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

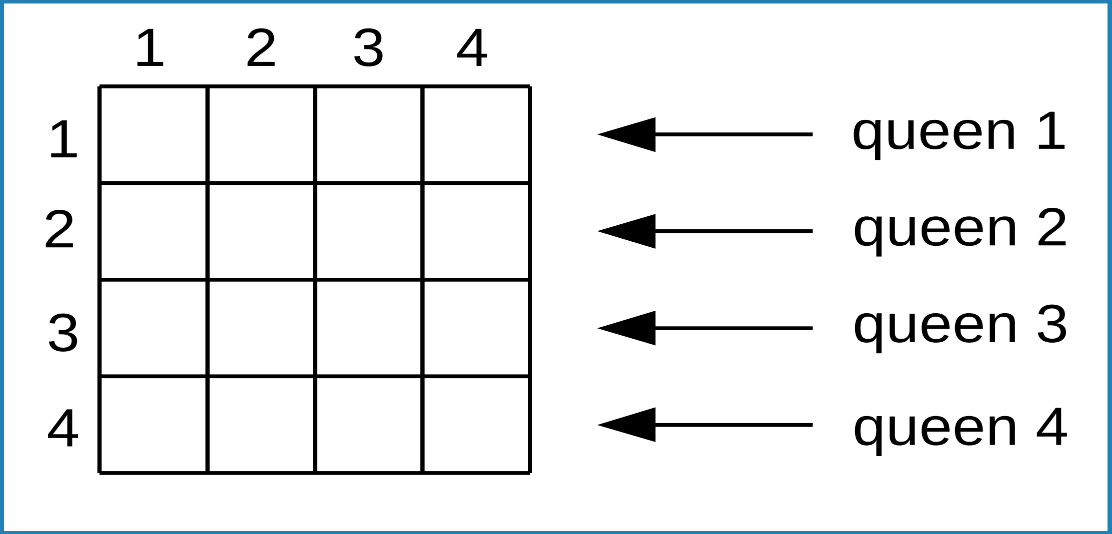
- *exhaustive search* (brute force)
  - useful only for small instances
- *dynamic programming*
  - applicable to some problems (e.g., the knapsack problem)
- *backtracking*
  - eliminates some unnecessary cases from consideration
  - yields solutions in reasonable time for many instances but worst case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

12.1 Backtracking

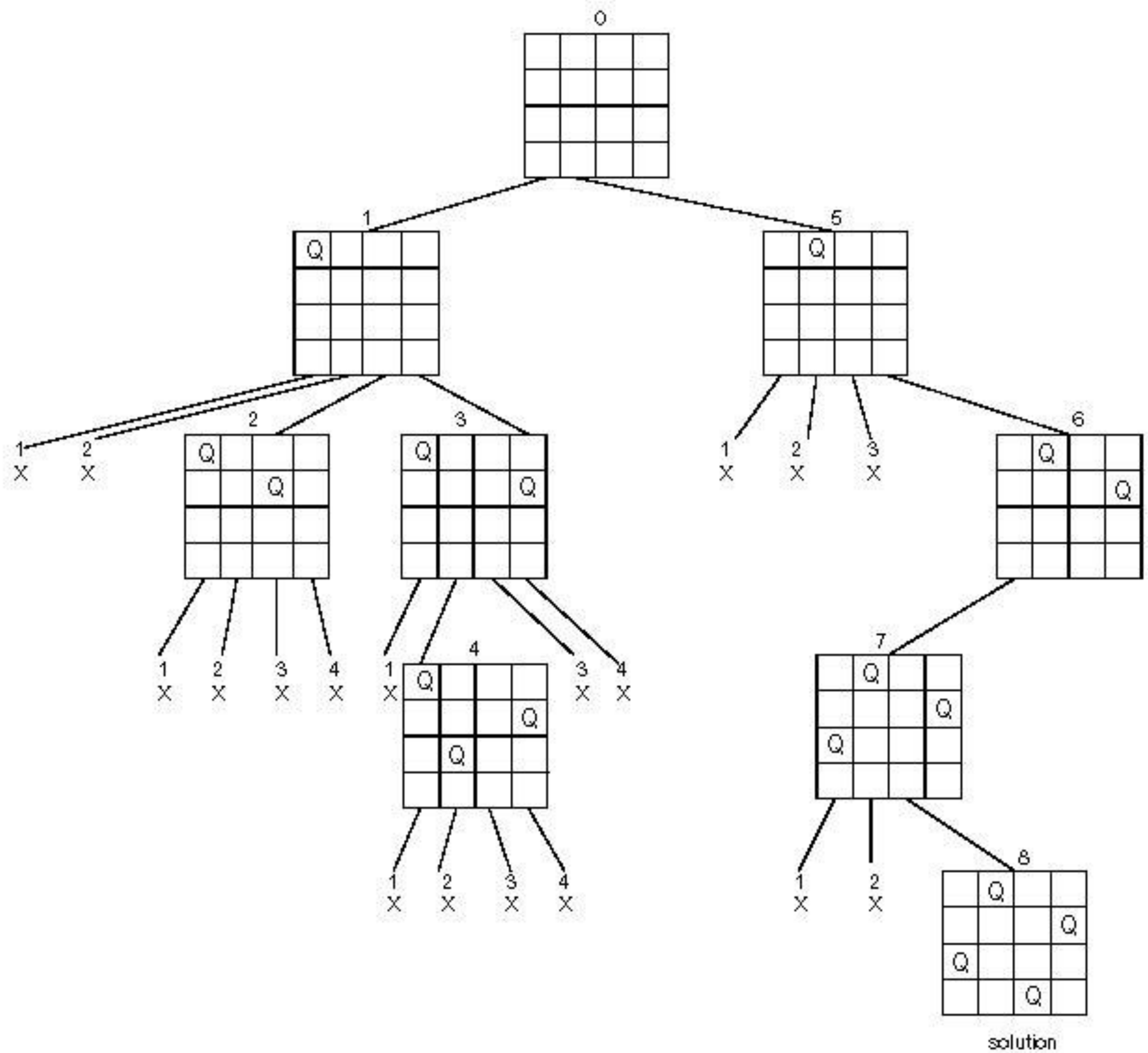
- Construct the ***state-space tree***
  - nodes: partial solutions
  - edges: choices in extending partial solutions
- Explore the state space tree using depth-first search
- "Prune" ***nonpromising nodes***
  - stop exploring subtrees rooted at nodes that cannot lead to a solution and backtracks to such a node's parent to continue the search

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

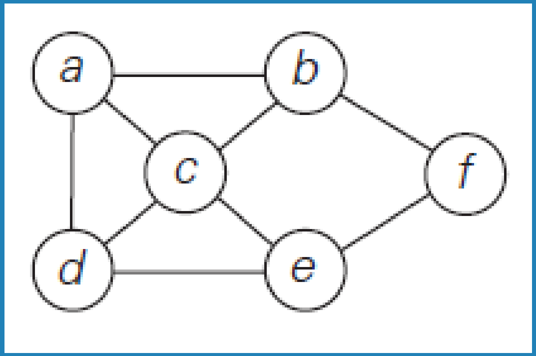


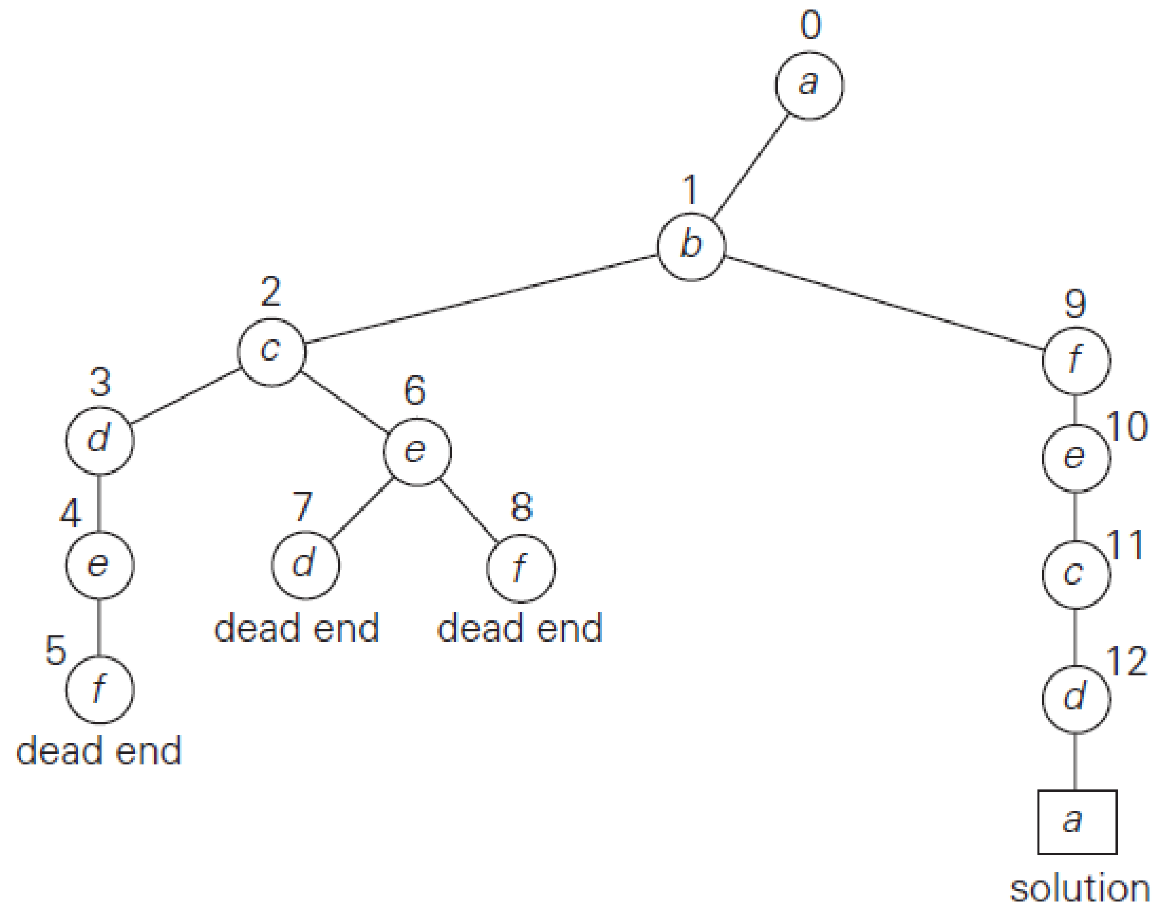
State-Space Tree of the 4-Queens Problem



× 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

Example: Hamiltonian Circuit Problem



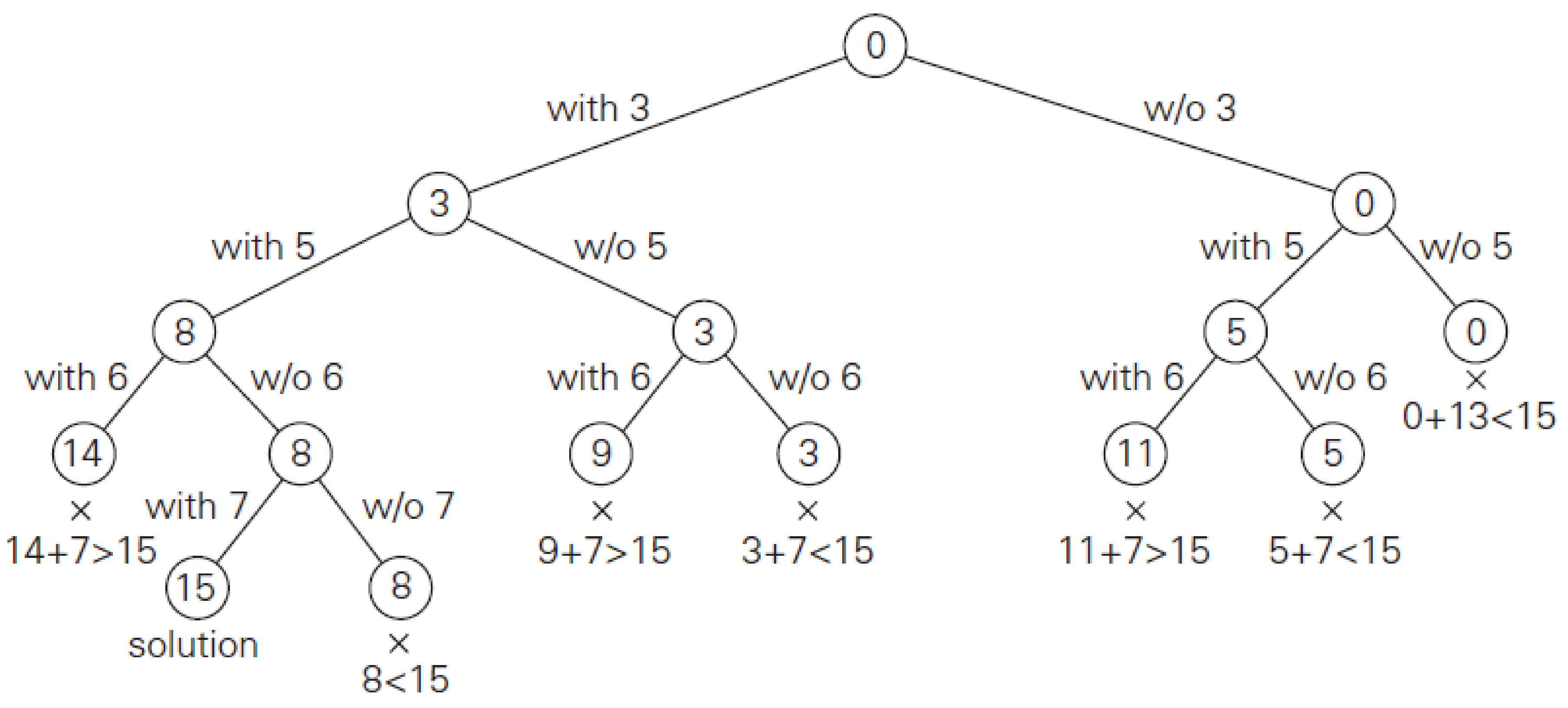


Example: Subset-Sum Problem

- **Find a subset of a given set $A = \{a_1, \ldots, a_n\}$ of n *positive integers whose sum is equal to a given positive* integer d.**

- **For example,**
  - **for $A = \{1, 2, 5, 6, 8\}$ *and* $d = 9$, *there are two solutions:* $\{1, 2, 6\}$ *and* $\{1, 8\}$.**

Example: Subset-Sum Problem (Cont.)



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

12.2 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**

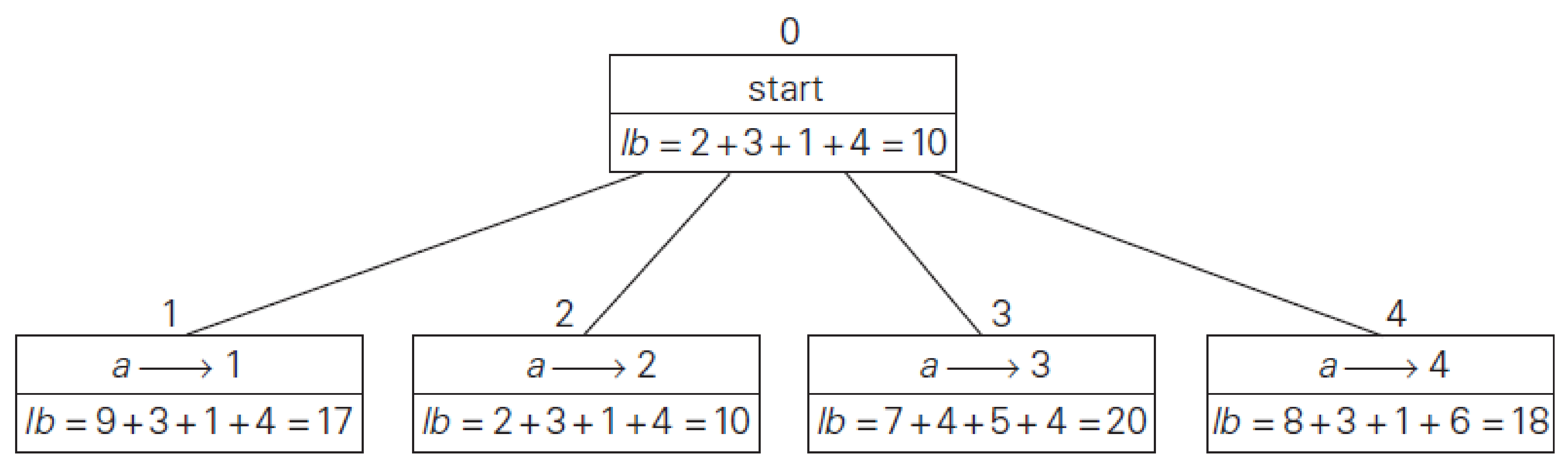
Example

	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

Lower bound: Any solution to this problem will have total cost at least: 2 + 3 + 1 + 4 (or 5 + 2 + 1 + 4)

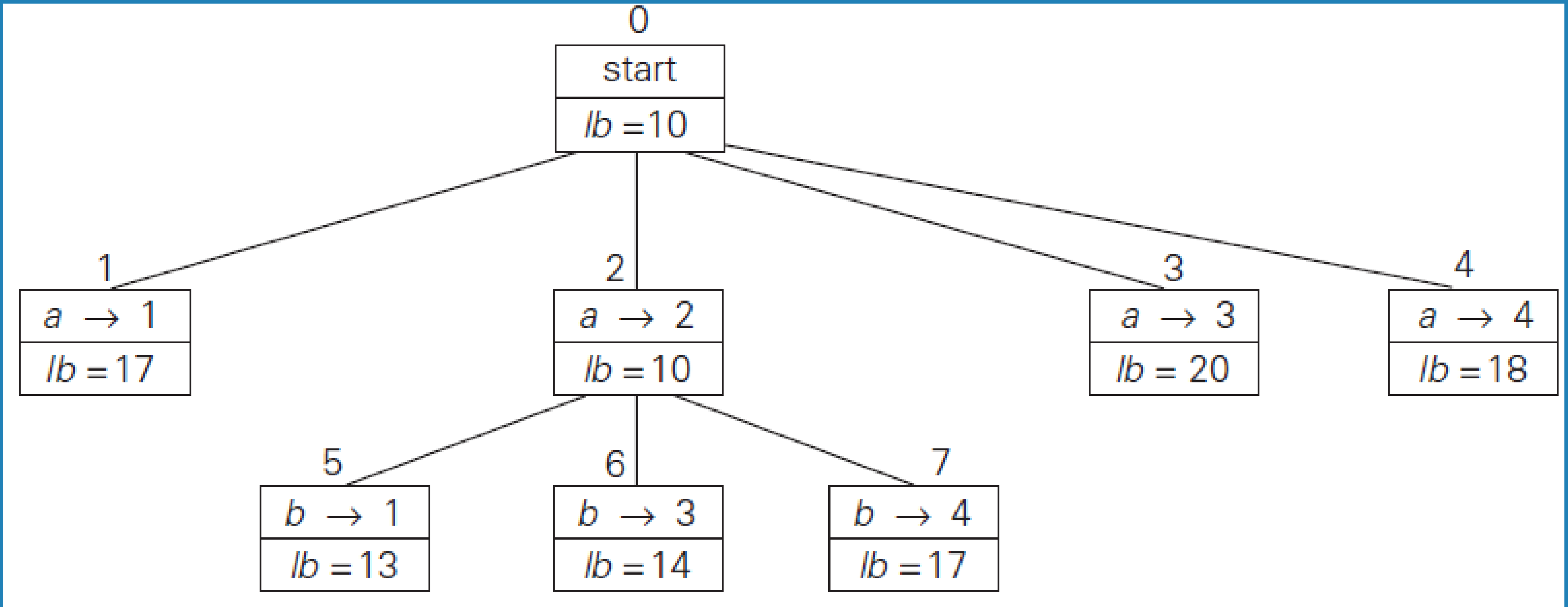
Example: First two levels of the state-space tree

- **Figure: 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***



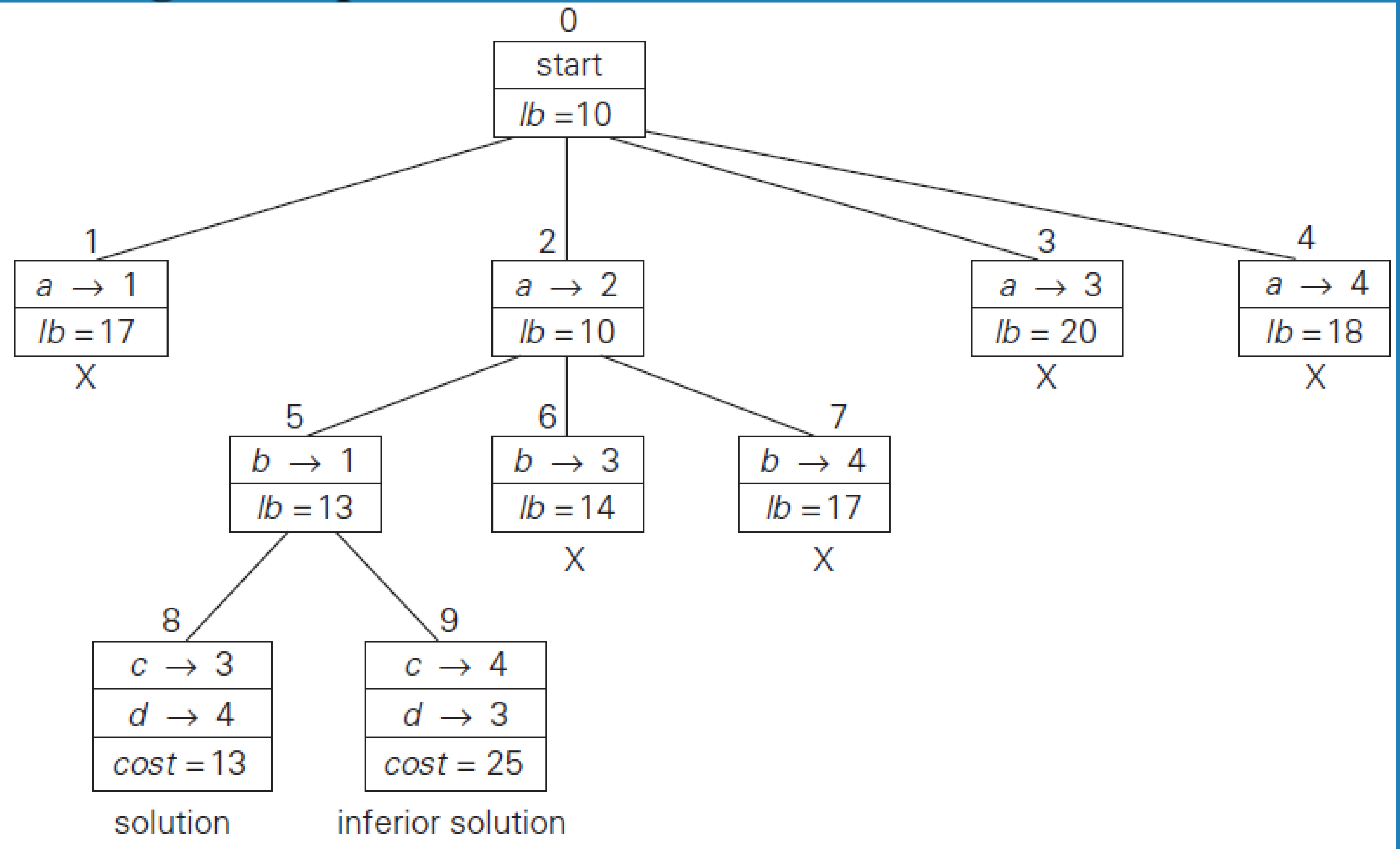
Example (cont.)

- **FIGURE: 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

- **FIGURE: Complete state-space tree for the instance of the assignment problem**



Example: Traveling Salesman Problem

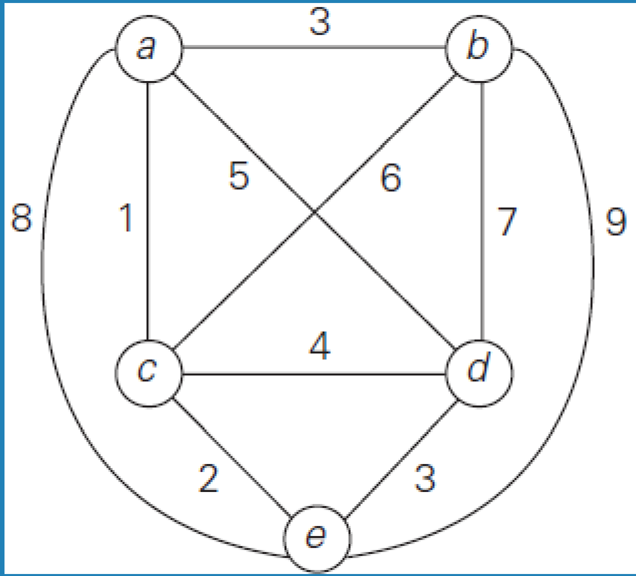
- **Lower bound**

$$lb = \left\lceil \sum_{i=1}^{n} s_i / 2 \right\rceil$$

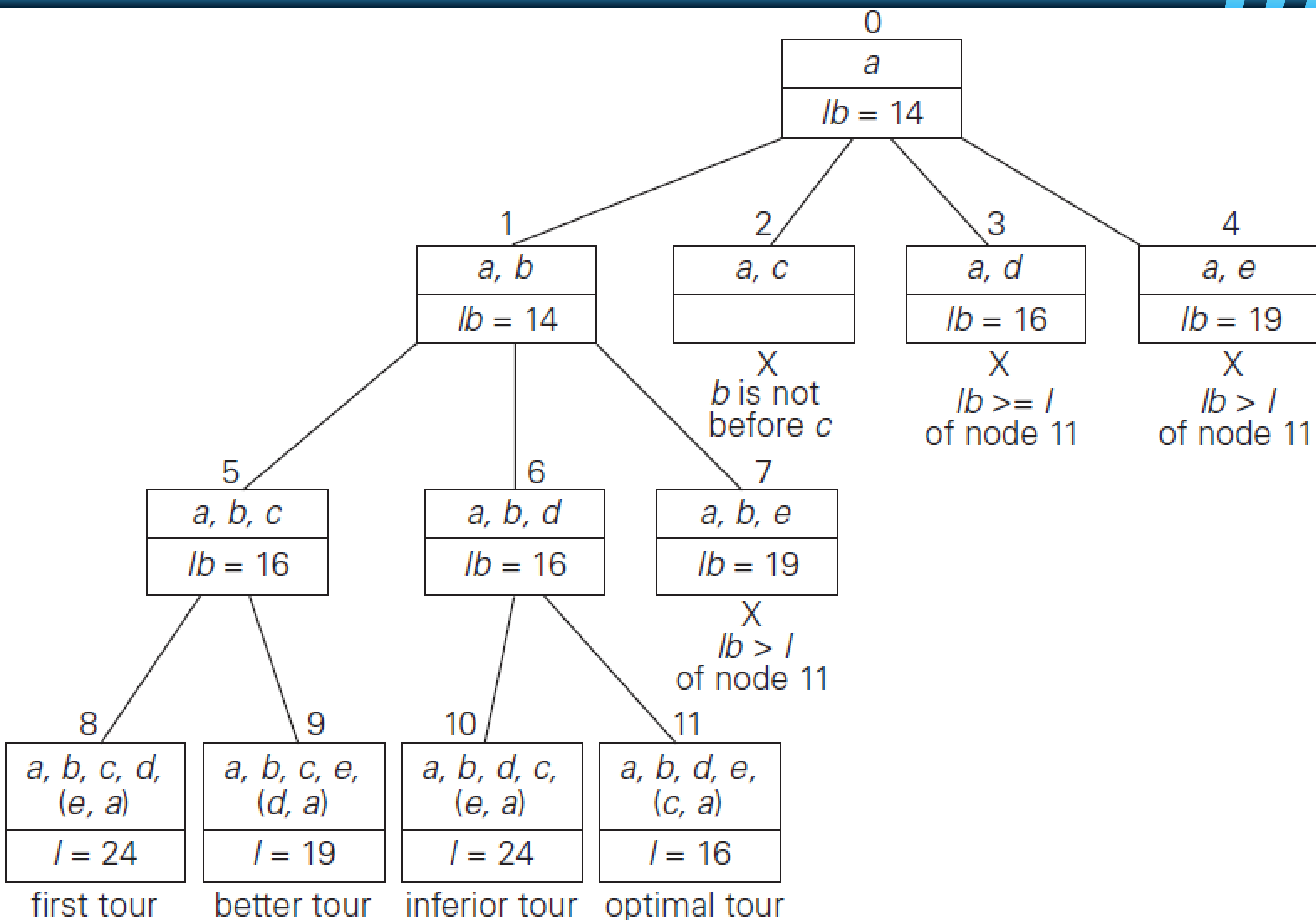
where s_i is the sum of the distances from city i to the two nearest cities, $1 \leq i \leq n$.

- **Example**

$$lb = \lceil [(1+3) + (3+6) + (1+2) + (3+4) + (2+3)]/2 \rceil = 14.$$



Example: Traveling Salesman Problem



Example: Knapsack Problem

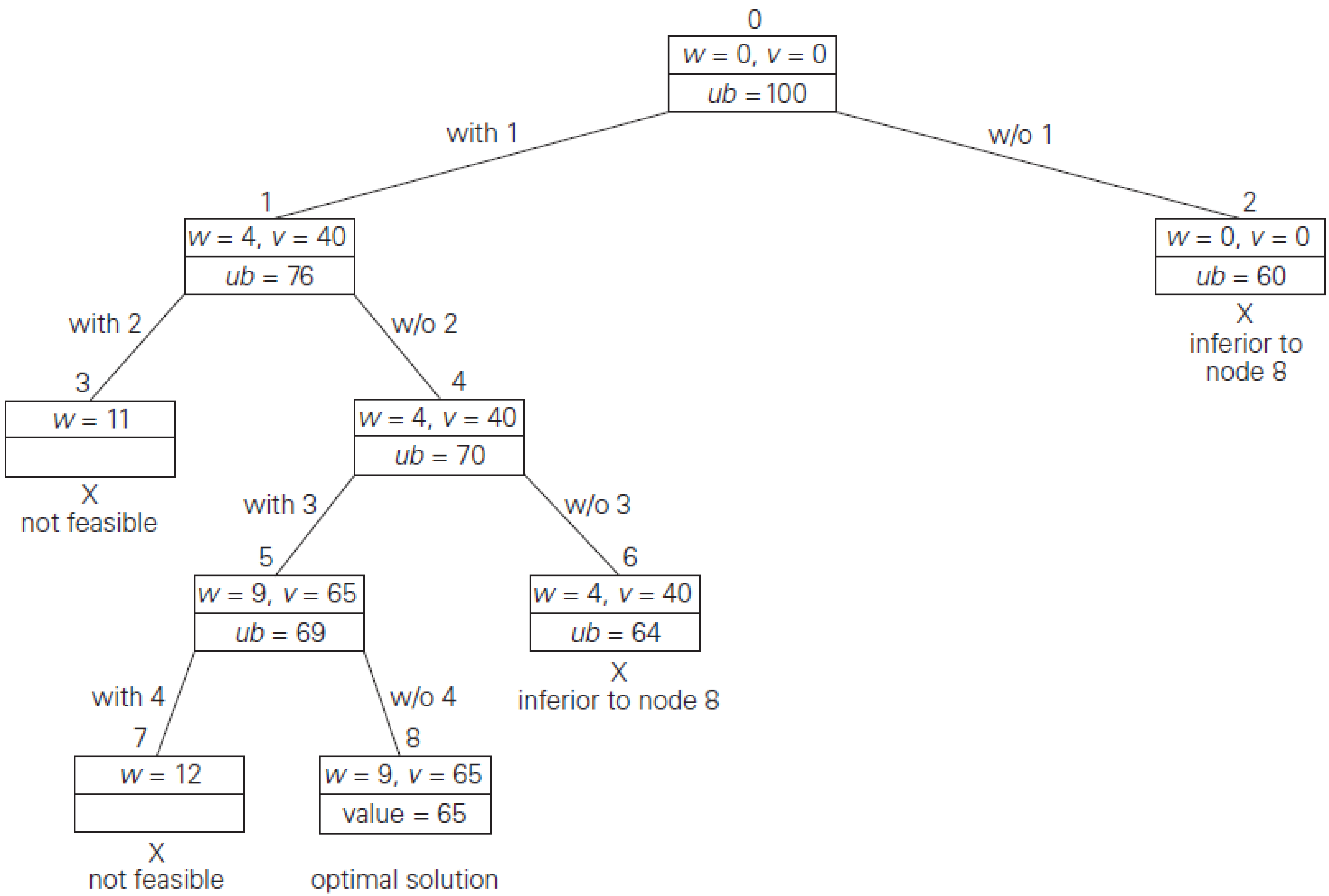
- The problem: given *n items* of known weights w_i *and values* v_i *, i = 1, 2, . . . , n, and a knapsack of capacity W,* find the most valuable subset of the items that fit in the knapsack.
- Assume: $v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_n/w_n$.
- Upper bound for the node *i*: $ub = v + (W - w)(v_{i+1}/w_{i+1})$.
  - *v* is total value so far
  - *w* is total capacity so far
- Example:

  The knapsack's capacity *W is 10*
  - *For node 0:*

  ub = 0 + (10-0)(40/4)=100*

item	weight	value	value / weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

Example: Knapsack Problem



12.3 Approximation Approach

Apply a fast (i.e., a polynomial-time) approximation algorithm to get a solution that is not necessarily optimal but hopefully close to it

Accuracy measures:

<u>accuracy ratio</u> **of an approximate solution s_a**

$$r(s_a) = f(s_a) / f(s^*) \quad \text{for minimization problems}$$

$$r(s_a) = f(s^*) / f(s_a) \quad \text{for maximization problems}$$

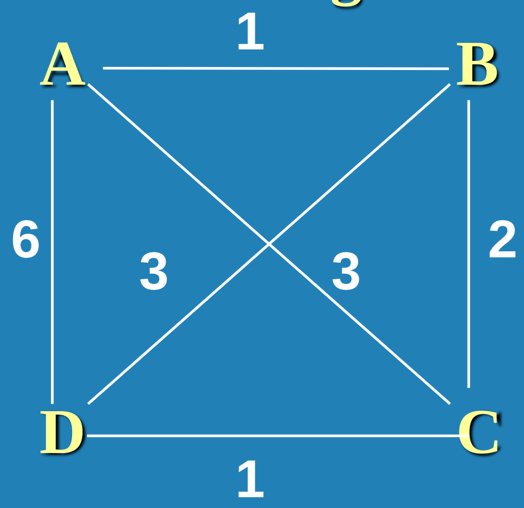
where $f(s_a)$ and $f(s^*)$ are values of the objective function f for the approximate solution s_a and actual optimal solution s^*

<u>performance ratio</u> **of the algorithm A**

the lowest upper bound of $r(s_a)$ on all instances

Nearest-Neighbor Algorithm for TSP

Starting at some city, always go to the nearest unvisited city, and, after visiting all the cities, return to the starting one



s_a : A – B – C – D – A of length 10

s^* : A – B – D – C – A of length 8

Note: Nearest-neighbor tour may depend on the starting city

Accuracy: $R_A = \infty$ (unbounded above) – make the length of AD arbitrarily large in the above example

Multifragment-Heuristic Algorithm

Stage 1: Sort the edges in nondecreasing order of weights. Initialize the set of tour edges to be constructed to empty set

Stage 2: Add next edge on the sorted list to the tour, skipping those whose addition would've created a vertex of degree 3 or a cycle of length less than *n*. Repeat this step until a tour of length *n* is obtained

Note: $R_A = \infty$, but this algorithm tends to produce better tours than the nearest-neighbor algorithm

Twice-Around-the-Tree Algorithm

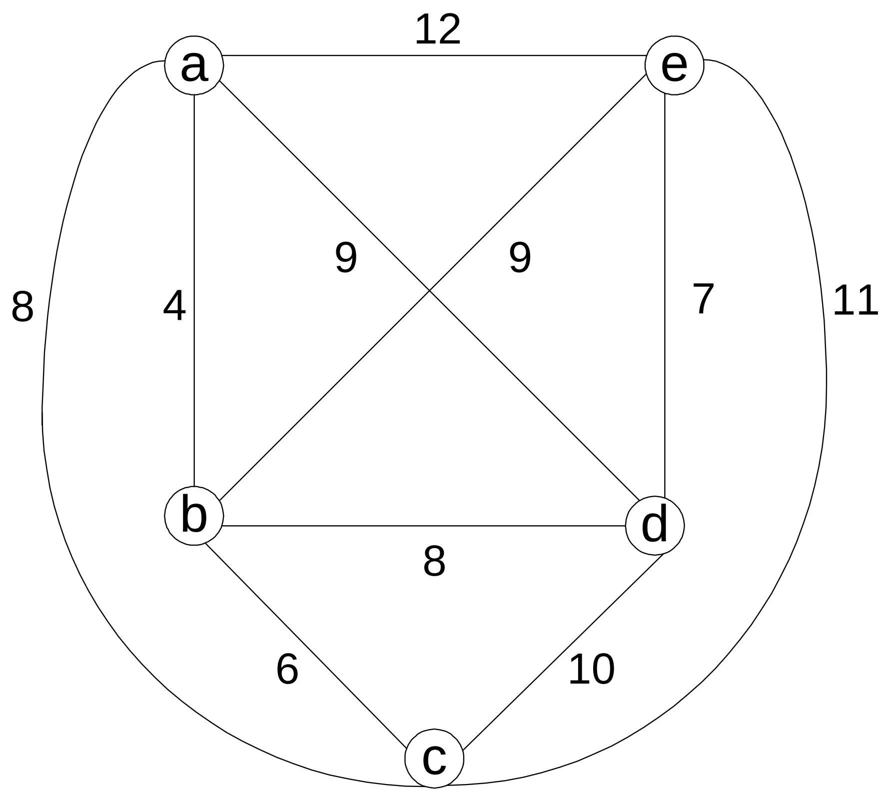
Stage 1: Construct a minimum spanning tree of the graph (e.g., by Prim's or Kruskal's algorithm)

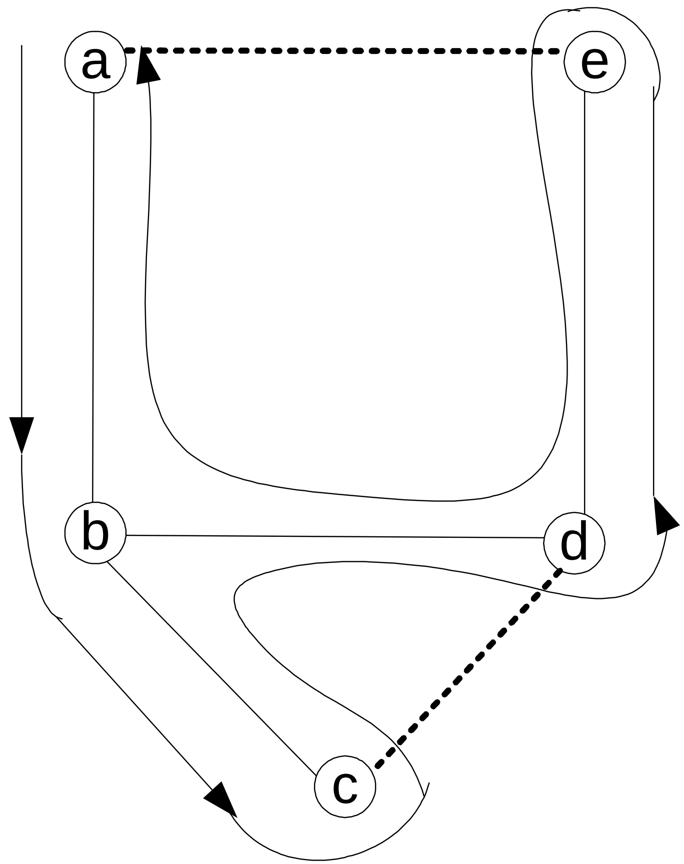
Stage 2: Starting at an arbitrary vertex, perform a walk around the minimum spanning tree recording all the vertices passed by. (This can be done by a DFS traversal.)

Stage 3: Scan the vertex list obtained in Step 2 and eliminate from it all repeated occurrences of the same vertex except the starting one at the end of the list. (This step is equivalent to making shortcuts in the walk.) The vertices remaining on the list will form a Hamiltonian circuit, which is the output of the algorithm

Note: $R_A = \infty$ for general instances, but this algorithm tends to produce better tours than the nearest-neighbor algorithm

Example





Walk: a – b – c – b – d – e – d – b – a

Tour: a – b – c – d – e – a

Christofides Algorithm

Stage 1: Construct a minimum spanning tree of the graph

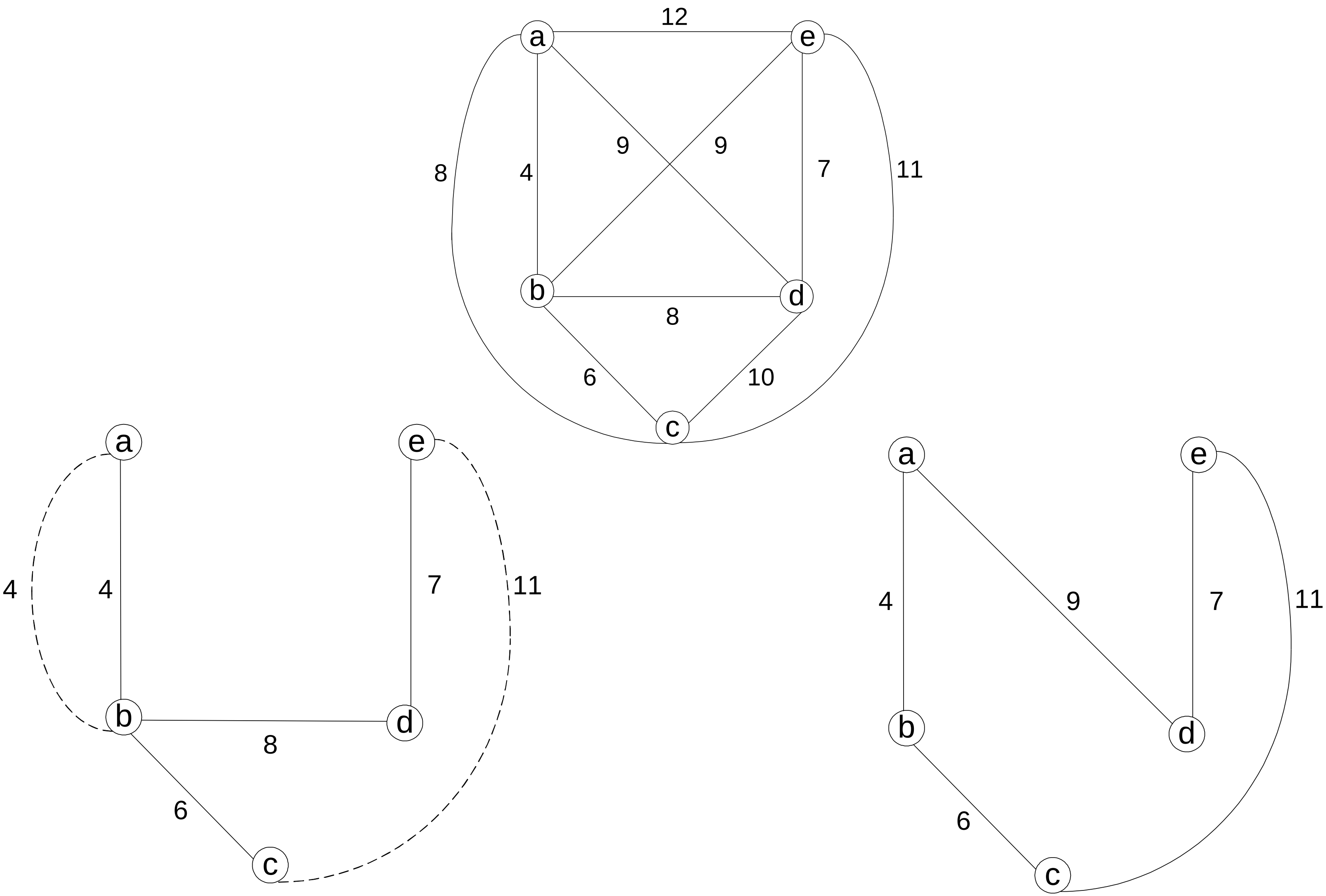
Stage 2: Add edges of a minimum-weight matching of all the odd vertices in the minimum spanning tree

Stage 3: Find an Eulerian circuit of the multigraph obtained in Stage 2

Stage 3: Create a tour from the path constructed in Stage 2 by making shortcuts to avoid visiting intermediate vertices more than once

$R_A = \infty$ for general instances, but it tends to produce better tours than the twice-around-the-minimum-tree alg.

Example: Christofides Algorithm



Euclidean Instances

Theorem If $P \neq NP$, there exists no approximation algorithm for TSP with a finite performance ratio.

Definition An instance of TSP is called *Euclidean*, if its distances satisfy two conditions:

1. *symmetry* $d[i, j] = d[j, i]$ for any pair of cities i and j
2. *triangle inequality* $d[i, j] \leq d[i, k] + d[k, j]$ for any cities i, j, k

For Euclidean instances:

approx. tour length / optimal tour length $\leq 0.5(\lceil \log_2 n \rceil + 1)$

for nearest neighbor and multifragment heuristic;

approx. tour length / optimal tour length $\leq$ 2

for twice-around-the-tree;

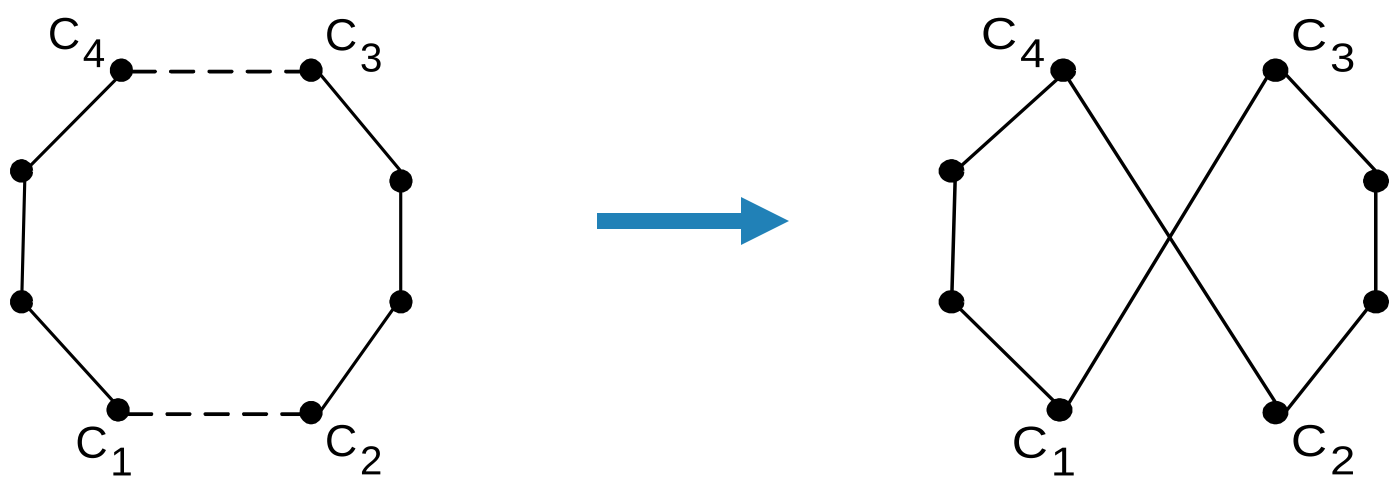
approx. tour length / optimal tour length $\leq$ 1.5

for Christofides

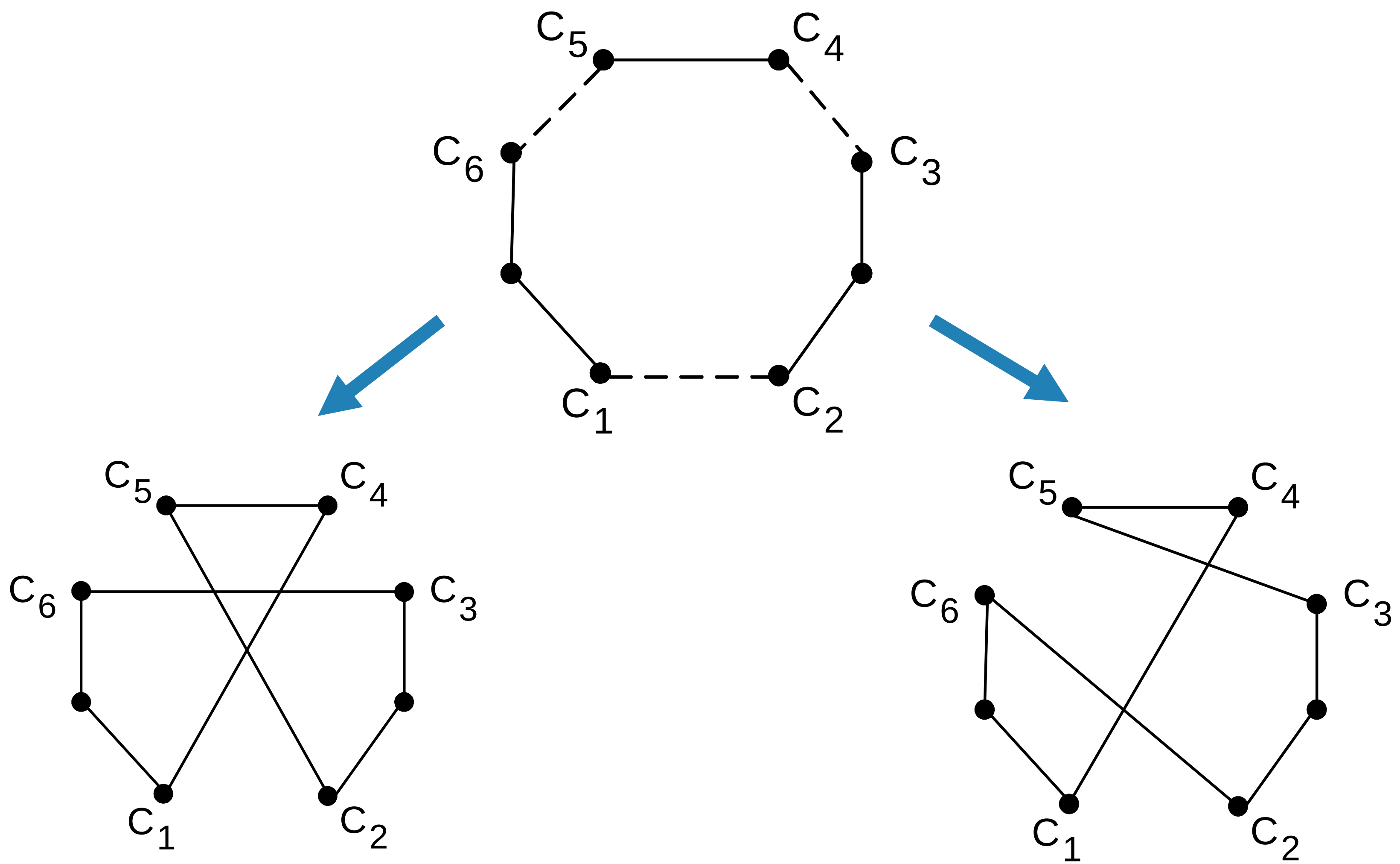
Local Search Heuristics for TSP

Start with some initial tour (e.g., nearest neighbor). On each iteration, explore the current tour's neighborhood by exchanging a few edges in it. If the new tour is shorter, make it the current tour; otherwise consider another edge change. If no change yields a shorter tour, the current tour is returned as the output.

Example of a 2-change



Example of a 3-change



Empirical Data for Euclidean Instances

TABLE 12.1 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

Greedy Algorithm for Knapsack Problem

Step 1: Order the items in decreasing order of relative values:
$v_1/w_1 \geq \cdots \geq v_n/w_n$

Step 2: Select the items in this order skipping those that don't fit into the knapsack

Example: The knapsack's capacity is 16

item	weight	value	v/w
1	2	\$40	20
2	5	\$30	6
3	10	\$50	5
4	5	\$10	2

Accuracy

- R_A is unbounded (e.g., $n = 2$, $C = m$, $w_1=1$, $v_1=2$, $w_2=m$, $v_2=m$)
- yields exact solutions for the continuous version

Approximation Scheme for Knapsack Problem

Step 1: Order the items in decreasing order of relative values:
$$v_1/w_1 \geq \cdots \geq v_n/w_n$$

Step 2: For a given integer parameter k, $0 \leq k \leq n$, generate all subsets of k items or less and for each of those that fit the knapsack, add the remaining items in decreasing order of their value to weight ratios

Step 3: Find the most valuable subset among the subsets generated in Step 2 and return it as the algorithm's output

- **Accuracy: $f(s^*) / f(s_a) \leq 1 + 1/k$ for any instance of size n**
- **Time efficiency: $O(kn^{k+1})$**
- **There are *fully polynomial schemes*: algorithms with polynomial running time as functions of both n and k**

12.4 Numerical Algorithms

- ***Numerical algorithms*** **concern with solving mathematical problems such as**
- **evaluating functions (e.g., $\sqrt{x}$, e^x, ln x, sin x)**
- **solving nonlinear equations**
- **finding extrema of functions**
- **computing definite integrals**

Most such problems are of “continuous” nature and can be solved only approximately

Principal Accuracy Metrics

- ***Absolute error*** **of approximation (of α^* by α)**

$$|\alpha - \alpha^*|$$

- ***Relative error*** **of approximation (of α^* by α)**

$$|\alpha - \alpha^*| / |\alpha^*|$$

  - **undefined for $\alpha^* = 0$**
  - **often quoted in %**

Two Types of Errors

- ***truncation errors***
  - Taylor's polynomial approximation

    $$e^x \approx 1 + x + x^2/2! + \cdots + x^n/n!$$

    absolute error $\leq M\,|x|^{n+1}/(n+1)!$ where $M = \max e^t$ for $0 \leq t \leq x$
  - composite trapezoidal rule

    $$\int_a^b f(x)dx \approx (h/2)\,[f(a) + 2\Sigma_{1\leq=i\leq=n-1} f(x_i) + f(b)],\ \ h = (b - a)/n$$

    absolute error $\leq (b\text{-}a)h^2 M_2 / 12$ where $M_2 = \max |f''(x)|$ for $a \leq x \leq b$

- ***round-off errors***

Solving Quadratic Equation

Quadratic equation $ax^2 + bx + c = 0$ $(a \neq 0)$

$$x_{1,2} = (-b \pm \sqrt{D})/2a \quad \text{where } D = b^2 - 4ac$$

Problems:

- computing square root

  use Newton's method: $x_{n+1} = 0.5(x_n + D/x_n)$

- subtractive cancellation

  use alternative formulas (see p. 411)

  use double precision for $D = b^2 - 4ac$

- other problems (overflow, etc.)

Notes on Solving Nonlinear Equations

- There exist no formulas with arithmetic ops. and root extractions for roots of polynomials

  $$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0 = 0 \text{ of degree } n \geq 5$$

- Although there exist special methods for approximating roots of polynomials, one can also use general methods for

  $$f(x) = 0$$

- Nonlinear equation $f(x) = 0$ can have one, many, infinitely many, and no roots at all

- Useful:
  - sketch graph of $f(x)$
  - separate roots

Three Classic Methods

Three classic methods for solving nonlinear equation

$$f(x) = 0$$

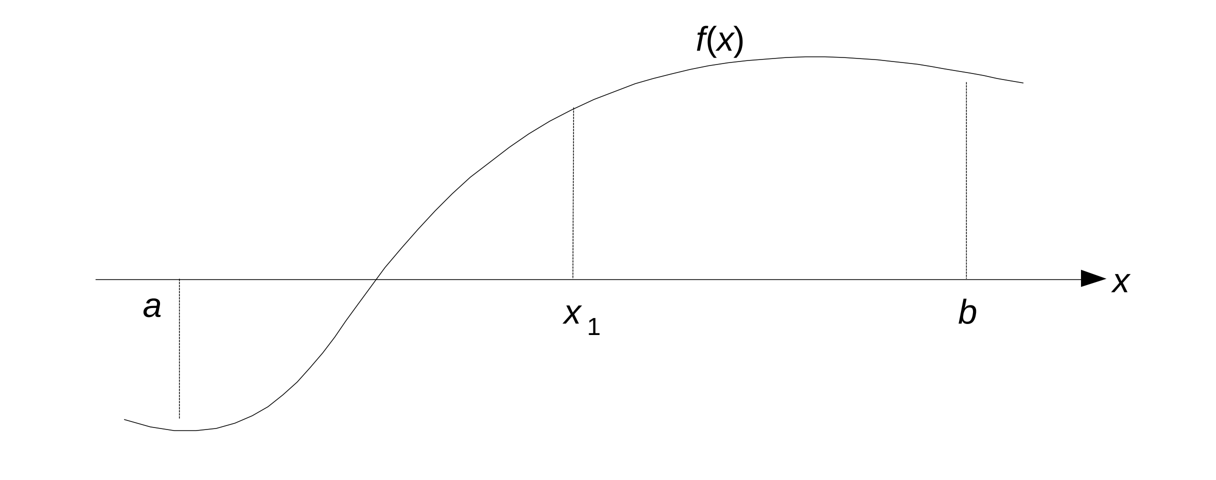
in one unknown:

- bisection method
- method of false position (regula falsi)
- Newton's method

Bisection Method

Based on

- Theorem: If $f(x)$ is continuous on $a \leq x \leq b$ and $f(a)$ and $f(b)$ have opposite signs, then $f(x) = 0$ has a root on $a < x < b$
- binary search idea



Approximations x_n are middle points of shrinking segments

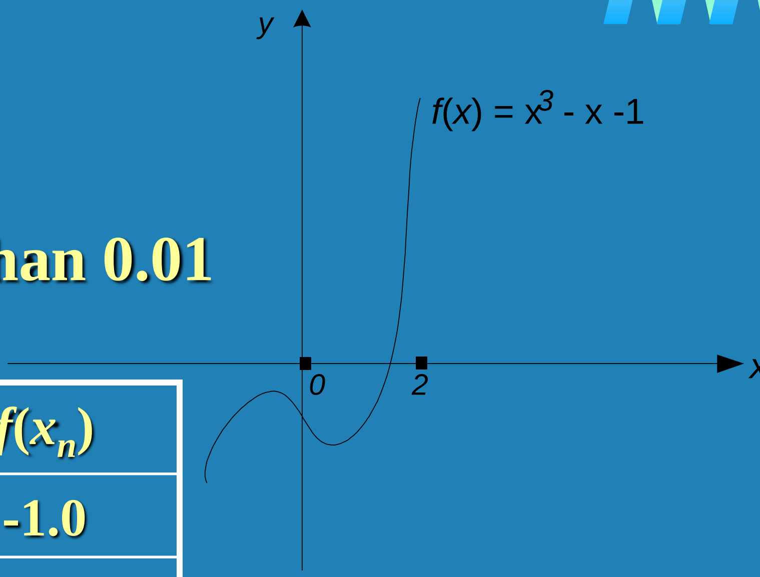
- $|x_n - x^*| \leq (b - a)/2^n$
- x_n always converges to root x^* but slower compared to others

Example of Bisection Method Application

Find the root of

$$x^3 - x - 1=0$$

with the absolute error not larger than 0.01

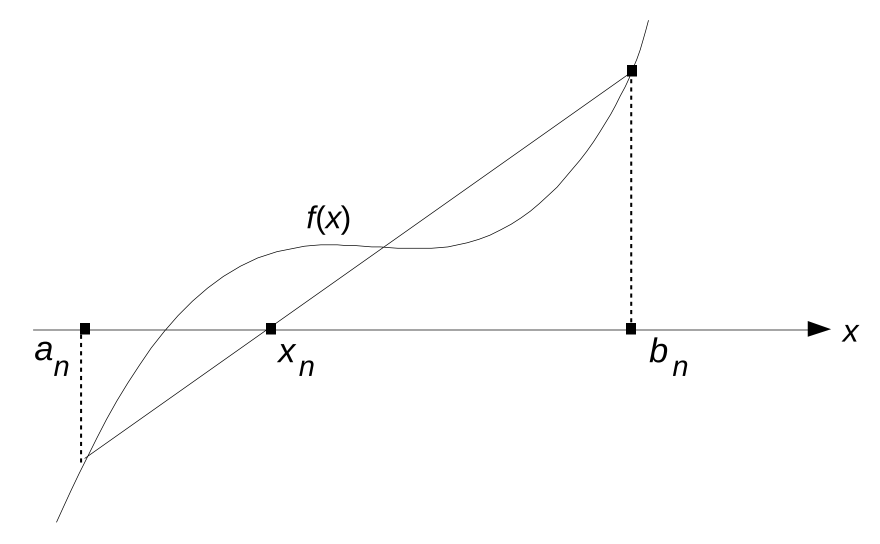


n	a_n	b_n	x_n	$f(x_n)$
1	0.0-	2.0+	1.0	-1.0
2	1.0-	2.0+	1.5	0.875
3	1.0-	1.5+	1.25	-0.296875
4				
5				
6				
7				
8	1.3125-	1.328125+	1.3203125	-0.018711

$x \approx 1.3203125$

Method of False Position

Similar to bisection method but uses *x*-intercept of line through (a, $f(a)$) and (b, $f(b)$) instead of middle point of [a,b]



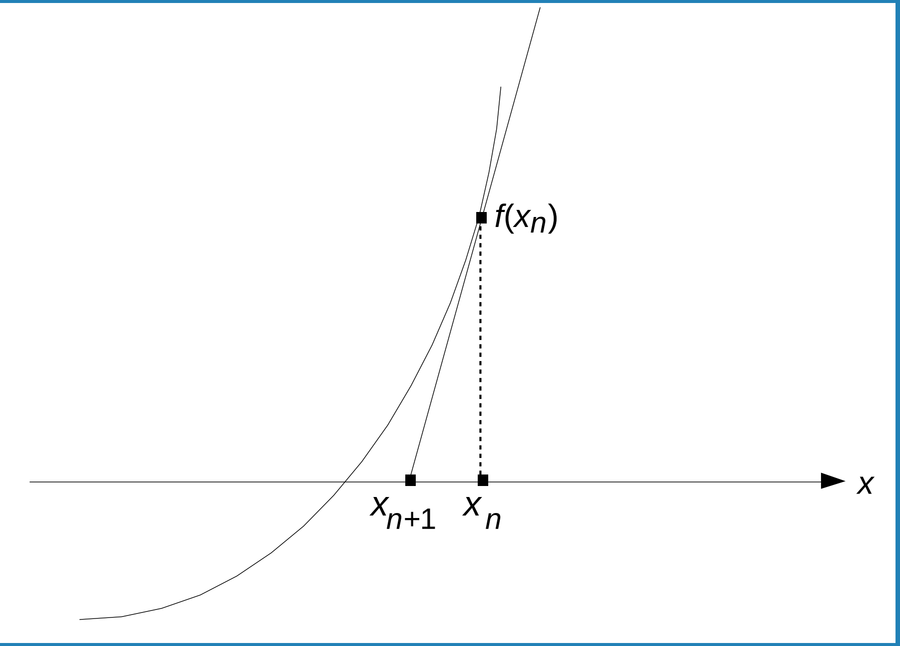
Approximations x_n are computed by the formula

$$x_n = [a_n f(b_n) - b_n f(a_n)] / [f(b_n) - f(a_n)]$$

- **Normally x_n converges faster than bisection method sequence but slower than Newton's method sequence**

Newton's Method

Very fast method in which x_n's are x-intercepts of tangent lines to the graph of $f(x)$



Approximations x_n are computed by the formula

$$x_{n+1} = x_n - f(x_n) / f'(x_n)$$

Notes on Newton's Method

- **Normally, approximations x_n converge to root very fast but can diverge with a bad choice of initial approximation x_0**

- **Yields a very fast method for computing square roots**

$$x_{n+1} = 0.5(x_n + D/x_n)$$

- **Can be generalized to much more general equations**

