Analisis Perbandingan Algoritma *BFS* dan *DFS* untuk Traversal Pencarian *File* Pada *Filesystem Linux*

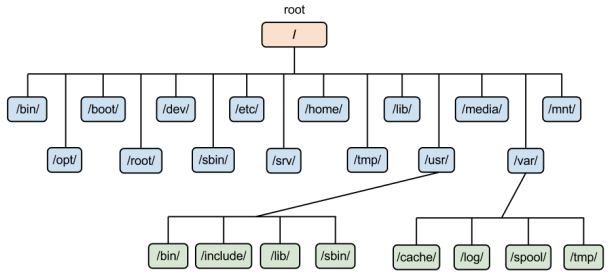
Renuno Yuqa Frinardi - 13524080
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: renunofrinardi@gmail.com , 13524080@std.stei.itb.ac.id

***Abstract*—Makalah ini membahas perbandingan algoritma Breadth-First Search (BFS) dan Depth-First Search (DFS) untuk pencarian file pada filesystem Linux. Pengujian dilakukan menggunakan program C pada beberapa kasus struktur direktori, yaitu wide-shallow, deep-narrow, mixed, dan no-target. Metrik yang diamati meliputi waktu eksekusi, jumlah entri yang diperiksa, ukuran maksimum space, dan kedalaman traversal. Hasil pengujian menunjukkan bahwa BFS lebih sesuai untuk struktur dangkal, sedangkan DFS lebih hemat memori dan dapat lebih unggul pada struktur dalam atau cabang tertentu.**

Keywords—Breadth-First Search; Depth-First Search; filesystem Linux; pencarian file; traversal direktori

I. Pendahuluan

Sistem operasi modern menyediakan sebuah fitur penyimpanan data yang memungkinkan pengguna mengelola data secara masif dan terstruktur, contohnya pada sistem operasi berbasis *Linux*. Pada sistem operasi berbasis *Linux*, data tidak hanya tersimpan sebagai kumpulan data biasa, tetapi juga tersusun dalam struktur hierarkis. Struktur ini membentuk hubungan *parent-child* antara direktori dan subdirektori, sehingga sistem bisa dipandang sebagai sebuah pohon atau graf. Dalam penggunaan sehari-hari, pengguna sering kali perlu mencari data atau *file* tertentu yang tersebar di berbagai direktori dengan kedalaman tertentu, misalnya mencari dokumen PDF, mencari file konfigurasi, atau menelusuri kumpulan *source code* dalam suatu *repository*.



Sumber: https://mahasiswa.ung.ac.id/532423015/home/2023/11/13/penjelasan-tentang-struktur-directory-linux.html

***Gambar 1.** Struktur filesystem Linux*

Pencarian *file* pada *filesystem* dapat dilakukan dengan berbagai algoritma. Dua algoritma dasar yang sering digunakan ini adalah *Breadth-First Search* (BFS) dan *Depth-First Search* (DFS). BFS menelusuri struktur data secara melebar dan DFS menelusuri struktur data secara mendalam.

Makalah ini ditujukan sebagai analisis perbandingan algoritma BFS dan DFS untuk traversal pencarian file pada *filesystem* Linux. Makalah akan berfokus pada performa kedua algoritma tersebut ketika digunakan untuk menelusuri struktur direktori *filesystem Linux*, serta bagaimana perbedaan strategi traversal memengaruhi performa pencarian. Perbandingan dilakukan melalui rancangan eksperimen sederhana menggunakan program traversal direktori. Program tersebut digunakan untuk mencari *file* target pada beberapa skenario struktur direktori, seperti struktur dangkal dan lebar, struktur dalam dan sempit, serta struktur campuran yang menyerupai direktori yang biasa digunakan sehari-hari.

II. Dasar Teori

A. Graf dan Traversal pada Graf

Graf merupakan salah satu struktur yang banyak digunakan untuk merepresentasikan hubungan antar objek. Suatu graf terdiri atas himpunan simpul dan himpunan sisi. Umumnya simpul menyatakan objek yang akan dimodelkan, sedangkan sisi menyatakan hubungan antara dua objek. Dalam banyak persoalan komputasi, graf dapat digunakan untuk merepresentasikan jaringan komputer, hubungan antar halaman web, peta jalan, hubungan sosial, maupun sebuah hirarki direktori pada *filesystem*.

Traversal pada graf adalah proses mengunjungi simpul-simpul di dalam graf dengan urutan tertentu secara terarah [1]. Tujuan traversal tidak selalu sama pada setiap persoalan yang ada. Pada satu kasus, traversal digunakan hanya untuk mencetak seluruh simpul. Pada kasus lain, traversal digunakan untuk menemukan simpul tertentu, menghitung jumlah simpul yang memenuhi kriteria tertentu, mencari lintasan dari simpul awal ke simpul tujuan, atau membangun struktur seperti pohon traversal.

Graf yang digunakan dalam sebuah proses pencarian dapat bersifat *statis* atau *dinamis*. Graf *statis* adalah graf yang seluruh simpul dan sisinya sudah tersedia sebelum proses pencarian dilakukan. Sebaliknya, graf *dinamis* adalah graf yang dibangun secara bertahap selama proses pencarian berlangsung [1]. Pencarian *file* pada *filesystem Linux* lebih dekat dengan konsep graf *dinamis*, karena program pencarian tidak selalu mengetahui struktur utuh direktori sejak awal. Program baru mengetahui bentuk dari suatu direktori setelah direktori tersebut dibuka dan dibaca isinya.

Pada sebuah struktur *filesystem* yang tidak mengandung *symbolic link* dengan bentuk rekursif, hubungan antar direktori dapat digambarkan sebagai pohon berakar. Direktori awal pencarian menjadi akar, setiap subdirektori menjadi simpul internal, dan file dapat dipandang sebagai simpul daun atau objek yang diperiksa pada saat suatu direktori dikunjungi. Namun, apabila symbolic link menuju direktori lain ikut ditelusuri, struktur tersebut tidak lagi selalu berbentuk pohon. *Symbolic link* dapat membuat suatu direktori menunjuk ke direktori yang sudah pernah dikunjungi, sehingga dapat terbentuk siklus.

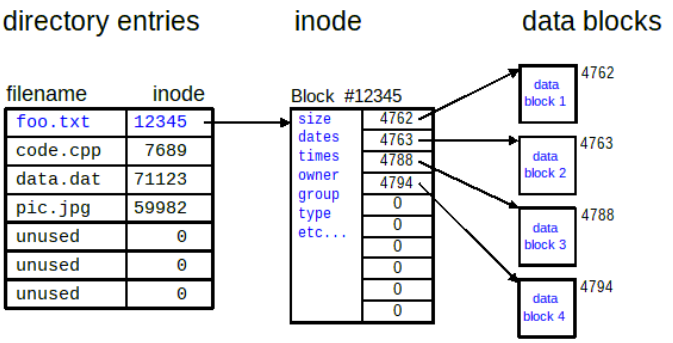
B. *Filesystem*, *VFS*, *Inode*, dan *Directory Entry*

Filesystem adalah mekanisme yang digunakan sistem operasi untuk mengatur penyimpanan dan pengambilan data pada media penyimpanan tetap, seperti *Hard Disk Drive* (HDD) atau *Solid State Drive* (SSD). *Filesystem* tidak hanya menyimpan isi file, tetapi juga menyimpan *metadata* seperti ukuran *file*, keterangan akses, pemilik *file*, waktu modifikasi, dan informasi lokasi fisik data pada penyimpanan. Pada *Linux*, terdapat berbagai jenis *filesystem* yang bisa digunakan seperti ext4, XFS, Btrfs, dan tmpfs. Agar program tidak perlu mengetahui detail internal setiap filesystem, Linux menyediakan abstraksi bernama *Virtual File System* (VFS).

VFS berperan sebagai *interface* umum antara aplikasi dan implementasi *filesystem*. Pada dokumentasi kernel Linux dijelaskan bahwa VFS digunakan oleh *system call* seperti open, stat, dan chmod. Dalam proses pencarian pathname, VFS memanfaatkan *directory entry cache* atau *dentry cache* untuk mempercepat translasi nama path menjadi objek file yang sesuai [3]. Sehingga, proses traversal direktori tidak hanya dipengaruhi oleh algoritma BFS atau DFS, tetapi juga oleh mekanisme *caching* yang dilakukan oleh kernel.

Pada *filesystem* ext4, direktori pada dasarnya dapat dipandang sebagai *file* khusus yang memetakan nama entri ke nomor *inode* [4]. Sehingga, sebuah direktori menyimpan daftar nama *file* atau subdirektori disertai *inode* yang bersesuaian. *Inode* sendiri merupakan struktur data yang menyimpan *metadata* tentang *file*. Pada *Linux*, informasi *inode* dapat diperoleh oleh aplikasi melalui *system call* seperti stat atau statx [5]. Informasi seperti nomor *inode*, ukuran *file*, mode *file*, pemilik, grup, serta *timestamp* merupakan contoh *metadata* yang dapat digunakan program sebagai deskripsi dari suatu entri.

Konsep *inode* dan *directory entry* penting dalam traversal *filesystem*. Nama file yang terlihat oleh pengguna tidak secara langsung menyimpan seluruh isi *file*, melainkan mengarah ke *inode* tertentu melalui entri direktori. Hal ini juga menjelaskan mengapa *hard link* dapat membuat beberapa nama berbeda merujuk ke *inode* yang sama.

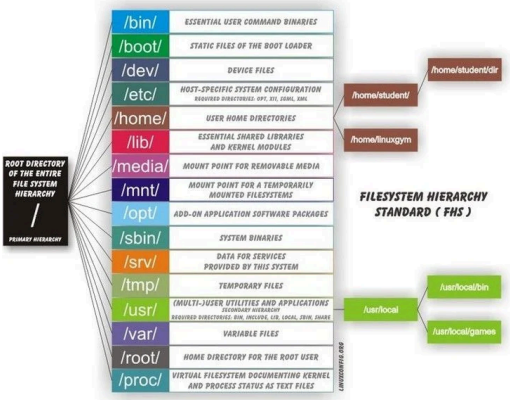


Sumber: https://azrael.digipen.edu/~mmead/www/Courses/CS180/FileSystems-1.html

Gambar 2. *Ilustrasi hubungan directory entry, inode, dan, data blocks pada filesystem ext-based*

C. Struktur Direktori pada Linux

Linux menggunakan struktur direktori hirarkis yang berawal pada direktori *root*, yaitu "/". Di bawah root terdapat berbagai direktori lainnya seperti /home, /etc, /usr, /var, /bin, dan banyak lagi. *Filesystem Hierarchy Standard* (FHS) mendefinisikan aturan penempatan file dan direktori pada sistem operasi UNIX-*like* agar memiliki kesamaan antara aplikasi, *tools* administrasi sistem, *script*, dan dokumentasi [6]. Aturan tersebut bertujuan agar pengguna dapat memiliki pemahaman umum mengenai lokasi *file* tertentu. Misalnya, konfigurasi sistem umumnya berada pada /etc, data pengguna berada pada /home, dan data yang sering dimodifikasi dapat berada pada /var.



Sumber: https://kkslinuxinfo.wordpress.com/2015/12/03/file-system-structure/

Gambar 3. *Diagram penjelasan Filesystem Hierarchy Standard (FHS)*

Meskipun secara graf struktur *filesystem Linux* tampak seperti satu pohon besar, beberapa bagian dari pohon tersebut dapat berasal dari *filesystem* yang berbeda juga. *Linux*

memungkinkan *filesystem* berjenis lain dipasang atau di-*mount* ke suatu direktori tertentu.

Dalam pencarian file, setiap direktori dapat dianggap sebagai ruang pencarian lokal. Ketika program mengunjungi suatu direktori, program akan membaca daftar entri di dalamnya. Entri tersebut dapat berupa *file* biasa, subdirektori, *symbolic link*, *device file*, *socket*, atau jenis *file* lain yang dikenal pada sistem UNIX-*like*.

D. *Pathname Resolution* dan *Symbolic Link*

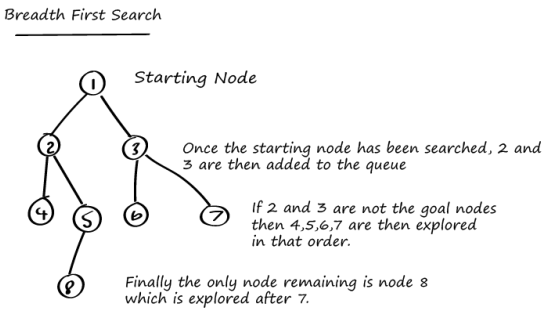
Pathname resolution adalah proses menerjemahkan *path* atau alamat *file/directory* dari bentuk teks menjadi objek file atau direktori yang sesuai. Path seperti /home/user/project/main.c terdiri atas beberapa komponen path, yaitu "home", "user", "project", dan "main.c". Sistem operasi harus memeriksa setiap komponen secara berurutan dari direktori awal tertentu. Apabila path dimulai dengan "/", pencarian dimulai dari direktori *root*. Apabila path bersifat relatif, pencarian dimulai dari *current working directory* proses.

Symbolic link adalah *file* khusus yang berisi referensi menuju *path* lain. Ketika suatu komponen *path* merupakan *symbolic link*, sistem dapat mentranslasikan atau *me-resolve symbolic link* tersebut untuk menemukan alamat sebenarnya [7]. Hal ini membuat *symbolic link* berguna untuk membuat alias atau jalan pintas ke file dan direktori lain. Namun, *symbolic link* juga dapat menimbulkan masalah dalam traversal rekursif karena *symbolic link* dapat menunjuk ke direktori induk atau direktori yang sudah pernah dikunjungi. Jika program mengikuti *symbolic link* tanpa mekanisme pendeteksian siklus, traversal dapat terjebak dalam *loop*.

Beberapa program dan *library* pencarian menyediakan *handling* khusus untuk menangani *symbolic link*. Pada *Python*, os.walk secara *default* tidak menelusuri *symbolic link* yang menunjuk ke direktori [8]. Pada GNU *find*, *symbolic link* juga dapat ditangani dengan beberapa mode, misalnya memeriksa *link* itu sendiri atau mengikuti target *link* [9].

E. *Breadth-First Search* (BFS)

Breadth-First Search (BFS) adalah algoritma traversal graf yang mengunjungi simpul secara melebar. Traversal dimulai dari simpul awal, kemudian semua simpul yang bertetangga langsung dengan simpul awal dikunjungi terlebih dahulu. Setelah seluruh simpul pada kedalaman tersebut selesai diproses, BFS melanjutkan pencarian ke simpul pada kedalaman berikutnya [1]. Dengan demikian, BFS memproses simpul berdasarkan level kedalaman.



Sumber: https://tutorialedge.net/artificial-intelligence/breadth-first-search-java/

Gambar 4. *Ilustrasi penjelasan algoritma BFS*

BFS umumnya diimplementasikan menggunakan struktur data *queue*. *Queue* bekerja dengan prinsip *First-In First-Out* (FIFO), yaitu elemen yang pertama dimasukkan akan menjadi elemen pertama yang dikeluarkan. Pada traversal BFS, simpul awal dimasukkan ke *queue*. Selama *queue* belum kosong, program mengambil simpul paling depan, memproses simpul tersebut, lalu memasukkan seluruh tetangga yang belum dikunjungi ke bagian belakang *queue*. Untuk menghindari kunjungan yang berulang, BFS menggunakan sebuah penanda seperti *array boolean visited* pada graf, atau himpunan *visited* [1].

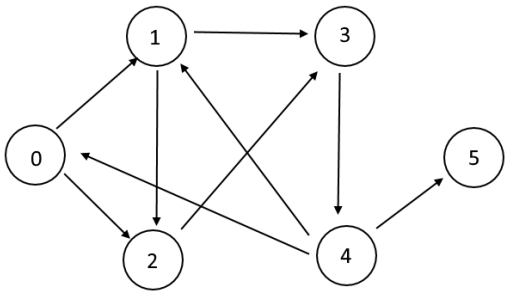
Dalam hal traversal direktori, simpul dapat direpresentasikan sebagai direktori. Ketika sebuah direktori diproses, program membaca seluruh entri di dalamnya. Jika entri merupakan *file*, nama *file* dibandingkan dengan target. Jika entri merupakan subdirektori, subdirektori tersebut dimasukkan ke *queue* untuk diproses setelah seluruh direktori lain pada level yang sama selesai. Dengan cara ini, BFS akan memeriksa file dan direktori yang lebih dekat dari direktori awal terlebih dahulu.

Kelebihan BFS pada pencarian *file* adalah kemampuannya menemukan target dengan jumlah level direktori minimum, apabila setiap langkah antar direktori dianggap memiliki biaya yang sama. Misalnya, jika file target berada langsung pada direktori awal atau pada subdirektori dangkal, BFS cenderung menemukannya lebih cepat dibanding DFS yang mungkin terlebih dahulu masuk ke cabang yang dalam. Namun, kelemahan BFS adalah kebutuhan memorinya. Pada struktur direktori yang sangat lebar, BFS harus menyimpan banyak direktori pada level yang sama di dalam *queue*. Oleh karena itu, BFS dapat memiliki penggunaan memori yang lebih besar dibanding DFS pada struktur direktori tertentu [2].

F. *Depth-First Search* (DFS)

Depth-First Search (DFS) adalah algoritma traversal graf yang mengunjungi simpul secara mendalam. Traversal dimulai dari simpul awal, kemudian algoritma memilih salah satu

simpul tetangga yang belum dikunjungi dan melanjutkan traversal dari simpul tersebut. Proses ini diulang sampai algoritma mencapai simpul yang tidak memiliki tetangga belum dikunjungi. Pada kondisi tersebut, algoritma melakukan *backtracking* ke simpul sebelumnya untuk mencari cabang lain yang belum dikunjungi [1].



Depth First Traversal - 0 1 3 4 5 2

Sumber: *https://medium.com/@that-software-PM/depth-first-search-dfs-algorithm-201dc95e524*

Gambar 5. *Ilustrasi traversal algoritma DFS*

DFS dapat diimplementasikan secara rekursif atau iteratif menggunakan *stack*. Pada implementasi rekursif, *stack* pemanggilan fungsi secara implisit menyimpan jalur traversal yang sedang ditempuh. Pada implementasi iteratif, program menyimpan simpul yang akan dikunjungi di dalam stack eksplisit. *Stack* bekerja dengan prinsip *Last-In First-Out* (LIFO), yaitu elemen terakhir yang dimasukkan akan menjadi elemen pertama yang dikeluarkan. Prinsip ini membuat DFS selalu melanjutkan pencarian ke cabang terakhir yang dipilih sampai cabang tersebut habis.

Dalam hal traversal direktori, DFS akan masuk ke salah satu subdirektori terlebih dahulu, lalu melanjutkan ke subdirektori di dalamnya, dan seterusnya sampai tidak ada subdirektori lagi atau *file* target ditemukan. Jika cabang tersebut tidak mengandung target, DFS kembali ke direktori sebelumnya dan mencoba cabang lain. Strategi ini membuat DFS cocok untuk struktur direktori yang dalam apabila target memang berada pada cabang yang dipilih lebih awal. Namun, DFS dapat menjadi kurang efisien apabila target berada pada level dangkal tetapi algoritma menelusuri cabang yang sangat dalam terlebih dahulu.

DFS umumnya memiliki kebutuhan memori yang lebih kecil dibanding BFS karena DFS hanya perlu menyimpan jalur dari akar ke simpul yang sedang diproses beserta beberapa alternatif cabang yang belum dieksplorasi. Akan tetapi, DFS tidak menjamin menemukan solusi dengan kedalaman minimum. Pada pencarian *file*, hal ini berarti DFS tidak selalu menemukan *file* yang paling dekat dari direktori awal apabila terdapat beberapa *file* dengan nama yang sama pada kedalaman berbeda.

G. Kompleksitas Algoritma

Kompleksitas algoritma adalah ukuran yang digunakan untuk menilai efisiensi suatu algoritma berdasarkan kebutuhan sumber daya yang diperlukan ketika besar masukan bertambah. Secara umum, kompleksitas algoritma dapat dilihat dari dua aspek, yaitu kompleksitas waktu dan kompleksitas ruang. Kompleksitas waktu, dinotasikan sebagai $T(n)$, menyatakan banyaknya tahapan komputasi atau operasi dasar yang dilakukan algoritma sebagai fungsi dari besar masukan n. Sementara itu, kompleksitas ruang, dinotasikan sebagai $S(n)$, menyatakan banyaknya memori yang digunakan oleh struktur data di dalam algoritma sebagai fungsi dari ukuran masukan n [10].

Dalam analisis algoritma, pengukuran waktu eksekusi secara eksak atau langsung dalam detik tidak selalu menjadi ukuran yang ideal karena hasilnya dapat dipengaruhi oleh arsitektur komputer, *compiler*, sistem operasi, dan kondisi *hardware*. Oleh karena itu, analisis kompleksitas lebih sering dilakukan secara abstrak dengan menghitung operasi khas yang mendasari algoritma, misalnya operasi perbandingan pada algoritma pencarian, operasi pertukaran pada pengurutan, atau operasi penambahan dan perkalian pada komputasi numerik [10].

Untuk membandingkan algoritma pada ukuran masukan yang besar, digunakan kompleksitas waktu asimptotik. Salah satu notasi yang paling umum digunakan adalah notasi O-Besar atau *Big-O Notation*. Notasi $T(n) = O(f(n))$ menyatakan bahwa pertumbuhan $T(n)$ dibatasi dari atas oleh fungsi $f(n)$ untuk nilai n yang cukup besar. Dengan kata lain, *Big-O* menggambarkan laju pertumbuhan kebutuhan waktu algoritma ketika ukuran masukan semakin besar, tanpa terlalu memperhatikan koefisien dan suku dengan orde lebih rendah [11]. Sebagai contoh, jika $T(n) = 2n^2 + 6n + 1$, maka untuk n yang besar suku dominannya adalah n^2, sehingga kompleksitasnya dapat ditulis sebagai $O(n^2)$.

III. Implementasi dan Pengujian

Implementasi pengujian pada makalah ini bertujuan untuk membandingkan performa algoritma *Breadth-First Search* (BFS) dan *Depth-First Search* (DFS) dalam melakukan traversal pencarian file pada *filesystem Linux*. Pengujian dilakukan dengan membangun program *command-line* dalam bahasa C yang dapat berjalan pada sistem operasi *Linux*, khususnya distro *ArchLinux*. Program ini menerima masukan berupa algoritma yang digunakan, direktori awal pencarian, dan nama *file* target. Setelah proses pencarian selesai, program menampilkan hasil pencarian pada terminal sekaligus menyimpan hasil tersebut ke dalam berkas CSV.

A. Lingkungan dan Struktur Implementasi

Program dikembangkan menggunakan bahasa C dengan *library standar* C dan POSIX *library* yang tersedia pada *Linux*. Pemilihan bahasa C dilakukan karena bahasa ini merupakan bahasa *low-level* yang dekat dengan sistem operasi, khususnya dalam traversal dan pembacaan direktori, pemeriksaan metadata file, dan pengukuran waktu eksekusi. Proses pembacaan direktori dilakukan menggunakan *interface* seperti *opendir* dan *readdir*, sedangkan pemeriksaan jenis file

dilakukan menggunakan *lstat* agar *symbolic link* tidak otomatis diikuti.

Secara umum, struktur repository yang digunakan adalah sebagai berikut.

```
bfs-dfs-filesystem-experiment/
├── Makefile
├── README.md
├── include/
├── src/
├── scripts/
├── testing/
├── results/
└── bin/
```

Pemisahan struktur tersebut bertujuan agar eksperimen mudah dijalankan berulang kali. Pengguna bisa langsung menjalankan *makefile* untuk melakukan kompilasi, *make* setup untuk membentuk struktur direktori pengujian, dan *make* experiments untuk menjalankan seluruh skenario pengujian.

B. Representasi Masalah Pencarian *File*

Dalam implementasi ini, pencarian *file* pada *filesystem Linux* direpresentasikan sebagai proses traversal pada pohon direktori. Direktori awal pencarian dilihat sebagai simpul akar. Setiap subdirektori di dalamnya dilihat sebagai simpul anak, sedangkan *file* biasa dipandang sebagai entri yang diperiksa namanya. Jika nama *file* sama dengan target yang dicari, maka pencarian dianggap berhasil.

Pada kasus terburuk, jika *file* target tidak ditemukan atau berada pada posisi terakhir dalam urutan traversal, BFS dan DFS dapat memeriksa seluruh entri dalam ruang pencarian. Oleh karena itu, kompleksitas waktu terburuk keduanya pada eksperimen filesystem dapat dinyatakan sebagai:

$$T_{BFS}(n) = O(n)$$
$$T_{DFS}(n) = O(n)$$

Meskipun kompleksitas waktu terburuk keduanya sama, urutan kunjungan yang berbeda dapat menghasilkan performa riil yang berbeda. BFS memeriksa direktori berdasarkan kedalaman dari direktori awal, sedangkan DFS menelusuri satu cabang sedalam mungkin sebelum berpindah ke cabang lain. Akibatnya, posisi file target dan bentuk struktur direktori sangat mempengaruhi jumlah entri yang dikunjungi sebelum target ditemukan.

C. Implementasi BFS pada Traversal Direktori

Algoritma BFS diimplementasikan menggunakan struktur data *queue*. *Queue* menyimpan daftar *path* direktori yang belum diproses. Pada awal pencarian, direktori root dimasukkan ke dalam *queue*. Selama *queue* tidak kosong, program mengambil direktori paling depan, membaca seluruh entri di dalam direktori tersebut, lalu memeriksa setiap entri. Jika entri berupa file biasa dan namanya sama dengan nama target, maka pencarian selesai. Jika entri berupa direktori, maka path direktori tersebut dimasukkan ke bagian belakang *queue*.

Langkah kerja BFS pada program adalah sebagai berikut.

1. Masukkan direktori awal ke dalam *queue*.
2. Selama *queue* tidak kosong, ambil direktori terdepan.
3. Baca seluruh entri direktori dan urutkan berdasarkan alfabet.
4. Untuk setiap entri, periksa apakah entri merupakan *file* target.
5. Jika entri merupakan direktori, masukkan ke *queue*.
6. Hentikan pencarian jika target ditemukan atau seluruh *queue* habis.

D. Implementasi DFS pada Traversal Direktori

Algoritma DFS diimplementasikan menggunakan struktur data *stack eksplisit*. Pemilihan *stack eksplisit* dilakukan untuk menghindari risiko *stack overflow* apabila struktur direktori sangat dalam. *Stack* menyimpan *path* direktori yang belum diproses. Pada awal pencarian, direktori root dimasukkan ke dalam *stack*. Selama stack tidak kosong, program mengambil direktori paling atas, membaca seluruh entri di dalamnya, lalu memeriksa *file* dan subdirektori yang ditemukan.

Langkah kerja DFS pada program adalah sebagai berikut.

1. Masukkan direktori awal ke dalam *stack*.
2. Selama *stack* tidak kosong, ambil direktori paling atas.
3. Baca seluruh entri direktori dan urutkan secara alfabetis.
4. Untuk setiap entri, periksa apakah entri merupakan *file* target.
5. Jika entri merupakan direktori, masukkan ke *stack*.
6. Hentikan pencarian jika target ditemukan atau seluruh *stack* habis.

E. Penanganan *Symbolic Link* dan Urutan Entri

Pada *filesystem Linux*, *symbolic link* dapat menunjuk ke file atau direktori lain. Apabila *symbolic link* menuju direktori diikuti secara rekursif, traversal dapat membentuk siklus. Sebagai contoh, sebuah *symbolic link* dapat menunjuk ke direktori induknya sendiri sehingga program dapat mengunjungi direktori yang sama berkali-kali. Untuk menghindari permasalahan tersebut, implementasi program tidak mengikuti *symbolic link* ke direktori.

Selain *symbolic link*, urutan pembacaan entri direktori juga perlu diperhatikan. Urutan entri yang dikembalikan oleh sistem tidak selalu dijamin terurut secara alfabetik. Jika program langsung mengikuti urutan tersebut, hasil BFS dan DFS dapat berbeda antara filesystem atau kondisi eksekusi yang berbeda. Oleh karena itu, setiap kali program membaca isi direktori, daftar entri akan diurutkan terlebih dahulu berdasarkan nama. Dengan demikian, urutan traversal menjadi lebih konsisten dan eksperimen lebih mudah diulang dengan lebih konsisten.

IV. Hasil Pengujian

Terdapat empat skenario pengujian, yaitu *wide_shallow*, *deep_narrow*, *mixed_project*, dan *no_target*. Setiap skenario dijalankan sebanyak 30 kali untuk masing-masing algoritma, sehingga total terdapat 240 data pengujian. Atribut atau

parameter yang dicatat meliputi waktu eksekusi, jumlah direktori yang dikunjungi, jumlah file dan entri yang diperiksa, ukuran maksimum *space* yang digunakan, serta kedalaman maksimum traversal.

Secara umum, seluruh pengujian berhasil dijalankan tanpa error. Nilai *skipped_dirs* dan *error_count* bernilai 0 pada seluruh skenario, sehingga tidak terdapat direktori yang gagal dibaca. Skenario *wide_shallow*, *deep_narrow*, dan *mixed_project* berhasil menemukan target, sedangkan skenario *no_target* tidak menemukan target sesuai rancangan dataset. Karena terdapat beberapa *outlier* pada waktu eksekusi, analisis utama menggunakan nilai median, bukan rata-rata.

Skenario	Algoritma	Direktori Dikunjungi	File Diperiksa	Entri Diperiksa	Maks. Space	Maks. Kedalaman
wide_shallow	BFS	51	151	201	50	1
wide_shallow	DFS	51	148	198	50	1
deep_narrow	BFS	31	31	61	1	30
deep_narrow	DFS	31	30	60	1	30
mixed_project	BFS	11	20	30	6	3
mixed_project	DFS	9	14	24	6	3
no_target	BFS	40	80	119	27	3
no_target	DFS	40	80	119	7	3

Tabel 1. *Hasil perbandingan BFS dan DFS pada kelima test case*

Pada skenario *wide_shallow*, BFS dan DFS memiliki performa yang hampir sama karena struktur direktori hanya memiliki kedalaman maksimum 1. BFS sedikit lebih cepat, tetapi selisihnya tidak signifikan. Pada skenario *deep_narrow*, DFS lebih cepat karena struktur direktori berbentuk rantai dalam dengan *branching factor* kecil. Akibatnya, BFS dan DFS menelusuri jalur yang hampir sama, tetapi DFS memeriksa lebih sedikit entri.

Pada skenario *mixed_project*, DFS menunjukkan hasil paling baik. DFS hanya mengunjungi 9 direktori dan memeriksa 24 entri, sedangkan BFS mengunjungi 11 direktori dan memeriksa 30 entri. Hal ini menunjukkan bahwa target berada pada cabang yang lebih cepat dicapai oleh urutan traversal DFS. Karena DFS masuk lebih dahulu ke salah satu cabang secara mendalam, target pada struktur proyek dapat ditemukan sebelum seluruh level direktori lain diperiksa.

Pada skenario *no_target*, BFS dan DFS sama-sama mengunjungi 40 direktori, memeriksa 80 file, dan memeriksa 119 entri. Hal ini sesuai dengan teori karena ketika target tidak ada, kedua algoritma harus menelusuri seluruh ruang pencarian. Perbedaan utama terlihat pada ukuran maksimum *space*. BFS memiliki maksimum *space* sebesar 27, sedangkan DFS hanya 7. Hasil ini mendukung teori bahwa BFS cenderung membutuhkan ruang lebih besar karena menyimpan banyak simpul pada level yang sama, sedangkan DFS lebih hemat memori karena menelusuri cabang secara mendalam [1], [2].

Secara keseluruhan, hasil pengujian sesuai dengan teori traversal graf. BFS lebih sesuai digunakan apabila target diperkirakan berada pada kedalaman dangkal dan pencarian berdasarkan level lebih diutamakan. Sebaliknya, DFS lebih sesuai untuk struktur direktori yang dalam atau ketika penggunaan memori perlu dihemat. Pada kasus terburuk, BFS dan DFS sama-sama dapat memiliki kompleksitas waktu *O(N)*, dengan N adalah jumlah entri yang diperiksa. Namun, perbedaan urutan traversal menyebabkan performa riil keduanya dapat berbeda tergantung bentuk struktur direktori dan posisi file target.

Dengan demikian, tidak dapat disimpulkan bahwa salah satu algoritma selalu lebih baik. BFS unggul pada kondisi tertentu, terutama ketika target dekat dengan direktori awal. DFS unggul pada kondisi lain, terutama ketika target berada pada cabang yang lebih cepat dicapai oleh traversal mendalam. Oleh karena itu, pemilihan algoritma pencarian file pada *filesystem Linux* perlu mempertimbangkan karakteristik struktur direktori, posisi target yang diharapkan, serta kebutuhan memori.

V. KESIMPULAN

Berdasarkan implementasi dan pengujian yang telah dilakukan, algoritma BFS dan DFS sama-sama dapat digunakan untuk traversal pencarian *file* pada *filesystem* Linux. Keduanya mampu menemukan *file* target pada skenario yang memiliki target dan memberikan hasil tidak ditemukan pada skenario tanpa target. Pada kasus terburuk, BFS dan DFS sama-sama dapat memiliki kompleksitas waktu *O(n)*, dengan *n* menyatakan jumlah entri *file* dan direktori yang diperiksa.

Hasil pengujian menunjukkan bahwa tidak ada algoritma yang selalu unggul pada seluruh skenario. BFS sedikit lebih cepat pada struktur *wide_shallow* dan *no_target*, sedangkan DFS lebih cepat pada struktur *deep_narrow* dan *mixed_project*. Perbedaan ini terjadi karena BFS menelusuri direktori berdasarkan level, sedangkan DFS menelusuri satu cabang secara mendalam terlebih dahulu. Oleh karena itu, posisi *file* target dan bentuk struktur direktori sangat memengaruhi performa praktis kedua algoritma.\

Sedangkan dari sisi penggunaan memori, DFS cenderung lebih hemat dibandingkan BFS, terutama pada struktur direktori yang memiliki banyak percabangan. Hal ini terlihat dari ukuran maksimum *space* pada skenario *no_target*, di mana BFS menyimpan lebih banyak direktori yang menunggu diproses dibandingkan DFS. Dengan demikian, BFS lebih sesuai digunakan apabila target diperkirakan berada pada kedalaman dangkal, sedangkan DFS lebih sesuai untuk struktur direktori yang dalam atau ketika penggunaan memori perlu ditekan. Pemilihan algoritma pencarian file sebaiknya disesuaikan dengan karakteristik struktur direktori dan kebutuhan pencarian.

LAMPIRAN LINK YOUTUBE

Video penjelasan dari makalah dapat dicek dan dilihatpada link *YouTube* berikut: https://youtu.be/QI2L_aQOU58

Lampiran Link GitHub

Kode program dan hasil pengujian dapat dicek dan digunakan pada link *GitHub* berikut: https://github.com/renuno-frinardi/IF2211-Makalah-StiMa-1

Ucapan Terima Kasih

Pertama-tama penulis ingin mengucapkan terima kasih kepada Tuhan Yang Maha Esa atas anugerah yang telah diberikan sehingga penulis bisa menyelesaikan makalah ini. Penulis juga ingin berterima kasih kepada dosen yang mengajar penulis pada mata kuliah IF2211 Strategi Algoritma, yaitu bapak Ir. Rila Mandala, M.Eng., Ph.D. Selanjutnya penulis juga ingin memberikan ucapan terima kasih kepada seluruh dosen dan asisten dari mata kuliah IF2130 Sistem Operasi, karena telah memberikan pembelajaran dan pembawaan materi secara menarik & menyenangkan mengenai *filesystem,* sehingga memberikan saya ide untuk menyusun makalah berkaitan dengan hal tersebut. Terakhir saya juga ingin mengucapkan terima kasih kepada keluarga dan teman-teman yang selalu memberikan dukungan selama perkuliahan yang dipadati dengan tugas-tugas di Semester 4 ini.

References

[1] R. Munir dan N. U. Maulidevi, "Breadth/Depth First Search (BFS/DFS) Bagian 1," Bahan Kuliah IF2211 Strategi Algoritmik, Program Studi Teknik Informatika, Institut Teknologi Bandung, 2026.

[2] R. Munir dan N. U. Maulidevi, "Breadth/Depth First Search (BFS/DFS) Bagian 2," Bahan Kuliah IF2211 Strategi Algoritma, Program Studi Teknik Informatika, Institut Teknologi Bandung, 2026.

[3] The Linux Kernel Documentation, "Overview of the Linux Virtual File System." [Online]. Available: https://docs.kernel.org/filesystems/vfs.html. [Diakses: 16 Juni 2026].

[4] The Linux Kernel Documentation, "ext4 Directory Entries." [Online]. Available: https://docs.kernel.org/filesystems/ext4/directory.html. [Diakses: 16 Juni 2026].

[5] M. Kerrisk, "inode(7) — Linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man7/inode.7.html. [Diakses: 16 Juni 2026].

[6] Linux Foundation, "Filesystem Hierarchy Standard 3.0," 2015. [Online]. Available: https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html. [Diakses: 16 Juni 2026].

[7] M. Kerrisk, "path_resolution(7) — Linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man7/path_resolution.7.html. [Diakses: 16 Juni 2026].

[8] Python Software Foundation, "os — Miscellaneous operating system interfaces." [Online]. Available: https://docs.python.org/3/library/os.html. [Diakses: 16 Juni 2026].

[9] GNU Project, "Symbolic Links, GNU Findutils Manual." [Online]. Available: https://www.gnu.org/software/findutils/manual/html_node/find_html/Symbolic-Links.html. [Diakses: 16 Juni 2026].

[10] R. Munir, "Kompleksitas Algoritma (Bagian 1)," Bahan Kuliah IF1220 Matematika Diskrit, Program Studi Teknik Informatika, STEI ITB, 2026.

[11] R. Munir, "Kompleksitas Algoritma (Bagian 2)," Bahan Kuliah IF1220 Matematika Diskrit, Program Studi Teknik Informatika, STEI ITB, 2026.

PERNYATAAN

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, 17 Juni 2026

Renuno Yuqa Frinardi
13524080