EL 7373 High Performance Switches & Routers

Lab 3 report

Single String Matching: Boyer-Moore Algorithm

group : G16

Yu chen yc1595

Yuqin Wang yw1724

Peixuan Ding pd1178

1. Pseudo codes

1.1 Naïve algorithm

```
MatchResult naive_str_match(const char* t, const char* p)
{
	/* t is the string for text; p is the string for pattern */
	size_t	m = strlen(t);
	size_t	n = strlen(p);

	/* declare a return structure and initialize */
	MatchResult ret;
	ret.n_match = 0;
	ret.n_compare = 0;

	/* Loop of comparison */
	for(size_t i = 0 ; i <= (m-n) ; i++)
	{
		int match = 1;

		for(size_t j=0 ; j < n ; j++)
		{
			ret.n_compare ++;
			if( p[j] != t[i+j])
			{
				match = 0;
				break;
			}
		}

		if( match == 1 )
		{
			ret.n_match ++;
			printf("A pattern match is found at location %lu.\n", i);
		}
	}

	return ret;
}
```

1.2 Boyer-Moore Algorithm

1.2.1 generate bad character form

```
void make_badCharacter(int *badCharacter, const char* p) {

    size_t   n = strlen(p);
    int i;
    for (i=0; i < ALPHABET_LEN; i++) {
        badCharacter[i] = n;
    }
    for (i=0; i < n-1; i++) {
        badCharacter[p[i]] = n-1 - i;
    }
}
```

explanation:

when mismatch happens,

1.if the text character does not exist in pattern.shift the pattern by pattern's length

```
text        abcdefghijklmn
pattern     abxde
              abxde
```

shift by pattern length

2. If the text character does exist in pattern. shift the pattern by badCharacter[p[i]] , which is the least distance between character p[i] and p[n-1]

```
text        ahcndscmssdddd
pattern     abcdcexm
              abcdce
```

shift by the rightmost character 'c'

1.2.2 generate good suffix form

```
int is_prefix(const char* p, int pos) {
	size_t	n = strlen(p);
	int i;
	int suffixlen =n - pos;
	for (i = 0; i < suffixlen; i++) {
		if (p[i] != p[pos+i]) {
			return 0;
		}
	}
	return 1;
}
```

explanation:
the function of "is_prefix" is to check whether prefix from beginning to current position is the same with suffix which has same length;
abcdfgrab

```
int suffix_length(const char* p, int pos) {
	size_t	n = strlen(p);
	int i;
	// increment suffix length i to the first mismatch or beginning
	// of the word
	for (i = 0; (p[pos-i] ==p[n-1-i]) && (i < pos); i++);
	return i;
}
```

explanation:
the function of "suffix_length" is to find the longest match of good suffix

```
text		**********babade**********
pattern			abcdefgbade
				abcdefgbade
```

```
void make_goodSuffix(int *goodSuffix, const char* p) {
	size_t	n = strlen(p);
	int q;
	int last_prefix_index = n-1;

	// first loop
	for (q=n-1; q>=0; q--) {
```

```
            if (is_prefix(p, q+1)) {
                last_prefix_index = q+1;
            }
            goodSuffix[q] = last_prefix_index + (n-1 - q);
        }

        // second loop
        for (q=0; q < n-1; q++) {
            int slen = suffix_length(p,q);
            if (p[q - slen] != p[n-1 - slen]) {
                goodSuffix[n-1 - slen] = n-1 - q + slen;
            }
        }
}
```

explanation:

make good suffix form by two step, using the above two functions separately.

1.2.3 execute the Boyer_Moore algorithm

```
MatchResult Boyer_Moore(const char* t, const char* p){
    size_t    m = strlen(t);
    size_t    n = strlen(p);
    int i;
    int badCharacter[ALPHABET_LEN];
    int *goodSuffix = (int *)malloc(n * sizeof(int));
    make_badCharacter(badCharacter, p);
    make_goodSuffix(goodSuffix, p);

    MatchResult ret;
    ret.n_match = 0;
    ret.n_compare = 0;
    i = n - 1; // text location
    while (i < m) {
    int j = n - 1; // pattern location
    while (j >= 0 && (t[i] == p[j])) {
        ret.n_compare++;
        i--;
        j--;
        };
        if (j < 0) {
        ret.n_match++;
```

```
            printf("A pattern match is found at location %d.\n", i+1);
            i += max(badCharacter[t[i + 1]], goodSuffix[j + 1]) + 1;
            }
            else{
            ret.n_compare++;
            i += max(badCharacter[t[i]], goodSuffix[j]);
            }
        }
        free(goodSuffix);
        return ret;
}
```

2. Performance Comparison

Execution time	naive		Boyer-Moore	
	1000	10000	1000	10000
anpanman	0.014200	0.034225	0.005808	0.027997
tobeornottobe	0.013744	0.036871	0.004936	0.027165
tobeornottobethatisthequestion	0.014504	0.038277	0.004387	0.02662

Number of character comparisons made	naive		Boyer-Moore	
	1000	10000	1000	10000
anpanman	1099937	1196416	160992	242146
tobeornottobe	1126756	1273137	122257	250864
tobeornottobethatisthequestion	1263018	1578412	86461	353426

3. Theoretical Analysis

(m is the length of text, n is the length of pattern.)

Best Case：$O(m/n)$
Best case will happens, if the pattern and text are not match at left most character. At the same time, the pattern shift pattern-length.

Worst Case: $O(mn)$
For example, pattern = aaa, text = aaaaaaaaaaaa. For each comparison, the pattern and the text compare n-1 character. Then, the pattern shift 1 after each comparison. The text shift m-n times in total. $O(mn - n^2 - m + n) = O(mn)$

Average Case: $O\ (m)$
In average case, we assume the pattern and the text compare n/2 character and then mismatch. After that, the pattern shifts half of pattern length. $O(\frac{m}{n/2} \times \frac{n}{2})$