Greedy Algorithms (Ch 4)

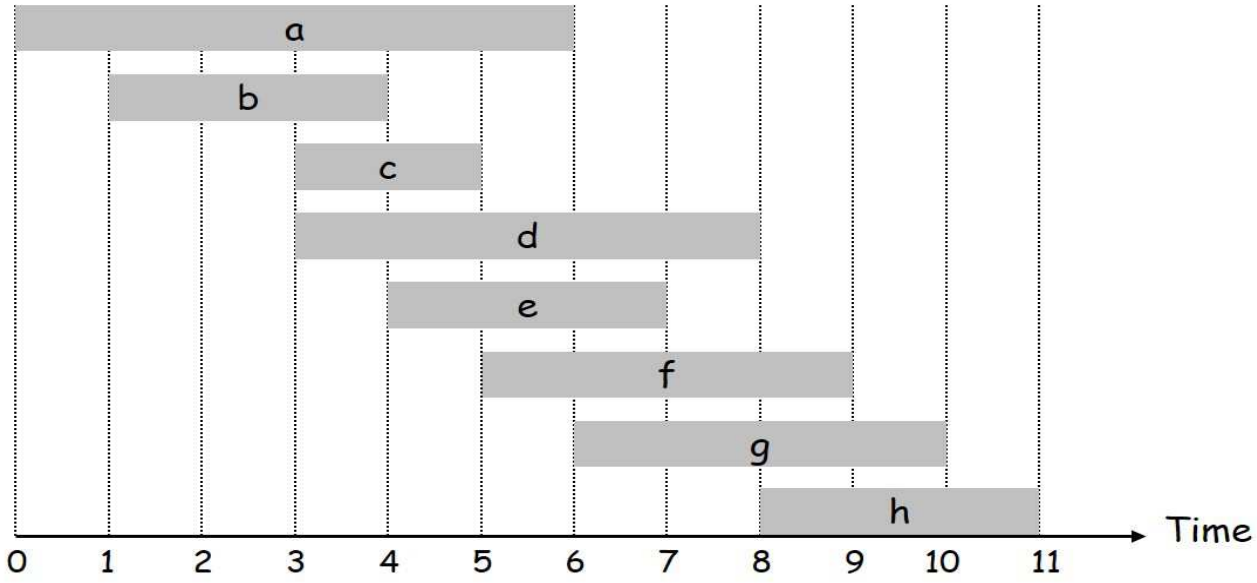
- **Greedy Algorithm for Interval Scheduling Problem**
- **Elements of Greedy Algorithms**
- **Greedy Algorithm for Minimizing the Maximum Lateness Problem**
- **Greedy Algorithm for Shortest Path Problem**
- **Greedy Algorithm for Clustering Problem**

The lecture notes/slides are adapted from those associated with the text book by J. Kleinberg and E. Tardos.

Greedy Algorithm for Interval Scheduling Problem

Interval scheduling problem

- **Given a set $S = \{a_1, .., a_n\}$ of proposed jobs that wish to use a resource which can serve one job at a time. Each a_i has a start time s_i and a finish time f_i with $0 \leq s_i < f_i < \infty$. If selected, a_i takes place in time interval $[s_i, f_i)$.**
- **Jobs a_i and a_j are compatible if $[s_i, f_i) \cap [s_j, f_j) = \emptyset$.**
- **Goal, Find a maximum subset of mutually compatible jobs from S.**



Greedy algorithm for interval scheduling problem

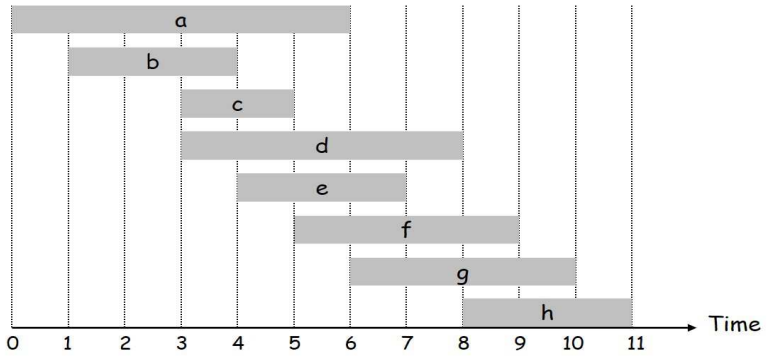
- **Find an optimal solution in steps, each step optimizes a partial solution locally.**
- **Consider jobs in some order, select each job compatible with the selected ones.**
- **Possible selections:**

  Earliest-start-time-first, consider jobs in ascending order of s_i.

  Earliest-finish-time-first, consider jobs in ascending order of f_i.

  Shortest-interval-first, consider jobs in ascending order of $f_i - s_i$.

  Fewest-incompatible-jobs-first, consider jobs in ascending order of the number of incompatible jobs.



Earliest-finish-time-first algorithm

- **In each step, select a job a_k with the earliest finish time.**
- **Intuition, select a_k which leaves resource available for as many other jobs as possible.**
- **Algorithm**

  Assume $f_1 \leq .. \leq f_n$. Let $S_k = \{a_i \in S | s_i \geq f_k\}$.

  Choose a_1; then choose an a_i with the smallest f_i from S_1;
  repeat this selection, assume a_k was chosen in the previous round of selection, choose an a_i from S_k, until no activity can be selected.
- **Sort jobs in ascending order of finish time takes $O(n \log n)$ time, the rest steps take $O(n)$ time, total running time $O(n \log n)$.**

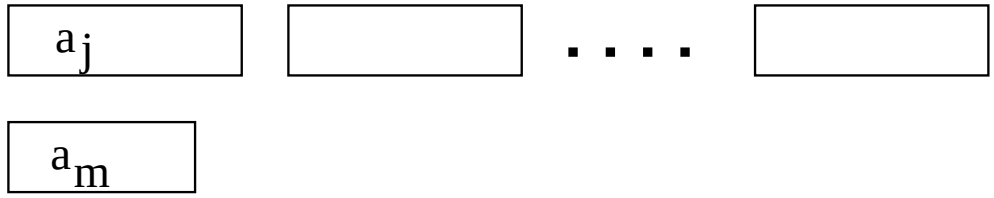
Example, set S of the following jobs:

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

- **Initially, partial solution $S^* = \{a_1\}$, then a job from S_1 is selected.**
- **Since $S_1 = \{a_4, a_6, a_7, a_8, a_9, a_{11}\}$ and a_4 has the smallest f_i in S_1,** $S^* = \{a_1\} \cup \{a_4\}$ **and then select a job from S_4.**
- **Since $S_4 = \{a_8, a_9, a_{11}\}$ and a_8 has the smallest f_i in S_4,** $S^* = \{a_1, a_4\} \cup \{a_8\}$ **and then select a job from S_8.**
- **Since $S_8 = \{a_{11}\}$, $S^* = \{a_1, a_4, a_8\} \cup \{a_{11}\}$ and then select a job from S_{11}.**
- **There is no job in S_{11} and the algorithm terminates.**

Theorem. For any nonempty S_k and $a_m \in S_k$ with the smallest f_m, a_m is included in a maximum-size subset of mutually compatible jobs of S_k.

Proof. Let A_k be a maximum-size subset of mutually compatible jobs in S_k and a_j be the job in A_k with the smallest f_j. If $a_j = a_m$ then the theorem is proved. Assume $a_j \neq a_m$. Let $A'_k = (A_k \setminus \{a_j\}) \cup \{a_m\}$. Since $a_j \in A_k$ and for any $a_i \in A_k$ with $a_i \neq a_j$, $f_j \leq f_i$ and a_j is compatible with a_i, $f_j \leq s_i$. Therefore and from $f_m \leq f_j$, $f_m \leq s_i$ for every $a_i \in A_k$. Thus, jobs in A'_k are mutually compatible. □



- **Earliest-start-time-first selection may not give an optimal solution.**
- **Shortest-interval-first selection may not give an optimal solution.**
- **Fewest-incompatible-jobs-first selection may not give an optimal solution.**

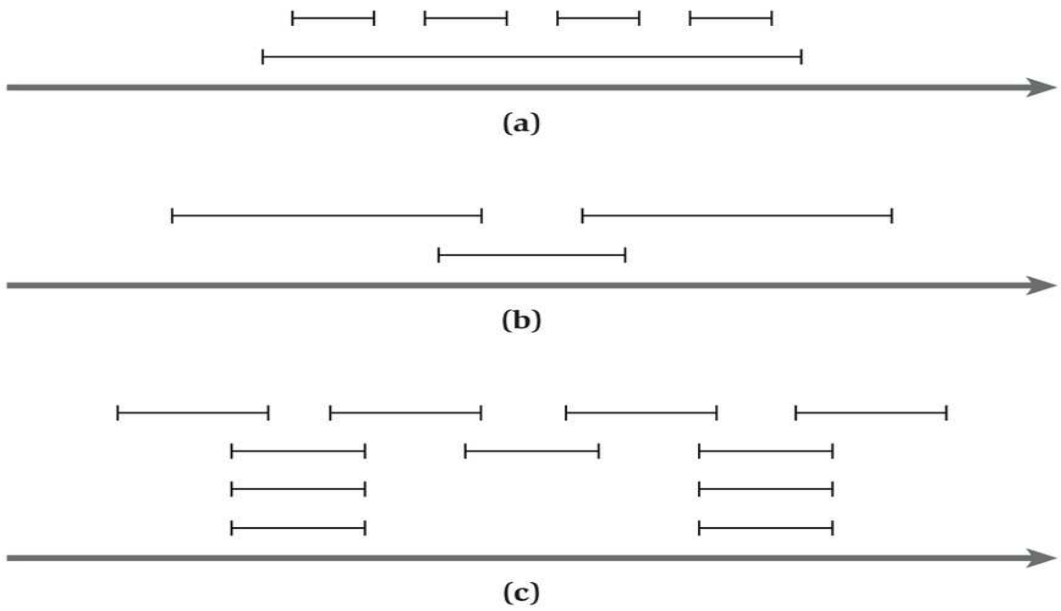


Figure 4.1 Some instances of the Interval Scheduling Problem on which natural greedy algorithms fail to find the optimal solution. In (a), it does not work to select the interval that starts earliest; in (b), it does not work to select the shortest interval; and in (c), it does not work to select the interval with the fewest conflicts.

Elements of Greedy Algorithms

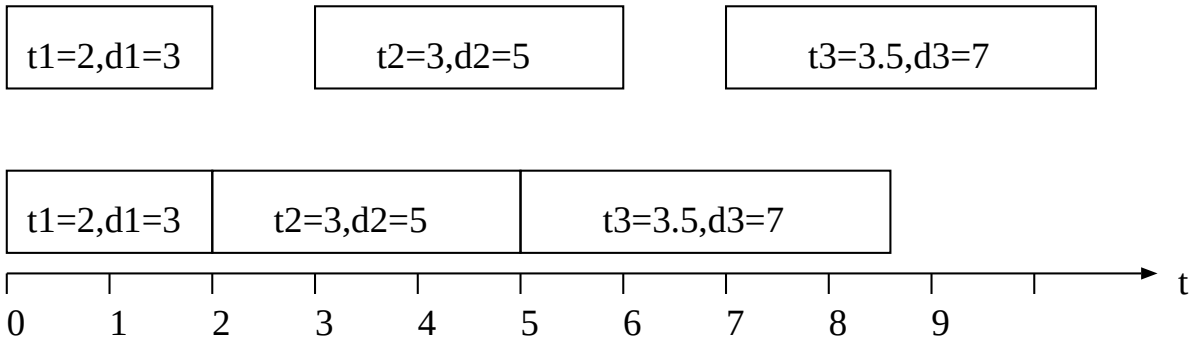
- **Major steps in designing a greedy algorithm**
  - **Formulate the optimization problem as a sequence of subproblems in which a local optimal (greedy) choice solves each subproblem.**
  - **Prove that a greedy choice will give an optimal solution to each subproblem.**
  - **Show optimal substructure: for each subproblem, a greedy optimal solution to the subproblem combined with a greedy choice can give an optimal solution to a remaining subproblem.**
- **Greedy-choice property: a global optimal solution can be assembled by making locally optimal greedy choices.**
- **Optimal substructure: Optimal solution contains optimal solutions to subproblems.**

Scheduling to minimize maximum lateness

- **Let $S = \{a_1, .., a_n\}$ be a set of proposed jobs that wish to use a resource which can serve one job at a time. Each a_i has deadline d_i and wishes a contiguous time interval t_i, before d_i to use the resource. For each a_i, a schedule S decides the time s_i at which a_i starts to use the resource. The finish time of a_i is $f_i = s_i + t_i$.**

  The lateness of a_i is defined as $l_i = f_i - d_i$ if $f_i > d_i$ otherwise 0.

  Optimization goal: find a schedule S s.t. the maximum lateness of S $l(S) = \max_{1 \leq i \leq n} l_i$ is minimized.



Maximum lateness=max{l1=0,l2=0,l3=1.5}=1.5

A greedy algorithm

- **Order the jobs in the order $a_1, .., a_n$ s.t. $d_1 \leq d_2 \leq ... \leq d_n$. Let t be the earliest available time of the resource (e.g., $t = 0$).**

  For a_1, let $s_1 = t$ and $f_1 = s_1 + t_1$;
  For a_i with $2 \leq i \leq n$, $s_i = f_{i-1}$ and $f_i = s_i + t_i$;
  Assign each a_i to resource at time interval $[s_i, f_i)$;

- **No idle time property: The resource serves each job at any time point from the start of the 1st served job until the finish time of the last served one.**

- **No inversion property: A schedule has an inversion if there are two jobs a_i and a_j s.t. $d_j < d_i$ and $s_j > s_i$. A schedule has no inversion if $s_i < s_j$ for $d_i < d_j$.**

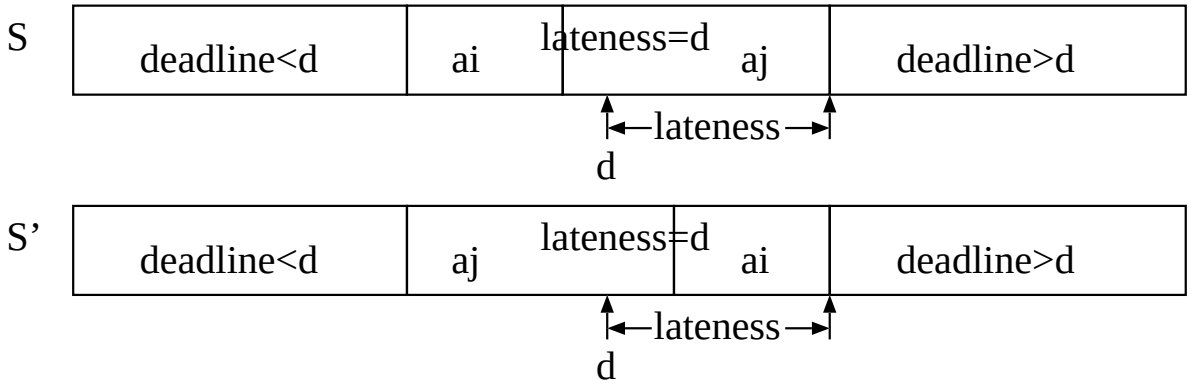
- **The algorithm above has no idle time and no inversion.**

- **Claim 1: There is an optimal schedule with no idle time.**

  For an optimal schedule S, assume that the resource is idle for a time interval $t = [s, f)$ before some jobs are served. Then for every job a_i, if a_i is served before time s then let $s'_i = s_i$, otherwise ($f \leq s_i$), let $s'_i = s_i - (f - s)$. Then we get a schedule S' with $l(S') \leq l(S)$. Repeat this process, we get an optimal schedule with no idle time property.

- **Claim 2: All schedules with no idle time and no inversions have the same maximum lateness.**

  Let S and S' be schedules with no idle time and no inversions. Then two jobs a_i and a_j are in different served orders in S and S' only if $d_i = d_j = d$. All jobs with deadline d are served consecutively in S and S' (after jobs with deadline$< d$ and before jobs with deadline$> d$. The maximum lateness of all jobs with deadline d is independent of the orders they served.



- **Claim 3: There is an optimal schedule that has no idle time and no inversions.**

  Let $S : a_{j_1}, .., a_{j_n}$ be an optimal schedule with no idle time (by Claim 1). If S has no inversion then the claim holds. Assume S has an inversion. Then there is pair a_{j_i} and $a_{j_{i+1}}$ s.t. $d_{j_i} > d_{j_{i+1}}$. Let S' be the schedule obtained by exchanging the ordr of a_{j_i} and $a_{j_{i+1}}$. Then S' has one less inversion than S. We prove $l(S') \leq l(S)$.

  Let l_{j_r} be the lateness and f_{j_r} be the finish time of a_{j_r} in S. Let l'_{j_r} be the lateness and f'_{j_r} be the finish time of a_{j_r} in S'. Then

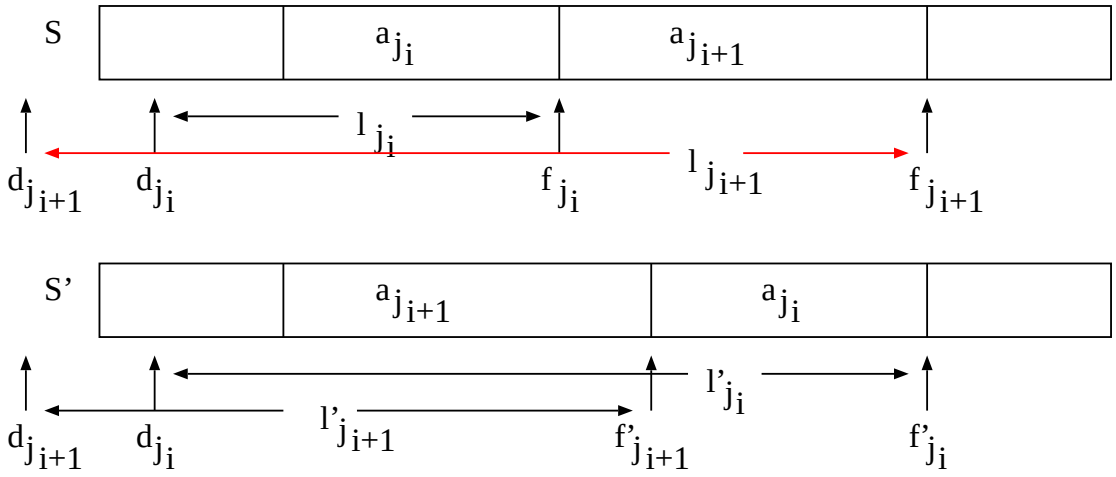
  $l_{j_r} = l'_{j_r}$ **for every** $r \neq i, i+1$,

  $l'_{j_{i+1}} = f'_{j_{i+1}} - d_{j_{i+1}} < f_{j_{i+1}} - d_{j_{i+1}} = l_{j_{i+1}}$ **and**

  $l'_{j_i} = f'_{j_i} - d_{j_i} = f_{j_{i+1}} - d_{j_i} < f_{j_{i+1}} - d_{j_{i+1}} = l_{j_{i+1}}$.

  We get $\max\{l'_{j_i}, l'_{j_{i+1}}\} \leq \max\{l_{j_i}, l_{j_{i+1}}\}$. **Thus,** $l(S') \leq l(S)$.

  Repeat the exchange process above, we get the claim.



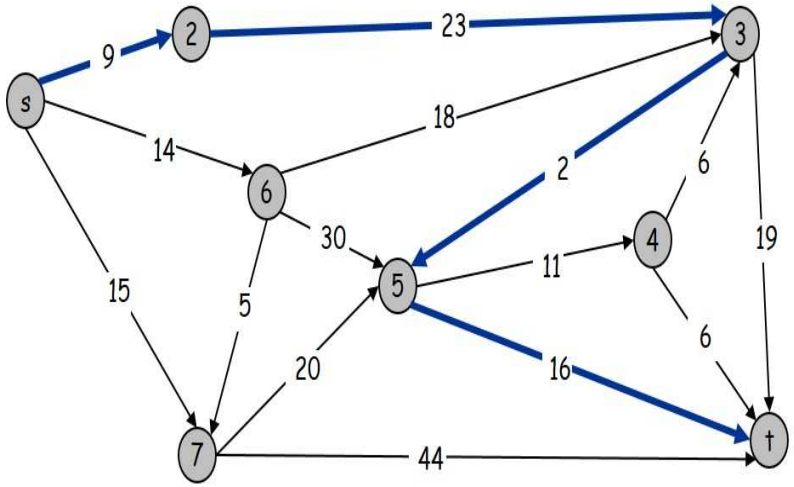
- **The greedy algorithm finds an optimal schedule in $t(n) + O(n)$ time, $t(n)$ is the time to sort n numbers.**

  Proof. By Claim 3, there is an optimal schedule with no idle time and no inversion. By Claim 2, all schedules with no idle time and on inversion have the same maximum lateness. Hence, a schedule with no idle time and no inversion is optimal. The algorithm finds a schedule with no idle time and no inversion, an optimal one.

  It takes $t(n)$ time to sort the jobs in ascending order of their deadlines and $O(n)$ time compute s_i and f_i. The running time of the algorithm is $t(n) + O(n)$. □

Greedy Algorithm for Shortest Path Problem

- $G(V, E)$, **weighted digraph with each arc** e **assigned a real value length** $d(e)$ **(or cost or weight or ..).**
- **For a path** P **consists of arcs** $e_1, e_2, .., e_k$**, the length of** P **is** $d(P) = \sum_{1 \leq i \leq k} d(e_i)$**.**
- **For nodes** s, t **in** G**, the shortest path from** s **to** t **is a path from** s **to** t **with the minimum length. The length of the shortest path from** s **to** t **is called the distance** $d(s, t)$ **from** s **to** t**.**



Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 50

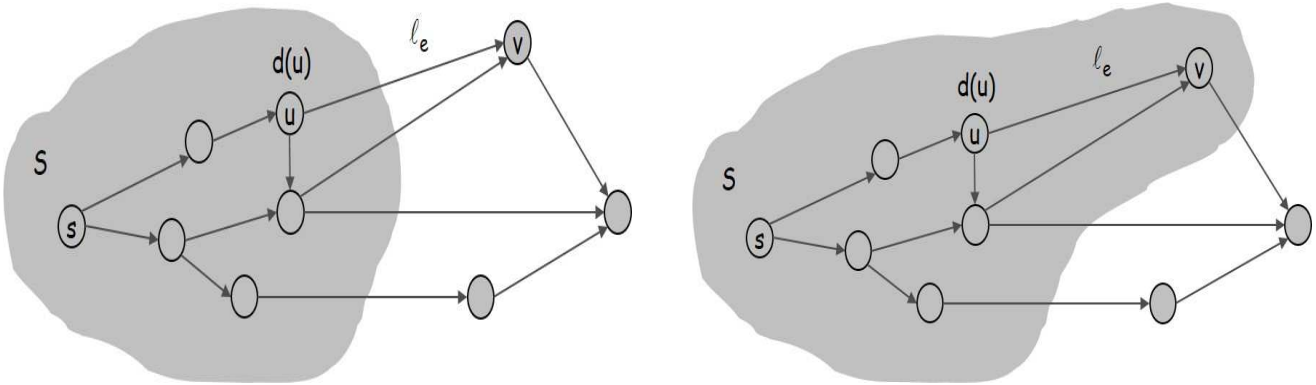
- **Single source shortest paths (SSSP) problem, find the shortest paths (distances) from a source node s to all other nodes.**
- **All pairs shortest paths (APSP) problem, for every pair of nodes u, v, find the shortest path (distance) from u to v.**

Dijkstra's algorithm for single source shortest paths problem

- **Greedy approach: Let s be a source node, maintain a set S of nodes, for each $u \in S$, $d(s,u) =$length of a shortest path $s \rightsquigarrow u$ has been found.**
  - **Initialize $S = \{s\}$, $d(s,s) = 0$;**
  - **In each step, choose a node $v \notin S$ which minimizes $\tilde{d}(s,v) = \min_{e=(u,v):u \in S} d(s,u) + d(e)$; $S = S \cup \{v\}$; $d(s,v) = \tilde{d}(s,v)$;**
  - **Repeat above until $S = V(G)$.**

S: solution to a subproblem (shortest distances to a subset of nodes found).

$\tilde{d}(s,v)$: (local) shortest distance of $s \rightsquigarrow v$ paths passing through nodes of S only.



Dijkstra Algorithm [Dijkstra 1956] for Shortest Distance

Input: Weighted digraph G with non-negative edge length and source s.

Output: Distance $d(s,u)$ to every node u.

Initially, $S := \{s\}; d(s,s) = 0;$

for every node $u \neq s$,

if $e = (s,u) \in E$ **then** $\tilde{d}(s,u) = d(e)$ **else** $\tilde{d}(s,u) = \infty;$

while $S \neq V$ **do**

$v = \arg\min_{v \in V \setminus S}\{\tilde{d}(s,v)\};$

$S := S \cup \{v\}; d(s,v) = \tilde{d}(s,v);$

for every edge $e = (v,v'), \tilde{d}(s,v') = \min\{\tilde{d}(s,v'), d(s,v) + d(e)\};$

end while

Dijkstra Algorithm [Dijkstra 1956] for Shortest Distance and Path

Input: Weighted digraph G with non-negative edge length and source s.

Output: Distance $d(s,u)$ to every node u and predecessor $P(u)$.

Initially, $S := \{s\}$; $d(s,s) = 0$; $P[s] = s$;

for every node $u \neq s$,

$P[u] = null$;

if $e = (s,u) \in E$ **then** $\tilde{d}(s,u) = d(e)$ **else** $\tilde{d}(s,u) = \infty$;

while $S \neq V$ **do**

$v = \arg\min_{v \in V \setminus S}\{\tilde{d}(s,v)\}$; let $e = (u,v)$ be the edge s.t. $\tilde{d}(s,v) = d(s,u) + d(e)$;

$S := S \cup \{v\}$; $d(s,v) = \tilde{d}(s,v)$; $P[v] := u$;

for every edge $e = (v,v')$, $\tilde{d}(s,v') = \min\{\tilde{d}(s,v'), d(s,v) + d(e)\}$;

end while

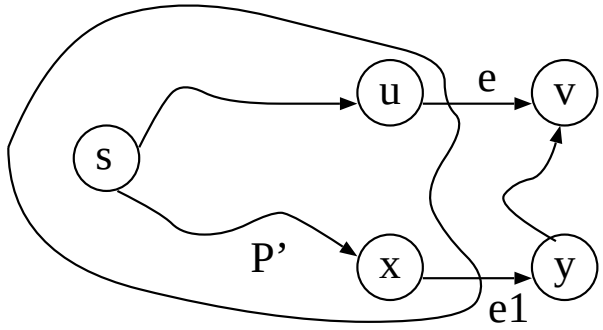
Theorem. [Dijkstra 1956] Dijkstra's Algorithm finds the distance $d(s,u)$ and the shortest path from s to every node in G.

Proof. Let P_u be the path $s, .., P[P[u]], P[u], u$. We prove the loop invariant: For each node $u \in S$, $d(s,u)$ =length of a shortest path $s \rightsquigarrow u$ and P_u is a shortest path $s \rightsquigarrow u$. For $|S| = 1$, $S = \{s\}$, $d(s,s) = 0$ and $P_s = s$, the invariant is true. Assume the invariant holds for $|S| \geq 1$.

Let v be the next node added to S and $e = (u,v)$ be the edge s.t. $\tilde{d}(s,v) = d(s,u) + d(e)$. Let P be any other path $s \rightsquigarrow v$, $e_1 = (x,y)$ be the 1st edge in P with $x \in S$ and $y \notin S$, and P' be the subpath $s \rightsquigarrow x$ of P. Then

$$d(P) \geq d(P') + d(e_1) \geq d(s,x) + d(e_1) \geq \tilde{d}(s,y) \geq \tilde{d}(s,v).$$

Thus, $d(s,v)$ =length of a shortest path $s \rightsquigarrow v$ and P_v is a shortest path $s \rightsquigarrow v$. □



Running time of Dijkstra's Algorithm,

- **For $S = \{s\}$, $O(n)$ time for initialization.**

  $n - 1$ iterations of the while loop.

  Each time a node v is added to S, $O(\text{outdeg}(v))$ updates $\tilde{d}(s, v') = \min\{\tilde{d}(s, v'), d(s, v) + d(v, v')\}$, total $O(m)$ updates.

- **Straightforward implementation**

  $O(n)$ time to find v with $\tilde{d}(s, v) = \min_{e=(u,v),u\in S}\{d(s, u) + d(e)\}$ minimal, total $O(n^2)$ time.

- **Improvements**

  Keep $\tilde{d}(s, v) = \min_{e=(u,v),u\in S}\{d(s, u) + d(e)\}$ in a priority queue.

  By a binary heap, $O(1)$ time to find v and $O(\log n)$ time to adjust the heap after an update $\tilde{d}(s, v') = \min\{\tilde{d}(s, v'), d(s, v) + d(v, v')\}$, total $O(m \log n)$ time.

  By a Fibonacci heap, $O(m + n \log n)$ time.

  Most recent result, $O(m(\log n)^{2/3})$ time (STOC2025)

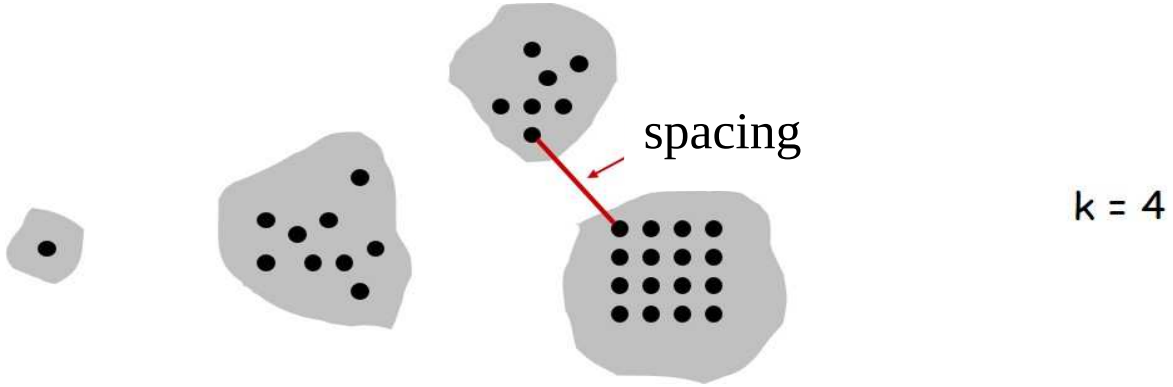
Clustering

- **Clustering instance: a set V of n points $p_1, .., p_n$ and a distance function $d(p_i, p_j)$ for $p_i, p_j \in V$ with $d(p_i, p_i) = 0$, $d(p_i, p_j) \geq 0$ for $i \neq j$ and $d(p_i, p_j) = d(p_j, p_i)$.**

  For subsets V_1 and V_2 of V, distance between V_1 and V_2 is defined as $d(V_1, V_2) = \min_{p_i \in V_1, p_j \in V_2}\{d(p_i, p_j)\}$.

- **k-Clustering problem**

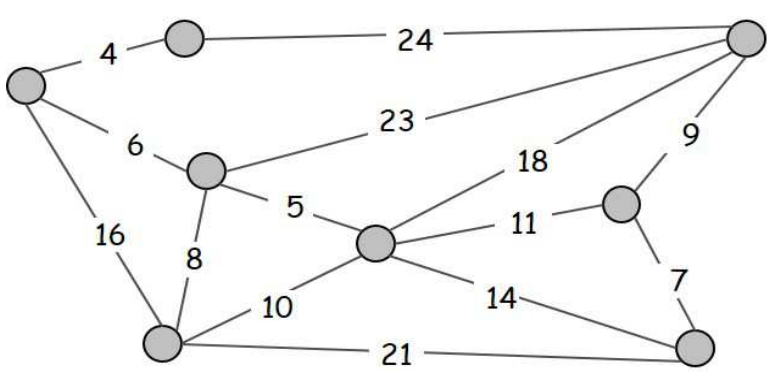
  Given a clustering instance and integer k, partition V into k subsets $V_1, .., V_k$ s.t. $\min_{i \neq j} d(V_i, V_j)$ is maximized. The problem can be solved by an algorithm for the minimum spanning tree problem.



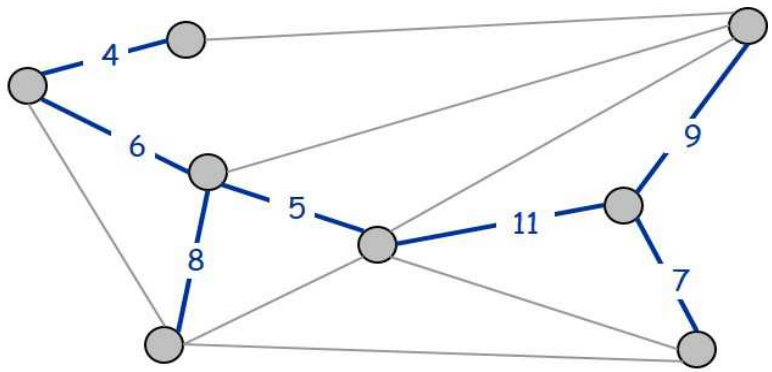
Minimum spanning tree problem

- **Let $G(V, E)$ be a connected graph. A subgraph T of G is a spanning tree of G if $V(T) = V(G)$ and T is a tree.**
- **If G is weighted (each edge e is assigned a non-negative weight $d(e)$ then each spanning tree T has a weight $\sum_{e \in E(T)} d(e)$.**

  Minimum spanning tree (MST) of G, a spanning tree T of G with the minimum weight.
- **Minimum spanning tree problem: Given G, find an MST of G.**



G = (V, E)



T, $\Sigma_{e \in T} c_e = 50$

Kruskal's Algorithm [Kruskal 1956]

Input: Weighted graph G.

Output: A MST of G.

$A = \emptyset$;

for each node $v \in V$ **create component** $c(v) = \{v\}$;

sort edges of G **into** $e_1, .., e_m$ **s.t.** $w(e_i) \leq w(e_j)$ **for** $i < j$;

for $i = 1, 2, .., m$ **do**

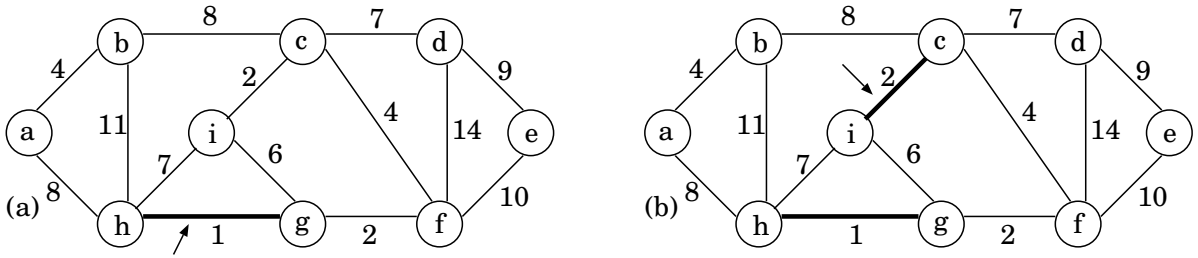
  if for edge $e_i = \{u, v\}$, $c(u) \neq c(v)$ **then**

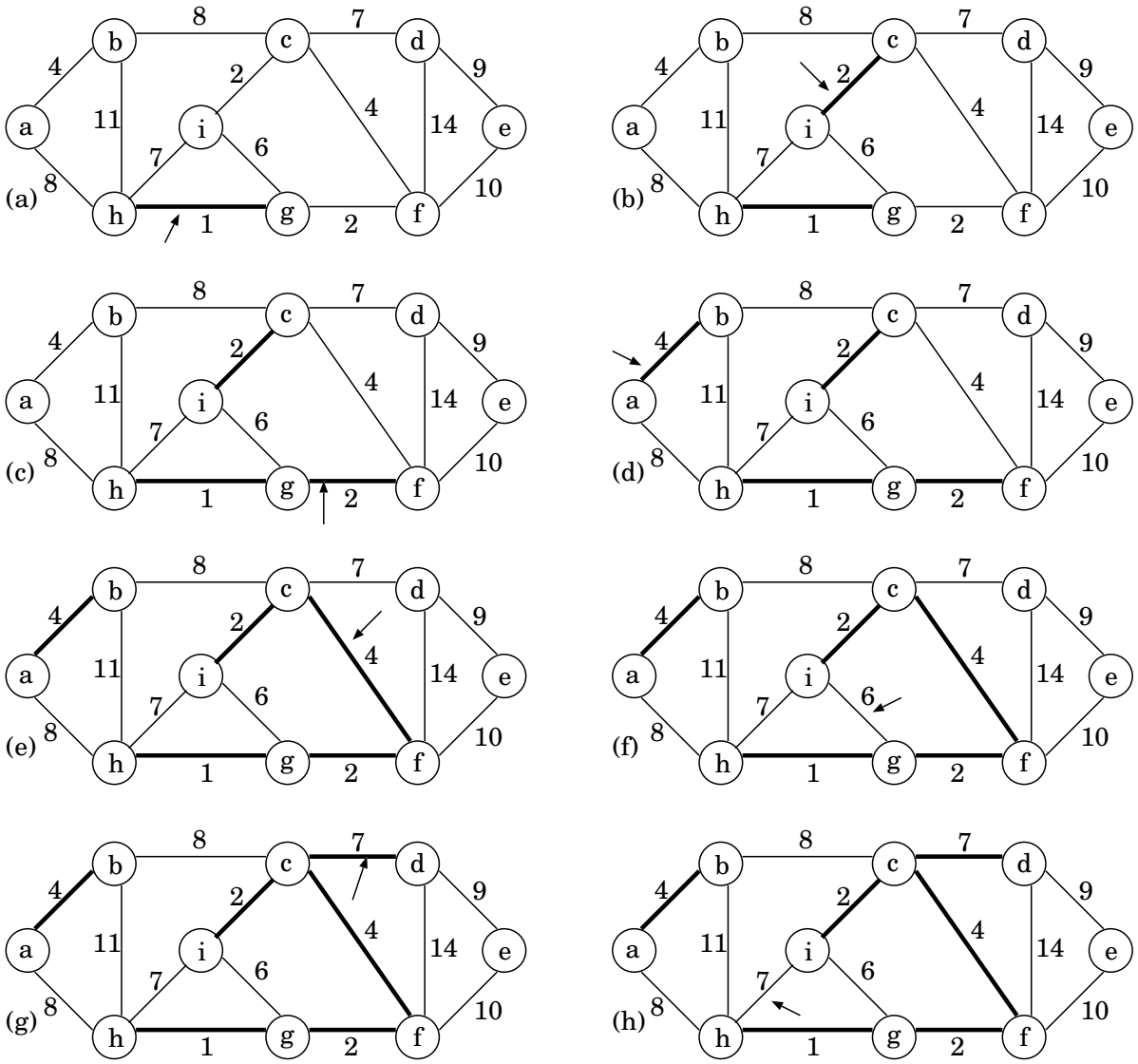
    $A = A \cup \{e_i\}$; $c(u) = c(u) \cup c(v)$; $c(v) = c(u)$;

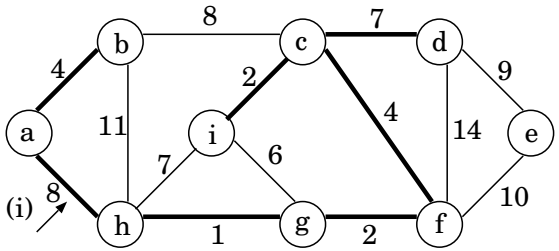
return A;

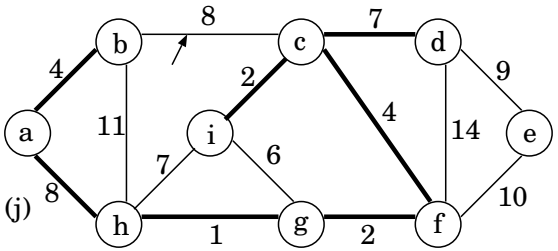
Disjoint-set data structure is used to maintain components of G_A.

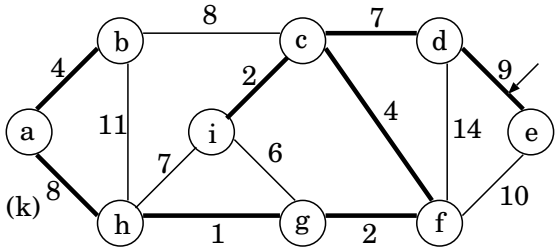
The running time of Kruskal's algorithm is $O(m \log n)$.

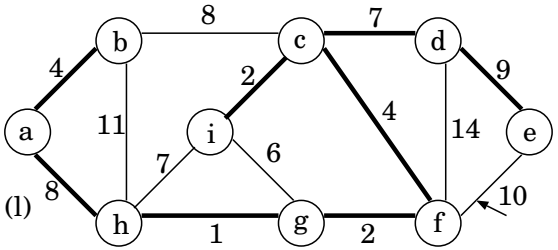


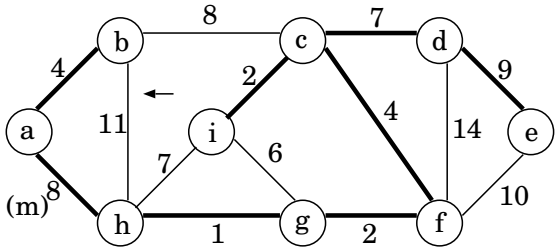

(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
4
9
14
7
6
10
1
2

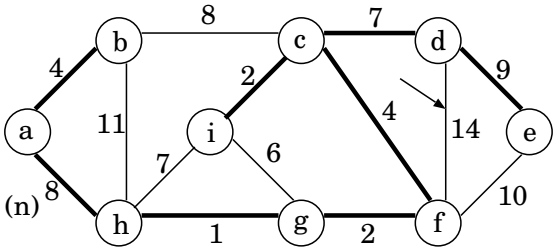

a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
7
6
8
10
1
2
(i)


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


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


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


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


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

Prim's Algorithm [Jarnik 1930, Dijkstra 1957, Prim 1959]

Input: Weighted graph G.

Output: A MST of G.

begin

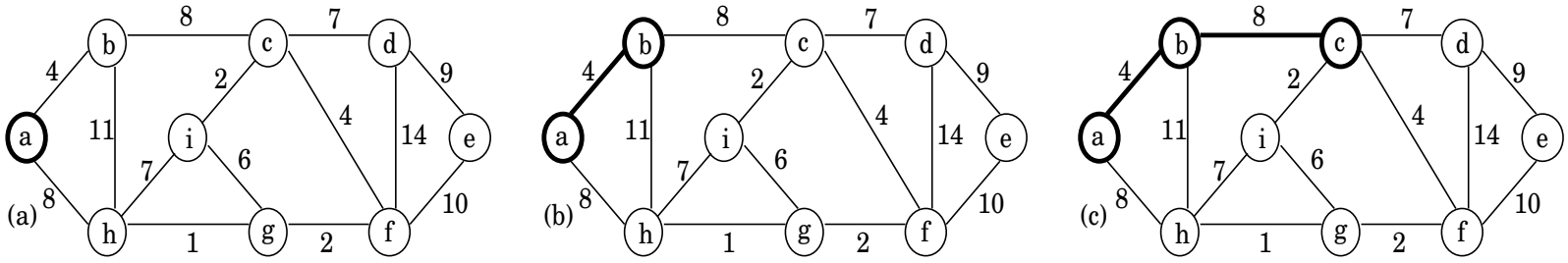
$S := \{s\}$; $E(T) := \emptyset$;

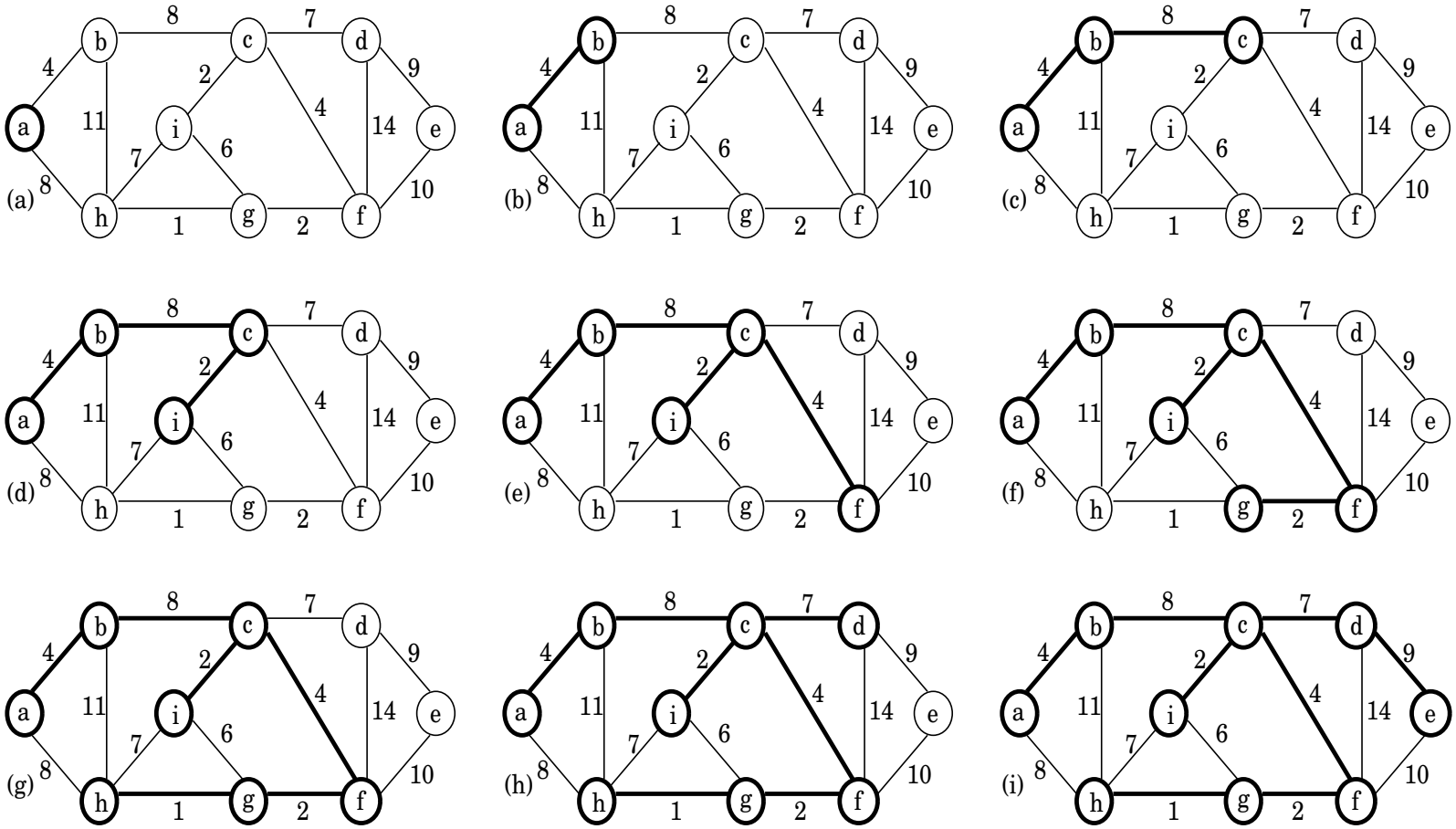
while $S \neq V(G)$ **do**

Choose an edge $e = \{u, v\}$ with $u \in S, v \in V(G) \setminus S$ and minimum $d(e)$;

$S := S \cup \{v\}$; $E(T) := E(T) \cup \{e\}$;

end while




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

MST and k-Clustering

- **Kruskal's algorithm and k-clustering problem.**

  Represent a clustering instance by a weighted complete graph G. Run Kruskal's algorithm with a data structure keeping the connected components created in the algorithm. Terminate the algorithm when the number of components becomes k.

- **MST and k-clustering problem.**

  Compute a MST T (e.g., by Prim's algorithm). Remove $k-1$ edges with the largest distances from T.