Pendekatan Algoritma dalam Word Search Puzzles Solver: Sebuah Perbandingan antara DFS, BFS, dan Brute-force

Zahira Dina Amalia - 13522085
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: 13522085@std.stei.itb.ac.id

Makalah ini mengevaluasi efisiensi metode Depth-First Search (DFS), Breadth-First Search (BFS), dan Brute-force dalam menyelesaikan Word Search Puzzle, yang ditandai dengan kotak yang berisi huruf-huruf di mana kata-kata disembunyikan ke berbagai arah. Kami menganalisis kompleksitas, penggunaan memori, dan waktu eksekusi setiap algoritma. Brute-force, meskipun sederhana, tidak efisien untuk kisi-kisi yang lebih besar karena mengeksplorasi semua konfigurasi tanpa pandang bulu. DFS sangat efisien dalam hal memori, menggali jauh ke dalam setiap cabang sebelum melakukan backtracking, sementara BFS, meskipun berpotensi lebih boros memori, secara sistematis mencari setiap level. Eksperimen yang dilakukan pada berbagai ukuran grid menunjukkan bahwa DFS secara umum menawarkan performa terbaik, terutama pada grid yang lebih besar atau lebih kompleks, karena kompleksitas ruang yang lebih rendah dan eksplorasi solusi yang lebih cepat. BFS dan brute-force, meskipun bermanfaat dalam kondisi tertentu, menunjukkan keterbatasan dalam skalabilitas dan efisiensi komputasi.

Keywords— Word Search Puzzles, Algorithm Comparison, Depth-First Search (DFS), Breadth-First Search (BFS), Brute-force, Computational Efficiency, Algorithmic Analysis, Puzzle Solving

I. PENDAHULUAN

Salah satu jenis permainan teka-teki yang populer sekaligus sulit adalah pencarian kata alias ”*Word Search Puzzle*”. Dalam teka-teki ini, pemain harus menemukan kata-kata tersembunyi di antara kisi-kisi huruf yang tampak disusun secara acak. Menemukan kata-kata dalam kisi-kisi bisa jadi sulit meskipun pada awalnya terlihat sederhana, terutama jika kisi-kisi tersebut berukuran besar dan kata-kata yang tersembunyi ditempatkan dalam pola dan arah yang aneh.

Teka-teki pencarian kata adalah jenis masalah pencarian yang menarik dalam ilmu komputer karena memerlukan pencarian dalam struktur data dua dimensi. Masalah ini dapat diselesaikan dengan menggunakan berbagai teknik algoritmik, seperti *brute-force*, *Depth-First Search* (DFS), dan *Breadth-First Search* (BFS). Setiap pendekatan ini memiliki kualitas, manfaat, dan kekurangan yang unik yang dapat mempengaruhi seberapa baik mereka bekerja dalam pekerjaan pencarian kata.

Algoritma pencarian DFS mencari jalur hingga sedalam mungkin sebelum berbalik dan mencoba rute yang berbeda. Karena penghematan memori algoritma ini, struktur yang menyerupai pohon atau grafik sering menggunakannya. BFS, di sisi lain, memeriksa setiap level secara individual dan, meskipun terkadang membutuhkan lebih banyak memori, menjanjikan untuk menemukan jalur terpendek dalam situasi tertentu.

Tujuan dari makalah ini adalah untuk menyelidiki dan membandingkan kemampuan keempat teknik algoritmik ini dalam konteks pemecahan teka-teki pencarian kata. Kompleksitas implementasi, konsumsi memori, dan waktu eksekusi dari setiap algoritma akan dinilai selama analisis. Penulis bermaksud untuk mempelajari lebih lanjut tentang variasi dan penggunaan setiap pendekatan untuk mendapatkan pemahaman yang lebih dalam tentang algoritme pencarian terbaik untuk skenario pencarian kata yang berbeda.

II. TEORI DASAR

A. Algoritma

Algoritma adalah sekumpulan instruksi yang digunakan untuk menyelesaikan masalah komputasi dan menyelesaikan tugas. Kualitas algoritme ditentukan oleh sejumlah faktor tambahan selain faktor utama, yaitu kemampuan algoritme untuk menyelesaikan masalah yang diberikan. Efektivitas algoritme dalam mendapatkan jawaban yang diinginkan dan jumlah sumber daya yang diperlukan untuk menjalankannya adalah dua faktor lainnya. Untuk mengukur parameter-parameter ini, maka diperlukan untuk mempelajari algoritma.
spesifikasi ini. Secara umum, ada dua jenis parameter algoritma yaitu kompleksitas ruang dan kompleksitas waktu. Kompleksitas ruang biasa dinotasikan dengan S(n) yakni jumlah memori yang dibutuhkan metode terkait dengan ukuran input, dinyatakan sebagai fungsi dari n. Adapun kompleksitas waktu, dengan notasi T(n), adalah jumlah tahapan komputasi (operasi) yang diperlukan untuk menjalankan metode sebagai fungsi dari ukuran input (ukuran input n). Dalam kebutuhan waktu algoritma, terdapat tiga notasi asimptotik yakni O(f(n)) sebagai batas lebih dari kebutuhan waktu algoritma. Ω(g(n))

sebagai batas bawah kebutuhan waktu algoritma, dan Θ(h(n)) jika dan hanya jika O(h(n)) = Ω (h(n)).

Dalam penyelesaian suatu masalah, terdapat strategi-strategi algoritma yang dapat digunakan sebagai pendekatan. Setiap strategi memiliki kelebihan dan kekurangannya masing-masing, baik dari segi kompleksitas waktu atau pun ruang. Pemilihan strategi untuk sebuah penyelesaian akan berbeda untuk penyelesaian lainnya. Diperlukan pemahaman atas kebutuhan dan jenis permasalahan yang dihadapi serta strategi yang akan diimplementasikan untuk kemudian memilih strategi yang tepat. Beberapa contoh trategi algoritma yakni Algoritma *Brute-Forc*e, Algoritma *Greedy*, Algoritma *Divide and Conquer*, Algoritma *Decrease and Conquer*, Algoritma *Breadth First-Search* (BFS), Algoritma *Depth First-Search* (DFS), Algoritma *Backtracking*, Algoritma *Branch and Bound*, dan *Dynamic Programming*.

B. Algoritma Brute-Force

Algoritma *brute-force*, terkadang disebut sebagai pendekatan "naif" atau "paksa", adalah teknik yang jelas dan tidak rumit untuk memecahkan masalah di mana setiap solusi potensial diselidiki secara menyeluruh hingga solusi yang tepat ditemukan. Karena kesederhanaan penggunaan dan efisiensinya dalam menyelesaikan masalah dengan ruang solusi yang terbatas atau kecil, metode ini sering digunakan. Berikut tiga karakteristik dari algoritma *brute-force.*

1. **Kesederhanaan**: Algoritma ini mudah dikembangkan dan diimplementasikan karena tidak memerlukan pemahaman khusus tentang masalah.
2. **Komprehensif**: Metode ini selalu menemukan solusi jika ada dengan mempertimbangkan semua kemungkinan yang ada, menjamin bahwa jawaban yang tepat akan ditemukan.
3. **Efisiensi dalam masalah-masalah kecil**: Brute force sering digunakan sebagai tolok ukur atau pembanding untuk algoritma yang lebih kompleks karena mungkin cukup efektif dalam memecahkan masalah dengan ukuran input yang kecil.

Untuk perihal kompleksitas algoritma *brute-force*, Seluruh jumlah skenario potensial yang harus diselidiki menentukan kompleksitas temporal algoritma brute force. Untuk tugas-tugas yang besar, strategi ini tidak efisien karena kompleksitasnya sering kali bersifat eksponensial. Untuk nilai k yang diberikan, kompleksitas waktu dapat mendekati O($n!$), O(2^n), atau O(n^k).

Terdapat beberapa contoh dari penggunaan algoritma *brute-force*. Dalah hal pencarian seperti membandingkan setiap elemen dalam daftar satu per satu untuk menentukan elemen mana yang terbesar atau terkecil. Dalam hal pengurutan, teknik yang secara terus menerus membandingkan dan menukar elemen hingga elemen-elemen tersebut diurutkan termasuk pengurutan gelembung dan pengurutan pilihan. Algortima *brute*-force juga dapat digunakan untuk pencocokan string melibatkan pencocokan setiap substring dalam teks satu per satu dengan pola yang telah ditentukan sebelumnya sampai ada kecocokan atau teks habis. Dapat juga untuk masalah Komputasi kombinatorial, termasuk di dalamnya adalah 0/1 *Knapsack Problem* dan *Travelling Salesman Problem* (TSP), yang memeriksa setiap kombinasi input yang mungkin.

Algoritma ini memang dapat digunakan dalam banyak hal, tetapi terkadang akan menjadi masalah ketika algoritma memiliki kompleksitas yang tinggi. Hal ini berdampak terhadap waktu yang dibutuhkan untuk mencapai solusi ketika input yang besar digunakan dalam algoritma, yang biasanya akan membuat waktu yang dibutuhkan sangat lama. Selain itu, algoritma brute force sering kali membutuhkan memori dan penggunaan CPU yang besar karena harus menyimpan dan memproses setiap solusi yang mungkin.

Meskipun algoritma brute force memiliki kekurangan yang signifikan, terutama dalam hal efisiensi dan skalabilitas, pendekatan ini masih memiliki nilai dalam komputasi, terutama pada tahap awal pengujian dan pengembangan algoritma, di mana kesederhanaan dan kepastian menemukan solusi adalah prioritas utama.

C. Algoritma Breadth First Search (BFS)

Breadth-First Search (BFS) adalah teknik penjelajahan graf yang sistematis dan berguna dalam berbagai aplikasi, seperti analisis jaringan sosial, pemetaan jaringan, dan pencarian jalur dalam labirin. Algoritma ini menjelajahi graf secara level demi level, memulai dari simpul sumber dan bergerak ke seluruh simpul yang dapat dijangkau dari sumber sebelum melanjutkan ke tingkat yang lebih dalam.

1. Karakteristik BFS
   - Penjelajahan Berbasis Level: BFS memulai dari simpul akar dan terlebih dahulu menjelajahi semua simpul pada level yang sama sebelum bergerak ke level berikutnya.
   - Menggunakan Antrian (FIFO): Untuk menjaga urutan penjelajahan, BFS menggunakan struktur data antrian, yang memungkinkan simpul-simpul dikeluarkan dari antrian sesuai urutan mereka ditambahkan (First In, First Out).
2. Implementasi Algoritma BFS

```
queue.enqueue(start_node)
while not queue.empty()
    node = queue.dequeue()
   for each child of node
        if child is goal
            return solution
        queue.enqueue(child)
```

   - Inisialisasi: Antrian diinisialisasi dengan simpul awal, yang juga ditandai sebagai dikunjungi.
   - Penjelajahan: Keluarkan simpul dari antrian, proses, dan tambahkan semua tetangga yang belum dikunjungi ke dalam antrian.

- Penyelidikan lapis demi lapis: Pastikan semua simpul pada kedalaman atau level saat ini sepenuhnya dieksplorasi sebelum bergerak ke level berikutnya.

3. Kompleksitas Waktu BFS

Kompleksitas waktu untuk BFS adalah O(V + E), di mana V adalah jumlah simpul dan E adalah jumlah tepi dalam graf. Hal ini karena setiap simpul dan setiap tepi diperiksa tepat satu kali selama penjelajahan.

- Kompleksitas Waktu: Proses BFS, seperti DFS, memiliki kompleksitas waktu O(V + E). Di sini, V menyatakan simpul-simpul dari graf dan E menyatakan sisi-sisinya. Keefektifan BFS dalam situasi-situasi dimana investigasi tingkat demi tingkat diperlukan ditunjukkan oleh hubungan liniernya dengan simpul dan sisi.

- Kompleksitas Ruang: BFS mungkin membutuhkan lebih banyak ruang, terutama ketika luas (lebar) graf jauh melebihi kedalamannya. BFS mengharuskan untuk menyimpan antrian dari setiap simpul pada tingkat kedalaman saat ini. Pada graf besar atau jaringan dengan sejumlah besar simpul yang terhubung pada kedalaman yang dangkal, hal ini dapat mengakibatkan penggunaan memori yang berlebihan.

Dalam memilih representasi graf, perlu dipertimbangkan kompleksitas operasional dan efisiensi penelusuran. Pendekatan dengan simpul dan sisi memerlukan pemeriksaan setiap simpul dan tepi, sementara operasi antrian dapat memberikan kompleksitas waktu yang lebih baik. Namun, representasi dengan daftar ketetanggaan memungkinkan BFS untuk menelusuri tetangga dengan lebih efisien. Pemilihan tergantung pada kebutuhan aplikasi dan penyeimbangan antara kompleksitas dan efisiensi.

4. Aplikasi Utama BFS

- Pencarian Jalur Terpendek: Dalam graf tanpa bobot, BFS efektif untuk menemukan jalur terpendek dari satu simpul ke semua simpul lain.

- Analisis Jaringan Sosial: Mengidentifikasi level hubungan dalam jaringan, yang berguna untuk mengukur derajat pemisahan antar individu.

- Pemodelan Penyebaran: Dalam epidemiologi, BFS dapat digunakan untuk mensimulasikan penyebaran penyakit atau informasi.

5. Dampak pada Praktik

Breadth-First Search (BFS) adalah pilihan yang tepat untuk masalah yang memerlukan eksplorasi berbasis level, di mana semua simpul pada tingkat tertentu harus dijelajahi sebelum melanjutkan ke tingkat yang lebih dalam. Namun, perlu diingat bahwa penggunaan BFS dapat menyebabkan overhead memori yang signifikan. Hal ini disebabkan oleh kebutuhan untuk menyimpan semua simpul dari level saat ini dalam antrian. Terutama pada graf yang besar atau sangat terhubung, ini dapat menjadi masalah, karena jumlah simpul yang harus disimpan dalam antrian dapat meningkat secara eksponensial seiring dengan kedalaman eksplorasi. Oleh karena itu, sementara BFS cocok untuk eksplorasi berbasis level, harus dipertimbangkan juga overhead memori yang mungkin timbul.

D. Algoritma Depth First Search (DFS)

Sebuah teknik penjelajahan graf dasar yang disebut *Depth-First Search* (DFS) menjelajahi sejauh mungkin setiap cabang graf sebelum berbalik. Teknik ini diterapkan secara luas dalam berbagai konteks, mulai dari analisis jaringan hingga pemecahan teka-teki..

1. Karakteristik DFS

- Mundur ke belakang (*backtracking*): DFS mengeksplorasi setiap cabang dari graf secara penuh, kembali ketika menemukan sebuah simpul tanpa ada tetangga yang belum dijelajahi atau jalan buntu.

- Implementasi: Rekursi, yang memanfaatkan tumpukan panggilan sistem, atau tumpukan (secara eksplisit) dapat digunakan untuk mengimplementasikan DFS.

- Utilitas: Sangat berguna untuk masalah-masalah yang membutuhkan investigasi terhadap setiap opsi; ini membuatnya sesuai untuk tugas-tugas seperti pencarian jalur dan pemecahan teka-teki di mana setiap opsi perlu dipertimbangkan.

2. Implementasi Algoritma DFS

```
function DFS(node)
    if node is a goal
        return solution
    for each child of node
        DFS(child)
```

- Inisialisasi: Mulai dari simpul akar, inisialisasi simpul tersebut dengan mendorongnya ke dalam tumpukan dan menandainya sebagai telah dikunjungi.

- Penjelajahan mendalam: Terus tambahkan simpul-simpul tetangga yang belum dijelajahi ke dalam tumpukan untuk menelusuri lebih jauh ke dalam graf.

- Penelusuran mundur: Untuk menyelidiki rute-rute yang berbeda, keluarkan sebuah simpul dari tumpukan ketika simpul tersebut tidak memiliki tetangga yang belum dijelajahi dan kembali ke simpul sebelumnya.

3. Kompleksitas Waktu DFS

Kompleksitas waktu untuk DFS adalah O(V + E), di mana V adalah jumlah simpul dan E adalah jumlah tepi dalam graf. Hal ini karena setiap simpul dan setiap tepi diperiksa tepat satu kali selama penjelajahan.

- Kompleksitas Waktu: Proses DFS memiliki kompleksitas waktu O(V + E), yang sebanding dengan BFS. Dalam hal ini, V adalah simpul-simpul graf dan E adalah sisi-sisinya
- Kompleksitas Ruang: Mungkin lebih hemat ruang daripada BFS, terutama dalam situasi dimana kedalaman graf sangat melebihi luasnya. Tidak seperti BFS, yang harus menyimpan semua simpul pada kedalaman yang tetap, tumpukan hanya meluas sedalam jalur terpanjang dari akar ke daun.

4. Aplikasi Utama BFS

- Komponen Terhubung: DFS memfasilitasi identifikasi berbagai komponen yang terhubung dalam sebuah grafik.
- Pemecahan Teka-teki: Sangat membantu dalam situasi yang membutuhkan pencarian yang komprehensif dan mendalam, seperti teka-teki delapan ratu atau pemecahan labirin.

5. Dampak pada Praktik

Implementasi rekursif Depth-First Search (DFS) dapat mempengaruhi sumber daya sistem karena kedalaman tumpukan panggilan, terutama pada grafik yang sangat dalam atau besar. Meskipun demikian, DFS memiliki efisiensi memori yang lebih baik dibandingkan BFS, karena hanya perlu menyimpan tumpukan simpul yang mewakili jalur saat ini yang sedang diselidiki. Hal ini bermanfaat terutama pada graf yang jarang, di mana DFS dapat mengkonsumsi lebih sedikit memori daripada BFS. DFS juga cocok untuk penjelajahan yang berorientasi pada kedalaman, memungkinkannya mencapai solusi dengan cepat dalam situasi di mana tidak perlu mengunjungi setiap simpul. Meskipun operasi tumpukan dalam DFS memiliki kompleksitas O(1), penggunaan daftar ketetanggaan (Adjacency List) dapat menyebabkan kompleksitas O(E) karena perlu mengeksplorasi setiap daftar tetangga secara menyeluruh, dengan setiap sisi dipertimbangkan hanya sekali.

E. Word Search Puzzle

Word Search Puzzle, juga dikenal sebagai pencarian kata atau teka-teki silang, adalah teka-teki di mana kata-kata disembunyikan dalam grid huruf yang besar. Pemain harus menemukan dan menandai semua kata yang tersembunyi di dalam grid tersebut. Kata-kata dapat ditempatkan secara horizontal, vertikal, diagonal, dan bahkan mungkin tertulis mundur atau dalam urutan yang terbalik. Teka-teki ini populer di kalangan semua kelompok usia dan sering digunakan dalam pendidikan sebagai alat pembelajaran dan latihan bahasa.

1. Konsep dan Struktur

- Grid: Word search terdiri dari grid persegi atau persegi panjang yang diisi dengan huruf acak dan kata-kata tersembunyi. Ukuran grid dapat bervariasi tergantung pada tingkat kesulitan dan jumlah kata yang harus ditemukan.
- Kata Tersembunyi: Kata-kata yang harus dicari biasanya diberikan di samping atau di bawah grid. Kata-kata ini bisa berkaitan dengan tema tertentu seperti binatang, nama kota, ilmu pengetahuan, atau acara tertentu, yang menambah elemen pendidikan atau personalisasi.
- Orientasi Kata: Kata-kata dalam teka-teki dapat muncul dalam delapan kemungkinan arah—horisontal kiri ke kanan, horisontal kanan ke kiri, vertikal atas ke bawah, vertikal bawah ke atas, dan empat arah diagonal. Keberadaan arah yang berbeda ini menambah tingkat kesulitan dalam menemukan kata-kata.

2. Cara Bermain

Pemain membaca daftar kata yang diberikan dan mencari masing-masing kata dalam grid. Setelah menemukan kata, pemain menandai kata tersebut dalam grid. Ini bisa dilakukan dengan menggarisbawahi, mencoret, atau menggunakan warna berbeda. Teka-teki dianggap selesai ketika semua kata dalam daftar telah ditemukan dan ditandai dalam grid.

3. Tantangan dalam Desain Word Search

- Keseimbangan Kesulitan: Pembuat teka-teki harus menyeimbangkan kesulitan teka-teki sehingga menantang namun tidak terlalu sulit sehingga menyebabkan frustrasi.
- Tema dan Kosakata: Memilih tema yang relevan dan menarik serta kata-kata yang sesuai dan bervariasi adalah penting untuk membuat teka-teki yang menarik dan pendidikan.

III. PERCOBAAN PERBANDINGAN DALAM PENYELESAIAN *WORD SEARCH PUZZLE*

A. Percobaan Algoritma Brute-Force

Algoritma Brute Force adalah pendekatan langsung untuk memecahkan masalah dengan mencoba setiap opsi yang mungkin sampai menemukan solusi. Metode ini menempatkan kata-kata dalam sebuah masalah dengan memilih lokasi dan arah secara acak hingga kata-kata tersebut cocok atau kehabisan percobaan. Dalam notasi pseudocode, berikut algoritma yang digunakan.

```
Function solve_puzzle():
  For each word in the list of words:
    For each cell in the grid:
      If the cell's letter matches the first
letter of the word:
```

```
            For each of the eight possible
directions:
                If check_word() for this direction
starting from this cell is true:
                    Mark the word as found in the
grid
                        Break out of the loop
after finding the word
```

Kompleksitas waktu dari kode Solver terutama ditentukan oleh beberapa loop bersarang yang mencakup pencarian setiap kata dalam daftar pada setiap sel dalam grid dan setiap kemungkinan arah dari sel tersebut. Mari kita analisis dengan lebih detail:

- Loop kata: Loop luar mengiterasi setiap kata dalam daftar kata self.words. Jika ada W kata dalam daftar, maka loop ini berjalan W kali.
- Loop sel grid: Untuk setiap kata, kode kemudian mengiterasi setiap sel dalam grid. Jika grid berukuran H (tinggi) x W (lebar), maka ada H*W sel yang akan diiterasi.
- Loop arah: Untuk setiap sel yang cocok dengan huruf pertama dari kata yang sedang diperiksa, kode menguji delapan arah yang mungkin untuk mencari kata tersebut. Jadi, untuk setiap sel yang cocok, delapan pemeriksaan akan dilakukan.
- Pemeriksaan kata: Dalam check_word(), setiap kata dicek sepanjang panjangnya, katakan L, untuk memastikan setiap huruf cocok di grid dalam arah tertentu.

Jadi, kompleksitas waktu kasar dari algoritma ini adalah O(W * h * w * 8 * L), dimana:

- W adalah jumlah kata.
- h * w adalah ukuran grid.
- 8 adalah jumlah arah yang mungkin.
- L adalah panjang rata-rata kata.

Hal ini menjadikan algoritma ini sangat tidak efisien, terutama untuk grid besar dengan banyak kata atau kata yang panjang, karena setiap kemungkinan lokasi dan arah dari setiap kata harus diperiksa secara menyeluruh.

Kompleksitas ruang dari kode ini terutama ditentukan oleh:

- Grid self.color_codes: Membutuhkan ruang sebesar H * W, sesuai dengan dimensi grid asli.
- Simpanan Grid Asli: Kita asumsikan self.puzzle.grid juga berukuran H * W.
- Daftar Kata: Ruang yang diperlukan untuk menyimpan kata tergantung pada jumlah kata dan panjang rata-rata mereka. Dalam konteks ini, kita anggap hanya referensi yang disimpan, sehingga tidak secara signifikan mempengaruhi kompleksitas ruang keseluruhan dibandingkan dengan grid.

Oleh karena itu, kompleksitas ruang adalah O(H * W), yang sebagian besar ditentukan oleh penyimpanan dua grid (asli dan grid kode warna).

Secara keseluruhan, kinerja dengan *brute-force* sangat tergantung pada ukuran grid0 dan jumlah kata yang perlu ditemukan, serta panjang kata-kata tersebut. Meskipun brute-force mudah diimplementasikan, ia mungkin tidak praktis untuk kasus penggunaan dengan parameter besar atau dalam lingkungan dengan keterbatasan sumber daya yang ketat.

B. Percobaan Algoritma Breadth First-Search

Pada algoritma BFS, proses dimulai dengan mengulang setiap kata dalam daftar `words`. Untuk setiap kata, algoritma mencari seluruh grid untuk mencocokkan huruf pertama kata tersebut dengan konten sel. Jika cocok, BFS dilakukan dari sel tersebut ke semua arah yang mungkin (seperti atas, bawah, kiri, kanan, dan diagonal). Ini menggunakan antrian untuk mengelola posisi saat ini dan indeks karakter dari kata. Pada setiap iterasi, algoritma mengecek jika posisi saat ini telah mencapai akhir kata, jika ya, fungsi `mark_word` akan dipanggil untuk menandai jalur kata di grid. Jika belum, posisi selanjutnya dihitung dan ditambahkan ke antrian jika memenuhi syarat. Proses ini berhenti segera jika kata ditemukan, dan loop untuk arah dan sel pun berakhir.

```
function BFSSolver:
  For each word in words:
     For each cell in the grid:
        If cell content matches the first letter
of the word:
           For each direction in possible
directions:
              Initialize a queue
              Enqueue the initial position
(row_index, col_index) and character index 0
                 While queue is not empty:
                    Dequeue to get current
position (x, y) and char_index
                    If char_index is the last
index of the word:
                       Call mark_word to
backtrack and mark the word's path in the grid
                          Set found to True
                          Break from the while
loop
                    Compute next position (nx,
ny) based on the current direction (dx, dy)
                    Increment the character index
to get next_char_index
```

```
                    If next position is within
grid bounds and matches the next character of the
word:
                        Enqueue the next position
and next_char_index
                If word is found:
                    Break from the direction loop
            If word is found:
                Break from the cell loop
```

Kompleksitas waktu dari kode tersebut sangat bergantung pada beberapa faktor:

- W: jumlah kata dalam daftar words
- h x w: dimensi grid (M baris dan N kolom)
- L: panjang rata-rata dari kata-kata dalam words

Kompleksitas waktu dapat diperkirakan sebagai O(W×M×N×L). Hal ini karena untuk setiap kata, kode tersebut mencoba mencocokkan dan menjelajahi dari setiap sel dalam grid untuk setiap arah hingga kedalaman maksimum sepanjang kata.

Kompleksitas ruang terutama dipengaruhi oleh:

- Ukuran antrian: Dalam kasus terburuk, antrian mungkin perlu menyimpan semua posisi yang mungkin dijelajahi selama BFS. Dalam skenario terburuk, ini dapat sebesar O $(M \times N)$, terutama jika kata yang dicari memiliki banyak kemungkinan posisi awal dan arah yang berpotensi menyebar ke seluruh grid
- Mark_word: Fungsi ini mungkin memerlukan tambahan ruang jika implementasi memodifikasi grid atau menyimpan jalur yang ditemukan.

Dengan demikian, kompleksitas ruangnya adalah $O(M \times N)$, terutama karena kebutuhan ruang untuk antrian dalam BFS.

C. Percobaan Algoritma Depth First-Search

Pada DFS, dibuat sebuah fungsi bernama DFSSolver. Fungsi ini mengiterasi setiap kata dalam word_list dan mencoba menemukan setiap kata tersebut di dalam grid menggunakan DFS. Untuk setiap kata, algoritma mencari seluruh sel dalam grid yang cocok dengan huruf pertama kata tersebut. Jika ditemukan, fungsi dfs_search dipanggil untuk setiap arah yang mungkin (misalnya ke atas, ke bawah, ke kiri, ke kanan, dan diagonal) dari sel tersebut. Jika dfs_search berhasil menemukan kata tersebut, kata itu akan ditandai di grid dan pencarian berhenti untuk kata tersebut. Jika tidak ditemukan, akan dicetak pesan bahwa kata tersebut tidak ditemukan.

Adapun untuk fungsi dfs_search, dilakukan pencarian DFS dari sebuah sel cell pada grid untuk menemukan sisa kata word mulai dari indeks karakter char_index. Pencarian dilakukan bergerak ke next_cell sesuai dengan direction yang diberikan. Jika char_index mencapai panjang kata (word_length - 1), maka kata tersebut berhasil ditemukan. Jika next_cell valid dan karakter pada next_cell sesuai dengan karakter berikutnya yang diharapkan pada kata, fungsi akan memanggil dirinya sendiri dengan next_cell dan indeks karakter yang ditingkatkan.

```
function DFSSolver:
  for each word in word_list:
    found = false
    for each starting cell in grid:
      if grid[cell] == word[0]:
      for each direction:
         if dfs_search(starting_cell, direction,
word, 0):
           mark_word(word, starting_cell,
direction)
           found = true
           break
    if found:
       break
  if not found:
     print word not found


function dfs_search(cell, direction, word,
char_index):
  if char_index == word_length - 1:
    return true
  next_cell = move in direction
  if next_cell is valid and grid[next_cell] ==
word[char_index + 1]:
    if dfs_search(next_cell, direction, word,
char_index + 1):
      return true
  return false
```

Kompleksitas waktu total adalah hasil dari pengulangan pencarian kata dalam grid untuk setiap kata di dalam word_list. Setiap kata dicari dengan mengiterasi setiap sel pada grid dan mencoba setiap arah yang mungkin dalam kasus terburuk. Dengan demikian, kompleksitas waktunya adalah: O(W×M×N×4^L)

- W adalah jumlah kata dalam word_list.
- M dan N adalah dimensi dari grid.
- 4^L adalah faktor yang berasal dari kemungkinan ekspansi pada setiap langkah rekursi dalam pencarian DFS, yang bisa meluas ke empat arah yang mungkin, untuk kata dengan panjang maksimum L.

Kompleksitas ruang total dipengaruhi oleh kedalaman rekursi maksimum yang terjadi selama operasi DFS. Kedalaman ini tergantung pada panjang kata terpanjang yang sedang dicari, karena rekursi DFS akan menyelam sepanjang kata itu: O(L)

- L adalah panjang maksimum kata yang dicari, karena ini akan menentukan seberapa dalam stack call bisa bertambah selama operasi rekursi.

Kompleksitas ruang tidak bergantung secara langsung pada jumlah kata atau ukuran grid, tetapi pada kedalaman rekursi yang diinduksi oleh panjang kata terpanjang dalam pencarian.

D. Perbandingan dari Percobaan

Perbandingan dilakukan dengan mencoba melakukan penyelesaian untuk puzzle yang sama untuk setiap algoritma. Berikut hasil percobaan yang dilakukan dengan puzzle 10x10 sebagai berikut.

```
Generated Puzzle:
c g b q v e t u r b
e i p r o k x n l t
c c h r e c e x n g
b w r t i a m q n f
f h f o p b d i t r
k c i e f e k t l p
r r r l w c d p h h
f a s c a q n u g f
e e t r i e j s e m
q s t o d k s a i h
```

Dengan daftar kata: ['back', 'tracking', 'breadth', 'depth', 'first', 'search', 'brute', 'force', 'algorithm']

- Brute-force:

```
Normal Solver starts solving:
Puzzle after solving:
c g b q v e t u r b
e i p r o k x n l t
c c h r e c e x n g
b w r t i a m q n f
f h f o p b d i t r
k c i e f e k t l p
r r r l w c d p h h
f a s c a q n u g f
e e t r i e j s e m
q s t o d k s a i h
==================================
Time taken: 0.0 ms
```

- BFS:

```
BFS Solver starts solving:
Puzzle after solving:
c g b q v e t u r b
e i p r o k x n l t
c c h r e c e x n g
b w r t i a m q n f
f h f o p b d i t r
k c i e f e k t l p
r r r l w c d p h h
f a s c a q n u g f
e e t r i e j s e m
q s t o d k s a i h
==================================
Time taken: 15.624523162841797 ms
```

- DFS:

```
DFS Solver starts solving:
Puzzle after solving:
c g b q v e t u r b
e i p r o k x n l t
c c h r e c e x n g
b w r t i a m q n f
f h f o p b d i t r
k c i e f e k t l p
r r r l w c d p h h
f a s c a q n u g f
e e t r i e j s e m
q s t o d k s a i h
==================================
Time taken: 0.0 ms
```

Kemudian dilakukan percobaan dengan peningkatan grid. Dengan puzzle digenerate secara random untuk setiap besar grid.

Grid	Waktu yang dibutuhkan dengan Algoritma (ms)		
	Brute-force	BFS	DFS
20x20	0.0	20.0846	1.9548
30x30	4.8892	23.8624	3.8095
40x40	14.3547	21.5271	0.9971
50x50	12.1694	14.6248	0.8545
60x60	16.5482	21.2212	0.5260
70x70	7.4086	14.0979	6.0568
80x80	3.6352	24.5194	2.0051
90x90	10.7093	16.0055	1.9999
100x100	5.1041	19.9046	1.9975
150x150	9.0559	17.4501	3.3853

E. Perbandingan dari Percobaan

Pada eksperimen yang dilakukan untuk memecahkan Word Search Puzzle dengan menggunakan tiga algoritma yang berbeda—Brute-Force, Breadth First Search (BFS), dan Depth First Search (DFS)—masing-masing algoritma menunjukkan karakteristik kinerja yang unik yang tergantung pada ukuran grid dan kompleksitas dari kata-kata yang dicari.

1. Brute-Force

Algoritma Brute-Force, meskipun mudah untuk diimplementasikan, menunjukkan kinerja yang kurang efisien dibandingkan dengan algoritma lainnya. Kelemahannya terletak pada kebutuhan untuk mencoba setiap kemungkinan posisi dan arah kata tanpa memanfaatkan struktur data yang lebih cerdas atau heuristik. Dalam eksperimen, algoritma ini cenderung lebih lambat, terutama saat ukuran grid meningkat. Namun, dalam grid kecil, ia dapat berkinerja cukup cepat karena jumlah kemungkinan yang harus dicoba masih terbatas.

2. Breadth First Search (BFS)

BFS menunjukkan peningkatan kinerja dibandingkan dengan Brute-Force, karena algoritma ini sistematis dalam mencari dari titik awal yang cocok ke semua arah yang mungkin menggunakan antrian. Ini memungkinkan penemuan kata yang lebih cepat dalam beberapa kasus. Namun, karena menyimpan banyak posisi grid dalam antrian, kompleksitas ruangnya dapat menjadi pertimbangan, terutama di grid besar. Namun, dalam beberapa kasus pada grid yang lebih besar, BFS menunjukkan waktu yang lebih konsisten, menandakan kemampuannya dalam mengelola eksplorasi lebih efisien daripada Brute-Force.

3. Depth First Search (DFS)

DFS muncul sebagai algoritma yang paling efisien di antara ketiganya dalam banyak kasus, terutama dalam grid yang lebih besar. Algoritma ini dapat mencapai kedalaman maksimal dengan cepat dan kembali jika solusi tidak ditemukan, meminimalkan jumlah operasi yang tidak perlu. Hal ini juga menunjukkan kompleksitas ruang yang lebih rendah karena hanya memerlukan kedalaman rekursi yang sesuai dengan panjang kata. Namun, jika kata yang dicari sangat panjang, kompleksitas waktunya bisa meningkat secara signifikan, tergantung pada arah eksplorasi.

4. Pengamatan dari Eksperimen

Dari data eksperimental, terlihat bahwa dengan meningkatnya ukuran grid, algoritma DFS cenderung memberikan performa waktu yang lebih baik, mengindikasikan skalabilitasnya yang efektif. Sementara itu, BFS menunjukkan konsistensi yang lebih baik dibandingkan dengan Brute-Force yang mengalami penurunan kinerja yang signifikan saat ukuran grid meningkat.

IV. KESIMPULAN

Dari percobaan yang telah dilakukan, terlihat bahwa dalam konteks *Word Search Puzzles*, pendekatan DFS dapat dibilang lebih baik dibandingkan pendekatan BFS dan *Brute-Force* (terutama untuk grid yang lebih besar atau kata-kata yang lebih kompleks). Penggunaan DFS tidak hanya mengurangi waktu yang dibutuhkan untuk menemukan kata-kata dalam puzzle, tetapi juga lebih efisien dalam penggunaan sumber daya, yang menjadikan mereka lebih cocok untuk aplikasi praktis yang memerlukan penyelesaian puzzle yang cepat dan efisien. DFS menawarkan keseimbangan terbaik antara kompleksitas waktu dan ruang, menjadikannya pilihan yang ideal untuk grid besar dan kata-kata yang lebih panjang.

VIDEO LINK AT YOUTUBE

https://youtu.be/oZbkDf-kue8

UCAPAN TERIMA KASIH

Penulis ingin mengucapkan terima kasih kepada berbagai pihak yang terlibat. Penulis pertama-tama mengucapkan terima kasih kepada Allah SWT atas rahmat dan karunia-Nya yang telah memungkinkan penulis menyelesaikan makalah berjudul "Pendekatan Algoritma dalam Penyelesaian Puzzle Pencarian Kata: Sebuah Perbandingan antara DFS, BFS, dan Brute-force". Selain itu, saya ingin mengucapkan terima kasih kepada Bapak Dr. Ir. Rinaldi Munir, M.T., dosen mata kuliah Strategi Algoritma saya, yang telah memberikan bimbingan kepada penulis selama kuliah.

REFERENSI

[1] Munir, Rinaldi. 2024. "Pengantar Strategi Algoritma" https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Pengantar-Strategi-Algoritma-(2024).pdf (Diakses 11 Juni 202)

[2] Munir, Rinaldi. 2024. "Algoritma Brute Force (Bagian 1)" https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Brute-Force-(2022)-Bag2.pdf (Diakses 12 Juni 202)

[3] Munir, Rinaldi. 2024. "Algoritma Brute Force (Bagian 2)" https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2022-2023/Pengantar-Strategi-Algoritma-(2023).pdf (Diakses 12 Juni 202)

[4] Munir, Rinaldi. 2024. " Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 1)" https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/BFS-DFS-2021-Bag1-2024.pdf (Diakses 12 Juni 202)

[5] Munir, Rinaldi. 2024. " Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 2)" https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf (Diakses 12 Juni 202)

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, 12 Juni 2024

Zahira Dina Amalia 1352085