Greedy Algorithms, 1

- Algorithms for Optimization Problems
  - Sequence of steps
  - Choices at each step
- With Dynamic Programming finding the best choice can be expensive
- Simpler, more efficient algorithms will do
Greedy Algorithms, 2

A Greedy Algorithm always makes the choice that looks best at the moment.

It makes a locally optimal choice in the hope that it will lead to a globally optimal solution.

Optimal solutions are greedily achieved by Gas for many problems (not for all).
Greedy Properties

Properties an OP should exhibit to admit a greedy solution

1. Greedy-choice property
   - Global solution via local greedy choices

2. Optimal substructure
   - Optimal solution is obtained from optimal solutions to sub-problems
Greedy-choice Property

A globally optimal solutions can be arrived at by making a locally optimal greedy choice

- The choice that looks best is made independently of results from sub-problems

Main difference from Dynamic Programming

- Choices depends on results from sub-probs
GA vs. DP

**DP:** Solutions proceed bottom-up
- Progressing from smaller sub-problems to larger

**GA:** The best choice is made
- Proceed to solve the corresponding sub-problem
- → A greedy strategy proceeds top-down, one greedy choice after another, reducing a problem instance to a smaller one
Optimal Substructure

- A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to sub-problems.
- Common with DP.
- Cleverness is required to show that a greedy choice at each step yields a globally optimal solution.
GA vs. DP: An example

- Optimal substructure is common to GA and DP
  - Could lead to use DP when GA suffices or to use GA when DP is needed
- Subtleties in the difference can be illustrated by two variants of a classical optimization problem: The Knapsack Problem
0-1 Knapsack

- Thief robbing $n$ items from a store
  - Item $i$ is worth $v_i$ $\$$ and weighs $w_i$ pounds
    ($v_i$ and $w_i$ are integers)
- Thief can carry only up to $W$ pounds in his/her knapsack
- Which items should the thief take to maximize the load?
  (0-1: Items either can be taken or left)
Fractional Knapsack

- Thief robbing n items from a store
- Thief can carry only up to W pounds in his/her knapsack
- Thief can take fractions of items instead of making binary choices (like in 0-1)
Knapsack Optimal Substructure

Variations have optimal substructure

- **0-1**: Consider the most valuable load weighting $\leq W$ and remove item $j$, the remaining load must be the most valuable load weighting $\leq W - w_j$ from $n-1$ items.

- **Fractional**: If we remove a weight $w$ of an item $j$, the remaining load must be most valuable load weighting $\leq W - w$ from the $n-1$ items plus $w_j - w$ from item $j$. 
Solvability Issues

- **0-1 Knapsack** is not solvable by a GA
- **Fractional** is
  - Compute value per pound: \( v_i / w_i \)
  - Greedy strategy: Take the items with the greatest value per pound first, till knapsack is full
  - So, by sorting the item by \( v_i / w_i \) the whole process requires \( O(n \log n) \)
Steps of the Greedy Design

Greedy algorithms are designed according to a series of simple steps

1. Describe the OP so that a choice leads to one sub-problem to be solved
2. Prove that there is always an optimal solution that makes the greedy choice
   - The greedy choice is safe
3. Demonstrate that greedy choice + optimal solution to sub-problem = optimal solution to the problem
Examples: Dijkstra Algorithm for Shortest Paths

INPUT:

- A directed graph $G=(V,E)$
- Source $s$
- A weight function $w:E \rightarrow \mathbb{R}^+$
  - $w(u,v) \geq 0$, $(u,v) \in E$

Maintain a set $S \subseteq V$ whose final shortest-path weights from $s$ have been determined
Dijkstra Algorithm

Dijkstra(G,w,s)
Initialize-Single-Source(G,s)
S = 0
Q = V
while Q ≠ 0 do
  u = Extract-Min(Q) // GREEDY CHOICE HERE
  S = S ∪ {u}
  for each vertex v ∈ Adj[u] do Relax(u,v,w)
Building MSTs

- We will build a set $A$ of edges
- Initially, $A$ has no edges
- As we add edges to $A$, maintain a loop invariant:
  
  **Loop invariant:** $A$ is a subset of some MST

- Add only edges that maintain the invariant
- If $A$ is a subset of some MST, an edge $(u,v)$ is **safe** for $A$ if and only if $A \cup \{(u, v)\}$ is also a subset of some MST (add only safe edges)
Generic MST algorithm

GENERIC-MST(G,w)

A = 0

while A is not a spanning tree do

find an edge (u, v) that is safe for A

A = A \cup \{(u, v)\}

return A
Correctness

- We use the loop invariant
  - **Initialization**: The empty set trivially satisfies the loop invariant
  - **Maintenance**: Since we add only safe edges, A remains a subset of some MST
  - **Termination**: All edges added to A are in an MST, so when we stop, A is a spanning tree that is also an MST
Finding a Safe Edge

A cut \((S,V \setminus S)\) of an undirected graph \(G\) is a partition of \(V\).

An edge \((u,v)\) crosses the cut \((S,V \setminus S)\) if one of its endpoints is in \(S\) and the other in \(V \setminus S\).

A cut respects a set of edges \(A\) if no edge in \(A\) crosses the cut.

An edge is a light edge crossing the cut if its weight is the minimum among all those that cross the cut.
Recognizing Safe Edges

Theorem: Let $G = (V, E)$ be a connected, undirected graph, and $w: E \rightarrow \mathbb{R}$. Let $A \subseteq E$ included in some MST of $G$. Let $(S, V \setminus S)$ any cut of $G$ that respects $A$ and let $(u, v)$ be a light edge crossing $(S, V \setminus S)$. Then edge $(u, v)$ is safe for $A$. 
Analysis of GENERIC-MST

- A is a forest containing connected components. Initially, each component is a single vertex.
- Any safe edge merges two of these components into one. Each component is a tree.
- Since an MST has exactly $|V|-1$ edges, the for loop iterates $|V|-1$ times. Equivalently, after adding $|V|-1$ safe edges, we are down to just one component.
Prim’s Algorithm for MST

- Builds one tree, so A is always a tree
- Starts from an arbitrary “root” r
- At each step, find a light edge crossing cut \((V_A, V\setminus V_A)\), where \(V_A\) = vertices that A is incident on
- Add this edge to A
Selecting Edges Efficiently

Use a priority queue Q:

- Each object is a vertex in $V \setminus V_A$
- Key of $v$ is minimum weight of any edge $(u,v)$, where $u \in V_A$
- The vertex returned by EXTRACT-MIN is $v$ such that there exists $u \in V_A$ and $(u,v)$ is a light edge crossing $(V_A, V \setminus V_A)$
- Key of $v$ is $\infty$ if $v$ is not adjacent to any vertices in $V_A$
Prim’s MST

The edges of A will form a rooted tree with root r:

- r is given as an input to the algorithm, but it can be any vertex
- Each vertex knows its parent in the tree by the attribute \( \pi[v] = \text{parent of } v \). \( \pi[v] = \text{NIL} \) if \( v = r \) or \( v \) has no parent
- As algorithm progresses, \( A = \{(v, \pi[v]) : v \in V \setminus \{r\} \setminus Q\} \)
- At termination, \( V_A = V \Rightarrow Q = 0 \), so MST is \( A = \{(v, \pi[v]) : v \in V \setminus \{r\}\} \)
Prim, the Algorithm

PRIM(G,w,r)
    for each u ∈ V do key[u]=∞; π[u]=NIL
    key[r]=0; Q=V
    while Q≠0 do
        u=EXTRACT-MIN(Q) // GREEDY CHOICE!
        for each v ∈ Adj[u] do
            if v ∈ Q and w(u,v) < key[v]
            then π[v]=u
            key[v]=w(u, v)
Huffman Codes

- Effective technique for compressing data (20-90% savings)
- Data = sequence of characters
- Uses a table of frequencies to build an optimal way of representing the data as a binary string
- Design a binary character code where each character is represented by a unique binary string
  - Fixed-length codes vs. variable length codes
Prefix Codes

- Prefix codes are codes in which no codeword is a prefix of some other codeword
  - No loss of generality
- Prefix codes simplify decoding
  - The codeword that begins an encoded file is unambiguous
Constructing a Huffman Code

- The Huffman Code is an optimal prefix code

**Assumptions**

- $C$ is a set of $n$ characters
- Every $c \in C$ has a frequency $f[c]$
- The tree corresponding to the optimal prefix code is built bottom-up
  - Begins with $|C|$ leaves and performs $|C|-1$ merging operations to create the tree
Huffman, the Algorithm

Huffman(C)
Q = C
for i = 1 to n-1 do
allocate a new node z
left[z] = x = Extract-Min(Q)
right[z] = y = Extract-Min(Q)
f[z] = f[x] + f[y]
insert(Q,z)
return Extract-Min(Q)
Analysis

- Q is implemented as a binary min-heap
- Initialization of Q is $O(n)$
- The loop contributes $O(n \log n)$
  - Executed $O(n)$ times
  - Each time heap operations require $O(\log n)$
- Total running time for $n$ characters
  $O(n \log n)$
Correctness

Lemma 1: Let $C$ be an alphabet where each $c \in C$ has frequency $f[c]$. Let $x$ and $y \in C$ be the characters with the lowest frequencies. Then there exists an optimal prefix code for $C$ in which the codewords for $x$ and $y$ have the same length and differ only in one bit.
“Greediness”

Lemma 1 → Merging the two characters with the lowest frequency is greedy and leads to an optimal tree.

It is greedy: Of all possible mergers at each step, Huffman chooses the one with minimal cost.

There is a Lemma 2 for showing optimal substructure.
Assignments

- Textbook, Chapter 16, pages 370—392
- Updated information on the class web page:
  www.ece.neu.edu/courses/eceg205/2004fa
Happy Thanksgiving!