

LAPORAN TUGAS KECIL 3 IF2211 STRATEGI ALGORITMA

PENYELESAIAN PUZZLE RUSH HOUR MENGGUNAKAN

ALGORITMA PATHFINDING



DISUSUN OLEH:

Asybel Bintang Pardomuan Sianipar 15223011

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

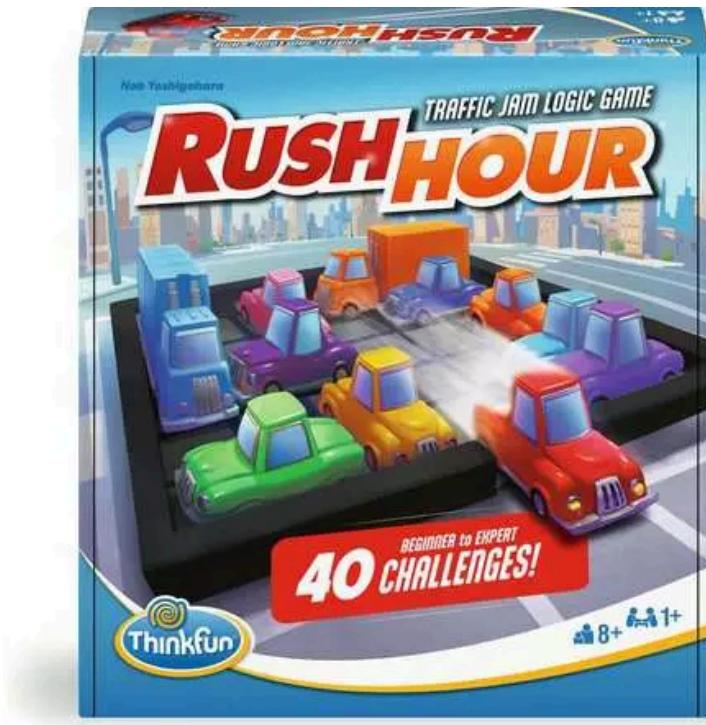
2025

DAFTAR ISI

DAFTAR ISI.....	1
BAB I – DESKRIPSI MASALAH.....	3
BAB II – LANDASAN TEORI.....	5
2.1. Algoritma Uniform Cost Search (UCS).....	5
2.2. Algoritma Greedy Best First Search.....	6
2.3. Algoritma A* Search.....	6
2.4. Iterative Deepening A*.....	7
BAB III – ANALISIS PEMECAHAN MASALAH.....	10
3.1. Analisis Pemecahan Masalah Permainan Rush Hour.....	10
3.2. Penjelasan Algoritma UCS, Greedy Best First Search, A*, dan Iterative Deepening A* dalam Penyelesaian Rush Hour.....	11
3.2.1. UCS.....	11
3.2.2. Greedy Best-First Search.....	12
3.2.3. A*.....	12
3.2.4. Iterative Deepening A* (IDA*).....	13
3.3. Fungsi f(n) dan g(n).....	13
3.4. Analisis Admissibility Heuristik A*	14
3.5. Perbandingan Algoritma UCS dan BFS pada Penyelesaian Rush Hour.....	16
3.6. Analisis Efisiensi A* dengan UCS pada Penyelesaian Rush Hour.....	17
3.7. Analisis Optimalitas Greedy Best First Search pada Penyelesaian Rush Hour.....	17
BAB IV – IMPLEMENTASI.....	19
4.2. Struktur Program.....	19
4.2. Cara Menjalankan Program.....	22
4.3. Implementasi (Source Code): Tipe Data.....	22
4.4. Implementasi (Source Code): Algoritma.....	24
4.5. Implementasi (Source Code): Utilities dan Helpers.....	32
BAB V – PENGUJIAN DAN ANALISIS.....	39
5.1. Hasil Pengujian.....	39
5.2. Analisis Pengujian.....	45
BAB VI – IMPLEMENTASI BONUS.....	49
6.1. Algoritma Pathfinding Alternatif: IDA*.....	49
6.2. Implementasi Dua Heuristik Alternatif.....	50
6.3. Antarmuka Grafis (GUI) dengan Next.js.....	51
BAB VII – PENUTUP.....	54
7.1. Kesimpulan.....	54
7.2. Saran.....	54
7.3. Refleksi.....	55
DAFTAR PUSTAKA.....	56

LAMPIRAN.....	57
A. Pranala Repository dan GUI.....	57
B. Tabel Ketercapaian.....	57

BAB I – DESKRIPSI MASALAH



Gambar 1. Rush Hour Puzzle

(Sumber: <https://www.thinkfun.com/en-US/products/educational-games/rush-hour-76582>)

Rush Hour adalah 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.

Komponen penting dari permainan Rush Hour terdiri dari:

- **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.

- **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.
- **Primary Piece** – Primary piece adalah kendaraan utama yang harus dikeluarkan dari papan (biasanya berwarna merah). Hanya boleh terdapat satu primary piece.
- **Pintu Keluar** – Pintu keluar adalah tempat primary piece dapat digerakkan keluar untuk menyelesaikan permainan
- **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.

BAB II – LANDASAN TEORI

2.1. Algoritma Uniform Cost Search (UCS)

Algoritma Uniform Cost Search (UCS) adalah salah satu algoritma route planning ‘penentuan rute’ tanpa adanya informasi tambahan mengenai tujuan pencarian (Uninformed Search/Blind Search). UCS mengevaluasi dan mengeksplorasi simpul dengan mempertimbangkan cost untuk mencapai suatu node dari node awal ($g(n)$). Dalam UCS, $f(n) = g(n)$, yaitu biaya yang sudah dikeluarkan dari simpul awal (root node) untuk mencapai simpul n. UCS memanfaatkan struktur data priority queue untuk menyimpan simpul-simpul yang menunggu diperiksa, di mana simpul dengan biaya terkecil akan diproses terlebih dahulu.

Setiap kali sebuah simpul diperiksa, algoritma ini akan memperluas simpul tersebut dengan mengunjungi semua tetangganya yang belum dieksplorasi atau yang bisa dicapai dengan biaya lebih rendah melalui jalur baru. Jika ditemukan jalur dengan biaya yang lebih rendah menuju tetangga tersebut, maka nilai biaya diperbarui dan tetangga dimasukkan kembali ke dalam antrian dengan prioritas yang diperbarui. Proses ini menjamin bahwa jalur yang ditemukan menuju simpul tujuan merupakan jalur dengan biaya minimum.

Langkah-langkah algoritma UCS:

1. Inisialisasi frontier (priority queue) dengan simpul awal start dan tetapkan biaya jalur $g(\text{start}) = 0$.
2. Buat himpunan kosong visited untuk melacak simpul-simpul yang telah dieksplorasi.
3. Selama frontier tidak kosong, pilih simpul dengan nilai biaya jalur terendah (g) dari frontier.
4. Jika simpul yang dipilih adalah simpul tujuan, rekonstruksi jalur dari simpul awal ke simpul tujuan dan hentikan pencarian.
5. Jika bukan, keluarkan simpul tersebut dari frontier dan tambahkan ke himpunan visited. Kemudian, perluas simpul tersebut dengan memeriksa tetangganya.
6. Untuk setiap tetangga yang belum ada di visited, hitung nilai sementara $g(n)$, dan jika nilai ini lebih kecil daripada nilai g yang sudah tercatat, perbarui biaya dan tetapkan simpul saat ini sebagai induk tetangga tersebut, kemudian masukkan tetangga ke frontier.
7. Ulangi langkah 3 sampai simpul tujuan ditemukan atau frontier kosong.

2.2. Algoritma Greedy Best First Search

Greedy Best First Search (GBFS) adalah algoritma pencarian yang menggunakan informasi heuristik untuk memperkirakan jarak atau biaya dari simpul saat ini menuju simpul tujuan, dengan tujuan menemukan jalur secara cepat dan efisien. Algoritma ini memilih untuk memperluas simpul yang memiliki nilai heuristik $h(n)$ terkecil terlebih dahulu, tanpa memperhitungkan biaya yang sudah dikeluarkan dari titik awal ($g(n)$ tidak digunakan). Oleh karena itu, GBFS hanya mengandalkan fungsi heuristik dalam menentukan urutan ekspansi simpul. Struktur data priority queue digunakan untuk menyimpan simpul-simpul yang akan diperiksa, diurutkan berdasarkan nilai heuristik.

Langkah-langkah sederhana dari algoritma GBFS:

1. Inisialisasi: Mulai dari simpul awal (misalnya simpul root atau titik start). Masukkan simpul ini ke dalam priority queue berdasarkan nilai heuristiknya.
2. Perluas Simpul: Ambil simpul dengan nilai heuristik terendah dari priority queue, kemudian evaluasi semua tetangganya. Berikan setiap tetangga nilai berdasarkan fungsi heuristik, yaitu estimasi jarak atau biaya menuju simpul tujuan.
3. Pilih Simpul Terbaik: Pilih simpul tetangga yang memiliki nilai heuristik terendah dan masukkan ke priority queue untuk diekspansi berikutnya.
4. Cek Tujuan: Jika simpul yang dipilih adalah simpul tujuan, hentikan pencarian dan rekonstruksi jalur dari simpul awal ke simpul tujuan.
5. Lanjutkan Pencarian: Jika simpul tujuan belum ditemukan, ulangi langkah 2 sampai 4 sampai tujuan tercapai atau priority queue kosong (tidak ada jalur menuju tujuan).

2.3. Algoritma A* Search

A* adalah algoritma pencarian yang menggabungkan kelebihan Uniform Cost Search (UCS) dan Greedy Best First Search (GBFS) dengan memanfaatkan fungsi evaluasi $f(n) = g(n) + h(n)$. Pada fungsi ini, $g(n)$ merupakan biaya aktual dari simpul awal menuju simpul n , sedangkan $h(n)$ adalah estimasi biaya heuristik dari simpul n menuju simpul tujuan. Dengan menggabungkan kedua komponen ini, A* dapat mencari jalur dengan biaya total minimum secara lebih efisien dibanding UCS yang hanya memperhitungkan biaya $g(n)$ dan GBFS yang hanya bergantung pada heuristik $h(n)$.

Jika heuristik yang digunakan bersifat admissible (tidak pernah melebih-lebihkan biaya sebenarnya menuju tujuan) dan konsisten (memenuhi sifat segitiga), maka A* dijamin menemukan solusi optimal. Dalam konteks puzzle seperti Rush Hour, heuristik yang umum dipakai misalnya adalah jumlah mobil yang menghalangi jalur keluar mobil target atau estimasi jarak terdekat menuju pintu keluar. Dengan demikian, pencarian menjadi lebih terarah dan cepat.

Langkah-langkah Algoritma A*:

1. Inisialisasi: Mulai dari simpul awal, masukkan simpul ini ke dalam priority queue dengan nilai $f(n) = g(n) + h(n)$, di mana $g(start) = 0$ dan $h(start)$ dihitung dari heuristik.
2. Perluas Simpul: Ambil simpul dengan nilai $f(n)$ terendah dari priority queue untuk diekspansi.
3. Cek Tujuan: Jika simpul yang diambil adalah simpul tujuan, hentikan pencarian dan rekonstruksi jalur.
4. Periksa Tetangga: Untuk setiap tetangga simpul yang sedang diperiksa, hitung biaya jalur baru $g(tetangga) = g(current) + cost(current + tetangga)$, dan nilai heuristik $h(tetangga)$.
5. Perbarui Frontier: Jika tetangga belum pernah dikunjungi atau nilai $f(tetangga)$ baru lebih kecil dari nilai sebelumnya, perbarui data tetangga dan masukkan atau perbarui posisi tetangga dalam priority queue.
6. Ulangi: Lanjutkan proses hingga tujuan ditemukan atau frontier kosong (solusi tidak ada).

2.4. Iterative Deepening A*

IDA* menggabungkan keunggulan A* dan Iterative Deepening DFS (IDDFS) untuk menemukan jalur optimal dengan memori yang jauh lebih rendah daripada A* tradisional. Algoritma ini melakukan serangkaian pencarian depth-first dengan ambang batas (threshold) pada nilai $f(n)=g(n)+h(n)$ yang meningkat secara iteratif. Pada setiap iterasi, algoritma hanya menelusuri cabang-cabang dengan $f(n)$ tidak melebihi ambang saat itu, dan ambang berikutnya diatur menjadi nilai minimum $f(n)$ yang melebihi ambang sebelumnya. Proses ini berulang hingga simpul tujuan ditemukan, menjamin optimalitas selama heuristik bersifat admissible dan konsisten, sambil menjaga penggunaan memori menjadi $O(b \cdot d)$ dengan b adalah faktor cabang dan d adalah kedalaman solusi (pencarian depth-first).

Iterative Deepening A* (IDA*) adalah varian depth-first search yang menggunakan fungsi evaluasi $f(n)=g(n)+h(n)$, di mana $g(n)$ adalah biaya aktual dari simpul awal ke n dan $h(n)$ adalah

estimasi biaya dari n ke tujuan. Berbeda dengan A* yang menyimpan seluruh frontier di memori, IDA* hanya menyimpan jalur saat ini (stack), sehingga memori yang diperlukan bersifat linier terhadap kedalaman. Algoritma ini memulai dengan ambang batas awal sama dengan $h(\text{start})$ dan melakukan depth-first search terbatas berdasarkan ambang tersebut

Contoh langkah-langkah algoritma IDA*

1. Inisialisasi:

- Tetapkan ambang awal bound = $h(\text{start})$, di mana start adalah simpul awal dan $h(\text{start})$ adalah heuristik yang dapat diterima (admissible) yang mengestimasi biaya ke simpul tujuan. Karena $g(\text{start}) = 0$, maka bound = $f(\text{start}) = h(\text{start})$.
- Inisialisasi path = [start] sebagai tumpukan (stack) untuk menyimpan jalur pencarian saat ini.
- Pastikan heuristik h bersifat admissible (tidak pernah melebihi biaya sebenarnya) dan sebaiknya konsisten untuk menjamin optimalitas dan efisiensi.)

2. Pencarian Terbatas (search(node, g, bound)):

- Hitung $f = g + h(\text{node})$, di mana g adalah biaya dari start ke node, dan $h(\text{node})$ adalah nilai heuristik. f digunakan untuk menentukan ambang iterasi berikutnya.
- Jika $f > \text{bound}$, kembalikan f (pangkas cabang ini dan kembalikan nilai f untuk pembaruan ambang).
- Jika node adalah simpul tujuan, kembalikan “FOUND” (menandakan solusi ditemukan).
- Inisialisasi $\text{min_t} = \text{tak hingga}$ untuk melacak nilai f terkecil dari cabang yang dipangkas.
- Untuk setiap tetangga succ dari node yang tidak ada di path (untuk menghindari siklus):
 - Tambahkan succ ke path.
 - Panggil result = search(succ, $g + \text{cost}(\text{node}, \text{succ})$, bound), di mana $\text{cost}(\text{node}, \text{succ})$ adalah biaya tepi (misalnya, 1 untuk graf tanpa bobot).
 - Jika result = FOUND, kembalikan “FOUND”.
 - Jika result \neq FOUND, perbarui $\text{min_t} = \min(\text{min_t}, \text{result})$.

- Hapus succ dari path (lakukan backtracking).
- Kembalikan min_t.

Catatan: Untuk graf besar, dapat digunakan hash set untuk memeriksa simpul yang sudah dikunjungi sebagai pengganti pemeriksaan linier pada path.

3. Iterasi:

- Panggil result = search(start, 0, bound).
- Jika result = FOUND, maka path berisi jalur optimal dari start ke tujuan.
- Jika result \neq FOUND, tetapkan bound = result (nilai f terkecil yang melebihi ambang saat ini) dan ulangi pencarian dengan bound baru.
- Hentikan pencarian jika tidak ada solusi (misalnya, bound melebihi batas yang ditentukan atau tidak ada simpul baru yang dieksplorasi).

BAB III – ANALISIS PEMECAHAN MASALAH

3.1. Analisis Pemecahan Masalah Permainan Rush Hour

Permainan Rush Hour melibatkan status papan permainan yang berbeda-beda dari posisi awal *primary piece* hingga *primary piece* dapat melewati pintu keluar. Untuk menyelesaikan permainan ini secara komputasional, algoritma pathfinding seperti Uniform-Cost Search (UCS), Greedy Best-First Search, A*, dan Iterative Deepening A* (IDA*) dapat diterapkan. Algoritma ini mencari jalur optimal dari status awal papan ke status tujuan, yaitu saat mobil utama mencapai titik keluar. Namun, diperlukan definisi status yang tepat agar proses pencarian dapat berjalan, berhasil mendapatkan solusi, dan efisien. Oleh karena itu, pertama-tama penulis harus mendefinisikan status dari permainan Rush Hour, seperti status valid dan kriteria status akhir.

Untuk menerapkan algoritma pathfinding, status permainan harus didefinisikan dengan jelas agar setiap kemungkinan konfigurasi papan dapat direpresentasikan, dievaluasi, dan dibandingkan secara efisien. Penulis mendefinisikan status permainan Rush Hour sebagai berikut:

1. Status Papan: Sebuah status adalah konfigurasi lengkap papan pada satu waktu tertentu, yang mencakup:
 - a. Posisi semua kendaraan: Setiap kendaraan (mobil atau truk) memiliki panjang, orientasi (horizontal atau vertikal), dan koordinat posisi (baris dan kolom). Koordinat ini biasanya diwakili oleh titik referensi kendaraan, seperti ujung kiri untuk kendaraan horizontal atau ujung atas untuk kendaraan vertikal.
 - b. Titik keluar: Koordinat baris dan kolom tempat mobil utama harus mencapai untuk menyelesaikan permainan (misalnya, exitRow dan exitCol).
2. Status Valid: Sebuah status dianggap valid jika:
 - a. Tidak ada kendaraan yang tumpang tindih (overlap) di papan.
 - b. Semua kendaraan berada dalam batas papan (6x6).
 - c. Setiap gerakan kendaraan mematuhi orientasinya (horizontal hanya bergerak ke kiri/kanan, vertikal hanya ke atas/bawah) dan tidak melintasi kendaraan lain atau dinding papan.
3. Kriteria Status Akhir (Goal State): Status akhir tercapai ketika mobil utama mencapai titik keluar, yaitu ujung depan mobil utama (bagian kanan untuk mobil horizontal) berada pada koordinat keluar yang ditentukan. Dalam beberapa kasus, ini berarti mobil utama telah "keluar" dari papan, yang diizinkan sebagai gerakan legal.

Permainan Rush Hour dapat dimodelkan sebagai masalah pencarian dalam ruang status (state-space search). Setiap status papan merepresentasikan sebuah node dalam graf, dan setiap gerakan legal (misalnya, memindahkan kendaraan satu petak) merepresentasikan edge yang menghubungkan dua status. Algoritma pathfinding digunakan untuk mencari jalur terpendek dari status awal (konfigurasi awal papan) ke status akhir (mobil utama di titik keluar). Untuk efisiensi, algoritma harus:

- Menghindari menjelajahi status yang sama berulang kali (menggunakan hash dari representasi string papan).
- Menggunakan heuristik untuk memandu pencarian ke arah solusi (khususnya untuk A* dan Greedy Best-First Search).
- Mengelola kompleksitas ruang status yang besar, karena jumlah kemungkinan konfigurasi papan bisa sangat banyak.

3.2. Penjelasan Algoritma UCS, Greedy Best First Search, A*, dan Iterative Deepening A* dalam Penyelesaian Rush Hour

Berikut adalah penjelasan konseptual dari empat algoritma pathfinding yang digunakan untuk menyelesaikan permainan Rush Hour, dengan fokus pada prinsip kerja dan penerapannya dalam konteks permainan ini.

3.2.1. UCS

Dalam penerapan UCS, penulis memandang setiap konfigurasi papan sebagai sebuah “node” dalam graf tak beraturan. Dari setiap node, program menghasilkan semua gerakan kendaraan legal—misalnya menggeser mobil A sejauh satu hingga tiga langkah, menghasilkan beberapa node baru.

UCS selalu memilih node yang memiliki biaya terendah dari status awal. Di Rush Hour, setiap gerakan mobil dihitung sebagai satu unit biaya, sehingga UCS akan mengexpansi terlebih dahulu semua node yang bisa dicapai dengan jumlah gerakan paling sedikit. Jika dua node memiliki biaya sama, maka pemilihan bisa berdasarkan urutan penambahan, tetapi secara umum prioritasnya pada biaya terkecil.

Dalam implementasi, penulis menyimpan:

- Biaya $g(n)$ untuk tiap node—yakni jumlah langkah hingga node itu.

- Antrian prioritas yang terurut menurut $g(n)$.
- Daftar visited berisi konfigurasi papan yang sudah dikunjungi dengan biaya tersimpan, agar penulis tidak kembali ke konfigurasi yang sama dengan biaya lebih tinggi.

UCS menjamin menemukan jalur dengan jumlah langkah minimal, tetapi sering kali harus mengeksplorasi banyak konfigurasi sebelum mencapai pintu keluar, terutama pada puzzle yang kompleks.

3.2.2. Greedy Best-First Search

Greedy Best-First Search hanya memperhatikan seberapa dekat sebuah status dengan pintu keluar berdasarkan heuristik, contohnya *manhattan distance*, tanpa mempedulikan berapa banyak langkah yang sudah ditempuh.

Pada setiap langkah, GBFS memilih status yang memiliki nilai heuristik ($h(n)$) terkecil—seolah-olah “melihat” langsung ke arah solusi. Disebabkan tidak mempertimbangkan biaya yang sudah terbuang, GBFS bisa sangat cepat menemukan sebuah solusi, tetapi tidak menjamin jumlah langkahnya minimal.

Dalam implementasi, frontier (antrian node yang menunggu untuk diekspansi) diurutkan hanya berdasarkan nilai heuristik, sedangkan $g(n)$ diabaikan. Daftar visited tetap digunakan agar tidak terjadi eksplorasi ulang status sama, tetapi tidak mencegah jalur yang mungkin lebih panjang secara total.

3.2.3. A*

A* mengombinasikan kekuatan UCS dan Greedy Best-First Search dengan memperhitungkan total estimasi biaya $f(n) = g(n) + h(n)$.

- $g(n)$ adalah jumlah langkah yang sudah dijalani dari status awal.
- $h(n)$ adalah perkiraan langkah yang diperlukan untuk mencapai pintu keluar, menggunakan heuristik yang admissible (tidak melebih-lebih).

Dengan demikian, A* mengeksplorasi status yang paling menjanjikan: tidak hanya yang sudah murah dilalui (g kecil), tetapi juga yang tampak dekat ke tujuan (h kecil). Frontier diurutkan berdasarkan nilai $f(n)$. Daftar visited berisi catatan g terbaik yang pernah dicapai untuk setiap konfigurasi papan, sehingga jika suatu konfigurasi muncul kembali melalui jalur lebih panjang, ia akan diabaikan.

A* menjamin menemukan solusi dengan jumlah langkah optimal (karena heuristiknya admissible), dan biasanya jauh lebih efisien daripada UCS karena memangkas cabang yang terlalu mahal.

3.2.4. Iterative Deepening A* (IDA*)

Iterative Deepening A* dirancang untuk mengurangi penggunaan memori A*, dengan tetap menjaga jaminan optimalitas. Alih-alih menyimpan seluruh frontier di dalam memori, IDA* melakukan serangkaian pencarian depth-first dengan ambang biaya f maksimal.

Prosesnya:

1. Tetapkan ambang pertama sama dengan nilai heuristik status awal.
2. Lakukan pencarian mendalam (depth-first), tetapi hanya menelusuri status yang $f(n) = g(n)+h(n)$ tidak melebihi ambang.
3. Jika tidak menemukan solusi, catat ambang terkecil berikutnya yang terlewati (yaitu nilai f terkecil yang melebihi ambang), dan ulangi pencarian dengan ambang baru tersebut.

Pada implementasi Rush Hour, setiap iterasi DFS menggunakan fungsi generateMoves yang sama, dan heuristik yang sama. Karena DFS tidak menyimpan frontier seluas A*, memori yang diperlukan hanya sebesar kedalaman solusi, tetapi total waktu bisa jauh lebih besar karena banyak iterasi dan eksplorasi ulang cabang.

3.3. Fungsi $f(n)$ dan $g(n)$

Dalam algoritma pathfinding, fungsi $f(n)$ dan $g(n)$ adalah komponen penting yang digunakan untuk mengevaluasi dan memprioritaskan status (node) dalam ruang pencarian. Fungsi $g(n)$ merepresentasikan biaya kumulatif dari status awal ke status saat ini (node n) dalam ruang pencarian. Dalam konteks pathfinding, $g(n)$ biasanya dihitung sebagai jumlah langkah atau biaya yang telah dikeluarkan untuk mencapai node tersebut dari node awal. Dalam Rush Hour, setiap gerakan kendaraan (misalnya, memindahkan mobil atau truk satu petak) dianggap memiliki biaya 1, sehingga $g(n)$ adalah jumlah total gerakan yang telah dilakukan untuk mencapai konfigurasi papan saat ini.

Fungsi $f(n)$ adalah biaya total yang diperkirakan untuk mencapai status akhir (goal state) melalui node n . Fungsi ini biasanya digunakan dalam algoritma pencarian terinformasi seperti A* dan IDA*. Dalam algoritma tersebut, $f(n)$ didefinisikan sebagai:

$$f(n) = g(n) + h(n)$$

keterangan:

- $g(n)$ adalah biaya kumulatif dari status awal ke node n (seperti dijelaskan di atas).
- $h(n)$ adalah nilai heuristik, yaitu perkiraan biaya dari node n ke status akhir. Heuristik harus bersifat *admissible* (tidak melebih-lebihkan biaya sebenarnya) agar algoritma menjamin solusi optimal).

Dalam program penyelesaian Rush Hour, fungsi $f(n)$ dan $g(n)$ diimplementasikan sebagai bagian dari objek GameState (didefinisikan dalam state.ts), yang merepresentasikan status papan permainan pada suatu waktu tertentu. Berikut adalah penerapan kedua fungsi tersebut dalam konteks algoritma yang digunakan:

3.4. Analisis Admissibility Heuristik A*

Dalam algoritma A*, heuristik berfungsi untuk mengarahkan pencarian menuju solusi dengan biaya minimum. Agar A* dapat menjamin solusi yang optimal, heuristik harus bersifat admissible, yaitu tidak boleh melebih-lebihkan biaya sebenarnya dari status saat ini ke status akhir (goal state). Secara matematis, untuk setiap node (n), heuristik ($h(n)$) harus memenuhi:

$$h(n) \leq h^*(n)$$

di mana ($h^* n$) adalah biaya sebenarnya dari jalur terpendek menuju status akhir. Dalam permainan Rush Hour, penulis menggunakan tiga heuristik: Manhattan distance, blocking piece count, dan blocking distance.

3.4.1. Manhattan Distance

Heuristik ini menghitung jarak linier dari ujung depan mobil utama (*primary piece*) ke titik keluar (*exit*) pada papan permainan. Jarak dihitung sebagai jumlah selisih absolut antara koordinat baris dan kolom.

Analisis Admissibility:

- Heuristik ini admissible karena jarak Manhattan adalah estimasi biaya minimum untuk memindahkan mobil utama ke titik keluar jika tidak ada hambatan. Dalam skenario tanpa kendaraan pemblokir, mobil utama bisa bergerak langsung ke titik keluar dengan biaya yang sama dengan jarak Manhattan.
- Dalam permainan Rush Hour, kendaraan lain sering menghalangi jalur, sehingga biaya sebenarnya biasanya lebih besar. Namun, karena heuristik ini tidak pernah melebih-lebihkan biaya sebenarnya, ia memenuhi syarat admissibility.
- Kelemahan: Heuristik ini kurang informatif saat banyak kendaraan menghalangi, sehingga A* mungkin harus mengeksplorasi lebih banyak status.

3.4.2. Blocking Piece Count

Heuristik ini menghitung jumlah kendaraan unik yang menghalangi jalur langsung mobil utama ke titik keluar. Setiap kendaraan di jalur tersebut dihitung sekali sebagai estimasi biaya.

Analisis Admissibility:

- Heuristik ini admissible karena setiap kendaraan pemblokir memerlukan setidaknya satu gerakan untuk dipindahkan dari jalur. Jumlah kendaraan adalah batas bawah (lower bound) dari total gerakan yang dibutuhkan.
- Dalam praktiknya, beberapa kendaraan mungkin butuh lebih dari satu gerakan, tetapi heuristik ini tidak melebih-lebihkan biaya sebenarnya, sehingga tetap admissible.
- Keunggulan: Lebih informatif daripada Manhattan distance saat ada banyak kendaraan pemblokir, meskipun bisa meremehkan biaya jika gerakan tambahan diperlukan.

3.4.3. Blocking Distance

Heuristik ini menghitung estimasi jumlah gerakan minimum yang diperlukan untuk memindahkan semua kendaraan pemblokir agar tidak lagi menghalangi jalur mobil utama ke titik keluar. Jarak pergerakan minimum untuk setiap kendaraan dijumlahkan.

Analisis Admissibility:

- Heuristik ini admissible karena menghitung biaya minimum untuk memindahkan setiap kendaraan pemblokir keluar dari koridor. Biaya sebenarnya bisa lebih besar jika ada ketergantungan antar kendaraan, tetapi heuristik ini tidak melebih-lebihkan.
- Keunggulan: Lebih akurat daripada dua heuristik sebelumnya karena mempertimbangkan jarak pergerakan sehingga lebih dekat ke biaya sebenarnya dan mengurangi jumlah status yang dieksplorasi oleh A*.

3.5. Perbandingan Algoritma UCS dan BFS pada Penyelesaian Rush Hour

Secara teoritis, dalam konteks teka-teki Rush Hour di mana setiap gerakan (menggeser satu kendaraan satu petak) memiliki biaya seragam (unit cost) sebesar 1, algoritma Uniform Cost Search (UCS) dan Breadth-First Search (BFS) menunjukkan perilaku yang sangat mirip dalam hal eksplorasi level-level status berdasarkan jumlah langkah.

Dalam Rush Hour, ruang pencarian direpresentasikan sebagai graf di mana simpul adalah konfigurasi papan dan sisi adalah gerakan valid dengan biaya 1. UCS memilih simpul dengan biaya kumulatif ($g(n)$) terendah menggunakan priority queue, yang dalam kasus biaya seragam berarti memilih simpul pada kedalaman terkecil, mirip dengan BFS yang menjelajahi simpul level demi level menggunakan queue First-In-First-Out (FIFO). Karena biaya setiap gerakan adalah 1, ($g(n)$) pada UCS sama dengan kedalaman simpul, sehingga UCS secara efektif menjelajahi graf berdasarkan kedalaman, seperti BFS. Akibatnya, kedua algoritma ini akan menghasilkan solusi dengan jumlah gerakan minimum yang sama, menjamin jalur terpendek ke status akhir (saat mobil utama mencapai titik keluar).

Namun, meskipun level-level status (berdasarkan jumlah langkah) dieksplorasi dalam urutan yang sama, urutan spesifik simpul dalam satu kedalaman dapat berbeda karena BFS mengikuti urutan penemuan (FIFO), sedangkan UCS bergantung pada implementasi priority queue yang mungkin tidak mempertahankan urutan penemuan untuk simpul dengan biaya yang sama. Selain itu, jika terdapat beberapa jalur optimal, jalur spesifik yang dipilih oleh UCS dan BFS dapat berbeda karena urutan ekspansi simpul dalam kedalaman yang sama tidak dijamin identik. Oleh karena itu, dalam Rush Hour dengan biaya seragam, UCS dan BFS setara dalam hal optimalitas solusi dan urutan eksplorasi level status, tetapi urutan pembangkitan simpul spesifik dan jalur yang dihasilkan mungkin berbeda tergantung pada implementasi dan struktur data yang digunakan.

3.6. Analisis Efisiensi A* dengan UCS pada Penyelesaian Rush Hour

Algoritma A* secara teoritis lebih efisien dibandingkan Uniform Cost Search (UCS) dalam menyelesaikan teka-teki Rush Hour karena kemampuannya memanfaatkan informasi heuristik untuk mengarahkan pencarian menuju solusi optimal dengan lebih cepat. Berikut adalah analisis efisiensi A* dibandingkan UCS berdasarkan beberapa aspek utama:

- Pruning Berbasis Heuristik: A* menggunakan fungsi evaluasi ($f(n) = g(n) + h(n)$), di mana ($g(n)$) adalah biaya kumulatif dari status awal ke status saat ini, dan ($h(n)$) adalah estimasi biaya heuristik ke status akhir. Dengan heuristik yang admissible seperti Manhattan distance, A* memprioritaskan simpul dengan ($f(n)$) terendah, sehingga secara efektif "memangkas" jalur-jalur yang tampak terlalu mahal untuk mencapai tujuan. Sebaliknya, UCS, sebagai algoritma pencarian buta, menjelajahi semua simpul secara bertahap berdasarkan kedalaman tanpa panduan heuristik, sehingga memproses banyak simpul yang tidak relevan dengan solusi.
- Jumlah Simpul yang Dikunjungi Lebih Sedikit: Berkat heuristiknya, A* cenderung melewati status yang jauh dari solusi, mengurangi ukuran frontier (antrean simpul yang akan dieksplorasi) dan closed set (himpunan simpul yang sudah dieksplorasi). Dalam Rush Hour, di mana ruang status bisa sangat besar karena banyaknya konfigurasi papan, pengurangan ini sangat signifikan. UCS, tanpa heuristik, harus menjelajahi semua simpul pada setiap kedalaman, yang meningkatkan jumlah simpul yang dikunjungi, terutama pada teka-teki yang kompleks.
- Waktu Eksekusi: Karena A* memproses lebih sedikit simpul dari frontier, waktu CPU yang dibutuhkan umumnya lebih rendah dibandingkan UCS untuk teka-teki yang sama. Meskipun A* memiliki sedikit overhead per simpul karena perhitungan nilai heuristik ($h(n)$), penghematan dalam jumlah simpul yang dieksplorasi jauh lebih besar, menghasilkan waktu eksekusi yang lebih cepat secara keseluruhan.

Secara keseluruhan, A* menawarkan *trade-off* yang menguntungkan: meskipun memerlukan sedikit biaya komputasi tambahan untuk menghitung heuristik, pengurangan signifikan dalam jumlah simpul yang diproses membuatnya lebih cepat dan lebih hemat memori dibandingkan UCS dalam penyelesaian Rush Hour. Oleh karena itu, A* adalah pilihan yang lebih efisien, terutama untuk teka-teki dengan ruang status yang besar dan kompleks.

3.7. Analisis Optimalitas Greedy Best First Search pada Penyelesaian Rush Hour

Algoritma Greedy Best-First Search (GBFS) tidak menjamin solusi optimal dalam penyelesaian teka-teki Rush Hour, berbeda dengan algoritma seperti Uniform Cost Search (UCS) dan A* yang

menjamin jalur dengan jumlah gerakan minimum. Ketidakoptimalan GBFS berasal dari strategi pencarinya yang hanya memprioritaskan nilai heuristik ($h(n)$), yaitu perkiraan jarak dari status saat ini ke status akhir, tanpa mempertimbangkan biaya kumulatif ($g(n)$) yang merepresentasikan jumlah gerakan yang telah dilakukan dari status awal.

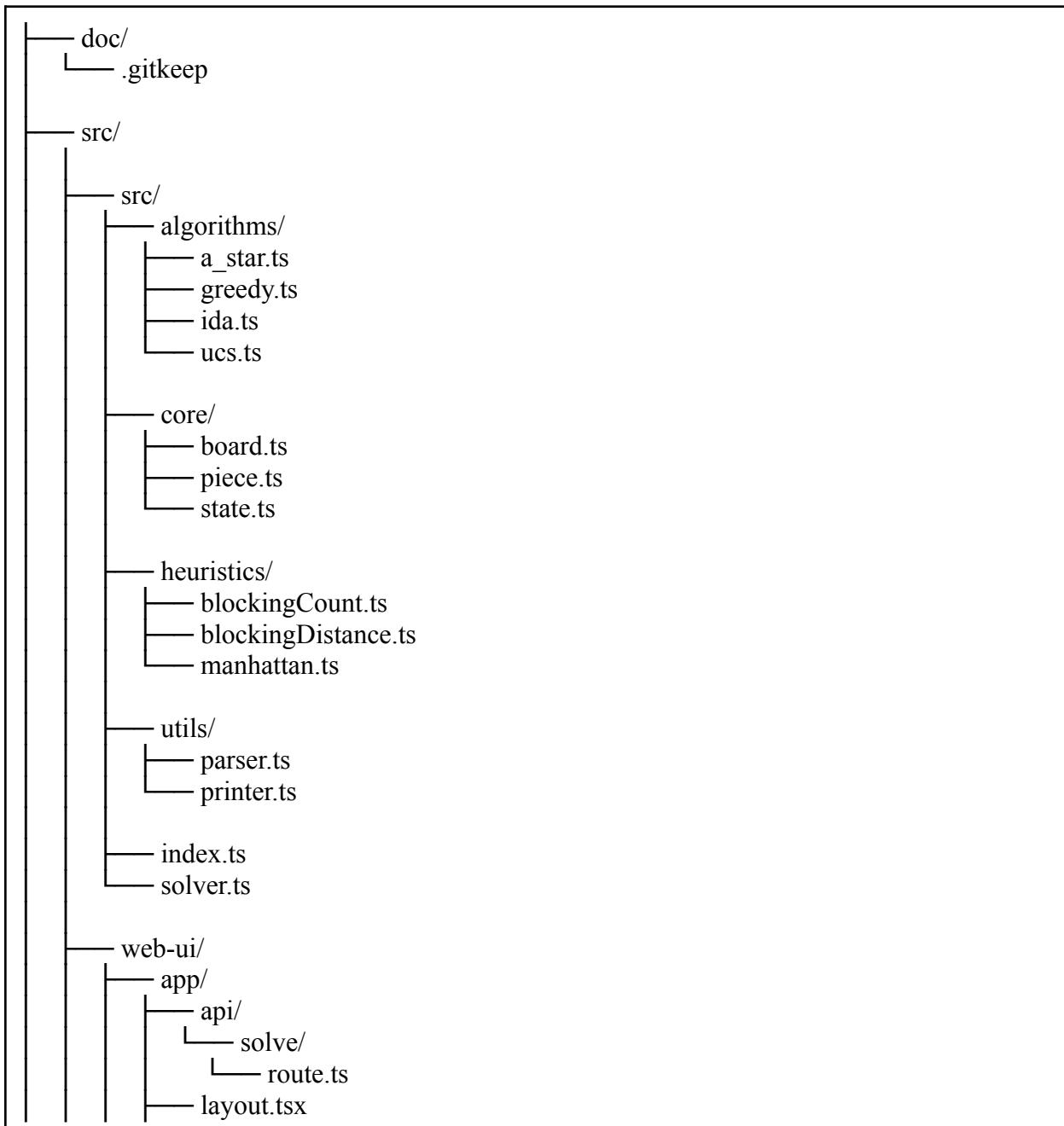
Dalam konteks Rush Hour, di mana tujuannya adalah memindahkan mobil utama ke titik keluar melalui serangkaian gerakan mobil atau truk lain, pendekatan ini dapat menyebabkan GBFS memilih jalur yang tampak menjanjikan secara heuristik tetapi sebenarnya memiliki jumlah gerakan yang lebih banyak.

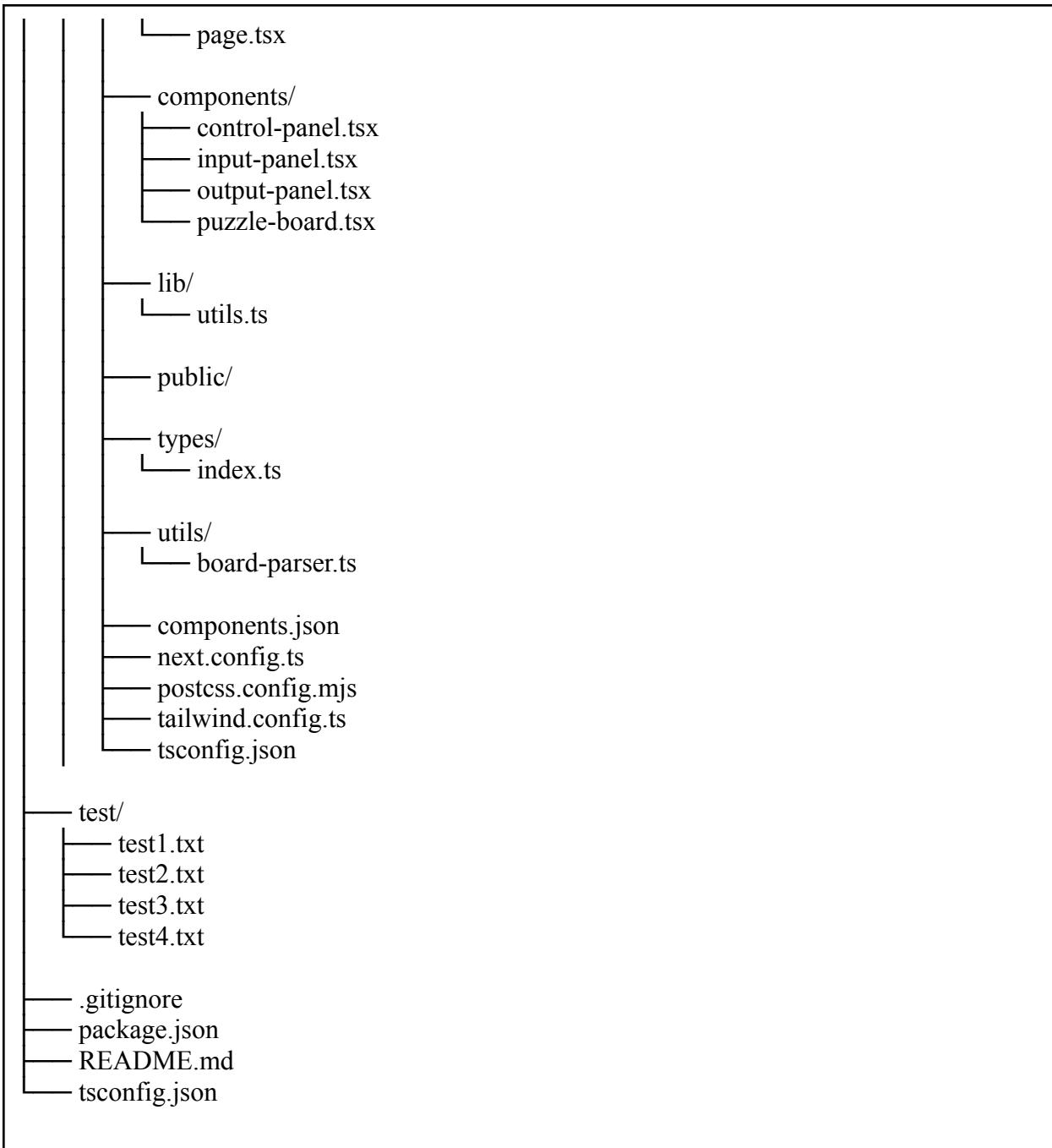
Misalnya, GBFS mungkin memprioritaskan status papan yang membawa mobil utama lebih dekat ke titik keluar berdasarkan heuristik seperti Manhattan distance, namun mengabaikan kebutuhan untuk melakukan gerakan tambahan, seperti memindahkan kendaraan pemblokir secara berulang atau melakukan langkah mundur untuk membuka jalur. Akibatnya, GBFS dapat "tertarik" ke jalur yang berbelit-belit, melewatkkan jalur terpendek yang mungkin memerlukan langkah awal yang kurang menjanjikan menurut heuristik, seperti menggeser kendaraan lain untuk membuka ruang manuver. Walaupun GBFS sering kali menemukan solusi dengan relatif cepat karena fokusnya pada heuristik, jumlah langkah dalam solusi yang dihasilkan cenderung lebih besar dibandingkan solusi optimal yang dihasilkan oleh UCS atau A*. Oleh karena itu, meskipun efisien dalam hal waktu eksekusi untuk teka-teki sederhana, GBFS tidak dapat diandalkan untuk menghasilkan solusi dengan jumlah gerakan minimum dalam Rush Hour, terutama pada konfigurasi papan yang kompleks dengan banyak kendaraan pemblokir.

BAB IV – IMPLEMENTASI

4.1. Struktur Program

Program ini terdiri dari dua bagian utama, yaitu logika pemrosesan utama dan frontend (GUI) berbasis Next.js untuk antarmuka pengguna. Hierarki proyek disusun ulang untuk kejelasan, dan penjelasan tentang tipe data serta algoritma disediakan untuk memberikan gambaran teknis yang komprehensif.





Penjelasan struktur direktori:

- `doc/`: Berisi dokumentasi proyek, dengan file `.gitkeep` untuk menjaga direktori tetap ada dalam kontrol versi.
- `src/`: Direktori utama untuk kode sumber:

- algorithms/: Implementasi algoritma pencarian seperti a_star.ts, greedy.ts, ida.ts, dan ucs.ts.
 - core/: Struktur data inti seperti board.ts (representasi papan permainan), piece.ts (representasi kendaraan), dan state.ts (status permainan dengan metrik seperti (g), (h), dan (f)).
 - heuristics/: Fungsi heuristik seperti manhattan.ts, blockingCount.ts, dan blockingDistance.ts untuk algoritma A* dan IDA*.
 - utils/: Utilitas seperti parser.ts (parsing input teka-teki) dan printer.ts (mencetak solusi di CLI).
 - index.ts dan solver.ts: File utama untuk menjalankan solver dan mengoordinasikan proses pencarian.
- web-ui/: Berisi antarmuka pengguna berbasis Next.js, termasuk:
 - app/: Komponen utama Next.js, termasuk layout.tsx (struktur halaman), page.tsx (halaman utama), dan api/solve/route.ts (endpoint API untuk pemrosesan teka-teki).
 - components/: Komponen React seperti input-panel.tsx (unggah teka-teki dan pemilihan algoritma), puzzle-board.tsx (visualisasi papan), output-panel.tsx (tampilan statistik), dan control-panel.tsx (navigasi langkah solusi).
 - lib/ dan utils/: Fungsi pendukung seperti utils.ts dan board-parser.ts untuk parsing dan manipulasi data di frontend.
 - types/: Definisi tipe TypeScript, seperti index.ts untuk SolutionStep[].
 - public/: Aset statis untuk frontend.
 - File konfigurasi seperti next.config.ts, tailwind.config.ts, dan tsconfig.json untuk mendukung pengembangan frontend.
- test/: Berisi file teka-teki uji (test1.txt hingga test4.txt) untuk menguji solver.
- File Root: File seperti .gitignore, package.json, README.md, dan tsconfig.json mendukung konfigurasi dan dokumentasi proyek secara keseluruhan.

4.2. Cara Menjalankan Program

Terdapat dalam Github: https://github.com/KalengBalsem/Tucil3_15223011/

4.3. Implementasi (*Source Code*): Tipe Data

Code	Keterangan
<pre>export class Piece { readonly id: string; readonly length: number; readonly orientation: Orientation; row: number; col: number;</pre>	Menyimpan informasi tentang blok individu, seperti ukuran, arah (horizontal/vertikal), dan posisi di papan.
<pre>export type Orientation = "H" "V";</pre>	Menunjukkan apakah blok diletakkan secara horizontal atau vertikal di papan.
<pre>export class Board { readonly width: number; readonly height: number; readonly pieces: Piece[]; readonly primary: Piece; readonly exitRow: number; readonly exitCol: number;</pre>	Mengelola ukuran papan, semua blok, blok utama, dan koordinat tujuan. Menyediakan fungsi seperti menghasilkan gerakan, memeriksa tujuan, dan serialisasi.
<pre>export interface Move { pieceId: string; direction: Direction; distance: number; }</pre>	Merepresentasikan aksi menggeser blok tertentu ke arah tertentu dengan jarak tertentu.
<pre>export type Direction = "up" "down" "left" "right";</pre>	Menunjukkan arah yang diizinkan untuk gerakan blok dalam teka-teki.

```
export interface GameState {  
    board: Board;  
    g: number;  
    h: number;  
    f: number;  
    parent?: GameState;  
    lastMove?: Move;  
}
```

Menyimpan status permainan, metrik biaya untuk algoritma pencarian, dan tautan untuk merekonstruksi solusi.

```
export interface SolveResult {  
    solution: GameState | null;  
    nodesExpanded: number;  
}
```

Mengembalikan solusi teka-teki (jika ada) dan statistik pencarian, seperti jumlah simpul yang dieksplorasi.

4.4. Implementasi (*Source Code*): Algoritma

ucs.ts

```
1 import { Board } from "../core/board";
2 import { GameState } from "../core/state";
3 import { PriorityQueue } from "../utils/priorityQueue";
4
5 export function ucs( initialBoard: Board ): {solution?: GameState; nodesExpanded: number} {
6     // Initial state
7     const initialState: GameState = {
8         board: initialBoard,
9         g: 0,
10        h: 0,
11        f: 0,
12        parent: undefined,
13        lastMove: undefined,
14    };
15
16    // frontier: priority queue ordered by f (here f = g)
17    const frontier = new PriorityQueue<GameState>();
18    frontier.push(initialState, initialState.f);
19
20    const visited = new Set<string>();
21
22    let nodesExpanded = 0;
23
24    while (frontier.size() > 0) {
25        const state = frontier.pop()!;
26        const key = state.board.serialize();
27        if (visited.has(key)) {
28            continue; // Skip already visited states
29        }
30        visited.add(key);
31
32        // check goal
33        if (state.board.isGoal()) {
34            return { solution: state, nodesExpanded };
35        }
36
37        // Expand
38        nodesExpanded++;
39        const moves = state.board.generateMoves();
40        for (const move of moves) {
41            // clone board and apply move
42            const boardClone = state.board.clone();
43            // find piece and move
44            const piece = boardClone.pieces.find((p) => p.id === move.pieceId)!;
45            // apply the slide move
46            piece.move(
47                move.direction === 'left' || move.direction === 'up'
48                ? -move.distance
49                : move.distance
50            );
51
52            const newState: GameState = {
53                board: boardClone,
54                g: state.g + 1,
55                h: 0,
56                f: state.g + 1,
57                parent: state,
58                lastMove: move,
59            };
60            frontier.push(newState, newState.f);
61        }
62    }
63
64    // no solution found
65    return { nodesExpanded };
66}
67
68
69
70 }
```

greedy.ts

```
1 import { Board } from './core/board';
2 import { GameState } from './core/state';
3 import { Move } from './core/move';
4 import { PriorityQueue } from './utils/priorityQueue';
5
6 // Import heuristic functions
7 import { manhattan } from './heuristics/manhattan';
8 import { blockingCount } from './heuristics/blockingPieces';
9 import { blockingDistance } from './heuristics/blockingDistance';
10 import { combined } from './heuristics/combined';
11
12 export function greedy( initialBoard: Board, heuristic: 'manhattan' | 'blocking' | 'blockingCount' | 'combined': string ): GameState; nodesExpanded: number } {
13   // select heuristic function
14   const heuristicFn = [
15     manhattan,
16     heuristic === 'blocking'
17       ? blockingDistance
18       : heuristic === 'blockingCount'
19       ? blockingCount
20       : undefined
21   ][ heuristic ];
22
23   const initialState: GameState = {
24     board: initialBoard,
25     g: 0,
26     h: heuristicFn(initialBoard),
27     f: heuristicFn(initialBoard),
28     parent: undefined,
29     lastMove: undefined,
30   };
31
32   const frontier = new PriorityQueue();
33   frontier.push(initialState, initialState.h);
34
35   const visited = new Set<string>();
36   let nodesExpanded = 0;
37
38   while (frontier.size() > 0) {
39     const state = frontier.pop();
40     const key = state.board.serialize();
41     if (visited.has(key)) continue; // Skip already visited states
42     visited.add(key);
43
44     if (state.board.isGoal()) {
45       return { solution: state, nodesExpanded };
46     }
47
48     // expand
49     nodesExpanded++;
50     const moves: Move[] = state.board.generateMoves();
51     for (const move of moves) {
52       const boardClone = state.board.clone();
53       const piece = boardClone.pieces.find(p => p.id === move.pieceId);
54       piece?.move(
55         move.direction === 'left' || move.direction === 'up'
56           ? -move.distance
57           : move.distance
58       );
59       const serialized = boardClone.serialize();
60       if (visited.has(serialized)) continue;
61
62       const h = heuristicFn(boardClone);
63       const newState: GameState = {
64         board: boardClone,
65         g: state.g + 1,
66         h,
67         f: h,
68         parent: state,
69         lastMove: move,
70       };
71       frontier.push(newState, h);
72     }
73   }
74
75   return { nodesExpanded }
76 }
77
```

a_star.ts

```
1 import { Board } from '../core/board';
2 import { GameState } from '../core/state';
3 import { Move } from '../core/move';
4 import { PriorityQueue } from '../utils/priorityQueue';
5
6 // Import heuristic functions
7 import { manhattan } from '../heuristics/manhattan';
8 import { blockingCount } from '../heuristics/blockingPieces';
9 import { blockingDistance } from '../heuristics/blockingDistance';
10 import { combined } from '../heuristics/combined';
11
12 export function aStar(
13     initialBoard: Board,
14     heuristic: 'manhattan' | 'blocking' | 'blockingCount' | 'combined' = 'manhattan'
15 ): { solution?: GameState; nodesExpanded: number } {
16     // select heuristic function
17     const heuristicFn =
18         heuristic === 'manhattan'
19             ? manhattan
20             : heuristic === 'blocking'
21                 ? blockingDistance
22                 : heuristic === 'blockingCount'
23                     ? blockingCount
24                     : combined;
25
26     // initial state
27     const h0 = heuristicFn(initialBoard);
28     const initialState: GameState = {
29         board: initialBoard,
30         g: 0,
31         h: h0,
32         f: h0,
33         parent: undefined,
34         lastMove: undefined,
35     };
36
37     // frontier ordered by f(n) = g(n) + h(n)
38     const frontier = new PriorityQueue<GameState>();
39     frontier.push(initialState, initialState.f);
40
41     const visited = new Map<string, number>();
42     let nodesExpanded = 0;
43
44     while (frontier.size() > 0) {
45         const state = frontier.pop()!;
46         const key = state.board.serialize();
47
48         if (visited.has(key) && visited.get(key)! <= state.g) continue; // Skip already visited states
49         visited.set(key, state.g);
50
51         // goal check
52         if (state.board.isGoal()) {
53             return { solution: state, nodesExpanded };
54         }
55
56         // expand
57         nodesExpanded++;
58         const moves: Move[] = state.board.generateMoves();
59         for (const move of moves) {
60             const boardClone = state.board.clone();
61             const piece = boardClone.pieces.find((p) => p.id === move.pieceId)!;
62             piece.move(
63                 move.direction === 'left' || move.direction === 'up'
64                     ? -move.distance
65                     : move.distance
66             );
67
68             const newG = state.g + 1;
69             const serialized = boardClone.serialize();
70             if (visited.has(serialized) && visited.get(serialized)! <= newG) continue;
71
72             const newH = heuristicFn(boardClone);
73             const newF = newG + newH;
74             const newState: GameState = {
75                 board: boardClone,
76                 g: newG,
77                 h: newH,
78                 f: newF,
79                 parent: state,
80                 lastMove: move,
81             };
82             frontier.push(newState, newF);
83         }
84     }
85
86     return { nodesExpanded };
87 }
```

ida.ts

```
1 // src/algorithms/ida.ts
2 import { Board } from '../core/board';
3 import { GameState } from '../core/state';
4 import { Move } from '../core/move';
5
6 import { manhattan } from '../heuristics/manhattan';
7 import { blockingCount } from '../heuristics/blockingPieces';
8 import { blockingDistance } from '../heuristics/blockingDistance';
9 import { combined } from '../heuristics/combined';
10
11 type HeuristicType = 'manhattan' | 'blockingCount' | 'blocking' | 'combined';
12
13 const heuristics: Record<HeuristicType, (board: Board) => number> = {
14   manhattan,
15   blockingCount,
16   blocking: blockingDistance,
17   combined,
18 };
19
20 /**
21 * IDA* Search: Iterative Deepening A* with selectable heuristic.
22 * Memory-efficient, but may re-expand nodes across iterations.
23 */
24 export function ida(
25   initialBoard: Board,
26   heuristic: HeuristicType = 'manhattan'
27 ): { solution: GameState; nodesExpanded: number } {
28   const hFn = heuristics[heuristic];
29   const startH = hFn(initialBoard);
30   let threshold = startH;
31   let nodesExpanded = 0;
32
33   // Path stack of GameStates
34   const rootState: GameState = {
35     board: initialBoard,
36     g: 0,
37     h: startH,
38     f: startH,
39     parent: undefined,
40     lastMove: undefined,
41   };
42   const path: GameState[] = [rootState];
43
44   let solution: GameState | undefined;
45
46   function dfs(limit: number): number {
47     const current = path[path.length - 1];
48     nodesExpanded++;
49     const f = current.g + current.h;
50     if (f > limit) {
51       return f;
52     }
53     if (current.board.isGoal()) {
54       solution = current;
55       return -Infinity; // signal found
56     }
57
58     let minThreshold = Infinity;
59     const moves = current.board.generateMoves();
60     for (const move of moves) {
61       // apply move, clone board and create new state
62       const boardClone = current.board.clone();
63       const piece = boardClone.pieces.find((p) => p.id === move.pieceId);
64       piece.move(
65         move.direction === 'left' || move.direction === 'up' ? -move.distance : move.distance
66       );
67       const serialized = boardClone.serialize();
68       // avoid immediate cycles
69       if (!path.some((s) => s.board.serialize() === serialized)) {
70         continue;
71       }
72       const g2 = current.g + 1;
73       const h2 = hFn(boardClone);
74       const state2: GameState = {
75         board: boardClone,
76         g: g2,
77         h: h2,
78         f: g2 + h2,
79         parent: current,
80         lastMove: move,
81       };
82
83       path.push(state2);
84       const t = dfs(limit);
85       if (t === -Infinity) {
86         return -Infinity; // found
87       }
88       if (t < minThreshold) {
89         minThreshold = t;
90       }
91       path.pop();
92     }
93     return minThreshold;
94   }
95
96   while (true) {
97     const t = dfs(threshold);
98     if (t === -Infinity) {
99       // solution found
100       break;
101     }
102     if (t === Infinity) {
103       // no solution
104       break;
105     }
106     threshold = t;
107   }
108
109   return { solution, nodesExpanded };
110 }
111
```

board.ts

```
1 import { Piece } from './piece';
2 import { Pawn } from './pawn';
3
4 /**
5  * Represents the game board, including all pieces and the wall.
6  */
7 export class Board {
8   // recently added: number;
9   // recently height: number;
10  // recently pieces: Piece[];
11  // recently pieces: Piece[];
12  // recently exitRow: number;
13  // recently wallRow: number;
14
15  constructor(
16    width: number,
17    height: number,
18    pieces: Piece[],
19    primary: Piece,
20    exitRow: number,
21    wallRow: number
22  ) {
23   this.width = width;
24   this.height = height;
25   this.pieces = pieces;
26   this.primary = primary;
27   this.exitRow = exitRow;
28   this.wallRow = wallRow;
29 }
30
31 // Clone the board and its pieces.
32 public clone(): Board {
33   const intersections = this.pieces.map((p) => p.location);
34   const primaryClone = pieces.find((p) => p.id === this.primary.id);
35   return new Board(
36     this.width,
37     this.height,
38     piecesClone,
39     primaryClone,
40     this.exitRow,
41     this.wallRow
42   );
43 }
44
45 // Check if the primary piece has reached the exit.
46 public isAtExit(): boolean {
47   const p = this.primary;
48   // Bottom exit?
49   if (p.location.row === this.height - 1) {
50     return (
51       p.orientation === 'W' &&
52       p.col === this.exitCol &&
53       p.row + p.length === this.exitRow
54     );
55   }
56   // Top exit?
57   if (p.location.row === -1) {
58     return (
59       p.orientation === 'W' &&
60       p.col === this.exitCol &&
61       p.row === 0
62     );
63   }
64   // Right exit?
65   if (p.location.col === this.width - 1) {
66     return (
67       p.orientation === 'H' &&
68       p.row === this.exitRow &&
69       p.col + p.length === this.exitCol
70     );
71   }
72   // Left exit?
73   if (p.location.col === -1) {
74     return (
75       p.orientation === 'H' &&
76       p.row === this.exitRow &&
77       p.col === 0
78     );
79   }
80   return false;
81 }
82
83 // Serialize the board into a unique string for hashing/dictation ref.
84 public serialize(): string {
85   const wall = Math.max(this.height, this.height + 1);
86   const wallWidth = Math.max(this.width, this.exitCol + 1);
87   const arr = Array.from({length: wall}, () => Array(wallWidth).fill(false));
88
89   // mark pieces
90   for (const p of this.pieces) {
91     for (let i = 0; i < p.length; i++) {
92       const r = p.location.row + i;
93       const c = p.col + (p.orientation === 'H' ? i : 0);
94       if (r > 0 && r < wall && c > 0 && c < wallWidth) {
95         arr[r][c] = true;
96       }
97     }
98   }
99
100 // build string, but if it's the exit cell, point to it
101 return arr
102   .map((row, r) =>
103     row
104       .map((occupied, c) =>
105         occupied
106           ? `#${cell(r, c)}_${p.id}` // if piece ID
107           : `#${this.exitRow - r + 1 + this.exitCol / 100} / ${c / 100}` // if exit
108       )
109   )
110   .join('');
111 }
```

manhattan.ts

```
● ○ ●
1 import { Board } from '../core/board';
2
3 export function manhattan(board: Board): number {
4   const p = board.primary;
5   if (board.isGoal()) return 0;
6
7   // compute number of cells from P's *front* to the exit
8   if (p.orientation === 'H') {
9     // P exits horizontally
10    const frontCol = p.col + p.length;
11    return Math.abs(board.exitCol - frontCol);
12  } else {
13    // P exits vertically
14    const frontRow = p.row + p.length;
15    return Math.abs(board.exitRow - frontRow);
16  }
17}
```

blockingPieces.ts

```
● ○ ●
1 import { Board } from '../core/board';
2
3 export function blockingCount(board: Board): number {
4   const p = board.primary;
5   if (board.isGoal()) return 0;
6
7   const blockers = new Set<string>();
8   const occ = board.buildOccMap();
9
10  // Walk the corridor cells
11  let r = p.row + (p.orientation === 'V' ? p.length : 0);
12  let c = p.col + (p.orientation === 'H' ? p.length : 0);
13  const dr = p.orientation === 'V' ? (board.exitRow > p.row ? 1 : -1) : 0;
14  const dc = p.orientation === 'H' ? (board.exitCol > p.col ? 1 : -1) : 0;
15
16  while (r !== board.exitRow + dr || c !== board.exitCol + dc) {
17    const id = board.cellAt(r, c);
18    if (id && id !== '.' && id !== 'K') blockers.add(id);
19    r += dr;
20    c += dc;
21  }
22
23  // P itself needs one slide plus each blocker
24  return 1 + blockers.size;
25}
```

blockingDistance.ts

```
1 import { Board } from '../core/board';
2
3 /**
4  * For each distinct vehicle blocking P's direct path to the exit,
5  * compute how many steps (slides) it must make to clear the corridor,
6  * and sum those distances. Admissible since each blocker must move
7  * at least that many times.
8 */
9 export function blockingDistance(board: Board): number {
10   const p = board.primary;
11   const occ = board.buildOccMap();
12   const seen = new Set<string>();
13   let total = 0;
14
15   // Determine the cells along P's exit corridor
16   const corridor: [number, number][] = [];
17   if (p.orientation === 'H') {
18     const row = p.row;
19     const start = p.col + p.length;
20     const end = board.exitCol;
21     const step = start < end ? 1 : -1;
22     for (let c = start; c !== end + step; c += step) {
23       corridor.push([row, c]);
24     }
25   } else {
26     const col = p.col;
27     const start = p.row + p.length;
28     const end = board.exitRow;
29     const step = start < end ? 1 : -1;
30     for (let r = start; r !== end + step; r += step) {
31       corridor.push([r, col]);
32     }
33   }
34
35   for (const [r, c] of corridor) {
36     const id = board.cellAt(r, c);
37     if (!id || id === '.' || id === 'K' || seen.has(id)) continue;
38     seen.add(id);
39
40     // Locate that piece
41     const piece = board.pieces.find((x) => x.id === id)!;
42
43     // Try sliding it 1, 2, 3... steps along its orientation
44     // until it no longer overlaps any corridor cell
45     let dist = 1;
46     outer: for (; dist++) {
47       // For each of the piece's cells after sliding 'dist':
48       for (let i = 0; i < piece.length; i++) {
49         const nr = piece.row + (piece.orientation === 'V' ? i : 0)
50           + (piece.orientation === 'V' ? dist * (board.exitRow > piece.row ? 1 : -1) : 0);
51         const nc = piece.col + (piece.orientation === 'H' ? i : 0)
52           + (piece.orientation === 'H' ? dist * (board.exitCol > piece.col ? 1 : -1) : 0);
53
54         // Allow stepping into the exit for the primary only
55         if (nr === board.exitRow && nc === board.exitCol && piece.id === p.id) {
56           continue;
57         }
58         // If this cell is still inside the grid, it must be free
59         if (nr >= 0 && nr < board.height && nc >= 0 && nc < board.width) {
60           if (occ[nr][nc]) break outer; // blocked, try next dist
61         }
62         // otherwise out of grid (not always allowed), but we only care clearing corridor
63       }
64       // Check if after sliding 'dist', none of the piece's new cells sit in any corridor cell
65       let stillInCorridor = false;
66       for (const [cr, cc] of corridor) {
67         if (piece.orientation === 'H') {
68           // corridor is horizontal in row=p.row; check any overlap
69           if (cr === piece.row &&
70             cc >= piece.col + dist * (board.exitCol > piece.col ? 1 : -1) &&
71             cc < piece.col + dist * (board.exitCol > piece.col ? 1 : -1) + piece.length
72           ) {
73             stillInCorridor = true;
74             break;
75           }
76         } else {
77           if (cc === piece.col &&
78             cr >= piece.row + dist * (board.exitRow > piece.row ? 1 : -1) &&
79             cr < piece.row + dist * (board.exitRow > piece.row ? 1 : -1) + piece.length
80           ) {
81             stillInCorridor = true;
82             break;
83           }
84         }
85       if (!stillInCorridor) {
86         total += dist;
87         break;
88       }
89     }
90   }
91
92   return total;
93 }
94 }
```

combined.ts

```
1 import { Board } from '../core/board';
2 import { manhattan } from './manhattan';
3 import { blockingDistance } from './blockingDistance';
4
5 export function combined(board: Board): number {
6   return manhattan(board) + blockingDistance(board);
7 }
```

4.5. Implementasi (*Source Code*): *Utilities* dan *Helpers*

parser.ts

```

● ● ●

1 import fs from "fs";
2 import path from "path";
3 import { Piece, Orientation } from "../core/piece";
4 import { Board } from "../core/board";
5
6 export function parsePuzzle(filePath: string): Board {
7   const content = fs.readFileSync(filePath, "utf-8").trim();
8   const lines = content.split(/\r?\n/).map(l => l.trim());
9
10  if (lines.length < 3) {
11    throw new Error("invalid file format: not enough lines");
12  }
13
14  // parse dimensions
15  const [dimLine, countLine, ...restLines] = lines;
16  const [N, M] = dimLine.split("x").map(Number);
17  if (!Number.isInteger(N) || !Number.isInteger(M)) {
18    throw new Error(`Invalid dimensions line: ${dimLine}`);
19  }
20
21  // Parse number of non-primary pieces (unused for logic, but can validate)
22  const Nc = parseInt(countLine, 10);
23  if (isNaN(Nc) || Nc < 0) {
24    throw new Error(`Invalid piece count: ${countLine}`);
25  }
26
27  // Process grid lines, handling exit 'K' outside the AxB grid
28  const processedRows: string[] = [];
29  let exitRow: number | null = null;
30  let exitCol: number | null = null;
31
32  for (let i = 0; i < restLines.length; i++) {
33    const rowStr = restLines[i];
34
35    // Case: extra line with exit on top/bottom
36    if (rowStr[0] === "K" && rowStr.slice(1).includes("K")) {
37      const c = rowStr.indexOf("K");
38
39      if (processedRows.length === 0) {
40        exitRow = -1;
41        exitCol = c;
42      } else if (processedRows.length === A) {
43        exitRow = A;
44        exitCol = c;
45      } else {
46        throw new Error(`Unexpected exit line at row ${i}: ${rowStr}`);
47      }
48      continue;
49    }
50
51    // Case: normal or side-exit row
52    if (rowStr.length === A) {
53      processedRows.push(rowStr);
54    } else if (rowStr.length === B + 1 && rowStr.includes("K")) {
55      // exit on left or right edge
56      if (rowStr[0] === "K") {
57        const exitRow = processedRows.length;
58        const exitCol = -1;
59        console.log(`Detected left exit at row ${exitRow}`);
60        processedRows.push(rowStr.slice(1));
61      } else if (rowStr[B] === "K") {
62        const exitRow = processedRows.length;
63        const exitCol = B;
64        processedRows.push(rowStr.slice(0, B));
65
66        throw new Error(`Invalid row with extra char: ${rowStr}`);
67      }
68    } else {
69      throw new Error(
70        `Grid line ${[processedRows.length]} length mismatch: expected ${B} or ${B + 1}, got ${rowStr.length}`
71      );
72    }
73
74    if (processedRows.length === A) {
75      throw new Error(`Expected ${A} grid rows, got ${[processedRows.length]}`);
76    }
77  if (exitRow === null || exitCol === null) {
78    throw new Error(`No exit (K) found on any edge`);
79  }
80
81  // Collect cells for places
82  type Cell = { char: string; row: number; col: number };
83  const cells: Cell[] = [];
84  for (let r = 0; r < A; r++) {
85    const rowStr = processedRows[r];
86    for (let c = 0; c < B; c++) {
87      for (let s = 0; s < rowStr.length; s++) {
88        const ch = rowStr[s];
89        if (ch === ".") continue;
90        cells.push({ char: ch, row: r, col: c });
91      }
92    }
93
94    // Group cells by place ID
95    const groups = new Map<string, Cell[]>();
96    for (const cell of cells) {
97      if (cell.char === "K") continue;
98      if (!groups.has(cell.char)) groups.set(cell.char, []);
99      groups.get(cell.char).push(cell);
100    }
101
102  for (const [id, pts] of groups.entries()) {
103    // Determine orientation
104    const sameRow = pts.every(p => p.row === pts[0].row);
105    const sameCol = pts.every(p => p.col === pts[0].col);
106    let orientation: Orientation;
107    if (sameRow && !sameCol) orientation = "H";
108    else if (!sameRow && !sameCol) orientation = "V";
109    else throw new Error(`Place ${id} has non-linear cells`);
110
111    // Assign to top-leftmost
112    pts.sort((a, b) => a.row - b.row || a.col - b.col);
113    const anchor = pts[0];
114    const length = pts.length;
115
116    const piece = new Piece(id, length, orientation, anchor.row, anchor.col);
117    if (id === "P") primary = piece;
118    pieces.push(piece);
119  }
120
121  if (!primary) throw new Error("No primary piece (P) found");
122  if (pieces.length !== N + 1) {
123    console.warn(`Warning: parsed ${pieces.length - 1} non-primary, expected ${N}`);
124  }
125
126  return new Board(B, A, pieces, primary, exitRow, exitCol);
127}

```

```

1  /**
2   * Pure in-memory parser for Rush Hour puzzles from text.
3   * Does not touch the filesystem.
4   *
5   * Expected input format:
6   * 1st line: "<cols> <rows>"
7   * 2nd line: "<nonPrimaryPieceCount>"
8   * Next <rows> lines: strings of length <cols> with letters for pieces and '.' for empty.
9   * Exit 'K' may appear on edges to indicate the exit location.
10 */
11 export function parsePuzzleFromString(content: string): Board {
12   const lines = content.trim().split(/\r?\n/).map(l => l.trim());
13   if (lines.length < 3) throw new Error('Insufficient lines in puzzle input');
14
15   // Parse dimensions
16   const [dimLine, countLine, ...restLines] = lines;
17   const [B, A] = dimLine.split('/').map(n => parseInt(n, 10)); // cols = B, rows = A
18   if (!Number.isInteger(A) || !Number.isInteger(B)) {
19     throw new Error(`invalid dimensions line: ${dimLine}`);
20   }
21
22   // Optional: validate non-primary piece count
23   const expectedNonPrimary = parseInt(countLine, 10);
24   if (isNaN(expectedNonPrimary) || expectedNonPrimary < 0) {
25     throw new Error(`invalid piece count: ${countLine}`);
26   }
27
28   // Handle grid lines and exit detection
29   const processedRows: string[] = [];
30   let exitRow: number | null = null;
31   let exitCol: number | null = null;
32
33   for (const rowRaw of restLines) {
34     const rowStr = rowRaw.replace(/\s/g, '');
35     // Extra line with only K (top/bottom exit)
36     if (rowStr.length <= B && rowStr.includes('K')) {
37       const c = rowStr.indexOf('K');
38       exitRow = processedRows.length === 0 ? -1 : A;
39       exitCol = c;
40       continue;
41     }
42     // Normal or side-exit row
43     if (rowStr.length === B) {
44       processedRows.push(rowStr);
45     } else if (rowStr.length === B + 1 && rowStr.includes('K')) {
46       // Left or right edge exit
47       if (rowStr[0] === 'K') {
48         exitRow = processedRows.length;
49         exitCol = 0;
50         processedRows.push(rowStr.slice(1));
51       } else if (rowStr[B] === 'K') {
52         exitRow = processedRows.length;
53         exitCol = B;
54         processedRows.push(rowStr.slice(0, B));
55       } else {
56         throw new Error(`Invalid row with extra char: ${rowStr}`);
57       }
58     } else {
59       throw new Error(
60         `Grid line ${processedRows.length} length mismatch: expected ${B} or ${B + 1}, got ${rowStr.length}`;
61     );
62   }
63   if (processedRows.length === A) break;
64 }
65
66 if (processedRows.length !== A) {
67   throw new Error(`Expected ${A} grid rows, got ${processedRows.length}`);
68 }
69 if (exitRow === null || exitCol === null) {
70   throw new Error(`No exit (K) found on any edge`);
71 }
72
73 // Collect non-empty cells
74 type Cell = { char: string; row: number; col: number };
75 const cells: Cell[] = [];
76 for (let r = 0; r < A; r++) {
77   for (let c = 0; c < B; c++) {
78     const ch = processedRows[r][c];
79     if (ch !== '.') cells.push({ char: ch, row: r, col: c });
80   }
81 }
82
83 // Group cells by piece ID (excluding 'K')
84 const groups = new Map();
85 for (const cell of cells) {
86   if (cell.char === 'K') continue;
87   const arr = groups.get(cell.char) || [];
88   arr.push(cell);
89   groups.set(cell.char, arr);
90 }
91
92 const pieces: Piece[] = [];
93 let primary: Piece | null = null;
94
95 for (const [id, pts] of groups.entries()) {
96   // Determine orientation
97   const sameRow = pts.every(p => p.row === pts[0].row);
98   const sameCol = pts.every(p => p.col === pts[0].col);
99   let orientation: Orientation;
100  if (sameRow && !sameCol) orientation = 'H';
101  else if (sameCol && !sameRow) orientation = 'V';
102  else throw new Error(`Piece ${id}' has non-linear cells`);
103
104  // Sort to find anchor
105  pts.sort((a, b) => a.row - b.row || a.col - b.col);
106  const anchor = pts[0];
107  const length = pts.length;
108
109  const piece = new Piece(id, length, orientation, anchor.row, anchor.col);
110  if (id === 'P') primary = piece;
111  pieces.push(piece);
112 }
113
114 if (!primary) throw new Error('No primary piece (P) found');
115 if (pieces.length !== expectedNonPrimary + 1) {
116   console.warn(`Warning: parsed ${pieces.length - 1} non-primary, expected ${expectedNonPrimary}`);
117 }
118
119 // Construct and return board
120 return new Board(B, A, pieces, primary, exitRow, exitCol);
121 }

```

priorityQueue.ts

```
 1  /**
 2   * Generic Min-Heap Priority Queue implementation.
 3   * Items with the lowest priority value are dequeued first.
 4   */
 5
 6  export class PriorityQueue<T> {
 7      private heap: { item: T; priority: number}[] = [];
 8
 9      // returns the number of elements in the queue
10     public size(): number {
11         return this.heap.length;
12     }
13
14     // add an item with the given priority to the queue
15     public push(item: T, priority: number): void {
16         const node = { item, priority };
17         this.heap.push(node);
18         this.bubbleUp(this.heap.length - 1);
19     }
20
21     // remove and returns the element with the highest priority (lowest value)
22     public pop(): T | undefined {
23         const heap = this.heap;
24         if (heap.length === 0) return undefined;
25         this.swap(0, heap.length - 1);
26         const node = heap.pop();
27         this.bubbleDown(0);
28         return node!.item;
29     }
30
31     private bubbleUp(index: number): void {
32         const heap = this.heap;
33         let idx = index;
34         while (idx > 0) {
35             const parentIdx = Math.floor((idx - 1) / 2);
36             if (heap[idx].priority >= heap[parentIdx].priority) break;
37             this.swap(idx, parentIdx);
38             idx = parentIdx;
39         }
40     }
41
42     private bubbleDown(index: number): void {
43         const heap = this.heap;
44         let idx = index;
45         const length = heap.length;
46
47         while (true) {
48             const leftIdx = 2 * idx + 1;
49             const rightIdx = 2 * idx + 2;
50             let smallest = idx;
51
52             if (leftIdx < length && heap[leftIdx].priority < heap[smallest].priority) {
53                 smallest = leftIdx;
54             }
55             if (rightIdx < length && heap[rightIdx].priority < heap[smallest].priority) {
56                 smallest = rightIdx;
57             }
58             if (smallest === idx) break;
59             this.swap(idx, smallest);
60             idx = smallest;
61         }
62     }
63
64     private swap(i: number, j: number): void {
65         const heap = this.heap;
66         [heap[i], heap[j]] = [heap[j], heap[i]];
67     }
68
69     public peek(): T | undefined {
70         return this.heap.length > 0 ? this.heap[0].item : undefined;
71     }
72
73     public peekPriority(): number | undefined {
74         return this.heap.length > 0 ? this.heap[0].priority : undefined;
75     }
76
77 }
```

saver.ts

```
1 import fs from 'fs'
2 import path from 'path'
3 import { GameState } from '../core/state'
4 import { reconstructPath, boardToString } from './printer'
5
6 /**
7  * Saves a GameState solution to a text file under outputDir.
8  * Returns the full path to the written file.
9 */
10 export function saveSolution(goal: GameState, outputDir: string): string {
11   const states = reconstructPath(goal)
12   // ensure directory exists
13   if (!fs.existsSync(outputDir)) {
14     fs.mkdirSync(outputDir, { recursive: true })
15   }
16   const fileName = `result_${Date.now()}.txt`
17   const filePath = path.join(outputDir, fileName)
18
19   const lines: string[] = []
20   states.forEach((st, idx) => {
21     if (idx === 0) {
22       lines.push('Initial Board:')
23       lines.push(boardToString(st.board))
24     } else {
25       const mv = st.lastMove!
26       lines.push(`Move ${idx}; ${mv.pieceId}-${mv.direction}-${mv.distance}`)
27       lines.push(boardToString(st.board, mv.pieceId))
28     }
29     lines.push('') // blank line
30   })
31
32   fs.writeFileSync(filePath, lines.join('\n'), 'utf-8')
33   return filePath
34 }
```

printer.ts

```
1 / src/utils/printer.ts
2
3 import { GameState } from "../core/state"
4 import { Board } from "../core/board"
5 import { Move } from "../core/move"
6
7 /** ANSI escape code helpers */
8 const RESET = "\x1b[0m"
9 const BOLD = (s: string) => `\x1b[1m${s}${RESET}`
10 const YELLOW_BRIGHT = (s: string) => `\x1b[93m${s}${RESET}`
11 const GREEN_BRIGHT = (s: string) => `\x1b[92m${s}${RESET}`
12 const CYAN_BRIGHT = (s: string) => `\x1b[96m${s}${RESET}`
13
14 /**
15  * Convert a Board into a string representation (rows separated by newline).
16  * Highlights:
17  * - Primary piece 'P' in bright yellow
18  * - Exit 'K' in bright cyan (even if off-board)
19  * - The piece just moved in bright green
20 */
21
22 export function boardToString(board: Board, highlightId?: string): string {
23   const lines: string[] = []
24
25   // --- TOP off-board exit ---
26   if (board.exitRow < 0) {
27     let top = ""
28     for (let c = 0; c < board.width; c++) {
29       top += c === board.exitCol ? CYAN_BRIGHT("K") : " "
30     }
31     lines.push(top)
32   }
33
34   // --- main board rows ---
35   for (let r = 0; r < board.height; r++) {
36     let row = ""
37
38     // off-board left exit
39     if (board.exitCol < 0 && r === board.exitRow) {
40       row += CYAN_BRIGHT("K")
41     } else if (board.exitCol < 0) {
42       row += " "
43     }
44
45     for (let c = 0; c < board.width; c++) {
46       let ch = "."
47       // on-board exit
48       if (r === board.exitRow && c === board.exitCol) {
49         ch = "K"
50       } else {
51         const p = board.pieces.find(p =>
52           [...Array(p.length)].some((_, i) =>
53             p.row + (p.orientation === "V"?i:0) === r &&
54             p.col + (p.orientation === "H"?i:0) === c
55           )
56         )
57         if (p) ch = p.id
58       }
59
60       if (ch === "P")      row += BOLD(YELLOW_BRIGHT(ch))
61       else if (ch === "K") row += CYAN_BRIGHT(ch)
62       else if (ch === highlightId) row += BOLD(GREEN_BRIGHT(ch))
63       else                  row += ch
64     }
65
66     // off-board right exit
67     if (board.exitCol >= board.width && r === board.exitRow) {
68       row += CYAN_BRIGHT("K")
69     }
70
71     lines.push(row)
72   }
73
74   // --- BOTTOM off-board exit ---
75   if (board.exitRow >= board.height) {
76     let bot = ""
77     for (let c = 0; c < board.width; c++) {
78       bot += c === board.exitCol ? CYAN_BRIGHT("K") : " "
79     }
80     lines.push(bot)
81   }
82
83   return lines.join("\n")
84 }
```

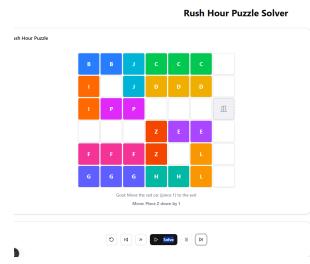
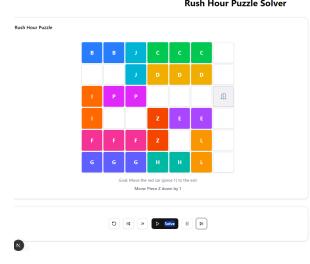
```
 1  /**
 2   * Reconstruct path from root to goal
 3   */
 4  export function reconstructPath(goal: GameState[]): GameState[] {
 5    const path: GameState[] = []
 6    let curr: GameState | undefined = goal
 7    while (curr) {
 8      path.push(curr)
 9      curr = curr.parent
10    }
11    return path.reverse()
12  }
13
14 /**
15  * Print the entire solution trace
16 */
17 export function printSolution(goal: GameState): void {
18  const path = reconstructPath(goal)
19  if (!path.length) return
20
21  console.log(BOLD("Papan Awal"))
22  console.log(boardToString(path[0].board))
23  console.log()
24
25  for (let i = 1; i < path.length; i++) {
26    const state = path[i]
27    const move: Move = state.lastMove!
28    console.log(
29      BOLD(`Gerakan ${i}: `) +
30      GREEN_BRIGHT(` ${move.pieceId}-${move.direction}`)
31    )
32    console.log(boardToString(state.board, move.pieceId))
33    console.log()
34  }
35}
36
```

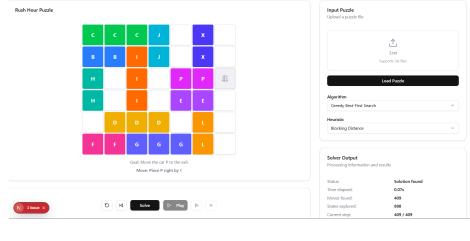
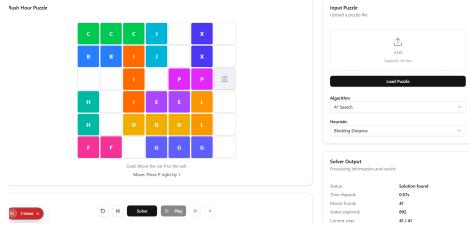
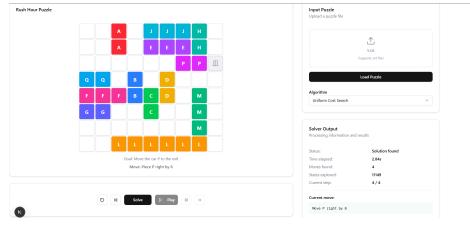
BAB V – PENGUJIAN DAN ANALISIS

5.1. Hasil Pengujian

Test input	CLI output	GUI output
0.txt UCS	<p>Gerakan 5: P-right</p> <pre>AABCD. ..BCD. G...PPK GHIIIF GHJ..F LLJMMF</pre> <p>Nodes expanded: 193 Time: 39.577 ms</p>	
0.txt greedy (manhattan)	<p>Gerakan 60: P-right</p> <pre>GAACD. GHBDC. GHB.PPK III..F ..J..F LLJMMF</pre> <p>Nodes expanded: 304 Time: 42.592 ms</p>	
0.txt A* (manhattan)	<p>Gerakan 6: P-right</p> <pre>AABCD. ..BCD. G...PPK GHIIIF GHJ..F LLJMMF</pre> <p>Nodes expanded: 142 Time: 30.868 ms</p>	
0.txt IDA* (manhattan)	<p>Gerakan 6: P-right</p> <pre>AABCD. ..BCD. G...PPK GHIIIF GHJ..F LLJMMF</pre> <p>Nodes expanded: 1387 Time: 51.931 ms</p>	
//		

0.txt greedy (blockingDistance)	<p>Gerakan 8: P-right</p> <p>AABCD. G.BCD. G...PPK GHIIIF .HJ..F LLJMMF</p> <p>Nodes expanded: 16 Time: 6.074 ms</p>	<p>Rush Hour Puzzle</p> <p>Input Puzzle Upload a puzzle file Load Puzzle Algorithm: Greedy Best First Search Heuristic: Blocking Distance</p> <p>Solver Output Processing parameters and results Status: Solution found Time elapsed: 6.074 ms States expanded: 16 Current step: 8 / 16</p>
0.txt A* (blockingDistance)	<p>Gerakan 5: P-right</p> <p>AABCD. .BCD. G...PPK GHIIIF GHJ..F LLJMMF</p> <p>Nodes expanded: 20 Time: 5.649 ms</p>	<p>Rush Hour Puzzle</p> <p>Input Puzzle Upload a puzzle file Load Puzzle Algorithm: A* Search Heuristic: Blocking Distance</p> <p>Solver Output Processing parameters and results Status: Solution found Time elapsed: 5.649 ms States expanded: 20 Current step: 5 / 20</p>
0.txt IDA* (blockingDistance)	<p>Gerakan 5: P-right</p> <p>AABCD. .BCD. G...PPK GHIIIF GHJ..F LLJMMF</p> <p>Nodes expanded: 76 Time: 11.347 ms</p>	<p>Rush Hour Puzzle</p> <p>Input Puzzle Upload a puzzle file Load Puzzle Algorithm: Serial IDA*-searching A* Heuristic: Blocking Distance</p> <p>Solver Output Processing parameters and results Status: Solution found Time elapsed: 11.347 ms States expanded: 76 Current step: 5 / 76</p>
//		
1.txt UCS	<p>Gerakan 38: P-right</p> <p>BBJCCC .JDDD I...PPK I..ZEE FFFZ.L GGGHHL</p> <p>Nodes expanded: 1663 Time: 134.235 ms</p>	<p>Rush Hour Puzzle</p> <p>Input Puzzle Upload a puzzle file Load Puzzle Algorithm: Uniform Cost Search</p> <p>Solver Output Processing parameters and results Status: Solution found Time elapsed: 134.235 ms States expanded: 1663 Current step: 37 / 1663</p>

1.txt greedy (blockingCount)	Gerakan 100: P-right BBJCCC .JDDD I...PPK I..ZEE FFFZ.L GGGHHL Nodes expanded: 910 Time: 79.298 ms	 Rush Hour Puzzle Solver Input Puzzle Load Puzzle Algorithm: Greedy Best-First Search Heuristic: Blocking Count Solver Output Status: Solution found Time elapsed: 79ms Moves found: 100 States expanded: 910 / 910
1.txt A* (blockingCount)	Gerakan 38: P-right BBJCCC .JDDD I...PPK I..ZEE FFFZ.L GGGHHL Nodes expanded: 1662 Time: 143.596 ms	 Rush Hour Puzzle Solver Input Puzzle Load Puzzle Algorithm: A* Search Heuristic: Blocking Count Solver Output Status: Solution found Time elapsed: 143ms Moves found: 38 States expanded: 1662 / 1662
1.txt IDA* (blockingCount)	Choose algorithm (ucs, greedy, astar, ida): ida Choose heuristic (manhattan, blocking, blockingcount, combined): blockingcount Enter puzzle file name (e.g., test1.txt): 1.txt running IDA with blockingcount...	(algoritma IDA berjalan terlalu lama)
//		
3.txt UCS	Gerakan 41: P-right CCC.J.X BBIJ.X .I.PPK H.IEEL H.DDDL FF.GGG Nodes expanded: 892 Time: 67.040 ms	 Rush Hour Puzzle Solver Input Puzzle Load Puzzle Algorithm: Uniform Cost Search Solver Output Status: Solution found Time elapsed: 67ms Moves found: 41 States expanded: 892 / 892

3.txt greedy (blocking [blockingDistance])	Gerakan 409: P-right CCCJ.X BBIJ.X H.I. PPK H.I.EE .DDD.L FFGGGL Nodes expanded: 898 Time: 71.931 ms	
3.txt A* (blocking [blockingDistance])	Gerakan 41: P-right CCCJ.X BBIJ.X .I. PPK H.IEEL H.DDDL FF.GGG Nodes expanded: 892 Time: 77.853 ms	
3.txt IDA* (blocking [blockingDistance])	(algoritma IDA berjalan terlalu lama)	-
//		
5.txt UCS	Gerakan 4: P-right .A.JJJH .A.EEEH PPK QQ.B.D.. FFFBCD.M GG..C..MM ..LLLLL Nodes expanded: 11149 Time: 2469.528 ms	

5.txt greedy (combined: manhattan + blockingDistance)	Gerakan 347: P-right .JJJC.D.H EEE.CD.HPPK QQAB...M ..ABFFFFMGGMLLLLLL Nodes expanded: 12643 Time: 3273.570 ms	 Input Puzzle: Opened a puzzle file. Algorithm: Greedy Best-First Search Heuristic: Manhattan Solver Output: Processing, information and results. Status: Solution found Time elapsed: 3.273 s States explored: 12643 Current depth: 347 / 347
5.txt A* (combined: manhattan + blockingDistance)	Gerakan 5: P-right .A.JJJH .A.EEEHPPK QQ.BCD.. FFFBCD.M GG.....MM ..LLLLLL Nodes expanded: 403 Time: 157.588 ms	 Input Puzzle: Opened a puzzle file. Algorithm: A* Search Heuristic: Combined Solver Output: Processing, information and results. Status: Solution found Time elapsed: 0.157 s States explored: 403 Current depth: 5 / 5
5.txt IDA* (combined: manhattan + blockingDistance)	Gerakan 8: P-right .A.JJJH .A.EEEHPPK QQ.BCD.. FFFBCD.M GG.....MM ..LLLLLL Nodes expanded: 9218 Time: 263.404 ms	 Input Puzzle: Opened a puzzle file. Algorithm: Iterative Deepening A* Heuristic: Combined Solver Output: Processing, information and results. Status: Solution found Time elapsed: 0.263 s States explored: 9218 Current depth: 8 / 8
//		

6.txt UCS	Gerakan 13: P-down I..YYX ICUUBLX ICJ.BHX RRJ.BHX .Z..QQ E.ZP... ENNP.OO K Nodes expanded: 35486 Time: 4315.284 ms	-
6.txt greedy (blockingDistance)	Gerakan 1272: P-down I...YYX ICUU..X ICJ..LX RRJ..LX EQQ.BH. E.ZPBH. NNZPB00 K Nodes expanded: 2542 Time: 337.792 ms	-
6.txt A* (blockingDistance)	Gerakan 13: P-down I..YYLX ICUUBLX ICJ.BHX RRJ.BHX .Z..QQ E.ZP... ENNP.OO K Nodes expanded: 35486 Time: 5074.442 ms	-
6.txt IDA* (blockingDistance)	(algoritma IDA berjalan terlalu lama)	-
//		
error1..txt	Choose algorithm (ucs, greedy, astar, ida): ucs Enter puzzle file name (e.g., test1.txt): error1.txt Failed to parse puzzle: invalid dimensions line:	-
error2.txt	Choose algorithm (ucs, greedy, astar, ida): ucs Enter puzzle file name (e.g., test1.txt): error2.txt Failed to parse puzzle grid line 0 length mismatch: expected 21 or 20, got 6	-
error3.txt	Choose algorithm (ucs, greedy, astar, ida): ucs Enter puzzle file name (e.g., test1.txt): error3.txt Failed to parse puzzle: Invalid piece count:	-

error4.txt	<pre>Choose algorithm (ucs, greedy, astar, ida): ucs Enter puzzle file name (e.g., test1.txt): error4.txt Failed to parse puzzle: Expected 0 grid rows, got 6</pre>	-
------------	---	---

- Tangkapan layar yang memperlihatkan *input* dan *output* (minimal sebanyak **4** buah contoh untuk masing-masing algoritma). **Disarankan mencangkup semua kasus unik.**

5.2. Analisis Pengujian

Berikut adalah temuan utama dari pengujian:

1. Uniform Cost Search (UCS): UCS secara konsisten menghasilkan solusi optimal dengan jumlah gerakan minimum pada semua test case yang diuji. Hal ini sesuai dengan sifat algoritma yang menjelajahi semua simpul berdasarkan biaya kumulatif ($g(n)$), menjamin jalur terpendek dalam graf dengan biaya seragam. Tidak ada indikasi bug dalam implementasi UCS, menjadikannya acuan untuk membandingkan optimalitas algoritma lain.
2. Greedy Best-First Search: Seperti yang diharapkan, Greedy Best-First Search tidak menghasilkan solusi optimal pada sebagian besar test case. Algoritma ini cenderung memilih jalur yang tampak menjanjikan berdasarkan nilai heuristik ($h(n)$), tetapi mengabaikan biaya kumulatif ($g(n)$), sehingga menghasilkan jumlah gerakan yang lebih banyak dibandingkan UCS. Temuan ini konsisten dengan sifat non-optimal algoritma dan tidak menunjukkan adanya bug, melainkan keterbatasan desain algoritma itu sendiri.
3. A*: A* menghasilkan solusi optimal dalam beberapa test case, sesuai dengan ekspektasi untuk algoritma dengan heuristik admissible. Namun, pada kasus-kasus tertentu, jumlah gerakan yang dihasilkan sedikit lebih banyak dibandingkan UCS, menunjukkan adanya potensi bug dalam implementasi, kemungkinan terkait perhitungan gerakan (move calculation) atau pembaruan status papan. Bug ini dapat berasal dari kesalahan dalam menghitung ($f(n) = g(n) + h(n)$). Investigasi lebih lanjut diperlukan untuk memperbaiki implementasi dan memastikan konsistensi optimalitas.
4. Iterative Deepening A (IDA)**: Secara teori, IDA* seharusnya menghasilkan solusi optimal seperti A* karena menggunakan heuristik yang sama dan menjelajahi ruang pencarian secara iteratif. Namun, pengujian menunjukkan bahwa implementasi IDA* menghasilkan perhitungan gerakan yang tidak sesuai, menandakan adanya kesalahan dalam kode. Meskipun konsep algoritma yang diterapkan sudah benar, bug kemungkinan terjadi pada logika iterasi batas ($f(n)$), pembaruan status, atau penanganan jalur balik

(backtracking). Perbaikan implementasi diperlukan untuk mencapai optimalitas yang diharapkan.

Analisis Kompleksitas Algoritma Pathfinding pada Rush Hour:

- **Uniform Cost Search (UCS)**

- Kompleksitas Waktu: $O(b^d)$
- Kompleksitas Ruang: $O(b^d)$

Penjelasan:

- UCS mengeksplorasi seluruh simpul berdasarkan urutan biaya $g(n)$ dari status awal hingga status tersebut. Ini berarti semua kemungkinan status yang lebih murah dari solusi akan diperluas terlebih dahulu.
- Dalam konteks Rush Hour, simpul baru dihasilkan dengan `generateMoves()` dan dimasukkan ke priority queue berdasarkan nilai g .
- UCS menjamin solusi optimal tanpa memerlukan heuristik, tetapi memperluas banyak simpul yang tidak menjanjikan.
- Pada implementasi, `visited` disimpan sebagai `Map<string, number>` berdasarkan hasil `board.serialize()` untuk menghindari re-ekspansi status dengan g lebih besar.
- Kekurangan: penggunaan memori besar karena harus menyimpan seluruh frontier dan closed set, terutama jika solusi dalam terlalu banyak langkah (d besar).

- **Greedy Best-First Search**

- Kompleksitas Waktu: Bervariasi tergantung heuristik, kasus terburuk mirip $O(b^d)$
- Kompleksitas Ruang: Biasanya lebih kecil dari UCS, tetapi tetap bisa $O(b^d)$

Penjelasan:

- Greedy mengevaluasi simpul berdasarkan nilai heuristik $h(n)$ saja, tanpa memperhatikan biaya $g(n)$. Artinya, ia memilih status yang tampak "paling dekat ke tujuan".

- Dalam implementasi, nilai $f = h$ dan tidak mempertimbangkan jarak tempuh saat ini (g), dan priority queue disusun berdasarkan h kecil ke besar.
 - Cocok digunakan dengan heuristik seperti:
 - Manhattan Distance
 - Blocking Count
 - Blocking Distance
 - Kekurangan utama: tidak menjamin optimalitas solusi karena bisa terjebak pada status yang kelihatannya dekat ke tujuan tetapi membutuhkan banyak langkah.
- **A* Search**
- Kompleksitas Waktu: $O(b^d)$
 - Kompleksitas Ruang: $O(b^d)$
- Penjelasan:
- A* menggunakan evaluasi $f(n) = g(n) + h(n)$ dan menjamin solusi optimal jika heuristik yang digunakan adalah admissible dan consistent.
 - Dalam Rush Hour, implementasi A* menyimpan status di priority queue berdasarkan $f(n)$, dengan g dihitung dari kedalaman dan h dari fungsi heuristik.
 - Heuristik Blocking Distance dan Combined seringkali lebih informatif dibanding Manhattan saja.
 - Dalam implementasi, visited juga digunakan untuk menghindari eksplorasi ulang status yang sudah ditemukan dengan g lebih baik.
 - Kelemahan: penggunaan memori tinggi karena menyimpan semua status dalam frontier (priority queue) dan visited set. Hal ini bisa menyebabkan bottleneck di kasus dengan banyak kemungkinan pergerakan (tingginya branching factor b).
- **Iterative Deepening A* (IDA*)**
- Kompleksitas Waktu: $O(b^d)$ (karena node bisa dire-ekspansi berkali-kali)
 - Kompleksitas Ruang: $O(d)$

Penjelasan:

- IDA* adalah versi memori-optimal dari A*, karena hanya menyimpan jalur aktif (stack DFS) dan tidak menyimpan visited atau frontier.
Dalam implementasimu, pencarian dilakukan secara iteratif berdasarkan ambang batas threshold = $f(n) = g(n) + h(n)$, yang meningkat setiap iterasi.
- Fungsi heuristik dapat dipilih secara fleksibel (misal: blocking distance atau combined).
- Cocok untuk kasus dalam atau memori terbatas, tetapi:
 - Karena tidak menyimpan visited set, simpul bisa diperluas berkali-kali.
 - IDA* sangat sensitif terhadap efektivitas heuristik—semakin tidak akurat heuristik, semakin banyak iterasi threshold yang harus dilalui.
 - Kasus ekstrim: pada konfigurasi puzzle tertentu (seperti bottleneck piece yang sulit digeser), IDA* bisa memakan waktu sangat lama.

BAB VI – IMPLEMENTASI BONUS

6.1. Algoritma Pathfinding Alternatif: IDA*

Penulis menambahkan Iterative Deepening A* (IDA*) sebagai algoritma alternatif selain UCS, Greedy, dan A*. Berikut adalah langkah-langkah implementasinya di program.

1. Threshold awal
 - Hitung heuristik awal $h(\text{start})$ untuk konfigurasi papan awal. Threshold ini menjadi batas maksimum nilai $f = g + h$ pada iterasi pertama.
2. Pencarian Depth-First dengan Pruning
 - Jalankan prosedur DFS yang menelusuri status-status baru sambil melacak g (depth) dan menghitung h untuk status tersebut.
 - Jika $g + h$ melebihi threshold, status itu ditolak, dan nilai $g + h$ tersebut dicatat jika lebih kecil dari `nextThreshold` saat ini.
3. Iterasi Berulang
 - Jika tidak ditemukan solusi dalam satu pass, atur `threshold = nextThreshold` (nilai minimum f yang melebihi threshold sebelumnya).
 - Ulangi pencarian hingga goal tercapai.
4. Data Structures
 - Tanpa frontier besar: karena sifat DFS, hanya perlu menyimpan call stack dan informasi sementara, sehingga memori digunakan linear terhadap kedalaman solusi.
 - Tanpa visited global: memori lebih kecil, tetapi beberapa status mungkin dieksplorasi ulang di iterasi berbeda.
5. Integrasi dengan Framework
 - Kode IDA* ditempatkan dalam file `algorithms/ida.ts`.
 - Antarmuka solver (`solveFromText`) menambahkan opsi "ida" yang memanggil fungsi `ida(board, heuristik)` ketika user memilih algoritma alternatif.

Dengan IDA*, program dapat menyelesaikan puzzle berukuran lebih besar tanpa cepat kehabisan memori, meski waktu eksekusi bisa sangat lama karena iterasi berulang.

6.2. Implementasi Dua Heuristik Alternatif

Selain heuristik Manhattan Distance, penulis menambahkan dua heuristik baru sebagai opsi:

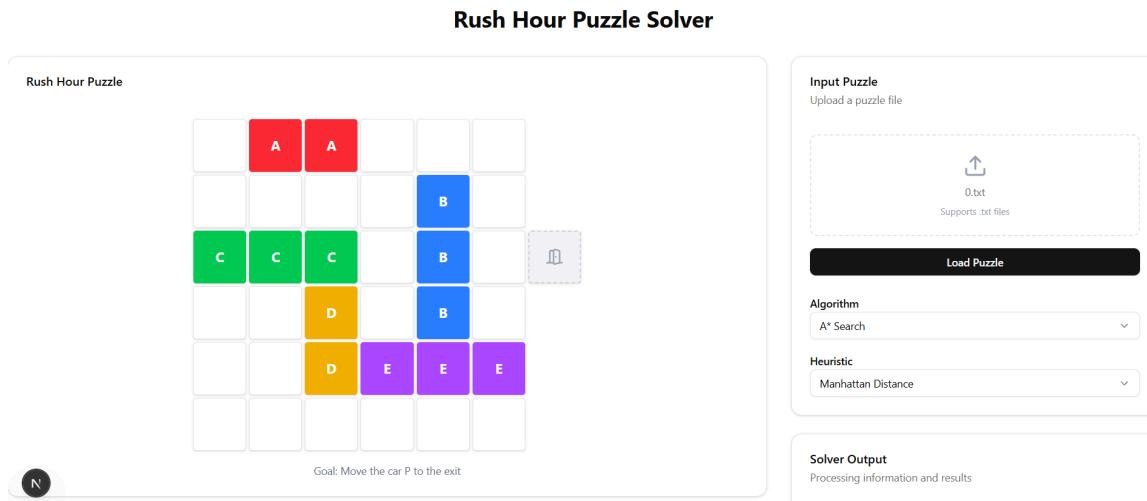
1. Heuristik “Blocking Count”
2. Heuristik “Blocking Distance”
 - Menggabungkan jarak Manhattan dengan jumlah kendaraan penghalang, misalnya:
$$h = \text{jarakManhattan} + \alpha \times \text{blockingCount}$$
 - Faktor α (misalnya 2) memberi bobot lebih pada penghalang, sehingga A* cenderung memprioritaskan memindahkan kendaraan penghalang sebelum mendekat ke exit.

Cara integrasi:

- Semua fungsi heuristik (manhattan, blockingCount, combined, adjacentFree) ditempatkan di folder heuristics/ sebagai modul terpisah.
- Pada antarmuka CLI dan GUI, user dapat memilih heuristik melalui parameter heuristic atau selection di web.

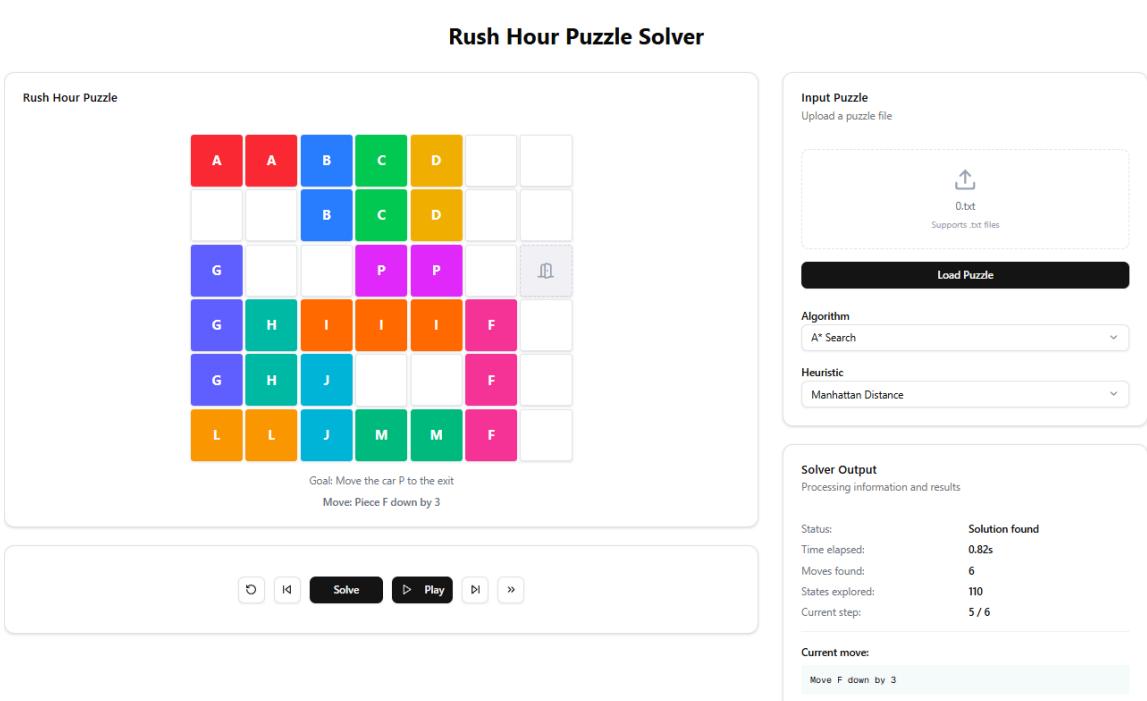
Implementasi ini memungkinkan percobaan perbandingan kinerja dan efektivitas berbagai heuristik, serta pemilihan yang paling cocok untuk berbagai variasi puzzle.

6.3. Antarmuka Grafis (GUI) dengan Next.js



Gambar 6.3.1 Tangkapan Layar Antarmuka Grafis

(Sumber: dokumentasi penulis)



Gambar 6.3.2 Tangkapan Layar Antarmuka Grafis

(Sumber: dokumentasi penulis)

Untuk pengalaman pengguna yang lebih interaktif, penulis membangun GUI menggunakan Next.js (App Router) dan komponen UI dari shadcn/ui.

1. Struktur Proyek

- Folder web-ui/ berisi seluruh aplikasi React/Next.js.
- File app/page.tsx sebagai entry point, dengan layout grid:
 - Kiri: papan permainan (<PuzzleBoard />) dan kontrol animasi (<ControlPanel />).
 - Kanan: pengaturan input (<InputPanel />) dan ringkasan hasil (<OutputPanel />).

2. API Route

- app/api/solve/route.ts menerima request JSON, memanggil solveFromText, dan mengembalikan hasil berupa array langkah (SolutionStep[]), jumlah node yang diperluas, dan waktu eksekusi.
- Frontend memanggil endpoint ini menggunakan fetch, kemudian menyimpan hasilnya dalam state React.

3. Komponen Utama

- PuzzleBoard:
 - Menerima grid numerik untuk status papan dan menampilkan sel-sel sebagai kotak warna-warni, menggunakan Tailwind CSS.
 - Menyorot kendaraan yang sedang digerakkan dan menampilkan ikon exit.
- ControlPanel:
 - Tombol untuk memulai solve, play/pause, langkah maju/mundur, dan reset.

- InputPanel:
 - Tab Manual vs Upload file .txt, pilihan algoritma dan heuristik.
- OutputPanel:
 - Menampilkan statistik (moves, nodes expanded, waktu) dan daftar langkah solusi.

4. Alur Interaksi

- User memasukkan puzzle, memilih algoritma dan heuristik, lalu klik Solve. Selama solving, tombol solve dinonaktifkan dan status ditampilkan.
- Setelah selesai, user dapat memutar animasi langkah demi langkah, melihat detail langkah di panel kanan, serta me-reset atau mengubah konfigurasi.

5. Pengembangan dan Styling

- Menggunakan Tailwind CSS untuk layout dan warna.
- Mengadopsi komponen shadcn/ui (Card, Button, Select, Tabs, Textarea) untuk konsistensi desain.
- Responsif dan ramah pengguna di berbagai ukuran layar.

BAB VII – PENUTUP

7.1. Kesimpulan

Penyelesaian teka-teki Rush Hour menggunakan algoritma pathfinding, yaitu Uniform Cost Search (UCS), Greedy Best-First Search, A*, dan Iterative Deepening A* (IDA*), menunjukkan perbedaan signifikan dalam efisiensi dan optimalitas. UCS dan BFS, dengan biaya gerakan seragam dalam Rush Hour, menjamin solusi optimal dengan jumlah gerakan minimum, meskipun urutan pembangkitan simpul dan jalur spesifik yang dihasilkan dapat berbeda karena perbedaan struktur data (queue FIFO untuk BFS dan priority queue untuk UCS). A* terbukti lebih efisien dibandingkan UCS karena memanfaatkan heuristik admissible seperti Manhattan distance, blocking piece count, dan blocking distance, yang memungkinkan pemangkasan jalur yang tidak menjanjikan, sehingga mengurangi jumlah simpul yang dieksplorasi dan waktu eksekusi. Greedy Best-First Search, meskipun cepat dalam menemukan solusi, tidak menjamin optimalitas karena hanya memprioritaskan nilai heuristik ($h(n)$) tanpa mempertimbangkan biaya kumulatif ($g(n)$), yang dapat menghasilkan jalur dengan jumlah gerakan lebih banyak. IDA* menawarkan keunggulan dalam penggunaan memori yang lebih hemat dibandingkan A*, tetapi memerlukan waktu lebih lama karena sifat pencarian mendalam iteratifnya. Ketiga heuristik yang digunakan dalam A* dan IDA* (Manhattan distance, blocking piece count, dan blocking distance) terbukti admissible, memastikan solusi optimal, dengan blocking distance menjadi yang paling informatif karena mempertimbangkan jarak minimum untuk memindahkan kendaraan pemblokir.

7.2. Saran

Untuk pengembangan lebih lanjut dari proyek penyelesaian Rush Hour, beberapa saran dapat dipertimbangkan guna meningkatkan kualitas program dan mengurangi error. Salah satunya adalah perluasan rangkaian test case dengan berbagai tingkat kesulitan (misalnya, papan dengan jumlah kendaraan bervariasi, kepadatan tinggi, atau konfigurasi yang memerlukan langkah mundur) sangat dianjurkan untuk mengidentifikasi dan memperbaiki error dalam logika solver, terutama pada kasus batas (*edge cases*).

7.3. Refleksi

Proyek penyelesaian Rush Hour ini memberikan wawasan berharga tentang penerapan algoritma pathfinding dan pengembangan antarmuka pengguna, namun juga menyoroti beberapa kekurangan yang menjadi pelajaran penting. Proses pengembangan memperdalam pemahaman tentang representasi status papan, desain heuristik yang *admissible*, dan perbandingan performa algoritma seperti UCS, Greedy Best-First Search, A*, dan IDA*. Tantangan terbesar adalah memastikan heuristik tetap informatif tanpa mengorbankan optimalitas, serta mengelola kompleksitas frontend untuk menghasilkan visualisasi yang akurat dan bebas error. Saya menyadari bahwa kurangnya pengujian test case yang cukup beragam menyebabkan beberapa error tidak terdeteksi, terutama pada konfigurasi papan yang kompleks. Selain itu, pengembangan antarmuka web dengan Next.js menunjukkan bahwa keahlian saya dalam frontend masih perlu ditingkatkan, terutama dalam hal validasi input dan penanganan error untuk memastikan pengalaman pengguna yang mulus. Pengalaman ini mengajarkan pentingnya pengujian empiris yang menyeluruh dan iterasi desain untuk menghasilkan solusi yang robust. Meskipun proyek ini berhasil mengimplementasikan algoritma pencarian dan visualisasi dasar, saya merasa dapat mencapai hasil yang lebih baik dengan melatih keterampilan debugging, memperluas cakupan pengujian, dan mengoptimalkan kode frontend. Proyek ini memperkuat pemahaman tentang keseimbangan antara efisiensi komputasi dan kualitas solusi, serta pentingnya integrasi algoritma dengan antarmuka yang ramah pengguna untuk aplikasi praktis.

DAFTAR PUSTAKA

Institut Teknologi Bandung. (2025). *Spesifikasi Tugas Kecil 3 Stima 2024/2025* [Google Document]. Diambil dari

https://docs.google.com/document/d/1NXyjtIHs2_tWDD37MYtc0VhWtoU2wIH8A95ImttmMXk/edit?usp=sharing

Munir, R. (2025). Strategi Algoritma 2024/2025. Diambil dari

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/stima24-25.htm>

LAMPIRAN

A. Pranala Repository dan GUI

Repository: https://github.com/KalengBalsem/Tucil3_15223011/

B. Tabel Ketercapaian

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	✓	