EE 3980 Algorithms

Homework 2. Random Data Searches

105061110 周柏宇

2020/3/22

1. Introduction

In this homework, we implemented 3 search algorithms: linear search, bidirection search and random-direction search. To evaluate the average and worse-case performance, we run each algorithm on lists of randomly-ordered English words with different numbers of entries. In the end, we plot out the CPU time against the input size to observe the trend and to verify our analysis of the complexity of these algorithms.

2. Analysis & Implementation

2.1 Linear search

Algorithm input and output parameters should be explained.

```
Algorithm Search(word, list, n)         // Linear Search
{
    for i := 1 to n do {                // compare all possible entries
        if (list[i] = word) return i;
    }
    return -1;                          // unsuccessful search
}
```

Font size can be larger.

In the linear search, we compare the entry of the list from head to the end. The best case occurs when the target word is the very first word of the list, while the worst case occurs when the target word is the last word of the list. If we search all words in the list, on average, it requires $\frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}$ times of comparison. As for the space complexity, we need n space for the list.

Best-case time complexity: $\mathcal{O}(1)$

Average time complexity: $\mathcal{O}(n)$

Worst-case time complexity: $\mathcal{O}(n)$

Space complexity: $\mathcal{O}(n)$

2.2 Bidirection search

```
Algorithm BDSearch(word, list, n)       // Bidirection Search
{
    for i := 1 to n / 2 do {            // compare entries from both sides
        if (list[i] = word) return i;
        if (list[n - i - 1] = word) return n - i - 1;
    }
    return -1;                          // unsuccessful search
}
```

In bidirection search, we compare the entries of the list from both directions. Therefore, the best case occurs when the target word is the first word of the list; the worst case occurs when the target word is in the center of the list. Therefore, we need to compare n times before finding it. The average performance is the same as the linear search when we search all words of the list, because we only change the searching order.

Best-case time complexity: $\mathcal{O}(1)$

Average time complexity: $\mathcal{O}(n)$

Worst-case time complexity: $\mathcal{O}(n)$

Space complexity: $\mathcal{O}(n)$

2.3 Random-direction search

```
Algorithm RDSearch(word, list, n)
{
    choose j from {0, 1} randomly;      // randomly select a direction
    if (j = 1) then                     // forward linear search
        for i := 1 to n do {
            if (list[i] = word) return i;
        }
    else                                // backward linear search
        for i := n to 1 do {
            if (list[i] = word) return i;
        }
    return -1;                          // unsuccessful search
}
```

In random-direction search, we first choose a direction and then perform the linear search either forward or backward according to the direction. Same as previous search algorithms, we need as least one and at most n comparisons to locate the target word. For the target word being the ith word in the list, assuming the direction is chosen uniformly at random, we expect to compare $0.5 * i + 0.5 * (n - i + 1) = \frac{n+1}{2}$ times.

Best-case time complexity: $\mathcal{O}(1)$

Average time complexity: $\mathcal{O}(n)$

Worst-case time complexity: $\mathcal{O}(n)$

Space complexity: $\mathcal{O}(n)$

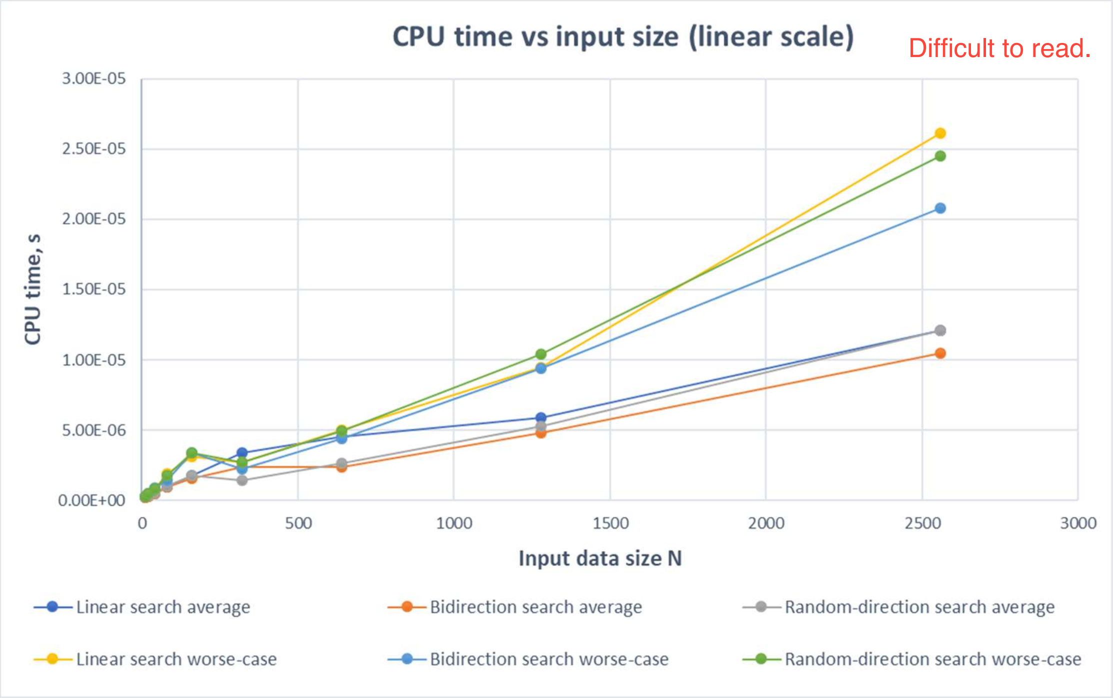
3. Result and Observation

To measure the average performance, we search all possible words in the list and average the CPU time over the number of words as average performance. As for the worse-case performance, we choose to search the word that requires most comparisons for each search algorithm. To exhibit and compare the growth of CPU time, we run three search algorithms with different numbers of input for several times. Specifically, the average performance is averaged over 500 trials and the worse-case performance is averaged over 5000 trials.

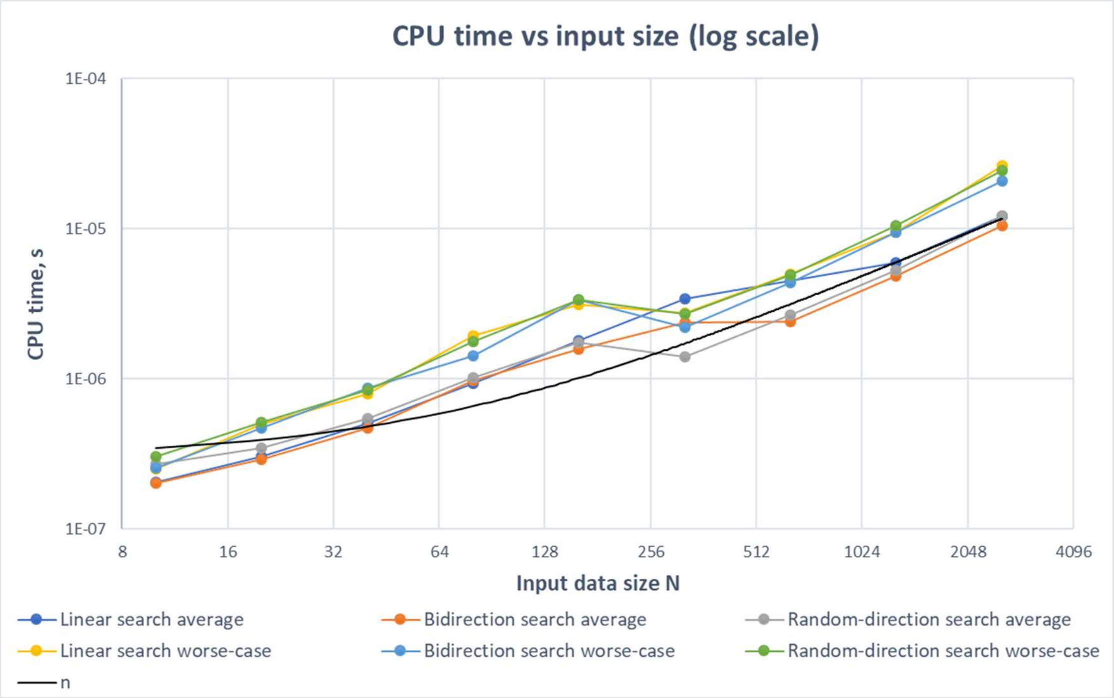
N	*Average*			*Worse-case*		
	Linear search	*Bidirection search*	*Random-direction search*	*Linear search*	*Bidirection search*	*Random-direction search*
10	0.206375	0.200796	0.270987	0.250006	0.25382	0.304413
20	0.305009	0.292397	0.346279	0.501442	0.469017	0.512171
40	0.506604	0.473344	0.548053	0.80018	0.861788	0.837994

80	0.930452	0.97487	1.01422	1.93582	1.41358	1.78337
160	1.78876	1.56901	1.75746	3.11975	3.37939	3.3812
320	3.40464	2.36826	1.39842	2.74181	2.21658	2.68579
640	4.51523	2.40808	2.66012	4.99859	4.39959	4.93641
1280	5.89816	4.81463	5.28454	9.4604	9.413	10.4376
2560	12.0937	10.4677	12.0814	26.1076	20.804	24.4856

Table 1. CPU time [μ s] vs input data size N



Difficult to read.



According to our analysis, we expect the performance to be the same in terms of the average and worst case since the only difference between them is the order of searching, and it roughly holds. The bidirection search seems to be a little bit faster due to our implementation which requires less operation at loop comparison. We also expect the worse-case performance to be around two times slower than that of the average case. In Table 1, for data set with a larger number of words, we certainly can see the relationship.

Why?

Since the average and worse-case time complexity are $O(n)$ for all three search algorithms, in the linear scale scatter plot, we can observe the linear trend for all the lines if we ignore the fluctuation at small input sizes. In the log scale scatter plot, we can compare the slope with the linear line to show that they are indeed linear, and the difference between average and worse-case lines is only the bias.

hw02.c

```c
// EE3980 HW02 Random Data Searches
// 105061110, 周柏宇
// 2020/03/22

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>

int N;              // input size
int R_AVG = 500;    // number of repititions for testing average case
int R_WORSE = 5000; // number of repititions for testing worse-case
int WORSE_CASE = 0; // flag for testing worse-case
char **data;        // input data
int (*search[3])(char *word, char **list, int n);   // store function pointers
char algNames[][17] = {                             // name of algorithms
    "Linear", "Bidirection", "Random-direction"
};

void readInput(void);                           // read all inputs
double GetTime(void);                           // get local time in seconds
int Search(char *word, char **list, int n);     // Linear Search
int BDSearch(char *word, char **list, int n);   // Bidirection Search
int RDSearch(char *word, char **list, int n);   // Random-direction Search
void freeMemory(char **list, int n);            // free allocated memory

int main(void)
{
    int i, j, k;                // index
    double t;                   // local time
    int worseIdx[3];            // index for worse-case word

    readInput();                // store inputs in array
    printf("n: %d\n", N);       // print input size
    search[0] = Search;         // store function pointers
    search[1] = BDSearch;
    search[2] = RDSearch;
    worseIdx[0] = N - 1;        // store worse-case index
    worseIdx[1] = N / 2;        // store worse-case index
    worseIdx[2] = 0;            // worse-case undetermined, initialize as 0
    for (i = 0; i < 3; i++) {
        t = GetTime();          // get local time
        for (j = 0; j < R_AVG; j++) {
            for (k = 0; k < N; k++) {           // list all words
                search[i](data[k], data, N);    // execute search algorithm
            }
        }
        t = (GetTime() - t) / (R_AVG * N);      // calculate average CPU time
        printf("%s search average CPU time: %.5e\n", algNames[i], t);
                                        // print algorithm name and CPU time
    }
    WORSE_CASE = 1;                     // testing for worse-case
    for (i = 0; i < 3; i++) {
        t = GetTime();                  // get local time
        for (j = 0; j < R_WORSE; j++) {
            search[i](data[worseIdx[i]], data, N); // execute search algorithm
        }
        t = (GetTime() - t) / R_WORSE;  // calculate average CPU time
        printf("%s search worse-case CPU time: %.5e\n", algNames[i], t);
                                        // print algorithm name and CPU time
    }
    freeMemory(data, N);                // free array data

    return 0;
}

void readInput(void)                // read all inputs
{
    int i;                          // index
    char tmpWord[1000];             // store input temporarily

    scanf("%d", &N);                // input number of entries
    data = (char **)malloc(sizeof(char *) * N); // allocate memory for pointers
    for (i = 0; i < N; i++) {
        scanf("%s", tmpWord);       // input a word
        // allocate memory just enough to fit the word
        data[i] = (char *)malloc(sizeof(char) * (strlen(tmpWord) + 1));
        strcpy(data[i], tmpWord);   // transfer the input to array
    }
}

double GetTime(void)                        // get local time in seconds
{
    struct timeval tv;                      // variable to store time

    gettimeofday(&tv, NULL);                // get local time

    return tv.tv_sec + 1e-6 * tv.tv_usec;   // return local time in seconds
}

int Search(char *word, char **list, int n)  // Linear Search
{
    int i;                      // index

    for (i = 0; i < n; i++) {   // compare all possible entries
        if (strcmp(list[i], word) == 0) {
            return i;
        }
```

```
 99     }
100 
101     return -1;                  // unsuccessful search
102 }
103 
104 int BDSearch(char *word, char **list, int n)    // Bidirection Search
105 {
106     int i;                                      // index
107 
108     for (i = 0; i < n / 2; i++) {
109         if (strcmp(list[i], word) == 0) {       // compare words from head
110             return i;
111         }
112         if (strcmp(list[n - i - 1], word) == 0) { // compare words from tail
113             return n - i - 1;
114         }
115     }
116 
117     return -1;                                  // unsuccessful search
118 }
119 
120 int RDSearch(char *word, char **list, int n)    // Random-direction Search
121 {
122     int j, i;               // index
123 
124     j = rand() % 2;         // choose direction randomly
125     // if we are testing worse-case but the target is not the worse-case word
126     if (WORSE_CASE && j) {
127         word = list[n - 1]; // change the target to the worse-case word
128     }
        lines 126-128 not in pseudo codes.
129     if (j == 1) {           // forward linear search
130         for (i = 0; i < n; i++) {
131             if (strcmp(list[i], word) == 0) {
132                 return i;
133             }
134         }
135     }
136     else {                  // backward linear search
137         for (i = n - 1; i >= 0; i--) {
138             if (strcmp(list[i], word) == 0) {
139                 return i;
140             }
141         }
142     }
143 
144     return -1;              // unsuccessful search
145 }
146 
147 void freeMemory(char **list, int n) // free allocated memory
```

```
{
    int i;                          // index

    for (i = 0; i < n; i++) {
        free(list[i]);              // free the memory that stores the words
    }
    free(list);                     // free the memory that stores the pointers
}
```

[Coding] hw02.c spelling errors: repititions(2)
[RDsearch] implementation is different from the pseudo code.
[Font] size of pseudo code can be larger.
[Pseudo] codes can still be improved.
[Average and worst-case] time complexities can be more clearly described.
[Figures] can be improved.

Score: 75