Adding domain knowledge to a monte carlo tree search algorithm in the game of Go

Wilco Tielman
3507718

Supervisor: Jan Broersen Universiteit Utrecht Date: 30-07-2012 ECTS: 7.5

Contents

Introduction

The game of Go has long been a challenge for programmers due to the nature of the game. One of the reasons is that it's very hard to determine whether a move is good or not, which means that it's hard to write an evaluation function for moves, although successful attempts have been made [12]. Another reason that it's hard is because go has got a huge amount of possible board positions and rules that hardly limit the game. Which in all means that it has a very large game-tree complexity.

One of the more popular methods in creating a program that can play go, has been a Monte Carlo Tree search approach, which is a search tree building method often used when it's not possible to get to the solution by building a normal search tree. This approach has been shown to improve the play of go playing algorithms, as opposed to that of algorithms that only use heuristics [3]. In this Thesis I will try to answer the following question:

"Is it viable to apply domain specific knowledge to a monte-carlo tree search go playing algorithm."

To answer this I will delve into one problem that comes with a Monte Carlo Tree search approach (MCTS), which is that an MCTS algorithm also searches in a lot "stupid" tree branches of the search tree and that a normal random simulation does not give an accurate assessment on the value of a move. To counter this I will add domain knowledge to cut off stupid moves and give a more accurate simulation heuristic, while still leaving enough room for the MCTS algorithm to learn and play properly, improving its play. This will be worked into an go-playing MCTS algorithm that will serve as a proof of concept. I will also take a short look at the kinds of domain knowledge that I use for this, and what its benefits and effects are on the algorithm.

This Thesis will have the following structure: I will give a general introduction to go, and what kind of problems it brings when trying to program it. After that I will give a short introduction to what MCTS is, and why it lends itself to a problem like go. This will be followed by a look into the domain knowledge that is used in the algorithm, followed by an outline of the algorithm itself. Finally I will show the results, analyze them, try to give an answer to the question stated here above and conclude with suggestions and possible improvements to the methods I have used.

Introduction to go

Go is a very old board game originating in China around the 2nd millennium B.C. It is generally played on a board with 19x19 positions to play on. Smaller boards are also used, for quicker games, and for learning the basics.

Unlike games like chess and checkers, the positions on a go board are the intersections of the lines on the board, and not the squares themselves. The game starts with black

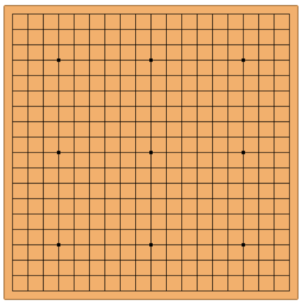

Figure 1: Example of a go board

making the first move, and after that black and white take turns until both parties have passed in succession, then the game ends. The purpose of the game is to occupy or surround as many positions as possible. Whether you want to surround as many positions, or occupy as may positions as possible, depends on if you use territory scoring, or area scoring. With territory scoring, you get points for the enemy stones you have captured, plus the amount of empty spaces you surround minus the empty areas that are "seki", which means that those places are undecided. With area scoring (which I will use in the algorithm), the score is counted by adding the stones you have on the board, plus the amount of empty points your stones surround.

You capture stones in go by removing all of its liberties using your stones. Liberties are the "life lines" of the stones/groups of stones, they are the empty spaces directly bordering a stone or a group of stones. So, a single stone placed in the center of an empty board, has got four liberties. If there are two stones in the center of the board, and both belong to the same player, they each have six liberties.

If a group loses all of its liberties, when all directly neighboring places are filled with stones, they are removed from the board. Another rule is that you are not allowed to commit suicide with a stone, so you cannot place your stone in a place where it will have no liberties, note though that capturing a stone goes before suicide. So you can place a stone where it will have no liberties at first, but due to that stone being placed, a bordering stone/group of stones gets killed and removed from the board. So it will get liberties and stay alive.

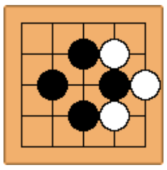

Figure 2: White to move.

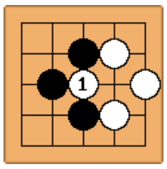

Figure 3: White captures a black stone.

You might ask yourself, after seeing the picture above, wouldn't it be simply possible to make another capture? Retaking the stone that was just placed to recapture that area. This is also not allowed, you cannot have repetition. But you are allowed to place a move somewhere else on the board, and then come back to an area like this later, and after having retaken that stone, place another one of your stones in the center, settling the Ko, as a situation like this is called. In certain rule sets, like the Chinese one, there is also a rule called superko, which means that repetition is not only prohibited from moves directly following each other, but its also not allowed to return to any previous board situation, which means that you can't get endlessly long repetitions. (otherwise the game could go on indefinitely).

In Go, there is also the possibility to give a handicap to a player (usually based on the difference in rank of the players). What the handicap entails, is that one of the players is allowed to place a number of stones already on the board, before actually starting. This is a simple, but very effective way to even the playing field, and make it easier on the less skilled player, while still giving a challenge for the more advanced player.

As I stated earlier, black starts, which gives him an initial advantage, In order to offset this advantage, the white player usually gets 6.5 points to start with (also called komi), to make it fair. This is also usually kept the same across all conventional board sizes, (from 9x9 to 19x19), due to the amount of difference a stone can make depending on the size of the board. Notice that because the komi is usually at 6.5, there is no possibility for a draw, since points are not handed out in halves. If the komi is not 6.5, but say the players decide to set it to 0, then you can get draws.

Besides the rules I have stated above, there are also some concepts that play an important role in a game of go. Note that these concepts have arisen from human players, and are used to solve/are taken into account, when making a move.

Groups, for starters, are stones that belong to one player, that are connected. So not only the stones that are connected by directly bordering each other to form a group. But the term group is also used on a set of stones that have a high likelihood of being directly connected, or form a certain set shape, that is considered connected because the opponent has no way to break a possible connection between those stones. Often in a case where they are supporting each other with either attacking or defending an area on the board.

With connected, I mean either directly bordering the stone, or is close to the stone, in such a way that they support each other, and have a large potential to be directly connected should the need arise. They are then often referred to as a single group, since they are "working" together on the same area on the board. If the stones are directly connected, they share their liberties.

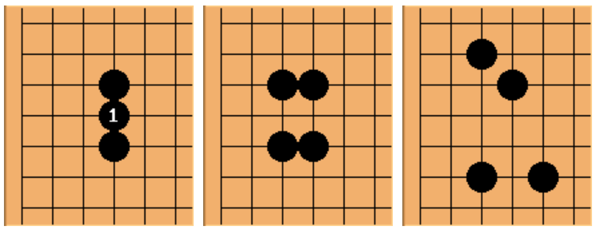

Figure 4: Three situations where the stones form a group

Territory is generally speaking the empty spaces around a players stone/group of stones, that is under their control at the end of the game. With control I mean, that the enemy player is not able to create a 'living' group within that area (the term 'living' is explained on page 6) . If there is a dispute on whether an area is under a players control at the time of scoring, it is decided based on the current rule set that is used. Rule sets are more detailed, and deal with rare situations like these.

Influence is also an important concept during a game of go. When the term influence is used in a game of go, it basically means how much influence a player has over an area on the board. For example, if the player has already placed a couple of moves in a corner, that corner "radiates" its influence to nearby areas. So if the opponent were to play close to that corner, in an area in which the player has got a lot of influence, that stone is going to have a hard time trying to stay alive. This because it doesn't have any nearby stones of his own color to connect to, to increase its chances to live, and increase territory, while this is a lot easier for the player that has got a lot of influence on that area. This also means that the stronger the group is, the stronger it “radiates” an area of influence around itself.

The concept of life and death is another important aspect in go. Life and death comes down to keeping your groups alive, and being able to kill those of your opponent. A group is deemed alive, if it has one or more liberties. A group of stones is unconditionally alive if there is no way for the opponent to kill that group, even if the player passes for the rest of the game. Life and death problems entail that the player has to estimate if a group is alive or dead. If he estimates wrongly that one of his groups is alive, he will probably waste moves on that group. If he estimates wrongly that a group is dead, it will probably end up dead. For the opponents groups the same thing holds, if the player wrongly estimates an enemy group alive, he won’t attack that group, and won’t be able to kill it.

In order to make a group unconditionally alive, it needs to have two eyes. Eyes are empty spaces surrounded by a player, where the enemy is not able to play due to the rule that you can’t commit suicide. If a group would just have one eye, it can still be captured, since capture comes before suicide. So you would be able to surround the opponents group, and then place on that one eye, capturing the group. But if there are two eyes, it is not possible since you can’t take away that groups last liberty, the eye, using the rule capture before suicide. During games between experienced go players, it doesn’t occur that often that groups become unconditionally alive, since they can see far ahead, so don’t play a sequence out on a board if they can already see that it won’t make any real difference.

Just like eyes, that can be recognized by their pattern, there are also other patterns that often occur during games, which are considered good or bad patterns by go experts. It is obvious that it is helpful that the player is able to recognize these, since the player then doesn’t have to go into a lot of detail to think about whether this is a good or bad move or that he is walking into a trap.

Introduction to Monte Carlo Tree search

In this section I will explain the basics of Monte Carlo tree search (MCTS) and some of the methods that are used to do each of the steps in an MCTS program.

MCTS is a tree search algorithm, aimed at giving an evaluation of a move (or decision, that can be represented as a tree). It does this by doing simulations of those moves, and building up the tree accordingly.

The MCTS algorithm is split up into different parts. The first one is the selection step. Here the tree is traversed down from the root until a leaf is encountered. The second step is expansion, once a leaf has been reached, it selects one of the possible moves, and creates a new leaf out of that, expanding the tree. Once the new leaf is added, it starts simulating a game until the end of that game. After that the final step comes, back propagation, the game is back propagated through all the visited nodes and leafs, which are updated based on the result of the simulated game, and the counter for how many times they have been visited is updated. This is then repeated to further build up the tree.

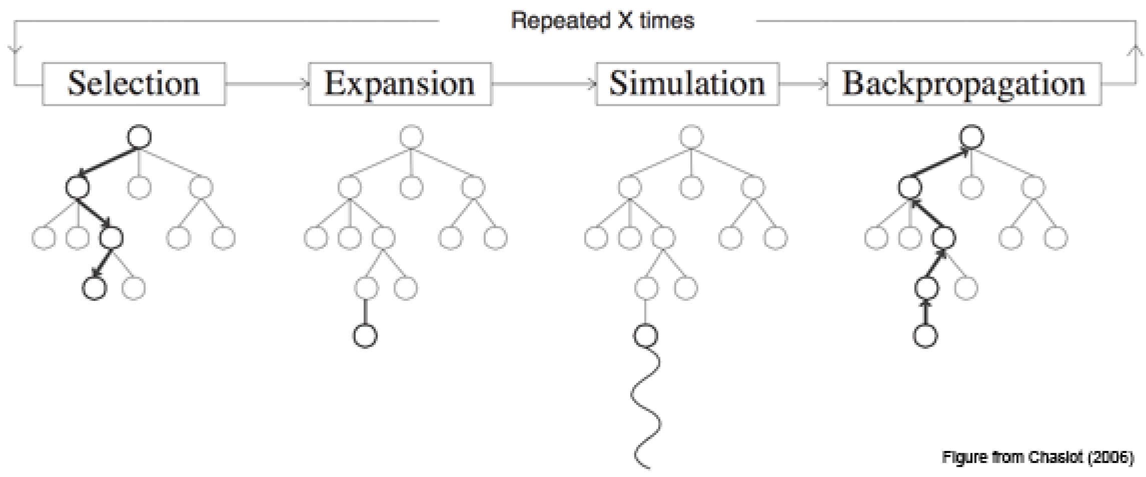

Figure 5: A visual representation of a MCTS algorithm

There are different ways to implement each step, which are continuesly being expanded opon [4]. During the selection step, the algorithm has to make the decision on which node to choose when walking down the tree. The more common and easy to implement function to make this decision is the UCT(upper confidence bounds for trees, Kocsis and Szepesv ari (2006)) .

$UCT_{value} = (v_i + C \times \sqrt{\frac{\ln n_p}{n_i}})$

Where the v_i is the value of node i, C is a constant, n_p and n_i are the times the parent node and the node i have been visited.

The goal is to balance the exploration of not yet/less visited nodes, and the exploration of already visited nodes that have a high value. You can alter this by changing the C value. This is also the method that I implemented in the algorithm, which will be discussed more in depth in 'The algorithm' section. Its important to note that when selecting a node to add to the game tree, it is picked at random from the set of possible nodes. Otherwise the algorithm will be too focused and play in one place on the board (the position you visit first).

There are also other variants, like OMC, PBBM, UCB1. Or things like [7], [10] and many others which all deal with the issue of selection. OMC and PBBM both use a two part function to determine which node to pick. Where the first part is an urgency estimate, which calculates their "urgency" value, based on the value of the nodes. After that it is passed to a fairness function, which also takes into account the amount of times a node has been visited, and selects the node which has the highest value. PBBM also takes into account the standard deviation when selecting.

The other methods (UCB1 and UCT) have these concepts worked into one formula. The UCB1, (UCB1-TUNED) also takes into account the variance in the value of the node, and its payoff.

After the selection comes the expansion. Normally this is just a case of adding 1 node, and then continuing to simulation. But there are also variants possible where multiple nodes are added each time, or to add all the nodes once it has been visited a number of times, so depending on your selection method, it limits your exploration. But the differences are minimal.

Then comes simulation. The goal of a simulation is to give an estimate of what the conclusion will be if you select that move. So it follows, that the better the simulation fits a normal play through, the better the estimate will be. With the standard MCTS approach it is usually just a random move, but this doesn't work in domains like go. There are of course a lot of different ways that you can implement this simulation heuristic. Depending on how much time is available and the value a game can add to the tree (information wise, is it a good simulation, or is it to simple/deterministic) , different simulation methods are useful. A fast method with many simulations, or a slower, but more accurate method.

The final step in the MCTS algorithm is back propagation. During the back propagation step the result of the simulated game is used to change the value of all the visited nodes. Most of the time this is done by calculating the average of the node, and updating the amount of times that node has been visited.

Domain knowledge

There is quite a lot of knowledge about go. About what kind of moves are considered good, what kind of mistakes a player can do. There are whole sequences of play that are considered good for both sides (The term for these is joseki), and within those sequences there are also possibilities to deviate from it in such a way that it still follows a set pattern, but the outcome is a bit different. Like it gains a bit more territory, or it creates a certain shape, etc.

Some of the mistakes, and “considered good moves” take on a very concrete shape. Like a certain kind of capping move, or when a player doesn’t want to sacrifice a group of stones, thus “wasting” moves on trying to save it, which would have been better spent somewhere else on the board. This all has come forth from a lot of people playing and writing about go since its invention.

The implementing of domain knowledge

The first number of computer go programs mainly used influence to decide where to play a move. After computers became more powerful, it was more readily possible to do searches, and still later, implementations with learning algorithms, and MCTS where implemented. The way in which domain knowledge has been added has taken, and will probably take, a lot of different forms. For example, Indigo, a well-known go playing program, gives an urgency value to each move, depending on a 3x3 pattern value and its current goal.

This method has been used/extended in different ways, like pre-selecting a number of possible moves, before using the urgency value to decide what to do. Or Playing within the 3x3 square from the opponents last move, which is a strategy used in MoGo (another go program).

The benefits of adding domain specific knowledge about go, to an MCTS program that plays go are obvious. Due to the nature of go, there are far to many possibilities when selecting a move to just use a brute force method. Therefore the kind of knowledge of go that player use, can very much help the level of play for an go playing algorithm. Interesting to note is that when using a MCTS approach to go, without any added benefit from domain knowledge, it will eventually be able to learn to play go well, there is a diminishing return of what it is able to learn from those random games [8]. This means that there is a diminishing return in the amount of ’insight’ the MCTS program is able to learn from random simulated games when increasing the amount of simulations.

There are many different ways to implement domain knowledge of go into an algorithm [9] [2] [5], depending on what kind of algorithm it is, as in, what kind of methods it is using. Since nearly all current go playing programs use a variety of methods to play go, the focus on each of these methods is different for each program. The most common kinds of methods that are used are the following: Tree-search, pattern-search [10], goal-oriented [11] and learning methods [2]. Each of which bring with them a different set of problems, strengths and ways to add domain specific knowledge. For example, with pattern search methods, very specific knowledge is required about those patterns. But with learning algorithms, that teach themselves patterns and their “meaning” (What kind of pattern it is, and with what purpose it will be used), less specific knowledge is needed by the programmer for the shapes of patterns, but bringing in the knowledge of an expert for information about what kind characteristics a pattern can hold, would be useful.

The main advantages of having a self learning program are big, the programmer doesn’t have to spend a lot of time thinking out heuristics for the game, doesn’t have to delve into all the knowledge that is available about the game, which usually come with exceptions to rules. But instead can set the program up in such a way that it teaches itself what is a good thing to do, and what not. But there are also down sides.
With MCTS there are quite a bit of parameters to set in order for the algorithm to function the way you want it to, and even if it is preforming well, it’s hard to get any knowledge out of the system about why it is preforming well. Whether it is because it has learned to play, quite deterministically against one player, and as such only is successful against that player, or whether it has actually learned anything meaningful about the game itself.

Using patterns to learn has also been quite successful. The go program Indigo, for example, uses patterns to play. On a 9x9 board, it scored higher using learned patterns, then patterns that had been given by experts, but on a 19x19 board it preformed significantly worse [5]. Later on Coulom (2007), further improved on this pattern learning, taking more information from the board into account, and also using patterns from professional game to learn, and play.

Also interesting to note is that combinatorial game theory has been used in go programs. It has led to a good results on certain sub problems in go, especially endgame positions [2] [1]. But has trouble with sub-problems which are not as clearly defined.

The domain knowledge in the algorithm

In this section I will go into the domain knowledge which I have worked into the simulation part of the program, which itself will be further detailed in the Algorithm section. The domain knowledge that is used, is worked in with the method that gives the algorithm a set of possible moves. With the reasoning that the knowledge is used to cut out as many useless moves as possible.

The rules used to select the moves is split up into three parts, one for the opening phase of the game, the second for the middle part of the game, and the last one for the endgame. I did it this way, to follow more closely the play style of a human player. Where at the start of the game joseki starts to play its role. The way the algorithm represents the phases of the game, is by having a phase value for each position on the board, representing in which phase it is currently in. After the opening moves, it is only

a matter of time before local fights play out, here the area of the board where the fight takes place is marked as a middle section. In an area of the board that is considered in its middle phase, life and death starts to play its role, keeping your groups alive, while keeping an open eye for any possibilities to expand influence and territory.
The further the game continues, the creation of eyes for groups starts to become more important. Once the groups and shapes on the board are settled, the endgame phase starts. Here things like taking the leftover empty spaces on the board that are not counted as anyone's territory yet are the focus.

The way these elements are represented is as follows. The values of a certain position, its phase, whether it is an eye or taken up by a stone, is stored for each position on the board. The decision making, and marking the areas around certain places (the size of the area depends on why it is marked and sometimes the size of the board) is usually done by checking the current situation for a single position on the board and its neighbours. At times groups of stones are involved, in which the algorithm does these kinds of checks for each of the stones in that group.

As I mentioned before, during the opening of the game joseki plays an important role. When viewing the games of go players, standard opening moves will be spotted. These, and their follow-ups are often part of a standard sequence of play, which fall under the term joseki. During the opening of the game, it is not smart to start a fight in one place, and then stay focused on that spot until its conclusion, since the opponent can then easily place his stones all over the rest of the board, setting up a stronger base there for expansion. Due to these things, the program limits the moves that are allowed to play first to the corners of the board (not the edges), after which it depends on the move of the opponent, if the opponent attacks (when the opponent plays very close to one of your own stones), it is given the possibility to play a defensive move (An area close to the stones in question is marked for this). The borders of the players influence are also marked as possible, and the areas close to its own stones for strengthening its position. The area's of influence are determined by marking the area's around each stone, where the closer it is to a stone, the higher the influence will be. If the areas of the two players come in contact, they either negate the other's value if their own is higher, or cancel each other out if they are the same value. This to give a similar output as the mathematical morphological operation [2].

Once a fight has broken out, when the stones of both sides are being played very close to each other and start touching, the close area around those stones will be marked as possible. There is no way implemented to deal with fights and life and death situations beyond the forcing of killing of groups that have only one liberty left, and the creation of eyes around stones that are being threatened. The algorithm determines whether a stone is being threatened by a simple calculation of its liberties, this makes sure that the program is forced to play defensive moves when it is necessary, while limiting the selection of moves as little as possible. So that the MCTS player can learn for itself in those situations. (reference to this in the conclusion, this likely does not really work efficiently, since its def possible to make an algorithm that does fights and life and death situations reasonably well, which could be used as a base for the learning, instead of hardly any base at all).

As the game continues, the groups start to settle, which means that they become unconditionally alive. Once all groups are settled, the endgame phase starts. Human players are normally able to determine when an area is settled a lot sooner, since they are able to see if an attack in an area will result in a favorable position, or if they are just wasted moves. The algorithm doesn't have a way to do this, so it plays out everything, until every group of stones is either dead and captured, or is unconditionally alive.
During the endgame phase, usually only a small number of moves are made, to settle all the leftover empty spaces, that are not yet eyes of one of both players. Once this is done as well, and both players have passed, the game ends. After which the score is calculated with territory scoring.

With the marking of the possible moves, also comes the calculation of more things beside the positions and influence. To calculate the groups and liberties, the same sort of method is used. The algorithm starts to check all of the players stones, once a stone is reached it looks around that stone to see what is on the places bordering the stones. With the creating and updating of groups, it walks through all directly neighboring stones, and marks them as being in the same group. With the calculating of liberties, the same is done, but then it also remembers and stores the amount of empty spaces nearby.

For the creation and marking of eyes, it checks those empty spaces bordering a group of stones, to see if they have started forming the shape of an eye. If so, it marks it as one, but it does not distinguish between real and false eyes. But just the marking of them, so as not to place stones there, and creating the basic shape quite fast, already is a big improvement over doing nothing.

The algorithm

In this part I will detail how the algorithm is structured, what each part does, and the design decisions made within those parts. The algorithm is built up in different parts. The main part is a game being played between the two players. With different versions for the player, one heuristic one which selects a random move between the moves that are marked with the highest priority by the heuristic part from the algorithm. The other one is an MCTS player which does not use any heuristics, but just selects random moves for the simulation. The final one is an MCTS player which uses the heuristics during its simulations.

During a game, the MCTS player builds up its search tree, by simulating x amount of games. After which it will decide on what move to do depending on the values if the UCT.

The main part of the algorithm can be summarized as follows:

```
while $game$ do
    while $PlayerOne'sTurn$ do
        $PossibleMoves \leftarrow poss(currentBoardSituation)$
        $doRandomMove(PossibleMoves)$
        $PlayerOne'sTurn \leftarrow false$
        $PlayerTwo'sTurn \leftarrow true$
    end while
    while $PlayerTwo'sTurn$ do
        $PossibleMoves \leftarrow poss(currentBoardSituation)$
        $GameTree \leftarrow simulate(PossibleMoves, currentBoardSituation)$
        $UCTvalues \leftarrow calculateUCT(GameTree, PossibleMoves)$
        $doMaxMove(UCTValues)$
        $PlayerOne'sTurn \leftarrow true$
        $PlayerTwo'sTurn \leftarrow false$
    end while
    $game \leftarrow ending(currentBoardSituation)$
end while
```

The most important part here is the simulation part. There is builds up the search tree using MCTS. The algorithm will do this for each move of player 2, so it will keep using MCTS throughout the game continuously updating the tree to the current situation. If you were to try to just run the simulation for very long for only the first move. It would take, depending on how deterministic both players play, a very long time to get any significant result due to the high branching factor in go and the duration length of a game.

The simulation part can be summarized as follows:

```
while $simulation$ do
    while $game$ do
        while $PlayerOne'sTurn$ do
            $PossibleMoves \leftarrow poss(currentBoardSituation)$
            $doRandomMove(PossibleMoves)$
            $PlayerOne'sTurn \leftarrow false$
            $PlayerTwo'sTurn \leftarrow true$
        end while
        while $PlayerTwo'sTurn$ do
            $PossibleMoves \leftarrow poss(currentBoardSituation)$
            for all $PossibleMoves$ do
                if $PossibleMoveisnotinGameTree$ then
                    $ExpandingMoves \leftarrow PossibleMove$
                end if
            end for
            if $ExpandingMoves = empty$ then
                $UCTvalues \leftarrow calculateUCT(GameTree, PossibleMoves)$
                $doMaxMove(UCTValues)$
                $BackpropagationList \leftarrow updateList(LastMove, BackpropagationList)$
            else
                if $HasDoneExpandingMove = false$ then
                    $doRandomMove(ExpandingMoves)$
                    $BackpropagationList \leftarrow updateList(LastMove, BackpropagationList)$
                    $HasDoneExpandingMove \leftarrow true$
                else
                    $doRandomMove(PossibleMoves)$
                end if
            end if
            $PlayerOne'sTurn \leftarrow true$
            $PlayerTwo'sTurn \leftarrow false$
        end while
        $game \leftarrow ending(currentBoardSituation)$
        if $game = false$ then
            $GameTree \leftarrow updateBackprop(BackpropagationList)$
        end if
    end while
    $simulation \leftarrow trainingLength(NumberOfGames)$
    if $simulation$ then
        $game \leftarrow true$
        $HasDoneExpandingMove \leftarrow false$
        $resetBoard(currentBoardSituation)$
    end if
end while
```

The program plays x number of games, and adds one node each game to the game tree. Once a game is finished, it calculates who won, and then using that and the list of visited nodes, updates them.
The domain knowledge of go is used on the selection of possible moves. The algorithm does not just select all possible moves, but using the domain knowledge, it makes a selection of moves, so that the amount of possible moves is reduced. (the picture in the MCTS section gives a likewise overview of the steps involved).

Not everything is mentioned in the pseudo code because it would have become to blurry to get a good overall view of the program. In this last part of this section I will go into the parts that have not yet been mentioned.

In this case, the MCTS player will always play as black, this due to the fact that I did not want it to get the advantage, since the program is not able to use the fact that it gets to start well, not enough to get over the 6.5 points white gets from the komi. Both players are using the same heuristics to determine the possible moves, since this is also used during the simulations the MCTS has the advantage of training against the player. If you were to play against a human opponent, it would be hard to train that amount of games against that player. Therefore the games of professionals are also used at times in other go programs, to learn from those.

Outline tests

This part of the thesis will look into the tests that were run, why they were run, and there significance in reference to the opening research question. The general goal of these tests is to serve as a proof of concept for the research question. The majority of the tests have been done on a 9x9 board, due to the amount of time it takes to play on a full 19x19 board. To start I will detail the meaning of the parameters involved.

The main parameters that are involved are the size of the board, the value that is assigned to the nodes on a win, the C value that is used in the UCT formula (which balances the exploitation of moves that have a good value, and exploration of moves that haven't been played that often), the komi value and a chance value. The chance value determines the chance that instead of creating a new node as soon as one is found, to ignore that and instead go into an already existing node to explore that more. Normally an MCTS algorithm always creates a new node when an unexplored one is encountered, so the C value handles the exploration versus exploitation between nodes that have already been visited. While the chance value, handles the exploration of new moves, or exploration of nodes that are already in the game tree. I did this because otherwise the depth of the game tree would always be quite shallow, with the addition of this chance value that can be changed.

There are three types of players involved in the tests. The first one is the normal MCTS player, which is the player that uses both MCTS and the heuristics for simulating games. The second one is a heuristic player. This player does not use MCTS or any other search method, but instead bases its moves only on the domain knowledge about go that was implemented in the algorithm (the same knowledge used for the simulations in the normal MCTS player). The third one is a MCTS player which does not use any heuristic function. So it does do MCTS simulations and learns from the games, but it does this from simulations where random legal moves are picked. The fourth and final player involved is also a MCTS player. But this player does not use any priority function in its heuristics. With the priority function I'm referring to an element in the set of possible moves the heuristics return, this priority forces the MCTS player to ignore all other normal moves, but to play the priority move (sometimes there are more), this is done in cases when either his own, or an opponent's stone/group of stones, has got only one liberty left, or in cases where a group is threatened to be captured (so before it has got only one liberty left).

The white player was always the heuristic player. I chose to use this heuristic player since I could not make my program anywhere near as good as the main go playing programs out there (they are often made by a number of people, and have been in development for years). Also, I would have had to add a way for my program to interact with the other programs if I would want to get any meaningful data.

For the first two tests, the goal was to see how large the effect is of having the selection of a set of possible moves being done using the heuristics. To do this, the first test ran a MCTS which did not use any heuristics in its simulations against a heuristic player, while the second test ran the normal MCTS player against a heuristic player. In both of those tests, the MCTS player ran 50 simulations to get a UCT value. Also note that the MCTS player that did not use any heuristics does not destroy any eyes it might form by accident.

After that the amount of simulations for the MCTS player was increased for the following three tests. Test number three ran with 500 simulations for each move, number four ran for 1000 simulations, and number five ran for 2000 simulations each move. This was done to see what for effect a larger amount of simulations would have on the results, if increasing the number dramatically would also lead to a big leap in the resulting play strength.

Finally, one more test was run of an MCTS player against a heuristic one, but the MCTS player did not use any priorities. This was done to see if the strong focus of the priorities on both capturing enemy groups that had only one liberty left and the focus on making sure that eyes were created, had any significant influence on the results.
No tests were done for a full 19x19 board due to time constraints.

Results

In this section I will go into the result of the tests that were run. Which will give insight into the initially posed research question: “Is it viable to apply domain specific knowledge to a Monte Carlo tree search go playing algorithm.”.

In the following table are the parameters for each test, with after that the results of the tests:

Values	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6
komi	6.5	6.5	6.5	6.5	6.5	6.5
v	1	1	1	1	1	1
c	5.0	5.0	5.0	5.0	5.0	5.0
chance	0.3	0.3	0.3	0.3	0.3	0.3
Simulation Number	50	50	500	1000	2000	50

In Test 1 all legal actions were marked as possible for the MCTS player, with no priority values, which means it plays a random game during the simulations instead of using domain knowledge to pre-select moves. In Test 2, 3, 4 and 5 the normal MCTS algorithm was used. In Test 6 no priorities were given to moves for the MCTS player.

Score	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6
Average score for black (MCTS)	44.63	47.78	52.27	52.09	52.80	50.52
Average score for white	40.11	37.50	32.27	32.96	32.45	34.71
Total games played	54	50	22	11	20	52

As you can see from the table above, the adding of domain knowledge to the MCTS algorithm did see an increase in its average score. A comparison between the games played with a random simulation strategy and the one that used domain knowledge showed that the random games had a lot more one sided games than the one that was using the domain knowledge. With one sided games, I mean that one side killed of as good as all the groups of the opponent, thus controlling basically the entire board. This was true for both black and white. In the case of a full victory for white, the reason was always that black was lacking eyes to protect its groups, which sometimes led to black not having any unconditionally alive groups. In the case of a full victory for black, it was because the heuristics do not recognize the difference between real and false eyes. This had as a concequence that the normal MCTS player used the heuristics to select its moves as if false eyes were a waste of a move. On the other hand, the random MCTS player, which did not have any of such notions, did play inside of those false eyes. Since white did not get any unconditionally alive groups in time, the random MCTS player was able to exploit those forms and win.

The increase in the amount of simulated games does see an increase in performance for the 50 and 500 tests. But there is not any noticeable increase in the performance once it goes up even more, to 1000 and 2000. This is likely due to fact that the games are being played on a 9x9 board and the inability of the heuristic function to see the difference between real and false eyes. Which means that both players basically give up to soon to kill an enemy group. My theory is that if the heuristics are changed in a way which does take this into account, and is complemented with more of these kinds of concepts to help its play, the MCTS player will play better when the number of simulations is increased.
Finally, test six shows that not using the priority (see 'Outline tests' for further information) to force certain moves within the heuristic function, does not affect the negatively. Test six even shows an increase in performance to that of the one that does use the priority (see test 2). This is likely because of the sometimes wrongly assigning a high priority to moves. Which shows that the balancing between the domain knowledge and searching is very important.

Future research

In the following section, with view on the results, I will mention some ways in which research could be done for further insight into the ways and usefulness for adding domain knowledge of go into an MCTS go playing algorithm.

1. Combining patterns from experts, with the learning from MCTS. Where the use of patterns and the use of MCTS is combined, to give the advantage of the local strength of patterns, and the stronger global play of an MCTS player outside of those patterns.

2. Looking at the effects on the building of the tree, specifically the determination of the value of the nodes. Like combining the score of the simulated game with MCTS, with a heuristic evaluation of a move. As in, use an heuristic evaluation function for giving insight into what are considered good moves and then combine those results with the values gotten from the MCTS algorithm.

3. To look at the benefit of the types of knowledge added to a MCTS algorithm. Whether it is more important to have the knowledge of forming and handling eyes is more important than a good opening strategy that uses joseki. Or whether the ability to handle life and death situations with a local MCTS algorithm preforms better or worse than an MCTS algorithm that uses patterns to get the value of the moves.

Conclusion

Due to the nature of go, the game has been and is a difficult problem for programmers who try to create a go playing programm. Not only becuse of the large board and the large amount of possible moves each turn, but also because of the many concepts that are involved in a game of go. Seeing the rise of computers playing games like chess on a growing skill level, go has also gotten a rising interest bt programmers. A method which is often used is Monte Carlo tree search (MCTS), which is a tree searching algorithm, that teaches itself the value of a given move using selection, expansion, simulation and backpropagation as its core elements. This thesis explored this area, trying to find an answer to the question if it is viable to apply domain specific knowledge to a Monte Carlo tree search go playing algorithm.

The results of the tests that were run, show that adding domain knowledge to a go playing Monte Carlo tree search algoritm is definatly viable. The combination of domain knowledge to help guide the learning element in MCTS, as a way of evaluating the moves that are possible, clearly has got its benifits. Although, depending on the algorithm, it is important to note that increasing the amount of simulations does not automaticly mean a better score. So the balance between the variables involved in the MCTS algorithm, and the knowledge of go added into the algorithm is needed when looking for the best results.

Combining the knowledge of the advantages and the problems that come with adding domain knowledge and MCTS, more ways come into view which can be implemented to help and give insight into the limits and possibilities of MCTS go. Like adding the ability to learn if the present domain knowledge is actually beneficial, or can be discarded, or to learn whether the current parameters are good or not within the MCTS algorithm, or other possibilities which are discussed in the 'Future research' section.

References

[1] E. Berlekamp, *Introductory overview of mathematical Go endgames*, in: Proceedings of Symposia in Applied Mathematics, 43, 1991.

[2] Bouzy, B. and Cazenave, T. (2001). *Computer Go: An AI Oriented Survey.* Artificial Intelligence, Vol. 132, No. 1, pp. 39–103.

[3] Bruno Bouzy, *Associating domain-dependent knowledge and Monte Carlo approaches within a Go program.* Information Sciences, Volume 175, Issue 4, 15 November 2005, Pages 247-257.

[4] Chaslot, G.M.J-B., Saito, J-T., Bouzy, B., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2006a). *Monte-Carlo Strategies for Computer Go.* Proceedings of the 18th BeNeLux Conference on Artificial Intelligence (eds. P.-Y. Schobbens, W. Vanhoof, and G. Schwanen), pp. 83–90.

[5] Guillaume Maurice Jean-Bernard Chaslot, *Monte-Carlo Tree Search*, PhD thesis, University Maastricht, 2010.

[6] Coulom, R. *Computing "Elo Ratings" of Move Patterns in the Game of Go*, ICGA Journal, Vol. 30, No. 4, pp. 199–208, 2007.

[7] David Tolpin, Solomon Eyal Shimony, *Doing Better Than UCT: Rational Monte Carlo Sampling in Trees.* CoRR, abs/1108.3711, 2011.

[8] Haruhiro Yoshimoto, Kazuki Yoshizoe, Tomoyuki Kaneko, Akihiro Kishimoto, Kenjiro Taura, *Monte Carlo go has a way to go*, AAAI'06 proceedings of the 21st national conference on Artificial intelligence - Volume 2, Pages 1070-1075

[9] Peter Drake and Steve Uurtamo. *Move ordering vs heavy playouts: Where should heuristics be applied in Monte Carlo Go?* In Proceedings of the 3rd North American Game-On Conference, 2007.

[10] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. *Modification of UCT with Patterns in Monte-Carlo Go.* Technical report, Inst. Nat. Rech. Inform. Auto. (INRIA), Paris, 2006.

[11] Tristan Cazenave, Bernard Helmstetter, *Combining tactical search and monte-carlo in the game of go*, IN: CIG'05, 2005, pages 171–175.

[12] Wolf T, *A Dynamical Systems Approach for Static Evaluation in Go.* IEEE Transactions on Computational Intelligence and AI in Games, June 2011, Pages 129 - 141.