CH08-320143: Assignment #13

Harit Krishan
h.krishan@jacobs-university.de

Jacobs University Bremen — May 10, 2020

1 Backtracking: n Horses Problem

2 Rabin-Karp String-Matching Algorithm

This algorithm makes use of elementary number-theoretic notions such as the equivalence of two numbers modulo a third. The Rabin-Karp string matching algorithm shares features with that of a hash table. The hashing technique is used to cut down the required number of computations.

2.1 Example

Assume we are given the string $< abedabc >$ and the task is to find the substring $< abc >$ in the given string. Since the pattern has a length of 3, we will use a hash function which takes a substring of length 3 and returns a hash key. This key will later be used to find the match. Assume the following encoding for the characters:

$$< a = 1, b = 2, c = 3, ..., z = 26 >$$

It is recommended to use a prime number close to the length of the substring to search. For this example, we will use 3. The following substrings will be checked:

$$< abe > , < bed >, < eda >, < dab >, < abc >$$

For each substring, a hash key will be generated as such:

$$\begin{aligned} val[0] * prime^{(0)} &+ val[1] * prime^{(1)} + ... \\ = \quad 1 * 3^0 &+ 2 * 3^1 + 5 * 3^2 \\ &= 52 \end{aligned}$$

This way we will compute the hash key for each of the substrings listed. The last check will be a match, something like this:

$$\begin{aligned} val[4] * prime^{(0)} &+ val[5] * prime^{(1)} + ... \\ = \quad 1 * 3^0 &+ 2 * 3^1 + 3 * 3^2 \\ &= 34 \end{aligned}$$

The matched hash key is 34. The indices of the occurrences of the given substring within the text will be returned. This is a simple demonstration of how this algorithm works.

2.2 Implementation

For implementing this algorithm, I used C++. I implement a function with void return type taking the text, pattern & the 'prime' value to use for the hashing. We define integer variables to store the respective sizes of the text and pattern. A snippet of the implementation of this algorithm can be seen in Figure 1.

```
#define d 256   // input alphabet -> ascii encoding

void rk_search(char pat[], char txt[], int q)  {
    // store string lengths
    int M = strlen(pat);
    int N = strlen(txt);

    int i, j;

    int p = 0; // initial pattern hash
    int t = 0; // initial text hash
    int h = 1;

    for (i = 0; i < M - 1; i++)  {
        h = (h * d) % q;
    }


    for (i = 0; i < M; i++)  {
        p = (d * p + pat[i]) % q;
        t = (d * t + txt[i]) % q;
    }

    // bruteforce check
    for (i = 0; i <= N - M; i++)  {
        if ( p == t )  {                    // check one by one
            for (j = 0; j < M; j++)  {
                if (txt[i+j] != pat[j]) {
                    break;
                }
            }
            if (j == M)  {
                cout<<"Match at index "<< i<< "!" << endl;
            }
        }

        if ( i < N-M )  {
            t = (d*(t - txt[i]*h) + txt[i+M])%q;
            if (t < 0)  {
                t = (t + q);            // in case negative value
            }
        }
    }
}
```

Figure 1: Implementation of Rabin-Karp String Search in C++