CME 305: Discrete Mathematics and Algorithms

Instructor: Reza Zadeh (rezab@stanford.edu)

HW#3 – Due at the beginning of class Thursday 03/02/17

1. Consider a model of a nonbipartite *undirected* graph in which two particles (starting at arbitrary positions) follow a random walk i.e. with each time step both particles uniform randomly move to one of the neighbors. Prove that the expected time until they collide is $O(n^6)$. A collision is when both particles are on the same node at the same time step.

   Solution: Let $G(V, E)$ denote the graph in question; we construct a new graph $H(W, F)$ in which an ordinary single particle random walk corresponds to the two particle random walk on G. Take $W = V \times V = \{(v_1, v_2) | v_1, v_2 \in V\}$ and $F = \{((v_1, v_2), (w_1, w_2)) | (v_1, w_1), (v_2, w_2) \in E\}$. The idea here is that a uniform random walk on H encodes the state of a two particle random walk on G (in the same way that a random walk on the path graph encodes the state of a drunk). Given a starting configuration $v = (v_1, v_2) \in W$ the expected time until the particles collide is bounded by the hitting time to the vertex $u = (v_1, v_1) \in W$. This hitting time is bounded by the cover time of H, i.e.

$$h_{vu} \leq C(H) = O(|W|^3) = O(n^6).$$

2. Let A be a $n \times n$ matrix, B a $n \times n$ matrix and C a $n \times n$ matrix. We want to design an algorithm that checks whether $AB = C$ without calculating the product AB. Provide a randomized algorithm that accomplishes this in $O(n^2)$ time with high probability.

   Solution: Pick $x \in \{0, 1\}^n$ such that $x_i = 1$ with probability $\frac{1}{2}$.

```
1. compute y = Bx, z = Ay, w = Cx
2. if z != w return false
3. return true
```

   First note that it takes $O(n^2)$ time to compute all the above matrix vector multiplications. Also note that we avoid roundoff error. Computing w and y involves no multiplication and there is no error in the computation of z assuming that AB can be computed exactly.

   Now, if $AB - C = 0$ the algorithm is always correct. So, assume $AB - C \neq 0$. We compute the probability that the algorithm returns true,

$$P(z = w) = P(ABx = Cx) = P((AB - C)x = 0).$$

Let $D = AB - C$ and let d_i be the i^{th} row of D. We have that that

$$\begin{aligned} P(Dx = 0) &\leq P(d_i^T x = 0) \\ &= P(\sum_j d_{ij} x_j = 0) \\ &\leq P(\sum_j d_{ij} x_j = 0 | x_2, \ldots, x_n) \\ &= \frac{1}{2} \end{aligned}$$

where the last step comes from the fact that we are only free to pick x_1 which has probability of $\frac{1}{2}$ regardless which value we pick. Thus the probability we make a mistake is $P(z = w) \leq \frac{1}{2}$. Repeating the algorithm some constant k times (say 10) we bound the probability of error by $\frac{1}{2^k} \approx 0.001$.

3. Given a connected, undirected graph $G = (V, E)$ and a set of terminals $S \subseteq V$, $S = \{s_1, s_2, \ldots, s_k\}$, a multiway cut is a set of edges whose removal disconnects the terminals from each other. The multiway cut problem asks for the minimum weight such set. The problem of finding a minimum weight multiway cut is NP-hard for any fixed $k \geq 3$. Observe that the case $k = 2$ is precisely the minimum $s - t$ cut problem.

   Define an *isolating cut* for s_i to be a set of edges whose removal disconnects s_i from the rest of the terminals. Consider the following algorithm

   - For each $i = 1, \ldots, k$, compute a minimum weight isolating cut for s_i, say C_i.
   - Discard the heaviest of these cuts, and output the union of the rest, say C.

   Each computation in Step 1 can be accomplished by identifying the *terminals* in $S - \{s_i\}$ into a single node, and finding a minimum cut separating this node from s_i; this takes one max-flow computation. Clearly, removing C from the graph disconnects every pair of terminals, and so is a multiway cut.

   (a) Prove that this algorithm achieves a $2 - 2/k$ approximation.

   Solution: This questions is from Vijay Vazirani's approximation algorithms book. Let A be an optimal multiway cut in G. We can view A as the union of k cuts as follows: The removal of A from G will create k connected components, each having one terminal (since A is a minimum weight multiway cut, no more than k components will be created). Let A_i be the cut separating the component containing $s_i$from the rest of the graph. Then $A = \cup_{i=1}^k A_i$. Since each edge of A is incident at two of these components, each edge will be in two of the cuts A_i. Hence,

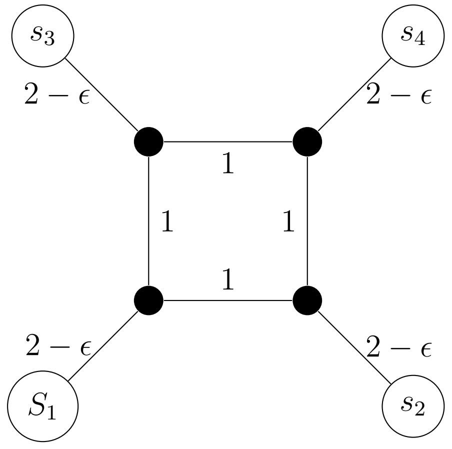
$$\sum_{i=1}^{k} w(A_i) = 2w(A)$$

   Clearly, A_i is an isolating cut for s_i. Since C_i is a minimum weight isolating cut for s_i, $w(C_i) \leq w(A_i)$. Notice that this already gives a factor 2 algorithm, by taking the union of all k cuts C_i. Finally, since C is obtained by discarding the heaviest of the cuts C_i,

$$w(C) \leq (1 - 1/k) \sum_{i=1}^{k} w(C_i) \leq (1 - 1/k) \sum_{i=1}^{k} w(A_i) = 2(1 - 1/k)w(A)$$

(b) Prove that this analysis is tight by providing an example graph where the approximation bound is exactly achieved.

Solution: Consider for $k = 4$.



For each terminal s_i, the minimum weight isolating cuts for s_i is given by the edge incident to s_i. So, the cut C returned by the algorithm has weight $(k-1)(2-\epsilon)$. On the other hand, the optimal multiway cut is given by the cycle edges, and has weight k.

4. Consider variants on the metric TSP problem in which the object is to find a simple path containing all the vertices of the graph. Three different problems arise, depending on the number (0, 1, or 2) of endpoints of the path that are specified. If zero or one endpoints are specified, obtain a 3/2 factor algorithm.

   Hint. Consider modifying Christofides algorithm for metric TSP.

   Solution: We prove the 1 endpoint case, and the zero endpoints will follow by arbitrarily selecting an endpoint. Call the single endpoint v.

   Construct a minimum spanning tree T of G. Determine the set of odd degree vertices, call it S. There will be an even number of them, i.e. $|S| =$ even. v may or may not be in S. If v is in S, take it out, along with another arbitrary vertex from S. If v is not in S, don't do anything.

   Find the minimum matching M on S, and add it to T, call the result G'. Notice that in G', all nodes have even degree, except for v and some other node. Thus we can find an eulerian path and shortcut it to obtain a hamiltonian path. It remains to bound the weight of G'.

Notice that M is a matching on at most $n-1 \geq |S|$ nodes, with this bound being tight only when n is odd and $v \notin S$, otherwise the bound is strict. Also notice that the optimal hamiltonian path P^* contains two edge-disjoint matchings on $n-1$ and $n-2$ nodes (just alternate edges along the path). Thus the weight of a minimum matching on S is at most half the weight of P^*, the optimal path.

Since P^* is a spanning tree, we also have the weight of T is less than P^*. Thus the union of the path and matching have at most 3/2 the weight of P^*.

5. Recall the *minimum vertex cover* problem: given a graph $G(V, E)$ find a subset $S^* \subseteq V$ with minimum cardinality such that every edge in E has at least one endpoint in S^*. Consider the following greedy algorithm. Find the highest degree vertex, add it to the vertex cover S and remove it along with all incident edges. Repeat iteratively. Prove that this algorithm has an unbounded approximation factor i.e. for any c there exists an input graph G such that $|S| \geq c$ OPT.

   Solution: Consider a bipartite graph with partition (A, B). Let $|A| = n$ and partition B into n disjoint sets $\{B_i\}_1^n$ with $|B_i| = \lfloor n/i \rfloor$. Then $|B| = n + \lfloor n/2 \rfloor + \lfloor n/3 \rfloor + \cdots + 1 = O(n \log n)$. For each vertex $a \in A$ place an edge to a vertex $b \in B_k$ such that each vertex in B_k has degree at least k. Repeat this for $k = 1, 2, \ldots, n$ and note that this construction is possible since each $|B_k| \leq n/k$.

   Clearly, OPT for this graph is just $|A| = n$. But the algorithm will (depending on how degree ties are broken) remove all vertices in B. The approximation ratio is ALG/OPT $= O(n \log(n))/n = O(\log(n))$ Since the ratio depends on n it is unbounded.

6. A dominating set of a graph is a subset of vertices such that every node in the graph is either in the set or adjacent to a member of the set. The DOMINATING-SET problem is as follows: given a graph G and a number k, determine if G contains a dominating set of size k or less.

   (a) Show the DOMINATING-SET problem is NP-complete.

   Solution: We will show that this problem is NP-Complete by reduction to VERTEX COVER. Assume we have a black box for solving dominating set instances in polynomial time, and let G be a graph and k a number. We will show that we can tell if G contains a vertex cover of k or fewer nodes in polynomial time.

   A problem is NP-complete if it is both NP and NP-hard. The DOMINATING SET problem is clearly in NP, as given a set S, a graph G, and a number k we can test if S is a dominating set of G of size k or less by first checking if its cardinality is less than or equal to k and then checking if every node in G is either in S or adjacent to a node in S. This process clearly takes polynomial time. We now show the problem is NP-hard.

   First, note that we can limit ourselves to the case where G is connected, as if G disconnected, we can break G into its connected components $G_1 ... G_r$ and compute for each of these connected subgraphs the minimum k_i such that there exists a vertex cover in the subgraph. We can compute k_i by simply performing a binary search for it. Adding the obtained covers together gives the minimum vertex cover of G. This is obviously a vertex cover, and it is minimum since any smaller vertex cover must include fewer vertices from some connected subgraph of G G_i. This contradicts the minimality of the cover of G_i we added to the cover of G. Thus we can safely assume that G is connected in our reduction.

Create a new graph G' from G as follows: for every node u in G, create a corresponding node u in G'. Further, for every edge (u, v) in G, create a corresponding node w_{uv} in G', and add the edges (u, w_{uv}), (v, w_{uv}), and (u, v). This graph has $m+n$ nodes, where n is the number of nodes in G and m is the number of edges. We will show that for any number p, G has a vertex cover of p nodes if and only if G' has a dominating set of p nodes.

We begin with the forward direction. Let G have a vertex cover of p nodes, and call it S. Then for every edge (u, v) in G, S contains either u or v, by definition. Now, when we built G', we took every node of G and created a copy of it in G'. Let $S' \subseteq G'$ be the set of all the nodes which correspond to the nodes of S in G. This is a set of size p, and is a dominating set in G': for every node corresponding to a node u S' contains either u or some neighbor of u (this neighbor exists since G is connected by assumption), and for any node corresponding to an edge w_{uv} S' contains either u or v by the construction of S– by the definition of G' one of these nodes neighbors w_{uv}. Thus, if G has a vertex cover of size p G' has a dominating set of size p.

We now prove the backwards direction. Let G' have a dominating set S' of p nodes. We construct a cover of G as follows: for every node of the form w_{uv} in S' (that is, for every node corresponding to an edge of G), add the node u or v to a set S. For every node z corresponding to a node of G, add it to S. This gives a set of size p in G, which is a vertex cover in G: for every edge (u, v) in G, w_{uv} was created in G'. This node can only be dominated by u or v, and so S' clearly must contain either u, v, or w_{uv}. Since we formed S by taking all of the nodes of S' and converting the nodes of the form w_{uv} to nodes of the form u, for every edge in G, S contains either u or v, and thus S is a vertex cover of size p of G. So if G' has a dominating set of size p, G has a vertex cover of size p.

With this, we see that G contains a vertex cover of k nodes if and only if G' contains a dominating set of k nodes. Thus, to tell if G has a vertex cover of k nodes or less, we form G' and apply our black box to it to see if it has a dominating set of k or fewer nodes. Since G' has $m+n = O(n^2)$ nodes and our black box runs in polynomial time, doing this on G' takes polynomial time. By our above proof, this algorithm is guaranteed to return YES if and only if G actually contains a vertex cover of k or fewer nodes. Thus, our reduction is complete: a polynomial time algorithm for DOMINATING SET implies a polynomial time algorithm for vertex cover, and thus DOMINATING SET is NP-complete.

(b) Obtain a $\ln(n)$-approximation to the DOMINATING-SET problem.

Solution: We will show that a simple greedy approach for this problem achieves the desired approximation ratio. Our algorithm is as follows: while there are still nodes in the graph, choose the node of highest degree, add it to our approximate dominating set S, and remove it and all of its neighbors from G. This clearly runs in polynomial time (we do at most n iterations of our loop, which checks the degree once and deletes at most n nodes), and returns a valid dominating set: a node is only deleted when it or a neighbor of it was added to S, and so since every node is eventually deleted, every node is either in S or adjacent to a node in S. We will show that this gives a $\ln(n)$-approximation for our problem.

Assume that G has n nodes and the minimum dominating set of G has size k. Call this sets S. Clearly, S must contain a vertex of degree at least $n/k - 1$, as if it didn't S could only dominate strictly fewer than $k(n/k - 1 + 1) = n$ nodes: it dominates k nodes by containing them and strictly less than $k(n/k - 1) = n - k$ of them by neighboring them. Thus, when we add the vertex of highest degree to S, we added a node with degree at least $n/k - 1$, and when we delete it an its neighbors from G, we delete at least n/k nodes. So after adding this one node of G to our set we get a new graph of $n - n/k = n(1-1/k)$ nodes to cover. Repeating this same argument on this new graph (utilizing the fact that at most k nodes from S cover it and so one of the nodes has degree at least a $1/k$-fraction of the number of remaining nodes minus 1) and recursing gives us that after adding r nodes to S, we will be left with a graph with at most $n(1 - 1/k)^r$ nodes. If we chooose $r = kln(n)$, we have a graph with $n(1-1/k)^{kln(n)} < ne^{-ln(n)} = n/n = 1$ nodes, and as the number of nodes must be an integer, we conclude that the graph at this point is empty. So, as our algorithm terminates after adding at most $kln(n) = ln(n)OPT$ nodes to S, we have that this algorithm is a $\ln(n)$ approximation to the DOMINATING SET problem, as desired.

7. An oriented incidence matrix B of a directed graph $G(V, E)$ is a matrix with $n = |V|$ rows and $m = |E|$ columns with entry B_{ve} equal to 1 if edge e enters vertex v and -1 if it leaves vertex v. Let $M = BB^T$.

   (a) Prove that $rank(M) = n - w$ where w is the number of connected components of G.

   (b) Show that for any $i \in \{1, \ldots, n\}$,

   $$\det M_{ii} = \sum_N (\det N)^2,$$

   where $M_{ii} = M \backslash \{i^{th} \text{ row and column}\}$, and N runs over all $(n-1) \times (n-1)$ submatrices of $B \backslash \{i^{th} \text{ row}\}$. Note that each submatrix N corresponds to a choice of $n - 1$ edges of G.

   (c) Show that

   $$\det N = \begin{cases} \pm 1 & \text{if edges form a tree} \\ 0 & \text{otherwise} \end{cases}$$

This implies that $t(G) = \det M_{ii}$, where $t(G)$ is the number of spanning trees of G. In this definition of a tree, we treat a directed edge as an undirected one.

(d) Show that for the complete graph on n vertices K_n,

$$\det M_{ii} = n^{n-2}.$$

Solution:

(a) Consider that for a graph with k connected components, we can transform L into a block diagonal matrix by relabeling the vertices of the graph such that each block corresponds to a single connected component. It is therefore sufficient to focus our analysis on a single connected component.

Next we claim Kernel(L) = Kernel$(B^T B)$ = Kernel$(\mathbb{1})$. Suppose $Bx = 0$ then we have $B^T Bx = 0$ and $Lx = 0$. Hence x in the kernel of B implies x is also in the kernel of L.

Suppose $Lx = 0$ then we have $x^T Lx = 0$ so $x^T B^T Bx = \|Bx\|_2^2 = 0$ which only holds if $Bx = 0$. Hence x in the kernel of L implies x is also in the kernel of B.

Lastly we have:

$$\begin{aligned} \|Bx\|_2^2 = 0 \;&\Leftrightarrow\; \sum_{(i,j)\in E} (x_i - x_j)^2 = 0 \\ &\Leftrightarrow\; \forall (i,j) \in E: \; x_i = x_j \\ &\Leftrightarrow\; x \in \text{Kernel}(\mathbb{1}) \end{aligned}$$

(b) Given an $r \times q$ matrix C and an $q \times r$ matrix D, the Cauchy-Binet formula states that $det(CD) = \sum_S det(C_S)det(D_S)$ where the sum is over all possible size r subsets S of $\{1, 2, \ldots, q\}$. Note that $M_{ii} = B_i B_i^T$ and apply the Cauchy-Binet formula with $r = n-1, q = m, C = B_i$ and $D = B_i^T$. The identity $det(A) = det(A^T)$ then yields the final result. A proof of the Cauchy-Binet formula can be found at the following link. `http://www.lacim.uqam.ca/~lauve/courses/su2005-550/BS3.pdf`

(c) Given N consider the subgraph of G induced by the choice of edges specified by N. Assume that this subgraph contains a cycle. Furthermore, for now, assume this cycle is directed. Index the vertices of the cycle with set $I = \{i_1, i_2, \ldots, i_k\}$. Define a n $n-1$ vector c with ones at indices of I and zeros elsewhere. Notice that $u = Nc = 0$. This is true since any jth entry of u is the sum of $+1$ and -1 indicating a cycle edge entering vertex j and leaving it respectively. This implies that the columns of N are linearly dependent and thus $det(N) = 0$.

For a cycle that is not directed note that we can set the entries in the vector c from $\{-1, 1\}$ and reverse the sign of any column and in doing so flip the direction of any edge in the cycle. Also, note that the removed row does not ruin the result since the corresponding entry is never in $u = Nc$.

Now assume we have a tree. First use the following procedure to create a sequence $\{i_1, i_2, \ldots, i_{n-1}\}$. Pick any leaf i_1 and remove it along with an incident edge e_{i_1}.

Then a leaf i_2 with incident edge e_{i_2} , etc... . At each step we still have a tree but with one less vertex. Continue the procedure until all edges of the tree have been enumerated. Now construct a permutation matrix P to to rearrange the rows and columns of N in the same order as they were removed i.e. send vertex i_k to row k and edge e_{i_k} to column k. Note that PNP is lower triangular since $i_k \notin e_{i_r}$ for $k < r$ (otherwise i_k would not be a leaf at kth step). Furthermore all entries on diagonal are ± 1 and so $det(N) = det(PNP) = \pm 1$.

Lastly, recall that a tree always has at least two leaves and so we can avoid picking the vertex corresponding to the removed row during this procedure.

(d) Since $det(M_{ii})$ is the number of spanning trees the total number of trees on n vertices will be given by the complete graph K_n since any possible set of edges may be selected.

Consider the diagonal entry of M_{ii}. A diagonal entry is the degree of vertex j which is $n-1$ for a complete graph. An off-diagonal entry is a dot product between rows corresponding to distinct vertices. Since we have a complete graph, we have an edge between them and so the dot product is $-1 \cdot 1 = -1$.

We can then write M_{ii} in form $nI - ee^T$ where e is a vector of all ones. The determinant can be computed using the ShermanMorrison formula as follows.

$$det(M_{ii} = (1 - \frac{e^T I e}{n} det(nI) = (1 - \frac{n-1}{n} n^{n-1} = n^{n-2}$$

8. Given an associative expression, we would like to evaluate it in some order which is *optimal.* Suppose we have as input n matrices $A_1, A_2, \ldots, A_n$ where for $1 \leq i \leq n$, A_i is a $p_{i-1} \times p_i$ matrix. Parenthesize the product $A_1 A_2 \ldots A_n$ so as to minimize the total cost. Assume that the cost of multiplying a $(p_{i-1} \times p_i)$ matrix by a $(p_i \times p_{i+1})$ matrix is $p_{i-1} \times p_i \times p_{i+1}$ flops.

Note that the algorithm does not perform the actual multiplications. It just determines the best order in which to perform the multiplication operations and returns the corresponding cost. Come up with a dynamic programming formulation which takes as input matrices $A_1, \ldots, A_n$ and returns as output a single number describing the optimal cost of multiplying the n matrices together. Your algorithm should require at most $O(n^3)$ work, where n denotes the number of input matrices.

Solution: We first define our sub-problems. Given a sequence of matrices in a matrix-multiply $A_0 \cdot A_1 \cdot \ldots \cdot A_{n-1}$, what feature of the optimal solution would we like to guess? Can't know the whole solution, since there are exponentially many different ways to group the operations. We might consider, "what's the last operation we do?" That is, guess the outer-most / last multiplication. For example, we are faced with multiplying $(A_i \cdot \ldots \cdot A_{k-1})$ with $(A_k \cdot \ldots \cdot A_{j-1})$. Here we see that we are not dealing with only a prefix or only a suffix anymore. We are dealing with a sub-string! The number of possible choices for k is $O(j - i + 1) = O(n)$.

We now define our recurrence,

$$\mathrm{DP}(i,j) = \min_{k \in i+1,\ldots,j-1} \left(DP(i,k) + DP(k,j) + [\text{cost of product of } A[i:k] \cdot A[k:j]] \right).$$

When we are asked to solve $DP(i,j)$ we need to know $DP(i,k)$ and $DP(k,j)$. Hence this is a bottom-up solution.

Consider the work required for each sub-problem. It takes constant time for each iteration of our minimization for-loop (i.e. when checking whether we have a min candidate for each $k \in i+1,\ldots,j-1$) since we have two free DP recursive (each constant time) and we calculate the theoretical cost of multiplying $A[i:k]$ with $A[k:j]$, which can be done in constant time. Hence $O(n)$ cost across for one minimization, i.e. $O(n)$ time per sub-problem.

We also consider the topological ordering: we go by increasing sub-string size. Total run-time: number of sub-problems $\times$ time per sub-problem is $O(n^2) \cdot O(n) = O(n^3)$. Note here n is the number of matrices we multiply, which hopefully is much smaller than the size of the matrices, hence allowing us to save time overall.

9. Prove that if G is connected, regular, and has an odd number of nodes, then G is Eularian.

   Solution: G Eularian $\iff$ all nodes in G have even degree. Since G is regular, all nodes have same degree. Assume toward contradiction all nodes have odd-degree. Since there are an odd # of nodes, this yields a contradiction to the handshake lemma. Hence if G regular and has an odd # of nodes, all nodes must have even degree. Hence G Eularian.