

Rainbow Perfect Matchings

Mats Rydberg & Martin Larsson

October 15, 2012

Algorithm

1. Fix prime $p \gg n$ ¹
2. For each colour $C \in [n]$
 Construct $n \times n$ matrix m_C
 For each $uv \in E$ with $c(uv) = C$
 Pick random integer $r \in (0, p]$
 Set $m_C[u, v] = r$
3. Set $B = \sum_{i=0}^{n-1} m_i$
4. Compute $d_B = \det(B) \bmod p$
5. If $d_B = 0$ return "no"
6. Else set $sum = 0$
7. For each $X \subset [n]$
 Initialize $M = \mathbf{0}$ ²
 For each $C \in X$
 Set $M = M + m_C$
 Set $sum = sum + (-1)^{n-|X|-1} \cdot \det(M) \bmod p$
8. If $d_B - sum = 0$ return "no" else return "yes"

¹ In our case, we selected $p = 32749$.

² $\mathbf{0}$ is the $n \times n$ all-zeroes matrix.

Our algorithm modifies the given Algorithmic Piece 1 on step 2, by creating n matrices, one for each colour. But in step 3 we combine them into the biadjacency matrix B (called A_G in the assignment) and do the same end condition for its determinant.

For Algorithmic Piece 2, we have modified the pseudo code to be defined via matrix sums instead. We construct the biadjacency matrix for the current set of colours by simple addition, and compute the determinant for every such matrix. We are not including $\det(B)$ in sum , so to get the signs right we subtract an extra 1 in the exponent for -1 .³ This is really just an optimization, as we could remove steps 3 and 4 and have Algorithmic Piece 2 more or less intact. Our algorithm is faster for "no"-instances, however. With this change in mind, the logic is the same as in the assignment for why a non-rainbow perfect matching will eliminate itself in the calculation of sum .

³ Another way to fix this would be to compute $d_B + sum$ in step 7 of the algorithm, but we thought this was cleaner.

Running Time

In our calculations, we define the basic, constant-time operations to be scalar additions and scalar multiplications. Any other operation we assume to take no time.

We are only interested in a running time bound with respect to n , hence we assume a worst-case $m = n^2$, and do not include m in our O -notation.

Step by step, we pay as follows:

1. No time.

2. n repetitions of

No time.

Worst case n edges of colour C from all n nodes $\Rightarrow T(n) = n^2$ repetitions of

No time.

No time.

3. $T(n) = n^3$ additions $\Rightarrow T(n) = n^3$

4. $T(n) = \frac{2}{3} \cdot n^3 \Rightarrow T(n) = \frac{2}{3}n^3$ ⁴

5. No time.

6. No time.

7. There are 2^n subsets to a set of n elements $\Rightarrow T(n) = 2^n - 2$ (the empty set and the full set) repetitions of

No time.

On average for each X , $T(n) = \frac{1}{2}n$ ⁵ repetitions of n^2 additions $\Rightarrow T(n) = n^2$

1 addition, worst case $n - 2$ multiplications of -1 , 1 multiplication and $T(n) = \frac{2}{3}n^3$ cost for determinant calculation gives $T(n) = 1 + (n - 2) + 1 + \frac{2}{3}n^3 = \frac{2}{3}n^3 + n$

8. 1 addition $\Rightarrow T(n) = 1$

This results in final running time of

$$T(n) = 0 + n(0 + n^2(0 + 0)) + n^3 + \frac{2}{3}n^3 + 0 + 0 + (2^n - 2)(0 + \frac{1}{2}n(n^2) + \frac{2}{3}n^3 + n) + 1$$

$$= \frac{5}{3}n^3 + (2^n - 2)(\frac{7}{6}n^3 + n) + 1$$

$$= \frac{5}{3}n^3 + 1 - \frac{7}{3}n^3 - 2n + 2^n(\frac{7}{6}n^3 + n)$$

⁴ Knowing our determinant calculation implements Gaussian elimination, this is the number of operations required. Normally, this would have exponential bit complexity, but our calculations are made modulus p , and are thus constant in bit complexity.

⁵ This follows from how frequent a given size of an element in the power set of $[n]$ is.

$$= 1 - 2n - \frac{2}{3}n^3 + 2^n(\frac{7}{6}n^3 + n)$$

The dominant factor here is, as expected, exponential in n . Thus we do have $O^*(2^n)$. We could also present a running time bound of $O(n^3 \cdot 2^n)$, in which we consider all factors with an exponential in n relevant, and take the largest-degree polynomial of this factor as our bound.

The exponential 2^n factor come from the fact that we are traversing the power set of an n -sized set. The polynomial n^3 factor comes from our determinant calculation, which runs this fast. Thus, the $O(n^3 \cdot 2^n)$ really summarizes our algorithm in a nice way; we do calculate a determinant for a matrix corresponding to every element of the power set of $[n]$.

Failure bound

Since we are working with random numbers, there is always some cases in which these random numbers doesn't work as we expect them to. In particular, two random numbers may become the same. This will provide our algorithm with false negatives, which's frequency we want to investigate.

First, we conclude that if there is exactly one Perfect Rainbow Matching, there will be no false negative. This follows because p is a prime, and $d_B - \text{sum}$ will consist of only one monomial x , which is a product of random numbers and hence it can not be prime, so $x \bmod p \neq 0$.

Second, if we have $k > 1$ Perfect Rainbow Matchings, we will have k monomials in $d_B - \text{sum}$, each of degree d such that $2 \leq d \leq n$, depending on how many edges the PRM represented by k_i shares with its counterpart k_j . Note that the minimum bound on d can not be 1, since we are not allowing any two edges uv and $(uv)'$ to have the same colour. The sum of the monomials may be equal to zero, in fact the probability that they are follows directly from the Schwartz-Zippel lemma as⁶

$$\Pr[f(r_1, r_2, \dots, r_m) = 0] \leq \frac{d}{|S|}$$

where f is the sum of our k monomials, with random numbers r_1 through r_m (in general) taken from the finite interval $[1, p-1] \Rightarrow |S| = p-1-1+1 = p-1$. In the worst case, we have $d = n$ which gives us the probability of $f = 0$ as

$$\frac{d}{|S|} \leq \frac{n}{p-1}$$

Analysis

Lol

The files in the data directory are:

three.txt The 4-vertex graph from Fig. 1.

Report part maybe

Transition probabilities

The transition matrix for the graph described in three.txt is⁷

$$P = \begin{pmatrix} 1 & 6 & \pi & 1 \\ 1 & 1/e & -2 & \cdots \\ 1 & 1 & 0 & \\ \vdots & & & \end{pmatrix},$$

text is text is text is text

three.txt	2 (36.6%)	1 (27.5%)	0 (18.4%)	3 (17.3%)
tiny.txt	[...]			
medium.txt				
wikipedia.txt				
p2p-Gnutellao8-mod.txt				

Implementation

This is our implementation of our Perfect Rainbow Matching algorithm. It is only sparsely commented, but we expect its behaviour to be fairly straight-forward. We have tried to follow our pseudo-code verbatim, putting aside Java programming practices to some extent.

```
package p1;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Random;
import java.util.Scanner;

/**
 * For all user interaction. Reads input, posts error messages, etc.
 */
```

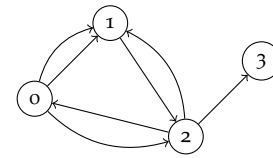


Figure 1: A directed multigraph.

⁷ Fill in the right values. Set $\alpha = \frac{85}{100}$.

```

* @author dt08mr7
* @author dt08ml5
*
*/
public class Main {
    private static final int PRIME = 32749; // largest prime smaller than 2^15

    public static void main(String[] args) throws FileNotFoundException {
        // Time measurement
        long start = System.currentTimeMillis();

        // 1
        int prime = PRIME;

        Scanner scan = new Scanner(new FileReader(args[0]));
        int n = scan.nextInt();
        init_inverses(prime);
        int m = scan.nextInt();

        ArrayList<HashSet<Integer>> colourSets =
            new ArrayList<HashSet<Integer>>();
        ArrayList<int[][]> colourMatrices = new ArrayList<int[][]>();
        HashSet<Integer> allColours = new HashSet<Integer>();
        for (int i = 0; i < n; i++) {
            colourMatrices.add(new int[n][n]);
            ArrayList<HashSet<Integer>> newColourSets =
                new ArrayList<HashSet<Integer>>();
            for (HashSet<Integer> set : colourSets) {
                HashSet<Integer> combination = new HashSet<Integer>(set);
                combination.add(i);
                newColourSets.add(combination);
            }
            HashSet<Integer> aloneColourSet = new HashSet<Integer>();
            aloneColourSet.add(i);
            allColours.add(i);
            colourSets.add(aloneColourSet);
            colourSets.addAll(newColourSets);
        }
        colourSets.remove(allColours);

        // 2
        Random rand = new Random();
        for (int i = 0; i < m; i++) {
            int from = scan.nextInt();
            int to = scan.nextInt();
            int colour = scan.nextInt();
            colourMatrices.get(colour)[from][to] = rand.nextInt(prime-1) + 1;
        }

        // 3
        int[][] allColourMatrix = new int[n][n];
    }
}

```

```

for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            allColourMatrix[j][k] = (allColourMatrix[j][k] +
                                     colourMatrices.get(i)[j][k]) % prime;
        }
    }
}

// 4
int dB = 0;
dB = determinant(allColourMatrix, prime);

// 5
if (dB == 0) {
    System.out.println("Running time (ms): " +
                       (System.currentTimeMillis() - start));
    System.out.println("No");
    return;
}

// 6
int sum = 0;

// 7
for (HashSet<Integer> X : colourSets) {
    int[][] M = new int[n][n];
    for (int i : X) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                M[j][k] =
                    (M[j][k] + colourMatrices.get(i)[j][k]) % prime;
            }
        }
    }
    sum += (int) Math.pow(-1, (n - X.size() - 1)) *
           determinant(M, prime);
}

// 8
/*
 * Since the % operator doesn't work as expected for negative numbers
 * in Java, we manually modulo the sum to the correct interval
 */
while (sum < 0) {
    sum += prime;
}
sum = sum % prime;
System.out.println("Running time (ms): " +
                  (System.currentTimeMillis() - start));
if (dB - sum == 0)

```

```

        System.out.println("No");
    else
        System.out.println("Yes");
}

// Determinant calculation taken from Andreas

static int inv[];

public static int modexp(int a, int e, int p) {
    if (e == 1) {
        return a;
    } else {
        int sq = modexp(a, e / 2, p);
        sq = (sq * sq) % p;
        if ((e & 1) == 1) {
            sq = (sq * a) % p;
        }
        return sq;
    }
}

public static void init_inverses(int p) {
    int i;
    inv = new int[p];
    for (i = 1; i < p; i++) {
        inv[i] = modexp(i, p - 2, p);
    }
}

public static int determinant(int[][] A, int p) {
    int n = A[0].length;
    int i, j, k;
    int rs[], vis[];
    int d = 1;
    int sgn = 0;
    rs = new int[n];
    vis = new int[n];
    for (i = 0; i < n; i++) {
        rs[i] = 0;
    }
    for (i = 0; i < n; i++) {
        int who = -1;
        for (j = 0; j < n; j++)
            if (rs[j] == 0 && A[j][i] > 0) {
                rs[j] = i + 1;
                who = j;
                d = (d * A[j][i]) % p;
                break;
            }
        if (who == -1) {

```

```

        d = 0;
        break;
    }
    for (j = 0; j < n; j++)
        if (rs[j] == 0 && A[j][i] > 0) {
            d = (d * inv[A[who][i]]) % p;

            for (k = n - 1; k >= i; k--) {
                A[j][k] = (A[who][i] * A[j][k] -
                    A[j][i] * A[who][k] + p * p) % p;
            }
        }
    }
    if (d > 0) {
        sgn = n;
        for (i = 0; i < n; i++) {
            vis[i] = 0;
        }
        for (i = 0; i < n; i++) {
            if (vis[i] == 0) {
                sgn--;
                j = i;
                while (vis[j] == 0) {
                    vis[j] = 1;
                    j = rs[j] - 1;
                }
            }
        }

        if ((sgn & 1) == 1)
            d = p - d;
    }
    return d;
}
}

```

References

- [1] Wikipedia, the Free Encyclopedia, *Power set*, http://en.wikipedia.org/wiki/Power_set#Relation_to_binomial_theorem, 10 October 2012 05:20
- [2] Wikipedia, the Free Encyclopedia, *Gaussian elimination*, http://en.wikipedia.org/wiki/Gaussian_elimination#Analysis, 29 September 2012 03:39
- [3] Wikipedia, the Free Encyclopedia, *Schwartz–Zippel lemma*, http://en.wikipedia.org/wiki/Schwartz%E2%80%93Zippel_lemma#Schwartz.Zippel_lemma, 3 October 2012 15:22