Design and Analysis of Algorithms

Dynamic Programming

Reetinder Sidhu

Department of Computer Science and Engineering

Design and Analysis of Algorithms

Dynamic Programming

Reetinder Sidhu

Department of Computer Science and Engineering

Dynamic Programming

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

PES UNIVERSITY ONLINE

- Dynamic Programming
  - **Computing a Binomial Coefficient**
  - The Knapsack Problem
  - Memory Functions
  - Warshall's and Floyd's Algorithms
- Limitations of Algorithmic Power
  - Lower-Bound Arguments
  - Decision Trees
  - P, NP, and NP-Complete, NP-Hard Problems
- Coping with the Limitations
  - Backtracking
  - Branch-and-Bound

Concepts covered

- Dynamic Programming
  - Introduction
  - Fibonacci numbers
  - Binomial Coefficients

Dynamic Programming

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

Dynamic Programming
Example: Fibonacci Numbers

- Recall definition of Fibonacci numbers:

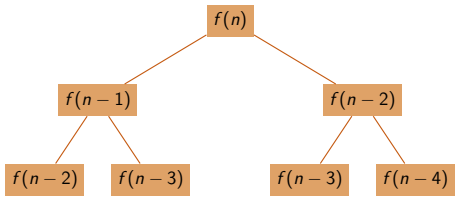
$$f(n) = f(n-1) + f(n-2)$$
$$f(0) = 0$$
$$f(1) = 1$$

Dynamic Programming
Example: Fibonacci Numbers

- Recall definition of Fibonacci numbers:

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

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



Dynamic Programming
Example: Fibonacci Numbers

Computing the nth Fibonacci number using bottom-up iteration and recording results:

$$
\begin{aligned}
f(0) &= 0 \\
f(1) &= 1 \\
f(2) &= 0 + 1 = 1 \\
f(3) &= 1 + 1 = 2 \\
f(4) &= 1 + 2 = 3 \\
&\vdots
\end{aligned}
$$

Dynamic Programming
Example: Fibonacci Numbers

Computing the nth Fibonacci number using bottom-up iteration and recording results:

$$
\begin{aligned}
f(0) &= 0 \\
f(1) &= 1 \\
f(2) &= 0 + 1 = 1 \\
f(3) &= 1 + 1 = 2 \\
f(4) &= 1 + 2 = 3 \\
&\vdots
\end{aligned}
$$

Efficiency:

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

Dynamic Programming
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

Dynamic Programming

Binomial Coefficient

- Binomial coefficients are coefficients of the binomial formula:

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

Dynamic Programming

Binomial Coefficient

- Binomial coefficients are coefficients of the binomial formula:

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

- Recurrence:

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

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

Dynamic Programming

Binomial Coefficient

- Binomial coefficients are coefficients of the binomial formula:

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

- Recurrence:

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

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

- 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)

Dynamic Programming
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** $C[n, k]$

Dynamic Programming
Binomial Coefficient Algorithm

PES UNIVERSITY ONLINE

Dynamic Programming Binomial Coefficient Algorithm

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

- Time: $\Theta(nk)$
- Space: $\Theta(nk)$

Dynamic Programming

Think About It

- What does dynamic programming have in common with divide-and-conquer? What is a principal difference between them?

Dynamic Programming

Think About It

- What does dynamic programming have in common with divide-and-conquer? What is a principal difference between them?
- 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?

Design and Analysis of Algorithms

The Knapsack Problem

Reetinder Sidhu

Department of Computer Science and Engineering

Design and Analysis of Algorithms

The Knapsack Problem

Reetinder Sidhu

Department of Computer Science and Engineering

The Knapsack Problem

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

- Dynamic Programming
  - Computing a Binomial Coefficient
  - **The Knapsack Problem**
  - Memory Functions
  - Warshall's and Floyd's Algorithms
- Limitations of Algorithmic Power
  - Lower-Bound Arguments
  - Decision Trees
  - P, NP, and NP-Complete, NP-Hard Problems
- Coping with the Limitations
  - Backtracking
  - Branch-and-Bound. Architecture (microprocessor instruction set)

Concepts covered

- The Knapsack Problem
  - Introduction
  - Recurrence
  - Example

The Knapsack Problem

Problem Definition

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

The Knapsack Problem
Kanpsack 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 id j ($j \leq W$)
- Then

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

The Knapsack Problem
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 6?

The Knapsack Problem
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

What is the maximum value that can be stored in a knapsack of capacity 6?

	capacity j				
i	1	2	3	4	5
1					
2					
3					
4					

The Knapsack Problem
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

What is the maximum value that can be stored in a knapsack of capacity 6?

	capacity j				
i	1	2	3	4	5
1	0				
2					
3					
4					

The Knapsack Problem
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

What is the maximum value that can be stored in a knapsack of capacity 6?

	capacity j				
i	1	2	3	4	5
1	0	12			
2					
3					
4					

The Knapsack Problem
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

What is the maximum value that can be stored in a knapsack of capacity 6?

	capacity j				
i	1	2	3	4	5
1	0	12	12		
2					
3					
4					

The Knapsack Problem
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

What is the maximum value that can be stored in a knapsack of capacity 6?

	capacity j				
i	1	2	3	4	5
1	0	12	12	12	
2					
3					
4					

The Knapsack Problem
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

What is the maximum value that can be stored in a knapsack of capacity 6?

	capacity j				
i	1	2	3	4	5
1	0	12	12	12	12
2					
3					
4					

The Knapsack Problem
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

	capacity j				
i	1	2	3	4	5
1	0	12	12	12	12
2					
3					
4					

What is the maximum value that can be stored in a knapsack of capacity 6?

The Knapsack Problem
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

	capacity j				
i	1	2	3	4	5
1	0	12	12	12	12
2	10				
3					
4					

What is the maximum value that can be stored in a knapsack of capacity 6?

The Knapsack Problem
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

	capacity j				
i	1	2	3	4	5
1	0	12	12	12	12
2	10	12			
3					
4					

What is the maximum value that can be stored in a knapsack of capacity 6?

The Knapsack Problem
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

	capacity j				
i	1	2	3	4	5
1	0	12	12	12	12
2	10	12	22		
3					
4					

What is the maximum value that can be stored in a knapsack of capacity 6?

The Knapsack Problem
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

What is the maximum value that can be stored in a knapsack of capacity 6?

	capacity j				
i	1	2	3	4	5
1	0	12	12	12	12
2	10	12	22	22	22
3					
4					

The Knapsack Problem
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

What is the maximum value that can be stored in a knapsack of capacity 6?

	capacity j				
i	1	2	3	4	5
1	0	12	12	12	12
2	10	12	22	22	22
3					
4					

The Knapsack Problem
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

What is the maximum value that can be stored in a knapsack of capacity 6?

	capacity j				
i	1	2	3	4	5
1	0	12	12	12	12
2	10	12	22	22	22
3	10				
4					

The Knapsack Problem
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

What is the maximum value that can be stored in a knapsack of capacity 6?

	capacity j				
i	1	2	3	4	5
1	0	12	12	12	12
2	10	12	22	22	22
3	10	12			
4					

The Knapsack Problem
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

What is the maximum value that can be stored in a knapsack of capacity 6?

	capacity j				
i	1	2	3	4	5
1	0	12	12	12	12
2	10	12	22	22	22
3	10	12	22		
4					

The Knapsack Problem
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

What is the maximum value that can be stored in a knapsack of capacity 6?

	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	
4					

The Knapsack Problem
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

What is the maximum value that can be stored in a knapsack of capacity 6?

	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					

The Knapsack Problem
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 6?

	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					

The Knapsack Problem
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 6?

	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				

The Knapsack Problem
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 6?

	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			

The Knapsack Problem
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 6?

	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		

The Knapsack Problem
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 6?

	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	

The Knapsack Problem
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 6?

	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

The Knapsack Problem
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 6?

	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**

The Knapsack Problem
Complexity

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

The Knapsack Problem

Think About It

The Knapsack Problem

Think About It

- Write pseudocode of the bottom-up dynamic programming algorithm for the knapsack problem

The Knapsack Problem

Think About It

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

The Knapsack Problem

Think About It

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

Design and Analysis of Algorithms

Memory Function Knapsack

Reetinder Sidhu

Department of Computer Science and Engineering

Design and Analysis of Algorithms

Memory Function Knapsack

Reetinder Sidhu

Department of Computer Science and Engineering

Memory Function Knapsack

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

Concepts covered

- Memory Function Knapsack
  - Motivation
  - Algorithm
  - Example

Memory Function Knapsack
Bottom Up Approach

Memory Function Knapsack
Bottom Up Approach

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

Memory Function Knapsack
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

Memory Function Knapsack
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

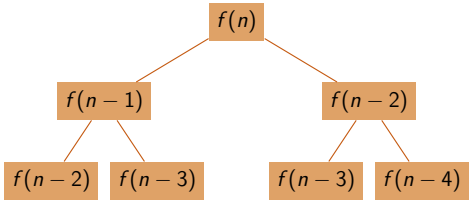
Memory Function Knapsack
Top Down Approach

Memory Function Knapsack
Top Down Approach

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

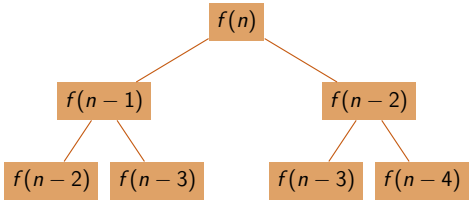
Memory Function Knapsack
Top Down Approach

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



Memory Function Knapsack
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

Memory Function Knapsack
Memory Function Dynamic Programming

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

Memory Function Knapsack
MF-DP Algorithm

Algorithm for Memory Function Dynamic Programming

procedure MFKNAPSACK(i, j)
    ▷ **Inputs:** i indicating the number items, and
    ▷ j, indicating the knapsack capacity
    ▷ **Output:** The value of an optimal feasible subset of the first i items
    ▷ **Note:** Uses global variables input arrays $Weights[1 \ldots n]$, $Values[1 \ldots n]$,
    ▷ and table $F[0 \ldots n, 0 \ldots W]$ whose entries are initialized with −1's except
    ▷ row 0 and column 0 is initialized with 0
    if $F[i, j] < 0$ **then**
        if $j < Weights[i]$ **then**
            $value \leftarrow MFKnapsack(i - 1, j)$
        else $value \leftarrow max(MFKnapsack(i - 1, j),\ \ Values[i] + MFKnapsack(i - 1, j - Weights[i]))$
            $F[i, j] \leftarrow value$
    return $F[i, j]$

Memory Function Knapsack
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?

Memory Function Knapsack
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	0	1	2	3	4	5
0	0					
1						
2						
3						
4						□

Memory Function Knapsack
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	0	1	2	3	4	5
0	0					
1						
2						
3						□
4						□

Memory Function Knapsack
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	0	1	2	3	4	5
0	0					
1						
2						
3				□		□
4						□

Memory Function Knapsack
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	0	1	2	3	4	5
0	0	□	□	□	□	□
1	□	□	□	□	□	□
2	□		□	□		□
3	□			□		□
4	□					□

Memory Function Knapsack
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	0	1	2	3	4	5
0	0	□	□	□	□	□
1	□	□	□	□	□	□
2	□	–	□	□	–	□
3	□	–	–	□	–	□
4	□	–	–	–	–	□

Memory Function Knapsack
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	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

Memory Function Knapsack
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	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 $(1, 2)$

Memory Function Knapsack
Complexity

- Constant factor imrpovement 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

Memory Function Knapsack

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 entires are filled?
  - How many are reused?

Design and Analysis of Algorithms

Transitive Closure (Warshall's Algorith

Reetinder Sidhu

Department of Computer Science and Engineering

Design and Analysis of Algorithms

Transitive Closure (Warshall's Algorithm)

Reetinder Sidhu

Department of Computer Science and Engineering

Transitive Closure (Warshall's Algorithm)

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

Concepts covered

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

Transitive Closure (Warshall's Algorithm)

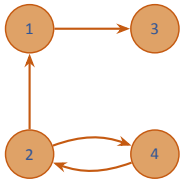
Transitive Closure

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

Transitive Closure (Warshall's Algorithm)

Transitive Closure

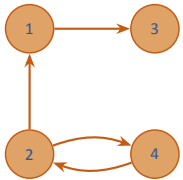
- Computes the transitive closure of a relation
- Alternatively: existence of all nontrivial paths in a digraph (directed graph)
- Example of transitive closure:



Transitive Closure (Warshall's Algorithm)

Transitive Closure

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

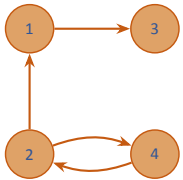


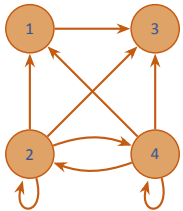
	1	2	3	4
1	0	0	1	0
2	1	0	0	1
3	0	0	0	0
4	0	1	0	0

Transitive Closure (Warshall's Algorithm)

Transitive Closure

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



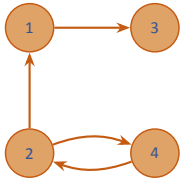


	1	2	3	4
1	0	0	1	0
2	1	0	0	1
3	0	0	0	0
4	0	1	0	0

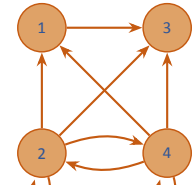
Transitive Closure (Warshall's Algorithm)

Transitive Closure

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



	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	**1**	**1**	1
3	0	0	0	0
4	**1**	1	**1**	**1**

Transitive Closure (Warshall's Algorithm)

Warshall's Algorithm

- Constructs transitive closure T as the last matrix in the sequence of $n \times n$ matrices $R^{(0)}, \ldots, R^{(k)}, \ldots, R^{(n)}$ 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 $R^{(k)}$

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

Transitive Closure (Warshall's Algorithm)

Warshall's Algorithm

- Constructs transitive closure T as the last matrix in the sequence of $n \times n$ matrices $R^{(0)}, \ldots, R^{(k)}, \ldots, R^{(n)}$ 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 $R^{(k)}$

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

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

Transitive Closure (Warshall's Algorithm)

Algorithm

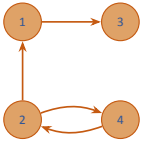
PES UNIVERSITY ONLINE

Transitive Closure (Warshall's Algorithm)

```
1: procedure WARSHALL(()A[1 . . . n, 1 . . . n])
2:     ▷ Input: The adjacency matrix A of a digraph with n vertices
3:     ▷ Output: The transitive closure of the digraph
4:     R^(0) ← A
5:     for k ← 1 to n do
6:         for i ← 1 to n do
7:             for j ← 1 to n do
8:                 R^(k)[i, j] ← R^(k−1)[i, j] or (R^(k−1)[i, k] and R^(k−1)[k, j]);
9:     return R
```

Transitive Closure (Warshall's Algorithm)

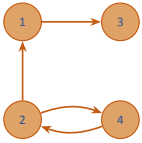
Warshall's Algorithm



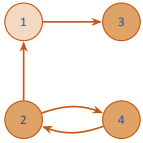
	1	2	3	4
1	0	0	1	0
2	1	0	0	1
3	0	0	0	0
4	0	1	0	0

Transitive Closure (Warshall's Algorithm)

Warshall's Algorithm



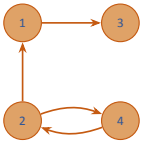
	1	2	3	4
1	0	0	1	0
2	1	0	0	1
3	0	0	0	0
4	0	1	0	0



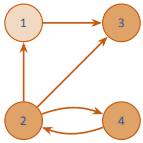
	1	2	3	4
1	0	0	1	0
2	1	0	0	1
3	0	0	0	0
4	0	1	0	0

Transitive Closure (Warshall's Algorithm)

Warshall's Algorithm



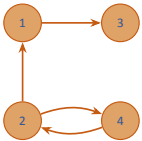
	1	2	3	4
1	0	0	1	0
2	1	0	0	1
3	0	0	0	0
4	0	1	0	0



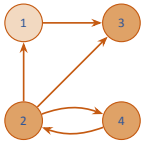
	1	2	3	4
1	0	0	1	0
2	1	0	**1**	1
3	0	0	0	0
4	0	1	0	0

Transitive Closure (Warshall's Algorithm)

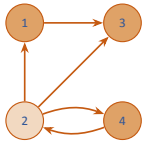
Warshall's Algorithm



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



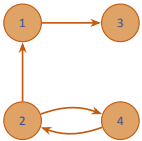
	1	2	3	4
1	0	0	1	0
2	1	0	**1**	1
3	0	0	0	0
4	0	1	0	0



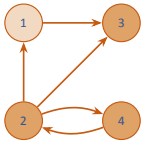
	1	2	3	4
1	0	0	1	0
2	1	0	1	1
3	0	0	0	0
4	0	1	0	0

Transitive Closure (Warshall's Algorithm)

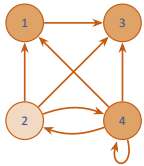
Warshall's Algorithm



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



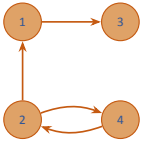
	1	2	3	4
1	0	0	1	0
2	1	0	**1**	1
3	0	0	0	0
4	0	1	0	0



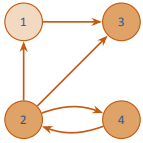
	1	2	3	4
1	0	0	1	0
2	1	0	1	1
3	0	0	0	0
4	**1**	1	**1**	**1**

Transitive Closure (Warshall's Algorithm)

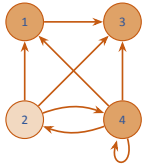
Warshall's Algorithm



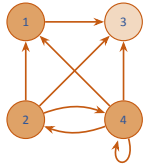
	1	2	3	4
1	0	0	1	0
2	1	0	0	1
3	0	0	0	0
4	0	1	0	0



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



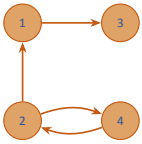
	1	2	3	4
1	0	0	1	0
2	1	0	1	1
3	0	0	0	0
4	**1**	1	**1**	**1**



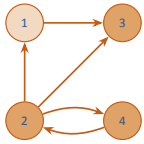
	1	2	3	4
1	0	0	1	0
2	1	0	1	1
3	0	0	0	0
4	1	1	1	1

Transitive Closure (Warshall's Algorithm)

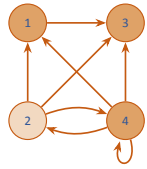
Warshall's Algorithm



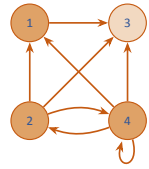
	1	2	3	4
1	0	0	1	0
2	1	0	0	1
3	0	0	0	0
4	0	1	0	0



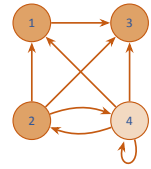
	1	2	3	4
1	0	0	1	0
2	1	0	**1**	1
3	0	0	0	0
4	0	1	0	0



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



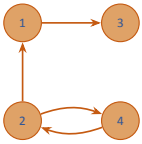
	1	2	3	4
1	0	0	1	0
2	1	0	1	1
3	0	0	0	0
4	1	1	1	1



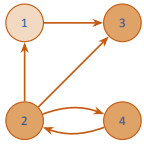
	1	2	3	4
1	0	0	1	1
2	1	0	1	1
3	0	0	0	0
4	1	1	1	1

Transitive Closure (Warshall's Algorithm)

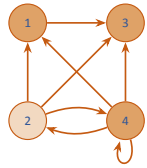
Warshall's Algorithm



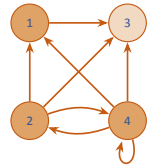
	1	2	3	4
1	0	0	1	0
2	1	0	0	1
3	0	0	0	0
4	0	1	0	0



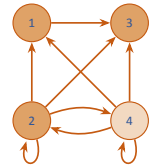
	1	2	3	4
1	0	0	1	0
2	1	0	**1**	1
3	0	0	0	0
4	0	1	0	0



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

Transitive Closure (Warshall's Algorithm)

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?

Design and Analysis of Algorithms

All Pairs Shortest Path (Floyd's Algorithm)

Reetinder Sidhu

Department of Computer Science and Engineering

Design and Analysis of Algorithms

All Pairs Shortest Path (Floyd's Algorithm)

Reetinder Sidhu

Department of Computer Science and Engineering

All Pairs Shortest Path (Floyd's Algorithm)

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

- Dynamic Programming
  - Computing a Binomial Coefficient
  - The Knapsack Problem
  - Memory Functions
  - **Warshall's and Floyd's Algorithms**
- Limitations of Algorithmic Power
  - Lower-Bound Arguments
  - Decision Trees
  - P, NP, and NP-Complete, NP-Hard Problems
- Coping with the Limitations
  - Backtracking
  - Branch-and-Bound. Architecture (microprocessor instruction set)

Concepts covered

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

All Pairs Shortest Path (Floyd's Algorithm)

Problem Definition

- Given a 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 $R^{(0)}, \ldots, R^{(k)}, \ldots, R^{(n)}$ 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 $D^{(0)}, \ldots, D^{(k)}, \ldots, D^{(n)}$ where $D^{(k)}[i,j]$ is the shortest path from i to j with only first k vertices allowed as intermediate vertices

All Pairs Shortest Path (Floyd's Algorithm)

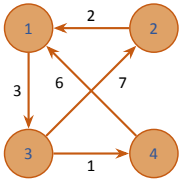
Example

- Example of all pairs shortest paths:

All Pairs Shortest Path (Floyd's Algorithm)

Example

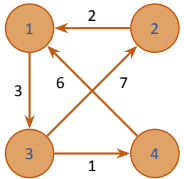
- Example of all pairs shortest paths:



All Pairs Shortest Path (Floyd's Algorithm)

Example

- Example of all pairs shortest paths:

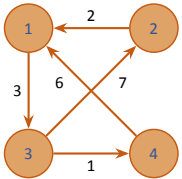


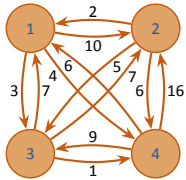
	1	2	3	4
1	0	∞	3	∞
2	2	0	∞	∞
3	∞	7	0	1
4	6	∞	∞	0

All Pairs Shortest Path (Floyd's Algorithm)

Example

- Example of all pairs shortest paths:



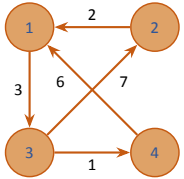


$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

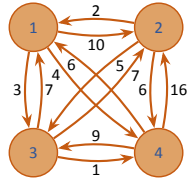
All Pairs Shortest Path (Floyd's Algorithm)

Example

- Example of all pairs shortest paths:



	1	2	3	4
1	0	∞	3	∞
2	2	0	∞	∞
3	∞	7	0	1
4	6	∞	∞	0



	1	2	3	4
1	0	10	3	4
2	2	0	5	6
3	7	7	0	1
4	6	16	9	0

All Pairs Shortest Path (Floyd's Algorithm)

Algorithm

Transitive Closure (Floyd's Algorithm)

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

All Pairs Shortest Path (Floyd's Algorithm)

Algorithm

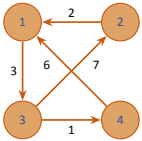
Transitive Closure (Floyd's Algorithm)

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

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

All Pairs Shortest Path (Floyd's Algorithm)

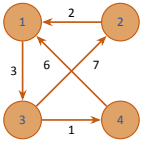
Example

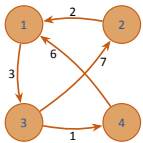


	1	2	3	4
1	0	∞	3	∞
2	2	0	∞	∞
3	∞	7	0	1
4	6	∞	∞	0

All Pairs Shortest Path (Floyd's Algorithm)

Example



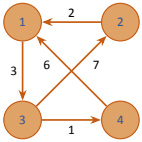


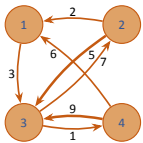
	1	2	3	4
1	0	∞	3	∞
2	2	0	∞	∞
3	∞	7	0	1
4	6	∞	∞	0

	1	2	3	4
1	0	∞	3	∞
2	2	0	∞	∞
3	∞	7	0	1
4	6	∞	∞	0

All Pairs Shortest Path (Floyd's Algorithm)

Example



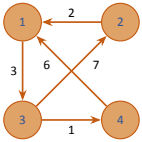


	1	2	3	4
1	0	∞	3	∞
2	2	0	∞	∞
3	∞	7	0	1
4	6	∞	∞	0

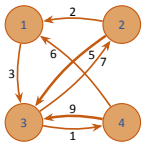
	1	2	3	4
1	0	∞	3	∞
2	2	0	**5**	∞
3	∞	7	0	1
4	6	∞	**9**	0

All Pairs Shortest Path (Floyd's Algorithm)

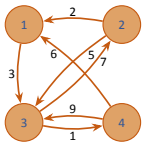
Example



	1	2	3	4
1	0	∞	3	∞
2	2	0	∞	∞
3	∞	7	0	1
4	6	∞	∞	0



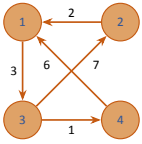
	1	2	3	4
1	0	∞	3	∞
2	2	0	**5**	∞
3	∞	7	0	1
4	6	∞	**9**	0



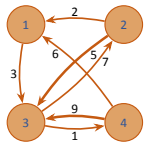
	1	2	3	4
1	0	∞	3	∞
2	2	0	5	∞
3	∞	7	0	1
4	6	∞	9	0

All Pairs Shortest Path (Floyd's Algorithm)

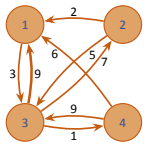
Example



	1	2	3	4
1	0	∞	3	∞
2	2	0	∞	∞
3	∞	7	0	1
4	6	∞	∞	0



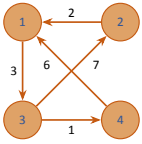
	1	2	3	4
1	0	∞	3	∞
2	2	0	**5**	∞
3	∞	7	0	1
4	6	∞	**9**	0



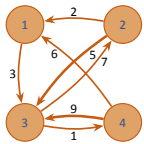
	1	2	3	4
1	0	∞	3	∞
2	2	0	5	∞
3	**9**	7	0	1
4	6	∞	9	0

All Pairs Shortest Path (Floyd's Algorithm)

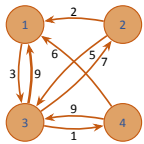
Example



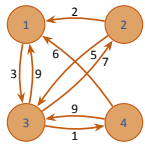
	1	2	3	4
1	0	∞	3	∞
2	2	0	∞	∞
3	∞	7	0	1
4	6	∞	∞	0



	1	2	3	4
1	0	∞	3	∞
2	2	0	**5**	∞
3	∞	7	0	1
4	6	∞	**9**	0



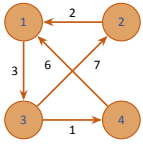
	1	2	3	4
1	0	∞	3	∞
2	2	0	5	∞
3	**9**	7	0	1
4	6	∞	9	0



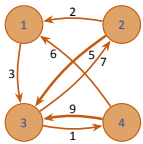
	1	2	3	4
1	0	∞	3	∞
2	2	0	5	∞
3	9	7	0	1
4	6	∞	9	0

All Pairs Shortest Path (Floyd's Algorithm)

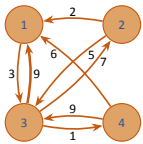
Example



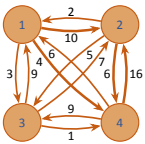
	1	2	3	4
1	0	∞	3	∞
2	2	0	∞	∞
3	∞	7	0	1
4	6	∞	∞	0



	1	2	3	4
1	0	∞	3	∞
2	2	0	**5**	∞
3	∞	7	0	1
4	6	∞	**9**	0



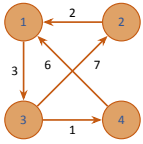
	1	2	3	4
1	0	∞	3	∞
2	2	0	5	∞
3	**9**	7	0	1
4	6	∞	9	0



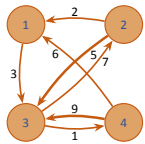
	1	2	3	4
1	0	**10**	3	**4**
2	2	0	5	**6**
3	9	7	0	1
4	6	**16**	9	0

All Pairs Shortest Path (Floyd's Algorithm)

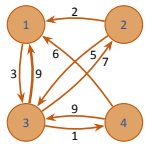
Example



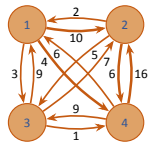
	1	2	3	4
1	0	∞	3	∞
2	2	0	∞	∞
3	∞	7	0	1
4	6	∞	∞	0



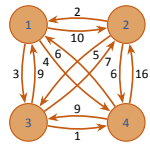
	1	2	3	4
1	0	∞	3	∞
2	2	0	**5**	∞
3	∞	7	0	1
4	6	∞	**9**	0



	1	2	3	4
1	0	∞	3	∞
2	2	0	5	∞
3	**9**	7	0	1
4	6	∞	9	0



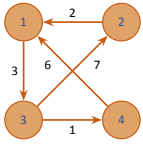
	1	2	3	4
1	0	**10**	3	**4**
2	2	0	5	**6**
3	9	7	0	1
4	6	**16**	9	0



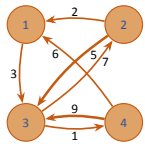
	1	2	3	4
1	0	10	3	4
2	2	0	5	6
3	9	7	0	1
4	6	16	9	0

All Pairs Shortest Path (Floyd's Algorithm)

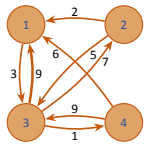
Example



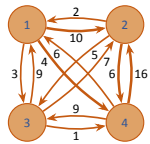
	1	2	3	4
1	0	∞	3	∞
2	2	0	∞	∞
3	∞	7	0	1
4	6	∞	∞	0



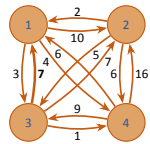
	1	2	3	4
1	0	∞	3	∞
2	2	0	**5**	∞
3	∞	7	0	1
4	6	∞	**9**	0



	1	2	3	4
1	0	∞	3	∞
2	2	0	5	∞
3	**9**	7	0	1
4	6	∞	9	0



	1	2	3	4
1	0	**10**	3	**4**
2	2	0	5	**6**
3	9	7	0	1
4	6	**16**	9	0



	1	2	3	4
1	0	10	3	4
2	2	0	5	6
3	**7**	7	0	1
4	6	16	9	0

All Pairs Shortest Path (Floyd's Algorithm)

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

Design and Analysis of Algorithms

Lower-Bound Arguments

Reetinder Sidhu

Department of Computer Science and Engineering

Design and Analysis of Algorithms

Lower-Bound Arguments

Reetinder Sidhu

Department of Computer Science and Engineering

Lower-Bound Arguments

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

- Dynamic Programming
  - Computing a Binomial Coefficient
  - The Knapsack Problem
  - Memory Functions
  - Warshall's and Floyd's Algorithms
  - Optimal Binary Search Trees
- Limitations of Algorithmic Power
  - **Lower-Bound Arguments**
  - Decision Trees
  - P, NP, and NP-Complete, NP-Hard Problems
- Coping with the Limitations
  - Backtracking
  - Branch-and-Bound

Concepts covered

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

Limitations of Algorithmic Power

PES UNIVERSITY ONLINE

- 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

Lower-Bound Arguments

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

Lower-Bound Arguments
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

Methods for Establishing Lower Bounds

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

Trivial Lower Bounds

PES UNIVERSITY

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

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 $a_1 < a_2 < \ldots < a_n$ 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

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 $\Omega(n \log n)$)

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

Design and Analysis of Algorithms

Decision Trees

Reetinder Sidhu

Department of Computer Science and Engineering

Design and Analysis of Algorithms

Decision Trees

Reetinder Sidhu

Department of Computer Science and Engineering

Decision Trees

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

- Dynamic Programming
  - Computing a Binomial Coefficient
  - The Knapsack Problem
  - Memory Functions
  - Warshall's and Floyd's Algorithms
  - Optimal Binary Search Trees
- Limitations of Algorithmic Power
  - Lower-Bound Arguments
  - **Decision Trees**
  - P, NP, and NP-Complete, NP-Hard Problems
- Coping with the Limitations
  - Backtracking
  - Branch-and-Bound

Concepts covered

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

Decision Trees
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

Decision Trees

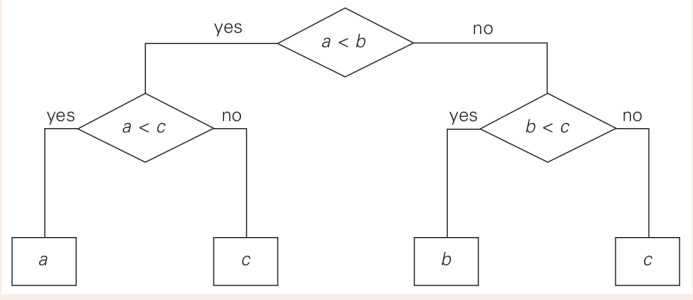
Introduction

- 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

Decision Trees

Example: Decision tree for minimum of three numbers

Decision tree for a determining the minimum of three numbers



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

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 $n!$

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

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

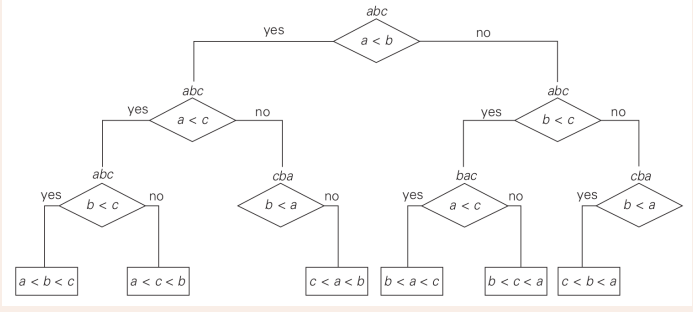
- Using Stirling's formula:

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

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

Decision Trees for Sorting Algorithms

Decision Tree for Three Element Selection Sort



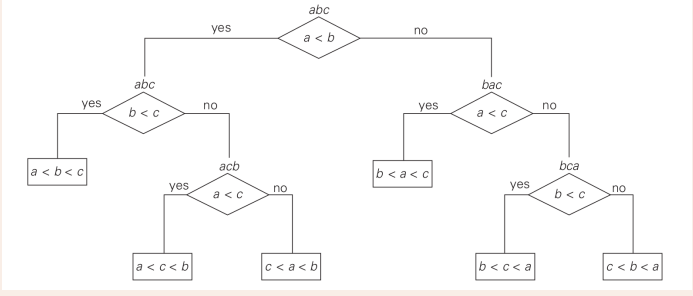
Decision Trees for Sorting Algorithms

- 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:

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

Decision Trees for Sorting Algorithms

Decision Tree for Three Element Insertion Sort



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 Cavg made by any comparison-based algorithm in sorting an n-element list has been proved

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

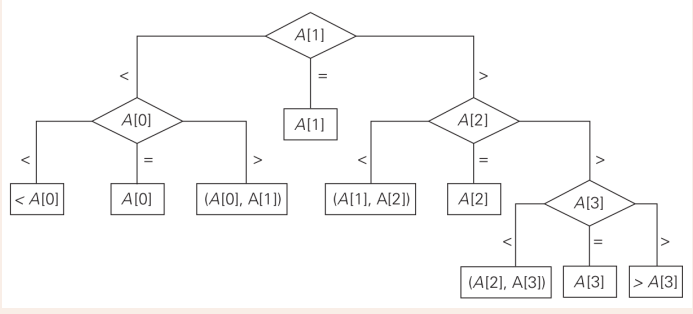
Decision Trees for Searching Algorithms

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

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

Decision Trees for Searching Algorithms

Ternary Decision Tree for Four Element Array Binary Search



Decision Trees for Searching Algorithms

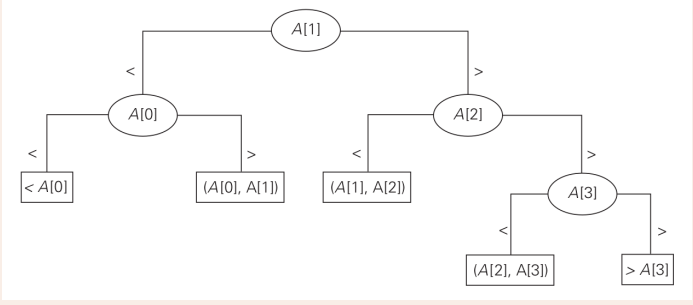
- For an array of n elements, all such decision trees will have 2n +1 leaves (n for successful searches and n + 1 for unsuccessful ones)
- Since the minimum height h of a ternary tree with l leaves is floor(log3l), we get the following lower bound on the number of worst-case comparisons:

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

- 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?

Decision Trees for Searching Algorithms

Binary Decision Tree for Four Element Array Binary Search



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

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

Design and Analysis of Algorithms

P, NP, and NP-Complete Problems

Reetinder Sidhu

Department of Computer Science and Engineering

Design and Analysis of Algorithms

P, NP, and NP-Complete Problems

Reetinder Sidhu

Department of Computer Science and Engineering

P, NP, and NP-Complete Problems

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

- Dynamic Programming
  - Computing a Binomial Coefficient
  - The Knapsack Problem
  - Memory Functions
  - Warshall's and Floyd's Algorithms
  - Optimal Binary Search Trees
- Limitations of Algorithmic Power
  - Lower-Bound Arguments
  - Decision Trees
  - **P, NP, and NP-Complete, NP-Hard Problems**
- Coping with the Limitations
  - Backtracking
  - Branch-and-Bound

Concepts covered

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

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 be proved that any algorithm takes exponential time
- unknown

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

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

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"

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:

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

- 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: $\Theta(n)$

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:

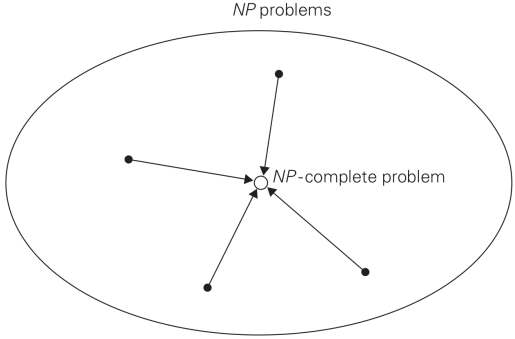
$$P \subseteq NP$$

- Big question:

$$P = NP\ ?$$

NP-Complete Problems

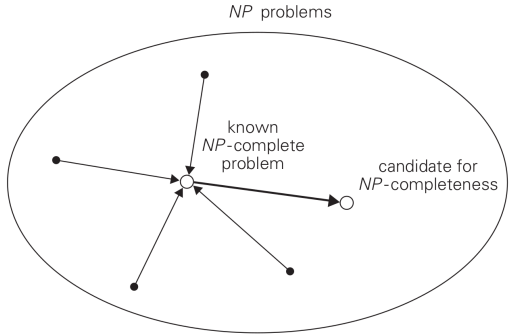
- A decision problem D is NP-complete if it's as hard as any problem in NP, i.e.,
  - D is in NP
  - every problem in NP is polynomial-time reducible to D



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

NP-Complete Problems

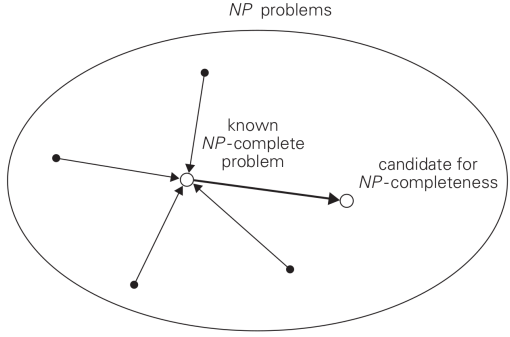
- Other NP-complete problems obtained through polynomial- time reductions from a known NP-complete problem



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

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 $P \neq NP$

Design and Analysis of Algorithms

Backtracking

Reetinder Sidhu

Department of Computer Science and Engineering

Design and Analysis of Algorithms

Backtracking

Reetinder Sidhu

Department of Computer Science and Engineering

BACKTRACKING

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

- Dynamic Programming
  - Computing a Binomial Coefficient
  - The Knapsack Problem
  - Memory Functions
  - Warshall's and Floyd's Algorithms
  - Optimal Binary Search Trees
- Limitations of Algorithmic Power
  - Lower-Bound Arguments
  - Decision Trees
  - P, NP, and NP-Complete, NP-Hard Problems
- Coping with the Limitations
  - **Backtracking**
  - Branch-and-Bound

Concepts covered

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

BACKTRACKING
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

BACKTRACKING
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

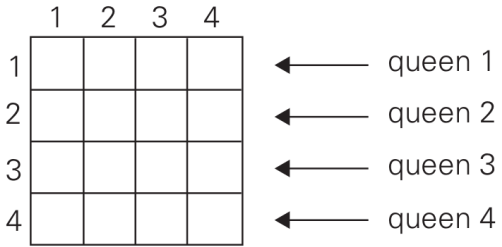
BACKTRACKING

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

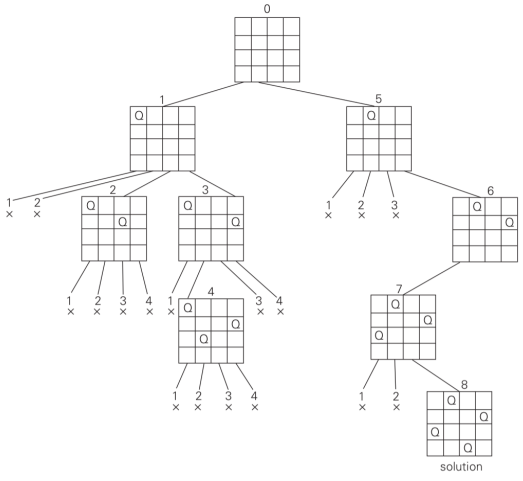
BACKTRACKING
Example: *N*-Queens Problem

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



BACKTRACKING

State-Space Tree of the 4-Queens Problem

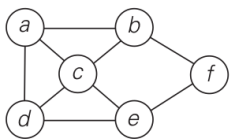


Example: Hamiltonian Circuit Problem

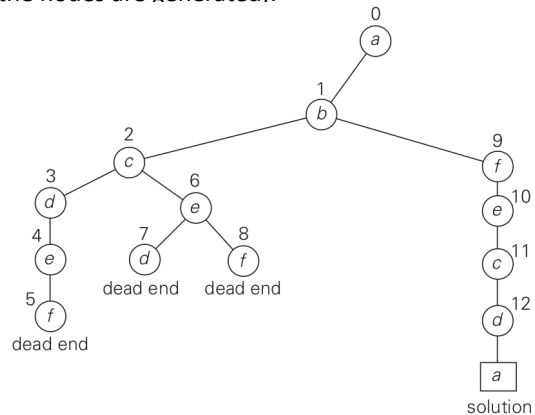
Hamiltonian Circuit

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):

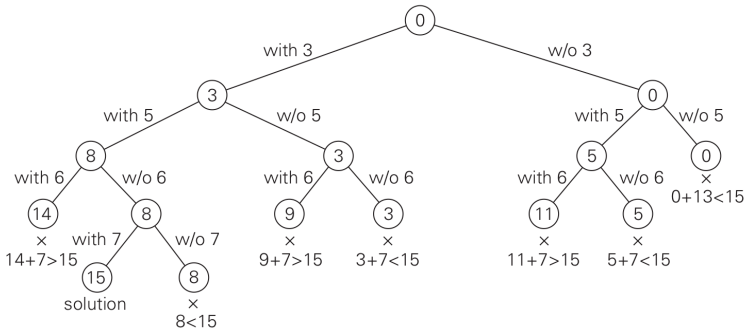


Example: Subset Sum Problem

Subset Sum Problem

Given set $A = \{a_1, \ldots, a_n\}$ 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):



BACKTRACKING
Algorithm

PES UNIVERSITY

Backtrack Algorithm

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

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

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

Design and Analysis of Algorithms

Branch and Bound

Reetinder Sidhu

Department of Computer Science and Engineering

Design and Analysis of Algorithms

Branch and Bound

Reetinder Sidhu

Department of Computer Science and Engineering

Branch and Bound

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

- Dynamic Programming
  - Computing a Binomial Coefficient
  - The Knapsack Problem
  - Memory Functions
  - Warshall's and Floyd's Algorithms
  - Optimal Binary Search Trees
- Limitations of Algorithmic Power
  - Lower-Bound Arguments
  - Decision Trees
  - P, NP, and NP-Complete, NP-Hard Problems
- Coping with the Limitations
  - Backtracking
  - **Branch and Bound**

Concepts covered

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

Branch and Bound

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

Branch and Bound
Example: Assignment Problem

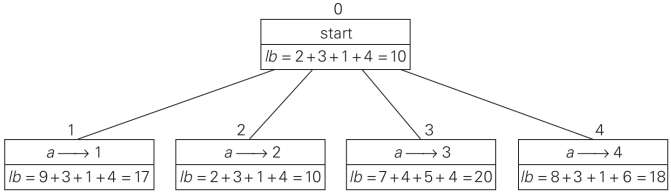
- Select one element in each row of the cost matrix C so that:
  - no two selected elements are in the same column
  - the sum is minimized
- Example

	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

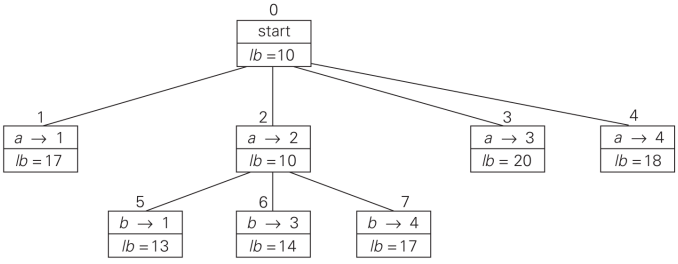
Branch and Bound

Example: First two levels of the state-space tree

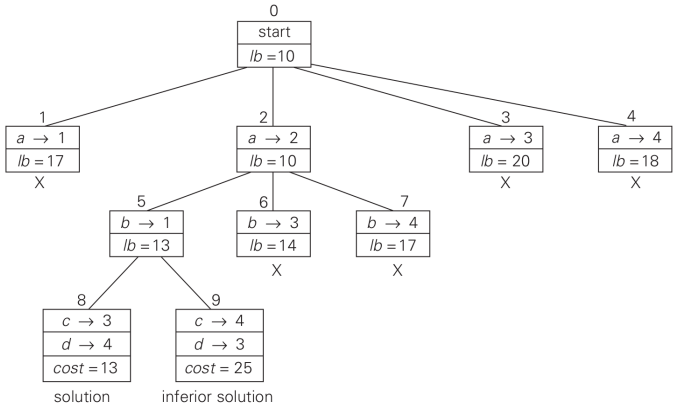


Branch and Bound

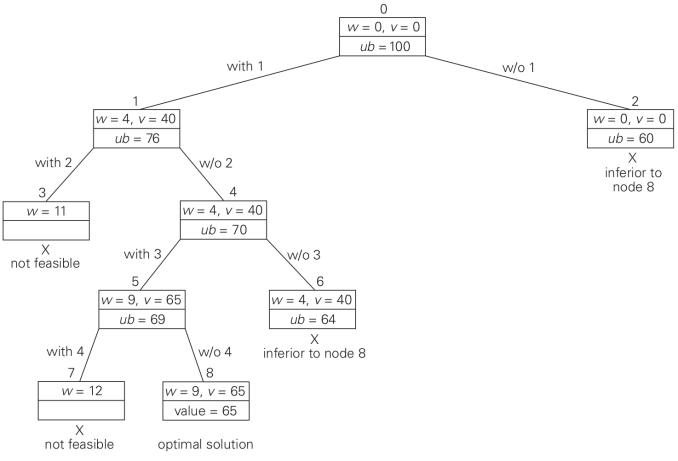
Example: First three levels of the state-space tree



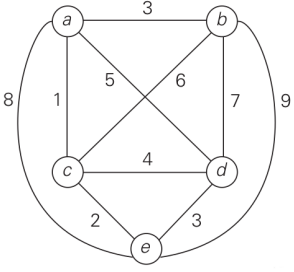
Branch and Bound
Example: Complete state-space tree

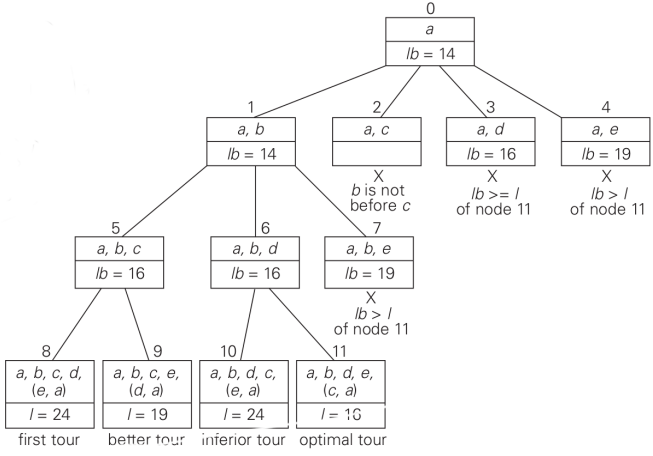


Branch and Bound
Example: Knapsack Problem



Branch and Bound
Example: Traveling Salesman Problem





Branch and Bound

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