CSC 8301- Design and Analysis of Algorithms

Lecture 4
Brute Force, Exhaustive Search,
Graph Traversal Algorithms

Brute-Force Approach

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

Example 1: Computing $a^n$ ($a > 0$, $n$ is a positive integer)

Example 2: Searching for a given value in a list
Brute-Force Algorithm for Sorting

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 element. Generally, on pass \(i (0 \leq i \leq n-2)\), find the smallest element in \(A[i..n-1]\) and swap it with \(A[i]\):

\[
A[0] \leq \ldots \leq A[i-1] \mid A[i], \ldots, A[min], \ldots, A[n-1]
\]

in their final positions

Example: 7 8 2 3 5

Analysis of Selection Sort

**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 \(A[min]\)

•  *Basic operation*
•  *Summation for \(C(n)\)*
• Summation for $C(n)$

$$C(n) = \sum_{0 \leq i \leq n-1} \sum_{i+1 \leq j \leq n-1} 1$$

**Closest-Pair Problem**

The closest-pair problem is to find the two closest points in a set of $n$ points in the Cartesian plane (with the distance between two points measured by the standard Euclidean distance formula).

**Brute-force algorithm for the closest-pair problem**

Compute the distance between every pair of distinct points and return the indexes of the points for which the distance is the smallest.
Closest-Pair Brute Force Algorithm (cont.)

**ALGORITHM** BruteForceClosestPoints(P)
//Input: A list P of n (n ≥ 2) points P_1 = (x_1, y_1), . . . , P_n = (x_n, y_n)
//Output: Indices index1 and index2 of the closest pair of points
\[ d_{\text{min}} \leftarrow \infty \]
for i ← 1 to n - 1 do
  for j ← 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 \( d < d_{\text{min}} \)
      \[ d_{\text{min}} \leftarrow d; \quad \text{index1} \leftarrow i; \quad \text{index2} \leftarrow j \]
return index1, index2

• Time efficiency:

• Noteworthy improvement:

General Notes on Brute-Force Approach

**Strengths:**
- wide applicability
- simplicity
- yields reasonable algorithms for some important problems (e.g., sorting, searching, matrix multiplication)

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

**Note:** Brute force can be a legitimate alternative in view of the human time vs. computer time costs
Exhaustive search

*Exhaustive search* is a brute force approach to solving a problem that involves searching for an element with a special property, usually among *combinatorial objects* such permutations, combinations, or subsets of a set.

Method:
- systematically construct all potential solutions to the problem (often using standard algorithms for generating combinatorial objects such as those in Sec. 4.3)
- evaluate solutions one by one, disqualifying infeasible ones and, for optimization problems, keeping track of the best solution found so far
- when search ends, return the (best) solution found

Example 1: 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.

Alternatively: Find shortest *Hamiltonian circuit* in a weighted connected graph.

Example:
TSP by exhaustive search

<table>
<thead>
<tr>
<th>Tour</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>a→b→c→d→a</td>
<td>2+3+7+5 = 17</td>
</tr>
<tr>
<td>a→b→d→c→a</td>
<td>2+4+7+8 = 21</td>
</tr>
<tr>
<td>a→c→b→d→a</td>
<td>8+3+4+5 = 20</td>
</tr>
<tr>
<td>a→c→d→b→a</td>
<td>8+7+4+2 = 21</td>
</tr>
<tr>
<td>a→d→b→c→a</td>
<td>5+4+3+8 = 20</td>
</tr>
<tr>
<td>a→d→c→b→a</td>
<td>5+7+3+2 = 17</td>
</tr>
</tbody>
</table>

Fewer tours?

Efficiency:

Example 2: Knapsack Problem

Given $n$ items with
weights: $w_1$ $w_2$ ... $w_n$
values: $v_1$ $v_2$ ... $v_n$
and a knapsack of capacity $W$,
find most valuable subset of the items that fit into the knapsack

Example: Knapsack capacity $W=16$

<table>
<thead>
<tr>
<th>item</th>
<th>weight</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>$20</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>$30</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>$50</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>$10</td>
</tr>
</tbody>
</table>
Knapsack by exhaustive search

<table>
<thead>
<tr>
<th>Subset</th>
<th>Total weight</th>
<th>Total value</th>
<th>Efficiency:</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1}</td>
<td>2</td>
<td>$20</td>
<td></td>
</tr>
<tr>
<td>{2}</td>
<td>5</td>
<td>$30</td>
<td></td>
</tr>
<tr>
<td>{3}</td>
<td>10</td>
<td>$50</td>
<td></td>
</tr>
<tr>
<td>{4}</td>
<td>5</td>
<td>$10</td>
<td></td>
</tr>
<tr>
<td>{1,2}</td>
<td>7</td>
<td>$50</td>
<td></td>
</tr>
<tr>
<td>{1,3}</td>
<td>12</td>
<td>$70</td>
<td></td>
</tr>
<tr>
<td>{1,4}</td>
<td>7</td>
<td>$30</td>
<td></td>
</tr>
<tr>
<td>{2,3}</td>
<td>15</td>
<td>$80</td>
<td></td>
</tr>
<tr>
<td>{2,4}</td>
<td>10</td>
<td>$40</td>
<td></td>
</tr>
<tr>
<td>{3,4}</td>
<td>15</td>
<td>$60</td>
<td></td>
</tr>
<tr>
<td>{1,2,3}</td>
<td>17</td>
<td>not feasible</td>
<td></td>
</tr>
<tr>
<td>{1,2,4}</td>
<td>12</td>
<td>$60</td>
<td></td>
</tr>
<tr>
<td>{1,3,4}</td>
<td>17</td>
<td>not feasible</td>
<td></td>
</tr>
<tr>
<td>{2,3,4}</td>
<td>20</td>
<td>not feasible</td>
<td></td>
</tr>
<tr>
<td>{1,2,3,4}</td>
<td>22</td>
<td>not feasible</td>
<td></td>
</tr>
</tbody>
</table>

Comments on Exhaustive Search

- Typically, exhaustive search algorithms run in a realistic amount of time only on very small instances.
- For some problems, there are much better alternatives
  - shortest paths
  - minimum spanning tree
  - assignment problem
- In many cases, exhaustive search (or variation) is the only known way to solve problem exactly for all its possible instances
  - TSP
  - knapsack problem
Graph Traversal

Many problems require processing all graph vertices (and edges) in systematic fashion, which can be considered “exhaustive search” algorithms.

Graph traversal algorithms:

- Depth-first search (DFS)

- Breadth-first search (BFS)

Depth-First Search (DFS)

- Visits graph’s vertices by always moving away from last visited vertex to unvisited one, backtracks if no adjacent unvisited vertex is available.

- “Redraws” graph in tree-like fashion (with tree edges and back edges for undirected graph)

**Depth-First Search (DFS)**

- Uses a stack
  - a vertex is pushed onto the stack first time it is reached
  - a vertex is popped off the stack when it becomes a dead end, i.e., when there is no adjacent unvisited vertex

**Example: DFS traversal of undirected graph**

- Assumption: ties broken alphabetically

**DFS traversal stack:**

**DFS forest:**
Pseudocode of DFS

ALGORITHM \( DFS(G) \)
//Implements a depth-first search traversal of a given graph
//Input: Graph \( G = (V, E) \)
//Output: Graph \( G \) with its vertices marked with consecutive integers
//in the order they’ve been first encountered by the DFS traversal
mark each vertex in \( V \) with 0 as a mark of being “unvisited”
\( \text{count} \leftarrow 0 \)
for each vertex \( v \) in \( V \) do
  if \( v \) is marked with 0
    \text{dfs}(v)
\text{dfs}(v)
//visits recursively all the unvisited vertices connected to vertex \( v \) by a path
//and numbers them in the order they are encountered
//via global variable \( \text{count} \)
\( \text{count} \leftarrow \text{count} + 1 \); mark \( v \) with \( \text{count} \)
for each vertex \( w \) in \( V \) adjacent to \( v \) do
  if \( w \) is marked with 0
    \text{dfs}(w)

Notes on DFS

- DFS can be implemented with graphs represented as:
  - adjacency matrices: \( \Theta(|V|^2) \)
  - adjacency lists: \( \Theta(|V|+|E|) \)

- Yields two distinct ordering of vertices:
  - order in which vertices are first encountered (pushed onto stack)
  - order in which vertices become dead-ends (popped off stack)

- Applications:
  - checking connectivity, finding connected components
  - checking acyclicity
  - searching state-space of problems for solution (AI)
Breadth-first search (BFS)

- Visits graph vertices by moving across to all the neighbors of last visited vertex

- “Redraws” graph in tree-like fashion (with tree edges and cross edges for undirected graph)

Pseudocode of BFS

```
ALGORITHM BFS(G)
// Implements a breadth-first search traversal of a given graph
// Input: Graph G = (V, E)
// Output: Graph G with its vertices marked with consecutive integers
// in the order they have been visited by the BFS traversal
mark each vertex in V with 0 as a mark of being “unvisited”
count ← 0
for each vertex v in V do
    if v is marked with 0
        bfs(v)
    // visits all the unvisited vertices connected to vertex v by a path
    // and assigns them the numbers in the order they are visited
    // via global variable count
    count ← count + 1; mark v with count and initialize a queue with v
    while the queue is not empty do
        for each vertex w in V adjacent to the front vertex do
            if w is marked with 0
                count ← count + 1; mark w with count
                add w to the queue
            remove the front vertex from the queue
```

Example of BFS traversal of undirected graph

- Instead of a stack, BFS uses a queue
- Assumption: neighbors visited in alphabetical order

BFS traversal queue: a b e f c d g h

BFS forest: i j q

Notes on BFS

- BFS has same efficiency as DFS and can be implemented with graphs represented as:
  - adjacency matrices: $\Theta(|V|^2)$
  - adjacency lists: $\Theta(|V|+|E|)$

- Yields single ordering of vertices (order added/deleted from queue is the same)

- Applications: same as DFS, but can also find paths from a vertex to all other vertices with the smallest number of edges
Homework

Exercises 3.1: 4, 5, 8, 11
Exercises 3.4: 1, 5, 6
Exercises 3.5: 1, 2, 4, 6

Reading:
- Sec. 3.1, 3.4 and 3.5

Next: Decrease-and-conquer (Chapter 4)