Limitations of Algorithm Power

V. Balasubramanian
SSN College of Engineering

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 Bound

- For example, selection sort, whose efficiency is quadratic, is a reasonably fast algorithm,
- whereas the algorithm for the Tower of Hanoi problem is very slow because its efficiency is exponential.
- selection sort has to be considered slow because there are O(n log n) sorting algorithms;
- the Tower of Hanoi algorithm

Lower Bound

- lower bound can tell us how much improvement we can hope to achieve in our quest for a better algorithm for the problem in question.
- If such a bound is tight, i.e., we already know an algorithm in the same efficiency class as the lower bound, we can hope for a constant-factor improvement at best.

Contd…

- If there is a gap between the efficiency of the fastest algorithm and the best lower bound known, the door for possible improvement remains open: either a faster algorithm matching the lower bound could exist or a better lower bound could be proved.

Contd…

- There are two approaches to attacking a problem.
- One is to try to develop a more efficient algorithm for the problem.
- The other is to try to prove that a more efficient algorithm is not possible.

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

TABLE 11.1 Problems often used for establishing lower bounds by problem reduction

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

Matrix Multiplication

- matrix multiplication requires a O(n^3) algorithm.
- Strassen's matrix multiplication algorithm with a time complexity in O($n^{2.81}$).
- Coppersmith-Winograd O($n^{2.38}$)

Matrix multiplication

- It is possible to prove that matrix multiplication requires an algorithm whose time complexity is at least quadratic.
- Whether matrix multiplications can be done in quadratic time remains an open question, because no one has ever created a quadratic-time algorithm for matrix multiplication, and no one has proven that it is not possible to create such an algorithm.

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.
- Since any algorithm must at least “read” all the items it needs to process and “write” all its outputs, such a count yields a trivial lower bound.

Polynomial Evaluation Ω(n)

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$$

Matrix Multipliaction

- $n \times n$ matrices is n^2 because any such algorithm has to process $2n^2$ elements
- Trivial lower bounds are often too low to be useful. For example, the trivial
- bound for the traveling salesman problem is $\Omega(n^2)$, because its input is $n(n - 1)/2$ intercity distances and its output is a list of $n + 1$ cities making up an optimal tour.

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 <u>must</u> be processed

Decision Trees

Decision tree — a convenient model of algorithms involving

comparisons in which:

- internal nodes represent comparisons
- leaves represent outcomes

- Such a bound is called the ***informationtheoretic lower bound***

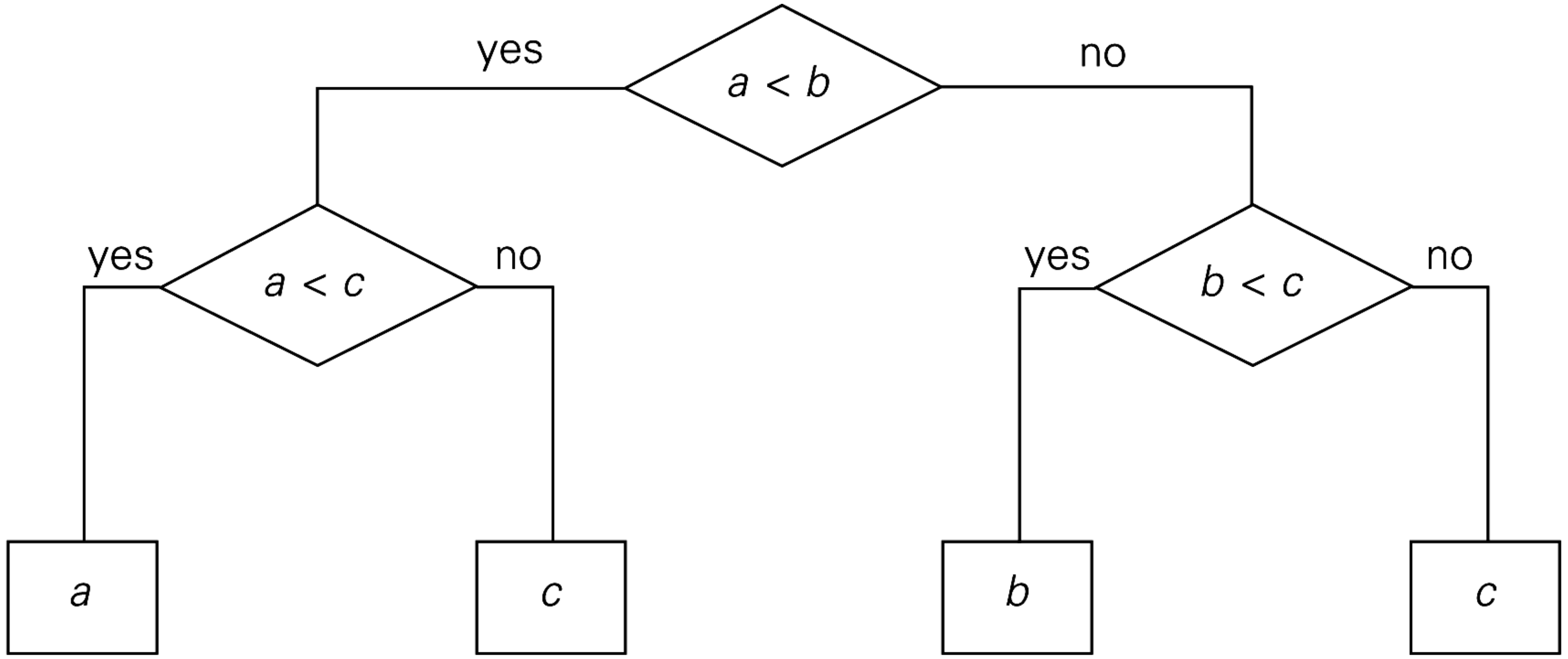
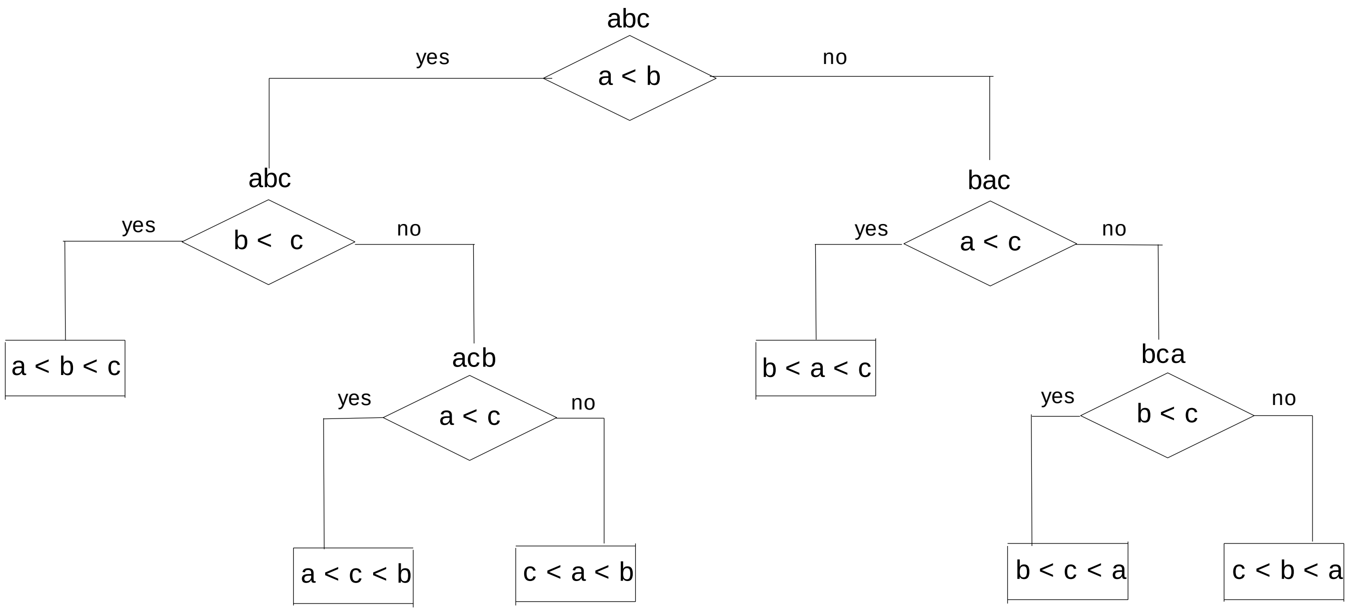


FIGURE 11.1 Decision tree for finding a minimum of three numbers

Decision tree for sorting three numbers. – Insertion sort



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

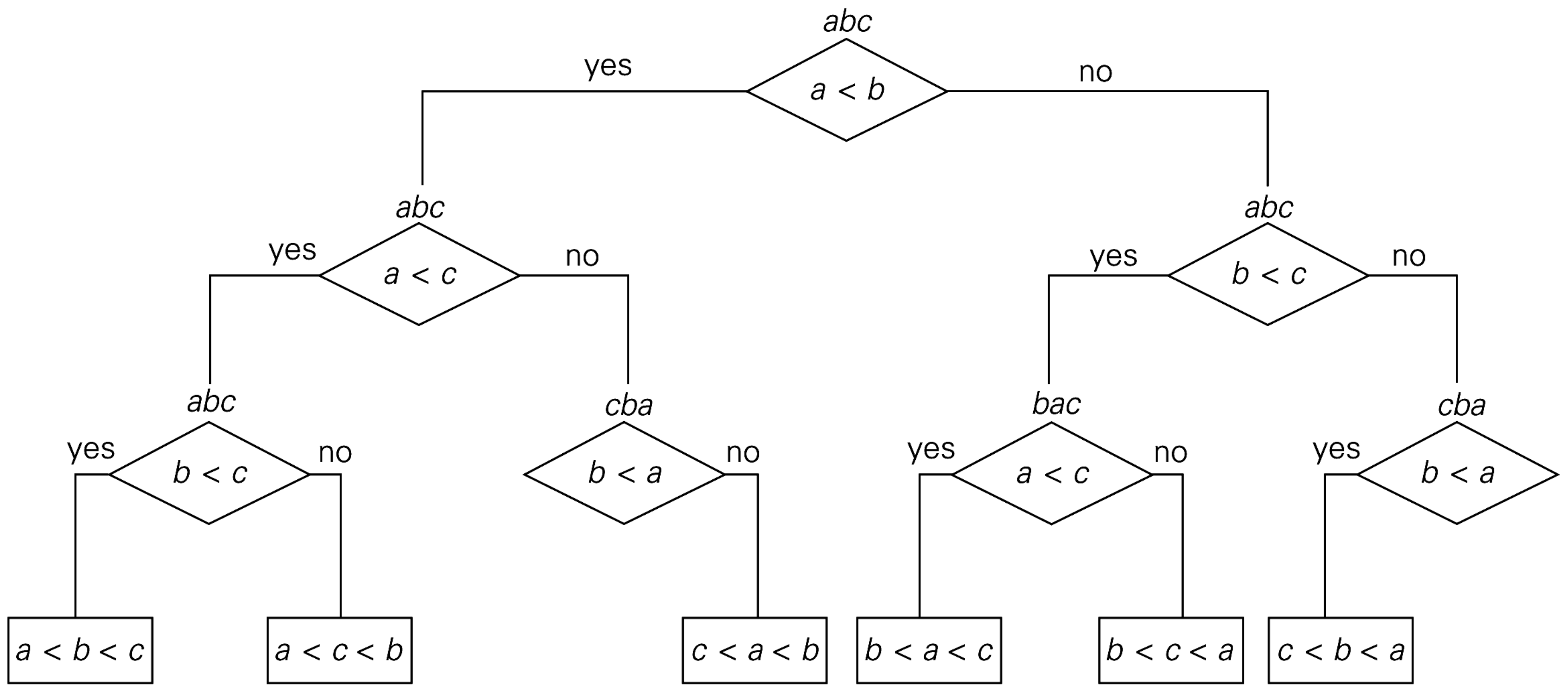


FIGURE 11.2 Decision tree for the three-element selection sort. A triple above a node indicates the state of the array being sorted. Note the two redundant comparisons $b < a$ with a single possible outcome because of the results of some previously made comparisons.

Sorting

- outcome of a sorting algorithm as finding a permutation of the element indices of an input list that puts the list’s elements in ascending order.
- the number of possible outcomes for sorting an arbitrary n-element list is equal to n!.

$$C_{worst}(n) \geq \lceil \log_2 n! \rceil. \qquad \textbf{(11.2)}$$

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

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

- We can also use decision trees for analyzing the average-case efficiencies of comparison-based sorting algorithms. We can compute the average number of comparisons for a particular algorithm as the average depth of its decision tree’s leaves, i.e., as the average path length from the root to the leaves.

Insertion sort

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

$$C_{avg}(n) \geq \log_2 n!.$$

Decision tree for sorting

- 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)

Searching

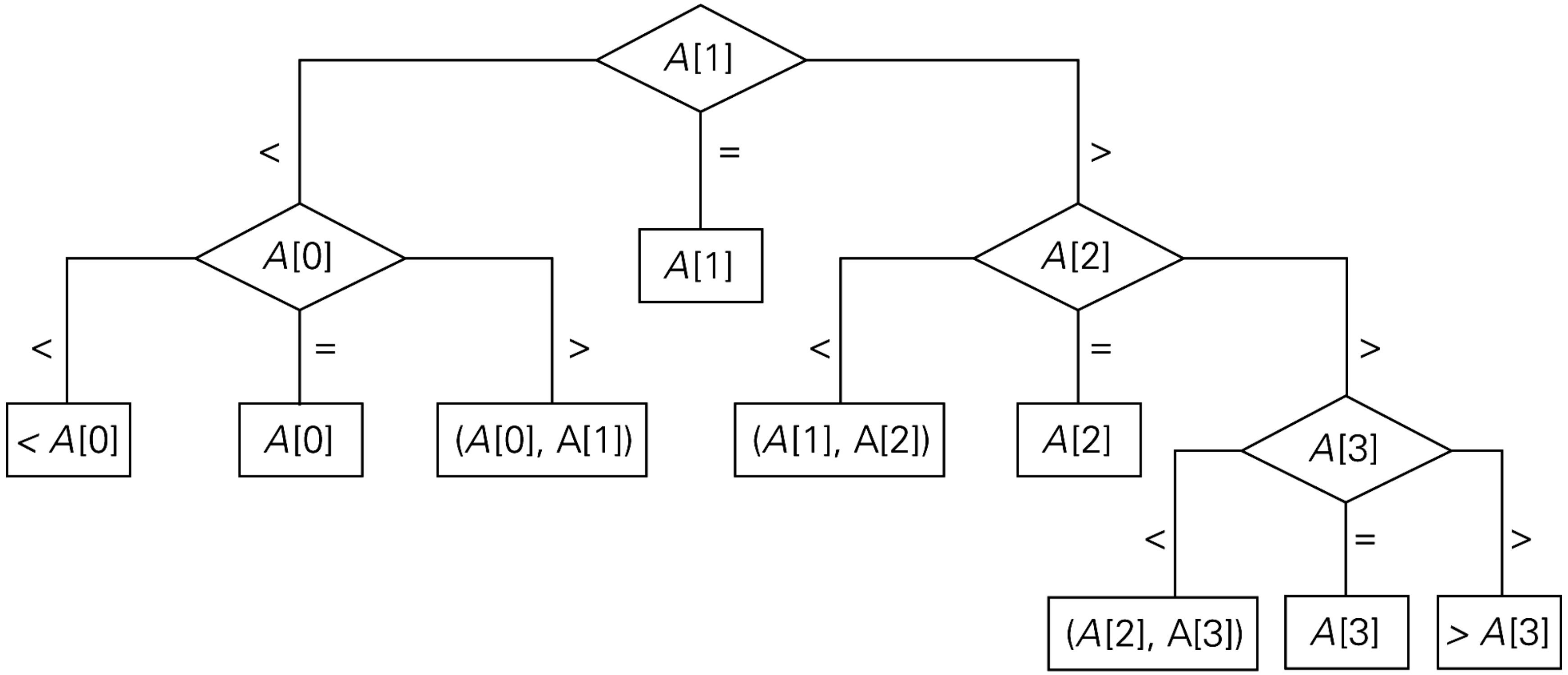


FIGURE 11.4 Ternary decision tree for binary search in a four-element array

- For an array of n elements, all such decision trees will have 2n + 1 leaves (n for successful searches and n + 1 for unsuccessful ones)

$$C_{worst}(n) \geq \lceil \log_3(2n+1) \rceil.$$

- To obtain a better lower bound, we should consider binary rather than ternary decision trees,
- Internal nodes in such a tree correspond to the same three way
- comparisons as before, but they also serve as terminal nodes for successful searches.
- Leaves therefore represent only unsuccessful searches, and there are n + 1 of them for searching an n-element array.

Binary Decision Tree

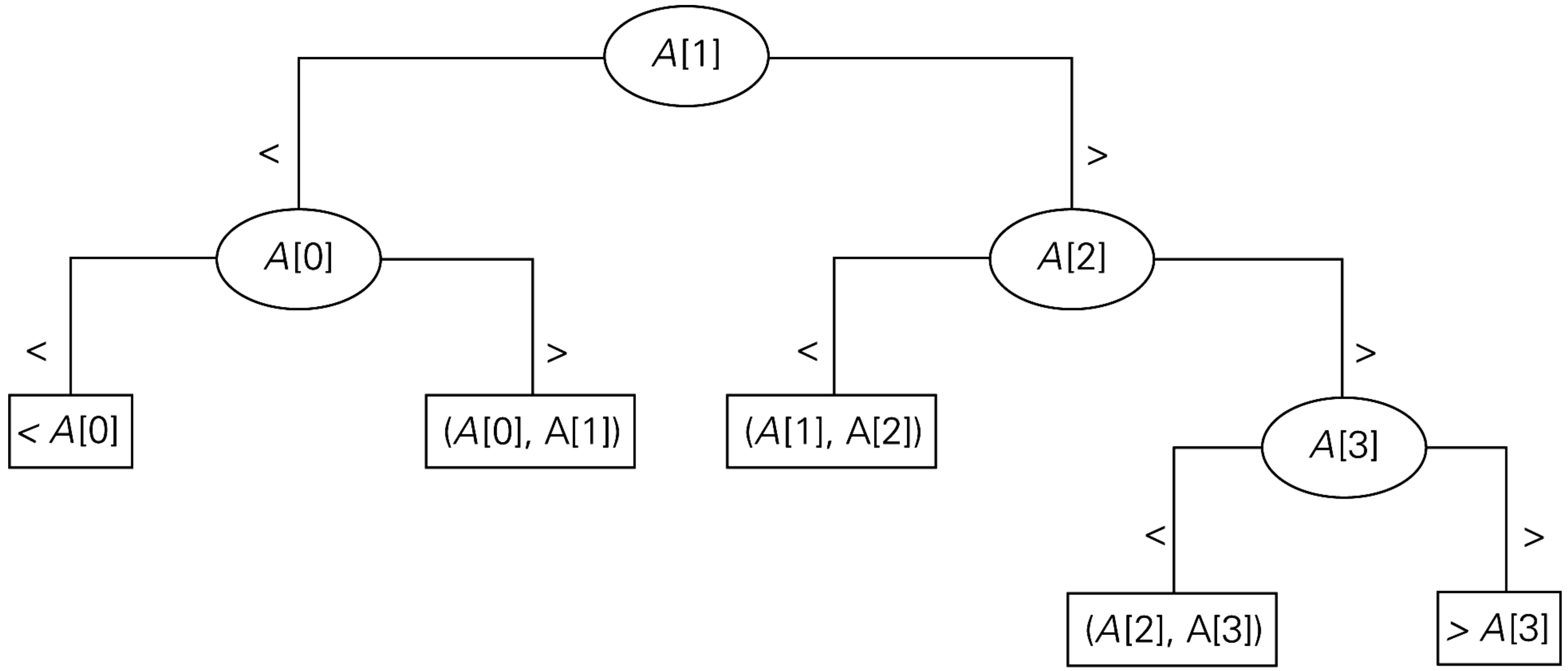


FIGURE 11.5 Binary decision tree for binary search in a four-element array

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

Problem

- Prove that the classic recursive algorithm for the Tower of Hanoi puzzle makes the minimum number of disk moves needed to solve the problem.

Solution

$M(n) \geq 2^n - 1$ for $n \geq 1$.

$$M(n+1) \geq (2^n - 1) + 1 + (2^n - 1) = 2^{n+1} - 1.$$

- Find a trivial lower-bound class for each of the following problems and indicate, if you can, whether this bound is tight.
- **a. finding the largest element in an array**
- **b. checking completeness of a graph represented by its adjacency matrix**
- **c. generating all the subsets of an n-element set**
- **d. determining whether n given real numbers are all distinct**

Solution –Largest element

- All n elements of a given array need to be processed to find its largest element, and just one item needs to be produced.
- Hence the trivial lower bound is linear.
- It is tight because the standard one-pass algorithm for this problem is in Thetha(n)

- b. Since the existence of an edge between all n(n - 1)/2 pairs of vertices needs to be verified in the worst case before establishing completeness of a graph with n vertices, the trivial lower bound is quadratic. It is tight because this is the amount of work done by the brute-force algorithm that simply checks all the elements in the upper-triangular part of the matrix until either a zero is encountered or no unchecked elements are left.

- The size of the problem’s output is 2^n
- The size of the problem’s input is n while the output is just one bit. Hence, the trivial lower bound is linear. It is not tight: according to the known result quoted in the section, the tight lower bound for this problem is n log n.

Consider the problem of identifying a lighter fake coin among n identical-looking coins with the help of a balance scale. Can we use the same information-theoretic argument as the one in the text for the number of questions in the guessing game to conclude that any algorithm for identifying the fake will need at least $\lceil \log_2 n \rceil$ weighings in the worst case?

Solution

- The answer is no. The problem can be solved with fewer weighings by dividing the coins into three rather than two subsets with about the same number of coins each. The information-theoretic argument, if used, has to take into account the fact that one weighing reduces uncertainty more than for the number-guessing problem because it can have three rather than two outcomes.

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 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

Example

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

Here is an adversary strategy that leads to the stated lower bound. The adversary maintains a list L of all possible values that are still legal for x. So initially the n integers $\{1, 2, 3, \ldots, n-1, n\}$ are placed in L. Each time A asks D if $x < y$, D counts how many of of the integers in L are greater than x. If at least half of the numbers in L are greater than x then D will respond "yes," and otherwise D will respond "no." If D responds "yes" (i.e., $x < y$) then all elements in the list that are $\geq y$ must be removed from L (or otherwise the adversary would be lying). Likewise, if D responds "no" (i.e., $x \geq y$) then all elements in the list that are $< y$ must be removed from L.

- $|L_0| = n$, and
- $|L_{i+1}| \geq \lceil |L_i|/2 \rceil$ for $i > 0$

From this it follows that $i = \lceil \log_2 n \rceil$ is the smallest possible value for i for which $|L_i| = 1$. (This could be proven by induction.) Let's do an example to illustrate this where $n = 100$. By recursive definition:

$$|L_0| = 100,\ |L_1| \geq 50,\ |L_2| \geq 25,\ |L_3| \geq 13,\ |L_4| \geq 7,\ |L_5| \geq 4,\ |L_6| \geq 2,\ |L_7| \geq 1$$

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 $\Omega(n\log n)$)

P Class

A ***polynomial-time algorithm*** is one whose worst-case time complexity is bounded above by a polynomial function of its input size. That is, if n is the input size, there exists a polynomial $p(n)$ such that

$$W(n) \in O(p(n)).$$

Algorithms with the following worst-case time complexities are all polynomial-time.

$$2n \qquad 3n^3 + 4n \qquad 5n + n^{10} \qquad n \lg n$$

Algorithms with the following worst-case time complexities are not polynomial-time.

$$2^n \qquad 2^{0.01n} \qquad 2^{\sqrt{n}} \qquad n!$$

$$n \lg n < n^2,$$

- intractable, there must be no polynomial-time algorithm that solves it.
- Obtaining a nonpolynomial-time algorithm for a problem does not make it intractable.
- For example, the brute-force algorithm for the Chained Matrix Multiplication Problem is non polynomial-time.
- So is the divide-and-conquer algorithm that uses the recursive property.
- However, the dynamic programming algorithm developed in that section is O(n3). The problem is not intractable, because we can solve it in polynomial time

There are three general categories of problems as far as intractability is concerned:

1. Problems for which polynomial-time algorithms have been found
2. Problems that have been proven to be intractable
3. Problems that have not been proven to be intractable, but for which polynomial-time algorithms have never been found

Undecidable

- A famous example of an undecidable problem was given by Alan Turing in 1936. The problem in question is called the ***halting problem: given a*** computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it.

- 0-1 Knapsack Problem,
- the Traveling Salesperson Problem, the Sum-of-Subjects Problem,
- the m-Coloring Problem for $m > 3$, the Hamiltonian Circuits Problem, and the problem of abductive inference in a belief network all fall into this category.

- ***Bin-packing problem***
- ***Partition problem Given n positive integers, determine whether it is possible :*** to partition them into two disjoint subsets with the same sum.

A game of chess can be posed as the following decision problem: given a legal positioning of chess pieces and information about which side is to move, determine whether that side can win. Is this decision problem decidable?

Yes, it's decidable. Theoretically, we can simply generate all the games (i.e., all the sequences of all legal moves of both sides) from the position given and check whether one of them is a win for the side that moves next.

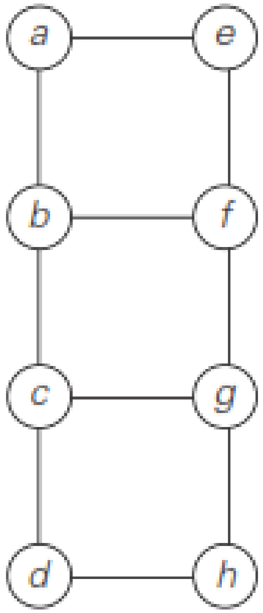
A certain problem can be solved by an algorithm whose running time is in $O(n^{\log_2 n})$. Which of the following assertions is true?

a. The problem is tractable.

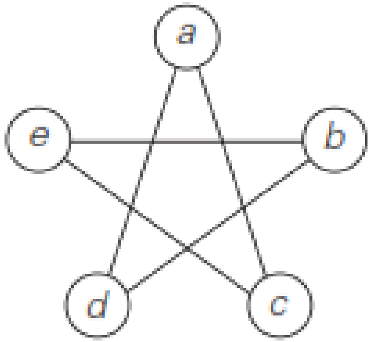
b. The problem is intractable.

c. Impossible to tell.

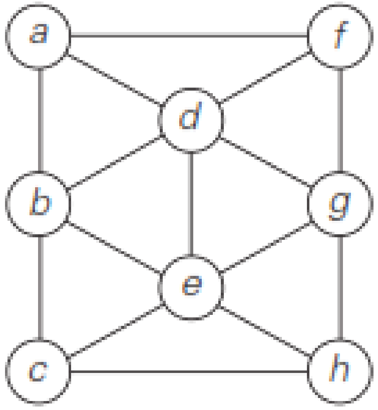
a.



b.



c.



A ***clique*** in an undirected graph $G = (V, E)$ is a subset W of V such that each vertex in W is adjacent to all the other vertices in W. For the graph in Figure 9.1, $\{v_2, v_3, v_4\}$ is a clique, whereas $\{v_1, v_2, v_3\}$ is not a clique because v_1 is not adjacent to v_3. A ***maximal clique*** is a clique of maximal size. The only maximal clique in the graph in Figure 9.1 is $\{v_1, v_2, v_4, v_5\}$.

The ***Clique Optimization Problem*** is to determine the size of a maximal clique for a given graph.

The ***Clique Decision Problem*** is to determine, for a positive integer k, whether there is a clique containing at least k vertices. This problem has the same parameters as the Clique Optimization Problem plus the additional parameter k.

Max Clique

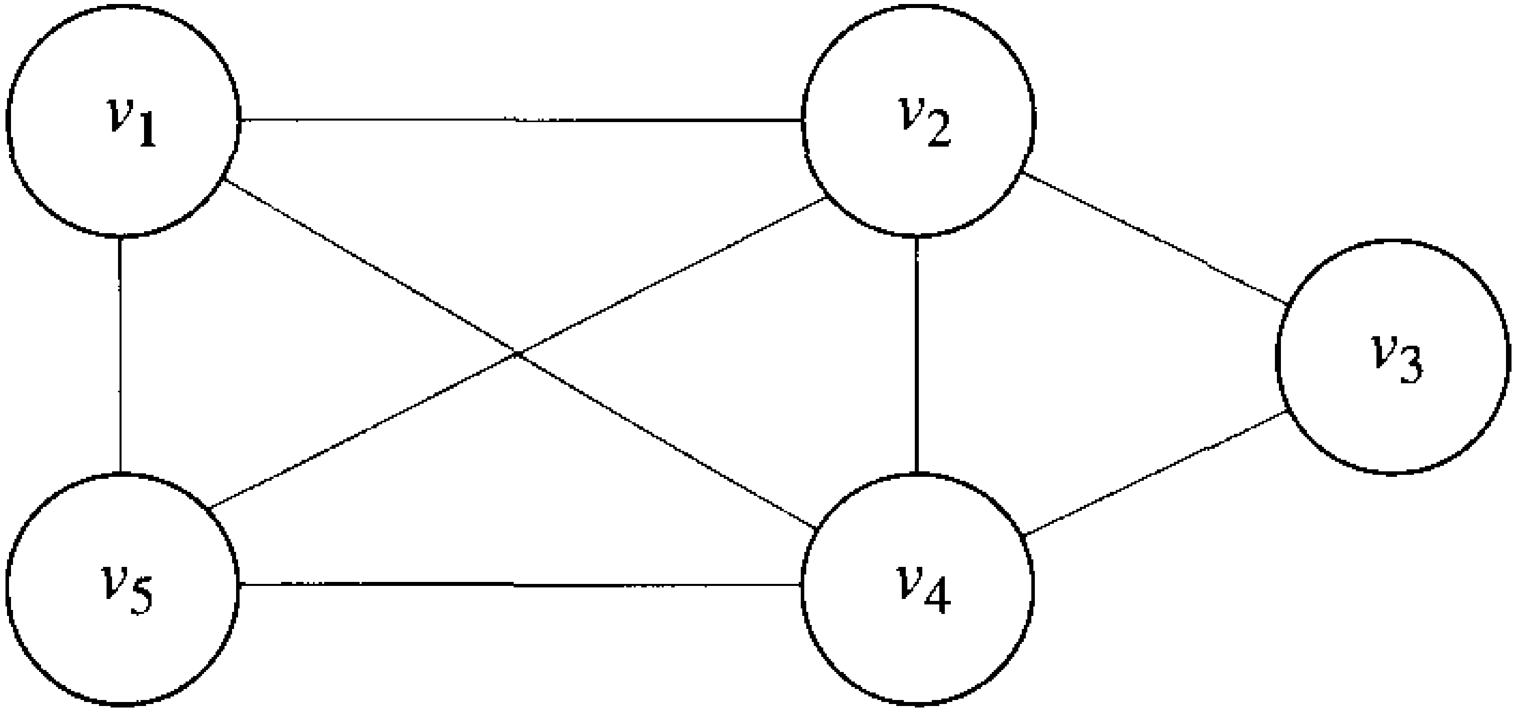
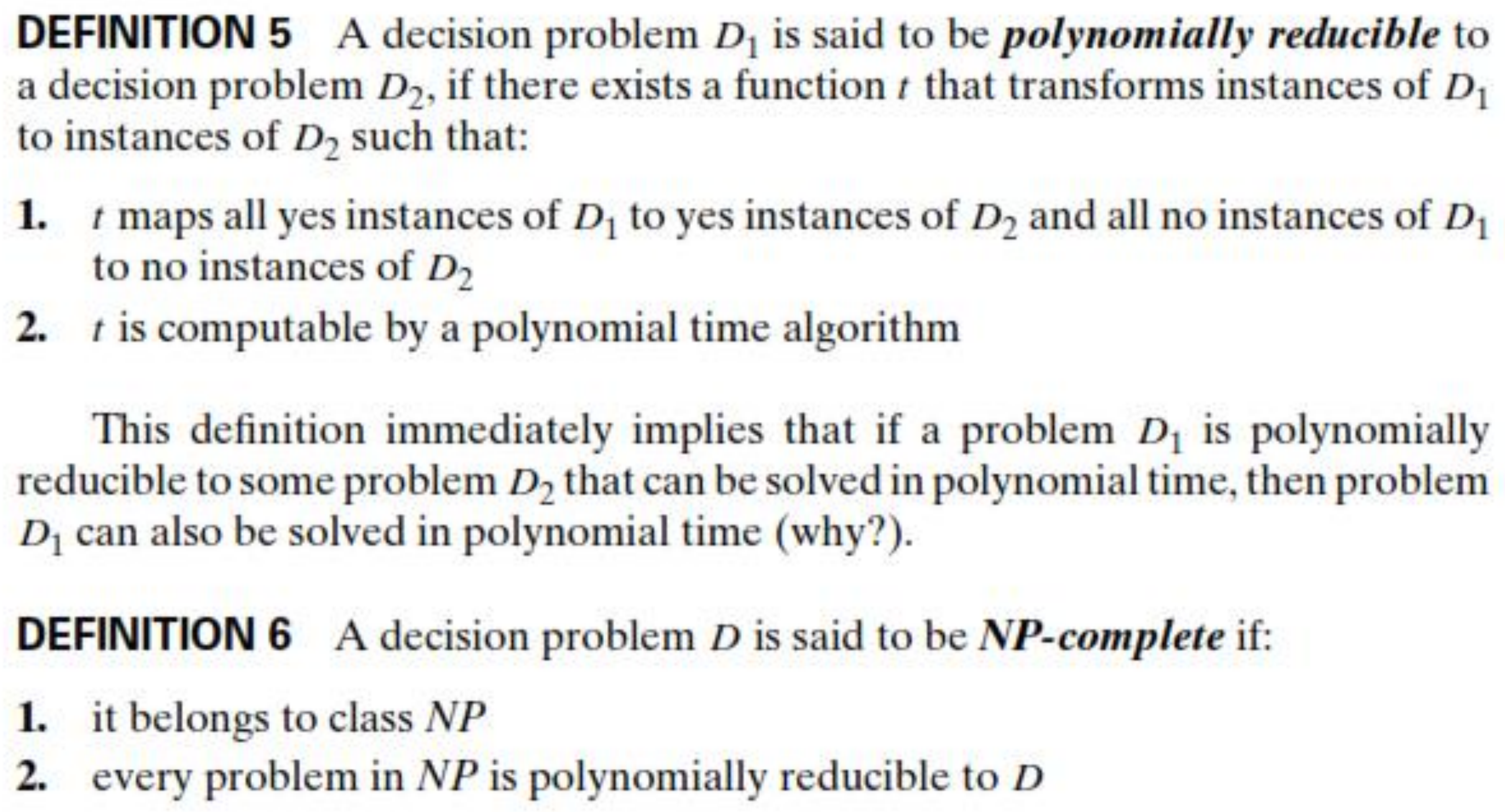


Figure 9.1 The maximal clique is $\{v_1, v_2, v_4, v_5\}$.

NP Complete



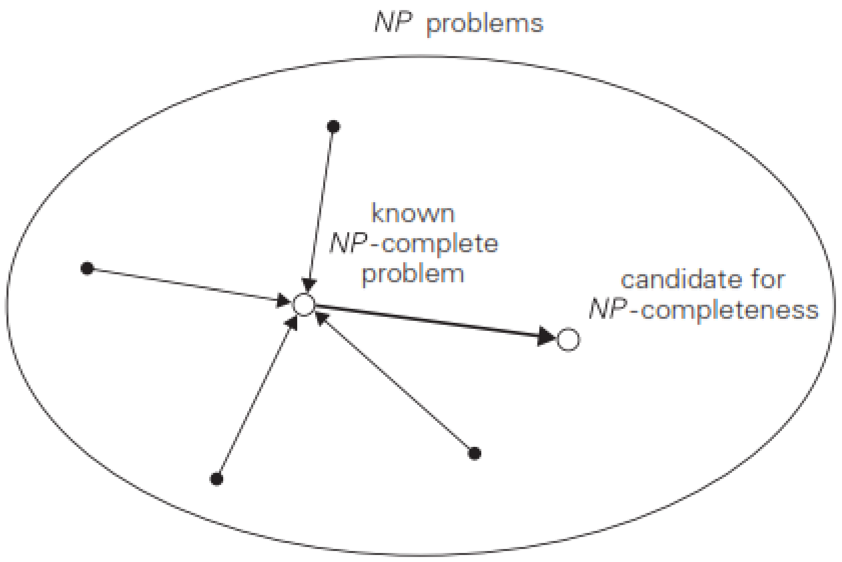
DEFINITION 5 A decision problem D_1 is said to be ***polynomially reducible*** to a decision problem D_2, if there exists a function t that transforms instances of D_1 to instances of D_2 such that:

1. t maps all yes instances of D_1 to yes instances of D_2 and all no instances of D_1 to no instances of D_2
2. t is computable by a polynomial time algorithm

This definition immediately implies that if a problem D_1 is polynomially reducible to some problem D_2 that can be solved in polynomial time, then problem D_1 can also be solved in polynomial time (why?).

DEFINITION 6 A decision problem D is said to be ***NP-complete*** if:

1. it belongs to class *NP*
2. every problem in *NP* is polynomially reducible to D


NP problems
known *NP*-complete problem
candidate for *NP*-completeness

Approximation Algorithms for the Traveling Salesman Problem

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

- *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

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:

accuracy ratio of an approximate solution s_a

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

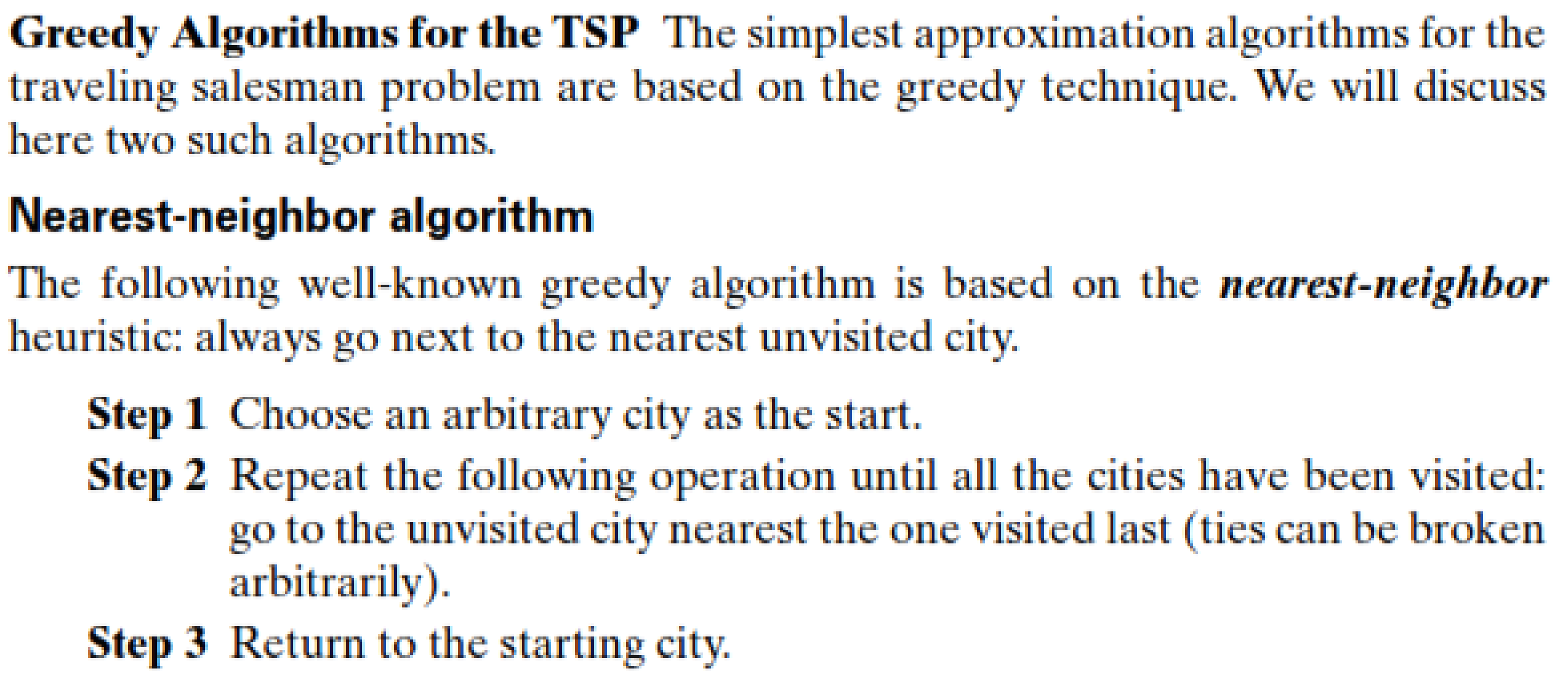
$$r(s_a) = f(s^*) / f(s_a) \quad \text{for maximization problems}$$

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^*

performance ratio of the algorithm A

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

Nearest-Neighbor Algorithm for TSP



Greedy Algorithms for the TSP The simplest approximation algorithms for the traveling salesman problem are based on the greedy technique. We will discuss here two such algorithms.

Nearest-neighbor algorithm

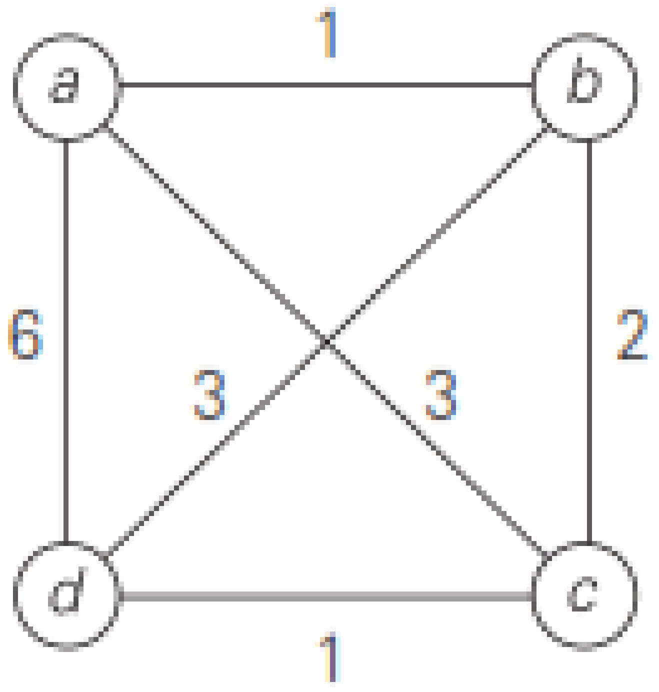
The following well-known greedy algorithm is based on the ***nearest-neighbor*** heuristic: always go next to the nearest unvisited city.

Step 1 Choose an arbitrary city as the start.

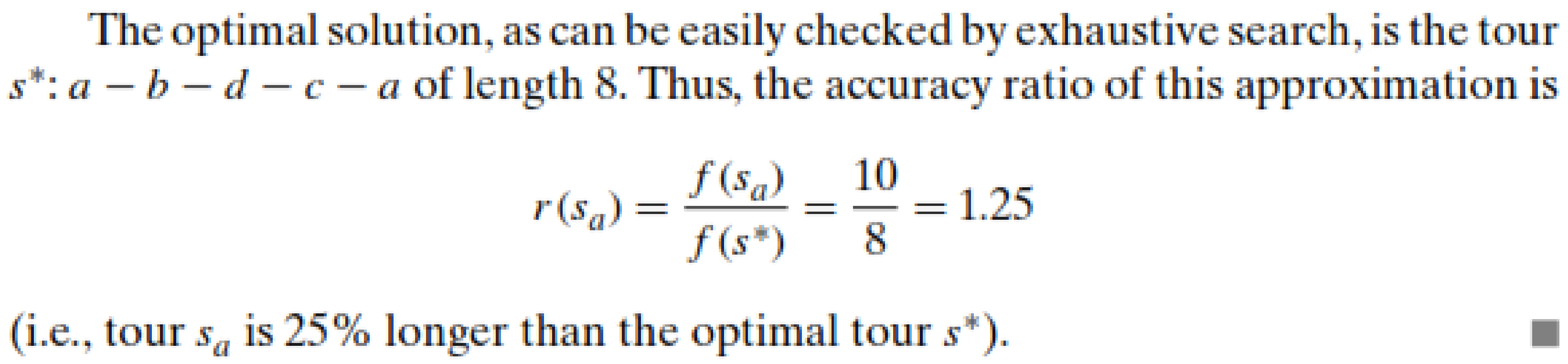
Step 2 Repeat the following operation until all the cities have been visited: go to the unvisited city nearest the one visited last (ties can be broken arbitrarily).

Step 3 Return to the starting city.

Example



- Sa : a - b - c - d - a of length 10.



The optimal solution, as can be easily checked by exhaustive search, is the tour $s^*: a - b - d - c - a$ of length 8. Thus, the accuracy ratio of this approximation is

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{10}{8} = 1.25$$

(i.e., tour s_a is 25% longer than the optimal tour s^*). ■

Minimum-Spanning-Tree–Based Algorithms

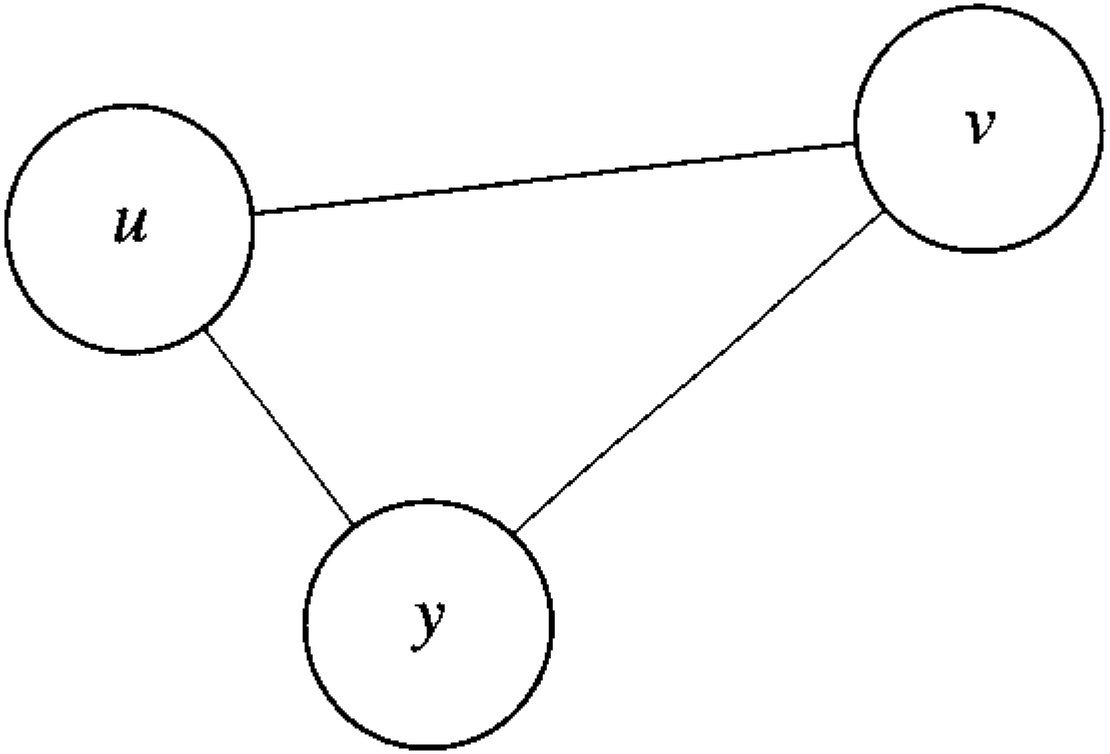
1. Determine a minimum spanning tree.
2. Create a path that visits every city by going twice around the tree.
3. Create a path that does not visit any vertex twice (that is, a tour) by taking shortcuts.

Traveling Salesperson Problem with Triangular Inequality

Let a weighted, undirected graph $G = (V, E)$ be given such that

1. there is an edge connecting every two distinct vertices
2. if $W(u, v)$ denotes the weight on the edge connecting vertex u to vertex v, then, for every other vertex y,

$$W(u, v) \leq W(u, y) + W(y, v).$$

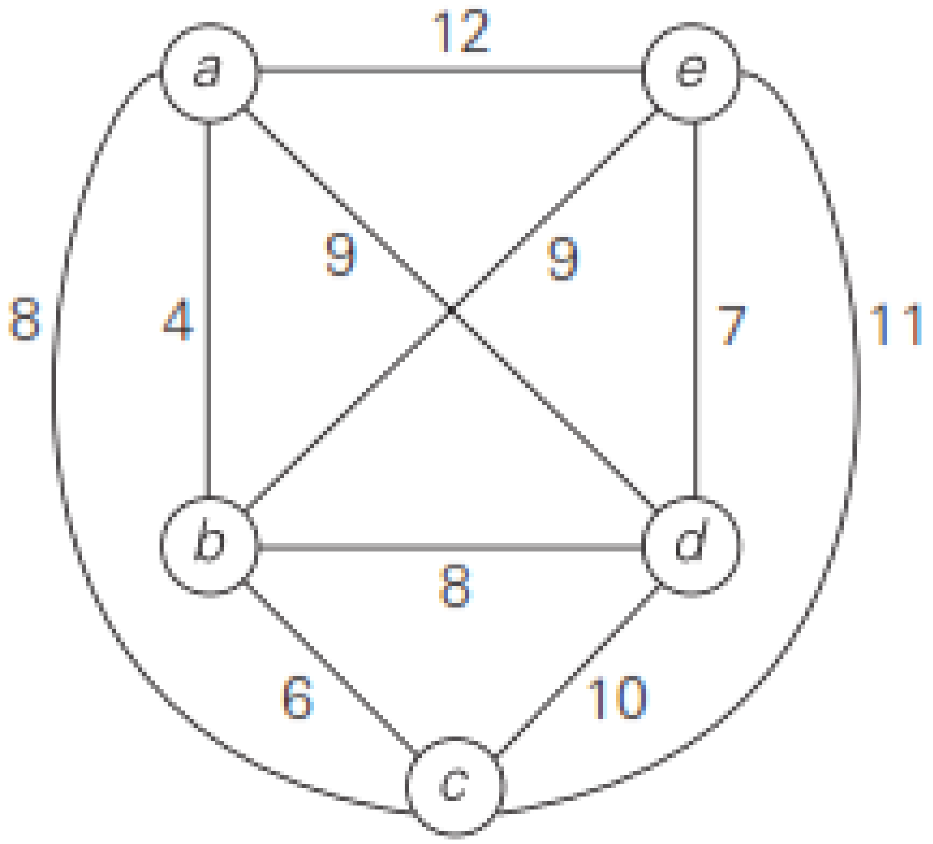

u
v
y

Twice-around-the-tree algorithm

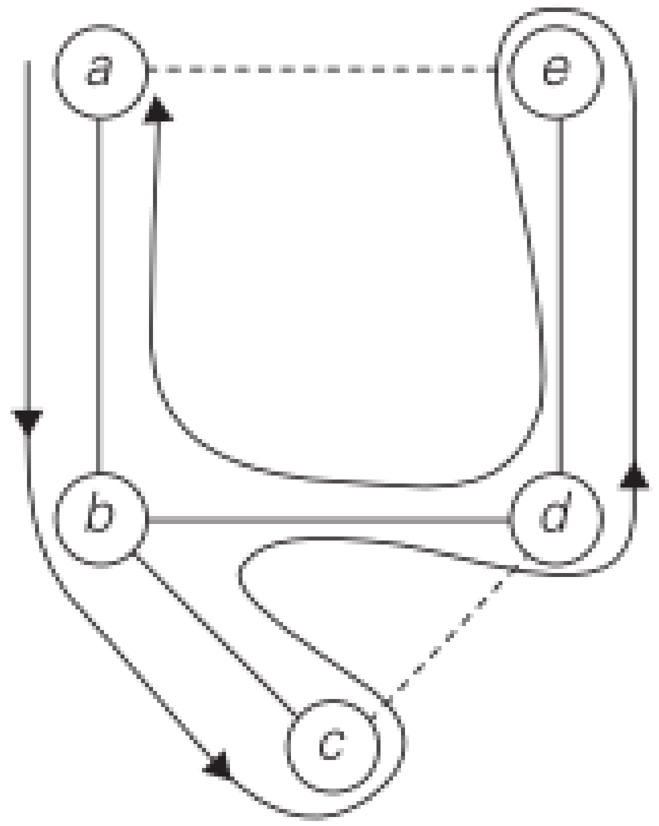
Step 1 Construct a minimum spanning tree of the graph corresponding to a given instance of the traveling salesman problem.

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

Step 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.


a
e
b
d
c
12
8
4
9
9
7
11
8
6
10

Walk around

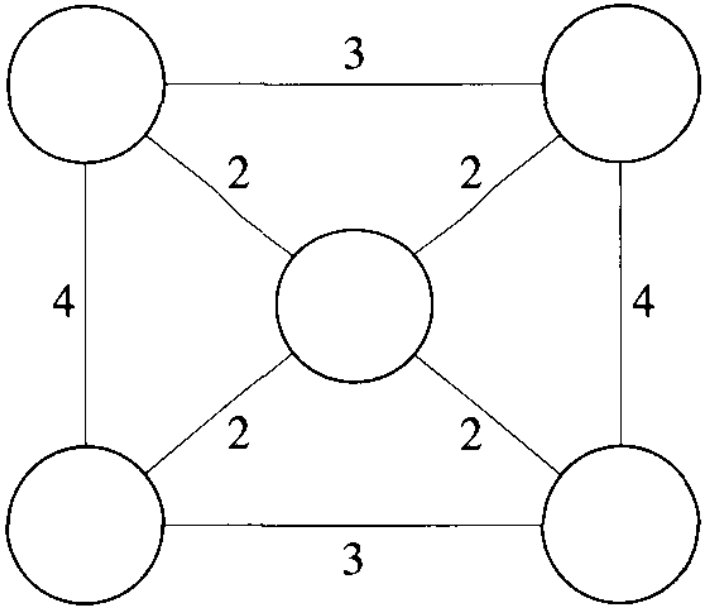


$$a, \quad b, \quad c, \quad b, \quad d, \quad e, \quad d, \quad b, \quad a.$$

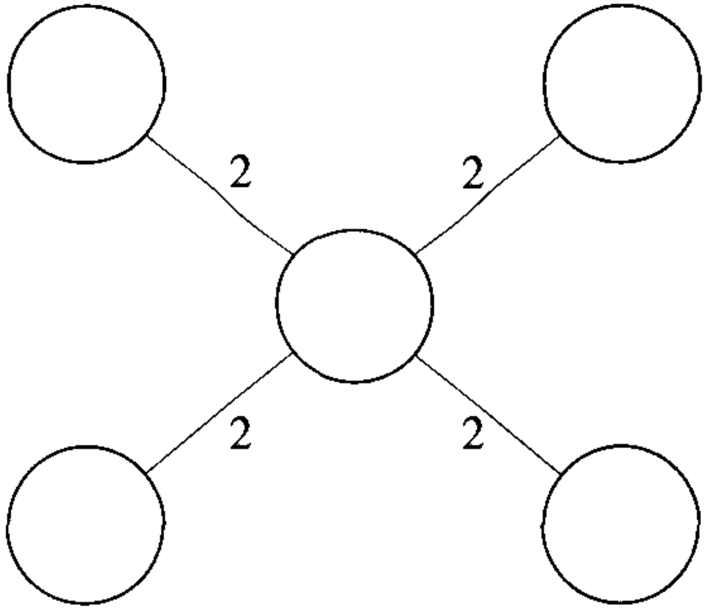
Eliminating the second b (a shortcut from c to d), the second d, and the third b (a shortcut from e to a) yields the Hamiltonian circuit

$$a, \quad b, \quad c, \quad d, \quad e, \quad a$$

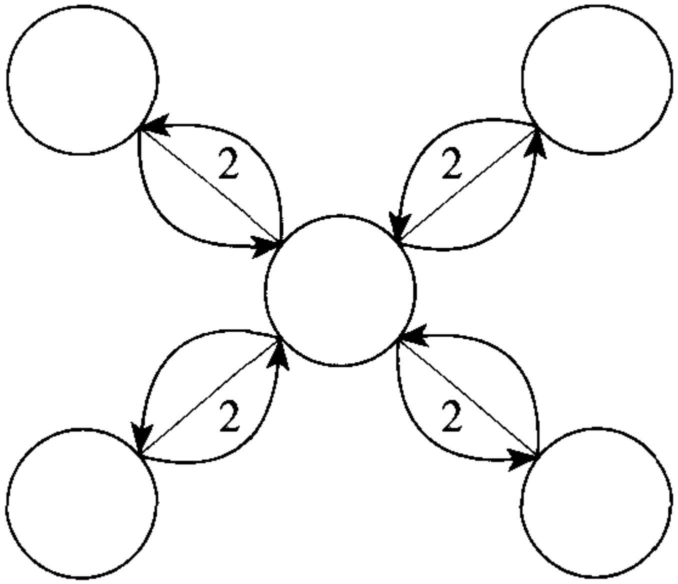
of length 39. ■



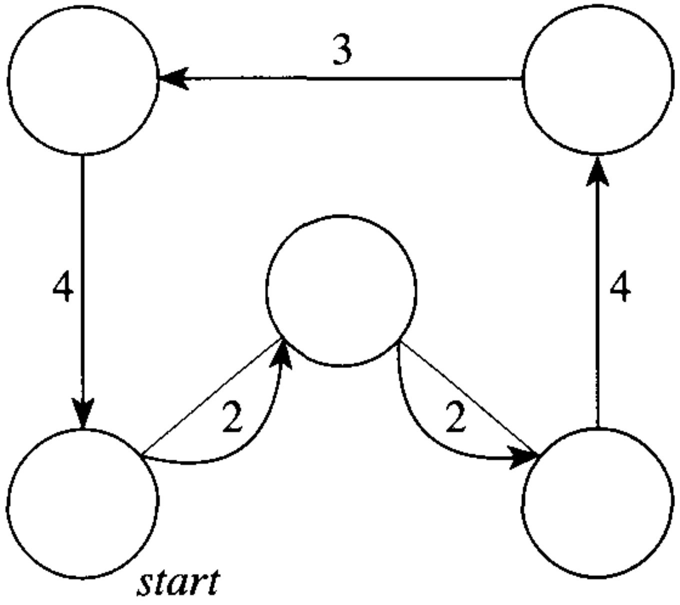
(a) A weighted, undirected graph



(b) A minimum spanning tree



(c) Twice around the tree



(d) A circuit with shortcuts

Apply the nearest-neighbor algorithm to the instance defined by the inter-city distance matrix below. Start the algorithm at the first city, assuming that the cities are numbered from 1 to 5.

$$\begin{bmatrix} 0 & 14 & 4 & 10 & \infty \\ 14 & 0 & 5 & 8 & 7 \\ 4 & 5 & 0 & 9 & 16 \\ 10 & 8 & 9 & 0 & 32 \\ \infty & 7 & 16 & 32 & 0 \end{bmatrix}$$

a. The nearest-neighbor algorithm yields the tour $1 - 3 - 2 - 5 - 4 - 1$ of length 58.

Greedy Algorithm for Knapsack Problem

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

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

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

capacity $W = 10$

(a)

subset	added items	value
∅	1, 3, 4	$69
{1}	3, 4	$69
{2}	4	$46
{3}	1, 4	$69
{4}	1, 3	$69
{1, 2}	not feasible	
{1, 3}	4	$69
{1, 4}	3	$69
{2, 3}	not feasible	
{2, 4}		$46
{3, 4}	1	$69

(b)