Greedy Algorithms

- **Problem:** set of \( n \) activities \( s_i, f_i \), start and end of activity \( i \).
- i incompatible with \( j \) if intervals do not intersect.
- Goal: find max # of compatible activities.

Let \( k \) have smallest \( f \) and let \( A \) be OPT solution.

Case 1: \( k \) in OPT. Claim: \( A-k \) is OPT for \( A \).

Assume not, let \( B \) be OPT for \( S' \), \( |B| > |A|-1 \).

But then add \( k \) to \( B \) and we get better than \( A \)!
Thus we compute \( k \), commit to it, compute \( S' \), and repeat!

Summary

- Take locally best choice and commit to it.
- Main issue: proof that we can commit without losing our chance to get an optimum solution.

Another greedy algorithm

- **Task defined by (duration, deadline), eg. HW.**
- **Goal:** find a schedule if one exists.

Assume that there exists a schedule

Claim: then there exists a schedule with:

- first job = job with smallest deadline.

We can exchange \( b \) and \( a \)!!

Huffman encoding

- Idea: represent often encountered letters by shorter codes.
- Prefix code: a code for \( x \) is not a prefix for any code-word for \( y \).

- In this example: \( c=010, e=1100 \)

Huffman encoding

- Assume we know symbol frequencies:

<table>
<thead>
<tr>
<th></th>
<th>50</th>
<th>40</th>
<th>5</th>
<th>3</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

- 50:1+40:2, 5:3, 3:4, 2:4 = 165, 1.65b/symbol instead of 3!

- Assume that \( a \) is a very common symbol.

Now: \( a = 0 \)
\( b = 100 \)
\( e = 1100 \)
Generating optimum encoding

- Claim: Let x and y be the lowest freq. characters. Then there exists a code where x and y differ only in 1 bit.

So does this mean that we do not need any other codes? [Hint: consider a sequence 1001001000100...]

Min-Cost Spanning Tree

- Applications:
  - Cable TV,
  - VLSI,
  - basic task for many optimization algts (eg. flow).

- Formally:
  - Undirected graph G=(V,E).
  - Weights w: E → R
  - Goal: find spanning tree of minimum weight.
    (spanning = connects all nodes in G)

Example

Example MST

Optimum Substructure

- Assume T is MST of any subtree of G

- Proof: "Cut-and-Paste" approach
  - Replace uV by wV

- Questions:
  - Why no more edges parallel to uV in T?? - Cycles!
  - Why uV exists at all ?? (walk in T until you hit [T-A]

Prim's Algorithm

- Main idea:
  - Pick a node v, set A={v}.
  - Repeat:
    - Find min-weight edge outgoing from A.
    - Add v to A.

- Need support for finding an edge that is:
  - outgoing.
  - Min-weight among all outgoing.
Implementing Prim's Alg

- First try:
  - Keep all edges ( outgoing and internal) from A in a heap.
  - New node: add all its edges to the heap.
- To get "next edge":
  - extract min-weight from heap.
  - check if internal. (how ???)
  - If yes, discard and repeat.

Time: $O(E)$ insertions and $O(E)$ deletions from heap:
$$\text{Total: } O(E \log V)$$

More about implementation

- Only $V-1$ edges were used, the rest - wasted.

Idea:
- Keep nodes in the heap, instead of edges.
- Key: distance of node from $A$ over a single edge.
- Initially: $\text{key}(v) = \infty$ for all $v$.

$x = \text{root}$
Repeat:
- $\forall v \in \text{set}_x$ do:
  - $\text{key}(v') = \min(\text{key}(v), \text{w}(x))$
- Pick smallest-key $x$, add $x$ to $A$
- So why does this work ????

Alternative Implementations

- Total: $O(E)$ decrease-key, $O(V)$ extract-min.

<table>
<thead>
<tr>
<th></th>
<th>extract-min</th>
<th>decrease-key</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>$O(V)$</td>
<td>$O(1)$</td>
<td>$O(E \log V)$</td>
</tr>
<tr>
<td>heap</td>
<td>$O(\log V)$</td>
<td>$O(V)$</td>
<td>$O(E \log V)$</td>
</tr>
<tr>
<td>Fib. heap</td>
<td>$O(\log V)$</td>
<td>$O(1)$</td>
<td>$O(V \log V + E)$</td>
</tr>
</tbody>
</table>

Kruskal's Algorithm

- Main loop:
  - scan edges in increasing order of weight
  - put edge in if no loop created.

Why does this result in MST ??
- Observation: min-weight edge is always in MST.

Proof: Assume there exists a tree without this edge.
- Add this edge to the tree - this creates a cycle.
- Delete max-weight edge on this cycle, we get a lighter tree !

Proof of Kruskal's algorithm

- Consider the instant when we are adding the first wrong edge, i.e. edge $xy$ that is not in any optimum tree:
  - blobs are current connected components.
  - There exists a path from $x$ to $y$ in the optimum tree.
  - $u$ and $v$ are not in our tree, thus they are heavier than $xy$ !
  - cut-and-paste to get a better tree, contradiction.

Implementation

- Given two nodes $u$ and $v$, need to know if they are in the same connected component, i.e. in the same set.

$$\text{Find-Set}(v)$$
- After adding edge $uv$, need to merge the set that includes $u$ with the set that includes $v$.

$$\text{Union}(\text{Find-Set}(u), \text{Find-Set}(v))$$
- Total: $O(V)$ Make-Set
  $$O(V) \text{ Find-Set}$$
  $$O(V) \text{ Union}$$
- Section 22.4 explains how to achieve these ops in $O(E, V)$ time.
  - where $g$ is inverse Ackermann function.

$$\text{log}_2 \log_{10} g(m,n) \leq 5 \text{ for } m,n \leq 10^6$$
Simple Union-Find Implementation

- **Main idea:**
  - Maintain every set as a linked list.
  - Maintain every element points to head of the list.
- **Work:**
  - Find-Set takes $O(1)$.
  - Union: $O(1)$ per element of the smaller list.
  - Each time an element is charged during union, his set at least doubles.
  - Total: $O(V \log V)$ work for all unions.

- **Total time:** $O(E + V \log V)$

Dynamic Programming

- **Main problem with greedy approaches:**
  - Sometimes we can not commit up-front.
- **Dynamic programming:**
  - Meta-technique, not a specific algorithm.
  - **Main idea:**
    - Solve many small subproblems.
    - Combine solution to several small subproblems to solve larger subproblems.
    - Continue combining until we solve the original problem.

Single-Source Shortest Paths

- **Main observation:**
  - If shortest path $s \to u$ goes through $v$, then its part up to $v$ is the shortest path from $s$ to $v$.

Bellman-Ford

- **Early termination:**
  - We can terminate at phase $k$ if, for all $v$.
  - $d(v) = d^2(v)$ since no more changes will happen in $d(v)$ for larger values of $k$.
  - (might terminate earlier than after $n-1$ phases)

- **Example:**
  - Matrix chain multiplication

Another example:

- **Consider the following chain:**
  - $A_{1,1} \times A_{2,2} \times \ldots \times A_{n,n}$
  - $A_{i,j}$ is an $i \times j$ matrix
  - $\text{time} = \sum_{i=1}^{n} m_{i-1} m_{i} n_{i}$

- **Example:**
  - $[5 \times 100] \times [100 \times 2] \times [2 \times 50]$

- **Order of multiplication affects the amount of work!**
Solving matrix chain multiplication

- Observation:
  - Consider last optimum multiplication: \( A_{j-1} \ldots A_n \).
  - Then both \( A_{j-1} \ldots A_k \) and \( A_k \ldots A_n \) were computed optimally!!
  - Why??

- Subproblems:
  - is best "time" to multiply

- Answer is \( m(1, n) \)

- Why can't we just use as subproblems the time to multiply matrices 1 to \( i \)??

\[
\begin{align*}
\text{if } i &\leq j \text{, then } C_{i,j} = 0 \\
\text{if } j < i < k &\text{, then } C_{i,j} = \min(C_{i,k} + C_{k+1,j}, C_{i,k-1} + C_{1,j}) \\
\text{if } i &\leq k \leq j \text{, then } C_{i,j} = \min(C_{i,k} + C_{k+1,j}, C_{i,k-1} + C_{1,j})
\end{align*}
\]

Matrix chain continued

- Let's try to analyze using recurrence relation:
  - \( C_{i,j} = \sum_{k=i}^{j} C_{i,k} + C_{k+1,j} \)
  - Why??

- Wrong approach! There are only \( O(n^2) \) different subproblems!

- Build the table bottom up, for increasing \( j-I \).
  - \( O(n) \) per each, total \( O(n^3) \).

Summary - Dynamic Programming

- Find optimum substructure.
- Define subproblems (not too many of them!)
- Organize subproblems into a table.
- Make sure there is a way to fill the table.

Longest common-subsequence

- Consider two sequences:
  - \( x = A B C A B \) \( |x| = m \)
  - \( y = B D C A B \) \( |y| = n \)

- Greedy: does not work! (Why??)

- Brute force: take any substring of \( x \), check against \( y \).
  - Total: \( O(2^m n) \), too slow!

Optimum Substructure

- Define subproblem: \( \text{LC}_{(x,y)} \) \( x_1 \ldots x_i, y_1 \ldots y_j \) \( i \leq j \)
- Observe that \( \text{LC}_{(x,y)} \) is the answer that we seek.

- Theorem:
  - \( \text{LC}_{(x,y)} = \text{LC}_{(x',y')} \) \( x' \subseteq x, y' \subseteq y \)
  - \( \text{LC}_{(x,y)} = \text{LC}_{(x',y')} \) \( x' \subseteq x, y' \subseteq y \)
  - \( \text{LC}_{(x,y)} = \text{LC}_{(x',y')} \) \( x' \subseteq x, y' \subseteq y \)

Proof: Case 1, \( x \neq y \)

- Theorem:
  - \( \text{LC}_{(x,y)} = \text{LC}_{(x',y')} \) \( x' \subseteq x, y' \subseteq y \)
  - \( \text{LC}_{(x,y)} = \text{LC}_{(x',y')} \) \( x' \subseteq x, y' \subseteq y \)
  - \( \text{LC}_{(x,y)} = \text{LC}_{(x',y')} \) \( x' \subseteq x, y' \subseteq y \)

Case 2:
- \( x_1 = y_1 \)
  - \( z_1 = x_1 \) \( (2a) \)
  - \( z_1 = y_1 \) \( (2b) \)

Proof: continued
Recursive algorithm

- We can use the theorem to construct a recursive algorithm. Consider its tree:

        3,4
     /   /
  1.3  1.2  1.3  1.2
 /   /   /   /
1.5 1.6 2.3 2.5

Analysis

- Depth of the tree is $O(m+n)$, leads to $O(3^{m+n})$ bound, too large!
- Main idea: we see repeating sub-question, only $O(mn)$ different ones!
- Memoization: after computing sub-problem answer, remember it.
- Dynamic programming: compute the table bottom-up.

Computing the table

- Fill the table starting from top-left corner, and going row-by-row:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

- Each element depends on the one above, one left, and if $x_i = y_i$, then it is one more than the diagonal up-left element.

Knapsack Problem

- Problem statement:
  1. We have $n$ items, $j$-th item costs $c(j)$ and weights $w(j)$.
  2. We have a knapsack that can hold total $W$ weight.
  3. Goal: maximize total value of items that we choose to put into the knapsack, without exceeding total allowed weight $W$.
- Abstraction of many real problems: from investing to telephone call routing.
- Fractional (allowed to take part of an item): easy! do greedy, choose best value-per-weight element.

Fractional vs. Integer Knapsack

- Consider the following example:
  -Greedy: $#1+#2$ gives $100$
  -Optimum: $#2+#3$, gives $110$
  -Fractional: $#1+#2 + 3/5$ of $#3$, gives $120$.

Optimum substructure:

Consider optimum solution: $x_1, x_2, ..., x_k$

Where $x_i = 0$ means we do not take the item, and $x_i = 1$ means we take it.

Claim: $x_1, x_2, ..., x_k$ is optimum for $S$, $w(S) = m(W, A)$.

Solving Knapsack

- Subproblems:
  - $C(i,w) =$ OPT solution using items 1 to $i$, knapsack $w$.
  - $C(i,w) =$ if $w \geq w(S)$, take $w$;
    otherwise

- Table size in $nW$, $O(1)$ per element, $\text{TOTAL} = O(nW)$
- But knapsack is N.P.-Hard!
  - Do we indeed have a contradiction here??

No contradiction since $W$ is not polynomial in the size of the input...
Graph Algorithms

Examples of graph problems:
- Direct applications:
  - City streets map: reachability, shortest path, congestion management
  - Communication networks: planning, fault-tolerance, reliability, topology alignment
- Indirect applications:
  - Assigning tasks to humans
  - Scheduling jobs on a multiprocessor
  - Searching within space
- Restate as a graph problem, solve
  - Map back

Depth First Search

- Visiting(u):
  - color(u) = gray; d(u) = time; time++;
  - for each neighbor w of u:
    - if w is white then Visit(w)
    - color(u) = black; f(u) = time; time++;

- Initially, set all nodes white.
- Examine nodes one-by-one, call Visit if node is still white.
- Node visited once, edge touched twice: Running time O(n+m)

At home: Read theorems 23.6 and 23.8 (we will only sketch the proofs)

Edge Classification

Classification of uw according to (color of u) -> (color of w): (when the edge is considered)
- Tree edge: gray -> white
- Back edge: gray -> gray
- Forward: gray -> black, u ancestor of w.
- Cross: other gray -> black edges.

How to distinguish forward and cross edges??
We can use d() time!

Parenthesis Theorem

Theorem:
For any two nodes u and v, the two intervals [d(u),f(u)] and [d(v),f(v)] either:
- Do not intersect, or
- [d(u),f(u)] includes [d(v),f(v)], v descendant of u, or
- [d(v),f(v)] includes [d(u),f(u)], u descendant of v.

Proof:
- Assume (wlog) d(u)<d(v).
- If v was not discovered before finishing u, then we have case 1 above.
- If v was discovered, then we have to finish it before returning and finishing u, leading to case 2.
- Case 3 is symmetric.

White-Path Lemma

In (directed or undirected) graph G, node v is descendant of u iff at d(u) (time when u was discovered) there is a path from v to u using only currently white nodes.

Proof:
- Assume v is descendant of u.
- Let w be edge on the uv path in the tree.
- If w was not white at d(u), then we will not be true edge.
- Thus, all nodes on the uv path are white when u is discovered.
- Assume that at d(u) there is a white path from u to v.
- Let w be the first edge on this path, with w closest to u so that w is descendant of u but w is not.
- Let w' be edge of the starting and before finishing u.
- By parenthesis theorem, w' is also a descendant of u, contradiction.

Simple Lemma

Lemma: If G is undirected, then only tree and back edges.
Proof: wlog, d(u)<d(v).
- Thus v must be discovered and finished before finishing u, since uv exists.
- If uv discovered from u, before v, it is a tree edge.
- If v was discovered before uv, then uv becomes a back edge.

Why does the proof break down in the directed case?
Discovering Cycles

- Claim: G acyclic if DFS yields no back edges.
- Proof:
  - Trivial to observe that back edge implies a cycle.
  - Assume there exists a cycle:
    - Let v be the node with smallest d on the cycle, and let u be an edge of the cycle.
    - All nodes in the cycle, including v, are white.
    - This, when v is scanned, we will discover an edge mark it as "back edge".

Topological Sort

- Directed acyclic graph G.
- Algorithm:
  - Call DFS to compute finishing times f[v] for each vertex v.
  - As each v is finished, insert it onto the front of finished list
  - Return the finished list.
- Claim: the output list is a legal topological sort.
  - Sufficient to prove that, for every u and v s.t. (uv) is an edge, we have f[v] < f[u]. (Why?)
  - Consider edge (uv) explored by DFS.
  - Observe that when (uv) is explored, v cannot be gray (back edge implies cycle)
  - If v white, it becomes descendant of u, and thus f[v] > f[u].
  - If v black, it finished before u started, so again f[v] > f[u].

Back to shortest paths: Dijkstra’s Algorithm

- We can do better than Bellman-Ford if no negative-weight edges.
- Algorithm:
  - Main idea: add node with shortest perceived distance.
- Time: n extract_min, m decrease_key
  - FB. Heap: O(m log n)