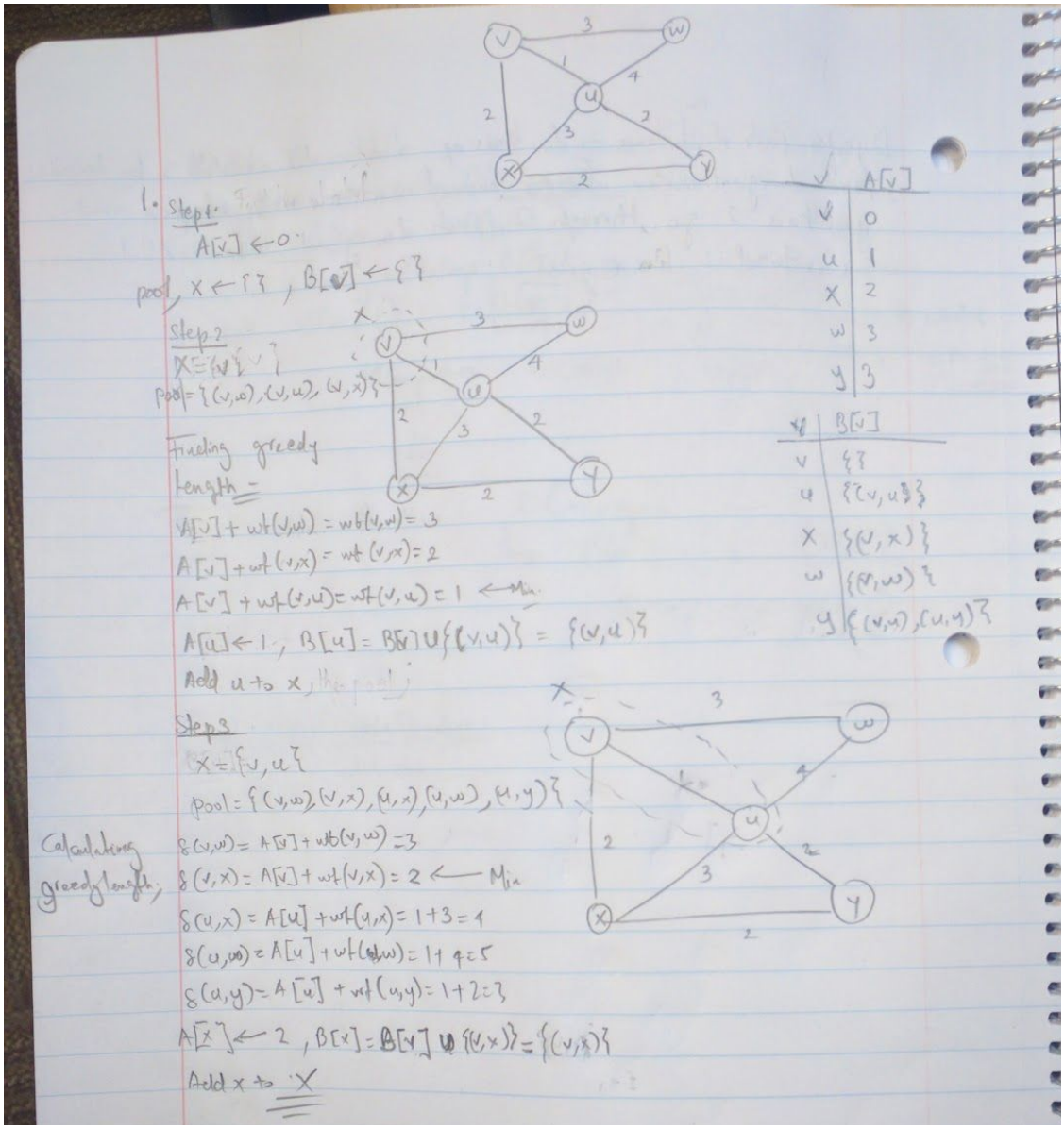
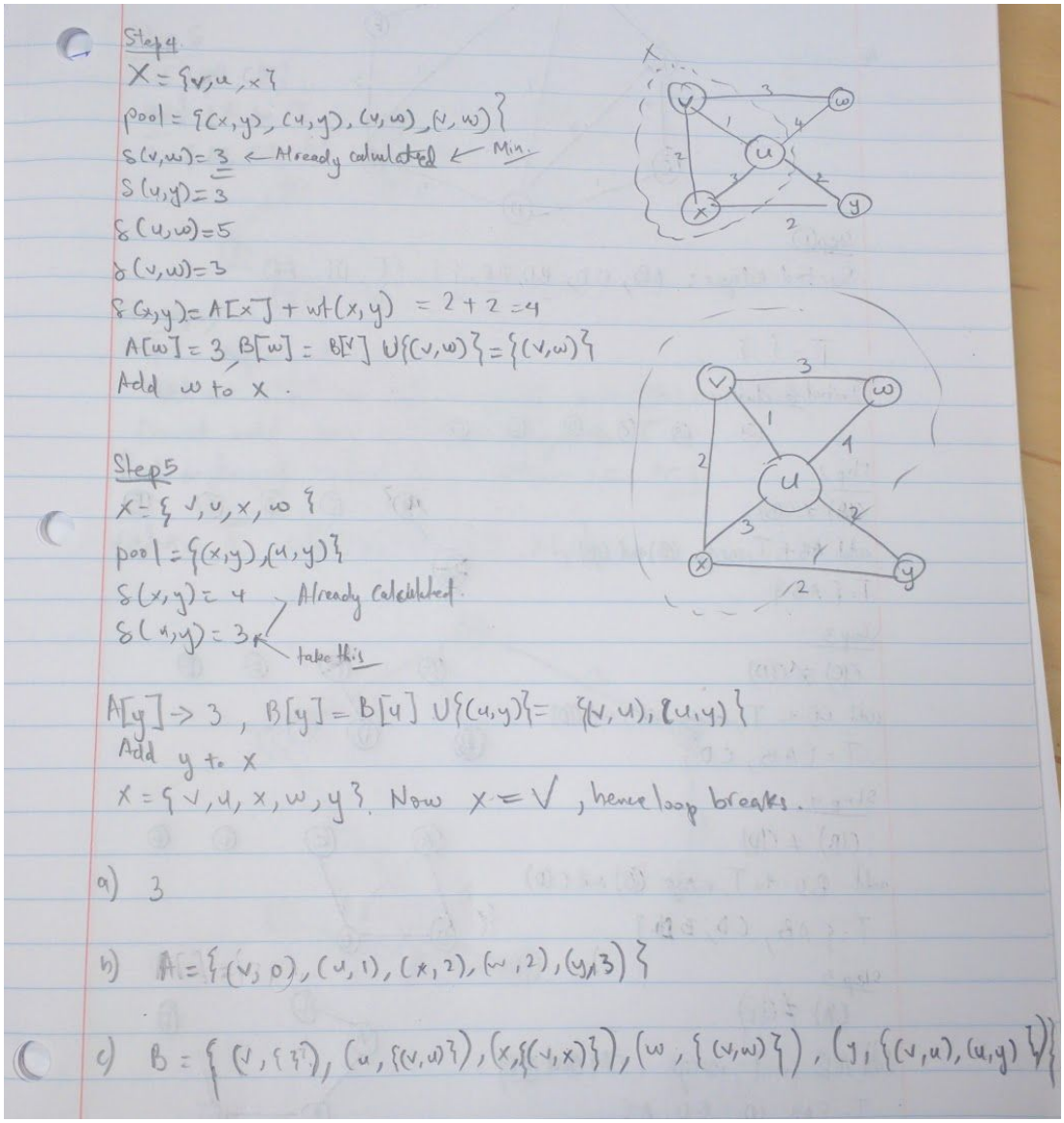
Problem 1


1. Step 1:
A[v] ← 0
pool, X ← {}, B[v] ← {}
Step 2
X = {v}
pool = {(v,w), (v,u), (v,x)}
Finding greedy length =
A[v] + wt(v,w) = wt(v,w) = 3
A[v] + wt(v,x) = wt(v,x) = 2
A[v] + wt(v,u) = wt(v,u) = 1 ← Min.
A[u] ← 1, B[u] = B[v] ∪ {(v,u)} = {(v,u)}
Add u to x, the pool
Step 3
X = {v,u}
pool = {(v,w), (v,x), (u,x), (u,w), (u,y)}
Calculating greedy length:
δ(v,w) = A[v] + wt(v,w) = 3
δ(v,x) = A[v] + wt(v,x) = 2 ← Min
δ(u,x) = A[u] + wt(u,x) = 1+3 = 4
δ(u,w) = A[u] + wt(u,w) = 1+4 = 5
δ(u,y) = A[u] + wt(u,y) = 1+2 = 3
A[x] ← 2, B[x] = B[v] ∪ {(v,x)} = {(v,x)}
Add x to X
v | A[v]
v | 0
u | 1
x | 2
w | 3
y | 3
v | B[v]
v | {}
u | {(v,u)}
x | {(v,x)}
w | {(v,w)}
y | {(v,u),(u,y)}



Problem 2

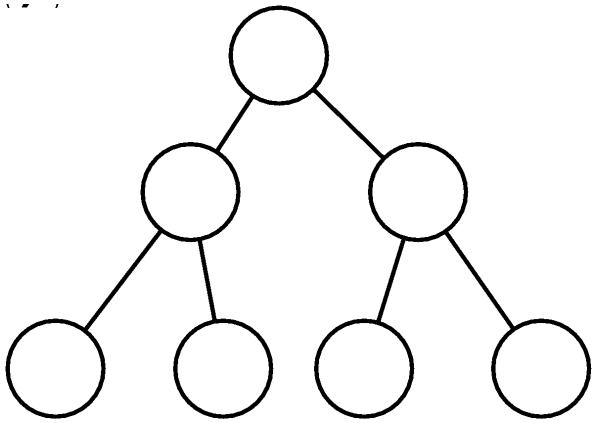
a) Dijkstra's algorithm attempts to find the shortest distance from a source vertex to any other vertex and hence the shortest path in the Graph given 3 conditions are satisfied;
   1) The graph is connected
   2) The graph does not have negative weights
   3) The graph is not directed

   In our case the graph has negative weights hence we cannot use Dijkstra's algorithm to calculate the distance.

b) BFS makes all the edge weights = 1 by creating new vertices. When the edge weights are big, many new vertices will be created, and this may lead to degraded performance of the BFS

Problem 3

We can use a hash map while building the heap-based priority queue that uses the heap key as the key and the node as the value in the map.

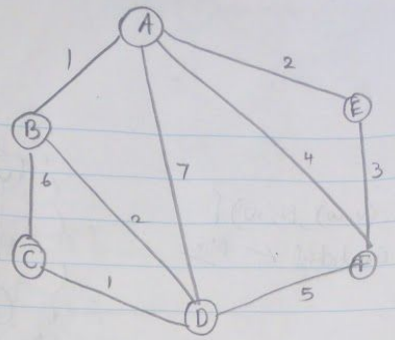


So the HashMap will have the following { A = node value, B = node value, C = node value, D = node value, E = node value, F = node value, G = node value}. The HashMap will be updated every time the operations on the heap tree are performed such as removeMin or insert.
If a node must be deleted for example node E, it is in the HashMap in O (1) time and swapped with the last value in the heap. Then a down heap or upheap can be done if necessary and this takes O (log n). Thus, the whole algorithm takes O (log n) time.

For example using the Array implementation of a priority queue, if the indices of the array are stored in a HashMap we can find the element to be deleted in O (1) time and do down heap or upheap in O (log n) time. Thus, the algorithm runs in O (log n)

Problem 4

4. Sorting



Step (1)

Sorted edges: AB, CD, BD, AE, EF, AF, DF, AD

$T = \{\ \}$

Initializing clusters.

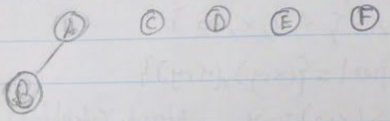
(A) (B) (C) (D) (E) (F)

Step 2

$C(A) \neq C(B)$

add AB to T, merge C(A) and C(B)

$T = \{AB\}$

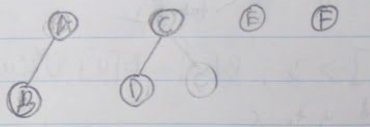


Step 3

$C(C) \neq C(D)$

add CD to T, merge C(C) and C(D)

$T = \{AB, CD\}$

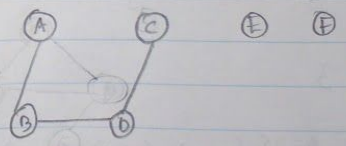


Step 4

$C(B) \neq C(D)$

add BD to T, merge C(B) and C(D)

$T = \{AB, CD, BD\}$

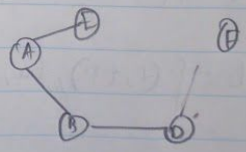


Step 5

$C(A) \neq C(E)$

Add AE to T, merge C(A) and C(E)

$T = \{AB, CD, BD, AE$

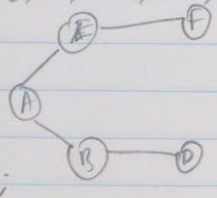


Step 6

C(E) ≠ C(F)

add EF to T and merge C(E) and C(F)

T = { AB, CD, BD, AE, EF }



For steps;

step 7, step 8 and step 9

C(B) == C(F) and C(D) == C(F) and C(A) == C(D)

Do not add any of those edges to T.

But instead break loop when i = n-1

return T. = { AB, CD, BD, AE, EF }