Winter 2016
COMP-250: Introduction to Computer Science
Lecture 24, April 7, 2016
Strings and Pattern Matching

- Insertion is $O(dn)$ when $d$ is the size of the alphabet.

A trie supports the following operations on a set $S$ of strings:

- An internal node can have $1$ to $d$ children where $d$ is the size of the alphabet.

Tries can be used to perform the following operations on a set $S$ of strings:

- A trie is a tree-based data structure for storing a set of strings.

Prefix queries search for the longest prefix of a given string $X$ that matches a string in the trie.

- The query terminates at node $v$. Let $X_1$ be the longest prefix of $X$ associated with $v$.
- The query terminates at an edge $(v, w)$ because $X_2$ is the rest of $X$. Let $Y_1$ be the part of $Y$ that $X$ matched to and $Y_2$ the rest of $Y$.

Prefix queries can end in the following ways:

- Each internal node in a compressed trie has at least two children and each external node is associated with a string from $S$, $v$ has at least two children, or $v$ is the root.
- Each edge of a compressed trie is labeled with a character from the alphabet.

To convert a standard trie to a compressed trie we replace an edge $(v_0, v_1)$ each chain on nodes $(v_0, v_1, v_2, \ldots, v_k)$ for $k \geq 2$ such that $v_1$ has only one child, $v$ is a critical node, each $v_i$ has at least two children, and $v_{i+1}$ is critical for $0 < i < k$.

Compressed Tries (cont.)

Prefix queries can be performed in $O(n)$ time.

Algorithm

\begin{algorithm}
\begin{algorithmic}
\State \textbf{Input:} String $X$
\State \textbf{Output:} Longest prefix of $X$ in $S$.
\begin{algorithmic}
\State $v \leftarrow \text{root}$
\State $l \leftarrow 0$
\While{$v$ is not a leaf and $X[l] \neq \text{label}(v)$}
\State $v \leftarrow \text{child}(v, X[l])$
\State $l \leftarrow l + 1$
\EndWhile
\State \textbf{Return} $X[0\ldots l-1]$
\end{algorithmic}
\end{algorithmic}
\end{algorithm}

Tries (cont.)

Deletion can be performed in $O(n)$ time.

Algorithm

\begin{algorithm}
\begin{algorithmic}
\State \textbf{Input:} String $X$
\State \textbf{Output:} String $X$.
\begin{algorithmic}
\State $v \leftarrow \text{root}$
\State $l \leftarrow 0$
\While{$v$ is not a leaf and $X[l] \neq \text{label}(v)$}
\State $v \leftarrow \text{child}(v, X[l])$
\State $l \leftarrow l + 1$
\EndWhile
\If{$v$ is a leaf}
\State Remove $v$ from the trie.
\EndIf
\State \textbf{Return} $X[0\ldots l-1]$
\end{algorithmic}
\end{algorithmic}
\end{algorithm}

Example: The standard trie over the alphabet $\Sigma = \{a, b\}$ for strings $ab$, $ba$, $aa$, $bb$, $aba$, $bab$, $abb$. Each internal node in a compressed trie has at least two children.

Tries

- Each edge of a compressed trie is labeled with a character from the alphabet.

Algorithms for insertion and deletion:

- Insertion: $O(dn)$ time.
- Deletion: $O(n)$ time.
Tries

- A trie is a tree-based data structure for storing strings in order to make pattern matching faster.

- Tries can be used to perform prefix queries for information retrieval. Prefix queries search for the longest prefix of a given string $X$ that matches a prefix of some string in the trie.

- A trie supports the following operations on a set $S$ of strings:

  - **insert**($X$): Insert the string $X$ into $S$
    - **Input**: String
    - **Output**: None

  - **remove**($X$): Remove string $X$ from $S$
    - **Input**: String
    - **Output**: None

  - **prefixes**($X$): Return all the strings in $S$ that have a longest prefix of $X$
    - **Input**: String
    - **Output**: Enumeration of strings
• Let $S$ be a set of strings from the alphabet $\Sigma$ such that no string in $S$ is a prefix to another string. A **standard trie** for $S$ is an ordered tree $T$ that:
  - Each edge of $T$ is labeled with a character from $\Sigma$
  - The ordering of edges out of an internal node is determined by the alphabet $\Sigma$
  - The path from the root of $T$ to any node represents a prefix in $\Sigma$ that is equal to the concatenation of the characters encountered while traversing the path.

• For example, the standard trie over the alphabet $\Sigma = \{a, b\}$ for the set $\{aabab, abaab, baaaa, bbbab\}$
Tries (cont.)

- An internal node can have 1 to \(d\) children when \(d\) is the size of the alphabet. Our example is essentially a binary tree.

- A path from the root of \(T\) to an internal node \(v\) at depth \(i\) corresponds to an \(i\)-character prefix of a string of \(S\).

- We can implement a trie with an ordered tree by storing the character associated with an edge at the child node below it.
Compressed Tries

- A **compressed trie** is like a standard trie but makes sure that each trie had a degree of at least 2. Single child nodes are compressed into a single edge.

- A **critical node** is a node \( v \) such that \( v \) is labeled with a string from \( S \), \( v \) has at least 2 children, or \( v \) is the root.

- To convert a standard trie to a compressed trie we replace an edge \((v_0, v_1)\) by a chain of nodes \((v_0, v_1...v_k)\) for \( k \geq 2 \) such that
  - \( v_0 \) and \( v_1 \) are critical but \( v_i \) is critical for \( 0 < i < k \)
  - each \( v_i \) has only one child

- Each internal node in a compressed trie has at least two children and each external is associated with a string. The compression reduces the total space for the trie from \( O(m) \) where \( m \) is the sum of the lengths of strings in \( S \) to \( O(n) \) where \( n \) is the number of strings in \( S \).
Compressed Tries (cont.)

• An example:

Strings and Pattern Matching

Prefix Queries on a Trie

Algorithm

prefixQuery(T, X):

Input: Trie T for a set S of strings and a query string X

Output: The node v of T such that the labeled nodes of the subtree of T rooted at v store the strings of S with a longest prefix in common with X

v ← T.root()

i ← 0

repeat

for each child w of v do

let e be the edge (v, w)

Y ← string(e)

l ← Y.length()  

Z ← X.substring(i, i+l-1)  

if Z = Y then

v ← w

i ← i+1

break out of the for loop

else if a proper prefix of Z matched a proper prefix of Y then

v ← w

break out of the repeat loop

until v is external or v ≠ w

return v

Insertion and Deletion

• Insertion: We first perform a prefix query for string X. Let us examine the ways a prefix query may end in terms of insertion.

- The query terminates at node v. Let X1 be the prefix of X that matched in the trie up to node v and X2 be the rest of X. If X2 is an empt string we label v with X and the end. Otherwise we creat a new external node w and label it with X.

- The query terminates at an edge e=(v, w) because a prefix of X match prefix(v) and a proper prefix of string Y associated with e. Let Y1 be the part of Y that X mathed to and Y2 the rest of Y. Likewise for X1 and X2. Then X=X1+X2 = prefix(v) +Y1+X2. We create a new node u and split the edges(v, u) and (u, w). If X2 is empty then w label u with X. Otherwise we creat a node z which is external and label it X.

• Insertion is O(dn) when d is the size of the alphabet and n is the length of the string t insert.
Prefix Queries on a Trie

Algorithm prefixQuery(T, X):
   Input: Trie $T$ for a set $S$ of strings and a query string $X$
   Output: The node $v$ of $T$ such that the labeled nodes of
            the subtree of $T$ rooted at $v$ store the strings
            of $S$ with a longest prefix in common with $X$

   $v \leftarrow T.root()$
   $i \leftarrow 0$ \{i is an index into the string $X$\}
   repeat
      for each child $w$ of $v$ do
         let $e$ be the edge $(v,w)$
         $Y \leftarrow \text{string}(e)$ \{Y is the substring associated with $e$\}
         $l \leftarrow Y.length()$ \{l=1 if $T$ is a standard trie\}
         $Z \leftarrow X.substring(i, i+l-1)$ \{Z holds the next $l$ characters of $X$\}
         if $Z = Y$ then
            $v \leftarrow w$
            $i \leftarrow i+l$ \{move to W, incrementing $i$ past $Z$\}
            break out of the for loop
         else if a proper prefix of $Z$ matched a proper prefix
                  of $Y$ then
            $v \leftarrow w$
            break out of the repeat loop
      until $v$ is external or $v \neq w$
   return $v$
Insertion and Deletion

- Insertion: We first perform a prefix query for string X. Let us examine the ways a prefix query may end in terms of insertion.
  - The query terminates at node v. Let X₁ be the prefix of X that matched in the trie up to node v and X₂ be the rest of X. If X₂ is an empty string we label v with X and the end. Otherwise we create a new external node w and label it with X.
  - The query terminates at an edge e=(v, w) because a prefix of X match prefix(v) and a proper prefix of string Y associated with e. Let Y₁ be the part of Y that X matched to and Y₂ the rest of Y. Likewise for X₁ and X₂. Then X=X₁+X₂ = prefix(v) +Y₁+X₂. We create a new node u and split the edges(v, u) and (u, w). If X₂ is empty then we label u with X. Otherwise we create a node z which is external and label it X.

- Insertion is O(dn) when d is the size of the alphabet and n is the length of the string t insert.
Insertion and Deletion (cont.)

Prefix Queries on a Trie

Algorithm

\[ \text{prefixQuery}(T, X): \]

Input: Trie \( T \) for a set \( S \) of strings and a query string \( X \)

Output: The node \( v \) of \( T \) such that the labeled nodes of the subtree of \( T \) rooted at \( v \) store the strings of \( S \) with a longest prefix in common with \( X \)

\[ v \leftarrow T.\text{root()} \]

\[ i \leftarrow 0 \]

\[ \text{repeat} \]

\[ \text{for each child } w \text{ of } v \text{ do} \]

\[ \text{let } e \text{ be the edge } (v, w) \]

\[ Y \leftarrow \text{string}(e) \]

\[ l \leftarrow Y.\text{length()} \]

\[ Z \leftarrow X.\text{substring}(i, i+l-1) \]

\[ \text{if } Z = Y \text{ then} \]

\[ v \leftarrow w \]

\[ i \leftarrow i + 1 \]

\[ \text{break out of the for loop} \]

\[ \text{else if a proper prefix of } Z \text{ matched a proper prefix of } Y \text{ then} \]

\[ v \leftarrow w \]

\[ \text{break out of the repeat loop} \]

\[ \text{until } v \text{ is external or } v \neq w \]

\[ \text{return } v \]

Insertion and Deletion

• Insertion: We first perform a prefix query for string \( X \). Let us examine the ways a prefix query may end in terms of insertion.

- The query terminates at node \( v \). Let \( X_1 \) be the prefix of \( X \) that matched in the trie up to node \( v \) and \( X_2 \) be the rest of \( X \). If \( X_2 \) is an empt string we label \( v \) with \( X \) and the end. Otherwise we create a new external node \( w \) and label it with \( X \).

- The query terminates at an edge \( e=(v, w) \) because a prefix of \( X \) match prefix \((v) \) and a proper prefix of string \( Y \) associated with \( e \). Let \( Y_1 \) be the part of \( Y \) that \( X \) mathed to and \( Y_2 \) the rest of \( Y \). Likewise for \( X_1 \) and \( X_2 \). Then \( X = X_1 + X_2 = \text{prefix}(v) + Y_1 + X_2 \). We create a new node \( u \) and split the edges \((v, u)\) and \((u, w)\). If \( X_2 \) is empty then \( w \) label \( u \) with \( X \). Otherwise we create a node \( z \) which is external and label it \( X \).

Insertion is \( O(dn) \) when \( d \) is the size of the alphabet and \( n \) is the length of the string \( t \) insert.
Strings and Pattern Matching

• A graphical example:

```
abab  baab  babbb
```

Insertion and Deletion (cont.)

Lempel Ziv Encoding (contd.)

```
how  now  brown  cow  in  town.
```

Compressed text:

```
12 3 4 5
```

Insertion and Deletion (cont.)

Trie:

```
insert(bbaabb)
```

• How to decode?

ASCII encoding:

```
a = “0”, j = “01”, v = “00”
a = “0”, j = “11”, v = “10”
```

Example:

```
java
```

Reconstructing the string:

```
man ascii code (type
```

File Compression

variable-length encoding

in order to reduce the space required to store a text

• text files are usually stored by representing each

- is this

- text:

```
010000
```

- a = “0”, j = “01”, v = “00”

```
110100
```

- a = “0”, j = “11”, v = “10”

- encoding:

```
a = “0”, j = “11”, v = “10”
```

Uses binary codes of different lengths for different characters; thus, we can assign fewer bits to frequently used characters, and more bits to rarely used characters.

Reconstructing the string:

```
man ascii code (type
```

For each non-zero nodeIndex, put the substring corresponding to that node into the new string.

Every time you see a ‘0’ in the compressed string, add the next character in the compressed string directly to the new string.

If you come across a letter you've already seen, scan through the text before, add it to the top level of the trie.

If you come across a letter you haven't seen, scan down the trie until you can't match any more characters, add a node to the trie representing the new character, and then scan down the trie again.

Constructing the trie:

```
phrase 0 be the null string.
```

\[ \text{search stops here} \]

\[ \text{insert(bbaabb)} \]
Lempel Ziv Encoding

• Constructing the trie:
  - Let phrase 0 be the null string.
  - Scan through the text
  - If you come across a letter you haven’t seen before, add it to the top level of the trie.
  - If you come across a letter you’ve already seen, scan down the trie until you can’t match any more characters, add a node to the trie representing the new string.
  - Insert the pair (nodeIndex, lastChar) into the compressed string.

• Reconstructing the string:
  - Every time you see a ‘0’ in the compressed string add the next character in the compressed string directly to the new string.
  - For each non-zero nodeIndex, put the substring corresponding to that node into the new string, followed by the next character in the compressed string.
Lempel Ziv Encoding (contd.)

- A graphical example:

Uncompressed text: 

\[ \text{how now brown cow in town.} \]

phrases: 

\[ (\text{nil}) \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \]

Compressed text: 

\[ 0h0o0w0_0n2w4b0r6n4c6_0i5_0t9. \]

Trie:
File Compression

- text files are usually stored by representing each character with an 8-bit ASCII code (type `man ascii` in a Unix shell to see the ASCII encoding)

- the ASCII encoding is an example of **fixed-length encoding**, where each character is represented with the same number of bits

- in order to reduce the space required to store a text file, we can exploit the fact that some characters are more likely to occur than others

- **variable-length encoding** uses binary codes of different lengths for different characters; thus, we can assign fewer bits to frequently used characters, and more bits to rarely used characters.

- Example:
  - text: `java`
  - encoding: `a = “0”, j = “11”, v = “10”`
  - encoded text: `110100` (6 bits)

- How to decode?
  - `a = “0”, j = “01”, v = “00”`
  - encoded text: `010000` (6 bits)
  - is this `java`, `jv`, `jaaa`...
Encoding Trie

- to prevent ambiguities in decoding, we require that the encoding satisfies the prefix rule, that is, no code is a prefix of another code
  - \(a = "0", \ j = "11", \ v = "10"\) satisfies the prefix rule
  - \(a = "0", \ j = "01", \ v = "00"\) does not satisfy the prefix rule (the code of \(a\) is a prefix of the codes of \(j\) and \(v\))

- we use an encoding trie to define an encoding that satisfies the prefix rule
  - the characters stored at the external nodes
  - a left edge means 0
  - a right edge means 1

![Encoding Trie Diagram]

- \(A = 010\)
- \(B = 11\)
- \(C = 00\)
- \(D = 10\)
- \(R = 011\)
Strings and Pattern Matching

Encoding Trie

• to prevent ambiguities in decoding, we require that the encoding satisfies the prefix rule, that is, no code is a prefix of another code. For example, a = "0", j = "11", v = "10" satisfies the prefix rule, while a = "0", j = "01", v = "00" does not satisfy the prefix rule (the code of a is a prefix of the codes of j and v).

• we use an encoding trie to define an encoding that satisfies the prefix rule:

- the characters stored at the external nodes
- a left edge means 0
- a right edge means 1

A = 010
B = 11
C = 00
D = 10
R = 011

Example of Decoding

• trie:

• encoded text:

01011011010000101001011011010

• text:

ABRACADABRA

Optimal Compression

• An issue with encoding tries is to insure that the encoded text is as short as possible:

ABRACADABRA

01011011010000101001011011010
29 bits

ABRACADABRA

001011000100001100101100
24 bits

• text:

ABRACADABRA

010 11 011 010 00 010 10 010 11 011 010
Strings and Pattern Matching

Encoding Trie

- to prevent ambiguities in decoding, we require that the encoding satisfies the prefix rule, that is, no code is a prefix of another code.
  - a = "0", j = "11", v = "10" satisfies the prefix rule
  - a = "0", j = "01", v = "00" does not satisfy the prefix rule (the code of a is a prefix of the codes of j and v).

- we use an encoding trie to define an encoding that satisfies the prefix rule.
  - the characters stored at the external nodes:
    - a left edge means 0
    - a right edge means 1

A = 010
B = 11
C = 00
D = 10
R = 011

Example of Decoding

- trie:
- encoded text:
  01011011010000101001011011010
- text:
  A = 010
  B = 11
  C = 00
  D = 10
  R = 011

See? Decodes like magic...

Trie this!

Optimal Compression

- An issue with encoding tries is to insure that the encoded text is as short as possible:
  - DB
  - C
  - R
  - 01
  - 0
  - 001
  - 1
  - 1
  - A
  - B
  - R
  - A
  - C
  - A
  - D
  - A
  - B
  - R
  - A

ABRACADABRA

01011011010000101001011011010

29 bits

ABRACADABRA

001011000100001100101100

24 bits

ABRACADABRA

100011111001001100011101111000101010011010100

29 bits

ROBERTOKNOWSCS

10 00 01111100 10 0110 00 11101111 00 0101010011010100

1000011111001001100011101111000101010011010100

1000011111001001100011101111000101010011010100
Strings and Pattern Matching

Encoding Trie

- To prevent ambiguities in decoding, we require that the encoding satisfies the prefix rule, that is, no code is a prefix of another code.
  - $a = \text{"0"}$, $j = \text{"11"}$, $v = \text{"10"}$ satisfies the prefix rule.
  - $a = \text{"0"}$, $j = \text{"01"}$, $v = \text{"00"}$ does not satisfy the prefix rule (the code of $a$ is a prefix of the codes of $j$ and $v$).

- We use an encoding trie to define an encoding that satisfies the prefix rule.
  - The characters stored at the external nodes.
  - A left edge means 0.
  - A right edge means 1.

\[
\begin{align*}
A &= 010 \\
B &= 11 \\
C &= 00 \\
D &= 10 \\
R &= 011 \\
\end{align*}
\]

Example of Decoding

- Trie:
- Encoded text: 01011011010000101001011011010
- Text:
  - $A =$ 010
  - $B =$ 11
  - $C =$ 00
  - $D =$ 10
  - $R =$ 011

See? Decodes like magic...

Optimal Compression

- An issue with encoding tries is to insure that the encoded text is as short as possible:

\[
\begin{align*}
\text{ABRACADABRA} & \quad 01011011010000101001011011010 \\
& \quad 29 \text{ bits} \\
\text{ABRACADABRA} & \quad 001011000100001100101100 \\
& \quad 24 \text{ bits}
\end{align*}
\]
Another Huffman Encoding Trie

Construction Algorithm

• with a Huffman encoding trie, the encoded text has minimal length

Algorithm

Huffman(X):

Input: String X of length n
Output: Encoding trie for X

Compute the frequency \( f(c) \) of each character \( c \) of X.

Initialize a priority queue \( Q \).

for each character \( c \) in X do

Create a single-node tree \( T \) storing \( c \) \( Q \).\( insertItem(\( f(c) \), T) \)

while \( Q \).size() > 1 do

\( f_1 \leftarrow Q \).minKey()

\( T_1 \leftarrow Q \).removeMinElement()

\( f_2 \leftarrow Q \).minKey()

\( T_2 \leftarrow Q \).removeMinElement()

Create a new tree \( T \) with left subtree \( T_1 \) and right subtree \( T_2 \).

\( Q \).insertItem(\( f_1 + f_2 \))

return \( tree \)

\( Q \).removeMinElement()

• running time for a text of length n with k distinct characters: \( O(n + k \log k) \)
Construction Algorithm

- with a Huffman encoding trie, the encoded text has minimal length

**Algorithm** Huffman($X$):

**Input:** String $X$ of length $n$

**Output:** Encoding trie for $X$

Compute the frequency $f(c)$ of each character $c$ of $X$. Initialize a priority queue $Q$.

for each character $c$ in $X$ do
  Create a single-node tree $T$ storing $c$
  $Q$.insertItem($f(c)$, $T$)
while $Q$.size() $>$ 1 do
  $f_1$ $\leftarrow$ $Q$.minKey()
  $T_1$ $\leftarrow$ $Q$.removeMinElement()
  $f_2$ $\leftarrow$ $Q$.minKey()
  $T_2$ $\leftarrow$ $Q$.removeMinElement()
  Create a new tree $T$ with left subtree $T_1$ and right subtree $T_2$.
  $Q$.insertItem($f_1 + f_2$)
return tree $Q$.removeMinElement()

- running time for a text of length $n$ with $k$ distinct characters: $O(n + k \log k)$
Image Compression

- we can use Huffman encoding also for binary files (bitmaps, executables, etc.)
- common groups of bits are stored at the leaves
- Example of an encoding suitable for b/w bitmaps
Data Representation/
Lossy Compression

- Sound formats
- Image formats
- Movie formats
Data Representation

- sound formats
Sound formats
AIFF Sound format

each sample is a signed 15 (or 23 or 31) bits value

176 samples ≈ 4 ms
(44 100 samples = 1 s)
AIFF Sound format

- 44 100 samples / second
- 16 b = 2 B / sample
  (or 24 b = 3 B / sample
  or 32 b = 4 B / sample)
- stereo = two channels
- $2 \times 2 \times 44 100 = 176.4$ kB/s
- CD ≈ 700 MB ≈ 75 minutes
AIFF Sound format

- why 44 100 samples / second?
- because it is in the correct range...
- because 44 100 is divisible by 2,3,4,5,6,7,9,10
MP3 Sound format

- Based on Fourier transform.
- 576 samples of amplitude / time are converted to 576 samples of distinct frequencies.

Bass

Treble
In human ears, the cochlea is mechanically performing a process analog to the Fourier Transform. The eardrum vibrates back and forth according to the wave-like representation of the sound. The frequency information stimulates a specific area in the cochlea.
MP3 Sound format

- Frequencies with small coefficients removed
- Waveform reconstructed is close to original
MP3 Sound format

Bass

Treble

HIGH quality low
Data Representation

- Image formats
TIFF image format
TIFF image format

- an 8x8 sub-region of a large image:

- each individual pixel uses 24 bites: 8b for red, 8b for blue, 8b for green.

- total size = number of pixels x 3 Bytes.
Animal eyes focus light on the retina where an image of the environment is produced.

This image is analysed according to 3 types of colour sensitive cones, mostly triggered near the red, green and blue bands.

A perceived colour is a triplet \((x,y,z)\) of excitations of the 3 types of cones.

Two combinations of colours yielding the same triplet \((x,y,z)\) are indistinguishable.
Using a transformation similar to Fourier transform (used for audio), a so called Discrete Cosine Transform is applied to each sub-bloc of size 8x8.

Notice: colours are used for abstract data. Dark means small, bright means large.
If no data is removed, the resulting image is nearly identical to the original. Imprecision in the transform causes small errors.
If all data very close to zero is removed, the resulting image is only slightly different from the original.

**Notice:** colours are used for abstract data. Dark means small, bright means large.
If all data close to zero is removed, the resulting image is somewhat different from the original.
If all data of small magnitude is removed, the resulting image is still very similar to the original.

Notice: colours are used for abstract data. Dark means small, bright means large.
If only data of large magnitude is kept, the resulting image is similar but quite different from the original. Most details are wiped out.
Data Representation

- movie formats
RAW movie format

- 720×576 pixels per frame
- 24 bits (colour) per pixel
- 30 frames per second
- $30 \times 3 \times 720 \times 576 \approx 37 \text{ MB/s} \approx 135 \text{ GB/hour}$
- typically 200 GB per movie !!! ($\approx 50 \text{ DVDs}$)
MPEG2 Movie Format
MPEG2 Movie Format
MPEG2 Movie Format
MPEG2 Movie Format
MPEG2 format

- Fixed Background images
saving is about 96%
MPEG2 format

- Travelling
Image par image
MPEG2 format

- Each image is encoded with JPEG or similar.
- Sound is encoded with MP3 or similar.
- Most frames use only small amount of info to construct from previous frames.
- A complete frame is displayed every so often to make sure the fix part or travelling part has not substantially changed.