# INT102
## Algorithmic Foundations And Problem Solving
### Coping with the Limitations of Algorithm Power

---

Dr Jia WANG

Department of Computer Science

西交利物浦大學
Xi'an Jiaotong-Liverpool University

Acknowledgment: The slides are adapted from ones by Dr. Prudence Wong

# Tackling Difficult Combinatorial Problems

There are two principal approaches to tackling difficult combinatorial problems (NP-hard problems):

- Use a strategy that guarantees solving the problem exactly but doesn't guarantee to find a solution in polynomial time
- Use an approximation algorithm that can find an approximate (sub-optimal) solution in polynomial time

# Exact Solution Strategies

- *exhaustive search* (brute force)
  - useful only for small instances
- *backtracking*
  - eliminates some unnecessary cases from consideration
  - yields solutions in reasonable time for many instances but worst case is still exponential
- *branch-and-bound*
  - further refines the backtracking idea for optimization problems
- *dynamic programming*
  - applicable to some problems (e.g., the knapsack problem)

# Algorithm Design Techniques

- Greedy
  - Shortest path, minimum spanning tree, …
- Divide and Conquer
  - Divide the problem into smaller subproblems, solve them, and combine into the overall solution
  - Often done recursively
  - Quick sort, merge sort are great examples
- Dynamic Programming
  - Brute force through all possible solutions, storing solutions to subproblems to avoid repeat computation
- Backtracking
  - A clever form of exhaustive search

# Backtracking

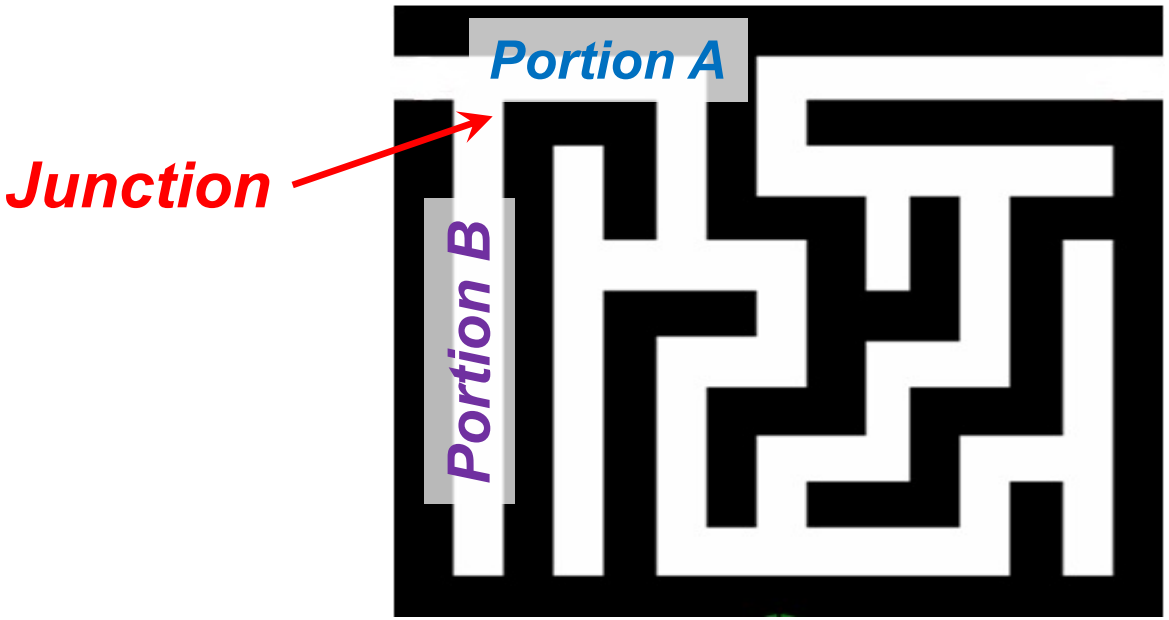
- Construct the *state-space tree*
  - nodes: partial solutions
  - edges: choices in extending partial solutions

- Explore the state space tree using depth-first search

- "Prune" *nonpromising nodes*
  - dfs stops exploring subtrees rooted at nodes that cannot lead to a solution and backtracks to such a node's parent to continue the search

# Backtracking: Idea

- Backtracking is a technique used to solve problems with a large search space, by systematically trying and eliminating possibilities.
- A standard example of backtracking would be going through a maze.
  - At some point, you might have two options of which direction to go:



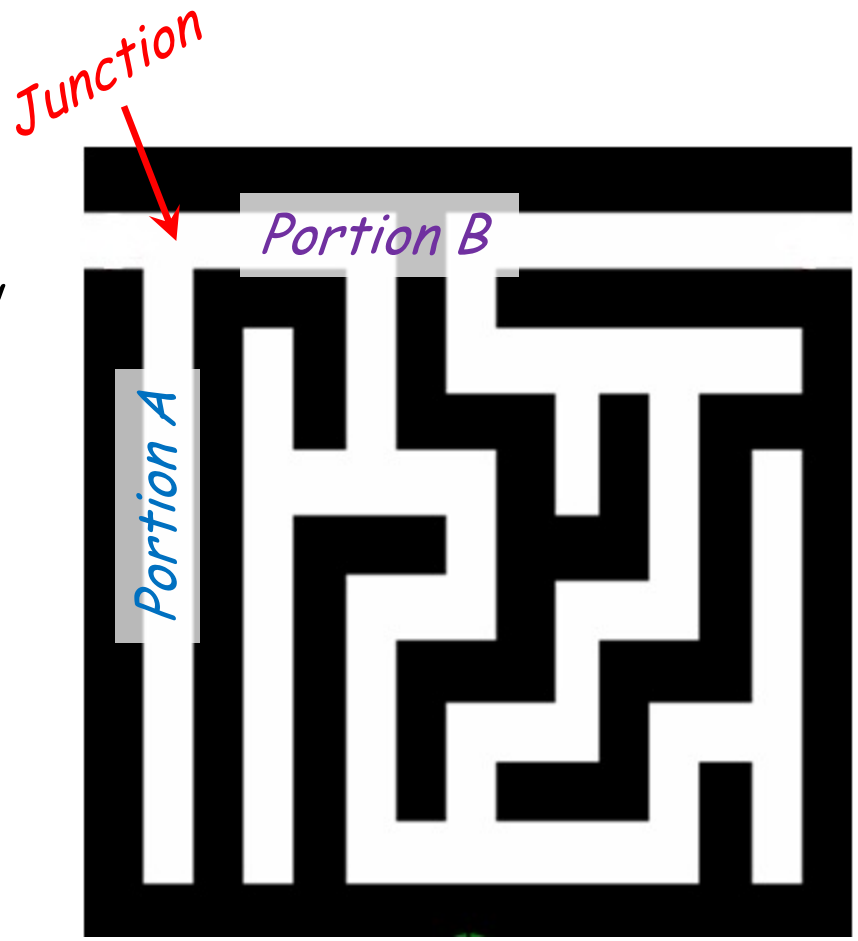
# Backtracking

One strategy would be to try going through Portion A of the maze.

If you get stuck before you find your way out, then you *"backtrack"* to the junction.

At this point in time you know that Portion A will *NOT* lead you out of the maze,

so you then start searching in Portion B

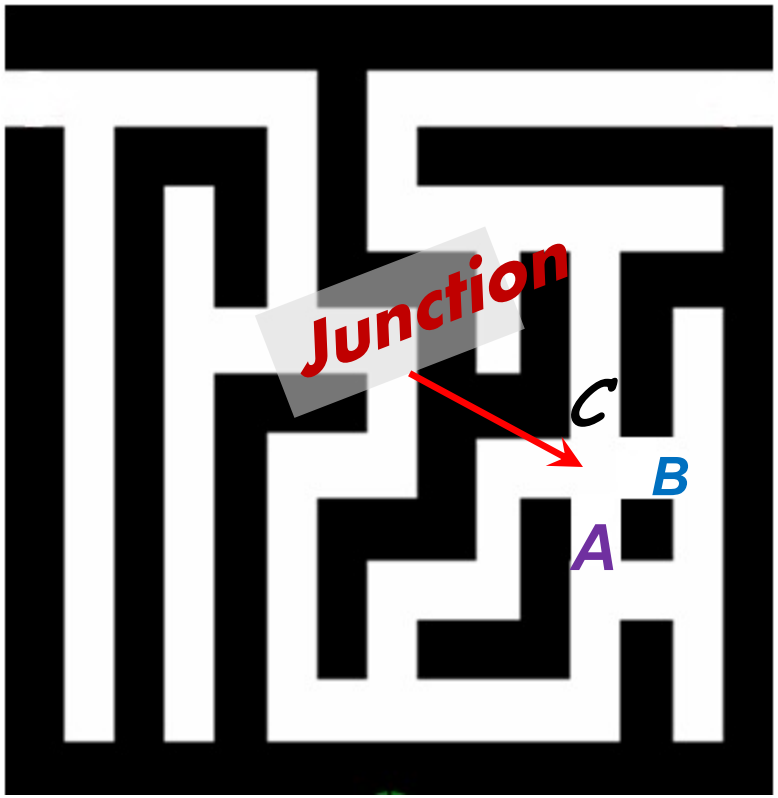


# Backtracking

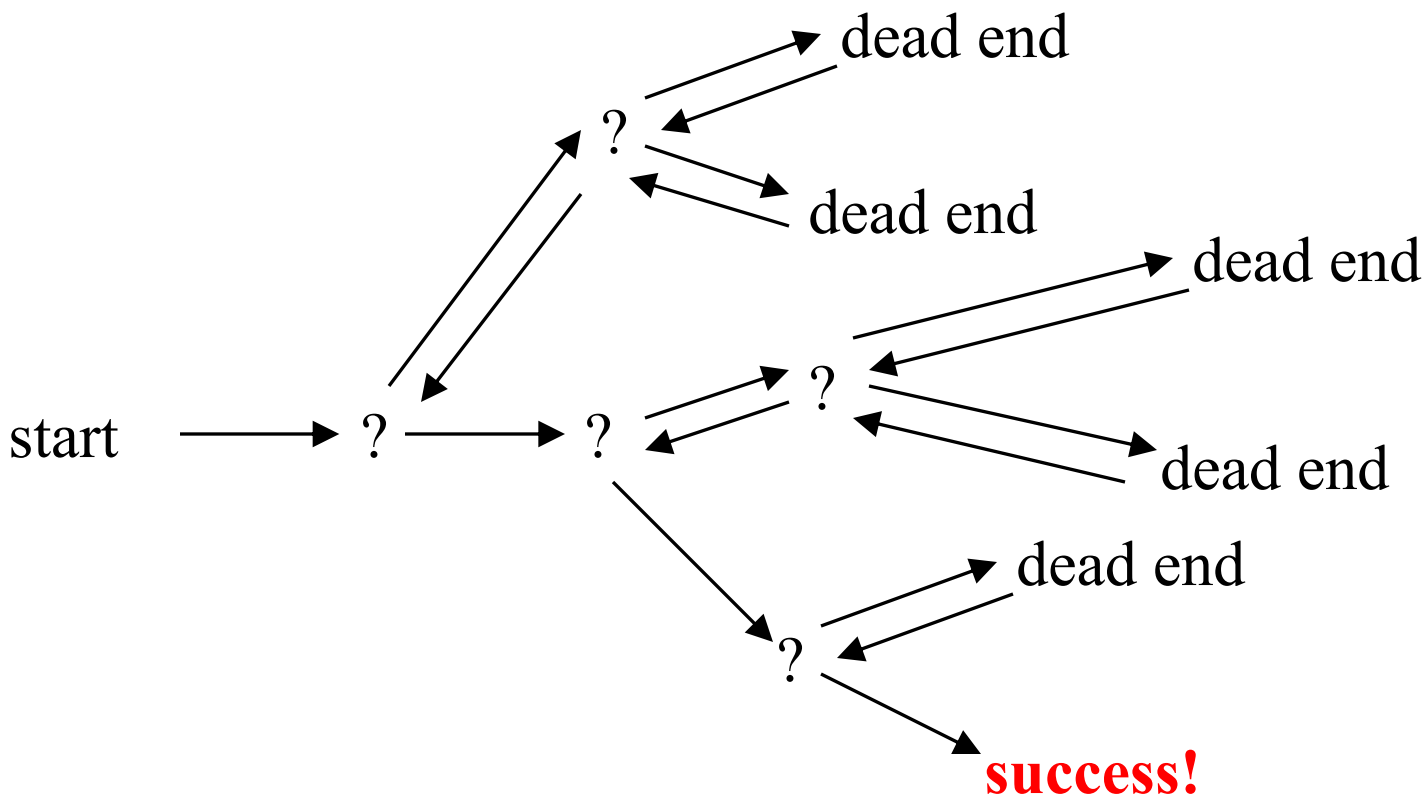
- Clearly, at a single junction you could have even more than 2 choices.

- The backtracking strategy says to try each choice, one after the other,
  - if you ever get stuck, *"backtrack"* to the junction and try the next choice.

- If you try all choices and never found a way out, then there IS no solution to the maze.



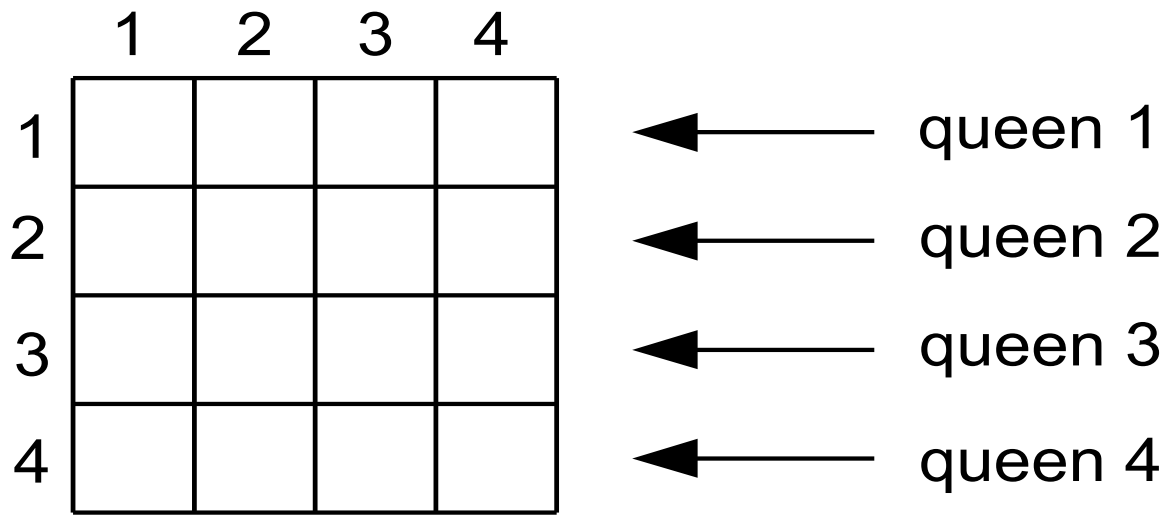
# Backtracking (animation)



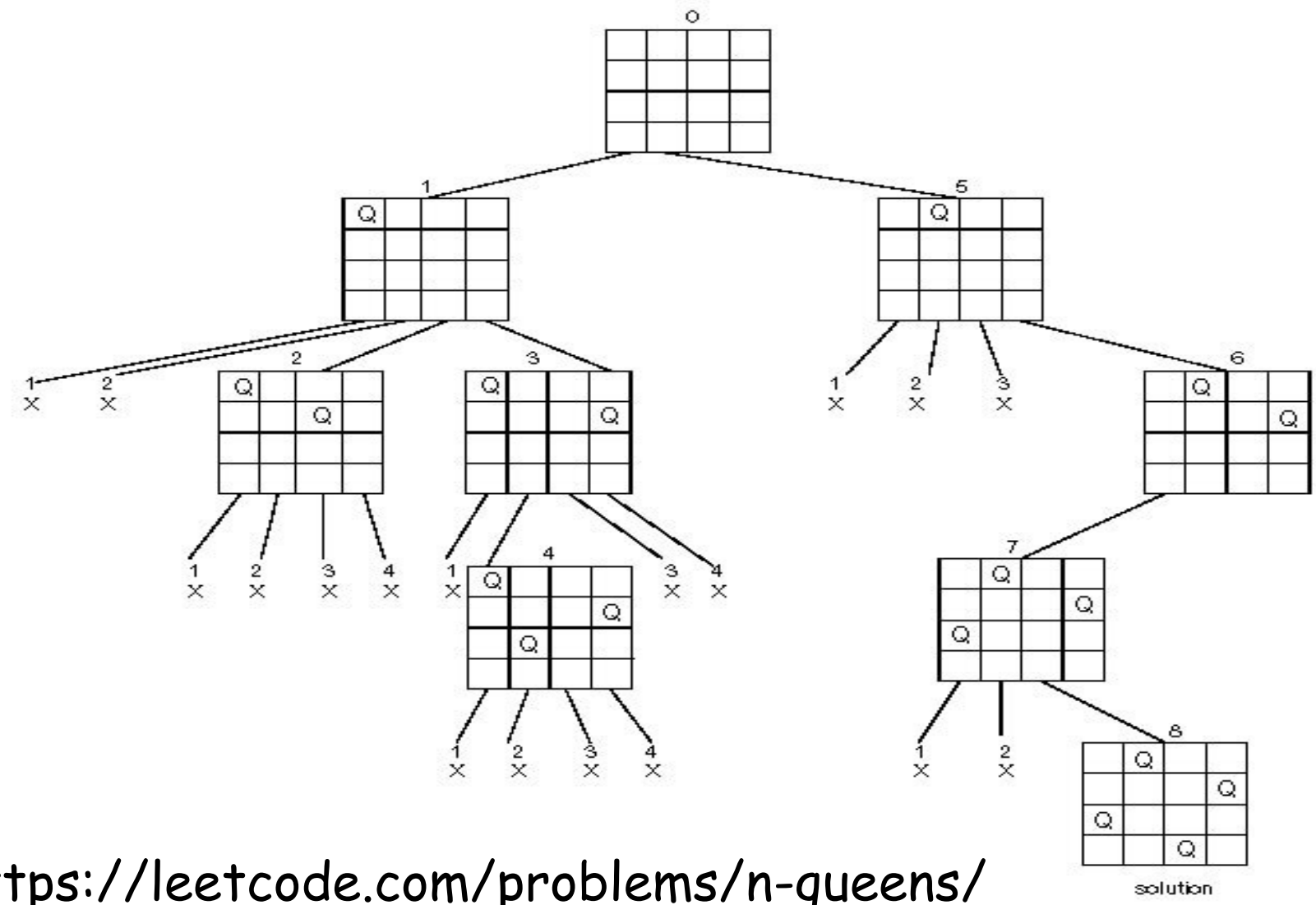
# Example: *n*-Queens Problem

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

http://en.wikipedia.org/wiki/Eight_queens_puzzle

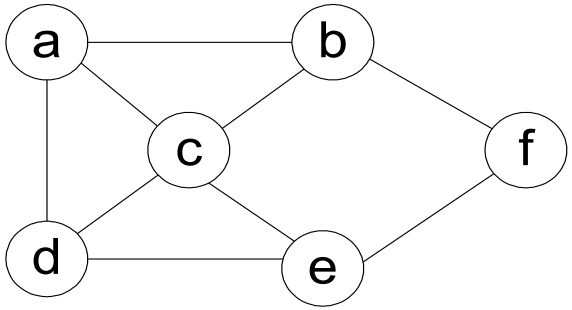


# State-Space Tree of the 4-Queens Problem



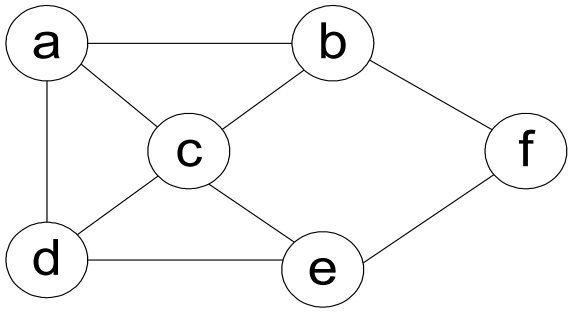

https://leetcode.com/problems/n-queens/

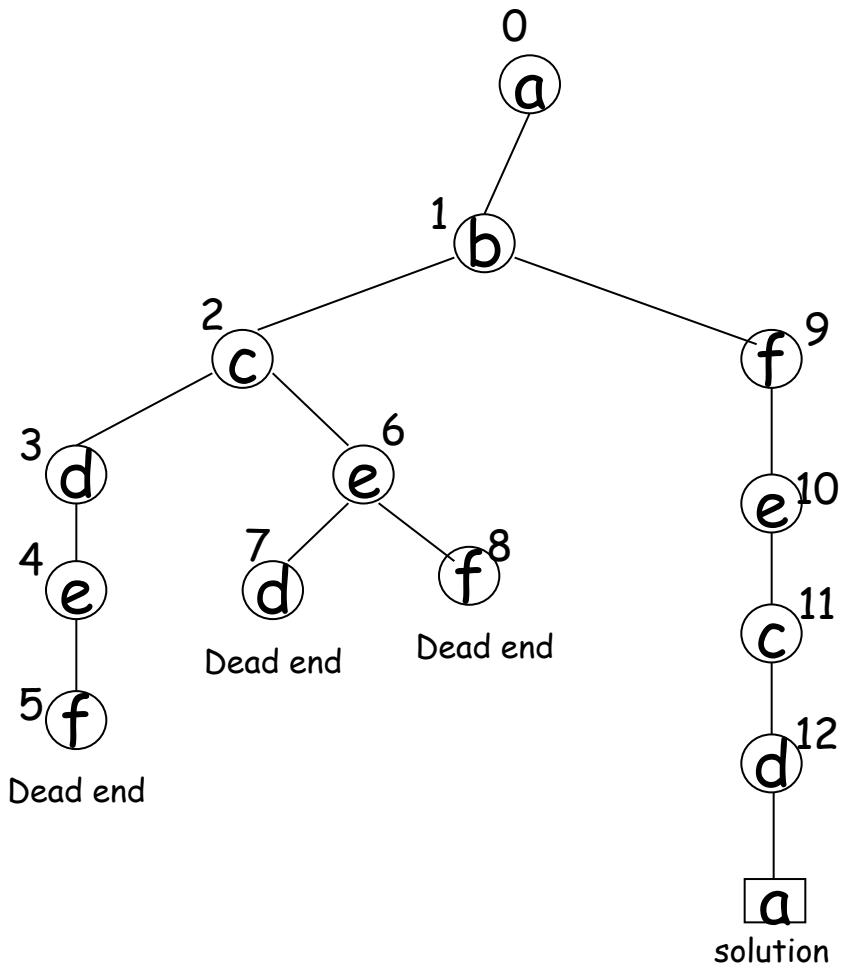
# Exercise: Hamiltonian Circuit Problem




- What is the state-space?
- What is the relationship between the states?

# Exercise: Hamiltonian Circuit Problem



# Example: Subset-Sum Problem

Find a subset of a given set $S=\{s_1, \ldots, s_n\}$ of n positive integers whose sum is equal to a give positive integer $d$.

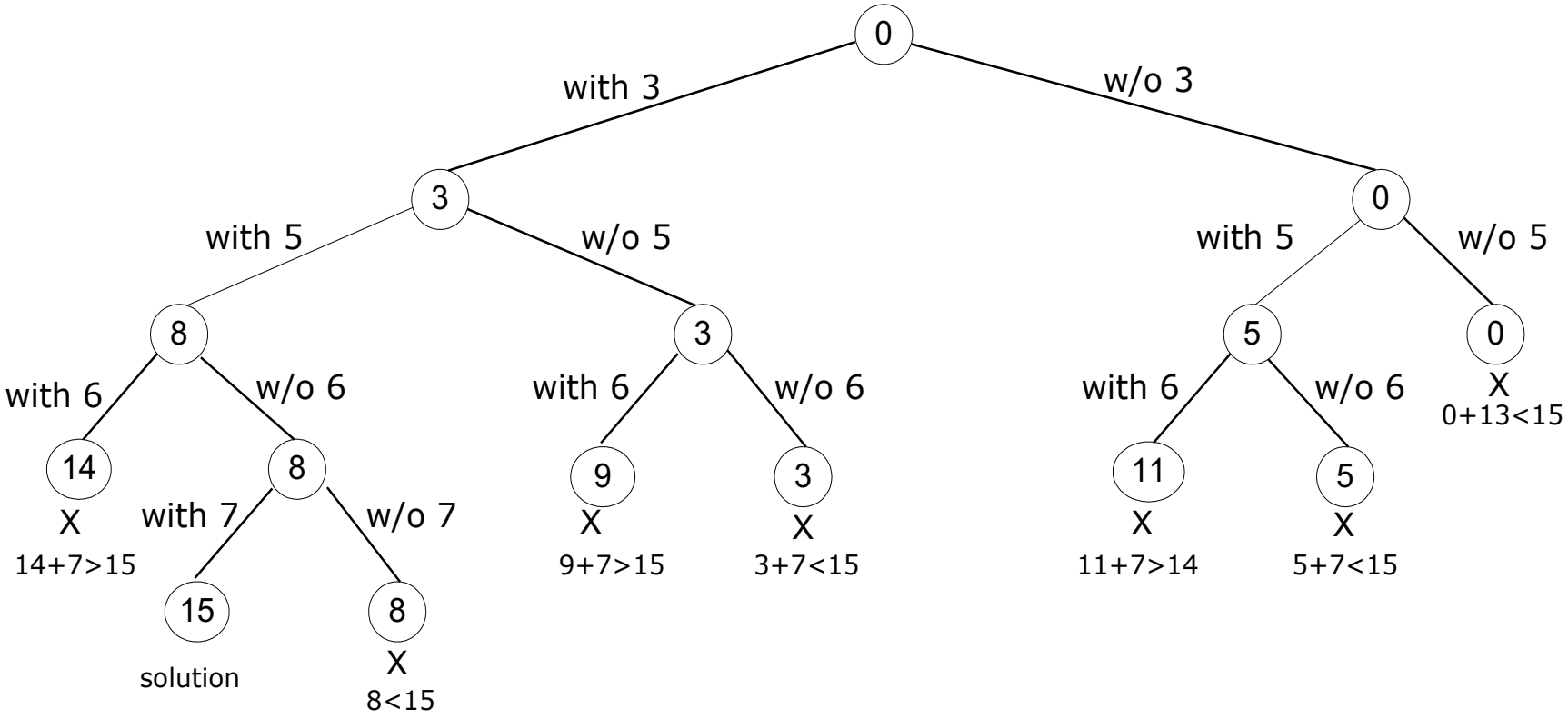
*For example: S ={ 3, 5, 6, 7}  d=15*

- What is the state-space?

- What is the relationship between the states?

# Example: Subset-Sum Problem

$S=\{3, 5, 6, 7\}$
$d=15$



# Comments on Backtracking

- Typically, backtrack is applied to difficult combinatorial problems for which no efficient algorithms for finding exact solutions possibly exist.

- Unlike exhaustive search approach, which is doomed to be extremely slow for all instances of a problem, backtrack at least holds a hope for solving some instances of nontrivial size in an acceptable amount of time.

# Branch-and-Bound

- An enhancement of backtracking
- Applicable to optimization problems
- 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)
- Uses the bound for:
  - ruling out certain nodes as "nonpromising" to prune the tree – if a node's bound is not better than the best solution seen so far
  - guiding the search through state-space

# Example: Assignment Problem

**Select one element in each row of the cost matrix *C* so that:**
- **no two selected elements are in the same column**
- **the sum is minimized**

**Example**

| | Job 1 | Job 2 | Job 3 | Job 4 |
|---|---|---|---|---|
| Person *a* | 9 | 2 | 7 | 8 |
| Person *b* | 6 | 4 | 3 | 7 |
| Person *c* | 5 | 8 | 1 | 8 |
| Person *d* | 7 | 6 | 9 | 4 |

**<u>Lower bound</u>: Any solution to this problem will have total cost at least: 2 + 3 + 1 + 4 (taking smallest value from row)**

**(or 5 + 2 + 1 + 4, taking smallest value from column)**

# Example: First two levels of the state-space tree

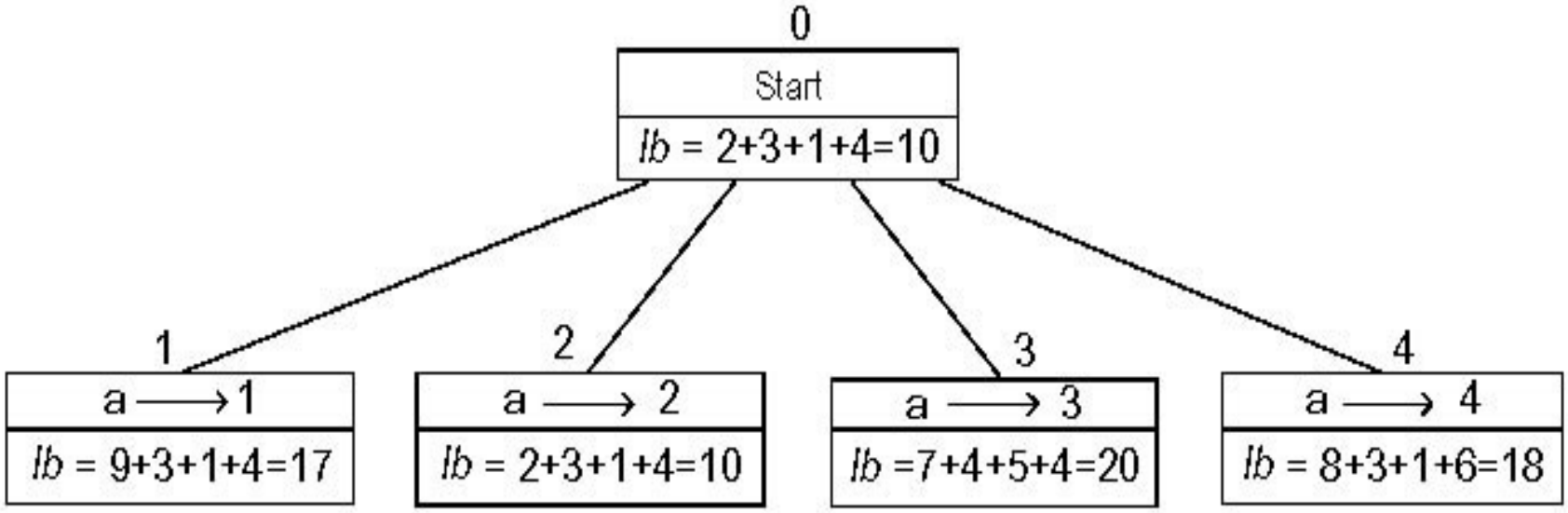



**Figure 11.5** Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person $a$ and the lower bound value, $lb$, for this node.

# Example (cont.)

| | Job 1 | Job 2 | Job 3 | Job 4 |
|---|---|---|---|---|
| Person *a* | 9 | 2 | 7 | 8 |
| Person *b* | 6 | 4 | 3 | 7 |
| Person *c* | 5 | 8 | 1 | 8 |
| Person *d* | 7 | 6 | 9 | 4 |

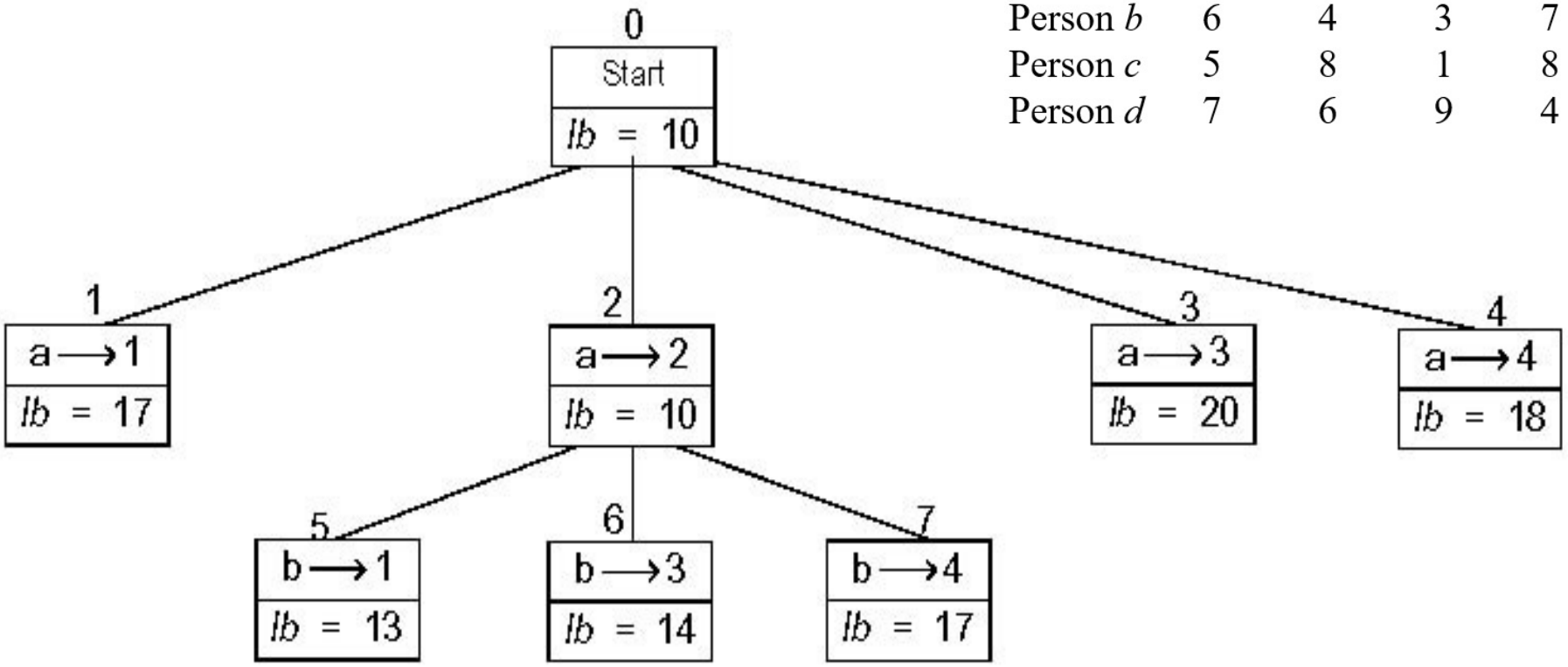
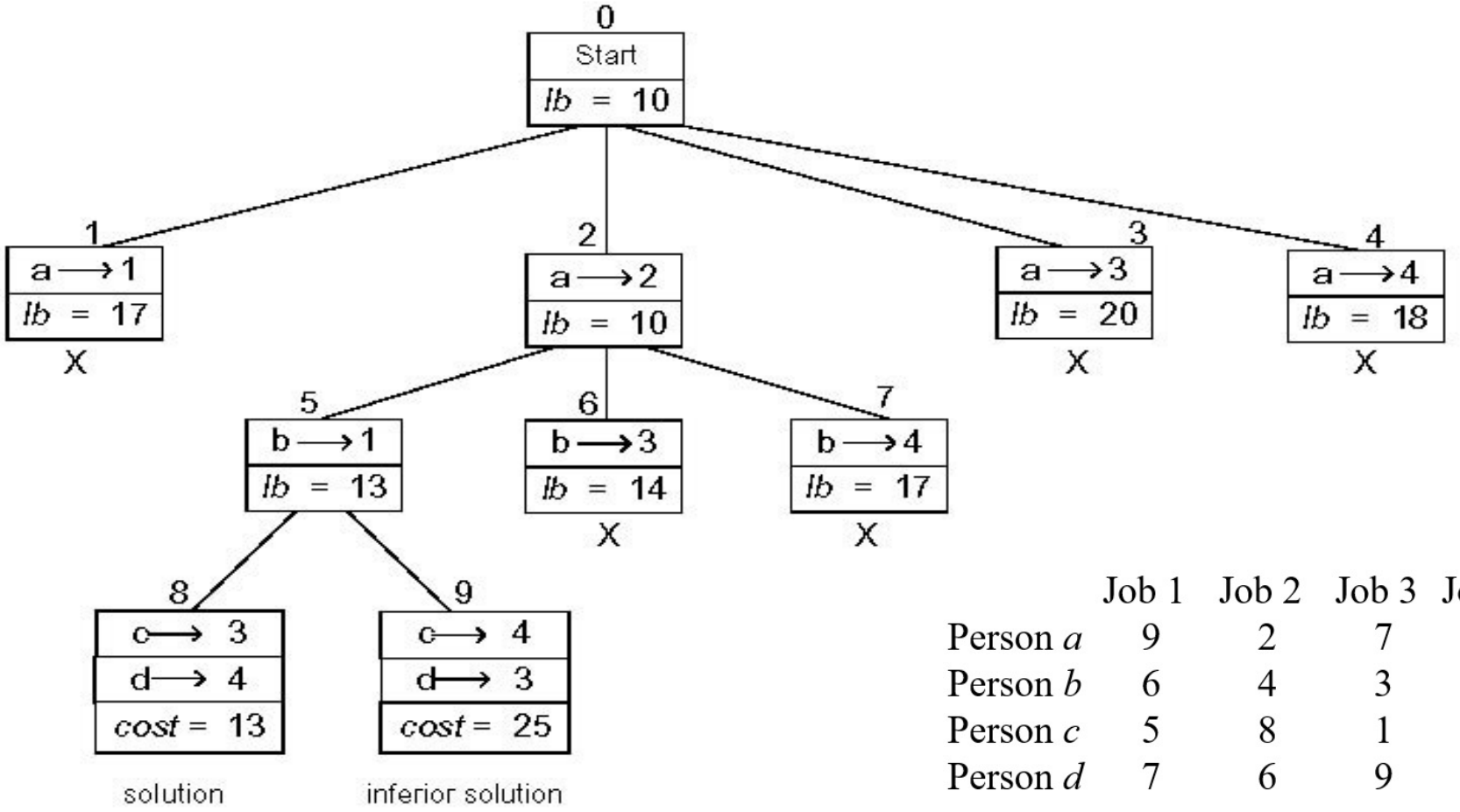



**Figure 11.6** Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm

# Example: Complete state-space tree



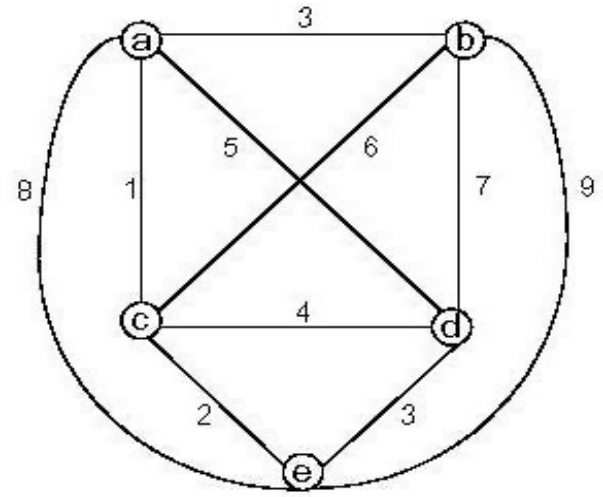

| | Job 1 | Job 2 | Job 3 | Job 4 |
|---|---|---|---|---|
| Person *a* | 9 | 2 | 7 | 8 |
| Person *b* | 6 | 4 | 3 | 7 |
| Person *c* | 5 | 8 | 1 | 8 |
| Person *d* | 7 | 6 | 9 | 4 |

**Figure 11.7** Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm

# The traveling salesperson optimization problem

- Given a graph, the TSP Optimization problem is to find a tour, starting from any vertex, visiting every other vertex and returning to the starting vertex, with minimal cost.
- It is NP-complete
- We try to avoid n! exhaustive search by the branch-and-bound technique.

# Example: Traveling Salesman Problem

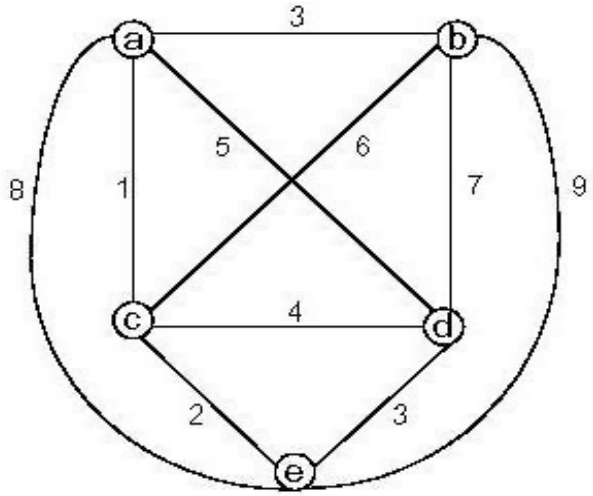



| | a | b | c | d | e |
|---|---|---|---|---|---|
| a | 0 | 3 | 1 | 5 | 8 |
| b | 3 | 0 | 6 | 7 | 9 |
| c | 1 | 6 | 0 | 4 | 2 |
| d | 5 | 7 | 4 | 0 | 3 |
| e | 8 | 9 | 2 | 3 | 0 |

- What is the state-space?
- What is the relationship between the states?
- What is a reasonable lower bound? (Using distance matrix)

lb = Σ[ (distances of two nearest cities of v in G)/2]

# Example: Traveling Salesman Problem



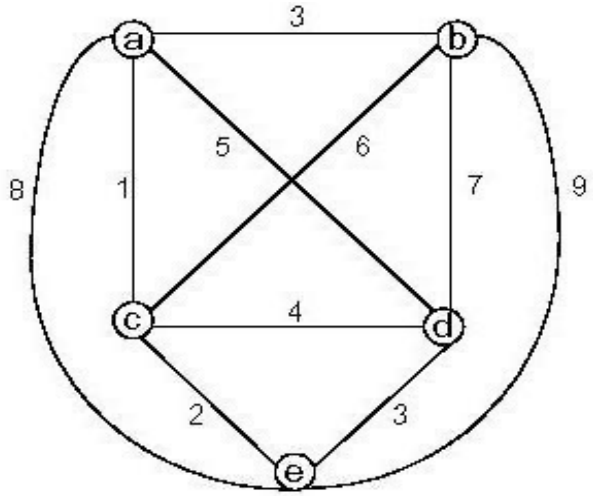

2+4.5+1.5+3.5+2.5

| | a | b | c | d | e |
|---|---|---|---|---|---|
| a | 0 | 3 | 1 | 5 | 8 |
| b | 3 | 0 | 6 | 7 | 9 |
| c | 1 | 6 | 0 | 4 | 2 |
| d | 5 | 7 | 4 | 0 | 3 |
| e | 8 | 9 | 2 | 3 | 0 |




By requiring *b* before *C*, we can reduce space by half

# Example: Traveling Salesman Problem




| | a | b | c | d | e |
|---|---|---|---|---|---|
| a | 0 | 3 | 1 | 5 | 8 |
| b | 3 | 0 | 6 | 7 | 9 |
| c | 1 | 6 | 0 | 4 | 2 |
| d | 5 | 7 | 4 | 0 | 3 |
| e | 8 | 9 | 2 | 3 | 0 |




https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/#:~:text=Given%20a%20set%20of%20cities,in%20figure%20on%20right%20side.x