Lecture 3

Brute Force

Lecture Contents

1. Selection Sort and Bubble Sort
2. Sequential Search and Brute-Force String Matching
3. Closest-Pair and Convex-Hull Problems by Brute Force
4. Exhaustive Search
5. DFS and BFS

Brute Force

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

Brute Force

- **Examples**:
  1. Given $p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$
     find the value of p at x_0, i.e., compute $p(x_0)$
  2. Computing a^n ($a > 0$, n a nonnegative integer)
     $a^n = a \times a \times \ldots \times a$
  3. Computing $n! = 1 \times 2 \times \ldots \times n$
  4. Multiplying two matrices
  5. Searching for a key of a given value in a list

Brute-Force Sorting Algorithm

- **Selection Sort**: Scan the array to find its smallest element and swap it with the first element. Then, starting with the second element, scan the elements to the right of it to find the smallest among them and swap it with the second elements. Generally, on pass i ($0 \le i \le n - 2$), find the smallest element in $A[i..n-1]$ and swap it with $A[i]$:

Example of Selection Sort

89	45	68	90	29	34	**17**
17	45	68	90	**29**	34	89
17	29	68	90	45	**34**	89
17	29	34	90	**45**	68	89
17	29	34	45	90	**68**	89
17	29	34	45	68	90	**89**
17	29	34	45	68	89	90

Selection Sort Algorithm

ALGORITHM *SelectionSort*($A[0..n-1]$)

//Sorts a given array by selection sort

//Input: An array $A[0..n-1]$ of orderable elements

//Output: Array $A[0..n-1]$ sorted in ascending order

for $i \leftarrow 0$ **to** $n-2$ **do**

$min \leftarrow i$

for $j \leftarrow i+1$ **to** $n-1$ **do**

if $A[j] < A[min]$ $min \leftarrow j$

swap $A[i]$ and $A[min]$

Analysis of Selection Sort

• The algorithm's basic operation is the key comparison $A[j] < A[min]$. The number of times it is executed depends only on the array's size and is given by the following sum:

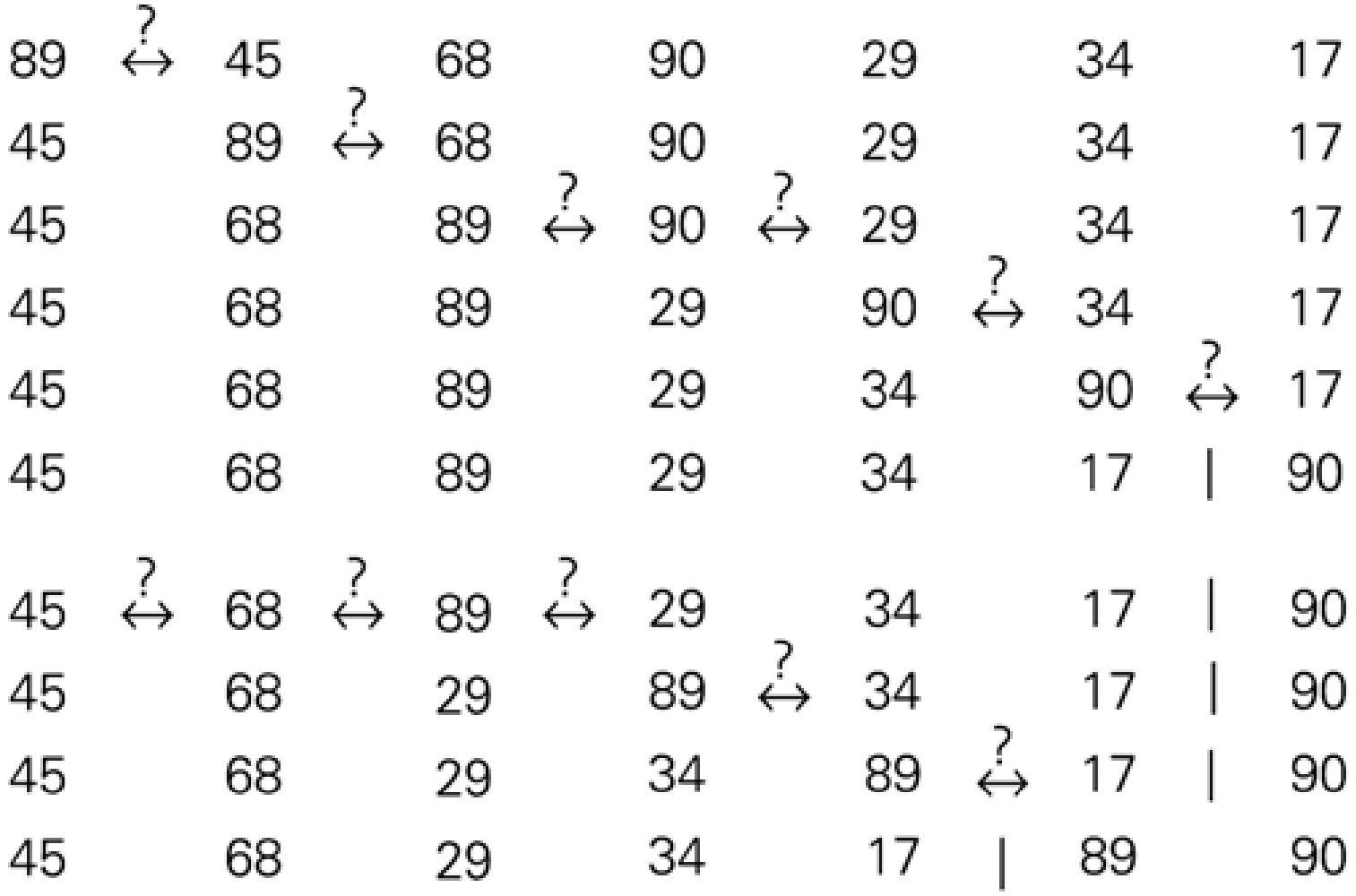
$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1]$$

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

Brute-Force Sorting Algorithm

- **Bubble Sort**: Compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, the largest element is moved to the last position in the list. The next pass, the second largest element is moved to the second last position in the list. Generally, the *i*th pass, the *i*th largest element is moved to the *i*th last position in the list, where $0 \leq i \leq n - 2$. After $n - 1$ passes, the list is sorted.

Example of Bubble Sort



etc.

Bubble Sort Algorithm

ALGORITHM *BubbleSort*($A[0..n-1]$)

//Sorts a given array by bubble sort

//Input: An array $A[0..n-1]$ of orderable elements

//Output: Array $A[0..n-1]$ sorted in ascending order

for $i \leftarrow 0$ **to** $n-2$ **do**

  for $j \leftarrow 0$ **to** $n-2-i$ **do**

    if $A[j] > A[j+1]$ swap $A[j]$ and $A[j+1]$

Analysis of Bubble Sort

- The algorithm's basic operation is the key comparison $A[j] > A[j+1]$. The number of times it is executed is given by the following sum:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1]$$

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Lecture Contents

1. Selection Sort and Bubble Sort
2. Sequential Search and Brute-Force String Matching
3. Closest-Pair and Convex-Hull Problems by Brute Force
4. Exhaustive Search
5. DFS and BFS

Sequential Search

ALGORITHM *SequentialSearch2*($A[0..n]$, K)

```
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0..n - 1] whose value is
//            equal to K or -1 if no such element is found
A[n] ← K
i ← 0
while A[i] ≠ K do
    i ← i + 1
if i < n return i
else return -1
```

Sequential Search

- The running time of this algorithm depends on inputs for the same list size n.
- In the worst case, the algorithm makes the largest number of key comparisons among all possible inputs of size n:

  - there is no matching element (i.e., $A[n] = K$):

$$C_{worst}(n) = n + 1.$$

  - the first matching element happens to be the last one on the list, $C_{worst}(n) = n$.

Sequential Search

- In the best case, inputs are lists of size n with their first elements equal to a search key. Accordingly, we have $C_{best}(n) = 1$.

(See Chapter 2, Section 2.1 for other details)

Brute-Force String Matching

- **pattern**: a string of m characters to search for
- **text**: a string of n characters to search in ($m \leq n$)
- **The String-Matching Problem**: asks to find a substring in the given text that matches the required pattern.

Brute-Force String Matching

- Brute-force Algorithm

Step 1: Align pattern at beginning of text

Step 2: Moving from left to right, compare each character of pattern to the corresponding character in text until

- all characters are found to match (successful search); or
- a mismatch is detected

Brute-Force String Matching

- Brute-force Algorithm

Step 3: While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

Examples of Brute-Force String Matching

```
N O B O D Y _ N O T I C E D _ H I M
N O T
  N O T
    N O T
      N O T
        N O T
          N O T
            N O T
              N O T
```

Brute-Force String Matching Pseudocode

ALGORITHM *BruteForceStringMatch*($T[0..n-1]$, $P[0..m-1]$)

```
//Implements brute-force string matching
//Input: An array T[0..n - 1] of n characters representing a text and
//       an array P[0..m - 1]) of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//matching substring or -1 if the search is unsuccessful
for i ← 0 to n - m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return -1
```

Analysis of Brute Force String Matching

- In the worst case, the algorithm may have to make all m comparisons before shifting the pattern, and this can happen for each of the $n - m + 1$ tries. Thus, in the worst case, the algorithm is in $\Theta(nm)$.
- In the average case, for a typical word search in a natural language text, we should expect that most shifts would happen after very few comparisons. Indeed, for searching in random texts, it has been shown to be linear, i.e., $\Theta(n + m) = \Theta(n)$.

Lecture Contents

1. Selection Sort and Bubble Sort
2. Sequential Search and Brute-Force String Matching
3. Closest-Pair and Convex-Hull Problems by Brute Force
4. Exhaustive Search
5. DFS and BFS

Closest-Pair Problem

- **The Closest-pair Problem** is to find the two closest points in a set of *n* points (in the two-dimensional Cartesian plane).

- Brute-force Algorithm

  Compute the distance between every pair of distinct points ($n(n - 1)/2$ pairs) and return the indexes of the points for which the distance is the smallest.

Closest-Pair Brute-Force Algorithm

ALGORITHM *BruteForceClosestPoints*(P)

//Finds two closest points in the plane by brute force

//Input: A list P of n ($n \geq 2$) points $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$

//Output: Indices *index*1 and *index*2 of the closest pair of points

$dmin \leftarrow \infty$

for $i \leftarrow 1$ **to** $n - 1$ **do**

  for $j \leftarrow i + 1$ **to** n **do**

    $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ //*sqrt* is the square root function

    if $d < dmin$

      $dmin \leftarrow d$; $index1 \leftarrow i$; $index2 \leftarrow j$

return *index*1, *index*2

Analysis of Closest-Pair Brute-Force Algorithm

• The basic operation of the algorithm is computing the Euclidean distance between two points. The number of times the basic operation will be executed can be computed as follows:

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1} (n-i)$$

$$= [(n-1) + (n-2) + \ldots + 1] = (n-1)n/2 \in \Theta(n^2).$$

Brute-Force Strengths

- Strengths

  - wide applicability

  - simplicity

  - yields reasonable algorithms for some important problems (e.g., matrix multiplication, sorting, searching, string matching)

Brute-Force Weaknesses

- Weaknesses
  - rarely yields efficient algorithms
  - some brute-force algorithms are unacceptably slow
  - not as constructive as some other design techniques

Convex-Hull Problem

• The ***convex-hull problem*** is the problem of constructing the convex hull for a given set S of n points.

• **Definition**: A set of points (finite or infinite) in the plane is called ***convex*** if for any two points P and Q in the set, the entire line segment with the endpoints at P and Q belongs to the set.

Convex Set Examples

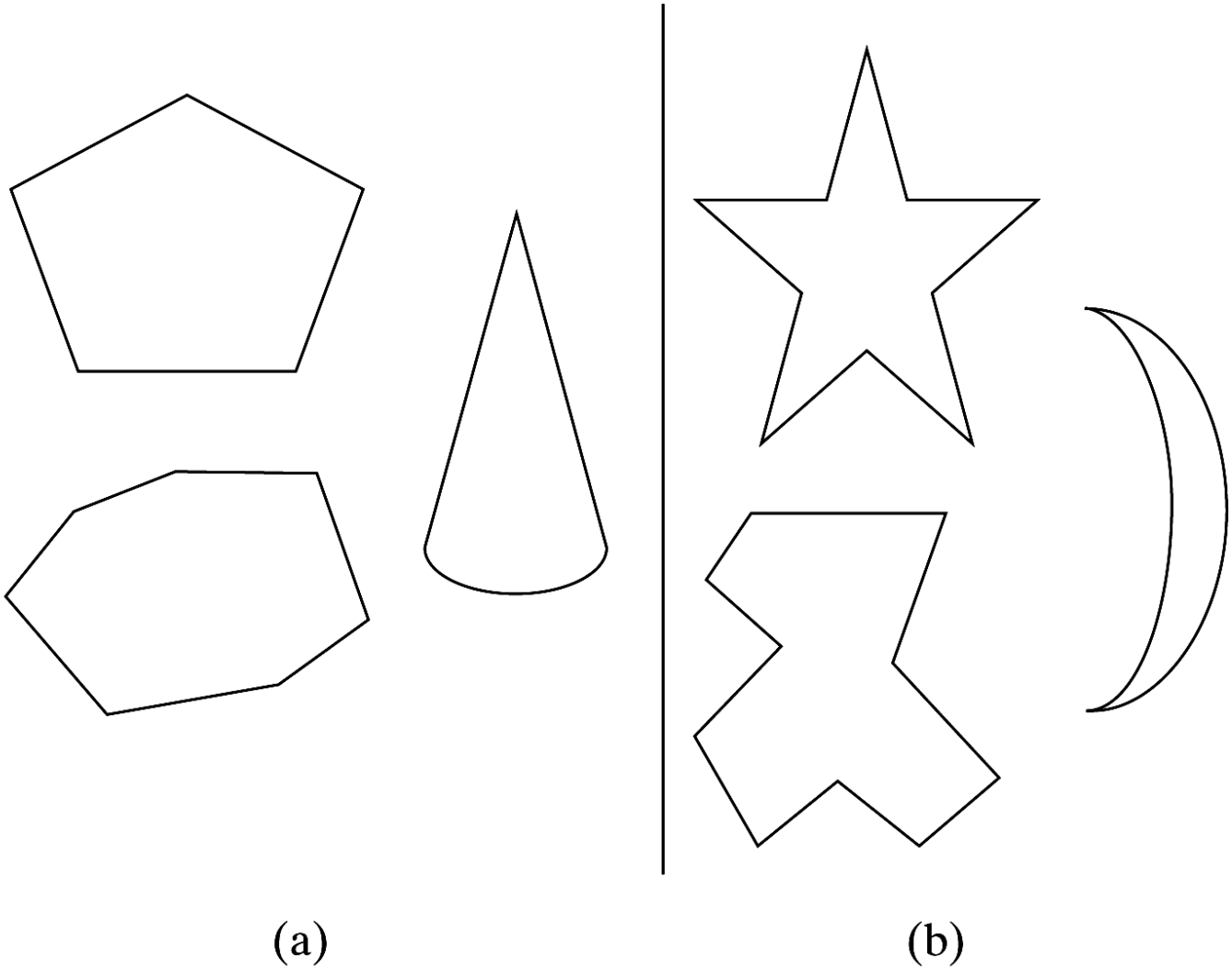


FIGURE 3.4 (a) Convex sets. (b) Sets that are not convex.

Convex-Hull Problem

• **Definition**: The ***convex hull*** of a set *S* of points is the smallest convex set/polygon containing *S*.

• **Theorem**: The convex hull of any set *S* of $n > 2$ points (not all on the same line) is a convex polygon with the vertices at some of the points of *S*.

Convex-Hull Example

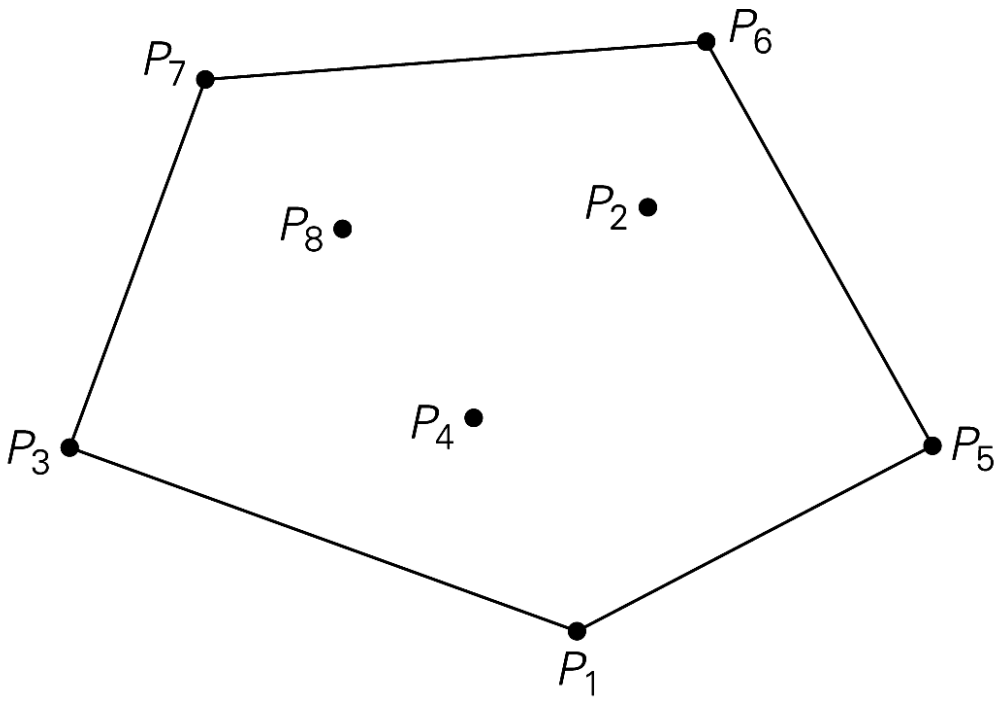


FIGURE 3.6 The convex hull for this set of eight points is the convex polygon with vertices at P_1, P_5, P_6, P_7, and P_3.

Convex-Hull Brute-Force Algorithm

- A line segment connecting two points P_i and P_j of a set of n points is a part of its convex hull's boundary if and only if all the other points of the set lie on the same side of the straight line through these two points.
- Repeating this test for every pair of points yields a list of line segments that make up the convex hull's boundary.

Convex-Hull Brute-Force Algorithm

- The straight line through $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$ can be defined by the equation

$$ax + by = c,$$

where $a = y_2 - y_1$, $b = x_1 - x_2$, $c = x_1y_2 - y_1x_2$.

Convex-Hull Brute-Force Algorithm

- The line P_1P_2 divides the plane into two half-planes:
  - for all the points in one of them, $ax + by > c$,
  - for all the points in the other, $ax + by < c$,
  - for the points on the line, $ax + by = c$.
- Thus, to check whether certain points lie on the same side of the line, we can check whether the expression $ax + by = c$ has the same sign at each of these points.

Analysis of Convex-Hull Brute-Force Algorithm

- The time efficiency of this algorithm is in $O(n^3)$.
  - for each of $n(n - 1)/2$ pairs of distinct points, we may need to find the sign of $ax + by - c$ for each of the other $n - 2$ points.

Lecture Contents

1. Selection Sort and Bubble Sort
2. Sequential Search and Brute-Force String Matching
3. Closest-Pair and Convex-Hull Problems by Brute Force
4. Exhaustive Search
5. DFS and BFS

Exhaustive Search

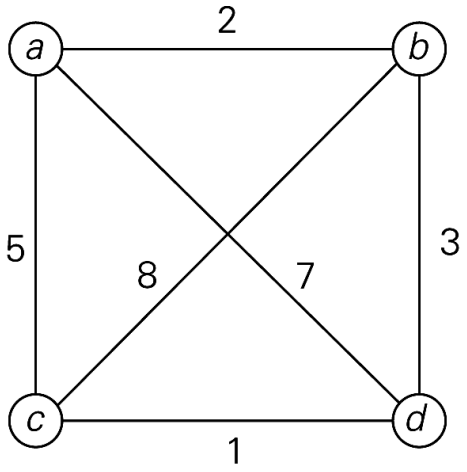
- Exhaustive Method:
  - generate a list of all potential solutions to the problem in a systematic manner
  - select an optimal solution (i.e., minimum or maximum one)

The Traveling Salesman Problem (TSP)

• Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city.

• Exhaustive search solution: we can get all the tours by generating all the permutations of *n* - 1 intermediate cities, compute the tour lengths, and find the shortest among them.

Example of TSP by Exhaustive Search



Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

FIGURE 3.7 Solution to a small instance of the traveling salesman problem by exhaustive search

The Knapsack Problem

- **The knapsack problem**: Given a knapsack of capacity W and n items of weights w_1, ..., w_n and values v_1, ..., v_n, find the most valuable subset of the items that fits into the knapsack.
- The **0/1** (or ***discrete***) knapsack problem: we are allowed either to take an item in its entirety or not to take it at all.

Knapsack Problem by Exhaustive Search

- The exhaustive search approach:
  - generate all the subsets of the set of n items given (i.e., 2^n subsets),
  - computing the total weight of each subset to identify feasible subsets (i.e., the ones with the total weight $\leq W$), and
  - finding a subset of the largest value among them.

Example of the Knapsack Problem

- **Example**: Solve the following instance of the 0/1 knapsack problem.

item	weight	value
1	7	\$42
2	3	\$12
3	4	\$40
4	5	\$25

$W = 10$

Example of the Knapsack Problem

Ø	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$36
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

Analysis of the Knapsack Problem

• The number of subsets of an n-element set is 2^n, the exhaustive search leads to a $\Omega(2^n)$ algorithm

The Assignment Problem

• There are *n* people who need to be assigned to *n* jobs, one person per job. The cost of assigning person *i* to job *j* is *C*[*i*, *j*]. Find an assignment that minimizes the total cost.

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

The Assignment Problem

- Brute-force strategy:
  - Generate all legitimate assignments (i.e., $n!$ permutations of integers 1, 2, ..., n),
  - Compute their costs, and select the cheapest one.

- In a permutation $(j_1, \ldots, j_n)$, the ith component, $i = 1, ..., n$, indicates the ith person (i.e., the ith row) is assigned to the j_ith job (i.e., the j_ith column)

Assignment Problem by Exhaustive Search

Assignment	Total Cost
1, 2, 3, 4	9 + 4 + 1 + 4 = 18
1, 2, 4, 3	9 + 4 + 8 + 9 = 30
1, 3, 2, 4	9 + 3 + 8 + 4 = 24
1, 3, 4, 2	9 + 3 + 8 + 6 = 26
1, 4, 2, 3	9 + 7 + 8 + 9 = 33
1, 4, 3, 2	9 + 7 + 1 + 6 = 23

Cost matrix *C*

	1	2	3	4
1	**9**	**2**	**7**	**8**
2	**6**	**4**	**3**	**7**
3	**5**	**8**	**1**	**8**
4	**7**	**6**	**9**	**4**

Assignment Problem by Exhaustive Search

Assignment	Total Cost
2, 1, 3, 4	**2 + 6 + 1 + 4 = 13**
2, 1, 4, 3	2 + 6 + 8 + 9 = 25
2, 3, 1, 4	2 + 3 + 5 + 4 = 14
2, 3, 4, 1	2 + 3 + 8 + 7 = 20
2, 4, 1, 3	2 + 7 + 5 + 9 = 23
2, 4, 3, 1	2 + 7 + 1 + 7 = 17

Cost matrix C

	1	2	3	4
1	**9**	**2**	**7**	**8**
2	**6**	**4**	**3**	**7**
3	**5**	**8**	**1**	**8**
4	**7**	**6**	**9**	**4**

Assignment Problem by Exhaustive Search

Assignment	Total Cost
3, 1, 2, 4	7 + 6 + 8 + 4 = 25
3, 1, 4, 2	7 + 6 + 8 + 6 = 27
3, 2, 1, 4	7 + 4 + 5 + 4 = 20
3, 2, 4, 1	7 + 4 + 8 + 7 = 26
3, 4, 1, 2	7 + 7 + 5 + 6 = 25
3, 4, 2, 1	7 + 7 + 8 + 7 = 29

Cost matrix *C*

	1	2	3	4
1	**9**	**2**	**7**	**8**
2	**6**	**4**	**3**	**7**
3	**5**	**8**	**1**	**8**
4	**7**	**6**	**9**	**4**

Assignment Problem by Exhaustive Search

Assignment	Total Cost
4, 1, 2, 3	8 + 6 + 8 + 9 = 31
4, 1, 3, 2	8 + 6 + 1 + 6 = 21
4, 2, 1, 3	8 + 4 + 5 + 9 = 26
4, 2, 3, 1	8 + 4 + 1 + 7 = 20
4, 3, 1, 2	8 + 3 + 5 + 6 = 22
4, 3, 2, 1	8 + 3 + 8 + 7 = 26

Cost matrix *C*

	1	2	3	4
1	**9**	**2**	**7**	**8**
2	**6**	**4**	**3**	**7**
3	**5**	**8**	**1**	**8**
4	**7**	**6**	**9**	**4**

Assignment Problem by Exhaustive Search

Assignment	Total Cost
2, 1, 3, 4	**2 + 6 + 1 + 4 = 13**

Cost matrix *C*

	1	2	3	4
1	**9**	**2**	**7**	**8**
2	**6**	**4**	**3**	**7**
3	**5**	**8**	**1**	**8**
4	**7**	**6**	**9**	**4**

• The optimal solution is: assign Person 1 to Job 2, Person 2 to Job 1, Person 3 to Job 3, and Person 4 to Job 4, with the total (minimal) cost of the assignment being 13.

Final Comments on Exhaustive Search

- Exhaustive-search algorithms run in a realistic amount of time only on very small instances
- In many cases, exhaustive search or its variation is the only known way to get exact solution

Lecture Contents

1. Selection Sort and Bubble Sort
2. Sequential Search and Brute-Force String Matching
3. Closest-Pair and Convex-Hull Problems by Brute Force
4. Exhaustive Search
5. DFS and BFS

Depth-First Search

Algorithm DFS // M.H. Alsuwaiyel
Input: A directed or undirected graph $G = (V, E)$.
Output: Numbering of the vertices in
depth-first search order.

1. *predfn* ← 1; *postdfn* ← 1
2. **for** each vertex $v \in V$
3. mark *v unvisited*
4. **end for**

Depth-First Search

5. **for** each vertex $v \in V$
6. **if** v is marked *unvisited* **then** *dfs*(v)
   // v is starting vertex
7. **end for**

Depth-First Search

Procedure *dfs*(*v*) // *v* is starting vertex, using stack

1. $S \leftarrow \{v\}$ // insert *v* into stack
2. mark *v visited*
3. **while** $S \neq \{\}$
4. $v \leftarrow$ Peek(*S*) // *v* is current vertex
5. find an unvisited neighbor *w* of *v*

Depth-First Search

```
6.    if w exists then
7.        Push(w, S)
8.        mark w visited
9.        predfn ← predfn + 1
10.   else
11.       Pop(S); postdfn ← postdfn + 1
12.   end if
13. end while
```

Breadth-First Search

Algorithm BFS // M.H. Alsuwaiyel

Input: A directed or undirected graph $G = (V, E)$.

Output: Numbering of the vertices in BFS order.

1. $bfn \leftarrow 1$
2. **for** each vertex $v \in V$
3. mark v *unvisited*
4. **end for**

Breadth-First Search

5. **for** each vertex $v \in V$
6. **if** v is marked *unvisited* **then** *bfs*(v)
   // v is starting vertex
7. **end for**

Breadth-First Search

Procedure *bfs*(*v*) // *v* is starting vertex, using queue

1. $Q \leftarrow \{v\}$ // insert *v* into queue
2. mark *v visited*
3. **while** $Q \neq \{\}$
4. $\quad v \leftarrow$ dequeue(Q) // *v* is current vertex
5. $\quad$ **for** each edge $(v, w) \in E$
6. $\quad\quad$ **if** *w* is marked *unvisited* **then**

Breadth-First Search

```
7.              enqueue(w, Q)
8.              mark w visited
9.              bfn ← bfn + 1
10.         end if
11.     end for
12. end while
```

Summary

1. Solve **traveling salesman problem (TSP)** using exhaustive search.

2. Solve **the 0/1 knapsack problem** using exhaustive search.

3. Solve **the assignment problem** using exhaustive search.

Summary

4. **The 8-queens problem**: Consider the classic puzzle of placing eight queens on an 8 × 8 chessboard so that no two queens are in the same row or in the same column or on the same diagonal.

Exercises

1. **Magic squares** A magic square of order n is an arrangement of the integers from 1 to n^2 in an $n \times n$ matrix, with each number occurring exactly once, so that each row, each column, and each main diagonal has the same sum.

a. Prove that if a magic square of order n exists, the sum in question must be equal to $n(n^2 + 1)/2$.

b. Design an exhaustive-search algorithm for generating all magic squares of order n.