Greedy

This course - 91.503, Analysis of Algorithms - follows 91.404 (Undergraduate Analysis of Algorithms) and assumes the students familiar with the detailed contents of that course. We will not re-cover any of the materials in the previous course, but we will use the details of them whenever appropriate. The Algorithms Qualifying Exam covers the contents of BOTH courses.

We start off with some coverage of Greedy Methods (Activity Selection, Knapsack Problem, Huffman Codes, etc.). Please look at the appropriate sections in the text to refresh your memories.

Greedy

What is a Greedy Algorithm?

Solves an optimization problem: the solution is “best” in some sense.

Greedy Strategy:
- At each decision point, do what looks best “locally”
- Choice does not depend on evaluating potential future choices or solving subproblems
- Top-down algorithmic structure
  - With each step, reduce problem to a smaller problem

Optimal Substructure:
- optimal solution contains in it optimal solutions to subproblems

Greedy Choice Property:
- “locally best” = globally best

Greedy Examples:
- Minimum Spanning Tree
- Dijkstra Shortest Path
- Huffman Codes
- Fractional Knapsack
- Activity Selection

Minimum Spanning Tree

Produces minimum weight tree of edges that includes every vertex.

Time:
O(|E|lg|E|) given fast
FIND-SET, UNION

Time:
O(|E|lg|V|) = O(|E|lg|E|) faster with fast priority queue

source: 91.503 textbook Cormen et al.
Greedy

Single Source Shortest Paths: Dijkstra’s Algorithm

for (nonnegative) weighted, directed graph G=(V,E)

Dijkstra(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 S ← ∅
3 Q ← V[G]
4 while Q ≠ ∅
5 do u ← EXTRACT-MIN(Q)
6 S ← S ∪ {u}
7 for each vertex v ∈ Adj[u]
8 do RELAX(u, v, w)

Greedy

Huffman Codes

Code table:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1111</td>
</tr>
<tr>
<td>b</td>
<td>1111</td>
</tr>
<tr>
<td>c</td>
<td>1111</td>
</tr>
<tr>
<td>d</td>
<td>1111</td>
</tr>
<tr>
<td>e</td>
<td>1111</td>
</tr>
<tr>
<td>f</td>
<td>1111</td>
</tr>
<tr>
<td>g</td>
<td>1111</td>
</tr>
</tbody>
</table>

Greedy

Fractional Knapsack

Value: $60 $100 $120

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>item1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>item2</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>item3</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

“knapsack”

Greedy

Activity Selection

Problem Instance:

- Set S = {1,2,..,n} of n activities
- Each activity i has:
  - start time: s_i
  - finish time: f_i
- Activities i, j are compatible iff non-overlapping: [s_i ≤ f_i] [s_j ≤ f_j]
- Objective:
  - select a maximum-sized set of mutually compatible activities

11/28/07
Greedy Algorithm

Algorithm:
- \( S' = \) presort activities in \( S \) by nondecreasing finish time
  - and renumber
- GREEDY-ACTIVITY-SELECTOR(\( S' \))
  - \( n \leftarrow \text{length}(S') \)
  - \( A \leftarrow \{1\} \)
  - \( j \leftarrow 1 \)
  - for \( i \leftarrow 2 \) to \( n \)
    - if \( s_i < f_j \)
      - then \( j \leftarrow i \)
    - \( A \leftarrow A \cup \{i\} \)
  - return \( A \)

Running time?

Greedy

Why does this all work?

Why does it provide an optimal (maximal) solution? It is clear that it will provide a solution (a set of non-overlapping activities), but there is no obvious reason to believe that it will provide a maximal set of activities.

We start with a restatement of the problem in such a way that a dynamic programming solution can be constructed. The dynamic programming solution will be shown to be maximal. It will then be modified into a greedy solution in such a way that maximality will be preserved.

Greedy

Consider the set \( S \) of activities, sorted in monotonically increasing order of finishing time:

\[
\begin{array}{cccccccccccc}
  i & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
  s_i & 1 & 3 & 0 & 5 & 3 & 5 & 6 & 8 & 8 & 2 & 12 \\
  f_i & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\end{array}
\]

Where the activity \( a_i \) corresponds to the time interval \( [s_i, f_i) \). The subset \( \{a_3, a_5, a_{11}\} \) consists of mutually compatible activities, but is not maximal. The subsets \( \{a_7, a_9, a_{11}\} \) and \( \{a_2, a_4, a_9, a_{11}\} \) are also compatible and are maximal.

First Step: find an optimal substructure. You need to define an appropriate space of subproblems.

\( S_{ij} = \{a_k \in S: f_i \leq s_k < f_k \leq s_j\} \)

is the set of all those activities compatible with \( a_i \) and \( a_j \) and that are also compatible with those activities that finish no later than \( a_i \) and start no earlier than \( a_j \).

Add two activities:

\( a_0: (-\infty, 0); \ a_{n+1}: (\text{inf, } \text{inf}); \)

which come before and after, respectively, all activities in \( S = S_{0,n+1} \).

Now assume the activities are sorted in non-decreasing order of finish: \( f_0 \leq f_1 \leq \ldots \leq f_n < f_{n+1} \).

• Then \( S_{ij} \) is empty whenever \( i \geq j \).
Greedy

From the original problem we now introduce a space of suproblems: find a maximum size subset of mutually compatible activities from \( S_{ij} \), \( 0 \leq i < j \leq n+1 \).

We now need to see the substructure of the problem, and show that it possesses the "optimal substructure property"

- **Substructure**: suppose a solution to \( S_{ij} \) contains some activity \( a_k \), \( f_i \leq s_k < f_k \leq s_j \). \( a_k \) generates two subproblems, \( S_{ik} \) and \( S_{kj} \). The solution to \( S_{ij} \) is the union of a solution to \( S_{ik} \), the singleton \( \{a_k\} \), and a solution to \( S_{kj} \). So any solution to a larger problem can be obtained by patching together solutions for smaller problems. Cardinality is also additive.

Greedy

- **Optimal Substructure**: assume \( A_{ij} \) is an optimal solution to \( S_{ij} \) containing an activity \( a_k \). Then \( A_{ij} \) contains a solution to \( S_{ik} \) (the activities that end before the beginning of \( a_k \)) and a solution to \( S_{kj} \) (the activities that begin after the end of \( a_k \)) . If these solutions were not already maximal, then one could obtain a maximal solution for, say, \( S_{ik} \), and splice it with \( \{a_k\} \) and the solution to \( S_{kj} \) to obtain a solution for \( S_{ij} \) of greater cardinality, thus violating the optimality condition assumed for \( A_{ij} \).

Greedy

- **Use Optimal Substructure** to construct an optimal solution:
  - Any solution to a nonempty problem \( S_{ij} \) includes some activity \( a_k \), and any optimal solution to \( S_{ij} \) must contain optimal solutions to \( S_{ik} \) and \( S_{kj} \).
  - For each \( a_k \) in \( S_{0,n+1} \), find (recursively) optimal solutions of \( S_{ik} \) and \( S_{kj} \), say \( A_{ik} \) and \( A_{kj} \). Splice them together, along with \( a_k \), to form solutions \( A_k \). Take a maximal \( A_k \) as an optimal solution \( A_{0,n+1} \).

  We need to look at the recursion in more detail...

Greedy

- **Second Step**: the recursive algorithm.
  - Let \( c[i,j] \) denote the maximum number of compatible activities in \( S_{ij} \). It is easy to see that \( c[i,j] = 0 \) whenever \( i \geq j \) - and \( S_{ij} \) is empty.
  - If, in a non-empty set of activities \( S_{ij} \), the activity \( a_k \) occurs as part of a maximal compatible subset, this generates two subproblems, \( S_{ik} \) and \( S_{kj} \), and the equation
    \[
    c[i,j] = c[i,k] + 1 + c[k,j],
    \]
    which tells us how the cardinalities are related. The problem is that \( k \) is not known a priori. Solution:
    \[
    c[i,j] = \max_{i<k<j} c[i,k] + 1 + c[k,j], \text{ if } S_{ij} \text{ is not empty.}
    \]
    And one can write an easy "bottom up" (dynamic programming) algorithm to compute a maximal solution.
Greedy

**A better mousetrap.**

Theorem 16.1. Consider any nonempty subproblem \( S_{ij} \) and let \( a_m \) be the activity in \( S_{ij} \) with the earliest finish time: \( f_m = \min \{ f_k : a_k \text{ in } S_{ij} \} \). Then

1. \( a_m \) is used in some maximum size subset of mutually compatible activities of \( S_{ij} \).
2. The subproblem \( S_{im} \) is empty, so that choosing \( a_m \) leaves the subproblem \( S_{mj} \) as the only one that may be nonempty.

Proof: 2) if \( S_{im} \) is nonempty then \( S_{ij} \) must contain an activity prior to \( a_m \). Contradiction.

1) Suppose \( A_{ij} \) is a maximal subset. Either \( a_m \) is in \( A_{ij} \), and we are done, or its is not. Let \( a_i \) be the earliest finishing activity in \( A_{ij} \). It can be replaced by \( a_m \) (why?) thus giving a maximal subset containing \( a_m \).

Greedy

The Algorithm:

\[ \text{R-A-S}(s, f, i, j) \]

1. \( m := i + 1 \)
2. while \( m < j \) and \( s_m < f_i \) // find first activity in \( S_{ij} \)
3. \( \text{do } m := m + 1 \)
4. if \( m < j \)
5. \( \text{then return } \{a_m\} + \text{R-A-S}(s, f, m, j) \)
6. \( \text{else return the empty set.} \)

The time is - fairly obviously - Theta(n).
Greedy

A Slightly Different Perspective.
Rather than start from a dynamic programming approach, moving to a greedy strategy, start with identifying the characteristics of the greedy approach.

1. Make a choice and leave a subproblem to solve.
2. Prove that the greedy choice is always safe - there is an optimal solution starting from a greedy choice.
3. Prove that, having made a greedy choice, the solution of the remaining subproblem can be added to the greedy choice providing a solution to the original problem.

Greedy Choice Property.
Choice that looks best in the current problem will work: no need to look ahead to its implications for the solution. This is important because it reduces the amount of computational complexity we have to deal with.

Can’t expect this to hold all the time: maximization of a function with multiple relative maxima on an interval - the gradient method would lead to a “greedy choice”, but may well lead to a relative maximum that is far from the actual maximum.

Greedy

Optimal Substructure.
While the greedy choice property tells us that we should be able to solve the problem with little computation, we need to know that the solution can be properly reconstructed: an optimal solution contains optimal “sub-solutions” to the subproblems. An induction can then be used to turn the construction around.

In the case of a problem possessing only the Optimal Substructure property there is little chance that we will be able to find methods more efficient than the dynamic programming (with memoization) methods.

Greedy

Some Theoretical Foundations: unifying the individual methods.

Matroids. A matroid is an ordered pair $M = (S, I)$ s.t.
1. $S$ is a finite non-empty set;
2. $I$ is a non-empty family of subsets of $S$, called the independent subsets of $S$, such that $B \subseteq I$ and $A \subseteq B$, then $A \in I$. We say that $I$ is hereditary if it satisfies this property. Note that $\emptyset$ is a member of $I$.
3. If $A \in I$ and $B \in I$, and $|A| < |B|$, then there is an element $x \in B - A$ such that $A \cup \{x\} \in I$. We say that $M$ satisfies the exchange property.

Ex.: the set of rows of a matrix.
Greedy

**Graphic Matroids:** $M_G = (S_G, I_G)$, where we start from an undirected graph $G=(V, E)$.

- The set $S_G$ is defined to be $E$, the set of edges of $G$;
- If $A$ is a subset of $E$, then $A \in I_G$ is and only if $A$ is acyclic - i.e., a set of edges is independent if and only if the subgraph $G_A = (V, A)$ forms a forest.

We have to prove that the object so created ($M_G$) is, in fact, a matroid. The relevant theorem is:

**Greedy**

**Theorem.** If $G$ is undirected graph, then $M_G = (S_G, I_G)$ is a matroid.

**Proof.** $S_G = E$ is finite;

$I_G$ is hereditary, since a subset of a forest is still a forest (removing edges cannot introduce cycles).

We are left with showing $M_G$ satisfies the exchange property. Let $G_A = (V, A)$ and $G_B = (V, B)$ be forests of $G$, with $|B| > |A|$. (A and B are acyclic sets of edges with B containing more edges than A).

Claim: a forest with $k$ edges has exactly $|V| - k$ trees.

**Proof:** start with a forest with no edges and add one edge at a time.

$G_A$ contains $|V| - |A|$ trees, while $G_B$ contains $|V| - |B|$ trees: $G_A$ contains more trees than $G_B$.

**Greedy**

$G_A$ must contain some tree $T$ whose vertices are in two trees of $G_A$ (everything is acyclic...). Since $T$ is connected, it must contain an edge $(u, v)$ such that vertices $u$ and $v$ are in different trees of $G_A$, and so can be added to $G_A$ without creating a cycle.

But this is exactly the exchange property.

All three properties are satisfied, and $M_G$ is a matroid.
Greedy

Ex.: Let $M_G$ be a graphic matroid for a connected, undirected graph $G$.

Every maximal independent subset of $M_G$ must be a free tree with exactly $|V| - 1$ edges: a spanning tree of $G$.

Def.: a matroid is weighted if there is a strictly positive weight function on the edges of the matroid. It can be extended to sets of edges by summation:

$$w(A) = \sum_{x \in A} w(x)$$

Ex.: minimum spanning tree problem. We must find a subset of the edges that connects all the vertices and has minimum total length. How is this a matroid problem??

Greedy

Let $M_G$ be a weighted matroid with weight function $w'(e) = w_0 - w(e)$, where $w(e)$ is the positive weight function on the edges and $w_0$ is a positive number larger than the weight of any edge.

Each maximal independent subset $A$ corresponds to a spanning tree and

$$w'(A) = (|V| - 1)w_0 - w(A)$$

for any such set, an independent set that maximizes $w'(A)$ is one that minimizes $w(A)$.

The algorithm is on the next slide. $S[M]$ denotes the edges, $I[M]$ denotes the independent sets.

Greedy

**Greedy**($M, w$)

1. $A \leftarrow \emptyset$
2. sort $S[M]$ into nonincreasing order by weight $w$
3. for each $x \in S[M]$, taken in non-increasing order by weight $w(x)$
4. \hspace{0.5cm} do if $A \cup \{x\} \in I[M]$
5. \hspace{1cm} then $A \leftarrow A \cup \{x\}$
6. return $A$

**Running Time:** let $n = |S|$. Then sorting takes $n \lg(n)$. Line 4 is executed $n$ times, once for each element of $S$. This requires a check that $A \cup \{x\}$ is independent, for time, say $O(f(n))$. Thus the total time is $O(n \lg(n) + n f(n))$. Furthermore, $A$ is independent.

Greedy

**Lemma.** (Matroids exhibit the greedy choice property)

Suppose that $M = (S, I)$ is a weighted matroid with weight function $w$ and that $S$ is sorted into nondecreasing order by weight. Let $x$ be the first element of $S$ such that $\{x\}$ is independent, if any such $x$ exists. If $x$ exists, then there exists an optimal subset $A$ of $S$ that contains $x$.

**Proof.** If no such $x$ exists, the only independent subset is the empty set, and we are done. Otherwise, let $B$ be any nonempty optimal subset. If $x \subseteq B$, let $A = B$, and we are done. If $x \not\subseteq B$, no element of $B$ has weight greater than $x$ ( $x$ is a "heaviest" independent element and every element of $B$ is independent by the hereditary property of $I$).
### Greedy

**Start with** $A = \{x\}$. $A$ is independent because $\{x\}$ is.

Using the exchange property, repeatedly find a new element of $B$ that can be added to $A$, while preserving the independence of $A$, until $|A| = |B|$. The construction gives that $A = (B - \{y\}) \cup \{x\}$ for some $y \in B$, and so

$$w(A) = w(B) - w(y) + w(x) \geq w(B).$$

Since $B$ is optimal, $A$ must also be optimal, and since $x \in A$, the result follows.

---

### Greedy

**Lemma.** Let $M = (S, I)$ be a matroid. If $x$ is an element of $S$ such that $x$ is not an extension of $\emptyset$, then $x$ is not an extension of any independent subset $A$ of $S$.

**Proof.** The contrapositive: assume $x$ is an extension of an independent subset $A$ of $S$. Then $A \cup \{x\}$ is independent. By the hereditary property $\{x\}$ is independent, which automatically implies that it is an extension of $\emptyset$.

Another way of stating this result is that any item than cannot be used "right now", cannot be used in the future...

---

### Greedy

**Lemma.** (Matroids exhibit the optimal substructure property) Let $x$ be the first element of $S$ chosen by Greedy for the weighted matroid $M = (S, I)$. The remaining problem of finding the maximum-weight independent subset containing $x$ reduces to finding a maximum-weight independent subset of the weighted matroid $M' = (S', I')$, where

- $S' = \{y \in S : \{x, y\} \in I\}$,
- $I' = \{B \subseteq S - \{x\} : B \cup \{x\} \in I\},$

and the weight function for $M'$ is the weight function for $M$, restricted to $S'$. (We call $M'$ the *contraction* of $M$ by the element $x$)

---

### Greedy

**Theorem.** (Correctness of the greedy algorithm on matroids) If $M = (S, I)$ is a weighted matroid with weight function $w$, then the call Greedy$(M, w)$ returns an optimal subset.

**Proof.**