Development of Artificial Intelligence players for the game Crazy 8

Group H

candidate numbers: 32171, 26527, 23498, 22510

Abstract

This project introduces an innovative application of artificial intelligence (AI) in the domain of card games, specifically focusing on the popular Crazy 8 game. The objective is to design and implement an intelligent virtual player capable of strategic decision-making and adaptive gameplay in the dynamic and unpredictable environment of the Crazy 8 card game. To do so, four different players were designed and tested in a repetitive competition where total wins and points obtained were analysed, using alpha-beta algorithms with two different evaluation functions, Q-state Reinforcement Learning, and Monte Carlo Tree Search. All four players showed promising results but also important limitations. Alpha-beta algorithms showed better results without needing a complex evaluation function, but they used perfect information, evaluating moves considering the cards of the opponent, a situation that cannot be extrapolated when playing humans. Monte Carlo Tree search and Q-state Reinforcement Learning algorithms suffered from slow playing and training periods, respectively, that limited their possibilities to defeat different players, but still remained competitive.

1 Introduction

In 2015, almost twenty years after IBM's machine Deep Blue defeated the Grandmaster Garry Kasparov in a chess match (Hassabis (2017)), DeepMind's AlphaGo beat Lee Sedol, one of the world's best Go players (Wang et al. (2016)). Those two victories are considered some of the greatest feats that AI has achieved, due to their complexity and prohibitively big possible game combinations, that didn't allow brute-force computing, and required advances in computation capability and algorithmic tuning (Greenemeier (2017)).

Those two games, besides complexity, show similar characteristics, as both are deterministic, and both players have complete information. Card games, on the other hand, do not share those characteristics, as a player won't ever have complete information, due to the inability to see the rest of the players' hands, and depend on a random environment where the cards in the deck are unknown. For these reasons, businesses and researchers have shown interest in developing solutions for different card games, considering an interesting open problem, with several different solutions in various games, like Poker (Moravčík et al. (2017), Zhao et al. (2022)), Blackjack (Schiller and Gobet (2012)), Bridge (Bouzy et al. (2020)), and many others.

The random environment, incomplete information and competition against players of card games result in the fact that algorithms designed for them can be extrapolated for different problems, like sequential security games where an interested party tries to protect

an important asset, like an airport from a terrorist attack, cybersecurity, or protecting wildlife from poachers (Lisy et al. (2016)); or medical recommendations with a robust decision making process (Chen and Bowling (2012)). Because of it, developing algorithms in the card game field present a special attractiveness.

In this study, we are playing the game known as "Crazy 8". According to game scholar David Parlett, this game was created in the 1930's, and gained popularity during World War II (Parlett (1992)). In it, any number of players between two and four must try to, in order, discard all their cards by playing a card that has either the same rank or suit that the card on top, with the special rule that eights are wildcards, and you can use it any time and change the suit that the next player has to play. Other rules can be invented to change the game, and the game UNO, based in Crazy 8, has become popular all over the world.

Our goal is to create AI algorithms capable of playing the game in a two player setup and evaluate the performance of different algorithms in this scenario. To do so, four different algorithms were trained, two following an AlphaBeta pruning model (Knuth (2014)), but with different utility functions, one under a MonteCarloTreeSearch (MCTS) model (Browne et al. (2012)), and a fourth using a Reinforcement Learning based model (Sutton and Barto (2018)). These four algorithms competed with each other in games with 200 rounds, alternating the first player to play, in order to determine the finest-tuned one, and detect problems, limitations, and opportunities.

To our knowledge, there has been almost no research in the development of Artificial Intelligence players for crazy eights, one example being found in Castillo et al. (2018), but lacks details, references, and comparison possibilities. An example of strategic players of this game can also be found in Sande and Sande (2019). Its variation UNO, has observed a little more interest in AI research, and Pfann (2021), Brown et al. (2020), and Harmendani and Zanetti (2019) showed that Reinforcement Learning players are able to outperform random players in a statistically significant way.

The rest of the paper is divided like this. In the next section, a formulation of the problem that describes the rules of the game and characteristics of the environment are presented. In the following section, the three algorithms used are presented, with the scenarios considered for their application in the experiment. After that, the results of the algorithms playing each other are presented and analysed, and the conclusions close the study highlighting the most important findings, limitations and opportunities.

2 Problem Formulation

2.1 Challenges for the AI agent

Creating an AI player for the Crazy 8 card game poses several challenges due to the diverse states and inherent randomness associated with the game dynamics. For this experiment, the game will be played with two players and following the basic rules of the game.

2.2 Rules of the game

The rules of the game for our setup are defined as followed:

Setup: The game is played with 2 players.

Card Values:

1. Number cards (2-10) except for 8 are worth their face value.
2. Face cards (jack, queen, king) are worth 10 points each.
3. The ace is worth 1 point.
4. The 8 card is special in Crazy 8 and is worth 50 points.

Gameplay:
Initially both the players are issued with 5 cards and another card is selected to be the card on top, all the cards are randomly drawn from the deck. Players take turns playing a card from their hand to replace the card on top. The card played must either match the rank or suit of the card on top. (For example, if the card is a 7 of hearts, players can play any other 7 or any heart card. If a player cannot play a card, they must draw cards from the deck until they draw one that can be played. The 8 cards are special. When played, the player must declare the next suit. The next player must then play a card of that suit or another 8.

Goal state:
The game continues until a player successfully plays all of their cards, the first player to empty their hand is the winner. The player will then receive points equal to the value of cards of the other player. The game ends in a tie when both the players have more than 1 card and the deck becomes empty.

2.3 AI Agent and its task environment

An AI agent, as defined by Russell and Norvig (2010), refers to a computational entity equipped with algorithms and mechanisms that enable it to perceive its environment, reason about information, learn from experiences, make decisions, and take actions autonomously. In order for the agent to simulate and produce the best possible move, we need to define the task environment, we do that using the PEAS framework.

Agent type	Performance Measure	Environment	Actuators	Sensors
Crazy 8 card game player	Total points/wins won against other players.	Cards of both the players, deck and card on top.	Suggest best moves.	Moves played by the other player.

Table 1: PEAS description of task environment

The AI-driven Crazy 8 card game player is designed with the primary goal of maximising performance, quantified by either total points accumulated or victories secured against opponents. Its actuators strategically suggest moves, taking into account the dynamic environment constituted by the cards of one or both players, the deck, and the top card on the discard pile. The player's decision-making process is informed by sensors focused on monitoring the opponent's moves, providing essential data for adaptive and strategic gameplay. This comprehensive framework aims to guide the AI player in navigating the complexities of the Crazy 8 card game, ensuring a strategic approach that enhances its likelihood of success in terms of points and wins. The environment in our case is to be

reproducible and has a continuous state space. This is done to make the most out of our agents, although this might hinder the applicability in real life.

3 Proposed Solution: agents to play the Crazy 8 card game

To solve this environment, three algorithms are studied. Each one of those three are adapted to the rules described in the prior section. In this section, all algorithms are explained, including considerations and time consumption of the algorithms.

3.1 Alpha-beta algorithm

Understanding the minimax method is essential before we comprehend what alpha-beta pruning does.

Minimax constitutes a backtracking algorithm employed in decision-making and game theory to determine the most advantageous move for a player, assuming that the opponent is also making optimal decisions. Its application is prevalent in two-player turn-based games, including but not limited to Tic-Tac-Toe, Chopsticks and Chess. These games fall under the category of zero-sum games, where the mathematical representation signifies that one player's victory corresponds to a positive outcome (+1), the opponent's victory results in a negative outcome (-1), and the scenario where neither player wins equates to a neutral outcome (0).

Alpha beta pruning: Our player uses alpha-beta pruning, an optimisation strategy under the minimax algorithm, this technique helps to bring down the computational costs. In the absence of alpha-beta pruning, all the possible states are explored before suggesting the best move. With it, the branches of certain aspects of the game are cut off when we already know the best possible move that maximises the utility function. This can be seen in the algorithm 1.

The time complexity of the Alpha-Beta Pruning algorithm is expressed in terms of the number of nodes explored in the game tree. In the best-case scenario, when the pruning is effective, the time complexity is significantly reduced compared to the naive minimax algorithm. Best case scenario: $O(b^{d/2})$ where b is the branching factor and d is the depth of the tree, effective pruning cuts down the exploration factor by half at every level.

Two different players with different utility functions were created for comparisons:

Alpha-beta player 1: This method adopts a sophisticated approach, leveraging a heuristic evaluation function to determine the points accrued by the player upon reaching a terminal state. The evaluation function takes the total points of a player in the terminal state, determined by the times each card i appears in the player hand, T_i, multiplied by the value v_i of each card according to the rules presented in the prior section.

$$Points = \sum_{\forall i} v_i \times T_i$$

Alpha-beta player 2: This method employs a concise yet effective utility function, where strategic decisions are guided by a clear points system. The utility function assigns 1000 points in the event of Player 1 emerging victorious, -1000 points if Player 2 secures the win, and awards 0 points for a tie. This streamlined approach streamlines decision-making, enhancing the player's ability to navigate the game tree efficiently.

Algorithm 1 Alpha-Beta prunning

```
1: function CRAZY_8_GAME(game_state)
2:     return MAX-VALUE(legal_possibles_moves, game_state, −∞, +∞)
3: function MAX-VALUE(legal_possibles_moves, game_state, α, β)
4:     if game.IS-TERMINAL(game_state) then
5:         return game.UTILITY(game_state, move)
6:     v ← −∞
7:     for every move in legal_possible_moves(game_state) do
8:         v2, a2 ← MIN-VALUE(updated_game_state, α, β)
9:         if v2 > v then
10:            v, move ← v2, a
11:        α ← max(α, v)
12:        if v ≥ β then
13:            return (v, move)
14:    return (v, move)
15: function MIN-VALUE(legal_possibles_moves, game_state, α, β)
16:    if game.IS-TERMINAL(game_state) then
17:        return game.UTILITY(game_state, move)
18:    v ← +∞
19:    for every move in legal_possible_moves(game_state) do
20:        v2, a2 ← MIN-VALUE(updated_game_state, α, β)
21:        if v2 < v then
22:            v, move ← v2, a
23:        β ← min(β, v)
24:        if v ≤ α then
25:            return (v, move)
26:    return (v, move)
```

3.2 Reinforcement Learning algorithm

Reinforcement Learning is a paradigm where an agent learns to make decisions by interacting with the environment, aiming to maximise the cumulative reward signal over a sequence of actions. At every step, the agent receives feedback according to the state and action followed. For this problem, an active Q-learning over a function approximator method was used.

Considering that there are over 2,500,000 initial hands possible in the game, an approximator with features of the environment is used to train our agent. The features considered where the number of cards in the hand of the player, the number of cards in the hand of the opponent, both capped in a value of 8 -meaning that for the agent have eight or more cards is equivalent-, the suit in play, the number of eights in the hand, the number of cards in the same rank, and the number of cards of the same suit, capped in a value of 4, as can be seen in the following equation. This definition gives importance to the playability of the hand and the moment of the game, instead of the exact representation of the environment,

while reducing the total states to 949,888. In the same way, only five different actions are defined: playing any of the suits, or playing an eight. The action of drawing a card is not included as part of the state-action pair (S, A) as it can only be done if there are not any other actions possible.

$$S = (\#card\_player, \#card\_rival, suit, \#\_eights, \#\_card\_same\_rank, \#\_card\_same\_suit)$$

$$A = (Hearts, Diamonds, Clubs, Spades, Eights)$$

The active Q-learning method was used to balance the exploration vs exploitation problem. This allows our agent to explore different actions in different states while exploiting its current knowledge. For its implementation, an epsilon of $\epsilon = 0.4$ was used, meaning that 40% of the time the player will choose a random possible action, while the other 60% of the time the player will follow a greedy strategy, selecting the action with a higher expected reward. After defining a rewards matrix R where the reward is 1 in case that the action implies a winning result and 0 in every other case, the Q function is initialised with values equal to zero and updated following the Bellman Equation presented at the end of the section. Due to the randomness of the card game and the dependence to the rival play to foresee the result of each play, the update of Q states has to be achieved by looking back to the previous state and action (S', A'), instead of doing a look up for the future deterministic result. To do so, the algorithm must be adapted as can be seen in the algorithm 2, and trained for a total of 600,000 games, using a step size α of 0.2, in order to update the Q-values and enhance performance.

Algorithm 2 Q-Learning Algorithm

```
Q ← 0                                                    ▷ Initialize Q-values
for each game do
    Start game
    for each move in the game do
        Calculate feature state
        Choose action following epsilon-greedy strategy
        Update Q-value of previous state and action following Bellman Equation
        Go to next move
```

$$\text{Bellman Equation} = Q(S', A') \leftarrow Q(S', A') + \alpha \left[R + \max Q(S, A) - Q(S', A')\right]$$

3.3 Monte Carlo Tree Search algorithm

The main idea behind the Monte Carlo Tree Search player is to generate a move by evaluating all the possible moves given the game's state via simulation. For each move, the player simulates the game a thousand times randomly after that move is played. Afterwards, the algorithm evaluates each move by looking at what proportion of the games the player won after playing that move and chooses the move with the highest proportion of wins. The key assumption of this approach is that, on average, the best move overall should lead to the highest proportion of wins if all the other moves are random.

A tradeoff faced by every Monte Carlo Tree Search algorithm is the one between exploration and exploitation. In essence, when making the decision on which move should be played next, the algorithm could rely on the current estimates of which moves are most promising based on the simulations performed so far (exploitation) at the expense of exploring the moves that have not been explored before extensively but could potentially perform the moves that have been explored. A common way to address that dilemma is to choose the next leaf to be expanded based on the upper confidence bound formula, which has the following form:

$$\frac{U}{N} + C \times \sqrt{\frac{log(Nparent)}{N}}$$

Where U is the total accumulated value of the node so far (the number of wins it led to), N is the number of times the current node has been visited, $Nparent$ is the number of times the parent node has been visited and C is a constant. The value is equal to infinity if $N = 0$ (which means the node has never been explored). This formula is the basis of addressing the exploitation-exploration tradeoff in our agent.

One of the advantages of Monte Carlo Tree Search is that one can implement the algorithm for any game (or problem in general) even with a very basic understanding of it. In particular, one does not need to create an evaluation function (which can be an especially difficult task for games without a straightforward way of assessing whether a position is winning) to implement the algorithm. The only necessary thing is to implement the transition model, that is, the way in which the state of the game changes as players play the moves. This characteristic makes the Monte Carlo Tree Search approach suitable for novel problems or ones that cannot be solved using simple rules of thumb.

The implementation of the Monte Carlo Tree Search algorithm used in this analysis was borrowed from the AIMA repository on GitHub (Bacon et al. (2024)). The algorithm consists of three main steps. The first step is the selection step, which involves choosing which unexplored move (a leaf of the tree) the algorithm should explore next. The next step is the expansion step, which involves playing all the possible moves that are available at that leaf (unless the game ends at that leaf, in which case the algorithm moves to the backpropagation step without expanding) and, in that way, creating several children nodes and choosing one of the children for simulation. Afterwards, the game is simulated randomly for that child until it terminates. Finally, the results are backpropagated all the way to the root node. An original object-oriented implementation of the Crazy Eights game was used to make the Monte Carlo Tree Search algorithm applicable to the problem at hand.

4 Numerical Experiments

The capability of the different algorithms were tested in a competition where all players played each other. To take into account randomness, both in the game and in different players' strategies, the results were aggregated after 200 games, considering both points earned in each game and total number of wins, ties, and losses. In half of them an algorithm will play first while the other plays second and in the other half the positions will shift. This competition was run on a desktop machine with 4 cores, each at 2.4 GHz, took 2:30 hours to run, and the results can be seen in table 2 and 3. In table 2, the results are presented as the

sum of points that each player earned during the competition against each other, presenting player 1's points first, and in table 3, we observe the wins of player 1 first, followed by the wins of player 2, and finally the ties, adding to 100 hundred for each pair of players in the corresponding order.

Player 2	Alpha-beta p1	Alpha-beta p2	MCTS player	RL player
Player 1				
Alpha-beta p1	NA	$1024 - 1331$	$1881 - 1007$	$1210 - 1123$
Alpha-beta p2	$1024 - 1007$	NA	$1750 - 1280$	$1091 - 912$
MCTS player	$1044 - 1648$	$1018 - 1577$	NA	$1192 - 1585$
RL player	$1285 - 875$	$922 - 1234$	$1401 - 1263$	NA

Table 2: Comparison of total points among different agents after 100 games in that order

Player 2	Alpha-beta p1	Alpha-beta p2	MCTS player	RL player
Player 1				
Alpha-beta p1	NA	$49 - 48 - 3$	$58 - 36 - 6$	$55 - 43 - 2$
Alpha-beta p2	$53 - 47 - 0$	NA	$48 - 41 - 11$	$55 - 45 - 0$
MCTS player	$47 - 47 - 6$	$48 - 48 - 4$	NA	$44 - 46 - 10$
RL player	$54 - 44 - 2$	$43 - 54 - 3$	$43 - 51 - 6$	NA

Table 3: Comparison of wins among different agents after 100 games in that order

We observe that the bot designed following an Alpha-beta algorithm obtained better results with either evaluation function, both in points obtained and in total number of wins. However, the simpler evaluation function that gave points for a win and gave negative points for a loss (Alpha-beta player 2) showed a better performance that every other player, included the Alpha-beta algorithm with a more complex evaluation function, averaging 56.1% of the points and a win ratio of 51%, plus a tie ratio of 3.5%.

It is also observed that the Monte Carlo Tree Search algorithm was the player with the worst results in points and in total wins as a second player, while also obtaining a larger amount of draws between the four players. Both results can be explained by the limitations of the algorithm due to computation capability, as it is computationally heavy and needs to be trimmed earlier, affecting its capability of finding a winning strategy, but still being able to pursue a tie.

On the other hand, the Reinforcement Learning algorithm shows results in between Alpha-Beta and Monte Carlo Tree Search. This is also explained due to this algorithm requiring training, as the initial Q-state matrix is empty and it is filled during the training session. However, the training was constrained by the computation capabilities, limiting it to only 600,000 games, even though there were almost 1 million states, meaning that probably were not enough repetitions to identify better strategies. Both Reinforcement Learning and Monte Carlo Tree search algorithms play with incomplete information, while Alpha-Beta algorithms have perfect perception of the environment.

5 Conclusions

As the game depends on high levels of randomness, better players still won't be able to win a vast majority of the games. However, good strategies could help to avoid losing and obtain higher points, and all three algorithms were able to learn and obtain results above random level.

The results were satisfactory as AI was used effectively to produce algorithms able to play the game with short move time, and produce strategies focused on obtaining higher points and more wins. Alpha-Beta algorithms were the most effective in that aspect, but all three showed strengths and limitations.

Despite its efficiency, the Alpha-Beta pruning function is constrained by its reliance on a deterministic environment. The assumption of full access to both players' cards and knowledge of the entire deck is pivotal for accurate point calculation. In real-life scenarios, these conditions may not always be feasible, introducing a limitation to the algorithm's applicability in situations where complete information about the game state is not available.

Monte Carlo Tree Search was able to tie players trained and with complete information in total wins, but was the slowest algorithm and obtained less points that the rest. On the other hand, even though the Q-state Reinforcement Learning algorithm was not designed considering the points of the hand in their rewards, it optimises the points obtained per game. However, due to the high number of states, it requires a larger training time to obtained optimal strategies.

This work set foundations for further research in the area of AI development in the Crazy 8 game. New research could work in open questions of this study in any of the algorithms designed, as there are interesting open questions in all of them. How could Monte Carlo Tree search could be optimised to take shorter run times and improve its results? Are there ways to minimise the Q-state Reinforcement Learning training steps, and if not, after how many repetitions will the algorithm be optimised? Can the alpha-beta algorithm be adapted to perform as effectively in an incomplete information setup?

References

Darius Bacon, Phil Ruggera, Peng Shao, Amit Patil, Ted Nienstedt, Jim Martin, and Ben Catanzariti. Aimacode/aima-python: Python implementation of algorithms from russell and norvig's "artificial intelligence - a modern approach", 2024. URL `https://github.com/aimacode/aima-python?tab=readme-ov-file`.

Bruno Bouzy, Alexis Rimbaud, and Véronique Ventos. Recursive monte carlo search for bridge card play. In *2020 IEEE Conference on Games (CoG)*, pages 229–236, 2020. doi: 10.1109/CoG47356.2020.9231667.

Olivia Brown, Diego Jasson, and Ankush Swarnakar. Winning uno with reinforcement learning, 2020. URL `https://web.stanford.edu/class/aa228/reports/2020/final79.pdf`.

Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

Matias Castillo, Benjamin Goeing, and Jesper Westell. An artificial intelligence for a crazy eights game, 2018. URL `https://web.stanford.edu/class/archive/cs/cs221/cs221.1192/2018/restricted/posters/jesperw/poster.pdf`.

Katherine Chen and Michael Bowling. Tractable objectives for robust policy optimization. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL `https://proceedings.neurips.cc/paper_files/paper/2012/file/ce5140df15d046a66883807d18d0264b-Paper.pdf`.

Larry Greenemeier. 20 years after deep blue: How ai has advanced since conquering chess, Jun 2017.

Pedro Harmendani and Márcia Zanetti. Aprendizado de máquina por reforço aplicado no jogo de cartas uno. *Revista de Sistemas e Computação, Salvador*, 9(2):245–251, 2019.

Demis Hassabis. Artificial Intelligence: Chess match of the century. *Nature*, 544(7651): 413–414, April 2017. ISSN 1476-4687. doi: 10.1038/544413a. URL `https://doi.org/10.1038/544413a`.

Donald E Knuth. *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014.

Viliam Lisy, Trevor Davis, and Michael Bowling. Counterfactual regret minimization in sequential security games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack:

Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017. doi: 10.1126/science.aam6960. URL https://www.science.org/doi/abs/10.1126/science.aam6960.

David Parlett. *A dictionary of card games*. Oxford University Press, 1992.

Bernhard Pfann. Tackling uno card game with reinforcement learning, Jan 2021. URL https://towardsdatascience.com/tackling-uno-card-game-with-reinforcement-learning-fad2fc19355c.

Stuart J Russell and Peter Norvig. *Artificial intelligence a modern approach*. London, 2010.

Warren Sande and Carter Sande. *Hello world!: computer programming for kids and other beginners*. Simon and Schuster, 2019.

Marvin R. G. Schiller and Fernand R. Gobet. A comparison between cognitive and ai models of blackjack strategy learning. In Birte Glimm and Antonio Krüger, editors, *KI 2012: Advances in Artificial Intelligence*, pages 143–155, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-33347-7.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Fei-Yue Wang, Jun Jason Zhang, Xinhu Zheng, Xiao Wang, Yong Yuan, Xiaoxiao Dai, Jie Zhang, and Liuqing Yang. Where does alphago go: from church-turing thesis to alphago thesis and beyond. *IEEE/CAA Journal of Automatica Sinica*, 3(2):113–120, 2016. doi: 10.1109/JAS.2016.7471613.

Enmin Zhao, Renye Yan, Jinqiu Li, Kai Li, and Junliang Xing. AlphaHoldem: High-Performance Artificial Intelligence for Heads-Up No-Limit Poker via End-to-End Reinforcement Learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36 (4):4689–4697, June 2022. doi: 10.1609/aaai.v36i4.20394. URL https://ojs.aaai.org/index.php/AAAI/article/view/20394.