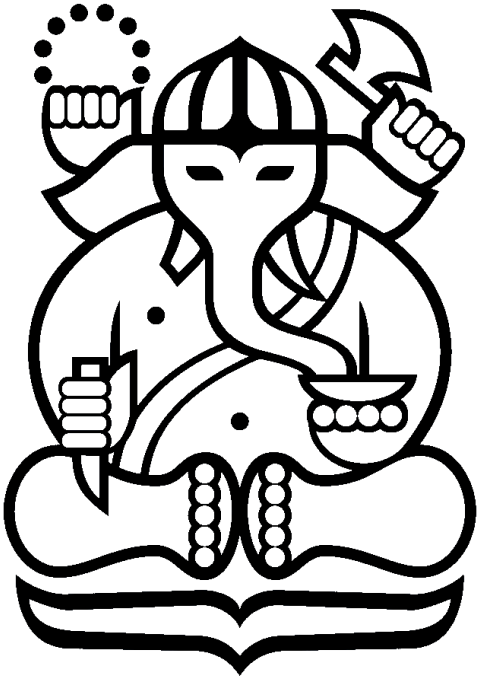
Laporan Tugas Kecil 3
IF2211 Strategi Algoritma 2025/2026
Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding



Disusun oleh:

Dave Daniell Yanni	13523003
Daniel Pedrosa Wu	13523099

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
JL. GANESA 10, BANDUNG
40132

2025

DAFTAR ISI

BAB I:
DESKRIPSI TUGAS

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:

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*.

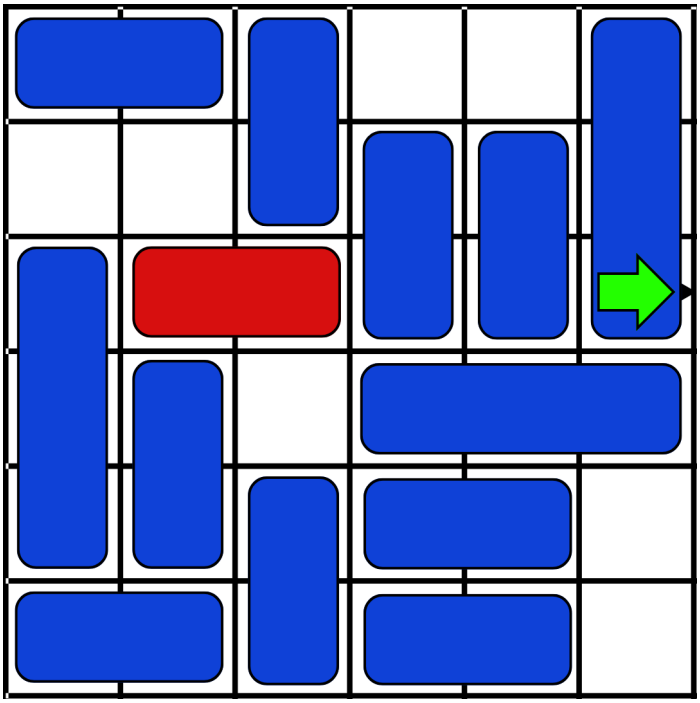
2. **Piece –** *Piece* adalah sebuah kendaraan di dalam papan. Setiap *piece* memiliki *posisi*, *ukuran*, dan *orientasi*. *Orientasi* sebuah *piece* hanya dapat berupa horizontal atau vertikal–tidak mungkin diagonal. *Piece* dapat memiliki beragam *ukuran*, yaitu jumlah *cell* yang ditempati oleh *piece*. 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

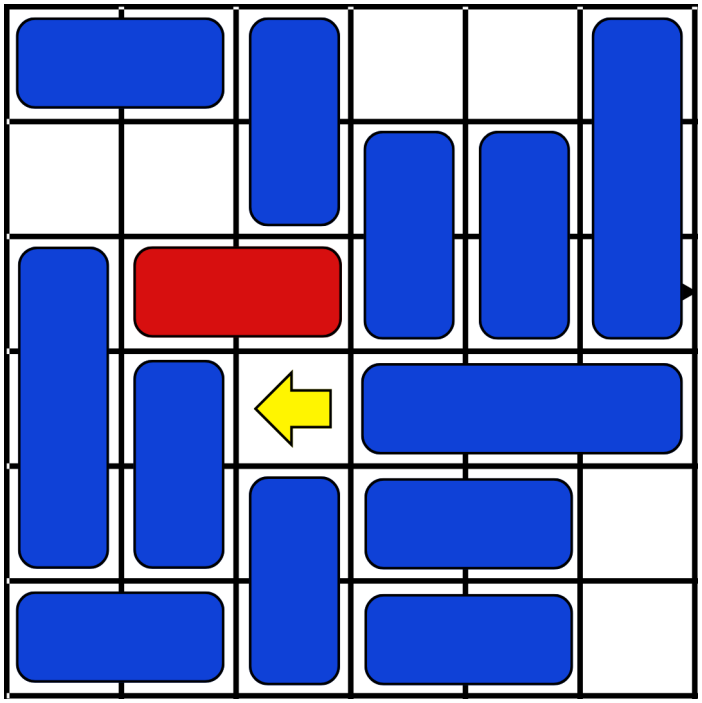
5. **Gerakan** — *Gerakan* yang dimaksudkan adalah pergeseran *piece* di dalam permainan. *Piece* hanya dapat bergerak/bergeser lurus sesuai orientasinya (atas-bawah jika vertikal dan kiri-kanan jika horizontal). Suatu *piece* tidak dapat digerakkan melewati/menembus *piece* yang lain.

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.

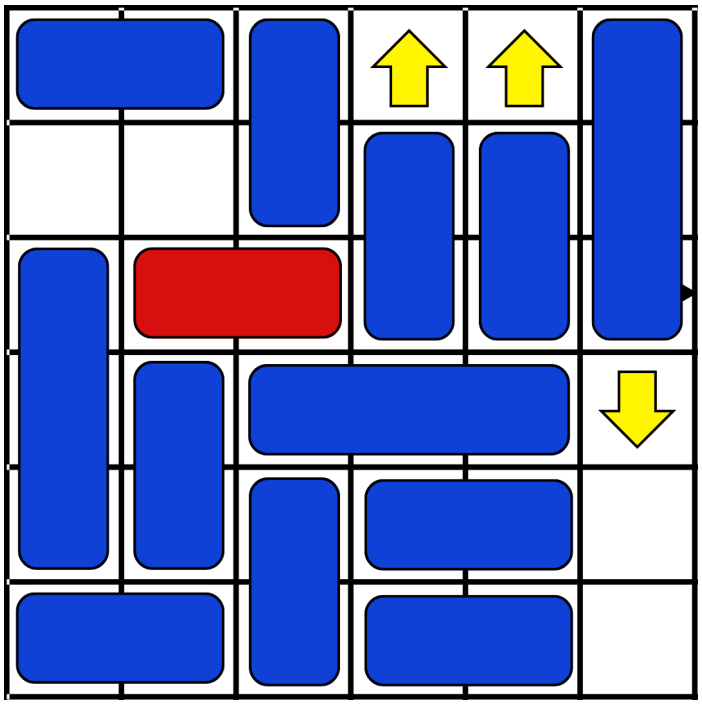


Gambar 2. Awal Permainan Game Rush Hour

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

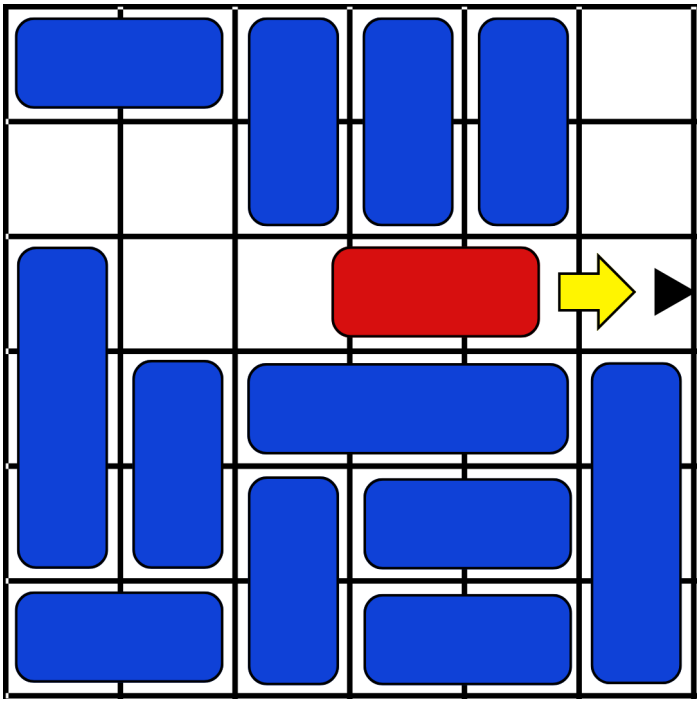


Gambar 3. Gerakan Pertama Game Rush Hour



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

BAB II:
PENJELASAN ALGORITMA

2.1. Algoritma Uniform Cost Search (UCS)

Algoritma Uniform Cost Search atau UCS adalah algoritma pencarian jalur terpendek yang bekerja dengan prinsip memperluas simpul yang memiliki biaya kumulatif terendah dari simpul awal hingga mencapai tujuan. UCS termasuk dalam kategori *uninformed search*. Artinya, algoritma ini tidak menggunakan informasi tambahan atau estimasi heuristik mengenai jarak ke tujuan, sehingga pencarian dilakukan secara sistematis dan menyeluruh berdasarkan bobot biaya yang sebenarnya.

Algoritma UCS menggunakan struktur data *priority queue*. Berbeda dengan *queue* biasa yang menerapkan prinsip FIFO (First-In, First-Out) di mana elemen yang pertama kali masuk adalah elemen yang pertama keluar, *priority queue* mengurutkan elemen-elemen berdasarkan nilai prioritasnya. Elemen dengan nilai prioritas tertinggi akan menjadi elemen yang pertama kali keluar. Ini memungkinkan algoritma untuk memproses simpul yang paling menjanjikan berdasarkan fungsi tertentu. Berikut adalah *pseudocode* untuk implementasi dari algoritma UCS.

```
FUNCTION UniformCostSearch(start_state):

    CREATE a priority queue openList ordered by cumulative cost
    CREATE a map closedList to record visited states and their lowest cost

    INITIALIZE start_node WITH:
        state <- start_state
        g(n) <- 0
        path <- empty list

    ADD start_node TO OPENLIST

    WHILE openList IS NOT empty:
        current_node <- REMOVE node FROM openList WITH lowest g(n)

        IF current_node.state IS goal:
            -> (found <- true, path <- current_node.path)

        IF current_node.state IN closedList WITH cost ≤ current_node.g(n):
            CONTINUE TO NEXT ITERATION

        RECORD current_node.state AND g(n) IN closedList

        FOR EACH valid action FROM current_node.state:
            next_state <- result of applying action
            new_cost <- current_node.g(n) + cost_of_action
            IF next_state NOT IN closedList OR new_cost < recorded cost:
                CREATE next_node WITH:
                    state <- next_state
                    f(n) <- new_cost
                    path <- current_node.path + action
```

```
                ADD next_node TO openList

    -> (found = false, path = empty)
```

Untuk permasalahan puzzle Rush Hour dapat diselesaikan dengan menggunakan algoritma UCS, puzzle ini harus direpresentasikan dalam bentuk suatu graf. Simpul dari graf merepresentasikan konfigurasi papan permainan saat ini, nilai f(n) sejauh ini, dan perjalanan sejauh ini. Pada algoritma UCS, terdapat dua *list* yaitu *open list* dan *closed list*. *Open list* adalah sebuah *priority queue* yang menyimpan setiap simpul yang telah ditemukan tetapi belum diproses, sementara *closed list* adalah sebuah *map* yang menyimpan simpul yang telah diproses beserta biaya terbaik sejauh ini.

Dalam konteks puzzle Rush Hour, biaya kumulatif direpresentasikan oleh jumlah langkah yang telah diambil dari awal permainan. Pada algoritma Uniform Cost Search, fungsi evaluasi yang digunakan untuk mengurutkan simpul dalam *open list* adalah f(n), yaitu nilai total biaya yang sudah dikeluarkan dari simpul awal hingga simpul saat ini. Pada UCS, f(n) sama dengan g(n), di mana g(n) merepresentasikan biaya kumulatif sejauh ini dari titik awal ke simpul n, dalam kasus ini adalah jumlah langkah yang telah dilalui.

Saat algoritma pertama kali dijalankan, simpul pertama yang masuk ke dalam *open list* merepresentasikan konfigurasi papan permainan di awal. Karena ini adalah konfigurasi awal, maka tidak akan ada entrinya di dalam *closed list*, sehingga dimasukkan ke dalam *closed list* dan dikeluarkan dari *open list*. Algoritma UCS kemudian akan memperluas simpul dengan mencari setiap satu langkah yang sahih dilakukan pada konfigurasi papan sekarang ini. Simpul-simpul tersebut akan dimasukkan ke dalam *open list* untuk diproses pada iterasi selanjutnya. Nilai f(n) akan diubah berdasarkan nilai g(n) yaitu jumlah langkah sejauh ini.

Jika ditemukan entri dari suatu konfigurasi papan di *closed list* yang telah memiliki nilai f(n) yang lebih rendah, maka langsung akan dilanjutkan ke iterasi selanjutnya namun jika nilai f(n) lebih rendah, maka entri dalam *closed list* akan diperbarui. Jika konfigurasi simpul telah mencapai solusi, maka fungsi akan mengembalikan langkah-langkah yang dilakukan untuk mencapai solusi tersebut.

Secara umum, algoritma UCS dijamin optimal asalkan biaya setiap aksi adalah bilangan non-negatif. Optimalitas ini terjamin karena algoritma selalu memproses simpul dengan nilai f(n) terendah, artinya solusi pertama yang ditemukan pasti adalah solusi dengan nilai kumulatif f(n) terendah. *Closed list* memastikan bahwa simpul yang telah dikunjungi tidak akan diproses secara tidak perlu.

2.2. Algoritma Greedy Best First Search

Algoritma Greedy Best First Search atau GBFS adalah algoritma pencarian yang termasuk dalam kategori *informed search*. Artinya, algoritma ini menggunakan informasi tambahan berupa fungsi heuristik yang memperkirakan jarak atau biaya sisa dari suatu simpul menuju tujuan. GBFS bekerja dengan prinsip memperluas simpul

yang menurut fungsi heuristiknya paling dekat atau paling menjanjikan menuju solusi, tanpa memperhitungkan biaya kumulatif yang telah dikeluarkan sebelumnya.

Secara teknis, GBFS menggunakan sebuah struktur data *priority queue* sebagai open list, yang mengurutkan simpul berdasarkan nilai heuristik h(n) saja, tanpa mempertimbangkan biaya kumulatif g(n) yang telah ditempuh. Ini berbeda dengan algoritma seperti Uniform Cost Search (UCS) atau A* yang menggunakan fungsi evaluasi f(n)=g(n)+h(n). Dengan cara ini, GBFS berupaya "mengikuti" jalur yang diperkirakan paling cepat atau paling dekat ke solusi. Berikut adalah *pseudocode* untuk implementasi dari algoritma GBFS.

```
FUNCTION GreedyBestFirstSearch(start_state, heuristicFunction):

    CREATE a priority queue openList ordered by heuristic h(n)
    CREATE a set closedList to record visited states

    INITIALIZE start_node WITH:
        state <- start_state
        h(n) <- heuristicFunction(start_state)
        path <- empty list

    ADD start_node TO openList

    WHILE openList IS NOT empty:
        current_node <- REMOVE node FROM openList WITH lowest h(n)

        IF current_node.state IS goal:
            -> (found <- true, path <- current_node.path)

        IF current_node.state IN closedList:
            CONTINUE TO NEXT ITERATION

        ADD current_node.state TO closedList

        FOR EACH valid action FROM current_node.state:
            next_state <- result of applying action
            heuristic_cost <- heuristicFunction(next_state)

            IF next_state NOT IN closedList:
                CREATE next_node WITH:
                    state <- next_state
                    h(n) <- heuristic_cost
                    path <- current_node.path + action

                ADD next_node TO openList

    -> (found = false, path = empty)
```

Dalam implementasinya, algoritma GBFS mirip dengan algoritma UCS di mana keduanya memanfaatkan *open list* yang merupakan sebuah *priority queue* dan *closed list* yang merupakan sebuah *map*. Algoritma ini memasukkan suatu simpul yang merepresentasikan konfigurasi awal papan permainan ke dalam *open list*. Kemudian, tiap simpul dalam *open list* akan diperluas satu per satu, menghasilkan simpul-simpul baru yang akan dimasukkan ke *open list*. Setiap simpul anak yang dihasilkan dihitung

nilai heuristiknya menggunakan fungsi heuristik yang sama, lalu dimasukkan ke open list jika belum ada dalam closed list. Proses ini berlanjut hingga simpul tujuan ditemukan atau *open list* habis, yang berarti tidak ada solusi.

Perbedaannya terletak pada cara mereka mengurutkan dan memilih simpul yang akan dikembangkan berikutnya. Pada UCS, simpul dipilih berdasarkan nilai f(n) = g(n), yaitu biaya kumulatif terendah yang telah dikeluarkan dari titik awal ke simpul tersebut. Sedangkan pada GBFS, simpul diprioritaskan berdasarkan nilai f(n) = h(n), yaitu nilai heuristik yang memperkirakan jarak atau biaya tersisa menuju tujuan. Dengan demikian, GBFS cenderung lebih "agresif" dalam mengikuti jalur yang terlihat paling menjanjikan secara heuristik, tanpa mempertimbangkan biaya perjalanan sejauh ini.

Dalam konteks puzzle Rush Hour, GBFS menggunakan fungsi heuristik h(n) untuk memperkirakan seberapa dekat konfigurasi papan saat ini dengan solusi tanpa mempertimbangkan biaya kumulatif. Dengan memprioritaskan simpul berdasarkan estimasi ini, algoritma GBFS sering kali dapat menemukan solusi dengan lebih cepat dibandingkan algoritma uninformed seperti UCS, terutama jika fungsi heuristiknya akurat dan informatif.

Namun, meskipun lebih efisien, GBFS tidak menjamin solusi yang ditemukan adalah solusi dengan langkah paling sedikit, terutama jika fungsi heuristik kurang akurat. Jika fungsi heuristik tidak konsisten atau kurang akurat, GBFS bisa terjebak pada jalur yang salah atau bahkan gagal menemukan solusi. Oleh karena itu, algoritma ini cocok digunakan ketika kecepatan lebih penting daripada optimalitas.

2.3. Algoritma A*

Algoritma A* adalah algoritma pencarian yang termasuk ke dalam kategori *informed search*, yang artinya algoritma ini memanfaatkan suatu heuristik. Tujuan dari algoritma ini adalah menggabungkan keunggulan algoritma Uniform Cost Search (UCS) dan Greedy Best First Search (GBFS). A* menggunakan fungsi evaluasi yang mengkombinasikan biaya kumulatif yang telah dikeluarkan dari simpul awal ke simpul saat ini g(n) dan estimasi biaya tersisa menuju tujuan yang didapatkan dari fungsi heuristik h(n). Fungsi evaluasi ini dinyatakan sebagai f(n) = g(n) + h(n).

Seperti algoritma UCS dan GBFS, A* memanfaatkan struktur data *priority queue* sebagai *open list* yang mengurutkan simpul berdasarkan nilai f(n). Selain itu, A* menggunakan *closed list* yang berfungsi menyimpan simpul yang sudah diproses beserta nilai biaya terbaik untuk menghindari eksplorasi ulang simpul yang sama dengan biaya yang lebih tinggi. Berikut adalah *pseudocode* untuk implementasi dari algoritma A*.

```
FUNCTION AStarSearch(start_state, heuristicFunction):

    CREATE a priority queue openList ordered by f(n) = g(n) + h(n)
    CREATE a map closedList to record visited states and their lowest g(n)

    INITIALIZE start_node WITH:
        state <- start_state
```

```
        g(n) <- 0
        h(n) <- heuristicFunction(start_state)
        f(n) <- g(n) + h(n)
        path <- empty list

    ADD start_node TO openList

    WHILE openList IS NOT empty:
        current_node <- REMOVE node FROM openList WITH lowest f(n)

        IF current_node.state IS goal:
            -> (found <- true, path <- current_node.path)

        IF current_node.state IN closedList WITH cost ≤ current_node.g(n):
            CONTINUE TO NEXT ITERATION

        RECORD current_node.state AND current_node.g(n) IN closedList

        FOR EACH valid action FROM current_node.state:
            next_state <- result of applying action
            new_g <- current_node.g(n) + cost_of_action
            new_h <- heuristicFunction(next_state)
            new_f <- new_g + new_h

            IF next_state NOT IN closedList OR new_g < recorded cost:
                CREATE next_node WITH:
                    state <- next_state
                    g(n) <- new_g
                    h(n) <- new_h
                    f(n) <- new_f
                    path <- current_node.path + action

                ADD next_node TO openList

    -> (found = false, path = empty)
```

Seperti dua implementasi sebelumnya, algoritma A* juga menggunakan sebuah *open list* dan sebuah *closed list*. Prinsip kerjanya juga sangat mirip dengan algoritma UCS dan GBFS, karena algoritma ini "menggabungkan" kedua algoritma tersebut. Dalam konteks puzzle Rush Hour, g(n) merepresentasikan jumlah langkah yang ditempuh untuk mencapai konfigurasi papan permainan saat ini sementara h(n) adalah estimasi biaya yang didapatkan melalui fungsi heuristik. Dengan menggabungkan kedua informasi ini, maka algoritma ini dapat memilih jalur yang optimal dengan tetap menjaga efisiensi dari algoritma.

Algoritma A* dijamin menemukan solusi yang optimal jika fungsi heuristik yang digunakan *admissible*. Fungsi heuristik h(n) dikatakan *admissible* jika untuk setiap simpul n, nilai heuristiknya tidak pernah melebihi biaya sebenarnya untuk mencapai tujuan dari n. Berikut adalah representasi matematisnya.

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

di mana h*(n) adalah biaya sebenarnya dari n ke tujuan. Sifat ini memastikan bahwa algoritma A* tidak melewatkan solusi optimalnya karena perkiraan heuristik yang terlalu optimis.

2.4. Algoritma Beam Search

Algoritma Beam Search adalah algoritma pencarian jalur yang termasuk ke dalam kategori *informed search*. Beam Search merupakan varian dari Greedy Best First Search (GBFS) yang membatasi jumlah simpul yang diperluas pada setiap tingkat kedalaman untuk menjaga efisiensi waktu dan penggunaan memori. Sama seperti algoritma GBFS, fungsi evaluasi f(n) yang digunakan pada algoritma Beam Search hanya mempertimbangkan estimasi biaya dari fungsi heuristik h(n).

Berbeda dengan ketiga algoritma sebelumnya, algoritma Beam Search tidak menggunakan *priority queue* dan membatasi jumlah node yang akan diproses dengan hanya memilih sejumlah k simpul terbaik pada setiap langkah berdasarkan nilai heuristik. Nilai k ini disebut sebagai lebar beam (beam width). Dengan cara ini, Beam Search mengurangi overhead komputasi dan memori dengan melakukan pruning secara agresif pada simpul-simpul yang dianggap kurang menjanjikan.

```
FUNCTION BeamSearch(start_state, heuristicFunction, beamWidth):

    INITIALIZE start_node WITH:
        state <- start_state
        h(n) <- heuristicFunction(start_state)
        path <- empty list

    SET beam <- list containing start_node

    WHILE beam IS NOT empty:
        INITIALIZE nextBeam as empty list

        FOR EACH currentNode IN beam:

            IF isSolutionFound(currentNode.state):
                -> (found <- true, path <- currentNode.path)

            SET validMoves <- getAllValidMoves(currentNode.state)

            FOR EACH move IN validMoves:
                next_state <- result of applying movePiece(currentNode.state,
move)

                newH <- heuristicFunction(next_state)

                CREATE newNode WITH:
                    state <- next_state
                    h(n) <- newH
                    path <- currentNode.path + move

                ADD newNode TO nextBeam

        SORT nextBeam BY h(n) ascending
        beam <- first beamWidth elements of nextBeam

    -> (found = false, path = empty)
```

Pada algoritma Beam Search, konfigurasi awal papan permainan juga akan

menjadi simpul pertama. Perbedaannya, algoritma ini tidak menggunakan *priority queue* sehingga semua simpul akan disimpan dalam *list* biasa. Setiap simpul yang ada di dalam *beam* akan diproses dan akan menghasilkan simpul-simpul baru berdasarkan tiap satu pergerakan yang mungkin dari simpul pendahulunya.

Algoritma ini hanya mempertimbangkan nilai estimasi yang berasal dari fungsi heuristiknya. Tiap simpul akan dihitung nilainya menggunakan fungsi evaluasi yang sama. Berdasarkan ukuran lebar *beam*, algoritma akan menentukan simpul-simpul mana yang paling menjanjikan dan itu akan dimasukkan ke dalam *beam* untuk diproses pada iterasi selanjutnya. Proses ini berlanjut hingga simpul tujuan ditemukan atau *beam* habis, yang berarti tidak ada solusi.

Algoritma Beam Search umumnya tidak menjamin keoptimalan solusi yang ditemukan. Hal ini disebabkan oleh sifatnya yang melakukan pruning secara agresif dengan hanya mempertahankan sejumlah k simpul terbaik (*beam width*) pada setiap tingkat pencarian. Karena hanya sebagian kecil simpul yang dipilih untuk dikembangkan lebih lanjut, ada kemungkinan jalur solusi optimal terbuang pada tahap awal jika simpul yang mengarah ke solusi tersebut tidak termasuk dalam beam. Dengan demikian, Beam Search cenderung lebih mengutamakan efisiensi waktu dan penggunaan memori dibandingkan menjamin solusi terbaik atau paling pendek.

BAB III:
ANALISIS

3.1. Fungsi Evaluasi

Fungsi evaluasi dalam algoritma pencarian jalur adalah fungsi yang digunakan untuk menilai seberapa baik suatu simpul dalam proses pencarian solusi. Fungsi ini berperan menentukan prioritas simpul yang akan diperluas dengan menggabungkan biaya yang sudah dikeluarkan untuk mencapai simpul tersebut dan perkiraan biaya yang diperlukan untuk mencapai tujuan. Nilai g(n) adalah biaya kumulatif yang telah dikeluarkan untuk mencapai simpul n dari titik awal, mencerminkan total langkah atau sumber daya yang telah digunakan sepanjang jalur tersebut. Sedangkan nilai h(n) adalah fungsi heuristik yang memberikan estimasi biaya atau jarak dari simpul n menuju simpul tujuan, digunakan untuk memandu pencarian agar lebih terarah.

Pada algoritma-algoritma yang diimplementasikan, perbedaan utama antara ketiga algoritma utama (UCS, GBFS, dan A*) terletak pada fungsi evaluasi f(n):

1. Uniform Cost Search

   Pada UCS, algoritma hanya mempertimbangkan biaya yang dikeluarkan untuk mencapai simpul saat ini. Algoritma ini termasuk kategori *uninformed search* yang artinya tidak mendapatkan informasi tambahan. Sebagai konsekuensinya, algoritma ini tidak menggunakan fungsi heuristik sama sekali. Fungsi evaluasi dari algoritma UCS didefinisikan sebagai:

   $$f(n) = g(n)$$

   di mana f(n) adalah fungsi evaluasi dan g(n) adalah biaya yang dikeluarkan sejauh ini.

   Dalam konteks permainan Rush Hour, biaya biasa didefinisikan sebagai jumlah langkah yang telah dilakukan dari simpul awal ke simpul saat ini. Artinya selama proses pencarian, nilai dari g(n) akan ekivalen dengan nilai kedalaman simpul pencarian. Dengan menggunakan pendekatan ini, UCS dapat menjamin menemukan jalur dengan biaya minimum (optimal), meskipun pencarian bisa memakan waktu lebih lama dibanding algoritma yang memanfaatkan heuristik.

2. Greedy Best First Search

   Pada GBFS, algoritma hanya mempertimbangkan fungsi heuristik untuk memperkirakan biaya sejauh ini, tanpa mempertimbangkan biaya sebelumnya. GBFS fokus untuk mempercepat pencarian dengan mengikuti jalur yang tampak paling dekat atau paling menjanjikan menuju solusi berdasarkan perkiraan heuristik. Fungsi evaluasi dari algoritma GBFS didefinisikan sebagai:

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

   di mana f(n) adalah fungsi evaluasi dan h(n) adalah nilai estimasi dari fungsi heuristik.

3. A*

   Algoritma A* adalah gabungan dari keunggulan UCS dan GBFS. Algoritma ini memanfaatkan biaya yang telah diakumulasikan sejauh ini serta nilai estimasi yang didapatkan dari fungsi heuristik. Ini membuat pencarian lebih

terarah dan efisien, sekaligus menjamin solusi yang ditemukan adalah solusi optimal jika heuristiknya *admissible*. Fungsi evaluasi dari algoritma A* didefinisikan sebagai:

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

di mana f(n) adalah fungsi evaluasi, g(n) adalah biaya yang dikeluarkan sejauh ini, dan h(n) adalah nilai estimasi dari fungsi heuristik. Seperti UCS, g(n) dihitung sebagai jumlah langkah yang sudah terjadi sampai simpul tersebut.

4. Beam Search

   Sama seperti algoritma GBFS, Beam Search mempertimbangkan hanya estimasi dari fungsi heuristik h(n). Beam Search berfokus pada efisiensi waktu dan memori dengan membatasi jumlah simpul yang diperluas. Fungsi evaluasi dari algoritma Beam Search didefinisikan sebagai:

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

di mana f(n) adalah fungsi evaluasi dan h(n) adalah nilai estimasi dari fungsi heuristik.

3.2. Ke-*admissible*-an Fungsi Heuristik

Suatu fungsi heuristik dikatakan *admissible* jika nilai yang dihasilkannya tidak ada yang melebih-lebihkan biaya sebenarnya. Secara matematis, fungsi heuristik h(n) dikatakan *admissible* jika:

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

di mana h*(n) adalah biaya aktual terkecil dari simpul n menuju simpul tujuan. Pada implementasi ini, terdapat 5 heuristik yang akan diuji, yaitu *distance to exit* (jarak ke titik keluar), *blocking cars* (mobil yang memblokir jalan), *recursive blocking cars* (mobil yang memblokir jalan secara rekursif), dan *move needed estimate* (perkiraan pergerakan yang diperlukan), serta ada heuristik yang merupakan kombinasi.

1. Distance To Exit

   Heuristik ini menentukan jarak antara mobil target dengan pintu keluar dalam kolom atau baris, dengan mempertimbangkan orientasi dari mobil. Jika satu pergerakan dihitung sebagai satu blok, maka heuristik ini *admissible* karena nilainya tidak akan melebihi biaya aslinya. Heuristik ini tidak mempertimbangkan jika ada mobil lain yang menghalang jalannya, akibatnya nilainya akan selalu lebih kecil atau sama dengan biaya aslinya. Namun jika pergerakan lebih dari satu blok dalam satu gerakan dihtiung sebagai satu, maka fungsi heuristik ini bisa melebih-lebihkan biaya dan menjadi tidak *admissible*.

2. Blocking Cars

   Heuristik ini mempertimbangkan jumlah mobil yang langsung memblokir jalan mobil ke jalan keluar. Heuristik ini *admissible* karena menghitung jumlah mobil yang memblokir adalah batas bawah jumlah gerakan yang perlu dilakukan. Jika terdapat n mobil yang memblokir, maka terdapat paling n pergerakan untuk membuat mobil target tidak terblokir. Heuristik ini tidak memperhitungkan

pergerakan lebih yang diperlukan atau pemblokir lain yang memblokir mobil pemblokir.

3. Recursive Blocking Cars

   Heuristik ini adalah perkembangan dari heuristik sebelumnya. Secara rekursif, fungsi heuristik ini menghitung jumlah mobil pemblokir, dan juga pemblokir dari pemblokir tersebut dengan mempertimbangkan pergerakan maju dan mundur dari mobil. Heuristik ini *admissible* karena fungsi ini mengasumsikan bahwa setiap mobil setidaknya harus bergerak setidaknya sekali agar tidak terblokir lagi dan mengasumsikan tidak akan perlu pergerakan lagi. Ini akan meremehkan biaya sebenarnya yang diperlukan karena nilainya pasti akan lebih kecil atau mengimbangi nilai sebenarnya.

4. Move Needed Estimate

   Heuristik ini tidak jauh berbeda dari heuristik Recursive Blocking Cars. Heuristik ini mengestimasikan jumlah pergerakan minimal yang perlu dilakukan agar mobil tidak terblokir. Sama seperti heuristik sebelumnya, heuristik ini *admissible* karena hanya mengakumulasikan apa yang diperkirakan sebagai nilai minimal. Karena algoritmanya terkait dengan *sliding*, bisa saja mengabaikan interaksi yang lebih kompleks sehingga fungsi optimistik dan tidak melebih-lebihkan biaya.

3.3. Perbandingan Algoritma

1. Perbandingan Algoritma UCS dan BFS

   Algoritma Breadth First Search (BFS) adalah algoritma pencariaan yang menjelajahi semua node pada level kedalaman tertentu sebelum melanjutkan ke level kedalaman berikutnya. Karena pencarian dilakukan *level by level*, maka algoritma BFS dapat digunakan untuk mencari jalur dengan jumlah langkah minimum. Algoritma UCS melakukan pencarian dengan menggunakan *priority queue* untuk memperluas simpul dengan biaya terkecil sejauh ini.

   Pada implementasi UCS ini, setiap pergerakan akan memiliki biaya sebesar satu. Jika setiap langkah pada graf memiliki biaya yang sama, maka algoritma UCS dan BFS akan melakukan ekspansi simpul dengan urutan yang sama. Karena setiap pergerakan memiliki biaya satu dan setiap anak dari simpul adalah setiap gerakan sahih yang bisa dilakukan dari konfigurasi saat ini, maka tiap simpul yang dihasilkan oleh simpul tersebut semuanya akan memiliki biaya yang sama, yang pasti melebihi setiap simpul yang ada pada level kedalaman sekarang.

   Karena algoritma UCS menggunakan *priority queue*, maka suatu simpul akan diproses berdasarkan nilai f(n). Namun karena tiap simpul yang dihasilkan pasti memiliki *priority* yang lebih tinggi, maka tiap level kedalaman pada akhirnya akan diproses terlebih dahulu sebelum lanjut ke level berikutnya, Ini adalah perilaku yang sama dengan algoritma BFS. Agar algoritma UCS tidak

menghasilkan hasil yang sama dengan algoritma BFS, maka tiap langkah tidak bisa memiliki biaya yang sama.

2. Perbandingan Algoritma A* dan BFS

   Secara teoritis, algoritma A* biasanya akan lebih cepat dibandingkan dengan algoritma UCS. Algoritma UCS sepenuhnya menggunakan nilai dari g(n) dan selalu memperluas simpul dengan biaya terkecil sejauh ini. Akibatnya, algoritma UCS akan memproses lebih banyak simpul tanpa pengaruh luar yang mempengaruhi eksplorasinya.

   Algoritma A* juga menggunakan nilai dari g(n), namun juga menggunakan fungsi heuristik h(n) yang bertujuan untuk memandu pencarian ke arah solusi yang dianggap menjanjikan. Akibatnya, algoritma dapat menghindari eksplorasi ke dalam cabang yang irrelevan dan jika heuristiknya *admissible*, maka solusinya akan optimal.

   Namun, penggunaan algoritma A* tidak selalu lebih cepat dibanding dengan algoritma UCS. Efisiensi dari A* sangat bergantung pada kualitas heuristiknya. Jika heuristik yang digunakan *admissible* dan informatif, maka A* akan sangat efisien. Namun jika heuristiknya buruk, maka performa dari A* bisa menurun drastis. Selain itu, fungsi heuristik yang sangat mahal untuk dihitung dapat membuat A* menjadi lambat meskipun secara teori lebih efisien dalam jumlah simpul yang diperluas.

3. Analisis Optimalitas Algoritma GBFS

   Secara teoritis, algoritma GBFS tidak menjamin solusinya adalah solusi yang optimal, atau bahkan solusi yang ada. Hal ini disebabkan karena GBFS hanya menggunakan fungsi heuristik h(n) untuk memilih node mana yang akan diperluas berikutnya, yaitu node yang tampak paling dekat atau paling menjanjikan menuju tujuan berdasarkan estimasi heuristik. Karena itu, GBFS cenderung mengikuti jalur yang dipandangnya baik menurut heuristiknya.

   Akibatnya, GBFS rawan terjebak dalam solusi non-optimal atau jalan yang buntu, tergantung pada kualitas heuristik. Berdasarkan prinsip *greedy*-nya, algoritma ini menentukan yang mana paling baik berdasarkan fungsi heuristiknya dan berharap bahwa simpul itu akan membawanya ke suatu solusi. Meskipun GBFS seringkali adalah algoritma yang cepat, dia menukar kecepatan ini dengan ketepatan sehingga tidak ada jaminan bahwa hasil yang dihasilkan optimal.

BAB IV:
SOURCE PROGRAM

4.1. Algorithms

A*

```
import { collapseBoard, getAllValidMoves, isSolutionFound, movePiece } from
"../boardUtils";
import { PrioQueue } from "../prioQueue";
import type { Board, PieceMap, Move, Node} from "../types"

export const aStar = (
    board: Board,
    pieces: PieceMap,
    heuristicFunction: (board: Board, pieces: PieceMap) => number
): {found: boolean, moveHistory: Move[], nodesVisited: number, timeTaken:
number} => {
    const startTime = performance.now();
    let nodesVisited = 0;

    const openList = new PrioQueue<Node>((a, b) => a.f - b.f);
    const closedList = new Map<string, number>();
    const initialG = 0;
    const initialH = heuristicFunction(board, pieces);

    const initialNode: Node = {
        board: board,
        pieces: pieces,
        f: initialG + initialH,
        g: initialG,
        h: initialH,
        moveHistory: []
    };

    openList.push(initialNode);

    while (!openList.isEmpty()) {
```

```
        const currentNode = openList.pop();
        if (!currentNode) {
            continue;
        }

        nodesVisited++;

        const boardKey = collapseBoard(currentNode.board);
        if (closedList.has(boardKey) && closedList.get(boardKey)! <=
currentNode.g!) {
            continue;
        }
        closedList.set(boardKey, currentNode.g!);

        if (isSolutionFound(currentNode.pieces)) {
            const timeTaken = performance.now() - startTime;
            return {
                found: true,
                moveHistory: currentNode.moveHistory,
                nodesVisited,
                timeTaken
            };
        }

        const validMoves = getAllValidMoves(currentNode.board,
currentNode.pieces);

        for (const validMove of validMoves) {
            const { board: newBoard, pieces: newPieces } =
movePiece(currentNode.board, currentNode.pieces, validMove);
            const newBoardKey = collapseBoard(newBoard);
            const newGValue = currentNode.g! + 1;
            const newHValue = heuristicFunction(newBoard, newPieces);
            const newFValue = newGValue + newHValue;
            if (closedList.has(newBoardKey) && closedList.get(newBoardKey)! <=
newGValue) {
                continue;
            }
```

```
            const newNode: Node = {
                board: newBoard,
                pieces: newPieces,
                f: newFValue,
                g: newGValue,
                h: newHValue,
                moveHistory: [...currentNode.moveHistory, validMove]
            };
            openList.push(newNode);
        }
    }

    const timeTaken = performance.now() - startTime;
    return {
        found: false,
        moveHistory: [],
        nodesVisited,
        timeTaken
    };
}
```

GBFS

```
import { collapseBoard, getAllValidMoves, isSolutionFound, movePiece } from
"../boardUtils";
import { PrioQueue } from "../prioQueue";
import type { Board, PieceMap, Move, Node} from "../types"

export const gbfs = (
    board: Board,
    pieces: PieceMap,
    heuristicFunction: (board: Board, pieces: PieceMap) => number
): {found: boolean, moveHistory: Move[], nodesVisited: number, timeTaken:
number} => {
    const startTime = performance.now();
```

```
    let nodesVisited = 0;

    const openList = new PrioQueue<Node>((a, b) => a.f - b.f);
    const closedList = new Set<string>();
    const initialH = heuristicFunction(board, pieces);

    const initialNode: Node = {
        board: board,
        pieces: pieces,
        f: initialH,
        h: initialH,
        moveHistory: []
    };

    openList.push(initialNode);

    while (!openList.isEmpty()) {
        const currentNode = openList.pop();
        if (!currentNode) {
            continue;
        }

        nodesVisited++;

        const boardKey = collapseBoard(currentNode.board);
        if (closedList.has(boardKey)) {
            continue;
        }
        closedList.add(boardKey);

        if (isSolutionFound(currentNode.pieces)) {
            const timeTaken = performance.now() - startTime;
            return {
                found: true,
                moveHistory: currentNode.moveHistory,
                nodesVisited,
                timeTaken
            };
```

```
        }

        const validMoves = getAllValidMoves(currentNode.board,
currentNode.pieces);

        for (const validMove of validMoves) {
            const { board: newBoard, pieces: newPieces } =
movePiece(currentNode.board, currentNode.pieces, validMove);
            const newBoardKey = collapseBoard(newBoard);
            const newHValue = heuristicFunction(newBoard, newPieces);
            const newFValue = newHValue;
            if (closedList.has(newBoardKey)) {
                continue;
            }

            const newNode: Node = {
                board: newBoard,
                pieces: newPieces,
                f: newFValue,
                h: newHValue,
                moveHistory: [...currentNode.moveHistory, validMove]
            };
            openList.push(newNode);
        }
    }

    const timeTaken = performance.now() - startTime;
    return {
        found: false,
        moveHistory: [],
        nodesVisited,
        timeTaken
    };
}
```

UCS

```
import { collapseBoard, getAllValidMoves, isSolutionFound, movePiece } from
"../boardUtils";
import { PrioQueue } from "../prioQueue";
import type { Board, PieceMap, Move, Node} from "../types"

export const ucs = (
    board: Board,
    pieces: PieceMap
): {found: boolean, moveHistory: Move[], nodesVisited: number, timeTaken:
number} => {
    const startTime = performance.now();
    let nodesVisited = 0;

    const openList = new PrioQueue<Node>((a, b) => a.f - b.f);
    const closedList = new Map<string, number>();
    const initialG = 0;

    const initialNode: Node = {
        board: board,
        pieces: pieces,
        f: initialG,
        g: initialG,
        moveHistory: []
    };

    openList.push(initialNode);

    while (!openList.isEmpty()) {
        const currentNode = openList.pop();
        if (!currentNode) {
            continue;
        }

        nodesVisited++;

        const boardKey = collapseBoard(currentNode.board);
```

```
        if (closedList.has(boardKey) && closedList.get(boardKey)! <=
currentNode.g!) {
            continue;
        }
        closedList.set(boardKey, currentNode.g!);

        if (isSolutionFound(currentNode.pieces)) {
            const timeTaken = performance.now() - startTime;
            return {
                found: true,
                moveHistory: currentNode.moveHistory,
                nodesVisited,
                timeTaken
            };
        }

        const validMoves = getAllValidMoves(currentNode.board,
currentNode.pieces);

        for (const validMove of validMoves) {
            const { board: newBoard, pieces: newPieces } =
movePiece(currentNode.board, currentNode.pieces, validMove);
            const newBoardKey = collapseBoard(newBoard);
            const newGValue = currentNode.g! + 1;
            const newFValue = newGValue;
            if (closedList.has(newBoardKey) && closedList.get(newBoardKey)! <=
newGValue) {
                continue;
            }

            const newNode: Node = {
                board: newBoard,
                pieces: newPieces,
                f: newFValue,
                g: newGValue,
                moveHistory: [...currentNode.moveHistory, validMove]
            };
            openList.push(newNode);
```

```
        }
    }

    const timeTaken = performance.now() - startTime;
    return {
        found: false,
        moveHistory: [],
        nodesVisited,
        timeTaken
    };
}
```

Beam Search

```
import { collapseBoard, getAllValidMoves, isSolutionFound, movePiece } from
"../boardUtils";
import type { Board, PieceMap, Move, Node } from "../types";

export const beamSearch = (
    board: Board,
    pieces: PieceMap,
    heuristicFunction: (board: Board, pieces: PieceMap) => number,
    beamWidth: number
): { found: boolean, moveHistory: Move[], nodesVisited: number, timeTaken:
number } => {
    const startTime = performance.now();
    let nodesVisited = 0;

    const initialH = heuristicFunction(board, pieces);
    const initialNode: Node = {
        board,
        pieces,
        f: initialH,
        h: initialH,
        moveHistory: []
    };
```

```
    let beam: Node[] = [initialNode];
    const visited = new Set<string>();

    while (beam.length > 0) {
        const nextBeam: Node[] = [];

        for (const currentNode of beam) {
            nodesVisited++;
            const boardKey = collapseBoard(currentNode.board);
            if (visited.has(boardKey)) continue;
            visited.add(boardKey);

            if (isSolutionFound(currentNode.pieces)) {
                const timeTaken = performance.now() - startTime;
                return {
                    found: true,
                    moveHistory: currentNode.moveHistory,
                    nodesVisited,
                    timeTaken
                };
            }

            const validMoves = getAllValidMoves(currentNode.board,
currentNode.pieces);

            for (const validMove of validMoves) {
                const { board: newBoard, pieces: newPieces } =
movePiece(currentNode.board, currentNode.pieces, validMove);
                const newKey = collapseBoard(newBoard);
                if (visited.has(newKey)) continue;

                const newH = heuristicFunction(newBoard, newPieces);
                const newNode: Node = {
                    board: newBoard,
                    pieces: newPieces,
                    f: newH,
                    h: newH,
```

```
                    moveHistory: [...currentNode.moveHistory, validMove]
                };

                nextBeam.push(newNode);
            }
        }

        nextBeam.sort((a, b) => a.h! - b.h!);
        beam = nextBeam.slice(0, beamWidth);
    }

    const timeTaken = performance.now() - startTime;
    return {
        found: false,
        moveHistory: [],
        nodesVisited,
        timeTaken
    };
};
```

4.2. Heuristics

Blocking Cars

```
import type { Board, PieceMap } from "../types"

export const blockingCars = (board: Board, pieces: PieceMap): number => {
    const primaryPiece = pieces["P"];
    const exitPiece = pieces["K"];

    if (!primaryPiece || !exitPiece || !primaryPiece.pos || !exitPiece.pos) {
        return 0;
    }

    const blockers = new Set<string>();
```

```
    const height = board.height;
    const width = board.width;

    if (primaryPiece.orientation === "Horizontal") {
        const row = primaryPiece.pos.row;
        const startCol = primaryPiece.pos.col;
        const endCol = startCol + primaryPiece.size - 1;

        if (exitPiece.pos.col === width) {
            for (let c = endCol + 1; c < width; c++) {
                const cell = board.grid[row][c];
                if (cell !== ".") blockers.add(cell);
            }
        } else if (exitPiece.pos.col === -1) {
            for (let c = 0; c < startCol; c++) {
                const cell = board.grid[row][c];
                if (cell !== ".") blockers.add(cell);
            }
        }
    } else {
        const col = primaryPiece.pos.col;
        const startRow = primaryPiece.pos.row;
        const endRow = startRow + primaryPiece.size - 1;

        if (exitPiece.pos.row === height) {
            for (let r = endRow + 1; r < height; r++) {
                const cell = board.grid[r][col];
                if (cell !== ".") blockers.add(cell);
            }
        } else if (exitPiece.pos.row === -1) {
            for (let r = 0; r < startRow; r++) {
                const cell = board.grid[r][col];
                if (cell !== ".") blockers.add(cell);
            }
        }
    }

    return blockers.size;
```

```
};
```

Combined Heuristic

```
import type { Board, PieceMap } from "../types";
import { blockingCars } from "./blockingCars";
import { distanceToExit } from "./distanceToExit";
import { moveNeededEstimate } from "./moveNeededEstimate";

export const combinedHeuristic = (board: Board, pieces: PieceMap): number => {
    return distanceToExit(board, pieces) + blockingCars(board, pieces) + 0.5 *
moveNeededEstimate(board, pieces);
}
```

Distance To Exit

```
import type { Board, PieceMap } from "../types";

export const distanceToExit = (_board: Board, pieces: PieceMap): number => {
    const primaryPiece = pieces["P"];
    const exitPiece = pieces["K"];
    if (!primaryPiece || !exitPiece) {
        return Infinity;
    }

    if (primaryPiece.orientation === "Horizontal") {
        if (primaryPiece.pos.row !== exitPiece.pos.row) {
            return Infinity;
        }

        const startCol = primaryPiece.pos.col;
        const endCol = startCol + primaryPiece.size - 1;
        const exitCol = exitPiece.pos.col;

        if (exitCol < startCol) {
            return startCol - exitCol;
```

```
        } else {
            return exitCol - endCol;
        }

    } else if (primaryPiece.orientation === "Vertical") {
        if (primaryPiece.pos.col !== exitPiece.pos.col) {
            return Infinity;
        }

        const startRow = primaryPiece.pos.row;
        const endRow = startRow + primaryPiece.size - 1;
        const exitRow = exitPiece.pos.row;

        if (exitRow < startRow) {
            return startRow - exitRow;
        } else {
            return exitRow - endRow;
        }
    }

    return Infinity;
};
```

Move estimation

```
import type { Board, Piece, PieceMap } from "../types";

export const moveNeededEstimate = (board: Board, pieces: PieceMap): number => {
    const visited = new Set<string>();
    const primaryPiece = pieces["P"];
    if (!primaryPiece) return 0;

    visited.add(primaryPiece.id);
    let total = 0;

    const directBlockers = getDirectBlockers(board, pieces, primaryPiece);
```

```
    for (const blocker of directBlockers) {
        const current = pieces[blocker];
        const spaceForward = estimateSpaceNeeded(primaryPiece, current, true);
        const spaceBackward = estimateSpaceNeeded(primaryPiece, current,
false);
        total += getRecursiveBlockValue(current, spaceForward, spaceBackward,
board, pieces, visited);
    }

    return total;
};

const getRecursiveBlockValue = (
    piece: Piece,
    spaceForward: number,
    spaceBackward: number,
    board: Board,
    pieces: PieceMap,
    visited: Set<string>
): number => {
    visited.add(piece.id);
    let cost = Math.min(spaceForward, spaceBackward);

    for (const other of Object.values(pieces)) {
        if (visited.has(other.id) || other.id === piece.id || other.id === "K")
{
            continue;
        }
        if (!isPhysicallyBlocking(piece, other, spaceForward, spaceBackward)) {
            continue;
        }

        const canForward = canSlide(piece, other, spaceForward, board, true);
        const canBackward = canSlide(piece, other, spaceBackward, board,
false);

        const needForward = estimateSpaceNeeded(piece, other, true);
        const needBackward = estimateSpaceNeeded(piece, other, false);
```

```
        let valueForward = Number.MAX_SAFE_INTEGER;
        let valueBackward = Number.MAX_SAFE_INTEGER;

        if (!canForward) {
            valueForward = getRecursiveBlockValue(other, needForward,
needBackward, board, pieces, visited);
        }
        if (!canBackward) {
            valueBackward = getRecursiveBlockValue(other, needForward,
needBackward, board, pieces, visited);
        }

        cost += Math.min(valueForward, valueBackward);
    }
    return cost;
};

const estimateSpaceNeeded = (piece: Piece, blocker: Piece, forward: boolean):
number => {
    if (piece.orientation === blocker.orientation) {
        return 1;
    }
    const pieceFixed = piece.orientation === "Horizontal" ? piece.pos.row :
piece.pos.col;
    const blockStart = piece.orientation === "Horizontal" ? blocker.pos.row :
blocker.pos.col;
    const blockEnd = blockStart + blocker.size - 1;

    if (forward) {
        return Math.max(0, blockEnd - pieceFixed + 1);
    } else {
        return Math.max(0, pieceFixed - blockStart + 1);
    }
};

const canSlide = (
    car: Piece,
```

```
    blocker: Piece,
    needed: number,
    board: Board,
    forward: boolean
): boolean => {
    if (car.orientation !== blocker.orientation) return false;
    if (!blocker.orientation || blocker.orientation === "Unknown") return
false;

    const width = board.width;
    const height = board.height;

    if (blocker.orientation === "Horizontal") {
        const leftmostCol = blocker.pos.col;
        const rightmostCol = blocker.pos.col + blocker.size - 1;

        if (forward) {
            let step = 1;
            while (rightmostCol + step < width) {
                const cell = board.grid[blocker.pos.row][rightmostCol + step];
                if (cell === ".") {
                    if (step >= needed) return true;
                    step++;
                } else {
                    break;
                }
            }
        } else {
            let step = 1;
            while (leftmostCol - step >= 0) {
                const cell = board.grid[blocker.pos.row][leftmostCol - step];
                if (cell === ".") {
                    if (step >= needed) return true;
                    step++;
                } else {
                    break;
                }
            }
```

```
        }
    } else if (blocker.orientation === "Vertical") {
        const topmostRow = blocker.pos.row;
        const bottommostRow = blocker.pos.row + blocker.size - 1;

        if (forward) {
            let step = 1;
            while (bottommostRow + step < height) {
                const cell = board.grid[bottommostRow + step][blocker.pos.col];
                if (cell === ".") {
                    if (step >= needed) return true;
                    step++;
                } else {
                    break;
                }
            }
        } else {
            let step = 1;
            while (topmostRow - step >= 0) {
                const cell = board.grid[topmostRow - step][blocker.pos.col];
                if (cell === ".") {
                    if (step >= needed) return true;
                    step++;
                } else {
                    break;
                }
            }
        }
    }

    return false;
};


const isPhysicallyBlocking = (
    piece: Piece,
    other: Piece,
    spaceForward: number,
```

```
    spaceBackward: number
): boolean => {
    const occupied = new Set<string>();

    for (let i = 0; i < other.size; i++) {
        const r = other.orientation === "Horizontal" ? other.pos.row :
other.pos.row + i;
        const c = other.orientation === "Horizontal" ? other.pos.col + i :
other.pos.col;
        occupied.add(`${r},${c}`);
    }

    for (let i = 1; i <= spaceForward; i++) {
        for (let j = 0; j < piece.size; j++) {
            const r = piece.orientation === "Horizontal" ? piece.pos.row :
piece.pos.row + j + i;
            const c = piece.orientation === "Horizontal" ? piece.pos.col + j +
i : piece.pos.col;
            const key = `${r},${c}`;
            if (occupied.has(key)) return true;
        }
    }

    for (let i = 1; i <= spaceBackward; i++) {
        for (let j = 0; j < piece.size; j++) {
            const r = piece.orientation === "Horizontal" ? piece.pos.row :
piece.pos.row + j - i;
            const c = piece.orientation === "Horizontal" ? piece.pos.col + j -
i : piece.pos.col;
            const key = `${r},${c}`;
            if (occupied.has(key)) return true;
        }
    }

    return false;
};
```

```
const getDirectBlockers = (board: Board, pieces: PieceMap, piece: Piece):
Set<string> => {
    const exitPiece = pieces["K"];
    const blockers = new Set<string>();
    const height = board.height;
    const width = board.width;

    if (!exitPiece || !exitPiece.pos) {
        return blockers;
    }

    if (piece.orientation === "Horizontal") {
        const row = piece.pos.row;
        const startCol = piece.pos.col;
        const endCol = startCol + piece.size - 1;

        if (exitPiece.pos.col === width) {
            for (let c = endCol + 1; c < width; c++) {
                const cell = board.grid[row][c];
                if (cell !== ".") blockers.add(cell);
            }
        } else if (exitPiece.pos.col === -1) {
            for (let c = 0; c < startCol; c++) {
                const cell = board.grid[row][c];
                if (cell !== ".") blockers.add(cell);
            }
        }
    } else {
        const col = piece.pos.col;
        const startRow = piece.pos.row;
        const endRow = startRow + piece.size - 1;

        if (exitPiece.pos.row === height) {
            for (let r = endRow + 1; r < height; r++) {
                const cell = board.grid[r][col];
                if (cell !== ".") blockers.add(cell);
            }
        } else if (exitPiece.pos.row === -1) {
```

```
            for (let r = 0; r < startRow; r++) {
                const cell = board.grid[r][col];
                if (cell !== ".") blockers.add(cell);
            }
        }
    }

    return blockers;
};
```

Recursive Blockers

```
import type { Board, Piece, PieceMap } from "../types";

export const recursiveBlockers = (board: Board, pieces: PieceMap): number => {
    const visited = new Set<string>();
    const primaryPiece = pieces["P"];
    if (!primaryPiece) return 0;

    visited.add(primaryPiece.id);
    let total = 0;

    const directBlockers = getDirectBlockers(board, pieces, primaryPiece);

    for (const blocker of directBlockers) {
        const current = pieces[blocker];
        const spaceForward = estimateSpaceNeeded(primaryPiece, current, true);
        const spaceBackward = estimateSpaceNeeded(primaryPiece, current,
false);
        total += getRecursiveBlockValue(current, spaceForward, spaceBackward,
board, pieces, visited);
    }

    return total;
};

const getRecursiveBlockValue = (
```

```
    piece: Piece,
    spaceForward: number,
    spaceBackward: number,
    board: Board,
    pieces: PieceMap,
    visited: Set<string>
): number => {
    visited.add(piece.id);
    let value = 1;

    for (const other of Object.values(pieces)) {
        if (visited.has(other.id) || other.id === piece.id || other.id === "K")
{
            continue;
        }
        if (!isPhysicallyBlocking(piece, other, spaceForward, spaceBackward)) {
            continue;
        }

        const canForward = canSlide(piece, other, spaceForward, board, true);
        const canBackward = canSlide(piece, other, spaceBackward, board,
false);

        const needForward = estimateSpaceNeeded(piece, other, true);
        const needBackward = estimateSpaceNeeded(piece, other, false);

        let valueForward = Number.MAX_SAFE_INTEGER;
        let valueBackward = Number.MAX_SAFE_INTEGER;

        if (!canForward) {
            valueForward = getRecursiveBlockValue(other, needForward,
needBackward, board, pieces, visited);
        }
        if (!canBackward) {
            valueBackward = getRecursiveBlockValue(other, needForward,
needBackward, board, pieces, visited);
        }
```

```
        value += Math.min(valueForward, valueBackward);
    }
    return value;
};

const estimateSpaceNeeded = (piece: Piece, blocker: Piece, forward: boolean):
number => {
    if (piece.orientation === blocker.orientation) {
        return 1;
    }
    const pieceFixed = piece.orientation === "Horizontal" ? piece.pos.row :
piece.pos.col;
    const blockStart = piece.orientation === "Horizontal" ? blocker.pos.row :
blocker.pos.col;
    const blockEnd = blockStart + blocker.size - 1;

    if (forward) {
        return Math.max(0, blockEnd - pieceFixed + 1);
    } else {
        return Math.max(0, pieceFixed - blockStart + 1);
    }
};

const canSlide = (
    car: Piece,
    blocker: Piece,
    needed: number,
    board: Board,
    forward: boolean
): boolean => {
    if (car.orientation !== blocker.orientation) return false;
    if (!blocker.orientation || blocker.orientation === "Unknown") return
false;

    const width = board.width;
    const height = board.height;

    if (blocker.orientation === "Horizontal") {
```

```
        const leftmostCol = blocker.pos.col;
        const rightmostCol = blocker.pos.col + blocker.size - 1;

        if (forward) {
            let step = 1;
            while (rightmostCol + step < width) {
                const cell = board.grid[blocker.pos.row][rightmostCol + step];
                if (cell === ".") {
                    if (step >= needed) return true;
                    step++;
                } else {
                    break;
                }
            }
        } else {
            let step = 1;
            while (leftmostCol - step >= 0) {
                const cell = board.grid[blocker.pos.row][leftmostCol - step];
                if (cell === ".") {
                    if (step >= needed) return true;
                    step++;
                } else {
                    break;
                }
            }
        }
    } else if (blocker.orientation === "Vertical") {
        const topmostRow = blocker.pos.row;
        const bottommostRow = blocker.pos.row + blocker.size - 1;

        if (forward) {
            let step = 1;
            while (bottommostRow + step < height) {
                const cell = board.grid[bottommostRow + step][blocker.pos.col];
                if (cell === ".") {
                    if (step >= needed) return true;
                    step++;
                } else {
```

```
                    break;
                }
            }
        } else {
            let step = 1;
            while (topmostRow - step >= 0) {
                const cell = board.grid[topmostRow - step][blocker.pos.col];
                if (cell === ".") {
                    if (step >= needed) return true;
                    step++;
                } else {
                    break;
                }
            }
        }
    }

    return false;
};


const isPhysicallyBlocking = (
    piece: Piece,
    other: Piece,
    spaceForward: number,
    spaceBackward: number
): boolean => {
    const occupied = new Set<string>();

    for (let i = 0; i < other.size; i++) {
        const r = other.orientation === "Horizontal" ? other.pos.row :
other.pos.row + i;
        const c = other.orientation === "Horizontal" ? other.pos.col + i :
other.pos.col;
        occupied.add(`${r},${c}`);
    }

    for (let i = 1; i <= spaceForward; i++) {
```

```
        for (let j = 0; j < piece.size; j++) {
            const r = piece.orientation === "Horizontal" ? piece.pos.row :
piece.pos.row + j + i;
            const c = piece.orientation === "Horizontal" ? piece.pos.col + j +
i : piece.pos.col;
            const key = `${r},${c}`;
            if (occupied.has(key)) return true;
        }
    }

    for (let i = 1; i <= spaceBackward; i++) {
        for (let j = 0; j < piece.size; j++) {
            const r = piece.orientation === "Horizontal" ? piece.pos.row :
piece.pos.row + j - i;
            const c = piece.orientation === "Horizontal" ? piece.pos.col + j -
i : piece.pos.col;
            const key = `${r},${c}`;
            if (occupied.has(key)) return true;
        }
    }

    return false;
};


const getDirectBlockers = (board: Board, pieces: PieceMap, piece: Piece):
Set<string> => {
    const exitPiece = pieces["K"];
    const blockers = new Set<string>();
    const height = board.height;
    const width = board.width;

    if (!exitPiece || !exitPiece.pos) {
        return blockers;
    }

    if (piece.orientation === "Horizontal") {
        const row = piece.pos.row;
```

```
        const startCol = piece.pos.col;
        const endCol = startCol + piece.size - 1;

        if (exitPiece.pos.col === width) {
            for (let c = endCol + 1; c < width; c++) {
                const cell = board.grid[row][c];
                if (cell !== ".") blockers.add(cell);
            }
        } else if (exitPiece.pos.col === -1) {
            for (let c = 0; c < startCol; c++) {
                const cell = board.grid[row][c];
                if (cell !== ".") blockers.add(cell);
            }
        }
    } else {
        const col = piece.pos.col;
        const startRow = piece.pos.row;
        const endRow = startRow + piece.size - 1;

        if (exitPiece.pos.row === height) {
            for (let r = endRow + 1; r < height; r++) {
                const cell = board.grid[r][col];
                if (cell !== ".") blockers.add(cell);
            }
        } else if (exitPiece.pos.row === -1) {
            for (let r = 0; r < startRow; r++) {
                const cell = board.grid[r][col];
                if (cell !== ".") blockers.add(cell);
            }
        }
    }

    return blockers;
};
```

4.3. Helpers

Validasi input

```
import type { Car, Piece, EdgeGrid } from "../types";

export const parseFileContents = (content: string) => {
  const result: {
    success: boolean;
    message?: string;
    newCars?: Car[];
    width?: number;
    height?: number;
    exitGrid?: EdgeGrid;
  } = { success: false };

  try {
    const normalizedContent = content.replace(/\r\n/g, "\n").replace(/\r/g,
"\n");
    const lines = normalizedContent.trim().split("\n");

    if (lines.length < 1) {
      result.message = "File is empty!";
      return result;
    }

    const dimensions = lines[0].split(" ");
    if (dimensions.length !== 2) {
      result.message = "Invalid board dimensions format!";
      return result;
    }

    const width = parseInt(dimensions[0]);
    const height = parseInt(dimensions[1]);

    if (isNaN(width) || isNaN(height)) {
      result.message = "Board dimensions must be valid numbers!";
      return result;
    }
```

```
    if (width <= 0 || height <= 0) {
      result.message = "Board dimensions must be positive numbers!";
      return result;
    }

    const carCount = parseInt(lines[1].trim());
    if (isNaN(carCount)) {
      result.message = "Car count must be a valid number!";
      return result;
    }

    if (carCount > 24) {
      result.message = "Max 24 cars on board!";
      return result;
    }

    const newCars: Piece[] = [];
    const exitPiece: Piece = {
      id: "K",
      pos: { row: 0, col: 0 },
      orientation: "Horizontal",
      size: 1,
    };

    for (let i = 2; i < lines.length; i++) {
      if (!lines[i].trim()) continue;

      const parts = lines[i].split("");
      for (let j = 0; j < parts.length; j++) {
        let carExists = false;
        const char = parts[j];

        if (char === " " || char === "." || char === "\r" || char === "\n" ||
!char.trim()) {
          continue;
        }
```

```
        if (char === "K") {
          if (exitPiece.orientation !== "Unknown") {
            exitPiece.orientation = "Unknown";
            exitPiece.pos = { row: i - 2, col: j };
          } else {
            result.message = "Multiple Exits found!";
            return result;
          }
          continue;
        }

        if (!/^[A-Z]$/.test(char)) {
          result.message = `Invalid car ID: "${char}". IDs must be single
capital alphabetical characters.`;
          return result;
        }

        for (let k = 0; k < newCars.length; k++) {
          if (newCars[k].id === char) {
            carExists = true;
            break;
          }
        }

        if (!carExists) {
          const newPiece: Piece = {
            id: char,
            pos: { row: i - 2, col: j },
            orientation: "Unknown",
            size: 1,
          };
          newCars.push(newPiece);
        } else {
          const pieceIndex = newCars.findIndex((p) => p.id === char);
          if (pieceIndex !== -1) {
            const piece = newCars[pieceIndex];
            const currentRow = i - 2;
            const currentCol = j;
```

```
            const isVerticalConnection = currentRow === piece.pos.row + 1;
            const isHorizontalConnection = currentCol === piece.pos.col + 1;

            if (piece.orientation === "Unknown") {
              if (isHorizontalConnection) {
                piece.orientation = "Horizontal";
              } else if (isVerticalConnection) {
                piece.orientation = "Vertical";
              } else {
                result.message = `Multiple pieces with id ${piece.id} found!`;
                return result;
              }
            }

            if (piece.orientation === "Horizontal" && isVerticalConnection) {
              result.message = "Invalid piece shape!";
              return result;
            } else if (piece.orientation === "Vertical" &&
isHorizontalConnection) {
              result.message = "Invalid piece shape!";
              return result;
            }

            piece.size++;
            newCars[pieceIndex] = piece;
          }
        }
      }
    }

    if (newCars.length - 1 !== carCount) {
      result.message = `Numbers of cars not the same as input (found
${newCars.length}, expected ${carCount} + 1(primary car))`;
      return result;
    }

    for (let i = 0; i < newCars.length; i++) {
```

```
      if (newCars[i].size <= 1) {
        result.message = `Car ${newCars[i].id} has size below 2`;
        return result;
      }
    }

    if (exitPiece.orientation === "Horizontal" && exitPiece.pos.row === 0 &&
exitPiece.pos.col === 0) {
      result.message = "No exit marker (K) found in the puzzle!";
      return result;
    }

    const isExitTop = exitPiece.pos.row == 0 && lines[exitPiece.pos.row +
2].split("").length != width + 1;
    const isExitLeft = exitPiece.pos.col == 0 && lines[exitPiece.pos.row +
2].split("").length == width + 1;
    const isExitRight = exitPiece.pos.col == width && lines[exitPiece.pos.row +
2].split("").length == width + 1;
    const isExitBottom = exitPiece.pos.row == height && lines[exitPiece.pos.row
+ 2].split("").length != width + 1;

    const maxCol = lines.length - 2;
    let maxRow = 0;
    for (let i = 0; i < lines.length; i++) {
      if (lines[i].trim().split("").length > maxRow) {
        maxRow = lines[i].trim().split("").length;
      }
    }

    if (isExitTop) {
      for (let i = 3; i < lines.length; i++) {
        if (lines[i].trim().split("").length != width) {
          result.message = `Row ${i} doesn't have ${width} columns!`;
          return result;
        }
      }
      if (maxCol != height + 1) {
        console.log(maxCol);
```

```
        result.message = `Amount of rows not the same as input!`;
        return result;
      }
    } else if (isExitLeft) {
      for (let i = 2; i < lines.length; i++) {
        if (lines[i].split("").length != width + 1) {
          result.message = `Row ${i} doesn't have ${width} columns!`;
          return result;
        }
      }
      if (maxCol != height) {
        result.message = `Amount of rows not the same as input!`;
        return result;
      }
    } else if (isExitRight) {
      for (let i = 2; i < lines.length; i++) {
        if (i == exitPiece.pos.row + 2 && lines[i].split("").length != width +
1) {
          result.message = `Row ${i} contains exit but doesn't have ${width +
1} columns!`;
          return result;
        } else if (lines[i].trim().split("").length != width && i !=
exitPiece.pos.row + 2) {
          result.message = `Row ${i} doesn't have ${width} columns!`;
          return result;
        }
      }
      if (maxCol != height) {
        result.message = `Amount of rows not the same as input!`;
        return result;
      }
    } else if (isExitBottom) {
      for (let i = 2; i < lines.length - 1; i++) {
        if (lines[i].trim().split("").length != width) {
          if (i == exitPiece.pos.row - 2 && lines[i].trim().split("").length !=
width + 1) {
            result.message = `Row ${i} doesn't have ${width} columns!`;
            return result;
```

```
          }
        }
      }
      if (maxCol != height + 1) {
        result.message = `Amount of rows not the same as input!`;
        return result;
      }
    }

    if (!(isExitTop || isExitLeft || isExitRight || isExitBottom)) {
      result.message = `Exit must be at the edge!`;
      return result;
    }

    const gameCars: Car[] = newCars.map((piece) => ({
      id: piece.id,
      isVertical: piece.orientation === "Vertical",
      size: piece.size,
      initialLeft: isExitLeft ? piece.pos.col - 1 : piece.pos.col,
      initialTop: isExitTop ? piece.pos.row - 1 : piece.pos.row,
      isPrimary: piece.id === "P",
    }));

    const exitGrid = {
      row: isExitTop ? exitPiece.pos.row : exitPiece.pos.row + 1,
      col: isExitLeft ? exitPiece.pos.col : exitPiece.pos.col + 1,
    };

    result.success = true;
    result.newCars = gameCars;
    result.width = width;
    result.height = height;
    result.exitGrid = exitGrid;

    return result;
  } catch (error) {
    result.message = `Error parsing file: ${error instanceof Error ?
error.message : String(error)}`;
```

```
    return result;
  }
};
```

Menghasilkan output txt

```
import type { Car, Piece, EdgeGrid } from "../types";

export const parseFileContents = (content: string) => {
  const result: {
    success: boolean;
    message?: string;
    newCars?: Car[];
    width?: number;
    height?: number;
    exitGrid?: EdgeGrid;
  } = { success: false };

  try {
    const normalizedContent = content.replace(/\r\n/g, "\n").replace(/\r/g,
"\n");
    const lines = normalizedContent.trim().split("\n");

    if (lines.length < 1) {
      result.message = "File is empty!";
      return result;
    }

    const dimensions = lines[0].split(" ");
    if (dimensions.length !== 2) {
      result.message = "Invalid board dimensions format!";
      return result;
    }

    const width = parseInt(dimensions[0]);
    const height = parseInt(dimensions[1]);
```

```
    if (parseInt(lines[1].trim()) > 24) {
      result.message = "Max 24 cars on board!";
      return result;
    }

    const newCars: Piece[] = [];
    const exitPiece: Piece = {
      id: "K",
      pos: { row: 0, col: 0 },
      orientation: "Horizontal",
      size: 1,
    };

    for (let i = 2; i < lines.length; i++) {
      if (!lines[i].trim()) continue;

      const parts = lines[i].split("");
      for (let j = 0; j < parts.length; j++) {
        let carExists = false;
        const char = parts[j];

        if (char === " " || char === "." || char === "\r" || char === "\n" ||
!char.trim()) {
          continue;
        }

        if (char === "K") {
          if (exitPiece.orientation !== "Unknown") {
            exitPiece.orientation = "Unknown";
            exitPiece.pos = { row: i - 2, col: j };
          } else {
            result.message = "Multiple Exits found!";
            return result;
          }
          continue;
        }
```

```
        for (let k = 0; k < newCars.length; k++) {
          if (newCars[k].id === char) {
            carExists = true;
            break;
          }
        }

        if (!carExists) {
          const newPiece: Piece = {
            id: char,
            pos: { row: i - 2, col: j },
            orientation: "Unknown",
            size: 1,
          };
          newCars.push(newPiece);
        } else {
          const pieceIndex = newCars.findIndex((p) => p.id === char);
          if (pieceIndex !== -1) {
            const piece = newCars[pieceIndex];
            const currentRow = i - 2;
            const currentCol = j;

            const isVerticalConnection = currentRow === piece.pos.row + 1;
            const isHorizontalConnection = currentCol === piece.pos.col + 1;

            if (piece.orientation === "Unknown") {
              if (isHorizontalConnection) {
                piece.orientation = "Horizontal";
              } else if (isVerticalConnection) {
                piece.orientation = "Vertical";
              } else {
                result.message = `Multiple pieces with id ${piece.id} found!`;
                return result;
              }
            }

            if (piece.orientation === "Horizontal" && isVerticalConnection) {
              result.message = "Invalid piece shape!";
```

```
              return result;
            } else if (piece.orientation === "Vertical" &&
isHorizontalConnection) {
              result.message = "Invalid piece shape!";
              return result;
            }

            piece.size++;
            newCars[pieceIndex] = piece;
          }
        }
      }
    }

    if (newCars.length - 1 !== parseInt(lines[1].trim())) {
      result.message = `Numbers of cars not the same as input (found
${newCars.length}, expected ${lines[1].trim()} + 1(primary car))`;
      return result;
    }

    const exitGrid = {
      row: exitPiece.pos.row ? exitPiece.pos.row + 1 : 0,
      col: exitPiece.pos.col ? exitPiece.pos.col + 1 : 0,
    };

    const gameCars: Car[] = newCars.map((piece) => ({
      id: piece.id,
      isVertical: piece.orientation === "Vertical",
      size: piece.size,
      initialLeft: exitPiece.pos.col ? piece.pos.col : piece.pos.col - 1,
      initialTop: exitPiece.pos.row ? piece.pos.row : piece.pos.row - 1,
      isPrimary: piece.id === "P",
    }));

    result.success = true;
    result.newCars = gameCars;
    result.width = width;
    result.height = height;
```

```
    result.exitGrid = exitGrid;

    return result;
  } catch (error) {
    result.message = `Error parsing file: ${error instanceof Error ?
error.message : String(error)}`;
    return result;
  }
};
```

4.4. UI

Halaman utama

```
import { useState, useRef, useEffect } from "react";
import DraggableCar from "./components/DraggableCar";
import ControlPanel from "./components/ControlPanel";
import { rules } from "./lib/constant/rules";
import type { Car, EdgeGrid, Board, PieceMap, Move, Piece, Direction } from
"./lib/types";

function App() {
  const [boardWidth, setBoardWidth] = useState<number>(6);
  const [boardHeight, setBoardHeight] = useState<number>(6);
  const [gridSize, setgridSize] = useState<number>(80);
  const [cars, setCars] = useState<Car[]>([]);
  const [selectedEdgeGrid, setSelectedEdgeGrid] = useState<EdgeGrid |
null>(null);

  const [solutionMoves, setSolutionMoves] = useState<Move[]>([]);
  const [currentSolutionStep, setCurrentSolutionStep] = useState<number>(0);
  const [isAnimatingStep, setIsAnimatingStep] = useState<boolean>(false);
  const [isAutoPlaying, setIsAutoPlaying] = useState<boolean>(false);
  const [isReverse, setIsReverse] = useState<boolean>(false);
  const [originalBoardState, setOriginalBoardState] = useState<Car[]>([]);
```

```
const [isDisplayable, setIsDisplayble] = useState<boolean>(true);

const boardRef = useRef<HTMLDivElement>(null);
const totalBoardWidth = boardWidth + 2;
const totalBoardHeight = boardHeight + 2;

const boardWidthPx = boardWidth * gridSize;
const boardHeightPx = boardHeight * gridSize;
const totalBoardWidthPx = totalBoardWidth * gridSize;
const totalBoardHeightPx = totalBoardHeight * gridSize;

const autoPlayIntervalRef = useRef<number | null>(null);

const startAutoPlay = () => {
  setIsAutoPlaying(true);

  if (autoPlayIntervalRef.current !== null) {
    window.clearInterval(autoPlayIntervalRef.current);
  }

  autoPlayIntervalRef.current = window.setInterval(() => {
    setCurrentSolutionStep((currentSolutionStep) => {
      if (currentSolutionStep < solutionMoves.length - 1) {
        setIsAnimatingStep(true);
        return currentSolutionStep + 1;
      } else {
        stopAutoPlay();
        return currentSolutionStep;
      }
    });
  }, 350);
};

const stopAutoPlay = () => {
  if (autoPlayIntervalRef.current !== null) {
    window.clearInterval(autoPlayIntervalRef.current);
    autoPlayIntervalRef.current = null;
```

```
    }
    setIsAutoPlaying(false);
  };

  useEffect(() => {
    return () => {
      if (autoPlayIntervalRef.current !== null) {
        window.clearInterval(autoPlayIntervalRef.current);
      }
    };
  }, []);

  useEffect(() => {
    setIsAnimatingStep(true);
    const animationTimer = setTimeout(() => {
      setIsAnimatingStep(false);
    }, 1);
    return () => {
      clearTimeout(animationTimer);
    };
  }, [currentSolutionStep, isReverse]);

  useEffect(() => {
    const maxGridSize = 80;
    const minGridSize = 20;
    const largest = Math.max(boardWidth, boardHeight);

    const newGridSize = Math.round(maxGridSize - ((maxGridSize - minGridSize) *
(largest - 3)) / 9);

    setgridSize(Math.max(minGridSize, Math.min(maxGridSize, newGridSize)));
  }, [boardWidth, boardHeight]);

  const isEdgeGrid = (row: number, col: number) => {
    return row === 0 || row === totalBoardHeight - 1 || col === 0 || col ===
totalBoardWidth - 1;
  };
```

```
  const isCornerCell = (row: number, col: number) => {
    return (row === 0 && col === 0) || (row === 0 && col === totalBoardWidth -
1) || (row === totalBoardHeight - 1 && col === 0) || (row === totalBoardHeight
- 1 && col === totalBoardWidth - 1);
  };

  const ExitMarker = ({ position, gridSize }: { position: EdgeGrid | null;
gridSize: number }) => {
    if (!position) return null;

    const style = {
      position: "absolute" as const,
      width: gridSize,
      height: gridSize,
      backgroundColor: "yellow",
      border: "2px solid black",
      borderRadius: "8px",
      zIndex: 5,
      top: position.row * gridSize,
      left: position.col * gridSize,
    };

    return <div style={style} className="exit-marker" />;
  };

  const renderGrid = () => {
    const grid = [];
    for (let row = 0; row < totalBoardHeight; row++) {
      for (let col = 0; col < totalBoardWidth; col++) {
        const isCorner = isCornerCell(row, col);
        const isEdge = isEdgeGrid(row, col);

        const dataAttributes = !isEdge
          ? {
              "data-row": row - 1,
              "data-col": col - 1,
            }
          : {};
```

```
        grid.push(
          <div
            key={`${row}-${col}`}
            className="border border-gray-300"
            style={{
              width: gridSize,
              height: gridSize,
              backgroundColor: isCorner || isEdge ? "gray" : "white",
            }}
            {...dataAttributes}
          />
        );
      }
    }
    return grid;
  };

  const updateCarPosition = (id: string, top: number, left: number) => {
    setCars((prevCars) => prevCars.map((car) => (car.id === id ? { ...car,
initialTop: top, initialLeft: left } : car)));
  };

  const deleteCarById = (id: string) => {
    setCars((prevCars) => prevCars.filter((car) => car.id !== id));
  };

  const convertCarsToBoard = (): { board: Board; pieces: PieceMap; overlaps:
boolean } => {
    const grid: string[][] = Array(boardHeight)
      .fill(null)
      .map(() => Array(boardWidth).fill("."));

    const piecesMap: PieceMap = {};
    let hasOverlaps = false;

    if (selectedEdgeGrid) {
      const exitPiece: Piece = {
```

```
        id: "K",
        pos: {
          row: selectedEdgeGrid.row - 1,
          col: selectedEdgeGrid.col - 1,
        },
        orientation: "Unknown",
        size: 1,
      };
      piecesMap["K"] = exitPiece;

      if (exitPiece.pos.row >= 0 && exitPiece.pos.row < boardHeight &&
exitPiece.pos.col >= 0 && exitPiece.pos.col < boardWidth) {
        grid[exitPiece.pos.row][exitPiece.pos.col] = "K";
      }
    }

    for (const car of cars) {
      const piece: Piece = {
        id: car.id,
        pos: {
          row: car.initialTop,
          col: car.initialLeft,
        },
        orientation: car.isVertical ? "Vertical" : "Horizontal",
        size: car.size,
      };
      piecesMap[car.id] = piece;

      for (let i = 0; i < car.size; i++) {
        const row = car.isVertical ? car.initialTop + i : car.initialTop;
        const col = car.isVertical ? car.initialLeft : car.initialLeft + i;

        if (grid[row][col] !== ".") {
          hasOverlaps = true;
        }
        grid[row][col] = car.id;
      }
    }
```

```
    return {
      board: {
        width: boardWidth,
        height: boardHeight,
        grid,
      },
      pieces: piecesMap,
      overlaps: hasOverlaps,
    };
  };

  const getReverseDirection = (direction: Direction): Direction => {
    switch (direction) {
      case "Up":
        return "Down";
      case "Down":
        return "Up";
      case "Left":
        return "Right";
      case "Right":
        return "Left";
      default:
        return direction;
    }
  };

  return (
    <main className="relative flex flex-row  items-center p-4 w-full
min-h-screen bg-gray-100">
      <div className="w-full flex flex-col items-center justify-center gap-10">
        <h1 className="text-3xl font-bold mb-6 text-center">Unblock Car
Game</h1>
        <ControlPanel
          boardWidth={boardWidth}
          setBoardWidth={setBoardWidth}
          boardHeight={boardHeight}
          setBoardHeight={setBoardHeight}
```

```
          gridSize={gridSize}
          cars={cars}
          setCars={setCars}
          selectedEdgeGrid={selectedEdgeGrid}
          setSelectedEdgeGrid={setSelectedEdgeGrid}
          totalBoardWidth={totalBoardWidth}
          totalBoardHeight={totalBoardHeight}
          solutionMoves={solutionMoves}
          setSolutionMoves={setSolutionMoves}
          solutionStep={currentSolutionStep}
          setSolutionStep={setCurrentSolutionStep}
          convertCarsToBoard={convertCarsToBoard}
          isAnimatingStep={isAnimatingStep}
          setIsAnimatingStep={setIsAnimatingStep}
          isAutoPlaying={isAutoPlaying}
          startAutoPlay={startAutoPlay}
          stopAutoPlay={stopAutoPlay}
          originalBoardState={originalBoardState}
          setOriginalBoardState={setOriginalBoardState}
          setIsReverse={setIsReverse}
          setIsDisplayable={setIsDisplayble}
        />
        <div className="flex flex-col items-center justify-center">
          <h2 className="font-bold text-3xl">Rules</h2>
          <div>
            {rules.map((rule, index) => (
              <p key={index}>
                {index + 1}. {rule}
              </p>
            ))}
          </div>
        </div>
      </div>
      {isDisplayable && (
        <>
          <div className="mb-8 relative w-full flex items-center
justify-center">
            <div
```

```
              ref={boardRef}
              className="grid bg-white"
              style={{
                gridTemplateColumns: `repeat(${totalBoardWidth},
${gridSize}px)`,
                gridTemplateRows: `repeat(${totalBoardHeight}, ${gridSize}px)`,
                width: totalBoardWidthPx,
                height: totalBoardHeightPx,
                position: "relative",
              }}
            >
              {renderGrid()}
              <ExitMarker position={selectedEdgeGrid} gridSize={gridSize} />
            </div>
          </div>

          {cars.map((car) => (
            <DraggableCar
              key={car.id}
              id={car.id}
              width={car.isVertical ? gridSize : car.size * gridSize}
              height={car.isVertical ? car.size * gridSize : gridSize}
              minTop={0}
              maxTop={boardHeightPx - (car.isVertical ? car.size : 1) *
gridSize + 1.75 * gridSize}
              minLeft={0}
              maxLeft={boardWidthPx - (car.isVertical ? 1 : car.size) *
gridSize + 1.75 * gridSize}
              initialTop={(car.initialTop + 1) * gridSize}
              initialLeft={(car.initialLeft + 1) * gridSize}
              parentRef={boardRef}
              onPositionChange={updateCarPosition}
              inputGridSize={gridSize}
              deleteCarById={deleteCarById}
              isPrimary={car.isPrimary}
              isExecutingMove={
                isAnimatingStep &&
                solutionMoves.length > 0 &&
```

```
                (isReverse ? currentSolutionStep < solutionMoves.length - 1 &&
solutionMoves[currentSolutionStep + 1]?.piece.id === car.id :
solutionMoves[currentSolutionStep]?.piece.id === car.id)
              }
              moveDirection={
                isAnimatingStep && solutionMoves.length > 0
                  ? isReverse
                    ? currentSolutionStep < solutionMoves.length - 1 &&
solutionMoves[currentSolutionStep + 1]?.piece.id === car.id
                      ? getReverseDirection(solutionMoves[currentSolutionStep +
1].direction)
                      : undefined
                    : solutionMoves[currentSolutionStep]?.piece.id === car.id
                    ? solutionMoves[currentSolutionStep].direction
                    : undefined
                  : undefined
              }
              moveSteps={
                isAnimatingStep && solutionMoves.length > 0
                  ? isReverse
                    ? currentSolutionStep < solutionMoves.length - 1 &&
solutionMoves[currentSolutionStep + 1]?.piece.id === car.id
                      ? solutionMoves[currentSolutionStep + 1].steps
                      : undefined
                    : solutionMoves[currentSolutionStep]?.piece.id === car.id
                    ? solutionMoves[currentSolutionStep].steps
                    : undefined
                  : undefined
              }
            />
          ))}
        </>
      )}
    </main>
  );
}

export default App;
```

Control Panel

```
import { useState, useRef } from "react";
import type { Car, EdgeGrid, PieceMap, Board, Move } from "../lib/types";
import { parseFileContents } from "../lib/helpers/validateInput";
import { aStar } from "../lib/algo/aStar";
import { gbfs } from "../lib/algo/gbfs";
import { ucs } from "../lib/algo/ucs";
import { distanceToExit } from "../lib/heuristics/distanceToExit";
import { blockingCars } from "../lib/heuristics/blockingCars";
import { recursiveBlockers } from "../lib/heuristics/recursiveBlockers";
import { combinedHeuristic } from "../lib/heuristics/combinedHeuristic";
import { moveNeededEstimate } from "../lib/heuristics/moveNeededEstimate";
import { downloadSolutionFile, generateBoardStates } from
"../lib/helpers/output";
import { beamSearch } from "../lib/algo/beamSearch";

interface ControlPanelProps {
  boardWidth: number;
  setBoardWidth: (width: number) => void;
  boardHeight: number;
  setBoardHeight: (height: number) => void;
  gridSize: number;
  cars: Car[];
  setCars: (cars: Car[]) => void;
  selectedEdgeGrid: EdgeGrid | null;
  setSelectedEdgeGrid: (grid: EdgeGrid | null) => void;
  totalBoardWidth: number;
  totalBoardHeight: number;
  solutionMoves: Move[];
  setSolutionMoves: (moves: Move[]) => void;
  solutionStep: number;
  setSolutionStep: (step: number) => void;
  convertCarsToBoard: () => { board: Board; pieces: PieceMap; overlaps: boolean
```

```
};
  isAnimatingStep?: boolean;
  setIsAnimatingStep?: (isAnimating: boolean) => void;
  isAutoPlaying?: boolean;
  startAutoPlay?: () => void;
  stopAutoPlay?: () => void;
  originalBoardState: Car[];
  setOriginalBoardState: (cars: Car[]) => void;
  setIsReverse: (bool: boolean) => void;
  setIsDisplayable: (bool: boolean) => void;
}

const ControlPanel = ({
  boardWidth,
  setBoardWidth,
  boardHeight,
  setBoardHeight,
  cars,
  setCars,
  selectedEdgeGrid,
  setSelectedEdgeGrid,
  totalBoardWidth,
  totalBoardHeight,
  solutionMoves,
  setSolutionMoves,
  solutionStep,
  setSolutionStep,
  convertCarsToBoard,
  isAnimatingStep,
  isAutoPlaying,
  startAutoPlay,
  stopAutoPlay,
  originalBoardState,
  setOriginalBoardState,
  setIsReverse,
  setIsDisplayable,
}: ControlPanelProps) => {
  const [selectedAlgorithm, setSelectedAlgorithm] = useState<string>("aStar");
```

```
  const [selectedHeuristic, setSelectedHeuristic] =
useState<string>("combined");
  const [showSolution, setShowSolution] = useState<boolean>(false);
  const [inputCarLength, setInputCarLength] = useState<number>(2);
  const [inputCarOrientation, setInputCarOrientation] =
useState<boolean>(false);
  const [isPrimary, setIsPrimary] = useState<boolean>(false);
  const [exitRow, setExitRow] = useState<number>(0);
  const [exitCol, setExitCol] = useState<number>(0);
  const [isSolving, setIsSolving] = useState<boolean>(false);
  const [nodesFound, setNodesFound] = useState<number>(0);
  const [timeTaken, setTimeTaken] = useState<number>(0);
  const [beamNumber, setBeamNumber] = useState<number>(1);

  const primaryCar = cars.find((car) => car.isPrimary);
  const isInitialState = solutionStep === -1;

  const fileInputRef = useRef<HTMLInputElement>(null);

  const handleFileUpload = (event: React.ChangeEvent<HTMLInputElement>) => {
    setIsDisplayable(true);
    const file = event.target.files?.[0];
    if (!file) return;

    if (file.type !== "text/plain" && !file.name.endsWith(".txt")) {
      alert("Please upload a valid .txt file");
      if (fileInputRef.current) fileInputRef.current.value = "";
      return;
    }

    setCars([]);
    setSelectedEdgeGrid(null);
    setShowSolution(false);

    const reader = new FileReader();
    reader.onload = (e) => {
      const text = e.target?.result as string;
      const result = parseFileContents(text);
```

```
      if (!result.success) {
        alert(result.message);
        return;
      }

      if (result.width) setBoardWidth(result.width);
      if (result.height) setBoardHeight(result.height);
      if (result.exitGrid) setSelectedEdgeGrid(result.exitGrid);
      if (result.newCars) setCars(result.newCars);

      if (fileInputRef.current) fileInputRef.current.value = "";
    };
    setShowSolution(false);
    reader.readAsText(file);
  };

  const getRandomCharacter = () => {
    const excludedChars = ["K", "P", ".", " "];
    const usedChars = cars.map((car) => car.id);

    for (let charCode = 65; charCode <= 90; charCode++) {
      const letter = String.fromCharCode(charCode);
      if (!excludedChars.includes(letter) && !usedChars.includes(letter)) {
        return letter;
      }
    }
  };

  const addCar = (size: number, isVertical: boolean, isPrimary: boolean) => {
    if (isPrimary && primaryCar) {
      alert("Only one primary car is allowed. Delete the existing primary car
first.");
      return;
    }

    const normalCarsCount = cars.filter((car) => !car.isPrimary).length;
```

```
  if (!isPrimary && normalCarsCount >= 24) {
    alert("Maximum of 24 normal cars reached");
    return;
  }

  const newCar: Car = {
    id: isPrimary ? "P" : getRandomCharacter() || "*",
    isVertical,
    size,
    initialLeft: 0,
    initialTop: 0,
    isPrimary,
  };

  setCars([...cars, newCar]);
};

const solveBoard = () => {
  if (!selectedEdgeGrid) {
    alert("Please select an exit position first");
    return;
  }

  if (!cars.some((car) => car.isPrimary)) {
    alert("Please add a primary car first");
    return;
  }

  const { overlaps, board, pieces } = convertCarsToBoard();

  if (overlaps) {
    alert("There are overlapping cars, fix the positioning!");
    return;
  }

  const primaryCar = cars.find((car) => car.isPrimary);

  if (primaryCar) {
```

```
      const exitPosition = {
        row: selectedEdgeGrid.row - 1,
        col: selectedEdgeGrid.col - 1,
      };

      if (!primaryCar.isVertical) {
        if (primaryCar.initialTop !== exitPosition.row) {
          alert("Error: Primary car must be in the same row as the exit when
oriented horizontally.");
          return;
        }

        const blockingCars = cars.filter(
          (car) =>
            !car.isPrimary &&
            !car.isVertical &&
            car.initialTop === exitPosition.row &&
            ((exitPosition.col > primaryCar.initialLeft && car.initialLeft >
primaryCar.initialLeft + primaryCar.size - 1 && car.initialLeft <=
exitPosition.col) ||
              (exitPosition.col < primaryCar.initialLeft && car.initialLeft +
car.size - 1 >= exitPosition.col && car.initialLeft < primaryCar.initialLeft))
        );

        if (blockingCars.length > 0) {
          alert(`Warning: There are ${blockingCars.length} horizontal cars
blocking the direct path to the exit. The puzzle is unsolvable.`);
          return;
        }
      }

      if (primaryCar.isVertical) {
        if (primaryCar.initialLeft !== exitPosition.col) {
          alert("Error: Primary car must be in the same column as the exit when
oriented vertically.");
          return;
        }
```

```
        const blockingCars = cars.filter(
          (car) =>
            !car.isPrimary &&
            car.isVertical &&
            car.initialLeft === exitPosition.col &&
            ((exitPosition.row > primaryCar.initialTop && car.initialTop > primaryCar.initialTop + primaryCar.size - 1 && car.initialTop <= exitPosition.row) ||
              (exitPosition.row < primaryCar.initialTop && car.initialTop + car.size - 1 >= exitPosition.row && car.initialTop < primaryCar.initialTop))
        );

        if (blockingCars.length > 0) {
          alert(`Warning: There are ${blockingCars.length} vertical cars blocking the direct path to the exit. The puzzle is unsolvable.`);
          return;
        }
      }
    }

    setOriginalBoardState([...cars]);

    setIsSolving(true);
    setSolutionMoves([]);
    setSolutionStep(-1);

    let heuristicFunction;
    switch (selectedHeuristic) {
      case "distance":
        heuristicFunction = distanceToExit;
        break;
      case "blocking":
        heuristicFunction = blockingCars;
        break;
      case "recursive":
        heuristicFunction = recursiveBlockers;
        break;
      case "moveNeeded":
```

```
        heuristicFunction = moveNeededEstimate;
        break;
      default:
        heuristicFunction = combinedHeuristic;
    }

    let result;
    try {
      switch (selectedAlgorithm) {
        case "gbfs":
          result = gbfs(board, pieces, heuristicFunction);
          break;
        case "beam":
          result = beamSearch(board, pieces, heuristicFunction, beamNumber);
          break;
        case "ucs":
          result = ucs(board, pieces);
          break;
        default:
          result = aStar(board, pieces, heuristicFunction);
      }

      if (result && result.found) {
        setNodesFound(result.nodesVisited);
        setTimeTaken(result.timeTaken);
        setSolutionMoves(result.moveHistory);
        setShowSolution(true);
        alert(`Solution found in ${result.moveHistory.length} moves!`);
      } else {
        alert("No solution found!");
      }
    } catch (error) {
      console.error("Error solving board:", error);
      alert(`Error: ${error instanceof Error ? error.message :
String(error)}`);
    } finally {
      setIsSolving(false);
    }
```

```
  };

  return (
    <div className="w-full max-w-4xl bg-white p-4 rounded-lg shadow-md">
      <div className="flex flex-wrap items-center justify-between gap-2 mb-3
border-b pb-3">
        <div className="flex items-center gap-2">
          <label className="cursor-pointer bg-blue-500 hover:bg-blue-600
text-white px-3 py-1 rounded text-sm">
            Upload
            <input ref={fileInputRef} type="file" accept=".txt,text/plain"
onChange={handleFileUpload} className="hidden" />
          </label>

          <div className="flex items-center">
            <label className="mr-1 text-sm">W:</label>
            <input type="number" min="2" value={boardWidth} onChange={(e) =>
setBoardWidth(parseInt(e.target.value))} className="w-12 p-1 border
border-gray-300 rounded text-sm" />
          </div>

          <div className="flex items-center">
            <label className="mr-1 text-sm">H:</label>
            <input type="number" min="2" value={boardHeight} onChange={(e) =>
setBoardHeight(parseInt(e.target.value))} className="w-12 p-1 border
border-gray-300 rounded text-sm" />
          </div>
        </div>

        <div className="flex items-center gap-2">
          <div className="flex items-center">
            <label className="mr-1 text-sm">Exit:</label>
            <input
              type="number"
              min={0}
              max={totalBoardHeight - 1}
              value={exitRow}
              onChange={(e) => {
```

```
                const val = Math.max(0, Math.min(totalBoardHeight - 1,
parseInt(e.target.value) || 0));
                setExitRow(val);
                let newCol = exitCol;
                if (val !== 0 && val !== totalBoardHeight - 1) {
                  if (exitCol !== 0 && exitCol !== totalBoardWidth - 1) {
                    newCol = 0;
                  }
                }
                if ((val === 0 || val === totalBoardHeight - 1) && (newCol ===
0 || newCol === totalBoardWidth - 1)) {
                  newCol = Math.min(Math.max(1, newCol), totalBoardWidth - 2);
                }
                setExitCol(newCol);
                setSelectedEdgeGrid({ row: val, col: newCol });
              }}
              className="w-12 p-1 border border-gray-300 rounded text-sm"
            />
            <span className="mx-1 text-sm">×</span>
            <input
              type="number"
              min={0}
              max={totalBoardWidth - 1}
              value={exitCol}
              onChange={(e) => {
                const val = Math.max(0, Math.min(totalBoardWidth - 1,
parseInt(e.target.value) || 0));
                setExitCol(val);
                let newRow = exitRow;
                if (val !== 0 && val !== totalBoardWidth - 1) {
                  if (exitRow !== 0 && exitRow !== totalBoardHeight - 1) {
                    newRow = 0;
                  }
                }
                if ((val === 0 || val === totalBoardWidth - 1) && (newRow === 0
|| newRow === totalBoardHeight - 1)) {
                  newRow = Math.min(Math.max(1, newRow), totalBoardHeight - 2);
                }
```

```
                setExitRow(newRow);
                setSelectedEdgeGrid({ row: newRow, col: val });
              }}
              className="w-12 p-1 border border-gray-300 rounded text-sm"
            />
          </div>

          <button
            onClick={() => {
              setCars([]);
              setSelectedEdgeGrid(null);
              setShowSolution(false);
            }}
            className="px-2 py-1 bg-red-500 text-white rounded text-sm
hover:bg-red-600"
          >
            Clear
          </button>
        </div>
      </div>
      <div className="flex flex-wrap items-center justify-between gap-2 mb-3
border-b pb-3">
        <div className="flex items-center gap-2">
          <label className="text-sm font-medium">Car:</label>
          <div className="flex items-center">
            <label className="mr-1 text-sm">Len</label>
            <input
              type="number"
              min="2"
              value={inputCarLength}
              onChange={(e) => setInputCarLength(Math.max(2, Math.min(10,
parseInt(e.target.value) || 2)))}
              className="w-12 p-1 border border-gray-300 rounded text-sm"
            />
          </div>

          <button className="px-2 py-1 border border-gray-300 rounded text-sm"
onClick={() => setInputCarOrientation(!inputCarOrientation)}>
```

```
            {inputCarOrientation ? "Vert" : "Horiz"}
          </button>

          <div className="flex items-center">
            <label className="mr-1 text-sm">Primary</label>
            <input
              type="checkbox"
              checked={isPrimary}
              onChange={() => {
                if (!primaryCar || isPrimary) {
                  setIsPrimary(!isPrimary);
                }
              }}
              disabled={primaryCar && !isPrimary}
              className="w-4 h-4"
            />
          </div>
        </div>

        <button onClick={() => addCar(inputCarLength, inputCarOrientation,
isPrimary)} className="px-3 py-1 bg-yellow-500 text-white rounded text-sm
hover:bg-yellow-600">
          Add Car
        </button>
      </div>
      <div className="flex flex-wrap items-center justify-between gap-2">
        <div className="flex items-center gap-2">
          <label className="text-sm font-medium">Solver:</label>
          <select value={selectedAlgorithm} onChange={(e) =>
setSelectedAlgorithm(e.target.value)} className="p-1 border border-gray-300
rounded text-sm" disabled={isSolving}>
            <option value="aStar">A*</option>
            <option value="gbfs">Greedy</option>
            <option value="ucs">UCS</option>
            <option value="beam">Beam</option>
          </select>
          {selectedAlgorithm !== "ucs" && (
            <select
```

```
            value={selectedHeuristic}
            onChange={(e) => setSelectedHeuristic(e.target.value)}
            className="p-1 border border-gray-300 rounded text-sm"
            disabled={isSolving || selectedAlgorithm === "ucs"}
          >
            <option value="combined">Combined</option>
            <option value="distance">Distance</option>
            <option value="blocking">Blocking</option>
            <option value="recursive">Recursive</option>
            <option value="moveNeeded">Move Est.</option>
          </select>
        )}
        {selectedAlgorithm === "beam" && (
          <input
            type="number"
            className="p-1 border border-gray-300 rounded text-sm"
            min={1}
            value={beamNumber}
            onChange={(e) => {
              if (!e.target.value) {
                setBeamNumber(1);
              } else {
                setBeamNumber(parseInt(e.target.value));
              }
            }}
          />
        )}
      </div>

      <button onClick={solveBoard} disabled={isSolving} className={`px-3 py-1
${isSolving ? "bg-gray-400" : "bg-green-500 hover:bg-green-600"} text-white
rounded text-sm`}>
        {isSolving ? "Solving..." : "Solve"}
      </button>
    </div>
    {showSolution && solutionMoves.length > 0 && (
      <div className="mt-3 pt-3 border-t">
        <div className="flex items-center justify-between">
```

```
            <span className="text-sm font-medium">Solution:
{solutionMoves.length} moves</span>
            <span className="text-sm font-medium">Nodes: {nodesFound}</span>
            <span className="text-sm font-medium">Time:
{timeTaken.toFixed(3)}ms</span>

            <div className="flex items-center gap-1">
              <button
                onClick={() => {
                  if (isAutoPlaying && stopAutoPlay) {
                    stopAutoPlay();
                  }
                  setIsReverse(true);
                  setSolutionStep(Math.max(-1, solutionStep - 1));
                  if (solutionStep === -1 && originalBoardState.length > 0) {
                    setCars([...originalBoardState]);
                  }
                }}
                className="px-2 py-1 bg-blue-500 text-white rounded text-xs
disabled:bg-gray-300"
              >
                ←
              </button>
              <span className="text-xs">
                Step {solutionStep + 1}/{solutionMoves.length}
              </span>
              <button
                onClick={() => {
                  if (isAutoPlaying && stopAutoPlay) {
                    stopAutoPlay();
                  }
                  setIsReverse(false);
                  setSolutionStep(Math.min(solutionMoves.length, solutionStep +
1));

                }}
                disabled={solutionStep === solutionMoves.length - 1}
                className="px-2 py-1 bg-blue-500 text-white rounded text-xs
disabled:bg-gray-300"
```

```
              >
                →
              </button>
            </div>
          </div>

          <div className="mt-3 flex justify-center gap-2">
            {!isAutoPlaying ? (
              <button
                onClick={() => {
                  if (startAutoPlay) {
                    startAutoPlay();
                  }
                }}
                disabled={solutionStep === solutionMoves.length ||
isAnimatingStep}
                className="px-3 py-1 bg-green-500 text-white rounded text-xs
disabled:bg-gray-300 hover:bg-green-600"
              >
                Auto Play
              </button>
            ) : (
              <button
                onClick={() => {
                  if (stopAutoPlay) {
                    stopAutoPlay();
                  }
                }}
                className="px-3 py-1 bg-red-500 text-white rounded text-xs
hover:bg-red-600"
              >
                Stop
              </button>
            )}
            <button
              onClick={() => {
                if (isAutoPlaying && stopAutoPlay) {
                  stopAutoPlay();
```

```
                }
                setTimeout(() => {
                  if (originalBoardState.length > 0) {
                    setCars([...originalBoardState]);
                  }
                  setSolutionStep(-1);
                }, 500);
              }}
              disabled={isAnimatingStep || isInitialState}
              className="px-3 py-1 bg-gray-500 text-white rounded text-xs
disabled:bg-gray-300 hover:bg-gray-600"
            >
              Reset Board
            </button>
          </div>
          <button
            onClick={() => {
              const { board, pieces } = convertCarsToBoard();
              downloadSolutionFile(board, solutionMoves,
generateBoardStates(board, solutionMoves, pieces), `unblock_car_solution.txt`);
            }}
            disabled={solutionMoves.length === 0}
            className="px-3 py-1 bg-purple-500 text-white rounded text-xs
disabled:bg-gray-300 hover:bg-purple-600"
          >
            Download Solution
          </button>
        </div>
      )}
    </div>
  );
};

export default ControlPanel;
```

Draggable Car

```tsx
import { useState, useCallback, useEffect, useRef } from "react";
import type { Direction } from "../lib/types";

interface DraggableCarProps {
  id: string;
  width: number;
  height: number;
  minTop: number;
  maxTop: number;
  minLeft: number;
  maxLeft: number;
  initialTop: number;
  initialLeft: number;
  onPositionChange?: (id: string, top: number, left: number) => void;
  parentRef: React.RefObject<HTMLDivElement | null>;
  inputGridSize: number;
  deleteCarById: (id: string) => void;
  isPrimary: boolean;
  moveDirection?: Direction | null;
  moveSteps?: number;
  isExecutingMove?: boolean;
}

const DraggableCar = ({
  id,
  width,
  height,
  minTop,
  maxTop,
  minLeft,
  maxLeft,
  initialTop,
  initialLeft,
  onPositionChange,
  parentRef,
```

```
  inputGridSize,
  deleteCarById,
  isPrimary,
  moveDirection,
  moveSteps,
  isExecutingMove,
}: DraggableCarProps) => {
  const [position, setPosition] = useState({ top: 0, left: 0 });
  const [dragging, setDragging] = useState(false);
  const [offset, setOffset] = useState({ x: 0, y: 0 });
  const [parentBounds, setParentBounds] = useState({ top: 0, left: 0 });
  const [zIndex, setZIndex] = useState(1);
  const [isAnimating, setIsAnimating] = useState(false);
  const animationTimerRef = useRef<number | null>(null);

  useEffect(() => {
    if (parentRef.current) {
      const parentBounds = parentRef.current.getBoundingClientRect();
      setPosition({
        top: parentBounds.top + minTop + initialTop,
        left: parentBounds.left + minLeft + initialLeft,
      });
      setParentBounds({
        top: parentBounds.top,
        left: parentBounds.left,
      });
    }
  }, [parentRef, minTop, minLeft, initialTop, initialLeft]);

  const isWithinParentBounds = useCallback(
    (top: number, left: number) => {
      if (!parentRef.current) return false;
      return top >= parentBounds.top + minTop && top <= parentBounds.top +
maxTop && left >= parentBounds.left + minLeft && left <= parentBounds.left +
maxLeft;
    },
    [parentBounds, minTop, maxTop, minLeft, maxLeft, parentRef]
  );
```

```
  const snapToGrid = useCallback(
    (position: number, min: number, max: number, isHorizontal: boolean) => {
      const parentPosition = isHorizontal ? parentBounds.left + inputGridSize :
parentBounds.top + inputGridSize;
      const relativePosition = position - (parentPosition + min);
      const snappedRelativePosition = Math.round(relativePosition /
inputGridSize) * inputGridSize;
      const snappedPosition = parentPosition + min + snappedRelativePosition;

      return Math.max(parentPosition + min, Math.min(parentPosition + max,
snappedPosition));
    },
    [parentBounds.top, parentBounds.left, inputGridSize]
  );

  const handleMouseDown = useCallback(
    (e: React.MouseEvent<HTMLDivElement>) => {
      if (isAnimating) return;
      setDragging(true);
      setOffset({
        x: e.clientX - position.left,
        y: e.clientY - position.top,
      });
      setZIndex(10);
    },
    [position, isAnimating]
  );

  const handleMouseMove = useCallback(
    (e: MouseEvent) => {
      if (!dragging) return;

      let newTop = e.clientY - offset.y;
      let newLeft = e.clientX - offset.x;

      if (window.innerWidth > 768) {
        newTop = Math.max(0, Math.min(newTop, window.innerHeight - height));
```

```
      }
      newLeft = Math.max(0, Math.min(newLeft, window.innerWidth - width));

      setPosition({
        top: newTop,
        left: newLeft,
      });
    },
    [dragging, offset, height, width]
  );

  const handleMouseUp = useCallback(() => {
    setDragging(false);
    setZIndex(1);

    setPosition((prev) => {
      if (isWithinParentBounds(prev.top, prev.left)) {
        const newTop = snapToGrid(prev.top, minTop, maxTop, false);
        const newLeft = snapToGrid(prev.left, minLeft, maxLeft, true);
        setTimeout(() => {
          const relativeTop = Math.round((newTop - parentBounds.top - minTop) /
inputGridSize) - 1;
          const relativeLeft = Math.round((newLeft - parentBounds.left -
minLeft) / inputGridSize) - 1;

          if (onPositionChange) {
            onPositionChange(id, relativeTop, relativeLeft);
          }
        }, 100);
        return {
          top: newTop,
          left: newLeft,
        };
      }
      deleteCarById(id);
      return prev;
    });
  }, [minTop, maxTop, minLeft, maxLeft, snapToGrid, isWithinParentBounds, id,
```

```
onPositionChange, parentBounds, deleteCarById, inputGridSize]);

  useEffect(() => {
    if (dragging) {
      window.addEventListener("mousemove", handleMouseMove);
      window.addEventListener("mouseup", handleMouseUp);
    } else {
      window.removeEventListener("mousemove", handleMouseMove);
      window.removeEventListener("mouseup", handleMouseUp);
    }

    return () => {
      window.removeEventListener("mousemove", handleMouseMove);
      window.removeEventListener("mouseup", handleMouseUp);
    };
  }, [dragging, handleMouseMove, handleMouseUp]);

  useEffect(() => {
    return () => {
      if (animationTimerRef.current !== null) {
        clearTimeout(animationTimerRef.current);
      }
    };
  }, []);

  useEffect(() => {
    if (isExecutingMove && moveDirection && moveSteps && moveSteps > 0 &&
!isAnimating) {
      if (animationTimerRef.current !== null) {
        clearTimeout(animationTimerRef.current);
      }

      setIsAnimating(true);
      setZIndex(5);

      let newTop = position.top;
      let newLeft = position.left;
```

```
      switch (moveDirection) {
        case "Up":
          newTop -= moveSteps * inputGridSize;
          break;
        case "Down":
          newTop += moveSteps * inputGridSize;
          break;
        case "Left":
          newLeft -= moveSteps * inputGridSize;
          break;
        case "Right":
          newLeft += moveSteps * inputGridSize;
          break;
      }

      newTop = Math.max(parentBounds.top + minTop, Math.min(parentBounds.top +
maxTop, newTop));
      newLeft = Math.max(parentBounds.left + minLeft,
Math.min(parentBounds.left + maxLeft, newLeft));

      setPosition({
        top: newTop,
        left: newLeft,
      });

      animationTimerRef.current = window.setTimeout(() => {
        const snappedTop = snapToGrid(newTop, minTop, maxTop, false);
        const snappedLeft = snapToGrid(newLeft, minLeft, maxLeft, true);

        setPosition({
          top: snappedTop,
          left: snappedLeft,
        });

        const relativeTop = Math.round((snappedTop - parentBounds.top - minTop)
/ inputGridSize) - 1;
        const relativeLeft = Math.round((snappedLeft - parentBounds.left -
minLeft) / inputGridSize) - 1;
```

```
        if (onPositionChange) {
          onPositionChange(id, relativeTop, relativeLeft);
        }

        setIsAnimating(false);
        setZIndex(1);
        animationTimerRef.current = null;
      }, 300);
    }
  }, [isExecutingMove, moveDirection, moveSteps, isAnimating, inputGridSize,
position, parentBounds, minTop, maxTop, minLeft, maxLeft, onPositionChange, id,
snapToGrid]);

  return (
    <div
      className="absolute cursor-move rounded-lg"
      style={{
        top: position.top,
        left: position.left,
        zIndex,
        width,
        height,
        transition: isAnimating ? "top 0.3s ease-out, left 0.3s ease-out" :
"none",
      }}
      onMouseDown={handleMouseDown}
      data-position={`${position.top},${position.left}`}
      data-parent-bounds={`${parentBounds.top}`}
    >
      <div
        className={`text-white flex justify-center items-center text-center
w-full h-full rounded-lg border-2 ${isPrimary ? "border-red-950 bg-red-500" :
"border-blue-950 bg-blue-500"}`}
        style={{
          boxShadow: isAnimating
            ? isPrimary
              ? "0 0 40px 10px gold"
```

```
              : "0 0 40px 10px cyan"
            : dragging
            ? isPrimary
              ? "0 0 40px 10px blue"
              : "0 0 40px 10px red"
            : isPrimary
            ? "0 0 20px 0 red"
            : "0 0 20px 0 blue",
          transition: "box-shadow 0.2s",
        }}
      >
        {id}
      </div>
    </div>
  );
};

export default DraggableCar;
```

Halaman utama

```
import { useState, useRef, useEffect } from "react";
import DraggableCar from "./components/DraggableCar";
import ControlPanel from "./components/ControlPanel";
import { rules } from "./lib/constant/rules";
import type { Car, EdgeGrid, Board, PieceMap, Move, Piece, Direction } from
"./lib/types";

function App() {
  const [boardWidth, setBoardWidth] = useState<number>(6);
  const [boardHeight, setBoardHeight] = useState<number>(6);
  const [gridSize, setgridSize] = useState<number>(80);
  const [cars, setCars] = useState<Car[]>([]);
  const [selectedEdgeGrid, setSelectedEdgeGrid] = useState<EdgeGrid |
null>(null);
```

```tsx
const [solutionMoves, setSolutionMoves] = useState<Move[]>([]);
const [currentSolutionStep, setCurrentSolutionStep] = useState<number>(0);
const [isAnimatingStep, setIsAnimatingStep] = useState<boolean>(false);
const [isAutoPlaying, setIsAutoPlaying] = useState<boolean>(false);
const [isReverse, setIsReverse] = useState<boolean>(false);
const [originalBoardState, setOriginalBoardState] = useState<Car[]>([]);

const [isDisplayable, setIsDisplayble] = useState<boolean>(true);

const boardRef = useRef<HTMLDivElement>(null);
const totalBoardWidth = boardWidth + 2;
const totalBoardHeight = boardHeight + 2;

const boardWidthPx = boardWidth * gridSize;
const boardHeightPx = boardHeight * gridSize;
const totalBoardWidthPx = totalBoardWidth * gridSize;
const totalBoardHeightPx = totalBoardHeight * gridSize;

const autoPlayIntervalRef = useRef<number | null>(null);

const startAutoPlay = () => {
  setIsAutoPlaying(true);

  if (autoPlayIntervalRef.current !== null) {
    window.clearInterval(autoPlayIntervalRef.current);
  }

  autoPlayIntervalRef.current = window.setInterval(() => {
    setCurrentSolutionStep((currentSolutionStep) => {
      if (currentSolutionStep < solutionMoves.length - 1) {
        setIsAnimatingStep(true);
        return currentSolutionStep + 1;
      } else {
        stopAutoPlay();
        return currentSolutionStep;
      }
    });
  }, 350);
```

```
  };

  const stopAutoPlay = () => {
    if (autoPlayIntervalRef.current !== null) {
      window.clearInterval(autoPlayIntervalRef.current);
      autoPlayIntervalRef.current = null;
    }
    setIsAutoPlaying(false);
  };

  useEffect(() => {
    return () => {
      if (autoPlayIntervalRef.current !== null) {
        window.clearInterval(autoPlayIntervalRef.current);
      }
    };
  }, []);

  useEffect(() => {
    setIsAnimatingStep(true);
    const animationTimer = setTimeout(() => {
      setIsAnimatingStep(false);
    }, 1);
    return () => {
      clearTimeout(animationTimer);
    };
  }, [currentSolutionStep, isReverse]);

  useEffect(() => {
    const maxGridSize = 80;
    const minGridSize = 20;
    const largest = Math.max(boardWidth, boardHeight);

    const newGridSize = Math.round(maxGridSize - ((maxGridSize - minGridSize) *
(largest - 3)) / 9);

    setgridSize(Math.max(minGridSize, Math.min(maxGridSize, newGridSize)));
  }, [boardWidth, boardHeight]);
```

```
  const isEdgeGrid = (row: number, col: number) => {
    return row === 0 || row === totalBoardHeight - 1 || col === 0 || col ===
totalBoardWidth - 1;
  };

  const isCornerCell = (row: number, col: number) => {
    return (row === 0 && col === 0) || (row === 0 && col === totalBoardWidth -
1) || (row === totalBoardHeight - 1 && col === 0) || (row === totalBoardHeight
- 1 && col === totalBoardWidth - 1);
  };

  const ExitMarker = ({ position, gridSize }: { position: EdgeGrid | null;
gridSize: number }) => {
    if (!position) return null;

    const style = {
      position: "absolute" as const,
      width: gridSize,
      height: gridSize,
      backgroundColor: "yellow",
      border: "2px solid black",
      borderRadius: "8px",
      zIndex: 5,
      top: position.row * gridSize,
      left: position.col * gridSize,
    };

    return <div style={style} className="exit-marker" />;
  };

  const renderGrid = () => {
    const grid = [];
    for (let row = 0; row < totalBoardHeight; row++) {
      for (let col = 0; col < totalBoardWidth; col++) {
        const isCorner = isCornerCell(row, col);
        const isEdge = isEdgeGrid(row, col);
```

```
        const dataAttributes = !isEdge
          ? {
              "data-row": row - 1,
              "data-col": col - 1,
            }
          : {};

        grid.push(
          <div
            key={`${row}-${col}`}
            className="border border-gray-300"
            style={{
              width: gridSize,
              height: gridSize,
              backgroundColor: isCorner || isEdge ? "gray" : "white",
            }}
            {...dataAttributes}
          />
        );
      }
    }
    return grid;
  };

  const updateCarPosition = (id: string, top: number, left: number) => {
    setCars((prevCars) => prevCars.map((car) => (car.id === id ? { ...car,
initialTop: top, initialLeft: left } : car)));
  };

  const deleteCarById = (id: string) => {
    setCars((prevCars) => prevCars.filter((car) => car.id !== id));
  };

  const convertCarsToBoard = (): { board: Board; pieces: PieceMap; overlaps:
boolean } => {
    const grid: string[][] = Array(boardHeight)
      .fill(null)
      .map(() => Array(boardWidth).fill("."));
```

```
    const piecesMap: PieceMap = {};
    let hasOverlaps = false;

    if (selectedEdgeGrid) {
      const exitPiece: Piece = {
        id: "K",
        pos: {
          row: selectedEdgeGrid.row - 1,
          col: selectedEdgeGrid.col - 1,
        },
        orientation: "Unknown",
        size: 1,
      };
      piecesMap["K"] = exitPiece;

      if (exitPiece.pos.row >= 0 && exitPiece.pos.row < boardHeight &&
exitPiece.pos.col >= 0 && exitPiece.pos.col < boardWidth) {
        grid[exitPiece.pos.row][exitPiece.pos.col] = "K";
      }
    }

    for (const car of cars) {
      const piece: Piece = {
        id: car.id,
        pos: {
          row: car.initialTop,
          col: car.initialLeft,
        },
        orientation: car.isVertical ? "Vertical" : "Horizontal",
        size: car.size,
      };
      piecesMap[car.id] = piece;

      for (let i = 0; i < car.size; i++) {
        const row = car.isVertical ? car.initialTop + i : car.initialTop;
        const col = car.isVertical ? car.initialLeft : car.initialLeft + i;
```

```
        if (grid[row][col] !== ".") {
          hasOverlaps = true;
        }
        grid[row][col] = car.id;
      }
    }

    return {
      board: {
        width: boardWidth,
        height: boardHeight,
        grid,
      },
      pieces: piecesMap,
      overlaps: hasOverlaps,
    };
  };

  const getReverseDirection = (direction: Direction): Direction => {
    switch (direction) {
      case "Up":
        return "Down";
      case "Down":
        return "Up";
      case "Left":
        return "Right";
      case "Right":
        return "Left";
      default:
        return direction;
    }
  };

  return (
    <main className="relative flex flex-row  items-center p-4 w-full
min-h-screen bg-gray-100">
      <div className="w-full flex flex-col items-center justify-center gap-10">
        <h1 className="text-3xl font-bold mb-6 text-center">Unblock Car
```

```
Game</h1>
        <ControlPanel
          boardWidth={boardWidth}
          setBoardWidth={setBoardWidth}
          boardHeight={boardHeight}
          setBoardHeight={setBoardHeight}
          gridSize={gridSize}
          cars={cars}
          setCars={setCars}
          selectedEdgeGrid={selectedEdgeGrid}
          setSelectedEdgeGrid={setSelectedEdgeGrid}
          totalBoardWidth={totalBoardWidth}
          totalBoardHeight={totalBoardHeight}
          solutionMoves={solutionMoves}
          setSolutionMoves={setSolutionMoves}
          solutionStep={currentSolutionStep}
          setSolutionStep={setCurrentSolutionStep}
          convertCarsToBoard={convertCarsToBoard}
          isAnimatingStep={isAnimatingStep}
          setIsAnimatingStep={setIsAnimatingStep}
          isAutoPlaying={isAutoPlaying}
          startAutoPlay={startAutoPlay}
          stopAutoPlay={stopAutoPlay}
          originalBoardState={originalBoardState}
          setOriginalBoardState={setOriginalBoardState}
          setIsReverse={setIsReverse}
          setIsDisplayable={setIsDisplayble}
        />
        <div className="flex flex-col items-center justify-center">
          <h2 className="font-bold text-3xl">Rules</h2>
          <div>
            {rules.map((rule, index) => (
              <p key={index}>
                {index + 1}. {rule}
              </p>
            ))}
          </div>
        </div>
```

```
      </div>
      {isDisplayable && (
        <>
          <div className="mb-8 relative w-full flex items-center
justify-center">
            <div
              ref={boardRef}
              className="grid bg-white"
              style={{
                gridTemplateColumns: `repeat(${totalBoardWidth},
${gridSize}px)`,
                gridTemplateRows: `repeat(${totalBoardHeight}, ${gridSize}px)`,
                width: totalBoardWidthPx,
                height: totalBoardHeightPx,
                position: "relative",
              }}
            >
              {renderGrid()}
              <ExitMarker position={selectedEdgeGrid} gridSize={gridSize} />
            </div>
          </div>

          {cars.map((car) => (
            <DraggableCar
              key={car.id}
              id={car.id}
              width={car.isVertical ? gridSize : car.size * gridSize}
              height={car.isVertical ? car.size * gridSize : gridSize}
              minTop={0}
              maxTop={boardHeightPx - (car.isVertical ? car.size : 1) *
gridSize + 1.75 * gridSize}
              minLeft={0}
              maxLeft={boardWidthPx - (car.isVertical ? 1 : car.size) *
gridSize + 1.75 * gridSize}
              initialTop={(car.initialTop + 1) * gridSize}
              initialLeft={(car.initialLeft + 1) * gridSize}
              parentRef={boardRef}
              onPositionChange={updateCarPosition}
```

```
              inputGridSize={gridSize}
              deleteCarById={deleteCarById}
              isPrimary={car.isPrimary}
              isExecutingMove={
                isAnimatingStep &&
                solutionMoves.length > 0 &&
                (isReverse ? currentSolutionStep < solutionMoves.length - 1 &&
solutionMoves[currentSolutionStep + 1]?.piece.id === car.id :
solutionMoves[currentSolutionStep]?.piece.id === car.id)
              }
              moveDirection={
                isAnimatingStep && solutionMoves.length > 0
                  ? isReverse
                    ? currentSolutionStep < solutionMoves.length - 1 &&
solutionMoves[currentSolutionStep + 1]?.piece.id === car.id
                      ? getReverseDirection(solutionMoves[currentSolutionStep +
1].direction)
                      : undefined
                    : solutionMoves[currentSolutionStep]?.piece.id === car.id
                    ? solutionMoves[currentSolutionStep].direction
                    : undefined
                  : undefined
              }
              moveSteps={
                isAnimatingStep && solutionMoves.length > 0
                  ? isReverse
                    ? currentSolutionStep < solutionMoves.length - 1 &&
solutionMoves[currentSolutionStep + 1]?.piece.id === car.id
                      ? solutionMoves[currentSolutionStep + 1].steps
                      : undefined
                    : solutionMoves[currentSolutionStep]?.piece.id === car.id
                    ? solutionMoves[currentSolutionStep].steps
                    : undefined
                  : undefined
              }
            />
          ))}
        </>
```

```
      )}
    </main>
  );
}

export default App;
```

Control Panel

```
import { useState, useRef } from "react";
import type { Car, EdgeGrid, PieceMap, Board, Move } from "../lib/types";
import { parseFileContents } from "../lib/helpers/validateInput";
import { aStar } from "../lib/algo/aStar";
import { gbfs } from "../lib/algo/gbfs";
import { ucs } from "../lib/algo/ucs";
import { distanceToExit } from "../lib/heuristics/distanceToExit";
import { blockingCars } from "../lib/heuristics/blockingCars";
import { recursiveBlockers } from "../lib/heuristics/recursiveBlockers";
import { combinedHeuristic } from "../lib/heuristics/combinedHeuristic";
import { moveNeededEstimate } from "../lib/heuristics/moveNeededEstimate";
import { downloadSolutionFile, generateBoardStates } from
"../lib/helpers/output";
import { beamSearch } from "../lib/algo/beamSearch";

interface ControlPanelProps {
  boardWidth: number;
  setBoardWidth: (width: number) => void;
  boardHeight: number;
  setBoardHeight: (height: number) => void;
  gridSize: number;
  cars: Car[];
  setCars: (cars: Car[]) => void;
  selectedEdgeGrid: EdgeGrid | null;
  setSelectedEdgeGrid: (grid: EdgeGrid | null) => void;
  totalBoardWidth: number;
```

```
  totalBoardHeight: number;
  solutionMoves: Move[];
  setSolutionMoves: (moves: Move[]) => void;
  solutionStep: number;
  setSolutionStep: (step: number) => void;
  convertCarsToBoard: () => { board: Board; pieces: PieceMap; overlaps: boolean
};
  isAnimatingStep?: boolean;
  setIsAnimatingStep?: (isAnimating: boolean) => void;
  isAutoPlaying?: boolean;
  startAutoPlay?: () => void;
  stopAutoPlay?: () => void;
  originalBoardState: Car[];
  setOriginalBoardState: (cars: Car[]) => void;
  setIsReverse: (bool: boolean) => void;
  setIsDisplayable: (bool: boolean) => void;
}

const ControlPanel = ({
  boardWidth,
  setBoardWidth,
  boardHeight,
  setBoardHeight,
  cars,
  setCars,
  selectedEdgeGrid,
  setSelectedEdgeGrid,
  totalBoardWidth,
  totalBoardHeight,
  solutionMoves,
  setSolutionMoves,
  solutionStep,
  setSolutionStep,
  convertCarsToBoard,
  isAnimatingStep,
  isAutoPlaying,
  startAutoPlay,
  stopAutoPlay,
```

```
  originalBoardState,
  setOriginalBoardState,
  setIsReverse,
  setIsDisplayable,
}: ControlPanelProps) => {
  const [selectedAlgorithm, setSelectedAlgorithm] = useState<string>("aStar");
  const [selectedHeuristic, setSelectedHeuristic] =
useState<string>("combined");
  const [showSolution, setShowSolution] = useState<boolean>(false);
  const [inputCarLength, setInputCarLength] = useState<number>(2);
  const [inputCarOrientation, setInputCarOrientation] =
useState<boolean>(false);
  const [isPrimary, setIsPrimary] = useState<boolean>(false);
  const [exitRow, setExitRow] = useState<number>(0);
  const [exitCol, setExitCol] = useState<number>(0);
  const [isSolving, setIsSolving] = useState<boolean>(false);
  const [nodesFound, setNodesFound] = useState<number>(0);
  const [timeTaken, setTimeTaken] = useState<number>(0);
  const [beamNumber, setBeamNumber] = useState<number>(1);

  const primaryCar = cars.find((car) => car.isPrimary);
  const isInitialState = solutionStep === -1;

  const fileInputRef = useRef<HTMLInputElement>(null);

  const handleFileUpload = (event: React.ChangeEvent<HTMLInputElement>) => {
    setIsDisplayable(true);
    const file = event.target.files?.[0];
    if (!file) return;

    if (file.type !== "text/plain" && !file.name.endsWith(".txt")) {
      alert("Please upload a valid .txt file");
      if (fileInputRef.current) fileInputRef.current.value = "";
      return;
    }

    setCars([]);
    setSelectedEdgeGrid(null);
```

```
    setShowSolution(false);

    const reader = new FileReader();
    reader.onload = (e) => {
      const text = e.target?.result as string;
      const result = parseFileContents(text);

      if (!result.success) {
        alert(result.message);
        return;
      }

      if (result.width) setBoardWidth(result.width);
      if (result.height) setBoardHeight(result.height);
      if (result.exitGrid) {
        setSelectedEdgeGrid(result.exitGrid);
        setExitRow(result.exitGrid.row);
        setExitCol(result.exitGrid.col);
      }
      if (result.newCars) setCars(result.newCars);

      if (fileInputRef.current) fileInputRef.current.value = "";
    };
    setShowSolution(false);
    reader.readAsText(file);
  };

  const getRandomCharacter = () => {
    const excludedChars = ["K", "P", ".", " "];
    const usedChars = cars.map((car) => car.id);

    for (let charCode = 65; charCode <= 90; charCode++) {
      const letter = String.fromCharCode(charCode);
      if (!excludedChars.includes(letter) && !usedChars.includes(letter)) {
        return letter;
      }
    }
  };
```

```
  const addCar = (size: number, isVertical: boolean, isPrimary: boolean) => {
    if (isPrimary && primaryCar) {
      alert("Only one primary car is allowed. Delete the existing primary car
first.");
      return;
    }

    const normalCarsCount = cars.filter((car) => !car.isPrimary).length;

    if (!isPrimary && normalCarsCount >= 24) {
      alert("Maximum of 24 normal cars reached");
      return;
    }

    const newCar: Car = {
      id: isPrimary ? "P" : getRandomCharacter() || "*",
      isVertical,
      size,
      initialLeft: 0,
      initialTop: 0,
      isPrimary,
    };

    setCars([...cars, newCar]);
  };

  const solveBoard = () => {
    if (!selectedEdgeGrid) {
      alert("Please select an exit position first");
      return;
    }

    if (!cars.some((car) => car.isPrimary)) {
      alert("Please add a primary car first");
      return;
    }
```

```
    const { overlaps, board, pieces } = convertCarsToBoard();

    if (overlaps) {
      alert("There are overlapping cars, fix the positioning!");
      return;
    }

    const primaryCar = cars.find((car) => car.isPrimary);

    if (primaryCar) {
      const exitPosition = {
        row: selectedEdgeGrid.row - 1,
        col: selectedEdgeGrid.col - 1,
      };

      if (!primaryCar.isVertical) {
        if (primaryCar.initialTop !== exitPosition.row) {
          alert("Error: Primary car must be in the same row as the exit when
oriented horizontally.");
          return;
        }

        const blockingCars = cars.filter(
          (car) =>
            !car.isPrimary &&
            !car.isVertical &&
            car.initialTop === exitPosition.row &&
            ((exitPosition.col > primaryCar.initialLeft && car.initialLeft >
primaryCar.initialLeft + primaryCar.size - 1 && car.initialLeft <=
exitPosition.col) ||
              (exitPosition.col < primaryCar.initialLeft && car.initialLeft +
car.size - 1 >= exitPosition.col && car.initialLeft < primaryCar.initialLeft))
        );

        if (blockingCars.length > 0) {
          alert(`Warning: There are ${blockingCars.length} horizontal cars
blocking the direct path to the exit. The puzzle is unsolvable.`);
          return;
```

```
        }
      }

      if (primaryCar.isVertical) {
        if (primaryCar.initialLeft !== exitPosition.col) {
          alert("Error: Primary car must be in the same column as the exit when
oriented vertically.");
          return;
        }

        const blockingCars = cars.filter(
          (car) =>
            !car.isPrimary &&
            car.isVertical &&
            car.initialLeft === exitPosition.col &&
            ((exitPosition.row > primaryCar.initialTop && car.initialTop >
primaryCar.initialTop + primaryCar.size - 1 && car.initialTop <=
exitPosition.row) ||
              (exitPosition.row < primaryCar.initialTop && car.initialTop +
car.size - 1 >= exitPosition.row && car.initialTop < primaryCar.initialTop))
        );

        if (blockingCars.length > 0) {
          alert(`Warning: There are ${blockingCars.length} vertical cars
blocking the direct path to the exit. The puzzle is unsolvable.`);
          return;
        }
      }
    }

    setOriginalBoardState([...cars]);

    setIsSolving(true);
    setSolutionMoves([]);
    setSolutionStep(-1);

    let heuristicFunction;
    switch (selectedHeuristic) {
```

```
      case "distance":
        heuristicFunction = distanceToExit;
        break;
      case "blocking":
        heuristicFunction = blockingCars;
        break;
      case "recursive":
        heuristicFunction = recursiveBlockers;
        break;
      case "moveNeeded":
        heuristicFunction = moveNeededEstimate;
        break;
      default:
        heuristicFunction = combinedHeuristic;
    }

    let result;
    try {
      switch (selectedAlgorithm) {
        case "gbfs":
          result = gbfs(board, pieces, heuristicFunction);
          break;
        case "beam":
          result = beamSearch(board, pieces, heuristicFunction, beamNumber);
          break;
        case "ucs":
          result = ucs(board, pieces);
          break;
        default:
          result = aStar(board, pieces, heuristicFunction);
      }

      if (result && result.found) {
        setNodesFound(result.nodesVisited);
        setTimeTaken(result.timeTaken);
        setSolutionMoves(result.moveHistory);
        setShowSolution(true);
        alert(`Solution found in ${result.moveHistory.length} moves!`);
```

```
      } else {
        alert("No solution found!");
      }
    } catch (error) {
      console.error("Error solving board:", error);
      alert(`Error: ${error instanceof Error ? error.message :
String(error)}`);
    } finally {
      setIsSolving(false);
    }
  };

  return (
    <div className="w-full max-w-4xl bg-white p-4 rounded-lg shadow-md">
      <div className="flex flex-wrap items-center justify-between gap-2 mb-3
border-b pb-3">
        <div className="flex items-center gap-2">
          <label className="cursor-pointer bg-blue-500 hover:bg-blue-600
text-white px-3 py-1 rounded text-sm">
            Upload
            <input ref={fileInputRef} type="file" accept=".txt,text/plain"
onChange={handleFileUpload} className="hidden" />
          </label>

          <div className="flex items-center">
            <label className="mr-1 text-sm">Board Width:</label>
            <input type="number" min="2" value={boardWidth} onChange={(e) =>
setBoardWidth(parseInt(e.target.value))} className="w-12 p-1 border
border-gray-300 rounded text-sm" />
          </div>

          <div className="flex items-center">
            <label className="mr-1 text-sm">Board Height:</label>
            <input type="number" min="2" value={boardHeight} onChange={(e) =>
setBoardHeight(parseInt(e.target.value))} className="w-12 p-1 border
border-gray-300 rounded text-sm" />
          </div>
        </div>
```

```
        <div className="flex items-center gap-2">
          <div className="flex items-center">
            <label className="mr-1 text-sm">Exit Row:</label>
            <input
              type="number"
              min={0}
              max={totalBoardHeight - 1}
              value={exitRow}
              onChange={(e) => {
                const val = Math.max(0, Math.min(totalBoardHeight - 1,
parseInt(e.target.value) || 0));
                setExitRow(val);
                let newCol = exitCol;
                if (val !== 0 && val !== totalBoardHeight - 1) {
                  if (exitCol !== 0 && exitCol !== totalBoardWidth - 1) {
                    newCol = 0;
                  }
                }
                if ((val === 0 || val === totalBoardHeight - 1) && (newCol ===
0 || newCol === totalBoardWidth - 1)) {
                  newCol = Math.min(Math.max(1, newCol), totalBoardWidth - 2);
                }
                setExitCol(newCol);
                setSelectedEdgeGrid({ row: val, col: newCol });
              }}
              className="w-12 p-1 border border-gray-300 rounded text-sm"
            />
            <span className="mx-1 text-sm">Exit Col:</span>
            <input
              type="number"
              min={0}
              max={totalBoardWidth - 1}
              value={exitCol}
              onChange={(e) => {
                const val = Math.max(0, Math.min(totalBoardWidth - 1,
parseInt(e.target.value) || 0));
                setExitCol(val);
```

```
              let newRow = exitRow;
              if (val !== 0 && val !== totalBoardWidth - 1) {
                if (exitRow !== 0 && exitRow !== totalBoardHeight - 1) {
                  newRow = 0;
                }
              }
              if ((val === 0 || val === totalBoardWidth - 1) && (newRow === 0
|| newRow === totalBoardHeight - 1)) {
                newRow = Math.min(Math.max(1, newRow), totalBoardHeight - 2);
              }
              setExitRow(newRow);
              setSelectedEdgeGrid({ row: newRow, col: val });
            }}
            className="w-12 p-1 border border-gray-300 rounded text-sm"
          />
        </div>

        <button
          onClick={() => {
            setCars([]);
            setSelectedEdgeGrid(null);
            setShowSolution(false);
          }}
          className="px-2 py-1 bg-red-500 text-white rounded text-sm
hover:bg-red-600"
        >
          Clear
        </button>
      </div>
    </div>
    <div className="flex flex-wrap items-center justify-between gap-2 mb-3
border-b pb-3">
      <div className="flex items-center gap-2">
        <label className="text-sm font-medium">Car:</label>
        <div className="flex items-center">
          <label className="mr-1 text-sm">Size:</label>
          <input
            type="number"
```

```
              min="2"
              value={inputCarLength}
              onChange={(e) => setInputCarLength(Math.max(2, Math.min(10,
parseInt(e.target.value) || 2)))}
              className="w-12 p-1 border border-gray-300 rounded text-sm"
            />
          </div>

          <button className="px-2 py-1 border border-gray-300 rounded text-sm
hover:bg-gray-100 cursor-pointer" onClick={() =>
setInputCarOrientation(!inputCarOrientation)}>
            {inputCarOrientation ? "Vertical" : "Horizontal"}
          </button>

          <div className="flex items-center">
            <label className="mr-1 text-sm">Primary</label>
            <input
              type="checkbox"
              checked={isPrimary}
              onChange={() => {
                if (!primaryCar || isPrimary) {
                  setIsPrimary(!isPrimary);
                }
              }}
              disabled={primaryCar && !isPrimary}
              className="w-4 h-4"
            />
          </div>
        </div>

        <button onClick={() => addCar(inputCarLength, inputCarOrientation,
isPrimary)} className="px-3 py-1 bg-yellow-500 text-white rounded text-sm
hover:bg-yellow-600">
          Add Car
        </button>
      </div>
      <div className="flex flex-wrap items-center justify-between gap-2">
        <div className="flex items-center gap-2">
```

```
          <label className="text-sm font-medium">Solver:</label>
          <select value={selectedAlgorithm} onChange={(e) =>
setSelectedAlgorithm(e.target.value)} className="p-1 border border-gray-300
rounded text-sm" disabled={isSolving}>
            <option value="aStar">A*</option>
            <option value="gbfs">Greedy</option>
            <option value="ucs">UCS</option>
            <option value="beam">Beam</option>
          </select>
          {selectedAlgorithm !== "ucs" && (
            <select
              value={selectedHeuristic}
              onChange={(e) => setSelectedHeuristic(e.target.value)}
              className="p-1 border border-gray-300 rounded text-sm"
              disabled={isSolving || selectedAlgorithm === "ucs"}
            >
              <option value="combined">Combined</option>
              <option value="distance">Distance</option>
              <option value="blocking">Blocking</option>
              <option value="recursive">Recursive</option>
              <option value="moveNeeded">Move Est.</option>
            </select>
          )}
          {selectedAlgorithm === "beam" && (
            <input
              type="number"
              className="p-1 border border-gray-300 rounded text-sm"
              min={1}
              value={beamNumber}
              onChange={(e) => {
                if (!e.target.value) {
                  setBeamNumber(1);
                } else {
                  setBeamNumber(parseInt(e.target.value));
                }
              }}
            />
          )}
```

```
        </div>

        <button onClick={solveBoard} disabled={isSolving} className={`px-3 py-1
${isSolving ? "bg-gray-400" : "bg-green-500 hover:bg-green-600"} text-white
rounded text-sm`}>
          {isSolving ? "Solving..." : "Solve"}
        </button>
      </div>
      {showSolution && solutionMoves.length > 0 && (
        <div className="mt-3 pt-3 border-t">
          <div className="flex items-center justify-between">
            <span className="text-sm font-medium">Solution:
{solutionMoves.length} moves</span>
            <span className="text-sm font-medium">Nodes: {nodesFound}</span>
            <span className="text-sm font-medium">Time:
{timeTaken.toFixed(3)}ms</span>

            <div className="flex items-center gap-1">
              <button
                onClick={() => {
                  if (isAutoPlaying && stopAutoPlay) {
                    stopAutoPlay();
                  }
                  setIsReverse(true);
                  setSolutionStep(Math.max(-1, solutionStep - 1));
                  if (solutionStep === -1 && originalBoardState.length > 0) {
                    setCars([...originalBoardState]);
                  }
                }}
                className="px-2 py-1 bg-blue-500 text-white rounded text-xs
disabled:bg-gray-300"
              >
                ←
              </button>
              <span className="text-xs">
                Step {solutionStep + 1}/{solutionMoves.length}
              </span>
              <button
```

```
                onClick={() => {
                  if (isAutoPlaying && stopAutoPlay) {
                    stopAutoPlay();
                  }
                  setIsReverse(false);
                  setSolutionStep(Math.min(solutionMoves.length, solutionStep +
1));
                }}
                disabled={solutionStep === solutionMoves.length - 1}
                className="px-2 py-1 bg-blue-500 text-white rounded text-xs
disabled:bg-gray-300"
              >
                →
              </button>
            </div>
          </div>

          <div className="mt-3 flex justify-center gap-2">
            {!isAutoPlaying ? (
              <button
                onClick={() => {
                  if (startAutoPlay) {
                    startAutoPlay();
                  }
                }}
                disabled={solutionStep === solutionMoves.length ||
isAnimatingStep}
                className="px-3 py-1 bg-green-500 text-white rounded text-xs
disabled:bg-gray-300 hover:bg-green-600"
              >
                Auto Play
              </button>
            ) : (
              <button
                onClick={() => {
                  if (stopAutoPlay) {
                    stopAutoPlay();
                  }
```

```
              }}
              className="px-3 py-1 bg-red-500 text-white rounded text-xs
hover:bg-red-600"
            >
              Stop
            </button>
          )}
          <button
            onClick={() => {
              if (isAutoPlaying && stopAutoPlay) {
                stopAutoPlay();
              }
              setTimeout(() => {
                if (originalBoardState.length > 0) {
                  setCars([...originalBoardState]);
                }
                setSolutionStep(-1);
              }, 500);
            }}
            disabled={isAnimatingStep || isInitialState}
            className="px-3 py-1 bg-gray-500 text-white rounded text-xs
disabled:bg-gray-300 hover:bg-gray-600"
          >
            Reset Board
          </button>
        </div>
        <button
          onClick={() => {
            const { board, pieces } = convertCarsToBoard();
            downloadSolutionFile(board, solutionMoves,
generateBoardStates(board, solutionMoves, pieces), `unblock_car_solution.txt`);
          }}
          disabled={solutionMoves.length === 0}
          className="px-3 py-1 bg-purple-500 text-white rounded text-xs
disabled:bg-gray-300 hover:bg-purple-600"
        >
          Download Solution
        </button>
```

```
        </div>
      )}
    </div>
  );
};

export default ControlPanel;
```

Draggable Car

```
import { useState, useCallback, useEffect, useRef } from "react";
import type { Direction } from "../lib/types";

interface DraggableCarProps {
  id: string;
  width: number;
  height: number;
  minTop: number;
  maxTop: number;
  minLeft: number;
  maxLeft: number;
  initialTop: number;
  initialLeft: number;
  onPositionChange?: (id: string, top: number, left: number) => void;
  parentRef: React.RefObject<HTMLDivElement | null>;
  inputGridSize: number;
  deleteCarById: (id: string) => void;
  isPrimary: boolean;
  moveDirection?: Direction | null;
  moveSteps?: number;
  isExecutingMove?: boolean;
}

const DraggableCar = ({
```

```
  id,
  width,
  height,
  minTop,
  maxTop,
  minLeft,
  maxLeft,
  initialTop,
  initialLeft,
  onPositionChange,
  parentRef,
  inputGridSize,
  deleteCarById,
  isPrimary,
  moveDirection,
  moveSteps,
  isExecutingMove,
}: DraggableCarProps) => {
  const [position, setPosition] = useState({ top: 0, left: 0 });
  const [dragging, setDragging] = useState(false);
  const [offset, setOffset] = useState({ x: 0, y: 0 });
  const [parentBounds, setParentBounds] = useState({ top: 0, left: 0 });
  const [zIndex, setZIndex] = useState(1);
  const [isAnimating, setIsAnimating] = useState(false);
  const animationTimerRef = useRef<number | null>(null);

  useEffect(() => {
    if (parentRef.current) {
      const parentBounds = parentRef.current.getBoundingClientRect();
      setPosition({
        top: parentBounds.top + minTop + initialTop,
        left: parentBounds.left + minLeft + initialLeft,
      });
      setParentBounds({
        top: parentBounds.top,
        left: parentBounds.left,
      });
    }
```

```
  }, [parentRef, minTop, minLeft, initialTop, initialLeft]);

  const isWithinParentBounds = useCallback(
    (top: number, left: number) => {
      if (!parentRef.current) return false;
      return top >= parentBounds.top + minTop && top <= parentBounds.top +
maxTop && left >= parentBounds.left + minLeft && left <= parentBounds.left +
maxLeft;
    },
    [parentBounds, minTop, maxTop, minLeft, maxLeft, parentRef]
  );

  const snapToGrid = useCallback(
    (position: number, min: number, max: number, isHorizontal: boolean) => {
      const parentPosition = isHorizontal ? parentBounds.left + inputGridSize :
parentBounds.top + inputGridSize;
      const relativePosition = position - (parentPosition + min);
      const snappedRelativePosition = Math.round(relativePosition /
inputGridSize) * inputGridSize;
      const snappedPosition = parentPosition + min + snappedRelativePosition;

      return Math.max(parentPosition + min, Math.min(parentPosition + max,
snappedPosition));
    },
    [parentBounds.top, parentBounds.left, inputGridSize]
  );

  const handleMouseDown = useCallback(
    (e: React.MouseEvent<HTMLDivElement>) => {
      if (isAnimating) return;
      setDragging(true);
      setOffset({
        x: e.clientX - position.left,
        y: e.clientY - position.top,
      });
      setZIndex(10);
    },
    [position, isAnimating]
```

```
  );

  const handleMouseMove = useCallback(
    (e: MouseEvent) => {
      if (!dragging) return;

      let newTop = e.clientY - offset.y;
      let newLeft = e.clientX - offset.x;

      if (window.innerWidth > 768) {
        newTop = Math.max(0, Math.min(newTop, window.innerHeight - height));
      }
      newLeft = Math.max(0, Math.min(newLeft, window.innerWidth - width));

      setPosition({
        top: newTop,
        left: newLeft,
      });
    },
    [dragging, offset, height, width]
  );

  const handleMouseUp = useCallback(() => {
    setDragging(false);
    setZIndex(1);

    setPosition((prev) => {
      if (isWithinParentBounds(prev.top, prev.left)) {
        const newTop = snapToGrid(prev.top, minTop, maxTop, false);
        const newLeft = snapToGrid(prev.left, minLeft, maxLeft, true);
        setTimeout(() => {
          const relativeTop = Math.round((newTop - parentBounds.top - minTop) /
inputGridSize) - 1;
          const relativeLeft = Math.round((newLeft - parentBounds.left -
minLeft) / inputGridSize) - 1;

          if (onPositionChange) {
            onPositionChange(id, relativeTop, relativeLeft);
```

```
          }
        }, 100);
        return {
          top: newTop,
          left: newLeft,
        };
      }
      deleteCarById(id);
      return prev;
    });
  }, [minTop, maxTop, minLeft, maxLeft, snapToGrid, isWithinParentBounds, id,
onPositionChange, parentBounds, deleteCarById, inputGridSize]);

  useEffect(() => {
    if (dragging) {
      window.addEventListener("mousemove", handleMouseMove);
      window.addEventListener("mouseup", handleMouseUp);
    } else {
      window.removeEventListener("mousemove", handleMouseMove);
      window.removeEventListener("mouseup", handleMouseUp);
    }

    return () => {
      window.removeEventListener("mousemove", handleMouseMove);
      window.removeEventListener("mouseup", handleMouseUp);
    };
  }, [dragging, handleMouseMove, handleMouseUp]);

  useEffect(() => {
    return () => {
      if (animationTimerRef.current !== null) {
        clearTimeout(animationTimerRef.current);
      }
    };
  }, []);

  useEffect(() => {
    if (isExecutingMove && moveDirection && moveSteps && moveSteps > 0 &&
```

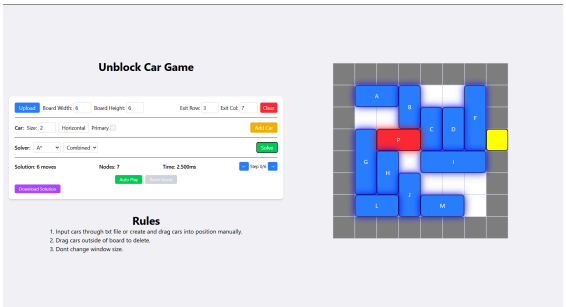
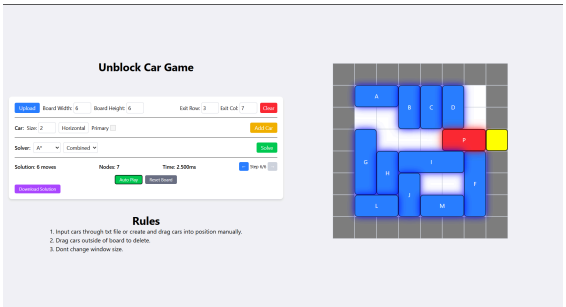
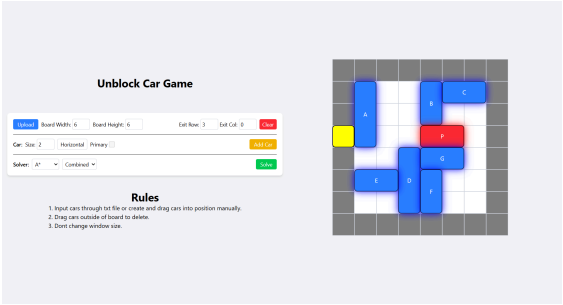
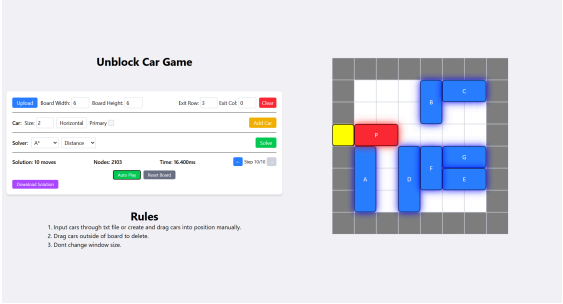
```
!isAnimating) {
      if (animationTimerRef.current !== null) {
        clearTimeout(animationTimerRef.current);
      }

      setIsAnimating(true);
      setZIndex(5);

      let newTop = position.top;
      let newLeft = position.left;

      switch (moveDirection) {
        case "Up":
          newTop -= moveSteps * inputGridSize;
          break;
        case "Down":
          newTop += moveSteps * inputGridSize;
          break;
        case "Left":
          newLeft -= moveSteps * inputGridSize;
          break;
        case "Right":
          newLeft += moveSteps * inputGridSize;
          break;
      }

      newTop = Math.max(parentBounds.top + minTop, Math.min(parentBounds.top +
maxTop, newTop));
      newLeft = Math.max(parentBounds.left + minLeft,
Math.min(parentBounds.left + maxLeft, newLeft));

      setPosition({
        top: newTop,
        left: newLeft,
      });

      animationTimerRef.current = window.setTimeout(() => {
        const snappedTop = snapToGrid(newTop, minTop, maxTop, false);
```

```
        const snappedLeft = snapToGrid(newLeft, minLeft, maxLeft, true);

        setPosition({
          top: snappedTop,
          left: snappedLeft,
        });

        const relativeTop = Math.round((snappedTop - parentBounds.top - minTop)
/ inputGridSize) - 1;
        const relativeLeft = Math.round((snappedLeft - parentBounds.left -
minLeft) / inputGridSize) - 1;

        if (onPositionChange) {
          onPositionChange(id, relativeTop, relativeLeft);
        }

        setIsAnimating(false);
        setZIndex(1);
        animationTimerRef.current = null;
      }, 300);
    }
  }, [isExecutingMove, moveDirection, moveSteps, isAnimating, inputGridSize,
position, parentBounds, minTop, maxTop, minLeft, maxLeft, onPositionChange, id,
snapToGrid]);

  return (
    <div
      className="absolute cursor-move rounded-lg"
      style={{
        top: position.top,
        left: position.left,
        zIndex,
        width,
        height,
        transition: isAnimating ? "top 0.3s ease-out, left 0.3s ease-out" :
"none",
      }}
      onMouseDown={handleMouseDown}
```

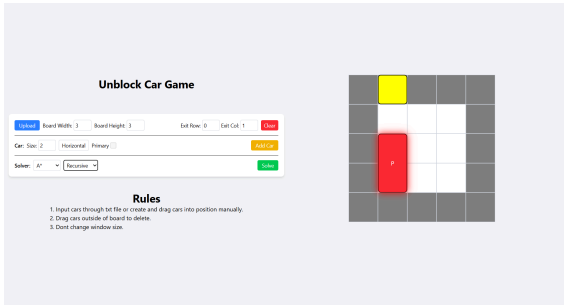
```
      data-position={`${position.top},${position.left}`}
      data-parent-bounds={`${parentBounds.top}`}
    >
      <div
        className={`text-white flex justify-center items-center text-center
w-full h-full rounded-lg border-2 ${isPrimary ? "border-red-950 bg-red-500" :
"border-blue-950 bg-blue-500"}`}
        style={{
          boxShadow: isAnimating
            ? isPrimary
              ? "0 0 40px 10px gold"
              : "0 0 40px 10px cyan"
            : dragging
            ? isPrimary
              ? "0 0 40px 10px blue"
              : "0 0 40px 10px red"
            : isPrimary
            ? "0 0 20px 0 red"
            : "0 0 20px 0 blue",
          transition: "box-shadow 0.2s",
        }}
      >
        {id}
      </div>
    </div>
  );
};

export default DraggableCar;
```

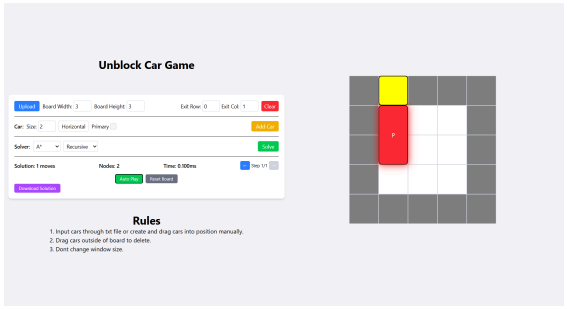
BAB V:
PENGUJIAN

Input Valid

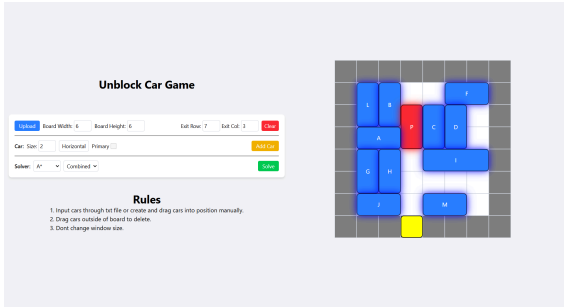
Name	Input	Output
A*	6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	 
	6 6 7 A..BCC A..B.. KA..PP. ..DGG. EEDF.. ..DF..	 

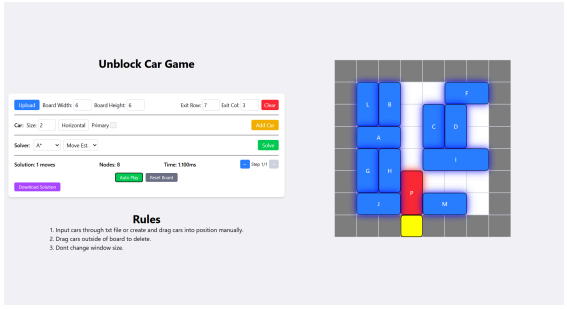
```
3 3
0
K
...
P..
P..
```

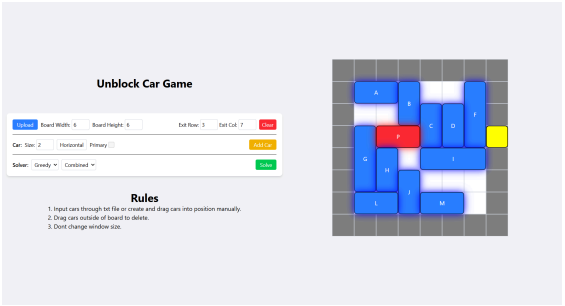
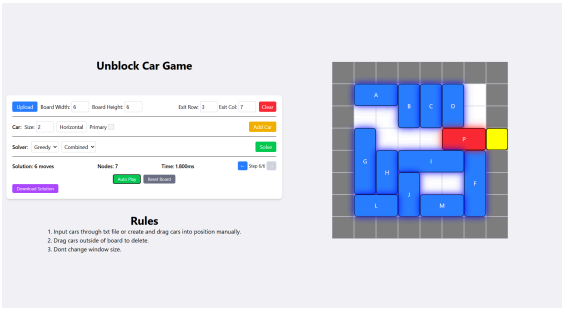
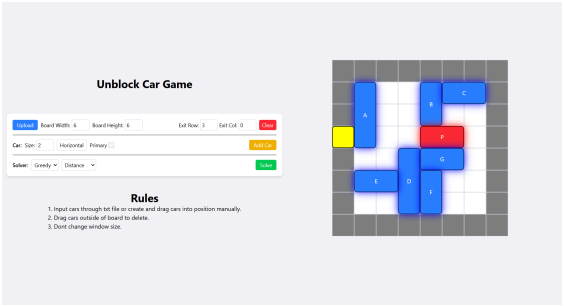
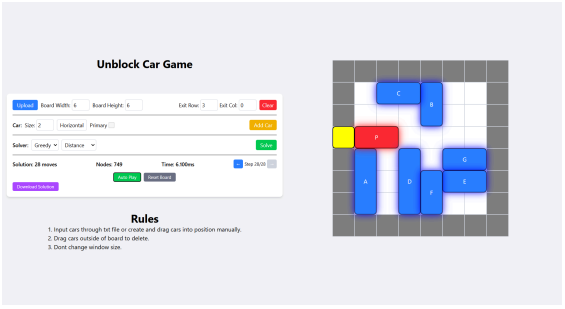
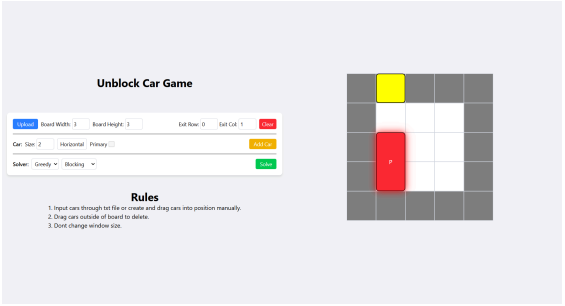


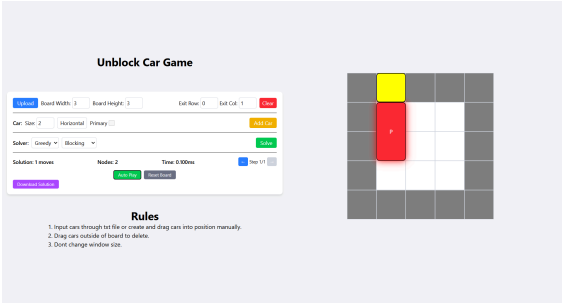
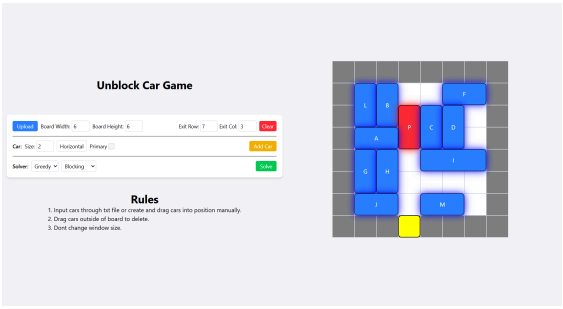


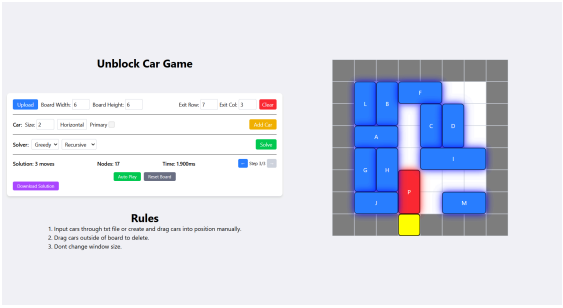
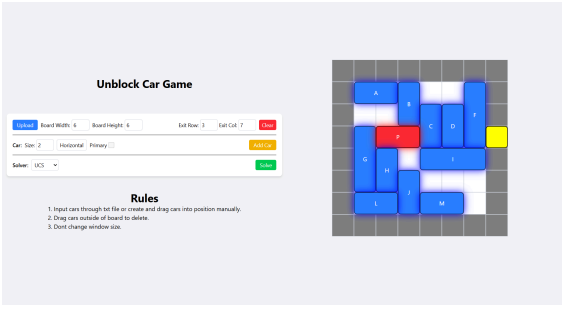

```
6 6
11
LB..FF
LBPCD.
AAPCD.
GH.III
GH....
JJ.MM.
  K
```

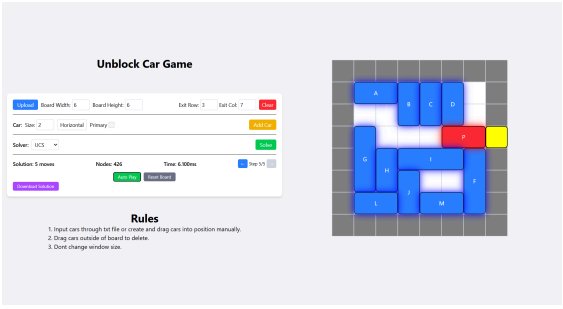


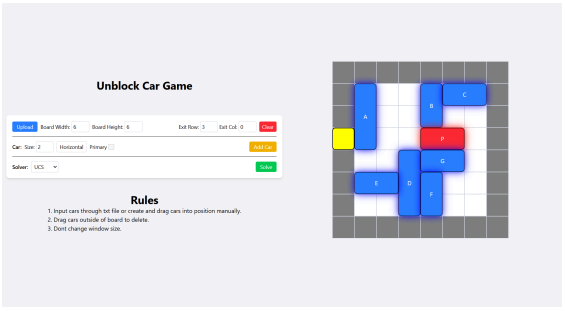
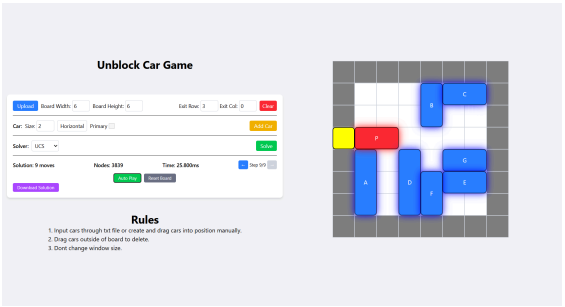
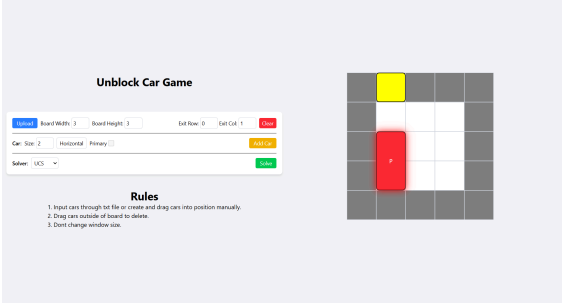
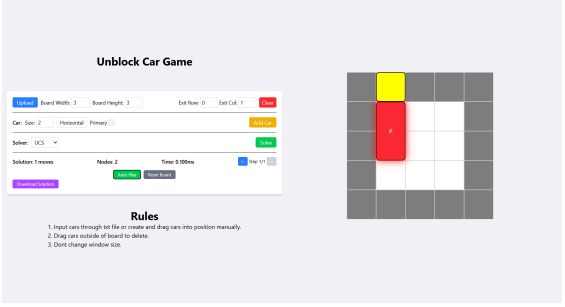


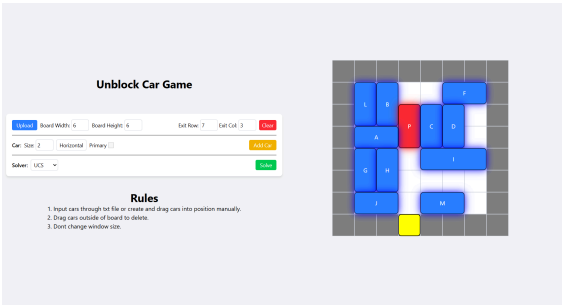
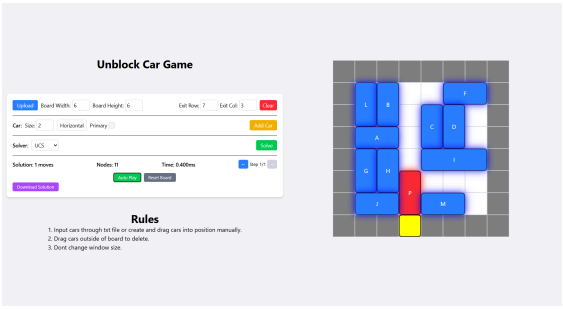
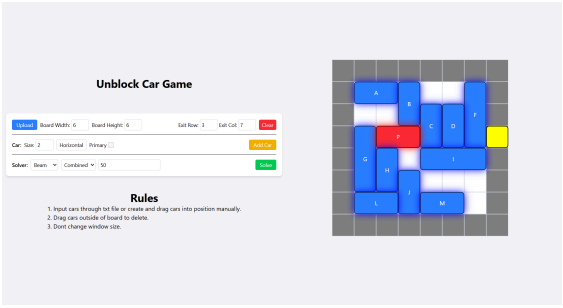
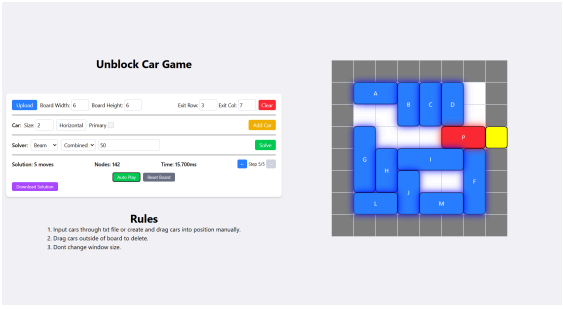
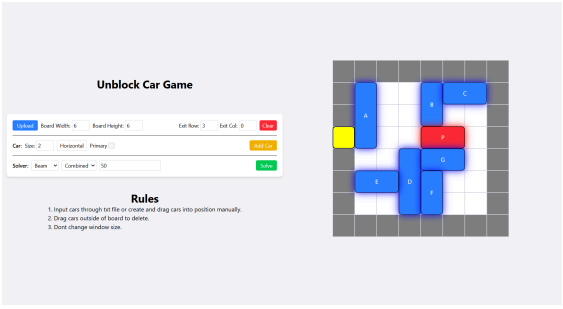
GBFS	6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	 
	6 6 7 A..BCC A..B.. KA..PP. ..DGG. EEDF.. ..DF..	 
	3 3 0 K ... P.. P..	

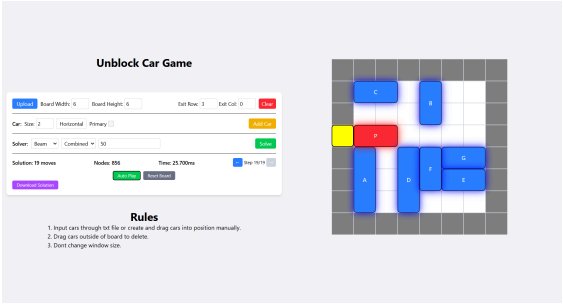
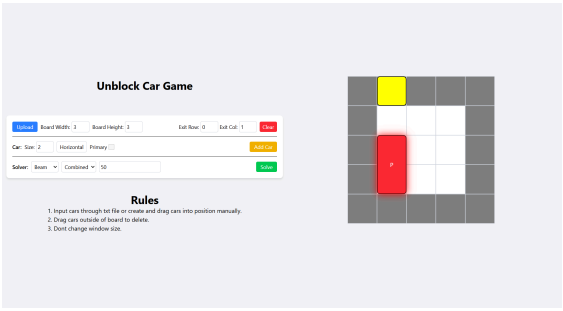
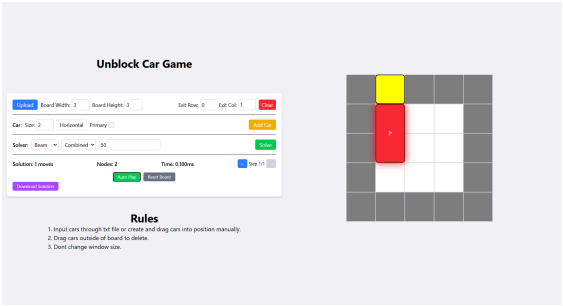
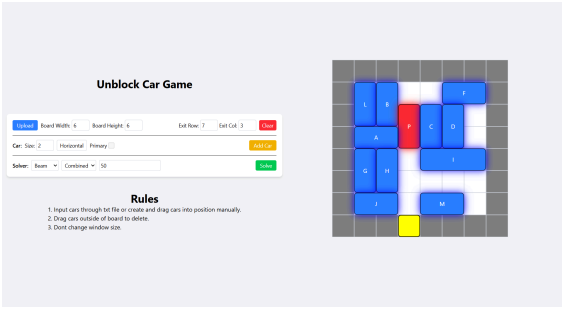
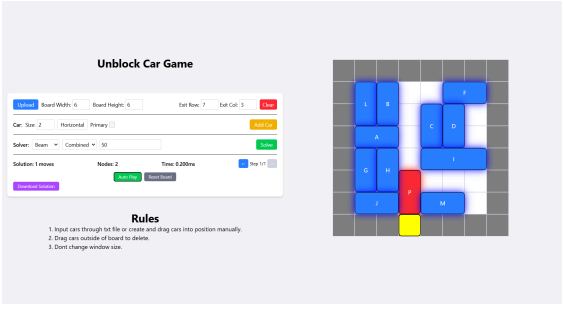
<table>
<tr><td></td><td></td><td> 
</td></tr>
<tr><td></td><td>6 6
11
LB..FF
LBPCD.
AAPCD.
GH.III
GH....
JJ.MM.
K</td><td> 

 
</td></tr>
<tr><td>UCS</td><td>6 6
11
AAB..F
..BCDF
GPPCDFK
GH.III
GHJ...
LLJMM.</td><td> 

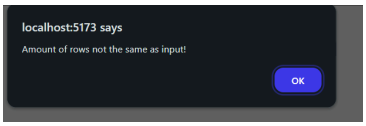
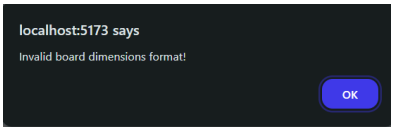
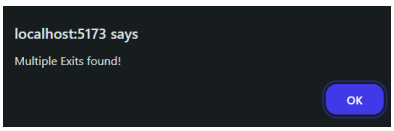
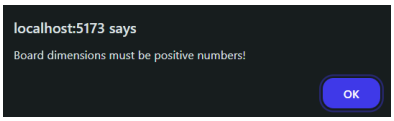
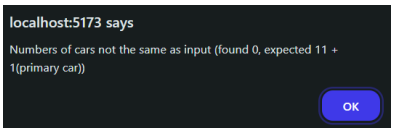
 
</td></tr>
</table>

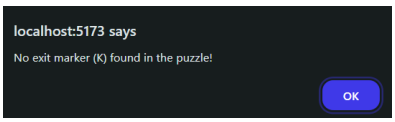
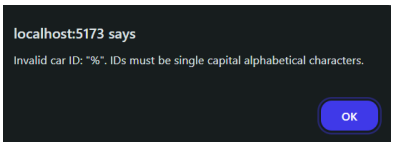
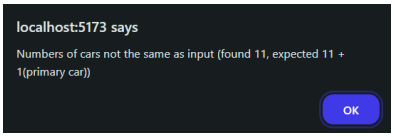
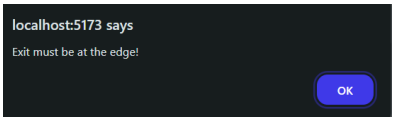
	6 6 7 A..BCC A..B.. KA..PP. ..DGG. EEDF.. ..DF..	 
	3 3 0 K ... P.. P..	 

	6 6 11 LB..FF LBPCD. AAPCD. GH.III GH.... JJ.MM. K	 
Beam Search	6 6 11 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	 
	6 6 7 A..BCC A..B.. KA..PP. ..DGG. EEDF.. ..DF..	

		
	3 3 0 K ... P.. P..	 
	6 6 11 LB..FF LBPCD. AAPCD. GH.III GH.... JJ.MM. K	 

Input Invalid

Input	Output	Description
4 4 0 ..PPK		Jumlah baris tidak sesuai input
		File kosong
6 6 10 K PAB..F PABCDF GGGCDF .H.... .HJ... LLJMM.K		Exit lebih dari 1
-3 -3 0 K.PP		Besar board negatif
6 6 11		Tidak ada mobil

3 3 0 .PP		Tidak ada exit
3 3 1 K %%. P.. P..		ID mobil tidak huruf alfabet kapital
6 6 11 AAB..F ..BCDF GPPCDFK GH.... GHJ... LLJMM.		
4 4 0 ...P ...P ....K		Exit tidak di luar board

1. Kompleksitas Algoritma Uniform Cost Search

Algoritma UCS bekerja dengan menggunakan suatu *priority queue*. *Priority queue* termasuk struktur data yang cukup kompleks karena perlu melakukan pengurutan setiap kali ada elemen baru yang masuk. Karena diimplementasikan menggunakan *binary heap*, maka operasi *push* dan *pop* memiliki kompleksitas waktu O($log\ n$) di mana n adalah jumlah simpul yang ada di dalam *priority queue*. Untuk tiap simpul, UCS akan memeriksa semua gerakan yang valid. Jika b adalah nilai faktor percabangan sementara m adalah nilai kedalaman maksimal, maka pada kasus terburuk kompeksitas

algoritmanya adalah O(b^m). Karena tiap simpul menggunakan *priority queue*, maka operasi *push* dan *pop* menyebabkan kompleksitas waktunya menjadi O($b^m \log n$).

Dari segi kompleksitas ruang, algoritma UCS memanfaatkan *list* yaitu *open list* dan *closed list*. *Open list* dapat memuat maksimal b^m simpul sekaligus pada kasus terburuk. *Closed list* menyimpan simpul-simpul yang telah ditemukan sehingga juga memuat maksimal b^m simpul. Kompleksitas ruang dari UCS adalah O(b^m).

2. Kompleksitas Algoritma Greedy Best First Search

Seperti halnya algoritma UCS, algoritma GBFS juga menggunakan priority queue. Karena diimplementasikan dengan binary heap, operasi push dan pop memiliki kompleksitas waktu O($\log n$), di mana n adalah jumlah simpul dalam priority queue. Untuk setiap simpul, GBFS akan mengevaluasi semua gerakan yang valid. Jika b adalah faktor percabangan dan m adalah kedalaman maksimal, maka pada kasus terburuk kompleksitas waktu algoritma GBFS adalah O(b^m). Karena setiap simpul dikelola dalam priority queue, operasi push dan pop menambah kompleksitas waktu menjadi O($b^m \log n$).

Dari sisi kompleksitas ruang, algoritma GBFS menggunakan dua list, yaitu open list dan closed list. Open list dapat menampung hingga b^m simpul pada kasus terburuk, sementara closed list menyimpan simpul yang sudah diekspansi, yang juga dapat menampung hingga b^m simpul. Sehingga, kompleksitas ruang GBFS adalah O(b^m). Meskipun kompleksitas waktu dan ruang GBFS serupa dengan UCS, dalam praktiknya GBFS biasanya lebih efisien karena lebih fokus pada simpul yang lebih dekat dengan tujuan, sementara UCS menjelajahi seluruh ruang pencarian.

3. Kompleksitas Algoritma A*

Algoritma A* bekerja dengan menggunakan priority queue yang menyimpan simpul berdasarkan nilai evaluasi f(n) = g(n) + h(n), di mana g(n) adalah biaya untuk mencapai simpul n dari simpul awal danh(n) adalah perkiraan biaya untuk mencapai tujuan dari simpul n (heuristik). A* bekerja dengan menggunakan priority queue untuk mengelola simpul yang akan dieksplorasi. Priority queue ini diimplementasikan dengan *binary heap*, sehingga operasi push dan pop masing-masing memiliki kompleksitas waktu O($\log n$), di mana n adalah jumlah simpul yang ada dalam priority queue.

Untuk setiap simpul yang diekspansi, A* akan memeriksa semua gerakan yang valid. Jika b adalah faktor percabangan (jumlah kemungkinan gerakan untuk setiap simpul) dan m adalah kedalaman maksimal (jumlah langkah dari simpul awal hingga tujuan), maka dalam kasus terburuk, A* akan mengeksplorasi semua simpul hingga kedalaman m. Karena setiap simpul menggunakan priority queue, operasi push dan pop pada setiap simpul menyebabkan kompleksitas waktu menjadi O($b^m \log n$).

A* memanfaatkan dua list utama, yaitu *open list* dan *closed list*. Karena A* harus menyimpan simpul di open list dan closed list, maka kompleksitas ruang algoritma A* adalah O(b^m).

4. Kompleksitas Algoritma Beam Search

Pada setiap tingkat kedalaman, *beam search* melakukan eksplorasi pada semua gerakan valid dari setiap simpul dalam beam. Jika b adalah faktor percabangan (jumlah gerakan yang valid), maka pada tingkat kedalaman pertama, beam search akan mengembangkan hingga b simpul. Setelah itu, hanya *beamWidth* simpul dengan nilai heuristik terbaik yang dipertahankan di beam. Namun, meskipun hanya *beamWidth* simpul yang dipertahankan pada setiap kedalaman, setiap simpul tetap akan mengembangkan b gerakan untuk tingkat kedalaman berikutnya, yang menyebabkan eksplorasi tetap dilakukan pada b simpul dari tiap simpul dalam beam.

Meskipun jumlah simpul yang disimpan terbatas, pada setiap level kedalaman, beam search tetap mengeksplorasi semua b simpul dari setiap simpul yang ada di beam. Kompleksitas waktu untuk setiap level kedalaman adalah O($b * beamWidth$) untuk menghasilkan b gerakan dari setiap simpul dalam beam, kemudian memilih beamWidth simpul terbaik. Hal ini dilakukan pada setiap tingkat kedalaman, dengan kedalaman pencarian yang maksimum adalah m. Dengan demikian, kompleksitas waktu total beam search adalah:O($b^m * beamWidth$).

Di setiap tingkat kedalaman, hanya beamWidth simpul yang disimpan. Beam search juga menggunakan struktur data untuk mencatat simpul yang telah dikunjungi, yang dapat menyimpan b^m simpul dalam kasus terburuk. Ini memastikan bahwa simpul yang sama tidak diproses dua kali. Kompleksitas ruang untuk beam search, jika kita juga mempertimbangkan visited set, adalah: O($beamWidth + b^m$).

5. Analisis

Berdasarkan analisis kompleksitas algoritma yang dilakukan, mayoritas dari algoritma yang digunakan memiliki kompleksitas waktu yang sama dalam notasi Big-O. Namun, hasil percobaan menunjukkan perbedaan yang cukup signifikan. Algoritma UCS selalu menghasilkan solusi yang optimal, namun waktu yang digunakan dan jumlah simpul yang dieksplorasi adalah yang paling banyak. Ini terjadi karena algoritma UCS sama sekali tidak memanfaatkan heuristik sama sekali. Akibatnya, tidak ada yang membimbing algoritma ke jalur yang lebih tepat sehingga semua simpul akan dieksplorasi.

Algoritma yang paling cepat terbukti adalah algoritma A*. Algoritma A* memanfaatkan g(n) dan h(n). Alhasil, hasilnya lebih terarah daripada UCS dan hasilnya tidak sebergantung pada kualitas heuristiknya dibandingkan dengan GBFS. Algoritma GBFS juga cukup cepat namun sangat bergantung pada heuristiknya. Karena tidak mempertimbangkan biaya sejauh ini, maka GBFS cenderung melalui jalur yang suboptimal sehingga menjadi lambat.

Algoritma dengan jumlah mengunjungi paling sedikit simpul adalah algoritma Beam Search. Hal ini memang wajar karena algoritma ini memang membatasi jumlah simpul yang akan diproses pada level selanjutnya. Sama seperti GBFS, algoritma ini sangat bergantung pada fungsi heuristik dan juga sangat bergantung pada nilai lebar *beam*-nya. Nilai ini menentukan berapa banyak simpul yang akan diproses pada iterasi selanjutnya. Nilai lebar *beam* yang semakin tinggi bukan berarti solusi akan lebih optimal, justru nilai yang tinggi dapat juga membawa ke keadaan yang suboptimal.

BAB VI:
KESIMPULAN, SARAN & REFLEKSI

6.1. Kesimpulan

Berdasarkan hasil implementasi dan analisis terhadap algoritma-algoritma pencarian jalur dalam konteks permainan Rush Hour, dapat ditarik beberapa kesimpulan sebagai berikut:

1. Perbandingan Algoritma

   Uniform Cost Search (UCS) selalu menghasilkan solusi optimal, namun membutuhkan waktu komputasi dan ruang memori yang lebih besar dibandingkan algoritma lainnya karena mengeksplorasi seluruh ruang pencarian tanpa panduan heuristik. Greedy Best First Search (GBFS) umumnya memiliki waktu eksekusi yang lebih cepat dibandingkan UCS, tetapi sangat bergantung pada kualitas fungsi heuristik dan tidak menjamin solusi optimal.

   A* memberikan keseimbangan terbaik antara efisiensi dan optimalitas, menggabungkan kelebihan UCS (jaminan optimalitas) dan GBFS (efisiensi pencarian) sehingga menghasilkan solusi yang optimal dengan waktu komputasi yang lebih efisien. Beam Search memiliki penggunaan memori yang paling efisien karena membatasi jumlah simpul yang dieksplorasi, namun dengan risiko tidak menemukan solusi optimal atau bahkan gagal menemukan solusi sama sekali jika lebar beam terlalu kecil.

2. Pengaruh Fungsi Heuristik

   Kualitas fungsi heuristik sangat mempengaruhi performa algoritma informed search (GBFS, A*, dan Beam Search). Heuristik yang admissible seperti *distance to exit*, *blocking cars*, *recursive blocking cars*, dan *move needed estimate* mampu menjamin solusi optimal pada algoritma A*. Fungsi heuristik yang baik secara signifikan mengurangi jumlah simpul yang perlu dieksplorasi, menghasilkan waktu komputasi yang lebih cepat.

3. Kompleksitas Algoritma

   Meskipun secara teoritis beberapa algoritma memiliki kompleksitas Big-O yang serupa, dalam praktiknya terdapat perbedaan signifikan dalam jumlah simpul yang dieksplorasi dan waktu eksekusi. Penggunaan struktur data yang tepat seperti priority queue sangat mempengaruhi efisiensi algoritma pencarian. Trade-off antara waktu komputasi, penggunaan memori, dan optimalitas solusi perlu dipertimbangkan dalam pemilihan algoritma yang sesuai untuk kasus penggunaan tertentu.

4. Aplikasi pada Rush Hour

   Algoritma A* dengan heuristik yang tepat merupakan pilihan terbaik untuk menyelesaikan permainan Rush Hour, menawarkan keseimbangan antara kecepatan dan jaminan solusi optimal. Untuk kasus dengan ruang pencarian yang sangat besar, Beam Search dapat menjadi alternatif yang layak jika optimalitas bukan prioritas utama.

6.2. Saran

Berdasarkan hasil penelitian ini, beberapa saran untuk pengembangan dan penelitian lanjutan adalah:

1. Pengembangan Heuristik
    a. Mengembangkan fungsi heuristik yang lebih spesifik dan efektif untuk permainan Rush Hour, misalnya dengan mempertimbangkan pola konfigurasi khusus yang diketahui sulit diselesaikan.
    b. Menggabungkan beberapa fungsi heuristik dengan pembobotan yang adaptif berdasarkan karakteristik puzzle.
2. Optimasi Algoritma
    a. Mengimplementasikan teknik optimasi seperti memoization atau iterative deepening untuk meningkatkan efisiensi algoritma A* dan UCS.
    b. Mengeksplorasi teknik pruning yang lebih canggih untuk Beam Search guna meningkatkan kemungkinan menemukan solusi optimal.
3. Penggunaan Komputasi Paralel
    a. Memanfaatkan teknik komputasi paralel untuk meningkatkan kecepatan algoritma, terutama untuk UCS dan A* yang memerlukan eksplorasi banyak simpul.
    b. Mengimplementasikan versi terdistribusi dari algoritma pencarian untuk mengatasi keterbatasan memori dan waktu pada puzzle kompleks.

6.3. Refleksi

Proses implementasi dan analisis algoritma pencarian jalur untuk permainan Rush Hour ini memberikan beberapa wawasan penting:

1. Peran Heuristik

    Desain heuristik yang baik membutuhkan pemahaman mendalam tentang domain masalah. Trade-off antara *admissibility* dan kecepatan perlu dipertimbangkan dengan seksama dalam pemilihan atau pengembangan heuristik.

2. Kesulitan dalam Analisis Performa

    Analisis teoritis (Big-O) tidak selalu mencerminkan performa algoritma dalam praktik. Faktor-faktor seperti *overhead* komputasi, pemanfaatan cache, dan karakteristik spesifik masalah mempengaruhi performa aktual.

Eksperimen dengan berbagai algoritma dan heuristik membantu mengembangkan intuisi tentang kelebihan dan kekurangan masing-masing pendekatan. Pada akhirnya, pemilihan algoritma yang tepat untuk menyelesaikan permainan Rush Hour bergantung pada kebutuhan spesifik: jika optimalitas solusi adalah prioritas utama, A* dengan heuristik *admissible* adalah pilihan terbaik, jika kecepatan lebih penting dan optimalitas bukan prioritas utama, GBFS atau Beam Search dapat menjadi alternatif yang baik.

LAMPIRAN

Daftar Pustaka

Maulidevi, Nur Ulfa 2025. "Penentuan Rute (Route/Path Planning) Bagian 1: BFS, DFS, UCS, Greedy Best First Search" (https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf), diakses pada 18 Mei 2025.

Munir, Rinaldi dan Maulidevi, N. U. 2025. "Penentuan Rute (Route/Path Planning) Bagian 2: Algoritma A*" (https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf, diakses pada 18 Mei 2025.

GeeksforGeeks. 2025. "Introduction to Beam Search Algorithm" (https://www.geeksforgeeks.org/introduction-to-beam-search-algorithm/) , diakses pada 20 Mei 2025.

Tautan Repository

Tautan dari *repository* dari Tugas Kecil 3 IF2211 Strategi Algoritma oleh Dave Daniel Yanni (13523003) dan Daniel Pedrosa Wu (13523099) adalah sebagai berikut:

https://github.com/d2v6/Tucil3_13523003_13523099

Tabel Pengerjaan

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	☑	☐