Pattern Matching in Huffman Encoded Texts

Shumel T. Klein and Dana Shapira

Presented by Thorin Tabor

Pattern Matching in Compressed Text

- Pattern matching in compressed text is a specialized case of the general pattern matching problem of finding some pattern *P* in the usually much larger text *T*.
- The usual approach to pattern matching in compressed text is for some encoding algorithm *E* and some decoding algorithm *D*, to first decompress the text and run a pattern matching on the encoded text (search for *P* in $D(E(T))$.)
- This is not always possible given the amount of space and time required for decompressing the full text and then storing the decompressed text.

The Aim of This Paper

- Instead what this paper proposes to do is to investigate and suggest an algorithm for the direct pattern matching of the encoded pattern in the encoded text (searching for $E(P)$ in $E(T)$.)
- This paper assumes that P is encoded the same way throughout the text. This is approaches the problem in terms of static Huffman encoding rather than adaptive Huffman encoding, Arithmetic encoding, etc.

Why This is Not Trivial

- The problem is not as simple as searching for all instances of $E(P)$ in $E(T)$, as not every instance of $E(P)$ will correspond to a match of P in T.
- This is due to the fact that parts of the match might cross-character encoding boundaries and this result in a mismatch.
- The problem is thus one of determining if a detected possible match is aligned with the boundaries between encoded characters.

An Example False Match

Huffman Code: {00, 010, 011, 100, 101, 1100, 1101, 111}

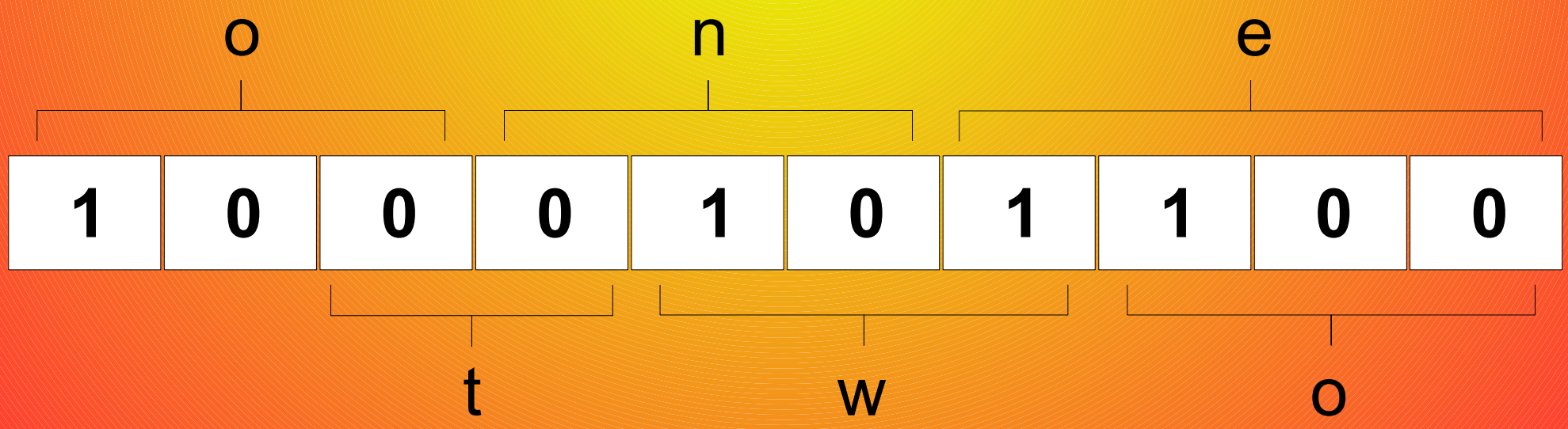
Characters: {T, N, A, O, W, E, B, C}

T = one

P = two

E(*T*) = 1000101100

E(*P*) = 00101100



One Possible Solution

- A simple solution would be to scan the encoded text from the beginning, locating all code boundaries until we come to our match.
- This, however, is a bad solution, as what we are effectively doing is decoding the entire text *T* from the beginning.
- Therefore, instead we need another solution.

Another Possible Solution

- Another possible solution is to prepare a small list of possible “entry points” into $E(T)$ where we know the entry point is a code boundary.
- This way if we want to determine if the match is aligned with code boundaries, we can simply decompress from the nearest entry point.
- This solution is bad because of the means of extracting these entry points is much like decompressing the entire text T.

The Solution Used

- If $E(P)$ has been found at index i, then jump back some constant number of bits K and start decoding from there.
- It might be that $i - K$ is not the boundary between codes, but this solution makes use of tendency of Huffman codes to resynchronize quickly after errors. Thus, if K is large enough, the boundaries ought to be determined by the time the decoding reaches i.

The Solution Used (Continued)

- Probabilistically, what's happening is that as *K* increases, the probability of a mismatch occurring at *i* decreases.
- Huffman codes tend to recover from errors quickly—typically within a 100 bits or so, therefore having a *K* of several hundred bits ought to be sufficient to reduce errors to near zero.
- Once we reach *i*, if *i* is at a code boundary we have a match. If not, we don't.

A Caveat

- If the particular Huffman code used with this approach has the *affix property*—no code is the prefix or suffix of any other code—then once synchronization is lost, it will never be regained.
- Thus, unless $i - K$ is the beginning of a code, the decoding will always be incorrect at i.
- It is of note, though, that *affix codes* are extremely rare.

The Proposed Algorithm

```
Encode P and generate E(P)
while E(T) is not empty
    i <-- search(E(P), E(T))
    if i = nil stop
    node <-- root
    for j <-- i – K to i – 1
        if jth bit of E(T) = 1
            node <-- left (node)
        else
            node <-- right (node)
        if current node is a leaf
            node <-- root
    if node = root
        declare match at address i
    delete first i bits in E(T)
```

encode the pattern to be searched
while there is encoded text left:
search for the next index i
stop if the end is reached
node points to root of tree
jump back K bits and go on
check for the beginnings and endings of the coding of characters and once we have a match then we will continue back from root to check for the next match.
if when we get to it, it is edge:
declare a match of P in T
remove that has been searched

Estimating the Number of False Matches

- To estimate the number of false matches many different Huffman codes were generated from natural language-like texts. Using these codes, many patterns *P* and texts *T* were created, and the algorithm used.
- Two types of false matches could appear: false positives and false negatives.

Types of False Matches

- False positives are times when the algorithm identifies an instance of $E(P)$ in $E(T)$ that does not correspond to P in T.
- False negatives occur when the algorithm fails to identify an instance of $E(P)$ in $E(T)$ that does correspond to P in T.

Experimental Results

K in bits	*True Positives*	*True Negatives*	*False Positives*	*False Negatives*
8	415	35	2	625
16	670	33	4	370
24	825	36	1	215
32	917	35	2	123
40	974	35	2	66
48	1013	37	0	27
56	1018	36	1	22
64	1036	37	0	4
72	1038	36	1	2
80	1036	36	1	4
88	1039	37	0	1
96	1040	37	0	0
...				
Start of file	1040	37	0	0

- The data from this table comes from an experiment run on an English corpus of text that contained editing instructions.

Experiment Results

- The experiment was run on three corpses:
  - An English corpus of editing instructions
  - A DNA file of a tobacco genome that contained six characters
  - A corpus derived from the first corpus but with each character independently and randomly generated
- As can be seen, the accuracy of the algorithm is a function of K.
- With a K of 96 bits, there was no difference in accuracy than beginning at the start of the file.