ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Discrete Mathematics

Course preparation group:
Nguyễn Khánh Phương
Đỗ Phan Thuận
Phạm Quang Dũng
Huỳnh Thanh Bình
Trần Vĩnh Đức
Bùi Quốc Trung
Đinh Viết Sang
Ban Hà Bằng

Contents of Part 1

PART 1
COMBINATORIAL THEORY
(Lý thuyết tổ hợp)

PART 2
GRAPH THEORY
(Lý thuyết đồ thị)

CONTENTS

Introduction to problem

- A problem asking to give a list of all combinations that satisfy a given number of properties is called the combinatorial enumeration problem.
- As the number of combinations to enumerate is usually very large even when the configuration size is not large:
  - The number of permutations of *n* elements is *n*!
  - The number of subset consisting *m* elements taken from *n* elements is *n*!/(*m*!(*n*-*m*)!)

Therefore, it is necessary to have the concept of solving combinatorial-enumeration problems

CONTENTS

Introduction to problem

- The combinatorial enumeration problem is solvable if we can define ***an algorithm*** so that all configurations could be built in turn.
- An enumeration algorithm must satisfy 2 basic requirements:
  - Do not repeat a configuration,
  - Do not miss a configuration.

Algorithm

- All algorithms must satisfy the following criteria:
  - (1) **Input**. The algorithm receives data from a certain set.
  - (2) **Output**. For each set of input data, the algorithm gives the solution to the problem.
  - (3) **Precision**. Each instruction is clear and unambiguous.
  - (4) **Finiteness**. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite (possibly very large) number of steps.
  - (5) **Uniqueness**. The intermediate results of each step of the algorithm are uniquely determined and depend only on the input and the result of the previous steps.
  - (6) **Generality.** The algorithm could be applied to solve any problem with a given form

Comparing Algorithms

- Given 2 or more algorithms to solve the same problem, how do we select the best one?
- Some criteria for selecting an algorithm:
  1) Is it easy to implement, understand, modify?
  2) How long does it take to run it to completion? **TIME**
  3) How much of computer memory does it use? **SPACE**

In this lecture we are interested in the second and third criteria:

- **Time complexity**: The amount of time that an algorithm needs to run to completion
- **Space complexity**: The amount of memory an algorithm needs to run

We will occasionally look at space complexity, but we are mostly interested in time complexity in this course. Thus in this course the better algorithm is the one which runs faster (has smaller time complexity)

3. Generating algorithm

3.1. Algorithm diagram

3.2. Generate basic combinatorial configurations

- Generate binary strings of length *n*
- Generate m-element subsets of the set of *n* elements
- Generate permutations of *n* elements

CONTENTS

1. Introduction to problem
2. Algorithm and Complexity
3. **Generating algorithm**
4. Backtracking algorithm

3.1. Algorithm diagram

The generating algorithm could be applied to solve the combinatorial enumeration problem if the following two conditions are fulfilled:

1) An order can be specified on the set of combinations to enumerate. From there, the first and last configuration could be defined in the order specified.
2) Build the algorithm to give the next configuration of the current configuration (not the final one yet).

The algorithm referred to in condition 2) is called Successive Generation Algorithm

Generate algorithm

```
void Generate ( )
{
  <Build the first configuration>;
  stop=false;
  while (!stop)
  {
        <Print out the current configuration>;
        if (the current configuration is not the final yet)
          Successive_Generation ( );
       else stop = true;
  }
}
```

<Successive_Generation> is the procedure create the next configuration of the current one in the order specified.

Note: Since the set of combinatorial configurations to enumerate is finite, we can always define an order on them. However, the order should be determined so that the successive generation algorithm could be built.

Generate binary strings of length *n*

Problem: Enumerate all binary strings of length n:

$b_1\ b_2 \ldots b_n$, where $b_i \in \{0, 1\}$.

Solution:

- Dictionary order:

Consider each binary string $b = b_1\ b_2 \ldots b_n$ as the binary representation of an integer number $p(b)$

We say binary string $b = b_1\ b_2 \ldots b_n$ is ***previous*** binary string $b' = b'_1\ b'_2 \ldots b'_n$ in dictionary order and denote as $b \prec b'$ if $p(b) < p(b')$.

3. Generating algorithm

3.1. Algorithm diagram

3.2. Generate basic combinatorial configurations

- **Generate binary strings of length *n***
- Generate m-element subsets of the set of n elements
- Generate permutations of n elements

Generate binary strings of length *n*

Successive_Generation algorithm:

- The first string is 0 0 ... 0,
- The last string is 1 1 ... 1.
- Assume $b_1\ b_2 \ldots b_n$ *be the current string:*
  - If this string consists of all 1 → finish,
  - Otherwise, the next string is obtained by adding 1 (module 2, memorize) to the current string.
- Thus, we have following rule to generate the next string:
  - Find the first i (in the order $i = n, n\text{-}1, \ldots, 1$) such that $b_i = 0$.
  - Reassign $b_i = 1$ and $b_j = 0$ for all $j > i$. The newly obtained sequence will be the string to find.

3. Generating algorithm

3.1. Algorithm diagram

3.2. Generate basic combinatorial configurations

- Generate binary strings of length *n*
- **Generate *m*-element subsets of the set of *n* elements**
- Generate permutations of *n* elements

Generate *m*-element subsets of set with *n* elements

We say subset $a = (a_1, a_2,..., a_m)$ is previous subset $a' = (a'_1, a'_2, ... , a'_m)$ in dictionary order and denote as $a \prec a'$, if one could find the index k ($1 \le k \le m$) such that:

$$a_1 = a'_1 , a_2 = a'_2, . . . , a_{k-1} = a'_{k-1},$$
$$a_k < a'_k .$$

Generate *m*-element subsets of set with *n* elements

Problem: Let $X = \{1, 2, ... , n\}$. Enumerate all *m*-element subsets of *X*.

Solution:

- Lexicographic order:

Each *m*-element subset of *X* could be represented by tuples of *m* elements

$$a = (a_1, a_2, ... , a_m)$$

satisfying

$$1 \le a_1 < a_2 < ... < a_m \le n.$$

Generate *m*-element subsets of set with *n* elements

Successive_Generation algorithm:

- The first subset is $(1, \quad 2, \quad ... , m)$
- The last subset is $(n\text{-}m+1, n\text{-}m+2, ..., n)$.
- Assume $a=(a_1, a_2, ... , a_m)$ is the current subset but not the final yet, then its next subset in the dictionary order could be built by using the following rules:
  - Scan from the right to the left of sequence $a_1, a_2,..., a_m$: find the first element $a_i \ne n\text{-}m+i$
  - Replace a_i by $a_i + 1$
  - Replace a_j by $a_i + j - i$, where $j = i+1, i+2,..., m$

3. Generating algorithm

3.1. Algorithm diagram

3.2. Generate basic combinatorial configurations

- Generate binary strings of length *n*
- Generate *m*-element subsets of the set of *n* elements
- **Generate permutations of *n* elements**

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Generate permutations of *n* elements

We say permutation $a = (a_1, a_2,..., a_n)$ is previous permutation $a' = (a'_1, a'_2, ... , a'_n)$ in dictionary order and denote as $a \prec a'$, if we could find the index k $(1 \le k \le n)$ such that:

$$a_1 = a'_1 , a_2 = a'_2, ... , a_{k-1} = a'_{k-1},$$

$$a_k < a'_k .$$

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Generate permutations of *n* elements

Problem: Give $X = \{1, 2, ... , n\}$, enumerate all permutations of n elements of X.

Solution:

- Dictionary order:
  - Each permutation of n elements of X could be represented by an ordered set of n elements:

$$a = (a_1, a_2, ... , a_n)$$

satisfy

$$a_i \in X,\ \ i = 1, 2,..., n,\ a_p \ne a_q,\ p \ne q.$$

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Generate permutations of *n* elements

Successive_Generation algorithm:

- The first permutation: $(1, 2, ... , n)$
- The last permutation: $(n, n\text{-}1, ..., 1)$.
- Assume $a = (a_1, a_2, ... , a_n)$ is not the last permutation, then its next permutation could be built using the following rules:
  - Find from the right to left: first index j satisfying $a_j < a_{j+1}$ (in other word: j is the max index satisfying $a_j < a_{j+1}$);
  - Find a_k be the smallest number among those on the right of a_j in the sequence and *greater than* a_j ;
  - Swap a_j and a_k ;
  - Invert the sequence from a_{j+1} to a_n .

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

CONTENTS

1. Introduction to problem
2. Algorithm and Complexity
3. Generating algorithm
4. **Backtracking algorithm**

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Backtracking diagram

- ***Enumeration problem* (*Q*):** *Given $A_1, A_2,..., A_n$ be finite sets. Denote*

  $A = A_1 \times A_2 \times ... \times A_n = \{ (a_1, a_2, ..., a_n):\ a_i \in A_i,\ i=1, 2, ..., n\}$.

  Assume P is a property on the set A. The problem is to enumerate all elements of the set A that satisfies the property P:

  $D = \{ a = (a_1, a_2, ..., a_n) \in A:\ a \text{ satisfy property } P \}$.
- Elements of the set D are called **feasible solution** (*lời giải chấp nhận được).*

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Backtracking

4.1. Algorithm diagram

4.2. Generate basic combinatorial configurations

- Generate binary strings of length *n*
- Generate *m*-element subsets of the set of *n* elements
- Generate permutations of *n* elements

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Backtracking diagram

All basic combinatorial enumeration problem could be rephrased in the form of Enumeration problem (Q).

Example:

- The problem of enumerating all binary string of length *n* leads to the enumeration of elements of the set:

  $B^n = \{(a_1, ..., a_n): a_i \in \{0, 1\}, i=1, 2, ..., n\}$.
- The problem of enumerating all *m*-element subsets of set $N = \{1, 2, ..., n\}$ requires to enumerate elements of the set:

  $S(m,n) = \{(a_1,..., a_m)\in N^m: 1 \le a_1 < ... < a_m \le n \}$.
- The problem of enumerating all permutations of natural numbers 1, 2, ..., *n* requires to enumerate elements of the set

  $\Pi_n = \{(a_1,..., a_n) \in N^n:\ a_i \neq a_j ;\ i \neq j \}$.

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Partial solution

The solution to the problem is an ordered tuple of n elements $(a_1, a_2, ..., a_n)$, where $a_i \in A_i$, $i = 1, 2, ..., n$.

Definition. The k-level partial solution ($0 \leq k \leq n$) is an ordered tuple of k elements

$$(a_1, a_2, ..., a_k),$$

where $a_i \in A_i$, $i = 1, 2, ..., k$.

- When $k = 0$, 0-level partial solution is denoted as ( ), and called as the empty solution.
- When $k = n$, we have a complete solution to a problem.

Backtracking diagram

- General step: Assume we have k-1 level partial solution:

$$(a_1, a_2, ..., a_{k-1}),$$

Now we need to build k-level partial solution:

$$(a_1, a_2, ..., a_{k-1}, a_k)$$

- Based on the property P, we determine which elements of set A_k could be selected as the kth component of solution.
- Such elements are called as candidates for the kth position of solution when k-1 first components have been chosen as $(a_1, a_2, ..., a_{k-1})$. Denote these candidates by S_k.
- Consider 2 cases:
  - $S_k \neq \varnothing$
  - $S_k = \varnothing$

Backtracking diagram

Backtracking algorithm is built based on the construction each component of solution one by one.

- Algorithm starts with empty solution ( ).
- Based on the property P, we determine which elements of set A_1 could be selected as the first component of solution. Such elements are called as **candidates** for the first component of solution. Denote candidates for the first component of solution as S_1. Take an element $a_1 \in S_1$, insert it into empty solution, we obtain 1-level partial solution: (a_1).

Backtracking diagram

- ***$S_k \neq \varnothing$:*** Take $a_k \in S_k$ to insert it into current (k-1)-level partial solution $(a_1, a_2, ..., a_{k-1})$, we obtain k-level partial solution $(a_1, a_2, ..., a_{k-1}, a_k)$. Then
  - If $k = n$, then we obtain a complete solution to the problem,
  - If $k < n$, we continue to build the (k+1)th component of solution.
- ***$S_k = \varnothing$:*** It means the partial solution $(a_1, a_2, ..., a_{k-1})$ can not continue to develop into the complete solution. In this case, we **backtrack** to find new candidate for (k-1)th position of solution (note: this new candidate must be an element of S_{k-1})
  - If one could find such candidate, we insert it into (k-1)th position, then continue to build the kth component.
  - If such candidate could not be found, we backtrack one more step to find new candidate for (k-2)th position,… If backtrack till the empty solution, we still can not find new candidate for 1st position, then the algorithm is finished.

Backtracking algorithm (recursive)

```
void Try(int k)
{
    <Build $S_k$ as the set to consist of candidates
   for the $k^{th}$ component of solution>;
   for y ∈ $S_k$  //Each candidate y of $S_k$
   {
   $a_k$ = y;
       if  (k == n) then <Record ($a_1$, $a_2$, ..., $a_k$)
   as a complete solution >;
   else Try(k+1);
       Return the variables to their old states;
  }
}
```

The call to execute backtracking algorithm: Try(1);

- If only one solution need to be found, then it is necessary to find a way to terminate the nested recursive calls generated by the call to Try(1) once the first solution has just been recorded.
- If at the end of the algorithm, we obtain no solution, it means that the problem does not have any solution.

Backtracking algorithm (not recursive)

```
void Backtraking ( )
{
 k=1;
 <Build $S_k$>;
 while (k > 0)  {
    while ($S_k \neq \varnothing$ ) {
        $a_k \leftarrow S_k$; // Take $a_k$ from $S_k$
        if  <(k == n) >  then <Record ($a_1$,$a_2$,...,$a_k$) as a
                              complete solution >;
        else {
               k = k+1;
               <Build $S_k$>;
        }
    }
    k = k - 1;   // Backtracking
 }
}
```

The call to execute backtracking algorithm: Bactraking ( );

Note

- If the length of complete solution is not known in advanced, and solutions are not needed to have the same length:

```
void Try(int k)
{
  <Build $S_k$ as the set to consist of candidates for the $k^{th}$ component of solution>;
  for y ∈ $S_k$  //Each candidate y of $S_k$
  {
     $a_k$ = y;
     if  (k == n) then <Record ($a_1$, $a_2$, ..., $a_k$) as a complete solution >;
     else Try(k+1);
     Return the variables to their old states;
  }
}
```

- Then we need to modify statement

if (*k* == *n*) then <Record (a_1, a_2, ..., a_k) as a complete solution >;
else Try(k+1);

to

if <(a_1, a_2, ..., a_k) is a complete solution> then <Record (a_1, a_2, ..., a_k) as a complete solution >;
else Try(k+1);

→ Need to build a function to check whether (a_1, a_2, ..., a_k) is the complete solution.

Two key issues

- In order to implement a backtracking algorithm to solve a specific combinatorial problems, we need to solve the following two basic problems:
  - Find algorithms to build candidate sets S_k
  - Find a way to describe these sets so that you can implement the operation to enumerate all their elements (implement the loop `for y ∈ Sk`).
    - The efficiency of the enumeration algorithm depends on whether we can accurately identify these candidate sets.

4. Backtracking

4.1. Algorithm diagram

4.2. Generate basic combinatorial configurations

- **Generate binary strings of length *n***
- Generate *m*-element subsets of the set of *n* elements
- Generate permutations of *n* elements

Example 1: Enumerate all binary string of length *n*

- Problem to enumerate all binary string of length n leads to the enumeration of all elements of the set:

  $A^n = \{(a_1, ..., a_n): a_i \in \{0, 1\}, i=1, 2, ..., n\}$.

- We consider how to solve two issue keys to implement backtracking algorithm:
  - Build candidate set S_k: We have $S_1 = \{0, 1\}$. Assume we have binary string of length k-1 $(a_1, ..., a_{k-1})$, then $S_k = \{0,1\}$. Thus, the candidate sets for each position of the solution are determined.
  - Implement the loop to enumerate all elements of S_k: we can use the loop `for`

```
for (y=0; y<=1; y++) in C/C++
```

Example 2. Generate *m*-element subsets of the set of *n* elements

Problem: Enumerate all m-element subsets of the set n elements $N = \{1, 2, ..., n\}$.

Example: Enumerate all 3-element subsets of the set 5 elements N = {1, 2, 3, 4, 5}

Solution: (1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5), (1, 4, 5), (2, 3, 4), (2, 3, 5), (2, 4, 5), (3, 4, 5)

➔ Equivalent problem: Enumerate all elements of set:

$$S(m,n)=\{(a_1,..., a_m)\in N^m: 1 \le a_1<...<a_m \le n\}$$

4. Backtracking

4.1. Algorithm diagram

4.2. Generate basic combinatorial configurations

- Generate binary strings of length n
- Generate m-element subsets of the set of n elements
- Generate permutations of n elements

Example 2. Generate *m*-element subsets of the set of *n* elements

We consider how to solve two issue keys to implement backtracking:

- Build candidate set S_k:
  - With the condition: : $1 \le a_1 < a_2 < ... < a_m \le n$

  we have $S_1 = \{1, 2, ..., n-(m-1)\}$.
  - Assume the current subset is $(a_1, ..., a_{k-1})$, with the condition $a_{k-1} < a_k < . . . < a_m \le n$, we have $S_k = \{a_{k-1}+1, a_{k-1}+2, ..., n-(m-k)\}$.
- Implement the loop to enumerate all elements of S_k: we can use the loop `for`

```
for (j=a[k-1]+1;j<=n-m+k;j++)..
```

4. Backtracking

4.1. Algorithm diagram

4.2. Generate basic combinatorial configurations

- Generate binary strings of length *n*
- Generate *m*-element subsets of the set of *n* elements
- **Generate permutations of *n* elements**

Example 3. Enumerate permutations

- Build candidate set S_k:
  - Actually $S_1 = N$. Assume we have current partial permutation $(a_1, a_2, ..., a_{k-1})$, with the condition $a_i \neq a_j$, for all $i \neq j$, we have

    $S_k = N \setminus \{ a_1, a_2, ..., a_{k-1} \}$.

Example 3. Enumerate permutations

Permutation set of natural numbers 1, 2, ..., *n* is the set:

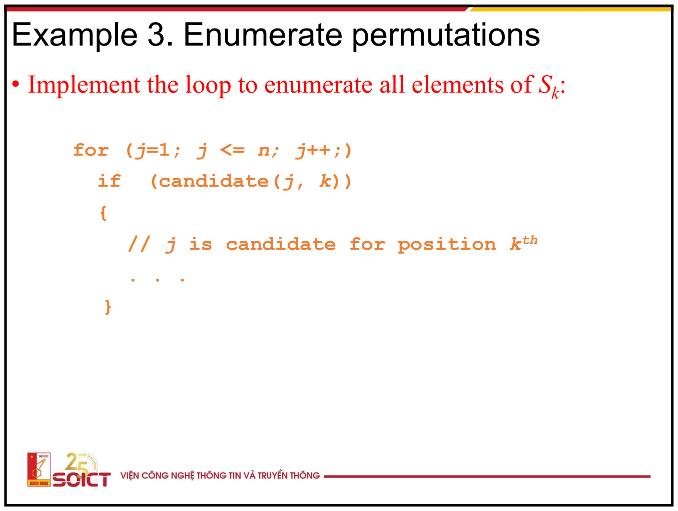
$$\Pi_n = \{(x_1,..., x_n) \in N^n: \ x_i \neq x_j \, , \, i \neq j \}.$$

Problem: Enumerate all elements of Π_n

Describe S_k

Build function to detect candidates:

```
bool candidate(int j, int k)
{
  //function returns true if and only if j ∈ Sk
   int i;
   for (i=1;i++;i<=k-1)
       if (j == a[i]) return false;
   return true;
}
```



Example 3. Enumerate permutations

- Implement the loop to enumerate all elements of S_k:

```
for (j=1; j <= n; j++;)
  if  (candidate(j, k))
  {
    // j is candidate for position k^th
    . . .
   }
```