Representing Prefix Codes

Can represent a prefix code by a binary tree:

- each path from the root to a leaf represents a code word (for symbol in C, the original alphabet)
  - 0010 = left, left, right, left

Fix a file and a tree T corresponding to a prefix code. How many bits does it take to encode the file?

- let $f(c)$ be the frequency with which the character c in alphabet C occurs in the file
- Let $d_T(c)$ be the length of the codeword for c (according to T)

Then the cost of T (number of bits used by T) is

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

It can be shown that optimal compression can be achieved by using a prefix code.

Constructing a Huffman Code

Assume that C is a set of characters and f describes the frequency of the characters in C.

Huffman(C, f) constructs a tree whose leaves are labeled by characters in C

- The path from the root to the node is the code for that chactacter
- The algorithm builds intermediate nodes in the tree by "merging" two nodes and giving them a parent.
  - if the cost of merging x and y is $f[x] + f[y]$, it chooses the cheapest nodes to merge

HUFFMAN(C, f)

```
1  n ← |C|
2  Q ← C   [Q is a priority queue, sorted by frequency]
3  for i ← 1 to n − 1 do
4       z ← ALLOCATE-NODE()    [create a new node]
5       x ← EXTRACT MIN(Q)
6       y ← EXTRACT MIN(Q)
7       left[z] ← x
8       right[z] ← y
9       f[z] ← f[x] + f[y]
10      INSERT(Q, z)   [replace x and y in Q by z]
11 return EXTRACT-MIN(Q)
```

After $n - 1$ steps, Q has exactly one element.

Running time: $O(n \log n)$

Huffman: Correctness

If x and y are the characters in Q with the lowest frequency, their codewords agree except in the last bit (since they have the same parent).

Lemma 1: If x and y are the two characters in C with the lowest frequency, there is an optimal prefix code in which the codewords for x and y have the same length and differ only in the last bit.

Proof: Suppose T is a tree representing an optimal prefix code.

- Every interior node has two children
- If not, can shrink T to get a better code

Idea: modify T so that x and y are sibling leaves.

Let b and c be sibling leaves of maximum depth.

- Suppose that $f(b) \leq f(c)$ and $f(x) \leq f(y)$.
- Switch b and x and switch c and y in the tree to get a better tree.

Bottom line: greedy merging is safe.

Lemma 2: If T represents the optimal prefix code for (C, f), x and y are characters in C that are siblings in T, and z is their parent. Let $C' = C - \{x, y\} \cup \{z\}$, and suppose that $f(z) = f(x) + f(y)$. Then $T' = T - \{x, y\}$ represents an optimal prefix code for C'.

Proof: First let's show that

$$B(T) = B(T') + f(x) + f(y)$$

Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$$\begin{aligned} B(T) &= (\Sigma_{c \in C' - \{z\}} f(c) d_T(c)) + f(x) d_T(x) + f(y) d_T(y) \\ &= (\Sigma_{c \in C' - \{z\}} f(c) d_T(c)) + (f(x) + f(y)(d_T(z) + 1) \\ &= (\Sigma_{c \in C' - \{z\}} f(c) d_T(c)) + f(z)) d_T(z) + f(x) + f(y) \\ &= B(T') + f(x) + f(y) \end{aligned}$$

If there exists a better prefix code T'' for C', add x and y as children of z to get a better prefix code for C, with cost $B(T'') + f(x) + f(y)$, by the same argument.

Theorem: HUFFMAN produces an optimal code.

Proof: By induction on the size of C, using Lemmas 1 and 2.

Back to the Knapsack Problem

Remember the 0-1 knapsack problem:

- there are n items
- Item i has value v_i and weight w_i.

You can put at most W pounds into a knapsack. Which items do you take?

- For each item, you either take it or leave it (0-1)
- You can't solve it using a greedy algorithm
- There is a dynamic programming algorithm that runs in time nW.

Is there an algorithm whose running time depends only on n, not on W?

- Sure: try every possible subsets of items, see if its weight is $\leq W$; if so, figure out its value.
- Choose the subset that gives the highest value.

There are 2^n subsets, so this takes time $O(2^n)$.

- Can we do better?

We don't think so ...; this is an *NP*-complete problem.

Intrinsic Complexity

Computer scientists tries to characterize the *intrinsic complexity* of a problem.

- How hard is sorting?
- How hard is multiplication?
- How hard is it to solve the 0-1 knapsack problem

Clearly the difficulty is a function of the input size.

- We can get an *upper bound* on the complexity by giving an algorithm.
- We also try to prove *lower bounds*.
  - They are supposed to hold for *any* algorithm that solves a problem.

Big success story: sorting.

- $O(n \log n)$ is an upper bound.
- There's a matching $\Theta(n \log n)$ lower bound for any algorithm that uses only comparisons.

NP-Completeness

An even bigger success story: NP-completeness.

- Many important problems are *NP complete*
  - Scheduling, 0-1 knapsack, travelling salesman problem, thousands of others ...
- They can be solved in *nondeterministic* polynomial time
  - It's hard to guess an answer (there are too many choices)
  - It's easy to verify that an answer is correct
- They are the *hardest* problems among those that can be solved in nondeterminstic polynomial time.
- All NP-complete problems are equally hard.
- If any one can be solved in polynomial time, they all can
- If you can prove that a problem is NP-complete, it's probably not worth trying to come up with an efficient algorithm
  - you should probably start looking for good approximation algorithms.

Polynomial Time

Why the focus on polynomial time (PTIME, or just P) in the literature?

- PTIME has generally been identified with "easy":
  - $O(n^{100})$ doesn't seem so easy but ...
  - Almost almost practical problems that have PTIME solutions end up having solution in $O(n^3)$ or better
- PTIME is a much more robust notion than, say, $O(n^2)$.
  - An algorithm that runs in $O(n^2)$ on one architecture might run in $O(n^3)$ on another
    - How are, say, array operations implemented?
  - Is n the number of vertices + edges or the number of bits required to describe them?
    - This could make a difference between $O(n^2)$ and $O(n^2 \log n)$.
  - All these subtleties disappear with PTIME
- Polynomials are closed under addition, multiplication, and composition, so the combination of PTIME problems is typically PTIME

Hamiltonian Paths vs. Eulerian Paths

Given a graph, an *Eulerian path* traverses every edge exactly once:

- This is the street-sweeper problem
- Can you sweep all the streets without going over any street twice

A *Hamiltonian path* visits every node exactly once

- This is the traveling salesman problem

An Eulerian/Hamiltonian cycle is an Eulerian/Hamiltonian path that starts and ends at the same vertex.

Deciding if a graph has an Eulerian path is easy:

- It has an Eulerian path iff either all edges have even degree or exactly two edges have odd degree.
- It has an Eulerian cycle iff all edges have even degree.
- If a graph has an Eulerian path, it's easy to find it.

The problem of finding a Hamiltonian path/cycle is NP-complete.

Decision Problems vs. Search Problems

Many problems of interest are *search problems*.

- *Find* an object with certain properties
  - find a Hamiltonian path,
  - find an optimal solution to the 0-1 knapsack problem,
  - find a shortest path

For technical reasons, complexity theory focuses on *decision problems*.

- Problems with yes/no answers
  - Does this graph have a Hamiltonian path?
  - Is there a way of filling the knapsack that has value at least V.
  - Is there a path from s to t of length less than k in this graph.

Can typically recast a decision problem as a search problem

- E.g., convert optimization problem to asking whether a solution with value at least k exists.

If we can solve the search problem, we can solve the corresponding decision problem.

- That means if we can prove the decision problem is hard, the search problem is also hard.

Conversely, solving the decision problem means we can usually solve the search problem quickly:

- E.g., in an optimization problem, use binary search to find the optimal solution.

Formal Language Theory

We can recast all decision problems as the question of deciding whether a given string is in a *language* (set of strings).

An *alphabet* is a finite set of symbols (e.g., 0, 1).

If Σ is an alphabet, Σ^* consists of all finite strings that use the symbols in the alphabet.

- E.g., $\{0,1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$
- ϵ denotes the empty string

A *language* (over Σ) is a subset of Σ^*.

If $\Sigma = \{a_1, \ldots, a_k\}$, can convert any string in Σ^* to a string in $\{0,1\}^*$ with a blowup of $\log(k)$:

- Use a fixed-length code:
  - every symbol in Σ becomes a string of 0s and 1s of length $\log(k)$.

As far as PTIME is concerned, there is no loss of generality in looking at languages in $\{0,1\}^*$.

Operations on languages:

- $L_1 \cup L_2$
- $\overline{L} = \Sigma^* - L$ (complement)
- $L_1 \cap L_2$
- *concatenation:* $L_1 L_2 = \{x_1 x_2 : x_1 \in L_1, x_2 \in L_2\}$
- $L^* = \epsilon \cup L \cup L^2 \cup L^3 \cup \ldots$

Examples: If $L_1 = \{0\}$, $L_2 = \{1\}$, then

- $L_1 L_2 = \{01\}$
- $L_1 \cup L_2 = \{0, 1\}$
- $(L_1 L_2)^* = \{\epsilon, 01, 0101, 010101, \ldots\}$
- $(L_1 \cup L_2)^* = \Sigma^*$

Encoding problems as languages

E.g.: Hamiltonian paths as a language problem
First step: describe a graph as a string in $\{0,1\}^*$.

- Assume the vertices are in $\{1, \ldots, n\}$
- First write description in the language $\{0, 1, \$\}^*$:
  - Description is string $n\$i_1\$j_1\$\ldots\$i_m\$j_m\$$
  - n is #nodes (written in binary)
  - $i_k\$j_k\$$ describes edge (i_k, j_k)
    - write i_k, j_k in binary
    - the \$ is used to separate vertices (so you know when the encoding of one vertex ends and the next begins
- Then convert the string to a string in $\{0,1\}^*$.
- Let L_H consists of all graphs with Hamiltonian paths under this encoding
- Problem: decide if a string x is in L_H

The problem of deciding whether a graph has a Hamiltonian path is equivalent to the problem of deciding which strings in $\{0,1\}^*$ represent graphs with Hamiltonian paths (under this encoding).

Another example: the 0-1 knapsack.

- Now the input consists of
  - an upper bound on the weight W that the knapsack can handle
  - a value V (you want to know if you can carry more than \$V worth of items)
  - a sequence of (weight, value) pairs (object i has weight w_i and value v_i
- Can encode this as a string $W\$V\$w_1\$v_1\$\ldots\$w_n\$v_n$
  - Can assume that $W, V, w_1, \ldots, w_n, v_1, \ldots, v_n$ are written in binary.
- Then convert from $\{0, 1, \$\}^*$ to $\{0,1\}^*$.

Acceptance vs. Decision

Key point: All decision problems can be viewed as deciding if a string is in an appropriate language.

- There may be more than one way of representing a language as a set of strings.
  - Most reasonable representations are polynomially related

Focus on algorithms A that take strings as input and return 0 or 1.

Let $L(A) = \{x : A(x) = 1\}$.

- $L(A)$ is the language *accepted by* A.

A accepts in PTIME if there is a k such that if $A(x) = 1$, then A halts in time $O(|x|^k)$.

- all bets are if $A(x) = 0$.
- $|x|$ is the length of the string x

A decides in PTIME if there is a k if A halts in time $O(|x|^k)$ for all inputs.

L is accepted/decided in PTIME if there is an algorithm A such that $L(A) = L$ and A accepts/decides in PTIME.

Theorem: L is accepted in in PTIME iff L is decided in PTIME.

Proof: Clearly if L is decided in PTIME then it is accepted in PTIME.

Conversely, suppose A accepts L in PTIME. That means that if A accepts x, it does so in time $c|x|^k$ for some c. Construct A' as follows:

- Run A, and keep track of the number of steps you've run.
- if, on input x, A has run for $c|x|^k$ and hasn't halted, output 0.

Given A and c, k, can construct A'. But the proof doesn't require that you are given A, c, and k. It suffices to know that they exist.

PTIME verification

Deciding if a graph has a Hamiltonian path seems hard.

Verifying that a given sequence of nodes is a Hamiltonian path is easy.

- NP consists of those languages that can be verified easily.
- It's much easier to check that a proof is correct than to generate a proof.

Formally, a two-argument algorithm $A(x, y)$ *verifies* L if $L = \{x :$ exists y such that $A(x, y) = 1\}$.

- A verifies L if for each $x \in L$, there is a y that A can use to prove that $x \in L$.

Examples:

- In the case of Hamiltonian paths, x is an encoding of the graph, and y as a Hamiltonian path.
  - it's easy to verify that y is/is not a Hamiltonian path for x
- In the case of the 0-1 knapsack problem, x is an encoding of a knapsack problem and y is a choice of items.

A language L is in NP (*nondeterministic polynomial time*) if there is a two-input polynomial time algorithm A and a constant c such that
$L = \{x :$ there exists y such that $|y| = O(|x|^c$ and $A(x, y) = 1\}$

- the certificate for x has to be polynomial in the size of x

Examples of problems in NP:

- Hamiltonian path/Hamiltonian cycle
- 0-1 knapsack
- ...