Q1- Knapsack problem

The problem involves n objects, $b_0, b_1, \ldots, b_{n-1}$ and a knapsack of capacity C. Each object b_i has a given positive weight or volume w_i and a given positive value v_i (profit, cost, etc) $i = 0, \ldots, n-1$. If a fraction f_i of object b_i is placed in the knapsack, then it contributes f_i to the total value of in the knapsack.

The goal is to place objects or fractions of objects in the knapsack without exceeding capacity, so that the total value off the objects in the knapsack is maximized.

Formal Statement:

$$\begin{gathered} \text{maximize} \quad \sum_{i=0}^{n-1} f_i v_i, \\ \text{subject to the constraints:} \quad \sum_{i=0}^{n-1} f_i w_i \leq C, \\ 0 \leq f_i \leq 1, \quad i = 0, \ldots, n-1. \end{gathered} \qquad (6.3.1)\ \text{p. } 248$$

\

Greedy Strategy: At each iteration, putting objects or fractions of objects into the knapsack according to decreasing rations (densities) v_i/w_i until the knapsack is full yields the optimal solution. Thus, the ratio v_i/w_i is the critical issue. Id.

Exercise 6.5, p. 281

Object Type b_i	**Quantity Available** w_i	**Total Cost for that Quantity** v_i
0	30	60
1	100	50
2	10	40
3	10	30
4	8	20
5	8	10
6	1	5
7	1	1

Figure 1.1 - Object b_i Inventory

Object Type **b_i**	**Ratio (Density)** **v_i/w_i**	**Fraction of Available Quantity Chosen** **f_i**	**Quantity Chosen** **f_iw_i**
6	5	1	1
2	4	1	10
3	3	1	10
4	2.5	1	8
0	2	1/30	1
5	1.25	0	0
7	1	0	0
1	0.5	0	0

Figure 1.2 - Solution for capacity C = 30.
Objects sorted in decreasing order of ratios v_i/w_i, i = 0,…, n-1

In Figure 1.1, we list the objects and amounts of objects that are available. Figure 1.2 sorts *the 8* objects, $b_0, b_1,\ldots, b_7$ in decreasing order of density v_i/w_i, i = 0,…, n-1. The capacity C of the knapsack is 30. Implementing the greedy strategy, we obtain the most valuable (maximized) knapsack (without exceeding capacity) if we first use as much as possible of the object *bi* whose density is the largest, then as much as possible of the object whose density is second-largest, and so on until we we reach 30.

Here, we place in decreasing order the total quantity available of objects b_6, b_2, b_3, b_4 because they have the highest density, 5, 4, 3, 2.5, respectively. At this point, C = 29, so we have 30-29 = 1 space remaining in the knapsack. The object with the highest density after b_4 is b_2 with a density of 2 and available quantity w_i = 30. We only have one space remaining in the knapsack, so the fraction of available quality chosen f_0 = 1/30. Since w_0 = 30, f_0*w_0 = 1. The total value of the knapsack equals (1*5) + (10*4) +(10*3) + (8*2.5) + (1*2) = 5 + 40 + 30 + 20 + 2 = 107, and satisfies capacity constraints (1*1) +(1*10) + (1*10) + (1*8) + [(1/30) * 30] = 30.
//
//

Q2 - Huffman Codes

The Huffman algorithm is a data compression algorithm based on a greedy strategy of for constructing *optimal binary prefix codes* for a given alphabet of symbols *A*. Binary prefix codes have the property that no binary string in the codes is a prefix of any other string in the code. Optimal binary prefix code are binary prefix codes that minimize the expected length of text generated using the symbols starting from *A*. The problem of finding an optimal binary prefix code is equivalent to finding a 2-tree *T* having minimum *weighted leaf path length WLPT(T) of T* defined by,

$WLPL(T) = \sum_{i=0}^{n-1} \lambda_i f_i.$	(6.4.1) p,256

where λ_i denotes the length of the path in *T* from the root to the leaf node corresponding to *ai*, $i = 0, \ldots,$ n-*1.*

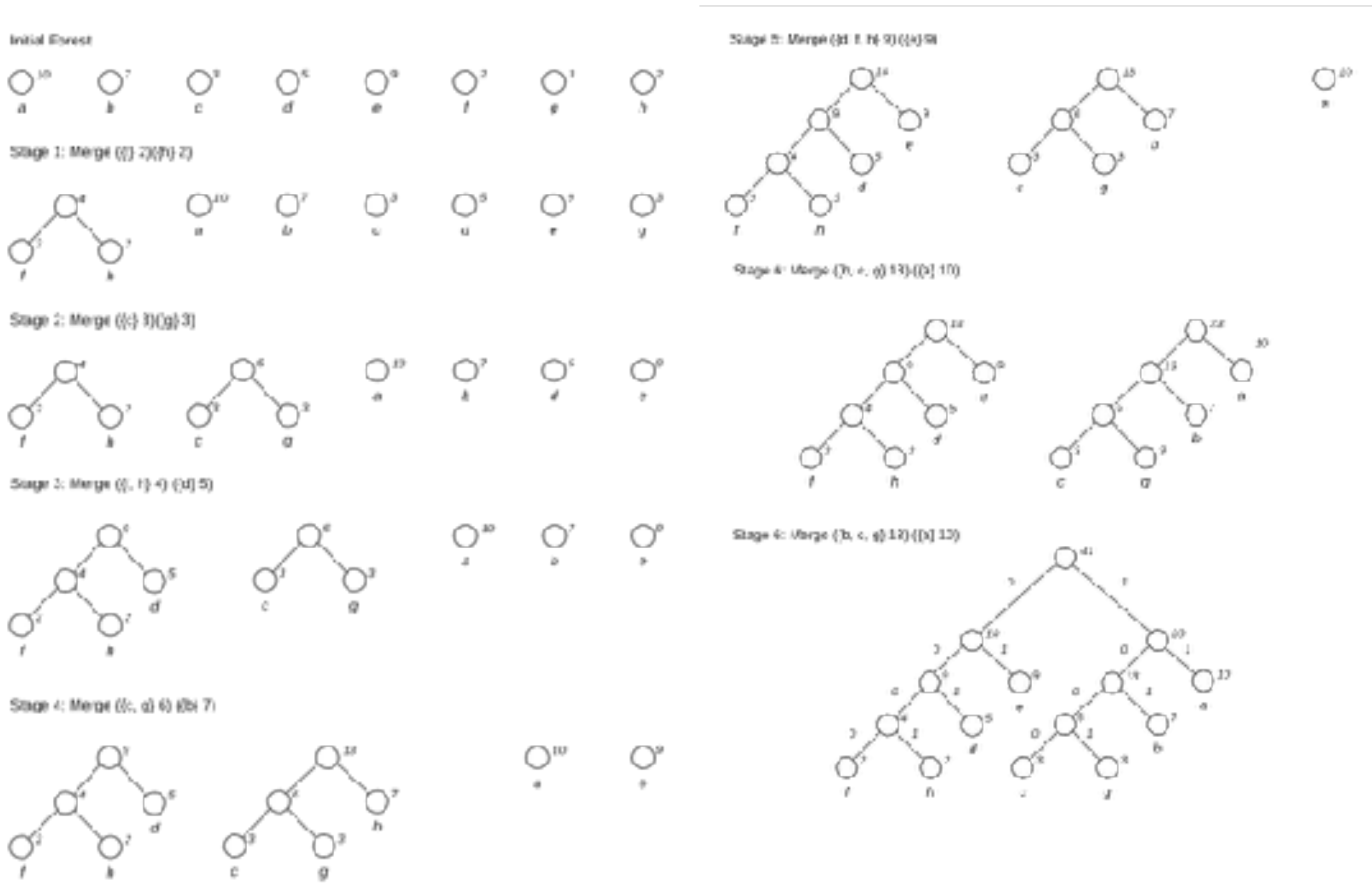
The goal of the Huffman Algorithm is to generate optimal binary prefix codes with respect to a given set frequencies f_i, $i = 0, \ldots, n-1$ for each symbol starting from *A,* thus minimizing the expected length of a coded symbol. The algorithm computes a 2-Tree *T* minimizing the *WLPL(T)* by constructing a sequence of forests. At each iteration, the root of each tree in the forest is assigned a frequency. The initial forest *F* consists of the *n* single node trees corresponding to the *n* elements of *A*. **At each stage iteration, the algorithm makes a greedy choice: it finds two trees T_1 and T_2 whose roots R_1 and R_2 have the smallest and second-smallest frequencies over all tree in the current forest.** A new internal node *R* is then added to the forest, together with two edges joining *R* to R_1 and *R* to R_2, so that a new tree is created with *R* as the root and T_1 and T_2 as the left and right subtrees of *R*. The frequency off the new root vertex *R* is taken to be the sum of the frequencies of the old root vertices R_1 and R_2. The Huffman tree is constructed after *n*-1 stages, involving the addition of *n-1* internal nodes.

//

//

//

//



Symbol	Frequency	Encoding
a	10	11
b	7	101
c	3	1000
d	5	001
e	9	01
f	2	0000
g	3	1001
h	2	0001

$WLPL(T)$ = (2)(10)+(3)(7)+(4)(3)+(3)(5)+(2)(9)+(4)(2)+(4)(3)+(4)(2) = 134

Figure 2 - Action of *HuffmanCode* for the alphabet
***A* = *{a,b,c,d,e,f,g,h}* with frequencies 10,7,3,5,9,2,3,2, respectively.**

Q3 - Minimum Spanning Tree

A spanning tree of *G* is a subset of the edges that connects all the vertices and has no cycles. A minimum spanning tree is a spanning tree that has the lowest possible weight, viz, the sum of the weights of the edges must be as low as possible.

6.14 Consider the following weighted graph *G*

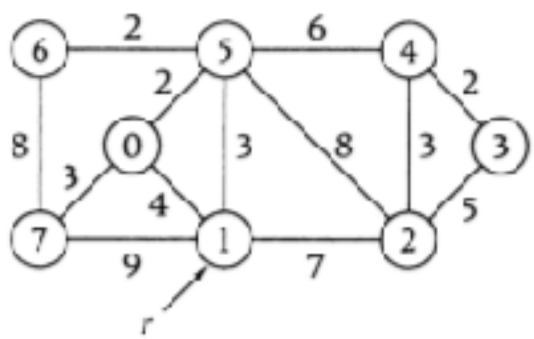


Figure 3.1 - The weighted, connected graph *G* given in Exercise 6.14, p. 283

a) Trace the action procedure *Kruskal* for *G*

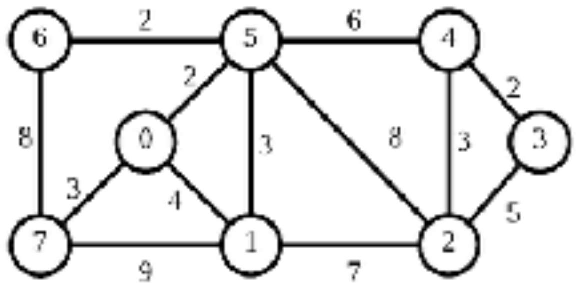
Kruskal's assumes that G is connected, and starts with a forest of *n* isolated trees consisting of a single node. **The algorithm uses a greedy strategy to grow a minimum spanning tree by adding to it the lightest edge that connects two trees on each iteration.**

The algorithm either sorts all of the edges by weight in advance and process them in order, or uses a min-heap priority queue. Next, the disjoint-set data structure is used to test whether the edge connects two components, rejecting those edges if a cycle is formed. If an edge is found, the *Kruskal's* joins the components using an intermixed sequence of the the disjoint-set ADT's *Union* and *Find* operations.

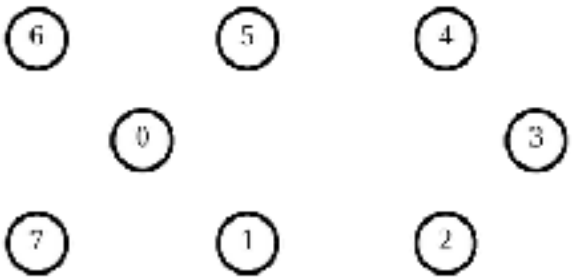
In Figure 3.2, we sort the edges of graph *G* in Figure 3.1 in increasing order of weight or cost. If connecting the two components creates a cycle, the edge is rejected. If a cycle is not formed, the edge is added to the tree. Finally, the weight of the tree is given at each stage or interaction of the process until all vertices are connected in the minimum spanning tree. *Kruskal*'s algorithm is illustrated for the graph in Exercise 6.14 in Figure 3.3.

Stage or Iteration	Edge	Edge Weight	Add or Reject? (cycle formed?)	Tree Weight
Initial	0	0	N/A	0
1	(0,5)	2	Add	2
2	(3,4)	2	Add	4
3	(5,6)	2	Add	6
4	(1,5)	3	Add	9
5	(2,4)	3	Add	12
6	(0,7)	3	Add	15
7	(0,1)	4	Reject	15
8	(2,3)	5	Reject	15
9	(5,4)	6	Add	21

Figure 3.1 - Action of procedure *Kruskal* for graph *G* given in Figure 3.1

$G =$ 

Stage 0

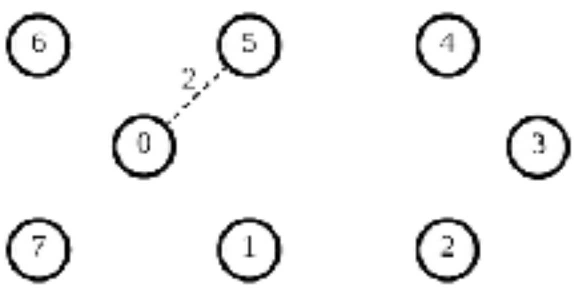


Forest ADT of trees composed of single nodes / vertrices

Disjoint-set ADT
e = [**0**],[**1**],[**2**],[**3**],[**4**],[**5**],[**6**],[**7**]

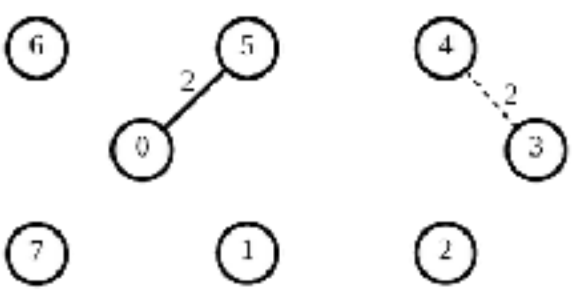
At the initial stage, each set within the disjoint-set ADT consists of a single marked node / vertex (marked vertices in **bold face**.)

Stage 1



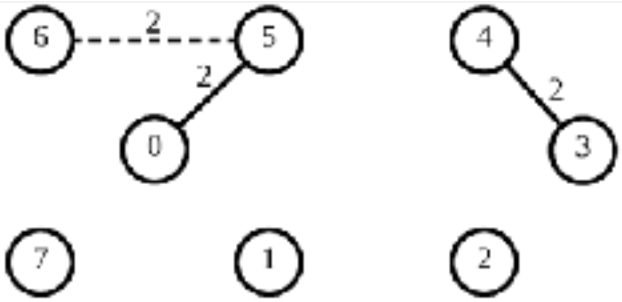
- *Find*(0)=0 ≠ *Find*(5)=5
- Add [0,5] to forest.
- Take *Union* of sets having marked elements [0,5]
  e = [0,5],[1],[2],[3],[4],[6],[7]

Stage 2



- *Find*(3)=3 ≠ *Find*(4)=4
- Add [3,4] to forest.
- Take *Union* of sets having marked elements [0,1]
  e = [**0**,5],[**1**][**2**],[**3**,4],[**6**],[**7**]

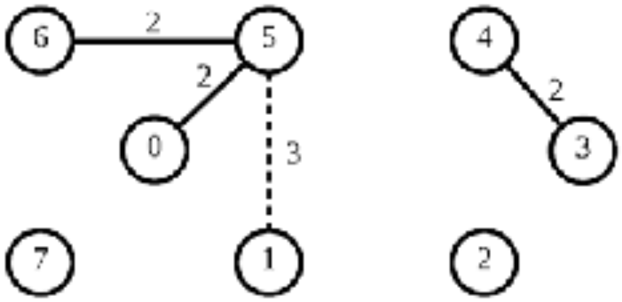
Stage 3



- *Find*(5)=0 ≠ *Find*(6)=6
- Add [5,6] to forest.
- Take *Union* of sets having marked elements [0,6]

  e = [**0**,5,6],[**1**][**2**],[**3**,4],[**7**]

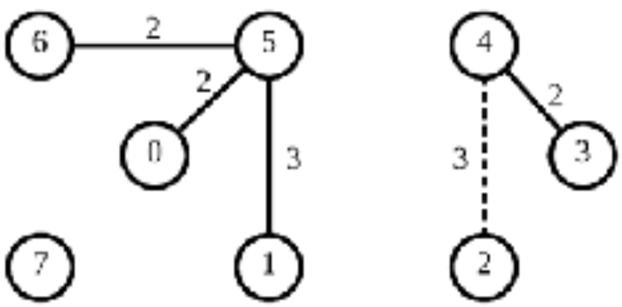
Stage 4



- *Find*(1)=1 ≠ *Find*(5)=0
- Add [**1**,5] to forest.
- Take *Union* of sets having marked elements [1,5]

e = [**0**,1,5,6],[**2**],[**3**,4],[**7**]

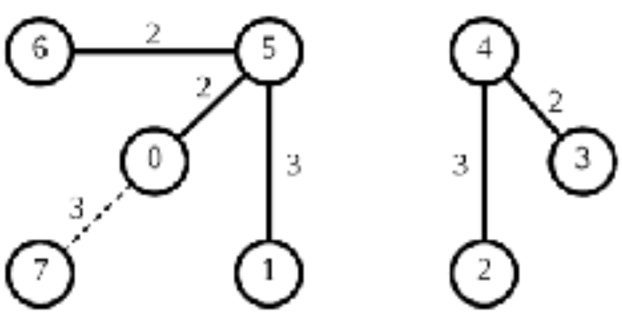
Stage 5



- *Find*(2)=2 ≠ *Find*(4)=3
- Add [2,4] to forest.
- Take *Union* of sets having marked elements [2,3]

e = [**0**,**1**,5,6],[**3**,2,**4**],[**7**]

Stage 6



- *Find*(0)=0 ≠ *Find*(7)=7
- Add [0,7] to forest.
- Take *Union* of sets having marked elements [0,7]

e = [**0**,1,5,6,7],[**3**,2,4]

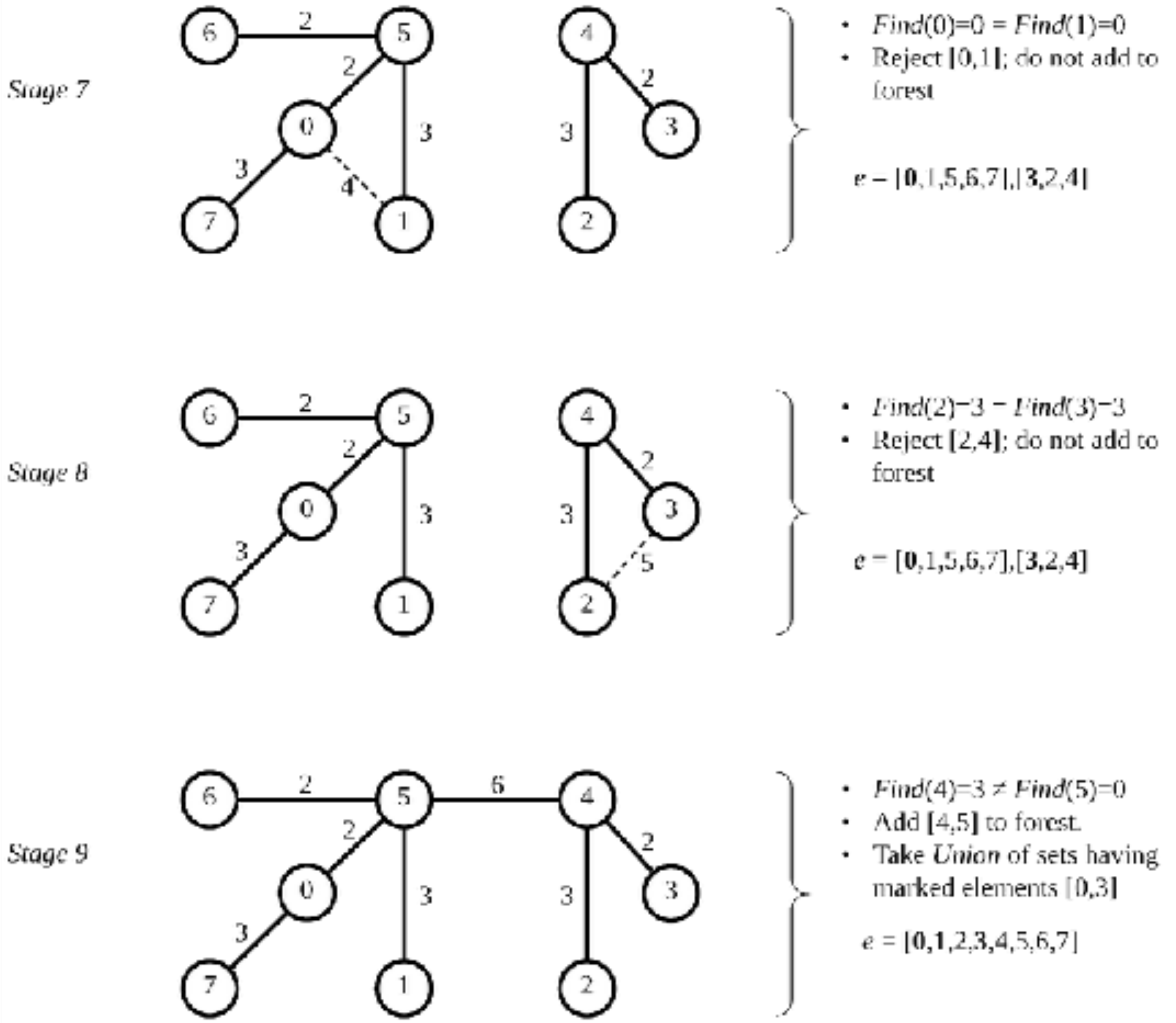


Figure 3.3 - Stages in *Kruskal's* Algorithm

b) Trace the action of procedure *Prim* for *G* with $r = 1$.

Unlike *Kruskal's* Algorithm, *Prim's* Algorithm maintains a tree at each stage instead of a forest. The initial tree T_i may be taken to be any single vertex *r* of the graph *G.* At any point, the graph *G* is composed of a single tree and a bunch of isolated vertices. The algorithm builds a sequence of *n* trees rooted at *r* $T_0, T_1, \ldots, T_{n-1}$, where T_{i+1} is obtained from T_i by adding a single edge e_i+1, $i = 0, \ldots, n - 2$. At every iteration, ***Prim's* makes a greedy choice and adds the smallest possible edge among all edges having exactly one vertex in T_i that connects the tree to an isolated vertex.**

The set of all edges in *G* having exactly one vertex in T_i is denoted by $Cut(T_i)$. A cut (*S, V-S*) of *G,* partitions the set of vertices *V(G)* into the sets *S* and *V-S.* The edges in *Cut(Ti)* contains only those edges which cross the cut. At each step, *Prim's* algorithm greedily chooses the lightest edges among all the edges in *Cut(Ti),* viz, the smallest edges which cross the cut. Thus, we can restrict our attention to the edges in *Cut(Ti).* Choosing the light edge at each iteration is always a safe edge, viz, an edge that maintains the minimum spanning tree at every step.

In Figure 3.4, we show the action of *Prim's* algorithm for graph *G* with $r = 1$ at each iteration. The initial stage shows the current tree, T_0 consisting of the single vertex $r = 1$ and the set of edges $Cut(T_i)$ available for growing the tree T_i. Subsequent stages also show the edge chosen, individual edge weight and current weight of the tree until all vertices are connected in the minimum spanning tree. *Prim*'s algorithm is illustrated for the graph in Exercise 6.14 in Figure 3.5.

Stage or Iteration	Edge e_i	T_i	$Cut(T_i)$	Edge Weight	Tree Weight
0 (*r = 1*)	∅	T_0	[1,0],[1,2],[1,5],[1,7]	0	0
1	(1,5)	T_1	[1,0],[1,2],[1,7],[5,0],[5,1],[5,4],[5,6]	3	3
2	(5,0)	T_2	[0,7],[1,2],[1,7],[5,0],[5,2],[5,4],[5,6]	2	5
3	(5,6)	T_3	[0,7],[1,2],[1,7],[5,2],[5,4],[6,7]	2	7
4	(0,7)	T_4	[1,2],[5,2],[5,4]	3	10
5	(5,4)	T_5	[1,2],[4,2],[4,3],[5,2]	6	16
6	(4,3)	T_6	[1,2],[4,2],[5,2]	2	18
7	(4,2)	T_7	∅	3	21

Figure 3.4 - Action of procedure *Prim* for graph *G* given in Figure 3.1

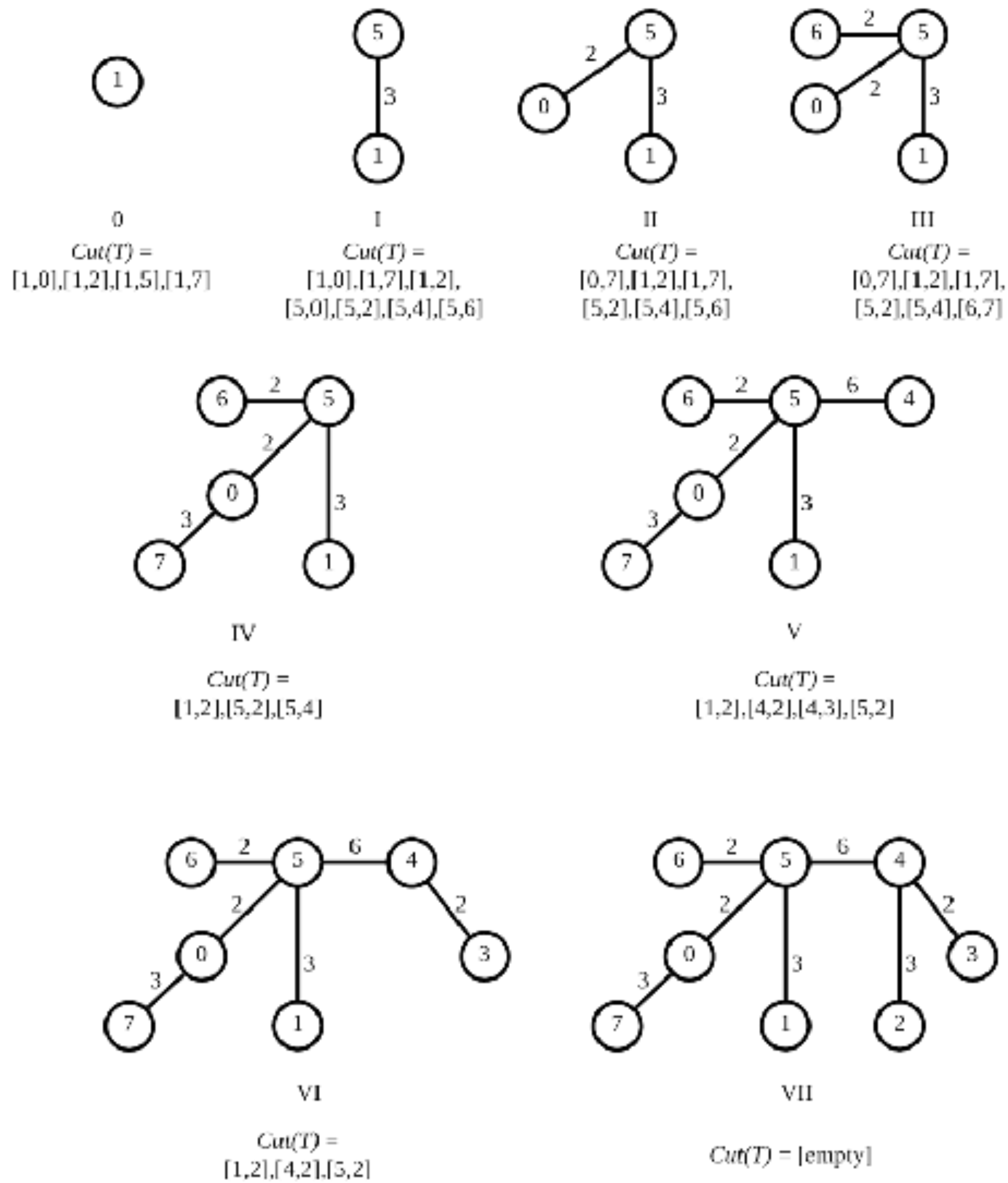


Figure 3.5 - Stages of *Prim's* Algorithm

Q4 - Shortest Paths