CS 124: Problem Set 7

Chao Cheng

February 4, 2021

Problem 1. (a) If the capacity of some edge e increases by 1, then the minimum cut value also increases by at most 1 (if e traverses the minimum cut), which in turn means the maximum flow capacity has increased by 1. Luckily, we are already given the flow along each edge. Thus, it suffices to check if there is an augmenting path from s to t in the residual graph by running one iteration of Ford-Fulkerson, and if there is, we augment the flow with this path; otherwise, the maximum flow remains the same.

The time complexity of this algorithm is $O(|V|+|E|)$ because we are running a single iteration of Ford-Fulkerson, which takes $O(|E|)$ time with DFS (and, since this is a maximum flow graph it must connected, so $|E| \geq |V| - 1$ and $O(|V| + |E|) = O(|E|)$.

(b) If the capacity of some edge e decreases by 1, then we consider two possibilities. Let $c(e)$ be the capacity through e before decreasing capacity, $c'(e) = c(e) - 1$ be the capacity through e after decreasing capacity, and $f(e)$ be the flow through e (since we are given the flow through each edge). If before decreasing capacity $c(e) \geq f(e) + 1$, then after decreasing capacity $c'(e) \geq f(e)$ which implies that no change is needed for the maximum flow; that is, the maximum flow remains the same. However, if before decreasing capacity $c(e) = f(e)$, then after decreasing capacity $c'(e) = c(e) - 1 < f(e)$, so the flow through e is over capacity and we must remove one unit of flow.

To accomplish this, suppose $e = (u, v)$, and run a BFS from s to u keeping track of any edges with flow greater than 0. Similarly, run a BFS from v to t, also keeping track of any edges with non-zero flow. Then choose any path along the aforementioned edges going through e from s to t, and remove 1 unit of flow from each edge along that path.

Thus we have removed one unit of flow from e so that $f(e) \leq c'(e)$. Now we consider that if e traversed the minimum cut, the minimum cut value (as well as the maximum flow capacity) will have decreased by 1, and since we have removed 1 unit of flow from the previous maximum flow, our new maximum flow is the maximum possible flow. However, if e does not traverse the minimum cut, then the maximum flow capacity is the same as before. Since we had removed one unit of flow, we need to add it back, which can be accomplished by running a single iteration of Ford Fulkerson, thus producing the new maximum flow.

The time complexity of this algorithm is $O(|V| + |E|)$. The two BFS each take $O(|V| + |E|)$ time, and we are running a single iteration of Ford-Fulkerson, which takes $O(|E|)$ time with DFS (and, since this is a maximum flow graph it must connected, so $|E| \geq |V| - 1$ and $O(|V| + |E|) = O(|E|)$.

Problem 2. (a) Let L_n be the number of independent sets in a line graph with n vertices. We have that $L_0 = 1$ (the empty set) and $L_1 = 2$ (the empty set and the set of the singular vertex).

We can then construct L_n from L_{n-1} and L_{n-2} as follows. Consider the nth vertex in the line. We can separate the total number of independent sets L_n into two groups (or cases): the independent sets that include the nth vertex and the independent sets that do not include the nth vertex.

- If the nth vertex is included in an independent set, then by definition, the $(n-1)$th vertex will not be in that set, so the total number of sets in this case would be equivalent L_{n-2}.
- If the nth vertex is not included in an independent set, then we are only considering the first $n-1$ vertices, which is equivalent to L_{n-1}.

Thus $L_n = L_{n-1} + L_{n-2}$, which is the Fibonacci sequence (where $F_0 = 0$, $F_1 = 1$, $F_2 = 1$, $F_3 = 2$, and so on). Since L_0 corresponds to F_2 and L_1 corresponds to F_3, we have $L_n = F_{n+2}$, i.e. the number of independent sets in a line graph with n vertices is the $(n+2)$th Fibonacci number.

(b) Let C_n be the number of independent sets in a cycle graph with n vertices. We have $C_0 = 1$ (the empty set), $C_1 = 2$ (the empty set and the set of the singular vertex), and $C_2 = 3$ (the empty set and the sets of either individual vertex).

Similar to above, we have that if the nth vertex is included in an independent set, then by definition the vertices on either side, the $(n-1)$th vertex and the first vertex, will not be included in that set, so this case is the same as L_{n-3}, since we are only considering a line with $n-3$ vertices. If the nth vertex is not included, then it is the same as L_{n-1}.

Thus, $C_n = L_{n-1} + L_{n-3}$, and going back to our analogy above, we have that the number of independent sets in a cycle graph with n vertices is equal to the sum of the $(n+1)$th and $(n-1)$th Fibonacci numbers.

(c) Let T_n be the number of independent sets in a complete binary tree of height n. Again, we have $T_0 = 1$ (the empty set) and T_1 (the empty set and the set of the singular vertex).

For T_n, we have that if the root is *not* included in an independent set, then we are considering independent sets in the two subtrees of height $n-1$; since the two subtrees are disjoint, any combination of an independent set in the left subtree and an independent set in the right subtree will yield an independent set in the original tree. Thus, there are T_{n-1}^2 such independent sets in this case. If the root is included in an independent set, then its children, by definition, will not be included in that set, so this is identical to the number of independent sets in the four grandchild subtrees of height $n-2$, in which case there are T_{n-2}^4 such combinations.

Thus, $T_n = T_{n-1}^2 + T_{n-2}^4$. Since a complete binary tree with 127 nodes has height 7, we have $T_7 = 133453460314446328414276439 06$.

Problem 3. 1. For the generalized randomized algorithm, we divide the vertices into k sets, deciding where each vertex goes by rolling a k-sided fair die. An edge crosses the cut when its endpoints belong to different sets, and the probability of this happening is $\frac{k-1}{k}$. Thus, on average, the number of edges crossing the cut will be $\frac{k-1}{k}|E|$, which is within a factor of $\frac{k}{k-1}$ of the optimal outcome, $|E|$. For $k = 2$, i.e. the MAX CUT problem, the randomized algorithm will produce a random assignment that is within a factor of $\frac{2}{2-1} = 2$ of the optimal outcome.

2. For the generalized local search algorithm, we split the vertices into sets $S_1, S_2, \ldots S_k$, and start with all vertices in S_1. Then, if we can switch a vertex to a different set such that the number of edges crossing the cut strictly increases, then we do so, and we repeat this until the cut can no longer be improved simply by switching any one vertex. In total, we switch vertices at most $|E|$ times, since each switch increases the number of edges by at least 1, and at most $|E|$ edges can cross the cut.

   Similar to lecture, we calculate a running sum for the cut as follows. Consider any vertex $v \in S_1$. For every vertex $w \notin S_1$ that v is connected to, we add $\frac{1}{2}$ to the running sum. We repeat for each vertex in $S_2, S_3, \ldots S_k$. Thus we have for the cut C that

$$C = \frac{1}{2}\left(\sum_{n=1}^{k}\sum_{v\in S_n}\left|\{w : (v,w) \in E, w \notin S_k\}\right|\right)$$

   Since we are using the local search algorithm, at least $\frac{k-1}{k}$ edges from any vertex v must point to vertices in other sets, because otherwise we would just switch v to another set. Thus, if v has degree $\delta(v)$, then we have

$$\begin{aligned} C &= \frac{1}{2}\left(\sum_{n=1}^{k}\sum_{v\in S_n}\left|\{w : (v,w) \in E, w \notin S_k\}\right|\right) \\ &\geq \frac{1}{2}\left(\sum_{n=1}^{k}\sum_{v\in S_n}\frac{(k-1)\delta(v)}{k}\right) \\ &= \frac{(k-1)}{2k}\sum_{v\in V}\delta(v) \\ &= \frac{k-1}{2k}\cdot 2|E| \end{aligned}$$

   where the last line comes from the fact that summing the degrees of all vertices gives twice the number of edges (since each edge is counted twice). Thus, we have

$$C \geq \frac{k-1}{k}|E|$$

   which is the same as the randomized algorithm (that is, within a factor of $\frac{k}{k-1}$ of the optimal outcome, $|E|$), though we would expect the local search algorithm to do a bit better on average.

   The runtimes of either algorithm, like their ungeneralized versions, are clearly polynomial.

Problem 4. Suppose G has a clique C of size k, and let the vertices in C be denoted as $1, 2, \ldots k$. We have that since C is a clique, any pair of vertices (i, j) where $1 \leq i, j \leq k \in C$ will have an edge between them.

Also, there will be k^2 vertices (i, j) in G' (where $1 \leq i, j \leq k$) that correspond to pairs of nodes in C. Let (i_1, j_1) and (i_2, j_2) be any two such vertices. By definition of a clique, we know that $(i_1, i_2) \in E$ or $i_1 = i_2$, and we also know that $(j_1, j_2) \in E$ or $j_1 = j_2$. Thus, $\{(i_1, j_1), (i_2, j_2)\} \in E'$ for any (i_1, j_1) and (i_2, j_2). Thus, these k^2 vertices in G' form a clique, which we will call C'.

To show that the size of C' cannot exceed k^2, suppose for the sake of contradiction that there is a vertex $(i, v) \in C'$ where $1 \leq i \leq k$ and $v > k$ (so that $v \notin C$). Since v is not in the original clique C, there must exist some $w \in C$ such that there is no edge between v and w in G. Then for some $(j, w) \in C'$ (where $j \in C$), we have that $\{(i, v), (j, w)\} \notin E'$ because $(v, w) \notin E$. Thus the only possible vertices in C' are the k^2 pairs of vertices in C, and therefore the maximum size of C' is k^2.

Now suppose the maximum clique in G has size k, so that the maximum clique in G' has size k^2. We are given a polynomial time algorithm for approximating the maximum clique in a graph to within a factor of 2. If we run this algorithm on G', it will give us a maximum clique with size at least $\frac{k^2}{2}$, which tells us that the maximum clique size for G is at least $\frac{k}{\sqrt{2}}$ — a better bound than $\frac{k}{2}$ (a factor of 2). Thus, by constructing G' (which takes polynomial time) and running the algorithm on it, we end up with a polynomial time algorithm for approximating the maximum clique of the original graph G to within a factor of $\sqrt{2}$.

We can generalize this by seeing that $G^n = G \times \cdots \times G$ will have a maximum clique of size k^n (for example, G' above is G^2), and thus by constructing G^n (which takes polynomial time since there are $|V|^n$ vertices and at most $|V|^{2n}$ edges) and running the algorithm on it, we end up with a polynomial time algorithm for approximating the maximum clique of the original graph G to within a factor of $\sqrt[n]{2}$ (since for G^n it gives us a maximum clique size of at least $\frac{k^n}{2}$, which tells us that the maximum clique size for G is at least $\frac{k}{\sqrt[n]{2}}$). Then, for any arbitrary $\epsilon > 0$ we can achieve a bound of $1 + \epsilon$ since there exists some n such that $\sqrt[n]{2} = 1 + \epsilon$.

Problem 5. Let the optimal completion time be t^*, and let the total load on the first machine be t_1 and the total load on the second machine be t_2.

Suppose for the sake of contradiction that the algorithm terminates in a stable state but the completion time is not within $\frac{4}{3}$ of optimal. We can assume that $t_1 \neq t_2$ since if they were equal, we would have an optimal configuration. Without loss of generality, suppose $t_1 > t_2$. Then the total completion time is t_1, so we have that $t_1 > \frac{4}{3}t^*$. Also, we know that the optimal completion time must be at least half of the sum of all running times. Thus we have

$$\begin{aligned}
\frac{t_1 + t_2}{2} &\leq t^* \\
t_1 + t_2 &\leq 2t^* \\
t_2 &\leq 2t^* - t_1 \\
t_2 &\leq 2t^* - t_1 < 2t - \frac{4}{3}t^* \\
t_2 &< \frac{2}{3}t^* \\
t_2 &< \frac{2}{3}t^* < \frac{1}{2}t_1 \\
t_2 &< \frac{1}{2}t_1
\end{aligned}$$

Using this last inequality, we consider several cases.

- **Machine 1 has 0 jobs:** we assumed that $t_1 > t_2$, so in this case the load on both machines is 0, which is optimal.
- **Machine 1 has 1 job:** if Machine 1 only has 1 job, then since we assumed $t_1 > t_2$, this must be already be the optimal configuration.
- **Machine 1 has 2 or more jobs:** if Machine 1 has multiple jobs, then the smallest job must have a runtime of at most $\frac{1}{2}t_1$ (otherwise it would be the biggest job). Because we know that $t_2 < \frac{1}{2}t_1$, we can safely swap this job from Machine 1 to Machine 2 without increasing the maximum completion time. We can keep swapping until the smallest job of Machine 1 has a greater runtime than the difference in total completion times between Machine 1 and Machine 2, in which case we would have $t_1 \leq \frac{4}{3}t^*$.

Thus in each case, the outcome is either optimal or the maximum completion time is within $\frac{4}{3}$ of optimal, which is a contradiction. Therefore, the local search algorithm will always terminate in a stable state with a completion time within a factor of $\frac{4}{3}$ of the optimal configuration.