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

$$A[0] \le \ . \ . \ . \ \le A[i-1] \ | \ A[i], \ . \ . \ . \ . \ , A[min], \ . \ . \ ., A[n-1]$$

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 ← 0 to n − 2 do
        min ← i
        for j ← i + 1 to n − 1 do
            if A[j] < A[min]   min ← j
        swap A[i] and A[min]
```

- *Basic operation*
- *Summation for C(n)*

- *Summation for $C(n)$*

$$C(n) = \Sigma_{0 \le i \le n-1} \Sigma_{i+1 \le j \le n-1} 1$$

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 $(n \geq 2)$ points $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$
//Output: Indices $index1$ and $index2$ 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 $index1, index2$

- 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:

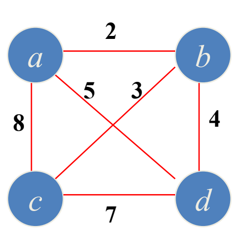
- systematically construct all potential solutions to the problem (often using standard algorithms for generating combinatorial objects such as those in Sec. 4.3)
- evaluate solutions one by one, disqualifying infeasible ones and, for optimization problems, keeping track of the best solution found so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

Tour	Cost
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17

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

item	weight	value
1	2	$20
2	5	$30
3	10	$50
4	5	$10

Knapsack by exhaustive search

Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	**17**	not feasible
{1,2,4}	12	$60
{1,3,4}	**17**	not feasible
{2,3,4}	**20**	not feasible
{1,2,3,4}	**22**	not feasible

Efficiency:

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

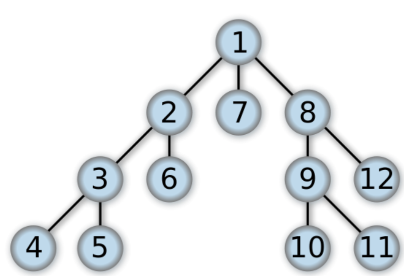
Many problems require processing all graph vertices (and edges) in systematic fashion, which can be considered "exhaustive search" algorithms

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.

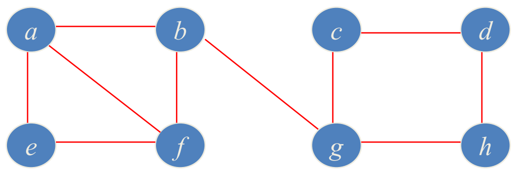


Source: Wikipedia, https://en.wikipedia.org/wiki/Depth-first_search

- "Redraws" graph in tree-like fashion (with tree edges and *back edges* for undirected graph)

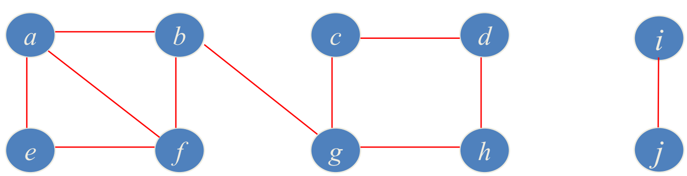
Depth-First Search (DFS)

- Uses a stack
  - a vertex is pushed onto the stack first time 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)*

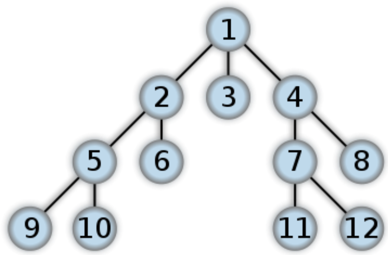
```
//Implements a depth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//in the order they've been first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        dfs(v)

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 count
count ← count + 1; mark v with count
for each vertex w in V adjacent to v do
    if w is marked with 0
        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



Source: Wikipedia, https://en.wikipedia.org/wiki/Breadth-first_search

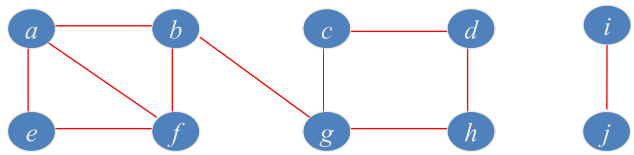
- "Redraws" graph in tree-like fashion (with tree edges and *cross edges* for undirected graph)

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)

    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:

BFS forest:

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)