Name: Patharapon Buathong Student ID: 6207050401 2

M07 Greedy Methods

1. Objectives / Outcome

1) Students will be able to recognize the concept of greedy methods to solve the problem, especially minimum spanning tree, single-source shortest paths, and coding problems.
2) Students will be able to design and implement the greedy methods for problem-solving, especially Dijkstra's algorithm.

2. Related Contents

Computer scientists consider it a general design technique even though it is applicable to optimization problems only. The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step—and this is the central point of this technique—the choice made must be:

- **Feasible**: it must satisfy the problem's constraints.
- **Locally optimal (with respect to some neighborhood definition)**: it must be the best local choice among all feasible choices available on that step.
- **Greedy (in terms of some measure), and Irrevocable:** it cannot be changed on subsequent steps of the algorithm.

These requirements explain the technique's name: on each step, it suggests a "greedy" grab of the best alternative available in the hope that a sequence of locally optimal choices will yield a (globally) optimal solution to the entire problem. As a rule, greedy algorithms are both intuitively appealing and simple. Given an optimization problem, it is usually easy to figure out how to proceed in a greedy manner, possibly after considering a few small instances of the problem. The application of the greedy strategy are as follows:

Name: Patfaropon Buathong Student ID: 6207050401 2

2.1. Problem: Minimum spanning tree

A **spanning tree** of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a **minimum spanning tree** is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges. *The **minimum spanning tree problem** is the problem of finding a minimum spanning tree for a given weighted connected graph.*

Note that It has direct applications to the design of all kinds of networks— including communication, computer, transportation, and electrical—by providing the cheapest way to achieve connectivity. It identifies clusters of points in data sets. It has been used for classification purposes in archeology, biology, sociology, and other sciences. It is also helpful for constructing approximate solutions to more difficult problems such the traveling salesman problem.

2.1.1. Prim's algorithm

Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees. The pseudocode of Prim's algorithm is

ALGORITHM *Prim*(G)

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
    such that v is in V_T and u is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

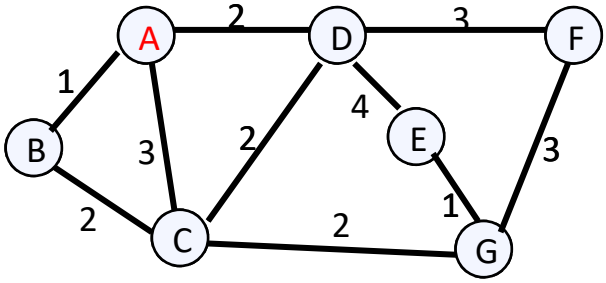
The summarized meaning is

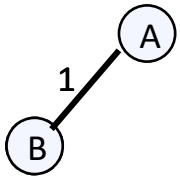
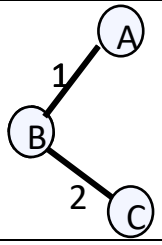
1. Add arbitrary vertex (node) from G in T.
2. Grow T by adding a vertex from G closest to those already in T with the corresponding edge.
3. Repeat 2. until all the vertices in G are added in T.

Name: Patharapon Buathong Student ID 6207050401 2

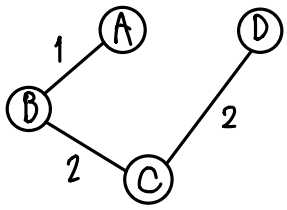
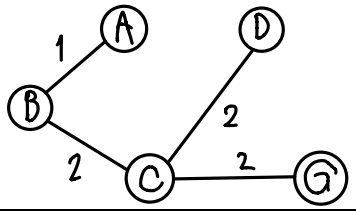
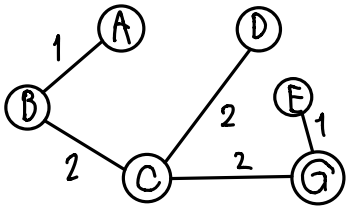
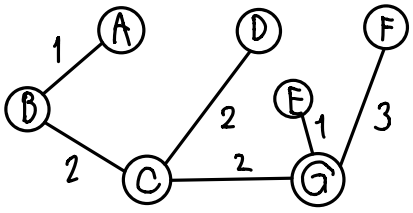
4. The algorithm does not necessarily give a unique MST.

The nature of Prim's algorithm makes it necessary to provide each vertex not in the current tree with the information about the shortest edge connecting the vertex to a tree vertex. For example, the application of Prim's algorithm is the parenthesized labels of a vertex in the middle column indicate the nearest tree vertex and edge weight, as follows:



Tree vertices	Remaining Vertices	Illustration
A(-,-)	B(A, 1) C(A, 3) D(A, 2) E(-, ∞) F(-, ∞) G(-, ∞)	
B(A, 1)	C(B,2) D(-, ∞) E(-, ∞) F(-, ∞) G(-, ∞)	
C(B ,2)	A(C,3) D(C,2) G(C,2) E(-, ∞) F(-, ∞)	

Name: Patharapon Buathong Student ID 6207050401 2

D(C,2)	A(D,2), E(D,4), F(D,3) F(-,∞)	
~~E~~ G(C,2)	E(G,1), F(G,3)	
~~D~~ E(G,1)	D(E,4), F(-,∞)	
~~F~~ F(G,3)		

∴Cost of MST
= Σ edge weight
= 1+2+2+2+1+3
= 11 //

We can also implement the priority queue as a min-heap. Namely, a min-heap is a complete binary tree in which every element is less than or equal to its children. All the principal properties of heaps remain valid for min-heaps, with some obvious modifications. In term of algorithm's efficiency: **O(n^2)** for weight matrix representation of graph and array implementation of priority queue. **O(m log n)** for adjacency lists representation of graph with n vertices and m edges and min-heap implementation of the priority queue.

2.1.2. Kruskal's algorithm

Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph G = {V, E} as an acyclic subgraph with |V| – 1 edges for which the sum of the edge weights is the smallest. The pseudocode of Kruskal's algorithm is

Name: Pattaraporn Busthong Student ID: 6207050401 2

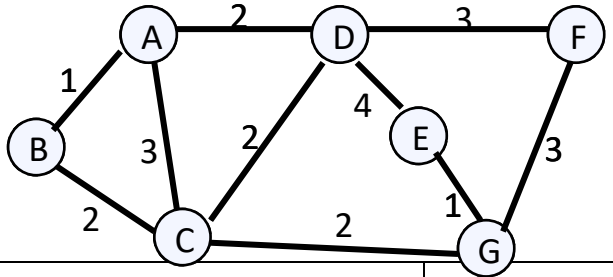
ALGORITHM *Kruskal(G)*

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \le \cdots \le w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
while $ecounter < |V| - 1$ **do**
  $k \leftarrow k + 1$
  if $E_T \cup \{e_{i_k}\}$ is acyclic
    $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $ecounter \leftarrow ecounter + 1$
return E_T

The summarized meaning is

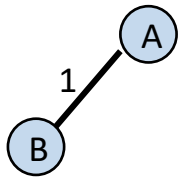
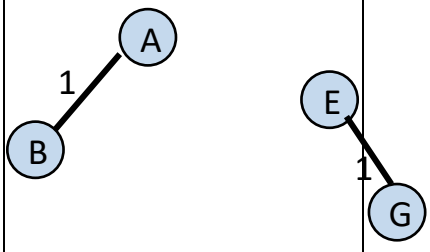
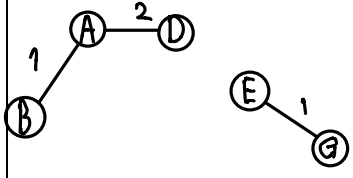
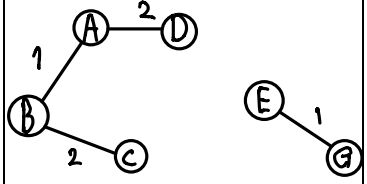
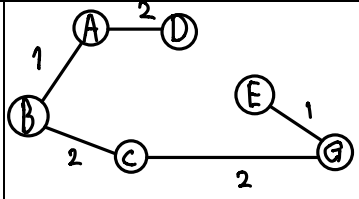
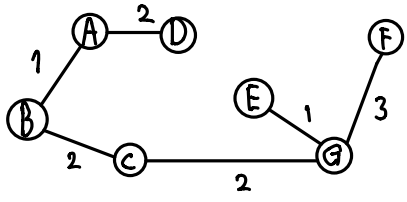
1. Sort the edges in the ascending order. T = **ϕ**
2. Select an edge with smallest weight and add to T.
3. Select the edge (u, v) with smallest weight such that T is acyclic if added.
4. Add the edge in Step 3 to T.
5. Repeat step 3 until all the nodes are included in T.

For example, the application of Kruskal's algorithm is shown below. As you trace the algorithm's operations, note the disconnectedness of some of the intermediate subgraphs, as follows:



Tree edges	Sorted lists of edges	Illustration
	BA EG AD BC CG CD AC GF DE 1 1 2 2 2 2 3 3 4	

Name: Patharapon Buathong Student ID: 6207050401 2

BA	BA EG AD BC CG CD AC DF GF DE 1 1 2 2 2 2 3 3 3 4	
EG	EG AD BC CG CD AC DF GF DE 1 2 2 2 2 3 3 3 4	
AD	AD BC CG CD AC DF GF DE 2 2 2 2 3 3 3 4	
BC	BC CG CD AC DF GF DE 2 2 2 3 3 3 4	
CG	CG CD AC DF GF DE 2 2 3 3 3 4	
GF	CD AC DF GF DE 2 3 3 3 4	

Name: Pattaraporn Busthong Student ID 6207050401 2

Note that this algorithm looks easier than Prim's but is harder to implement (checking for cycles!). The cycle checking is a cycle is created iff added edge connects vertices in the same connected component. Fortunately, there are efficient algorithms for doing so, including the crucial check for whether two vertices belong to the same tree. They are called **union-find algorithms**. Hence, with an efficient sorting algorithm, the time efficiency of Kruskal's algorithm will be in **O(m log m)** time, with m = |E| or **O(|E| log |E|).**

2.2. Problem: Single-Source Shortest Paths Problem

Single Source Shortest Paths Problem: Given a weighted connected (directed) graph G, find shortest paths from source vertex s to each of the other vertices The single-source shortest-paths problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may, of course, have edges in common.

2.2.1. Dijkstra's algorithm

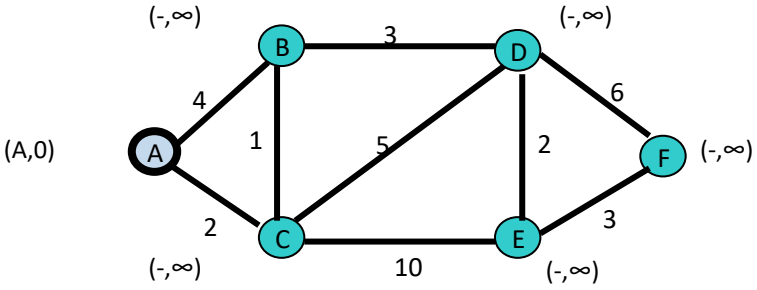
Similar to Prim's MST algorithm, with a different way of computing numerical labels: Among vertices not already in the tree, it finds vertex u with the smallest sum $\mathbf{d_v + w(v,u)}$. where v is a vertex for which shortest path has been already found on preceding iterations (such vertices form a tree rooted at s), d_v is the length of the shortest path from source s to v, and w(v,u) is the length (weight) of edge from v to u. The pseudocode of Dijkstra's algorithm is

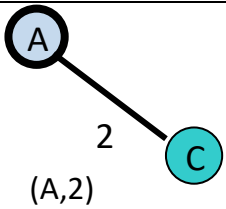
Name: Patfaropon Buathong Student ID 6207050401 2

ALGORITHM *Dijkstra*(G, s)

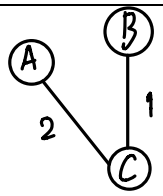
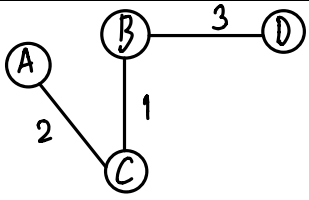
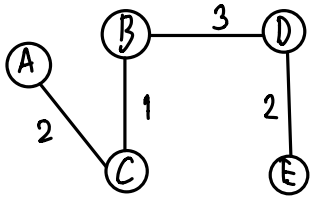
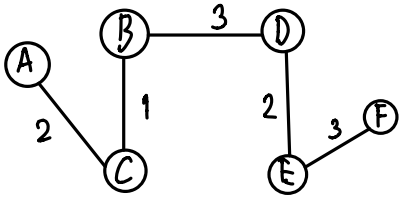
```
//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph G = ⟨V, E⟩ with nonnegative weights
//        and its vertex s
//Output: The length d_v of a shortest path from s to v
//         and its penultimate vertex p_v for every vertex v in V
Initialize(Q)  //initialize priority queue to empty
for every vertex v in V
    d_v ← ∞;   p_v ← null
    Insert(Q, v, d_v)  //initialize vertex priority in the priority queue
d_s ← 0;  Decrease(Q, s, d_s)  //update priority of s with d_s
V_T ← ∅
for i ← 0 to |V| − 1 do
    u* ← DeleteMin(Q)  //delete the minimum priority element
    V_T ← V_T ∪ {u*}
    for every vertex u in V − V_T that is adjacent to u* do
        if d_u* + w(u*, u) < d_u
            d_u ← d_u* + w(u*, u);   p_u ← u*
            Decrease(Q, u, d_u)
```

For example, the application of Dijkstra's algorithm is



Tree vertices	Remaining Vertices	Illustration
A(-,0)	C(A,2) B(A,4) D(-,∞) E (-,∞)F (-,∞)	(A,0) 
C(A,2)	B(C,3) D(C,7) E(C,12) F (-,∞)	

Name: Patfaropon Buathong Student ID: 6207050401 2

B(C,1) B(A,3)	B(D,6), E(-,∞), F(-,∞)	
D(B,3) D(A,6)	E(D,8), F(D,14)	
E(D,2) E(A,8)	F(E,3)	
F(E,3) F(A,11)	-	

Note that, a variety of practical applications of the shortest-paths problem have made the problem a very popular object of study. The obvious but probably most widely used applications are transportation planning and packet routing in communication networks, including the Internet. Multitudes of less obvious applications include finding shortest paths in social networks, speech recognition, document formatting, robotics, compilers, and airline crew scheduling. In the world of entertainment, one can mention pathfinding in video games and finding best solutions to puzzles using their state-space graphs. Efficiency: $O(|V|^2)$ for graphs represented by weight matrix and array

Name: Pattaraporn Buathong Student ID: 6207050401 2

implementation of priority queue, and **O(|E| log |V|)** for graphs represented by adj. lists and min-heap implementation of priority queue.

2.3. Problem: Coding Problem

Coding is an assignment of bit strings to alphabet characters. While, Code-words is a bit strings assigned for characters of alphabet. Two types of codes include 1) **fixed-length encoding** that assigns to each symbol a bit string of the same length. This is exactly what the standard ASCII code does. 2) **Variable-length encoding**, which assigns code-words of different lengths to different symbols, introduces a problem that fixed-length encoding does not have. **The problem is how can we tell how many bits of an encoded text represent the first (or, more generally, the i^{th}) symbol?** To avoid this complication, we can limit ourselves to the so-called prefix-free (or simply prefix) codes. The one with the shortest average code length. The average code length represents on the average how many bits are required to transmit or store a character.

2.3.1. Huffman tree

Any binary tree with edges labeled with 0's and 1's yields a prefix-free code of characters assigned to its leaves. We can do better by using Prefix-free codes like Huffman Code. Optimal binary tree minimizing the average length of a code-word can be constructed as follows:

<u>Huffman's algorithm</u>

Step 1 Initialize **n** one-node trees and label them with the symbols of the alphabet given. Record the frequency of each symbol in its tree's root to indicate the tree's weight. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

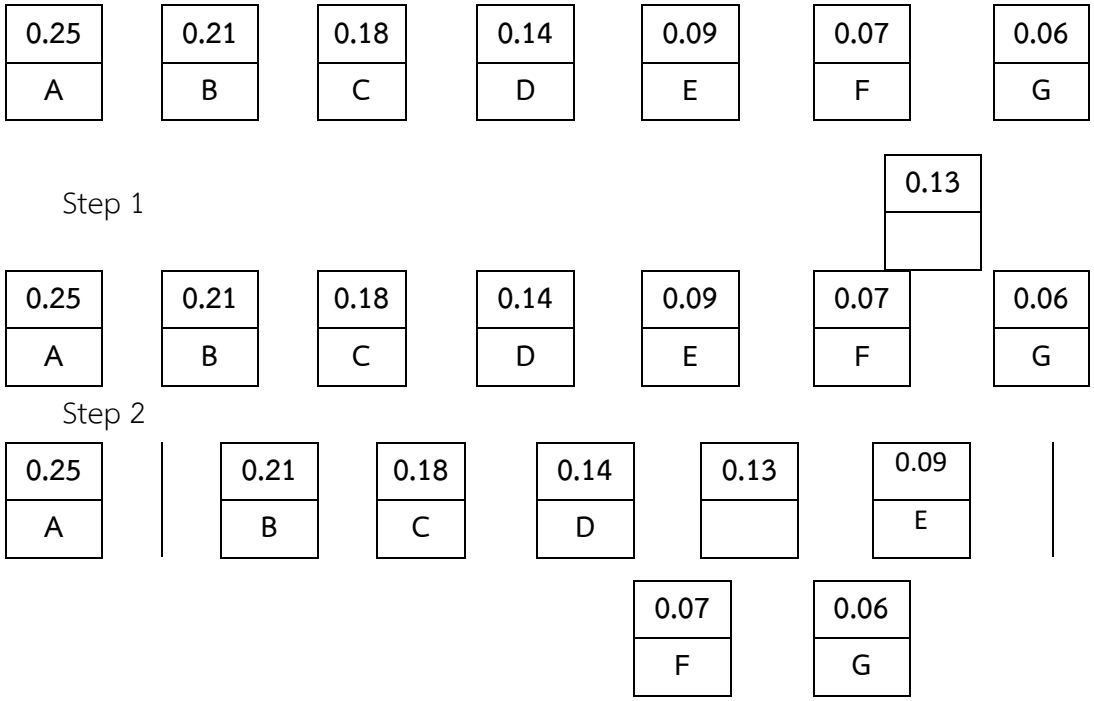
Step 2 Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight (ties can be broken arbitrarily, but see Problem 2 in this section's exercises). Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

Name: Pattaraporn Buathong Student ID: 6207050401 2

A tree constructed by the above algorithm is called a **Huffman tree**. It defines—in the manner described above—a **Huffman code**.

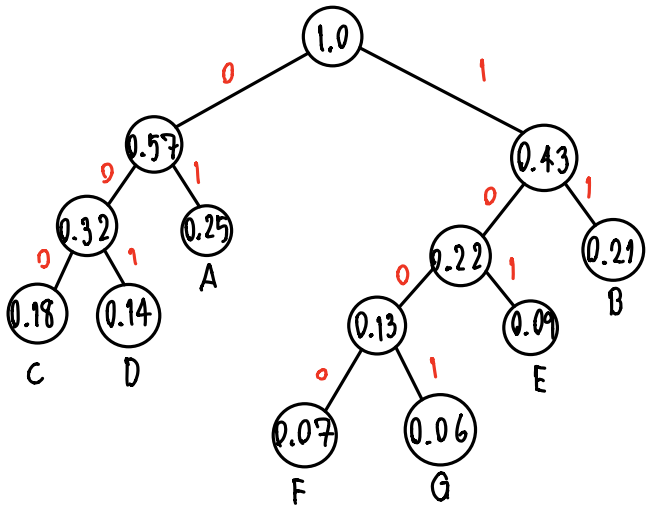
For example, Consider the five-symbol alphabet {A, B, C, D, _} with the following occurrence frequencies in a text made up of these symbols:

symbol	A	B	C	D	E	F	G
frequency	0.25	0.21	0.18	0.14	0.09	0.07	0.06



Left traversal = 0
Right traversal = 1

The result codewords are as follows:



Name: Pattaraporn Buathong Student ID 6207050401 2

symbol	A	B	C	D	E	F	G
Frequency	0.25	0.21	0.18	0.14	0.09	0.07	0.06
codeword	10	11	000	001	101	1000	1001
Total bit	0.5	0.42	0.54	0.42	0.27	0.28	0.24 = 2.67

With the occurrence frequencies given and the codeword lengths obtained, the average number of bits per symbol in this code is $\sum_{i=1}^{n} l_i * p_i$, where li is the length of the path from the root to the ith leaf, indicates the average number of questions needed to "guess" the chosen number with a game strategy represented by its decision tree. Then, the result is 2.67/7 = 0.38.

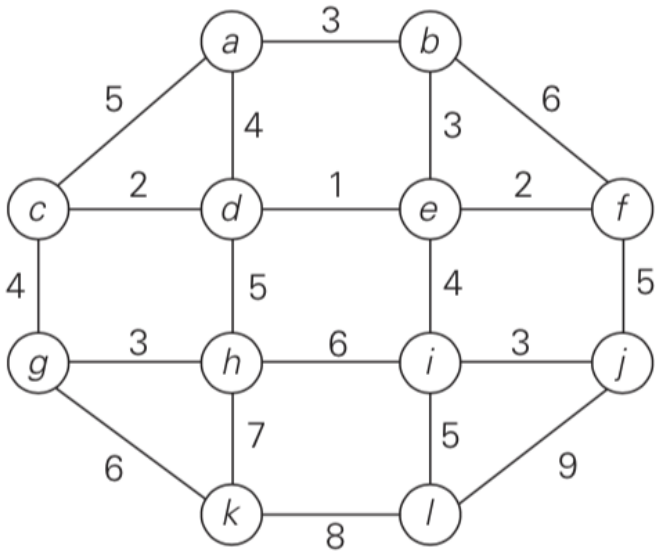
Had we used a fixed length encoding for the same alphabet, we would have to use at least 3 bits per each symbol. The **compression ratio**, which is a standard measure of a compression algorithm's Effectiveness, is $\frac{8}{2.67}$*100% = 299.63%. In other words, Huffman's encoding of the text will use 66.62% less memory than its fixed-length encoding.

Compression

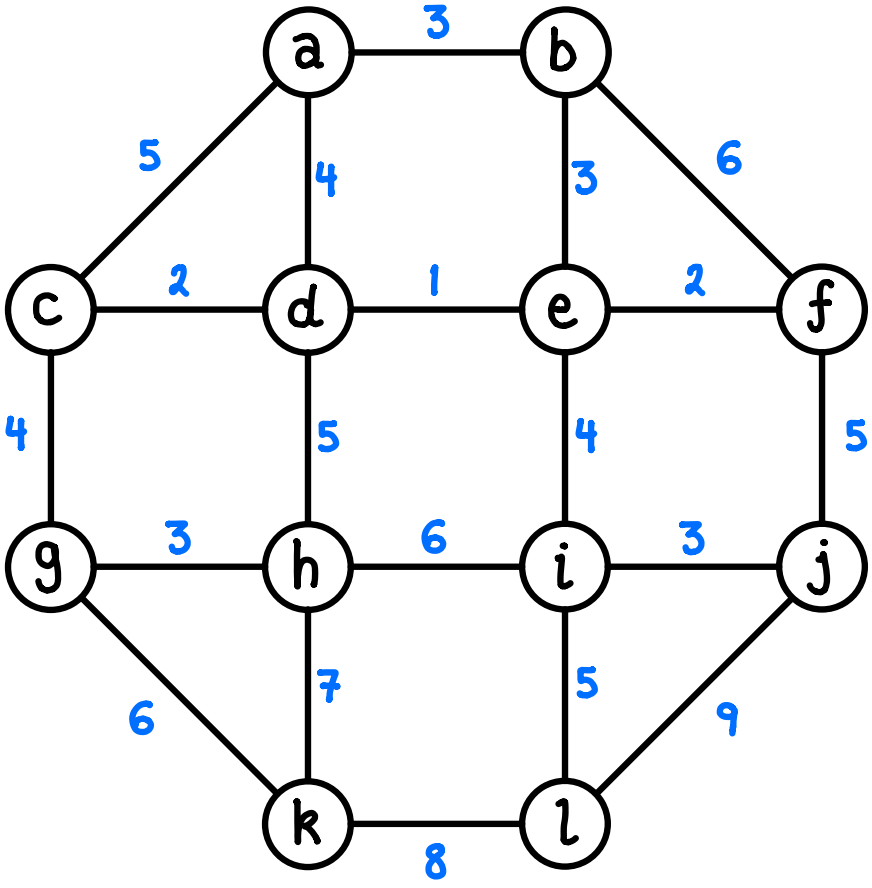
$= \frac{8}{2.67} \times 100$

3. **Exercises (Group Works)** Group : Samkok

3.1. Apply **Kruskal's algorithm** to find a minimum spanning tree of the following graph:



Apply Kruskal's algorithm to find a minimum spanning tree of the following graph :

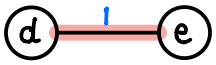
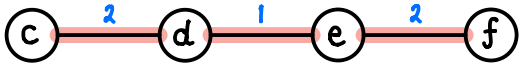
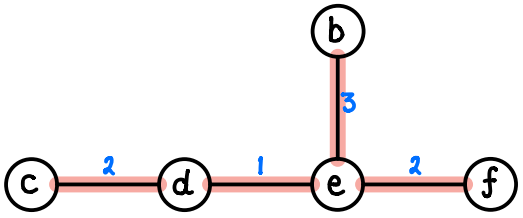
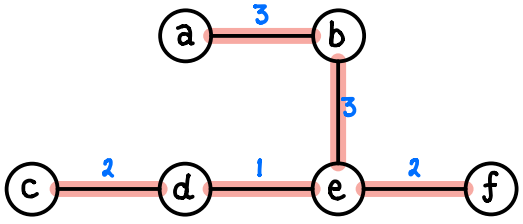
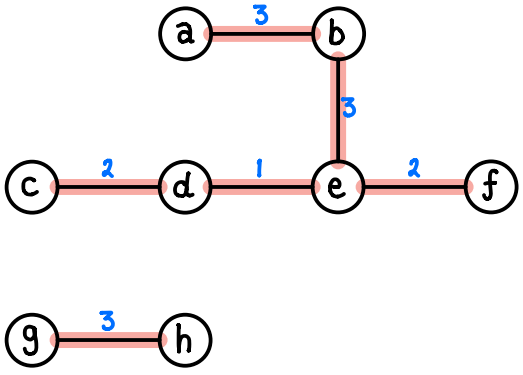


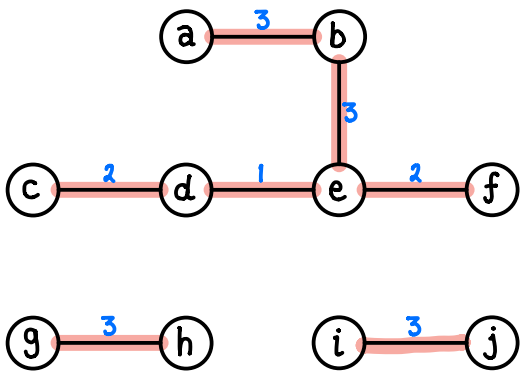
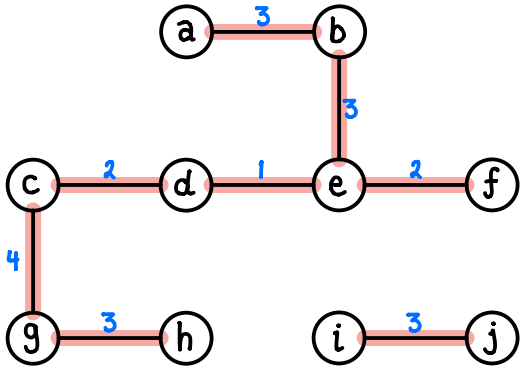
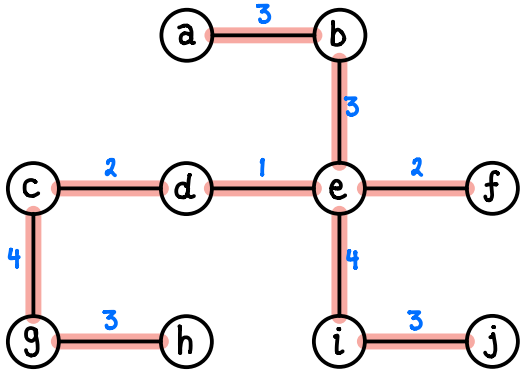
Step 1: Remove all loops and Parallel Edges

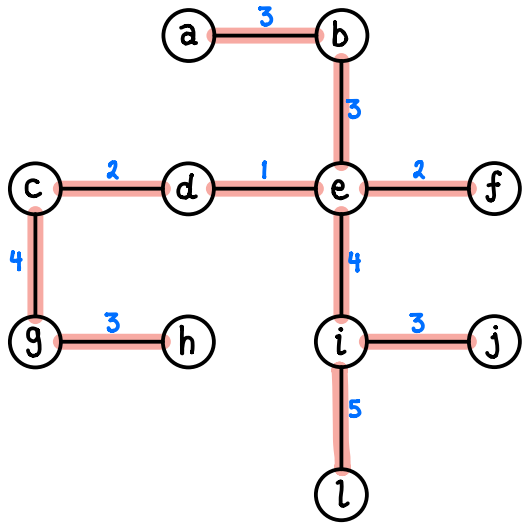
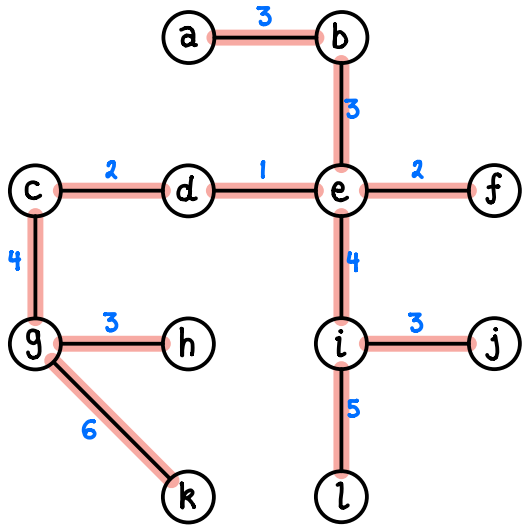
Step 2: Arrange all edges in their increasing order of weight

Step 3: Add the edge which has the least weightage

↳ In the process, ignore all edges that create a circuit

Edge	Sorted list of edges	Illustration
de	de dc ef eb ab gh ij gc 1 2 2 3 3 3 3 4 ei il gk 4 5 6	
dc	dc ef eb ab gh ij gc ei 2 2 3 3 3 3 4 4 il gk 5 6	
ef	ef eb ab gh ij gc ei il 2 3 3 3 3 4 4 5 gk 6	
eb	eb ab gh ij gc ei il gk 3 3 3 3 4 4 5 6	
ab	ab gh ij gc ei il gk 3 3 3 4 4 5 6	
gh	gh ij gc ei il gk 3 3 4 4 5 6	

ij	ij gc ei il gk 3 4 4 5 6	
gc	gc ei il gk 4 4 5 6	
ei	ei il gk 4 5 6	

il	il gk 5 6	
gk	gk 6	

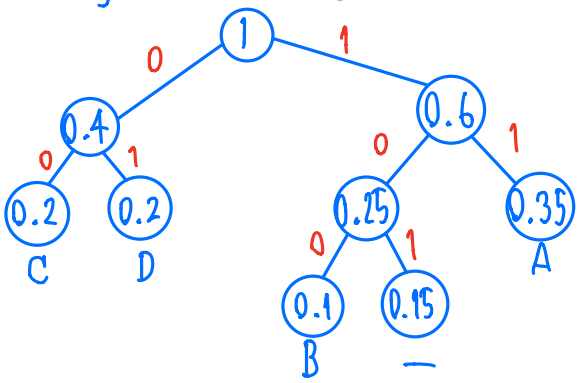
Name: Patsaraporn Buathong Student ID 6207050401 2

3.2. From the following symbol frequencies

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

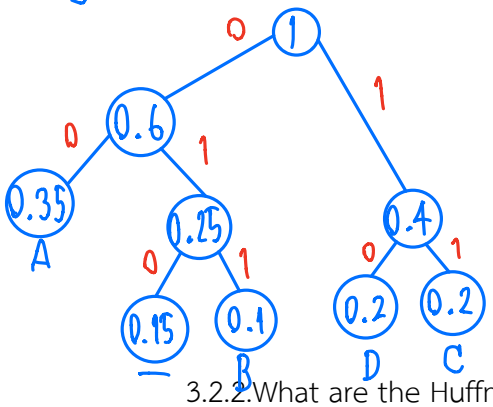
3.2.1.Create Huffman trees and the resulting code-words when the weights are sorted ascendingly and descending.

Ascending B, _, C, D, A



symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
Codeword	11	100	00	01	101

Descending A, C, D, _, B



symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	00	011	11	10	010

3.2.2.What are the Huffman codes for the input BAD_AD?

Ascending : 100111011 1101

Descending : 0110010 0100010

Name: Pattaraporn Buathong Student ID 6207050401 2

4. LAB (Ass05) Assignments

4.1. Implement the Dijkstra's algorithm that finds a single-source shortest path tree. The program performs the following steps:

1. Accept the number of vertices n between 2 and 10 from the user.
2. Generate a (symmetric) cost matrix of a random graph with n vertices where the probability of having an undirected edge from vertex i to vertex j is 0.4 and the cost is randomly selected from the range 1 to 10. The cost between two vertices with no edge should be set to -1 and the cost to itself should be set to 0.
3. Accept the source vertex s between 1 and n from the user.
4. Run the Dijkstra's algorithm to find the shortest path from s to all the other vertices and print out the result.

In Step 2, to determine if there should be an undirected edge from vertex i to j, you can generate a random number from the range 0 and 1. If the random number is less than 0.4, a directed edge is assigned.

The sample run is

```
Enter the number of vertices (2 to 10):5
Generate a cost matrix of an unidirected random graph ..
[[ 0  3  4  3  7]
 [ 3  0  1  1  7]
 [ 4 -1  0  2  1]
 [ 3 -1  2  0 10]
 [ 7  7 -1 10  0]]


Enter the source node(1 to 10):3
Shortest path from  3  to every node ..

1 <- 2 <- 3

2 <- 3
```

5. Reference

Levitin, A. (2012) Introduction to the Design & Analysis of Algorithms (3rd edition). Pearson. (Chapter 9)