Greedy Algorithms

CSci 4041: Algorithms and Data Structures

Instructor: Arindam Banerjee

March 31, 2015

Greedy Algorithms

- Uses greedy choice
  - Choice that looks the best at the moment
- Greedy algorithms
  - Not always optimal
  - Can be optimal at times
  - Can give good approximations at times
  - Usually much faster than dynamic programming

Activity Selection Problem

- Set $S = \{a_1, a_2, \ldots, a_n\}$ of activities
  - Each activity a_i has start time s_i and finish time f_i
  - Activity a_i takes place in the interval $[s_i, f_i)$
- Two non-overlapping activities a_i and a_j are compatible
  - Intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap
  - Either $s_i \geq f_j$ or $s_j \geq f_i$
- Example: Activities sorted by finish times

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	9	9	10	11	12	14	16

- Maximum size subset of mutually compatible activities
  - $\{a_3, a_9, a_{11}\}$ are compatible, but not maximum
  - Two maximum subsets: $\{a_1, a_4, a_8, a_{11}\}$, $\{a_2, a_4, a_9, a_{11}\}$

Optimal Substructure of Activity Selection

- S_{ij}: Activities after a_i finishes and before a_j starts
- A_{ij}: Maximum compatible subset in S_{ij}
  - Assume A_{ij} includes activity a_k
  - Consider S_{ik}, and let $A_{ik} = A_{ij} \cap S_{ik}$
  - Consider S_{kj}, and let $A_{kj} = A_{ij} \cap S_{kj}$
  - We have $A_{ij} = A_{ik} \cap \{a_k\} \cap A_{kj}$ so that

$$|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$$

- Optimal A_{ij} includes optimal solutions to S_{ik} and S_{kj}
  - The 'cut-and-paste' argument
- For S_{kj}, if there was a better solution A'_{kj}, with $|A'_{kj}| > |A_{kj}|$
  - We could get a better solution $A'_{ij} = A_{ik} \cup \{a_k\} \cup A'_{kj}$
  - In particular, $|A'_{ij}| = |A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$
  - But that violates the optimality of A_{ij}

Optimal Substructure of Activity Selection (Contd.)

- Size of the optimal solution for S_{ij} be $c[i,j]$

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \ , \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset \ . \end{cases}$$

- One can develop a dynamic programming algorithm
- But we can take advantage of structure in the problem

Greedy Choice

- Pick the activity which leaves most time left for others
- Pick activity a_1 with earliest finish time
  - There will always be a first activity
  - f_1 is the earliest finish time among all activities
  - Leaves the most room for other activities
- Remaining problem: Finding activities that start after a_1 finishes
- Let $S_k = \{a_i \in S : s_i \geq f_k\}$
- If a_1 is in the optimal subset, we can focus only on S_1

Greedy Choice (Contd.)

- Theorem: Let a_m be an activity in S_k with earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k
- Proof by the 'cut-and-paste' style argument
  - Let A_k be an optimal solution
  - Let $a_j \in A_k$ be the activity with the earliest finish time
  - If $a_j = a_m$, then we are done
  - If $a_j \neq a_m$, we can replace a_j by a_m and still get an optimal solution
- Idea: Always pick the activity with the earliest finish time, then focus on the rest

Algorithm: Recursive-Activity-Selector

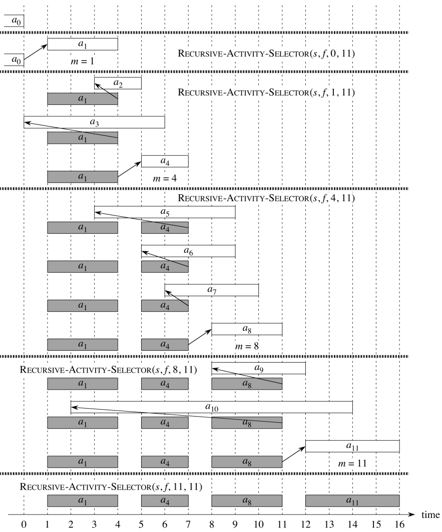
REC-ACTIVITY-SELECTOR(s, f, k, n)

$m = k + 1$
while $m \le n$ and $s[m] < f[k]$ // find the first activity in S_k to finish
    $m = m + 1$
if $m \le n$
    return $\{a_m\} \cup$ REC-ACTIVITY-SELECTOR(s, f, m, n)
else return $\emptyset$

- Initial call RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$
- Use a fictitious activity a_0 with $f_0 = 0$
- Always picks the activity with the earliest finish time among compatible activities

Example: Recursive Activity Selector

k	s_k	f_k
0	–	0
1	1	4
2	3	5
3	0	6
4	5	7
5	3	9
6	5	9
7	6	10
8	8	11
9	8	12
10	2	14
11	12	16



Algorithm: Greedy-Activity-Selector

```
GREEDY-ACTIVITY-SELECTOR(s, f)
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}
        k = m
return A
```

- k indexes the most recent addition to A

$$f_k = \max\{f_i : a_i \in A\}$$

- Considers activities with $s_m \geq f_k$
- Picks the ones with the earliest finish time
- Activities are already sorted by finish time

Elements of Greedy Strategy

- Overall structure we followed was a bit indirect
- Determine the optimal substructure of the problem
- Develop a recursive solution
- If we make the greedy choice, only one subproblem remains
- It is always safe to make the greedy choice
- Develop recursive algorithm implementing the greedy strategy
- Convert the recursive algorithm to an iterative algorithm

Elements of Greedy Strategy (Contd.)

- Simpler approach for a direct strategy
  - Optimization problem: Make a choice, left with one subproblem
  - Prove that greedy choice is always safe
  - There is always an optimal solution that makes the greedy choice
- Demonstrate optimal substructure
  - Make the greedy choice, get a subproblem
  - Combine greedy choice with optimal solution to subproblem
  - Get optimal solution to overall problem

Greedy Choice Property

- Greedy choice gives locally optimal solution
- Dynamic programming works bottom up
  - From smaller to larger problems
  - Relies on solutions to subproblems
- Greedy choice may depend on past choices
  - Does not depend on solutions to subproblems
  - Does not depend on future decisions
- Need to show that greedy choice leads to optimum
- Greedy choice is usually more efficient

Optimal Substructure

- Optimal global solution
  - Contains optimal solutions to subproblems
- Needed for dynamic programming and greedy algorithms
- Greedy algorithms use a more direct structure
  - A greedy choice results in a subproblem
  - Prove that greedy choice + optimal solution to subproblem = optimal solution to original problem

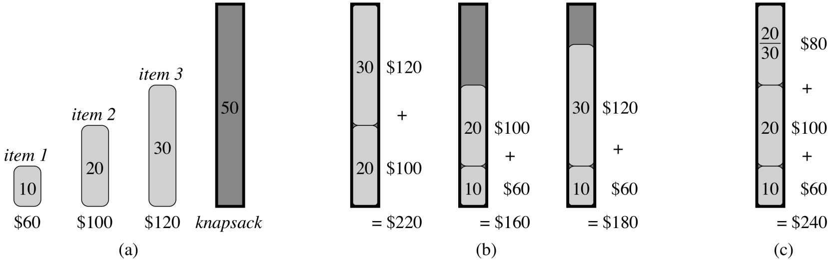
Greedy vs. Dynamic Programming

- Comparison based on knapsack problems
- 0-1 knapsack problem
  - n items, a knapsack which carry W pounds
  - Item i has value v_i and weight w_i
  - How to choose items to maximize value in knapsack?
- Fractional knapsack problems
  - Same setup as above
  - Additionally, can take fractional amount of an item
- Gold ingots/bars (0-1) vs. gold dust (fractional)

Optimal Substructure in Knapsack Problems

- 0-1 Knapsack
  - Consider most valuable itemset with at most W pounds
  - Remove item j
  - Remaining must be most valuable itemset with at most $W - w_j$ pounds
- Fractional knapsack
  - Compute value per pound v_i/w_i for each item
  - Sort items based on value per pound
  - Start filling with most valuable, then next most valuable, etc.
  - Continue till knapsack is full
- 0-1 Knapsack cannot be solved by a greedy strategy

Example: Knapsack Problems



- Item 1 is the most valuable
- For 0-1, item 1 left out, not greedy

Compression by Character Coding

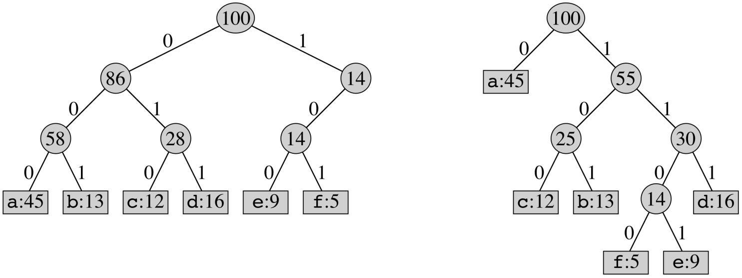
	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- Designing a binary character code
- Each character is represented as a unique binary string
- Fixed length code: 300,000 bits
- Variable length code: 224,000 bits

Prefix Codes

- No codeword is a prefix of another
- Suitable prefix code can always achieve optimal compression
- Encoding: Just concatenate the codewords
  For abc, we get $0 \cdot 101 \cdot 100 = 0101100$
- Decoding: Sequentially, identify codeword, output character
  $0010111001 = 0 \cdot 0 \cdot 101 \cdot 1101$ decodes to aabe

Example: Decoding and Binary Trees



(a) (b)

- Traverse down the tree, decode character at leaf
- Optimal code is always a full binary tree
  - Every nonleaf node has two children
- For C characters, $|C|$ leaves, $|C| - 1$ internal nodes

Huffman Coding

- T: binary tree, $d_T(c)$: length of codeword c
- Given a binary tree T, total number of bits needed

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

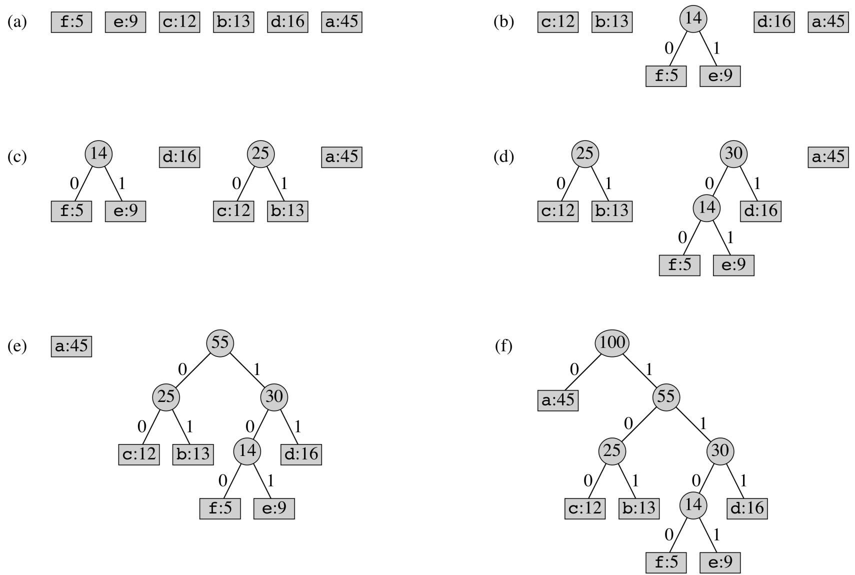
- Huffman coding: Optimal prefix code with a greedy algorithm
  - Begins with $|C|$ leaves
  - Sequential $|C| - 1$ merges to form tree

Algorithm: HUFFMAN

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)    // return the root of the tree
```

- Merge two nodes with lowest frequencies
- Create a new (meta)node after merging
- Total $O(n \log n)$
  - Build min heap in $O(n)$
  - $(n-1)$ heap operations, each $O(\log n)$

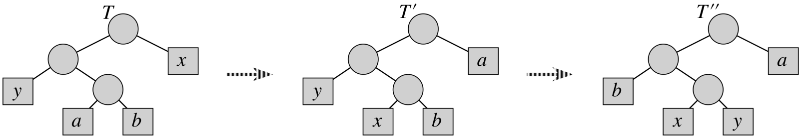
Example: Huffman Coding



Correctness of Huffman's Algorithm

- Lemma: x and y be characters in C with the lowest frequencies. There exists an optimal prefix code where x and y have the same length, and differ only in the last bit
- Let T be an optimal code
  - a and b are siblings in T at maximum depth
  - 'Wlog' $a.freq \leq b.freq, x.freq \leq y.freq$
  - We have $x.freq \leq a.freq, y.freq \leq b.freq$
  - Assume $x.freq \neq b.freq$, otherwise problem is trivial
- Create new tree by swapping
  - Swap x and a to get T'
  - Then swap y and b to get T''

Correctness of Huffman's Algorithm (Contd.)



- T' is as good as T since

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\ &= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\ &= (a.freq - x.freq)(d_T(a) - d_T(x)) \ \geq \ 0 \end{aligned}$$

- Similarly, T'' is as good as T', so $B(T'') \leq B(T)$
- But since T is optimal, $B(T'') = B(T)$
- Thus, T'' is optimal, with x, y as siblings at max depth

Correctness of Huffman's Algorithm (Contd.)

- Allows for greedy choice for merge
  - Merge the two least frequent nodes
  - Always part of an optimal solution
  - No need to consider other nodes/frequencies
- Merge x and y to get new node z
  - $z.freq = x.freq + y.freq$
  - C' be a reduced character set: $C' = C - \{x, y\} \cup \{z\}$
- Lemma: T' be an optimal prefix code tree for C', then T obtained from T' by replacing z with an internal node with x, y as children is an optimal prefix code for C

Correctness of Huffman's Algorithm (Contd.)

- By construction

  $$x.freq \cdot d_T(x) + y.freq \cdot d_T(y) = z.freq \cdot d_{T'}(z) + (x.freq + y.freq)$$

- As a result

  $$B(T') = B(T) - x.freq - y.freq$$

- Proof by contradiction: Suppose T is not optimal
  - There exists T'' such that $B(T'') < B(T)$
  - Then, T'' has x and y as siblings
  - Construct T''' by merging x, y to get node z

- Note that we are using the cut-and-paste argument

Correctness of Huffman's Algorithm (Contd.)

- By construction

$$\begin{aligned} B(T''') = & \ B(T'') - x.freq - y.freq \\ < & \ B(T) - x.freq - y.freq = B(T) \end{aligned}$$

- Contradicts the optimality of T'
- Thus, T must be optimal for C