เอกสารประกอบการอบรม

Backtracking and Branch-and Bound

ค่ายคอมพิวเตอร์โอลิมปิก สอวน. ค่าย 2 2/2567
ศูนย์โรงเรียนสามเสนวิทยาลัย - มหาวิทยาลัยธรรมศาสตร์
ระหว่างวันที่ 10 มีนาคม – 26 มีนาคม 2568

โดย

สาขาวิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์และเทคโนโลยี
มหาวิทยาลัยธรรมศาสตร์

- More on Exhaustive Search
- Backtracking
- Branch-and-Bound

ผศ.ดร.ฐาปนา บุญชู

17 มีนาคม 2568

ทบทวน Exhaustive Search

Exhaustive Search

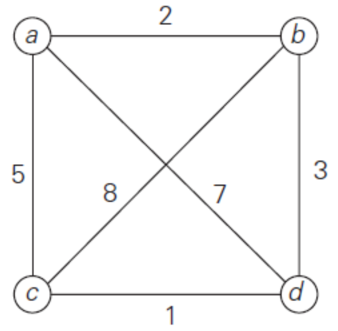
- ปัญหาสำคัญหลายปัญหาเป็นการค้นหาองค์ประกอบ (Element) ที่มีคุณลักษณะเฉพาะใน Space ที่เติบโต (Grow) อย่างรวดเร็วในระดับ Exponential หรือเร็วกว่านั้น ตาม Instance size
- โดยทั่วไปปัญหาเหล่านี้มักจะเกี่ยวข้องกับ Combinatorial objects
    - Permutations
    - Combinations
    - Subsets ของ Set ที่สนใจ
- หลาย ๆ ปัญหาเป็น Optimization problems ที่ค้นหาองค์ประกอบที่หาค่าสูงหรือต่ำสุดของคุณลักษณะบางอย่าง เช่น Path length หรือ Assignment cost

Exhaustive Search or Complete Search

- Also known as brute force or (recursive) backtracking.
- A method for solving a problem by traversing the entire (or part of the) search space to obtain the required solution.
  - ในระหว่างค้นหา เราสามารถ Prune บางส่วนของ Search space ได้ถ้าเราพิจารณาแล้วว่ามันไม่น่าจะมี Solution อยู่
- เราจะใช้ Exhaustive Search เมื่อเรารู้ว่าไม่มี Algorithms อื่นที่ใช้แก้ปัญหานี้ได้แล้ว
  - อย่าลืมว่า Exhaustive Search นั้นง่าย และไม่เกิด Wrong Answers! แต่จะเกิด Time Limit Exceeded (TLE) แทน :P

Traveling salesman problem (TSP):

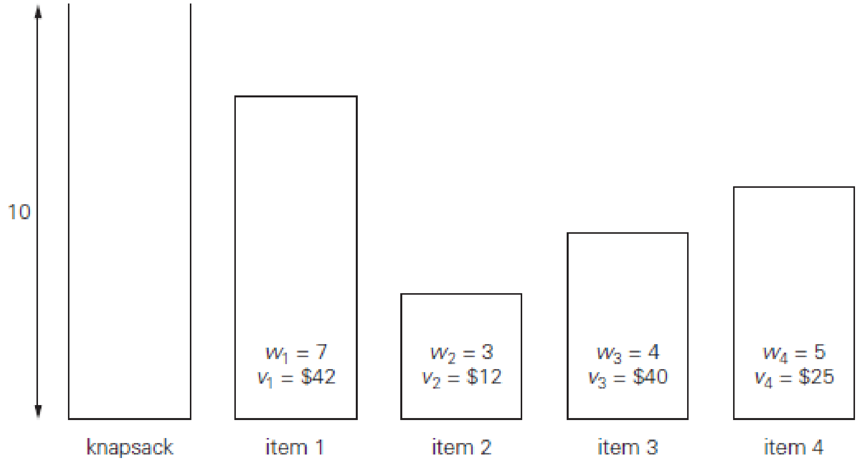
- **Traveling salesman problem (TSP):** ปัญหาคือการค้นหา Shortest tour จาก Set ของ n เมืองที่ Tour นี้จะเยี่ยมแต่ละเมืองเพียงครั้งเดียวเท่านั้นก่อนที่เดินทางกลับเมืองเริ่มต้น
  - โดยเราสามารถจำลองปัญหานี้มาเป็น Weighted graph ได้ (Vertices: เมือง, Edge weight: ระยะทางระหว่างเมือง)
  - เป็นปัญหาการหา **Shortest Hamiltonian circuit** ของกราฟ ที่ถูกนิยามโดย ลำดับ n + 1 adjacent vertices $v_{i_0}, v_{i_1}, \ldots, v_{i_{n-1}}, v_{i_0}$ โดย $v_{i_1}, \ldots, v_{i_{n-1}}$ จะไม่ซ้ำกัน
  - เราสามารถ Generate ทุก Tours ที่เป็นไปได้โดย Permutations ของ n-1 เมืองระหว่างกลาง หาเส้นทางของ Tour เลือกเส้นที่สั้นที่สุด
  - จำนวน Permutation ทั้งหมดคือ $\frac{1}{2}(n-1)!$



Tour	Length	
a —> b —> c —> d —> a	l = 2 + 8 + 1 + 7 = 18	
a —> b —> d —> c —> a	l = 2 + 3 + 1 + 5 = 11	optimal
a —> c —> b —> d —> a	l = 5 + 8 + 3 + 7 = 23	
a —> c —> d —> b —> a	l = 5 + 1 + 3 + 2 = 11	optimal
a —> d —> b —> c —> a	l = 7 + 3 + 8 + 5 = 23	
a —> d —> c —> b —> a	l = 7 + 1 + 8 + 2 = 18	

Knapsack Problem

- มีของจำนวน n ชิ้น โดยแต่ละชิ้นมีน้ำหนัก (Weight) คือ $w_1, \dots, w_n$ และมูลค่า (Value) $v_1, \dots, v_n$ และถุง (Knapsack) ที่มีความจุ W ให้หา Subset ของที่สามารถบรรจุลงในถุงได้โดยที่มูลค่ารวมใน Subset นั้นมีมูลค่ามากที่สุด
- **Exhaustive search** คือการ **Generate ทุก Subset ที่เป็นไปได้** แล้วหาผลรวมของ Weights และ Values ใน Subset เหล่านั้นที่เป็นไปตามข้อกำหนด
- เนื่องจากจำนวนของ Subset ที่เป็นไปได้ทั้งหมดคือ 2^n ทำให้ Exhaustive search ใช้เวลา $\Omega(2^n)$ => NP-hard problems (ไม่มีใครรู้จัก Polynomial-time algorithm ใดที่แก้ NP-hard Problem – ยังไม่มีการพิสูจน์แต่ Computer scientists เชื่อแบบนั้น)
    - Backtracking และ Branch and Bound สามารถแก้บาง Instances ของปัญหานี้ได้



(a)

Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

(b)

Assignment Problem

- มีจำนวน n คนเพื่อถูกมอบหมาย (Assign) ให้ทำงาน n งาน (1 คนต่อ 1 งาน)
- โดย C[i, j] คือราคา (Cost) ที่ต้องจ่ายเมื่อมอบหมายงานคนที่ i-th ไปยังงานที่ j-th
  - โดย i, j = 1, 2, …, n
- โจทย์คือต้องการหาการมอบหมาย (Assignment) ที่มีราคาน้อยมากที่สุด
- พิจารณา Cost matrix ด้านบน โจทย์คือ ในแต่ละแถวให้เลือกหนึ่งค่าโดยที่แต่ละแถวจะเลือกค่าในคอลัมน์ที่ซ้ำกันไม่ได้และผลรวมทั้งหมดทั้งต้องมีค่าน้อยมากที่สุด
- Exhaustive search คือการที่เราต้อง **<u>Generate ทุก Permutation ของ 1, …, n</u>** หา Cost รวมแล้วหาค่าที่น้อยที่สุด
- สามารถใช้ Hungarian method แก้ได้อย่างมีประสิทธิภาพมากขึ้น

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4> cost = 9 + 4 + 1 + 4 = 18
<1, 2, 4, 3> cost = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4> cost = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2> cost = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3> cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2> cost = 9 + 7 + 1 + 6 = 23

Iterative Complete Search

- พิจารณาปัญหาต่อไปนี้
  - ให้หาตัวเลข 5 หลัก (5-digit numbers) ที่ใช้เลขโดด 0-9 เพียงครั้งเดียว โดยที่เลขตัวแรกถูกหารด้วยเลขตัวที่ 2 แล้วเท่ากับ N (2 <= N <= 79) กล่าวคือ abcde/fghij = N โดยที่ตัวอักษรแต่ละตัวแทนเลขโดดใน 0-9 เลขตัวแรกเป็น 0 ได้
  - ตัวอย่าง 79546/01283 = 62; 94736/01528 = 62;
  - วิเคราะห์ปัญหา
    - fghij (ตัวหาร) เป็นได้ตั้งแต่ 01234 – 98765 (ประมาณ 100K ตัวเลขที่เป็นไปได้)
    - พิจารณา N=2 จะลดความเป็นไปได้จาก 100K ตัวเลขเหลือ 50K ตัวเลข เนื่องจากช่วงที่ fghij จะเป็นตั้งแต่ 01234 – 98765/2 และยิ่งลดลงไปเมื่อเพิ่ม N
    - แต่ละ Solution หาจากการเอา fghij x N แล้วเช็คว่าเท่ากับ abcde ไหม? และเช็คต่อว่า abcde และ fghij นั้นต่างกันไหม

Iterative Complete Search

```
for (int fghij = 1234; fghij <= 98765/N; ++fghij) {
  int abcde = fghij*N;                            // as discussed above
  int tmp, used = (fghij < 10000);                // flag if f = 0
  tmp = abcde; while (tmp) { used |= 1<<(tmp%10); tmp /= 10; }
  tmp = fghij; while (tmp) { used |= 1<<(tmp%10); tmp /= 10; }
  if (used == (1<<10)-1)                          // all 10 digits are used
    printf("%05d / %05d = %d\n", abcde, fghij, N);
}
```

ปัญหานี้ใช้ประมาณ 50K x 10 = 500K

*Prune บาง Search space ออกไป

Bit shifting:

a << b

หมายถึง ให้ทำการ shift bit ตัวเลข a ไปทางซ้าย b bits (ถ้าเกินให้ discard ไป)

a >> b

หมายถึง ให้ทำการ shift bit ตัวเลข a ไปทางซ้าย b bits (ถ้าเกินให้ discard ไป)

Iterative Complete Search (Many Nested Loops)

- พิจารณาปัญหาต่อไปนี้
  - ให้เลขตัวนวนเต็มมา 1 ชุด (6 < n < 13) โดยเลขชุดนี้ถูกเรียงลำดับมาแล้ว ให้หาทุก subset ขนาด 6 ที่เป็นไปได้ของเลขชุดนี้
  - วิเคราะห์ปัญหา
    - เนื่องจาก subset ถูกกำหนดว่าต้องมีขนาด 6 เท่านั้น และ output ยังต้องเรียงลำดับ
    - Solution คือ ลูป 6 ชั้น (Upper bound อยู่ที่ C(12, 6) = 924 เท่านั้น)

```
for (int i = 0; i < k; ++i) scanf("%d", &S[i]);  // input: k sorted ints
for (int a = 0  ; a < k-5; ++a)                  // six nested loops!
  for (int b = a+1; b < k-4; ++b)
    for (int c = b+1; c < k-3; ++c)
      for (int d = c+1; d < k-2; ++d)
        for (int e = d+1; e < k-1; ++e)
          for (int f = e+1; f < k  ; ++f)
            printf("%d %d %d %d %d %d\n",S[a],S[b],S[c],S[d],S[e],S[f]);
```

Iterative Complete Search (Loops+Pruning)

- กำหนดตัวเลข 3 ตัว A B และ C โดย (1 <= A, B, C <= 10000) ให้หา x y และ z ที่แตกต่างกันที่ทำให้ x+y+z = A และ x*y*z = B และ $x^2+y^2+z^2 = C$
- วิเคราะห์ปัญหา
  - C ค่าที่ใหญ่ที่สุดที่เป็นไปได้คือ 10000
  - ถ้าพิจารณา $x^2+y^2+z^2 = C$
    - กรณีที่ y=1, z=2, ดังนั้น x จะสามารถเป็นไปได้ในช่วง [-100, 100]

```
bool sol = false; int x, y, z;
for (x = -100; x <= 100; ++x)                        // ~201^3 ~= 8M operations
  for (y = -100; y <= 100; ++y)
    for (z = -100; z <= 100; ++z)
      if ((y != x) && (z != x) && (z != y) &&        // all 3 must be different
          (x+y+z == A) && (x*y*z == B) && (x*x + y*y + z*z == C)) {
        if (!sol) printf("%d %d %d\n", x, y, z);
        sol = true;
      }
```

ชวนคิด: ลองคิดว่าจะมี Idea ที่ Prune ได้เยอะกว่านี้

Iterative Complete Search (Permutations)

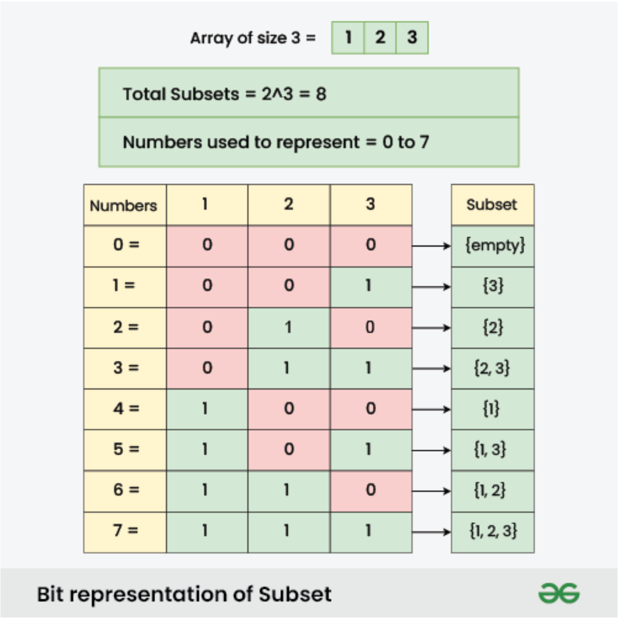
- มีคนชมภาพยนตร์ n คน (1 <= n <= 8) โดยทุกคนจะนั่งติดกัน แต่อาจจะมีข้อจำกัดต่าง ๆ (มีไม่เกิน 20 ข้อจำกัด) ว่า คู่ของคนใน 8 คนนี้จะต้องนั่งห่างกันอย่างน้อย m ที่นั่ง จะมีทั้งหมดกี่ arrangements ที่เป็นไปได้
- วิเคราะห์ปัญหา
  - ปัญหานี้คือปัญหา Permutations
  - พิจารณาแต่ละ Permutations แล้วตรวจสอบทุกข้อจำกัด จำนวนทั้งหมดที่ต้องตรวจสอบทั้งหมดประมาณ n! x 20 และ 1 <= n <= 8 ดังนั้น 8! x 20 = 806400 การดำเนินการ (ยังอยู่ในขอบเขตที่รับได้)

```
#include <bits/stdc++.h>  // next_permutation is inside C++ STL <algorithm>
// the main routine
int i, n = 8, p[8] = {0, 1, 2, 3, 4, 5, 6, 7};   // the first permutation
do {                                             // try all n! permutations
  // test each permutation 'p' in O(m)
}
while (next_permutation(p, p+n));                // complexity = O(n! * m)
```

Iterative Complete Search (Subsets)

- มีชุดตัวเลขจำนวนเต็ม (1 <= n <= 20) มี subset ของเลขชุดดังกล่าวนี้หรือไม่ที่มีผลรวมเท่ากับค่า X
  - วิเคราะห์ปัญหา
    - เราสามารถลองทุก ๆ 2^n Subsets ที่เป็นไปได้ แล้วหาผลรวมของทุก ๆ ค่าใน Subset ใน O(n) จะได้ว่า Time complexity คือ $O(n \times 2^n)$
    - Input size ที่ใหญ่ที่สุดคือ 21M
    - วิธีการเขียนโปรแกรมเพื่อ generate ทุก ๆ subset ที่เป็นไปได้คือการใช้ binary representation ของเลข 0 ถึง $2^n - 1$ และคำนวณได้เลย

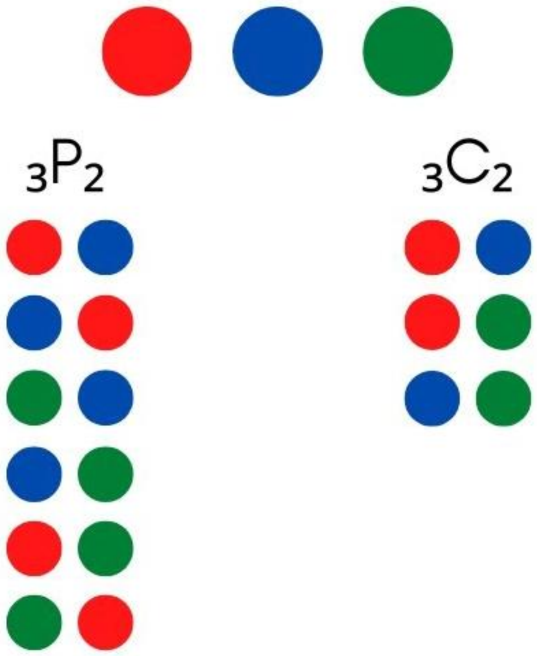
```
// the main routine, variable 'i' (the bitmask) has been declared earlier
for (i = 0; i < (1<<n); ++i) {                    // for each subset, O(2^n)
  int sum = 0;
  for (int j = 0; j < n; ++j)                     // check membership, O(n)
    if (i & (1<<j))                               // see if bit 'j' is on?
      sum += l[j];                                // if yes, process 'j'
  if (sum == X) break;                            // the answer is found
}
```



Bit representation of Subset

ข้อสังเกตุ

- ขนาดของข้อมูลเข้าของปัญหาที่ผ่านมา ?
- Combination (^nC_r) vs Permutation (^nP_r)
- ใช้ next_permutation ในการสร้าง combination ?
  - ถ้ามี Element ที่ซ้ำกันใน next_permutation จะเกิดอะไรขึ้น




```
#include <iostream>
#include <algorithm>
#include <vector>


int main() {

    int a[] = {1, 1, 3, 4};
    int count=0;

    do {
        count++;
    } while (std::next_permutation(a, a+4));


    std::cout << count << std::endl;

    return 0;
}
```

```
#include <iostream>
#include <algorithm>
#include <vector>


int main() {

    int a[] = {1, 2, 3, 4};
    int count=0;

    do {
        count++;
    } while (std::next_permutation(a, a+4));


    std::cout << count << std::endl;

    return 0;
}
```

combination

```
#include <iostream>
#include <vector>
#include <algorithm>

void generate_combinations(std::vector<int> &arr, int k) {
    std::sort(arr.begin(), arr.end()); // Sort to start with the smallest permutation

    std::vector<bool> select(arr.size(), false);
    std::fill(select.end() - k, select.end(), true); // Mark k elements as true

    do {
        // Print the current combination
        for (size_t i = 0; i < arr.size(); ++i) {
            if (select[i]) {
                std::cout << arr[i] << " ";
            }
        }
        std::cout << "\n";
    } while (std::next_permutation(select.begin(), select.end())); // Generate next combination
}
```

แบบฝึกหัด

- ให้นักเรียนเขียนโปรแกรมต่อไปนี้ด้วย Exhaustive search
  - ปัญหา 5-digit (5-digit.cpp/.c)
  - ปัญหา Sorted subsets (sorted-subsets.cpp/.c)
  - ปัญหาการนั่งในโรงภาพยนตร์ (movies-sitting.cpp/.c)
  - ปัญหา Subset sum (subset-sum.cpp/.c)
- เวลา 30 นาที
- ส่งภายใน 18/03/2568

Backtracking and Branch-and-Bound

Backtracking and Branch-and-Bound

- Backtracking (BT) and Branch-and-Bound (BB) เป็นขั้นตอนวิธีที่ช่วยแก้ Combinatorial Problems ที่ยาก ที่มีขนาดใหญ่บางปัญหา (Some large instances of problems)
- ต่างจาก Exhaustive search โดยที่ BT และ BB สร้างค่อย ๆ สร้าง Candidate solution บางส่วน
  - ถ้า ค่าส่วนที่เหลือไม่มี Potential values ขององค์ประกอบส่วนอื่นที่เหลือ ก็ไม่จำเป็นต้อง Generate ต่อไป
- ทั้ง BT และ BB นั้นเกี่ยวข้องกับการสร้าง State-space tree โดยที่แต่ละ Node จะหมายถึงทางเลือกเฉพาะ (Specific choices) สำหรับองค์ประกอบของ Solutions
- ทั้งสองวิธีจะหยุดค้นหาในทางเลือกนั้นทันทีเมื่อรับประกันได้ว่าไม่เจอ Solution แล้วแน่นอน

Backtracking and Branch-and-Bound (ต่อ)

- BT และ BB นั้นต่างกันที่ธรรมชาติของปัญหาที่มันจะ
  - BB นั้นแก้ปัญหาได้เฉพาะ Optimization problems เนื่องจากมันคือการคำนวณค่า Bound บนค่าที่เป็นไปได้ของ Objective function ของปัญหา
  - BT นั้นมักจะแก้ปัญหาที่เป็น Non-optimization problems
- BT และ BB ยังต่างกันที่ลำดับการ Generate nodes ใน State-space tree อีกด้วย
  - BT มักใช้ DFS
  - BB มักใช้ Best-first Search

Backtracking

Backtracking

Exhaustive Search การค้นหาโดยการ Generate ทุก ๆ Candidate solutions และเลือกอันที่มีคุณสมบัติที่ต้องการ (มากสุด, น้อยสุด, …)

- Backtracking search มีความฉลาดเหนือกว่า Exhaustive search โดยมันจะพิจารณา Partially constructed solution ว่ามันสามารถพัฒนาต่อไปได้ไหมโดยไม่ละเมิดข้อจำกัด (Constraints)
  - ถ้าไม่ได้มันจะหยุดการค้นหาใน Branch นั้น ๆ และทำการ Backtrack กับขึ้นมาที่ก่อนหน้าที่มันจะไปเจอทางที่ละเมิดดังกล่าวและเลือกไปในทางอื่นแทน
- Root คือ Initial states ก่อน Search
  - Node แรก คือทางเลือกแรกของ Partially constructed solution
  - Node ต่อมา คือทางเลือกต่อ ๆ มา ของ Partially constructed solution
- โดยเราจะบอกว่า Node ใน State-space tree นั้น **Promising** ถ้ามันเป็นส่วนหนึ่งของ Partially constructed solution ถ้าไม่เช่นนั้นเราเรียกว่า **Nonpromising**

Backtracking

- โดย Leaves คือ Complete solution หรือ Nonpromising dead-end ที่พบ
- State-space tree นี้ส่วนใหญ่เราสร้างแบบ **Depth-first search**
  - ถ้า Node ปัจจุบันเป็น **Promising** แล้ว Node นี้จะถูกเพิ่มเป็นทางเลือกต่อไป และ Algorithm จะเคลื่อนลงไปที่ทางเลือกดังกล่าวนี้
  - ถ้า Node ปัจจุบันเป็น Nonpromising แล้ว Algorithm จะ Backtrack กลับไปที่ Parent และเลือกทางเลือกอื่น ถ้าไม่มีทางเลือกแล้วให้ ขึ้นไปอีกหนึ่งเลเวล
  - เมื่อ Algorithm พบ Complete solution มันจะหยุดหรือค้นหาต่อไปสำหรับ Solution อื่น ๆ

Backtracking Algorithms

- โดยทั่วไป Output ของ Backtracking algorithm มักจะอยู่ในรูปของ n-tuple $(x_1, \ldots, x_n)$ โดยที่แต่ละ Coordinate x_i คือองค์ประกอบของ Ordered set S_i
- ทุก Solution tuples จะมี Length ที่เท่ากัน (N-Queens, Hamiltonian circuit) หรือไม่เท่ากันก็ได้ (Subset-sum)
- BB จะสร้าง State-space tree ที่ Node จะหมายถึง Partially constructed tuples
- ถ้า $(x_1, \ldots, x_i)$ ยังไม่ใช่ Solution มันจะหาทางเลือกอื่น S_{i+1} ที่สอดคล้องกับ $(x_1, \ldots, x_i)$ และเพิ่มลงไปใน Tuple ในตำแหน่ง (i+1)st
  - ถ้าไม่มีทางเลือกใดที่เป็นไปได้ใน S_{i+1} มันจะ Backtrack กลับไปหาค่าต่อไปของ x_i

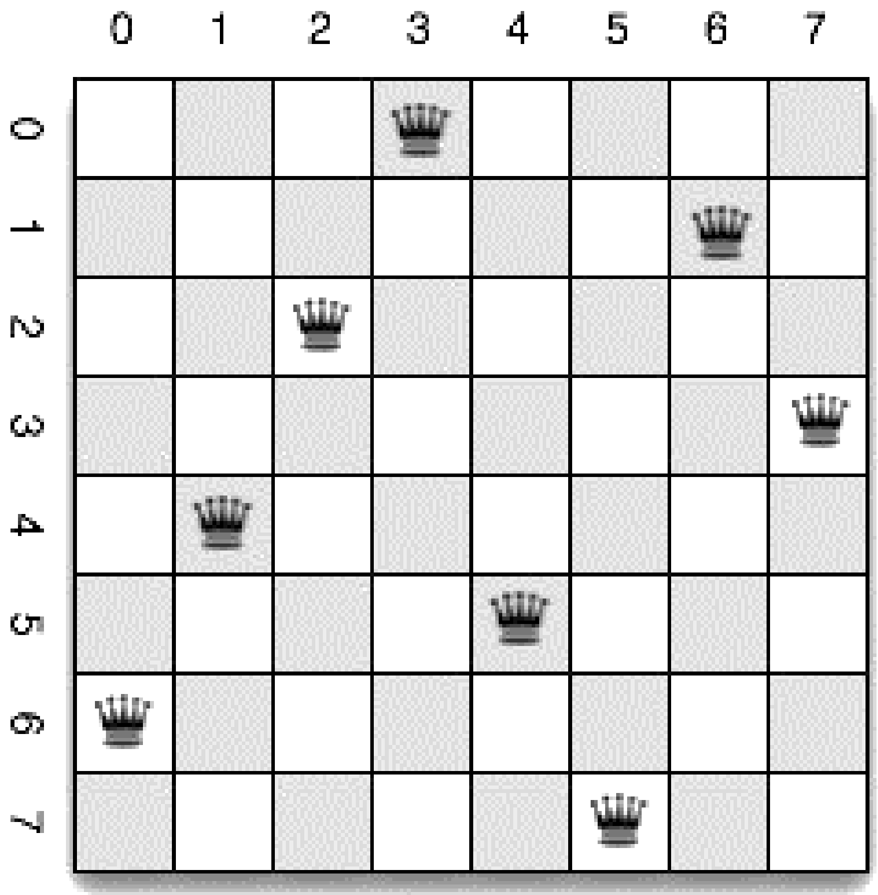
ALGORITHM *Backtrack*$(X[1..i])$

//Gives a template of a generic backtracking algorithm
//Input: $X[1..i]$ specifies first i promising components of a solution
//Output: All the tuples representing the problem's solutions
if $X[1..i]$ is a solution **write** $X[1..i]$
else //see Problem 9 in this section's exercises
  for each element $x \in S_{i+1}$ consistent with $X[1..i]$ and the constraints **do**
    $X[i+1] \leftarrow x$
    Backtrack$(X[1..i+1])$

Backtracking Algorithms

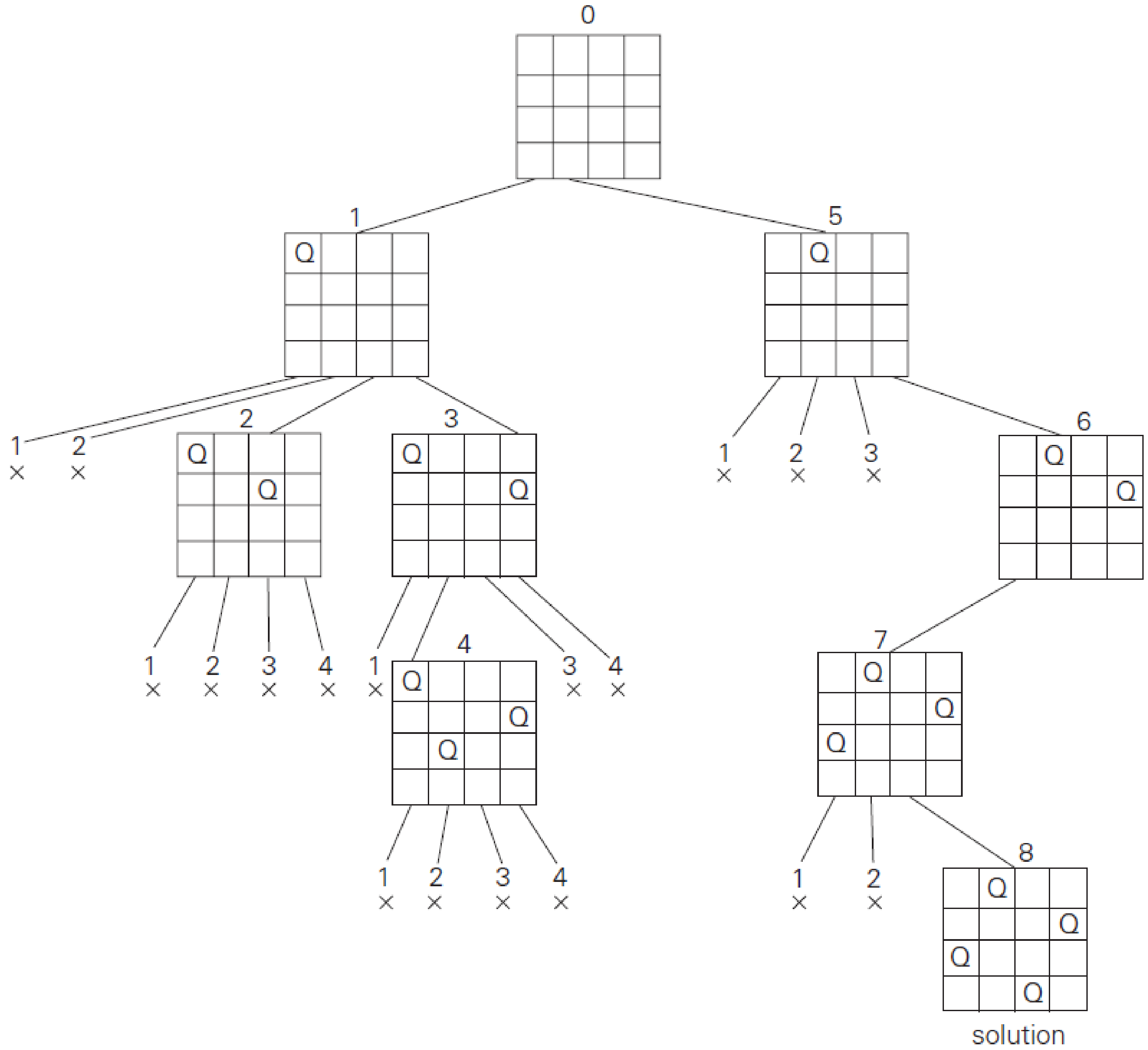
- ใน Worst case นั้น BT นั้นก็อาจจะต้อง Generate ทุก Possible candidates ในเวลา Exponential หรือ มากกว่า
- สำหรับ BT Algorithm
  - มักจะถูกใช้กับ **Combinatorial problems** ที่ยากและไม่มี Efficient algorithms ในการหา **Exact solution**
  - BT นั้นอาจจะมีประสิทธิภาพดีกว่า (Prune State-space ได้บางส่วน) Exhaustive search และมักจะแก้ปัญหาได้ในเวลาที่รับได้ (Acceptable amount of time)
  - แม้ BT จะไม่สามารถ Prune อะไรได้เลย แต่ก็ยังถือเป็นวิธีการหนึ่งที่ช่วยเรา generate ทุก ๆ Possible candidates ได้

N-Queens Problem



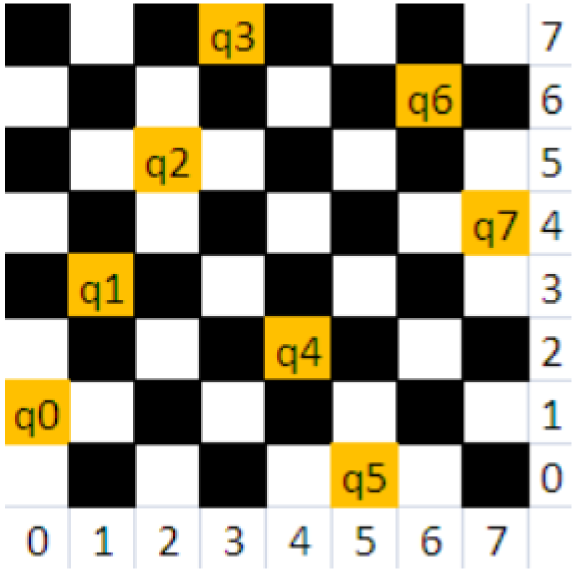
N-Queens Problem

- ปัญหา N-Queens คือการวาง Queen ลงใน Chessboard N x N โดย **<u>ห้าม</u>**มี Queen โจมตีกันได้ (Queens จะโจมตีกันเมื่อมันอยู่ใน แถว คอมลัมน์ หรือ แนวทแยงเดียวกัน)
- สำหรับ 8-Queen
  - ${}^{64}C_8$ = 4B (All combination)



N-Queens Problem

- Naïve I: All combinations ${}^{64}C_8 \approx 4B$
- Naïve II: Queen สามารถอยู่ได้ Column ละคนเท่านั้น การค้นหาเหลือ $8^8 \approx 17M$
- Faster **(Permutation)**: Queen สามารถอยู่ได้ Column หรือ Row ละคนเท่านั้น ดังนั้นถ้าเราแทนปัญหาด้วย row = {1,3,5,7,2,0,6,4} ให้การ search ลดจาก 8^8 เหลือ $8! \approx 40K$ เท่านั้น (คือหา permutation ของ row = {0,1,2,3,4,5,6,7})
- Faster II **(Recursive)**: Queen 2 คนไม่สามารแชร์แนวทแยงได้
  - Queen A (i, j) และ Queen B (k, l)
  - ถ้า abs(i-k) == abs(j-l) แล้ว Queen โจมตีกัน



N-Queens Problem

```
#include <bits/stdc++.h>
using namespace std;

int row[8], a, b, lineCounter;                    // global variables

bool canPlace(int r, int c) {
  for (int prev = 0; prev < c; ++prev)            // check previous Queens
    if ((row[prev] == r) || (abs(row[prev]-r) == abs(prev-c)))
      return false;                               // infeasible
  return true;
}

void backtrack(int c) {
  if ((c == 8) && (row[b] == a)) {                // a candidate solution
    printf("%2d      %d", ++lineCounter, row[0]+1);
    for (int j = 1; j < 8; ++j) printf(" %d", row[j]+1);
    printf("\n");
    return;                                       // optional statement
  }
  for (int r = 0; r < 8; ++r) {                   // try all possible row
    if ((c == b) && (r != a)) continue;           // early pruning
    if (canPlace(r, c))                           // can place a Queen here?
      row[c] = r, backtrack(c+1);                 // put here and recurse
  }
}

int main() {
  int TC; scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &a, &b); --a; --b;             // to 0-based indexing
    memset(row, 0, sizeof row); lineCounter = 0;
    printf("SOLN       COLUMN\n");
    printf(" #      1 2 3 4 5 6 7 8\n\n");
    backtrack(0);                                 // sub 8! operations
    if (TC) printf("\n");
  }
  return 0;
}
```

UVa00750: 8 Queens Chess Problem

Find all possible arrangements of eight chess queens on an 8x8 chessboard so that no two queens threaten each other (i.e., no two queens share the same row, column, or diagonal) given the initial position of one queen

```cpp
bool solveNQUtil(int board[N][N], int col)
{
    // base case: If all queens are placed
    // then return true
    if (col >= N)
        return true;

    // Consider this column and try placing
    // this queen in all rows one by one
    for (int i = 0; i < N; i++) {

        // Check if the queen can be placed on
        // board[i][col]
        if (isSafe(board, i, col)) {

            // Place this queen in board[i][col]
            board[i][col] = 1;

            // recur to place rest of the queens
            if (solveNQUtil(board, col + 1))
                return true;

            // If placing queen in board[i][col]
            // doesn't lead to a solution, then
            // remove queen from board[i][col]
            board[i][col] = 0; // BACKTRACK
        }
    }

    // If the queen cannot be placed in any row in
    // this column col  then return false
    return false;
}
```

```cpp
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;

    // Check this row on left side
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;

    // Check upper diagonal on left side
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;

    // Check lower diagonal on left side
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}
```

https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/

```cpp
bool solveNQ()
{
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 } };

    if (solveNQUtil(board, 0) == false) {
        cout << "Solution does not exist";
        return false;
    }

    printSolution(board);
    return true;
}
```

```cpp
int main()
{
    solveNQ();
    return 0;
}
```

```cpp
#include <bits/stdc++.h>
#define N 4
using namespace std;
```

แบบฝึกหัด

- ให้เขียนโปรแกรมโดยใช้แนวคิด Backtracking เพื่อแก้ปัญหา
  - N-Queens (NQP-BT.cpp)
  - 20 นาที (ในห้องเรียน)
  - ส่งภายใน 18/03/2568

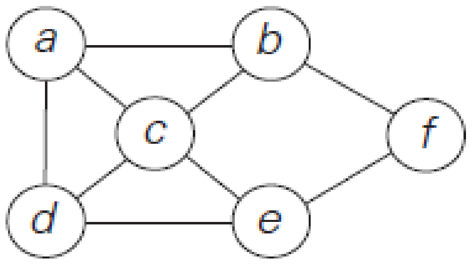
Hamiltonian Circuit Problem

Image source: https://www.dogonews.com/2012/9/30/scientists-learn-how-bumblebees-solve-complex-traveling-salesman-problem-so-effortlessly

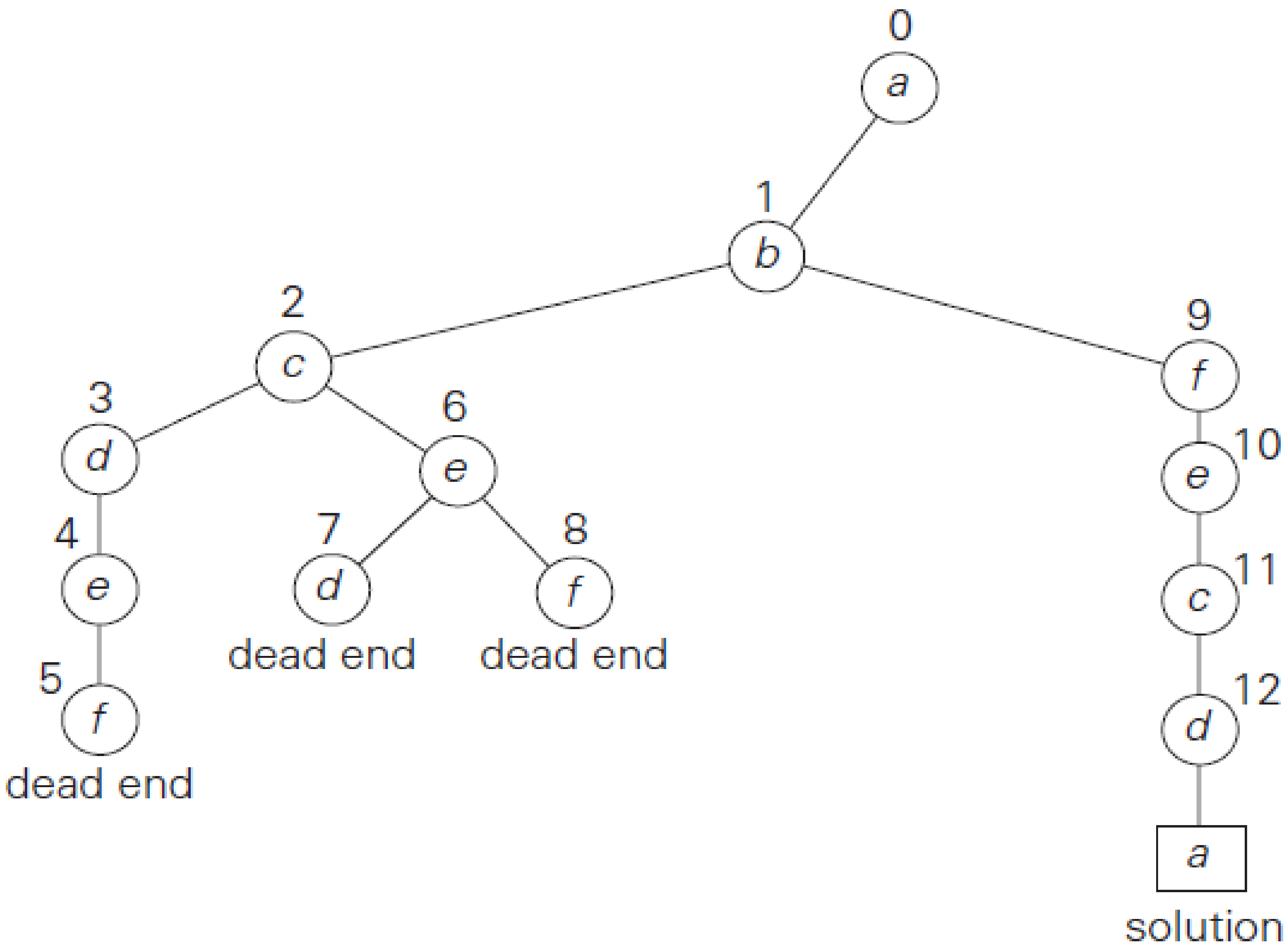
Hamiltonian Circuit Problem

- Hamiltonian Cycle or Circuit in a graph G is a cycle that visits every vertex of G exactly once and returns to the starting vertex.

Hamiltonian Circuit Problem



(a)



(b)

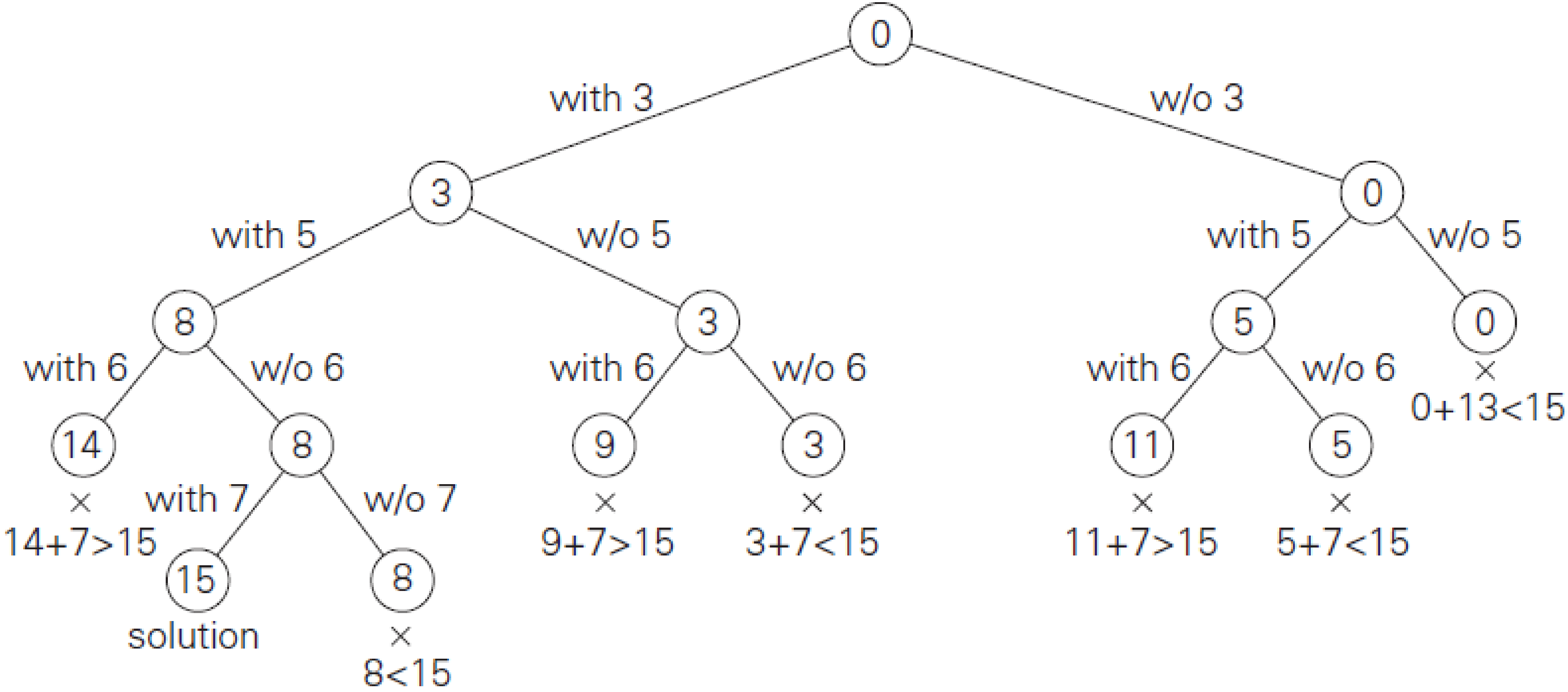
Subset-Sum Problem

Image source: https://www.dogonews.com/2012/9/30/scientists-learn-how-bumblebees-solve-complex-traveling-salesman-problem-so-effortlessly

Subset-Sum Problem

- กำหนดให้เซต $A = \{a_1, \dots, a_n\}$ ที่มีจำนวนสมาชิกทั้งหมด n ตัว ปัญหา Subset-sum คือการหา Subset ของเซต A สำหรับจำนวนเต็มบวก n จำนวน โดยผลรวมนั้นเท่ากับ d
- เช่น $A = \{1, 2,5,6,8\}$ และ $d = 9$ จะพบว่ามี 2 Solutions คือ {1,2,5,6,8} และ {1,8}
- บาง Instance ของปัญหานี้ไม่มีคำตอบ
- เพื่อแก้ปัญหานี้เราอาจจะเรียงลำดับจากน้อยไปมาก $a_1 < a_2 < \dots < a_n$

Subset-Sum Problem



แบบฝึกหัด

- เขียนโปรแกรมโดยใช้แนวคิด **Backtracking** เพื่อแก้ปัญหา
  - Subset-sum (SSP-BT.cpp)
  - ให้ใช้ตัวอย่างในไสลด์เป็นข้อมูลนำเข้าและส่งออก
  - 20 นาที (ในห้องเรียน)
  - ส่งภายใน 18/03/2568

(General)

Backtracking Algorithms

Image source: https://www.dogonews.com/2012/9/30/scientists-learn-how-bumblebees-solve-complex-traveling-salesman-problem-so-effortlessly

Backtracking Algorithms

- โดยทั่วไป Output ของ Backtracking algorithm มักจะอยู่ในรูปของ n-tuple $(x_1, \dots, x_n)$ โดยที่แต่ละ Coordinate x_i คือองค์ประกอบของ Ordered set S_i
- ทุก Solution tuples จะมี Length ที่เท่ากัน (N-Queens, Hamiltonian circuit) หรือไม่เท่ากันก็ได้ (Subset-sum)
- BB จะสร้าง State-space tree ที่ Node จะหมายถึง Partially constructed tuples
- ถ้า $(x_1, \dots, x_i)$ ยังไม่ใช่ Solution มันจะหาทางเลือกอื่น S_{i+1} ที่สอดคล้องกับ $(x_1, \dots, x_i)$ และเพิ่มลงไปใน Tuple ในตำแหน่ง (i+1)st
  - ถ้าไม่มีทางเลือกใดที่เป็นไปได้ใน S_{i+1} มันจะ Backtrack กลับไปหาค่าต่อไปของ x_i

ALGORITHM *Backtrack*$(X[1..i])$

//Gives a template of a generic backtracking algorithm
//Input: $X[1..i]$ specifies first i promising components of a solution
//Output: All the tuples representing the problem's solutions
if $X[1..i]$ is a solution **write** $X[1..i]$
else //see Problem 9 in this section's exercises
  for each element $x \in S_{i+1}$ consistent with $X[1..i]$ and the constraints **do**
    $X[i+1] \leftarrow x$
    Backtrack$(X[1..i+1])$

Backtracking Algorithms

- ใน Worst case นั้น BT นั้นก็อาจจะต้อง Generate ทุก Possible candidates ในเวลา Exponential หรือ มากกว่า
- สำหรับ BT Algorithm
  - มักจะถูกใช้กับ **Combinatorial problems** ที่ยากและไม่มี Efficient algorithms ในการหา **Exact solution**
  - BT นั้นอาจจะมีประสิทธิภาพดีกว่า (Prune State-space ได้บางส่วน) Exhaustive search และมักจะแก้ปัญหาได้ในเวลาที่รับได้ (Acceptable amount of time)
  - แม้ BT จะไม่สามารถ Prune อะไรได้เลย แต่ก็ยังถือเป็นวิธีการหนึ่งที่ช่วยเรา generate ทุก ๆ Possible candidates ได้

Branch-and-Bound

Branch-and-Bound

- แนวคิดของ BT คือการตัด (Prune) Branches ที่ไม่นำไปสู่ Solution
- BB นั้นเราจะต่อยอดแนวคิดของ BT เพื่อประยุกต์กับ Optimization problem
  - Optimization problem: ปัญหาที่หาค่าต่ำที่สุดหรือสูงที่สุดของ Objective function
  - Feasible solutions: เป็น Complete assignments ที่ไม่ขัดกับ Constraints ใด
  - Optimal solutions: เป็น Feasible solutions ที่ให้ค่าจาก Objective function ที่ดีที่สุด
- ถ้าเปรียบเทียบกับ BT แล้ว BB มีสิ่งต่อไปนี้
  - วิธีที่จะสามารถหาค่า Bound (Lower bound สำหรับ Minimization problem และ Upper bound สำหรับ Maximization problem) สำหรับ Best value ของ Objective function สำหรับทุก Solution ที่สามารถได้มาจากการเพิ่ม Component ไปยัง Partially constructed solution
  - ค่าที่ดีที่สุดของ Solution ที่ดีที่สุดที่เคยพบเจอมาทั้งหมด
- **<u>ถ้าค่าของ Bound ไม่ดีกว่า Best value ที่เคยพบเจอมา กล่าวคือ ไม่น้อยกว่าสำหรับ Minimization algorithm และไม่มากกว่าสำหรับ Maximization algorithm แล้ว Node ดังกล่าวจะถือว่า Nonpromising และถูก Pruned ทิ้งไป</u>**

Branch-and-Bound

- เราจะ Terminate ออกจาก Search path สำหรับ Branch ใน State-space tree ปัจจุบันเมื่อ
  - ค่า Bound ของ Node ไม่ดีกว่า Best solution แน่นอน
  - ละเมิด Problem constraints

Assignment Problem

Assignment Problem

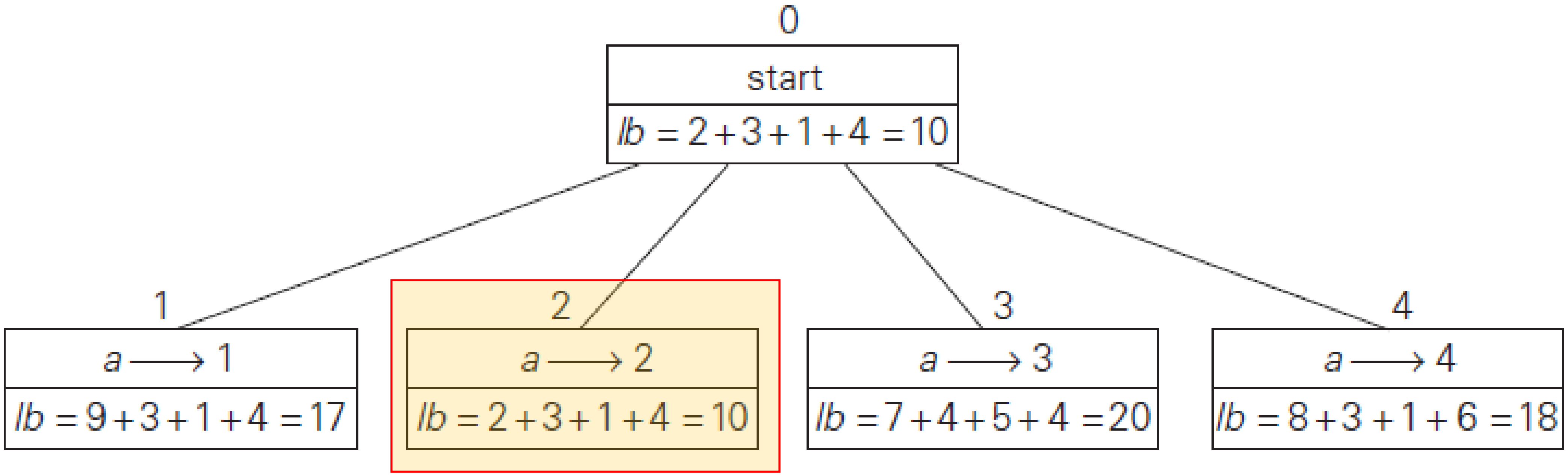
$$C = \begin{array}{c} \\ \\ \\ \\ \\ \end{array}\begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \\ \end{array}$$

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \begin{array}{l} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{array}$$

- ในปัญหานี้: **Cost ของทุก Solutions ไม่มีทางน้อยไปกว่าผลรวมของค่าที่น้อยที่สุดของแต่ละแถวบวกกัน**
  - เช่น 2 + 3 + 1 + 4 = 10 (Lower bound)
    (Note: ค่านี้ไม่ใช่ Legitimate selection – ค่า 3 และ 1 มาจากColumn เดียวกัน)
  - เมื่อเราเลือก 9 จากแถวแรกเราจะพบว่า Lower bound จะขยับเป็น 9 + 3 + 1 + 4 = 17
- เรื่องที่สำคัญอีกเรื่องหนึ่งคือ “ลำดับการที่ Node ถูก Generated”
  - เราจะพยายาม Generate **MOST** promising nodes จาก Nonterminating leaves บนต้นไม้ปัจจุบัน
  - คำถามคือ => เราจะรู้ได้อย่างไรว่า Node ใดคือ **MOST** promising node ?
    - คำตอบ => ให้เราดูจาก Lower bound ของแต่ละ Node (ไม่การันตีว่าจะเป็นส่วนหนึ่งของ Optimal)
    - วิธีการแบบนี้เรียกว่า **Best-first branch-and-bound**

Assignment Problem

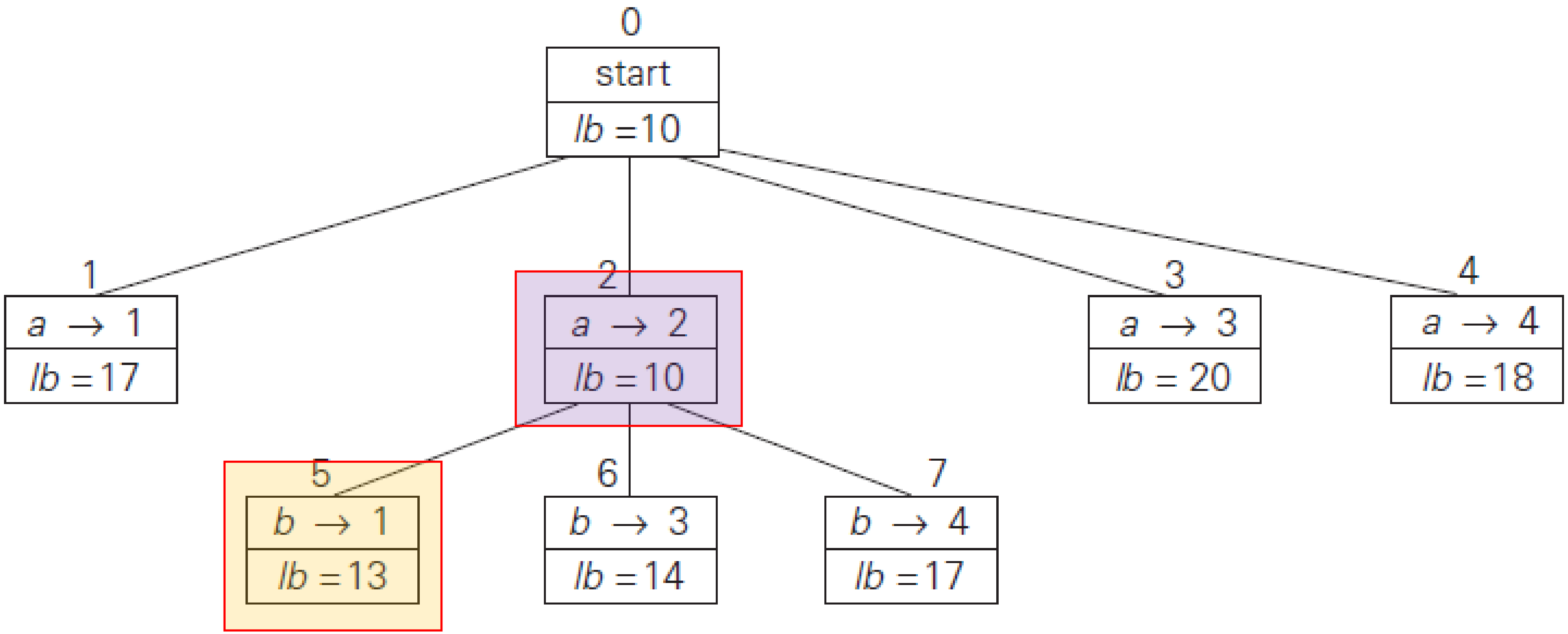
$$
\begin{array}{ccccc}
 & \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \\
C = & \left[\begin{array}{cccc} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{array}\right] & & &
\end{array}
\begin{array}{l} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{array}
$$



Assignment Problem

$$C = \begin{array}{c} \\ \\ \\ \\ \\ \end{array}\begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \\ \end{array}$$

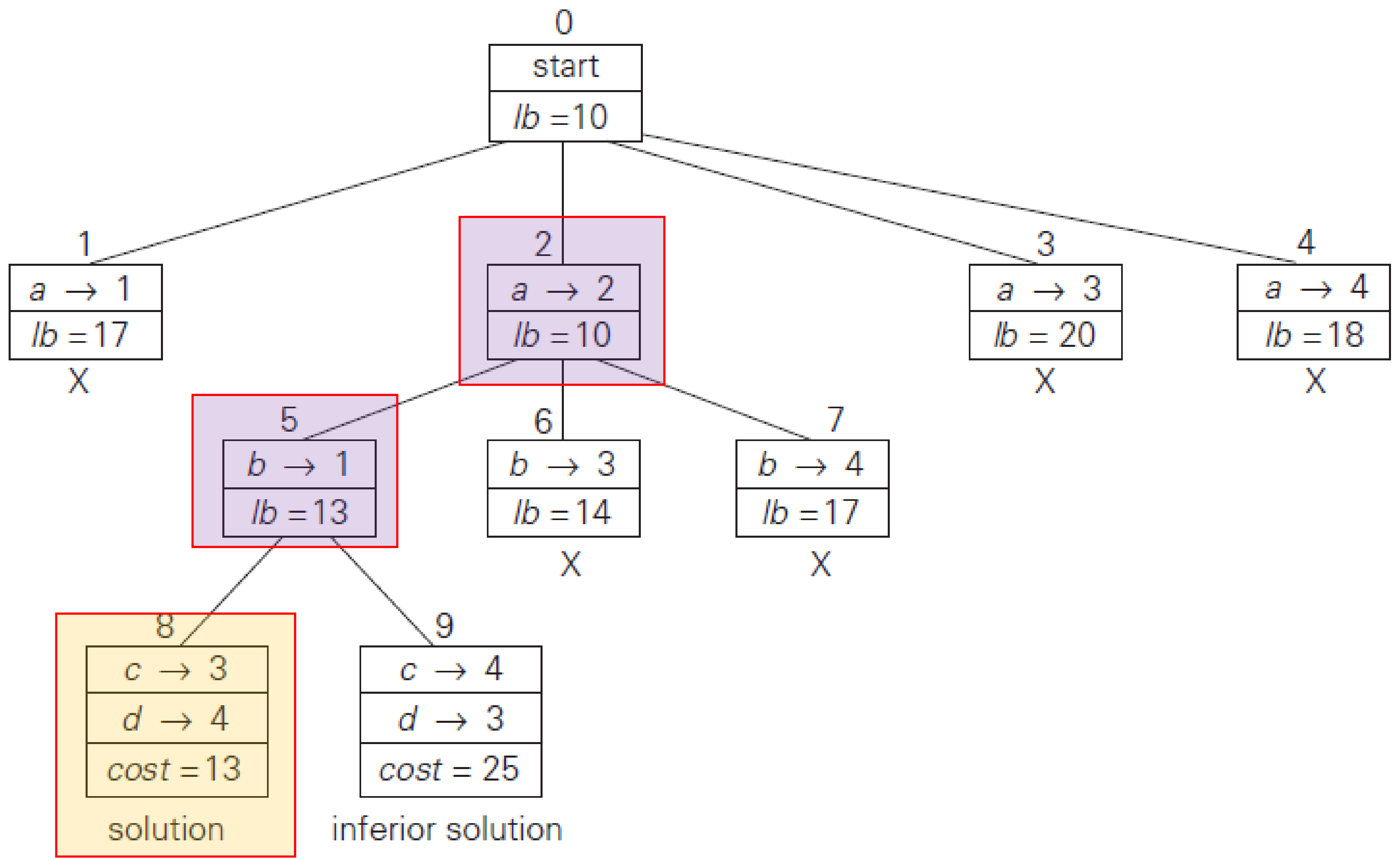
$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \begin{array}{l} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{array}$$



Assignment Problem

$$C = \begin{array}{c} \\ \\ \end{array}\begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{array}$$

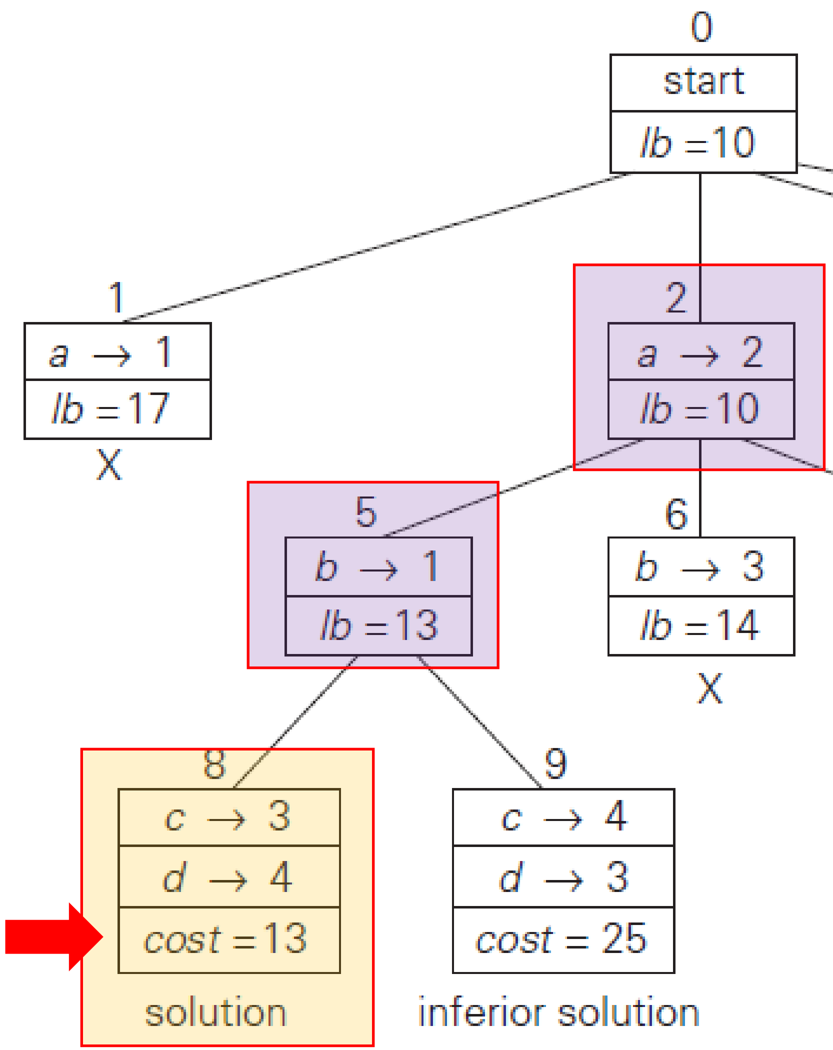
$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \begin{array}{l} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{array}$$



Assignment Problem

$$C = \begin{array}{c} \\ \\ \\ \\ \end{array}\begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \\ \end{array}$$

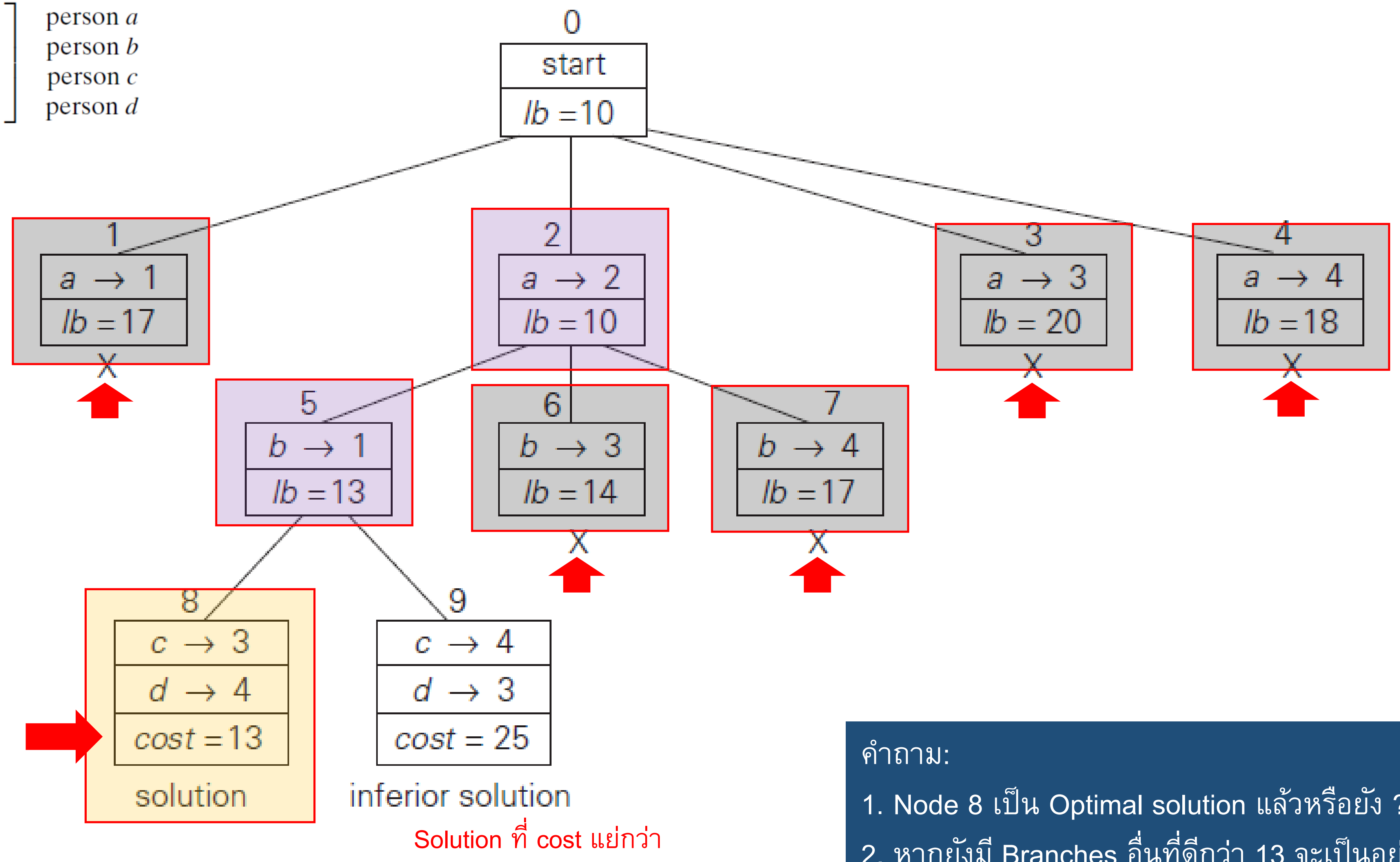
$$C = \begin{array}{cccc|l} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} & \\ \hline 9 & 2 & 7 & 8 & \text{person } a \\ 6 & 4 & 3 & 7 & \text{person } b \\ 5 & 8 & 1 & 8 & \text{person } c \\ 7 & 6 & 9 & 4 & \text{person } d \end{array}$$



Assignment Problem

เนื่องจาก Branch อื่น ๆ ถูก Pruned ไปหมด เนื่องจากค่า Lower bound ของ Branches เหล่านั้นแย่กว่าของ Best solution (lb = 13) ที่เจอ

$$
\begin{array}{cc}
 & \begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{array} \\
C = & \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}
\end{array}
\begin{array}{l} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{array}
$$



คำถาม:

1. Node 8 เป็น Optimal solution แล้วหรือยัง ? ทำไม ?
2. หากยังมี Branches อื่นที่ดีกว่า 13 จะเป็นอย่างไร ?

Hungarian Method

- https://byjus.com/maths/hungarian-method/

Knapsack Problem

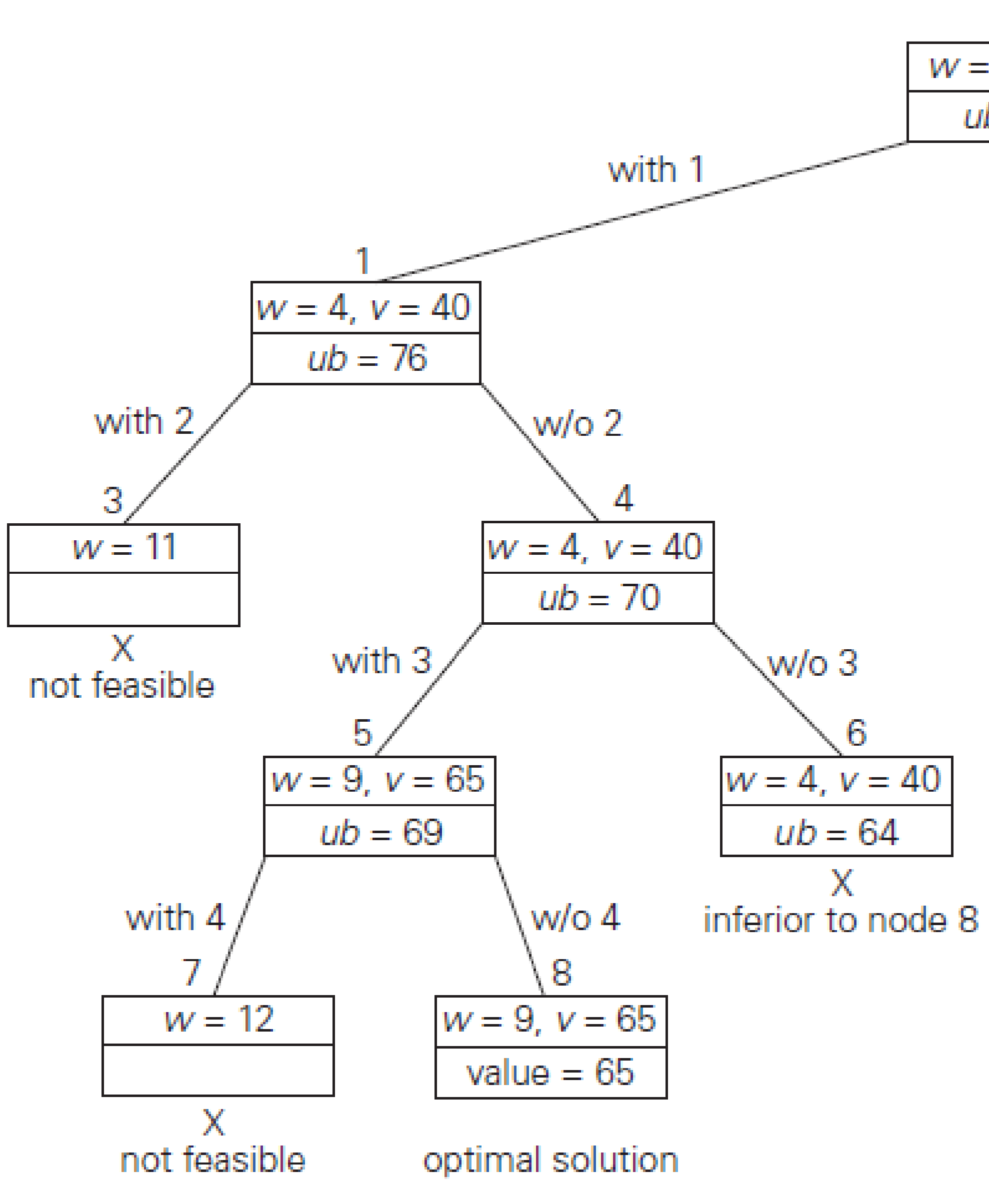
Knapsack Problem

- (ทบทวน) มีของจำนวน n ชิ้น โดยแต่ละชิ้นมีน้ำหนัก (Weight) คือ $w_1, \ldots, w_n$ และมูลค่า (Value) $v_1, \ldots, v_n$ และถุง (Knapsack) ที่มีความจุ W ให้หา Subset ของที่สามารถบรรจุลงในถุงได้โดยที่มูลค่ารวมใน Subset นั้นมีมูลค่ามากที่สุด
- สำหรับ BB
  - ให้เราเรียงลำดับตาม Value-to-weight ratios (มากไปหาน้อย) $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$
  - ในเลเวลที่ i-th ใน Search tree จะเป็นการแทนทุก Subsets ของ n items ที่รวม/ไม่รวม i items แรกที่เรียงลำดับ ($\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$)
  - ในแต่ละ Node จะบันทึก Total weight และ Total values รวมถึงค่า upper bound ของ Value
  - ค่า Upper bound (ub) = $v + (W - w)(v_{i+1}/w_{i+1})$

Knapsack Problem

item	weight	value	value/weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

The knapsack's capacity W is 10.



Upper bound ของ Root = 100

เนื่องจากยังไม่มีใครถูกเลือกดังนั้น w = 0, v = 0

ไอเท็มต่อไปคือ 40/4 = 10

ub = 0 + 10*10 = 100

Upper bound (ub) = $v + (W - w)(v_{i+1}/w_{i+1})$

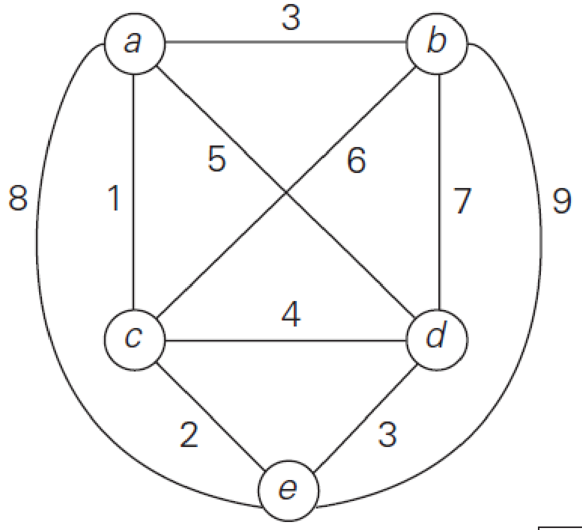
Traveling Salesman Problem

Traveling Salesman Problem

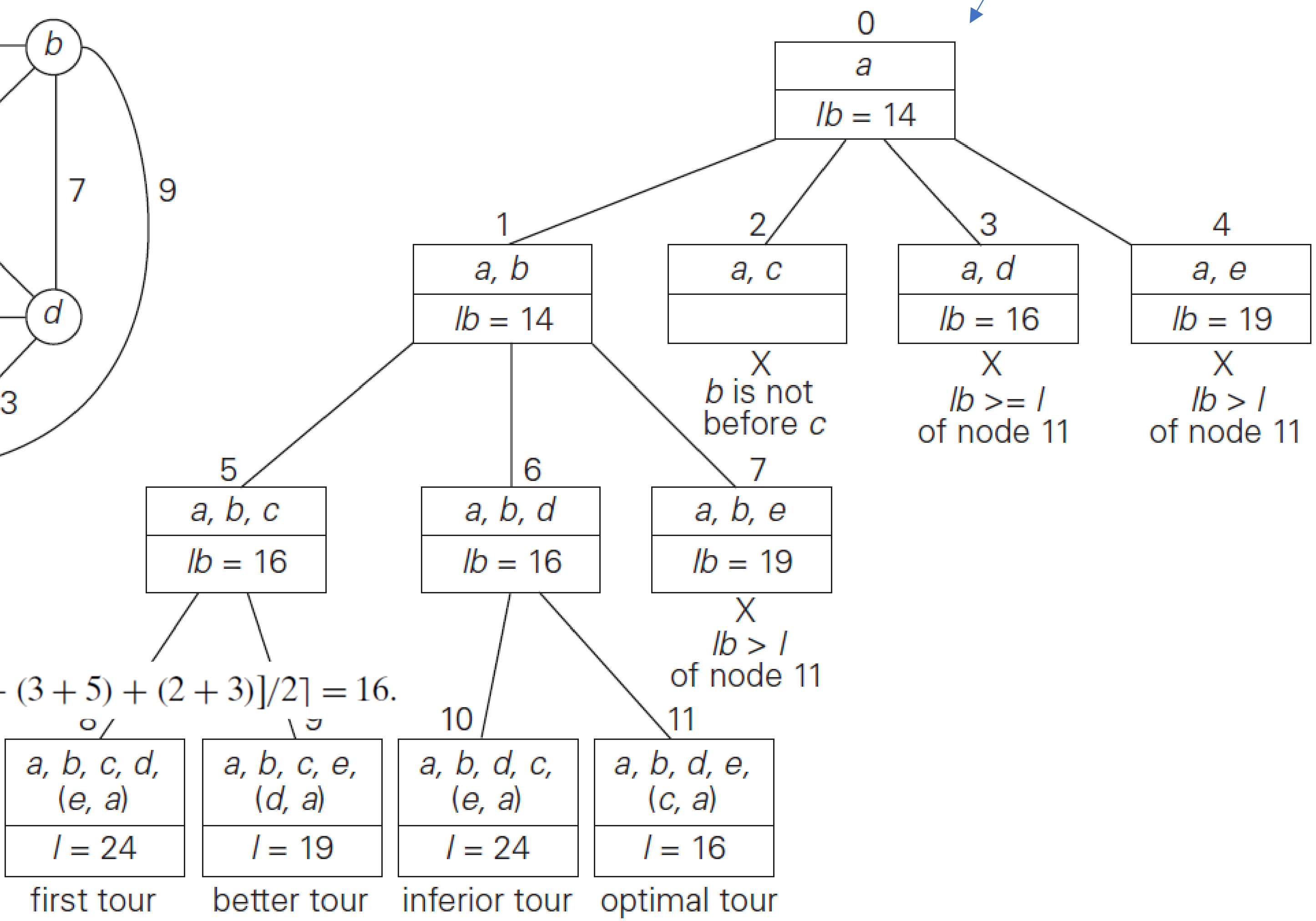
- **[1] Lower bound** คำนวณได้ Edge ที่มีค่าน้อยที่สุด (Element ที่น้อยที่สุดใน Intercity distance matrix) และคูณกับจำนวนเมืองทั้งหมด
- **[2] Lower bound** ก็สามารถคำนวณได้จาก สำหรับแต่ละเมือง $i, 1 \leq i \leq n$ หาผลรวม S_i ของ Distances จาก City i ไปยัง 2 เมืองที่ใกล้เมือง i ที่สุด คำนวณผลรวม S จาก n จำนวณนี้ $lb = \frac{\lceil s_1 + \cdots + s_n \rceil}{2} = \frac{\lceil s \rceil}{2}$

Traveling Salesman Problem

$lb = \lceil[(1+3)+(3+6)+(1+2)+(3+4)+(2+3)]/2\rceil = 14.$



(a)



$\lceil[(1+5)+(3+6)+(1+2)+(3+5)+(2+3)]/2\rceil = 16.$

Interesting Questions

- UVa 10285 - Longest Run on a Snowboard
  - https://onlinejudge.org/index.php?option=onlinejudge&Itemid=8&page=show_problem&problem=1226
- UVa 10350 - Liftless EME
  - https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1291

Tips

- Filtering versus Generating
- Prune Infeasible/Inferior Search Space Early
- Utilize Symmetries
- Pre-Computation a.k.a. Pre-Calculation
- Try Solving the Problem Backwards
- Data Compression

References

- Introduction to the Design and Analysis of Algorithms (3^{rd} Edition)
- Competitive Programming 3 by Steven Halim (Author)