CMPSCI 611 Advanced Algorithms	**Fall 2000**

Lecture 14: October 31

Lecturer: Micah Adler *Scribe: Wei Wei, Huan Li*

14.1 Application of Schwartz's Algorithms

In this section we will illustrate how to use Schwartz's Algorithm to solve the perfect matching problem. Consider a bipartite graph $G = (U, V, E)$ with the independent sets of vertices $U = \{u_1, ..., u_n\}$ and $V = \{v_1, ...v_n\}$. A matching is a collection of edges $M \subseteq E$ such that each vertex occurs at most once in M. A *perfect matching* is a matching of size n.

14.1.1 Perfect Matching Problem

Input: A bipartite graph $G = (U, V, E)$, where $|U| = |V| = n$.
Question: Does G contain a perfect matching (i.e., a matching of size n)?

14.1.2 Network Flow Solution

We can use Network Flow to solve this problem and we know, from Lecture 12, that the Even-Tarjan[ET75] algorithm can solve it in time $O(m\sqrt{n})$, where m is the number of edges.

14.1.3 Use of Schwartz's Algorithm

Alternatively we can use Schwartz's Algorithm, which is discussed in this section.

Let $M[G]$ be the $n \times n$ adjacency matrix for $G = (U, V, E)$, defined as follows:

$$M[G](i,j) = \begin{cases} 1 & \text{if } (u_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Let $A[G]$ be an $n \times n$ matrix (of variables) derived from G as follows:

$$A[G](i,j) = \begin{cases} x_{ij} & \text{if } M[G](i,j) = 1 \\ 0 & \text{otherwise} \end{cases}$$

Example:
Here is a bipartite graph $G = (U, V, E)$, where $|U| = |V| = 4$:

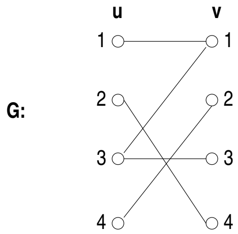

Figure 14.1: Example of a Bipartite Graph

From this graph, G, in Figure14.1, we see that $E = \{(u_1, v_1), (u_2, v_4), (u_3, v_1), (u_3, v_3), (u_4, v_2)\}$, so:

$$A[G] = \begin{bmatrix} x_{11} & 0 & 0 & 0 \\ 0 & 0 & 0 & x_{24} \\ x_{31} & 0 & x_{33} & 0 \\ 0 & x_{42} & 0 & 0 \end{bmatrix}$$

Theorem 14.1 (Edmonds' Theorem) *A bipartite graph G contains a perfect matching if and only if* $\det(A[G]) \not\equiv 0$.

Proof:

Definition 14.2 *A* permutation *σ of the set of integers $\{1, 2, \ldots, n\}$ is an arrangement of these integers in some order without omissions or repetitions.*

Example: There are six different permutations of the set of integers $\{1, 2, 3\}$. These are:

$$(1, 2, 3), (2, 1, 3), (3, 1, 2), (1, 3, 2), (2, 3, 1), \text{and } (3, 2, 1)$$

If we exchange two arbitrary numbers i and j in a permutation and keep the other numbers unchanged we get a new permutation. This is called a *pairwise transposition.*

Definition 14.3 *If we let $c(\sigma)$ be the minimum number of pairwise transpositions needed to construct a permutation σ, then:*

$$sgn(\sigma) = \begin{cases} 1 & \text{if } c(\sigma) = 0 \ (mod\ 2) \\ -1 & \text{if } c(\sigma) = 1 \ (mod\ 2) \end{cases}$$

Example: For the permutation $\sigma(i) = (2, 3, 1, 5, 4, 6)$, shown below, we have $c(\sigma) = 3$, and $sgn(\sigma) = -1$.

i	$\sigma(i)$
1	2
2	3
3	1
4	5
5	4
6	6

Definition 14.4 *Let A be a square matrix. The* determinant function *is denoted by* det*, and we define* $\det(A)$ *to be the sum of all signed elementary products from A, thusly:*

$$\det(A) = \sum_{\sigma} \left(sgn(\sigma) \prod_{i=1}^{n} A(i, \sigma(i)) \right)$$

Example: For a 3×3 matrix

$$M = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

and

$$\sigma_1 = (1,2,3), \sigma_2 = (1,3,2), \sigma_3 = (2,1,3), \sigma_4 = (2,3,1), \sigma_5 = (3,1,2), \sigma_6 = (3,2,1)$$

We have:

$$\det(M) = aei - afh - bdi + bfg + cdh - ceg$$

Let $A(G)$ be the $n \times n$ matrix obtained from a bipartite graph $G = (U, V, E)$, where $|U| = |V| = n$. Since each indeterminate variable x_{ij} occurs at most once in $A(G)$ there can be no cancellation of items in the summation. Therefore the determinant is not identically zero if and only if there is a permutation σ for which the corresponding term in the summation is non-zero. The latter happens if and only if each of the entries $A[G](i, \sigma(i))$, for $1 \le i \le n$, is non-zero. This is equivalent to having a perfect matching(the one corresponding to σ) in G.

Thus :
no perfect matching $\Longrightarrow$ every term $\equiv 0$;
no terms can cancel;
perfect matching $\Longrightarrow \det[A[G]] \not\equiv 0$.
■

We can now construct a simple randomized test for the existence of a perfect matching.

One obvious way to check if there is a perfect matching is to calculate the determinant symbolically. The problem is that the number of permutations is exponential. That is, computing the determinant is actually the same as looking at every possible perfect matching. In the following, we'll use Schwartz's algorithm to determine whether or not $\det(A[G])$ is equivalent to zero. This approach is much faster.

Algorithm:
Input: $G = (U, V, E)$

> Let $M = A[G]$
> Let $S = \{0, \pm 1, \pm 2, \cdots, \pm n\}$, where n is the degree of the polynomial
> $\forall i, j$ s.t. x_{ij} appears in M, set x_{ij} = some random element in S.
> Evaluate $\det(M)$,
> **if** $\det(M) = 0$, output "no perfect matching"
> **otherwise** output "perfect matching"

If there is no perfect matching then $Pr(\text{correct}) = 1$. If there is a perfect matching, then $Pr(\text{correct}) = 1 - \frac{n}{2n+1} > \frac{1}{2}$. We can keep repeating the runs and each run is independent, so that eventually we can acheive an arbitrarily small probability of error.

Run time of $\det(M)$.

We said earlier that computing $\det(M)$ symbolically is exponential. When substituting numbers in the matrix there turns out to be a much faster way. Roughly, the running time is the same as that of matrix multiplication. That is, suppose $\mu(n)$ is the speed of matrix multiplication, then the running time is: $O(\mu(n)) = O(n^{2.37})$.

For a non-random (deterministic) algorithm, the running time is: $O(m\sqrt{n})$. When the graph is dense, it can be $O(n^{2.5})$.

Therefore, asymptotically speaking, the random algorithm is faster than the non-random algorithm. Its advantages in practice are:

- It is simple and straightforward. There are plenty of numerical packages that will compute the determinant of a matrix.
- There is an easy way to extend the algorithm from a bipartite graph to an arbitrary graph; using a non-random algorithm to deal with an arbitrary graph is much more complicated and slower.

14.2 Primality Testing

In the following, we'll look at more random algorithms. One is primality testing. The problem is:

Input: n.
Question: Is n prime?

14.2.1 Algorithm 1

Test every integer in $[2, \cdots, \sqrt{n}]$ to see if it divides n.
Problem: Because the input size is $\lceil logn \rceil$, the running time is exponential.

14.2.2 Algorithm 2

The basis of Algorithm 2 is the following theorem from number theory.

Theorem 14.5 (Fermats Little Theorem) *If n is prime then $\forall a \in [1, n], a^{n-1} \equiv 1 \pmod{n}$.*

Algorithm 2:

> Choose a uniformly random from $[1, n]$.
> **if** $a^{n-1} \not\equiv 1 \pmod{n}$, then n is composite.
> **otherwise** n is prime.

Run time: Polynomial because $a^{n-1} \bmod n$ can be computed in time $O(log^c n)$ for some constant c. This is left as an exercise.

For most numbers the algorithm returns the correct answer, but for some the answer is wrong. For example, for *Carmichael numbers*, which are specific composites such that $a^{n-1} \equiv 1 \pmod{n} \forall a \in [1, n]$, the algorithm does not return the correct answer. Carmichael numbers are rare. The percentage of Carmichael numbers goes to 0 as n goes to infinity. The first 5 Carmichael numbers: 561, 1105, 1729, 2465, and 2821.

The trouble with this algorithm is that the probablity of a correct answer depends on the input, which is not appealing. We'll see another better algorithm that is based on the Miller-Rabin test in the following.

Miller-Rabin test: [CLR90, Chapter 33]
If n is composite then for at least half of the $a \in (1, n)$, either $a^{n-1} \not\equiv 1 \pmod n$ or $\exists k$, s.t. $r = \frac{n-1}{2^k}$ is an integer and $1 < \gcd(a^r - 1, n) < n$.

Running time: Polynomial. We know that checking if $a^{n-1} \neq 1 \pmod n$ is polynomial, that $k \leq \log n$, and that the gcd part is also polynomial.

14.2.3 Algorithm 3

repeat
  Pick $a \in (1, n)$ uniformly at random.
  if the Miller-Rabin test says n is composite, return "composite" and stop
until the chance of a composite surviving is neligible.
return "prime."

After t tries, if n is prime the algorithm returns "prime," and if n is composite then the probability that the algorithm returns "prime" is at most 2^{-t}.

14.2.4 Algorithm 4

In addition to the Miller-Rabin test we have the **Adelman-Huang test** (not described here) which also takes polynomial time. It is sort of opposite to Miller-Rabin test in terms of performance, that is, if n is composite then the test always returns "composite", whereas if n is prime then the test returns "composite" with probability $\leq \frac{1}{2}$.

So, how can we combine the two tests into one single algorithm?

Algorithm:
  Use Miller-Rabin test. If it returns composite, then return "composite" and stop.
  Use Adelman-Huang Test. If it returns prime, then return "prime" and stop.
  Repeat the two tests. Eventually, one of them will give us the correct answer.

Running time: is random, so this is a Las Vegas algorithm.

References

A81 H. Anton, *Elementary Linear Algebra*, 1981, pp. 57–63.

CLR90 T. Cormen, and C. Leiserson and R. Rivest, *Introduction to Algorithms*, MIT Press, 1990.