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

State of the Art

Julien Castiaux
Supervisor: Jean-Marie Jacquet

June 2021

Contents

State of the Art **2**
Literature review 2
Game solvers 4
Two-players 4
Single-player 4
Solitaire card games 5
Naive Tree Searches 6
Depth-First and Breadth-First 7
Iterative-deepening 7
Informed Tree Searches 9
Best-first search 9
Multi-step exploration 9
Two-player searches 10
And/Or search 11
Minimax 11
Alpha-beta pruning 11
Goal Generation 13
Learning 14
Genetic Programing 14
Reinforcement Learning 15

References **17**

State of the Art

Literature review

The first step of our study was to identify 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 introduce others two-player information complete 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 asses 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 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" [34] to our reading pile. The book is very complete and explains in depth the key ideas behind autonomous learning; it explains the relationship between exploration (to gain knowledge) and exploitation (of that knowledge), it explains the state-action-reward machinery and various algorithms so the machine quickly gain and exploit knowledge. 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 fulfill 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 science of artificial intelligence and game solvers and 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 3 organises the papers according to (X-axis) their category, (color) 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.

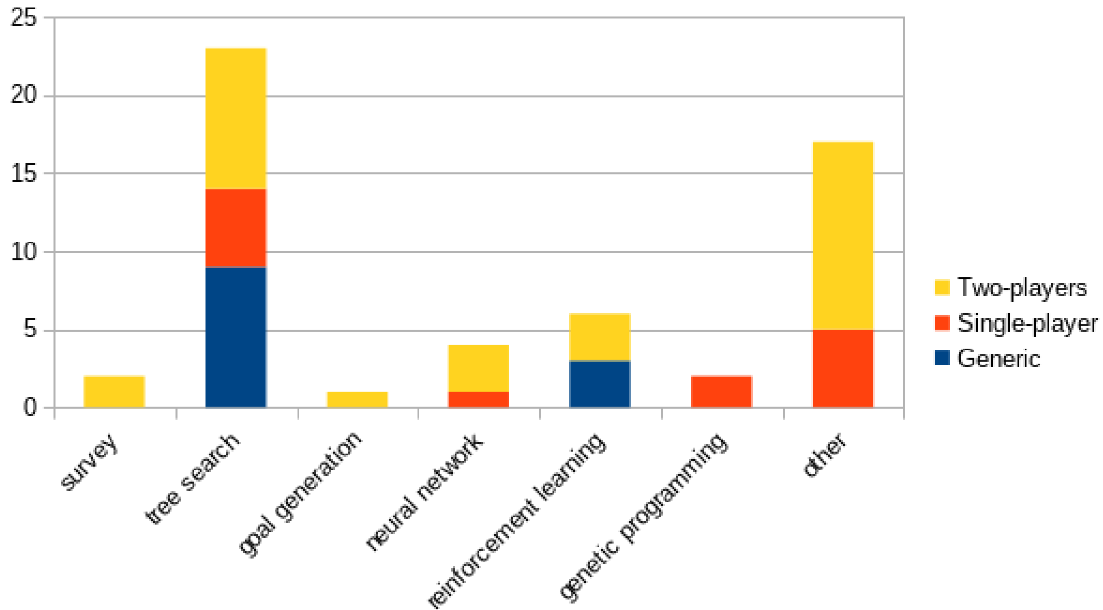


Figure 1: Paper classification chart

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.

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 [28]. During the nineties, thanks to a lot of the 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, 33].

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 to player two-player game which is still in use today [31]. 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” get rejected much faster, and the machine is capable of evaluating much more moves. Not only Chess could benefit from the those advances: other game solvers like Chinook [29, 30] for Chess and Logistello [8] for Othello use the same approaches and both defeated the human world champion in 1992 and in 1994 respectively.

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 [33] 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 [32].

Since this study focuses on single-player games, we recommand 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 [26]. 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

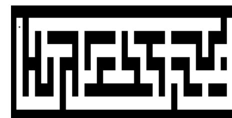


Figure 2: Maze

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.

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 invites more clever algorithms to solve. 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.

Figure 3: Mystic square

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 color 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 colors no longer match.

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 color 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 [22].

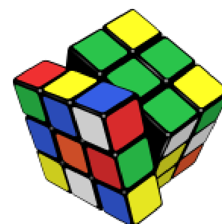


Figure 4: Rubik's cube

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 [22] 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.*

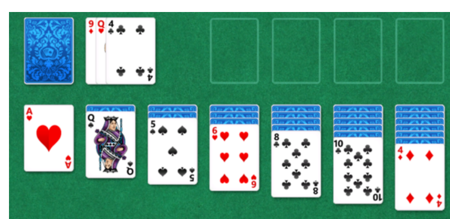


Figure 5: Klondike

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.

Figure 6: Bristol

The three games are also different when it comes to allowed moves. In Klondike and Freecell, the player must alternate colors 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	Color & suit rules	Supermove
Klondike	yes	yes	yes	yes
Thoughtful	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.

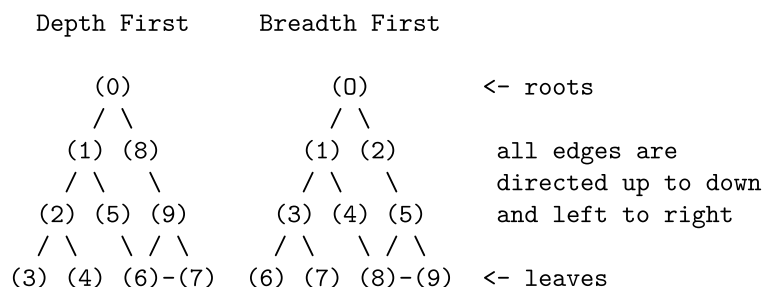
Naive Tree Searches

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 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.

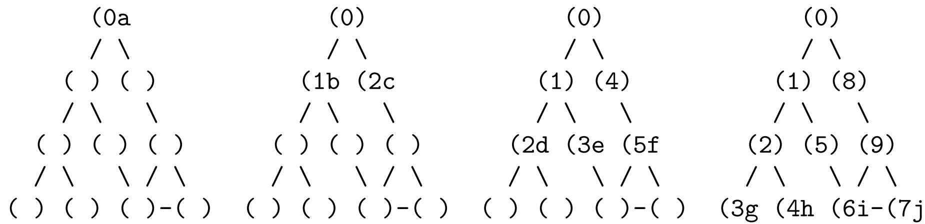
A graph search algorithm is an algorithm that traverses a graph looking for a precise node (or in our case, a node with a given property: game won) that is also capable of determining how to reach that node. In that matter, according to our previous diagram, the two algorithm are different. DF finds 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 just like Depth-First Search (DFS) does 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
```



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, that layer is the most resource greedy ones to compute. The higher the branching factor, the more the last layer is resource greedy compared to the previous layers. It is proven [23] 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 requires a single stack and is memory efficient. Since it iterativaly extends the maximum depth, it finds the shortest path to the node it is searching for.

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

Although it is often described in the literature, extra attention must be accorded 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 [20]. The figure aside showcases a minimalistic directected cyclic graph (DCG), 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)
```

When traversing such graphs, it is necessary 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 minimalistic 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.

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, 21] 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.

Informed Tree Searches

Naive tree searches are virtually never used to solve games. Remember each node is a possible game configuration and that edges are valid moves between configuration. 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 [24].

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 that is capable to explore 1 million nodes per second like Deep Blue is capable 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, 21].

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 [34]. 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 explore 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 new initial node.

To evaluate the next nodes, best-first search uses an evaluation function, a function that takes a game state in 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 [35].

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.

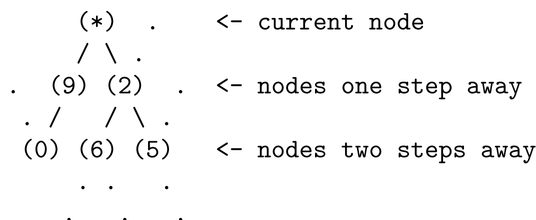
Often the function fails to compute a score that precisely ranks all states. The bird's-eye distance in a maze is an example of an evaluation function that uses an heuristic that is too greedy. It is possible to generate a maze where that heuristic completely fail, a maze where best-first would traverse 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 [19].

Multi-step exploration

Best-first performs a greedy one-step lookahead 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 lookahead 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.



A common belief in the early days of Computer Chess was that the deeper a tree search algorithm was capable of exploring, the better the solver was. It turned out that, in some cases, a 5-steps lookahead solver performed better than a 6-steps lookahead solver. In those cases the 5-steps lookahead solver was making better strategic decisions than it's 6-steps lookahead 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[25], 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 2. To temporarily store cards in the only freecells is basic and offers limited card mobility, some moves require 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 in move, an evaluation function would give a poor score to the game state as the mobility decreased which would ultimately wrongly discard the move.

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 ?"

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 looses. 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.

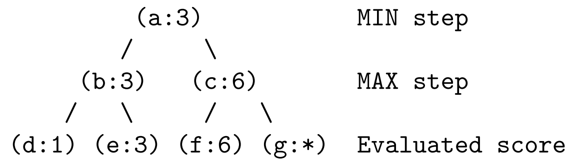
Because it uses heuristics, it is possible to only explore a part of the game graph, to only explore the states n-moves away, and to use the evaluation function to evaluate each explored state. Note that selective deepening considerations do stand, see the “Multi-step exploration” chapter of this work.

Because it uses heuristics, except when the game is over, the evaluation function is incapable 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, those branches can be discarded. We explain “Alpha-beta”, one way to prune a Minimax tree.

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.



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` have 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`, 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 [20] published a survey where the implementation is game solvers is discussed.

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 is possible to minic humans playing Freecell to solve the game ?"

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 chapter 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* [27] 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 over the cells 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 overtime 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 are 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 [34]:

> Reinforcement learning [. . .] is a computationnal 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.

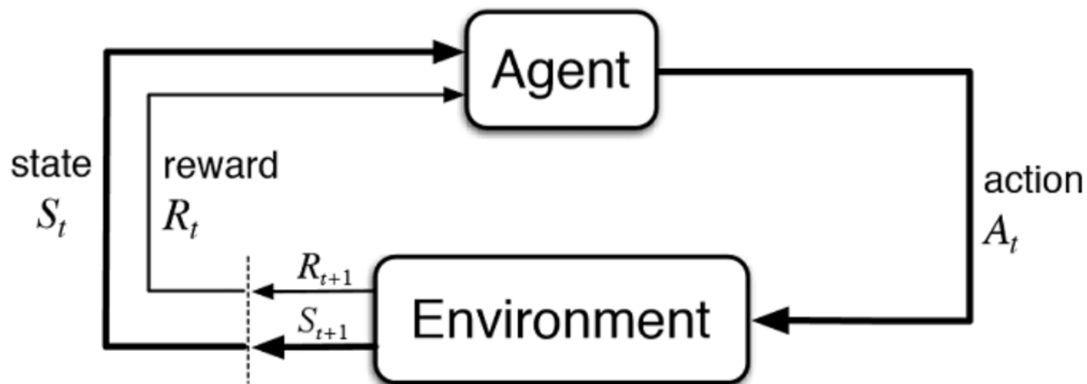


Figure 7: The agent-environment interaction

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 using 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, if the agent uses a same *policy* thoughtful 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.

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 it 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 in knowledge and to improve the approximation of q_π. This stage is known as *the evaluation of q_π for π*.

When the on going evaluation of $q_p i$ 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.

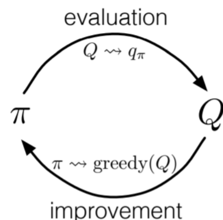


Figure 8: Monte Carlo control process

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 Temporare 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 [33] and Alpha Go Zero[32] (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.

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

[20] 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.

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

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

[23] 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.

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

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

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

[35] 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.