Introduction to Algorithms

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

Greedy Algorithms

- Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step.
- For many optimization problems, using dynamic programming to determine the best choices is overkill, since, simpler, more efficient algorithms will do.
- A **greedy algorithm** always makes the choice that looks best at the moment. It makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
- The greedy method is quite powerful and works well for a wide range of problems, e.g.:
  - Minimum-spanning-tree algorithms;
  - Dijkstra's algorithm for shortest paths from a single source;
  - Chvátal's greedy set-covering heuristic.

Subsection 1

An Activity Selection Problem

The Activity Selection Problem

- Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed **activities** that wish to use a resource, able to serve only one activity at a time.
- Each activity a_i has a **start time** s_i and a **finish time** f_i, where $0 \leq s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$.
- Activities a_i and a_j are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.

  That is, a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$.
- In the **activity-selection problem**, we wish to select a maximum-size subset of mutually compatible activities.
- We assume that the activities are sorted in monotonically increasing order of finish time: $f_1 \leq f_2 \leq f_3 \leq \cdots \leq f_{n-1} \leq f_n$.

Example

- Consider the following set S of activities:

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

- For this example, the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities.
- It is not a maximum subset, since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger.
- In fact, $\{a_1, a_4, a_8, a_{11}\}$ is a largest subset of mutually compatible activities.
- Another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.

From Dynamic Programming to Greedy Algorithms

- We shall solve this problem in several steps:
  - We start by thinking about a dynamic-programming solution, in which we consider several choices when determining which subproblems to use in an optimal solution.
  - We shall then observe that we need to consider only one choice, the greedy choice, and that when we make the greedy choice, only one subproblem remains.
  - Based on these observations, we shall develop a recursive greedy algorithm to solve the activity-scheduling problem.
  - We shall complete the process by converting the recursive algorithm to an iterative one.
- We develop a greedy algorithm while emphasizing the relationship between greedy algorithms and dynamic programming.

The Optimal Substructure of Activity Selection

- We denote by S_{ij} the set of activities that start after activity a_i finishes and that finish before activity a_j starts.
- Suppose that we wish to find a maximum set of mutually compatible activities in S_{ij}.
- If such a maximum set A_{ij} includes some activity a_k, we are left with two subproblems:
  - Finding mutually compatible activities in the set S_{ik} (activities that start after activity a_i finishes and that finish before activity a_k starts);
  - Finding mutually compatible activities in the set S_{kj} (activities that start after activity a_k finishes and that finish before activity a_j starts).
- Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$:
  - A_{ik} contains the activities in A_{ij} that finish before a_k starts;
  - A_{kj} contains the activities in A_{ij} that start after a_k finishes.
- Thus, we have $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$.

  So the maximum-size set A_{ij} of mutually compatible activities in S_{ij} consists of $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ activities.

Optimality

- Cut-and-paste shows that the optimal solution A_{ij} must also include optimal solutions to the two subproblems for S_{ik} and S_{kj}.

  If we could find a set A'_{kj} of mutually compatible activities in S_{kj} where $|A'_{kj}| > |A_{kj}|$, then we could use A'_{kj}, rather than A_{kj}, in a solution to the subproblem for S_{ij}.

  We would have constructed a set of

  $$|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$$

  mutually compatible activities, which contradicts the optimality assumption on A_{ij}.

  A symmetric argument applies to the activities in S_{ik}.

A Recursive Formula

- If we denote the size of an optimal solution for the set S_{ij} by $c[i,j]$, then we would have the recurrence

$$c[i,j] = c[i,k] + c[k,j] + 1.$$

- But we do not know that an optimal solution for the set S_{ij} includes activity a_k.
- So we have to examine all activities in S_{ij} to find which one to choose.

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

- We could then develop a recursive algorithm and memoize it, or we could work bottom-up and fill in table entries as we proceed.

Making the Greedy Choice

- Intuition tells us to choose the activity a_1 in S with the earliest finish time so as to leave the resource available for as many other activities as possible.
- If we make the greedy choice, we have only one remaining subproblem to solve: Finding activities that start after a_1 finishes.
  - We do not have to consider activities that finish before a_1 starts, since $s_1 < f_1$, and f_1 is the earliest finish time of any activity, so no activity can have a finish time less than or equal to s_1.
  - Furthermore, we have already established that the activity-selection problem exhibits optimal substructure.
- Let $S_k = \{a_i \in S : s_i \geq f_k\}$ be the set of activities that start after activity a_k finishes. If we make the greedy choice of activity a_1, then S_1 remains as the only subproblem to solve.

  Optimal substructure tells us that if a_1 is in the optimal solution, then an optimal solution to the original problem consists of activity a_1 and all the activities in an optimal solution to the subproblem S_1.

Justification of the Greedy Choice

- Is our intuition correct, i.e., is the greedy choice always part of some optimal solution?

Theorem

Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum size subset of mutually compatible activities of S_k.

- Let A_k be a maximum-size subset of mutually compatible activities in S_k. Let a_j be the activity in A_k with the earliest finish time.
  - If $a_j = a_m$, we are done, since we have shown that a_m is in some maximum-size subset of mutually compatible activities of S_k.
  - If $a_j \neq a_m$, consider the set $A'_k = (A_k - \{a_j\}) \cup \{a_m\}$.
    The activities in A'_k are disjoint, which follows because the activities in A_k are disjoint, a_j is the first activity in A_k to finish, and $f_m \leq f_j$.
    Since $|A'_k| = |A_k|$, A'_k is a maximum-size subset of mutually compatible activities of S_k and it includes a_m.

Top-Down design vs. Dynamic Programming

- Instead of solving the activity-selection problem with dynamic programming, we can:
  - Repeatedly choose the activity that finishes first;
  - Keep only the activities compatible with this activity;
  - Repeat until no activities remain.
- Since we always choose the activity with the earliest finish time, the finish times of the activities we choose must strictly increase.
- An algorithm to solve the activity selection problem does not need to work bottom-up, like a table based dynamic programming algorithm.
- It can work top-down, choosing an activity to put into the optimal solution and then solving the subproblem of choosing activities from those that are compatible with those already chosen.
- Greedy algorithms typically have this top-down design: Make a choice and then solve a subproblem.

A Recursive Greedy Algorithm

- The procedure RECURSIVEACTIVITYSELECTOR takes as input:
    - The arrays s and f with the start and finish times;
    - The index k defining the subproblem S_k to solve;
    - The size n of the original problem.
- It returns a maximum size set of mutually compatible activities in S_k.
- We assume that the n input activities are already ordered by monotonically increasing finish time.

  If this is not the case, we can sort them into this order in $O(n \log n)$ time, breaking ties arbitrarily.

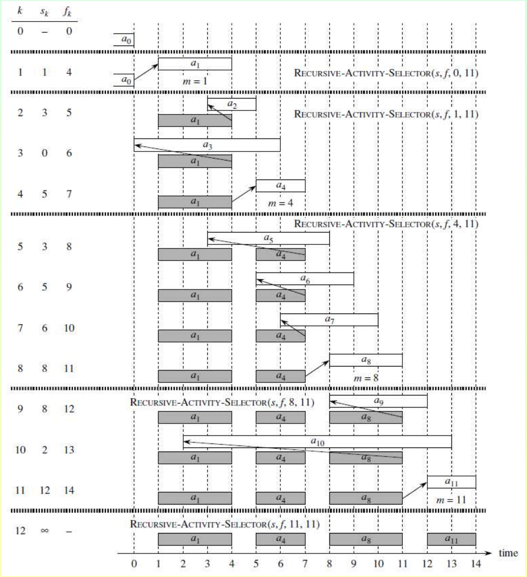
A Recursive Greedy Algorithm (Cont'd)

- To start, we add the fictitious activity a_0 with $f_0 = 0$.
- To solve the entire problem call $\textsc{RecursiveActivitySelector}(s, f, 0, n)$.

$\textsc{RecursiveActivitySelector}(s, f, k, n)$

```
1. $m = k + 1$
2. while $m \leq n$ and $s[m] < f[k]$ //find the first activity in $S_k$ to finish
3.     $m = m + 1$
4. if $m \leq n$
5.     return $\{a_m\} \cup \textsc{RecursiveActivitySelector}(s, f, m, n)$
6. else return $\emptyset$
```

How RECURSIVEACTIVITYSELECTOR Works



k	s_k	f_k
0	–	0
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14
12	∞	–

RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 11)
RECURSIVE-ACTIVITY-SELECTOR(s, f, 1, 11)
RECURSIVE-ACTIVITY-SELECTOR(s, f, 4, 11)
RECURSIVE-ACTIVITY-SELECTOR(s, f, 8, 11)
RECURSIVE-ACTIVITY-SELECTOR(s, f, 11, 11)
m = 1
m = 4
m = 8
m = 11
time

In a given recursive call, the while loop of Lines 2-3 looks for the first activity in S_k to finish. The loop examines a_{k+1}, a_{k+2}, ..., a_n, until it finds the first activity a_m that is compatible with a_k, such an activity has $s_m \geq f_k$. If the loop terminates because it finds such an activity, Line 5 returns the union of $\{a_m\}$ and the maximum size subset of S_m returned by a recursive call.

Time Requirements

- If the activities have already been sorted by finish times, the running time of the call $\textsc{RecursiveActivitySelector}(s, f, 0, n)$ is $\Theta(n)$, which we can see as follows:

  Over all recursive calls, each activity is examined exactly once in the while loop test of Line 2.

  In particular, activity a_i is examined in the last call made in which $k < i$.

An Iterative Greedy Algorithm

- RECURSIVEACTIVITYSELECTOR is almost "tail recursive": It ends with a recursive call to itself followed by a union operation.
- As written, RECURSIVEACTIVITYSELECTOR works for subproblems S_k, i.e., subproblems that consist of the last activities to finish.
- The procedure GREEDYACTIVITYSELECTOR is an iterative version of the procedure RECURSIVEACTIVITYSELECTOR.
- It assumes that the input activities are ordered by finish time.

GREEDYACTIVITYSELECTOR(s, f)

```
1. $n = s$.length
2. $A = \{a_1\}$
3. $k = 1$
4. for $m = 2$ to $n$
5.    if $s[m] \geq f[k]$
6.       $A = A \cup \{a_m\}$
7.       $k = m$
8. return $A$
```

How GreedyActivitySelector Works

- The variable k indexes the most recent addition to A, corresponding to the activity a_k in the recursive version.
- Since activities are in order of increasing finish time, f_k is always the maximum finish time of any activity in A, i.e., $f_k = \max\{f_i : a_i \in A\}$.
- Lines 2-3 select activity a_1, initialize A to contain just this activity, and initialize k to index this activity.
- The for loop of Lines 4-7 finds the earliest activity in S_k to finish.

  The loop considers each activity a_m in turn and adds a_m to A if it is compatible with all previously selected activities.

  If activity a_m is compatible, then Lines 6-7 add activity a_m to A and set k to m.
- Like the recursive version, GreedyActivitySelector schedules a set of n activities in $\Theta(n)$ time, assuming that the activities were already sorted initially by their finish times.

Subsection 2

Elements of the Greedy Strategy

Steps to Develop a Greedy Strategy

- A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices.
- At each decision point, the algorithm makes the choice that seems best at the moment.
- This heuristic strategy does not always produce an optimal solution, but sometimes it does.
- To develop a greedy algorithm, we went through the following steps:
  1. Determine the optimal substructure of the problem.
  2. Develop a recursive solution.
  3. Show that if we make the greedy choice, then only one subproblem remains.
  4. Prove that it is always safe to make the greedy choice.
  5. Develop a recursive algorithm that implements the greedy strategy.
  6. Convert the recursive algorithm to an iterative algorithm.
- In going through these steps, we saw in great detail the dynamic programming underpinnings of a greedy algorithm.

A More Direct Approach

- We may also start with an optimal substructure having a greedy choice in mind, so that the choice leaves just one subproblem to solve.
- In the activity selection problem, we could have started by dropping the second subscript and defining subproblems of the form S_k.

  Then, we could have proven that a greedy choice (the first activity a_m to finish in S_k), combined with an optimal solution to the remaining set S_m of compatible activities, yields an optimal solution to S_k.
- Thus, we design greedy algorithms according to the following:
  1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
  2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
  3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution.

Greedy Choice Property

- In dynamic programming, we make a choice at each step, but the choice usually depends on the solutions to subproblems.

  Typically dynamic programming problems are solved in a bottom-up manner, progressing from smaller subproblems to larger subproblems.
- In a greedy algorithm, we make whatever choice seems best at the moment and then solve the subproblem that remains.

  The choice made may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems.
- We must prove that a greedy choice at each step yields a globally optimal solution: The proof examines a globally optimal solution to a subproblem and shows how to modify the solution to substitute the greedy choice for some other choice, resulting in a smaller subproblem.
- We can usually make the greedy choice more efficiently than when we have to consider a wider set of choices.

Optimal substructure

- A problem exhibits **optimal substructure** if an optimal solution to the problem contains within it optimal solutions to subproblems.
- For greedy algorithms, all we need to do is argue that an optimal solution to the subproblem, combined with the greedy choice already made, yields an optimal solution to the original problem.
- This scheme implicitly uses induction on the subproblems to prove that making the greedy choice produces an optimal solution.

0-1 Knapsack and Fractional Knapsack Problems

- **0-1 Knapsack Problem**:

  A thief robbing a store finds n items.
  The i-th item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers.
  The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W.
  Which items should he take?

- **Fractional Knapsack Problem**:

  The setup is the same.
  However, the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item.
  Which fraction of each item should he take?

Knapsack Problems and Optimal Substructure

- Both knapsack problems exhibit the optimal-substructure property:
  - For the 0-1 problem, consider the most valuable load that weighs at most W pounds.
    If we remove item j from this load, the remaining load must be the most valuable load weighing at most $W - w_j$ that the thief can take from the $n - 1$ original items excluding j.
  - For the comparable fractional problem, consider that if we remove a weight w of one item j from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ that the thief can take from the $n - 1$ original items plus $w_j - w$ pounds of item j.

Applicability of Greedy Strategy for Fractional Knapsack

- We can solve the fractional knapsack problem by a greedy strategy.
- We first compute the value per pound $\frac{v_i}{w_i}$ for each item.
- Obeying a greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound.
- If the supply is exhausted and he can still carry more, he takes as much as possible of the item with the next greatest value per pound.
- He continues in the same way, until he reaches his weight limit W.
- By sorting the items by value per pound, the greedy algorithm runs in $O(n \log n)$ time.

Non-Applicability of Greedy Strategy for 0-1 Knapsack

- We cannot solve the 0-1 problem by such a strategy.
- Consider 3 items and a knapsack that can hold 50 pounds.
  - Item 1 weighs 10 pounds and is worth 60 dollars.
  - Item 2 weighs 20 pounds and is worth 100 dollars.
  - Item 3 weighs 30 pounds and is worth 120 dollars.

  The values per pound in decreasing order are:
  - Item 1, 6 dollars per pound;
  - Item 2, 5 dollars per pound;
  - Item 3, 4 dollars per pound.

  The greedy strategy, therefore, would take item 1 first.

  However, the optimal solution takes items 2 and 3, leaving item 1 behind.

Subsection 3

Huffman Codes

Data Compression

- Huffman codes compress data:
  - The data is a sequence of characters.
  - Huffman's greedy algorithm uses a table giving how often each character occurs (its frequency) to build up an optimal way of representing each character as a binary string.
- We consider the problem of designing a binary character code (or code for short) in which each character is represented by a unique binary string, which we call a **codeword**.
- If we use a **fixed length code**, we need, e.g., 3 bits to represent 6 characters. Thus, a 100,00 character file consisting of 6 character words requires 300,000 bits.
- A **variable length code** can do considerably better than a fixed length code, by giving frequent characters short codewords and infrequent characters long codewords.

Prefix Codes

- We consider only codes in which no codeword is also a prefix of some other codeword, called **prefix codes**.
- A prefix code can always achieve the optimal data compression among any character code, and so we suffer no loss of generality by restricting our attention to prefix codes.
- Encoding is always simple for any binary character code. We just concatenate the codewords representing each character of the file.
- Prefix codes simplify decoding. Since no codeword is a prefix of any other, the initial codeword in a file is unambiguous.
  - We identify the initial codeword;
  - We translate it back to the original character;
  - We repeat the decoding process on the remainder of the encoded file.
- The decoding process needs a convenient representation for the prefix code so that the initial codeword can be easily picked off.

  We can use a binary tree whose leaves are the given characters.

Binary Tree Representation of a Prefix Code

- We interpret the binary codeword for a character as the path from the root to that character, where 0 means "go to the left child" and 1 means "go to the right child".
- An optimal code for a file is always represented by a full binary tree, in which every nonleaf node has two children.
- If C is the alphabet from which the characters are drawn and all character frequencies are positive, then the tree for an optimal prefix code has exactly $|C|$ leaves and $|C| - 1$ internal nodes.

Number of Bits Encoding a File

- Given a tree T corresponding to a prefix code, we compute the number of bits required to encode a file.
- For each character c in the alphabet C, let:
  - $c.\text{freq}$ denote the frequency of c in the file;
  - $d_T(c)$ denote the depth of c's leaf in the tree.
    $d_T(c)$ is also the length of the codeword for character c.
- The number of bits required to encode a file is thus

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

- We define $B(T)$ as the **cost** of the tree T.

Constructing a Huffman Code

- Huffman invented a greedy algorithm that constructs an optimal prefix code called a **Huffman code**.
- Its proof of correctness relies on the greedy-choice property and optimal substructure.
- In the pseudocode that follows, we assume that C is a set of n characters and that each character $c \in C$ is an object with an attribute c.freq giving its frequency.
- The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner.
    - It begins with a set of $|C|$ leaves and performs a sequence of $|C| - 1$ "merging" operations to create the final tree.
    - The algorithm uses a min-priority queue Q, keyed on the freq attribute, to identify the two least-frequent objects to merge together.
    - When we merge two objects, the result is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

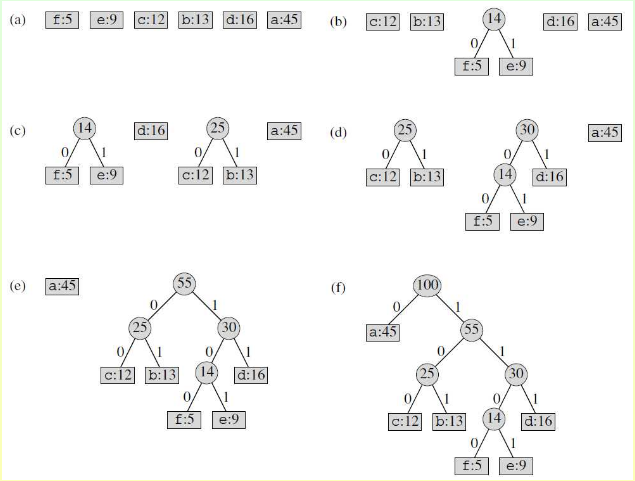
The Huffman Procedure

HUFFMAN(C)

```
1. $n = |C|$
2. $Q = C$
3. for $i = 1$ to $n - 1$
4.     allocate a new node $z$
5.     $z$.left $= x =$ EXTRACTMIN($Q$)
6.     $z$.right $= y =$ EXTRACTMIN($Q$)
7.     $z$.freq $= x$.freq $+ y$.freq
8.     INSERT($Q, z$)
9. return EXTRACTMIN($Q$)
```

Illustration of HUFFMAN

Symbol	a	b	c	d	e	f
Frequency	45	13	12	16	9	5



How Huffman Works

- Line 2 initializes the min-priority queue Q with the characters in C.
- The for loop in Lines 3-8 repeatedly extracts the two nodes x and y of lowest frequency from the queue, and replaces them in the queue with a new node z representing their merger.

  The frequency of z is computed as the sum of the frequencies of x and y in Line 7.

  The node z has x as its left child and y as its right child.

  (This order is arbitrary; switching the left and right child of any node yields a different code of the same cost.)
- After $n-1$ mergers, the one node left in the queue, the root of the code tree, is returned in Line 9.

Time Requirements of the Huffman Procedure

- To analyze the running time of Huffman's algorithm, we assume that Q is implemented as a binary min-heap.
    - For a set C of n characters, we can initialize Q in Line 2 in $O(n)$ time using the BuildMinHeap procedure.
    - The for loop in Lines 3-8 executes exactly $n-1$ times.
      Each heap operation requires time $O(\log n)$.
      So the loop contributes $O(n \log n)$ to the running time.

  Thus, the total running time of Huffman on a set of n characters is $O(n \log n)$.
- We note that can reduce the running time to $O(n \log \log n)$ by replacing the binary min-heap with a van Emde Boas tree.

Correctness of Huffman's Algorithm: Greedy Choice

Lemma

Let C be an alphabet in which each character $c \in C$ has frequency c.freq. Let x and y be two characters in C having 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 the last bit.

- The idea is to take the tree T representing an arbitrary optimal prefix code and modify it to make a tree representing another optimal prefix code such that x and y appear as sibling leaves of maximum depth in the new tree. Then the codewords for x and y will have the same length and differ only in the last bit.

  Let a and b be two characters that are sibling leaves of maximum depth in T. Without loss of generality, we assume that a.freq $\leq b$.freq and x.freq $\leq y$.freq. Since x.freq and y.freq are the two lowest leaf frequencies, in order, and a.freq and b.freq are two arbitrary frequencies, in order, we have x.freq $\leq a$.freq and y.freq $\leq b$.freq.

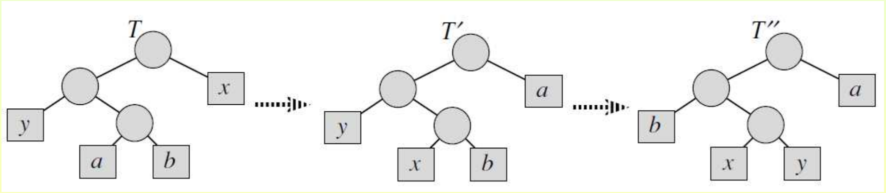
Correctness: Greedy Choice (Cont'd)

- In the remainder of the proof, it is possible that we could have x.freq $=$ a.freq or y.freq $=$ b.freq. However, if we had x.freq $=$ b.freq, then we would have a.freq $=$ b.freq $=$ x.freq $=$ y.freq, and the lemma would be trivially true. Thus, we assume x.freq $\neq$ b.freq. So $x \neq b$.

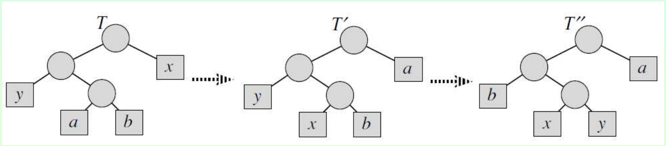
  We exchange the positions in T of a and x to produce a tree T'.

  Then we exchange the positions in T' of b and y to produce T''.

  In T'', x and y and sibling leaves of maximum depth.



Correctness: Greedy Choice (Conclusion)



- The difference in cost between T and T' is $B(T) - B(T')$

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

  because both $a.\mathrm{freq} - x.\mathrm{freq}$ and $d_T(a) - d_T(x)$ are nonnegative.
  Similarly, exchanging y and b does not increase the cost, and, so, $B(T') - B(T'')$ is nonnegative. Therefore, $B(T'') \leq B(T)$. Since T is optimal, $B(T) \leq B(T'')$. This implies $B(T'') = B(T)$.
  Thus, T'' is an optimal tree in which x and y appear as sibling leaves of maximum depth.

Correctness of Huffman's Algorithm: Optimal Substructure

Lemma

Let C be a given alphabet with frequency $c.\text{freq}$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with the characters x and y removed and a new character z added, so that $C' = (C - \{x, y\}) \cup \{z\}$. Define freq for C' as for C, except that $z.\text{freq} = x.\text{freq} + y.\text{freq}$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

- We express the cost $B(T)$ of tree T in terms of the cost $B(T')$ of tree T': For each $c \in C - \{x, y\}$, we have that $d_T(c) = d_{T'}(c)$, and, hence, $c.\text{freq} d_T(c) = c.\text{freq} d_{T'}(c)$. Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have $x.\text{freq} d_T(x) + y.\text{freq} d_T(y) = (x.\text{freq} + y.\text{freq}) d_T(x) = (x.\text{freq} + y.\text{freq})(d_{T'}(z) + 1) = z.\text{freq} d_{T'}(z) + (x.\text{freq} + y.\text{freq})$. So $B(T) = B(T') + x.\text{freq} + y.\text{freq}$ or $B(T') = B(T) - x.\text{freq} - y.\text{freq}$.

Proof of Optimal Substructure (Cont'd)

- We now prove the lemma by contradiction.
  Suppose that T does not represent an optimal prefix code for C. Then there exists an optimal tree T'', such that $B(T'') < B(T)$. Without loss of generality, by the preceding lemma, T'' has x and y as siblings. Let T''' be the tree T'' with the common parent of x and y replaced by a leaf z with frequency $z.\text{freq} = x.\text{freq} + y.\text{freq}$. Then

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

  This contradicts the assumption that T' represents an optimal prefix code for C'. Thus, T must represent an optimal prefix code for the alphabet C.

Theorem

Procedure HUFFMAN produces an optimal prefix code.

Subsection 4

Matroids and Greedy Methods

Matroids

- A **matroid** is an ordered pair $M = (S, \mathcal{I})$ satisfying the following conditions:
  1. S is a finite set.
  2. $\mathcal{I}$ is a nonempty family of subsets of S, called the **independent subsets** of S, such that if $B \in \mathcal{I}$ and $A \subseteq B$, then $A \in \mathcal{I}$.
     We say that $\mathcal{I}$ is **hereditary** if it satisfies this property.
     Note that the empty set $\emptyset$ is necessarily a member of $\mathcal{I}$.
  3. If $A \in \mathcal{I}$, $B \in \mathcal{I}$, and $|A| < |B|$, then there exists some element $x \in B - A$, such that $A \cup \{x\} \in \mathcal{I}$.
     We say that M satisfies the **exchange property**.

Matric and Graphic Matroids

- The word “matroid” is due to Hassler Whitney.
- He was studying **matric matroids**, in which the elements of S are the rows of a given matrix and a set of rows is independent if they are linearly independent in the usual sense.

  This structure defines a matroid.
- As another example of matroids, consider the **graphic matroid** $M_G = (S_G, \mathcal{I}_G)$, defined in terms of a given undirected graph $G = (V, E)$ as follows:
  - The set S_G is defined to be E, the set of edges of G.
  - If A is a subset of E, then $A \in \mathcal{I}_G$ if and only if A is acyclic. That is, a set of edges A is independent if and only if the subgraph $G_A = (V, A)$ forms a forest.

Graphic Matroids are Matroids

Theorem

If $G = (V, E)$ is an undirected graph, then $M_G = (S_G, \mathcal{I}_G)$ is a matroid.

- Clearly, $S_G = E$ is a finite set. Furthermore, $\mathcal{I}_G$ is hereditary, since removing edges from an acyclic set of edges cannot create cycles.
  It remains to show that M_G satisfies the exchange property.
  Suppose that $G_A = (V, A)$ and $G_B = (V, B)$ are forests of G and that $|B| > |A|$. That is, A and B are acyclic sets of edges, and B contains more edges than A does.
  Claim: A forest $F = (V_F, E_F)$ contains exactly $|V_F| - |E_F|$ trees.
  To see why, suppose that F consists of t trees, where the i-th tree contains v_i vertices and e_i edges. Then, we have

$$|E_F| = \sum_{i=1}^{t} e_i = \sum_{i=1}^{t} (v_i - 1) = \sum_{i=1}^{t} v_i - t = |V_F| - t.$$

  This implies that $t = |V_F| - |E_F|$.

The Exchange Property of Graphic Matroids

- Forest G_A contains $|V| - |A|$ trees, and G_B contains $|V| - |B|$ trees.

  Thus, forest G_B has fewer trees than forest G_A.

  So forest G_B must contain some tree T whose vertices are in two different trees in forest G_A.

  Moreover, since T is connected, it must contain an edge (u, v) such that vertices u and v are in different trees in forest G_A.

  Since the edge (u, v) connects vertices in two different trees in forest G_A, we can add the edge (u, v) to forest G_A without creating a cycle.

  Therefore, M_G satisfies the exchange property, completing the proof that M_G is a matroid.

Extensions

- Given a matroid $M = (S, \mathcal{I})$, we call an element $x \notin A$ an **extension** of $A \in \mathcal{I}$ if we can add x to A while preserving independence,

  i.e., x is an extension of $A \in \mathcal{I}$ if $A \cup \{x\} \in \mathcal{I}$.

  Example: Consider a graphic matroid $M_G = (S_G, \mathcal{I}_G)$.

  If A is an independent set of edges, then edge e is an extension of A if and only if e is not in A and the addition of e to A does not create a cycle.

Maximal Independent Sets

- If A is an independent subset in a matroid M, we say that A is **maximal** if it has no extensions, i.e., A is maximal if it is not contained in any larger independent subset of M.

Theorem

All maximal independent subsets in a matroid have the same size.

- Suppose A is a maximal independent subset of M and there exists another larger maximal independent subset B of M. The exchange property implies that, for some $x \in B - A$, we can extend A to a larger independent set $A \cup \{x\}$, contradicting the maximality of A.

Weighted Matroids

Example: Consider a graphic matroid $M_G = (S_G, \mathcal{I}_G)$ for a connected, undirected graph G.

Every maximal independent subset of M_G must be a free tree with exactly $|V| - 1$ edges that connects all the vertices of G.

Such a tree is called a **spanning tree** of G.

- We say that a matroid $M = (S, \mathcal{I})$ is **weighted** if it is associated with a weight function w that assigns a strictly positive weight $w(x)$ to each element $x \in S$.
- The weight function w extends to subsets of S by summation:

$$w(A) = \sum_{x \in A} w(x), \text{ for any } A \subseteq S.$$

Example: Suppose $w(e)$ denote the weight of an edge e in a graphic matroid M_G. Then $w(A)$ is the total weight of the edges in set A.

Greedy Algorithms on Weighted Matroids

- Many problems for which a greedy approach provides optimal solutions can be formulated in terms of finding a **maximum weight independent subset in a weighted matroid**:

  Given a weighted matroid $M = (S, \mathcal{I})$, find an independent set $A \in \mathcal{I}$, such that $w(A)$ is maximized.
- In a weighted matroid, a subset that is independent and has maximum possible weight is called an **optimal subset**.
- Because the weight $w(x)$ of any element $x \in S$ is positive, an optimal subset is always a maximal independent subset.

Minimum Spanning Tree

- The **minimum spanning tree problem**:

  Given a connected undirected graph $G = (V, E)$ and a length function w, such that $w(e)$ is the (positive) length of edge e, find a subset of the edges that connects all the vertices and has minimum total length.

- Any algorithm that can find an optimal subset A in an arbitrary matroid can solve the minimum-spanning-tree problem.

  Consider the weighted matroid M_G with weight function w', where $w'(e) = w_0 - w(e)$ and w_0 is larger than the max length of any edge.

  In the weighted matroid, all weights are positive and an optimal subset is a spanning tree of minimum total length in the graph:

  Each maximal independent subset A corresponds to a spanning tree with $|V| - 1$ edges. We have $w'(A) = \sum_{e \in A} w'(e) = \sum_{e \in A}(w_0 - w(e)) = (|V| - 1)w_0 - \sum_{e \in A} w(e) = (|V| - 1)w_0 - w(A)$.

  So, for any maximal independent subset A, an independent subset that maximizes the quantity $w'(A)$ must minimize $w(A)$.

A Greedy Matroid Procedure

- We give a greedy algorithm that works for any weighted matroid.
- It takes as input a weighted matroid $M = (S, \mathcal{I})$, with an associated positive weight function w, and returns an optimal subset A.
- $M.S$, $M.\mathcal{I}$ denote the components and w the weight function.
- The algorithm is greedy because it considers in turn each element $x \in S$, in order of monotonically decreasing weight, and immediately adds it to the set A being accumulated if $A \cup \{x\}$ is independent.

```
GREEDY($M, w$)
1. $A = \emptyset$
2. sort $M.S$ into monotonically decreasing order by weight $w$
3. for each $x \in M.S$, taken in monotonically decreasing order by weight $w(x)$
4.     if $A \cup \{x\} \in M.\mathcal{I}$
5.         $A = A \cup \{x\}$
6. return $A$
```

Operation, Correctness and Running Time

- Line 4 checks whether adding each element x to A would maintain A as an independent set.
  - If A would remain independent, then Line 5 adds x to A.
  - Otherwise, x is discarded.
- Since the empty set is independent, and since each iteration of the for loop maintains A's independence, the subset A is always independent, by induction.
- Therefore, $\textsc{Greedy}$ always returns an independent subset A.
- We will also show that A is a subset of maximum possible weight.
- The running time of $\textsc{Greedy}$ is easy to analyze: Let n denote $|S|$. The sorting phase of $\textsc{Greedy}$ takes time $O(n \log n)$. Line 4 executes exactly n times, once for each element of S. Each execution of Line 4 requires a check on whether or not the set $A \cup \{x\}$ is independent. If each such check takes time $O(f(n))$, the entire algorithm runs in time $O(n \log n + nf(n))$.

Correctness: Greedy Returns an Optimal Subset

Lemma (Matroids Exhibit the Greedy-Choice Property)

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

- If no such x exists, then the only independent subset is the empty set and the lemma is vacuously true.

  Otherwise, let B be any nonempty optimal subset. Assume that $x \notin B$; otherwise, letting $A = B$ gives an optimal subset of S that contains x. No element of B has weight greater than $w(x)$. To see why, observe that $y \in B$ implies that $\{y\}$ is independent, since $B \in \mathcal{I}$ and $\mathcal{I}$ is hereditary. Our choice of x therefore ensures that $w(x) \geq w(y)$, for any $y \in B$.

GREEDY Returns an Optimal Subset: Construction of A

- Construct the set A as follows:
  - Begin with $A = \{x\}$. By the choice of x, set A is independent.
  - Using the exchange property, repeatedly find a new element of B that we can add to A until $|A| = |B|$, while preserving the independence of A.
  - At that point, A and B are the same except that A has x and B has some other element y.

We have, $A = (B - \{y\}) \cup \{x\}$, for some $y \in B$. So

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

Because set B is optimal, set A, which contains x, must also be optimal.

No Skipped Element is an Option Later

- If an element is not an option initially, then it cannot be an option later.

Lemma

Let $M = (S, \mathcal{I})$ be any matroid. If x is an element of S that is an extension of some independent subset A of S, then x is also an extension of $\emptyset$.

- Since x is an extension of A, we have that $A \cup \{x\}$ is independent. Since $\mathcal{I}$ is hereditary, $\{x\}$ must be independent. Thus, x is an extension of $\emptyset$.

Corollary

Let $M = (S, \mathcal{I})$ be any 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.

- The corollary is the contrapositive of the lemma.

Matroids Exhibit the Optimal-Substructure Property

Lemma (Matroids Exhibit the Optimal-Substructure Property)

Let x be the first element of S chosen by GREEDY for the weighted matroid $M = (S, \mathcal{I})$. The remaining problem of finding a maximum-weight independent subset containing x reduces to finding a maximum-weight independent subset of the weighted matroid $M' = (S', \mathcal{I}')$, where:

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

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**.

- If A is any maximum-weight independent subset of M containing x, then $A' = A - \{x\}$ is an independent subset of M'.
  Conversely, any independent subset A' of M' yields an independent subset $A = A' \cup \{x\}$ of M. In both cases $w(A) = w(A') + w(x)$.
  So a maximum-weight solution in M containing x yields a maximum-weight solution in M', and vice versa.

Correctness of the Greedy Algorithm on Matroids

Theorem (Correctness of the Greedy Algorithm on Matroids)

If $M = (S, \mathcal{I})$ is a weighted matroid with weight function w, then $\textsc{Greedy}(M, w)$ returns an optimal subset.

- By the corollary, any elements that $\textsc{Greedy}$ passes over initially because they are not extensions of $\emptyset$ can never be useful.

  Once $\textsc{Greedy}$ selects the first element x, the lemma implies that the algorithm does not err by adding x to A, since there exists an optimal subset containing x.

  Finally, the lemma implies that the remaining problem is one of finding an optimal subset in the matroid M' that is the contraction of M by x. After the procedure $\textsc{Greedy}$ sets A to $\{x\}$, all remaining steps act in the matroid $M' = (S', \mathcal{I}')$, because B is independent in M' if and only if $B \cup \{x\}$ is independent in M, for all sets $B \in \mathcal{I}'$.

  Thus, the subsequent operation will find an optimal subset for M'.

Subsection 5

A Task Scheduling Problem as a Matroid

Scheduling Unit-Time Tasks with Deadlines and Penalties

- A **unit-time task** is a job, such as a program to be run on a computer, that requires exactly one unit of time to complete.
- Given a finite set S of unit-time tasks, a **schedule** for S is a permutation of S specifying the order in which to perform these tasks.
  - The first task in the schedule begins at time 0 and finishes at time 1;
  - The second task begins at time 1 and finishes at time 2, and so on.
- The problem of **scheduling unit-time tasks with deadlines and penalties for a single processor** has the following inputs:
  - A set $S = \{a_1, a_2, \ldots, a_n\}$ of n unit-time tasks;
  - A set of n integer **deadlines** $d_1, d_2, \ldots, d_n$, such that each d_i satisfies $1 \leq d_i \leq n$, and task a_i is supposed to finish by time d_i;
  - A set of n nonnegative **weights** or **penalties** $w_1, w_2, \ldots, w_n$, such that we incur a penalty of w_i if task a_i is not finished by time d_i, and we incur no penalty if a task finishes by its deadline.

We wish to find a schedule for S that minimizes the total penalty incurred for missed deadlines.

Early-First and Canonical Forms

- Given a schedule, we say that a task is **late** in this schedule if it finishes after its deadline. Otherwise, the task is **early** in the schedule.
- We can always transform an arbitrary schedule into **early-first form**, in which the early tasks precede the late tasks.

  To see why, note that if some early task a_i follows some late task a_j, then we can switch the positions of a_i and a_j, and a_i will still be early and a_j will still be late.
- We can always transform an arbitrary schedule into **canonical form**, in which the early tasks precede the late tasks and we schedule the early tasks in order of monotonically increasing deadlines:
  - Put the schedule into early-first form;
  - As long as there exist two early tasks a_i and a_j finishing at respective times k and $k+1$ in the schedule such that $d_j < d_i$, we swap the positions of a_i and a_j.

Early-First and Canonical Forms (Cont'd)

- Since a_j is early before the swap, $k+1 \leq d_j$. Therefore, $k+1 < d_i$, and so a_i is still early after the swap.
- Because task a_j is moved earlier in the schedule, it remains early after the swap.

- The search for an optimal schedule, thus, reduces to finding a set A of tasks that we assign to be early in the optimal schedule.
- Having determined A, we can create the actual schedule by:
  - Listing the elements of A in order of monotonically increasing deadlines;
  - Then listing the late tasks (i.e., $S - A$) in any order;

  thus, producing a canonical ordering of the optimal schedule.

Independence of Tasks

- We say that a set A of tasks is **independent** if there exists a schedule for these tasks such that no tasks are late.
- Clearly, the set of early tasks for a schedule forms an independent set of tasks.
- Let $\mathcal{I}$ denote the set of all independent sets of tasks.
- We would like to characterize independence in order to be able to determine whether a given set A of tasks is independent.
- For $t = 0, 1, \ldots, n$, let $N_t(A)$ denote the number of tasks in A whose deadline is t or earlier.
- Note that $N_0(A) = 0$, for any set A.

Characterization of Independence

Lemma

For any set of tasks A, the following statements are equivalent:

1. The set A is independent.
2. For $t = 0, 1, 2, \ldots, n$, we have $N_t(A) \leq t$.
3. If the tasks in A are scheduled in order of monotonically increasing deadlines, then no task is late.

(1)⇒(2) If $N_t(A) > t$ for some t, then there is no way to make a schedule with no late tasks for set A, because more than t tasks must finish before time t.

(2)⇒(3) There is no way to “get stuck” when scheduling the tasks in order of monotonically increasing deadlines, since (2) implies that the ith largest deadline is at least i.

(3)⇒(1) Obvious.

Tasks with Independent Sets Form a Matroid

- The problem of minimizing the sum of the penalties of the late tasks is the same as the problem of maximizing the sum of the penalties of the early tasks.

Theorem

If S is a set of unit-time tasks with deadlines, and $\mathcal{I}$ is the set of all independent sets of tasks, then the system $(S, \mathcal{I})$ is a matroid.

- Every subset of an independent set of tasks is certainly independent.

  For exchange, suppose that B and A are independent sets of tasks and that $|B| > |A|$. Let k be the largest t such that $N_t(B) \leq N_t(A)$ (such a t exists, since $N_0(A) = N_0(B) = 0$). Since $N_n(B) = |B|$ and $N_n(A) = |A|$, but $|B| > |A|$, we must have that $k < n$ and that $N_j(B) > N_j(A)$, for all j in the range $k+1 \leq j \leq n$. Therefore, B contains more tasks with deadline $k+1$ than A does. Let a_i be a task in $B - A$ with deadline $k+1$. Let $A' = A \cup \{a_i\}$.

Tasks with Independent Sets Form a Matroid (Cont'd)

Claim: A' is independent.

We use Property 2 of the preceding lemma. For $0 \leq t \leq k$, we have $N_t(A') = N_t(A) \leq t$, since A is independent. For $k < t \leq n$, we have $N_t(A') \leq N_t(B) \leq t$, since B is independent. Therefore, A' is independent.

- By the theorem, we can use a greedy algorithm to find a maximum-weight independent set of tasks A.

  We can then create an optimal schedule having the tasks in A as its early tasks.
- This method is an efficient algorithm for scheduling unit-time tasks with deadlines and penalties for a single processor:

  The running time is $O\left(n^2\right)$ using GREEDY, since each of the $O(n)$ independence checks made by that algorithm takes time $O(n)$.

Example

- An instance of the problem of scheduling unit time tasks with deadlines and penalties for a single processor is

a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

In this example, the greedy algorithm selects, in order, tasks a_1, a_2, a_3 and a_4, then rejects a_5 (because $N_4(\{a_1, a_2, a_3, a_4, a_5\}) = 5$) and a_6 (because $N_4(\{a_1, a_2, a_3, a_4, a_6\}) = 5$), and finally accepts a_7.

The final optimal schedule is $\langle a_2, a_4, a_1, a_3, a_7, a_5, a_6 \rangle$.

It has a total penalty incurred of $w_5 + w_6 = 50$.