Recursion, Backtracking and Branch-and-Bound

Pham Quang Dung and Do Phan Thuan

Computer Science Department, SoICT,
Hanoi University of Science and Technology.

August 15, 2016

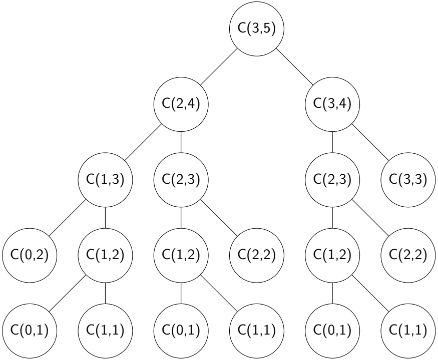
Recursive procedures

- A procedure calls itself
- Basic cases: the results are computed trivially

```
int fact(int n){
  if(n <= 1) return 1;
  return n*fact(n-1);
}
int C(int k, int n){
  if(k == n || k == 0) return 1;
  return C(k-1,n-1) + C(k,n-1);
}
```

Recursion and Memoization

- Procedures with the same parameters may be called several times
- A procedure with a given set of parameters is triggered for the first time is executed, and the results will be stored into memory
- Later on, if that procedure with the same set of parameters is triggered, the procedure will not execute. Rather, the results of that procedure available in the memory will be returned directly



Recursion and Memoization

```java
public class Ckn {
  private int[][] M;
  public int C(int k, int n){
    if(k == 0 || k == n) M[k][n] = 1;
    else if(M[k][n] < 0){
      M[k][n] = C(k-1,n-1) + C(k,n-1);
    }
    return M[k][n];
  }
  public void test(){
    M = new int[100][100];
    for(int i = 0; i < 100; i++)
      for(int j = 0; j < 100; j++)
        M[i][j] = -1;

    System.out.println(C(15,30));
  }
}
```

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 (not consider)
  - BackTracking algorithm

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 1: TRY(k)

```
Construct a candidate set S_k;
foreach y ∈ S_k do
    a_k ← y;
    if (a_1, ..., a_k) is a complete configuration then
        ProcessConfiguration(a_1, ..., a_k);
    else
        TRY(k + 1);
```

Algorithm 2: Main()

```
TRY(1);
```

BackTracking algorithm - binary sequence

- A configuration is represented by $b_1, b_2, \ldots, b_n$
- Candidates for b_i is $\{0, 1\}$

BackTracking algorithm - binary sequence

```
public class ListingBinary {
  private int[] a;
  private int n;
  private void TRY(int i){
    for(int v = 0; v <= 1; v++){
      a[i] = v;
      if(i == n-1){
        for(int j = 0; j < n; j++) System.out.print(a[j]);
        System.out.println();
      }else{
        TRY(i+1);
      }
    }
  }
  public void list(int n){
    this.n = n;
    a = new int[n];
    TRY(0);
  }
  public static void main(String[] args) {
    ListingBinary LB = new ListingBinary();
    LB.list(4);
  }

}
```

BackTracking algorithm - combination

- A configuration is represented by $(c_1, c_2, \ldots, c_k)$
  - dummy $c_0 = 1$
  - Candidates for c_i being aware of $\langle c_1, c_2, \ldots, c_{i-1} \rangle$:
    $c_{i-1} + 1 \leq c_i \leq n - k + i, \forall i = 1, 2, \ldots, k$

BackTracking algorithm - combination

Algorithm 3: TRY(i)

```
foreach v = c_{i-1} + 1, ..., n - k + i do
    c_i ← v;
    if i == k then
        printConfiguration();
    else
        TRY(i + 1);
```

Algorithm 4: MainCombinationGeneration(n, k)

```
c_0 ← 0;
TRY(1);
```

BackTracking algorithm - permutation

- A configuration: $p_1, p_2, \ldots, p_k$
- Candidates for p_i being aware of $\langle p_1, p_2, \ldots, p_{i-1} \rangle$:
  $\{1, 2, \ldots, n\} \setminus \{p_1, p_2, \ldots, p_{i-1}\}$
- Use an array of booleans for making values used $b_1, b_2, \ldots, b_n$
  - $b_v = 1$, if value v is already used (appear in $p_1, p_2, \ldots, p_{i-1}$)
  - $b_v = 0$, otherwise

BackTracking algorithm - permutation

Algorithm 5: TRY(i)

```
foreach v = 1, ..., n do
    if visited[v] = FALSE then
        p_i ← v;
        visited[v] ← TRUE;
        if i == n then
            printConfiguration();
        else
            TRY(i + 1);
        visited[v] ← FALSE;
```

Algorithm 6: MainPermutationGeneration(n, k)

```
foreach v = 1, ..., n do
    visited[v] ← FALSE;
TRY(1);
```

BackTracking algorithm - Linear integer equation

Solve the linear equations in a set of positive integers

$$x_1 + x_2 + \cdots + x_n = M$$

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

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

BackTracking algorithm - Linear Integer Equation

Algorithm 7: TRY(i)

```
if i = n then
    M̄ ← M − f;
    M̲ ← M − f;
else
    M̄ ← M − f − (n − i);
    M̲ ← 1;
foreach v = M̲, ..., M̄ do
    x_i ← v;
    f ← f + v;
    if i == n then
        printConfiguration();
    else
        TRY(i + 1);
    f ← f − v;
```

Algorithm 8: MainLinearEquation(n, M)

```
f ← 0;
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

Algorithm 9: Candidate(v, i)

foreach $j = 1, \ldots, i-1$ **do**
  if $x_j = v \vee |x_j - v| = |j - i|$ **then**
    return FALSE;
  return TRUE;

Algorithm 10: TRY(i)

foreach $v = 1, \ldots, n$ **do**
  if *Candidate*(v, i) **then**
    $x_i \leftarrow v$;
    if $i == n$ **then**
      Solution();
    else
      TRY($i + 1$);

Algorithm 11: MainQueen(n)

TRY(1);

- Use arrays for marking forbidden cells
  - $r[1..n]$: $r[i]$ = false if the cells on row i are forbidden
  - $d_1[1-n..n-1]$: $d_1[q]$ = false if cells (r,c) s.t. $c-r=q$ are forbiden
    - in Java, 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

Algorithm 12: TRY(i)

```
foreach v = 1, ..., n do
    if r[v] ∧ d1[i − v] ∧ d2[i + v] then
        x_i ← v;
        r[v] ← FALSE;
        d1[i − v] ← FALSE;
        d2[i + v] ← FALSE;
        if i = n then
            Solution();
        else
            TRY(i + 1);
        r[v] ← TRUE;
        d1[i − v] ← TRUE;
        d2[i + v] ← TRUE;
```

Algorithm 13: MainQueenRefine(n)

foreach $v = 1, \dots, n$ **do**
    $r[v] \leftarrow$ TRUE;

foreach $v = 1-n, \dots, n-1$ **do**
    $d_1[v] \leftarrow$ TRUE;

foreach $v = 2, \dots, 2n$ **do**
    $d_2[v] \leftarrow$ TRUE;

TRY(1);

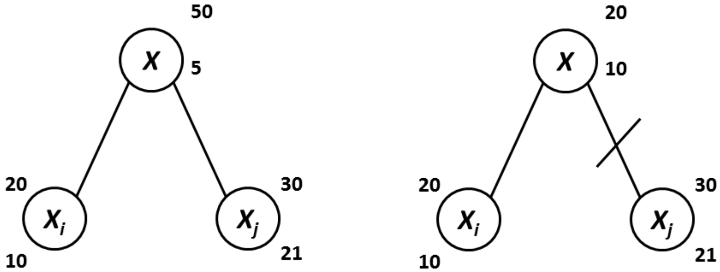
Combinatorial Optimization Problems

- $z = \min \{f(x) : x \in X\}$
- Applications
  - Vehicle Routing
  - Scheduling
  - Timetabling
  - Bin Packing
  - Resource allocations
  - ...

Generic schema of Branch and Bound

- Branch-and-Bound splits the given problem into smaller and smaller subproblems until they become easy to solve (Branching)
  - X is splited into subsets $X_1 \ldots, X_k (k \geq 2)$ such that $\bigcup_{i=1,\ldots,k} X_i = X$
  - Recursive application of splitting defines a tree structure: *search tree* (each node is a subset of X)
- Normally, the size of the search tree is too large (exponential)
- Bounding
  - For each set $X_i (\forall i = 1, \ldots, k)$
    - ★ $z^i = \min \{f(x) : x \in X_i\}$
    - ★ compute $\underline{z}^i$ and $\overline{z}^i$ respectively the lower bound and upper bound of z^i: $\underline{z}^i \leq z^i \leq \overline{z}^i$
  - If there exist $i \neq j$ s.t. $\overline{z}^i \leq \underline{z}^j$, then the set X_j can be removed from the search space since $z^j \geq z^i$ (no need to explore X_j)
  - Suppose that z^* is incumbent (best solution found so far). If $\underline{z}^i \geq z^*$, then X_i can be removed (no need to explore X_i since $z^* \leq \underline{z}^i \leq z^i$)

Generic schema of Branch and Bound - example



Generic schema of Branch and Bound algorithms (minimization problems)

Algorithm 14: TRY(k)

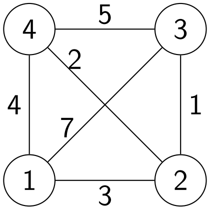
Construct a candidate set S_k;
foreach $y \in S_k$ **do**
    $a_k \leftarrow y$;
    if $(a_1, \ldots, a_k)$ *is a complete configuration* **then**
        if $f(a_1, \ldots, a_k) < z^*$ **then**
            $z^* \leftarrow f(a_1, \ldots, a_k)$;
    else
        if $\underline{z}(a_1, \ldots, a_k) < z^*$ **then**
            TRY$(k+1)$;

Algorithm 15: Main()

$z^* \leftarrow +\infty$;
TRY(1);

Traveling Salesman Problem

- Given a list of n cities with pairwise distances
- Find the shortest route that visits each city exactly once and returns to the origin city
- $x = (x_1, \ldots, x_n)$, route is $x_1 \to x_2 \to \cdots \to x_n \to x_1$
- $f(x) = c(x_1, x_2) + c(x_2, x_3) + \cdots + c(x_n, x_1)$



$$c = \begin{pmatrix} 0 & 3 & 7 & 4 \\ 3 & 0 & 1 & 2 \\ 7 & 1 & 0 & 5 \\ 4 & 2 & 5 & 0 \end{pmatrix}$$

Traveling Salesman Problem - Simple Branch-and-Bound

- A subproblem
  - Correspond to a prefix of the solution: $x_1, x_2, ..., x_k$
  - Lower bound: $\underline{z}(x_1, ..., x_k) = c(x_1, x_2) + ... + c(x_{k-1}, x_k) + (n - k + 1) * cmin$ where $cmin$ is the minimum element of the cost matrix (exclusive elements of the diagonal)
  - Recursive procedure **extend**($\langle x_1, ..., x_{k-1} \rangle$) will extend current partial solution

Algorithm 16: TRY(k)

Input: k: the index of k^{th} city to be visited

$n, c, x, f^*, f, visited$ are global variables

Output: Extend current partial solution $x_1, \ldots, x_{k-1}$ by assigning a value to x_k

foreach $v = 1, \ldots, n$ **do**

- **if** $visited[v] = FALSE$ **then**
  - $x_k \leftarrow v$;
  - $visited[v] \leftarrow TRUE$;
  - $f \leftarrow f + c(x_{k-1}, x_k)$;
  - **if** $k = n$ **then**
    - **if** $f + c(x_n, x_1) < f^*$ **then**
      - $f^* \leftarrow f + c(x_n, x_1)$;
  - **else**
    - $\underline{z} \leftarrow f + (n - k + 1) * cmin$;
    - **if** $\underline{z} < f^*$ **then**
      - TRY($k + 1$);
  - $f \leftarrow f - c(x_{k-1}, x_k)$;
  - $visited[v] \leftarrow FALSE$;

Traveling Salesman Problem - Simple Branch-and-Bound

Algorithm 17: MainSimpleBBTSP(n, c)

Input: n, c: number of cities n and distance matrix c
x, f^*, f, *visited* are initialized as global variables
Output: The length of the shortest tour
foreach $v = 1, \ldots, n$ **do**
  $visited[v] \leftarrow FALSE$;
$f^* \leftarrow \infty$;
$f \leftarrow 0$;
$x_1 \leftarrow 1$;
$visited[x_1] \leftarrow TRUE$;
TRY(2);
return f^*;

Traveling Salesman Problem - Second Branch-and-Bound

- Lower bound
  - A Tour is associated with a set S of n cells of the cost matrix in which each row, column of the cost matrix contain exactly one element of S.
  - Hence the optimal Tour does not change if we subtract each cell of a given row (or column) with a same value.
  - Algorithm **reduce** will compute the lower bound of the optimal tour

Algorithm 18: reduce(C)

```
1..k is the size of the cost matrix C;
S ← 0;
foreach i ∈ 1..k do
    minRow ← minimum value of row i of C;
    if minRow > 0 then
        foreach j ∈ 1..k do
            C[i][j] = C[i][j] − minRow;
        S ← S + minRow;
foreach j ∈ 1..k do
    minCol ← minimum value of column j of C;
    if minCol > 0 then
        foreach i ∈ 1..k do
            C[i][j] = C[i][j] − minCol;
        S ← S + minCol;
return S;
```

Traveling Salesman Problem - Branching

- Select an arc (u, v) for branching (computed by **bestEdge** below)
  - Tours contain (u, v)
    - Remove row u and column v
    - Set $C[v][u] = \infty$
    - If u is a terminating node of a path $\langle x_1, x_2, ..., u \rangle$ and v is a starting node of a path $\langle v, y_1, ..., y_k \rangle$, then $C[y_k][x_1] = \infty$ to prevent sub-tour
  - Tours do not contain (u, v)
    - Set $C[u][v] = \infty$

Traveling Salesman Problem - Branching

- When the reduced matrix has size 2×2

$$\begin{array}{c|cc} & w & x \\ \hline u & 0 & \infty \\ v & \infty & 0 \end{array} \qquad \begin{array}{c|cc} & w & x \\ \hline u & \infty & 0 \\ v & 0 & \infty \end{array}$$

a. admit (u, w) and (v, x) b. admit (u, x) and (v, w)

Traveling Salesman Problem - Branching

Algorithm 19: bestEdge(C)

```
1..k is the size of the cost matrix C;
best ← −∞;
foreach i ∈ 1..k do
    foreach j ∈ 1..k do
        if C[i][j] = 0 then
            minRow ← smallest element of row i which is different from C[i][j];
            minCol ← smallest element of column j which is different from C[i][j];
            total ← minRow + minCol;
            if total > best then
                best ← total;
                selRow ← i;
                selCol ← j;
return (selRow, selCol)
```

Traveling Salesman Problem

	1	2	3	4	5	6	r[i]
1	∞	3	93	13	33	9	3
2	4	∞	77	42	21	16	4
3	45	17	∞	36	16	28	16
4	39	90	80	∞	56	7	7
5	28	46	88	33	∞	25	25
6	3	88	18	46	92	∞	3
s[i]	0	0	15	8	0	0	

a. Original Cost matrix

	1	2	3	4	5	6
1	∞	0	75	2	30	6
2	0	∞	58	30	17	12
3	29	1	∞	12	0	12
4	32	83	58	∞	49	0
5	3	21	48	0	∞	0
6	0	85	0	35	89	∞

Lower bound = 81

b. Reduced matrix

Traveling Salesman Problem

Set of Tours is divided into 2 cases:

	1	2	4	5	6
1	∞	0	2	30	6
2	0	∞	30	17	12
3	29	1	12	0	∞
4	32	83	∞	49	0
5	3	21	0	∞	0

Tours contain (6,3), lower bound = 81

	1	2	3	4	5	6
1	∞	0	75	2	30	6
2	0	∞	58	30	17	12
3	29	1	∞	12	0	12
4	32	83	58	∞	49	0
5	3	21	48	0	∞	0
6	0	85	∞	35	89	∞

Tours do not contain (6,3), lower bound = 129

Traveling Salesman Problem

Set of Tours containing (6,3) is divided into 2 cases:

	1	2	4	5
1	∞	0	2	30
2	0	∞	30	17
3	29	1	∞	0
5	3	21	0	∞

Tours contain (6,3), (4,6), lower bound = 81

	1	2	4	5	6
1	∞	0	2	30	6
2	0	∞	30	17	12
3	29	1	12	0	∞
4	32	83	∞	49	0
5	3	21	0	∞	0

Tours contain (6,3), not (4,6), lower bound = 113

Traveling Salesman Problem

Set of Tours containing (6,3), (4,6) is divided into 2 cases:

	2	4	5
1	∞	2	28
3	0	∞	0
5	20	0	∞

Tours contain (6,3), (4,6), (2,1), lower bound = 84

	1	2	4	5
1	∞	0	2	30
2	∞	∞	30	17
3	29	1	∞	0
5	3	21	0	∞

Tours contain (6,3), (4,6), not (2,1), lower bound = 101

Traveling Salesman Problem

Set of Tours containing (6,3), (4,6), (2,1) is divided into 2 cases:

	2	5
3	∞	0
5	0	∞

Tours contain (6,3), (4,6), (2,1), (1,4), lower bound = 84

Add arcs (3,5) and (5,2), we obtain a solution cost = 104

	2	4	5
1	∞	∞	0
3	0	∞	0
5	20	0	∞

Tours contain (6,3), (4,6), (2,1), not (1,4), lower bound = 112

Traveling Salesman Problem

Set of Tours containing (6,3), (4,6), not (2,1) is divided into 2 cases:

	2	4	5
1	0	0	∞
2	∞	11	0
3	1	∞	0

Tours contain (6,3), (4,6), not (2,1), (5,1), lower bound = 103

	1	2	4	5
1	∞	0	2	30
2	∞	∞	13	0
3	0	1	∞	0
5	∞	21	0	∞

Tours contain (6,3), (4,6), not (2,1), not (5,1), lower bound = 127

Traveling Salesman Problem

Set of Tours containing (6,3), (4,6), (5,1), not (2,1)
is divided into 2 cases:

	2	5
2	∞	0
3	1	∞

Tours contain (6,3), (4,6), not (2,1), (5,1), (1,4), lower bound = 103

	2	4	5
1	0	∞	∞
2	∞	0	0
3	1	∞	0

Tours contain (6,3), (4,6), not (2,1), (5,1), not (1,4), lower bound = 114

- Finally, the best Tour has cost 104

Branch-and-Bound: MaxClique

- Description
  - Input: undirected graph $G = (V, E)$,
  - Subgraph: Let $G(S)$ be the graph (S, E_S) in which $E_S = \{(u, v) \mid u, v \in S \wedge (u, v) \in E\}$. $G(S)$ is called subgraph induced by S ($\forall S \subseteq V$)
  - Output: maximal complete subgraph (or clique) of G
- Branch-and-Bound
  - Partial solution Q: set of nodes, two nodes of Q are adjacent
  - Candidate nodes *Cand* for expansion: each node of *Cand* is adjacent with all nodes of Q
  - Upper Bound
    - ⋆ Δ is the number of colors used to color nodes of *Cand* such that two adjacent nodes $u, v \in Cand$ must be colored by different colors.
    - ⋆ The size of every complete subgraph of $G(Cand)$ is less than or equal to Δ
    - ⋆ $|Q| + \Delta$ is the upper bound of the size of cliques expanded from Q
    - ⋆ If $|Q| + \Delta \leq |Qmax|$, then do not expand Q

Branch-and-Bound: MaxClique

Algorithm 20: MaxClique($G = (V, E)$)

Input: Graph $G = (V, E)$
Output: Maximal complete subgraph of G
$Qmax \leftarrow \{\}$;
$Q \leftarrow \{\}$;
$Cand \leftarrow$ list of nodes of V;
$\Delta \leftarrow$ Sort($Cand$);
Expand($Cand$);
return $Qmax$;

Branch-and-Bound: MaxClique

Algorithm 21: Expand($Cand$)

Input: Sorted List of candidates $Cand$, $G = (V, E)$ and $Q, Qmax$ are global variables
Output: Expanding the partial solution Q
foreach $i = 0, \ldots, lenght(Cand) - 1$ **do**
    $u \leftarrow Cand[i]$;
    $Q \leftarrow Q \cup \{u\}$;
    if $|Q| > |Qmax|$ **then**
        $Qmax \leftarrow Q$;
    $Cand' \leftarrow \{v \in Cand \mid v \neq u \wedge (u, v) \in E\}$;
    $\Delta \leftarrow$ Sort($Cand'$);
    if $|Q| + \Delta > |Qmax|$ **then**
        Expand($Cand'$);
    $Q \leftarrow Q \setminus \{u\}$;

Branch-and-Bound: MaxClique

Algorithm 22: Sort($Cand$)

Input: Sort the List of candidates $Cand$
Output: Updated $Cand$ and return the number classes

$maxNo \leftarrow 0$;
$C_1 \leftarrow \{\}$;
foreach $u \in Cand$ **do**
    $k \leftarrow 1$;
    while $\exists v \in C_k \mid (u, v) \in E$ **do**
        $k \leftarrow k + 1$;
    if $k > maxNo$ **then**
        $maxNo \leftarrow k$;
        $C_k \leftarrow \{\}$;
    $C_k \leftarrow C_k \cup \{u\}$
$L \leftarrow []$;
foreach $k = 1, \dots, maxNo$ **do**
    foreach $v \in C_k$ **do**
        $L \leftarrow L :: v$;
foreach $i = 0, \dots, length(L) - 1$ **do**
    $Cand[i] \leftarrow L[length(L) - i - 1]$;
return $maxNo$;

Exercises

- Nurses Scheduling
- Balanced Courses Assignment
- Packing 2D rectangle items into the container
- MaxClique
- Networks analysis (count number of k-paths on a graph, a tree)