Part 7: Greedy Algorithms

Contents

[document finalized]

Greedy Algorithms

Lecture 28

> Greed, for lack of a better word,
> is good! Greed is right! Greed works!
> (Gordon Gekko in "Wall Street", 1987)

In this section we will study problems in which greedy decisions lead to optimal solutions.

As a first example consider the **coin-change** problem:

Give change for a specific amount n with the least number of coins of denominations $d_1 < d_2 < \ldots < d_m$.

For example: $d_1 = 1, d_2 = 5, d_3 = 10, d_4 = 25$ cents

How to change 48 cents using such coins?

One obvious solution is $25 + 10 + 10 + 1 + 1 + 1 = 48$.

We could have started with a smaller coin, but our "greedy" thinking leads to choosing a quarter first because it reduces the remaining amount the most.

It can be proved that for above denominations the greedy solution is in fact optimal, but this is not always true.

Leave out nickels, for instance, and try to change 30 cents.

The greedy approach suggests $25 + 1 + 1 + 1 + 1 + 1$, whereas $10 + 10 + 10$ is better.

There are general cases for which the greedy approach computes optimal solutions, such as

$$(1, c, c^2, \ldots, c^{k-1}, c^k), k \geq 0, c \geq 2$$

and it is known that smallest counter examples x for the optimality of the greedy algorithm for

$$(1, d_2, d_3, \ldots, d_{m-1}, d_m)$$

lie in range

$$d_3 + 1 < x \leq d_{m-1} + d_m$$

(Kozen and Zaks, 1994)

In general, the greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached.

At each step the choice made must be

- feasible, i.e. it has to satisfy the problem's constraints
- locally optimal, i.e. it has to be the best local choice among all feasible choices available on that step
- irrevocable, i.e. once made, it cannot be undone.

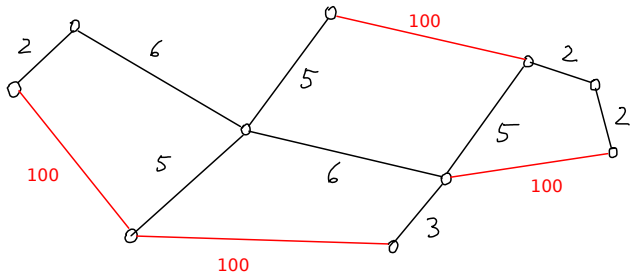
Greedy algorithms are both intuitively appealing and simple. The underlying theory is based on combinatorial structures called matroids, which we don't have time to study here (see for instance CLRS)

Minimum Spanning Trees

"Given n points, find the cheapest possible way to connect them."

Points $\sim$ vertices in graph

Connections $\sim$ weighted edges



Definitions:

A **spanning tree** of a finite connected graph $G = (V, E)$ is a tree $T = (V, E_T)$ with $E_T \subseteq E$.

A **minimum spanning tree** (MST) of a finite weighted graph $G = (V, E, w)$ is a spanning tree of G with minimum weight. I.e.

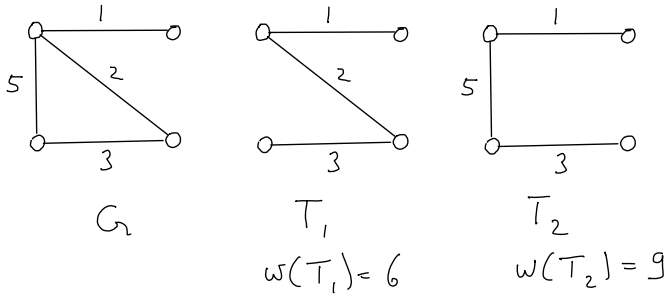
$$T = (V, E_T) \text{ is MST of } G$$

$$\Leftrightarrow$$

T is a spanning tree of G and $w(T) := \sum_{e \in E_T} w(e)$ is minimal over all spanning trees of G.

Note that minimum spanning trees are well defined because the number of subgraphs of finite graphs is finite.

Example

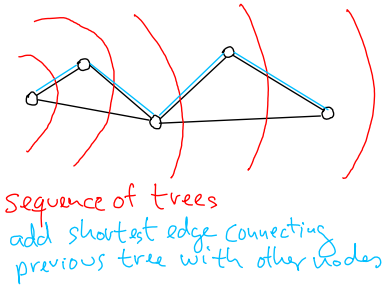


T_1 is the only MST of G

It seems to be the result of growing the tree greedily by choosing edges with small weights first.

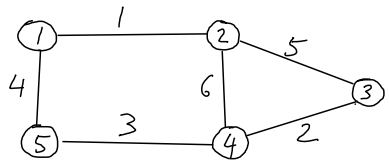
Indeed, MSTs can be computed by greedy algorithms: Prim-Jarník's (1957,1930) and Kruskal's (1956).

Prim-Jarník's Idea:




```
// input:  connected finite graph G = (V, E)
// output:  MST of G
function PrimJarnik((V = {v_1 ... v_n}, E))
V_T = {v_1}
E_T = ∅
for i ← 1 to n − 1 do
   find minimum-weight edge e* = {v*, u*}
     among all edges {v, u} ∈ E so that
     v ∈ V_T and u ∈ V − V_T
   V_T ← V_T ∪ {u*}
   E_T ← E_T ∪ {e*}
end
return (V_T, E_T)
```


Example



V_T	$V - V_T$	T	min. dist. V_T to $V - V_T$
$\{1\}$	$\{2,3,4,5\}$	1	$\{2,1\}$ $w = 1$
$\{1,2\}$	$\{3,4,5\}$	1–2	$\{5,1\}$ $w = 4$
$\{1,2,5\}$	$\{3,4\}$	1–2, 1–5	$\{4,5\}$ $w = 3$
$\{1,2,4,5\}$	$\{3\}$	1–2, 1–5, 5–4	$\{3,4\}$ $w = 2$
$\{1,2,3,4,5\}$	$\{\}$	1–2, 1–5, 5–4, 4–3	$w(T) = 10$

Theorem: Prim-Jarník's algorithm is correct.

Proof:

Claim: Each of the trees $T^{(i)} = (V_T^{(i)}, E_T^{(i)})$ for $i = 0, \ldots, n-1$ generated by the algorithm is a subgraph of some MST of G.

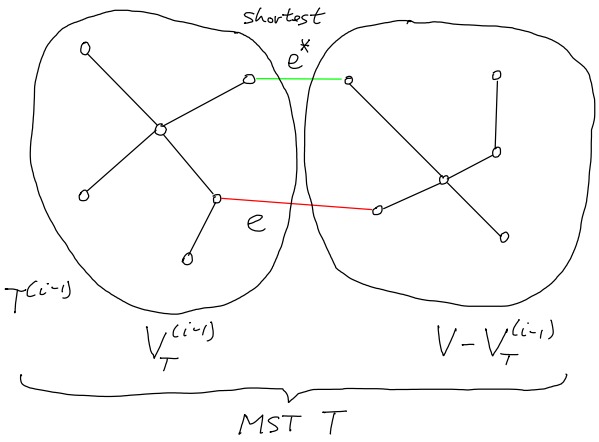
This implies that $T^{(n-1)}$ is an MST of G because it contains all vertices of G.

Induction Base $i = 0$:

$T^{(0)} = (\{v_1\}, \emptyset)$ has no edges, so the claim is trivially true.

Induction Step $i - 1 \rightsquigarrow i$:

Suppose $T^{(i-1)}$ is a subtree of some MST T of G in which e with $w(e) \geq w(e^*)$ connects $V_T^{(i-1)}$ with $V - V_T^{(i-1)}$.



Then $T' = T - e + e^*$ is also an MST of G because

$$w(T') = w(T) - w(e) + w(e^*) \leq w(T).$$

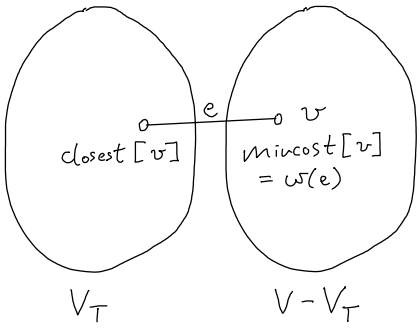
Thus, e^* is part of an MST, and $T^{(i)}$ is a subtree of an MST. □

What is the runtime of Prim-Jarník's algorithm?

This depends on the chosen data structures.

For example, assuming nodes are named $1 \ldots n$ and weights are stored in a matrix, consider three arrays:

`closest[`v`]`: vertex in V_T closest to v
`mincost[`v`]`: weight of edge connecting v with V_T
`intree[`v`]`: true iff v is in V_T



Initialization: Time $O(|V|)$

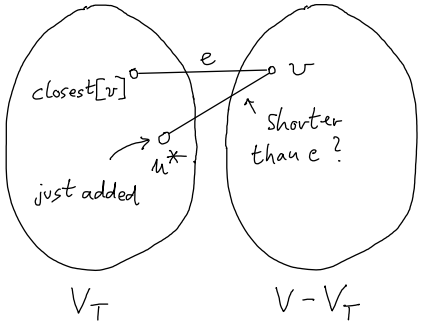
```
closest[2,...,n] ← 1
mincost[i] ← w({1,i}) for all i ∈ V
intree[2..n] ← false; intree[1] ← true
```


Find minimum edge: Time $O(|V|)$

```
find u* in V - V_T with
        smallest mincost value
edge to add:  e* = {closest[u*], u*}
```

Update: Time $O(|V|)$

```
for each v ∈ V - V_T do
   if mincost[v] > w({u*, v}) then
      mincost[v] ← w({u*, v})
      closest[v] ← u*
   end
end
intree[u*] ← true
```



Lecture 29

Total time: $O(|V|^2)$

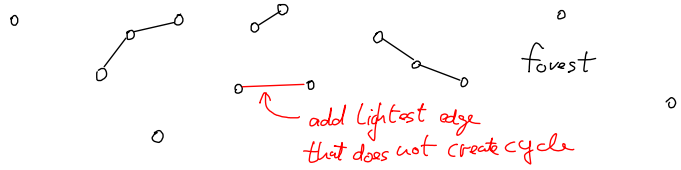
This is optimal for dense graphs ($|E| \in \Theta(|V|^2)$)

For sparse graphs ($|E| \in O(|V|)$) we can do better:

Assuming that the graph is represented by adjancy lists and we store the current distances from nodes to V_T in a min-heap, it is not hard to see that the runtime of the Prim-Jarník algorithm using these data structures is

$$O(|E| \log |V|)$$

There is a second greedy approach available for computing MSTs: Kruskal's algorithm, which grows forests by adding lightest edges that do not create a cycle.

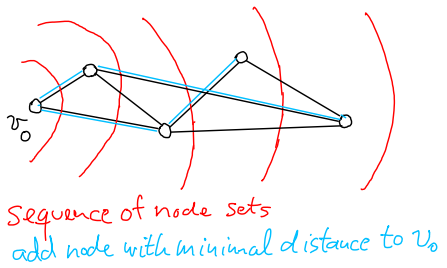


Runtime: $O(|E| \log |E|)$

Single-Source Shortest Paths

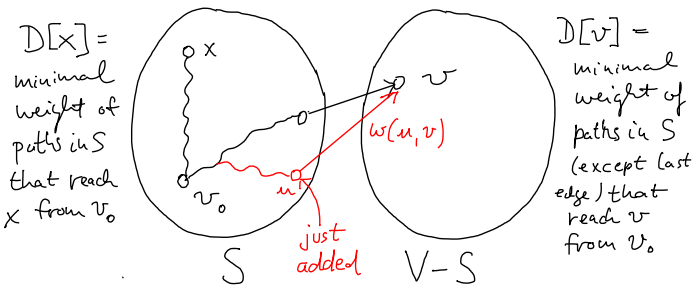
Given a directed weighted graph $G = (V, E, w)$ with non-negative weights and a node v_0, we want to find minimal weights of paths from v_0 to all other nodes.

Dijkstra's Idea (1959):



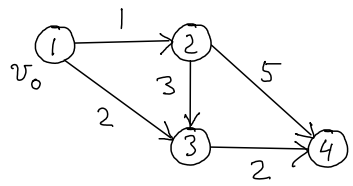
Starting with v_0 grow node set S by adding nodes $v \in V - S$ maintaining

- the minimal weights of paths from v_0 to $v \in S$ that stay in S, and
- the minimal weights of paths from v_0 to $v \in V - S$ that stay in S except for the last edge.




```
// input:  directed weighted graph, v_0
//         weights ≥ 0,  w(i,i) = 0 for all i ∈ V
// output:  array of shortest distances from v_0
function Dijkstra((V = {1...n}, E, w), v_0)
S ← {v_0}
for each v ∈ V do D[v] ← w(v_0, v) end
while S ≠ V do
   choose u ∈ V − S such that D[u] minimal
   S ← S ∪ {u}
   for all v ∈ V − S do
      D[v] ← min{D[v], D[u] + w(u, v)}
   end
end
return D
```


Example



S	D[1]	D[2]	D[3]	D[4]
{1}	0	**1**	2	∞
{1,2}	0	1	**2**	6 = 1+5 < ∞
{1,2,3}	0	1	2	**4** = 2+2 < 6
{1,2,3,4}	0	1	2	4

Lecture 30

Claim: Dijkstra's algorithm computes the minimal weight of paths from v_0 to any other node in V in time $O(|V|^2)$, if arrays are used.

Proof: The algorithm terminates because in each iteration S grows by one element. So $S = V$ after $|V| - 1$ iterations.

Each iteration takes $O(|V|)$ time, for a total runtime of $O(|V|^2)$.

To prove that the output is correct, we show the following statements by induction: after the k-th iteration

i) for every node $v \in S$, $D[v]$ is the minimal weight of paths from v_0 to v which we call $\delta(v_0, v)$, and

ii) for every node $v \notin S$, $D[v]$ is the minimal weight of paths from v_0 to v that contain only (besides v) vertices in S.

Induction Base $k = 0$: the initialization of $S = \{v_0\}$ and D ensures that i) and ii) hold before the first iteration.

Induction Step $k \rightsquigarrow k + 1$:

Assume the induction hypothesis holds prior to the $(k+1)$-st iteration.

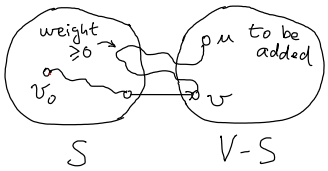
To prove i), let u be the vertex added to S at the $(k+1)$-st iteration.

So, $u \notin S$ and $D[u]$ is the smallest D value of any vertex not in S at the end of the k-th iteration.

From the induction hypothesis: vertices $v \in S$ before the $(k+1)$-st iteration have $D[v] = \delta(v_0, v)$.

Also, $D[u] = \delta(v_0, u)$: by the induction hypothesis (part ii) we know that $D[u]$ is the minimum weight of paths from v_0 to u with intermediate nodes in S.

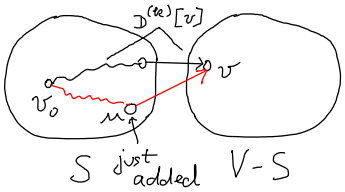
If there was a shorter path P from v_0 to u, consider the first vertex v outside of S along P. Then $D[v] < D[u]$ (because weights are ≥ 0), which contradicts the choice of u.



Thus i) holds at the end of the $(k+1)$-st iteration.

To prove ii), let u again be the vertex added to S and let v be a vertex not in S after the $(k+1)$-st iteration.

A shortest path from v_0 to v inside S (except for v) either contains u or it does not.



If it does not contain u, then by the induction hypothesis its weight is $D^{(k)}[v]$ – the previous iteration's value.

If it does contain u, then it must be made up of a path from v_0 to u of shortest possible weight inside S followed by edge (u, v).

In this case the path's weight is $D^{(k)}[u] + w(u, v)$ by the induction hypothesis part i).

This proves ii), because

$$D^{(k+1)}[v] = \min\{D^{(k)}[v], D^{(k)}[u] + w(u, v)\}$$

according to the algorithm.

Thus, the claim holds for all k, and when $S = V$, Dijkstra's algorithm has computed all $D[v]$ correctly $\square$

Using a min-heap and adjacency lists for representing the graph leads to runtime

$$O(|E| \log |V|)$$

which is better for sparse graphs (exercise).

Huffman Codes

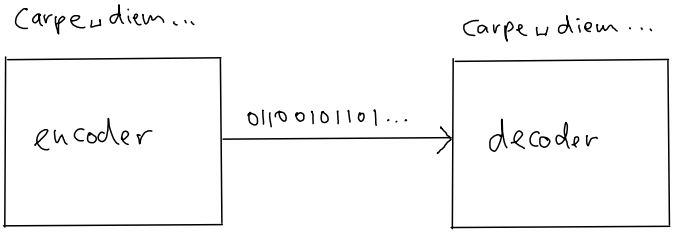
Suppose you want to send text via a digital channel (such as a computer network).

We are looking for mapping (also called code)

$$C : \text{Words} \to \text{0-1-Words}$$

such that

- transmitted words can be decoded uniquely
- transmitted words are short



Simplifying assumption:

C maps individual characters (also called symbols) to 0-1 sequences

I.e.

$$C(a_1 a_2 \cdots a_n) = C(a_1)C(a_2)\cdots C(a_n)$$

(By grouping symbols, the compression rate can be increased)

Example:

Encode words over alphabet $\{a, b, c\}$ using the following code C:

$$\begin{aligned} C(a) &= 0 \\ C(b) &= 00 \\ C(c) &= 1 \end{aligned}$$

Can we decode transmitted code words uniquely?

No:

$C(aa) = C(a)C(a) = 00$

$C(b) = 00$

When seeing 00 the decoder can't decide whether the original message was aa or b

How to construct uniquely decodable codes?

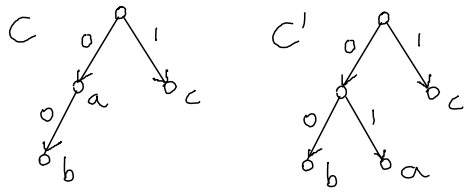
The problem with the previous code is that

$C(a) = 0$ is a **prefix** of $C(b) = 00$

So when seeing 0 there are two possible choices.

In **prefix codes** no code word is a prefix of any other code word

0-1 codes can be represented as binary trees:



Path to symbol corresponds to code word

$C(a)$ is prefix of $C(b)$, b is a node in a subtree rooted in a

Now consider code C':

$$\begin{aligned} C'(a) &= 01 \\ C'(b) &= 00 \\ C'(c) &= 1 \end{aligned}$$

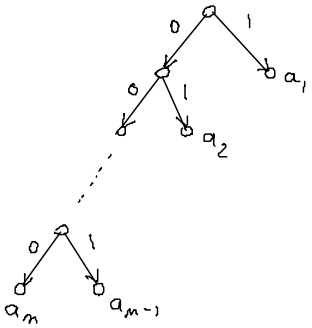
In the tree corresponding to C' all symbol nodes are leaves $\Rightarrow C'$ is prefix code

Prefix codes are easy to decode:

1. Start at root
2. read bits one by one, follow path
3. print symbol when reaching leaf
4. go to 1

Given an alphabet $\{a_1, \ldots, a_n\}$ we are looking for prefix code C that maps a_i to 0-1 word $C(a_i)$.

Easy:



But: code words can be long (here $n-1$)

How to tell good from bad codes?

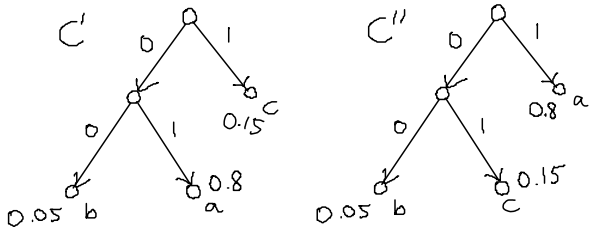
Suppose every symbol a_i has a corresponding probability p_i to appear next in the message.

When transmitting a source text using code C, we are interested in minimizing its expected code length:

$$L(C) = \sum_{i=1}^{n} p_i \cdot \text{length}(w_i)$$

So, if p_i is high, then $w_i = C(a_i)$ should be short

Examples:



$$L(C') = 0.8 \cdot 2 + 0.05 \cdot 2 + 0.15 \cdot 1 = 1.85,$$

i.e. on average, 1.85 bits are sent for each source symbol

$$L(C'') = 0.8 \cdot 1 + 0.05 \cdot 2 + 0.15 \cdot 2 = 1.2 \ !$$

Lecture 31 How to construct codes with minimal expected code length?

Huffman's idea (1952): Perhaps greed works.

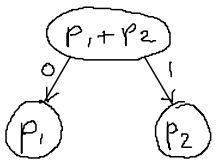
Huffman Algorithm

Input: symbols $1, \ldots, n$ with probabilities $p_1, \ldots, p_n$

Output: a binary tree corresponding to a code with minimal expected code length

Build tree recursively:

$n = 2$: return tree T^*:



$n > 2$:

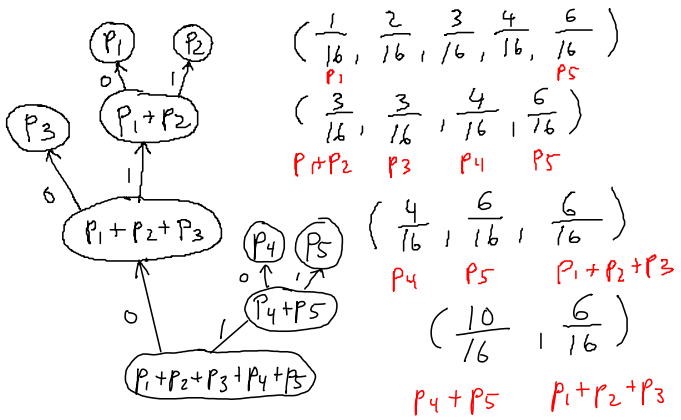
i) Sort p_i's in non-decreasing order, say

$$p_1 \le p_2 \le \ldots \le p_n$$

ii) Construct tree T for $(p_1+p_2, p_3, \ldots, p_n)$ recursively

iii) return tree that results from replacing the (p_1+p_2) leaf in T by tree T^*

Example



Define the **weighted path length** of binary tree T with leaves $1, \ldots, n$ and corresponding probabilites $p_1, \ldots, p_n$ as

$$L(T) := \sum_{i=1}^{n} p_i \cdot \text{depth}(i)$$

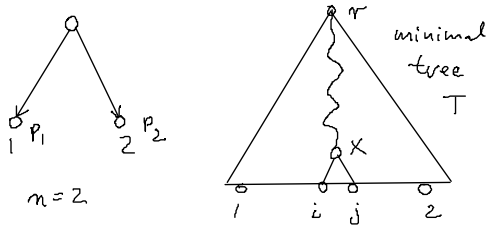
$\sim$ expected code length

Theorem: A tree the Huffman algorithm generates for input $p_1 \leq p_2 \leq \cdots \leq p_n$ has minimal weighted path length among all full binary trees (out-degree 0 or 2) with leaf probabilities $p_1, \ldots, p_n$

Proof: Induction on n

$\underline{n = 2:}$

There are only two full binary trees with two leaves:



Both have weighted path length $p_1 \cdot 1 + p_2 \cdot 1 = p_1 + p_2$

$\underline{< n \rightarrow n:}$

Suppose $n \geq 3$ and the claim is true for smaller n.

Number of full binary trees with n leaves finite $\Rightarrow$ $\exists$ tree T for which $L(T)$ is minimal.

Consider internal node x of T with maximal distance to root r

If nodes $1, 2$ are not the children i, j of x, then exchange nodes $1, 2$ with them.

Because T is minimal, nodes $1, 2$ must have had the same depth as nodes i, j (otherwise, exchanging nodes would decrease the weighted path length)

$\Rightarrow$ there exists a minimal tree with nodes $1, 2$ being the children of an internal node with maximum depth.

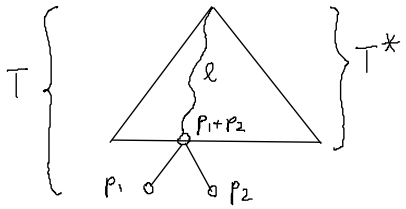
Now, consider a tree T^* the Huffman algorithm generates for the updated symbol frequencies created from the original by merging nodes 1 and 2.

By the induction hypothesis T^* is minimal.

Now, expand the leaf node corresponding to $p_1 + p_2$ by adding the original leaves 1 and 2. Call the resulting tree T.

We need to show that T is minimal.

Consider the weighted path lengths of both trees:



$$L(T) = L(T^*) - l \cdot (p_1 + p_2) + (l + 1) \cdot (p_1 + p_2)$$
$$= L(T^*) + p_1 + p_2$$

So, if T^* is minimal (as asserted by the induction hypothesis), then T is also minimal, because:

If there was a tree $\bar{T}$ with $L(\bar{T}) < L(T)$ with nodes $1, 2$ on the last level, then by merging $1, 2$ as before we can create a tree $\bar{T}^*$ from it with

$$L(\bar{T}^*) + p_1 + p_2 = L(\bar{T}) < L(T) = L(T^*) + p_1 + p_2$$

I.e., $L(\bar{T}^*) < L(T^*)$, which contradicts the induction hypothesis. $\square$

Claim: The Huffman algorithm can be implemented such that it runs in time $O(n \log n)$ given an unsorted probability sequence $(p_1, \ldots, p_n)$.

Proof: Exercise.

Theorem: If $w_1 \ldots w_n$ are the binary codes assigned by the Huffman's algorithm to symbols with probabilities $p_1 \ldots p_n$. Then

$$p_i < p_j \;\Rightarrow\; \text{length}(w_i) \geq \text{length}(w_j)$$

Proof: Exercise.

How long can Huffman code words get for n symbols?

Maximum is $n - 1$.

A better upper bound is given in the following theorem:

Theorem (Buro 1993):
Given $0 < p_1 \leq p_2 \leq \cdots \leq p_n$ no Huffman code word is longer than

$$\min\{\lfloor \log_\Phi(1/p_1) \rfloor, n-1\},$$

where $\Phi = \frac{1+\sqrt{5}}{2} \doteq 1.618$

Proof: Based on Fibonacci numbers.

Example: $p_1 = \frac{1}{3500} \Rightarrow |w_i| \leq 16$

Huffman codes are used in many compression tools such as

gzip, bzip2, lzw, zlib