Efficient Algorithms and Problem Complexity

– More Greed –

Frank Drewes
Department of Computing Science
Umeå University
Today’s Menu

1. Example: Dijkstra’s Algorithm

2. Huffman Codes
Some terminology

Let \( G = (V, E, w) \) be a weighted directed graph (with non-negative weights and, for simplicity, no parallel edges).

- A path is a sequence \( p = v_0 \cdots v_k \) of nodes such that \( k \geq 0 \) and \( (v_{i-1}, v_i) \in E \) for \( i = 1, \ldots, k \).
- Such a path is also said to be a path from \( v_0 \) to \( v_k \).
- The length of the path is its weight \( w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \) (similar to the weight of a spanning tree).
- The distance \( \delta(v, v') \) between nodes \( v, v' \) is the length of a shortest path between them.
The Single-Source Shortest Paths Problem

**Input:** A weighted graph \( G = (V, E, w) \) as above, and a node \( v_0 \in V \).

**Task:** Determine \( \delta(v_0, v) \) for every \( v \in V \).

**Note:** If we know \( \delta(v_0, v) \) for every \( v \in V \), we read off the actual shortest path from \( v_0 \) to \( v \). [HOW?]

\[ \Rightarrow \] we can consider the corresponding problems to be “the same”.

**Question:** What is a suitable graph structure for the description of a collection of shortest paths?
How can we determine shortest paths?

**Theorem: Extending Shortest Paths**

Let $G = (V, E, w)$, $v_0 \in V$, and $\delta_v = \delta(v_0, v)$ for all $v \in V$. Let $\{v_0\} \subseteq V' \subseteq V$ be such that $\delta_u \leq \delta_v$ for all $u \in V'$, $v \in V \setminus V'$. For all $v \in V \setminus V'$, if

$$dist_v = \min\{\delta_u + w(u, v) \mid (u, v) \in E\},$$

then $dist_v = \delta_v$ for all $v \in V'$ for which $dist_v$ is minimal.

**Note 1:** By convention, $\min\emptyset = \infty$.

**Note 2:** This is extremely similar to the property exploited by Prim’s algorithm, which basically puts

$$dist_v = \min\{w(u, v) \mid (u, v) \in E\}.$$
Dijkstra’s algorithm, rough description

For Dijkstra’s algorithm, $G$ is represented by adjacency lists.

1. Put $G′ = (V′, E′) = (\{v_0\}, \emptyset)$ and $\delta(v_0) = 0$.

2. Greedy rule: While $|V′| < |V|$, choose $e = (u, v) \in E$ such that
   - $u \in V′$ and $v \in V \setminus V′$ and
   - $\delta(u) + w(u, v)$ is minimal among all edges with this property.

3. Add $v$ to $V′$ and $e$ to $E′$, set $\delta(v) = \delta(u) + w(u, v)$, and repeat.

4. Return $(V′, E′)$.

Similar to Prim’s algorithm, the detailed implementation keeps track of
$\min \{\delta(u) + w(u, v) \mid u \in V′ \text{ and } (u, v) \in E\}$ for all $u \in V \setminus V′$.

Each time a new $u$ is added to $V′$, the numbers are updated.
From Prim to Dijkstra

Prim(adj) where \( V = \{1, \ldots, n\} \) and \( v_0 = 1 \)
initialize minheap \( h \) with \( \text{key}(1) = 0 \), \( \text{key}(u) = \infty \) for \( u > 1 \)
\( \text{parent}[1] \leftarrow \bot \)
while \( h \) not empty do
  \( u \leftarrow h.\text{del}() \)  // retrieve \( u \) with smallest key
  for \( (v, \text{weight}) \) in \( \text{adj}[u] \) do
    if \( v \) is in \( h \) \&\& \( \text{weight} < h.\text{keyval}(v) \) then
      \( h.\text{decrease}(v, \text{weight}) \)
      \( \text{parent}(v) \leftarrow u \)
  return \( \text{parent} \)

Note: You may also want to rename \( \text{parent} \) to, e.g., \( \text{predecessor} \) ...

Question: What happens if the graph is disconnected?
Running Time of Dijkstra’s Algorithm

Clearly, the running time of Dijkstra’s Algorithm is the same as that of Prim’s. Thus, it runs in time $O(m \log n)$, where $m$ is the number of edges in the input graph and $n$ is the number of nodes.
Prefix codes

Let $\Sigma$ be an alphabet, $|\Sigma| \geq 2$, and consider a mapping $\kappa: \Sigma \to \{0, 1\}^*$.

- The extension of $\kappa$ to $\Sigma^*$ is
  \[
  \kappa^*(\epsilon) = \epsilon \quad \text{and} \quad \kappa^*(aw) = \kappa(a)\kappa^*(w) \quad \text{for} \quad a \in \Sigma, w \in \Sigma^*.\n  \]
  (In other words, $\kappa^*(a_1 \cdots a_n) = \kappa(a_1) \cdots \kappa(a_n)$. Such mappings are called homomorphisms.)

- $\kappa$ is a code if $\kappa^*$ is injective (like ASCII).

- $\kappa$ is a prefix code if no $\kappa(a)$ is a prefix of any $\kappa(b)$, for all $a, b \in \Sigma$.

- Thus, unlike ASCII, prefix codes can be variable length codes.
Questions:

- Is a prefix code a code?
- Is ASCII a prefix code?
- Why is the prefix property particularly useful for coding?
A prefix code can be represented as a binary coding tree whose leaves are the symbols in $\Sigma$.

Example:

<table>
<thead>
<tr>
<th>000</th>
<th>01</th>
<th>10</th>
<th>11</th>
<th>0011</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$b$</td>
<td>$c$</td>
<td>$d$</td>
<td>$e$</td>
</tr>
</tbody>
</table>

A Huffman code is a prefix code whose coding tree is a full binary tree (just contract edges to children that lack a sibling).
Optimal Huffman codes

Suppose we are given a frequency for the letters in $\Sigma$, $f : \Sigma \rightarrow \mathbb{N}$. We want the code to minimize the length of text encodings.

$\Rightarrow$ We need to minimize $wpl(\kappa) = \sum_{a \in \Sigma} f(a)|\kappa(a)|$. (*)

A Huffman code that does this is called optimal.

(*) $wpl =$ weighted path length
Huffman frequency trees

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>01</th>
<th>10</th>
<th>11</th>
<th>001</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

coding tree

frequency tree $T$

$$wpl(T) = wpl(\kappa) = 37$$
Properties of optimal Huffman coding trees

Theorem

1. There always exists an optimal Huffman frequency tree (HFT) in which two minimal frequencies have the same parent.
2. Deleting two sibling leaves $f_1, f_2$ in a HFT $T$ for $f_1, \ldots, f_n$ yields a HFT $S$ for $f_1 + f_2, f_3, \ldots, f_n$ with $\text{wpl}(T) = \text{wpl}(S) + f_1 + f_2$.

Proof sketch:

1. Rearranging leaves at the same level does not affect $\text{wpl}(T)$. If $T$ is optimal, the frequencies at the lowest level are minimal in $T$. (Otherwise, interchanging two frequencies would decrease $\text{wpl}(T)$.) $\Rightarrow$ leaves at the lowest level can be reordered, so that the two smallest frequencies $f_1$ and $f_2$ become siblings.
2. Immediate from the definition.
Huffman’s greedy rule

Replace the two smallest frequencies \(f_1, f_2\) with \(f_1 + f_2\). Compute the optimal Huffman coding tree for these \(n - 1\) frequencies, then add \(f_1\) and \(f_2\) as children to \(f_1 + f_2\).

**Correctness** (by induction on the number of frequencies \(f_1, \ldots, f_n\)):

- For \(n = 2\), the two possible HFTs \(T, T'\) satisfy \(\text{wpl}(T) = \text{wpl}(T')\).
- For \(n > 2\), assume that \(f_1, f_2\) are minimal. Let \(T\) be the output HFT, and let \(T'\) be \(T\) without \(f_1, f_2\) (i.e., the result obtained recursively). By I.H., \(T'\) is optimal. Now, if \(S\) is an optimal HFT in which \(f_1, f_2\) are siblings, and \(S'\) is \(S\) with \(f_1, f_2\) deleted, then

\[
\text{wpl}(T) = \text{wpl}(T') + f_1 + f_2 \leq \text{wpl}(S') + f_1 + f_2 = \text{wpl}(S),
\]

so \(T\) is optimal.
Huffman’s greedy rule, reformulated

Let \( \{f_1, \ldots, f_n\} \) be the given frequencies, with \( f_1, f_2 \) being smallest. Create a frequency \( f_1 + f_2 \) and make it the parent frequency of \( f_1 \) and \( f_2 \). If \( n > 2 \), repeat with \( \{f_1 + f_2, f_3, \ldots, f_n\} \).
Huffman Codes

Implementation of Huffman’s algorithm

\[
Huffman(a) \text{ where } a[1 \ldots, n] \text{ is an array of frequencies}
\]

initialize minheap \( h \) to new node\((a[1])\), \ldots, new node\((a[n])\)

for \( i = 1, \ldots, n - 1 \) do

\[
\begin{align*}
& v_1 \leftarrow h.\text{del()} \quad // \text{retrieve nodes with} \\
& v_2 \leftarrow h.\text{del()} \quad // \text{smallest frequencies} \\
& v \leftarrow \text{new node}(v_1.\text{frequency} + v_2.\text{frequency}) \\
& v.\text{setChildren}(v_1, v_2) \\
& h.\text{insert}(v) \\
\end{align*}
\]

return \( h.\text{del()} \)

Initialization takes time \( O(n) \), deletion/insertion take time \( O(\log n) \) ⇒ \( O(n \log n) \) steps in total.
Until the next lecture, please read the remainder of the Chapter on greedy algorithms.