Hanoi University of Science and Technology
School of Information and Communications Technology

Discrete Mathematics

Nguyễn Khánh Phương

Department of Computer Science
School of Information and Communication Technology
E-mail: phuongnk@soict.hust.edu.vn

PART 1
COMBINATORIAL THEORY
(Lý thuyết tổ hợp)

PART 2
GRAPH THEORY
(Lý thuyết đồ thị)

Contents of Part 1

Chapter 0: Sets, Relations

Chapter 1: Counting problem

Chapter 2: Existence problem

Chapter 3: Enumeration problem

Chapter 4: Combinatorial optimization problem

Contents of Part 1: Combinatorial Theory

Chapter 1. Counting problem
- This is the problem aiming to answer the question: "How many ways are there that satisfy given conditions?" The counting method is usually based on some basic principles and some results to count simple configurations.
- Counting problems are effectively applied to evaluation tasks such as calculating the probability of an event, calculating the complexity of an algorithm

Chapter 2. Existence problem

In the counting problem, configuration existence is obvious; in the existence problem, we need to answer the question: "Is there a combinatorial configuration that satisfies given properties ?"

Chapter 3. Enumeration problem

This problem is interested in giving all the configurations that satisfy given conditions.

Chapter 4. Combinatorial optimization problem
- Unlike the enumeration problem, this problem only concerns the "best" configuration in a certain sense.
- In the optimization problems, each configuration is assigned a numerical value (which is the use value or the cost to construction the configuration), and the problem is that among the configurations that satisfy the given conditions, find the configuration with the maximum or minimum value assigned to it

Hanoi University of Science and Technology
School of Information and Communications Technology

Chapter 4

Combinatorial optimization problem

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

CONTENTS

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

1. Introduction to problem

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

1.1. General problem

- In many practical application problems of combinatorics, each configuration is assigned to a value equal to the rating of the worth using of the configuration for a particular use purpose.
- Then it appears the problem: Among possible combination configurations, determine the one that the worth using is the best. Such kind of problems is called the combinatorial optimization problem.

Example: The assignment problem: there are a number of *artists* and a number of *pictures*. Any artist can be assigned to perform any picture, incurring some *cost* that may vary depending on the artist-picture assignment. Find a way to assign pictures to artists such that the *total cost* of the assignment is minimized.

Example: The nurse scheduling problem

It involves the assignment of shifts and holidays to nurses. Each nurse has their own wishes and restrictions, as does the hospital. The problem is described as finding a schedule that both respects the constraints of the nurses and fulfills the objectives of the hospital. Conventionally, a nurse can work 3 shifts: day shift, night shift, late night shift.

Some constraints are:

- A nurse does not work the day shift, night shift and late night shift on the same day.
- A nurse may go on a holiday and will not work shifts during this time.
- A nurse does not do a late night shift followed by a day shift the next day.

State the combinatorial optimization problem

The combinatorial optimization problem in general could be stated as follows:

Find the min (or max) of function

$f(x) \to \min (\max)$,

with condition:

$x \in D$,

where D is a finite set.

Terminologies:

- $f(x)$ – objective function of problem,
- $x \in D$ - a solution
- D – set of solutions of problem.
- Set D is often described as a set of combinatorial configurations that satisfy given properties.
- Solution $x^* \in D$ having minimum (maximum) value of the objective function is called ***optimal solution***, and the value $f^* = f(x^*)$ is called ***optimal value*** of the problem.

1. Introduction to problem

1.1. General problem

1.2. Traveling salesman problem

1.3. Knapsack problem

1.4. Bin backing problem

Traveling Salesman Problem – TSP (Bài toán người du lịch)

- A salesman wants to travel n cities: 1, 2, 3,…, n.
- *Itinerary is a way of starting from a city, and going through all the remaining cities, each city exactly once, and then back to the starting city.*
- Given c_{ij} is the cost of going from city i to city j ($i, j = 1, 2, ..., n$),
- Find the itinerary with minimum total cost.

Traveling Salesman Problem – TSP

We have a 1-1 correspondence between a *itinerary*

$\pi(1) \to \pi(2) \to \ldots \to \pi(n) \to \pi(1)$

and a permutation $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$ of n natural numbers 1, 2,..., n.

Set the cost of itinerary:

$f(\pi) = c_{\pi(1),\pi(2)} + \ldots + c_{\pi(n-1),\pi(n)} + c_{\pi(n),\pi(1)}.$

Denote:

Π - set of all permutations of n natural numbers 1, 2, ..., n.

Traveling Salesman Problem – TSP

- Then, the TSP could be stated as the following combinatorial optimization problem:

  $$\min \{ f(\pi) : \pi \in \Pi \}.$$

- One could see that the number of possible itineraries is $n!$, but there are only $(n-1)!$ itineraries if the starting city is fixed.

1. Introduction to problem

1.1. General problem

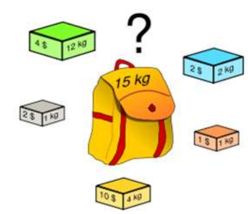
1.2. Traveling salesman problem

1.3. Knapsack problem

1.4. Bin backing problem

1.3. Knapsack Problem (Bài toán cái túi)

- Problem Definition
  - Want to carry essential items in one bag
  - Given a set of items, each has
    - An weight (i.e., 12kg)
    - A value (i.e., 4$)



- Goal
  - To determine the # of each item to include in the bag so that
    - The total weight is less than some given weight that the bag can carry
    - And the total value is as large as possible

1.3. Knapsack Problem (Bài toán cái túi)

- Three Types:
  - 0/1 Knapsack Problem
    - restricts the number of each kind of item to zero or one
  - Bounded Knapsack Problem
    - restricts the number of each item to a specific value
  - Unbounded Knapsack Problem
    - places no bounds on the number of each item

- Complexity Analysis
  - The general knapsack problem is known to be NP-hard
    - No polynomial-time algorithm is known for this problem

0/1 Knapsack Problem

- Problem: John wishes to take n items on a trip
  - The weight of item i is w_i and items are all different
  - The items are to be carried in a knapsack whose weight capacity is c
    - When sum of item weights $\leq$ c, all n items can be carried in the knapsack
    - When sum of item weights > c, some items must be left behind

Which items should be taken/left?

0/1 Knapsack Problem

- John assigns a profit p_i to item i
  - All weights and profits are positive numbers
- John wants to select a subset of the n items to take
  - The weight of the subset should not exceed the capacity of the knapsack (constraint)
  - Cannot select a fraction of an item (constraint)
  - The profit of the subset is the sum of the profits of the selected items (optimization function)
  - The profit of the selected subset should be maximum (optimization criterion)

- Let $x_i = 1$ when item i is selected and $x_i = 0$ when item i is not selected
  - Because this is a 0/1 Knapsack Problem, you can choose the item or not choose it.

0/1 Knapsack Problem

- A subset of items that John can carry with him can be represented by a binary vector of length n: $x = (x_1, x_2, \ldots, x_n)$, where $x_j = 1$ when item j is selected and $x_j = 0$ when item j is not selected, $j = 1,..,n$
- For each solution x, the profit of carried items is

$$f(x) = \sum_{j=1}^{n} p_j x_j,$$

The weight of carried items is

$$g(x) = \sum_{j=1}^{n} w_j x_j$$

0/1 Knapsack Problem

0/1 Knapsack problem could be stated in the form of the following combinatorial optimization problem:

Among binary vectors of length n that satisfy the condition $g(x) \leq c$, determine the vector x^* giving the maximum value of objective function $f(x)$:

$$\max \{ f(x): x \in A^n, g(x) \leq c \}.$$

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

1. Introduction to problem

1.4. Bin packing (Bài toán đóng thùng)

- Given n items: the weight of items are $w_1, w_2, ..., w_n$. Need to find a way to place these n items into bins of same size b such that the number of bins used is minimal.
- We have the assumption:

  $$w_i \leq b,\ i = 1, 2,.., n.$$
- Therefore, the number of bins needed to hold all these n items is not more than n. The problem is to find the minimum possible number of bins:
  - We will give user n bins. The problem is to help user to determine: each of the n item will be placed in which of the n bins, so that the number of bins having items in them is minimum.

1.4. Bin packing (Bài toán đóng thùng)

- We have the boolean variable

  $x_{ij} = 1$, if item i is placed in bin j,

  0, otherwise.

Then, bin backing problem is stated in the form:

$$\sum_{j=1}^{n} \operatorname{sgn}\left(\sum_{i=1}^{n} x_{ij}\right) \to \min,$$

$$\sum_{j=1}^{n} x_{ij} = 1, \quad i = 1, 2, ..., n$$

$$\sum_{i=1}^{n} w_i x_{ij} \leq b, \quad j = 1, 2, ..., n;$$

$$x_{ij} \in \{0,1\}, \ i, j = 1, 2, ..., n.$$

Note: The signum function of a real number x is defined as follows:

$$\operatorname{sgn}(x) := \begin{cases} -1 & \text{if } x < 0, \\ 0 & \text{if } x = 0, \\ 1 & \text{if } x > 0. \end{cases}$$

CONTENTS

1. Introduction to problem
2. **Brute force (Duyệt toàn bộ)**
3. Branch and bound (Thuật toán nhánh cận)

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

Method description

- One of the most obvious methods to solve the combinatorial optimization problem is: On the basis of the combinatorial enumeration algorithms, we go through each solution of the problem, and for each solution, we calculate its value of objective function; then compare values of objective functions of all solutions to find the optimal solution whose objective function is minimal (maximal).
- The approach based on such principles is called the brute force (phương pháp duyệt toàn bộ).

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

Example: 0/1 knapsack problem

- Consider 0/1 knapsack problem

$$\max\{f(x) = \sum_{j=1}^{n} p_j x_j : x \in D\},$$

$$\text{where } D = \{x = (x_1, x_2, \ldots, x_n) \in A^n : \sum_{j=1}^{n} w_j x_j \le c\}$$

- p_j, w_j, c are positive integers, $j=1,2,\ldots, n$.
- Need algorithm to enumerate all elements of set D

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

Backtracking: enumerate all possible solutions

- **Construct set S_k:**
  - $S_1=\{0, t_1\}$, where $t_1=1$ if $c \ge w_1$; $t_1 = 0$, otherwise
  - Assume the current partial solution is $(x_1, \ldots, x_{k-1})$. Then:
    - The remaining capacity of the bag is:

      $c_{k-1}= c - w_1x_1 - \ldots - w_{k-1}x_{k-1}$
    - The value of items already in the bag is:

      $f_{k-1}= p_1x_1 + \ldots + p_{k-1}x_{k-1}$

  Therefore: $S_k = \{0, t_k\}$, where $t_k=1$ if $c_{k-1} \ge w_k$; $t_k = 0$, otherwise
- **Implement S_k?**

```
for y in range (0, tk+1)
//in C++: for (y = 0; y++; y <= tk)
```

Program in pseudo code

```
int x[20], xopt[20], p[20], w[20];
int  n,b, ck, fk, fopt;

void InputData ( )
{
     <Enter value of n, p, w, b>;
}
void PrintSolution ( )
{
    <Optimal solution: xopt;
     Optimal value of objective
    function: fopt>;
}
```

```
int main( )
{
    InputData( );
    ck=c;
    fk= 0;
    fopt= 0;
    BackTrack(1);
    PrintSolution( );
}
```

```
void  BackTrack(int k)
{
    int j, t;
   if (ck>=w[k])  t=1  else t=0;      //if: Trọng lượng còn lại của túi >= trọng lượng đồ vật i
   for (j= t; j--; j>=0)
   {
      x[k] = j;//j = 1→ x[k] = 1 tức là cho đồ vật k vào túi; j = 0→ x[k] = 0 tức là không cho đồ vật k vào túi
      ck= ck - w[k]*x[k];  //bk: trọng lượng còn lại của túi
      fk= fk + p[k]*x[k];  //fk: giá trị hiện tại của túi
      if  (k == n)   //nếu tất cả n đồ vật đều đã được xét
      {
         if (fk>fopt) {   //nếu giá trị hiện tại của túi > giá trị tốt nhất của túi hiện có
               xopt:=x; fopt:=fk;  //Cập nhật giá trị tốt nhất
         }
      }
      else BackTrack(k+1);  //xét tiếp đồ vật thứ k+1
      ck = ck+w[k]*x[k];
      fk = fk - p[k]*x[k];
   }
}
```


Comment

- Brute force is difficult to do even on the most modern super computer. Example to enumerate all

15! = 1 307 674 368 000

permutations on the machine with the calculation speed of 1 billions operations per second, and if to enumerate one permutation requires 100 operations, then we need 130767 seconds > 36 hours!

20! ===> 7645 years

Comment

- However, it must be emphasized that in many cases (for example, in the traveling salesman problem, the knapsack, bin backing problem), we have not found yet any effective methods other than the brute force.

Comment

- Then, a problem arises that in the process of enumerating all solutions, we need to make use of the found information to eliminate solutions that are definitely not optimal.
- In the next section, we will look at such a search approach to solve the combinatorial optimization problems. In literature, it is called Branch and bound algorithm (Thuật toán nhánh cận).

CONTENTS

3. Branch and bound (Thuật toán nhánh cận)

3.1. General diagram

- Branch and bound algorithm consists of 2 procedures:
  - Branching Procedure (Phân nhánh)
  - Bounding Procedure (Tính cận)
- **Branching procedure:** The process of partitioning the set of solutions into subsets of size decreasing gradually until the subsets consists only one element. (*Quá trình phân hoạch tập các phương án ra thành các tập con với kích thước càng ngày càng nhỏ cho đến khi thu được phân hoạch tập các phương án ra thành các tập con một phần tử)*
- **Bounding procedure:** It is necessary to give an approach to calculate the bound for the value of the objective function on each subset A in the partition of the set of solutions. (*Cần đưa ra cách tính cận cho giá trị hàm mục tiêu của bài toán trên mỗi tập con A trong phân hoạch của tập các phương án.)*

3.1. General diagram

- We will describe the idea of algorithm on the model of the following general combinatorial optimization problem:

$$\min \{ f(x) : x \in D \},$$

where D is the finite set.

- Assume set D is described as following:

$$D = \{x = (x_1, x_2, ..., x_n) \in A_1 \times A_2 \times ... \times A_n:$$
$$x \text{ satisfies property } P\},$$

where $A_1, A_2, ..., A_n$ are finite set, and P is property on the Descartes product $A_1 \times A_2 \times ... \times A_n$.

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

Comment

- The requirement about describe the set D is to be able to use the backtracking algorithm to enumerate all solutions of the problem.
- Problem

$$\max \{f(x): x \in D\}$$

is the equivalent of the problem

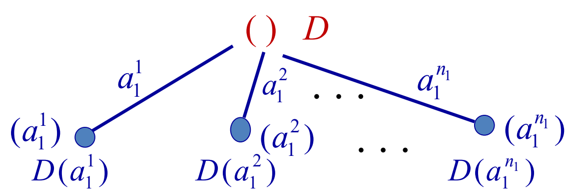
$$\min \{g(x): x \in D\}, \text{ where } g(x) = -f(x)$$

Therefore, we can limit to considering the minimize problem

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

Branching procedure

Branching procedure can implement by using backtracking:



where $D(a_1^i) = \{x \in D : x_1 = a_1^i\}, i = 1, 2, ..., n_1$

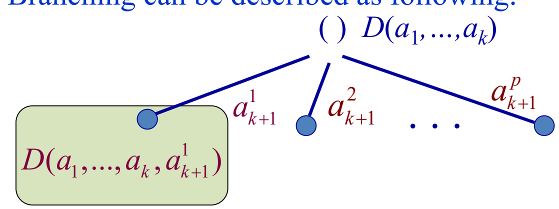
is the set of solutions which can be obtained from partial solution (a_1^i)

We have the partition:

$$D = D(a_1^1) \cup D(a_1^2) \cup ... \cup D(a_1^{n_1})$$

Branching

- Branching can be described as following:



➢ We have partition: $D(a_1, ..., a_k) = \bigcup_{i=1}^{p} D(a_1, ..., a_k, a_{k+1}^i)$

$D(a_1,..., a_k) = \{ x \in D: x_i = a_i, i = 1,..., k \}$ is a subset of solutions where the first k elements of solutions are already known: $x_1 = a_1, x_2 = a_2, ..., x_k = a_k$

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

Bounding

- We need to determine function g defined on the set of all partial solutions that satisfies the following inequality:

$$g(a_1,\dots, a_k) \le \min\{\boxed{f(x): x\in D(a_1, \dots, a_k)}\} \quad (*)$$

The value of objective function of all solutions having the first k elements as $(a_1, a_2,\dots,a_k)$

For each k-level partial solution $(a_1, a_2, \dots, a_k)$, $k = 1, 2, \dots$

- The inequality (*) means that the value of g of partial solution $(a_1, a_2, \dots, a_k)$ is not greater than the minimum value of objective function of solution set $D(a_1, \dots,a_k)$

$$D(a_1,\dots, a_k)= \{ x\in D: x_i = a_i , i = 1,\dots, k \},$$

In other word, $g(a_1, a_2, \dots, a_k)$ is the **lower bound** of the value of objective function of solution set $D(a_1, a_2, \dots, a_k)$.

Cut branch by using lower bound

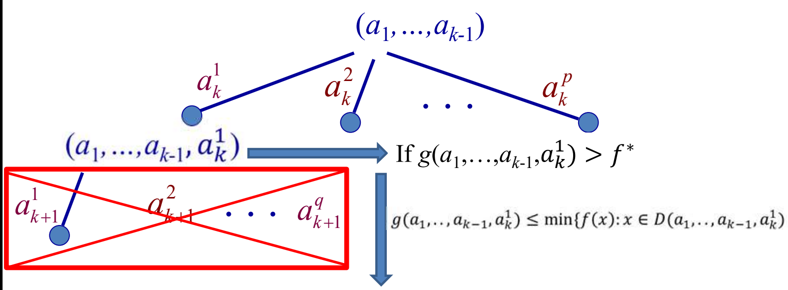
- Assume we already have function g defined as above. We will use this function to reduce the amount of searching during the process to consider all possible solutions in the backtracking algorithm.
- In the process to enumerate solutions, assume we already obtain some solutions. Thus, denote x^* the solution with objective function is minimum among all solutions obtained so far, and denote $f^* = f(x^*)$
- We call
  - x^* is the current best solution (optimal solution),
  - f^* is the current best value of objective function (optimal objective value).

Cut branch by using lower bound

f^* is the current best objective value

If $g(a_1,\dots,a_{k-1},a_k^2) > f^*$

If $g(a_1,\dots,a_{k-1},a_k^p) > f^*$



→all solutions with first k elements as $(a_1,\dots,a_{k-1},a_k^1)$ certainly have the objective value > f^* → we do not need to browse this branch

$g(a_1,\dots, a_k)$ is lower bound of partial solution $(a_1,\dots, a_k)$

Branch and bound

```
void  Try(int k)
{
    //Construct x_k from partial soluion (x_1, x_2, ..., x_k-1)
   for  a_k∈A_k
      if   (a_k∈S_k )
      {
        x_k =  a_k;
        if   (k == n)  <Update Record>;
        else if   (g(x_1,..., x_k)  ≤ f*) Branch(k+1);
       }
}
```

Note that if in the procedure Try, we replace the statement

```
if   (k == n)  <Update Record>;
else if   (g(x_1,..., x_k)  ≤ f*) Try(k+1);
```

by

```
if   (k == n)  <Update Record>;
else  Try(k+1);
```

Then the algorithm is the exhaustive search, not Branch and bound anymore

```
void  BranchAndBound ( )
{
   f* = +∞;
   //if you know any solution x* then set f* = f(x*)
   Try(1);
   if  (f*  <  +∞)
            <f* is the optimal objective value, x* is optimal solution >
   else  <  problem does not have any solutions >;
}
```

Note:

$g(a_1,..., a_k) \le \min\{f(x): x \in D(a_1, ..., a_k)\}$ (*)

The construction of g function depends on each specific combinatorial optimization problem. Usually we try to build it so that:

- Calculating the value of g must be simpler than solving the combinatorial optimization problem on the right side of (*).
- The value of $g(a_1, ..., a_k)$ must be close to the value of the right side of (*).

Unfortunately, these two requirements are often contradictory in practice.

3. Branch and bound

3.1. General diagram

3.2. Example

3.2.1. Traveling salesman problem

3.2.2. Knapsack problem

Sir William Rowan Hamilton
1805 - 1865

Traveling Salesman Problem – TSP (Bài toán người du lịch)

- A salesman wants to travel n cities: 1, 2, 3,…, n.
- *Itinerary is a way of starting from a city, and going through all the remaining cities, each city exactly once, and then back to the starting city.*
- Given c_{ij} is the cost of going from city i to city j (i, j = 1, 2, ..., n),
- Find the itinerary with minimum total cost.

3.2.1. Traveling salesman problem

Fix the starting city as city 1, the TSP leads to the problem:

- Determine the minimum value of

$f(1,x_2,..., x_n) = c[1,x_2]+c[x_2,x_3]+...+c[x_{n-1},x_n] + c[x_n,1]$

where

$(x_2, x_3, ..., x_n)$ is permutation of natural numbers 2, ..., n.

Lower bound function (Hàm cận dưới)

- Denote

  $c_{min} = \min \{ c[i, j] , i, j = 1, 2, ..., n, i \neq j \}$

  the smallest cost between all pairs of cities.
- We need to evaluate the lower bound for the partial solution $(1, x_2, ..., x_k)$ corresponding to the partial journey that has passed through k cities

  $1 \to x_2 \to . . . \to x_{k-1} \to x_k$

Lower bound function

- The cost need to pay for this partial solution is

  $\sigma = c[1,x_2] + c[x_2, x_3] + ... + c[x_{k-1}, x_k]$.
- To develop it as the complete journey:

$$\underbrace{1 \to x_2 \to . . . \to x_{k-1} \to x_k}_{\text{Cost: } \sigma} \underbrace{\to \overset{①}{x_{k+1}} \to \overset{②}{x_{k+2}} \to \overset{③}{.....} \to x_n \to 1}_{\text{Cost: } (n-k+1)c_{min}}$$

We still need to go through n-k+1 segments, each segment with the cost at least c_{min}, thus the lower bound of the partial solution $(1, x_2, ..., x_k)$ can be calculated by the formula:

$$g(1, x_2, ..., x_k) = \sigma + (n-k+1)\, c_{min}$$

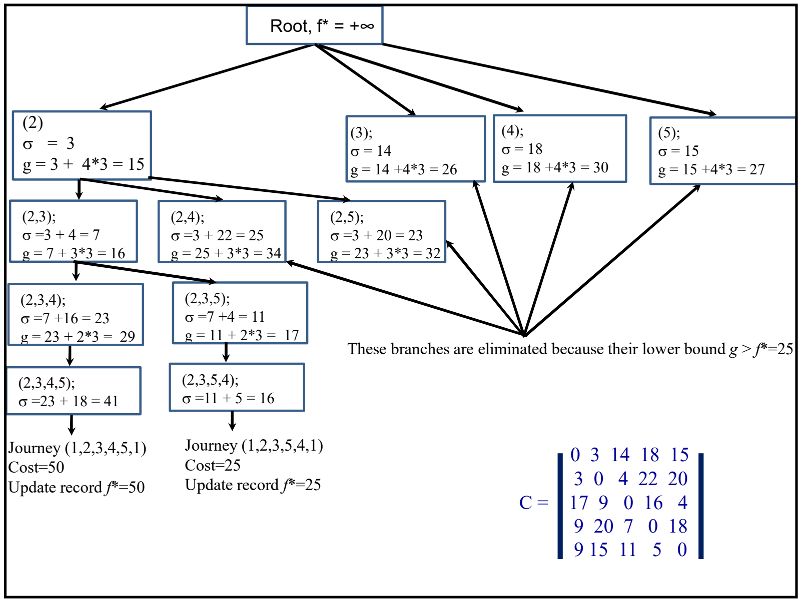
Example 1

Give 5 cities {1, 2, 3, 4, 5}. Solve the TSP where the salesman starts from the city 1, and the cost matrix:

$$C = \begin{vmatrix} 0 & 3 & 14 & 18 & 15 \\ 3 & 0 & 4 & 22 & 20 \\ 17 & 9 & 0 & 16 & 4 \\ 9 & 20 & 7 & 0 & 18 \\ 9 & 15 & 11 & 5 & 0 \end{vmatrix}$$

Example 1

- We have c_{min} = 3. The process executing the algorithm is described by the solution search tree.
- Information written in each box is the following in order:
  - elements of partial solution,
  - σ is the cost of partial solution,
  - g – lower bound of partial solution.



Result

Terminate the algorithm, we obtain:

- Optimal solution (1, 2, 3, 5, 4, 1) correspond to the journey

  $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 1$,

- The minimum cost is 25.

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

Implementation

```
void  Try(int k)
{
    for (int v = 1; v<=n;v++) {
       if (visited[v] == FALSE) {
              x_k = v; visited[v] = TRUE;
              f = f + c(x_{k-1},x_k);
              if (k == n) //Update record
              {  if (f + c(x_n,x_1) < f*) f* = f + c(x_n,x_1); }
              else {
                     g = f + (n-k + 1)*cmin; //calculate bound
                     if (g < f* ) Try(k + 1);
                  }
               f = f - c(x_{k-1},x_k);
               visited[v] = FALSE;
        }//end if
    }//end for
}
```

```
void  Try(int k)
{
    //Construct x_k from partial solution (x_1, x_2, ..., x_{k-1})
   for  a_k∈A_k
      if   (a_k∈S_k )
      {
        x_k =  a_k;
        if   (k == n)  <Update Record>;
        else if   (g(x_1,..., x_k)  ≤ f*) Try(k+1);
       }
}
```

Implement

```
void  BranchAndBound()
{
    f* = +∞;
    for (v = 1; v<=n;v++) visited[v] = FALSE;
    f=0; x_1 = 1; visited[x_1] = TRUE;
    Try(2);
    return f*;
}
```

```
void  BranchAndBound ( )
{
   f* = +∞;
   //if you know any solution x* then set f* = f(x*)
   Branch(1);
   if  (f*  <  +∞)
            <f* is the optimal objective value, x* is optimal solution >
   else  <  problem does not have any solutions >;
}
```

Example 2

Consider 5 cities {1, 2, 3, 4, 5}. Using the branch and bound algorithm to solve the TSP where salesman starts from the city 1 and the cost matrix:

$$\begin{vmatrix} 0 & 4 & 10 & 2 & 17 \\ 4 & 0 & 4 & 11 & 9 \\ 10 & 4 & 0 & 12 & 3 \\ 2 & 11 & 12 & 0 & 13 \\ 17 & 9 & 3 & 6 & 0 \end{vmatrix}$$

3. Branch and bound

3.1. General diagram

3.2. Example

3.2.1. Traveling salesman problem

3.2.2. Knapsack problem

Sir William Rowan Hamilton
1805 - 1865

3.2.2. Knap sack problem (Bài toán cái túi)

- There are n types of items.
- Item type j has
  - weight w_j and
  - profit p_j $(j = 1, 2,..., n)$.
- We need to select subsets of these items to put it into the bag of capacity c such that the total profit obtained from items loaded in the bag is maximum.

3.2.2. Knap sack problem (Bài toán cái túi)

- We have the variable

  x_j – number items type j loaded in the bag, j=1,2, ..., n
- Mathematical model of problem: Find

$$f^* = \max \{ f(x) = \sum_{j=1}^{n} p_j x_j : \sum_{j=1}^{n} w_j x_j \le c, x_j \in Z_+, j = 1, 2, ..., n \}$$

where Z_+ is the set of nonnegative integers

Knapsack problem with integer variables

- Denote D the set of solutions to the problem:

$$D = \{ x = (x_1, ..., x_n) : \sum_{j=1}^{n} p_j x_j \le c, x_j \in Z_+, j = 1, 2, ..., n \}$$

Construct upper bound

- Assume we index the item in the order such that the following inequality is satisfied:

$$p_1 / w_1 \geq p_2 / w_2 \geq . . . \geq p_n / w_n .$$

(it means items are ordered in descending order of profit per one unit of weight)

- To construct the upper bound function, we consider the following $\boxed{\text{Knapsack continuous variables (KPC)}}$: Find

$$\boxed{g^* = \max \{ f(x) = \sum_{j=1}^{n} p_j x_j : \sum_{j=1}^{n} w_j x_j \leq c, \boxed{x_j \geq 0,} j = 1,2,...,n \}}$$

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

Construct upper bound function

Proposition. *The optimal solution to the KPC is vector ($x^* = x_1^*, x_2^*,.., x_n^*$) where elements are determined by the formula:*

$x_1^* = c/w_1, x_2^* = x_3^* = \cdots . = x_n^* = 0$

and the optimal value is $g^* = v_1 b / w_1$.

Proof. Consider $x = (x_1,..., x_n)$ as a solution to the KPC. Then

$$p_j \leq (p_1 / w_1) w_j , j = 1, 2, ..., n$$

as $x_j \geq 0$, we have

$$p_j x_j \leq (p_1 / w_1) w_j x_j , j = 1, 2, ..., n.$$

- Therefore

$$\sum_{j=1}^{n} p_j x_j \leq \sum_{j=1}^{n} (p_1 / w_1) w_j x_j$$
$$= (p_1 / w_1) \sum_{j=1}^{n} w_j x_j$$
$$\leq (p_1 / w_1) c = g^*$$

Proposition is proved.

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

Calculate the upper bound

- Now we have the k-level partial solution: $(u_1, u_2, ..., u_k)$, then the profit of items currently loaded in the bag is

$$\sigma_k = p_1 u_1 + p_2 u_2 + . . . + p_k u_k$$

and the remaining capacity of the bag is

$$c_k = c - (w_1 u_1 + w_2 u_2 + . . . + w_k u_k)$$

- We have:

$$\max\{ f(x) : x \in D, x_j = u_j, j = 1,2,...,k\}$$
$$= \max \{\sigma_k + \sum_{j=k+1}^{n} p_j x_j : \sum_{j=k+1}^{n} w_j x_j \leq c_k, \ x_j \in Z_+, j = k+1, k+2,...,n\}$$
$$\leq \sigma_k + \max \{\sum_{j=k+1}^{n} p_j x_j : \sum_{j=k+1}^{n} w_j x_j \leq c_k, \ x_j \geq 0, j = k+1, k+2,...,n\}$$
$$= \sigma_k + \boxed{p_{k+1} c_k / w_{k+1}}$$

With remaining capacity c_k: take the item (k+1)th having the most profit to put into the bag

- Thus, we can calculate the upper bound for the partial solution $(u_1, u_2, ..., u_k)$ by formula

$$g(u_1, u_2,..., u_k) = \sigma_k + p_{k+1} c_k / w_{k+1}$$

Calculate the upper bound

- **Note:** When continuing build the $(k+1)$th element of solution, candidates for x_{k+1} are $0, 1, ..., [c_k / w_{k+1}]$
- Using the result of the proposition, when selecting value for x_{k+1}, we browse candidates for x_{k+1} in the descending order: $[c_k / w_{k+1}], [c_k / w_{k+1}]-1,…,1, 0$

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST

Example 1

- Solve the knap sack problem using the branch and bound algorithm:

$$f(x) = 10\,x_1 + 5\,x_2 + 3\,x_3 + 6\,x_4 \to \max,$$
$$5\,x_1 + 3\,x_2 + 2\,x_3 + 4\,x_4 \le 8,$$
$$x_j \in Z_+,\ j = 1, 2, 3, 4.$$

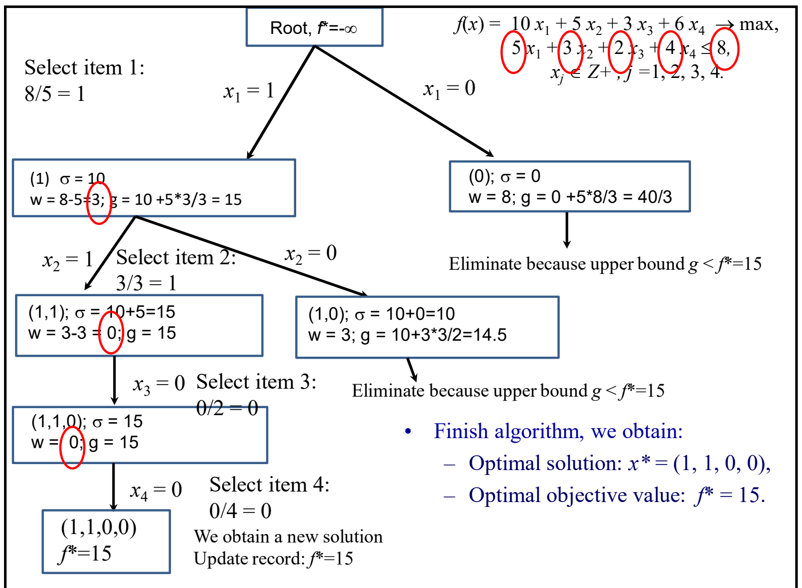
- Note that in this example, all four items are already sorted in descending order of profit on an unit weight

$$\frac{10}{5} = 2 > \frac{5}{3} \approx 1,66 > \frac{3}{2} = 1,5 = \frac{6}{4}$$

Example 1

- The process executing the algorithm is described by the solution search tree.
- Information written in each box is the following in order:
  - elements of partial solution,
  - σ is the cost of partial solution (profit of items currently loaded in the bag),
  - w : remaining capacity of the bag,
  - g : upper bound of partial solution.

NGUYỄN KHÁNH PHƯƠNG
CS-SOICT-HUST



- Finish algorithm, we obtain:
  - Optimal solution: $x^* = (1, 1, 0, 0)$,
  - Optimal objective value: $f^* = 15$.

Example 2

- Solve the knap sack problem using the branch and bound algorithm:

$$5x_1 + 8x_2 + x_3 \to \max$$
$$2x_1 + 3x_2 + x_3 \le 13$$
$$x_1, x_2, x_3 \ge 0,\ \text{integers}$$

Note that in this example, all three items are NOT yet sorted in descending order of profit on an unit weight. We need to reorder them as :

2, 1, 3