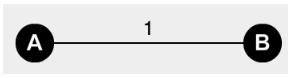
#3 Assignment - CMPT 405

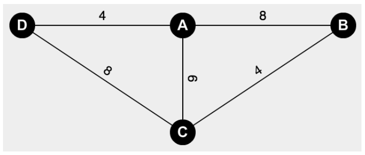
Luiz Fernando Peres de Oliveira - 301288301 - lperesde@sfu.ca

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

#2)
#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: $G = (V, E)$
  $E' \leftarrow$ copy $E$
  for $i$ from 1 to $n$ do
      for $j$ from 1 to $n$ do
          for $k$ from 1 to $n$ do
              if $\{(i,j),(i,k),(k,j)\} \subseteq E'$, where $(i,j) \neq (i,k) \neq (k,j)$ then
                  $E' \leftarrow E' - (i,k)$
              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:

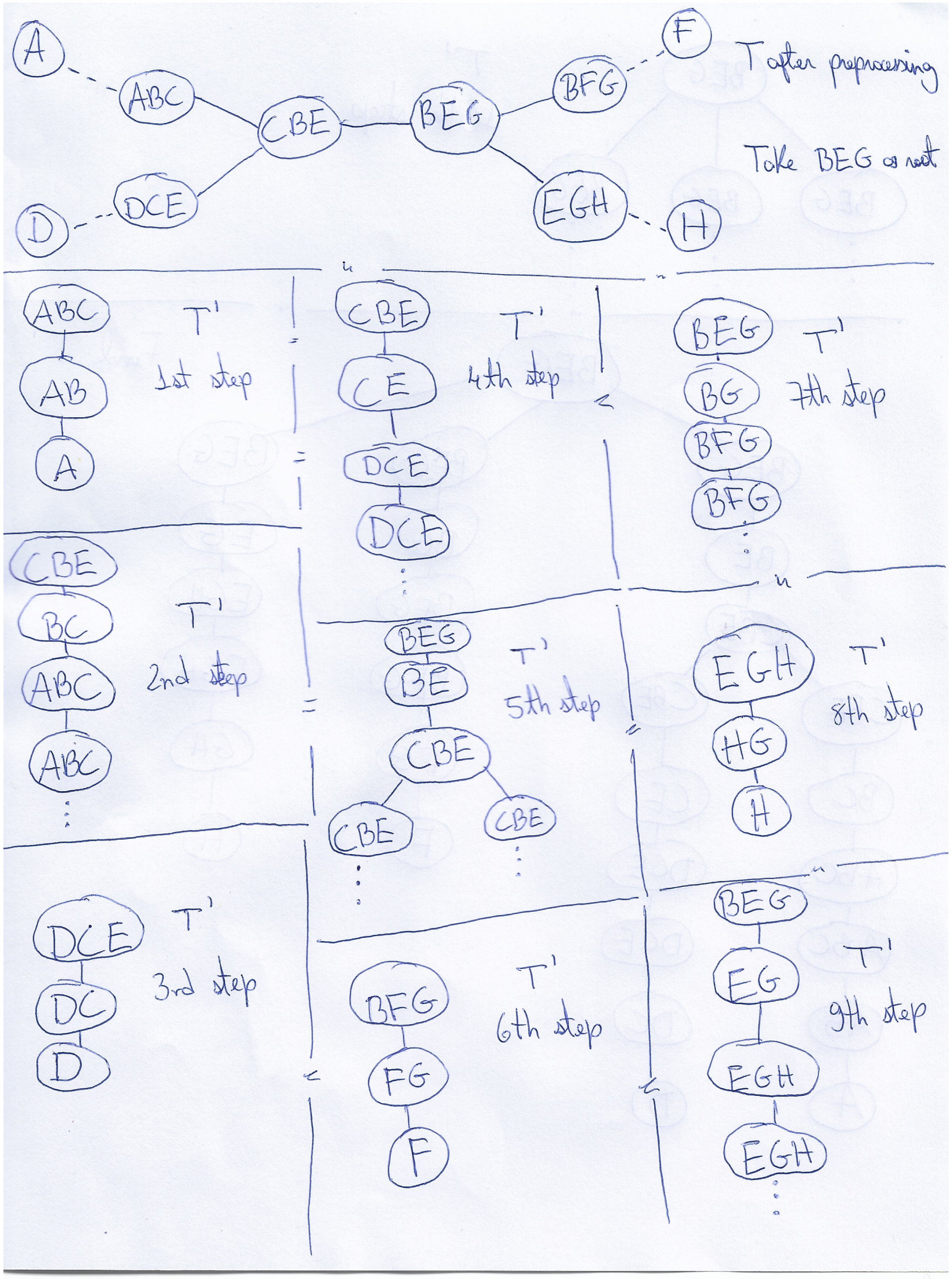
Finally, if b is the root, we only need to create a join node of all bags $children(b)$ in our final nice tree decomposition T'.

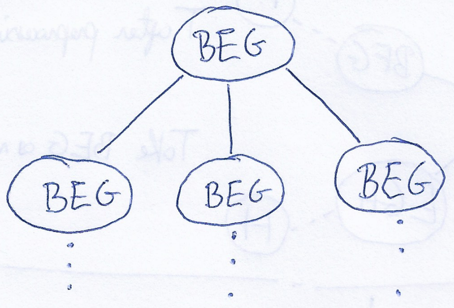
The algorithm runs in $O(nk)$ (as per the pseudocode and demonstration below)

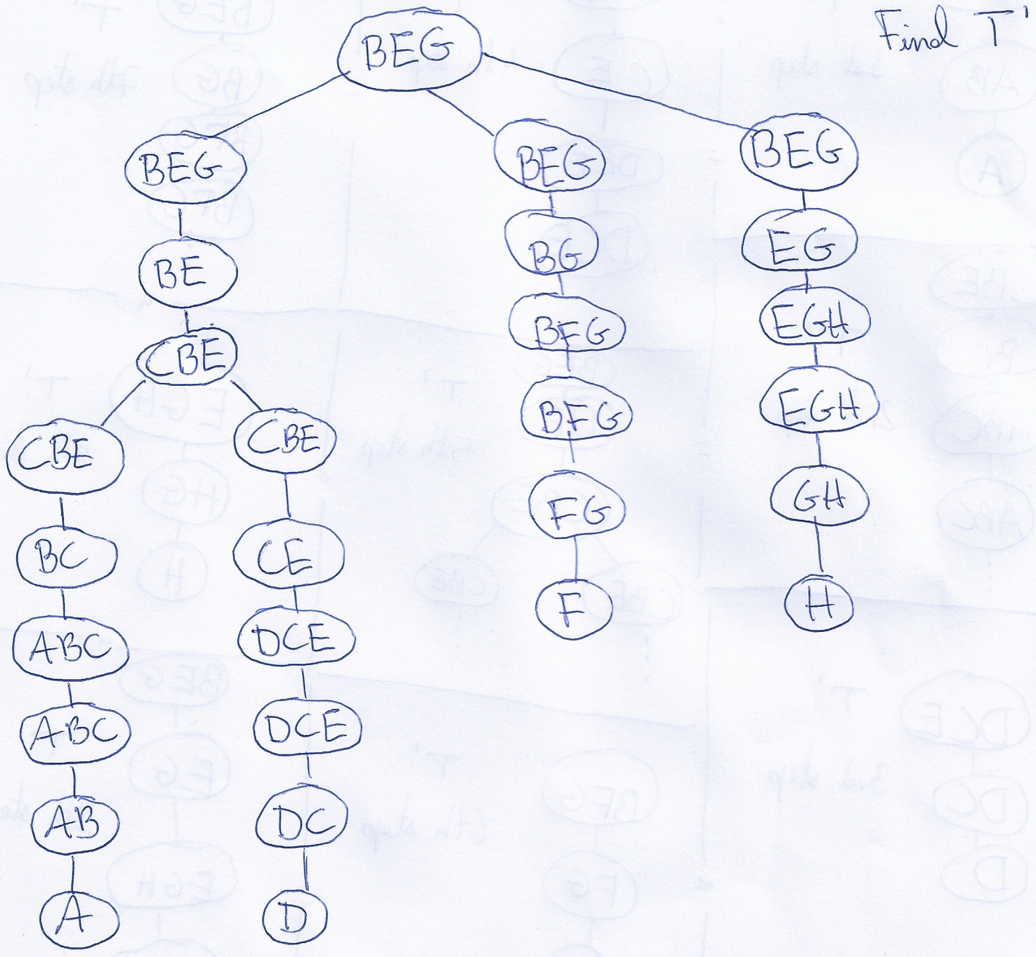
Algorithm:

```
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


Find T'

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, please ignore it. Imagine we made T a binary tree and worked from there.

#5)
References