Advanced Graph Algorithms Jan-Apr 2014

Lecture 21 — March 26, 2014

Lecturer: Saket Saurabh *Scribe: Diptapriyo Majumdar*

1 Overview

In the last few lectures, we had learnt how to solve matchings in graphs by using rank of a matrix, determinant of a matrix and also matroid parity.

From this lecture we have started a new module. In this lecture, we will see how to multiply two matrices, multiply two boolean matrices, computing transitive closure of a graph, solving *max-cut* problem by using matrix multiplication.

2 Matrix Multiplication Algorithms

Two matrices A and B can be multiplied only when number of rows of B and number of columns of A match. More importantly Matrix Multiplication is not necessarily commutative. Given two matrices $A_{m\times n}$ and $B_{n\times p}$, product of these two matrices $C_{m\times p} = A.B$.

$$\forall i, j \in [n] : C_{i,j} = \sum_{k=1}^{n} A_{i,k}.B_{k,j}$$

2.1 Naive Algorithm to multiply two matrices

Let we are multiplying two matrices $A_{n\times n}$ and $B_{n\times n}$. In order to compute $C_{i,j}(C = A.B)$ for all $i, j \in [n]$, we have to compute $A_{i,1}.B_{1,j}, A_{i,2}.B_{2,j}, \ldots, A_{i,n}.B_{n,j}$. So, we need n multiplications and we have to compute $\sum_{k=1}^{n} A_{i,k}.B_{k,j} = C_{i,j}$. So, we also need n additions. Now, C has n^2 entries. So, it requires n^3 multiplications. Therefore running time of this algorithm is $O(n^3)$.

Algorithm 1 Calculate $C = A.B$

```
for i = 1, 2, ..., n do
  for j = 1, 2, ..., n do
    C[i, j] ← 0
    for k = 1, 2, ..., n do
      C[i, j] = C[i, j] + A[i, k] * B[k, j]
    end for
  end for
end for
```

2.2 Strassen's Algorithm for Matrix Multiplication

Strassen came up with an idea to multiply two matrices $A_{2\times 2}$ and $B_{2\times 2}$. Then, he extended this idea to multiply two $A_{n\times n}$ and $B_{n\times n}$ using divide and conqueror approach.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ B_{11} & B_{12} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C = A.B = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

where

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Now, Strassen observed the following things. He defined $P_1, \ldots, P_7$ as follows.

$$\begin{aligned} P_1 &= A_{11}(B_{12} - B_{22}) = A_{11}B_{12} - A_{11}B_{22} \\ P_2 &= (A_{11} + A_{12})B_{22} = A_{11}B_{22} + A_{12}B_{22} \\ P_3 &= (A_{21} + A_{22})B_{11} = A_{21}B_{11} + A_{22}B_{11} \\ P_4 &= A_{22}(B_{21} - B_{11}) = A_{22}B_{21} - A_{22}B_{11} \\ P_5 &= (A_{11} + A_{22})(B_{11} + B_{22}) = A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} \\ P_6 &= (A_{12} - A_{22})(B_{21} + B_{22}) = A_{12}B_{21} + A_{12}B_{22} - A_{22}B_{21} - A_{22}B_{22} \\ P_7 &= (A_{11} - A_{21})(B_{11} + B_{12}) = A_{11}B_{11} + A_{11}B_{12} - A_{21}B_{11} - A_{21}B_{12} \end{aligned}$$

Now, we can see the followings:

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} = P_5 + P_4 + P_6 - P_2 \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} = P_1 + P_2 \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} = P_3 + P_4 \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} = P_5 + P_1 - P_3 - P_7 \end{aligned}$$

We can see that it needs 7 multiplication operations which is better than the naive algorithm that requires 8 multiplications. But, strassen's method compromises since it requires more additions.

But, still this method is better since multiplication is more expensive(since its not a primitive operation) than addition(which is a primitive operation) operation.

Now, to use this to multiply two $n \times n$ matrices, we use it as follows.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Now, $A_{11}, B_{11}, A_{12}, B_{12}, A_{21}, B_{21}, A_{22}, B_{22}$ are all $\frac{n}{2} \times \frac{n}{2}$ matrices. Now, we recursively compute $P_1, \ldots, P_7$ and compute $C_{11}, \ldots, C_{22}$ accordingly. Algorithm is as follows.

Algorithm 2 Calculate $C = A.B$

Divide A into 4 equal parts $A_{11}, A_{12}, A_{21}, A_{22}$
Divide B into 4 equal parts $B_{11}, B_{12}, B_{21}, B_{22}$
Compute $P_1 = A_{11}.(B_{12} - B_{22})$
Compute $P_2 = (A_{11} + A_{12}).B_{22}$
Compute $P_3 = (A_{21} + A_{22}).B_{11}$
Compute $P_4 = A_{22}.(B_{21} - B_{11})$
Compute $P_5 = (A_{11} + A_{22}).(B_{11} + B_{22})$
Compute $P_6 = (A_{12} - A_{22}).(B_{21} + B_{22})$
Compute $P_7 = (A_{11} - A_{21}).(B_{11} + B_{12})$
Compute $C_{11} = P_5 + P_4 + P_6 - P_2$
Compute $C_{12} = P_1 + P_2$
Compute $C_{21} = P_3 + P_4$
Compute $C_{22} = P_5 + P_1 - P_3 - P_7$
Output C

Since first 7 lines of this algorithm are recursive calls, therefore it recursively performs 7 matrix multiplication operations each of size $\frac{n}{2} \times \frac{n}{2}$ that computes $P_1, \ldots, P_7$ and outputs all elements of C. Last 4 lines of the algorithm performs matrix additions which take $O(n^2)$ time.

$$T(n) = 7T(\frac{n}{2}) + O(n^2)$$

Solving this recurrence, we get $T(n) = O(n^{2.81})$. Hence, strassen's matrix multiplication is an improvement to the naive matrix multiplication algorithm.

Note: Strassen's Matrix Multiplication idea works for Rings. It does not work for arbitrary structure.

Recent Devolopment: But, there is further improvement by Coppersmith[1] and Winograd. This algorithm runs in $O(n^\omega)$ time where $\omega = 2.376$.

3 Boolean Matrix Multiplication

Given two matrices whose entries are only 0 and 1. We define two operations $\wedge, \vee, \neg$ where for any boolean variable a, $a \vee 1 = 1, a \wedge 0 = 0, \neg(1) = 0, \neg(0) = 1$. Let $A_{n \times n}, B_{n \times n}$ be two boolean

matrices. $C = A.B$. Then, we have

$$\forall i, j \in [n] : C_{i,j} = \vee_{k=1}^{n}(A_{i,k} \wedge B_{k,j})$$

Now, if we again use the analogous naive algorithm which substitutes multiplication of two numbers by AND operation of two bits and addition of two numbers by OR of two bits, then it requires $O(n^3)$ boolean operations.

3.1 Strassen's Matrix Multiplication Idea in Boolean Matrix Multiplication

Now, we can't directly use strassen's matrix multiplication idea to multiply boolean matrices since 1 has no inverse in $\{0, 1\}$ under $\vee$ operation. So, we need to do the following.

Whenever, it requires to do $\wedge$ operation, then we need to do multiplication. Otherwise we replace $a \vee b$ by $(a+b)\text{mod}(n+1)$ and check if the result is non-zero. If it is zero, then we put 0, otherwise we put 1. Pseudocode of the naive algorithm is as follows.

Algorithm 3 Calculate $C = A.B$

```
for i = 1, 2, ..., n do
  for j = 1, 2, ..., n do
    C[i, j] ← 0
    for k = 1, 2, ..., n do
      C[i, j] = C[i, j] + A[i, k] * B[k, j]
    end for
    x = C[i, j]mod(n + 1)
    if x is zero then
      C[i, j] = x
    else
      C[i, j] = 1
    end if
  end for
end for
```

Now, we can easily see that we can use Strassen's Matrix Multiplication algorithm to compute the product of two boolean matrices since now we are working over rings.

3.2 Computing A^n where $A_{n\times n}$ is a boolean matrix

We do it as follows.

We clearly see that this algorithm requires $O(log_2 n)$ matrix multiplications. At each time, we output the intermediate matrix products which takes $O(n^2)$ time and one matrix multiplication operation takes $O(n^{2.81})$ time if we use strassen's matrix multiplication algorithm. Therefore, running time of this algorithm is $O(n^{2.81} log_2 n)$.

Algorithm 4 Calculate A^n

```
if n is even then
    C ← A^(n/2)
    return C.C;
else
    C ← A^((n-1)/2)
    return A.C.C
end if
```

Using Coppersmith's Algorithm will take $O(n^{\omega} log_2 n)$ time. In the next lecture, we will see how to compute A^n where n is boolean in $O(n^{\omega})$ time where coppersmith's algorithm is used as a black-box.

4 Max Cut in an undirected graph

Now, we will see the max cut problem in an undirected graph.
Cut: Cut in a graph is defined as an order partition $(S, V \setminus S)$.
We denote a cut of a graph as S where $S \subseteq V$.
Weight of a cut is $W(S) = \sum_{(u,v) \in E : u \in S, v \in V \setminus S} w(u,v)$ where $w(u,v)$ is the weight of the edge $(u,v) \in E$. Problem statement is as follows.

Input: $G = (V, E)$ an undirected graph and $w : E \to R^+$
Goal: Find a cut of maximum weight. More precisely, find $S \subseteq V$ such that $W(S)$ is maximized.

Number of possible cuts in a graph: There are n vertices. Each of which is either there in the subset $S \subseteq V$ or not there in the subset $S \subseteq V$. Therefore, number of possible cuts equals 2^n.

Now, as a naive algorithm, we can check with all possible cuts of G and return the cut with maximum weight. It takes $O(2^n.poly(n,m))$ time.

Now, in order to solve max-cut problem, we will use $findingMaxWtTriangle$ algorithm as a subroutine. $findingMaxWtTriangle$ is an algorithm that identifies a triangle in an weighted undirected graph and that triangle has the highest weight among all triangles in the graph. $findMaxWtTriangle$ algorithm runs in in $O(W.n^{\omega})$ time where W is the weight of that maximum weighted triangle.

Now, we are given an undirected graph $G = (V, E)$. We create a new graph $P(G) = (P(V), P(E))$ and $y : P(E) \to R^+$ where $\grave{cup}$ denotes disjoint union

$$V = V_1 \cup V_2 \cup V_3$$

$$|V_1| = |V_2| = |V_3| = \frac{|V|}{3}$$

$$P(V) = P(V_1) \cup P(V_2) \cup P(V_3)$$

$$P(V_i) = \{(S, V_i \setminus S) | \text{ S is a cut of } V_1\} \forall i = 1, 2, 3$$

$$P(E) = (P(V_1) \times P(V_2)) \cup P(V_1) \times P(V_3)) \cup (P(V_2) \times P(V_3))$$

$$y((S, V_i \setminus S), (T, V_j \setminus T)) = r + \frac{1}{2}(p + q)$$

$$p = \sum_{(a,b):a \in S, b \in V_i \setminus S} w(a, b)$$

$$q = \sum_{(a,b):a \in T, b \in V_j \setminus T} w(a, b)$$

$$r = \sum_{(a,b):(a \in S, b \in V_j \setminus T) \vee (a \in V_i \setminus S, b \in T)} w(a, b)$$

Number of triangles in $P(G)$: A triangle can exist only when one of its vertices is in $P(V_1)$, one of its vertices is in $P(V_2)$ and the other is in $P(V_3)$. So, we have $2^{\frac{n}{3}}$ choices for a vertex from V_1. For each of them, we have $2^{\frac{n}{3}}$ choices for a vertex from V_2. And for each of them, we have $2^{\frac{n}{3}}$ choices for a vertex from V_3. Therefore, total number of possible triangles in $P(G)$ is $(2^{\frac{n}{3}})^3 = 2^n$ and it equals the number of cuts of G.

So, we can establish a bijection between the set of triangles in $P(G)$ and the set of cuts in G. Moreover, weight of a triangle is equal to the weight of the corresponding cut of G. So, max weighted triangle in $P(G)$ is the max cut on G.

Now, our algorithm just finds a triangle with maximum weight in $P(G)$. That gives the corresponding cut of that in G.

Algorithm 5 Calculate $maxCut(G = (V, E))$

Partition $V = V_1 \cup V_2 \cup V_3$ arbitrarily
Construct $P(G) = (P(V), P(E))$
Compute $P(T) = findMaxWtTriangle(P(G))$
Output the cut corresponding to the triangle $P(T)$

$P(G)$ contains $3.2^{\frac{n}{3}}$ vertices and $poly(3.2^{\frac{n}{3}})$ edges. Construction of $P(G)$ takes $O(poly(3.2^{\frac{n}{3}}))$ time. Now, computing maximum weighted triangle takes $O(W.(3.2^{\frac{n}{3}})^{\omega}.poly(n, m))$ time where W is the weight of the max cut in the original graph. Therefore, our algorithm solves max cut problem in $O(2^{\frac{n\omega}{3}}.poly(n, m))$ time which is better than $O(2^n.poly(n, m))$.

References

[1] Don Coppersmith, Shmuel Winnogard: *Matrix Multiplication via Arithmetic Progressions*, Journal of Symbolic Computation (1990) 9, 251-280.