CS 170 DIS 11

Released on 2018-11-13

1 Local Search for Max Cut

Sometimes, local search algorithms can give good approximations to NP-hard problems. In the Max-Cut problem, we have a graph $G(V, E)$ and want to find a cut (S, T) with as many edges crossing as possible. One local search algorithm is as follows: Start with any cut, and while some vertex v in S has more neighbors in S than T, we move v from S to T (we do the same for any vertex v in T with more neighbors in T than S). Note that any time we move a vertex across the cut, the number of edges crossing the cut increases.

(a) Give an upper bound on the number of iterations this algorithm can run for (i.e. the total number of times we move a vertex).

(b) Show that when all vertices have more neighbors on the opposite side of the cut, at least half the edges in the graph cross the cut.

Solution:

(a) $|E|$ iterations. Each iteration increases the number of edges crossing the cut by at least 1. The number of edges crossing the cut is between 0 and $|E|$, so there must be at most $|E|$ iterations.

(b) Let $\delta_{in}(v)$ be the number of edges from v to other vertices on the same side of the cut, and $\delta_{out}(v)$ be the number of edges from v to vertices on the opposite side of the cut. Then, the total number of edges crossing the cut is $\frac{1}{2}\sum_{v\in V}\delta_{out}(v)$ whereas the total number of edges in the graph is $\frac{1}{2}\sum_{v\in V}(\delta_{in}(v)+\delta_{out}(v))$. We know that $\delta_{out}(v) > \delta_{in}(v)$ for all v, so the former is at least half as large as the latter.

2 Multiway Cut

In the multiway cut problem, we are given a graph $G(V, E)$ with k special vertices $s_1, s_2 \ldots s_k$. Our goal is to find the smallest set of edges F which when removed from the graph disconnect the graph into at least k components where each s_i is in a different component. When $k = 2$, this is exactly the min s-t cut problem, but if $k \geq 3$ the problem becomes NP-hard.

Consider the following algorithm: Let F_i be the set of edges in the minimum cut with s_i one one side and all other special vertices on the other side. Output F, the union of all F_i. Note that this is a multiway cut because removing F_i from G isolates s_i in its own component.

(a) Explain how each F_i can be found in polynomial time.

(b) Let F^* be the smallest multiway cut. Consider the components that removing F^* disconnects G into, and let C_i be the vertices in the component with s_i. Let F_i^* be the set of edges in F^* with exactly one endpoint in C_i. How many different F_i^* does each edge in F^* appear in? How do the size of F_i and F_i^* compare?

(c) Using your answer to the previous part, show that $|F| \leq 2|F^*|$. (Challenge: How could you modify this algorithm to output F such that $|F| \leq (2 - \frac{2}{k})|F^*|$?)

(As an aside, consider the minimum k-cut problem, where we want to find the smallest set of edges F whose removal disconnects the graph into at least k components. The following greedy algorithm for minimum k-cut gets a $(2 - \frac{2}{k})$-approximation: Initialize F to the empty set. While $G(V, E - F)$ has less than k components, find the minimum cut within each component of $G(V, E - F)$, and add the edges in the smallest of these cuts to F. Showing this is a $(2 - \frac{2}{k})$-approximation is fairly difficult.)

Solution:

(a) Consider adding a vertex t to the graph and connecting t to all special vertices except s_i with infinite capacity edges. Then F_i is the minimum s_i-t cut, which we know how to find in polynomial time.

(b) Each edge in F^* appears in exactly two of the sets F_i^*.

Note that F_i^* is the set of edges in a cut which disconnects s_i from the other special vertices. Then by definition F_i has fewer edges than F_i^* since F_i is the minimum cut disconnecting s_i from all other special vertices.

(c) We combine the answers to the previous part and note that F's size is at most the total size of all F_i to get:

$$|F| \leq \sum_i |F_i| \leq \sum_i |F_i^*| = 2|F^*|$$

To get the $(2 - \frac{2}{k})$-approximation, after computing all F_i, we instead output F as the union of all F_i except for the one with the most edges. Let this be F_j. This is still a multiway cut because each s_j is still disconnected from all other s_i. Then:

$$|F| \leq \sum_{i \neq j} |F_i| \leq (1 - \frac{1}{k}) \sum_i |F_i| \leq (1 - \frac{1}{k}) \sum_i |F_i^*| = (2 - \frac{2}{k})|F^*|$$

3 Fast Modular Exponentiation

Give a polynomial time algorithm for computing $a^{b^c} \mod p$ for prime p and integers a, b, and c.

Solution: We know how to compute $x^y \mod z$ efficiently for any x, y, z: Square x and apply $\mod z$ repeatedly to compute $x, x^2, x^4 \ldots$ all $\mod z$. Then x^y can be written as some product of these (e.g. $x^5 = x * x^4$), so we can compute x^y easily.

Then, we show how to reduce this problem to two instances of finding $x^y \mod z$:

- Since p is prime, by Fermat's Little Theorem, we know $a^{p-1} \mod p = 1$. So we first find $d = b^c \mod (p-1)$.
- We then note that $a^{b^c} \mod p = a^d \mod p$. Then, we just compute $a^d \mod p$.

4 Fermat's Little Theorem as a Primality Test

Recall that Fermat's Little Theorem states the following:

"For a prime p and a coprime with p, $a^{p-1} \equiv 1 \pmod p$."

Assume for a general (not necessarily prime) p, we want to determine if p is prime. It may be tempting to try to use Fermat's Little Theorem as a test for primality. That is, pick some random a and compute $a^{p-1} \pmod p$. If this is equal to 1, return that p is prime, else return that it is composite. In this question we will investigate how effective this method actually is.

(a) Suppose we wanted to test if 15 was prime. What is a choice of a that would trick us into thinking it is prime? What is a choice of a that would lead us to the correct answer? For choices of a that trick us into believing p is prime, we often say that p is "Fermat pseudoprime" to base a.

(b) Suppose there exists some a in $\{1, \ldots p-1\}$ such that $a^{p-1} \not\equiv 1 \pmod p$, where a is coprime with p. Show that p is not Fermat pseudoprime to at least half the numbers in $\pmod p$. How might we use this to make our algorithm more effective?

(c) Given the improvement from the previous question, why might our algorithm still fail to be a good primality test?

Solution:

(a) A choice of a that would trick us into thinking 15 is prime is 4. There are a few other numbers we could have used here. A choice of a that would lead us to the correct answer is 7.

(b) Let's assume there is at least one number b such that $b^{p-1} \equiv 1 \pmod p$. $(a * b)^{p-1} \not\equiv 1 \pmod p$. Further more, for each possible choice of b, $a * b$ will be a unique number. This is the case since a necessarily has an inverse in mod p, making the function $f(x) = a * x \pmod p$ a bijection. For every b that p is Fermat pseudoprime to, we have a unique a that would have led us to the correct answer. Thus at least half the numbers $\pmod p$ would lead us to the correct answer.

We can improve our algorithm by checking multiple a rather than just 1. This doesn't increase our runtime substantially, but will sharply decrease the probability of a false positive.

(c) For prime p we will always arrive at the correct answer. For non-prime p, we know that when there exists an a coprime with p such that $a^{p-1} \not\equiv 1 \pmod p$, we will probably arrive at the correct answer. However, we are not guaranteed the existence of such an a in the first place. There are potentially numbers where no such a exists. These numbers are called Carmichael numbers.