

UE19CS251

DESIGN AND ANALYSIS OF ALGORITHMS

UNIT 5: Limitations of Algorithmic Power and
Coping with the Limitations

Dynamic Programming

PES University

Outline

Concepts covered

- Dynamic Programming
 - Introduction
 - Fibonacci numbers
 - Binomial Coefficients

1 Introduction

Dynamic Programming is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems

- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS
- “Programming” here means “planning”
- Main idea:
 - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
 - solve smaller instances once
 - record solutions in a table
 - extract solution to the initial instance from that table

2 Example: Fibonacci Numbers

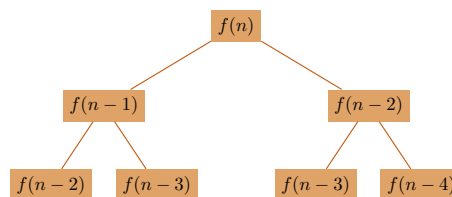
- Recall definition of Fibonacci numbers:

$$f(n) = f(n-1) + f(n-2)$$

$$f(0) = 0$$

$$f(1) = 1$$

- <2-> Computing the n^{th} Fibonacci number recursively (top-down):



3 Example: Fibonacci Numbers

Computing the n^{th} Fibonacci number using bottom-up iteration and recording results:

$$f(0) = 0$$

$$f(1) = 1$$

$$f(2) = 0 + 1 = 1$$

$$f(3) = 1 + 1 = 2$$

$$f(4) = 1 + 2 = 3$$

\vdots

Efficiency:

- time: $\Theta(n)$
- space: $\Theta(n)$ or $\Theta(1)$

4 Algorithm Examples

- Computing a binomial coefficient
- Warshall's algorithm for transitive closure
- Floyd's algorithm for all-pairs shortest paths
- Constructing an optimal binary search tree
- Some instances of difficult discrete optimization problems:
 - traveling salesman
 - knapsack

5 Binomial Coefficient

- Binomial coefficients are coefficients of the binomial formula:

$$(a + b)^n = C(n, 0)a^n b^0 + \dots + C(n, k)a^{n-k}b^k + \dots + C(n, n)a^0 b^n$$

- <2-> Recurrence:

$$C(n, k) = C(n - 1, k) + C(n - 1, k - 1) \quad \text{for } n > k > 0$$

$$C(n, 0) = 1, C(n, n) = 1 \quad \text{for } n \geq 0$$

- <3-> Value of $C(n, k)$ can be computed by filling a table:

	0	1	2	.	.	.	k-1	k
0	1							
1	1	1						
.								
.								
.								
n-1							$C(n-1, k-1)$	$C(n-1, k)$
n								$C(n, k)$

6 Binomial Coefficient Algorithm

Dynamic Programming Binomial Coefficient Algorithm

```

1: procedure BINOMIAL( $n, k$ )
2:   ▷ Input: Integers  $n \geq 0, k \geq 0$ 
3:   ▷ Output:  $C(n, k)$ 
4:   for  $i \leftarrow 0$  to  $n$  do
5:     for  $j \leftarrow 0$  to  $\min(i, k)$  do
6:       if  $j=0$  or  $j=i$  then
7:          $C(i, j) \leftarrow 1$ 
8:       else  $C[i, j] = C[i - 1, j] + C[i - 1, j - 1]$ 
9:   return  $C[n, k]$ 

```

- <2-> Time: $\Theta(nk)$
- <2-> Space: $\Theta(nk)$

7 Think About It

- What does dynamic programming have in common with divide-and-conquer? What is a principal difference between them?
- <2-> The coin change problem does not have an optimal greedy solution in all cases (ex: coins 1,20,25 and amount 40). Is there a dynamic programming based algorithm that can solve all cases of the coin change problem?

UE19CS251
DESIGN AND ANALYSIS OF ALGORITHMS
UNIT 5: Limitations of Algorithmic Power and
Coping with the Limitations

The Knapsack Problem

PES University

Outline

Concepts covered

- The Knapsack Problem
 - Introduction
 - Recurrence
 - Example

1 Problem Definition

- Given
 - n items of integer weights : $w_1 \quad w_2 \quad \dots \quad w_n$
values : $v_1 \quad v_2 \quad \dots \quad v_n$
 - knapsack of capacity W (integer $W > 0$)
- Find the most valuable subset of items such that sum of their weights does not exceed W

2 Knapsack Recurrence

- To design a dynamic programming algorithm, we need to derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller subinstances

- Consider the smaller knapsack problem where number of items is i ($i \leq n$) and the knapsack capacity is j ($j \leq W$)
- <2-> Then

$$F(i, j) = \begin{cases} \max(F(i-1, j), v_i + F(i-1, j - w_i)) & \text{if } j - w_i \geq 0 \\ F(i-1, j) & \text{if } j - w_i < 0 \end{cases}$$

3 Example

$$F(i, j) = \begin{cases} \max(F(i-1, j), v_i + F(i-1, j - w_i)) & \text{if } j - w_i \geq 0 \\ F(i-1, j) & \text{if } j - w_i < 0 \end{cases}$$

Dynamic Programming Example

item i	weight w_i	value v_i
1	2	12
2	1	10
3	3	20
4	2	15

What is the maximum value that can be stored in a knapsack of capacity 5?

	capacity j				
i	1	2	3	4	5
1	0	12	12	12	12
2	10	12	22	22	22
3	10	12	22	30	32
4	10	15	25	30	37

Given above 6 items, maximum value that can be stored in a knapsack of capacity 5 is **37**

4 Complexity

- Space complexity: $\Theta(nW)$
- Time complexity: $\Theta(nW)$
- Time to compose optimal solution: $O(n)$

5 Think About It

- <2-> Write pseudocode of the bottom-up dynamic programming algorithm for the knapsack problem
- <3-> True or False:
 1. <3-> A sequence of values in a row of the dynamic programming table for the knapsack problem is always nondecreasing?
 2. <4-> A sequence of values in a column of the dynamic programming table for the knapsack problem is always nondecreasing?

UE19CS251
DESIGN AND ANALYSIS OF ALGORITHMS
UNIT 5: Limitations of Algorithmic Power and
Coping with the Limitations

Memory Function Knapsack

PES University

Concepts covered

- Memory Function Knapsack
 - Motivation
 - Algorithm
 - Example

1 Bottom Up Approach

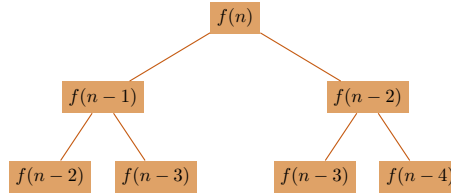
- Advantage of bottom up approach: each value computed only once
- Example computed bottom up:

	capacity j				
i	1	2	3	4	5
1	0	12	12	12	12
2	10	12	22	22	22
3	10	12	22	30	32
4	10	15	25	30	37

- Disadvantage of bottom up approach: values not required also computed

2 Top Down Approach

- Disadvantage of top down approach: same problem solved multiple times
- Example computed top down:



- Advantage of top down approach: only the required subproblems solved

3 Memory Function Dynamic Programming

- Combine the advantages of bottom up and top down approaches:
 - compute each subproblem only once
 - compute only the required subproblems

4 MF-DP Algorithm

Algorithm for Memory Function Dynamic Programming

```

1: procedure MFKNAPSACK( $i, j$ )
2:   ▷ Inputs:  $i$  indicating the number items, and
3:   ▷  $j$ , indicating the knapsack capacity
4:   ▷ Output: The value of an optimal feasible subset of the first  $i$  items
5:   ▷ Note: Uses global variables input arrays  $Weights[1 \dots n]$ ,
6:   ▷ and table  $F[0 \dots n, 0 \dots W]$  whose entries are initialized with  $-1$ 's
   except
7:   ▷ row 0 and column 0 is initialized with 0
8:   if  $F[i, j] < 0$  then
9:     if  $j < Weights[i]$  then
10:       $value \leftarrow MFKnapsack(i - 1, j)$ 
11:    else  $value \leftarrow \max(MFKnapsack(i - 1, j), \quad Values[i] +$ 
12:       $MFKnapsack(i - 1, j - Weights[i]))$ 
13:       $F[i, j] \leftarrow value$ 
14:   return  $F[i, j]$ 

```

5 Example

$$F(i, j) = \begin{cases} \max(F(i-1, j), \quad v_i + F(i-1, j - w_i)) & \text{if } j - w_i \geq 0 \\ F(i-1, j) & \text{if } j - w_i < 0 \end{cases}$$

Dynamic Programming Example

item i	weight w_i	value v_i
1	2	12
2	1	10
3	3	20
4	2	15

What is the maximum value that can be stored in a knapsack of capacity 5?

i	capacity j					
	0	1	2	3	4	5
0	0	□0	□0	□0	□0	□0
1	□0	□0	□12	□12	□12	□12
2	□0	–	□12	□22	–	□22
3	□0	–	–	□22	–	□32
4	□0	–	–	–	–	□37

Knapsack problem solved by

- computing 21 out 30 possible subproblems
- reusing subproblem entry (1, 2)

6 Complexity

- Constant factor improvement in efficiency
 - Space complexity: $\Theta(nW)$
 - Time complexity: $\Theta(nW)$
 - Time to compose optimal solution: $O(n)$
- Bigger gains possible where computation of a subproblem takes more than constant time

7 Think About It

- Consider the use of the MF technique to compute binomial coefficient using the recurrence

$$C(n, k) = C(n-1, k-1) + C(n-1, k)$$

- How many table entries are filled?
- How many are reused?

UE19CS251

DESIGN AND ANALYSIS OF ALGORITHMS

UNIT 5: Limitations of Algorithmic Power and
Coping with the Limitations

Transitive Closure (Warshall's
Algorithm)

PES University

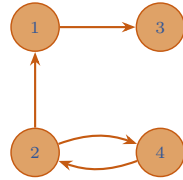
Outline

Concepts covered

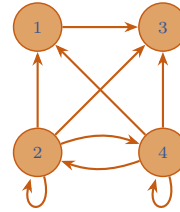
- Transitive Closure (Warshall's Algorithm)
 - Motivation
 - Algorithm
 - Example

1 Transitive Closure

- Computes the transitive closure of a relation
- Alternatively: existence of all nontrivial paths in a digraph (directed graph)
- Example of transitive closure:



$$\begin{array}{c}
 \begin{array}{cccc}
 & 1 & 2 & 3 & 4 \\
 \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}
 \end{array}
 \end{array}$$



$$\begin{array}{c}
 \begin{array}{cccc}
 & 1 & 2 & 3 & 4 \\
 \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}
 \end{array}
 \end{array}$$

2 Warshall's Algorithm

- Constructs transitive closure T as the last matrix in the sequence of $n \times n$ matrices $R^{(0)}, \dots, R^{(k)}, \dots, R^{(n)}$ where $R^{(k)}[i, j] = 1$ iff there is nontrivial path from i to j with only first k vertices allowed as intermediate vertices
 - $R^{(0)} = A$ (adjacency matrix), $R^{(n)} = T$ (transitive closure)
- On the k^{th} iteration, the algorithm computes $R^{(k)}$

$$R^{(k)}[i, j] = \begin{cases} 1 & \text{if path from } i \text{ to } k \text{ and } k \text{ to } j, \text{ i.e., } R^{(k-1)}[i, k] = R^{(k-1)}[k, j] = 1 \\ R^{(k-1)}[i, j] & \text{otherwise} \end{cases}$$

$$R^{(k)}[i, j] = R^{(k-1)}[i, j] \text{ or } (R^{(k-1)}[i, k] \text{ and } R^{(k-1)}[k, j])$$

3 Algorithm

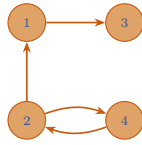
Transitive Closure (Warshall's Algorithm)

```

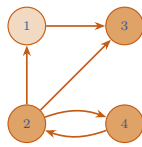
1: procedure WARSHALL( $A[1 \dots n, 1 \dots n]$ )
2:    $\triangleright$  Input: The adjacency matrix  $A$  of a digraph with  $n$  vertices
3:    $\triangleright$  Output: The transitive closure of the digraph
4:    $R^{(0)} \leftarrow A$ 
5:   for  $k \leftarrow 1$  to  $n$  do
6:     for  $i \leftarrow 1$  to  $n$  do
7:       for  $j \leftarrow 1$  to  $n$  do
8:          $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j] \text{ or } (R^{(k-1)}[i, k] \text{ and } R^{(k-1)}[k, j]);$ 
9:   return  $R^{(n)}$ 

```

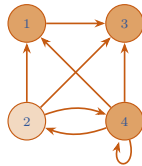
4 Warshall's Algorithm



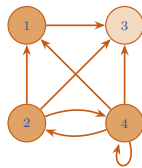
	1	2	3	4
1	0	0	1	0
2	1	0	0	1
3	0	0	0	0
4	0	1	0	0



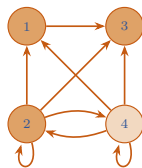
	1	2	3	4
1	0	0	1	0
2	1	0	1	1
3	0	0	0	0
4	0	1	0	0



	1	2	3	4
1	0	0	1	0
2	1	0	1	1
3	0	0	0	0
4	1	1	1	1



	1	2	3	4
1	0	0	1	0
2	1	0	1	1
3	0	0	0	0
4	1	1	1	1



	1	2	3	4
1	0	0	1	0
2	1	1	1	1
3	0	0	0	0
4	1	1	1	1

5 Think About It

- Is Warshall's algorithm efficient for sparse graphs? Why / why not?
- Can Warshall's algorithm be used to determine if a graph is a DAG (Directed Acyclic Graph)? If so, how?

UE19CS251

DESIGN AND ANALYSIS OF ALGORITHMS
UNIT 5: Limitations of Algorithmic Power and
Coping with the Limitations

All Pairs Shortest Path (Floyd's Algorithm)

PES University

Outline

Concepts covered

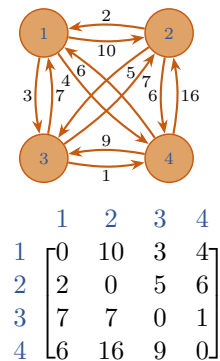
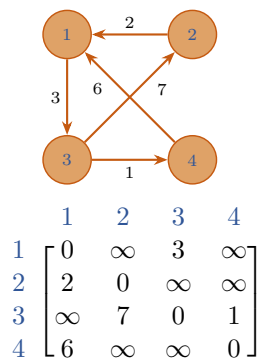
- All Pairs Shortest Path (Floyd's Algorithm)
 - Definition
 - Algorithm
 - Example

1 Problem Definition

- Given an undirected or directed graph, with weighted edges, find the shortest path between every pair of vertices
 - Dijkstra's algorithm found shortest paths from given vertex to remaining $n - 1$ vertices ($\Theta(n)$ paths)
 - Current problem is to find the shortest path between every pair of vertices ($\Theta(n^2)$ paths)
- Solution approach is similar to the transitive closure approach: Compute transitive closure via sequence of $n \times n$ matrices $R^{(0)}, \dots, R^{(k)}, \dots, R^{(n)}$ where $R^{(k)}[i, j] = 1$ iff there is nontrivial path from i to j with only first k vertices allowed as intermediate vertices
- Compute all pairs shortest paths via sequence of $n \times n$ matrices $D^{(0)}, \dots, D^{(k)}, \dots, D^{(n)}$ where $D^{(k)}[i, j]$ is the shortest path from i to j with only first k vertices allowed as intermediate vertices

2 Example

- Example of all pairs shortest paths:



3 Algorithm

Transitive Closure (Floyd's Algorithm)

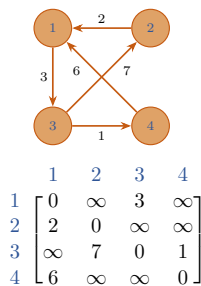
```

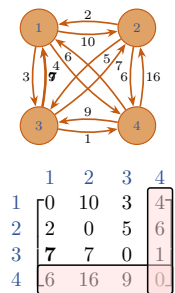
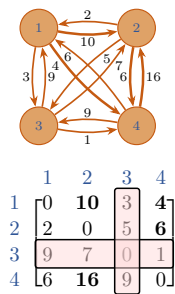
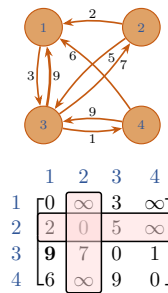
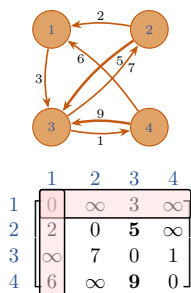
1: procedure FLOYD( $(A[1 \dots n, 1 \dots n])$ )
2:    $\triangleright$  Input: Weight matrix  $A$  of a graph with no negative length cycles
3:    $\triangleright$  Output: Distance matrix of shortest paths
4:    $D \leftarrow A$ 
5:   for  $k \leftarrow 1$  to  $n$  do
6:     for  $i \leftarrow 1$  to  $n$  do
7:       for  $j \leftarrow 1$  to  $n$  do
8:          $D[i, j] \leftarrow \min(D[i, j], D[i, k] + D[k, j])$ 
9:   return  $D$ 

```

- Complexity: $\Theta(n^3)$

4 Example





5 Think About It

- Give an example of a graph with negative weights for which Floyd's algorithm does not yield the correct result
- Enhance Floyd's algorithm so that shortest paths themselves, not just their lengths, can be found

UE19CS251
DESIGN AND ANALYSIS OF ALGORITHMS
UNIT 5: Limitations of Algorithmic Power and
Coping with the Limitations

Lower-Bound Arguments

PES University

Outline

Concepts covered

- Lower-Bound Arguments
 - Trivial lower bounds
 - Adversary arguments
 - Problem reduction

1 Limitations of Algorithmic Power

- There are no algorithms to solve some problems
 - Ex: halting problem
- Other problems can be solved algorithmically, but not in polynomial time
 - Ex: traveling salesman problem
- For polynomial time algorithms also, there are lower bounds

2 Definition

Lower Bound

An estimate on a minimum amount of work needed to solve a given problem (estimate can be less than the minimum amount of work but not greater)

- 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 $n \times n$ matrices

3 Bound Tightness

- A 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 a sorted array	$\Omega(\log n)$	yes
Element uniqueness	$\Omega(n \log n)$	yes
Integer multiplication ($n \times n$)	$\Omega(n)$	unknown
Matrix multiplication ($n \times n$)	$\Omega(n^2)$	unknown

4 Methods for Establishing Lower Bounds

- Trivial lower bounds
- Information-theoretic arguments (decision trees)
- Adversary arguments
- Problem reduction

5 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

6 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 $a_1 < a_2 < \dots < a_n$ and $b_1 < b_2 < \dots < b_n$
 - Adversary: $a_i < b_j$ iff $i < j$
Output $b_1 < a_1 < b_2 < a_2 < \dots < b_n < a_n$ requires $2n - 1$ comparisons of adjacent elements

7 Problem Reduction

- Basic 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 $\Omega(n \log n)$)

8 Think About It

- Prove that the classic recursive algorithm for the Tower of Hanoi puzzle makes the minimum number of disk moves
- Find a trivial lower-bound class and indicate if the bound is tight:
 - finding the largest element in an array
 - generating all the subsets of an n -element set
 - determining whether n given real numbers are all distinct

UE19CS251

DESIGN AND ANALYSIS OF ALGORITHMS

UNIT 5: Limitations of Algorithmic Power and
Coping with the Limitations

Decision Trees

PES University

Outline

Concepts covered

- Decision Trees
 - Smallest of three numbers
 - Sorting
 - Searching

1 Problem Types: Optimization and Decision

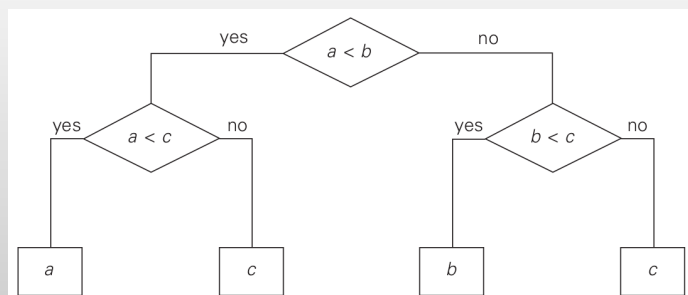
- Optimization problem: find a solution that maximizes or minimizes some objective function
- Decision problem: answer yes/no to a question
- Many problems have decision and optimization versions
 - Ex: traveling salesman problem
 - optimization: find Hamiltonian cycle of minimum length
 - decision: find Hamiltonian cycle of length m
- Decision problems are more convenient for formal investigation of their complexity

2 Introduction

- Many important algorithms, especially those for sorting and searching, work by comparing items of their inputs
- We can study the performance of such algorithms with a device called the decision tree

3 Example: Decision tree for minimum of three numbers

Decision tree for a determining the minimum of three numbers



4 Central Idea

- The central idea behind this model lies in the observation that a tree with a given number of leaves, which is dictated by the number of possible outcomes, has to be tall enough to have that many leaves
- Specifically, it is not difficult to prove that for any binary tree with l leaves and height h

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

- A binary tree of height h with the largest number of leaves has all its leaves on the last level
- Hence, the largest number of leaves in such a tree is 2^h
- In other words, $2^h \geq l$ which implies $h \geq \lceil \log_2 l \rceil$

5 Decision Trees for Sorting Algorithms

- Most sorting algorithms are comparison-based, i.e., they work by comparing elements in a list to be sorted
- By studying properties of binary decision trees, for comparison-based sorting algorithms, we can derive important lower bounds on time efficiencies of such algorithms
- We can interpret an 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
- For example, for the outcome $a < c < b$ obtained by sorting a list a, b, c
- The number of possible outcomes for sorting an arbitrary n -element list is equal to $n!$

6 Decision Trees for Sorting Algorithms

- The height of a binary decision tree for any comparison-based sorting algorithm and hence the worst -case number of comparisons made by such an algorithm cannot be less than

$$C_{worst}(n) \geq \lceil \log_2 n! \rceil$$

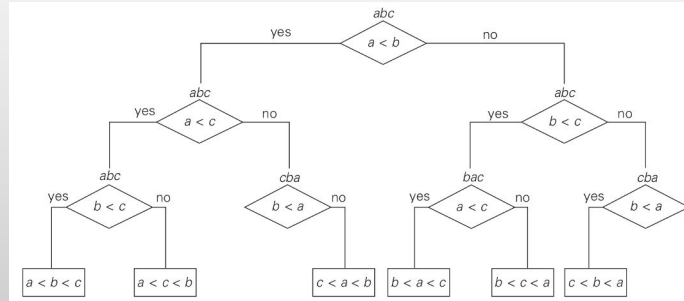
- Using Stirling's formula:

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

- About $n \log_2 n$ comparisons are necessary to sort an arbitrary n -element list by any comparison-based sorting algorithm

7 Decision Trees for Sorting Algorithms

Decision Tree for Three Element Selection Sort



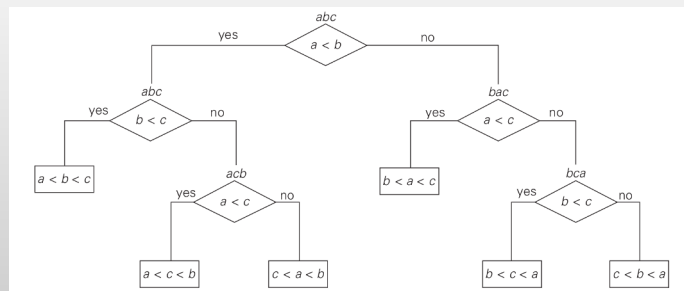
8 Decision Trees for Sorting Algorithms

- We can also use decision trees for analyzing the average-case behavior of a comparison based sorting algorithm
- 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
- For example, for the three-element insertion sort this number is:

$$\frac{2 + 3 + 3 + 2 + 3 + 3}{6} = 2\frac{2}{3}$$

9 Decision Trees for Sorting Algorithms

Decision Tree for Three Element Insertion Sort



10 Decision Trees for Sorting Algorithms

- Under the standard assumption that all $n!$ outcomes of sorting are equally likely, the following lower bound on the average number of

comparisons C_{avg} made by any comparison-based algorithm in sorting an n -element list has been proved

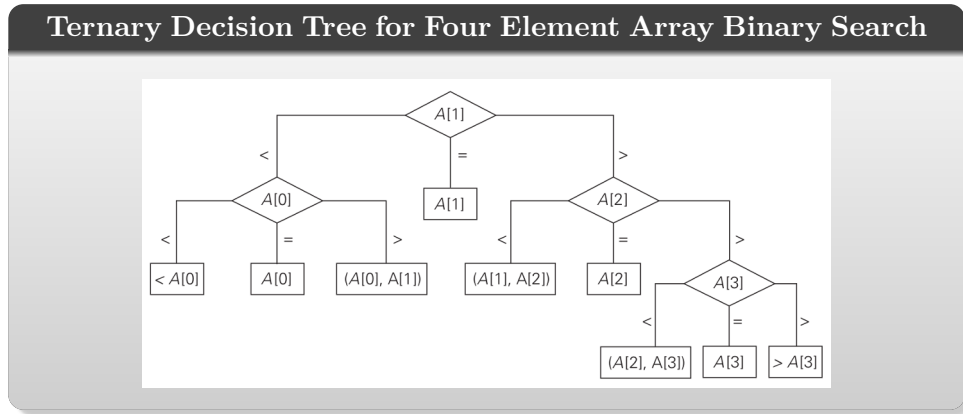
$$C_{avg}(n) \geq \log_2 n!$$

11 Decision Trees for Searching Algorithms

- Decision trees can be used for establishing lower bounds on the number of key comparisons in searching a sorted array of n keys: $A[0] < A[1] < \dots < A[n-1]$
- The number of comparisons made by binary search in the worst case:

$$C_{worst}^{bs}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil$$

12 Decision Trees for Searching Algorithms



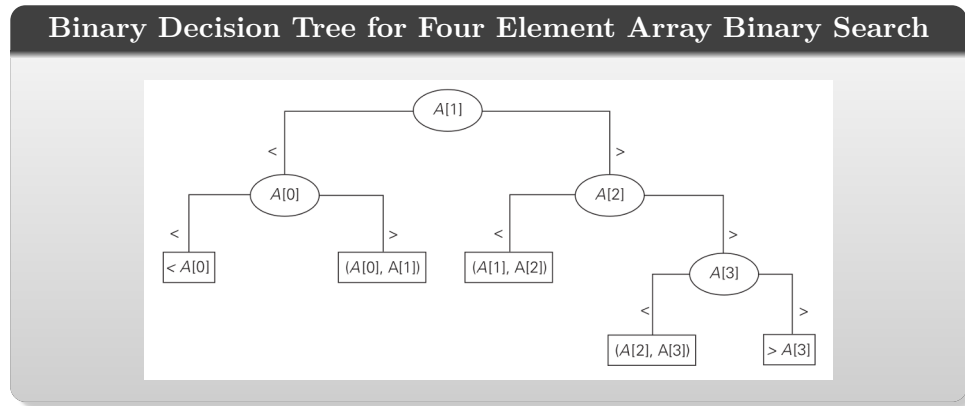
13 Decision Trees for Searching Algorithms

- 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)
- Since the minimum height h of a ternary tree with l leaves is $\text{floor}(\log_3 l)$, we get the following lower bound on the number of worst-case comparisons:

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

- This lower bound is smaller than $\lceil \log_2(n+1) \rceil$, the number of worst-case comparisons for binary search
- Can we prove a better lower bound, or is binary search far from being optimal?

14 Decision Trees for Searching Algorithms



- The binary decision tree is simply the ternary decision tree with all the middle subtrees eliminated

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

15 Think About It

- Consider the problem of finding the median of a three-element set a , b , c of orderable items
 - What is the information-theoretic lower bound for comparison-based algorithms solving this problem?
 - Draw a decision tree for an algorithm solving this problem
 - Is the above bound tight?

UE19CS251

DESIGN AND ANALYSIS OF ALGORITHMS

UNIT 5: Limitations of Algorithmic Power and
Coping with the Limitations

P, NP, and NP-Complete Problems

PES University

Outline

Concepts covered

- Class P
- Class NP
- NP-Complete
- NP-Hard

1 Classifying Problem Complexity

- Is the problem tractable, i.e., is there a polynomial-time ($O(p(n))$) algorithm that solves it?
- Possible answers:
 - yes
 - no
 - * because it's been proved that no algorithm exists at all (e.g., Turing's halting problem)
 - * because it's been proved that any algorithm takes exponential time
 - unknown

2 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
- Example: 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

3 Class P

Class P (Polynomial)

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

- searching
- element uniqueness
- graph connectivity
- graph acyclicity
- primality testing

4 Class NP

Class 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”

5 Example: CNF satisfiability

Boolean Satisfiability (CNF)

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: Consider the Boolean expression in CNF form:

$$(a + \bar{b} + \bar{c})(\bar{a} + b)(\bar{a} + \bar{b} + \bar{c})$$

- Can values *false* and *true* (or 0 and 1) be assigned to a , b and c such that above expression evaluates to 1?
- $a = 1, b = 1, c = 0$
- Checking phase: $\Theta(n)$

6 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 (but no guessing is necessary), so we have:

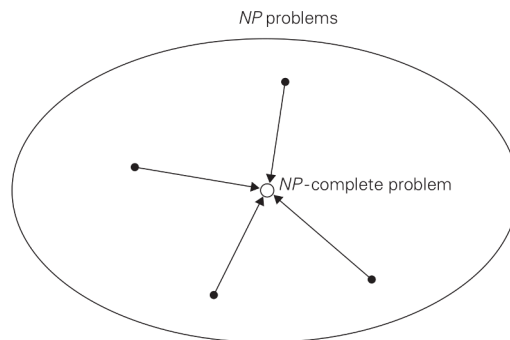
$$P \subseteq NP$$

- Big question:

$$P = NP ?$$

7 NP-Complete Problems

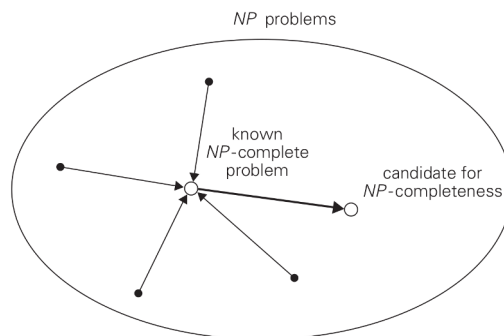
- A decision problem D is NP -complete if it's as hard as any problem in NP , i.e.,
 - D is in NP
 - every problem in NP is polynomial-time reducible to D



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

8 NP-Complete Problems

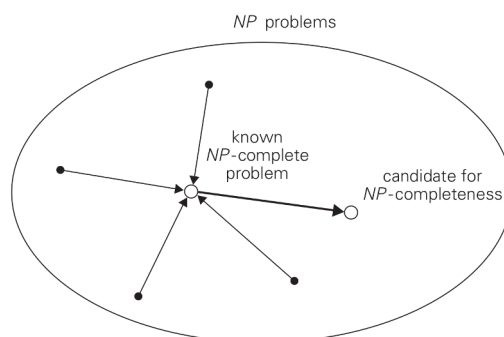
- Other NP-complete problems obtained through polynomial- time reductions from a known NP-complete problem



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

9 $P = NP$? Dilemma Revisited

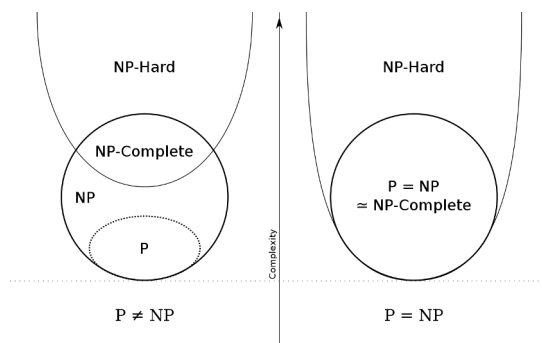
- $P = NP$ would imply that every problem in NP , including all NP -complete problems, could be solved in polynomial time
- If a polynomial-time algorithm for just one NP -complete problem is discovered, then every problem in NP can be solved in polynomial time, i.e., $P = NP$



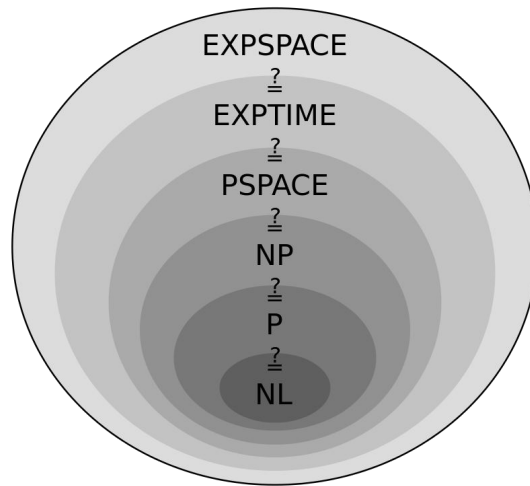
- Most but not all researchers believe that $P \neq NP$
 - Though others like Stephen Cook, Leonid Levin and Donald Knuth don't

10 NP-Hard Problems

- A decision problem D is NP -hard iff:
 - D is in NP
 - every problem in NP is polynomial-time reducible to D



11 Complexity Hierarchy



UE19CS251

DESIGN AND ANALYSIS OF ALGORITHMS

UNIT 5: Limitations of Algorithmic Power and
Coping with the Limitations

Backtracking

PES University

Outline

Concepts covered

- Backtracking
 - Introduction
 - N Queens
 - Hamiltonian Circuit
 - Subset Sum
 - Algorithm

1 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

2 Exact Solution Strategies

- Exhaustive search (brute force)

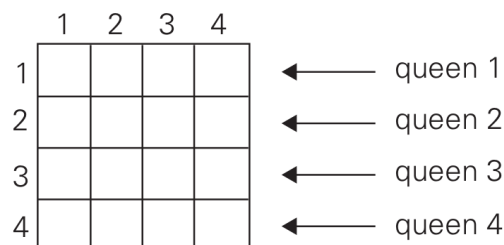
- useful only for small instances
- Dynamic programming
 - applicable to some problems (e.g., the knapsack problem)
- Backtracking
 - eliminates some unnecessary cases from consideration
 - yields solutions in reasonable time for many instances but worst case is still exponential
- Branch-and-bound
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

3 Introduction

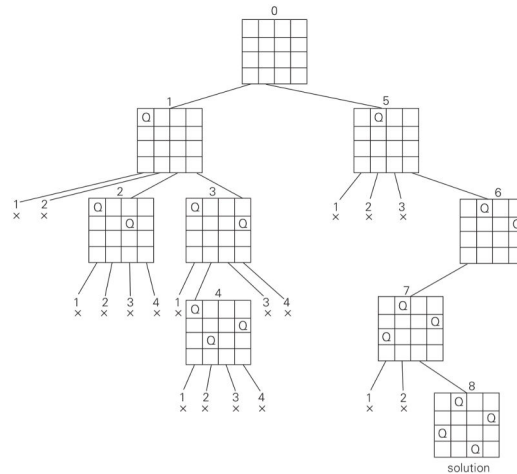
- Construct the *state-space tree*
 - nodes: partial solutions
 - edges: choices in extending partial solutions
- Explore the state space tree using depth-first search
- “Prune” *nonpromising nodes*
 - DFS stops exploring subtrees rooted at nodes that cannot lead to a solution and backtracks to such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

4 Example: N -Queens Problem

- Place N queens on an $N \times N$ chess board so that no two of them are in the same row, column, or diagonal



5 State-Space Tree of the 4-Queens Problem

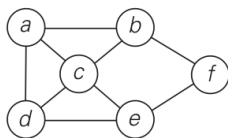


6 Example: Hamiltonian Circuit Problem

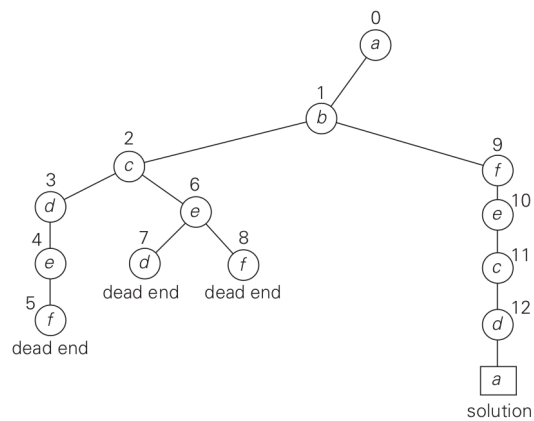
Hamiltonian Circuit

A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once.

- Example graph:



- State-space tree for finding a Hamiltonian circuit (numbers above the nodes of indicate the order in which the nodes are generated):

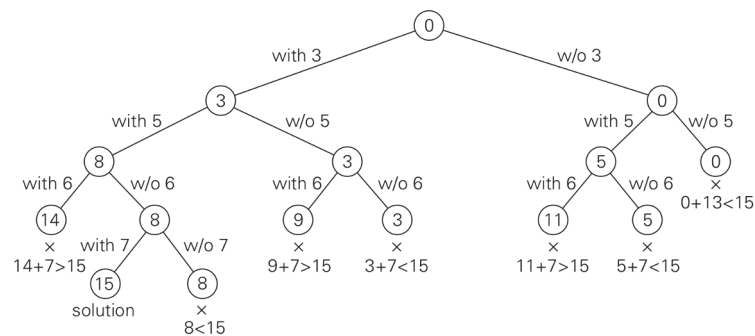


7 Example: Subset Sum Problem

Subset Sum Problem

Given set $A = \{a_1, \dots, a_n\}$ of n positive integers, find a subset whose sum is equal to a given positive integer d

- State space tree for $A = \{3, 5, 6, 7\}$ and $d = 15$ (number in each node is the sum so far):



8 Algorithm

Backtrack Algorithm

```

1: procedure BACKTRACK( $X[1 \dots i]$ )
2:   ▷ Input:  $X[1 \dots i]$  specifies first  $i$  promising components of a solution
3:   ▷ Output: All the tuples representing the problem's solutions
4:   if  $X[1 \dots i]$  is a solution then
5:     write  $X[1 \dots i]$ 
6:   else
7:     for each element  $x \in S_{i+1}$  consistent with  $X[1 \dots i]$  and the
       constraints do
8:        $X[i+1] \leftarrow x$ 
9:       Backtrack ( $X[1 \dots i+1]$ )

```

- Output: n -tuples (x_1, x_2, \dots, x_n)
- Each $x_i \in S_i$, some finite linearly ordered set

9 Think About It

- Continue the backtracking search for a solution to the four-queens problem, to find the second solution to the problem
- Explain how the board's symmetry can be used to find the second solution to the four-queens problem

UE19CS251

DESIGN AND ANALYSIS OF ALGORITHMS

UNIT 5: Limitations of Algorithmic Power and
Coping with the Limitations

Branch and Bound

PES University

Outline

Concepts covered

- Backtracking
 - General Approach
 - Knapsack Problem
 - Assignment Problem
 - Travelling Salesman Problem

1 Introduction

- 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

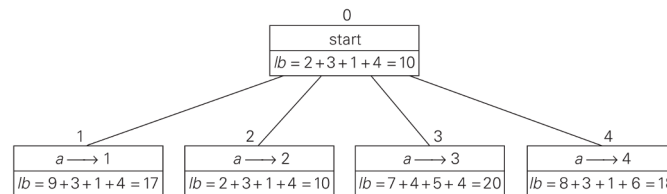
2 Example: Assignment Problem

- 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

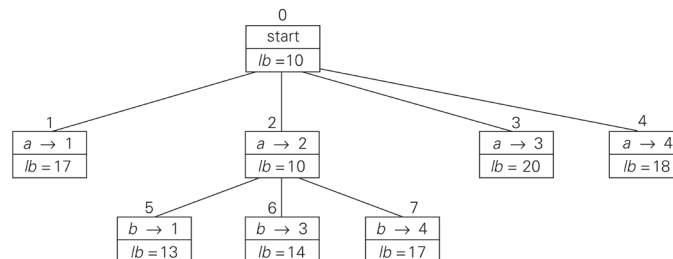
	Job 1	Job 2	Job 3	Job 4
Person a	9	2	7	8
Person b	6	4	3	7
Person c	5	8	1	8
Person d	7	6	9	4

- Lower bound (sum of smallest elements in each row): $2+3+1+4 = 10$
- Best-first branch-and-bound variation: Generate all the children of the most promising node

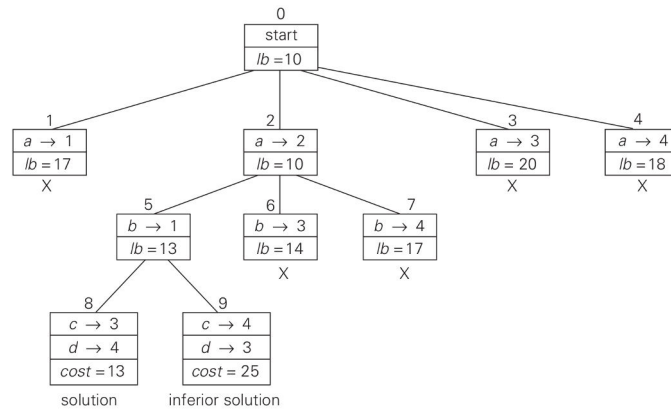
3 Example: First two levels of the state-space tree



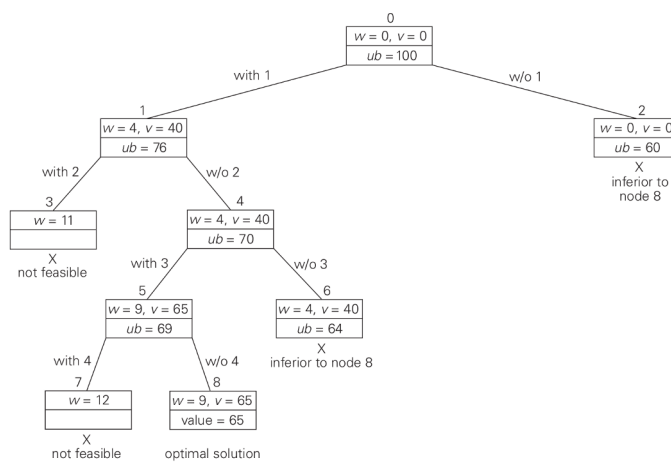
4 Example: First three levels of the state-space tree



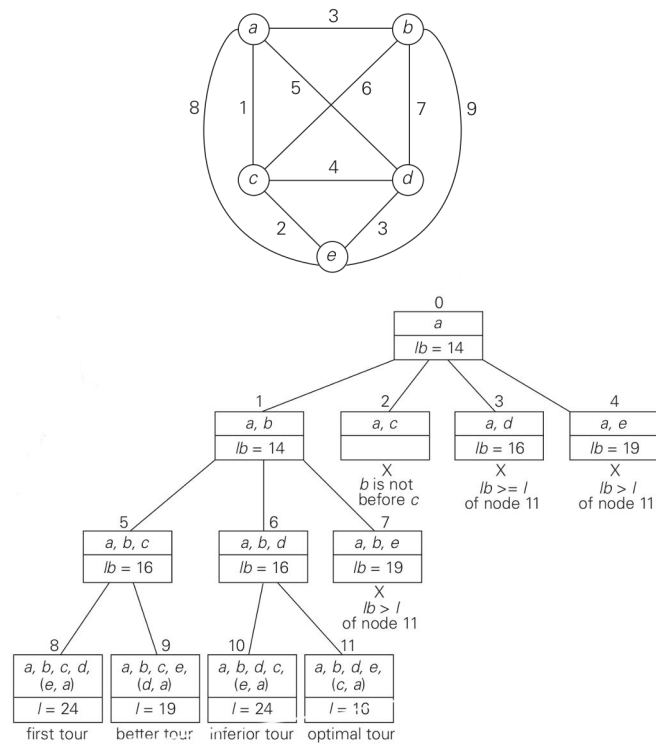
5 Example: Complete state-space tree



6 Example: Knapsack Problem



7 Example: Traveling Salesman Problem



8 Think About It

- What data structure would you use to keep track of live nodes in a best-first branch-and-bound algorithm?
- Solve the assignment problem by the best-first branch-and-bound algorithm with the bounding function based on matrix columns rather than rows