CS 124 Homework 7: Spring 2021

Your name: Sean Ty

Collaborators:

No. of late days used on previous psets: 2

No. of late days used after including this pset: 2

Homework is due Friday 2021-04-30 at 11:59pm ET. You are allowed up to **twelve** (college)/**forty** (extension school) late days through the semester, but the number of late days you take on each assignment must be a nonnegative integer at most **two** (college)/**four** (extension school).

Try to make your answers as clear and concise as possible; style will count in your grades. Be sure to read and know the collaboration policy in the course syllabus. Assignments must be submitted in pdf format on Gradescope. If you do assignments by hand, you will need to scan in your results to turn them in.

For all homework problems where you are asked to design or give an algorithm, you must prove the correctness of your algorithm and prove the best upper bound that you can give for the running time. Generally better running times will get better credit; generally exponential time algorithms (unless specifically asked for) will receive no or little credit. You should always write a clear informal description of your algorithm in English. You may also write pseudocode if you feel your informal explanation requires more precision and detail, but keep in mind pseudocode does NOT substitute for an explanation. Answers that consist solely of pseudocode will receive little or not credit. Again, try to make your answers clear and concise.

1. (a) **(5 points)** We know that that all NP-complete problems reduce to each other. It would be nice if this meant that an approximation for one NP-hard problem would lead to another, but this is not the case. Consider the case of Minimum Vertex Cover, for which we have a 2-approximation; that is, we can find a vertex cover of size within a factor of 2 of optimal. A set C is a vertex cover in a graph $G = (V,E)$ if and only if $V - C$ is an independent set in V. Explain why this does not yield an approximation algorithm that is within a constant factor of optimal for Maximum Independent Set. That is, show that for every constant $c > 1$, there exists a graph and a 2-approximation of its Minimum Vertex Cover such that the corresponding independent set is not within a factor of c of the Maximum Independent Set.

   Consider a graph of n vertices that is just a cycle (e.g. looks a regular n-gon), for which the minimum vertex cover size is $\lceil n/2 \rceil$ while the maximum independent set size is $\lfloor n/2 \rfloor$. On the other hand, the approximation algorithm for vertex cover could give us $n-1$ vertices (and this is at most $2\lceil n/2 \rceil$ so this is valid), so the maximal independent set here has size 1, so we are off by a factor of $\lfloor n/2 \rfloor$, and we can take n arbitrarily large to give an example for any c (say $n = \lceil 69c \rceil$).

   (b) **(10 points)** Prove that it's NP-hard to approximate the size of the maximum independent set in a graph to within 124 vertices.

   (**Hint:** Reduce from the problem of finding a maximum-size independent set in a graph G, which is NP-hard. Consider a graph G' that's many disjoint copies of G (that is, no edges between the copies).)

   We first reduce this problem to finding the size of the maximum independent set. To do this, suppose we have an algorithm A that obtains an approximation for max independent set size within 124 vertices on a graph $G = (V,E)$. Then consider the graph $G' = (V',E')$, which consists of 6969 pairwise disjoint copies of G (i.e. no edges between copies, and the edges between vertices of each G are the same; call them $G_1, G_2, \ldots G_{6969}$ for convenience). Then suppose we run A on G' to output an independent set $S \subseteq V'$. Partition $S = \bigcup_{i=1}^{6969} S_i$ such that S_i consists of only all vertices of S that are contained in G_i. Note that this construction is in polynomial time since all we are doing is making a fixed number of copies of G (so it is in fact linear).

   Let M be a max independent set of G' and partition it into $M_1, M_2, \ldots M_{6969}$ just as the S's (so M_i contains only vertices in G_i that are in M). Note that since the S_i's are independent sets, it follows that $|M_i| \geq |S_i|$ for all i (also clearly M_i would be a max independent set in each G_i). Likewise, $|M| \geq |S|$.

   By assumption on A, it follows that $|M| - |S| \leq 124$. Now suppose for the sake of contradiction that for all $1 \leq i \leq 6969$, we have $|M_i| - |S_i| \geq 1$. Then by summing all these inequalities, we get $\sum_{i=1}^{6969} |M_i| - |S_i| \geq 6969 > 124$, which implies $|M| - |S| > 124$, contradiction to our original assumption. It follows that for some $1 \leq i \leq 6969$, we have $|M_i| - |S_i| = 0$. Now consider an algorithm A' that outputs the size of the largest S_i (or the exact set, doesn't really matter). It would output S_i since by definition this is the largest. This would mean that for graph G_i, we have an algorithm to exactly find the size of the max independent set. But we know this problem is NP-hard, so the conclusion follows.

   (c) **(15 points)** Prove that if there exists a polynomial time algorithm for approximating the maximum independent set in a graph G to within a factor of 2, then for every $\varepsilon > 0$, there is a polynomial time algorithm for approximating the maximum independent set in a graph to within a factor of $(1+\varepsilon)$. The degree of the polynomial may depend on ε.

   (**Hint:** for a starting graph $G = (V,E)$, consider the graph $G \times G = (V',E')$, where the vertex set V' of $G \times G$ is the set of ordered pairs $V' = V \times V$, and $\{(u,v),(w,x)\} \in E'$ if and only if

$$\{u,w\} \in E \text{ or } \{v,x\} \in E.$$

   If G has an independent set of size k, then how large an independent set does G' have?)

   We first have the following claim:

Claim: If there exists a polynomial time algorithm to approximate the max independent set size within a factor of k, then there exists a polynomial time algorithm to approximate within a factor of $\sqrt{k}$.

Proof: Let $G = (V,E)$ be a graph with a max independent set size of n, and construct the graph $G' = (V',E')$ as in the hint. Note that this construction takes polynomial time, since we have $O(|V|^2)$ vertices and $O(|E|^2)$ edges. Now let the maximum independent set size in G' have cardinality m. Then by our algorithm, we can find an independent set $S \subseteq V'$ such that $|S| \geq \frac{m}{k}$. Also, note that $m \geq n^2$, since if T is an independent set of size n in G, then $T' = \{(v,w)|v,w \in T\}$ is an independent set of size n^2 in G since if two vertices in $(v,w),(v',w') \in T'$ have an edge, then $\{v,v'\}$ or $\{w,w'\}$ is in E, contradiction to T being an independent set. Thus, we have $|S| \geq \frac{m}{k} \geq \frac{n^2}{k}$.

Let $S_1 = \{v|(v,w) \in S \text{ for some } w \in V\}$ and $S_2 = \{w|(v,w) \in S \text{ for some } v \in V\}$. Note that if $v_1, v_2 \in S_1$, then there exist $w_1, w_2 \in V$ such that $(v_1,w_1) \in S, (v_2,w_2) \in S \implies \{(v_1,w_1),(v_2,w_2)\} \notin E' \implies \{v_1,v_2\} \notin E$ by logic. It follows that S_1 is an independent set. Similarly, S_2 is an independent set. Our algorithm would output whichever of S_1, S_2 is larger (in cardinality). If there are ties, output any of them, say S_1.

Now note that by definition, $S \subseteq \{(v,w)|v \in S_1, w \in S_2\}$. Thus, $|S| \leq |S_1||S_2| \implies \max(|S_1|,|S_2|) \geq \frac{n}{\sqrt{k}}$ from the inequality earlier. This gives us an independent set of G that is within $\sqrt{k}$ of the optimal. Since the construction of S_1, S_2 are polynomial in $|S|$ (it is $O(|S|)$ and $|S|$ is polynomial in $|V|, |E|$ (it is quadratic), this algorithm also takes polynomial time, which proves the claim.

Back to the main problem, it follows that we can inductively continue this process to find a polynomial time algorithm to approximate the maximum independent set size to within a factor of $\sqrt[2^m]{2}$ for any nonnegative integer m (since we are inductively doing this finitely many times). We can then choose m large enough, say $m = 6969 \cdot \left\lceil \log_2 \left(\frac{\log 2}{\log(1+\varepsilon)} \right) \right\rceil + 1337$,[1] to obtain $2^m \log(1+\varepsilon) > \log 2$, or $\sqrt[2^m]{2} < 1+\varepsilon$. The desired conclusion then follows.

[1] For a tighter bound, just $\left\lceil \log_2 \left(\frac{\log 2}{\log(1+\varepsilon)} \right) \right\rceil + 1$ probably works, but where's the fun in that? don't dock plz

2. Consider the problem MAX-k-CUT, which is like the MAX CUT algorithm, except that we partition the vertices into k disjoint sets, and we want to maximize the number of edges between sets.

   (a) **(10 points)** Give a deterministic algorithm that finds a partition within a factor of $1-\frac{1}{k}$ of optimal.

   Let $G=(V,E)$ be the graph and say that the vertices are labeled as $v_1, v_2, \ldots v_n$. Let $S_1, S_2, \ldots S_k$ be our partition sets for V. Call a *cross edge* to be an edge whose endpoints are in different sets. We perform the following algorithm: first place all vertices in some set, say S_1. Then in each step, if we can increase the number of cross edges by moving a vertex v_i to some other set S_j, we perform that move.[2] The algorithm terminates once it is not possible to increase the number of cross edges.

   First note that the algorithm must terminate. This is because in each step, the number of cross edges strictly increases. Clearly the number of cross edges can be at most $|E|$–in particular, it must be bounded. Then, since it increases by at least 1 in each move, it follows that we can only move finitely many times, so the algorithm must terminate at some point.

   Now consider the state when the algorithm terminates. For convenience, for each $v \in V$ let $S(v)$ denote the set S_i it is in, and let $f(v,S_i)$ be the number of neighbors v has in S_i. Then we claim that at this state we must have $f(v,S_i) \geq f(v,S(v))$ for all i such that $S_i \neq S(v)$. To see this, assume the contrary. Then by moving v from $S(v)$ to S_i, the status (cross or not) of edges unrelated to v clearly don't change. The cross edges from v with endpoints in other S_j's (i.e. $S_j \neq S_i, S(v)$) still remain cross edges. Thus, the only edges that change in status are edges counted in $f(v,S(v))$ and $f(v,S_i)$. In particular, the number of cross edges is increased by $f(v,S_i) - f(v,S(v)) > 0$. This is a contradiction, since then the algorithm would not have terminated in this state. The claim follows.

   From the claim, it follows that $f(v,S_i) \geq f(v,S(v))$ for all $S_i \neq S(v)$. Adding these inequalities with the equality $f(v,S(v)) = f(v,S(v))$, it follows that

$$\sum_{i=1}^{k} f(v,S_i) \geq k f(v,S(v)) \implies f(v,S(v)) \leq \frac{\delta(v)}{k},$$

   where $\delta(v)$ denotes the degree of v.[3]

   Finally, we count the number of cross edges. By definition (using the same trick as in lecture), this is

$$\frac{1}{2}\sum_{i=1}^{k}\sum_{v\in S_i} |\{w|(v,w)\in E, w\notin S_i\}| = \frac{1}{2}\sum_{i=1}^{k}(\delta(v) - f(v,S(v))) \geq \frac{1}{2}\sum_{i=1}^{k}\sum_{v\in S_i}\left(1-\frac{1}{k}\right)\delta(v),$$

   by setting each endpoint of each edge a count of $\frac{1}{2}$ and using this to count the number of edges. The first equality above comes from the definition of $f(v,S(v))$, and the inequality comes from the bound proven earlier. Now note that the last quantity we have is just the number of edges in the graph using the same double counting scheme earlier, since $\sum_{i=1}^{k}\sum_{v\in S_i}\delta(v) = \sum_{v\in V}\delta(v) = 2|E|$. It follows that the quantity is at most $\left(1-\frac{1}{k}\right)|E|$. Now clearly the optimal number of cross edges is also $|E|$ since this is the maximum number of edges, so our algorithm's partition of $\{S_i\}_{i=1}^{k}$ does indeed give a cross edge count within a factor of $1-\frac{1}{k}$ of the optimal, as desired.

   (b) **(5 points)** Give a randomized algorithm that's a generalization of the randomized algorithm for MAX CUT from class that finds a partition that, in expectation, is within a factor of $1-\frac{1}{k}$ of optimal.

   Use the same notation as earlier. Consider i.i.d. discrete uniform random variables $U_1, U_2, \ldots U_n$ with support $\{1,2,\ldots k\}$ (i.e. so each element here has probability $1/k$ of being chosen). Then consider sets

[2] We are effectively performing hill climbing, where a neighbor here is a solution where one vertex moves from one set to another.

[3] This follows from the fact that the degree of v is the number of neighbors, which is the sum of the number of neighbors it has in each set since the S_i's are a partition of V.

$S_1, S_2, \ldots S_k$ that are constructed such that vertex v_i is placed in S_j iff $U_i = j$. Then note that for each edge, the probability that it connects two vertices from different sets S_i is $\frac{k-1}{k}$ since if the first vertex is in some set, then the second has $k-1$ of k possibilities). Hence, the expected number of edges that cross between sets is $\frac{k-1}{k}|E|$, and since the most number of such crossing edges is clearly $|E|$, it follows that the optimal is also at most $|E|$. So we are within a factor of $\frac{k-1}{k} = 1 - \frac{1}{k}$ of the optimal, as desired.

3. We consider the following scheduling problem, similar to one that we studied before: we have two machines, and a set of jobs $j_1, j_2, j_3, \ldots, j_n$ that we have to process. We place a subset of the jobs on each 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 completion time, sometimes called the *makespan*, which is the maximum load over all machines.

   Consider the following local search algorithm. Start with any arbitrary assignment of jobs to machines. We then repeatedly *swap* a single job from one machine to another, if that swap will *strictly reduce* the completion time. (We won't make a move if the completion time stays the same, and only one job moves in each swap.) If a swap is not possible, we are in a stable state. For example, suppose we had jobs with running times 1,2,3,4, and 5, and we started with the jobs with running times 1,2, and 3 on machine 1, and the jobs with running times 4 and 5 on machine 2. This is a stable state, but it is not optimal; the minimum possible completion time is 8, and this stable state has completion time 9.

   (a) **(5 points)** Prove that the local search algorithm eventually terminates in a stable state (as opposed to running forever).

   Let each job j_i have run time r_i. Then note that there are at most 2^n possible allocations of jobs, since for each job we choose if it will be placed in machine 1 or 2. In particular, there are finitely many possible allocations of jobs, which means there are finitely many possible completion times since each allocation gives rise to exactly one completion time. Now since the completion time strictly decreases after each swap, it follows that there can only be finitely many swaps, and so the algorithm must terminate. It must terminate in a stable state since if didn't, then another swap could have been performed. The desired conclusion follows.

   (b) **(15 points)** Prove that for any assignment on which the local search algorithm terminates, the completion time is within a factor of 4/3 of the optimal.

   (**Hint:** One approach is to prove by contradiction. Suppose that you reached a stable state whose completion time was not within a factor of 4/3 of the optimal. What can you derive from this assumption?)

   Suppose that the optimal completion time is s and that the algorithm terminated with a completion time of $t > \frac{4}{3}s$. WLOG let machine 1 be the machine with load t. Since the optimal is s, it follows that we the total load must be at most $2s$, since otherwise the optimal would be greater than s. Then machine 2 would have load at most $2s - t$.

   Now note that machine 1 must have at least 2 jobs. If not, then there is a job that takes $t > s$ time, which means the optimal s should be t, contradiction. Thus, there must be a job in machine 1 with run time at most $t/2$. By moving this job to machine 2, its completion time becomes $2s - \frac{t}{2} < 2s - \frac{2s}{3} = \frac{4}{3}s < t$. But machine 1 would also have a completion time of less than t, so the completion time decreased. This is a contradiction since the algorithm should have performed this swap before it terminated. Thus we must have $t \leq \frac{4}{3}s$, and the conclusion follows.

4. **(0 points, optional)**[4] Consider the following scheduling problem, similar to one in the comic below. The input is a set of people, a list of subsets of people ("events"), and a nonnegative integer k. The answer is yes if it's possible to assign to each event a positive integer ("day") at most k such that no person is double-booked: that is, intersecting events are assigned distinct numbers. (In the Figure[5] below, each event is assigned a different day, but, e.g., games and a movie could be scheduled simultaneously.) (Vaccine timings aren't part of our scheduling problem, because our scheduling problem is already NP-hard without them.) Prove that this scheduling problem is NP-complete.

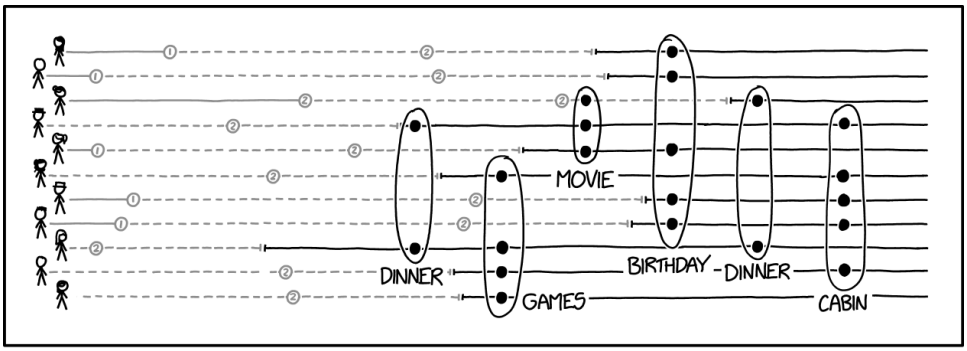

Figure 1: "As if these problems weren't NP-hard enough."

[4]We won't use this question for grades. Try it if you're interested. It may be used for recommendations/TF hiring.
[5]From xkcd, CC BY-NC 2.5 license.