

Laporan Tugas Kecil 3
IF2211 Strategi Algoritma

Penyelesaian Puzzle Rush Hour Menggunakan
Algoritma Pathfinding



Disusun oleh:
Aryo Wisanggeni (13523100)

Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2025

Daftar Isi

I. Pendahuluan	4
II. Penjelasan dan Analisis Algoritma	5
A. Algoritma Uniform Cost Search (UCS)	5
B. Algoritma Greedy Best First Search (GBFS)	5
C. Algoritma A Star (A*)	6
D. Algoritma Iterative Deepening A Star (IDA*)	7
III. Source Code	8
A. Struktur File Program	8
B. Program Utama	9
C. Struktur Data	10
D. Fungsi Pembantu Algoritma	12
E. Fungsi Algoritma	14
IV. Pengujian (Input dan Output)	15
V. Hasil Analisis Percobaan	30
A. Algoritma Uniform Cost Search	30
B. Algoritma Greedy Best First Search	30
C. Algoritma A*	31
D. Algoritma IDA*	31
VI. Implementasi Bonus	32
A. Algoritma Alternatif	32
B. Heuristik Alternatif	32
C. GUI	33
VII. Lampiran	35

I. Pendahuluan

Tugas kecil 3 Strategi Algoritma 2025 berupa proyek pembuatan perangkat lunak berbasis GUI (bonus) yang bertujuan untuk menerapkan algoritma pathfinding dalam menyelesaikan masalah. Khususnya, masalah yang ditinjau adalah penyelesaian puzzle Rush Hour. 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.

Dalam penyelesaian game ini, algoritma pathfinding merupakan cara yang efektif untuk menyelesaikannya, ditambah dengan algoritma yang menggunakan heuristic, seperti Greedy Best First Search dan A*, penyelesaian puzzle ini menjadi lebih menarik dengan percobaan berbagai heuristic. Dari itu, implementasi penyelesaian puzzle ini sangat variatif dan hasil paling optimal pun dapat diperoleh dengan cara langsung atau cara yang lebih boros bergantung pada heuristic yang dipilih untuk algoritma pathfinding yang tidak blind.

II. Penjelasan dan Analisis Algoritma

A. Algoritma Uniform Cost Search (UCS)

Algoritma Uniform Cost Search merupakan algoritma pathfinding uninformed atau blind search. Alasan dari ini adalah UCS hanya menggunakan informasi seberapa jauh saat ini yang telah ditempuh sebagai penentu jalan yang akan ditempuh berikutnya, bisa disebut algoritma ini adalah BFS dengan pembobotan atau prioritas lebih ke jalur yang jaraknya terendah dari awal pencarian. Fungsi jalur yang mengukur jaraknya dari status awal yang digunakan UCS biasa disebut $g(n)$.

Dalam implementasinya di permasalahan puzzle Rush Hour, jarak antara satu kondisi papan dengan kondisi papan setelah suatu mobil bergerak biasa dihitung sebagai satu gerakan. Walaupun satu pergerakan ke sel lain juga disebut sebagai satu gerakan, algoritma UCS masih saja mencari secara buta dengan mengeksplorasi simpul dengan $g(n)$ terendah dan selalu merupakan simpul dari kedalaman tertentu saja sebelum mencari mendalam. Karena $g(n)$ yang seragam tersebut, pada puzzle ini, efektif UCS adalah sama dengan algoritma BFS. Dari itu, karakteristik BFS juga akan ada pada UCS ini. Sebagai contoh, UCS di puzzle ini akan selalu sukses menemukan solusi paling optimal (paling sedikit langkahnya). Namun, sama seperti BFS, UCS akan mengunjungi banyak simpul lain sebelum mendapatkan solusi paling optimal tersebut. Dari itu, secara teoritis, UCS akan sangat baik untuk mencari solusi, namun memiliki kelemahan akan menggunakan banyak memori dalam pemrosesannya.

Selain itu, berikut merupakan prosedur dari UCS:

1. Inisiasi dengan simpul-simpul yang terhubung langsung dengan kondisi awal
2. Hitung $g(n)$ dari simpul-simpul tersebut dan pilih simpul dengan nilai $g(n)$ terendah untuk dieksplorasi berikut
3. Simpul yang dieksplorasi akan menambahkan simpul-simpul yang langsung terhubung dengannya (selain simpul asal) dan proses mengulang ke (2)
4. Proses berhenti ketika tujuan tercapai atau semua simpul telah dieksplorasi

B. Algoritma Greedy Best First Search (GBFS)

Algoritma Greedy Best First Search (GBFS) adalah algoritma pencarian yang termasuk ke dalam kategori informed search. Hal ini karena GBFS menggunakan informasi tambahan berupa heuristic untuk memandu pencariannya. GBFS memilih simpul berdasarkan nilai heuristic $h(n)$, yang merepresentasikan estimasi jarak dari simpul tersebut ke tujuan. Tujuan utama GBFS adalah mencari secepat mungkin ke arah goal, tanpa mempertimbangkan seberapa jauh simpul tersebut dari awal, sehingga tidak menghitung $g(n)$, tidak seperti UCS.

Dalam konteks puzzle Rush Hour, nilai $h(n)$ bisa berupa jumlah mobil yang menghalangi mobil utama mencapai tujuan, atau variasi lain seperti jumlah langkah minimal untuk memindahkan penghalang. GBFS akan memilih simpul yang paling "menjanjikan" mencapai goal berdasarkan nilai $h(n)$ terendah. Dari itu sendiri, GBFS memiliki kelebihan dari UCS karena menggunakan heuristik yang memandunya ke arah tujuan. Ini pasti akan lebih cepat dari UCS yang memeriksa setiap simpul secara buta.

Namun, karena tidak mempertimbangkan $g(n)$, algoritma ini tidak menjamin solusi optimal. GBFS bisa saja memilih jalur cepat ke depan yang ternyata berujung pada jalan buntu atau panjang. Dalam kasus Rush Hour, GBFS sangat cepat dalam menemukan solusi, tetapi seringkali solusi tersebut tidak minimum dalam jumlah langkah. Keoptimalan algoritma ini sangat bergantung pada heuristik yang digunakan dan kasus puzzle yang sesuai dengan heuristik. Namun demikian, lagi-lagi heuristik memiliki range sangat luas. Karena itu, GBFS tidak akan menjamin solusi yang optimal setiap kali, karena heuristik berbeda akan punya kelebihan masing-masing yang perlu dipikirkan dalam mengimplementasi GBFS. Jadi, kesimpulannya GBFS dapat memberi solusi optimal jika heuristik yang digunakan sangat bagus dan sesuai dengan masalah yang dihadapi secara teori.

Prosedur GBFS adalah:

1. Inisiasi dengan simpul awal yang langsung terhubung dengan kondisi awal dan hitung $h(n)$ untuk setiap simpul tersebut.
2. Pilih simpul dengan nilai $h(n)$ terendah untuk dieksplorasi.
3. Tambahkan tetangga dari simpul tersebut dan hitung nilai $h(n)$ mereka
4. Ulangi proses dari (2) hingga mencapai tujuan atau semua simpul telah dikunjungi.

C. Algoritma A Star (A^*)

A^* adalah algoritma pencarian *informed* yang mengkombinasikan kelebihan dari UCS dan GBFS. A^* menggunakan fungsi evaluasi $f(n) = g(n) + h(n)$, dengan $g(n)$ sebagai fungsi biaya dari simpul awal ke simpul saat ini (sama seperti UCS), dan $h(n)$ sebagai estimasi biaya dari simpul saat ini ke tujuan (sama seperti GBFS yang berbasis heuristik).

Dengan menggabungkan kedua komponen ini, A^* mampu menyeimbangkan antara eksplorasi dan eksploitasi. Dalam puzzle Rush Hour, $g(n)$ umumnya adalah jumlah langkah dari kondisi awal, sedangkan $h(n)$ bisa berupa jumlah mobil yang menghalangi atau pendekatan estimasi lainnya, seperti yang sudah dijelaskan pada bagian GBFS. A^* dikenal sebagai algoritma optimally efficient untuk semua algoritma dengan heuristik admissible. Heuristik disebut admissible jika tidak pernah melebihi biaya aktual dari simpul ke tujuan. Selain itu, A^* juga pasti akan lebih efisien dari UCS karena menggunakan heuristik yang memandunya ke arah tujuan dengan lebih cepat, namun memang terdapat juga heuristik yang bisa sabotase A^* sehingga bisa

lebih lambat dari UCS. Sehingga dari ini, jika heuristik memang bertujuan untuk memandu ke tujuan, pasti A* lebih cepat, namun jika tidak akan ada kemungkinan lebih lambat dari UCS.

Pada implementasi Rush Hour, jika heuristik memenuhi kriteria ini (contohnya hanya menghitung jumlah blok penghalang tanpa prediksi pergerakan lebih), A* akan selalu menghasilkan solusi optimal. Prosedur A*:

1. Inisiasi simpul awal dengan simpul yang terhubung langsung dengan kondisi awal, hitung $f(n) = g(n) + h(n)$.
2. Pilih simpul dengan nilai $f(n)$ terkecil untuk dieksplorasi.
3. Tambahkan tetangga dan hitung $g(n)$, $h(n)$, $f(n)$.
4. Ulangi (2) hingga mencapai tujuan atau semua simpul telah dieksplorasi.

D. Algoritma Iterative Deepening A Star (IDA*)

IDA* adalah varian dari A* yang menggabungkan kelebihan A* dalam menggunakan heuristik dengan keunggulan DFS dalam efisiensi memori. IDA* melakukan pencarian secara depth-first namun dengan batas $f(n)$ yang meningkat secara iteratif. Pada setiap iterasi, IDA* menggunakan ambang batas threshold untuk $f(n) = g(n) + h(n)$, sama persis dengan A*. Jika dalam proses DFS ditemukan simpul dengan $f(n) > \text{threshold}(n)$, simpul tersebut tidak dilanjutkan dan dicatat nilai minimum $f(n)$ berikutnya sebagai ambang untuk iterasi selanjutnya.

Dalam konteks Rush Hour, IDA* cocok untuk sistem dengan keterbatasan memori, namun secara performa waktu bisa lebih lambat dari A* karena sifat iteratif dan rekursifnya yang dapat mengulang pengecekan simpul yang sama di tiap iterasi. Prosedur IDA*:

1. Inisialisasi threshold awal sebagai $f(\text{start})$.
2. Lakukan DFS dengan batas $f(n) \leq \text{threshold}$.
3. Jika tujuan tidak ditemukan, tingkatkan threshold ke nilai minimum $f(n)$ yang melebihi threshold sebelumnya.
4. Ulangi hingga ditemukan solusi atau semua simpul telah dilewatkan.

III. Source Code

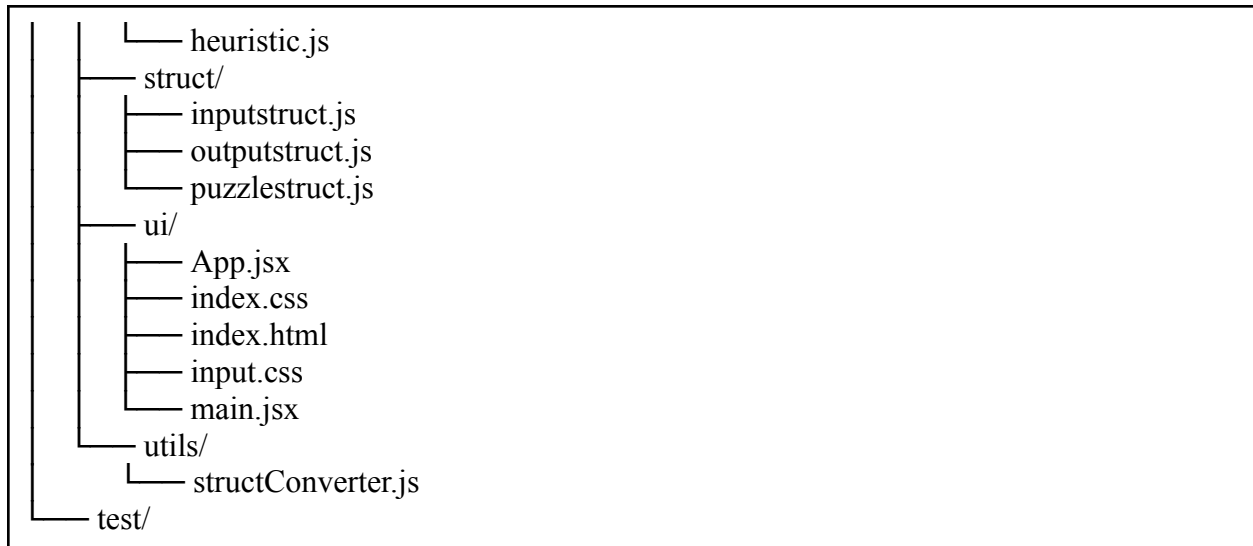
A. Struktur File Program

Secara garis besar, terdapat lima direktori: bin, dev, doc, src, dan test. Bin merupakan direktori yang mengandung file executable untuk menjalankan program (namun pada tugas ini kosong karena menggunakan javascript yang tidak perlu di-compile). Dev merupakan direktori yang berisi tes modular yang digunakan dalam pengembangan program ini. Doc merupakan direktori untuk menyimpan laporan tugas ini. Src merupakan direktori yang menyimpan kode sumber program. Terakhir, test merupakan direktori yang menyimpan test case yang digunakan pada bab IV.

Pada program ini, main.js dalam src berperan sebagai main driver program yang memanggil modul-modul lain, sedangkan preload.js merupakan penghubung frontend dan backend dari program ini.

Selain dua program itu, terdapat enam direktori lain dalam src yang membantu proses program ini. Pertama adalah algorithms yang merupakan direktori yang berisi algoritma-algoritma yang digunakan untuk mencari solusi dari puzzle. Kedua adalah io yang berisi program untuk menerima masukan file dan mengeluarkan luaran file berekstensi .txt. Ketiga, solver berisi fungsi-fungsi pembantu untuk mengimplementasi algoritma yang digunakan. Keempat, direktori struct berisi struktur-struktur data yang berbentuk class-class objek untuk menampung data yang digunakan selama jalannya program. Kelima, utils berisi satu fungsi untuk mengubah struktur dari input ke bentuk yang dipakai pada pemrosesan puzzle. Terakhir, direktori ui berisi file-file yang konfigurasi frontend dari program ini menggunakan vite, react, dan tailwind. Berikut merupakan struktur lengkap dari program.





B. Program Utama

Program utama berupa main.js yang ada pada src. Fungsinya untuk memberi API yang merupakan gabungan dari berbagai komponen backend untuk dipanggil untuk kepentingan frontend. Penghubung main.js dengan frontend di direktori ui adalah preload.js yang ada seperti berikut.

main.js

```
// VARIABEL CACHE
let cachedInput = null;
let cachedFilePath = null;
let cachedOutput = null;

// FUNGSI
/**
 * Creates the main application window for the Rush Hour solver.
 * Loads the frontend from the Vite build output ('dist/index.html')
 * and delays showing the window until it's fully ready.
 */
function createWindow()

/**
 * Initializes the Electron app once it's ready and sets up the main window.
 */
app.whenReady().then(createWindow);

/**
 * Handles the 'open-input' IPC event from the renderer.
 * Opens a file picker dialog, parses and validates the selected text file,
```



```

* and caches the parsed input structure for future use.
*
* @returns {Promise<Object|null>} Parsed input data with file name, or null if canceled.
*/
ipcMain.handle('open-input', async ())

/**
 * Handles the 'run-solver' IPC event from the renderer.
 * Executes the selected pathfinding algorithm using the cached input.
 * Returns structured output including timing, moves, and step-by-step states.
 *
 * @param {Electron.IpcMainInvokeEvent} event - The IPC event.
 * @param {{ algorithm: string, heuristic: string }} config - Configuration object.
 * @returns {Promise<Object>} Serialized solution data or error.
 */
ipcMain.handle('run-solver', async (event, config) => ()

/**
 * Handles the 'export-output' IPC event from the renderer.
 * Prompts the user to choose a file location and writes the solution output to a `.txt` file.
 *
 * @returns {Promise<Object>} Result object with success status or error message.
 */
ipcMain.handle('export-output', async () => ()

```

preload.js

```

contextBridge.exposeInMainWorld('electronAPI', {
  pickInputFile: () => ipcRenderer.invoke('open-input'),
  runSolver: (config) => ipcRenderer.invoke('run-solver', config),
  exportOutput: () => ipcRenderer.invoke('export-output'),
});

```

C. Struktur Data

Berikut adalah struktur data berupa kelas objek yang diimplementasi pada aplikasi ini.

inputstruct.js

```

class InputStruct {
  /**
   * @param {number} row - The number of rows in the puzzle board.
   * @param {number} col - The number of columns in the puzzle board.

```

```

    * @param {number} count - The total number of distinct cars parsed.
    * @param {string[]} state - The board state as an array of strings (each row as a string).
    * @param {[number, number]} goalPos - The position of the goal cell, represented as [row,
col].
    * @param {string[]} [errors=[]) - An optional list of error messages from input validation.
    */
}

```

puzzlestruct.js

```

class CarInfo {
    /**
     * @param {'-' | '|'} orientation - The orientation of the car: horizontal '-' or vertical '|'.
     * @param {number} length - The length of the car (typically 2 or 3).
     * @param {[number, number]} position - The top-left position of the car as [row, col].
     */
}

class PuzzleState {
    /**
     * @param {number} row - Number of rows in the board.
     * @param {number} col - Number of columns in the board.
     * @param {string} board - 1D string representing the board contents row-major.
     * @param {[number, number]} goalPos - Goal position on the board as [row, col].
     * @param {Map<string, CarInfo>} cars - Map of car symbols to their information.
     */
}

class SearchNode {
    /**
     * @param {PuzzleState} state - The puzzle state associated with this node.
     * @param {number} cost - The total cost used for prioritization (e.g.,  $g(n) + h(n)$ ).
     * @param {SearchNode|null} [parent=null] - The parent node in the search tree.
     * @param {number} [g=0] - The cost from the start node to this node.
     */
}

class SearchOutput {
    /**
     * @param {SearchNode|null} node - The final node leading to the solution, or null if not
found.
     * @param {number} totalMove - Total number of unique states visited during search.
     */
}

class SearchNodePriorityQueue

```

outputstruct.js

```
class OutputStruct{  
  /**  
   * @param {number} time - Time taken to solve the puzzle (in milliseconds or custom unit).  
   * @param {number} totalMove - Total number of node expansions or visited states.  
   * @param {number} moveCount - Number of moves in the final solution path.  
   * @param {OutputState[]} states - List of states traversed (each with message and board state).  
   * @param {string} [mainMessage=""] - Optional summary or result message.  
  */  
}  
  
class OutputState{  
  /**  
   * @param {string} message - A message describing this state  
   * @param {PuzzleState} state - The board state object  
  */  
}
```

D. Fungsi Pembantu Algoritma

helper.js

```
function expandNode(node, costFn) {  
  const children = [];  
  
  const currentState = node.state;  
  const gNext = node.g + 1;  
  
  const nextStates = generateNextStates(currentState);  
  
  for (const newState of nextStates) {  
    const tempNode = new SearchNode(newState, 0, node, gNext);  
    tempNode.cost = costFn(tempNode);  
    children.push(tempNode);  
  }  
  
  return children;  
}  
  
function generateNextStates(state){  
  Fungsi pembantu expandNode yang mengambil semua gerakan berikutnya yang mungkin
```

dari setiap piece dalam game

}

function **canMove**(state, car, dr, dc, step){

Fungsi pembantu generateNextStates untuk menentukan apakah suatu mobil dapat bergerak ke arah tertentu atau tidak.

}

function **moveCar**(state, carId, dr, dc, step) {

Fungsi pembantu generateNextStates untuk memindahkan mobil setelah dicek oleh canMove

}

heuristic.js

function **getCostFunction**(strategy, heuristicFn) {

return node => {
if (isPrimaryOnGoal(node.state)) return -1;

const h = heuristicFn(node.state);

if (strategy === "UCS") return node.g;

if (strategy === "Greedy") return h;

if (strategy === "AStar") return node.g + h;

};

}

function **heuristicBlockerCount**(state){

Implementasi heuristic yang menghitung jumlah mobil penghalang primary piece dari exit

}

function **heuristicDistanceToFreedom**(state){

Implementasi heuristic yang menghitung jumlah mobil penghalang secara rekursif

}

function **freedomDepth**(state, carId, visited = new Set()) {

Fungsi pembantu heuristicDistanceToFreedom untuk proses rekursi

}

function **checkHorCarBlockingExit**(state){

Fungsi cek di awal untuk cek apakah ada mobil yang menghalangi mobil target dari target dan sejajar dengannya, jika iya langsung output tidak bisa diselesaikan puzzle-nay

}

E. Fungsi Algoritma

Dengan bantuan dari fungsi pembantu, implementasi UCS, GBFS, dan A* sangat mirip, sebagai berikut.

Program di algorithms/

```
function solver(initState, heuristicFn // parameter ini tidak ada pada UCS){
  if (checkHorCarBlockingExit(initState)){
    return new SearchOutput(null, 0);
  }
  const startNode = new SearchNode(initState, 0, null, 0);
  const queue = new SearchNodePriorityQueue();
  queue.enqueue(startNode);
  const visited = new Set();
  const costFn = getCostFunction("Greedy", heuristicFn); // tidak ada pada UCS
  let totalMove = 0;

  while (queue.length > 0) {
    const current = queue.dequeue();

    const stateKey = current.state.board;
    if (visited.has(stateKey)) continue;
    visited.add(stateKey);

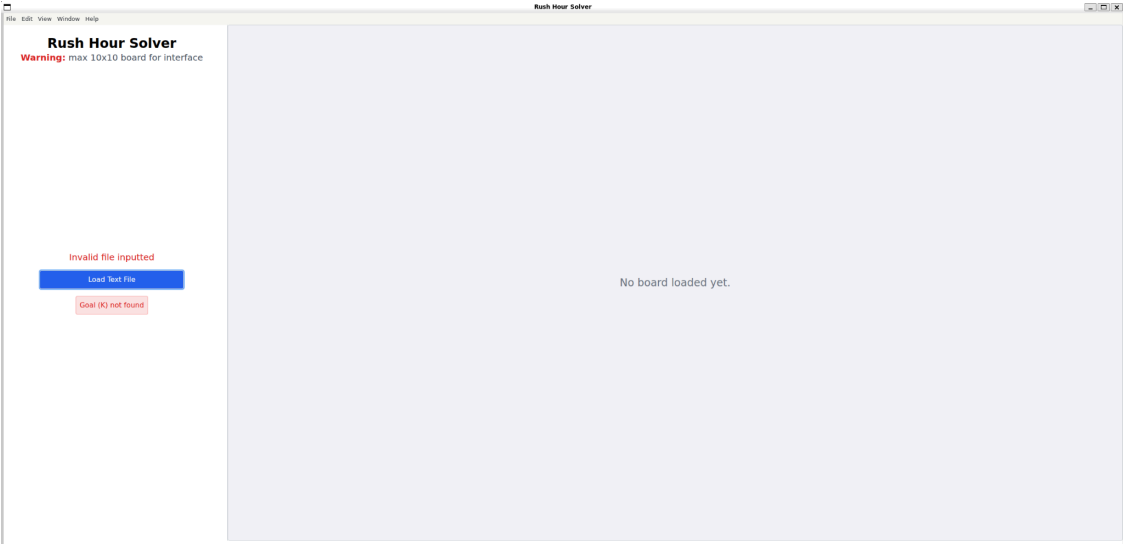
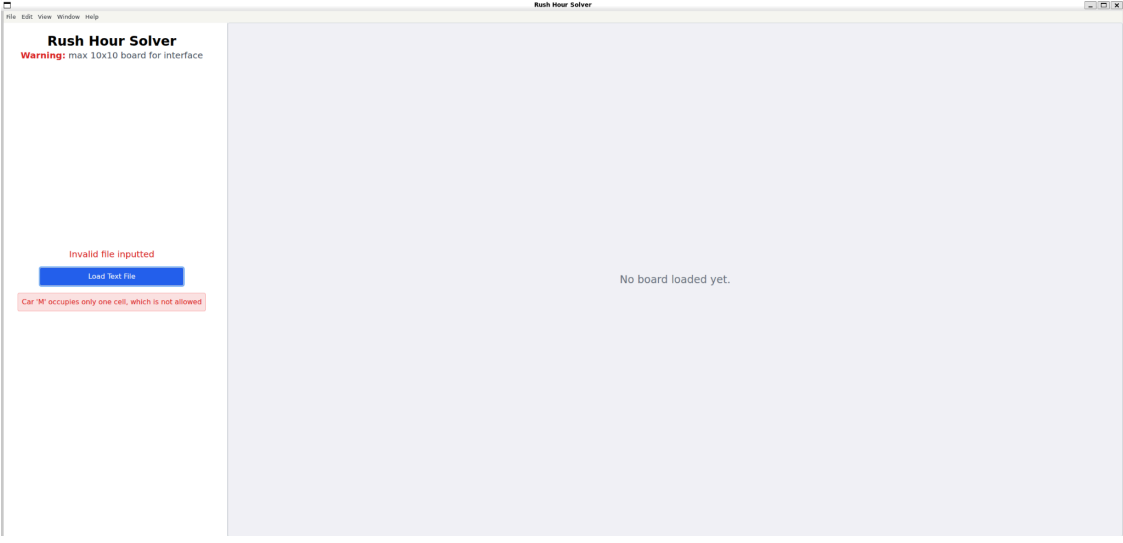
    if (isPrimaryOnGoal(current.state)) {
      return new SearchOutput(current, totalMove);
    }
    totalMove++

    const neighbors = expandNode(current, costFn);

    for (const neighbor of neighbors) {
      if (!visited.has(neighbor.state.board)) {
        queue.enqueue(neighbor);
      }
    }
  }

  return new SearchOutput(null, totalMove); // No solution found
}
```

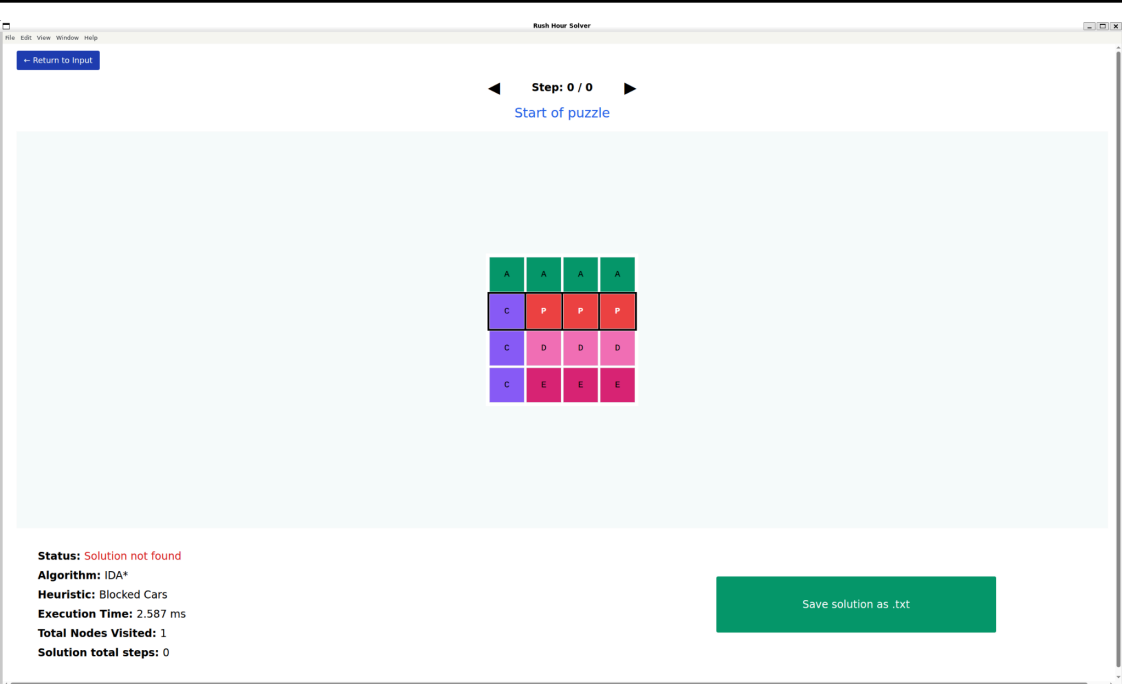
IV. Pengujian (Input dan Output)

No	Penjelas Status	Input/Output
1	Gagal input (1)	 <p>The screenshot shows the 'Rush Hour Solver' application window. On the left sidebar, there is a red error message 'Invalid file inputted' above a blue 'Load Text File' button, and a red message 'Goal (K) not found' below it. The main area on the right is a large light blue rectangle with the text 'No board loaded yet.' in the center.</p>
2	Gagal input (2)	 <p>The screenshot shows the 'Rush Hour Solver' application window. On the left sidebar, there is a red error message 'Car 'M' occupies only one cell, which is not allowed' below the 'Load Text File' button. The main area on the right is a large light blue rectangle with the text 'No board loaded yet.' in the center.</p>

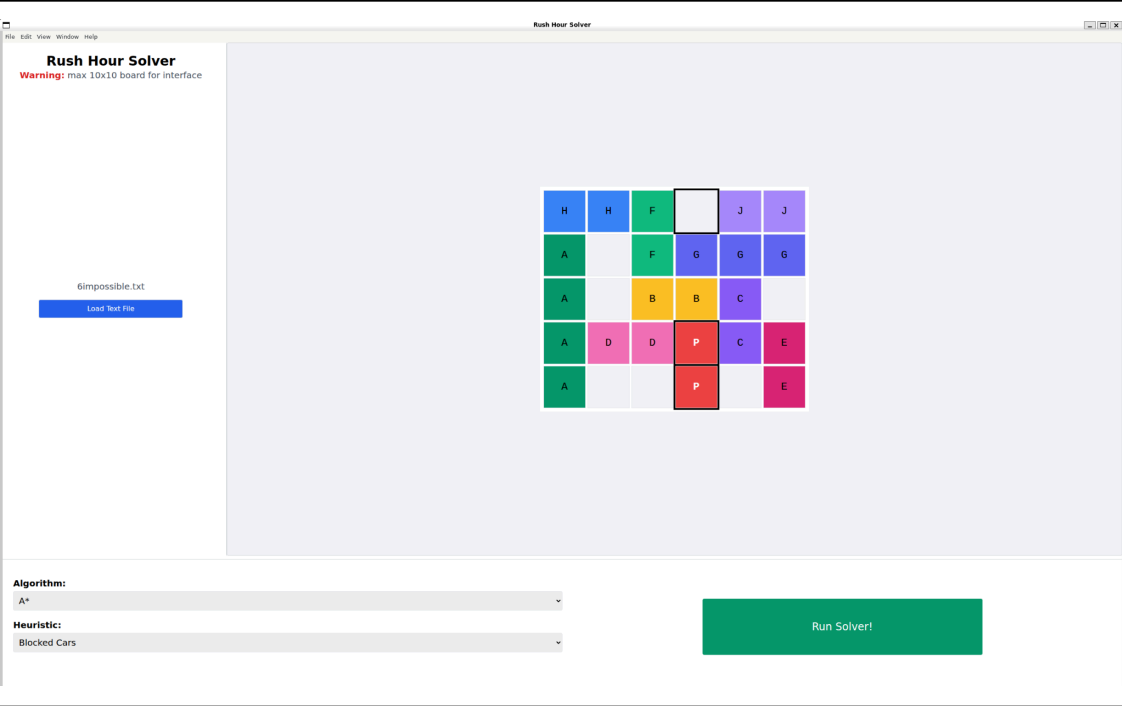
3	Input board besar (11x11)	<div><div><div><div><div>Rush Hour Solver</div><div>Warning: max 10x10 board for interface</div><div>11.txt</div><div>Load Text File</div></div><div>Board successfully loaded! But the board is too big to display :(</div></div></div><div><div>Algorithm: UCS</div><div>Heuristic: Blocked Cars</div><div>Run Solver!</div></div></div>																																				
4	Input mobil sejajar di depan mobil primary	<div><div><div><div><div>Rush Hour Solver</div><div>Warning: max 10x10 board for interface</div><div>4front.txt</div><div>Load Text File</div></div><div><table><tr><td></td><td></td><td>A</td><td>B</td><td>B</td><td>B</td></tr><tr><td></td><td></td><td>A</td><td></td><td></td><td></td></tr><tr><td>P</td><td>P</td><td>D</td><td>C</td><td>C</td><td>E</td></tr><tr><td>F</td><td>F</td><td>D</td><td>G</td><td>G</td><td>E</td></tr><tr><td>H</td><td>I</td><td>J</td><td>J</td><td></td><td>E</td></tr><tr><td>H</td><td>I</td><td>L</td><td>L</td><td></td><td></td></tr></table></div></div><div><div>Algorithm: UCS</div><div>Heuristic: Blocked Cars</div><div>Run Solver!</div></div></div></div>			A	B	B	B			A				P	P	D	C	C	E	F	F	D	G	G	E	H	I	J	J		E	H	I	L	L		
		A	B	B	B																																	
		A																																				
P	P	D	C	C	E																																	
F	F	D	G	G	E																																	
H	I	J	J		E																																	
H	I	L	L																																			

	<p>Output mobil sejajar di depan mobil primary</p>	<div><div><div>Rush Hour Solver</div><div>File Edit View Window Help</div><div><div>Return to Input</div><div>Step: 0 / 0</div><div>Start of puzzle</div></div><div><table><tr><td></td><td></td><td>A</td><td>B</td><td>B</td><td>B</td></tr><tr><td></td><td></td><td>A</td><td></td><td></td><td></td></tr><tr><td>P</td><td>P</td><td>D</td><td>C</td><td>C</td><td>E</td></tr><tr><td>F</td><td>F</td><td>D</td><td>G</td><td>G</td><td>E</td></tr><tr><td>H</td><td>I</td><td>J</td><td>J</td><td></td><td>E</td></tr><tr><td>H</td><td>I</td><td>L</td><td>L</td><td></td><td></td></tr></table></div><div><div>Status: Solution not found</div><div>Algorithm: UCS</div><div>Execution Time: 5.036 ms</div><div>Total Nodes Visited: 0</div><div>Solution total steps: 0</div><div>Save solution as .txt</div></div></div></div>			A	B	B	B			A				P	P	D	C	C	E	F	F	D	G	G	E	H	I	J	J		E	H	I	L	L		
		A	B	B	B																																	
		A																																				
P	P	D	C	C	E																																	
F	F	D	G	G	E																																	
H	I	J	J		E																																	
H	I	L	L																																			
5	<p>Input mobil sama sekali tidak bisa gerak</p>	<div><div><div>Rush Hour Solver</div><div>File Edit View Window Help</div><div><div>Rush Hour Solver</div><div>Warning: max 10x10 board for interface</div><div><div>5move.txt</div><div>Load Text File</div></div><div><table><tr><td></td><td>A</td><td>A</td><td>A</td><td>A</td></tr><tr><td>C</td><td>P</td><td>P</td><td>P</td><td></td></tr><tr><td>C</td><td>D</td><td>D</td><td>D</td><td></td></tr><tr><td>C</td><td>E</td><td>E</td><td>E</td><td></td></tr></table></div></div><div><div>Algorithm: UCS</div><div>Heuristic: Blocked Cars</div><div>Run Solver!</div></div></div></div>		A	A	A	A	C	P	P	P		C	D	D	D		C	E	E	E																	
	A	A	A	A																																		
C	P	P	P																																			
C	D	D	D																																			
C	E	E	E																																			

Output mobil sama sekali tidak bisa gerak



6 Input mustahil diselesaikan




Output mustahil diselesaikan UCS

Rush Hour Solver

File Edit View Window Help

← Return to Input

Step: 0 / 0
Start of puzzle



Status: Solution not found
Algorithm: UCS
Execution Time: 23.673 ms
Total Nodes Visited: 362
Solution total steps: 0

Save solution as .txt


Output mustahil diselesaikan GBFS

Rush Hour Solver

File Edit View Window Help

← Return to Input

Step: 0 / 0
Start of puzzle



Status: Solution not found
Algorithm: GBFS
Heuristic: Blocked Cars
Execution Time: 11.845 ms
Total Nodes Visited: 362
Solution total steps: 0

Save solution as .txt

Output mustahil diselesaikan A*

File Edit View Window Help

← Return to Input

Step: 0 / 0
Start of puzzle

H	H	F		J	J
A		F	G	G	G
A		B	B	C	
A	D	D	P	C	E
A			P		E

Status: Solution not found

Algorithm: A*

Heuristic: Blocked Cars

Execution Time: 14.892 ms

Total Nodes Visited: 362

Solution total steps: 0

Save solution as .txt

Output mustahil diselesaikan IDA*

File Edit View Window Help

← Return to Input

Step: 0 / 0
Start of puzzle

H	H	F		J	J
A		F	G	G	G
A		B	B	C	
A	D	D	P	C	E
A			P		E

Status: Solution not found

Algorithm: IDA*

Heuristic: Blocked Cars

Execution Time: 1948.458 ms

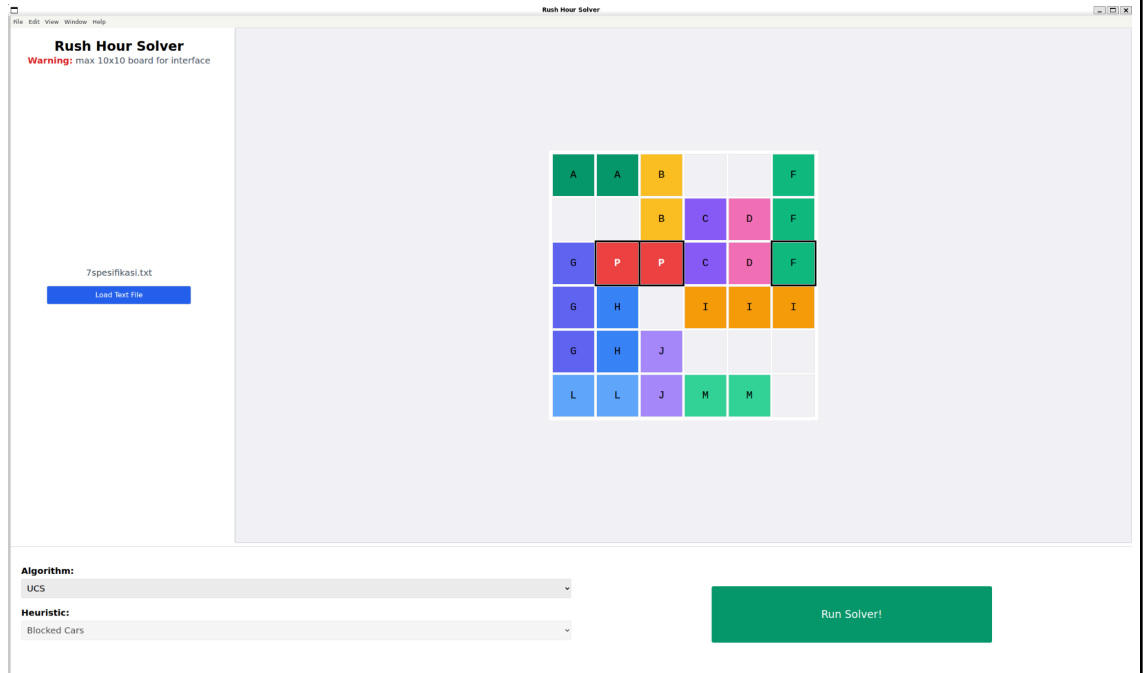
Total Nodes Visited: 362

Solution total steps: 0

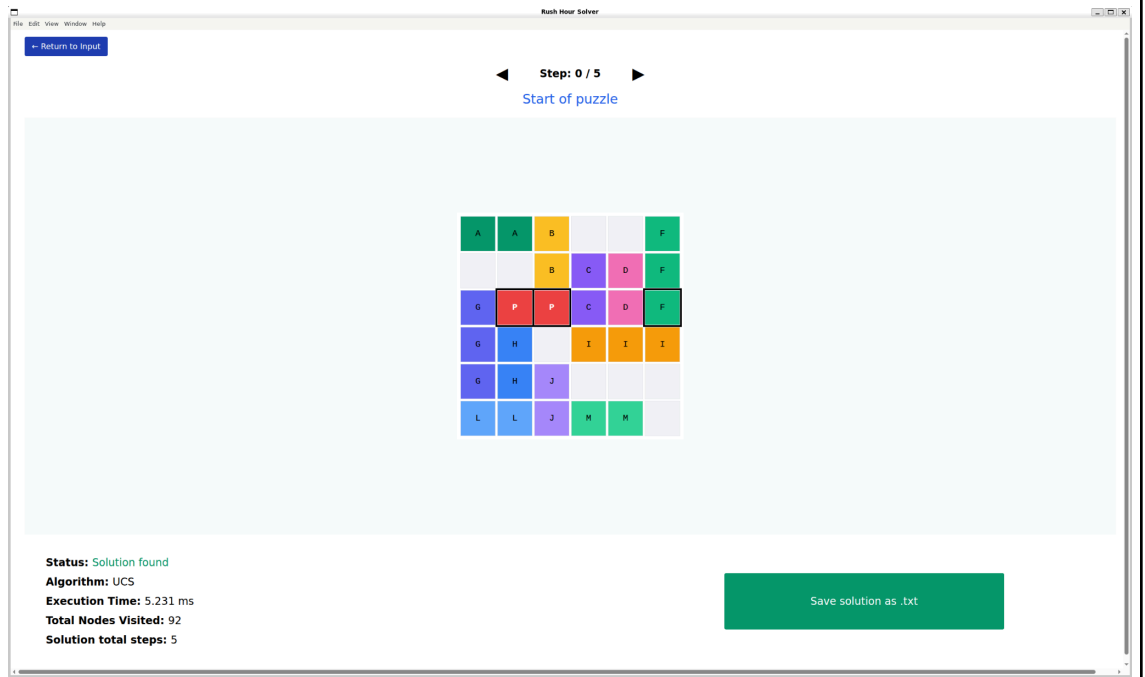
Save solution as .txt

7

Input spesifikasi



Output UCS



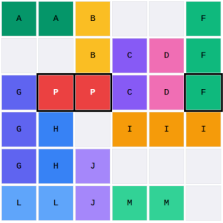
Output GBFS
heuristik Blocked
Cars

Rush Hour Solver

File Edit View Window Help

Return to Input

Step: 0 / 5
Start of puzzle



The grid is a 6x7 array of colored squares. Row 1: Green(A), Green(A), Yellow(B), White, White, White, Green(F). Row 2: White, White, Yellow(B), Purple(C), Pink(D), Pink(F). Row 3: Green(G), Red(P), Red(P), Purple(C), Pink(D), Green(F). Row 4: Green(G), Blue(H), White, Orange(I), Orange(I), Orange(I). Row 5: Green(G), Blue(H), Purple(J), White, White, White. Row 6: Blue(L), Blue(L), Purple(J), Green(M), Green(M), White.

Status: Solution found
Algorithm: GBFS
Heuristic: Blocked Cars
Execution Time: 1.780 ms
Total Nodes Visited: 8
Solution total steps: 5

Save solution as .txt

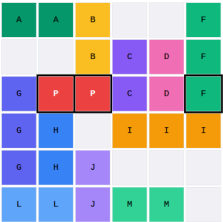
Output GBFS
heuristik
Distance to
Freedom

Rush Hour Solver

File Edit View Window Help

Return to Input

Step: 0 / 5
Start of puzzle



The grid is a 6x7 array of colored squares. Row 1: Green(A), Green(A), Yellow(B), White, White, White, Green(F). Row 2: White, White, Yellow(B), Purple(C), Pink(D), Pink(F). Row 3: Green(G), Red(P), Red(P), Purple(C), Pink(D), Green(F). Row 4: Green(G), Blue(H), White, Orange(I), Orange(I), Orange(I). Row 5: Green(G), Blue(H), Purple(J), White, White, White. Row 6: Blue(L), Blue(L), Purple(J), Green(M), Green(M), White.

Status: Solution found
Algorithm: GBFS
Heuristic: Distance to Freedom
Execution Time: 1.592 ms
Total Nodes Visited: 5
Solution total steps: 5

Save solution as .txt

Output A*
heuristik Blocked
Cars

Rush Hour Solver

File Edit View Window Help

-- Return to Input

Step: 0 / 5
Start of puzzle

Status: Solution found
Algorithm: A*
Heuristic: Blocked Cars
Execution Time: 2.532 ms
Total Nodes Visited: 27
Solution total steps: 5

Save solution as .txt

Output A*
heuristik
Distance to
Freedom

Rush Hour Solver

File Edit View Window Help

-- Return to Input

Step: 0 / 5
Start of puzzle

Status: Solution found
Algorithm: A*
Heuristic: Distance to Freedom
Execution Time: 1.763 ms
Total Nodes Visited: 12
Solution total steps: 5

Save solution as .txt

Output IDA*
heuristik Blocked
Cars

File Edit View Window Help

← Return to Input

Step: 0 / 5
Start of puzzle

A	A	B			F
		B	C	D	F
G	P	P	C	D	F
G	H		I	I	I
G	H	J			
L	L	J	H	H	

Status: Solution found

Algorithm: IDA*

Heuristic: Blocked Cars

Execution Time: 1.494 ms

Total Nodes Visited: 10

Solution total steps: 5

Save solution as .txt

Output IDA*
heuristik
Distance to
Freedom

File Edit View Window Help

← Return to Input

Step: 0 / 5
Start of puzzle

A	A	B			F
		B	C	D	F
G	P	P	C	D	F
G	H		I	I	I
G	H	J			
L	L	J	H	H	

Status: Solution found

Algorithm: IDA*

Heuristic: Distance to Freedom

Execution Time: 1.426 ms

Total Nodes Visited: 6

Solution total steps: 5

Save solution as .txt

8	Input 6x6 dengan solusi optimal 51 gerakan	<div><div><div><div><div>Rush Hour Solver</div><div>Warning: max 10x10 board for interface</div><div>8hardest51.txt</div><div>Load Text File</div></div><div><table><tr><td>A</td><td>B</td><td>B</td><td></td><td>C</td><td></td></tr><tr><td>A</td><td>D</td><td>E</td><td></td><td>C</td><td>F</td></tr><tr><td>A</td><td>D</td><td>E</td><td>P</td><td>P</td><td>F</td></tr><tr><td>G</td><td>G</td><td>G</td><td>H</td><td></td><td>F</td></tr><tr><td></td><td></td><td>I</td><td>H</td><td>J</td><td>J</td></tr><tr><td>L</td><td>L</td><td>I</td><td>M</td><td>M</td><td></td></tr></table></div><div><div>Algorithm: UCS</div><div>Heuristic: Distance to Freedom</div><div>Run Solver!</div></div></div></div></div>	A	B	B		C		A	D	E		C	F	A	D	E	P	P	F	G	G	G	H		F			I	H	J	J	L	L	I	M	M	
A	B	B		C																																		
A	D	E		C	F																																	
A	D	E	P	P	F																																	
G	G	G	H		F																																	
		I	H	J	J																																	
L	L	I	M	M																																		
	Output 6x6 dengan solusi optimal 51 gerakan UCS	<div><div><div><div><div>Rush Hour Solver</div><div>Return to Input</div><div>Step: 0 / 51</div><div>Start of puzzle</div></div><div><table><tr><td>A</td><td>B</td><td>B</td><td></td><td>C</td><td></td></tr><tr><td>A</td><td>D</td><td>E</td><td></td><td>C</td><td>F</td></tr><tr><td>A</td><td>D</td><td>E</td><td>P</td><td>P</td><td>F</td></tr><tr><td>G</td><td>G</td><td>G</td><td>H</td><td></td><td>F</td></tr><tr><td></td><td></td><td>I</td><td>H</td><td>J</td><td>J</td></tr><tr><td>L</td><td>L</td><td>I</td><td>M</td><td>M</td><td></td></tr></table></div><div><div>Status: Solution found</div><div>Algorithm: UCS</div><div>Execution Time: 104.638 ms</div><div>Total Nodes Visited: 2810</div><div>Solution total steps: 51</div><div>Save solution as .txt</div></div></div></div></div>	A	B	B		C		A	D	E		C	F	A	D	E	P	P	F	G	G	G	H		F			I	H	J	J	L	L	I	M	M	
A	B	B		C																																		
A	D	E		C	F																																	
A	D	E	P	P	F																																	
G	G	G	H		F																																	
		I	H	J	J																																	
L	L	I	M	M																																		

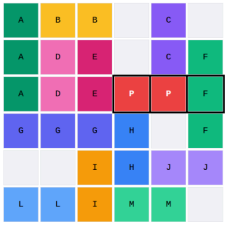
Output 6x6
dengan solusi
optimal 51
gerakan GBFS
heuristik
Distance to
Freedom

Rush Hour Solver

File Edit View Window Help

Return to Input

Step: 0 / 64
Start of puzzle



The 6x6 grid contains the following pieces (row by row):
Row 1: A (green), B (yellow), B (yellow), empty, C (purple), empty
Row 2: A (green), D (pink), E (pink), empty, C (purple), F (green)
Row 3: A (green), D (pink), E (pink), P (red), P (red), F (green)
Row 4: G (blue), G (blue), G (blue), H (blue), empty, F (green)
Row 5: empty, empty, I (orange), H (blue), J (purple), J (purple)
Row 6: L (blue), L (blue), I (orange), H (blue), M (green), empty

Status: Solution found
Algorithm: GBFS
Heuristic: Distance to Freedom
Execution Time: 58.898 ms
Total Nodes Visited: 1310
Solution total steps: 64

Save solution as .txt

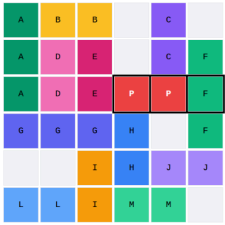
Output 6x6
dengan solusi
optimal 51
gerakan A*
heuristik
Distance to
Freedom

Rush Hour Solver

File Edit View Window Help

Return to Input

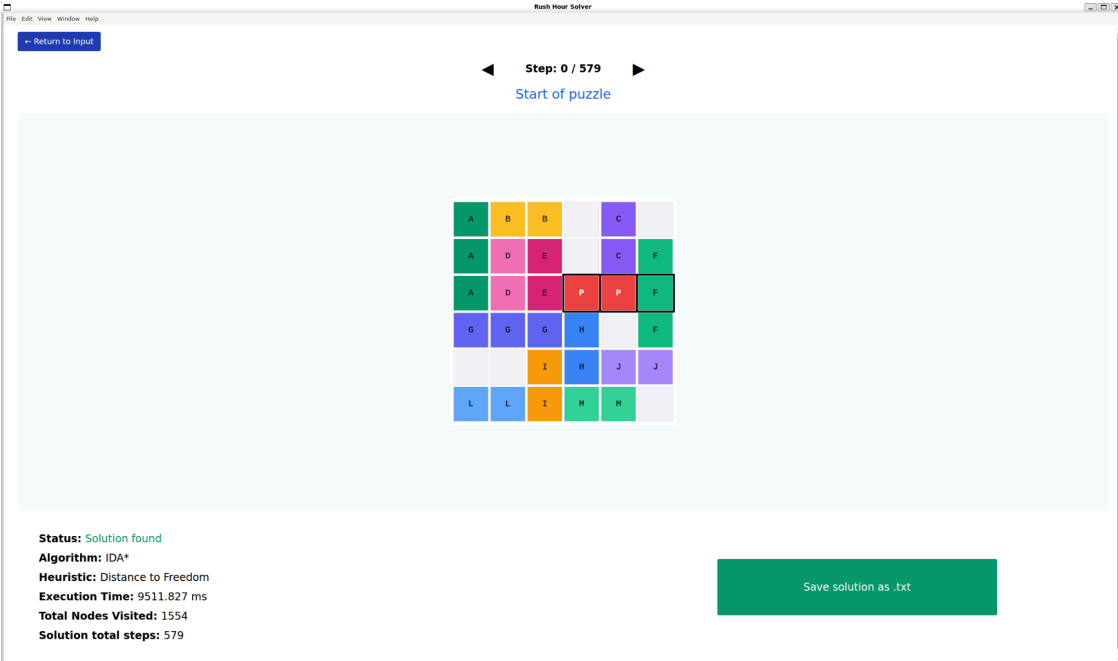
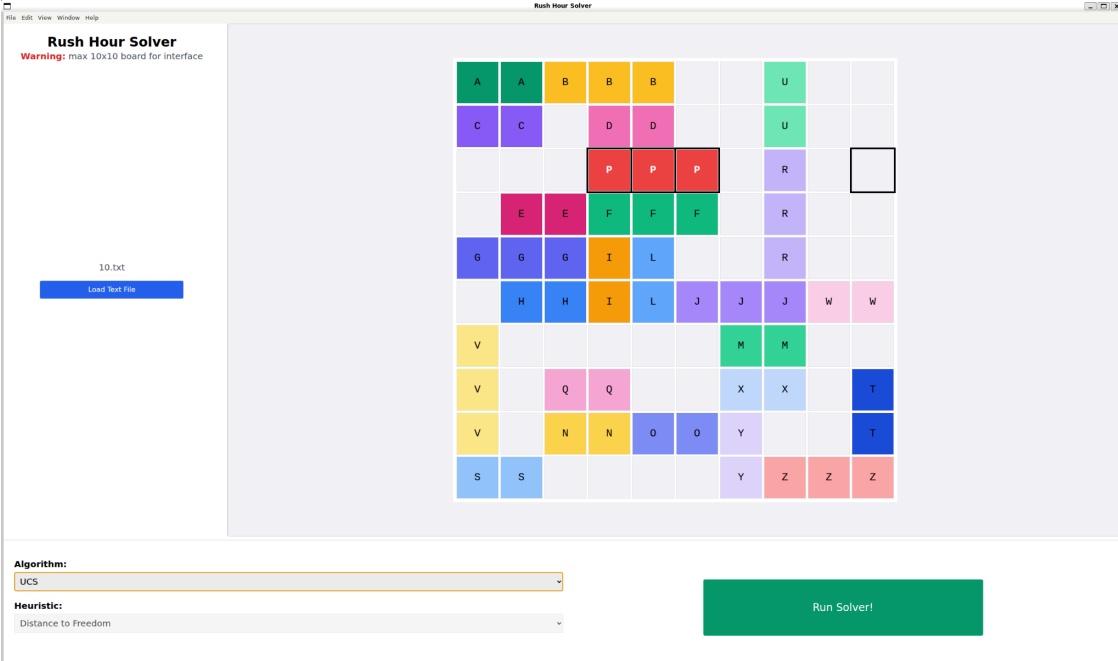
Step: 0 / 51
Start of puzzle



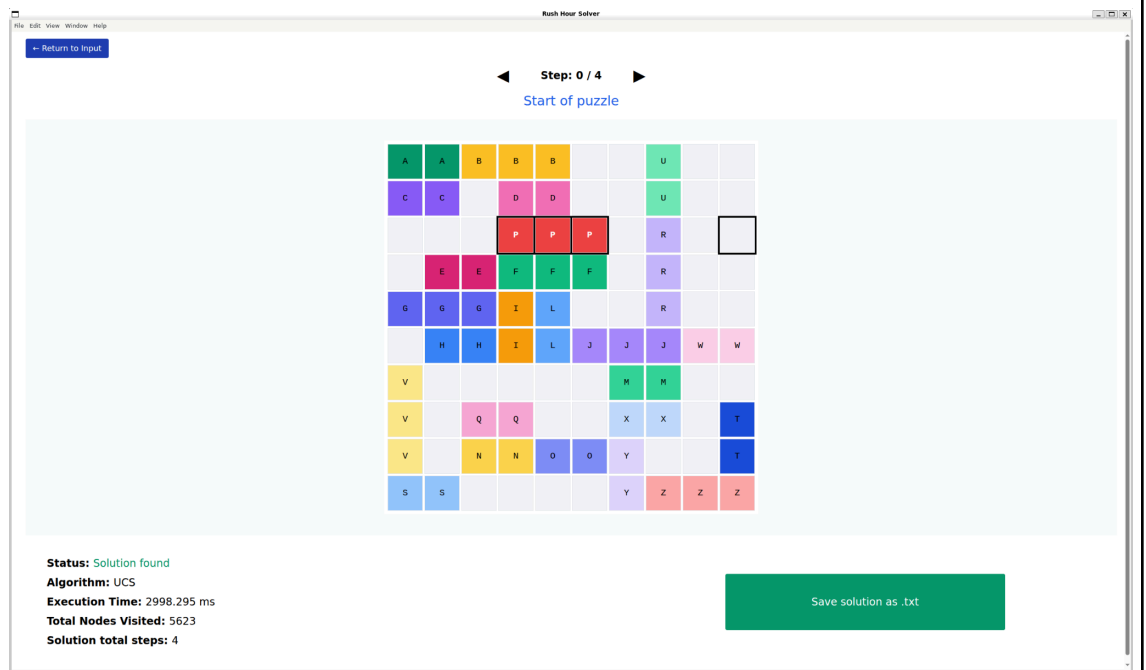
The 6x6 grid contains the following pieces (row by row):
Row 1: A (green), B (yellow), B (yellow), empty, C (purple), empty
Row 2: A (green), D (pink), E (pink), empty, C (purple), F (green)
Row 3: A (green), D (pink), E (pink), P (red), P (red), F (green)
Row 4: G (blue), G (blue), G (blue), H (blue), empty, F (green)
Row 5: empty, empty, I (orange), H (blue), J (purple), J (purple)
Row 6: L (blue), L (blue), I (orange), H (blue), M (green), empty

Status: Solution found
Algorithm: A*
Heuristic: Distance to Freedom
Execution Time: 93.025 ms
Total Nodes Visited: 2082
Solution total steps: 51

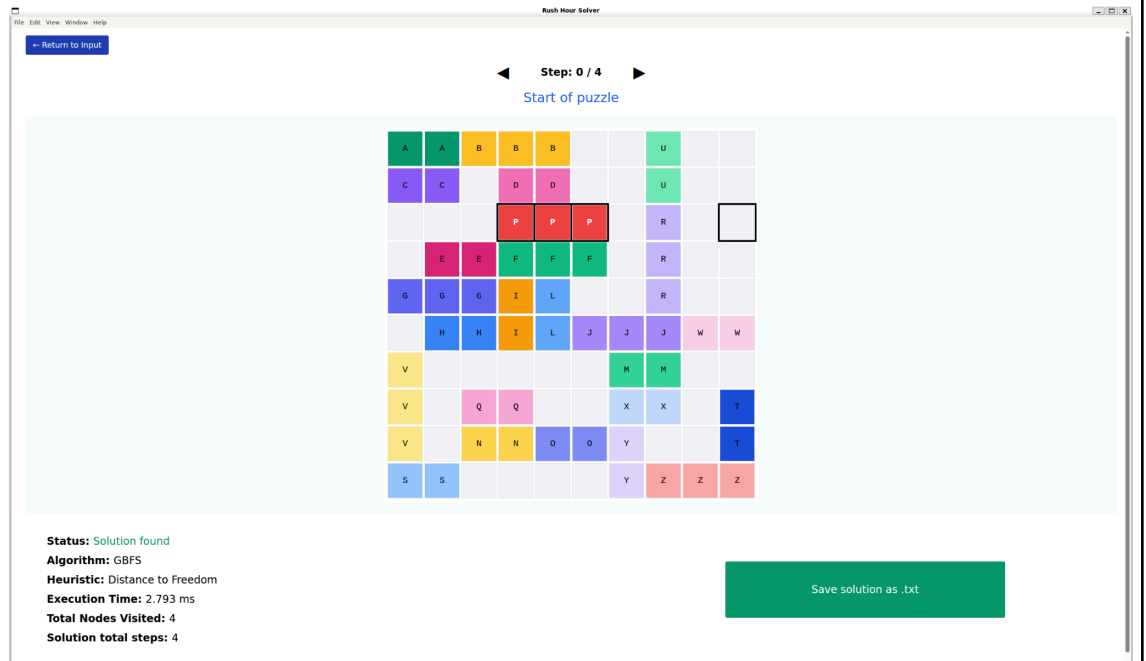
Save solution as .txt

	<p>Output 6x6 dengan solusi optimal 51 gerakan IDA* heuristik Distance to Freedom</p>	
9	<p>Input 10x10</p>	

Output 10x10
UCS



Output 10x10
GBFS heuristic
Distance to
Freedom



Output 10x10 A*
heuristic
Distance to
Freedom

Rush Hour Solver

File Edit View Window Help

← Return to Input

Step: 0 / 4
Start of puzzle

Status: Solution found
Algorithm: A*
Heuristic: Distance to Freedom
Execution Time: 5.029 ms
Total Nodes Visited: 4
Solution total steps: 4

Save solution as .txt

Output 10x10
IDA* heuristic
Distance to
Freedom

Rush Hour Solver

File Edit View Window Help

← Return to Input

Step: 0 / 6
Start of puzzle

Status: Solution found
Algorithm: IDA*
Heuristic: Distance to Freedom
Execution Time: 2.134 ms
Total Nodes Visited: 7
Solution total steps: 6

Save solution as .txt

V. Hasil Analisis Percobaan

A. Algoritma Uniform Cost Search

Secara garis besar dari hasil percobaan, algoritma ini cukup bisa diandalkan untuk pencarian solusi paling optimal. Bahkan bisa menjamin karena sifatnya sebagai BFS pada tugas kecil ini.

Kompleksitas waktu: $O(b^d)$

UCS memiliki kompleksitas waktu eksponensial karena pada dasarnya merupakan bentuk dari Breadth-First Search (BFS), tetapi dengan pembobotan. Di dalam pencarian tanpa informasi heuristik ini, UCS menjelajahi semua simpul yang memiliki nilai $g(n)$ rendah terlebih dahulu, tanpa mempedulikan seberapa dekat simpul tersebut ke tujuan. Oleh karena itu, dalam kasus terburuk, UCS akan mengeksplorasi hampir seluruh ruang pencarian hingga kedalaman d , dengan b adalah merupakan branching factor (jumlah kemungkinan per langkah), dan d adalah kedalaman dari solusi optimal. Karena UCS harus menyimpan semua simpul yang telah dijelajahi untuk mencegah eksplorasi ulang, hal ini membuatnya sangat lambat untuk masalah besar, meskipun selalu menjamin solusi optimal.

Kalau dijabarkan lagi, kompleksitas waktu itu berdasarkan penjumlahan bahwa pada kondisi awal ada 1, dengan branching factor b atau jumlah kemungkinan pada langkah berikutnya, penjumlahan kompleksitas waktunya menjadi 1, b , b^2 , b^3 , ..., b^d . Dengan penjumlahan pada notasi algoritma biasanya mengambil ekspresi terbesar atau b^d .

B. Algoritma Greedy Best First Search

Algoritma GBFS memiliki kelebihan dalam kecepatan eksekusi program karena dapat mencapai tujuan dengan heuristik yang dikembangkan. Tetapi karena itu juga merupakan kelemahan dari GBFS, karena heuristik dapat berupa apa pun, bisa saja heuristik tersebut membuat pencarian tujuan lebih lama dari seharusnya.

Kompleksitas waktu: $O(b^m)$

Greedy Best First Search memiliki kompleksitas waktu yang juga eksponensial pada umumnya, yakni $O(b^m)$, m adalah kedalaman maksimal pencarian (bukan solusi optimal). Hal ini terjadi karena GBFS sangat bergantung pada heuristik $h(n)$ untuk menentukan arah pencarian. Jika heuristik yang digunakan buruk atau tidak akurat, maka pencarian bisa berjalan menyamping atau bahkan mundur, menyebabkan eksplorasi area yang luas dan tidak relevan. GBFS akan terus memilih simpul dengan estimasi $h(n)$ terkecil tanpa memperhatikan seberapa jauh simpul tersebut dari awal ($g(n)$), yang membuatnya bisa terjebak pada jalur yang tampaknya

menjanjikan namun sebenarnya panjang atau buntu. Meskipun dalam banyak kasus bisa cepat, performanya sangat tidak stabil tergantung pada kualitas heuristik.

C. Algoritma A*

Dari hasil percobaan dapat disimpulkan A* merupakan algoritma yang menyampurkan kelebihan dari UCS dan GBFS. Ini dapat dilihat dari performa A* yang kebanyakan kali dapat solusi optimal atau setidaknya sangat mendekati solusi optimal (kelebihan UCS). Tetapi tanpa kekurangan boros tempat dan waktu, akibat heuristik dari fungsi $h(n)$ yang juga digunakan pada GBFS.

Kompleksitas waktu: $O(b^d)$ (dengan heuristik admissible)

A* secara teori memiliki kompleksitas waktu yang juga eksponensial dalam kasus terburuk, sama seperti UCS, yaitu $O(b^d)$, tetapi bisa jauh lebih efisien jika heuristik $h(n)$ yang digunakan *admissible* (tidak melebihi biaya sebenarnya ke tujuan). Kompleksitas ini muncul karena A* tetap harus menjelajahi banyak simpul sampai menemukan solusi, tetapi fungsi biayanya $f(n) = g(n) + h(n)$ yang menggabungkan informasi dari awal dan estimasi ke tujuan, algoritma ini memangkas banyak simpul yang tidak perlu dibanding UCS atau GBFS. Jika $h(n)$ semakin mendekati nilai sesungguhnya, maka jumlah simpul yang perlu dievaluasi akan jauh lebih kecil. Maka, kasus terburuknya tetap eksponensial, A* adalah salah satu algoritma paling efisien dan optimal dalam praktik.

D. Algoritma IDA*

Dari percobaan, algoritma ini dapat menjadi sangat efisien dengan heuristik tertentu, bahkan dapat mengalahkan A* biasa dan GBFS. Namun, terkadang pemrosesannya sangat lama secara waktu.

Kompleksitas waktu: $O(b^d)$ (dengan heuristik admissible)

IDA* memiliki kompleksitas waktu yang sama dengan A*, yaitu $O(b^d)$, namun dengan *overhead tambahan* karena pendekatannya yang berbasis *iterasi*. IDA* melakukan pencarian dengan cara DFS namun dengan batasan $f(n)$ (bukan depth, tapi $f(n) = g(n) + h(n)$ seperti pada A*), dan menaikkan batasan tersebut secara bertahap. Ini membuat IDA* harus mengulang banyak pencarian pada simpul yang sama di setiap iterasi. Meskipun ini meningkatkan efisiensi penggunaan memori menjadi hanya $O(d)$, waktu yang dibutuhkan bisa menjadi lebih besar daripada A* karena harus menjelajahi ulang banyak simpul. Namun, untuk perangkat dengan keterbatasan memori, IDA* merupakan kompromi yang layak antara performa dan efisiensi ruang.

VI. Implementasi Bonus

A. Algoritma Alternatif

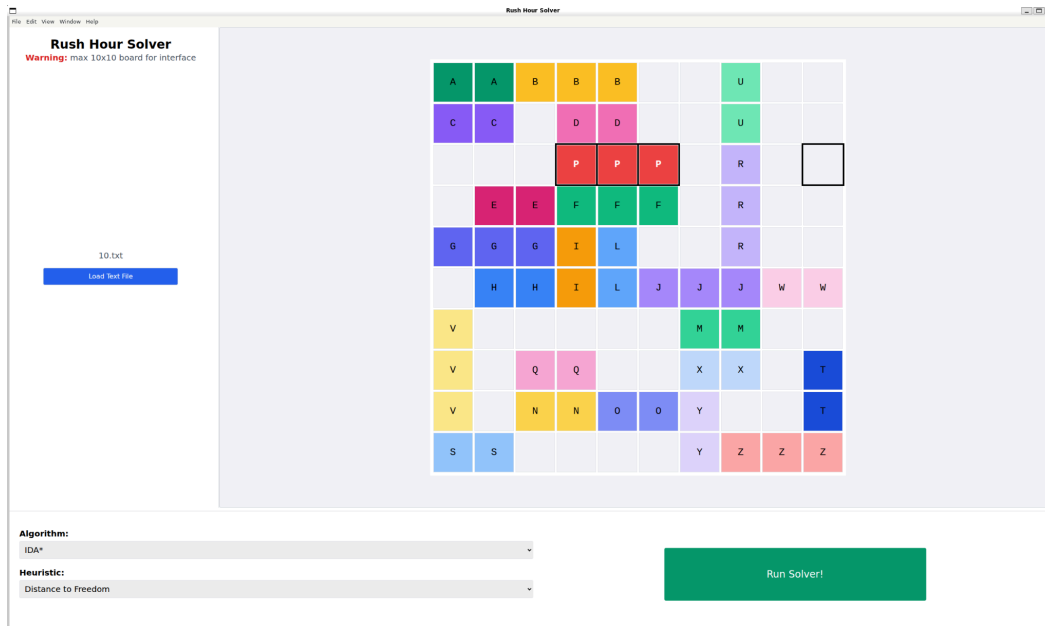
Algoritma alternatif yang dikembangkan adalah IDA* atau Iterative Deepening A*. Ini adalah algoritma yang berbasis DFS yang dibatasi kedalaman pencariannya menggunakan fungsi heuristik dari A. Setiap kali menemukan jarak yang melebihi threshold dilakukan penambahan threshold yang perlu diikuti node lainnya. Dengan algoritma ini, dapat dinikmati keunggulan dari A* yang sangat diandalkan ketepatannya dalam mencari solusi optimal dan juga keunggulan DFS yang cepat dan tidak boros memori.

B. Heuristik Alternatif

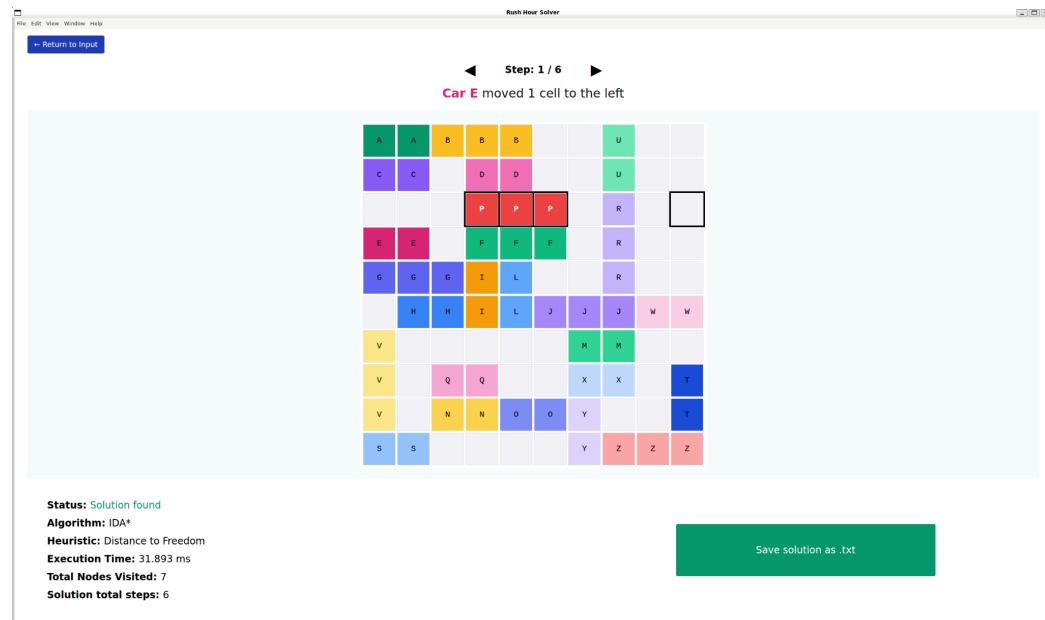
Pada Tupil ini, terdapat dua heuristik yang diimplementasi, yaitu Blocked Cars dan Distance to Freedom. Blocked cars sangat simple, hanya menjumlahkan mobil-mobil yang berbeda yang ada di antara primary car dan target. Sedangkan untuk Distance to Freedom, ini merupakan bagian rekursi dari Blocked Cars yang juga mempertimbangkan mobil-mobil yang juga menghalangi mobil menghalangi mobil yang menghalangi primary car langsung.

Kedua heuristik ini admissible. Untuk blocked cars, pasti iya, soalnya tidak mempertimbangkan apa yang menghalangi mobil-mobil yang di depan primary car. Dari itu, blocked car pasti admissible karena sudah memasang asumsi semua mobil di depannya bisa digerakan. Sedangkan untuk Distance to Freedom, sama halnya dengan blocked cars pasti admissible, karena walaupun kita secara rekursi menentukan mobil mana yang bisa digerakan, belum tentu gerakan itu pasti membuka jalan baru untuk mobil yang dihalangi olehnya.

C. GUI



Gambar VI.C.1. Interface input papan puzzle



Gambar VI.C.2. Interface output papan puzzle

Untuk menggunakan program ini, dikembangkan GUI untuk mempermudah pengguna dalam menggunakan aplikasi ini. Implementasi GUI menggunakan Electron dari javascript untuk menggunakan aplikasi desktop yang secara perkembangan sangat mirip dengan website dan mempermudah pembuatannya. Untuk styling, digunakan vite, tailwind, dan react dan melancarkan demo singkat ini.

Di bagian input program, terdapat berbagai fitur. Pertama itu adalah tombol untuk input tombol ini untuk memasukan file .txt yang diinginkan untuk analisis pathfinding. Di sebelah kanan itu, terdapat interface papan yang di loading di file .txt yang diberikan. Terakhir di bawah, terdapat dua dropdown, yakni pemilihan algoritma dan heuristik yang ingin digunakan selama pemrosesan.

Di bagian output program, fitur pertama yang ada adalah penampilan papan di tengah yang dapat digunakan tanda panah untuk memindahkan step atau langkah yang ditampilkan papan beserta dengan message yang berupa mobil apa gerak ke mana untuk mempermudah proses analisis pada solusi. Pada bagian kiri atas, terdapat tombol untuk mengembalikan interface ke input. Sedangkan, pada bagian bawah terdapat detail hasil pemrosesan seperti waktu eksekusi, jumlah node yang dikunjungi, algoritma, heuristik, dan total langkah solusi yang ditampilkan. Untuk luaran, tidak merupakan output yang memainkan sendiri, namun dapat dikontrol oleh pengguna untuk mempermudah proses analisis solusi yang dikeluarkan.

VII. Lampiran

Tautan repositori: https://github.com/Staryo40/Tucil3_13523100

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	✓	