NEGATIVE COSTS

The given figure shows a graph with vertices labeled as A, B, C, and D. The edges and their respective costs are as follows:

- A to B: 3
- B to C: 3
- C to D: -7
- D to C: 5
- A to D: -10

The text explains:

- The path from A to C via B does not make sense, as it leads to $-\infty$.
- It is a loop with a negative cost.

The text also states:

- The path from A to C via D makes sense.
- There is no loop with a negative cost.

The shortest distance from A to C is calculated as:

$$3 + 5 - 7 = 1$$
Dijkstra (greedy) algorithm does not work

Step 1.
\[ s = \{ a \} \quad V - s = \{ b, c, d \} \]

Step 2.
\[ s = \{ a, b \} \quad V - s = \{ c, d \} \]

Step 3.
\[ s = \{ a, b, c \} \quad \text{no change} \quad V - s = \{ d \} \]

Step 4.
\[ s = \{ a, b, c, d \} \quad \text{no change} \quad V - s = \emptyset \]
Some notation often used.

- **INITIALIZE** \((s)\)
  
  for each vertex \(v \in V\) do
  
  \[
  \text{begin} \quad d[v] \leftarrow \infty; \\
  \quad p[v] \leftarrow \text{nil}
  \]
  
  \[
  \text{end}
  \]

  \[
  d[s] \leftarrow 0
  \]

- **RELAX** \((u, v)\)
  
  if \(d[v] > d[u] + \omega(u, v)\)
  
  then \[
  \text{begin} \quad d[v] \leftarrow d[u] + \omega(u, v); \\
  \quad p[v] \leftarrow u
  \]
  
  \[
  \text{end}
  \]

**ALTERNATIVE START**

\[
\text{begin} \quad d[v] \leftarrow \omega(s, v); \\
\quad p[v] \leftarrow s
\]

\[
\omega(, ) \quad \text{the incidence matrix}
\]

**HISTORICAL NAME**
IN THIS NOTATION:

DIJKSTRA (s)

INITIALIZE (s);
$S \leftarrow \emptyset$;
Queue $\leftarrow V$;

while Queue $\neq \emptyset$ do

begin
    $u \leftarrow$ MINIMUM (Queue);
    $S \leftarrow S \cup \{u\}$; \text{ \texttt{INSERT(S,u)}}

    for each vertex $v$ adjacent to $u$
    do RELAX ($u,v$)

end
Bellman-Ford algorithm

\[ |V| = n \]

**BELLMAN-FORD (s)**

1. \text{INITIALIZE}(s);
2. \text{for } i = 1 \text{ to } n-1 \\
3. \text{do for each } (u,v) \in E \text{ do } \text{RELAX}(u,v); \\
4. \text{for each } (u,v) \in E \text{ do } \\
5. \quad \text{if } d[v] > d[u] + w(u,v) \text{ then return (False);} \\
6. \text{return (True)}

Complexity
1. is \( O(n) \)
2. is \( O(1E) \) or \( O(1E1) \)
3. Overall \( O(|V|1E1) \)

Slower than DIJKSTRA but works for negative costs!
AND “TRUE” IS RETURNED
- Path recovery: Identical to Dijkstra!!

**Special case:** The graph is acyclic!

- Now we can do it in \( \Theta(IV + IE) \), so better than Dijkstra and Bellman-Ford.

**DAG** - Directed **Acyclic Graph**

\[
\text{DAG-Peak-Path} (s) \\
\text{topologically sort the vertices of } G \\
\text{INITIALIZE} (s) \\
\{ \\
\text{for each } u \in V, \text{ taken in topologically sorted order} \\
\text{do for each } v \text{ on the adjacency list of } u \\
\text{do RELAX}(u, v) \\
\}\]

Complexity: \( \Theta(IV + IE) \) time
**ALL PAIRS SHORTEST PROBLEM**

- For each pair of nodes $u, v$ find the shortest path from $u$ to $v$.

- **No negative costs**: Apply Dijkstra's algorithm $n$ times ($n = |V|$), $m = |E|$.
  - Complexity: $O(n^3)$ - Adjacency matrix, dense graphs.
  - Complexity: $O(nm \log n)$ - Adjacency list, sparse graphs.

- **Negative costs**: Apply Bellman-Ford algorithm $n$ times.
  - Complexity: $O(n^2m)$.

- Can we do better? Remember, Big O notation forgets about constant overheads!
FLOYD'S ALGORITHM

- IDEA: THE SAME AS FOR DIJKSTRA'S AND BELLMAN-FORD, BUT ALL CASES CONSIDERED, I.E. NO GREED NEEDED!

\[ S_k = \{1, 2, \ldots, k\}, \quad k < n \]

Suppose \( D_{S_k}(i, j) \), a shortest distance from \( i \) to \( j \) using only nodes from \( S_k \), is known.

Let \( S_{kn+1} = \{1, 2, \ldots, k, k+1\} \)

\[ D_{S_{kn+1}}(i, j) = \min \left( D_{S_k}(i, j), D_{S_k}(i, k+1) + D_{S_k}(k+1, j) \right) \]

We compute: \( D_{S_0}(i, j), D_{S_1}(i, j), \ldots, D_{S_n}(i, j) = D(i, j) \)

THE SHORTEST DISTANCE FROM \( i \) TO \( j \)
NO OVERHEAD SOLUTION!

Notation: \( C[i,j] \) - weight at \( (i,j) \)
\( A[i,j] = D_{ik}(i,j) \)
\( P[i,j] = k \) mean \( k \) is on a currently shortest path from \( i \) to \( j \).

FLOYD'S ALGORITHM

Initialize \( A[i,j] = C[i,j] \), \( P[i,j] = 0 \) for \( i,j = 1, \ldots, n \)

For \( k = 1 \) to \( n \) do
  For \( i = 1 \) to \( n \) do
    For \( j = 1 \) to \( n \) do
        begin
          \( P[i,j] = k \);
        end
      End If
    End For
  End For
End For

Complexity: \( O(n^3) \)
Path Recovery:

Shortest Path from $i$ to $j$ is given by the following procedure:

PATH($i,j$)

$k = P[i,j]$ ← $k$ is on the shortest path from $i$ to $j$

If $k \neq 0$ then

begin

PATH($i,k$);
WRITE($k$);
PATH($k,j$);

end

end PATH
**HUFFMAN CODES AND DATA COMPRESSION,**

**ANOTHER POPULAR GREEDY ALGORITHM.**

- **WE HAVE A SET OF CHARACTERS:** \( A = \{a_1, a_2, ..., a_k\} \)
- **A STRING OR MESSAGE:** \( x_1 x_2 ... x_n \) where \( x_i \in A \)
- **FOR EACH** \( a_i \in A \), \( f(a_i) \) **IS THE FREQUENCY OF APPEARANCE** \( a_i \) **IN THE MESSAGE** (or a probability)
  
  \[ \sum_{a_i \in A} f(a_i) = 1 \]

- **ENCODING:** **ASSIGN A BINARY CODE** \( c(a_i) \) **FOR EACH** \( a_i \), **AND EXTEND** \( c \) **TO STRINGS BY**
  
  \[ c(x_1 x_2 ... x_n) = c(x_1) c(x_2) ... c(x_n) \]

- **DECODING:** **GIVEN A CODE** \( b_1 b_2 ... b_m \) **FIND THE UNIQUE MESSAGE** \( x_1 x_2 ... x_n \) **SUCH THAT**

  \[ c(x_1 x_2 ... x_n) = b_1 b_2 ... b_m \]

- **ENCODING LENGTH/AVERAGE CODE LENGTH:**

  \[ \sum_{a_i \in A} f(a_i) \text{ length}(c(a_i)) \]
<table>
<thead>
<tr>
<th>CHARACTER</th>
<th>FREQUENCY</th>
<th>CODE 1</th>
<th>CODE 2</th>
<th>CODE 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>.30</td>
<td>000</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>b</td>
<td>.10</td>
<td>001</td>
<td>0010</td>
<td>00</td>
</tr>
<tr>
<td>c</td>
<td>.10</td>
<td>010</td>
<td>0011</td>
<td>01</td>
</tr>
<tr>
<td>d</td>
<td>.10</td>
<td>011</td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>.40</td>
<td>100</td>
<td>000</td>
<td></td>
</tr>
</tbody>
</table>

**Average Code Length**

<table>
<thead>
<tr>
<th>CHARACTER</th>
<th>CODE 1</th>
<th>CODE 2</th>
<th>CODE 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>3</td>
<td>2.1</td>
<td>1.7</td>
</tr>
</tbody>
</table>

**Prefix Property:** $c(a_i)$ is not a prefix of $c(a_j)$ for any $i \neq j$.

**No Prefix Property:** No Decoding!

Code 1, Code 2 have prefix property.

Code 3 doesn't have prefix property.
BINARY TREE REPRESENTATION OF CODES

CODE 1
- \( c(a) = 000 \)
- \( c(b) = 001 \)
- \( c(c) = 010 \)
- \( c(d) = 011 \)
- \( c(e) = 100 \)

CODE 2
- \( c(a) = 01 \)
- \( c(b) = 0010 \)
- \( c(c) = 0011 \)
- \( c(d) = 0000 \)
- \( c(e) = 1 \)

PREFIX PROPERTY
- ALL LETTERS AS LEAVES

NO PREFIX PROPERTY
- SOME LETTERS AS INTERIOR NODES
**HUFFMAN CODE**: AN OPTIMAL (MINIMAL LENGTH) PREFIX CODE.

**ALGORITHM**: IT FINDS HUFFMAN CODE \( C(a_i) \) FOR EACH \( a_i \).

HUFFMAN \( (\{a_1, a_2, \ldots, a_n\}) \)

Let \( a_i \) and \( a_j \) are such that \( f(a_i) \) and \( f(a_j) \) are the lowest among \( a_i, \ldots, a_n \).

Let \( a' \) be a new character and we set 
\[ f(a') = f(a_i) + f(a_j) \]

Define \( A' = (\{a_1, a_2, \ldots, a_n, a'\} - \{a_i, a_j\}) \cup \{a'\} \)

Call HUFFMAN \( (A') \)

\[ C(a_i) = C(a') 0; \]
\[ C(a_j) = C(a') 1; \]

END HUFFMAN

**Example**: \( \{a, b, c\} \), \( f(a) = 0.5 \), \( f(b) = 0.3 \), \( f(c) = 0.2 \)

HUFFMAN \( (\{a, b, c\}) \Rightarrow f(\{bc\}) = 0.5 \) HUFFMAN \( (\{a, \{bc\}\}) \)

\[ \downarrow \]

\[ C(a) = 0 \]
\[ C(b) = 10 \]
\[ C(c) = 11 \]

THE PROCEDURE IS BETTER UNDERSTOOD IF PRESENTED IN TERMS OF TREES.
**Example**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>0.10</td>
<td>0.20</td>
<td>0.05</td>
<td>0.05</td>
<td>0.10</td>
<td>0.30</td>
<td>0.10</td>
<td>0.10</td>
</tr>
</tbody>
</table>

We start with a forest

(1) 0.10 0.20 0.05 0.05 0.10 0.30 0.10 0.10

a b c d e f g h

(2) 0.10 0.20 0.10 0.05 0.05 0.10 0.30 0.10 0.10

a b c d e f g h

(3) 0.20 0.10 0.10 0.10 0.05 0.05 0.30 0.10 0.10

b e f g h
• Solutions are not unique but the value of \( \sum_{i=1}^{n} f(a_i) \text{ length}(c(a_i)) \) is always the same and it is the minimal value for all binary prefix codes.

• Complexity:
  
  • K-1 iterations, each consists of finding two minimal values and merging, can easily be done in \( O(k) \), so totally \( O(k^2) \).

  • If priority queues implemented as heaps are used, then finding two minimal elements is \( O(\log k) \), merging is \( O(1) \), so totally \( O(k \log k) \).

• Ideas can be extended, see pages 175-177 in textbook.