Summary

This thesis studies the use of Monte-Carlo simulations for tree-search problems. The Monte-Carlo technique we investigate is Monte-Carlo Tree Search (MCTS). It is a best-first search method that does not require a positional evaluation function in contrast to $\alpha\beta$ search. MCTS is based on a randomized exploration of the search space. Using the results of previous explorations, MCTS gradually builds a game tree in memory, and successively becomes better at accurately estimating the values of the most promising moves. MCTS is a general algorithm and can be applied to many problems. The most promising results so far have been obtained in the game of Go, in which it outperformed all classic techniques. Therefore Go is used as the main test domain.

Chapter 1 provides a description of the search problems that we aim to address and the classic search techniques which are used so far to solve them. The following problem statement guides our research.

> **Problem statement**: *How can we enhance Monte-Carlo Tree Search in such a way that programs improve their performance in a given domain?*

To answer the problem statement we have formulated five research questions. They deal with (1) Monte-Carlo simulations, (2) the balance between exploration and exploitation, (3) parameter optimization, (4) parallelization, and (5) opening-book generation.

Chapter 2 describes the test environment to answer the problem statement and the five research questions. It explains the game of Go, which is used as the test domain in this thesis. The chapter provides the history of Go, the rules of the game, a variety of game characteristics, basic concepts used by humans to understand the game of Go, and a review of the role of Go in the AI domain. The Go programs MANGO and MOGO, used for the experiments in the thesis, are briefly described.

Chapter 3 starts with discussing earlier research about using Monte-Carlo evaluations as an alternative for a positional evaluation function. This approach is hardly used anymore, but it established an important step towards MCTS. Subsequently, a general framework for MCTS is presented in the chapter. MCTS consists of four main steps: (1) In the *selection step* the tree is traversed from the root node until we reach a node, where we select a child that is not part of the tree yet. (2) Next, in the *expansion step* a node is added to the tree. (3) Subsequently, during the *simulation step* moves are played in self-play until the end of the

game is reached. (4) Finally, in the *backpropagation step*, the result of a simulated game is propagated backwards, through the previously traversed nodes.

Each step has a strategy associated that implements a specific policy. Regarding selection, the UCT strategy is used in many programs as a specific selection strategy because it is simple to implement and effective. A standard selection strategy such as UCT does not take domain knowledge into account, which could improve an MCTS program even further. Next, a simple and efficient strategy to expand the tree is creating one node per simulation. Subsequently, we point out that building a simulation strategy is probably the most difficult part of MCTS. For a simulation strategy, two balances have to be found: (1) between search and knowledge, and (2) between exploration and exploitation. Furthermore, evaluating the quality of a simulation strategy has to be assessed together with the MCTS program using it. The best simulation strategy without MCTS is not always the best one when using MCTS. The backpropagation strategy that is the most successful is taking the average of the results of all simulated games made through a node.

Finally, we give applications of MCTS to different domains such as Production Management Problems, Library Performance Tuning, SameGame, Morpion Solitaire, Sailing Domain, Amazons, Lines of Action, Chinese Checkers, Settlers of Catan, General Game Playing, and in particular Go.

The most basic Monte-Carlo simulations consist of playing random moves. *Knowledge* transforms the plain random simulations into more sophisticated *pseudo-random* simulations. This has led us to the first research question.

> **Research question 1**: *How can we use knowledge to improve the Monte-Carlo simulations in MCTS?*

Chapter 4 answers the first research question. We explain two different simulation strategies that apply knowledge: urgency-based and sequence-like simulation. Based on the experience gathered from implementing them in INDIGO and MOGO, respectively, we make the following three recommendations. (1) Avoiding big mistakes is more important than playing good moves. (2) Simulation strategies using sequence-like simulations or patterns in urgency-based simulations are efficient because they simplify the situation. (3) The simulation strategy should not become too stochastic, nor too deterministic, thus balancing exploration and exploitation.

Moreover, we develop the first efficient method for learning automatically the knowledge of the simulation strategy. We proposed to use *move evaluations* as a fitness function instead of learning from the results of simulated games. A coefficient is introduced that enables to balance the amount of exploration and exploitation. The algorithm is adapted from the tracking algorithm of Sutton and Barto. Learning is performed for 9×9 Go, where we showed that the Go program INDIGO with the learnt patterns performed better than the program with expert patterns.

In MCTS, the selection strategy controls the balance between exploration and exploitation. The selection strategy should favour the most promising moves (exploitation). However, less promising moves should still be investigated sufficiently (exploration), because their low scores might be due to unlucky simulations. This move-selection task can be facilitated by applying knowledge. This idea has guided us to the second research question.

> **Research question 2**: *How can we use knowledge to arrive at a proper balance between exploration and exploitation in the selection step of MCTS?*

Chapter 5 answers the second research question by proposing two methods that integrate knowledge into the selection step of MCTS: progressive bias and progressive widening. Progressive bias uses knowledge to direct the search. Progressive widening first reduces the branching factor, and then increases it gradually. We refer to them as "progressive strategies" because the knowledge is dominant when the number of simulations is small in a node, but loses influence progressively when the number of simulations increases.

First, the progressive strategies are tested in MANGO. The incorporated knowledge is based on urgency-based simulation. From the experiments with MANGO, we observe the following. (1) Progressive strategies, which focus initially on a small number of moves, are better in handling large branching factors. They increase the level of play of the program MANGO significantly, for every board size. (2) On the 19×19 board, the combination of both strategies is much stronger than each strategy applied separately. The fact that progressive bias and progressive widening work better in combination with each other shows that they have complementary roles in MCTS. This is especially the case when the board size and therefore branching factor grows. (3) Progressive strategies can use relatively expensive domain knowledge with hardly any speed reduction.

Next, the performance of the progressive strategies in other game programs and domains is presented. Progressive bias increases the playing strength of MOGO and of the Lines-of-Action program MC-LOA, while progressive widening did the same for the Go program CRAZY STONE. In the case of MOGO, progressive bias is successfully combined with RAVE, a similar technique for improving the balance between exploitation and exploration. These results give rise to the main conclusion that the proposed progressive strategies are essential enhancements for an MCTS program.

MCTS is controlled by several parameters, which define the behaviour of the search. Especially the selection and simulation strategies contain several important parameters. These parameters have to be optimized in order to get the best performance out of an MCTS program. This challenge has led us to the third research question.

> **Research question 3**: *How can we optimize the parameters of an MCTS program?*

Chapter 6 answers the third research question by proposing to optimize the search parameters of MCTS by using an evolutionary strategy: the Cross-Entropy Method (CEM). CEM is related to Estimation-of-Distribution Algorithms (EDAs), a new area of evolutionary computation. The fitness function for CEM measures the winning rate for a batch of games. The performance of CEM with a fixed and variable batch size is tested by tuning 11 parameters in MANGO. Experiments reveal that using a batch size of 500 games gives the best results, although the convergence is slow. A small (and fast) batch size of 10 still gives a reasonable result when compared to the best one. A variable batch size performs a little bit worse than a fixed batch size of 50 or 500. However, the variable batch size converges faster than a fixed batch size of 50 or 500.

Subsequently, we show that MANGO with the CEM parameters performs better against GNU GO than the MANGO version without. In four self-play experiments with different time settings and board sizes, the CEM version of MANGO defeats the default version

convincingly each time. Based on these results, we may conclude that a hand-tuned MCTS-using game engine may improve its playing strength when re-tuning the parameters with CEM.

The recent evolution of hardware has gone into the direction that nowadays personal computers contain several cores. To get the most out of the available hardware one has to parallelize MCTS as well. This has led us to the fourth research question.

Research question 4: *How can we parallelize MCTS?*

Chapter 7 answers the fourth research question by investigating three methods for parallelizing MCTS: leaf parallelization, root parallelization and tree parallelization. Leaf parallelization plays for each available thread a simulated game starting from the leaf node. Root parallelization consists of building multiple MCTS trees in parallel, with one thread per tree. Tree parallelization uses one shared tree from which games simultaneously are played.

Experiments are performed to assess the performance of the parallelization methods in the Go program MANGO on the 13×13 board. In order to evaluate the experiments, we propose the strength-speedup measure, which corresponds to the time needed to achieve the same strength. Experimental results indicate that leaf parallelization is the weakest parallelization method. The method leads to a strength speedup of 2.4 for 16 processor threads. The simple root parallelization turns out to be the best way for parallelizing MCTS. The method leads to a strength speedup of 14.9 for 16 processor threads. Tree parallelization requires two techniques to be effective. First, using local mutexes instead of a global mutex doubles the number of games played per second. Second, the virtual-loss enhancement increases both the games-per-second and the strength of the program significantly. By using these two techniques, we obtain a strength speedup of 8.5 for 16 processor threads.

Modern game-playing programs use opening books in the beginning of the game to save time and to play stronger. Generating opening books in combination with an $\alpha\beta$ program has been well studied in the past. The challenge of generating automatically an opening book for MCTS programs has led to the fifth research question.

Research question 5: *How can we automatically generate opening books by using MCTS?*

Chapter 8 answers the fifth research question by combining two levels of MCTS. The method is called Meta Monte-Carlo Tree Search (Meta-MCTS). Instead of using a relatively simple simulation strategy, it uses an entire MCTS program (MOGO) to play a simulated game. We describe two algorithms for Meta-MCTS: Quasi Best-First (QBF) and Beta-Distribution Sampling (BDS). The first algorithm, QBF, is an adaptation of greedy algorithms that are used for the regular MCTS. QBF favours therefore exploitation. During actual game play we observe that despite the good performance of the opening book, some branches are not explored sufficiently. The second algorithm, BDS, favours exploration. In contrast to UCT, BDS does not need an exploration coefficient to be tuned. The algorithm draws a move according to its likelihood of being the best move (considering the number of wins and losses). This approach created an opening book which is shallower and wider. The BDS book has the drawback to be less deep against computers, but the advantage is

that it stayed longer in the book in official games against humans. Experiments on the Go server CGOS reveal that both QBF and BDS were able to improve MOGO. In both cases the improvement is more or less similar. Based on the results, we may conclude that QBF and BDS are able to generate an opening book which improves the performance of an MCTS program.

The last chapter of the thesis returns to the five research questions and the problem statement as formulated in Chapter 1. Taking the answers to the research questions above into account we see that there are five successful ways to improve MCTS. First, learning from move evaluations improves the knowledge of the simulation strategy. Second, progressive strategies enhance the selection strategy by incorporating knowledge. Third, CEM optimizes the search parameters of an MCTS program in such a way that its playing strength is increased. Fourth, MCTS benefits substantially from parallelization. Fifth, Meta-MCTS generates an opening book that improves the performance of an MCTS program. Yet, we are able to provide additional promising directions for future research. Finally, the question of understanding the nature of MCTS is still open.

Samenvatting

Dit proefschrift bestudeert het gebruik van Monte-Carlo simulaties voor zoekproblemen. De Monte-Carlo techniek die wij onderzoeken is Monte-Carlo Tree Search (MCTS). Het is een *best-first* zoekmethode die in tegenstelling tot het $\alpha\beta$ zoekalgoritme geen positionele evaluatiefunctie vereist. MCTS is gebaseerd op een willekeurige verkenning van de zoekruimte. Met behulp van de resultaten van eerdere verkenningen, bouwt MCTS geleidelijk een spelboom op in het computergeheugen en gaat dan de waarden van de veelbelovende zetten steeds beter schatten. MCTS is een generiek algoritme en kan worden toegepast op veel problemen. De meest veelbelovende resultaten zijn tot dusver verkregen in het spel Go, waarin MCTS beter presteert dan de klassieke technieken. Go wordt daarom gebruikt als testdomein in dit proefschrift.

Hoofdstuk 1 geeft een beschrijving van de zoekproblemen die we beogen aan te pakken en de klassieke zoektechnieken die tot dusver zijn gebruikt om ze op te lossen. De volgende probleemstelling is geformuleerd.

> **Probleemstelling** : *Hoe kunnen we Monte-Carlo Tree Search op zo'n manier verder ontwikkelen dat programma's hun prestaties in een gegeven domein verbeteren?*

Voor de beantwoording van de probleemstelling hebben we vijf onderzoeksvragen geformuleerd. Ze gaan over (1) Monte-Carlo simulaties, (2) de balans tussen exploratie en exploitatie, (3) parameter optimalisatie, (4) parallellisatie, en (5) openingsboek generatie.

Hoofdstuk 2 beschrijft de testomgeving die gebruikt wordt om de probleemstelling en de vijf onderzoeksvragen te beantwoorden. Het geeft een uitleg van het spel Go, dat als testdomein in dit proefschrift wordt gebruikt. Het hoofdstuk geeft de geschiedenis van Go, de regels, verscheidene spelkarakteristieken, enkele basisprincipes, en een beschouwing over de rol van Go in het AI-domein. De Go programma's MANGO en MOGO, die worden gebruikt voor de experimenten, worden kort beschreven.

Hoofdstuk 3 begint met een bespreking van eerder onderzoek over het gebruik van Monte-Carlo evaluaties als een alternatief voor een positionele evaluatiefunctie. Deze aanpak wordt nauwelijks meer gebruikt, maar is een belangrijke stap geweest op weg naar MCTS. Hierna wordt in het hoofdstuk een algemeen raamwerk voor MCTS gepresenteerd. MCTS bestaat uit vier hoofdstappen: (1) In de *selectie stap* wordt de boom vanaf de wortel doorkruist totdat we arriveren in een knoop waar een kind geselecteerd wordt dat nog geen

onderdeel is van de zoekboom. (2) Daarna wordt er in de *expansie stap* een knoop toegevoegd aan de boom. (3) Vervolgens, wordt er gedurende de *simulatie stap* een gesimuleerde partij gespeeld. (4) In de *terugpropagatie stap* wordt dan het resultaat van die gesimuleerde partij verwerkt in de knopen langs het afgelegde pad.

Aan elke MCTS stap is een strategie verbonden dat een specifiek beleid uitvoert. Voor selectie wordt in veel programma's de UCT-strategie gebruikt omdat ze eenvoudig uit te voeren en effectief is. Een standaard selectie strategie, zoals UCT, gebruikt geen domeinkennis. Een eenvoudige en efficiënte strategie voor het expanderen van de boom is om de eerste positie die we tegenkomen in de gesimuleerde partij toe te voegen. Vervolgens wijzen wij erop dat het creeëren van een simulatie strategie waarschijnlijk het moeilijkste onderdeel is in MCTS. Voor een simulatie strategie moeten er twee balansen worden gevonden: (1) tussen zoeken en kennis, en (2) tussen exploratie en exploitatie. Bovendien moet de kwaliteit van een simulatie strategie altijd samen worden geëvalueerd met het MCTS programma waarin het wordt gebruikt. De beste simulatie strategie zonder MCTS is niet altijd de beste met MCTS. De terugpropagatie strategie, die het meest succesvol is, neemt gewoon het gemiddelde over de resultaten van alle gesimuleerde partijen in de desbetreffende knoop.

Tenslotte geven we enige toepassingen van MCTS in verschillende domeinen zoals Productie Management Problemen, Libary Performance Tuning, SameGame, Morpion Solitaire, Sailing Domain, Amazons, Lines of Action, Chinese Checkers, Kolonisten van Catan, General Game Playing, en in het bijzonder Go.

De meest basale Monte-Carlo simulaties bestaan uit het willekeurig spelen van zetten. Het gebruik van kennis kan deze simulaties in meer verfijnd spel transformeren. Dit heeft ons tot de eerste onderzoeksvraag gebracht.

> **Onderzoeksvraag 1**: *Hoe kunnen we kennis gebruiken om Monte-Carlo simulaties in MCTS te verbeteren?*

Hoofdstuk 4 geeft antwoord op de eerste onderzoeksvraag. We leggen twee verschillende simulatie strategieën uit die kennis toepassen: urgentie-gebaseerde en sequentie-gebaseerde simulaties. Op basis van de opgedane ervaringen in INDIGO en MOGO, doen we de volgende drie aanbevelingen. (1) Het vermijden van grote fouten is belangrijker dan het doen van goede zetten. (2) Simulatie strategieën zijn efficiënt als ze de situatie vereenvoudigen zoals in urgentie-gebaseerde of sequentie-gebaseerde simulaties. (3) De simulatie strategie moet niet te stochastisch noch te deterministisch zijn; dus de strategie moet balanceren tussen exploratie en exploitatie.

Verder ontwikkelen we de eerste efficiënte methode voor het automatisch leren van de kennis gebruikt in de simulatie strategie. Wij hebben voorgesteld om zetevaluaties te gebruiken als een fitheidsfunctie in plaats van leren op basis van de resultaten van gesimuleerde spelen. Een coëfficiënt wordt geïntroduceerd die het mogelijk maakt exploratie en exploitatie te balanceren. Het leeralgoritme is een aanpassing van het *tracking* algoritme van Sutton en Barto. De experimenten zijn uitgevoerd voor 9×9 Go, waar we laten zien dat het Go programma INDIGO met de geleerde patronen beter presteert dan het programma gebaseerd op expert patronen.

In MCTS regelt de selectie strategie de balans tussen exploratie en exploitatie. Aan de ene kant moet de selectie strategie zich richten op de veelbelovende zetten (exploitatie).

Aan de andere kant moeten de minder rooskleurige zetten nog steeds voldoende worden onderzocht (exploratie). Deze taak kan worden gefaciliteerd door kennis. Dit idee heeft ons tot de tweede onderzoeksvraag gebracht.

> **Onderzoeksvraag 2**: *Hoe kunnen we gebruik maken van kennis om te komen tot een goede balans tussen exploratie en exploitatie in de selectie stap van MCTS?*

Hoofdstuk 5 geeft antwoord op de tweede onderzoeksvraag door twee technieken voor te stellen die kennis integreren in de selectie stap: *progressive bias* en *progressive widening*. Progressive bias gebruikt kennis om het zoekproces bij te sturen. Op basis van kennis reduceert progressive widening eerst de vertakkingsgraad, om deze vervolgens geleidelijk weer te vergroten. We verwijzen naar deze twee technieken als "progressieve strategieën", omdat de kennis dominant is als het aantal simulaties klein is in een knoop, maar geleidelijk aan invloed verliest als het aantal simulaties toeneemt.

Eerst worden de progressieve strategieën getest in MANGO. De ingebouwde kennis is gebaseerd op de urgentie-gebaseerde simulatie. Op grond van de experimenten met MANGO observeren we het volgende. (1) Progressieve strategieën, die in zich in eerste instantie richten op een klein aantal zetten, zijn beter in het verwerken van een grote vertakkingsgraad. Ze vergroten het spelniveau van het programma MANGO aanzienlijk, voor elke bord grootte. (2) Op het 19×19 bord is de combinatie van beide strategieën veel sterker dan elke strategie afzonderlijk. Het feit dat progressive bias en progressive widening beter werken in combinatie met elkaar laat zien dat ze elkaar aanvullen in MCTS. Dit is vooral het geval wanneer de bord grootte en derhalve de vertakkingsgraad groeit. (3) Progressieve strategieën kunnen gebruik maken van relatief dure domeinkennis bijna zonder de snelheid te verlagen.

Vervolgens worden de prestaties van de progressieve strategieën in andere spelprogramma's en domeinen gepresenteerd. Progressive bias verhoogt de speelsterkte van MOGO en van het Lines-of-Action programma MC-LOA, terwijl progressive widening het Go programma CRAZY STONE verbetert. In het geval van MOGO is progressive bias succesvol gecombineerd met RAVE, een vergelijkbare techniek voor de verbetering van de balans tussen exploitatie en exploratie. Deze resultaten geven aanleiding tot de belangrijkste conclusie dat de voorgestelde progressieve strategieën essentiële verbeteringen zijn voor een MCTS programma.

MCTS wordt gecontroleerd door een aantal parameters, die het zoekgedrag bepalen. Vooral de selectie en simulatie strategieën bevatten een aantal belangrijke parameters. Deze parameters moeten worden geoptimaliseerd om de beste prestaties te krijgen voor een MCTS programma. Deze uitdaging heeft ons gebracht tot de derde onderzoeksvraag.

> **Onderzoeksvraag 3**: *Hoe kunnen we de parameters van een MCTS programma optimaliseren?*

Hoofdstuk 6 geeft antwoord op de derde onderzoeksvraag door voor te stellen om de MCTS zoekparameters te optimaliseren met behulp van een evolutionaire strategie: de *Cross-Entropy Method* (CEM). CEM is gerelateerd aan *Estimation-of-Distribution Algorithms* (EDAs). De fitheidsfunctie voor CEM meet het winspercentage voor een bepaald

aantal (*batch*) partijen. De prestaties van CEM met een vaste batch en een variabele batch worden getest door 11 parameters af te stellen in MANGO. Experimenten tonen aan dat het gebruik van een batch grootte van 500 partijen de beste resultaten geeft, hoewel de convergentie traag is. Een kleine (en snelle) batch grootte van 10 partijen geeft nog steeds een redelijk resultaat in vergelijking met de beste batch grootte. Een variabele grootte presteert iets minder dan een vaste grootte van 50 of 500 partijen. Echter, de variabele batch convergeert sneller dan een vaste batch grootte van 50 of 500 partijen.

Vervolgens laten we zien dat MANGO met CEM parameters beter presteert tegen GNU GO dan de MANGO versie met de oude parameters. In vier zelfspel experimenten met verschillende tijdsinstellingen en bord groottes verslaat de CEM versie van MANGO de standaardversie elke keer overtuigend. Gebaseerd op deze resultaten kunnen we concluderen dat een met de hand afgesteld MCTS programma zijn spelsterkte kan verbeteren door CEM toe te passen.

De recente evolutie van hardware is gegaan in de richting dat tegenwoordig PC's meerdere processorkernen bevatten. Om het maximale uit de beschikbare hardware te halen moet men MCTS parallelliseren. Dit heeft geleid tot de vierde onderzoeksvraag.

Onderzoeksvraag 4: *Hoe kunnen we MCTS parallelliseren?*

Hoofdstuk 7 geeft antwoord op de vierde onderzoeksvraag door het onderzoeken van drie methoden voor de parallellisatie van MCTS: blad-, wortel-, en boomparallellisatie. Bladparallellisatie simuleert voor elke beschikbare processorkern een partij, startend in hetzelfde blad. Wortelparallellisatie bestaat uit het construeren van meerdere MCTS bomen, waarvoor geldt dat elke zoekboom zijn eigen processorkern heeft. Boomparallellisatie maakt gebruik van een gedeelde zoekboom waarin gelijktijdig meerdere simulaties worden gespeeld.

Experimenten worden uitgevoerd om de prestaties van de parallellisatie methoden te beoordelen voor het Go programma MANGO op het 13×13 bord. Om de experimenten te evalueren, introduceren wij de *sterkte-versnelling* maat, die overeenkomt met de hoeveelheid denktijd die nodig is om dezelfde sterkte te bereiken. De experimentele resultaten wijzen erop dat bladparallellisatie de zwakste parallellisatie methode is. De methode leidt tot een sterkte-versnelling van 2,4 voor 16 processorkernen. De eenvoudige wortelparallellisatie blijkt de beste manier om MCTS te parallelliseren. De methode leidt tot een sterkte-versnelling van 14,9 voor 16 processorkernen. Boomparallellisatie vereist twee technieken om effectief te zijn. Ten eerste, het gebruik van lokale *mutexen* in plaats van één globale mutex verdubbelt het aantal gespeelde simulaties per seconde. Ten tweede, de *virtueel-verlies* techniek verhoogt zowel het aantal simulaties als de kracht van het programma aanzienlijk. Door het gebruik van deze twee technieken krijgen we een sterkte-versnelling van 8,5 voor 16 processorkernen.

Moderne spelprogramma's gebruiken een openingsboek om in het begin van het spel tijd uit te sparen en sterker te spelen. Het genereren van een openingsboek voor een $\alpha\beta$ programma is goed bestudeerd. De uitdaging om een openingsboek automatisch te genereren voor een MCTS programma heeft geleid tot de vijfde onderzoeksvraag.

Onderzoeksvraag 5: *Hoe kunnen we automatisch een openingsboek genereren door gebruik te maken van MCTS?*

Hoofdstuk 8 beantwoordt de vijfde onderzoeksvraag door twee niveaus van MCTS te combineren. De methode heet Meta Monte-Carlo Tree Search (Meta-MCTS). In plaats van een relatief eenvoudige simulatie strategie te gebruiken, wordt een volledig MCTS programma (MoGo) gebruikt om een simulatie uit te voeren. We beschrijven twee algoritmen voor Meta-MCTS: *Quasi Best-First* (QBF) en *Beta-Distribution Sampling* (BDS). Het eerste algoritme, QBF, is een aanpassing van *greedy* algoritmen die worden gebruikt voor de reguliere MCTS. QBF bevordert exploitatie. Voor toernooipartijen constateren we dat ondanks de goede prestaties van het openingsboek, sommige openingen niet voldoende zijn onderzocht. Het tweede algoritme, BDS, bevordert exploratie. In tegenstelling tot de selectie strategie UCT, heeft BDS geen exploratie constante die moet worden afgesteld. Het algoritme trekt een zet op basis van de waarschijnlijkheid dat het de beste zet is (rekening houdend met het aantal overwinningen en nederlagen). Deze benadering maakt het openingsboek minder diep maar breder. Het BDS boek heeft als nadeel dat men niet zo lang in het boek blijft tegen computer programma's, maar het voordeel dat men langer in het boek blijk in officiële wedstrijden tegen mensen. Experimenten op de Go server CGOS onthullen dat zowel QBF als BDS het programma MoGo verbeteren. In beide gevallen is de verbetering min of meer vergelijkbaar. Gebaseerd op de resultaten kunnen we concluderen dat QBF en BDS in staat zijn om een openingsboek te genereren dat de prestaties van een MCTS programma verbetert.

In het laatste hoofdstuk keren we terug naar de vijf onderzoeksvragen en de probleemstelling zoals die in hoofdstuk 1 zijn geformuleerd. Rekening houdend met de hierboven gegeven antwoorden op de onderzoeksvragen zien we dat er vijf succesvolle manieren zijn om de prestaties van een MCTS programma te verbeteren. (1) Het leren door middel van zetevaluaties verbetert de kennis van de simulatie strategie. (2) Progressieve strategieën versterken de selectie strategie door kennis te integreren. (3) CEM optimaliseert de zoekparameters van een MCTS programma op een zodanige wijze dat de speelsterkte toeneemt. (4) MCTS profiteert aanzienlijk van parallellisatie. (5) Meta-MCTS genereert een openingsboek dat de prestaties van een MCTS programma verbetert. Hierna geven we veelbelovende richtingen van vervolgonderzoek aan. Tenslotte, de vraag van het begrijpen van de aard van MCTS is nog open.