Homework 7
HUID: 71238520

Out: April 10, 2017
Due: April 21, 2017 (4:59 pm)

1 Problem 1

If we restrict the problems we look at, sometimes hard problems like counting the number of independent sets are in a graph become solvable. For instance, consider a graph that is a line on n vertices. (That is, the vertices are labelled 1 to n, and there is an edge from 1 to 2, 2 to 3, etc.) How many independent sets are there on a line graph? Also, how many independent sets are there on a cycle of n vertices? (Hint: In this case, we want to express your answer in terms of a family of numbers like For n vertices the number of independent sets is the nth prime. And thats not the answer.)

Similarly, describe how you could quickly compute the number of independent sets on a complete binary tree. (Here, just explain how to compute this number.) Calculate the number of independent sets on a complete binary tree with 127 nodes. (Warning: its a pretty big number.)

1.1 Solution

2 Problem 2

Consider the problem MAX-k-CUT,which is like the MAXCUT algorithm, except that we divide the vertices into k disjoint sets, and we want to maximize the number of edges between sets. Explain how to generalize both the randomized and the local search algorithms for MAX CUT to MAX-k-CUT and prove bounds on their performance.

2.1 Solution

To generalize the MAXCUT randomized algorithm to MAX-k-CUT, we randomly assign vertices to the k disjoint sets, by using a random number generator.

The probability that a given edge crosses sets is 1 - the probability that both its vertices are in the same set, which is $1 - k(\frac{1}{k} \cdot \frac{1}{k}) = 1 - \frac{1}{k} = \frac{k-1}{k}$. This means that on average, a fraction of $\frac{k}{k-1}$ of the total edges will cross between sets. Since the most we could have is for all edges to cross between sets, this random assignment will be, on average, within a factor of $\frac{k-1}{k}$ of the optimum.

For local search algorithms, we start with a randomized solution. We then select a vertex in the first of the k sets, v in S_1, and check if moving it to a different set, S_2, would increase the number of edges across the cut. If this wouldn't increase the number of edges, we check if moving v to S_3 would increase the edges crossing the cut, and so on, until S_k. The first increase that is found is the one that is made. We repeat this action for all vertices v in each set, sequentially, and then start over, until the cut partition can no longer be improved by a single switch. We switch vertices at most $|E|$ times, as shown in lecture notes.

We can count the edges in the cut in the following way: consider any vertex $v \in S_1$. For every vertex w in $S_2 ... S_k$ that it is connected to by an edge, we add 1/2 to a running sum. We do the same for each vertex in $S_2 ... S_k$. Note that each edge crossing the cut contributes 1 to the sum 1/2 for each vertex of the edge.

Hence the cut C satisfies

$$\begin{aligned} C = \frac{1}{2}(& \sum_{v \in S_1} |\{w : (v,w) \in E, w \in S_2 \text{ or } S_3 \text{ or } ... \text{ or } S_k\}| + \\ & \sum_{v \in S_2} |\{w : (v,w) \in E, w \in S_1 \text{ or } S_3 \text{ or } ... \text{ or } S_k\}| + \\ & ... + \\ & \sum_{v \in S_k} |\{w : (v,w) \in E, w \in S_1 \text{ or } S_2 \text{ or } ... \text{ or } S_{k-1}\}|) \end{aligned}$$

Due to our usage of a terminating local search, at least $\frac{k-1}{k}$ of the edges from any vertex v must lie in sets other than v, otherwise, we could move the set of v, and improve the cut. We know this via contradiction, since if less than $\frac{k-1}{k}$ of edges from v crossed the sets, we would have moved v to another set. This implies, following from the math in lecture 20 notes:

$$C \geq \frac{1}{2}(\sum_{v \in S_1} \delta(v) \frac{k-1}{k} + \sum_{v \in S_2} \delta(v) \frac{k-1}{k} + ... + \sum_{v \in S_K} \delta(v) \frac{k-1}{k}) \qquad (1)$$

Combining the summations,

$$C \geq \frac{1}{2}(\sum_{u \in V} \delta(u)\frac{k-1}{k}) \tag{2}$$

Therefore,

$$C \geq \frac{k-1}{2k}\sum_{u \in V} \delta(u) \tag{3}$$

Since the sum of the degrees of all the vertices is twice the number of edges,

$$C \geq \frac{k-1}{k}|E| \tag{4}$$

Since the most we could have is for all edges to cross between sets, the local search algorithm will yield an approximation, on average, within a factor of $\frac{k-1}{k}$ of the optimum.

3 Problem 3

Prove that if there exists a polynomial time algorithm for approximating the maximum clique in a graph to within a factor of 2, then there is a polynomial time algorithm for approximating the maximum clique in a graph to within a factor of $(1 + \epsilon)$ for any constant $\epsilon > 0$. 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

$$[\{(u, w)\} \in E \text{ or } u = w] \text{ and } [\{(v, x)\} \in E \text{ or } v = x]. \tag{5}$$

If G has a clique of size k, then how large a clique does G' have?

3.1 Solution

4 Problem 4

We consider the following scheduling problem, similar to one that we studied before: we have two machines, and a set of jobs $j_1, j_2, j_3, ..., j_n$ 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.

Prove that the local search algorithm always terminates in a stable state, and that the completion time is within a factor of 4/3 of the optimal.

4.1 Solution

Given that we have a set of jobs $j_1, j_2, j_3, ..., j_n$, and two machines M_1 and M_2. We use local search to find an optimum assignment of jobs to M_1 or M_2. The solution space S can be represented by an array of length n that contains 1 or 2, where S_i corresponds to the machine assignment of j_i. There are therefore 2^n solutions in the solution space. If for some job assignment,

Even for the best possible solution, the *makespan* is at least as big as the biggest job, or at least as big as half the sum of the jobs. [From PSET3.]