LAPORAN TUGAS KECIL 3
IF2211 - Strategi Algoritma
Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding

Disusun oleh:
Muhammad Hazim Ramadhan Prajoda 13523009

Dosen Pengajar:
Dr. Nur Ulfa Maulidevi, S.T, M.Sc

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
Semester II tahun 2024/2025

Kata Pengantar

Segala puji dan syukur saya panjatkan ke hadirat Allah Subhanahu wa Ta'ala, Tuhan Yang Maha Esa, atas limpahan rahmat dan karunia-Nya, sehingga saya dapat menyelesaikan Laporan Tugas Kecil 3 untuk mata kuliah IF2211 Strategi Algoritma yang berjudul "Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding" tepat pada waktunya. Laporan ini disusun sebagai salah satu bentuk tugas perkuliahan yang bertujuan untuk mengaplikasikan teori algoritma pathfinding yang telah dipelajari selama perkuliahan dalam menyelesaikan permasalahan puzzle logika Rush Hour.

Dalam pengerjaan tugas kecil ini, saya telah mengimplementasikan tiga algoritma pathfinding yaitu Greedy Best First Search, Uniform Cost Search (UCS), dan A* dengan berbagai fungsi heuristik untuk menemukan solusi optimal dalam permainan Rush Hour. Melalui tugas ini, saya telah memperdalam pemahaman tentang bagaimana algoritma-algoritma tersebut bekerja dalam konteks nyata serta bagaimana membandingkan efektivitas dan efisiensinya.

Saya menyampaikan rasa terima kasih yang sebesar-besarnya kepada dosen pengampu mata kuliah IF2211 Strategi Algoritma atas bimbingan dan arahannya selama proses perkuliahan. Ucapan terima kasih juga saya sampaikan kepada para asisten dosen yang telah memberikan penjelasan tambahan terhadap tugas-tugas di mata kuliah ini serta teman-teman yang telah berdiskusi dan saling mendukung dalam penyelesaian tugas ini. Dengan pengembangan aplikasi web interaktif menggunakan Next.js dan React, saya telah berusaha memvisualisasikan proses pencarian solusi sehingga dapat lebih mudah dipahami bagaimana algoritma pathfinding bekerja dalam konteks permainan Rush Hour. Implementasi ini diharapkan dapat membantu mahasiswa lain untuk memahami konsep algoritma pathfinding dengan cara yang lebih intuitif.

Akhir kata, saya berharap laporan ini sudah memenuhi ketentuan untuk tugas ini. Saya menyadari bahwa laporan ini masih memiliki kekurangan. Oleh karena itu, saya dengan senang hati menerima kritik dan saran yang membangun demi perbaikan di masa mendatang.

Bandung, 21 Mei 2025
Muhammad Hazim Ramadhan Prajoda

Contents

1 Penjelasan Algoritma UCS, Greedy Best First Search, dan A*

1.1 Uniform Cost Search (UCS)

Uniform Cost Search (UCS) adalah algoritma pencarian berbasis grafik yang mengeksplorasi state space dengan prioritas berdasarkan biaya kumulatif dari state awal. Algoritma ini bekerja dengan memperluas node yang memiliki biaya terendah terlebih dahulu.

Algoritma UCS dapat dijelaskan sebagai berikut:

1. Inisialisasi frontier (priority queue) dengan state awal, biaya 0, dan jalur kosong.
2. Inisialisasi explored set sebagai himpunan kosong.
3. Loop hingga frontier kosong:
   (a) Keluarkan node dengan biaya terendah dari frontier.
   (b) Jika node adalah state tujuan, return jalur solusi.
   (c) Tambahkan node ke explored set.
   (d) Untuk setiap successor dari node:
      i. Jika successor tidak dalam explored set dan tidak dalam frontier:
         A. Tambahkan successor ke frontier dengan biaya = biaya node + biaya aksi.
      ii. Jika successor sudah dalam frontier dengan biaya lebih tinggi:
         A. Perbarui biaya successor dan jalurnya dalam frontier.
4. Return failure jika frontier kosong.

1.2 Greedy Best First Search

Greedy Best First Search menggunakan fungsi heuristik untuk menentukan node mana yang akan diperluas selanjutnya, dengan memilih node yang memiliki estimasi biaya terendah ke tujuan.

Algoritma Greedy Best First Search dapat dijelaskan sebagai berikut:

1. Inisialisasi frontier (priority queue) dengan state awal.
2. Inisialisasi explored set sebagai himpunan kosong.
3. Loop hingga frontier kosong:
   (a) Keluarkan node dengan nilai heuristik terendah dari frontier.
   (b) Jika node adalah state tujuan, return jalur solusi.
   (c) Tambahkan node ke explored set.
   (d) Untuk setiap successor dari node:
      i. Jika successor tidak dalam explored set dan tidak dalam frontier:
         A. Hitung nilai heuristik h(successor).
         B. Tambahkan successor ke frontier dengan prioritas berdasarkan h(successor).

ii. Jika successor sudah dalam frontier dengan nilai heuristik lebih tinggi:
    A. Perbarui nilai heuristik successor dalam frontier.

4. Return failure jika frontier kosong.

1.3 A* Search

A* Search menggabungkan informasi dari biaya sejauh ini (seperti UCS) dan estimasi biaya ke tujuan (seperti Greedy) untuk menentukan node mana yang akan diperluas selanjutnya.

Algoritma A* Search dapat dijelaskan sebagai berikut:

1. Inisialisasi frontier (priority queue) dengan state awal, g(start)=0, dan f(start)=h(start).
2. Inisialisasi explored set sebagai himpunan kosong.
3. Loop hingga frontier kosong:
    (a) Keluarkan node dengan nilai f(n) terendah dari frontier.
    (b) Jika node adalah state tujuan, return jalur solusi.
    (c) Tambahkan node ke explored set.
    (d) Untuk setiap successor dari node:
        i. Hitung g(successor) = g(node) + biaya dari node ke successor.
        ii. Jika successor tidak dalam explored set dan tidak dalam frontier:
            A. Hitung f(successor) = g(successor) + h(successor).
            B. Tambahkan successor ke frontier dengan prioritas berdasarkan f(successor).
        iii. Jika successor sudah dalam frontier dengan nilai g lebih tinggi:
            A. Perbarui g(successor) dan f(successor) dalam frontier.
4. Return failure jika frontier kosong.

2 Analisis Algoritma UCS, Greedy Best First Search, dan A*

2.1 Definisi dari f(n) dan g(n)

Pada algoritma pencarian, fungsi evaluasi memainkan peran krusial dalam menentukan urutan eksplorasi node:

- **Uniform Cost Search (UCS):**

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

  Di mana $g(n)$ adalah biaya aktual dari state awal menuju state n. UCS hanya mempertimbangkan biaya yang sudah dilalui, tanpa memperhitungkan estimasi biaya ke tujuan.

- **Greedy Best First Search:**

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

  Di mana $h(n)$ adalah estimasi biaya dari state n menuju state tujuan. Greedy hanya mempertimbangkan estimasi biaya ke depan, mengabaikan biaya yang sudah dilalui.

- **A* Search:**

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

  A* mempertimbangkan keduanya: biaya yang sudah dilalui $g(n)$ dan estimasi biaya ke depan $h(n)$.

2.2 Admissibility Heuristik pada Algoritma A*

Heuristik $h(n)$ dikatakan admissible jika tidak pernah overestimasi biaya sebenarnya dari node n ke tujuan. Secara formal:

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

Di mana $h^*(n)$ adalah biaya sebenarnya dari node n ke tujuan.

Dalam konteks Rush Hour, heuristik Manhattan Distance (jarak kendaraan utama ke pintu keluar) adalah admissible karena tidak pernah melebihi jumlah langkah minimum yang diperlukan untuk mencapai pintu keluar. Setiap gerakan kendaraan hanya dapat menggeser satu atau beberapa langkah dalam satu arah, sehingga jarak langsung tidak pernah melebihi jumlah langkah yang diperlukan.

Heuristik Blocking Pieces juga admissible karena setiap kendaraan yang menghalangi pasti memerlukan minimal satu langkah untuk dipindahkan, sehingga jumlah kendaraan penghalang tidak pernah melebihi jumlah langkah minimum yang diperlukan.

2.3 Perbandingan Algoritma UCS dengan BFS pada Rush Hour

Pada penyelesaian Rush Hour, algoritma UCS dan BFS akan menghasilkan urutan eksplorasi node dan solusi yang sama jika semua gerakan memiliki biaya yang sama, dengan kondisi:

- Setiap gerakan kendaraan dianggap memiliki biaya 1, terlepas dari berapa langkah kendaraan tersebut digeser.
- Tidak ada biaya tambahan untuk gerakan tertentu (misalnya, menggeser kendaraan yang lebih besar).

Dalam implementasi Rush Hour pada tugas ini, setiap pergeseran kendaraan dianggap memiliki biaya yang sama (1), sehingga UCS berperilaku seperti BFS – mengeksplorasi semua state pada kedalaman d sebelum state pada kedalaman $d + 1$. Hal ini memberikan jaminan bahwa solusi yang ditemukan memiliki jumlah langkah minimum.

2.4 Efisiensi Algoritma A* dibandingkan dengan UCS pada Rush Hour

Secara teoritis, A* lebih efisien dibandingkan UCS pada penyelesaian Rush Hour karena:

- A* menggunakan informasi heuristik untuk mengarahkan pencarian menuju state yang lebih menjanjikan.
- UCS mengeksplorasi state berdasarkan jumlah langkah dari state awal, tanpa mempertimbangkan seberapa dekat sebuah state dengan tujuan.
- Dengan heuristik yang baik, A* akan memprioritaskan eksplorasi state yang lebih dekat dengan solusi.

Efisiensi A* sangat tergantung pada kualitas heuristik yang digunakan. Dalam Rush Hour, heuristik seperti Manhattan Distance dan jumlah kendaraan penghalang dapat memberikan informasi yang cukup baik untuk mengarahkan pencarian, sehingga A* biasanya mengeksplorasi node yang lebih sedikit dibanding UCS untuk mencapai solusi yang sama.

2.5 Optimalitas Algoritma Greedy Best First Search pada Rush Hour

Greedy Best First Search tidak menjamin solusi optimal untuk penyelesaian Rush Hour karena:

- Algoritma ini hanya mempertimbangkan nilai heuristik (seberapa dekat state dengan tujuan) tanpa mempertimbangkan berapa langkah yang sudah ditempuh.
- Greedy dapat terjebak pada local optima, di mana state tampak dekat dengan tujuan berdasarkan heuristik, tetapi sebenarnya memerlukan lebih banyak langkah untuk mencapai solusi.
- Dalam Rush Hour, Greedy mungkin memilih jalur yang tampaknya membawa kendaraan utama lebih dekat ke pintu keluar, tetapi mengabaikan strategi jangka panjang yang mungkin lebih efisien.

Misalnya, dalam konfigurasi di mana kendaraan utama dihalangi oleh beberapa kendaraan lain, Greedy mungkin memilih untuk menggeser kendaraan penghalang yang langsung berhadapan dengan kendaraan utama, meskipun ini mungkin menyebabkan kemacetan yang lebih kompleks di langkah berikutnya. UCS dan A* akan mempertimbangkan total biaya jalur, sehingga lebih mungkin menemukan solusi dengan jumlah langkah minimum.

3 Implementasi Program dan Tangkapan Layar

3.1 Implementasi Program

Implementasi program dilakukan dalam bahasa pemrograman C++ yang dioptimalkan untuk melakukan pencarian jalur menggunakan algoritma UCS, Greedy Best First Search, dan A*. Berikut adalah struktur utama program:

- **Node Representation:** Setiap state dari permainan (misal: posisi kendaraan pada Rush Hour) direpresentasikan sebagai node.
- **Graph Representation:** Relasi antara node disusun dalam bentuk graf berarah dengan bobot sesuai biaya pergerakan.
- **Priority Queue:** Untuk UCS dan A*, digunakan antrian prioritas yang mengurutkan node berdasarkan nilai $f(n)$ yang dihitung.
- **Heuristic Function:** Pada Greedy dan A*, fungsi heuristik disesuaikan dengan estimasi jarak minimum menuju solusi.

3.2 Struktur Kode Program

Struktur kode disusun sebagai berikut:

```
Tucil3_13523009
├── README.md
├── .gitignore
├── package.json
├── next.config.js
├── tailwind.config.js
├── postcss.config.mjs
├── bin
├── doc
├── test
├── public
└── src
    ├── app
    ├── components
    ├── lib
    └── utils.js
```

```
src
├── app ..... Aplikasi Next.js
│   ├── page.js ..... Home page (homepage)
│   ├── layout.js
│   ├── globals.css
│   ├── game ..... Game page
│   │   └── page.js
│   └── creator ..... Creator page
│       └── page.js
├── components ..... Reusable UI components
│   ├── Button.js
│   ├── Typography.js
│   ├── NavBar.js
│   └── RushHour ..... Komponen khusus game Rush Hour
│       ├── Board.js
│       ├── Piece.js
│       ├── Controls.js
│       ├── Stats.js
│       ├── FileInput.js
│       ├── PuzzleInput.js
│       └── index.js
└── lib ..... Core logic
    ├── models.js ..... Models (Board, Piece, GameState)
    ├── services ..... Fungsi utama game
    │   ├── index.js
    │   └── gameService.js
    ├── algorithms ..... Implementasi algoritma
    │   ├── index.js
    │   ├── Greedy.js
    │   ├── UCS.js
    │   └── AStar.js
    └── heuristics ..... Implementasi heuristik
        ├── index.js
        ├── ManhattanDistance.js
        ├── BlockingPieces.js
        └── CombinedHeuristic.js
```

3.3 Implementasi Algoritma Pencarian

Berikut adalah implementasi ketiga algoritma pencarian yang digunakan dalam program. Ketiga algoritma ini memiliki struktur yang serupa namun berbeda dalam cara mengevaluasi dan memprioritaskan state.

```
export function greedyBestFirstSearch(initialBoard, heuristicFn) {
  const initialState = new GameState(initialBoard);
  const queue = [initialState];
  const visited = new Set();
  let nodesVisited = 0;

  while (queue.length > 0) {
    // Sort queue by heuristic value (lowest first)
    queue.sort((a, b) => heuristicFn(a.board) - heuristicFn(b.board));

    const currentState = queue.shift();
    const stateHash = currentState.hash();
    nodesVisited++;

    if (visited.has(stateHash)) continue;
    visited.add(stateHash);

    if (currentState.isGoal()) {
      return { path: currentState.getPath(), nodesVisited, solved: true
          };
    }

    const nextStates = currentState.getNextStates();
    for (const nextState of nextStates) {
      if (!visited.has(nextState.hash())) {
        queue.push(nextState);
      }
    }
  }

  return { path: [], nodesVisited, solved: false };
}
```

Listing 1: Implementasi Greedy Best First Search

```
export function uniformCostSearch(initialBoard) {
  const initialState = new GameState(initialBoard);
  const queue = [initialState];
  const visited = new Set();
  let nodesVisited = 0;

  while (queue.length > 0) {
    // Sort queue by cost (lowest cost first)
    queue.sort((a, b) => a.cost - b.cost);

    const currentState = queue.shift();
    const stateHash = currentState.hash();
    nodesVisited++;

    if (visited.has(stateHash)) continue;
    visited.add(stateHash);

    if (currentState.isGoal()) {
      return { path: currentState.getPath(), nodesVisited, solved: true
          };
    }

    const nextStates = currentState.getNextStates();
    for (const nextState of nextStates) {
      if (!visited.has(nextState.hash())) {
        queue.push(nextState);
      }
    }
  }

  return { path: [], nodesVisited, solved: false };
}
```

Listing 2: Implementasi Uniform Cost Search (UCS)

```
export function aStarSearch(initialBoard, heuristicFn) {
  const initialState = new GameState(initialBoard);
  const openSet = [initialState];
  const closedSet = new Set();
  let nodesVisited = 0;

  while (openSet.length > 0) {
    // Sort by f(n) = g(n) + h(n)
    openSet.sort((a, b) => {
      const aScore = a.cost + heuristicFn(a.board);
      const bScore = b.cost + heuristicFn(b.board);
      return aScore - bScore;
    });

    const currentState = openSet.shift();
    const stateHash = currentState.hash();
    nodesVisited++;

    if (closedSet.has(stateHash)) continue;
    closedSet.add(stateHash);

    if (currentState.isGoal()) {
      return { path: currentState.getPath(), nodesVisited, solved: true
          };
    }

    const nextStates = currentState.getNextStates();
    for (const nextState of nextStates) {
      const nextHash = nextState.hash();
      if (closedSet.has(nextHash)) continue;

      if (!openSet.some((state) => state.hash() === nextHash)) {
        openSet.push(nextState);
      }
    }
  }

  return { path: [], nodesVisited, solved: false };
}
```

Listing 3: Implementasi A* Search Algorithm

3.4 Contoh Input dan Output

Berikut adalah beberapa contoh tangkapan layar hasil eksekusi program:

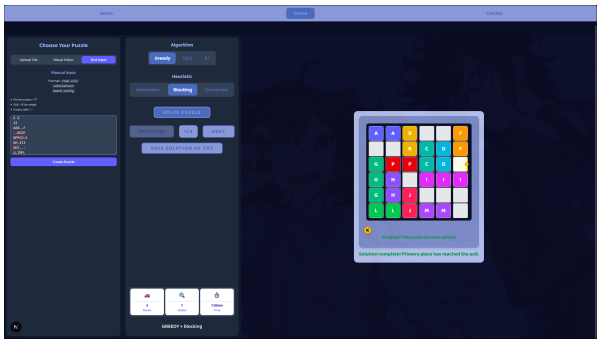

Figure 3.1: Tampilan awal eksekusi algoritma Greedy Best First Search dengan exit berada di kanan

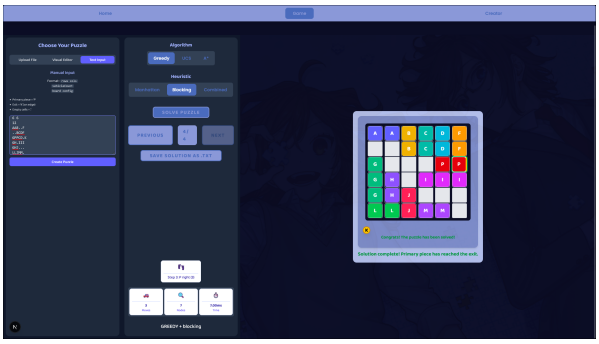

Figure 3.2: Tampilan akhir eksekusi algoritma Greedy Best First Search dengan exit berada di kanan

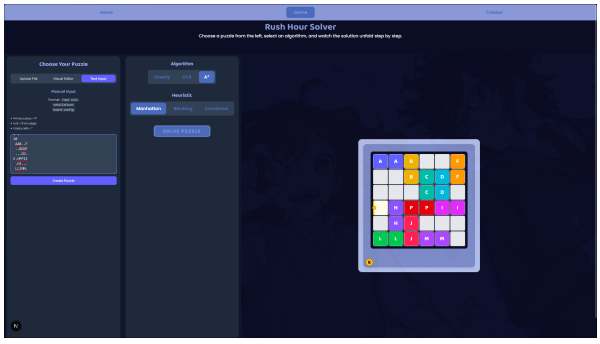

Figure 3.3: Tampilan awal eksekusi algoritma A* dengan exit berada di kiri

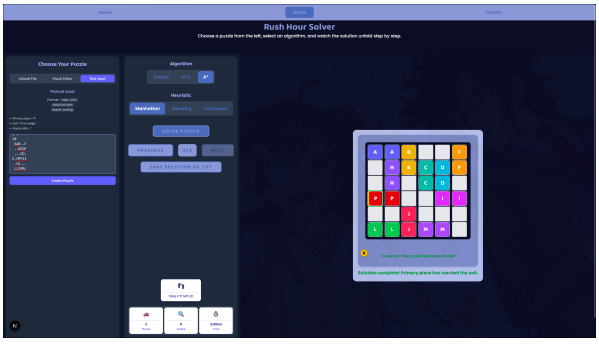

Figure 3.4: Tampilan akhir eksekusi algoritma A* dengan exit berada di kiri

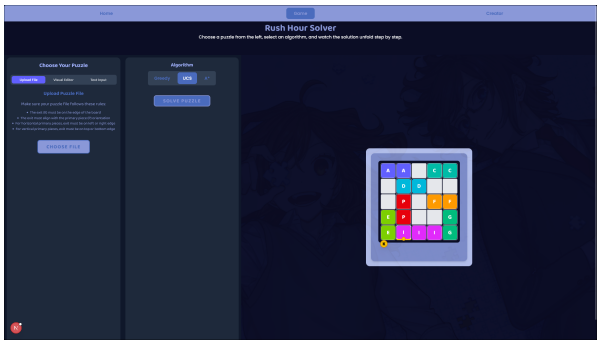

Figure 3.5: Tampilan awal eksekusi algoritma UCS dengan exit berada di bawah

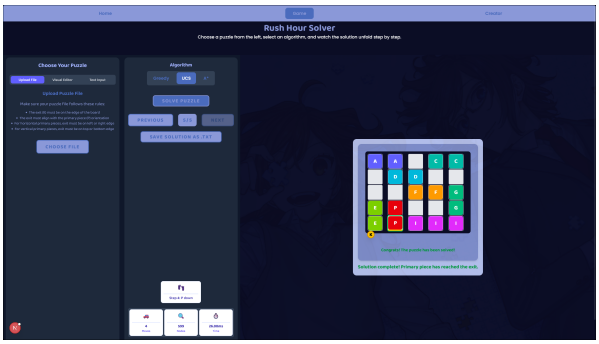

Figure 3.6: Tampilan akhir eksekusi algoritma UCS dengan exit berada di bawah

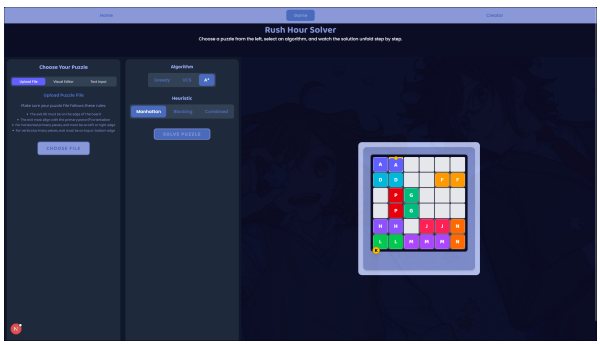

Figure 3.7: Tampilan awal eksekusi algoritma A* dengan exit berada di atas

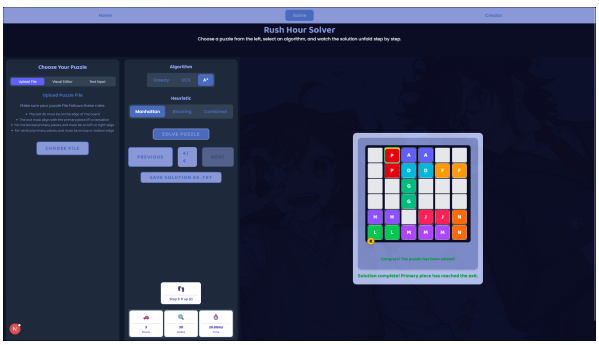

Figure 3.8: Tampilan akhir eksekusi algoritma A* dengan exit berada di atas

4 Hasil Analisis Percobaan Algoritma Pathfinding

4.1 Analisis Kompleksitas Algoritma

Berikut adalah analisis kompleksitas dari ketiga algoritma pathfinding yang diimplementasikan dalam penyelesaian puzzle Rush Hour:

- **Uniform Cost Search (UCS):** Kompleksitas waktu dan ruang adalah $O(b^{c/\epsilon})$, di mana b adalah branching factor (jumlah langkah yang mungkin pada setiap state), c adalah biaya dari solusi, dan ϵ adalah biaya langkah minimum. Dalam kasus terburuk, jika semua langkah memiliki biaya yang sama, kompleksitasnya menjadi $O(b^d)$ di mana d adalah kedalaman solusi optimal. Dalam konteks Rush Hour, branching factor dapat bervariasi berdasarkan posisi kendaraan dan kepadatan papan permainan, dan karena setiap langkah biasanya memiliki biaya yang sama, kompleksitas praktisnya menjadi $O(b^d)$. UCS menjamin solusi optimal karena mengeksplorasi node berdasarkan biaya terendah (jumlah langkah).

- **Greedy Best First Search:** Kompleksitas waktu dan ruang adalah $O(b^m)$, dengan m adalah kedalaman maksimum dalam graph. Greedy menggunakan heuristik sebagai panduan utama tanpa mempertimbangkan biaya actual path sejauh ini. Dalam implementasi Rush Hour, hal ini menyebabkan eksplorasi lebih cepat menuju arah yang menjanjikan, namun tidak menjamin solusi optimal.

- **A* Search:** Kompleksitas waktu dan ruang adalah $O(b^d)$, mirip dengan UCS, namun dengan heuristik yang admissible, A* lebih efisien karena fokus eksplorasinya terarah. A* menggabungkan pertimbangan biaya sejauh ini ($g(n)$) dan estimasi biaya ke tujuan ($h(n)$).

4.2 Perbandingan Performa Algoritma

Untuk menguji efektivitas algoritma, dilakukan percobaan menggunakan konfigurasi puzzle yang sama pada ketiga algoritma. Berikut adalah konfigurasi board yang digunakan:

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

Di mana "P" adalah primary piece (kendaraan utama) dan "K" adalah pintu keluar. Hasil percobaan pada konfigurasi ini ditampilkan dalam tabel berikut:

Algoritma	Jumlah Langkah	Node Dievaluasi	Waktu Eksekusi (ms)
UCS	51	11,631	347.00
Greedy	54	8,002	1,388.00
A*	51	2,819	757.00

Table 1: Perbandingan performa algoritma pada konfigurasi puzzle yang sama

4.3 Analisis Hasil Percobaan

Berdasarkan hasil percobaan pada konfigurasi puzzle yang sama, dapat diamati beberapa karakteristik penting dari ketiga algoritma:

1. **Optimalitas Solusi**:
    - UCS dan A* keduanya menemukan solusi optimal dengan 51 langkah.
    - Greedy menghasilkan solusi sub-optimal dengan 54 langkah, membuktikan bahwa Greedy tidak menjamin solusi optimal.
2. **Jumlah Node yang Dievaluasi**:
    - A* paling efisien, hanya mengevaluasi 2,819 node untuk menemukan solusi optimal.
    - Greedy mengevaluasi 8,002 node tetapi menghasilkan solusi sub-optimal.
    - UCS mengevaluasi paling banyak node (11,631) meskipun menemukan solusi optimal, menunjukkan eksplorasi yang lebih luas.
3. **Waktu Eksekusi**:
    - Secara mengejutkan, UCS memiliki waktu eksekusi tercepat (347 ms) meskipun mengevaluasi paling banyak node.
    - A* membutuhkan waktu menengah (757 ms) meskipun mengevaluasi paling sedikit node.
    - Greedy membutuhkan waktu terlama (1,388 ms) meskipun mengevaluasi lebih sedikit node daripada UCS.
4. **Efisiensi Algoritma**:
    - A* paling efisien dalam hal jumlah node yang dievaluasi, menunjukkan bahwa heuristik yang digunakan efektif dalam mengarahkan pencarian.
    - Meskipun UCS mengevaluasi lebih banyak node, waktu eksekusi per node lebih cepat, menunjukkan overhead yang lebih rendah dalam komputasi.
    - Greedy memiliki waktu komputasi per node yang paling tinggi, kemungkinan karena kompleksitas perhitungan heuristik tanpa manfaat optimalitas.

4.4 Analisis Kompleksitas Program

Selain kompleksitas algoritma, beberapa aspek implementasi yang mempengaruhi performa program:

- **Representasi Board**: Menggunakan grid 2D dengan notasi karakter untuk mewakili kendaraan memerlukan kompleksitas ruang $O(R \times C)$ di mana R dan C adalah dimensi papan.
- **Hash Function**: Implementasi hash function untuk mendeteksi state duplikat memiliki kompleksitas $O(R \times C)$ tetapi sangat penting untuk menghindari eksplorasi state berulang.

- **Move Generation**: Untuk setiap state, menghasilkan semua gerakan yang mungkin memiliki kompleksitas $O(P \times D)$ di mana P adalah jumlah kendaraan dan D adalah dimensi papan (karena setiap kendaraan dapat bergerak maksimal sepanjang papan).
- **Priority Queue Operations**: Operasi pada priority queue (insert, extract-min) dalam implementasi UCS dan A* memiliki kompleksitas $O(\log N)$ di mana N adalah jumlah node dalam queue.
- **Overhead Heuristik**: Perhitungan heuristik dalam Greedy dan A* menambahkan kompleksitas komputasi per node, yang dapat menjelaskan perbedaan waktu eksekusi per node.

```
// Contoh implementasi hash function untuk Board
string Board::hash() {
    return this->grid
      .map((row) => {
        if (!Array.isArray(row)) {
          console.error("Invalid row in board grid:", row);
          return String(row);
        }
        return row.join("");
      })
      .join("");
}
```

Listing 4: Implementasi hash function untuk mencegah eksplorasi state duplikat

4.5 Kesimpulan Analisis

Dari hasil percobaan pada konfigurasi puzzle yang sama, dapat disimpulkan bahwa:

- **Optimalitas vs. Efisiensi**: UCS dan A* menjamin solusi optimal, tetapi dengan trade-off yang berbeda. A* mengevaluasi jauh lebih sedikit node, sementara UCS memiliki overhead komputasi per node yang lebih rendah.
- **Anomali Waktu Eksekusi**: Meskipun Greedy mengevaluasi lebih sedikit node daripada UCS, waktu eksekusinya jauh lebih lama. Ini mungkin disebabkan oleh overhead komputasi heuristik atau perbedaan dalam strategi pengelolaan memori dan kompleksitas implementasi.
- **Keseimbangan Terbaik**: A* memberikan keseimbangan yang baik antara jumlah node yang dievaluasi dan optimalitas solusi, meskipun bukan yang tercepat dalam kasus ini. Untuk puzzle yang lebih kompleks, keuntungan dari evaluasi node yang lebih sedikit kemungkinan akan semakin signifikan.
- **Implikasi Praktis**: Untuk puzzle Rush Hour dengan kompleksitas sedang hingga tinggi, A* tetap menjadi pilihan yang direkomendasikan karena skalabilitasnya yang lebih baik, meskipun UCS menunjukkan performa yang mengejutkan baik dalam kasus uji ini.

Percobaan ini menegaskan keunggulan penggunaan heuristik yang tepat dalam mengurangi ruang pencarian, tetapi juga menunjukkan pentingnya efisiensi implementasi dalam menentukan performa algoritma secara keseluruhan.

5 Penjelasan mengenai Implementasi Bonus

5.1 Implementasi Heuristik Alternatif

Dalam proyek ini, selain heuristik dasar yang digunakan pada algoritma Greedy dan A*, dua heuristik alternatif telah diimplementasikan untuk meningkatkan efisiensi pencarian:

5.1.1 Manhattan Distance

Heuristik Manhattan Distance menghitung jarak antara primary piece dan pintu keluar berdasarkan jumlah langkah horizontal atau vertikal yang diperlukan.

```
export function manhattanDistance(board) {
  const primaryPiece = board.getPrimaryPiece();
  if (!primaryPiece) return Infinity;

  // Get the exit position and direction
  const [exitRow, exitCol] = board.exitPosition;
  const exitDirection = board.exitDirection;

  // Find the closest position of the primary piece to consider
  let relevantPosition = null;

  // Choose the correct position based on exit direction
  if (exitDirection === "up") {
    // For top exit, we care about the top-most position
    relevantPosition = primaryPiece.positions.reduce(
      (min, pos) => (pos[0] < min[0] ? pos : min),
      primaryPiece.positions[0]
    );
  } else if (exitDirection === "down") {
    // For bottom exit, we care about the bottom-most position
    relevantPosition = primaryPiece.positions.reduce(
      (max, pos) => (pos[0] > max[0] ? pos : max),
      primaryPiece.positions[0]
    );
  }
  // Similar logic for left and right exits...

  // Calculate distance to exit based on direction
  let distance = 0;

  if (exitDirection === "up") {
    // Top exit - measure vertical distance to top edge
    distance = relevantPosition[0]; // Distance to top edge (row 0)
  } else if (exitDirection === "down") {
    // Bottom exit - measure vertical distance to bottom edge
    distance = board.size[0] - 1 - relevantPosition[0];
  }
  // Similar calculations for left and right exits...

  return distance;
}
```

Listing 5: Implementasi heuristik Manhattan Distance

5.1.2 Blocking Pieces

Heuristik Blocking Pieces menghitung jumlah kendaraan yang menghalangi jalur langsung dari primary piece menuju pintu keluar.

```
export function blockingPieces(board) {
  const primaryPiece = board.getPrimaryPiece();
  if (!primaryPiece) return Infinity;

  const exitPosition = board.exitPosition;
  const exitDirection = board.exitDirection;
  const grid = board.grid;

  // Count blocking pieces
  let blockingCount = 0;
  const blockers = new Set();

  // Find relevant position of primary piece based on exit direction
  let relevantRow = 0;
  let relevantCol = 0;

  // Logic to determine the relevant position based on exit direction
  // ...

  // Count blocking pieces based on exit direction
  if (exitDirection === "up") {
    // Count pieces from primary to top edge
    for (let row = relevantRow - 1; row >= 0; row--) {
      const cell = grid[row][relevantCol];
      if (cell !== "." && cell !== "P" && !blockers.has(cell)) {
        blockingCount++;
        blockers.add(cell);
      }
    }
  }
  // Similar logic for other directions...

  // Multiply by 2 to give more weight to blocking pieces
  return blockingCount * 2;
}
```

Listing 6: Implementasi heuristik Blocking Pieces

5.1.3 Combined Heuristic

Selain itu, implementasi juga menyediakan heuristik gabungan yang mengkombinasikan Manhattan Distance dan Blocking Pieces untuk estimasi yang lebih akurat.

```
export function combinedHeuristic(board) {
  const distance = manhattanDistance(board);
  const blocking = blockingPieces(board);

  // Combine both heuristics with weights
  return distance + blocking;
}
```

Listing 7: Implementasi heuristik Combined

5.2 Implementasi Graphical User Interface

Proyek ini dilengkapi dengan antarmuka pengguna grafis (GUI) yang memudahkan interaksi dengan puzzle Rush Hour dan visualisasi solusi.

5.2.1 Komponen Utama GUI

- **Interactive Board**: Papan permainan yang menampilkan posisi kendaraan dan pergerakan animasi.
- **Algorithm Controls**: Pemilihan algoritma dan heuristik yang akan digunakan.
- **Input Methods**: Berbagai metode untuk memasukkan konfigurasi puzzle (file upload, visual editor, manual input).
- **Solution Controls**: Antarmuka untuk navigasi langkah-langkah solusi dengan animasi.
- **Performance Statistics**: Visualisasi statistik kinerja algoritma (node yang dikunjungi, waktu eksekusi).

```
export default function Board({
  size,
  configuration,
  primaryPiece,
  exit,
  exitDirection: propExitDirection,
  moves,
  currentMove,
  completedState,
}) {
  // State for board, exit position and direction
  const [board, setBoard] = useState([]);
  const [exitPosition, setExitPosition] = useState(null);
  const [exitDirection, setExitDirection] = useState(null);

  useEffect(() => {
    // Initialize board based on configuration
    const newBoard = [];
    for (let i = 0; i < size[0]; i++) {
      const row = [];
      for (let j = 0; j < size[1]; j++) {
        row.push(configuration[i][j] || ".");
      }
      newBoard.push(row);
    }
    setBoard(newBoard);

    // Set exit position and direction
    // ...
  }, [configuration, size, exit, propExitDirection]);

  // Get current move being animated
  const getCurrentMoveInfo = () => {
    if (!moves || currentMove === undefined ||
        currentMove < 0 || currentMove >= moves.length) {
      return null;
    }
    return moves[currentMove];
  };

  // Render board with pieces and animations
  return (
    <div className="bg-secondary...">
      <div className="relative">
        <div
          className="grid..."
          style={{
            gridTemplateRows: `repeat(${size[0]}, minmax(0, 1fr))`,
            gridTemplateColumns: `repeat(${size[1]}, minmax(0, 1fr))`,
          }}
        >
          {/* Grid cells rendering */}
          {/* ... */}
        </div>

        {/* Exit marker */}
        {/* ... */}
      </div>
    </div>
  );
}
```


Listing 8: Implementasi komponen Board untuk papan permainan interaktif

5.2.2 Fitur Interaktif dan Visualisasi

GUI diimplementasikan dengan Next.js dan React, memungkinkan fitur interaktif berikut:

- **Animasi Pergerakan**: Menggunakan Framer Motion untuk animasi pergerakan kendaraan yang smooth.
- **Highlighting**: Penanda visual untuk primary piece, pintu keluar, dan kendaraan yang sedang bergerak.
- **Step-by-Step Navigation**: Kontrol untuk menelusuri solusi langkah demi langkah dengan tombol Previous dan Next.
- **Export Solution**: Kemampuan untuk menyimpan solusi dalam format text file sesuai spesifikasi tugas.
- **Real-time Statistics**: Tampilan statistik performa yang diperbarui secara real-time.

```
export default function Piece({
  id,
  isPrimary,
  isMoving,
  moveDistance = 1,
  moveDirection = null,
}) {
  // Generate consistent color for each piece
  const generateColor = (id) => {
    const colors = [
      "bg-blue-500", "bg-green-500", "bg-purple-500",
      // ... more colors
    ];

    // Use character code to select a color
    const charCode = id.charCodeAt(0) % colors.length;
    return colors[charCode];
  };

  // Calculate animation based on direction and distance
  const getAnimationProps = () => {
    if (!isMoving || !moveDirection || moveDistance <= 0) {
      return {};
    }

    // Calculate scaled distance
    const distance = moveDistance * 48; // 48px per cell

    // Define movement animation based on direction
    let x = 0, y = 0;
    switch (moveDirection) {
      case "up": y = -distance; break;
      case "down": y = distance; break;
      case "left": x = -distance; break;
      case "right": x = distance; break;
    }

    // Create a smooth sliding animation
    return {
      x, y,
      transition: {
        duration: Math.min(0.3 + moveDistance * 0.1, 0.8),
        ease: "easeInOut",
      },
    };
  };
  return
  ...
```

Listing 9: Implementasi komponen Piece dengan animasi menggunakan Framer Motion

5.2.3 Visual Editor untuk Konfigurasi Puzzle

Selain input file, implementasi GUI juga menyediakan visual editor untuk membuat dan mengedit konfigurasi puzzle secara interaktif:

- **Drag-and-Drop Interface**: Antarmuka untuk placement kendaraan.
- **Rotation Tool**: Kemampuan untuk memutar kendaraan (horizontal/vertikal).
- **Exit Placement**: Penempatan pintu keluar yang fleksibel dengan validasi otomatis.
- **Board Size Adjustment**: Pengaturan ukuran papan yang fleksibel.

Lampiran

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	✔	

Table 2: Checklist Penilaian

Github Repository

https://github.com/SayyakuHajime/Tucil3_13523009