PROJECT REPORT

Submitted by

V. S. DHARINEESH (RA2111003010008)
PREETHI. V (RA2111003010042)

for the course 18CSC204J Design and Analysis of Algorithms

Under the guidance of
Dr. SARANYA SURESH
Assistant Professor, Department of Computing Technologies

in partial fulfillment for the award of the degree

of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

of

FACULTY OF ENGINEERING AND TECHNOLOGY

S.R.M. Nagar, Kattankulathur, Chengalpattu District

APRIL 2023

SRM UNIVERSITY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this project report titled "Implementing Directory Search using Greedy and Dynamic Technique" is the bonafide work of V. S. DHARINEESH (RA2111003010008) and V. PREETHI (RA2111003010042) who carried out the project work under our supervision. Certified further, that to the best of our knowledge, the work reported herein does not form any other project report or dissertation based on which a degree or award was conferred on an earlier occasion on this or any other candidate.

SIGNATURE
Dr. P. SARANYA SURESH
Assistant Professor

ACKNOWLEDGEMENT

We express our heartfelt thanks to our honorable **Vice Chancellor Dr. C. MUTHAMIZHCHELVAN**, for being the beacon in all our endeavors. We would like to express my warmth of gratitude to our **Registrar Dr. S. Ponnusamy,** for his encouragement. We express our profound gratitude to our **Dean (College of Engineering and Technology) Dr. T. V. Gopal,** for bringing out novelty in all executions. We would like to express our heartfelt thanks to the Chairperson, School of Computing **Dr. Revathi Venkataraman,** for imparting confidence to complete my course project.

We are highly thankful to my Course project Faculty **Dr. P. Saranya Suresh Assistant Professor,** for her assistance, timely suggestion, and guidance throughout the duration of this course project.

Finally, we thank our parents and friends near and dear ones who directly and indirectly contributed to the successful completion of our project. Above all, I thank the almighty for showering his blessings on us to complete our Course project.

Design and Analysis of Algorithms
Group Project

Group Member: V. S. DHARINEESH (RA2111003010008)
PREETHI. V (RA2111003010042)

Section: A1

Branch: CSE Core

Problem Statement:

Implementing Directory Search using Greedy and Dynamic Technique

Real time example:

Suppose you are working on a large software project that contains thousands of source code files organized in a directory structure. You need to frequently search for a specific file to make changes or additions to the code. However, manually searching for the file every time you need it is time-consuming and inefficient.

GREEDY PROGRAMING METHOD

Abstract:

The above program implements a directory search algorithm using a Greedy Technique in C. It takes as input a directory path and a target string, and recursively searches the directory structure for files or directories that contain the target string. When a match is found, the path to the matching file/directory is printed. This algorithm uses a Greedy Technique to search for the target string in each file or

directory it encounters, and continues searching until a match is found or the entire directory structure has been traversed. This program demonstrates a useful application of recursive functions and directory traversal in C.

Suppose you have a set of tasks to complete, each with a duration and a deadline. You want to complete as many tasks as possible before their respective deadlines. The Greedy method for this problem would be:

1. Sort the tasks in ascending order of their deadlines.
2. Start with the task with the earliest deadline and complete it.
3. Move to the next task with the earliest deadline that can be completed before its deadline.
4. Repeat step 3 until all tasks have been completed.

By completing tasks in this order, you are prioritizing those with earlier deadlines, which maximizes the number of tasks you can complete before their deadlines. However, note that this approach does not take into account the duration of the tasks or the possibility of completing more tasks by delaying some with later deadlines. Therefore, it may not always provide the optimal solution, but it is often a reasonable and efficient approach.

Algorithm:

1. Start by taking the directory path and target string as input from the user.
2. Open the directory using the opendir() function.
3. Check if the directory was opened successfully, and if not, print an error message and return.
4. Traverse the directory structure using readdir() function.

5. For each entry in the directory:
    1. If it is a subdirectory, recursively call the search_directory() function with the subdirectory path and the target string.
    2. If it is a file and contains the target string, print its path and store it as a match.
6. Close the directory using the closedir() function.
7. If a match was found, print the path to the matching file/directory.
8. Exit the function.

Pseudocode:

```
function search directory(directory,
  target): open     directory    for
        each  file    or
    directory in directory:
    if file/directory contains
      target: print
      file/directory path
      mark as match break
      if directory is not marked as visited:
        recursively call search directory on
    subdirectory close directory

  if match was found: print "Match found at: " +
    file/directory path
```

Time Complexity Analysis:

The time complexity of the directory search algorithm using a Greedy Technique in the worst case scenario is O(nm), where n is the total number of files and directories in the directory structure, and m is the length of the target string.

This is because the algorithm needs to traverse every file and directory in the structure, and for each one, it needs to search through its name to see if it contains the target string. The worst-case time complexity of searching through a string of

length m is O(m), so the time complexity of searching all files and directories is O(nm).

However, in practice, the algorithm may be able to find a match early and exit the loop, resulting in a lower time complexity. Additionally, the algorithm may be optimized by pruning branches of the directory tree that do not need to be searched, such as directories that have already been searched or do not match the target string. These optimizations can improve the practical running time of the algorithm.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>

void search_directory(char *dir_path, char *target) { DIR *dir
   = opendir(dir_path); if (!dir) { fprintf(stderr,
   "Error opening directory '%s'\n", dir_path); return;
   }

   struct dirent
   *entry; char
   *match = NULL;

   while ((entry = readdir(dir)) != NULL) {
      if (entry->d_type == DT_DIR && strcmp(entry->d_name,
"."") != 0 && strcmp(entry->d_name, "..") != 0) { // Recursively
search the
         subdirectory char sub_dir[strlen(dir_path) + strlen(entry-
```

```
        >d_name) + 2]; sprintf(sub_dir, "%s/%s", dir_path, entry-
        >d_name); search_directory(sub_dir, target);
      } else if (entry->d_type == DT_REG && strstr(entry->d_name,
      target) != NULL)

{
        // If the entry is a file and contains the target string,
        print its path char file_path[strlen(dir_path) +
        strlen(entry->d_name) + 2]; sprintf(file_path, "%s/%s",
        dir_path, entry->d_name);
        printf("%s\n", file_path); match = file_path; break;
      } }

   closedir(dir)

   ;

   // If we found a match, print the path to the matching
   file/directory if (match != NULL) {
      printf("Found match at: %s\n", match);
   }
}


int main() {
   char
   dir_path[256];
   char target[256];

   printf("Enter directory path: ");
   scanf("%s", dir_path);

   printf("Enter target

   string: ");

   scanf("%s", target);
```

```
    search_directory(dir_path,

    target); return 0;

}
```

Sample Input:

Directory path:

/home/user/Documents Target

string: report

Sample Output:

/home/user/Documents/reports/report1.docx
/home/user/Documents/reports/report2.docx
Match found at: /home/user/Documents/reports/report1.docx

Explanation:

The above code is a directory search algorithm that uses a Greedy Technique to traverse through the directory structure and search for a target string within the names of the files and directories. The code takes two inputs from the user: the directory path to search and the target string to look for.

The algorithm starts by opening the directory specified by the user using the opendir() function, and then loops through each file and subdirectory using the readdir() function. For each file or subdirectory, the algorithm checks if its name contains the target string. If it does, the algorithm prints the path to that file or subdirectory, marks it as a match, and exits the loop. If the current entry is a subdirectory, the algorithm recursively calls itself on that subdirectory to search through its files and subdirectories. If a match

was found, the algorithm prints the path to the matching file or subdirectory.

The time complexity of the algorithm is O(nm), where n is the total number of files and directories in the directory structure, and m is the length of the target string. In the worst case scenario, the algorithm needs to search through every file and directory to find a match, resulting in a time complexity of O(nm).
Overall, the directory search algorithm using a Greedy Technique is a useful tool for quickly searching through large directory structures to find specific files or directories.

DYNAMIC PROGRAMMING METHOD

Abstract:

This program uses Dynamic Programming techniques to search through a given directory structure to find a target string within the names of the files and directories. Dynamic Programming is a powerful optimization technique that stores the results of subproblems to avoid redundant calculations, leading to faster execution times.

The algorithm takes two inputs from the user: the directory path to search and the target string to look for. It uses a bottom-up approach to build a table of subproblems that store the results of searching through each file and directory in the directory structure. By using previously computed subproblems, the algorithm avoids searching through the same files and directories multiple times, resulting in faster execution times.

To solve this problem, you can implement a directory search function using dynamic programming technique. Here's how it works:

1. When the program starts up, it recursively scans the directory structure and builds a data structure that maps each file name to its path.

2. When you need to search for a specific file, you call the search function, passing in the root directory, the target file name, and a dictionary to store the memoized results.

3. The search function recursively searches each subdirectory and memoizes the results, so that if the same search is performed again, the result can be returned quickly from the memoized dictionary without having to search the entire directory structure again.

4. The search function returns the path to the target file if it is found, or None if it is not found.

With this implementation, you can search for a file in the directory structure quickly and efficiently, without having to manually navigate through the directories every time. This can save a lot of time and improve your productivity when working on large software projects

Algorithm:

1. Define a struct to store a match.
2. Define a table to store the results of subproblems.
3. Define a function to search for matches in a directory using dynamic programming.
    - Open the directory and loop through each file and subdirectory in the directory.
    - For each file, loop through each character in its name and use dynamic programming to check for a match with the target string.
    - If a match is found, store the path of the file in a Match struct and add it to an array of matches.
    - Recursively search any subdirectories.

- Close the directory and return the number of matches found.

4. Define a function to print the matches found.
5. In the main function, get the directory path and target string from the user.
6. Call the search_directory function with the directory path, target string, and an empty array of matches.
7. Call the print_matches function with the array of matches and the number of matches found.

Time Complexity Analysis:

The time complexity of the directory search algorithm using dynamic programming depends on the number of files and directories in the search path, as well as the length of the target string.

Let n be the number of files and directories in the search path, and let m be the length of the target string.
The time complexity of searching for a match in a single file name using dynamic programming is O(m*n). This is because we loop through each character in the file name, and for each character we loop through each character in the target string, resulting in a total of m*n operations.

Therefore, the overall time complexity of the algorithm is O(n*m*n), or O(n^2*m). This is because we search for a match in each file name, and for each file name we perform O(m*n) operations. Since we perform this operation for each of the n files in the directory, the total number of operations is O(n^2*m).

In the worst case, where every file name in the directory contains the target string, the algorithm will have to perform O(n^3) operations. However, in practice, the number of files in a directory is usually much smaller than the length of the target string, so the time complexity is typically closer to O(n*m) than O(n^2*m).

Program:

```
#include
<stdio.h>
#include
<stdlib.h>
#include
<string.h>
#include
<dirent.h>

#define MAX_FILENAME_LEN 256
#define MAX_DIR_LEN 256
#define MAX_MATCHES 100

// Define a struct to store a
match typedef struct {
   char path[MAX_DIR_LEN + MAX_FILENAME_LEN];
} Match;

// Define a table to store results of subproblems
int
dp_table[MAX_FILENAME_LEN][MAX_FILE
NAME_LEN]; //
Define a function to search for matches in a
directory int search_directory(char* dir_path,
char* target, Match* matches, int* match_count)
{ DIR* dir; struct dirent*
entry; int found_match = 0;

   // Open the directory and check for
   errors dir = opendir(dir_path); if
   (dir == NULL) {
      perror("Unable to open
      directory"); return -1;
   }

   // Loop through each file and subdirectory in the
   directory while ((entry = readdir(dir)) != NULL) {
```

```
if (entry->d_type == DT_DIR) {

   // If the current entry is a subdirectory, recursively search it if
(strcmp(entry->d_name, ".") != 0 && strcmp(entry->d_name,
   "..") != 0) { char subdir_path[MAX_DIR_LEN +
   MAX_FILENAME_LEN]; sprintf(subdir_path, "%s/%s",
   dir_path, entry->d_name);
      search_directory(subdir_path, target, matches, match_count);
   }
} else {
   // If the current entry is a file, search for a match in
   its name char* filename = entry->d_name; int
   filename_len = strlen(filename); int i, j;

   // Loop through each character in the file name and check for a
   match for (i = 0; i < filename_len; i++) {
      for (j = 0; j < strlen(target); j++) {
      if (dp_table[i][j] == 1 && filename[i+j] != target[j]) {
         dp_table[i][j] = 0;
         }
      if (filename[i+j] ==
         target[j]) {
         dp_table[i+1][j+1] = 1;
         }
         if (dp_table[i+1][j+1] == 1 && j == strlen(target)-1) {
            // If a match is found, store it in the
            matches array Match match;
            sprintf(match.path, "%s/%s", dir_path,
            filename); matches[*match_count] =
            match;
            (*match_count
            )++;
            found_match =
            1; break;
         }
      }
```

```
        if (found_match) {
          break;

          }
        }

        // Reset the DP table for the next file
        memset(dp_table, 0, sizeof(dp_table));
      }
    }

    // Close the directory and return the number of matches found
    closedir(dir);
    return *match_count;
}

// Define a function to print the matches found void
printf matches(Match* matches, int
match_count) { int i;
    printf("Matches
    found:\n"); for (i = 0; i <
    match count; i++) {
      printf("%s\n", matches[i].path);
    }
}

int main() {
    char
    dir_path[MAX_DIR_L
    EN]; char
    target[MAX_FILENAME_LEN];
    Match
    matches[MAX_MATCHES];
    int match count = 0;

    // Get the directory path and target string from the
    user printf("Directory path: ");
```

Sample Input:

Directory path:
./testdir Target
string: example

Sample Output:

Matches found:
./testdir/subdir1/example.txt
./testdir/subdir3/subsubdir1/example.txt
./testdir/subdir3/subsubdir2/example.txt

Explanation:

The directory search algorithm using dynamic programming is a modified version of the naive approach that uses the Greedy technique to search for a target string in a directory of files and subdirectories. In the naive approach, we compare the target string to each file name in the directory, and if the target string is a substring of a file name, we add that file to a list of matches.

However, this approach has a time complexity of O(n*m*k) where n is the number of files and directories in the search path, m is the length of the target string, and k is the maximum length of any file name in the directory. This is because for each file name in the directory, we need to compare it to the target string character by character, resulting in a total of m*k operations. Since we need to do this for each of the n files in the directory, the total number of operations is n*m*k.

The dynamic programming approach reduces the time complexity of this problem by using memoization to store the results of previous computations. Specifically, we use a 2D array dp to store whether or

not a given substring of the target string is a prefix of any file name in the directory. We then loop through each file name in the directory and compare it to the target string by checking whether any of its

substrings match a prefix of the target string. If a match is found, we add that file to a list of matches.

Why were these Techniques Chosen?

The Greedy and Dynamic Programming techniques were chosen for the directory search algorithm due to their efficiency and ability to solve similar types of problems.

The Greedy technique was chosen because it is a simple and intuitive approach to string matching. The idea is to search through each file name in the directory, comparing it character by character to the target string. If we find a match, we add that file to a list of matches. While the Greedy approach has a worst-case time complexity of O(n*m*k), where n is the number of files in the directory, m is the length of the target string, and k is the maximum length of any file name in the directory, it is often more efficient than other string matching algorithms in practice, especially when the length of the target string is small.

The Dynamic Programming technique was chosen as an improvement over the Greedy approach. By using memoization to store the results of previous computations, we can reduce the number of operations required to compare each file name to the target string. The Dynamic Programming approach has a time complexity of O(n*m^2), which is much better than the worst-case time complexity of the Greedy approach.

Additionally, the Dynamic Programming approach is more versatile and can be used to solve a wider range of string matching problems, making it a useful tool in algorithm design.

In summary, the Greedy and Dynamic Programming techniques were chosen for the directory search algorithm due to their simplicity,

efficiency, and versatility in solving similar types of problems.

CONCLUSION:

In conclusion, the directory search algorithm is a useful tool for searching for specific files within a directory and its subdirectories. The algorithm can be implemented using different techniques, such as Greedy and Dynamic Programming. The Greedy approach is simple and intuitive but can be slow for large directories or long target strings. The Dynamic Programming approach is an improvement over the Greedy approach, using memorization to reduce the number of operations required to compare each file name to the target string.

Overall, the choice of algorithm depends on the specific problem being solved and the constraints of the system. The directory search algorithm using Dynamic Programming is a good choice for larger directories or longer target strings, where the time complexity of the algorithm is a concern. However, for smaller directories or shorter target strings, the Greedy approach may be sufficient and more efficient in practice.