CS 124, Problem Set 3

Jessica Wang

February 26, 2016

1) Solution

Some cycle of $r_{i_1,i_2}, r_{i_2,i_3}, ..., r_{i_{k-1,k}}, r_{i_k,i_1}$ represents a risk-free currency exchange if $r_{i_1,i_2} * r_{i_2,i_3} * ... * r_{i_{k-1,k}} * r_{i_k,i_1} > 1$. Thus $\frac{1}{r_{i_1,i_2}} * \frac{1}{r_{i_2,i_3}} * ... * \frac{1}{r_{i_{k-1,k}}} * \frac{1}{r_{i_k,i_1}} < 1$ You can take the log of both side $log(\frac{1}{r_{i_1,i_2}} * \frac{1}{r_{i_2,i_3}} * ... * \frac{1}{r_{i_{k-1,k}}} * \frac{1}{r_{i_k,i_1}}) < log(1)$. This inequality simplifies to $-log(r_{i_1,i_2}) - log(r_{i_2,i_3}) - ... - log(r_{i_{k-1,k}}) - log(r_{i_k,i_1}) < 0$. Create a complete graph with each vertex as a currency c_i and each edge between c_i and c_j have the weight $-log(r_{i,j})$. A risk-free currency exchange will occur whenever this graph has a cycle with an overall negative weight. To determine whether this negative cycle exists implement Bellman-Ford Algorithm. Bellman-Ford Algorithm will terminate whenever a negative cycle is found because it attempts to find the shortest path. If Bellman-Ford Algorithm terminates it indicates that a negative cycle exists within the graph and therefore there is a risk-free currency exchange. If Bellman-Ford Algorithm successfully finds the shortest path without terminating it indicates that a negative cycles does not exist and therefore there is not a risk-free currency exchange. In this method, using an iteration of Bellman-Ford Algorithm, we can detect whether there is a risk-free currency exchange.

2) Solution

This algorithm is based on DFS. Let e_1 and e_2 be the vertices on the edge e and w_e be the weight of edge e. Begin DFS on vertex e_1 and only traverse an edge if the edge's weight is less than w_e. Terminate the DFS if the algorithm ever reaches vertex e_2. If the algorithm terminates, it indicates that e is in a cycle of the graph and is the largest edge weight in the cycle. If the algorithm terminates by reaching e_2, then e is not contained within some MST. If the algorithm completes then e is in some MST. To prove correctness of the algorithm split up into the following cases:
Case 1 (e is not in a cycle): If e is not in a cycle there is only one path that can be used to include e_1 and e_2 in the MST. Therefore e must be in the MST. The algorithm will complete because since there is no cycle, there is no way for the DFS to reach e_2. Thus the algorithm works in this case.
Case 2(e is in a cycle, but not the heaviest weight): By the cycle property of MST's the edge in a cycle with the heaviest weight cannot be included in the MST, therefore e would be included in the MST. The algorithm will complete because since the largest weight in the cycle is greater than w_e, the algorithm will not transerve the largest weight edge, ending the cycle and preventing the algorithm from reaching e_2. Thus the algorithm will not terminate and complete, indicating that e is in the MST as it should.
Case 3(e is in a cycle and the heaviest weight): By the cycle propert of MST's the e would not be included in the MST because it is the heaviest weight in the cycle. The algorithm would terminate because it would traverse the other edges in the cycle, since the weight of those edges are all less than w_e and reach e_2 indicating a cycle with e as the largest weight. Thus the algorithm will terminate and indicate that e is not in the MST as it should.

3) Solution

Consider the set $X = \{1, 2, ..., 3^b\}$ for any $b > 3$ and consider the subsets as the following:
1: All $x \cong 0(mod3)$
2: All $x \cong 1(mod3)$
3: All $x \cong 2(mod3)$
4 through (4+b): Loop through i from 0 to b and each subset will have the elements from $3^{i-1} + 1$ to 3^i (ex. the second of these sets would contain the elements $\{4, 5, 6, 7, 8, 9\}$.)
The first three of these subsets will always be the most optimal solution for $b > 3$ because there will be b other subsets (4 through (4+b)). So call the first three subsets the optimal subsets and the remaining the suboptimal subsets. The greedy algorithm will always choose the suboptimal subsets because the biggest suboptimal subset will have size $3^i - 3^{i-1}$ and the optimal subsets have size 3^{i-1}. This is true because $3^i - 3^{i-1} = 3 * 3^{i-1} - 3^{i-1} = 2 * 3^{i-1} > 3^{i-1}$. The next subset that the greedy algorithm will choose will the next biggest suboptimal subset by the same reasoning when considering only the reamining unchoose elements. This can be telescoped to show that the greedy algorithm will always choose a suboptimal subsets, thus choosing b subsets overall while the optimal choice will always be the 3 optimal subsets. We have shown in the algorithm above that the set cover returned by the greedy algorithm is of size $b = \Omega(logn)$.
You can generalize this for other values of k by considering the set $X = \{1, 2, ..., k^b\}$ for any $b > k$ and consider the subsets of going through all mod values of k (similar to subsets 1-3 in the example above) and subsets containing numbers $k^{i-1} + 1$ to k^i for all $0 < i \leq b$ (similar to subsets 4 through (4+b) above).

4) Solution

First consider which sequence of j_i will result in the upperbound of time using the greedy algorithm. The upperbound in the run time will be caused by the biggest difference in run times between the two machines. In the greedy algorithm, the run time the difference between the run times of both machines will be caused by the final job added, because no matter what that job will be added to the machine with the current lowest run time. Thus the job with the biggest run time should be the very last job to be added and at the end of the sequence. When adding the last job, it would be best to add it to two machines who had as close to equal overall running times, becuase that will cause the last job to make the biggest difference between the final running times of both machines. Thus the upperbound in the run time will be caused by a sequence of j_i such that $j_1, j_2, ..., j_{n-1}$ are in the most optimal sequence while j_n is the job with the largest running time.
To show that the greedy algorithm runs within 3/2 factor of the optimal time let j_m represent the job with the largest running time, r_m. Split into the following cases:
Case 1 ($r_m \geq r_1 + r_2 + ... + r_n - r_m$): Since the largest run time r_m is greater than the sum of all the other run times, the optimal run time will have the lower bound of r_m because r_m is greater than the sum of all the run times divided by 2. When considering the upperbound of the run time sequence of jobs as described above, before adding the last job j_m with run time r_m the machines will run $j_1, j_2, ..., j_n$ j_m optimally. These jobs have a combined run time less than r_m thus running these optimally will have at most half the run time of r_m and will take at most $\frac{1}{2}r_m$ time to run. Adding the last job, j_m which takes r_m time to run will result in an overall run time which is bounded above by $\frac{1}{2}r_m + r_m = \frac{3}{2}r_m$. Case 2 ($r_m \geq r_1 + r_2 + ... + r_n - r_m$): Since the largest run time r_m is less than the sum of all the other run times, the optimal run time will be the sum of all the run times divided by 2. Let this time be *op*. When considering the upperbound of the run time sequence of jobs, as described above, before adding the last job j_m with run time r_m the machines will run $j_1, j_2, ..., j_n$ j_m optimally. Since all the jobs run optimally at time *op*, running all the jobs without j_m should run at an optimal time bounded by $op - \frac{r_m}{2}$. Adding the job j_m after this will

have a running time bounded above by $op - \frac{r_m}{2} + r_m = op + \frac{r_m}{2}$. But since in this case $r_m < op$ then $op + \frac{r_m}{2} < \frac{3}{2}op$.
An example of where the upperbound is achieved is using the sequence of run times of $\{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 10\}$. The optimal solution of this is having both machines have a run time of 10 resulting in an overall run time of 10. But ordering the sequence as $\{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 10\}$ will resulting in machine 1 have a run time of 15 and machine 2 have a run time of 5. $15 = \frac{3}{2} * 10$.
This can be generalize to show that with m machines the greedy algorithm will always return a run time that is within a factor of $\frac{m+1}{m}$ of the optimal run time. An family of examples that shows this upperbound of running time. is if there are $m + 1$ jobs with m having run times of 1 and one job with run time m. This will always have an optimal run time of m, but when the sequence is arranged so the job with run time m is at the end, the greedy algorithm will produce run time $m+1$. Therefore greedy algorithm will always be within a factor $\frac{m+1}{m}$ of the optimal running time.

5) Solution

Let $X = a10^{\frac{2n}{3}} + b10^{\frac{n}{3}} + c$ and $Y = d10^{\frac{2n}{3}} + e10^{\frac{n}{3}} + f$. Thus $XY = 10^{\frac{4n}{3}}(ad) + 10^n(bd + ae) + 10^{\frac{2n}{3}}(af + be + cd) + 10^{\frac{n}{3}}(bf + ce) + cf$. That currently takes 9 multiplications. This can be reduced by using the multiplications $(a+b+c)(d+e+f) = ad + (bd + ae) + (af + be + cd) + (bf + ce) + cf$, $(a + b)(d + e) = ad + (ae + bd) + be)$ and $(b + c)(e + f) = cf + (bf + ce) + be$. This allows us to only use the multiplications $(a + b + c)(d + e + f), (a + b)(d + e), (b + c)(e + f), ad, cf$ and be will allow us to us adding and subtracting to produce all nine multiplications.
The run time of this algorithm would have recurrance $T(n) = 6T(\frac{n}{3}) + O(n)$. Using Master's Theorem $T(n) = n^{log_3 6} \approx n^{1.63}$. Since splitting it into two parts has a run time approximatly of $n^{1.59}$ you would rather split the multiplication into two parts.
If you could do 5 multiplications instead of 6, the algorithm would have a recurrance of $T(n) = 5T(\frac{n}{3}) + O(n)$. By Master's Theorem $T(n) = n^{log_3 5} \approx n^{1.46}$. Since splitting it into two parts has a run time approximatly of $n^{1.59}$ you would rather split the multiplication into three parts in this case.