CST207
DESIGN AND ANALYSIS OF ALGORITHMS

Lecture 7: The Greedy Approach

Lecturer: Dr. Yang Lu

Email: luyang@xmu.edu.my

Office: A1-432

Office hour: 2pm-4pm Mon & Thur

Outline

- The Coin Change Problem
- Minimum Spanning Trees
- Dijkstra’s Algorithm for Single-Source Shortest Paths
- Scheduling
- Huffman Code
- The Knapsack Problem

The Greedy Approach

- The greedy approach grabs data items in sequence, *each time taking the one that is deemed "best" according to some criterion, without regard for the choices it has made before or will make in the future*.
  - Don't think greedy approach is evil due to its name "greedy" with negative meaning. It often lead to very efficient and simple solution.
- Compared with dynamic programming, the greedy approach is much more straigtforward.
  - No recursive property is needed.
- Each time, select the step with *local optimal*.
  - No gaurantee of global optimal, which should be determined based on the problem.

THE COIN CHANGE PROBLEM

The Coin Change Problem

- Given an amount N and unlimited supply of coins with denominations $d_1, d_2, \ldots, d_n$, compute the smallest number of coins needed to get N.
- Example:
  - For $N = 86$ (cents) and $d_1 = 1, d_2 = 2, d_3 = 5, d_4 = 10, d_5 = 25, d_6 = 50, d_7 = 100.$
  - The optimal change is: one 50, one 25, one 10, and one 1.
- Can greedy approach obtain optimal solution?

The Coin Change Problem

- The greedy approach:
  1. Select the largest coin.
  2. Check if adding the coin makes the change exceed the amount.
     a. No, add the coin.
     b. Yes, set the largest coin as the second largest coin and go back to step 1.
  3. Check if the total value of the change equals the amount.
     a. No, go back to step 1.
     b. Yes, problem solved.

The Coin Change Problem

- Successful example:
  - For $N = 86$ (cents) and $d_1 = 1, d_2 = 2, d_3 = 5, d_4 = 10, d_5 = 25, d_6 = 50, d_7 = 100.$
  - The greedy approach is optimal: 50, 25, 10, 1.
- Failed example:
  - For $N = 86$ (cents) and $d_1 = 1, d_2 = 2, d_3 = 5, d_4 = 10, d_5 = 18, d_6 = 25, d_7 = 50, d_8 = 100.$
  - The greedy approach is not optimal: 50, 25, 10, 1.
  - The optimal solution: 50, 18, 18.
- For this problem, the greedy approach does not gaugantee an optimal solution.
  - For each problem, we should first analyze it that whether the greedy approach can always yield an optimal solution.

Process of the Greedy Approach

- A greedy algorithm starts with an empty set and iteratively adds items to the set in sequence until the set represents a solution to an instance of a problem.
- Each iteration consists of the following components:
  - A **selection procedure** chooses the next item to add to the set. The selection is performed according to a greedy criterion that satisfies some locally optimal consideration at the time.
    - E.g. select the largest coin.
  - A **feasibility check** determines if the new set is feasible by checking whether it is possible to complete this set in such a way as to give a solution to the instance.
    - E.g. whether exceed the amount.
  - A **solution check** determines whether the new set constitutes a solution to the instance.
    - E.g. whether equal the amount.

MINIMUM SPANNING TREES

Undirected Graph

- We denote an undirected graph as $G = (V, E)$.
- An undirected graph is called *connected* if there is a path between every pair of vertices.
- A path from a vertex to itself, which contains at least three distinct vertices, is called a *simple cycle*.
- An undirected graph with no simple cycles is called *acyclic*.
- A *tree* is an acyclic, connected, undirected graph.

(a) A connected, weighted, undirected graph G.

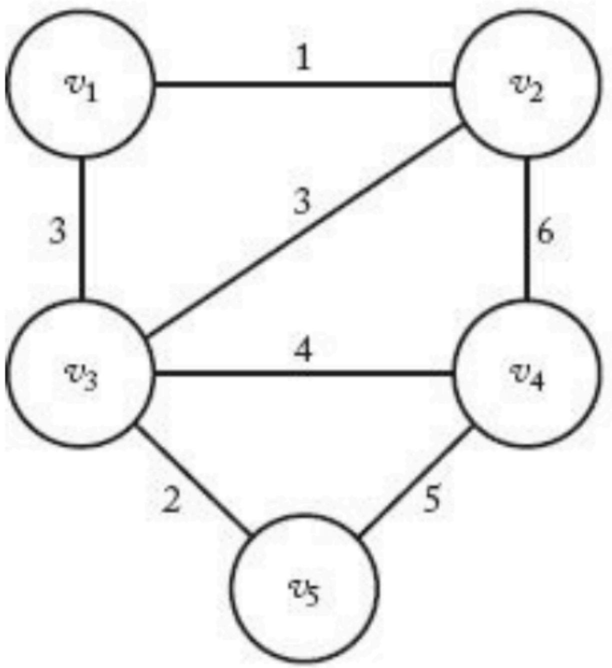


Image source: Figure 4.3, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Minimum Spanning Tree

- A *spanning tree* for G is a connected subgraph that contains all the vertices in G and is a tree.
  - Figure (c) and (d) are spanning trees of G.
- A spanning tree with minimum weight is called a *minimum spanning tree*.
- Our goal is to develop an algorithm to construct the minimum spanning tree from a undirected weighted graph G.
- In this example:
  - $V = \{v_1, v_2, v_3, v_4, v_5\}$.
  - $E = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_2, v_4), (v_3, v_4), (v_3, v_5), (v_4, v_5)\}$.

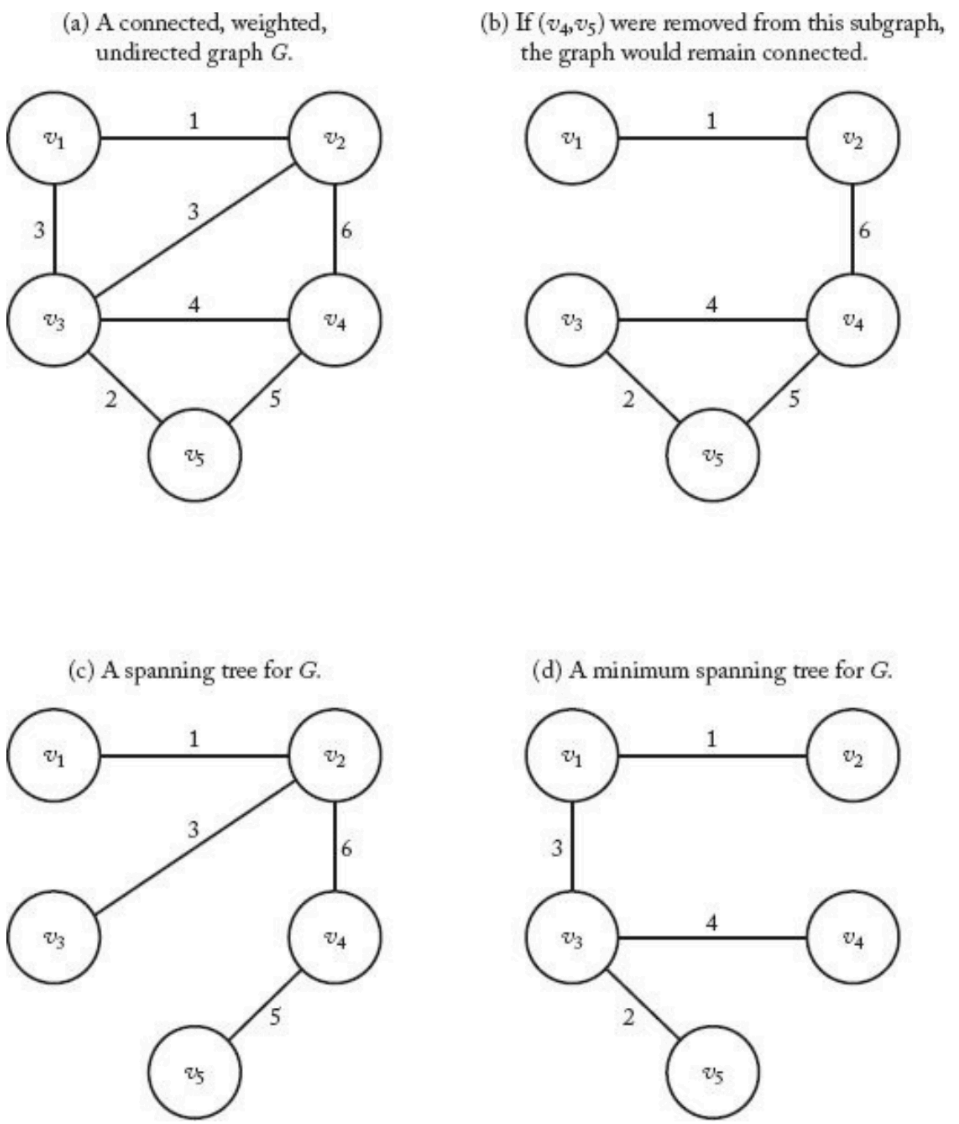


Image source: Figure 4.3, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Prim’s Algorithm

High level pseudocode:

- Initialize $F = \emptyset$ and $Y = \{v_1\}$.
- Iterate when the instance is not solved:
  - Select a vertex in $V - Y$ that is nearest to Y.
  - Add the vertex to Y.
  - Add the edge to F.
  - Check whether $Y == V$.
    - Yes, the instance is solved.

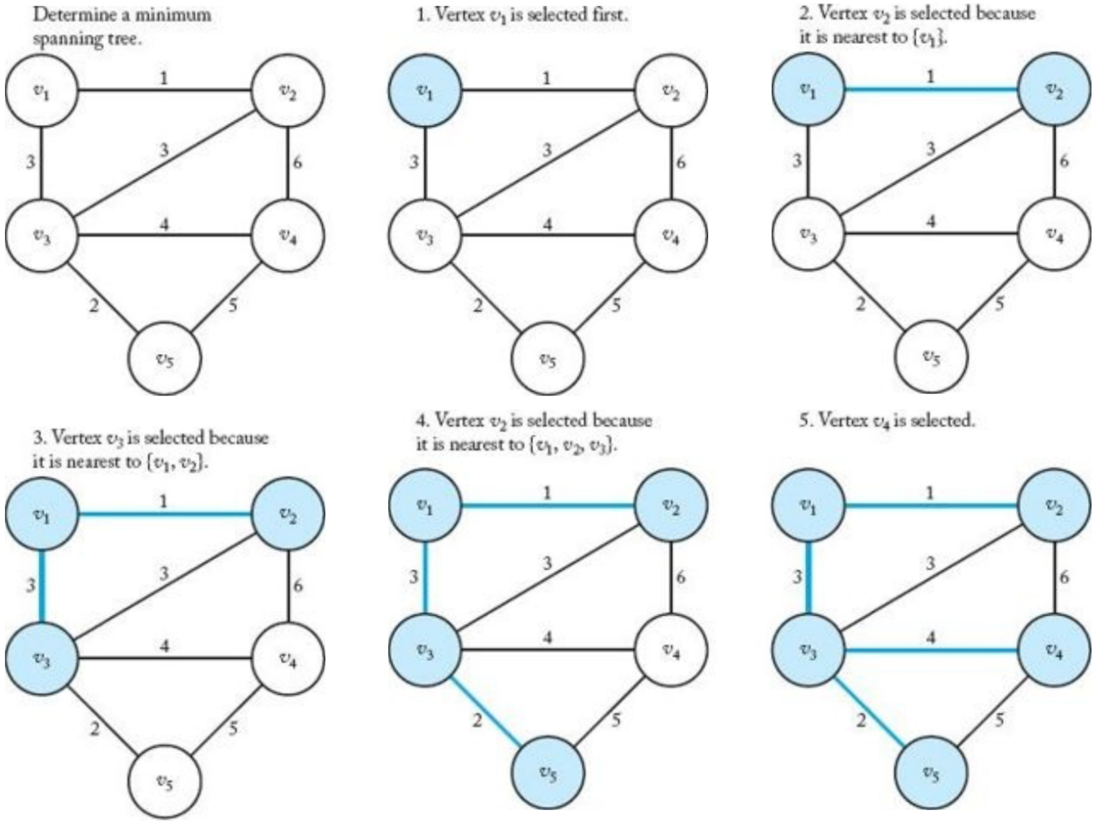


Image source: Figure 4.4, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Pseudocode of Prim's Algorithm

- Every-case time complexity:

$$T(n) = 2(n-1)(n-1) \in \Theta(n^2)$$

```
void prim (int n,
          const number W[][],
          edge_set& F)
{
    index i;
    number min;
    edge e;
    index vnear; // index of selected vertex to be added
    index nearest[2...n]; //index of the vertex in Y nearest to v_i
    number distance[2...n]; // weight on edge between v_i and v_nearest[i]

    F = ∅;
    for (i = 2; i <= n; i++){
        nearest[i] = 1;
        distance[i] = W[1][i];
    }

    repeat (n - 1 times){
        min = inf;
        for (i = 2; i <= n; i++)
            if (0 <= distance[i] < min){
                min = distance[i];
                vnear = i;
            }
        e = edge between v_vnear and v_nearest[vnear];
        add e to F;
        distance[vnear] = -1;
        for (i = 2; i <= n; i++)
            if (W[i][vnear] < distance[i]){
                distance[i] = W[i][vnear];
                nearest[i] = vnear;
            }
    }
}
```

Optimality

- It is easy to develop a greedy algorithm, but difficult to prove whether or not a greedy algorithm always produces an optimal solution.

Optimality Proof for Prim's Algorithm

Theorem 1

Prim's algorithm correctly computes an minimum spanning tree.

Proof:

- Prove by induction.
- The induction hypothesis: after each iteration, the tree F is a subgraph of some minimum spanning tree T.
- Basis step: it is trivially true at the start, since initially F is just a single node and no edges.
- Induction Step:
  - Now suppose that at some point in the algorithm we have F which is a subgraph of T, and Prim's algorithm tells us to add the edge e. We need to prove that $F \cup \{e\}$ is also a subtree of some minimum spanning tree.

Image source: Figure 4.3, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Optimality Proof for Prim's Algorithm

Proof (cont'd):

- We discuss in two cases: $e \in T$ and $e \notin T$.
- If $e \in T$:
  - It is clearly true, since by induction F is a subtree of T and $e \in T$ and thus $F \cup \{e\}$ is a subtree of T.
- If $e \notin T$:
  - Adding e to T creates a cycle, because adding any edge to a spanning tree creates a cycle.
  - Since e has one endpoint vertex in F and one endpoint vertex not in F, there has to be some other edge e' in this cycle that has exactly one endpoint in F.
  - So Prim's algorithm could have added e' but instead chose to add e, which means that the weight of e must be smaller or equal to e'.

Optimality Proof for Prim's Algorithm

Proof (cont'd):

- If we add e to T and remove e' we will obtain a new tree T' with smaller total weight than T.
- However, it is impossible because T is a minimum spanning tree, which indicate the weight of e must be equal to the weight of e'.
- Therefore, T' is also a minimum spanning tree and $F \cup \{e\}$ is a subtree of some minimum spanning tree.
- This maintains the induction, so proves the theorem.

Kruskal's Algorithm

High level pseudocode:

- $F = \emptyset$.
- Create disjoint subsets of V, one for each vertex and containing only that vertex.
- Sort the edges in E in nondecreasing order.
- Iterate when the instance is not solved:
  - Select the next edge in order;
  - Check whether the edge connects two vertices in disjoint subsets.
    - Yes, merge the subsets and add the edge to F.
  - Check whether all the subsets are merged.
    - Yes, the instance is solved.

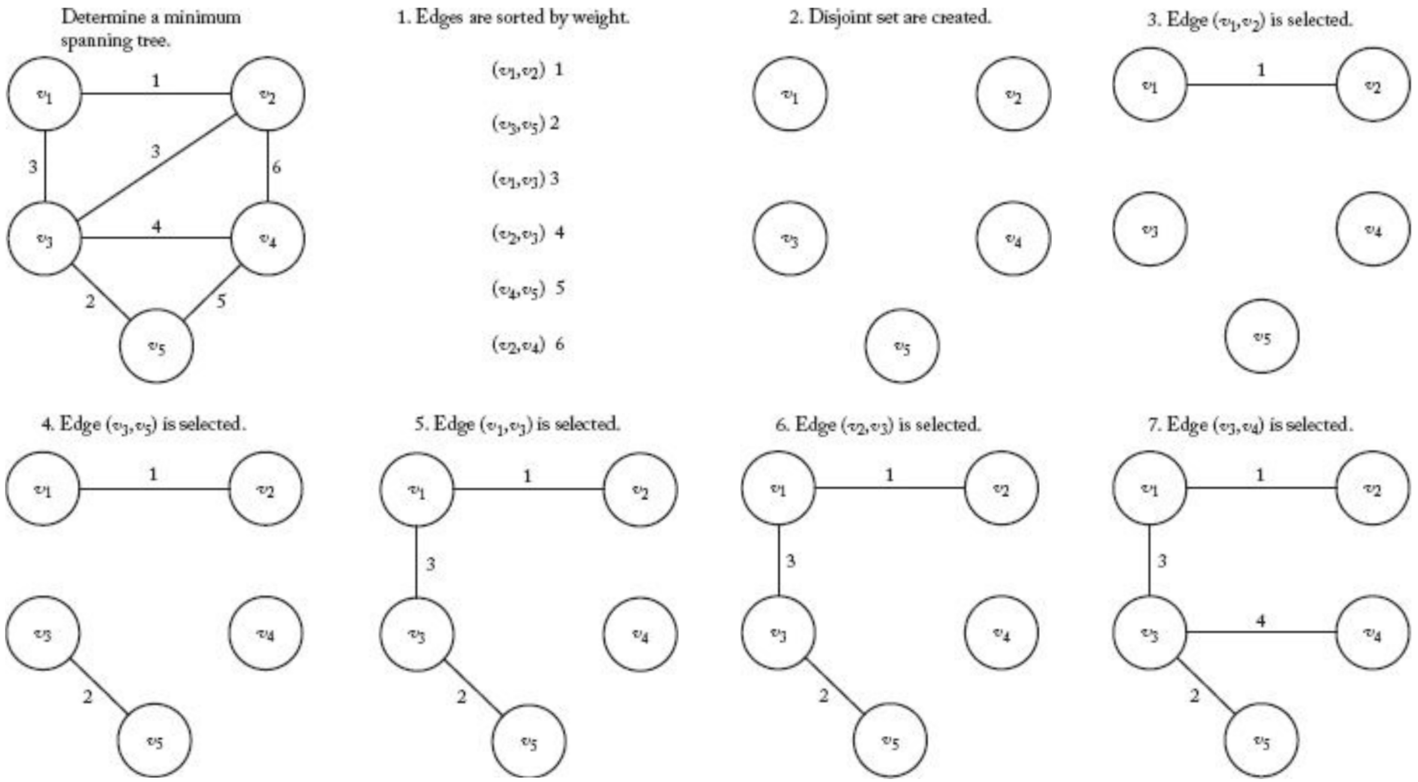


Image source: Figure 4.7, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Pseudocode of Kruskal's Algorithm

- We first define a data structure to represent disjoint set and use set_pointer to refer.
- The following functions will be used:
  - *initial*(*n*) initializes *n* disjoint subsets, each of which contains exactly one of the indices between 1 and *n*.
  - *p* = *find*(*i*) makes *p* point to the set containing index *i*.
  - *merge*(*p, q*) merges the two sets, to which *p* and *q* point, into the set.
  - *equal*(*p, q*) returns true if *p* and *q* both point to the same set.

```
void kruskal (int n, int m,
              edge_set E,
              edge_set& F)
{
    index i, j;
    set_pointer p, q;
    edge e;
    sort the m edges in E by weight in nondecreasing order;
    F = ø;
    initial(n);
    while (number of edges in F is less than n - 1){
        e = edge with least weight not yet consisdered;
        i, j = indices of vertices connected by e;
        p = find(i);
        q = find(j);
        if (!equal(p, q)){
            merge(p, q);
            add e to F;
        }
    }
}
```

Worst-case Time Complexity of Kruskal’s Algorithm

There are three considerations in this algorithm:

- The time to sort the edges: $\Theta(m \lg m)$.
- The time in the while loop.
  - In the worst case, every edge is considered before the while loop is exited, which means there are m passes through the loop.
  - The time complexity for m passes through a loop that contains a constant number of calls to routines *find*, *equal*, and *merge* is $\Theta(m \lg m)$ (try to implement functions for disjoint sets and prove this complexity).
- The time to initialize n disjoint sets: $\Theta(n)$.

Worst-case Time Complexity of Kruskal's Algorithm

- In the worst case, every vertex can be connected to every other vertex, which would mean that

$$m = \frac{n(n-1)}{2} \in \Theta(n^2).$$

- Therefore, the worst-case time complexity in terms of n is:

$$W(n) \in \Theta(n^2 \lg n^2) = \Theta(n^2 \lg n).$$

Optimality Proof for Kruskal’s Algorithm

- Almost same as the proof of Prim’s Algorithm.
  - Try to prove it by yourself (maybe appear in the exam).

Comparing Prim's Algorithm with Kruskal's Algorithm

- We obtained the following time complexities:
  - Prim's algorithm: $T(n) \in \Theta(n^2)$.
  - Kruskal's algorithm: $W(m) = \Theta(m \lg m)$ and $W(n) \in \Theta(n^2 \lg n)$.
- In a connected graph:

$$n - 1 \leq m \leq \frac{n(n-1)}{2}.$$

- Therefore, the conclusion is:
  - For a graph whose m is near the low end of these limits (the graph is very sparse), Kruskal's algorithm is faster with time complexity $\Theta(n \lg n)$.
  - For a graph whose m is near the high end (the graph is highly connected), Prim's algorithm is faster with time complexity $\Theta(n^2)$.

DIJKSTRA'S ALGORITHM FOR SINGLE-SOURCE SHORTEST PATHS

Single-Source Shortest Paths

- We developed a $\Theta(n^3)$ algorithm for determining the shortest paths between *each pair of vertices* in a weighted, directed graph by Floyd's algorithm with dynamic programming.
- If we want to know only the shortest paths from *one particular vertex to all the others* (called the Single-Source Shortest Paths problem), that algorithm would be overkill.
- We will use the greedy approach to develop a $\Theta(n^2)$ algorithm for this problem.
  - It is just like Prim's algorithm.

Dijkstra’s Algorithm

High level pseudocode:

- Initialize $F = \emptyset$ and $Y = \{v_1\}$.
- Iterate when the instance is not solved:
  - Select a vertex v in $V - Y$ that has a shortest path from v_1, using only vertices in Y as intermediates.
  - Add v to Y.
  - Add the edge that touches v to F.
  - Check whether $Y == V$.
    - Yes, the instance is solved.

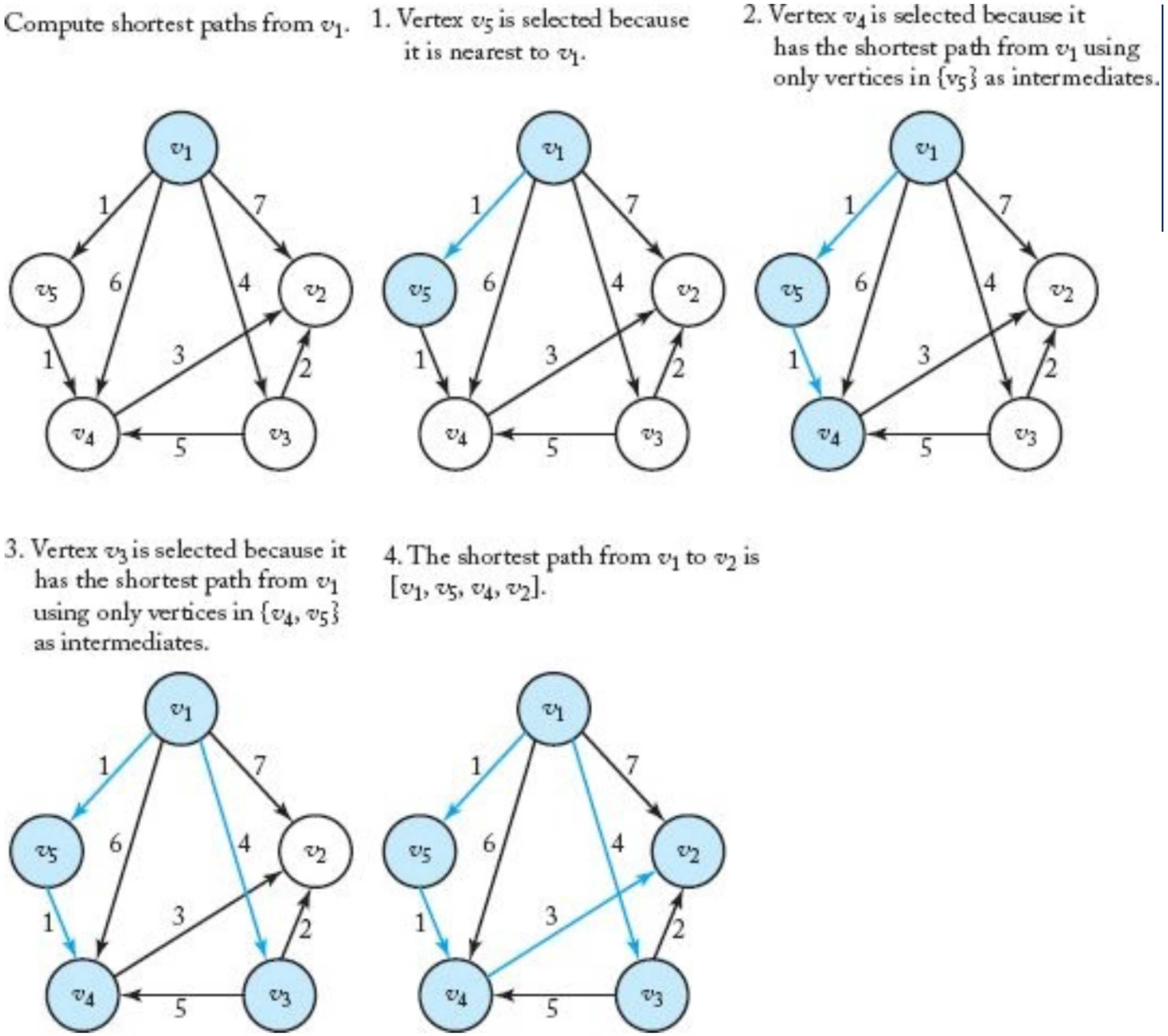


Image source: Figure 4.8, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Dijkstra's Algorithm

- Similar to Prim's algorithm, the every-case time complexity is:

$$T(n) = 2(n-1)^2 \in \Theta(n^2).$$

- Proof of Dijkstra's algorithm is also similar.

```
void dijkatra (int n,
               const number W[][],
               edge_set& F)
{
    index i;
    number min;
    edge e;
    index vnear; // index of selected vertex to be added
    index touch[2...n]; // index of vertex in Y such that the edge <v, v_i> is
                        // the last edge on the current shortest path from v_1
                        // to v_i using only vertices in Y as intermediates.
    number length[2...n]; // length of the current shortest path from v_1 to v_i
                          // using only vertices in Y as intermediates.

    F = ∅;
    for (i = 2; i <= n; i++){
        touch[i] = 1;
        length[i] = W[1][i];
    }

    repeat (n - 1 times){
        min = inf;
        for (i = 2; i <= n; i++)
            if (0 <= length[i] < min){
                min = length[i];
                vnear = i;
            }
        e = edge from v_touch[vnear] to v_vnear;
        add e to F;
        for (i = 2; i <= n; i++)
            if (length[vnear] + W[vnear][i] < length[i]){
                length[i] = length[vnear] + W[vnear][i];
                touch[i] = vnear;
            }
        length[vnear] = -1;
    }
}
```

SCHEDULING

Scheduling Problem

- Suppose a hair stylist has several customers waiting for different treatments.
  - E.g. massage, simple cut, wash+cut+style, permanent, hair dye...
- The treatments don’t all take the same amount of time, but the stylist knows how long each takes.
- A reasonable goal would be to schedule the customers in such a way as to *minimize the total time they spend* both waiting and being served.
- The problem of minimizing the total time in the system has many applications.
  - For example, we may want to schedule users’ access to a disk drive to minimize the total time they spend waiting and being served.

Scheduling Problem

- Suppose there are three jobs and the service times for these jobs are

$$t_1 = 5, \qquad t_2 = 10, \qquad t_3 = 4$$

- If we schedule them in the order 1, 2, 3, the times spent in the system for the three jobs are as follows:
  - Job 1: 5 (service time).
  - Job 2: 5 (wait for job 1) + 10 (service time).
  - Job 3: 5 (wait for job 1) + 10 (wait for job 2) + 4 (service time).
- The total time in the system for this schedule is

$$\underbrace{5}_{\text{Time for job 1}} + \underbrace{(5+10)}_{\text{Time for job 2}} + \underbrace{(5+10+4)}_{\text{Time for job 3}} = 39$$

Scheduling Problem

- This same method of computation yields the following list of all possible schedules and total times in the system:

Schedule	Total Time in the System
[1, 2, 3]	5 + (5 + 10) + (5 + 10 + 4) = 39
[1, 3, 2]	5 + (5 + 4) + (5 + 4 + 10) = 33
[2, 1, 3]	10 + (10 + 5) + (10 + 5 + 4) = 44
[2, 3, 1]	10 + (10 + 4) + (10 + 4 + 5) = 43
[3, 1, 2]	4 + (4 + 5) + (4 + 5 + 10) = 32
[3, 2, 1]	4 + (4 + 10) + (4 + 10 + 5) = 37

- Schedule [3, 1, 2] is optimal with a total time of 32.

Scheduling Problem

- The algorithm is straightforward (even without a name):
  - Sort the jobs by service time in nondecreasing order.
  - Iterate when the instance is not solved.
    - Schedule the next job.
    - Check whether there are no more jobs.
      - Yes, the instance is solved.
- The worst-case time complexity is mainly on the sorting part: $W(n) \in \Theta(n \lg n)$.

Optimality Proof of the Algorithm for Scheduling Problem

Theorem 2

The only schedule that minimizes the total time in the system is one that schedules jobs in nondecreasing order by service time.

Proof:

- We show this using proof by contradiction.
- Let t_i be the service time for the ith job scheduled in some particular optimal schedule.
- If they are not scheduled in nondecreasing order, then for at least one i where $1 \leq i \leq n-1$,

$$t_i > t_{i+1}.$$

Optimality Proof of the Algorithm for Scheduling Problem

Proof (cont'd):

- We can rearrange our original schedule by swapping the ith and $(i+1)$st jobs with total time T':

$$T' = T + t_{i+1} - t_i < T,$$

because $t_i > t_{i+1}$, which contradicts the optimality of our original schedule.

Multiple-Server Scheduling Problem

- It is straightforward to generalize our algorithm to handle the Multiple-Server Scheduling problem with m servers.
  - Order the jobs again by service time in nondecreasing order.
  - Let the first server serve the first job, the second server the second job, ... , and the mth server the mth job.
  - The first server will finish first because that server serves the job with the shortest service time.
  - Then, the first server serves the $(m+1)$st job. Similarly, the second server serves the $(m+2)$nd job, and so on.

Multiple-Server Scheduling Problem

- The scheme is as follows:
  - Server 1 serves jobs $1, 1+m, 1+2m, 1+3m, \ldots$
  - Server 2 serves jobs $1, 2+m, 2+2m, 3+3m, \ldots$
  - …
  - Server i serves jobs $i, i+m, i+2m, i+3m, \ldots$
  - …
  - Server m serves jobs $m, 2m, 3m, 4m, \ldots$
- Clearly, the jobs end up being processed in the following order:
  - $1, 2, \ldots, m, 1+m, 2+m, \ldots, 2m, 1+2m, \ldots$

HUFFMAN CODE

Data Compression by Binary Code

- Given a data file, it would be desirable to find a way to store the file as efficiently as possible.
- The problem of data compression is to find an efficient method for encoding a data file.
- A common way to represent a file is to use a *binary code*.
- In such a code, each character is represented by a unique binary string, called the *codeword*.
- A *fixed-length binary code* represents each character using the same number of bits.
  - For example, suppose our character set is {a, b, c}.
  - Then we could use 2 bits to code each character: a: 00, b: 01, c: 11.
  - Given this code, if our file is ababcbbbc, our encoding will be 000100011101010111.

Data Compression by Binary Code

- We can obtain a more efficient coding using a *variable-length binary code*.
  - Such a code can represent different characters using different numbers of bits.
- In the previous example:
  - We can code one of the characters as 0.
  - Since ‘b’ occurs most frequently, it would be most efficient to code ‘b’ as 0.
  - However, then ‘a’ could not be coded as ‘00’ because we would not be able to distinguish one ‘a’ from two ‘b’s.
  - Furthermore, we would not want to code ‘a’ as ‘01’ because when we encountered a 0, we could not determine if it represented a ‘b’ or the beginning of an ‘a’.
  - So we could code by: a: 10, b: 0, c: 11.
  - This file would be encoded as: 1001001100011.

Optimal Binary Code Problem

- This second coding method takes 13 bits to represent that is better than the first one with 18 bits.
- Given a file, the optimal binary code problem is to find a binary character code for the characters in the file, which represents the file in the least number of bits.

Prefix Codes

- One particular type of variable-length code is a *prefix code*.
- In a prefix code no codeword for one character constitutes the beginning of the codeword for another character.
- For example, if 01 is the code word for '*a*', then 011 could not be the codeword for '*b*'.
- The advantage of a prefix code is that there is no ambiguity when interpreting the codes.
- Every prefix code can be represented by a binary tree whose leaves are the characters that are to be encoded.

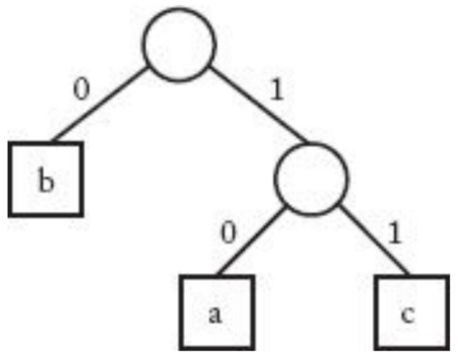


Image source: Figure 4.9, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Prefix Codes

- To parse, we start at the first bit on the left in the file and the root of the tree.
- We sequence through the bits, and go left or right down the tree depending on whether a 0 or 1 is encountered.
- When we reach a leaf, we obtain the character at that leaf; then we return to the root and repeat the procedure starting with the next bit in sequence.
- Try to parse the tree: 1001001100011 -> ababcbbbc.

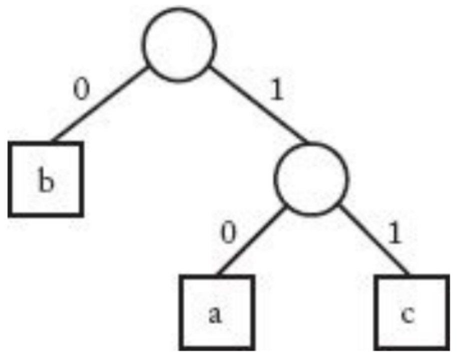


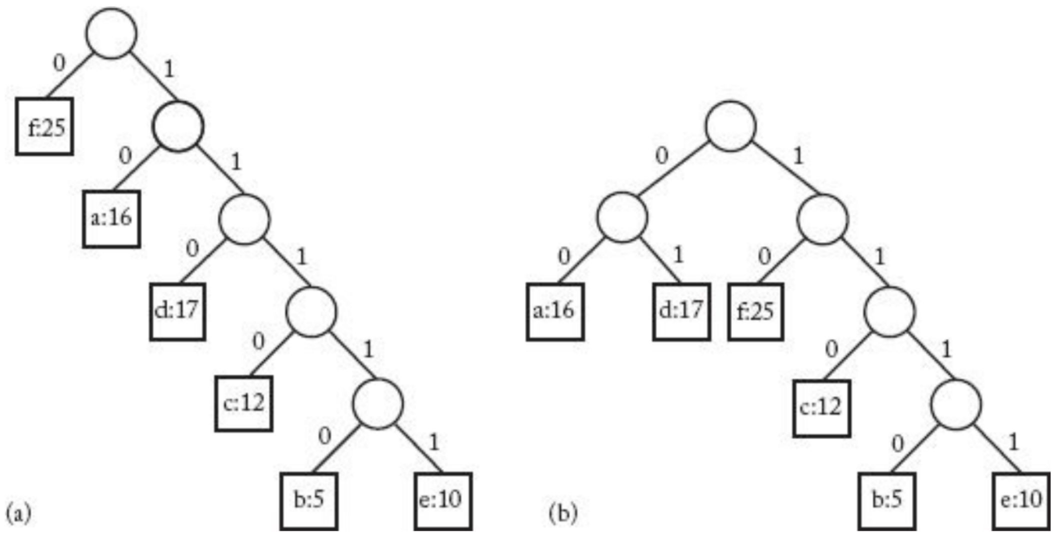
Image source: Figure 4.9, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Example of Optimal Binary Code Problem

- That table also shows three different codes we could use to encode the file with character set {a, b, c, d, e, f} .
- The number of bits for each encoding:
  - Bits(C1)=16(3)+5(3)+12(3)+17(3)+10(3)+25(3)=255.
  - Bits(C2)=16(2)+5(5)+12(4)+17(3)+10(5)+25(1)=231.
  - Bits(C3)=16(2)+5(4)+12(3)+17(2)+10(4)+25(2)=212.

Character	Frequecy	C1 (Fixed-Length)	C2 (Variable-Length)	C3 (Huffman)
a	16	000	10	00
b	5	001	11110	1110
c	12	010	1110	110
d	17	011	110	01
e	10	100	11111	1111
f	25	101	0	10

Example of Optimal Binary Code Problem



C2 C3

Character	Frequecy	C1 (Fixed-Length)	C2 (Variable-Length)	C3 (Huffman)
a	16	000	10	00
b	5	001	11110	1110
c	12	010	1110	110
d	17	011	110	01
e	10	100	11111	1111
f	25	101	0	10

Image source: Figure 4.10, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Optimal Binary Code Problem

- As can be seen from the preceding example, the number of bits it takes to encode a file given the binary tree *T* corresponding to some code is given by

$$bits(T) = \sum_{i=1}^{n} frequency(v_i)depth(v_i)$$

- It is similar to the optimal binary search tree problem, but with no constraint that the tree should be a search tree ($Key_{left\_child} \leq Key_{node} \leq Key_{right\_child}$).

Huffman's Algorithm

- We need to use a *priority queue*.
- In a priority queue, the element with the highest priority is always removed (dequeue) next.
  - In this case, the element with the highest priority is the character with the lowest frequency in the file.
- A priority queue can be implemented as a linked list, but more efficiently as a heap.

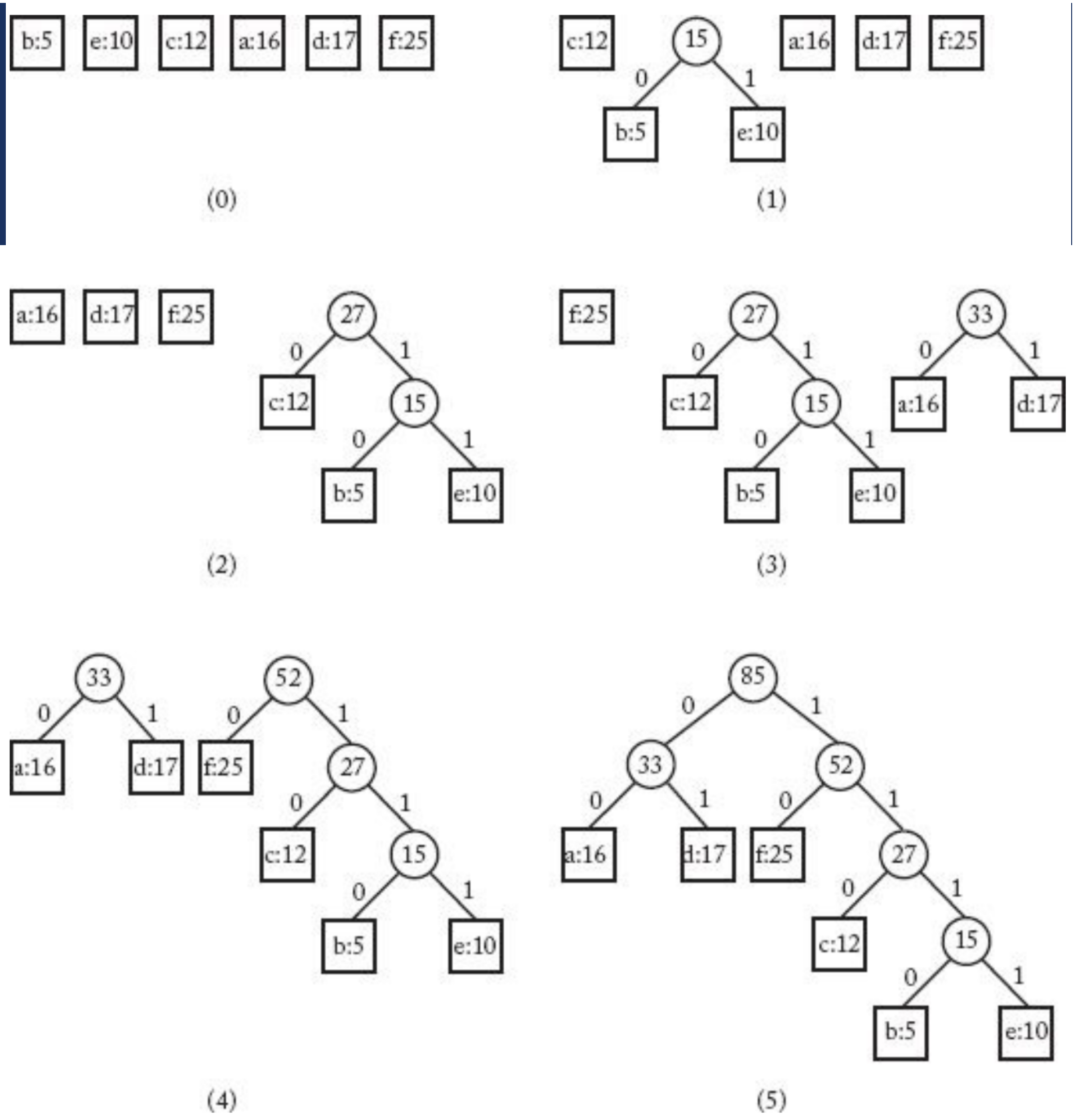


Image source: Figure 4.11, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Huffman’s Algorithm

```
struct nodetype
{
    char symbol;
    int frequency;

    nodetype* left;
    nodetype* right;
};
typedef nodetype* node_pointer;
```

```
nodetype huffman(int n,
                 const char symbol[],
                 const number frequency[])
{
    priority_queue pq;
    node_pointer p, q;

    for (i = 1; i <= n; i++){
        r = new nodetype;
        r->symbol = symbol[i];
        r->frequency = frequency[i];
        r->left = r->right = NULL;
        enqueue(pq, r);
    }

    for (i = 1; i <= n - 1; i++){
        p = dequeue(pq);
        q = dequeue(pq);
        r = new nodetype;
        r->left = p;
        r->right = q;
        r->frequency = p->frequency + q->frequency;
        enqueue(pq, r);
    }
    return dequeue(pq);
}
```

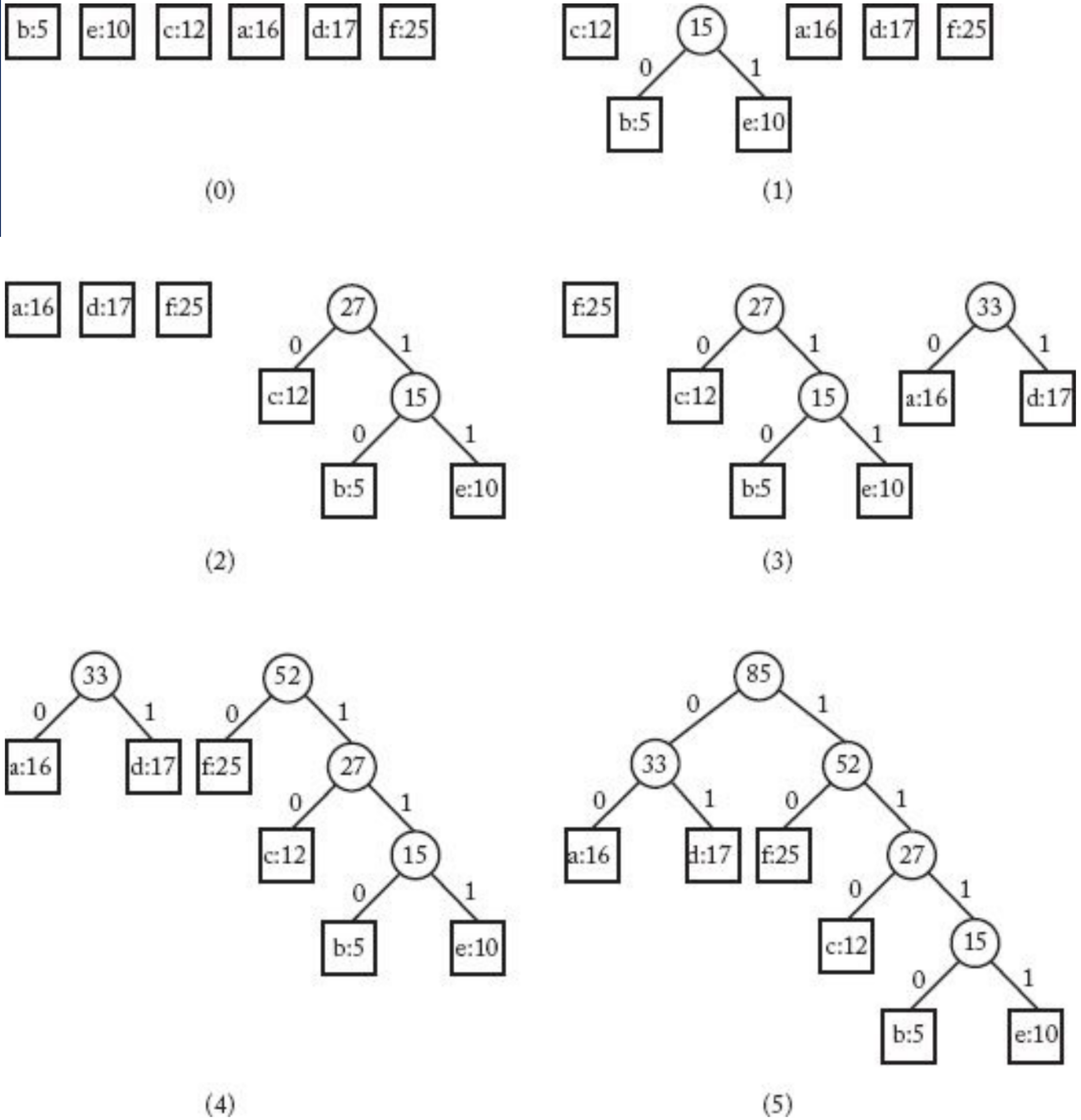


Image source: Figure 4.11, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Huffman's Algorithm

- If a priority queue is implemented as a heap, it can be initialized in $\Theta(n)$ time.
- Furthermore, dequeue and enqueue in heap requires $\Theta(\lg n)$ time.
- Since there are $n-1$ passes through the for loop, the algorithm runs in $\Theta(n \lg n)$ time.

Optimality Proof Huffman’s Algorithm

- Before the optimality proof huffman’s algorithm, we have the following definitions
  - Two nodes are called *siblings* in a tree if they have the same parent.
  - A *branch* with root v in tree T is the subtree whose root is v.

Optimality Proof Huffman's Algorithm

Theorem 3

Huffman's algorithm produces an optimal binary code.

Proof:

- The proof is by induction.
- Assuming the set of trees obtained in the ith step are branches in an optimal binary tree, we show that the set of trees obtained in the $(i + 1)$st step are also branches in an optimal binary tree.

Optimality Proof Huffman's Algorithm

Proof (cont'd):

- Basis step:
  - Clearly, the set of single nodes obtained in the 0th step are branches in a binary tree corresponding to an optimal code.
- Induction step:
  - Assume the set of trees obtained in the ith step are branches in some optimal binary tree T.
  - Let u and v be the roots of the trees to be combined in the $(i+1)$st step of Huffman's algorithm.
  - If u and v are siblings in T, then we are done because the set of trees obtained in the $(i+1)$st step of Huffman's algorithm are branches in T.

Optimality Proof Huffman's Algorithm

Proof (cont'd):

- Otherwise, without loss of generality, assume u is at a level in T at least as low as v.
- Because we construct a branch with two children at each step, u must have some sibling w in T.
- Since the tree with root v is chosen by Huffman's algorithm in this step

$$frequency(v) \leq frequency(w).$$

- And in T

$$depth(v) \leq depth(w).$$

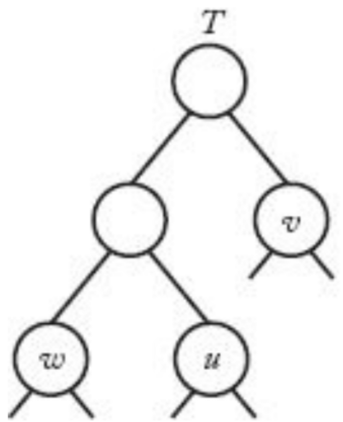
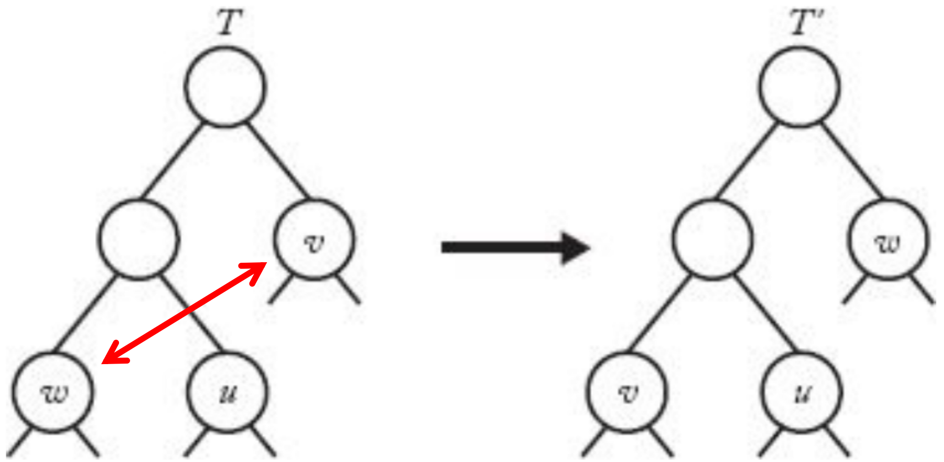


Image source: Figure 4.12, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Optimality Proof Huffman's Algorithm



Proof (cont'd):

- We can create a new binary tree T' by swapping the positions of the branches rooted at v and w in T such that

$$bits(T') = bits(T) + [depth(w) - depth(v)][frequency(v) - frequency(w)] \le bits(T).$$

- It means that T' is also optimal. Otherwise it contradicts the fact that T is optimal.
- Clearly, the set of trees obtained in the $(i + 1)$st step of Huffman's algorithm are branches in some optimal binary tree.

Image source: Figure 4.12, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

THE KNAPSACK PROBLEM

Knapsack Problem Recall

- Problem description:
  - Given n items and a "knapsack.”
  - Item i has weight $w_i > 0$ and has value $v_i > 0$.
  - Knapsack has capacity of W.
  - Goal: Fill knapsack so as to maximize total value.
- Mathematical description:
  - Given two n-tuples of positive numbers $< v_1, v_2, \dots, v_n >$ and $< w_1, w_2, \dots, w_n >$, and $W > 0$, we wish to determine the subset $T \subseteq \{1,2,\dots,n\}$ that

$$\text{maximize} \sum_{i \in T} v_i \qquad\qquad \text{subject to} \sum_{i \in T} w_i \leq W$$

- Can greedy approach obtain optimal solution?

Example

- Weight capacity $W = 5\text{kg}$.
- The possible ways to fill the knapsack:
  - {1, 2, 3} has value $37 with weight 4kg.
  - {3, 4} has value $35 with weight 5kg. (greedy)
  - {1, 2, 4} has value $42 with weight 5kg. (optimal)
- The greedy approach by always selecting the item with highest value is not optimal.

i	v_i	w_i
1	$10	1kg
2	$12	1kg
3	$15	2kg
4	$20	3kg

The Fractional Knapsack Problem

- The previous problem is also called the 0-1 knapsack problem.
  - Each item can only be taken or not taken as a whole.
- Now, we change the problem to enable one to take any fraction of the item.
  - Both weight and value follow the fraction.
  - This is called the fractional knapsack problem.
  - A greedy approach can be developed by always choosing the item with the largest value-weight ratio.

The Fractional Knapsack Problem

- Weight capacity $W = 5\text{kg}$.
- By the greedy approach:
  - Take item 2: remain 4kg and total value is 12.
  - Take item 1: remain 3kg and total value is 22.
  - Take item 3: remain 1kg and total value is 37.
  - Take 1/3 of item 4: remain 0kg and total value is 43.67.
- It is optimal. Try to prove it.

i	v_i	w_i	v_i/w_i
1	\$10	1kg	10\$/kg
2	\$12	1kg	12\$/kg
3	\$15	2kg	7.5\$/kg
4	\$20	3kg	6.67\$/kg

Dynamic Programming vs the Greedy Approach

- In common: find optimal solution for subinstance of the problem.
- Difference:
  - The greedy approach: any optimal solution for subinstance is a part of the final optimal solution.
  - Dynamic programming: only a subset of optimal solution for subinstances construct the final optimal solution.
- Different approaches are used for similar problems with only little difference.
  - Shortest path problem vs single-source shortest path problem.
  - Optimal binary search tree vs optimal binary code.
  - 0-1 knapsack problem vs fractional knapsack problem.
- Analyzing the problem is really important when desinging an algorithm.

Conclusion

After this lecture, you should know:

- What is greedy approach.
- How to design a greedy approach.
- How to prove if a problem can be solved by a greedy approach.
  - Induction with contradiction.
- What is the difference between dynamic programming and the greedy approach.

Assignment 3

- Assignment 3 is released. The deadline is **18:00, 1st June**.

Thank you!

- Any question?
- Don't hesitate to send email to me for asking questions and discussion. ☺