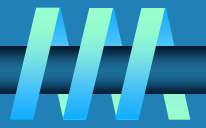
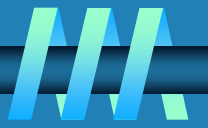


Limitations of Algorithmic Power

Lower-Bound Arguments, Decision Trees, P, NP, and NP Complete Problems

Lower Bounds

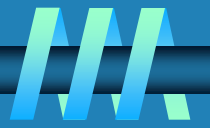


***Lower bound*: an estimate on a minimum amount of work needed to solve a given problem**

Examples:

- **number of comparisons needed to find the largest element in a set of *n* numbers**
- **number of comparisons needed to sort an array of size *n***
- **number of comparisons necessary for searching in a sorted array**
- **number of multiplications needed to multiply two *n*-by-*n* matrices**

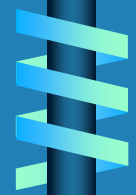
Lower Bounds (cont.)



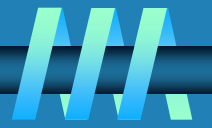
- Lower bound can be
  - an exact count
  - an efficiency class (Ω)

- ***Tight*** lower bound: there exists an algorithm with the same efficiency as the lower bound

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



Methods for Establishing Lower Bounds



- trivial lower bounds
- information-theoretic arguments (decision trees)
- adversary arguments
- problem reduction

Trivial Lower Bounds

Trivial lower bounds: based on counting the number of items that must be processed in input and generated as output

Examples

- finding max element
- polynomial evaluation
- sorting
- element uniqueness
- Hamiltonian circuit existence

Conclusions

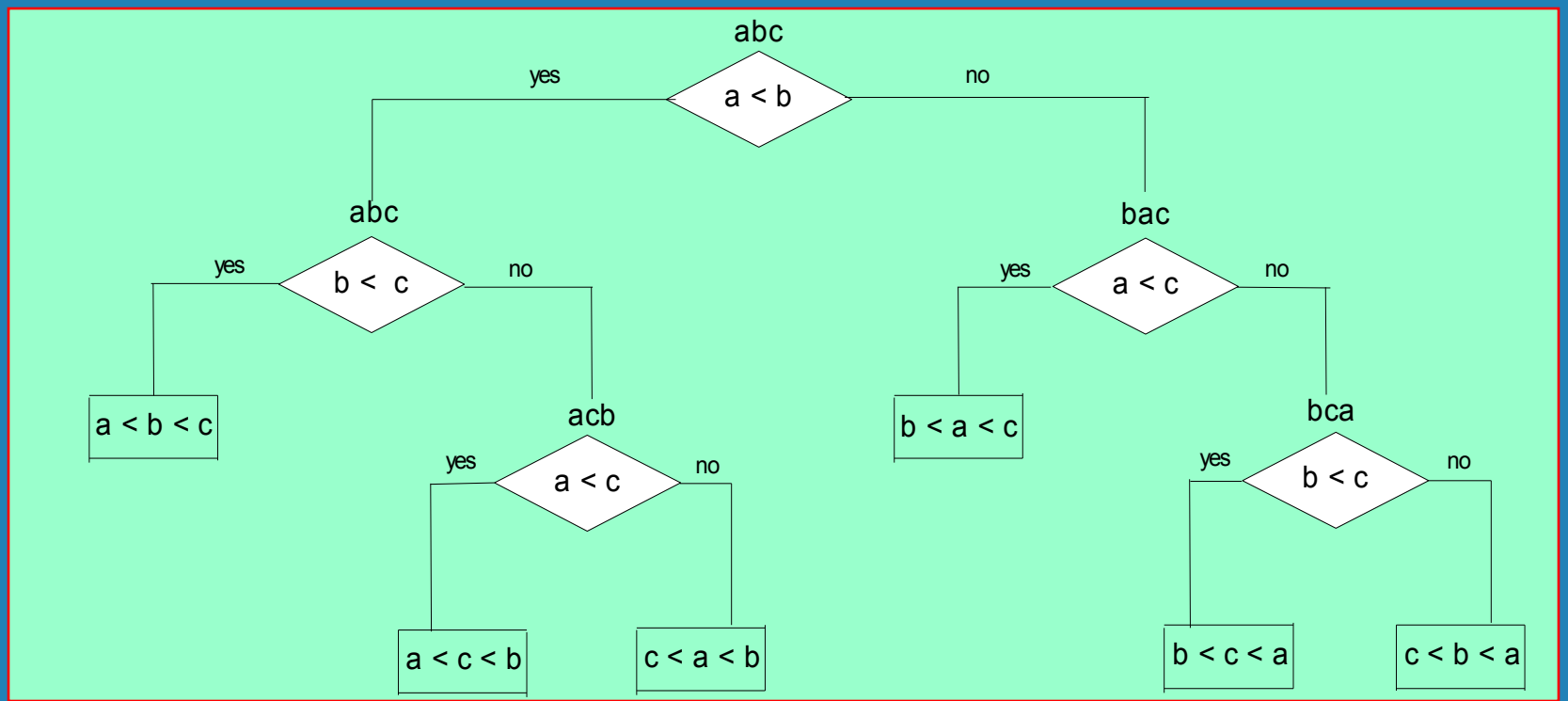
- may and may not be useful
- be careful in deciding how many elements must be processed

Decision Trees

Decision tree — a convenient model of algorithms involving comparisons in which:

- internal nodes represent comparisons
- leaves represent outcomes

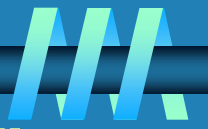
Decision tree for 3-element insertion sort



Decision Trees and Sorting Algorithms

- Any comparison-based sorting algorithm can be represented by a decision tree
- 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
- $\lceil \log_2 n! \rceil \approx n \log_2 n$
- This lower bound is tight (mergesort)

Adversary Arguments



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

Example 1: "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

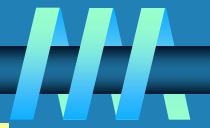
Example 2: Merging two sorted lists of size n

$a_1 < a_2 < \ldots < a_n$ and $b_1 < b_2 < \ldots < b_n$

Adversary: $a_i < b_j$ iff $i < j$

Output $b_1 < a_1 < b_2 < a_2 < \ldots < b_n < a_n$ requires $2n$-1 comparisons of adjacent elements

Lower Bounds by Problem Reduction



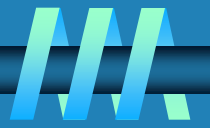
Idea: If problem P is at least as hard as problem Q, then a lower bound for Q is also a lower bound for P.

Hence, find problem Q with a known lower bound that can be reduced to problem P in question.

Example: P is finding MST for n points in Cartesian plane

Q is element uniqueness problem (known to be in $\Omega(n \log n)$)

Classifying Problem Complexity

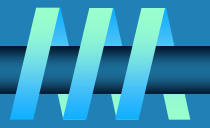


Is the problem *tractable*, i.e., is there a polynomial-time (O($p(n)$)) algorithm that solves it?

Possible answers:

- **yes (give examples)**
- **no**
  - **because it's been proved that no algorithm exists at all (e.g., Turing's *halting problem*)**
  - **because it's been be proved that any algorithm takes exponential time**
- **unknown**

Problem Types: Optimization and Decision



- ***Optimization problem***: find a solution that maximizes or minimizes some objective function

- ***Decision problem***: answer yes/no to a question

Many problems have decision and optimization versions.

E.g.: traveling salesman problem

- *optimization*: find Hamiltonian cycle of minimum length
- *decision*: find Hamiltonian cycle of length $\leq m$

Decision problems are more convenient for formal investigation of their complexity.

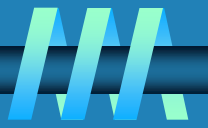
Class *P*

P: the class of decision problems that are solvable in O($p(n)$) time, where $p(n)$ is a polynomial of problem's input size n

Examples:

- searching
- element uniqueness
- graph connectivity
- graph acyclicity
- primality testing (finally proved in 2002)

Class *NP*



NP (***nondeterministic polynomial***): class of decision problems whose proposed solutions can be verified in polynomial time = solvable by a *nondeterministic polynomial algorithm*

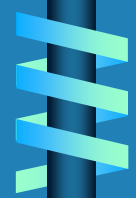
A ***nondeterministic polynomial algorithm*** is an abstract two-stage procedure that:

- generates a random string purported to solve the problem
- checks whether this solution is correct in polynomial time

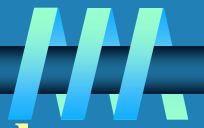
By definition, it solves the problem if it's capable of generating and verifying a solution on one of its tries

Why this definition?

- led to development of the rich theory called "computational complexity"



Example: CNF satisfiability



Problem: Is a boolean expression in its conjunctive normal form (CNF) satisfiable, i.e., are there values of its variables that makes it true?

This problem is in *NP*. Nondeterministic algorithm:

- **Guess truth assignment**
- **Substitute the values into the CNF formula to see if it evaluates to true**

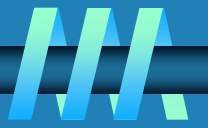
Example: (A | ¬B | ¬C) & (A | B) & (¬B | ¬D | E) & (¬D | ¬E)

Truth assignments:

A	B	C	D	E
0	0	0	0	0
.	.	.		
1	1	1	1	1

Checking phase: O(*n*)

What problems are in *NP*?

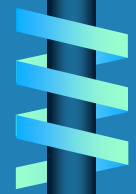


- Hamiltonian circuit existence
- Partition problem: Is it possible to partition a set of n integers into two disjoint subsets with the same sum?
- Decision versions of TSP, knapsack problem, graph coloring, and many other combinatorial optimization problems. (Few exceptions include: MST, shortest paths)

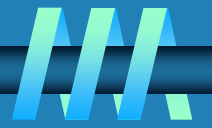
- All the problems in *P* can also be solved in this manner (no guessing is necessary), so we have:

$$P \subseteq NP$$

- Big question: $P = NP$?

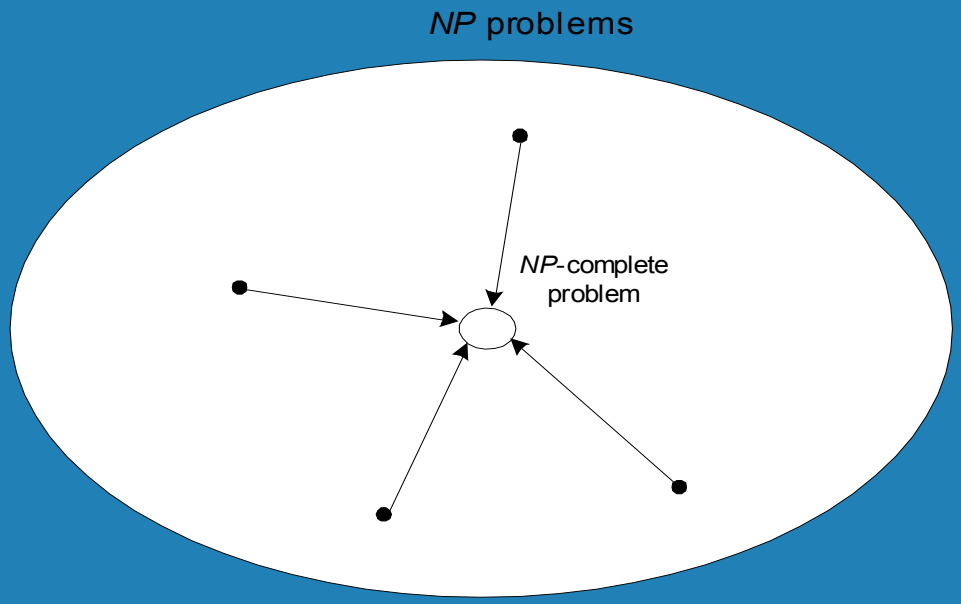


NP-Complete Problems



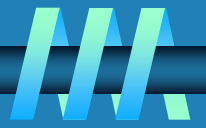
A decision problem *D* is *NP*-complete if it's as hard as any problem in *NP*, i.e.,

- ***D* is in *NP***
- **every problem in *NP* is polynomial-time reducible to *D***

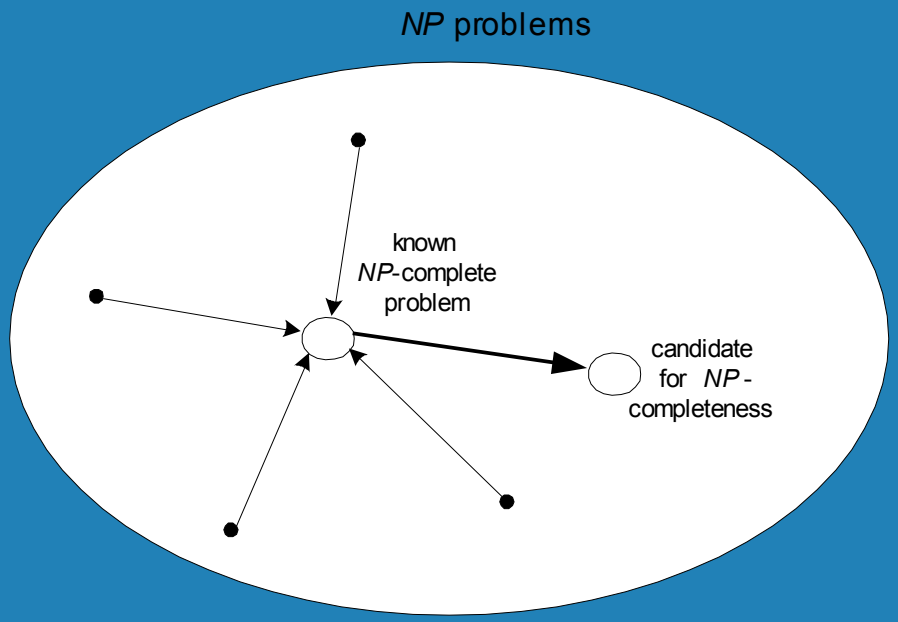


Cook's theorem (1971): CNF-sat is *NP*-complete

NP-Complete Problems (cont.)



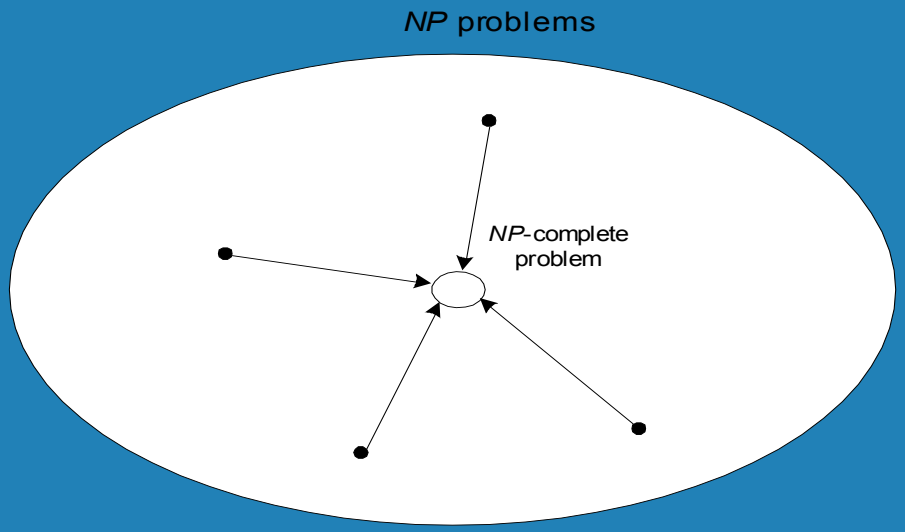
Other *NP*-complete problems obtained through polynomial-time reductions from a known *NP*-complete problem



Examples: TSP, knapsack, partition, graph-coloring and hundreds of other problems of combinatorial nature

P = *NP* ? Dilemma Revisited

- ***P* = *NP*** would imply that every problem in ***NP***, including all ***NP*-complete problems, could be solved in polynomial time**
- **If a polynomial-time algorithm for just one *NP*-complete problem is discovered, then every problem in *NP* can be solved in polynomial time, i.e., *P* = *NP***



- **Most but not all researchers believe that *P* ≠ *NP* , i.e. *P* is a proper subset of *NP***

Backtracking
n Queens Problem

Prepared by
Dr.Rashmi S

BACKTRACKING

- Principal idea is to construct solutions one component at a time
- Evaluate such partially constructed candidates as follows
- If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component
- If there is no legitimate option for the next component, no alternatives for *any remaining component* need to be considered
- In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option.

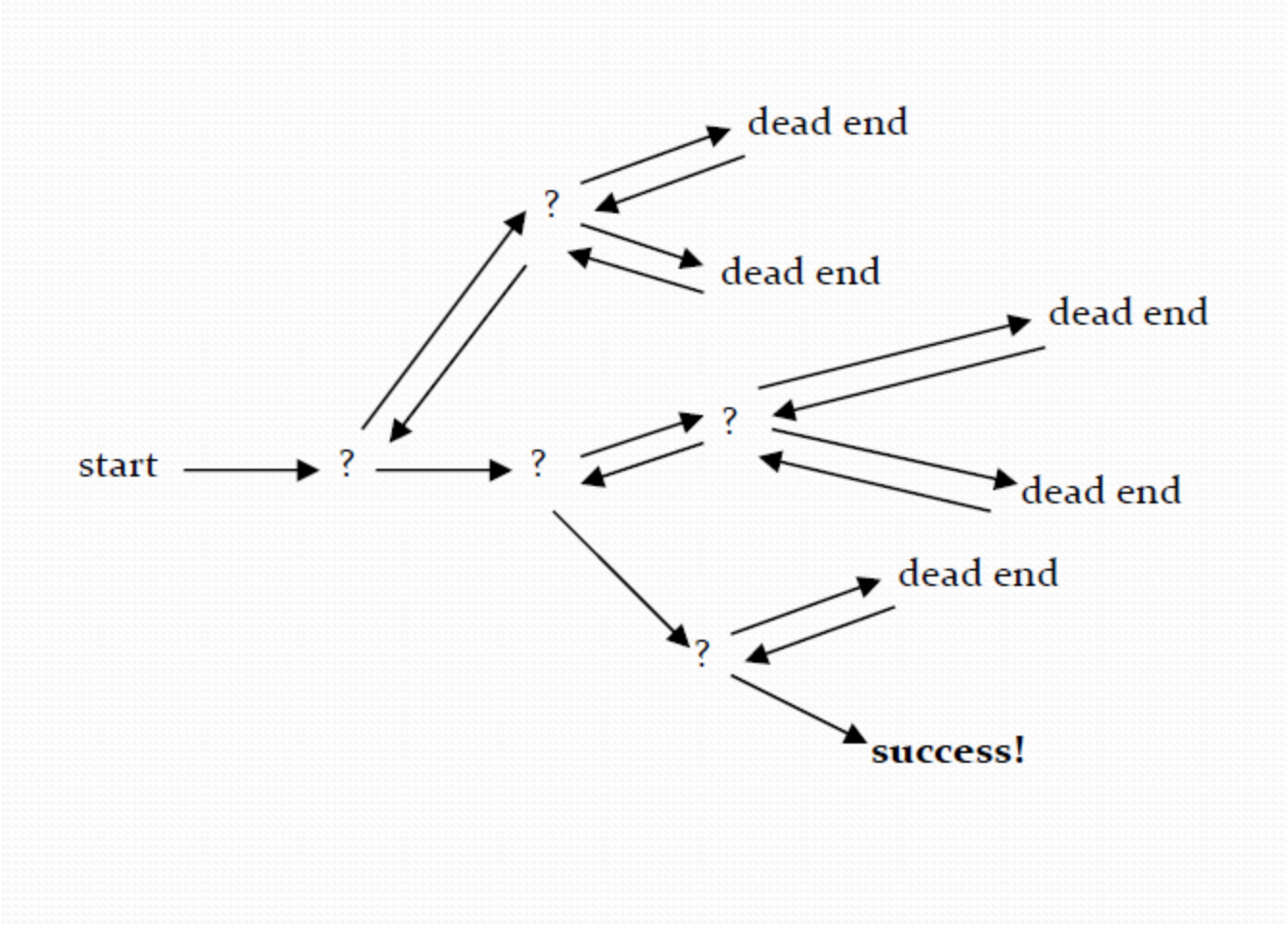
State space tree

- Backtracking is a modified depth-first search of a tree
- Root represents an initial state before the search for a solution begins.
- A node in a state-space tree is ***promising if it corresponds to a*** partially constructed solution that may still lead to a complete solution; otherwise,it is called ***nonpromising.***
- ***Leaves represent either nonpromising dead ends or*** complete solutions found by the algorithm.

State space tree

- If the current node is promising, its child is generated and the processing moves to this child
- If the current node turns out to be nonpromising, the algorithm backtracks to the node's parent to consider the next possible option for its last component
  - if there is no such option, it backtracks one more level up the tree, and so on.
- Finally, if the algorithm reaches a complete solution to the problem, it either stops or continues searching for other possible solutions

BACKTRACKING



N-Queen Problem

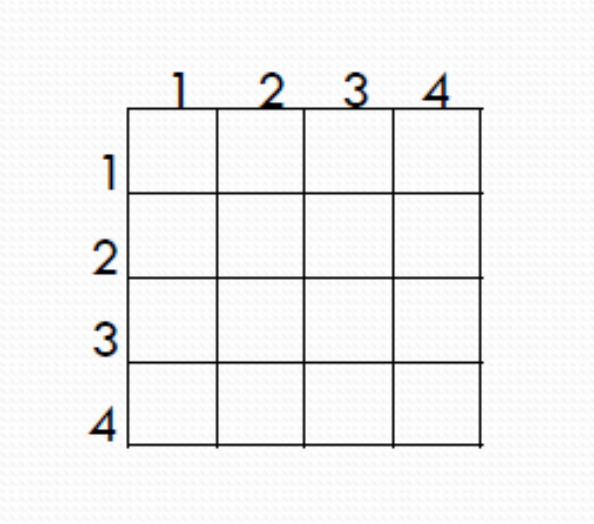
Problem:-

The problem is to place n queens on an n-by-n chessboard so that no two queens attack each other by being in the same row, or in the same column, or in the same diagonal

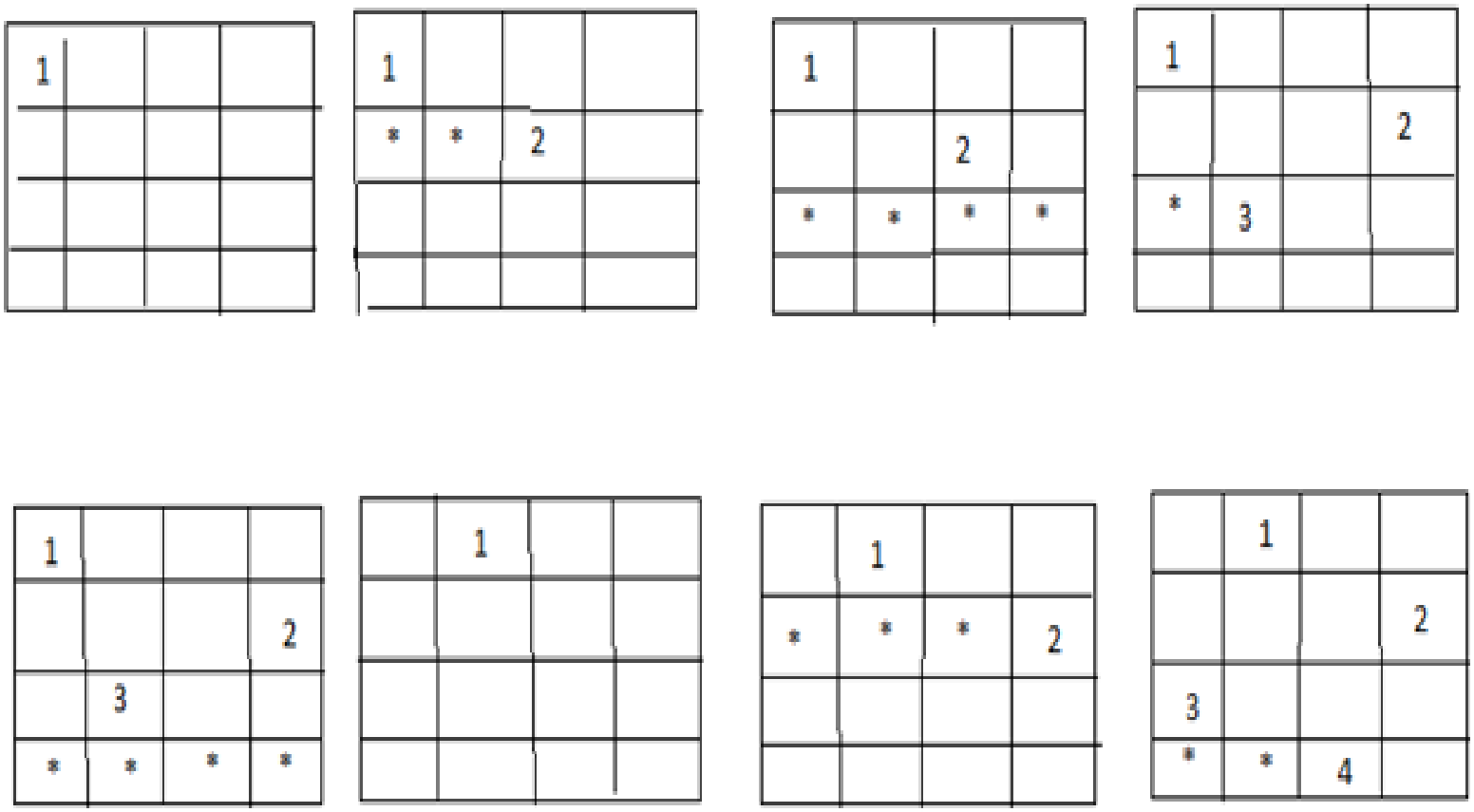
4-Queens Problem

Consider n=4 ,

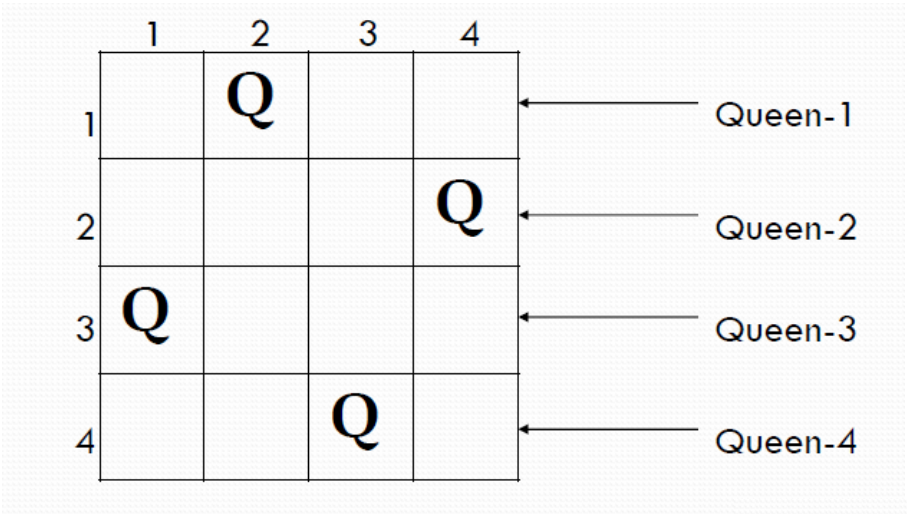
Use a 4-by-4 chessboard where 4-Queens have to be placed in such a way so that no two queen can attack each other

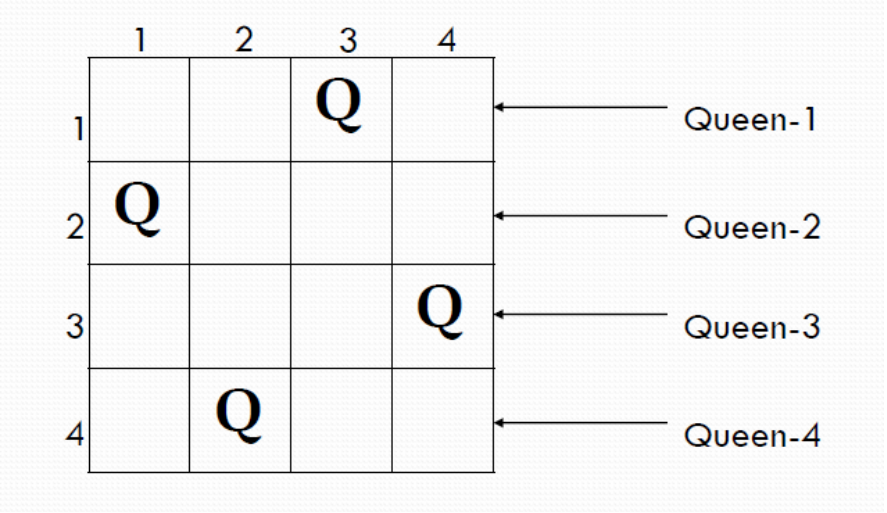


Step-by-step solution
4- Queens problem



2 solutions to 4-Queens problem





State space tree for 4-Queens

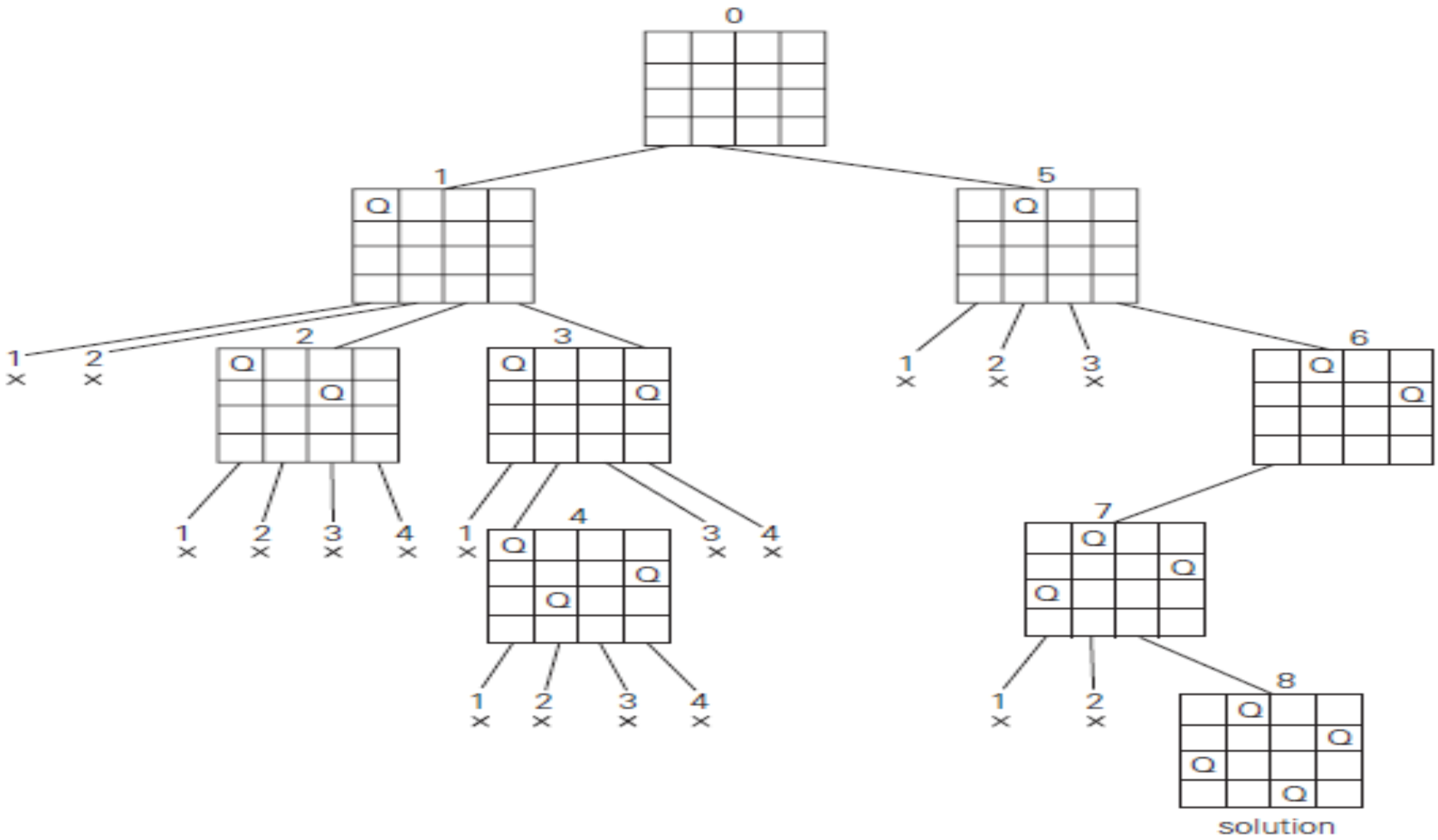
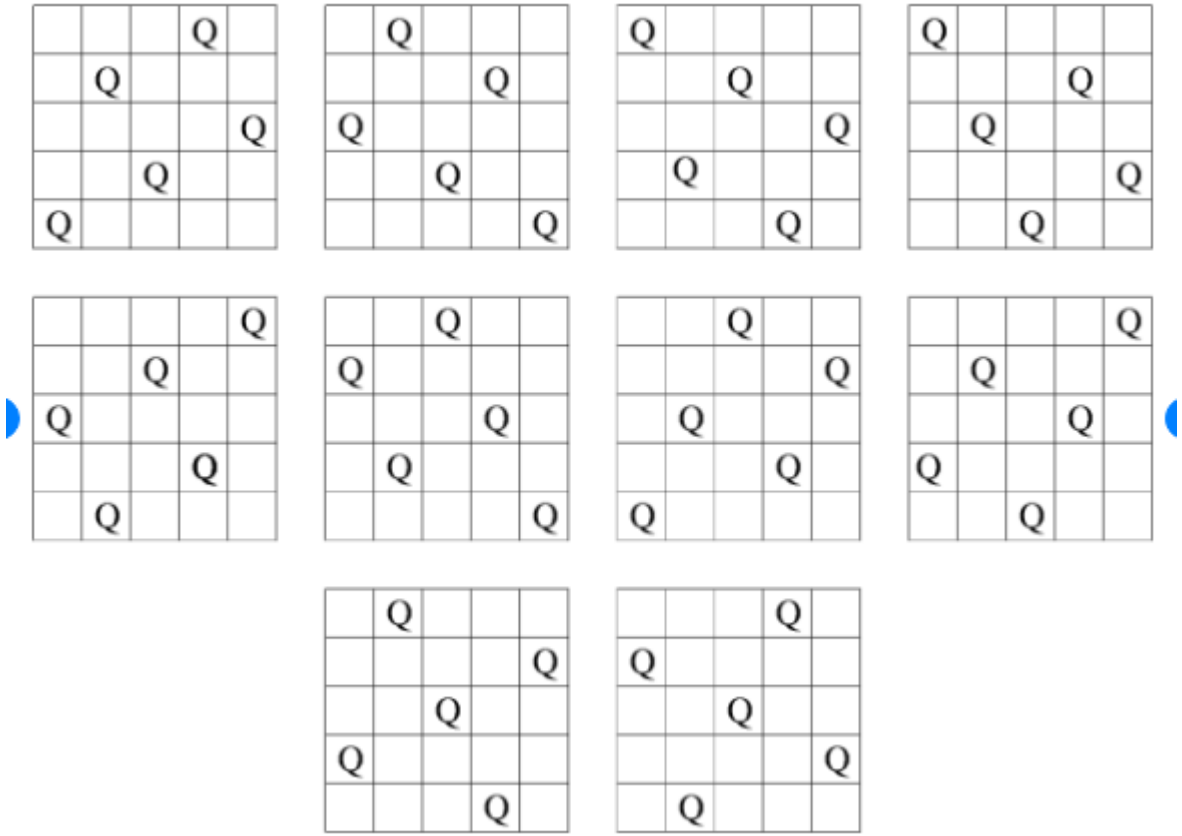


FIGURE 12.2 State-space tree of solving the four-queens problem by backtracking. × denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.

10 solutions to 5-Queens problem



N- Queens

A single solution to the *n-queens problem* for any $n \geq 4$ *can be found in linear time*

SUM-OF-SUBSET PROBLEM

- **Subset-sum Problem:** The problem is to find a subset of a given set $S = \{s_1, s_2, \ldots, s_n\}$ of **'n'** positive integers whose sum is equal to a given positive integer **'d'**.

- **Observation :** It is convenient to sort the set's elements in increasing order, $S_1 \leq S_2 \leq \ldots\ldots \leq S_n$. And each set of solutions don't need to be necessarily of fixed size.

- **Example :** For S = {3, 5, 6, 7} and d = 15, the solution is shown below :-

  Solution = {3, 5, 7}

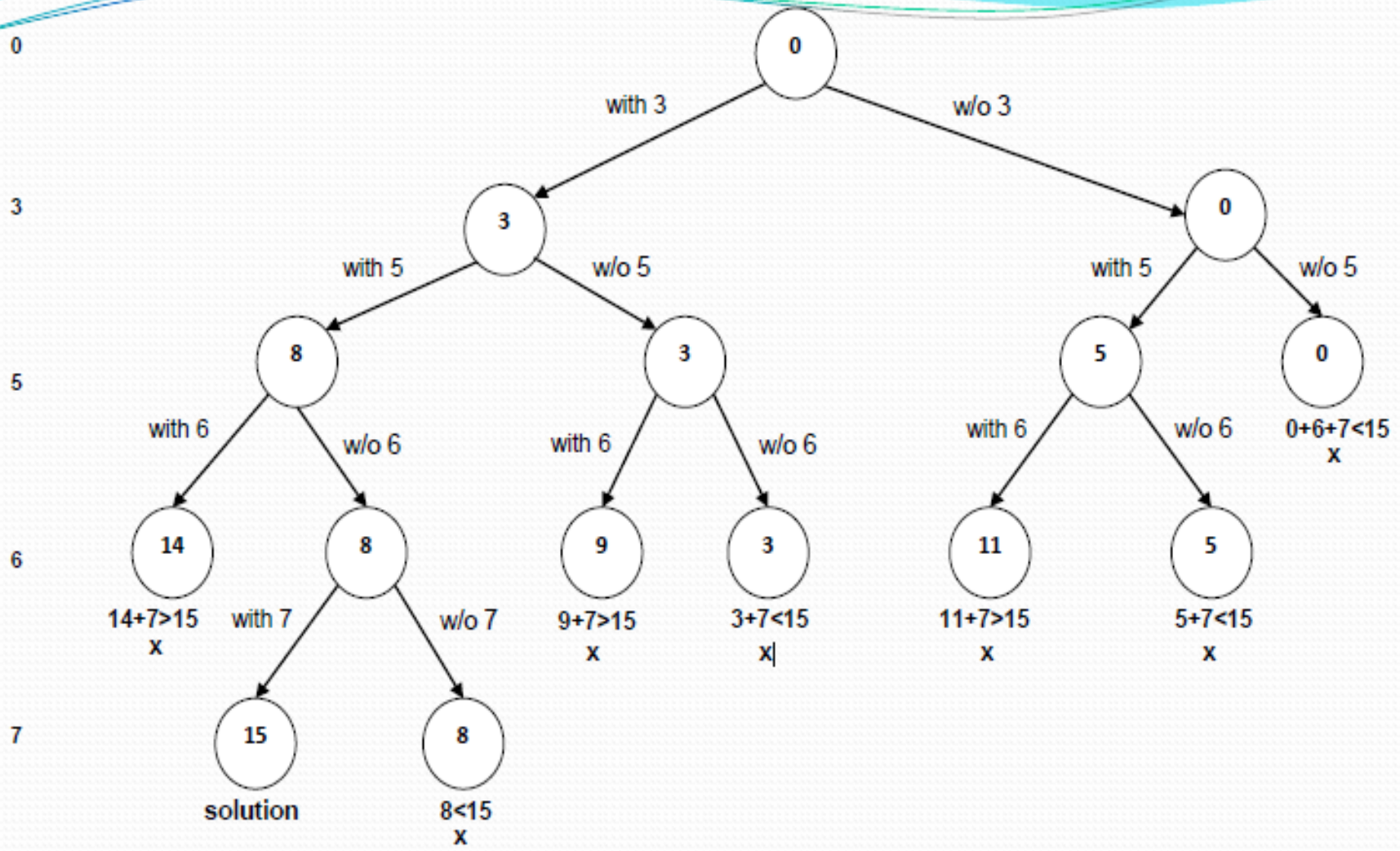


Figure : Compete state-space tree of the backtracking algorithm applied to the instance **S = {3, 5, 6, 7}** and **d = 15** of the subset-sum problem. The number inside a node is the sum of the elements already included in subsets represented by the node. The inequality below a leaf indicates the reason for its termination.

SUM-OF-SUBSET

- Record the value of *s, the sum of these numbers, in the node*
- *If s is equal* to *d, we have a solution to the problem*
  - *report this result and stop or*
  - if all the solutions need to be found, continue by backtracking to the node's parent.
- If *s is not equal to d, we can terminate the node as nonpromising if either* of the following two inequalities holds:

$$s + a_{i+1} > d \quad \text{(the sum } s \text{ is too large)},$$

$$s + \sum_{j=i+1}^{n} a_j < d \quad \text{(the sum } s \text{ is too small)}.$$

General Pseudo code

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$(X[1..i+1])$

Thank you