Algorithms and Data Structures

Charles A. Wuethrich

Bauhaus-University Weimar - CogVis/MMC

May 8, 2019

String searching algorithm

- Introduction
  - Brute Force (Naive) Algorithm
  - Knuth-Morris-Pratt
  - Boyer-Moore
  - Karp-Rabin
- Conclusion
- Other variants
  - Multiple Searches
  - Pattern matching, Parsing

Introduction

- Given a text string of length N and a pattern of length M, find an occurrence of the pattern whithin the text.
- The text is larger than the pattern.
- The pattern and searched text are concatenations of alphabet or binary alphabet (0,1) or DNA alphabet (A,C,G,T) in bioinformatics.
- Field of application: word processors,full-text search.

Introduction

- The pattern-matching problam can be characterized as a searching problem with the pattern as the key.
- The traditionally searching algorithms do not apply directly because the pattern can be long and because it "lines up" with the text in an unknown way.

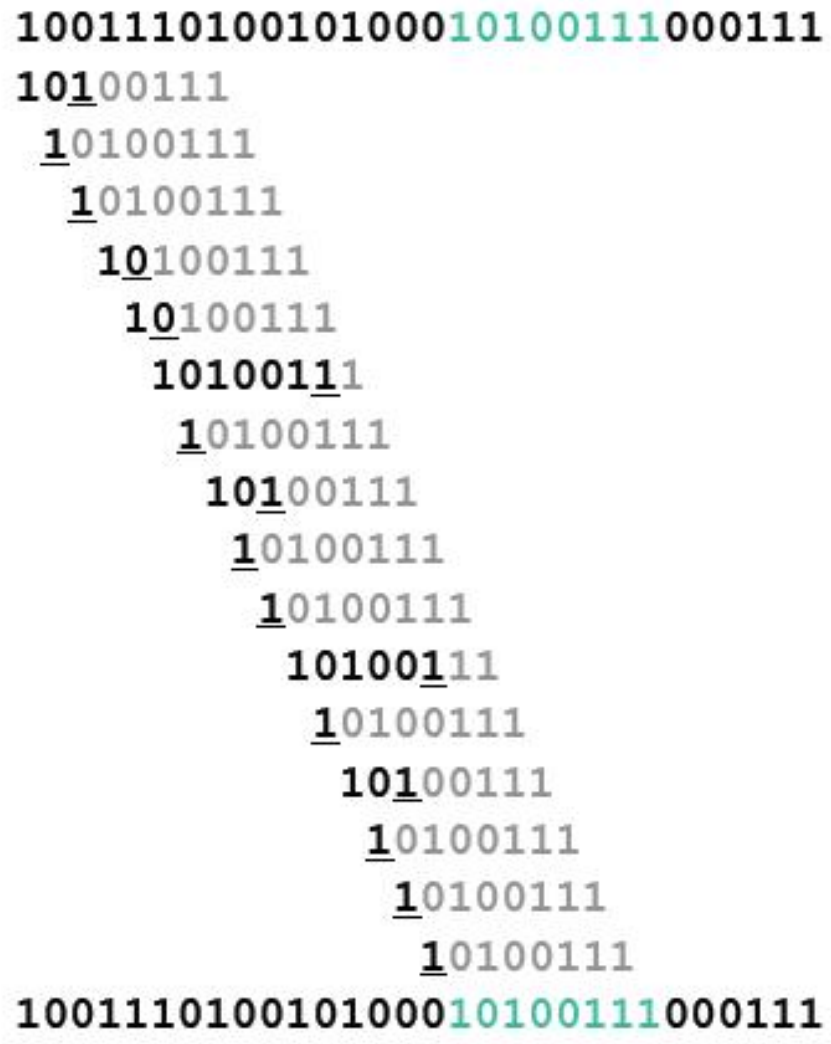
Brute Force (Naive) Algorithm

- The simplest and least efficient way to see where one pattern occurs inside a text.
- The idea consists of trying to match any substring of length M in the text with the pattern.

Brute Force (Naive) Algorithm

Example:

- N=30, M=8
- Pattern on Index 16
- 43 comparisons necessary



Brute Force (Naive) Algorithm

Pseudo-code:

```
int brutesearch(char *p, char *t)
{
  int i,j,M = strlen(p), N = strlen(t);
  for (i=0,j=0; j<M && i<N; i++,j++)
   while (t[i] != p[j]) {i-=j-1; j=0}
  if (i == M) return i-M else return i;
}
```

- The algorithm stops on either the first occurrence of the pattern, or upon reaching the end of the text.

Brute Force (Naive) Algorithm

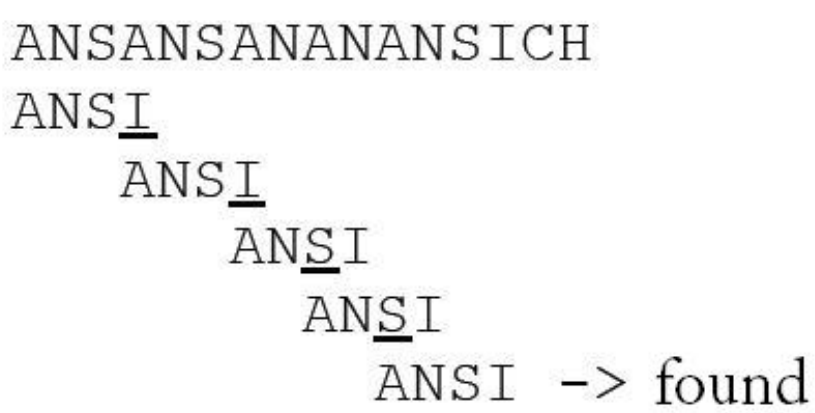
Complexity

- Worst case:
  - compares pattern to each substring of text.
  - Total number of comparisons: M (N-M+1)
  - Worst case time complexity: O(MN)
- Normal case:
  - Perhaps a few char matches occur prior to a mismatch.
  - Total number of comparisons: (N + M)
  - Normal case time complexity: O(N)

Knuth-Morris-Pratt

- Idea is to prevent index from ever going "backward" in the text string.

  Example:



Knuth-Morris-Pratt

Example:

```
MANANANASNICH
NANAS
 NANAS
  NANAS
   NANAS
        NANAS  -> The Pattern not found in Text
```

- Problem: The pattern match itself at the point of the mismatch.
- Solution: The array next[ ], which will be used to determine how far to back up when a mismatch is detected.

Knuth-Morris-Pratt

Example:

j	next[j]	
1	0	NANAS NANAS
2	0	NANAS NANAS
3	1	NANAS NANAS
4	2	NANAS NANAS

Knuth-Morris-Pratt

Pseudo-code:

```
int kmpsearch(char *p, char *t) {
  int i,j,M=strlen(p), N = strlen(t);
  initnext(p);
  for (i=0, j=0;j<M && i<N; i++, j++)
    while ((j>=0) && (a[i] != p[j])) j = next[j];
  if (j == M ) return i-M; else return i;
}

initnext(char *p) {
  int i, j, M = strlen(p); next[0] = -1;
  for (i=0, j=-1;i<M; i++,j++ next[i]=j)
    while ((j>=0) && p[i] != p[j])) j=next[j];
}
```

Knuth-Morris-Pratt

Complexity

- Improves the worst case of the Brute Force (Naive) Algorithm, but not the normal case.
- Worst case time complexity: $O(N + M)$
- However, since $M \ll N$ one can say that complexity: $O(N)$

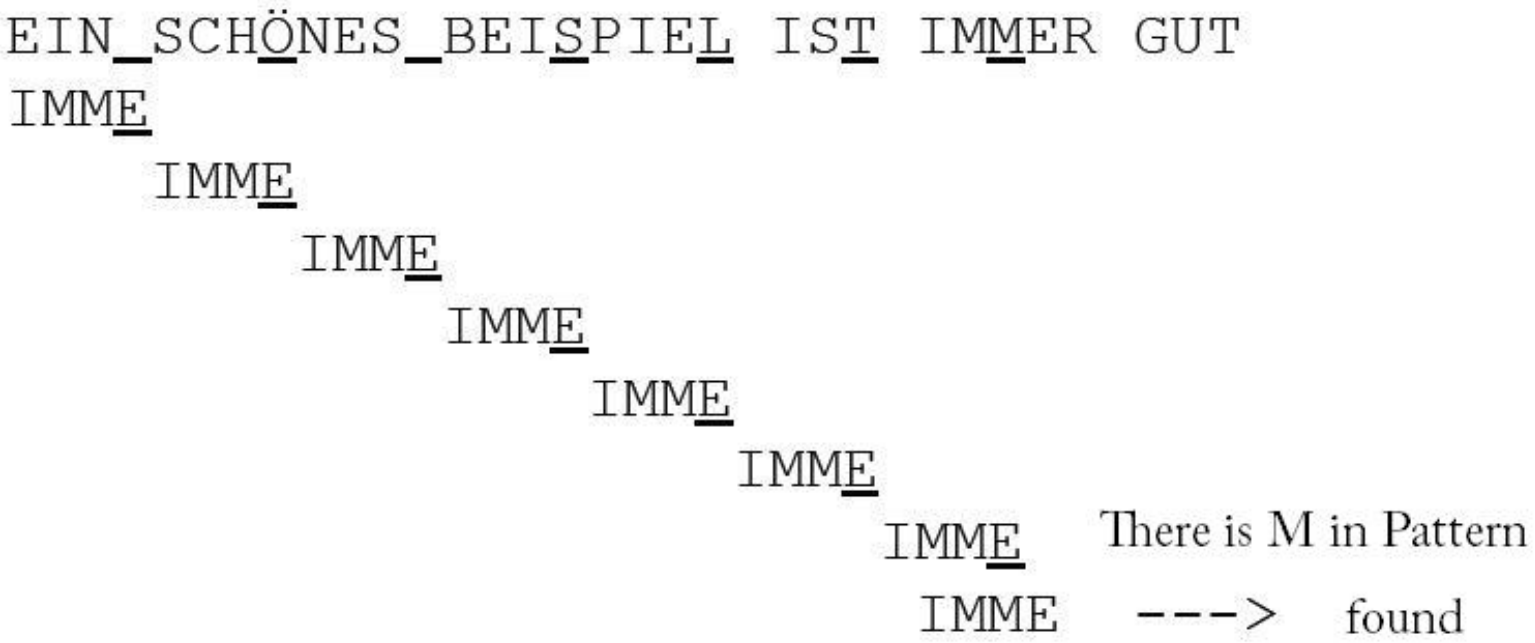
Boyer-Moore

Idea:

- Look at the pattern from right to left instead of left to right.
- Now, if a character is early mismatched, we have the potential to skip many characters with only one comparison.

Boyer-Moore

Example:



Boyer-Moore

Pseudo-code:

```
int heuristicsearch(char *p, char *t) {
   int i,j,k,M = strlen(p), N = strlen(t);
   initskip(p);
   for (i=M-1,j= M-1; j>0; i--,j--)
     while (t[i] != p[j]) {
        k = skip[index(a[i])];
        i += (M-j >k) ? M-j: k;
        if (i>=N) return (N);
        j = M-1;
     }
   return i;
}
```

Boyer-Moore

- *index()*: Is a function that takes a character as an argument and returns 0 for blanks and *i* for the *i* th letter of the alphabet.
- *initskip()*: Initializes the *skip* array to M for characters not in the pattern and then, for j from 0 to M-1, sets *skip[index(P[j])]* to M-j-1.

Boyer-Moore

Complexity

- Worst case: Boyer-Moore algorithm is linear in the same way as the Knuth-Morris: O (M+N)
- Normal case: String searching uses about N/M steps if the pattern is not long and the alphabet is not small.
- Problem: The algorithm won't help much for binary strings, because there are only two possibilities for characters.
- Solution: The bits can be grouped together to make"characters".

Karp-Rabin

- Use hashing
- The algorithm calculates a hash value for the pattern, and for each M-character subsequence of text to be compared.
  - If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence.
  - If the hash values are equal, the algorithm will do a Brute Force comparison between the pattern and the M-character sequence.

Karp-Rabin

Hash function

- $h(k)=k \ MOD \ q$
- q is a large prime. In this case, nothing will be stored in the hash table, so q can be taken to be very large.
- Consider an M-character sequence as an M-digit number in base d, where d is the number of letters in the alphabet.
- The text subsequence $a[i \ .. \ i+M-1]$ is mapped to the number k.

Karp-Rabin

- $k = a[i]d^{M-1} + a[i+1]d^{M-2} + ... + a[i+M-1]$
- Assume that we know the value of *h(k)=k MOD q*. But shifting one position right in the text simply corresponds to replacing k by $(k - a[i]d^{M-1})d + a[i+M]$

Karp-Rabin

Pseudo-code:

```
#define q 33554393
#define d 32
int krsearch(char *p, char *t) {
  int i, dM = 1, h1=0, h2=0;
  int M = strlen(p), N = strlen(t);
  for (i=1;i<M; i++) dM = (d*dM) % q;
  for (i=0;i<M; i++) {
    h1 = (h1*d+index(p[i])) % q;
    h2 = (h2*d+index(t[i])) % q; }
  for (i=0; h1!=h2; i++) {
    h2 = (d*q+h2-index(t[i])*dM) % q;
    h2 = (h2*d+index(t[i+M])) % q;
    if (i>N-M) return N; }
  return i; }
```

Karp-Rabin

- *h1*: The hash value for the pattern.
- *h2*: The hash value for the first M characters of the text.
- The program proceeds through the text string and compute the hash function for M characters starting at position i for each i and comparing each new hash value to *h1*.

Karp-Rabin

Complexity

- The algorithm takes time proportional to N+M.
- Note that it only finds a position in the text that has the same hash value as the pattern.
- The use of the very large value of q makes it extremely unlikely that a collision will occur.
- In the (unbelievably) worst case the algorithm could take O(NM) steps.

Conclusion

Algorithms	averagecase	Preferred implementation
Brute Force	O(N)	simple implementation
Knuth-Morris	O(N)	high repeating in text and pattern
Boyer-Moore	O(N/M)	long alphabet
Karp-Rabin	O(N)	worst case O(NM) happens rarely

Multiple Searches

- The text string is treated as N overlapping "keys".
- The *i* th key defined to be *a[i]*, ... , *a[N]*, the entire text string starting at position *i*.
- Advantage: Constant searching time.
- disadvantage:
  - Time-consuming preparation
  - Not generally applicable

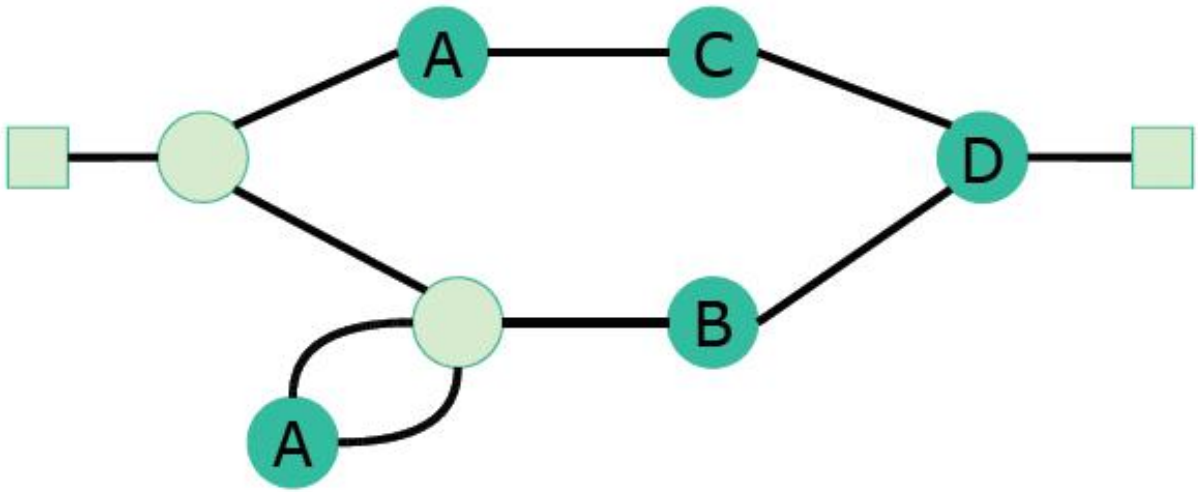
Pattern matching

Describing patterns

- *Concatenation*: If two characters are adjacent in the pattern, then there is a match if and only if the same two characters are adjacent in the text.
  - AB means A followed B
- *Or* : There is a match if and only if either of the characters occurs in the text.
  - A+B means either A or B
- *Closure*: There is a match if and only if the symbol occurs any number of times (including 0)
  - AB* means strings consisting of an A followed by any number of B's.

Pattern matching

- A nondeterministic finite-state machine that could be used to search for the pattern description *(A*B+AC)D* in a text string.



- Last step: Simulate the operation of a non-deterministic pattern-matching machine.

Parsing

- Pattern matching is the basic for the parsing.
- Parsing is directly related to the study of the structure of language in general.
- Case of interest: Translating from a "high-level" computer language like C to a "Low-Level" assembly or machine language.
- Two general approaches are used for parsing:
  - Top-down methods
  - Bottom-up methods