Discrete Mathematics

Generating combinatorial configurations

Pham Quang Dung

Hanoi, 2012

Outline

Introduction

- List all configurations satisfying some given constraints
  - permutations
  - subsets of a given set
  - etc.
- $A_1, \ldots, A_n$ are finite sets and $X = \{(a_1, \ldots, a_n) \mid a_i \in A_i, \forall 1 \leq i \leq n\}$
- $\mathcal{P}$ is a property on X
- Generate all configurations $(a_1, \ldots, a_n)$ having $\mathcal{P}$

Introduction

- In many cases, listing is a final way for solving some combinatorial problems
- Two popular methods
  - Generating method
  - BackTracking algorithm

Outline

Generating method

- Define an order on a set of configurations
- Generating algorithm: generate a successive configuration from a current (not final) configuration

Algorithm 1: Generate()

```
C ← Generate an initial configuration;
STOP ← FALSE;
while not STOP do
    C ← GenerateNext(C);
    if C = ∅ then
        STOP ← true;
```

Generating method: Lexical order

- $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$
- $A < B$ if there exists $1 \leq k \leq n$ such that
  - $a_i = b_i, \forall i = 1, \ldots, k-1$
  - $a_k < b_k$

Generating method: binary sequence

00000	01000	10000	11000
00001	01001	10001	11001
00010	01010	10010	11010
00011	01011	10011	11011
00100	01100	10100	11100
00101	01101	10101	11101
00110	01110	10110	11110
00111	01111	10111	11111

- Current configuration: 010110111
- Next configuration: 010111000

Generating method: binary sequence

Generate next configuration of $(b_1, \ldots, b_n)$

- From right to left, find the first position k s.t. $b_k = 0$
- set $b_k = 1$
- set $b_i = 0, \forall i = k+1, \ldots, n$

Generating method: binary sequence

```
int stop = 0;
while(!stop){
int k = n;
while(k >= 1 && b[k] == 1)
  k--;
  if(k >= 1){
    b[k] = 1;
    for(int i = k+1; i <= n; i++)
      b[i] = 0;
    printConfiguration();
  }else{
    stop = 1;
  }
}
```

Generating method: combination

1 2 3 4 5	1 2 4 5 7	1 4 5 6 7
1 2 3 4 6	1 2 4 6 7	2 3 4 5 6
1 2 3 4 7	1 2 5 6 7	2 3 4 5 7
1 2 3 5 6	1 3 4 5 6	2 3 4 6 7
1 2 3 5 7	1 3 4 5 7	2 3 5 6 7
1 2 3 6 7	1 3 4 6 7	2 4 5 6 7
1 2 4 5 6	1 3 5 6 7	3 4 5 6 7

- $n = 9, k = 6$
- Current configuration:1 2 3 7 8 9
- Next configuration: 1 2 4 5 6 7

Generating method: combination

Generate next configuration of $(c_1, \ldots, c_k)$

- From right to left, find the first position i s.t. $c_k < n - k + i$
- Increase c_i by 1
- Set $c_{j+1} = c_j + 1, \forall j = i, \ldots, k-1$

Generating method: combination

```
for(int i = 1; i <= k; i++)
  c[i] = i;
printConfiguration();

int stop = 0;
while(!stop){
  int i = k;
  while(i >= 1 && c[i] >= n-k+i)
    i--;
  if(i <= 0)
    stop = 1;
  else{
    c[i] = c[i] + 1;
    for(int j = i; j <= k-1; j++)
      c[j+1] = c[j] + 1;
    printConfiguration();
  }
}
```

Generating method: permutation

1 2 3 4	2 1 3 4	3 1 2 4	4 1 2 3
1 2 4 3	2 1 4 3	3 1 4 2	4 1 3 2
1 3 2 4	2 3 1 4	3 2 1 4	4 2 1 3
1 3 4 2	2 3 4 1	3 2 4 1	4 2 3 1
1 4 2 3	2 4 1 3	3 4 1 2	4 3 1 2
1 4 3 2	2 4 3 1	3 4 2 1	4 3 2 1

- Current configuration: 8 7 6 3 5 4 2 1
- Next configuration: 8 7 6 4 1 2 3 5

Generating method: permutation

Generate next configuration of $(p_1, \ldots, p_n)$

- From right to left, find the first position k s.t. $p_k < p_{k+1}$
- From position $k+1$ to the right, find the first position i s.t. $p_k < p_i$
- Swap p_k and p_i
- Reverse the $p_{k+1}, \ldots, p_n$

Generating method - permutation

```
int stop = 0;
while(!stop){
  int k = n-1;
  while(k >= 1 && p[k] > p[k+1])
    k--;
  if(k <= 0)
    stop = 1;
  else{
    int i = k+1;
    while(p[i] > p[k]) i++;
    i--;
    swap(p[k],p[i]);
    reverse(p,k+1,n);
    printConfiguration();
  }
}
```

Outline

BackTracking algorithm

Construct elements of the configuration step-by-step

- Initialization: Constructed configuration is null: ()
- Step 1:
  - Compute (base on $\mathcal{P}$) a set S_1 of candidates for the first position of the configuration under construction
  - Select an item of S_1 and put it in the first position

BackTracking algorithm

At Step k: Suppose we have partial configuration $a_1, \ldots, a_{k-1}$

- Compute (base on $\mathcal{P}$) a set S_k of candidates for the k^{th} position of the configuration under construction
  - If $S_k \neq \emptyset$, then select an item of S_k and put it in the k^{th} position and obtain $(a_1, \ldots, a_{k-1}, a_k)$
    - If $k = n$, then process the complete configuration $a_1, \ldots, a_n)$
    - Otherwise, construct the $k + 1^{th}$ element of the partial configuration in the same schema
  - If $S_k = \emptyset$, then backtrack for trying another item a'_{k-1} for the $k - 1^{th}$ position
    - If a'_{k-1} exists, then put it in the $k - 1^{th}$ position
    - Otherwise, backtrack for trying another item for the $k - 2^{th}$ position, ...

BackTracking algorithm

Algorithm 2: BackTracking(k)

Construct a candidate set S_k;
foreach $y \in S_k$ **do**
    $a_k \leftarrow y$;
    if $(a_1, \ldots, a_k)$ *is a complete configuration* **then**
        ProcessConfiguration$(a_1, \ldots, a_k)$;
    else
        BackTracking$(k+1)$;

Algorithm 3: Main()

BackTracking(1);

BackTracking algorithm - binary sequence

```
void BackTracking(int k){
  for(int i = 0; i <= 1; i++){
    b[k] = i;
    if(k == n)
      printConfiguration();
    else
      BackTracking(k+1);
  }
}
```

BackTracking algorithm - combination

```
void BackTracking(int i){
  for(int j = c[i-1]+2; j <= n-k+i; j++){
    c[i] = j;
    if(i == k){
      printConfiguration();
    }else
      BackTracking(i+1);
  }
}
```

BackTracking algorithm - permutation

```
void BackTracking(int k){
  for(int i = 1; i <= n; i++){
    if(!b[i]){
      p[k] = i;
      b[i] = 1;
      if(k == n){
        printConfiguration();
      }else
        BackTracking(k+1);
      b[i] = 0;
    }
  }
}
```

Solve the linear equations in a set of positive integers

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = M$$

where $(a_i)_{1\leq i\leq n}$ and M are positive integers

- Partial solution $(x_1, x_2, \ldots, x_{k-1})$
- $m = \sum_{i=1}^{k-1} a_ix_i$
- $A = \sum_{i=k+1}^{n} a_i$
- $\overline{M} = M - m - A$
- Candidates of x_k is $\{v \in \mathbb{Z} \mid 1 \leq v \leq \frac{\overline{M}}{a_k}\}$

BackTracking algorithm - Linear integer equation

```
void TRY(int k){ // try a value for variable x[k]
  for(int val = 1; val <= (M-m-A)/a[k]; val++){
    x[k] = val;
    m = m + a[k]*x[k];
    A = A - a[k];
    if(k == n){
      if(m==M)
      printSolution();
    }else
      TRY(k+1);
    m = m - a[k]*x[k];
    A = A + a[k];
  }
}
int main(int argc, char** argv){
  m = 0;
  A = 0;
  for(int i = 2; i <= n; i++)
    A = A + a[i];
  TRY(1);
}
```

BackTracking algorithm - n-queens problem

- Problem: Place n queens on a chess board such that no two queens attack each other
- Solution model: $(x_1, x_2, \ldots, x_n)$ where x_i represents the row on which the queen in column i is located
- Constraints:
  - $x_i \neq x_j, \forall 1 \leq i < j \leq n$
  - $|x_i - x_j| \neq |i - j|, \forall 1 \leq i < j \leq n$

BackTracking algorithm - n-queens problem

```
int x[100];
int n;
int candidate(int k, int v){
  for(int i = 1; i <= k-1; i++)
    if(x[i] == v || abs(x[i]-v)==abs(i-k)) return 0;
  return 1;
}
void BTrack(int k){
  for(int v = 1; v <= n; v++)
    if(candidate(k,v) == 1){
      x[k] = v;
      if(k == n)
        printSolution();
      else
        BTrack(k+1);
    }
}
int main(int agrc, char** args){
  n = 8;
  BTrack(1);
}
```

BackTracking algorithm - n-queens problem - refinement

- Use arrays for marking forbiden cells
  - $r[1..n]$: $r[i]$ = false if the cells on row i are forbiden
  - $d_1[1-n..n-1]$: $d_1[q]$ = false if cells (r, c) s.t. $c - r = q$ are forbiden
    - in C++, indices of elements of an array cannot be negative (i.e., indices are 0, 1, ...). Hence making a deplacement: $d_1[q+n-1]$ instead of $d_1[q]$
  - $d_2[2..2n-2]$: $d_2[q]$ =false if cells (r, c) s.t. $r + c = q$ are forbiden

BackTracking algorithm - n-queens problem

```
void BTrack(int i){// try values for x[i]
  for(int val = 1; val <= n; val++){
    if(r[val] == true && d1[i-val+n-1] == true && d2[i+val]
       == true){
      x[i] = val;
      r[val] = false;// marking forbiden cells
      d1[i-val+n-1] = false;// marking forbiden cells
      d2[i+val] = false;// marking forbiden cells
      if(i == n){
        printSolution();
      }else
        BTrack(i+1);
      r[val] = true;// recovering marking
      d1[i-val+n-1] = true;// recovering marking
      d2[i+val] = true;// recovering marking
    }
  }
}
```

BackTracking algorithm - n-queens problem

```
int main(int argc, char** argv){
  n = atoi(argv[1]);

  for(int i = 1; i <= n; i++)
    r[i] = true;
  for(int i = 0; i <= 2*n; i++){
    d1[i] = true;
    d2[i] = true;
  }

  BTrack(1);
}
```

- Sudoku problem
- Subset Sum problem
- List all the ways to decompose a positive integer N into a sum of positive integers