

Laporan Tugas Kecil 3 IF2211 Strategi Algoritma
Semester II tahun 2024/2025

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding



Disusun oleh:

Refki Alfarizi	(13523002)
Muhammad Jibril Ibrahim	(13523085)

**Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2025**

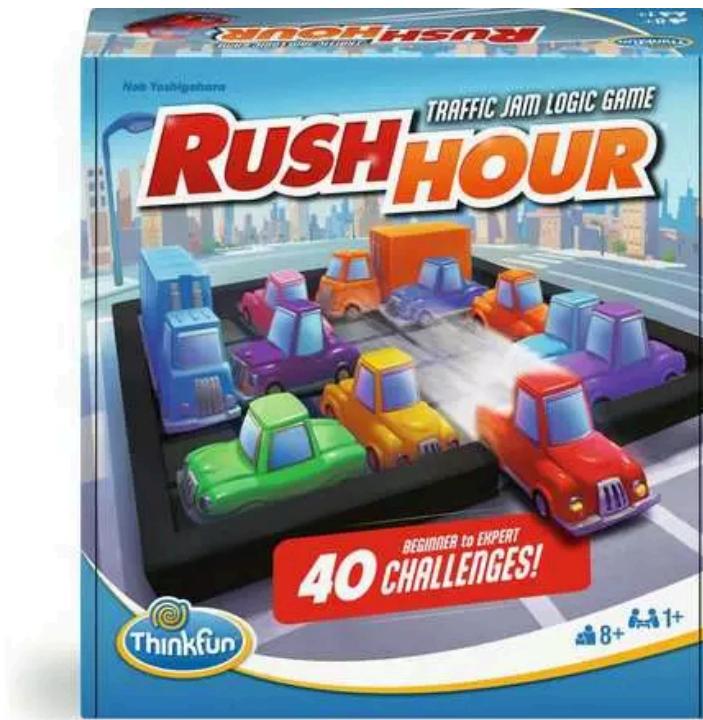
Daftar Isi

Daftar Isi	1
Pendahuluan	4
Ilustrasi kasus :	5
Algoritma Penentuan Rute	8
Uniform Cost Search (UCS)	8
Greedy Best-First Search (GBFS)	9
A* Search	9
Beam Search	10
Heuristik	11
Distance	11
Kompleksitas Komputasi	11
Kelebihan dan Kekurangan	11
Aplikasi pada Rush Hour	11
Mengapa tidak menggunakan Manhattan Distance?	11
Kompleksitas Komputasi	11
Kelebihan dan Kekurangan	11
Aplikasi pada Rush Hour	12
Kompleksitas Komputasi	12
Kelebihan dan Kekurangan	12
Aplikasi pada Rush Hour	12
Analisis Algoritma Penentuan Rute dan Heuristik	13
Uniform Cost Search (UCS)	13
Definisi $g(n)$ dan $f(n)$	13
$UCS \equiv BFS$ di Rush Hour	13
Efisiensi Teoritis vs. A*	13
Greedy Best-First Search (GBFS)	13
Definisi $g(n)$ dan $f(n)$	13
Kelengkapan (Completeness)	13
Optimalitas	13
A* Search	14
Definisi $g(n)$ dan $f(n)$	14
Admissibility Heuristik	14
Completeness & Optimality	14
Efisiensi Teoritis vs. UCS	14
Beam Search	14
Definisi $g(n)$ dan $f(n)$	14
Completeness & Optimality	14
Efisiensi Teoritis vs. UCS	15
Source Code	16
1. Model	16

2. Algoritma	21
3. Heuristic	25
Pengujian Kasus Uji	34
4. 6x6, 12 Kendaraan, terdapat solusi	34
5. A atau B tidak valid	35
6. N tidak Valid	36
7. Baris kosong di papan	36
8. Tinggi papan tidak sama dengan yang diberikan	37
9. Lebar papan tidak sama dengan yang diberikan	37
10. Posisi pintu keluar tidak valid	38
11. Pintu keluar duplikat	38
12. Tidak ada pintu keluar	39
13. Simbol Kendaraan duplikat	39
14. Kendaraan 1x1	40
15. Besar kendaraan tidak valid (tidak 1xN atau Nx1)	40
16. Pintu keluar dengan kendaraan pemain tidak satu baris	41
17. Simbol Kendaraan tidak valid	41
18. Ruang kosong disebelah kiri papan	42
19. Tidak ada satu ruang kosong disebelah kiri papan saat pintu keluar di kiri	42
20. Jumlah kendaraan tidak sama dengan yang diberikan	43
21. Tidak dapat diselesaikan: kendaraan lain menghalangi pintu keluar	43
22. Algoritma A*, Blocking Heuristic, Terdapat Solusi	44
23. Algoritma UCS, Blocking Heuristic, Terdapat Solusi	45
24. Algoritma GBFS, Blocking Heuristic, Terdapat Solusi	46
25. Algoritma Beam, Blocking Heuristic, Terdapat Solusi	47
26. Algoritma A*, Recursive Blocking Heuristic, Terdapat Solusi	48
27. Algoritma UCS, Recursive Blocking Heuristic, Terdapat Solusi	49
28. Algoritma GBFS, Recursive Blocking Heuristic, Terdapat Solusi	50
29. Algoritma Beam, Recursive Blocking Heuristic, Terdapat Solusi	51
30. Algoritma A*, Distance Heuristic, Terdapat Solusi	52
31. Algoritma UCS, Distance Heuristic, Terdapat Solusi	53
32. Algoritma GBFS, Distance Heuristic, Terdapat Solusi	54
33. Algoritma Beam, Distance Heuristic, Tidak ada Solusi	55
34. Tidak Ada Solusi	56
Algoritma Pencarian Rute Alternatif	58
Heuristik Alternatif	58
Graphical User Interface (GUI)	58
Puzzle Loader	59
Pemilihan Algoritma dan Heuristik	59
Animated Solution	60
Performance Stats	61

Save Result	62
Immersive Experience	62
BitSet Board Representation	63
Precomputed Vehicle Masks	63
Priority Queue for Open Set	63
HashMap for Best-Seen Costs	63
Efficient Neighbor Generation	63
Lampiran	64
Tautan Repotori	64
Tabel Pernyataan	64
Daftar Pustaka	65
Akhir Kata	66

Pendahuluan



Gambar 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:

1. **Papan** – Papan merupakan tempat permainan dimainkan.

Papan terdiri atas *cell*, yaitu sebuah *singular point* dari papan. Sebuah *piece* akan menempati *cell-cell* pada papan. Ketika permainan dimulai, semua *piece* telah diletakkan di dalam papan dengan konfigurasi tertentu berupa lokasi *piece* dan *orientasi*, antara *horizontal* atau *vertikal*.

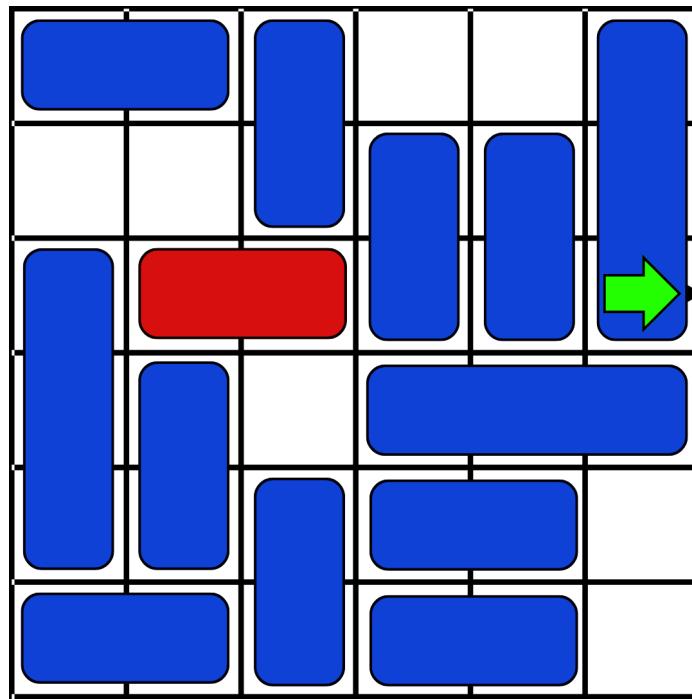
Hanya ***primary piece*** yang dapat digerakkan ***keluar papan melewati pintu keluar***. *piece* yang bukan *primary piece* tidak dapat digerakkan keluar papan. Papan memiliki satu

pintu keluar yang pasti berada di *dinding papan* dan sejajar dengan orientasi *primary piece*.

2. **Piece** – *Piece* adalah sebuah kendaraan di dalam papan. Setiap *piece* memiliki *posisi*, *ukuran*, dan *orientasi*. *Orientasi* sebuah *piece* hanya dapat berupa horizontal atau vertikal—tidak mungkin diagonal. *Piece* dapat memiliki beragam *ukuran*, yaitu jumlah *cell* yang ditempati oleh *piece*. Secara standar, variasi *ukuran* sebuah *piece* adalah *2-piece* (menempati 2 *cell*) atau *3-piece* (menempati 3 *cell*). Suatu *piece* tidak dapat digerakkan melewati/menembus *piece* yang lain.
3. **Primary Piece** – *Primary piece* adalah kendaraan utama yang harus dikeluarkan dari *papan* (biasanya berwarna merah). Hanya boleh terdapat satu *primary piece*.
4. **Pintu Keluar** – *Pintu keluar* adalah tempat *primary piece* dapat digerakkan keluar untuk menyelesaikan permainan
5. **Gerakan** — *Gerakan* yang dimaksudkan adalah pergeseran *piece* di dalam permainan. *Piece* hanya dapat bergerak/bergeser lurus sesuai orientasinya (atas-bawah jika vertikal dan kiri-kanan jika horizontal). Suatu *piece* tidak dapat digerakkan melewati/menembus *piece* yang lain.

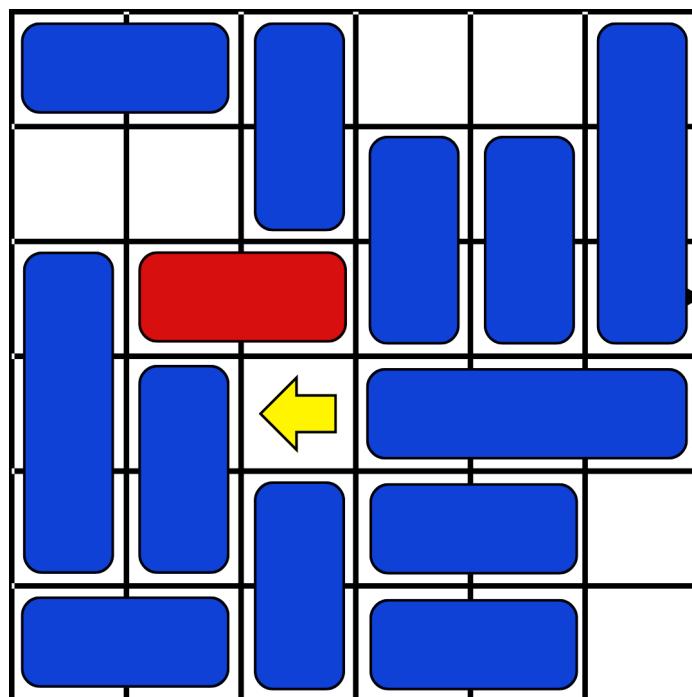
Ilustrasi kasus :

Diberikan sebuah *papan* berukuran 6 x 6 dengan 12 *piece* kendaraan dengan 1 *piece* merupakan *primary piece*. *Piece* ditempatkan pada *papan* dengan posisi dan orientasi sebagai berikut.

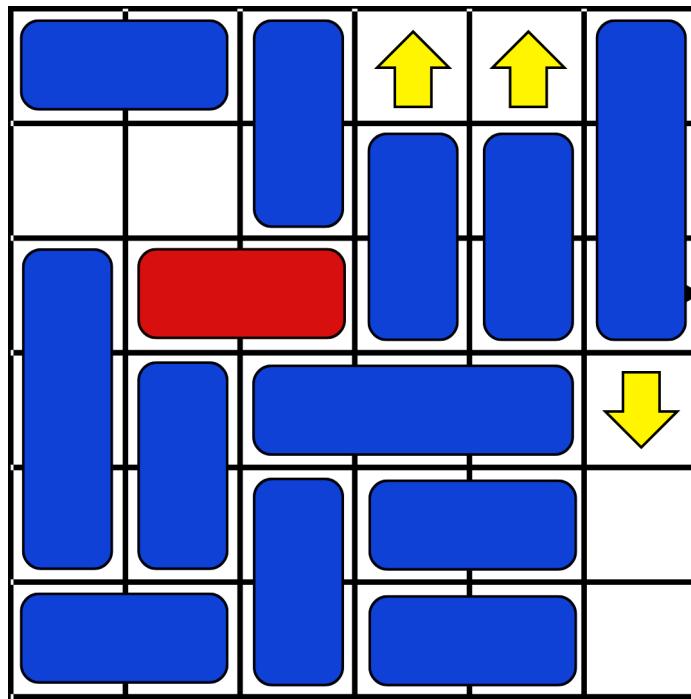


Gambar 2. Awal Permainan Game Rush Hour

Pemain dapat menggeser-geser *piece* (termasuk *primary piece*) untuk membentuk jalan lurus antara *primary piece* dan *pintu keluar*.

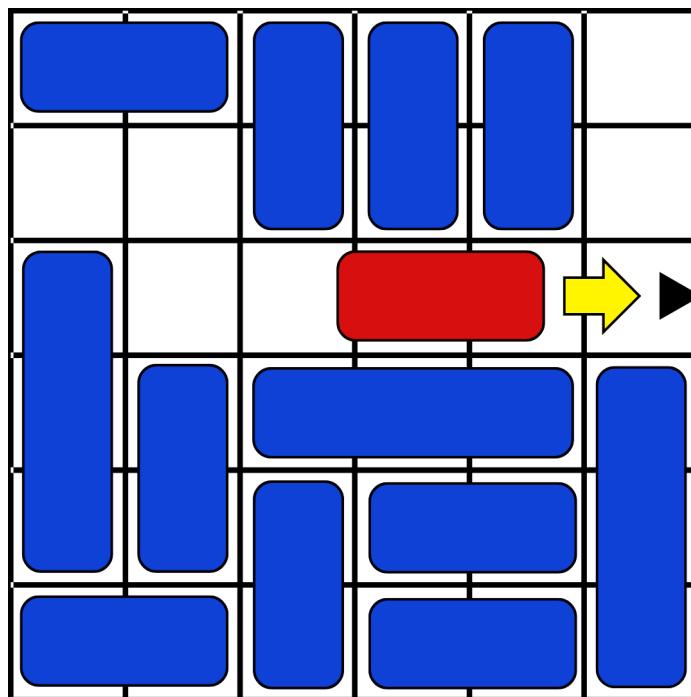


Gambar 3. Gerakan Pertama Game Rush Hour



Gambar 4. Gerakan Kedua Game Rush Hour

Puzzle berikut dinyatakan telah selesai apabila *primary piece* dapat digeser keluar papan melalui *pintu keluar*.



Gambar 5. Pemain Menyelesaikan Permainan

Algoritma Penentuan Rute

Berikut adalah penjelasan algoritma Uniform Cost Search, Greedy Best-First Search, A* Search, dan Beam Search sebagai algoritma penentuan rute yang dipilih untuk menyelesaikan Rush Hour Puzzle di aplikasi yang kami buat. Sebagai catatan, seluruh algoritma yang dipilih diimplementasikan dengan *priority queue* sehingga di penjelasan algoritma selanjutnya hanya menjelaskan dengan penggunaan *priority queue*.

Uniform Cost Search (UCS)

Uniform Cost Search (UCS) adalah algoritma pencarian tak berinformasi (*uninformed search*) yang selalu mengembangkan simpul dengan biaya jalur terakumulasi terkecil terlebih dahulu. Misalkan graf pencarian $G = (V, E)$ memiliki fungsi biaya $c(n, n') \geq 0$ untuk tiap sisi (n, n') . Setiap simpul n menyimpan nilai $g(n)$, yaitu biaya minimum yang diketahui dari simpul awal n_0 hingga n . UCS menggunakan **antrean prioritas** (*priority queue*) yang diurutkan menurut $g(n)$.

Berikut adalah prosedur dari algoritma UCS.

1. Inisialisasi antrean prioritas dengan simpul awal, $g(n_0) = 0$.
2. Ulangi hingga antrean prioritas kosong:
 - a. Hapus dan ambil simpul n dengan nilai $g(n)$ terkecil.
 - b. Jika n memenuhi kondisi tujuan, kembalikan jalur solusi.
 - c. Untuk setiap tetangga n' dari n :
 - i. Hitung biaya baru $g_{baru}(n') = g(n) + c(n, n')$.
 - ii. Jika n' belum pernah dikunjungi atau $g_{baru}(n') < g(n')$, perbarui $g(n') \leftarrow g_{baru}(n')$ dan masukkan ke antrean.

Dengan asumsi faktor percabangan yang berhingga (*finite branching*) dan biaya minimum per langkah $\epsilon > 0$, UCS bersifat **lengkap** (*complete*) dan **optimal** (menjamin menemukan jalur biaya minimum). Kompleksitas waktu dan ruangnya adalah

$$O(b^{(1+C/\epsilon)})$$

di mana b adalah faktor percabangan dan C adalah biaya solusi optimal.

Greedy Best-First Search (GBFS)

Greedy Best-First Search adalah algoritma pencarian rute dengan heuristik (*informed search*) yang hanya menggunakan **fungsi heuristik** $h(n)$ untuk memperkirakan jarak atau biaya tersisa menuju status tujuan. Antrean prioritas diurutkan berdasarkan nilai $h(n)$ terkecil, **tanpa** memperhitungkan biaya jalur sejauh ini $g(n)$.

Berikut adalah prosedur dari algoritma GBFS:

1. Inisialisasi antrean dengan simpul awal $f(n_0) = h(n_0)$.
2. Ulangi hingga antrean kosong:
 - a. Hapus simpul n dengan nilai $h(n)$ terkecil.
 - b. Jika n adalah tujuan, kembalikan solusi.
 - c. Kembangkan n dan masukkan setiap tetangga n' yang belum pernah dikunjungi dengan kunci $h(n')$ (kunci digunakan untuk *HashMap* untuk memeriksa siklus).

Karena hanya menggunakan $h(n)$, GBFS **tidak optimal** (bisa menghasilkan solusi lebih panjang) dan **tidak lengkap** tanpa pemeriksaan siklus (*loop checks*). Namun, jika heuristik yang dipilih informatif, GBFS sering kali lebih cepat dalam menemukan solusi daripada UCS.

A* Search

A* Search memadukan biaya jalur sejauh ini $g(n)$ dan heuristik $h(n)$ dalam fungsi evaluasi $f(n) = g(n) + h(n)$.

Antrean prioritas diurutkan menurut $f(n)$. Algoritmanya mirip UCS, namun memperhitungkan heuristik juga.

Berikut adalah prosedur A*:

1. Inisialisasi antrean prioritas dengan simpul awal, $g(n_0) = 0$, $f(n_0) = h(n_0)$.
2. Ulangi hingga antrean kosong:
 - a. Hapus simpul n dengan nilai $f(n)$ terkecil.
 - b. Jika n adalah tujuan, kembalikan solusi optimal.
 - c. Untuk setiap tetangga n' :

- i. Hitung $g_{baru}(n') = g(n) + c(n, n')$.
- ii. Jika n' baru atau $g_{baru}(n') < g(n')$ atur

$$g(n') \leftarrow g_{baru}(n'), f(n') \leftarrow g(n') + h(n')$$
 dan perbarui antrean.

Jika $h(n)$ **admissible** (tidak melebih-lebihkan biaya sebenarnya) maka A* **optimal**. Jika **consistent** (monoton), simpul tidak perlu diekspansi ulang. Kompleksitas terburuknya tetap eksponensial, tetapi heuristik yang baik dapat memangkas sebagian besar ruang pencarian.

Beam Search

Beam Search, dalam lingkup implementasi ini, adalah varian A* yang membatasi ukuran antrean prioritas (*priority queue*) pada setiap tingkat kedalaman menjadi paling banyak k simpul yang biasa disebut sebagai **beam width**. Alih-alih menyimpan semua simpul dalam antrean prioritas lengkap, pada setiap iterasi Beam Search akan menentukan simpul terbaik untuk disimpan pada antrean prioritas dengan ukuran *beam width*.

Berikut adalah prosedur Beam Search:

1. Inisialisasi antrean prioritas dengan simpul awal, $g(n_0) = 0, f(n_0) = h(n_0)$.
2. Ulangi hingga antrean kosong:
 - a. Hapus simpul n dengan nilai $f(n)$ terkecil.
 - b. Jika n adalah tujuan, kembalikan solusi optimal.
 - c. Untuk setiap tetangga n' :
 - i. Hitung $g_{baru}(n') = g(n) + c(n, n')$.
 - ii. Jika n' baru atau $g_{baru}(n') < g(n')$ atur

$$g(n') \leftarrow g_{baru}(n'), f(n') \leftarrow g(n') + h(n')$$
 dan perbarui antrean.
 - iii. Batasi anteran hanya k (*beam width*) simpul dengan f terendah sebagai *beam* untuk iterasi berikutnya.

Karena membatasi lebar pencarian, Beam Search **mengurangi penggunaan memori** dan jumlah ekspansi, tetapi **tidak menjamin** solusi optimal maupun lengkap. Jika solusi sesungguhnya/optimal terletak di luar k terbaik menurut f , Beam Search akan melewatkannya.

Heuristik

Semua heuristik yang kami gunakan bersifat **admissible**, untuk penjelasan dan argumennya ada pada pembahasan berikut.

Distance

$h(n) = 0$, Jika di depan mobil *primary* tidak terhalang langsung ke pintu keluar

$h(n) = 1$, Jika terhalang oleh mobil

Kompleksitas Komputasi

$$O(1)$$

Kelebihan dan Kekurangan

- Memberi sedikit informasi.
- Terlalu lemah, tidak membedakan tingkat kesulitan blok.

Aplikasi pada Rush Hour

Mempercepat pemilihan status “jalur bersih” tanpa menghitung blocker.

Mengapa tidak menggunakan Manhattan Distance?

Manhattan Distance sering tampak intuitif, tetapi **tidak admissible** dalam konteks Rush Hour ini. Pada implementasi kita, **setiap pemindahan kendaraan, sekecil apa pun atau sejaug apa pun, dihitung sebagai satu langkah**. Sementara itu, Manhattan Distance mengukur jarak dalam *unit sel*, sehingga jika sebuah mobil perlu digeser 3 kotak, heuristik ini akan memberikan nilai 3, padahal sebenarnya hanya diperlukan **1 langkah** untuk memindahkan mobil tersebut sejaug itu.

Dengan demikian $h_{\text{Manhattan}}(n)$ bisa **melebih-lebihkan** biaya asli $h^*(n)$, melanggar definisi *admissible* (yang menuntut $h(n) \leq h^*(n)$). Akibatnya, penggunaan Manhattan Distance bisa memecah satu perpindahan panjang menjadi banyak “langkah” palsu.

Blocking

$h(n) = |\{\text{kendaraan dalam jalur lurus primary ke pintu keluar}\}|$

Kompleksitas Komputasi

$O(m)$, dengan m sebagai banyak baris atau kolom tergantung orientasi dari mobil *primary*

Kelebihan dan Kekurangan

- Heuristik **lebih informatif**, menghitung hambatan langsung.
- Tidak memperhitungkan kompleksitas memindahkan blocker lain.

Aplikasi pada Rush Hour

Memberikan panduan yang jelas seberapa banyak langkah minimal diperlukan untuk membuka jalur.

Recursive Blocking

$$h(n) = \sum_{c \in blockers} (1 + h_{sub}(c))$$

Dengan $h_{sub}(c)$ menghitung jumlah blocker yang menghalangi setiap c .

*Blocker yang sama tidak akan terhitung lebih dari satu kali.

Kompleksitas Komputasi

$O(m + k)$ dengan k total sub-blocker.

Kelebihan dan Kekurangan

- Heuristik **sangat informatif**, menganalisis *chaining blocker*.
- Komputasi yang lebih berat.

Aplikasi pada Rush Hour

Efektif pada puzzle dengan beberapa lapis blocker.

Analisis Algoritma Penentuan Rute dan Heuristik

Uniform Cost Search (UCS)

Definisi $g(n)$ dan $f(n)$

Pada UCS,

$$g(n) = \text{jumlah langkah (biaya) dari keadaan awal ke } n, f(n) = g(n)$$

Antrean prioritas diurutkan semata-mata hanya oleh $g(n)$.

UCS \equiv BFS di Rush Hour

Karena setiap perpindahan kendaraan dihitung satu langkah (biaya = 1), UCS akan mengekspansi semua simpul dalam urutan *level-by-level* sama persis seperti Breadth First Search. Baik urutan ekspansi maupun jalur yang ditemukan (jumlah langkah minimal) identik.

Efisiensi Teoritis vs. A*

UCS tanpa heuristik harus mengeksplorasi setiap simpul dengan $g(n) < C^*$, sehingga kompleksitasnya $O(b^{(1+C/\epsilon)})$. A*, dengan heuristik admissible, hanya mengekspansi simpul dengan $f(n) \leq C^*$, artinya secara teoritis jumlah ekspansi A* **tidak pernah melebihi** UCS, serta biasanya jauh lebih sedikit.

Greedy Best-First Search (GBFS)

Definisi $g(n)$ dan $f(n)$

GBFS tetap menghitung $g(n)$ untuk mencegah siklus, tetapi antrean prioritas hanya menggunakan

$$f(n) = h(n)$$

dengan $h(n)$ adalah estimasi langkah tersisa.

Kelengkapan (*Completeness*)

Karena GBFS dapat “terperangkap” dalam mengekspansi simpul yang heuristiknya rendah tetapi jauh dari solusi, tanpa *loop-checks* eksplorasi bisa tak berujung atau melewatkannya solusi. Implementasi yang kami lakukan telah menambahkan pruning berdasarkan kunjungan sebelumnya, sehingga menjadi **complete** di domain terbatas Rush Hour, namun secara teoretis GBFS murni **tidak lengkap**.

Optimalitas

GBFS **tidak menjamin** jalur minimal. Ia mencari solusi cepat berdasarkan perkiraan h , tetapi karena mengabaikan $g(n)$ dalam urutan, jalur yang ditemukan bisa lebih panjang dari optimum.

A* Search

Definisi $g(n)$ dan $f(n)$

A* menggunakan

$$g(n) = \text{langkah sejauh ini}, f(n) = g(n) + h(n)$$

Antrean prioritas diurutkan menurut $f(n)$.

Admissibility Heuristik

Heuristik dikatakan **admissible** jika selalu

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

dengan $h(n)$ adalah biaya minimum sejati dari n ke tujuan. Heuristik *Blocking* dan *Recursive Blocking* yang kami gunakan mengukur jumlah minimal perpindahan yang benar-benar diperlukan, sehingga **tidak pernah melebih-lebihkan**, memenuhi definisi *admissible* salindia kuliah.

Completeness & Optimality

Dengan heuristik admissible, A* adalah **lengkap** (menemukan solusi jika ada) dan **optimal** (menjamin jumlah langkah paling sedikit). Jika heuristik juga **consistent**, simpul yang sudah diekspansi tidak akan diperluas ulang.

Efisiensi Teoritis vs. UCS

A* hanya mengekspansi simpul-simpul pada area “garis batas” $f(n) \leq C^*$, sehingga jumlah simpul yang dikunjungi **selalu lebih sedikit atau sama** dengan UCS. Pada Rush Hour, uji coba (ada pada bagian berikutnya) menunjukkan pengurangan eksplorasi yang cukup signifikan pada kebanyakan kasus.

Beam Search

Definisi $g(n)$ dan $f(n)$

Sama seperti A*, tetapi antrean prioritas pada setiap tingkat dibatasi **beam width** k simpul terbaik menurut

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

Completeness & Optimality

Pemangkasan antrean prioritas berarti Beam Search **tidak lengkap** (solusi bisa terbuang) dan **tidak optimal** (jalur terbaik mungkin tereliminasi).

Efisiensi Teoritis vs. UCS

Memori turun signifikan menjadi $O(k)$ per level, waktu per level menjadi $O(b * k)$. Asalkan $k \ll b^d$, Beam Search jauh lebih cepat dan hemat memori, tetapi dengan risiko melewatkkan solusi optimal.

Source Code

Berikut adalah kode program utama yang kami implementasikan. Untuk kode program yang lebih lengkap, mohon kunjungi repositori kami pada tautan berikut https://github.com/I0stplains/Tucil3_13523002_13523085.

1. Model

- **Board.java**

```
package tucil_3_stima.model;

import java.util.ArrayList;
import java.util.BitSet;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import tucil_3_stima.strategy.Heuristic;

public class Board {
    private final int rows, cols;
    private final int exitRow, exitCol;
    private final boolean exitHorizontal;
    private final Vehicle[] vehicles;
    private final List<Map<Integer, BitSet>> vehicleMasks;
    private static final int[] DIRS = { -1, +1 };

    public Board(int rows, int cols, int exitRow, int exitCol, boolean
exitHorizontal, Vehicle[] vehicles) {
        this.rows = rows;
        this.cols = cols;
        this.exitRow = exitRow;
        this.exitCol = exitCol;
        this.exitHorizontal = exitHorizontal;
        this.vehicles = vehicles.clone();
        this.vehicleMasks = precomputeMasks();
    }

    private List<Map<Integer, BitSet>> precomputeMasks() {
        List<Map<Integer, BitSet>> list = new ArrayList<>();
        int total = rows * cols;
        for (Vehicle v : vehicles) {
            Map<Integer, BitSet> map = new HashMap<>();
            for (int base = 0; base < total; base++) {
                int r = base / cols;
                int c = base % cols;
                // Check full fit within bounds
                boolean fits;
                if (v.isHorizontal()) {
                    fits = (c + v.length() - 1) < cols;
                } else {
                    fits = (r + v.length() - 1) < rows;
                }
                map.put(base, new BitSet());
                if (fits) {
                    map.get(base).set(c);
                }
            }
            list.add(map);
        }
        return list;
    }
}
```

```

        if (!fits) continue;
        // Build mask
        BitSet mask = new BitSet(rows * cols);
        for (int d = 0; d < v.length(); d++) {
            int rr = v.isHorizontal() ? r : r + d;
            int cc = v.isHorizontal() ? c + d : c;
            mask.set(rr * cols + cc);
        }
        map.put(base, mask);
    }
    list.add(map);
}
return list;
}

public boolean atExit(State state) {
    int[] pos      = state.getPositions();
    int base      = pos[0];
    Vehicle primary = vehicles[0];
    int row       = base / cols;
    int col       = base % cols;

    boolean horiz = primary.isHorizontal();
    int frontRow = horiz ? row : (row + primary.length() - 1);
    int frontCol = horiz ? (col + primary.length() - 1) : col;

    if (horiz) {
        return row == exitRow && (frontCol == exitCol || col == exitCol);
    }
    else {
        return col == exitCol && (frontRow == exitRow || row == exitRow);
    }
}

public BitSet occupancy(State state) {
    BitSet occ = new BitSet(rows * cols);
    int[] pos = state.getPositions();
    for (int i = 0; i < vehicles.length; i++) {
        BitSet m = vehicleMasks.get(i).get(pos[i]);
        if (m != null) occ.or(m);
    }
    return occ;
}

public int heuristic(State state, Heuristic h) {
    return h.evaluate(this, state);
}

public List<State> neighbors(State state, Heuristic h) {
    // worst case each car can slide at most max(rows, cols) squares
    // in one direction
    int maxSlides = Math.max(rows, cols);
    List<State> list = new ArrayList<>(vehicles.length * maxSlides);
}

```

```

        BitSet occ = occupancy(state);
        int[] pos = state.getPositions();

        for (int i = 0; i < vehicles.length; i++) {
            Vehicle v = vehicles[i];
            int oldBase = pos[i];
            Map<Integer, BitSet> masks = vehicleMasks.get(i);
            BitSet orig = masks.get(oldBase);

            // remove this car from the occupancy
            occ.xor(orig);

            // sliding in both directions
            for (int dir : DIRS) {
                int base = oldBase;
                while (true) {
                    int next = v.isHorizontal()
                        ? base + dir
                        : base + dir * cols;

                    BitSet m = masks.get(next);
                    if (m == null || m.intersects(occ)) {
                        break;
                    }

                    // valid move then allocate state n cost
                    State ns = state.copy();
                    int dist = v.isHorizontal()
                        ? next - oldBase
                        : (next - oldBase) / cols;
                    ns.setLastMovement(i, dist);
                    ns.setPosition(i, next);
                    ns.incrementG();
                    ns.setH(heuristic(ns, h));
                    ns.setParent(state);
                    list.add(ns);

                    base = next;
                }
            }

            // restore this car's bits
            occ.or(orig);
        }

        return list;
    }

    public BitSet getVehicleMask(int vehicleIdx, int base) {
        return vehicleMasks.get(vehicleIdx).get(base);
    }

    public int getRows() { return rows; }
    public int getCols() { return cols; }
    public int getExitRow() { return exitRow; }
    public int getExitCol() { return exitCol; }
}

```

```

    public boolean getExitHorizontal() { return exitHorizontal; }
    public Vehicle getVehicle(int i) { return vehicles[i]; }
    public Vehicle[] getVehicles() { return vehicles; }
}

```

- **State.java**

```

// State.java
package tucil_3_stima.model;

import javafx.util.Pair;

import java.util.Arrays;

public class State {
    private final int[] positions;
    private int g, h;
    private State parent;
    private Pair<Integer, Integer> lastMovement;

    public State(int[] initial) {
        this.positions = initial.clone();
        this.lastMovement = null;
    }

    public State copy() {
        State s = new State(this.positions);
        s.g      = this.g;
        s.h      = this.h;
        s.parent = this.parent;
        s.lastMovement = this.lastMovement;
        return s;
    }

    public int[] getPositions() { return positions; }
    public int getG()          { return g; }
    public int getH()          { return h; }
    public int getF()          { return g + h; }
    public State getParent()   { return parent; }
    public Pair<Integer, Integer> getLastMovement() { return
lastMovement; }

    public void setPosition(int idx, int val) { positions[idx] = val; }
    public void incrementG()                  { g++; }
    public void setH(int h)                  { this.h = h; }
    public void setParent(State p)          { this.parent = p; }
    public void setLastMovement(int index, int dist) { lastMovement =
new Pair<Integer, Integer>(index, dist); }

    @Override
    public int hashCode() {
        return Arrays.hashCode(positions);
    }

    @Override

```

```

public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof State)) return false;
    return Arrays.equals(positions, ((State)o).positions);
}
}

```

- **Vehicle.java**

```

package tucil_3_stima.model;

import java.util.BitSet;

public class Vehicle {
    private final boolean horizontal;
    private final int length;
    private final char symbol;

    public Vehicle(boolean horizontal, int length, char symbol) {
        this.horizontal = horizontal;
        this.length = length;
        this.symbol = symbol;
    }

    public boolean isHorizontal() { return horizontal; }
    public int length() { return length; }
    public char getSymbol() { return symbol; }
}

```

- **SearchResult.java**

```

package tucil_3_stima.strategy;

import tucil_3_stima.model.State;

public class SearchResult {
    private final State solution;
    private final int nodesExpanded;
    private final int nodesGenerated;
    private final int maxOpenSize;
    private final long durationNanos;
    private final int solutionDepth;

    public SearchResult(State solution,
                       int nodesExpanded,
                       int nodesGenerated,
                       int maxOpenSize,
                       long durationNanos) {
        this.solution      = solution;
        this.nodesExpanded = nodesExpanded;
        this.nodesGenerated = nodesGenerated;
        this.maxOpenSize   = maxOpenSize;
        this.durationNanos = durationNanos;
    }
}

```

```

        this.solutionDepth = solution != null ? solution.getG() : -1;
    }

    public State getSolution() { return solution; }
    public int getNodesExpanded() { return nodesExpanded; }
    public int getNodesGenerated() { return nodesGenerated; }
    public int getMaxOpenSize() { return maxOpenSize; }
    public long getDurationNanos() { return durationNanos; }
    public double getDurationMillis(){ return durationNanos /
1_000_000.0; }
    public int getSolutionDepth() { return solutionDepth; }
}

```

2. Algoritma

- **SearchStrategy.java**

```

package tucil_3_stima.strategy;

import tucil_3_stima.model.Board;
import tucil_3_stima.model.State;

public interface SearchStrategy {
    SearchResult solve(Board board, State start, Heuristic heuristic);
}

```

- **AbstractSearch.java**

```

// AbstractSearch.java
package tucil_3_stima.strategy;

import tucil_3_stima.model.Board;
import tucil_3_stima.model.State;

import java.util.*;

public abstract class AbstractSearch implements SearchStrategy {
    @Override
    public SearchResult solve(Board board, State start, Heuristic heuristic) {
        int nodesExpanded = 0;
        int nodesGenerated = 0;
        int maxOpenSize = 0;

        long t0 = System.nanoTime();

        PriorityQueue<State> open = new
PriorityQueue<>(Comparator.comparingInt(this::priority));
        Map<State, Integer> best = new HashMap<>();

        start.setH(board.heuristic(start, heuristic));

```

```

        open.add(start);
        best.put(start, 0);
        maxOpenSize = 1;

        State goal = null;
        while (!open.isEmpty()) {
            State cur = open.poll();
            nodesExpanded++;

            if (board.atExit(cur)) {
                goal = cur;
                break;
            }

            for (State nxt : board.neighbors(cur, heuristic)) {
                nodesGenerated++;
                if (shouldAdd(cur, nxt, best)) {
                    best.put(nxt, nxt.getG());
                    open.add(nxt);
                    maxOpenSize = open.size();
                }
            }
        }

        long t1 = System.nanoTime();
        return new SearchResult(goal,
            nodesExpanded,
            nodesGenerated,
            maxOpenSize,
            t1 - t0);
    }

    protected abstract int priority(State s);
    protected abstract boolean shouldAdd(State cur, State nxt,
Map<State, Integer> best);
}

```

- **AStar.java**

```

package tucil_3_stima.strategy;

import tucil_3_stima.model.State;

import java.util.Map;

public class AStar extends AbstractSearch {
    @Override protected int priority(State s) { return s.getF(); }

    @Override
    protected boolean shouldAdd(State cur, State nxt, Map<State,
Integer> best) {
        return nxt.getG() < best.getOrDefault(nxt, Integer.MAX_VALUE);
    }
}

```

- **UCS.java**

```
package tucil_3_stima.strategy;

import tucil_3_stima.model.State;

import java.util.Map;

public class UCS extends AbstractSearch {
    @Override protected int priority(State s) { return s.getG(); }

    @Override
    protected boolean shouldAdd(State cur, State nxt, Map<State,
Integer> best) {
        return nxt.getG() < best.getOrDefault(nxt, Integer.MAX_VALUE);
    }
}
```

- **GBFS.java**

```
package tucil_3_stima.strategy;

import tucil_3_stima.model.State;

import java.util.Map;

public class GBFS extends AbstractSearch {
    @Override protected int priority(State s) { return s.getH(); }

    @Override
    protected boolean shouldAdd(State cur, State nxt, Map<State,
Integer> best) {
        return !best.containsKey(nxt);
    }
}
```

- **BeamSearch.java**

```
// BeamSearch.java
package tucil_3_stima.strategy;

import tucil_3_stima.model.Board;
import tucil_3_stima.model.State;

import java.util.*;

public class BeamSearch implements SearchStrategy {
    private final int beamWidth;

    public BeamSearch(int beamWidth) {
        if (beamWidth < 1) throw new IllegalArgumentException();
        this.beamWidth = beamWidth;
    }
}
```

```

@Override
public SearchResult solve(Board board, State start, Heuristic
heuristic) {
    int nodesExpanded = 0, nodesGenerated = 0, maxOpenSize = 0;
    long t0 = System.nanoTime();

    start.setH(board.heuristic(start, heuristic));
    List<State> beam = List.of(start);
    Set<State> visited = new HashSet<>(beam);
    maxOpenSize = 1;
    State goal = null;

    while (!beam.isEmpty()) {
        // goal check
        for (State s : beam) {
            if (board.atExit(s)) {
                goal = s;
                break;
            }
        }
        if (goal != null) break;

        // expand
        List<State> candidates = new ArrayList<>();
        for (State s : beam) {
            nodesExpanded++;
            for (State nxt : board.neighbors(s, heuristic)) {
                if (visited.add(nxt)) {
                    nodesGenerated++;
                    // incremental or fresh heuristic:
                    nxt.setH(board.heuristic(nxt, heuristic));
                    candidates.add(nxt);
                }
            }
        }
        if (candidates.isEmpty()) break;

        // top-K selection via a fixed-size max-heap
        PriorityQueue<State> topK = new PriorityQueue<>(
            beamWidth,
            Comparator.comparingInt(State::getF).reversed()
        );
        for (State c : candidates) {
            if (topK.size() < beamWidth) {
                topK.add(c);
            } else if (c.getF() < topK.peek().getF()) {
                topK.poll();
                topK.add(c);
            }
        }
        beam = new ArrayList<>(topK);
        maxOpenSize = Math.max(maxOpenSize, beam.size());
    }

    long t1 = System.nanoTime();
}

```

```

        return new SearchResult(goal, nodesExpanded, nodesGenerated,
maxOpenSize, t1 - t0);
    }
}

```

3. Heuristic

- **Heuristic.java**

```

package tucil_3_stima.strategy;

import tucil_3_stima.model.Board;
import tucil_3_stima.model.State;

public interface Heuristic {
    int evaluate(Board board, State state);
}

```

- **BlockingHeuristic.java**

```

package tucil_3_stima.strategy;

import tucil_3_stima.model.Board;
import tucil_3_stima.model.State;
import tucil_3_stima.model.Vehicle;

import java.util.BitSet;

public class BlockingHeuristic implements Heuristic {
    @Override
    public int evaluate(Board board, State state) {
        BitSet occ = board.occupancy(state);
        int[] pos = state.getPositions();
        int base = pos[0];
        Vehicle primary = board.getVehicle(0);
        int count = 0;

        int cols = board.getCols();
        int row = base / cols;
        int col = base % cols;

        int exitRow = board.getExitRow();
        int exitCol = board.getExitCol();

        if (primary.isHorizontal()) {
            if (exitCol > col) {
                for (int x = col + primary.length(); x <= exitCol; x++) {
                    if (occ.get(row * cols + x)) count++;
                }
            } else {
                for (int x = col - 1; x >= exitCol; x--) {
                    if (occ.get(row * cols + x))

```

```

                if (occ.get(row * cols + x)) count++;
            }
        }
    } else {
        if (exitRow > row) {
            for (int y = row + primary.length(); y <= exitRow; y++) {
                if (occ.get(y * cols + col)) count++;
            }
        } else {
            for (int y = row - 1; y >= exitRow; y--) {
                if (occ.get(y * cols + col)) count++;
            }
        }
    }

    return count;
}
}

```

- **DistanceHeuristic.java**

```

package tucil_3_stima.strategy;

import tucil_3_stima.model.Board;
import tucil_3_stima.model.State;

public class DistanceHeuristic implements Heuristic {
    @Override
    public int evaluate(Board board, State state) {
        BlockingHeuristic bh = new BlockingHeuristic();
        int blocking = bh.evaluate(board, state);

        // yes, only this
        // because we count any k displace as one movement
        if(blocking == 0)
            return 1;
        else
            return 0;
    }
}

```

- **RecursiveBlockingHeuristic.java**

```

package tucil_3_stima.strategy;

import tucil_3_stima.model.Board;
import tucil_3_stima.model.State;
import tucil_3_stima.model.Vehicle;

import java.util.*;

public class RecursiveBlockingHeuristic implements Heuristic {

```

```

private Board board;
private Set<Integer> seenVehicles;

@Override
public int evaluate(Board board, State state) {
    this.board = board;
    this.seenVehicles = new HashSet<>();

    if (isAtExit(state)) {
        return 0;
    }

    return calcMinMoves(state);
}

private boolean isAtExit(State state) {
    Vehicle primary = board.getVehicle(0);
    int pos = state.getPositions()[0];
    int cols = board.getCols();
    int row = pos / cols;
    int col = pos % cols;

    boolean horizontal = primary.isHorizontal();
    int exitPos = horizontal ? board.getExitCol() :
board.getExitRow();

    if (horizontal) {
        return col + primary.length() - 1 == exitPos;
    } else {
        return row + primary.length() - 1 == exitPos;
    }
}

private int calcMinMoves(State state) {
    seenVehicles.add(0);

    int moveCount = 1;

    for (Integer blockingVehicle : findBlockers(state)) {
        int frontSpace = calcSpaceNeeded(0, blockingVehicle, 0,
true, state);
        int backSpace = calcSpaceNeeded(0, blockingVehicle, 0,
false, state);

        moveCount += calcObstructionValue(blockingVehicle,
frontSpace, backSpace, state);
    }

    return moveCount;
}

private List<Integer> findBlockers(State state) {
    List<Integer> blockers = new ArrayList<>();

    Vehicle primary = board.getVehicle(0);
    boolean isRedHorizontal = primary.isHorizontal();
}

```

```

        int[] vehiclePositions = state.getPositions();
        int redPos = vehiclePositions[0];
        int cols = board.getCols();
        int redRow = redPos / cols;
        int redCol = redPos % cols;

        int exitPos = isRedHorizontal ? board.getExitCol() :
board.getExitRow();
        int fixedAxis = isRedHorizontal ? redRow : redCol;
        int redLength = primary.length();

        for (int i = 1; i < vehiclePositions.length; i++) {
            Vehicle vehicle = board.getVehicle(i);

            if (isRedHorizontal == vehicle.isHorizontal()) {
                continue;
            }

            int vehPos = vehiclePositions[i];
            int vehRow = vehPos / cols;
            int vehCol = vehPos % cols;

            int crossAxis = isRedHorizontal ? vehCol : vehRow;
            int vehAxisVal = isRedHorizontal ? vehRow : vehCol;

            // skip if vehicle is behind red car's front
            if (isRedHorizontal) {
                if (crossAxis < redCol + redLength) {
                    continue;
                }
            } else {
                if (crossAxis < redRow + redLength) {
                    continue;
                }
            }

            // check if vehicle blocks red car's path
            int vehLength = vehicle.length();
            if (fixedAxis >= vehAxisVal && fixedAxis < vehAxisVal +
vehLength) {
                blockers.add(i);
            }
        }

        return blockers;
    }

    private int calcObstructionValue(int vehicleIdx, int
frontSpaceNeeded, int backSpaceNeeded, State state) {
        seenVehicles.add(vehicleIdx);

        int moveVal = 1;
        int[] vehiclePositions = state.getPositions();

        for (int nextVehicle = 0; nextVehicle < vehiclePositions.length;
nextVehicle++) {

```

```

        if (nextVehicle == vehicleIdx ||  

seenVehicles.contains(nextVehicle)) {  

            continue;  

}  
  

        if (!doVehiclesOverlap(vehicleIdx, nextVehicle, state)) {  

            continue;  

}  
  

        int forwardVal = 0, backwardVal = 0;  
  

        boolean canMoveForward = checkMovement(vehicleIdx,  

nextVehicle, frontSpaceNeeded, true, state);  

        boolean canMoveBackward = checkMovement(vehicleIdx,  

nextVehicle, backSpaceNeeded, false, state);  
  

        int forwardSpaceNeeded = calcSpaceNeeded(vehicleIdx,  

nextVehicle, frontSpaceNeeded, true, state);  

        int backwardSpaceNeeded = calcSpaceNeeded(vehicleIdx,  

nextVehicle, backSpaceNeeded, false, state);  
  

        if (!canMoveForward) {  

            forwardVal = calcObstructionValue(nextVehicle,  

forwardSpaceNeeded, backwardSpaceNeeded, state);  

        } else if (isEdgeBlocking(vehicleIdx, frontSpaceNeeded,  

true, state)) {  

            forwardVal = Integer.MAX_VALUE;  

}  
  

        if (!canMoveBackward) {  

            backwardVal = calcObstructionValue(nextVehicle,  

forwardSpaceNeeded, backwardSpaceNeeded, state);  

        } else if (isEdgeBlocking(vehicleIdx, backSpaceNeeded,  

false, state)) {  

            backwardVal = Integer.MAX_VALUE;  

}  
  

        moveVal += Math.min(forwardVal, backwardVal);  

    }  
  

    return moveVal;
}  
  

private boolean checkMovement(int vehicleIdx, int nextVehicle, int  

spaceNeeded, boolean isForward, State state) {  

    boolean isBlockerBehind = isPositionedBehind(vehicleIdx,  

nextVehicle, state);  
  

    if ((isBlockerBehind && isForward) || (!isBlockerBehind &&  

!isForward)) {  

        return true;  

}  
  

    int availableSpace = getAvailableSpace(vehicleIdx, nextVehicle,  

isForward, state);  

    return availableSpace >= spaceNeeded;
}

```

```

}

private int calcSpaceNeeded(int vehicleIdx, int nextVehicle, int
baseSpaceNeeded, boolean isForward, State state) {
    Vehicle vehicle = board.getVehicle(vehicleIdx);
    Vehicle nextVeh = board.getVehicle(nextVehicle);

    // if same orientation, use diff calculation
    if (vehicle.isHorizontal() == nextVeh.isHorizontal()) {
        int availSpace = getAvailableSpace(vehicleIdx, nextVehicle,
isForward, state);
        return baseSpaceNeeded - availSpace;
    }

    int[] positions = state.getPositions();
    int cols = board.getCols();

    int vehPos = positions[vehicleIdx];
    int nextPos = positions[nextVehicle];

    int vehRow = vehPos / cols;
    int vehCol = vehPos % cols;
    int nextRow = nextPos / cols;
    int nextCol = nextPos % cols;

    int vehFixed = vehicle.isHorizontal() ? vehRow : vehCol;
    int nextVar = nextVeh.isHorizontal() ? nextCol : nextRow;

    if (isForward) {
        return Math.abs(vehFixed - nextVar) + 1;
    }

    int nextLength = nextVeh.length();
    return Math.abs(vehFixed - (nextVar + nextLength));
}

private int getAvailableSpace(int vehicleIdx, int nextVehicle,
boolean isForward, State state) {
    Vehicle vehicle = board.getVehicle(vehicleIdx);
    Vehicle nextVeh = board.getVehicle(nextVehicle);
    int[] positions = state.getPositions();
    int cols = board.getCols();

    int vehPos = positions[vehicleIdx];
    int nextPos = positions[nextVehicle];

    int vehRow = vehPos / cols;
    int vehCol = vehPos % cols;
    int nextRow = nextPos / cols;
    int nextCol = nextPos % cols;

    int vehVar = vehicle.isHorizontal() ? vehCol : vehRow;
    int vehEnd = vehVar + vehicle.length();

    // different orientation logic
    if (vehicle.isHorizontal() != nextVeh.isHorizontal()) {

```

```

        int nextFixed = nextVeh.isHorizontal() ? nextRow : nextCol;

        if (isForward) {
            return Math.abs(vehEnd - nextFixed);
        }

        return Math.abs(vehVar - nextFixed) + 1;
    }

    // same orientation logic
    int nextVar = nextVeh.isHorizontal() ? nextCol : nextRow;

    if (isForward) {
        return Math.abs(vehEnd - nextVar);
    }

    int nextEnd = nextVar + nextVeh.length();
    return Math.abs(vehVar - nextEnd);
}

private boolean isEdgeBlocking(int vehicleIdx, int spaceNeeded,
boolean isForward, State state) {
    Vehicle vehicle = board.getVehicle(vehicleIdx);
    int[] positions = state.getPositions();
    int cols = board.getCols();
    int rows = board.getRows();

    int vehPos = positions[vehicleIdx];
    int vehRow = vehPos / cols;
    int vehCol = vehPos % cols;

    int vehVar = vehicle.isHorizontal() ? vehCol : vehRow;
    int vehEnd = vehVar + vehicle.length();
    int boardSize = vehicle.isHorizontal() ? cols : rows;

    if (isForward && (vehEnd + spaceNeeded > boardSize)) {
        return true;
    }

    if (!isForward && (vehVar - spaceNeeded < 0)) {
        return true;
    }

    return false;
}

private boolean doVehiclesOverlap(int vehicleIdx, int nextVehicle,
State state) {
    Vehicle vehicle = board.getVehicle(vehicleIdx);
    Vehicle nextVeh = board.getVehicle(nextVehicle);
    int[] positions = state.getPositions();
    int cols = board.getCols();

    int vehPos = positions[vehicleIdx];
    int nextPos = positions[nextVehicle];

```

```

        int vehRow = vehPos / cols;
        int vehCol = vehPos % cols;
        int nextRow = nextPos / cols;
        int nextCol = nextPos % cols;

        int vehFixed = vehicle.isHorizontal() ? vehRow : vehCol;
        int nextFixed = nextVeh.isHorizontal() ? nextRow : nextCol;

        // same orientation check - need same fixed coordinate
        if (vehicle.isHorizontal() == nextVeh.isHorizontal()) {
            return vehFixed == nextFixed;
        }

        // different orientation - check overlap
        int nextVar = nextVeh.isHorizontal() ? nextCol : nextRow;
        int nextEnd = nextVar + nextVeh.length();

        return vehFixed >= nextVar && vehFixed < nextEnd;
    }

    private boolean isPositionedBehind(int vehicleIdx, int nextVehicle,
State state) {
    Vehicle vehicle = board.getVehicle(vehicleIdx);
    Vehicle nextVeh = board.getVehicle(nextVehicle);
    int[] positions = state.getPositions();
    int cols = board.getCols();

    int vehPos = positions[vehicleIdx];
    int nextPos = positions[nextVehicle];

    int vehRow = vehPos / cols;
    int vehCol = vehPos % cols;
    int nextRow = nextPos / cols;
    int nextCol = nextPos % cols;

    int vehVar = vehicle.isHorizontal() ? vehCol : vehRow;
    int nextVar = nextVeh.isHorizontal() ? nextCol : nextRow;

    // same orientation check
    if (vehicle.isHorizontal() == nextVeh.isHorizontal()) {
        int nextEnd = nextVar + nextVeh.length();
        return nextEnd <= vehVar;
    }

    // different orientation check
    int vehEnd = vehVar + vehicle.length();
    int nextFixed = nextVeh.isHorizontal() ? nextRow : nextCol;

    return nextFixed < vehEnd;
}
}

```

- **ZeroHeuristic.java**

```
package tucil_3_stima.strategy;
```

```
import tucil_3_stima.model.Board;
import tucil_3_stima.model.State;

public class ZeroHeuristic implements Heuristic {
    @Override
    public int evaluate(Board board, State state) {
        return 0;
    }
}
```

Pengujian Kasus Uji

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

4. 6x6, 12 Kendaraan, terdapat solusi

Input	Output
6 6 12 GBB.L. GHI.LM KGHIPPM CCCZ.M .JZDD EEJFF.	<p>STATS</p> <p>Time (Ms): 80.5106</p> <p>Expanded Node: 327</p> <p>Generated Node: 2394</p> <p>Steps to Finish: 11</p> <p>Reset Prev Play/Stop Next 1x</p>

The screenshot shows a terminal window with the title bar 'solution.txt'. The window contains the following text:

```
Solution Stats
Time : 80.5106
Expanded Node : 327
Generated Node : 2394
Solution Depth : 11

Initial Board
GBB.L.
GHI.LM
GHIPPM
CCCZ.M|
..JZDD
EEJFF.
```

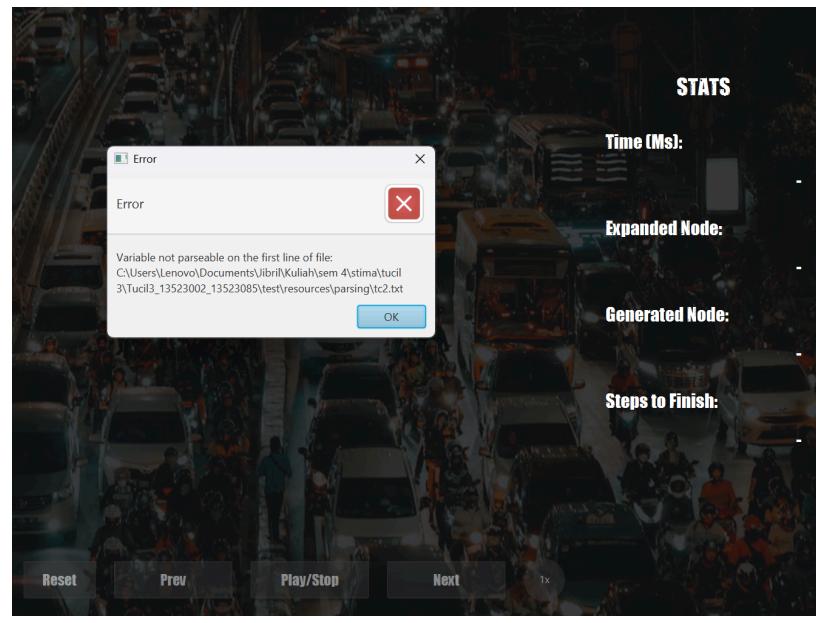
5. A atau B tidak valid

Input	Output
-------	--------

```

6 a
12
GBB.L.
GHI.LM
KGHIPPM
CCCZ.M
..JZDD
EEJFF.

```



6. N tidak Valid

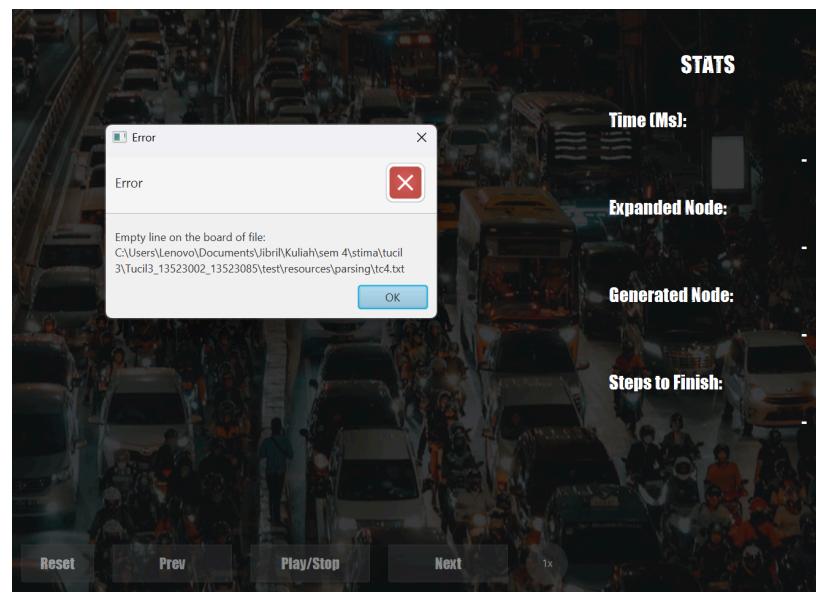
Input	Output
<pre> 6 6 12 adwada GBB.L. GHI.LM KGHIPPM CCCZ.M ..JZDD EEJFF. </pre>	<p>The screenshot shows a software interface with a dark background featuring a complex circuit board or mechanical assembly. In the foreground, there is a modal error dialog box. The dialog has a white background with a red 'X' button at the top right. It contains the text "Error" and "Variable not parseable on the Second line of file: C:\Users\Lenovo\Documents\libril\Kuliah\sem 4\stima\tcucil3\Tcucil3_13523002_13523085\test\resources\parsing\tc3.txt". At the bottom right of the dialog is a blue "OK" button.</p>

7. Baris kosong di papan

Input	Output

6 6
12
GBB.L.
GHI.LM
KGHIPPM

CCCZ.M
.JZDD
EEJFF.



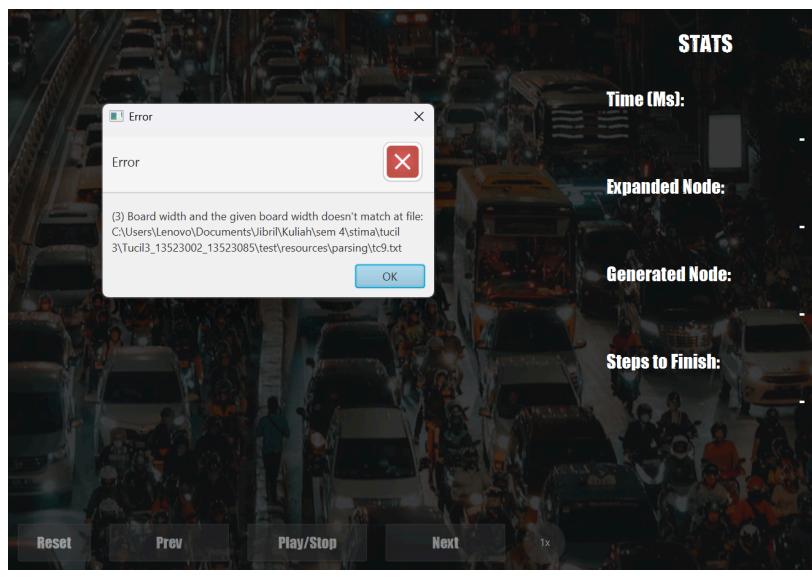
8. Tinggi papan tidak sama dengan yang diberikan

Input	Output
6 6 12 GBB.L. GHI.LM KGHIPPM CCCZ.M .JZDD	<p>The screenshot shows a user interface for a backtracking algorithm. On the left, there is a text input area containing the provided input. On the right, there is a visualization of a board state with pieces and a control panel with buttons like 'Reset', 'Prev', 'Play/Stop', 'Next', and 'Tx'. Overlaid on the interface is an 'Error' dialog box with the message: '(3) Board height and the given board height doesn't match at file: C:\Users\Lenovo\Documents\Jibril\Kuliah\sem 4\stima\tucil3\tucil3_13523002_13523085\test\resources\parsing\tc5.txt'. The 'OK' button of the dialog is highlighted.</p>

9. Lebar papan tidak sama dengan yang diberikan

Input	Output
-------	--------

6 6
 12
 GBB.L.
 GHI.LM
 GHIPPMK
 CCCZ.M
 ..JZDDD
 EEJFF.



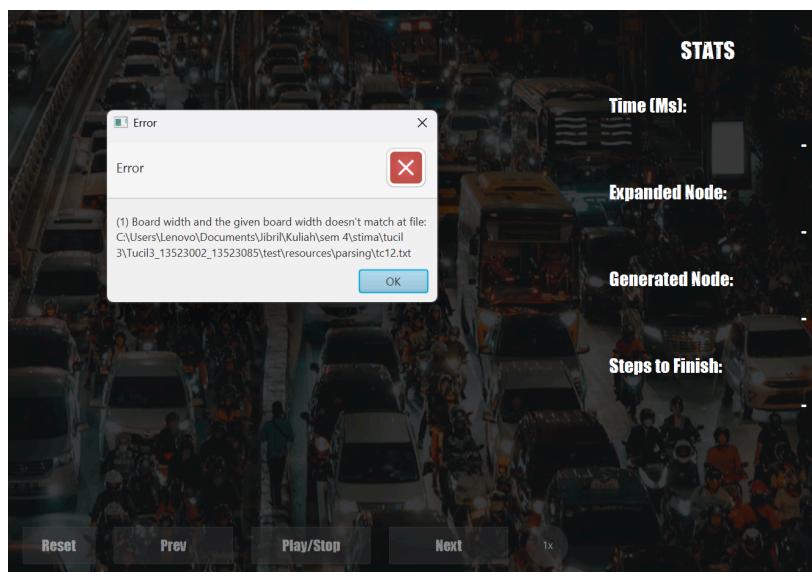
10.Posisi pintu keluar tidak valid

Input	Output
<p> 6 6 12 K GBB.L. GHI.LM GHIPPMK CCCZ.M ..JZDD EEJFF. </p>	

11.Pintu keluar duplikat

Input	Output
-------	--------

6 6
12
GBB.L.
GHI.LM
GHIPPMK
CCCZ.M
.JZDDK
EEJFF.



12. Tidak ada pintu keluar

Input	Output
6 6 12 GBB.L. GHI.LM GHIPPM CCCZ.M .JZDD EEJFF.	<p>The screenshot shows a search interface with a dark background featuring a blurred image of a crowded street scene. On the right side, there are several status indicators: 'STATS', 'Time (Ms):' (with a large red value), 'Expanded Node:', 'Generated Node:', and 'Steps to Finish:'. At the bottom, there are buttons for 'Reset', 'Prev', 'Play/Stop', 'Next', and 'Tx'. A modal dialog box titled 'Error' is displayed in the center, containing the text: 'No exit found on the board of file: C:\Users\Lenovo\Documents\Jibril\Kuliah\sem 4\stima\tc13.txt'. There is an 'OK' button at the bottom right of the dialog.</p>

13. Simbol Kendaraan duplikat

Input	Output
-------	--------

<p>6 6 12 GBB.L. GHI.LM GHIPP.MK CCCZ.M GGJZDD EEJFF.</p>	
---	--

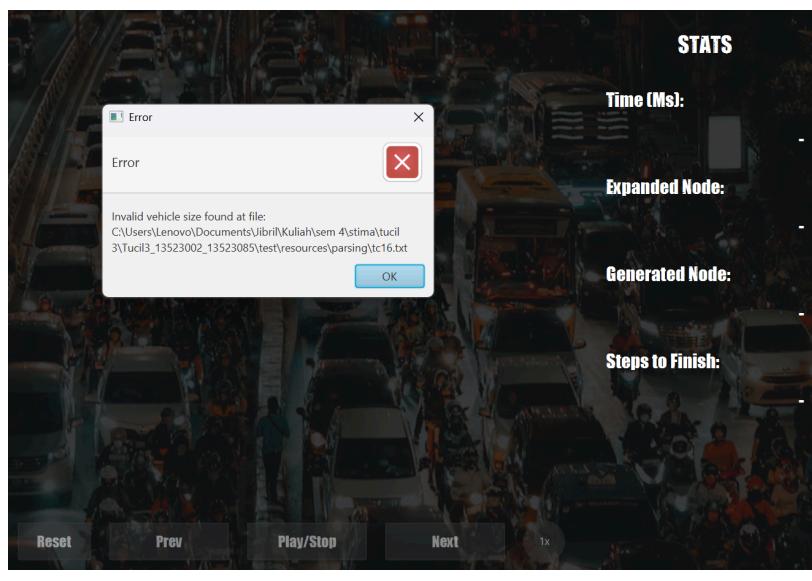
14. Kendaraan 1x1

Input	Output
<p>6 6 12 GBB.L. GHI.LM GHIPP.K CCCZ.. .JZDD EEJFF.</p>	

15. Besar kendaraan tidak valid (tidak 1xN atau Nx1)

Input	Output
-------	--------

6 6
 12
 GBBLL.
 GHILLM
 GHIPPMK
 CCCZ.M
 ..JZDD
 EEJFF.



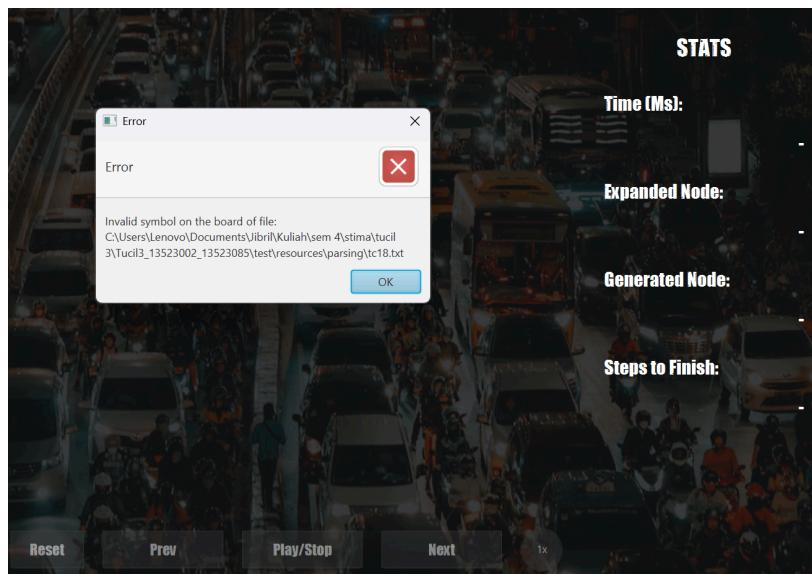
16. Pintu keluar dengan kendaraan pemain tidak satu baris

Input	Output
<p> 6 6 12 GBB.L. GHI.LM GHIPPM CCCZ.M ..JZDD EEJFF. K </p>	

17. Simbol Kendaraan tidak valid

Input	Output

6 6
12
GBB.L.
GHI.LM
GHIPPMK
CCCZ.M
aaJZDD
EEJFF.



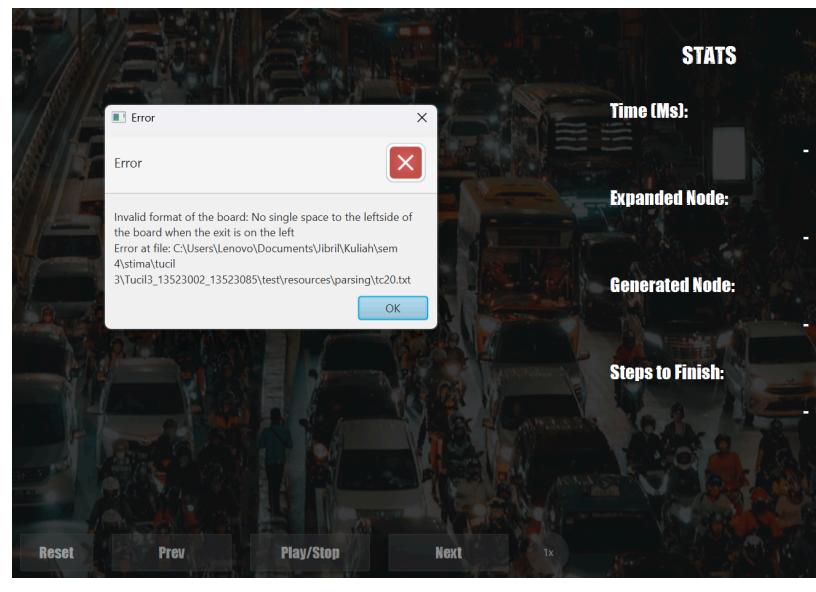
18. Ruang kosong disebelah kiri papan

Input	Output
6 6 12 GBB.L. GHI.LM GHIPPMK CCCZ.M .JZDD EEJFF.	

19. Tidak ada satu ruang kosong disebelah kiri papan saat pintu keluar di kiri

Input	Output
-------	--------

6 6
 12
 GBB.L.
 GHI.LM
 KGHIPPM
 CCCZ.M
 ..JZDD
 EEJFF.



20. Jumlah kendaraan tidak sama dengan yang diberikan

Input	Output
<p> 6 6 11 GBB.L. GHI.LM GHIPPMK CCCZ.M ..JZDD EEJFF. </p>	

21. Tidak dapat diselesaikan: kendaraan lain menghalangi pintu keluar

Input	Output
-------	--------

6 6

12

K

GBB.L.

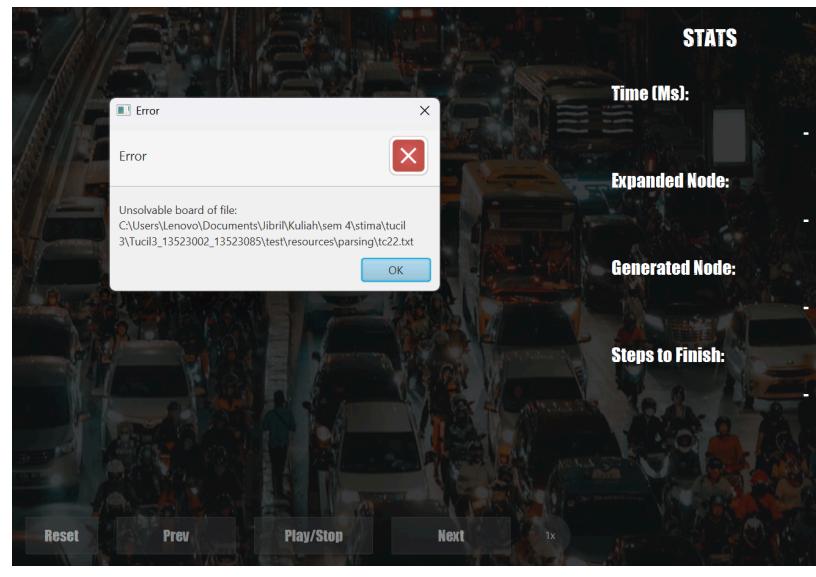
GHI.LM

GHI.PM

CCCZPM

..JZDD

EEJFF.



22. Algoritma A*, Blocking Heuristic, Terdapat Solusi

Input	Output
8 8 23 BBQMSSSD A.QM.OED ACCC.OEV GG...OHV WX.PPPHUK WX..FNNU IXTTFLLU IRRYYJJ	A screenshot of a Rubik's cube solving application. The interface shows a 3x3x3 Rubik's cube with various colored faces (blue, orange, red, green, yellow, purple). A central pink face labeled "Player" is positioned in the middle of the cube. Above the cube, text indicates "Step 0 of 41". To the right, a "STATS" panel displays performance metrics: "Time (Ms): 2059.4273", "Expanded Node: 174290", "Generated Node: 1842147", and "Steps to Finish: 41". At the bottom, control buttons include "Reset", "Prev", "Play/Stop", and "Next".

The screenshot shows a terminal window with a dark background and light-colored text. The title bar says "solution.txt". The menu bar includes "File", "Edit", and "View". The main content displays the following information:

```
Solution Stats
Time : 2340.4397
Expanded Node : 174290
Generated Node : 1842147
Solution Depth : 41

Initial Board
BBQMSSSD
A.QM.OED
ACCC.OEV
GG...OHV
WX.PPPHU
WX..FNUU
IXTTFLLU
IRRRYYJJ
```

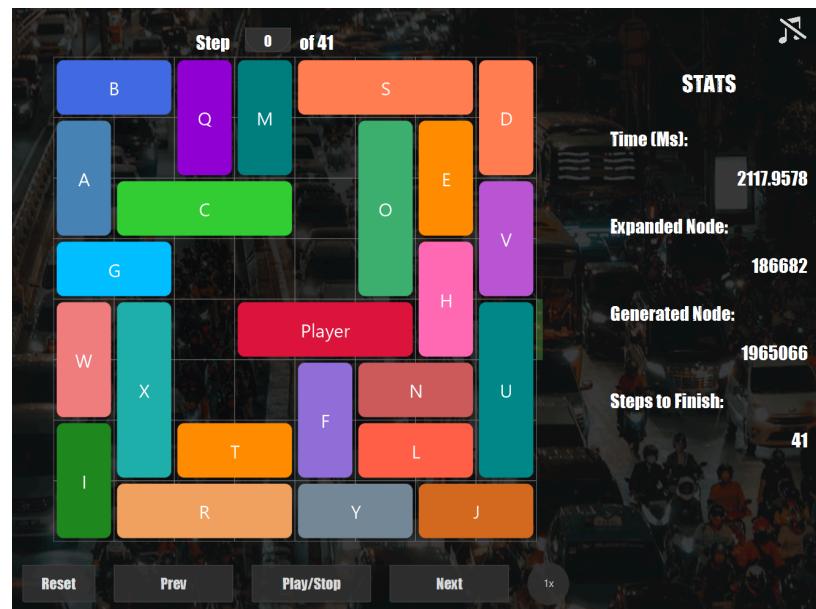
23. Algoritma UCS, Blocking Heuristic, Terdapat Solusi

Input	Output
-------	--------

```

8 8
23
BBQMSSSD
A.QM.OED
ACCC.OEV
GG...OHV
WX.PPPHUK
WX..FNNU
IXTTFLLU
IRRRYYJJ

```



```

solution.txt
File Edit View

Solution Stats
Time : 2080.2891
Expanded Node : 186682
Generated Node : 1965066
Solution Depth : 41

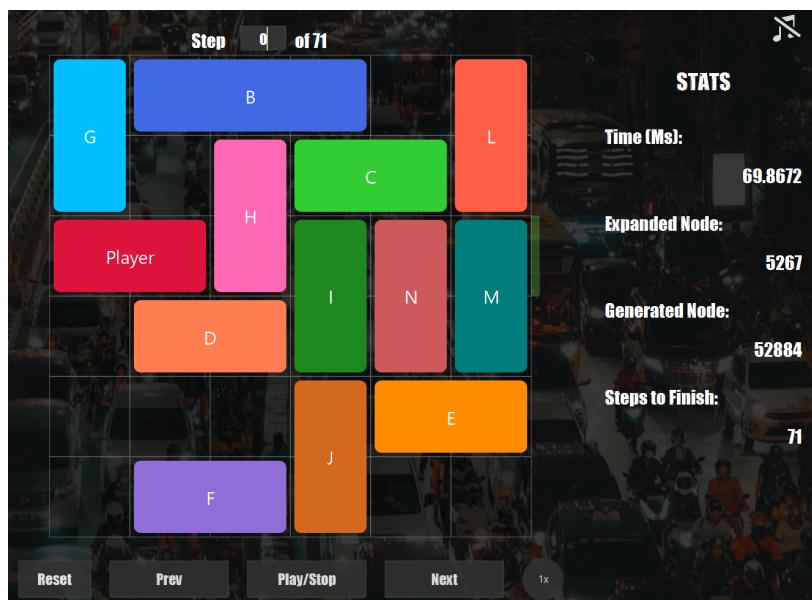
Initial Board
BBQMSSSD
A.QM.OED
ACCC.OEV
GG...OHV
WX.PPPHU
WX..FNNU
IXTTFLLU
IRRRYYJJ

```

24. Algoritma GBFS, Blocking Heuristic, Terdapat Solusi

Input	Output
-------	--------

```
6 6  
12  
GBBB.L  
G.HCCL  
PPHINMK  
.DDINM  
...JEE  
.FFJ..
```



```
solution.txt

File Edit View

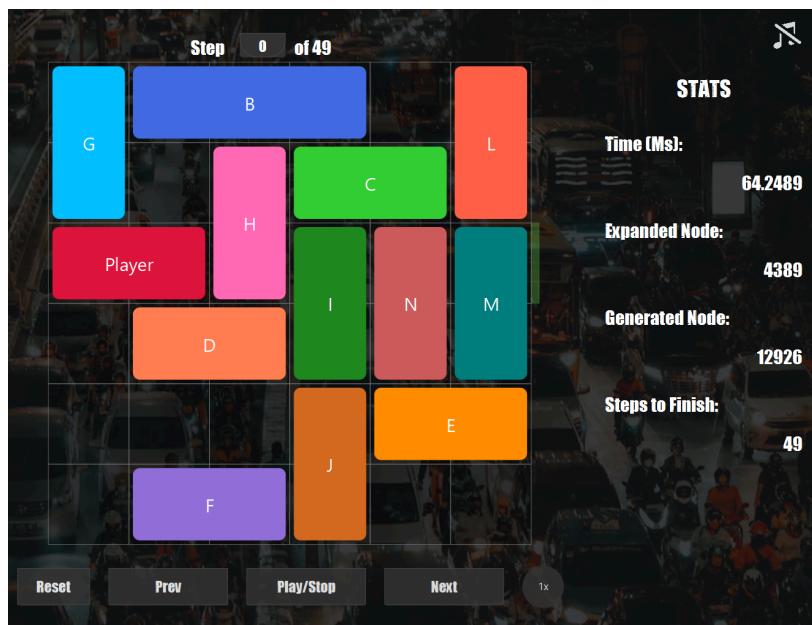
Solution Stats
Time : 69.8672
Expanded Node : 5267
Generated Node : 52884
Solution Depth : 71

Initial Board
GBBB.L
G.HCCL
PPHINMK
.DDI|NM
...JEE
.FFJ..
```

25. Algoritma Beam, Blocking Heuristic, Terdapat Solusi

Input	Output
-------	--------

```
6 6  
12  
GBBB.L  
G.HCCL  
PPHINMK  
.DDINM  
.JEE  
.FFJ..
```



The terminal window shows the contents of the `solution.txt` file:

```
solution.txt
File Edit View

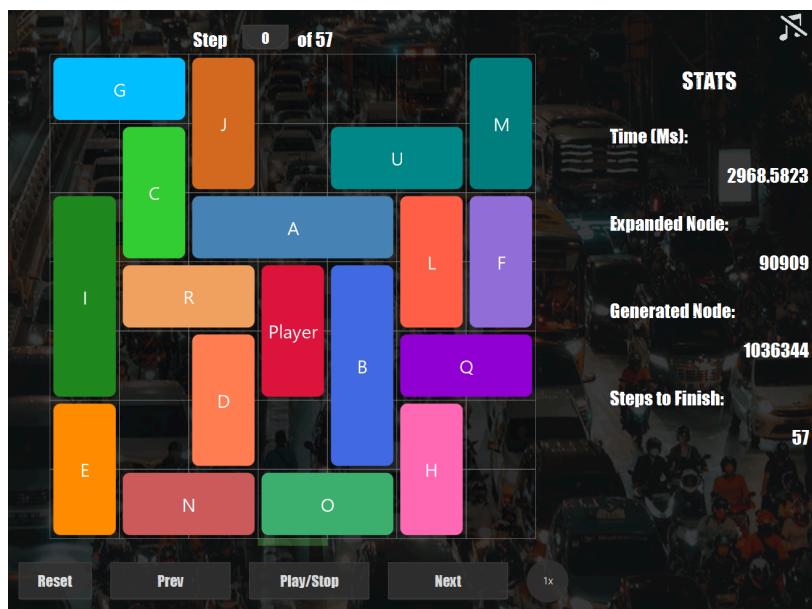
Solution Stats
Time : 64.2489
Expanded Node : 4389
Generated Node : 12926
Solution Depth : 49

Initial Board
GBBB.L
G.HCCL
PPHINMK
.DDI NM
...JEE
.FFJ..
```

26. Algoritma A*, Recursive Blocking Heuristic, Terdapat Solusi

Input	Output
-------	--------

```
7 7  
17  
GGJ...M  
.CJ.UUM  
ICAAALF  
IRRPBLF  
I.DPBQQ  
E.D.BH.  
ENNOOH.  
K
```



```
solution.txt
File Edit View
Solution| Stats
Time : 2968.5823
Expanded Node : 90909
Generated Node : 1036344
Solution Depth : 57

Initial Board
GGJ...M
.CJ.UUM
ICAAALF
IRRPBLF
I.DPBQQ
E.D.BH.
ENNOOH.
```

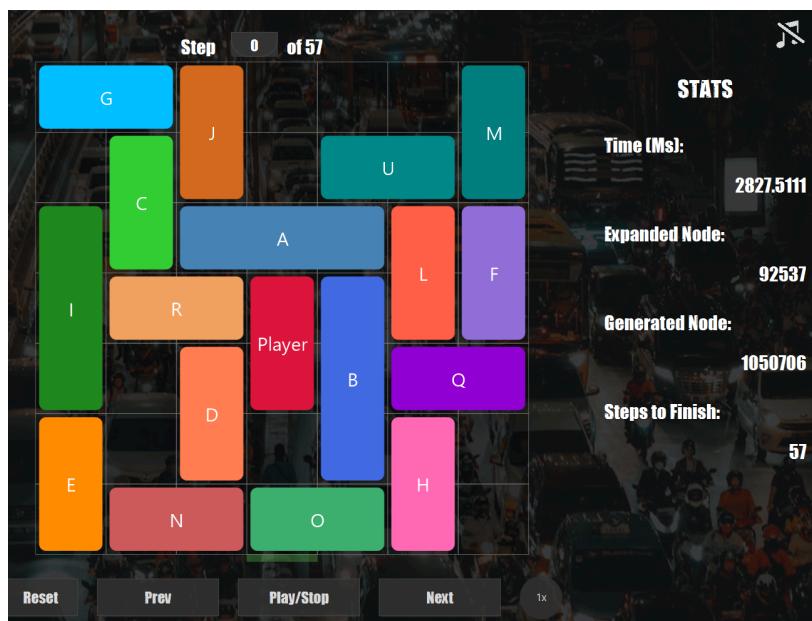
27. Algoritma UCS, Recursive Blocking Heuristic, Terdapat Solusi

Input	Output
-------	--------

```

7 7
17
GGJ...M
.CJ.UUM
ICAAALF
IRRPBLF
I.DPBQQ
E.D.BH.
ENNOOH.
K

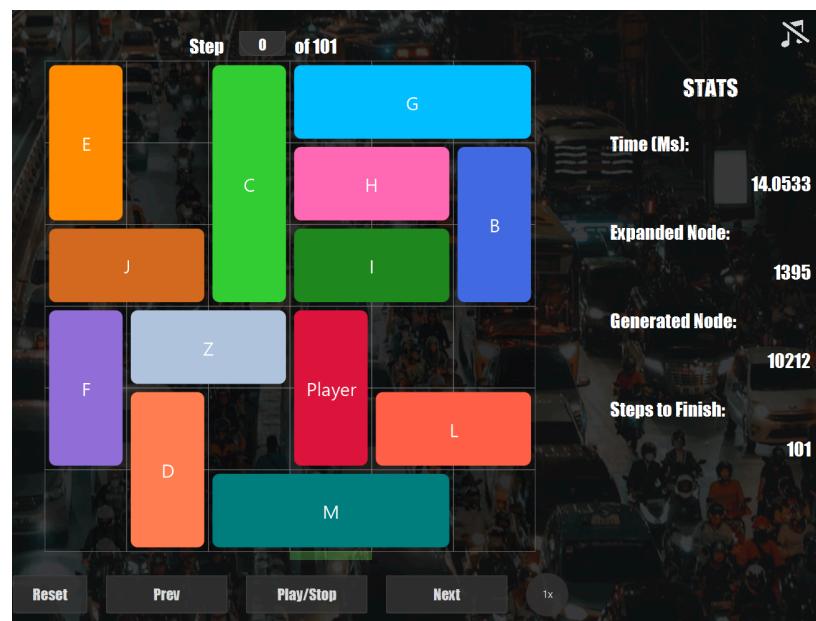
```



28. Algoritma GBFS, Recursive Blocking Heuristic, Terdapat Solusi

Input	Output
-------	--------

```
6 6  
12  
E.CGGG  
E.CHHB  
JJCIIB  
FZZP..  
FD.PLL  
.DMMM.  
K
```

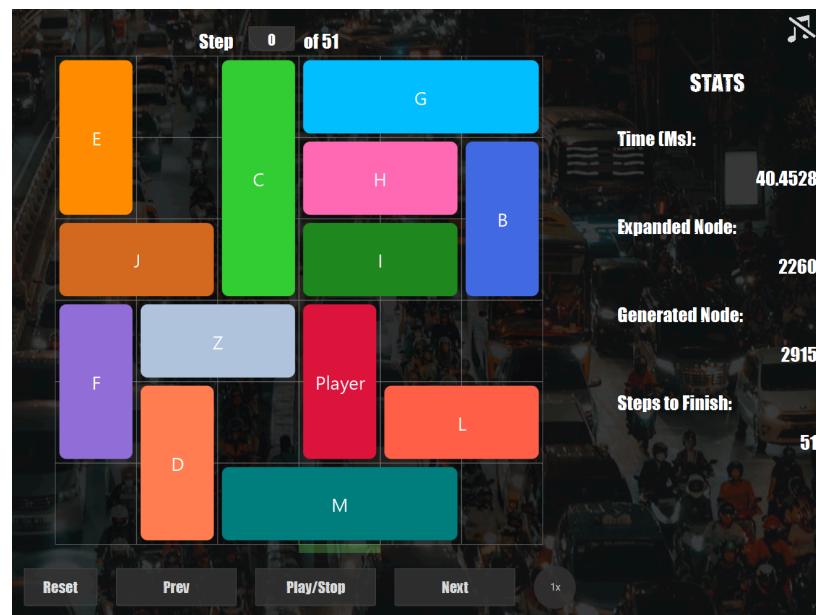


```
solution.txt  
File Edit View  
  
Solution Stats  
Time : 14.0533  
Expanded Node : 1395  
Generated Node : 10212  
Solution Depth : 101  
  
Initial Board  
E.CGGG  
E.CHHB  
JJCIIB  
FZZP..  
FD.PLL  
.DMMM.
```

29. Algoritma Beam, Recursive Blocking Heuristic, Terdapat Solusi

Input	Output
-------	--------

```
6 6  
12  
E.CGGG  
E.CHBB  
JJCIIB  
FZZP..  
FD.PLL  
.DMMM.  
K
```

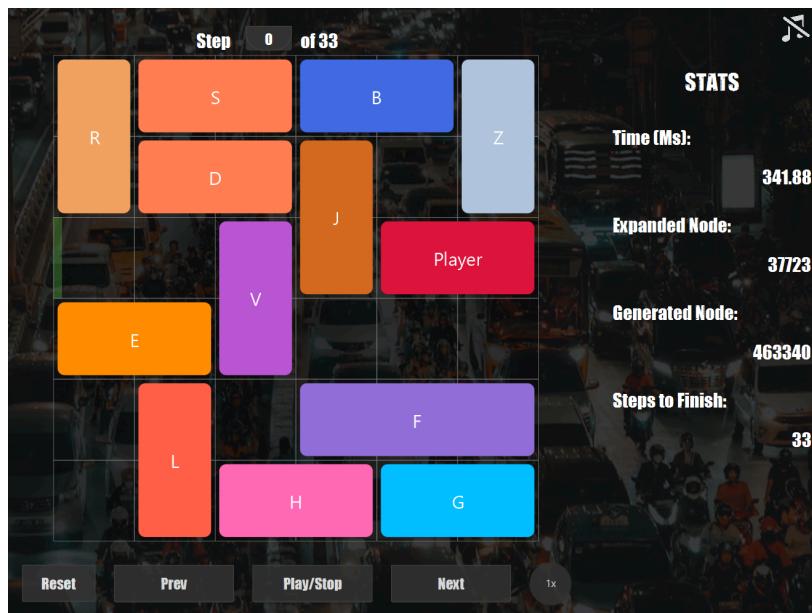


```
solution.txt  
File Edit View  
  
Solution Stats  
Time : 40.4528  
Expanded Node : 2260  
Generated Node : 2915  
Solution Depth : 51  
  
Initial Board  
E.CGGG  
E.CHBB  
JJCIIB  
FZZP..  
FD.PLL  
.DMMM.
```

30. Algoritma A*, Distance Heuristic, Terdapat Solusi

Input	Output
-------	--------

```
6 6  
12  
RSSBBZ  
RDDJ.Z  
K..VJPP  
EEV...  
.L.FFF  
.LHHGG
```



```
solution.txt
File Edit View

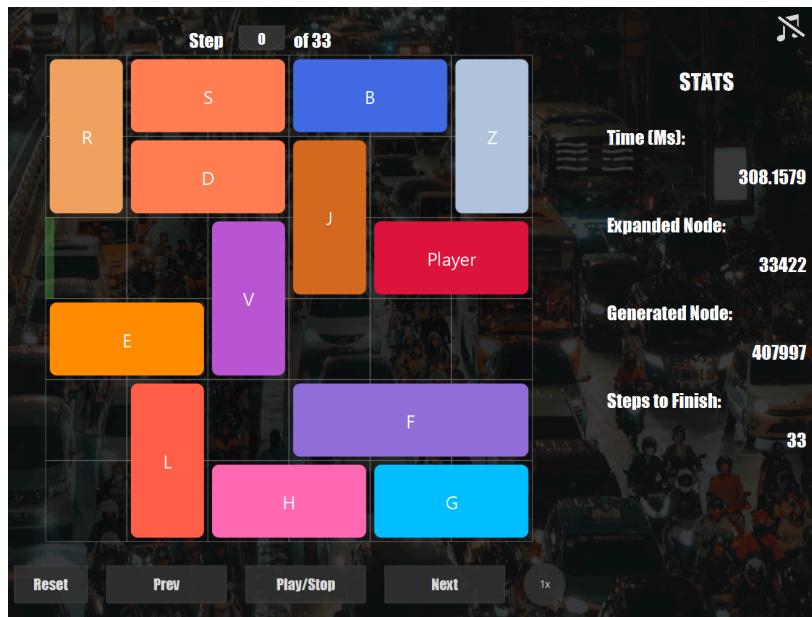
Solution Stats
Time : 341.88
Expanded Node : 37723
Generated Node : 463340
Solution Depth : 33

Initial Board
RSSBBZ
RDDJ.Z
..VJPP
EEV...
.L.FFF
.LHHGG
```

31. Algoritma UCS, Distance Heuristic, Terdapat Solusi

Input	Output
-------	--------

```
6 6  
12  
RSSBBZ  
RDDJ.Z  
K..VJPP  
EEV...  
.L.FFF  
.LHHGG
```



```
solution.txt
File Edit View

Solution| Stats
Time : 308.1579
Expanded Node : 33422
Generated Node : 407997
Solution Depth : 33

Initial Board
RSSBBZ
RDDJ.Z
..VJPP
EEV...
.L.FFF
.LHHGG
```

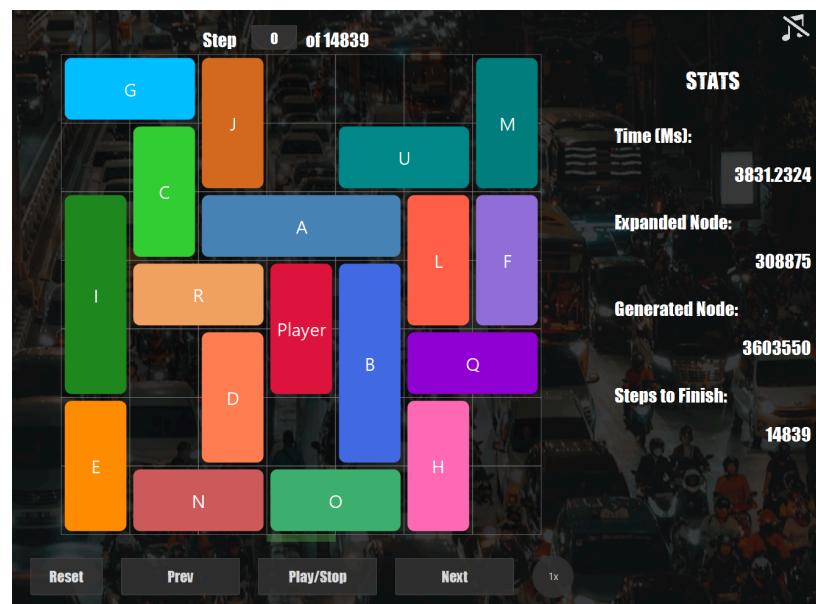
32. Algoritma GBFS, Distance Heuristic, Terdapat Solusi

Input	Output
-------	--------

```

7 7
17
GGJ...M
.CJ.UUM
ICAAALF
IRRPBLF
I.DPBQQ
E.D.BH.
ENNOOH.
K

```



```

solution.txt

File Edit View

Solution Stats
Time : 3831.2324
Expanded Node : 308875
Generated Node : 3603550
Solution Depth : 14839

Initial Board
GGJ...M
.CJ.UUM
ICAAALF
IRRPBLF
I.DPBQQ
E.D.BH.
ENNOOH.

```

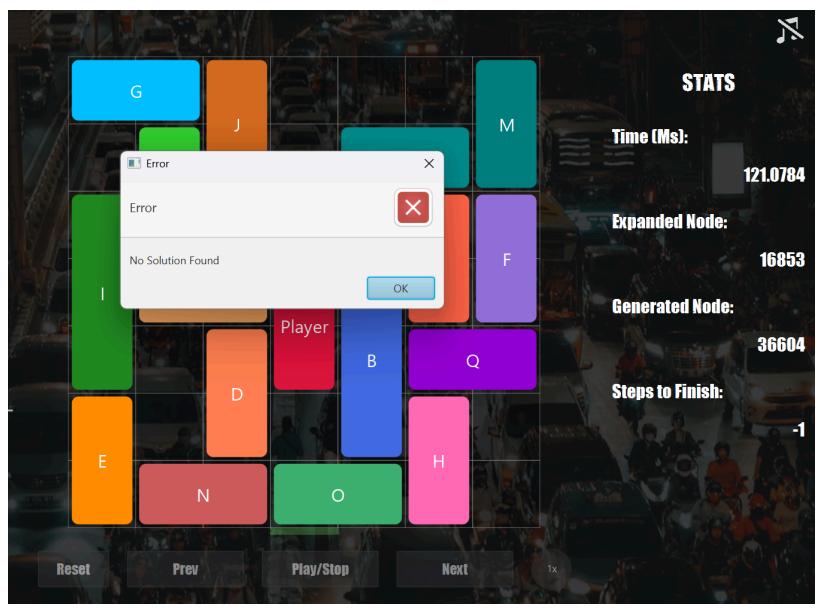
33. Algoritma Beam, Distance Heuristic, Tidak ada Solusi

Input	Output
-------	--------

```

7 7
17
GGJ...M
.CJ.UUM
ICAAALF
IRRPBLF
I.DPBQQ
E.D.BH.
ENNOOH.
K

```



34. Tidak Ada Solusi

Input	Output
<pre> 6 6 12 K FF.IZ. DDDIZM .CCIZM BBP.JJ A.PHHL A.GGGL </pre>	<p>STATS</p> <ul style="list-style-type: none"> Time (Ms): 2.7805 Expanded Node: 383 Generated Node: 2597 Steps to Finish: -1

Analisis Hasil Pengujian

Berdasarkan hasil pengujian, ditemukan beberapa constraint format dari papan yang dapat diberikan oleh pengguna aplikasi. Aplikasi ini sudah menangani banyak kasus kesalahan saat menginput papan yang disediakan oleh pengguna. Mulai dari salah besar papan dengan nilai yang diberikan sampai kesalahan dalam meletakkan pintu keluar mobil.

Melanjutkan ke pengujian yang dapat menerima papan pengguna, dapat kita temukan perbedaan antara algoritma dan heuristik yang digunakan baik dari sisi waktu pencarian solusi, optimalitas, ataupun jumlah *state* yang dikunjungi. Terlihat bahwa waktu pencarian algoritma Beam lebih cepat dibandingkan algoritma yang lain. Akan tetapi, algoritma ini menemukan solusi yang tidak optimal atau bahkan tidak menemukan solusi. Algoritma yang kedua tercepat adalah algoritma GBFS namun algoritma ini sangat tidak optimal melebihi ketidakoptimalan algoritma Beam. Jika algoritma lain bisa mendapatkan solusi dalam 50 hingga 100 langkah, Algoritma GBFS akan memiliki solusi dengan 1000 langkah.

Dari sisi optimalitas solusi, ditemukan bahwa algoritma A* dan UCS memiliki optimalitas terbaik dibandingkan algoritma lain. Akan tetapi, kedua algoritma ini menggunakan memory lebih besar dibandingkan algoritma lainnya.

Bonus

Algoritma Pencarian Rute Alternatif

Di samping tiga algoritma utama (UCS, GBFS, A*), kami juga mengimplementasikan **Beam Search** (pada ranah projek ini sebagai ekstensi dari A*) sebagai opsi alternatif. Beam Search membatasi antrean prioritas pada k simpul terbaik menurut $f(n) = g(n) + h(n)$ di setiap tingkat kedalaman, sehingga secara signifikan menurunkan penggunaan memori dan waktu. Meskipun tidak lengkap atau optimal, Beam Search sering kali cukup efektif untuk puzzle berukuran besar dengan batasan sumber daya. Dalam implementasi ini, kami menjadikan algoritma Beam Search sebagai alternatif dari GBFS.

Untuk penjelasan lebih lengkap, mohon merujuk ke bagian [Beam Search](#)

Heuristik Alternatif

Selain heuristik dasar yang telah dijelaskan (Distance, Blocking, Recursive Blocking), keempat heuristik tersebut menjadi pilihan utama:

- **Distance**

Mengembalikan 0 jika jalur target lurus tanpa blocker, atau 1 jika ada block. Sederhana dan sangat cepat dihitung, memberi panduan minimal untuk A*.

- **Blocking**

Menghitung jumlah kendaraan dalam jalur lurus target menuju pintu keluar. Memberi informasi jumlah langkah minimal untuk membuka jalan.

- **Recursive Blocking**

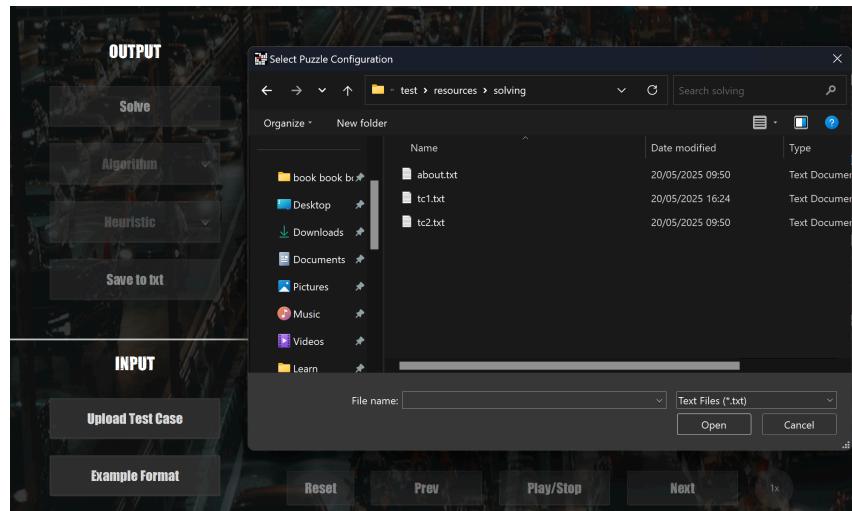
Menambahkan biaya blocker dari setiap blocker secara rekursif. Lebih informatif pada puzzle dengan rantai hambatan berlapis, meski komputasinya lebih berat.

Untuk penjelasan lebih lengkap, mohon merujuk ke bagian [Heuristik](#)

Graphical User Interface (GUI)

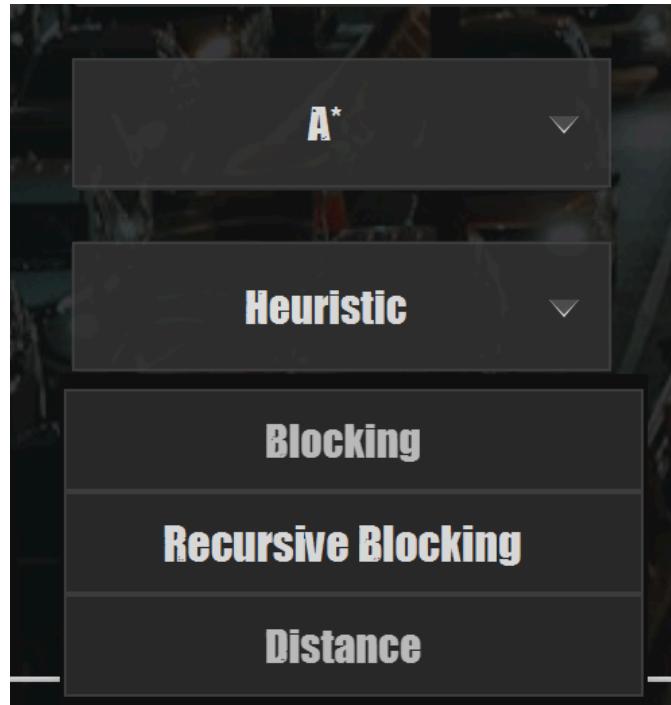
Antarmuka JavaFX yang ramah pengguna, *chaos (as it should be)*, dan imersif yang menghidupkan suasana nyata. *Upload* puzzle, pilih algoritma dan heuristik, dan tonton solusinya beranimasi *brick by brick* (tentu saja dengan *solver* yang *blazingly fast* ⚡).

Puzzle Loader



Memungkinkan pengguna membuka berkas .txt puzzle, memvalidasi format, dan menampilkan papan dengan visual kendaraan.

Pemilihan Algoritma dan Heuristik



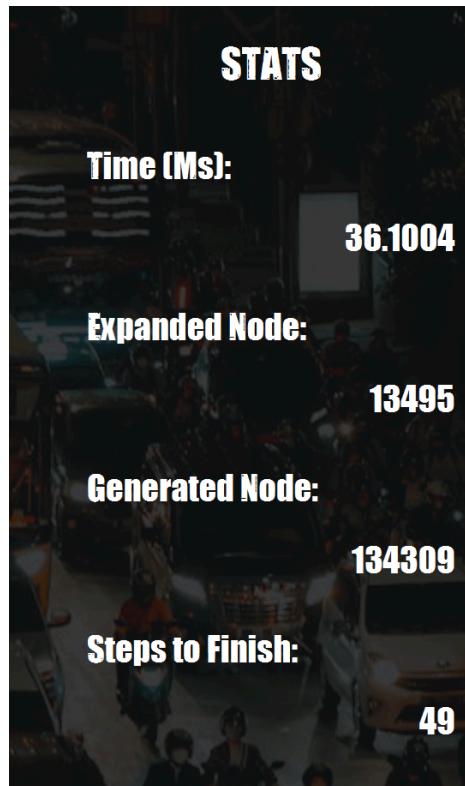
Menu dropdown untuk memilih UCS, GBFS, A*, atau Beam Search, serta satu dari tiga heuristik (Distance, Blocking, Recursive).

Animated Solution



Kontrol play/pause dan slider langkah solusi, memungkinkan pengguna mengikuti atau melompat ke langkah mana saja.

Performance Stats



Tampilan metrik eksekusi: jumlah node diekspansi, node dihasilkan, langkah pergerakan solusi, dan waktu komputasi.

Save Result

```
Solution Stats
Time       : 36.1004
Expanded Node : 13495
Generated Node : 134309
Solution Depth : 49

Initial Board
BBBZLM
HCCZLM
H.PPLM
DDJ...
.IJEE.
.IFFGG

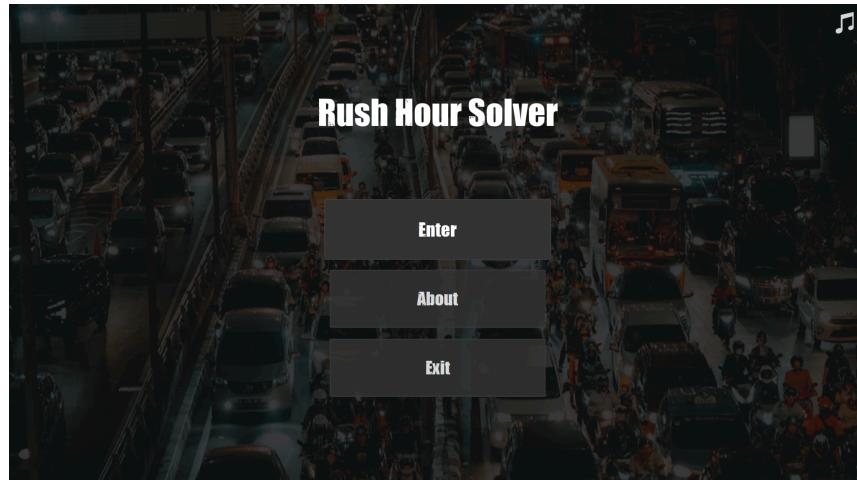
Step 1: P-left
BBBZLM
HCCZLM
HPP.LM
DDJ...
.IJEE.
.IFFGG

Step 2: Z-down
BBB.LM
HCC.LM
HPZLM
DDJZ..
.IJEE.
.IFFGG

Ln 1, Col 1 | 3.032 characters | 100% | Unix (LF) | UTF-8
```

Tombol untuk mengekspor laporan .txt berisi detail hasil (jalur solusi, statistik, pilihan algoritma/heuristik).

Immersive Experience



Latar belakang gaya “kemacetan Indonesia” dan efek suara menambah suasana nyata dan ke-chaos-an.

Optimisasi

Untuk memaksimalkan performa solver yang kami buat, sejumlah optimasi *low level* dan algoritmik diterapkan. Tiap teknik menargetkan bottleneck berbeda, dari churn memori, kecepatan lookup, hingga aritmetika bitwise, sehingga puzzle “terberat” pun bisa selesai dalam hitungan millisecond.

BitSet Board Representation

Setiap posisi kendaraan direpresentasikan sebagai BitSet mask pada grid yang sudah di-flatten. Operasi bitwise seperti `and`, `or`, dan `intersects` memungkinkan **pengujian tabrakan (collision) dan okupansi dalam beberapa CPU cycle**, alih-alih iterasi per-sel. Hal ini sangat menguntungkan ketika generator neighbor harus membuat ribuan state baru.

Precomputed Vehicle Masks

Alih-alih membangun mask kendaraan secara dinamis, kami melakukan *precomputation* untuk setiap kendaraan dan setiap posisi dasar yang valid sebuah BitSet footprint. Saat search, cukup panggil `vehicleMasks.get(i).get(pos)`, tanpa loop atau alokasi baru, hanya lookup peta yang super duper cepat.

Priority Queue for Open Set

Kami menggunakan `PriorityQueue<State>` yang diurutkan oleh nilai $f = g + h$ (atau hanya g untuk UCS). Dengan kompleksitas $O(\log n)$ untuk insert dan poll, operasi frontier tetap **ekstrem cepat** meski ribuan state berada di antrean.

HashMap for Best-Seen Costs

Sebuah `Map<State, Integer>` menyimpan best g-score yang pernah ditemui per state. Sebelum menambah neighbor ke open set, kami melakukan cek `best.get(nxt)` dalam $O(1)$ untuk memangkas jalur yang tidak memberikan “perbaikan” cost. Hal ini menghindari ekspansi redundant dan menjaga penggunaan memori terkontrol.

Efficient Neighbor Generation

Dengan **melepas mask kendaraan** dari occupancy BitSet sebelum sliding, setiap pergerakan inkremental (while loop) bisa diuji secara terpisah. Kami hanya menyalin dan memodifikasi `State` saat menemukan move valid, lalu menghitung heuristik-nya secara lazy. Juga reuse ketat `State.copy()`. Hal ini meminimalkan churn objek dan tekanan GC (*Garbage Collector*).

Lampiran

Tautan Repotori

https://github.com/l0stplains/Tucil3_13523002_13523085

Tabel Pernyataan

No	Poin	Ya	Tidak
1	Program berhasil dikompilasi tanpa kesalahan	✓	
2	Program berhasil dijalankan	✓	
3	Solusi yang diberikan program benar dan mematuhi aturan permainan	✓	
4	Program dapat membaca masukan berkas .txt dan menyimpan solusi berupa print board tahap per tahap dalam berkas .txt	✓	
5	[Bonus] Implementasi algoritma pathfinding alternatif	✓	
6	[Bonus] Implementasi 2 atau lebih heuristik alternatif	✓	
7	[Bonus] Program memiliki GUI	✓	
8	Program dan laporan dibuat (kelompok) sendiri	✓	

Daftar Pustaka

M. Rinaldi. *Route Planning (Bagian 1)*. Diakses pada 15 Mei 2025 dari [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-\(2025\)-Bagian1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf)

M. Rinaldi. *Route Planning (Bagian 2)*. Diakses pada 15 Mei 2025 dari [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-\(2025\)-Bagian2.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf)

F. Michael. *Solving Rush Hour, the Puzzle*. Diakses pada 17 Mei 2025 dari <https://www.michaelfogleman.com/rush/>

Akhir Kata

“Im tubesing it.

Read A Regression Tale of Cultivation

It's so peak.”

--- Jibril ---

“Saya tunggu kabar baik selanjutnya dari Pak Rin”



--- refki ---