

LAPORAN TUGAS KECIL 3

IF2211 Strategi Algoritma

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding



Disusun oleh:

Nathanael Rachmat (13523142)

Andrew Tedjapratama (13523148)

**PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG**

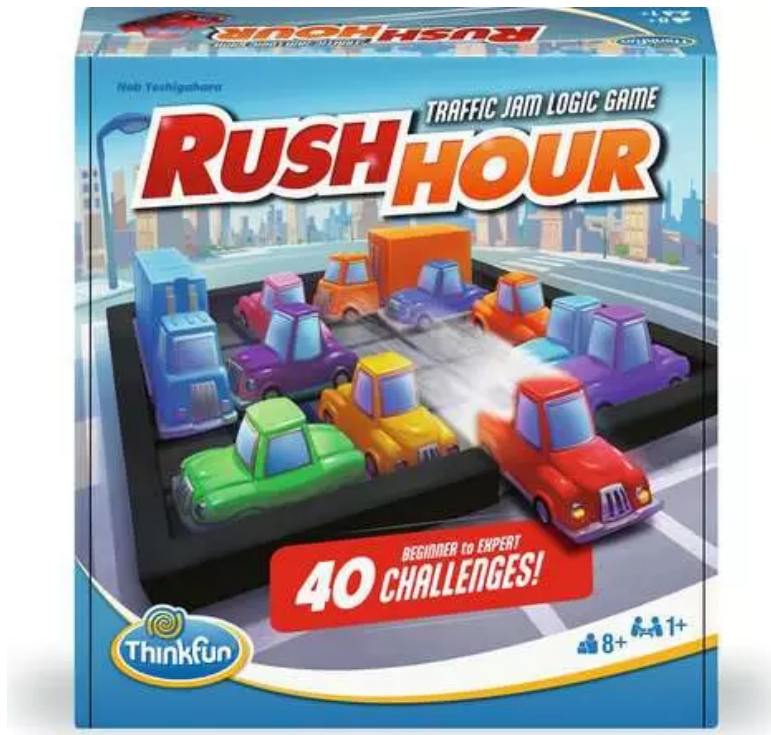
2025

DAFTAR ISI

DAFTAR ISI.....	1
BAB I.....	2
BAB II.....	5
2.1 Input.....	5
2.2 Output.....	6
BAB III.....	7
3.1 Overview.....	7
3.2 Fungsi Evaluasi $f(n)$, $g(n)$, dan $h(n)$	7
3.3 Heuristik dan Admissibility.....	7
3.4 Uniform Cost Search (UCS).....	8
3.4.1 Perbandingan UCS dengan BFS (Breadth-First Search).....	9
3.5 Greedy Best First Search (GBFS).....	10
3.6 A* Search.....	11
3.6.1 Keunggulan dan Optimalitas A* dibanding UCS.....	12
3.7 Kompleksitas Algoritma.....	13
BAB IV.....	15
4.1. Uniform Cost Search (UCS).....	15
4.2. Metode Successors.....	17
4.3. Greedy Best-First Search (GBFS).....	19
4.4. A* Search (A-Star).....	21
4.4. Heuristik yang Dipakai.....	22
4.4.1 Heuristik Jarak (Distance Heuristic).....	22
4.4.2 Heuristik Jumlah Penghalang (Blocking Car Count).....	23
4.4.2 Heuristik Composite.....	23
BAB V.....	24
5.1 Implementasi Heuristik Alternatif.....	24
5.1.1 Heuristik Blocker (Blocking Car Count).....	24
5.1.2 Heuristik Composite (Distance + Blocking Car Count).....	25
5.2 Graphical User Interface.....	26
BAB VI.....	28
6.1 Konfigurasi dan Board Input.....	28
6.2 Data Pengujian.....	29
6.3 Analisis Hasil Pengujian.....	30
BAB VII.....	33
BAB VIII.....	34

BAB I

DESKRIPSI MASALAH



Gambar 1.1 Rush Hour Puzzle

(Sumber: <https://www.thinkfun.com/en-US/products/educational-games/rush-hour-76582>)

Dalam ranah permainan logika dan algoritma, pencarian solusi optimal untuk sebuah masalah merupakan tantangan yang fundamental dan menarik. Salah satu permainan puzzle klasik yang menguji kemampuan perencanaan strategis dan penyelesaian masalah adalah 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.

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.

Tugas Kecil 3 dari mata kuliah Strategi Algoritma ini meminta sebuah penyelesaian masalah dengan merancang dan mengimplementasikan program penyelesaian puzzle Rush Hour menggunakan berbagai algoritma *pathfinding*. Program ini dirancang untuk membaca konfigurasi awal papan Rush Hour dari sebuah file teks ataupun input *graphical* melalui GUI, yang mencakup dimensi papan, jumlah dan posisi kendaraan, serta letak pintu keluar. Pengguna kemudian dapat memilih algoritma pencarian spesifik yang akan digunakan, antara lain Greedy Best First Search, Uniform Cost Search (UCS), dan A*, untuk menemukan urutan pergerakan yang valid hingga mobil utama berhasil keluar.

Algoritma pathfinding yang diterapkan akan bekerja dengan cara membangun dan menjelajahi sebuah graf keadaan, di mana setiap simpul (*node*) dalam graf merepresentasikan satu *state* atau konfigurasi papan permainan pada setiap pergeseran, dan setiap sisi (*edge*) merepresentasikan satu langkah pergeseran kendaraan yang valid. Algoritma seperti *Uninformed Cost Search* (UCS) akan berfokus pada penemuan solusi dengan total biaya langkah terendah, sementara algoritma *A** dan *Greedy Best-First Search* (GBFS) akan memanfaatkan fungsi heuristik untuk optimasi pencarian.

Program akan menghasilkan output berupa urutan langkah solusi, visualisasi konfigurasi papan pada setiap tahap pergerakan, jumlah total, serta total waktu eksekusi yang dibutuhkan untuk menemukan solusi. Tugas ini juga menawarkan berbagai implementasi bonus, seperti penambahan algoritma pathfinding alternatif, implementasi dua atau lebih fungsi heuristik yang dapat dipilih pengguna, dan pembuatan Antarmuka Pengguna Grafis (*Graphical User Interface/GUI*) untuk memvisualisasikan proses penyelesaian puzzle secara animasi.

BAB II

SPESIFIKASI TUGAS

Buatlah program sederhana dalam bahasa C/C++/Java/Javascript yang mengimplementasikan algoritma pathfinding Greedy Best First Search, UCS (Uniform Cost Search), dan A* dalam menyelesaikan permainan Rush Hour. Algoritma pathfinding minimal menggunakan satu heuristic (2 atau lebih jika mengerjakan bonus) yang ditentukan sendiri. Jika mengerjakan bonus, heuristic yang digunakan ditentukan berdasarkan input pengguna. Algoritma dijalankan secara terpisah. Algoritma yang digunakan ditentukan berdasarkan Input pengguna.

2.1 Input

Sebagai input masukan, program akan membaca dan menerima:

1. Konfigurasi permainan/*test case* dalam format ekstensi .txt. File *test case* tersebut berisi:
 - Dimensi Papan terdiri atas dua buah variabel A dan B yang membentuk papan berdimensi A x B
 - Banyak piece BUKAN primary piece direpresentasikan oleh variabel integer N.
 - Konfigurasi papan yang mencakup penempatan piece dan primary piece, serta lokasi pintu keluar. Primary Piece dilambangkan dengan huruf P dan pintu keluar dilambangkan dengan huruf K. Piece dilambangkan dengan huruf dan karakter selain P dan K, dan huruf/karakter berbeda melambangkan piece yang berbeda. Cell kosong dilambangkan dengan karakter '.' (titik). (Catatan: ingat bahwa pintu keluar pasti berada di dinding papan dan sejajar dengan orientasi primary piece)

File .txt yang akan dibaca memiliki format sebagai berikut:

```
A B
N
konfigurasi_papan
```

Contoh Input

```
6 6
```

```

11
AAB..F
..BCDF
GPCDFK
GH.III
GHJ...
LLJMM.

```

keterangan: “K” adalah pintu keluar, “P” adalah primary piece, Titik (“.”) adalah cell kosong.

Contoh konfigurasi papan lain yang mungkin berdasarkan letak pintu keluar (X adalah piece/cell random)

K	XXX	XXX
XXX	KXXX	XXX
XXX	XXX	XXX
XXX		K

2. Algoritma pathfinding yang digunakan
3. Heuristic yang digunakan (bonus)

2.2 Output

Setelah program dijalankan, program akan mengeluarkan output berupa:

1. Banyaknya gerakan yang diperiksa (alias banyak ‘node’ yang dikunjungi)
2. Waktu eksekusi program
3. Konfigurasi papan pada setiap tahap pergerakan/pergeseran.

BAB III

TEORI SINGKAT

3.1 Overview

Untuk pencarian solusi dalam permasalahan permainan Rush Hour, digunakan tiga algoritma utama dalam strategi pencarian jalur optimal: **Uniform Cost Search (UCS)**, **Greedy Best First Search (GBFS)**, dan **A* Search**. UCS termasuk dalam kategori *blind search*, sementara GBFS dan A* Search termasuk dalam algoritma pencarian dengan informasi atau pengetahuan khusus atau disebut juga dengan istilah *informed search* atau *heuristic search*. Informasi ini membantu algoritma untuk lebih efisien dalam menemukan solusi, terkadang lebih cepat dibanding pencarian yang tidak menggunakan informasi tambahan atau *blind search*. Dalam bab teori singkat ini, akan dianalisis perbedaan algoritma UCS, GBFS, dan A* Search.

3.2 Fungsi Evaluasi $f(n)$, $g(n)$, dan $h(n)$

Dalam algoritma pencarian jalur (*pathfinding*), keputusan untuk mengeksplorasi suatu node ditentukan oleh fungsi evaluasi. Bagian-bagian dan alat yang dipakai dalam fungsi evaluasi meliputi:

- **$g(n)$** : total biaya yang dibutuhkan dari start node menuju node n , atau dapat dibilang biaya sejauh ini untuk mencapai titik n . $g(n)$ sangat diandalkan untuk algoritma UCS dan A* Search.
- **$h(n)$** : estimasi biaya dari n ke tujuan, dihitung menggunakan heuristik, sehingga dapat dibilang seberapa jauh lagi ke tujuan menurut dugaan kita. Algoritma GBFS dan A* sangat mengandalkan ini karena memanfaatkan heuristik yang ditentukan secara logis.
- **$f(n)$** : estimasi total biaya jalur melalui titik n menuju tujuan. Fungsi evaluasi heuristik ini digunakan untuk menentukan prioritas eksplorasi suatu node. $f(n)$ merupakan kombinasi dari $g(n)$ dan $h(n)$, tergantung algoritmanya yang akan dibahas lebih lanjut di sub bab berikutnya.

3.3 Heuristik dan *Admissibility*

Dalam algoritma pencarian heuristik seperti GBFS dan A*, peran heuristik sangat krusial dalam menentukan efisiensi dan kualitas solusi yang dihasilkan. Heuristik ($h(n)$) adalah fungsi estimasi

biaya dari suatu node n menuju goal. Keakuratan dan karakteristik dari heuristik inilah yang menentukan apakah solusi yang ditemukan akan optimal atau tidak.

Heuristik dapat dikatakan *admissible* jika untuk setiap node, nilai heuristik $h(n)$ tidak pernah melebihi biaya aktual dari node tersebut ke goal (disebut $h^*(n)$), maka:

$$h(n) \leq h^*(n), \text{ untuk semua } n$$

Dalam istilah lain, heuristik *admissible* itu bisa dibilang optimis tapi realistis, sehingga dapat dibilang heuristik *admissible* harus memperkirakan sisa biaya dari sekarang ke goal, bukan dari awal atau campur-campur yang bikin perhitungannya dobel. Dalam konteks pencarian solusi di Rush Hour, contoh heuristik *admissible* dapat berupa jumlah kotak kosong di depan *primary piece* menuju pintu keluar, atau jumlah kendaraan yang menghalangi jalur *primary piece*.

3.4 Uniform Cost Search (UCS)

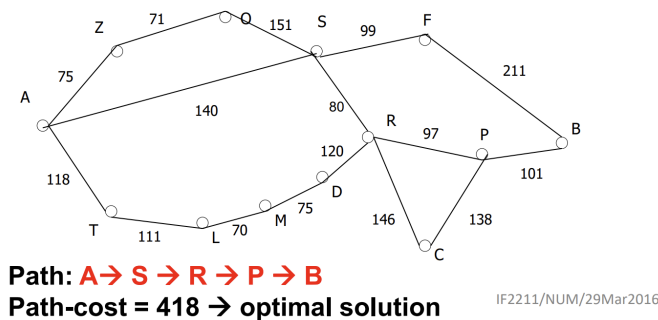
Uniform Cost Search (UCS) pada dasarnya merupakan versi khusus dari algoritma Dijkstra yang tidak menggunakan heuristik, di mana pencarian dilakukan dengan menelusuri node berdasarkan biaya total terkecil dari node awal ke node goal pada sebuah graf berbobot. Dengan cara ini, UCS menjamin bahwa ketika mencapai node goal, jalur yang ditemukan adalah jalur dengan biaya aktual paling rendah tanpa memperkirakan posisi goal. Oleh karena itu, solusi dari UCS selalu optimal karena solusi yang lebih murah pasti sudah ditemukan.

Fungsi evaluasi yang digunakan oleh UCS adalah:

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

Di mana:

- $g(n)$ menyatakan total biaya aktual dari start node menuju node n .
- Fungsi evaluasi hanya melihat $g(n)$ dan mengabaikan $h(n)$ ($h(n) = 0$), di sebabkan tidak dipakainya heuristik sama sekali pada algoritma ini.



Simpul-E	Simpul Hidup
A	$Z_{A-75}, T_{A-118}, S_{A-140}$
Z_{A-75}	$T_{A-118}, S_{A-140}, O_{A2-146}$
T_{A-118}	$S_{A-140}, O_{A2-146}, L_{AT-229}$
S_{A-140}	$O_{A2-146}, R_{AS-220}, L_{AT-229}, F_{AS-239}, O_{AS-291}$
O_{A2-146}	$R_{AS-220}, L_{AT-229}, F_{AS-239}, O_{AS-291}$
R_{AS-220}	$L_{AT-229}, F_{AS-239}, O_{AS-291}, P_{ASR-317}, D_{ASR-340}, C_{ASR-366}$
L_{AT-229}	$F_{AS-239}, O_{AS-291}, M_{ATL-299}, P_{ASR-317}, D_{ASR-340}, C_{ASR-366}$
F_{AS-239}	$O_{AS-291}, M_{ATL-299}, P_{ASR-317}, D_{ASR-340}, C_{ASR-366}, B_{ASF-450}$
O_{AS-291}	$M_{ATL-299}, P_{ASR-317}, D_{ASR-340}, C_{ASR-366}, B_{ASF-450}$
$M_{ATL-299}$	$P_{ASR-317}, D_{ASR-340}, D_{ATLM-364}, C_{ASR-366}, B_{ASF-450}$
$P_{ASR-317}$	$D_{ASR-340}, D_{ATLM-364}, C_{ASR-366}, B_{ASRP-418}, C_{ASRP-455}, B_{ASF-450}$
$D_{ASR-340}$	$D_{ATLM-364}, C_{ASR-366}, B_{ASRP-418}, C_{ASRP-455}, B_{ASF-450}$
$D_{ATLM-364}$	$C_{ASR-366}, B_{ASRP-418}, C_{ASRP-455}, B_{ASF-450}$
$C_{ASR-366}$	$B_{ASRP-418}, C_{ASRP-455}, B_{ASF-450}$

Gambar 3.2 Ilustrasi *Uniform Cost Search*

(Sumber: [Informatika Rinaldi Munir - Route Planning Part 1](#))

3.4.1 Perbandingan UCS dengan BFS (*Breadth-First Search*)

Pada kasus penyelesaian Rush Hour, sesuai implementasi program kami dan dari spesifikasi, ditetapkan bahwa biaya untuk penelusuran node dihitung per gerakan dan independen dari jarak. Artinya setiap step penelusuran node atau *edge* dalam UCS memiliki biaya yang tetap. Sebagai contoh, mobil A bisa bergerak 1, 2, atau lebih blok sekaligus dalam sekali jalan, sehingga dihitung sebagai 1 biaya, dan UCS akan melihat itu sebagai satu aksi yang lebih murah.

Hal ini mirip seperti BFS, karena dalam BFS, penelusuran dilakukan secara bertahap dengan eksplorasi node berdasarkan kedalaman atau *depth*. Dalam BFS, setiap langkah (*edge*) dianggap biayanya sama, dan akan menemukan solusi dengan jumlah langkah paling sedikit. Meskipun berbeda dengan UCS, di mana BFS tidak melihat biaya total minimum, saat semua biaya dianggap 1 maka keduanya bisa dibilang saling bertepatan.

Kedua algoritma tentunya beda secara teori dan struktur, di mana BFS memakai *queue* (antrian biasa) dan UCS memakai *priority queue* (antrian prioritas biaya), namun keduanya tetap berperilaku sama pada hal ini karena semua node memiliki prioritas yang sama. Sebagai kesimpulan, UCS akan sama dengan BFS dalam urutan *node* dan *path* jika semua aksi memiliki biaya yang konstan atau tetap, di mana seperti yang diimplementasikan pada program ini.

3.5 Greedy Best First Search (GBFS)

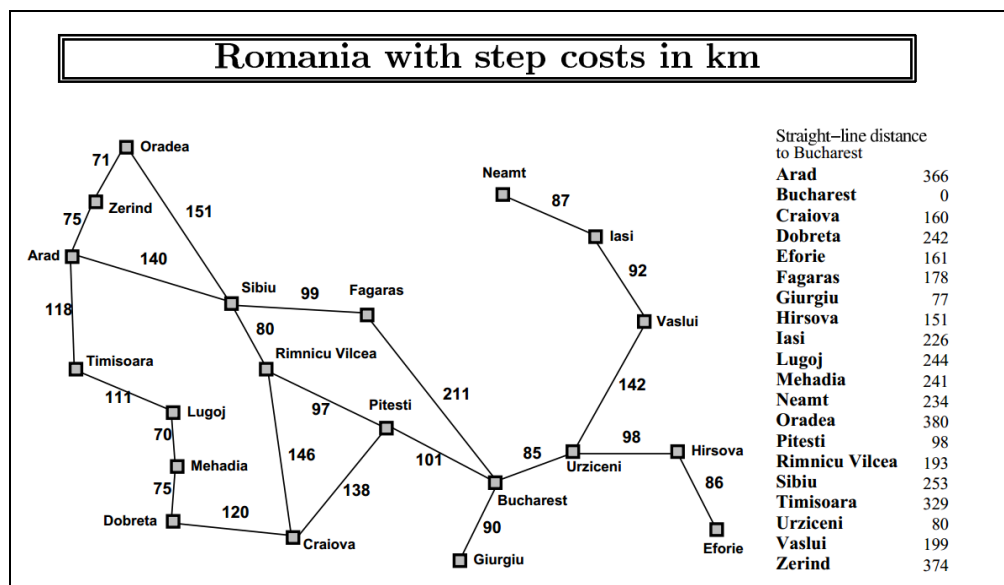
Greedy Best First Search (GBFS) merupakan salah satu algoritma pencarian heuristik yang berfokus pada eksplorasi node berdasarkan seberapa dekat node tersebut ke goal, dengan tambahan menurut perkiraan heuristik. Tidak seperti UCS yang mempertimbangkan biaya total dari awal ($g(n)$), GBFS mengabaikan biaya langkah sebelumnya dan hanya mempertimbangkan estimasi sisa perjalanan, yaitu fungsi heuristik $h(n)$.

Fungsi evaluasi yang digunakan oleh GBFS adalah:

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

Di mana:

- $h(n)$ menyatakan estimasi biaya dari node n ke goal berdasarkan fungsi heuristik
- Fungsi evaluasi mengabaikan $g(n)$, dengan arti dia tidak peduli seberapa mahal jalur yang sudah ditempuh, asalkan terlihat dekat melalui heuristik dengan goal..



Gambar 3.1 Ilustrasi *Greedy Best First Search* untuk tujuan Bucharest, berisi map titik-titik tempat di Romania dengan jaraknya dalam km

(Sumber: [Informatika Rinaldi Munir - Route Planning Part 2](#))

Secara teoritis, GBFS tidak menjamin solusi optimal karena biaya yang riil ($g(n)$) tidak diperhitungkan dan hanya melihat estimasi heuristik atau jalur yang terlihat “dekat” ke goal. Jadi, jalur yang terlihat dekat tersebut bisa jadi sebenarnya panjang atau mahal. Jika heuristiknya

terlalu optimis atau tidak mempertimbangan *obstacle* yang ada, maka GBFS bisa salah arah atau lebih lama.

Misalnya, jika heuristik yang dipakai adalah berdasarkan kedekatan *primary piece* ke pintu keluar, *primary piece* dapat tampak dekat ke pintu keluar meskipun terhalang oleh 3 mobil besar. GBFS mungkin tetap mengejar itu tanpa menyadari membutuhkan banyak langkah yang riil untuk membebaskannya, padahal secara teori bisa ada jalur lain dengan $h(n)$ lebih tinggi, namun dengan langkah riil lebih sedikit. Sebagai kesimpulan, GBFS tidak terjamin optimal dalam pencarian solusi karena terlalu bergantung pada heuristik dan mengabaikan langkah riil atau biaya aktual, tetapi biasanya lebih cepat dalam waktu eksplorasi karena langsung "ngejar" tujuan.dengan memanfaatkan heuristik yang kelihatan menjanjikan..

3.6 A* Search

A* Search merupakan algoritma pencarian yang menggabungkan keunggulan dari UCS (yang mempertimbangkan biaya langkah aktual) dan GBFS (yang memanfaatkan estimasi heuristik). A* menjadi salah satu algoritma yang paling populer dan efektif dalam berbagai aplikasi pathfinding karena kemampuannya untuk menemukan solusi secara efisien sekaligus optimal, selama heuristik yang digunakan bersifat *admissible*.

Fungsi evaluasi yang digunakan oleh A* adalah:

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

Di mana:

- $g(n)$ menyatakan total biaya riil dari node awal ke node n.
- $h(n)$ menyatakan estimasi biaya dari node n ke goal, dihitung berdasarkan fungsi heuristik.

optimalitas, di mana optimal dapat didefinisikan sebagai solusi dengan jumlah langkah (atau biaya) terkecil, atau dengan definisi tidak ada solusi yang lebih pendek, A* dan UCS memiliki optimalitas yang sama.

3.7 Kompleksitas Algoritma

Ketiga algoritma yang dipakai dalam proyek ini untuk menyelesaikan puzzle Rush Hour memiliki kompleksitas waktu dan ruang yang berbeda-beda, tergantung strategi eksplorasi node dan pemanfaatan fungsi evaluasi.

UCS menjamin solusi optimal dengan menelusuri semua jalur berdasarkan biaya total ($g(n)$), namun eksplorasi bisa meluas karena tidak ada arah langsung ke goal, sehingga memperhatikan biaya solusi optimal (C) serta biaya langkah minimum (e). GBFS cenderung mengejar node yang tampak “dekat” ke goal berdasarkan heuristik ($h(n)$), tetapi mengabaikan biaya aktual, sehingga dipengaruhi kedalaman maksimum dalam graf (m).

A* menggabungkan kelebihan UCS dan GBFS, dengan mempertimbangkan baik biaya aktual maupun heuristik estimasi ke goal, sehingga biasanya lebih efisien tergantung kualitas dan *admissibility* heuristik yang dipakai, memperhatikan kedalaman solusi optimal (d) dalam graf keadaan.

Ringkasan kompleksitas ruang maupun waktu untuk **Uniform Cost Search (UCS)**, **Greedy Best First Search (GBFS)**, dan **A* Search** dapat dilihat pada tabel berikut:

Tabel 3.7.1 Perbandingan kompleksitas, pemakaian heuristik, dan solusi optimal pada UCS, GBFS, dan A*

Algoritma	Kompleksitas Waktu	Kompleksitas Ruang	Tergantung pada Heuristik	Solusi Optimal
UCS	$O(b^{C/e})$	$O(b^{C/e})$	Tidak (X)	Ya (V)
GBFS	$O(b^m)$	$O(b^m)$	Ya (V)	Tidak (X)
A*	$O(b^d)$	$O(b^d)$	Ya (V)	Ya (V)

Di mana:

- b : *branching factor*, yaitu jumlah rata-rata cabang dari setiap node

- d: kedalaman solusi optimal dalam graf keadaan
- C: biaya solusi optimal dari awal ke goal
- e: biaya minimum dari setiap langkah (*edge*)
- m: kedalaman maksimum dari graf atau tree pencarian

BAB IV

DESKRIPSI ALGORITMA

Terdapat tiga jenis algoritma yang dipakai untuk menyelesaikan permasalahan rush hour. Setiap macam algoritma memiliki cara kerja yang berbeda, dan ada algoritma dengan heuristik yang bisa divariasikan sehingga menghasilkan jawaban *path* yang berbeda. Berikut ini merupakan deskripsi algoritma yang dibuat untuk menentukan solusi dari rush hour.

4.1. Uniform Cost Search (UCS)



```
1 Game.prototype.uniformCostSearch = function () {
2   const start = this.getInitialNode();
3
4   const frontier = [];
5   const push = (n) => {
6     frontier.push(n);
7     frontier.sort((a, b) => a.cost - b.cost);
8   };
9   const pop = () => frontier.shift();
10
11   const visited = new Map();
12
13   // stats
14   let nodeCount = 0;
15   const startTime = Date.now();
16
17   push(start);
18
19   while (frontier.length) {
20     const node = pop();
21     nodeCount++;
22
23     if (this.isGoal(node)) {
24       const endTime = Date.now();
25       let runtime = endTime - startTime;
26       return {
27         path: node.getPath(),
28         nodePath: node.getNodePath(),
29         runtime,
30         nodeCount,
31       };
32     }
33
34     const key = serializeState(node.state);
35     if (visited.has(key) && visited.get(key) <= node.cost) continue;
36     visited.set(key, node.cost);
37
38     for (const child of this.successors(node)) push(child);
39   }
40
41   return null;
42 };
```

Gambar 4.1.1 Algoritma UCS

Algoritma UCS dimulai dengan mendapatkan node awal dan mendefinisikan prioqueue (frontier). Fungsi push dan pop didefinisikan untuk push frontier secara terurut dan pop dari frontier. Kemudian didefinisikan sebuah map bernama visited agar menyimpan daftar

konfigurasi papan (state) dan biaya terkecil akan dikunjungi nantinya. Hal ini mencegah algoritma untuk kembali ke state yang pernah dikunjungi. Node count dan startTime digunakan untuk menghitung jumlah node yang dikunjungi dan lama waktunya. Push node pertama ke frontier dilakukan sebagai node pertama yang dikunjungi dalam queue frontier.

Setelah konfigurasi selesai, sekarang algoritma masuk ke bagian *looping* utamanya. Setiap iterasinya, node yang dikunjungi didapatkan dari pop() frontier, dan nodeCount bertambah agar mencatat jumlah node yang dikunjungi. Kalau state papan dari node sekarang itu bukan goal (bersebelahan dengan exit), maka program akan membuat representasi unik dari konfigurasi papan saat ini agar bisa digunakan sebagai key dari map visited (agar tidak dikunjungi kembali nantinya). Setelah itu program mengecek apakah konfigurasi sudah pernah dikunjungi dengan biaya lebih murah atau sama? Kalau iya, simpul ini diabaikan, karena UCS memilih jalur dengan cost paling kecil. Kalau tidak, program lanjut menyimpan konfigurasi papan ke visited bersama dengan biaya untuk mencapainya.

Agar pencarian node lanjut, program push setiap anak dari node ini ke frontier. Penentuan anak tersebut dicapai dengan metode successors yang akan dijelaskan di bagian berikut ini. Selama program memiliki anak yang valid, belum mencapai tujuan, atau visited state-nya belum dikunjungi atau memiliki state dengan biaya yang lebih kecil, program akan terus diiterasikan hingga:

1. Board mencapai state tujuan
2. Tidak ada node yang dapat diekspansi lagi.

Ketika node mencapai state tujuan, program akan menghitung runtime dan mengembalikan path, nodepath, runtime, dan nodecount. Namun, program hanya akan mengembalikan null jika tidak dapat diekspansi dan belum mencapai state tujuan.

4.2. Metode Successors

```
1  Game.prototype.successors = function (node) {
2    const next = [];
3    const grid = this._rebuildBoard(node.state);
4
5    for (const [sym, meta] of this.pieceMap.entries()) {
6      const { length, orientation } = meta;
7      const { row, col } = node.state.get(sym);
8
9      for (const dir of [-1, 1]) {
10       let dist = 1;
11       while (true) {
12         const r = orientation === "H" ? row : row + dir * dist;
13         const c = orientation === "H" ? col + dir * dist : col;
14
15         let checkR;
16         if (orientation === "H") {
17           checkR = row;
18         } else {
19           // V
20           if (dir === 1) {
21             checkR = row + length - 1 + dist;
22           } else {
23             checkR = row - dist;
24           }
25         }
26
27         let checkC;
28         if (orientation === "V") {
29           checkC = col;
30         } else {
31           // H
32           if (dir === 1) {
33             checkC = col + length - 1 + dist;
34           } else {
35             checkC = col - dist;
36           }
37         }
38
39         // break kalo nabrak
40         if (!this._isEmpty(grid, checkR, checkC)) {
41           break;
42         }
43
44         const newState = new Map(node.state);
45         newState.set(sym, { row: r, col: c });
46         next.push(
47           new Node(
48             newState,
49             node,
50             { piece: sym, dir, count: dist },
51             node.cost + 1
52           )
53         );
54         dist++;
55       }
56     }
57   }
58   return next;
59 };
60
```

Gambar 4.2.1 Metode Successors

Tujuan utama metode ini adalah untuk menghasilkan semua konfigurasi anak yang dapat dicapai dari langkah yang valid dari posisi saat ini. Metode ini dimulai dengan mendeklarasikan array `next` untuk menyimpan semua konfigurasi anak (next states) yang valid. Kemudian, program akan merekonstruksi papan permainan dari state yang sedang diperiksa menggunakan `rebuildBoard(node.state)` sehingga bisa digunakan untuk pengecekan apakah suatu posisi kosong atau tidak.

Metode `rebuildBoard` membangun ulang papan permainan (grid) berdasarkan state saat ini. Papan ini direpresentasikan sebagai array 2D yang diisi ulang dari posisi tiap kendaraan yang ada di state. Untuk setiap kendaraan, program akan menandai kotak-kotak yang ditempati oleh simbol kendaraan tersebut berdasarkan panjang dan orientasinya (horizontal atau vertikal). Papan ini dipakai untuk pengecekan posisi kosong atau tabrakan.

Setelah itu, program akan melakukan iterasi terhadap setiap kendaraan yang terdapat di papan permainan. Untuk tiap kendaraan (`sym`), informasi seperti panjang kendaraan (`length`) dan orientasi kendaraan (`orientation`) diambil dari `pieceMap`, sedangkan posisi kendaraan (`row`, `col`) diambil dari `node.state`.

Program kemudian mencoba menggerakkan kendaraan ke dua arah, yaitu -1 (ke atas atau kiri) dan 1 (ke bawah atau kanan), tergantung dari orientasinya. Untuk setiap arah tersebut, program melakukan pencarian sejauh mungkin selama langkah tersebut masih valid (tidak menabrak kendaraan lain atau keluar dari papan).

Pada setiap percobaan pergerakan, program menghitung posisi baru kendaraan:

- Jika kendaraan horizontal, maka posisi baris tetap dan kolom digeser.
- Jika kendaraan vertikal, maka posisi kolom tetap dan baris digeser.

Namun sebelum menganggap langkah itu valid, program juga menghitung posisi `checkR` dan `checkC`, yaitu posisi yang akan ditempati bagian ujung kendaraan setelah digeser. Posisi ini dicek menggunakan `_isEmpty(grid, checkR, checkC)` untuk memastikan tidak menabrak kendaraan lain. Jika tidak kosong, maka loop `while` akan dihentikan dan kendaraan tidak akan digerakkan lebih jauh ke arah itu.

Jika ruang tersebut kosong, maka program akan membuat `newState` berupa salinan dari state lama. Posisi kendaraan diubah ke posisi baru, dan konfigurasi baru tersebut dikemas ke dalam sebuah Node baru yang menyimpan:

- State baru setelah pergerakan
- Node sebelumnya (`parent`)
- Aksi yang dilakukan (siapa yang bergerak, arah, dan sejauh apa)
- Biaya total yang sudah dikeluarkan (`biaya parent + 1`)

Node baru ini dimasukkan ke dalam next, dan pergerakan selanjutnya ($\text{dist} \pm 1$) dicoba lagi dalam arah yang sama.

Setelah semua kendaraan dan arah telah dicoba, serta semua konfigurasi yang valid sudah dihitung, maka fungsi ini mengembalikan array next yang berisi semua anak dari node saat ini. Nilai ini digunakan dalam algoritma UCS, GBFS, dan A* untuk melanjutkan eksplorasi solusi rush hour.

4.3. Greedy Best-First Search (GBFS)

```
1  Game.prototype.greedyBestFirstSearch = function () {
2    const start = this.getInitialNode();
3
4    const frontier = [];
5    const push = (n) => {
6      frontier.push(n);
7      frontier.sort(
8        (a, b) => this.heuristicFn(a.state) - this.heuristicFn(b.state)
9      );
10   };
11   const pop = () => frontier.shift();
12
13   const visited = new Set();
14   // stats
15   let nodeCount = 0;
16   const startTime = Date.now();
17
18   push(start);
19
20   while (frontier.length) {
21     const node = pop();
22     nodeCount++;
23
24     if (this.isGoal(node)) {
25       const endTime = Date.now();
26       let runtime = endTime - startTime;
27       return {
28         path: node.getPath(),
29         nodePath: node.getNodePath(),
30         runtime,
31         nodeCount,
32       };
33     }
34
35     const key = serializeState(node.state);
36     if (visited.has(key)) continue;
37     visited.add(key);
38
39     for (const child of this.successors(node)) push(child);
40   }
41
42   return null;
43 };
44
```

Gambar 4.3.1 Algoritma GBFS

Algoritma Greedy Best-First Search (GBFS) dimulai dengan mengambil node awal dari kondisi awal permainan. Frontier didefinisikan sebagai prioritas queue yang akan diisi berdasarkan hasil evaluasi fungsi heuristik terhadap state masing-masing node. Push dan pop digunakan untuk

memasukkan node ke frontier dan mengambil node dengan nilai heuristik terkecil (paling menjanjikan mendekati tujuan). Karena GBFS hanya peduli seberapa dekat ke tujuan dan tidak memperhitungkan cost sejauh ini, sorting hanya dilakukan berdasarkan nilai heuristik.

Sebuah visited set digunakan untuk menyimpan state yang sudah pernah dikunjungi agar tidak diproses ulang, karena GBFS tidak melakukan update cost seperti UCS. nodeCount dan startTime digunakan untuk mencatat jumlah node yang dikunjungi dan lama waktu pencarian. Push pertama dilakukan pada node awal.

Setelah konfigurasi selesai, algoritma masuk ke perulangan utama. Setiap iterasinya, node dengan heuristik terkecil di-pop dari frontier dan jumlah node yang dikunjungi bertambah. Jika node tersebut adalah goal, program akan menghitung runtime dan mengembalikan hasil pencarian. Jika belum, representasi unik dari state sekarang disimpan ke visited agar tidak diproses ulang nantinya, sama seperti UCS. Kemudian, sama seperti UCS juga, semua anak dari node saat ini dihitung melalui successors, dan masing-masing di-*push* ke frontier agar node berikutnya dikunjungi.

Selama masih ada node dalam frontier dan goal belum ditemukan, pencarian akan terus berjalan. Nilai yang dikembalikan ketika mencapai tujuan state sama seperti nilai yang dikembalikan oleh UCS. Pada dasarnya, cara kerja GBFS mirip dengan UCS. Yang membedakan hanya cara menentukan sort dari queue nya saja. Makanya tidak jauh berbeda dengan algoritma yang digunakan oleh UCS.

4.4. A* Search (A-Star)

```
1  Game.prototype.aStarSearch = function () {
2    const start = this.getInitialNode();
3
4    const frontier = [];
5    const push = (n) => {
6      frontier.push(n);
7      frontier.sort(
8        (a, b) =>
9          a.cost +
10           this.heuristicFn(a.state) -
11           (b.cost + this.heuristicFn(b.state))
12      );
13    };
14    const pop = () => frontier.shift();
15
16    const visited = new Map();
17    let nodeCount = 0;
18    const startTime = Date.now();
19
20    push(start);
21
22    while (frontier.length) {
23      const node = pop();
24      nodeCount++;
25
26      if (this.isGoal(node)) {
27        const endTime = Date.now();
28        let runtime = endTime - startTime;
29        return {
30          path: node.getPath(),
31          nodePath: node.getNodePath(),
32          runtime,
33          nodeCount,
34        };
35      }
36
37      const key = serializeState(node.state);
38      if (visited.has(key) && visited.get(key) <= node.cost) continue;
39      visited.set(key, node.cost);
40
41      for (const child of this.successors(node)) push(child);
42    }
43
44    return null;
45  };
```

Gambar 4.4.1 Algoritma A* Search

Algoritma A* Search (A-Star) dimulai dengan mendapatkan node awal, lalu mendefinisikan frontier sebagai queue prioritas. Fungsi push digunakan untuk memasukkan node baru ke frontier sambil mengurutkannya berdasarkan penjumlahan antara biaya dari awal hingga node saat ini (cost) dan hasil dari fungsi heuristik untuk state tersebut. Fungsi pop mengambil node dengan nilai $f(n)$ terkecil dari frontier.

Visited didefinisikan sebagai Map yang menyimpan state beserta cost terkecil yang pernah dicapai untuk sampai ke state tersebut. Ini mencegah algoritma mengunjungi ulang node dengan cost yang lebih besar. Variabel nodeCount dan startTime berfungsi untuk mencatat jumlah node yang dikunjungi dan waktu eksekusinya.

Node awal langsung dipush ke frontier sebagai titik awal pencarian. Setelah setup selesai, algoritma masuk ke loop utama. Di setiap iterasi, node dengan nilai $f(n)$ terkecil di-pop dari frontier dan dihitung sebagai node yang dikunjungi. Jika node tersebut merupakan goal, maka waktu selesai dihitung dan hasil pencarian dikembalikan. Jika belum, state dari node ini diubah menjadi bentuk yang bisa dijadikan key (`serializeState`) dan dicek apakah sudah dikunjungi dengan cost lebih kecil atau sama. Jika belum, state tersebut disimpan ke `visited`, dan semua anak dari node ini dipush ke frontier menggunakan fungsi `successors`.

Selama masih ada node dalam frontier dan goal belum ditemukan, pencarian akan terus berjalan. Nilai yang dikembalikan ketika mencapai state tujuan sama seperti nilai yang dikembalikan oleh UCS, yaitu `path`, `nodePath`, `runtime`, dan `nodeCount`. Pada dasarnya, cara kerja A* mirip dengan UCS karena sama-sama mempertimbangkan cost dan menggunakan `visited map`. Bedanya, A* menambahkan heuristik ke dalam perhitungan sorting pada frontier, sehingga dapat lebih cepat mencapai goal tanpa mengorbankan optimalitas. Karena itulah, struktur algoritmanya tidak jauh berbeda dengan UCS maupun GBFS, hanya berbeda dari segi perhitungan urutan pemrosesan nodenya.

4.4. Heuristik yang Dipakai

Heuristik adalah fungsi yang digunakan untuk memperkirakan seberapa dekat suatu konfigurasi papan dengan tujuan. Dalam algoritma GBFS dan A*, heuristik ini digunakan sebagai dasar penyortiran node di frontier agar algoritma bisa memprioritaskan langkah yang lebih menjanjikan. Pada implementasi Rush Hour ini, terdapat tiga jenis heuristik utama yang digunakan:

4.4.1 Heuristik Jarak (Distance Heuristic)

Heuristik ini menghitung jarak antara ujung kendaraan utama "P" dengan posisi pintu keluar "K", tergantung orientasi kendaraan:

- Jika kendaraan "P" horizontal, maka jarak dihitung dari ujung kanan kendaraan ke kolom pintu keluar.
- Jika vertikal, dihitung dari ujung bawah ke baris pintu keluar.

Heuristik ini bekerja dengan cepat dan sederhana, tapi belum mempertimbangkan adanya kendaraan lain yang menghalangi jalan keluar, sehingga bisa tidak akurat dalam situasi kompleks.

4.4.2 Heuristik Jumlah Penghalang (Blocking Car Count)

Heuristik ini menghitung jumlah kendaraan yang menghalangi jalur kendaraan "P" menuju pintu keluar. Grid papan dibangun ulang, lalu setiap kotak dalam jalur antara ujung kendaraan "P" ke arah pintu keluar diperiksa apakah terisi kendaraan lain. Semakin banyak kendaraan yang menghalangi, semakin besar nilai heuristiknya.

Heuristik ini memberikan informasi yang lebih realistis karena memperhitungkan hambatan yang perlu diatasi, bukan hanya jarak.

4.4.2 Heuristik Composite

Heuristik ini adalah gabungan dari dua heuristik sebelumnya: jarak + jumlah penghalang. Nilai akhir diperoleh dengan menjumlahkan:

- Jarak dari kendaraan "P" ke pintu keluar
- Jumlah kendaraan yang menghalangi jalan keluar

Heuristik ini diharapkan memberikan estimasi yang lebih akurat dan seimbang, karena mempertimbangkan baik seberapa jauh tujuan maupun apa saja yang harus dilalui untuk sampai ke sana.

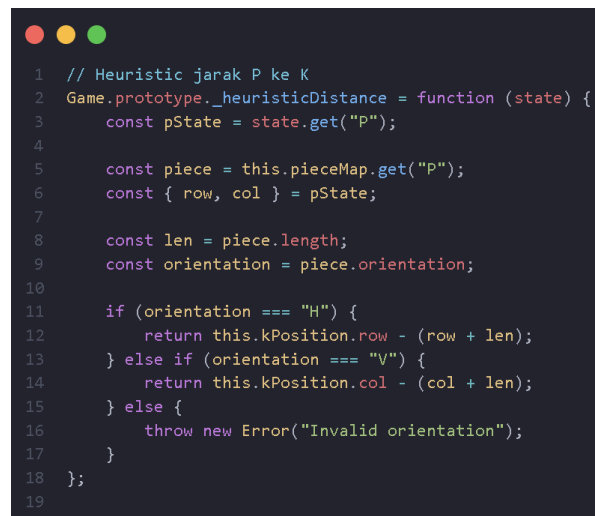
BAB V

IMPLEMENTASI BONUS

Selain spesifikasi wajib, dalam proyek ini tiga fitur bonus diimplementasikan, yaitu implementasi algoritma alternatif tambahan, implementasi heuristik alternatif tambahan, serta Graphical User Interface (GUI).

5.1 Implementasi Heuristik Alternatif

Pada implementasi program, heuristik yang dipakai dalam algoritma A* Search maupun Greedy Best First Search berupa heuristik jarak (*distance heuristic*) dengan mengembalikan jumlah sel di depan *primary piece* sebelum pintu keluar. Sebagai tambahan, dipakai 2 heuristik tambahan yang dipakai sebagai opsi untuk dipakai dengan tujuan dapat lebih jauh mengoptimasi pencarian dengan heuristik yang *admissible* dan informatif.



```
1 // Heuristic jarak P ke K
2 Game.prototype._heuristicDistance = function (state) {
3     const pState = state.get("P");
4
5     const piece = this.pieceMap.get("P");
6     const { row, col } = pState;
7
8     const len = piece.length;
9     const orientation = piece.orientation;
10
11     if (orientation === "H") {
12         return this.kPosition.row - (row + len);
13     } else if (orientation === "V") {
14         return this.kPosition.col - (col + len);
15     } else {
16         throw new Error("Invalid orientation");
17     }
18 };
19
```

Gambar 5.1.1 Fungsi heuristic distance (hardcode)

5.1.1 Heuristik Blocker (*Blocking Car Count*)

Heuristik blocker merupakan heuristik yang dipakai dengan tujuan untuk menghitung jumlah mobil yang menghalangi jalur *primary piece* atau mobil utama dengan ID “P” ke arah pintu keluar.

Untuk implementasi fungsi tersebut, dibuat board ulang berdasarkan state menggunakan `rebuildBoard`, di mana posisi grid yang kosong akan terisi sebagai `null`. Kemudian fungsi ini menyimpan jumlah blockers, atau penghalang dari *primary piece*.

Setelah itu, akan diiterasi untuk mengecek mobil didepannya hingga menunggu K atau exit, dengan mengecek apakah grid di depannya berupa `null`. Jika tidak berupa `null`, artinya ada mobil lain yang menghalangi di depannya sehingga dilakukan *increment* terhadap blockers.

```
1 // Heuristic blocking car count
2 Game.prototype._heuristicBlocker = function (state) {
3   const grid = this._rebuildBoard(state);
4   const pState = state.get("P");
5
6   const piece = this.pieceMap.get("P");
7   const { row, col } = pState;
8   const len = piece.length;
9   const orientation = piece.orientation;
10  let blockers = 0;
11
12  if (orientation === "H") {
13    for (let c = col + len; c <= this.kPosition.col; c++) {
14      if (grid[row][c] !== null) {
15        blockers++;
16      }
17    }
18  } else if (orientation === "V") {
19    for (let r = row + len; r <= this.kPosition.row; r++) {
20      if (grid[r][col] !== null) {
21        blockers++;
22      }
23    }
24  } else {
25    throw new Error("Unknown orientation for P");
26  }
27
28  return blockers;
29 };
```

Gambar 5.1.1.1 Fungsi heuristic blocker

5.1.2 Heuristik Composite (*Distance + Blocking Car Count*)

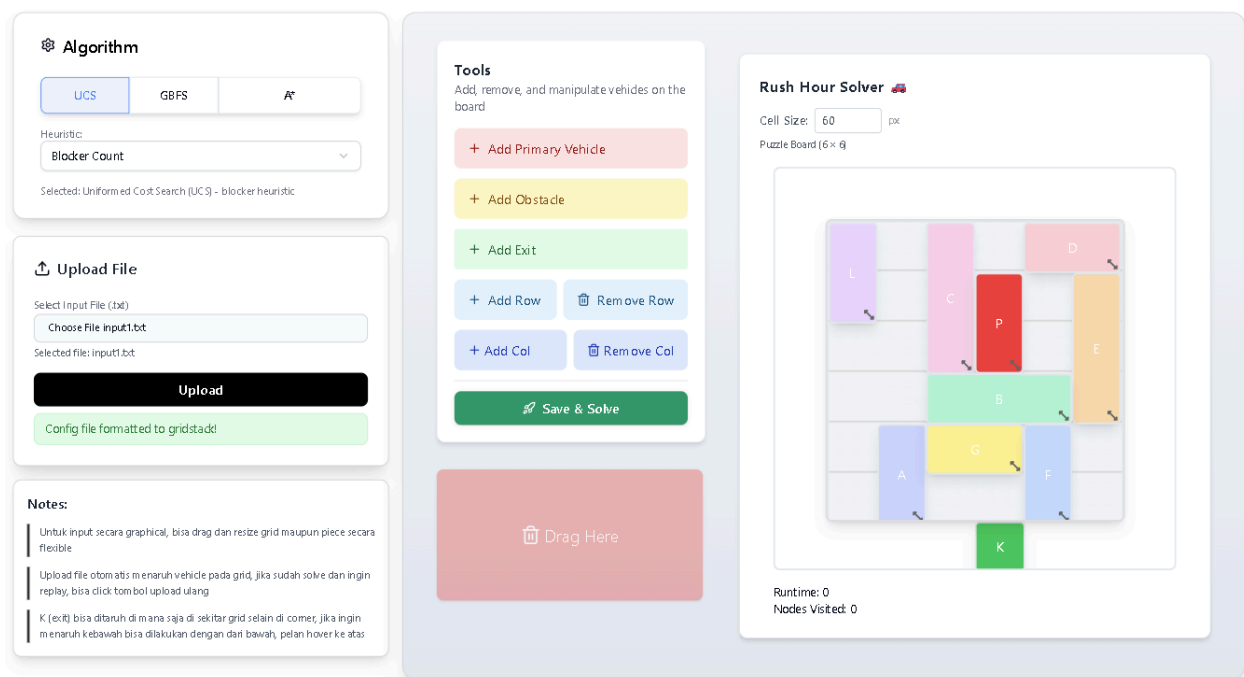
Heuristik composite merupakan heuristik yang menggabungkan dua aspek utama dalam perhitungan estimasi menuju solusi, yaitu jarak *primary piece* dengan ID “P” ke arah pintu keluar maupun jumlah penghalang (blockers) yang ada di jalur tersebut. Oleh karena itu,

heuristik ini diharapkan bisa menghasilkan *composite value* atau nilai dari kedua heuristik tergabung yang dapat lebih mengoptimasi proses pencarian solusi.

```
1 // Composite heuristic (block + distance)
2 Game.prototype._heuristicComposite = function (state) {
3     const grid = this._rebuildBoard(state);
4     const { row, col } = state.get("P");
5     const len = this.pieceMap.get("P").length;
6
7     let distance = this.kPosition.col - (col + len);
8     let blockers = 0;
9     for (let c = col + len; c <= this.kPosition.col; c++) {
10         if (grid[row][c] !== null) {
11             blockers++;
12         }
13     }
14     return distance + blockers;
15 };
```

Gambar 5.1.2.1 Fungsi heuristic composite

5.2 Graphical User Interface



Gambar 5.2.1 Website Rush Hour Puzzle Solver

Fitur GUI bertujuan untuk memvisualisasikan papan puzzle Rush Hour yang terisi oleh blok-blok mobil berwarna. Interface GUI dalam program ini dibuat agar dapat menerima input secara *graphical*, atau dengan kata lain, pengguna dapat menaruh mobil-mobil maupun posisi pintu *exit* secara langsung di GUI. Sebagai keluaran, interface mengeluarkan output berupa animasi gerakan-gerakan dari awal permainan sampai mencapai solusi.

GUI pada program ini dibuat berupa website yang dikembangkan menggunakan framework Next.js dalam bahasa Javascript untuk frontend maupun backend. Untuk fitur input secara *graphical* maupun visualisasi animasi gerakan langkah pencarian, digunakan library Gridstack.js untuk membuat suatu antarmuka UI berbasis grid yang flexible atau bisa disebut *drag-and-drop grid layout system*.

Library Gridstack.js memungkinkan program untuk bisa menambahkan, menghapus, atau mengatur posisi maupun ukuran mobil-mobil langsung dengan cara *drag-and-drop*, dan menunjukkan animasi gerakan-gerakan setiap langkah pencarian. Selain itu, sebagai kreatifitas tambahan, pemanfaatan library Gridstack.js juga memungkinkan pengguna untuk dapat memindahkan dan *me-resize* mobil secara fleksible dan bermain secara langsung. Oleh karena itu, GUI berbasis web ini sangat bermanfaat dan memperbolehkan antarmuka yang simple dan mudah untuk digunakan.

BAB VI

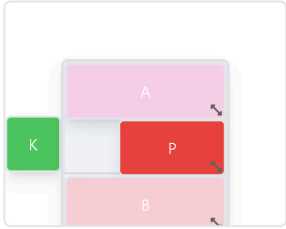
PENGUJIAN DAN ANALISIS

(Link seluruh input dan solusi per langkah data uji terdapat pada folder /test dalam repository)

6.1 Konfigurasi dan Board Input

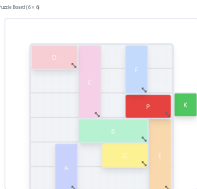

Tabel 6.1 Konfigurasi dan Board Input

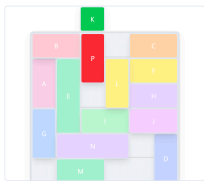
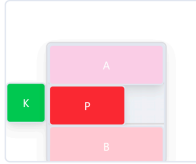
No	Input Config (.txt)	Input Board
1	<pre> 6 6 4 ..C.DD ..C...E PPC...EK .ABBBE .AGGF.F. </pre>	
2	<pre> 6 6 9 K .DD.F. .BB.F. CCC... ..P...E .APGGE .A...E </pre>	
3	<pre> 6 6 13 K ABBCCD AEPFFD GEP.HH GEIIJJ NNNL.. ...LMM </pre>	

4	<pre> 3 3 2 AAA K.PP BBB </pre>	
---	---------------------------------	--

6.2 Data Pengujian

Tabel 6.2 Data hasil pengujian 4 papan dengan perbedaan algoritma dan heuristik yang dipakai

Input			Output			
No	Board Config (.txt)	Heuristic	Output Board	UCS (Uniform Cost Search)	GBFS (Greedy Best First Search)	A* Search
1	<pre> 6 6 4 ..C.DD ..C..E PPC..EK .ABBBE .AGGF.F. </pre>	Distance		Runtime: 61ms Nodes Visited: 756 Move Count: 15	Runtime: 65ms Nodes Visited: 756 Move Count: 15	Runtime: 58ms Nodes Visited: 756 Move Count: 15
		Blocking Cars				
		Composite				
2	<pre> 6 6 9 K .DD.F. .BB.F. CCC... ..P..E .APGGE .A...E </pre>	Distance		Runtime: 47ms Nodes Visited: 473 Move Count: 4	Runtime: 51ms Nodes Visited: 473 Move Count: 4	Runtime: 47ms Nodes Visited: 473 Move Count: 4
		Blocking Cars			Runtime: 45ms Nodes Visited: 473	Runtime: 44ms Nodes Visited: 473 Move Count: 4

					Move Count: 4	
		Composite			Runtime: 49ms Nodes Visited: 473 Move Count: 4	Runtime: 51ms Nodes Visited: 473 Move Count: 4
3	6 6 13 K ABBCCD AEPFFD GEP.HH GEIIJJ NNNL.. ...LMM	Distance		Runtime: 101ms Nodes Visited: 798 Move Count: 22	Runtime: 89 Nodes Visited: 798 Move Count: 22	Runtime: 86 Nodes Visited: 798 Move Count: 22
		Blocking Cars			Runtime: 80 Nodes Visited: 798 Move Count: 22	Runtime: 69 Nodes Visited: 798 Move Count: 22
		Composite			Runtime: 71 Nodes Visited: 798 Move Count: 22	Runtime: 115 Nodes Visited: 798 Move Count: 22
4	3 3 2 AAA K.PP BBB	Distance		Runtime: 2 ms Nodes Visited: 4 Move Count: 1	Runtime: 0 Nodes Visited: 4 Move Count: 1	Runtime: 0 Nodes Visited: 4 Move Count: 1
		Blocking Cars			Runtime: 1 Nodes Visited: 4 Move Count: 1	Runtime: 0 Nodes Visited: 4 Move Count: 1
		Composite			Runtime: 0 Nodes Visited: 4 Move Count: 1	Runtime: 0 Nodes Visited: 4 Move Count: 1

6.3 Analisis Hasil Pengujian

Berdasarkan hasil percobaan yang telah ditinjau, *move count* dan *nodes visited* menjadi salah satu komponen pengujian yang mencolok, di mana sebagian besar sama antara ketiga algoritma dan ketiga heuristik. Semua algoritma menemukan solusi yang optimal, dan heuristik tidak berdampak pada jumlah simpul yang dikunjungi. Perbedaan antara output hasil percobaan dalam satu *test case* hanya berada dalam *runtime*-nya. Hal ini terjadi karena beberapa alasan. Test case

yang kurang memberi variasi langkah valid dalam pergerakannya (kompleks), dan penentuan biaya dalam UCS.

Saat ini, test case yang digunakan untuk percobaan tidak menyediakan kasus kompleks yang mendukung pencarian node dalam jumlah yang banyak. Hal ini ditandai dengan jumlah langkah solusi (*move count*) yang relatif kecil dan struktur papan yang cenderung linier atau tidak terlalu memaksa eksplorasi cabang-cabang alternatif.

Penentuan biaya pada UCS saat ini belum memiliki implementasi yang merepresentasikan jarak pergerakan aktual kendaraan karena setiap langkah biaya nya ditandai dengan 1. Baik hanya bergerak satu langkah, maupun bergerak lebih dari satu langkah, biaya bernilai satu. Hal ini membuat UCS, A*, dan bahkan GBFS cenderung mengikuti pola pencarian yang sama, yang terjadi karena rumus prioritas biaya yang identik.

Jika biaya tidak berpengaruh terhadap prioritas pencarian node, seharusnya heuristik memberi perbedaan. Tapi dari percobaan antar heuristik, output yang dihasilkan juga sama. Hal ini menunjukkan bahwa heuristik yang digunakan saat ini belum cukup informatif atau diskriminatif untuk membedakan prioritas antar simpul dalam pencarian. Beberapa alasan yang menyebabkan ini adalah nilai heuristik yang terlalu kecil dan seragam (nilai *distance* dan *blocking cars* mirip antar simpul), heuristik yang hanya penjumlahan sederhana, dan tidak adanya normalisasi (pembatasan nilai negatif). Akibatnya, semua heuristik gagal memberikan pengaruh terhadap urutan eksplorasi simpul.

Untuk memperbaiki hasil percobaan dan sebagai evaluasi teknis, beberapa hal dapat diperbaiki. Pertama, biaya gerak harus mencerminkan jarak pergerakan kendaraan. Saat ini, semua gerakan dikenai biaya tetap satu, sehingga UCS dan A* tidak bisa membedakan jalur yang lebih efisien. Kedua, nilai heuristik perlu dinormalisasi agar tidak negatif, karena nilai negatif dapat mengganggu penilaian prioritas dalam algoritma pencarian. Ketiga, heuristik yang digunakan perlu diperkuat agar lebih informatif, misalnya dengan memberi bobot pada jumlah penghalang atau mempertimbangkan kesulitan memindahkan kendaraan. Terakhir, test case yang digunakan dapat dibuat lebih kompleks agar algoritma dipaksa mengeksplorasi lebih banyak simpul, sehingga perbedaan efektivitas heuristik dan strategi pencarian bisa terlihat lebih jelas. Meskipun beberapa aspek implementasi saat ini masih dapat ditingkatkan, penting untuk dicatat bahwa

semua algoritma tetap berhasil menemukan solusi yang benar dan optimal pada seluruh test case. Hal ini menunjukkan bahwa kerangka dasar algoritma dan sistem heuristik telah bekerja secara fungsional

BAB VII

KESIMPULAN DAN SARAN

7.1 Kesimpulan

Implementasi penyelesaian puzzle Rush Hour dilakukan dalam Tugas Kecil 2 IF2211 Strategi Algoritma dengan menggunakan algoritma pencarian UCS (Uniform Cost Search), GBFS (Greedy Best First Search), dan A* Search. Program berhasil memenuhi seluruh spesifikasi wajib dengan memberikan solusi jalur yang valid untuk mengeluarkan primary piece menuju pintu keluar. Selain itu, fitur bonus seperti pemilihan heuristik dan algoritma tambahan, serta GUI (*Graphical User Interface*) dan visualisasi animasi pergerakan berhasil diimplementasikan dengan baik dan selesai.

Dari hasil implementasi, dapat disimpulkan bahwa pemanfaatan strategi *informed* atau *heuristic search* terbukti sangat efektif dalam mempercepat pencarian solusi pada permasalahan Rush Hour. Khususnya, algoritma A* berhasil menggabungkan keoptimalan UCS dan efisiensi GBFS, menjadikannya solusi ideal dan cepat untuk puzzle berbasis grid yang kompleks, dan menghasilkan solusi yang optimal bersama dengan algoritma UCS. Selain itu, fitur visualisasi pergerakan langkah demi langkah yang dilakukan di website memperkuat pemahaman pengguna terhadap proses eksplorasi dan strategi pemecahan yang diterapkan.

7.2 Saran

Program dapat dikembangkan lebih lanjut, terutama dalam sisi optimasi eksplorasi graf untuk proses pencarian yang lebih cepat lagi pada konfigurasi puzzle yang kompleks. Hal ini mungkin dapat dicapai ke depannya dengan penerapan heuristik yang lebih adaptif atau spesifik agar algoritma A* maupun GBFS dapat memanfaatkan heuristik yang lebih efisien untuk menghindari eksplorasi jalur tidak relevan. Setelah itu, algoritma-algoritma lain dapat digunakan untuk mengeksplorasi kemungkinan pencarian yang lebih bervariasi dan lebih efektif. Terakhir, validasi input dapat ditingkatkan lebih lengkap lagi untuk mencegah kesalahan *parsing*, maupun dengan pengembangan fitur visualisasi yang lebih bagus secara UI/UX (*User Interface/User Experience*) sehingga meningkatkan pengalaman pengguna yang lebih baik lagi.

BAB VIII

LAMPIRAN

8.1 Link Repository

https://github.com/lynaten/Tucil3_13523142_13523148

8.2 Tabel Hasil

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	✓	