Design and Analysis of Algorithms (UE17CS251)

Unit V - Limitations of Algorithm Power and Techniques to cope with

Mr. Channa Bankapur
channabankapur@pes.edu

Ingredients of an introductory course in Algorithms

1. Design **strategies** for designing algorithms.
2. **Tools** for analyzing algorithms.
3. Identifying and coping with the **limitations** of algorithms.

Comparing asymptotic efficiency classes of algorithms:

- Unfair comparison: Sequential Search algorithm, whose worst-case time efficiency is O(n), is faster than Mergesort because of its worst-case time efficiency O(n log n).
- Fair comparison: Mergesort, whose worst-case time efficiency is O(n log n), is faster than Selection sort because its worst-case time efficiency is O(n^2).

Lower Bound: an estimate on a minimum amount of work needed to solve a given problem.

Examples:

- Minimum number of comparisons needed to sort an array of size n.
- Minimum number of multiplications needed to multiply two n-by-n matrices.

What could be the lowest possible worst-case time efficiency of an algorithm, which solves the sorting problem?
That’s the lower bound for the sorting problem.

For a given problem, there cannot be an algorithm with lesser worst-case efficiency than the lower bound.

Lower bound can be

- An exact count.
- An efficiency class (Ω)

Tight lower bound: there exists an algorithm with the same efficiency class as the lower bound. We can hope for a constant-factor improvement at best.
Eg: O(n logn) is the lower-bound for comparison-based sorting algorithms and we know O(n logn) sorting algorithms.

Methods for Establishing Lower Bounds

- Trivial lower bounds
- Information-theoretic arguments (decision trees)
- Adversary arguments
- Problem reduction

Trivial Lower Bounds: based on counting the number of items that must be processed in **input** and generated as **output**.

Examples:

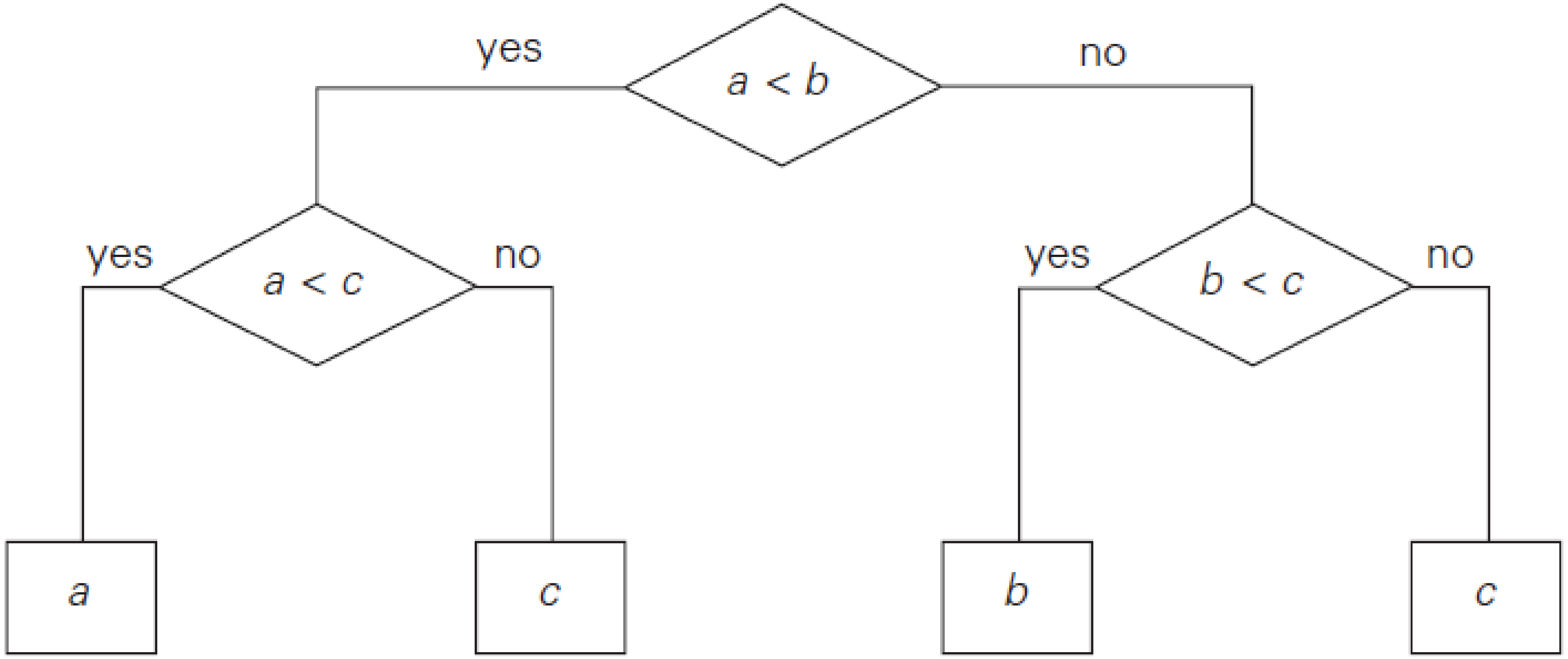
- **n** in searching an element in unordered array
- **n**, degree of the polynomial in Polynomial Evaluation
- **n** in Sorting algorithms
- **n** in Element Uniqueness problem
- $\mathbf{n^2}$ in finding the product of two n-by-n matrices

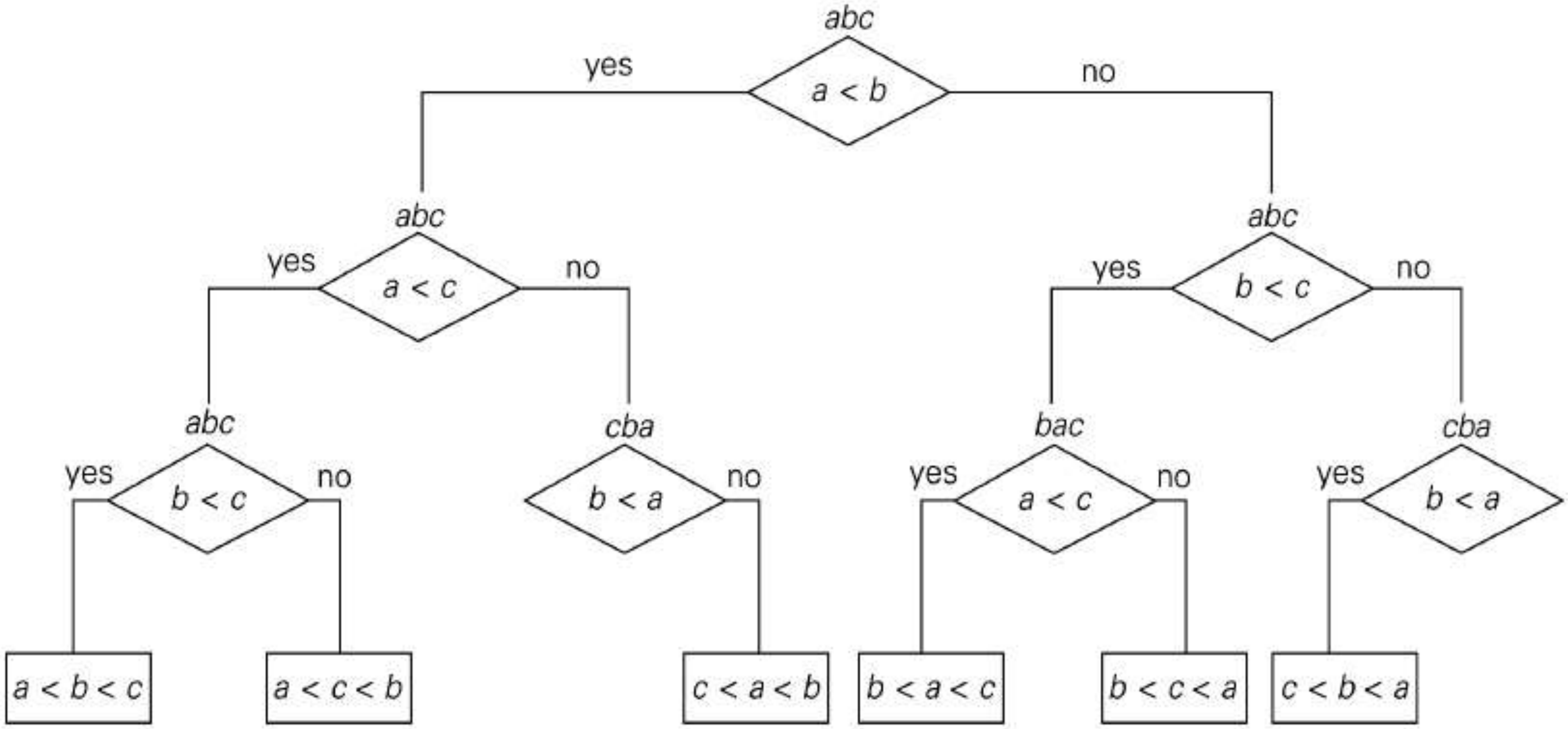
Remarks:

- Not an useful result in most cases.
- Be careful in deciding how many elements must be processed.

Decision Trees: Comparison-based algorithms can be studied with decision trees.

- Each leaf represents a possible outcome of the algorithm's run on some input of size n.
- The number of leaves must be at least as large as the number of possible outcomes.





Comparison-based sorting algorithms

- Sorting can be interpreted as finding a permutation of the list of n items which is in a desired order.
- The number of possible outcomes is n!

The algorithm’s work on a particular input of size n can be traced by a path from the root to a leaf in its decision tree, and the number of comparisons made by the algorithm on such a run is equal to the length of this path. Hence, the number of comparisons in the worst case is equal to the height of the algorithm’s decision tree.

A tree with a given number of leaves, which is dictated by the number of possible outcomes, has to be tall enough to have that many leaves. Specifically, it is not difficult to prove that for any binary tree with *l* leaves and height h,

$$h \geq \lceil \log_2 l \rceil$$

$$2^h \geq l$$

Decision-tree model

A decision tree can model the execution of any comparison sort:

- One tree for each input size n.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.

Decision Tree and Sorting Algorithms

- Number of leaves (outcomes) $\geq n!$
- Height of binary tree with $n!$ leaves $\geq \lceil \log_2 n! \rceil$
- Minimum number of comparisons in the worst case $\geq \lceil \log_2 n! \rceil$ for any comparison-based sorting algorithm

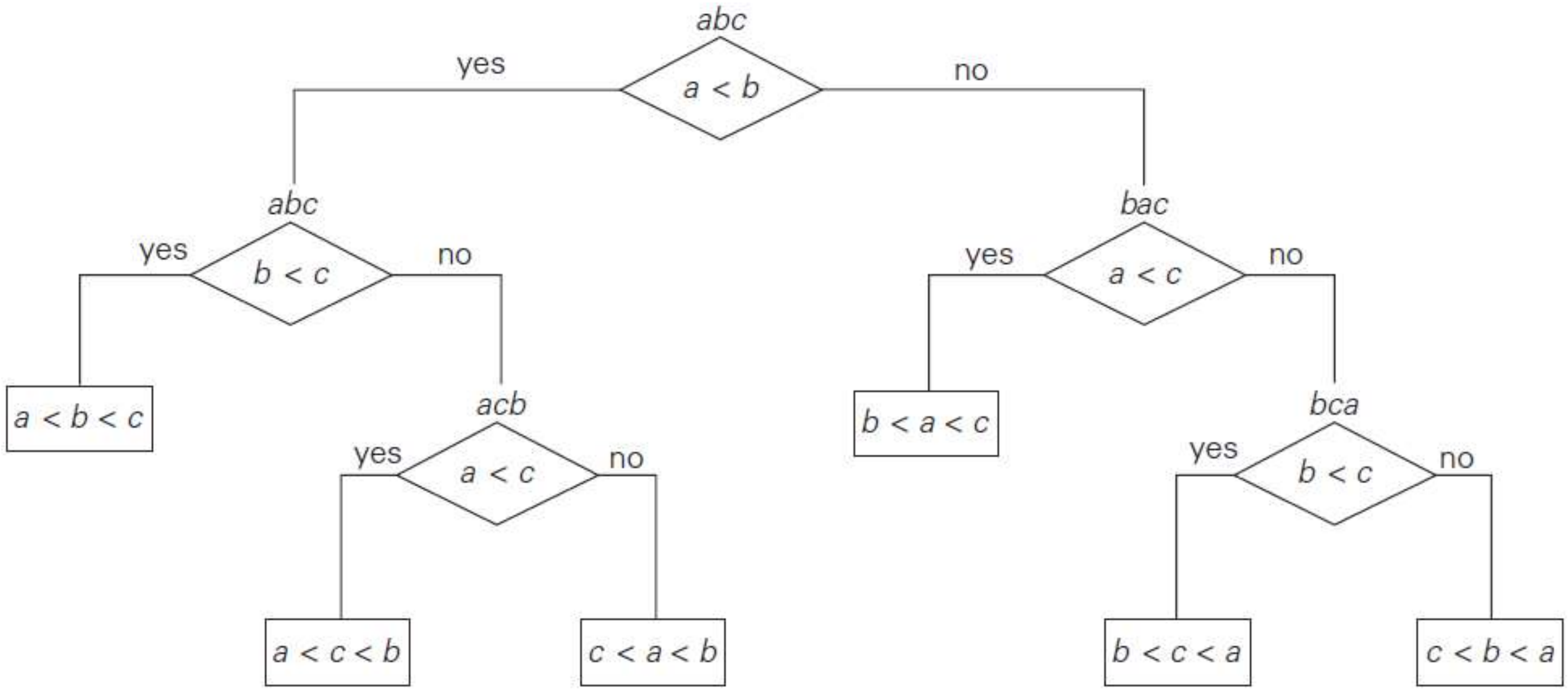
  $C_{worst}(n) \geq \lceil \log_2 n! \rceil$

  Using Stirling's formula for $n!$, we get

  $$\lceil \log_2 n! \rceil \approx \log_2 \sqrt{2\pi n}(n/e)^n$$
  $$= n \log_2 n - n \log_2 e + \frac{\log_2 n}{2} + \frac{\log_2 2\pi}{2} \approx n \log_2 n$$

- $\lceil \log_2 n! \rceil \approx n \log_2 n$

  about $n \log_2 n$ comparisons are necessary in the worst case to sort an arbitrary n-element list by any comparison-based sorting algorithm.
- This lower bound is tight (e.g. mergesort)

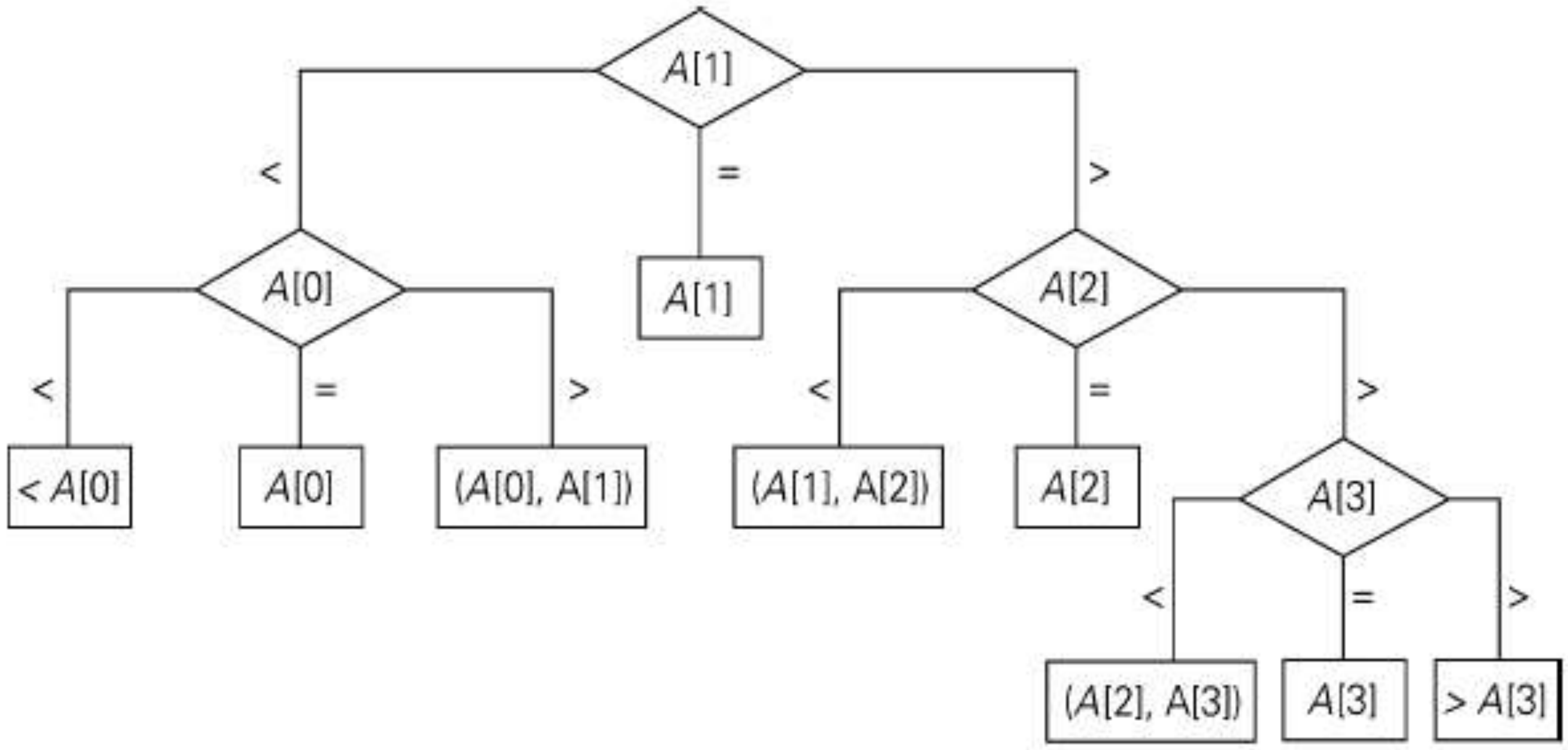


Decision tree for the three-element insertion sort

Average-case efficiency = Average depth of the leaves

= (2 + 3 + 3 + 2 + 3 + 3)/6 = 2.66

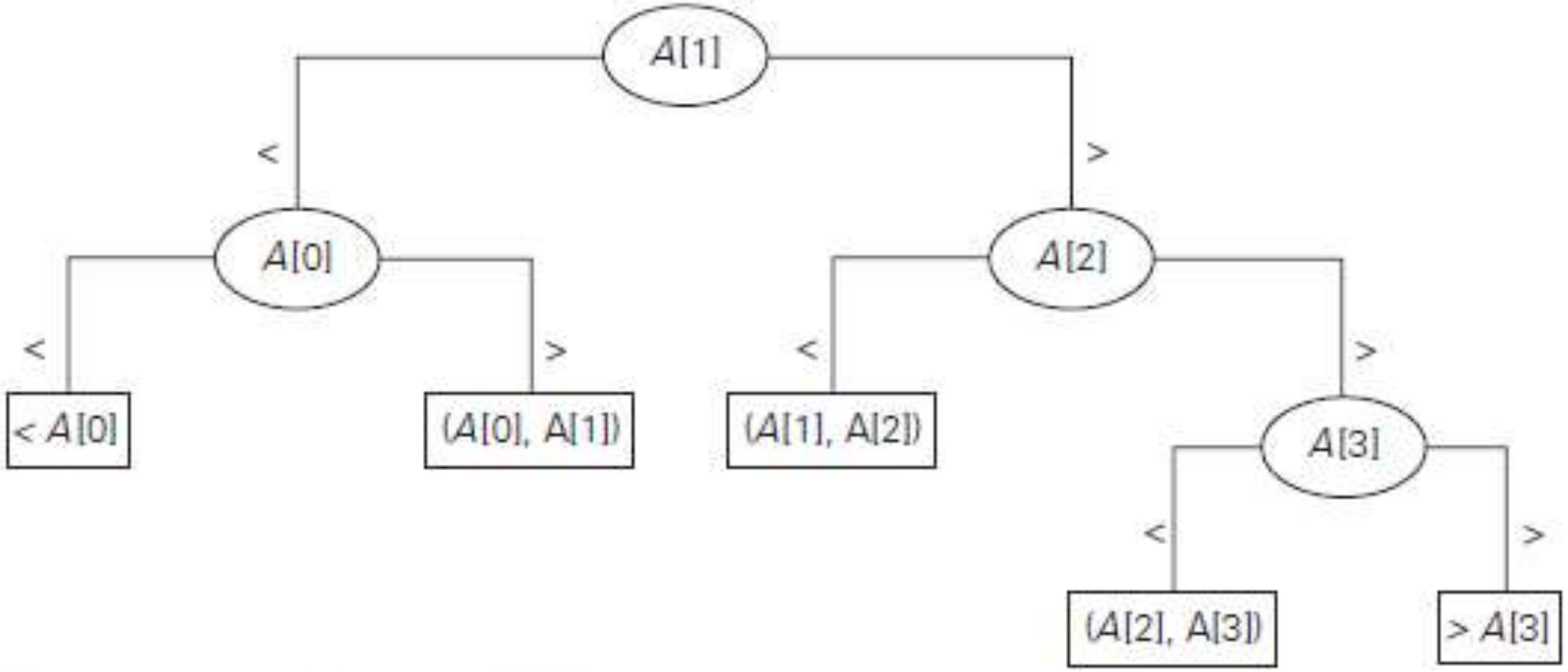
Decision Tree and Searching a Sorted Array



Decision Tree and Searching a Sorted Array

- Number of leaves (outcomes) = $n + n+1 = 2n+1$
- Height of ternary tree with $2n+1$ leaves $\geq \lceil \log_3 (2n+1) \rceil$
- This lower bound is NOT tight (the number of worst-case comparisons for binary search is $\lceil \log_2 (n+1) \rceil$, and $\lceil \log_3 (2n+1) \rceil \leq \lceil \log_2 (n+1) \rceil$)
- Can we find a better lower bound or find an algorithm with better efficiency than binary search?

Binary decision tree for binary search in a four-element array



$$C_{worst}(n) \geq \lceil \log_2(n+1) \rceil$$

Problem	**Lower bound**	**Tightness**
sorting	$\Omega(n \log n)$	yes
searching in a sorted array	$\Omega(\log n)$	yes
element uniqueness	$\Omega(n \log n)$	yes
n-digit integer multiplication	$\Omega(n)$	unknown
multiplication of n-by-n matrices	$\Omega(n^2)$	unknown

Adversary argument: a method of proving a lower bound by playing role of adversary that makes algorithm work the hardest by adjusting input.

Example: “Guessing” a number between 1 and n with yes/no questions

Adversary: Puts the number in a larger of the two subsets generated by last question

Lower Bounds by Problem Reduction

If problem P is at least as hard as problem Q, then a lower bound for P is the lower bound of Q. Hence, find problem Q with a known lower bound that can be reduced to problem P in question.

Example: P is finding Euclidean MST for n points in Cartesian plane and Q is element uniqueness problem, which is known to be in O(nlogn). Therefore, a lower bound for Euclidean MST is also O(nlogn).

P (E-MST) is at least as hard as Q (element uniqueness problem) because Q can be mapped to an instance of P by considering (x_i , 0) as coordinates of P where x_i 's are elements of Q.

P, NP and NP-complete Problems:

- **Undecidable** problems are unsolvable problems. They are those that cannot be solved at all by any algorithm. Eg: Halting Problem.
- There are decidable, but not every decision problem can be solved in polynomial time.
- We say that an algorithm solves a problem in polynomial time if its worst-case time efficiency belongs to O(p(n)) where p(n) is a polynomial of the problem’s input size n.
- Problems that can be solved in polynomial time are called **tractable**, and problems that cannot be solved in polynomial time are called **intractable**.

P, NP and NP-complete Problems:

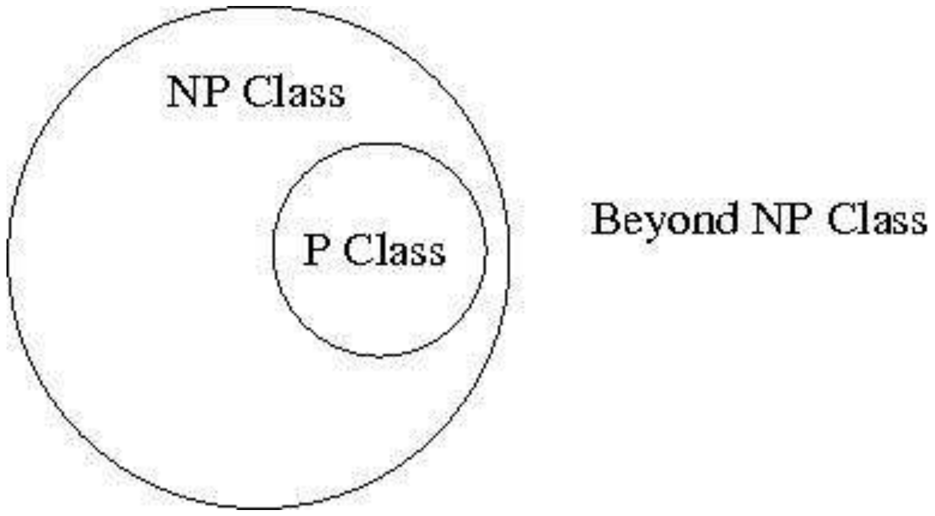
- We have a long list of problems, which are decidable, but it’s not yet proved if they are intractable or not.
  - Hamiltonian Circuit
  - Knapsack Problem
  - Graph Coloring
- **Class P** is a class of decision problems that can be solved in **polynomial** time by (deterministic) algorithms.

Hamiltonian Circuit:

- Determine whether a given graph has a circuit having all the vertices exactly once.
- We know an exponential-time algorithm, but no polynomial-time algorithm.
- Interestingly, determining whether there is an Eulerian circuit in a given graph has a polynomial-time algorithm.
- Though determining a Hamiltonian circuit doesn’t have a polynomial-time algorithm (yet!), validating whether the proposed solution actually solves the problem can be done in polynomial-time.
- Does a problem which can be verified in polynomial-time always has a polynomial-time algorithm to solve the problem in the first place?

Class NP is the class of decision problems, given a solution found by a nondeterministic polynomial-time algorithm, can be verified by a (deterministic) polynomial-time algorithm. This class of problems are called **nondeterministic polynomial (NP)**.

Eg: Finding Hamiltonian Circuit



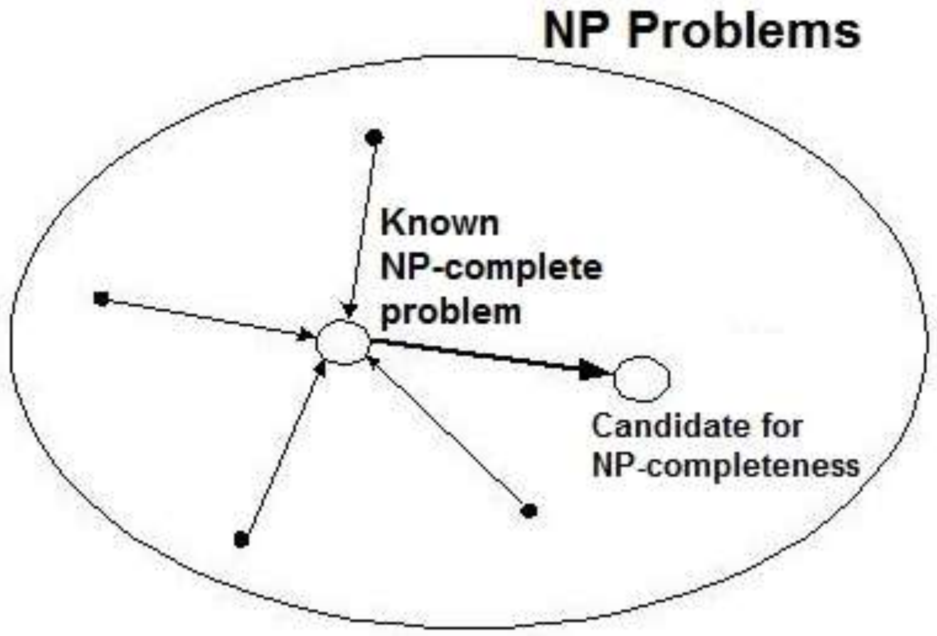
P ⊆ NP because problems in Class P can also be verified in polynomial-time.

Cook's theorem (1971): CNF-SAT is NP-complete

Examples of NP-complete problems: Hamilton Circuit, k-Clique, Knapsack, Graph-coloring and hundreds of other problems.

An **NP-complete Problem** is a problem in NP that is as difficult as any other problem in this class because any other problem in NP can be reduced to it in polynomial-time.

To prove a new problem which is NP is also NP-complete, it needs to show that an already known NP-complete problem can be reduced to the problem in question.



NP-completeness implies that if there exists a deterministic polynomial-time algorithm for just one NP-complete problem, then every problem in NP can be solved in polynomial-time by a deterministic algorithm, and hence P = NP.

Though it’s most likely P ≠ NP, it’s not proved yet.

$P \overset{?}{=} NP$ (it’s a million dollar question!)

In practice, whenever you come across a NP-complete problem, it’s wiser to find ways to **cope with it** than trying to find a polynomial-time algorithm unless you are aiming for a **Turing Award!**

www.claymath.org/millennium-problems/p-vs-np-problem

CMI

ABOUT PROGRAMS MILLENNIUM PROBLEMS PEOPLE PUBLICATIONS EVENTS EUCLID

P vs NP Problem

Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice. This is an example of what computer scientists call an NP-problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair taken from your coworker's list also appears on the list from the Dean's office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical. Indeed, the total number of ways of choosing one hundred students from the four hundred applicants is greater than the number of atoms in the known universe! Thus no future civilization could ever hope to build a supercomputer capable of solving the problem by brute force; that is, by checking every possible combination of 100 students. However, this apparent difficulty may only reflect the lack of ingenuity of your programmer. In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer. Stephen Cook and Leonid Levin formulated the P (i.e., easy to find) versus NP (i.e., easy to check) problem independently in 1971.

Rules:

Rules for the Millennium Prizes

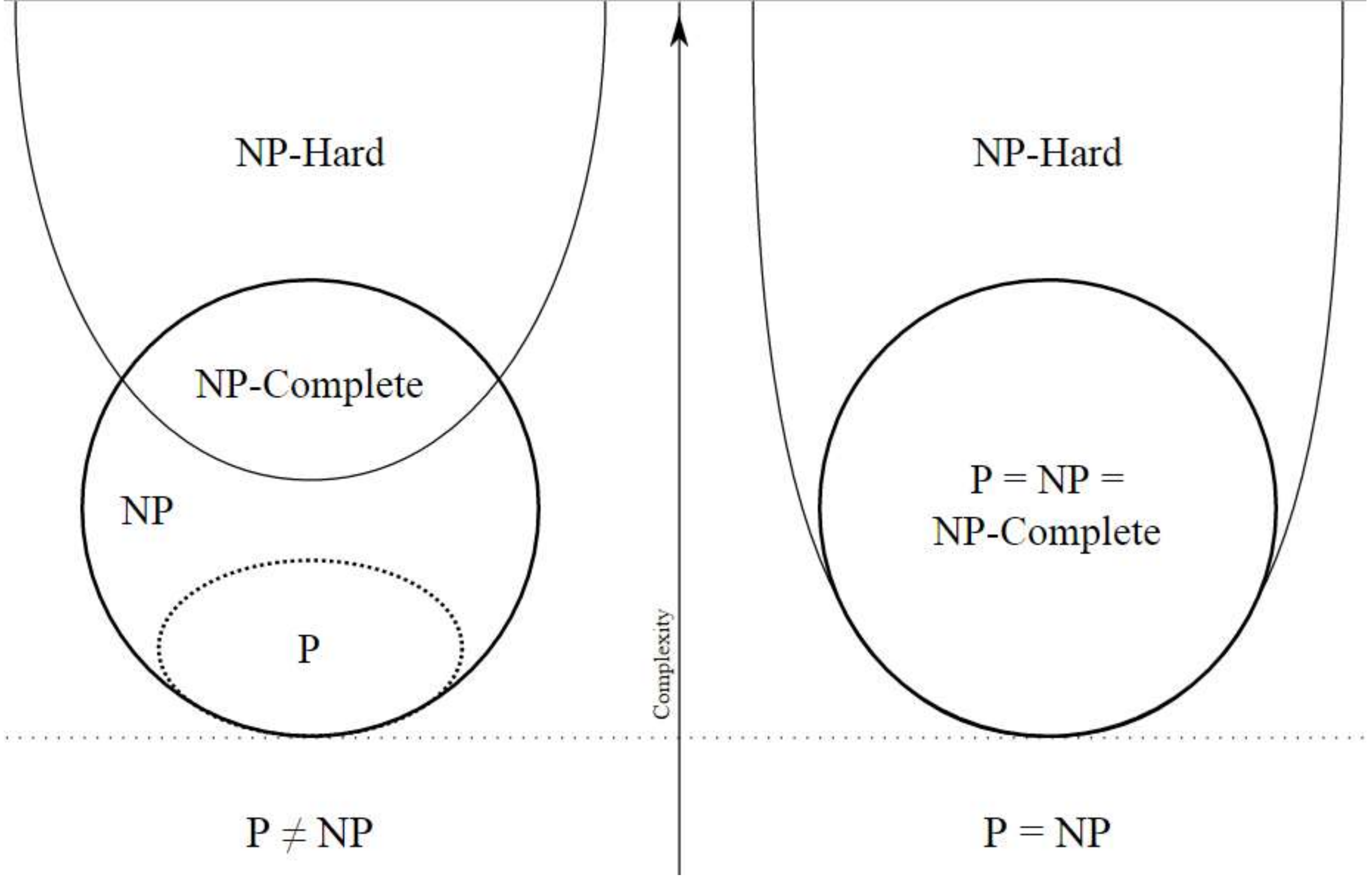
Related Documents:

Official Problem Description

Minesweeper

Related Links:

Lecture by Vijaya Ramachandran


NP-Hard
NP-Complete
NP
P
P ≠ NP
Complexity
NP-Hard
P = NP =
NP-Complete
P = NP

Coping with the Limitations of Algorithm Power

How to tackle difficult combinatorial problems?

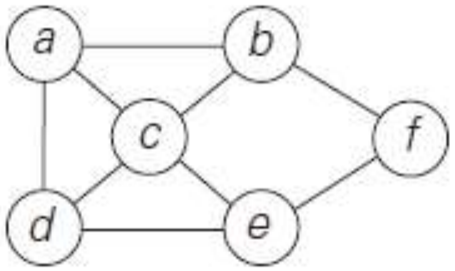
- Exact algorithms
  - Exhaustive Search
    - Useful only for small instances
  - Backtracking
    - Eliminates infeasible cases quickly
  - Branch and Bound
    - Refined idea of backtracking for optimization problems
- Approximation algorithms
  - Sub-optimal solution in polynomial time

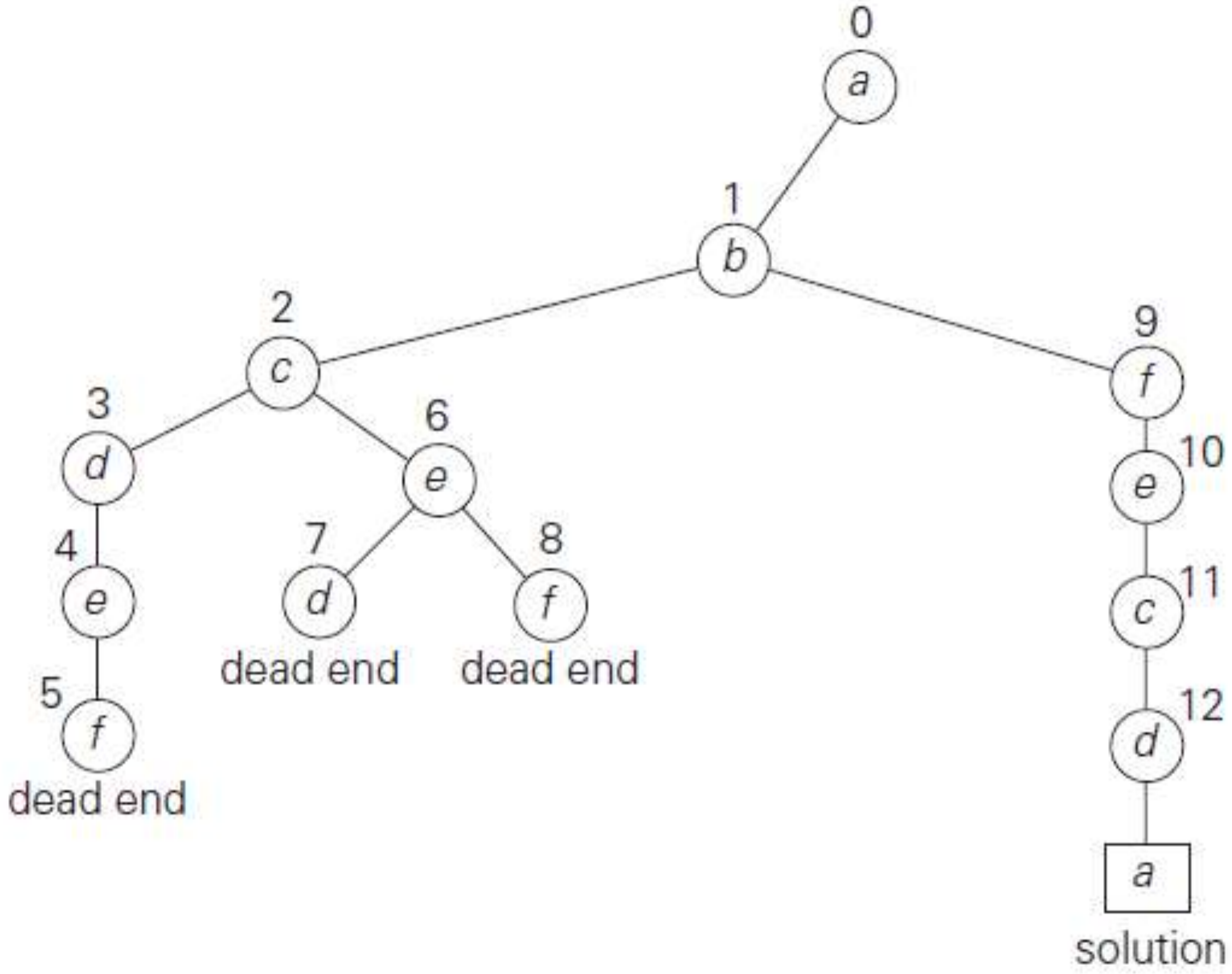
Backtracking

The exhaustive search technique suggests generating all candidate solutions and then identifying the one (or the ones) with a desired property.

- Construct solutions one component at a time and evaluate such partially constructed candidates.
- Construct the **state-space tree**
    - **non-leaf nodes**: promising nodes with partial solutions
    - **leaves**: non-promising nodes or solutions
    - **edges**: choices in extending partial solutions
- Explore the state space tree using **depth-first search**.
- **Backtrack** at non-promising nodes.

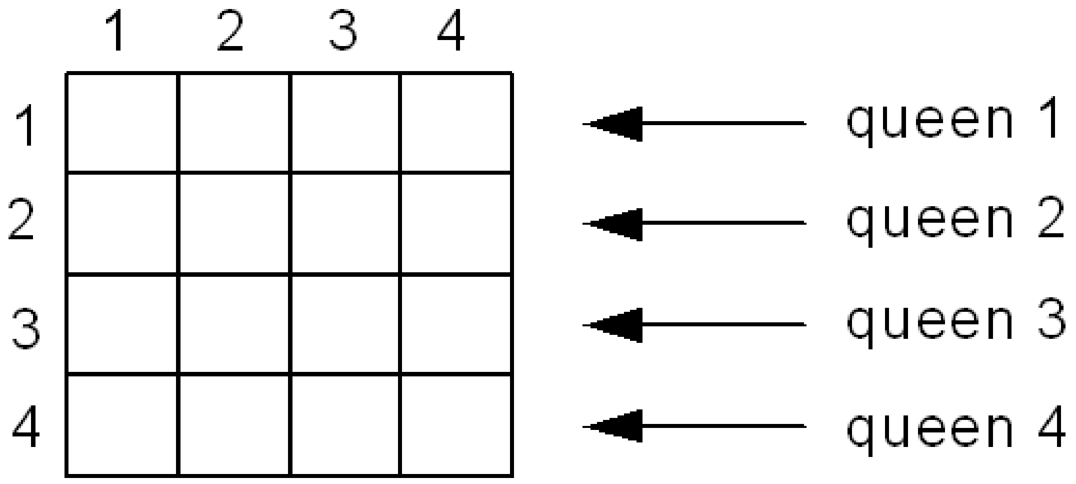
Backtracking - Hamiltonian Circuit Problem



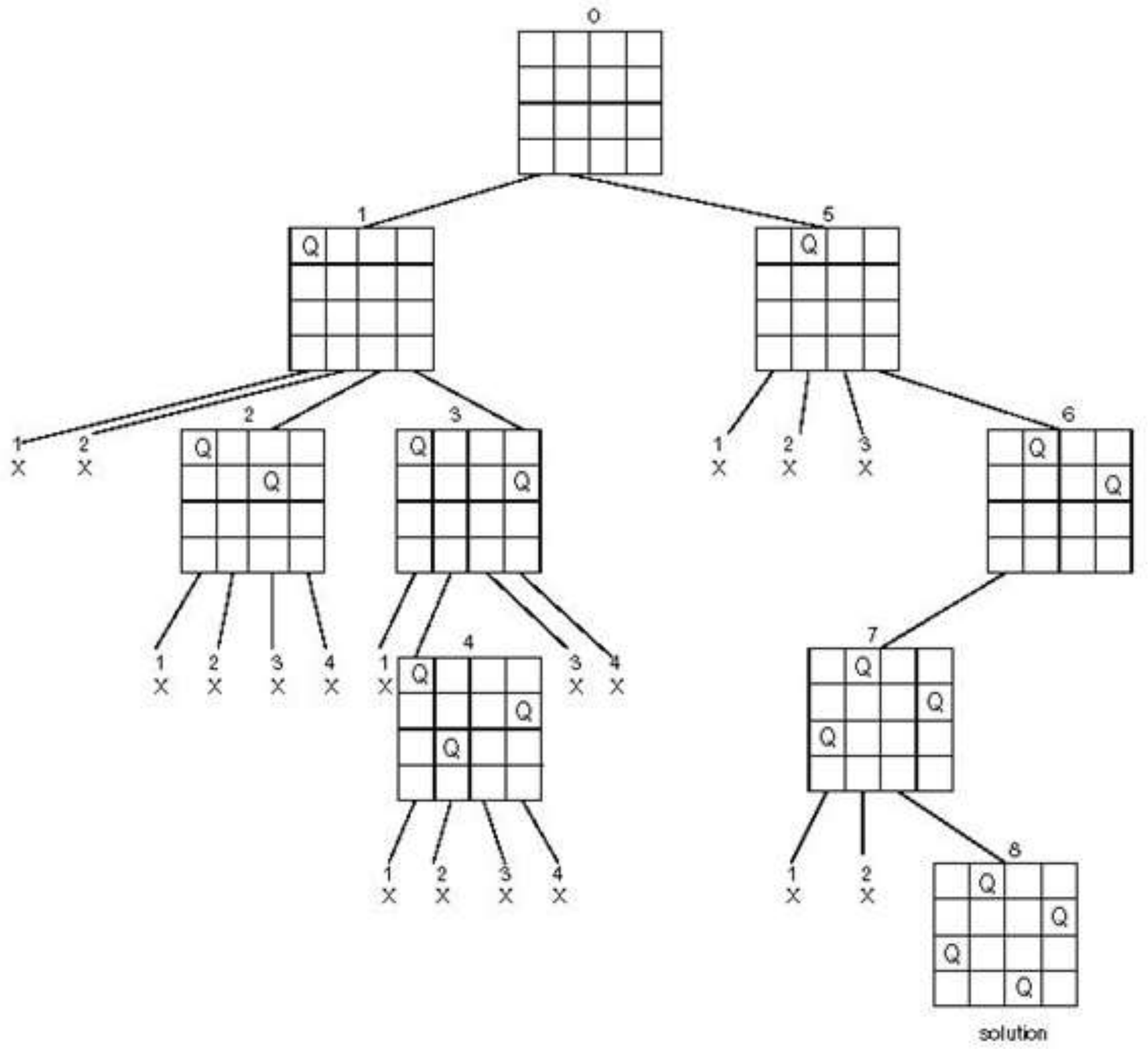


Backtracking - n-Queens Problem

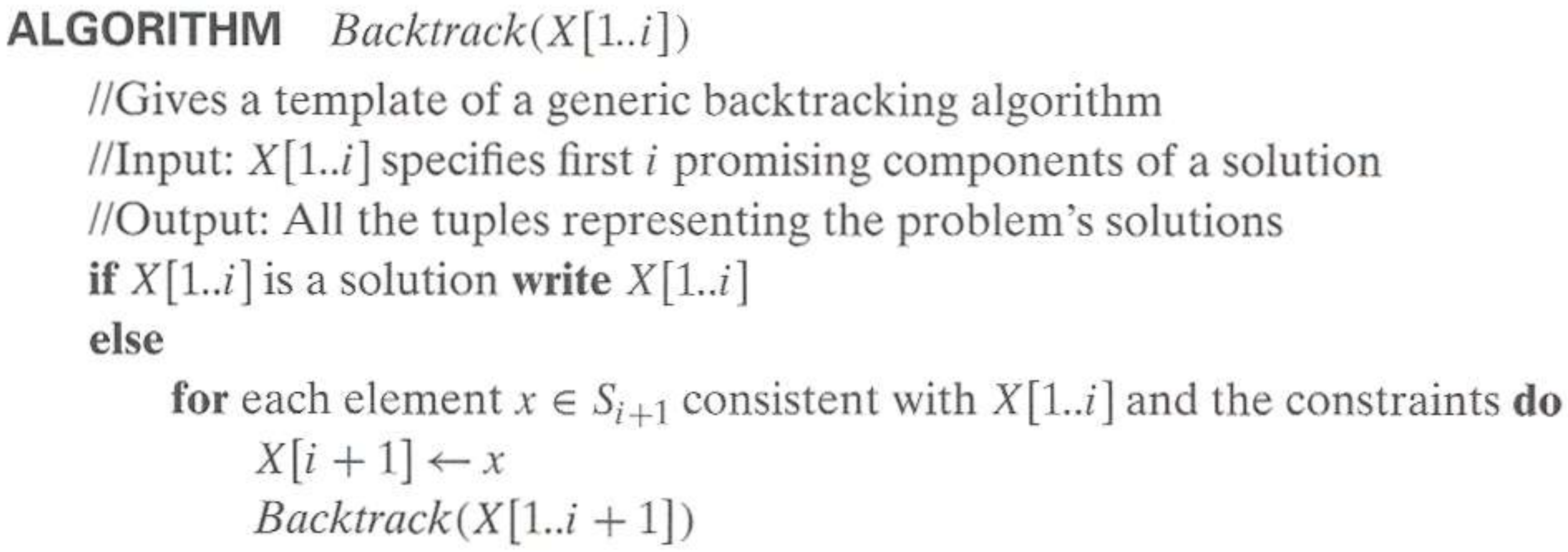
Place n queens on an n-by-n chessboard so that no two of them are in the same row, column, or diagonal.



Backtracking - n-Queens Problem



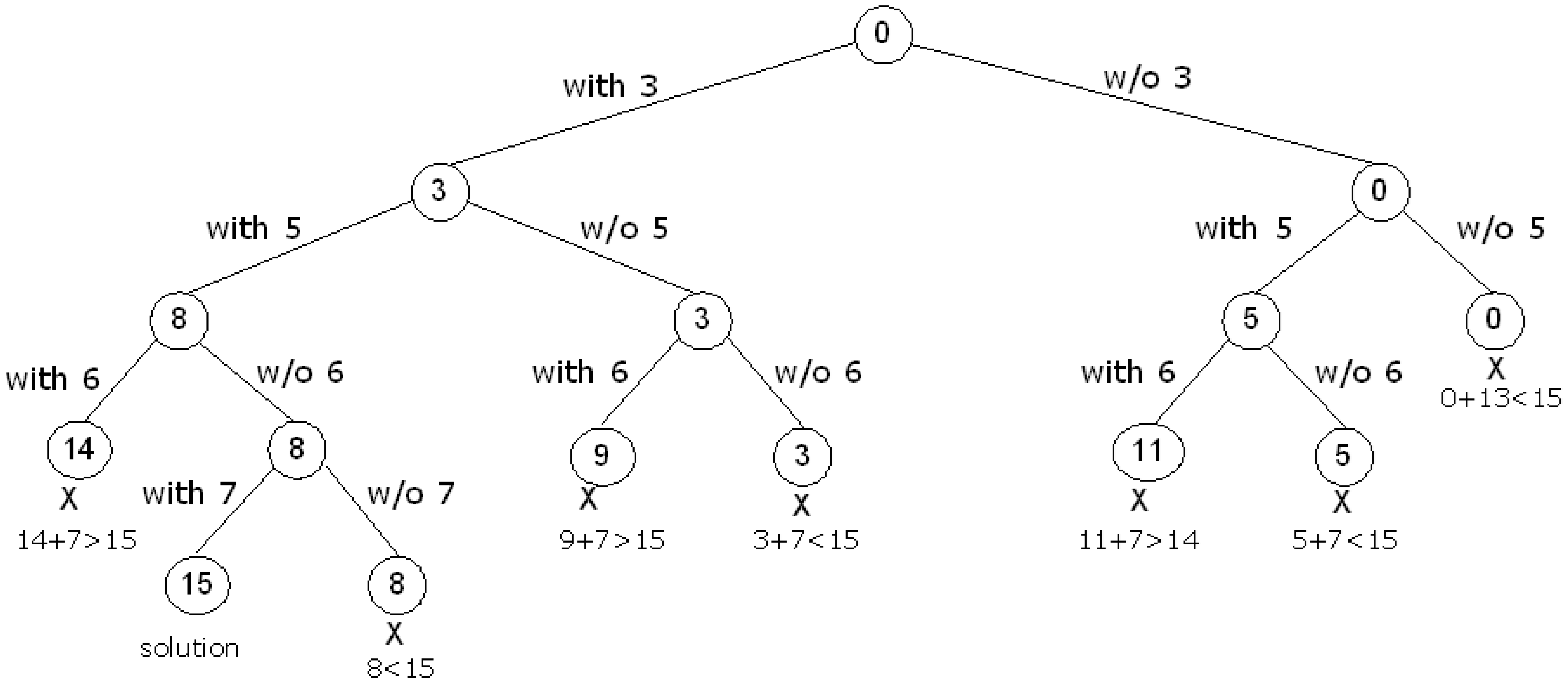
Backtracking - template algorithm



```
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
        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 - Subset-Sum Problem

S = {3, 5, 6, 7} and d = 15



Branch-and-Bound

- An enhancement of backtracking
- Applicable to optimization problems
- Makes a note of the best solution seen so far
- For each node (partial solution) of a state-space tree, computes a **bound** on the value of the objective function for all descendants of the node (extensions of the partial solution)

Branch-and-Bound - Knapsack Problem

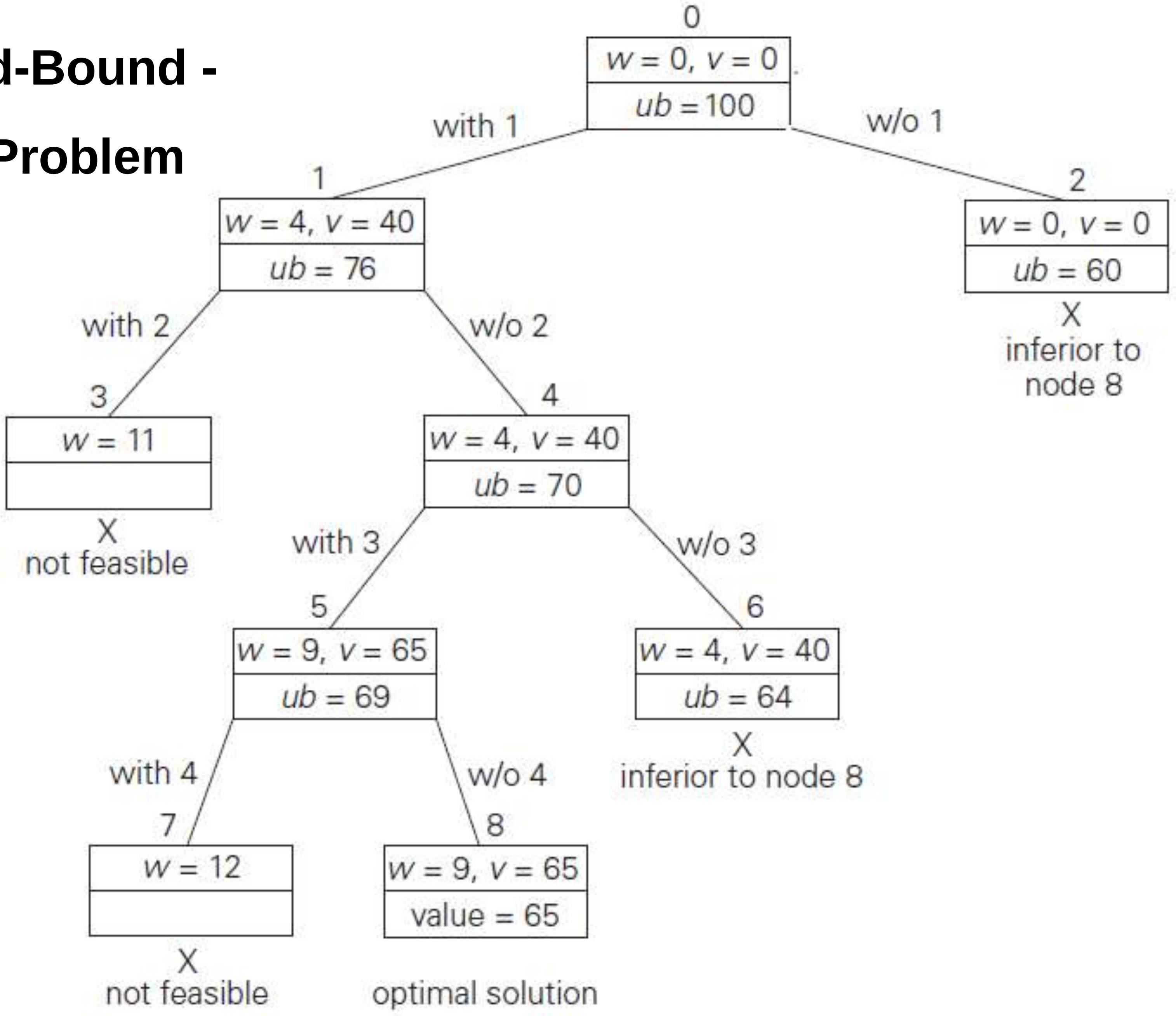
$$v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n.$$

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

item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

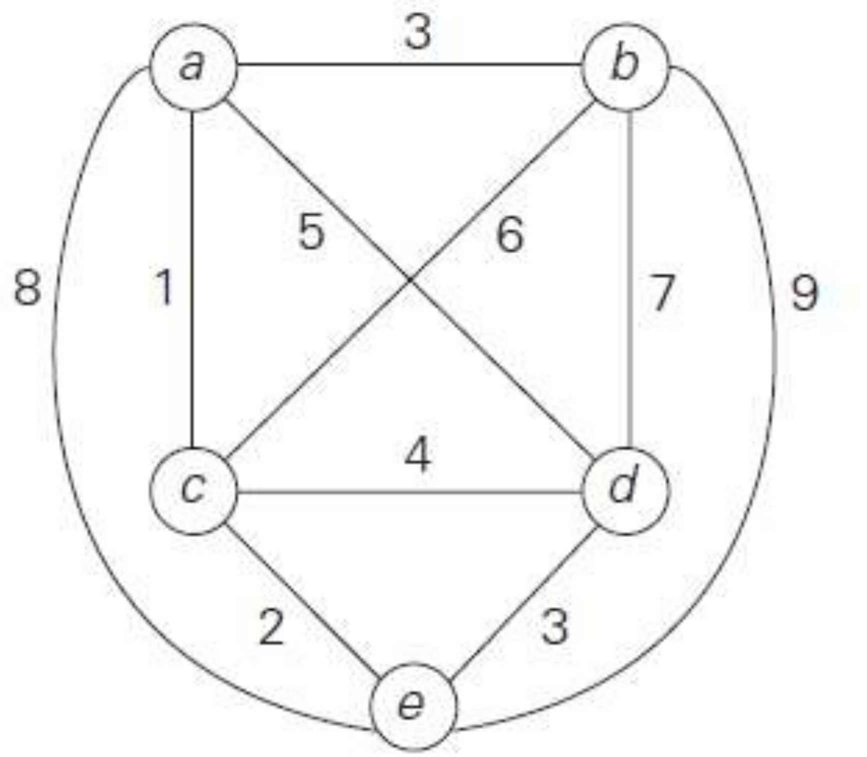
The knapsack's capacity W is 10.

Branch-and-Bound - Knapsack Problem



Branch-and-Bound - Travelling Salesman Problem

Objective is to **minimize** the cost of the circuit. The effort of minimizing does not help if the lower bound is already higher than a solution found so far at an intermediate stage.



Branch-and-Bound - Travelling Salesman Problem

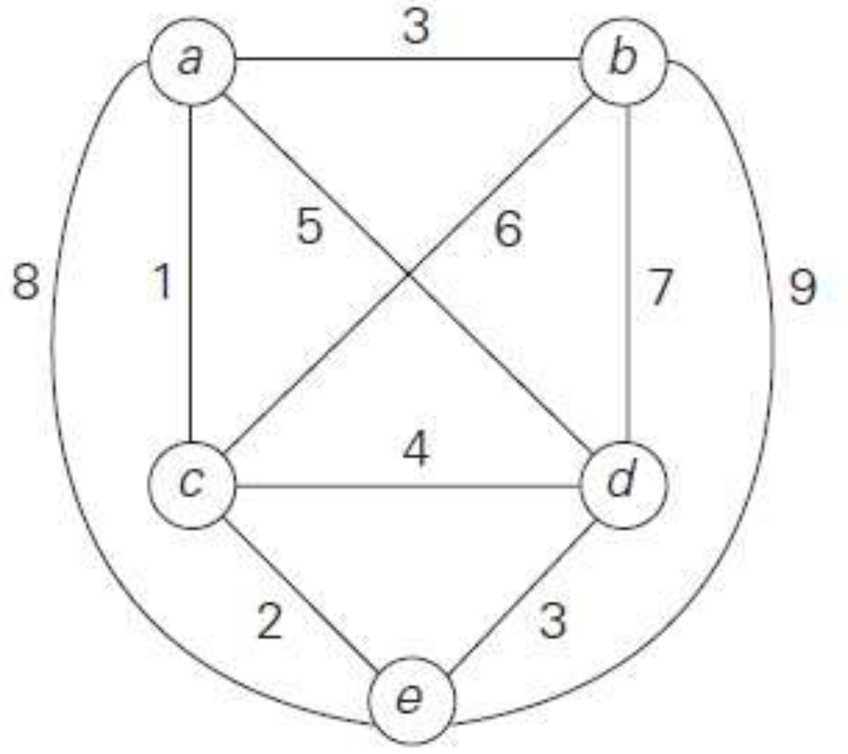
- One very simple lower bound can be obtained by finding the smallest element in the intercity distance matrix D and multiplying it by the number of cities n.
  - Lower bound = n * (cost of the least cost edge)
- A better method is for each city i, $1 \leq i \leq n$, find the sum s_i of the distances from city i to the two nearest cities; compute the sum s of these n numbers, divide the result by 2
  - Lower bound = ceil(s/2)

Branch-and-Bound - Travelling Salesman Problem

Initial lower bound =

ceil(**((1 + 3) + (3 + 6) + (1 + 2) +**

(3 + 4) + (2 + 3)) / 2) = 14.



Lower bound for all the

Hamiltonian circuits of the graph

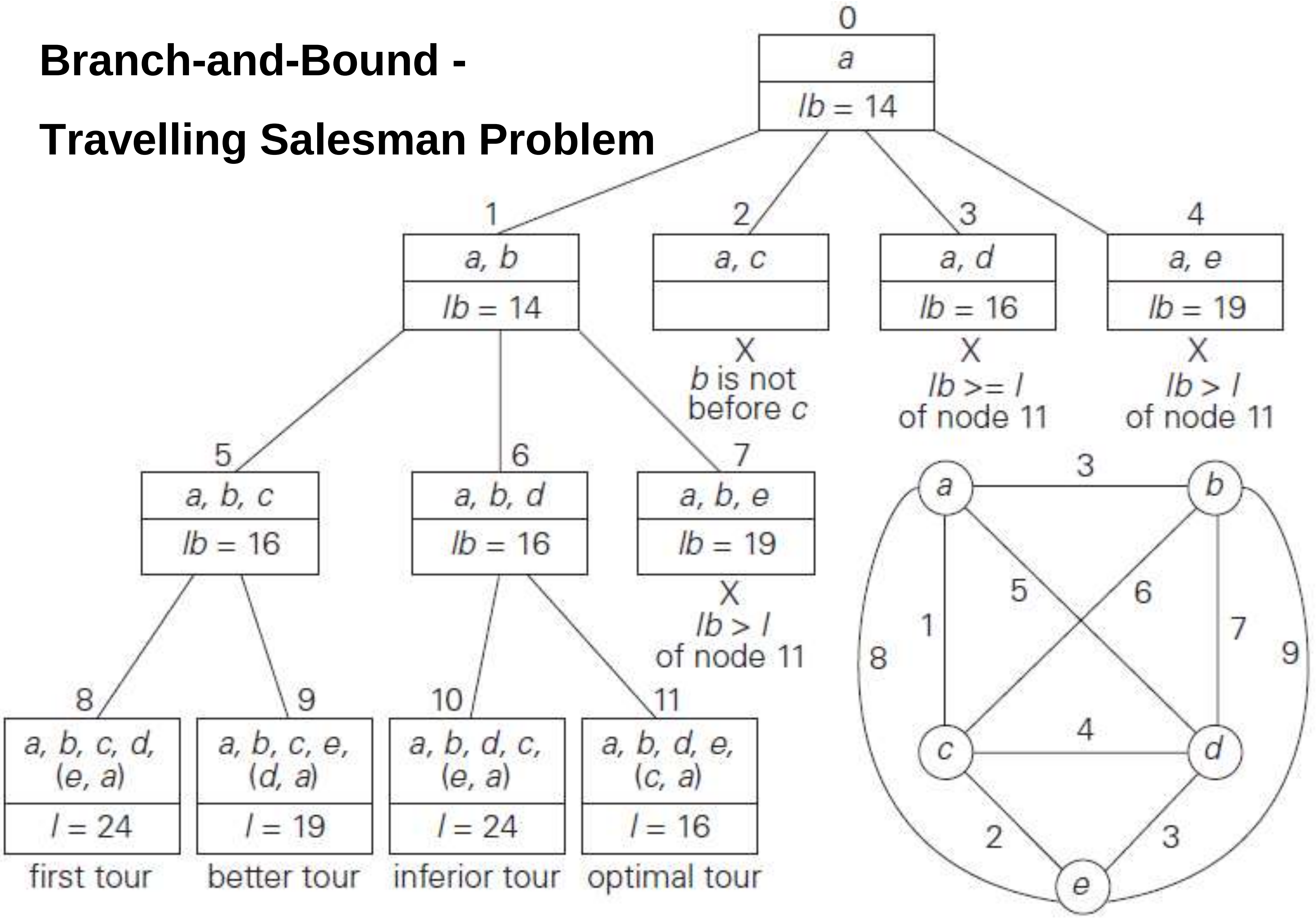
that must include edge (a, d) =

ceil(((1 + **5**) + (3 + 6) + (1 + 2) +

(**5** + 3) + (2 + 3)) / 2) = **16**.

Branch-and-Bound -

Travelling Salesman Problem



Branch-and-Bound - Assignment Problem

Objective is to **minimize** the cost of the assignment. The effort of minimizing does not help if the lower bound is already higher than a solution found so far at an intermediate stage.

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

Approximation Algorithms for NP-Hard Problems

- For NP-hard problems, there are no known polynomial-time algorithms.
- For some of these problems, a faster algorithm could exist which may not give an optimal solution all the time. Such Approximation Algorithms are sensible for some applications which otherwise has a NP-hard problem.
- Eg: Travelling Salesman Problem, Knapsack Problem

Limitations keep mortals mortal!

</
Limitations of Algorithm Power and
Coping with the Limitations of Algorithm Power
>