Study notes - Computer Go: An IA oriented survey

Julien Castiaux Jean-Marie Jacquet

January 2021

This work are study notes of the survey published in 2001 by Bruno Bouzy and Tristan Cazenave entitled "Computer Go: An IA oriented survey" published in 2001 in the *Artificial Intelligence* journal volume 132.[1]

Context

The game of go is a two players turn-based strategic board game popular in Eastern Asia. Two players play one after the other to place stones on the board in order to control the most territory. One player can capture some of his opponent territory by surrounding his opponent's stones. The game ends when the two players agree they don't have any valuable move left to do, the one who controls the most territory wins the game.

Computer Go is a branch of artificial intelligence that studies and develops new ways and tools to create a software capable of playing the game of go better than humans do. "Computer Go: An IA oriented survey" is a survey that lists various artificial intelligence methods that have been studied to create such a software. It evaluates every of those methods and compare them. The authors also list what are the promising approaches, the approaches that give interesting results but which have not been studied enough yet.

Comparison to others games

The paper first begins by introducing other games like Chess, Checker and Othello, it briefly explain the rules of each games, what are the computing challenges and elaborate a bit about the current best artificial players for each game. We learn that the most commonly used techniques to build strong players are related to tree-search. The above three games are considered solved because it exists at least one program that is capable of defeating the best professional human players. They all three use tree-search as their primary technique.

The introduction also explain how artificial players are evaluated. From time to time, either during seasonal competition either via simple invitation, the

programs play against each other or against human players of various ranks. When a program is capable of beating players of a certain rank, we say the program achieved the same rank. When the world human champion is no more capable of beating an artificial player, the game is considered solved.

One way to determine how difficult a strong artificial game player will be hard to code is to compute the theoretical game complexity. By taking (A) the size of board and (E) the tree complexity, it is possible to rank games by complexity. The correlation between artificial player strength and theoretical game complexity stands for a few games :

- Checker, log10(A) = 32, log10(E) = 17, the best artificial player plays better than the human world champion
- Othello, log10(A) = 30, log10(E) = 58, the best artificial player plays better than the human world champion
- Chess, log10(A) = 50, log10(E) = 123, the best artificial player plays is as strong as the human world champion
- Go 19, log10(A) = 160, log10(E) = 400, the best artificial player is much weaker than the human world champion

But it does not stand to a few other candidates like Go-Moku and Go 9. Go-Moku is solved, i.e. it exists a way to always win no matter what the other player plays although the game's theoretical complexity ranks between Chess and Go 19. The best artificial Go 9 players is much weaker than the world best human player although the game's theoretical complexity ranks between Othello and Chess.

The introduction concludes the classical techniques, while they give excellent results when applied to other games, fail to capture Go's complexity. Even if the subject is academically studied since the sixties, no strong artificial players have been built up the paper's publication. The question about what techniques should be used to build a strong artificial go player is still open.

Classical informed tree-search

The most commonly used techniques for artificial game players are based on tree searches. As most board games are turn-based, it is possible to capture the game current state and its history in tree structure. Nodes are board states, edges are player moves. The root node is the initial game state (the empty board in go), the leaves are states where the game is over and resulted in one's player win.

Searching in a tree structure is the operation of exploring nodes (game states) by following edges (allowed game moves) in order to find a node that meet some expectations. In the case of a game, searching means simulating the two players moves until it finds a route that always lead to the one player's victory.

Often it is not possible to perform a search deep enough to ensure a player's victory because of combinatorial explosion, e.g. the very simple game of tic-tac-toe has as many 26,830 different nodes in the fully explored tree. Artificial

players don't explore the entire tree to ensure their win, they instead search for nodes that tends toward their win, good situations to be, good nodes.

By itself, a tree traversal algorithm have no clue what good nodes are. It is up to the developer to code what is called an *evaluation function*, a function that takes a node and returns a single *score* value that describe how good the given node is according to some heuristics about the game. In games like Chess, such evaluation function look for properties like the king safety, if our king cannot escape an opponent piece nor can kill it then the evaluation function gives a disastrous score to the node.

In order to discover good nodes, it is necessary to explore enough move to find interesting nodes. The exploration is often performed using iterative-deeping with alpha-beta pruning. Iterative-deeping is an informed tree traversal algorithm, it performs a full deep-first traversal to a fixed depth and repeat the search using an increased depth if nothing was found. Alpha-beta pruning is a technique to *not explore* branches that could not give better results than the ones explored already. Together they are very effective tree search algorithm both temporal and memory wise. In Computer Go, other algorithms are often used too, like the quiescence search algorithm which keep searching as long as the board is not stable or decomposition search which split the game in many independent sub-games in order to have smaller trees and to run optimized search algorithms.

When it comes to do tree searches in the game of go, academics face a two fold problem : (1) It does not exist an evaluation function capable of capturing the game's situation in Go that is as simple and as easy to compute as in Chess. Academics do not agree about the heuristics that should be used and how to compute them efficiency. Each node takes more time to evaluate than other game. (2) Go have a much higher theoretical game complexity than Chess, for every node, there are up to 160 possible moves while there are up to about only 50 in chess. There are a magnitude of many more nodes to explore then other games for the same depth.

While tree search is the primary used technique in other games, authors of Go players say there are many more important things than tree search [1].

Evaluation function

The authors first a very simple evaluation function, this function can only determine a score for games where every intersection has a stone or is surrounded by 4 stones of same color. The evaluation function is very fast to compute a score but because it does not require any knowledge other than "what's the color of the stone at intersection (x, y)" and if there's no stone "what's the color of the neighbors of the stone at intersection (x, y)." The evaluation function is very fast but there is the hard requirement it only operates on filled boards. On average, the depth of the search tree is multiplied by a factor of 3 when you have no territory knowledge and that you fill the board. While this evaluation function is very fast to operate, because there are just too many node to explore

due to the branching factor, it is impossible with current's day computational power to apply it.

The authors then introduce a conceptual evaluation function, such function requires much more knowledge and takes much more time to compute but they show it can run at any stage of the board, not necessary leaf nodes. This evaluation functions works with a lot of concepts extracted from the human way to reason about the game, it works with concepts like territory, influence, connected groups, death/life, eyes, etc. The papers explain in a long chapter what each concept is, how to compute it and how it plays with other concepts.

Each concept uses dedicated tools to be computed. The concept of territories is mathematically known as morphological groups and techniques taken from computer vision and image processing are used so the artificial player see the territories and can gasp influence. The concept of groups and particularly how to connect them reuse tree searches but it uses many locally tree searches in order to connect different groups using strings. Other concepts like, number of eyes, interaction, life/death problems uses their own dedicated tools. The authors insists that some concepts are more important to get right than other, e.g. if the evaluation function fails to detect the incoming depth of a group it can lead to disastrous results.

Local search algorithms

While the papers shows that conducting global three searches is impossible in the game of go, it is instead possible to decompose the board in a few sub-boards around contested areas (sub-games). With the decreased search space, tree searches become possible. The difficulty is to localize the sub-games, extensive knowledge is required to understand what stones play a role, have an influence in a given problem. In some situation, even a node that seems far away at first glance may have a determining impact late in a sub problem solution. The question of which stones are near and which ones are far remain an open question.

The question about stone influence eases with the late game. In late game, the overall board may be stable enough to ensure parts of the board are fully independent from one another. In such situations, the powerful *decomposition search* algorithm may be used. The algorithm is a Conway's Combinatorial game theory application, it takes advantages of the independence and thus stability of each sub-game in order to split the global tree search in many smaller trees. Because the trees as sure to be independent and because as they are much less deep, the tree complexity dramatically decreases such that optimized full-search can be conducted to find the best solution.

Move generation

Doing tree searches using an evaluation function does not play the game by itself, it is just a technique which generates knowledge in order to select an interesting

move. What matters is the selected move, instead of trying to achieve the single goal "win the game," it is interesting to win sub-games like to occupy a big empty point, capture an opponent's territory, form an eye, cut a string, ... Each of those sub-game may be a temporary goal for the artificial player, a goal that wouldn't ensure he wins but that would better his overall situation.

Just like the conceptual evaluation function, the various goals use a lot of vocabulary that is pretty intuitive to human players but that must be explained to the machine. Because they are goals just like "win the game" but simpler, we can use fast dedicated evaluation functions and tree search to reason about goals. For goals that are very local it is also possible to recognize patterns that were pre-generated and store in a database.

When the board is stable enough, that there is no obvious moves to do. The artificial player can takes some time to reason about the various goals he knows about. He evaluate each goal on many different places of the board and to select the one that gives the better outcomes. When a goal has been selected, the evaluation function of the artificial player is temporary biased so it prefers to play moves according to the selected goal. The operation is repeated many times during the game.

Knowledge generation

The authors show that Go makes an extensive use of knowledge they explain it is interesting to automatically generate that knowledge, in other word, learn to associate patterns to moves. Various techniques with various degree of efficiency have been tried : neural network with backpropagation, temporal difference learning and retrograde analysis.

The only use of backpropagation is in the Golem artificial go player in one of the sub-goal: whether a string can be captured. When the local tree is too much complicated, it uses a neural network to discard some moves. The network was trained using games from professional players and is right 87% of the time.

Temporal difference techniques shows interesting results, the only program to efficiently uses this technique is NeuroGo, a very short C artificial player of about 5000 C lines of code. It ranks in the top 3 best artificial go player. It takes in input information string of stones like how many liberties a string have, how many stones are in the string, what's the possibility the string of being captured. It also takes information about intersections like what's the probability for the intersection to remain in our territory if we play it, what's the probability to capture the opponent, what's the probability to form an eye. The papers stays vague about how works the temporal difference algorithm and we require further researches to elaborate it.

Retrogrades analysis is a technique used to build end-game databases. It starts from endgame position that are either win, loose or draw and play in reverse to propagate this information to prior moves. It is possible to generate a database

where the ultimate outcome of many boards is known. When the artificial players reaches a board stored in the database, he knows all the possible outcomes of all possible moves from that point. The AND/OR algorithm can be performed to select a path that is guaranteed to lead to a win. Alternatively if the algorithm shows it exists a solution for the opponent to always win, the artificial player may try to mislead his opponent into making a mistake. Storing the result of a retrograde analysis demands a lot of storage, in Go a 5x3 rectangle (4% of the board) in the center has about 6.10^13 possible rules which makes this technique of lesser interest.

Simulated Annealing

The Monte Carlo method (named after the Monaco casino) is a technique that uses random samplings in order to find extrema in optimization problems that would require too much computational power to find a solution using a deterministic algorithm. The Monte Carlo method is by example used in computer physic to simulate fluid.

An Hill-climbing algorithm is an optimization algorithm that perform random small changes to a function. The function keep the new parameters every time the result is better and try new random small changes using the new parameters. On the contrary, whenever the result is worse, the new parameters are discarded. The search stops when an extremum has been found, when the computational budget is exhausted or that a result is acceptable. Because the Hill-climbing algorithm always go for the better solution, it often find a local extremum and fail to find the global extremum.

The Simulated Annealing algorithm is a kind of Monte Carlo method applied to an hill-climbing algorithm. Instead of discarding new parameters when the result is worse, there is a change the parameters are selected anyway. The probability of keeping the parameters when they are worse decrease with time. This technique greatly increases the probability to find a global extremum.

The algorithm have been applied only once in the artificial go player Gobble. The only concern of Gobble is to determine the order in which to play stones at the many intersections of the board. The only rule the player know beside the rules of the game is to not fill his eyes. In gobble, the algorithm create full game sequences, from the initial empty board to the endgame, compute the score of each player at the endgame and determine using simulated annealing the order in which to play stones. The program show good results as it is capable of competing with other artificial go players on 9x9 boards.

Personal conclusion and future work

As of 2000, the primary techniques to solve Go were heavily influenced by the human way to play the game. From a 2020 perspective, I would describe them as advanced game solvers. In my opinion they lack the necessary autonomous

learning capacities to be described as *intelligent*. My vocabulary describes this feeling, throughout this report I hardly described the various programs as *artificial intelligence* and preferred to qualify them as the more neutral *artificial players*. In the conclusion of the paper, the authors highlights the only two techniques that do not make an extensive use of tree search and evaluation functions : (i) neural network with temporal difference (reinforcement learning) and (ii) simulated annealing.

As future work it would be interesting to apply the various techniques described in the survey to other classic games. While a lot of classical 2 players boards games have been studied, my own research show a lack of studies when it comes to single players games such as the few available in the Microsoft Solitaire Collections : Klondike, Spider, Freecell, Pyramid and TriPeaks. Freecell is a particularly interesting game because just like Go and Chess, it is information completes, i.e. there are no hidden card, the player has all the information available from the start to the end of the game.

References

[1] 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.