CSCE 350: Data Structures and Algorithms

Chin-Tser Huang
huangct@cse.sc.edu

University of South Carolina
Announcement

- Homework 2 will be returned on Thursday; solution will be available on class website today
- Midterm Exam study guide has been passed out last Thursday
- HW score has been updated at CSE dropbox [https://dropbox.cse.sc.edu](https://dropbox.cse.sc.edu)
Closest-Pair Problem

Find the two closest points in a set of \( n \) points (in the two-dimensional Cartesian plane).

\[ \text{Brute-force algorithm} \]
Compute the distance between every pair of distinct points and return the indexes of the points for which the distance is the smallest.
**Brute-Force Closest-Pair Algorithm**

**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
\[ d_{\text{min}} \leftarrow \infty \]
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 $d < d_{\text{min}}$
      \[ d_{\text{min}} \leftarrow d; \quad index1 \leftarrow i; \quad index2 \leftarrow j \]
return $index1$, $index2$

**Efficiency:**

Can we make it faster?
Convex Hull

- We won’t cover it here, but the book studies a brute-force algorithm for computing convex hulls
Exhaustive Search

- A brute-force approach to combinatorial problem
  - Generate each and every element of the problem’s domain
  - Then compare and select the desirable element that satisfies the set constraints
  - Involve combinatorial objects such as *permutations*, *combinations*, and *subsets* of a given set
  - The time efficiency is usually bad – usually the complexity grows *exponentially* with the input size

- Three examples
  - Traveling salesman problem
  - Knapsack problem
  - Assignment problem
Example 1: Traveling Salesman Problem

- Find the shortest tour through a given $n$ cities that visits each city exactly once before returning to the starting city

- Alternatively, finding shortest Hamiltonian Circuit – a cycle that passes through all the vertices of the graph exactly once

- Exhaustive search:
  - List all the possible Hamiltonian circuits (starting from any vertex)
  - Ignore the direction
  - How many candidate circuits do we have? $\Rightarrow (n-1)!/2$
  - Very high complexity
TSP Example

```
Tour         Length
a ---> b ---> c --->d ---> a   / = 2 + 8 + 1 + 7 = 18
a ---> b ---> d ---> c ---> a   / = 2 + 3 + 1 + 5 = 11      optimal
a ---> c ---> b ---> d ---> a   / = 5 + 8 + 3 + 7 = 23
a ---> c ---> d ---> b ---> a   / = 5 + 1 + 3 + 2 = 11      optimal
a ---> d ---> b ---> c ---> a   / = 7 + 3 + 8 + 5 = 23
a ---> d ---> c ---> b ---> a   / = 7 + 1 + 8 + 2 = 18
```
Example 2: Knapsack Problem

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

Exhaustive search:
- Find all $2^n$ subset of the $n$ items
  - each item can be selected or not selected
- Select the one with the largest value while satisfies the capacity constraint.
- Complexity $2^n$ is very high.
Knapsack Example

$w_1 = 7$
$v_1 = \$42$

$w_2 = 3$
$v_2 = \$12$

$w_3 = 4$
$v_3 = \$40$

$w_4 = 5$
$v_4 = \$25$
## Knapsack Example

<table>
<thead>
<tr>
<th>Subset</th>
<th>Total weight</th>
<th>Total value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>0</td>
<td>$0$</td>
</tr>
<tr>
<td>{1}</td>
<td>7</td>
<td>$42$</td>
</tr>
<tr>
<td>{2}</td>
<td>3</td>
<td>$12$</td>
</tr>
<tr>
<td>{3}</td>
<td>4</td>
<td>$40$</td>
</tr>
<tr>
<td>{4}</td>
<td>5</td>
<td>$25$</td>
</tr>
<tr>
<td>{1, 2}</td>
<td>10</td>
<td>$36$</td>
</tr>
<tr>
<td>{1, 3}</td>
<td>11</td>
<td>not feasible</td>
</tr>
<tr>
<td>{1, 4}</td>
<td>12</td>
<td>not feasible</td>
</tr>
<tr>
<td>{2, 3}</td>
<td>7</td>
<td>$52$</td>
</tr>
<tr>
<td>{2, 4}</td>
<td>8</td>
<td>$37$</td>
</tr>
<tr>
<td>{3, 4}</td>
<td>9</td>
<td>$65$</td>
</tr>
<tr>
<td>{1, 2, 3}</td>
<td>14</td>
<td>not feasible</td>
</tr>
<tr>
<td>{1, 2, 4}</td>
<td>15</td>
<td>not feasible</td>
</tr>
<tr>
<td>{1, 3, 4}</td>
<td>16</td>
<td>not feasible</td>
</tr>
<tr>
<td>{2, 3, 4}</td>
<td>12</td>
<td>not feasible</td>
</tr>
<tr>
<td>{1, 2, 3, 4}</td>
<td>19</td>
<td>not feasible</td>
</tr>
</tbody>
</table>
Complexity of TSP and Knapsack

Both TSP and Knapsack are so-called *NP-hard problem*. No polynomial-time algorithm is known for any NP-hard problem.

<table>
<thead>
<tr>
<th>Function</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
</tr>
<tr>
<td>$\log n$</td>
<td>logarithmic</td>
</tr>
<tr>
<td>$n$</td>
<td>linear</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>$n \log n$</td>
</tr>
<tr>
<td>$n^2$</td>
<td>quadratic</td>
</tr>
<tr>
<td>$n^3$</td>
<td>cubic</td>
</tr>
<tr>
<td>$2^n$</td>
<td>exponential</td>
</tr>
<tr>
<td>$n!$</td>
<td>factorial</td>
</tr>
</tbody>
</table>
Example 3: Assignment Problem

- \( n \) people to be assigned to execute \( n \) jobs, one person per job. \( C[i,j] \) is the cost if person \( i \) is assigned to job \( j \). Find an assignment with the smallest total cost

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person 1</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Person 2</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Person 3</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Person 4</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

- Exhaustive search:
  - How many kinds of different assignments?
  - The permutation of \( n \) persons → \( n! \) → Very high complexity
  - Hungarian method – much more efficient → polynomial
Assignment Problem by Exhaustive Search

\[ C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \]

- \( <1, 2, 3, 4> \) cost = \( 9 + 4 + 1 + 4 = 18 \)
- \( <1, 2, 4, 3> \) cost = \( 9 + 4 + 8 + 9 = 30 \)
- \( <1, 3, 2, 4> \) cost = \( 9 + 3 + 8 + 4 = 24 \)
- \( <1, 3, 4, 2> \) cost = \( 9 + 3 + 8 + 6 = 26 \)
- \( <1, 4, 2, 3> \) cost = \( 9 + 7 + 8 + 9 = 33 \)
- \( <1, 4, 3, 2> \) cost = \( 9 + 7 + 1 + 6 = 23 \)

etc.
Comments on Exhaustive Search

- Exhaustive-search algorithms run in a realistic amount of time only on very small instances.

- In some cases, there are much better alternatives!
  - Euler circuits
  - shortest paths
  - minimum spanning tree
  - assignment problem

- In many cases, exhaustive search or its variation is the only known way to get exact solution.
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.

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

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

\( dfs(v) \)
// 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 \( \text{count} \)
\[ \text{count} \leftarrow \text{count} + 1; \text{mark } v \text{ with } \text{count} \]

for each vertex \( w \) in \( V \) adjacent to \( v \) do
    if \( w \) is marked with 0
        \( dfs(w) \)
DFS Example – Undirected Graph

Input Graph

Stack push/pop

DFS forest

(Tree edge / Back edge)

02/07/2012
Applications of DFS

- To check the *connectivity* of a graph, check whether all the graph’s vertices have been visited after the dfs procedure halts
  - If so, the graph is connected; otherwise not connected
  - Can continue to find out each connected components

- To check the *acyclicity* of a graph, check if the DFS forest has any back edge
  - If not, the graph is acyclic; otherwise, the graph has a cycle
Notes on DFS

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

- Yields two distinct ordering of vertices:
  - preorder: as vertices are first encountered (pushed onto stack)
  - postorder: as vertices become dead-ends (popped off stack)