HUST

ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.

FUNDAMENTALS OF OPTIMIZATION

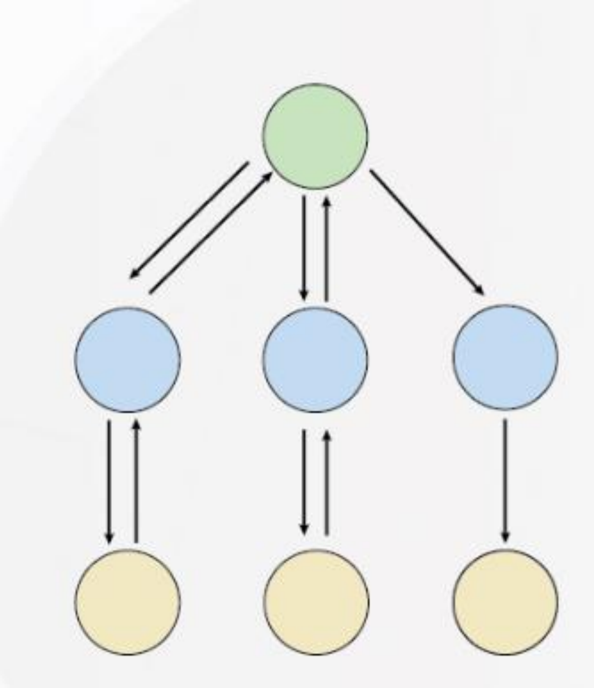
Branch-and-bound algorithm

Outline

- **Backtracking algorithm**
  - **Introduction to backtracking algorithm**
  - Backtracking algorithm for generation tasks
  - Backtracking algorithm for solving the N-Queen problem
  - Backtracking algorithm for solving the TSP
- Branch-and-bound algorithm
  - Introduction to branch-and-bound algorithm
  - Branch-and-bound algorithm for solving the TSP
  - Exercises:
    - CBUS
    - Count the number of feasible solutions for a linear equation

Backtracking Algorithm

- Backtracking is a problem-solving algorithmic technique that involves finding a solution incrementally by trying **different options** and **undoing** them if they lead to a **dead end**. It is commonly used in situations where you need to explore multiple possibilities to solve a problem, like searching for a path in a maze or solving puzzles like Sudoku. When a dead end is reached, the algorithm backtracks to the previous decision point and explores a different path until a solution is found or all possibilities have been exhausted.



- Backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.

A simple implementation of backtracking algorithm using a recursive technique

```
TRY(k)
  Begin
    Foreach v in A_k
      if check(v,k) /* Check for feasibility of assigning v to x_k  */
        Begin
          x_k = v;
          [Update some data structures]
          if(k = n) save a feasible solution;
          else TRY(k+1);
          [Recovery some data structures]
        End
  End
Main()
Begin
  TRY(1);
End
```

Simple recursive technique for backtracking algorithm

- **Candidate**: A candidate is a potential choice or element that can be added to the current solution.
- **Solution**: The solution is a valid and complete configuration that satisfies all problem constraints.
- **Partial Solution**: A partial solution is an intermediate or incomplete configuration being constructed during the backtracking process.
- **Decision Space**: The decision space is the set of all possible candidates or choices at each decision point.
- **Decision Point**: A decision point is a specific step in the algorithm where a candidate is chosen and added to the partial solution.

```
TRY(k)
  Begin
    Foreach v in A_k
      if check(v,k)
        Begin
          x_k = v;
          [Update some data structures]
          if(k = n) save a feasible solution;
          else TRY(k+1);
          [Recovery some data structures]
        End
  End
Main()
Begin
  TRY(1);
End
```

Simple recursive technique for backtracking algorithm

- **Feasible Solution**: A feasible solution is a partial or complete solution that adheres to all constraints.
- **Dead End**: A dead end occurs when a partial solution cannot be extended without violating constraints.
- **Backtrack**: Backtracking involves undoing previous decisions and returning to a prior decision point.
- **Search Space**: The search space includes all possible combinations of candidates and choices.
- **Optimal Solution**: In optimization problems, the optimal solution is the best possible solution.

```
TRY(k)
  Begin
    Foreach v in A_k
      if check(v,k)
        Begin
          x_k = v;
          [Update some data structures]
          if(k = n) save a feasible solution;
          else TRY(k+1);
          [Recovery some data structures]
        End
  End
Main()
Begin
  TRY(1);
End
```

Outline

- **Backtracking algorithm**
  - Introduction to backtracking algorithm
  - **Backtracking algorithm for generation tasks**
  - Backtracking algorithm for solving the N-Queen problem
  - Backtracking algorithm for solving the TSP
- Branch-and-bound algorithm
  - Introduction to branch-and-bound algorithm
  - Branch-and-bound algorithm for solving the TSP
  - Exercises:
    - CBUS
    - Count the number of feasible solutions for a linear equation

Generate binary strings and permutations of a set

```python
n = 3
x = [-1] * n
def Try(k):
    if k == n:
        print(x)
    else:
        for i in range(2):
            x[k] = i
            Try(k+1)

Try(0)
```

```
[0, 0, 0]
[0, 0, 1]
[0, 1, 0]
[0, 1, 1]
[1, 0, 0]
[1, 0, 1]
[1, 1, 0]
[1, 1, 1]
```

```python
n = 3
x = [-1] * n
visited = [False] * (n)

def Try(k):
    if k == n:
        print(x)
        return

    for i in range(0, n):
        if visited[i] == True:
            continue

        x[k] = i
        visited[i] = True
        Try(k+1)
        visited[i] = False

Try(0)
```

```
[0, 1, 2]
[0, 2, 1]
[1, 0, 2]
[1, 2, 0]
[2, 0, 1]
[2, 1, 0]
```

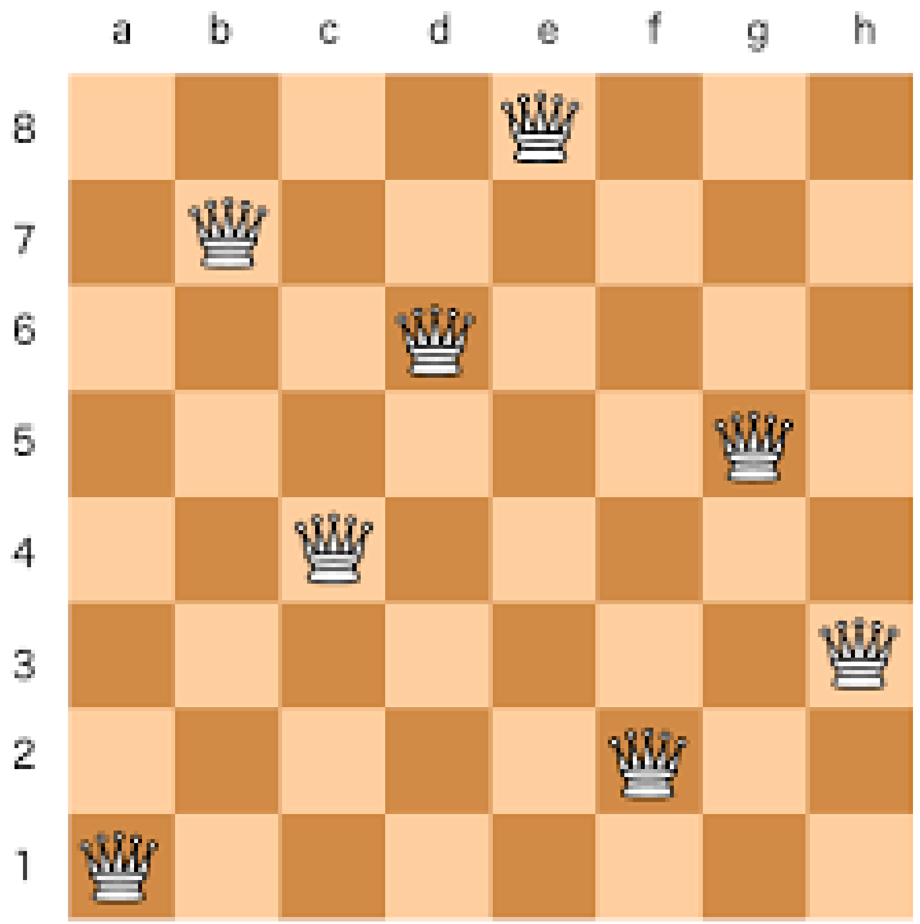
Outline

- **Backtracking algorithm**
  - Introduction to backtracking algorithm
  - Backtracking algorithm for generation tasks
  - **Backtracking algorithm for solving the N-Queen problem**
  - Backtracking algorithm for solving the TSP
- Branch-and-bound algorithm
  - Introduction to branch-and-bound algorithm
  - Branch-and-bound algorithm for solving the TSP
  - Exercises:
    - CBUS
    - Count the number of feasible solutions for a linear equation

Modelling N-Queen

Problem N-Queen, $CSP = (X, D, C)$

- Variables: $X = \{x_1, \ldots, x_n\}$, in which x_i is the row of the queen in column $i, \forall\, i \in \{1, \ldots, n\}$
- Domains: $D_i = D(x_i) = \{1, \ldots, n\}, \forall i \in \{1, \ldots, n\}$
- Constraints: For all pair $(i, j),\ 1 \le i < j \le n$:
  - $x_i \neq x_j$
  - $x_i + i \neq x_j + j$
  - $x_i - i \neq x_j - j$



Backtracking for N-Queen

```python
import sys

n = 4

x = [-1] * n
visit = [False] * n

def Try(k):
    if k == n:
        print(x)
        #sys.exit()
    else:
        for v in range(n):
            if visit[v] == False:
                feasible = True
                for i in range(k):
                    if x[i] + i == v + k or x[i] - i == v - k:
                        feasible = False
                        break
                if feasible == True:
                    x[k] = v
                    visit[v] = True
                    Try(k+1)
                    visit[v] = False

Try(0)
```

Outline

- **Backtracking algorithm**
  - Introduction to backtracking algorithm
  - Backtracking algorithm for generation tasks
  - Backtracking algorithm for solving the N-Queen problem
  - **Backtracking algorithm for solving the TSP**
- Branch-and-bound algorithm
  - Introduction to branch-and-bound algorithm
  - Branch-and-bound algorithm for solving the TSP
  - Exercises:
    - CBUS
    - Count the number of feasible solutions for a linear equation

Modelling Traveling Salesman Problem (TSP)

- **Input**:
  - n number of cities
  - $c(i, j)$ traveling cost from the city i to the city j
- **Variables**: Variable x_i with $i \in \{1, \ldots, n\}$ is the i^{th} city in the optimal tour
- **Domains**: $D(x_i) = \{1, \ldots, n\}$ for each $i \in \{1, \ldots, n\}$
- **Constraints**: The traveler visits each city exactly once: $x_i \neq x_j, \forall\, i, j \in \{1 \ \ldots, n\}, i \neq j$
- **Objective**: $Minimize\ c(x_n, x_1) + \sum_{i \in \{1,\ldots,n\}} c(x_i, x_{i+1})$

Backtracking algorithm for solving TSP

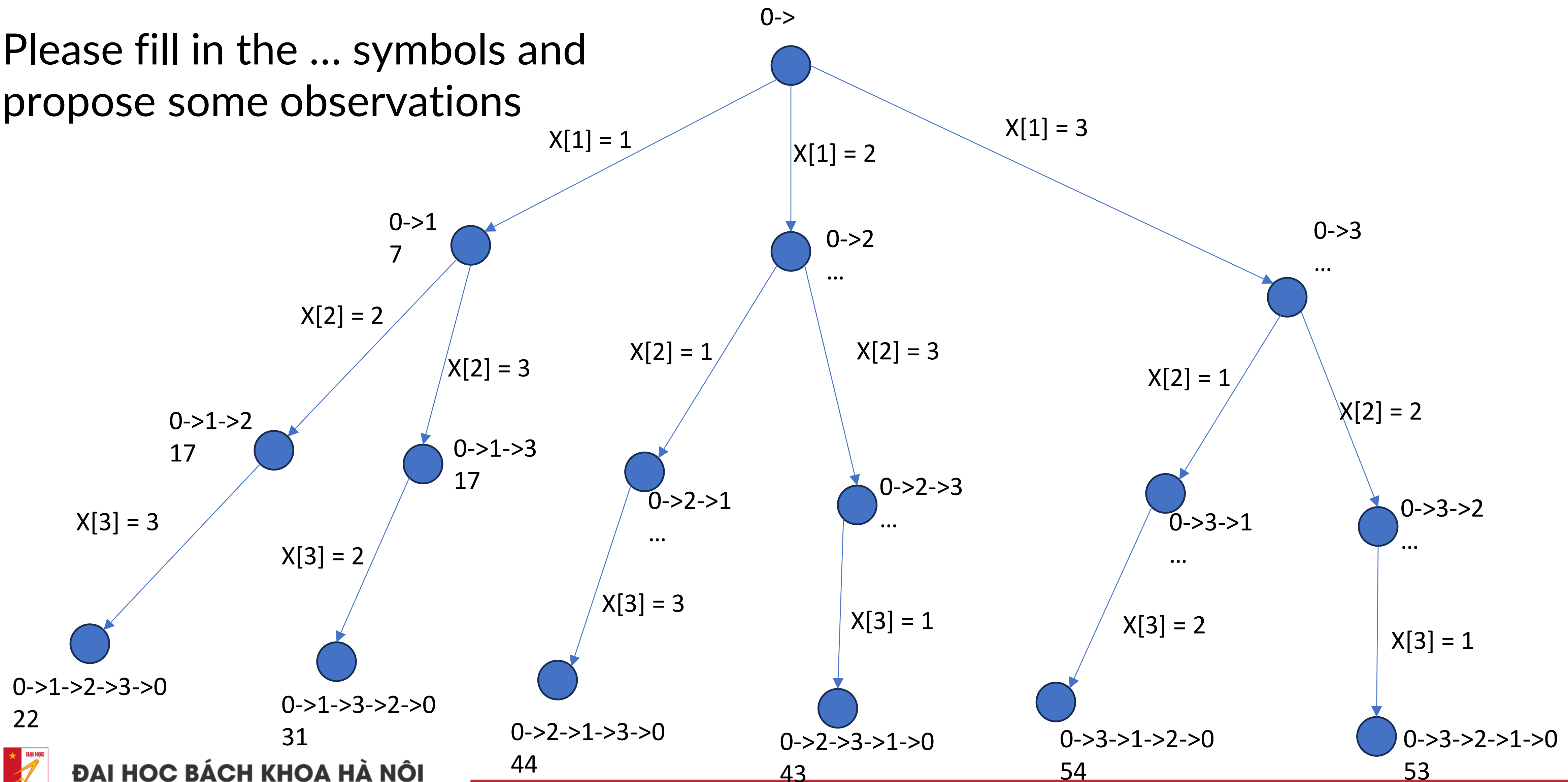
```python
# Input
n = 4
D = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

# Primary variables
x = [-1] * n

# Status variables
sol_val = 0
visit = [False] * n
ans_val = np.sum(D)
ans = [-1] * n
```

```python
# Recursive function
def Try(k):
    global x
    global sol_val
    global visit
    global ans_val
    global ans

    if k == n:
        #Update the status variable
        sol_val += D[x[n-1]][0]

        if sol_val < ans_val: #Update the incumbent
            ans_val = sol_val
            ans = x.copy()

        #Release the status variable
        sol_val -= D[x[n-1]][0]

        print(x)
    else:
        for v in range(n):
            #Check for feasible assignment
            if visit[v] == False:
                x[k] = v

                #Update the status variables
                sol_val += D[x[k-1]][x[k]]
                visit[v] = True

                Try(k+1)
                #Release the status variables
                sol_val -= D[x[k-1]][x[k]]
                visit[v] = False
```

```python
x[0] = 0
visit[0] = True
Try(1)

print("\nBest solution")
print(ans)
print(ans_val)
```

Search tree generated by the backtracking algorithm for solving the TSP

Please fill in the ... symbols and propose some observations



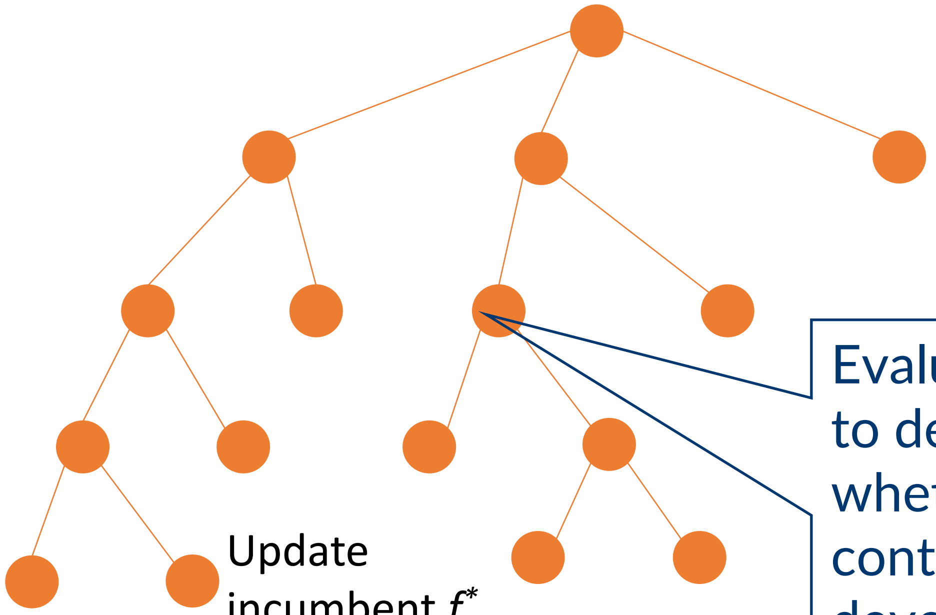
Outline

- Backtracking algorithm
  - Introduction to backtracking algorithm
  - Backtracking algorithm for generation tasks
  - Backtracking algorithm for solving the N-Queen problem
  - Backtracking algorithm for solving the TSP
- **Branch-and-bound algorithm**
  - **Introduction to branch-and-bound algorithm**
  - Branch-and-bound algorithm for solving the TSP
  - Exercises:
    - CBUS
    - Count the number of feasible solutions for a linear equation

Branch-and-bound algorithm

- The **Branch and Bound (B&B)** algorithm is a method used in **combinatorial optimization problems** to systematically search for the best solution. It works by dividing the problem into smaller subproblems, or branches, and then eliminating certain branches based on bounds on the optimal solution. This process continues until the best solution is found or all branches have been explored.
- **B&B** algorithm is commonly used in problems like the **Traveling Salesman Problem** and **job scheduling**.

A simple implementation of a B&B algorithm using a recursive technique



Evaluate bound to decide whether to continue developing the partial solution or not

Bounding function

```
TRY(k) {
  Foreach v in A_k
    if check(v,k) {
     x_k = v;
     [Update some data structures]
     if(k = n) {
          save a feasible solution;
        Update the incumbent f*;
     }else{
        if g(x1,…,x_k) < f*
          TRY(k+1);
     }
     [Recovery some data structures]
   }
}
Main(){
  f* = +∞;
  TRY(1);
}
```

Outline

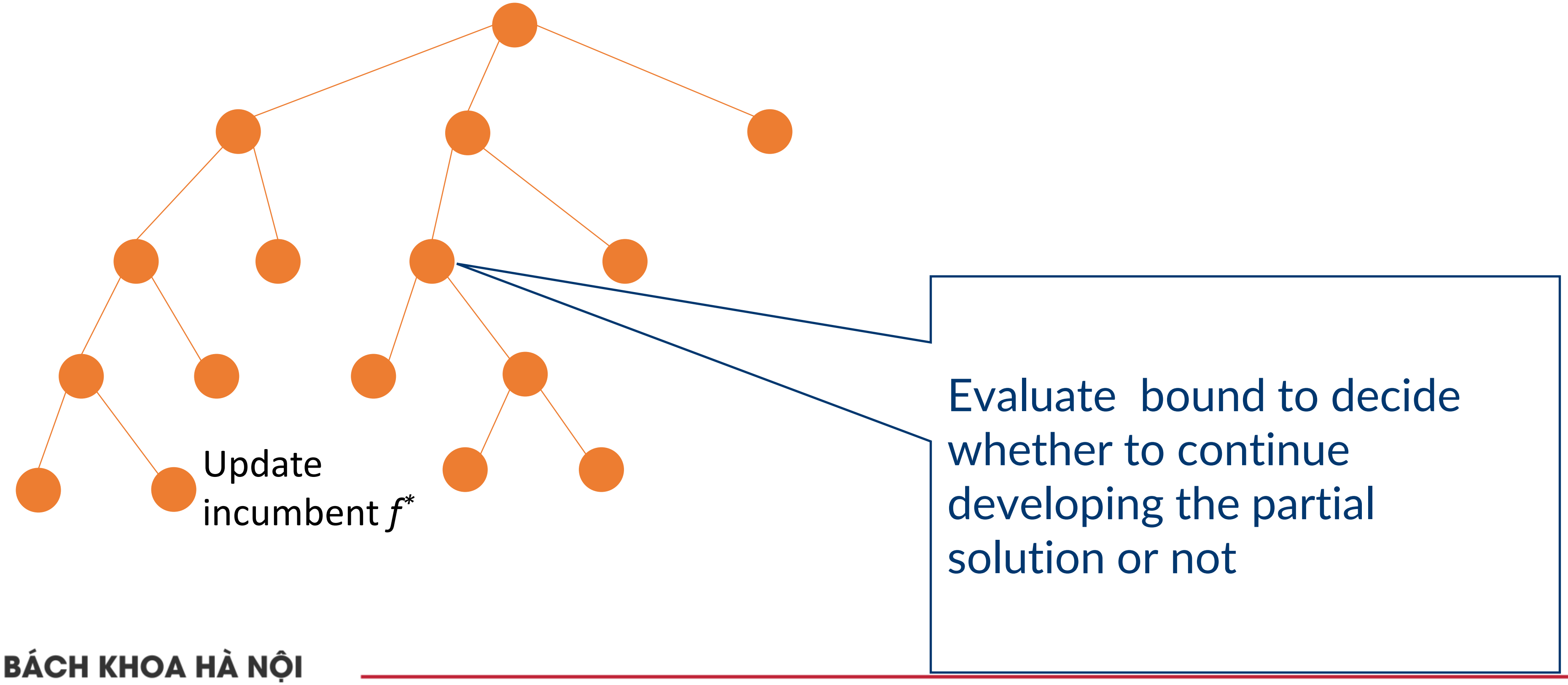
- Backtracking algorithm
  - Introduction to backtracking algorithm
  - Backtracking algorithm for generation tasks
  - Backtracking algorithm for solving the N-Queen problem
  - Backtracking algorithm for solving the TSP
- **Branch-and-bound algorithm**
  - Introduction to branch-and-bound algorithm
  - **Branch-and-bound algorithm for solving the TSP**
  - Exercises:
    - CBUS
    - Count the number of feasible solutions for a linear equation

Propose a B&B algorithm for solving the TSP

- **Model**
  - ***Inputs***:
    - n number of cities
    - $c(i, j)$ traveling cost from the city i to the city j
  - ***Variables***: Variable x_i with $i \in \{1, \dots, n\}$ is the i^{th} city in the optimal tour
  - ***Domains***: $D(x_i) = \{1, \dots, n\}$ for each $i \in \{1, \dots, n\}$
  - ***Constraints***: The traveler visits each city exactly once: $x_i \neq x_j, \forall\, i, j \in \{1 \ \dots, n\}, i \neq j$
  - ***Objective***: $Minimize\ c(x_n, x_1) + \sum_{i \in \{1,\dots,n\}} c(x_i, x_{i+1})$
- **Propose a B&B algorithm** for solving TSP using the above model
  - What is a bounding function for the B&B algorithm?

Propose a B&B algorithm for solving the TSP

- What is a proposed bounding function?
  - $Bounding = Cost\_Of\_The\_Path\_Already\_Travel + Number\_Of\_Remaining\_Cities * C_{min}$
  - C_{min} is the minimum cost between two cities



Propose a B&B algorithm for solving the TSP

```python
import numpy as np
#Input
n = 4
D = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]
minD = np.min(D)

# Primary variables
x = [-1] * n

# Status variables
sol_val = 0
visit = [False] * n
ans_val = np.sum(D)
ans = [-1] * n
```

```python
def Try(k):
    global x
    global sol_val
    global visit
    global ans_val
    global ans

    if k == n:
        print(x)
        #Update the status variable
        sol_val += D[x[n-1]][0]

        if sol_val < ans_val: #Update the incumbent
            ans_val = sol_val
            ans = x.copy()

        #Release the status variable
        sol_val -= D[x[n-1]][0]
    else:
        for v in range(n):
            if visit[v] == False:  #Check for feasible assignment
                x[k] = v

                #Update the status variable
                sol_val += D[x[k-1]][x[k]]
                visit[v] = True

                if sol_val + (n-k)*minD < ans_val:
                    Try(k+1)

                #Release the status variable
                sol_val -= D[x[k-1]][x[k]]
                visit[v] = False

x[0] = 0
visit[0] = True
Try(1)

print("\nBest solution")
print(ans)
print(ans_val)
```

Observations on the current B&B algorithm for solving the TSP

- If we have a good incumbent solution before starting the solving process, the search tree might be significantly smaller, or the total computational time could be sharply reduced.
- If we have an efficient bounding function, meaning its estimated value is as close as possible to the optimal solution, the search tree might be much smaller, or the total computational time could be significantly reduced.
- The assignment order of the variables is fixed in lexical order, x_1, x_2, ..., x_n, which might not be ideal in a general situation

A generic B&B algorithm

Problem:

- Input: Problem P (e.g., an optimization problem with a minimum objective function f and constraints)
- Output: Optimal solution S^* and its value $f(S^*)$

1. **Initialize**
   - $Best\_Solution\ (S^*) = None$
   - $Best\_Value = +\infty$
   - Queue $Q = \{Root\ Node\}$ (representing the entire problem P)
   - $Bound\_Root = ComputeBound(Root)$
2. **While Q is not empty**
   a) Select and remove a node N from Q
   b) If N is a feasible solution:
      - Compute $f(N)$ (Objective value of solution in N)
      - If $f(N)$ is better than $Best\_Value$, update $Best\_Solution = N$ and $Best\_Value = f(N)$
   c) Else:
      - $Bound(N) = ComputeBound(N)$
      - If $Bound(N)$ is better than $Best\_Value$,
        - Branch N into subproblems $\{N_1, N_2, \ldots, N_k\}$;
        - For each subproblem N_i, $Bound(N_i) = ComputeBound(N_i)$ if $Bound(N_i)$ is better than $Best\_Value$ add N_i into Q
      - Else: Prune N
3. **Return** $Best\_Solution\ (S^*)$ and $Best\_Value\ (f(S^*))$

A new version of B&B algorithm for solving the TSP

```python
import heapq
import math

class Node:
    def __init__(self, level, path, bound):
        self.level = level  # Current level in the search tree
        self.path = path    # Current path taken
        self.bound = bound  # Lower bound of this node

    def __lt__(self, other):
        return self.bound < other.bound

# Helper function to calculate the lower bound
# It estimates the cost of completing the tour from the current state
```

A new version of B&B algorithm for solving the TSP

```python
def calculate_bound(matrix, path):
    n = len(matrix)
    bound = 0

    # Mark visited cities
    visited = [False] * n
    for i in path:
        visited[i] = True

    # Add the cost of edges in the current path
    for i in range(1, len(path)):
        bound += matrix[path[i - 1]][path[i]]

    # Add minimum edge costs for unvisited cities
    for i in range(n):
        if not visited[i]:
            min_cost = math.inf
            for j in range(n):
                if not visited[j] and matrix[i][j] > 0:
                    min_cost = min(min_cost, matrix[i][j])
            if min_cost != math.inf:
                bound += min_cost

    return bound
```

A new version of B&B algorithm for solving the TSP

```python
# Branch-and-bound algorithm for TSP
def tsp_branch_and_bound(matrix):
    n = len(matrix)
    priority_queue = []

    # Start with the root node (city 0 as the starting point)
    root = Node(0, [0], calculate_bound(matrix, [0]))
    heapq.heappush(priority_queue, root)

    best_cost = math.inf
    best_path = None

    while priority_queue:
        # Get the node with the smallest bound
        current_node = heapq.heappop(priority_queue)

        # If this node's bound is worse than the current best cost, prune it
        if current_node.bound >= best_cost:
            continue
```

A new version of B&B algorithm for solving the TSP

```python
        # If all cities are visited, calculate the total cost of the tour
        if current_node.level == n - 1:
            # Complete the tour by returning to the start city
            last_city = current_node.path[-1]
            total_cost = calculate_bound(matrix, current_node.path) + matrix[last_city][0]

            # Update the best solution if found
            if total_cost < best_cost:
                best_cost = total_cost
                best_path = current_node.path + [0]

        else:
            # Expand the current node by visiting unvisited cities
            for next_city in range(n):
                if next_city not in current_node.path:
                    new_path = current_node.path + [next_city]
                    new_bound = calculate_bound(matrix, new_path)

                    # If the bound is promising, add the node to the queue
                    if new_bound < best_cost:
                        heapq.heappush(priority_queue, Node(current_node.level + 1, new_path, new_bound))

    return best_cost, best_path
```

A new version of B&B algorithm for solving the TSP

```python
# Example usage
distance_matrix = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

best_cost, best_path = tsp_branch_and_bound(distance_matrix)
print("Best Cost:", best_cost)
print("Best Path:", best_path)
```

```
Best Cost: 80
Best Path: [0, 1, 3, 2, 0]
```

Outline

- Backtracking algorithm
  - Introduction to backtracking algorithm
  - Backtracking algorithm for generation tasks
  - Backtracking algorithm for solving the N-Queen problem
  - Backtracking algorithm for solving the TSP
- **Branch-and-bound algorithm**
  - Introduction to branch-and-bound algorithm
  - Branch-and-bound algorithm for solving the TSP
  - **Exercises:**
    - CBUS
    - Count the number of feasible solutions for a linear equation

Exercises: CBUS

There are n passengers $1, 2, \ldots, n$. The passenger i want to travel from point i to point $i + n$ $(i = 1,2, \ldots, n)$. There is a bus located at point 0 and has k places for transporting the passengers (it means at any time, there are at most k passengers on the bus). You are given the distance matrix c in which $c(i, j)$ is the traveling distance from point i to point j $(i, j = 0,1, \ldots, 2n)$. Compute the shortest route for the bus, serving n passengers and coming back to point 0.

CBUS - Hint

- Apply the B&B technique
- Modelling a complete solution by a vector of $2n$ variables that represent a permutation of pick-up and drop-off points
- Status variables:
  - $Load$: An integer represents the number of passengers on the bus
  - $Visit$: A Boolean array to mark a point visited or not
- Function $Try(k)$ attempts to assign a value to $x[k]$. For a feasible value v for $x[k]$, performing the following:
  - Update the status variable, $Load = Load + 1$ if $v \leq n$ (pick-up point), and $Load = Load$ – 1 if $v > n$
  - Update the status variable, $Visit[v] = True$
  - If $k = 2n$, save a feasible solution, update the incumbent (or the best solution so far) if necessary; Otherwise, recursively call $Try(k + 1)$
- Apply the bounding function as the B&B algorithm for solving TSP

Exercises: Count the number of feasible solutions for a linear equation

Given a positive integer n and n positive integers $a_1, a_2, \dots, a_n$. Compute the number of positive integer solutions to the equation:

$$a_1X_1 + a_2X_2 + \ldots + a_nX_n = M$$

Count the number of feasible solutions for a linear equation - Hint

- Apply backtracking technique
- Modelling a complete solution by a vector of n variables $(x_1, x_2, \ldots, x_n)$
- Function $Try(k)$ attempts to assign a value v to x_k.
  - Feasible value v for x_k ranges from 1 to $\frac{M - \sum_{i=1}^{k-1} a_i x_i - \sum_{i=k+1}^{n} a_i}{a_k}$
  - Assign $x_k = v$
  - Update the status variable $f = f + a_k x_k$
  - If $k < n$, we call recursively $Try(k+1)$, Else Count a feasible solution

THANK YOU !

hust.edu.vn fb.com/dhbkhn