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

1.1.1 Line Graph

We start by looking at some small graphs.

A graph with 0 nodes has 1 independent set.
A graph with 1 node has 2 (empty set and itself).
A graph with 2 nodes has 1 (empty) + 2 (each node itself) = 3.
A graph with 3 nodes has 1 (empty) + 3 (each node itself) + 1 (1st and 3rd node) = 5.
A graph with 4 nodes has 1 (empty) + 4 (each node itself) + 2 (alternating nodes) + 1 (extreme ends) = 8.

A recurrence starts to emerge, where IS(n) = IS(n-1) + IS(n-2), with base cases of IS(0) = 1 and IS(1) = 2. This is the recurrence for the Fibonacci Numbers, shifted one down.

Therefore, for n vertices in a line graph there are Fib(n + 1) independent sets.

The proof that this formula succeeds is by induction. We know that the base case is true since IS(0) can have only one independent set, the empty set, and therefore is 1. IS(1)'s independent sets are the empty set as well as a set of the node itself.

Assuming that the values of IS preceding it are correct, that is, that IS(0) ... IS(k - 2), IS(k - 1) are all calculated correctly, the value of IS(k) can be derived as follows.

The set of independent sets for k vertices, $S(k)$ can be obtained by modifying a version of the independent sets for k - 1 vertices, namely by duplicating every set in $S(K-1)$ that does not contain v_{k-1} (since there is an edge between v_k and v_{k-1}) and adding v_k to one of the duplicates. Notice, however, that every set in $S(K-1)$ that does not contain v_{k-1} is equivalent to every set in $S(K-2)$. Therefore, instead of duplicating and modifying part of $S(K-1)$, we simply add $S(K-2)$ to $S(K-1)$. IS(k) = IS(k-1) + IS(k-2) follows from this, therefore proving by induction that our family of values supplied is correct.

1.1.2 Cycle Graph

This is a modified version of the line graph, where the first and last vertices cannot be in the same set together. Some small graphs:
C(0) = 1
C(1) = 1
C(2) = 3
C(3) = 4
C(4) = 7
C(5) = 11
C(6) = 18
This also reveals a fibonacci-like pattern with different base cases: $C(0) = 1$, $C(1) = 1$, $C(2) = 3$, and $C(n) = C(n-1)+C(n-2)$ for $n > 2$. Therefore, for n vertices in a cyclical graph, there are C(n) independent sets as described below.

$$C = \begin{cases} 1 & n = 0, n = 1 \\ 3 & n = 2 \\ C(n-1) + C(n-2) & n > 2 \end{cases}$$

The proof for this formula also follows from induction. We know that the base case is true, since C(0) is the empty set = 1, and C(1) is also only the empty set since v_1 is connected to itself. C(2) is the empty set, as well as the sets containing each individual vertex = 3.

Assuming that the values of C preceding it are correct, that is, that C(0) ... C(k - 2), C(k - 1) are all calculated correctly, the value of C(k) can be derived as follows.

The set of independent sets for k vertices, $S(k)$ can be obtained by modifying a version of the independent sets for k - 1 vertices, specifically by doing the following:

1. duplicating every set in $S(K-1)$ that does not contain v_1 or v_{k-2} and adding v_{k-1} to the duplicate, since v_{k-1} is no longer connected to v_1.
2. duplicating every set in $S(K-1)$ that does not contain v_{k-1} (since there is an edge between v_k and v_{k-1}) and adding v_k to one of the duplicates
3. removing from the result of (2) above (the modified duplicates), every set that contains v_1, since the cyclical graph places an edge between v_1 and v_k.

Notice, however, that every set in $S(K-1)$ that does not contain v_{k-1} is equivalent to every set in $S(K-2)$. And that the numerical effects of (1) and (3) reverse each other. Therefore, instead of duplicating and modifying part of $S(K-1)$, we simply add $S(K-2)$ to $S(K-1)$. C(k) = C(k-1) + C(k-2) follows from this, therefore proving by induction that our family of values supplied is correct.

1.1.3 Complete Binary Tree

Defining a formula B(n) that calculates the number of independent vertices in a complete binary tree with n levels.

$$B = \begin{cases} 1 & n = 0 \\ 2 & n = 1 \\ B(n-1)^2 + B(n-2)^4 & n > 1 \end{cases}$$

The proof for this formula also follows from induction. We know that the base case is true, since B(0) is the empty set = 1, and B(1) is equal to the empty set and the set containing the root, which is the only vertex at the first level.

Assuming that the values of C preceding it are correct, that is, that B(0) ... B(k - 2), B(k - 1) are all calculated correctly, the value of B(k) can be derived as follows.

The set of independent sets for k levels, $S(k)$ can be obtained by modifying a version of the independent sets for k - 1 levels, specifically by doing the following:

1. Similar to the previous algorithms, we add all the new vertices added at step k to the independent sets of $S(k-2)$. Since, at this step, we are adding $2 * k - 1$ new vertices, all possible combinations of elements of the powerset of the new vertices can be added to each set in $S(k-2)$, yielding a $2^{2*k-1} \cdot S(k-2) = 4^{k-1} \cdot S(k-2)$ term.

2. For each of the independent sets in $S(k-1)$, we add the new vertices to sets that do not contain their parents. Since parent nodes will appear in $\frac{1}{2*k-2}$ of the sets, we add a term $\frac{2(k-2)-1}{2(k-2)} S(k-1)$.

Each of these terms can be represented as the ones in the recurrence provided by noting that the terms can also be defined recursively and by thinking about parents and children the independent sets of a node's parents ($S(k-1)$) is combined with itself since there are two children coming from each parent, therefore the $B(k-1)^2$ term, and for the independent sets of a node's grandparents, the independent sets are combined with themselves in 2^2 ways since their grandchildren are a factor of 4 more common. The inclusion of both the parents and grandparents is necessary because we want sets that both include and exclude, recursively, the root node.

1.1.4 Independent Sets on a Complete Binary Tree with 127 Nodes

I calculated the number of independent sets on a complete binary tree with 127 nodes using the following code in Mathematica:

RecurrenceTable $\left[\left\{a(n) = a(n-2)^4 + a(n-1)^2, a(0) = 1, a(1) = 2\right\}, a, \{n, 1, 7\}\right]$

Which yields the output,

$\{2, 5, 41, 2306, 8143397, 94592167328105, 133453460314446328414276439 06\}$

Therefore, for a binary tree with 127 nodes, there are 133453460314446328414276439 06 independent sets.

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

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

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:

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

Combining the summations,

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

Therefore,

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

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

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

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

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

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

3.1 Solution

For $G' = G \times G$, G' has $k \times k$ vertices which are completely connected, as follows from the definition of G', that forms vertices from completely connected sub-vertices, and also follows from the fact that G has k vertices that are will always be connected (definition of clique). Therefore, the clique of G' is of size k^2.

By the definition of G', we also know that if G' has a clique of k^2, G has one of k. This is since for any vertex (u, v) in G' there is a corresponding vertex u and v in G, and once repeats are discarded due to the property of vertex uniqueness, we have k vertices. We know that these k vertices are completely connected using the definition of edges for G'.

This logic can be generalized to greater products as well, like $G \times G \times G \times G$.

To prove that there is a polynomial time algorithm for approximating the maximum clique in a graph to within a factor of $(1 + \epsilon)$, we take higher products of G as described above, say the nth product, which requires multiplying G by itself and multiplying the product by itself and repeating the process n times, almost like repeated squaring from homework 1. This yields G'^n, which, by the logic above, has a clique size of k^{2n}. We use the given polynomial time 2-approximation algorithms to obtain the max clique size between k^{2n} and $\frac{k^{2n}}{2}$. We use this result to calculate the max clique size by taking the square root of the algorithm's output, which, in the worst case

is $\sqrt[2n]{\frac{k^{2n}}{2}}$. As n approaches infinity, this value approaches k, which means that we can get closer to the actual maximum clique by taking higher-order products.

We use this finding to now find a relationship between epsilon and n, or the order of the product required to obtain the desired accuracy.

$$\sqrt[2n]{\frac{k^{2n}}{2}} = k(\epsilon + 1)$$

$$k \sqrt[2n]{\frac{1}{2}} = k(\epsilon + 1)$$

$$2^{-\frac{1}{2n}} = \epsilon + 1$$

Which, using Mathematica, gives the formula for n

$$n = \frac{1}{2\left(\frac{\log\left(-\frac{1}{-1-e}\right)}{\log(2)} + \frac{2i\pi c_1}{\log(2)}\right)}, c_1 \in \mathbb{Z}$$

This method of computing the maximum clique is still in polynomial time since its various components are in polynomial time: computing n is a polynomial-time step, constructing the exponentiated product graph is also in polynomial time since n is a predetermined constant, and the 2-approximation algorithm is still polynomial. Taking the appropriate root of the result from the 2-approximation algorithm is also still polynomial.

We have therefore proved that there is a polynomial time algorithm for approximating the maximum clique in a graph to within a factor of $(1 + \epsilon)$.

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.

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

Case 1: Optimal solution is at least as big as the biggest job j_b. In this case, the optimal solution is to place the largest job on one machine and the other jobs on the second machine. With any initial assignment, say with j_b initially placed in M_1, the algorithm would move all jobs j_i, $i! = b$ that are in M_1 to M_2, since this would reduce the *makespan*. Once all the jobs except j_b are moved to M_2, the algorithm halts, and has also achieved the optimal solution. Therefore the algorithm terminates in a stable state and yields a completion time within the required factor of 4/3 of the optimal.

Case 2: Optimal solution is at least as big as half the sum of the jobs. The best optimal solution, in this case, would be half the sum of the jobs if the jobs can be evenly split between each other. Let us represent the solution determined by local search in the following way: the load on M_1 is l_1 and the load on M_2 by $l_1 + x$, where x is the offset between the load on M_1 and M_2. Since this is the termination state of the local search, we know that $x \leq l_1$, since if the opposite were true, the local search would not have terminated and would have moved an element from M_2 to M_1.

The *makespan*, in this case, is $l_1 + x$, since that is the larger of the two machine loads. Using this representation, the optimal solution can be represented as $\frac{2 \cdot l_1 + x}{2}$. We prove that $l_1 + x \leq \frac{4}{3} \frac{2 \cdot l_1 + x}{2}$ by contradiction. If the converse were true, then:

$$\begin{aligned} 6 \cdot l_1 + 6 \cdot x &> 8 \cdot l_1 + 4 \cdot x \\ 2 \cdot x &> 2 \cdot l_1 \\ x &> l_1 \end{aligned}$$

However, we already established that $x \leq l_1$, and so we arrive at a contradiction, and prove that $l_1 + x \leq \frac{4}{3} \frac{2 \cdot l_1 + x}{2}$ is true. The completion time for this case is within a factor of 4/3 of the optimal.

Therefore, we have proved that the completion time for both possible cases is within 4/3 of the optimal.

We know that the algorithm arrives at a stable solution no matter the case, since failure to arrive at a stable solution would mean that the *makespan* was decreased infinitely, eventually becoming a negative number. This is impossible since runtimes, composed of sums of positive numbers (job values) cannot be negative.