Homework 5

Solution 1:

Procedure: Contract(G(V,E)):

Randomly pick an edge from G and contract. Report all remaining edges.

This procedure can be viewed as an execution of constructing the minimum spanning tree of Kruskal's algorithm. As the graph is represented using adjacency list, we rewrite the contraction in the following steps:

Steps:

1. Assign a each edge a random weight
2. Run Kruskal Algorithm to find the MST of the graph.
3. Remove the heaviest edge from the MST which create 2 components/ 2 set of nodes.
4. Find the edges between 2 set of nodes.
5. This edges represents the min-cut.

Solution 2:

Let, x = Number of tons gold ore from source A

y = Number of tons gold ore from source B

A yields 2 oz of gold per ton

B yields 3 oz of gold per ton

Then

$p = 2x + 3y$ ……(i)

and we can also write

$x + y \geq 3$ ……(ii)

$20x + 10y \leq 80$ …… (iii)

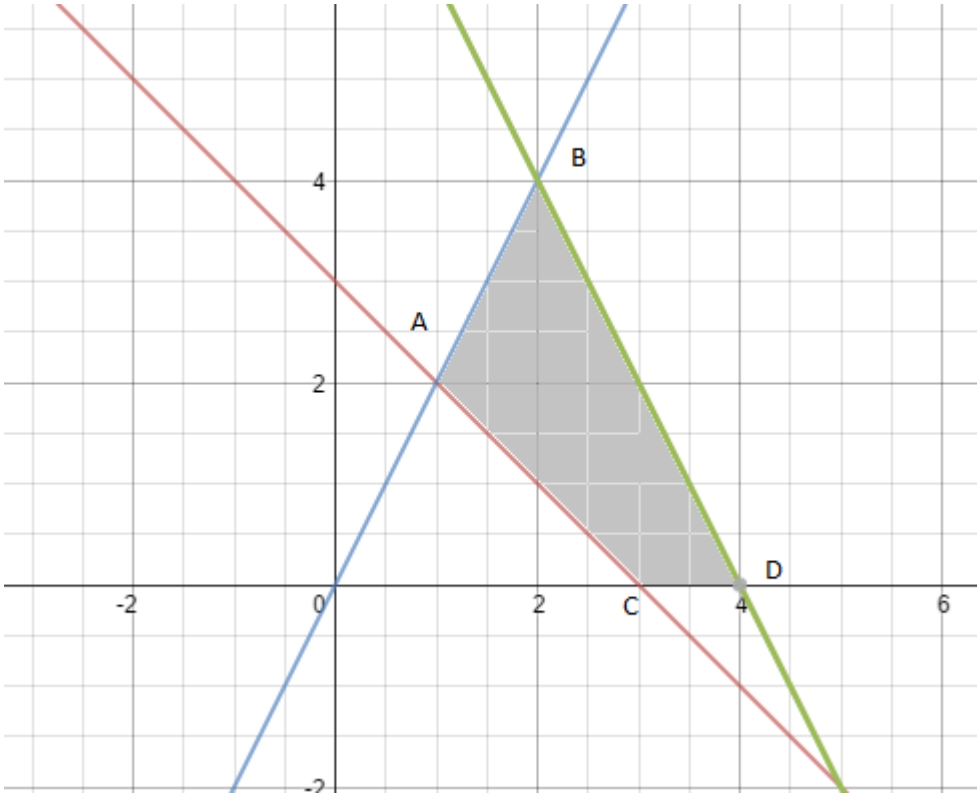
$y \leq 2x$ ……..(iv)

$x \geq 0$

$y \geq 0$

From these inequalities,

we can draw the following figure



From equation (ii) and (iv)

$x + y = 3$

$y = 2x$

we get A(x,y) = (1,2)

from equation (iii) and (iv)

$20x+10y = 80$
$y = 2x$

$B(x,y) = (2,4)$
Now we can get the max profit by putting values of A(1,2), B(2,4), C(3,0) and D(4,0) to
$p = 2x+3y$

for A(1,2), $p = 8$,
for B(2,4), $p = 16$,
for C(3,0), $p = 6$,
for D(4,0), $p = 8$

For B(2,4), we get the maximum profit 16. So, from source A 2 tons and source B 5 tons of ore must be processed each day.

Solution 3:

Let A[] represents a list of real numbers. Following is the algorithm for finding a pair of numbers in A[] whose product is exactly 1.

Algorithm:

Part 1:

Input: A[]

Output: Find a pair of numbers in A[] whose product is 1.

Steps:

```
1. Sort all the numbers A[]
2. for(i=1;i<=n;i++)
   {
        x = binary_search(1/A[i],A)
        if(x is not null)
             return (x,i);

   }
```

Running time: O(nlogn)

Part 2:

We can get a faster algorithm by using a hash function. By using a hash function we will insert the given list of real numbers into the hash table.

Steps:

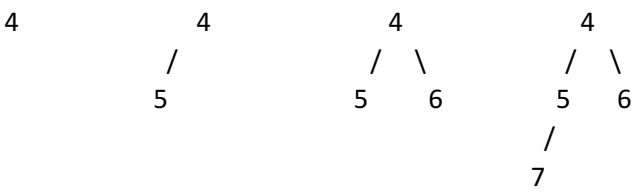
```
1. Insert n real numbers using hash function
2. for(i = 1; i<=n;i++)
   {
        x=h(1/A[i]);
        if(x is not null)
             return (x,i);

   }
```

Solution 4:

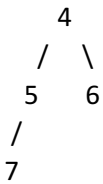
Suppose A[] is an array of 9 distinct number. A = { 5,6,4,3,2, 4,1,8,7,9 }
Now, numbers are inserted into a heap in random order. Inserting order is like

4 5 6 7 3 8 2 9 1

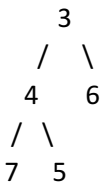
So, the heap will be like this



Now, 3 is inserted



And the tree changes to



So, the heap changes when 3, 2 and 1 minimum values are inserted

Let, X_i be a random variable which represents the i^{th} value inserted into the heap
then,

X_i = { 1 i^{th} minimum value inserted and tree changes
0 else

}

Let, X be a random variable that represents the number of times that the minimum value in the tree changes
Then, X = $\sum_{i=1}^{n} X_i$

The minimum value its inserted after the (i-1)th trial

$E(X_i) = 1/i$

Therefore, $E(X) = E(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} E(X_i)$

$= \sum_{i=1}^{n} 1/i$

$= \log n$

Solution 5:

Suppose, a row of 7 cards is ordered following

cards[] = {2, 1, 3, 4 , 5 , 6, 7}.

i=1 will be local minima if cards[1]<= cards[2].

i=7 will local minima if cards[7]<= cards[6].

if first first and last element is not local minima, take mid = (1+7)/2 = 4.

Now check cards[mid-1] >=cards[mid] <=cards[mid+1]

= cards[4-1] >=cards[4] <=cards[4+1]

= cards[3] >= cards[4] <=cards[5]

= 3>= 4 <= 5 == not true == not local minima

As 3<4, we will look into the left part, calculate the newmid then look into the left neighbors.

Then we will get, cards[1]>=cards[2]<=cards[3]

= 2 > 1< 3 , so i = 2 is the position and cards[2] = 1 is the local minima.

We can use binary search to search for local minima.

Algorithm:

Input: A row of n cards. Each one has a number Xi written on it, where i ranges from 1 to n.

Output: local minima Xi where $X_{i-1} >= X_i <= X_{i+1}$

```
SearchForLocalMinima(begin,end)
{
        if(begin ==1 && cards[1]<=cards[2])
        {
                return  cards[1];
        }
        else if(end == n && cards[n-1]>=cards[n])
        {
                return  cards[n];
        }
        mid = (begin+end)/2;
        if(cards[mid-1]>= cards[mid] <= cards[mid+1])
        {
                return cards[mid];
        }


        else if(cards[mid-1] < cards[mid]) {
                SearchForLocalMinima(begin,mid-1);
         }
```

```
        else if(cards[mid+1] < cards[mid])
        {
                SearchForLocalMinima(mid+1,end);
        }
}
```

Running Time: Time complexity is O(logn).

Solution 6:

A weighted directed graph G(V,E). Two vertices s,t ϵ V.

X_{ij} be the path that exist from V_i to V_j.

X_{ij} = 1 if arc(i,j) is in the shortest path
X_{ij} = 0 else

let, d_{ij} be the edge from V_i to V_j

Then applying linear programming formulation

min $\sum_{i,j\epsilon A} X_{ij} d_{ij}$

i) $\sum_{i:(i,k)\epsilon A} X_{ik}$ - $\sum_{j:(k,j)\epsilon A} X_{kj}$ =-1 when k= s
$\sum_{i:(i,k)\epsilon A} X_{is} = 0$ (Shortest path doesn't include edges to source)
$\sum_{j:(k,j)\epsilon A} X_{js} = 1$ (There exist one edge from source in the shortest path)

ii) $\sum_{i:(i,k)\epsilon A} X_{ik}$ - $\sum_{j:(k,j)\epsilon A} X_{kj}$ =1 when k= 1
$\sum_{i:(i,k)\epsilon A} X_{it} = 1$ (There exist one edge to destination)
$\sum_{j:(k,j)\epsilon A} X_{jt} = 0$ (There exist no edge from destination in the shortest path)

iii) $\sum_{i:(i,k)\epsilon A} X_{ik}$ - $\sum_{j:(k,j)\epsilon A} X_{kj}$ = 0 when k= A\{s,t}
If k ≠s,t and k is in shortest path
then,
$\sum_{i:(i,k)\epsilon A} X_{ik}$ = 1 (There exist one edge to k in the shortest path)
$\sum_{j:(k,j)\epsilon A} X_{kj}$ =1

If k ≠s,t and k is not in shortest path

$=> \sum_{i:(i,k)\epsilon A} X_{ik}$ = $\sum_{j:(k,j)\epsilon A} X_{kj}$ =0

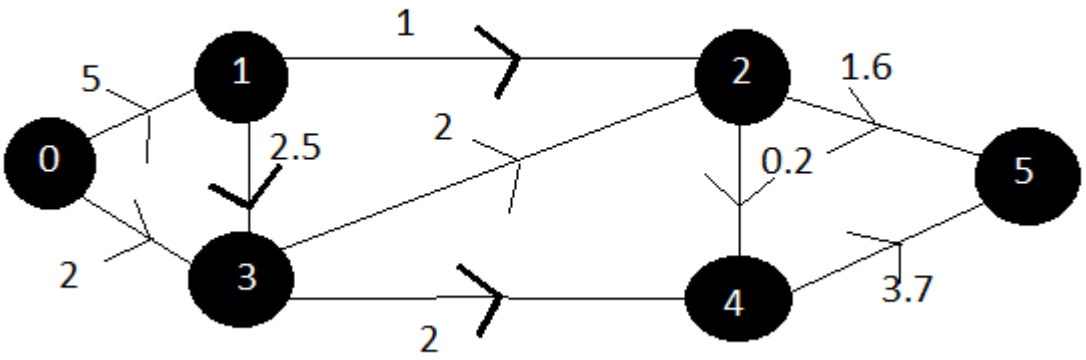
LP:

min$\sum_{i,j\epsilon A} X_{ij} d_{ij}$

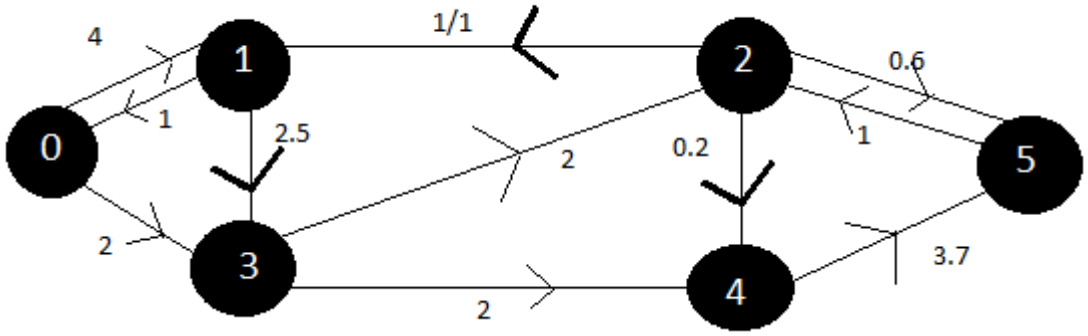
$\sum_{i:(i,k)\epsilon A} X_{ik}$ - $\sum_{j:(k,j)\epsilon A} X_{kj}$ = { -1 when k= s
1 when k= 1
0 when k= A\{s,t}
}

Solution 7:

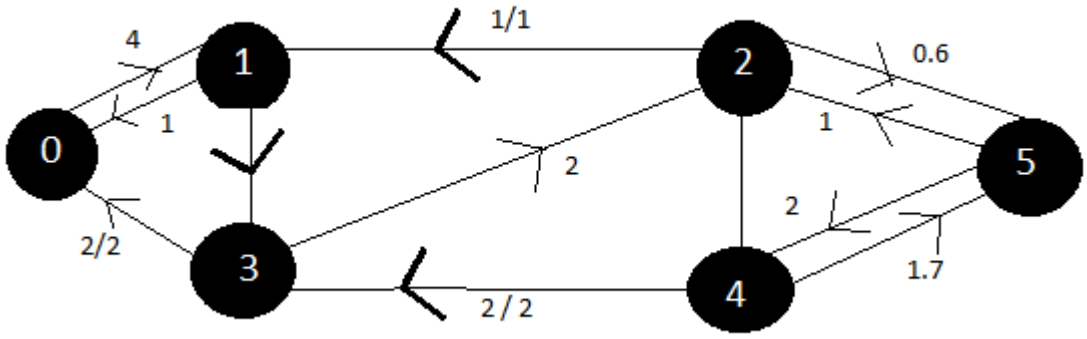
Step 1:



Step 2:



Step 3:



Solution 8:

For any bipartite graph, the maximum size of a matching is equal to the minimum size of a vertex cover. If we could find the maximum size of a matching in a bipartite graph, we can find the minimum size of a vertex cover. Suppose, BG(V,E) is the given bipartite graph and A,B are the two sets of vertices of that bipartite graph.

Algorithm:

Steps:
1. Add two vertices source s, sink t to BG(V,E)
2. Add edges from s to every vertex in A
3. From every vertex in B add edges to t
4. Make all the edge capacity 1 and direction to source s to sink t.
5. Run Ford–Fulkerson algorithm where the source is s.
6. From the flow graph find the maximum matching.

The maximum size of a matching is equal to the minimum size of a vertex cover

Solution 9:

Observe that a negative variable *x* can be viewed as the difference between two non-negative variables y-z.