Bitap algorithm

From Wikipedia, the free encyclopedia

The **bitap algorithm** (also known as the **shift-or**, **shift-and** or **Baeza-Yates–Gonnet** algorithm) is an approximate string matching algorithm. The algorithm tells whether a given text contains a substring which is "approximately equal" to a given pattern, where approximate equality is defined in terms of Levenshtein distance — if the substring and pattern are within a given distance *k* of each other, then the algorithm considers them equal. The algorithm begins by precomputing a set of bitmasks containing one bit for each element of the pattern. Then it is able to do most of the work with bitwise operations, which are extremely fast.

The bitap algorithm is perhaps best known as one of the underlying algorithms of the Unix utility agrep, written by Udi Manber, Sun Wu, and Burra Gopal. Manber and Wu's original paper gives extensions of the algorithm to deal with fuzzy matching of general regular expressions.

Due to the data structures required by the algorithm, it performs best on patterns less than a constant length (typically the word length of the machine in question), and also prefers inputs over a small alphabet. Once it has been implemented for a given alphabet and word length *m*, however, its running time is completely predictable — it runs in O(*mn*) operations, no matter the structure of the text or the pattern.

The bitap algorithm for exact string searching was invented by Bálint Dömölki in 1964[1][2] and extended by R. K. Shyamasundar in 1977[3], before being reinvented in the context of fuzzy string searching by Manber and Wu in 1991[4][5] based on work done by Ricardo Baeza-Yates and Gaston Gonnet[6]. The algorithm was improved by Baeza-Yates and Navarro in 1996[7] and later by Gene Myers for long patterns in 1998[8].

Exact searching [edit]

The bitap algorithm for exact string searching, in full generality, looks like this in pseudocode:

```
algorithm bitap_search(text : string, pattern : string) returns string
    m := length(pattern)

    if m == 0
        return text

    /* Initialize the bit array R. */
    R := new array[m+1] of bit, initially all 0
    R[0] = 1

    for i = 0; i < length(text); i += 1:
        /* Update the bit array. */
        for k = m; k >= 1; k -= 1:
          R[k] = R[k-1] & (text[i] == pattern[k-1])

        if R[m]:
            return (text+i - m) + 1

    return nil
```

Bitap distinguishes itself from other well-known string searching algorithms in its natural mapping onto simple bitwise operations, as in the following modification of the above program. Notice that in this implementation, counterintuitively, each bit with value zero indicates a match, and each bit with value 1 indicates a non-match. The same algorithm can be written with the intuitive semantics for 0 and 1, but in that case we must introduce another instruction into the inner loop to set `R |= 1`. In this implementation, we take advantage of the fact that left-shifting a value shifts in zeros on the right, which is precisely the behavior we need.

Notice also that we require `CHAR_MAX` additional bitmasks in order to convert the `(text[i] == pattern[k-1])` condition in the general implementation into bitwise operations. Therefore, the bitap algorithm performs better when applied to inputs over smaller alphabets.

```
#include <string.h>
```

```
#include <limits.h>

const char *bitap_bitwise_search(const char *text, const char *pattern)
{
    int m = strlen(pattern);
    unsigned long R;
    unsigned long pattern_mask[CHAR_MAX+1];
    int i;

    if (pattern[0] == '\0') return text;
    if (m > 31) return "The pattern is too long!";

    /* Initialize the bit array R */
    R = ~1;

    /* Initialize the pattern bitmasks */
    for (i=0; i <= CHAR_MAX; ++i)
      pattern_mask[i] = ~0;
    for (i=0; i < m; ++i)
      pattern_mask[pattern[i]] &= ~(1UL << i);

    for (i=0; text[i] != '\0'; ++i) {
        /* Update the bit array */
        R |= pattern_mask[text[i]];
        R <<= 1;

        if (0 == (R & (1UL << m)))
          return (text + i - m) + 1;
    }

    return NULL;
}
```

Fuzzy searching [edit]

To perform fuzzy string searching using the bitap algorithm, it is necessary to extend the bit array *R* into a second dimension. Instead of having a single array *R* that changes over the length of the text, we now have *k* distinct arrays $R_{1..k}$. Array R_i holds a representation of the prefixes of *pattern* that match any suffix of the current string with *i* or fewer errors. In this context, an "error" may be an insertion, deletion, or substitution; see Levenshtein distance for more information on these operations.

The implementation below performs fuzzy matching (returning the first match with up to *k* errors) using the fuzzy bitap algorithm. However, it only pays attention to substitutions, not to insertions or deletions — in other words, a Hamming distance of *k*. As before, the semantics of 0 and 1 are reversed from their intuitive meanings.

```
#include <stdlib.h>
#include <string.h>
#include <limits.h>

const char *bitap_fuzzy_bitwise_search(const char *text, const char *pattern, int
k)
{
    const char *result = NULL;
    int m = strlen(pattern);
    unsigned long *R;
    unsigned long pattern_mask[CHAR_MAX+1];
    int i, d;

    if (pattern[0] == '\0') return text;
    if (m > 31) return "The pattern is too long!";

    /* Initialize the bit array R */
    R = malloc((k+1) * sizeof *R);
    for (i=0; i <= k; ++i)
        R[i] = ~1;

    /* Initialize the pattern bitmasks */
    for (i=0; i <= CHAR_MAX; ++i)
        pattern_mask[i] = ~0;
    for (i=0; i < m; ++i)
```

```
        pattern_mask[pattern[i]] &= ~(1UL << i);

    for (i=0; text[i] != '\0'; ++i) {
        /* Update the bit arrays */
        unsigned long old_Rd1 = R[0];

        R[0] |= pattern_mask[text[i]];
        R[0] <<= 1;

        for (d=1; d <= k; ++d) {
            unsigned long tmp = R[d];
            /* Substitution is all we care about */
            R[d] = (old_Rd1 & (R[d] | pattern_mask[text[i]])) << 1;
            old_Rd1 = tmp;
        }

        if (0 == (R[k] & (1UL << m))) {
            result = (text+i - m) + 1;
            break;
        }
    }

    free(R);
    return result;
}
```

External links and references [edit]

1. ^ Bálint Dömölki, An algorithm for syntactical analysis, Computational Linguistics 3, Hungarian Academy of Science pp. 29–46, 1964.
2. ^ Bálint Dömölki, A universal compiler system based on production rules, BIT Numerical Mathematics, 8(4), pp 262–275, 1968. doi:10.1007/BF01933436
3. ^ R. K. Shyamasundar, Precedence parsing using Dömölki's algorithm, International Journal of Computer Mathematics, 6(2)pp 105–114, 1977
4. ^ Udi Manber, Sun Wu. "Fast text searching with errors." Technical Report TR-91-11. Department of Computer Science, University of Arizona, Tucson, June 1991. (gzipped PostScript)
5. ^ Udi Manber, Sun Wu. "Fast text search allowing errors." *Communications of the ACM*, 35(10): pp. 83–91, October 1992, doi:10.1145/135239.135244.
6. ^ Ricardo A. Baeza-Yates, Gastón H. Gonnet. "A New Approach to Text Searching." *Communications of the ACM*, 35(10): pp. 74–82, October 1992.
7. ^ R. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In Dan Hirchsberg and Gene Myers, editors, *Combinatorial Pattern Matching* (CPM'96), LNCS 1075, pages 1–23, Irvine, CA, June 1996.
8. ^ G. Myers. "A fast bit-vector algorithm for approximate string matching based on dynamic programming." *Journal of the ACM* 46 (3), May 1999, 395–415.
9. libbitap, a free implementation that shows how the algorithm can easily be extended for most regular expressions. Unlike the code above, it places no limit on the pattern length.
10. Baeza-Yates. *Modern Information Retrieval*. 1999. ISBN 0-201-39829-X.
11. bitap.py - Python implementation of Bitap algorithm with Wu-Manber modifications.

Categories: String matching algorithms

This page was last modified on 30 July 2015, at 08:55.

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.

Privacy policy About Wikipedia Disclaimers Contact Wikipedia Developers Mobile view