[CS1961: Lecture 1] Introduction, Basic Counting Techniques

Instructor: Chihao Zhang; Scribed by Yuchen He

September 19, 2022

1 Introduction to Combinatorics in Computer Science

Combinatorics is an area of mathematics mainly focusing on discrete objects. In Combinatorics, one often asks questions about

- existence / construction of objects satisfying certain properties;
- enumeration / counting objects satisfying certain properties;
- and picking the *best* object satisfying certain properties.

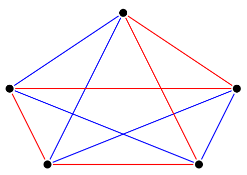
In this course, we will mainly focus on the first two classes of problems, namely existence / construction and enumeration / counting. The third, frequently referred to as combinatorial optimization, is nearly identical algorithm design and will be covered in future courses. We will examine some typical examples today, and demonstrate their connections with computer science.

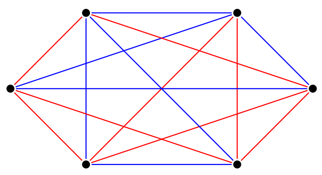
1.1 Existence/Construction

1.1.1 Ramsey Number

A typical problem in extremal combinatorics is the study of the Ramsey number. It states the a fact that for any six people, we can find three among them who either know each other or do not know each other.

In the language of graph theory, we can use a complete graph to describe the acquaintanceship between these people. Each edge is dyed red or blue to represent whether the two ends are acquaint with each other or not respectively. The Ramsey number $R(s,t)$ is the minimum vertices number of a complete graph such that there exists a blue clique of size s or a red clique of size t.


$R(3,3) > 5$


$R(3,3) = 6$

One can verify that $R(3,3) = 6$ by hand. For general s and t, we do not know the Ramsey number exactly. It is not hard to show that $R(s,t)$ is always finite. We will give a lower bound of $R(s,t)$ using the probabilistic method in the course.

1.1.2 Lovász Local Lemma

An important decision problem in computer science is **SAT**. Given a CNF ϕ,[1], the problem asks whether ϕ is satisfiable.

[1] A conjunction normal form (CNF) ϕ is a propositional formula in form like $\phi = C_1 \wedge C_2 \wedge \cdots C_m$ where each clause C_i is a disjunctive of several variables.

The Lovász Local Lemma gives a sufficient condition for a formula ϕ to be satisfiable.

Lemma 1 (Lovász Local Lemma) *If the CNF ϕ satisfies the following three conditions:*

- *ϕ is k-regular;* [2]
- *each variable appears in at most d clauses;*
- *$d < \frac{2^k}{ek}$;* [3]

[2] This means each clause of ϕ contains k variables. In other words, ϕ is a k-CNF.

[3] e is the base of natural logarithms.

Then ϕ is satisfiable.

Lemma 1 is a statement about the existence of feasible solutions. How to construct such a solution? In computer science, one often asks for *efficient* algorithms, which usually indicate those algorithms who can find a solution in polynomial-time. Clearly one can enumerate every possible assignments of a CNF ϕ to find a solution. However, this may need $2^{O(n)}$ (n is the number of variables) trials and is inefficient.

In a promient work, Robin Moser and Gábor Tardos proposed an algorithm to find a feasible solution and proved its efficiency under conditions similar to Theorem 1. The algorithm starts by assigning random values to the variables in ϕ. We denote this initial assignment by σ. In each round, if ϕ is satisfied, we output σ as the feasible solution and end the algorithm. Otherwise, pick an arbitrary unsatisfied clause, resample the variables in it and repeat. This algorithm only needs expected linear time if ϕ satisfies the conditions in Lemma 1.

We will study the analysis of this algorithm in later courses.

1.2 Enumeration/Counting

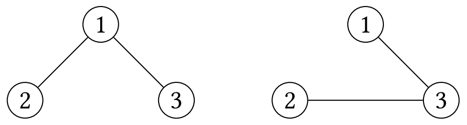
1.2.1 Distinct Labeled trees

Suppose there are n labeled vertices. How many distinct labeled trees can we construct? The Cayley's formula states that this number is n^{n-2}. Note that this result is very concise that we can immediately get the answer given any n. We call this a closed form solution[4].

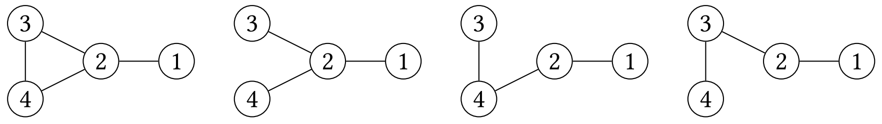
[4] Closed form is a mathematical expression which usually contains only basic operations and finite variables.

1.2.2 Spanning trees

Given a connected graph G, we can always generate a tree by removing some edges and this is called the spanning tree of G. Note that spanning tree may not be unique. For example, the following graph has three spanning trees in total.



Two distinct labeled trees



Let A and D be the adjacent matrix and degree matrix of G respectively. Then the corresponding A and D for the above graph is:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}, \quad D = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}.$$

To count the total number of spanning trees, we have the following theorem.

Theorem 2 (Kirchhoff's Theorem) *The number of spanning trees equals to* $\det[D-A]_{ii}$ *where* $[D-A]_{ii}$ *is the minor of the element at line i and column i for any i.*

For an $n \times n$ matrix B, its determinant $\det(B)$ is defined as $\sum_{\sigma \in S_n} (-1)^{\|\sigma\|} \prod_{i=1}^{n} B(i, \sigma(i))$ where S_n is all the permutations of $[n]$ and $\|\sigma\|$ is the parity of σ. The determinant can be calculated in polynomial time by Gaussian elimination.

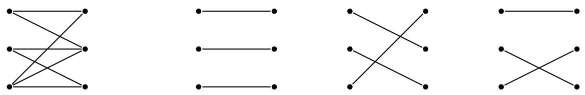
1.2.3 Perfect Matchings in Bipartite

Given a bipartite graph with n vertices and its adjacent matrix A, its perfect matching number $\text{per}(A)$ can be calculated by

$$\text{per}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} A(i, \sigma(i).$$

Although this formula looks like the definition of determinant, it is much harder than computing determinant. Actually, computing the number of perfect matchings on bipartite graphs is #P-complete [5].

[5] Informally, #P problems can be defined as those counting the number of solutions of NP problems. Thus, #P-complete problems is at least as hard as NP-compelte problems



A Bipartite and Its Perfect Matchings

2 Basic Counting Techniques

2.1 Basic Counting Techniques

Suppose we are setting a password with length k. Each character of the password can be chosen from $\{A, B, \ldots, Z, a, b, \ldots, z, 0, 1, \ldots, 9\}$. Consider the following four situations.

(1) In the normal setting, since every character has 62 choices, there are 62^k ways to set the password. More generally, we will have n^k possible password sequences if there are n options for each character.

(2) If it does not allow repetition, the subsequence will have smaller optional scope once the preceding characters are fixed. In this situation, there are $(n)_k$ possible ways to set the password.

$(n)_k \triangleq n(n-1)\cdots(n-k+1)$

(3) Based on (2), we further assume that the order of the sequence does not matter. Then the problem is the same with choosing k numbers from $[n]$ and the number of possible ways is $\binom{n}{k}$.

$\binom{n}{k} \triangleq \frac{n!}{k!(n-k)!}$

(4) If repetition is allowed and sequence order does not matter, it is equivalent to choosing a multi-set of size k from $[n]$. This multi-set number is written as $\left(\binom{n}{k}\right)$.

We can construct a bijection between the set of multi-sets and a subset of $\{0,1\}^{n+k}$. Note that a binary string which ends at 1 and has n 1's and k 0's can be comprehended in the following way. The n 1's represents n types of characters. The number of consecutive 0's immediately before the i-th 1 means the number of occurrence of the i-th character in the corresponding password.[6]

[6] For example, when the characters are chosen from $\{A, B, C, D\}$ and $k = 3$, the password AAC can be encoded as 0011011.

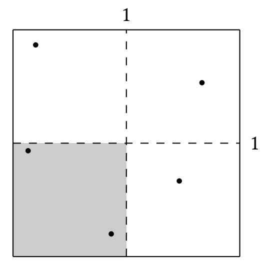
Therefore, the number of different passwords is exactly the number of such binary strings, i.e., $\left(\binom{n}{k}\right) = \binom{n+k-1}{k}$.

2.2 The Pigeonhole Principle

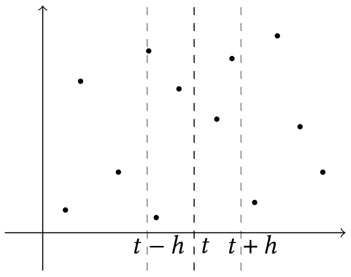
The pigeonhole principle states that when there are more pigeons than the pigeonholes, there must exist some pigeonholes with more than 1 pigeon. It is also called the drawer principle.

Example 1 *Assume* 5 *points are put in a square with side length* 1*. Then there must exist two points with distance no larger than* $\frac{1}{\sqrt{2}}$.

This can be easily proved by the pigeonhole principle. Note that if we divide the square into four smaller ones as the figure shows, the longest segment in each block is the diagonal line whose length is $\frac{1}{\sqrt{2}}$*. By the pigeonhole principle, there must be two points in the same block. That is, there must be two points with distance no larger than* $\frac{1}{\sqrt{2}}$.



Recall the problem of finding the closest pair of points on the plane. Suppose there are n points $v_1, v_2, \ldots, v_n$ in $\mathbb{R}^2$ and the coordinate of v_i is (x_i, y_i) for $i \in [n]$. The distance between two points is defined as the Euclidean distance, i.e., $\text{dist}(v_i, v_j) \triangleq \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. Let $S = \{v_1, v_2, \ldots, v_n\}$. Our aim is to design an algorithm $\mathbf{Find}(S)$ which can output the shortest distance among all the pairs of points. The simplest way is to use brute force, i.e., enumeration method, which leads to a time complexity of $O(n^2)$. A more efficient and frequently used method is divide and conquer.



As a preprocessing step, we sort the points by its abscissa and ordinate respectively for later use. This takes $O(n \log n)$.

The first step is to draw a vertical line $x = t$ which divides the points set S evenly into the right part S_R and left part S_L. With the preprocessing step, this only takes $O(1)$ time. Then we calculate $h_L = \mathbf{Find}(S_L)$ and $h_R = \mathbf{Find}(S_R)$ recursively. Let $h = \min\{h_L, h_R\}$.

The last step is to find the closest pair in $S_M = \{v_i \mid x_i \in [t-h, t+h]\}$ and compare the distance with h. If we can do this in $O(n)$, then the total time cost $T(n)$ will satisfie $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$ and consequentially we have $T(n) = O(n \log n)$.

To achieve this goal, we first sort the points in S_M by its ordinate with the help of preprocesssing in $O(n)$ time. For each $v_j \in S_M$, we check the distances between v_j and $\{v_{j+1}, v_{j+2}, \ldots, , v_{j+7}\}$ respectively[7] and compare these with h. If there is a distance smaller than h, we update h with this smaller distance. When the process ends, we output the current h. This process takes $O(n)$ time.

[7] Without loss of generality, we assume that the subscript of these points is in the order of the ordinate. That is, $y_1 \leq y_2 \leq \cdots \leq y_j \leq y_{j+1} \leq \cdots \leq y_{|S_M|}$.

The remaining thing is to prove the correctness of the last step. That is, we need prove checking only 7 points one time is sufficient. Assume there is a point v_k whose ordinate is no less than v_j and $\text{dist}(v_j, v_k) < h$. Then v_k and v_j must be in a rectangle as the figure shows. With Example 1, we know that there are at most 4 points in the right square or otherwise it will violate the fact that $h \leq \min\{h_L, h_R\}$. Similarly there are at most 4 points in the left part. So the number of such v_k's is no larger than 7.