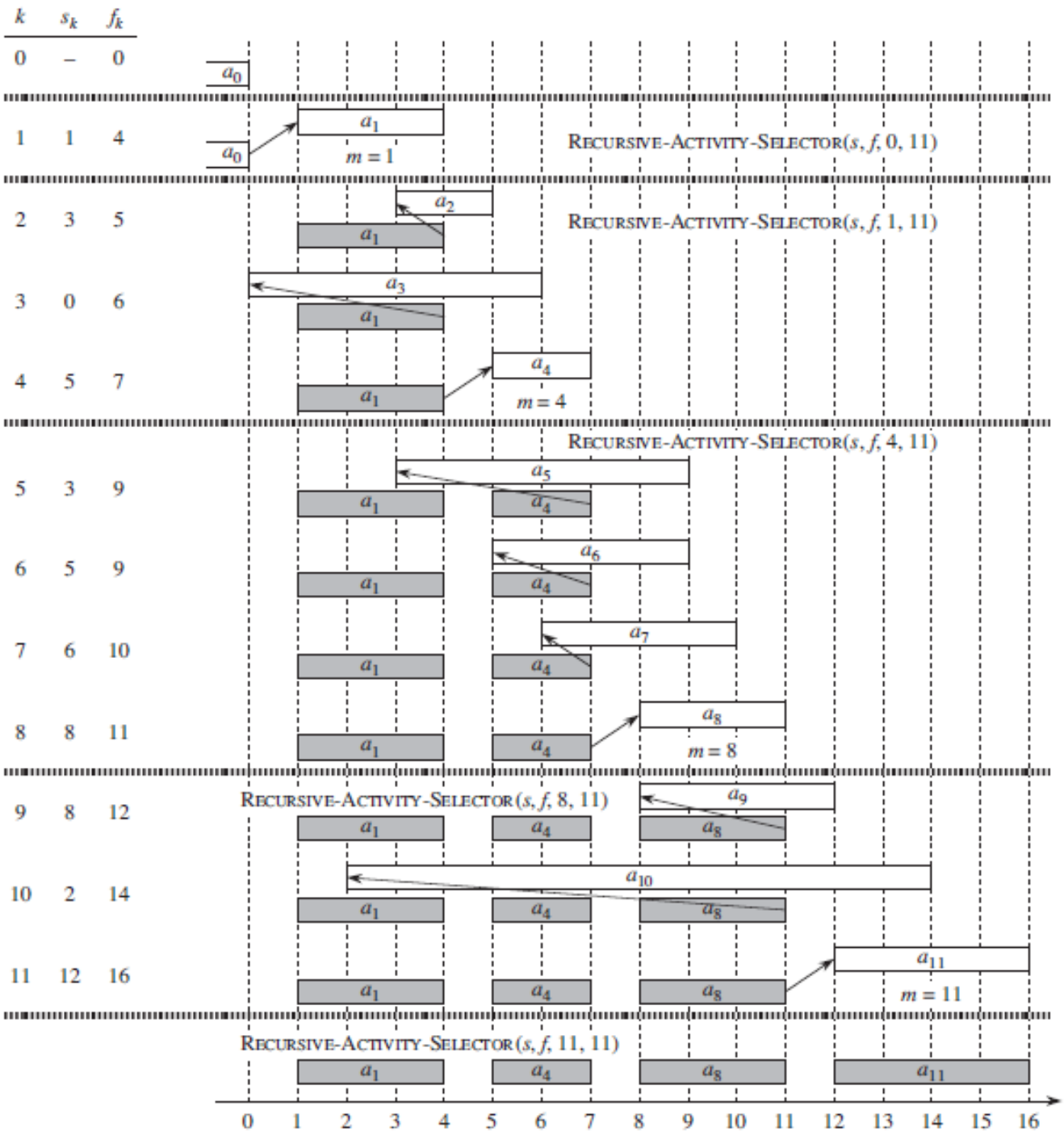
The initial call, which solves the entire problem, is

RECURSIVE-ACTIVITY-SELECTOR($s, f, 0, n$).

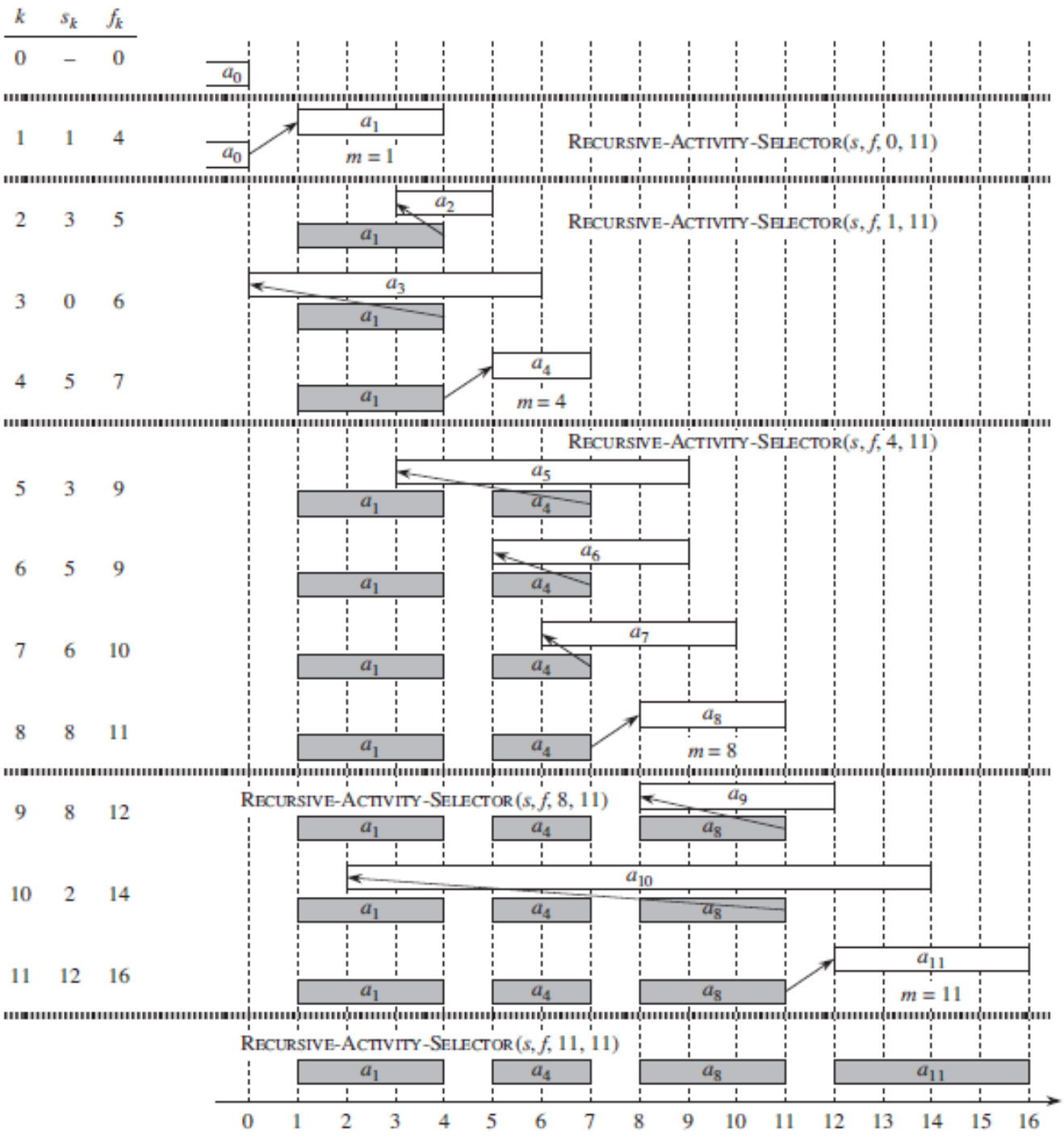
Recursive algorithm

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

Activities considered in each recursive call appear between horizontal lines.
The fictitious activity a_0 finishes at time 0, and the initial call R-A-S(s,f,0,11) , selects activity a1.
In each recursive call, the activities that have already been selected are shaded, and the activity shown in white is being considered.



If the starting time of an activity occurs before the time of the most recently added activity (the arrow between them points left), it is rejected. Otherwise (the arrow points directly up or to the right), it is selected. The last recursive call, R-A-S(s, f, 11, 11), returns NULL. The resulting set of selected activities is (a1, a4, a8, a11).



The running time

Assuming that the activities have already been sorted by finish times, the running time of the call RECURSIVE-ACTIVITY-SELECTOR($s, f, 0, n$) is $\Theta(n)$:

Over all recursive calls, each activity is examined exactly once in the **while** loop test of line 2. In particular, activity a_i is examined in the last call made in which $k < i$.

An iterative greedy algorithm

- The recursive procedure can be easily converted to an iterative one.
- The procedure RECURSIVE-ACTIVITY-SELECTOR is almost "tail recursive" : it ends with a recursive call to itself followed by a union operation. It is usually a straightforward task to transform a tail-recursive procedure to an iterative form; in fact, some compilers for certain programming languages perform this task automatically.
- As written, RECURSIVE-ACTIVITY-SELECTOR works for subproblems S_k i.e., subproblems that consist of the last activities to finish.

Iterative procedure

- The procedure GREEDY-ACTIVITY-SELECTOR is an iterative version of the procedure RECURSIVE-ACTIVITY-SELECTOR.
- It also assumes that the input activities are ordered by monotonically increasing finish time. It collects selected activities into a set A and returns this set when it is done.

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5      if s[m] ≥ f[k]
6          A = A ∪ {a_m}
7          k = m
8  return A
```

Iterative procedure

- The variable k indexes the most recent addition to A, corresponding to the activity a_k in the recursive version. Since the activities are ordered with monotonically increasing finish time, f_k is always the maximum finish time of any activity in A.
- The **for** loop finds the earliest activity in S_k to finish. It considers each activity a_m and adds it to A if it is compatible with all previously selected activities; such an activity is the earliest to finish.

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5      if s[m] ≥ f[k]
6          A = A ∪ {a_m}
7          k = m
8  return A
```

Comparison

- The set A returned by the call GREEDY-ACTIVITY-SELECTOR(s, f) is precisely the set returned by the call RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n).
- Like the recursive version, GREEDY-ACTIVITY-SELECTOR schedules a set of n activities in $\Theta(n)$ time, assuming that the activities were already sorted initially by their finish times.

Greedy Algorithm Properties

A general procedure for creating a greedy algorithm is:

1. Determine the optimal substructure (like dynamic programming)
2. Derive a recursive solution (like dynamic programming)
3. For every recursion, show *one* of the optimal solutions is the *greedy* one.
4. Demonstrate that by selecting the *greedy* choice, *all* other subproblems are *empty*.
5. Develop a recursive/iterative implementation.

Usually we try to cast the problem such that we only need to consider one subproblem and that the greedy solution to the subproblem is optimal.

Then the subproblem along with the greedy choice produces the optimal solution to the original problem.

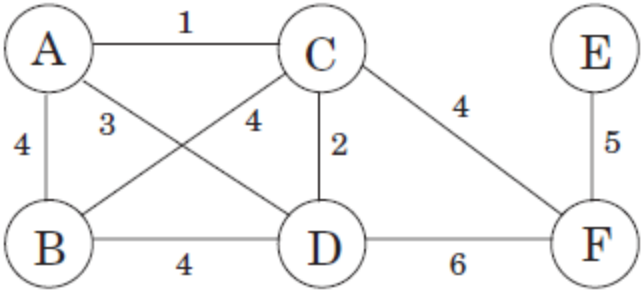
Minimum spanning trees

- Consider creating a network of a collection of computers by linking selected pairs of them.
- This translates into a graph problem in which nodes are computers, undirected edges are potential links, and the goal is to pick enough of these edges that the nodes are connected.
- Each link also has a maintenance cost, reflected in that edge's weight.
- What is the cheapest possible network?

Minimum spanning trees

- Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together.
- To interconnect a set of n pins, we can use an arrangement of n -1 wires, each connecting two pins.
- Of all such arrangements, the one that uses the least amount of wire is usually the most desirable.
- We can model this wiring problem with a connected, undirected graph $G = (V, E)$, where V is the set of pins, E is the set of possible interconnections between pairs of pins.
- For each edge $(v, u) \subseteq E$, we have a weight $w(u, v)$ specifying the cost (amount of wire needed) to connect u and v.

A graph



One immediate observation is that the optimal set of edges cannot contain a cycle, because removing an edge from this cycle would reduce the cost without compromising connectivity:

Removing a cycle edge cannot disconnect a graph.

So the solution must be connected and acyclic: undirected graphs of this kind are called *trees*.

The particular tree we want is the one with minimum total weight, known as the *minimum spanning tree*.

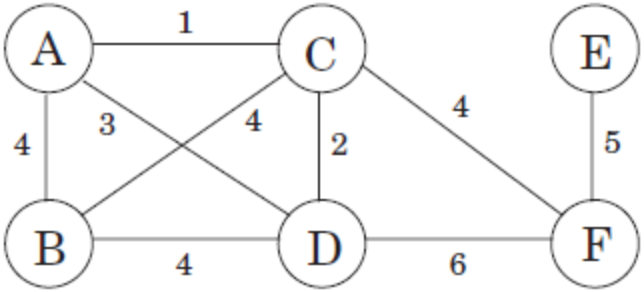
Formal definition

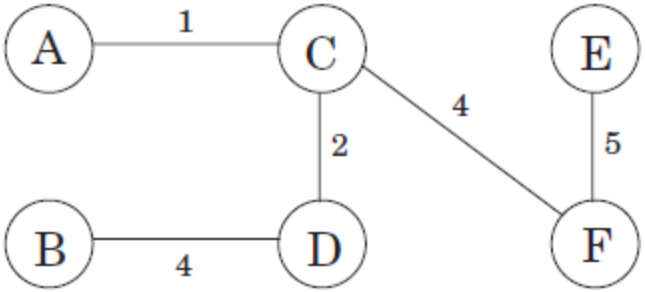
Input: An undirected graph $G = (V, E)$; edge weights w_e.

Output: A tree $T = (V, E)$, with $E' \subseteq E$, that minimizes

$$weight(T) = \sum_{e \in E'} w_e$$

The minimum spanning tree has a cost of 16:





This is not the only optimal solution. Can you spot another?

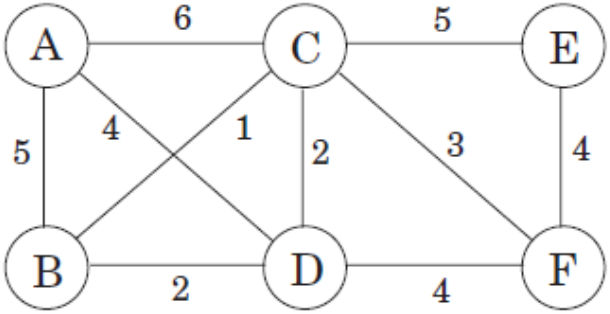
A greedy approach

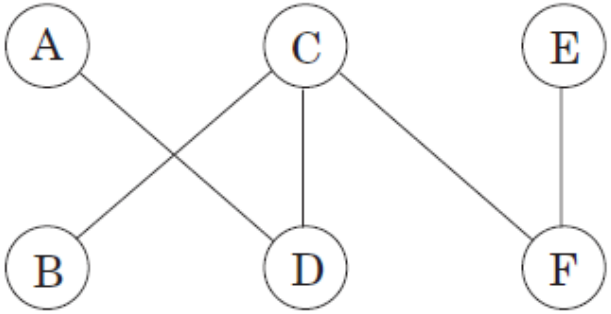
Kruskal's minimum spanning tree algorithm starts with the empty graph and then selects edges from E according to the following rule.

Repeatedly add the next lightest edge that doesn't produce a cycle.

It constructs the tree edge by edge and, apart from taking care to avoid cycles, simply picks whichever edge is cheapest at the moment.

This is a *greedy* algorithm: every decision it makes is the one with the most obvious immediate advantage.



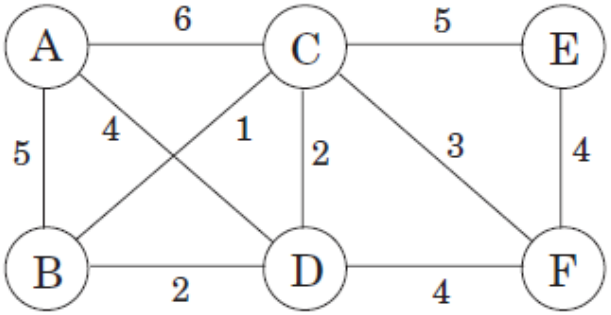


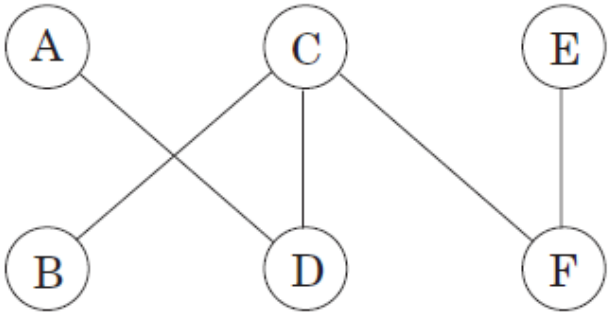
Example

- Start with an empty graph and then attempt to add edges in increasing order of weight (ties are broken arbitrarily):

$B-C,\ C-D,\ B-D,\ C-F,\ D-F,\ E-F,\ A-D,\ A-B,\ C-E,\ A-C.$

- The first two succeed, but the third, $B - D$, would produce a cycle if added.
- So we ignore it and move along. The final result is a tree with cost 14, the minimum possible.





Trees and their properties

A tree is an undirected graph that is connected and acyclic.

Property 1 ***Removing a cycle edge cannot disconnect a graph.***

Property 2 ***A tree on n nodes has n − 1 edges.***

Start from an empty graph. Initially each of the n nodes is disconnected from the others. As edges are added, these components merge.

Since each edge unites two different components, exactly $n - 1$ edges are added by the time the tree is fully formed.

When a particular edge $\{u, v\}$ comes up, u and v lie in separate connected components (no cycles!).

Adding the edge then merges these two components, thereby reducing the total number of connected components by one.

Trees and their properties

Property 3 ***Any connected, undirected graph $G = (V, E)$ with $|E| = |V| - 1$ is a tree.***

Show that G is acyclic. One way to do this is to run the following iterative procedure on it: while the graph contains a cycle, remove one edge from this cycle.

The process terminates with some graph $G = (V, E)$, $E \subseteq E$, which is acyclic and, by Property 1, is also connected.

Therefore G is a tree, whereupon $|E| = |V| - 1$ by Property 2.

So $E = E$, no edges were removed, and G was acyclic to start with.

Trees and their properties

Property 4 ***An undirected graph is a tree if and only if there is a unique path between any pair of nodes.***

In a tree, any two nodes can only have one path between them; for if there were two paths, the union of these paths would contain a cycle.

On the other hand, if a graph has a path between any two nodes, then it is connected.

If these paths are unique, then the graph is also acyclic (since a cycle has two paths between any pair of nodes).

The cut property

The correctness of Kruskal's method follows from a certain *cut property, which* justifies many other minimum spanning tree algorithms.

Suppose edges X are part of a minimum spanning tree of $G = (V, E)$.

Pick any subset of nodes S for which X does not cross between S and $V - S$, and let e be the lightest edge across this partition.

Then $X \cup \{e\}$ is part of some MST.

A cut is any partition of the vertices into two groups, S and $V - S$.

It is always safe to add the lightest edge across any cut (that is, between a vertex in S and one in $V - S$), provided X has no edges across the cut.

The cut property

A cut is any partition of the vertices into two groups, S and $V-S$.

It is always safe to add the lightest edge across any cut (that is, between a vertex in S and one in $V - S$).

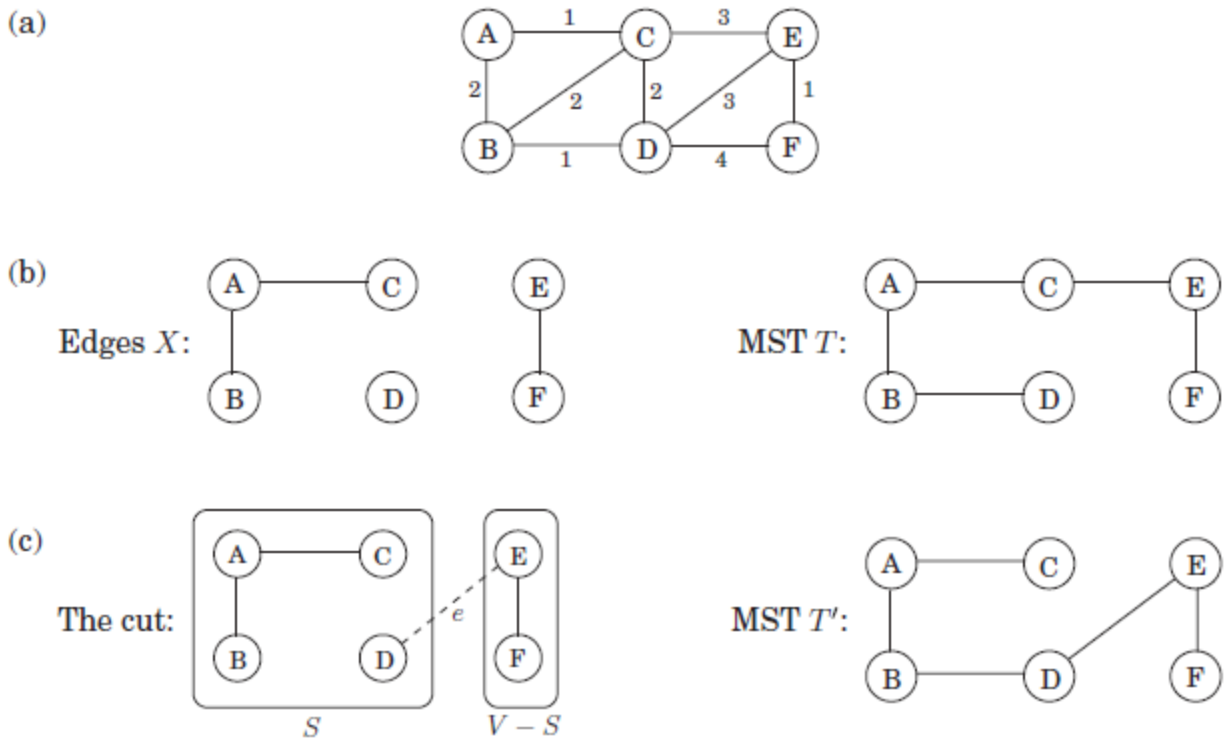
Edges X are part of some MST T.

Add edge e to T. Since T is connected, it already has a path between the endpoints of e, so adding e creates a cycle. This cycle must also have some other edge e across the cut $(S, V - S)$. What is left $T = T \cup \{e\} - \{e\}$ should be a tree.

T is connected by Property 1, since e is a cycle edge. And it has the same number of edges as T; so by Properties 2 and 3, 2 and 3, it is also a tree.

The cut property

- (a) An undirected graph. (b) Set X has three edges, and is part of the MST T on the right. (c) If $S = \{A, B, C, D\}$, then one of the minimum-weight edges across the cut $(S, V - S)$ is $e = \{D, E\}$. $X \cup \{e\}$ is part of MST T', shown on the right.



Kruskal's algorithm

- At any given moment, the edges already chosen form a partial solution, a collection of connected components each of which has a tree structure.
- The next edge e to be added connects two of these components, $T1$ and $T2$.
- Since e is the lightest edge that doesn't produce a cycle, it is certain to be the lightest edge between $T1$ and $V - T1$ and therefore satisfies the cut property.

Kruskal algorithm

- Model the algorithm's state as a collection of disjoint sets, each of which contains the nodes of a particular component. Initially each node is in a component by itself:

  makeset(x): *create a singleton set containing just x.*

- Repeatedly test pairs of nodes to see if they belong to the same set.

  find(x): *to which set does x belong?*

- Whenever an edge is added, two components are merged:

  union(x, y): *merge the sets containing x and y.*

Kruskal's minimum spanning tree algorithm

```
procedure kruskal (G, w)
Input:  A connected undirected graph G = (V, E) with edge weights w_e
output: A minimum spanning three defined by the edges X

for all u ∈ V:
   makeset (u)

X = {}
sort the edges E by weight
for all edges {u, v} ∈ E, in increasing order of weight:
   if find(u) ≠ find(v):
      add edge {u, v} to X
      union(u, v)
```

A data structure for disjoint sets

Union by rank:

One way to store a set is as a directed tree.

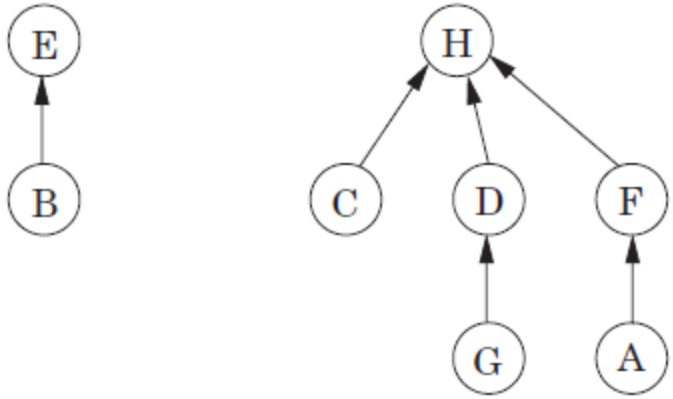
Nodes of the tree are elements of the set, arranged in no particular order, and each has parent pointers that eventually lead up to the root of the tree.

This root element is a convenient representative, or name, for the set.

It is distinguished from the other elements by the fact that its parent pointer is a self-loop.

A data structure for disjoint sets

A directed-tree representation of two sets $\{B, E\}$ *and* $\{A, C, D, F, G, H\}$:



Union by rank

In addition to a parent pointer π, each node also has a rank that, for the time being, should be interpreted as the height of the subtree hanging from that node.

```
procedure makeset(x)
π(x) = x
rank(x) = 0

function find(x)
while x ≠ π(x): x = π(x)
return x
```

Makeset

makeset is a constant-time operation.

find follows parent pointers to the root of the tree and therefore takes time proportional to the height of the tree.

The tree actually gets built via the third operation, *union*, and so this procedure should keep trees shallow.

Merging two sets

Make the root of one point to the root of the other.

If the representatives (roots) of the sets are r_x and r_y, should we make

r_x point to r_y or the other way around?

A good strategy is to make the root of the shorter tree point to the root of the taller tree.

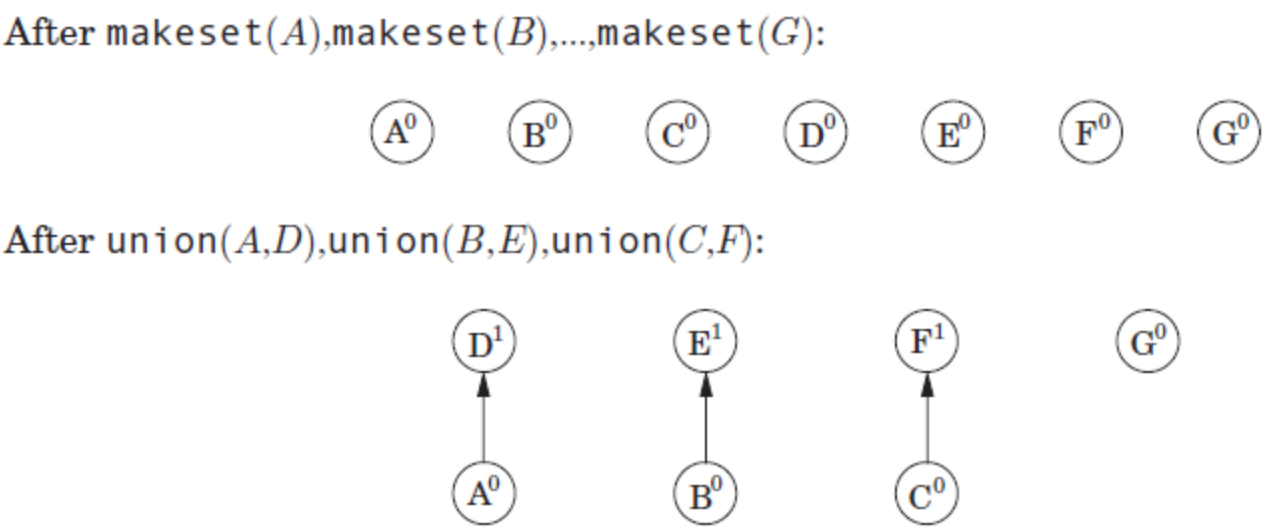
This way, the overall height increases only if the two trees being merged are equally tall. Instead of explicitly computing heights of trees, we will use the rank numbers of their root nodes—which is why this scheme is called *union by rank*.

Union by rank

```
procedure union(x, y)
r_x = find(x)
r_y = find(y)
if r_x = r_y: return
if rank(r_x) > rank(r_y):
    π(r_y) = r_x
else:
    π(r_x) = r_y
    if rank(r_x) = rank(r_y): rank(r_y) = rank(r_y) + 1
```

Example

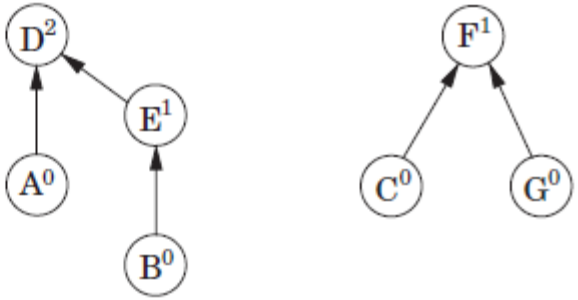
A sequence of disjoint-set operations. Superscripts denote rank.



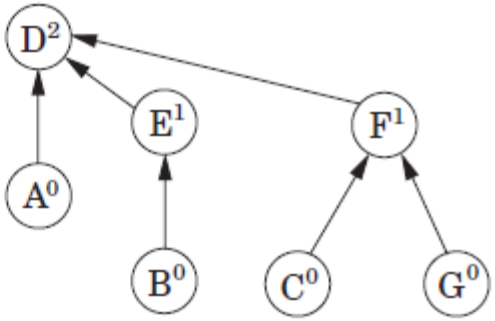
Example

A sequence of disjoint-set operations. Superscripts denote rank.

After union(C,G),union(E,A):



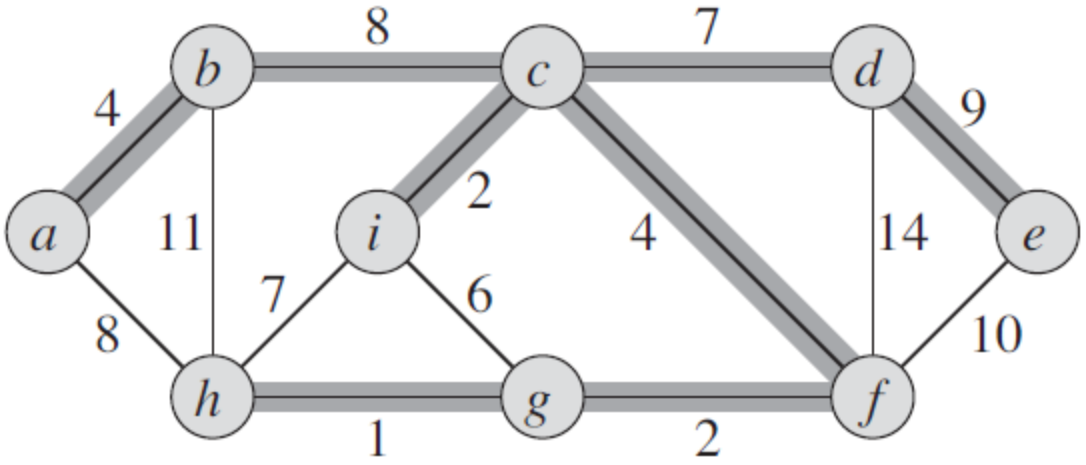
After union(B,G):



How the Kruskal works

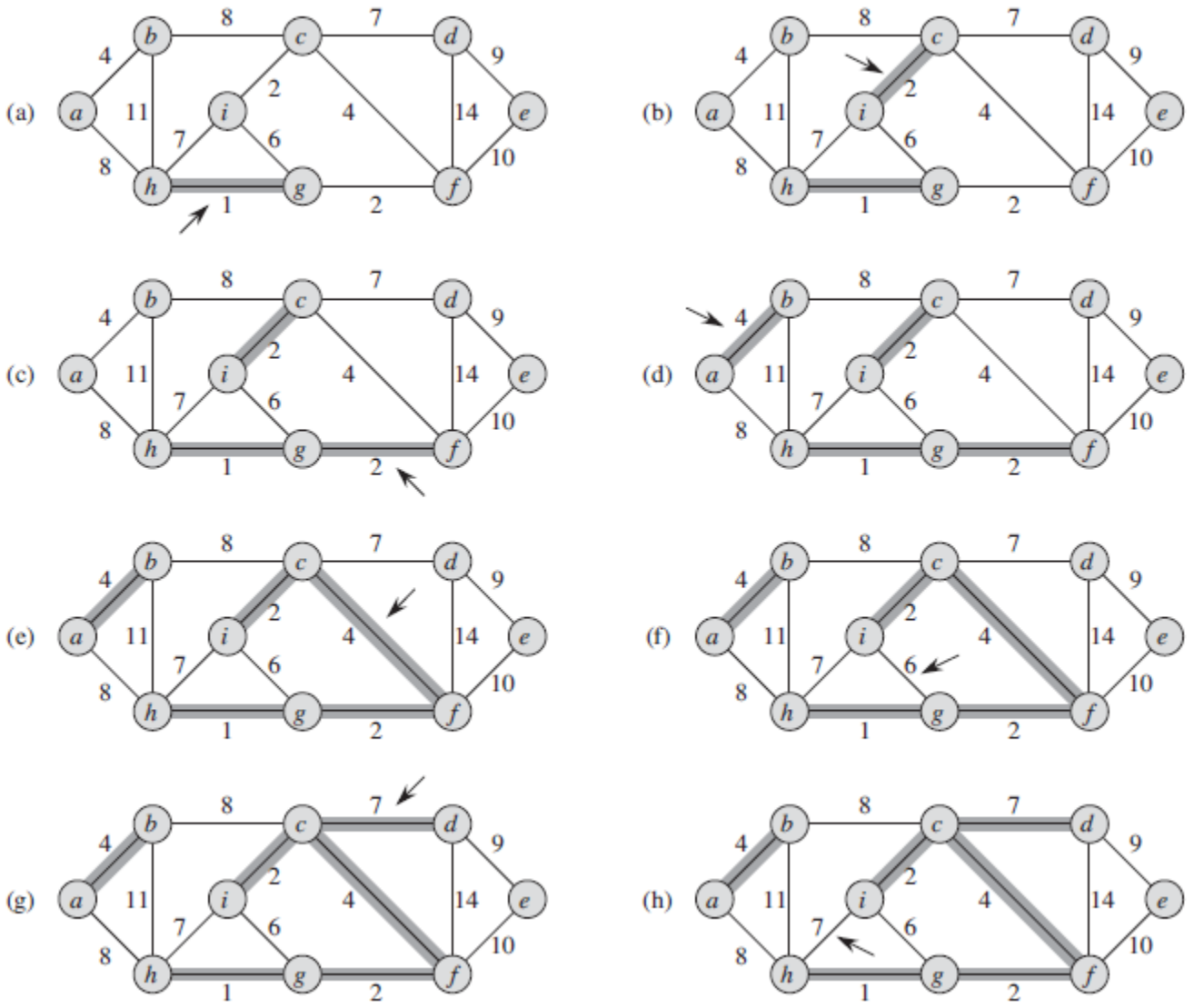
Let's consider minimum spanning tree for a connected graph.

The weights on edges are shown, and the edges in a minimum spanning tree are shaded. The total weight of the tree is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another spanning tree with weight 37.

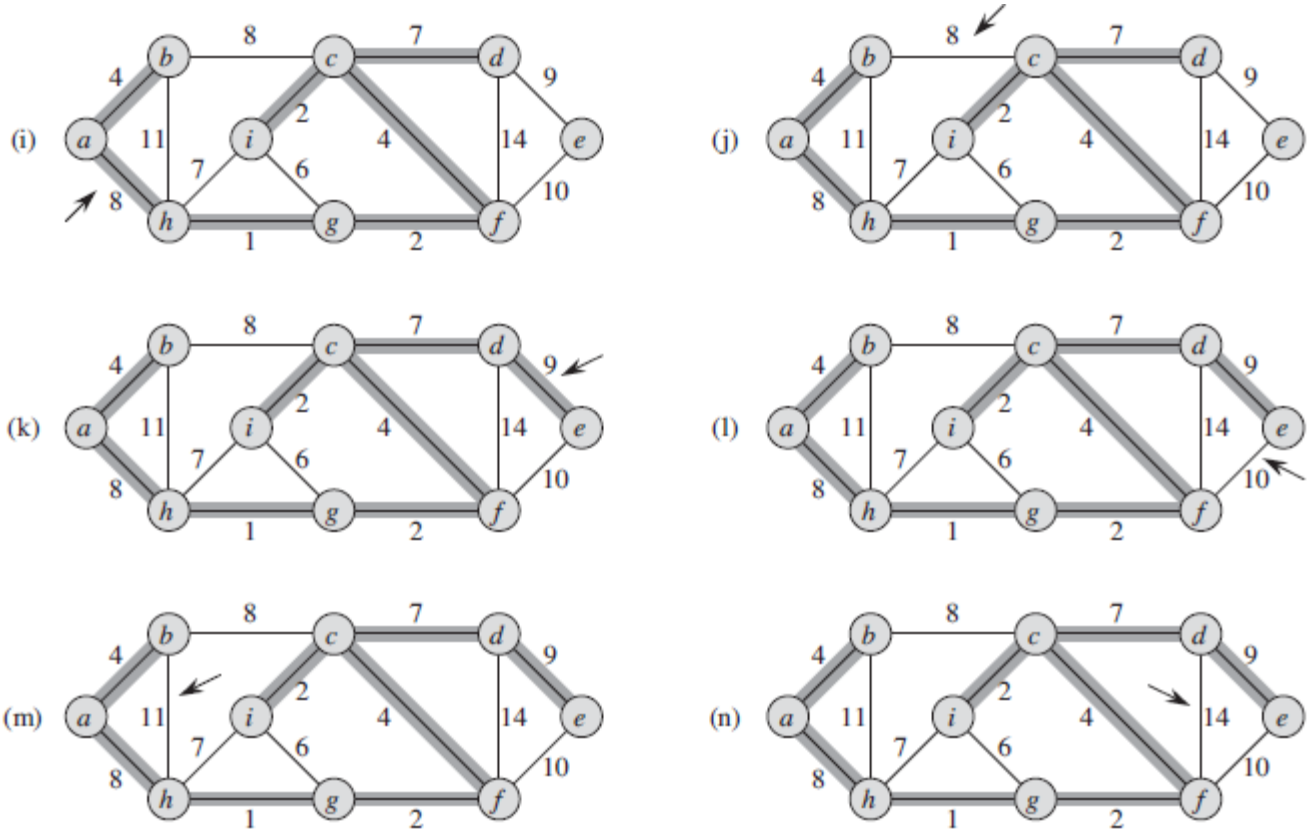


How the Kruskal works

- The algorithm considers each edge in sorted order by weight.
- An arrow points to the edge under consideration at each step of the algorithm.
- If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
7
6
9
14
10
1
2

Further steps



Property

By design, the rank of a node is exactly the height of the subtree rooted at that node.

This means, for instance, that as moving up a path toward a root node, the *rank* values along the way are strictly increasing.

Property 1: ***For any x, rank*(*x*) < *rank*(π(*x*)).**

A root node with rank k is created by the merger of two trees with roots of rank $k-1$.

Property 2

Any root node of rank k has at least 2^k nodes in its tree.

This extends to internal (nonroot) nodes as well: a node of rank k has at least 2^k descendants.

After all, any internal node was once a root, and neither its rank nor its set of descendants has changed since then.

Moreover, different rank-k nodes cannot have common descendants, since by Property 1 any element has at most one ancestor of rank k.

Property 3

If there are n elements overall, there can be at most $n/2^k$ nodes of rank k.

This last observation implies that the maximum rank is $\log n$. Therefore, all the trees have height $\leq \log n$, and this is an upper bound on the running time of **find** and **union**.

Path compression

The total time for Kruskal's algorithm becomes $O(|E| \log |V|)$ for sorting the edges ($\log |E| \approx \log |V|$) $O(|E| \log |V|)$ for the **union** and **find** operations that dominate the rest of the algorithm.

So there seems to be little incentive to make our data structure any more efficient.

But what if the edges are given as sorted?

Or if the weights are small (say, $O(|E|)$) so that sorting can be done in linear time?

Then the data structure part becomes the bottleneck, and it is useful to think about improving its performance beyond log n per operation.

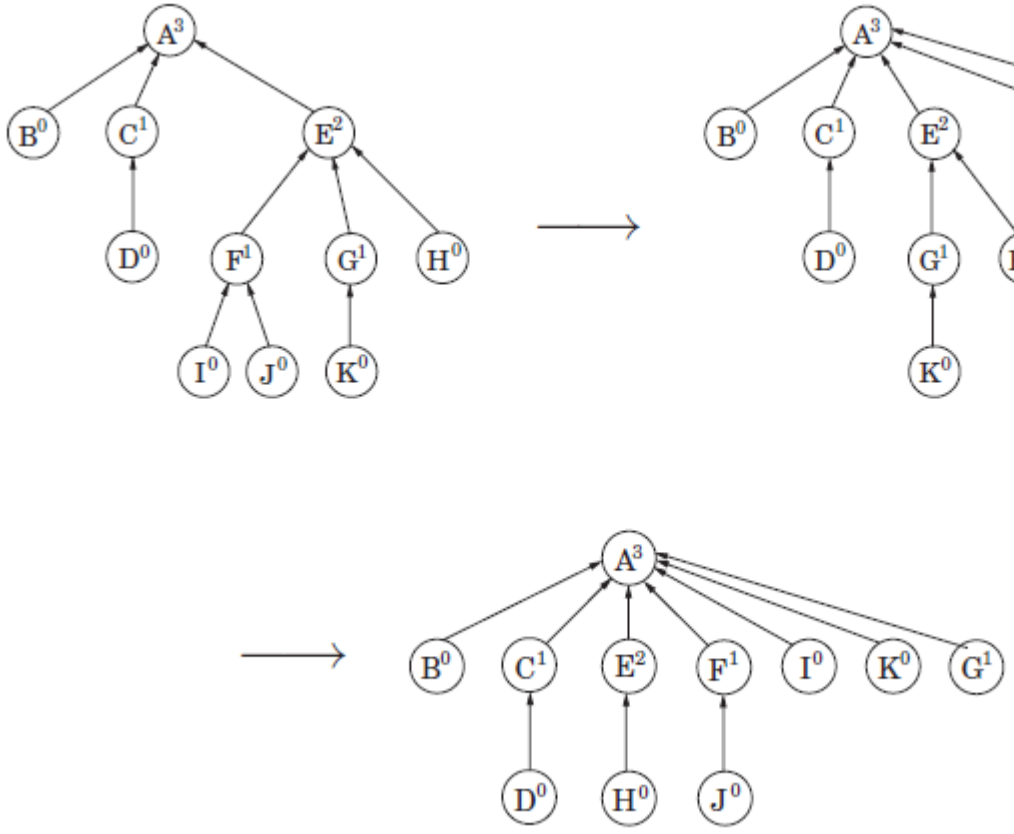
Path compression

- During each find, when a series of parent pointers is followed up to the root of a tree, change all these pointers so that they point directly to the root.
- This *path compression heuristic only slightly increases the* time needed for a find and is easy to code.

```
function find(x)
if x ≠ π(x): π(x) = find(π(x))
return π(x)
```

Path compression

The effect of path compression: **find**(*I*) followed by **find**(*K*):



Path compression

- Look at sequences of **find** and **union** operations, starting from an empty data structure, and determine the average time per operation.
- This *amortized cost* turns out to be just barely more than $O(1)$, down from the earlier $O(\log n)$.
- The data structure are having a "top level" consisting of the root nodes, and below it, the insides of the trees.
- There is a division of labor: **find** operations (with or without path compression) only touch the insides of trees, whereas **union**'s only look at the top level.
- Thus path compression has no effect on union operations and leaves the top level unchanged.

Path compression

If there are n elements, their rank values can range from 0 to $\log n$ by Property 3.

Let's divide the nonzero part of this range into certain carefully chosen intervals:

$\{1\}, \{2\}, \{3, 4\}, \{5, 6, \ldots, 16\}, \{17, 18, \ldots, 2^{16} = 65536\}, \{65537, 65538, \ldots, 2^{65536}\}, \ldots$

Each group is of the form $\{k+1, k+2, \ldots, 2^k\}$, where k is a power of 2.

The number of groups is $\log^* n$, that need to be applied to n to bring it down to 1 (or below 1). For instance, $\log^* 1000 = 4$ since $\log \log \log \log 1000 \leq 1$.

In practice there will just be the first five of the intervals shown; more are needed only if $n \geq 265536$, in other words never.

Details

- In a sequence of find operations, some may take longer than others. Let's bound the overall running time using some creative accounting. Specifically, let's give each node a certain amount of pocket money, such that the total money doled out is at most $n\log^* n$ dollars.
- It can be showed that each find takes $O(\log^* n)$ steps, plus some additional amount of time that can be "paid for" using the pocket money of the nodes involved—one dollar per unit of time. Thus the overall time for m **find**'s is $O(m\log^* n)$ plus at most $O(n\log^* n)$.
- A node receives its allowance as soon as it ceases to be a root, at which point its rank is fixed. If this rank lies in the interval $\{k+1, \ldots, 2k\}$, the node receives $2k$ dollars. By Property 3, the number of nodes with $rank > k$ is bounded by

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leq \frac{n}{2^k}.$$

Details

- The total money given to nodes in this particular interval is at most n dollars, and since there are $\log^* n$ intervals, the total money disbursed to all nodes is $\leq n \log^* n$.
- The time taken by a specific **find** is simply the number of pointers followed. Consider the ascending rank values along this chain of nodes up to the root. Nodes x *on the chain fall into two categories: either the rank of* $\pi(x)$ *is in a higher interval* than the rank of x, *or else it lies in the same interval. There are at most* $\log^* n$ *nodes* of the first type, so the work done on them takes $O(\log^* n)$ time.
- The remaining nodes—whose parents' ranks are in the same interval as theirs—have to pay a dollar out of their pocket money for their processing time.

Prim's algorithm

For minimum spanning tree:

Any algorithm conforming to the following greedy schema is

```
X = { } (edges picked so far)
repeat until |X| = |V| − 1:
  pick a set S ⊂ V for which X has no edges between S and V − S
  let e ∈ E be the minimum-weight edge between S and V − S
  X = X ∪ {e}
```

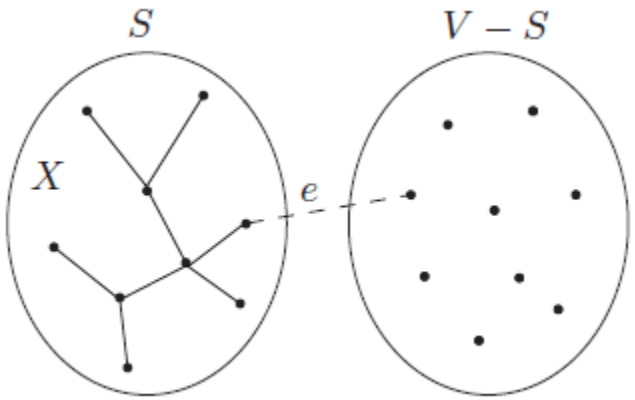
A popular alternative to Kruskal's algorithm is Prim's, in which the intermediate set of edges X always forms a subtree, and S is chosen to be the set of this tree's vertices.

Prim's algorithm

On each iteration, the subtree defined by X grows by one edge, namely, the lightest edge between a vertex in S and a vertex outside S.

Equivalently, think of S as growing to include the vertex $v \in S$ of smallest cost: $\texttt{cost}(v) = \min_{u \in S} w(u, v)$.

Prim's algorithm: the edges X form a tree, and S consists of its vertices

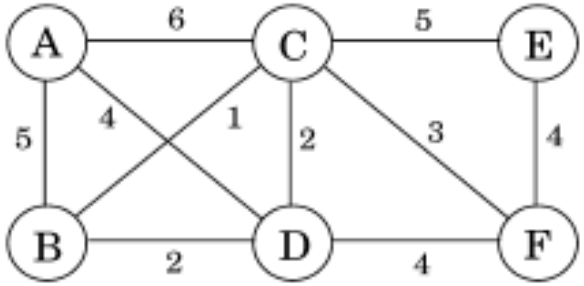


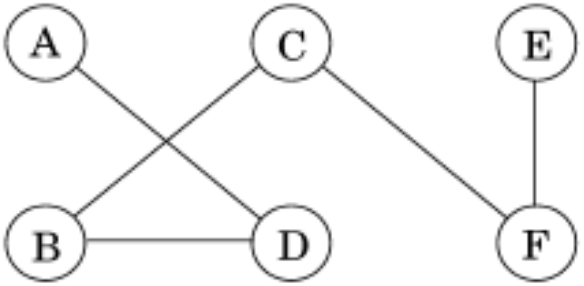
Prim's algorithm

```
procedure prim (G, w)
Input:  A connected undirected graph G = (V, E) with edge weights w_e
output: A minimum spanning tree defined by the array prev

for all u ∈ V:
    cost(u) = ∞
    prev(u) = nil
pick any initial node u_0
cost(u_0) = 0

H = makequeue (V)  (priority queue, using cost-values as keys)
while H is not empty:
    v = deletemin (H)
    for each {v, z} ∈ E:
        if cost(z) > w(v, z):
            cost(z) > w(v, z)
            prev(z) = v
            decreasekey(H, z)
```

Prim's algorithm

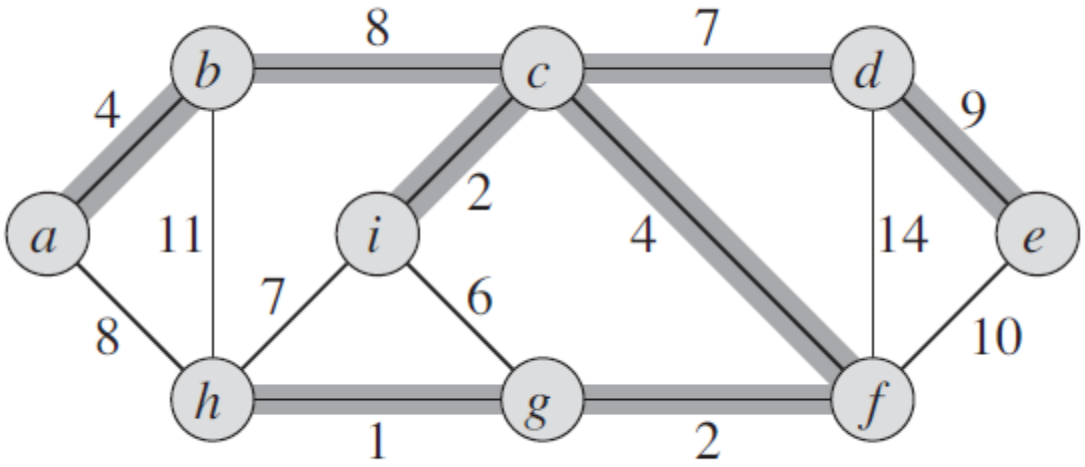
Starting at node A.

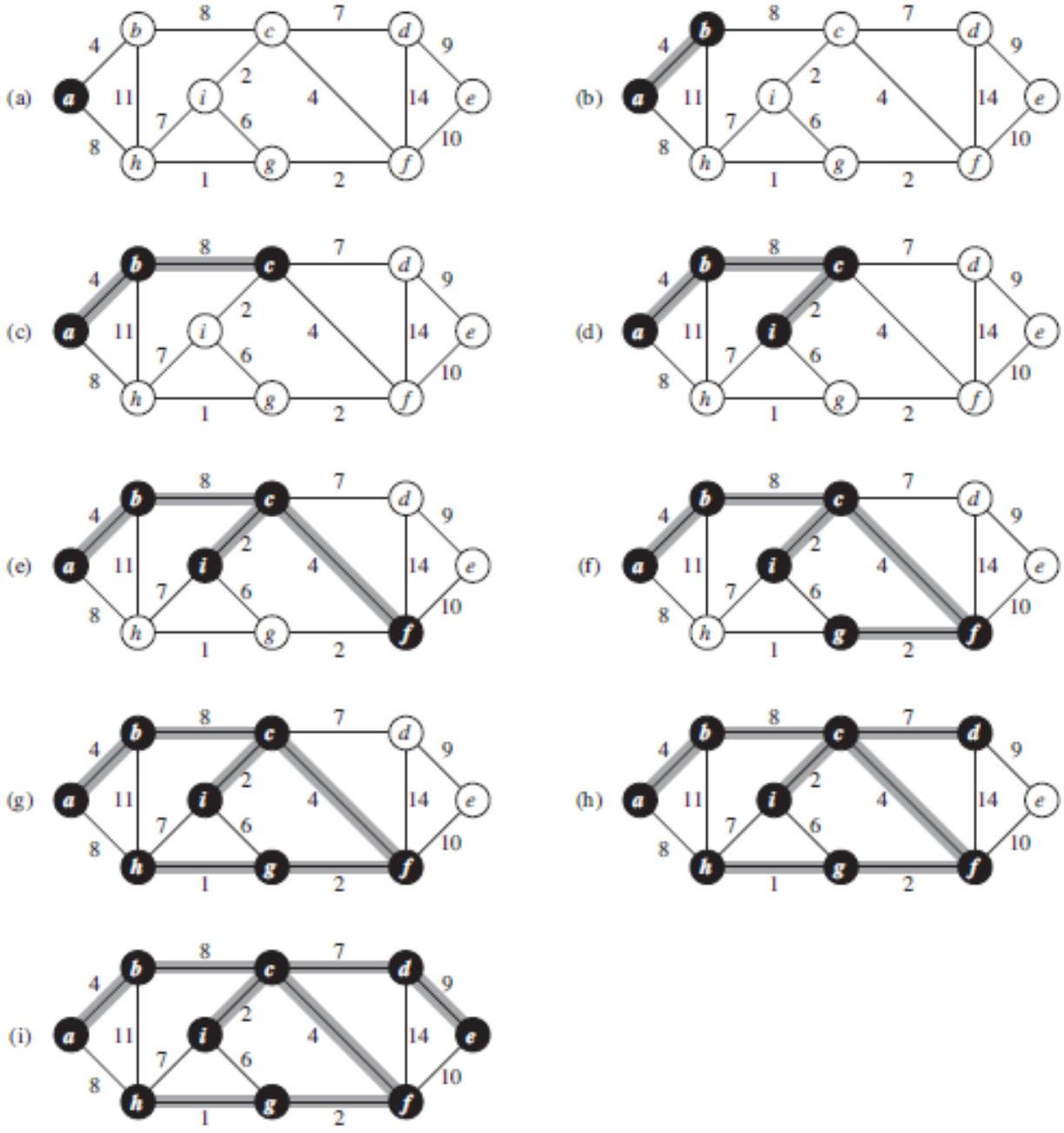




Set S	A	B	C	D	E	F
$\{\}$	0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
A		$5/A$	$6/A$	$4/A$	∞/nil	∞/nil
A, D		$2/D$	$2/D$		∞/nil	$4/D$
A, D, B			$1/B$		∞/nil	$4/D$
A, D, B, C					$5/C$	$3/C$
A, D, B, C, F					$4/F$	

Prim's algorithm: example




(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
a
b
c
d
e
f
g
h
i
4
8
8
7
9
11
2
4
14
7
6
10
8
1
2