Monte-Carlo Tree Search in Production Management Problems

Guillaume Chaslot Steven de Jong Jahn-Takeshi Saito
Jos Uiterwijk

Department of Computer Science, Universiteit Maastricht, The Netherlands
Email: {g.chaslot, steven.dejong, j.saito, uiterwijk}@cs.unimaas.nl

Abstract

Classical search algorithms rely on the existence of a sufficiently powerful evaluation function for non-terminal states. In many task domains, the development of such an evaluation function requires substantial effort and domain knowledge, or is not even possible. As an alternative in recent years, Monte-Carlo evaluation has been succesfully applied in such task domains.

In this paper, we apply a search algorithm based on Monte-Carlo evaluation, *Monte-Carlo Tree Search*, in the task domain of production management problems. These can be defined as single-agent problems which consist of selecting a sequence of actions with side effects, leading to high quantities of one or more goal products. They are challenging and can be constructed with highly variable difficulty. Earlier research yielded an offline learning algorithm that leads to good solutions, but requires a long time to run. We show that Monte-Carlo Tree Search leads to a solution in a shorter period of time than this algorithm, with improved solutions for large problems. Our findings can be generalized to other task domains.

1 Introduction

Search and evaluation. Traditional tree-search algorithms such as $\alpha\beta$-Search and A* Search have been successfully applied in many task domains, for instance board games such as Checkers and Chess (e.g., [1, 2, 3]). Starting from the current state of such a game, a tree is developed that reflects the various possible moves. If we were able to expand this tree up to the level of final states, an optimal strategy could be found for any state using Minimax backpropagation. However, almost all games require a game tree that cannot be completely constructed. Therefore, search techniques rely heavily on the existence of an *evaluation function*: Instead of expanding the game tree until the end of the game is reached, we expand it just a few (e.g., 5 to 10) moves ahead and then estimate the value of the (non-final) states obtained using an evaluation function which is usually heuristic [4, 5]. Research in the combinatorial games domain has shown that obtaining a sufficiently accurate evaluation function can be difficult (if not impossible) and/or requires a substantial amount of parameter tuning [6].

Monte-Carlo techniques. To address the problems outlined above, Monte-Carlo sampling is used in many combinatorial games as an evaluation function [7, 8, 9, 10]. The idea of Monte-Carlo sampling is basically to perform a large number of (randomized) simulations, and then analyse the results of these simulations statistically in order to approximate some outcome which cannot be computed exactly. Monte-Carlo techniques were first applied to compute integrals and subsequently became very popular in physics and computer science (e.g., Reinforcement Learning [11]).

In combinatiorial games, Monte-Carlo sampling can be applied without including any domain knowledge, except for the rules of the game at hand. It evaluates a state s by playing

random games until the end. The outcome of these games is then statistically analyzed, e.g., by calculating the mean outcome. This value then constitutes the evaluation of s. Monte Carlo evaluation has been applied successfully in non-deterministic games, such as Bridge [10], Scrabble [8] and Poker [9]. In addition, it was introduced recently in deterministic games [7]. In the game of Go, which is resistant to easily developed evaluation functions [6], Monte-Carlo sampling techniques have yielded convincing results recently, posing an alternative to common knowledge-intensive evaluation functions.

Given the success of search algorithms that use Monte-Carlo evaluation in adversarial games (i.e., a multi-agent setting), this article examines the possibility of applying such a search algorithm, *Monte-Carlo Tree Search* (MCTS), to a single-agent task domain.

Production management problems. As a task domain for a comparison between our proposed algorithm MCTS and other algorithms, we use *production management problems* (PMPs) [12]. In summary, these problems can be defined as planning problems which require parameter optimization over time and can be addressed by the selection of actions with side effects. They contain the following four elements. First, there is a fixed set of *products* – the *size* of the problem is equivalent to the size of this set. Second, there is a fixed set of *production actions*, which are used to convert certain products into other products or to obtain one or more products. These actions may require time and money. Some actions may also produce money (selling actions). Third, there are *constraints* such as a limited amount of time or money available. Fourth, the goal for a problem solver is to produce as much as possible of some of the products, denoted as the *goal products*. This can be achieved by developing an optimal sequence (or plan) of actions, given the available products, actions, constraints and money. The complexity of these problems grows exponentially with the number of products and actions, as will be shown below.

In [12], PMPs have been addressed by applying two planning algorithms; one using a fixed heuristic (Fixed Planning Heuristics or FPH), and one using an evolutionarily optimized heuristic (Evolutionary Planning Heuristics or EPH). The authors conclude that EPH performs better than FPH, mainly because EPH is able to take into account the current state of the problem and because no limit was imposed on the time available to find a solution. It was observed that large PMPs (i.e., with 30 products) require a very long optimization time before a satisfactory outcome is reached, and FPH sometimes leads to better results in these problems, as EPH terminates with a clearly suboptimal solution.

Our contribution. In this paper, we introduce Monte-Carlo Tree Search (MCTS) as a potential method for production management problems. We show that (i) MCTS reaches at least the same score as EPH in problems of sizes $n \leq 24$ while using substantially less calculation time; (ii) MCTS outperforms EPH both in score and runtime in problems of size $24 < n \leq 40$ and (iii) the MCTS algorithm is able to address even larger problems in reasonable time.

The performance of MCTS on PMPs gives us information on the expected performance of this algorithm on other task domains related to search and games, for four reasons, viz. (1) PMPs resemble the problems typically addressed by search algorithms in games (i.e., complete information, discrete time steps), (2) PMPs are easy to formalize and implement, (3) PMPs are intractable problems, yet they are scalable to any desired size and (4) it is possible to develop difficult PMPs, even if these problems do not require adversarial planning or reasoning with uncertainty.

Structure of this article. In Section 2, we present a formalization of PMPs and briefly discuss the complexity of such problems. In Section 3, we discuss the algorithms used in previous research and this research. In Section 4, we look at the experimental setup and the results obtained with regard to performance and running times of the algorithms discussed. In Section 5, we discuss results, conclude and look at future work.

2 Formalizing production management problems

Formalization. In this section, we present a formalization of production management problems. As in [12], we focus on a simplified version for which we ignore non-determinism and earning additional money that can be utilized to cover the costs of actions. The resulting problem can be formalized as follows.

- P is the set of products $\{p_1, \ldots, p_n\}$. The *size* of the problem, $|P|$, is denoted by n.
- The possibly infinite set of possible problem states is denoted by S. The problem state $s_t \in S$ at a certain moment in time (starting with s_0) is a tuple $(m_{s_t}, q_{s_t}(p))$. Here, m_{s_t} is the money available in this state, and the function $q_{s_t} : P \to \mathbb{N}$ defines the quantity available for each product in the state s_t.
- $G \subset P$ is the set of goal products. The reward per goal product, $r : P \to \mathbb{R}^+$, is specified by $p \notin G \to r(p) = 0$ and $p \in G \to r(p) > 0$. The total reward in a state s_t can be calculated using the function $R(s_t) = \sum_{p \in G} r(p) \cdot q_{s_t}(p)$. The goal of the problem is to reach a state s_t in which $R(s_t)$ is optimal.
- A denotes the set of actions that enable the transition from one state to another. Here, $c : A \to \mathbb{N}^+$ denotes the cost of action a. Due to this cost, we will have less money to spend in every subsequent state, and thus we ensure that any problem will have a finite number of possible actions. $t : A \to \mathbb{N}$ denotes the time action a takes to complete, assuming discrete timesteps. The function $in : A \to \mathcal{P}(P \times \mathbb{N}^+)$ denotes the amounts of products required to execute a; $out : A \to \mathcal{P}(P \times \mathbb{N}^+)$ denotes the amounts of products produced by a if it is executed.

Additional constraints. For this research, we introduce five constraints in addition to this formalism, also introduced in [12]. PMPs based on the formalism and these additional constraints fall more or less within the same class of difficulty and possess some convenient properties such as a strict ordering of actions and guaranteed solvability. However, PMPs that adhere to these additional constraints are not essentially easier than PMPs that do not adhere to them.

1. Every action requires at most two products and produces one or two products. Every product is produced by exactly two actions.
2. Costs and durations of actions can be selected from a limited domain for all $a \in A$: $c(a) \in \{1, 2, 3, 4\}$ and $t(a) = 1$. We limit the domain for the coefficients x for required and produced products p by implying the condition $(p, x) \in in(a) \vee (p, x) \in out(a) \to x \in \{1, 2, 3, 4\}$.
3. Cycles in the product chain are prevented: An action cannot produce products with a lower index than the highest index of its required products, plus one.
4. We define $G = \{p_n\}$ and $r(p_n) = 1$.
5. The problem solver starts with an initial amount of money equal to $20 \cdot n$ and none of the products present. The number $20 \cdot n$ is intentionally rather small to keep the problem complexity manageable. Thus, $s_0 = (m_{s_0}, q_0)$ with $m_{s_0} = 20 \cdot n$ and $p \in P \to q_0(p) = 0$.

A small example PMP that adheres to the formalism and constraints presented here, is shown in Figure 1. For legibility, all coefficients have been omitted. Circles represent actions and squares represent products. For example, action A requires products p_2 and p_3 and produces the product p_4. The goal product is p_4, which can be produced by using various product chains, e.g., by using action C to produce products p_1 and p_2, and then using action B to produce p_3 and p_4. Some other product chains are also possible.

Complexity. Experimental results obtained using an algorithm that generates random instances of PMPs adhering to both the formalism and the additional constraints, show that $|A| \approx n + 1$. Solving such PMPs is a hard task. Using a mathematical formalization of PMPs, we were able to reduce the NP-Hard 3SAT problem to a PMP, showing that PMPs are also NP-Hard. This reduction is not included here due to space constraints.

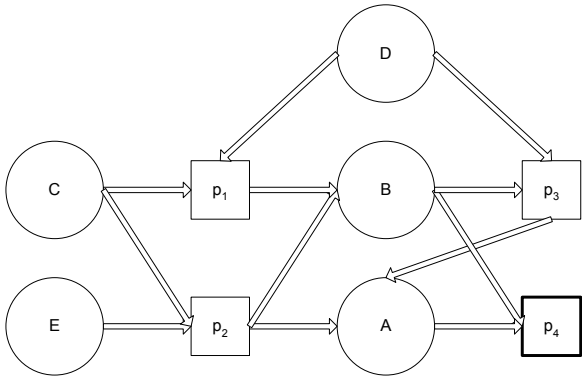


Figure 1: Example of a PMP that adheres to the formalism and constraints presented in this paper.

3 Methods

Fixed Planning Heuristics. In [12], two algorithms, both based on abductive planning, are proposed and tested, viz. Fixed Planning Heuristics (FPH) and Evolutionary Planning Heuristics (EPH). The FPH algorithm generates a plan every time an action must be selected. As a first step, the algorithm determins the actions that produce the goal product p_n (in Figure 1, $n = 4$ and actions A and B produce the goal product p_4). From these actions, it chooses the one that contributes the most to the goal at the lowest price. If this action is executable, it is returned as the action that will be executed. If it is not executable, the algorithm finds the best action that makes it as executable as possible. This criterium is determined using a heuristic that provides approximate information; it cannot be calculated exactly for all actions, due to the problem structure [12]. This process continues until an action is found that is executable, and this action is then chosen as the current action.

Note that the requirement for a heuristic in FPH has a lot in common with the requirement of an evaluation function in search algorithms. For a complete explanation of the heuristic designed for PMPs, we refer to [12].

Evolutionary Planning Heuristics. The Evolutionary Planning Heuristics (EPH) algorithm is a combination of FPH and evolutionary computation. Instead of a manually designed heuristic for the traversal of the plan-space search tree, the algorithm uses a heuristic that is generated by a neural network. The input of this network consists of the current problem state s_t. The output consists of a preference value for each action. The neural network is trained by weight optimization with a genetic algorithm, by means of offline learning. A single problem is repeatedly offered until the algorithm, starting from the state s_0, is able to reach a state s in which the reward $R(s)$ is satisfactory.

This algorithm is interesting because it eliminates the need for a manually developed heuristic, or more broadly, the need for a manually developed evaluation function. As has been remarked above, the elimination of this need is a valuable contribution to the field of planning and search algorithms because many domains are resistant to manually developed evaluation functions. Since the research presented in [12] indicates that EPH, an algorithm without a manually developed evaluation function, is able to address PMPs rather successfully, we are interested in applying another such algorithm: Monte-Carlo Tree Search.

Monte-Carlo Tree Search. In summary, Monte-Carlo Tree Search is a tree search and expansion algorithm which uses Monte-Carlo sampling as its evaluation function and is both explorative and exploitative. Each node of the search tree contains a current problem

state, the possible actions, and some other information about this state, i.e., the mean value and the standard deviation of the simulations made starting from this node.

The algorithm works in two steps. First, the root node is expanded recursively to a certain depth (6 or 7 in our experiments). The current score of all newly created nodes is initially set to 0, because it is not known yet how to build the goal product. Second, a certain number of *simulated games* are played, according to Algorithm 1, which uses Algorithm 2 to traverse the inner nodes of the tree.[1] After every simulated game, the current score of all nodes on the path to the root of the tree is updated in Algorithm 1 according to the information obtained in the simulated game.

Data: Current game tree
Result: Update $current\_score$
$current\_node \leftarrow root\_node$
while *current_node is not a leaf node* **do**
  $current\_node \leftarrow action\_selection(current\_node)$
end
while $number\_of\_possible\_actions \geq 1$ **do**
  $Perform\_random\_action()$;
end
$value\_of\_this\_simulation \leftarrow amount\_of\_goal\_product\_created$
if $value\_of\_this\_simulation > score$ **then**
  $score \leftarrow value\_of\_this\_simulation$
end

Algorithm 1: Playing a simulated game.

Data: Current node
Result: Next node
$O_{bj} \leftarrow value\_of\_the\_current\_best\_child\_node$
foreach *child node* m **do**
  $V_m \leftarrow mean\_value\_of\_node$
  $\sigma_m \leftarrow standard\_deviation\_of\_V_m$
  $U_m(O_{bj}) \leftarrow Erfc(G \cdot \frac{O_{bj}-V_m}{\sqrt{2}\cdot\sigma_m})$
end
foreach *child node* m **do**
  $P_m \leftarrow \frac{U_m(O_{bj})}{\sum_{i\in M} U_i(O_{bj})}$.
end
return node m with a probability P_m

Algorithm 2: Action-selection strategy.

In Algorithm 2, $Erfc$ represents the complementary error function, i.e.,

$$Erfc(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-u^2} du$$

Moreover, $meanvalue\_of\_node$ is the average of all the random games that have been played from this node, which is updated incrementally. G denotes the *greediness* of the algorithm. With a greediness of 1, the algorithm would stop considering an action only when it has been proved to be inferior to its siblings. With such a setting, the depth which can be achieved is very shallow. Otherwise than in adversarial searches, it is not mandatory

[1] Algorithm 2 is based on the Central Limit Theorem and has common points with the algorithms developed for the Multi-Armed Bandit Problem such as Softmax or Exp3 [11]. Implementing and comparing such other algorithms with MCTS will be a topic of interest for future research.

	Size	10	15	20	25	30	35	40	50
	Problems tested	400	400	400	500	432	500	400	200
EPH	Score	131.7	182.6	171.8	170.6	216.6	183.0	152.4	-
	Score var	6.1	9.8	8.2	7.6	13.4	7.7	8.8	-
	Time (s)	6.7	22.2	48.1	85.6	191.5	293.3	552.0	-
MCTS	Score	112.0	158.6	159.5	178.9	241.4	226.7	243.4	278.4
	Score var	6.2	9.9	8.2	7.6	13.5	8.0	9.9	24.4
	Time (s)	5.4	11.5	20.4	40.2	92.9	170.3	241.2	462.0
Ran	MCTS>EPH	0.0	0.0	0.0	0.999	1.0	1.0	1.0	-
	tMCTS<tEPH	1.0	1.0	1.0	1.0	1.0	1.0	1.0	-
Count	MCTS>EPH	20.0%	24.3%	36.5%	55.6%	61.9%	72.9%	88.0%	-

Table 1: Results for EPH and MCTS on problems with various sizes.

that the algorithm explores every possible answer of the opponent. Thus, the greediness parameter can be set to a larger value (around 100 in our experiments). The algorithm then performs so-called *forward pruning* and is able to explore to a depth greater than 200.

4 Experiments and results

In this section, we will discuss our experimental setup and results obtained. In our experiments, we offer a large number of PMPs, varying in size between $n = 10$ and $n = 50$. All problems adhere to the formalism and the additional constraints presented in Section 2. They are generated randomly, but use a seed so that we can compare the algorithms on the same problems.

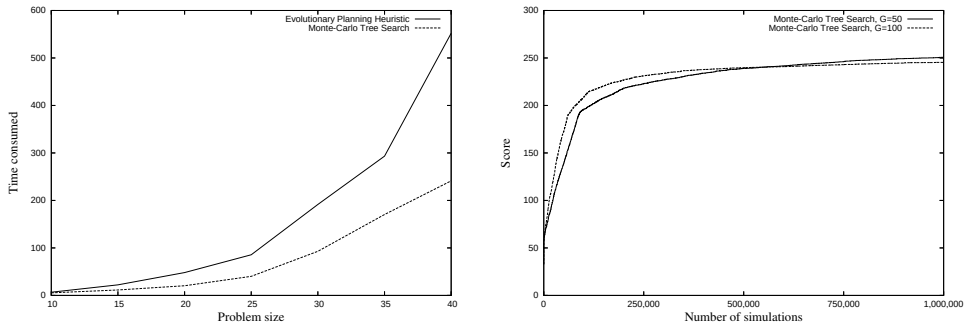
Comparison of EPH with FPH. In [12], the authors compared FPH and EPH on problems of size $n \leq 30$. The average scores obtained by EPH for problems of sizes up to 30 were better than those obtained by FPH. A randomization test [13] was used to determine the probability that EPH was indeed better. For sizes up to $n = 21$, this probability was 1; for larger problems, it decreased slowly to 0.889 for size $n = 30$. Further results for FPH are omitted here. For a complete overview, we refer the interested reader to [12].

Comparison of MCTS with EPH. In our experiments, we compare EPH and MCTS on problems of size $10, 15, \ldots, 40$. Both the MCTS and EPH algorithms are allowed to run until no further improvement is made for a fixed number of iterations (100 generations of 50 individuals in EPH, 50,000 simulations in MCTS). Results are given in Table 1. First, we indicate the number of problems experimented on. Second, we show average scores including sample variances (i.e. $\sigma/\sqrt{n}$), and average running times per problem. A graphical representation of the average running times of MCTS and EPH is given in Figure 2(a). Third, we include the results of two randomization tests, indicating the probabilities that (i) MCTS reaches a better average score than EPH and (ii) MCTS is faster on average than EPH. All randomization tests were performed with 10,000 iterations. Fourth, we print the percentage of problems on which MCTS achieved a better score than EPH.

In the table, we see that, given the current termination conditions, MCTS achieves a lower score than EPH on smaller problems, but is faster. When we increase the problem size, MCTS eventually scores higher than EPH, while maintaining faster performance.[2]

MCTS on even larger problems. MCTS, unlike EPH, is able to find a solution for prob-

[2]Since MCTS is both running faster and scoring less for sizes up to $n = 20$, we assessed the effect of changing the termination conditions for MCTS: we increase the number of simulations allowed without further improvement to 75,000. We observed that MCTS becomes significantly slower than EPH, but still does not manage to reach the same score. Thus, EPH is clearly a better algorithm for small problems and MCTS wins on large problems. The exact problem size for which MCTS becomes both faster and better than EPH with high probability depends on the settings chosen (for MCTS: greediness, termination conditions; for EPH: population size, mutation probability, et cetera) and the actual problems addressed.



(a) (b)

Figure 2: (a) Time consumed by MCTS and by EPH. (b) Speed of convergence of MCTS.

lems with $n = 50$ within reasonable time. Using the current termination conditions, EPH is too slow to solve large numbers of such problems. With less time-consuming conditions, the algorithm's performance drops rapidly (as the authors observed while tuning them). In MCTS however, we can change the termination conditions in order to lower the running time without losing much of the score, as will be shown below. The results presented for $n = 50$ in Table 1 were obtained by running exactly 500,000 simulations for every problem.

Convergence of MCTS. We conduct an experiment on our test set of problems of size $n = 30$ to determine the convergence speed of MCTS when we (i) change the greediness parameter and (ii) change the termination conditions (i.e., we perform more or less simulations). We run a fixed number of simulations (1,000,000) on our test set and average the results found during and after these simulations. Two different greediness factors G have been tested: $G = 50$ and $G = 100$. Results are reported in Figure 2(b).

One can observe that changing the greediness by a factor of two does not influence the average result much, neither at the end of the simulation, nor during the simulation. Moreover, it can be seen that the algorithm would have performed nearly as well (230 on average instead of 241) with only half of the simulations (250,000 instead of 500,000); i.e., the algorithm would have been 50% faster and would have scored only 4% lower. Thus, changing the termination conditions does not drastically influence the score achieved, and we can make the algorithm substantially faster by allowing a slightly lower score.

5 Discussion and conclusion

In this paper, we presented a Monte-Carlo search framework for single-agent problems and games. It was compared to an adaptive algorithm presented in earlier research. The task domain for this comparison, the production management problem, has three major characteristics: (i) the problem is NP-Hard, (ii) the search space is scalable and (iii) it is difficult to build a good evaluation function. Hence, this problem is an adequate testbed for new search methods.

Earlier research [12] resulted in an evolutionarily optimized heuristic for a planning algorithm (EPH), and in this article, we present a search algorithm that employs a Monte-Carlo based evaluation function (MCTS). Experiments show that on small problems (e.g., with 10 products), EPH achieves a better score than MCTS. On larger problems (i.e., with 25 or more products), MCTS is faster than EPH and achieves much better scores. Furthermore, we see that MCTS can be modified in such a way that running times decrease significantly while scores decrease only slightly. Whether running times or scores are chosen as the main criterium, depends on the task domain.

In general, Monte-Carlo Tree Search can lead to good results in task domains which have the following characteristics: (i) the search space is large; (ii) there is no obvious evaluation function; and (iii) there are several different paths to achieve a good score.We show in this paper that Monte-Carlo Tree Search constitutes an interesting alternative to an evolutionary algorithm. Our next research issue will be to assess the performance of our algorithm in other complex task domains to which other computational intelligence methods have already been applied, including adversarial settings.

We believe that the research reported here can still be improved in several ways. Among them, promising to us are playing pseudo-random games instead of completely random games (which requires some domain knowledge) and the automatic adjustment of the greediness parameter in the algorithm.

Acknowledgments This work is financed by the Dutch Organisation for Scientific Research (NWO) in the framework of the project Go for Go, grant number 612.066.409. It is partially funded by the 'Breedtestrategie' programme of the Universiteit Maastricht. The authors wish to thank Ida Sprinkhuizen-Kuyper, Nico Roos and Jeroen Kuipers for their contributions to this research. And finally, thanks to Sophia Katrenko for providing wireless facilities when needed.

References

[1] Judea Pearl. The solution for the branching factor of the alpha–beta pruning algorithm and its optimality. *Communications of the ACM*, 25:559–564, 1982.

[2] Jonathan Schaeffer, Robert Lake, Paul Lu, and Martin Bryant. CHINOOK: The world man-machine checkers champion. *AI Magazine*, 17(1):21–29, 1996.

[3] Dennis Breuker. *Memory versus Search*. PhD thesis, University of Maastricht, 1998.

[4] Bruno Bouzy. History and territory heuristics for Monte-Carlo Go. In K. Chen et al, editor, *Joint Conference on Information Sciences*, page 4, 2005.

[5] A. Nareyek. Choosing search heuristics by non-stationary reinforcement learning. In M. G. C. Resende and J. P. de Sousa, editors, *Metaheuristics: Computer Decision-Making*, pages 523–544. Kluwer Academic Publishers, 2003.

[6] Martin Müller. Not like other games – why tree search in Go is different. In *Fifth Joint Conference on Information Sciences (JCIS)*, 2000.

[7] Bruno Bouzy and Bernd Helmstetter. Monte Carlo Go Developments. In H. Jaap van den Herik, Hiroyuki Iida, and Ernst A. Heinz, editors, *Proceedings of the Advances in Computer Games Conference (ACG-10)*, pages 159–174. Kluwer Academic Publishers, 2003.

[8] Brian Sheppard. World-championship-caliber scrabble. *Artificial Intelligence*, 134(1):241–275, 2002.

[9] Darse Billings, Aaron Davidson, Jonathan Schaeffer, and Duane Szafron. The challenge of poker. *Artificial Intelligence*, 134(1):201–240, 2002.

[10] Stephen Smith, Dana Nau, and Tom Throop. Computer Bridge - A Big Win for AI planning. *AI Magazine*, 19(2):93–105, 1998.

[11] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[12] Steven de Jong, Nico Roos, and Ida Sprinkhuizen-Kuyper. Evolutionary planning heuristics in production management. In *Proceedings of the BNAIC 2005, Brussels, Belgium*, 2005.

[13] Paul R. Cohen. *Empirical Methods for Artificial Intelligence*. MIT Press, 2002.

In this paper, we apply a search algorithm based on Monte-Carlo evaluation, Monte-Carlo Tree Search, in the task domain of production management problems. These can be defined as single-agent problems which consist of selecting a sequence of actions with side effects, leading to high quantities of one or more goal products. They are challenging and can be constructed with highly variable difficulty. Earlier research yielded an offline learning algorithm that leads to good solutions, but requires a long time to run. We show that Monte-Carlo Tree Search leads to a solution in a shorter period of time than this algorithm, with improved solutions for large problems. Our findings can be generalized to other task domains. 3 Monte Carlo Tree Search. 4 Opponent modeling for Poker: predicting moves. 5 Simulating showdowns: predicting holdings.Â We built a computer Poker player using these algorithms and Monte Carlo Tree Search: an anytime algorithm that estimates the expected values of moves using simulations. The resulting program can hold its own in 2 player games against both novices and experienced human opponents. i. Monte-Carlo Tree Search has among others been applied successfully to the game of Go [5, 7], the Production Management Problem [4], the game of Clobber [9], Amazons [10], and the Sailing Domain [9]. These games are either one-player games with a chance factor or two-player games without a chance factor.Â Furthermore, thanks to the works of Tesauro, it has already been proved that a very strong AI can be written for this game [11]. The actual approach is to train a neural network by playing millions of oine games. In this paper we present a contrasting approach consisting of using only online simulated games from the current position. 2 The game of Backgammon. Backgammon is a board game for two players in which checkers are moved according to the roll of two dice.