Tugas Kecil 3
Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding
IF2211 Strategi Algoritma

Disusun Oleh:
Irgiansyah Mondo - 13521167

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2024/2025

A. ALGORITMA

1. Uniform Cost Search

Algoritma Uniform Cost Search (UCS) merupakan algoritma pencarian yang memperluas simpul dengan biaya kumulatif terendah terlebih dahulu. Dalam konteks permainan Rush Hour, algoritma UCS mencari solusi optimal dengan menjelajahi semua kemungkinan pergerakan kendaraan berdasarkan biaya pergerakan.

UCS menggunakan struktur data priority queue untuk menyimpan simpul-simpul yang akan dijelajahi, dengan prioritas berdasarkan biaya jalur dari simpul awal hingga simpul saat ini. Simpul dengan biaya terendah akan dieksplorasi terlebih dahulu.

Implementasi UCS dalam permainan Rush Hour dapat dilihat pada fungsi solveWithUCS():

```
export const solveWithUCS = (initialState) => {
  const frontier = [{ state: initialState, cost: 0 }];

  const explored = new Set();

  const cameFrom = new Map();

  let nodesVisited = 0;

  while (frontier.length > 0) {
    frontier.sort((a, b) => a.cost - b.cost);
    const { state, cost } = frontier.shift();
    nodesVisited++;

    if (isSolved(state)) {
      console.log("Solution found:", state);
      const path = reconstructPath(cameFrom, state);
      console.log("Path found with length:", path.length);
      console.log("First step in path:", path[0]);
      console.log("Last step in path:", path[path.length - 1]);
      return {
        path: path,
```

```
      nodesVisited
    };
  }

  const stateStr = boardToString(state.board);

  if (explored.has(stateStr)) continue;

  explored.add(stateStr);

  const possibleMoves = getPossibleMoves(state);

  for (const nextState of possibleMoves) {
    const nextStateStr = boardToString(nextState.board);

    if (!explored.has(nextStateStr)) {
      frontier.push({ state: nextState, cost: cost + 1 });

      cameFrom.set(nextStateStr, state);
    }
  }
 }

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

Pada algoritma UCS ini, array frontier berfungsi sebagai priority queue yang menampung semua state yang belum dieksplorasi. Setiap iterasi, frontier diurutkan berdasarkan biaya kumulatif (cost) sehingga state dengan biaya terendah akan diproses terlebih dahulu melalui frontier.shift(). Setiap pergerakan kendaraan dianggap memiliki biaya yang sama yaitu 1, sehingga biaya baru adalah biaya sebelumnya ditambah 1.

UCS menggunakan Set explored untuk melacak state yang sudah dikunjungi, mencegah pengulangan eksplorasi state yang sama. Map cameFrom menyimpan hubungan antara state dan parent-nya, yang akan digunakan untuk merekonstruksi jalur solusi. Algoritma ini menjamin menemukan solusi dengan jumlah langkah minimal (optimal).

2. Greedy Best-First Search

Algoritma Greedy Best-First Search adalah algoritma pencarian informed yang menggunakan fungsi heuristik untuk menentukan simpul mana yang akan dijelajahi selanjutnya. Tidak seperti UCS yang mempertimbangkan biaya perjalanan, Greedy Best-First Search hanya melihat nilai heuristik yang memperkirakan jarak ke tujuan.Pada permainan Rush Hour, heuristik yang digunakan adalah jumlah kendaraan yang menghalangi jalur keluar kendaraan target dan jarak kendaraan target dari posisi keluar. Berikut implementasi Greedy Best-First Search:

```
export const solveWithGreedy = (initialState) => {
  const heuristic = (state) => {
    // Hitung jumlah kendaraan yang menghalangi jalur keluar
    let blockers = 0;
    const { board, target } = state;
    const targetCar = board.find(car => car.id === target);

    // Hitung jarak target ke posisi keluar
    const distanceToExit = board[0].length - (targetCar.x + targetCar.length);

    // Hitung jumlah penghalang
    for (const car of board) {
      if (car.id !== target && car.orientation === 'horizontal' &&
          car.y === targetCar.y && car.x > targetCar.x) {
        blockers++;
      }
    }

    return blockers + distanceToExit;
  };

  const frontier = [{ state: initialState, hValue: heuristic(initialState) }];

  const explored = new Set();

  const cameFrom = new Map();

  let nodesVisited = 0;

  while (frontier.length > 0) {
    frontier.sort((a, b) => a.hValue - b.hValue);
    const { state, hValue } = frontier.shift();
```

```
    nodesVisited++;

    if (isSolved(state)) {
      const path = reconstructPath(cameFrom, state);
      return {
        path: path,
        nodesVisited
      };
    }

    const stateStr = boardToString(state.board);

    if (explored.has(stateStr)) continue;

    explored.add(stateStr);

    const possibleMoves = getPossibleMoves(state);

    for (const nextState of possibleMoves) {
      const nextStateStr = boardToString(nextState.board);

      if (!explored.has(nextStateStr)) {
        frontier.push({ state: nextState, hValue: heuristic(nextState) });

        cameFrom.set(nextStateStr, state);
      }
    }
  }

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

Dalam implementasi ini, fungsi heuristic() menghitung nilai heuristik berdasarkan jumlah kendaraan yang menghalangi jalur keluar kendaraan target dan jarak kendaraan target dari posisi keluar. Frontier diurutkan berdasarkan nilai heuristik, sehingga state dengan nilai heuristik terkecil (yang terlihat lebih "dekat" dengan solusi) akan diproses terlebih dahulu. Greedy Best-First Search tidak menjamin solusi optimal, tetapi biasanya lebih cepat daripada UCS karena fokus pada simpul yang tampak lebih dekat ke tujuan berdasarkan heuristik.

3. A* Search

Algoritma A* menggabungkan keuntungan dari UCS dan Greedy Best-First Search. A* menggunakan fungsi evaluasi f(n) = g(n) + h(n), di mana g(n) adalah biaya dari simpul awal ke simpul n, dan h(n) adalah estimasi heuristik dari simpul n ke tujuan. Implementasi A* dalam permainan Rush Hour:

```
export const solveWithAStar = (initialState) => {
  const heuristic = (state) => {
    // Hitung jumlah kendaraan yang menghalangi jalur keluar
    let blockers = 0;
    const { board, target } = state;
    const targetCar = board.find(car => car.id === target);

    // Hitung jarak target ke posisi keluar
    const distanceToExit = board[0].length - (targetCar.x + targetCar.length);

    // Hitung jumlah penghalang
    for (const car of board) {
      if (car.id !== target && car.orientation === 'horizontal' &&
          car.y === targetCar.y && car.x > targetCar.x) {
        blockers++;
      }
    }

    return blockers + distanceToExit;
  };

  const frontier = [{
    state: initialState,
    cost: 0,
    hValue: heuristic(initialState),
    fValue: heuristic(initialState) // f(n) = g(n) + h(n), g(n) = 0 untuk initial state
  }];

  const explored = new Set();

  const cameFrom = new Map();
  const gScore = new Map();

  gScore.set(boardToString(initialState.board), 0);

  let nodesVisited = 0;

  while (frontier.length > 0) {
```

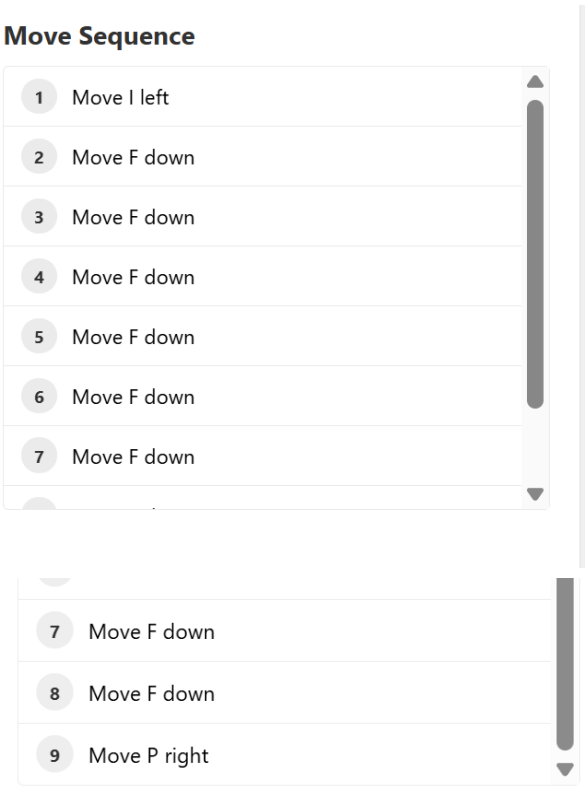
```
    frontier.sort((a, b) => a.fValue - b.fValue);
    const { state, cost } = frontier.shift();
    nodesVisited++;

    if (isSolved(state)) {
      const path = reconstructPath(cameFrom, state);
      return {
        path: path,
        nodesVisited
      };
    }

    const stateStr = boardToString(state.board);

    if (explored.has(stateStr)) continue;

    explored.add(stateStr);

    const possibleMoves = getPossibleMoves(state);

    for (const nextState of possibleMoves) {
      const nextStateStr = boardToString(nextState.board);
      const tentativeGScore = (gScore.get(stateStr) || 0) + 1;

      if (!gScore.has(nextStateStr) || tentativeGScore < gScore.get(nextStateStr)) {
        cameFrom.set(nextStateStr, state);
        gScore.set(nextStateStr, tentativeGScore);

        const h = heuristic(nextState);
        const f = tentativeGScore + h;

        frontier.push({
          state: nextState,
          cost: tentativeGScore,
          hValue: h,
          fValue: f
        });
      }
    }
  }

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

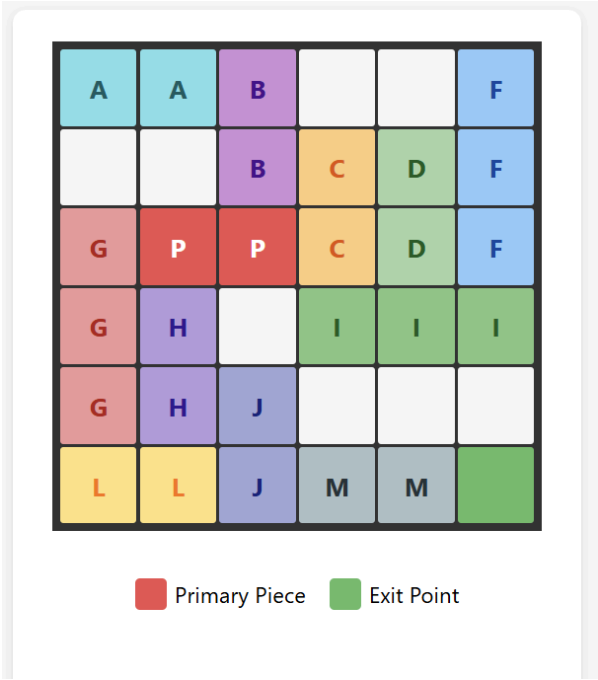
Dalam implementasi A*, setiap state dalam frontier memiliki tiga nilai: cost (g-value), hValue (h-value), dan fValue (f-value = g + h). Frontier diurutkan berdasarkan f-value, sehingga state dengan total biaya + heuristik terendah akan diproses terlebih dahulu.

Map gScore menyimpan biaya terbaik yang diketahui dari initial state ke setiap state yang ditemui. Jika ditemukan jalur yang lebih baik ke state yang sudah ada di frontier, maka informasi state tersebut akan diperbarui. Jika heuristik yang digunakan adalah admissible (tidak pernah overestimasi jarak ke tujuan), A* menjamin menemukan solusi optimal seperti UCS, tetapi biasanya lebih efisien karena menggunakan heuristik untuk mengarahkan pencarian.

B. PENGUJIAN

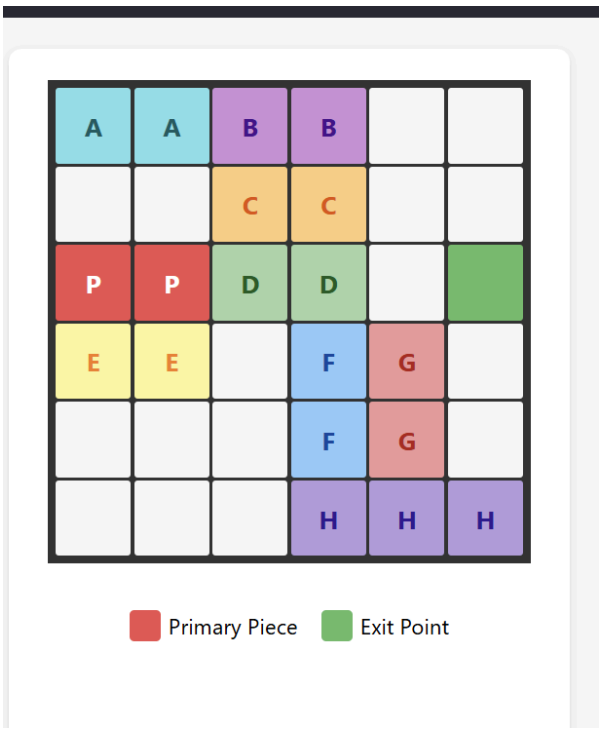
1. Test Case 1

Input	Ouput
6 6 11 AAB..F ..BCDF GPPCDF GH.III GHJ... LLJMM.K	

2. **Test Case 2**

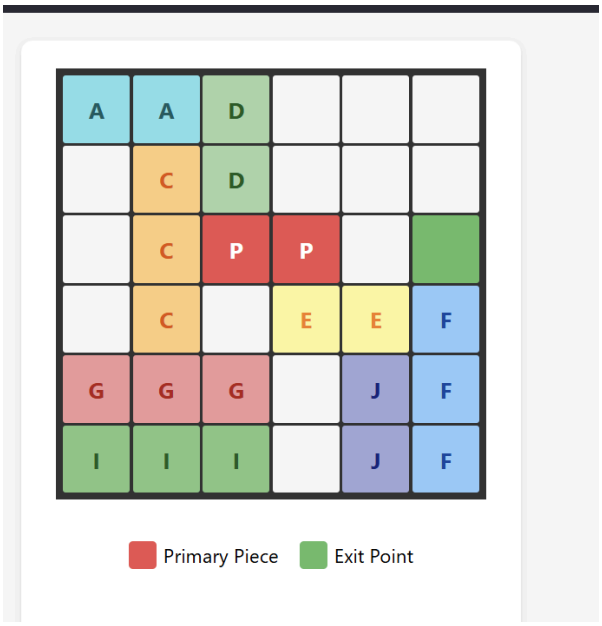
Input	Ouput
6 6 9 AABB.. ..CC.. PPDD..K EE.FG. ...FG. ...HHH	**Execution Time:** 576.20 ms **Nodes Visited:** 7630 **Heuristic Used:** Distance to Exit

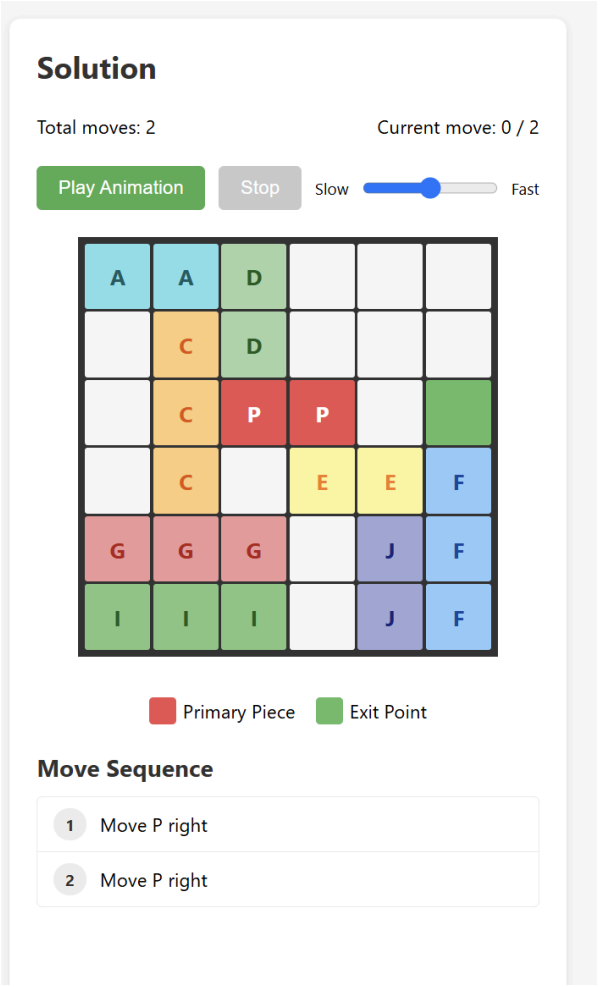
	

3. Test Case 3

Input	Ouput

6 6
9
AAD...
.CD...
.CPP..K
.C.EEF
GGG.JF
III.JF





C. LAMPIRAN

Link repository : https://github.com/irgimondo/Tucil3_13521167.git

-

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	✅	