Greedy Algorithms, Continued

DPV Chapter 5, Part 2

Jim Royer

March 4, 2019

(Unless otherwise credited, all images are from DPV.)

Huffman Encoding, 1

A toy example:

- Suppose our alphabet is $\{ A, B, C, D \}$.
- Suppose T is a text of 130 million characters.
- What is a shortest binary string representing T? *(A hard question.)*

Encoding 1

$A \mapsto 00$, $B \mapsto 01$, $C \mapsto 10$, $D \mapsto 11$. Total: 260 megabits.

Statistics on T

Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

Idea: Use variable length codes
A's code $\ll$ D's code $\ll$ B's code

Encoding 2

$A \mapsto 0$, $B \mapsto 100$, $C \mapsto 101$, $D \mapsto 11$.
Total: 213 megabits — 17% better.

Q: How to unambiguously decode?
Q: How to come up with the code?
Q: How good is the result?

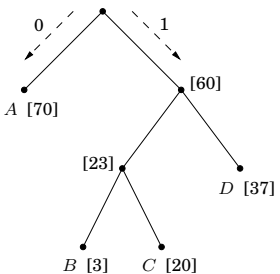
Huffman Encoding, 2

Definition

A ***prefix-free*** code is a code in which no codeword is the prefix of another.

Prefix-free codes can be represented by *full* binary trees
(i.e., trees in which each non-leaf node has two children).
Example:

Symbol	Codeword
A	0
B	100
C	101
D	11



Question: How do you use such a tree to decode a file? *Sample:* 01101001010

Huffman Encoding, 3

Goal: Find an optimal coding tree for the frequencies given.

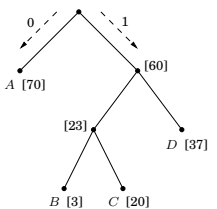
$$\text{cost of a tree} = \sum_{i=1}^{n} f[i] \cdot (\text{depth of the } i\text{th symbol in tree})$$
$$= \sum_{i=1}^{n} f[i] \cdot (\text{\# of bits required for the } i\text{th symbol})$$

Assigning frequencies to all tree nodes

(a) Leaf nodes get the frequency of their character.

(b) Internal nodes get the sum of the freqs of the leaf nodes below them.

Symbol	Codeword
A	0
B	100
C	101
D	11



Huffman Encoding, 4

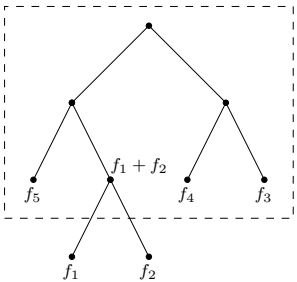
Observation

In an optimal code tree: The two lowest freq. characters must be at the children of the lowest internal node. (Why? Try a replacement argument)

Greedy Strategy

Find these two characters, build this node, repeat
(where some nodes are groups of characters as we go along).

procedure Huffman(f)
  // **Input:** An array $f[1 \ldots n]$ of freqs
  // **Output:** An encoding tree with n leaves
  $H \leftarrow$ a priority queue of integers, ordered by f
  for $i \leftarrow 1$ **to** n **do** insert($H, i, f[i]$)
  for $k \leftarrow n+1$ **to** $2n-1$ **do**
    $i \leftarrow$ deletemin(H); $j \leftarrow$ deletemin(H)
    create a node numbered k with children i, j
    $f[k] \leftarrow f[i] + f[j]$; insert($H, k, f[k]$)



Huffman Encoding, 5

```
procedure Huffman(f)
  // Input: An array f[1...n] of freqs
  // Output: An encoding tree with n leaves
  H ← a priority queue of integers, ordered by f
  for i ← 1 to n do insert(H, i, f[i])
  for k ← n + 1 to 2n − 1 do
    i ← deletemin(H)
    j ← deletemin(H)
    create a node numbered k with children i, j
    f[k] ← f[i] + f[j]
    insert(H, k, f[k])
  return deletemin(H)
```

Example

a :	45%
b :	13%
c :	12%
d :	16%
e :	9%
f :	5%

[Trace on board]

Huffman Encoding, 6

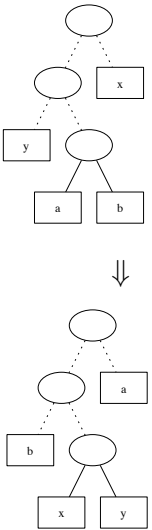
```
procedure Huffman(f)
  // Input: An array f[1...n] of freqs
  // Output: An encoding tree with n leaves
  H ← a priority queue of integers, ordered by f
  for i ← 1 to n do insert(H, i, f[i])
  for k ← n + 1 to 2n − 1 do
    i ← deletemin(H)
    j ← deletemin(H)
    create a node numbered k with children i, j
    f[k] ← f[i] + f[j]
    insert(H, k, f[k])
  return deletemin(H)
```

Runtime Analysis

- *initializing H:* $\Theta(n)$ time
- *for-loop iterations:* $n - 1$
- *deletemin's & insert's:* cost $O(\log n)$ each

Total: $\Theta(n) + (n-1)O(\log n)$
$= O(n \log n)$.

Huffman Encoding, 7: Correctness

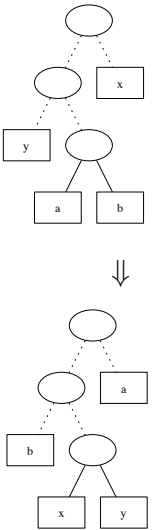


Suppose x & y are the two chars with the smallest freqs with $f[x] \leq f[y]$.

Lemma (1)

There is an optimal code tree in which x and y have the same length and differ only in their last bit.

Huffman Encoding, 7: Correctness



Suppose x & y are the two chars with the smallest freqs with $f[x] \leq f[y]$.

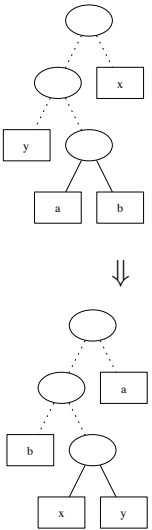
Lemma (1)

There is an optimal code tree in which x and y have the same length and differ only in their last bit.

Proof.

Suppose T is an optimal code tree and characters a and b which are max-depth siblings in T where $f[a] \leq f[b]$.

Huffman Encoding, 7: Correctness



Suppose x & y are the two chars with the smallest freqs with $f[x] \leq f[y]$.

Lemma (1)

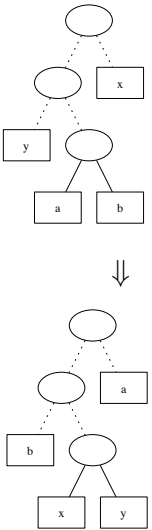
There is an optimal code tree in which x and y have the same length and differ only in their last bit.

Proof.

Suppose T is an optimal code tree and characters a and b which are max-depth siblings in T where $f[a] \leq f[b]$.

Let T' be the result of swapping $a \leftrightarrow x$ and $b \leftrightarrow y$. Then:

Huffman Encoding, 7: Correctness



Suppose x & y are the two chars with the smallest freqs with $f[x] \leq f[y]$.

Lemma (1)

There is an optimal code tree in which x and y have the same length and differ only in their last bit.

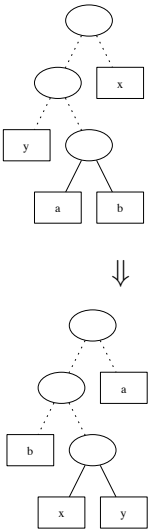
Proof.

Suppose T is an optimal code tree and characters a and b which are max-depth siblings in T where $f[a] \leq f[b]$.

Let T' be the result of swapping $a \leftrightarrow x$ and $b \leftrightarrow y$. Then:

$$\begin{aligned} cost(T) - cost(T') &= f[x] \cdot (d_T(x) - d_T(a)) + f[y] \cdot (d_T(y) - d_T(b)) \\ &\quad + f[a] \cdot (d_T(a) - d_T(x)) + f[b] \cdot (d_T(b) - d_T(y)) \\ &= (f[a] - f[x]) \cdot (d_T(a) - d_T(x)) \\ &\quad + (f[b] - f[y]) \cdot (d_T(b) - d_T(y)) \;\geq 0. \end{aligned}$$

Huffman Encoding, 7: Correctness



Suppose x & y are the two chars with the smallest freqs with $f[x] \leq f[y]$.

Lemma (1)

There is an optimal code tree in which x and y have the same length and differ only in their last bit.

Proof.

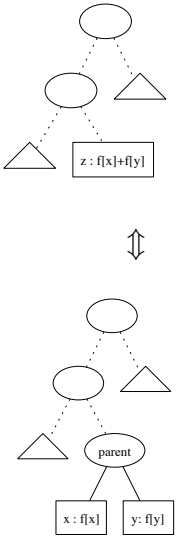
Suppose T is an optimal code tree and characters a and b which are max-depth siblings in T where $f[a] \leq f[b]$.

Let T' be the result of swapping $a \leftrightarrow x$ and $b \leftrightarrow y$. Then:

$$\begin{aligned} cost(T) - cost(T') &= f[x] \cdot (d_T(x) - d_T(a)) + f[y] \cdot (d_T(y) - d_T(b)) \\ &\quad + f[a] \cdot (d_T(a) - d_T(x)) + f[b] \cdot (d_T(b) - d_T(y)) \\ &= (f[a] - f[x]) \cdot (d_T(a) - d_T(x)) \\ &\quad + (f[b] - f[y]) \cdot (d_T(b) - d_T(y)) \;\geq 0. \end{aligned}$$

So, $cost(T) \geq cost(T')$. $\therefore$ Since T is optimal, so is T'. □

Huffman Encoding, 8: Correctness

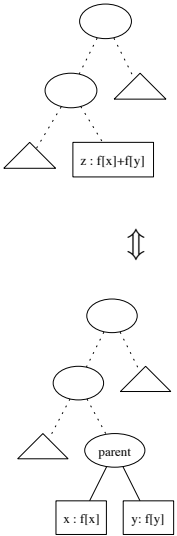


Suppose x & y are the two chars with the smallest freqs with $f[x] \leq f[y]$.

Lemma (2)

Replace x and y by a new character z with frequency $f[x] + f[y]$. Suppose T' is an optimal code tree for the new character set.
***Then** swapping the z-node for a node with children x and y results in an optimal code tree T for the old character set.*

Huffman Encoding, 8: Correctness



Suppose x & y are the two chars with the smallest freqs with $f[x] \leq f[y]$.

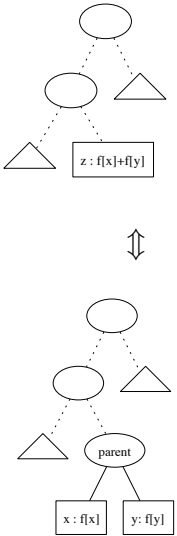
Lemma (2)

Replace x and y by a new character z with frequency $f[x] + f[y]$. Suppose T' is an optimal code tree for the new character set.
***Then** swapping the z-node for a node with children x and y results in an optimal code tree T for the old character set.*

Proof.

Then $cost(T) = cost(T') + f[x] + f[y]$. (Why?)

Huffman Encoding, 8: Correctness



Suppose x & y are the two chars with the smallest freqs with $f[x] \leq f[y]$.

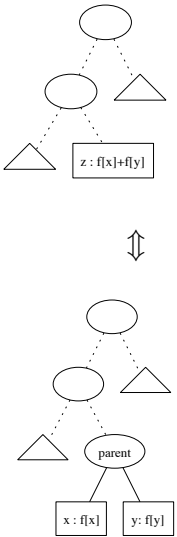
Lemma (2)

Replace x and y by a new character z with frequency $f[x] + f[y]$. Suppose T' is an optimal code tree for the new character set.
***Then** swapping the z-node for a node with children x and y results in an optimal code tree T for the old character set.*

Proof.

Then $cost(T) = cost(T') + f[x] + f[y]$. (Why?)
Suppose T'' is an optimal code tree for the old char. set.

Huffman Encoding, 8: Correctness



Suppose x & y are the two chars with the smallest freqs with $f[x] \leq f[y]$.

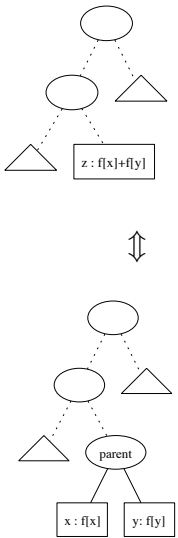
Lemma (2)

Replace x and y by a new character z with frequency $f[x] + f[y]$. Suppose T' is an optimal code tree for the new character set.
***Then** swapping the z-node for a node with children x and y results in an optimal code tree T for the old character set.*

Proof.

Then $cost(T) = cost(T') + f[x] + f[y]$. (Why?)
Suppose T'' is an optimal code tree for the old char. set.
WLOG, T'' has x and y as siblings of max depth. (Why?)

Huffman Encoding, 8: Correctness



Suppose x & y are the two chars with the smallest freqs with $f[x] \leq f[y]$.

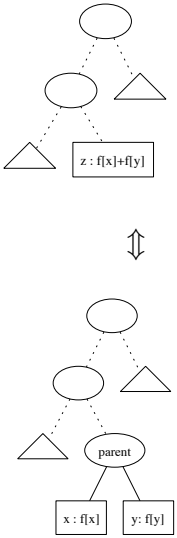
Lemma (2)

Replace x and y by a new character z with frequency $f[x] + f[y]$. Suppose T' is an optimal code tree for the new character set.
***Then** swapping the z-node for a node with children x and y results in an optimal code tree T for the old character set.*

Proof.

Then $cost(T) = cost(T') + f[x] + f[y]$. (Why?)
Suppose T'' is an optimal code tree for the old char. set.
WLOG, T'' has x and y as siblings of max depth. (Why?)
Replace x's and y's parent's subtree with a node for z with frequency $f[x] + f[y]$ and call the tree T'''.

Huffman Encoding, 8: Correctness



Suppose x & y are the two chars with the smallest freqs with $f[x] \leq f[y]$.

Lemma (2)

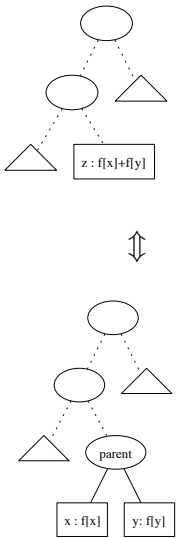
Replace x and y by a new character z with frequency $f[x]+f[y]$. Suppose T' is an optimal code tree for the new character set.
Then *swapping the z-node for a node with children x and y results in an optimal code tree T for the old character set.*

Proof.

Then $cost(T) = cost(T') + f[x] + f[y]$. (Why?)
Suppose T'' is an optimal code tree for the old char. set.
WLOG, T'' has x and y as siblings of max depth. (Why?)
Replace x's and y's parent's subtree with a node for z with frequency $f[x]+f[y]$ and call the tree T'''. Then $cost(T''')$

$$= cost(T'') - f[x] - f[y] \leq cost(T) - f[x] - f[y] = cost(T').$$

Huffman Encoding, 8: Correctness



Suppose x & y are the two chars with the smallest freqs with $f[x] \leq f[y]$.

Lemma (2)

Replace x and y by a new character z with frequency $f[x] + f[y]$. Suppose T' is an optimal code tree for the new character set.
***Then** swapping the z-node for a node with children x and y results in an optimal code tree T for the old character set.*

Proof.

Then $cost(T) = cost(T') + f[x] + f[y]$. (Why?)
Suppose T'' is an optimal code tree for the old char. set.
WLOG, T'' has x and y as siblings of max depth. (Why?)
Replace x's and y's parent's subtree with a node for z with frequency $f[x] + f[y]$ and call the tree T'''. Then $cost(T''')$

$$= cost(T'') - f[x] - f[y] \leq cost(T) - f[x] - f[y] = cost(T').$$

But as T' is optimal, so is T'''. $\therefore cost(T) = cost(T'')$ & T is also opt. □

Huffman Encoding, 9: Correctness

Suppose x & y are the two chars with the smallest frequencies with $f[x] \leq f[y]$.

Lemma 1: The greedy choice is safe

There is an optimal code tree in which x and y have the same length and differ only in their last bit.

Lemma 2: Optimal code trees have optimal substructure

Replace x and y by a new character z with frequency $f[x] + f[y]$. Suppose T' is an optimal code tree for the new character set. ***Then*** *swapping the z-node for a node with children x and y results in an optimal code tree T for the old char. set.*

```
procedure Huffman(f)
    // Input: An array f[1...n] of freqs
    // Output: An encoding tree with n leaves
    H ← a priority queue of integers, ordered by f
    for i ← 1 to n do insert(H,i,f[i])
    for k ← n+1 to 2n−1 do
        i ← deletemin(H);  j ← deletemin(H)
            // Safe by Lemma 1
        create a node numbered k with children i,j
        f[k] ← f[i] + f[j];  insert(H,k,f[k])
            // Safe by Lemma 2
```

Improving on Huffman: LZ Compression

- LZ = Abraham Lempel and Jacob Ziv
- The rough idea: Start with Huffman, but . . .
  - Keep statistics on frequencies in a sliding window of a few K.
  - Keep readjusting the Huffman coding to fix the freqs of the sliding window (& and note the change in coding in the compressed file).
- Huffman $\approx$ LV with the sliding window = the whole file
- There are many variations on this, see: `http://en.wikipedia.org/wiki/LZ77_and_LZ78`.

Propositional Logic

- The ***formulas*** of propositional logic are given by the grammar:

$$P \;::=\; \mathit{Var} \;\mid\; \neg P \;\mid\; P \wedge P \;\mid\; P \vee P \;\mid\; P \Rightarrow P \qquad \mathit{Var} \;::=\; \text{standard syntax}$$

- A ***truth assignment*** is a function $\mathcal{I} : \text{Variables} \to \{\,\mathsf{False}, \mathsf{True}\,\}$.
- A truth assignment $\mathcal{I}$ determines the value of a formula as follows:

$$\begin{aligned}
\mathcal{I}[\![x]\!] &= \mathsf{True} \;\; \textit{iff} \;\; \mathcal{I}(x) = \mathsf{True} \quad (x \text{ a variable}) \\
\mathcal{I}[\![\neg p]\!] &= \mathsf{True} \;\; \textit{iff} \;\; \mathcal{I}[\![p]\!] = \mathsf{False} \\
\mathcal{I}[\![p \wedge q]\!] &= \mathsf{True} \;\; \textit{iff} \;\; \mathcal{I}[\![p]\!] = \mathcal{I}[\![q]\!] = \mathsf{True}. \\
\mathcal{I}[\![p \vee q]\!] &= \mathsf{True} \;\; \textit{iff} \;\; \mathcal{I}[\![p]\!] = \mathsf{True} \quad \text{or} \quad \mathcal{I}[\![q]\!] = \mathsf{True}. \\
\mathcal{I}[\![p \Rightarrow q]\!] &= \mathsf{True} \;\; \textit{iff} \;\; \mathcal{I}[\![p]\!] = \mathsf{False} \quad \text{or} \quad \mathcal{I}[\![q]\!] = \mathsf{True}.
\end{aligned}$$

- A ***satisfying assignment*** for a formula p is an $\mathcal{I}$ with $\mathcal{I}[\![p]\!] = \mathsf{True}$.
- Finding satisfying assignments for general propositional formulas seems ***hard***. (See Chapter 8.)

Horn clauses

Definition

A **Horn clause** is a propositional logic formula of one of two special forms:

Positive Implications:

$$Var \wedge \cdots \wedge Var \Rightarrow Var$$

Pure negative clauses:

$$\neg Var \vee \cdots \vee \neg Var$$

A **Horn formula** is the conjunction of a set of Horn clauses.

Example Horn Formula

$$toddler \Rightarrow child$$
$$(child \wedge male) \Rightarrow boy$$
$$infant \Rightarrow child$$
$$(child \wedge female) \Rightarrow girl$$
$$\Rightarrow toddler$$
$$\Rightarrow female$$
$$\neg girl$$

Example from: http://bluehawk.monmouth.edu/~rscherl/Classes/KF/slides6.pdf

Satisfying Horn Formulas, 1

A ***Horn clause*** is a propositional logic formula of one of two special forms:

Positive Implications:
$Var \wedge \cdots \wedge Var \Rightarrow Var$

Pure negative clauses:
$\neg Var \vee \cdots \vee \neg Var$

A ***Horn formula*** is the conjunction of a set of Horn clauses.

Finding Satisfying Assignments for Sets of Clauses

Given: A set of Horn clauses: $\{ c_1, \ldots, c_n \}$.
Find: Find a truth assignment $\mathcal{I}$ that satisfies each of $c_1, \ldots, c_n$ ***or else*** report that there is no such $\mathcal{I}$.

Observation:

1. The positive implications push us to make things true.
2. The pure negative clauses push us to make things false.

Strategy:

- We greedily build up a satisfying assignment $\mathcal{I}$ for the positive implications — *making a few variables* True *as possible*.
- We check that $\mathcal{I}$ also satisfies the pure negative clauses.

Satisfying Horn Formulas, 2

Trace with:

$(w \wedge y \wedge z) \Rightarrow x, \quad (x \wedge z) \Rightarrow w, \quad x \Rightarrow y, \quad \Rightarrow x, \quad (x \wedge y) \Rightarrow w, \quad (\neg w \vee \neg x \vee \neg y), \quad (\neg z)$

// **Input:** H, a Horn formula
// **Output:** a satisfying assignment, if one exists
$T \leftarrow \emptyset$ // T = the set of vars set to True
// **Invariant:** Each $x \in T$ must be set to True in
// *any* satisfying assignment.
while (there is an $(x_1 \wedge \cdots \wedge x_k) \Rightarrow x_0$ in H with $x_1, \ldots, x_k \in T$ but $x_0 \notin T$) **do**
  $T \leftarrow T \cup \{ x_0 \}$
for (each pure negative clause $(\neg x_1 \vee \cdots \vee \neg x_k)$ in H) **do**
  if $x_1, \ldots, x_k \in T$
    then return No satisfying assignment
return T

Satisfying Horn Formulas, 2

Trace with:

$(w \wedge y \wedge z) \Rightarrow x, \quad (x \wedge z) \Rightarrow w, \quad x \Rightarrow y, \quad \Rightarrow x, \quad (x \wedge y) \Rightarrow w, \quad (\neg w \vee \neg x \vee \neg y), \quad (\neg z)$

// **Input:** H, a Horn formula
// **Output:** a satisfying assignment, if one exists
$T \leftarrow \emptyset$ // T = the set of vars set to True
// **Invariant:** Each $x \in T$ must be set to True in
// *any* satisfying assignment.

while $\begin{pmatrix}\text{there is an } (x_1 \wedge \cdots \wedge x_k) \Rightarrow x_0 \text{ in} \\ H \text{ with } x_1, \ldots, x_k \in T \text{ but } x_0 \notin T\end{pmatrix}$ **do**
  $T \leftarrow T \cup \{ x_0 \}$
for $\begin{pmatrix}\text{each pure negative clause} \\ (\neg x_1 \vee \cdots \vee \neg x_k) \text{ in } H\end{pmatrix}$ **do**
  if $x_1, \ldots, x_k \in T$
    then return No satisfying assignment
return T

Step 1. $T \leftarrow \emptyset$

Satisfying Horn Formulas, 2

Trace with:

$(w \wedge y \wedge z) \Rightarrow x, \quad (x \wedge z) \Rightarrow w, \quad x \Rightarrow y, \quad \Rightarrow x, \quad (x \wedge y) \Rightarrow w, \quad (\neg w \vee \neg x \vee \neg y), \quad (\neg z)$

// **Input:** H, a Horn formula
// **Output:** a satisfying assignment, if one exists
$T \leftarrow \emptyset$ // T = the set of vars set to True
// **Invariant:** Each $x \in T$ must be set to True in
// *any* satisfying assignment.

while $\begin{pmatrix}\text{there is an } (x_1 \wedge \cdots \wedge x_k) \Rightarrow x_0 \text{ in} \\ H \text{ with } x_1, \ldots, x_k \in T \text{ but } x_0 \notin T\end{pmatrix}$ **do**
  $T \leftarrow T \cup \{ x_0 \}$
for $\begin{pmatrix}\text{each pure negative clause} \\ (\neg x_1 \vee \cdots \vee \neg x_k) \text{ in } H\end{pmatrix}$ **do**
  if $x_1, \ldots, x_k \in T$
    then return No satisfying assignment
return T

Step 1. $T \leftarrow \emptyset$

Step 2. $T \leftarrow T \cup \{ x \}$
because of: $\Rightarrow x$ and $\emptyset \subseteq T$

Satisfying Horn Formulas, 2

Trace with:

$(w \wedge y \wedge z) \Rightarrow x, \quad (x \wedge z) \Rightarrow w, \quad x \Rightarrow y, \quad \Rightarrow x, \quad (x \wedge y) \Rightarrow w, \quad (\neg w \vee \neg x \vee \neg y), \quad (\neg z)$

// **Input:** H, a Horn formula
// **Output:** a satisfying assignment, if one exists
$T \leftarrow \emptyset$ // T = the set of vars set to True
// **Invariant:** Each $x \in T$ must be set to True in
// *any* satisfying assignment.

while (there is an $(x_1 \wedge \cdots \wedge x_k) \Rightarrow x_0$ in H with $x_1, \ldots, x_k \in T$ but $x_0 \notin T$) **do**
  $T \leftarrow T \cup \{ x_0 \}$

for (each pure negative clause $(\neg x_1 \vee \cdots \vee \neg x_k)$ in H) **do**
  if $x_1, \ldots, x_k \in T$
    then return No satisfying assignment

return T

Step 1. $T \leftarrow \emptyset$

Step 2. $T \leftarrow T \cup \{ x \}$
because of: $\Rightarrow x$ and $\emptyset \subseteq T$

Step 3. $T \leftarrow T \cup \{ y \}$
because of: $x \Rightarrow y$ and $\{ x \} \subseteq T$

Satisfying Horn Formulas, 2

Trace with:

$(w \wedge y \wedge z) \Rightarrow x, \quad (x \wedge z) \Rightarrow w, \quad x \Rightarrow y, \quad \Rightarrow x, \quad (x \wedge y) \Rightarrow w, \quad (\neg w \vee \neg x \vee \neg y), \quad (\neg z)$

// **Input:** H, a Horn formula
// **Output:** a satisfying assignment, if one exists
$T \leftarrow \emptyset$ // T = the set of vars set to True
// **Invariant:** Each $x \in T$ must be set to True in
// *any* satisfying assignment.

while (there is an $(x_1 \wedge \cdots \wedge x_k) \Rightarrow x_0$ in H with $x_1, \ldots, x_k \in T$ but $x_0 \notin T$) **do**
  $T \leftarrow T \cup \{ x_0 \}$

for (each pure negative clause $(\neg x_1 \vee \cdots \vee \neg x_k)$ in H) **do**
  if $x_1, \ldots, x_k \in T$
    then return No satisfying assignment

return T

Step 1. $T \leftarrow \emptyset$

Step 2. $T \leftarrow T \cup \{ x \}$
because of: $\Rightarrow x$ and $\emptyset \subseteq T$

Step 3. $T \leftarrow T \cup \{ y \}$
because of: $x \Rightarrow y$ and $\{ x \} \subseteq T$

Step 4. $T \leftarrow T \cup \{ w \}$
because of: $(x \wedge y) \Rightarrow w$ and $\{ x, y \} \subseteq T$

Satisfying Horn Formulas, 2

Trace with:

$(w \wedge y \wedge z) \Rightarrow x, \quad (x \wedge z) \Rightarrow w, \quad x \Rightarrow y, \quad \Rightarrow x, \quad (x \wedge y) \Rightarrow w, \quad (\neg w \vee \neg x \vee \neg y), \quad (\neg z)$

// **Input:** H, a Horn formula
// **Output:** a satisfying assignment, if one exists
$T \leftarrow \emptyset$ // T = the set of vars set to True
// **Invariant:** Each $x \in T$ must be set to True in
// *any* satisfying assignment.

while (there is an $(x_1 \wedge \cdots \wedge x_k) \Rightarrow x_0$ in H with $x_1, \ldots, x_k \in T$ but $x_0 \notin T$) **do**
  $T \leftarrow T \cup \{ x_0 \}$

for (each pure negative clause $(\neg x_1 \vee \cdots \vee \neg x_k)$ in H) **do**
  if $x_1, \ldots, x_k \in T$
    then return No satisfying assignment

return T

Step 1. $T \leftarrow \emptyset$

Step 2. $T \leftarrow T \cup \{ x \}$
because of: $\Rightarrow x$ and $\emptyset \subseteq T$

Step 3. $T \leftarrow T \cup \{ y \}$
because of: $x \Rightarrow y$ and $\{ x \} \subseteq T$

Step 4. $T \leftarrow T \cup \{ w \}$
because of: $(x \wedge y) \Rightarrow w$ and $\{ x, y \} \subseteq T$

Step 5. The while loop exits and $(\neg w \vee \neg x \vee \neg y)$ is unsatisfiable since $w, x, y \in T$

Satisfying Horn Formulas, 3

// **Input:** H, a Horn formula (i.e., a set of Horn clauses)
// **Output:** a satisfying assignment, if one exists
$T \leftarrow \emptyset$ // = the set of vars set to True
// **Invariant:** Each $x \in T$ must be set to True in *any* satisfying assignment.
while (there is an $(x_1 \wedge \cdots \wedge x_k) \Rightarrow x_0$ in H with $x_1, \ldots, x_k \in T$ but $x_0 \notin T$) **do**
  $T \leftarrow T \cup \{ x_0 \}$
for each pure negative clause $(\neg x_1 \vee \cdots \vee \neg x_k)$ in H **do**
  if $x_1, \ldots, x_k \in T$ **then return** No satisfying assignment
return T

Why does this work?

- **Claim 1:** The invariant holds in the while-loop. (Why?)
- **Claim 2:** The while-loop eventually terminates. (Why?)
- **Claim 3:** When the while-loop terminates, T = the set of variables that must be true in any satisfying assignment for H's positive implications. (Why?)
- **Claim 4:** The algorithm is correct. (Why?)

Satisfying Horn Formulas, 4

// **Input:** H, a Horn formula (i.e., a set of Horn clauses)
// **Output:** a satisfying assignment, if one exists
$T \leftarrow \emptyset$ // = the set of vars set to True
// **Invariant:** Each $x \in T$ must be set to True in *any* satisfying assignment.
while (there is an $(x_1 \wedge \cdots \wedge x_k) \Rightarrow x_0$ in H with $x_1, \ldots, x_k \in T$ but $x_0 \notin T$) **do**
  $T \leftarrow T \cup \{ x_0 \}$
for each pure negative clause $(\neg x_1 \vee \cdots \vee \neg x_k)$ in H **do**
  if $x_1, \ldots, x_k \in T$ **then return** No satisfying assignment
return T

Runtime Analysis

- $n =$ the number of characters in the Horn formula.
- Naïvely, $O(n^2)$ time. (Why?)

!!! But see DPV Exercise 5.33.

Note: This is in part a setup for Chapter 8.

Set Cover, 1

Suppose B is a set and $S_1, \ldots, S_m \subseteq B$.

Definition

(a) A ***set cover*** of B is a $\{ S'_1, \ldots, S'_k \} \subseteq \{ S_1, \ldots, S_m \}$ with $B \subseteq \cup_{i=1}^{k} S'_i$

(b) A ***minimal set cover*** of B is a set cover of B using as few of the S_i-sets as possible.

The Set Cover Problem (SCP)

Given: B and $S_1, \ldots, S_m$ as above.
Find: A minimal set cover of B.

Example

For: $B = \{ 1, \ldots, 14 \}$ and

$$S_1 = \{ 1, 2 \}$$
$$S_2 = \{ 3, 4, 5, 6 \}$$
$$S_3 = \{ 7, 8, 9, 10, 11, 12, 13, 14 \}$$
$$S_4 = \{ 1, 3, 5, 7, 9, 11, 13 \}$$
$$S_5 = \{ 2, 4, 6, 8, 10, 12, 14 \}$$

the solution to SCP is $\{ S_4, S_5 \}$.

Set Cover, 2

A Greedy Approx. to the Set Cover Problem

// **Input:** B and $S_1, \ldots, S_m \subseteq B$ as above.
// **Output:** A set cover of B which is close to minimal.
$\mathcal{C} \leftarrow \emptyset$
while (some element of B is not yet covered) **do**
  Pick the S_i with the largest number
    of uncovered B-elements
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{ S_i \}$
return $\mathcal{C}$

Example

For: $B = \{ 1, \ldots, 14 \}$ and

$$S_1 = \{ 1, 2 \}$$
$$S_2 = \{ 3, 4, 5, 6 \}$$
$$S_3 = \{ 7, 8, 9, 10, 11, 12, 13, 14 \}$$
$$S_4 = \{ 1, 3, 5, 7, 9, 11, 13 \}$$
$$S_5 = \{ 2, 4, 6, 8, 10, 12, 14 \}$$

The algorithm returns $\{ S_1, S_2, S_3 \}$ — which is **not** optimal, but not too bad.

Set Cover, 3

A Greedy Approx. to SCP

// **Input:** B and $S_1, \ldots, S_m \subseteq B$
// **Output:** A near min. set cover
$\mathcal{C} \leftarrow \emptyset$
while (all of B is not covered) **do**
  Pick the S_i with the largest number of uncovered B-elms
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{ S_i \}$
return $\mathcal{C}$

Claim

Suppose B contains n elements and the min. cover has k sets. **Then** the greedy algorithm will use at most $(k \ln n)$ sets.

Proof: Let
n_t = the number of uncovered elms after t-iterations
So $n_0 = n$.
After iteration $t > 0$:

- there are n_t elms left.
- k many sets cover them
- So there must be some set with at least n_{t-1}/k many elements.
- So by the greedy choice,

$$n_t \leq n_{t-1} - \frac{n_{t-1}}{k} = n_{t-1}\left(1 - \frac{1}{k}\right) = n_0\left(1 - \frac{1}{k}\right)^t.$$

Set Cover, 4

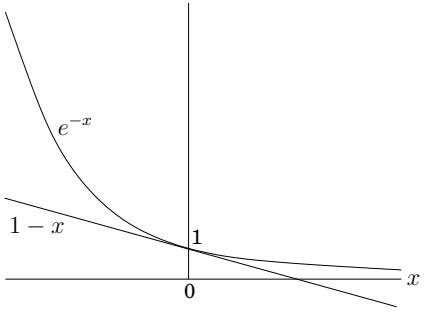
A Greedy Approx. to SCP

// **Input:** B and $S_1, \ldots, S_m \subseteq B$
// **Output:** A near min. set cover
$\mathcal{C} \leftarrow \emptyset$
while (all of B is not covered) **do**
  Pick the S_i with the largest number of uncovered B-elms
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{ S_i \}$
return $\mathcal{C}$

Claim

Suppose B contains n elements and the min. cover has k sets. **Then** the greedy algorithm will use at most $(k \ln n)$ sets.

n_t = the number of uncovered elms after t-iterations
We know: $n_t \leq n\left(1 - \frac{1}{k}\right)^t$.
Fact: $1 - x \leq e^{-x}$ for all x, with equality iff $x = 0$.



So: $n_t \leq n\left(1 - \frac{1}{k}\right)^t < n(e^{-1/k})^t = ne^{-t/k}$

Set Cover, 5

A Greedy Approx. to SCP

// **Input:** B and $S_1, \ldots, S_m \subseteq B$
// **Output:** A near min. set cover
$\mathcal{C} \leftarrow \emptyset$
while (all of B is not covered) **do**
  Pick the S_i with the largest number of uncovered B-elms
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{ S_i \}$
return $\mathcal{C}$

Claim

Suppose B contains n elements and the min. cover has k sets. **Then** the greedy algorithm will use at most $(k \ln n)$ sets.

n_t = the number of uncovered elms after t-iterations
We know: $n_t < n \cdot e^{-t/k}$ for $t > 0$.

$\therefore$ When $t > k \log_e n$,

$$n_t < n \cdot e^{-t/k} < n \cdot e^{-\log_e n} = \frac{n}{n} = 1.$$

i.e., we must have covered all of B.

So the greedy algorithm is optimal within a $\log_e n$ factor.

Fact: **If** *certain widely-held complexity assumptions hold*, **then** no poly-time algorithm has a better than an $(\log_e n)$-approximation factor.
(More on this in Chapters 8 and 9.)

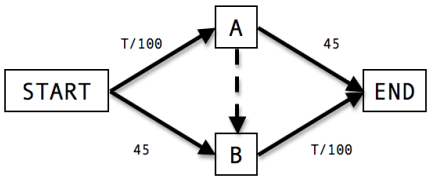
Braess's Paradox

By adding capacity to a network, can actually reduce(!!!) its throughput when "rational actors" can choose their routes through the network.

Example (Part 1)

- A road network of four roads with 4000 drivers.
- All want to go from START to END.
- Roads START→B and A→END have a 45 min travel time.
- Roads START→A and B→END have a T/100 min. travel time, where T = the number of travelers on that road.

- If 2000 drivers go the north route and 2000 go the south route, every one has a travel time of 45 + (2000/100) = 65 mins, which is optimal.

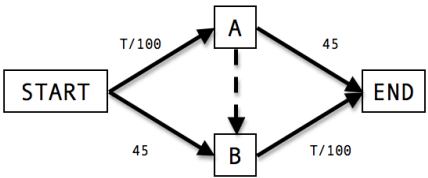


The example + image are from `http://en.wikipedia.org/wiki/Braess's_paradox`.

Aside: Braess's Paradox, 2

Example (Part 2)

- Now add the $A \to B$ road with travel time 0.
- Since all the drivers are "rational" (i.e., greedy), they will all take the $START \to A \to B$ route, since they can arrive at B five minutes faster than the $START \to B$ route.
- But then their **total** travel time to END is 80 mins.
- If any one driver tries another route, that driver gets a worse outcome (i.e., an $\approx$ 85 minute travel time).
- Since the drivers are all "rational", no one changes routes.
- So the travel time of the new network is 80 minutes.



Aside: Braess's Paradox, 3

Example (Part 3)

- This is what economists call a ***market failure***.
- See the Wikipedia article for
  - references to mathematically rigorous versions of the above, and
  - examples of actual networks that improved travel times *by closing roads*.
- The problem for computer scientists:
  - Many networks are inhabited by "rational actors".
  - How do we avoid situations like this?