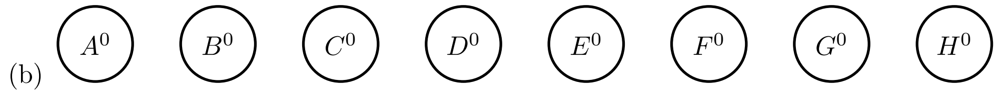
CS 3510 - Spring 2009
Homework 3
Due: March 25

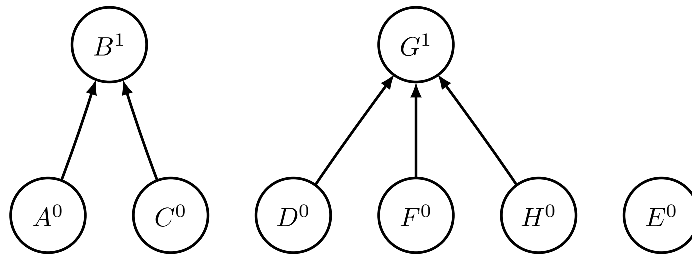
1. Problem 5.2 from [DPV]. You do not need to use path compression.

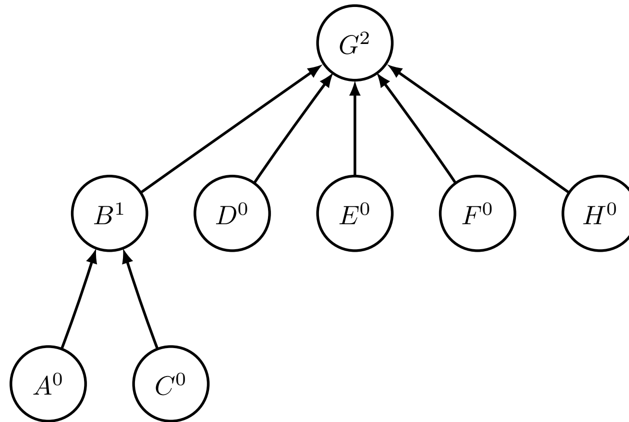
(a)

Set S	A	B	C	D	E	F	G	H
{}	0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
A		$1/A$	∞/nil	∞/nil	$4/A$	$8/A$	∞/nil	∞/nil
A, B			$2/B$	∞/nil		$6/B$	$6/B$	∞/nil
A, B, C				$3/C$		$6/B$	$2/C$	∞/nil
A, B, C, G				$1/G$		$1/G$		$1/G$
A, B, C, G, D								
A, B, C, G, D, F								
A, B, C, G, D, F, H								
A, B, C, G, D, F, H, E								

(b)



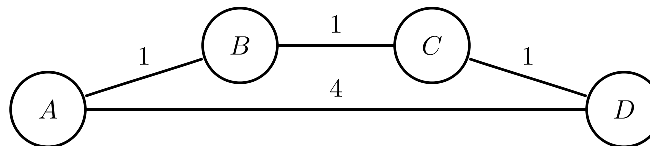




2. Problem 5.5 from [DPV]

   (a) Suppose that the minimum spanning tree was initially T. After each edge weight is increased by 1, the MST changes to $\hat{T}$. Therefore there will be at least one edge $(u, v) \in T$ but $(u, v) \notin \hat{T}$. Suppose we add edge (u, v) to the tree $\hat{T}$. $T^* = \hat{T} + (u, v)$. Since (u, v) was not in $\hat{T}$, (u, v) must therefore be the longest edge in the cycle formed in T^*. Since (u, v) is the longest edge in this cycle, it will be the longest edge in the same cycle in T because all edges in the cycle are decreased by the same amount, 1. Therefore (u, v) will not be in T either. This contradicts $(u, v) \in T$. So T and $\hat{T}$ are equivalent.

   (b) Consider the following counter-example. Intially, the shortest path from A to D is of length 3, $ABCD$. However, once we increase each edge weight by 1, the previous shortest path is now length 6 and the new shortest path is of length 5, AD.



3. Problem 5.14 from [DPV]

   (a)

Symbol	Codeword
a	0
b	10
c	111
d	1100
e	1101

(b) $1000000 * (1/2) * 1$ bit $+1000000 * (1/4) * 2$ bits $+1000000 * (1/8) * 3$ bits $+1000000 * (1/16) * 4$ bits $+1000000 * (1/16) * 4$ bits $= 1,875,000$ bits

4. Problem 5.23 from [DPV]

   (a) ($e \notin E'$ and $\hat{w}(e) > w(e)$): If we run Kruskal's algorithm with the new weights we will end up choosing the same edges as with the original weights, so the MST will remain the same. When we ran Kruskal's algorithm with the original weights we did not add the edge e, so it must form a cycle by the time we have considered all lighter edges. With the new weight we will be considering it at the same time, or later, so it will again form a cycle and not be included in the MST. Therefore the tree stays the same and no update is necessary.

   (b) ($e \notin E'$ and $\hat{w}(e) < w(e)$): The edge e, may now enter the MST, forcing out some other edge. We look at the cycle that e induces in the original graph and if there is some edge with a larger weight than the new value for e, then it will be replaced by e in the MST. It is possible that this cycle includes all nodes of the original graph and so that each must be checked to find the edge with maximum weight on the cycle, leading to a worst case $O(n)$ running time.

   (c) ($e \in E'$ and $\hat{w}(e) < w(e)$): The edge e, may now have a lower weight than some edge considered previously (when running Kruskal's algorithm). However, since e is in the MST, this earlier edge did not form a cycle with e. This remains true even if we change the order of the selection of these edges to occur after we select e. As a result, the MST remains the same.

   (d) ($e \in E'$ and $\hat{w}(e) > w(e)$): Since the weight of the edge e has increased, it is possible that it will be replaced in the MST by some edge currently not in the tree. Removing e from the MST, will create two disconnected subgraphs and we must examine all edges that connect these two subgraphs. If the weight of an edge is less than that of e, then it will fill e's role as connecting the subgraphs with the minimum cost. To find the lightest edge connecting the two subgraphs we need to examine each edge and determine, in linear time, whether both endpoints are in the same connected component. Now we recall that we can find connected components in linear time using DFS!! Just run DFS on the MST with e removed, and label all the vertices with "cc1" if they are in the first part of the tree and "cc2" if they are in the other part of the tree. Now that we have these labels, we

just examine all the edges and, among the ones with different labels on their endpoints, we find the lightest edge e'. The new MST is $E' \setminus \{e\} \cup \{e'\}$.

5. Problem 6.1 from [DPV]

   We want to compute intermediate sums starting at the initial position. If the running sum ever drops below zero, then we know that if a new maximum sum does exist, it will occur starting at the position $j + 1$, where j was the position that the running sum dropped below zero. So we start a new running sum beginning at position $j + 1$.

```
maximum-sum(int[] sequence)
1  max = 0, sum = 0
2  for i = 1 to sequence.length
3    if (sum > max) //update max
4      max = sum
5    elseif(sum < 0) //negative sum
6      sum = 0 //reset running sum
7  return max
```

   This algorithm loops n times and contains only constant time operations so it is $O(n)$.

6. Problem 6.4 from [DPV]

   (a) Consider a 2-D array of boolean values $T(i, j)$, where $T(i, j)$ is true if and only if $s[1 \ldots j]$ can be reconstituted as a sequence of valid words. The fact that the current word $s[i..j]$ is only a valid reconstitution if its prefix is one allows us to express T in terms of this subproblem, $T(i, j) = T(1, i-1) \wedge$ dict$[s[i \ldots j]]$. Since the first row has no prefix, it is initalized to dict$[s[1 \ldots j]]$ for j = 1 to n. So if we walk through the matrix row by row, we can compute all values for T in $O(n^2)$ time.

```
has-recon(s[1...n])
1  Create an n x n Matrix T
2  for j = 1 to n
3    T(1, j) = dict[s[1...j]]
4  for i = 2 to n
5    for j = i to n
6      T(i, j) = T(1, i - 1)∧ dict[s[i...j]]
7      if j equals n and T(i, j) is true
8        return true
9  return false
```

(b) In order to accommodate finding this sequence of words, we can alter T so that it contains pointers to the last word in the reconstitution up to position j, and null otherwise. Once we satisfy the truth condition, we can return the last word s_n. We can retrieve each subsequent word in the following way, $s_{j-1} = T(1,$ j - length$(s_j))$ and so on until $j - length(s_j)$ becomes zero.This returns the words in reverse order, but they can be easily reversed.

7. Problem 6.8 from [DPV]

Given two strings, S of length m and T of length n, find the longest strings which is a substring of both S and T. You first find the longest common suffix for all pairs of prefixes of the strings. The longest common suffix is

$$\text{lcsuffix}(S_{1\ldots p}, T_{1\ldots q}) = \begin{cases} \text{lcsuffix}(S_{1\ldots p-1}, T_{1\ldots q-1}) + 1) & \text{if} S[p] = T[q] \\ 0 & \text{otherwise} \end{cases}$$

The maximal of these longest common suffixes of possible prefixes must be the longest common substrings of S and T.

8. Problem 6.17 from [DPV]

We define $D(v)$ as a predicate which evaluates to true if it is possible to make change for v using available demoninations $x_1, x_2, \ldots, x_n$. If it is possible to make change, then it is possible to make change for $v - x_i$, using the same dominations with one coin of x_i fixed. Since we do not know, for which i that this condition holds, we will do a logical or over all i. The recursive definition of $D(v)$ written as follows.

$$D(v) = V_{1 \leq i \leq n} \begin{cases} D(v - x_i) & x_i \leq v \\ \text{false} & x_i > v \end{cases}$$

Using this definition, the algorithm is as follows.

make-change$(x_1, \ldots, v)$
1 Create an array D of size $v + 1$
2 $D[0] =$ true
3 for $i = 1$ to v
4 for $j = 1$ to n
5 if $x_j \leq i$
6 $D[i] = D[i] \vee D[i - x_j]$
7 else
8 $D[i] = false$
9 return $D[v]$

The outer loop runs v times, while the inner loop runs n giving an overall $O(nv)$ algorithm.