Greedy Algorithms

Subhash Suri

April 10, 2019

1 Introduction

- Greedy algorithms are a commonly used paradigm for combinatorial algorithms. Combinatorial problems intuitively are those for which feasible solutions are *subsets* of a finite set (typically from items of input). Therefore, in principle, these problems can always be solved optimally in exponential time (say, $O(2^n)$) by examining each of those feasible solutions. The goal of a greedy algorithm is find the optimal by searching only a tiny fraction.

- In the 1980's iconic movie *Wall Street*, Michael Douglas shouts in front of a room full of stockholders: "Greed is good. Greed is right... Greed works." In this lecture, we will explore how well and when greed can work for solving computational or optimization problems.

- Defining precisely what a greedy algorithm is hard, if not impossible. In an informal way, an algorithm follows the Greedy Design Principle if it makes a series of choices, and each choice is *locally optimized*; in other words, when viewed in isolation, that step is performed optimally.

- The tricky question is when and why such myopic strategy (looking at each step individually, and ignoring the global considerations) can still lead to globally optimal solutions. In fact, when a greedy strategy leads to an optimal solution, it says something interesting about the *structure* (nature) of the problem itself! In other cases, even if the greedy does not give optimal, in many cases it leads to *provably good* (not too far from optimal) solution.

- Let us start with a trivial problem, but it will serve to illustrate the basic idea: *Coin Changing.*

- The US mint produces coins in the following four denominations: $25, 10, 5, 1$.

- Given an integer X between 0 and 99, making change for X involves finding coins that sum to X *using the least number of coins.* Mathematically, we can write $X = 25a + 10b + 5c + 1d$, so that $a + b + c + d$ is minimum where $a, b, c, d \geq 0$ are all integers.

- Greedy Coin Changing.

  - Choose as many quarters as possible. That is, find largest a with $25a \leq X$.
  - Next, choose as many dimes as possible to change $X - 25a$, and so on.
  - An example. Consider $X = 73$.
  - Choose 2 quarters, so $a = 2$. Remainder: $73 - 2 \times 25 = 23$.
  - Next, choose 2 dimes, so $b = 2$. Remainder: $23 - 2 \times 10 = 3$.
  - Choose 0 nickels, so $c = 0$. Remainder: 3.
  - Finally, choose 3 pennies, so $d = 3$. Remainder: $3 - 3 = 0$.
  - Solution is $a = 2, b = 2, c = 0, d = 3$.

- Does Greedy Always Work for Coin Changing? Prove that the greedy always produces optimal change for US coin denominations.

- Does it also work for other denominations? In other words, does the correctness of Greedy Change Making depend on the choice of coins?

- No, the greedy does not always return the optimal solution. Consider the case with coins types $\{12, 5, 1\}$. For $X = 15$, the greedy uses 4 coins: $1 \times 12 + 0 \times 5 + 3 \times 1$. The optimal uses 3 coins: 3×5. Moral: Greed, the quick path to success or to ruin!

2 Activity Selection, or Interval Scheduling

- We now come to a simple but interesting optimization problem for which a greedy strategy works, but the strategy is not obvious.
- The *input* to the problem is a list of N activities (jobs), each specified with a start and end time, which require the use of some resource.
- Only one activity can be be scheduled on the resource at a time; once an activity is started, it must be run to completion; no pre-preemption allowed.
- *What is the maximum possible number of activities we can schedule?*
- This is an abstraction that fits that many applications. For instance, activities can be computation tasks and resource the processor, or activities can be college classes and resource a lecture hall.
- More formally, we denote the list of activities as $S = \{1, 2, \ldots, n\}$.
- Each activity has a specific start time and a specific finish time; the duration of different activities can be different. Specifically, activity i is given as tuple $(s(i), f(i))$, where $s(i) \leq f(i)$, namely, that the finish time must be after the start time.
- For instance, suppose the input is $\{(3,6),(1,4),(1.2,2.5),(6,8),(0,2)\}$. Activities $(3,6)$ and $(6,8)$ are compatible—they can both be scheduled—since they do not overlap in their duration (endpoints are ok).
- Clearly a combinatorial problem: input is a list of n objects, and output is a *subset* of those objects.
- Each activity is pretty inflexible; if chosen, it must start at time $s(i)$ and end at $f(i)$.
- A subset of activities is a *feasible schedule* if no two activities overlap (in time).
- *Objective:* Design an algorithm to find a feasible schedule with as many activities as possible.

2.1 Potential Greedy Strategies

- The first obvious one is to pick the one that starts first. Remove those activities that overlap with it, and repeat.

```
  ----     ----     ----     ----
-----------------------------------
```

It is easy to see how this is not always optimal. Greedy picks just one (the longest one), while optimal has 4.

- A more sophisticated algorithm might repeatedly pick the activity with the smallest duration (and does not overlap with those already chosen). However, a simple example shows that this can also fail.

```
------------      -------------
         ------------
```

- Yet another possibility is to count the number of other jobs that overlap with each activity, and then choose the one with the smallest (overlap) count. This seems a bit better, and does get optimal for both the earlier two cases. Still, this also fails to guarantee optimality some times.

```
--------- ---------- --------- ---------
-----------     ---------    -----------
-----------                  -----------
-----------                  -----------
-----------                  -----------
```

The greedy starts by picking the one in the middle, which right away ensures that it cannot have more than 3. The optimal chooses the 4 in the top row.

2.2 The Correct Greedy Strategy for Interval Scheduling

- All these false starts and counterexamples should give you pause whether any greedy can ensure that it will find optimal in *all* cases.
- Fortunately, it turns out that there is such a strategy, though it may not be the one that seems the most natural.
- The correct strategy is to choose jobs in the *Earliest Finish Time* order!
- More precisely, we sort the jobs in the increasing order of their finish time. By simple relabeling of jobs, let us just assume that

$$f(j_1) \leq f(j_2) \leq f(j_3) \cdots \leq f(j_n)$$

- An example instance of the problem.

Activity	Start	Finish
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

- We can visualize the scheduling problem as choosing non-overlapping intervals along the time-axis. (In my example, jobs are already labeled this way.)

```
    -------------
            ----------
                      --------
                      ----------------
                          -----------------
                                  --------------
                                  -------------------
             ---------------------                   ----------
         -------------------------------------------------
--------------------------
0   1    2   3   4    5   6   7   8   9   10   11   12   13   14
```

- The greedy algorithm can now be described as follows:

```
A = {1}; j = 1;  // accept job 1
for i = 2 to n do
 if s(i) >= f(j) then
    A  = A + {i}; j = i;
return A
```

- In our example, the greedy algorithm first chooses 1; then skips 2 and 3; next it chooses 4, and skips 5, 6, 7; so on.

2.3 Analysis: Correctness

- It is not obvious that this method will (should) always return the optimal solution. After all, the previous 3 methods failed.
- But the method clearly finds a feasible schedule; no two activities accepted by this method can conflict; this is guaranteed by the *if* statement.
- In order to show optimality, let us argue in the following manner. Suppose OPT is an optimal schedule. Ideally we would like to show that it is always the case that $A \equiv OPT$; but this is too much to ask; in many cases there can be multiple different optima, and the best we can hope for is that their cardinalities are the same: that is, $|A| = |OPT|$; the contain the same number of activities.
- The proof idea, which is a typical one for greedy algorithms, is to show that the greedy *stays ahead of the optimal solution* at all times. So, step by step, the greedy is doing at least as well as the optimal, so in the end, we can't lose.
- Some formalization and notation to express the proof.
- Suppose $a_1, a_2, \ldots, a_k$ are the (indices of the) set of jobs in the Greedy schedule, and $b_1, b_2, \ldots b_m$ the set of jobs in an optimal schedule OPT. We would like to argue that $k = m$.
- Mnemonically, a_i's are jobs picked by our **a**lgorithm while b_i's are **b**est (optimal) schedule jobs.
- In both schedules the jobs are listed in increasing order (either by start or finish time; both orders are the same).
- Our intuition for greedy is that it chooses jobs in such a way as to make the resource *free again as soon as possible.* Thus, for instance, our choice of greedy ensures that

$$f(a_1) \leq f(b_1)$$

  (that is, a_1 finishes no later than b_1.)

- This is the sense in which greedy "stays ahead." We now turn this intuition into a formal statement.
- **Lemma:** *For any $i \leq k$, we have that $f(a_i) \leq f(b_i)$.*
- That is, the ith job chosen by greedy finishes no later than the ith job chosen by the optimal.

- Note that the jobs have non-overlapping durations in both greedy and optimal, so they are uniquely ordered left to right.
- **Proof.**
  - We already saw that the statement is true for $i = 1$, by the design of greedy.
  - We inductively assume this is true for all jobs up to $i - 1$, and prove it for i.
  - So, the induction hypothesis says that $f(a_{i-1}) \leq f(b_{i-1})$.
  - Since clearly $f(b_{i-1}) \leq s(b_i)$, we must also have $f(a_{i-1}) \leq s(b_i)$.
  - That is, the ith job selected by optimal is also available to the greedy for choosing as its ith job. The greedy may pick some other job instead, but if it does, it must be because $f(a_i) \leq f(b_i)$. Thus, the induction step is complete.
- With this technical insight about the greedy, it is now a simple matter to wrap up the greedy's proof of optimality.
- **Theorem:** The greedy algorithm returns an optimal solution for the activity selection problem.
- **Proof.** By contradiction. If A were not optimal, then an optimal solution OPT must have more jobs than A. That is, $m > k$.
- Consider what happens when $i = k$ in our earlier lemma. We have that $f(a_k) \leq f(b_k)$. So, the greedy's last job has finished by the time OPT's kth job finishes.
- If $m > k$, meaning there is at least one other job that optimal accepts, that job is also available to Greedy; it cannot conflict with anything greedy has scheduled. Because the greedy does not stop until it no longer has any acceptable jobs left, this is a contradiction.

2.4 Analysis: Running time

- The greedy strategy can be implemented in worst-case time $O(n \log n)$. We begin by sorting the jobs in increasing order of their finish times, which takes $O(n \log n)$. After that, the algorithm simply makes one scan of the list, spending a constant time per job.
- So total time complexity is $O(n \log n) + O(n) = O(n \log n)$.

3 Interval Partitioning Problem

- Let us now consider a different scheduling problem:

  > *given the set of activities, we must schedule them all using the minimum number of machines (rooms).*

- An example.
- An obvious greedy algorithm to try is the following:

  > *Use the Interval Scheduling algorithm to find the max number of activities that can be scheduled in one room. Delete and repeat on the rest, until no activities left.*

- Surprisingly, this algorithm does not always produce the optimal answer.

```
   a      b                c
------ ------ ----------------------------

          d               e          f
---------------------  -------   ---------
```

  These activities can be scheduled in 2 rooms, but Greedy will need 3, because d and c cannot be scheduled in the same room.

- Instead a different, and simpler, Greedy works.

```
Sort activities by start time.
Start Room 1 for activity 1.
for i = 2 to n
      if activity i can fit in any existing room, schedule it in that room
      otherwise start a new room with activity i
```

- **Proof** of Correctness. Define *depth* of activity set as the maximum number of activities that are concurrent at any time.
- Let depth be D. Optimal must use at least D rooms. Greedy uses no more than D rooms.

4 Data Compression: Huffman Codes

- Huffman coding is an example of a beautiful algorithm working behind the scenes, used in digital communication and storage. It is also a fundamental result in theory of data compression.
- For instance, mp3 audio compression scheme basically works as follows:
  1. The audio signal is digitized by sampling at, say, 44KHz.
  2. This produces a sequence of real numbers $s_1, s_2, \ldots, s_T$. For instance, a 50 min symphony corresponds to $T = 50 \times 60 \times 44000 = 130M$ numbers.
  3. Each s_i is *quantized*, using a finite set G (e.g., 256 values for 8 bit quantization.) The quantization is fine enough that human ear doesn't perceive the difference.
  4. The quantized string of length T over *alphabet* G is encoded in binary.
  5. This last step uses Huffman encoding.
- To get a feel for compression, let's consider a toy example: a data file with $100,000$ characters.
- Assume that the cost of storage or transmission is proportional to the number of bits required. What is the best way to store or transmit this file?
- In our example file, there are only 6 different characters (G), with their frequencies as shown below.

Char	a	b	c	d	e	f
Freq(K)	45	13	12	16	9	5

- We want to design binary codes to achieve maximum compression. Suppose we use fixed length codes. Clearly, we need 3 bits to represent six characters. One possible such set of codes is:

Char	a	b	c	d	e	f
Code	000	001	010	011	100	101

- Storing the 100K character requires 300K bits using this code. Is it possible to improve upon this?
- **Huffman Codes.** We can improve on this using *Variable Length Codes.*

- Motivation: use shorter codes for more frequent letters, and longer codes for infrequent letters.
- (A similar idea underlies Morse code: $e =$ dot; $t =$ dash; $a =$ dot-dash ; etc. But Morse code is a heuristics; not optimal in any formal sense.)
- One such set of codes shown below.

Char	a	b	c	d	e	f
VLC	0	101	100	111	1101	1100

- Note that some codes are smaller (1 bit), while others are longer (4 bits) than the fixed length code. Still, using this code2, the file requires

$$1 \times 45 + 3 \times 13 + 3 \times 12 + 3 \times 16 + 4 \times 9 + 4 \times 5$$

  Kbits, which is 224 Kbits.
- Improvement is 25% over fixed length codes. In general, variable length codes can give $20 - 90\%$ savings.
- **Problems with Variable Length Codes.** We have a potential problem with variable length codes: while with fixed length coding, decoding is trivial, it is not the case for variable length codes.
- Example: Suppose 0 and 000 are codes for letters x and y. What should decoder do upon receiving 00000?
- We could put special marker codes but that reduce efficiency.
- Instead we consider **prefix codes**: *no codeword is a prefix of another codeword.*
- So, 0 and 000 will not be prefix codes, but $(0, 101, 100, 111, 1101, 1100)$, the example shown earlier, do form a prefix code.
- To encode, just concatenate the codes for each letter of the file; to decode, extract the first valid codeword, and repeat.
- Example: Code for 'abc' is 0101100. And '001011101' uniquely decodes to 'aabe'.

4.1 Representing Codes by a Tree

- Instead of the table-based format, the coding and decoding is more intuitive to describe using a Binary Tree format, in which characters are associated with leaves.
- Code for a letter is the sequence of bits between root and that leaf.

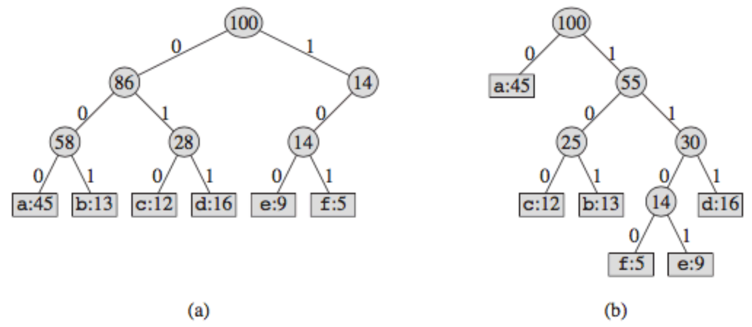

Figure 16.4 Trees corresponding to the coding schemes in Figure 16.3. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. **(a)** The tree corresponding to the fixed-length code a = 000, ..., f = 101. **(b)** The tree corresponding to the optimal prefix code a = 0, b = 101, ..., f = 1100.

4.2 Measuring Optimality

- Before we claim the optimality of a coding scheme, we must agree on a precise quantitative *measure* by which we evaluate the goodness (figure of merit). Let us formalize this.

- Let C denote the alphabet. Let $f(p)$ be the frequency of a letter p in C. Let T be the tree for a prefix code; let $d_T(p)$ be the depth of p in T. The number of bits needed to encode our file using this code is:

$$B(T) = \sum_{p \in C} f(p) d_T(p)$$

- Think of this as *bit complexity.* We want a code that achieves the minimum possible value of $B(T)$.
- **Optimal Tree Property.** *An optimal tree must be full: each internal node has two children. Otherwise we can improve the code.*
- Thus, by inspection, the fixed length code above is not optimal!
- **Greedy Strategies.** Ideas for optimal coding??? Simple obvious heuristic ideas do not work; a useful exercise will be to try to "prove" the correctness of your suggested heuristic.
- **Huffman Story:** Developed his coding procedure, in a term paper he wrote while a graduate student at MIT. Joined the faculty of MIT in 1953. In 1967, became the founding faculty member of the Computer Science Department at UCSC. Died in 1999.
- Excerpt from an Scientific American article about this:

> In 1951 David A. Huffman and his classmates in an electrical engineering graduate course on information theory were given the choice of a term paper or a final exam. For the term paper, students were asked to find the most efficient method of representing numbers, letters or other symbols using a binary code. $\cdots$ Huffman worked on the problem for months, developing a number of approaches, but none that he could prove to be the most efficient. Finally, he despaired of ever reaching a solution and decided to start studying for the final. Just as he was throwing his notes in the garbage, the solution came to him. "It was the most singular moment of my life," Huffman says. "There was the absolute lightning of sudden realization."
>
> Huffman says he might never have tried his hand at the problem—much less solved it at the age of 25—if he had known that Fano, his professor, and Claude E. Shannon, the creator of information theory, had struggled with it.
>
> Huffman Codes are used in nearly every application that involves the compression and transmission of digital data, such as fax machines, modems, computer networks, and high-definition television.

- Huffman's Algorithm. The algorithm constructs the binary tree T representing the optimal code.
- Initially, each letter represented by a single-node tree. The weight of the tree is the letter's frequency.
- Huffman repeatedly chooses the two smallest trees (by weight), and merges them. The new tree's weight is the sum of the two children's weights.
- If there are n letters in the alphabet, there are $n-1$ merges.
- In the pseudo-code, below Q is a priority Queue (say, heap).

```
1. $Q \leftarrow C$
2. for $i = 1$ to $n-1$ do
     – $z \leftarrow allocateNode()$
     – $x \leftarrow left[z] \leftarrow DeleteMin(Q)$
     – $y \leftarrow right[z] \leftarrow DeleteMin(Q)$
     – $f[z] \leftarrow f[x] + f[y]$
     – $Insert(Q, z)$
3. return $FindMin(Q)$
```

8. Illustration of Huffman Algorithm.

Initial	f:5	e:9	c:12	b:13	d:16	a:45
Merge/Reorder	c:12	b:13	f+e:14	d:16	a:45	
Next	f+e:14	d:16	c+b:25	a:45		
Next	c+b:25	(f+e)+d:30	a:45			
Next	a:45	(c+b)+((f+e)+d):55				

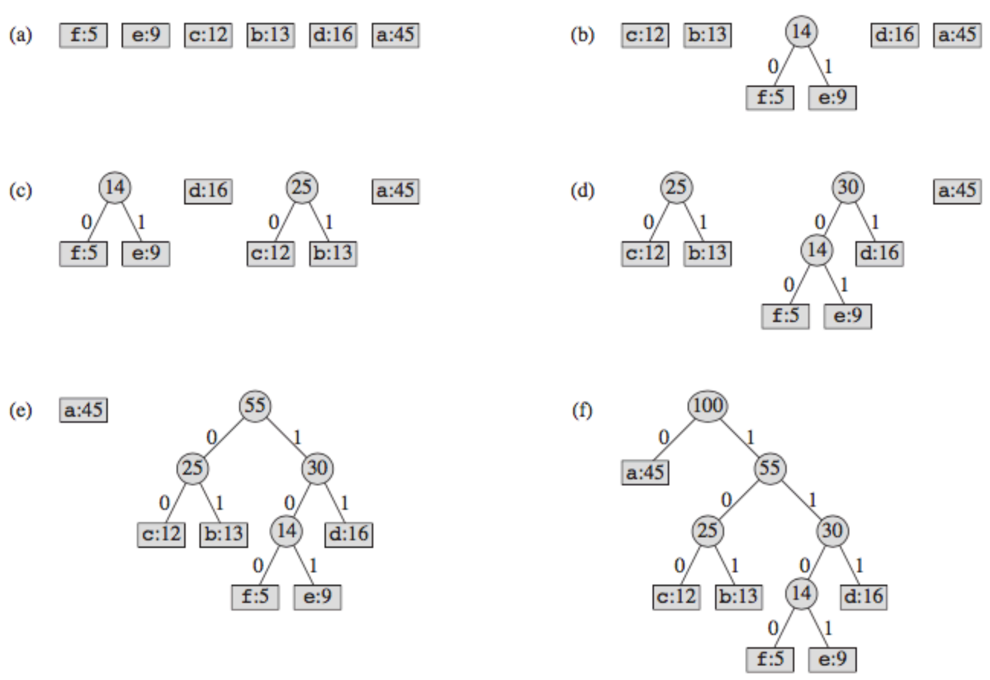

Figure 16.5 The steps of Huffman's algorithm for the frequencies given in Figure 16.3. Each part shows the contents of the queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged. Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of their children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child. The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter. **(a)** The initial set of $n = 6$ nodes, one for each letter. **(b)–(e)** Intermediate stages. **(f)** The final tree.

4.3 Analysis of Huffman

- Time complexity is $O(n \log n)$. Initial sorting plus n heap operations.
- We now prove that the prefix code generated is optimal. It is a greedy algorithm, and we use the swapping argument.

- Proof intuition:
  1. We will use proof by induction on the size of the alphabet $|C|$.

2. The base case of $|C| = 2$ is trivial: we have a depth 1 tree, with two leaves, each with codelength 1.
3. In the general case, we will assume that induction holds for $|C| = n - 1$, and prove it for $|C| = n$.
4. Which character x_i should we drop for induction? As a thought experiment, suppose we drop x_n. Then, by induction, we have an optimal solution for the subproblem on $\{x_1, x_2, \ldots, x_{n-1}\}$.
5. This is a tree with $n - 1$ leaves. How can we grow it into an n-leaf tree? We get stuck!
6. Instead, we are going to try a different idea. We take the last two characters x_{n-1} and x_n. We combine them into a single new character z, and set $f(z) = f(x_{n-1}) + f(x_n)$.
7. We now remove x_{n-1} and x_n from C and replace them with z. The new, reduced set has size $|C'| = n - 1$.
8. By induction, we find the optimal code tree of C'. This tree has z at some leaf. We now expand z by attaching the original nodes x_{n-1} and x_n, which turns the tree for C' into one for C, while preserving the prefix property.
9. We are going to show that given optimal tree for C', this new tree will be optimal for C.
10. This still has one problem: in our construction, the nodes x_{n-1} and x_n will necessarily end up as *siblings.* (That is, the codes for these two will be identical except in the last bit.)
11. *How can we choose x_{n-1} and x_n at the onset so that in the optimal tree they are guaranteed to have this property?*
12. This is where Huffman's greedy choice enters the proof: *we will choose two lowest freq. characters.*

- **Lemma:** *Suppose x and y are the two letters of lowest frequency. Then, there exists an optimal prefix code in which codewords for x and y have the same (and maximum) length and they differ only in the last bit.*
- **Proof.** The idea of the proof is to take the tree T representing an optimal prefix code, and modify it to make a tree representing another optimal prefix code in which the characters x and y appear as sibling leaves of max depth.

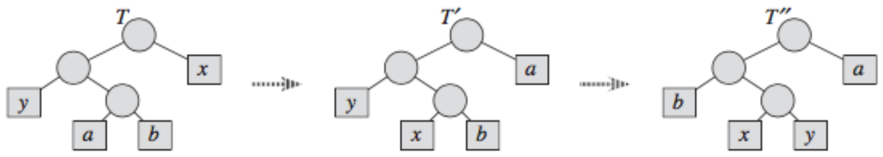

Figure 16.6 An illustration of the key step in the proof of Lemma 16.2. In the optimal tree T, leaves a and b are two siblings of maximum depth. Leaves x and y are the two characters with the lowest frequencies; they appear in arbitrary positions in T. Assuming that $x \neq b$, swapping leaves a and x produces tree T', and then swapping leaves b and y produces tree T''. Since each swap does not increase the cost, the resulting tree T'' is also an optimal tree.

- In that case, x and y will have the same code length, with only the last bit different.
- Assume an optimal tree that does not satisfy the claim. Assume, without loss of generality, that a and b are the two characters that are sibling leaves of max depth in T. Without loss of generality, assume that

$$f(a) \leq f(b) \quad \text{and} \quad f(x) \leq f(y)$$

- Because $f(x)$ and $f(y)$ are 2 lowest frequencies, we get:

$$f(x) \leq f(a) \quad \text{and} \quad f(y) \leq f(b)$$

- (Remark. Note that x, y, a, b need not all be distinct; for instance, may be y lies at the max depth and is therefore one of a or b.)
- We first transform T into T' by swapping the positions of x and a.
- Intuitively, since $d_T(a) \geq d_T(x)$ and $f(a) \geq f(x)$, the swap does not increase the *frequency* $\times$ *depth* cost. Specifically,

$$\begin{aligned}
B(T) - B(T') &= \sum_p [f(p)d_T(p)] - \sum_p [f(p)d'_T(p)] \\
&= [f(x)d_T(x) + f(a)d_T(a)] - [f(x)d'_T(x) + f(a)d'_T(a)] \\
&= [f(x)d_T(x) + f(a)d_T(a)] - [f(x)d_T(a) + f(a)d_T(x)] \\
&= [f(a) - f(x)] \times [d_T(a) - d_T(x)] \\
&\geq 0
\end{aligned}$$

- Thus, this transformation does not increase the total bit cost.
- Similarly, we then transform T' into T'' by exchanging y and b, which again does not increase the cost.

  So, we get that $B(T'') \leq B(T') \leq B(T)$. If T was optimal, so is T'', but in T'' x and y are sibling leaves and they are at the max depth.
- This completes the proof of the lemma.

- We can now finish the proof of Huffman's optimality.
- But need to be very careful about the proper use of induction. For instance, here is a simple and bogus idea: *Base case: two characters. Huffman's algorithm is trivially optimal. Suppose it's optimal for $n-1$, and consider n characters. Delete the largest frequency character, and build the tree for the remaining $n-1$ characters. Now, add the nth character as follows: create a new root; make the nth character left child, and hang the $(n-1)$ subtree as its right child. Even though this tree will have the properly of Lemma above (two smallest characters being deepest leaves, and the largest frequency having the shortest code), this proof is all wrong—it neither shows that the resulting tree is optimal, nor is that tree even the output of actual Huffman's algorithm.*
- Instead, we will do induction by removing two smallest keys, replacing them with their "union" key, and looking at the difference in the tree when those leaves are added back in.

- When x and y are merged; we pretend a new character z arises, with $f(z) = f(x)+f(y)$.
- We compute the optimal code/tree for these $n-1$ letters: $C + \{z\} - \{x, y\}$. Call this tree T_1.
- We then attach two new leaves to the node z, corresponding to x and y, obtaining the tree T. This is now the Huffman Code tree for character set C.
- **Proof of optimality.** The cost $B(T)$ can be expressed in terms of cost $B(T_1)$, as follows. For each character p not equal to x and y, its depth is the same in both trees, so no difference.
- Furthermore, $d_T(x) = d_T(y) = d_{T_1}(z) + 1$, so we have

$$\begin{aligned} f(x)d_T(x) + f(y)d_T(y) &= [f(x) + f(y)] \times [d_{T_1}(z) + 1] \\ &= f(z)d_{T_1}(z) + [f(x) + f(y)] \end{aligned}$$

- So, $B(T) = B(T_1) + f(x) + f(y)$.
- We now prove the optimality of Huffman algorithm by contradiction. Suppose T is not an optimal prefix code, and another tree T' is claimed to be optimal, meaning $B(T') < B(T)$.
- By the earlier lemma, T' has x and y as siblings. Let T_1' be this tree with the common parent of x and y replaced by a leave z, whose frequency is $f(z) = f(x) + f(y)$.
- Then,

$$B(T_1') \; = \; B(T') - f(x) - f(y) \; < \; B(T) - f(x) - f(y) \; < \; B(T_1)$$

  which contradicts the assumption that T_1 is an optimal prefix code for the character set $C' = C + \{z\} - \{x, y\}$. End of proof.

4.4 Beyond Huffman codes

- Is Huffman coding the end of the road, or are there other coding schemes that are even better than Huffman?
- Depends on problem assumptions.

- Huffman code does not adapt to variations in the text. For instance, if the first half is mostly a, b, and the second half c, d, one can do better by adaptively changing the encoding.
- One can also get better-than-Huffman codes by coding longer words instead of individual characters. This is done in arithmetic coding. Huffman coding is still useful because it is easier to compute (or you can rely on a table, for instance using the frequency of characters in the English language).
- There are also codes that serve a different purpose than Huffman coding: error detecting and error correcting codes, for example. Such codes are very important in some areas, in particular in industrial applications.
- One can also do better by allowing lossy compression.
- Even if the goal is lossless compression, depending on the data, Huffman code might not be suitable: music, images, movies, and so on.
- One practical disadvantage of Huffman coding is that it requires 2 passes over the data: one to construct to code table, and second to encode, which means it can be slow and also not suitable for streaming data. Not being error-correcting is also a weakness.

5 Horn Formulas

- Horn formulas are a particular form of boolean logic, and often used in AI systems for logical reasoning.
- Each boolean variable represents an event (or possibility), such as
  1. $x =$ the murder took place in the kitchen
  2. $y =$ the butler is innocent
  3. $z =$ the colonel was asleep at 8 pm.
- A Boolean variable can only take one of two values { true, false }. A *literal* is either a variable x or its negation $\bar{x}$.
- In Horn Formula, constraints among variables is represented by two kinds of clauses.
- **Implication Clauses:** whose left-hand-side is an AND of any number of *positive* literals, and whose right-hand-side is a single *positive* literal.

$$(z \wedge y) \;\Rightarrow\; x$$

  It asserts that "if the colonel was asleep at 8 pm, and the murder took place at 8 pm, then the murder took places in the kitchen."

  A degenerate statement of the type " $\Rightarrow x$" means that x is unconditionally true. For instance, "the murder definitely occurred in the kitchen."

- **Negative Clauses.** Such a clause consists of an OR of any number of *negative* literals, as in $(\bar{u} \;\vee\; \bar{t} \;\vee\; \bar{y})$, where u, t, y, resp., means that constable, colonel, and butler is innocent. This clause means asserts that "they can't all be innocent."
- A Horn formula is a set of implications and negative clauses.
- **Problem:** Given a Horn formula, determine it is satisfiable. That is, is there a true/false assignment of variables where all clauses are satisfied. Such an assignment is called a *satisfying assignment.*
- Examples:

1. The following Horn formula

$$\Rightarrow\ x,\quad \Rightarrow\ y,\quad x \wedge u\ \Rightarrow\ z,\quad \bar{x} \vee \bar{y} \vee \bar{z}$$

   has a satisfying assignment

$$u = 0,\ x = 1,\ y = 1,\ z = 0$$

2. But the following formula

$$\Rightarrow\ x,\quad \Rightarrow\ y,\quad x \wedge y\ \Rightarrow\ z,\quad \bar{x} \vee \bar{y} \vee \bar{z}$$

   is not satisfiable.

3. How do we decide this for a general Horn formula, such as the following?

$$\Rightarrow x,\quad (x \wedge z) \Rightarrow w,\quad (w \wedge y \wedge z) \Rightarrow x$$
$$x \Rightarrow y,\quad (x \wedge y) \Rightarrow w,\quad (\bar{w} \vee \bar{x} \vee \bar{y})$$

- Trying each setting of boolean variables requires testing 2^n possible settings.
- The nature of Horn clauses suggests a natural greedy algorithm:
  1. Initially set all variables to false.
  2. While there is an unsatisfied Implication clause, set its RHS to true.
  3. If all pure negative clauses are satisfied, return the assignment; otherwise, formula is not satisfiable.
- In the example formula above, we start with $w = x = y = z = 0$.
- Then, the first implication forces $x = 1$, which in turns uses $x \Rightarrow y$ to set $y = 1$.
- Once $x = y = 1$, we are forced to have $w =$.
- However, with $x = y = w = 1$, the negative clause cannot be satisfied, and so this formula is not satisfiable.
- To prove that this algorithm is correct, we reason as follows. Clearly, if the algorithm returns a satisfying assignment, then it is a valid assignment because it satisfies all negative and implication clauses.
- In order to show that if the algorithm does not find a satisfying assignment, there is none, we observe that the algorithm maintains the following invariant.

If a certain set of variables is set to true, then they must be true in any satisfying assignment. Namely, we only set a variable true when it is forced upon us.

- Horn formulas lie at the heart of Prolog programming language, and with some care the greedy algorithm can be implemented in linear time (in the length of the formula).

6 Set Cover

- The set cover problem is a fundamental problem, often used to model complex decision making.
- In the basic formulation, we have a *base set* $\mathcal{B} = \{1, 2, \ldots, n\}$ of n elements, and a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ of m subsets, with each $S_i \subseteq \mathcal{B}$.
- The problem is to choose the smallest number of subsets whose union is $\mathcal{B}$. That is, the chosen sets jointly cover all the elements of $\mathcal{B}$.
- As one example of Set Cover's modeling power, consider the following problem. We have a collection of n towns in a county. The county is deciding where to put schools, subject to the following two constraints: (1) each school should be in a town, and (2) no one should have to travel more than 30 miles to reach one of the schools.
- For the purposes of modeling, we assume each town is small enough that it can be represented as a point on the map, and so the inter-town distances are really the inter-point distances.
- For each town x, let S_x be the subset of towns within 30 miles of it. A school at x will essentially "cover" all the towns of S_x. How many sets S_x must be picked to cover all the towns in the county?
- The problem suggests a natural greedy algorithm.
  Repeat until all elements of $\mathcal{B}$ are covered.
  Pick the set S_t containing the largest number of still-uncovered elements.
- Example from DPV book.
- The greedy algorithm will pick four sets. But there is a better solution with 3 sets.
- So, while the greedy does not give an optimal answer, its solution is not far from the optimal.
- **Claim.** Suppose B contains n elements, and the optimal cover consists of k sets. Then, the greedy algorithm returns a solution with at most $k \ln n$ sets.
- **Proof.**
  1. Let n_t be the number of elements still not covered after t iterations of the greedy algorithm. That is, $n_0 = n$.

2. Since these remaining elements are covered by the optimal k sets, there must be some set that covers at least n_t/k of them.

   (Do we need to worry that this set may not be available any more—because greedy may have already picked it? No, because we are focusing on only those elements that are still uncovered—all elements of any set added to G are already in the covered portion. Equivalently, consider the reduced problem where the base set is just the set of remaining items, and now consider the optimal solution's behavior on this reduced set.)

3. Since the greedy chooses a set covering most uncovered elements, we have

$$t_{t+1} \;\leq\; n_t - \frac{n_t}{k} \;=\; n_t(1 - \frac{1}{k})$$

4. By repeated application, we get $n_t \;\leq n_0(1 - 1/k)^t$.

5. By using the inequality $1 - x \leq e^{-x}$, for all x, we get

$$n_t \;\leq n_0(1 - \frac{1}{k})^t \;<\; n_0 e^{-t/k} \;=\; n e^{-t/k}$$

6. At $t = k \ln n$, therefore n_t is strictly smaller than $ne^{-\ln} = 1$, which means that no elements remain to be covered. This completes the proof.

7 Greedy Algorithms in Graphs

- The shortest path algorithm of Dijkstra and minimum spanning algorithms of Prim and Kruskal are instances of greedy paradigm.
- We briefly review the Dijkstra and Kruskal algorithms, and sketch their proofs of optimality, as examples of greedy algorithm proofs.
- Dijkstra's algorithm can be described as follows (somewhat different from the way it's implemented, but equivalent.)
- **Dijkstra's Algorithm.**
  1. Let S be the set of *explored nodes.*
  2. Let $d(u)$ be the shortest path distance from s to u, for each $u \in S$.
  3. Initially $S = \{s\}$, and $d(s) = 0$.
  4. While $S \neq V$ do
     (a) Select $v \notin S$ with the minimum value of
     $$d'(v) \;=\; \min_{(u,v), u \in S} \{d(u) + cost(u, v)\}$$
     (b) Add v to S, set $d(v) = d'(v)$.
- Example.
- **Proof of Correctness.**
  1. We show that at any time $d(u)$ is the shortest path distance to u for all u in S.
  2. Consider the instant when node v is chosen by the algorithm. Let (u, v) be the edge, with $u \in S$, that is incident to v.
  3. Suppose, for the sake of contradiction, that $d(u) + cost(u, v)$ is not the shortest path distance to v. Instead a shorter path P exists to v.
  4. Since that path starts at s, it has to leave S at some node. Let x be that node, and let $y \notin S$ be the edge that goes from S to $\overline{S}$.

5. So our claim is that $length(P) = d(x) + cost(x, y) + length(y, v)$ is shorter than $d(u) + cost(u, v)$. But note that the algorithm chose v over y, so it must be that $d(u) + cost(u, v) \leq d(x) + cost(x, y)$.
6. In addition, since $length(y, v) > 0$, this contradicts our hypothesis that P is shorter than $d(u) + cost(u, v)$.
7. Thus, the $d(v) = d(u) + cost(u, v)$ is correct shortest path distance.

- Kruskal's MST algorithm has a similar proof of correctness.

  1. For simplicity, assume that all edge costs are distinct so that the MST is unique. Otherwise, add a tie-breaking rule to consistency order the edges.
  2. Proof by contradiction: let (v, w) be the first edge chosen by Kruskal that is not in the optimal MST.
  3. Consider the state of the Kruskal just before (v, w) is considered.
  4. Let S be the set of nodes connected to v by a path in this graph. Clearly, $w \notin S$.
  5. The optimal MST does not contain (v, w) but must contain a path connecting v to w, by virtue of being spanning.
  6. Since $v \in S$ and $w \notin S$, this path must contain at least one edge (x, y) with $x \in S$ and $y \notin S$.
  7. Note that (x, y) cannot be in Kruskal's graph *at the time* (v, w) *was considered* because otherwise y will have been in S.
  8. Thus, (x, y) is more expensive than (v, w) because it came after (v, w) in Kruskal's scan order.
  9. If we replace (x, y) with (v, w) in the optimal MST, it remains spanning and has lower cost, which contradicts its optimality.
  10. So, the hypothesis that (v, w) is not in optimal must be false.