Hugo Kitano
Daniel Gallegos
Marco Monteiro

An Artificial Intelligence For Checkers

Introduction

Checkers is a century-old game that Computer Scientists have studied since the 1950s. The game is extremely complex, with roughly $5x10^{20}$ possible positions, about a million times more complex than Connect Four. In 1992 a computer program won the right to play in the Checkers World Championships, and later in 1994 a computer became World Champion. In 2007, after more than twenty years of research, Jonathon Schaeffer at the University of Alberta announced that his team had solved checkers and created a program “Chinook” that played it perfectly. They manually solved every board with fewer than ten pieces, and from there showed that all other boards could be reduced to a set of already solved boards. Then, they showed that all boards in any set had the same outcome. Though Chinook is very powerful, and impossible to beat (with perfect play it will draw), it was very computationally intensive. We could not find any less computationally intensive AIs online that were still challening to beat. For our final project, we will be creating an artificial intelligence for the game of checkers.

Task Definition

Our project’s focus on the game of checkers will be to create an artificial intelligence that a user can play against. The main task of this artificial intelligence is to gather the available moves, select the strongest action, and update the state every turn. In checkers, this is more complicated than what may seem at first glance, since every action could have ramifications several turns in advance, and strategy in positioning of pieces is crucial to good gameplay. Given our limited computational power and time (unlike the researchers who created Chinook), our main objective is to provide a artificial intelligence that can beat most mid-level checker players.

We used several methods to evaluate the agents we created. One of our evaluation metrics compares our agents' moves to those of checkers grandmasters. Given a specific board derived from checkers transcripts of grandmasters' games, we compared the moves our agents suggest with the moves the grandmasters make. Second, we played our agents against each other to see which among them is the best. Third, we played our agents against Chinook, our perfect oracle, to see how well they held up. Finally, we evaluated our best algorithm against us and other Stanford students to see how they fared.

Infrastructure

We created a few tools to help us create the artificial intelligence. Our goal in creating the infrastructure was to emphasize flexibility, compartmentalization, and seamless transfer. This organization would help us minimize code duplication and easily debug errors. We also decided to use JavaScript for most of our codebase because of its ease of integration into our web interface. Ultimately, we ended up writing over five thousand lines of code.

Game

We modeled the game of checkers as a two-player, adversarial, zero-sum game. We defined a state as an 8x8 grid with each cell denoting the piece present (or lack thereof). We described the game's start state, end states, actions, and successors, following the definition of a game. One rule we decided to add to our game of checkers was requiring the capture of pieces if possible, an official rule in competitive gameplay not commonly adhered to in casual play. The implementation of the rules of checkers was complex, especially in the instances where we promoted a pawn or captured multiple pieces in one move. The majority of the work in the creation of the checkers game class was spent enumerating the legal actions for an agent given a state.

Agents

The agent classes we created abstract out the complexities of the games, relying on only a few inputs to choose the strongest possible move. These inputs are current state and a list of possible actions from our checkers game class, the factors necessary for the evaluation function from the factors class, and and a gameplay algorithm from the gameplay class. By creating the agents with this framework, we can mix and match different modules to create a variety of agents.

Evaluation

We created several scripts that evaluate how good our agents are. One script simulates games without the user interface, usually by having two agents compete against each other. In the case of comparing our agents to grandmasters, we created an entire process to scrape game information from World Checkers Championships in the 2000's. We translated the traditional checkers notation into our game framework to compare our agents' moves to the grandmasters'.

Interface

We created a simple user interface using Facebook React to visualize our game and to play against the various agents we created. It allows users to click on pieces they want to move, and shows them the legal moves possible for that piece. The user can then select the move they desire. This user interface also helped us debug the model of the checkers game we created.

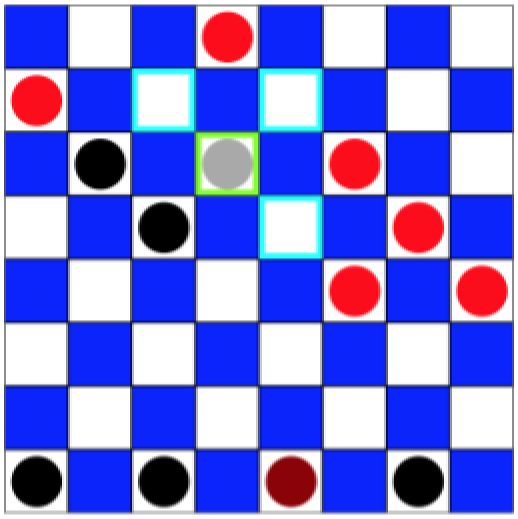


Fig 1. Example of Checkers Interface

Approach

Our baseline agent was simple: choosing a random move from the legal actions available. Of course, it was terrible, using no intuition and looking no moves ahead. From there, we implemented artificial intelligence concepts we learned in class and beyond to create a variety of agents.

Search

The rest of our baseline agents used a minimax framework, the standard algorithm for zero-sum games, because it assumes the other player will play their best move (rather than expectimax, for example). Since the move tree has many branches and is too deep to search through in entirety, we limited our search at depth = 4, at which point we used a heuristic to estimate how favorable the state we reached was. We implemented alpha-beta pruning to increase our efficiency and speed. Finally, we implemented quiescence search, an algorithm that changes its maximum depth of search depending on the situation. For our agents, we searched deeper when there were situations where pieces were taken, as the number of possible moves is greatly decreased when a player can take a piece. This change allows us to search deeper when we “trade” pieces, decreasing the chances that we will reach the end of our search depth before properly analyzing the full extent of trade situations.

Our first “smart” agent, leveraging this minimax framework, featured the simplest heuristic imaginable: the difference between each player’s number of pieces. It was surprisingly difficult for us to beat the AI, fulfilling the objective we had set for ourselves at the beginning of the project (to be able to compete decently with average checkers players). It also set up a second baseline (other than our random agent) from which to compare our other agents to.

Heuristic

For more advanced agents, we decided to beef up our heuristic’s features and train its weights vectors. To make our heuristic more comprehensive, we created 19 factors that took into account features like the number of pawns and kings, control of the center of the board, and “safe” pieces on the edges of the board. We got the ideas for these factors from a paper by researchers at the Warsaw University of Technology, who also tried to use heuristics to create a checkers AI. We mix and match these factors for our various agents, and calculate linearly.

We also realized an innate property of the game of checkers that we decided to implement in our heuristic. We noticed that our agents do well in the beginning and middle of the game, but struggle in the endgame, when positional rather than tactical play is more important. Specifically, we found that kings would not patrol the center of the board as checkers theory suggests, and instead idles back and forth along the periphery. Thus, we created a separate vector of weights for the endgame, which we defined as a state where either player has 5 or less pieces.

Training

We set the weights for our features in two separate ways. Our first method was an evolutionary algorithm that followed a repeated two-step process: it would select the best weight via a tournament among a group of weights, and randomly generate new weights around that weight. We then ran this process almost a thousands times to get our results. The second is a simple handpicked strategy, where we simply set the weights based on our own game knowledge.

We also used attempted to learn the weights with TD learning. The general approach was to randomly initialize a weight vector, and use that vector to play two agents agents against each other. Initially, actions only had a reward when an agent won or lost. Also, the agent that learned from TD-learning used no lookahead. It choose moves by running its evaluation function on all successor states, and chose the state with the highest score. We used the following formula to update weights.

$$w \leftarrow w - n[V(s; w) - (r + \gamma V(s'; w))]\nabla_w V(s; w)$$

We used a linear model so the gradient in the above equation was just our feature vector. We found that without assigning rewards between moves, the learning algorithm did a very poor job of learning. This was because with essentially random move making, both AI's very infrequently captured pieces, and very rarely did games in in a win or loss. To assist the learning, we added a reward for capturing pieces, and changed our learning agent to have two levels of look ahead with minimax. This helped the learning make more "interesting" moves that it could learn from.

Our Agents

We ended up creating nine agents, as follows.

Agent	Description
Random (1)	Picks a move at random
Naive Minimax (2)	Minimax, using a heuristic that simply calculates the difference between pieces
8-Factor Evolutionary (3)	Minimax, uses the 8 most basic factors in a heuristic, with weights trained via our evolutionary algorithm.
8-Factor with Endgame Evolutionary (4)	Minimax, uses the 8 most basic factors in a heuristic that has separate weights for endgame play, with weights trained via our evolutionary algorithm.

19-Factor Evolutionary (5)	Minimax, uses all 19 factors in a heuristic, with weights trained via our evolutionary algorithm
TD-Learning (6)	Minimax, uses the 8 basic factors in a heuristic, with weights trained via TD-Learning
Handpicked (7)	Minimax, uses the 8 most basic factors in a heuristic, with weights chosen from our own game knowledge.
Handpicked with Endgame (8)	Minimax, uses the 8 most basic factors in a heuristic that has separate weights for endgame play, with weights chosen from our own game knowledge.
Handpicked with Endgame with Quiescence (9)	Minimax with quiescence search, uses the 6 most basic factors in a heuristic that has separate weights for endgame play, with weights chosen from our own game knowledge.

Literature Review

The history of creating artificial intelligence for checkers is deep and fascinating. It started with Arthur Samuel in the 1950's, who used rote learning and minimax to create a competent AI, despite poor computation power in that era. Interestingly, Samuel was one of the first computer scientists to implement alpha-beta pruning, a technique we implemented. However, his evaluation heuristic was simpler than ours, and for at least part of his research, he kept his weights entirely constant, preferring to optimize his search in other ways (including rote learning). Nevertheless, our program and his are aligned in most ways: both use similar frameworks, and do not reach expert levels of play.

The artificial intelligence we aligned our project most with is that of a group of researchers from the Warsaw University of Technology. We ended up using some of the nineteen board features that they developed for their heuristic. Our projects take into account stages of the game for our heuristic: they divide the game into three stages, and we divided it into two. We both used an evolutionary genetic algorithm to train it, though ours is a simplified version of theirs.

The most successful checkers program is Chinook, the program the University of Alberta created that plays the game optimally. The basic approach was to create an endgame database with a definite outcome, and prove that every state in the game would be reduced to one of these endgames. Since this approach's goal is to completely solve the game of checkers, it is entirely different from our framework, which cannot search deep enough. However, this program is a great oracle that we can compare our agents with, even though it's impossible to beat and very, very difficult to draw.

Error Analysis

Our first evaluation of the success of our agents was through holding a tournament between all of them. Each agent got to play every other agent starting as red and black. The number in the table, below corresponds in the enumeration of agents in the chart above. For all pairings, the agent listed on column 1 played black.

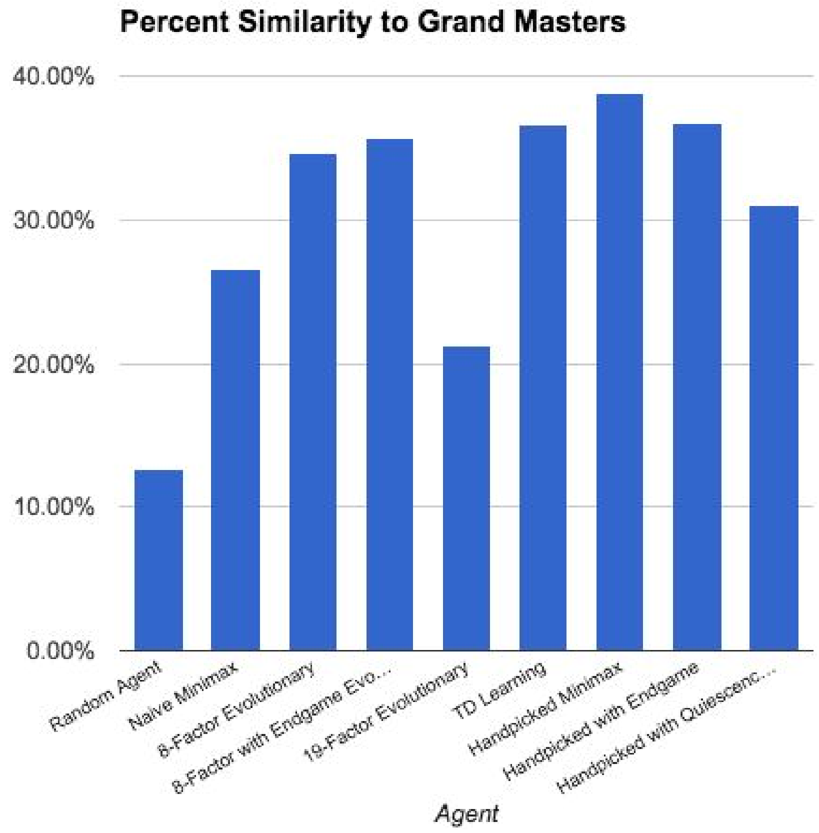
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
(1)		Tie	(3)	Tie	(5)	Tie	Tie	Tie	(9)
(2)	Tie		Tie	Tie	(5)	Tie	Tie	Tie	Tie
(3)	(3)	Tie		Tie	Tie	Tie	(3)	(3)	Tie
(4)	Tie	(4)	Tie		Tie	Tie	Tie	Tie	Tie
(5)	(5)	Tie	Tie	Tie		Tie	(5)	(5)	(9)
(6)	Tie	(6)	(3)	Tie	Tie		(7)	Tie	Tie
(7)	Tie	Tie	Tie	Tie	Tie	(7)		Tie	Tie
(8)	Tie	Tie	Tie	(8)	Tie	Tie	Tie		Tie

(9)	(9)	Tie	Tie	(9)	Tie	Tie	Tie	Tie	■

These results are really helpful to discern the relative strengths of each agent. However, it does not tell us the absolute strength the agents against either human players or Chinook, our oracle. The majority of games ended in draws. This is to be expected, given that most high-level checkers play ends in draws. For example, over 60% of World Championship Games, all end in draws. Furthermore, in the end game, all of our AI's are very risk adverse. From watching games, we saw that the agents tend to move their pieces to opposite ends of the board, and stay there. This makes sense given unless the agent sees a series of moves within its depth that will result in it winning pieces, there is very little incentive to attack. Once solution is to add a weight to our feature vector that encourages aggressive moves. Agents (5) - TD Learning, and (9) - Handpicked weights with optimized endgame and quiescence search had the most wins.

Our second evaluation was a comparison between our agents and grandmasters. We extracted over 400 moves across 5 Checkers World Championships, and measured how many times our agents made the same move as the grandmaster.[1] These results give us an idea of how similar our agents think to the best human players. This analysis does not take into account the fact that some moves are much more important than others. Another shortcoming of this analysis is that it doesn't differentiate between average, or bad moves. Any time the agent doesn't match the grandmaster, it is counted as a mismatch. Finally, even though grand masters are the best human players in the world, they are not guaranteed to always make the best moves. In some of these test cases, our AI might make better moves than the grandmasters.

[1] Links to Checker World Championship transcripts can be found here: http://www.usacheckers.com/show_game_start.php?event=2005wtm



Of all the trained Heuristics, TD Learning performed the best against grand masters. Of the handpicked heuristics, the simple handpicked minimax performed the best. The 19-factor evolutionary agent performed significantly worse than all of the other non-random agents. The extra parameters could have caused over-training.

Lastly, we played all of our agents against Chinook, our oracle. All of them lost, as expected. However, TD-Learning, and the simple Handpicked Heuristic agent were able to make it to the endgame with equal numbers of pieces on both sides, then crumbled due to poor positional play. A perfect checkers player, alas, is too strong for the vast majority of AI's and people.

Overall, with the exception of the 19-factor evolutionary agent, and naive minimax agent, nobody could beat any of our trained, non-random agents. We think both the evolutionary, and TD-learning agents have great potential. The next step for the TD-learning algorithm is to train through a neural network. Hidden layers could detect insights about the game we couldn't find. Also, some of our features might have been nonlinear. The greatest room for improvement is in the endgame. All of the agents could not understand how to gain positional advantages in the endgame, including strategies like gaining central control and chasing other pieces.

Conclusion

Ultimately, we met our initial objective: to create a checkers artificial intelligence that would defeat average checkers players. In fact, our handpicked agent went completely unbeaten among us and our peers. Our efforts to create stronger agents through more comprehensive heuristics and weight training met some success as well. Clearly, heuristic X and heuristic Y were strong.

From a broader perspective, our motivation for this project was to gain insight into game AI's. Through the process of creating a framework that included algorithms like minimax, alpha-beta pruning, a complex heuristic definition, and evolutionary algorithms, we learned how to build an AI from the bottom up, and can readily apply them to future projects we may encounter in the field of artificial intelligence and machine learning.

Bibliography

Kusiak, M., Waledzik, K., Mandziuk, J.: Evolutionary approach to the game of checkers. In: Adaptive and Natural Computing Algorithms, pp. 432–440. Springer Berlin Heidelberg (2007)
(http://www.mini.pw.edu.pl/~mandziuk/PRACE/ICANNGA07-1.pdf)

Samuel, A.L.: Some studies in machine learning using the game of checkers. IBM J. Res. Dev. 210–229 (1959)
(http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.368.2254&rep=rep1&type=pdf)

Schaeffer, J., N. Burch, Y. Bjornsson, A. Kishimoto, M. Muller, R. Lake, P. Lu, and S. Sutphen. "Checkers Is Solved." Science 317.5844 (2007): 1518-522. Web. (http://www.cs.cornell.edu/courses/CS6700/2013sp/readings/06-b-Checkers-Solved-Science-2007.pdf)