Data Structures and Algorithms

Spring 2014

Cesar Agustin Garcia Vazquez

March 4, 2014

1 Problem Set 04

1. Suppose you are given a weighted graph $G = (V, E)$ and an edge $e \in E$. You are asked whether e is in some minimum spanning tree of G. Give a linear time algorithm for the problem. (It it helps, first consider the case where all edge weights are distinct - this will get you most of the credit - and then try the more general case.)

   Answer: In order for e to be in some minimum spanning tree, then e must have a weigh less than the $|V|$-th minimum element, which does not generate a cycle.

   To design this algorithm we are going to take a step-by-step approach. First, we iterate through all the elements and we compare if e is strictly greater than the current edge, if so, we add 1 to our counter of minimum elements. After the iteration is over, if the counter of minimum elements is less than the number of nodes minus 2, which means that e is the $(|V| - 1)$-th element in the set of the minimum spanning tree. So iterating over the elements and one comparison at the end gives an $O(n)$ algorithm.

   The above algorithm is very straightforward as naive as well, because it does not consider the case when an edge generates a cycle in the minimum spanning tree. i.e., even though the counter of minimum edges is greater than $|V| - 2$, that does not mean that e is not in a minimum spanning tree, as can be seen in Figure 1, where we have 6 MSTs.

   If we consider the graph in Figure 1 and the edges $E = \{(A, B), (A, C), (B, C), (B, D), (C, D)\}$, and we are required to determine if $e = (C, D)$ is in a minimum spanning tree, will count the first 3 edges less than e and conclude that e is not part of any minimum spanning tree. This is mainly because that graph have multiple minimum spanning trees of weight 2 as can be seen in Figure 2.

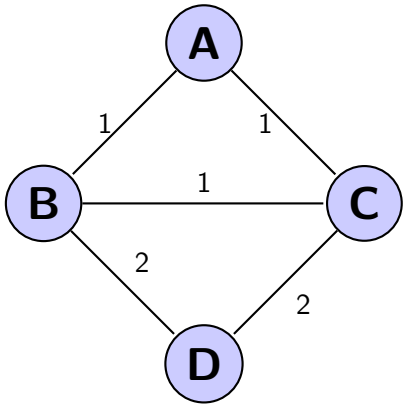
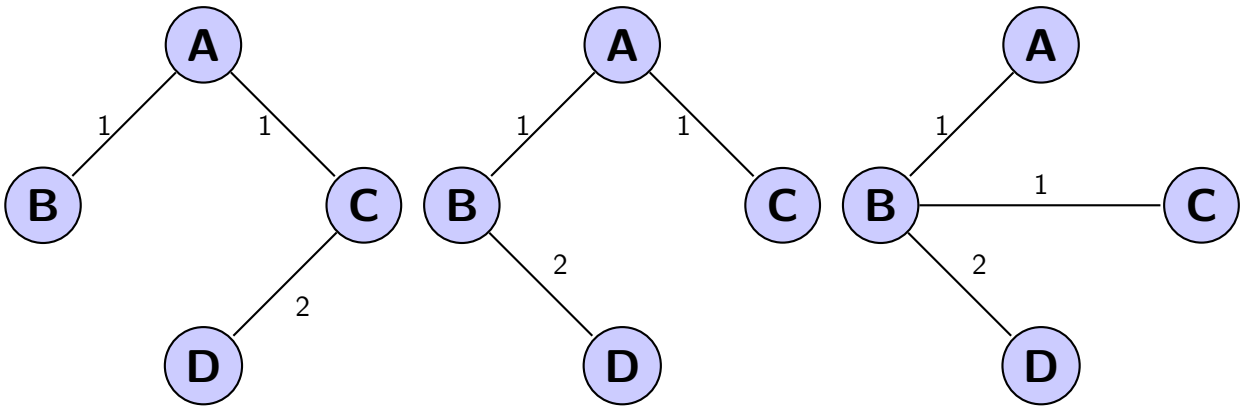


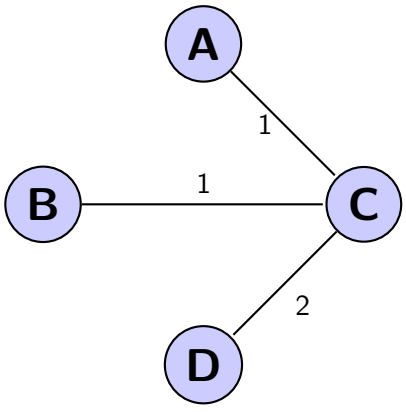
Figure 1: Graph with 6 MSTs



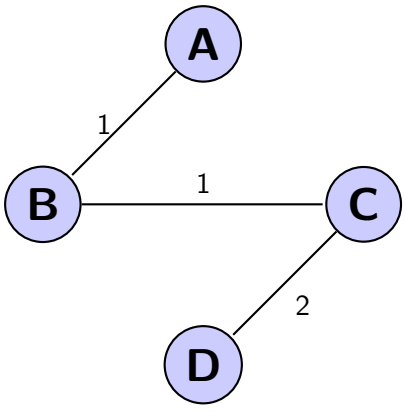
(a) 1st MST

(b) 2nd MST

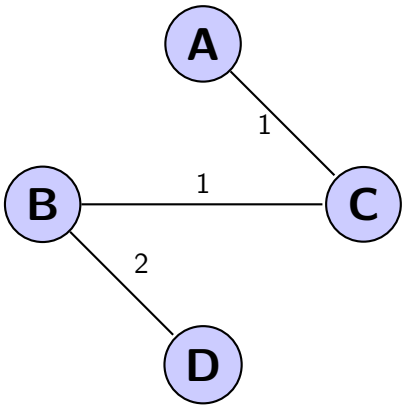
(c) 3rd MST



(d) 4th MST



(e) 5th MST



(f) 6th MST

Figure 2: Minimum Spanning Trees from graph in Figure 1

Notice that in this case, the algorithm finishes but the edges considered do not form a minimum spanning tree since the node D is not in any edge counted. So what we need to consider as well as is the number of nodes in the edges that we countered is equal to the number of nodes in the graph.

So the algorithm 1 solves the problem in linear time, where the union operation can take constant time if the set is implemented using a hash map or if we just add a property flag to the node and set it to true when it is added to the set.

Algorithm 1: Algorithm to determine if an edge e is in a MST

```
1 funcion isInMST(V, E, e)
2 NodesInMST = { }
3 foreach edge ∈ E do
4 |   if edge.weight < e.weight then
5 |   |   NodesInMST ∪= { edge.sourceNode, edge.destinationNode }
6 if |NodesInMST| < |V| then
7 |   return true
8 return false
```

2. Given a family of set cover problems where the set to be covered has n elements, the minimum set cover is size $k = 3$, and the greedy algorithm returns a cover of size $\Omega(\log n)$. That is, you should give a description of a of a set cover problem that works for a set of values of n that grows to infinity - you might begin, for example, by saying, "Consider the set $X = \{1, 2, \ldots, 2^b\}$ for any $b \geq 10$, and consider subsets of X of the form ...", and finish by saying "We have shown that for the example above, the set cover returned by the greedy algorithm is of size $b = \Omega(\log n)$." (Your actual wording may differ substantially, of course, but this is the sort of thing we're looking for.) Explain briefly how to generalize your construction for other (constant) values of k. (You need not give a complete proof of your generalization, but explain the types of changes needed from the case of $k = 3$.)

3. Consider the following scheduling problem: we have two machines, and a set of jobs $j_1, j_2, j_3, \ldots, j_n$ that we have to process one at a time. To process a job, we place it on a machine. Each job j_i has an associated running time r_i. The load on the machine is the sum of the running times of the jobs placed on it. The goal is to minimize the competition time, which is the maximum load over all machines.
   Suppose we adopt a greedy algorithm: each job j_i is put on the machine with the minimum load after the first $i - 1$ jobs. (Ties can be broken arbitrarily.)

   a) Show that this strategy yields a completion time within a factor of 3/2 of the best possible placement of jobs. (Hint: Think of the best possible placement of jobs. Even for the best placement, the completion time is at least as big as the biggest job, and at least as big as half the sum of the jobs. You may want to use both of these facts.)

      Answer: We are going to do the proof by induction on the number of jobs. If we have only one job, then the greedy algorithm places in either m_1 or m_2 and this solution is optimal. Also, if we have two jobs, the greedy algorithm places

the job j_1 in m_1 and then places job j_2 in m_2 and this solution is optimal.
As induction hypothesis, we assume that the greedy algorithm had yielded an optimal solution of the placement of jobs in $j_1, \ldots, j_k$. For the inductive step, we have to consider what is the optimal case. If we have k jobs, then one possible optimal workload is $\frac{1}{2}\sum_{i=1}^{k} r_i$, i.e., all the jobs were distributed equally over the two machines.
Another possible optimal solution happens when the greatest running for a job j_i, is $r_i > \frac{1}{2}\sum_{i=1}^{k} r_i$. So let j_{k+1} be last job to be assigned to the machine m_1, which means that m_1 had smaller workload than m_2 after k jobs have been assigned. That means that workload of m_1 after k jobs $w_1 \leq \max\{\frac{1}{2}\sum_{i=1}^{k} r_i, \max\{r_i\}\}$
If we assume that $\max\{\frac{1}{2}\sum_{i=1}^{k} r_i, \max\{r_i\}\} = \frac{1}{2}\sum_{i=1}^{k} r_i$ and we call this optimum value OV, then the workload, after the job j_{k+1} is assigned, is $w_1 \leq r_{k+1} + OV$.
At this point, we just need to bound r_{k+1}. This can be easily done if we consider the condition of assignment to m_1 which had a workload less than or equal to m_2, and also by our assumption $r_{k+1} \leq \frac{1}{2}\sum_{i=1}^{k} r_i$. Which would give us a bound of 2, i.e., the strategy yields a competition time within a factor of 2 of the best possible placement of jobs.
We can tight a little more if we consider that $r_{k+1} \leq w_2/2$ due to the induction hypothesis, from which we also have that $w_1, w_2 \leq OV$, then $r_{k+1} \leq OV/2$.
Hence we have that $w_{k+1} \leq r_{k+1} + OV \leq \frac{1}{2}OV + OV = \frac{3}{2}OV$.

b) Give an example where a factor of 3/2 is achieved.

Answer: If we consider the jobs j_1, j_2 and j_3 and their associated running times $r_1 = 2, r_2 = 2$ and $r_3 = 4$. The algorithm will pick the job j_1 and assign it to the machine m_1. Then the job j_2 will be assigned to the machine m_2 because its load is 0 and the load of m_1 is 2. Finally the algorithm will pick the last job j_3 and assign arbitrarily to any machine and finish. At the end of the assignments, one machine will have a workload of 2 and the other machine will have a workload of 6, when the optimal arrangement is to place the jobs j_1 and j_2 in one machine and the job j_3 in another to have workload of 4.

c) Suppose now instead of 2 machines we have m machines. What is the performance of the greedy solution, compared to the optimal, as a function of m?

Answer: We can generalize the Optimum Value from $\frac{1}{2}\sum_{i=1}^{k} r_i$ to $\frac{1}{m}\sum_{i=1}^{k} r_i$. By the previous exercise, we found an upper bound with a factor of 2, which is independent of how many machines we have, and then we decrease the value base considering the number of machines that we had (2). What we did was to take the Optimal Value and divide it by the number of machines, so in this case it would 2 ·OV - $\frac{1}{m}$OV = $\left(2 - \frac{1}{m}\right)$OV.

d) Give a family of examples (that is, one of each m - if they are very similar, it will be easier to write down!) where the factor separating the optimal and the

greedy solution is as large as you can make it.

Answer: We can consider the family of jobs whose running times are $r_1 = 1 + d, r_2 = 2 + d, \ldots, r_k = k + d$. So the Optimum Value is going to be given by $\frac{1}{m} \sum_{i=1}^{k} i + \frac{d \cdot k}{m}$.

4. Consider an algorithm for integer multiplication of two n-digit numbers where each number is split into three parts, each with $n/3$ digits.

   a) Design and explain such an algorithm, similar to the integer multiplication algorithm presented in class. Your algorithm should describe how to multiply the two integers using only six multiplications on the smaller parts (instead of the straightforward nine)

   Answer: To solve this problem, let's consider multiplying $x = 123$ by $y = 456$, so $n = 3$, and we have six groups of numbers of just one digit. Using standard multiplication the result is $56,088$.

   We can generalize the procedure considering $x = a \cdot 10^{2n/3} + b \cdot 10^{n/3} + c$ and $y = d \cdot 10^{2n/3} + e \cdot 10^{n/3} + f$ and hence xy yields the result in equation (1).

$$\begin{aligned} xy &= (a \cdot 10^{2n/3} + b \cdot 10^{n/3} + c)(d \cdot 10^{2n/3} + e \cdot 10^{n/3} + f) \\ &= ad \cdot 10^{4n/3} + ae \cdot 10^{n} + af \cdot 10^{2n-3} + bd \cdot 10^{n} + be \cdot 10^{2n/3} + \\ &\quad bf \cdot 10^{n/3} + cd \cdot 10^{2n/3} + cd \cdot 10^{n/3} + cf \\ &= ad \cdot 10^{4n/3} + (ae + bd)10^{n} + (af + be + cd)10^{2n/3} + \\ &\quad (bf + ce)10^{n/3} + cf \end{aligned} \tag{1}$$

   If we take into account the multiplication in 2, we can reduce the number of multiplications from 9 to 7.

$$(a + b + c)(d + e + f) = ad + ae + af + bd + be + bf + cd + ce + cf \tag{2}$$

   If we consider the five multiplications (3), (4), (5), (6), (7), then we can have $(af + bf + cd)$ from $(a+b+c)(d+e+f) - ad - e(a+c) - b(d+f) - cf$. And if we compute ae, then we can have ce from $e(a+c) - ae$. And if we compute bd, we can get bf from $b(d+f) - bd$.

$$(a + b + c)(d + e + f) \tag{3}$$
$$ad \tag{4}$$
$$cf \tag{5}$$
$$e(a + c) \tag{6}$$

$$b(d+f) \tag{7}$$

Hence, we have $a=1$, $b=2$, $c=3$, $d=4$, $e=5$ and $f=6$ and the operations required are shown in equations (8).

$a+b+c$	$1+2+3$	6
$d+e+f$	$4+5+6$	15
$(a+b+c)(d+e+f)$	$(6)(15)$	90
ad	$(1)(4)$	4
cf	$(3)(6)$	18
$e(a+c)$	$5(1+3)$	20
$b(d+f)$	$2(4+6)$	20
$af+be+cd$	$90-4-20-20-18$	28
ec	$(5)(3)$	15
bf	$(2)(6)$	12
$e(a+c)-ec$	$20-15$	5
$b(d+f)-bf$	$20-12$	8
$ae+bd$	$5+8$	13
$bf+ce$	$12+15$	27

(8)

Thefore, $xy = 4\cdot10^4+13\cdot10^3+28\cdot10^2+27\cdot10+18 = 40,000+13,000+2,800+270+18 = 53,000,+2,800+288 = 53,000+2,800+288 = 55,800+288 = 56088$

b) Determine the asymptotic running time of your algorithm. Would you rather split it into two parts or three parts?

Answer: For the technique of splitting into three parts and 7 multiplications we have the recurrence shown in equation (9), which from the Master Theorem we have $a=7>b^k=3^1$ and $T(n)$ is $\Theta(n^{\log_3 7}) \approx \Theta(n^{1.7712})$

$$T(n) = 7\cdot T\left(\frac{n}{3}\right)+n \tag{9}$$

For the technique of splitting into three parts and 6 multiplications we have the recurrence shown in equation (10), which from the Master Theorem we have $a=6>b^k=3^1$ and $T(n)$ is $\Theta(n^{\log_3 6}) \approx \Theta(n^{1.6309})$

$$T(n) = 6\cdot T\left(\frac{n}{3}\right)+n \tag{10}$$

Based on the above results, splitting into two parts is preferable than splitting into three due to the complexity and the overhead caused by controlling more multiplications and splits.

c) Suppose you could use only five multiplications instead of six. Then determine the asymptotic running time of such an algorithm. In this case, would you rather split it into two parts or three parts?

For the technique of splitting into three parts and 5 multiplications we have the recurrence shown in equation (11), which from the Master Theorem we have $a = 5 > b^k = 3^1$ and $T(n)$ is $\Theta(n^{\log_3 5}) \approx \Theta(n^{1.4649})$

$$T(n) \quad = \quad 5 \cdot T\left(\frac{n}{3}\right) + n \tag{11}$$

If we compare the powers 1.4649 and 1.5849, it might be preferable to use the algorithm which splits the numbers into 3 and uses only 5 multiplications, but to achieve only 5 multiplications we might need a lot of additions and subtractions which would increase the overhead in a way the algorithm is not practical. So in this case, I prefer to use the algorithm that splits into two.