CSCE 411
Design and Analysis of Algorithms

Set 2: Brute Force & Exhaustive Search
Slides by Prof. Jennifer Welch
Spring 2014
Brute Force & Exhaustive Search

- Straightforward way to solve a problem, based on the definition of the problem itself; often involves checking all possibilities

**Pros:**
- widely applicable
- easy
- good for small problem sizes

**Con:**
- often inefficient for large inputs
Brute Force Sorting

- **Selection sort**
  - scan array to find smallest element
  - scan array to find second smallest element
  - etc.

- **Bubble sort**
  - scan array, swapping out-of-order neighbors
  - continue until no swaps are needed

- Both take $\Theta(n^2)$ time in the worst case.
Brute Force Searching

- Sequential search:
  - go through the entire list of \( n \) items to find the desired item
- Takes \( \Theta(n) \) time in the worst case
Brute Force Searching in a Graph

- (Review graph terminology and basic algorithms)
- Breadth-first search:
  - go level by level in the graph
- Depth-first search:
  - go as deep as you can, then backtrack
- Both take $\Theta(V+E)$ time, where $|V|$ is the number of vertices and $|E|$ is the number of edges
Brute Force for Combinatorial Problems

- Traveling Salesman Problem (TSP):
  - given a set of $n$ cities and distances between all pairs of cities, determine order for traveling to every city exactly once and returning home with minimum total distance

- Solution: Compute distance for all “tours” and choose the shortest.

- Takes $\Theta(n!)$ time (terrible!)
Do we need to consider more tours?
Something odd about the “distances”?
TSP Applications

- transportation and logistics (school buses, meals on wheels, airplane schedules, etc.)
- drilling printed circuit boards
- analyzing crystal structure
- overhauling gas turbine engines
- clustering data

tsp.gatech.edu/apps/index.html
iris.gmu.edu/~khoffman/papers/trav_salesman.html
Brute Force for Combinatorial Problems

- Knapsack Problem:
  - There are \( n \) different items in a store
  - Item \( i \) weighs \( w_i \) pounds and is worth \( v_i \)
  - A thief breaks in
  - He can carry up to \( W \) pounds in his knapsack
  - What should he take to maximize his haul?

- Solution: Consider every possible subset of items, calculate total value and total weight and discard if more than \( W \); then choose remaining subset with maximum total value.

- Takes \( \Omega(2^n) \) time
Knapsack Applications

- Least wasteful way to use raw materials
- Selecting capital investments and financial portfolios
- Generating keys for the Merkle-Hellman cryptosystem

Knapsack Example

- item 1: 7 lbs, $42
- item 2: 3 lbs, $12
- item 3: 4 lbs, $40
- item 4: 5 lbs, $25
- W = 10

- need to check 16 possibilities

<table>
<thead>
<tr>
<th>subset</th>
<th>total weight</th>
<th>total value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</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>$54</td>
</tr>
<tr>
<td>{1,3}</td>
<td>11</td>
<td>infeasible</td>
</tr>
<tr>
<td>{1,4}</td>
<td>12</td>
<td>infeasible</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>etc.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Brute Force For Closest Pair

Closest-Pair Problem:
- Given $n$ points in $d$-dimensional space, find the two that are closest

Applications:
- airplanes close to colliding
- which post offices should be closed
- which DNA sequences are most similar
Brute Force For Closest Pair

- Brute-force Solution (for 2-D case):
  - compute distances between all pairs of points
    - $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$
  - scan all distances to find smallest

- Running time: $\Theta(n^2)$, assuming each numerical operation is constant time (including square root?)

- Improvements:
  - drop the square root
  - don’t compute distance for same 2 points twice
Brute Force For Convex Hull

- Convex Hull Problem: Given a set of points in 2-D, find the smallest convex polygon s.t. each point in the set is enclosed by the polygon
  - polygon: sequence of line segments that ends where it begins
  - convex: all points on a line segment between 2 points in the polygon are also in the polygon
Convex Hull Applications

- In computer graphics or robot planning, a simple way to check that two (possibly complicated) objects are not colliding is to compute their convex hulls and then check if the hulls intersect.
- Estimate size of geographic range of a species, based on observations (geocat.kew.org/about)
Brute Force For Convex Hull

- Key idea for solution: line passing through \((x_i, y_i)\) and \((x_j, y_j)\) is:
  \[ ax + by = c \]
  where \(a = (y_j - y_i), b = (x_i - x_j), c = x_i y_j - y_i x_j \)
- The 2 pts are on the convex hull iff all other pts are on same side of this line:
Brute Force For Convex Hull

- For each (distinct) pair of points in the set, compute \(a\), \(b\), and \(c\) to define the line \(ax + by = c\).
  - For each other point, plug its \(x\) and \(y\) coordinates into the expression \(ax + by - c\).
  - If they all have the same sign (all positive or all negative), then this pair of points is part of the convex hull.

- Takes \(\Theta(n^3)\) time.
Brute Force for Two Numeric Problems

- Problem: Compute $a^n$
  - Solution: Multiply $a$ by itself $n-1$ times
  - Takes $\Theta(n)$ time, assuming each multiplication takes constant time.

- Problem: Multiply two $nxn$ matrices $A$ and $B$ to create product matrix $C$
  - Solution: Follow the definition, which says the $(i,j)$ entry of $C$ is $\sum a_{ik} \times b_{kj}$, $k = 1$ to $n$
  - Takes $\Theta(n^3)$ time, assuming each basic operation takes constant time.
Brute Force/Exhaustive Search Summary

- sorting: selection sort, bubble sort
- searching: sequential search
- graphs: BFS, DFS
- combinatorial problems: check all possibilities for TSP and knapsack
- geometric: check all possibilities for closest pair and for convex hull
- numerical: follow definition to compute $a^n$ or matrix multiplication
Applications of DFS

Now let’s go more in depth on two applications of depth-first search:

- topological sort
- finding strongly connected components of a graph
Depth-First Search

- **Input:** $G = (V,E)$
- for each vertex $u$ in $V$ do
  - mark $u$ as unvisited
  - $\text{parent}[u] := \text{nil}$
- $\text{time} := 0$
- for each unvisited vertex $u$ in $V$ do
  - $\text{parent}[u] := u$ // a root
  - call recursiveDFS($u$)

---

- **recursiveDFS($u$):**
  - mark $u$ as visited
  - $\text{time}++$
  - $\text{disc}[u] := \text{time}$
  - for each unvisited outgoing neighbor $v$ of $u$ do
    - $\text{parent}[v] := u$
    - call recursiveDFS($v$)
  - $\text{time}++$
  - $\text{fin}[u] := \text{time}$
Nested Intervals

- Let interval for vertex $v$ be $[\text{disc}[v], \text{fin}[v]]$.
- **Fact:** For any two vertices, either one interval precedes the other or one is enclosed in the other.
  - because recursive calls are nested
- **Corollary:** $v$ is a descendant of $u$ in the DFS forest if and only if $v$'s interval is inside $u$'s interval.
Classifying Edges

- Consider edge \((u, v)\) in directed graph \(G = (V, E)\) w.r.t. DFS forest
  - **tree edge**: \(v\) is a child of \(u\)
  - **back edge**: \(v\) is an ancestor of \(u\)
  - **forward edge**: \(v\) is a descendant of \(u\) but not a child
  - **cross edge**: none of the above
Example of Classifying Edges

In DFS forest:
- Tree edges:
  - a -> b
  - b -> c
  - c -> d
  - d -> e
- Back edges:
  - b -> a
- Cross edges:
  - b -> d

Not in DFS forest:
- Tree edges:
  - c -> f
  - f -> c
  - e -> d

Edges:
- Forward:
  - a -> b
  - b -> d

Graph:
- Nodes: a, b, c, d, e, f
- Edges:
  - a -> b
  - b -> c
  - c -> d
  - d -> e
  - b -> a
  - b -> d
  - c -> f
  - f -> c
  - e -> d
DFS Application: Topological Sort

- Given a directed acyclic graph (DAG), find a linear ordering of the vertices such that if (u,v) is an edge, then u precedes v.
- DAG indicates precedence among events:
  - events are graph vertices, edge from u to v means event u has precedence over event v
- Partial order because not all events have to be done in a certain order
Precedence Example

- Tasks that have to be done to eat breakfast:
  - get glass, pour juice, get bowl, pour cereal, pour milk, get spoon, eat.

- Certain events must happen in a certain order (ex: get bowl before pouring milk)

- For other events, it doesn't matter (ex: get bowl and get spoon)
Precedence Example

Order: glass, juice, bowl, cereal, milk, spoon, eat.
Why Acyclic?

Why must directed graph be acyclic for the topological sort problem?
Otherwise, no way to order events linearly without violating a precedence constraint.
Idea for Topological Sort Alg.

- Run DFS on the input graph

consider reverse order of finishing times:
spoon, bowl, cereal, milk, glass, juice, eat
Topological Sort Algorithm

input:  DAG G = (V,E)
1. call DFS on G to compute finish[v] for all vertices v
2. when each vertex's recursive call finishes, insert it on the front of a linked list
3. return the linked list

Running Time:  O(V+E)
Correctness of T.S. Algorithm

Show that if (u,v) is an edge, then v finishes before u finishes. Thus the algorithm correctly orders u before v.

Case 1: u is discovered before v is discovered. By the way DFS works, u does not finish until v is discovered and v finishes.

Then v finishes before u finishes.
Correctness of T.S. Algorithm

Show that if \((u,v)\) is an edge, then \(v\) finishes before \(u\) finishes. Thus the algorithm correctly orders \(u\) before \(v\).

\[ u \rightarrow v \]

Case 2: \(v\) is discovered before \(u\) is discovered. Suppose \(u\) finishes before \(v\) finishes (i.e., \(u\) is nested inside \(v\)).

Show this is impossible...
Correctness of T.S. Algorithm

- v is discovered but not yet finished when u is discovered.
- Then u is a descendant of v.
- But that would make (u,v) a back edge and a DAG cannot have a back edge (the back edge would form a cycle).
- Thus v finishes before u finishes.
DFS Application: Strongly Connected Components

- Consider a directed graph.
- A strongly connected component (SCC) of the graph is a maximal set of vertices with a (directed) path between every pair of vertices.
- Problem: Find all the SCCs of the graph.
What Are SCCs Good For?

- Packaging software modules:
  - Construct directed graph of which modules call which other modules
  - A SCC is a set of mutually interacting modules
  - Pack together those in the same SCC

- Solving the “2-satisfiability problem”, which in turn is used to solve various geometric placement problems (graph labeling, VLSI design), as well as data clustering and scheduling

www.cs.princeton.edu/courses/archive/fall07/cos226/lectures.html

wikipedia
SCC Example

four SCCs
How Can DFS Help?

- Suppose we run DFS on the directed graph.
- All vertices in the same SCC are in the same DFS tree.
- But there might be several different SCCs in the same DFS tree.
  - Example: start DFS from vertex h in previous graph
Main Idea of SCC Algorithm

- DFS tells us which vertices are reachable from the roots of the individual trees
- Also need information in the "other direction": is the root reachable from its descendants?
- Run DFS again on the "transpose" graph (reverse the directions of the edges)
SCC Algorithm

input: directed graph $G = (V,E)$

1. call DFS($G$) to compute finishing times
2. compute $G^T$ // transpose graph
3. call DFS($G^T$), considering vertices in decreasing order of finishing times
4. each tree from Step 3 is a separate SCC of $G$
SCC Algorithm Example

input graph - run DFS
After Step 1

Order of vertices for Step 3: f, g, h, a, e, b, d, c
After Step 2

transposed input graph - run DFS with specified order of vertices
After Step 3

SCCs are \{f,h,g\} and \{a,e\} and \{b,c\} and \{d\}. 
Running Time of SCC Algorithm

- Step 1: $O(V+E)$ to run DFS
- Step 2: $O(V+E)$ to construct transpose graph, assuming adjacency list rep.
- Step 3: $O(V+E)$ to run DFS again
- Step 4: $O(V)$ to output result
- Total: $O(V+E)$
Correctness of SCC Algorithm

- Proof uses concept of component graph, $G^{\text{SCC}}$, of G.
- Vertices are the SCCs of G; call them $C_1$, $C_2$, ..., $C_k$
- Put an edge from $C_i$ to $C_j$ iff G has an edge from a vertex in $C_i$ to a vertex in $C_j$
Example of Component Graph

based on example graph from before
Facts About Component Graph

- **Claim:** $G^{SCC}$ is a directed acyclic graph.
- **Why?**
  - Suppose there is a cycle in $G^{SCC}$ such that component $C_i$ is reachable from component $C_j$ and vice versa.
  - Then $C_i$ and $C_j$ would not be separate SCCs.
Facts About Component Graph

- Consider any component C during Step 1 (running DFS on G)
- Let \( d(C) \) be *earliest* discovery time of any vertex in C
- Let \( f(C) \) be *latest* finishing time of any vertex in C
- **Lemma**: If there is an edge in \( G^{\text{SCC}} \) from component \( C' \) to component C, then \( f(C') > f(C) \).
Proof of Lemma

- **Case 1:** $d(C') < d(C)$.
- Suppose $x$ is first vertex discovered in $C'$.
- By the way DFS works, all vertices in $C'$ and $C$ become descendants of $x$.
- Then $x$ is last vertex in $C'$ to finish and finishes after all vertices in $C$.
- Thus $f(C') > f(C)$.
Proof of Lemma

Case 2: $d(C') > d(C)$.

Suppose $y$ is first vertex discovered in $C$.

By the way DFS works, all vertices in $C$ become descendants of $y$.

Then $y$ is last vertex in $C$ to finish.

Since $C' \rightarrow C$, no vertex in $C'$ is reachable from $y$, so $y$ finishes before any vertex in $C'$ is discovered.

Thus $f(C') > f(C)$.
SCC Algorithm is Correct

Prove this theorem by induction on number of trees found in Step 3 (running DFS on $G^T$).

Hypothesis is that the first $k$ trees found constitute $k$ SCCs of $G$.

Basis: $k = 0$. No work to do!
SCC Algorithm is Correct

- **Induction:** Assume the first \( k \) trees constructed in Step 3 (running DFS on \( G^T \)) correspond to \( k \) SCCs; consider the \((k+1)\)st tree.

- Let \( u \) be the root of the \((k+1)\)st tree.

- \( u \) is part of some SCC, call it \( C \).

- By the inductive hypothesis, \( C \) is not one of the \( k \) SCCs already found and all so vertices in \( C \) are unvisited when \( u \) is discovered.
  - By the way DFS works, all vertices in \( C \) become part of \( u \)'s tree.
SCC Algorithm is Correct

- Show *only* vertices in C become part of u's tree. Consider an outgoing edge from C.
SCC Algorithm is Correct

- By lemma, in Step 1 (running DFS on $G$) the last vertex in $C'$ finishes after the last vertex in $C$ finishes.
- Thus in Step 3 (running DFS on $G^T$), some vertex in $C'$ is discovered before any vertex in $C$ is discovered.
- Thus in Step 3, all of $C'$, including $w$, is already visited before $u$'s DFS tree starts.