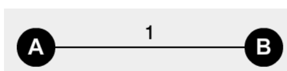
#3 Assignment - CMPT 405

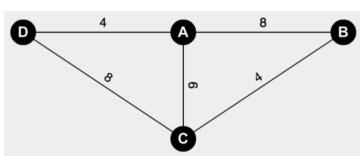
Luiz Fernando Peres de Oliveira - 301288301 - lperesde@sfu.ca

October 26, 2018

#1a)
Let G_1 be a graph with two vertices A and B and an edge (A, B) with weight 1. For every shortest path tree T_v, $v \in V$, T_v is also a MST (it is easy to see, as there is only one tree).



#1b)
Let G_2 be a graph with vertices A, B, C and D and edges $(A, B), (A, D), (A, C), (B, C)$ and (C, D), with weights $8, 4, 6, 4$ and 8, respectively. Then, no shortest path tree T_v given by Dijkstra's algorithm is a MST.



MST $= (A, C), (A, D), (B, C)$
$T_a = (A, B), (A, C), (A, D)$
$T_b = (A, B), (A, D), (B, C)$
$T_c = (A, C), (B, C), (C, D)$
$T_d = (A, B), (A, D), (C, D)$

#2a)
Intuition: Considering an unweighted graph $G = (V, E)$, every edge comprises same weight (let us say weight 1) and therefore, the least number of edges used, the better. For example while the path $i \to j$ will have total distance 1 by summing through all weights in the path, the path $i \to ... \to k ... \to j$ will have at least total weight equal to 2 (in case that the path is only $i \to k \to j$).

(1) Because of the intuition above, it is correct to say that, for a distance d_{ij}, every edge used in d_{ij} will corroborate with exactly 1 (their weight) in the distance of the path i through j. Let $(i, k) \in E$ be such edge. Then, $d_{ij} \leq d_{ik} + d_{kj}$,

where $d_{ik} = 1$. Therefore, $d_{ij} \leq 1 + d_{kj}$ and $d_{ij} - 1 \leq d_{kj}$.

(2) Likewise, we do the same for $d_{kj} \leq d_{ki} + d_{ij}$, as $(i,k) = (k,i) \in E$. So, $d_{kj} \leq 1 + d_{ij}$.

Finally, we can join notions (1) and (2) so that we prove that $d_{ij} - 1 \leq d_{kj} \leq d_{ij} + 1$.

#2b)

To show that if d_{ij} is even, then $d'_{kj} \geq d_{ij}$, we have the cases:
Case 1: $d_{ij} = 0$, then $d'_{ij} = 0$ and $d'_{kj} \in \{0, 1, 2\} \geq d'_{ij}$
Case 2: d_{ij} is even and $d_{ij} \geq 2$, then $d'_{ij} = 0$ and $d'_{kj} \in \{0, 1, 2\} \geq d'_{ij}$
Case 3: $d_{ij} = 2$, then $d'_{ij} = 2$, and $d'_{kj} = 2$, thus $d'_{kj} \geq d'_{ij}$.

To show that if d_{ij} is odd, then $d'_{kj} \leq d'_{ij}$ and $d'_{kj} < d'_{ij}$ for at least one k. Because of distance matrix D', we have two cases:
Case 1: $d_{ij} = 1$, the it is proven by part a that $d'_{kj} \leq d'_{ij}$, because of the symmetry in the distance matrix it holds that for at least one k, $d'_{kj} < d'_{ij}$.
Case 2: d_{ij} is odd and $d_{ij} \geq 3$. Then $d'_{ij} = 0$ and therefore any d'_{kj} is also equal to zero, thus $d'_{kj} \leq d'_{ij}$ holds.

#2c)
#2d)

#3)
The idea of the algorithm is to use an approach similar to the Floyd-Warshall algorithm for transitive closures, as transitive reductions are essentially the inverse of transitive closures. In Floyd-Warshall, we augment the number of edges whenever we find a path $i \to k \to j$ with total weight less than the weight of the path $i \to j$. However, in transitive reduction, we are interested in minimizing the number of edges that are used, thus, we always prefer paths $i \to j$ than $i \to k \to j$. The algorithm is very straightforward and can be thought as a modification of Floyd-Warshall and has same complexity ($O(n^3)$):
Algorithm:
Input: $G = (V, E)$

```
E' ← copy E
for i from 1 to n do
    for j from 1 to n do
        for k from 1 to n do
            if {(i,j), (i,k), (k,j)} ⊆ E', where (i,j) ≠ (i,k) ≠ (k,j) then
                E' ← E' − (i,k)
            end if
        end for
    end for
```

end for
return $G' = (V, E')$

#4)
Let T be the unrooted tree decomposition of G and T' be a nice tree decomposition of T. The idea of the algorithm is to make any node of T (preferably a internal node with many edges or minimum bag width) its root. So to ease the algorithm, we preprocess the input T: after we root T, we go through all the bag leaves b in T and create a new bag for every $v_b \in (b - parent(b))$ and make b their parent. We then run a postorder traversal on T and apply the following rules:

\- **Case 1.** If bag b is a leaf in T:

If $b \neq parent(b)$, it must have come from the preprocessing of the original bag $b' - parent(b')$, and therefore $|b| = 1 \leq |parent(b)|$, meaning that we need to add introduce nodes to our nice tree decomposition T' by adding some vertices $v_{parent} \in parent(b)$ and stop when they have the same elements, so we can join the bags later; otherwise, if b and $parent(b)$ have the same elements, we are done.

\- **Case 2.** If bag b is an internal node in T:

We know that if b is an internal node, then b is a parent of at least one bag b'. Then, the first step is to add a join node in T' for b and every b', where $parent(b') = b$. Also, because b is an internal node, b has a parent. We need first to get rid of all nodes in b that are not elements of $parent(b)$ (by adding forget nodes to T') and finally add some vertices $v_{parent} \in parent(b)$ and stop when b and $parent(b)$ have the same elements, so we can join the bags later (by adding introduce nodes to T').

\- **Case 3.** If bag b is the root in T:

Finally, if b is the root, we only need to create a join node of all bags $children(b)$ in our final nice tree decomposition T'.

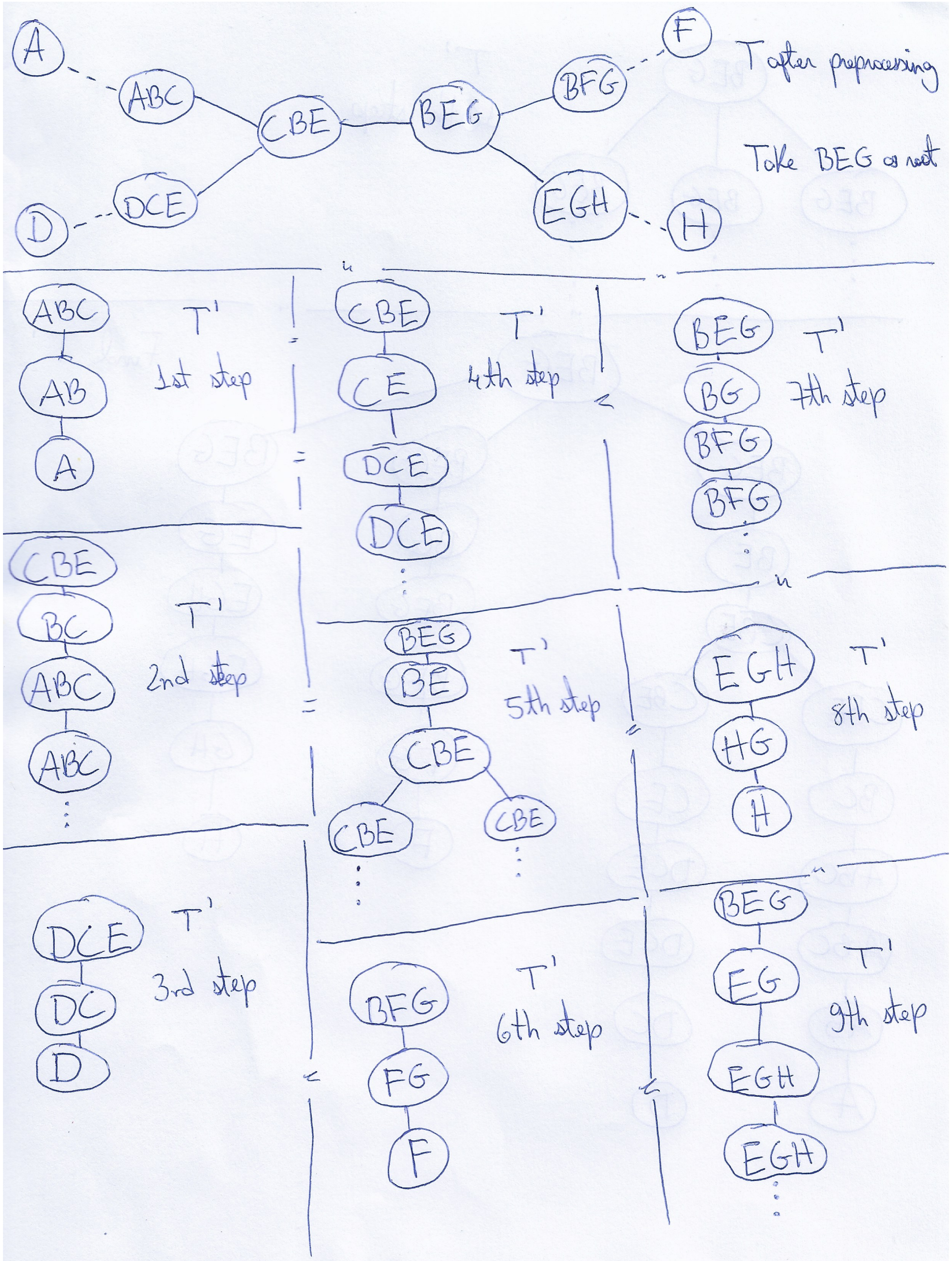
The algorithm runs in $O(nk)$ (as per the pseudocode and demonstration below)

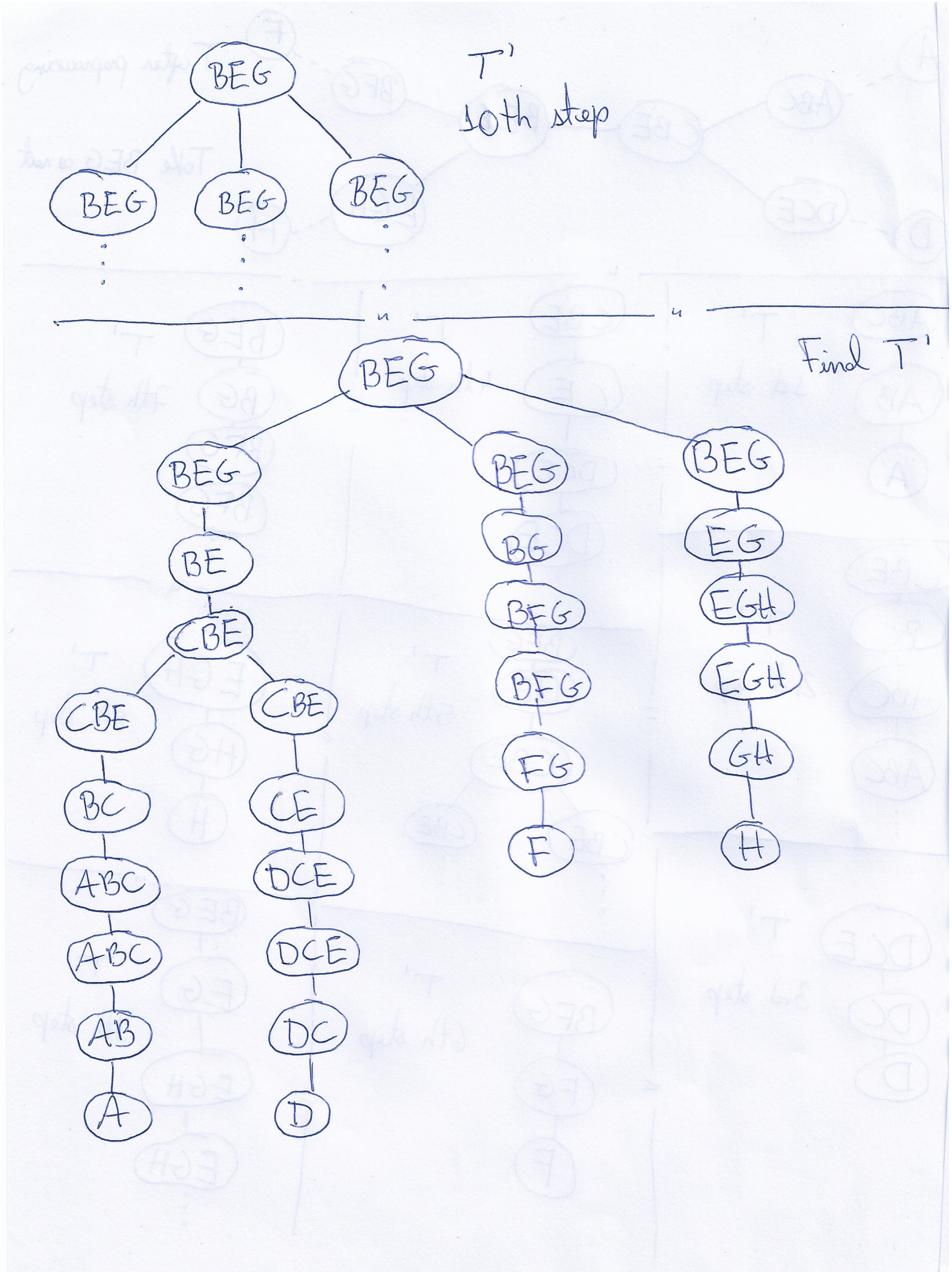
Algorithm:
Input: T

Make any internal bag (or the bag with minimum width) of T its root.
for each bag $b \in T, children(b) = \emptyset$ **do** *// all leaves*
    $free_{vs} \leftarrow b - parent(b)$
    for each $v \in free_{vs}$ **do**
        $T \leftarrow T \cup \{v\}$ such that $parent(v) = b$
    end for
end for
$T' \leftarrow \emptyset$
for each bag $b \in T_{postorder}$ **do**

$b_{aux} \leftarrow b$
if *children*$(b) = \emptyset$ **then** *// if leaf*
$T' \leftarrow T' \cup b_{aux}$ *// add leaf node*
while $b_{aux} \neq parent(b)$ **do**
add introduce node $b_{aux} \cup \{v_{parent}\}$ in T', for any $v_{parent} \in parent(b)$,
such that $b_{aux} \cup \{v_{parent}\} = parent(b_{aux})$
end while
else
Create join node in T'
if $parent(b) \neq \emptyset$ **then** *// if not root*
while $\exists v \in b$, such that $v \notin parent(b)$ **do**
add forget node $b_{aux} - \{v\}$ in T', for any $v \notin parent(b)$,
such that $b_{aux} - \{v\} = parent(b_{aux})$
end while
while $b_{aux} \neq parent(b)$ **do**
add introduce node $b_{aux} \cup \{v_{parent}\}$ in T',
for any $v_{parent} \in parent(b)$, such that $b_{aux} \cup \{v_{parent}\} = parent(b_{aux})$
end while
end if
end if

end for
return T'


T after preprocessing
Take BEG as root
A
ABC
CBE
BEG
BFG
F
D
DCE
EGH
H
T'
1st step
ABC
AB
A
T'
2nd step
CBE
BC
ABC
ABC
T'
3rd step
DCE
DC
D
T'
4th step
CBE
CE
DCE
DCE
T'
5th step
BEG
BE
CBE
CBE
CBE
T'
6th step
BFG
FG
F
T'
7th step
BEG
BG
BFG
BFG
T'
8th step
EGH
HG
H
T'
9th step
BEG
EG
EGH
EGH


T'
10th step
BEG
BEG
BEG
BEG
Find T'
BEG
BEG
BE
CBE
CBE
BC
ABC
ABC
AB
A
CBE
CE
DCE
DCE
DC
D
BEG
BG
BFG
BFG
FG
F
BEG
EG
EGH
EGH
GH
H

Note.: I know joins are only possible with two bags. It was too late to fix that when I realized I did this mistake in my examples and algorithm on #4, please ignore it. Imagine we made T a binary tree and worked from there (check image on question #5).

#5)

Definition: (As per lecture 17). Let B_x be the vertices appearing in node x and let V_x be the vertices in the subtree rooted at x. Let $M[x, S]$ be a matrix that keeps the size a maximum independent set $I \subseteq V_x$ with $I \cap B_x = S$. At the end of the algorithm, the maximum independent set size will them be on $M[root, S'_{root}]$, where S'_{root} is the best S for the root.

Recurrence:

- Leaf: B_x has no children
$M[x, S] = 1$

- Introduce: 1 child y, $B_x = B_y \cup \{v\}$, for some vertex v

$$M[x, S] = \begin{cases} M[y, S] & \text{if } v \notin S \\ M[y, S - \{v\}] + |v| & \text{if } v \in S \text{ but } v \text{ has no neighbor in } S \\ -\infty & \text{if } S \text{ contains } v \text{ and its neighbors} \end{cases}$$

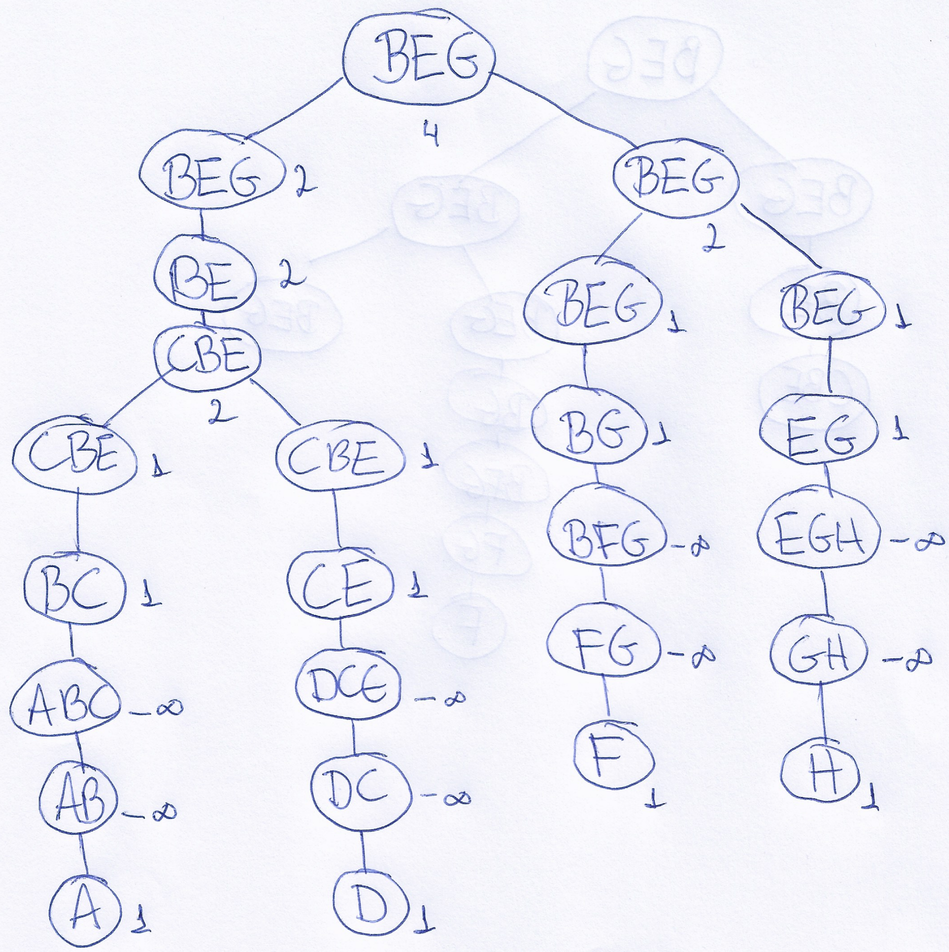
- Forget: 1 child y, $B_x = B_y - \{v\}$, for some vertex v
$M[x, S] = max\big(M[y, S], M[y, S \cup \{v\}]\big)$

- Join: 2 children y_1, y_2, $B_x = B_{y1} = B_{y2}$
$M[x, S] = M[y_1, S] + M[y_2, S] - |S|$

Algorithm:
Input: $T', root$

for each $x \in T' postorder$ **do**
    for each $I \subseteq V_x$ with $S = I \cap B_x$ **do**
        compute recurrence $M[x, S]$ as described above
        keep best S in S' for every x
    end for
end for
return $M[root, S'_{root}]$

Demonstration: There are at most $2^{k+1} \times n$ subproblems (entries) in M and therefore its running time is bounded on M. Demonstrate that would take a matrix of size $2^4 \times 25$, which is very large. Thus, the image below summarizes it:


BEG
4
BEG 2
BE 2
CBE
2
CBE 1
BC 1
ABC −∞
AB −∞
A 1
CBE 1
CE 1
DCE −∞
DC −∞
D 1
BEG
2
BEG 1
BG 1
BFG −∞
FG −∞
F 1
BEG 1
EG 1
EGH −∞
GH −∞
H 1