LAPORAN TUGAS KECIL 3
IF2211 STRATEGI ALGORITMA

Penyelesaian *Puzzle Rush Hour* Menggunakan Algoritma *Pathfinding*

Disusun Oleh:

Grace Evelyn Simon	13523087
Ahmad Ibrahim	13523089

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2025

DAFTAR ISI

DAFTAR GAMBAR

BAB 1
DESKRIPSI TUGAS

Rush Hour merupakan sebuah permainan *puzzle* logika berbasis *grid* yang menantang pemain untuk menggeser kendaraan di dalam sebuah kotak (biasanya berukuran 6x6) agar mobil utama (biasanya berwarna merah) dapat keluar dari kemacetan melalui pintu keluar di sisi papan.

Setiap kendaraan hanya bisa bergerak lurus ke depan atau ke belakang sesuai dengan orientasinya (horizontal atau vertikal), dan tidak dapat berputar. Tujuan utama dari permainan ini adalah memindahkan mobil merah ke pintu keluar dengan jumlah langkah seminimal mungkin. Terdapat beberapa komponen penting dalam permainan *Rush Hour*, yakni:

1. **Papan** – Papan merupakan tempat permainan dimainkan.
   Papan terdiri atas *cell*, yaitu sebuah *singular point* dari papan. Sebuah *piece* akan menempati *cell-cell* pada papan. Ketika permainan dimulai, semua *piece* telah diletakkan di dalam papan dengan konfigurasi tertentu berupa lokasi *piece* dan orientasi, antara horizontal atau vertikal. Hanya *primary piece* yang dapat digerakkan keluar papan melewati pintu keluar. *Piece* yang bukan *primary piece* tidak dapat digerakkan keluar papan. Papan memiliki satu pintu keluar yang pasti berada di dinding papan dan sejajar dengan orientasi *primary piece*.

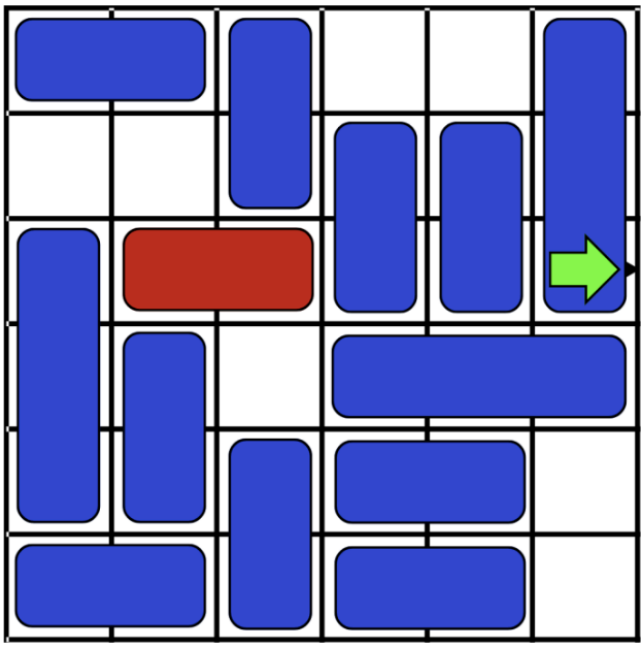
2. ***Piece*** – *Piece* adalah sebuah kendaraan di dalam papan.
   Setiap piece memiliki posisi, ukuran, dan orientasi. Orientasi sebuah *piece* hanya dapat berupa horizontal atau vertikal–tidak mungkin diagonal. *Piece* dapat memiliki beragam ukuran, yaitu jumlah *cell* yang ditempati oleh *piece*. Secara standar, variasi ukuran sebuah *piece* adalah 2-*piece* (menempati 2 *cell*) atau 3-*piece* (menempati 3 *cell*). Suatu *piece* tidak dapat digerakkan melewati/menembus *piece* yang lain.

3. ***Primary Piece*** – *Primary piece* adalah kendaraan utama yang harus dikeluarkan dari papan (biasanya berwarna merah). Hanya boleh terdapat satu *primary piece*.

4. **Pintu Keluar** – Pintu keluar adalah tempat *primary piece* dapat digerakkan keluar untuk menyelesaikan permainan.

5. **Gerakan** — Gerakan yang dimaksudkan adalah pergeseran *piece* di dalam permainan. *Piece* hanya dapat bergerak/bergeser lurus sesuai orientasinya (atas-bawah jika vertikal dan kiri-kanan jika horizontal). Suatu *piece* tidak dapat digerakkan melewati/menembus *piece* yang lain.

Dalam tugas ini, kami diminta untuk membuat suatu program sederhana dalam bahasa C/C++/Java/Javascript yang mengimplementasikan algoritma *pathfinding Greedy Best First Search*, UCS (*Uniform Cost Search*), dan A* dalam menyelesaikan permainan *Rush Hour*. Algoritma *pathfinding* ini minimal diimplementasikan menggunakan satu heuristik (dua atau lebih apabila mengerjakan bonus) yang ditentukan sendiri. Algoritma dijalankan secara terpisah dan ditentukan berdasarkan *input* dari pengguna.



Gambar 1. Ilustrasi Permainan *Rush Hour*

BAB 2
TEORI DASAR

2.1 Penjelasan Algoritma

Pathfinding merupakan proses menemukan rute terbaik antara dua buah titik dalam sebuah peta, graf, atau ruang keadaan (*space state*). *Searching* dalam konteks *pathfinding* adalah proses eksplorasi sistematis dari berbagai kemungkinan jalur dari titik awal hingga titik tujuan. Algoritma pencarian akan:

1. Memulai dari *state*/*node* awal.
2. Membangkitkan *state*/*node* tetangga (suksesor) yang bisa dicapai.
3. Memilih salah satu suksesor untuk dieksplorasi lebih lanjut.
4. Mengulangi proses ini hingga *state*/*node* tujuan ditemukan atau semua kemungkinan telah dieksplorasi tanpa menemukan tujuan.

Algoritma pencarian menyimpan informasi tentang *state* mana yang sudah dikunjungi untuk menghindari pemrosesan ulang dan *loop* tak terbatas, serta jalur yang telah ditempuh untuk merekonstruksi solusi. Terdapat dua jenis algoritma dalam *searching*, yakni *uninformed search* dan *informed search*.

1. *Uninformed Search*

   Uninformed search atau dikenal juga dengan sebutan *blind search* merupakan teknik pencarian yang tidak menggunakan informasi tambahan di luar definisi masalah. Algoritma ini menjelajahi ruang pencarian secara buta tanpa mempertimbangkan biaya atau kemungkinan mencapai sebuah solusi. Contoh algoritma *uninformed search* adalah BFS, DFS, DLS, IDS, dan UCS.

2. *Informed Search*

   Informed search atau dikenal juga dengan sebutan *heuristic search* merupakan teknik pencarian yang menggunakan informasi tambahan, seperti heuristik untuk memandu

proses pencarian dan mencari solusi yang lebih efisien. Contoh algoritma *informed search* adalah *Greedy Best First Search* dan A* *Search*.

2.2 Algoritma *Uniform Cost Search* (UCS)

UCS adalah sebuah algoritma pencarian graf yang digunakan untuk menemukan jalur terpendek dari simpul awal ke simpul tujuan dalam sebuah graf berbobot. Algoritma UCS dianggap sebagai perpanjangan dari algoritma BFS, namun dengan mempertimbangkan nilai atau biaya dari setiap jalur. Algoritma ini bertujuan untuk menemukan sebuah solusi optimal dengan memprioritaskan jalur dengan biaya terendah. Cara kerja algoritma ini adalah dengan menggunakan antrian prioritas untuk menyimpan simpul yang akan dieksplorasi, dengan prioritas berdasarkan biaya jalur dari simpul awal. Kompleksitas waktu dari algoritma UCS adalah $O(b^d)$ dengan b adalah faktor percabangan pohon pencarian dan d adalah kedalaman simpul tujuan.

2.3 Algoritma *Iterative Deepening Search* (IDS)

IDS adalah sebuah strategi pencarian yang menggunakan metode pencarian kedalaman-pertama (DFS/*Depth First Search*) yang dibatasi kedalamannya secara berulang dengan meningkatkan batas kedalaman. IDS menggabungkan kelebihan DFS dan BFS dengan menggunakan lebih sedikit memori tetapi tetap menjamin solusi terpendek. Cara kerja algoritma ini adalah dengan memulai dari batas kedalaman nol, lalu meningkatkan batas ini secara berulang. Setelah solusi ditemukan, IDS menjamin solusi terpendek karena menjelajahi semua *node* pada level yang lebih rendah sebelum mencapai *node* pada level yang lebih tinggi. Kompleksitas waktu dari algoritma IDS adalah $O(b^d)$ dengan b adalah faktor percabangan pohon pencarian dan d adalah kedalaman simpul tujuan.

2.4 Algoritma *Greedy Best First Search* (GBFS)

GBFS adalah algoritma pencarian yang menggunakan pendekatan heuristik untuk mencari solusi terbaik dengan selalu memilih simpul yang memiliki estimasi biaya terendah menuju tujuan. Berbeda dengan algoritma UCS yang mempertimbangkan biaya aktual dari jalur yang ditempuh, GBFS berfokus pada biaya heuristik, yakni nilai yang memberikan gambaran mengenai seberapa dekat atau jauh simpul tertentu dari simpul tujuan. Meskipun

GBFS memiliki keunggulan dalam memilih arah yang tampak paling menguntungkan, algoritma ini tidak menjamin solusi yang paling efisien. Hal ini disebabkan karena GBFS tidak mempertimbangkan biaya dari jalur yang ditempuh. Akibatnya, dalam beberapa kasus, algoritma ini dapat tersesat atau terjebak dalam jalur yang terlihat menjanjikan, namun sebenarnya lebih panjang daripada algoritma yang paling efisien. Kompleksitas waktu dari algoritma IDS adalah $O(b^d)$ dengan b adalah faktor percabangan pohon pencarian dan d adalah kedalaman simpul tujuan.

2.5 Algoritma A*

A* adalah algoritma pencarian yang digunakan untuk menemukan jalur terpendek antara titik awal dan titik akhir. Algoritma ini adalah salah satu metode algoritma yang paling banyak digunakan dalam ilmu komputer dan kecerdasan buatan untuk menemukan jalur terpendek atau solusi yang optimal. Dalam algoritma A*, simpul-simpul dalam struktur data diberi dua jenis nilai, yakni biaya aktual dari simpul awal ke simpul saat ini dan biaya heuristik dari simpul saat ini ke simpul tujuan. Biaya heuristik adalah perkiraan seberapa jauh simpul tertentu dari simpul tujuan, yang dalam hal ini kami mengimplementasikannya menggunakan metode jarak Manhattan. Algoritma ini dilakukan dengan cara mengevaluasi simpul-simpul ke dalam antrian prioritas, dengan simpul yang memiliki gabungan biaya aktual + biaya heuristik terendah yang menjadi prioritas untuk dieksplorasi terlebih dahulu. Dalam setiap langkah, algoritma memilih simpul dengan total biaya terendah, lalu mengembangkan simpul tersebut dengan menambahkan simpul-simpul tetangganya ke dalam antrian prioritas. Proses ini akan terus berulang sampai semua simpul telah dieksplorasi tanpa menemukan solusi. Notasi yang dipakai oleh algoritma A* adalah $f(n) = g(n) + h(n)$ dengan f(n) adalah biaya estimasi terendah, g(n) adalah biaya dari *node* awal ke *node* n, dan h(n) adalah perkiraan biaya dari *node* n ke *node* tujuan.

BAB 3
ANALISIS DAN IMPLEMENTASI ALGORITMA

3.1 Analisis Algoritma

Dalam upaya menyelesaikan permainan *Rush Hour*, berbagai algoritma pencarian jalur (*pathfinding*) dapat diterapkan, masing-masing dengan karakteristik dan efektivitasnya sendiri. Pemahaman mendalam mengenai komponen-komponen algoritma ini, seperti fungsi evaluasi dan sifat heuristik merupakan faktor yang krusial. Salah satu algoritma pencarian terinformasi yang populer adalah A* (A-Star). Kunci dari A* terletak pada fungsi evaluasi heuristiknya, f(n), yang mengestimasi total biaya dari jalur solusi yang melalui simpul n menuju ke tujuan. Fungsi ini merupakan penjumlahan dari dua komponen penting, yakni g(n) dan h(n). Sesuai dengan salindia "Penentuan Rute (*Route*/*Path Planning*) Bagian 2: Algoritma A*", g(n) didefinisikan sebagai biaya aktual yang telah dikeluarkan untuk mencapai simpul n dari simpul awal. Sementara itu, h(n) adalah estimasi biaya (heuristik) dari simpul n untuk mencapai simpul tujuan. Dengan demikian, rumus lengkap dari A* adalah $f(n) = g(n) + h(n)$ untuk mendapatkan rute yang paling optimal.

Keoptimalan solusi yang ditemukan oleh A* sangat bergantung pada sifat fungsi heuristik h(n) yang digunakan. Sebuah heuristik h(n) dikatakan *admissible* jika nilainya tidak pernah melebih-lebihkan (*overestimate*) biaya sebenarnya untuk mencapai tujuan dari simpul n, atau secara formal $h(n) \leq h^*(n)$, dengan $h^*(n)$ adalah biaya sebenarnya. Salindia kuliah menekankan bahwa heuristik yang *admissible* bersifat optimis. Contoh yang diberikan adalah jarak garis lurus yang selalu lebih kecil atau sama dengan jarak tempuh sebenarnya. Dalam konteks *Rush Hour*, heuristik umum seperti "jarak Manhattan mobil utama ('P') ke pintu keluar" atau "jumlah mobil yang menghalangi jalur langsung mobil 'P'" cenderung bersifat *admissible*. Jarak Manhattan adalah jarak terpendek absolut tanpa rintangan, dan adanya mobil lain hanya akan menambah jumlah langkah. Demikian pula, jumlah mobil penghalang memberikan perkiraan minimum langkah yang diperlukan untuk membersihkan jalur. Karena heuristik ini tidak melebih-lebihkan biaya sebenarnya, heuristik pada A* bersifat *admissible*.

Pada dasarnya, untuk penyelesaian *Rush Hour*, algoritma UCS akan berperilaku identik dengan BFS, baik dalam urutan *node* yang dieksplorasi maupun *path* yang dihasilkan. Hal ini disebabkan karena setiap gerakan mobil dalam *Rush Hour* dihitung sebagai satu langkah dengan biaya seragam (yakni biaya 1). UCS bekerja dengan mengeksplorasi *node* berdasarkan biaya kumulatif terendah dari awal (g(n)), sedangkan BFS mengeksplorasi berdasarkan kedalaman terendah. Ketika biaya setiap langkah adalah 1, maka g(n) akan sama dengan kedalaman *node* tersebut. Oleh karena itu, kedua algoritma akan membuat pilihan ekspansi yang sama dan menghasilkan solusi optimal dalam jumlah gerakan.

Mengenai efisiensi, secara teoritis, algoritma A* dengan heuristik yang baik dan *admissible* umumnya lebih efisien dibandingkan UCS pada penyelesaian Rush Hour. UCS, sebagai algoritma *uninformed search*, akan menjelajahi semua kemungkinan jalur secara merata berdasarkan biaya kumulatif dari awal, tanpa panduan menuju tujuan. Ini berarti algoritma UCS dapat menyebabkan eksplorasi banyak cabang yang tidak relevan. Sebaliknya, A*, sebagai *informed search*, menggunakan heuristik h(n) untuk memperkirakan kedekatan dengan tujuan. Hal ini memungkinkan A* untuk memprioritaskan node yang lebih menjanjikan (memiliki f(n) = g(n) + h(n) yang rendah) dan cenderung memangkas cabang-cabang yang diperkirakan mahal atau jauh dari solusi. Akibatnya, A* akan mengeksplorasi jumlah *node* yang jauh lebih sedikit untuk menemukan solusi optimal, terutama pada masalah dengan ruang pencarian besar seperti *Rush Hour*.

Penting untuk dicatat bahwa algoritma *Greedy Best First Search* (*Greedy* BFS) secara teoritis tidak menjamin solusi optimal untuk penyelesaian *Rush Hour*. *Greedy* BFS hanya mempertimbangkan nilai heuristik h(n) untuk memilih *node* berikutnya, mengabaikan biaya g(n) yang telah ditempuh. Sifat "rakus" ini membuatnya cenderung mengambil jalur yang tampak paling dekat dengan tujuan secara lokal, namun bisa jadi jalur tersebut secara keseluruhan lebih panjang atau mahal. Algoritma ini dapat terjebak pada keputusan lokal yang tampak baik namun mengarah ke solusi suboptimal. Salindia "Penentuan Rute (*Route*/*Path Planning*) Bagian 1" memberikan contoh dan menjelaskan bahwa *Greedy* BFS tidak lengkap dan bisa terjebak di minimum lokal, yang keduanya berimplikasi pada ketidakoptimalan algoritma ini.

Secara keseluruhan, untuk penyelesaian *game Rush Hour*, algoritma seperti *Uniform Cost Search* (UCS) menjamin ditemukannya solusi optimal dalam hal jumlah langkah karena sifat biaya langkah yang seragam, meskipun kurang efisien dalam menjelajahi ruang pencarian yang besar. *Iterative Deepening Search* (IDS) juga menjamin solusi optimal dan menawarkan efisiensi memori yang lebih baik dibandingkan UCS, namun dengan mengorbankan waktu karena adanya eksplorasi ulang *node*. Algoritma *Greedy Best First Search*, meskipun berpotensi lebih cepat karena dipandu heuristik, tidak memberikan jaminan optimalitas dan bisa terjebak pada solusi suboptimal. Di sisi lain, A*, dengan heuristik yang *admissible*, menawarkan keseimbangan terbaik dengan menjamin solusi optimal sekaligus meningkatkan efisiensi pencarian secara signifikan dibandingkan UCS maupun IDS dengan cara memfokuskan eksplorasi pada jalur-jalur yang paling menjanjikan menuju tujuan.

3.2 Algoritma UCS

3.2.1 *Source Code* Algoritma UCS

Fungsi	Deskripsi

`#include "ucs.hpp"`
`#include "../../utils/utils.hpp"`
 	
`Solution ucs::search_ucs(const Board& initial_board, const std::vector<Piece>& initial_pieces) {`
`    const int COST_PER_MOVE = 1;`
 	
`    Utils::SearchParams ucs_params;`
`    ucs_params.algorithm_name = "UCS";`
 	
`ucs_params.calculate_initial_val = [](const Board&, const vector<Piece>&) {`
`        return 0;`
`    };`
 	
`    ucs_params.successor_val = [COST_PER_MOVE](const Board&, const vector<Piece>&, const SearchNode& parent_node) {`
`        return parent_node.val + COST_PER_MOVE;`
`    };`
 	
`    ucs_params.get_node_exploration = [](const SearchNode& node) {`
`        return godot::String("Cost: ") + godot::String::num_int64(node.val);`
`    };`
 	
`    ucs_params.get_solution_details`
	Fungsi ini bertujuan untuk menemukan solusi permainan *Rush Hour* menggunakan algoritma *Uniform Cost Search* (UCS). Fungsi ini menerima konfigurasi papan awal (initial_board) dan daftar bidak awal (initial_pieces) sebagai input. Di dalamnya, ia menginisialisasi struktur Utils::SearchParams yang dikhususkan untuk UCS. Fungsi menghitung nilai awal (calculate_initial_val) diatur untuk mengembalikan 0, karena biaya untuk mencapai *state* awal dari dirinya sendiri adalah nol. Kunci dari UCS terletak pada bagaimana ia menghitung biaya untuk mencapai suksesor (successor_val). Dalam implementasi ini, biaya suksesor dihitung sebagai biaya *node* induk (parent_node.val) ditambah COST_PER_MOVE (yang bernilai 1), sehingga SearchNode.val dapat mengakumulasi total biaya atau jumlah langkah dari *state* awal.

= [](const SearchNode& node, const Solution&) {
return godot::String("Optimal steps: ") + godot::String::num_int64(node.val);
};
return Utils::search(initial_board, initial_pieces, ucs_params);
}	

3.2.2 Langkah Penyelesaian Algoritma UCS

1. Inisialisasi:
    - Sebuah struktur data *priority queue* disiapkan untuk menyimpan *node-node* yang akan dieksplorasi. *Node-node* diurutkan berdasarkan biaya kumulatif terendah (g(n)) dari *node* awal.
    - Sebuah struktur data disiapkan untuk menyimpan *state-state* yang sudah pernah dikunjungi beserta biaya terendah untuk mencapainya. Ini bertujuan untuk menghindari pemrosesan ulang *state* yang sama dan mencegah siklus tak terbatas.
    - Buat SearchNode awal (initial_node) berdasarkan initial_board dan initial_pieces yang diberikan.
    - Biaya (val atau g(n)) dari initial_node diatur ke 0, karena belum ada langkah yang diambil.
    - Masukkan initial_node ke dalam antrian prioritas.
2. Loop Utama:

    Algoritma kemudian masuk ke dalam loop yang berlanjut selama antrian prioritas tidak kosong:

    - Ambil (dan keluarkan) *node* (current_node) dari antrian prioritas yang memiliki biaya g(n) paling rendah. Ini adalah node yang akan dieksplorasi selanjutnya.

- Jumlah *node* yang diekspansi (result.node) di-*increment*.
- Konversikan konfigurasi *piece* dari current_node menjadi representasi string yang unik (current_state_str) menggunakan Utils::state_to_string. Periksa apakah current_state_str sudah ada di dalam set *visited*. Jika sudah ada, berarti *state* ini telah dieksplorasi sebelumnya. Dalam kasus ini, current_node diabaikan, dan *loop* melanjutkan ke iterasi berikutnya untuk mengambil *node* lain dari antrian prioritas. Jika current_state_str belum ada di *visited*, tambahkan current_state_str ke *visited* untuk menandai bahwa state ini sekarang sedang dieksplorasi.
- Periksa apakah current_node merupakan state tujuan menggunakan fungsi Utils::is_exit(current_node.board, current_node.pieces). Jika *true* (state tujuan tercapai), proses pencarian berhasil.
- Jika current_node bukan *state* tujuan, bangkitkan semua kemungkinan *state* suksesor yang valid dari current_node menggunakan fungsi Utils::generate_next.

3. Terminasi:

    - Jika antrian prioritas menjadi kosong dan *state* tujuan belum ditemukan, berarti tidak ada jalur solusi yang mungkin dari *state* awal ke *state* tujuan. Dalam kasus ini, catat durasi, jumlah *node* yang diekspansi, dan kembalikan objek Solution dengan is_solved = false.

3.3 Algoritma IDS

3.3.1 *Source Code* Algoritma IDS

Fungsi	Deskripsi

`#include "ids.hpp"`
 	
`// helper`
`SearchNode dls(SearchNode initial_node, const Board& initial_board, int limit, int&`
	Fungsi helper dls adalah versi iteratif yang menggunakan *stack* manual untuk menghindari potensi *stack overflow* yang bisa terjadi pada implementasi rekursif.

nodes_total, set<string>& visited) {
    stack<SearchNode> s;
    s.push(initial_node);
 	
    while (!s.empty()) {
        SearchNode current_node = s.top();
        s.pop();
 	
        nodes_total++;
 	
        if (current_node.piece_moved != ' ') {
            Coordinates new_position = {-1,-1};
            Coordinates original_position = current_node.original_position;
 	
            for(const auto& p_state : current_node.pieces) {
                if (p_state.id == current_node.piece_moved) {
                    new_position = p_state.coordinates;
                    break;
                }
            }
        }
 	
        if (Utils::is_exit(initial_board, current_node.pieces)) {
            return current_node;
        }
 	
        if (current_node.val >= limit) {
            continue;
        }
 	
        vector<SearchNode> next = Utils::generate_next(current_node, [](const Board&, const
	Fungsi ini menerima *node* awal untuk DLS (initial_node), papan, batas kedalaman (limit), referensi ke total *node* yang diekspansi (nodes_total), dan referensi ke set *visited* untuk melacak *state* yang sudah dimasukkan ke *stack* atau diproses dalam panggilan DLS saat ini. DLS bekerja dengan mengambil *node* dari *stack*, mengecek apakah itu solusi atau apakah sudah mencapai batas kedalaman. Jika belum dan bukan solusi, ia akan membangkitkan suksesor, dan suksesor yang belum ada di *visited* akan dimasukkan ke *stack* untuk eksplorasi lebih lanjut.
	
	Fungsi ids::search_ids bertujuan untuk menemukan solusi permainan *Rush Hour* menggunakan algoritma *Iterative Deepening Search* (IDS). Fungsi ini menerima konfigurasi papan awal (initial_board) dan daftar bidak awal (initial_pieces). Mekanisme utama IDS adalah melakukan serangkaian pencarian DLS dengan batas kedalaman (depth_limit) yang terus meningkat, dimulai dari 0 hingga MAX_DEPTH_LIMIT. Untuk setiap depth_limit, fungsi search_ids menginisialisasi SearchNode awal dengan kedalaman 0 dan sebuah *set* visited_dls untuk melacak *state* yang dikunjungi khusus

vector<Piece>&, const SearchNode& parent_node) {
            return parent_node.val + 1;
        });
 	
        reverse(next.begin(), next.end());
 	
        for (const auto& next_node : next) {
            string next_state = Utils::state_to_string(next_node.pieces);
 	
            if (visited.find(next_state) == visited.end()) {
 	
visited.insert(next_state);
                s.push(next_node);
            }
        }
    }
    return SearchNode({}, initial_board, {}, -1);
}
 	
Solution ids::search_ids(const Board& initial_board, const vector<Piece>& initial_pieces) {
    auto time_start = chrono::high_resolution_clock::now();
    Solution result;
    result.node = 0;
 	
    for (int depth_limit = 0; depth_limit <= MAX_DEPTH_LIMIT; ++depth_limit) {
        SearchNode initial_node(initial_pieces, initial_board, {}, 0);
        set<string> visited_dls;
	untuk iterasi DLS tersebut guna mencegah siklus dalam satu panggilan DLS. Kemudian, ia memanggil fungsi helper dls. Jika dls berhasil menemukan solusi, search_ids akan mencatat informasi solusi seperti *path*, durasi, dan jumlah total *node* yang diekspansi, lalu mengembalikan solusi tersebut. Jika DLS pada depth_limit saat ini tidak menemukan solusi, loop akan berlanjut ke depth_limit berikutnya. Jika semua iterasi hingga MAX_DEPTH_LIMIT selesai tanpa solusi, fungsi akan mengembalikan hasil yang menandakan solusi tidak ditemukan.

```
visited_dls.insert(Utils::state_to_st
ring(initial_node.pieces));

        SearchNode found =
dls(initial_node, initial_board,
depth_limit, result.node,
visited_dls);

        if (found.val != -1) {
            result.is_solved = true;
            result.moves =
found.path;
            auto time_end =
chrono::high_resolution_clock::now();
            result.duration =
chrono::duration<double,
milli>(time_end - time_start);

            return result;
        }
    }
    auto time_end =
chrono::high_resolution_clock::now();
    result.duration =
chrono::duration<double,
milli>(time_end - time_start);
    result.is_solved = false;

    return result;
}
```

3.3.2 Langkah Penyelesaian Algoritma IDS

1. Inisialisasi:
    - Catat waktu mulai untuk menghitung durasi total pencarian.
    - Inisialisasi objek Solution (result), dengan result.node (total node yang diekspansi di semua iterasi) diatur ke 0 dan result.is_solved diatur ke false.
    - Tentukan MAX_DEPTH_LIMIT, yaitu batas kedalaman maksimum yang akan dicoba oleh IDS.
2. Loop Utama:

Algoritma masuk ke dalam *loop* yang mengiterasi depth_limit dari 0 hingga MAX_DEPTH_LIMIT:

- Buat SearchNode awal (initial_node) berdasarkan initial_board dan initial_pieces. Kedalaman (val) dari initial_node diatur ke 0.
- Buat sebuah *set string* yang akan digunakan khusus untuk iterasi DLS saat ini. Set ini bertujuan untuk mencegah pemrosesan ulang *state* yang sama dan siklus dalam satu panggilan DLS pada depth_limit tertentu. *Set* ini di-*reset* di setiap awal iterasi depth_limit.
- Masukkan representasi *string* dari initial_node.pieces ke dalam visited_dls.
- Panggil fungsi dls dengan parameter: initial_node, initial_board, depth_limit saat ini, referensi ke result.node (untuk akumulasi *node* yang diekspansi), dan visited_dls.

3. Evaluasi DLS

   - Setelah didapat kembali fungsi dls, periksa nilai val dari SearchNode yang dikembalikan. Jika found.val != -1, *set* result.is_solved = true, salin found.path ke result.moves, catat waktu selesai dan hitung durasi, cetak pesan sukses beserta detail solusi (jumlah langkah, node yang diekspansi, durasi), dan kembalikan result. Jika found.val == -1, DLS pada depth_limit tersebut tidak menemukan solusi.

4. Terminasi:

   - Jika *loop* depth_limit selesai (mencapai MAX_DEPTH_LIMIT) dan solusi belum juga ditemukan, catat waktu selesai dan hitung durasi. result.is_solved akan tetap false. Cetak pesan bahwa tidak ada solusi yang ditemukan dalam MAX_DEPTH_LIMIT.

3.4 Algoritma *Greedy* BFS

3.4.1 *Source Code* Algoritma *Greedy* BFS

Fungsi	Deskripsi

`#include "bfs.hpp"`
`#include "../../utils/utils.hpp"`
 	
`Solution bfs::search_bfs(const Board& initial_board, const std::vector<Piece>& initial_pieces) {`
`    Utils::SearchParams bfs_params;`
`    bfs_params.algorithm_name = "BFS";`
 	
`    bfs_params.calculate_initial_val = [](const Board& board, const vector<Piece>& pieces) {`
`        return Utils::calculate(board, pieces);`
`    };`
 	
`    bfs_params.successor_val = [](const Board& board_state, const vector<Piece>& next_pieces_state, const SearchNode&) {`
`        return Utils::calculate(board_state, next_pieces_state);`
`    };`
 	
`    bfs_params.get_node_exploration = [](const SearchNode& node) {`
`        return godot::String("Heuristic: ") + godot::String::num_int64(node.val);`
`    };`
 	
`    bfs_params.get_solution_details = [](const SearchNode&, const Solution& sol) {`
`        return godot::String("Moves: ") +`
	Fungsi bfs::search_bfs mengimplementasikan algoritma *Greedy Best First Search*. Fungsi ini menerima konfigurasi papan awal (initial_board) dan daftar bidak awal (initial_pieces). Di dalamnya, ia menyiapkan struktur Utils::SearchParams dengan nama algoritma "BFS". Fungsi untuk menghitung nilai awal (calculate_initial_val) dan nilai suksesor (successor_val) keduanya diatur untuk menggunakan Utils::calculate(board, pieces). Fungsi Utils::calculate ini adalah fungsi heuristik yang mengestimasi jarak atau biaya dari *state* saat ini ke tujuan (misalnya, jarak mobil 'P' ke pintu keluar). Karena *Greedy Best First Search* memilih *node* untuk dieksplorasi hanya berdasarkan nilai heuristik (h(n)), penggunaan Utils::calculate untuk kedua parameter ini konsisten dengan perilaku *Greedy* BFS, dengan SearchNode.val akan menyimpan nilai heuristik tersebut.

godot::String::num_int64(sol.moves.size());
    };
 	
    return Utils::search(initial_board, initial_pieces, bfs_params);
}	

3.4.2 Langkah Penyelesaian Algoritma *Greedy* BFS

1. Inisialisasi:
    - Sebuah struktur data *priority queue* disiapkan untuk menyimpan *node-node* yang akan dieksplorasi. *Node-node* diurutkan berdasarkan nilai heuristik terendah (h(n)).
    - Sebuah struktur data disiapkan untuk menyimpan *state-state* yang sudah pernah dikunjungi untuk menghindari pemrosesan ulang *state* yang sama dan mencegah siklus tak terbatas.
    - Buat SearchNode awal (initial_node) berdasarkan initial_board dan initial_pieces yang diberikan.
    - Hitung nilai heuristik (h(n)) untuk initial_node menggunakan fungsi Utils::calculate(initial_board, initial_pieces). Nilai ini disimpan dalam initial_node.val.
    - Masukkan initial_node ke dalam antrian prioritas.
2. Loop Utama

    Algoritma kemudian masuk ke dalam loop yang berlanjut selama antrian prioritas tidak kosong:

    - Ambil (dan keluarkan) *node* (current_node) dari antrian prioritas yang memiliki nilai heuristik h(n) paling rendah. Ini adalah *node* yang dianggap paling dekat dengan tujuan dan akan dieksplorasi selanjutnya.
    - Jumlah *node* yang diekspansi (result.node) di-*increment*.

- Konversikan konfigurasi *piece* dari current_node menjadi representasi string yang unik (current_state_str) menggunakan Utils::state_to_string. Periksa apakah current_state_str sudah ada di dalam *set visited*. Jika sudah ada, current_node diabaikan, dan *loop* melanjutkan ke iterasi berikutnya. Jika current_state_str belum ada di *visited*, tambahkan current_state_str ke *visited*.
- Periksa apakah current_node merupakan *state* tujuan menggunakan fungsi Utils::is_exit(current_node.board, current_node.pieces). Jika true, proses pencarian berhasil. Path solusi adalah current_node.path. Jika current_node bukan *state* tujuan, bangkitkan semua kemungkinan *state* suksesor yang valid dari current_node menggunakan fungsi Utils::generate_next.
- Untuk setiap successor_node yang dihasilkan, hitung nilai heuristik (h(successor_node)) untuk successor_node menggunakan fungsi Utils::calculate(successor_node.board, successor_node.pieces). Nilai ini disimpan dalam successor_node.val. *Path* untuk successor_node diperbarui dengan menambahkan gerakan dari current_node ke successor_node ke current_node.path. Periksa apakah next_state_str ada di visited. Jika belum ada di *visited*, maka successor_node dimasukkan ke dalam antrian prioritas.

3. Terminasi

    - Jika antrian prioritas menjadi kosong dan *state* tujuan belum juga ditemukan, berarti tidak ada jalur solusi yang dapat ditemukan oleh algoritma. Catat durasi, jumlah *node* yang diekspansi, dan kembalikan objek Solution dengan is_solved = false.

3.5 Algoritma A*

3.5.1 *Source Code* Algoritma A*

Fungsi	Deskripsi

`#include "astar.hpp"`
 	
`vector<AStarSearchNode>`
	Fungsi search_astar mengimplementasikan algoritma A*. Algoritma ini memulai dengan memasukkan konfigurasi ke dalam

```
astar::generate_successors_astar(cons
t AStarSearchNode&
current_astar_node, const int
COST_PER_MOVE) {
    vector<AStarSearchNode>
successors;
    const vector<Piece>&
current_pieces =
current_astar_node.pieces;
    const Board& board_node =
current_astar_node.board;

    for (size_t piece_idx = 0;
piece_idx < current_pieces.size();
++piece_idx) {
        const Piece& piece_to_move =
current_pieces[piece_idx];
        if (piece_to_move.id == 'K')
continue;

        for (int direction = -1;
direction <= 1; direction += 2) {
            for (int steps = 1; ;
++steps) {
                vector<Piece>
next_pieces_state = current_pieces;
                Piece&
piece_moved_in_state =
next_pieces_state[piece_idx];
                bool move_possible =
true;
                Coordinates
original_coords_of_moving_piece =
piece_to_move.coordinates;

                if
(piece_to_move.is_vertical) {
                    for (int s = 1; s
<= steps; ++s) {
                        int y_check;
                        if (direction
> 0) { y_check =
piece_to_move.coordinates.y +
piece_to_move.size - 1 + s;}
```

priority_queue. Setiap simpul diberi prioritas berdasarkan f-cost, yang merupakan penjumlahan g-cost dan h-cost. Algoritma ini mengambil secara berulang simpul dengan f-cost terendah dari priority_queue. Jika simpul dikunjungi, akan dicatat ke visited_states untuk menghindari siklus. Algoritma juga memeriksa apakah state saat ini adalah goal state atau tidak menggunakan Utils::is_exit. Apabila simpul saat ini bukan merupakan state tujuan, fungsi generate_successors_astar akan dipanggil. Fungsi ini bertugas untuk menghasilkan successor_nodes yang valid dari simpul saat ini. Jika sebuah gerakan valid, sebuah simpul successor akan dibuat. Simpul ini mewarisi induknya dan memperbarui g-cost dengan menambahkan COST_PER_MOVE dan menghitung h-cost menggunakan Utils::calculate. Simpul successor akan dimasukkan ke priority_queue.

```
                        else {
y_check = piece_to_move.coordinates.y
- s; }
                        if
(!Utils::is_cell_clear(board_node,
y_check, piece_to_move.coordinates.x,
current_pieces, piece_to_move.id)) {

move_possible = false; break;
                        }
                    }
                    if
(move_possible) {
piece_moved_in_state.coordinates.y +=
(direction * steps); }
                } else {
                    for (int s = 1; s
<= steps; ++s) {
                        int x_check;
                        if (direction
> 0) { x_check =
piece_to_move.coordinates.x +
piece_to_move.size - 1 + s; }
                        else {
x_check = piece_to_move.coordinates.x
- s; }
                        if
(!Utils::is_cell_clear(board_node,
piece_to_move.coordinates.y, x_check,
current_pieces, piece_to_move.id)) {

move_possible = false; break;
                        }
                    }
                    if
(move_possible) {
piece_moved_in_state.coordinates.x +=
(direction * steps); }
                }

                if (!move_possible) {
                    break;
                }
```

```
                vector<PieceMove>
next_path = current_astar_node.path;
                PieceMove current_pm;

current_pm.old_coordinates =
original_coords_of_moving_piece;

current_pm.new_coordinates =
piece_moved_in_state.coordinates;

next_path.push_back(current_pm);

                int g_cost_successor
= current_astar_node.actual_g_cost +
COST_PER_MOVE;
                int h_cost_successor
= Utils::calculate(board_node,
next_pieces_state);


successors.emplace_back(next_pieces_s
tate, board_node, next_path,
g_cost_successor, h_cost_successor,
piece_to_move.id,
original_coords_of_moving_piece);
            }
        }
    }
    return successors;
}

Solution astar::search_astar(const
Board& initial_board, const
vector<Piece>& initial_pieces) {
    auto time_start =
chrono::high_resolution_clock::now();
    Solution result;
    result.is_solved = false;
    result.node = 0;

    const int COST_PER_MOVE = 1;

    priority_queue<AStarSearchNode,
vector<AStarSearchNode>,
```

```
greater<AStarSearchNode>> pq;
    set<string> visited_states;

    int initial_g_cost = 0;
    int initial_h_cost =
Utils::calculate(initial_board,
initial_pieces);
    AStarSearchNode
initial_astar_node(initial_pieces,
initial_board, {}, initial_g_cost,
initial_h_cost);
    pq.push(initial_astar_node);

    while (!pq.empty()) {
        AStarSearchNode current_node
= pq.top();
        pq.pop();
        result.node++;

        string current_state_str =
current_node.get_state_string();
        if
(visited_states.count(current_state_s
tr)) {
            continue;
        }

visited_states.insert(current_state_s
tr);

        if (current_node.piece_moved
!= ' ') { // piece_moved dari base
            Coordinates
new_pos_for_log = {-1, -1};
            for(const auto& p_state :
current_node.pieces) { // pieces dari
base
                if (p_state.id ==
current_node.piece_moved) {
                    new_pos_for_log =
p_state.coordinates;
                    break;
                }
            }
```

```
        }

        if
(Utils::is_exit(current_node.board,
current_node.pieces)) { // board dan
pieces dari base
            result.is_solved = true;
            result.moves =
current_node.path; // path dari base
            break;
        }

        vector<AStarSearchNode>
successors =
generate_successors_astar(current_nod
e, COST_PER_MOVE);

        for (const auto&
successor_node : successors) {
            string next_state_str =
successor_node.get_state_string();
            if
(!visited_states.count(next_state_str
)) {

pq.push(successor_node);
            }
        }
    }

    auto time_end =
chrono::high_resolution_clock::now();
    result.duration =
chrono::duration<double,
milli>(time_end - time_start);

    if (!result.is_solved) {

godot::UtilityFunctions::print("ASTAR
: No solution found. Nodes visited: "
+
godot::String::num_int64(static_cast<
int64_t>(result.node)));
    }
```

`    return result;` `}`	

3.5.2 Langkah Penyelesaian Algoritma A*

1. Inisialisasi:
    - Sebuah struktur data *priority queue* disiapkan untuk menyimpan *node-node* yang akan dieksplorasi. *Node-node* diurutkan berdasarkan nilai f-cost terendah.
    - Sebuah struktur data disiapkan untuk menyimpan *state-state* yang sudah pernah dikunjungi untuk menghindari pemrosesan ulang *state* yang sama dan mencegah siklus tak terbatas.
    - Buat SearchNode awal (initial_node) berdasarkan initial_board dan initial_pieces yang diberikan.
    - Nilai heuristik awal dihitung menggunakan Utils::calculate(initial_board, initial_pieces). G-cost awal diatur ke 0.
    - Masukkan initial_node ke dalam antrian prioritas.
2. Loop Utama

    Algoritma kemudian masuk ke dalam loop yang berlanjut selama antrian prioritas tidak kosong:

    - Ambil (dan keluarkan) *node* (current_node) dari antrian prioritas yang memiliki nilai f-cost paling rendah. Ini adalah *node* yang dianggap paling dekat dengan tujuan dan akan dieksplorasi selanjutnya.
    - Jumlah *node* yang diekspansi (result.node) di-*increment*.
    - Konversikan konfigurasi *piece* dari current_node menjadi representasi string yang unik (current_state_str) menggunakan Utils::state_to_string. Periksa apakah current_state_str sudah ada di dalam *set visited*. Jika sudah ada, current_node diabaikan, dan *loop* melanjutkan ke iterasi berikutnya. Jika current_state_str belum ada di *visited*, tambahkan current_state_str ke *visited*.

- Periksa apakah current_node merupakan *state* tujuan menggunakan fungsi Utils::is_exit(current_node.board, current_node.pieces). Jika true, proses pencarian berhasil. Path solusi adalah current_node.path. Jika current_node bukan *state* tujuan, bangkitkan semua kemungkinan *state* suksesor yang valid dari current_node menggunakan fungsi Utils::generate_next.
- Untuk setiap successor_node yang dihasilkan, hitung f-cost untuk successor_node menggunakan fungsi Utils::calculate(successor_node.board, successor_node.pieces). Nilai ini disimpan dalam successor_node.val. *Path* untuk successor_node diperbarui dengan menambahkan gerakan dari current_node ke successor_node ke current_node.path. Periksa apakah next_state_str ada di visited. Jika belum ada di *visited*, maka successor_node dimasukkan ke dalam antrian prioritas.

3. Terminasi

   - Jika antrian prioritas menjadi kosong dan *state* tujuan belum juga ditemukan, berarti tidak ada jalur solusi yang dapat ditemukan oleh algoritma. Catat durasi, jumlah *node* yang diekspansi, dan kembalikan objek Solution dengan is_solved = false.

3.6 Fungsi *Helper*

Fungsi	Deskripsi

`void Utils::print_board(Board& board, std::vector<Piece>& pieces) {`
 	
`UtilityFunctions::print("Board:");`
`vector<vector<char>> board_representation(board.rows + 2 * board.piece_padding, vector<char>(board.cols + 2 * board.piece_padding, ' '));`
 	
`board_representation[board.exit_coordinates.y][board.exit_coordinates.x] = 'K';`
	Fungsi Utils::print_board dibuat untuk mencetak representasi visual dari papan permainan ke konsol atau *output* standar.

```
    for (Piece& piece : pieces) {
        for (int i = 0; i <
piece.size; i++) {
            int x =
piece.coordinates.x;
            int y =
piece.coordinates.y;
            for (int j = 0; j <
piece.size; j++) {
                if
(piece.is_vertical) {

board_representation[y + j][x] =
piece.id;
                } else {

board_representation[y][x + j] =
piece.id;
                }
            }
        }
    }
    for (int i = 0; i <
board_representation.size(); i++) {
        string line = "";
        for (int j = 0; j <
board_representation[i].size(); j++)
{
            if
(board_representation[i][j] == ' ' &&
i >= board.piece_padding && i <
board_representation.size() -
board.piece_padding && j >=
board.piece_padding && j <
board_representation[i].size() -
board.piece_padding) {
                line += '.';
            } else {
                line +=
board_representation[i][j];
            }
        }

UtilityFunctions::print(stringToGodot
```

String(line)); } }	

godot::String
Utils::stringToGodotString(const
std::string& stdString) {
    return
godot::String(stdString.c_str());
}
	Fungsi Utils::stringToGodotString adalah sebuah utilitas konversi sederhana yang mengambil std::string standar C++ sebagai input dan mengembalikannya sebagai objek godot::String.

std::string
Utils::godotStringToString(const
godot::String& godotString) {
    return
std::string(godotString.utf8().get_da
ta());
}
	Fungsi Utils::godotStringToString melakukan konversi dari godot::String ke std::string standar C++. Fungsi ini menerima objek godot::String dan mengembalikan representasi std::string.

int Utils::calculate(const Board&
initial_board, const vector<Piece>&
current_pieces) {
    const Piece* primary_piece =
get_primary_piece(current_pieces);
 	
    if (!primary_piece) {
 	
godot::UtilityFunctions::printerr("GB
FS Error: Primary piece not found in
heuristic calculation.");
        return -1; // kalau ga ada
primary piece
    }
 	
    int distance = 0;
    if (!primary_piece->is_vertical)
{ // bidak horizontal
        // jika pintu keluar ada di
kanan bidak utama
        if
(initial_board.exit_coordinates.x >
primary_piece->coordinates.x) {
            distance =
	Fungsi Utils::calculate berperan sebagai fungsi heuristik yang mengestimasi “jarak” atau “biaya” dari konfigurasi bidak saat ini (current_pieces) ke *state* tujuan (pintu keluar) pada papan (initial_board). Pertama, fungsi ini mencoba mendapatkan bidak utama menggunakan get_primary_piece. Jika bidak utama tidak ditemukan, fungsi akan mencetak pesan *error* dan mengembalikan -1. Jika ditemukan, ia menghitung jarak berdasarkan orientasi bidak utama dan posisinya relatif terhadap pintu keluar. Untuk bidak horizontal, jarak dihitung sebagai selisih koordinat x antara tepi bidak utama yang relevan (kanan atau kiri) dan koordinat x pintu keluar. Untuk bidak vertikal, perhitungan serupa dilakukan

initial_board.exit_coordinates.x - (primary_piece->coordinates.x + primary_piece->size - 1);
        }
        else { // jika pintu keluar ada di kiri bidak utama
            distance = primary_piece->coordinates.x - initial_board.exit_coordinates.x;
        }
    }
    else { // bidak vertikal
        // jika pintu keluar ada di bawah bidak utama
        if (initial_board.exit_coordinates.y > primary_piece->coordinates.y) {
            distance = initial_board.exit_coordinates.y - (primary_piece->coordinates.y + primary_piece->size - 1);
        }
        else { // jika pintu keluar ada di atas bidak utama
            distance = primary_piece->coordinates.y - initial_board.exit_coordinates.y;
        }
    }
    return abs(distance);
}
	menggunakan koordinat y. Nilai absolut dari selisih ini kemudian dikembalikan sebagai nilai heuristik.

bool Utils::is_exit(const Board& initial_board, const vector<Piece>& current_pieces) {
    const Piece* primary_piece = get_primary_piece(current_pieces);
    if (!primary_piece) {
 	
godot::UtilityFunctions::printerr("GBFS Error: Primary piece not found in goal state check.");
        return false;
    }
	Fungsi Utils::is_exit bertugas untuk menentukan apakah konfigurasi bidak saat ini (current_pieces) sudah mencapai *state* tujuan, yaitu apakah bidak utama telah berhasil keluar dari papan (initial_board).

```
    if (!primary_piece->is_vertical)
{ // horizontal
        if
(primary_piece->coordinates.y !=
initial_board.exit_coordinates.y)
return false;

        // jika pintu keluar di kanan
        if
(initial_board.exit_coordinates.x >=
(primary_piece->coordinates.x +
primary_piece->size -1) ) {
            return
(primary_piece->coordinates.x +
primary_piece->size - 1) ==
initial_board.exit_coordinates.x;
        }
        else { // pintu keluar di
kiri
            return
primary_piece->coordinates.x ==
initial_board.exit_coordinates.x;
        }
    }
    else { // vertikal
        if
(primary_piece->coordinates.x !=
initial_board.exit_coordinates.x)
return false;

        // jika pintu keluar di bawah
        if
(initial_board.exit_coordinates.y >=
(primary_piece->coordinates.y +
primary_piece->size -1)) {
            return
(primary_piece->coordinates.y +
primary_piece->size - 1) ==
initial_board.exit_coordinates.y;
        }
        else { // pintu keluar di
atas
            return
primary_piece->coordinates.y ==
```

initial_board.exit_coordinates.y;
        }
    }
    return false;
}	

const Piece*
Utils::get_primary_piece(const vector<Piece>& pieces_list) {
    for (const auto& p : pieces_list) {
        if (p.is_primary) {
            return &p;
        }
    }
    return nullptr;
}
	Fungsi ini bertujuan untuk menemukan dan mengembalikan *pointer* ke bidak yang ditandai sebagai bidak utama (is_primary == true). Jika bidak utama ditemukan dalam daftar, *pointer* ke objek Piece tersebut akan dikembalikan. Jika tidak ada bidak dalam daftar yang ditandai sebagai utama, fungsi ini akan mengembalikan *nullptr*.

string Utils::state_to_string(const vector<Piece>& current_pieces) {
    string s = "";
    vector<Piece> sorted_pieces = current_pieces;
 	
    // urutin berdasarkan id
    sort(sorted_pieces.begin(), sorted_pieces.end(), [](const Piece& a, const Piece& b) {
        return a.id < b.id;
    });
 	
    // buat string gabungan per piece yang berisi info penting terkait koordinat dan arah
    for (const auto& piece : sorted_pieces) {
        if (piece.id == 'K') continue;
        s += piece.id;
        s += ':';
        s += to_string(piece.coordinates.x);
        s += ',';
        s +=
	Fungsi Utils::state_to_string berfungsi untuk menghasilkan representasi string yang unik untuk konfigurasi bidak saat ini (current_pieces).

to_string(piece.coordinates.y);
        s += (piece.is_vertical ? "V" : "H");
        s += ";";
    }
    return s;
}	

bool Utils::is_cell_clear(const Board& initial_board, int r, int c, const vector<Piece>& pieces_state, char moving_piece_id) {
    if (r < initial_board.piece_padding || r >= initial_board.rows + initial_board.piece_padding || c < initial_board.piece_padding || c >= initial_board.cols + initial_board.piece_padding) {
        const Piece* p_moving = nullptr;
        for(const auto& p_iter : pieces_state) {
            if(p_iter.id == moving_piece_id) {
                p_moving = &p_iter;
                break;
            }
        }
        if (p_moving && p_moving->is_primary &&
            r == initial_board.exit_coordinates.y &&
            c == initial_board.exit_coordinates.x) {
            return true;
        }
        return false;
    }
 	
    for (const auto& piece : pieces_state) {
        if (piece.id == moving_piece_id || piece.id == 'K') {
// jangan cek exit
	Fungsi Utils::is_cell_clear bertugas untuk memeriksa apakah sebuah sel tertentu (r, c) pada papan (initial_board) kosong dan dapat ditempati oleh bidak yang sedang bergerak (moving_piece_id), dengan mempertimbangkan konfigurasi bidak saat ini (pieces_state).

<pre>continue;
}
if (piece.is_vertical) {
if (c ==
piece.coordinates.x && r >=
piece.coordinates.y && r <
piece.coordinates.y + piece.size) {
return false;
}
}
else {
if (r ==
piece.coordinates.y && c >=
piece.coordinates.x && c <
piece.coordinates.x + piece.size) {
return false; //
ditempati piece lain
}
}
}
return true;
}</pre>	

<pre>vector<SearchNode>
Utils::generate_next(const
SearchNode& current_node,
ValueCalculator calculate_node_value)
{
vector<SearchNode> successors;
const vector<Piece>&
current_pieces = current_node.pieces;
const Board& board_node =
current_node.board;
 	
for (size_t piece_idx = 0;
piece_idx < current_pieces.size();
++piece_idx) {
const Piece& piece_to_move =
current_pieces[piece_idx];
if (piece_to_move.id == 'K')
continue; // jangan gerakin exit
 	
// direction: -1 = kiri/atas,
1 = kanan/bawah
for (int direction = -1;</pre>
	Fungsi Utils::generate_next bertujuan untuk membangkitkan semua kemungkinan *state* suksesor (langkah berikutnya yang valid) dari SearchNode saat ini (current_node). Ia menerima current_node dan sebuah ValueCalculator (fungsi lambda yang akan menghitung nilai, misalnya biaya atau heuristik, untuk *node* suksesor).

```
direction <= 1; direction += 2) {
            if (direction == 0)
continue;
            for (int steps = 1; ;
++steps) {
                vector<Piece>
next_pieces_state = current_pieces;
                Piece& piece_moved =
next_pieces_state[piece_idx];
                bool move = true;

                if
(piece_to_move.is_vertical) {
                    for (int s = 1; s
<= steps; ++s) {
                        int y;
                        if (direction
> 0) { // bergerak ke bawah
                            y =
piece_to_move.coordinates.y +
piece_to_move.size - 1 + s;
                        }
                        else { //
bergerak ke atas
                            y =
piece_to_move.coordinates.y - s;
                        }
                        if
(!is_cell_clear(board_node, y,
piece_to_move.coordinates.x,
current_pieces, piece_to_move.id)) {
                            move =
false;
                            break;
                        }
                    }
                    if (move) {

piece_moved.coordinates.y +=
(direction * steps);
                    }
                }
                else { // bergerak
horizontal
```

```
                    for (int s = 1; s
<= steps; ++s) {
                        int x;
                        if (direction
> 0) { // bergerak ke kanan
                            x =
piece_to_move.coordinates.x +
piece_to_move.size - 1 + s;
                        }
                        else { //
bergerak ke kiri
                            x =
piece_to_move.coordinates.x - s;
                        }
                        if
(!is_cell_clear(board_node,
piece_to_move.coordinates.y, x,
current_pieces, piece_to_move.id)) {
                            move =
false;
                            break;
                        }
                    }
                    if (move) {

piece_moved.coordinates.x +=
(direction * steps);
                    }
                }

                if (!move) {
                    break;
                }

                vector<PieceMove>
next_path = current_node.path;
                PieceMove current_pm;

current_pm.old_coordinates =
piece_to_move.coordinates;

current_pm.new_coordinates =
piece_moved.coordinates;
```

next_path.push_back(current_pm);
 	
int val = calculate_node_value(board_node, next_pieces_state, current_node);
 	
successors.emplace_back(next_pieces_state, board_node, next_path, val, piece_to_move.id, piece_to_move.coordinates);
}
}
}
return successors;
}	

Solution Utils::search(const Board& initial_board, const vector<Piece>& initial_pieces, const SearchParams& params) {
auto time_start = chrono::high_resolution_clock::now();
Solution result;
result.is_solved = false;
result.node = 0;
 	
priority_queue<SearchNode, vector<SearchNode>, greater<SearchNode>> pq;
set<string> visited;
 	
int initial_val = params.calculate_initial_val(initial_board, initial_pieces);
SearchNode initial_node(initial_pieces, initial_board, {}, initial_val);
pq.push(initial_node);
 	
while (!pq.empty()) {
SearchNode current_node = pq.top();
pq.pop();
result.node++;
	Fungsi Utils::search adalah kerangka kerja pencarian generik yang digunakan oleh algoritma seperti UCS dan *Greedy* BFS. Fungsi ini menerima konfigurasi papan dan bidak awal, serta objek SearchParams yang mendefinisikan perilaku spesifik algoritma pencarian. Loop utama kemudian berjalan selama pq tidak kosong dan diperiksa apakah itu *state* tujuan menggunakan Utils::is_exit. Jika ya, solusi ditemukan dan dikembalikan. Jika tidak, fungsi ini membangkitkan semua suksesor menggunakan Utils::generate_next dan memasukkan suksesor yang belum dikunjungi ke pq. Jika pq kosong dan solusi belum ditemukan, dikembalikan hasil yang menandakan tidak ada solusi. Durasi pencarian juga diukur dan dicatat.

```
        string current_state_str =
state_to_string(current_node.pieces);
        if
(visited.count(current_state_str)) {
            continue;
        }

visited.insert(current_state_str);

        if
(is_exit(current_node.board,
current_node.pieces)) {
            result.is_solved = true;
            result.moves =
current_node.path;
            break;
        }

        vector<SearchNode> successors
= generate_next(
            current_node,
            params.successor_val
        );

        for (const auto&
successor_node : successors) {
            string next_state_str =
state_to_string(successor_node.pieces
);
            if
(!visited.count(next_state_str)) {

pq.push(successor_node);
            }
        }
    }

    auto time_end =
chrono::high_resolution_clock::now();
    result.duration =
chrono::duration<double,
milli>(time_end - time_start);

    if (!result.is_solved) {
```

godot::UtilityFunctions::print(godot::String::utf8(params.algorithm_name.c_str()) + ": No solution found. Nodes visited: " + godot::String::num_int64(static_cast<int64_t>(result.node)));
    }
    return result;
}	

BAB 4

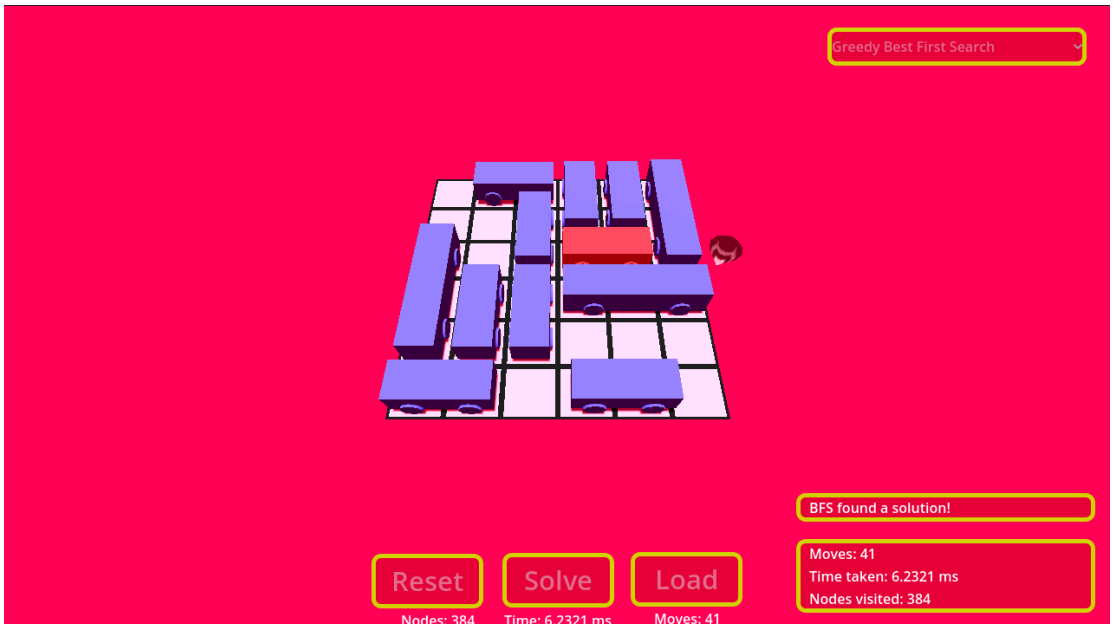
PENGUJIAN PROGRAM

4.1 Pengujian Program

4.1.1. Test Case 1 (Normal)

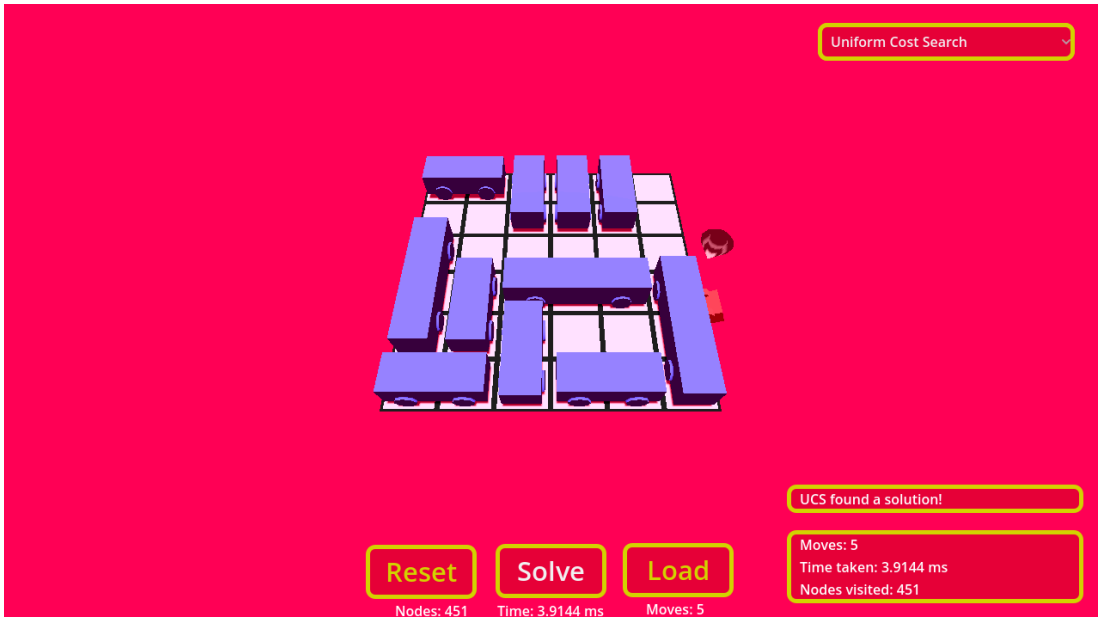
```
6 6
11
AAB..F
..BCDF
GPPCDFK
GH.III
GHJ...
LLJMM.
```

a. Greedy Best First Search



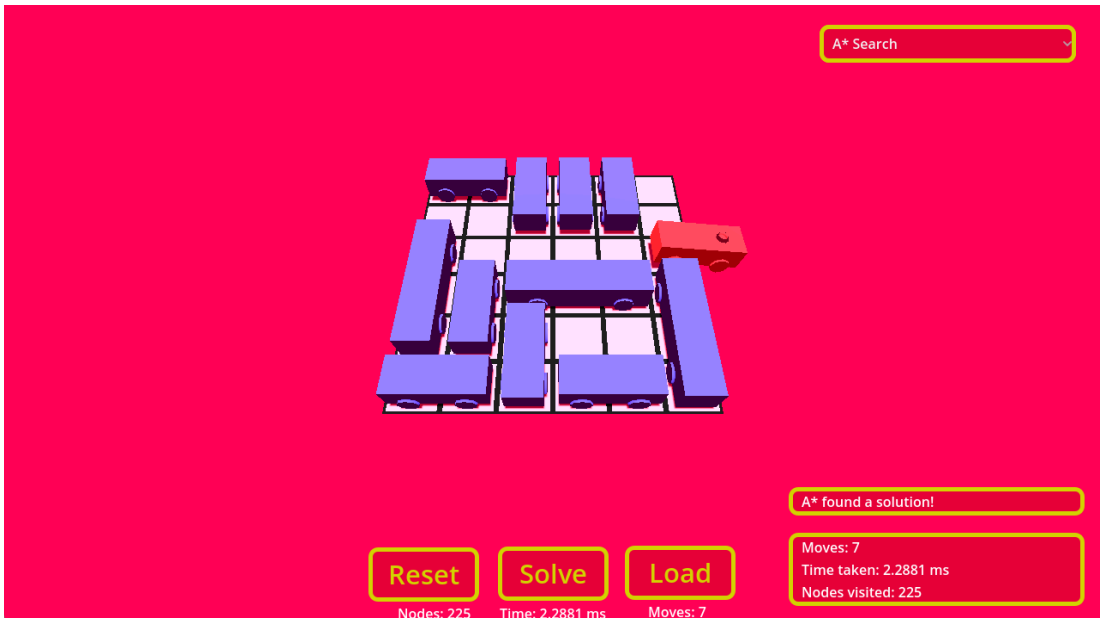
Gambar 2. Pengujian Program GBFS

b. Uniform Cost Search



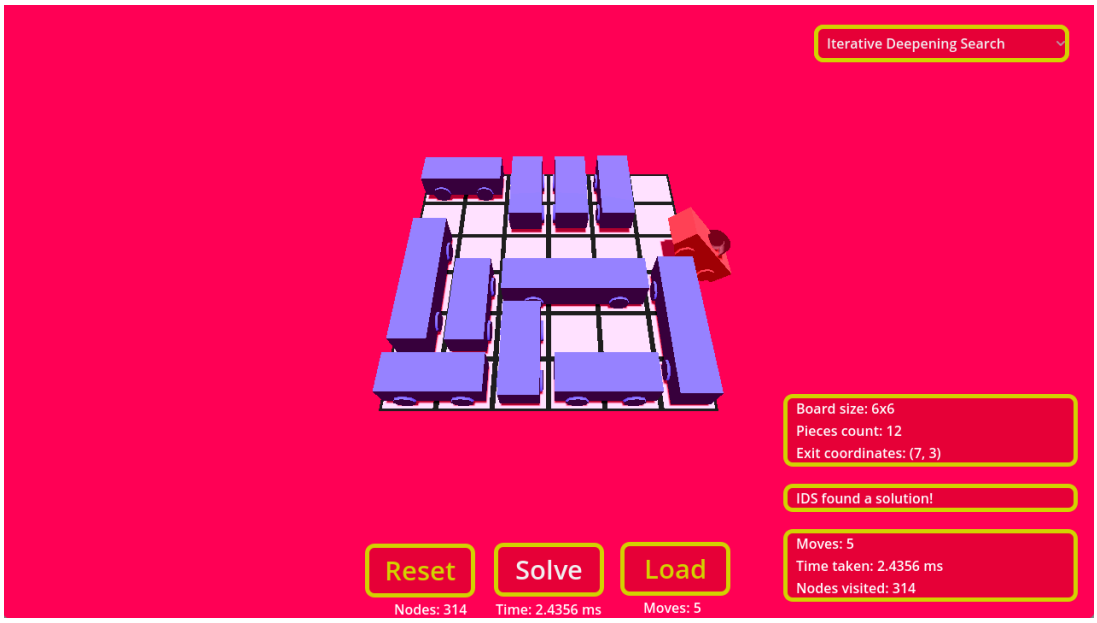
Gambar 3. Pengujian Program UCS

c. A*



Gambar 4. Pengujian Program A*

d. Iterative Deepening Search

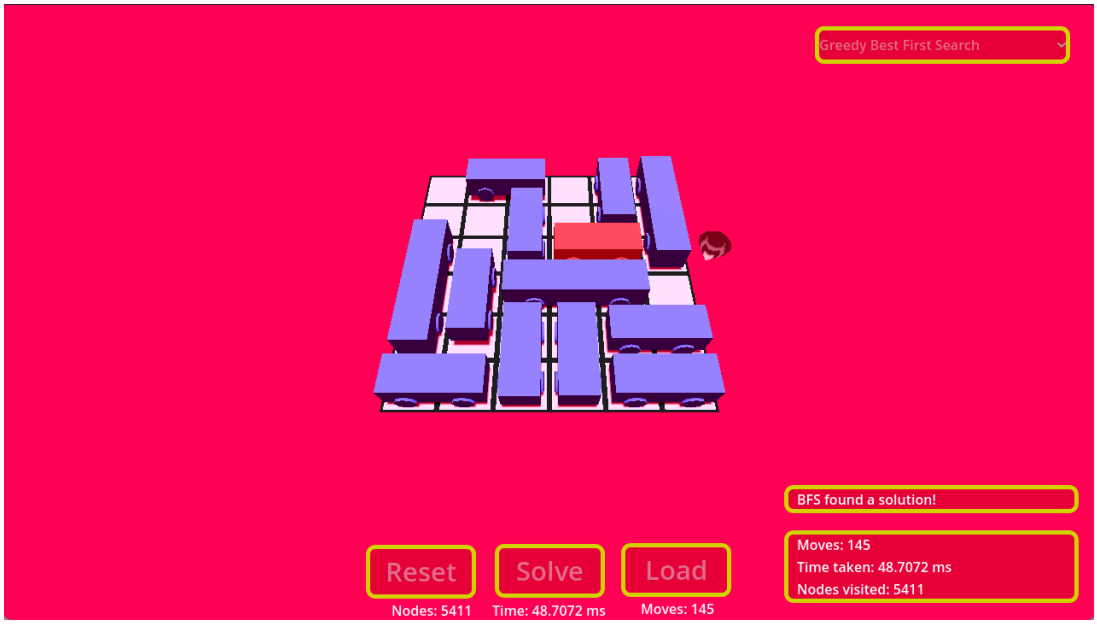


Gambar 5. Pengujian Program IDS

4.1.2. Test Case 2 (Hardest 6x6)

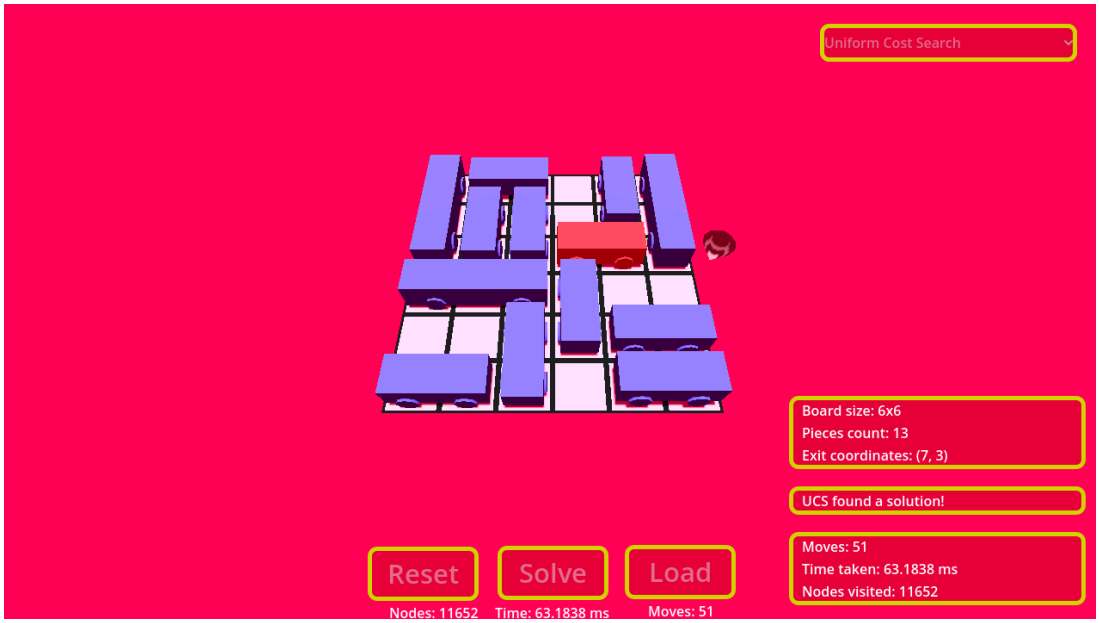
```
6 6
12
AEE.H.
AFG.HI
AFGPPIK
BBBJ.I
..DJLL
CCDMM.
```

a. Greedy Best First Search



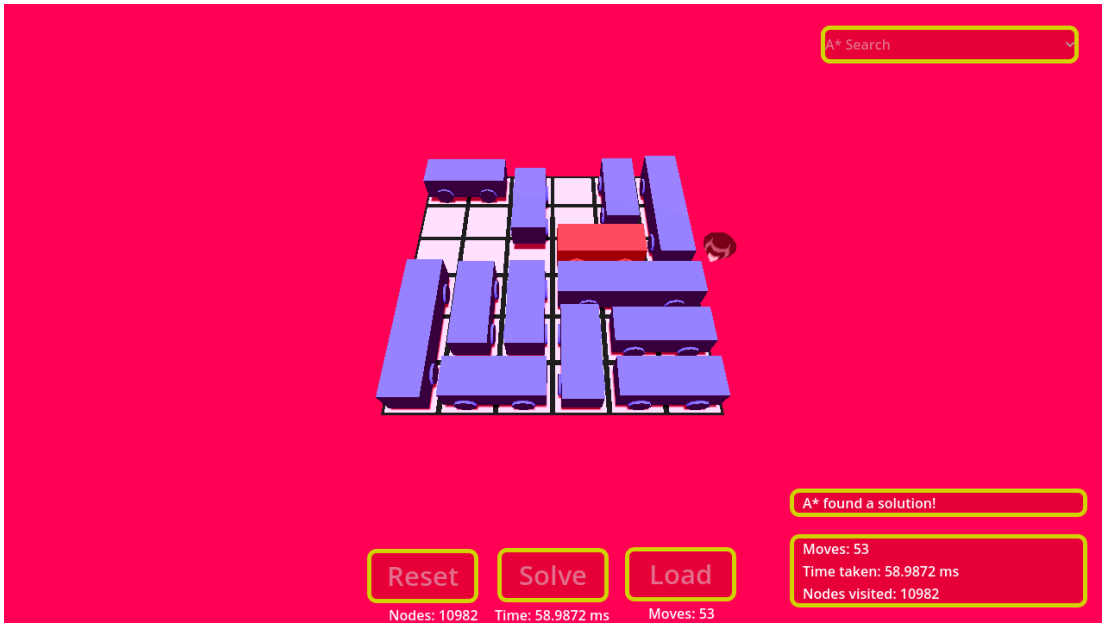
Gambar 6. Pengujian Program GBFS

b. Uniform Cost Search



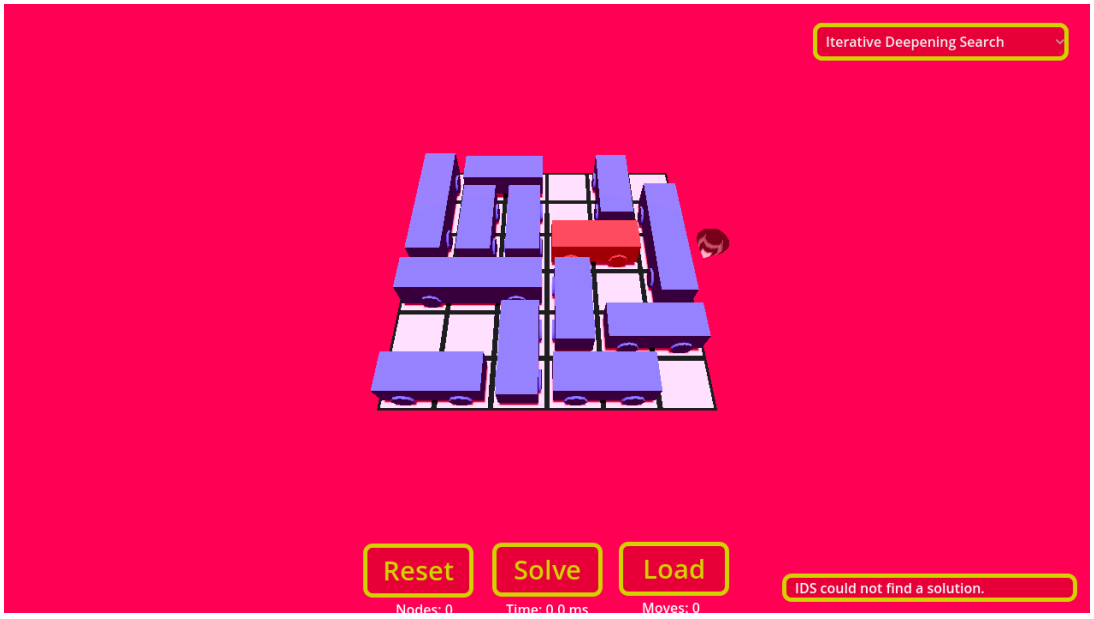
Gambar 7. Pengujian Program UCS

c. A*



Gambar 8. Pengujian Program A*

d. Iterative Deepening Search

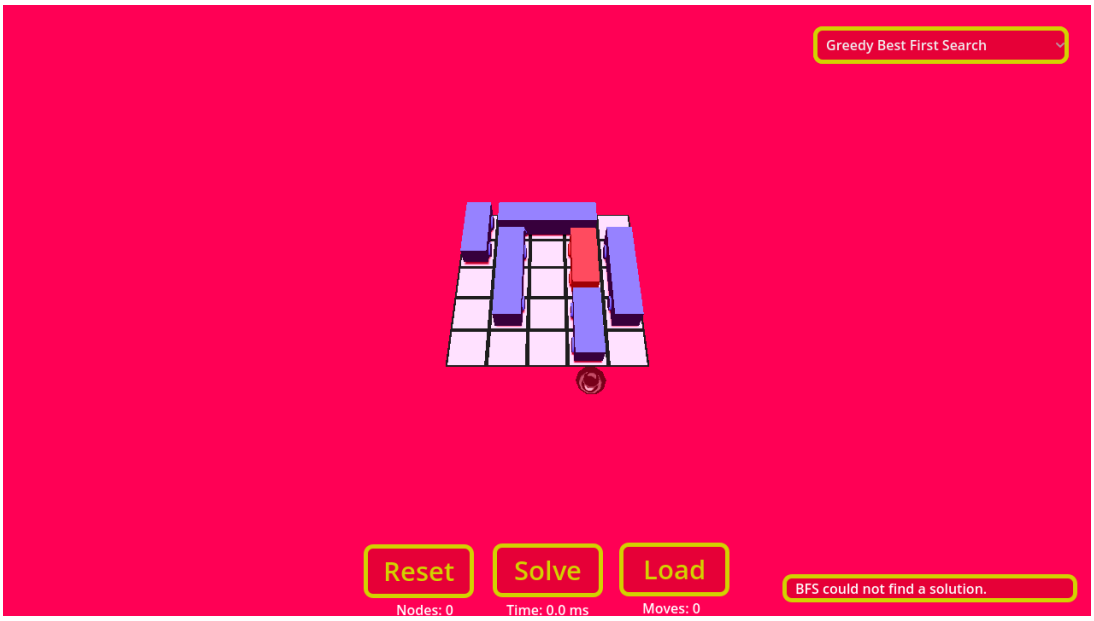


Gambar 9. Pengujian Program IDS

4.1.3. Test Case 3 (Tidak ada solusi)

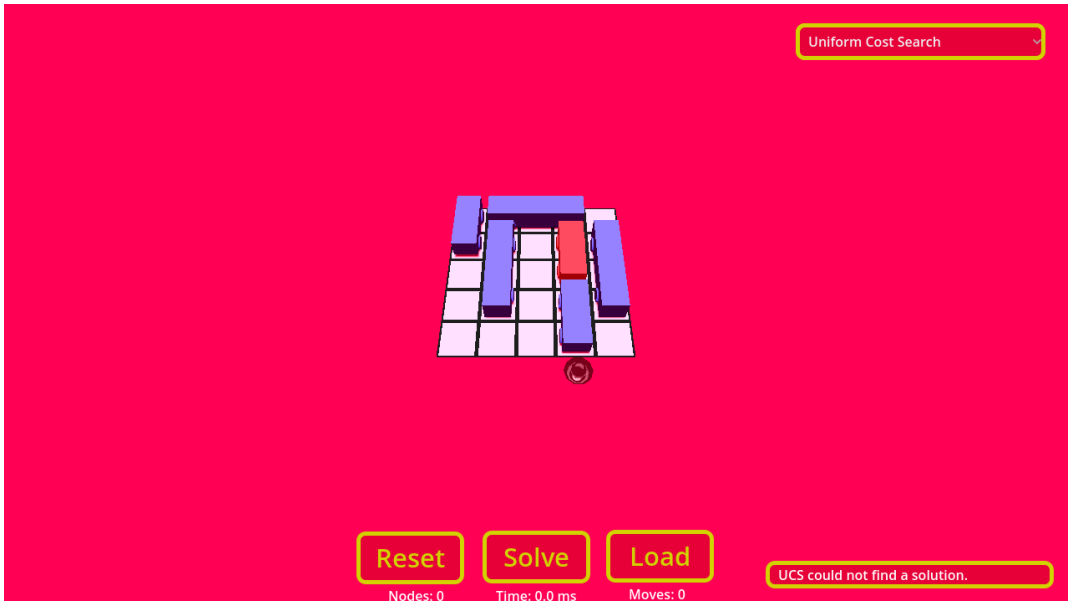
```
5 5
5
EAAA.
ED.PB
.D.PB
.D.CB
...C.
   K
```

a. Greedy Best First Search



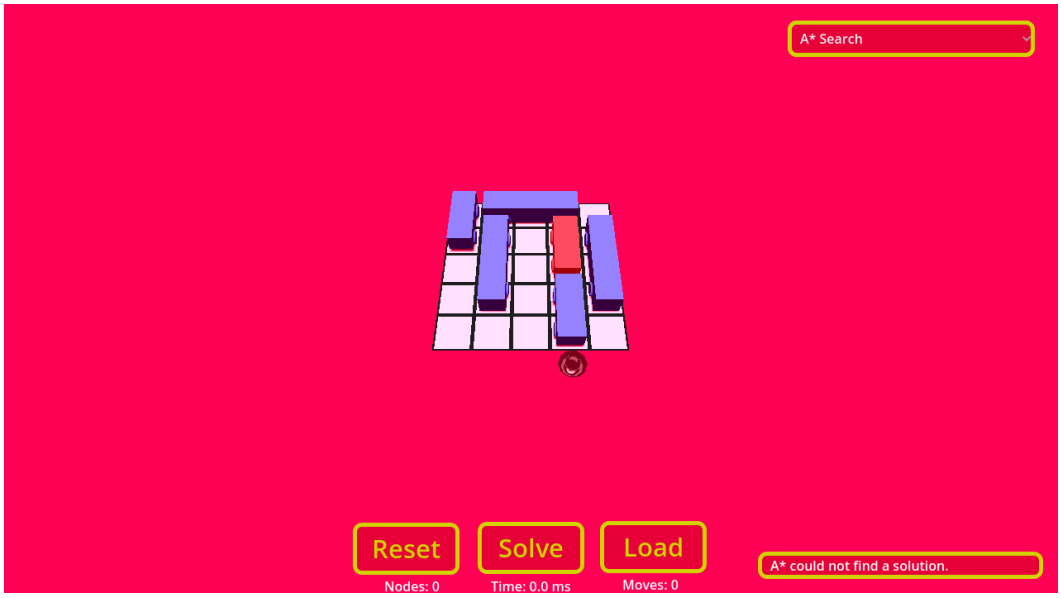
Gambar 10. Pengujian Program GBFS

b. Uniform Cost Search



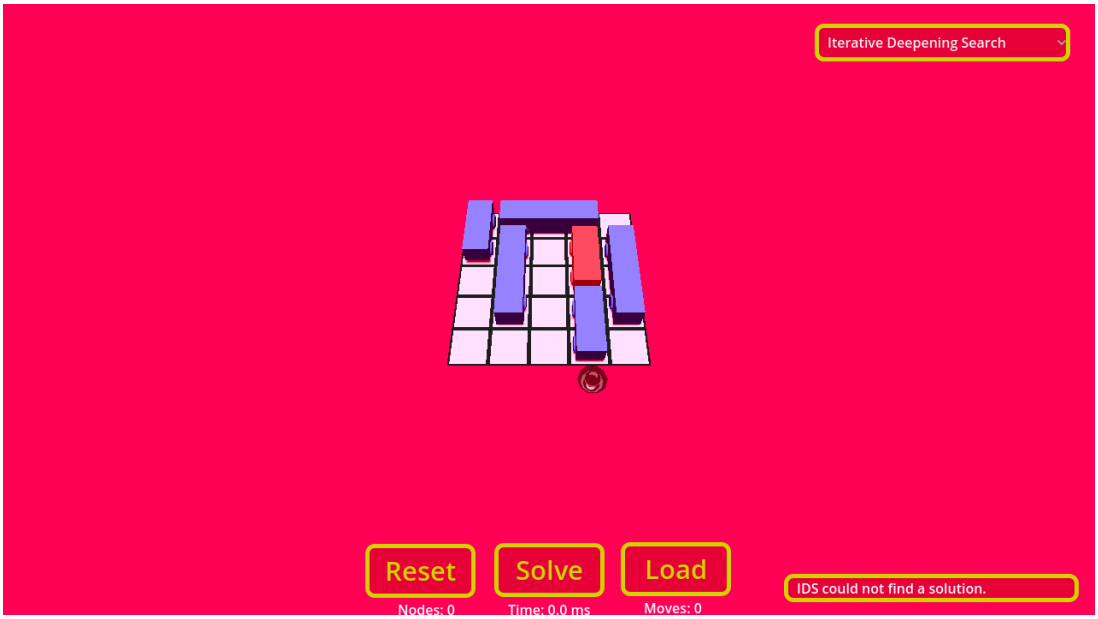
Gambar 11. Pengujian Program UCS

c. A*



Gambar 12. Pengujian Program A*

d. Iterative Deepening Search

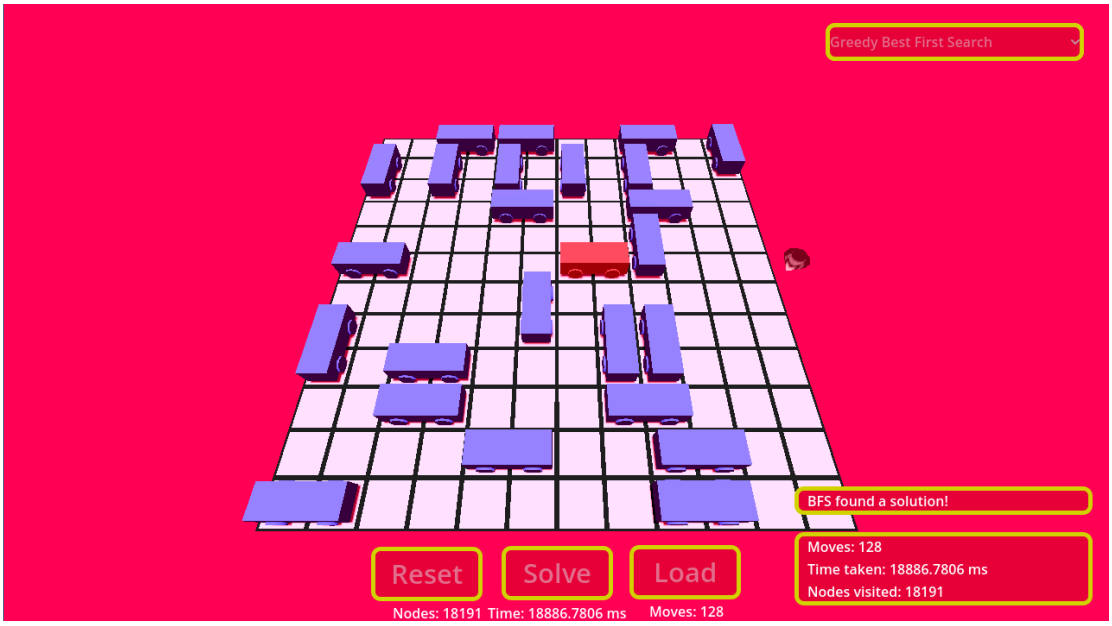


Gambar 13. Pengujian Program IDS

4.1.4 Test Case 4 (25 Piece)

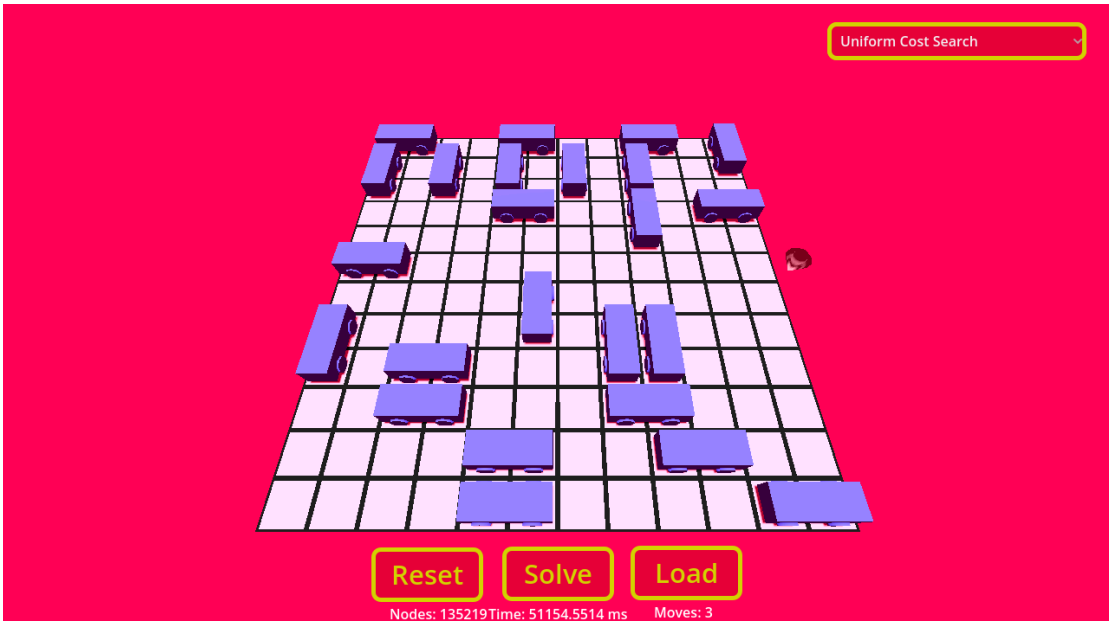
```
12 12
24
AA..BB..EE.D
F.G.H.I.J..D
F.G.H.I.J...
....LL..MM..
........O...
NN..PP..O...K
.....X......
C....X.RQ...
C.SS...RQ...
..TT...UU...
....WW..YY..
....VV....ZZ
```

a. Greedy Best First Search



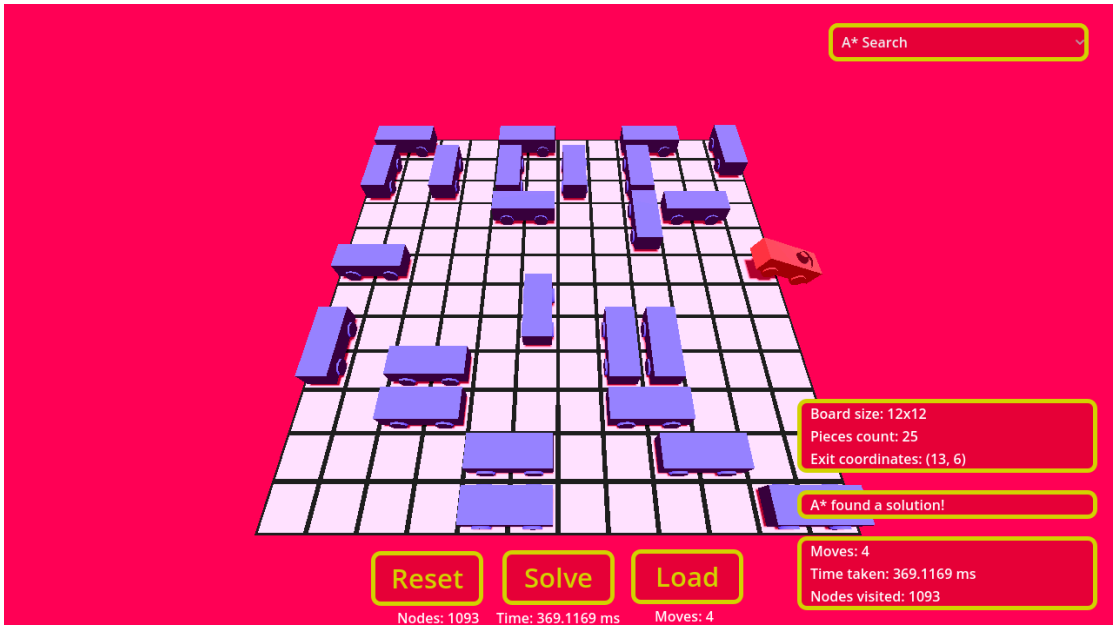
Gambar 14. Pengujian Program GBFS

b. Uniform Cost Search



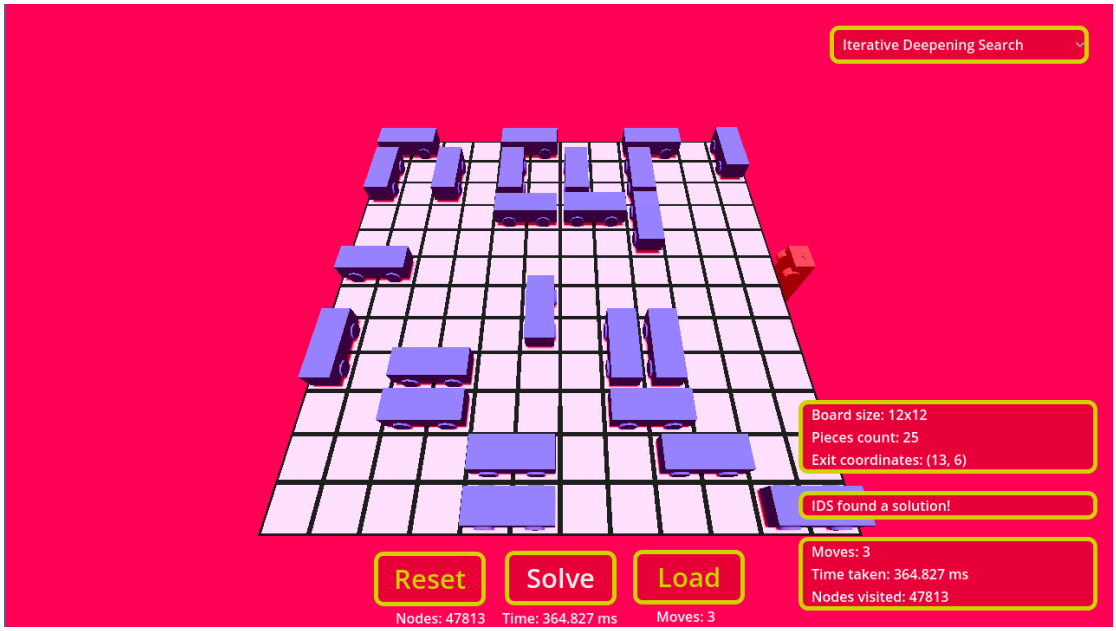
Gambar 15. Pengujian Program UCS

c. A*



Gambar 16. Pengujian Program A*

d. Iterative Deepening Search



Gambar 17. Pengujian Program IDS

4.2 Analisis Hasil Pengujian

Analisis Performa Algoritma Berdasarkan Kasus Uji:

4.2.1. Optimalitas Solusi (Jumlah Langkah)

a. Pada Test Case 1 (Normal), UCS dan IDS berhasil menemukan solusi dengan jumlah langkah paling sedikit, yaitu 5 langkah. Ini sesuai dengan teori bahwa UCS (yang berperilaku seperti BFS ketika biaya langkah seragam, yaitu 1) dan IDS menjamin solusi optimal dalam hal jumlah langkah. Algoritma A* menemukan solusi dalam 7 langkah, sedangkan GBFS menghasilkan solusi yang jauh dari optimal dengan 41 langkah. Meskipun A* dengan heuristik yang *admissible* diharapkan optimal, hasil ini menunjukkan bahwa heuristik yang digunakan belum menghasilkan jalur terpendek absolut dalam kasus ini. GBFS, yang hanya mempertimbangkan nilai heuristik (h(n)) tanpa biaya aktual (g(n)), terbukti tidak optimal, sesuai dengan sifat “rakus”-nya.

b. Pada Test Case 2 (Hardest 6x6), UCS menemukan solusi dengan 51 langkah, sementara A* membutuhkan 53 langkah. GBFS kembali menghasilkan solusi dengan langkah terbanyak, yaitu 145 langkah. IDS tidak berhasil menemukan solusi untuk kasus uji ini, yang akan dibahas lebih lanjut. Perbandingan antara UCS dan A* kembali menunjukkan UCS lebih unggul dalam menemukan jalur yang lebih pendek pada kasus ini.

c. Untuk Test Case 3 (Tidak ada solusi), tidak ada yang dapat menemukan solusi.

d. Pada Test Case 4 (23 Piece), UCS dan IDS menemukan solusi optimal dengan 3 langkah. A* menemukan solusi yang sangat baik dengan 4 langkah. GBFS menghasilkan solusi yang sangat tidak optimal dengan 128 langkah. Ini menunjukkan bahwa untuk kasus ini, UCS dan IDS berhasil mencapai optimalitas, A* sangat mendekati, sementara GBFS jauh dari optimal.

4.2.2. Efisiensi (Waktu Eksekusi dan Node yang Dikunjungi)

a. **Test Case 1 (Normal)**

- A* menunjukkan efisiensi terbaik dengan waktu eksekusi tercepat (2.2881 ms) dan jumlah *node* dikunjungi paling sedikit (225). Ini sejalan dengan teori bahwa A* dengan heuristik yang baik dapat secara signifikan mengurangi ruang pencarian.
- IDS menjadi yang tercepat kedua (2.4356 ms) dengan 314 *node*.
- UCS membutuhkan waktu lebih lama (3.9144 ms) dan mengunjungi lebih banyak *node* (451) dibandingkan A* dan IDS, karena UCS menjelajahi semua kemungkinan jalur secara merata berdasarkan biaya kumulatif.
- GBFS, meskipun menghasilkan solusi non-optimal, memiliki waktu eksekusi 6.2321 ms dan mengunjungi 384 *node*.

b. **Test Case 2 (Hardest 6x6)**

- GBFS secara mengejutkan menjadi yang tercepat (48.7072 ms), meskipun dengan jumlah langkah yang sangat tidak optimal dan jumlah *node* yang dikunjungi cukup banyak (5411). Kecepatan ini mungkin karena sifatnya yang "rakus" yang dengan cepat menuju ke suatu solusi, meskipun solusi tersebut buruk.
- A* membutuhkan waktu 58.9872 ms dan mengunjungi 10982 *node*.
- UCS menjadi yang paling lambat di antara yang berhasil menemukan solusi, dengan waktu 63.1838 ms dan mengunjungi 11652 *node*.
- Jumlah *node* yang dikunjungi oleh A* dan UCS meningkat drastis pada kasus yang lebih sulit ini, namun A* tetap lebih efisien dalam hal jumlah *node* yang dieksplorasi dibandingkan UCS.
- IDS gagal menemukan solusi, yang mengindikasikan bahwa untuk kasus yang lebih kompleks atau memerlukan kedalaman pencarian besar, IDS bisa

menjadi tidak efisien karena eksplorasi ulang *node* pada setiap iterasi kedalaman atau mencapai batas kedalaman maksimum yang ditetapkan (MAX_DEPTH_LIMIT).

c. **Test Case 4 (23 Piece)**

   - Waktu Eksekusi: IDS adalah yang tercepat (364.827 ms), diikuti sangat dekat oleh A* (369.1169 ms). GBFS jauh lebih lambat (18886.7806 ms), dan UCS adalah yang paling lambat secara signifikan (51154.5514 ms).

   - Node Dikunjungi: A* mengunjungi jumlah *node* paling sedikit (1093). GBFS mengunjungi 18191 *node*. IDS mengunjungi 47813 *node*, dan UCS mengunjungi jumlah *node* terbanyak (135219).

   - Pada kasus yang kompleks dengan solusi yang ternyata dangkal (3-4 langkah), A* menunjukkan efisiensi luar biasa dalam hal *node* yang dikunjungi dan waktu. IDS juga sangat cepat karena menemukan solusi pada kedalaman yang relatif kecil. UCS, meskipun menemukan solusi optimal, menjadi sangat tidak efisien karena harus menjelajahi sejumlah besar *node* pada level yang sama sebelum menemukan solusi dangkal tersebut, menggambarkan kompleksitas $O(b^d)$-nya dalam skenario pencarian melebar.

4.2.3. Kompleksitas Algoritma

a. Secara teoritis, UCS dan IDS memiliki kompleksitas waktu $O(b^d)$ (dengan b adalah faktor percabangan dan d adalah kedalaman solusi). Peningkatan jumlah *node* yang dikunjungi pada Test Case 2 dan terutama Test Case 4 untuk UCS mencerminkan sifat eksponensial ini, terutama jika solusi berada pada kedalaman yang dangkal namun faktor percabangan besar. Kegagalan IDS pada Test Case 2 juga dapat disebabkan oleh batasan praktis dari pencarian mendalam berulang pada ruang status yang besar dan dalam.

b. A* memiliki potensi efisiensi yang jauh lebih baik jika heuristiknya berkualitas tinggi. Hasilnya menunjukkan A* umumnya menjelajahi lebih sedikit *node* daripada UCS, terutama pada kasus yang lebih besar dan pada Test Case 4.

c. GBFS, yang hanya berfokus pada heuristik h(n), tidak memiliki jaminan optimalitas atau kelengkapan. Meskipun cepat dalam beberapa skenario (seperti Test Case 2, meski tidak optimal), ketidakoptimalannya sangat jelas pada Test Case 1 dan Test Case 4.

4.2.4. Dampak Heuristik

a. Heuristik yang digunakan ("jarak Manhattan mobil utama ('P') ke pintu keluar") bersifat *admissible*. Pada Test Case 4, A* sangat mendekati optimal (4 langkah vs 3 langkah) dengan efisiensi yang sangat baik.

b. Untuk GBFS, terdapat heuristik, tetapi karena mengabaikan biaya g(n) (jumlah langkah yang sudah ditempuh), ia menghasilkan solusi yang panjang atau sangat panjang.

4.2.5. Kelengkapan dan Penanganan Kasus Sulit/Tanpa Solusi

a. Kegagalan IDS pada Test Case 2 menunjukkan bahwa meskipun secara teoritis lengkap, implementasi praktisnya dapat dibatasi oleh sumber daya (waktu atau batas kedalaman). Namun, pada Test Case 4, IDS berhasil menemukan solusi optimal dengan cepat.

b. Untuk Test Case 3 (Tidak ada solusi), semua algoritma yang lengkap (UCS, A* dengan *visited set*, IDS dalam batasnya) akan melaporkan bahwa tidak ada solusi ditemukan, sesuai dengan mekanisme terminasi mereka.

BAB 5
KESIMPULAN

UCS dan IDS (ketika berhasil) konsisten memberikan solusi optimal dalam hal jumlah langkah untuk permainan *Rush Hour* ini, dengan biaya setiap langkah adalah seragam. Ini terkonfirmasi kuat di Test Case 1 dan Test Case 4. A* menawarkan keseimbangan yang sangat baik antara efisiensi (waktu dan *node* yang dieksplorasi) dan optimalitas. Pada Test Case 4, A* adalah yang paling efisien dalam hal *node* yang dikunjungi dan sangat cepat, dengan solusi yang hanya satu langkah lebih banyak dari solusi optimal. Heuristik yang digunakan berperan penting dalam kinerjanya. GBFS, meskipun terkadang cepat (seperti pada Test Case 2), secara konsisten sangat tidak dapat diandalkan untuk menemukan solusi yang optimal, seperti yang terlihat jelas pada Test Case 1 dan terutama Test Case 4. IDS dapat mengalami kesulitan pada kasus uji yang sangat kompleks atau memerlukan kedalaman pencarian yang besar (seperti kegagalannya di Test Case 2), tetapi sangat efektif dan cepat jika solusi optimal berada pada kedalaman yang relatif dangkal (seperti pada Test Case 4). Pada Test Case 4, terlihat bagaimana UCS bisa menjadi sangat tidak efisien pada masalah dengan ruang pencarian besar jika solusi optimalnya dangkal, karena ia akan tetap menjelajahi banyak *node* secara melebar. Analisis ini menunjukkan bahwa pilihan algoritma sangat tergantung pada prioritas:

- Jika optimalitas adalah kunci dan solusi mungkin tidak terlalu dalam, IDS bisa menjadi pilihan yang sangat baik karena juga efisien secara memori dan cepat dalam skenario tersebut. UCS juga menjamin optimalitas tetapi bisa sangat lambat untuk masalah kompleks.
- Jika keseimbangan antara efisiensi tinggi dan solusi mendekati optimal (atau optimal) adalah yang dicari, A* adalah kandidat yang kuat, terutama apabila disertai dengan heuristik yang baik.
- GBFS mungkin berguna hanya jika kecepatan menemukan solusi lebih penting daripada kualitas solusi tersebut, namun hasil pengujian menunjukkan bahkan kecepatannya tidak selalu unggul jika dibandingkan dengan A* atau IDS yang menemukan solusi lebih baik.

LAMPIRAN

Tautan Repository Github

https://github.com/aibrahim185/Tucil3_13523087_13523089

Tautan Executable

https://github.com/aibrahim185/Tucil3_13523087_13523089

Hasil Akhir Tugas Kecil 3

Poin	Ya	Tidak
1. Program berhasil dikompilasi tanpa kesalahan	✔	
2. Program berhasil dijalankan	✔	
3. Solusi yang diberikan program benar dan mematuhi aturan permainan	✔	
4. Program dapat membaca masukan berkas .txt dan menyimpan solusi berupa print board tahap per tahap dalam berkas .txt	✔	
5. [Bonus] Implementasi algoritma pathfinding alternatif	✔	
6. [Bonus] Implementasi 2 atau lebih heuristik alternatif		✔
7. [Bonus] Program memiliki GUI	✔	
8. Program dan laporan dibuat (kelompok) sendiri	✔	

DAFTAR PUSTAKA

Godot Engine. (n.d.). *Godot Engine documentation*. Diakses pada 19 Mei 2025, https://docs.godotengine.org/en/stable/

Maulidevi, N. U. (2025). *Penentuan Rute (Route/Path Planning) Bagian 1: BFS, DFS, UCS, Greedy Best First Search* [Bahan Kuliah IF2211 Strategi Algoritma]. Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf

Maulidevi, N. U., & Munir, R. (2025). *Penentuan Rute (Route/Path Planning) Bagian 2: Algoritma A** [Bahan Kuliah IF2211 Strategi Algoritma]. Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf

Munir, R. (2025). *BFS-DFS (Bagian 2)* [Bahan Kuliah IF2211 Strategi Algoritma]. Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/14-BFS-DFS-(2025)-Bagian2.pdf