So you want to solve Freecell ?
An academic journey into crafting a solitaire game solver.

UNIVERSITÉ
DE NAMUR

Master Thesis

Julien Castiaux
Supervisor: Jean-Marie Jacquet

June 2021

Contents

Foreword

When I was 21 I decided to both work full time in the industry as a software developer and to complete my bachelor degree with a master degree by attending night classes. I was hoping to learn a ton of new stuff, to be challenged and to spend some of my spare time wisely.

I couldn't imagine it would be *that* challenging. The classes were great, I enjoyed attending them. I enjoyed taking a few weeks off from work to study and my exams too; it may sounds crazy but it was actually quite relaxing. On the other hand, I hated every single assignment I received. Not that the assignments were unfair, too hard or not interesting. The problem was they were very time consuming. Time is a scarce resource when you leave home at 8 AM to only come back at 10 PM two to three times a week.

This work was no exception: it was very time consuming.

Nevertheless I would like to thank both the staff and the teachers from the Computer Science faculty of the University of Namur. Despite the challenge it is to both work and study, the humanity you all show to the students is very much appreciated. I keep excellent memories of my time spent at the university: the courageous people I met and worked with, the classes I've been taught by even more courageous teachers and the university staff who is very committed too. Thank you.

This work is about artificial intelligence and card games, I hope you enjoy it.

Introduction

Freecell is a solitaire card game. All cards are randomly dealt face visible over 8 columns (called cascades), the first four columns have 7 cards, the last four have 6. There are 4 additional free slots (called freecells) where it is possible to temporary store one card. There are 4 empty home foundations, one for each suit. The goal of the player is to complete the 4 suits in the foundation, sorted from ace to king by only moving one card at a time and by following a few other rules.

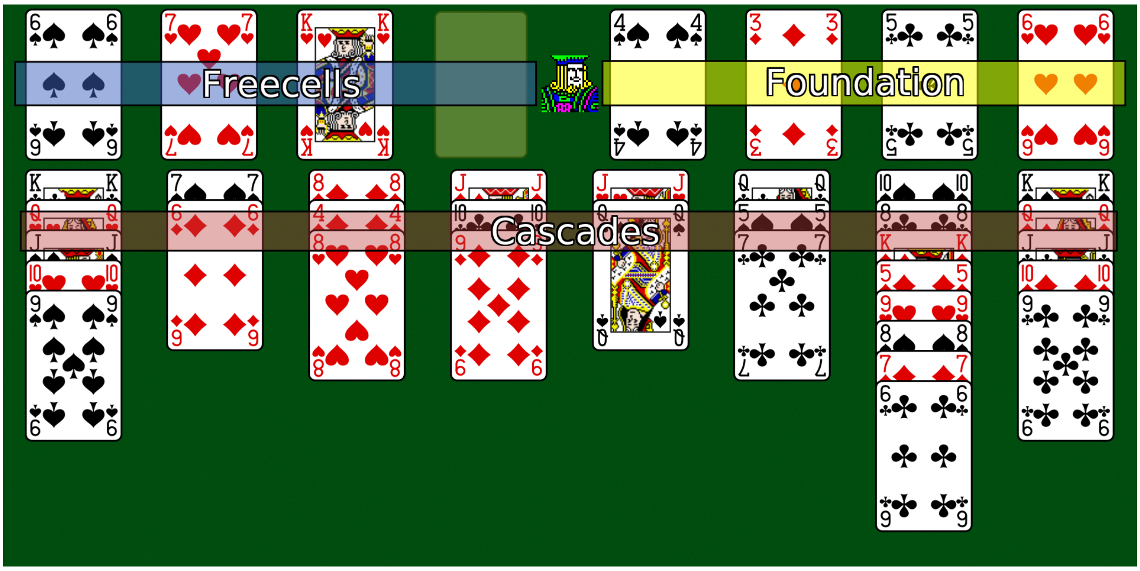

Figure 0.1: A game of freecell after a few moves

The player can only move a card stored in any of the freecell or a card at the bottom-most of a cascade. He can only move that card to an empty freecell, to an empty column, to another column if the bottom-most card is of opposite colour and of value + 1 or to his home foundation if it continues the sorted suit.

The game was invented during the 20th century and is part of the Microsoft Solitaire Collections card games along with Klondike (also known as “Solitaire”) since Windows XP. The XP version had about 32.000 different deals and there are millions since Windows 7.

From a computer science standpoint the game is quite interesting. The game is hard, there are about 52! different initial deals, about 12 possible moves at each step and, on average, 120 moves are required to solve the puzzle. Because of the combinatorial explosion, simple techniques such as naive tree searches are unfeasible. On the other hand, the game has a few key properties that makes it quite elegant. Like Checker, Chess and Go but unlike Klondike, Minesweeper or Othello, Freecell is a game of perfect information. It means that the player, before making any decision, is perfectly informed of all the events that have previously occurred. There is no secret, no hidden card, no randomness, no luck. As the game is a single-player game, the player only decides what steps to perform. He does not compete with any other uncontrollable player who would try to defeat him. Another property of the game is that, in his classic 52 cards, 8 cascades, 4 freecells configuration, more than 99.999% of deals have a solution[23]. From the original Windows XP 32k games, only game numbered #11982 has been documented as unsolvable, all the others having documented solutions. Finally, it is possible to change the configuration to soften or harden the game, with more (less) columns and freecells, the game is easier (harder). Dealing with less cards (e.g. only from ace to 8) makes the game faster with a shorter solution.

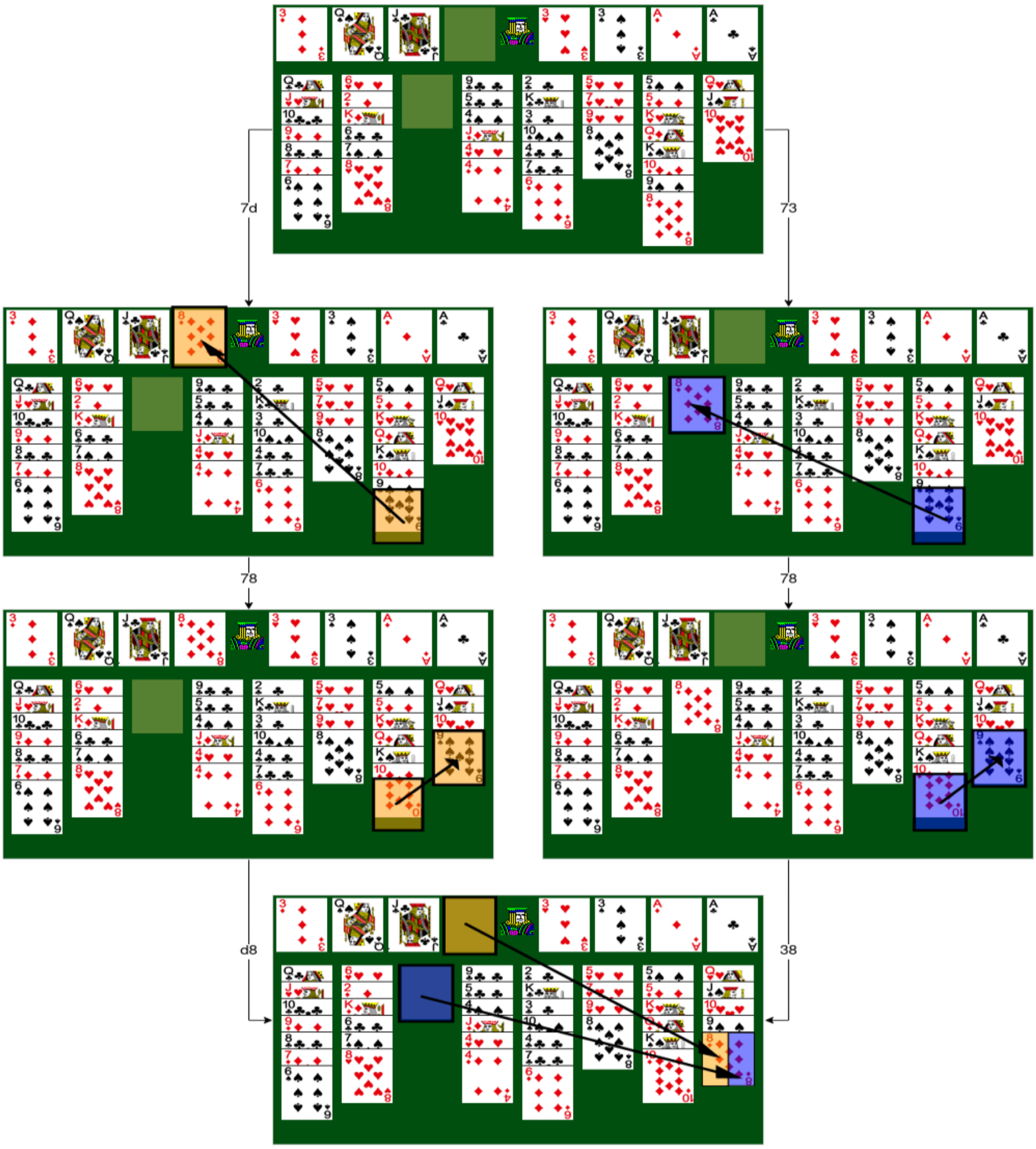

Figure 0.2: Move 8 of diamond and 9 of spade onto 10 of heart

The paper is constructed in three main chapters each divided in several sections. The first chapter is a state of the art for game solvers, we will explain how we screened for papers, describe a few games and tackle the many techniques used by solvers to play the games. In the second chapter we will describe Freecell, the game this work is about. In the third chapter we will design two solvers in regard to two techniques we judged applicable to Freecell and discuss the results obtained with our own implementation.

Chapter 1

State of the Art

1.1 Literature review

The first step of our study was to find an exhaustive survey in the field of artificial intelligence applied to game solvers. The purpose was to quickly identify and name the many existing techniques for further reading.

We selected the excellent work of Bruno Bouzy and Tristan Cazenave: "Computer Go: An AI oriented survey" [7], a 50 pages paper published in 2000 where the authors study in depth many approaches used to play the game of Go. The authors start by explaining the rules of Go and showcase a few common strategies used by human players. They also introduce several two-player perfect information games such as Chess, Checker or Othello and compare their complexity with Go. After the introduction, they systematically describe every technique used in two-player game solvers and assess their results using existing implementations.

The paper has an important bibliography of 149 references, all of them directly cited in the text. Using this survey as a starting point, we selected 42 references for further reading. In the conclusion of the survey, the two authors cite two promising reinforcement learning techniques to solve games: Temporal Difference and Monte Carlo. Because we lacked information about those two techniques we included the book by Richard S. Sutton and Andrew G. Barto: "Reinforcement Learning - An Introduction" [38] to our readings. The book is very complete and explains in depth the key ideas behind autonomous learning. It demonstrates the relationship between exploration (to gain knowledge) and exploitation (of that knowledge), and it also explains how we can quickly gather knowledge by using state-action-reward mechanisms and various algorithms. While the book is great to understand in depth the basis of learning, the many algorithms it showcases all share a hard requirement that is impossible to fulfil in the game of Freecell. Details are given in the *Learning* section.

From our initial readings, the following terms have been identified (in any order): tree search, heuristic search, minimax, alpha-beta pruning, iterative deepening, transposition table, proof-number search, mathematical morphology, computer vision, neural network, Monte Carlo, planning, temporal difference, knowledge acquisition, best-first, A*, IDA*, reinforcement learning.

Even if the initial readings were fruitful to understand most of those techniques, the papers were quite old (published before 2000) and some domains related to learning (neural network, genetic programming and reinforcement learning) were not correctly addressed. In order to fill the gaps, we used Research Gate and Google Scholar to search for recent papers about the techniques mentioned above. We also searched for papers about single-player games such as Freecell, Klondike and Rush-Hour.

As we were efficient to discover interesting papers via snowball and via search engines, we did not perform a rigorous systematic review. Today our bibliography is strong of about 60 papers in the domain of artificial intelligence and game solvers. We are confident that we read the majority of papers about informed tree search algorithms. We are also confident that we *did not* read enough about solvers using deep-neural-network techniques like AlphaGo. The reason we did not learn much about those techniques is that classic reinforcement learning methods are not applicable to Freecell.

The chart in Figure 1.1 organises the papers according to (X-axis) their category, (colour) if they were published for a single-player solver, a two-player solver or outside of a game solver. Most of the papers at our disposal are about Tree Searches, under this category are papers about naive tree traversal, selective deepening, minimax, proof-number, best-first, etc. We only have a very few papers about Goal Generation as this technique has only been studied in the specific case of Goliat, a strong Go solver. As compared to classic tree searches, the several learning techniques have not been used widely in game solvers yet.

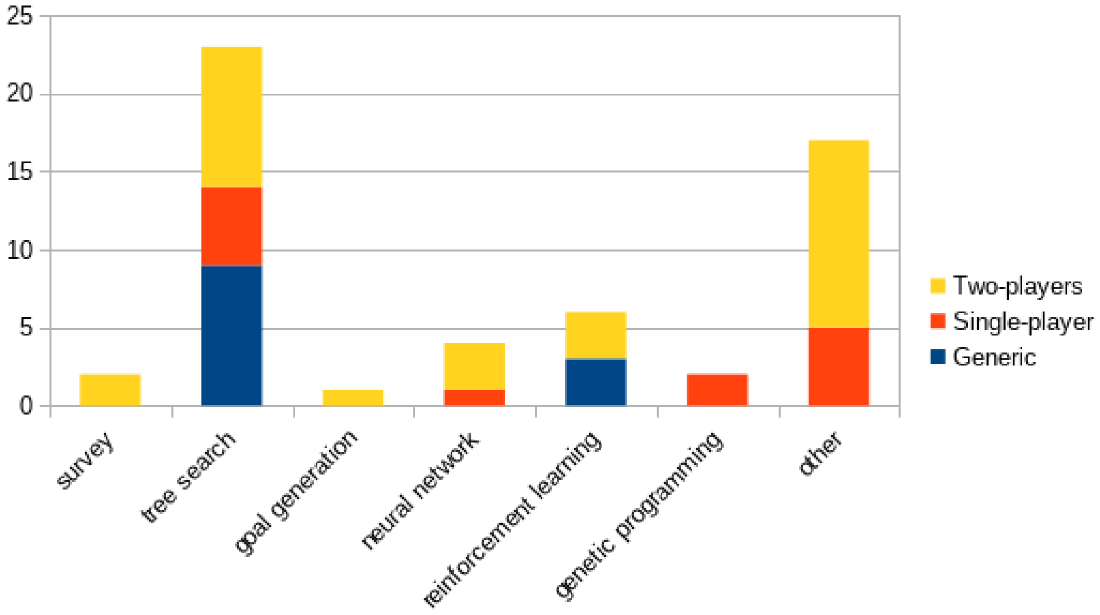

Figure 1.1: Paper classification chart

The state of the art is constructed as follow: (i) we first enumerate a few games and explain how they are generally solved, (ii) we explain the graph data structure and describe three generic algorithms to traverse a graph, (iii) we dive into the world of informed tree searches, the most commonly used technique to write a solver, (iv) we quickly describe a few two-player only solvers, (v) with goal generation we study how to exploit a lot of specific game knowledge in order to strategically select moves, (vi) ultimately, we study two learning approaches: genetic programming and reinforcement learning so the machine learns how to play the game optimally by itself.

1.2 Game solvers

For more than 70 years, since the beginning of computers, scientists have used games as a playground for artificial intelligence. Many games have been studied since then with various degree of success [32]. During the nineties, thanks to the increase in computational power, effective tree search algorithms and carefully crafted heuristics, the machines have been capable of defeating best human players in some popular two-player board games. Lately, with advances in machine learning via methods like neural networks, reinforcement learning or genetic programming, the machines no longer play like human players but, instead, developed their own strategies and ways to play games [16, 37].

Two-players

Computer Chess is the field of artificial intelligence dedicated to study and elaborate artificial agents capable of playing the game of Chess. This field is studied since 1950. Turing described the rules as simple enough to be coped by a machine, yet the game is in its whole complicated enough to be challenging [12]. Shannon described the Minimax algorithm, a general algorithm for two-player games which is still in use today [35]. Half a century later, in 1997, the IBM machine DeepBlue [18] was the first to defeat the human Chess world champion in a highly publicised match.

The techniques used in DeepBlue were not much more sophisticated than they were in the early days of Computer Chess. The general idea in the fifties was to evaluate many moves and to select one that gives promising results; the ideas in the nineties were globally the same. The breakthrough was possible thanks both to advances in tree search algorithms and better hardware. “Bad moves” got rejected much faster and the machine was capable of evaluating much more moves. Not only Chess could benefit from these advances: other game solvers like Chinook [33, 34] for Chess and Logistello [8] for Othello used the same approaches and both respectively defeated the human world champion in 1992 and 1994.

In the beginning of the 21$^{\text{th}}$ century, while most two-player board games were solved or were near to be solved, the best artificial players were still nowhere close to defeat human players at both Shogi and Go, two board games famous in eastern Asia. The breakthrough occurred fifteen years later when the Google machine AlphaGo [37] was the first to defeat the best European Go player. The programmatic model used by AlphaGo is very different from the one used by DeepBlue. While DeepBlue was mostly crunching numbers in a way dictated by his programmers, AlphaGo learned to play Go mostly by itself. Two years later, the same program learned to play Chess and Shogi and was capable of defeating their respective human world champions [36].

Since this study focuses on single-player games, we recommend to read the second chapter “Other games” of “Computer Go: An AI oriented survey” (2001) by Bouzy and Cazenave [7] to learn more about game solvers for two-player board games.

Single-player

Aside from two-player board games, many single-player puzzle games are studied too. Puzzles are appreciated by the artificial intelligence community as they provide good benchmarking environments for a wide range of algorithms [13]. The techniques used to solve games can later be generalized to solve other, more practical, problems [17].

Maze

An example is a maze, an entity is trapped inside a maze and must find its way to the exit. The techniques used to find the exit, and thus solve the maze, can be generalized to path-finding algorithms; algorithms later reused in integrated circuits to automatically link various electronic components [9]. A maze is easily captured in a program, yet it is a rich environment to test various search algorithms [30]. A maze can be generated with or without loops, a maze with (without) loops can be captured in a graph (tree) structure. Various algorithms can be implemented to solve the maze depending on various requirements such as the memory available, the time available or how short the solution must be. When the memory is not a problem and when one requires the shortest path to the exit, *breadth-first* search can be implemented. When one wants to find a somewhat short solution in a timely manner, *best-first* search can be implemented.

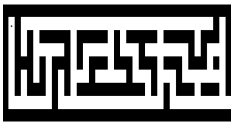

Figure 1.2: Maze

15 puzzle

Next to a maze, more complicated puzzles exist. The 15 puzzle (also called Game of Fifteen or Mystic Square) is one of them. It is a game where 15 square tiles numbered from 1 to 15 are placed in a frame that is 4 by 4 tiles. One tile is left unoccupied, it is possible to slide one orthogonally adjacent tile in it, “moving” the unoccupied tile. The goal of the player is, starting in a position where tiles are shuffled, to sort the tiles on the frame by sliding one tile at a time. This puzzle is interesting as it needs cleverer algorithms

Figure 1.3: Mystic square

to be solved. The definition of the game tells that there are 2, 3 or 4 possible moves at each step. It is possible to soften or harden the game by respectively decreasing or increasing the frame dimensions. Solving the puzzle in an efficient manner requires to guide the search algorithm so it tests interesting moves first. In this game, an interesting move is for example a move that increases the sum of tiles at their correct place. Such moves are not always leading to a solution; an algorithm cannot just greedily use that heuristic and expect to solve the puzzle.

Rubik's cube

Another puzzle that uses similar heuristics is the Rubik's cube. It is a 6x3x3 cubes (6 faces, 3x3 tiles per face) where all tiles on a same face share the same colour when the puzzle is solved. Each face can be rotated by 90, 180 or 270 degrees which affects the faces above, right, bellow and left to the one rotated. When the cube is shuffled, the tiles are moved from face to face and the colours no longer match.

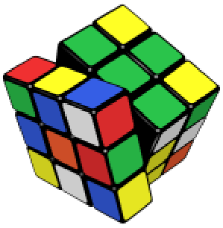

Figure 1.4: Rubik's cube

The goal of the player is to rotate the faces in a way that all tiles are replaced in their original configuration, to rotate the faces so the colour re-match. The difficulty of this puzzle is that each rotation of one face affects 4 others. The Rubik's cube is an example of a game that has been solved thanks to applied mathematics. There exists a strategy that, given that the cube has been placed to an initial position where at least 5 orthogonal tiles are correct on a same face, solves the puzzle systematically. The strategy consists of multiple sequences of rotations, each sorting the cube a bit more without violating the previously sorted one. This strategy requires up to 100 moves to solve the puzzle [26].

In artificial intelligence, we are not interested in using this systematic strategy to solve the puzzle. We are interested into solving the puzzle using as few rotations as possible. We will be interested in Korf's solver [26] that uses a technique known as *Iterative Deepening A**. This solver is capable of solving the cube with a mean of 18 rotations. While this is impressive compared to the 100 rotations required by the systematic solution, Korf's solver is of magnitudes slower than the systematic solution.

Solitaire card games

Solitaire card games like Klondike, Freecell and Bristol are also interesting puzzles that the artificial intelligence community uses to test various algorithms. The purpose of the three games is, starting with a random distribution of a standard 52-card deck, to build up four stacks of cards starting with Ace and ending with King in the so-called *foundation*.

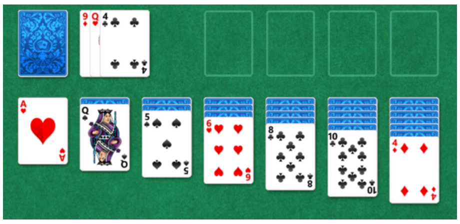

Figure 1.5: Klondike

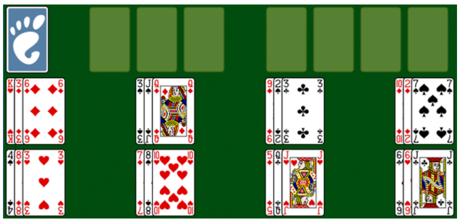

Figure 1.6: Bristol

The rules of the three games vary in term of deal : (i) in Freecell, all cards are dealt face visible over 8 columns, (ii) in Bristol, only 3 cards are dealt face visible over 8 columns, the other cards are accessible via the *talon*, (iii) in Klondike, only 28 cards are distributed over 7 columns (1 card in the first column, 2 in the second, . . . , 7 in the 7th column) but only the bottom-most card is visible, the other cards are accessible via the *talon*. There exists a Klondike variant named Thoughtful where all cards are distributed face visible.

The three games are also different when it comes to allowed moves. In Klondike and Freecell, the player must alternate colours in the cascades and he must stack cards according to their suit in the foundation. In Bristol, none of those two rules stand. In Klondike the player is allowed to supermove (moving multiple cards from the same column at once) while it is forbidden in both Freecell and Bristol.

Game	Talon	Hidden cards	Colour & suit rules	Supermove
Klondike	yes	yes	yes	yes
Thoughful	yes	no	yes	yes
Freecell	no	no	yes	no
Bristol	yes	yes	no	no

Like other board games and puzzles, various artificial intelligence algorithms are applicable to solve those three solitaire card games. Techniques like tree search [5] and genetic programming[14, 15] share interesting results.

We introduce our first research question:

> **RQ1**: "How can the Freecell solitaire game be solved using artificial intelligence ?"

In the following chapters we will dive into the actual meaning of the various techniques introduced here. We will first introduce various Tree search algorithms as a way to simulate many moves in order to select one that leads to a better situation. We will continue with pattern matching and expert systems, where the machine does no longer simulate moves and line of plays but instead studies the current situation to decide what to perform next. We will conclude with multiple learning algorithms, multiple ways in which the solver learn to play optimally the game by itself.

1.3 Naive Tree Searches

The games we are studying, whether they are single or multi-player can all be captured in a graph structure. A graph is a structure composed of nodes interconnected via edges. In our case of board and card games, each game state is a node in the graph, each movement allowed by the rules of the game is an edge connecting one node to another, i.e. one game state to another. In many games, if a move from state 1 to state 2 is allowed, the inverse move from state 2 to state 1 is forbidden. An example is the role of the Pawn in a Chess game: the Pawn can always move to the cell just above it (as long as that cell is empty) but it can never move to the cell just bellow it according to the rules of the game. This kind of restrictions qualifies the graph as directed: the edges have a direction (an origin node and a destination node) and it is not permitted to follow an edge backwards from the destination node to the origin node. Another property of graphs is that it is often possible to reach a distant node (one that is not directly connected) following different paths. Figure 0.2 is an example, starting in the same state, it is possible to follow two different paths to reach the other state. If it was only possible to connect any pair of nodes with one unique path, the graph would have been qualified as a tree; this is not the case here. A final property is that it is often possible to loop in the game by doing a series of moves that ultimately resets the player in the same original state. An example in Freecell is to be in a situation with two empty cascades and to move one card back and forth between those two cascades: moving a card to an empty column is permitted but such a cycle is pretty pointless. Such graphs are qualified as cyclic. The games we are studying all share the qualification of directed cyclic graph.

Depth-First and Breadth-First

There exists various ways to browse a graph. The two most common ways are Depth First (DF) and Breadth First (BF). The Depth First graph traversal algorithm keeps exploring a path until it reaches a dead-end, i.e. a node without edge or whose all edges lead to nodes already explored. When it reaches a dead-end, the algorithm backtracks to the previous node and explores another path. The Breadth First graph traversal algorithm explores first all nodes directly connected then explores all nodes connected to them and repeat until all nodes are in dead-ends. The diagram bellow shows the order in which the nodes are traversed, first (left) using the DF then (right) using BF. In all our diagrams, edges are directed from top to bottom (`0 -> 1`, `0 -> 8` in the bellow BF diagram) and left to right (`6 -> 7`).

```
  Depth First     Breadth First

      (0)              (0)        <- roots
      / \              / \
    (1) (8)          (1) (2)       all edges are
    / \   \          / \   \       directed up to down
  (2) (5) (9)      (3) (4) (5)     and left to right
  / \   \ / \      / \   \ / \
(3) (4) (6)-(7)  (6) (7) (8)-(9)  <- leaves
```

Implementation wise, DF is interesting as it is very cheap in memory. It only requires a single stack to traverse the graph. On the other hand, because it needs to remember all nodes of the previous layer, BF uses a lot of memory.

While it is interesting to know about graph traversal algorithms, they are not very useful by themselves when it comes to solving games. Remember that our goal is to find a solution (in single-player games) or to defeat the opponent (in multi-player games) by carefully selecting the right moves throughout the game. In our previous diagrams, say the bottom right node (node n°7 in the first graph, node n°9 in the second) is the solution to our puzzle like a solved board in Freecell or a checkmate in Chess. The objective is to find that node and to determine how to reach it.

A graph search algorithm is an algorithm that traverses a graph looking for a precise node (or in our case, a node with a given property: game won) that is also capable of determining how to reach that node. In that matter, according to our previous diagram, the two algorithms are different. DF finds a solution quicker than BF (7th visited node in DF vs 9th in BF) but it also finds a solution that is longer (4 moves: `0 -> 1 -> 5 -> 6 -> 7` in DF vs 3: `0 -> 2 -> 5 -> 9` in BF). DF does not guarantee to find a solution quicker than BF: if the solution was on the second node of the second row (node n°8 in DF, n°2 in BF) then DF would have required a lot more time than BF to find it. On the other hand, BF does have the guaranty to find the shortest solution.

Iterative-deepening

The two algorithms can be combined into in the Iterative Deepening Depth-First Search (IDS) algorithm. IDS works like Depth-First Search (DFS) with the exception that it can only search up to a maximum depth. When the algorithm does not find the node it was searching for, it re-searches from the beginning with an extended maximum depth. The diagram bellow shows an example of the Iterative Deepening algorithm applied to tree traversal. The empty nodes are the nodes that the algorithm did not visit during this iteration, the nodes with a letter are the new nodes discovered during this iteration sorted by discovery order, the nodes with a number are the nodes traversed this iteration sorted by traverse order.

```
  Iterative Deepening Search

      (0a              (0)              (0)              (0)
      / \              / \              / \              / \
    ( ) ( )          (1b (2c          (1) (4)          (1) (8)
    / \   \          / \   \          / \   \          / \   \
  ( ) ( ) ( )      ( ) ( ) ( )      (2d (3e (5f      (2) (5) (9)
  / \   \ / \      / \   \ / \      / \   \ / \      / \   \ / \
( ) ( ) ( )-( )  ( ) ( ) ( )-( )  ( ) ( ) ( )-( )  (3g (4h (6i-(7j
```

What seems to be a waste of computational resources, to re-explore all the previous stages of the graph, is actually not really a problem. Most nodes are located on the last layer, which is the one being the most resource greedy to compute. The higher the branching factor, the more the last layer is resource greedy compared to the previous layers. It is proven [27] that the time complexity of an Iterative Deepening traversal is the same as the two classic Depth-First and Breadth-First traversal. Since it uses Depth-First internally to traverse the graph it only requires a single stack, thus making it memory efficient. Since it iterativaly extends the maximum depth, it finds the shortest path to the node it is searching for.

Although it is often described in the literature, extra attention must be given to the properties of the graph when using iterative deepening. When the graph is not a perfect tree, the algorithm shares the same problems as DF: there is no guarantee that we do not visit some nodes multiple times [24]. The figure aside showcases a minimalist directed cyclic graph (DCG) where the node 3 is accessible from three nodes: 1, 2 and 5. When traversing the graph using DF and a maximum depth of 3 nodes, the node are traversed with the following sequence: `0, 1, 3, 4, 5, 2, 3, 4, 5`. Notice that the nodes 3, 4 and 5 are traversed twice, once via `0 -> 1`, once via `0 -> 2`. The problem gets worse when we traverse the graph with an extended depth due to the cycle between the nodes 3 and 5, the algorithm gets stuck in an infinite traversal loop, looping on the sequence `3, 4, 5, 3, 4, 5, ...` (the node 4 is visited because we visit the nodes on the left first).

```
  (0)
  / \
(1) (2)
  \ /
  (3)<--\
  / \   |
(4) (5)-/
```

When traversing such graphs, it is mandatory to ensure that the algorithm does not visit nodes that have already been traversed. On the other hand, when searching for the shortest path to a node using iterative-deepening, one must re-explore a portion of the graph when it finds a shorter path to an intermediary node. This case is shown in the minimalist graph aside, the distance between nodes 0 and 3 is 2 (`0 -> 2 -> 3`) but if one first explores the left path then it reaches the node 3 at depth 3 (`0 -> 1 -> 2 -> 3`) which is not the shortest path, when it explores the right path it needs to re-visit node 2 because the current path (`0 -> 2`) is shorter than the previous path (`0 -> 1 -> 2`), same goes for the node 3.

```
  (0)
  / \
(1)-(2)
    /
  (3)
```

The exact condition is to not re-explore nodes that have a shorter distance to the origin than the current distance to the origin.

Often, iterative deepening is not used alone but rather in pair with another informed tree search algorithm like selective deepening [2–4], alpha-beta [7, 10, 25] or IDA*[15]. Iterative deepening is not used to find the shortest path to a precise node. It is used as a way to explore close nodes (i.e. nodes with a somewhat short distance to an origin node) first and to expand the search when necessary.

1.4 Informed Tree Searches

Naive tree searches are virtually never used to solve games. Remember that each node is a possible game configuration and that edges are valid moves between configurations. A tree search just simulates moves in order to discover many configurations. The problem is, even the simplistic game Tic-tac-toe has a game graph that is too big to be explored efficiently by naive tree search algorithms [28].

A game of Freecell starts with a random deal of 52 cards. On average there are 12 possible actions per game state and, on average, humans solve the game in 120 actions. To explore the entire graph it would be necessary to explore 12^{120} nodes. A solver able to explore 1 million nodes per second (like Deep Blue is able for Chess [7]) would need more than 10^{100} times the age of the universe to explore all the possible 120 first moves. The maths are just cruel: with the branching factor and the average length of a solution in Freecell, it is impossible to solve it using naive tree searches [13, 25].

The problem with naive tree searches is that they do not use any kind of knowledge of the game. They do not exploit the mathematical properties of the games to explore winning strategies first. From a scientific standpoint, a winning strategy is a policy that maps game states to actions in order to select moves in a way to maximize the player's chances of winning [38]. Such policies can be determined by carefully crafting heuristics from the internal game logic. They can also be determined with the help of game experts or via machine learning.

Best-first search

Best-first search is an informed graph traversal algorithm. Starting in an initial node, it explores the nodes directly accessible via an edge and evaluate every node. The node that is evaluated to be the best is selected and used as a new initial node.

To evaluate the next nodes, best-first search uses an evaluation function that takes a game state as input and returns a score that describes "how good" is the state. In a game like Chess, an evaluation function computes a score using multiple properties like how mobile are the pieces, how many pawns are left, how secure is the king or who controls the centre [7]. In a maze, if we know both our current coordinates and the coordinates of the exit, an evaluation function could be our bird's-eye distance to the exit [39].

If the evaluation function was to be applied to every possible game state, then the states where the game is won should all have the best score, the states closer to victory should have a better score than states further from victory, themselves having a better score than states closer to the defeat. In other words, a perfect evaluation function sorts all the game states by "winnability", it orders them by how close to the victory they are.

Often the function fails to compute a score that precisely ranks all states. The bird's-eye distance in a maze is an example of an evaluation function that uses an heuristic that is too greedy. It is possible to generate a maze where that heuristic completely fail, a maze where best-first would traverse all nodes before finding the exit.

A counter example is the 15 puzzle introduced before: the game can be greedily solved with best-first and an evaluation function. The evaluation function counts how many tiles are in the correct place. The moves that place the tiles at their correct place are prioritized, which ultimately solves the puzzle [20].

Multi-step exploration

Best-first performs a greedy one-step look-ahead which means that it only explores the game states immediately accessible from the current one to decide which action to perform. As explained above, this strategy can lead to dead-ends and local-optima [5]. A better solution is to perform a multi-step depth-first look-ahead to *explore* the graph and *gain in knowledge* before *exploiting* that knowledge to decide which action to perform.

The figure below showcases the issue, each visited node has been given a score by the evaluation function. By looking only one node away, one would select the left branch as it leads to a better state than the right one. By looking two nodes away, the left path is discarded as it only leads to a dead-end, a node where the game is lost.

```
      (*)  .    <- current node
      / \ .
 .  (9) (2)  .  <- nodes one step away
  . /   / \ .
  (0) (6) (5)   <- nodes two steps away
      . .   .
     .   .   .
```

A common belief in the early days of Computer Chess was that the deeper a tree search algorithm was able to explore, the better the solver was. It turned out that, in some cases, a 5-steps look-ahead solver performed better than a 6-steps look-ahead solver. In those cases the 5-steps look-ahead solver was making better strategic decisions than it's 6-steps look-ahead counterpart [2]. In Chess as well as in other games such as Go and Freecell, some moves require a very deep search to determine their outcome. Stopping the search too early leaves the outcome imprecise.

Many articles describe this phenomenon and offer solutions: B* tree search [4], conspirary numbers[29], singular extension [2] or null-moves[3]. All of them aim to implement a sense of tactical solidity (also called *quiet* positions) into the tree search algorithm. When a move leads to a state that is considered not stable enough, not quiet enough, the tree search depth is dynamically extended to better determine the outcome of that move. Instead of *iteratively* deepening the tree search and explore *all* states at depth n+1, the tree search is *selectively* deepened to only explore *some* disputed states. Using these techniques, solvers are capable of analysing deep lines of play and to determine what moves are worth it.

In Freecell, like in Chess, there are positions qualified as unstable. When a player wants to move multiple cards at once from one cascade to another, he uses the freecells to temporary store the cards at the bottom of the origin cascade, moves the made-accessible card from the origin cascade to the destination cascade and moves the cards back from the freecells into the destination cascade. This operation is called a supermove and is showcased in the introduction in Figure 0.2. To temporarily store cards in the freecells only is basic and offers limited card mobility, some moves requiring more than 4 freecells to be performed. It is possible to store cards on other cascades (given the move is authorized) or to create a temporary stack in an empty column, greatly extending card mobility. While human players are used to perform these tricks and are able to quickly determine whether the supermove is possible, a solver needs to perform a deep search to validate it. If the search is stopped too early, there is a risk that the freecells would still be occupied by the cards of the supermove. Thus an evaluation function would give a poor score to the game state as the mobility decreased which would ultimately wrongly discard the supermove.

Two research question can be asked in regard to informed tree search:

> **RQ2**: “Can an informed tree search be conducted to solve Freecell ?”
>
> **RQ3**: “What properties can describe how good or bad is a Freecell board ?”

1.5 Two-player searches

When playing a non-cooperative two-player game, the opponent tries to defeat us. This is the definition of a zero-sum game: both players search to win, which is the same as to search the opponent loss. When simulating lines of play like tree-search related algorithms do, it is important to correctly simulate the opponent’s willingness to win. Those algorithms are outside of our scope as we study single-player game solvers but they are very important in the science of solvers. Many of the algorithms cited above are used primary alongside Minimax which is an informed search algorithm dedicated to two-player tree searches.

And/Or search

And/Or search is one of the most basic algorithm that is dedicated to two-player tree search. It formulates the following postulate:

> Out of the available moves, each player always selects (if it exists) a move > that ultimately leads to his victory.

We identify two players, “me”: the player who uses And/Or to select moves, and “him”: the player “I” am trying to defeat.

The algorithm first explores the entire game graph to find all leaves that is all states where the game is over and where one of the two players won. With the outcome of all moves, it uses the mentioned postulate to backtrack the information one step before the end.

If the step before the end is “our” turn, “we” could select any move that leads to “our” victory and discard the others. This is known as a OR (any) node. The state is marked as victorious if it exists at least one move that leads to a state known as victorious. Otherwise, if all moves lead to states known as defeated, then this state is marked as defeated.

If the step before the end is “his” turn, “he” would select a move that leads to “his” victory, “we” have to ensure that there is no such move, “we” have to ensure that all moves lead to “our” victory. This is known as a AND (all) node. This state is marked as victorious (to “us”) if all moves lead to states known as victorious (to “us”). Otherwise, if there exists a move that leads to a state known as defeated (victorious to “him”) then this state is marked as defeated.

When all states one step before the end have been marked as victorious or defeated, it is possible to backtrack one step before (so 2 steps before the end). At the end, when the information have been backtracked to all steps, the outcome of the game is fully determined.

While this algorithm is fully deterministic and applicable to any turn-based information complete two-player game, it is impossible to use in practice due to its requirements. It requires to simulate all moves in order to generate the entire game graph. Then, using the ultimate outcome of all moves, to backtrack the information to all previous steps. Basically the algorithm traverse the entire game graph twice, once to discover it, once to backtrack the information. Just like other naive tree search algorithms, it is impossible to apply to most games.

Minimax

Minimax is the informed equivalent of And/Or. It uses heuristics to determine the probability of one player winning at every step. The postulate is reformulated as:

> Out of the available moves, each player always selects a move that maximizes its chances of winning.

Because it uses heuristics, it is possible to only explore a part of the game graph, to only explore the states n-moves away, and to use the evaluation function to evaluate each explored state. Note that selective deepening considerations do stand, see the “Multi-step exploration” chapter of this work.

Because it uses heuristics, except when the game is over, the evaluation function is unable of determining a winner. It only gives a score that indicates our chances of winning the game. When it is our turn, we can select the move that leads to the Maximum score (=our best chance to win the game), this is a Max node. When it is the opponent’s turn, he most likely will select a move that optimizes his own chances of winning the game, he will select a move that leads to the Minimum score (=his best chance to win the game), this is a Min node.

Alpha-beta pruning

Alpha-beta is a way to prune a graph that is traversed by Minimax. It is possible that some branches cannot alter the final score of the parent node, and thus can be discarded.

In the example below, there are 7 nodes numbered from `a` to `g`. The four leaves `d`, `e`, `f` and `g` have received a score thanks to an evaluation function. We are applying Minimax to compute the score of node `a`. Alpha-beta tells us the score of node `g` cannot impact the score of node `a` so we can safely discard that branch.

```
         (a:3)           MIN step
        /     \
     (b:3)   (c:6)       MAX step
    /   \     /   \
(d:1) (e:3) (f:6) (g:*)  Evaluated score
```

Remember that Minimax traverses the graph post-order depth-first. The nodes are traversed: `d -> e -> b -> f -> g -> c -> a`. To compute the score of `a`, a naive Minimax implementation would compute the score of all its descendants. The reality is, we can already assign a *running, temporary* score to `a` as soon as `d` has been evaluated.

After `d` has been evaluated, the *current, temporary* score of `b` (that is a MAX node) is equal to the minimum score of its immediate children that have already been evaluated. That score is `1`, the score of `d`. Going one step above, the *running, temporary* score of `a` (that is a MIN node) is equals to `1`, the score of `b`. When the next node `e` is evaluated, the score of `b` and `a` are updated to `3`. When the next node `f` is evaluated, the *running, temporary* score of `c` is set as the value of `f`, `6`.

Because `c` is a MAX node, only values higher than 6 would update its score. Because `a` is a MIN node, only values lower than 3 would update its score. If `g` has a value lower than `6`, then it would be discarded because it does not update the score of `c`. If it has a value higher than 3, then it is discarded because it does not update the score of `a`. Whatever the score of `g` is, it cannot change the score of `a`. The branch (including all its descendants) can safely be pruned.

Another common approach to two-player tree search than Minimax with Alpha-beta pruning is Proof-Number search (PNS). It operates similarly but search to *proof/disproof* branches in order to prune them. Allis [1] published the definition of PNS and Kishimito [24] published a survey where the implementation in game solvers is discussed.

1.6 Goal Generation

Quoting Bouzy and Cazenave [7]:

> The aim of a game playing program is neither to search within trees nor to evaluate boards - the two activities are only means. The aim is to generate moves, and to select one.

In tree-search related techniques, knowledge is gained by simulating moves and evaluating the resulting game states. The whole knowledge is captured in a single value computed by an evaluation function. The solver simply selects moves that lead to the game state that was evaluated as better than the others. This technique completely fails when the game knowledge is too complex to be captured by a single evaluation function.

In Chess, it is possible to capture the many properties like the King's safety or the overall mobility of each piece into a single value. Each piece can radiate an influence anywhere in the board given a few moves. In the game of Go, those properties no longer stand. Each stone only radiates an influence to its immediate surroundings throughout the entire game. Concepts like territory or objects like strings, eyes and blocks are dependent on their surroundings too. Locality is a very important concept in Go. It starts as one stone then, as more stones are added, they become a block, blocks can be linked to each other to form an area. Stones in a same area can radiate an influence over a large territory, yet it can be totally independent from another area.

A single global evaluation function fails to understand all those concepts [11] and, more importantly, it fails to help decide which move to perform next.

Goal generation uses a different approach to select which move to perform next. Instead of gaining knowledge by simulating moves, it gains knowledge by studying the current board in depth. In Go, it starts by detecting the many objects: strings, eyes, blocks, etc. Then, it computes the influence and territory of each player. Using the accumulated knowledge, it runs many highly sophisticated analyses according to various predefined strategies. Such strategies are based on the "human" way of playing the game. There are some macro strategies like occupying a big empty point, attacking or defending a territory. There are some micro strategies like forming strings between groups, cutting an opponent string, making or destroying an eye, and so on.

Once this extended analysis has been performed, it becomes necessary to select which strategy to apply for the next move. Each strategy is analysed to determine which one should be executed first. By construction, all strategies do improve the player's situation; they are not evaluated to determine *how good* it is to apply them, they are evaluated to determine *how bad* it would be *not to* apply them *now*. A notion of priority is granted to each strategy and is used to determine the next move.

Many Go playing softwares used this approach in the nineties with some good results. We can cite Goliath and Many Faces of Go, two solvers that ranked in the top 3 of best go playing softwares around 2000. While they were the best in computer against computer matches, none of them could compete with professional human players.

One of the struggle of Goal Generation oriented solver is that the strategies are mostly static. Once a human understands what the strategies are, he can trick the machine into taking bad decisions and ultimately reverse the actions to its advantage. In single-player puzzles, it is possible to craft game configurations where the many strategies fail to play optimally too.

Another problem that is also due the handcrafted nature of the strategies is that it is possible to craft several conflicting strategies. This problem is documented in the literature [7]: the common way to address a situation where a solver plays sub-optimal moves is to add a new strategy or a new set of strategies that play the optimal moves in this situation. While the solvers does now play in this particular situation efficiently, this strategy is also computed and analysed in all other situations and can result in weak plays elsewhere.

In regards to Goal Generation, we ask our fourth question:

> **RQ4**: "Is it possible to mimic humans playing Freecell to solve the game ?"

1.7 Learning

In the previous section about informed tree searches, we noted the importance of a good evaluation function. The better the evaluation function, the faster a tree search finds its way to the goal. In the previous section about goal generation, we noted that it is sometimes too difficult to capture enough game knowledge into a single heuristic. Like in Chess, an evaluation function often measures multiple properties and tries to fit them all into a single score. Many of the tree-search based Freecell solvers use a similar approach to measure multiple properties and to fit them all in a single score. A problem highlighted in Samadi *et al's* [31] work is that combining multiple measures into a single score is cumbersome. It requires to solve two important problems: (i) how to combine heuristics by arithmetic means and (ii) in what game configuration to apply each heuristic.

A solution to face this problem of combining multiple heuristics is to turn to *learning*. To let the machine simulate many games in order to improve an existing heuristic or to determine its own.

Genetic Programing

In biology, organisms are fantastic reproduction machines. We distinguish two different reproduction mechanisms: mitosis and meiosis. In short, most cells replicate via mitosis: they use energy to duplicate themselves, they replicate their DNA and split into separated cells. Some cells, in sexually-reproducing organisms (like mammals), are capable of meiosis: two independent organisms each produce a cell that only has half of the genetic material, those two cells are capable of merging together into a new cell that has the two halves of genetic material combined, producing a new third organism.

When the cells duplicate via mitosis, it is possible that a mutation occurs in the duplicated DNA. During the duplication, it is possible that an amino acid gets wrongly inserted / modified / removed which leads to a modification of the cell or a modification of the proteins that the cell is capable of producing.

Such mutations happen all the time in all organisms. By mitosis and meiosis the mutation gets replicated and propagated to new organisms. Over generations, new organisms are different enough from their ancestors to form a new specie. Thanks to natural selection, when a mutation gives a competitive advantage to one cell, it survives better in its environment than other cells and replicates more. In this context of the Covid-19 pandemic, this is the reason why variants of the original virus exist and replace over time the original virus. When a variant is better adapted to contaminate a human cell, it infects more cells and thus replicates more than another variant and becomes dominant in its human host. As this variant spreads to new humans, it becomes the dominant variant.

Genetic programming is an approach that simulates evolution. We explained above that there is not a definitive way to combine multiple heuristics together, nor there is a definitive way to correctly weight a heuristic, nor there is a definitive way to select when to apply each heuristic in each game situation. A solution is to try some values, observe if they are capable of solving the puzzle then try different values and observe how good the new values perform compared to the first ones.

Genetic programming is a way to automate this process of trial -> observation -> mutation. It first starts as a normal evaluation function: multiple heuristics about the game are measured and combined into a single score. This evaluation function is then used along with an informed tree search algorithm which play against thousands of puzzles. Various statistics about each solution are saved: time and memory usage, number of nodes expanded, length of the obtained solution and more. The genetic algorithm then replicates the evaluation function many times with various mutations, each new evaluation function is then used against the same thousands of puzzles, still gathering statistics about each solution. Once this new generation is completely studied, the statistics of each mutation can be compared. The mutated evaluation functions that outperform their siblings are selected into the next generation; they “survive”. The other evaluation functions are not selected; they “die”. The process is restarted with the new selected generations, each is slightly mutated, each mutation is used to solve the same puzzles, etc.

The current best Freecell solver is GA-Freecell [14, 15]. It uses the same tree search function as Bjarnson’s [5] solver, the previously best freecell solver. The difference between GA-Freecell and Bjarnson’s solver is the heuristics used by the evaluation function. In Bjarnson’s the heuristics are static, they were determined thanks to domain experts. In GA-Freecell, the heuristics have been determined thanks to a genetic algorithm.

Reinforcement Learning

Quoting Sutton and Barto [38]:

> Reinforcement learning [. . .] is a computational approach to learning whereby an agent tries to maximize a total amount of reward while interacting with a > complex, uncertain environment.

In reinforcement learning (RL), an *agent* (e.g. a player) is placed in an *environment* (e.g. a Freecell game). The agent seeks to select *actions* (e.g. to play moves) in a way to maximize a *reward* throughout an *episode* (e.g. a game). Each time the agent selects an action, the environment changes, the agent receives a reward and he is placed in a new *state* where new actions are possible.

The reward is a stimulus, positive or negative, that the agent seeks to maximize in the long term. In the case of Freecell, the goal of the game is to sort the 52 cards in the foundation as fast as possible and use as few moves as possible. The reward could be -1 any time the agent moves any card and use an arbitrary large negative reward when he looses. To maximize its reward (minimize its loss), the agent would search for short solutions.

Similar to an evaluation function in a informed tree-search technique, a *value* function is a function that determines *how good* it is to end up in a particular state. The difference with an evaluation function is that the value function gives a score according to the sum of predicted future rewards, also called the *return*. To be precise, the return can only be determined if the agent is deterministic in its way to choose an action and if the agent uses the same *policy* through an episode. Mathematically speaking, a policy is just a mapping between states and actions on the one hand and the probability to select this action in this state on the other hand.

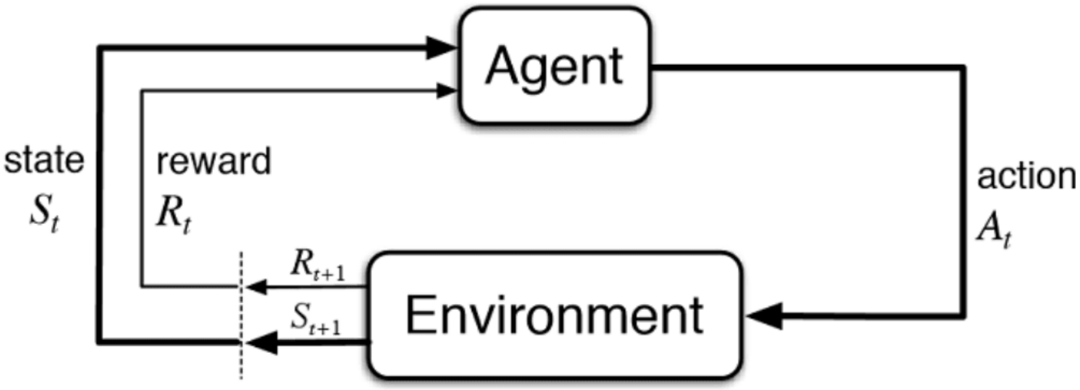

Figure 1.7: The agent-environment interaction

Bellman Optimality Equation

To solve a reinforcement learning problem, one must find a policy (a way to choose an action in each state) that has the best expected return in all states. This policy is called the optimal policy and can be calculated using the *Bellman optimality equation*:

$$v_*(s) = \max_{a \in A(s)} q_{\pi_*}(s,a) = \max_{a \in A(s)} \sum_{s',r} p(s',r|s,a)[r + v_*(s')]$$

where

- $v_*(s)$ is the value of state s under optimal policy
- $q_{\pi_*}(s,a)$ is the value of taking action a in state s under optimal policy
- $p(s',r|s,a)$ is the probability of an action a in state s to end in state s' with reward r
- $r + v_*(s')$ is the recursive return of s under optimal policy
- $A(s)$ is the set of all possible actions in state s

Using this equation, we can write down a system and solve it. The hic is that the system has as many equations and unknown as there are different states. For many practical problems including Freecell there are too many states and the system is too much complex to be solved. A solution is to use another approach to approximate the optimal policy, a policy that is good enough to play the game near optimaly.

Policy iteration

One of those different approaches at solving a RL problem is via experience. The agent is placed in the environment and simulates a full episode according to a policy π. When the episode is completed, the return is determined and can be back propagated to all state-action pairs throughout this episode. The operation must be repeated to many episodes in order to gain knowledge and to improve the approximation of q_π. This stage is known as *the evaluation of* q_π *for* π.

When the evaluation of q_pi gets precise enough, it is possible to determine a new π' policy that always selects the best action according to q_π. This stage is known as *the improvement of* π.

It is possible to keep improving a policy until it reaches a local maximum, a local best policy, by continuously evaluating then improving the current policy.

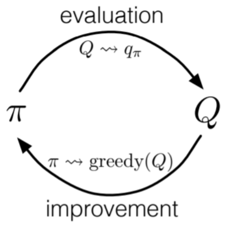

Figure 1.8: Monte Carlo control process

Future work

Because the state-action domain of Freecell is gigantic (about $52! \times 12^{120}$)
we are not confident that Reinforcement Learning is an approach that would give interesting results. The various *Dynamic Programming* (chapter 2), *Monte Carlo* (chapter 3) and *Temporal Difference* (chapter 4) approaches introduced in Sutton and Barto's book all require to explore a large proportion of possible states-actions every time a policy is evaluated and improved.

From the various algorithms introduced in the first chapters, SARSA (described in page 129 of Sutton and Barto's book) is the one we consider the most promising. This algorithm uses the Temporal Difference approach that is out of the three the most efficient ones. This algorithm stands out because it allows to inject knowledge from external sources. Using the solutions stored in a giant databases like FreeCell Solutions [6] would be possible to bootstrap the learning process. Other approaches like those used in Alpha Go [37] and Alpha Go Zero[36] (two RL solvers capable of playing very hard two-player games such as Go, Shogi and Chess at a master level) were not studied as part of this work and might give clues about how to use RL in very large state-action domains.

Chapter 2

Freecell

As mentioned in the introduction, Freecell is a solitaire card game with perfect information. It is hard yet fair, the game has an important branching factor and quite long solutions but a solution exists for almost all deals.

This chapter recalls the rules, defines the vocabulary used throughout this work and explains multiple Freecell properties.

2.1 Rules and goal

Remember that the board is separated into three specifics regions : the freecells, the foundation and the cascades. The game starts with a random deal of a standard 52-cards deck over the 8 cascades, all the slots in the freecells and in the foundation start empty. A standard 52-cards deck has 4 suits : **S**pade (black), **H**eart (red), **C**lub (black) and **D**iamond (red); each suit has 13 cards: **A**ce (1), 2, 3, 4, 5, 6, 7, 8, 9, 10, **J**ack (11), **Q**ueen (12) and **K**ing (13).

The player can move only one card from the cascades or from the freecells. He can move cards to the cascades, the freecells or the foundation under some conditions. Once a card has been moved to the foundation, it cannot be moved back.

It is possible to store any card in a freecell as long as it is empty. It is possible to move a card to a cascade if the cascade is empty or if the bottom-most card is of opposite colour and of rank + 1 (e.g. `9H` to `10S`). It is only possible to move a card to the foundation if the card is an ace and the slot is empty or if the top-most card is of the same suit and of rank - 1 (e.g. `9H` to `8H`).

To move a card that is not the bottom-most one from the cascade, all cards below must be moved first. Looking at the board configuration in the figure aside, only cards `{JS, 10S, 9H, KS}` can be moved, cards `{KH, QS, JH, 10H, QH}` are blocked until all the cards bellow are moved elsewhere.

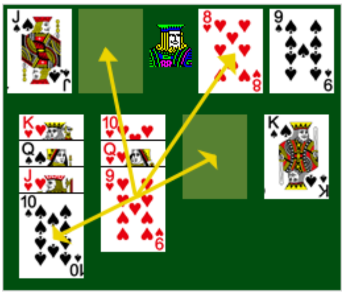

Figure 2.1: Move 9 of heart

The goal is to move all the cards to the foundation. When it is impossible to move any card, the game is over; if all cards are in the foundation then the player wins, else he looses.

2.2 Solitaire order and sequence

All cards are placed in sequence in each cascade. The top-most card is the first card of the cascade, the card just below the top-most one is the second card, . . . , the bottom-most card is the last card of the cascade.

If two cards are following each other in the same cascade, are of opposite colours and the below one has the rank before the card above, then these two cards are said to respect the *solitaire order*. At the beginning of the game, cards are dealt randomly with no regard to this order. When a card is moved as part of the game to a cascade, it is mandatory that it follows this order.

When multiple cards are in sequence and respect the solitaire order, the sequence is qualified as a *solitaire sequence*.

In Figure 2.1, the cards `{KH, QS, JH, 10S}` are all in the same cascade and respect the solitaire order, they are in a solitaire sequence. Note that in this sequence, `{KH, QS, JH}`, `{QS, JH, 10S}`, `{KH, QS}`, `{QS, JH}` and `{JH, 10S}` are solitaire sequences too.

Definition 2.2.1 (Solitaire order)**.** Two cards of opposite colour where the card below has the previous rank of the card above.

Definition 2.2.2 (Solitaire sequence)**.** A sequence (cascade wise) of cards that respects the solitaire order.

2.3 Move notation

Several games use a domain specific language (DSL) to transcript a game to text. A common example is the Chess notation to describe each move that occurred during a game. In freecell, each move is described using two letters: the first letter is the card origin, the second the card destination. Each cascade is labelled from 1 to 8, each freecell is labelled from a to d, the foundation is described by the letter h. Moving a card from the third cascade to the second freecell is typed 3b. Moving a card from the first freecell to its home foundation is typed ah. Figure 0.2 shows an example of two move sequences that achieve moving 8D and 9S (7th column) to 10H (8th column). The two sequences are `[7d, 78, d8]` and `[73, 78, 38]`. The first sequence uses a freecell to temporarily hold 8D in a freecell, the second sequence uses an empty column. Temporarily stacking cards on freecells and empty columns to move a sequence of sorted cards from one cascade to another is a supermove. Transcripting every move including those used to supermove is the detailed notation, transcripting moves except those used to supermove is the standard notation. In detailed notation, moving 8D and 9S from the 7th column onto 10H of the 8th column is `[7d, 78, d8]`, in standard notation it is shortened to `[78]` which means "move a maximum of cards at once, from the 7th column onto the 8th column".

2.4 Supermove

By the rules of the game, any move in the cascade must respect the solitaire order. That is, if a card is at the bottom of a cascade, only a card of opposite colour and of rank - 1 can be moved below it, such a card is a card of interest in regard to this cascade. When such a card of interest exists in the cascades and is the bottom-most card of its own cascade already, it is possible to move that card right away. When such a card of interest exists in the cascades and is the top-most card of a solitaire sequence that is at the bottom-most of its own cascade and if there are enough empty freecells and empty columns, it is possible to supermove that sequence.

In Figure 2.2, the bottom-most card of the 4th cascade is a black king, only a red queen can be moved under this king. The first card of the 2nd column is a red queen but the rules forbid to move the card right away because it is not the bottom-most card of its column. The queen is part of a solitaire sequence that includes the bottom-most card of the cascade, the sequence is candidate for a supermove.

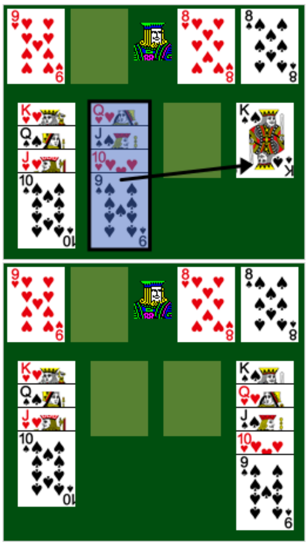

Figure 2.2: Supermove 24

To be performed, a supermove requires to temporary store cards in *empty slots* (either empty freecells, empty cascades or both). When there are no empty slots, it is impossible to temporary store cards, therefore it is impossible to supermove. Two kinds of supermoves are distinguished: the *simple* supermove and the *deep* supermove.

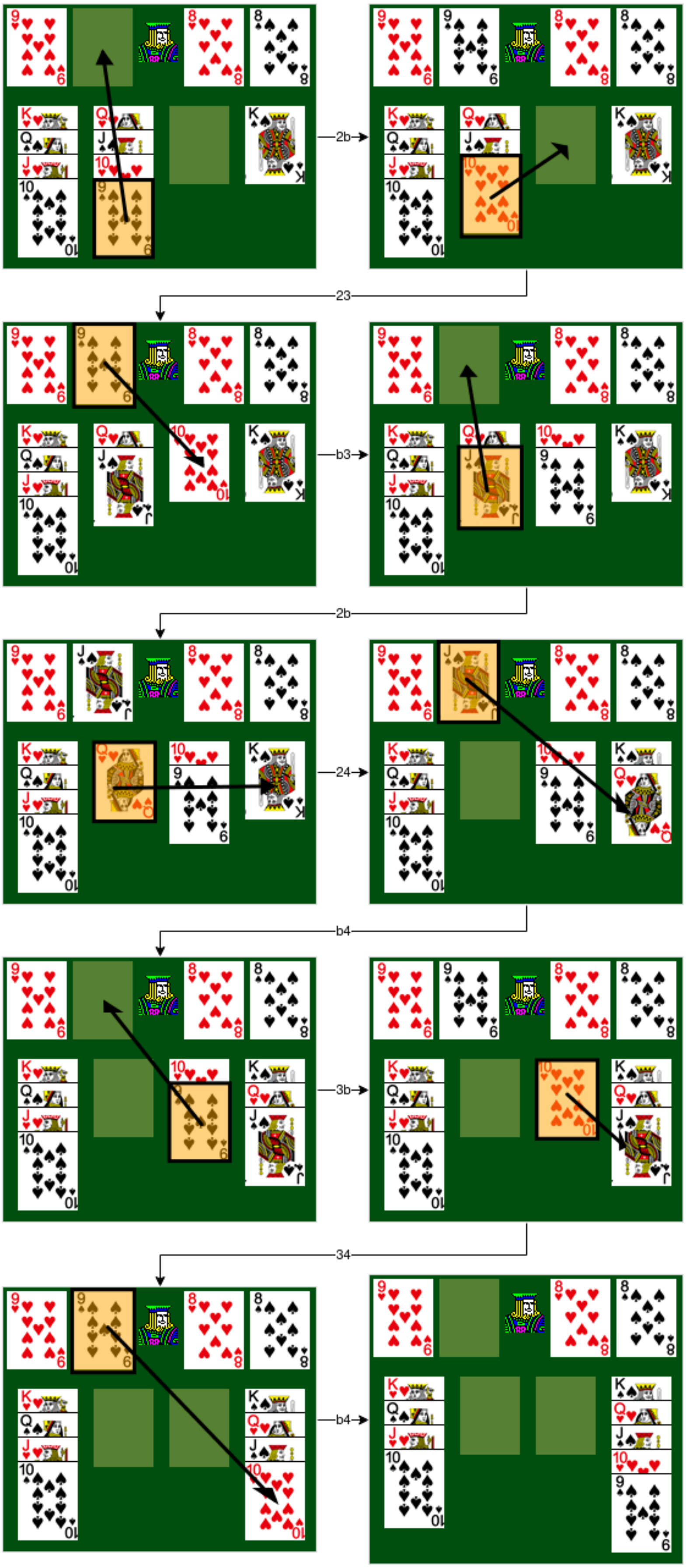

Figure 2.3: Supermove 24 (detailed)

Simple supermove

The simple supermove stores a single card per empty slot until the card of interest is accessible. Once the card of interest is accessible, it can be moved to the destination cascade. Because they initially were in a solitaire sequence, all the cards that have been moved as part of the supermove to empty slots can be moved to the destination cascade too; the cards are to be moved in the reverse order that they were moved to empty slots. The simple supermove offers a *card mobility* (m) equals to 1 + the count of empty freecells (f) and empty cascades (c).

$$m = 1 + f + c$$

Deep supermove

The deep supermove takes advantage of an empty cascades to temporarily store more cards: it supermoves as many card as possible to an empty cascade and repeats until the card of interest can be simple-supermoved to the destination cascade. Once the card of interest is accessible, it can be simple-supermoved to the destination cascade. Because they initially were in a solitaire sequence and that they were simple-supermoved using as many available free slots as possible, all the cards temporarily stored in empty cascades can be simple-supermoved to the destination cascade; they are to be simple-supermoved in the reverse order that they were simple-supermoved to the empty cascades. The deep supermove offers an extended card mobility.

$$m = 1 + \sum_{i=f}^{f+c} i$$

In Figure 2.2, we want to supermove `{QH, JS, 10H, 9S}` from the second cascade to `{KS}` on the fourth cascade. We need a card mobility of 4 to achieve this supermove. There is one empty cascade and one empty column. Using a simple supermove, because we store a single card per empty slot, only achieves a card mobility equals to 3 which is not enough. Using a deep supermove, because it uses the empty cascade to stack multiple cards, achieves a card mobility of 4 which is enough. This deep supermove is decomposed into 3 simple supermoves: (i) `{10H, 9S}` are simple-supermoved to the empty cascade via the following sequence: `{2b, 23, b3}`, (ii) `{QH, JS}` are simple-supermoved to the fourth cascade via `{2b, 24, b4}` and (iii) `{10H, 9S}` are simple-supermoved again, this time to the fourth cascade via `{3b, 34, b4}`. Every single move is showcased in Figure 2.3.

Definition 2.4.1 (Supermove). *verb* The action to move as many cards from one cascade to another given that it is possible to decompose the move in a succession of valid single moves.

Definition 2.4.2 (Supermove). *noun* A succession of single moves that, played together, move a solitaire sequence found at the bottom of a cascade to another cascade.

2.5 Automove

The goal is to move all cards to the foundation. By the rules of the game, a card can be moved to the foundation only if it continues the sequence of its suit in ascending rank order. Another rule is that, when a card has been moved to the foundation, it cannot be moved back to the cascades or the freecells. From a certain standpoint, when a card is moved to the foundation, it is like removed from the game. When a card is moved to the foundation, it can no longer be used in the cascades, it is impossible to use the card in a solitaire sequence to move a card of opposite colour and of rank - 1 under it.

Moving cards to the foundation too early may cause the puzzle to get unsolvable. It is important to determine when it is safe to move a card to the foundation and when it is not. This problem is showcased in the example next page.

Figure 2.4 showcases a board where almost all spade cards have been moved to the foundation but where more than half heart cards are still in the cascades. The '7H' is the next heart card to be moved to the foundation, but the card is the first card of the first cascade. The 6 other cards in the first cascade must be moved elsewhere to free '7H'.

This situation is quite complex because the first cascade is not a solitaire sequence: it is impossible to supermove many cards elsewhere. All the 6 cards have to be moved one by one but there are only 4 empty slots.

To solve this complex situation, there are two possible strategies: (i) move the two spade cards to the foundation to empty the second cascade, (ii) use the king and queen of spade to build solitaire sequences in the cascades.

Figure 2.5 showcases the first strategy. First, the two spade cards are moved to the foundation: '2h, 2h'. The second cascade is now empty, there are 5 empty slots. The second action is to move the cards from the first cascade to the empty slots: '12, 13, 14, 1a, 1b'. Game over, it is impossible to move any card and all the cards are not in the foundation.

Figure 2.6 showcases the second strategy. The two spade cards must be used wisely. 'QS' is a black card of rank 12, it can be moved under a red king (rank 13) and a red jack (rank 11) can be moved under the queen. 'KS' is a black card of rank 13, it can only be moved to an empty cascade and a red queen (rank 12) can be moved under it.

First, the red king is moved to an empty cascade and the black queen is moved under it: '13, 23'. Second, the black king is already the first card of its cascade so there is no need to move it around, the red queen is moved under it: '12'. Third, the red jack is moved under the black queen: '13'. Finally the 8, 9 and 10 red are moved to empty slots: '1a, 1b, 14'. '7H' has been freed, the game continues.

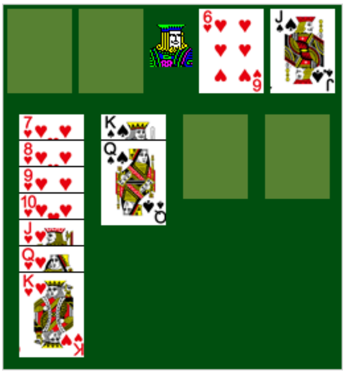

Figure 2.4: 7H stuck

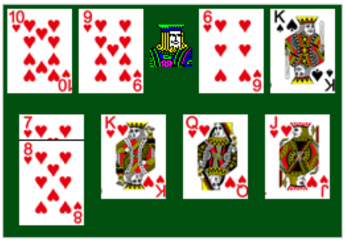

Figure 2.5: First strategy

Figure 2.6: Second strategy

When a card can still be used in a cascade to move another card under it, it is better *not to* move that card to the foundation yet. The opposite stands too, when all the cards that could be moved under a card in the cascades are in the foundation already, this card can safely be moved to the foundation.

This property is very important to understand. It is safe to move a card to the foundation when that card is no longer useful in the board. The card is non longer useful in the board when all the cards that could be moved under it (the cards of opposite colour and of previous rank) are in the foundation already.

Because those cards (low card) will be moved to the foundation once they are accessible, they will not be moved below a card of opposite colour and of next rank (high card). Even if the low card *could* be moved under the high card, such a move is useless as it is better to move the low card to the foundation. Because no low card will be moved under the high card, it is safe to move the high card to the foundation too.

Definition 2.5.1 (Automove)**.** *verb* The action to move a card that is no longer useful in the cascades to the foundation.

Definition 2.5.2 (Automove)**.** *noun* A valid move to the foundation where the moved card was non longer useful in the cascades.

Theorem 2.5.1 (Freecell Automove)**.** A card located in a freecell or at the bottom of a cascade can safely be moved to the foundation if the two cards of opposite colour and of previous rank are in the foundation already.

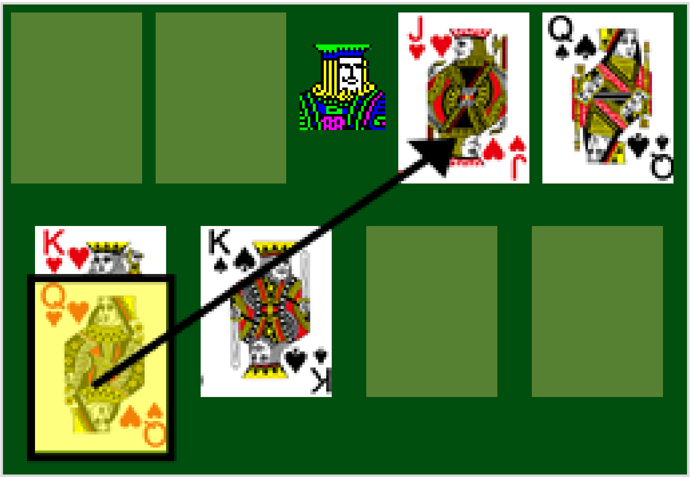

Figure 2.7: No card can be moved under the red queen

Corollary. A card located in a freecell or at the bottom of a cascade can safely be moved to the foundation if the two cards of opposite colour and of previous rank will be automoved to the foundation instead.

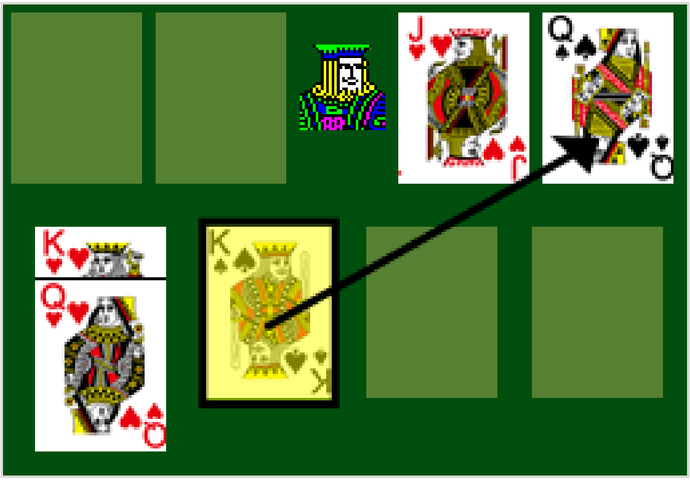

Figure 2.8: The card that could go below the black king will be automoved instead

2.6 Subgoal: sort all cascades

The rules establish an order in which the cards of a same suit are to be moved to the foundation: lower ranks first. The automove theorem establishes a relation between colours in which the cards can safely be moved to the foundation: it is safe to move a card to the foundation when both the cards of opposite colour and of rank - 2 are in the foundation already.

Taken together, the cards of higher ranks, regardless of their suit, are moved to the foundation later than the cards of lower ranks. This means that when a cascade is fully sorted in descending rank order from top to bottom: those cards are also fully sorted in ascending automove order from bottom to top.

When all the cascades are sorted in descending rank order from top to bottom, it is possible to automove all cards. When all the cascades are sorted, the game is won.

Figures 2.9 and 2.10 illustrate this subgoal. On the left (2.9) is a random deal and (2.10) is another deal near to be solved. On the right, we find the same two boards but the cards have been coloured on a grey scale according to their rank. In Figure 2.9, the cards are just spread around with no particular order: there are still many moves to do to find a solution. In Figure 2.10, all the cards of lower ranks in the cascades are placed under the cards of higher ranks: there are just a few moves to do win the game.

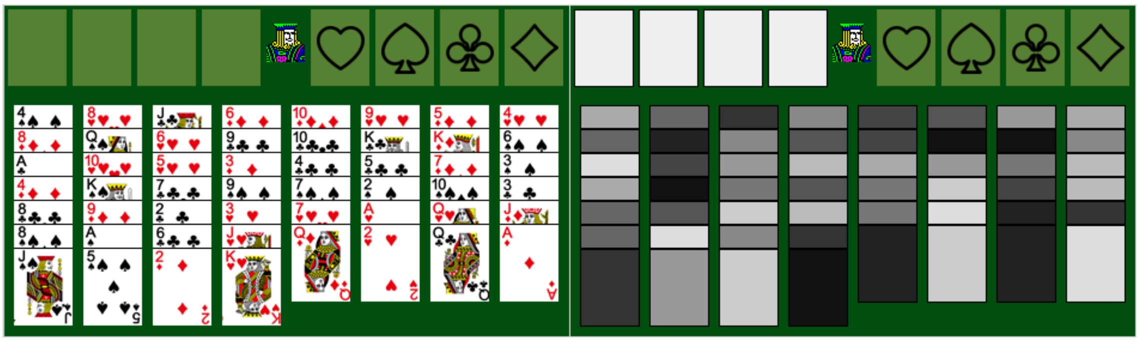

Figure 2.9: Start of a game, no cascade is sorted

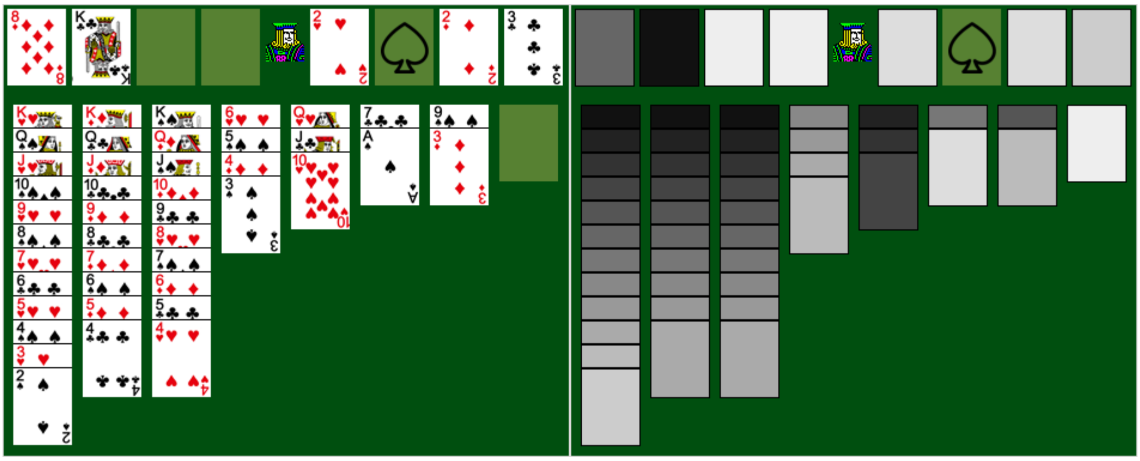

Figure 2.10: End of a game, all the cascades are sorted

Chapter 3

Crafting a solitaire solver

Quoting RQ1:

> How can the Freecell solitaire game be solved using artificial intelligence?

From the literature, Tree Search clearly stands out as the technique of choice when crafting a game solver. The technique has been implemented countless times for many games and puzzles. Checker, Chess, Rubik's Cube, Mystic Square and Maze are all solved using tree-search related techniques. Therefore, Freecell should be solvable using tree-search related techniques too.

Another technique that we consider worth trying is Goal Generation, a technique used in the nineties to solve Go with some degree of success. From our research, this technique has not been used to solve any single-player game.

Those two techniques will be discussed in turn in this chapter.

3.1 Tree Search Solver

Quoting RQ2:

> Can an informed tree search be conducted to solve Freecell?

In section 1.4 "Informed Tree Searches", we learned that tree-search was the dominant technique to solve games. Using a fine-tuned evaluation function, a multi-step exploration that achieves tactical solidity and by simulating many moves, we expect that Freecell can be solved with this method.

Evaluation functions

Quoting RQ3:

> What properties can describe how good or bad is a Freecell board?

We studied multiple properties of a Freecell board in the previous chapter. The supermove and the automove are among them. The supermove describes a way to move multiple cards while the automove defines when it is safe to move a card to the foundation. Using the automove, a sub goal was identified and described, namely, ordering all cards of the board in descending order.

To be efficient, an evaluation function should evaluate each board in regard to those properties. It should favour boards where (i) there are many cards in the foundation (main goal) while (ii) respecting the automove theorem and its corollary, where (iii) the cascades are pretty well sorted (sub goal), and where (iv) there are many freecells and empty columns (supermove). It would also be interesting that the evaluation function succeeds to guide the tree search into finding (v) short solutions.

Number of Cards in Foundation

Evaluation function for the main goal.

Trivial to compute: count how many cards are in the foundation.

Automove Violation

Evaluation function for the automove theorem.

When a card is moved to the foundation in violation of the automove theorem or its corollary, the move is considered unsafe because it may prevent building a solitaire sequence in the cascades. See section 2.4 (Automove).

A board where the Automove theorem has been violated should be ranked worse than a board where there are as many cards in the foundation and where the Automove theorem has been respected.

Take the highest card of every suit in the foundation, using only those cards: the malus is the accumulated absolute difference between the cards of opposite colours if that difference exceeds 2.

Cascade Sortedness

Evaluation function for the sub goal.

When all cascades are sorted top to bottom by descending rank value, the game is trivial to solve.

The score depends only on the cards in the cascades: the freecells and the foundation can be ignored. The cascades that are empty can be ignored too. The score is increased (bonus) when a cascade is fully sorted, the score is decreased (malus, negative bonus) when it is not.

Because the best specification is code [19] and that the actual computation is quite complicated, below is an actual C implementation of how to compute the cascade sortedness:

```c
int compute_cascade_sortedness(Board *board) {
    int col, sortedness, depth;
    Card *highcard, *card;

    sortedness = 0;
    for (col = 0; col < 8; col++) {
        if (is_empty(board, col)) {}
        else if (is_fully_sorted(board, col)) {
            // Bonus: highest rank + cascade length
            sortedness += board->cascade[col][1].rank + board->cslen[col];
        } else {
            // Malus: #moves (actual + potential) required to access blocked cards
            depth = 0;
            for (card = bottom_card(board, col) - 1; !is_nullcard(*card); card--) {
                depth++
                if (card->rank < (card+1)->rank) {
                    sortedness -= depth;
                    sortedness -= ((card+1)->rank - card->rank);
                    depth = 0;
                }
            }
        }
    }
    return sortedness;
}
```

Our rationals for this computation are as follows:

- When cascade have been emptied, placing a card with a high rank allows to place up to $rank - 1$ cards below. When a high rank is placed in an empty cascade, the potential cascade length is increased. This is `sortedness += board->cascade[col][1].rank`.
- When a cascade is fully sorted, there are as many cards as there are in this cascade that respect the Automove theorem. This is `sortedness += board->cslen[col]`.
- When a cascade is not fully sorted, there are cards that are blocked. When a card is blocked (it is lower than any next card), then one must move all the cards bellow in order to access it. This is the *actual* required moves to access the blocked card: `sortedness -= depth`. When there is a big difference between two cards, like a king that blocks an ace, there are potentially many cards that can be moved to the cascade, further blocking the low rank card. These are the *potentially* required moves to access a block card: `sortedness -= ((card+1)->rank - card->rank)`.

Card Mobility

Evaluation function for the supermove.

The deep-supermove card mobility equation: $m = 1 + \sum_{i=f}^{f+c} i$, can be factorised to get rid of the summation: $m = 1 + f(c+1) + \frac{c(c+1)}{2}$, where f is the number of empty freecells and c is the number of empty cascades.

Solution Length

Evaluation function to find a short solution.

Trivial to compute: retrieve the depth in the search tree of the current board.

Move simulation and Tactical solidity

In section 1.4 (Multi-step Exploration), we explained that it is important to simulate many moves in order to gain knowledge. We also warned that an evaluation function may give imprecise results when a situation is unstable; we explained it is required to identify the unstable situations and to further explore them to gain knowledge and to increase the precision of the score returned by the evaluation function.

In that section, we explained that, during a supermove, the board was unstable in regard to the *Card Mobility* criterion. Because it uses empty freecells and empty cascades to move cards from one cascade to another, if the supermove is interrupted early, there is a risk that cards are still in the freecells or the cascades and the evaluation function for the *Card Mobility* criterion would return a decreased score.

The solution that we suggest in this regard is to simulate supermoves instead of single moves. In the graph structure that capture game states in nodes and link nodes via edges, each edge would be a supermove. Instead of playing a single move to move from one node to another, a sequence of moves would be played instead; in that way, only stable boards would be evaluated.

Implementation

As said above, the most comprehensive specification is code. In this section, we give the rationals that lie behind the implementation of our first solver. The reader is advised to read the source code which is available at GitHub: https://github.com/julien00859/cfreecell/releases/tag/a-star [21].

The key ideas are the following:

1. It performs a fixed 5-steps look-ahead using single moves to discover new game states. Every discovered state is evaluated and added in a priority queue that is ordered by state score. See the `depth_search()` function.

2. The evaluation function is $score = n - \frac{d}{4}$ where n is the number of cards in the foundation and d is the depth in the search tree of the current state. See the `evaluate()` function.

3. To prevent loops, every time a state is discovered, it is verified that the state is not in the priority queue already. If the state is in the queue already, the move is cancelled. Thanks to this security, while Freecell should be a Directed Cyclic Graph, we can capture it in a Tree instead. See section 1.3 (Naive Tree Searches).
4. When the algorithm is done exploring all 5 states ahead, it gets the best state from the priority queue and performs a new 5-steps discovery starting from this best state. See the `astar_search()` function.
5. Every time the best state is obtained from the priority queue, its score is updated to $-\infty$ and its position in the priority queue is updated accordingly. A score equal to $-\infty$ denotes a state that has been exploited and should not be selected again. When a state with a score equal to $-\infty$ is obtained from the priority queue, it means that all discovered states have been exploited and that the game has no solution.
6. To save memory, every node in the graph only saves the last move (two pointers: card origin `*fromcard` in the board and card destination `*tocard` in the board) instead of the whole board (which is 248 bytes fat).
7. The actual board is a mutable structure where "the absence of card" (like an empty freecell or the space under the bottom-most card of a cascade) is represented by a special "null" card. A move consists of swapping the card we want to move with a "null" card. It is possible to reverse (undo) the move by simply inverting the origin and the destination: the "null" card and the other card are swapped to their original configuration. See the `play()` function.

Discussion

The solver we implemented in regard to the Tree-Search approach suffers from two important limitations: it uses an evaluation function that is too naive and explores the game using single moves instead of supermoves. Due to those two limitations, the solver has to explore many nodes before finding a solution: it uses a lot of memory and searches for a long time.

Below is a table where the results of our solver are aggregated. The solver was run on 10 different deals, it was able to find a solution for 9 of them but ran out of memory after about 40 minutes when it was solving the 10th one.

metric	min	max	average	median
Solution length	80	121	100	98
Calls to `move`	32,320,566	301,262,098	97,251,847	78,680,001
Boards evaluated	3,783,015	42,891,813	12,555,856	9,580,117
Elapsed time (mm:ss.00)	01:17.79	16:26.28	04:16.92	03:08.78
Memory used (kB)	1,000,616	7,512,060	2,885,044	1,939,478

The results highlight the limitations of the implementation. While most Freecell solvers are capable of solving most puzzles near instantly, ours requires several minutes and at least 1Gb of RAM. We did not implement anything related to the supermove, the automove or the subgoal because this solver predates the formal definition of our subgoal and because we had not yet implemented a supermove algorithm.

3.2 Goal Generation Solver

Quoting RQ4:

> Is it possible to mimic humans playing Freecell to solve the game ?

The various Freecell abstractions like the supermove and the automove are not unknown to the players. The subgoal is no secret either. To our knowledge, most players tend to create long solitaire sequences in empty cascades.

From our observation, the main difference between a human and a tree-search algorithm is that human players rarely cancel moves (they rarely use the “undo” button) and rather study the current board and plan the next moves to achieve an objective. The objectives include : (i) empty a cascade, (ii) spread card from a cascade to “liberate” a specific card, (iii) build long solitaire sequences starting from a king (or another high rank card) in empty cascades.

In the matter of those objectives, players speak about cascades they want to *build* and the cascades they want to *destroy*. Every movement *from* a cascade is a destruction, every movement *to* a cascade is a construction. When a cascade is fully sorted, they want to build more on that cascade. When there are cards that block other important cards (like one that they could move to the foundation or one that they could move to a fully sorted cascade) they want to destroy it.

In regard of the automove, some game softwares automatically move any card that respect the automove theorem and its corollary to the foundation; hence the name.

Strategies

The following strategies have been carefully crafted by hand to mimic a human approach to the game. They are to be evaluated in order as the first ones are the preferred (or automatic) strategies played by human players, the later ones are what they do when there is nothing more important to do first.

Automove

When any card is immediately accessible (it is in a freecell or it the bottom-most card of a cascade) and it respects the automove theorem or its corollary, it is moved to the foundation.

Build down

When it is possible to move a card from any freecell to any cascade, the move is played.

When it is possible to supermove cards from a cascade to another cascade that is better sorted, the moves are played. See section 3.1 (Cascade Sortedness).

Build empty

When a cascade is empty, supermove the highest ranking card to the empty cascade. Partial supermoves, i.e. supermoves that would only move a part of the solitaire sequence, are forbidden. Moves from freecell are considered too.

Empty cascade

When a cascade is not fully sorted and that it is possible to spread all cards to other cascades (excluding empty ones) and freecells, those moves are played.

Access low card

Search for the smallest cards still in the cascade and in the freecells. If it is possible to spread the cards that block it to other cascades (including empty ones), first play the moves and then move the card made accessible to the foundation (even if it is not an automove).

Access build card

For cascades that are fully sorted, search for a card that could be placed below. If the card is located in a cascade that is not fully sorted yet (because it would conflict with *Build down*) and if it is possible to spread the cards that are blocking it, play the moves and move the card made accessible to the cascade.

Any move between cascades

If it is possible to supermove any card from any cascade (excluding fully sorted ones) to any other cascade then play the moves.

Any move to freecells

If there are empty freecells, move any card from a cascade that is not fully sorted.

Implementation

As for Tree-Search, the reader is advised to read the source code which is available at GitHub: https://github.com/Julien00859/cfreecell/releases/tag/goal-gen [22].

The key ideas are the following:

1. Every strategy is accompanied by an evaluation function that is dedicated to verify whether it is possible to play the moves dictated by the strategy. When a strategy matches, i.e when its evaluation function detects that the moves are possible, the moves are immediately played.
2. Every time a new board is evaluated, it starts by evaluating the first strategy, then the second, then the third, etc. When no strategy matches, the solver undoes the last moves to restore the previous board and resume the strategy evaluation where it stopped. For example, if the previous strategy matched “Build down, supermove 27” and if we have to backtrack, then the next evaluated strategy is “Build down, supermove 28”, the internal loops of every strategy can be restored to a precise step.
3. Every time the board changes, it verifies if the new board has not been visited yet (it uses an hashset). If it has, the last moves are cancelled and the strategy is backtracked.
4. Two algorithms, `supermove()` and `superaccess()`, have been implemented to move multiple cards at once. The first function implements a deep supermove as described in section 2.3 (Deep supermove). The second function implements a way to *spread* cards from a cascade until a specific card is made accessible or until the cascade has been emptied.
5. It reuses a lot of code that was not Tree-search specific from our last solver. The internal structures such as `Card` and `Board` have been reused, the function `play()` (renamed `move()`) too. There are much more utility functions such as `bottom_card()`, `highest_sorted_card()`, `search_card()` and many others to ease the board manipulation.

Discussion

Thanks to a clever use of the game knowledge, this new solver gives much better results than the previous one. We ran it against 120.000 different random deals, there were 80k deals (65.5%) our solver was capable of solving each under 10 seconds. The statistics about the solved boards are aggregated in the table below.

metric	min	max	average	median
Solution length	164	707,816	375	300
Elapsed time	0	9.97*	0.13	0.01
Memory used (kB)	7,596	80,320	9,672	9,448

It is interesting to note that the solver is able of solving more than half the deals in 0.01 seconds with a somewhat short solution (300 moves) and that it is overall quite cheap in memory usage (under 10kB).

The solutions that need some very long lines of play have been studied to understand why the solver needed thousands of moves to solve some boards that human are capable to solve using about 120 moves. One of the problems is that, in some situations, the two “Build empty” and “Empty cascade” strategies are fighting each other: the first strategy places a card in an empty cascade, the second strategy removes that card from the cascade. This cycle problem is a limitation of our anti-cycle mechanism (hash the board, cancel the move if the hash exists in a set already). There is no order in the freecells, there is no order between different cascades either: two cards in the freecells or two cascades can be swapped without altering the game. Two games that are only different in the order that the cards are placed in the freecells or in the order of the cascades are equivalent. This property is ignored by our hashing function, it should return the same value when two boards are equivalent but it returns a different value whenever two boards are not strictly identical. Because it considers some equivalent boards to be different, whenever there are both empty freecells and empty cascades, the two strategies explore all the possible combinations of cards: the “Build empty” strategy moves every single accessible card sequentially to every empty cascade while the “Empty cascade” strategy moves back the card that has just been moved by “Build empty” to a different freecell. At the end, when all combination have been explored, it tests a new strategy which stops the cycle and resumes the exploration. To solve this problem, the hashing function should return the same value for all boards that are equivalent.

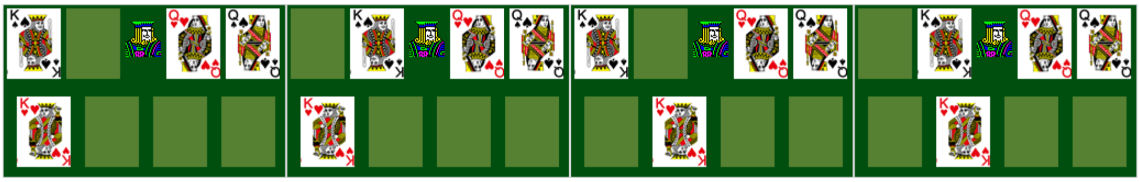

Figure 3.1: Four boards that are equivalent but that our solver wrongly consider different

Conclusion & Future Work

Freecell is a popular solitaire card game that is played by millions of people worldwide. There exist many implementations of the game for many different platforms (PC, Mac OS, Mobiles, etc.). There have been multiple efforts to document solutions in online databases and there exist several autonomous solvers but only one of them is documented in the literature. The game has been studied by a few as a playground for some activities such as Planning, Neural Network or Genetic Programming. Prior to this work, the game itself - its mechanisms and internal structures - had not been formally studied yet.

In the first chapter, we documented several approaches that have been used to solve some one-player and two-player boards since the early age of computers. We approached three primary techniques: (i) tree-search, (ii) goal generation and (iii) learning. Each technique has been successfully used to solve different games such as the Rubik's Cube, Chess and Go. We described each of those techniques, explained how they are used in game solvers, coined some of their downsides and pointed resources for further reading.

In the second chapter, we studied the Freecell solitaire game in depth. The rules have been carefully analysed to formally describe the morphological properties of the game. The terms that are commonly used by human players have been defined too.

In the third chapter, we described how to implement two solvers both in regard to the Tree-Search and the Goal Generation approaches. In the final chapter, we also studied our own C implementation for both solvers, assessed their results and gave the direction for further improvement.

We are quite critical about the implementation of our solver that uses the Tree-Search approach. While our second solver is also far from perfect, the results that have been gathered show that a solver that uses more knowledge is more efficient to solve the game. As future work, we suggest to harden the evaluation function that has been used in our tree-search solver and to repeat the experimentation.

References

[1] Allis, L.V. et al. 1994. Proof-number search. *Artificial Intelligence.* 66, (Mar. 1994), 91–124. DOI:https://doi.org/10.1016/0004-3702(94)90004-3.

[2] Anantharaman, T. et al. 1988. Singular extensions: Adding selectivity to brute-force Searching1. *ICGA Journal.* 11, (Dec. 1988), 135–143. DOI:https://doi.org/10.3233/ICG-1988-11402.

[3] Beal, D. 1990. A generalised quiescence search algorithm. *Artificial Intelligence.* 43, (Apr. 1990), 85–98. DOI:https://doi.org/10.1016/0004-3702(90)90072-8.

[4] Berliner, H. 1979. The b* tree search algorithm: A best-first proof procedure. *Artificial Intelligence.* 12, (May 1979), 23–40. DOI:https://doi.org/10.1016/0004-3702(79)90003-1.

[5] Bjarnason, R. et al. 2007. Searching solitaire in real time. *ICGA Journal.* 30, (Sep. 2007), 131–142. DOI:https://doi.org/10.3233/ICG-2007-30302.

[6] Bortnik, Y. 2008. *FreeCell solutions to 1000000 games.*

[7] Bouzy, B. and Cazenave, T. 2001. Computer go: An AI oriented survey. *Artificial Intelligence.* 132, (Aug. 2001). DOI:https://doi.org/10.1016/S0004-3702(01)00127-8.

[8] Buro, M. 1994. Methods for the evaluation of game positions using examples. (Jun. 1994).

[9] Caulfield, H. and Kinser, J. 1999. Finding the shortest path in the shortest time using PCNN's. *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council.* 10, (Feb. 1999), 604–6. DOI:https://doi.org/10.1109/72.761718.

[10] Cazenave, T. 2020. Monte carlo game solver.

[11] Chen, K.-H. 2000. Some practical techniques for global search in go. *Journal of The International Computer Games Association.* 23, (Jun. 2000). DOI:https://doi.org/10.3233/ICG-2000-23202.

[12] Copeland, J. and Prinz, D. 2017. Computer chess—the first moments.

[13] Dunphy, A. and Heywood, M.I. 2003. "FreeCell" neural network heuristics. *Proceedings of the International Joint Conference on Neural Networks* (Aug. 2003), 2288–2293 vol.3.

[14] Elyasaf, A. et al. 2012. Evolutionary design of FreeCell solvers. *Computational Intelligence and AI in Games, IEEE Transactions on.* 4, (Dec. 2012). DOI:https://doi.org/10.1109/TCIAIG.2012.2210423.

[15] Elyasaf, A. et al. 2011. GA-FreeCell: Evolving solvers for the game of FreeCell. *Genetic and Evolutionary Computation Conference, GECCO'11* (Jan. 2011), 1931–1938.

[16] Enzenberger, M. 1999. The integration of a priori knowledge into a go playing neural network. (Apr. 1999).

[17] Helmstetter, B. and Cazenave, T. 2003. Searching with analysis of dependencies in a solitaire card game. (Jan. 2003), 343–360.

[18] Hsu, F. et al. 1990. A grandmaster chess machine. *Scientific American - SCI AMER.* 263, (Oct. 1990), 44–50. DOI:https://doi.org/10.1038/scientificamerican1090-44.

[19] Issartial, E. et al. 2016. A very comprehensive and precise spec. *CommitStrip.*

[20] Julien00859/15-puzzle: *https://doi.org/10.5281/zenodo.4737854*. Accessed: 2021-05-04.

[21] Julien00859/cfreecell: a*: *https://doi.org/10.5281/zenodo.4737904*. Accessed: 2021-05-04.

[22] Julien00859/cfreecell: Goal generation: *https://doi.org/10.5281/zenodo.4737911*. Accessed: 2021-05-04.

[23] Keller, M. 2021. FreeCell FAQ and link. *Solitaire Laboratory.*

[24] Kishimoto, A. et al. 2012. Game-tree search using proof numbers: The first twenty years. *ICGA journal.* 35, (Sep. 2012), 131–156. DOI:https://doi.org/10.3233/ICG-2012-35302.

[25] Kishimoto, A. and Müller, M. 2017. Game solvers.

[26] Korf, R. 1997. Finding optimal solutions to rubik's cube using pattern databases. *Proceedings of the National Conference on Artificial Intelligence.* (Jun. 1997).

[27] Korf, R. 1985. Korf, r.e.: Depth-first iterative-deepening: An optimal admissible tree search. Artif. Intell. 27, 97-109. *Artificial Intelligence.* 27, (Sep. 1985), 97–109. DOI:https://doi.org/10.1016/0004-3702(85)90084-0.

[28] Krivokuća, M. 2019. Minimax with alpha-beta pruning in python. *Stack Abuse.* Stack Abuse.

[29] Mcallester, D. 1988. Conspiracy numbers for min-max search. *Artif. Intell.* 35, (Jul. 1988), 287–310. DOI:https://doi.org/10.1016/0004-3702(88)90019-7.

[30] Permana, S. et al. 2018. Comparative analysis of pathfinding algorithms a *, dijkstra, and BFS on maze runner game. *IJISTECH (International Journal Of Information System & Technology).* 1, (May 2018), 1. DOI:https://doi.org/10.30645/ijistech.v1i2.7.

[31] Samadi, M. et al. 2008. Learning from multiple heuristics. *Proceedings of the National Conference on Artificial Intelligence* (Jan. 2008), 357–362.

[32] Samuel, A.L. 1959. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development.* 3, (Jan. 1959), 211–229.

[33] Schaeffer, J. et al. 1992. A world championship caliber checkers program. *Artificial Intelligence.* 53, (Feb. 1992), 273–289. DOI:https://doi.org/10.1016/0004-3702(92)90074-8.

[34] Schaeffer, J. 1997. One jump ahead: Challenging human supremacy in checkers. *ICGA Journal.* 20, (Jun. 1997), 93–93. DOI:https://doi.org/10.3233/ICG-1997-20207.

[35] Shannon, C. 1950. Programming a computer for playing chess. (1950).

[36] Silver, D. et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science.* 362, (Dec. 2018), 1140–1144. DOI:https://doi.org/10.1126/science.aar6404.

[37] Silver, D. et al. 2016. Mastering the game of go with deep neural networks and tree search. *Nature.* 529, (Jan. 2016), 484–489. DOI:https://doi.org/10.1038/nature16961.

[38] Sutton, R.S. and Barto, A.G. 2018. *Reinforcement learning: An introduction.* MIT press.

[39] Zarembo, I. and Kodors, S. 2015. Pathfinding algorithm efficiency analysis in 2D grid. *Environment. Technology. Resources. Proceedings of the International Scientific and Practical Conference.* 2, (Aug. 2015), 46. DOI:https://doi.org/10.17770/etr2013vol2.868.