Laporan Tugas Kecil 3 IF2211 Strategi Algoritma

Penyelesaian Puzzle Rush Hour Menggunakan Algoritma Pathfinding

Muhammad Fathur Rizky
13523105
13523105@std.stei.itb.ac.id

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

Daftar Isi

Problem Statement

Rush Hour merupakan sebuah permainan teka-teki berbasis grid yang mengharuskan pemain untuk menggeser kendaraan dalam sebuah papan berukuran n×n hingga mobil utama (dilambangkan dengan simbol P) dapat keluar melalui sebuah pintu yang terletak di sisi papan (dilambangkan dengan simbol K). Kendaraan dalam permainan ini hanya dapat bergerak dalam arah yang sesuai dengan orientasinya (horizontal atau vertikal), dan tidak boleh saling tumpang tindih. Meskipun terdengar sederhana, konfigurasi kendaraan yang beragam dan ruang gerak yang terbatas menyebabkan jumlah kemungkinan posisi (state-space) membesar secara eksponensial. Oleh karena itu, diperlukan pendekatan sistematis untuk menentukan urutan gerakan paling efisien yang memungkinkan mobil utama keluar dari papan dalam jumlah langkah sesedikit mungkin.

General Approach

Untuk menyelesaikan Rush Hour secara optimal dan membandingkan performa pendekatan yang digunakan, diujikan tiga algoritma berikut:

1. **Uniform Cost Search (UCS)**
   Setiap gerakan kendaraan dianggap memiliki cost yang sama, yaitu 1. UCS bekerja dengan selalu memilih state dengan nilai $g(n)$ (biaya dari awal hingga node tersebut) paling kecil. Dalam konteks cost yang seragam, UCS akan berperilaku serupa dengan Breadth-First Search dan menghasilkan solusi dengan jumlah langkah minimum

2. **Greedy Best-First Search (GBFS)**
   Dalam algoritma ini, pencarian diarahkan sepenuhnya berdasarkan nilai heuristic $h(n)$, yaitu estimasi seberapa dekat suatu state terhadap goal. State dengan nilai $h(n)$ terkecil akan diproses lebih dahulu. Pendekatan ini bersifat agresif karena hanya mempertimbangkan jarak estimasi tanpa memedulikan jalur yang telah ditempuh sehingga meskipun lebih cepat, hasilnya belum tentu optimal.

3. **A***
   Kombinasi antara jalur yang telah ditempuh dan estimasi ke goal digunakan melalui fungsi evaluasi $f(n) = h(n) + g(n)$. Selama heuristik yang digunakan bersifat admissible (tidak pernah melebih-lebihkan estimasi jarak ke goal), A* akan memberikan solusi optimal dan tetap efisien dalam eksplorasi node.

Untuk memperluas eksperimen, juga disiapkan infrastruktur yang memungkinkan pengujian algoritma alternatif dan heuristic lainnya. Pendekatan-pendekatan tersebut disusun dalam arsitektur modular agar dapat dievaluasi dan dibandingkan dengan mudah.

Implementation on Code

Structuring the Folder

```
// Struktur direktori utama (disederhanakan)

include/
├── Algorithm/
│   ├── Algorithm.hpp
│   ├── BestFirstSearch.hpp
│   ├── AStar.hpp
│   ├── UniformCostSearch.hpp
│   └── BeamSearch.hpp
├── Metric/
│   ├── IHeuristic.hpp
│   ├── ManhattanHeuristic.hpp
│   └── BlockingHeuristic.hpp
├── Model/
│   ├── Board.hpp
│   ├── Piece.hpp
│   └── State.hpp
src/
├── Algorithm/
├── Metric/
├── Model/
└── main.cpp
```

Dalam implementasi, sistem disusun secara modular agar fleksibel dan mudah diperluas. Folder include/ berisi seluruh header file yang mendefinisikan antarmuka dan struktur utama. Folder src/ berisi implementasi logika dari masing-masing modul.

Modul Algorithm/ memuat semua algoritma pencarian yang mengimplementasikan interface Algorithm. Modul Metric/ digunakan untuk menyimpan strategi heuristic yang digunakan pada GBFS dan A*. Semua struktur data seperti papan permainan (Board), kendaraan (Piece), dan representasi status (State) disimpan dalam Model/.

Modularisasi seperti ini memungkinkan pengguna untuk mengganti algoritma atau heuristic tanpa mengubah bagian lain dari sistem. Misalnya, untuk menambahkan heuristic baru, cukup membuat satu file .cpp dan .hpp di dalam Metric/, kemudian inject ke algoritma A*.

Data Structure Modeling

Board

```
class Board {
 public:
  Board(int, int);
  Board();
  ~Board();

  void setPosition(int x, int y, char c);
  char getPosition(int x, int y) const;
  int getWidth() const { return w_; }
  int getHeight() const { return h_; }

  void print(char) const;
  bool isSafe(int x, int y) const;
  Board clone() const;
  void setExit(int, int);
  std::pair<int, int> getExit() const;
  void printToStream(std::ostream& os) const;

 private:
  int w_, h_;
  int exitX_, exitY_;
  std::vector<std::vector<char>> board_;
};
```

Struktur data utama yang digunakan untuk merepresentasikan papan permainan Rush Hour diimplementasikan dalam kelas Board. Kelas ini bertanggung jawab untuk menyimpan dan memanipulasi kondisi papan saat ini, termasuk posisi seluruh kendaraan, ruang kosong, dan pintu keluar. Dalam implementasinya, papan direpresentasikan sebagai matriks dua dimensi std::vector<std::vector<char>> board_, di mana setiap sel menyimpan satu karakter yang mewakili isi dari posisi tersebut. Misalnya, titik '.' melambangkan ruang kosong, huruf kapital seperti 'A', 'B', dan 'P' merepresentasikan kendaraan tertentu, dan 'K' adalah simbol untuk pintu keluar.

Selain matriks papan, atribut w_ dan h_ digunakan untuk menyimpan lebar dan tinggi papan. Informasi ini digunakan untuk validasi koordinat dan iterasi dalam metode manipulasi papan. Untuk mendukung pengecekan apakah suatu sel berada dalam batas papan dan aman untuk dilalui, digunakan fungsi isSafe(int x, int y) yang mengecek apakah posisi yang dimaksud berada di dalam batas papan dan berisi sel yang dapat dilewati (kosong atau pintu keluar).

Pintu keluar disimpan secara eksplisit melalui pasangan koordinat (exitX_, exitY_) sehingga algoritma dapat dengan cepat menentukan apakah kendaraan primer telah mencapai titik tersebut tanpa harus memindai seluruh papan. Fungsi setExit(int x, int y) digunakan untuk

menetapkan lokasi pintu keluar berdasarkan input file, sedangkan getExit() akan mengembalikannya dalam bentuk pasangan std::pair<int, int>.

Untuk mendukung pencetakan papan ke terminal, kelas ini menyediakan dua metode print(char movedSymbol) dan printToStream(std::ostream&). Metode pertama mencetak papan ke layar dengan pewarnaan ANSI, menyorot kendaraan yang digerakkan, kendaraan primer, dan pintu keluar dengan warna yang berbeda. Hal ini memudahkan pengguna dalam mengikuti pergerakan kendaraan selama proses pencarian solusi. Metode printToStream digunakan untuk mencetak papan ke file teks dengan format yang bersih, tanpa pewarnaan.

Piece

```
enum Orientation { HORIZONTAL, VERTICAL };

class Piece {
 public:
  Piece(char symbol, int x, int y, int length, Orientation orientation,
        bool isPrimary);
  ~Piece();

  char getSymbol() const;
  int getX() const;
  int getY() const;
  int getLength() const;
  Orientation getOrientation() const;
  bool isPrimary() const;

  void setPosition(int x, int y);

  std::vector<std::pair<int, int>> getOccupiedCells() const;
  std::vector<std::pair<int, int>> getEdgeCells(int dx, int dy) const;

 private:
  char symbol;
  int x, y;
  int length;
  Orientation orientation;
  bool primary;
};
```

Struktur data Piece digunakan untuk merepresentasikan satu kendaraan pada permainan Rush Hour. Setiap objek Piece menyimpan informasi tentang identitas kendaraan tersebut, termasuk simbol yang digunakan untuk menandainya pada papan (symbol), posisi koordinat awal (x, y), panjang kendaraan (length), orientasi (orientation), serta penanda apakah kendaraan tersebut merupakan kendaraan utama atau tidak (primary). Informasi ini cukup untuk memodelkan semua perilaku kendaraan dalam permainan, termasuk gerakannya dalam satu arah (horizontal atau vertikal) dan pengecekan posisi yang ditempati.

Orientasi kendaraan disimpan dalam bentuk enumerasi Orientation yang dapat bernilai HORIZONTAL atau VERTICAL. Ini menentukan arah pergerakan kendaraan, serta bagaimana fungsi lain seperti getOccupiedCells() akan bekerja. Misalnya, kendaraan horizontal dengan panjang 3 dan posisi awal (x, y) akan menempati (x, y), (x+1, y), dan (x+2, y), sedangkan kendaraan vertikal akan menempati (x, y), (x, y+1), dan (x, y+2). Fungsi getOccupiedCells() mengembalikan daftar koordinat ini secara otomatis berdasarkan properti internal kendaraan.

Fungsi getEdgeCells(int dx, int dy) digunakan saat kendaraan hendak digerakkan, untuk mengetahui sel mana yang akan menyentuh area baru ketika kendaraan bergeser. Ini penting dalam algoritma pencarian agar hanya perlu memeriksa sel terluar dari kendaraan untuk validasi pergerakan. Fungsi ini menghitung bagian depan kendaraan sesuai arah pergerakan dan mengembalikannya dalam bentuk vektor koordinat.

Kelas ini juga menyediakan metode setPosition(int x, int y) yang digunakan untuk memperbarui posisi kendaraan pada saat terjadi pergerakan. Hal ini memungkinkan algoritma untuk mereplikasi pergerakan kendaraan tanpa perlu membuat ulang objek dari awal. Sifat isPrimary() digunakan untuk membedakan kendaraan utama dari kendaraan lain, karena hanya kendaraan utama yang menjadi kunci dalam pengecekan apakah kondisi goal telah tercapai.

State

```
struct State {
  Board board;
  std::vector<Piece> pieces;
  std::vector<std::string> path;
  std::vector<Board> history;

  State(const Board& b, const std::vector<Piece>& p,
        const std::vector<std::string>& moves,
        const std::vector<Board>& hist_ = {})
      : board(b.clone()), pieces(p), path(moves), history(hist_) {}

  std::string serialize() const;
};
```

Struktur State berfungsi sebagai representasi satu kondisi (state) dalam pencarian solusi permainan Rush Hour. Dalam konteks algoritma pencarian graf, satu simpul pada graf diwakili oleh satu objek State. Struktur ini menyimpan informasi lengkap tentang konfigurasi papan permainan saat ini, urutan gerakan yang telah dilakukan untuk mencapainya, serta riwayat visual dari tiap langkah yang telah dilewati. Komponen utama dari State meliputi: board, pieces, path, dan history.

Atribut board menyimpan kondisi papan saat ini menggunakan objek Board yang merupakan salinan dari papan pada saat state tersebut dibentuk. Clone dilakukan untuk memastikan bahwa perubahan pada satu state tidak mempengaruhi state lain—penting agar eksplorasi state-space

tetap independen antar cabang pencarian. Atribut pieces adalah vektor dari objek Piece yang merepresentasikan seluruh kendaraan yang ada di papan, lengkap dengan posisi, orientasi, dan atribut lainnya. Penyimpanan ini memastikan bahwa setiap State benar-benar merepresentasikan snapshot lengkap dari permainan pada waktu tertentu.

Atribut path berupa vektor string yang berisi daftar gerakan yang telah dilakukan sejak state awal hingga state ini. Setiap string mencatat satu gerakan kendaraan, misalnya "P-kanan2" yang artinya kendaraan P digerakkan ke kanan sejauh 2 langkah. Informasi ini sangat penting karena menjadi dasar untuk menampilkan solusi akhir kepada pengguna dan untuk menghitung fungsi evaluasi seperti $g(n)$ dalam UCS atau A*.

Atribut tambahan history menyimpan urutan objek Board yang telah dilewati selama pencarian. Ini memungkinkan visualisasi progres per langkah, termasuk pewarnaan kendaraan yang baru digerakkan. Meskipun tidak diperlukan untuk logika pencarian inti, history sangat berguna untuk keperluan pelaporan atau antarmuka pengguna, seperti mencetak solusi berwarna di terminal.

Fungsi serialize() yang dideklarasikan tetapi tidak ditunjukkan dalam potongan kode ini biasanya digunakan untuk menghasilkan representasi unik dari suatu State (misalnya dalam bentuk string) agar dapat digunakan sebagai kunci dalam struktur data seperti std::unordered_set atau std::unordered_map. Hal ini penting untuk menghindari eksplorasi ulang terhadap state yang sama.

File Processor

```
class FileProcessor {
 public:
  FileProcessor();
  ~FileProcessor();

  void load(Board& board, std::vector<Piece>& pieces, int& algorithmChoice,
            int& heuristicChoice);
  void loadFromFile(const std::string& fileName, Board& board,
                    std::vector<Piece>& pieces);

  void save(const std::vector<std::string>& moveLog,
            const std::vector<Board>& history, double time, int nodes,
            const std::string& fileName);
  void saveToFile(const std::string& fileName,
                  const std::vector<Board>& history,
                  const std::vector<std::string>& moveLog, double time,
                  int nodes);

 private:
  std::string rtrim(const std::string& s) {
    return std::string(s.begin(),
                       std::find_if(s.rbegin(), s.rend(), [](unsigned char
ch) {
```

```
                        return !std::isspace(ch);
                    }).base());
  }

  std::string trim(const std::string& s) { return rtrim(s); }
};
```

Kelas FileProcessor dirancang untuk menangani seluruh proses input dan output file dalam sistem penyelesaian puzzle Rush Hour. Fungsinya mencakup membaca konfigurasi awal permainan dari file input serta menyimpan solusi ke file output setelah pencarian selesai. Kelas ini memastikan bahwa format data dari file eksternal dapat diterjemahkan dengan benar menjadi struktur internal program, seperti objek Board dan daftar Piece, dan sebaliknya, menghasilkan output yang mudah dibaca dan sesuai spesifikasi.

Metode load() berperan sebagai antarmuka utama untuk proses input. Dalam penggunaannya, pengguna akan diminta memasukkan nama file input, memilih algoritma pencarian yang akan digunakan, serta—jika diperlukan—memilih heuristic yang digunakan untuk algoritma A*. Seluruh informasi ini kemudian dikombinasikan dan diteruskan ke loadFromFile() yang bertugas membaca isi file secara detail. Metode ini mengonversi baris-baris teks dalam file menjadi bentuk representasi papan dan kendaraan sesuai struktur data internal program.

Sementara itu, fungsi save() digunakan untuk menyimpan hasil solusi ke dalam file output. Informasi yang disimpan meliputi langkah-langkah gerakan kendaraan, jejak konfigurasi papan (history), waktu eksekusi, serta jumlah simpul yang dikunjungi selama proses pencarian. Seluruh proses penulisan file difasilitasi oleh saveToFile() yang bertanggung jawab menuliskan data dengan format yang rapi dan terstruktur ke dalam file tujuan.

Kelas ini juga menyertakan fungsi bantu trim() dan rtrim() yang digunakan untuk membersihkan whitespace pada setiap baris input dari file, guna mencegah kesalahan parsing. Dengan struktur yang terpisah dan tanggung jawab yang jelas, FileProcessor mendukung prinsip pemisahan logika I/O dari logika algoritma yang penting dalam menjaga keterbacaan dan pemeliharaan kode.

The Heuristic Function

Manhattan Distance

```
int ManhattanHeuristic::evaluate(const Board& board) const {
  auto [exitX, exitY] = board.getExit();

  int minX = board.getWidth(), maxX = -1;
  int minY = board.getHeight(), maxY = -1;

  for (int y = 0; y < board.getHeight(); ++y) {
    for (int x = 0; x < board.getWidth(); ++x) {
```

```
      if (board.getPosition(x, y) == 'P') {
        minX = std::min(minX, x);
        maxX = std::max(maxX, x);
        minY = std::min(minY, y);
        maxY = std::max(maxY, y);
      }
    }
  }

  if (maxX == -1 || maxY == -1) return 0;

  int dx = 0, dy = 0;
  if (exitX < minX)
    dx = minX - exitX;
  else if (exitX > maxX)
    dx = exitX - maxX;

  if (exitY < minY)
    dy = minY - exitY;
  else if (exitY > maxY)
    dy = exitY - maxY;

  return dx + dy;
}
```

Fungsi ManhattanHeuristic::evaluate bertujuan untuk memperkirakan seberapa dekat kendaraan utama (primer) P dengan pintu keluar K, menggunakan jarak Manhattan sebagai ukuran. Heuristic ini bekerja dengan mencari jarak minimum dalam arah horizontal dan vertikal dari posisi kendaraan primer ke posisi pintu keluar, dan menjumlahkan keduanya. Karena hanya menggunakan jarak aktual dan tidak memperhitungkan hambatan lain (seperti kendaraan yang menghalangi), heuristic ini dijamin admissible, artinya tidak akan pernah melebih-lebihkan cost sebenarnya untuk mencapai goal.

Langkah pertama dalam fungsi ini adalah mengambil koordinat pintu keluar menggunakan getExit(), lalu menentukan batas koordinat kendaraan primer: minX, maxX, minY, dan maxY. Ini dilakukan dengan memindai seluruh papan dan mencatat semua sel yang mengandung simbol 'P' yang mewakili bagian dari kendaraan primer. Nilai minimum dan maksimum dari posisi horizontal dan vertikal kendaraan primer dicatat untuk membentuk “bounding box” kendaraan tersebut.

Setelah posisi kendaraan utama diketahui, fungsi membandingkan posisi bounding box kendaraan dengan posisi pintu keluar. Jika pintu keluar berada di luar batas kanan kendaraan (lebih besar dari maxX), maka dihitung jarak horizontal dari sisi kanan kendaraan ke pintu. Jika pintu berada di sisi kiri, maka digunakan selisih antara minX dan exitX. Hal serupa juga dilakukan secara vertikal dengan membandingkan exitY dengan minY dan maxY.

Nilai akhir heuristic dikembalikan sebagai penjumlahan dari dx dan dy yang merepresentasikan jarak Manhattan antara kendaraan utama dan pintu keluar. Karena heuristic ini hanya memperhitungkan jarak lurus tanpa mengasumsikan hambatan atau jalur terblokir, maka nilai yang dihasilkan selalu lebih kecil atau sama dengan jumlah langkah sebenarnya yang dibutuhkan

The Block Counter

```
int BlockingHeuristic::evaluate(const Board& board) const {
  auto [exitX, exitY] = board.getExit();
  int width = board.getWidth();
  int height = board.getHeight();

  std::unordered_set<char> blockers;

  if (exitX == 0 || exitX == width - 1) {
    for (int x = 1; x < width - 1; ++x) {
      char c = board.getPosition(x, exitY);
      if (c != '.' && c != 'K' && c != 'P') blockers.insert(c);
    }
  }

  if (exitY == 0 || exitY == height - 1) {
    for (int y = 1; y < height - 1; ++y) {
      char c = board.getPosition(exitX, y);
      if (c != '.' && c != 'K' && c != 'P') blockers.insert(c);
    }
  }

  return static_cast<int>(blockers.size());
}
```

Fungsi BlockingHeuristic::evaluate di atas merupakan implementasi heuristic sederhana yang menghitung jumlah kendaraan penghalang langsung antara posisi pintu keluar dan bagian dalam papan permainan. Tidak seperti heuristic lain yang menitikberatkan pada posisi kendaraan primer (misalnya mencari bounding box kendaraan P), pendekatan ini berfokus sepenuhnya pada lokasi pintu keluar dan menghitung berapa banyak kendaraan yang berada pada jalur keluar kendaraan utama. Pendekatan ini cenderung lebih fleksibel dan tidak bergantung pada arah atau bentuk kendaraan primer.

Fungsi dimulai dengan mengambil koordinat pintu keluar (exitX, exitY) serta ukuran papan. Kemudian, dua kasus dipertimbangkan: apakah pintu keluar berada pada sisi kiri/kanan (horizontal), atau pada sisi atas/bawah (vertikal). Jika pintu berada di sisi horizontal, maka jalur yang diperiksa adalah seluruh baris exitY (dari kolom 1 hingga kolom lebar−2) kecuali tepi. Sebaliknya, jika pintu terletak di sisi vertikal, maka diperiksa seluruh kolom exitX (dari baris 1 hingga baris tinggi−2).

Untuk setiap sel pada jalur tersebut, simbol kendaraan diambil dan diperiksa apakah bukan merupakan ruang kosong ('.'), pintu keluar ('K'), maupun kendaraan primer ('P')—ketiganya dikecualikan dari perhitungan karena bukan blocker yang relevan. Jika simbol memenuhi syarat sebagai penghalang, maka dimasukkan ke dalam struktur std::unordered_set<char> blockers yang secara otomatis memastikan bahwa hanya kendaraan unik yang dihitung satu kali. Nilai akhir dari heuristic adalah ukuran set tersebut, yakni jumlah blocker unik yang menghalangi jalur keluar.

Pendekatan ini memiliki dua keuntungan utama: pertama, ia lebih efisien secara komputasi karena tidak perlu mengidentifikasi posisi kendaraan primer; kedua, ia tetap bersifat admissible karena hanya menghitung blocker yang benar-benar berada di jalur keluar, tanpa melebih-lebihkan cost.

The Algorithm

Algorithm Base Class

```
class Algorithm {
 public:
  Algorithm();
  virtual ~Algorithm();

  virtual State solve(const Board& board, const std::vector<Piece>& pieces) =
0;

  int getNodeVisitedCount() const;
  double getExecutionTimeMs() const;

 protected:
  int nodeVisitedCount;
  std::chrono::high_resolution_clock::time_point startTime;
  std::chrono::high_resolution_clock::time_point endTime;

  void startTimer();
  void endTimer();
  void reconstructPath();
  bool isGoal(const State& state);
  void printSolution(const Board& initialBoard,
                     const std::vector<Piece>& initialPieces,
                     const State& goalState);
  std::vector<State> expand(const State& currentState);
  bool canMove(const Board& board, const Piece& piece, int dx, int dy);
};
```

Header kelas Algorithm merupakan deklarasi dari kelas abstrak yang berfungsi sebagai antarmuka umum bagi seluruh algoritma pencarian yang digunakan dalam penyelesaian puzzle Rush Hour. Kelas ini dirancang menggunakan prinsip pewarisan dan polimorfisme, di mana

setiap algoritma spesifik seperti Uniform Cost Search, A*, atau Greedy Best-First Search akan menurunkan kelas ini dan mengimplementasikan fungsi utama solve() sesuai dengan strategi pencarian masing-masing.

Fungsi solve bersifat virtual murni (pure virtual) yang artinya kelas ini tidak dapat diinstansiasi langsung dan harus diturunkan terlebih dahulu. Fungsi ini menjadi titik masuk bagi proses pencarian solusi dan akan diimplementasikan dalam kelas turunan. Kelas ini juga menyertakan beberapa fungsi pendukung seperti expand() untuk menghasilkan tetangga dari suatu state, isGoal() untuk memeriksa apakah state yang sedang dievaluasi merupakan solusi, serta printSolution() untuk mencetak jalur solusi dan papan permainan secara visual.

Selain logika pencarian, kelas ini juga menyediakan alat pengukuran performa seperti startTimer() dan endTimer() yang mencatat waktu eksekusi dengan presisi tinggi menggunakan std::chrono, dan getExecutionTimeMs() yang mengembalikan waktu dalam satuan milidetik. Fungsi getNodeVisitedCount() digunakan untuk mengakses jumlah simpul yang telah dieksplorasi selama pencarian.

Semua data dan fungsi pembantu disimpan dalam akses protected, agar dapat digunakan oleh kelas turunan namun tetap tersembunyi dari pemanggilan eksternal. Dengan desain seperti ini, Algorithm menyediakan kerangka dasar yang konsisten dan reusable sehingga setiap algoritma dapat dibangun dengan logika pencarian spesifik tanpa harus menulis ulang fungsi-fungsi utilitas yang bersifat umum.

Best First Search

```
struct CompareHeuristic {
  bool operator()(const std::pair<State, int>& a,
                  const std::pair<State, int>& b) const {
    return a.second > b.second;
  }
};

State BestFirstSearch::solve(const Board& initialBoard,
                             const std::vector<Piece>& initialPieces) {
  startTimer();

  std::priority_queue<std::pair<State, int>, std::vector<std::pair<State,
int>>,
                      CompareHeuristic>
      pq;
  std::unordered_set<std::string> visited;

  State start(initialBoard, initialPieces, {}, {});
  pq.push({start, heuristic->evaluate(initialBoard)});

  while (!pq.empty()) {
    auto [currentState, currentHeuristic] = pq.top();
```

```
    pq.pop();
    std::string key = currentState.serialize();

    if (visited.count(key)) continue;
    visited.insert(key);
    nodeVisitedCount++;

    if (isGoal(currentState)) {
      endTimer();
      printSolution(initialBoard, initialPieces, currentState);
      return currentState;
    }

    for (const State& nextState : expand(currentState)) {
      std::string nextKey = nextState.serialize();
      if (!visited.count(nextKey)) {
        pq.push({nextState, heuristic->evaluate(nextState.board)});
      }
    }
  }

  printSolution(initialBoard, initialPieces, start);
  endTimer();
  return State(initialBoard, initialPieces, {}, {});
}
```

Kode di atas merupakan implementasi fungsi solve dari algoritma Greedy Best-First Search (GBFS) dalam konteks permainan Rush Hour. Algoritma ini memanfaatkan struktur data priority queue untuk memilih state berikutnya yang akan dieksplorasi, berdasarkan nilai heuristic h(n) dari state tersebut. Fokus utama GBFS adalah mendekati goal secepat mungkin dengan mempercayai estimasi jarak (heuristic) ke tujuan, tanpa memperhitungkan cost yang telah ditempuh (tidak mempertimbangkan g(n)).

Struktur pembanding CompareHeuristic digunakan oleh priority queue untuk memastikan bahwa state dengan nilai heuristic terkecil memiliki prioritas tertinggi. Dalam konteks ini, priority queue menyimpan pasangan std::pair<State, int>, di mana int mewakili nilai h(n) dari state tersebut. Semakin kecil nilai heuristic, semakin dekat estimasi state tersebut terhadap goal, dan semakin tinggi prioritasnya untuk diproses lebih dahulu.

Fungsi solve() dimulai dengan menginisialisasi state awal menggunakan konfigurasi papan dan kendaraan yang diberikan. State ini kemudian dimasukkan ke dalam antrian prioritas dengan nilai heuristic yang dihitung berdasarkan papan awal. Selanjutnya, selama priority queue tidak kosong, algoritma akan mengambil state dengan nilai heuristic terkecil (state yang tampaknya paling dekat ke goal) untuk diperiksa.

Setiap state yang diambil dari antrian akan dicek apakah sudah pernah dikunjungi menggunakan visited set yang menyimpan hasil serialisasi state (biasanya berupa representasi string unik). Jika belum, maka state akan diproses: dicatat sebagai dikunjungi, dicek apakah

sudah mencapai kondisi goal, dan jika belum, semua tetangganya (state hasil gerakan satu langkah dari state saat ini) akan dihasilkan melalui fungsi expand(). Untuk setiap tetangga yang belum dikunjungi, heuristic-nya dihitung dan dimasukkan ke antrian.

Apabila goal berhasil ditemukan, fungsi akan menghentikan pencatatan waktu, mencetak solusi, dan mengembalikan state akhir yang merupakan jalur solusi. Jika antrian habis tanpa menemukan goal, maka fungsi akan tetap mencetak state awal sebagai fallback dan mengembalikan state kosong. Karena GBFS hanya menggunakan h(n) dan mengabaikan g(n), ia cenderung lebih cepat dalam menemukan solusi, namun tidak menjamin bahwa solusi tersebut optimal (jumlah langkah terpendek). Hal ini membedakannya secara mendasar dari algoritma A* yang mempertimbangkan keduanya.

Uniform Cost Search

```
struct CompareG {
  bool operator()(const std::pair<State, int>& a,
                  const std::pair<State, int>& b) const {
    return ((int)a.first.path.size()) > ((int)b.first.path.size());
  }
};

State UniformCostSearch::solve(const Board& initialBoard,
                               const std::vector<Piece>& initialPieces) {
  startTimer();

  std::priority_queue<std::pair<State, int>, std::vector<std::pair<State,
int>>,
                      CompareG>
      pq;
  std::unordered_set<std::string> visited;

  State start(initialBoard, initialPieces, {}, {});
  pq.push({start, 0});

  while (!pq.empty()) {
    auto [current, g] = pq.top();
    pq.pop();
    std::string key = current.serialize();
    if (visited.count(key)) continue;
    visited.insert(key);
    nodeVisitedCount++;

    if (isGoal(current)) {
      endTimer();
      printSolution(initialBoard, initialPieces, current);
      return current;
    }

    auto neighbors = expand(current);
```

```
    for (const State& nxt : neighbors) {
      std::string nk = nxt.serialize();
      if (!visited.count(nk)) {
        pq.push({nxt, (int)nxt.path.size()});
      }
    }
  }

  printSolution(initialBoard, initialPieces, start);
  endTimer();
  return State(initialBoard, initialPieces, {}, {});
}
```

Kode di atas merupakan implementasi dari fungsi solve pada kelas UniformCostSearch, yaitu algoritma pencarian yang memperluas simpul berdasarkan biaya terkecil dari simpul awal (root) ke simpul saat ini. Dalam konteks permainan Rush Hour, setiap gerakan kendaraan dianggap memiliki cost tetap sebesar 1 sehingga algoritma ini pada dasarnya identik dengan Breadth-First Search (BFS) yang mencari solusi dengan jumlah langkah minimum. Namun, implementasi ini tetap menggunakan priority queue dengan comparator kustom (CompareG) untuk mendukung kemudahan ekspansi algoritma yang lebih umum.

Struktur CompareG berfungsi sebagai pembanding elemen dalam std::priority_queue. Fungsi ini membandingkan dua state berdasarkan panjang path yang telah dilalui, yaitu g(n)—biaya kumulatif dari awal hingga state tersebut. Semakin pendek path-nya, semakin tinggi prioritas state tersebut dalam antrian. Dengan demikian, priority queue akan selalu memproses state dengan jumlah langkah terkecil lebih dahulu.

Algoritma dimulai dengan membuat State awal berdasarkan konfigurasi papan dan kendaraan. State ini dimasukkan ke dalam priority queue dengan biaya awal 0. Selanjutnya, selama antrian tidak kosong, algoritma akan mengambil state dengan g(n) terkecil. Jika state tersebut belum pernah dikunjungi (diperiksa menggunakan visited set berbasis hasil serialisasi unik), maka state akan diperiksa apakah telah mencapai kondisi goal. Jika ya, maka solusi ditemukan dan fungsi segera mengakhiri proses dengan mencetak solusi dan mengembalikan state akhir.

Jika goal belum tercapai, semua state tetangga dari state saat ini dihasilkan menggunakan fungsi expand(), lalu diperiksa satu per satu. Jika state tetangga belum dikunjungi, maka dimasukkan ke dalam priority queue dengan biaya kumulatif yang dihitung dari panjang path-nya (jumlah langkah hingga state itu). Karena semua gerakan memiliki cost yang sama (1), panjang path dapat langsung dianggap sebagai g(n) tanpa perhitungan tambahan.

Keunggulan dari algoritma Uniform Cost Search adalah kemampuannya dalam menemukan solusi yang optimal secara jumlah langkah, selama semua edge (gerakan) memiliki cost yang sama. Kekurangannya adalah potensi eksplorasi state yang sangat luas, karena tidak memanfaatkan informasi heuristic untuk mengarahkan pencarian. Oleh karena itu, meskipun

UCS menjamin optimalitas solusi, ia sering kali membutuhkan lebih banyak waktu dan eksplorasi dibanding algoritma seperti A*.

A* Search

```
struct CompareF {
  bool operator()(const std::pair<State, int>& a,
                  const std::pair<State, int>& b) const {
    int fa = (int)a.first.path.size() + a.second;
    int fb = (int)b.first.path.size() + b.second;
    return fa > fb;
  }
};

State AStar::solve(const Board& initialBoard,
                   const std::vector<Piece>& initialPieces) {
  startTimer();

  std::priority_queue<std::pair<State, int>, std::vector<std::pair<State,
int>>,
                      CompareF>
      open;

  std::unordered_map<std::string, int> bestG;
  State start(initialBoard, initialPieces, {}, {});
  open.push({start, heuristic->evaluate(initialBoard)});
  bestG[start.serialize()] = 0;

  while (!open.empty()) {
    auto [cur, h] = open.top();
    open.pop();
    std::string key = cur.serialize();
    int g = (int)cur.path.size();

    if (bestG[key] < g) continue;

    if (isGoal(cur)) {
      endTimer();
      printSolution(initialBoard, initialPieces, cur);
      return cur;
    }

    auto neighbors = expand(cur);
    for (auto& nxt : neighbors) {
      std::string sk = nxt.serialize();
      int ng = (int)nxt.path.size();

      if (!bestG.count(sk) || ng < bestG[sk]) {
        bestG[sk] = ng;
        int hn = heuristic->evaluate(nxt.board);
        open.push({nxt, hn});
```

```
      }
    }
  }

  printSolution(initialBoard, initialPieces, start);
  endTimer();
  return State(initialBoard, initialPieces, {}, {});
}
```

Kode di atas merupakan implementasi algoritma A* dalam konteks permainan Rush Hour. A* adalah algoritma pencarian informatif yang memadukan antara biaya perjalanan dari titik awal ke suatu state $g(n)$ dan estimasi biaya dari state tersebut ke tujuan $h(n)$ sehingga total fungsi evaluasi yang digunakan adalah $f(n) = g(n) + h(n)$. Implementasi ini mengutamakan efisiensi dengan tetap menjamin solusi yang optimal selama heuristic yang digunakan bersifat admissible (tidak melebih-lebihkan cost sebenarnya ke goal).

Struktur CompareF digunakan untuk mengatur elemen-elemen di dalam priority queue. Fungsi ini membandingkan dua elemen berdasarkan nilai $f(n)$, yaitu hasil penjumlahan antara panjang path yang telah ditempuh $g(n)$ dan nilai heuristic $h(n)$ dari state yang bersangkutan. Dengan demikian, priority queue akan selalu memproses state dengan total nilai $f(n)$ terkecil terlebih dahulu yang merupakan inti dari strategi A*.

Di awal fungsi solve, state awal dibentuk dari konfigurasi papan dan daftar kendaraan, lalu dihitung nilai heuristic-nya dan dimasukkan ke dalam priority queue. Peta bestG digunakan untuk menyimpan nilai minimum $g(n)$ yang diketahui untuk setiap state berdasarkan hasil serialisasi. Hal ini mencegah algoritma mengeksplorasi kembali state yang sudah dikunjungi dengan path yang lebih panjang, menjaga efisiensi dan menghindari redundansi.

Setiap kali state diproses dari priority queue, dilakukan pengecekan apakah state tersebut sudah pernah ditemukan sebelumnya dengan nilai $g(n)$ yang lebih baik. Jika tidak, dan state tersebut merupakan goal (yaitu kendaraan primer mencapai pintu keluar), maka proses pencarian dihentikan dan solusi dicetak. Jika belum, maka semua neighbor dari state saat ini dihasilkan melalui fungsi expand, lalu dimasukkan ke priority queue jika belum ditemukan atau jika ditemukan dengan path yang lebih pendek.

Keunggulan algoritma A* terletak pada keseimbangan antara eksplorasi dan eksploitasi. Dengan menggabungkan $g(n)$ dan $h(n)$, algoritma tidak hanya mempertimbangkan sejauh mana sudah bergerak, tapi juga seberapa menjanjikan state tersebut menuju goal. Dengan catatan bahwa heuristic yang digunakan harus admissible, A* akan menemukan solusi optimal lebih cepat dibanding UCS karena mampu “menghindari” jalur yang jauh dan tidak produktif. Secara keseluruhan, implementasi ini menunjukkan penerapan A* yang efisien, modular, dan siap dikombinasikan dengan berbagai jenis heuristic melalui dependency injection.

Beam Search

```
struct CompareHeuristic {
  bool operator()(const std::pair<State, int>& a,
                  const std::pair<State, int>& b) const {
    return a.second > b.second;
  }
};

State BeamSearch::solve(const Board& initialBoard,
                        const std::vector<Piece>& initialPieces) {
  startTimer();

  std::vector<std::pair<State, int>> currentLevel;
  std::unordered_set<std::string> visited;

  State start(initialBoard, initialPieces, {}, {});
  int h0 = heuristic->evaluate(initialBoard);
  currentLevel.push_back({start, h0});
  visited.insert(start.serialize());

  while (!currentLevel.empty()) {
    std::vector<std::pair<State, int>> nextLevel;

    for (auto& [currentState, currentHeuristic] : currentLevel) {
      nodeVisitedCount++;
      if (isGoal(currentState)) {
        endTimer();
        printSolution(initialBoard, initialPieces, currentState);
        return currentState;
      }

      auto neighbors = expand(currentState);
      for (auto& nxt : neighbors) {
        std::string key = nxt.serialize();
        if (visited.count(key) == 0) {
          int h = heuristic->evaluate(nxt.board);
          nextLevel.emplace_back(nxt, h);
          visited.insert(key);
        }
      }
    }

    if ((int)nextLevel.size() > beamWidth_) {
      std::nth_element(
          nextLevel.begin(), nextLevel.begin() + beamWidth_, nextLevel.end(),
          [](const auto& a, const auto& b) { return a.second < b.second; });
      nextLevel.resize(beamWidth_, nextLevel[beamWidth_]);
    }

    currentLevel = std::move(nextLevel);
  }
```

```
  endTimer();
  printSolution(initialBoard, initialPieces, start);
  return State(initialBoard, initialPieces, {}, {});
}
```

Kode di atas merupakan implementasi algoritma Beam Search yang merupakan variasi dari Greedy Best-First Search (GBFS) dengan pembatasan lebar pencarian (beam width). Algoritma ini mencoba memadukan kecepatan GBFS dalam memilih jalur yang menjanjikan berdasarkan heuristic, namun dengan penghematan memori dan waktu eksplorasi melalui pembatasan jumlah simpul yang diperluas pada setiap level kedalaman pencarian.

Algoritma dimulai dengan membuat state awal dari konfigurasi papan dan daftar kendaraan, lalu menghitung nilai heuristic awal. State awal ini dimasukkan ke dalam vektor currentLevel bersama dengan nilai heuristic-nya dan juga dicatat dalam struktur visited untuk menghindari eksplorasi ulang. Vektor ini merepresentasikan kumpulan state yang sedang dieksplorasi pada kedalaman saat ini.

Pada setiap iterasi utama, setiap state di currentLevel akan diperiksa satu per satu. Jika salah satu state merupakan goal, pencarian dihentikan, timer dihentikan, dan solusi diprint. Jika belum, semua tetangga dari state tersebut dihasilkan melalui fungsi expand(). Tetangga yang belum pernah dikunjungi sebelumnya ditambahkan ke nextLevel beserta nilai heuristicnya, dan langsung dicatat dalam visited untuk menghindari duplikasi eksplorasi.

Setelah seluruh tetangga di level tersebut terkumpul di nextLevel, algoritma menerapkan pembatasan beam width dengan memilih hanya beamWidth_ state dengan nilai heuristic terbaik. Proses ini dilakukan dengan std::nth_element untuk menemukan posisi pivot pada urutan nilai heuristic, lalu resize() untuk memotong vektor sehingga hanya tersisa simpul-simpul yang dianggap paling menjanjikan untuk eksplorasi berikutnya. Ini merupakan inti pembeda Beam Search dari GBFS biasa, di mana GBFS tanpa batas akan menyimpan semua state yang ditemukan sehingga berpotensi sangat boros memori.

Terakhir, currentLevel di-set ulang menjadi nextLevel yang sudah dipangkas, dan iterasi dilanjutkan hingga solusi ditemukan atau tidak ada lagi state untuk dieksplorasi. Dengan membatasi lebar pencarian, Beam Search menawarkan trade-off antara efisiensi komputasi dan kemungkinan menemukan solusi optimal sehingga cocok untuk masalah dengan ruang pencarian sangat besar seperti Rush Hour.

The Theoretical Analysis

Dalam algoritma A*, terdapat dua fungsi penting yaitu $f(n)$ dan $g(n)$. Fungsi $f(n)$ merupakan estimasi total biaya terendah dari simpul awal hingga ke simpul tujuan melalui simpul n. Nilainya didapat dari penjumlahan $g(n) + h(n)$, di mana $g(n)$ adalah biaya sebenarnya dari simpul awal menuju simpul n, dan $h(n)$ adalah estimasi biaya dari simpul n ke simpul tujuan (disebut juga

sebagai fungsi heuristik). Definisi ini umum digunakan dalam A* dan sejalan dengan yang dijelaskan dalam salindia kuliah.

Heuristik yang digunakan pada algoritma A* dikatakan admissible apabila ia tidak pernah melebihi biaya minimum sebenarnya dari simpul saat ini ke simpul tujuan. Dengan kata lain, heuristik tersebut selalu optimis, dan tidak memberikan estimasi yang lebih tinggi dari kenyataannya. Berdasarkan definisi ini, apabila heuristik pada algoritma A* memenuhi sifat admissible, maka algoritma A* dijamin akan menemukan solusi optimal, karena ia tidak akan melewati jalur yang lebih mahal akibat kesalahan estimasi dari heuristik.

Pada penyelesaian permainan Rush Hour, algoritma Uniform Cost Search (UCS) dan Breadth-First Search (BFS) bisa dianggap serupa dalam urutan node yang dibangkitkan dan solusi yang dihasilkan hanya jika semua langkah memiliki biaya yang sama (misalnya setiap langkah memiliki bobot 1). Dalam konteks ini, UCS dan BFS akan menelusuri simpul berdasarkan kedalaman yang sama (karena biaya sama) sehingga urutan node dan path yang dihasilkan akan identik. Namun, jika biaya antar langkah berbeda, UCS akan memperhitungkan biaya minimum dan bisa menghasilkan urutan node yang berbeda dari BFS.

Secara teoritis, algoritma A* lebih efisien dibandingkan UCS pada penyelesaian Rush Hour, asalkan heuristik yang digunakan admissible dan cukup informatif. UCS akan menelusuri semua kemungkinan berdasarkan biaya, tanpa panduan arah ke tujuan. Sementara A* memanfaatkan heuristik untuk lebih cepat mengarahkan pencarian ke solusi. Karena A* bisa mengabaikan banyak simpul yang tidak menjanjikan, ia secara umum lebih efisien dalam jumlah node yang dibangkitkan dan waktu eksekusi.

Sebaliknya, algoritma Greedy Best First Search tidak menjamin solusi optimal untuk penyelesaian Rush Hour. Hal ini karena Greedy hanya memperhatikan nilai heuristik $h(n)$ tanpa mempertimbangkan biaya aktual $g(n)$ dari simpul awal. Jika heuristiknya menyesatkan atau tidak admissible, maka algoritma ini dapat memilih jalur yang tampaknya menjanjikan tapi sebenarnya lebih mahal sehingga solusi yang ditemukan tidak optimal. Oleh karena itu, Greedy BFS bersifat lebih cepat tetapi tidak menjamin optimalitas solusi.

Complexity Analysis

Best First Search

Greedy Best-First Search (GBFS) memiliki kompleksitas waktu dalam kasus terburuk sebesar $O(b^m)$, dengan b sebagai branching factor atau jumlah kemungkinan gerakan dari suatu state, dan m sebagai kedalaman maksimum dari pohon pencarian. Karena GBFS hanya mempertimbangkan nilai heuristic $h(n)$ dan mengabaikan $g(n)$, algoritma ini akan mengejar state yang tampak paling dekat ke goal menurut estimasi heuristic. Jika heuristic yang digunakan cukup akurat, jumlah node yang dikunjungi bisa jauh lebih sedikit. Namun, karena

tidak mempertimbangkan total cost, GBFS bisa terjebak dalam jalur yang kelihatannya baik tetapi sebenarnya panjang atau buntu. Kompleksitas ruang GBFS juga $O(b^m)$, karena harus menyimpan semua state dalam frontier (priority queue) dan visited set. Operasi pada priority queue menambahkan overhead logaritmik $O(\log n)$ per penyisipan atau pengambilan state terbaik, sedangkan akses ke visited set melalui unordered_set memiliki rata-rata kompleksitas O(1).

Uniform Cost Search

Uniform Cost Search (UCS) memiliki kompleksitas waktu $O(b^d)$, di mana d adalah kedalaman solusi optimal. Karena UCS selalu memilih state dengan nilai g(n) terkecil, ia menjelajahi semua state yang memiliki biaya lebih rendah terlebih dahulu tanpa bantuan arah dari heuristic. UCS menjamin solusi optimal dalam konteks cost uniform seperti pada permainan Rush Hour, namun kekurangannya adalah eksplorasi yang sangat luas, terutama jika solusi berada cukup dalam. Kompleksitas ruang UCS juga $O(b^d)$, karena ia harus menyimpan seluruh frontier dan visited set yang mungkin terus membesar. Struktur data priority queue digunakan untuk memilih state dengan g(n) terkecil, memberikan overhead $O(\log n)$ setiap kali dilakukan penambahan atau pengambilan elemen, sedangkan visited set menggunakan unordered_set dengan akses rata-rata O(1).

A* Search

A* Search menggabungkan kelebihan UCS dan GBFS dengan fungsi evaluasi $f(n) = g(n) + h(n)$. Kompleksitas waktunya dalam kasus terburuk juga $O(b^d)$, namun dalam praktik seringkali lebih baik dibanding UCS karena heuristic membantu mengarahkan pencarian lebih efisien. Dengan heuristic yang admissible dan konsisten, A* menjamin solusi optimal. Kompleksitas ruang A* tetap O(b^d), karena perlu menyimpan semua state yang dikunjungi dan masih dalam frontier. Priority queue digunakan untuk memilih state dengan f(n) terkecil, dengan overhead $O(\log n)$ per operasi. A* juga menyimpan nilai g(n) terbaik untuk setiap state yang ditemukan menggunakan unordered_map yang memberi akses efisien rata-rata O(1). A* biasanya menjadi pilihan ideal jika tersedia heuristic yang cukup informatif, karena mampu menyeimbangkan antara optimalitas dan efisiensi waktu. Namun, dibanding GBFS, A* akan menggunakan lebih banyak memori karena menyimpan dan mengevaluasi lebih banyak informasi per state.

Beam Search

Algoritma Beam Search memiliki kompleksitas waktu dan ruang yang bergantung pada parameter beam width yang digunakan. Secara umum, Beam Search mengurangi kompleksitas eksplorasi state-space dengan membatasi jumlah state yang dipertahankan dan dikembangkan pada setiap level pencarian hanya sebanyak nilai beam width sehingga kompleksitas waktunya

kira-kira menjadi $O(b \times w \times d)$, di mana b adalah rata-rata branching factor, w adalah beam width, dan d adalah kedalaman solusi. Dengan membatasi eksplorasi pada subset state yang paling menjanjikan berdasarkan heuristic, Beam Search mampu menghindari ledakan eksponensial yang biasanya terjadi pada algoritma pencarian lengkap seperti UCS atau A*. Namun, karena hanya sebagian dari state yang diperluas di setiap level, algoritma ini tidak menjamin menemukan solusi optimal, terutama jika beam width terlalu kecil sehingga jalur optimal mungkin terpotong. Dari sisi ruang, Beam Search membutuhkan memori sebesar O(w) per level karena hanya menyimpan beam width state terpilih, jauh lebih efisien dibanding algoritma yang menyimpan seluruh frontier. Overhead tambahan berasal dari penghitungan heuristic untuk setiap tetangga yang dihasilkan dan operasi seleksi menggunakan fungsi seperti nth_element yang memiliki kompleksitas O(n) dengan n adalah jumlah tetangga. Dengan demikian, Beam Search memberikan kompromi antara performa komputasi dan kualitas solusi, menjadikannya pilihan menarik untuk masalah dengan ruang pencarian besar seperti Rush Hour ketika sumber daya terbatas.

Extras

TXT Output

```
void FileProcessor::save(const std::vector<std::string>& moveLog,
                         const std::vector<Board>& history, double time,
                         int nodes) {
  std::string outputName;
  std::cout << "Simpan hasil sebagai (mis. hasil1.txt): ";
  std::cin >> outputName;

  ofstream file("test/output/" + outputName);
  if (!file.is_open()) {
    cerr << "Gagal menyimpan ke file: test/output/" << outputName << endl;
    return;
  }

  file << "Total gerakan: " << moveLog.size() << endl;
  file << "Node dikunjungi: " << nodes << endl;
  file << "Waktu eksekusi: " << time << " ms\n\n";

  for (int i = 0; i < (int)moveLog.size(); ++i) {
    file << "Langkah " << i + 1 << ": " << moveLog[i] << endl;
    const Board& b = history[i];
    for (int y = 0; y < b.getHeight(); ++y) {
      for (int x = 0; x < b.getWidth(); ++x) {
        file << b.getPosition(x, y);
      }
      file << '\n';
    }
```

```
    file << '\n';
  }

  file.close();

  cout << "Hasil disimpan di: test/output/" << outputName << endl;
}
```

Fungsi FileProcessor::save bertanggung jawab untuk menyimpan hasil solusi dari permainan Rush Hour ke dalam sebuah file teks berformat .txt. Proses penyimpanan ini dilakukan setelah algoritma pencarian selesai dijalankan dan telah menghasilkan jalur solusi berupa daftar langkah (moveLog) serta jejak papan (history) yang merepresentasikan kondisi papan setelah setiap langkah.

Pertama, pengguna diminta untuk memberikan nama file output melalui terminal yang kemudian dikombinasikan dengan path test/output/ untuk menentukan lokasi penyimpanan file. Setelah file dibuka menggunakan std::ofstream, fungsi akan mencatat informasi ringkasan di bagian atas file, seperti total gerakan yang dilakukan, jumlah simpul yang dikunjungi selama pencarian, serta waktu eksekusi dalam satuan milidetik.

Setelah bagian ringkasan, file diisi dengan rincian setiap langkah yang dilakukan. Untuk setiap langkah dalam moveLog, akan dicetak teks "Langkah X: <deskripsi>" yang menjelaskan kendaraan mana yang digerakkan, ke arah mana, dan sejauh apa. Di bawahnya, kondisi papan (Board) setelah langkah tersebut dicetak baris per baris, dengan menuliskan karakter pada setiap posisi papan. Ini memungkinkan pembaca file memahami visual perkembangan permainan secara kronologis.

Fungsi ini memastikan bahwa hasil solusi dapat ditinjau ulang dengan mudah di luar program, baik untuk keperluan debugging, pelaporan, maupun dokumentasi. File disimpan dalam format teks yang sederhana dan terbaca manusia, tanpa penggunaan format khusus atau binary sehingga dapat dibuka oleh editor teks apa pun. Jika file gagal dibuka, pesan kesalahan akan ditampilkan ke terminal. Setelah proses selesai, pengguna diberi konfirmasi lokasi file output disimpan.

Executing Test Cases

Test Case #1 (4 Exit Door Possibilities using Random Algorithm)

Input	Output
4 4 1 ..P.	Solusi Ditemukan! Papan Awal: ######

..P. .... K 1 1 **Notes:** Algo: GBFS Heuristic Function: Blocking Counter	#..P.# #..P.# #....# #....# ###K## Langkah 1: P-bawah1 ###### #....# #..P.# #..P.# #....# ###K## Langkah 2: P-bawah2 ###### #....# #....# #....# #..P.# ###P## Papan Akhir: ###### #....# #....# #....# #..P.# ###P## Total langkah: 2 Total simpul dikunjungi: 4 Waktu eksekusi: 0.195138 ms
4 4 1 .... K.PP. .... 1 1 **Notes:** Algo: GBFS Heuristic Function: Blocking Counter	Solusi Ditemukan! Papan Awal: ###### #....# K.PP.# #....# #....# ###### Langkah 1: P-kiri2 ###### #....# PP...# #....# #....# ###### Papan Akhir:

	###### #....# PP...# #....# #....# ###### Total langkah: 1 Total simpul dikunjungi: 3 Waktu eksekusi: 0.255954 ms
4 4 1 K .... ..P. ..P. .... 1 2 **Notes:** Algo: GBFS Heuristic Function: Manhattan Distance	Solusi Ditemukan! Papan Awal: ###K## #....# #..P.# #..P.# #....# ###### Langkah 1: P-atas2 ###P## #..P.# #....# #....# #....# ###### Papan Akhir: ###P## #..P.# #....# #....# #....# ###### Total langkah: 1 Total simpul dikunjungi: 2 Waktu eksekusi: 0.079959 ms
4 4 1 .... .PP.K .... 1 2 **Notes:** Algo: UCS Heuristic Function:	Solusi Ditemukan! Papan Awal: ###### #....# #.PP.K #....# #....# ###### Langkah 1: P-kanan2 ###### #....#

Manhattan Distance	#...PP #....# #....# ###### Papan Akhir: ###### #....# #...PP #....# #....# ###### Total langkah: 1 Total simpul dikunjungi: 4 Waktu eksekusi: 0.172796 ms

Test Case #2 (Test Case Specification using 3 Algorithms (Blocking Counter))

Input	Output
6 6 12 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. 1 1	Solusi Ditemukan! Papan Awal: ######## #AAB..F# #..BCDF# #GPPCDFK #GH.III# #GHJ...# #LLJMM.# ######## Langkah 1: D-atas1 ######## #AAB.DF# #..BCDF# #GPPC.FK #GH.III# #GHJ...# #LLJMM.# ######## Langkah 2: C-atas1 ######## #AABCDF# #..BCDF# #GPP..FK #GH.III# #GHJ...# #LLJMM.#

```
########

Langkah 3: I-kiri1
########
#AABCDF#
#..BCDF#
#GPP..FK
#GHIII.#
#GHJ...#
#LLJMM.#
########

Langkah 4: F-bawah3
########
#AABCD.#
#..BCD.#
#GPP...K
#GHIIIF#
#GHJ..F#
#LLJMMF#
########

Langkah 5: P-kanan2
########
#AABCD.#
#..BCD.#
#G..PP.K
#GHIIIF#
#GHJ..F#
#LLJMMF#
########

Langkah 6: P-kanan2
########
#AABCD.#
#..BCD.#
#G....PP
#GHIIIF#
#GHJ..F#
#LLJMMF#
########

Papan Akhir:
########
#AABCD.#
#..BCD.#
#G....PP
#GHIIIF#
#GHJ..F#
#LLJMMF#
########
```

	Total langkah: 6 Total simpul dikunjungi: 13 Waktu eksekusi: 5.16511 ms
6 6 12 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. 2 1	Solusi Ditemukan! Papan Awal: ######## #AAB..F# #..BCDF# #GPPCDFK #GH.III# #GHJ...# #LLJMM.# ######## Langkah 1: D-atas1 ######## #AAB.DF# #..BCDF# #GPPC.FK #GH.III# #GHJ...# #LLJMM.# ######## Langkah 2: C-atas1 ######## #AABCDF# #..BCDF# #GPP..FK #GH.III# #GHJ...# #LLJMM.# ######## Langkah 3: I-kiri1 ######## #AABCDF# #..BCDF# #GPP..FK #GHIII.# #GHJ...# #LLJMM.# ######## Langkah 4: F-bawah3 ######## #AABCD.# #..BCD.# #GPP...K #GHIIIF# #GHJ..F#

	#LLJMMF# ######## Langkah 5: P-kanan4 ######## #AABCD.# #..BCD.# #G....PP #GHIIIF# #GHJ..F# #LLJMMF# ######## Papan Akhir: ######## #AABCD.# #..BCD.# #G....PP #GHIIIF# #GHJ..F# #LLJMMF# ######## Total langkah: 5 Total simpul dikunjungi: 205 Waktu eksekusi: 66.8196 ms
6 6 12 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM. 3 1	Solusi Ditemukan! Papan Awal: ######## #AAB..F# #..BCDF# #GPPCDFK #GH.III# #GHJ...# #LLJMM.# ######## Langkah 1: I-kiri1 ######## #AAB..F# #..BCDF# #GPPCDFK #GHIII.# #GHJ...# #LLJMM.# ######## Langkah 2: F-bawah3 ######## #AAB...# #..BCD.#

```
#GPPCD.K
#GHIIIF#
#GHJ..F#
#LLJMMF#
########

Langkah 3: C-atas1
########
#AABC..#
#..BCD.#
#GPP.D.K
#GHIIIF#
#GHJ..F#
#LLJMMF#
########

Langkah 4: D-atas1
########
#AABCD.#
#..BCD.#
#GPP...K
#GHIIIF#
#GHJ..F#
#LLJMMF#
########

Langkah 5: P-kanan4
########
#AABCD.#
#..BCD.#
#G....PP
#GHIIIF#
#GHJ..F#
#LLJMMF#
########

Papan Akhir:
########
#AABCD.#
#..BCD.#
#G....PP
#GHIIIF#
#GHJ..F#
#LLJMMF#
########

Total langkah: 5
Total simpul dikunjungi: 37
Waktu eksekusi: 7.88328 ms
```

Test Case #3 (No Solution: Just for Benchmarking the Performance)

Input	Output
6 6 12 AAB..F ..BCDF GPPCDF GH.III GHJ... LLJMM. K 1 1	Solusi tidak ditemukan. Total langkah: 0 Total simpul dikunjungi: 8509 Waktu eksekusi: 36482.9 ms
6 6 12 AAB..F ..BCDF GPPCDF GH.III GHJ... LLJMM. K 2 1	Solusi tidak ditemukan. Total langkah: 0 Total simpul dikunjungi: 8509 Waktu eksekusi: 6771.92 ms
6 6 12 AAB..F ..BCDF GPPCDF GH.III GHJ... LLJMM. K 3 1	Solusi tidak ditemukan. Total langkah: 0 Total simpul dikunjungi: 8601 Waktu eksekusi: 5556.8 ms

Test Case #4 (Breaking the Config)

Input	Output
4 4 1 .... K.PP.K	Pastikan **format file** sesuai dengan yang diharapkan. Jumlah baris tidak sesuai dengan tinggi papan make: *** [Makefile:30: run] Error 1
4 4 1	Pastikan **format file** sesuai dengan yang diharapkan.

.... K.PP.K	Jumlah kolom tidak sesuai dengan lebar papan make: *** [Makefile:30: run] Error 1
4 4 1 .... .PP.	Pastikan format file sesuai dengan yang diharapkan. Tidak ada posisi keluar yang ditemukan make: *** [Makefile:30: run] Error 1

Benchmarking

Input	Output			
	GBFS	UCS	A*	Beam Search
6 6 12 AAB..F ..BCDF GPPCDFK GH.III GHJ... LLJMM.	Langkah: 6 Simpul: 13 Waktu: 6.8282 ms	Langkah: 5 Simpul: 205 Waktu: 53.119 ms	Langkah: 5 Simpul: 37 Waktu: 7.15628 ms	Langkah: 5 Simpul: 31 Waktu: 10.5233 ms Lebar Beam: 6

Penjelasan

- GBFS tercepat dari segi waktu dan simpul dikunjungi, namun solusi yang ditemukan sedikit lebih panjang.
- UCS menjamin solusi optimal, tetapi eksplorasi state lebih besar dan waktu lebih lama.
- A* memberikan solusi optimal dengan efisiensi lebih baik berkat heuristic.
- Beam Search, meskipun menggunakan batasan beam width, tetap berhasil menemukan solusi optimal dengan jumlah simpul dan waktu yang efisien.

Secara keseluruhan, hasil ini memperlihatkan trade-off antara kecepatan, jumlah simpul yang diperiksa, dan optimalitas solusi antaralgoritma. Beam Search menawarkan kompromi menarik antara efisiensi dan kualitas solusi, khususnya untuk konfigurasi dengan ukuran dan kompleksitas seperti ini.

Appendix

Tautan Repositori: https://github.com/fathurwithyou/Tucil3_13523105

Tabel I. Checklist Pengerjaan

No	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	✓	

References

Pearl, J. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984. p. 48.

Russell, Stuart J.; Norvig, Peter (2018). *Artificial intelligence a modern approach* (4th ed.). Boston: Pearson. ISBN 978-0134610993. OCLC 1021874142.