Backtracking and Branch-and-Bound

CS3230: Design and Analysis of Algorithms

Roger Zimmermann

National University of Singapore

Spring 2017

Chapter 12: Limits of Algorithms

- This chapter focuses on techniques to solve very difficult problems.
- There are two principal approaches to tackle intractable problems:
  1. Use a strategy that guarantees solving the problem exactly but does not guarantee to find a solution in polynomial time, or
  2. Use an approximation algorithm that can find an approximate (sub-optimal) solution in polynomial time.

Exact Solutions

- Exhaustive search (brute force):
  - Generate all candidate solutions and identify one with a desired property.
  - This approach is useful only for small instances.

- Improvement over exhaustive search: backtracking and branch-and-bound.

Backtracking and Branch-and-Bound

- The idea:
  - Construct candidate solutions one component at a time based on a certain rule.
  - If no potential values of the remaining components can lead to a solution, the remaining components are not generated at all.
- This approach makes it possible to solve some large instances of difficult combinatorial problems.
- However, in the worst case the complexity is still exponential.
- Construction of a state-space tree whose nodes reflect specific choices made for a solution's components.

Backtracking and Branch-and-Bound

- Difference between BT and B&B:
  1. Apply to different problems.
     - Backtracking: applied more often to non-optimization problems.
     - Branch-and-bound: applied only to optimization problems.
  2. The way a new component is generated.
     - Backtracking: usually developed depth first (similar to DFS).
     - Branch-and-bound: best-first rule (most natural).

Backtracking and Branch-and-Bound

- Advantages and disadvantages:
  - + Cut down on the search space.
  - + Provide fast solutions for some instances.
  - – The worst case is still exponential.

Backtracking

- Construct the state-space tree:
  - Root represents an initial state.
  - Nodes reflect specific choices made for a solution's components (partial solution).
    - Promising and non-promising nodes.
    - Leaves $\Rightarrow$ possibly solutions.
- Explore the state space tree using depth-first search.
- "Prune" non-promising nodes:
  - DFS stops exploring sub-trees rooted at nodes leading to no solutions, and ...
  - "backtracks" to its parent node.

Backtracking

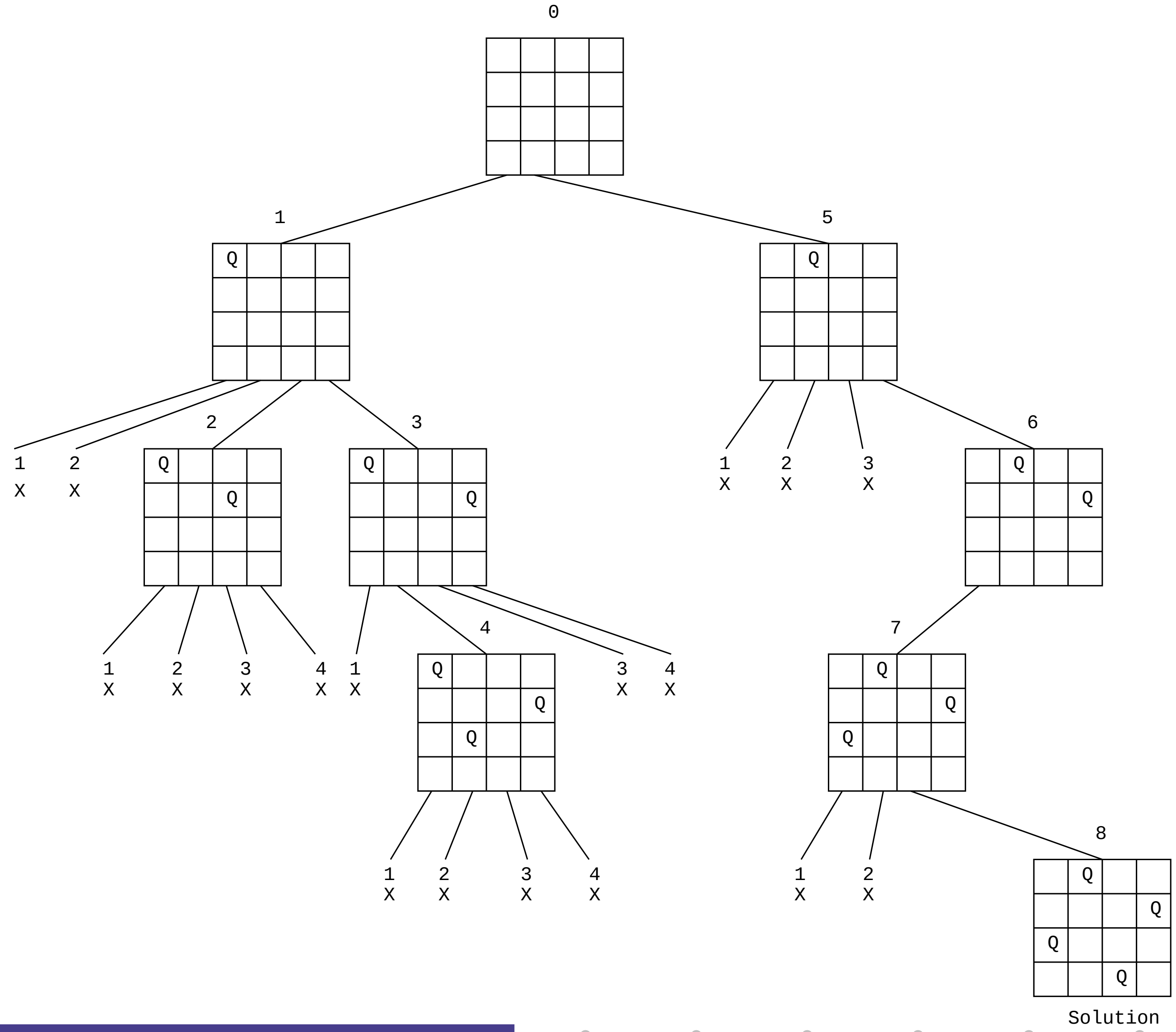
- Promising node:
  - Represents a partially constructed solution that can be further developed without violating the problem's constraints.
  - May still lead to a complete solution.
- Non-promising node:
  - Represents a partially constructed solution where there is no legitimate option for the next component.
  - No alternatives for any remaining component need to be considered.

Example: The n-Queens Problem

- Place n queens on an n-by-n chess board such that no two of them ‘attack’ each other (i.e., are in the same row, column or diagonal).
- Ex.: $n = 4$:

<table>
<tr><td></td><td>Q</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td>Q</td></tr>
<tr><td>Q</td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td>Q</td><td></td></tr>
</table>

State-Space Tree of 4-Queens Problem



Exercises

- Continue the backtracking search for a solution to the four-queens problem to find the second solution to the problem.
  - Hint: the board is symmetric, obtain another solution by reflections.
- Get a solution to the 5-queens problem found by the back-tracking algorithm?
- Can you (quickly) find at least 3 other solutions?

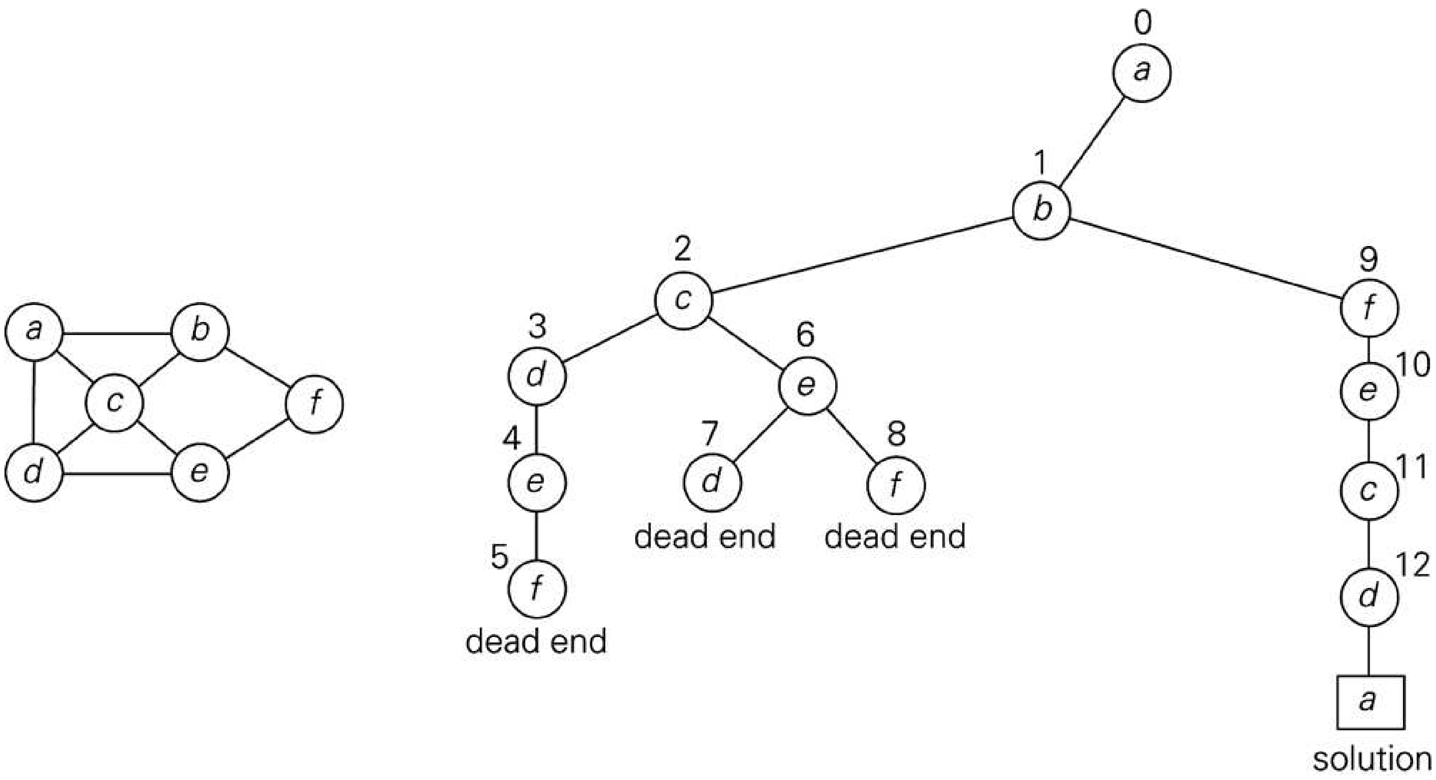
Hamiltonian Circuit Problem

- Hamiltonian Circuit:
  Determine whether a given graph has a path that starts and ends at the same vertex and passes through all the other vertices exactly once.

- Traveling Salesman:
  Find the shortest Hamiltonian circuit in a complete graph with positive integer weights.

Hamiltonian Circuit Problem

- Without loss of generality, start at vertex a.



(a) (b)

Subset-Sum Problem

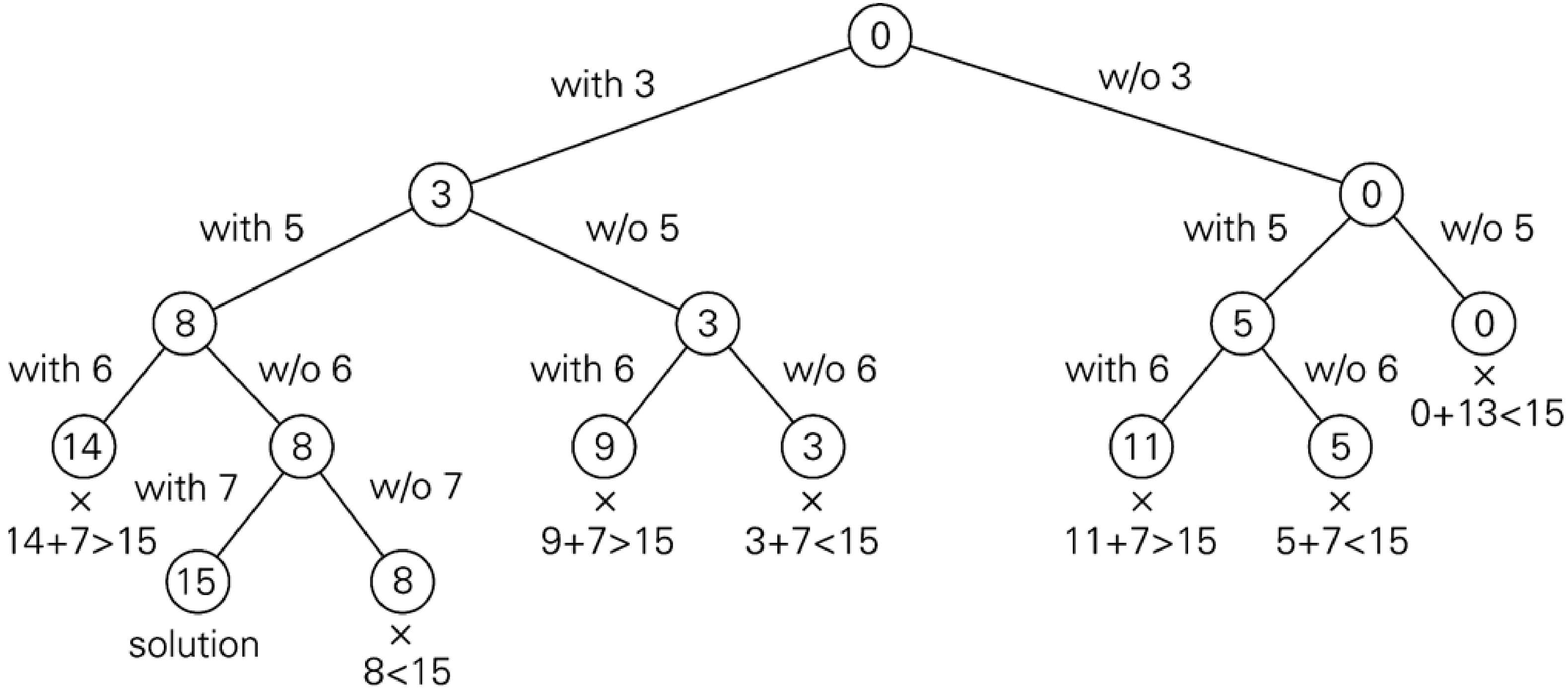
- The problem: find a subset of a given set $S = \{s_1, s_2, \ldots, s_n\}$ of n positive integers whose sum is equal to a given positive integer d. (Ex.: $S = \{3, 5, 6, 7\}$, $d = 15$.)
- We record the value of s', the sum of the numbers, in the node.
- If s' is equal to d then we have a solution.
- If s' is not equal to d, we can terminate the node as nonpromising if either of the following two inequalities holds:

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

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

Subset-Sum Problem

- The problem: 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. (Ex.: $S = \{3, 5, 6, 7\}$, $d = 15$.)



Example: Subset-Sum Problem

- Mapping logical disks to physical disks.

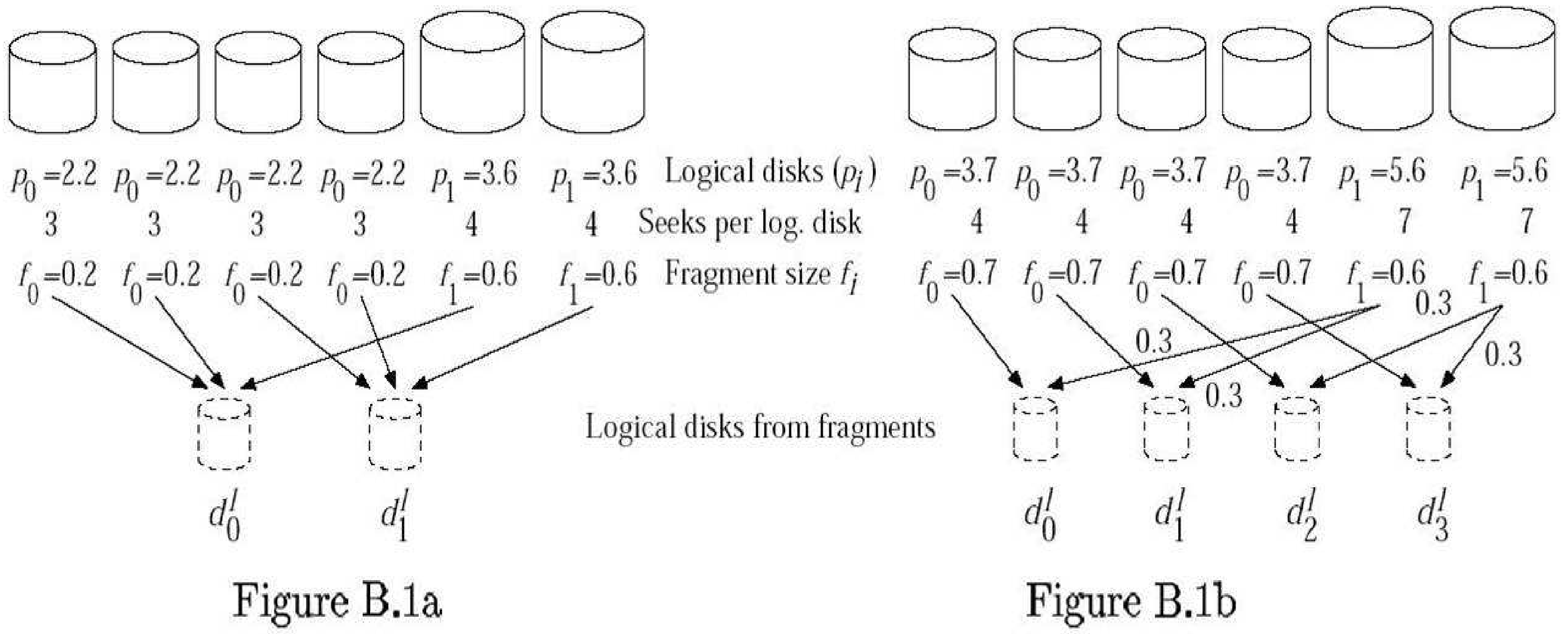


Figure B.1: Two possible configurations with Disk Merging.

General Backtracking Algorithm

- Output: n-tuple $(x_1, x_2, \ldots, x_n)$.
- Ex.: n-queens problem, each S_i is the set of integers (column numbers) 1 through n.

  ALGORITHM $Backtrack$(X[1..i])
  // Template of a generic backtracking algorithm
  // In: X[1..i] first i promising components of a solution
  // Out: the tuples representing the problem's solutions
  if X[1..i] is a solution write X[1..i] // Tuple is a solution.
  else
      for each element $x \in S_{i+1}$ consistent with X[1..i] and the problem constraints do
          X[1..i] $\leftarrow x$
          $Backtrack$(X[1..$i+1$])

General Remarks

- Exhaustive search versus backtracking.
  - Exhaustive search is guaranteed to be very slow in every problem instance.
  - Backtracking provides the hope to solve some problem instances of non-trivial sizes by pruning non-promising branches of the state-space tree.
- The success of backtracking varies from problem to problem and from instance to instance.
  - Backtracking possibly generates all possible candidates in an exponentially growing state-space tree.
  - But still it provides a systematic technique to do so.

Branch and Bound

- An enhancement of backtracking.
  - Similarity to BT.
    - A state-space tree is used to solve a problem.
  - Difference to BT.
    - The branch-and-bound algorithm does not limit us to any particular way of traversing the tree and is used only for optimization problems.
    - The backtracking algorithm requires the use of DFS traversal and is used for non-optimization problems (“Is it possible to find a solution continuing this path: yes/no”).

Branch and Bound

- Terminology:
  - Objective function: maximize or minimize a problem's "value".
  - Feasible solution: a point in the problem's search space that satisfies all the problem's constraints.
  - Optimal solution: a feasible solution with the best value for the objective function.

Branch and Bound

- The idea: Set up a bounding function, which is used to compute a bound (for the value of the objective function) at a node of a state-space tree and determine if it is promising.
  - Promising: if the bound is better than the value of the best solution so far, expand beyond the node.
  - Non-promising: if the bound is no better than the value of the best solution so far, do not expand beyond the node (pruning the state-space tree).

Ex.: Knapsack Problem

- Given n items of known weights w_i and values $v_i, i = 1, 2, \ldots, n$, and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.
  - Compute value-to-weight ratios and order by best payoff per weight to worst payoff per weight: $v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_n/w_n$.
  - Binary state-space tree: go left $\Rightarrow$ include next item i, go right $\Rightarrow$ exclude next item i.
  - Record the total weight w and the total value v at each node.
  - Upper bound: $ub = v + (W - w)(v_{i+1}/w_{i+1})$.

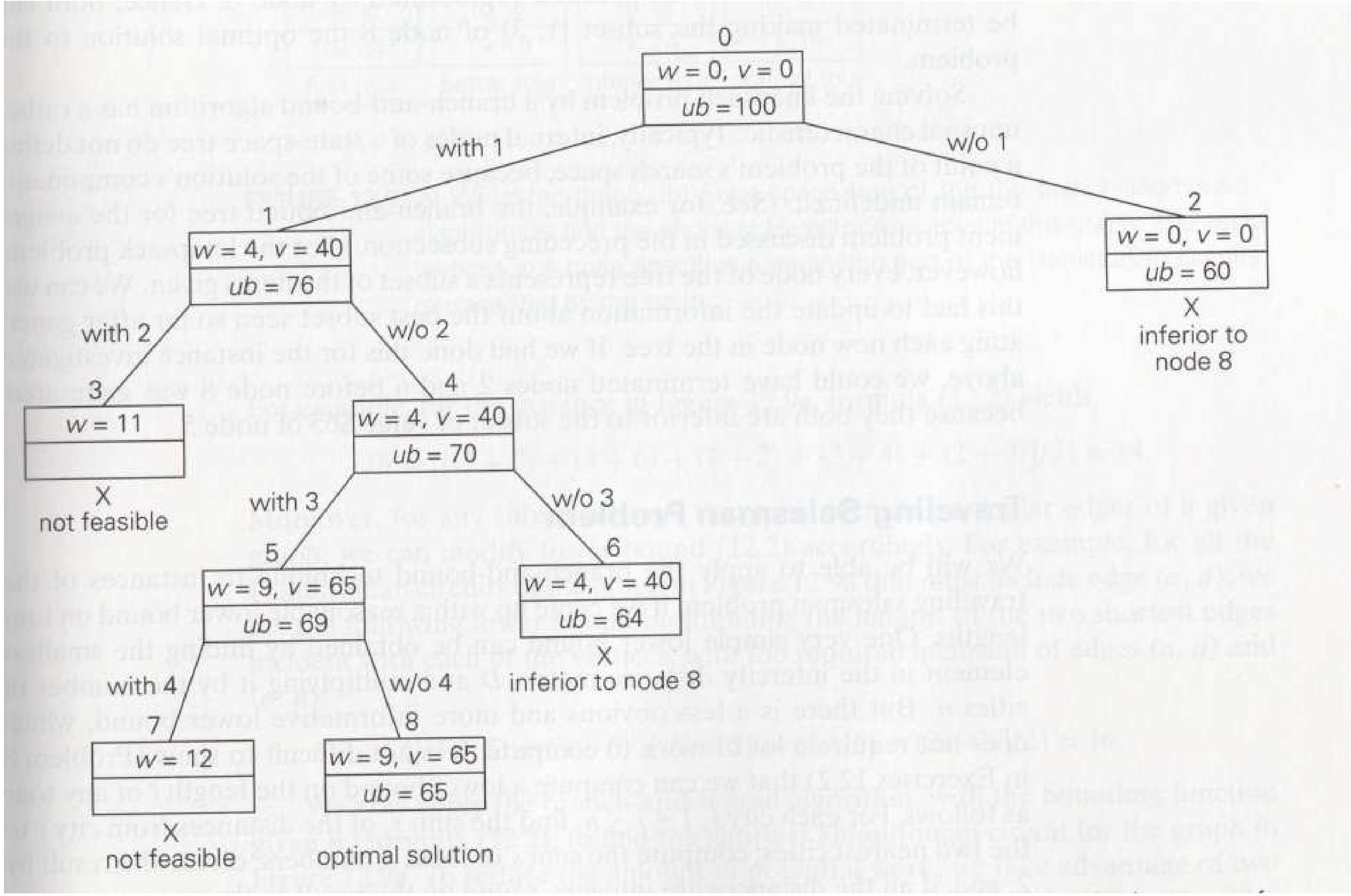
Q: Why is ub an upper bound?

Ex.: Knapsack Problem

- The knapsack's capacity W is 10.

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

Ex.: Knapsack Problem



Traveling Salesman Problem

- An obvious way to construct the TSP state-space tree:
  - A node: a node in the state-space tree; a vertex: a vertex in the graph.
  - A node that is not a leaf represents all the tours that start with the path stored at that node; each leaf represents a tour (or non-promising node).
  - Branch-and-bound: we need to determine a lower bound for each node.
    - For example, to determine a lower bound for node [1, 2] means to determine a lower bound on the length of any tour that starts with edge 1—2.

Traveling Salesman Problem

- Expand each promising node, and stop when all the promising nodes have been expanded. During this procedure, prune all the non-promising nodes.
  - Promising node: the node's lower bound is less than current minimum tour length.
  - Non-promising node: the node's lower bound is no less than current minimum tour length.

TSP – Bounding Function 1

- Because a tour must leave every vertex exactly once, a lower bound on the length of a tour is the sum of the minimum cost of leaving every vertex.
  - The lower bound on the cost of leaving vertex v_1 is given by the minimum of all the nonzero entries in row 1 of the adjacency matrix.
  - $\dots$
  - The lower bound on the cost of leaving vertex v_n is given by the minimum of all the nonzero entries in row n of the adjacency matrix.
- Note: This is not to say that there is a tour with this length. Rather, it says that there can be no shorter tour.
- Assume that the tour starts with v_1.

TSP – Bounding Function 2

- Because every vertex must be entered and exited exactly once, a lower bound on the length of a tour is the sum of the minimum cost of entering and leaving every vertex.
  - For a given edge (u, v), think of half of its weight as the exiting cost of u, and half of its weight as the entering cost of v.
  - The total length of a tour equals the total cost of visiting (entering and exiting) every vertex exactly once.

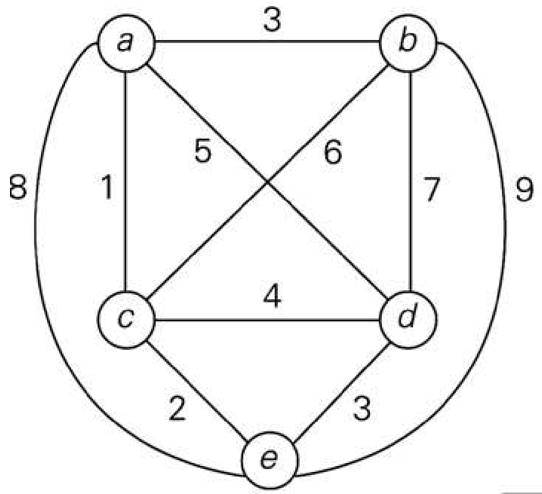
TSP – Bounding Function 2

– The lower bound of the length of a tour equals the lower bound of the total cost of visiting (entering and exiting) every vertex exactly once.
  – For each vertex, pick the top two shortest adjacent edges (their sum divided by 2 is the lower bound of the total cost of entering and exiting the vertex).
  – Add up these summations for all the vertices.

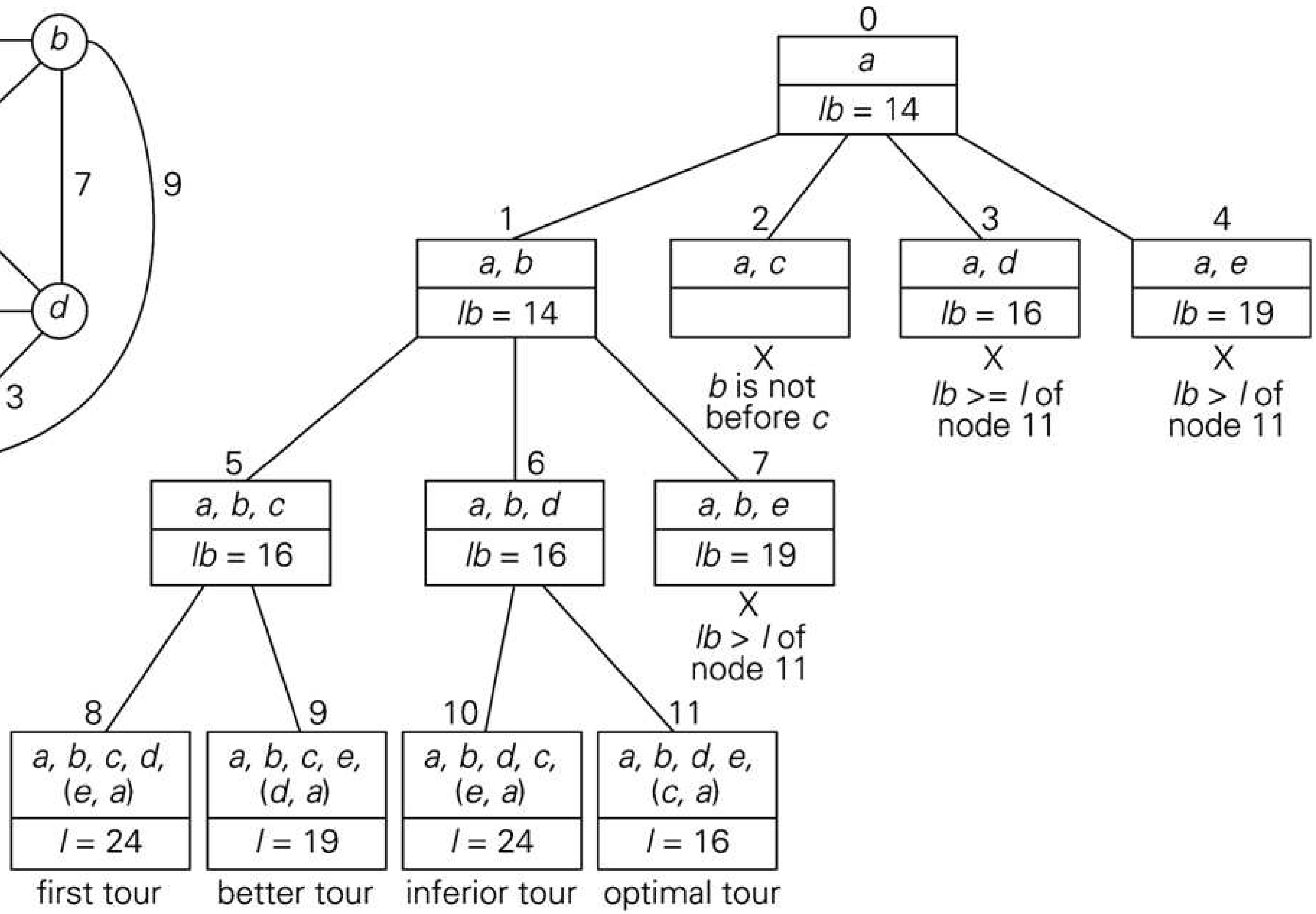
TSP – Bounding Functions

- Ex. Bounding Function 2: $lb = \lceil s/2 \rceil$.
  $lb = \lceil [(1+3)+(3+6)+(1+2)+(3+4)+(2+3))]/2 \rceil = 14$.

TSP Illustration



(a)



Approximation Algorithms

- The decision versions of certain combinatorial problems are *NP*-complete.
- The optimization versions of these problems are *NP*-hard, hence no polynomial-time algorithms are known.
- What to do if such a problem is of practical importance?
- Recall: good performance for branch-and-bound algorithms cannot be guaranteed.
- Different approach: solve the problem approximately, but fast.

Approximation Algorithms

- Why may an approximation be appealing:
  - Sometimes a good (but not necessary optimal) solution will suffice.
  - In practice the input data may be inaccurate.
- Approximation algorithms are based on problem-specific heuristics.
- A heuristic is a rule drawn from experience, rather than a mathematically proved assertion.
- Ex.: "*Go to the nearest city in the TSP problem.*"

Approximation Algorithms

- If we use an approximation algorithm we would like to know: how accurate is it?
- Relative error:

$$re(s_a) = \frac{f(s_a) - f(s^*)}{f(s^*)}$$

  where s^* is an exact solution to the problem.
- Accuracy ratio:

$$r(s_a) = \frac{f(s_a)}{f(s^*)} \quad \text{or} \quad r(s_a) = \frac{f(s^*)}{f(s_a)}$$

  (Ratio ≥ 1 for both minimization (left) and maximization (right) problems.)

Approximation Algorithms

- Typically $f(s^*)$ is unknown (the optimal value of the objective function).
- Hence, compute a good upper bound on the value of $f(s_a)$.

- DEF.: A polynomial time approximation algorithm is said to be a c-approximation algorithm, where $c \geq 1$, if the accuracy ratio of the approximation it produces does not exceed c for any instance of the problem in question: $r(s_a) \leq c$.

- The smallest value of c is called the performance ratio, denoted R_A. We would like it to be as close to 1 as possible.

Ex.: Traveling Salesman Problem

- Note: there exists no c-approximation algorithm for the general TSP with a finite performance ratio (i.e., in this case $R_A = \infty$).

Ex. 1: Traveling Salesman Problem

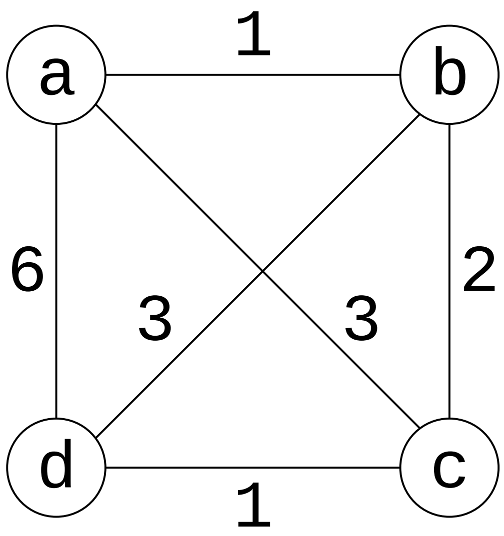
- Approx. Algorithm 1 for TSP: Nearest-neighbor algorithm.
- Greedy algorithm based on nearest-neighbor heuristic.

Step 1. Choose an arbitrary city as a start.

Step 2. Repeat the following operation until all the cities have been visited: go to the unvisited city nearest the one visited last (ties can be broken arbitrarily).

Step 3. Return to the starting city.

Ex. 1: Traveling Salesman Problem



- Nearest-neighbor algorithm $s_a : a - b - c - d$ of length 10.
- Optimal solution $s^* : a - b - d - c$ of length 8. Thus $r(s_a) = 1.25$.

Ex. 1: Traveling Salesman Problem

- Problem: it may force us to traverse a very large edge on the last leg of the tour: $R_A = \infty$.
- Example: replace edge (a, d) with w

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{4 + w}{8}$$

Ex. 2: Traveling Salesman Problem

- Approx. Algorithm 2 for TSP: Multifragment-heuristic algorithm.
- Focus on edges, rather than vertices.

Step 1. Sort the edges in increasing order of their weight. (Ties can be broken arbitrarily.) Initialize the set of tour edges to be constructed to the empty set.

Step 2. Repeat this step until a tour of length n is obtained, where n is the number of cities in the instance being solved; add the next edge on the sorted edge list to the set of tour edges, provided this addition does not create a vertex of degree 3 or a cycle of length less than n; otherwise skip the edge.

Step 3. Return the set of tour edges.

Ex. 2: Traveling Salesman Problem

- Approx. Algorithm 2 for TSP: Multifragment-heuristic algorithm.
  - In the example, yields: $\{(a,b),(c,d),(b,c),(a,d)\}$ of length 10.
  - However, in general produces better results than the nearest-neighbor algorithm.
  - Performance ratio is unbounded.

Euclidean TSP

- An important subset of instances of the Traveling Salesman Problem are called Euclidean.
- Euclidean intercity distances satisfy the following natural conditions:
  - Triangle inequality
    $d[i, j] \leq d[i, k] + d[k, j]$ for any triple of cities i, j, and k.
  - Symmetry
    $d[i, j] = d[j, i]$ for any pair of cities i and j.

Euclidean TSP

- Given Euclidean instances of the TSP, the accuracy ratio of the nearest-neighbor and the multifragment-heuristic algorithms is as follows:

$$\frac{f(s_a)}{f(s^*)} \leq \frac{1}{2}(\lceil \log_2 n \rceil + 1)$$

- $f(s_a)$: heuristic tour.
- $f(s^*)$: shortest tour.
- Performance ratio is unbounded.

Ex. 3: Traveling Salesman Problem

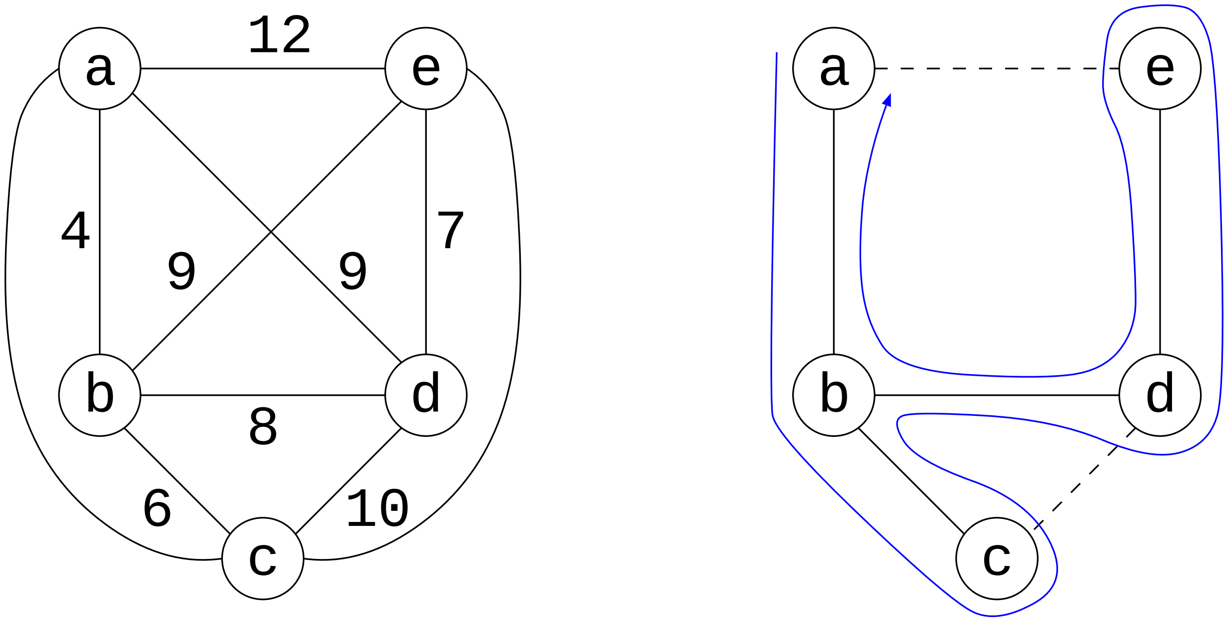
- Approx. Algorithm 3 for TSP: Minimum-spanning-tree–based algorithm (twice-around-the-tree algorithm).

Step 1. Construct a minimum spanning tree of the graph corresponding to a given instance of the traveling salesman problem.

Step 2. Starting at an arbitrary vertex, perform a walk around the minimum spanning tree recording all the vertices passed by.

Step 3. Scan the vertex list obtained in Step 2 and eliminate from it all repeated occurances of the same vertex except the starting one at the end of the list. The vertices remaining on the list will form a Hamiltonian circuit, which is the output of the algorithm.

Ex. 1: Twice-Around-the-Tree



- Left: Graph.
- Right: Walk around the minimum spanning tree with the shortcuts. Tour: $a, b, c, b, d, e, d, b, a \rightarrow a, b, c, d, e, a$.

Ex. 3: Traveling Salesman Problem

- Approx. Algorithm 3 for TSP: Minimum-spanning-tree–based algorithm (twice-around-the-tree algorithm).
- The twice-around-the-tree algorithm is a 2-approximation algorithm for the TSP with Euclidean distances.
  - Show: $f(s_a) \leq 2f(s^*)$.
  - $f(s^*) > w(T) \geq w(T^*)$ (T: spanning tree, T^*: minimum spanning tree).
  - $\Rightarrow 2f(s^*) > 2w(T^*)$.
  - The length of the walk obtained in Step 2 $\geq$ the length of the tour s_a.
  - $\Rightarrow 2f(s^*) > f(s_a)$.

TSP Held-Karp Bound

- Lower bound: Held-Karp.
- Based on linear programming. Fast to compute.
- Typically very close (< 1%) to the length of an optimal tour.
- Hence, estimate $r(s_a) = f(s_a)/f(s^*) \approx f(s_a)/HK(s^*)$.

Approximate TSP

- Average tour quality and running times for various heuristics on the 10,000-city random uniform Euclidean instances [Joh02].

Heuristic	% excess over the Held-Karp bound	Running time (seconds)
nearest neighbor	24.79	0.28
multifragment	16.42	0.20
Christofides	9.81	1.04
2-opt	4.70	1.41
3-opt	2.88	1.50
Lin-Kernighan	2.00	2.06