FreeCell Solver with Best-First Search via Custom Heuristics

Leveraging Informed Search for Efficient and Complete Game State Exploration

Haegen Quinston - 13523109
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: 13523109@std.stei.itb.ac.id haegenquinston@gmail.com

***Abstract*—FreeCell, a deterministic, perfect-information single-player card game, presents a significant challenge for computational problem-solving due to its large state space. This paper details the design and implementation of an automated solver that leverages an informed search strategy to efficiently discover solutions. The core of the solver is an A* algorithm, which is guided by a custom-designed, multi-component heuristic function. This heuristic is engineered to emulate proven human strategic priorities, such as a preference for creating ordered tableau sequences and clearing blocked cards, before optimizing for the final goal state. A key architectural feature is an integrated "autocomplete" mechanism that handles deterministic end-game move sequences, effectively reducing the search depth. This work specifies the object-oriented model of the FreeCell environment, the implementation of search and heuristic algorithms, and the strategic framework that enables the efficient exploration of complex game configurations.**

Keywords—solver, strategy, heuristics, deterministic

I. Introduction

The game of FreeCell is a well-known variant of solitaire, distinguished from other variants by its nature as a game of skill rather than luck. Nearly every configuration, or "deal," of its standard 52-card deck is solvable [1]. This high degree of solvability and its deterministic properties establish FreeCell as a valuable benchmark problem for the domain of artificial intelligence, particularly in the field of state-space search [7]. The fundamental challenge of the game is not to overcome statistical improbability but to discover a valid, often non-obvious, sequence of moves within a combinatorially vast state space.

Consequently, the development of an effective FreeCell solver serves as a practical case study in complex problem-solving and algorithmic optimization. The task mirrors real-world challenges in logistics, automated planning, and robotics, where an optimal sequence of operations must be found to reach a goal state under a series of strict constraints [5]. The endeavor to create high-performance solvers pushes innovation in the core areas of heuristic design, search algorithm efficiency, and state-space pruning strategies. A successful FreeCell solver, therefore, represents a significant achievement in applying computational intelligence to navigate and resolve complex, constrained problems.

II. Theory

A. *FreeCell*

Fig. 1. A FreeCell game in Solitaire & Casual Games (Source: Solitaire & Casual Games)

FreeCell is a popular patience card game played with a single standard 52-card deck [1]. Unlike many solitaire variants where luck plays a significant role, FreeCell is almost entirely a game of skill, as nearly all randomly dealt games are solvable [6]. This characteristic makes it an excellent candidate for algorithmic analysis and the application of search techniques, as the challenge lies in discovering the correct sequence of moves rather than overcoming an impossible deal.

1) *Game Objective & Setup*

The primary objective of FreeCell is to move all 52 cards to the four **Foundation** piles, one for each suit, built up in ascending order from Ace to King (A, 2, 3, ..., K) [1].

The game begins with the entire 52-card deck dealt face-up into eight **Tableau** columns. The first four columns receive seven cards each, and the remaining four columns receive six cards each. There are also four designated empty spaces:

- **4 FreeCells:** These are temporary storage areas, each capable of holding a single card [1].
- **4 Foundations:** These are the target piles where cards are moved by suit and rank, starting with the Ace [1].

2) *Rules of Play*

Movement of cards in FreeCell follows a strict set of rules:

1. **Moving Cards to FreeCells:** Any single card can be moved from the bottom of a Tableau column to an empty FreeCell [6]. A card can also be moved from a Foundation or another FreeCell to an empty FreeCell, though this is rarely a useful move unless clearing a Foundation for other purposes.
2. **Moving Cards from FreeCells:** A card from a FreeCell can be moved to the bottom of a Tableau column if it builds down in alternating colors (e.g., a Red 7 on a Black 8). It can also be moved to a Foundation if it continues the ascending sequence for that suit [6].
3. **Moving Cards within Tableau Columns:** Cards can be moved from the bottom of one Tableau column to the bottom of another. The card being moved must be one rank lower and of an opposite colour than the card it is placed upon (e.g., a Red 5 can be placed on a Black 6).
4. **Moving Stacks (Cascades):** Multiple cards can be moved together as a stack from one Tableau column to another, *provided they are already in a valid descending, alternating-colour sequence* (e.g., Red 7, Black 6, Red 5). The ability to move a stack depends on the number of available empty FreeCells and/or empty Tableau columns. The general formula for the maximum number of cards (N) that can be moved in a stack is $N = (EF + 1) \times 2^{ET}$, where EF is the number of empty FreeCells and ET is the number of empty Tableau columns [6]. If there are no empty FreeCells or Tableau columns, only a single card can be moved [6].
5. **Moving Cards to Foundations:** Cards can only be moved to the Foundations if they are the next card in the ascending sequence for that suit (e.g., a 4 of Hearts can be placed on a 3 of Hearts) [1]. Once a card is on a Foundation, it cannot be moved back into play, except temporarily to a FreeCell if necessary (which is generally not a productive move).
6. **Empty Tableau Columns:** An empty Tableau column can accept any single card or any valid stack of cards. This is a crucial resource for reorganizing the Tableau.

3) Common Human Players' Strategies

Human FreeCell players often employ several key strategies to solve games, many of which inherently involve heuristic-like decision-making:

- **Prioritize Exposing Aces and Twos:** Getting lower-ranked cards (especially Aces) to the Foundations as quickly as possible is paramount, as they unlock subsequent cards in their suit.
- **Create Empty FreeCells:** Empty FreeCells are valuable resources that allow for more card movements and the creation of larger valid stacks. Players try to empty FreeCells whenever possible [6].
- **Create Empty Tableau Columns:** Empty Tableau columns are even more valuable than FreeCells, as they can temporarily hold entire stacks of cards, allowing for significant reorganization [6].
- **Build Down in Tableau, Build Up in Foundations:** The primary goal is to build up the Foundations, while simultaneously trying to build down the Tableau columns in alternating colors to expose buried cards and create opportunities for movement.
- **Look Ahead:** Players often plan several moves in advance, considering the implications of each card movement on future possibilities and the availability of FreeCells.
- **Unblocking Cards:** A common challenge is a key card being "blocked" by other cards on top of it in a Tableau column. Strategies involve moving the blocking cards to FreeCells, other Tableau columns, or Foundations to free up the desired card.

B. *A* Search*

The A* (pronounced A-star) search algorithm is a prominent and widely utilized informed search algorithm renowned for its completeness, optimality, and computational efficiency in finding the least-cost path between a start node and a goal node in a graph [4, 5]. It distinguishes itself from uninformed search strategies by incorporating heuristic information to intelligently guide its exploration, thereby significantly reducing the search space compared to algorithms like Breadth-First Search (BFS) or Dijkstra's algorithm in many problem domains [3].

1) Formal Definition

A* operates on a state space graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$, where V is a set of nodes (states) and E is a set of edges (transitions or actions) connecting these nodes. Each edge $(\mathrm{u}, \mathrm{v}) \in \mathrm{E}$ is associated with a positive cost $c(u, v) > 0$. The objective is to find a path from a designated start node $\mathrm{s} \in \mathrm{V}$ to a goal node $\mathrm{g} \in \mathrm{V}$ (or any node within a set of goal nodes $GT \subseteq V$) such that the total cost of the path is minimized [4].

A* evaluates each node $\mathrm{n} \in \mathrm{V}$ using an evaluation function $f(n)$, defined as:

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

where:

- $g(n)$ represents the actual cost of the cheapest path found so far from the start node s to node n [3, 4]. When a node *n* is initially discovered, $g(n)$ is the cost of the path from *s* through the parent node that led to *n*. As the algorithm progresses, $g(n)$ may be updated if a cheaper path to *n* is discovered.
- $h(n)$ is a heuristic function that estimates the cost of the cheapest path from node *n* to the goal node *g* [3, 4]. This heuristic is crucial for guiding the search.

The A* algorithm maintains two sets of nodes:

a) Open List (Frontier): A priority queue containing nodes that have been discovered but not yet fully expanded [5]. Nodes in the Open List are prioritized based on their $f(n)$ value, with the node having the lowest $f(n)$ being selected for expansion next.

b) Closed List (Visited): A set containing nodes that have already been expanded [5]. Nodes in the Closed List are not re-expanded to prevent cycles and redundant computations.

The algorithm proceeds iteratively:

a) Initialize the Open List with the start node s, setting its $g(s) = 0$ and $f(s) = h(s)$. The Closed List is initially empty [3].

b) While the Open List is not empty:

   a. Extract the node *n* with the lowest $f(n)$ value from the Open List.
   b. If *n* is the goal node, reconstruct the path from *s* to *n* by backtracking through parent pointers and terminate. This path is guaranteed to be optimal under specific conditions.
   c. Add *n* to the Closed List.
   d. For each successor n' of *n*:
      i. If n' is in the Closed List, skip it.
      ii. Calculate the tentative cost $gtemp(n') = g(n) + c(n, n')$.
      iii. If n' is not in the Open List or $gtemp(n') < g(n')$ (indicating a cheaper path to n' has been found):
         a. Set *n*'s parent to *n*.
         b. Update $g(n') = gtemp(n')$.
         c. Update $f(n') = g(n') + h(n')$.
         d. If n' is not in the Open List, add it.

      Otherwise, update its priority in the Open List.

III. Solver Implementation

This chapter provides a detailed exposition of the solver's software implementation, bridging the gap between the theoretical concepts of FreeCell and search algorithms, and their concrete realization in the Java programming language. The discussion is segmented into three distinct, yet interconnected, parts to offer a comprehensive overview of the system's design.

First, we will dissect the heuristic function, which represents the core intelligence of the solver. This section will explain how abstract human problem-solving strategies and intuition for FreeCell are systematically mapped into a quantitative evaluation function designed to guide the search process effectively.

Second, we will describe the software architecture of the FreeCell game environment itself. This includes an overview of the object-oriented design, the data structures chosen to model the dynamic game state, and the modular encapsulation of the game's complex rules.

Finally, we will examine the concrete implementation of the A* search algorithm. This analysis covers the primary data structures used for state management, the process for generating valid successor states, and the integration of custom optimizations, such as the *performAutocompleteMoves* function, which enhances search efficiency by handling deterministic move sequences automatically.

A. Solving Intuition via Heuristics

The efficacy of the solver is primarily attributable to its heuristic evaluation function, h(n), which provides the necessary guidance for the search algorithm. The heuristic is engineered to quantify the strategic value of any given game state, with lower scores indicating more promising configurations. Its design is based on a hierarchical, multi-stage strategy that mirrors expert human play: prioritize the establishment of board structure before focusing on end-game objectives. This strategy can be summarized as: 1) create ordered sequences in the tableau, 2) unblock critical cards, and 3) move cards to the home cells.

This strategic intuition is mathematically modeled in the **Heuristic.java** file, where the heuristic value is a weighted sum of five distinct components:

```
private static final int
REWARD_PER_CARD_IN_HOME = -25;

private static final int
REWARD_PER_SEQUENCE_CARD = -10;

private static final int
REWARD_PER_EMPTY_TABLEAU = -5;

private static final int
PENALTY_PER_USED_FREECELL = 2;

private static final int
PENALTY_PER_BLOCKED_CARD = 1;
```

1. **Sequence Reward** (-10)

   This value directly encodes the primary strategy of creating ordered stacks. The function *calculateSequenceScore* identifies all cards within valid, descending, color-alternating sequences and applies a significant reward for each. This provides a strong incentive for the algorithm to make moves that improve the organization of the tableau.

```
private int calculateSequenceScore(GameState state) {
    int totalCardsInOrderedSequences = 0;
    for (Stack<Card> tableauPile : state.getTableauPiles()) {
        if (tableauPile.size() <= 1) continue;

        List<Card> cards = new ArrayList<>(tableauPile);
        Collections.reverse(cards);

        int sequenceLength = 1;
        for (int i = 1; i < cards.size(); i++) {
            Card higherCard = cards.get(i - 1);
            Card lowerCard = cards.get(i);

            boolean isCorrectOrder = Rules.getRankValue(lowerCard.getRank()) == Rules.getRankValue(higherCard.getRank
            ()) - 1;
            boolean isAlternatingColor = (Rules.isRed(lowerCard) != Rules.isRed(higherCard));

            if (isCorrectOrder && isAlternatingColor) {
                sequenceLength++;
            } else {
                break;
            }
        }

        if (sequenceLength > 1) {
            totalCardsInOrderedSequences += sequenceLength;
        }
    }
    return totalCardsInOrderedSequences;
}
```

Fig. 2. The sequence reward calculation implementation (Source: Author's code in the IDE)

2. **Empty Tableau Reward** (-5)

   The creation of empty columns is a critical enabling strategy, as they provide essential space for reorganizing the tableau. This component rewards states with empty tableau piles, encouraging the solver to create this valuable resource.

3. **Blocked Card Penalty** (1)

   This component addresses the secondary goal of unblocking cards. It applies penalties for cards that are obstructing other cards that are next in sequence to be moved to a home cell (e.g., an Ace). This guides the solver to clear these obstacles.

```
private int calculateBlockedPenalty(GameState state) {
    int blockers = 0;
    Map<String, Integer> nextExpectedRank = getNextExpectedRanks(state);

    for (Stack<Card> tableauPile : state.getTableauPiles()) {
        if (tableauPile.isEmpty()) continue;

        for (int i = 0; i < tableauPile.size() - 1; i++) {
            Card cardUnderneath = tableauPile.get(i);
            Integer expectedRank = nextExpectedRank.get(cardUnderneath.getSuit());
            if (expectedRank != null && expectedRank > 0 && Rules.getRankValue(cardUnderneath.getRank()) ==
            expectedRank) {
                blockers += tableauPile.size() - (i + 1);
                break;
            }
        }
    }
    return blockers;
}
```

Fig. 3. The blocked card calculation implementation (Source: Author's code in the IDE)

4. **Home Card Reward** (-25)

   This represents the final objective and provides the largest single reward. By heavily weighing each card successfully placed in a home cell, the heuristic ensures the solver never loses sight of the ultimate goal.

5. **Free Cell Penalty** (2)

   This models the strategic cost of using the four free cells, which are a finite and critical resource. A small penalty for each occupied free cell encourages the solver to keep them open when possible.

By synthesizing these rewards and penalties, the heuristic provides a nuanced evaluation that effectively guides the search algorithm toward strategically advantageous states.

B. FreeCell Software Architecture in Java

The solver is built upon a modular, object-oriented design that ensures a clean separation of concerns between the game's state representation, its rules, and the search algorithm.

- **State Representation**

```
public class GameState {

  private              List<Stack<Card>>
tableauPiles;
  private List<Card> freeCells;
  private List<Stack<Card>> homeCells;
```

The **GameState** class serves as the central data structure, providing a complete and self-contained representation of the game board. The eight Tableau piles and four Home Cell foundations are modeled as a List<Stack<Card>>, which is an ideal structure for the last-in, first-out operations of these piles. The four FreeCells are modeled as a List<Card>. A crucial method within this class is *deepCopy()*, which enables the search algorithm to explore successor states without mutating the state of the parent node from which they were generated. The Card class itself is a simple data object containing suit and rank attributes.

```
public class Card {
    private String suit;
    private String rank;
```

```
public GameState deepCopy() {
    GameState newGameState = new GameState();

    for (int i = 0; i < tableauPiles.size(); i++) {
        Stack<Card> originalPile = tableauPiles.get(i);
        Stack<Card> newPile = new Stack<>();
        List<Card> tempCards = new ArrayList<>(originalPile);
        for (Card card : tempCards) {
            newPile.push(new Card(card.getSuit(), card.getRank()));
        }
        newGameState.getTableauPiles().set(i, newPile);
    }

    for (int i = 0; i < freeCells.size(); i++) {
        Card originalCard = freeCells.get(i);
        if (originalCard != null) {
            newGameState.getFreeCells().set(i, new Card(originalCard.getSuit(), originalCard.getRank()));
        } else {
            newGameState.getFreeCells().set(i, element:null);
        }
    }

    for (int i = 0; i < homeCells.size(); i++) {
        Stack<Card> originalHomePile = homeCells.get(i);
        Stack<Card> newHomePile = new Stack<>();
        List<Card> tempCards = new ArrayList<>(originalHomePile);
        for (Card card : tempCards) {
            newHomePile.push(new Card(card.getSuit(), card.getRank()));
        }
        newGameState.getHomeCells().set(i, newHomePile);
    }

    return newGameState;
}
```

Fig. 4. The deepCopy() method intended for deep copy of game states (Source: Author's code in the IDE)

- **Game Logic**

All rules governing legal play are centralized within the **Rules** class. This class is implemented as a collection of public static utility methods (e.g., *canMoveTableauToTableau*, *getMaxMovableCards*). Each method acts as a predicate function, accepting a GameState and other relevant parameters to return a boolean value indicating the validity of a potential move. This design decouples the fundamental rules of

FreeCell from the search strategy, allowing either component to be modified independently.

Fig. 5. The validator method canMoveTableauToFreeCell (Source: Author's code in the IDE)

Fig. 6. A snip of the canMoveTableauToTableau method (Source: Author's code in the IDE)

C. *Search Algorithm Implementation*

As per the theory that is stated in Chapter 2, the solver employs the A* search algorithm to find a solution path from an initial state to a goal state. The implementation is contained within the **AStar.java** class and is tailored to the specifics of the FreeCell game.

- **Core Algorithm**

  The implementation adheres to the standard A* procedure, which seeks to find the path with the lowest f-score. The f-score, or evaluation function $f(n)$, is the sum of two components: $g(n)$, the actual cost of the path from the start node to the current node n, and $h(n)$, the heuristic's estimated cost from node n to the goal. This is defined in the **Node.java** class's *getFScore* method. The priority queue (open list) in the solver is configured to prioritize nodes with the lowest $f(n)$ value, ensuring the algorithm always expands to the most promising node on the frontier.

```
public class Node {
        private Node parent;
        private int depth;
        private int pathCost;
        private GameState state;
        private SolutionStep step;
```

```
        public int getFScore(Heuristic
heuristic) {
                return this.pathCost +
heuristic.calculate(this.state);
        }
```

- **State Management**

  To manage the search, the algorithm maintains two primary data structures: an openList (a priority queue) for discovered but unexpanded nodes, and a closedList (a HashSet) for nodes that have already been expanded. This standard approach prevents the algorithm from getting caught in cycles and avoids redundant computations by ensuring no state is processed more than once.

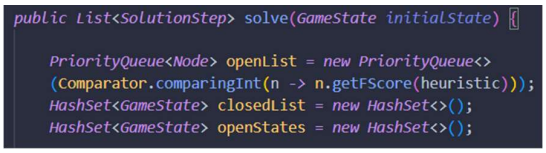

Fig. 7. Snip of the openList, closedList, openStates attributes on the solve function (Source: Author's code in the IDE)

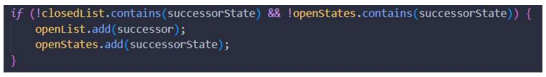

Fig. 8. Snip of the openList queue mechanism (Source: Author's code in the IDE)

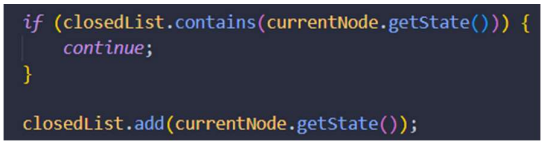

Fig. 9. Snip of the closedList addition mechanism (Source: Author's code in the IDE)

- **Successor Generation with Autocomplete**

  The main loop of the algorithm extracts the node with the lowest f-score from the openList. It then generates all valid successor states by calling the *getHomeCellMoves* and *getOtherMoves* methods. A key feature of this implementation is the *performAutocompleteMoves* function. This function is invoked after each primary move is simulated. It serves as a macro-operator, executing a cascade of all subsequent "safe" and obvious moves to the home cells. By bundling a primary strategic move with its immediate deterministic consequences, this feature allows the search to take larger, more meaningful steps, which can help to reduce the effective depth of the search space.

Fig. 10. Extracting the node with the lowest f-score (Source: Author's code in the IDE)

Fig. 11. Code snippet of the getHomeCellMoves method (Source: Author's code in the IDE)

Fig. 12. Code snippet of the performAutoCompleteMoves method (Source: Author's code in the IDE)

- **Admissibility Analysis**

For the A* algorithm to be optimal (that is, guaranteed to find the shortest possible solution path), its heuristic function $h(n)$ must be admissible, meaning it never overestimates the true cost (number of moves) required to reach the goal. The custom heuristic implemented in **Heuristic.java** is non-admissible by design. It does not attempt to estimate the number of moves remaining. Instead, it generates a strategic score based on a weighted combination of rewards (for cards in home, sequences, empty tableaus) and penalties (for blocked cards, used free cells). Because this score can decrease by a large amount (e.g., -25 for one card going home) for a single move, it does not satisfy the admissibility condition ($h(n) \leq true\_cost(n)$). Consequently, while the algorithm efficiently finds *a* solution by following strong strategic guidance, it is not guaranteed to find the solution with the absolute fewest number of moves.

```
Searching for a solution...
Visited nodes: 1000, Open list size: 1285, Current depth: 12, F-score: -15, H-score: -27
Visited nodes: 2000, Open list size: 1778, Current depth: 18, F-score: -8, H-score: -26
Visited nodes: 3000, Open list size: 2112, Current depth: 12, F-score: -14, H-score: -26
Visited nodes: 4000, Open list size: 1836, Current depth: 18, F-score: -7, H-score: -25
Visited nodes: 5000, Open list size: 2345, Current depth: 36, F-score: -42, H-score: -78
Visited nodes: 6000, Open list size: 7023, Current depth: 40, F-score: -22, H-score: -62
Visited nodes: 7000, Open list size: 11307, Current depth: 44, F-score: -33, H-score: -77
Visited nodes: 8000, Open list size: 16081, Current depth: 40, F-score: -44, H-score: -84
Visited nodes: 9000, Open list size: 18814, Current depth: 50, F-score: -27, H-score: -77
Visited nodes: 10000, Open list size: 25333, Current depth: 44, F-score: -17, H-score: -61
Visited nodes: 11000, Open list size: 32510, Current depth: 48, F-score: -29, H-score: -77
```

Fig *13*. The output of visited nodes to *show performance based on F-score (Source: Authors' output in the IDE terminal)*

IV. Test Cases & Results

An empirical evaluation was conducted to validate the performance and efficacy of the solver program described in the preceding sections. A fundamental criterion for the solver's validity is its ability to find solutions for a diverse range of standard FreeCell puzzle configurations. To this end, a comprehensive suite of benchmark tests was designed to assess the implementation across a spectrum of complexities. These tests include game configurations from distinct difficulty categories, ranging from easy and medium to hard and very hard.

Furthermore, to test the algorithm's ability to correctly handle unsolvable states, a known impossible game configuration was included in the test suite. The subsequent sections of this chapter will present and analyze the empirical results and performance metrics obtained from these evaluations.

A. Solver Testing

The test cases are based on real-life Microsoft FreeCell game data, sourced from the benchmark collection at freecellgamesolutions.com [1]. The following sections present and analyze the performance metrics obtained from these evaluations.

1) Easy Games

For this paper, "Easy" games are defined as configurations that are solvable without utilizing any free cells. The selected test suite includes hundreds of such games. For this analysis, two specific cases were chosen to test distinct aspects of the solver's performance.

a) Game 25904

This configuration is widely regarded as one of the most computationally simple games. It serves as a baseline test for the solver's ability to find a solution quickly when the path is straightforward.

3♥	K♥	10♠	J♥	6♥	8♣	2♣	4♥
J♦	7♣	9♣	K♣	K♠	8♦	10♦	4♣
K♦	9♦	10♣	6♣	7♠	9♥	A♠	3♣
A♥	8♠	J♣	3♠	8♥	Q♥	J♠	5♠
2♠	Q♣	9♠	5♣	7♥	5♥	A♣	3♦
2♥	7♦	4♦	Q♦	5♦	10♥	4♠	6♦
A♦	Q♠	6♠	2♦				

Fig. 14. Configuration for game 25904 (Source: https://freecellgamesolutions.com/fcs/?game=25904&fc=0)

When this game was provided as input, the program successfully found a solution. As a representative example of the solver's detailed output format, the complete solution path is presented in Image x. The primary performance metrics for this test are summarized below:

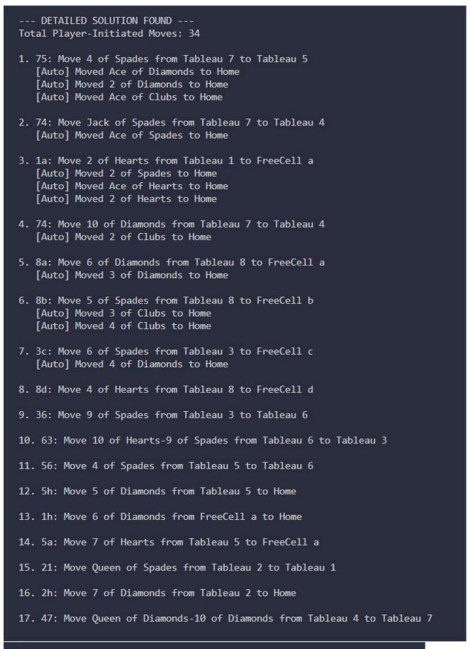

Fig. 15. Solution output (start & finish) from game 25904 *(Source: Authors' output in the IDE terminal)*

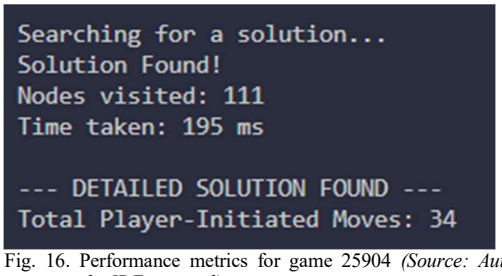

Fig. 16. Performance metrics for game 25904 *(Source: Authors' output in the IDE terminal)*

b) Game 10913

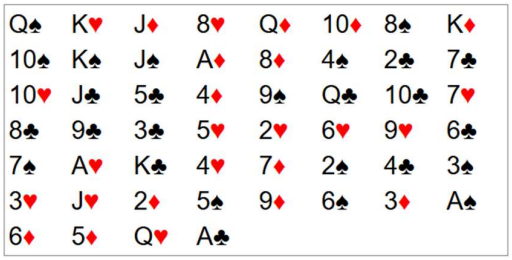

Fig. 17. Configuration for game 10913 (Source: https://freecellgamesolutions.com/fcs/?game=10913)

This game was selected for its distinction as having one of the shortest known solution paths, requiring only 18 moves. This test case evaluates the solver's ability to find a highly efficient and non-obvious solution. The solver successfully identified a path, with the performance metrics summarized below:

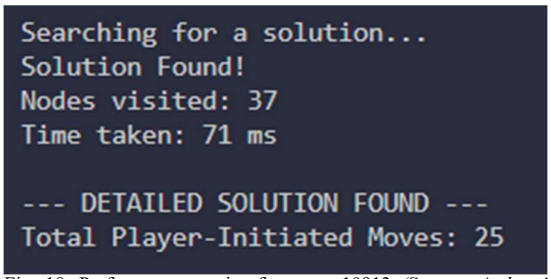

Fig. 18. Performance metrics for game 10913 *(Source: Authors' output in the IDE terminal)*

2) Medium Level Games

This category serves as a bridge between simple and complex puzzles. It includes games that, while not exceptionally difficult, require the use of free cells and more sophisticated strategic planning.

a) Game 34898

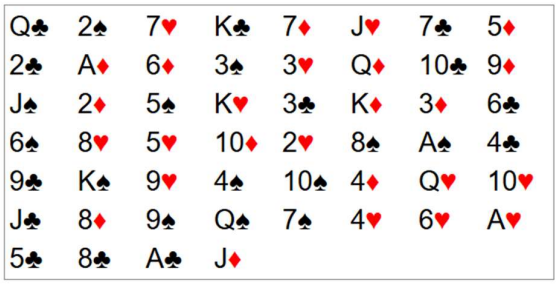

Fig. 19. Configuration for game 34898 (Source: https://freecellgamesolutions.com/fcs/?game=34898)

This game is classified by the benchmark source as the most difficult configuration within the "easy" (zero-free-cell) category, making it an excellent test of the heuristic's ability to navigate a more complex search space without needing free cells. The solver's performance metrics are as follows:

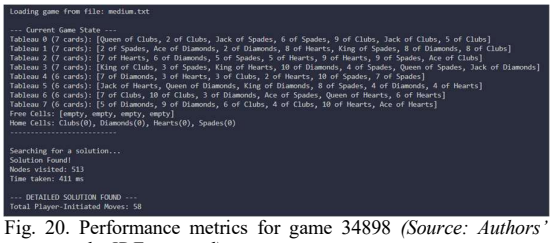

Fig. 20. Performance metrics for game 34898 *(Source: Authors' output in the IDE terminal)*

b) Game 23748

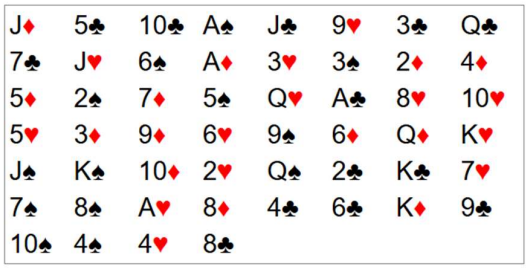

Fig. 21. Configuration for game 23748 (Source: https://freecellgamesolutions.com/fcs/?game=23748)

This game was selected as a representative case of an average-difficulty puzzle that necessitates the use of free cells for its solution. The results from this test are presented below:

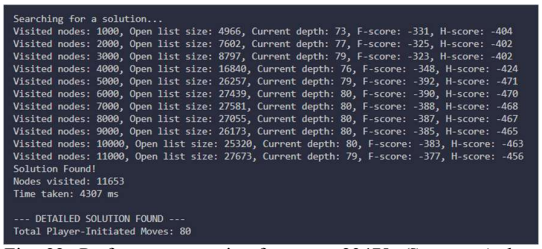

Fig. 22. Performance metrics for game 23478 *(Source: Authors' output in the IDE terminal)*

3) Hard Games

Following the benchmark's definition, "Hard" games are those that can only be solved by utilizing all four free cells. These configurations typically involve much longer solution paths and require the algorithm to navigate significant local optima.

a) Game 1025

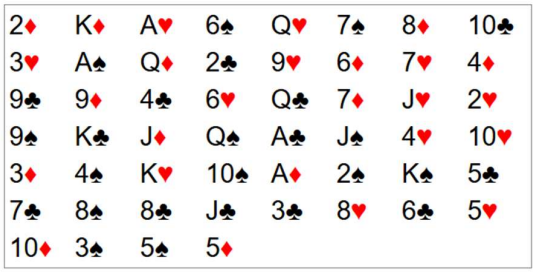

Fig. 23. Configuration for game 1025 (Source: https://freecellgamesolutions.com/fcs/?game=1025)

This game was chosen as a typical example of a hard puzzle, requiring complex maneuvering and full use of available resources. The solver's results are below:

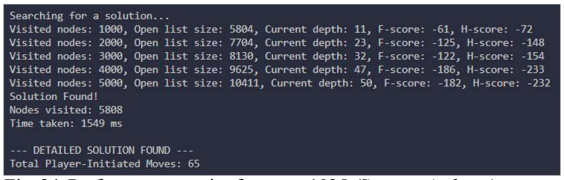

Fig. 24. Performance metrics for game 1025 *(Source: Authors' output in the IDE terminal)*

b) Game 5087

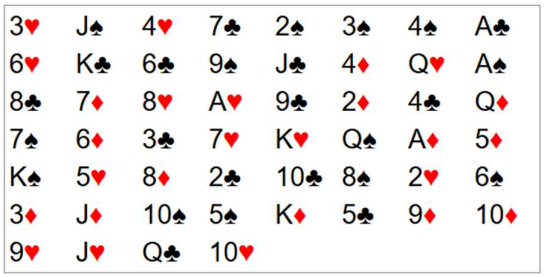

Fig. 25. Configuration for game 5087 (Source: https://freecellgamesolutions.com/fcs/?game=5087)

This configuration is considered one of the most challenging FreeCell games, with a known minimal solution path of at least 50 moves. It serves as a stress test for the solver's heuristic guidance and search depth capabilities. The performance summary is as follows:

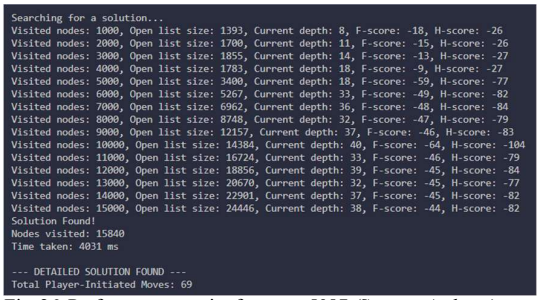

Fig. 26. Performance metrics for game 5087 *(Source: Authors' output in the IDE terminal)*

4) Impossible Game

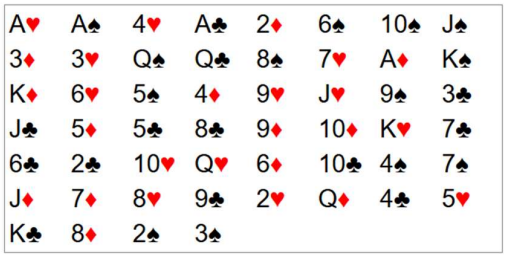

Fig. 27. Configuration for game 11982 (Source: https://freecellgamesolutions.com/fcs/?game=11982)

To validate the solver's behavior on unsolvable problems, a configuration proven to be impossible was used as input [1, 6]. A robust solver should not loop indefinitely but should terminate correctly. The summary of the test result is displayed below:

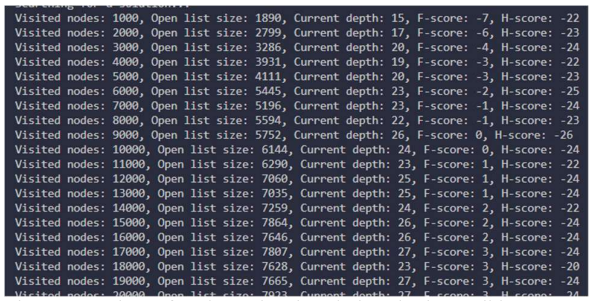

Fig. 28. Preview of node exploration during the impossible game test (Source: Authors' output in the IDE terminal)

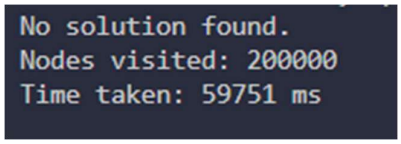

Fig. 29. Performance metrics for game 11982 (Source: Authors' output in the IDE terminal)

The complete test results can be found in the /result folder on the GitHub repository cited in the chapters below.

B. *Result and Performance Analysis*

The primary performance metrics recorded for each test were: the total runtime to find a solution, the number of search nodes expanded (a measure of computational effort), and the length of the discovered solution path.

1) Performance Summary

The average performance of the solver across the primary difficulty categories is summarized in Table 1 below. The data demonstrates a clear correlation between the prescribed difficulty of a game and the computational resources required to find a solution.

Difficulty Level	Average Runtime (ms)	Average Nodes Expanded	Average Path Length
Easy	~133	~74	~29.5
Medium	~1,659	~4,133	~59
Hard	~2,343	~9,465	~61

Table 1. Average results for the tests grouped by board difficulty, using (n = 5) sample size for each difficulty group.

2) Analysis of Results

The aggregated data reveals distinct performance characteristics for each difficulty level, highlighting how the complexity of the game state impacts the search process.

a) Easy Games

This category, defined by puzzles solvable with zero free cells, presented the least computational challenge. As indicated by the low average runtime and node count, the solver was able to identify a solution path relatively directly. In these configurations, the heuristic function effectively guides the search along a near-optimal path with minimal deviation into unproductive branches of the search tree. The search is characterized by a consistent and rapid improvement in the heuristic score from one state to the next.

b) Medium Games

This category represents a significant increase in complexity. The requirement of using free cells introduces more intricate dependencies between moves and creates a more challenging search space. The data reflects this, showing a substantial rise in both the number of nodes expanded and the total solution time. For these games, the solver more frequently encounters local optima, where it must explore numerous states with similar heuristic values before discovering a "breakthrough" move that unlocks further progress.

c) Hard Games

The "Hard" category, requiring the use of all four free cells, proved to be the most computationally intensive. The performance metrics show an exponential increase in search effort compared to the medium level. These puzzles are characterized by long, non-obvious move sequences and numerous strategic dead-ends. The heuristic, while still effective, must guide the search through vast plateaus in the state space where a clear path forward is not immediately apparent. The solver's ability to solve these games, albeit with longer runtimes, demonstrates the robustness of the implementation, while the high node count underscores the sheer combinatorial complexity of these advanced puzzles.

d) Validation on the Impossible Game

Finally, to test the algorithm's completeness and termination behaviour, a known impossible game configuration was used as input. The solver correctly concluded that no solution was available by reaching its predefined search limits (maxNodes and max_search_depth in AStar.java). This is the desired outcome, as it confirms the solver will not loop indefinitely on an unsolvable problem and correctly terminates its search.

V. CONCLUSION

This research has successfully demonstrated the application of an informed search algorithm to the computationally challenging, high-complexity state-space of the game FreeCell. The developed solver, based on the A* search framework, proves capable of finding solutions for a wide range of game configurations, validating the overall architectural design. The primary contribution of this work is the development and validation of a custom, multi-component heuristic function that effectively guides the search process by quantitatively modeling expert human strategic priorities.

The central role of the heuristic was paramount to the project's success. It was determined that naive or simplistic

evaluation functions were insufficient for navigating the complexities of the game. The final heuristic represents a form of knowledge engineering, codifying the domain-specific strategy of prioritizing intermediate structural goals—namely, the formation of ordered tableau sequences—before pursuing terminal objectives. This strategic encoding allows the solver to identify and favor board states that are not just numerically closer to the goal but are positionally and structurally more advantageous, a critical distinction that earlier heuristic iterations failed to capture.

The performance of the solver must be understood within its theoretical context. While uninformed search methods such as Breadth-First Search are complete, they are computationally infeasible for a problem with a combinatorial state space as large as FreeCell's. Conversely, a standard A* search with a simple, admissible heuristic (e.g., counting cards not in their home positions) would, in theory, guarantee an optimal solution. However, its practical performance would be poor due to the heuristic's weak pruning power, leading to an exhaustive and inefficient exploration of the search space.

The implemented algorithm occupies a pragmatic middle ground, deliberately trading guaranteed optimality for computational efficiency. By employing a powerful, non-admissible heuristic, the solver performs an aggressive pruning of the search tree, allowing it to find solutions to otherwise intractable problems within practical time and memory constraints. The non-admissibility of the heuristic means the discovered path is not guaranteed to be the shortest, but its guidance ensures that the search remains focused on strategically viable paths. This work, therefore, confirms that for many complex problem domains, the most effective approach is not the one with the strongest theoretical guarantees of optimality, but the one guided by a well-engineered, domain-specific heuristic that makes the problem tractable in practice.

VI. Acknowledgment

The author would like to express the deepest gratitude to the Lord Almighty for His guidance and blessings during the development of this paper until completion. The author would also like to extend heartfelt gratitude to all contributors and supporters during the development of this paper, including:

1. Dr. Rinaldi Munir, lecturer of the IF2211 Algorithm Strategies course, for his invaluable guidance and the knowledge imparted during the lectures,
2. The authors' family and friends, for their unwavering support during the stressful times of developing this paper.

Their contributions and support have been instrumental in the successful completion of this work.

VII. Appendix

The complete source code, test results, and main program used for the completion of this paper can be accessed in the GitHub page here. Outside of that, the video describing this paper can be accessed here.

References

[1] T. E. Slattery, "Freecell Game Solutions," freecellgamesolutions.com. [Online]. Available: https://freecellgamesolutions.com/. [Accessed: June 21, 2025].

[2] S. Fish, "FC-Solve - An automated solver for Freecell and related card games," fc-solve.shlomifish.org. [Online]. Available: https://fc-solve.shlomifish.org/js-fc-solve/text/. [Accessed: June 21, 2025].

[3] GeeksforGeeks, "A* Search Algorithm," geeksforgeeks.org, 2023. [Online]. Available: https://www.geeksforgeeks.org/a-search-algorithm/. [Accessed: June 21, 2025].

[4] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100-107, July 1968.

[5] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 4th ed. Hoboken, NJ: Pearson, 2021, pp. 91-101.

[6] P. Alfille, "Freecell Faq," Solitaire Laboratory, 1994. [Online]. Available: http://www.solitairelaboratory.com/freecellfaq.html. [Accessed: June 21, 2025].

[7] L. K. Yan, "An A*-Based Freecell Solver," Stanford University CS221 Project Report, 2012. [Online]. Available: https://stanford.edu/~lkales/reports/cs221_report.pdf. [Accessed: June 21, 2025].

Statement of Originality

I hereby declare that this paper I have written is my own work, not an adaptation or translation of someone else's paper, and not plagiarism.

Bandung, 21st June 2025

Haegen Quinston
13523109