Analysis of Algorithms

Greedy Strategy

Hikmat Farhat

April 7, 2020

Job Scheduling

- Assume we have n jobs, each with weight w_i and length l_i, $1 \leq i \leq n$.
- The jobs share some resource (i.e. CPU) so we run them in a consecutive manner.
- If we run the jobs in the order $1, 2, 3, \ldots$ then job i has completion time $c_i = \sum_{k=1}^{i} l_k$.
- our goal is to minimize the quantity $f = \sum_{k=1}^{n} w_k \cdot c_k$
- In particular, we would like to have a greedy algorithm that minimizes f.
- To do that we start by looking at special cases

Special case 1

- In this special case we assume that all jobs have the same weight then

$$\begin{aligned} f &= w(l_1 + (l_1 + l_2) + (l_1 + l_2 + l_3) + \ldots + (l_1 + \cdots + l_n)) \\ &= w(n \cdot l_1 + (n-1) \cdot l_2 + (n-2) \cdot l_3 + \ldots + 1 \cdot l_n) \end{aligned}$$

- Clearly f will be minimized if we choose $l_1 \leq l_2 \leq \ldots \leq l_n$

Special case 2

- The second special case is when all jobs have the same length l then

$$f = l(w_1 + 2 \cdot w_2 + 3 \cdot w_3 + \ldots n \cdot w_n)$$

- Clearly, f will be minimized if $w_1 \geq w_2 \geq w_3 \geq \ldots \geq w_n$

looking for the general case

- It seems that f increases with w and decreases with l.
- we need a combination that behaves the same
- we can try $w_i - l_i$ and w_i/l_i
- Which one should be choose?
- We try another special case
- Suppose we have only two jobs: $w_1 = 2$, $l_1 = 1$ and $w_2 = 6, l_2 = 4$.
- if we choose $f = w - l$ then $f_1 = 1$ and $f_2 = 2$ this means that job 2 will start first and we get
- $6 \times 4 + 2 \times 5 = 34$.
- if we choose $f = w/l$ then $f_1 = 2$ and $f_2 = 1.5$ this means that job 1 will start first and we get
- $2 \times 1 + 6 \times 5 = 32$. Therefore the first method does not lead always to the optimal solution.
- Does the second method lead to the optimal solution?

Proof by exchane argument

- Let σ be an optimal sequence of jobs which is **not** greedy.
- Since σ is not greedy then it contains consecutive jobs i, k such that $w_i/l_i < w_k/l_k$.
- we can write

$$X_\sigma = \sum_{j \neq i,k} w_j \cdot c_j + w_i \cdot c_i + w_k \cdot c_k$$

- Let σ_1 be the sequence obtained from σ by exchanging the order of i and k.
- It is clear that the completion time of all jobs other than i and k remains unchanged. Let c be the sum of completion times of all jobs occurring before i then in the sequence σ we have $c_i = c + l_i$ and $c_k = c + l_i + l_k$

- In the sequence σ_1 we have $c_i = c + l_i + l_k$ and $c_k = c + l_k$.
- Then

$$\begin{aligned} X_\sigma - X_{\sigma_1} &= (w_i \cdot c + w_i \cdot l_i + w_k \cdot c + w_k \cdot l_i + w_k \cdot l_k) \\ &\quad - (w_i \cdot c + w_i \cdot l_i + w_i \cdot l_k + w_k \cdot c + w_k \cdot l_k) \\ &= w_k \cdot l_i - w_i \cdot l_k > 0 \\ &\Rightarrow X_\sigma > X_{\sigma_1} \end{aligned}$$

- The last two lines follow from $w_i/l_i < w_k/l_k \Rightarrow w_i \cdot l_k < w_k \cdot l_i$.
- This is a contradiction because σ was assumed to be optimal.

Interval Scheduling

- Consider a set of n intervals (s, e) where s and e are the starting and ending time respectively.
- We would like to choose a non-overlapping subset of those intervals such that the total number of selected intervals is maximum.
- For example, consider the intervals $\{(1, 5), (2, 7), (5, 8)\}$. The largest subset of non-overlapping intervals is $\{(1, 5), (5, 8)\}$.

We are looking for a greedy solution to this optimization problem. What property of the intervals should the greedy method select? There are many options:

1. shortest interval first
2. The interval with the smallest starting time
3. The interval with the smalles number of overlaps
4. etc...

Shortest iterval first

Figure: Shortest interval counterexample

Earliest starting time first

Figure: counterexample for earliest starting time first

Smallest overlap first

Figure: Smallest overlap first

Greedy Solution

The greedy solution consists of choose the next compatible interval with the smallest finishing time. We build a min-heap based on finishing times. Let I be the set of intervals and T the desired solution

$Q \leftarrow I$
$T \leftarrow \emptyset$
$last \leftarrow -1$
while $Q \neq \emptyset$ **do**
    $(s, f) \leftarrow$ `Delete-Min(Q)`
    if $s \geq last$ **then**
        $T \leftarrow T \cup \{(s, f)\}$
        $last \leftarrow f$
    end
end

- Let I be the set of intervals. A solution to the interval scheduling is a subset $S = \{s_1, f_1), \ldots, (s_k, f_k)\} \subseteq I$ such that for all $i < j$ we have $f_i \leq s_j$.
- Our goal is to find the optimal solution, i.e. the solution with the largest number of intervals.
- We will show that the greedy solution is optimal in the sense that the number of intervals in the greedy solution is equal to the size of the optimal solution.
- Let $G = \{s_1, f_1), \ldots, (s_k, f_k)\}$ be the greedy solution and $O = \{(\gamma_1, \phi_1), \ldots, (\gamma_m, \phi_m)\}$ be an optimal solution.
- First we need the following lemma:

Lemma: $\forall i,\ f_i \leq \phi_i$.

Proof of lemma

We prove the lemma by induction.
base case: clearly $f_1 \leq \phi_1$ since f_1 is the smallest value of all finishing times as chosen by the greedy algorithm.
hypothesis: assume $f_i \leq \phi_i$
induction step: Since both G and O are solutions then we have $f_i \leq s_{i+1}$ and $\phi_i \leq \gamma_{i+1}$. Using the induction hypothesis $f_i \leq \phi_i$ it follows that $f_i \leq \gamma_{i+1}$.
This means that (s_i, f_i) and $(\gamma_{i+1}, \phi_{i+1})$ are compatible. Now the greedy algorithm always chooses the next compatible interval with the smallest finishing time thus $f_{i+1} \leq \phi_{i+1}$ which completes the proof.

Greedy solution is optimal

By way of contradiction, assume that the greedy solution $G = \{s_1, f_1), \ldots, (s_k, f_k)\}$ is not optimal then there exists an optimal solution $O = \{(\gamma_1, \phi_1), \ldots, (\gamma_m, \phi_m)\}$ with $k < m$. This means that the greedy algorithms stops after adding (s_k, f_k) to the solution. In other words, when the greedy algorithm removes (s_k, f_k) from the queue there are no more intervals in the queue that are compatible with (s_k, f_k). Using the previous lemma we know that $f_k \leq \phi_k$ and since (γ_k, ϕ_k) is compatible with $(\gamma_{k+1}, \phi_{k+1})$ then $(\gamma_{k+1}, \phi_{k+1})$ is compatible with (s_k, f_k) which is a contradiction. Thus the greedy solution is optimal.

Fractional Knapsack

- Given n items having value $v_1, \ldots, v_n$ and weights $w_1, \ldots, w_n$ and a knapsack of size W
- We need to maximize

$$\sum_{i=1}^{n} x_i \cdot v_i$$

- Subject to the condition

$$\sum_{i=1}^{n} x_i \cdot w_i \leq W$$

- Where $0 \leq x_i \leq 1$ is a faction of item i that is used.

Greedy Solution

- A greedy solution is obtained by adding repeatedly items with biggest ration v/w until the next item does not "fit" in knpacksack so we add a fraction of it.

Proof of correctness

- Consider a knapsack of size W and n items with weight w_i and value v_i relabeled such that

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$$

- Let $O_{k-1} =< y_1, \ldots, y_m >$ be an optimal solution where $y_j = 1$ for $0 \leq j \leq k-1$ and $0 \leq y_j \leq 1$ for $k \leq j \leq m$.
- Construct a new solution O_k as follows.
- For $i = 1$ to $k-1$ the fractions for O_k are the same $y'_i = y_i$
- Replace y_k by 1, i.e. $y'_k = 1$. This increases the total weight of the solution by $(1 - y_k)w_k$.
- To compensate we need to subtract this value which we distribute evenly over the remaining values $y_{k+1}, \ldots, y_m$
- Therefore for $i = k+1$ to m replace y_i by $y'_i = y_i - \frac{(1-y_k)w_k}{(m-k-1)w_i}$
- You can check that $\sum_{i=1}^{m} y'_i w_i = \sum_{i=1}^{m} y_i w_i$.

- The total value for O_k is

$$\sum_{i=1}^{k-1} y_i' v_i + y_k' v_k + \sum_{i=k+1}^{n} y_i' v_i$$

- Replacing the value of y_i' in each part we get

$$\sum_{i=1}^{k-1} y_i v_i + v_k + \sum_{i=k+1}^{m} y_i v_i - \frac{(1-y_k)w_k}{m-k-1} \sum_{i=k+1}^{m} \frac{v_i}{w_i}$$

- Subtracting the value of O_{k-1} from O_k we get

$$(1-y_k)v_k - \frac{(1-y_k)w_k}{m-k-1}\sum_{i=k+1}^{m}\frac{v_i}{w_i}$$

- Since $\frac{v_i}{w_i} \leq \frac{v_k}{w_k}$ for all $i > k$ then

$$\sum_{i=k+1}^{m}\frac{v_i}{w_i} \leq (m-k-1)\frac{v_k}{w_k}$$

$$\frac{(1-y_k)w_k}{m-k-1}\sum_{i=k+1}^{m}\frac{v_i}{w_i} \leq (1-y_k)v_k$$

- Therefore $Ok - O_{k-1} \geq 0$ and O_k is optimal.

- Recall that O_{k-1} is an optimal solution where the first $k-1$ fractions are 1.
- The above procedure can be performed to show that $O_{k+1} \ldots O_{n-1}$ is optimal and therefore greedy is optimal.

Huffman Coding

- Hauffman coding is a greedy recursive algorithm to obtain optimal prefix code given an alphabet with frequencies of occurrences

- First a prefix code is equivalent to a binary tree so once we build the optimal binary tree then we "read off" the encoding.
- The basic idea of HC is to build the optimal tree recursively in a greedy manner
  1. The optimal tree T for k symbols is obtained by constructing the optimal tree T' for $k-1$ symbols where T' is the same as T except replacing the two nodes with the smallest frequencies in T, x and y by a single node having the sum of the frequencies: $f_w = f_x + f_y$

$$T' = T - \{x, y\} \cup \{w\}$$

The optimal tree has the following properties:

- It is full. Suppose that y is a single child of node w. By replacing w by y we obtain a "better" tree
- For any two leaves x, y if $f_x > f_y$ then $d_x < d_y$. This can be shown by an exchange argument.

Note that there could be many optimal trees. Let x, y be the symbols with the least frequencies then there exists an optimal tree in which x, y are siblings.

Huffman coding: Python Code

```
import queue
class Node(object):
    def __init__(self, left=None, right=None):
        self.left = left
        self.right = right
    def children(self):
        return((self.left, self.right))
freq = [
    (25,'a'), (24, 'b'), (28, 'c'), (18, 'd'),(5, 'e')]

def create_tree(frequencies):
    p = queue.PriorityQueue()
    for value in frequencies:
        p.put(value)
    while p.qsize() > 1:
        l, r = p.get(), p.get()
        node = Node(l, r)
        p.put((l[0]+r[0], node))
    return p.get()
```

```
# Recursively walk the tree down to the leaves,
#   assigning a code value to each symbol

def walk_tree(node, prefix=""):

    if isinstance(node[1],Node):
        l,r=node[1].children()
        walk_tree(l,prefix+"1")
        walk_tree(r,prefix+"0")
    else:
        code[node[1]]=prefix

node = create_tree(freq)
code={}
walk_tree(node)
```

Properties of optimal coding trees

- An optimal coding tree is full. This is true because if a node is a single child it could replace its parent and reduce the encoding by 1.

Theorem

For all nodes x, y in an optimal coding tree if $f_x > f_y$ then $d(x) \leq d(y)$.

Proof.

By way of contradiction suppose for an optimal tree T, $\exists x, y$ such that $f_x > f_y$ and $d(x) > d(y)$. Let $A(T)$ be the average number of bits for T an construct T' from T by swapping the places of nodes x and y. Then

$$\begin{aligned} A(T) - A(T') &= f_x \cdot d(x) + f_y \cdot d(y) - f_x \cdot d(y) - f_y \cdot d(x) \\ &= (f_x - f_y)(d(x) - d(y)) \end{aligned}$$

since $f_x > f_y$ and $d(x) > d(y)$ then $A(T) > A(T')$ which is a contradiction since T is optimal $\square$

Theorem

Let x and y be the symbols with the smallest frequencies in the alphabet. There exists an optimal tree in which the nodes corresponding to x and y are siblings.

Proof.

We know that the node corresponding to the lowest frequency has a sibling, say w. Since y is the second lowest then $f_y \leq f_w$. By swapping the nodes of y and w we obtain, according to the previous theorem, a tree that is at least as good as the original, therefore it is optimal. □

Huffman coding: Optimality

To show that Hauffman coding is optimal we need first to find an expression for the average number of bits as a function of the size of the tree. Let T be a Hauffman tree for an alphabet S of size k whose two lowest frequency letters are x and y. Let T' be the tree obtained by replacing x and y in T by w with $f_w = f_x + f_y$. The average number of bits used for a given encoding can be written as

$$
\begin{aligned}
A(T) &= \sum_{u \in S} f_u \cdot depth_T(u) \\
&= \sum_{u \neq x,y} f_u \cdot depth_T(u) + f_x \cdot depth_T(x) + f_y \cdot depth_T(y) \\
&= \sum_{u \neq x,y} f_u \cdot depth_T(u) + (f_x + f_y) \cdot (depth_{T'}(w) + 1) \\
&= \sum_{u \neq x,y} f_u \cdot depth_T(u) + (f_w) \cdot (depth_{T'}(w) + 1) \\
&= \sum_{u \in S'} f_u \cdot depth_{T'}(u) + f_w \\
&= A(T') + f_w
\end{aligned}
$$

We prove by induction on the size of the alphabet that the Huffman algorithm produces optimal code. Clearly, if the alphabet contains only two letters then Huffman codes produces the optimal result since each letter will use only one bit.
Hypothesis: assume that our algorithm produces optimal codes for all alphabets of size up to $k-1$. Consider an alphabet S with size k where x and y are the letters in S that have the smallest frequencies.
Assume by way of contradiction that the Huffman tree is not optimal. From a previous theorem we know that there is an optimal tree Z in which the nodes corresponding to the two lowest frequencies, x and y, are siblings. Now construct the tree Z' by removing x and y from Z and replacing it with w such that $f_w = f_x + f_y$. From our previous result we know that $A(Z) = A(Z') + f_w$ and $A(T) = A(T') + f_w$. Combining both results we get that $A(Z') < A(T')$ which is a contradiction since the hypothesis states that T' is optimal.

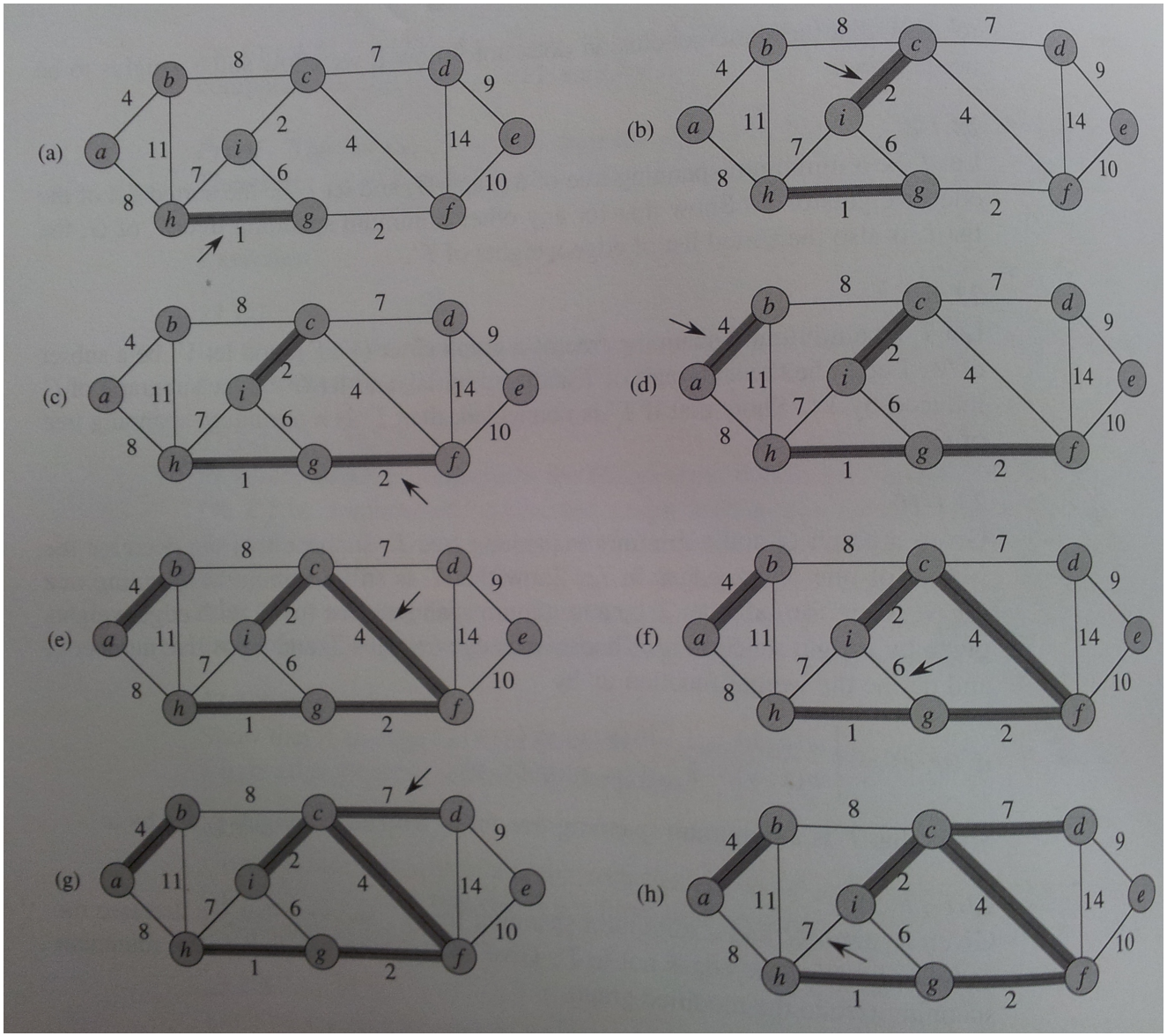
Minimum Spanning Tree

- In many application, when the system is represented by a graph we need to find a **Minimum Spanning Tree** (MST).
- As the name suggest this collection of nodes is
  1. **A tree**.
  2. **Spanning**. meaning includes all the nodes of the graph.
  3. It has the **least total cost** of all such trees.

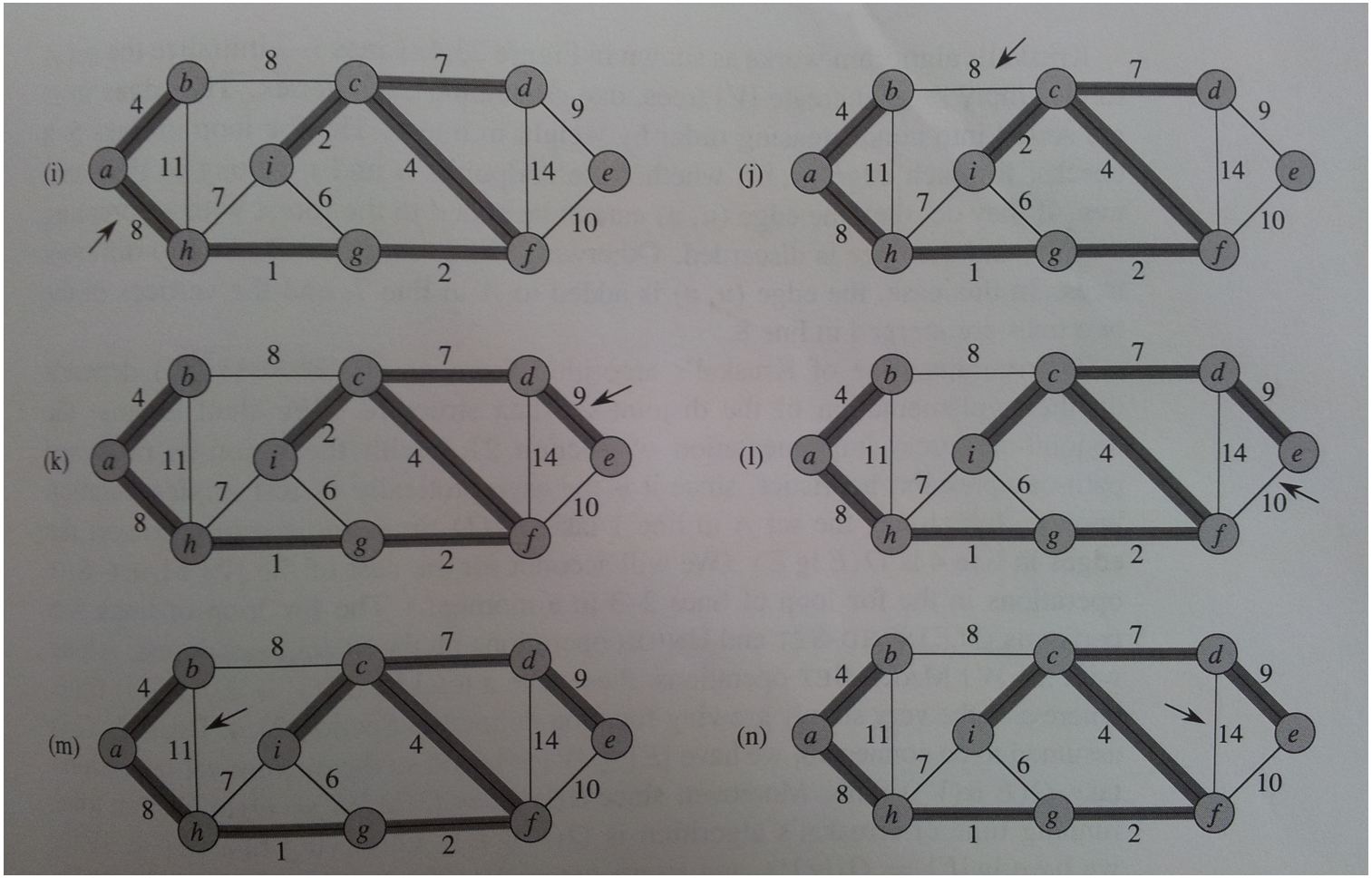
MST:Kruskal's Algorithm

- Kruskal's algorithm computes a MST of a given graph.
- Every edge has an associated weight or cost.
- The idea is to build the MST by adding an edge every iteration.
- The edges are considered by increasing order.
- An edge is added if it doesn't create a cycle.
- The algorithm stops when there are no more edges to consider.

Example



Example



```
MST-KRUSKAL(G)
A ← ∅
foreach v ∈ V do
    MAKE-SET(v)
end
F ← SORT-EDGES(E)
foreach (u, v) ∈ F do
    if FIND-SET(u) ≠ FIND-SET(v) then
        A ← A ∪ {(u, v)}
        UNION(u, v)
    end
end
```

Correctness of Kruskal's algorithm

- Let $T = \{e_1, \ldots, e_{n-1}\}$ be the tree obtained using Kruskal's algoritm, in that order.
- Let T_o be an mst that shares $e_1, \ldots, e_{k-1}$ with T but not $e_k, \ldots, e_{n-1}$.
- Construct the graph $G = T_o \cup \{e_k\}$ which clearly has a cycle C passing by e_k.
- C contains an edge $e \notin \{e_1, \ldots, e_k\}$ because Kruskal's algorithm does not create a cycle
- Furthermore $w_e \geq w_{e_k}$ because otherwise the algorithm would have chosen e instead of e_k.
- Now construct the tree $T_{o_2} = T_o - \{e\} \cup \{e_k\}$ which is clearly an mst that shares k edges with T
- repeating the above procedure we obtain that T is an mst.

Single Source Shortest Path

- Given a graph $G = (V, E)$ with a real-valued weight function w we often as the question:
- What is the minimal cost (shortest) path from $s \in V$ to all other vertices of the graph.
- First we need some definitions and theorems.

- Given a graph $G = (V, E)$ and a real-valued weight function $w : E \to R$.
- weight of path $p = (v_0, \ldots, v_k)$ sometimes written as

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- The shortest path cost δ

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from u to v} \\ \infty & \text{otherwise} \end{cases}$$

Properties of Shortest Path

- Subpaths of shortest path are subpath: Given a graph $G = (V, E)$ and weight function $w : E \to \mathbf{R}$ let $p = (v_1, \ldots v_k)$ be a shortest path from v_1 to v_k then for any $1 \leq i, j \leq k$, $p_{ij} = (v_i, \ldots, v_j)$ is a shortest path from v_i to v_j.
- **Proof**: we write $v_1 \overset{p}{\rightsquigarrow} v_k$ which can be decomposed into $v_1 \overset{p_i}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_j}{\rightsquigarrow} v_k$
- Then $w(p) = w(p_i) + w(p_{ij}) + w(p_j)$ so if p_{ij} is not the shortest path then $\exists p'_{ij}$ with $w(p'_{ij}) < w(p_{ij})$ then we can write
- $w(p') = w(p_i) + w(p'_{ij}) + w(p_j) < w(p)$ a contradiction since p is the shortest path from v_1 to v_k.

Negative weight

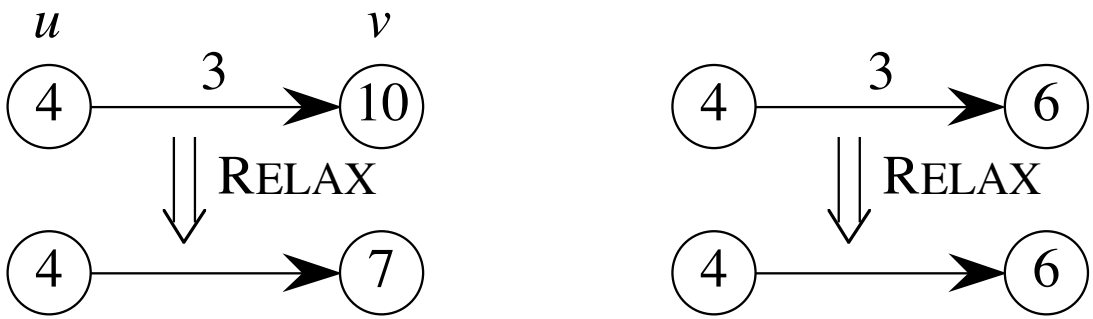
- Even if a path contains edges with negative weight a shortest path can still be defined.
- It is undefined if the path contains a negative weight **cycle**.
- This is because we can "cross" the cycle as many times as we want, every time lower the cost.
- Therefore in the case when there is a negative cycle on a path from u to v then we set $\delta(u, v) = -\infty$ where $\delta(a, b)$ is the shortest path cost from a to b.

Representation of Shortest Paths

- In all the algorithms that we will deal with, we maintain for every vertex v its predecessor $v.p$ (which could be NULL)
- At **termination** $v.p$ will be the predecessor of v on a shortest path from source s to v.
- We also maintain a value $v.d$ which at termination will be the value of the shortest path cost from source s to v.
- During the execution of the algorithm $v.d$ will be **an upper bound** on the value of the shortest path cost.

Relaxation

- **Relaxing** an edge (u, v) means testing if we can improve the shortest path cost of v by using the edge (u, v).
- If we can then we update $v.d$ and $v.p$.



- In the figure to the left the cost of v was changed to the new cost (7) whereas to the right it was not changed since the new cost (7) is bigger than the current (6).
- What is NOT shown is the change to $v.p$ in the first case.

Initialization and Relaxation

- Initially all vertices (except the source) have cost ∞ and no predecessors (including the source).

```
INITIALIZE(G,s)
foreach v ∈ V do
    v.d ← ∞
    v.p ← NULL
end
s.d ← 0

RELAX(u,v)
if v.d > u.d + w(u,v) then
    v.d ← u.d + w(u,v)
    v.p ← u
end
```

Dijkstra's Algorithm

- Dijkstra's algorithm is another single source shortest path.
- It works when all weights are **positive**.
- We will see that it is faster than the Bellman-Ford algorithm.
- It maintains a set S of nodes whose shortest paths have been determined
- All other nodes are kept in a min-priority queue to keep track of the next node to process.

Dijkstra Pseudo Code

```
DIJKSTRA(G,s);
INITIALIZE(G,s)
S ← ∅
Q ← V
while Q ≠ ∅ do
    u ← EXTRACT-MIN(Q)
    S ← S ∪ {u}
    foreach v ∈ Adj[u] do
        RELAX(u,v)
    end
end
```

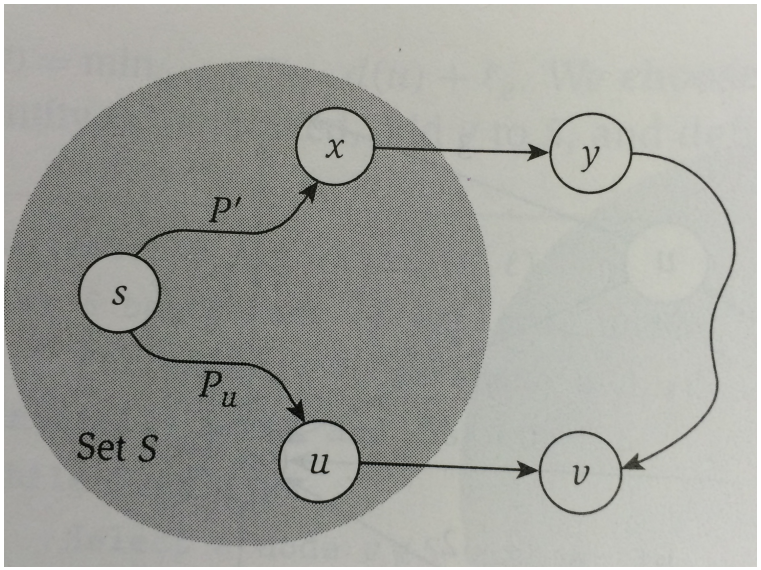
Example

Complexity

- The running time of Dijkstra's algorithm depends on the implementation of the queue.
- Using a min-heap on a sparse graph gives complexity of $O((V + E) \log V)$.
- This is because the while loop executes V times. The extract-min is $O(\log V)$ for a cost of $V \log V$. The relax includes an key update which means $\log V$. Since each edge is relaxed at most once then the total is E with a cost of $E \log V$.

Correctness

We show that for every iteration Dijkstra's algorithm adds node u then $u.d$ is the shortest path distance from the source to u. Note that the algorithm adds a node to S and removes it from Q. Base case: initially it adds the source and $s.d = 0$ which is correct. Assume that all added nodes are such that $u.d$ is shortest path. Induction step: the next step the algorithm adds a node v such that $v.d$ is minimum among all nodes in the set Q (see figure below).



Suppose that the path $s \rightsquigarrow u \rightarrow v$ is not the shortest path. Then there exists a shorter path $s \rightsquigarrow x \rightarrow y \rightsquigarrow v$. But $y.d = x.d + c(x, y) > v.d$, otherwise the algorithm would have selected y. Since the algorithm is used with positive costs only then the cost of the path $s \rightsquigarrow x \rightarrow y \rightsquigarrow v$ is greater than $s \rightsquigarrow u \rightarrow v$