Tugas Kecil 3 IF2211 Strategi Algoritma

Semester II tahun 2024/2025

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding

Muhammad Ra'if Alkautsar	13523011
Muhammad Aditya Rahmadeni	13523028

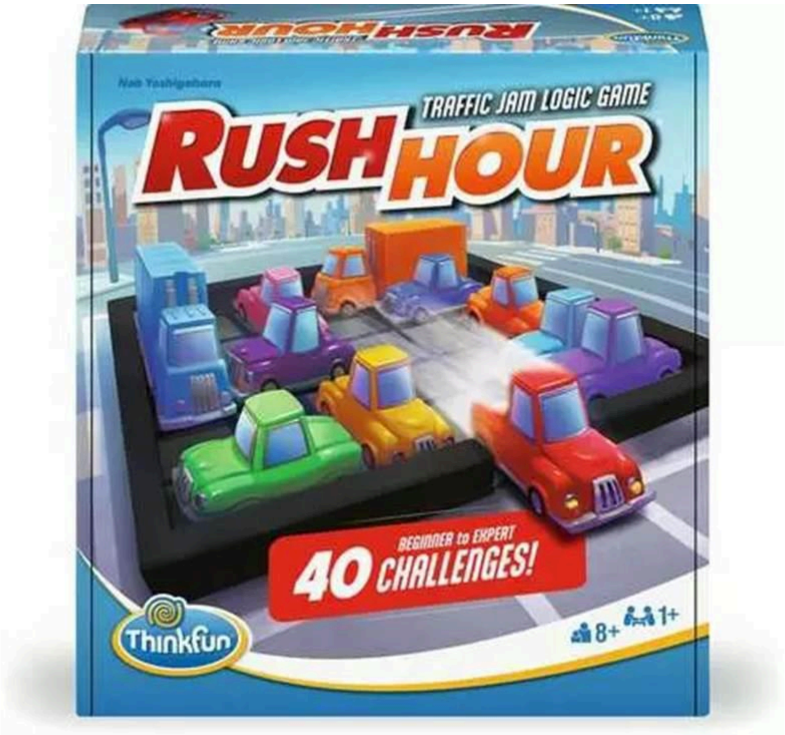
PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2025

BAB 1
DESKRIPSI PERMASALAHAN



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:

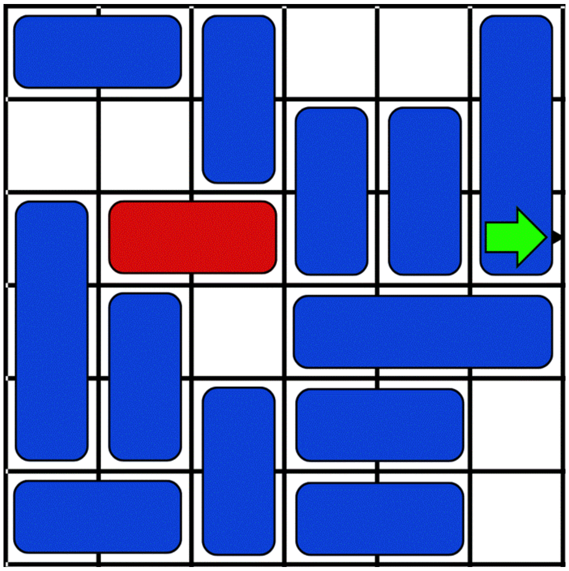
1. **Papan** – *Papan* merupakan tempat permainan dimainkan.
   Papan terdiri atas *cell*, yaitu sebuah *singular point* dari papan. Sebuah *piece* akan menempati *cell-cell* pada papan. Ketika permainan dimulai, semua *piece* telah diletakkan di dalam papan dengan konfigurasi tertentu berupa lokasi 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.

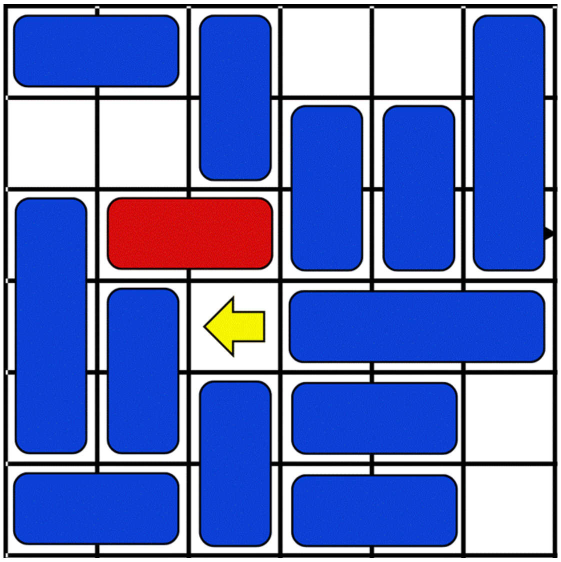
Ilustrasi kasus :

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.



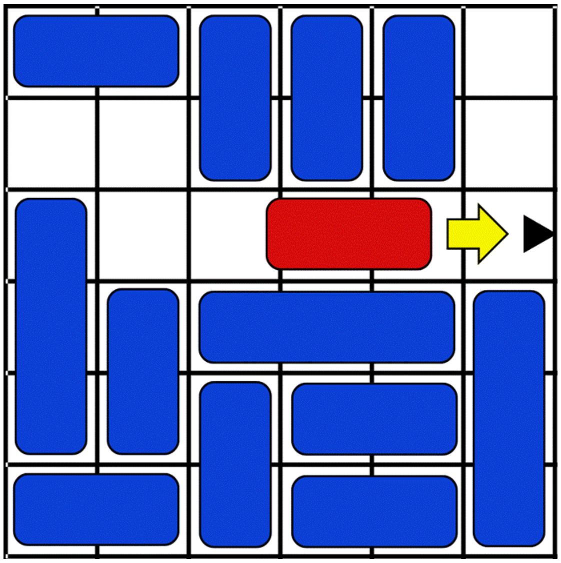
Gambar 2. Awal Permainan Game Rush Hour

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

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



Gambar 4. Pemain Menyelesaikan Permainan

BAB II

ALGORITMA PATHFINDING DAN HEURISTIK BIAYA

A. Uniform Cost Search

Uniform Cost Search Atau UCS adalah algoritma pencari jalur atau pathfinding dengan memprioritaskan simpul dengan biaya terkecil terlebih dahulu. Algoritma ini menggunakan antrian prioritas dengan biaya terkecil terlebih dahulu dalam memproses pencarian jalur hingga mencapai simpul tujuannya.

Pada program ini, graf atau pohon yang ditelusuri merupakan *unweighted* atau tidak memiliki biaya untuk menjelajahi simpul sehingga bisa diasumsikan bahwa setiap simpul memiliki biaya satu. Maka, implementasi dari algoritma ini sama dengan algoritma *Breadth First Search* (BFS)

Berikut adalah langkah-langkah dari cara kerja algoritma UCS

1. Inisialisasi antrian dan senarai simpul yang sudah dikunjungi dengan simpul akar sebagai simpul awal.

UCS.java

```
 public State find(State initialState) {
        queue.add(initialState);
                                  costMap.put(getStateHash(initialState),
initialState.cost);
        visitedNodeCount = 0;
...
```

2. Lakukan pengulangan untuk semua simpul di dalam antrian hingga antrian kosong.

UCS.java

```
while (!queue.isEmpty()) {
      ...
}
```

3. Untuk setiap pengulangan, cek apakah simpul sudah sama dengan kondisi akhir atau kondisi tujuan. Jika benar, kembalikan kondisi tersebut dan jalurnya.

UCS.java

```
if (currentState.isReached(width, height, exitDirection)) {
        return currentState;
}
```

4. Jika salah, maka cek semua simpul tetangga dari simpul sekarang. Jika belum pernah dikunjungi maka masukkan ke dalam antrian.

UCS.java

```
List<State>  successors  =  currentState.generateNextStates(width,
height);

for (State successor : successors) {

      String successorHash = getStateHash(successor);

       if (!costMap.containsKey(successorHash) || successor.cost <
costMap.get(successorHash)) {

           costMap.put(successorHash, successor.cost);

           queue.add(successor);

       }
```

B. A* Search

Pencarian A* adalah algoritma *pathfinding* yang menggabungkan pendekatan *Uniform Cost Search*(UCS) dan strategi *greedy*. Algoritma ini memprioritaskan simpul berdasarkan jumlah dua nilai $f(n)$, yaitu biaya dari simpul akar ke simpul sekarang $g(n)$

nilai estimasi dari simpul saat ini hingga simpul tujuan menggunakan fungsi heuristik $h(n)$.

Dengan menggabungkan kedua algoritma tersebut, algoritma A* tidak hanya memeprtimbangkan biaya minimum, tetapi juga jarak ke tujuan sehingga algoritma ini lebih efisien dalam menemukan solusi optimal atau terpendek.

Berikut adalah langkah-langkah dari cara kerja algoritma pencarian A*:

1. Inisialisasi antrian prioritas dan peta biaya dengan simpul awal

   Pada awal proses, algoritma akan menambahkan simpul awal ke dalam antrian prioritas. Simpul diprioritaskan berdasarkan nilai $f(n) = g(n) + h(n)$. Peta biaya (costMap) menyimpan biaya terendah untuk mencapai suatu konfigurasi simpul.

   AStar.java

   ```
   this.queue = new PriorityQueue<>((s1, s2) -> {

               int f1 = s1.cost + calculateHeuristic(s1);

               int f2 = s2.cost + calculateHeuristic(s2);

               return Integer.compare(f1, f2);

           })
   ```

   AStar.java

   ```
   public State find(State initialState) {

           queue.add(initialState);

                             costMap.put(getStateHash(initialState),
   initialState.cost);
   ```

```
        visitedNodeCount = 0;
```

2. Ulangi proses hingga antrian kosong dengan ambil simpul dengan nilai $f(n)$ terkecil untuk diproses.

AStar.java

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

       State currentState = queue.poll();

       visitedNodeCount++

        ...

}
```

3. Cek apakah simpul yang sedang diproses adalah tujuan. Jika kondisi simpul yang sedang diproses sesuai dengan kriteria akhir (mencapai posisi keluar dengan arah tertentu), maka simpul tersebut dikembalikan sebagai hasil.

AStar.java

```
if (currentState.isReached(width, height, exitDirection)) {

                return currentState;

            }
```

4. Proses semua simpul tetangga dari simpul sekarang dengan memasukkan simpul tetangga ke antrian,

AStar.java

```
List<State>  successors  =  currentState.generateNextStates(width,
height);

            for (State successor : successors) {

                String successorHash = getStateHash(successor);

                       if  (!costMap.containsKey(successorHash)  ||
successor.cost < costMap.get(successorHash)) {

                    costMap.put(successorHash, successor.cost);

                    queue.add(successor);

                }

            }
```

C. Greedy Best First Search

Algoritma *Greedy Best First Search* adalah algoritma *pathfinding* yang menggunakan perkiraan jarak atau biaya ke tujuan dari sebuah simpul.Algoritma ini hanya mempertimbangkan seberapa dekat sebuah simpul ke tujuan berdasarkan fungsi heuristik. Oleh karena itu, Algoritma ini lebih “serakah” dalam memilih simpul yang terlihat paling dekat ke simpul tujuan meskipun tidak menjamin solusi optimal.

Algoritma ini sangat bergantung pada fungsi heuristik yang memperkirakan jarak dari suatu simpul ke tujuan. Semakin baik fungsi heuristiknya, semakin efisien pencarian jalurnya.

Berikut adalah langkah-langkah dari cara kerja algoritma *Greedy Best First Search*.

1. Inisialisasi antrian prioritas dan peta simpul yang telah dikunjungi

GreedyBFS.java

```
public State find(State initialState) {
        queue.add(initialState);
                        visitedMap.put(getStateHash(initialState),
calculateHeuristic(initialState));
        visitedNodeCount = 0;
...
}
```

2. Pengulangan untuk semua simpul di dalam antrian dan mengambil simpul pertama dalam antrian.

GreedyBFS.java

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

             State currentState = queue.poll();

             visitedNodeCount++;

...

}
```

3. Cek apakah simpul saat ini sudah mencapai simpul tujuan. Jika benar maka kembalikan simpul tersebut dan jalurnya.

GreedyBFS.java

```
if    (currentState.isReached(width,    height,    kRow,    kCol,
exitDirection)) {
```

```
                    return currentState;

                }

}
```

4. Jika tidak, maka cek untuk semua simpul tetangga dari simpul sekarang. Jika belum pernah dikunjungi maka masukkan ke dalam antrian.

GreedyBFS.java

```
List<State>  successors  =  currentState.generateNextStates(width,
height);

for (State successor : successors) {

    String successorHash = getStateHash(successor);

    if (!visitedMap.containsKey(successorHash)) {

                                    visitedMap.put(successorHash,
calculateHeuristic(successor));

             queue.add(successor);

     }

}
```

D. Iterative Deepening A*

*Iterative Deepening A** atau IDA* adalah algoritma pencari jalur yang menggabungkan kelebihan dari algoritma A* dan *Depth First Search*. Algoritma ini menggunakan iteratif dengan nilai ambang batas atau *threshold* berdasarkan perkiraan biaya $f(n) = g(n) + h(n)$ dengan $g(n)$ adalah biaya dari simpul akar ke simpul n dan $h(n)$ adalah perkiraan biaya dari simpul n ke simpul tujuan.

Algoritma ini melakukan pencarian berbasis DFS yang dibatasi oleh nilai f tertentu, dan jika tidak menemukan solusi dalam batas tersebut, ia meningkatkan threshold berdasarkan nilai f terkecil yang melebihi ambang batas sebelumnya.

Berikut adalah langkah-langkah dari cara kerja IDA*:

1. Inisialisasi *threshold*

IDAStar.java

```
int threshold = calculateHeuristic(initialState);
```

2. Lakukan pencarian DFS hingga menyentuh nilai batas

IDAStar.java

```
while (threshold < Integer.MAX_VALUE) {
      int nextThreshold = Integer.MAX_VALUE;
      Set<String> visitedStates = new HashSet<>();
      SearchResult  result  =  search(initialState,  0,  threshold,
visitedStates, nextThreshold);
```

3. Jika simpul tujuan ditemukan, kembalikan simpul dan jalurnya.

IDAStar.java

```
if (result.state != null) {
          return result.state;
 }
```

4. Jika simpul tujuan belum ditemukan, nilai batas ditinggikan dan ulangi pencarian

IDAStar.java

```
threshold = result.nextThreshold;
```

E. Heuristik

Berikut adalah heuristik yang digunakan pada program ini

1. Mobility Score

Heuristik ini menilai keadaan berdasarkan mobilitas kendaraan, yaitu jumlah langkah yang bisa dilakukan oleh semua mobil, terutama mobil utama. Tujuan utamanya adalah mencari keadaan saat mobil lebih leluasa bergerak.Pada heuristik ini, gerakan yang lebih leluasa dianggap lebih mendekati tujuan akhir.

2. **Distance to Exit**

   Heuristik ini menghitung jarak lurus ke arah keluar. Heuristik ini berfokus pada seberapa jauh ujung mobil utama dari ujung papan yang memiliki pintu keluar

3. **Combine**

   Heuristik ini merupakan gabungan dari dua pendekatan, yaitu jarak ke arah keluar dan jumlah mobil yang menghalangi jalur keluar. Tujuannya adalah menghasilkan evaluasi yang lebih realistis dan informatif terhadap keadaan di papan.

4. **BlockingCars**

   Heuristik ini menghitung jumlah mobil unik yang menghalangi jalur mobil utama menuju pintu keluar. Ide utama dari heuristik ini adalah semakin banyak mobil yang menghalangi, maka semakin banyak langkah minimum yang diperlukan untuk menyelesaikan *puzzle*.

5. **Distance Blocking Car**

   Heuristik ini mengkombinasikan heuristik yang menghitung jarak langsung mobil ke pintu keluar dan jumlah minimum langka yang dibutuhkan oleh setiap mobil penghalang untuk membersihkan jalur.

BAB III
ANALISIS ALGORITMA

Pertanyaan 1

Definisi f(n) dan g(n)

Dalam konteks algoritma pencarian jalur seperti A*:

- g(n) adalah biaya sebenarnya dari jalur dari simpul awal hingga simpul n saat ini. Dalam Rush Hour, ini mewakili jumlah gerakan yang telah dilakukan untuk mencapai kondisi papan saat ini.
- f(n) adalah perkiraan total biaya jalur dari simpul awal ke simpul tujuan melalui simpul n. Dihitung sebagai f(n) = g(n) + h(n), di mana h(n) adalah fungsi heuristik yang memperkirakan biaya dari simpul n ke tujuan. Untuk Rush Hour, f(n) mewakili perkiraan total jumlah gerakan yang diperlukan untuk menyelesaikan puzzle jika kita melalui kondisi saat ini.

Pertanyaan 2

Apakah heuristik yang digunakan pada algoritma A* admissible?

Sebuah heuristik disebut admissible jika tidak pernah melebih-lebihkan biaya untuk mencapai tujuan dari simpul manapun.

- Distance: Heuristik yang menghitung jarak langsung dari mobil utama ke pintu keluar. Admissible karena mobil harus bergerak setidaknya sejumlah ini untuk mencapai pintu keluar, menjadikannya batas bawah dari gerakan yang sebenarnya diperlukan.
- BlockingCars: Menghitung jumlah mobil yang menghalangi jalur mobil utama ke pintu keluar. Admissible karena setiap mobil penghalang membutuhkan setidaknya satu gerakan untuk membersihkan jalur, sehingga tidak pernah melebih-lebihkan.
- CombinedHeuristic: Menambahkan jarak ke pintu keluar ditambah dua kali jumlah mobil penghalang. Ini dirancang dengan hati-hati agar tetap admissible - setiap mobil penghalang membutuhkan setidaknya satu gerakan untuk dihapus, dan faktor pembobotan (2) memastikan kita tidak melebih-lebihkan.
- BlockingCarDistance: Menghitung gerakan minimum yang dibutuhkan untuk menghapus mobil penghalang ditambah jarak ke pintu keluar. Ini admissible karena memperhitungkan gerakan minimum yang diperlukan untuk mobil utama dan mobil penghalang.
- MobilityScore: Heuristik ini didasarkan pada seberapa bebas mobil dapat bergerak. Ketika diimplementasikan dengan komponen jarak, tetap admissible karena

memprioritaskan kondisi dengan mobilitas lebih tinggi sambil tetap mempertimbangkan jarak minimum yang diperlukan.

Pertanyaan 3

Apakah UCS sama dengan BFS untuk Rush Hour?

Ya, untuk Rush Hour, UCS secara fungsional setara dengan BFS karena:

1. Semua gerakan memiliki biaya yang sama (1 langkah per gerakan mobil)
2. Grafik tidak berbobot (semua tepi memiliki bobot sama)
3. Implementasi menggunakan antrian prioritas yang diurutkan berdasarkan biaya (jumlah gerakan)

Karena setiap tindakan memiliki biaya seragam sebesar 1, simpul-simpul dikembangkan berdasarkan kedalaman mereka dari kondisi awal, sama seperti BFS. Jalur yang dihasilkan akan identik, dan kedua algoritma akan menemukan jalur optimal (terpendek) ke tujuan.

Pertanyaan 4

Apakah A* lebih efisien dibandingkan dengan UCS pada penyelesaian Rush Hour?

Secara teoritis, ya, A* lebih efisien daripada UCS untuk teka-teki Rush Hour karena:

1. A* menggunakan heuristik untuk mengarahkan pencarian ke arah yang menjanjikan, sementara UCS menjelajah secara seragam ke segala arah.
2. Ruang kondisi dalam Rush Hour sangat besar (banyak konfigurasi yang mungkin), membuat pencarian terinformasi sangat penting untuk efisiensi.
3. Dengan heuristik admissible seperti yang diimplementasikan, A* tetap menjamin solusi optimal sambil menjelajahi simpul yang jauh lebih sedikit daripada UCS.
4. Heuristik dalam kode (terutama BlockingCarDistance dan CombinedHeuristic) memberikan informasi bermakna tentang struktur masalah, memungkinkan A* memprioritaskan kondisi yang lebih mungkin mengarah ke tujuan.

Dalam praktiknya, A* dengan heuristik yang baik dapat mengurangi ruang pencarian dengan beberapa kali lipat dibandingkan dengan UCS, sambil tetap menemukan solusi optimal.

Pertanyaan 5

Secara teoritis, apakah algoritma Greedy Best First Search menjamin solusi optimal untuk penyelesaian Rush Hour?

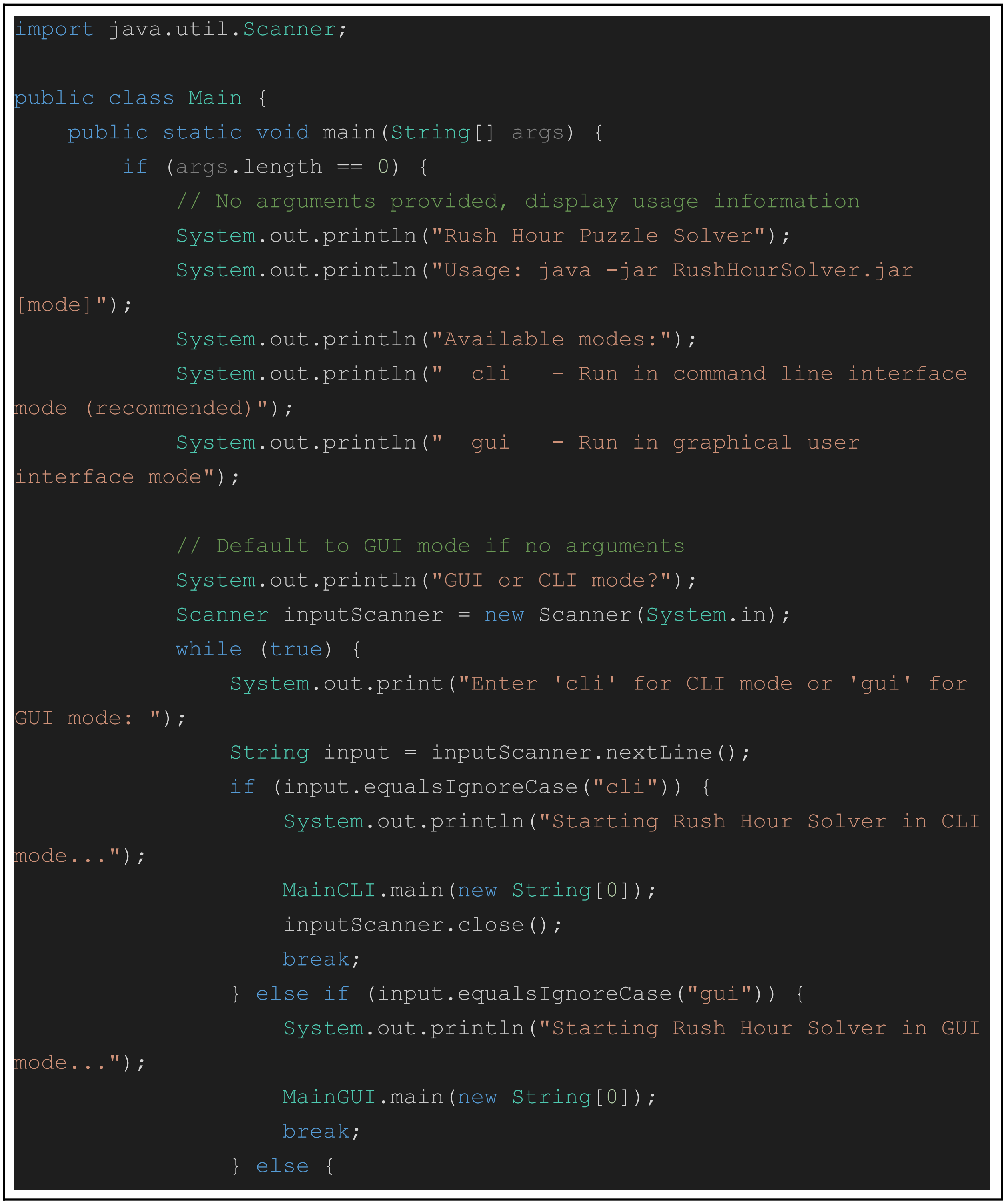
Tidak, Greedy Best First Search tidak menjamin solusi optimal untuk teka-teki Rush Hour. Ini karena:

1. Greedy BFS hanya mempertimbangkan nilai heuristik h(n) saat memilih simpul, sepenuhnya mengabaikan biaya jalur g(n) dari simpul awal.
2. Greedy BFS bisa terjebak mengikuti jalur yang awalnya tampak menjanjikan tetapi mengarah ke solusi suboptimal.
3. Ini bisa berarti menemukan solusi yang membutuhkan lebih banyak gerakan daripada yang diperlukan karena algoritma memprioritaskan kondisi yang tampak lebih dekat ke tujuan menurut heuristik, terlepas dari berapa banyak gerakan yang diperlukan untuk mencapai kondisi tersebut.

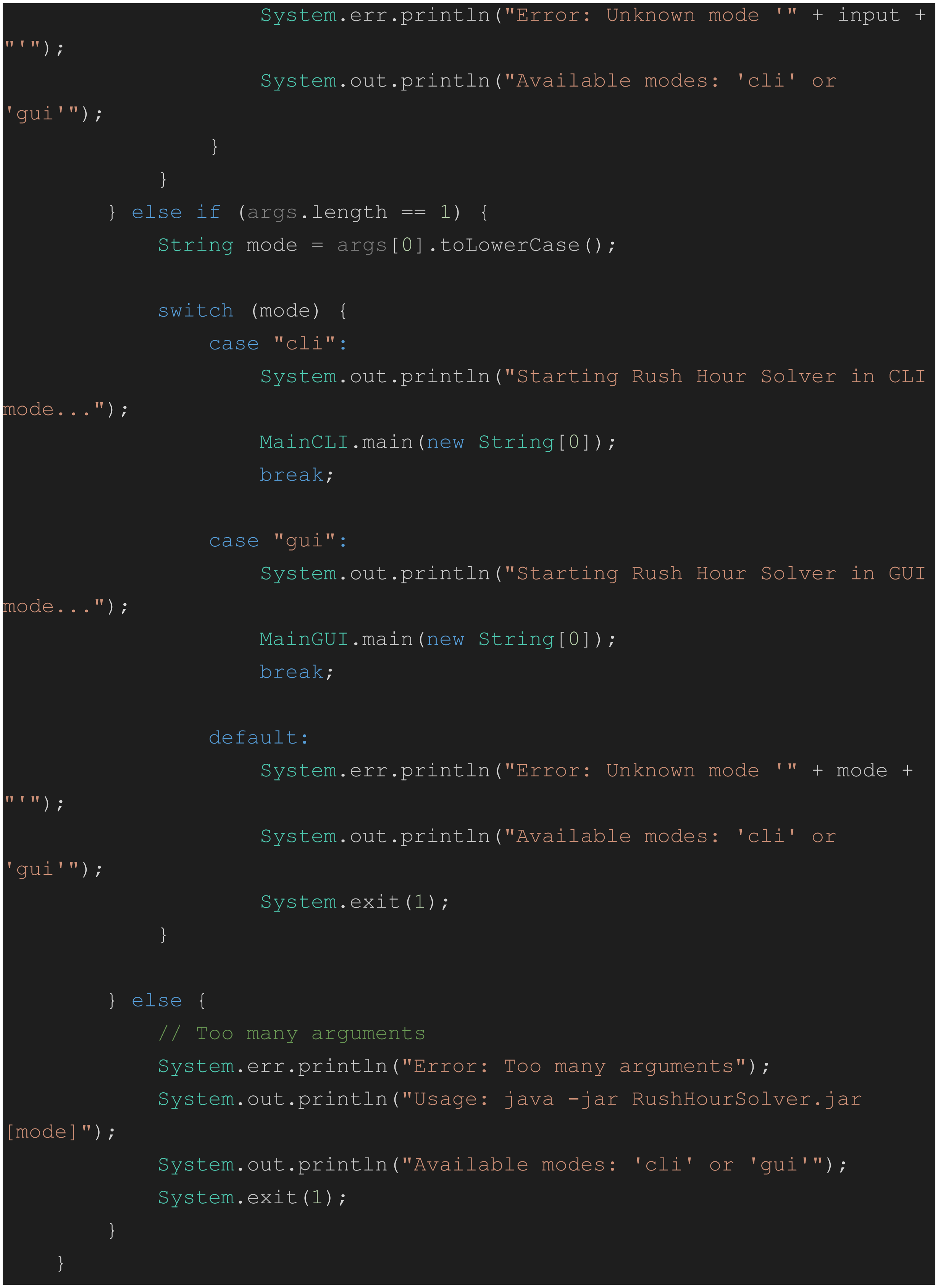
Meskipun sering lebih cepat dari algoritma lain, Greedy BFS mengorbankan optimalitas demi kecepatan, menjadikannya berguna untuk menemukan solusi "cukup baik" dengan cepat tetapi tidak untuk menemukan jalur terpendek.

BAB IV
KODE SUMBER

A. Main



```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        if (args.length == 0) {
            // No arguments provided, display usage information
            System.out.println("Rush Hour Puzzle Solver");
            System.out.println("Usage: java -jar RushHourSolver.jar [mode]");
            System.out.println("Available modes:");
            System.out.println("  cli    - Run in command line interface mode (recommended)");
            System.out.println("  gui    - Run in graphical user interface mode");

            // Default to GUI mode if no arguments
            System.out.println("GUI or CLI mode?");
            Scanner inputScanner = new Scanner(System.in);
            while (true) {
                System.out.print("Enter 'cli' for CLI mode or 'gui' for GUI mode: ");
                String input = inputScanner.nextLine();
                if (input.equalsIgnoreCase("cli")) {
                    System.out.println("Starting Rush Hour Solver in CLI mode...");
                    MainCLI.main(new String[0]);
                    inputScanner.close();
                    break;
                } else if (input.equalsIgnoreCase("gui")) {
                    System.out.println("Starting Rush Hour Solver in GUI mode...");
                    MainGUI.main(new String[0]);
                    break;
                } else {
```

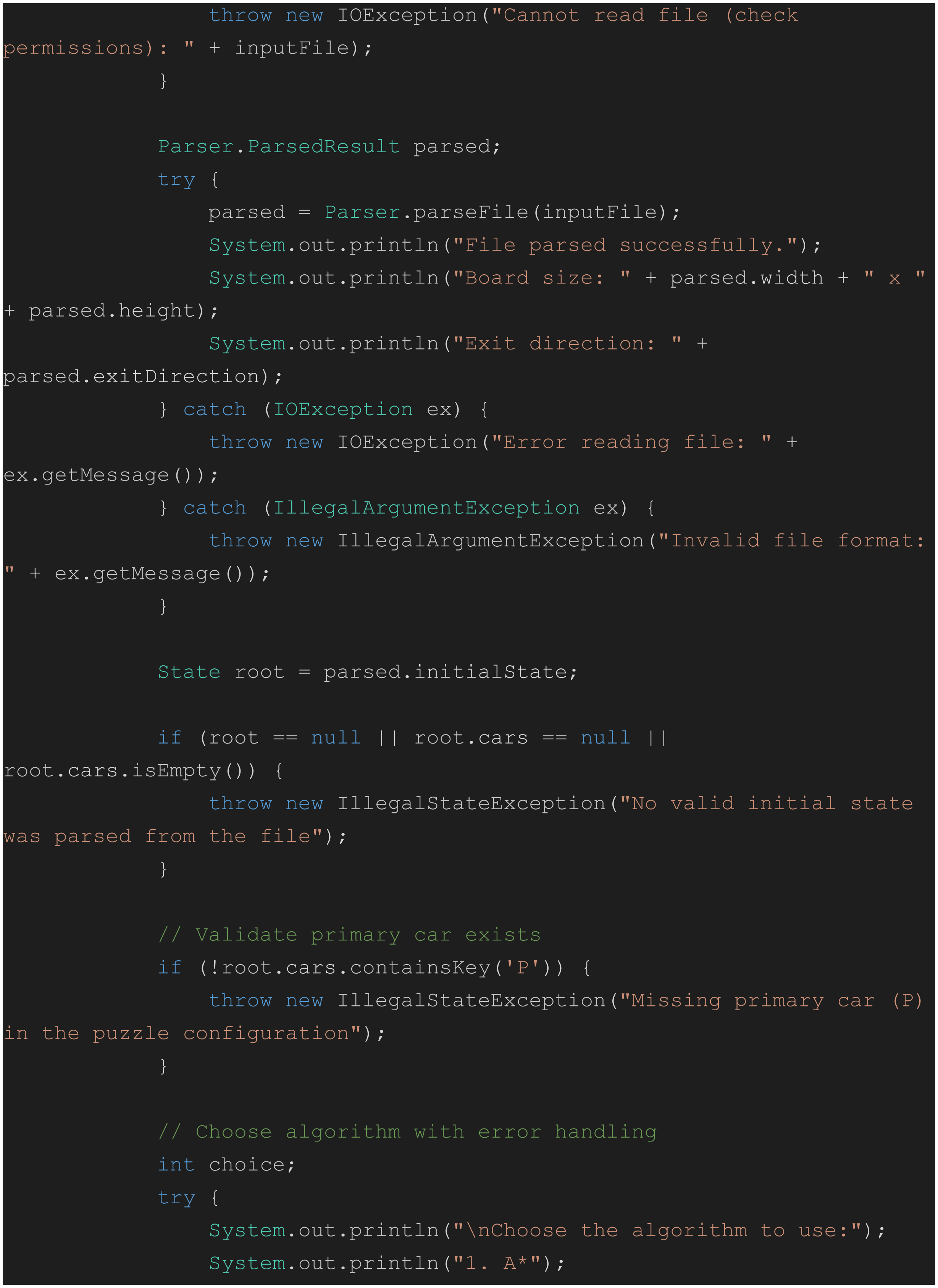


```
                    System.err.println("Error: Unknown mode '" + input +
"'");
                    System.out.println("Available modes: 'cli' or
'gui'");
                }
            }
        } else if (args.length == 1) {
            String mode = args[0].toLowerCase();

            switch (mode) {
                case "cli":
                    System.out.println("Starting Rush Hour Solver in CLI
mode...");
                    MainCLI.main(new String[0]);
                    break;

                case "gui":
                    System.out.println("Starting Rush Hour Solver in GUI
mode...");
                    MainGUI.main(new String[0]);
                    break;

                default:
                    System.err.println("Error: Unknown mode '" + mode +
"'");
                    System.out.println("Available modes: 'cli' or
'gui'");
                    System.exit(1);
            }

        } else {
            // Too many arguments
            System.err.println("Error: Too many arguments");
            System.out.println("Usage: java -jar RushHourSolver.jar
[mode]");
            System.out.println("Available modes: 'cli' or 'gui'");
            System.exit(1);
        }
    }
```

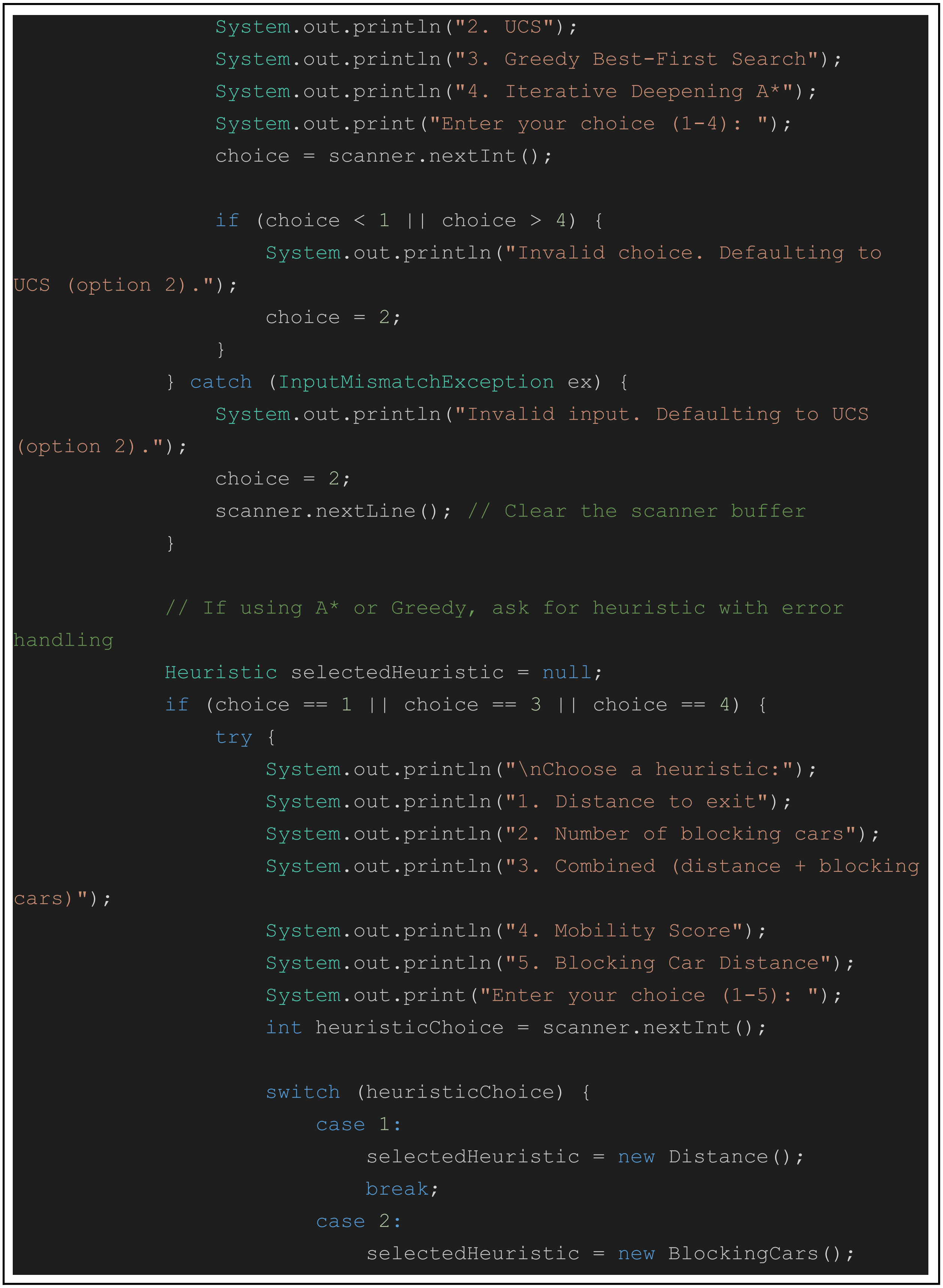
```
}
```

B. MainCLI

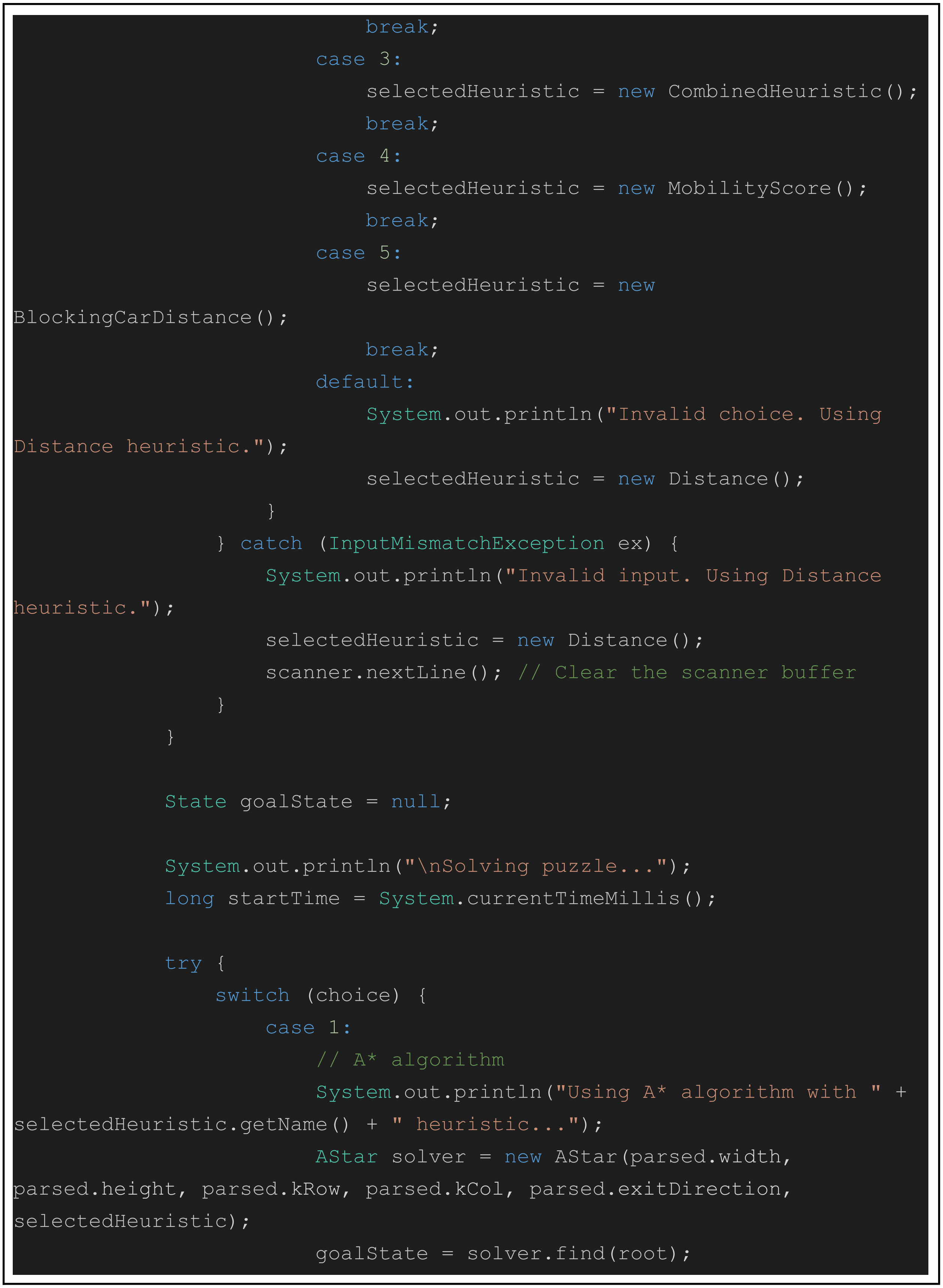
```
import util.Parser;
import util.State;
import pathfinding.*;
import util.Car;
import util.BoardPrinter;
import heuristic.*;
import java.io.IOException;
import java.io.File;
import java.util.Map;
import java.util.Scanner;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
import java.util.InputMismatchException;

public class MainCLI {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {
            System.out.print("Enter the input file path: ");
            String inputFile = scanner.nextLine();

            // Validate file path
            File file = new File(inputFile);
            if (!file.exists()) {
                throw new IOException("File does not exist: " +
inputFile);
            }
            if (!file.isFile()) {
                throw new IOException("Path is not a file: " +
inputFile);
            }
            if (!file.canRead()) {
```



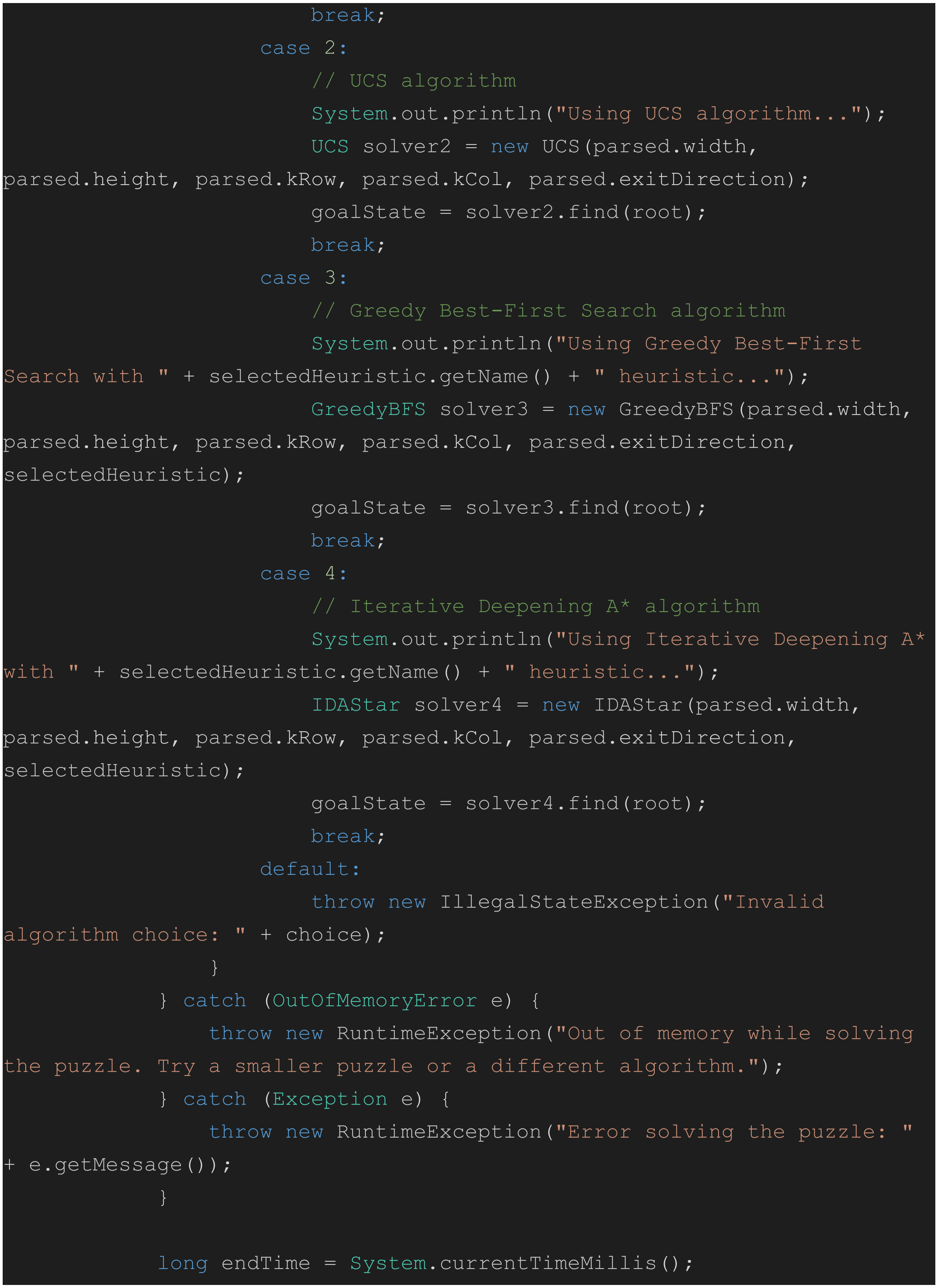
```java
                throw new IOException("Cannot read file (check permissions): " + inputFile);
            }

            Parser.ParsedResult parsed;
            try {
                parsed = Parser.parseFile(inputFile);
                System.out.println("File parsed successfully.");
                System.out.println("Board size: " + parsed.width + " x " + parsed.height);
                System.out.println("Exit direction: " + parsed.exitDirection);
            } catch (IOException ex) {
                throw new IOException("Error reading file: " + ex.getMessage());
            } catch (IllegalArgumentException ex) {
                throw new IllegalArgumentException("Invalid file format: " + ex.getMessage());
            }

            State root = parsed.initialState;

            if (root == null || root.cars == null || root.cars.isEmpty()) {
                throw new IllegalStateException("No valid initial state was parsed from the file");
            }

            // Validate primary car exists
            if (!root.cars.containsKey('P')) {
                throw new IllegalStateException("Missing primary car (P) in the puzzle configuration");
            }

            // Choose algorithm with error handling
            int choice;
            try {
                System.out.println("\nChoose the algorithm to use:");
                System.out.println("1. A*");
```



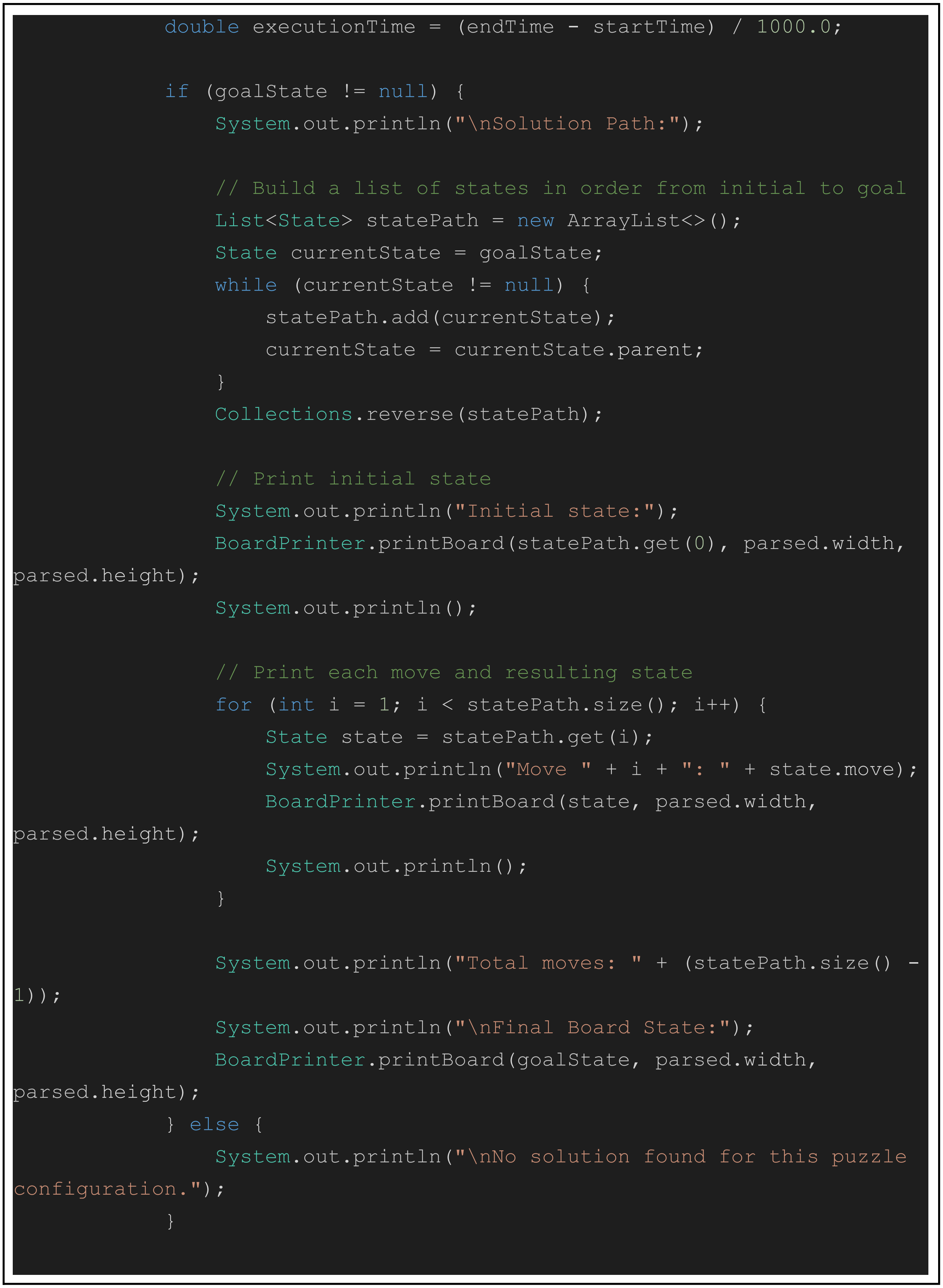
```
                System.out.println("2. UCS");
                System.out.println("3. Greedy Best-First Search");
                System.out.println("4. Iterative Deepening A*");
                System.out.print("Enter your choice (1-4): ");
                choice = scanner.nextInt();

                if (choice < 1 || choice > 4) {
                    System.out.println("Invalid choice. Defaulting to
UCS (option 2).");
                    choice = 2;
                }
            } catch (InputMismatchException ex) {
                System.out.println("Invalid input. Defaulting to UCS
(option 2).");
                choice = 2;
                scanner.nextLine(); // Clear the scanner buffer
            }

            // If using A* or Greedy, ask for heuristic with error
handling
            Heuristic selectedHeuristic = null;
            if (choice == 1 || choice == 3 || choice == 4) {
                try {
                    System.out.println("\nChoose a heuristic:");
                    System.out.println("1. Distance to exit");
                    System.out.println("2. Number of blocking cars");
                    System.out.println("3. Combined (distance + blocking
cars)");
                    System.out.println("4. Mobility Score");
                    System.out.println("5. Blocking Car Distance");
                    System.out.print("Enter your choice (1-5): ");
                    int heuristicChoice = scanner.nextInt();

                    switch (heuristicChoice) {
                        case 1:
                            selectedHeuristic = new Distance();
                            break;
                        case 2:
                            selectedHeuristic = new BlockingCars();
```



```
                            break;
                        case 3:
                            selectedHeuristic = new CombinedHeuristic();
                            break;
                        case 4:
                            selectedHeuristic = new MobilityScore();
                            break;
                        case 5:
                            selectedHeuristic = new
BlockingCarDistance();
                            break;
                        default:
                            System.out.println("Invalid choice. Using
Distance heuristic.");
                            selectedHeuristic = new Distance();
                    }
                } catch (InputMismatchException ex) {
                    System.out.println("Invalid input. Using Distance
heuristic.");
                    selectedHeuristic = new Distance();
                    scanner.nextLine(); // Clear the scanner buffer
                }
            }

            State goalState = null;

            System.out.println("\nSolving puzzle...");
            long startTime = System.currentTimeMillis();

            try {
                switch (choice) {
                    case 1:
                        // A* algorithm
                        System.out.println("Using A* algorithm with " +
selectedHeuristic.getName() + " heuristic...");
                        AStar solver = new AStar(parsed.width,
parsed.height, parsed.kRow, parsed.kCol, parsed.exitDirection,
selectedHeuristic);
                        goalState = solver.find(root);
```



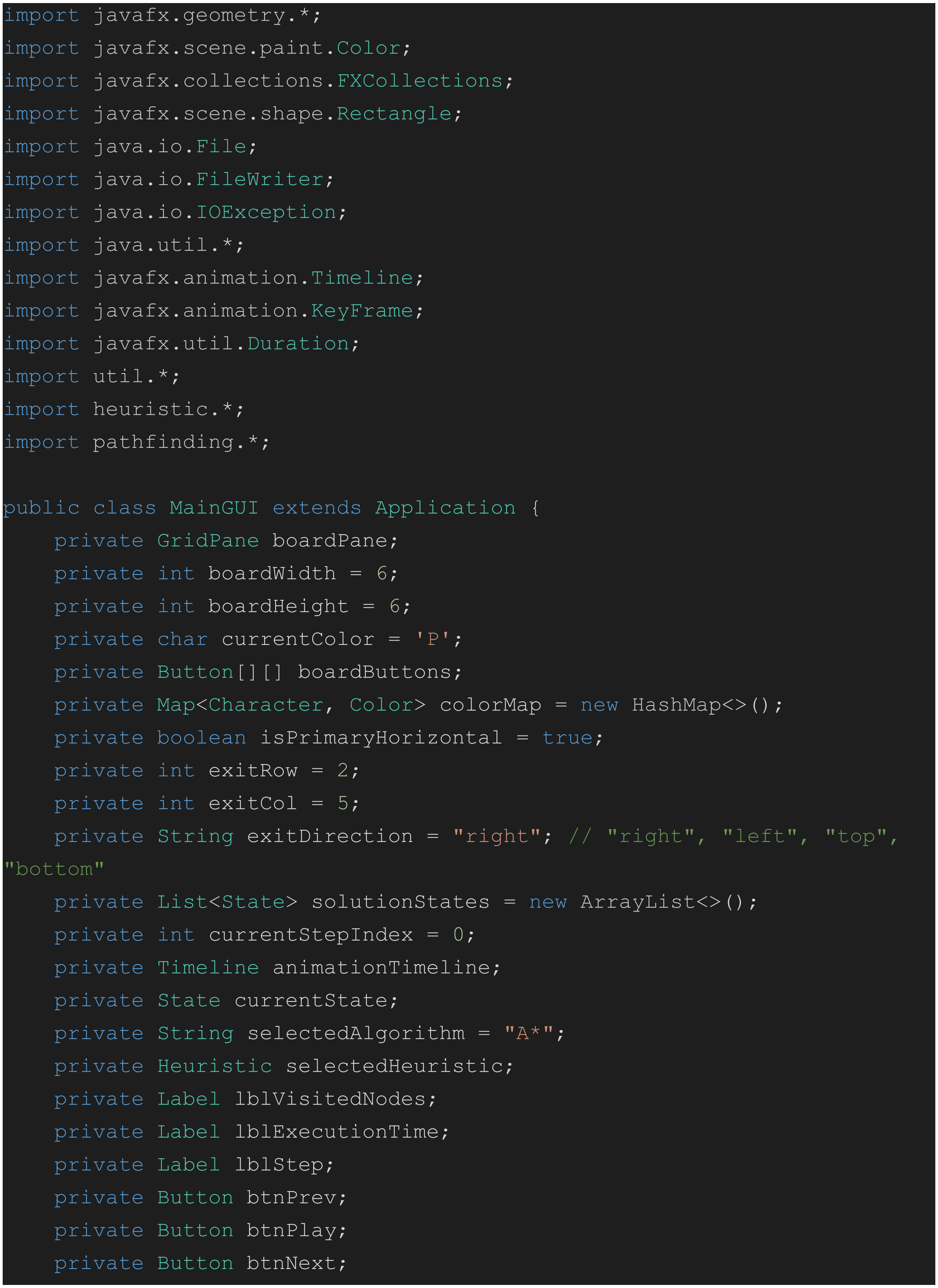
```
                        break;
                    case 2:
                        // UCS algorithm
                        System.out.println("Using UCS algorithm...");
                        UCS solver2 = new UCS(parsed.width,
parsed.height, parsed.kRow, parsed.kCol, parsed.exitDirection);
                        goalState = solver2.find(root);
                        break;
                    case 3:
                        // Greedy Best-First Search algorithm
                        System.out.println("Using Greedy Best-First
Search with " + selectedHeuristic.getName() + " heuristic...");
                        GreedyBFS solver3 = new GreedyBFS(parsed.width,
parsed.height, parsed.kRow, parsed.kCol, parsed.exitDirection,
selectedHeuristic);
                        goalState = solver3.find(root);
                        break;
                    case 4:
                        // Iterative Deepening A* algorithm
                        System.out.println("Using Iterative Deepening A*
with " + selectedHeuristic.getName() + " heuristic...");
                        IDAStar solver4 = new IDAStar(parsed.width,
parsed.height, parsed.kRow, parsed.kCol, parsed.exitDirection,
selectedHeuristic);
                        goalState = solver4.find(root);
                        break;
                    default:
                        throw new IllegalStateException("Invalid
algorithm choice: " + choice);
                }
            } catch (OutOfMemoryError e) {
                throw new RuntimeException("Out of memory while solving
the puzzle. Try a smaller puzzle or a different algorithm.");
            } catch (Exception e) {
                throw new RuntimeException("Error solving the puzzle: "
+ e.getMessage());
            }

            long endTime = System.currentTimeMillis();
```



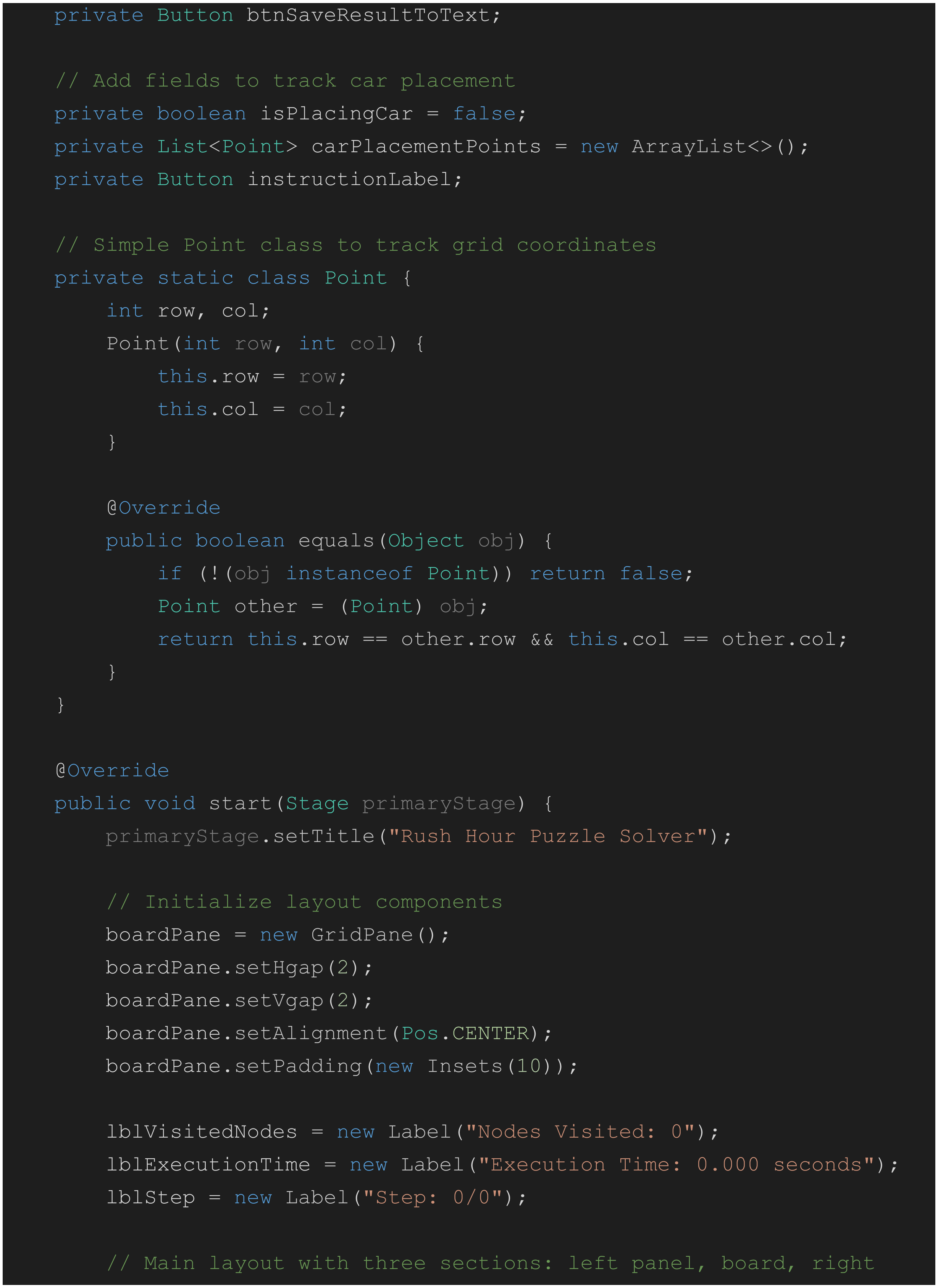
```
            double executionTime = (endTime - startTime) / 1000.0;

            if (goalState != null) {
                System.out.println("\nSolution Path:");

                // Build a list of states in order from initial to goal
                List<State> statePath = new ArrayList<>();
                State currentState = goalState;
                while (currentState != null) {
                    statePath.add(currentState);
                    currentState = currentState.parent;
                }
                Collections.reverse(statePath);

                // Print initial state
                System.out.println("Initial state:");
                BoardPrinter.printBoard(statePath.get(0), parsed.width,
parsed.height);
                System.out.println();

                // Print each move and resulting state
                for (int i = 1; i < statePath.size(); i++) {
                    State state = statePath.get(i);
                    System.out.println("Move " + i + ": " + state.move);
                    BoardPrinter.printBoard(state, parsed.width,
parsed.height);
                    System.out.println();
                }

                System.out.println("Total moves: " + (statePath.size() -
1));
                System.out.println("\nFinal Board State:");
                BoardPrinter.printBoard(goalState, parsed.width,
parsed.height);
            } else {
                System.out.println("\nNo solution found for this puzzle
configuration.");
            }
            System.out.printf("\nExecution time: %.3f seconds\n",
executionTime);

        } catch (IOException e) {
            System.err.println("Error with input file: " +
e.getMessage());
        } catch (IllegalArgumentException e) {
            System.err.println("Invalid input format: " +
e.getMessage());
        } catch (IllegalStateException e) {
            System.err.println("Problem with puzzle state: " +
e.getMessage());
        } catch (RuntimeException e) {
            System.err.println("Runtime error: " + e.getMessage());
        } catch (Exception e) {
            System.err.println("Unexpected error occurred: " +
e.getMessage());
            e.printStackTrace();
        } finally {
            try {
                scanner.close();
            } catch (Exception e) {
                // Ignore errors while closing scanner
            }
            System.out.println("\nProgram terminated.");
        }
    }
}
```

C. MainGUI

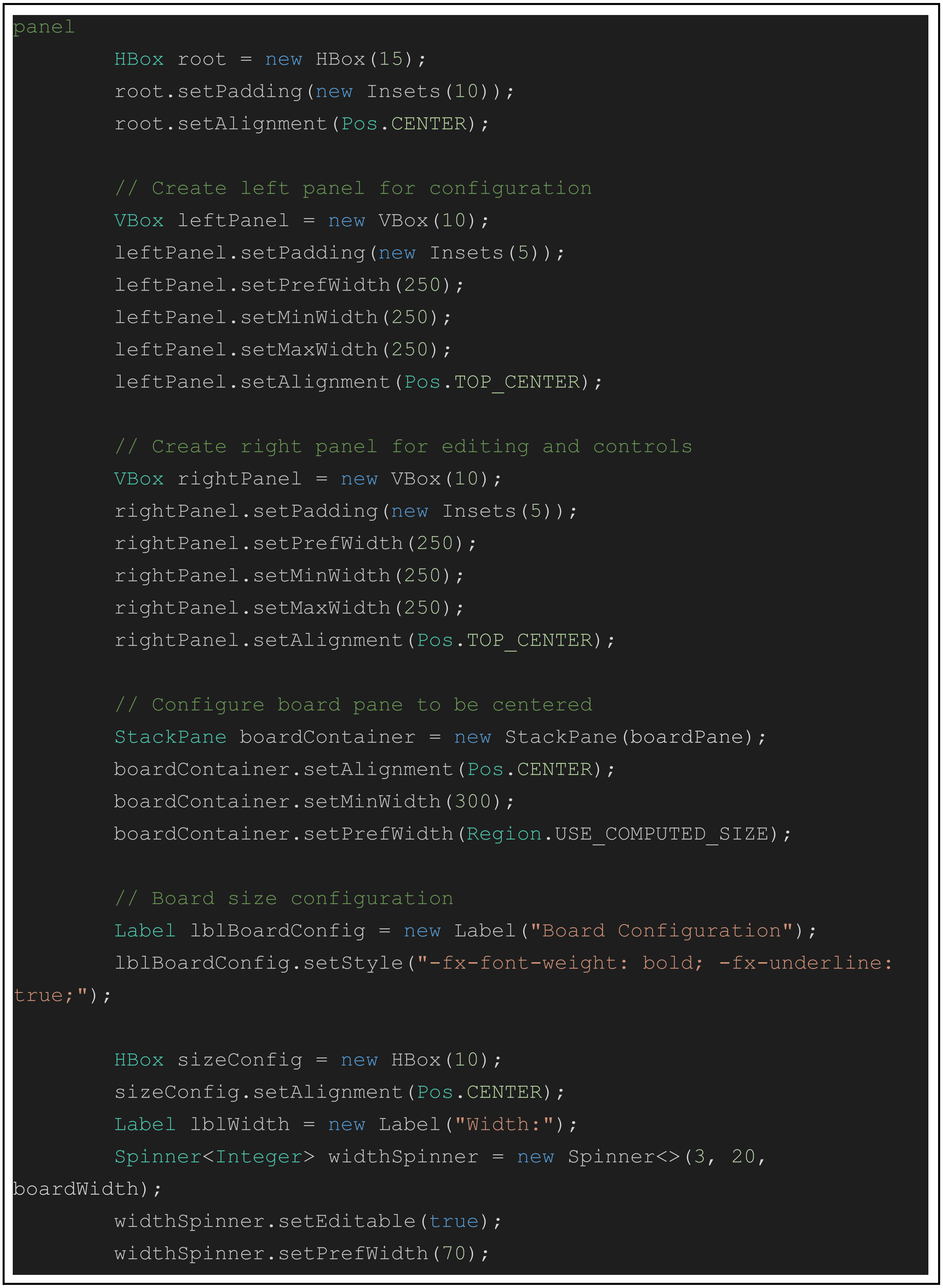
```
import javafx.application.Application;
import javafx.application.Platform;
import javafx.stage.Stage;
import javafx.stage.FileChooser;
import javafx.scene.Scene;
import javafx.scene.layout.*;
import javafx.scene.control.*;
```



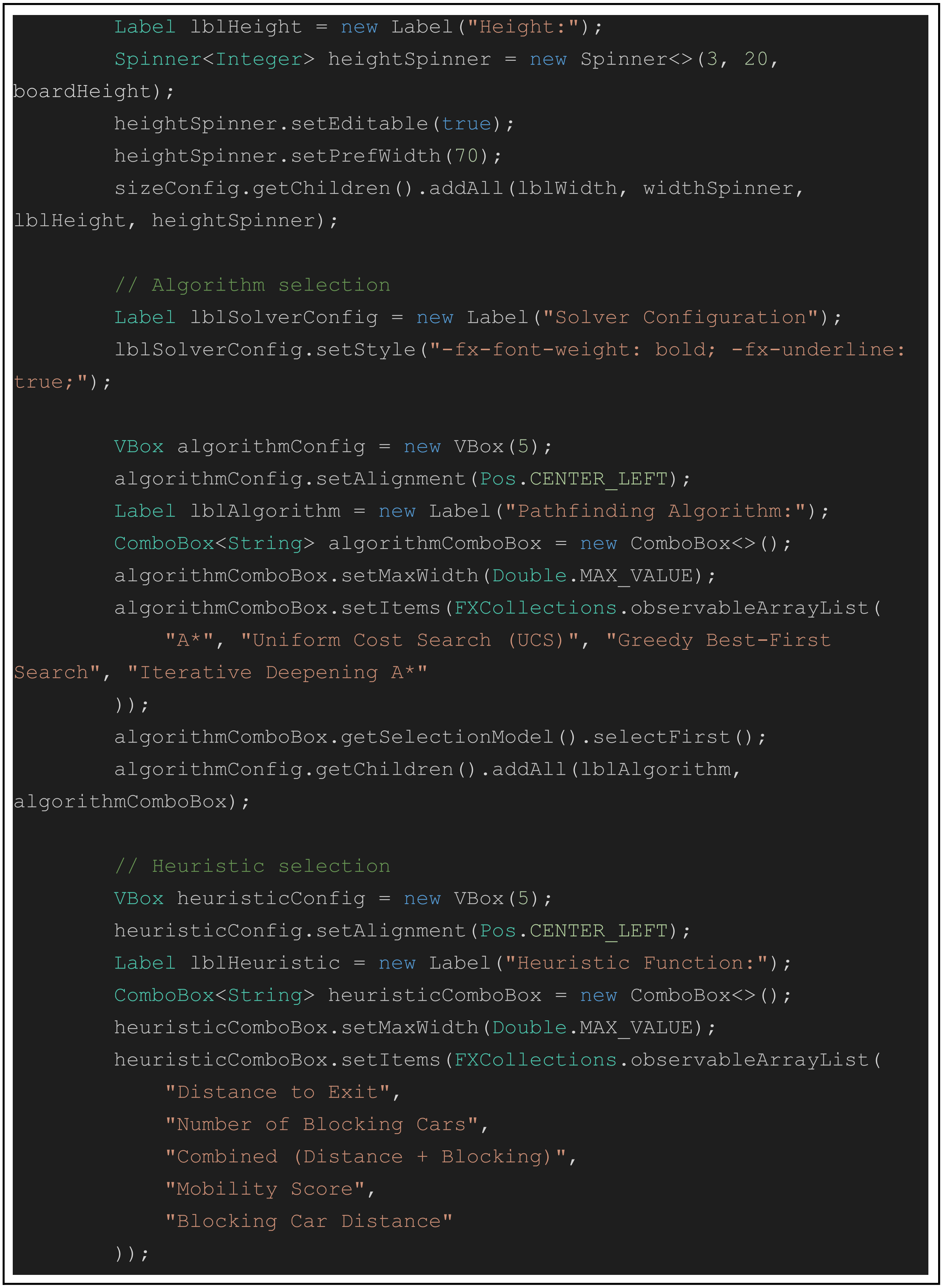
```java
import javafx.geometry.*;
import javafx.scene.paint.Color;
import javafx.collections.FXCollections;
import javafx.scene.shape.Rectangle;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.*;
import javafx.animation.Timeline;
import javafx.animation.KeyFrame;
import javafx.util.Duration;
import util.*;
import heuristic.*;
import pathfinding.*;

public class MainGUI extends Application {
    private GridPane boardPane;
    private int boardWidth = 6;
    private int boardHeight = 6;
    private char currentColor = 'P';
    private Button[][] boardButtons;
    private Map<Character, Color> colorMap = new HashMap<>();
    private boolean isPrimaryHorizontal = true;
    private int exitRow = 2;
    private int exitCol = 5;
    private String exitDirection = "right"; // "right", "left", "top",
"bottom"
    private List<State> solutionStates = new ArrayList<>();
    private int currentStepIndex = 0;
    private Timeline animationTimeline;
    private State currentState;
    private String selectedAlgorithm = "A*";
    private Heuristic selectedHeuristic;
    private Label lblVisitedNodes;
    private Label lblExecutionTime;
    private Label lblStep;
    private Button btnPrev;
    private Button btnPlay;
    private Button btnNext;
```



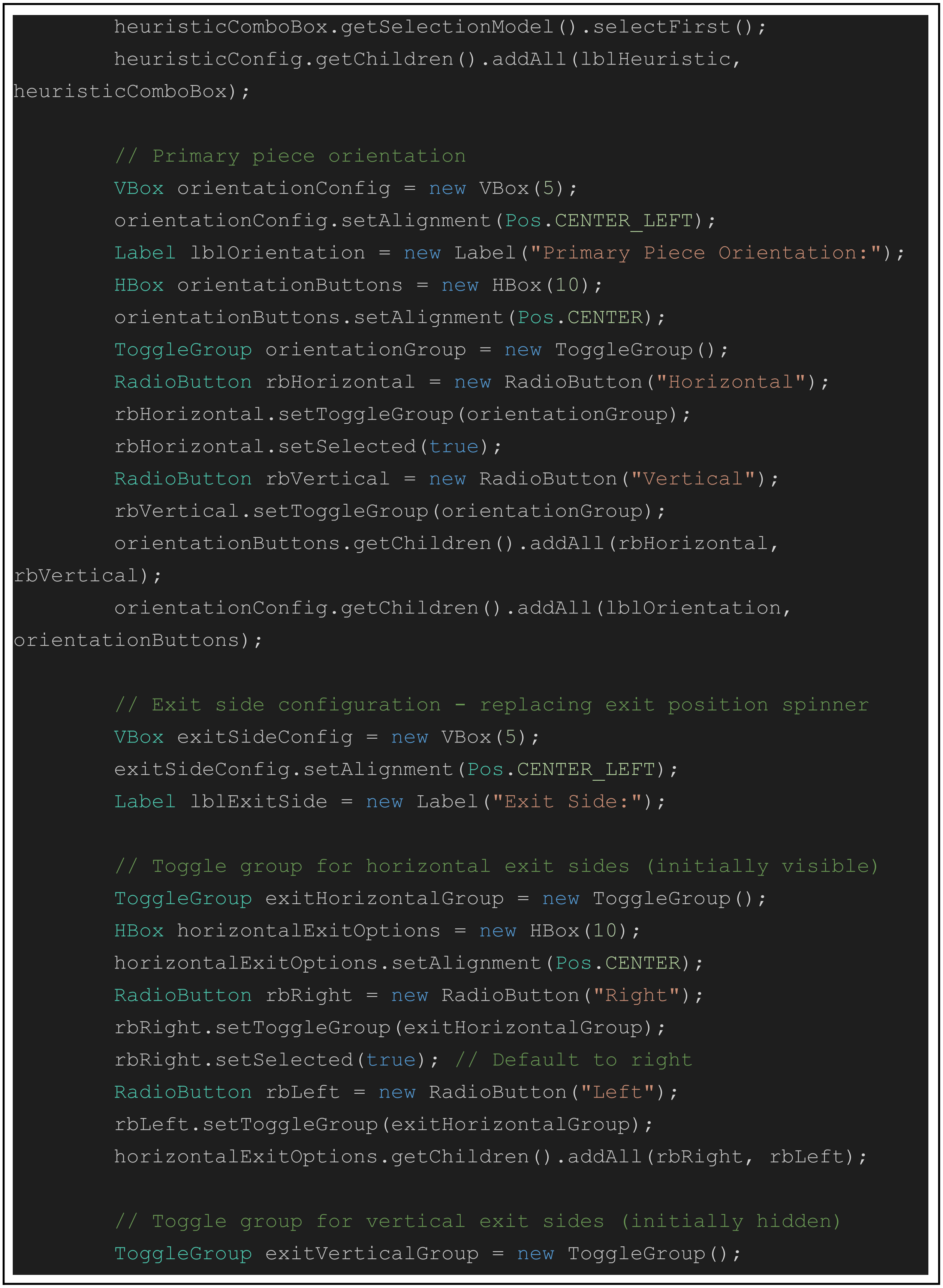
```
    private Button btnSaveResultToText;

    // Add fields to track car placement
    private boolean isPlacingCar = false;
    private List<Point> carPlacementPoints = new ArrayList<>();
    private Button instructionLabel;

    // Simple Point class to track grid coordinates
    private static class Point {
        int row, col;
        Point(int row, int col) {
            this.row = row;
            this.col = col;
        }

        @Override
        public boolean equals(Object obj) {
            if (!(obj instanceof Point)) return false;
            Point other = (Point) obj;
            return this.row == other.row && this.col == other.col;
        }
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Rush Hour Puzzle Solver");

        // Initialize layout components
        boardPane = new GridPane();
        boardPane.setHgap(2);
        boardPane.setVgap(2);
        boardPane.setAlignment(Pos.CENTER);
        boardPane.setPadding(new Insets(10));

        lblVisitedNodes = new Label("Nodes Visited: 0");
        lblExecutionTime = new Label("Execution Time: 0.000 seconds");
        lblStep = new Label("Step: 0/0");

        // Main layout with three sections: left panel, board, right
```



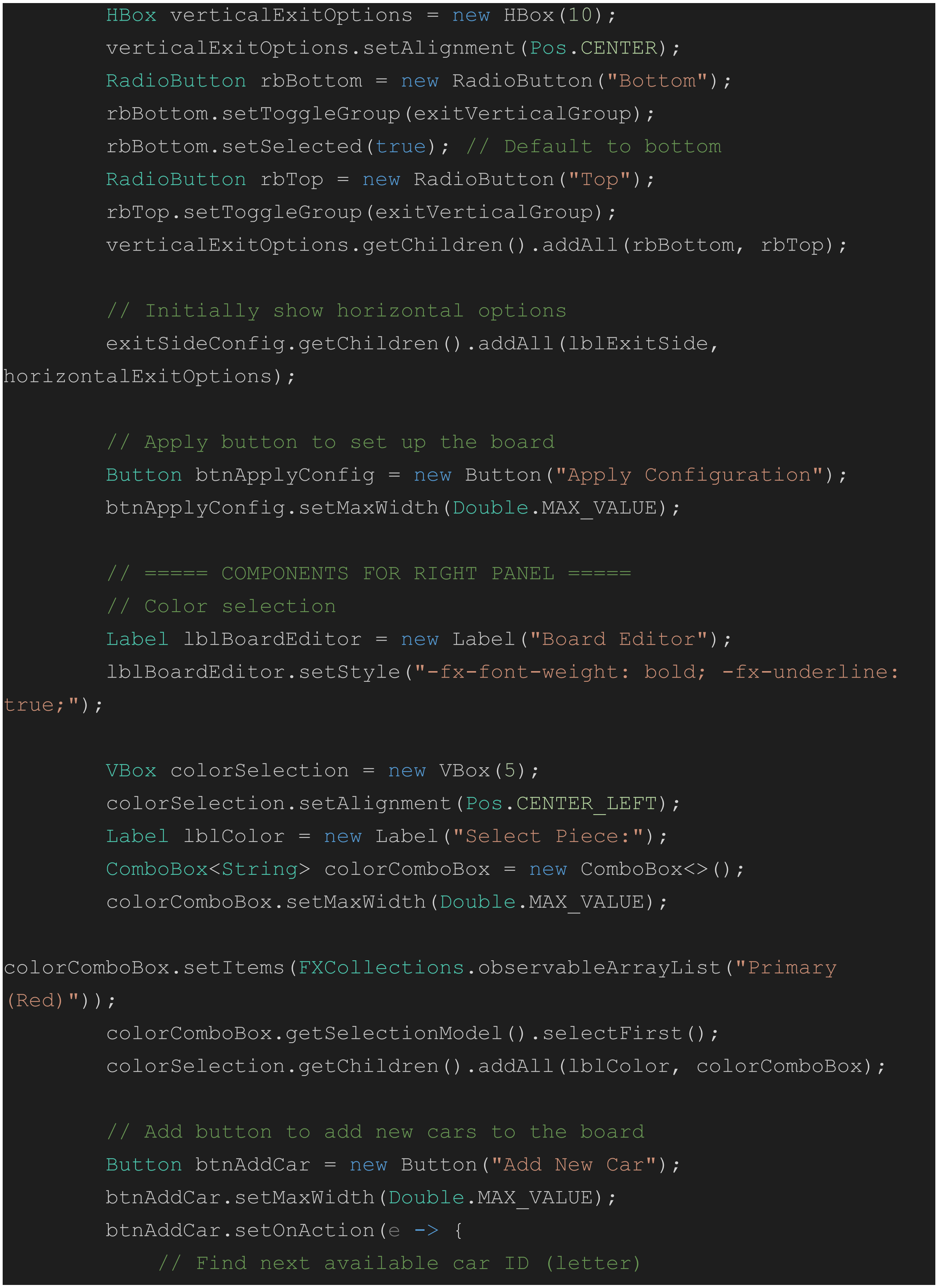
```
panel
        HBox root = new HBox(15);
        root.setPadding(new Insets(10));
        root.setAlignment(Pos.CENTER);

        // Create left panel for configuration
        VBox leftPanel = new VBox(10);
        leftPanel.setPadding(new Insets(5));
        leftPanel.setPrefWidth(250);
        leftPanel.setMinWidth(250);
        leftPanel.setMaxWidth(250);
        leftPanel.setAlignment(Pos.TOP_CENTER);

        // Create right panel for editing and controls
        VBox rightPanel = new VBox(10);
        rightPanel.setPadding(new Insets(5));
        rightPanel.setPrefWidth(250);
        rightPanel.setMinWidth(250);
        rightPanel.setMaxWidth(250);
        rightPanel.setAlignment(Pos.TOP_CENTER);

        // Configure board pane to be centered
        StackPane boardContainer = new StackPane(boardPane);
        boardContainer.setAlignment(Pos.CENTER);
        boardContainer.setMinWidth(300);
        boardContainer.setPrefWidth(Region.USE_COMPUTED_SIZE);

        // Board size configuration
        Label lblBoardConfig = new Label("Board Configuration");
        lblBoardConfig.setStyle("-fx-font-weight: bold; -fx-underline:
true;");

        HBox sizeConfig = new HBox(10);
        sizeConfig.setAlignment(Pos.CENTER);
        Label lblWidth = new Label("Width:");
        Spinner<Integer> widthSpinner = new Spinner<>(3, 20,
boardWidth);
        widthSpinner.setEditable(true);
        widthSpinner.setPrefWidth(70);
```



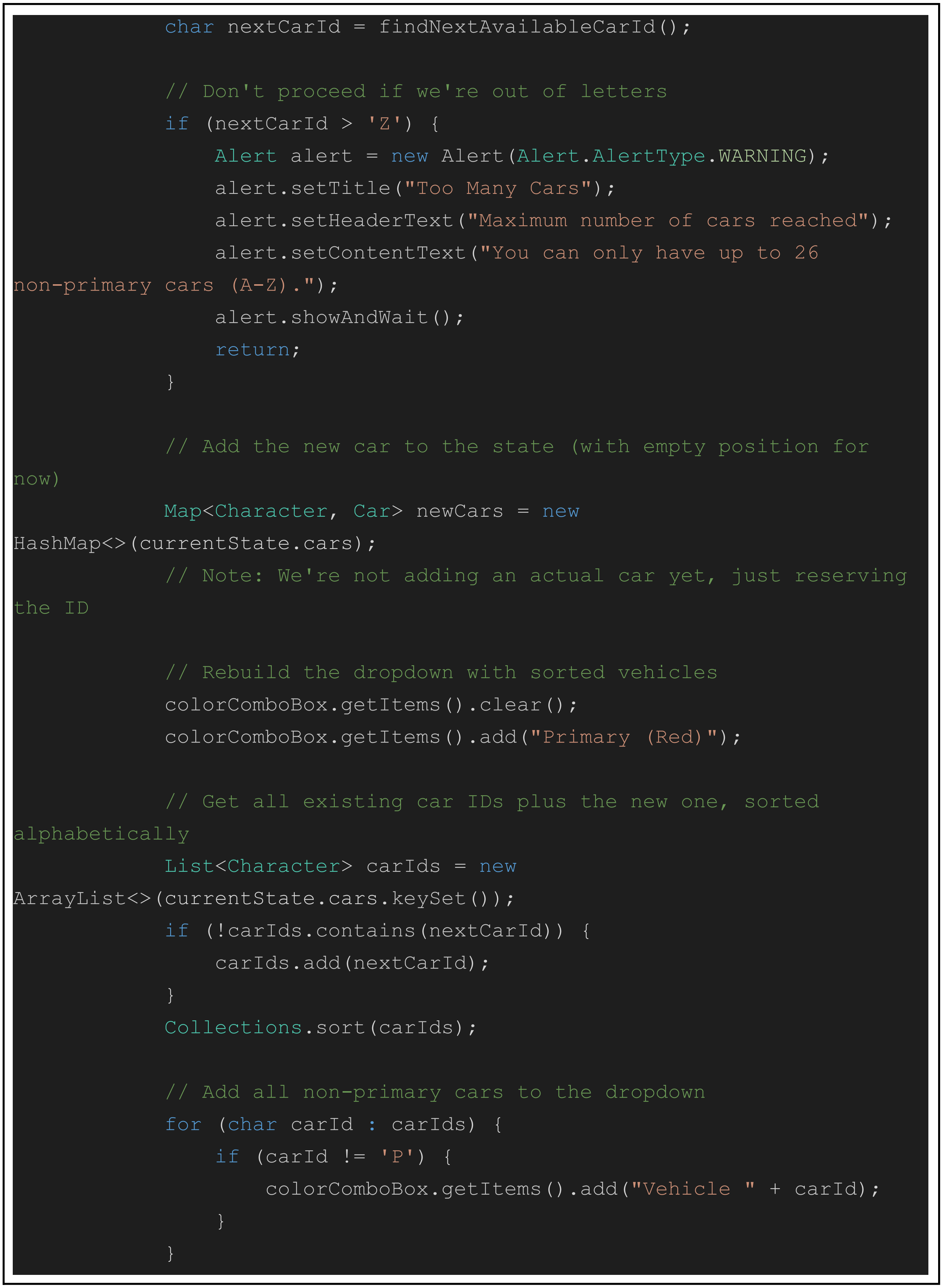
```
        Label lblHeight = new Label("Height:");
        Spinner<Integer> heightSpinner = new Spinner<>(3, 20, boardHeight);
        heightSpinner.setEditable(true);
        heightSpinner.setPrefWidth(70);
        sizeConfig.getChildren().addAll(lblWidth, widthSpinner, lblHeight, heightSpinner);

        // Algorithm selection
        Label lblSolverConfig = new Label("Solver Configuration");
        lblSolverConfig.setStyle("-fx-font-weight: bold; -fx-underline: true;");

        VBox algorithmConfig = new VBox(5);
        algorithmConfig.setAlignment(Pos.CENTER_LEFT);
        Label lblAlgorithm = new Label("Pathfinding Algorithm:");
        ComboBox<String> algorithmComboBox = new ComboBox<>();
        algorithmComboBox.setMaxWidth(Double.MAX_VALUE);
        algorithmComboBox.setItems(FXCollections.observableArrayList(
            "A*", "Uniform Cost Search (UCS)", "Greedy Best-First Search", "Iterative Deepening A*"
        ));
        algorithmComboBox.getSelectionModel().selectFirst();
        algorithmConfig.getChildren().addAll(lblAlgorithm, algorithmComboBox);

        // Heuristic selection
        VBox heuristicConfig = new VBox(5);
        heuristicConfig.setAlignment(Pos.CENTER_LEFT);
        Label lblHeuristic = new Label("Heuristic Function:");
        ComboBox<String> heuristicComboBox = new ComboBox<>();
        heuristicComboBox.setMaxWidth(Double.MAX_VALUE);
        heuristicComboBox.setItems(FXCollections.observableArrayList(
            "Distance to Exit",
            "Number of Blocking Cars",
            "Combined (Distance + Blocking)",
            "Mobility Score",
            "Blocking Car Distance"
        ));
```



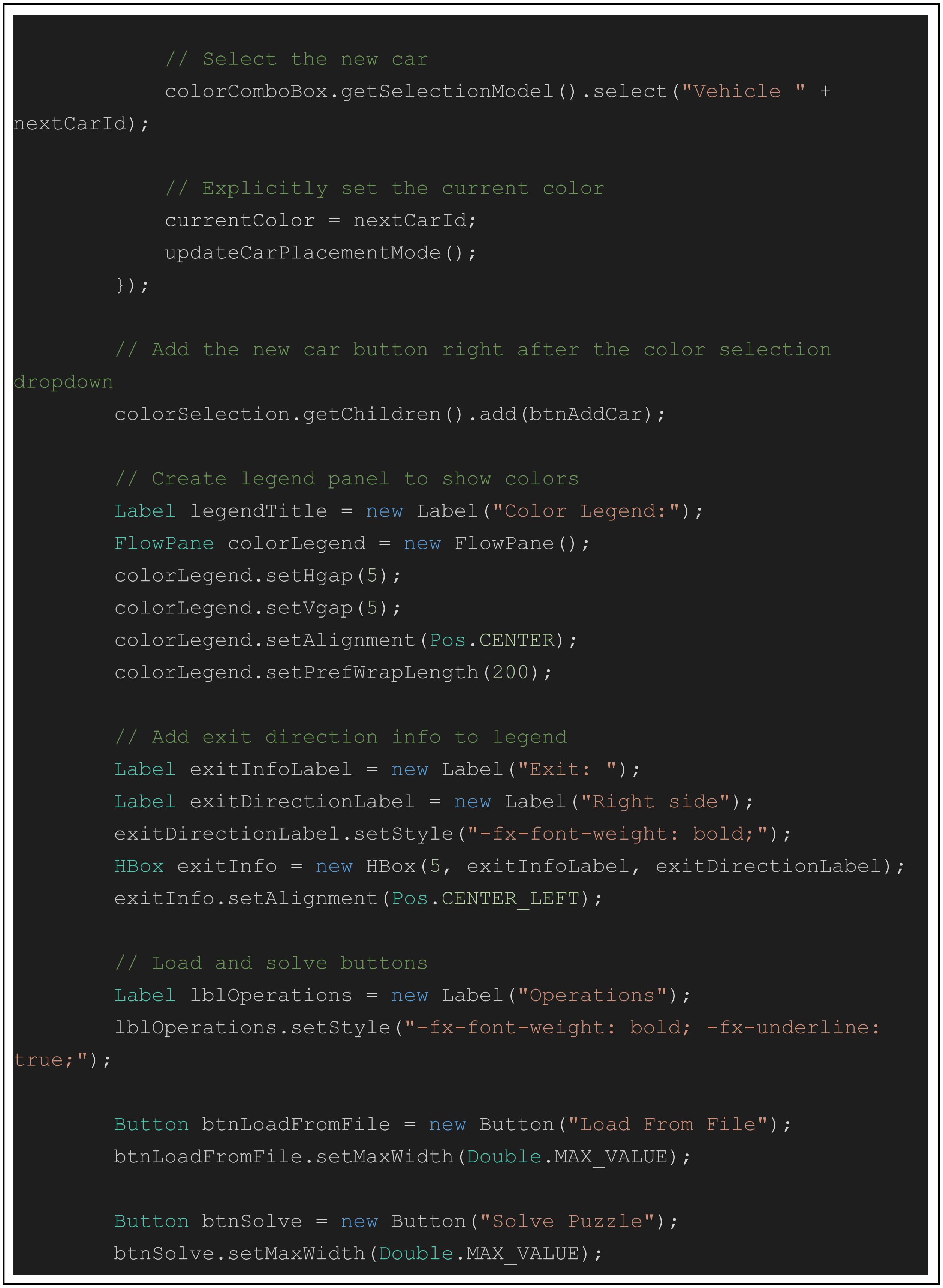
```
        heuristicComboBox.getSelectionModel().selectFirst();
        heuristicConfig.getChildren().addAll(lblHeuristic,
heuristicComboBox);

        // Primary piece orientation
        VBox orientationConfig = new VBox(5);
        orientationConfig.setAlignment(Pos.CENTER_LEFT);
        Label lblOrientation = new Label("Primary Piece Orientation:");
        HBox orientationButtons = new HBox(10);
        orientationButtons.setAlignment(Pos.CENTER);
        ToggleGroup orientationGroup = new ToggleGroup();
        RadioButton rbHorizontal = new RadioButton("Horizontal");
        rbHorizontal.setToggleGroup(orientationGroup);
        rbHorizontal.setSelected(true);
        RadioButton rbVertical = new RadioButton("Vertical");
        rbVertical.setToggleGroup(orientationGroup);
        orientationButtons.getChildren().addAll(rbHorizontal,
rbVertical);
        orientationConfig.getChildren().addAll(lblOrientation,
orientationButtons);

        // Exit side configuration - replacing exit position spinner
        VBox exitSideConfig = new VBox(5);
        exitSideConfig.setAlignment(Pos.CENTER_LEFT);
        Label lblExitSide = new Label("Exit Side:");

        // Toggle group for horizontal exit sides (initially visible)
        ToggleGroup exitHorizontalGroup = new ToggleGroup();
        HBox horizontalExitOptions = new HBox(10);
        horizontalExitOptions.setAlignment(Pos.CENTER);
        RadioButton rbRight = new RadioButton("Right");
        rbRight.setToggleGroup(exitHorizontalGroup);
        rbRight.setSelected(true); // Default to right
        RadioButton rbLeft = new RadioButton("Left");
        rbLeft.setToggleGroup(exitHorizontalGroup);
        horizontalExitOptions.getChildren().addAll(rbRight, rbLeft);

        // Toggle group for vertical exit sides (initially hidden)
        ToggleGroup exitVerticalGroup = new ToggleGroup();
```



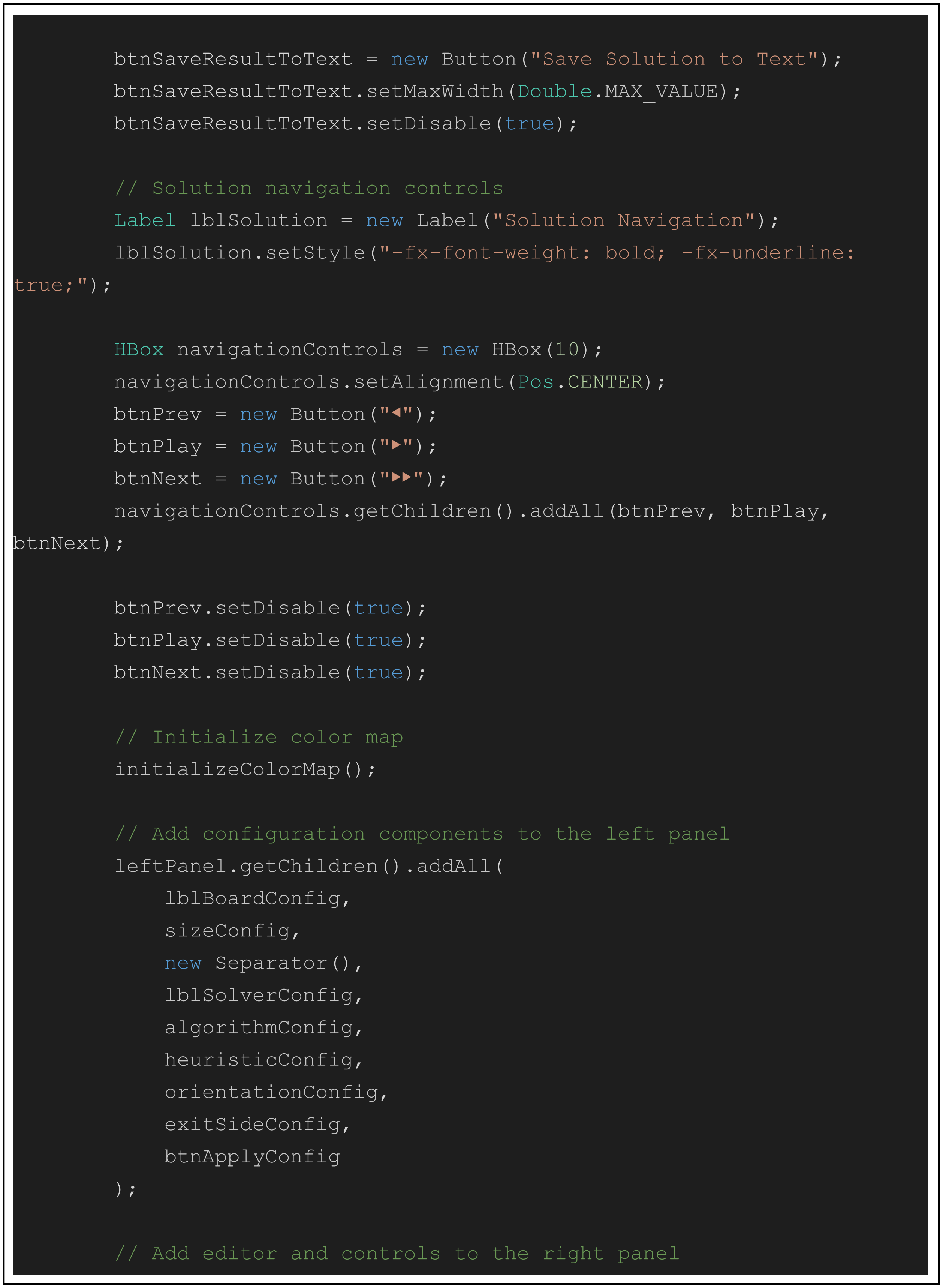
```
        HBox verticalExitOptions = new HBox(10);
        verticalExitOptions.setAlignment(Pos.CENTER);
        RadioButton rbBottom = new RadioButton("Bottom");
        rbBottom.setToggleGroup(exitVerticalGroup);
        rbBottom.setSelected(true); // Default to bottom
        RadioButton rbTop = new RadioButton("Top");
        rbTop.setToggleGroup(exitVerticalGroup);
        verticalExitOptions.getChildren().addAll(rbBottom, rbTop);

        // Initially show horizontal options
        exitSideConfig.getChildren().addAll(lblExitSide,
horizontalExitOptions);

        // Apply button to set up the board
        Button btnApplyConfig = new Button("Apply Configuration");
        btnApplyConfig.setMaxWidth(Double.MAX_VALUE);

        // ===== COMPONENTS FOR RIGHT PANEL =====
        // Color selection
        Label lblBoardEditor = new Label("Board Editor");
        lblBoardEditor.setStyle("-fx-font-weight: bold; -fx-underline:
true;");

        VBox colorSelection = new VBox(5);
        colorSelection.setAlignment(Pos.CENTER_LEFT);
        Label lblColor = new Label("Select Piece:");
        ComboBox<String> colorComboBox = new ComboBox<>();
        colorComboBox.setMaxWidth(Double.MAX_VALUE);

colorComboBox.setItems(FXCollections.observableArrayList("Primary
(Red)"));
        colorComboBox.getSelectionModel().selectFirst();
        colorSelection.getChildren().addAll(lblColor, colorComboBox);

        // Add button to add new cars to the board
        Button btnAddCar = new Button("Add New Car");
        btnAddCar.setMaxWidth(Double.MAX_VALUE);
        btnAddCar.setOnAction(e -> {
            // Find next available car ID (letter)
```



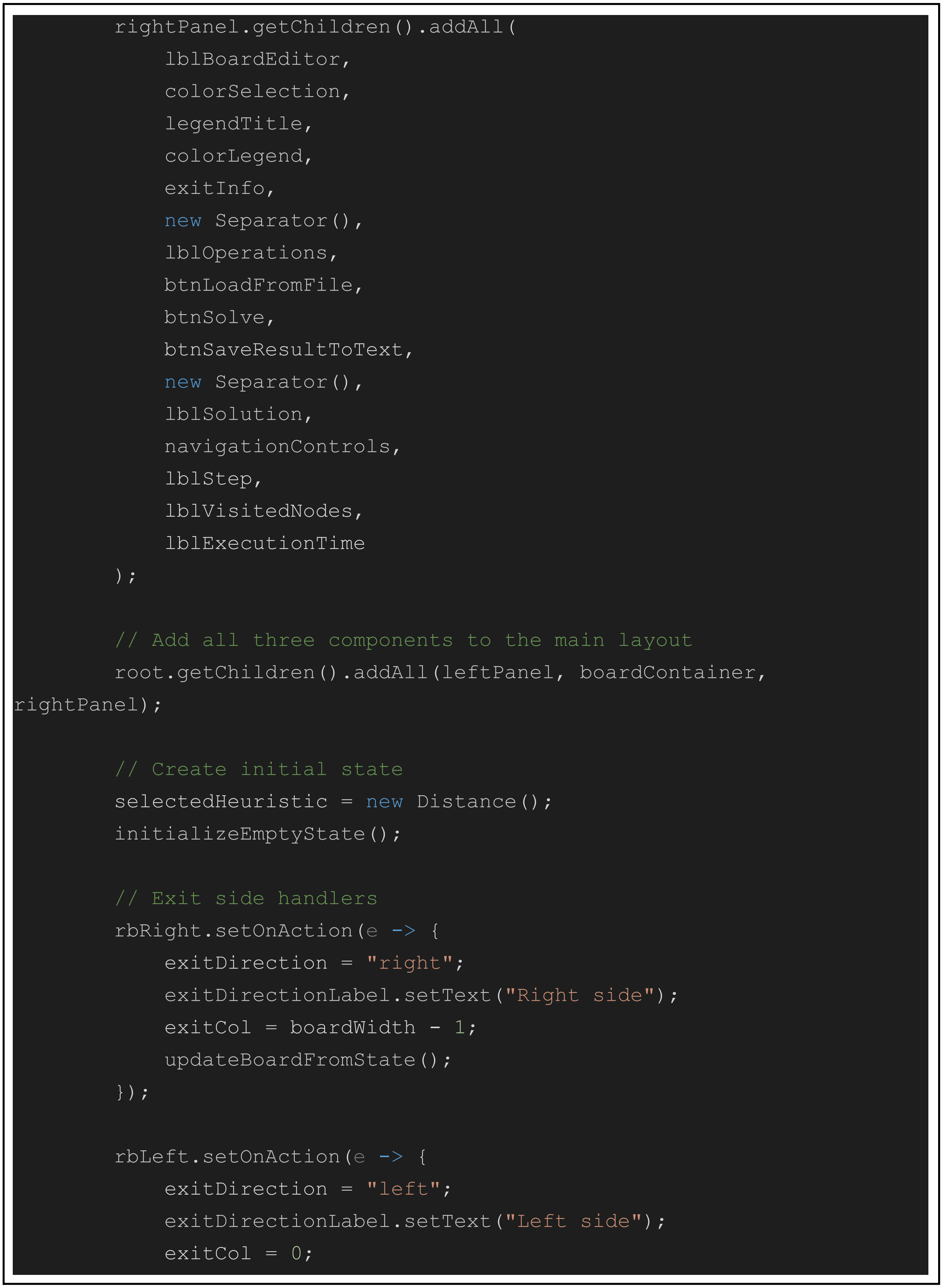
```
            char nextCarId = findNextAvailableCarId();

            // Don't proceed if we're out of letters
            if (nextCarId > 'Z') {
                Alert alert = new Alert(Alert.AlertType.WARNING);
                alert.setTitle("Too Many Cars");
                alert.setHeaderText("Maximum number of cars reached");
                alert.setContentText("You can only have up to 26
non-primary cars (A-Z).");
                alert.showAndWait();
                return;
            }

            // Add the new car to the state (with empty position for
now)
            Map<Character, Car> newCars = new
HashMap<>(currentState.cars);
            // Note: We're not adding an actual car yet, just reserving
the ID

            // Rebuild the dropdown with sorted vehicles
            colorComboBox.getItems().clear();
            colorComboBox.getItems().add("Primary (Red)");

            // Get all existing car IDs plus the new one, sorted
alphabetically
            List<Character> carIds = new
ArrayList<>(currentState.cars.keySet());
            if (!carIds.contains(nextCarId)) {
                carIds.add(nextCarId);
            }
            Collections.sort(carIds);

            // Add all non-primary cars to the dropdown
            for (char carId : carIds) {
                if (carId != 'P') {
                    colorComboBox.getItems().add("Vehicle " + carId);
                }
            }
```



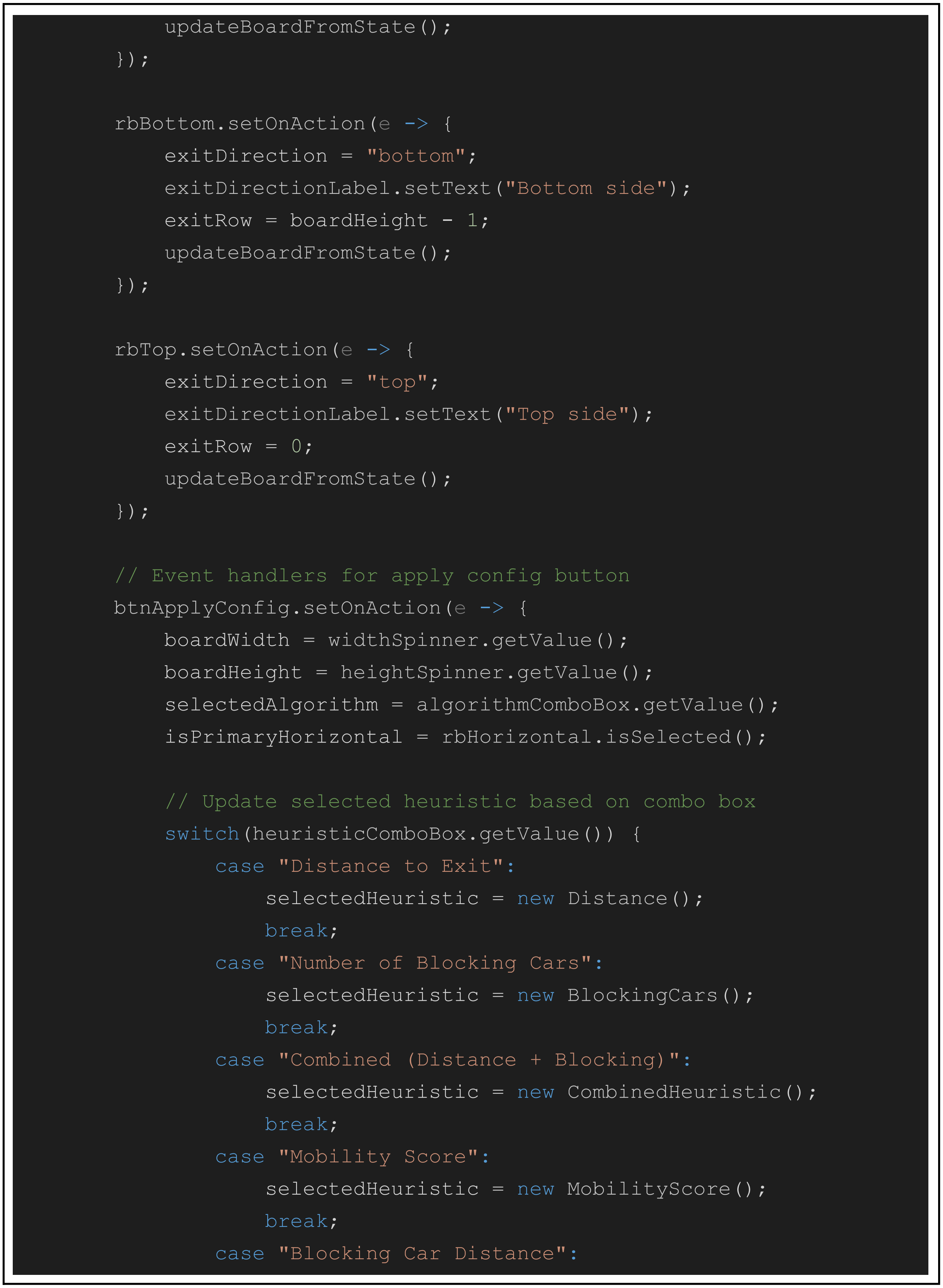
```
            // Select the new car
            colorComboBox.getSelectionModel().select("Vehicle " +
nextCarId);

            // Explicitly set the current color
            currentColor = nextCarId;
            updateCarPlacementMode();
        });

        // Add the new car button right after the color selection
dropdown
        colorSelection.getChildren().add(btnAddCar);

        // Create legend panel to show colors
        Label legendTitle = new Label("Color Legend:");
        FlowPane colorLegend = new FlowPane();
        colorLegend.setHgap(5);
        colorLegend.setVgap(5);
        colorLegend.setAlignment(Pos.CENTER);
        colorLegend.setPrefWrapLength(200);

        // Add exit direction info to legend
        Label exitInfoLabel = new Label("Exit: ");
        Label exitDirectionLabel = new Label("Right side");
        exitDirectionLabel.setStyle("-fx-font-weight: bold;");
        HBox exitInfo = new HBox(5, exitInfoLabel, exitDirectionLabel);
        exitInfo.setAlignment(Pos.CENTER_LEFT);

        // Load and solve buttons
        Label lblOperations = new Label("Operations");
        lblOperations.setStyle("-fx-font-weight: bold; -fx-underline:
true;");

        Button btnLoadFromFile = new Button("Load From File");
        btnLoadFromFile.setMaxWidth(Double.MAX_VALUE);

        Button btnSolve = new Button("Solve Puzzle");
        btnSolve.setMaxWidth(Double.MAX_VALUE);
```



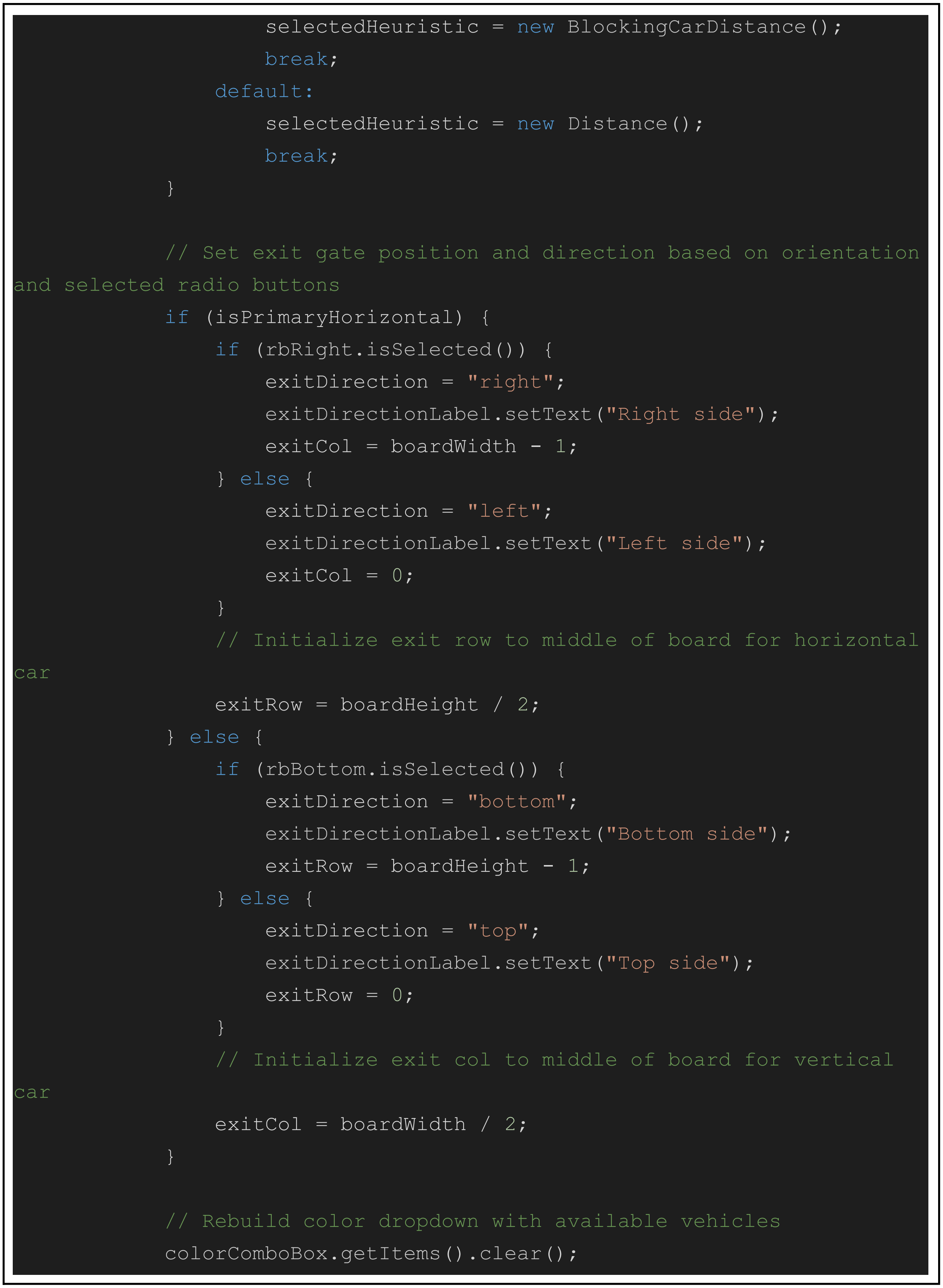
```
        btnSaveResultToText = new Button("Save Solution to Text");
        btnSaveResultToText.setMaxWidth(Double.MAX_VALUE);
        btnSaveResultToText.setDisable(true);

        // Solution navigation controls
        Label lblSolution = new Label("Solution Navigation");
        lblSolution.setStyle("-fx-font-weight: bold; -fx-underline:
true;");

        HBox navigationControls = new HBox(10);
        navigationControls.setAlignment(Pos.CENTER);
        btnPrev = new Button("◄");
        btnPlay = new Button("►");
        btnNext = new Button("►►");
        navigationControls.getChildren().addAll(btnPrev, btnPlay,
btnNext);

        btnPrev.setDisable(true);
        btnPlay.setDisable(true);
        btnNext.setDisable(true);

        // Initialize color map
        initializeColorMap();

        // Add configuration components to the left panel
        leftPanel.getChildren().addAll(
            lblBoardConfig,
            sizeConfig,
            new Separator(),
            lblSolverConfig,
            algorithmConfig,
            heuristicConfig,
            orientationConfig,
            exitSideConfig,
            btnApplyConfig
        );

        // Add editor and controls to the right panel
```



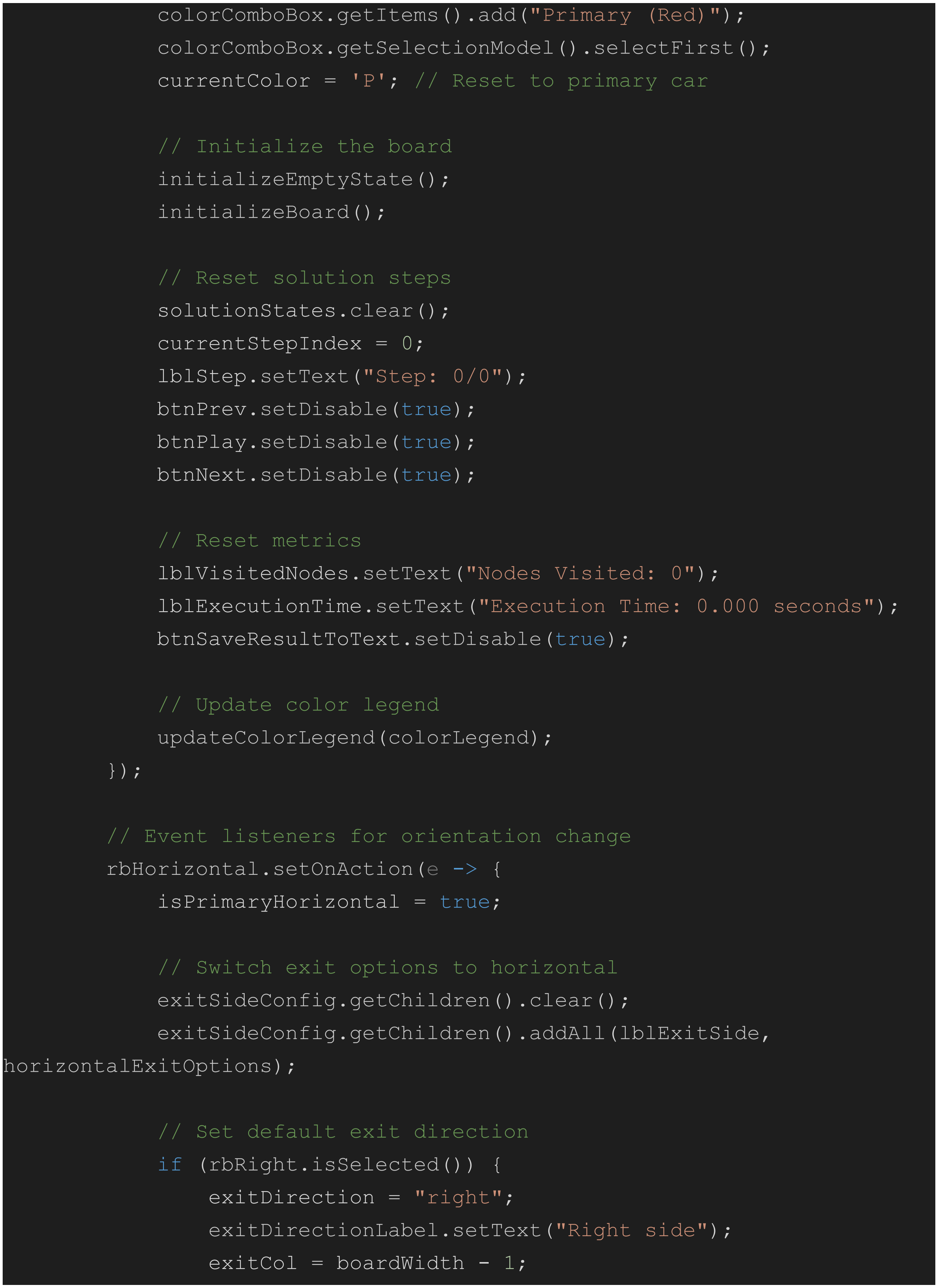
```
        rightPanel.getChildren().addAll(
            lblBoardEditor,
            colorSelection,
            legendTitle,
            colorLegend,
            exitInfo,
            new Separator(),
            lblOperations,
            btnLoadFromFile,
            btnSolve,
            btnSaveResultToText,
            new Separator(),
            lblSolution,
            navigationControls,
            lblStep,
            lblVisitedNodes,
            lblExecutionTime
        );

        // Add all three components to the main layout
        root.getChildren().addAll(leftPanel, boardContainer,
rightPanel);

        // Create initial state
        selectedHeuristic = new Distance();
        initializeEmptyState();

        // Exit side handlers
        rbRight.setOnAction(e -> {
            exitDirection = "right";
            exitDirectionLabel.setText("Right side");
            exitCol = boardWidth - 1;
            updateBoardFromState();
        });

        rbLeft.setOnAction(e -> {
            exitDirection = "left";
            exitDirectionLabel.setText("Left side");
            exitCol = 0;
```



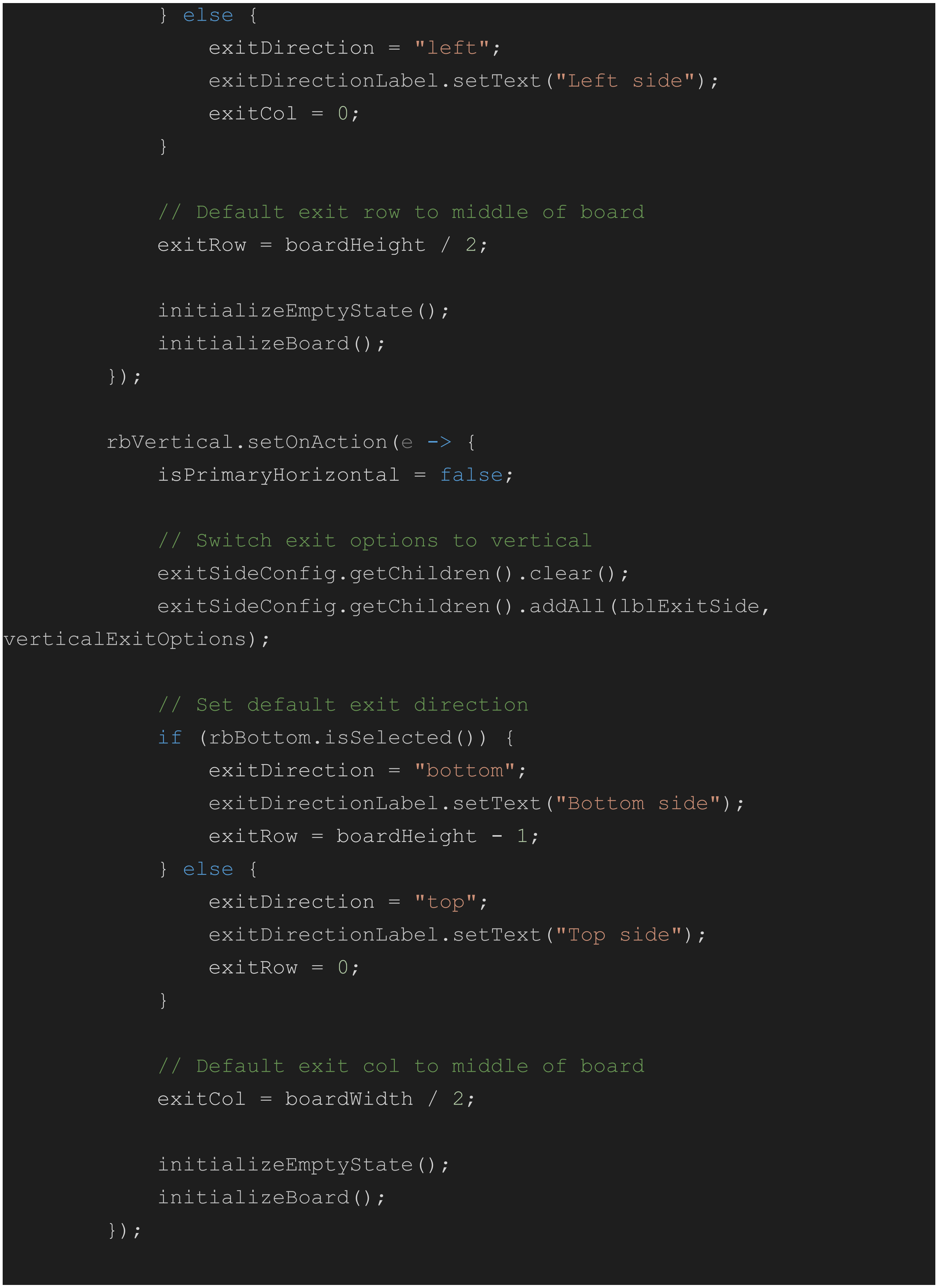
```
            updateBoardFromState();
        });

        rbBottom.setOnAction(e -> {
            exitDirection = "bottom";
            exitDirectionLabel.setText("Bottom side");
            exitRow = boardHeight - 1;
            updateBoardFromState();
        });

        rbTop.setOnAction(e -> {
            exitDirection = "top";
            exitDirectionLabel.setText("Top side");
            exitRow = 0;
            updateBoardFromState();
        });

        // Event handlers for apply config button
        btnApplyConfig.setOnAction(e -> {
            boardWidth = widthSpinner.getValue();
            boardHeight = heightSpinner.getValue();
            selectedAlgorithm = algorithmComboBox.getValue();
            isPrimaryHorizontal = rbHorizontal.isSelected();

            // Update selected heuristic based on combo box
            switch(heuristicComboBox.getValue()) {
                case "Distance to Exit":
                    selectedHeuristic = new Distance();
                    break;
                case "Number of Blocking Cars":
                    selectedHeuristic = new BlockingCars();
                    break;
                case "Combined (Distance + Blocking)":
                    selectedHeuristic = new CombinedHeuristic();
                    break;
                case "Mobility Score":
                    selectedHeuristic = new MobilityScore();
                    break;
                case "Blocking Car Distance":
```



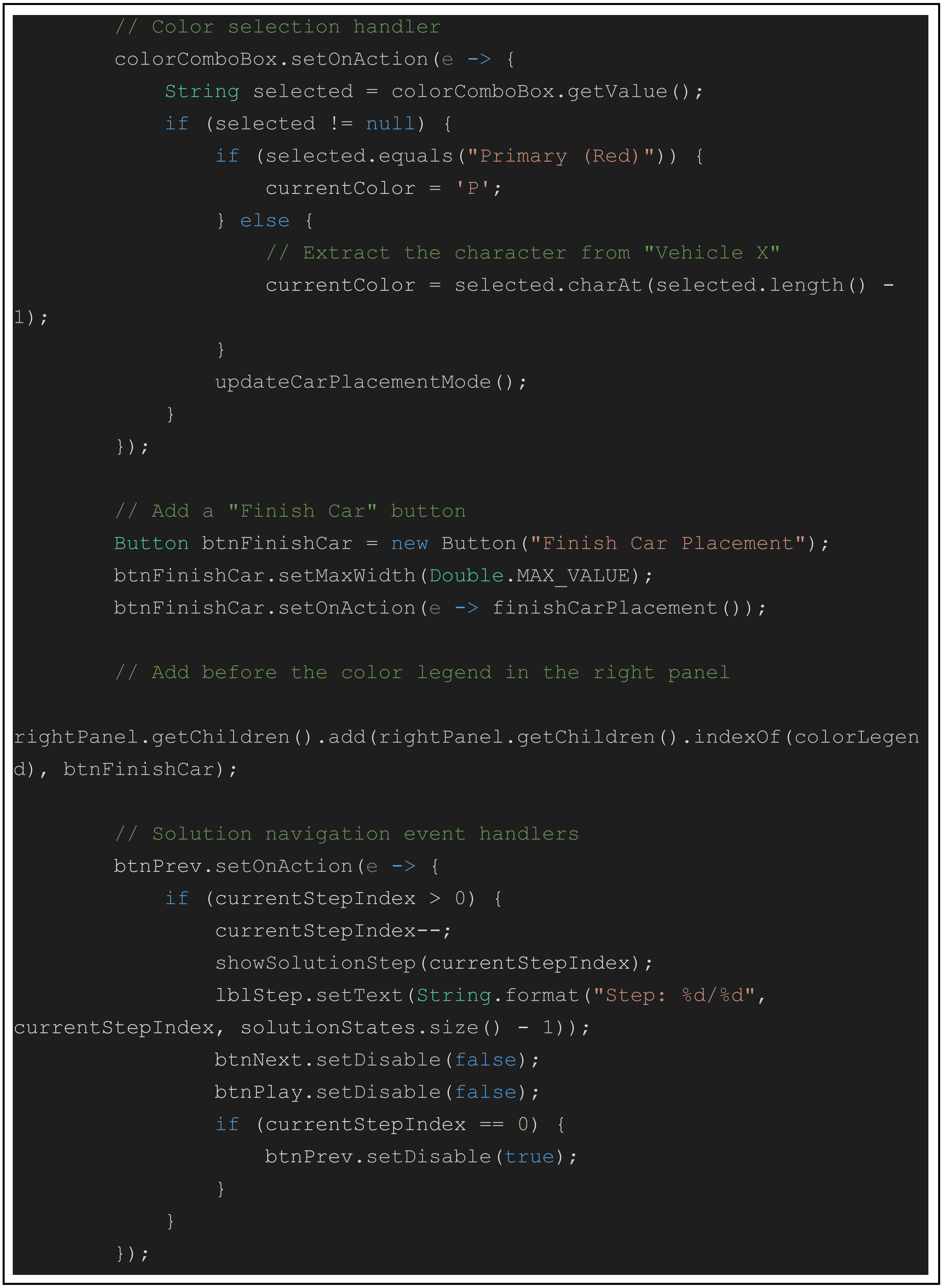
```java
                    selectedHeuristic = new BlockingCarDistance();
                    break;
                default:
                    selectedHeuristic = new Distance();
                    break;
            }

            // Set exit gate position and direction based on orientation and selected radio buttons
            if (isPrimaryHorizontal) {
                if (rbRight.isSelected()) {
                    exitDirection = "right";
                    exitDirectionLabel.setText("Right side");
                    exitCol = boardWidth - 1;
                } else {
                    exitDirection = "left";
                    exitDirectionLabel.setText("Left side");
                    exitCol = 0;
                }
                // Initialize exit row to middle of board for horizontal car
                exitRow = boardHeight / 2;
            } else {
                if (rbBottom.isSelected()) {
                    exitDirection = "bottom";
                    exitDirectionLabel.setText("Bottom side");
                    exitRow = boardHeight - 1;
                } else {
                    exitDirection = "top";
                    exitDirectionLabel.setText("Top side");
                    exitRow = 0;
                }
                // Initialize exit col to middle of board for vertical car
                exitCol = boardWidth / 2;
            }

            // Rebuild color dropdown with available vehicles
            colorComboBox.getItems().clear();
```



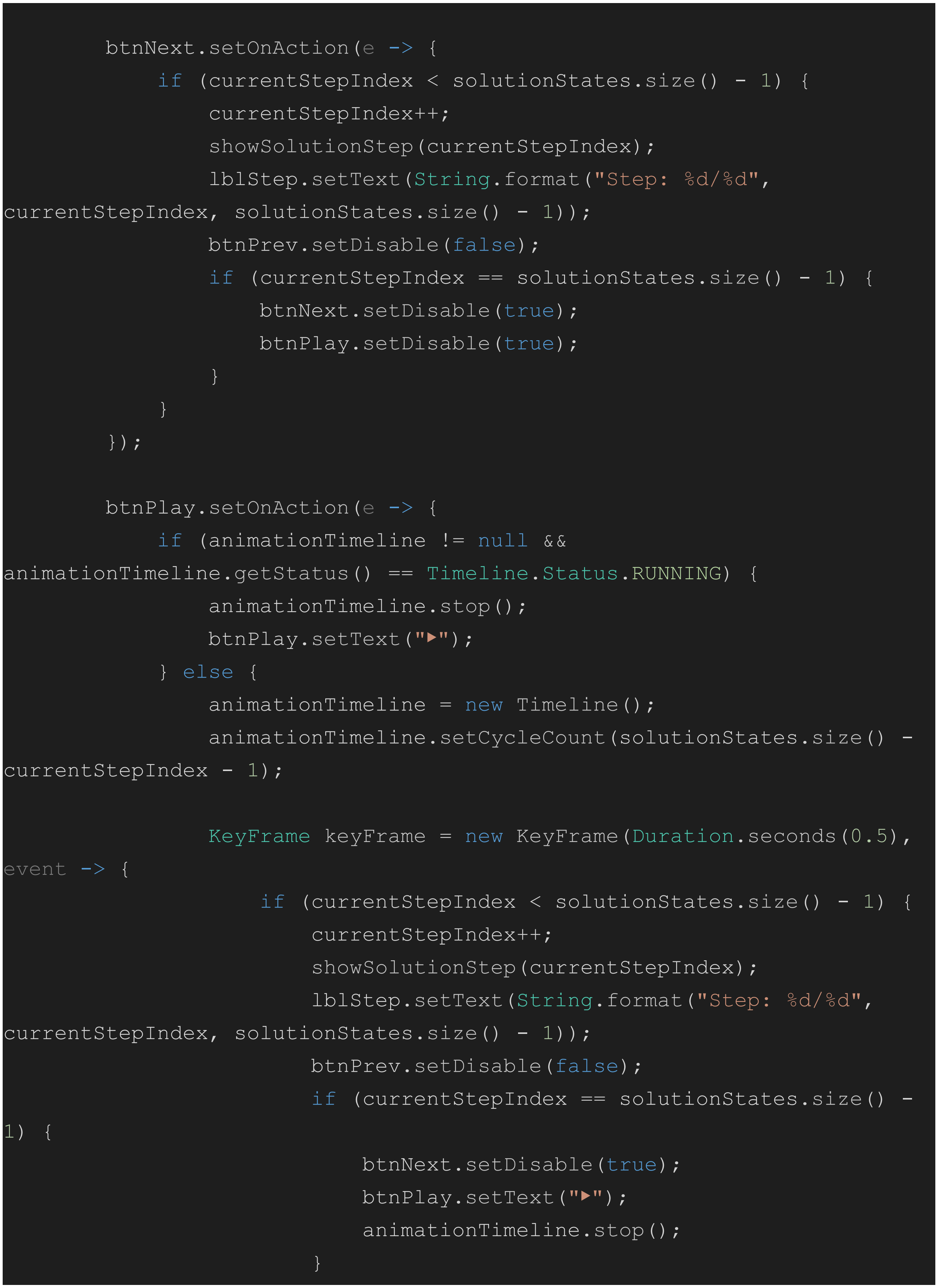
```
            colorComboBox.getItems().add("Primary (Red)");
            colorComboBox.getSelectionModel().selectFirst();
            currentColor = 'P'; // Reset to primary car

            // Initialize the board
            initializeEmptyState();
            initializeBoard();

            // Reset solution steps
            solutionStates.clear();
            currentStepIndex = 0;
            lblStep.setText("Step: 0/0");
            btnPrev.setDisable(true);
            btnPlay.setDisable(true);
            btnNext.setDisable(true);

            // Reset metrics
            lblVisitedNodes.setText("Nodes Visited: 0");
            lblExecutionTime.setText("Execution Time: 0.000 seconds");
            btnSaveResultToText.setDisable(true);

            // Update color legend
            updateColorLegend(colorLegend);
        });

        // Event listeners for orientation change
        rbHorizontal.setOnAction(e -> {
            isPrimaryHorizontal = true;

            // Switch exit options to horizontal
            exitSideConfig.getChildren().clear();
            exitSideConfig.getChildren().addAll(lblExitSide,
horizontalExitOptions);

            // Set default exit direction
            if (rbRight.isSelected()) {
                exitDirection = "right";
                exitDirectionLabel.setText("Right side");
                exitCol = boardWidth - 1;
```



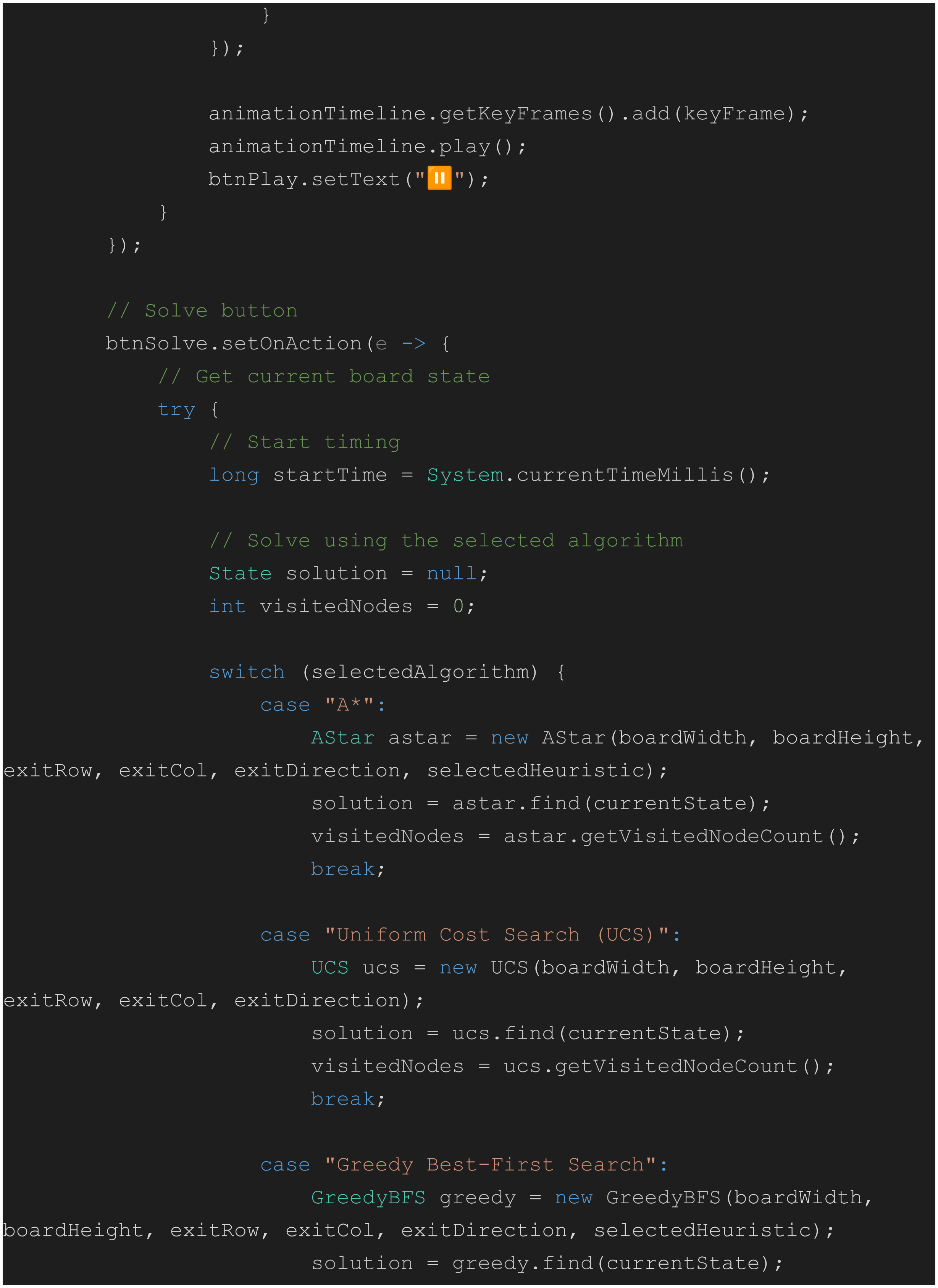
```
            } else {
                exitDirection = "left";
                exitDirectionLabel.setText("Left side");
                exitCol = 0;
            }

            // Default exit row to middle of board
            exitRow = boardHeight / 2;

            initializeEmptyState();
            initializeBoard();
        });

        rbVertical.setOnAction(e -> {
            isPrimaryHorizontal = false;

            // Switch exit options to vertical
            exitSideConfig.getChildren().clear();
            exitSideConfig.getChildren().addAll(lblExitSide,
verticalExitOptions);

            // Set default exit direction
            if (rbBottom.isSelected()) {
                exitDirection = "bottom";
                exitDirectionLabel.setText("Bottom side");
                exitRow = boardHeight - 1;
            } else {
                exitDirection = "top";
                exitDirectionLabel.setText("Top side");
                exitRow = 0;
            }

            // Default exit col to middle of board
            exitCol = boardWidth / 2;

            initializeEmptyState();
            initializeBoard();
        });
```



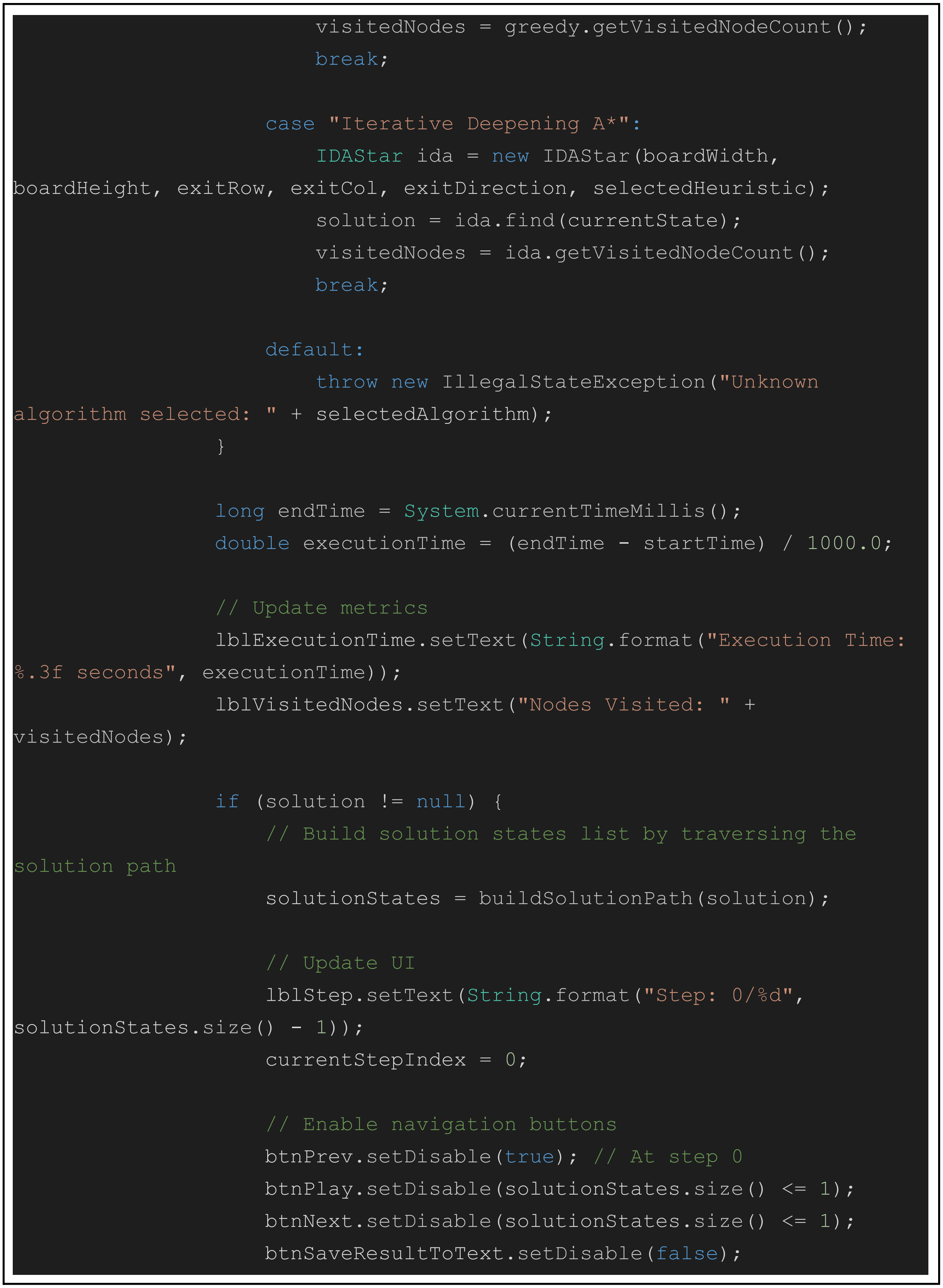
```
        // Color selection handler
        colorComboBox.setOnAction(e -> {
            String selected = colorComboBox.getValue();
            if (selected != null) {
                if (selected.equals("Primary (Red)")) {
                    currentColor = 'P';
                } else {
                    // Extract the character from "Vehicle X"
                    currentColor = selected.charAt(selected.length() -
1);
                }
                updateCarPlacementMode();
            }
        });

        // Add a "Finish Car" button
        Button btnFinishCar = new Button("Finish Car Placement");
        btnFinishCar.setMaxWidth(Double.MAX_VALUE);
        btnFinishCar.setOnAction(e -> finishCarPlacement());

        // Add before the color legend in the right panel

rightPanel.getChildren().add(rightPanel.getChildren().indexOf(colorLegen
d), btnFinishCar);

        // Solution navigation event handlers
        btnPrev.setOnAction(e -> {
            if (currentStepIndex > 0) {
                currentStepIndex--;
                showSolutionStep(currentStepIndex);
                lblStep.setText(String.format("Step: %d/%d",
currentStepIndex, solutionStates.size() - 1));
                btnNext.setDisable(false);
                btnPlay.setDisable(false);
                if (currentStepIndex == 0) {
                    btnPrev.setDisable(true);
                }
            }
        });
```



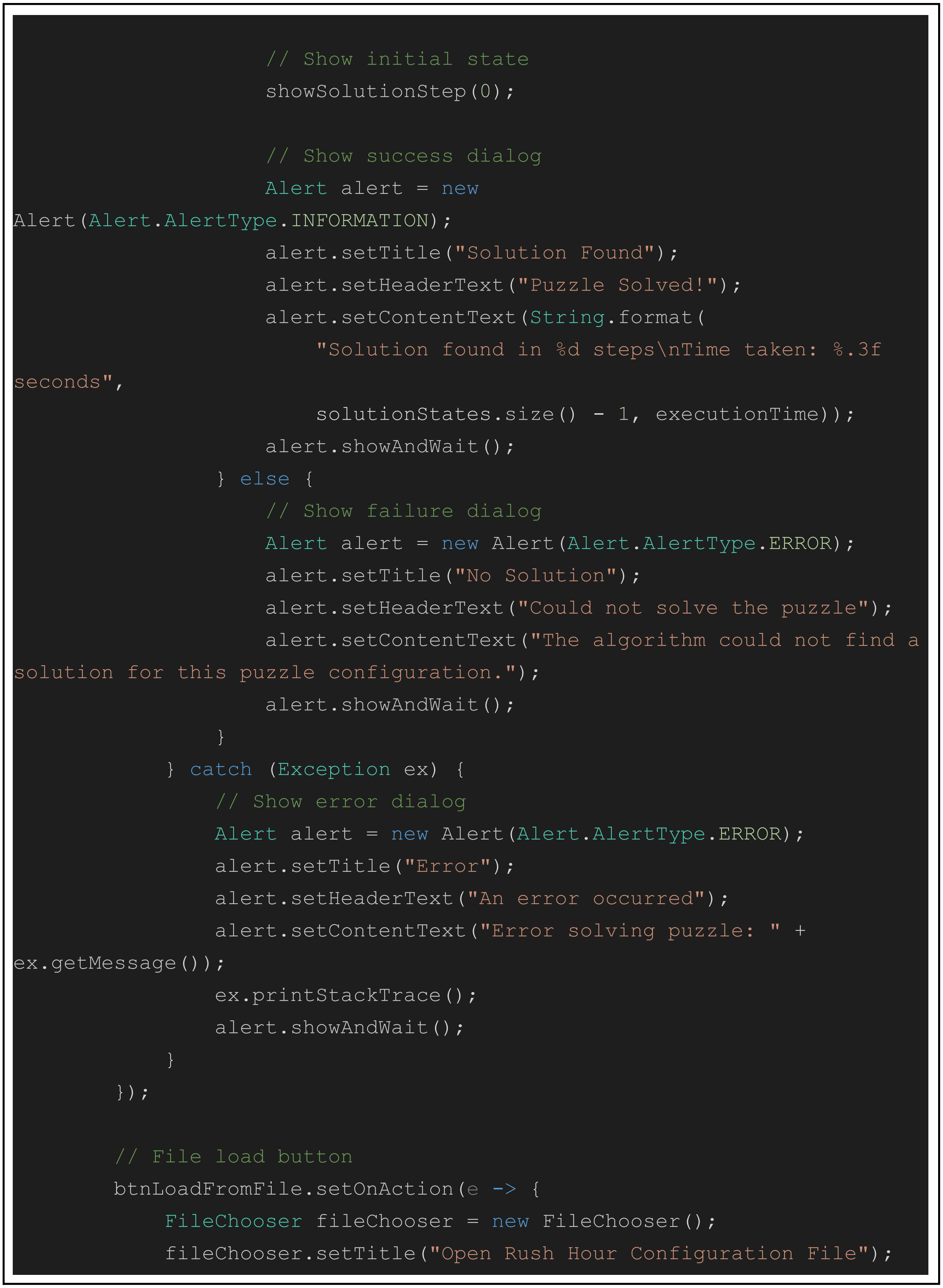
```
        btnNext.setOnAction(e -> {
            if (currentStepIndex < solutionStates.size() - 1) {
                currentStepIndex++;
                showSolutionStep(currentStepIndex);
                lblStep.setText(String.format("Step: %d/%d",
currentStepIndex, solutionStates.size() - 1));
                btnPrev.setDisable(false);
                if (currentStepIndex == solutionStates.size() - 1) {
                    btnNext.setDisable(true);
                    btnPlay.setDisable(true);
                }
            }
        });

        btnPlay.setOnAction(e -> {
            if (animationTimeline != null &&
animationTimeline.getStatus() == Timeline.Status.RUNNING) {
                animationTimeline.stop();
                btnPlay.setText("▶");
            } else {
                animationTimeline = new Timeline();
                animationTimeline.setCycleCount(solutionStates.size() -
currentStepIndex - 1);

                KeyFrame keyFrame = new KeyFrame(Duration.seconds(0.5),
event -> {
                    if (currentStepIndex < solutionStates.size() - 1) {
                        currentStepIndex++;
                        showSolutionStep(currentStepIndex);
                        lblStep.setText(String.format("Step: %d/%d",
currentStepIndex, solutionStates.size() - 1));
                        btnPrev.setDisable(false);
                        if (currentStepIndex == solutionStates.size() -
1) {
                            btnNext.setDisable(true);
                            btnPlay.setText("▶");
                            animationTimeline.stop();
                        }
```



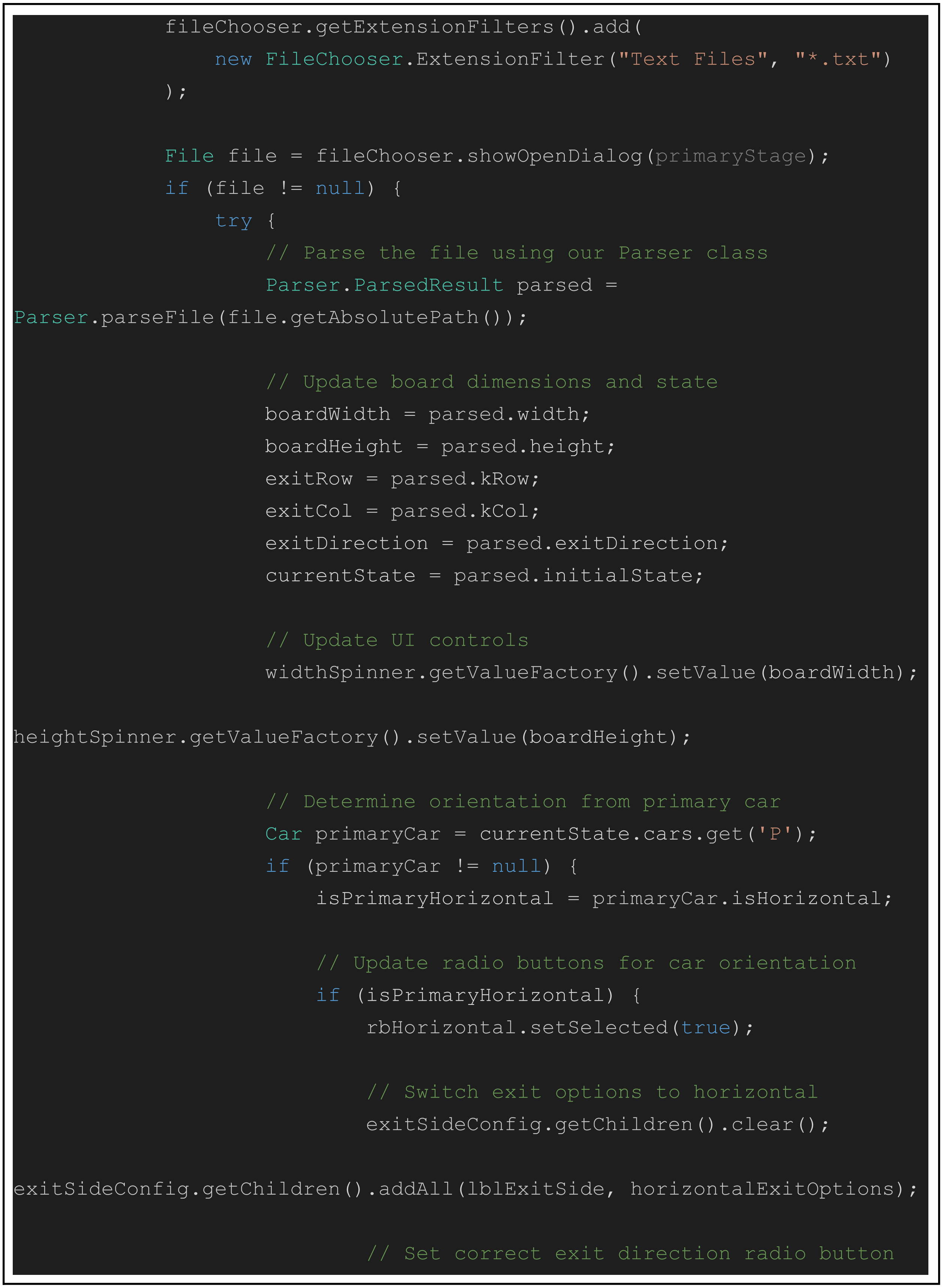
```java
                    }
                });

                animationTimeline.getKeyFrames().add(keyFrame);
                animationTimeline.play();
                btnPlay.setText("⏸");
            }
        });

        // Solve button
        btnSolve.setOnAction(e -> {
            // Get current board state
            try {
                // Start timing
                long startTime = System.currentTimeMillis();

                // Solve using the selected algorithm
                State solution = null;
                int visitedNodes = 0;

                switch (selectedAlgorithm) {
                    case "A*":
                        AStar astar = new AStar(boardWidth, boardHeight,
exitRow, exitCol, exitDirection, selectedHeuristic);
                        solution = astar.find(currentState);
                        visitedNodes = astar.getVisitedNodeCount();
                        break;

                    case "Uniform Cost Search (UCS)":
                        UCS ucs = new UCS(boardWidth, boardHeight,
exitRow, exitCol, exitDirection);
                        solution = ucs.find(currentState);
                        visitedNodes = ucs.getVisitedNodeCount();
                        break;

                    case "Greedy Best-First Search":
                        GreedyBFS greedy = new GreedyBFS(boardWidth,
boardHeight, exitRow, exitCol, exitDirection, selectedHeuristic);
                        solution = greedy.find(currentState);
```



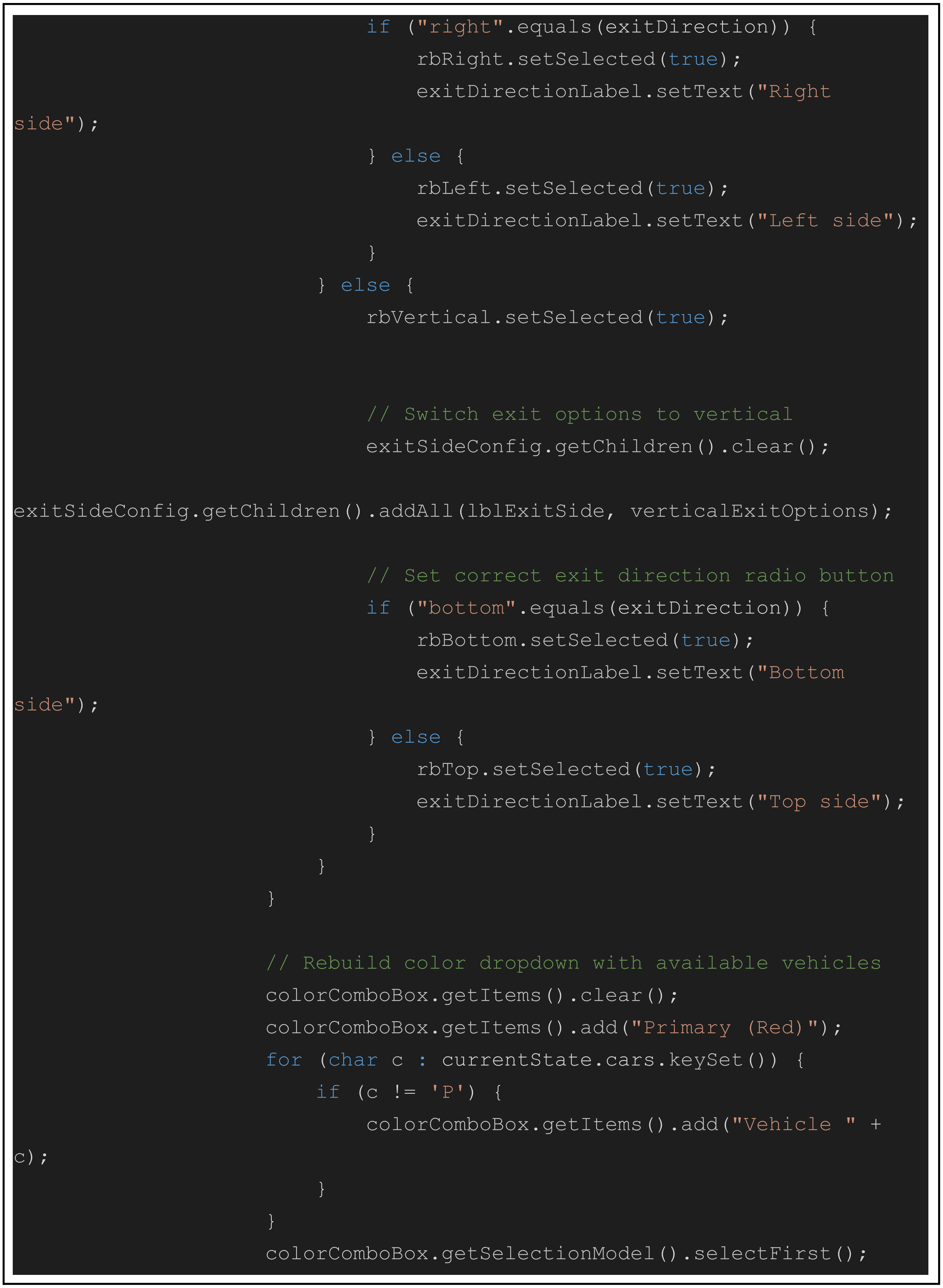
```
                        visitedNodes = greedy.getVisitedNodeCount();
                        break;

                    case "Iterative Deepening A*":
                        IDAStar ida = new IDAStar(boardWidth,
boardHeight, exitRow, exitCol, exitDirection, selectedHeuristic);
                        solution = ida.find(currentState);
                        visitedNodes = ida.getVisitedNodeCount();
                        break;

                    default:
                        throw new IllegalStateException("Unknown
algorithm selected: " + selectedAlgorithm);
                }

                long endTime = System.currentTimeMillis();
                double executionTime = (endTime - startTime) / 1000.0;

                // Update metrics
                lblExecutionTime.setText(String.format("Execution Time:
%.3f seconds", executionTime));
                lblVisitedNodes.setText("Nodes Visited: " +
visitedNodes);

                if (solution != null) {
                    // Build solution states list by traversing the
solution path
                    solutionStates = buildSolutionPath(solution);

                    // Update UI
                    lblStep.setText(String.format("Step: 0/%d",
solutionStates.size() - 1));
                    currentStepIndex = 0;

                    // Enable navigation buttons
                    btnPrev.setDisable(true); // At step 0
                    btnPlay.setDisable(solutionStates.size() <= 1);
                    btnNext.setDisable(solutionStates.size() <= 1);
                    btnSaveResultToText.setDisable(false);
```



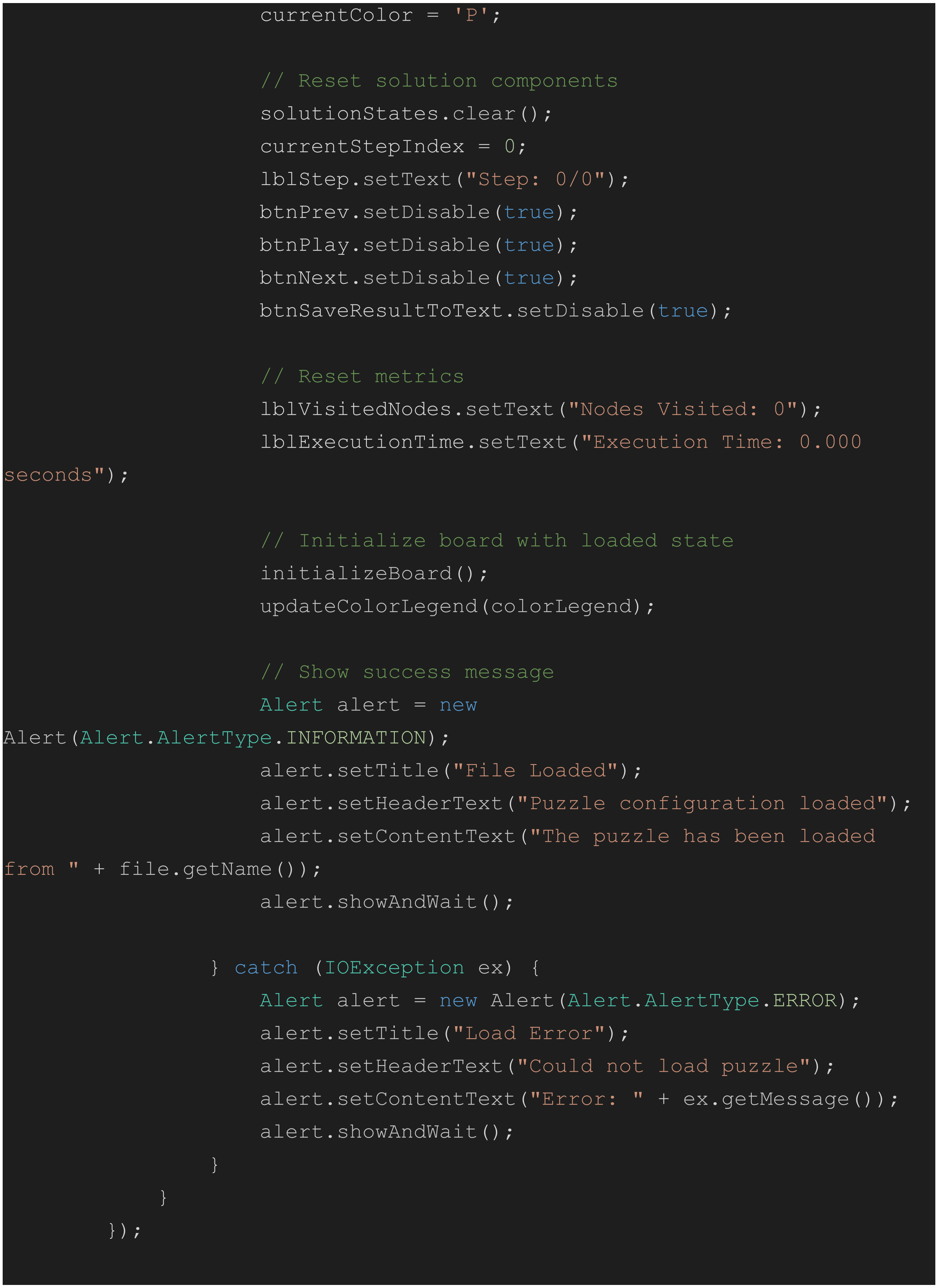
```
                    // Show initial state
                    showSolutionStep(0);

                    // Show success dialog
                    Alert alert = new
Alert(Alert.AlertType.INFORMATION);
                    alert.setTitle("Solution Found");
                    alert.setHeaderText("Puzzle Solved!");
                    alert.setContentText(String.format(
                        "Solution found in %d steps\nTime taken: %.3f
seconds",
                        solutionStates.size() - 1, executionTime));
                    alert.showAndWait();
                } else {
                    // Show failure dialog
                    Alert alert = new Alert(Alert.AlertType.ERROR);
                    alert.setTitle("No Solution");
                    alert.setHeaderText("Could not solve the puzzle");
                    alert.setContentText("The algorithm could not find a
solution for this puzzle configuration.");
                    alert.showAndWait();
                }
            } catch (Exception ex) {
                // Show error dialog
                Alert alert = new Alert(Alert.AlertType.ERROR);
                alert.setTitle("Error");
                alert.setHeaderText("An error occurred");
                alert.setContentText("Error solving puzzle: " +
ex.getMessage());
                ex.printStackTrace();
                alert.showAndWait();
            }
        });

        // File load button
        btnLoadFromFile.setOnAction(e -> {
            FileChooser fileChooser = new FileChooser();
            fileChooser.setTitle("Open Rush Hour Configuration File");
```



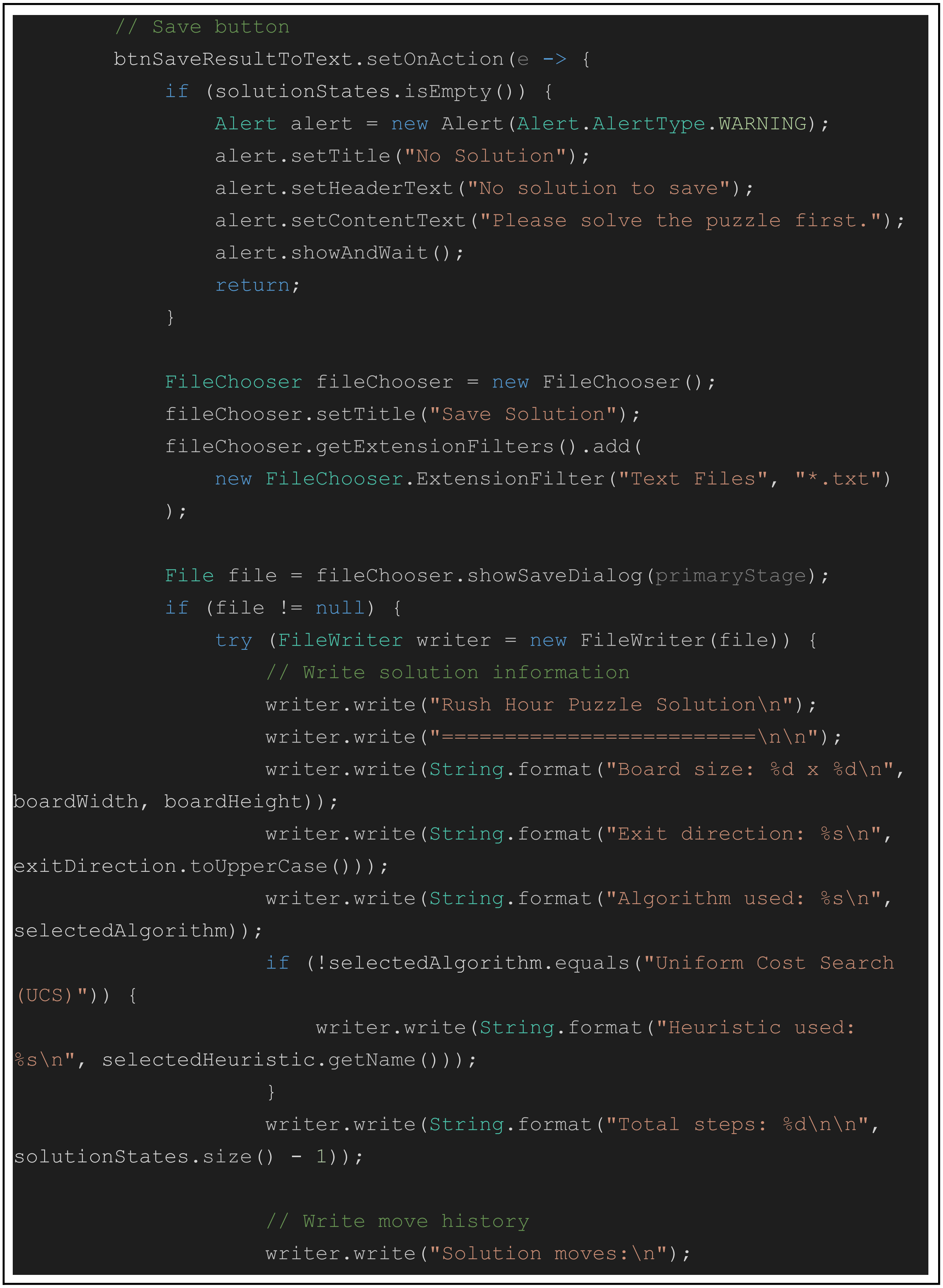
```
            fileChooser.getExtensionFilters().add(
                new FileChooser.ExtensionFilter("Text Files", "*.txt")
            );

            File file = fileChooser.showOpenDialog(primaryStage);
            if (file != null) {
                try {
                    // Parse the file using our Parser class
                    Parser.ParsedResult parsed =
Parser.parseFile(file.getAbsolutePath());

                    // Update board dimensions and state
                    boardWidth = parsed.width;
                    boardHeight = parsed.height;
                    exitRow = parsed.kRow;
                    exitCol = parsed.kCol;
                    exitDirection = parsed.exitDirection;
                    currentState = parsed.initialState;

                    // Update UI controls
                    widthSpinner.getValueFactory().setValue(boardWidth);

heightSpinner.getValueFactory().setValue(boardHeight);

                    // Determine orientation from primary car
                    Car primaryCar = currentState.cars.get('P');
                    if (primaryCar != null) {
                        isPrimaryHorizontal = primaryCar.isHorizontal;

                        // Update radio buttons for car orientation
                        if (isPrimaryHorizontal) {
                            rbHorizontal.setSelected(true);

                            // Switch exit options to horizontal
                            exitSideConfig.getChildren().clear();

exitSideConfig.getChildren().addAll(lblExitSide, horizontalExitOptions);

                            // Set correct exit direction radio button
```



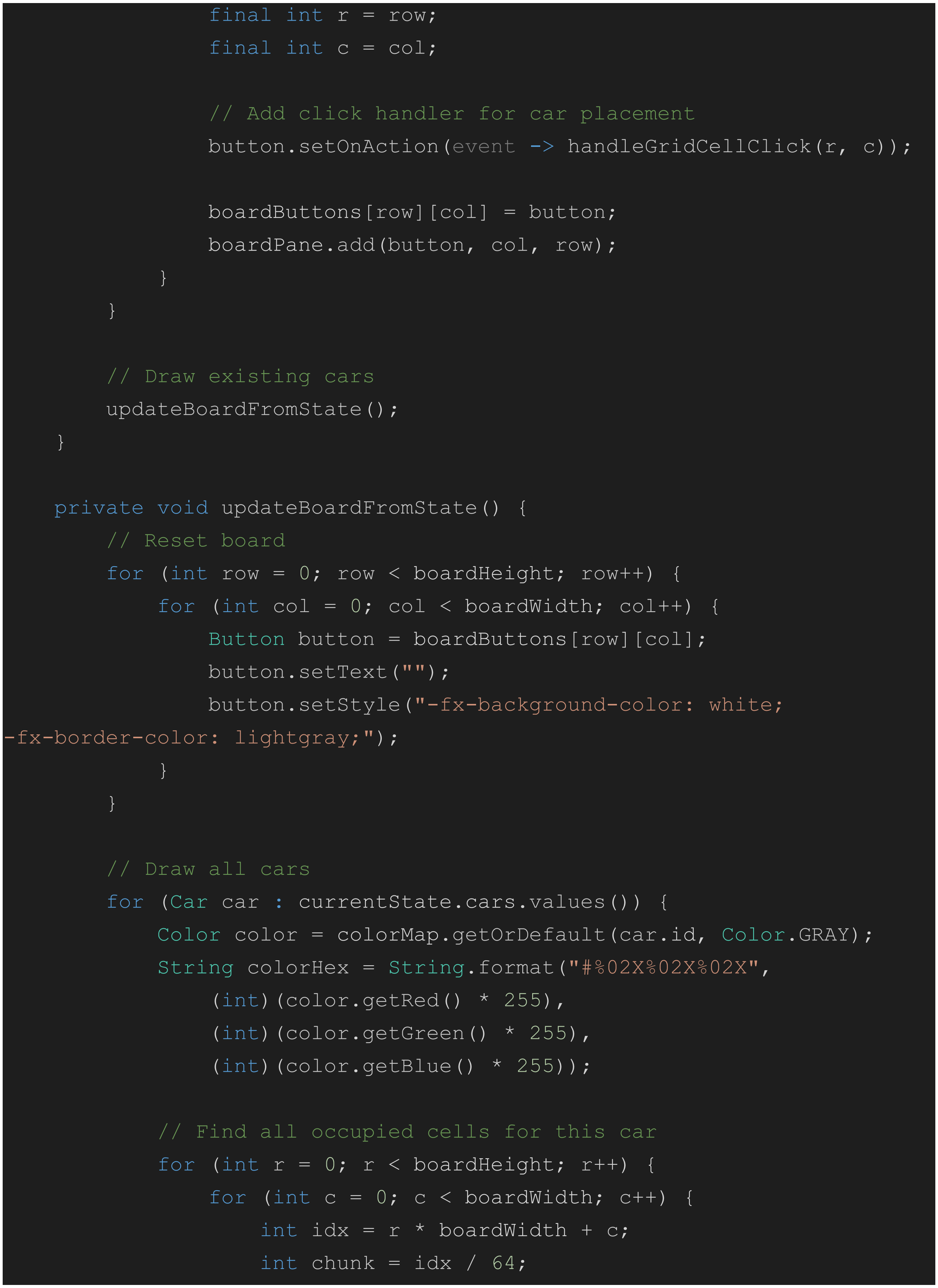
```
                            if ("right".equals(exitDirection)) {
                                rbRight.setSelected(true);
                                exitDirectionLabel.setText("Right
side");
                            } else {
                                rbLeft.setSelected(true);
                                exitDirectionLabel.setText("Left side");
                            }
                        } else {
                            rbVertical.setSelected(true);


                            // Switch exit options to vertical
                            exitSideConfig.getChildren().clear();

exitSideConfig.getChildren().addAll(lblExitSide, verticalExitOptions);

                            // Set correct exit direction radio button
                            if ("bottom".equals(exitDirection)) {
                                rbBottom.setSelected(true);
                                exitDirectionLabel.setText("Bottom
side");
                            } else {
                                rbTop.setSelected(true);
                                exitDirectionLabel.setText("Top side");
                            }
                        }
                    }

                    // Rebuild color dropdown with available vehicles
                    colorComboBox.getItems().clear();
                    colorComboBox.getItems().add("Primary (Red)");
                    for (char c : currentState.cars.keySet()) {
                        if (c != 'P') {
                            colorComboBox.getItems().add("Vehicle " +
c);
                        }
                    }
                    colorComboBox.getSelectionModel().selectFirst();
```



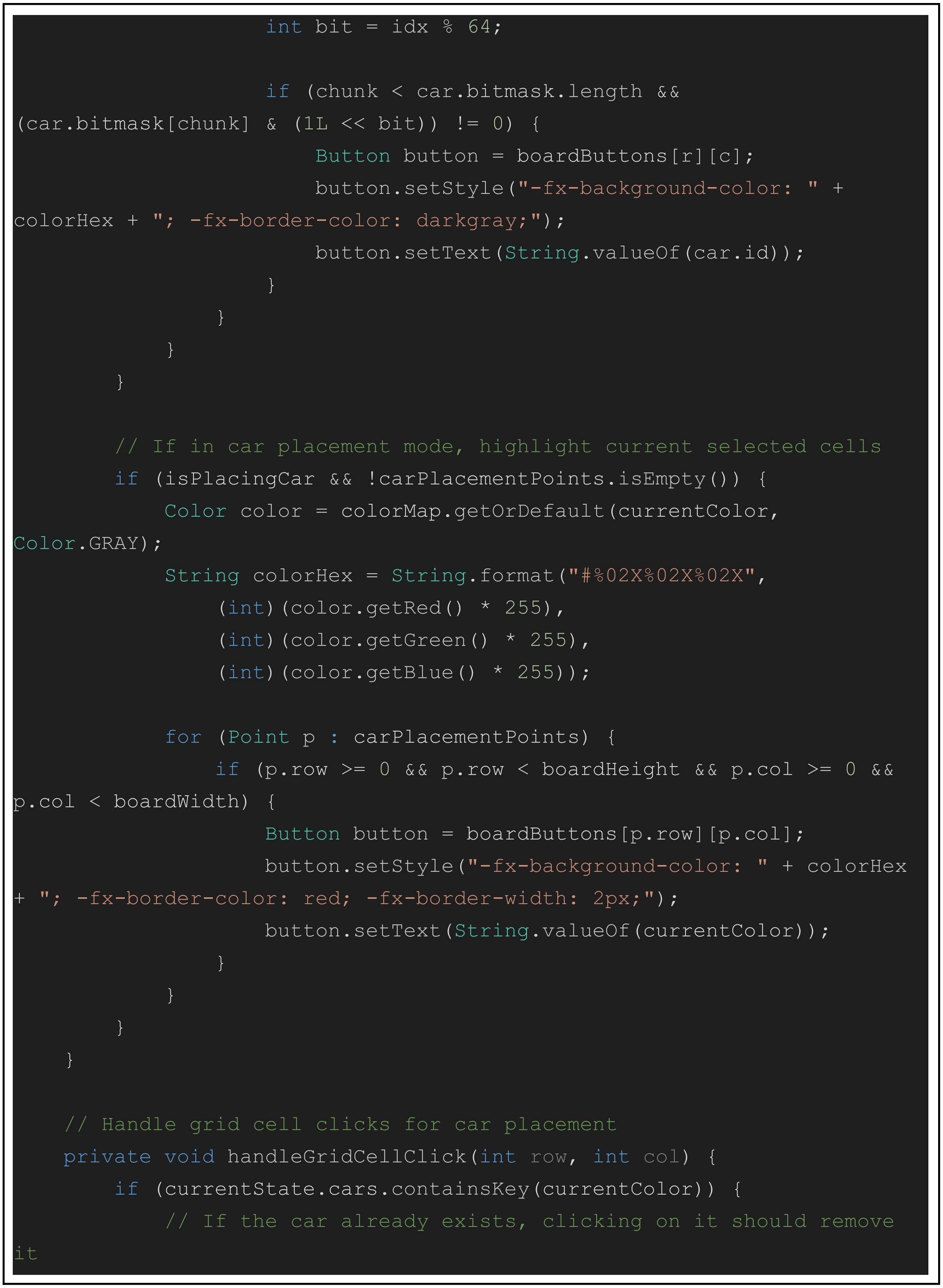
```
                    currentColor = 'P';

                    // Reset solution components
                    solutionStates.clear();
                    currentStepIndex = 0;
                    lblStep.setText("Step: 0/0");
                    btnPrev.setDisable(true);
                    btnPlay.setDisable(true);
                    btnNext.setDisable(true);
                    btnSaveResultToText.setDisable(true);

                    // Reset metrics
                    lblVisitedNodes.setText("Nodes Visited: 0");
                    lblExecutionTime.setText("Execution Time: 0.000
seconds");

                    // Initialize board with loaded state
                    initializeBoard();
                    updateColorLegend(colorLegend);

                    // Show success message
                    Alert alert = new
Alert(Alert.AlertType.INFORMATION);
                    alert.setTitle("File Loaded");
                    alert.setHeaderText("Puzzle configuration loaded");
                    alert.setContentText("The puzzle has been loaded
from " + file.getName());
                    alert.showAndWait();

                } catch (IOException ex) {
                    Alert alert = new Alert(Alert.AlertType.ERROR);
                    alert.setTitle("Load Error");
                    alert.setHeaderText("Could not load puzzle");
                    alert.setContentText("Error: " + ex.getMessage());
                    alert.showAndWait();
                }
            }
        });
```



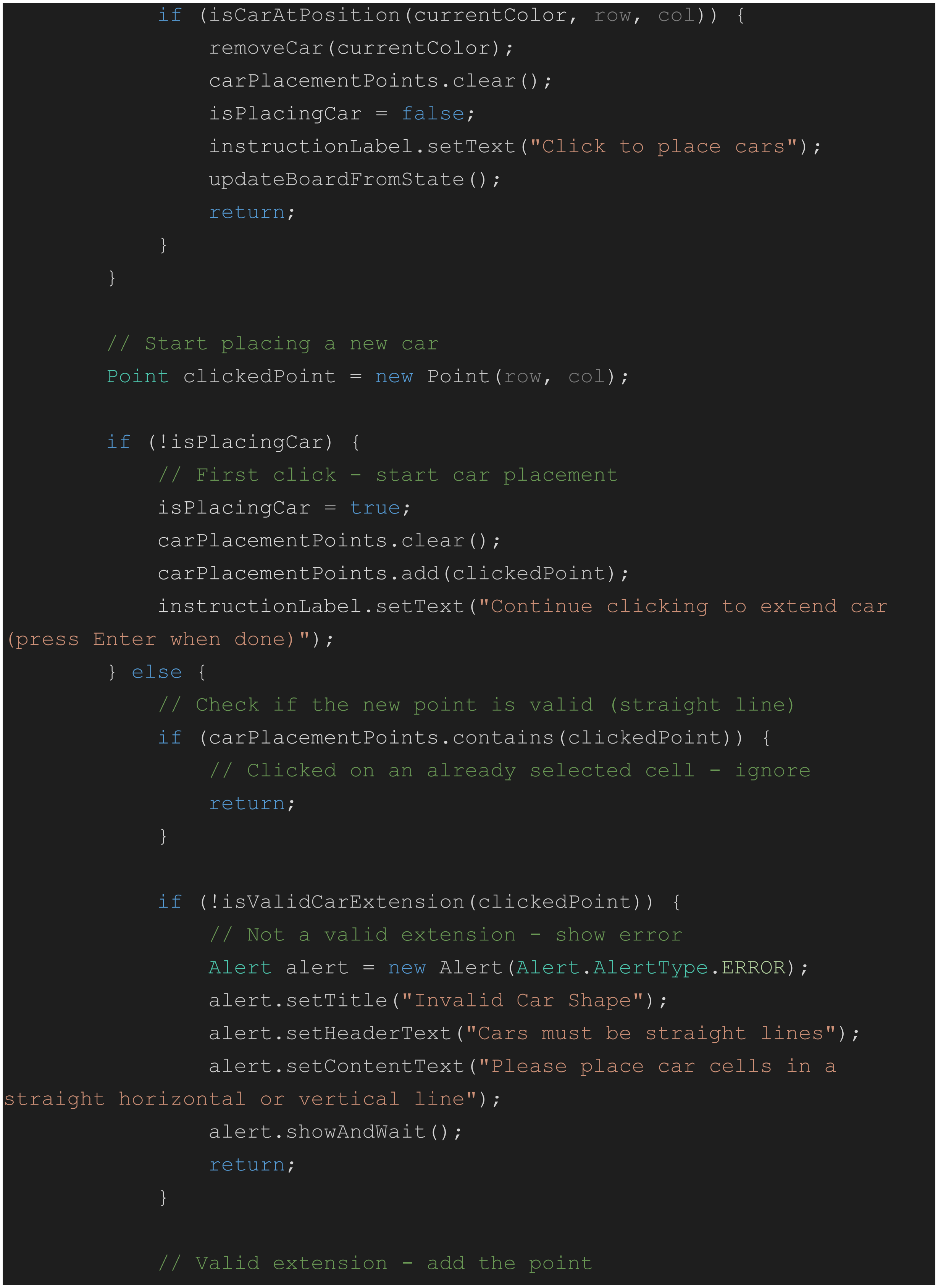
```
        // Save button
        btnSaveResultToText.setOnAction(e -> {
            if (solutionStates.isEmpty()) {
                Alert alert = new Alert(Alert.AlertType.WARNING);
                alert.setTitle("No Solution");
                alert.setHeaderText("No solution to save");
                alert.setContentText("Please solve the puzzle first.");
                alert.showAndWait();
                return;
            }

            FileChooser fileChooser = new FileChooser();
            fileChooser.setTitle("Save Solution");
            fileChooser.getExtensionFilters().add(
                new FileChooser.ExtensionFilter("Text Files", "*.txt")
            );

            File file = fileChooser.showSaveDialog(primaryStage);
            if (file != null) {
                try (FileWriter writer = new FileWriter(file)) {
                    // Write solution information
                    writer.write("Rush Hour Puzzle Solution\n");
                    writer.write("=========================\n\n");
                    writer.write(String.format("Board size: %d x %d\n", 
boardWidth, boardHeight));
                    writer.write(String.format("Exit direction: %s\n", 
exitDirection.toUpperCase()));
                    writer.write(String.format("Algorithm used: %s\n", 
selectedAlgorithm));
                    if (!selectedAlgorithm.equals("Uniform Cost Search 
(UCS)")) {
                        writer.write(String.format("Heuristic used: 
%s\n", selectedHeuristic.getName()));
                    }
                    writer.write(String.format("Total steps: %d\n\n", 
solutionStates.size() - 1));

                    // Write move history
                    writer.write("Solution moves:\n");
                    State lastState =
solutionStates.get(solutionStates.size() - 1);
                    List<String> moves = lastState.getMoveHistory();
                    for (int i = 0; i < moves.size(); i++) {
                        writer.write(String.format("%2d. %s\n", i + 1,
moves.get(i)));
                    }

                    Alert alert = new
Alert(Alert.AlertType.INFORMATION);
                    alert.setTitle("Save Successful");
                    alert.setHeaderText("Solution saved");
                    alert.setContentText("The solution has been saved to
" + file.getAbsolutePath());
                    alert.showAndWait();

                } catch (IOException ex) {
                    Alert alert = new Alert(Alert.AlertType.ERROR);
                    alert.setTitle("Save Error");
                    alert.setHeaderText("Could not save solution");
                    alert.setContentText("Error: " + ex.getMessage());
                    alert.showAndWait();
                }
            }
        });

        // Initialize the board
        initializeBoard();
        updateColorLegend(colorLegend);

        Scene scene = new Scene(root, 900, 600);
        primaryStage.setScene(scene);
        primaryStage.centerOnScreen();
        primaryStage.show();
    }

    private void initializeColorMap() {
        // Setup fixed colors for pieces
        colorMap.put('P', Color.rgb(220, 20, 60)); // This line sets the primary car color to red

        // Regular pieces A-Z
        colorMap.put('A', Color.GREEN);
        colorMap.put('B', Color.BLUE);
        colorMap.put('C', Color.YELLOW);
        colorMap.put('D', Color.MAGENTA);
        colorMap.put('E', Color.CYAN);
        colorMap.put('F', Color.ORANGERED);
        colorMap.put('G', Color.DARKGREEN);
        colorMap.put('H', Color.DARKBLUE);
        colorMap.put('I', Color.GOLD);
        colorMap.put('J', Color.PURPLE);
        colorMap.put('K', Color.DARKTURQUOISE);
        colorMap.put('L', Color.DARKORANGE);

        // Generate additional colors if needed
        for (char c = 'M'; c <= 'Z'; c++) {
            double hue = ((c - 'M') / (double)('Z' - 'M')) * 360;
            Color color = Color.hsb(hue, 0.8, 0.9);
            colorMap.put(c, color);
        }
    }

    private void initializeEmptyState() {
        Map<Character, Car> cars = new HashMap<>();

        int totalBits = boardWidth * boardHeight;
        int chunkCount = (totalBits + 63) / 64;

        // Create primary car
        long[] primaryBitmask = new long[chunkCount];

        if (isPrimaryHorizontal) {
            // Place a 2-cell horizontal primary car in the middle row
            int row = boardHeight / 2;
            int col = 1;  // Start near the left
            primaryBitmask[0] |= (1L << (row * boardWidth + col));
            primaryBitmask[0] |= (1L << (row * boardWidth + col + 1));

            cars.put('P', new Car('P', true, 2, primaryBitmask, -1,
row));
        } else {
            // Place a 2-cell vertical primary car in the middle column
            int col = boardWidth / 2;
            int row = 1;  // Start near the top

            primaryBitmask[0] |= (1L << (row * boardWidth + col));
            primaryBitmask[0] |= (1L << ((row + 1) * boardWidth + col));

            cars.put('P', new Car('P', false, 2, primaryBitmask, col,
-1));
        }

        currentState = new State(cars, null, "", 0);
    }

    private void initializeBoard() {
        boardPane.getChildren().clear();
        boardButtons = new Button[boardHeight][boardWidth];

        // Calculate button size based on board dimensions
        double buttonSize = calculateButtonSize();

        for (int row = 0; row < boardHeight; row++) {
            for (int col = 0; col < boardWidth; col++) {
                Button button = new Button();
                button.setMinSize(buttonSize, buttonSize);
                button.setPrefSize(buttonSize, buttonSize);
                button.setMaxSize(buttonSize, buttonSize);

                // Style for empty cell (no green indicator)
                button.setStyle("-fx-background-color: white;
-fx-border-color: lightgray;");

                // Store row and column in the button properties
```



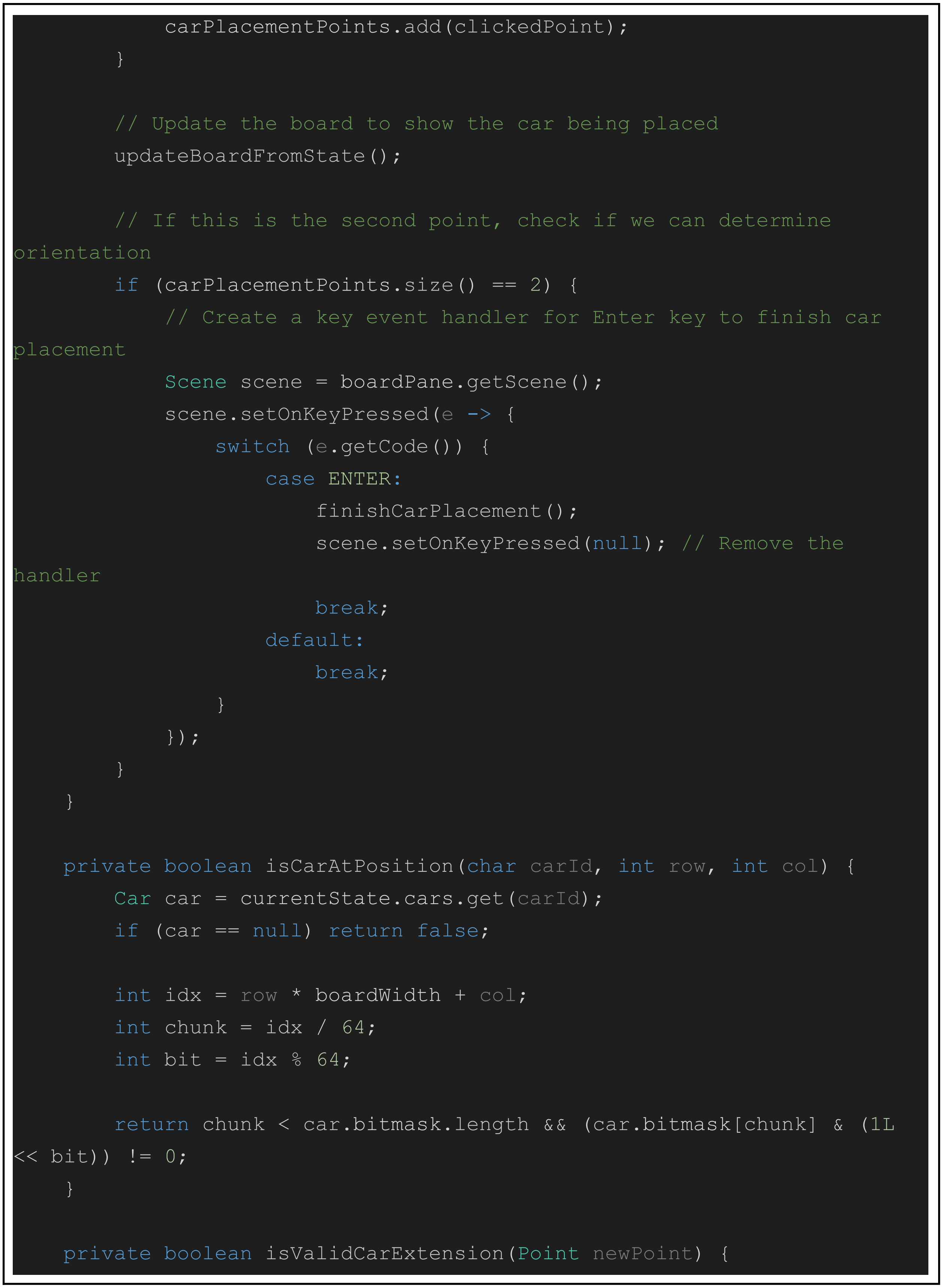
```
                final int r = row;
                final int c = col;

                // Add click handler for car placement
                button.setOnAction(event -> handleGridCellClick(r, c));

                boardButtons[row][col] = button;
                boardPane.add(button, col, row);
            }
        }

        // Draw existing cars
        updateBoardFromState();
    }

    private void updateBoardFromState() {
        // Reset board
        for (int row = 0; row < boardHeight; row++) {
            for (int col = 0; col < boardWidth; col++) {
                Button button = boardButtons[row][col];
                button.setText("");
                button.setStyle("-fx-background-color: white;
-fx-border-color: lightgray;");
            }
        }

        // Draw all cars
        for (Car car : currentState.cars.values()) {
            Color color = colorMap.getOrDefault(car.id, Color.GRAY);
            String colorHex = String.format("#%02X%02X%02X",
                (int)(color.getRed() * 255),
                (int)(color.getGreen() * 255),
                (int)(color.getBlue() * 255));

            // Find all occupied cells for this car
            for (int r = 0; r < boardHeight; r++) {
                for (int c = 0; c < boardWidth; c++) {
                    int idx = r * boardWidth + c;
                    int chunk = idx / 64;
```



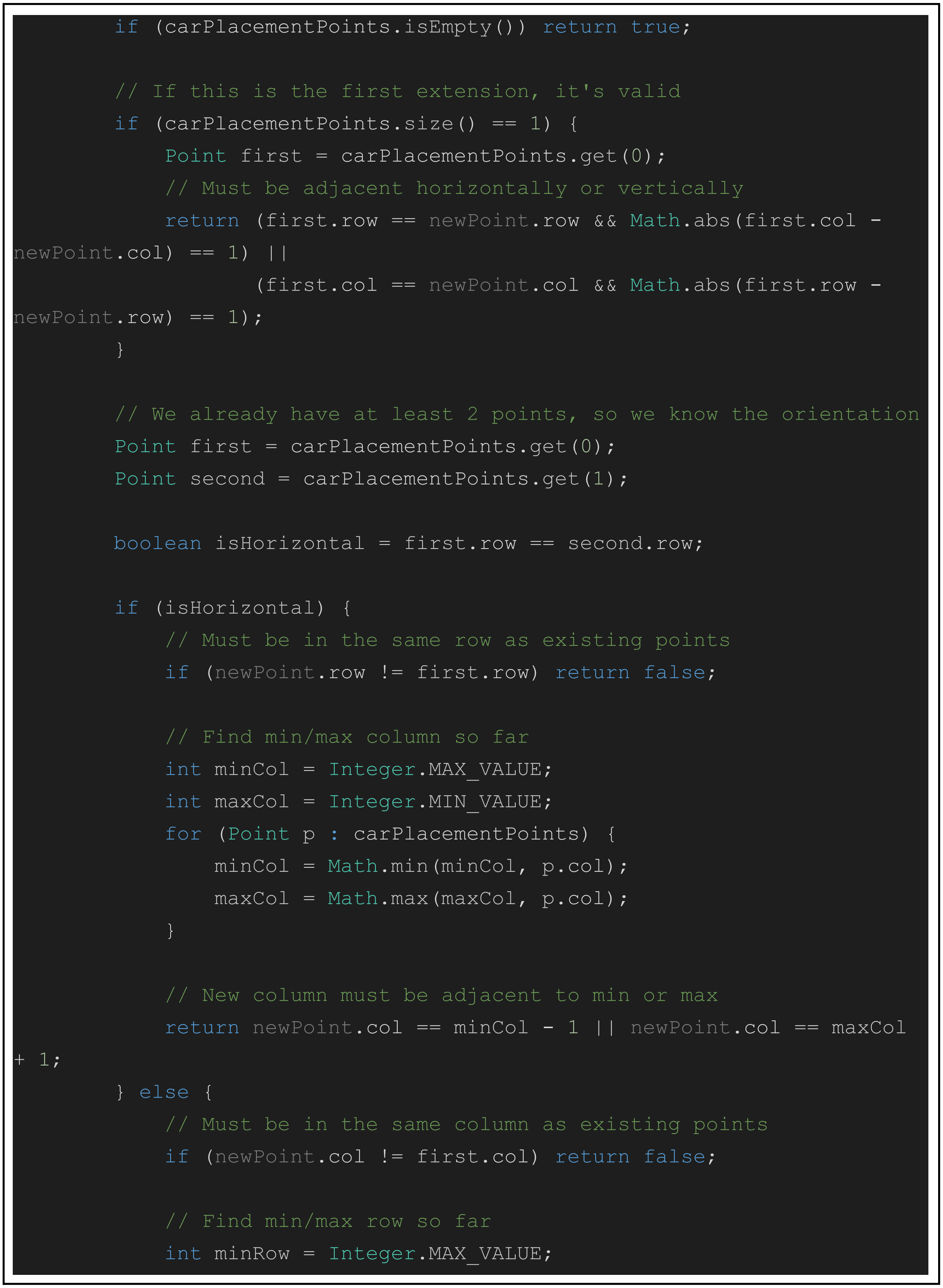
```
                    int bit = idx % 64;

                    if (chunk < car.bitmask.length &&
(car.bitmask[chunk] & (1L << bit)) != 0) {
                        Button button = boardButtons[r][c];
                        button.setStyle("-fx-background-color: " +
colorHex + "; -fx-border-color: darkgray;");
                        button.setText(String.valueOf(car.id));
                    }
                }
            }
        }

        // If in car placement mode, highlight current selected cells
        if (isPlacingCar && !carPlacementPoints.isEmpty()) {
            Color color = colorMap.getOrDefault(currentColor,
Color.GRAY);
            String colorHex = String.format("#%02X%02X%02X",
                (int)(color.getRed() * 255),
                (int)(color.getGreen() * 255),
                (int)(color.getBlue() * 255));

            for (Point p : carPlacementPoints) {
                if (p.row >= 0 && p.row < boardHeight && p.col >= 0 &&
p.col < boardWidth) {
                    Button button = boardButtons[p.row][p.col];
                    button.setStyle("-fx-background-color: " + colorHex
+ "; -fx-border-color: red; -fx-border-width: 2px;");
                    button.setText(String.valueOf(currentColor));
                }
            }
        }
    }

    // Handle grid cell clicks for car placement
    private void handleGridCellClick(int row, int col) {
        if (currentState.cars.containsKey(currentColor)) {
            // If the car already exists, clicking on it should remove
it
```



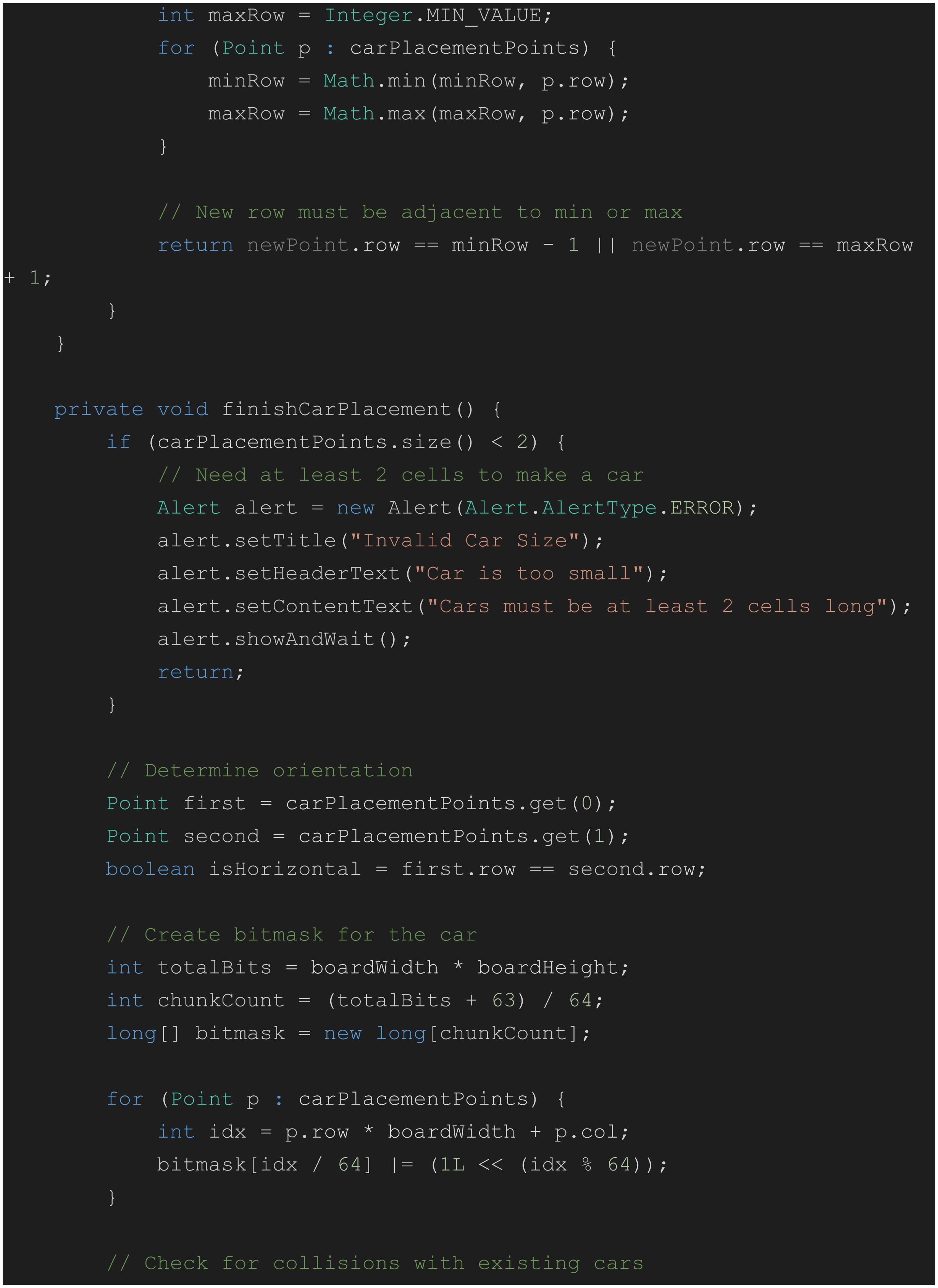
```
            if (isCarAtPosition(currentColor, row, col)) {
                removeCar(currentColor);
                carPlacementPoints.clear();
                isPlacingCar = false;
                instructionLabel.setText("Click to place cars");
                updateBoardFromState();
                return;
            }
        }

        // Start placing a new car
        Point clickedPoint = new Point(row, col);

        if (!isPlacingCar) {
            // First click - start car placement
            isPlacingCar = true;
            carPlacementPoints.clear();
            carPlacementPoints.add(clickedPoint);
            instructionLabel.setText("Continue clicking to extend car
(press Enter when done)");
        } else {
            // Check if the new point is valid (straight line)
            if (carPlacementPoints.contains(clickedPoint)) {
                // Clicked on an already selected cell - ignore
                return;
            }

            if (!isValidCarExtension(clickedPoint)) {
                // Not a valid extension - show error
                Alert alert = new Alert(Alert.AlertType.ERROR);
                alert.setTitle("Invalid Car Shape");
                alert.setHeaderText("Cars must be straight lines");
                alert.setContentText("Please place car cells in a
straight horizontal or vertical line");
                alert.showAndWait();
                return;
            }

            // Valid extension - add the point
```



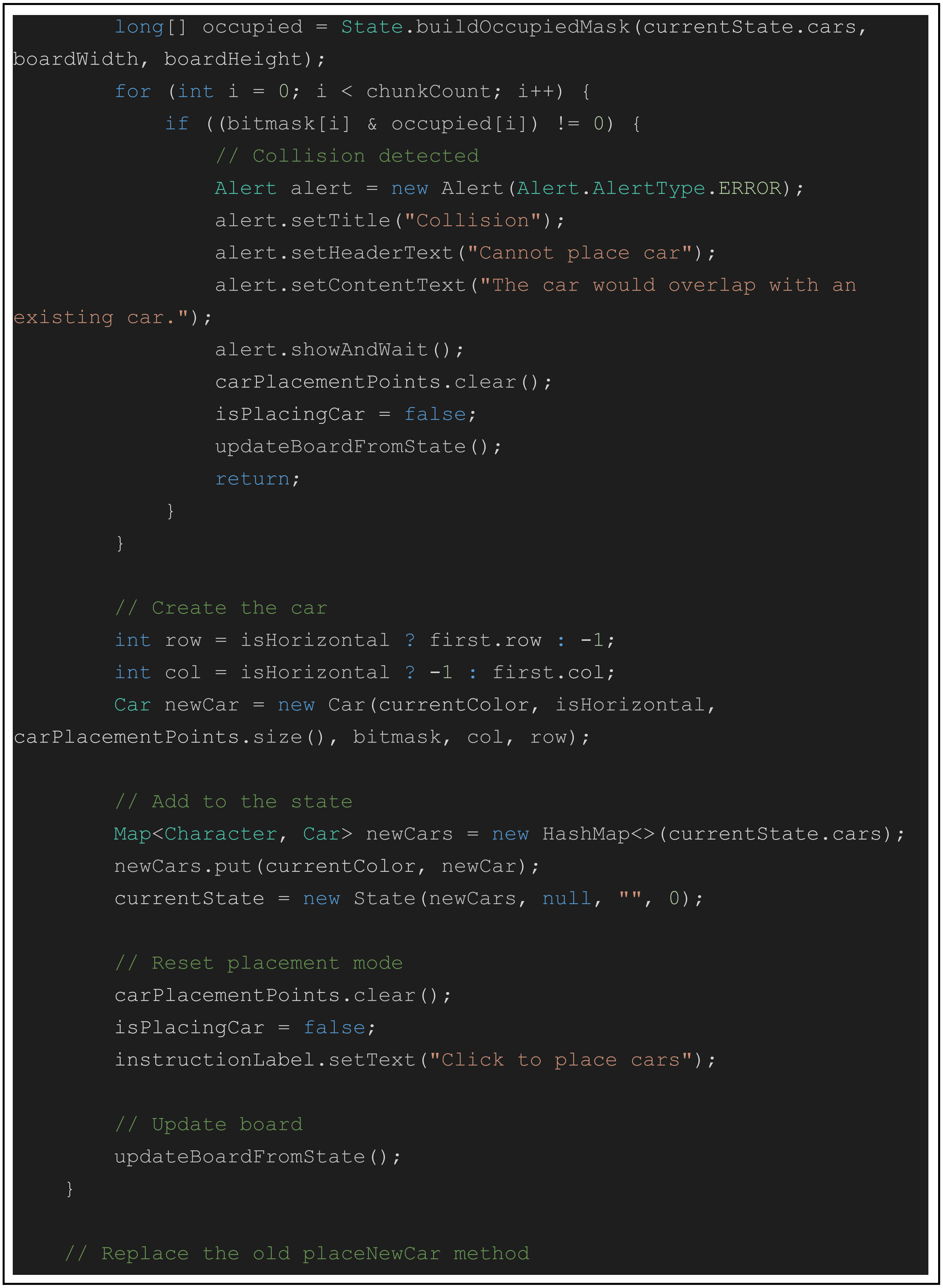
```
            carPlacementPoints.add(clickedPoint);
        }

        // Update the board to show the car being placed
        updateBoardFromState();

        // If this is the second point, check if we can determine
orientation
        if (carPlacementPoints.size() == 2) {
            // Create a key event handler for Enter key to finish car
placement
            Scene scene = boardPane.getScene();
            scene.setOnKeyPressed(e -> {
                switch (e.getCode()) {
                    case ENTER:
                        finishCarPlacement();
                        scene.setOnKeyPressed(null); // Remove the
handler
                        break;
                    default:
                        break;
                }
            });
        }
    }

    private boolean isCarAtPosition(char carId, int row, int col) {
        Car car = currentState.cars.get(carId);
        if (car == null) return false;

        int idx = row * boardWidth + col;
        int chunk = idx / 64;
        int bit = idx % 64;

        return chunk < car.bitmask.length && (car.bitmask[chunk] & (1L
<< bit)) != 0;
    }

    private boolean isValidCarExtension(Point newPoint) {
```



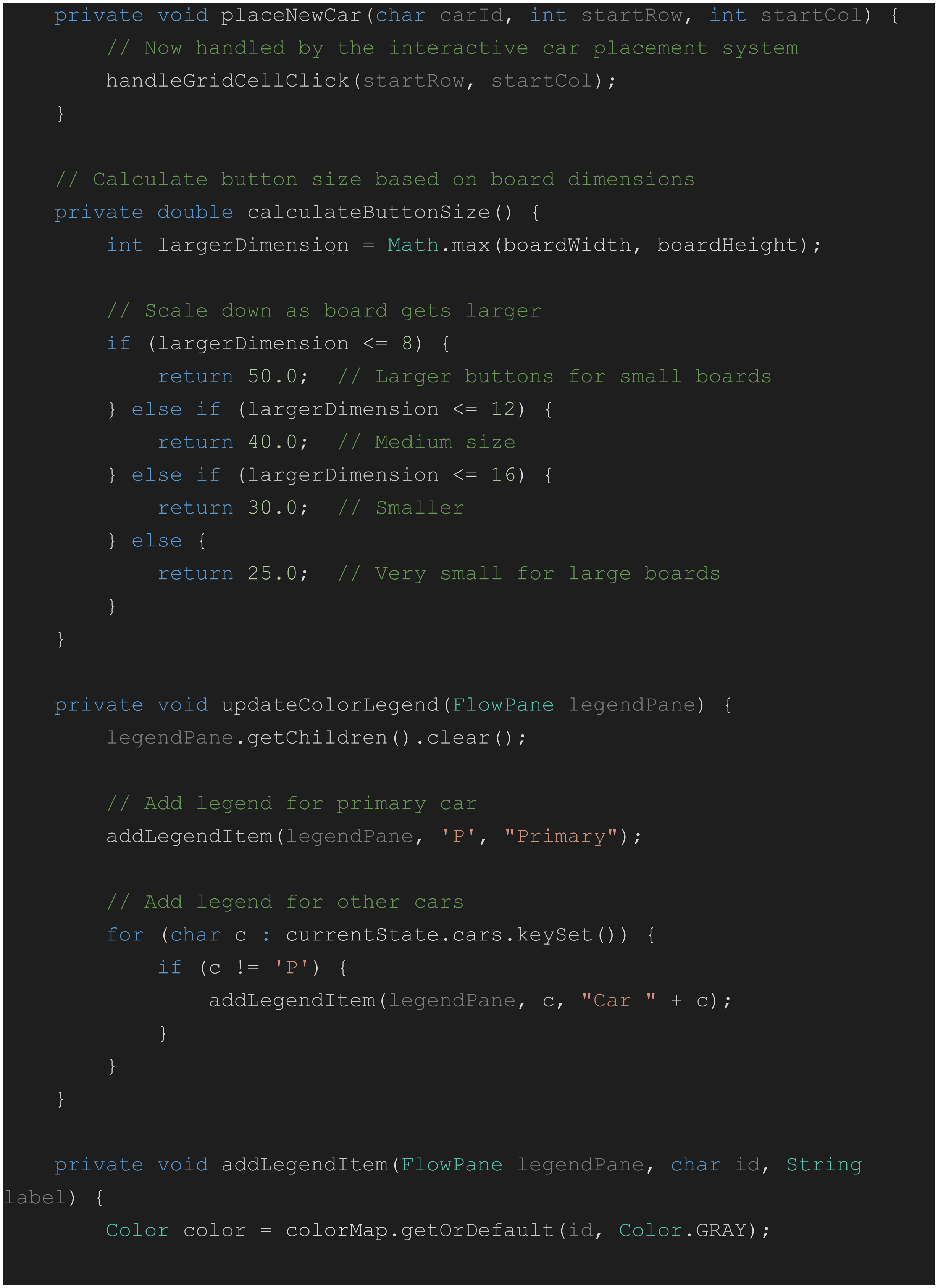
```
        if (carPlacementPoints.isEmpty()) return true;

        // If this is the first extension, it's valid
        if (carPlacementPoints.size() == 1) {
            Point first = carPlacementPoints.get(0);
            // Must be adjacent horizontally or vertically
            return (first.row == newPoint.row && Math.abs(first.col -
newPoint.col) == 1) ||
                   (first.col == newPoint.col && Math.abs(first.row -
newPoint.row) == 1);
        }

        // We already have at least 2 points, so we know the orientation
        Point first = carPlacementPoints.get(0);
        Point second = carPlacementPoints.get(1);

        boolean isHorizontal = first.row == second.row;

        if (isHorizontal) {
            // Must be in the same row as existing points
            if (newPoint.row != first.row) return false;

            // Find min/max column so far
            int minCol = Integer.MAX_VALUE;
            int maxCol = Integer.MIN_VALUE;
            for (Point p : carPlacementPoints) {
                minCol = Math.min(minCol, p.col);
                maxCol = Math.max(maxCol, p.col);
            }

            // New column must be adjacent to min or max
            return newPoint.col == minCol - 1 || newPoint.col == maxCol
+ 1;
        } else {
            // Must be in the same column as existing points
            if (newPoint.col != first.col) return false;

            // Find min/max row so far
            int minRow = Integer.MAX_VALUE;
```



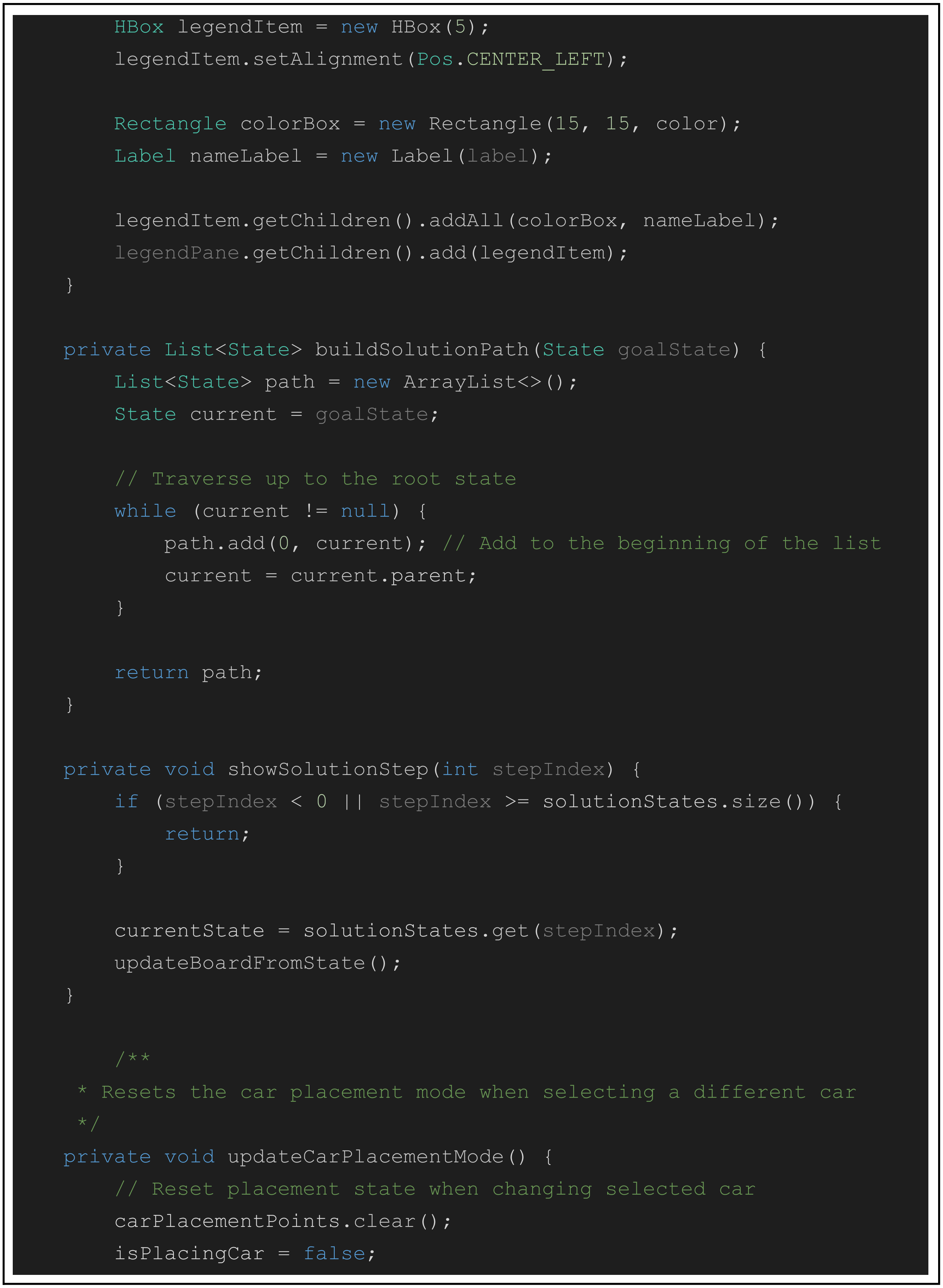
```
            int maxRow = Integer.MIN_VALUE;
            for (Point p : carPlacementPoints) {
                minRow = Math.min(minRow, p.row);
                maxRow = Math.max(maxRow, p.row);
            }

            // New row must be adjacent to min or max
            return newPoint.row == minRow - 1 || newPoint.row == maxRow
+ 1;
        }
    }

    private void finishCarPlacement() {
        if (carPlacementPoints.size() < 2) {
            // Need at least 2 cells to make a car
            Alert alert = new Alert(Alert.AlertType.ERROR);
            alert.setTitle("Invalid Car Size");
            alert.setHeaderText("Car is too small");
            alert.setContentText("Cars must be at least 2 cells long");
            alert.showAndWait();
            return;
        }

        // Determine orientation
        Point first = carPlacementPoints.get(0);
        Point second = carPlacementPoints.get(1);
        boolean isHorizontal = first.row == second.row;

        // Create bitmask for the car
        int totalBits = boardWidth * boardHeight;
        int chunkCount = (totalBits + 63) / 64;
        long[] bitmask = new long[chunkCount];

        for (Point p : carPlacementPoints) {
            int idx = p.row * boardWidth + p.col;
            bitmask[idx / 64] |= (1L << (idx % 64));
        }

        // Check for collisions with existing cars
```



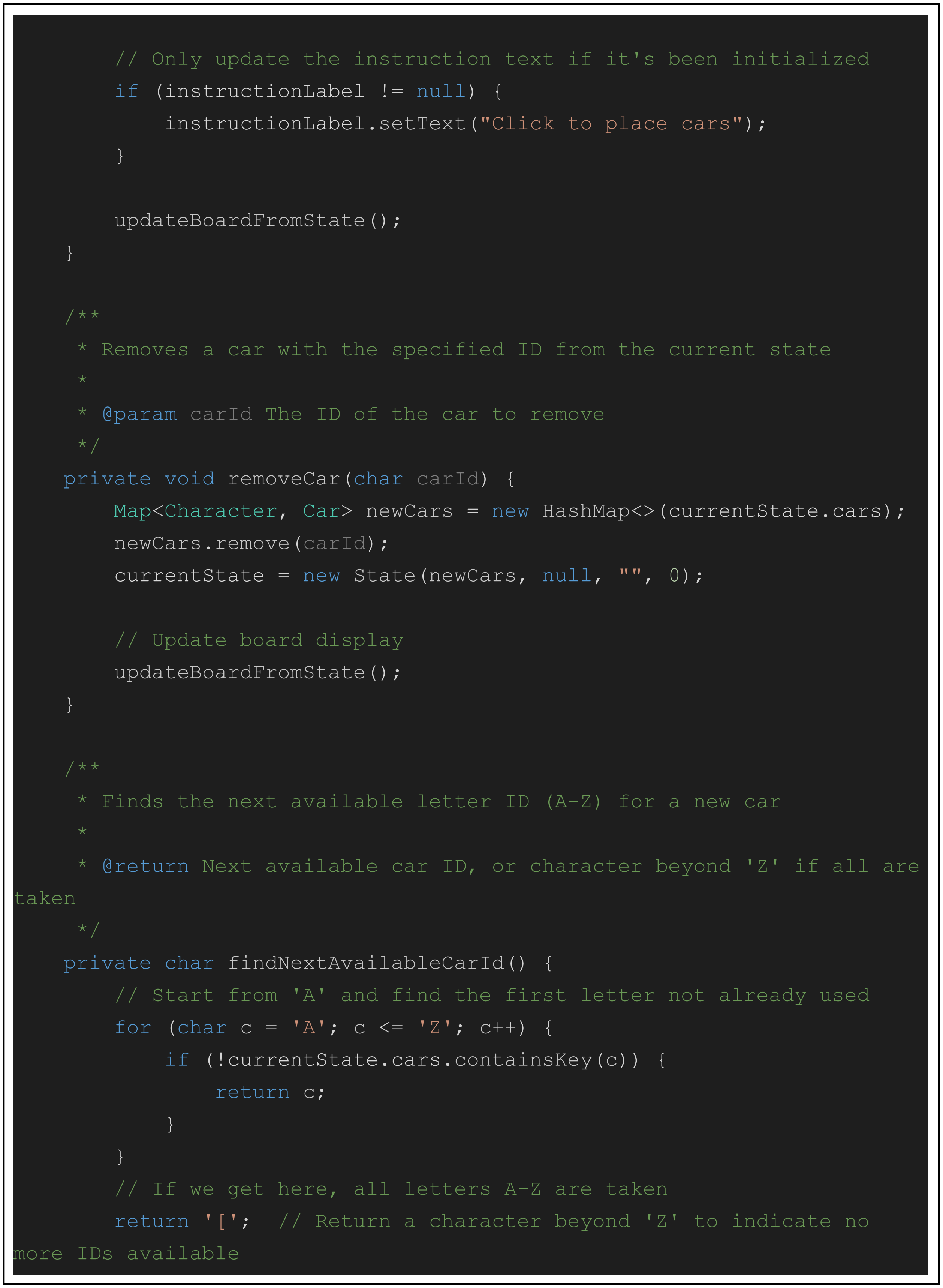
```
        long[] occupied = State.buildOccupiedMask(currentState.cars,
boardWidth, boardHeight);
        for (int i = 0; i < chunkCount; i++) {
            if ((bitmask[i] & occupied[i]) != 0) {
                // Collision detected
                Alert alert = new Alert(Alert.AlertType.ERROR);
                alert.setTitle("Collision");
                alert.setHeaderText("Cannot place car");
                alert.setContentText("The car would overlap with an
existing car.");
                alert.showAndWait();
                carPlacementPoints.clear();
                isPlacingCar = false;
                updateBoardFromState();
                return;
            }
        }

        // Create the car
        int row = isHorizontal ? first.row : -1;
        int col = isHorizontal ? -1 : first.col;
        Car newCar = new Car(currentColor, isHorizontal,
carPlacementPoints.size(), bitmask, col, row);

        // Add to the state
        Map<Character, Car> newCars = new HashMap<>(currentState.cars);
        newCars.put(currentColor, newCar);
        currentState = new State(newCars, null, "", 0);

        // Reset placement mode
        carPlacementPoints.clear();
        isPlacingCar = false;
        instructionLabel.setText("Click to place cars");

        // Update board
        updateBoardFromState();
    }

    // Replace the old placeNewCar method
```



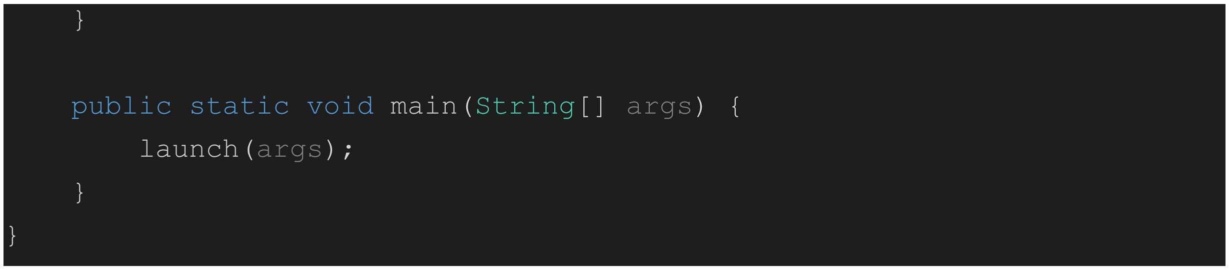
```
    private void placeNewCar(char carId, int startRow, int startCol) {
        // Now handled by the interactive car placement system
        handleGridCellClick(startRow, startCol);
    }

    // Calculate button size based on board dimensions
    private double calculateButtonSize() {
        int largerDimension = Math.max(boardWidth, boardHeight);

        // Scale down as board gets larger
        if (largerDimension <= 8) {
            return 50.0;  // Larger buttons for small boards
        } else if (largerDimension <= 12) {
            return 40.0;  // Medium size
        } else if (largerDimension <= 16) {
            return 30.0;  // Smaller
        } else {
            return 25.0;  // Very small for large boards
        }
    }

    private void updateColorLegend(FlowPane legendPane) {
        legendPane.getChildren().clear();

        // Add legend for primary car
        addLegendItem(legendPane, 'P', "Primary");

        // Add legend for other cars
        for (char c : currentState.cars.keySet()) {
            if (c != 'P') {
                addLegendItem(legendPane, c, "Car " + c);
            }
        }
    }

    private void addLegendItem(FlowPane legendPane, char id, String
label) {
        Color color = colorMap.getOrDefault(id, Color.GRAY);
```



```
        HBox legendItem = new HBox(5);
        legendItem.setAlignment(Pos.CENTER_LEFT);

        Rectangle colorBox = new Rectangle(15, 15, color);
        Label nameLabel = new Label(label);

        legendItem.getChildren().addAll(colorBox, nameLabel);
        legendPane.getChildren().add(legendItem);
    }

    private List<State> buildSolutionPath(State goalState) {
        List<State> path = new ArrayList<>();
        State current = goalState;

        // Traverse up to the root state
        while (current != null) {
            path.add(0, current); // Add to the beginning of the list
            current = current.parent;
        }

        return path;
    }

    private void showSolutionStep(int stepIndex) {
        if (stepIndex < 0 || stepIndex >= solutionStates.size()) {
            return;
        }

        currentState = solutionStates.get(stepIndex);
        updateBoardFromState();
    }

        /**
     * Resets the car placement mode when selecting a different car
     */
    private void updateCarPlacementMode() {
        // Reset placement state when changing selected car
        carPlacementPoints.clear();
        isPlacingCar = false;
```

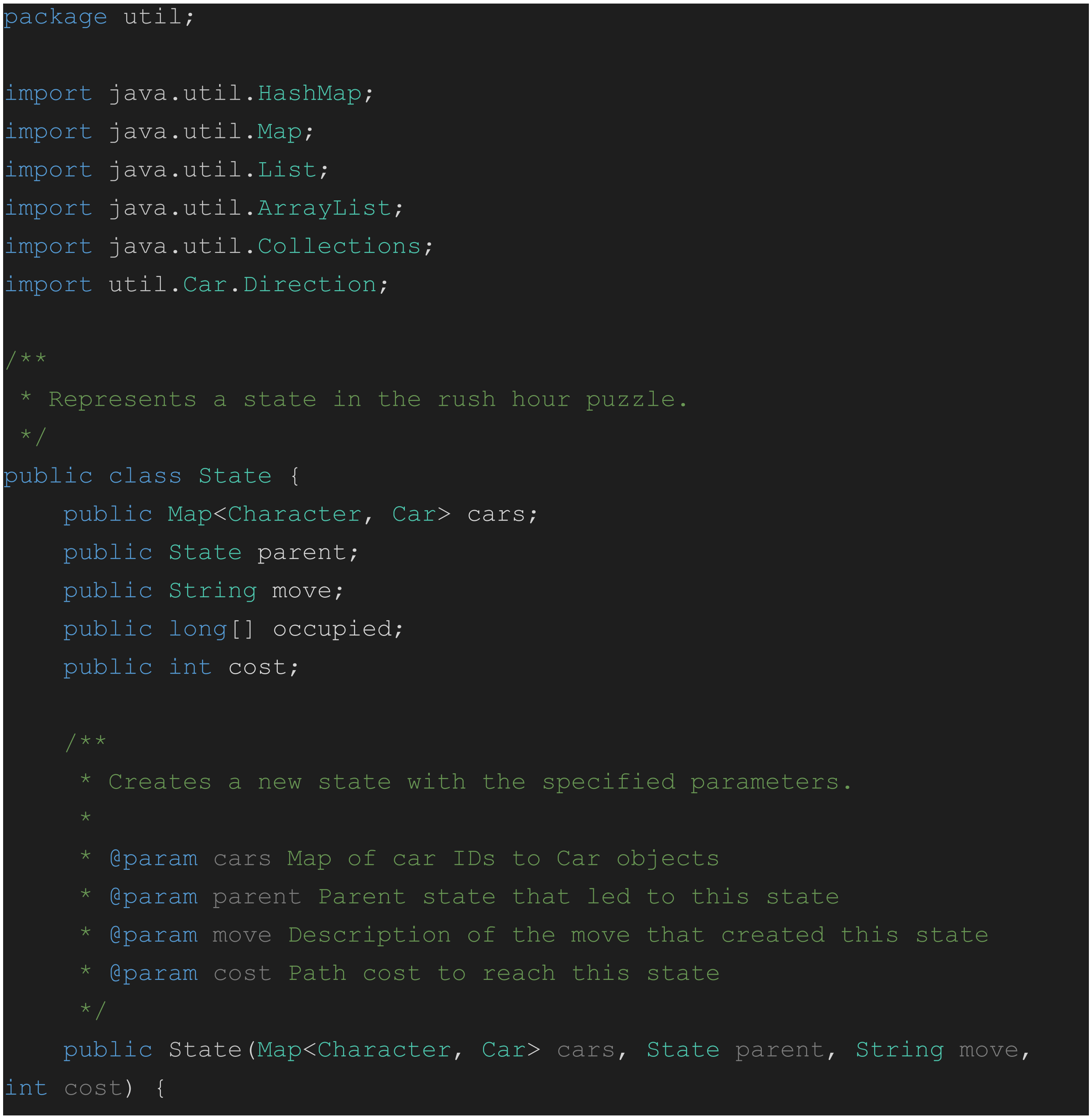


```
        // Only update the instruction text if it's been initialized
        if (instructionLabel != null) {
            instructionLabel.setText("Click to place cars");
        }

        updateBoardFromState();
    }

    /**
     * Removes a car with the specified ID from the current state
     *
     * @param carId The ID of the car to remove
     */
    private void removeCar(char carId) {
        Map<Character, Car> newCars = new HashMap<>(currentState.cars);
        newCars.remove(carId);
        currentState = new State(newCars, null, "", 0);

        // Update board display
        updateBoardFromState();
    }

    /**
     * Finds the next available letter ID (A-Z) for a new car
     *
     * @return Next available car ID, or character beyond 'Z' if all are
taken
     */
    private char findNextAvailableCarId() {
        // Start from 'A' and find the first letter not already used
        for (char c = 'A'; c <= 'Z'; c++) {
            if (!currentState.cars.containsKey(c)) {
                return c;
            }
        }
        // If we get here, all letters A-Z are taken
        return '[';  // Return a character beyond 'Z' to indicate no
more IDs available
```

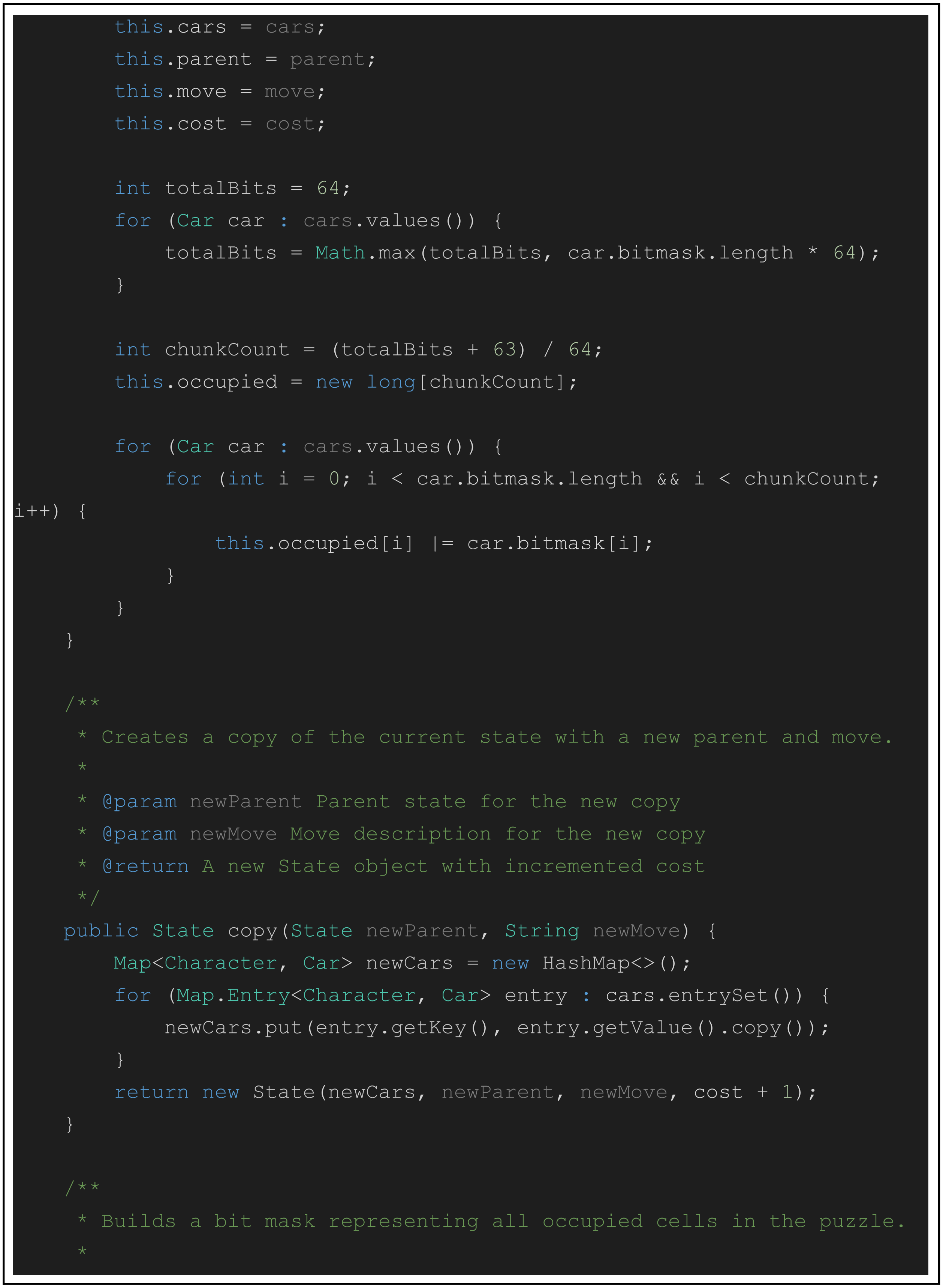


```
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

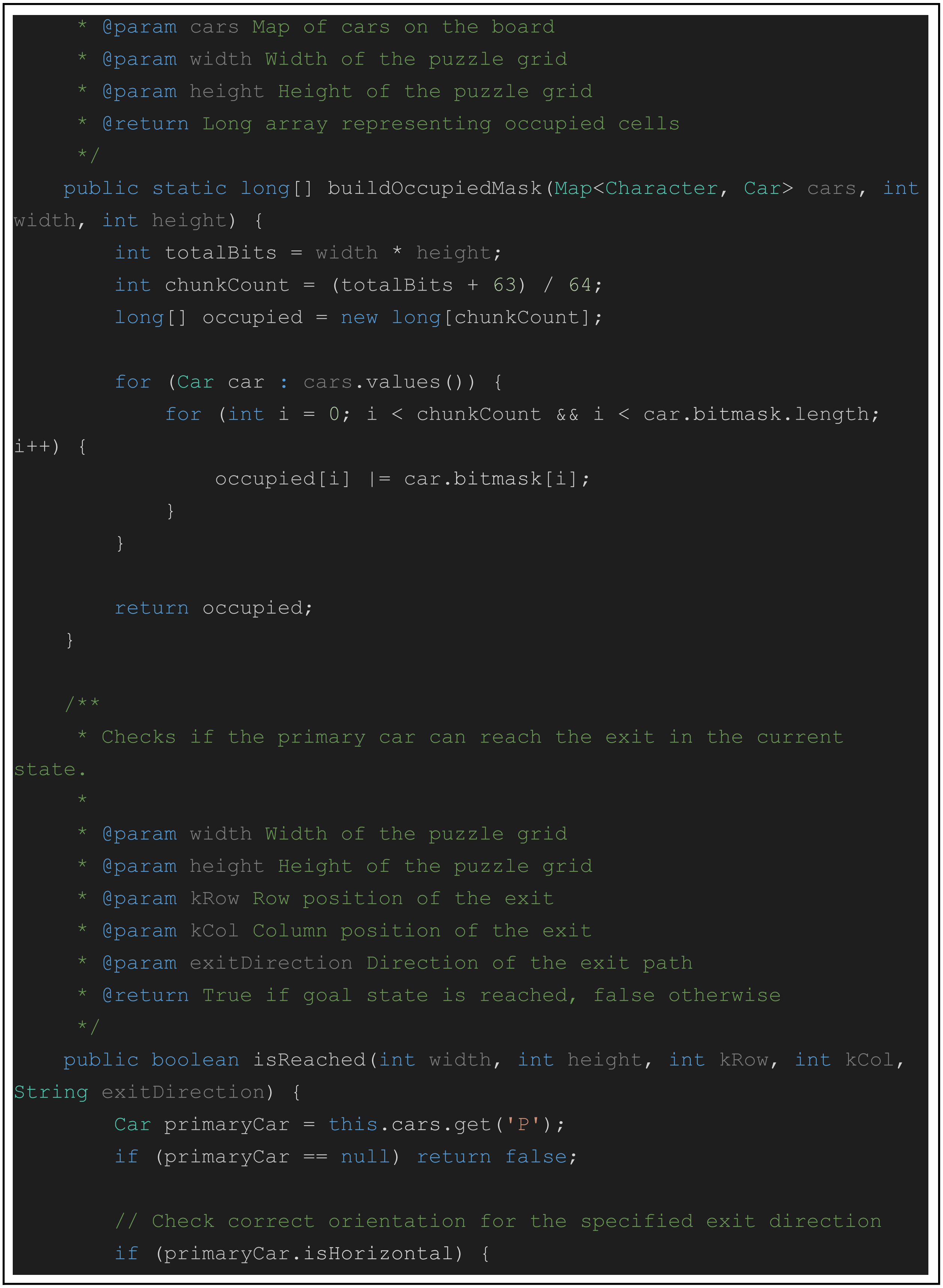
D. State



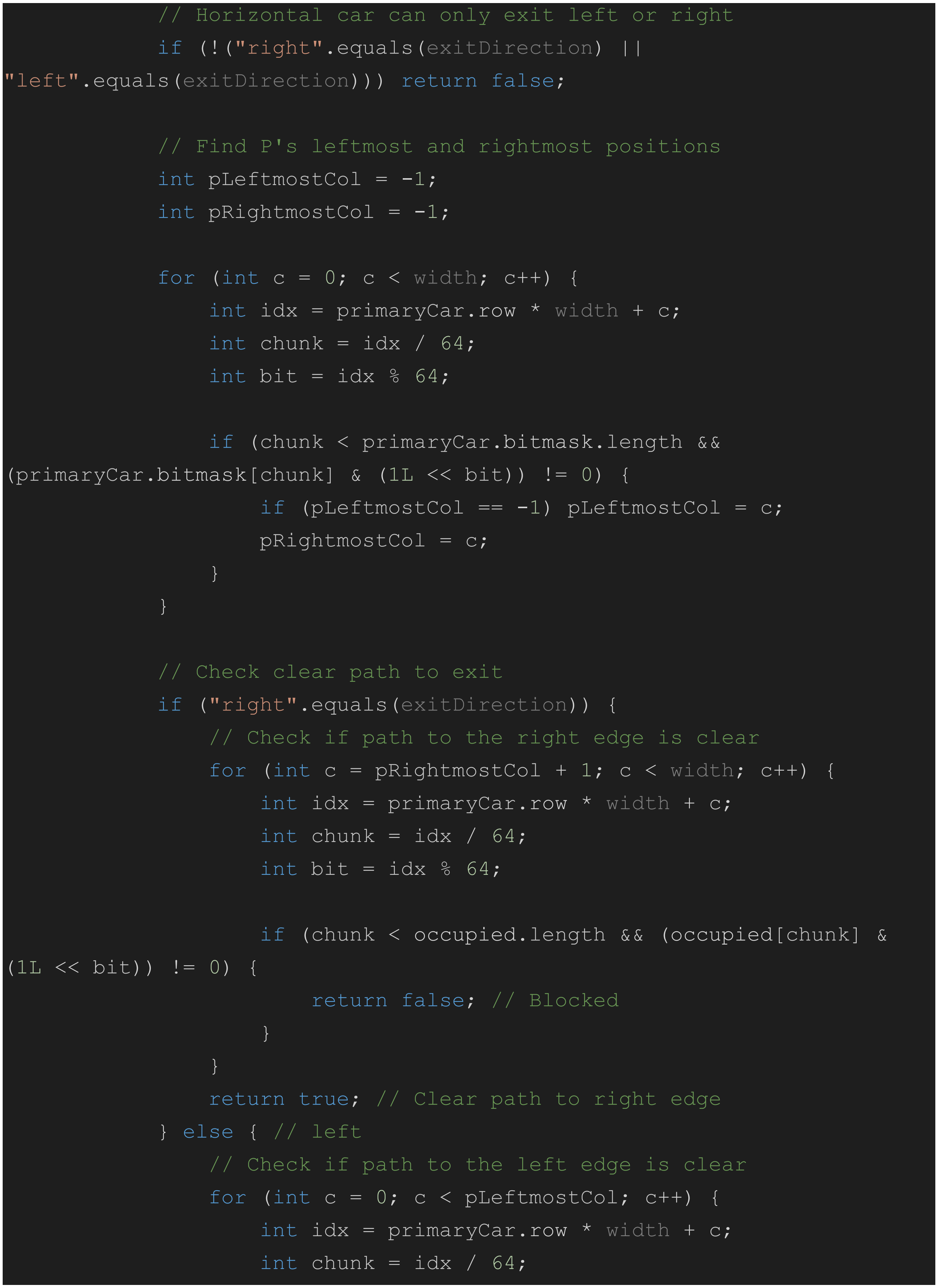
```
package util;

import java.util.HashMap;
import java.util.Map;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
import util.Car.Direction;

/**
 * Represents a state in the rush hour puzzle.
 */
public class State {
    public Map<Character, Car> cars;
    public State parent;
    public String move;
    public long[] occupied;
    public int cost;

    /**
     * Creates a new state with the specified parameters.
     *
     * @param cars Map of car IDs to Car objects
     * @param parent Parent state that led to this state
     * @param move Description of the move that created this state
     * @param cost Path cost to reach this state
     */
    public State(Map<Character, Car> cars, State parent, String move,
int cost) {
```



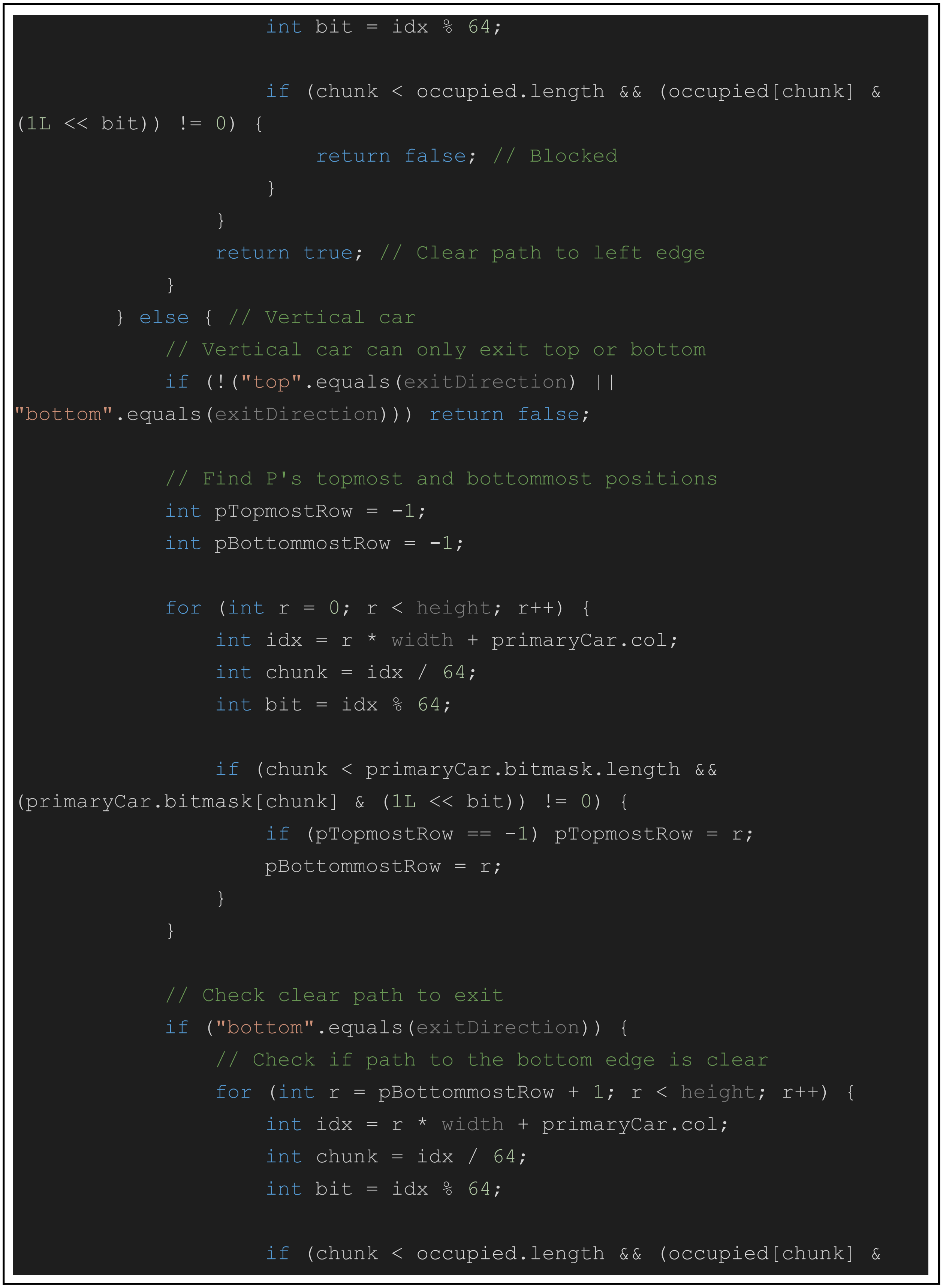
```
        this.cars = cars;
        this.parent = parent;
        this.move = move;
        this.cost = cost;

        int totalBits = 64;
        for (Car car : cars.values()) {
            totalBits = Math.max(totalBits, car.bitmask.length * 64);
        }

        int chunkCount = (totalBits + 63) / 64;
        this.occupied = new long[chunkCount];

        for (Car car : cars.values()) {
            for (int i = 0; i < car.bitmask.length && i < chunkCount;
i++) {
                this.occupied[i] |= car.bitmask[i];
            }
        }
    }

    /**
     * Creates a copy of the current state with a new parent and move.
     *
     * @param newParent Parent state for the new copy
     * @param newMove Move description for the new copy
     * @return A new State object with incremented cost
     */
    public State copy(State newParent, String newMove) {
        Map<Character, Car> newCars = new HashMap<>();
        for (Map.Entry<Character, Car> entry : cars.entrySet()) {
            newCars.put(entry.getKey(), entry.getValue().copy());
        }
        return new State(newCars, newParent, newMove, cost + 1);
    }

    /**
     * Builds a bit mask representing all occupied cells in the puzzle.
     *
```



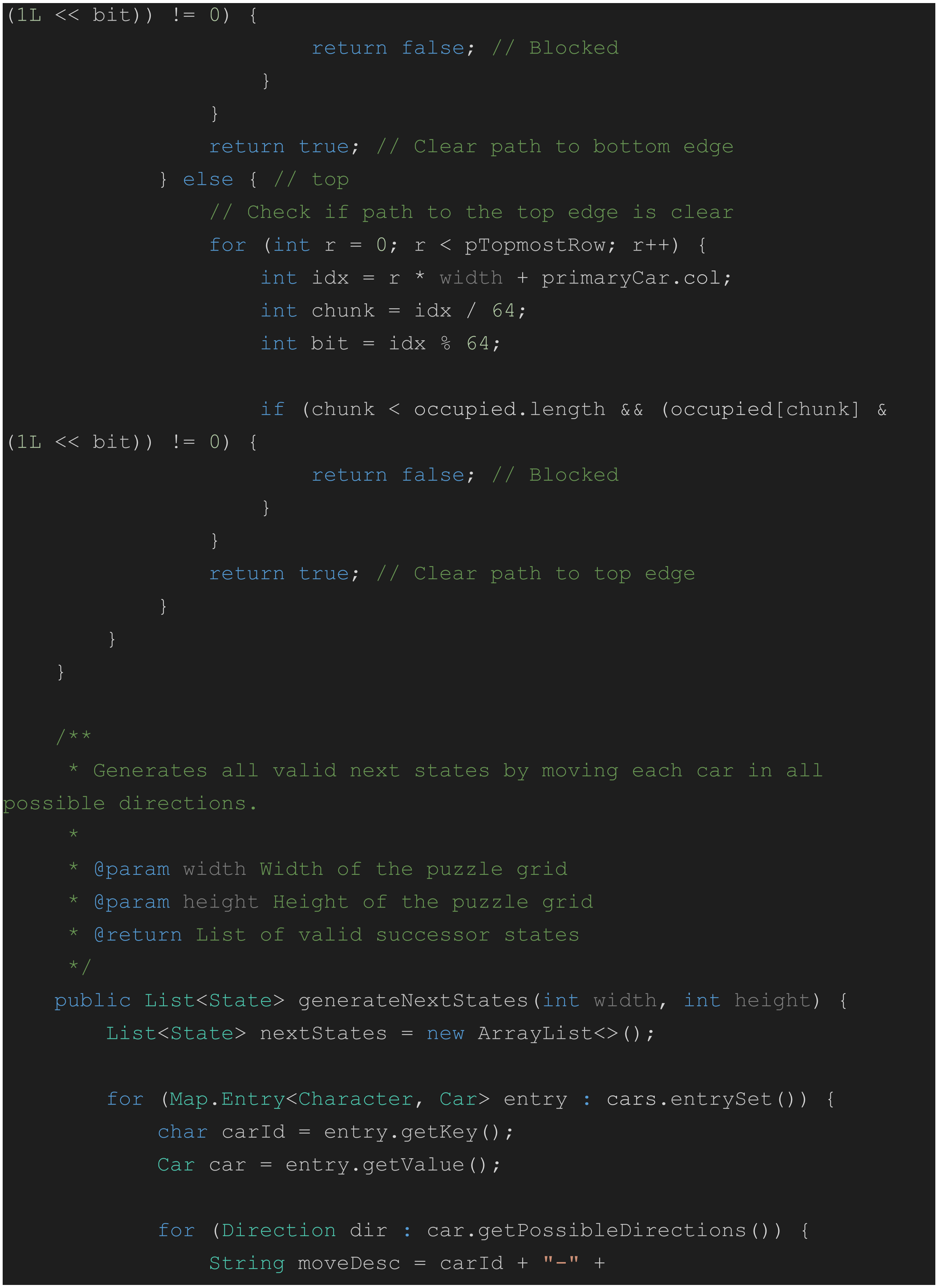
```java
     * @param cars Map of cars on the board
     * @param width Width of the puzzle grid
     * @param height Height of the puzzle grid
     * @return Long array representing occupied cells
     */
    public static long[] buildOccupiedMask(Map<Character, Car> cars, int width, int height) {
        int totalBits = width * height;
        int chunkCount = (totalBits + 63) / 64;
        long[] occupied = new long[chunkCount];

        for (Car car : cars.values()) {
            for (int i = 0; i < chunkCount && i < car.bitmask.length; i++) {
                occupied[i] |= car.bitmask[i];
            }
        }

        return occupied;
    }

    /**
     * Checks if the primary car can reach the exit in the current state.
     *
     * @param width Width of the puzzle grid
     * @param height Height of the puzzle grid
     * @param kRow Row position of the exit
     * @param kCol Column position of the exit
     * @param exitDirection Direction of the exit path
     * @return True if goal state is reached, false otherwise
     */
    public boolean isReached(int width, int height, int kRow, int kCol, String exitDirection) {
        Car primaryCar = this.cars.get('P');
        if (primaryCar == null) return false;

        // Check correct orientation for the specified exit direction
        if (primaryCar.isHorizontal) {
```



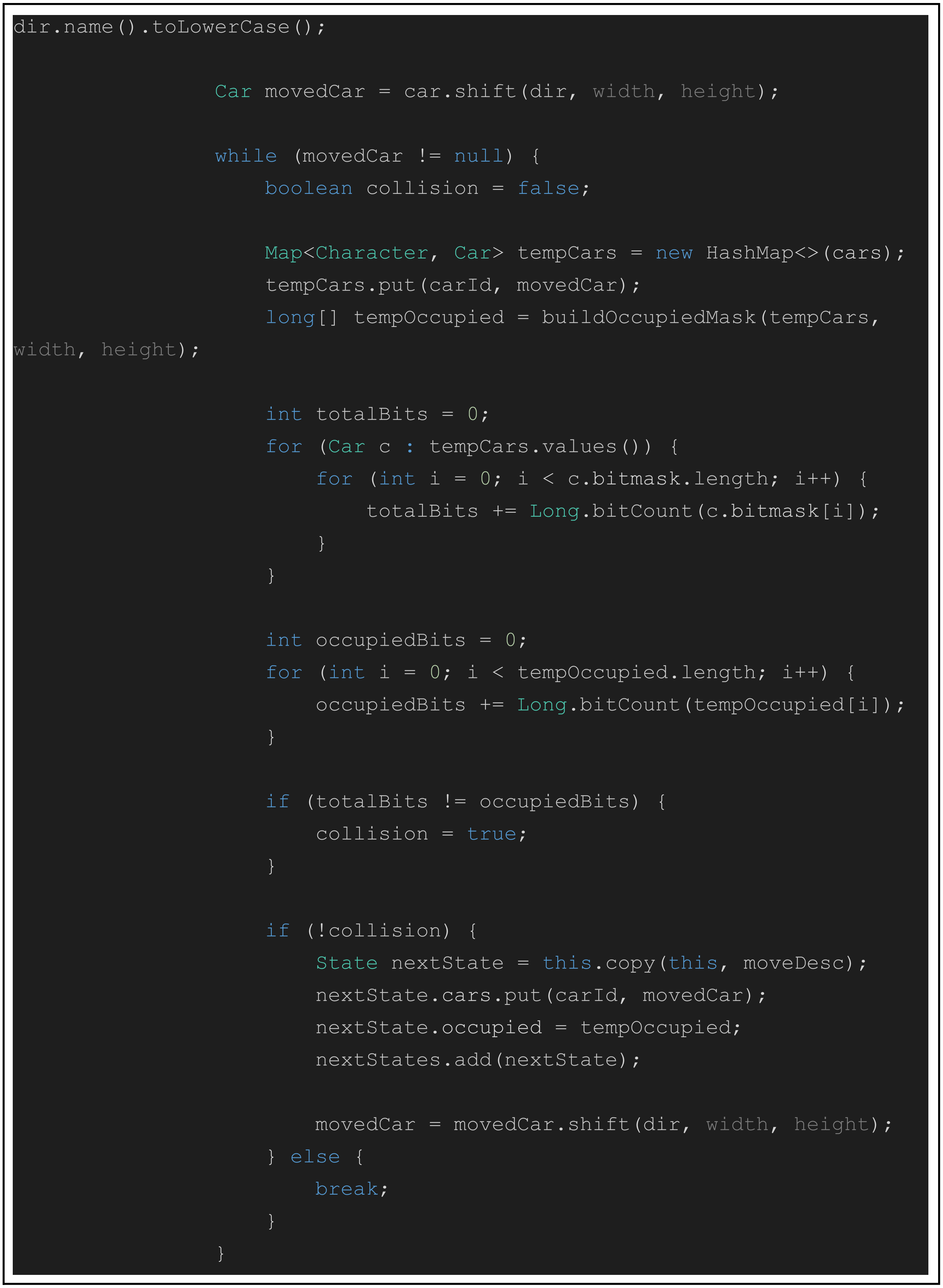
```
            // Horizontal car can only exit left or right
            if (!("right".equals(exitDirection) ||
"left".equals(exitDirection))) return false;

            // Find P's leftmost and rightmost positions
            int pLeftmostCol = -1;
            int pRightmostCol = -1;

            for (int c = 0; c < width; c++) {
                int idx = primaryCar.row * width + c;
                int chunk = idx / 64;
                int bit = idx % 64;

                if (chunk < primaryCar.bitmask.length &&
(primaryCar.bitmask[chunk] & (1L << bit)) != 0) {
                    if (pLeftmostCol == -1) pLeftmostCol = c;
                    pRightmostCol = c;
                }
            }

            // Check clear path to exit
            if ("right".equals(exitDirection)) {
                // Check if path to the right edge is clear
                for (int c = pRightmostCol + 1; c < width; c++) {
                    int idx = primaryCar.row * width + c;
                    int chunk = idx / 64;
                    int bit = idx % 64;

                    if (chunk < occupied.length && (occupied[chunk] &
(1L << bit)) != 0) {
                        return false; // Blocked
                    }
                }
                return true; // Clear path to right edge
            } else { // left
                // Check if path to the left edge is clear
                for (int c = 0; c < pLeftmostCol; c++) {
                    int idx = primaryCar.row * width + c;
                    int chunk = idx / 64;
```



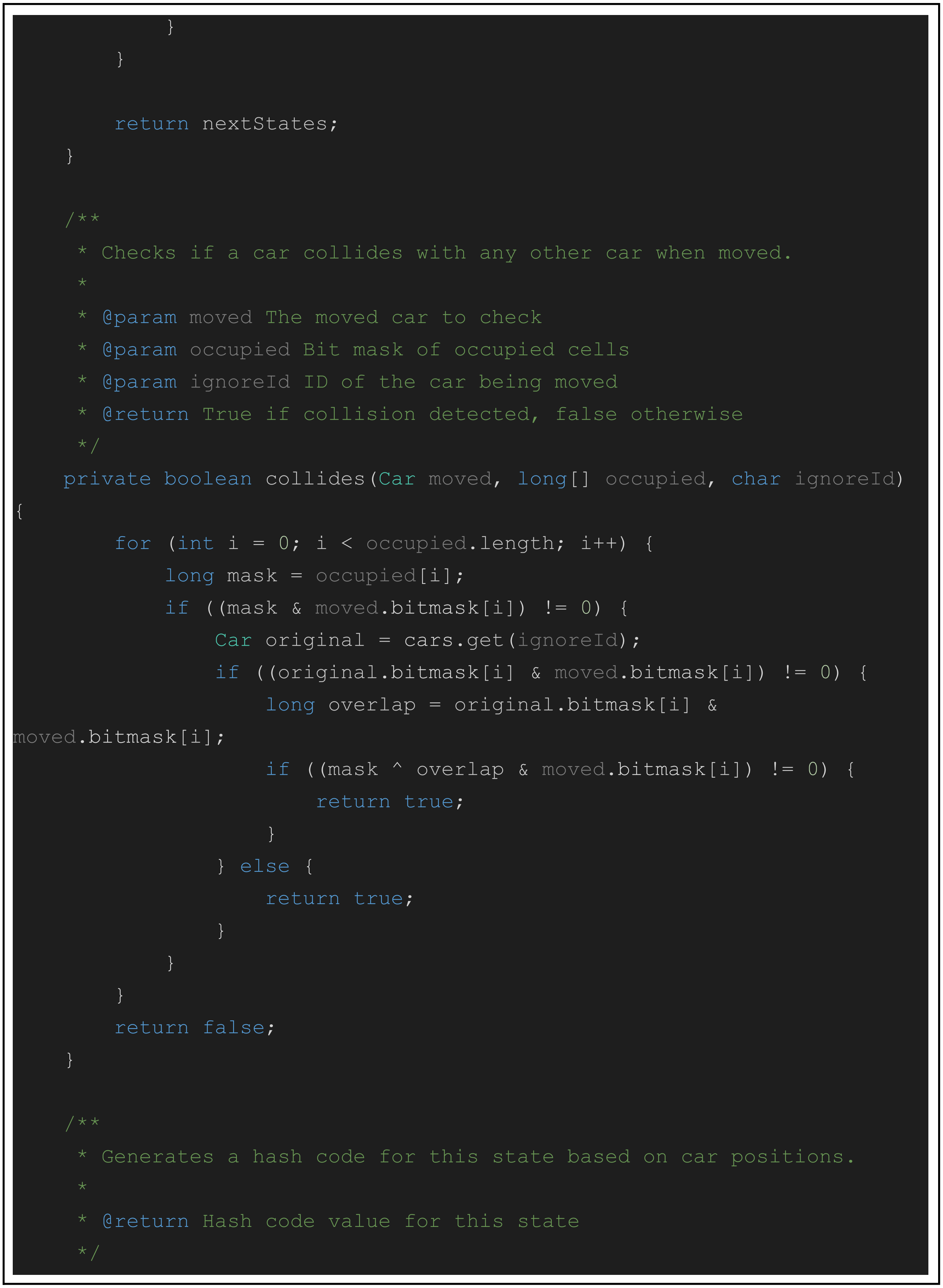
```
                    int bit = idx % 64;

                    if (chunk < occupied.length && (occupied[chunk] & 
(1L << bit)) != 0) {
                        return false; // Blocked
                    }
                }
                return true; // Clear path to left edge
            }
        } else { // Vertical car
            // Vertical car can only exit top or bottom
            if (!("top".equals(exitDirection) || 
"bottom".equals(exitDirection))) return false;

            // Find P's topmost and bottommost positions
            int pTopmostRow = -1;
            int pBottommostRow = -1;

            for (int r = 0; r < height; r++) {
                int idx = r * width + primaryCar.col;
                int chunk = idx / 64;
                int bit = idx % 64;

                if (chunk < primaryCar.bitmask.length && 
(primaryCar.bitmask[chunk] & (1L << bit)) != 0) {
                    if (pTopmostRow == -1) pTopmostRow = r;
                    pBottommostRow = r;
                }
            }

            // Check clear path to exit
            if ("bottom".equals(exitDirection)) {
                // Check if path to the bottom edge is clear
                for (int r = pBottommostRow + 1; r < height; r++) {
                    int idx = r * width + primaryCar.col;
                    int chunk = idx / 64;
                    int bit = idx % 64;

                    if (chunk < occupied.length && (occupied[chunk] & 
```



```
(1L << bit)) != 0) {
                        return false; // Blocked
                    }
                }
                return true; // Clear path to bottom edge
            } else { // top
                // Check if path to the top edge is clear
                for (int r = 0; r < pTopmostRow; r++) {
                    int idx = r * width + primaryCar.col;
                    int chunk = idx / 64;
                    int bit = idx % 64;

                    if (chunk < occupied.length && (occupied[chunk] &
(1L << bit)) != 0) {
                        return false; // Blocked
                    }
                }
                return true; // Clear path to top edge
            }
        }
    }

    /**
     * Generates all valid next states by moving each car in all
possible directions.
     *
     * @param width Width of the puzzle grid
     * @param height Height of the puzzle grid
     * @return List of valid successor states
     */
    public List<State> generateNextStates(int width, int height) {
        List<State> nextStates = new ArrayList<>();

        for (Map.Entry<Character, Car> entry : cars.entrySet()) {
            char carId = entry.getKey();
            Car car = entry.getValue();

            for (Direction dir : car.getPossibleDirections()) {
                String moveDesc = carId + "-" +
```



```
dir.name().toLowerCase();

                Car movedCar = car.shift(dir, width, height);

                while (movedCar != null) {
                    boolean collision = false;

                    Map<Character, Car> tempCars = new HashMap<>(cars);
                    tempCars.put(carId, movedCar);
                    long[] tempOccupied = buildOccupiedMask(tempCars,
width, height);

                    int totalBits = 0;
                    for (Car c : tempCars.values()) {
                        for (int i = 0; i < c.bitmask.length; i++) {
                            totalBits += Long.bitCount(c.bitmask[i]);
                        }
                    }

                    int occupiedBits = 0;
                    for (int i = 0; i < tempOccupied.length; i++) {
                        occupiedBits += Long.bitCount(tempOccupied[i]);
                    }

                    if (totalBits != occupiedBits) {
                        collision = true;
                    }

                    if (!collision) {
                        State nextState = this.copy(this, moveDesc);
                        nextState.cars.put(carId, movedCar);
                        nextState.occupied = tempOccupied;
                        nextStates.add(nextState);

                        movedCar = movedCar.shift(dir, width, height);
                    } else {
                        break;
                    }
                }
```

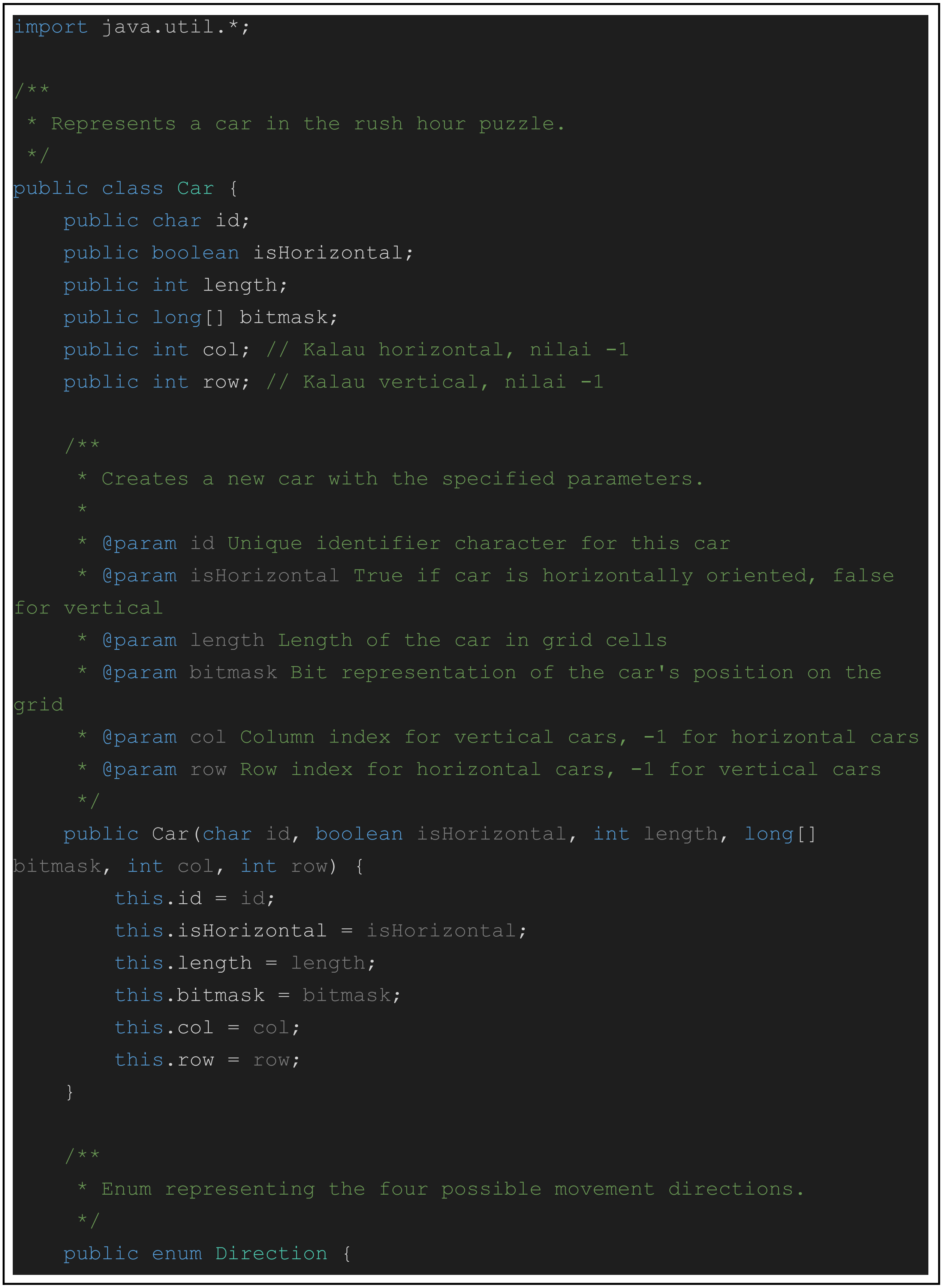


```
            }
        }

        return nextStates;
    }

    /**
     * Checks if a car collides with any other car when moved.
     *
     * @param moved The moved car to check
     * @param occupied Bit mask of occupied cells
     * @param ignoreId ID of the car being moved
     * @return True if collision detected, false otherwise
     */
    private boolean collides(Car moved, long[] occupied, char ignoreId)
{
        for (int i = 0; i < occupied.length; i++) {
            long mask = occupied[i];
            if ((mask & moved.bitmask[i]) != 0) {
                Car original = cars.get(ignoreId);
                if ((original.bitmask[i] & moved.bitmask[i]) != 0) {
                    long overlap = original.bitmask[i] &
moved.bitmask[i];
                    if ((mask ^ overlap & moved.bitmask[i]) != 0) {
                        return true;
                    }
                } else {
                    return true;
                }
            }
        }
        return false;
    }

    /**
     * Generates a hash code for this state based on car positions.
     *
     * @return Hash code value for this state
     */
```

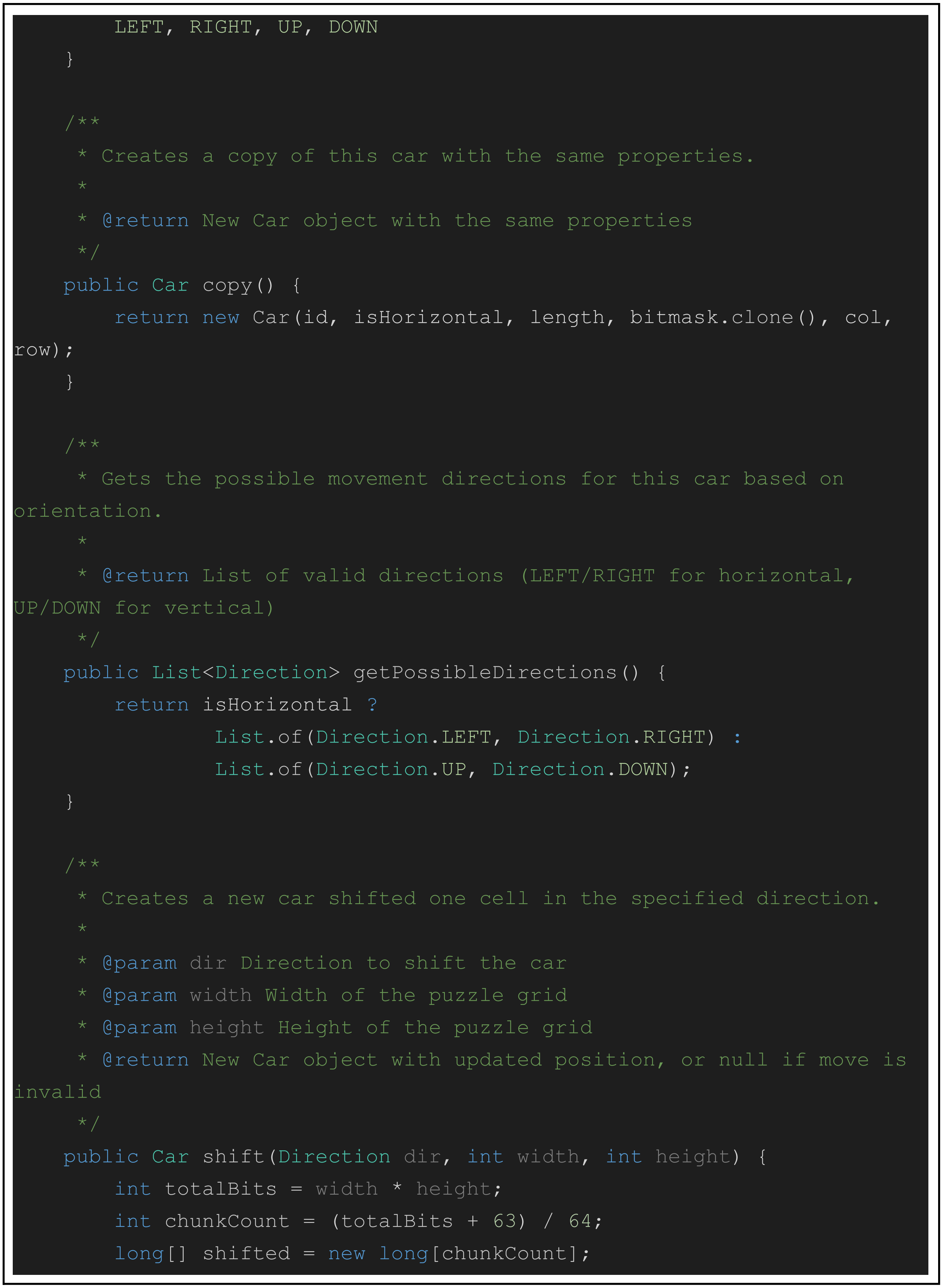
```java
    @Override
    public int hashCode() {
        return cars.hashCode();
    }

    /**
     * Compares this state with another state for equality.
     *
     * @param obj Object to compare with
     * @return True if states are equal, false otherwise
     */
    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof State other)) return false;
        return this.cars.equals(other.cars);
    }

    /**
     * Gets the complete history of moves that led to this state.
     *
     * @return List of move descriptions in order from initial state
     */
    public List<String> getMoveHistory() {
        List<String> moves = new ArrayList<>();
        State cur = this;
        while (cur != null && cur.move != null) {
            moves.add(cur.move);
            cur = cur.parent;
        }
        Collections.reverse(moves);
        return moves;
    }
}
```

E. Car

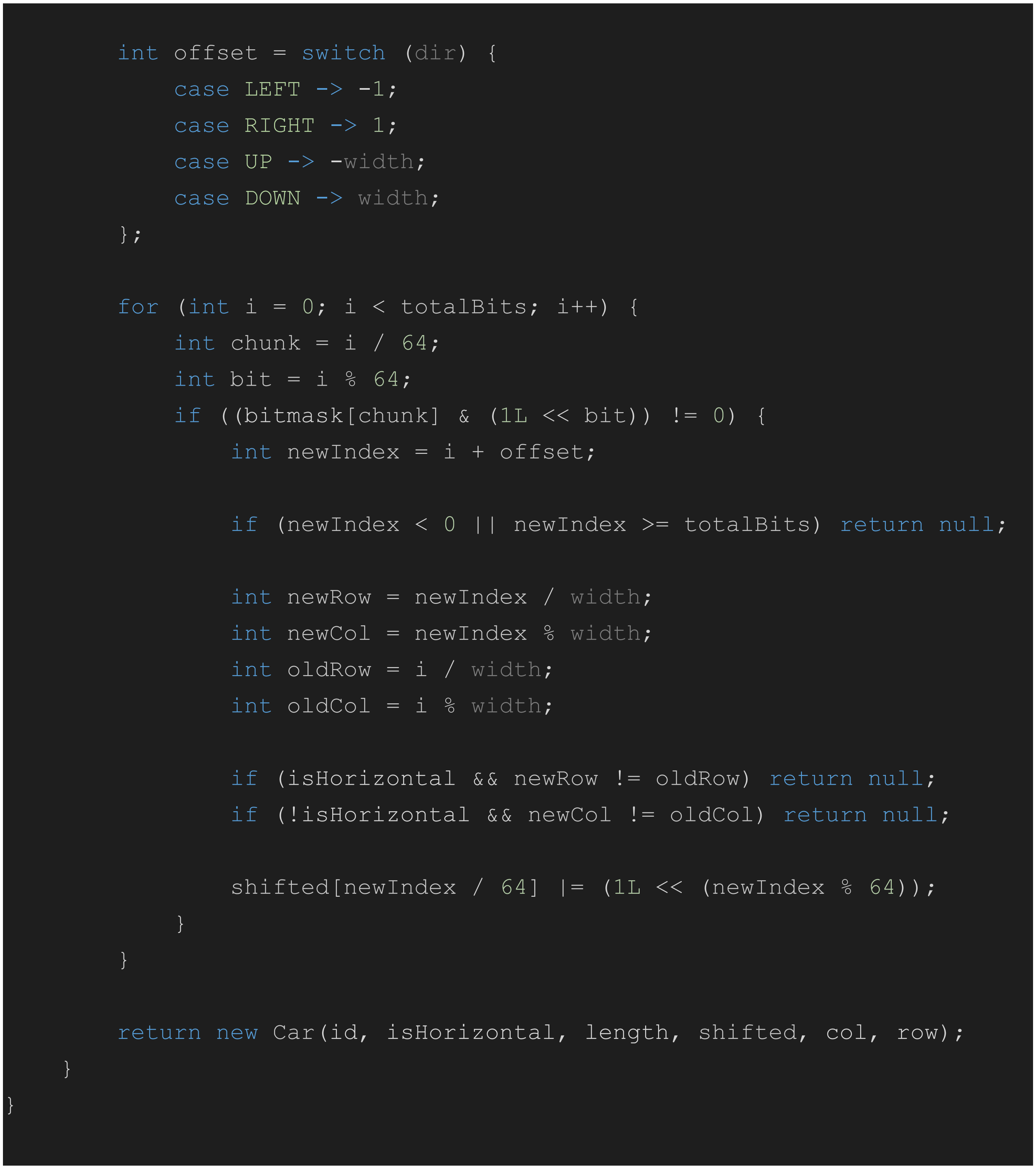
```java
package util;

```



```
import java.util.*;

/**
 * Represents a car in the rush hour puzzle.
 */
public class Car {
    public char id;
    public boolean isHorizontal;
    public int length;
    public long[] bitmask;
    public int col; // Kalau horizontal, nilai -1
    public int row; // Kalau vertical, nilai -1

    /**
     * Creates a new car with the specified parameters.
     *
     * @param id Unique identifier character for this car
     * @param isHorizontal True if car is horizontally oriented, false
for vertical
     * @param length Length of the car in grid cells
     * @param bitmask Bit representation of the car's position on the
grid
     * @param col Column index for vertical cars, -1 for horizontal cars
     * @param row Row index for horizontal cars, -1 for vertical cars
     */
    public Car(char id, boolean isHorizontal, int length, long[]
bitmask, int col, int row) {
        this.id = id;
        this.isHorizontal = isHorizontal;
        this.length = length;
        this.bitmask = bitmask;
        this.col = col;
        this.row = row;
    }

    /**
     * Enum representing the four possible movement directions.
     */
    public enum Direction {
```

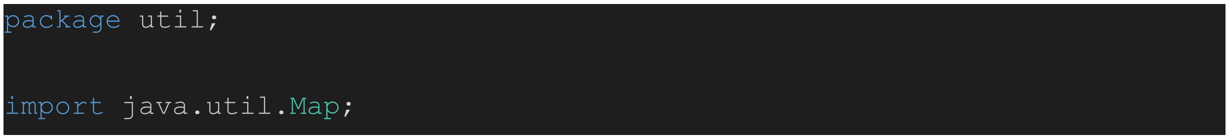


```
        LEFT, RIGHT, UP, DOWN
    }

    /**
     * Creates a copy of this car with the same properties.
     *
     * @return New Car object with the same properties
     */
    public Car copy() {
        return new Car(id, isHorizontal, length, bitmask.clone(), col, 
row);
    }

    /**
     * Gets the possible movement directions for this car based on 
orientation.
     *
     * @return List of valid directions (LEFT/RIGHT for horizontal, 
UP/DOWN for vertical)
     */
    public List<Direction> getPossibleDirections() {
        return isHorizontal ?
                List.of(Direction.LEFT, Direction.RIGHT) :
                List.of(Direction.UP, Direction.DOWN);
    }

    /**
     * Creates a new car shifted one cell in the specified direction.
     *
     * @param dir Direction to shift the car
     * @param width Width of the puzzle grid
     * @param height Height of the puzzle grid
     * @return New Car object with updated position, or null if move is 
invalid
     */
    public Car shift(Direction dir, int width, int height) {
        int totalBits = width * height;
        int chunkCount = (totalBits + 63) / 64;
        long[] shifted = new long[chunkCount];
```

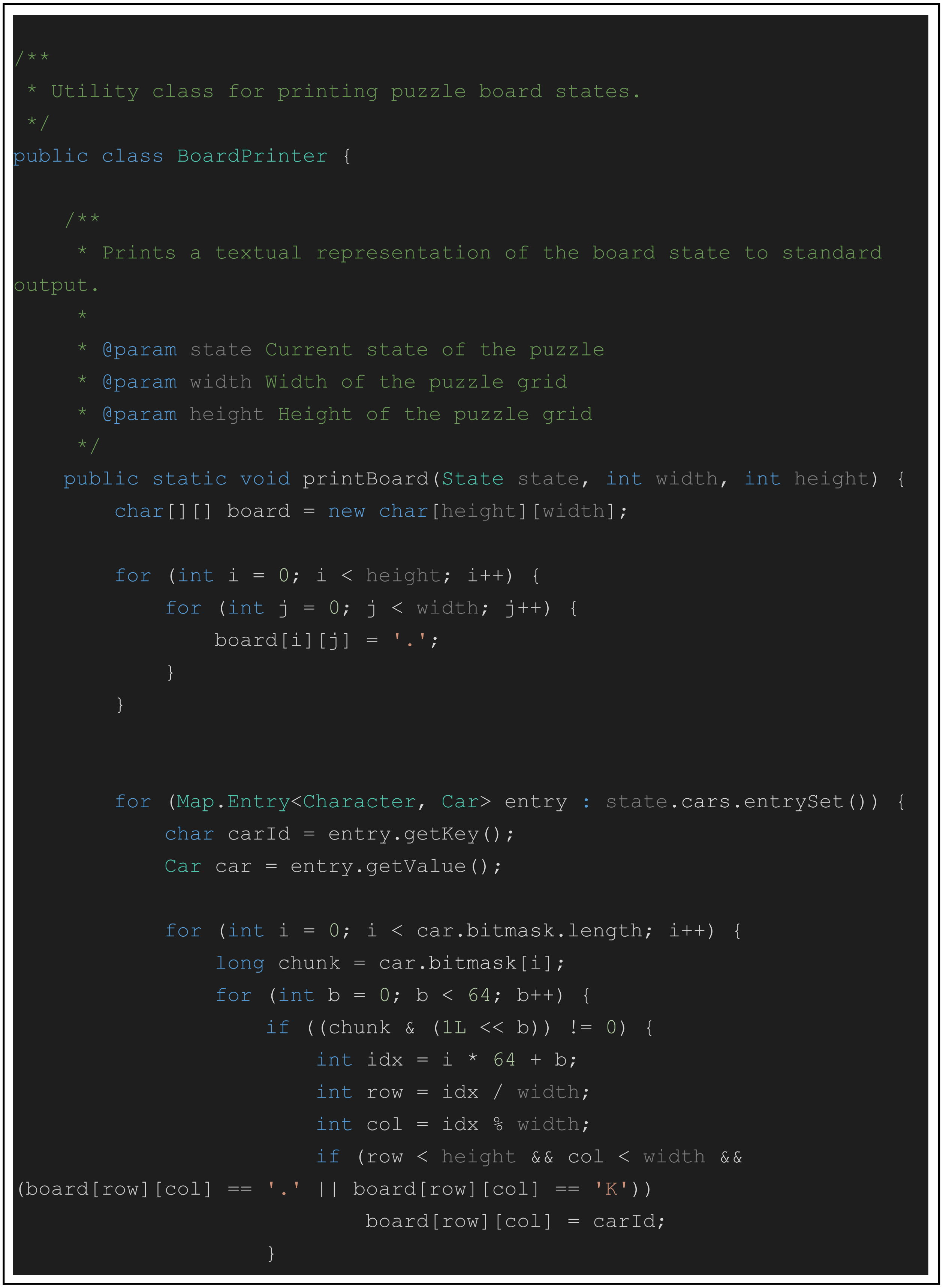


```
        int offset = switch (dir) {
            case LEFT -> -1;
            case RIGHT -> 1;
            case UP -> -width;
            case DOWN -> width;
        };

        for (int i = 0; i < totalBits; i++) {
            int chunk = i / 64;
            int bit = i % 64;
            if ((bitmask[chunk] & (1L << bit)) != 0) {
                int newIndex = i + offset;

                if (newIndex < 0 || newIndex >= totalBits) return null;

                int newRow = newIndex / width;
                int newCol = newIndex % width;
                int oldRow = i / width;
                int oldCol = i % width;

                if (isHorizontal && newRow != oldRow) return null;
                if (!isHorizontal && newCol != oldCol) return null;

                shifted[newIndex / 64] |= (1L << (newIndex % 64));
            }
        }

        return new Car(id, isHorizontal, length, shifted, col, row);
    }
}
```

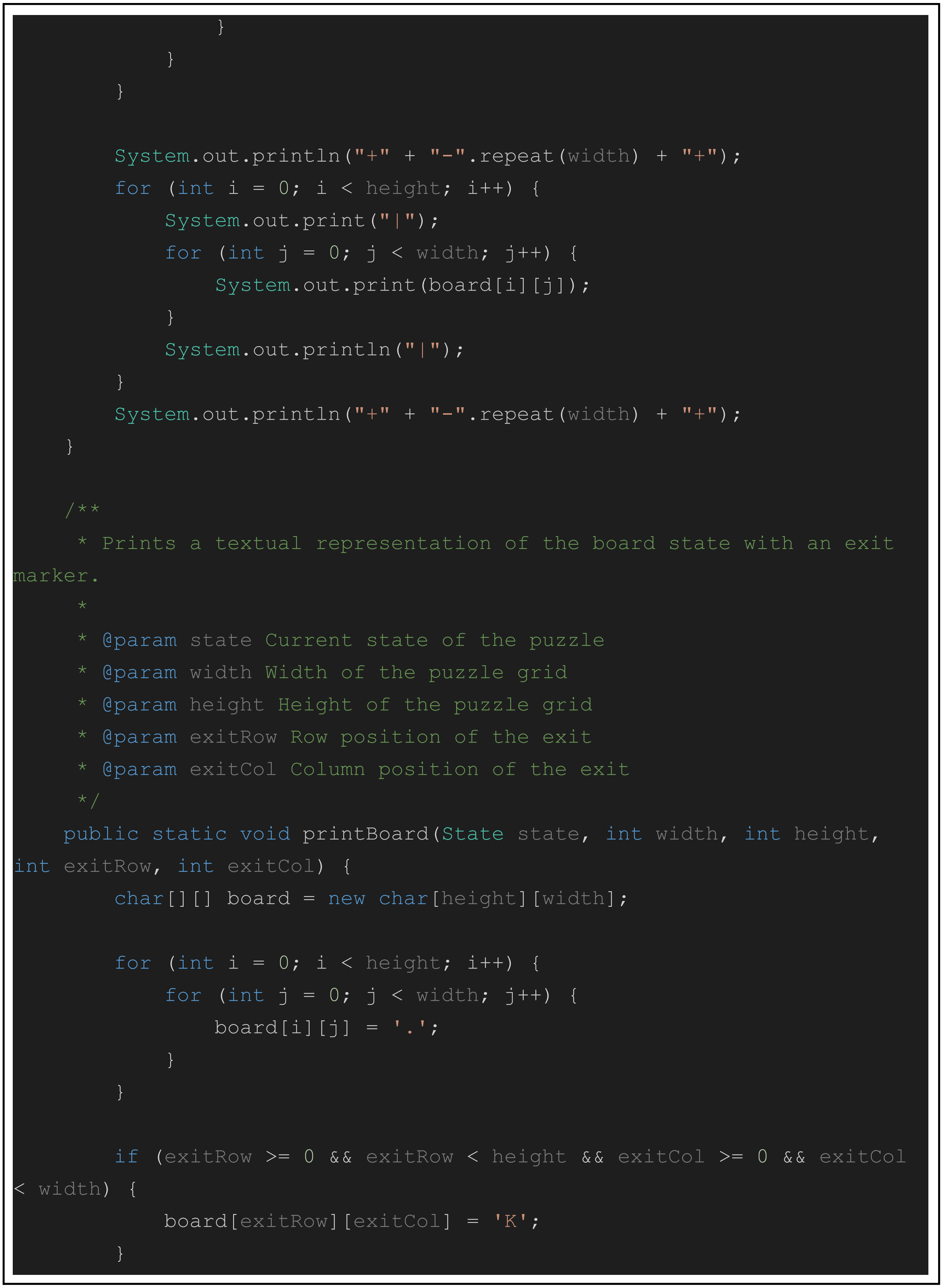
F. BoardPrinter



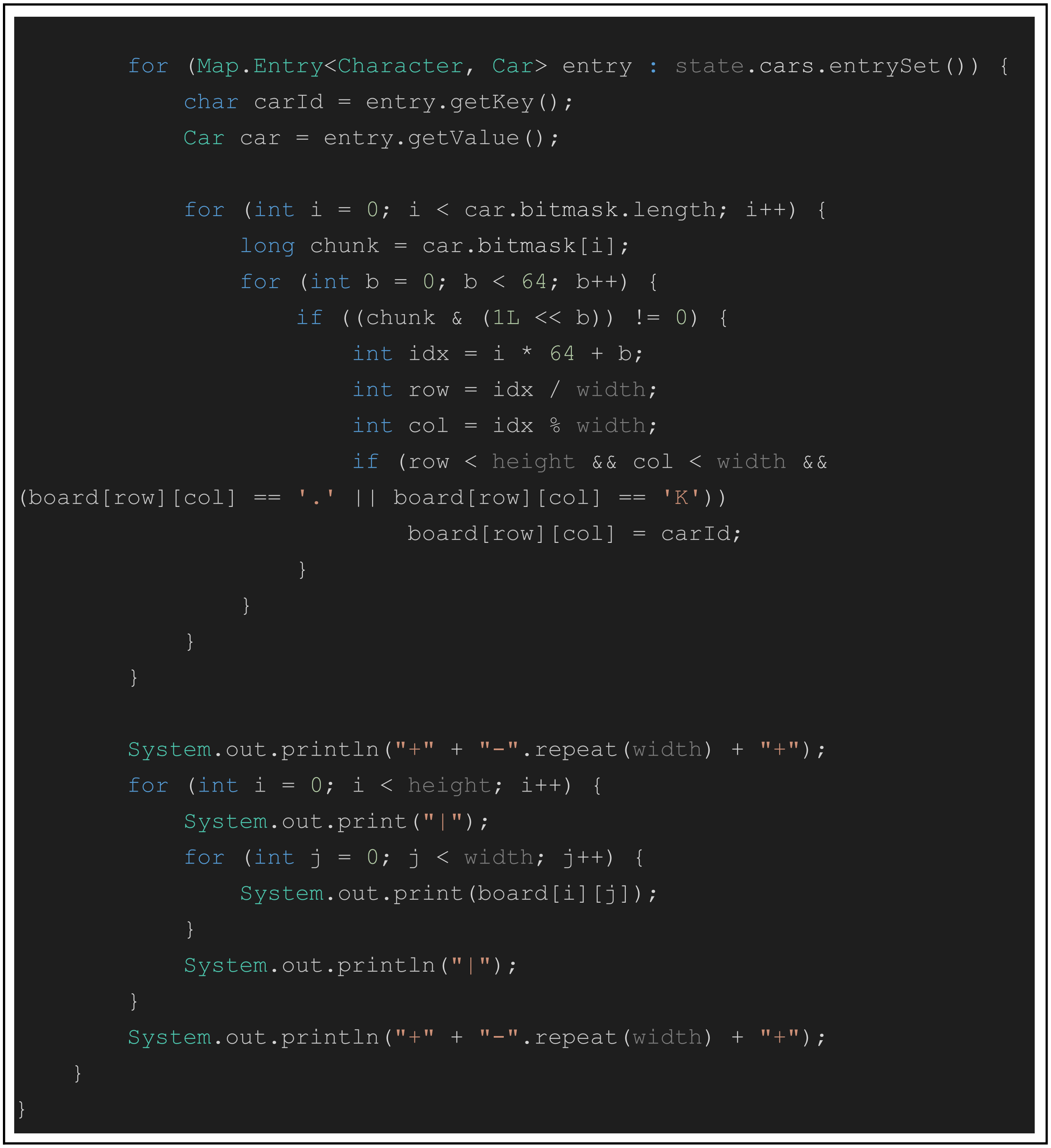
```
package util;

import java.util.Map;
```



```
/**
 * Utility class for printing puzzle board states.
 */
public class BoardPrinter {

    /**
     * Prints a textual representation of the board state to standard
output.
     *
     * @param state Current state of the puzzle
     * @param width Width of the puzzle grid
     * @param height Height of the puzzle grid
     */
    public static void printBoard(State state, int width, int height) {
        char[][] board = new char[height][width];

        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                board[i][j] = '.';
            }
        }


        for (Map.Entry<Character, Car> entry : state.cars.entrySet()) {
            char carId = entry.getKey();
            Car car = entry.getValue();

            for (int i = 0; i < car.bitmask.length; i++) {
                long chunk = car.bitmask[i];
                for (int b = 0; b < 64; b++) {
                    if ((chunk & (1L << b)) != 0) {
                        int idx = i * 64 + b;
                        int row = idx / width;
                        int col = idx % width;
                        if (row < height && col < width &&
(board[row][col] == '.' || board[row][col] == 'K'))
                            board[row][col] = carId;
                    }
```

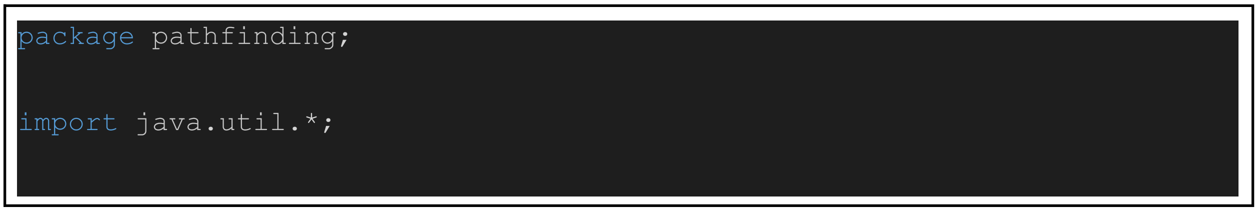


```
                }
            }
        }

        System.out.println("+" + "-".repeat(width) + "+");
        for (int i = 0; i < height; i++) {
            System.out.print("|");
            for (int j = 0; j < width; j++) {
                System.out.print(board[i][j]);
            }
            System.out.println("|");
        }
        System.out.println("+" + "-".repeat(width) + "+");
    }

    /**
     * Prints a textual representation of the board state with an exit
marker.
     *
     * @param state Current state of the puzzle
     * @param width Width of the puzzle grid
     * @param height Height of the puzzle grid
     * @param exitRow Row position of the exit
     * @param exitCol Column position of the exit
     */
    public static void printBoard(State state, int width, int height,
int exitRow, int exitCol) {
        char[][] board = new char[height][width];

        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                board[i][j] = '.';
            }
        }

        if (exitRow >= 0 && exitRow < height && exitCol >= 0 && exitCol
< width) {
            board[exitRow][exitCol] = 'K';
        }
```

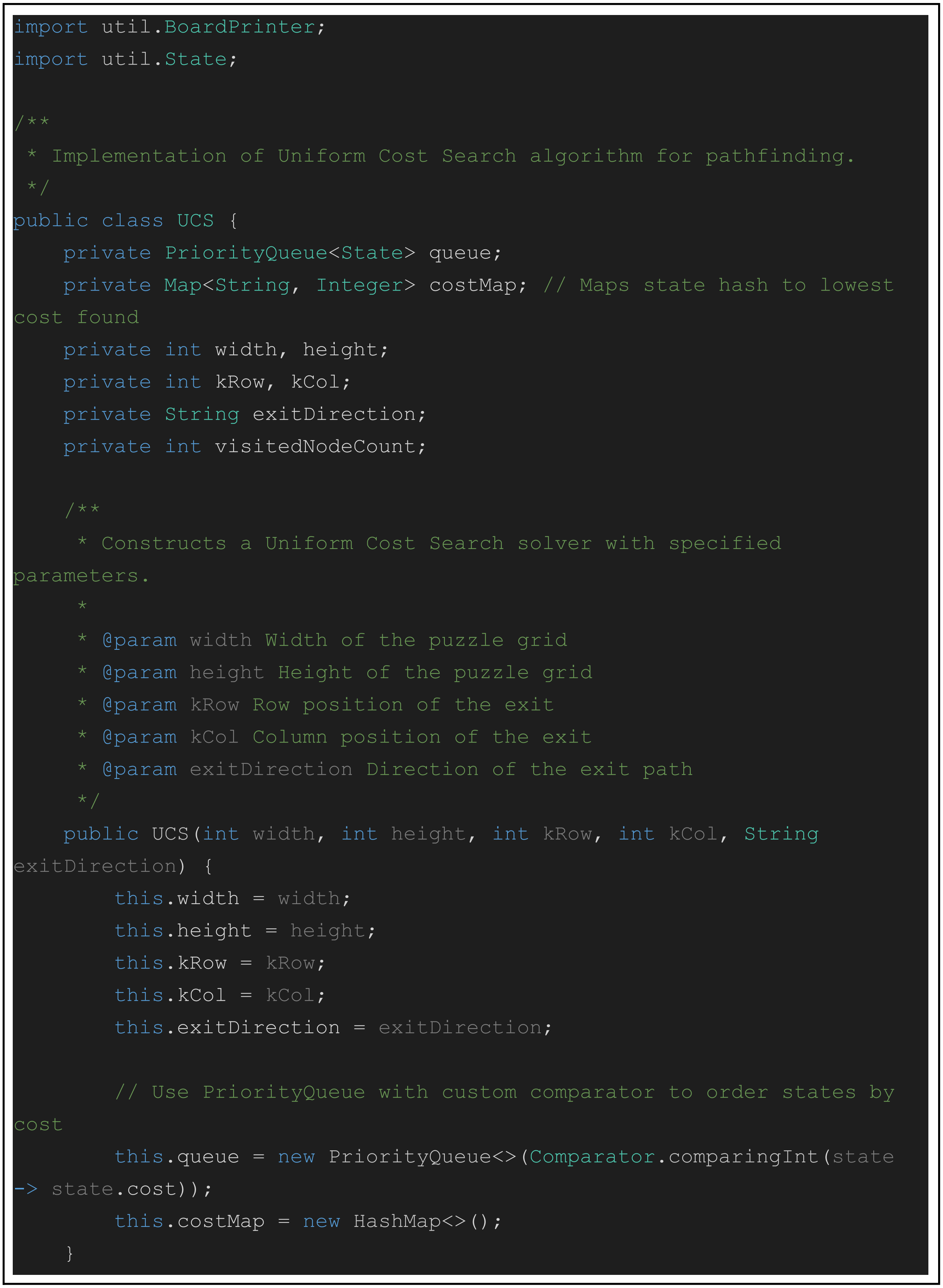


```
        for (Map.Entry<Character, Car> entry : state.cars.entrySet()) {
            char carId = entry.getKey();
            Car car = entry.getValue();

            for (int i = 0; i < car.bitmask.length; i++) {
                long chunk = car.bitmask[i];
                for (int b = 0; b < 64; b++) {
                    if ((chunk & (1L << b)) != 0) {
                        int idx = i * 64 + b;
                        int row = idx / width;
                        int col = idx % width;
                        if (row < height && col < width &&
(board[row][col] == '.' || board[row][col] == 'K'))
                            board[row][col] = carId;
                    }
                }
            }
        }

        System.out.println("+" + "-".repeat(width) + "+");
        for (int i = 0; i < height; i++) {
            System.out.print("|");
            for (int j = 0; j < width; j++) {
                System.out.print(board[i][j]);
            }
            System.out.println("|");
        }
        System.out.println("+" + "-".repeat(width) + "+");
    }
}
```

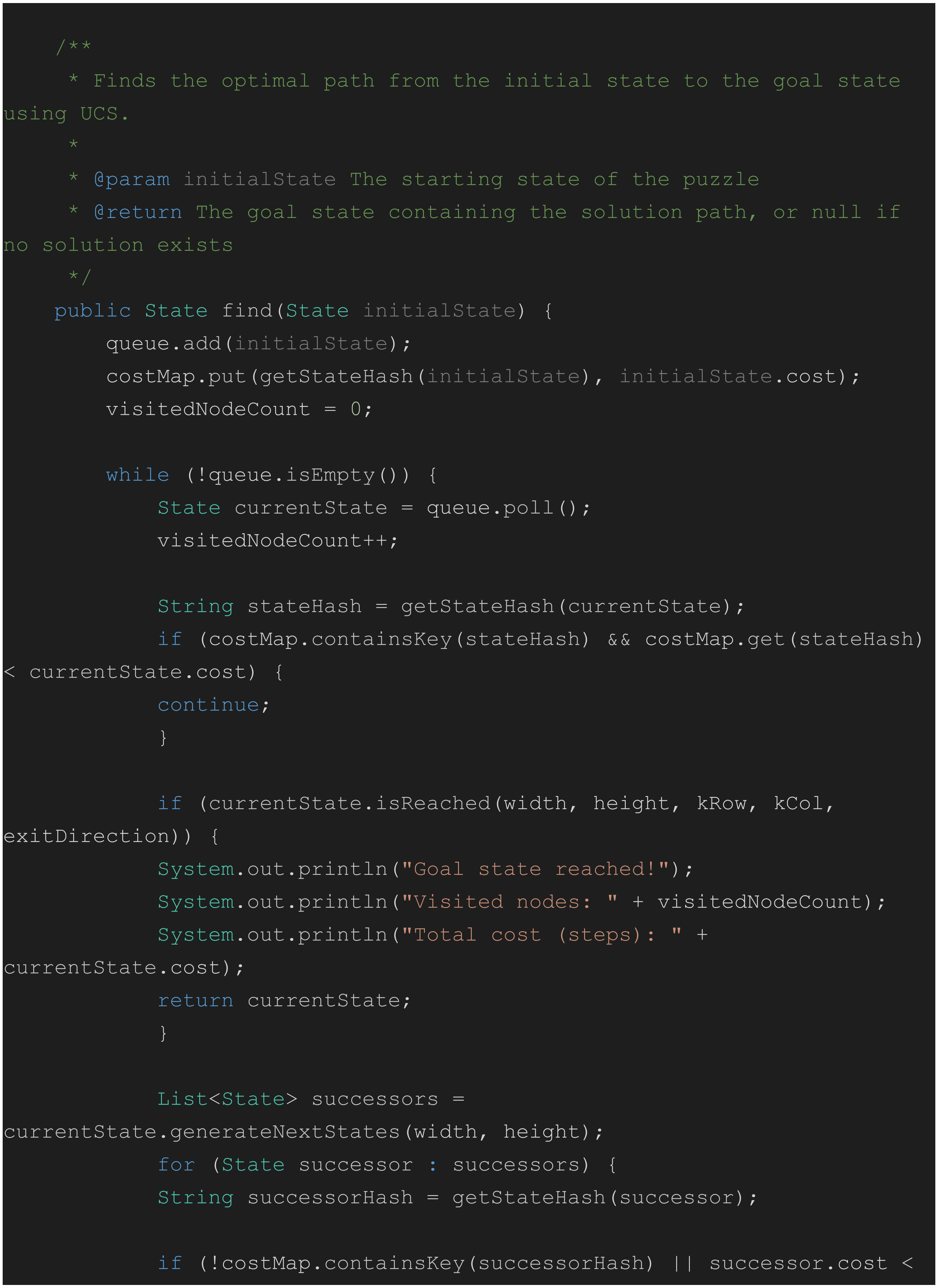
G. UCS



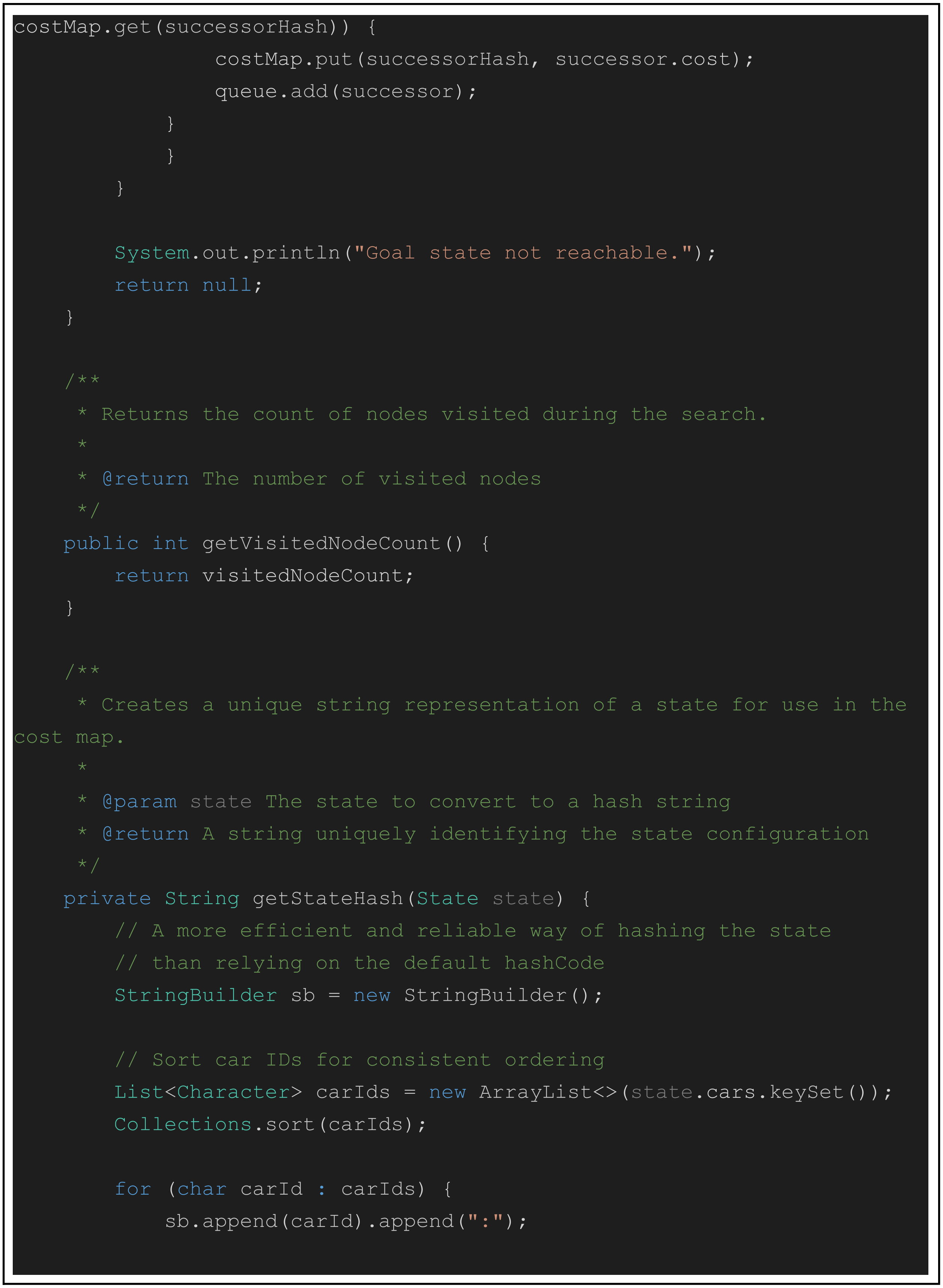
```
package pathfinding;

import java.util.*;
```



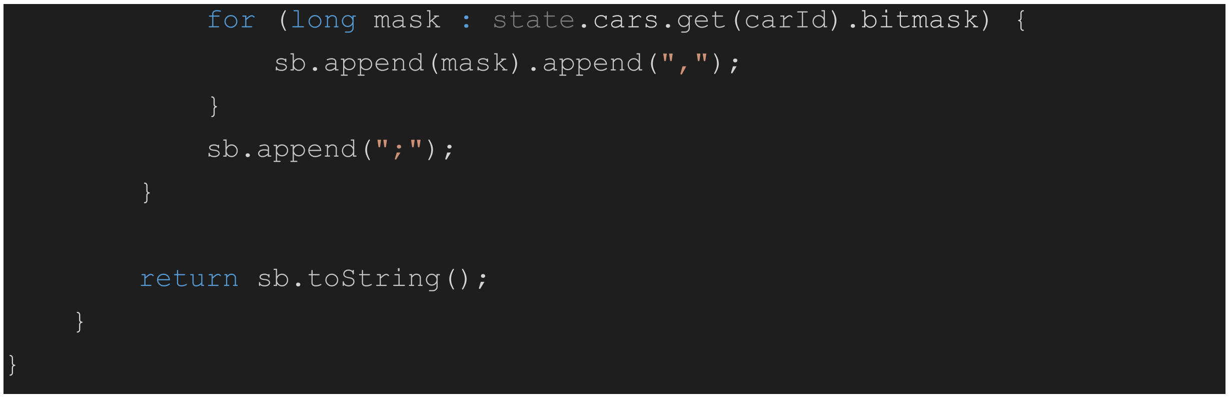
```
import util.BoardPrinter;
import util.State;

/**
 * Implementation of Uniform Cost Search algorithm for pathfinding.
 */
public class UCS {
    private PriorityQueue<State> queue;
    private Map<String, Integer> costMap; // Maps state hash to lowest
cost found
    private int width, height;
    private int kRow, kCol;
    private String exitDirection;
    private int visitedNodeCount;

    /**
     * Constructs a Uniform Cost Search solver with specified
parameters.
     *
     * @param width Width of the puzzle grid
     * @param height Height of the puzzle grid
     * @param kRow Row position of the exit
     * @param kCol Column position of the exit
     * @param exitDirection Direction of the exit path
     */
    public UCS(int width, int height, int kRow, int kCol, String
exitDirection) {
        this.width = width;
        this.height = height;
        this.kRow = kRow;
        this.kCol = kCol;
        this.exitDirection = exitDirection;

        // Use PriorityQueue with custom comparator to order states by
cost
        this.queue = new PriorityQueue<>(Comparator.comparingInt(state
-> state.cost));
        this.costMap = new HashMap<>();
    }
```



```
    /**
     * Finds the optimal path from the initial state to the goal state using UCS.
     *
     * @param initialState The starting state of the puzzle
     * @return The goal state containing the solution path, or null if no solution exists
     */
    public State find(State initialState) {
        queue.add(initialState);
        costMap.put(getStateHash(initialState), initialState.cost);
        visitedNodeCount = 0;

        while (!queue.isEmpty()) {
            State currentState = queue.poll();
            visitedNodeCount++;

            String stateHash = getStateHash(currentState);
            if (costMap.containsKey(stateHash) && costMap.get(stateHash) < currentState.cost) {
            continue;
            }

            if (currentState.isReached(width, height, kRow, kCol, exitDirection)) {
            System.out.println("Goal state reached!");
            System.out.println("Visited nodes: " + visitedNodeCount);
            System.out.println("Total cost (steps): " + currentState.cost);
            return currentState;
            }

            List<State> successors = currentState.generateNextStates(width, height);
            for (State successor : successors) {
            String successorHash = getStateHash(successor);

            if (!costMap.containsKey(successorHash) || successor.cost <
```

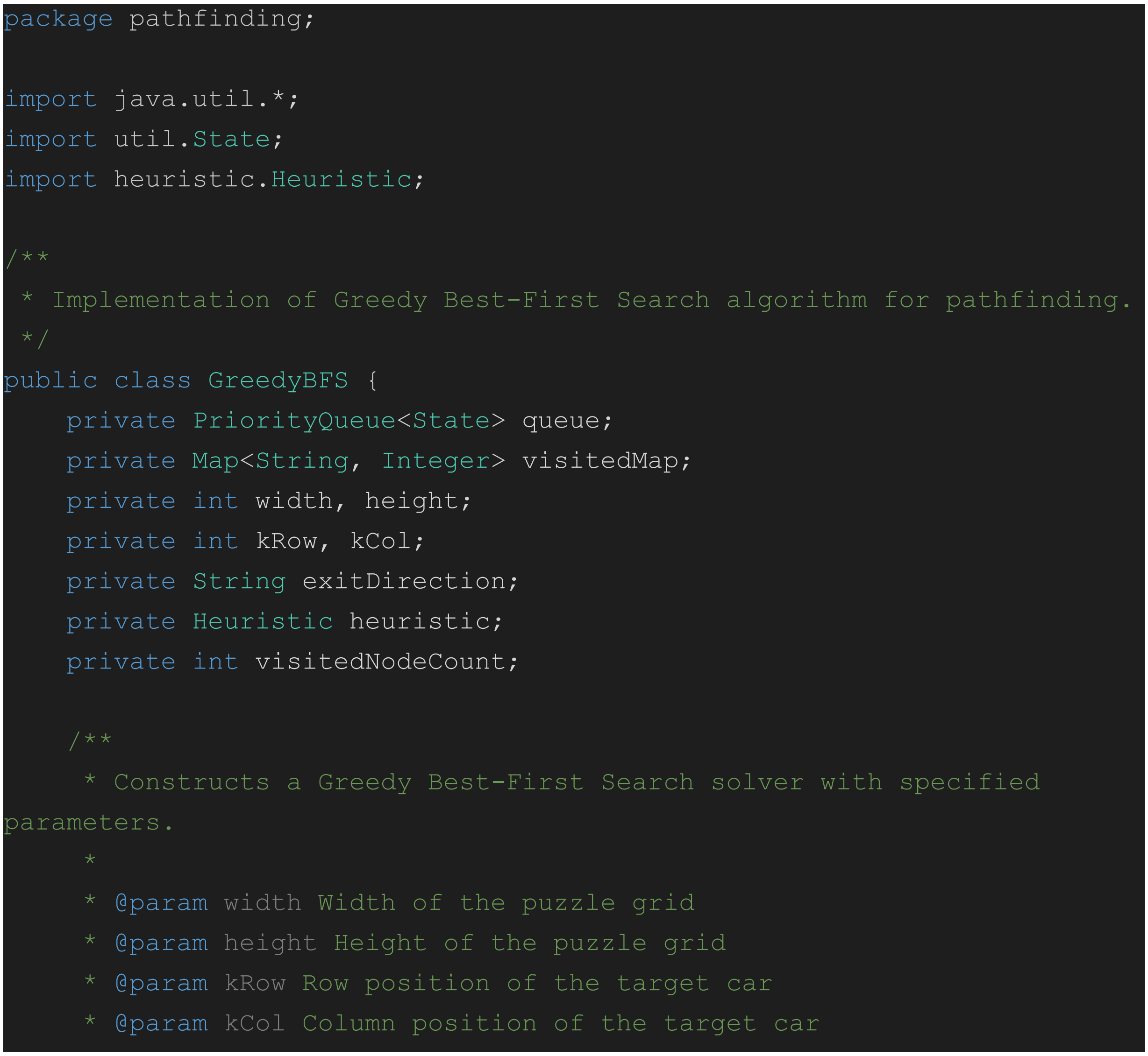


```
costMap.get(successorHash)) {
                costMap.put(successorHash, successor.cost);
                queue.add(successor);
            }
            }
        }

        System.out.println("Goal state not reachable.");
        return null;
    }

    /**
     * Returns the count of nodes visited during the search.
     *
     * @return The number of visited nodes
     */
    public int getVisitedNodeCount() {
        return visitedNodeCount;
    }

    /**
     * Creates a unique string representation of a state for use in the
cost map.
     *
     * @param state The state to convert to a hash string
     * @return A string uniquely identifying the state configuration
     */
    private String getStateHash(State state) {
        // A more efficient and reliable way of hashing the state
        // than relying on the default hashCode
        StringBuilder sb = new StringBuilder();

        // Sort car IDs for consistent ordering
        List<Character> carIds = new ArrayList<>(state.cars.keySet());
        Collections.sort(carIds);

        for (char carId : carIds) {
            sb.append(carId).append(":");
```

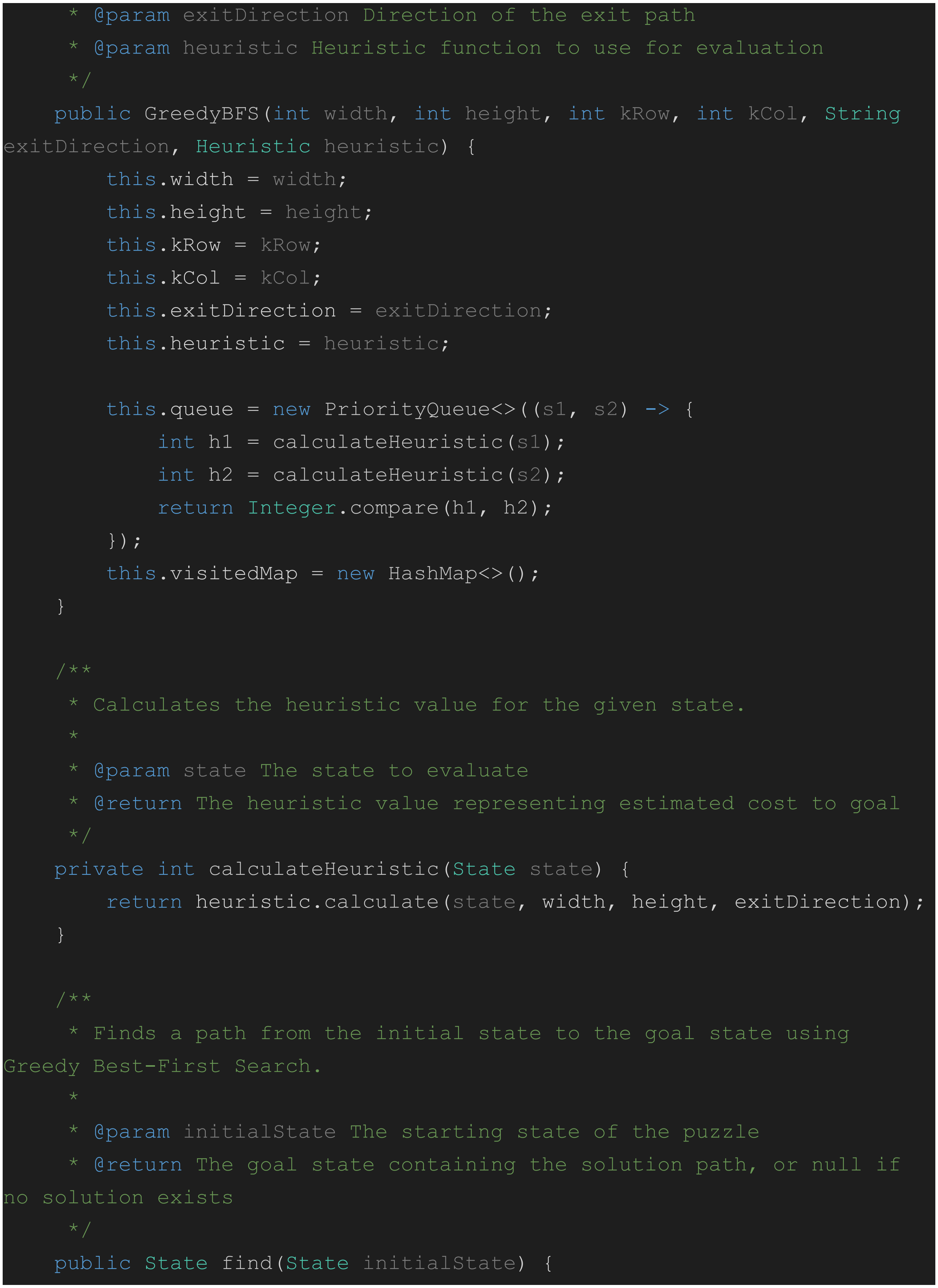


```
            for (long mask : state.cars.get(carId).bitmask) {
                sb.append(mask).append(",");
            }
            sb.append(";");
        }

        return sb.toString();
    }
}
```

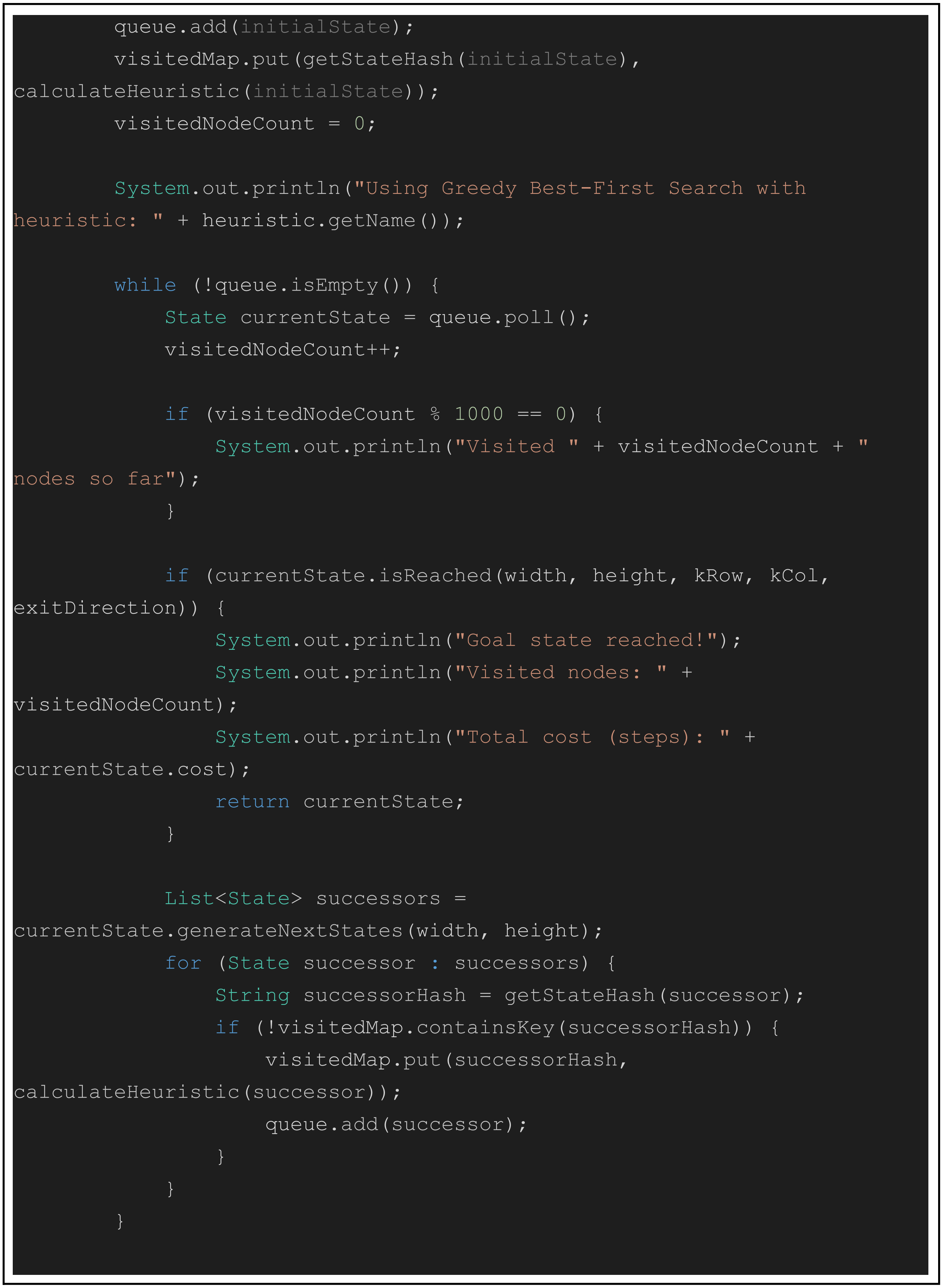
H. GreedyBFS



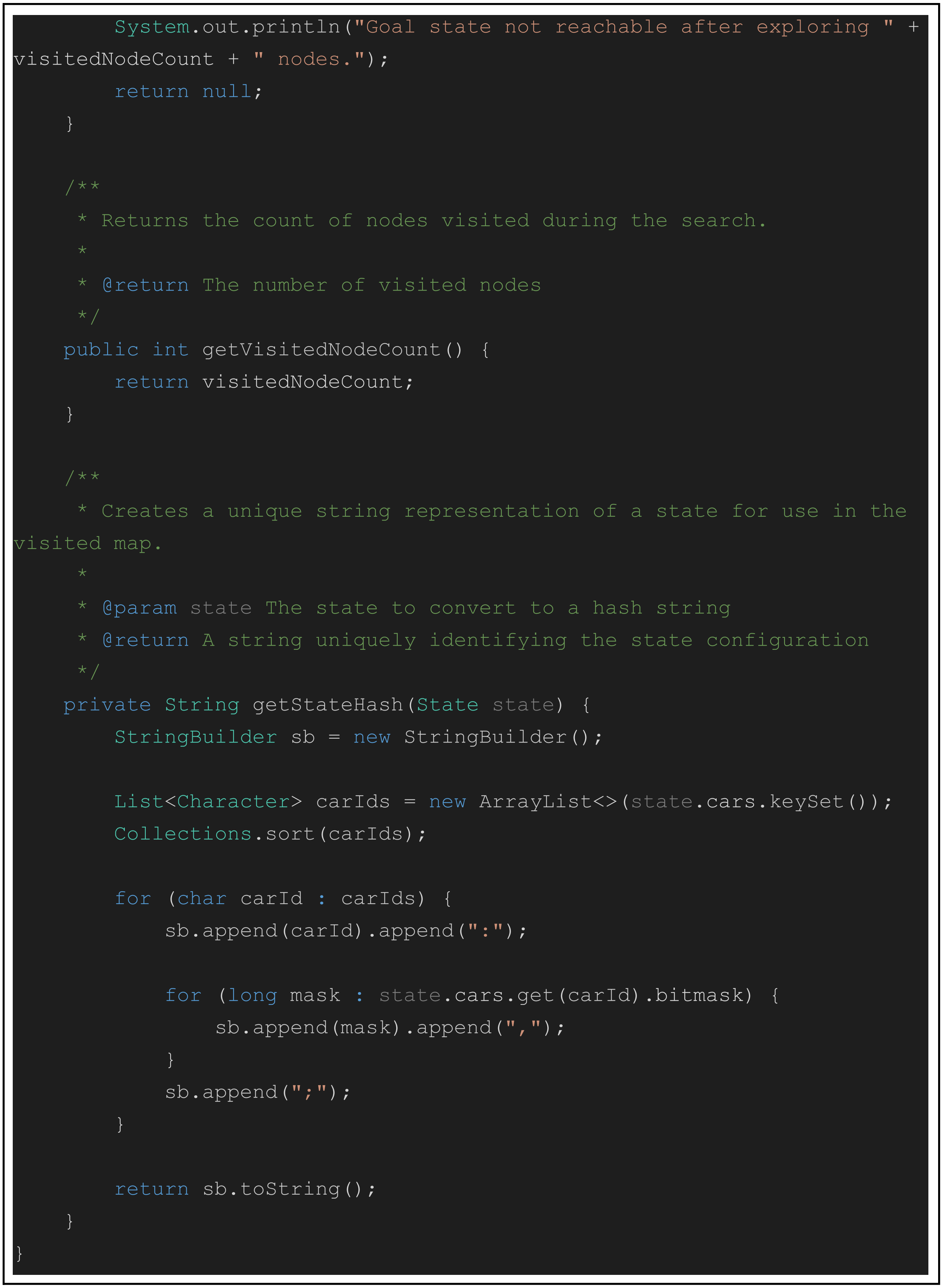
```
package pathfinding;

import java.util.*;
import util.State;
import heuristic.Heuristic;

/**
 * Implementation of Greedy Best-First Search algorithm for pathfinding.
 */
public class GreedyBFS {
    private PriorityQueue<State> queue;
    private Map<String, Integer> visitedMap;
    private int width, height;
    private int kRow, kCol;
    private String exitDirection;
    private Heuristic heuristic;
    private int visitedNodeCount;

    /**
     * Constructs a Greedy Best-First Search solver with specified
parameters.
     *
     * @param width Width of the puzzle grid
     * @param height Height of the puzzle grid
     * @param kRow Row position of the target car
     * @param kCol Column position of the target car
```



```
     * @param exitDirection Direction of the exit path
     * @param heuristic Heuristic function to use for evaluation
     */
    public GreedyBFS(int width, int height, int kRow, int kCol, String exitDirection, Heuristic heuristic) {
        this.width = width;
        this.height = height;
        this.kRow = kRow;
        this.kCol = kCol;
        this.exitDirection = exitDirection;
        this.heuristic = heuristic;

        this.queue = new PriorityQueue<>((s1, s2) -> {
            int h1 = calculateHeuristic(s1);
            int h2 = calculateHeuristic(s2);
            return Integer.compare(h1, h2);
        });
        this.visitedMap = new HashMap<>();
    }

    /**
     * Calculates the heuristic value for the given state.
     *
     * @param state The state to evaluate
     * @return The heuristic value representing estimated cost to goal
     */
    private int calculateHeuristic(State state) {
        return heuristic.calculate(state, width, height, exitDirection);
    }

    /**
     * Finds a path from the initial state to the goal state using Greedy Best-First Search.
     *
     * @param initialState The starting state of the puzzle
     * @return The goal state containing the solution path, or null if no solution exists
     */
    public State find(State initialState) {
```



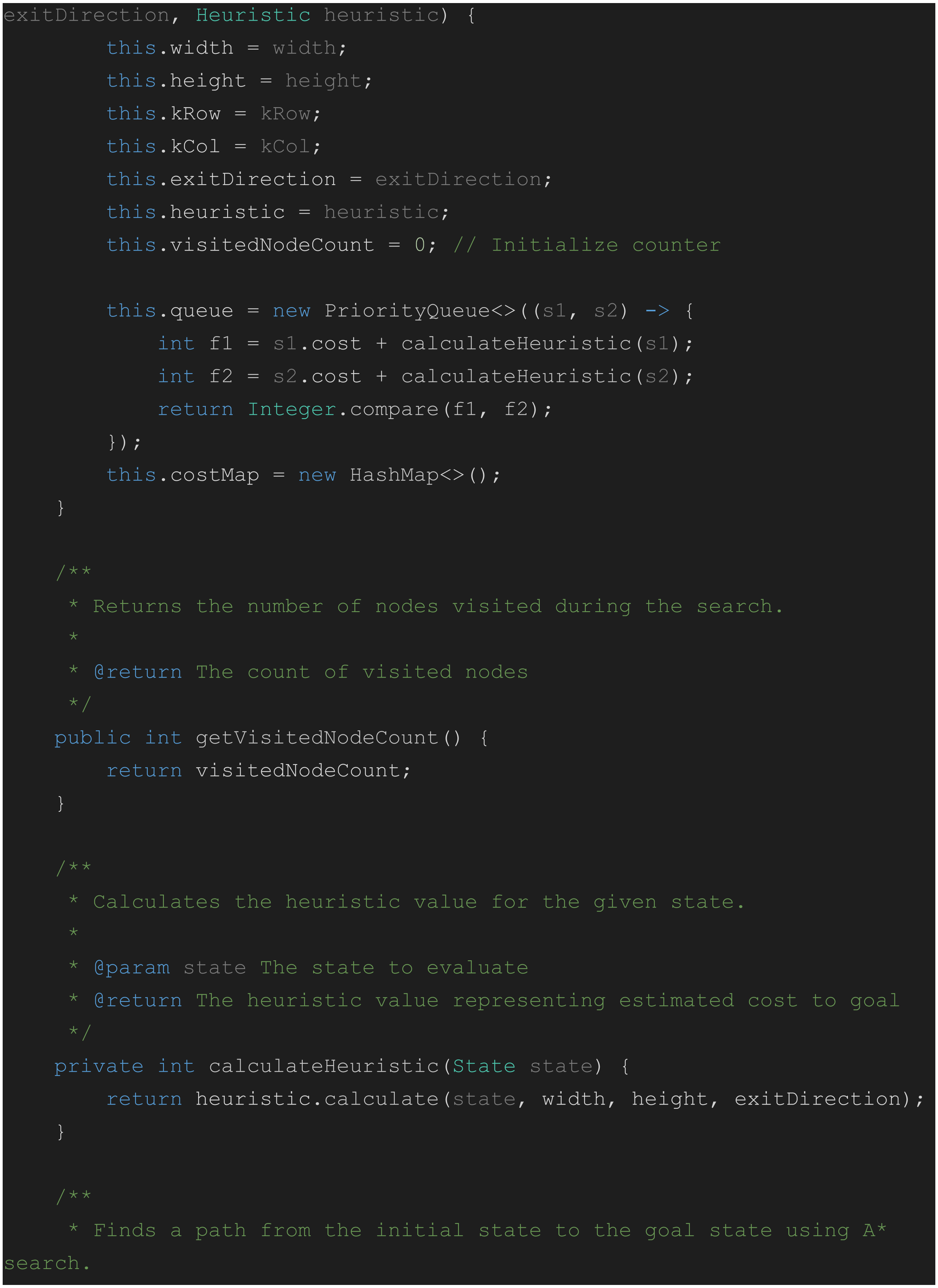
```
        queue.add(initialState);
        visitedMap.put(getStateHash(initialState),
calculateHeuristic(initialState));
        visitedNodeCount = 0;

        System.out.println("Using Greedy Best-First Search with
heuristic: " + heuristic.getName());

        while (!queue.isEmpty()) {
            State currentState = queue.poll();
            visitedNodeCount++;

            if (visitedNodeCount % 1000 == 0) {
                System.out.println("Visited " + visitedNodeCount + "
nodes so far");
            }

            if (currentState.isReached(width, height, kRow, kCol,
exitDirection)) {
                System.out.println("Goal state reached!");
                System.out.println("Visited nodes: " +
visitedNodeCount);
                System.out.println("Total cost (steps): " +
currentState.cost);
                return currentState;
            }

            List<State> successors =
currentState.generateNextStates(width, height);
            for (State successor : successors) {
                String successorHash = getStateHash(successor);
                if (!visitedMap.containsKey(successorHash)) {
                    visitedMap.put(successorHash,
calculateHeuristic(successor));
                    queue.add(successor);
                }
            }
        }
```



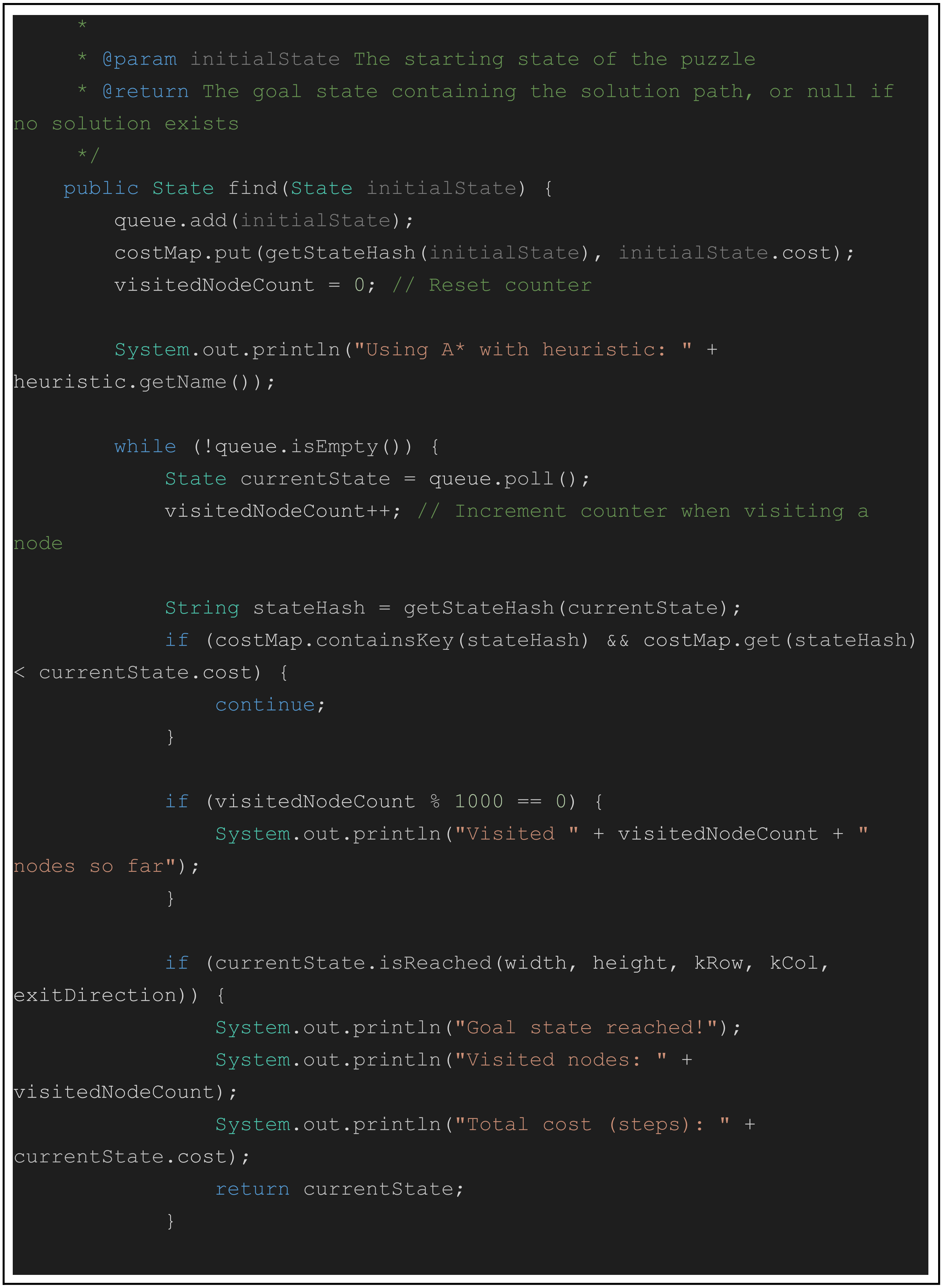
```
        System.out.println("Goal state not reachable after exploring " +
visitedNodeCount + " nodes.");
        return null;
    }

    /**
     * Returns the count of nodes visited during the search.
     *
     * @return The number of visited nodes
     */
    public int getVisitedNodeCount() {
        return visitedNodeCount;
    }

    /**
     * Creates a unique string representation of a state for use in the
visited map.
     *
     * @param state The state to convert to a hash string
     * @return A string uniquely identifying the state configuration
     */
    private String getStateHash(State state) {
        StringBuilder sb = new StringBuilder();

        List<Character> carIds = new ArrayList<>(state.cars.keySet());
        Collections.sort(carIds);

        for (char carId : carIds) {
            sb.append(carId).append(":");

            for (long mask : state.cars.get(carId).bitmask) {
                sb.append(mask).append(",");
            }
            sb.append(";");
        }

        return sb.toString();
    }
}
```

I. AStar

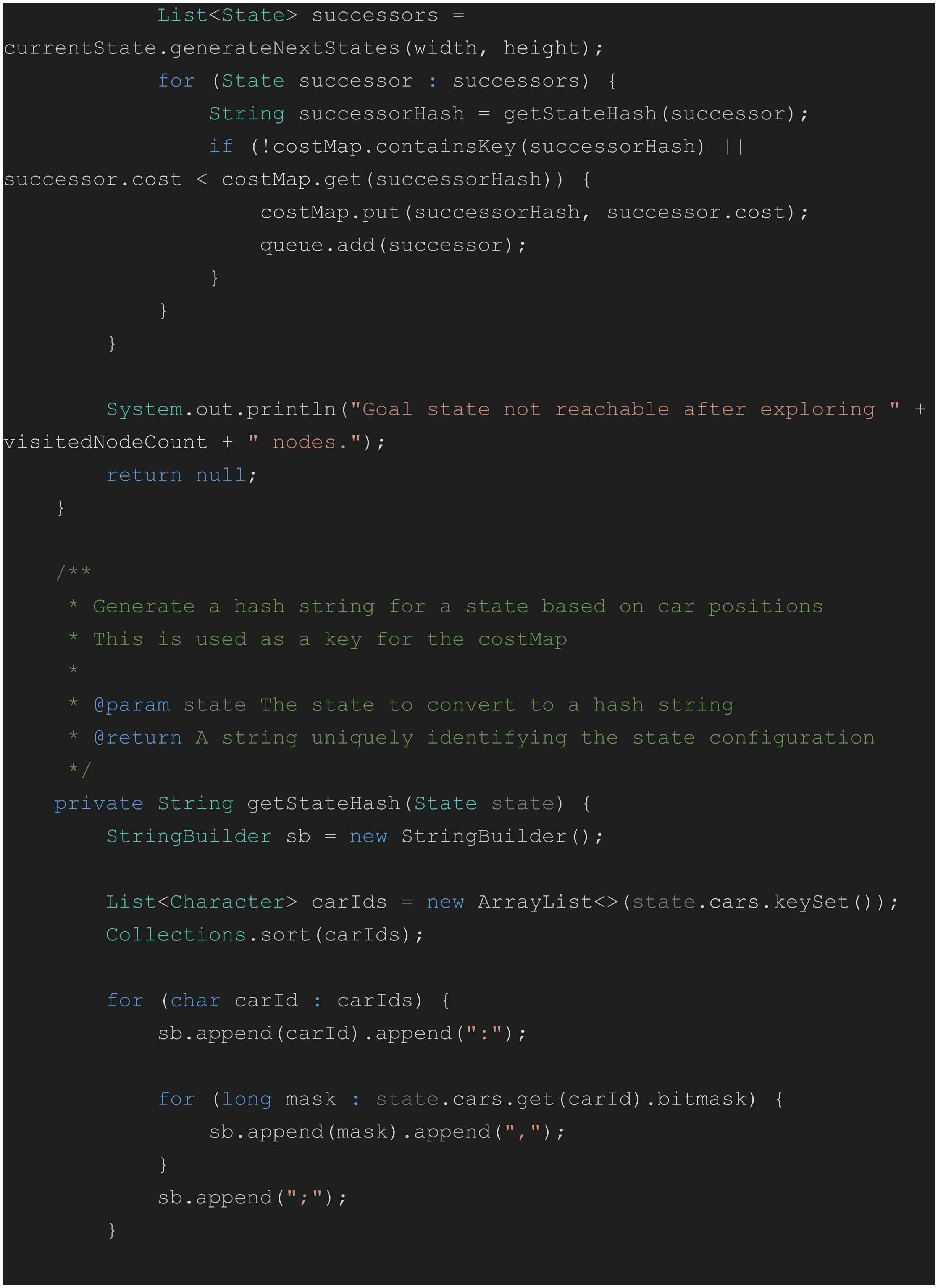
```
package pathfinding;

import java.util.*;

import util.BoardPrinter;
import util.State;
import heuristic.Heuristic;
import heuristic.Distance;

/*
Dalam malam bertabur bintang,
Langkah awal terpampang,
Menapaki simpul-simpul harapan,
Menuju terang dalam kegelapan.
*/

public class AStar {
    private PriorityQueue<State> queue;
    private Map<String, Integer> costMap;
    private int width, height;
    private int kRow, kCol;
    private String exitDirection;
    private Heuristic heuristic;
    private int visitedNodeCount; // Add field to store visited node
count

    /**
     * Constructs an A* search solver with specified parameters.
     *
     * @param width Width of the puzzle grid
     * @param height Height of the puzzle grid
     * @param kRow Row position of the exit
     * @param kCol Column position of the exit
     * @param exitDirection Direction of the exit path
     * @param heuristic Heuristic function to use for evaluation
     */
    public AStar(int width, int height, int kRow, int kCol, String
```



```java
exitDirection, Heuristic heuristic) {
        this.width = width;
        this.height = height;
        this.kRow = kRow;
        this.kCol = kCol;
        this.exitDirection = exitDirection;
        this.heuristic = heuristic;
        this.visitedNodeCount = 0; // Initialize counter

        this.queue = new PriorityQueue<>((s1, s2) -> {
            int f1 = s1.cost + calculateHeuristic(s1);
            int f2 = s2.cost + calculateHeuristic(s2);
            return Integer.compare(f1, f2);
        });
        this.costMap = new HashMap<>();
    }

    /**
     * Returns the number of nodes visited during the search.
     *
     * @return The count of visited nodes
     */
    public int getVisitedNodeCount() {
        return visitedNodeCount;
    }

    /**
     * Calculates the heuristic value for the given state.
     *
     * @param state The state to evaluate
     * @return The heuristic value representing estimated cost to goal
     */
    private int calculateHeuristic(State state) {
        return heuristic.calculate(state, width, height, exitDirection);
    }

    /**
     * Finds a path from the initial state to the goal state using A*
search.
```



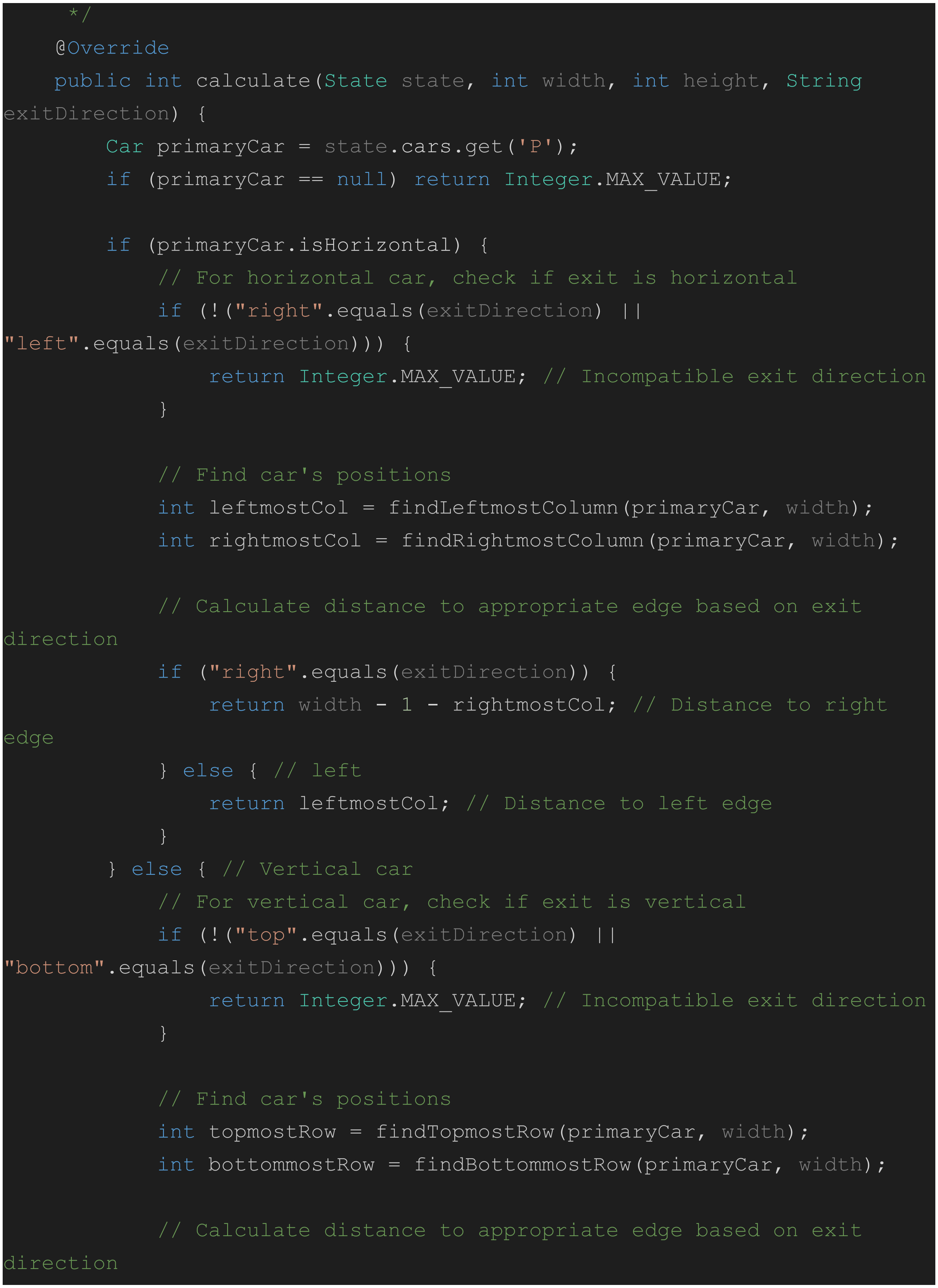
```
     *
     * @param initialState The starting state of the puzzle
     * @return The goal state containing the solution path, or null if
no solution exists
     */
    public State find(State initialState) {
        queue.add(initialState);
        costMap.put(getStateHash(initialState), initialState.cost);
        visitedNodeCount = 0; // Reset counter

        System.out.println("Using A* with heuristic: " +
heuristic.getName());

        while (!queue.isEmpty()) {
            State currentState = queue.poll();
            visitedNodeCount++; // Increment counter when visiting a
node

            String stateHash = getStateHash(currentState);
            if (costMap.containsKey(stateHash) && costMap.get(stateHash)
< currentState.cost) {
                continue;
            }

            if (visitedNodeCount % 1000 == 0) {
                System.out.println("Visited " + visitedNodeCount + "
nodes so far");
            }

            if (currentState.isReached(width, height, kRow, kCol,
exitDirection)) {
                System.out.println("Goal state reached!");
                System.out.println("Visited nodes: " +
visitedNodeCount);
                System.out.println("Total cost (steps): " +
currentState.cost);
                return currentState;
            }
```



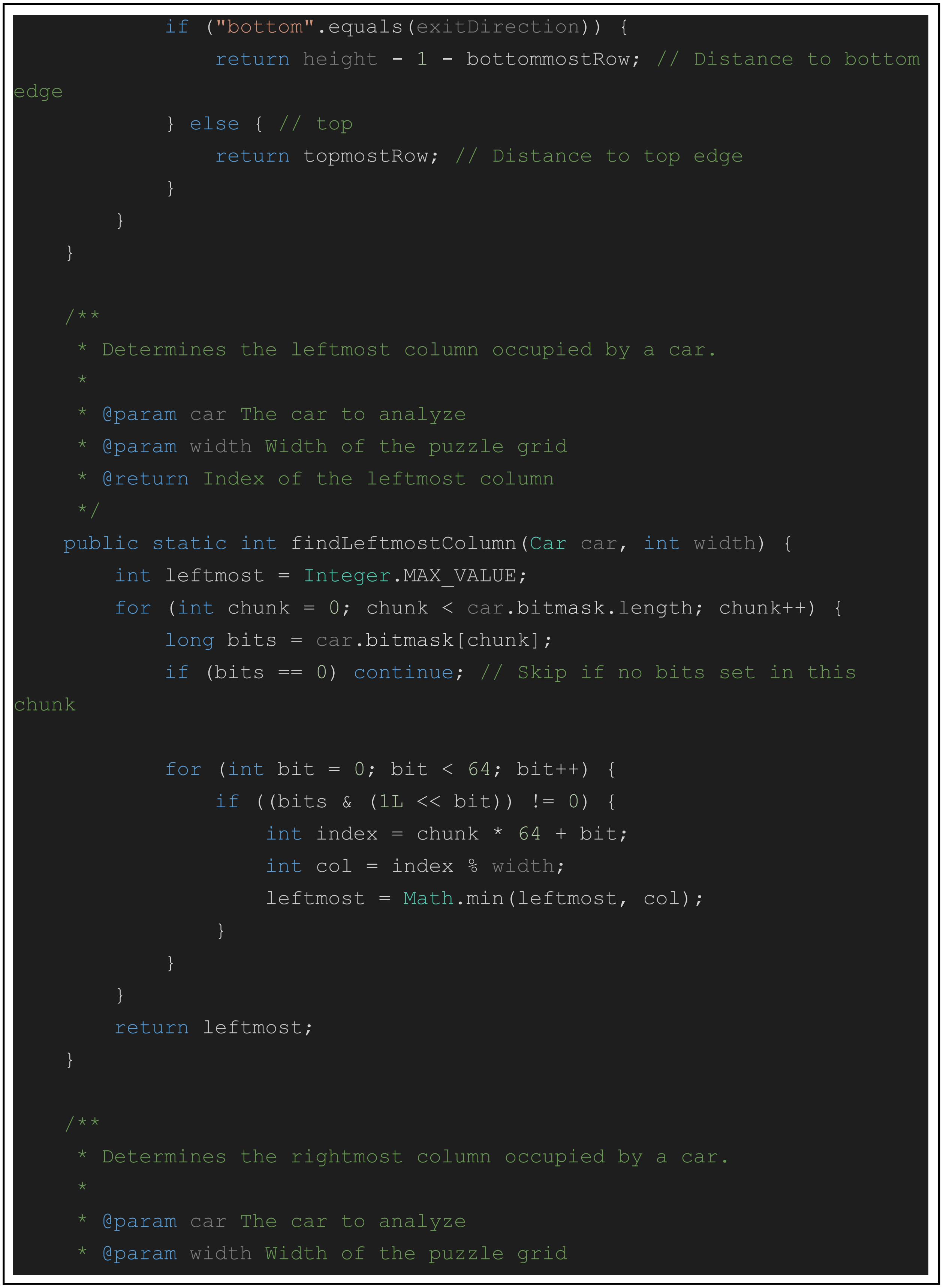
```java
            List<State> successors =
currentState.generateNextStates(width, height);
            for (State successor : successors) {
                String successorHash = getStateHash(successor);
                if (!costMap.containsKey(successorHash) ||
successor.cost < costMap.get(successorHash)) {
                    costMap.put(successorHash, successor.cost);
                    queue.add(successor);
                }
            }
        }

        System.out.println("Goal state not reachable after exploring " +
visitedNodeCount + " nodes.");
        return null;
    }

    /**
     * Generate a hash string for a state based on car positions
     * This is used as a key for the costMap
     *
     * @param state The state to convert to a hash string
     * @return A string uniquely identifying the state configuration
     */
    private String getStateHash(State state) {
        StringBuilder sb = new StringBuilder();

        List<Character> carIds = new ArrayList<>(state.cars.keySet());
        Collections.sort(carIds);

        for (char carId : carIds) {
            sb.append(carId).append(":");

            for (long mask : state.cars.get(carId).bitmask) {
                sb.append(mask).append(",");
            }
            sb.append(";");
        }
        return sb.toString();
    }
}
```

J. Distance

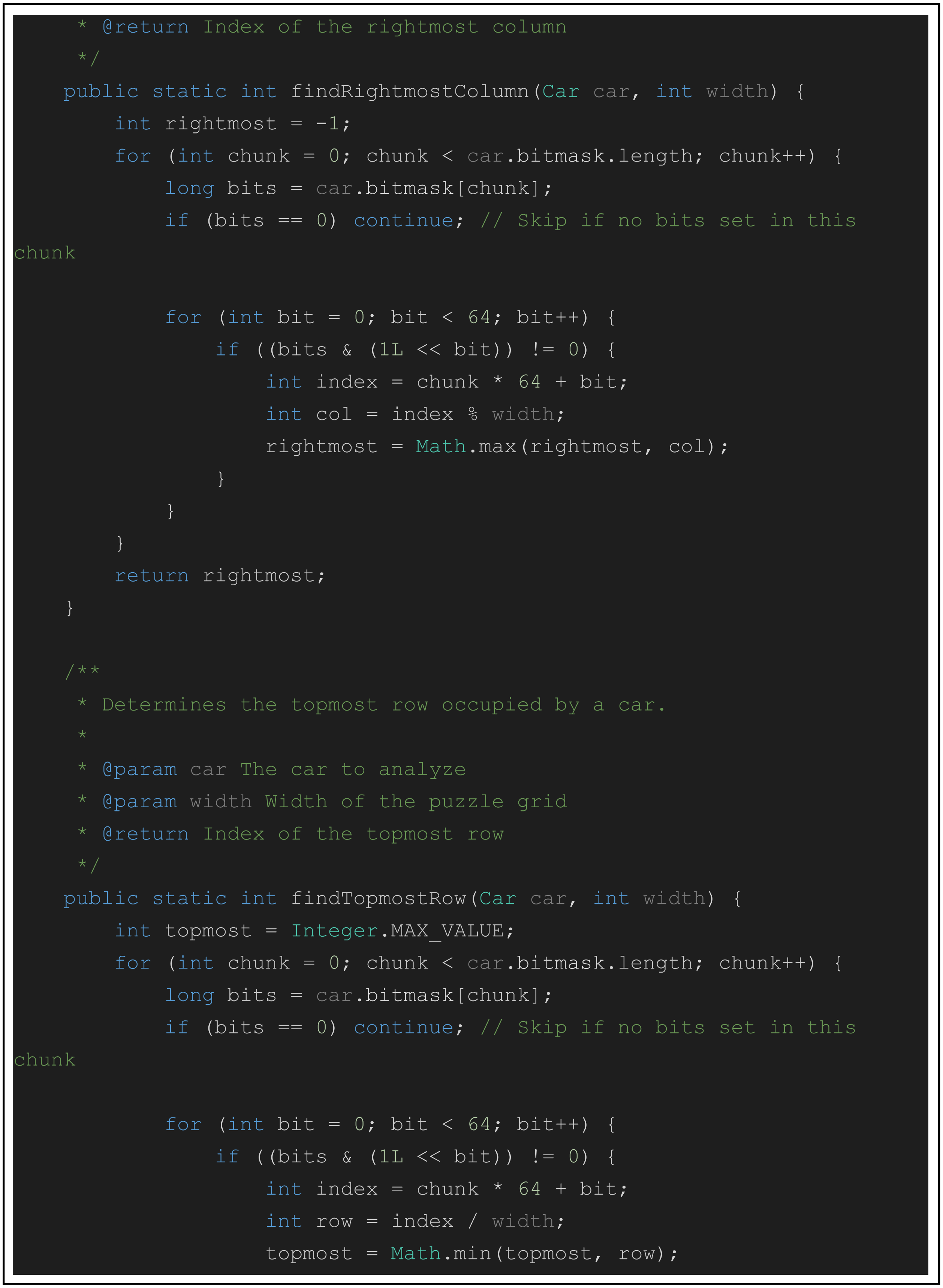
```
package heuristic;

import util.Car;
import util.State;

/**
 * Simple heuristic that calculates the distance from the primary car to the exit.
 */
public class Distance implements Heuristic {

    /**
     * Returns the name of this heuristic function.
     *
     * @return String name of the heuristic
     */
    @Override
    public String getName() {
        return "Distance to Exit";
    }

    /**
     * Calculates the direct distance from the primary car to the exit edge.
     *
     * @param state The current puzzle state
     * @param width Width of the puzzle grid
     * @param height Height of the puzzle grid
     * @param exitDirection Direction of the exit ("left", "right", "top", "bottom")
     * @return Distance to exit or MAX_VALUE if car orientation is incompatible with exit
```



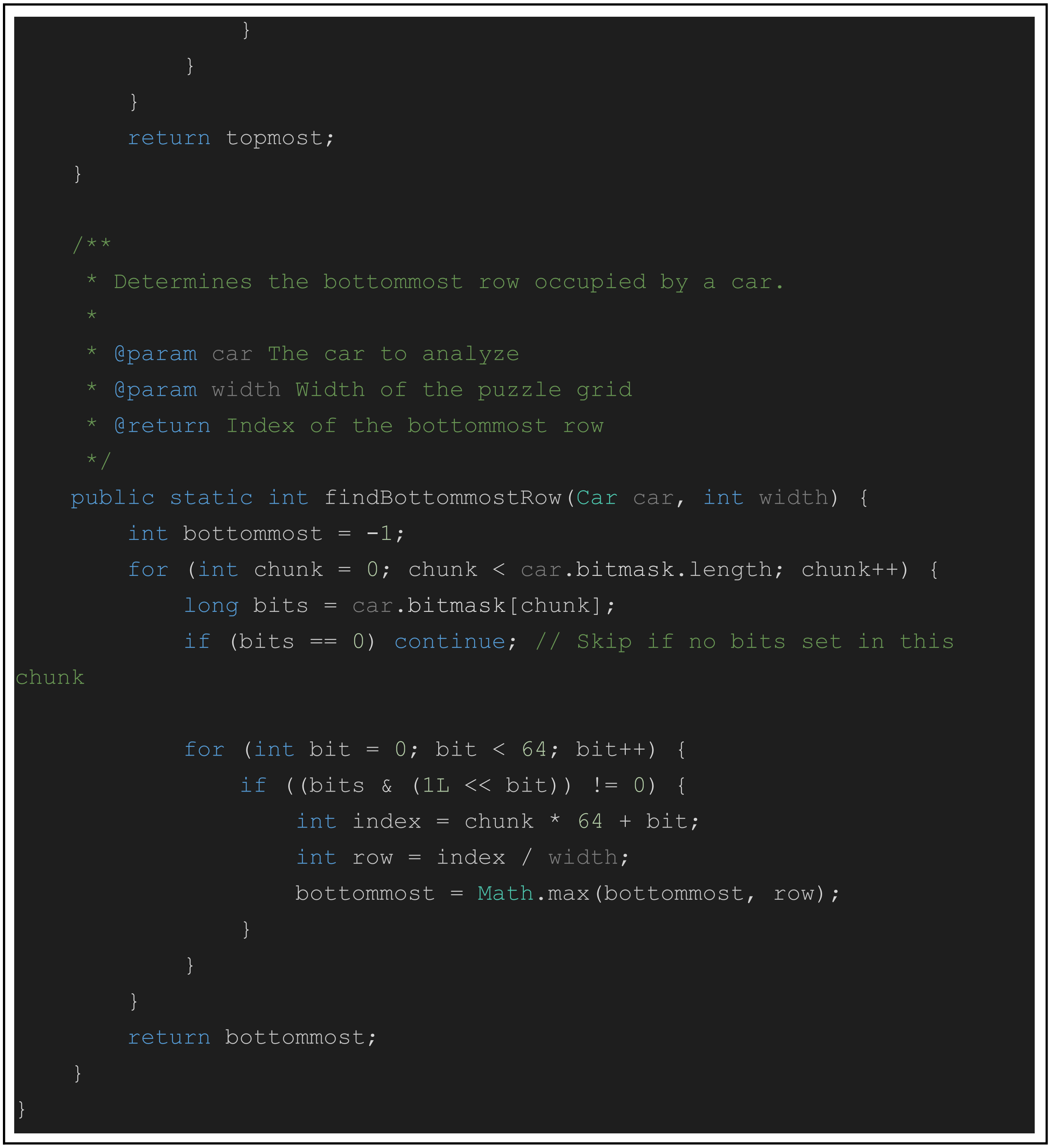
```
     */
    @Override
    public int calculate(State state, int width, int height, String
exitDirection) {
        Car primaryCar = state.cars.get('P');
        if (primaryCar == null) return Integer.MAX_VALUE;

        if (primaryCar.isHorizontal) {
            // For horizontal car, check if exit is horizontal
            if (!("right".equals(exitDirection) ||
"left".equals(exitDirection))) {
                return Integer.MAX_VALUE; // Incompatible exit direction
            }

            // Find car's positions
            int leftmostCol = findLeftmostColumn(primaryCar, width);
            int rightmostCol = findRightmostColumn(primaryCar, width);

            // Calculate distance to appropriate edge based on exit
direction
            if ("right".equals(exitDirection)) {
                return width - 1 - rightmostCol; // Distance to right
edge
            } else { // left
                return leftmostCol; // Distance to left edge
            }
        } else { // Vertical car
            // For vertical car, check if exit is vertical
            if (!("top".equals(exitDirection) ||
"bottom".equals(exitDirection))) {
                return Integer.MAX_VALUE; // Incompatible exit direction
            }

            // Find car's positions
            int topmostRow = findTopmostRow(primaryCar, width);
            int bottommostRow = findBottommostRow(primaryCar, width);

            // Calculate distance to appropriate edge based on exit
direction
```



```
            if ("bottom".equals(exitDirection)) {
                return height - 1 - bottommostRow; // Distance to bottom edge
            } else { // top
                return topmostRow; // Distance to top edge
            }
        }
    }

    /**
     * Determines the leftmost column occupied by a car.
     *
     * @param car The car to analyze
     * @param width Width of the puzzle grid
     * @return Index of the leftmost column
     */
    public static int findLeftmostColumn(Car car, int width) {
        int leftmost = Integer.MAX_VALUE;
        for (int chunk = 0; chunk < car.bitmask.length; chunk++) {
            long bits = car.bitmask[chunk];
            if (bits == 0) continue; // Skip if no bits set in this chunk

            for (int bit = 0; bit < 64; bit++) {
                if ((bits & (1L << bit)) != 0) {
                    int index = chunk * 64 + bit;
                    int col = index % width;
                    leftmost = Math.min(leftmost, col);
                }
            }
        }
        return leftmost;
    }

    /**
     * Determines the rightmost column occupied by a car.
     *
     * @param car The car to analyze
     * @param width Width of the puzzle grid
```

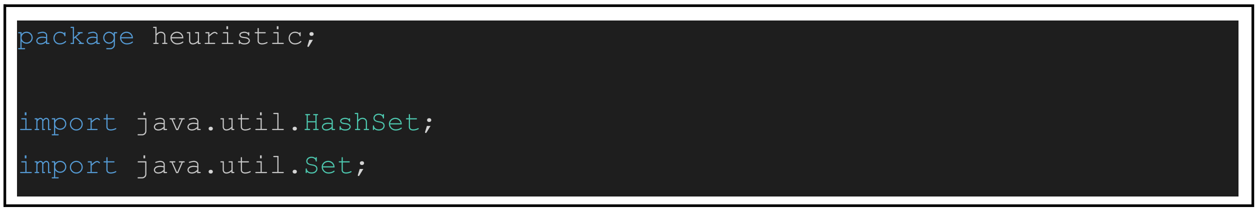


```
     * @return Index of the rightmost column
     */
    public static int findRightmostColumn(Car car, int width) {
        int rightmost = -1;
        for (int chunk = 0; chunk < car.bitmask.length; chunk++) {
            long bits = car.bitmask[chunk];
            if (bits == 0) continue; // Skip if no bits set in this
chunk

            for (int bit = 0; bit < 64; bit++) {
                if ((bits & (1L << bit)) != 0) {
                    int index = chunk * 64 + bit;
                    int col = index % width;
                    rightmost = Math.max(rightmost, col);
                }
            }
        }
        return rightmost;
    }

    /**
     * Determines the topmost row occupied by a car.
     *
     * @param car The car to analyze
     * @param width Width of the puzzle grid
     * @return Index of the topmost row
     */
    public static int findTopmostRow(Car car, int width) {
        int topmost = Integer.MAX_VALUE;
        for (int chunk = 0; chunk < car.bitmask.length; chunk++) {
            long bits = car.bitmask[chunk];
            if (bits == 0) continue; // Skip if no bits set in this
chunk

            for (int bit = 0; bit < 64; bit++) {
                if ((bits & (1L << bit)) != 0) {
                    int index = chunk * 64 + bit;
                    int row = index / width;
                    topmost = Math.min(topmost, row);
```

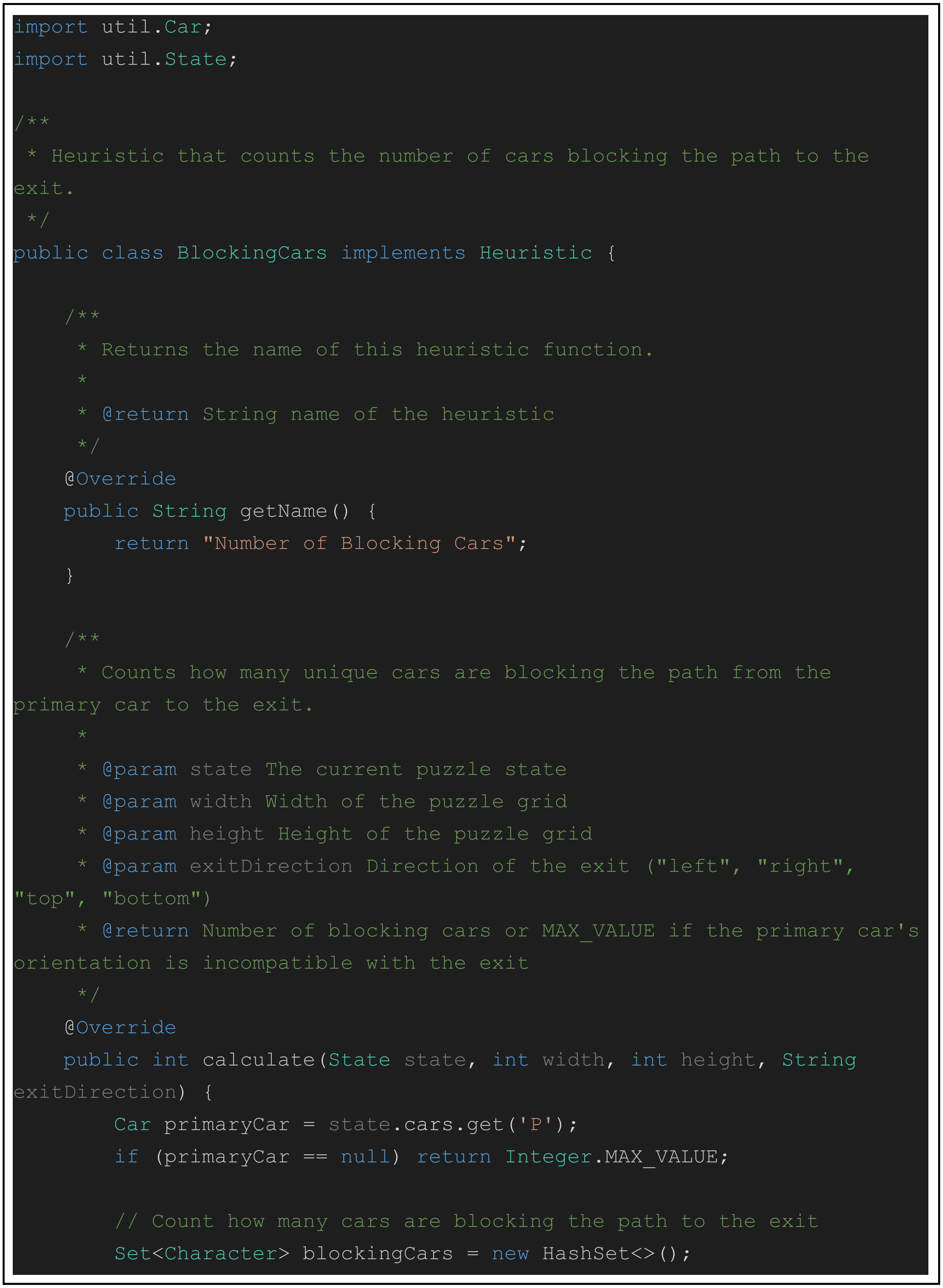


```java
                }
            }
        }
        return topmost;
    }

    /**
     * Determines the bottommost row occupied by a car.
     *
     * @param car The car to analyze
     * @param width Width of the puzzle grid
     * @return Index of the bottommost row
     */
    public static int findBottommostRow(Car car, int width) {
        int bottommost = -1;
        for (int chunk = 0; chunk < car.bitmask.length; chunk++) {
            long bits = car.bitmask[chunk];
            if (bits == 0) continue; // Skip if no bits set in this
chunk

            for (int bit = 0; bit < 64; bit++) {
                if ((bits & (1L << bit)) != 0) {
                    int index = chunk * 64 + bit;
                    int row = index / width;
                    bottommost = Math.max(bottommost, row);
                }
            }
        }
        return bottommost;
    }
}
```

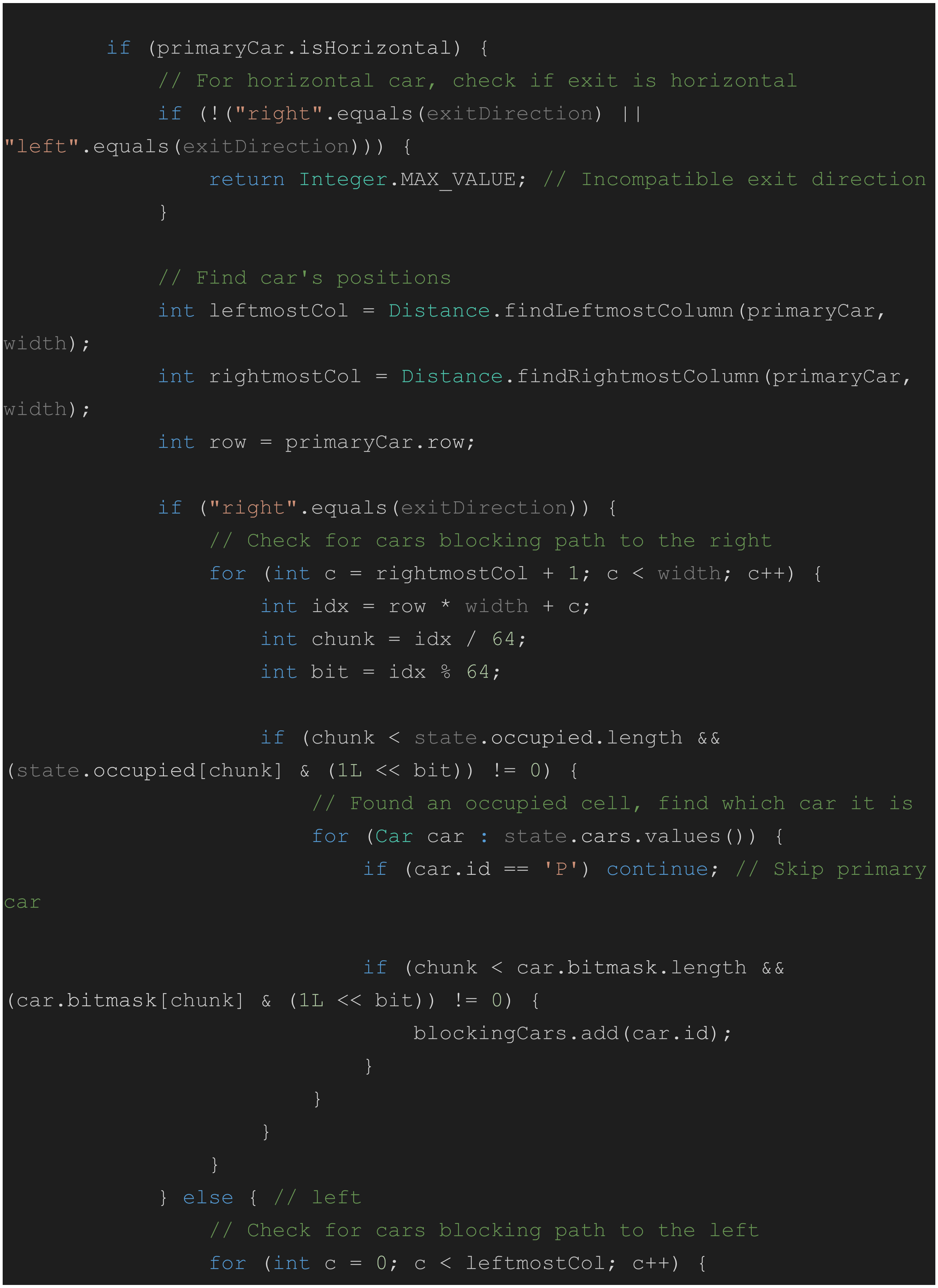
K. BlockingCars



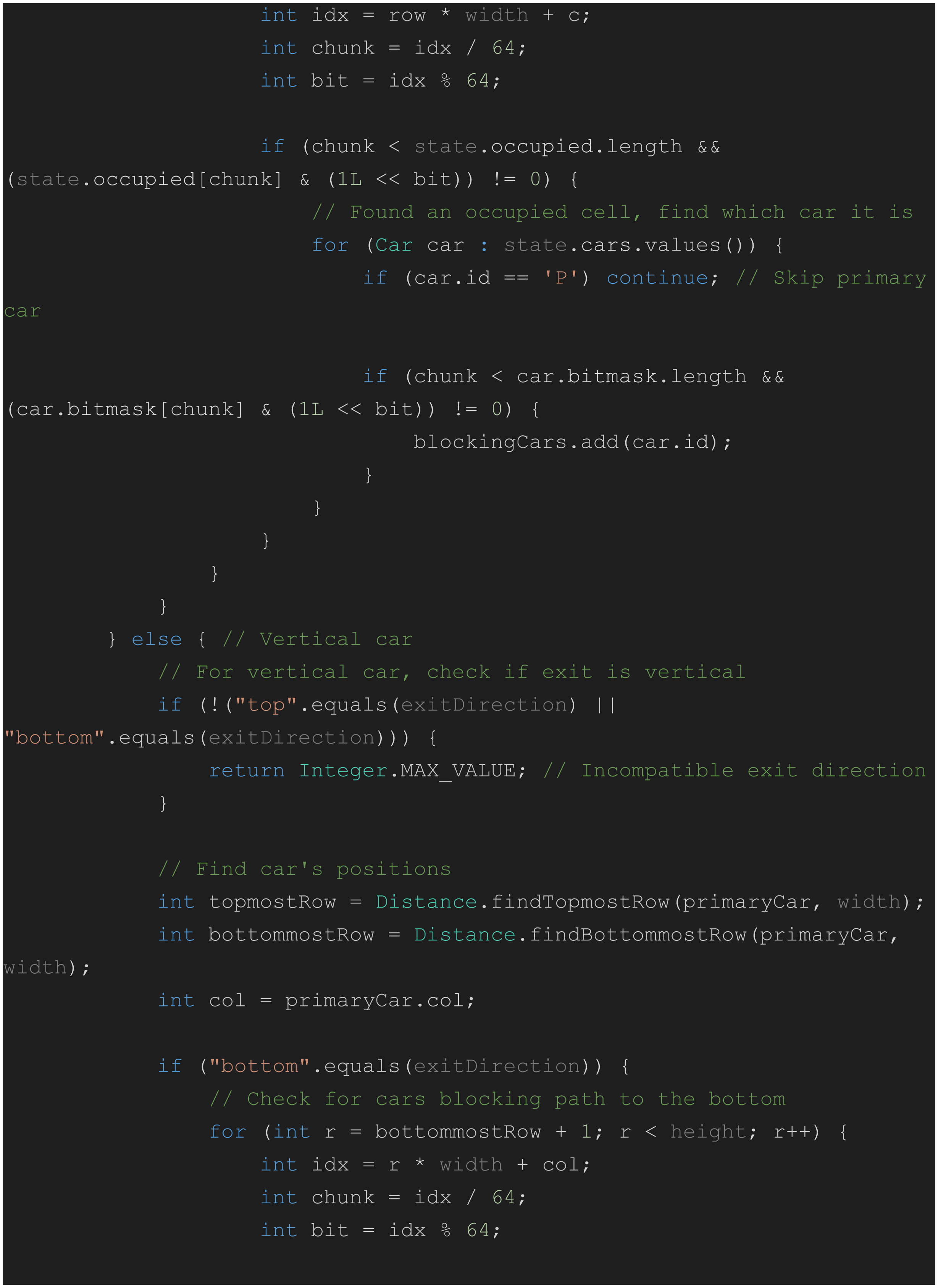
```java
package heuristic;

import java.util.HashSet;
import java.util.Set;
```



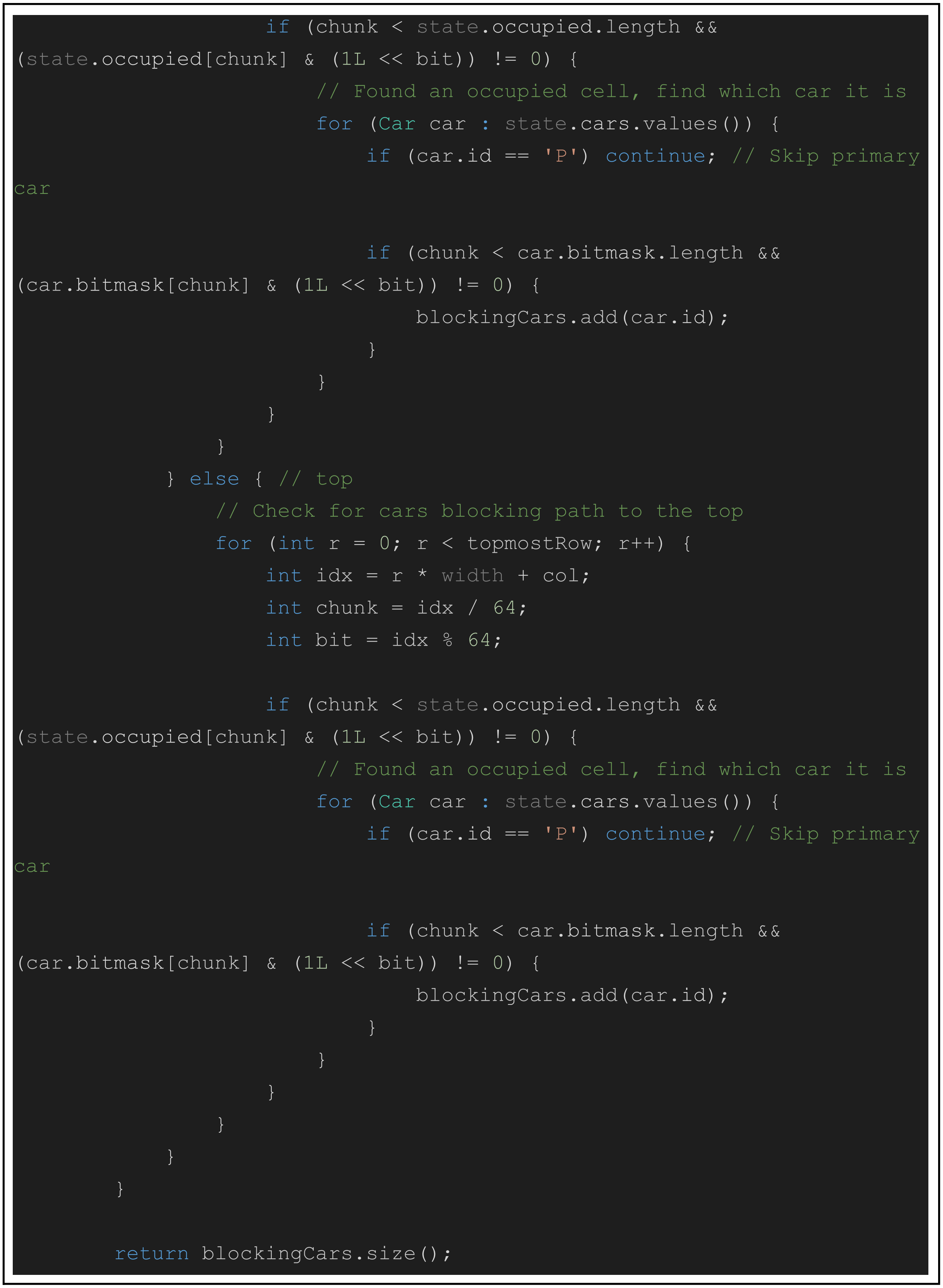
```
import util.Car;
import util.State;

/**
 * Heuristic that counts the number of cars blocking the path to the exit.
 */
public class BlockingCars implements Heuristic {

    /**
     * Returns the name of this heuristic function.
     *
     * @return String name of the heuristic
     */
    @Override
    public String getName() {
        return "Number of Blocking Cars";
    }

    /**
     * Counts how many unique cars are blocking the path from the primary car to the exit.
     *
     * @param state The current puzzle state
     * @param width Width of the puzzle grid
     * @param height Height of the puzzle grid
     * @param exitDirection Direction of the exit ("left", "right", "top", "bottom")
     * @return Number of blocking cars or MAX_VALUE if the primary car's orientation is incompatible with the exit
     */
    @Override
    public int calculate(State state, int width, int height, String exitDirection) {
        Car primaryCar = state.cars.get('P');
        if (primaryCar == null) return Integer.MAX_VALUE;

        // Count how many cars are blocking the path to the exit
        Set<Character> blockingCars = new HashSet<>();
```



```
        if (primaryCar.isHorizontal) {
            // For horizontal car, check if exit is horizontal
            if (!("right".equals(exitDirection) ||
"left".equals(exitDirection))) {
                return Integer.MAX_VALUE; // Incompatible exit direction
            }

            // Find car's positions
            int leftmostCol = Distance.findLeftmostColumn(primaryCar,
width);
            int rightmostCol = Distance.findRightmostColumn(primaryCar,
width);
            int row = primaryCar.row;

            if ("right".equals(exitDirection)) {
                // Check for cars blocking path to the right
                for (int c = rightmostCol + 1; c < width; c++) {
                    int idx = row * width + c;
                    int chunk = idx / 64;
                    int bit = idx % 64;

                    if (chunk < state.occupied.length &&
(state.occupied[chunk] & (1L << bit)) != 0) {
                        // Found an occupied cell, find which car it is
                        for (Car car : state.cars.values()) {
                            if (car.id == 'P') continue; // Skip primary
car

                            if (chunk < car.bitmask.length &&
(car.bitmask[chunk] & (1L << bit)) != 0) {
                                blockingCars.add(car.id);
                            }
                        }
                    }
                }
            } else { // left
                // Check for cars blocking path to the left
                for (int c = 0; c < leftmostCol; c++) {
```



```
                    int idx = row * width + c;
                    int chunk = idx / 64;
                    int bit = idx % 64;

                    if (chunk < state.occupied.length &&
(state.occupied[chunk] & (1L << bit)) != 0) {
                        // Found an occupied cell, find which car it is
                        for (Car car : state.cars.values()) {
                            if (car.id == 'P') continue; // Skip primary
car

                            if (chunk < car.bitmask.length &&
(car.bitmask[chunk] & (1L << bit)) != 0) {
                                blockingCars.add(car.id);
                            }
                        }
                    }
                }
            }
        } else { // Vertical car
            // For vertical car, check if exit is vertical
            if (!("top".equals(exitDirection) ||
"bottom".equals(exitDirection))) {
                return Integer.MAX_VALUE; // Incompatible exit direction
            }

            // Find car's positions
            int topmostRow = Distance.findTopmostRow(primaryCar, width);
            int bottommostRow = Distance.findBottommostRow(primaryCar,
width);
            int col = primaryCar.col;

            if ("bottom".equals(exitDirection)) {
                // Check for cars blocking path to the bottom
                for (int r = bottommostRow + 1; r < height; r++) {
                    int idx = r * width + col;
                    int chunk = idx / 64;
                    int bit = idx % 64;
```



```
                    if (chunk < state.occupied.length &&
(state.occupied[chunk] & (1L << bit)) != 0) {
                        // Found an occupied cell, find which car it is
                        for (Car car : state.cars.values()) {
                            if (car.id == 'P') continue; // Skip primary
car

                            if (chunk < car.bitmask.length &&
(car.bitmask[chunk] & (1L << bit)) != 0) {
                                blockingCars.add(car.id);
                            }
                        }
                    }
                }
            } else { // top
                // Check for cars blocking path to the top
                for (int r = 0; r < topmostRow; r++) {
                    int idx = r * width + col;
                    int chunk = idx / 64;
                    int bit = idx % 64;

                    if (chunk < state.occupied.length &&
(state.occupied[chunk] & (1L << bit)) != 0) {
                        // Found an occupied cell, find which car it is
                        for (Car car : state.cars.values()) {
                            if (car.id == 'P') continue; // Skip primary
car

                            if (chunk < car.bitmask.length &&
(car.bitmask[chunk] & (1L << bit)) != 0) {
                                blockingCars.add(car.id);
                            }
                        }
                    }
                }
            }
        }

        return blockingCars.size();
```

```
    }
}
```

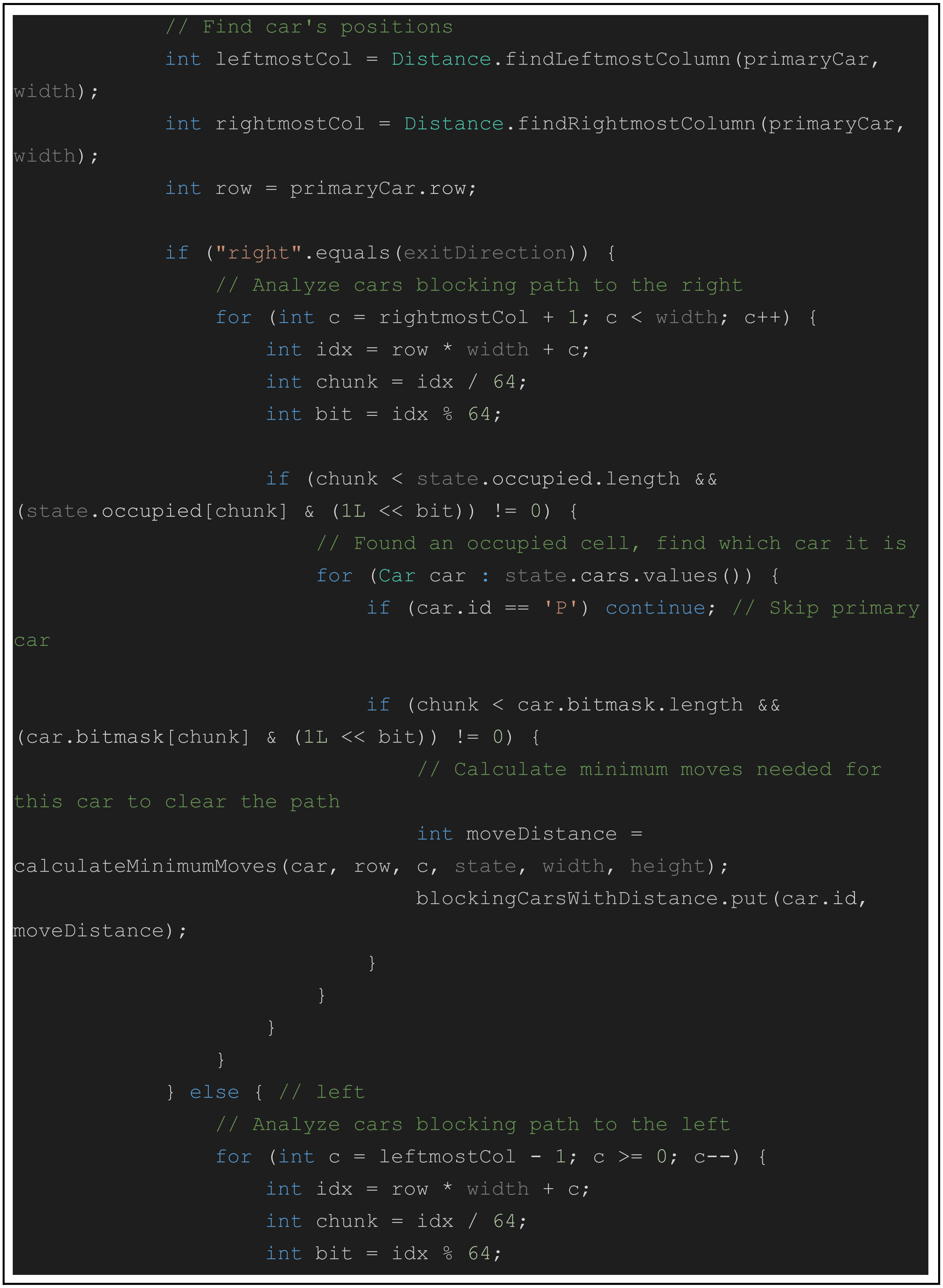
L. CombinedHeuristic

```
package heuristic;

import util.State;

/**
 * Combined heuristic that considers both distance to exit and blocking
cars.
 */
public class CombinedHeuristic implements Heuristic {

    /**
     * Returns the name of this heuristic function.
     *
     * @return String name of the heuristic
     */
    @Override
    public String getName() {
        return "Distance + Blocking Cars";
    }

    /**
     * Calculates a combined heuristic value using both Distance and
BlockingCars.
     * Adds the distance to exit and twice the number of blocking cars.
     *
     * @param state The current puzzle state
     * @param width Width of the puzzle grid
     * @param height Height of the puzzle grid
     * @param exitDirection Direction of the exit ("left", "right",
"top", "bottom")
     * @return Combined heuristic value or MAX_VALUE if state is invalid
     */
    @Override
    public int calculate(State state, int width, int height, String
exitDirection) {
        Distance distance = new Distance();
        BlockingCars blockingCars = new BlockingCars();

        int distValue = distance.calculate(state, width, height,
exitDirection);
        int blockingValue = blockingCars.calculate(state, width, height,
exitDirection);

        // If either heuristic returns MAX_VALUE, the state is invalid
        if (distValue == Integer.MAX_VALUE || blockingValue ==
Integer.MAX_VALUE) {
            return Integer.MAX_VALUE;
        }

        // Combine both heuristics - distance plus 2x the number of
blocking cars
        // The multiplier can be adjusted for different behavior
        return distValue + (2 * blockingValue);
    }
}
```

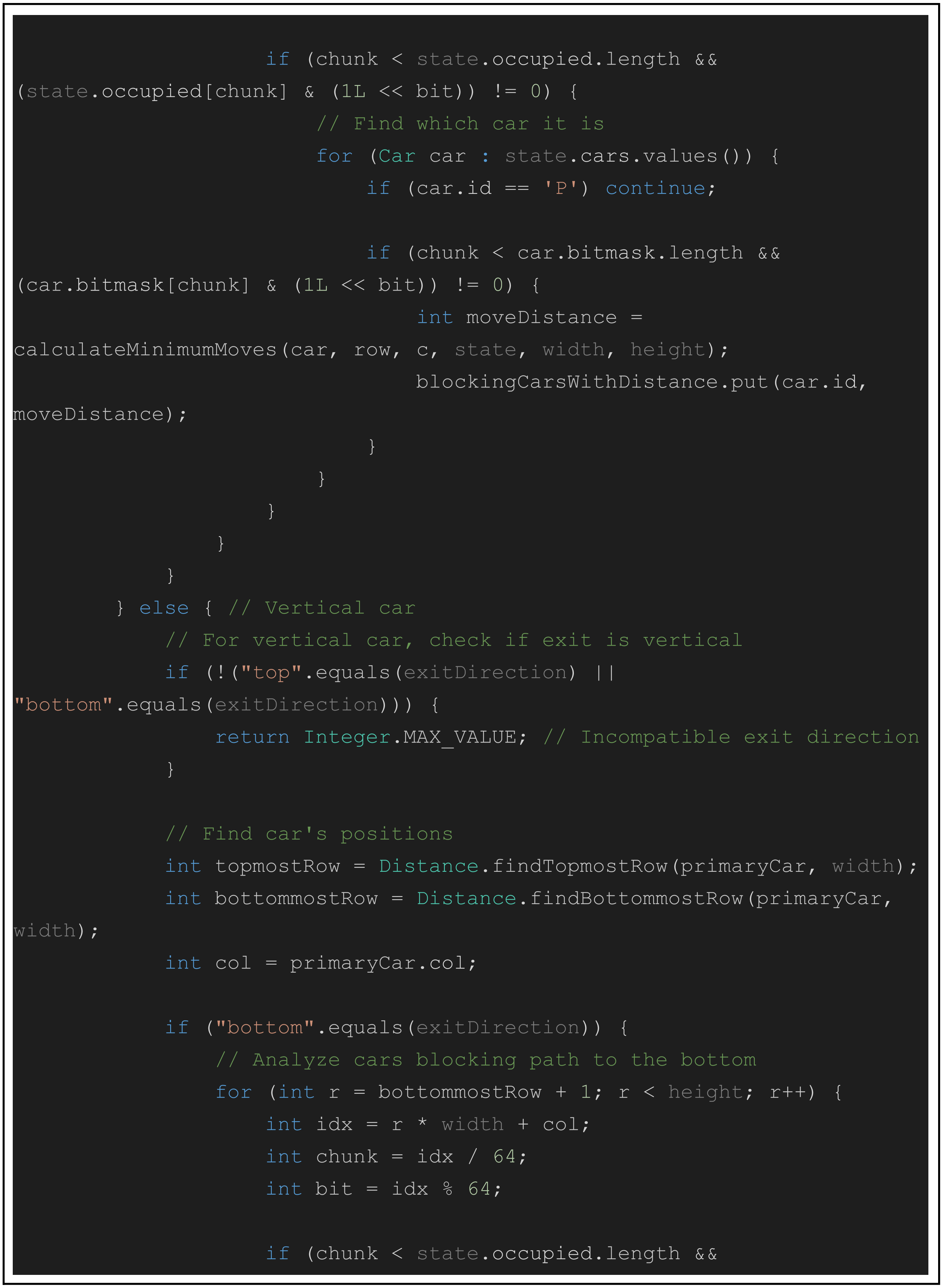
M. BlockingCarDistance

```
package heuristic;

import java.util.*;
import util.Car;
import util.State;

/**
 * Heuristic that considers distance to exit plus the minimum moves
needed to clear blocking cars.
 */
public class BlockingCarDistance implements Heuristic {

    /**
```

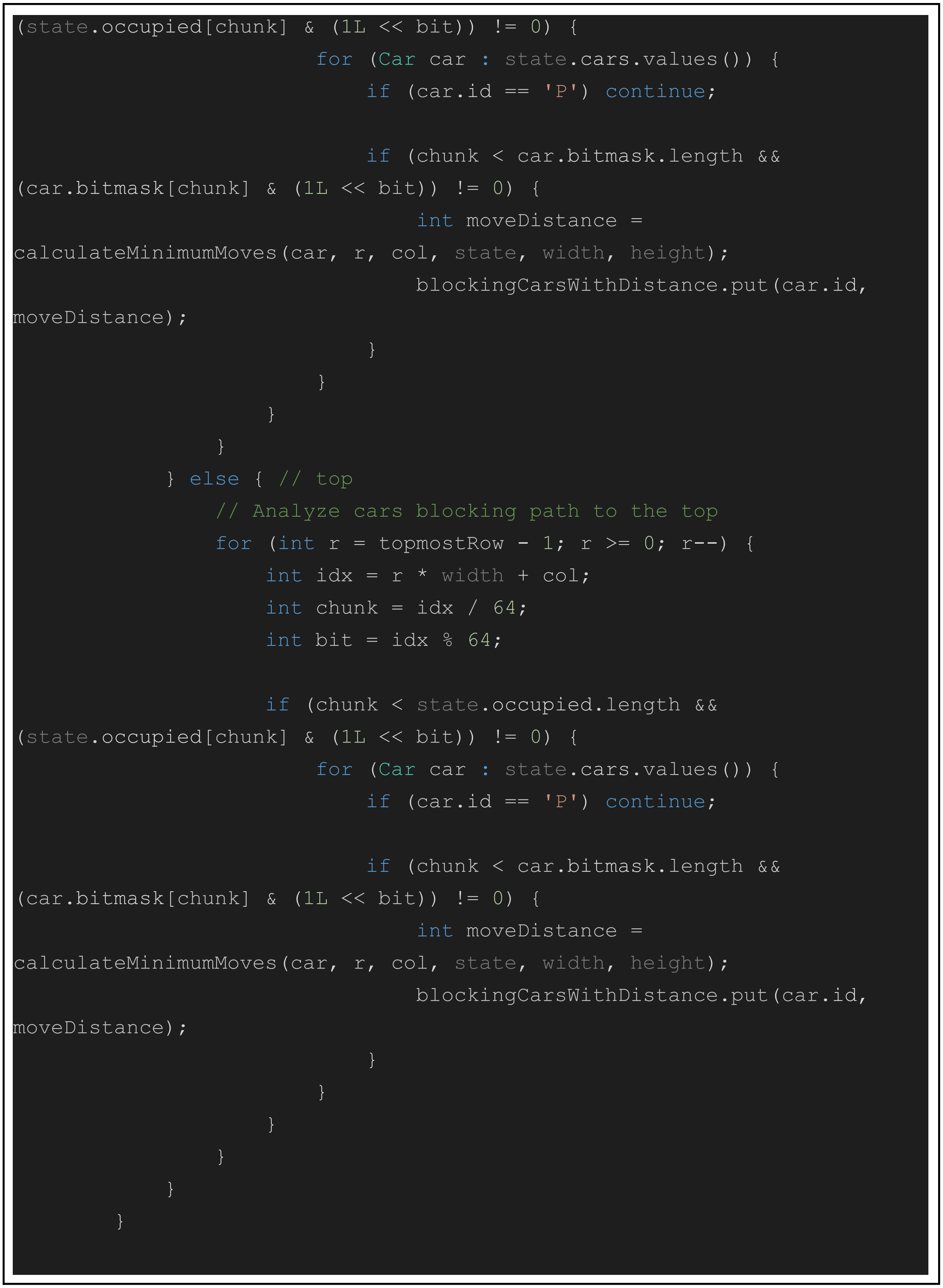
```
     * Returns the name of this heuristic function.
     *
     * @return String name of the heuristic
     */
    @Override
    public String getName() {
        return "Blocking Car Distance";
    }

    /**
     * Calculates the heuristic value by combining distance to exit and
the minimum moves
     * needed for blocking cars to clear the path.
     *
     * @param state The current puzzle state
     * @param width Width of the puzzle grid
     * @param height Height of the puzzle grid
     * @param exitDirection Direction of the exit ("left", "right",
"top", "bottom")
     * @return Combined heuristic value or MAX_VALUE if car orientation
is incompatible with exit
     */
    @Override
    public int calculate(State state, int width, int height, String
exitDirection) {
        Car primaryCar = state.cars.get('P');
        if (primaryCar == null) return Integer.MAX_VALUE;

        // Get the blocking cars
        Map<Character, Integer> blockingCarsWithDistance = new
HashMap<>();

        if (primaryCar.isHorizontal) {
            // For horizontal car, check if exit is horizontal
            if (!("right".equals(exitDirection) ||
"left".equals(exitDirection))) {
                return Integer.MAX_VALUE; // Incompatible exit direction
            }
```



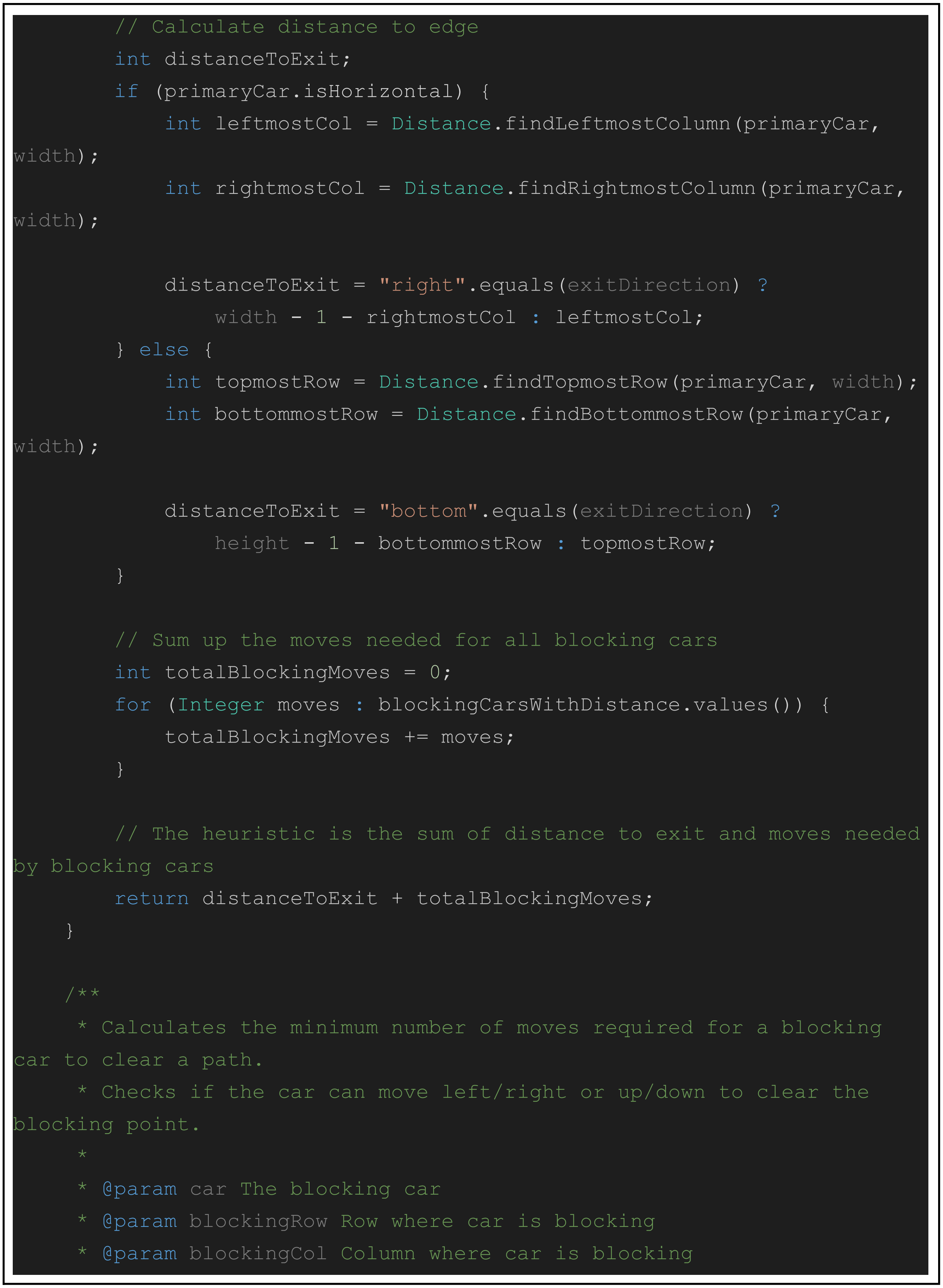
```
            // Find car's positions
            int leftmostCol = Distance.findLeftmostColumn(primaryCar,
width);
            int rightmostCol = Distance.findRightmostColumn(primaryCar,
width);
            int row = primaryCar.row;

            if ("right".equals(exitDirection)) {
                // Analyze cars blocking path to the right
                for (int c = rightmostCol + 1; c < width; c++) {
                    int idx = row * width + c;
                    int chunk = idx / 64;
                    int bit = idx % 64;

                    if (chunk < state.occupied.length &&
(state.occupied[chunk] & (1L << bit)) != 0) {
                        // Found an occupied cell, find which car it is
                        for (Car car : state.cars.values()) {
                            if (car.id == 'P') continue; // Skip primary
car

                            if (chunk < car.bitmask.length &&
(car.bitmask[chunk] & (1L << bit)) != 0) {
                                // Calculate minimum moves needed for
this car to clear the path
                                int moveDistance =
calculateMinimumMoves(car, row, c, state, width, height);
                                blockingCarsWithDistance.put(car.id,
moveDistance);
                            }
                        }
                    }
                }
            } else { // left
                // Analyze cars blocking path to the left
                for (int c = leftmostCol - 1; c >= 0; c--) {
                    int idx = row * width + c;
                    int chunk = idx / 64;
                    int bit = idx % 64;
```



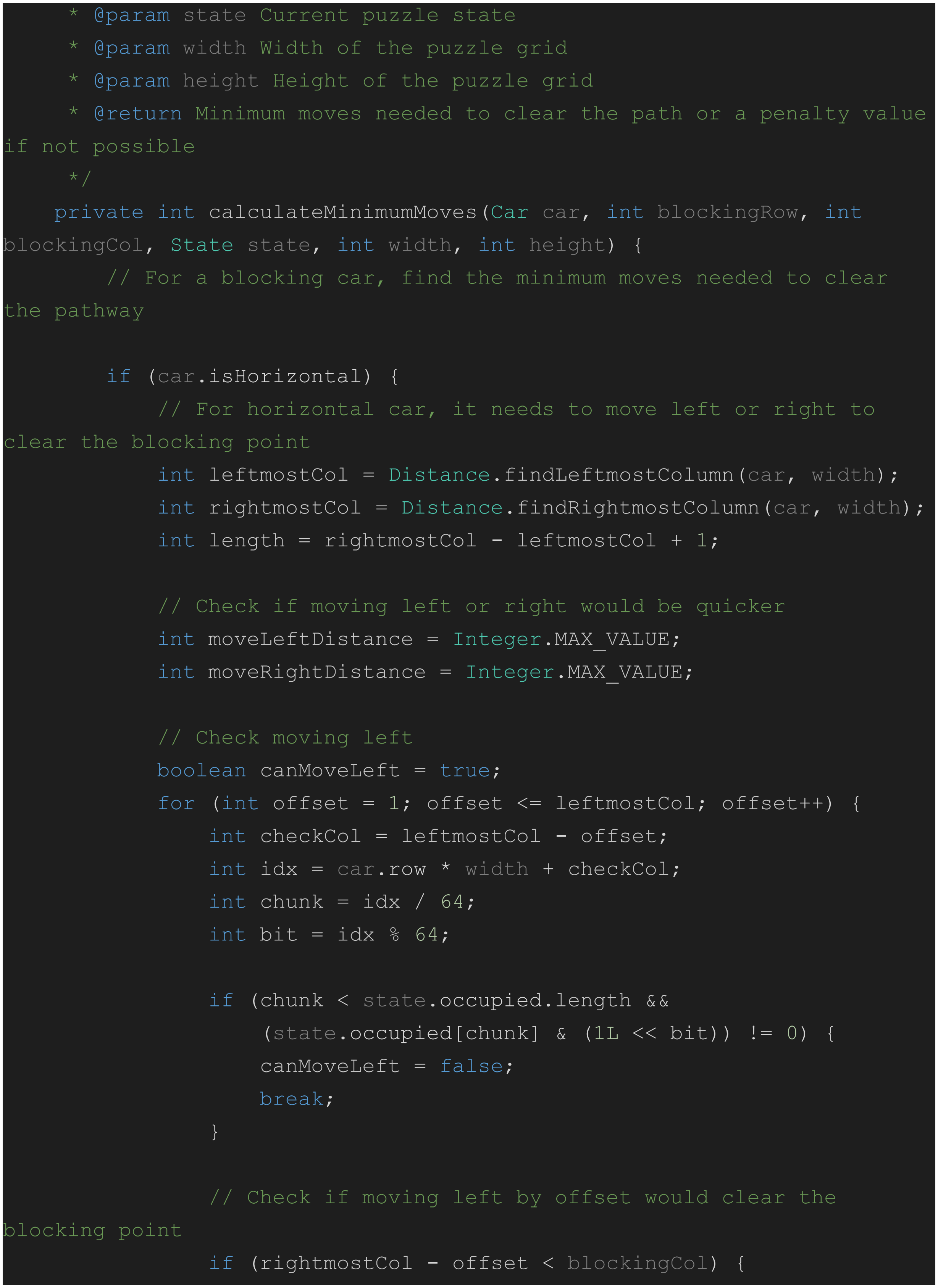
```
                    if (chunk < state.occupied.length &&
(state.occupied[chunk] & (1L << bit)) != 0) {
                        // Find which car it is
                        for (Car car : state.cars.values()) {
                            if (car.id == 'P') continue;

                            if (chunk < car.bitmask.length &&
(car.bitmask[chunk] & (1L << bit)) != 0) {
                                int moveDistance =
calculateMinimumMoves(car, row, c, state, width, height);
                                blockingCarsWithDistance.put(car.id,
moveDistance);
                            }
                        }
                    }
                }
            }
        } else { // Vertical car
            // For vertical car, check if exit is vertical
            if (!("top".equals(exitDirection) ||
"bottom".equals(exitDirection))) {
                return Integer.MAX_VALUE; // Incompatible exit direction
            }

            // Find car's positions
            int topmostRow = Distance.findTopmostRow(primaryCar, width);
            int bottommostRow = Distance.findBottommostRow(primaryCar,
width);
            int col = primaryCar.col;

            if ("bottom".equals(exitDirection)) {
                // Analyze cars blocking path to the bottom
                for (int r = bottommostRow + 1; r < height; r++) {
                    int idx = r * width + col;
                    int chunk = idx / 64;
                    int bit = idx % 64;

                    if (chunk < state.occupied.length &&
```



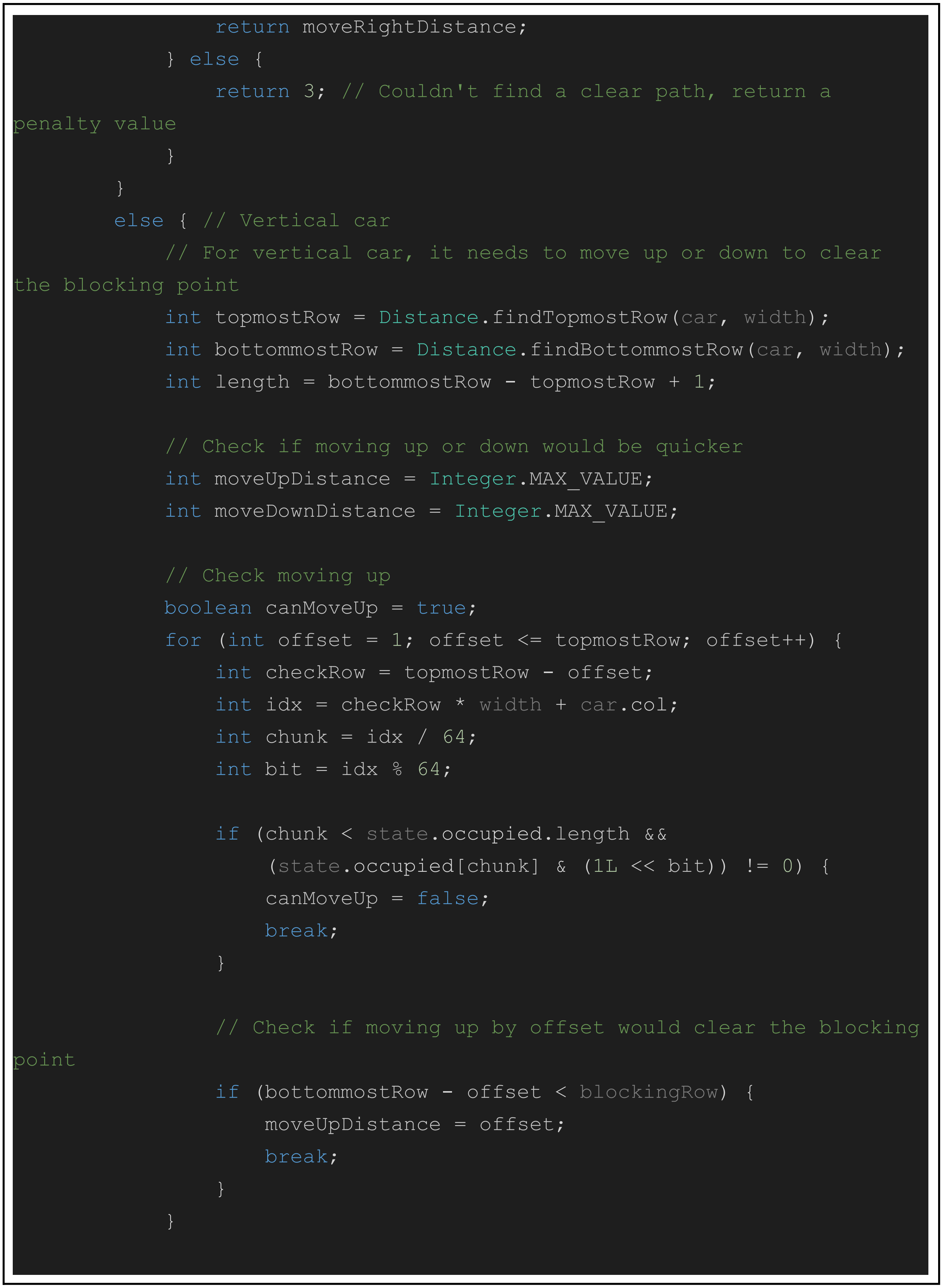
```
(state.occupied[chunk] & (1L << bit)) != 0) {
                        for (Car car : state.cars.values()) {
                            if (car.id == 'P') continue;

                            if (chunk < car.bitmask.length &&
(car.bitmask[chunk] & (1L << bit)) != 0) {
                                int moveDistance =
calculateMinimumMoves(car, r, col, state, width, height);
                                blockingCarsWithDistance.put(car.id,
moveDistance);
                            }
                        }
                    }
                }
            } else { // top
                // Analyze cars blocking path to the top
                for (int r = topmostRow - 1; r >= 0; r--) {
                    int idx = r * width + col;
                    int chunk = idx / 64;
                    int bit = idx % 64;

                    if (chunk < state.occupied.length &&
(state.occupied[chunk] & (1L << bit)) != 0) {
                        for (Car car : state.cars.values()) {
                            if (car.id == 'P') continue;

                            if (chunk < car.bitmask.length &&
(car.bitmask[chunk] & (1L << bit)) != 0) {
                                int moveDistance =
calculateMinimumMoves(car, r, col, state, width, height);
                                blockingCarsWithDistance.put(car.id,
moveDistance);
                            }
                        }
                    }
                }
            }
        }
```



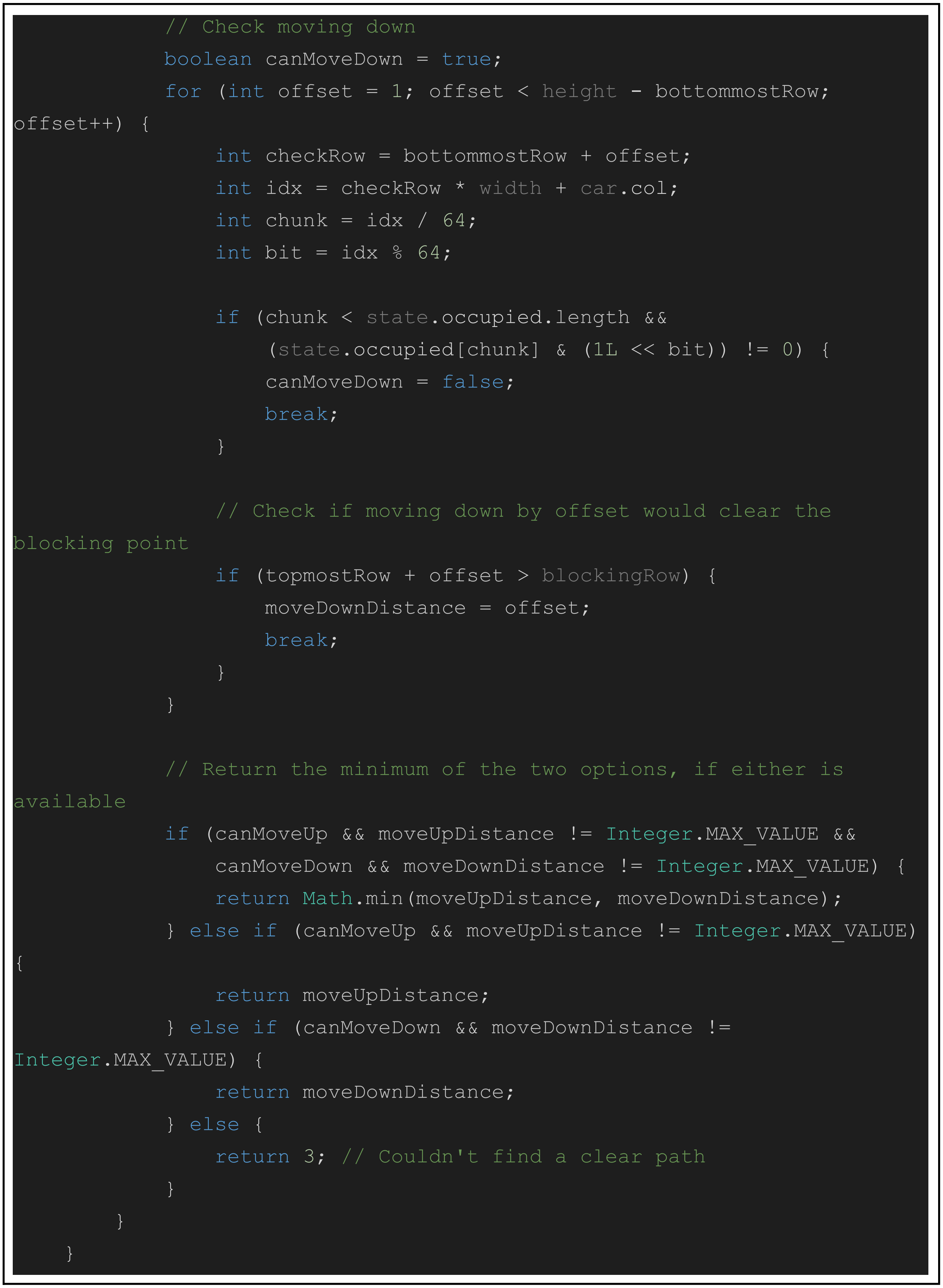
```
        // Calculate distance to edge
        int distanceToExit;
        if (primaryCar.isHorizontal) {
            int leftmostCol = Distance.findLeftmostColumn(primaryCar, width);
            int rightmostCol = Distance.findRightmostColumn(primaryCar, width);

            distanceToExit = "right".equals(exitDirection) ?
                width - 1 - rightmostCol : leftmostCol;
        } else {
            int topmostRow = Distance.findTopmostRow(primaryCar, width);
            int bottommostRow = Distance.findBottommostRow(primaryCar, width);

            distanceToExit = "bottom".equals(exitDirection) ?
                height - 1 - bottommostRow : topmostRow;
        }

        // Sum up the moves needed for all blocking cars
        int totalBlockingMoves = 0;
        for (Integer moves : blockingCarsWithDistance.values()) {
            totalBlockingMoves += moves;
        }

        // The heuristic is the sum of distance to exit and moves needed by blocking cars
        return distanceToExit + totalBlockingMoves;
    }

    /**
     * Calculates the minimum number of moves required for a blocking car to clear a path.
     * Checks if the car can move left/right or up/down to clear the blocking point.
     *
     * @param car The blocking car
     * @param blockingRow Row where car is blocking
     * @param blockingCol Column where car is blocking
```



```
     * @param state Current puzzle state
     * @param width Width of the puzzle grid
     * @param height Height of the puzzle grid
     * @return Minimum moves needed to clear the path or a penalty value
if not possible
     */
    private int calculateMinimumMoves(Car car, int blockingRow, int
blockingCol, State state, int width, int height) {
        // For a blocking car, find the minimum moves needed to clear
the pathway

        if (car.isHorizontal) {
            // For horizontal car, it needs to move left or right to
clear the blocking point
            int leftmostCol = Distance.findLeftmostColumn(car, width);
            int rightmostCol = Distance.findRightmostColumn(car, width);
            int length = rightmostCol - leftmostCol + 1;

            // Check if moving left or right would be quicker
            int moveLeftDistance = Integer.MAX_VALUE;
            int moveRightDistance = Integer.MAX_VALUE;

            // Check moving left
            boolean canMoveLeft = true;
            for (int offset = 1; offset <= leftmostCol; offset++) {
                int checkCol = leftmostCol - offset;
                int idx = car.row * width + checkCol;
                int chunk = idx / 64;
                int bit = idx % 64;

                if (chunk < state.occupied.length &&
                    (state.occupied[chunk] & (1L << bit)) != 0) {
                    canMoveLeft = false;
                    break;
                }

                // Check if moving left by offset would clear the
blocking point
                if (rightmostCol - offset < blockingCol) {
                    moveLeftDistance = offset;
                    break;
                }
            }

            // Check moving right
            boolean canMoveRight = true;
            for (int offset = 1; offset < width - rightmostCol;
offset++) {
                int checkCol = rightmostCol + offset;
                int idx = car.row * width + checkCol;
                int chunk = idx / 64;
                int bit = idx % 64;

                if (chunk < state.occupied.length &&
                    (state.occupied[chunk] & (1L << bit)) != 0) {
                    canMoveRight = false;
                    break;
                }

                // Check if moving right by offset would clear the
blocking point
                if (leftmostCol + offset > blockingCol) {
                    moveRightDistance = offset;
                    break;
                }
            }

            // Return the minimum of the two options, if either is
available
            if (canMoveLeft && moveLeftDistance != Integer.MAX_VALUE &&
                canMoveRight && moveRightDistance != Integer.MAX_VALUE)
{
                return Math.min(moveLeftDistance, moveRightDistance);
            } else if (canMoveLeft && moveLeftDistance !=
Integer.MAX_VALUE) {
                return moveLeftDistance;
            } else if (canMoveRight && moveRightDistance !=
Integer.MAX_VALUE) {
```



```
                return moveRightDistance;
            } else {
                return 3; // Couldn't find a clear path, return a
penalty value
            }
        }
        else { // Vertical car
            // For vertical car, it needs to move up or down to clear
the blocking point
            int topmostRow = Distance.findTopmostRow(car, width);
            int bottommostRow = Distance.findBottommostRow(car, width);
            int length = bottommostRow - topmostRow + 1;

            // Check if moving up or down would be quicker
            int moveUpDistance = Integer.MAX_VALUE;
            int moveDownDistance = Integer.MAX_VALUE;

            // Check moving up
            boolean canMoveUp = true;
            for (int offset = 1; offset <= topmostRow; offset++) {
                int checkRow = topmostRow - offset;
                int idx = checkRow * width + car.col;
                int chunk = idx / 64;
                int bit = idx % 64;

                if (chunk < state.occupied.length &&
                    (state.occupied[chunk] & (1L << bit)) != 0) {
                    canMoveUp = false;
                    break;
                }

                // Check if moving up by offset would clear the blocking
point
                if (bottommostRow - offset < blockingRow) {
                    moveUpDistance = offset;
                    break;
                }
            }
```

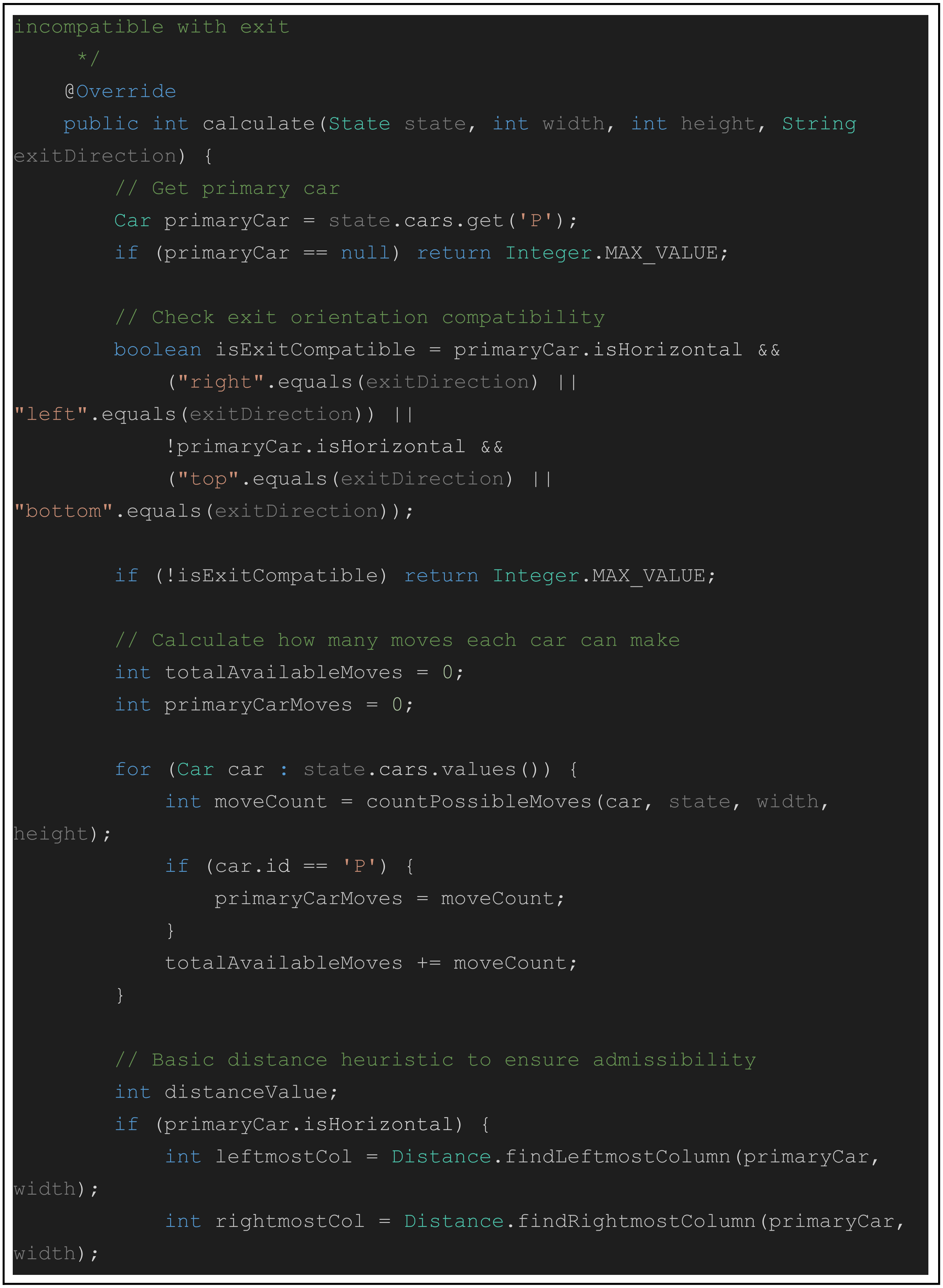


```
            // Check moving down
            boolean canMoveDown = true;
            for (int offset = 1; offset < height - bottommostRow;
offset++) {
                int checkRow = bottommostRow + offset;
                int idx = checkRow * width + car.col;
                int chunk = idx / 64;
                int bit = idx % 64;

                if (chunk < state.occupied.length &&
                    (state.occupied[chunk] & (1L << bit)) != 0) {
                    canMoveDown = false;
                    break;
                }

                // Check if moving down by offset would clear the
blocking point
                if (topmostRow + offset > blockingRow) {
                    moveDownDistance = offset;
                    break;
                }
            }

            // Return the minimum of the two options, if either is
available
            if (canMoveUp && moveUpDistance != Integer.MAX_VALUE &&
                canMoveDown && moveDownDistance != Integer.MAX_VALUE) {
                return Math.min(moveUpDistance, moveDownDistance);
            } else if (canMoveUp && moveUpDistance != Integer.MAX_VALUE)
{
                return moveUpDistance;
            } else if (canMoveDown && moveDownDistance !=
Integer.MAX_VALUE) {
                return moveDownDistance;
            } else {
                return 3; // Couldn't find a clear path
            }
        }
    }
```

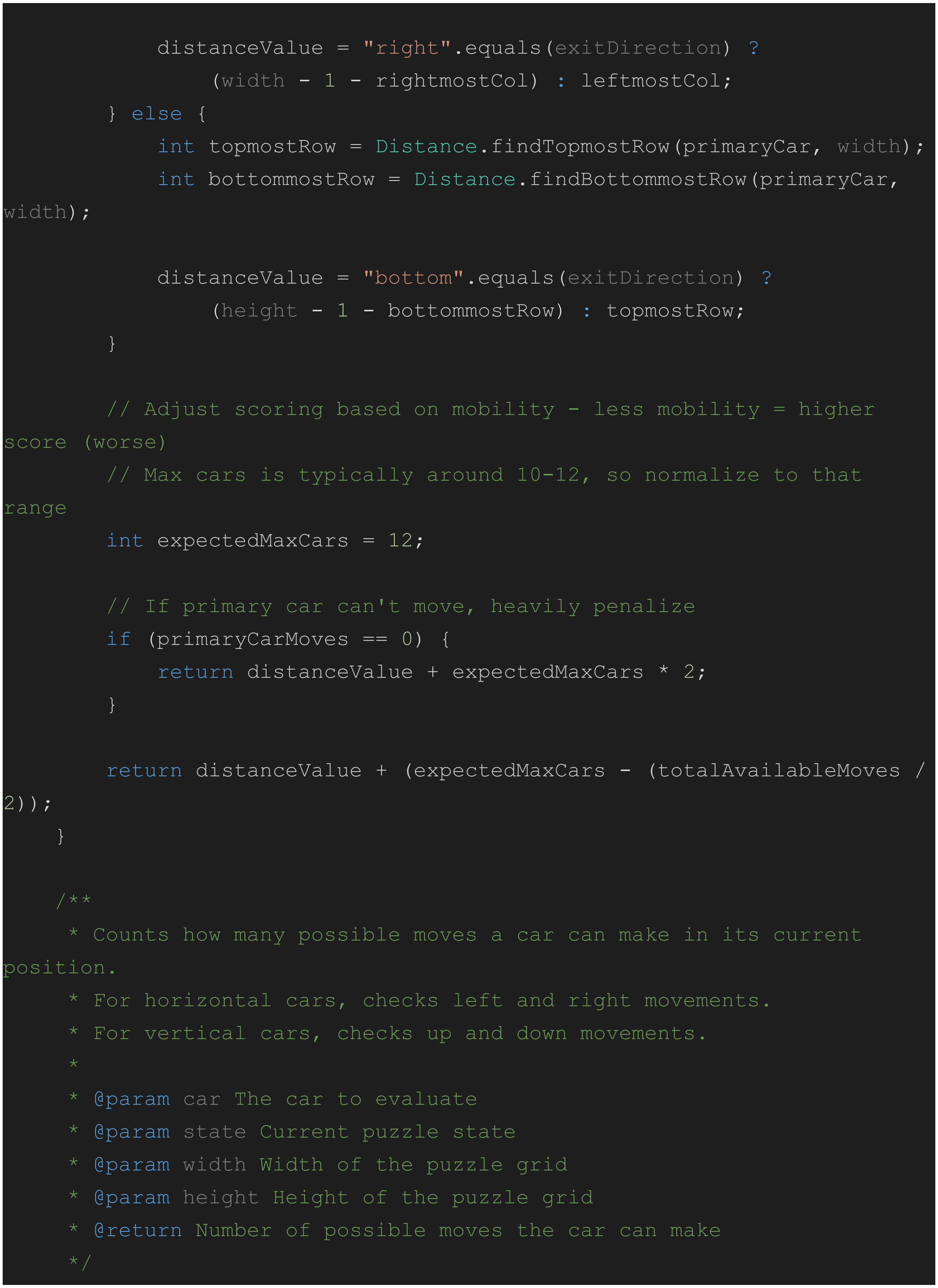
```
}
```

N. MobiltyScore

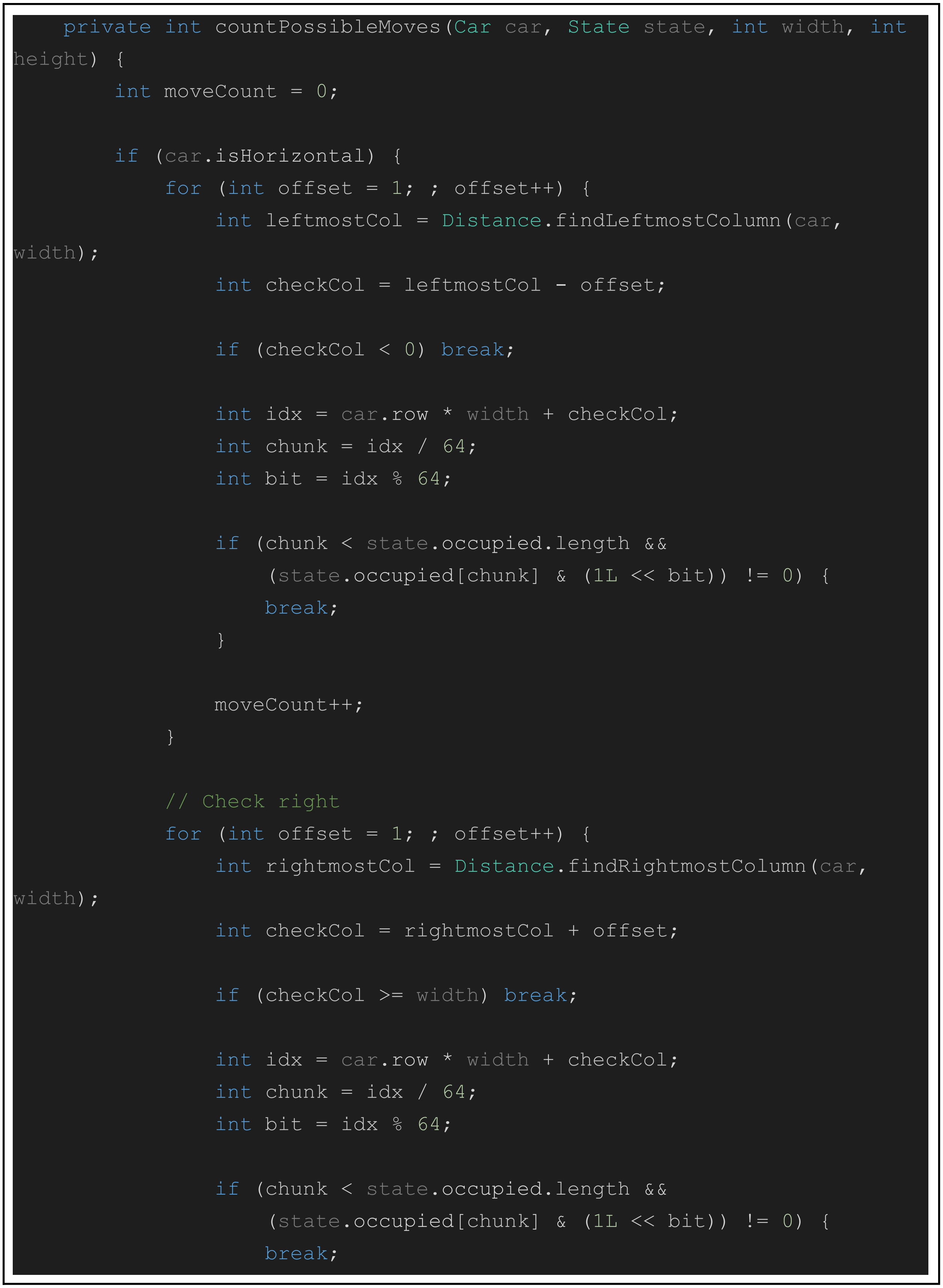
```
package heuristic;

import java.util.*;
import util.Car;
import util.State;

/**
 * Heuristic that evaluates states based on car mobility (available
moves).
 */
public class MobilityScore implements Heuristic {

    /**
     * Returns the name of this heuristic function.
     *
     * @return String name of the heuristic
     */
    @Override
    public String getName() {
        return "Mobility Score";
    }

    /**
     * Calculates a heuristic value based on mobility of cars in the
puzzle.
     * Higher mobility (more available moves) results in a lower
heuristic value.
     *
     * @param state The current puzzle state
     * @param width Width of the puzzle grid
     * @param height Height of the puzzle grid
     * @param exitDirection Direction of the exit ("left", "right",
"top", "bottom")
     * @return Heuristic value based on car mobility or MAX_VALUE if
```



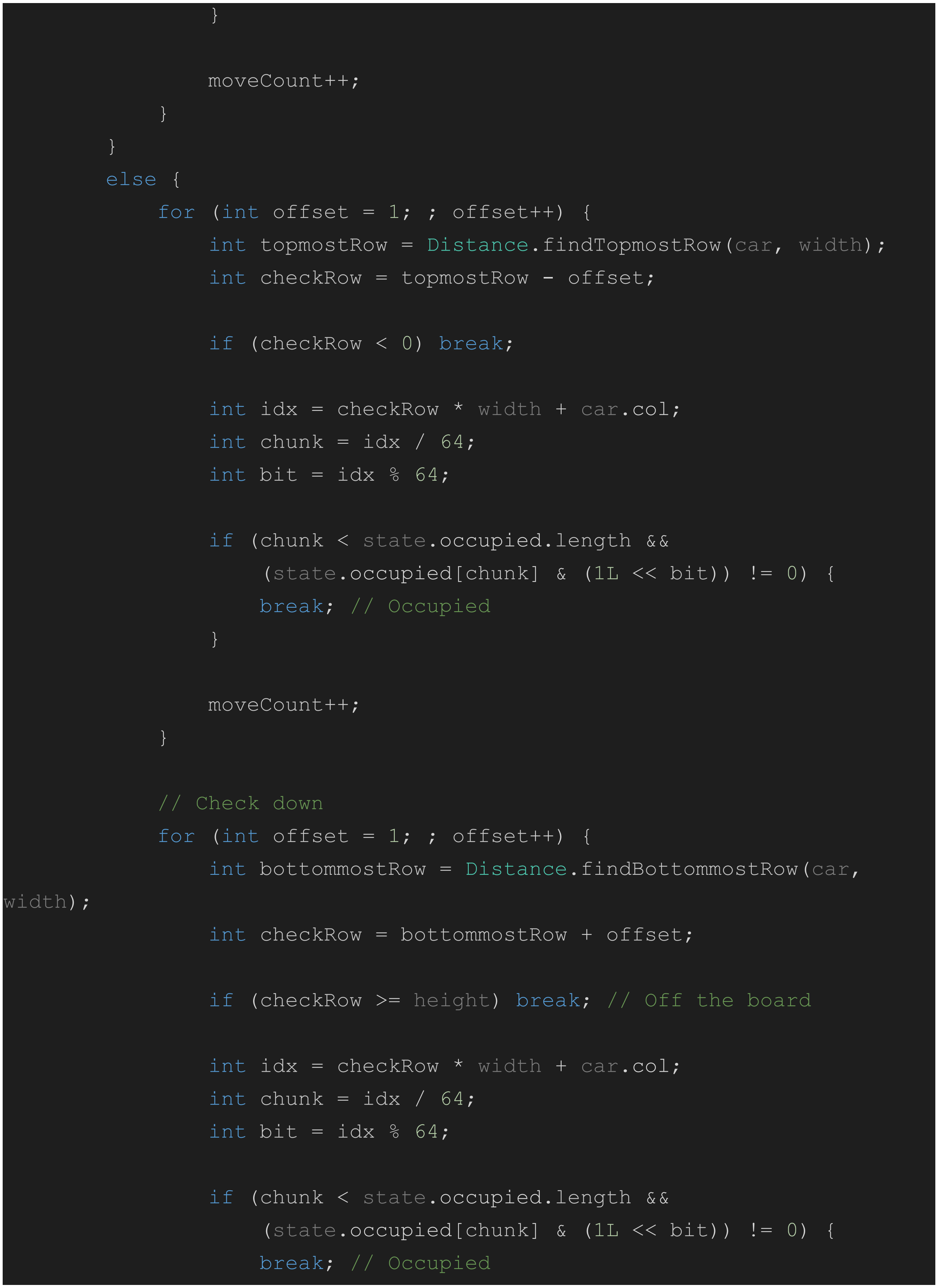
```
incompatible with exit
     */
    @Override
    public int calculate(State state, int width, int height, String
exitDirection) {
        // Get primary car
        Car primaryCar = state.cars.get('P');
        if (primaryCar == null) return Integer.MAX_VALUE;

        // Check exit orientation compatibility
        boolean isExitCompatible = primaryCar.isHorizontal &&
            ("right".equals(exitDirection) ||
"left".equals(exitDirection)) ||
            !primaryCar.isHorizontal &&
            ("top".equals(exitDirection) ||
"bottom".equals(exitDirection));

        if (!isExitCompatible) return Integer.MAX_VALUE;

        // Calculate how many moves each car can make
        int totalAvailableMoves = 0;
        int primaryCarMoves = 0;

        for (Car car : state.cars.values()) {
            int moveCount = countPossibleMoves(car, state, width,
height);
            if (car.id == 'P') {
                primaryCarMoves = moveCount;
            }
            totalAvailableMoves += moveCount;
        }

        // Basic distance heuristic to ensure admissibility
        int distanceValue;
        if (primaryCar.isHorizontal) {
            int leftmostCol = Distance.findLeftmostColumn(primaryCar,
width);
            int rightmostCol = Distance.findRightmostColumn(primaryCar,
width);
```



```
            distanceValue = "right".equals(exitDirection) ?
                (width - 1 - rightmostCol) : leftmostCol;
        } else {
            int topmostRow = Distance.findTopmostRow(primaryCar, width);
            int bottommostRow = Distance.findBottommostRow(primaryCar,
width);

            distanceValue = "bottom".equals(exitDirection) ?
                (height - 1 - bottommostRow) : topmostRow;
        }

        // Adjust scoring based on mobility - less mobility = higher
score (worse)
        // Max cars is typically around 10-12, so normalize to that
range
        int expectedMaxCars = 12;

        // If primary car can't move, heavily penalize
        if (primaryCarMoves == 0) {
            return distanceValue + expectedMaxCars * 2;
        }

        return distanceValue + (expectedMaxCars - (totalAvailableMoves /
2));
    }

    /**
     * Counts how many possible moves a car can make in its current
position.
     * For horizontal cars, checks left and right movements.
     * For vertical cars, checks up and down movements.
     *
     * @param car The car to evaluate
     * @param state Current puzzle state
     * @param width Width of the puzzle grid
     * @param height Height of the puzzle grid
     * @return Number of possible moves the car can make
     */
```



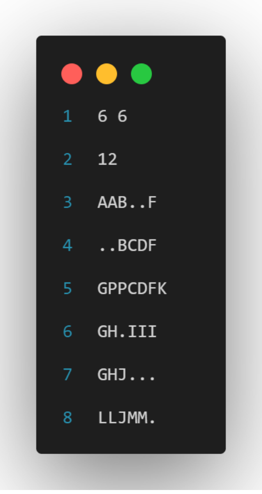
```
    private int countPossibleMoves(Car car, State state, int width, int
height) {
        int moveCount = 0;

        if (car.isHorizontal) {
            for (int offset = 1; ; offset++) {
                int leftmostCol = Distance.findLeftmostColumn(car,
width);
                int checkCol = leftmostCol - offset;

                if (checkCol < 0) break;

                int idx = car.row * width + checkCol;
                int chunk = idx / 64;
                int bit = idx % 64;

                if (chunk < state.occupied.length &&
                    (state.occupied[chunk] & (1L << bit)) != 0) {
                    break;
                }

                moveCount++;
            }

            // Check right
            for (int offset = 1; ; offset++) {
                int rightmostCol = Distance.findRightmostColumn(car,
width);
                int checkCol = rightmostCol + offset;

                if (checkCol >= width) break;

                int idx = car.row * width + checkCol;
                int chunk = idx / 64;
                int bit = idx % 64;

                if (chunk < state.occupied.length &&
                    (state.occupied[chunk] & (1L << bit)) != 0) {
                    break;
```

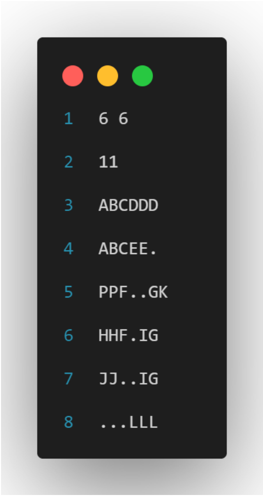
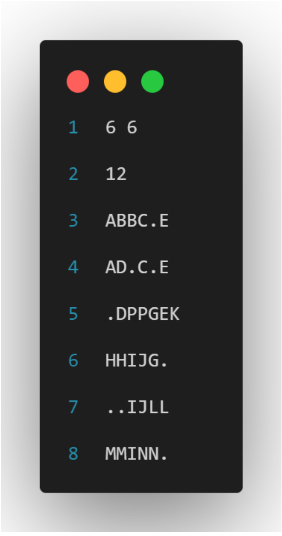
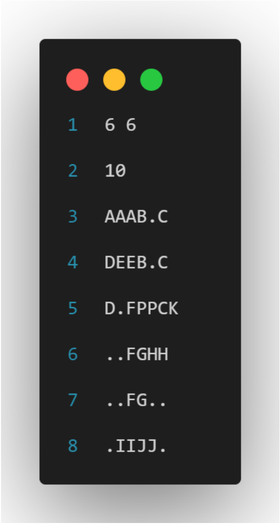


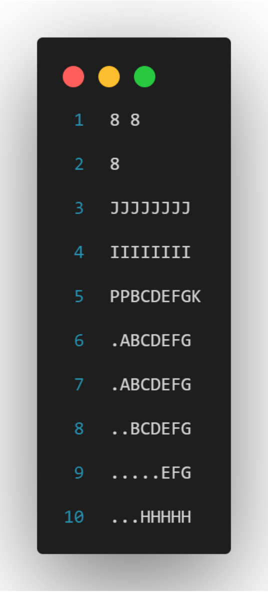
```
                }

                moveCount++;
            }
        }
        else {
            for (int offset = 1; ; offset++) {
                int topmostRow = Distance.findTopmostRow(car, width);
                int checkRow = topmostRow - offset;

                if (checkRow < 0) break;

                int idx = checkRow * width + car.col;
                int chunk = idx / 64;
                int bit = idx % 64;

                if (chunk < state.occupied.length &&
                    (state.occupied[chunk] & (1L << bit)) != 0) {
                    break; // Occupied
                }

                moveCount++;
            }

            // Check down
            for (int offset = 1; ; offset++) {
                int bottommostRow = Distance.findBottommostRow(car,
width);
                int checkRow = bottommostRow + offset;

                if (checkRow >= height) break; // Off the board

                int idx = checkRow * width + car.col;
                int chunk = idx / 64;
                int bit = idx % 64;

                if (chunk < state.occupied.length &&
                    (state.occupied[chunk] & (1L << bit)) != 0) {
                    break; // Occupied
                }

                moveCount++;
            }
        }

        return moveCount;
    }
}
```

BAB IV
PENGUJIAN

Akan digunakan lima test case yang berasal dari (1) asisten, (2) level intermediate dari Rush Hour Deluxe Edition, (3) level expert dari Rush Hour Deluxe Edition, (4) level grandmaster dari Rush Hour Deluxe Edition, dan (5) papan 8x8.

No.	Sumber	Input
1.	Asisten	 6 6 12 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.

2.	Level Intermediate dari Rush Hour Deluxe Edition	 6 6 11 ABCDDD ABCEE. PPF..GK HHF.IG JJ..IG ...LLL
3.	Level Expert dari Rush Hour Deluxe Edition	 6 6 12 ABBC.E AD.C.E .DPPGEK HHIJG. ..IJLL MMINN.
4.	Level Grandmaster dari Rush Hour Deluxe Edition	 6 6 10 AAAB.C DEEB.C D.FPPCK ..FGHH ..FG.. .IIJJ.

5.	Penulis	 8 8 8 JJJJJJJJ IIIIIIII PPBCDEFGK .ABCDEFG .ABCDEFG ..BCDEFG .....EFG ...HHHHH

A. Uniform Cost Search

(1)

```
Solving puzzle...
Using UCS algorithm...
Goal state reached!
Visited nodes: 109
Total cost (steps): 4

Solution Path:
Initial state:
+-------+
|AAB..F|
|..BCDF|
|GPPCDF|
|GH.III|
|GHJ...|
|LLJMM.|
+-------+

Move 1: C-up
+-------+
|AABC.F|
|..BCDF|
|GPP.DF|
|GH.III|
|GHJ...|
|LLJMM.|
+------+

Move 2: I-left
+------+
|AABC.F|
|..BCDF|
|GPP.DF|
|GHIII.|
|GHJ...|
|LLJMM.|
+------+

Move 3: F-down
+------+
|AABC..|
|..BCD.|
|GPP.D.|
|GHIIIF|
|GHJ..F|
|LLJMMF|
+------+

Move 4: D-up
+------+
|AABCD.|
|..BCD.|
|GPP...|
|GHIIIF|
|GHJ..F|
|LLJMMF|
+------+

Total moves: 4

Final Board State:
+------+
|AABCD.|
|..BCD.|
|GPP...|
|GHIIIF|
|GHJ..F|
|LLJMMF|
+------+
Execution time: 0,038 seconds
```

(2)

```
Solving puzzle...
Using UCS algorithm...
Goal state reached!
Visited nodes: 83
Total cost (steps): 3

Solution Path:
Initial state:
+------+
|ABCDDD|
|ABCEE.|
|PPF..G|
|HHF.IG|
|JJ..IG|
|...LLL|
+------+

Move 1: F-down
+------+
|ABCDDD|
|ABCEE.|
|PP...G|
|HHF.IG|
|JJF.IG|
|...LLL|
+------+

Move 2: L-left
+------+
|ABCDDD|
|ABCEE.|
|PP...G|
|HHF.IG|
|JJF.IG|
|.LLL..|
+------+

Move 3: G-down
+------+
|ABCDDD|
|ABCEE.|
|PP....|
|HHF.IG|
|JJF.IG|
|.LLL.G|
+------+

Total moves: 3

Final Board State:
+------+
|ABCDDD|
|ABCEE.|
|PP....|
|HHF.IG|
|JJF.IG|
|.LLL.G|
+------+

Execution time: 0,038 seconds
```

(3)

```
Solving puzzle...
Using UCS algorithm...
Goal state reached!
Visited nodes: 7015
Total cost (steps): 29

Solution Path:
Initial state:
+------+
|ABBC.E|
|AD.C.E|
|.DPPGE|
|HHIJG.|
|..IJLL|
|MMINN.|
+------+

Move 1: A-down
+------+
|.BBC.E|
|AD.C.E|
|ADPPGE|
|HHIJG.|
|..IJLL|
|MMINN.|
+------+

Move 2: N-right
+------+
|.BBC.E|
|AD.C.E|
|ADPPGE|
|HHIJG.|
|..IJLL|
|MMI.NN|
+------+

Move 3: E-down
+------+
|.BBC..|
|AD.C.E|
|ADPPGE|
|HHIJGE|
|..IJLL|
|MMI.NN|
+------+

Move 4: B-left
+------+
|BB.C..|
|AD.C.E|
|ADPPGE|
|HHIJGE|
|..IJLL|
|MMI.NN|
+------+

Move 5: G-up
+------+
|BB.CG.|
|AD.CGE|
|ADPP.E|
|HHIJ.E|
|..IJLL|
|MMI.NN|
+------+

Move 6: P-right
+------+
|BB.CG.|
|AD.CGE|
|AD.PPE|
|HHIJ.E|
|..IJLL|
|MMI.NN|
+------+

Move 7: J-down
+------+
|BB.CG.|
|AD.CGE|
|AD.PPE|
|HHI..E|
|..IJLL|
|MMIJNN|
+------+

Move 8: I-up
+------+
|BBICG.|
|ADICGE|
|ADIPPE|
|HH...E|
|...JLL|
|MM.JNN|
+------+

Move 9: H-right
+------+
|BBICG.|
|ADICGE|
|ADIPPE|
|...HHE|
|...JLL|
|MM.JNN|
+------+

Move 10: I-down
+------+
|BB.CG.|
|AD.CGE|
|AD.PPE|
|..IHHE|
|..IJLL|
|MMIJNN|
+------+

Move 11: D-down
+------+
|BB.CG.|
|A..CGE|
|A..PPE|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 12: P-left
+------+
|BB.CG.|
|A..CGE|
|APP..E|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 13: C-down
+------+
|BB..G.|
|A..CGE|
|APPC.E|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 14: G-down
+------+
|BB....|
|A..CGE|
|APPCGE|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 15: B-right
+------+
|....BB|
|A..CGE|
|APPCGE|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+
Move 16: A-up
+------+
|A...BB|
|A..CGE|
|.PPCGE|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 17: C-up
+------+
|A..CBB|
|A..CGE|
|.PP.GE|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 18: P-right
+------+
|A..CBB|
|A..CGE|
|..PPGE|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 19: D-up
+------+
|AD.CBB|
|AD.CGE|
|..PPGE|
|..IHHE|
|..IJLL|
|MMIJNN|
+------+

Move 20: P-left
+------+
|AD.CBB|
|AD.CGE|
|PP..GE|
|..IHHE|
|..IJLL|
|MMIJNN|
+------+

Move 21: I-up
+------+
|ADICBB|
|ADICGE|
|PPI.GE|
|...HHE|
|...JLL|
|MM.JNN|
+------+

Move 22: H-left
+------+
|ADICBB|
|ADICGE|
|PPI.GE|
|HH...E|
|...JLL|
|MM.JNN|
+------+

Move 23: J-up
+------+
|ADICBB|
|ADICGE|
|PPIJGE|
|HH.J.E|
|....LL|
|MM..NN|
+------+

Move 24: L-left
+------+
|ADICBB|
|ADICGE|
|PPIJGE|
|HH.J.E|
|LL....|
|MM..NN|
+------+

Move 25: N-left
+------+
|ADICBB|
|ADICGE|
|PPIJGE|
|HH.J.E|
|LL....|
|MM.NN.|
+------+

Move 26: E-down
+------+
|ADICBB|
|ADICG.|
|PPIJG.|
|HH.J.E|
|LL...E|
|MM.NNE|
+------+

Move 27: J-down
+------+
|ADICBB|
|ADICG.|
|PPI.G.|
|HH.J.E|
|LL.J.E|
|MM.NNE|
+------+

Move 28: I-down
+------+
|AD.CBB|
|AD.CG.|
|PP..G.|
|HHIJ.E|
|LLIJ.E|
|MMINNE|
+------+

Move 29: G-down
+------+
|AD.CBB|
|AD.C..|
|PP....|
|HHIJGE|
|LLIJGE|
|MMINNE|
+------+

Total moves: 29

Final Board State:
+------+
|AD.CBB|
|AD.C..|
|PP....|
|HHIJGE|
|LLIJGE|
|MMINNE|
+------+

Execution time: 0,338 seconds
```

(4)

```
Solving puzzle...
Using UCS algorithm...
Goal state reached!
Visited nodes: 1854
Total cost (steps): 37

Solution Path:
Initial state:
+------+
|AAAB.C|
|DEEB.C|
|D.FPPC|
|..FGHH|
|..FG..|
|.IIJJ.|
+------+

Move 1: J-right
+------+
|AAAB.C|
|DEEB.C|
|D.FPPC|
|..FGHH|
|..FG..|
|.II.JJ|
+------+

Move 2: G-down
+------+
|AAAB.C|
|DEEB.C|
|D.FPPC|
|..F.HH|
|..FG..|
|.IIGJJ|
+------+

Move 3: D-down
+------+
|AAAB.C|
|.EEB.C|
|..FPPC|
|D.F.HH|
|D.FG..|
|.IIGJJ|
+------+

Move 4: I-left
+------+
|AAAB.C|
|.EEB.C|
|..FPPC|
|D.F.HH|
|D.FG..|
|II.GJJ|
+------+

Move 5: F-down
+------+
|AAAB.C|
|.EEB.C|
|...PPC|
|D.F.HH|
|D.FG..|
|IIFGJJ|
+------+

Move 6: H-left
+------+
|AAAB.C|
|.EEB.C|
|...PPC|
|D.FHH.|
|D.FG..|
|IIFGJJ|
+------+
Move 7: C-down
+------+
|AAAB..|
|.EEB..|
|...PPC|
|D.FHHC|
|D.FG.C|
|IIFGJJ|
+------+

Move 8: P-left
+------+
|AAAB..|
|.EEB..|
|PP...C|
|D.FHHC|
|D.FG.C|
|IIFGJJ|
+------+

Move 9: B-down
+------+
|AAA...|
|.EEB..|
|PP.B.C|
|D.FHHC|
|D.FG.C|
|IIFGJJ|
+------+

Move 10: A-right
+------+
|...AAA|
|.EEB..|
|PP.B.C|
|D.FHHC|
|D.FG.C|
|IIFGJJ|
+------+

Move 11: E-left
+------+
|...AAA|
|EE.B..|
|PP.B.C|
|D.FHHC|
|D.FG.C|
|IIFGJJ|
+------+

Move 12: F-up
+------+
|..FAAA|
|EEFB..|
|PPFB.C|
|D..HHC|
|D..G.C|
|II.GJJ|
+------+

Move 13: I-right
+------+
|..FAAA|
|EEFB..|
|PPFB.C|
|D..HHC|
|D..G.C|
|.IIGJJ|
+------+

Move 14: D-down
+------+
|..FAAA|
|EEFB..|
|PPFB.C|
|...HHC|
|D..G.C|
|DIIGJJ|
+------+

Move 15: H-left
+------+
|..FAAA|
|EEFB..|
|PPFB.C|
|HH...C|
|D..G.C|
|DIIGJJ|
+------+

Move 16: B-down
```

```
+------+
|..FAAA|
|EEF...|
|PPFB.C|
|HH.B.C|
|D..G.C|
|DIIGJJ|
+------+

Move 17: F-down
+------+
|...AAA|
|EE....|
|PPFB.C|
|HHFB.C|
|D.FG.C|
|DIIGJJ|
+------+

Move 18: E-right
+------+
|...AAA|
|....EE|
|PPFB.C|
|HHFB.C|
|D.FG.C|
|DIIGJJ|
+------+

Move 19: F-up
+------+
|..FAAA|
|..F.EE|
|PPFB.C|
|HH.B.C|
|D..G.C|
|DIIGJJ|
+------+

Move 20: B-up
+------+
|..FAAA|
|..FBEE|
|PPFB.C|
|HH...C|
|D..G.C|
|DIIGJJ|
+------+

Move 21: H-right
+------+
|..FAAA|
|..FBEE|
|PPFB.C|
|...HHC|
|D..G.C|
|DIIGJJ|
+------+

Move 22: D-up
+------+
|..FAAA|
|..FBEE|
|PPFB.C|
|D..HHC|
|D..G.C|
|.IIGJJ|
+------+

Move 23: I-left
+------+
|..FAAA|
|..FBEE|
|PPFB.C|
|D..HHC|
|D..G.C|
|II.GJJ|
+------+

Move 24: F-down
+------+
|...AAA|
|...BEE|
|PP.B.C|
|D.FHHC|
|D.FG.C|
|IIFGJJ|
+------+

Move 25: P-right
+------+
|...AAA|
|...BEE|
|.PPB.C|
|D.FHHC|
|D.FG.C|
|IIFGJJ|
+------+

Move 26: D-up
+------+
|D..AAA|
|D..BEE|
|.PPB.C|
|..FHHC|
|..FG.C|
|IIFGJJ|
+------+

Move 27: P-left
+------+
|D..AAA|
|D..BEE|
|PP.B.C|
|..FHHC|
|..FG.C|
|IIFGJJ|
+------+

Move 28: F-up
+------+
|D.FAAA|
|D.FBEE|
|PPFB.C|
|...HHC|
|...G.C|
|II.GJJ|
+------+

Move 29: H-left
+------+
|D.FAAA|
|D.FBEE|
|PPFB.C|
|HH...C|
|...G.C|
|II.GJJ|
+------+
Move 30: G-up
+------+
|D.FAAA|
|D.FBEE|
|PPFB.C|
|HH.G.C|
|...G.C|
|II..JJ|
+------+

Move 31: F-down
+------+
|D..AAA|
|D..BEE|
|PP.B.C|
|HHFG.C|
|..FG.C|
|IIF.JJ|
+------+

Move 32: J-left
+------+
|D..AAA|
|D..BEE|
|PP.B.C|
|HHFG.C|
|..FG.C|
|IIFJJ.|
+------+

Move 33: P-right
+------+
|D..AAA|
|D..BEE|
|.PPB.C|
|HHFG.C|
|..FG.C|
|IIFJJ.|
+------+

Move 34: D-down
+------+
|...AAA|
|D..BEE|
|DPPB.C|
|HHFG.C|
|..FG.C|
|IIFJJ.|
+------+

Move 35: A-left
+------+
|AAA...|
|D..BEE|
|DPPB.C|
|HHFG.C|
|..FG.C|
|IIFJJ.|
+------+

Move 36: B-up
+------+
|AAAB..|
|D..BEE|
|DPP..C|
|HHFG.C|
|..FG.C|
|IIFJJ.|
+------+

Move 37: C-down
+------+
|AAAB..|
|D..BEE|
|DPP...|
|HHFG.C|
|..FG.C|
|IIFJJC|
+------+

Total moves: 37

Final Board State:
+------+
|AAAB..|
|D..BEE|
|DPP...|
|HHFG.C|
|..FG.C|
|IIFJJC|
+------+
Execution time: 0,111 seconds
```

(5)

```
Solving puzzle...
Using UCS algorithm...
Goal state reached!
Visited nodes: 664
Total cost (steps): 7

Solution Path:
Initial state:
+--------+
|JJJJJJJJ|
|IIIIIIII|
|PPBCDEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|.....EFG|
|...HHHHH|
+--------+

Move 1: H-left
+--------+
|JJJJJJJJ|
|IIIIIIII|
|PPBCDEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|.....EFG|
|HHHHH...|
+--------+

Move 2: B-down
+--------+
|JJJJJJJJ|
|IIIIIIII|
|PP.CDEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..B..EFG|
|HHHHH...|
+--------+
Move 3: G-down
+--------+
|JJJJJJJJ|
|IIIIIIII|
|PP.CDEF.|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..B..EFG|
|HHHHH..G|
+--------+

Move 4: E-down
+--------+
|JJJJJJJJ|
|IIIIIIII|
|PP.CD.F.|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..B..EFG|
|HHHHHE.G|
+--------+

Move 5: F-down
+--------+
|JJJJJJJJ|
|IIIIIIII|
|PP.CD...|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..B..EFG|
|HHHHHEFG|
+--------+

Move 6: D-down
+--------+
|JJJJJJJJ|
|IIIIIIII|
|PP.C....|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..B.DEFG|
|HHHHHEFG|
+--------+

Move 7: C-down
+--------+
|JJJJJJJJ|
|IIIIIIII|
|PP......|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|HHHHHEFG|
+--------+

Total moves: 7

Final Board State:
+--------+
|JJJJJJJJ|
|IIIIIIII|
|PP......|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|HHHHHEFG|
+--------+

Execution time: 0,069 seconds
```

B. A* Search

(1)

```
Solving puzzle...
Using A* algorithm with Distance + Blocking Cars
heuristic...
Using A* with heuristic: Distance + Blocking Cars
Goal state reached!
Visited nodes: 12
Total cost (steps): 5

Solution Path:
Initial state:
+------+
|AAB..F|
|..BCDF|
|GPPCDF|
|GH.III|
|GHJ...|
|LLJMM.|
+------+

Move 1: C-up
+------+
|AABC.F|
|..BCDF|
|GPP.DF|
|GH.III|
|GHJ...|
|LLJMM.|
+------+

Move 2: D-up
+------+
|AABCDF|
|..BCDF|
|GPP..F|
|GH.III|
|GHJ...|
|LLJMM.|
+------+

Move 3: P-right
+------+
|AABCDF|
|..BCDF|
|G..PPF|
|GH.III|
|GHJ...|
|LLJMM.|
+------+

Move 4: I-left
+------+
|AABCDF|
|..BCDF|
|G..PPF|
|GHIII.|
|GHJ...|
|LLJMM.|
+------+
Move 5: F-down
+------+
|AABCD.|
|..BCD.|
|G..PP.|
|GHIIIF|
|GHJ..F|
|LLJMMF|
+------+

Total moves: 5

Final Board State:
+------+
|AABCD.|
|..BCD.|
|G..PP.|
|GHIIIF|
|GHJ..F|
|LLJMMF|
+------+

Execution time: 0,024 seconds
```

(2)

```
Solving puzzle...
Using A* algorithm with Distance + Blocking Cars heuristic...
Using A* with heuristic: Distance + Blocking Cars
Goal state reached!
Visited nodes: 13
Total cost (steps): 4

Solution Path:
Initial state:
+------+
|ABCDDD|
|ABCEE.|
|PPF..G|
|HHF.IG|
|JJ..IG|
|...LLL|
+------+

Move 1: F-down
+------+
|ABCDDD|
|ABCEE.|
|PP...G|
|HHF.IG|
|JJF.IG|
|...LLL|
+------+

Move 2: P-right
+------+
|ABCDDD|
|ABCEE.|
|...PPG|
|HHF.IG|
|JJF.IG|
|...LLL|
+------+

Move 3: L-left
+------+
|ABCDDD|
|ABCEE.|
|...PPG|
|HHF.IG|
|JJF.IG|
|..LLL.|
+------+

Move 4: G-down
+------+
|ABCDDD|
|ABCEE.|
|...PP.|
|HHF.IG|
|JJF.IG|
|..LLLG|
+------+

Total moves: 4
```

```
Final Board State:
+------+
|ABCDDD|
|ABCEE.|
|...PP.|
|HHF.IG|
|JJF.IG|
|..LLLG|
+------+

Execution time: 0,028 seconds
```

(3)

```
Solving puzzle...
Using A* algorithm with Distance + Blocking Cars heuristic...
Using A* with heuristic: Distance + Blocking Cars
Visited 1000 nodes so far
Visited 2000 nodes so far
Visited 3000 nodes so far
Goal state reached!
Visited nodes: 3688
Total cost (steps): 32

Solution Path:
Initial state:
+------+
|ABBC.E|
|AD.C.E|
|.DPPGE|
|HHIJG.|
|..IJLL|
|MMINN.|
+------+

Move 1: G-up
+------+
|ABBCGE|
|AD.CGE|
|.DPP.E|
|HHIJ..|
|..IJLL|
|MMINN.|
+------+

Move 2: P-right
+------+
|ABBCGE|
|AD.CGE|
|.D.PPE|
|HHIJ..|
|..IJLL|
|MMINN.|
+------+

Move 3: E-down
+------+
|ABBCG.|
|AD.CGE|
|.D.PPE|
|HHIJ.E|
|..IJLL|
|MMINN.|
+------+

Move 4: N-right
+------+
|ABBCG.|
|AD.CGE|
|.D.PPE|
|HHIJ.E|
|..IJLL|
|MMI.NN|
+------+

Move 5: A-down
+------+
|.BBCG.|
|AD.CGE|
|AD.PPE|
|HHIJ.E|
|..IJLL|
|MMI.NN|
+------+

Move 6: J-down
+------+
|.BBCG.|
|AD.CGE|
|AD.PPE|
|HHI..E|
|..IJLL|
|MMIJNN|
+------+

Move 7: B-left
+------+
|BB.CG.|
|AD.CGE|
|AD.PPE|
|HHI..E|
|..IJLL|
|MMIJNN|
+------+

Move 8: I-up
+------+
|BBICG.|
|ADICGE|
|ADIPPE|
|HH...E|
|...JLL|
|MM.JNN|
+------+

Move 9: H-right
+------+
|BBICG.|
|ADICGE|
|ADIPPE|
|...HHE|
|...JLL|
|MM.JNN|
+------+

Move 10: I-down
+------+
|BB.CG.|
|AD.CGE|
|AD.PPE|
|..IHHE|
|..IJLL|
|MMIJNN|
+------+

Move 11: D-down
+------+
|BB.CG.|
|A..CGE|
|A..PPE|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 12: P-left
+------+
|BB.CG.|
|A..CGE|
|APP..E|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 13: C-down
+------+
|BB..G.|
|A..CGE|
|APPC.E|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 14: G-down
+------+
|BB....|
|A..CGE|
|APPCGE|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 15: B-right
+------+
|....BB|
|A..CGE|
|APPCGE|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 16: C-up
+------+
|...CBB|
|A..CGE|
|APP.GE|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 17: P-right
+------+
|...CBB|
|A..CGE|
|A.PPGE|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 18: D-up
+------+
|.D.CBB|
|AD.CGE|
|A.PPGE|
|..IHHE|
|..IJLL|
|MMIJNN|
+------+

Move 19: A-up
+------+
|AD.CBB|
|AD.CGE|
|..PPGE|
|..IHHE|
|..IJLL|
|MMIJNN|
+------+

Move 20: P-left
+------+
|AD.CBB|
|AD.CGE|
|PP..GE|
|..IHHE|
|..IJLL|
|MMIJNN|
+------+

Move 21: I-up
+------+
|ADICBB|
|ADICGE|
|PPI.GE|
|...HHE|
|...JLL|
|MM.JNN|
+------+

Move 22: H-left
+------+
|ADICBB|
|ADICGE|
|PPI.GE|
|.HH..E|
|...JLL|
|MM.JNN|
+------+

Move 23: J-up
+------+
|ADICBB|
|ADICGE|
|PPIJGE|
|.HHJ.E|
|....LL|
|MM..NN|
+------+

Move 24: L-left
+------+
|ADICBB|
|ADICGE|
|PPIJGE|
|.HHJ.E|
|.LL...|
|MM..NN|
+------+

Move 25: G-down
+------+
|ADICBB|
|ADIC.E|
|PPIJ.E|
|.HHJGE|
|.LL.G.|
|MM..NN|
+------+

Move 26: J-down
+------+
|ADICBB|
|ADIC.E|
|PPI..E|
|.HHJGE|
|.LLJG.|
|MM..NN|
+------+

Move 27: H-left
+------+
|ADICBB|
|ADIC.E|
|PPI..E|
|HH.JGE|
|.LLJG.|
|MM..NN|
+------+

Move 28: L-left
+------+
|ADICBB|
|ADIC.E|
|PPI..E|
|HH.JGE|
|LL.JG.|
|MM..NN|
+------+

Move 29: I-down
+------+
|AD.CBB|
|AD.C.E|
|PP...E|
|HHIJGE|
|LLIJG.|
|MMI.NN|
+------+

Move 30: P-right
+------+
|AD.CBB|
|AD.C.E|
|...PPE|
|HHIJGE|
|LLIJG.|
|MMI.NN|
+------+

Move 31: N-left
+------+
|AD.CBB|
|AD.C.E|
|...PPE|
|HHIJGE|
|LLIJG.|
|MMINN.|
+------+

Move 32: E-down
+------+
|AD.CBB|
|AD.C..|
|...PP.|
|HHIJGE|
|LLIJGE|
|MMINNE|
+------+

Total moves: 32

Final Board State:
+------+
|AD.CBB|
|AD.C..|
|...PP.|
|HHIJGE|
|LLIJGE|
|MMINNE|
+------+

Execution time: 0,271 seconds
```

(4)

```
Solving puzzle...
Using A* algorithm with Distance + Blocking Cars heuristic...
Using A* with heuristic: Distance + Blocking Cars
Visited 1000 nodes so far
Goal state reached!
Visited nodes: 1561
Total cost (steps): 38

Solution Path:
Initial state:
+------+
|AAAB.C|
|DEEB.C|
|D.FPPC|
|..FGHH|
|..FG..|
|.IIJJ.|
+------+

Move 1: J-right
+------+
|AAAB.C|
|DEEB.C|
|D.FPPC|
|..FGHH|
|..FG..|
|.II.JJ|
+------+

Move 2: G-down
+------+
|AAAB.C|
|DEEB.C|
|D.FPPC|
|..F.HH|
|..FG..|
|.IIGJJ|
+------+

Move 3: D-down
+------+
|AAAB.C|
|.EEB.C|
|..FPPC|
|D.F.HH|
|D.FG..|
|.IIGJJ|
+------+

Move 4: H-left
+------+
|AAAB.C|
|.EEB.C|
|..FPPC|
|D.FHH.|
|D.FG..|
|.IIGJJ|
+------+
Move 5: I-left
+------+
|AAAB.C|
|.EEB.C|
|..FPPC|
|D.FHH.|
|D.FG..|
|II.GJJ|
+------+

Move 6: F-down
+------+
|AAAB.C|
|.EEB.C|
|...PPC|
|D.FHH.|
|D.FG..|
|IIFGJJ|
+------+

Move 7: C-down
+------+
|AAAB..|
|.EEB..|
|...PPC|
|D.FHHC|
|D.FG.C|
|IIFGJJ|
+------+

Move 8: P-left
+------+
|AAAB..|
|.EEB..|
|PP...C|
|D.FHHC|
|D.FG.C|
|IIFGJJ|
+------+

Move 9: B-down
+------+
|AAA...|
|.EEB..|
|PP.B.C|
|D.FHHC|
|D.FG.C|
|IIFGJJ|
+------+

Move 10: A-right
+------+
|...AAA|
|.EEB..|
|PP.B.C|
|D.FHHC|
|D.FG.C|
|IIFGJJ|
+------+
Move 11: E-left
+------+
|...AAA|
|EE.B..|
|PP.B.C|
|D.FHHC|
|D.FG.C|
|IIFGJJ|
+------+

Move 12: F-up
+------+
|..FAAA|
|EEFB..|
|PPFB.C|
|D..HHC|
|D..G.C|
|II.GJJ|
+------+

Move 13: I-right
+------+
|..FAAA|
|EEFB..|
|PPFB.C|
|D..HHC|
|D..G.C|
|.IIGJJ|
+------+

Move 14: D-down
+------+
|..FAAA|
|EEFB..|
|PPFB.C|
|...HHC|
|D..G.C|
|DIIGJJ|
+------+

Move 15: H-left
+------+
|..FAAA|
|EEFB..|
|PPFB.C|
|HH...C|
|D..G.C|
|DIIGJJ|
+------+

Move 16: F-down
+------+
|...AAA|
|EE.B..|
|PPFB.C|
|HHF..C|
|D.FG.C|
|DIIGJJ|
+------+

Move 17: B-down
+------+
|...AAA|
|EE....|
|PPFB.C|
|HHFB.C|
|D.FG.C|
|DIIGJJ|
+------+

Move 18: E-right
+------+
|...AAA|
|....EE|
|PPFB.C|
|HHFB.C|
|D.FG.C|
|DIIGJJ|
+------+

Move 19: F-up
+------+
|..FAAA|
|..F.EE|
|PPFB.C|
|HH.B.C|
|D..G.C|
|DIIGJJ|
+------+

Move 20: B-up
+------+
|..FAAA|
|..FBEE|
|PPFB.C|
|HH...C|
|D..G.C|
|DIIGJJ|
+------+

Move 21: H-right
+------+
|..FAAA|
|..FBEE|
|PPFB.C|
|...HHC|
|D..G.C|
|DIIGJJ|
+------+

Move 22: D-up
+------+
|..FAAA|
|..FBEE|
|PPFB.C|
|D..HHC|
|D..G.C|
|.IIGJJ|
+------+

Move 23: I-left
+------+
|..FAAA|
|..FBEE|
|PPFB.C|
|D..HHC|
|D..G.C|
|II.GJJ|
+------+

Move 24: F-down
+------+
|...AAA|
|...BEE|
|PP.B.C|
|D.FHHC|
|D.FG.C|
|IIFGJJ|
+------+

Move 25: P-right
+------+
|...AAA|
|...BEE|
|.PPB.C|
|D.FHHC|
|D.FG.C|
|IIFGJJ|
+------+

Move 26: D-up
+------+
|D..AAA|
|D..BEE|
|.PPB.C|
|..FHHC|
|..FG.C|
|IIFGJJ|
+------+

Move 27: P-left
+------+
|D..AAA|
|D..BEE|
|PP.B.C|
|..FHHC|
|..FG.C|
|IIFGJJ|
+------+

Move 28: F-up
+------+
|D.FAAA|
|D.FBEE|
|PPFB.C|
|...HHC|
|...G.C|
|II.GJJ|
+------+

Move 29: H-left
+------+
|D.FAAA|
|D.FBEE|
|PPFB.C|
|HH...C|
|...G.C|
|II.GJJ|
+------+

Move 30: F-down
+------+
|D..AAA|
|D..BEE|
|PP.B.C|
|HHF..C|
|..FG.C|
|IIFGJJ|
+------+

Move 31: P-right
+------+
|D..AAA|
|D..BEE|
|.PPB.C|
|HHF..C|
|..FG.C|
|IIFGJJ|
+------+

Move 32: D-down
+------+
|...AAA|
|D..BEE|
|DPPB.C|
|HHF..C|
|..FG.C|
|IIFGJJ|
+------+

Move 33: A-left
+------+
|AAA...|
|D..BEE|
|DPPB.C|
|HHF..C|
|..FG.C|
|IIFGJJ|
+------+

Move 34: B-up
+------+
|AAAB..|
|D..BEE|
|DPP..C|
|HHF..C|
|..FG.C|
|IIFGJJ|
+------+

Move 35: P-right
+------+
|AAAB..|
|D..BEE|
|D..PPC|
|HHF..C|
|..FG.C|
|IIFGJJ|
+------+

Move 36: G-up
+------+
|AAAB..|
|D..BEE|
|D..PPC|
|HHFG.C|
|..FG.C|
|IIF.JJ|
+------+

Move 37: J-left
+------+
|AAAB..|
|D..BEE|
|D..PPC|
|HHFG.C|
|..FG.C|
|IIFJJ.|
+------+

Move 38: C-down
+------+
|AAAB..|
|D..BEE|
|D..PP.|
|HHFG.C|
|..FG.C|
|IIFJJC|
+------+

Total moves: 38

Final Board State:
+------+
|AAAB..|
|D..BEE|
|D..PP.|
|HHFG.C|
|..FG.C|
|IIFJJC|
+------+

Execution time: 0,121 seconds
```

(5)

```
Solving puzzle...
Using A* algorithm with Distance + Blocking Cars heuristic...
Using A* with heuristic: Distance + Blocking Cars
Goal state reached!
Visited nodes: 24
Total cost (steps): 9

Solution Path:
Initial state:
+--------+
|JJJJJJJJ|
|IIIIIIII|
|PPBCDEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|.....EFG|
|...HHHHH|
+--------+

Move 1: B-down
+--------+
|JJJJJJJJ|
|IIIIIIII|
|PP.CDEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..B..EFG|
|...HHHHH|
+--------+

Move 2: C-down
+--------+
|JJJJJJJJ|
|IIIIIIII|
|PP..DEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BC.EFG|
|...HHHHH|
+--------+

Move 3: D-down
+--------+
|JJJJJJJJ|
|IIIIIIII|
|PP...EFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|...HHHHH|
+--------+

Move 4: P-right
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PPEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|...HHHHH|
+--------+

Move 5: H-left
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PPEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|.HHHHH..|
+--------+

Move 6: F-down
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PPE.G|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|.HHHHHF.|
+--------+

Move 7: G-down
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PPE..|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|.HHHHHFG|
+--------+

Move 8: H-left
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PPE..|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|HHHHH.FG|
+--------+

Move 9: E-down
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PP...|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|HHHHHEFG|
+--------+

Total moves: 9

Final Board State:
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PP...|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|HHHHHEFG|
+--------+

Execution time: 0,032 seconds
```

C. Greedy Best First Search

(1)

```
Solving puzzle...
Using Greedy Best-First Search with Distance + Blocking
Cars heuristic...
Using Greedy Best-First Search with heuristic: Distance +
Blocking Cars
Goal state reached!
Visited nodes: 17
Total cost (steps): 6

Solution Path:
Initial state:
+------+
|AAB..F|
|..BCDF|
|GPPCDF|
|GH.III|
|GHJ...|
|LLJMM.|
+------+
Move 1: C-up
+------+
|AABC.F|
|..BCDF|
|GPP.DF|
|GH.III|
|GHJ...|
|LLJMM.|
+------+

Move 2: D-up
+------+
|AABCDF|
|..BCDF|
|GPP..F|
|GH.III|
|GHJ...|
|LLJMM.|
+------+

Move 3: P-right
+------+
|AABCDF|
|..BCDF|
|G..PPF|
|GH.III|
|GHJ...|
|LLJMM.|
+------+

Move 4: G-up
+------+
|AABCDF|
|G.BCDF|
|G..PPF|
|GH.III|
|.HJ...|
|LLJMM.|
+------+

Move 5: I-left
+------+
|AABCDF|
|G.BCDF|
|G..PPF|
|GHIII.|
|.HJ...|
|LLJMM.|
+------+

Move 6: F-down
+------+
|AABCD.|
|G.BCD.|
|G..PP.|
|GHIIIF|
|.HJ..F|
|LLJMMF|
+------+

Total moves: 6

Final Board State:
+------+
|AABCD.|
|G.BCD.|
|G..PP.|
|GHIIIF|
|.HJ..F|
|LLJMMF|
+------+

Execution time: 0,027 seconds
```

(2)

```
Solving puzzle...
Using Greedy Best-First Search with Distance + Blocking
Cars heuristic...
Using Greedy Best-First Search with heuristic: Distance +
Blocking Cars
Goal state reached!
Visited nodes: 12
Total cost (steps): 5

Solution Path:
Initial state:
+------+
|ABCDDD|
|ABCEE.|
|PPF..G|
|HHF.IG|
|JJ..IG|
|...LLL|
+------+

Move 1: F-down
+------+
|ABCDDD|
|ABCEE.|
|PP...G|
|HHF.IG|
|JJF.IG|
|...LLL|
+------+

Move 2: P-right
+------+
|ABCDDD|
|ABCEE.|
|...PPG|
|HHF.IG|
|JJF.IG|
|...LLL|
+------+

Move 3: A-down
+------+
|.BCDDD|
|ABCEE.|
|A..PPG|
|HHF.IG|
|JJF.IG|
|...LLL|
+------+

Move 4: L-left
+------+
|.BCDDD|
|ABCEE.|
|A..PPG|
|HHF.IG|
|JJF.IG|
|..LLL.|
+------+

Move 5: G-down
+------+
|.BCDDD|
|ABCEE.|
|A..PP.|
|HHF.IG|
|JJF.IG|
|..LLLG|
+------+

Total moves: 5

Final Board State:
+------+
|.BCDDD|
|ABCEE.|
|A..PP.|
|HHF.IG|
|JJF.IG|
|..LLLG|
+------+

Execution time: 0,027 seconds
```

(3)

```
Solving puzzle...
Using Greedy Best-First Search with Distance + Blocking
Cars heuristic...
Using Greedy Best-First Search with heuristic: Distance +
Blocking Cars
Visited 1000 nodes so far
Visited 2000 nodes so far
Goal state reached!
Visited nodes: 2849
Total cost (steps): 90

Solution Path:
Initial state:
+------+
|ABBC.E|
|AD.C.E|
|.DPPGE|
|HHIJG.|
|..IJLL|
|MMINN.|
+------+

Move 1: G-up
+------+
|ABBCGE|
|AD.CGE|
|.DPP.E|
|HHIJ..|
|..IJLL|
|MMINN.|
+------+

Move 2: P-right
+------+
|ABBCGE|
|AD.CGE|
|.D.PPE|
|HHIJ..|
|..IJLL|
|MMINN.|
+------+

Move 3: I-up
+------+
|ABBCGE|
|AD.CGE|
|.DIPPE|
|HHIJ..|
|..IJLL|
|MM.NN.|
+------+

Move 4: N-left
+------+
|ABBCGE|
|AD.CGE|
|.DIPPE|
|HHIJ..|
|..IJLL|
|MMNN..|
+------+

Move 5: I-up
+------+
|ABBCGE|
|ADICGE|
|.DIPPE|
|HHIJ..|
|...JLL|
|MMNN..|
+------+
Move 6: A-down
+------+
|.BBCGE|
|ADICGE|
|ADIPPE|
|HHIJ..|
|...JLL|
|MMNN..|
+------+

Move 7: B-left
+------+
|BB.CGE|
|ADICGE|
|ADIPPE|
|HHIJ..|
|...JLL|
|MMNN..|
+------+

Move 8: N-right
+------+
|BB.CGE|
|ADICGE|
|ADIPPE|
|HHIJ..|
|...JLL|
|MM..NN|
+------+

Move 9: M-right
+------+
|BB.CGE|
|ADICGE|
|ADIPPE|
|HHIJ..|
|...JLL|
|..MMNN|
+------+

Move 10: I-down
+------+
|BB.CGE|
|AD.CGE|
|ADIPPE|
|HHIJ..|
|..IJLL|
|..MMNN|
+------+

Move 11: M-left
+------+
|BB.CGE|
|AD.CGE|
|ADIPPE|
|HHIJ..|
|..IJLL|
|.MM.NN|
+------+

Move 12: I-up
+------+
|BBICGE|
|ADICGE|
|ADIPPE|
|HH.J..|
|...JLL|
|.MM.NN|
+------+

Move 13: N-left
+------+
|BBICGE|
|ADICGE|
|ADIPPE|
|HH.J..|
|...JLL|
|.MMNN.|
+------+

Move 14: M-left
+------+
|BBICGE|
|ADICGE|
|ADIPPE|
|HH.J..|
|...JLL|
|MM.NN.|
+------+

Move 15: I-down
+------+
|BB.CGE|
|AD.CGE|
|ADIPPE|
|HHIJ..|
|..IJLL|
|MM.NN.|
+------+

Move 16: E-down
+------+
|BB.CG.|
|AD.CGE|
|ADIPPE|
|HHIJ.E|
|..IJLL|
|MM.NN.|
+------+

Move 17: N-left
+------+
|BB.CG.|
|AD.CGE|
|ADIPPE|
|HHIJ.E|
|..IJLL|
|MMNN..|
+------+

Move 18: I-up
+------+
|BBICG.|
|ADICGE|
|ADIPPE|
|HH.J.E|
|...JLL|
|MMNN..|
+------+

Move 19: N-right
+------+
|BBICG.|
|ADICGE|
|ADIPPE|
|HH.J.E|
|...JLL|
|MM..NN|
+------+

Move 20: M-right
+------+
|BBICG.|
|ADICGE|
|ADIPPE|
|HH.J.E|
|...JLL|
|..MMNN|
+------+

Move 21: H-right
+------+
|BBICG.|
|ADICGE|
|ADIPPE|
|.HHJ.E|
|...JLL|
|..MMNN|
+------+

Move 22: E-up
+------+
|BBICGE|
|ADICGE|
|ADIPPE|
|.HHJ..|
|...JLL|
|..MMNN|
+------+

Move 23: M-left
+------+
|BBICGE|
|ADICGE|
|ADIPPE|
|.HHJ..|
|...JLL|
|MM..NN|
+------+

Move 24: N-left
+------+
|BBICGE|
|ADICGE|
|ADIPPE|
|.HHJ..|
|...JLL|
|MMNN..|
+------+

Move 25: A-down
+------+
|BBICGE|
|.DICGE|
|.DIPPE|
|AHHJ..|
|A..JLL|
|MMNN..|
+------+

Move 26: N-right
+------+
|BBICGE|
|.DICGE|
|.DIPPE|
|AHHJ..|
|A..JLL|
|MM.NN.|
+------+

Move 27: M-right
+------+
|BBICGE|
|.DICGE|
|.DIPPE|
|AHHJ..|
|A..JLL|
|.MMNN.|
+------+

Move 28: N-right
+------+
|BBICGE|
|.DICGE|
|.DIPPE|
|AHHJ..|
|A..JLL|
|.MM.NN|
+------+
Move 29: J-down
+------+
|BBICGE|
|.DICGE|
|.DIPPE|
|AHH...|
|A..JLL|
|.MMJNN|
+------+

Move 30: H-right
+------+
|BBICGE|
|.DICGE|
|.DIPPE|
|A...HH|
|A..JLL|
|.MMJNN|
+------+

Move 31: I-down
+------+
|BB.CGE|
|.DICGE|
|.DIPPE|
|A.I.HH|
|A..JLL|
|.MMJNN|
+------+

Move 32: D-down
+------+
|BB.CGE|
|..ICGE|
|.DIPPE|
|ADI.HH|
|A..JLL|
|.MMJNN|
+------+

Move 33: M-left
+------+
|BB.CGE|
|..ICGE|
|.DIPPE|
|ADI.HH|
|A..JLL|
|MM.JNN|
+------+

Move 34: J-up
+------+
|BB.CGE|
|..ICGE|
|.DIPPE|
|ADIJHH|
|A..JLL|
|MM..NN|
+------+

Move 35: N-left
+------+
|BB.CGE|
|..ICGE|
|.DIPPE|
|ADIJHH|
|A..JLL|
|MMNN..|
+------+

Move 36: B-right
+------+
|.BBCGE|
|..ICGE|
|.DIPPE|
|ADIJHH|
|A..JLL|
|MMNN..|
+------+

Move 37: N-right
+------+
|.BBCGE|
|..ICGE|
|.DIPPE|
|ADIJHH|
|A..JLL|
|MM.NN.|
+------+

Move 38: M-right
+------+
|.BBCGE|
|..ICGE|
|.DIPPE|
|ADIJHH|
|A..JLL|
|.MMNN.|
+------+

Move 39: N-right
+------+
|.BBCGE|
|..ICGE|
|.DIPPE|
|ADIJHH|
|A..JLL|
|.MM.NN|
+------+

Move 40: I-down
+------+
|.BBCGE|
|...CGE|
|.DIPPE|
|ADIJHH|
|A.IJLL|
|.MM.NN|
+------+

Move 41: M-right
+------+
|.BBCGE|
|...CGE|
|.DIPPE|
|ADIJHH|
|A.IJLL|
|..MMNN|
+------+

Move 42: A-down
+------+
|.BBCGE|
|...CGE|
|.DIPPE|
|.DIJHH|
|A.IJLL|
|A.MMNN|
+------+

Move 43: I-up
+------+
|.BBCGE|
|..ICGE|
|.DIPPE|
|.DIJHH|
|A..JLL|
|A.MMNN|
+------+

Move 44: D-down
+------+
|.BBCGE|
|..ICGE|
|..IPPE|
|.DIJHH|
|AD.JLL|
|A.MMNN|
+------+

Move 45: M-left
+------+
|.BBCGE|
|..ICGE|
|..IPPE|
|.DIJHH|
|AD.JLL|
|AMM.NN|
+------+

Move 46: N-left
+------+
|.BBCGE|
|..ICGE|
|..IPPE|
|.DIJHH|
|AD.JLL|
|AMMNN.|
+------+

Move 47: I-down
+------+
|.BBCGE|
|...CGE|
|..IPPE|
|.DIJHH|
|ADIJLL|
|AMMNN.|
+------+

Move 48: D-up
+------+
|.BBCGE|
|...CGE|
|.DIPPE|
|.DIJHH|
|A.IJLL|
|AMMNN.|
+------+

Move 49: B-left
+------+
|BB.CGE|
|...CGE|
|.DIPPE|
|.DIJHH|
|A.IJLL|
|AMMNN.|
+------+

Move 50: N-right
+------+
|BB.CGE|
|...CGE|
|.DIPPE|
|.DIJHH|
|A.IJLL|
|AMM.NN|
+------+

Move 51: J-down
+------+
|BB.CGE|
|...CGE|
|.DIPPE|
|.DI.HH|
|A.IJLL|
|AMMJNN|
+------+
Move 52: I-up
+------+
|BBICGE|
|..ICGE|
|.DIPPE|
|.D..HH|
|A..JLL|
|AMMJNN|
+------+

Move 53: H-left
+------+
|BBICGE|
|..ICGE|
|.DIPPE|
|.DHH..|
|A..JLL|
|AMMJNN|
+------+

Move 54: E-down
+------+
|BBICG.|
|..ICGE|
|.DIPPE|
|.DHH.E|
|A..JLL|
|AMMJNN|
+------+

Move 55: H-right
+------+
|BBICG.|
|..ICGE|
|.DIPPE|
|.D.HHE|
|A..JLL|
|AMMJNN|
+------+

Move 56: I-down
+------+
|BB.CG.|
|...CGE|
|.DIPPE|
|.DIHHE|
|A.IJLL|
|AMMJNN|
+------+

Move 57: A-up
+------+
|BB.CG.|
|A..CGE|
|ADIPPE|
|.DIHHE|
|..IJLL|
|.MMJNN|
+------+

Move 58: M-left
+------+
|BB.CG.|
|A..CGE|
|ADIPPE|
|.DIHHE|
|..IJLL|
|MM.JNN|
+------+

Move 59: I-down
+------+
|BB.CG.|
|A..CGE|
|AD.PPE|
|.DIHHE|
|..IJLL|
|MMIJNN|
+------+

Move 60: D-down
+------+
|BB.CG.|
|A..CGE|
|A..PPE|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 61: P-left
+------+
|BB.CG.|
|A..CGE|
|APP..E|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 62: G-down
+------+
|BB.C..|
|A..CGE|
|APP.GE|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 63: C-down
+------+
|BB....|
|A..CGE|
|APPCGE|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 64: B-right
+------+
|....BB|
|A..CGE|
|APPCGE|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 65: C-up
+------+
|...CBB|
|A..CGE|
|APP.GE|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 66: P-right
+------+
|...CBB|
|A..CGE|
|A.PPGE|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 67: A-up
+------+
|A..CBB|
|A..CGE|
|..PPGE|
|.DIHHE|
|.DIJLL|
|MMIJNN|
+------+

Move 68: D-up
+------+
|AD.CBB|
|AD.CGE|
|..PPGE|
|..IHHE|
|..IJLL|
|MMIJNN|
+------+

Move 69: P-left
+------+
|AD.CBB|
|AD.CGE|
|PP..GE|
|..IHHE|
|..IJLL|
|MMIJNN|
+------+

Move 70: I-up
+------+
|ADICBB|
|ADICGE|
|PPI.GE|
|...HHE|
|...JLL|
|MM.JNN|
+------+

Move 71: H-left
+------+
|ADICBB|
|ADICGE|
|PPI.GE|
|HH...E|
|...JLL|
|MM.JNN|
+------+

Move 72: I-down
+------+
|AD.CBB|
|AD.CGE|
|PP..GE|
|HHI..E|
|..IJLL|
|MMIJNN|
+------+

Move 73: P-right
+------+
|AD.CBB|
|AD.CGE|
|..PPGE|
|HHI..E|
|..IJLL|
|MMIJNN|
+------+

Move 74: A-down
+------+
|.D.CBB|
|AD.CGE|
|A.PPGE|
|HHI..E|
|..IJLL|
|MMIJNN|
+------+
Move 75: G-down
+------+
|.D.CBB|
|AD.C.E|
|A.PPGE|
|HHI.GE|
|..IJLL|
|MMIJNN|
+------+

Move 76: J-up
+------+
|.D.CBB|
|AD.C.E|
|A.PPGE|
|HHIJGE|
|..IJLL|
|MMI.NN|
+------+

Move 77: N-left
+------+
|.D.CBB|
|AD.C.E|
|A.PPGE|
|HHIJGE|
|..IJLL|
|MMINN.|
+------+

Move 78: P-left
+------+
|.D.CBB|
|AD.C.E|
|APP.GE|
|HHIJGE|
|..IJLL|
|MMINN.|
+------+

Move 79: J-up
+------+
|.D.CBB|
|AD.C.E|
|APPJGE|
|HHIJGE|
|..I.LL|
|MMINN.|
+------+

Move 80: L-left
+------+
|.D.CBB|
|AD.C.E|
|APPJGE|
|HHIJGE|
|..ILL.|
|MMINN.|
+------+

Move 81: E-down
+------+
|.D.CBB|
|AD.C..|
|APPJG.|
|HHIJGE|
|..ILLE|
|MMINNE|
+------+

Move 82: A-up
+------+
|AD.CBB|
|AD.C..|
|.PPJG.|
|HHIJGE|
|..ILLE|
|MMINNE|
+------+

Move 83: G-up
+------+
|AD.CBB|
|AD.CG.|
|.PPJG.|
|HHIJ.E|
|..ILLE|
|MMINNE|
+------+

Move 84: P-left
+------+
|AD.CBB|
|AD.CG.|
|PP.JG.|
|HHIJ.E|
|..ILLE|
|MMINNE|
+------+

Move 85: I-up
+------+
|ADICBB|
|ADICG.|
|PPIJG.|
|HH.J.E|
|...LLE|
|MM.NNE|
+------+

Move 86: L-left
+------+
|ADICBB|
|ADICG.|
|PPIJG.|
|HH.J.E|
|.LL..E|
|MM.NNE|
+------+

Move 87: G-down
+------+
|ADICBB|
|ADIC..|
|PPIJ..|
|HH.JGE|
|.LL.GE|
|MM.NNE|
+------+

Move 88: J-down
+------+
|ADICBB|
|ADIC..|
|PPI...|
|HH.JGE|
|.LLJGE|
|MM.NNE|
+------+

Move 89: L-left
+------+
|ADICBB|
|ADIC..|
|PPI...|
|HH.JGE|
|LL.JGE|
|MM.NNE|
+------+

Move 90: I-down
+------+
|AD.CBB|
|AD.C..|
|PP....|
|HHIJGE|
|LLIJGE|
|MMINNE|
+------+

Total moves: 90

Final Board State:
+------+
|AD.CBB|
|AD.C..|
|PP....|
|HHIJGE|
|LLIJGE|
|MMINNE|
+------+

Execution time: 0,205 seconds
```

(4)

```
Solving puzzle...
Using Greedy Best-First Search with Distance + Blocking
Cars heuristic...
Using Greedy Best-First Search with heuristic: Distance +
Blocking Cars
Goal state reached!
Visited nodes: 768
Total cost (steps): 101

Solution Path:
```

```
Initial state:
+------+
|AAAB.C|
|DEEB.C|
|D.FPPC|
|..FGHH|
|..FG..|
|.IIJJ.|
+------+

Move 1: J-right
+------+
|AAAB.C|
|DEEB.C|
|D.FPPC|
|..FGHH|
|..FG..|
|.II.JJ|
+------+

Move 2: I-right
+------+
|AAAB.C|
|DEEB.C|
|D.FPPC|
|..FGHH|
|..FG..|
|..IIJJ|
+------+

Move 3: D-down
+------+
|AAAB.C|
|.EEB.C|
|..FPPC|
|..FGHH|
|D.FG..|
|D.IIJJ|
+------+

Move 4: E-left
+------+
|AAAB.C|
|EE.B.C|
|..FPPC|
|..FGHH|
|D.FG..|
|D.IIJJ|
+------+

Move 5: I-left
+------+
|AAAB.C|
|EE.B.C|
|..FPPC|
|..FGHH|
|D.FG..|
|DII.JJ|
+------+

Move 6: J-left
+------+
|AAAB.C|
|EE.B.C|
|..FPPC|
|..FGHH|
|D.FG..|
|DIIJJ.|
+------+

Move 7: F-up
+------+
|AAAB.C|
|EEFB.C|
|..FPPC|
|..FGHH|
|D..G..|
|DIIJJ.|
+------+

Move 8: D-up
+------+
|AAAB.C|
|EEFB.C|
|D.FPPC|
|D.FGHH|
|...G..|
|.IIJJ.|
+------+

Move 9: J-right
+------+
|AAAB.C|
|EEFB.C|
|D.FPPC|
|D.FGHH|
|...G..|
|.II.JJ|
+------+

Move 10: G-down
+------+
|AAAB.C|
|EEFB.C|
|D.FPPC|
|D.F.HH|
|...G..|
|.IIGJJ|
+------+

Move 11: I-left
+------+
|AAAB.C|
|EEFB.C|
|D.FPPC|
|D.F.HH|
|...G..|
|II.GJJ|
+------+

Move 12: H-left
+------+
|AAAB.C|
|EEFB.C|
|D.FPPC|
|D.FHH.|
|...G..|
|II.GJJ|
+------+

Move 13: F-down
+------+
|AAAB.C|
|EE.B.C|
|D..PPC|
|D.FHH.|
|..FG..|
|IIFGJJ|
+------+

Move 14: D-down
+------+
|AAAB.C|
|EE.B.C|
|...PPC|
|D.FHH.|
|D.FG..|
|IIFGJJ|
+------+

Move 15: P-left
+------+
|AAAB.C|
|EE.B.C|
|PP...C|
|D.FHH.|
|D.FG..|
|IIFGJJ|
+------+

Move 16: F-up
+------+
|AAAB.C|
|EEFB.C|
|PPF..C|
|D.FHH.|
|D..G..|
|II.GJJ|
+------+

Move 17: I-right
+------+
|AAAB.C|
|EEFB.C|
|PPF..C|
|D.FHH.|
|D..G..|
|.IIGJJ|
+------+

Move 18: H-right
+------+
|AAAB.C|
|EEFB.C|
|PPF..C|
|D.F.HH|
|D..G..|
|.IIGJJ|
+------+

Move 19: G-up
+------+
|AAAB.C|
|EEFB.C|
|PPF..C|
|D.FGHH|
|D..G..|
|.II.JJ|
+------+

Move 20: I-right
+------+
|AAAB.C|
|EEFB.C|
|PPF..C|
|D.FGHH|
|D..G..|
|..IIJJ|
+------+

Move 21: D-down
+------+
|AAAB.C|
|EEFB.C|
|PPF..C|
|..FGHH|
|D..G..|
|D.IIJJ|
+------+

Move 22: B-down
+------+
|AAA..C|
|EEFB.C|
|PPFB.C|
|..FGHH|
|D..G..|
|D.IIJJ|
+------+
Move 23: A-right
+------+
|..AAAC|
|EEFB.C|
|PPFB.C|
|..FGHH|
|D..G..|
|D.IIJJ|
+------+

Move 24: I-left
+------+
|..AAAC|
|EEFB.C|
|PPFB.C|
|..FGHH|
|D..G..|
|DII.JJ|
+------+

Move 25: J-left
+------+
|..AAAC|
|EEFB.C|
|PPFB.C|
|..FGHH|
|D..G..|
|DIIJJ.|
+------+

Move 26: F-down
+------+
|..AAAC|
|EE.B.C|
|PPFB.C|
|..FGHH|
|D.FG..|
|DIIJJ.|
+------+

Move 27: E-right
+------+
|..AAAC|
|.EEB.C|
|PPFB.C|
|..FGHH|
|D.FG..|
|DIIJJ.|
+------+

Move 28: J-right
+------+
|..AAAC|
|.EEB.C|
|PPFB.C|
|..FGHH|
|D.FG..|
|DII.JJ|
+------+

Move 29: I-right
+------+
|..AAAC|
|.EEB.C|
|PPFB.C|
|..FGHH|
|D.FG..|
|D.IIJJ|
+------+

Move 30: D-up
+------+
|..AAAC|
|.EEB.C|
|PPFB.C|
|D.FGHH|
|D.FG..|
|..IIJJ|
+------+

Move 31: E-left
+------+
|..AAAC|
|EE.B.C|
|PPFB.C|
|D.FGHH|
|D.FG..|
|..IIJJ|
+------+

Move 32: I-left
+------+
|..AAAC|
|EE.B.C|
|PPFB.C|
|D.FGHH|
|D.FG..|
|.II.JJ|
+------+

Move 33: G-down
+------+
|..AAAC|
|EE.B.C|
|PPFB.C|
|D.F.HH|
|D.FG..|
|.IIGJJ|
+------+

Move 34: H-left
+------+
|..AAAC|
|EE.B.C|
|PPFB.C|
|D.FHH.|
|D.FG..|
|.IIGJJ|
+------+

Move 35: F-up
+------+
|..AAAC|
|EEFB.C|
|PPFB.C|
|D.FHH.|
|D..G..|
|.IIGJJ|
+------+

Move 36: D-down
+------+
|..AAAC|
|EEFB.C|
|PPFB.C|
|..FHH.|
|D..G..|
|DIIGJJ|
+------+

Move 37: C-down
+------+
|..AAA.|
|EEFB..|
|PPFB.C|
|..FHHC|
|D..G.C|
|DIIGJJ|
+------+

Move 38: F-down
+------+
|..AAA.|
|EE.B..|
|PPFB.C|
|..FHHC|
|D.FG.C|
|DIIGJJ|
+------+

Move 39: E-right
+------+
|..AAA.|
|.EEB..|
|PPFB.C|
|..FHHC|
|D.FG.C|
|DIIGJJ|
+------+

Move 40: D-up
+------+
|..AAA.|
|.EEB..|
|PPFB.C|
|D.FHHC|
|D.FG.C|
|.IIGJJ|
+------+

Move 41: C-up
+------+
|..AAA.|
|.EEB.C|
|PPFB.C|
|D.FHHC|
|D.FG..|
|.IIGJJ|
+------+

Move 42: A-right
+------+
|...AAA|
|.EEB.C|
|PPFB.C|
|D.FHHC|
|D.FG..|
|.IIGJJ|
+------+

Move 43: E-left
+------+
|...AAA|
|EE.B.C|
|PPFB.C|
|D.FHHC|
|D.FG..|
|.IIGJJ|
+------+

Move 44: F-up
+------+
|..FAAA|
|EEFB.C|
|PPFB.C|
|D..HHC|
|D..G..|
|.IIGJJ|
+------+

Move 45: H-left
+------+
|..FAAA|
|EEFB.C|
|PPFB.C|
|DHH..C|
|D..G..|
|.IIGJJ|
+------+
Move 46: G-up
+------+
|..FAAA|
|EEFB.C|
|PPFB.C|
|DHHG.C|
|D..G..|
|.II.JJ|
+------+

Move 47: J-left
+------+
|..FAAA|
|EEFB.C|
|PPFB.C|
|DHHG.C|
|D..G..|
|.IIJJ.|
+------+

Move 48: C-down
+------+
|..FAAA|
|EEFB..|
|PPFB..|
|DHHG.C|
|D..G.C|
|.IIJJC|
+------+

Move 49: D-down
+------+
|..FAAA|
|EEFB..|
|PPFB..|
|.HHG.C|
|D..G.C|
|DIIJJC|
+------+

Move 50: H-left
+------+
|..FAAA|
|EEFB..|
|PPFB..|
|HH.G.C|
|D..G.C|
|DIIJJC|
+------+

Move 51: F-down
+------+
|...AAA|
|EEFB..|
|PPFB..|
|HHFG.C|
|D..G.C|
|DIIJJC|
+------+

Move 52: A-left
+------+
|AAA...|
|EEFB..|
|PPFB..|
|HHFG.C|
|D..G.C|
|DIIJJC|
+------+

Move 53: B-up
+------+
|AAAB..|
|EEFB..|
|PPF...|
|HHFG.C|
|D..G.C|
|DIIJJC|
+------+

Move 54: F-down
+------+
|AAAB..|
|EE.B..|
|PPF...|
|HHFG.C|
|D.FG.C|
|DIIJJC|
+------+

Move 55: E-right
+------+
|AAAB..|
|.EEB..|
|PPF...|
|HHFG.C|
|D.FG.C|
|DIIJJC|
+------+

Move 56: C-up
+------+
|AAAB.C|
|.EEB.C|
|PPF..C|
|HHFG..|
|D.FG..|
|DIIJJ.|
+------+

Move 57: J-right
+------+
|AAAB.C|
|.EEB.C|
|PPF..C|
|HHFG..|
|D.FG..|
|DII.JJ|
+------+

Move 58: G-down
+------+
|AAAB.C|
|.EEB.C|
|PPF..C|
|HHF...|
|D.FG..|
|DIIGJJ|
+------+

Move 59: B-down
+------+
|AAA..C|
|.EE..C|
|PPFB.C|
|HHFB..|
|D.FG..|
|DIIGJJ|
+------+

Move 60: E-right
+------+
|AAA..C|
|..EE.C|
|PPFB.C|
|HHFB..|
|D.FG..|
|DIIGJJ|
+------+

Move 61: C-down
+------+
|AAA...|
|..EE..|
|PPFB.C|
|HHFB.C|
|D.FG.C|
|DIIGJJ|
+------+

Move 62: E-right
+------+
|AAA...|
|....EE|
|PPFB.C|
|HHFB.C|
|D.FG.C|
|DIIGJJ|
+------+

Move 63: B-up
+------+
|AAAB..|
|...BEE|
|PPF..C|
|HHF..C|
|D.FG.C|
|DIIGJJ|
+------+

Move 64: G-up
+------+
|AAAB..|
|...BEE|
|PPF..C|
|HHFG.C|
|D.FG.C|
|DII.JJ|
+------+

Move 65: J-left
+------+
|AAAB..|
|...BEE|
|PPF..C|
|HHFG.C|
|D.FG.C|
|DIIJJ.|
+------+

Move 66: C-down
+------+
|AAAB..|
|...BEE|
|PPF...|
|HHFG.C|
|D.FG.C|
|DIIJJC|
+------+

Move 67: B-down
+------+
|AAA...|
|...BEE|
|PPFB..|
|HHFG.C|
|D.FG.C|
|DIIJJC|
+------+

Move 68: A-right
+------+
|...AAA|
|...BEE|
|PPFB..|
|HHFG.C|
|D.FG.C|
|DIIJJC|
+------+
Move 69: F-up
+------+
|..FAAA|
|..FBEE|
|PPFB..|
|HH.G.C|
|D..G.C|
|DIIJJC|
+------+

Move 70: H-right
+------+
|..FAAA|
|..FBEE|
|PPFB..|
|.HHG.C|
|D..G.C|
|DIIJJC|
+------+

Move 71: D-up
+------+
|..FAAA|
|..FBEE|
|PPFB..|
|DHHG.C|
|D..G.C|
|.IIJJC|
+------+

Move 72: I-left
+------+
|..FAAA|
|..FBEE|
|PPFB..|
|DHHG.C|
|D..G.C|
|II.JJC|
+------+

Move 73: J-left
+------+
|..FAAA|
|..FBEE|
|PPFB..|
|DHHG.C|
|D..G.C|
|IIJJ.C|
+------+

Move 74: C-up
+------+
|..FAAA|
|..FBEE|
|PPFB.C|
|DHHG.C|
|D..G.C|
|IIJJ..|
+------+

Move 75: J-right
+------+
|..FAAA|
|..FBEE|
|PPFB.C|
|DHHG.C|
|D..G.C|
|II..JJ|
+------+

Move 76: I-right
+------+
|..FAAA|
|..FBEE|
|PPFB.C|
|DHHG.C|
|D..G.C|
|..IIJJ|
+------+

Move 77: D-down
+------+
|..FAAA|
|..FBEE|
|PPFB.C|
|.HHG.C|
|D..G.C|
|D.IIJJ|
+------+

Move 78: I-left
+------+
|..FAAA|
|..FBEE|
|PPFB.C|
|.HHG.C|
|D..G.C|
|DII.JJ|
+------+

Move 79: H-left
+------+
|..FAAA|
|..FBEE|
|PPFB.C|
|HH.G.C|
|D..G.C|
|DII.JJ|
+------+

Move 80: G-down
+------+
|..FAAA|
|..FBEE|
|PPFB.C|
|HH...C|
|D..G.C|
|DIIGJJ|
+------+

Move 81: H-right
+------+
|..FAAA|
|..FBEE|
|PPFB.C|
|...HHC|
|D..G.C|
|DIIGJJ|
+------+

Move 82: F-down
+------+
|...AAA|
|...BEE|
|PPFB.C|
|..FHHC|
|D.FG.C|
|DIIGJJ|
+------+

Move 83: D-up
+------+
|...AAA|
|...BEE|
|PPFB.C|
|D.FHHC|
|D.FG.C|
|.IIGJJ|
+------+

Move 84: I-left
+------+
|...AAA|
|...BEE|
|PPFB.C|
|D.FHHC|
|D.FG.C|
|II.GJJ|
+------+

Move 85: F-down
+------+
|...AAA|
|...BEE|
|PP.B.C|
|D.FHHC|
|D.FG.C|
|IIFGJJ|
+------+

Move 86: P-right
+------+
|...AAA|
|...BEE|
|.PPB.C|
|D.FHHC|
|D.FG.C|
|IIFGJJ|
+------+

Move 87: D-up
+------+
|D..AAA|
|D..BEE|
|.PPB.C|
|..FHHC|
|..FG.C|
|IIFGJJ|
+------+

Move 88: P-left
+------+
|D..AAA|
|D..BEE|
|PP.B.C|
|..FHHC|
|..FG.C|
|IIFGJJ|
+------+

Move 89: F-up
+------+
|D..AAA|
|D.FBEE|
|PPFB.C|
|..FHHC|
|...G.C|
|II.GJJ|
+------+

Move 90: I-right
+------+
|D..AAA|
|D.FBEE|
|PPFB.C|
|..FHHC|
|...G.C|
|.IIGJJ|
+------+

Move 91: F-up
+------+
|D.FAAA|
|D.FBEE|
|PPFB.C|
|...HHC|
|...G.C|
|.IIGJJ|
+------+
Move 92: H-left
+------+
|D.FAAA|
|D.FBEE|
|PPFB.C|
|HH...C|
|...G.C|
|.IIGJJ|
+------+

Move 93: I-left
+------+
|D.FAAA|
|D.FBEE|
|PPFB.C|
|HH...C|
|...G.C|
|II.GJJ|
+------+

Move 94: F-down
+------+
|D..AAA|
|D..BEE|
|PP.B.C|
|HHF..C|
|..FG.C|
|IIFGJJ|
+------+

Move 95: P-right
+------+
|D..AAA|
|D..BEE|
|.PPB.C|
|HHF..C|
|..FG.C|
|IIFGJJ|
+------+

Move 96: G-up
+------+
|D..AAA|
|D..BEE|
|.PPB.C|
|HHFG.C|
|..FG.C|
|IIF.JJ|
+------+

Move 97: J-left
+------+
|D..AAA|
|D..BEE|
|.PPB.C|
|HHFG.C|
|..FG.C|
|IIFJJ.|
+------+

Move 98: C-down
+------+
|D..AAA|
|D..BEE|
|.PPB..|
|HHFG.C|
|..FG.C|
|IIFJJC|
+------+

Move 99: D-down
+------+
|...AAA|
|D..BEE|
|DPPB..|
|HHFG.C|
|..FG.C|
|IIFJJC|
+------+

Move 100: A-left
+------+
|AAA...|
|D..BEE|
|DPPB..|
|HHFG.C|
|..FG.C|
|IIFJJC|
+------+

Move 101: B-up
+------+
|AAAB..|
|D..BEE|
|DPP...|
|HHFG.C|
|..FG.C|
|IIFJJC|
+------+

Total moves: 101

Final Board State:
+------+
|AAAB..|
|D..BEE|
|DPP...|
|HHFG.C|
|..FG.C|
|IIFJJC|
+------+

Execution time: 0,085 seconds
```

(5)

```
Solving puzzle...
Using Greedy Best-First Search with Distance + Blocking Cars heuristic...
Using Greedy Best-First Search with heuristic: Distance + Blocking Cars
Goal state reached!
Visited nodes: 27
Total cost (steps): 13

Solution Path:
Initial state:
+--------+
|JJJJJJJJ|
|IIIIIIII|
|PPBCDEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|.....EFG|
|...HHHHH|
+--------+

Move 1: B-down
+--------+
|JJJJJJJ|
|IIIIIII|
|PP.CDEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..B..EFG|
|...HHHHH|
+--------+

Move 2: C-down
+--------+
|JJJJJJJ|
|IIIIIII|
|PP..DEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BC.EFG|
|...HHHHH|
+--------+

Move 3: P-right
+--------+
|JJJJJJJ|
|IIIIIII|
|..PPDEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BC.EFG|
|...HHHHH|
+--------+

Move 4: D-down
+--------+
|JJJJJJJ|
|IIIIIII|
|..PP.EFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|...HHHHH|
+--------+
Move 5: P-right
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PPEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|...HHHHH|
+--------+

Move 6: A-down
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PPEFG|
|..BCDEFG|
|..BCDEFG|
|.ABCDEFG|
|.ABCDEFG|
|...HHHHH|
+--------+

Move 7: H-left
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PPEFG|
|..BCDEFG|
|..BCDEFG|
|.ABCDEFG|
|.ABCDEFG|
|..HHHHH.|
+--------+

Move 8: G-down
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PPEF.|
|..BCDEFG|
|..BCDEFG|
|.ABCDEFG|
|.ABCDEFG|
|..HHHHHG|
+--------+

Move 9: H-left
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PPEF.|
|..BCDEFG|
|..BCDEFG|
|.ABCDEFG|
|.ABCDEFG|
|.HHHHH.G|
+--------+

Move 10: F-down
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PPE..|
|..BCDEFG|
|..BCDEFG|
|.ABCDEFG|
|.ABCDEFG|
|.HHHHHFG|
+--------+

Move 11: A-up
+--------+
|JJJJJJJJ|
|IIIIIIII|
|.A.PPE..|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|..BCDEFG|
|.HHHHHFG|
+--------+

Move 12: H-left
+--------+
|JJJJJJJJ|
|IIIIIIII|
|.A.PPE..|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|..BCDEFG|
|HHHHH.FG|
+--------+

Move 13: E-down
+--------+
|JJJJJJJ|
|IIIIIII|
|.A.PP...|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|..BCDEFG|
|HHHHHEFG|
+--------+

Total moves: 13

Final Board State:
+--------+
|JJJJJJJ|
|IIIIIII|
|.A.PP...|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|..BCDEFG|
|HHHHHEFG|
+--------+

Execution time: 0,034 seconds
```

D. Iterative Deepening A* Search

(1)

```
Solving puzzle...
Using Iterative Deepening A* with Distance + Blocking Cars
heuristic...
Using IDA* with heuristic: Distance + Blocking Cars
Current threshold: 9
Goal state reached!
Visited nodes: 61
Total cost (steps): 7

Solution Path:
Initial state:
```

```
+------+
|AAB..F|
|..BCDF|
|GPPCDF|
|GH.III|
|GHJ...|
|LLJMM.|
+------+

Move 1: C-up
+------+
|AABC.F|
|..BCDF|
|GPP.DF|
|GH.III|
|GHJ...|
|LLJMM.|
+------+

Move 2: P-right
+------+
|AABC.F|
|..BCDF|
|G.PPDF|
|GH.III|
|GHJ...|
|LLJMM.|
+------+

Move 3: D-up
+------+
|AABCDF|
|..BCDF|
|G.PP.F|
|GH.III|
|GHJ...|
|LLJMM.|
+------+

Move 4: P-left
+------+
|AABCDF|
|..BCDF|
|GPP..F|
|GH.III|
|GHJ...|
|LLJMM.|
+------+

Move 5: P-right
+------+
|AABCDF|
|..BCDF|
|G..PPF|
|GH.III|
|GHJ...|
|LLJMM.|
+------+

Move 6: I-left
+------+
|AABCDF|
|..BCDF|
|G..PPF|
|GHIII.|
|GHJ...|
|LLJMM.|
+------+

Move 7: F-down
+------+
|AABCD.|
|..BCD.|
|G..PP.|
|GHIIIF|
|GHJ..F|
|LLJMMF|
+------+

Total moves: 7

Final Board State:
+------+
|AABCD.|
|..BCD.|
|G..PP.|
|GHIIIF|
|GHJ..F|
|LLJMMF|
+------+

Execution time: 0,022 seconds
```

(2)

```
Solving puzzle...
Using Iterative Deepening A* with Distance + Blocking Cars
heuristic...
Using IDA* with heuristic: Distance + Blocking Cars
Current threshold: 8
Goal state reached!
Visited nodes: 98
Total cost (steps): 6

Solution Path:
Initial state:
+------+
|ABCDDD|
|ABCEE.|
|PPF..G|
|HHF.IG|
|JJ..IG|
|...LLL|
+------+

Move 1: F-down
+------+
|ABCDDD|
|ABCEE.|
|PP...G|
|HHF.IG|
|JJF.IG|
|...LLL|
+------+

Move 2: P-right
+------+
|ABCDDD|
|ABCEE.|
|.PP..G|
|HHF.IG|
|JJF.IG|
|...LLL|
+------+

Move 3: P-right
+------+
|ABCDDD|
|ABCEE.|
|..PP.G|
|HHF.IG|
|JJF.IG|
|...LLL|
+------+

Move 4: P-right
+------+
|ABCDDD|
|ABCEE.|
|...PPG|
|HHF.IG|
|JJF.IG|
|...LLL|
+------+

Move 5: L-left
+------+
|ABCDDD|
|ABCEE.|
|...PPG|
|HHF.IG|
|JJF.IG|
|..LLL.|
+------+

Move 6: G-down
+------+
|ABCDDD|
|ABCEE.|
|...PP.|
|HHF.IG|
|JJF.IG|
|..LLLG|
+------+

Total moves: 6

Final Board State:
+------+
|ABCDDD|
|ABCEE.|
|...PP.|
|HHF.IG|
|JJF.IG|
|..LLLG|
+------+
Execution time: 0,022 seconds
```

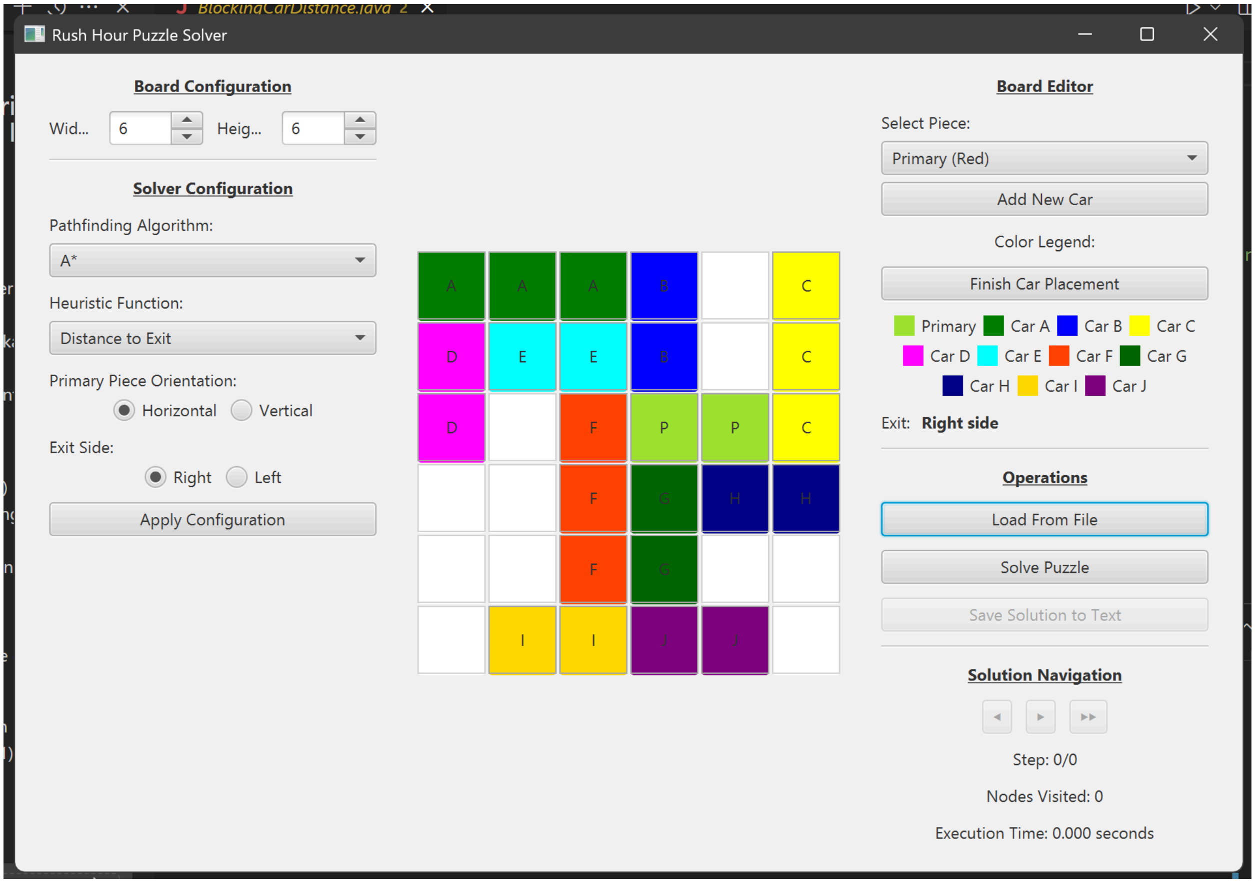
(3)

```
6 6
12
 F..BAA
 FDCB..
KFDCPPG
 III.HG
 ...JHG
 .LLJMM


Solving puzzle...
Using Iterative Deepening A* with Distance + Blocking Cars
heuristic...
Nodes visited: 56

Execution time: 0,019 seconds

Solution Path:
Initial state:
+------+
|F..BAA|
|FDCB..|
|FDCPPG|
|III.HG|
|...JHG|
|.LLJMM|
+------+S

Move 1: C-up
+------+
|F.CBAA|
|FDCB..|
|FD.PPG|
|III.HG|
|...JHG|
|.LLJMM|
+------+

Move 2: P-left
+------+
|F.CBAA|
|FDCB..|
|FDPP.G|
|III.HG|
|...JHG|
|.LLJMM|
+------+

Move 3: D-up
+------+
|FDCBAA|
|FDCB..|
|F.PP.G|
|III.HG|
|...JHG|
|.LLJMM|
+------+

Move 4: P-left
+------+
|FDCBAA|
|FDCB..|
|FPP..G|
|III.HG|
|...JHG|
|.LLJMM|
+------+

Move 5: B-down
+------+
|FDC.AA|
|FDCB..|
|FPPB.G|
|III.HG|
|...JHG|
|.LLJMM|
+------+

Move 6: I-right
+------+
|FDC.AA|
|FDCB..|
|FPPB.G|
|.IIIHG|
|...JHG|
|.LLJMM|
+------+

Move 7: F-down
+------+
|.DC.AA|
|.DCB..|
|.PPB.G|
|FIIIHG|
|F..JHG|
|FLLJMM|
+------+

Total moves: 7

Final Board State:
+------+
|.DC.AA|
|.DCB..|
|.PPB.G|
|FIIIHG|
|F..JHG|
|FLLJMM|
+------+

Execution time: 0,019 seconds
```

(5)

```
Solving puzzle...
Using Iterative Deepening A* with Distance + Blocking Cars
heuristic...
Using IDA* with heuristic: Distance + Blocking Cars
Current threshold: 18
Goal state reached!
Visited nodes: 208
Total cost (steps): 14

Solution Path:
Initial state:
+--------+
|JJJJJJJJ|
|IIIIIIII|
|PPBCDEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|.....EFG|
|...HHHHH|
+--------+

Move 1: B-down
+--------+
|JJJJJJJ|
|IIIIIII|
|PP.CDEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..B..EFG|
|...HHHHH|
+--------+

Move 2: P-right
+--------+
|JJJJJJJ|
|IIIIIII|
|.PPCDEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..B..EFG|
|...HHHHH|
+--------+

Move 3: A-down
+--------+
|JJJJJJJ|
|IIIIIII|
|.PPCDEFG|
|..BCDEFG|
|.ABCDEFG|
|.ABCDEFG|
|..B..EFG|
|...HHHHH|
+--------+

Move 4: C-down
+--------+
|JJJJJJJ|
|IIIIIII|
|.PP.DEFG|
|..BCDEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BC.EFG|
|...HHHHH|
+--------+
Move 5: P-right
+--------+
|JJJJJJJJ|
|IIIIIIII|
|..PPDEFG|
|..BCDEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BC.EFG|
|...HHHHH|
+--------+

Move 6: A-up
+--------+
|JJJJJJJJ|
|IIIIIIII|
|..PPDEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BC.EFG|
|...HHHHH|
+--------+

Move 7: D-down
+--------+
|JJJJJJJJ|
|IIIIIIII|
|..PP.EFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|...HHHHH|
+--------+

Move 8: P-right
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PPEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|...HHHHH|
+--------+

Move 9: H-left
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PPEFG|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|..HHHHH.|
+--------+

Move 10: G-down
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PPEF.|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|..HHHHHG|
+--------+

Move 11: H-left
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PPEF.|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|.HHHHH.G|
+--------+

Move 12: F-down
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PPE..|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|.HHHHHFG|
+--------+

Move 13: H-left
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PPE..|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|HHHHH.FG|
+--------+

Move 14: E-down
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PP...|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|HHHHHEFG|
+--------+

Total moves: 14

Final Board State:
+--------+
|JJJJJJJJ|
|IIIIIIII|
|...PP...|
|.ABCDEFG|
|.ABCDEFG|
|..BCDEFG|
|..BCDEFG|
|HHHHHEFG|
+--------+

Execution time: 0,025 seconds
```

GUI



Rush Hour Puzzle Solver

Board Configuration

Wid... 6 Heig... 6

Solver Configuration

Pathfinding Algorithm: A*

Heuristic Function: Distance to Exit

Primary Piece Orientation: ● Horizontal ○ Vertical

Exit Side: ● Right ○ Left

A	A	A	B		C
D	E	E	B		C
D		F	P	P	C
		F	G	H	H
		F	G		
	I	I	J	J	

Board Editor

Select Piece: Primary (Red)

Color Legend:

🟩 Primary 🟩 Car A 🟦 Car B 🟨 Car C
🟪 Car D 🟦 Car E 🟥 Car F 🟩 Car G
🟦 Car H 🟨 Car I 🟪 Car J

Exit: **Right side**

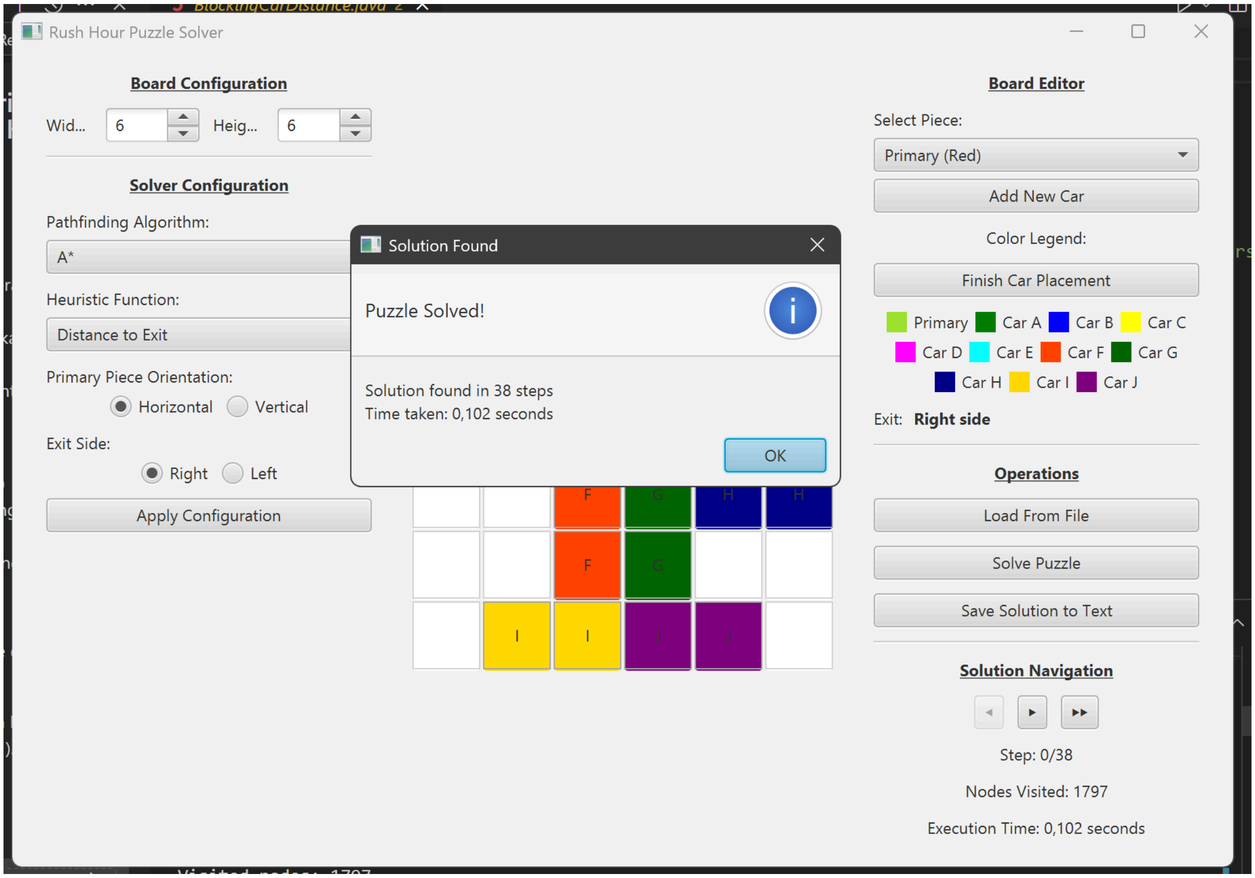
Operations

Solution Navigation

Step: 0/0

Nodes Visited: 0

Execution Time: 0.000 seconds



Rush Hour Puzzle Solver

Board Configuration

Wid... 6 Heig... 6

Solver Configuration

Pathfinding Algorithm: A*

Heuristic Function: Distance to Exit

Primary Piece Orientation: ● Horizontal ○ Vertical

Exit Side: ● Right ○ Left

Solution Found

Puzzle Solved!

Solution found in 38 steps
Time taken: 0,102 seconds

Board Editor

Select Piece: Primary (Red)

Color Legend:

Exit: **Right side**

Operations

Solution Navigation

Step: 0/38

Nodes Visited: 1797

Execution Time: 0,102 seconds

BAB V
ANALISIS IMPLEMENTASI

UCS (Uniform Cost Search)

Kompleksitas Waktu

- O(b^d * log(b^d)) dimana:
  b = faktor percabangan (jumlah gerakan yang mungkin per state)
  d = kedalaman solusi (jumlah langkah minimum ke solusi)
  log(b^d) = kompleksitas operasi antrian prioritas

Operasi utama:

- Ekstraksi dari antrian prioritas: O(log N)
- Pemeriksaan state yang sudah dikunjungi: O(1) dengan HashMap
- Penghasilan successor states: O(b) per node

Kompleksitas Ruang

- O(b^d) untuk menyimpan semua state dalam antrian dan costMap
  Untuk Rush Hour, UCS efektif sama dengan BFS karena setiap gerakan memiliki biaya yang sama (1).

A* (A Star)

Kompleksitas Waktu

- Kasus terburuk: O(b^d * log(b^d)) seperti UCS
- Kasus terbaik: O(d) dengan heuristik sempurna
- Rata-rata: O(b^(εd)) dimana ε < 1 tergantung kualitas heuristik

Operasi tambahan dibanding UCS:

- Perhitungan heuristik untuk setiap state

Kompleksitas Ruang

- O(b^d) sama seperti UCS
  A* lebih efisien dari UCS dengan heuristik yang baik seperti BlockingCarDistance.

Greedy BFS (Greedy Best-First Search)

Kompleksitas Waktu

- Kasus terburuk: O(b^m * log(b^m)) dimana m = kedalaman maksimum
- Rata-rata: Jauh lebih baik dari A dan UCS* tetapi tidak menjamin optimalitas

Kompleksitas Ruang

- O(b^m) untuk menyimpan antrian dan visitedMap
- Greedy BFS lebih cepat tetapi bisa menghasilkan solusi suboptimal karena hanya mempertimbangkan h(n) tanpa g(n).

IDA* (Iterative Deepening A*)

Kompleksitas Waktu

- O(b^d) tetapi dengan konstanta yang lebih tinggi karena mengulangi pencarian
- Melakukan multiple DFS dengan batas threshold yang meningkat

Kompleksitas Ruang

- O(d) - keuntungan utama dibanding algoritma lain
- Hanya menyimpan jalur saat ini dan set state yang dikunjungi dalam iterasi saat ini
- IDA* sangat efisien dalam penggunaan memori tetapi bisa lebih lambat karena mengulangi pencarian.

LAMPIRAN

Pranala repositori : https://github.com/Kurosue/Tucil3_13523011_13523028

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	✓	