CS 124 Problem Set 7
HARVARD ID: 10939860

1. **Solution.**

   We can see that each question has a recursive nature. For the first, considering a line graph, the recursion or pattern is easy to see after drawing a few examples. As a base case, if we say that $T_1(n)$ gives the number of independent sets on a line graph of n vertices, then $T_1(0) = 1$, since the empty set is an independent set. Further, $T_1(1) = 2$, because we include the empty set and the vertex by itself. With these base cases, we compute the next few examples, and see that $T_1(2) = 3, T_1(3) = 5, T_1(4) = 8$, and so on. It becomes clear that this recurrence follows the Fibonacci sequence. Thus, for a line graph of n vertices, the number of independent sets is the $(n + 1)th$ Fibonacci number, assuming that the Fibonacci sequence begins at 1.

   For a cycle of n vertices, let $T_2(n)$ be the number of independent sets. Our base cases will remain the same, and we will look at the values of $T_2(3)$ and above to see a new pattern. For a cycle of 3 vertices, we see that it has 4 independent sets, specifically each of the vertices by themselves in addition to the empty set, as no two vertices can be in the same independent set in this scenario. For a cycle of 4 vertices, we see that there are 7 independent sets. Considering these and computing further examples, we see that $T_2(3) = 4, T_2(4) = 7, T_2(5) = 12$, and so on. It becomes clear that the recurrence follows the Fibonacci numbers minus 1. Thus, for a cycle of n vertices, the number of independent sets is $F_{n+1} - 1$, assuming that the Fibonacci number sequence starts at 1 and F_{n+1} is the $(n + 1)th$ Fibonacci number.

   Finally, let $T_3(n)$ be the number of independent sets on a complete binary tree of n levels. Drawing out an example, we can write a recurrence of $T(n) = T(n - 1)^2 + T(n - 2)^4$. This comes from the fact that to compute the number of independent sets of the whole tree, one must start from the top, computing the number of independent sets on the 2 subtrees on the next level down, which requires one to compute the number of independent sets on the 4 subtrees on the next level down, and recurse as such. We know that a binary tree with 127 nodes has 7 levels, so by inputting the aforementioned recurrence into WolframAlpha with base cases $T(0) = 1$ and $T(1) = 2$ and looking at the value for $T(7)$, we see that a complete binary tree of 127 nodes has 1.335 x 10^{28} independent sets.

2. **Solution.**

   First, we consider the randomized algorithm generalized to the MAX-k-CUT problem. For every vertex, we randomly place it into one of the k sets, with equal probability. This is similar to the MAX-CUT process of flipping a coin, though instead of there being a 1/2 probability of a vertex getting placed into one of the two sets, each vertex has a $1/k$ chance of being placed into any specific set. On average, $1/k$ of edges in the graph will go from a set into itself, leaving the probability that an edge goes from one set to any other set as $1 - \frac{1}{k} = \frac{k-1}{k}$, so this is the bound on the randomized algorithm's performance, as we expect approximately this many edges to cross the cut on average, and our answer will be within a factor of $\frac{k}{k-1}$ of the optimal solution.

   Next, consider the local search algorithm from MAX-CUT generalized to MAX-k-CUT. Similar to the original algorithm, we begin with all vertices in one set. For every vertex, check it against each of the other $k - 1$ sets, and determine whether switching that vertex to a different set would increase the number of edges crossing the cut. If it would, then swap the vertex, and if not, keep searching, and if no other set would improve the cut, then leave the vertex in that set. In this case, we see that our bound is still $\frac{k-1}{k}$. Only as many as $\frac{1}{k}$ of the edges stay within the same set until we know that we can make an advantageous swap, which leaves us with an average of $\frac{k-1}{k}$ that leave from the original set to any other set, and our answer will again be within a factor of $\frac{k}{k-1}$ of the optimal solution.

3. **Solution.**

By the hint, we can see that if G has a clique of size k, then graph $G' = G$ x G has a clique of size k^2. We can generalize this to say that if we consider the graph $G^c = G_1$ x G_2 x ... x G_c, then G^c has a clique of size k^c if and only if G has a clique of size k. This is because in G, given a set of vertices that make a clique $(a_1, ..., a_i)$, every vertex is connected to the other. Once a cross product is taken, no matter the new number of vertices, they will still all be connected, since the set was connected to begin with, remaining a clique. In other words, in the graph $G' = G$ x G, an edge exists between two vertices (i, j) and (u, v) if and only an edge exists between the pair of vertices i and u and the pair j and v in G, or $i = u$ or $j = v$ or a combination.

We can then also see that given a clique of size k in graph G^c, we can easily find a clique of size $\sqrt[c]{k}$ in graph G. If K is the clique found in graph G^c, and V_i is the set of vertices used in the *ith* coordinate of the vertices in K, then every V_i must be a clique in G. Further, since the product of all V_i is greater or equal then the size of K, we know there must be a V_i such that $|V_i| \geq \sqrt[c]{K}$, proving the claim.

So, let us begin with a graph G and assume that the max clique in the graph is of size k. We then take the cross graph n times to get G^{2^n}, which has the max clique of size k^{2^n}, as proved above. Then, by the assumption given in the problem, we have that $APP \leq 2 * k^{2^n}$, where APP is our approximation. We assume the approximation is less than or equal to this factor because if the 2-approximation is accurate, then it will never overestimate k. We take the square root repeatedly (n times) of both sides so we get the approximation in terms of k. Thus, after one square root, we have that $\sqrt{APP} \leq \sqrt{2} * k^{2^{n-1}}$, and we continue this until we get down to k. It is easy to see that with sufficiently large n, $\sqrt[2^n]{2}$ approaches 1, giving the approximation $APP \leq (1 + \epsilon)k$ for some constant ϵ. The time involved in this algorithm will involve the time required to square G n times, as well as square root it n times. The space of the cross graph will be $|V|^{2^n}$.

4. **Solution.**

For this, we will use a proof by contradiction. We will assume that the local search algorithm always terminates in a stable state, which is intuitive. For any finite set of jobs, there must be some optimal way to break up those jobs between two machines, and the local search algorithm will always give a good approximation, as we will prove next. Thus, the local search algorithm will operate until it reaches a point where it cannot make any more improvements, which is a stable state.

Next, assume that the algorithm terminates in a stable state and that the approximation does not give a solution time within 4/3 of the optimal, namely $APP > 4/3 * OPT$. Knowing that job j_i has a corresponding running time r_i, and letting M_1 be machine 1 and M_2 be machine 2, we can elaborate this to say $max(\Sigma r_{M_1}, \Sigma r_{M_2}) > (4/3) * OPT$, and further that $max(\Sigma r_{M_1}, \Sigma r_{M_2}) > \frac{4}{3}(\frac{\Sigma r_i}{2})$, because the optimal solution is that the running time of the jobs are split evenly between the two machines, and we assume that our approximation is strictly greater than the optimal within the given approximation factor.

Now, let x_1 be the load on machine 1, and x_2 be the load on machine 2. For our purposes, we arbitrarily choose that $x_1 > x_2$. One of them has to be larger, because if they were equal than that would be optimal. From here, we can substitute these values into our approximation equation and simplify:

$$x_1 > \frac{4}{3}(\frac{x_1 + x_2}{2})$$

$$x_1 > \frac{2}{3}(x_1 + x_2)$$

$$\frac{1}{3}x_1 > \frac{2}{3}x_2$$

$$x_1 > 2x_2$$

Now, we are left with two cases. In the first case, there is only one job in x_1, meaning that our solution would be the optimal, so our initial assumption that $APP > (4/3) * OPT$ is a contradiction. In the second case, there is more than one job in x_1. Then, we can see that there would be a swap still possible that could be made to reduce the maximum running time between the two machines, since the load of x_1 is over twice as much as the load of x_2 and there are multiple jobs available to swap. In this case, our assumption that the algorithm terminates in a stable state is violated, raising a contradiction. Since we reach a contradiction in all cases, we have proved that $APP \leq (4/3) * OPT$.

I collaborated on this problem set with Nisreen Shiban, Kate Schember, and Nathan Lee.