Implementation of Depth-Limited Search and Pattern Matching Algorithms for the 'Search' Command Feature in IF2230-2024 Operating System

Shulha - 13522087
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: shulhafahmi@gmail.com

Abstract*—This electronic document is a "live" template and already defines the components of your paper [title, text, heads, etc.] in its style sheet. **CRITICAL: Do Not Use Symbols, Special Characters, or Math in Paper Title or Abstract. (*Abstract*)

Keywords*—component; formatting; style; styling; insert (key words)

I. Introduction

The development of operating systems involves a deep understanding of various subsystems and their interactions. As part of the IF2230 Operating System class in 2024 in Informatics Engineering ITB, students are tasked with creating the IF2230-2024 Operating System, a comprehensive project that serves as an introduction to kernel development and the practical demonstration of operating system subsystems. The target platform for this operating system is the 32-bit Protected Mode on the x86 architecture, executed using QEMU. The course covers a wide range of topics including Toolchain, Kernel, Global Descriptor Table (GDT), Interrupts, Drivers, File System, Paging, User Mode, Shell, Processes, Scheduler, and Multitasking.

One of the critical components of this project is the implementation of a shell in user mode, which allows users to interact with the operating system via a command line interface. The shell supports various commands such as \`cd\` (change directory), \`ls\` (list files and directories), \`mkdir\` (create a new directory), \`cat\` (display the contents of a text file), \`find\` (search for files or directories in the file system), \`help\` (list available commands), \`clear\` (clear the screen), among others. These commands facilitate basic file system navigation and management, enhancing user interaction with the operating system.

In this paper, the author proposes the implementation of a new command, \`search\`. The \`search\` command is designed to accept a string as a parameter and search through the file system for text files that match the input string. The implementation of this command will leverage graph traversal algorithms, specifically Breadth-First Search (BFS) and Depth-First Search (DFS), as well as pattern matching algorithms to efficiently locate and identify matching files.

By integrating BFS and DFS algorithms, the command will explore directories and files in a structured manner, ensuring comprehensive coverage of the file system. Pattern matching algorithms will be employed to accurately identify text files that contain the specified string, enhancing the command's utility and precision.

This paper will detail the design, implementation, and testing of the "search" command, highlighting the integration of theoretical concepts with practical application.these components, incorporating the applicable criteria that follow.

II. Fundamental Theorem

A. Graph Traversal

A graph is a set of objects called vertices, connected by edges. Graphs can represent various problems. Graph traversal refers to the process of searching for solutions to problems represented by graphs (assuming the graph is connected). In the search for solutions, there are two approaches to graph representation: static graphs and dynamic graphs. A static graph is a graph that is fully formed before the search process begins (represented as a data structure). A dynamic graph is a graph that forms during the search process (the graph is not available before the search, it is built during the search for the solution).

Algorithms for graph traversal involve visiting the vertices in the graph systematically. Common algorithms for uninformed/blind search graph traversal include Breadth-First Search (BFS) and Depth-First Search (DFS).

For dynamic graphs, the search for solutions is conducted by building a dynamic tree. Each vertex is examined to determine if the solution (goal) has been reached. If a vertex represents a solution, the search can be terminated (for a single solution) or continued to find other solutions (for all solutions). The dynamic tree is represented by a state space tree, where vertices denote problem states (viable to form solutions), the root vertex represents the initial state, and leaf vertices represent solution/goal states. The solution is the path to the goal state.

B. Depth-First Search (DFS) Algorithm

The Depth-First Search (DFS) algorithm is a graph traversal algorithm that starts from a specific node and explores as many unvisited nodes as possible before backtracking. The process is as follows:

1. Visit node v.
2. Visit node w, which is adjacent to node v.
3. Repeat DFS starting from node w.
4. When a node u is reached such that all its adjacent nodes have been visited, backtrack to the last visited node that has unvisited adjacent nodes.
5. The search ends when there are no more unvisited nodes that can be reached from the visited nodes.

The construction of the status space tree using DFS is initialized with the initial state as the root. Generation is performed on all child nodes from the current node first (until there are no more) before moving on to the neighboring nodes. While DFS is less efficient in terms of time complexity, it is more efficient in terms of space complexity. DFS can sometimes choose the wrong path, leading to a longer or even infinite path. Therefore, step selection is crucial.

Depth-First Search (DFS) possesses distinct properties. Firstly, completeness is assured as long as the branching factor (b) is finite, and there is adequate handling of redundant paths and repeated states. However, DFS lacks optimality, meaning it does not guarantee the shortest path to a solution. In terms of time complexity, DFS operates with $O(b^m)$, where b is the branching factor and m is the maximum depth of the state space tree. Conversely, the space complexity of DFS is O(bm), reflecting its efficiency in utilizing memory, given that b is the branching factor and m is the maximum depth of the state space tree. In conclusion, while DFS is less efficient in terms of time complexity, it is advantageous in terms of space complexity.

The following example illustrates the sequence of nodes visited in DFS order from A → A, B, D, E, H, F, G, C and is depicted as a tree.

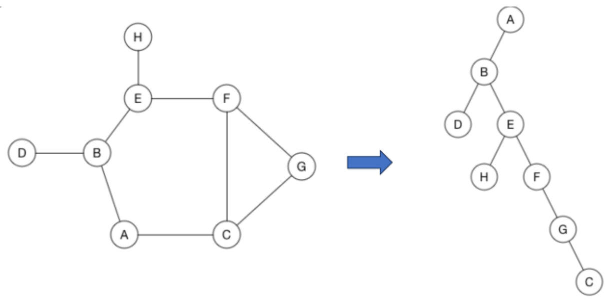


Fig. 1. Graph Traversal Using DFS (Source: [X])

C. Depth-Limited Search (DLS) Algorithm

Depth-Limited Search (DLS) is a variation of Depth-First Search (DFS) that incorporates a depth constraint to manage the depth of the search, thus addressing the primary issue of DFS potentially leading to very long or infinite paths due to incorrect step selections. Unlike Breadth-First Search (BFS), which ensures the discovery of the path with the fewest steps but demands significant memory to maintain the search status, DFS is more memory-efficient but does not guarantee finding the shortest path to the solution. DLS mitigates this by treating nodes at a specified depth limit, l, as having no successors, effectively preventing the search from extending indefinitely. However, the primary challenge with DLS lies in determining the appropriate depth limit, which should be at least equal to the depth of the shallowest goal.

Despite this improvement, DLS is not complete because it might fail to find a solution if the solution lies deeper than the set limit. It is also not optimal as it does not necessarily find the shortest path to the goal. The time complexity of DLS is $O(b^l)$, where b is the branching factor and l is the depth limit, and its space complexity is also $O(b^l)$, since it only needs to store the nodes along the current path and their depths. In essence, DLS combines the memory efficiency of DFS with a depth constraint to avoid infinite paths, but this comes at the cost of sacrificing completeness and optimality, making the determination of the correct depth limit critical for its effectiveness.

D. Pattern Matching Algorithm

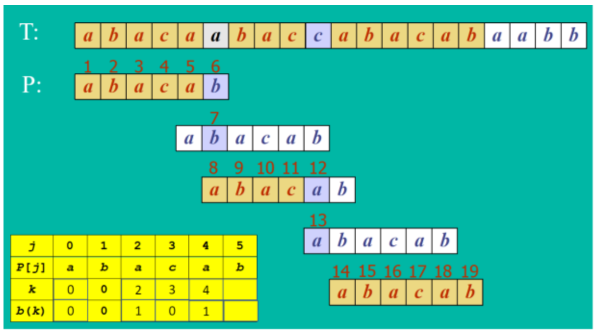
Pattern matching is the process of finding the first location in a text that matches a certain pattern. In this context, we are given T, or text, which is a long string with a length of n characters, and P, or pattern, which is a string with a length of m characters (assuming m is much smaller than n) that will be searched for within the text. The goal of pattern matching is to find the first location in the text that corresponds to the given pattern.

E. Knuth-Morris-Pratt (KMP) Algorithm

The Knuth-Morris-Pratt (KMP) algorithm searches for a pattern in a text from left to right, similar to the brute force algorithm. However, the difference is that the KMP algorithm shifts the pattern more intelligently than the brute force algorithm, thus avoiding many unnecessary comparisons.

The creator of this algorithm is Donald E. Knuth, a computer scientist and Professor Emeritus at Stanford University. Knuth is renowned for his monumental work, "The Art of Computer Programming," and is considered the father of algorithm analysis. His work has been highly influential in the development of rigorous analysis of computational complexity of algorithms and in the formulation of formal mathematical techniques for algorithms.

In the KMP algorithm, when there is a mismatch between the text and the pattern at position P at index j (with T[i] not equal to P[j]), the algorithm shifts the pattern in the most efficient way to avoid redundant comparisons. The solution is to shift the pattern as far as possible while ensuring that the shift covers the largest prefix of P[0..j-1] that is also a suffix of P[1..j-1]. This allows the algorithm to bypass comparisons that are guaranteed to fail, thereby increasing the efficiency of pattern searching in the text.



j	0	1	2	3	4	5
P[j]	a	b	a	c	a	b
k	0	0	2	3	4	
b(k)	0	0	1	0	1	

Fig. 2. Illustration of string matching with KMP (Source: [X])

The time complexity of KMP for computing the prefix function is O(m) and for string searching is O(n), resulting in a total time complexity of O(m+n). The advantage of KMP is that it does not require backtracking in the input. However, its disadvantage is that its performance degrades as the alphabet size increases.

F. Boyer-Moore (BM) Algorithm

The Boyer-Moore algorithm is a pattern-matching algorithm based on two main techniques. The first technique is called the "looking-glass" technique, where the search for pattern P in text T is conducted by moving backward through P, starting from its end. The second technique is the "character-jump" technique. When a mismatch occurs at position T[i] and the character in pattern P[j] does not match T[i], three possibilities are tried in sequence.

In the first case, if pattern P contains character x, then P is shifted to the right to align the last occurrence of x in P with T[i]. In the second case, if pattern P contains character x but shifting to the last occurrence of x is not possible, then P is shifted right by one character to align with T[i+1]. In the third case, if neither of the previous cases applies, P is shifted to align P[0] with T[i+1].

The Last Occurrence function is used in the Boyer-Moore algorithm to map pattern P and alphabet A into a function L(x) that stores the index of the last occurrence of each character x in A. L(x) is defined as the largest index i where P[i] = x, or -1 if no such index exists.

In terms of analysis, the worst-case time complexity of the Boyer-Moore algorithm is O(nm + A), where the algorithm is fast if the alphabet (A) is large and slow if the alphabet is small. For instance, this algorithm performs well for English text but is less optimal for binary text. Boyer-Moore is significantly faster compared to the brute force method in searching English text.

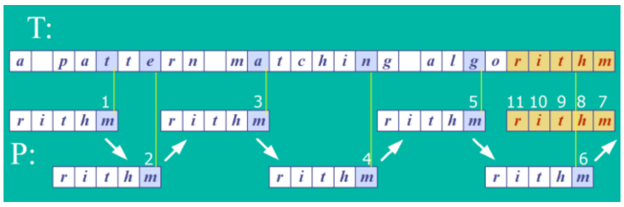


Fig. 3. Illustration of string matching with BM (Source: [X])

G. IF2240 (2024) Operating System Project

The project use the 32-bit Protected Mode on the x86architecture and also use FAT32 filesystem.

The 32-bit Protected Mode is a mode of operation in the x86 architecture that allows the operating system to manage memory and resources more efficiently. In this mode, the processor divides the 32-bit address space into four segments: code, data, stack, and extra. Each segment has its own base address and limit, which are stored in the segment registers (CS, DS, SS, and ES). This segmentation provides a higher level of memory protection and isolation between different parts of the operating system and applications.

In 32-bit Protected Mode, the operating system uses the Global Descriptor Table (GDT) to manage the segments. The GDT is a data structure that contains the base and limit of each segment, as well as other attributes such as the segment type and access rights. The operating system uses the GDT to set up the segment registers and to switch between different segments. This allows the operating system to dynamically allocate and deallocate memory, and to provide a higher level of security and isolation between different parts of the system.

One of the key benefits of 32-bit Protected Mode is that it allows the operating system to use a flat memory model, where all memory is addressed using a single 32-bit address space. This simplifies memory management and allows the operating system to use a more efficient memory allocation strategy. Additionally, the segmentation provided by 32-bit Protected Mode helps to prevent memory corruption and other security vulnerabilities by isolating different parts of the system from each other.

III. Implementation

A. Limitations

- The feature ‘search’ command is added to IF2230-2024 OS Project by UsusBuntu group which is not perfect in the implementation.
- Maximum depth set to ten, potentially limiting deeper data exploration.
- Search functionality restricted to .txt files, excluding other file formats.
- Search queries limited to 20 characters, affecting the comprehensiveness of searches.

B. Scope of Discussion

The discussion example in this paper will use the graph as shown on Fig. 4.

The “tes.txt” file contains the string “Informatika berjiwa satria, tidak pernah mengenal keluh kesah” and the “stima.txt” file contains the string “aaaa cape stima mati cape stima aaa”.

Author will discuss five different cases of strings that are matched or not matched any of the files.

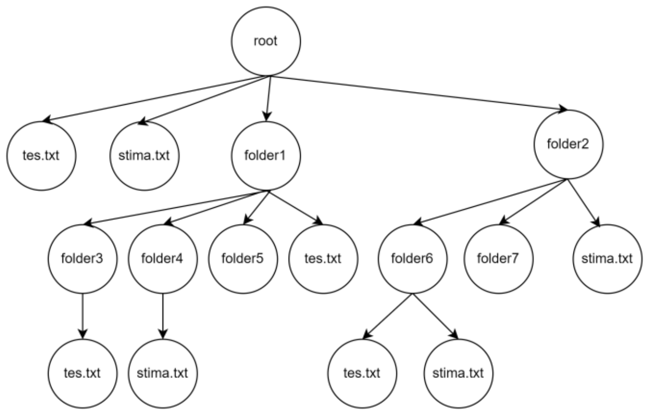


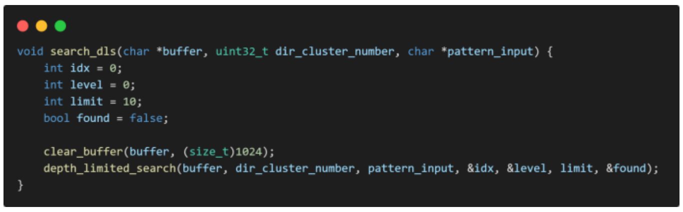
Fig. 4. Illustration of graph used in this paper (Source: Primary)

C. *Depth-Limited Search Implementation*

Implementing DLS to the illustration will not violate the level limit of 10 as in our limitations. The DLS traversal is implemented with the `search_dls` and `depth_limited_search` functions.

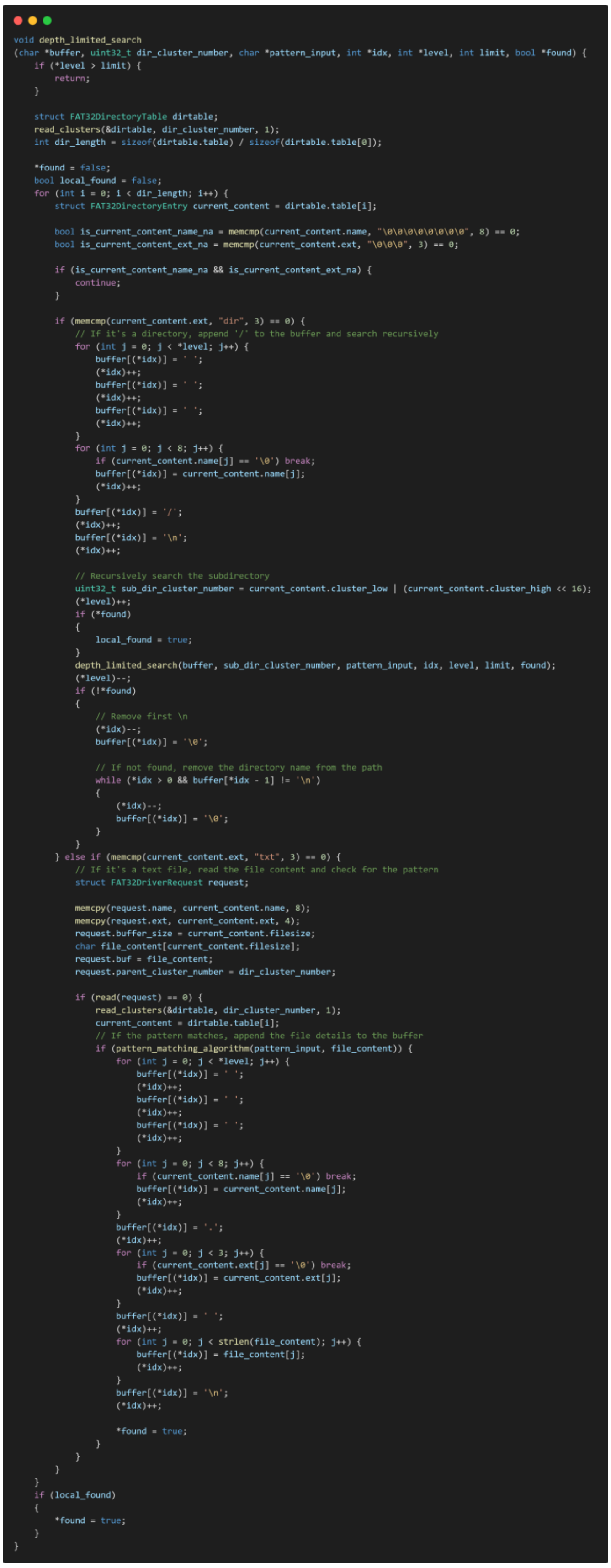
The primary function, `search_dls`, initializes a buffer, sets a directory cluster number and search pattern, and calls the `depth_limited_search` function to perform the actual search.

The `depth_limited_search` function operates recursively, navigating the directory structure up to a specified depth limit, we use ten as limitations. It reads directory entries and determines if each entry is a directory or a text file. If it's a directory, the function appends the directory name to the buffer and recursively searches within it. If a text file is found, its content is read, and the pattern matching algorithm is used to search for the pattern within the file. If the pattern is found/matched, the file details and content are appended to the buffer. The function ensures the buffer only retains relevant paths by removing directory names if the search within them is unsuccessful. Overall, the code effectively traverses the file system to find and list text files containing the desired pattern, handling both directory and file cases with appropriate recursion and buffer management.



```c
void search_dls(char *buffer, uint32_t dir_cluster_number, char *pattern_input) {
    int idx = 0;
    int level = 0;
    int limit = 10;
    bool found = false;

    clear_buffer(buffer, (size_t)1024);
    depth_limited_search(buffer, dir_cluster_number, pattern_input, &idx, &level, limit, &found);
}
```

Fig. 5. Code implementation of DLS Traversal (Source: Personal)



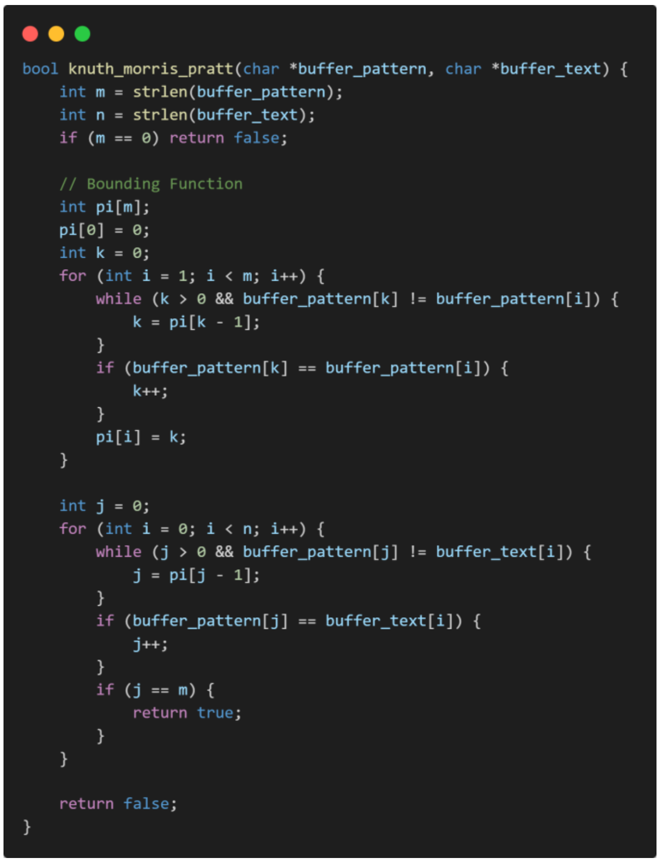
```c
void depth_limited_search
(char *buffer, uint32_t dir_cluster_number, char *pattern_input, int *idx, int *level, int limit, bool *found) {
    if (*level > limit) {
        return;
    }

    struct FAT32DirectoryTable dirtable;
    read_clusters(&dirtable, dir_cluster_number, 1);
    int dir_length = sizeof(dirtable.table) / sizeof(dirtable.table[0]);

    *found = false;
    bool local_found = false;
    for (int i = 0; i < dir_length; i++) {
        struct FAT32DirectoryEntry current_content = dirtable.table[i];

        bool is_current_content_name_na = memcmp(current_content.name, "\0\0\0\0\0\0\0\0", 8) == 0;
        bool is_current_content_ext_na = memcmp(current_content.ext, "\0\0\0", 3) == 0;

        if (is_current_content_name_na && is_current_content_ext_na) {
            continue;
        }

        if (memcmp(current_content.ext, "dir", 3) == 0) {
            // If it's a directory, append '/' to the buffer and search recursively
            for (int j = 0; j < *level; j++) {
                buffer[(*idx)] = ' ';
                (*idx)++;
                buffer[(*idx)] = ' ';
                (*idx)++;
                buffer[(*idx)] = ' ';
                (*idx)++;
            }
            for (int j = 0; j < 8; j++) {
                if (current_content.name[j] == '\0') break;
                buffer[(*idx)] = current_content.name[j];
                (*idx)++;
            }
            buffer[(*idx)] = '/';
            (*idx)++;
            buffer[(*idx)] = '\n';
            (*idx)++;

            // Recursively search the subdirectory
            uint32_t sub_dir_cluster_number = current_content.cluster_low | (current_content.cluster_high << 16);
            (*level)++;
            if (*found)
            {
                local_found = true;
            }
            depth_limited_search(buffer, sub_dir_cluster_number, pattern_input, idx, level, limit, found);
            (*level)--;
            if (!*found)
            {
                // Remove first \n
                (*idx)--;
                buffer[(*idx)] = '\0';

                // If not found, remove the directory name from the path
                while (*idx > 0 && buffer[*idx - 1] != '\n')
                {
                    (*idx)--;
                    buffer[(*idx)] = '\0';
                }
            }
        } else if (memcmp(current_content.ext, "txt", 3) == 0) {
            // If it's a text file, read the file content and check for the pattern
            struct FAT32DriverRequest request;

            memcpy(request.name, current_content.name, 8);
            memcpy(request.ext, current_content.ext, 4);
            request.buffer_size = current_content.filesize;
            char file_content[current_content.filesize];
            request.buf = file_content;
            request.parent_cluster_number = dir_cluster_number;

            if (read(request) == 0) {
                read_clusters(&dirtable, dir_cluster_number, 1);
                current_content = dirtable.table[i];
                // If the pattern matches, append the file details to the buffer
                if (pattern_matching_algorithm(pattern_input, file_content)) {
                    for (int j = 0; j < *level; j++) {
                        buffer[(*idx)] = ' ';
                        (*idx)++;
                        buffer[(*idx)] = ' ';
                        (*idx)++;
                        buffer[(*idx)] = ' ';
                        (*idx)++;
                    }
                    for (int j = 0; j < 8; j++) {
                        if (current_content.name[j] == '\0') break;
                        buffer[(*idx)] = current_content.name[j];
                        (*idx)++;
                    }
                    buffer[(*idx)] = '.';
                    (*idx)++;
                    for (int j = 0; j < 3; j++) {
                        if (current_content.ext[j] == '\0') break;
                        buffer[(*idx)] = current_content.ext[j];
                        (*idx)++;
                    }
                    buffer[(*idx)] = ' ';
                    (*idx)++;
                    for (int j = 0; j < strlen(file_content); j++) {
                        buffer[(*idx)] = file_content[j];
                        (*idx)++;
                    }
                    buffer[(*idx)] = '\n';
                    (*idx)++;

                    *found = true;
                }
            }
        }
    }
    if (local_found)
    {
        *found = true;
    }
}
```

Fig. 6. Code implementation of DLS Traversal (Source: Personal)

D. *Pattern (String) Matching Implementation*

The implementation of Boyer-Moore and Knuth-Morris-Pratt Algorithm is in the `boyer_moore` and `knuth_morris_pratt` functions.

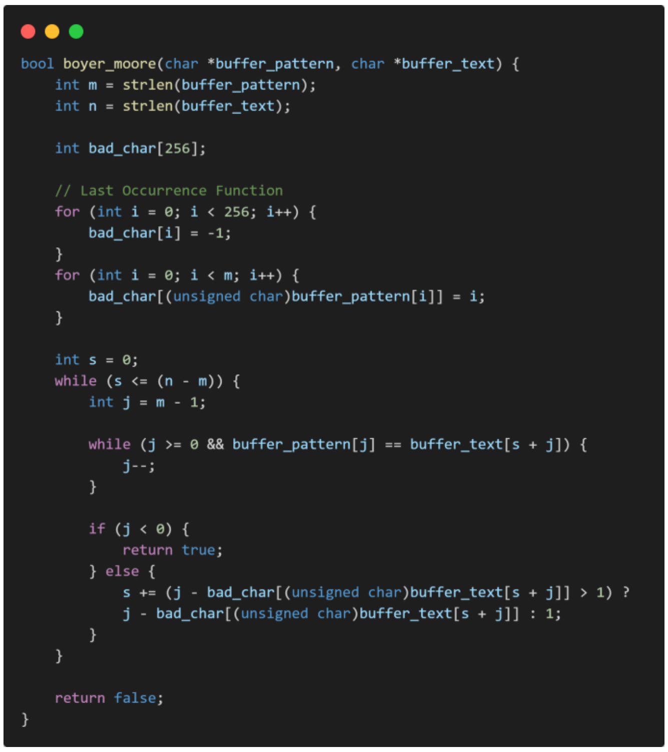
The `knuth_morris_pratt` function starts by checking the lengths of the pattern and text, and if the pattern length is zero, it returns false, indicating no match. It then constructs the bounding function array `pi`, which is used to store the length of the longest prefix which is also a suffix. This array helps in avoiding redundant comparisons. The algorithm iterates through the text and pattern, comparing characters. If a mismatch occurs, it uses the bounding function array to shift the pattern efficiently without rechecking previously matched characters. If a complete match of the pattern is found in the text, it returns true. If no match is found by the end of the text, it returns false.



```c
bool knuth_morris_pratt(char *buffer_pattern, char *buffer_text) {
    int m = strlen(buffer_pattern);
    int n = strlen(buffer_text);
    if (m == 0) return false;

    // Bounding Function
    int pi[m];
    pi[0] = 0;
    int k = 0;
    for (int i = 1; i < m; i++) {
        while (k > 0 && buffer_pattern[k] != buffer_pattern[i]) {
            k = pi[k - 1];
        }
        if (buffer_pattern[k] == buffer_pattern[i]) {
            k++;
        }
        pi[i] = k;
    }

    int j = 0;
    for (int i = 0; i < n; i++) {
        while (j > 0 && buffer_pattern[j] != buffer_text[i]) {
            j = pi[j - 1];
        }
        if (buffer_pattern[j] == buffer_text[i]) {
            j++;
        }
        if (j == m) {
            return true;
        }
    }

    return false;
}
```

Fig. 7. Code implementation of KMP (Source: Personal)

The `boyer_moore` function starts by initializing an array bad_char to store the last occurrence of each character in the pattern. This array helps in determining the shift distance when a mismatch occurs. The function then iterates through the text from left to right, comparing the pattern characters from right to left. If a mismatch is found, the algorithm uses the bad_char array to determine the optimal shift distance, ensuring that the pattern is aligned with the next possible match position in the text. If a complete match is found, the function returns true. If the pattern is not found by the end of the text, it returns false.

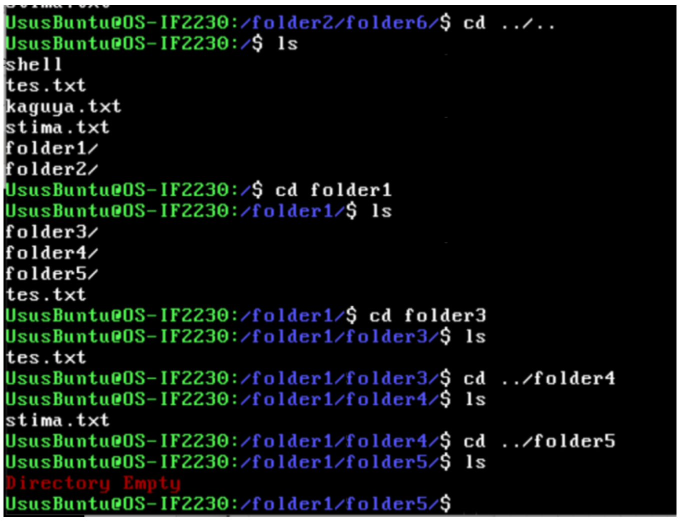


```c
bool boyer_moore(char *buffer_pattern, char *buffer_text) {
    int m = strlen(buffer_pattern);
    int n = strlen(buffer_text);

    int bad_char[256];

    // Last Occurrence Function
    for (int i = 0; i < 256; i++) {
        bad_char[i] = -1;
    }
    for (int i = 0; i < m; i++) {
        bad_char[(unsigned char)buffer_pattern[i]] = i;
    }

    int s = 0;
    while (s <= (n - m)) {
        int j = m - 1;

        while (j >= 0 && buffer_pattern[j] == buffer_text[s + j]) {
            j--;
        }

        if (j < 0) {
            return true;
        } else {
            s += (j - bad_char[(unsigned char)buffer_text[s + j]] > 1) ?
            j - bad_char[(unsigned char)buffer_text[s + j]] : 1;
        }
    }

    return false;
}
```

Fig. 8. Code implementation of Boyer-Moore (Source: Personal)

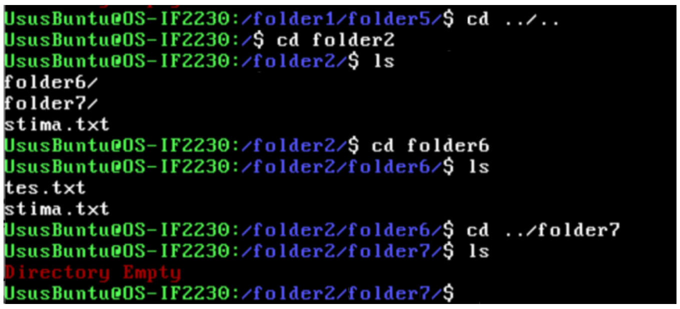
E. *Experiments of DLS and Pattern Matching Algorithm*

The folder structure of the graph illustration as shown in Fig 4. is shown in Fig. 9 and Fig. 10 below.



```
UsusBuntu@OS-IF2230:/folder2/folder6/$ cd ../..
UsusBuntu@OS-IF2230:/$ ls
shell
tes.txt
kaguya.txt
stima.txt
folder1/
folder2/
UsusBuntu@OS-IF2230:/$ cd folder1
UsusBuntu@OS-IF2230:/folder1/$ ls
folder3/
folder4/
folder5/
tes.txt
UsusBuntu@OS-IF2230:/folder1/$ cd folder3
UsusBuntu@OS-IF2230:/folder1/folder3/$ ls
tes.txt
UsusBuntu@OS-IF2230:/folder1/folder3/$ cd ../folder4
UsusBuntu@OS-IF2230:/folder1/folder4/$ ls
stima.txt
UsusBuntu@OS-IF2230:/folder1/folder4/$ cd ../folder5
UsusBuntu@OS-IF2230:/folder1/folder5/$ ls
Directory Empty
UsusBuntu@OS-IF2230:/folder1/folder5/$
```

Fig. 9. Illustration Folder Structure (Source: Personal)



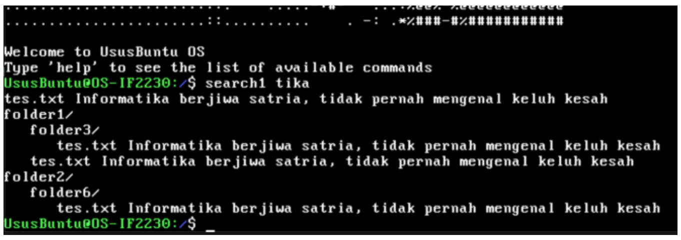
```
UsusBuntu@OS-IF2230:/folder1/folder5/$ cd ../..
UsusBuntu@OS-IF2230:/$ cd folder2
UsusBuntu@OS-IF2230:/folder2/$ ls
folder6/
folder7/
stima.txt
UsusBuntu@OS-IF2230:/folder2/$ cd folder6
UsusBuntu@OS-IF2230:/folder2/folder6/$ ls
tes.txt
stima.txt
UsusBuntu@OS-IF2230:/folder2/folder6/$ cd ../folder7
UsusBuntu@OS-IF2230:/folder2/folder7/$ ls
Directory Empty
UsusBuntu@OS-IF2230:/folder2/folder7/$
```

Fig. 10. Illustration Folder Structure (Source: Personal)

Generally, using DLS to the graph illustration will traverse through root – tes.txt – stima.txt – folder1 – folder1/folder3 – folder1/folder3/tes.txt – folder1/folder4/stima.txt – folder1/folder5 – folder1/tes.txt – folder2 – folder2/folder6 – folder2/folder6/tes.txt – folder2/folder6/stima.txt – folder2/folder7 – folder2/stima.txt.

1) Command "search tika"

This command should print all the tes.txt files directory including their contents. The tes.txt files are in root, folder1, folder3, and folder6.

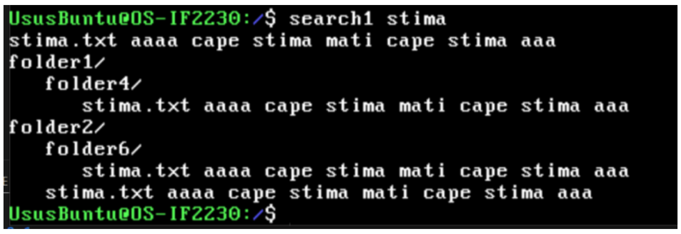


```
.........................::..........    . -: .*%###-#%###########

Welcome to UsusBuntu OS
Type 'help' to see the list of available commands
UsusBuntu@OS-IF2230:/$ search1 tika
tes.txt Informatika berjiwa satria, tidak pernah mengenal keluh kesah
folder1/
   folder3/
      tes.txt Informatika berjiwa satria, tidak pernah mengenal keluh kesah
   tes.txt Informatika berjiwa satria, tidak pernah mengenal keluh kesah
folder2/
   folder6/
      tes.txt Informatika berjiwa satria, tidak pernah mengenal keluh kesah
UsusBuntu@OS-IF2230:/$ _
```

Fig. 11. Command "search tika" (Source: Personal)

2) Command "search stima"

This command should print all the stima.txt files directory including their contents. The tes.txt files are in root, folder4, folder6, and folder2.

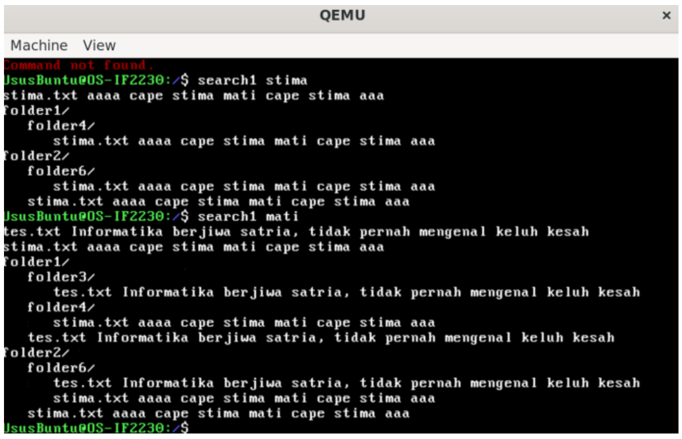


```
UsusBuntu@OS-IF2230:/$ search1 stima
stima.txt aaaa cape stima mati cape stima aaa
folder1/
   folder4/
      stima.txt aaaa cape stima mati cape stima aaa
folder2/
   folder6/
      stima.txt aaaa cape stima mati cape stima aaa
   stima.txt aaaa cape stima mati cape stima aaa
UsusBuntu@OS-IF2230:/$
```

Fig. 12. Command "search stima" (Source: Personal)

3) Command "search mati"

This command should print all the stima.txt and tes.txt files directory including their contents.

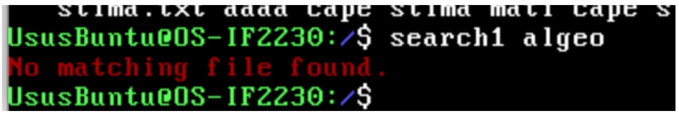


```
Command not found.
UsusBuntu@OS-IF2230:/$ search1 stima
stima.txt aaaa cape stima mati cape stima aaa
folder1/
   folder4/
      stima.txt aaaa cape stima mati cape stima aaa
folder2/
   folder6/
      stima.txt aaaa cape stima mati cape stima aaa
   stima.txt aaaa cape stima mati cape stima aaa
UsusBuntu@OS-IF2230:/$ search1 mati
tes.txt Informatika berjiwa satria, tidak pernah mengenal keluh kesah
stima.txt aaaa cape stima mati cape stima aaa
folder1/
   folder3/
      tes.txt Informatika berjiwa satria, tidak pernah mengenal keluh kesah
   folder4/
      stima.txt aaaa cape stima mati cape stima aaa
   tes.txt Informatika berjiwa satria, tidak pernah mengenal keluh kesah
folder2/
   folder6/
      tes.txt Informatika berjiwa satria, tidak pernah mengenal keluh kesah
      stima.txt aaaa cape stima mati cape stima aaa
   stima.txt aaaa cape stima mati cape stima aaa
UsusBuntu@OS-IF2230:/$
```

Fig. 13. Command "search mati" (Source: Personal)

4) Command "search algeo"

This command should print that no files match.

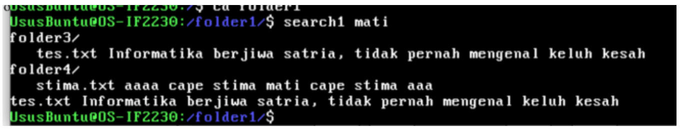


```
UsusBuntu@OS-IF2230:/$ search1 algeo
No matching file found.
UsusBuntu@OS-IF2230:/$
```

Fig. 14. Command "search algeo" (Source: Personal)

5) Command "search mati" within folder1 directory

This command should print all the stima.txt and tes.txt files directory including their contents in folder1 directory, that is tes.txt in folder1 itself and also folder3 and stima.txt in folder4.



```
UsusBuntu@OS-IF2230:/$ cd folder1
UsusBuntu@OS-IF2230:/folder1/$ search1 mati
folder3/
   tes.txt Informatika berjiwa satria, tidak pernah mengenal keluh kesah
folder4/
   stima.txt aaaa cape stima mati cape stima aaa
tes.txt Informatika berjiwa satria, tidak pernah mengenal keluh kesah
UsusBuntu@OS-IF2230:/folder1/$
```

Fig. 15. Command "search mati" in folder1 directory (Source: Personal)

IV. Conclusion

The search command implementation for the IF2230-2024 Operating System successfully integrates depth-limited search (DLS) and pattern matching algorithms to enhance file system navigation. By leveraging DLS, the command efficiently explores directories up to a specified depth, while the Knuth-Morris-Pratt (KMP) and Boyer-Moore (BM) algorithms provide accurate pattern matching within text files.

This approach addresses the need for efficient and precise file searches, as outlined in the project introduction. Despite some limitations, such as the depth constraint and restricted pattern length, the search command significantly improves the operating system's usability, demonstrating a practical application of theoretical concepts in real-world scenarios.

Video Link at YouTube *(Heading 5)*

YouTube Video Link https://youtu.be/IHwueKVv-os. Github repository on YouTube caption.

Acknowledgment

The author expresses gratitude to Allah SWT for His blessings, enabling the completion of this paper. The author would also like to especially thank the lecturer of the IF2211 Algorithm Strategies course,Dr. Ir. Rinaldi Munir, M.T. for his inspiring and dedicated teaching and also for providing plenty sources on the courses as this paper would not have been possible without them. The author acknowledges the presence of imperfections in this work and welcomes suggestions and critiques for future improvement.

References

[1] C. Hernand, "Programming Embedded Systems in C and C++," University of Concepción, 2005. [Online]. Available: http://www.inf.udec.cl/~chernand/sc/links/embedded_pmode.pdf. Accessed on: June 12, 2024. J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68-73.

[2] Distributed System Laboratory, "Book I: Protected Mode x86, IF2230 Operating System, 2nd Edition" Informatics Engineering ITB, 2024. [Online]. Unpublished.

[3] M. H. Nour, "Computer Organization and Assembly Language Lecture Notes," University of Science and Technology, 2018. [Online]. Available: https://csit.ust.edu.sd/files/2018/10/lec2-COAsm2018.pdf. Accessed on: June 12, 2024.

[4] R. Munir, "BFS dan DFS Bagian 1," *IF2211 Strategi Algoritma,* 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/BFS-DFS-2021-Bag1-2024.pdf. Accessed on: June 12, 2024.

[5] R. Munir, "BFS dan DFS Bagian 2," *IF2211 Strategi Algoritma,* 2021. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf. Accessed on: June 12, 2024.

[6] R. Munir, "Pencocokan String," *IF2211 Strategi Algoritma,* 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/ Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf. Accessed on: June 12, 2024

STATEMENT

I hereby declare that the paper I have written is my own work, not a reproduction or translation of someone else's paper, and is not plagiarized.

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, June 12, 2024

[illegible]

13522087 Shulha