Non-Linear Monte-Carlo Search in Civilization II

S.R.K. Branavan **David Silver *** **Regina Barzilay**

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{branavan, regina}@csail.mit.edu

* Department of Computer Science
University College London
d.silver@cs.ucl.ac.uk

Abstract

This paper presents a new Monte-Carlo search algorithm for very large sequential decision-making problems. We apply non-linear regression within Monte-Carlo search, online, to estimate a state-action value function from the outcomes of random roll-outs. This value function generalizes between related states and actions, and can therefore provide more accurate evaluations after fewer roll-outs. A further significant advantage of this approach is its ability to automatically extract and leverage domain knowledge from external sources such as game manuals. We apply our algorithm to the game of Civilization II, a challenging multi-agent strategy game with an enormous state space and around 10^{21} joint actions. We approximate the value function by a neural network, augmented by linguistic knowledge that is extracted automatically from the official game manual. We show that this non-linear value function is significantly more efficient than a linear value function, which is itself more efficient than Monte-Carlo tree search. Our non-linear Monte-Carlo search wins over 78% of games against the built-in AI of Civilization II. [1]

1 Introduction

Monte-Carlo search is a simulation-based search paradigm that has been successfully applied to complex games such as Go, Poker, Scrabble, multi-player card games, and real-time strategy games, among others [Gelly *et al.*, 2006; Tesauro and Galperin, 1996; Billings *et al.*, 1999; Sheppard, 2002; Schäfer, 2008; Sturtevant, 2008; Balla and Fern, 2009]. In this framework, the values of game states and actions are estimated by the mean outcome of *roll-outs*, i.e., simulated games. These values are used to guide the selection of the best action. However, in complex games such as Civilization II, the extremely large action space makes simple Monte-Carlo search impractical, as prohibitively many roll-outs are needed to determine the best action. To achieve good performance in such domains, it is crucial to generalize from the outcome of roll-outs to the values of other states and actions.

[1] The code and data for this work are available from http://groups.csail.mit.edu/rbg/code/civ/

> *Several factors influence a city's production of shields: the terrain within your city radius is most important.*
>
> *You might find it worthwhile to set Settlers to improving the terrain squares within your city radius.*
>
> *Beyond terrain , the form of government your civilization chooses can cause each city to spend some of its raw materials as maintenance*

Figure 1: An extract from the game manual of Civilization II.

A common approach to value generalization is *value function approximation* (VFA), in which the values of all states and actions are approximated using a smaller number of parameters. Monte-Carlo search can use VFA to estimate state and action values from the roll-out scores. For instance, *linear Monte-Carlo search* [Silver *et al.*, 2008] uses linear VFA: it approximates the value function by a linear combination of state features and weights.

In this paper, we investigate the use of *Non-linear VFA* within Monte-Carlo search to provide a richer and more compact representation of value. This value approximation results in better generalization between related game states and actions, thus providing accurate evaluations after fewer roll-outs. We apply non-linear VFA *locally in time*, to fit an approximate value function to the scores from the current set of roll-outs. These roll-outs are all generated from the current state, i.e., they are samples from the sub game starting from *now*. The key features of our method are:

- **Fitting Non-Linear VFA to local game context** Previous applications of non-linear VFA [Tesauro, 1994] estimated a global approximation of the value function. While effective for games with relatively small state spaces and smooth value functions, this approach has so far not been successful in larger, more complex games, such as Go or Civilization II. Non-linear VFA, applied globally, does not provide a sufficiently accurate representation in these domains. In contrast, *the local value function* which focuses solely on the dynamics of the current subgame, which may be quite specialized, can be considerably easier to approximate than the full global value function which models the entire game.
- **Leveraging domain knowledge automatically extracted from text** Game manuals and other textual resources provide valuable sources of information for selecting a game strategy. An example of such advice is shown in Figure 1. For instance, the system may learn

that the action *irrigate-land* should be selected if the words "improve terrain" are present in text. However, the challenge in using such information is in identifying which segments of text are relevant for the current game context; for example, the system needs to identify the second sentence in Figure 1 as advice pertinent to the irrigation action. These decisions can be effectively modeled via hidden variables in a non-linear VFA.

We model our non-linear VFA using a four-layer neural network, and estimate its parameters via reinforcement learning within the Monte-Carlo search framework. The first layer of this network represents features of the game state, action, and the game manual. The second layer represents a linguistic model of the game manual, which encodes sentence relevance and the predicate structure of sentences. The third layer corresponds to a set of fixed features which perform simple, predefined transformations on the outputs of the first two layers. The fourth and final layer represents the value function. During Monte-Carlo search, the roll-out scores are used to train the entire value function online, including the linguistic model. The weights of the neural network are updated by error backpropagation, so as to minimize the mean squared error between the value function (the network output) and the roll-out scores.

We apply our technique to Civilization II, a notoriously large and challenging game.[2] It is a cooperative, multi-agent decision-making problem with around 10^{21} joint actions and an enormous state space of over 10^{188} states.[3] To make the game tractable, we assume that actions of agents are independent of each other given the game state. We use the official game manual as a source of domain knowledge in natural language form, and demonstrate a significant boost to the performance of our non-linear Monte-Carlo search. In full-length games, our method wins over 78% of games against the built-in, hand-crafted AI of Civilization II.

2 Related Work

Monte-Carlo search has previously been combined with *global* non-linear VFA. The original TD-Gammon [Tesauro, 1994] used a multilayer neural network that was trained from games of self-play to approximate the value function. Tesauro and Galperin [1996] subsequently used TD-Gammon's neural network to score the roll-outs in a simple Monte-Carlo search algorithm; however, the neural network was *not* adjusted online from these roll-outs. In contrast, our algorithm re-learns its value function, online, from each new set of roll-outs – just as if TD-Gammon had re-trained its neural network after each move, specifically for the subgame of Backgammon starting from that new position.

Monte-Carlo search has previously been extended to the special case of linear VFA [Silver *et al.*, 2008]. Generalizing to non-linear VFA provides two fundamental advantages. Firstly, due to its greater representational power, it generalizes better from small numbers of roll-outs. This is particularly important in complex games, such as Civilization II, in which simulation is computationally intensive, severely limiting the number of roll-outs. Secondly, non-linear VFA allows for multi-layered representations, such as neural networks, which can automatically learn features of the game state. Furthermore, we show that a multi-layered, non-linear VFA can automatically incorporate domain knowledge from natural language documents, so as to produce better features of the game state, and therefore a better value function and better overall performance.

A number of other approaches have previously been applied to turn-based strategy games [Amato and Shani, 2010; Wender and Watson, 2008; Bergsma and Spronck, 2008; Madeira *et al.*, 2006; Balla and Fern, 2009]. However, given the enormous state and action space in most strategy games, much of this previous work has focused on smaller, simplified or limited components of the game. For example, Amato and Shani [2010] use reinforcement learning to dynamically select between two hard-coded strategies in the game Civilization IV. Wender and Watson [2008] focus on city building in Civilization IV. Madeira et al. [2006] address only the combat aspect of strategy games, learning high-level strategies, and using hard-coded algorithms for low-level control. Balla and Fern [2009] apply Monte-Carlo tree search with some success, but focus only on a tactical assault planning task. In contrast to such previous work, our algorithm learns to control *all* aspects of the game Civilization II, and significantly outperforms the hand-engineered, built-in AI player.

3 Background

Our task is to play and win a strategy game against a given opponent. Formally, we represent the game as a Markov Decision Process (MDP) $\langle S, A, T, R \rangle$, where S is the set of possible states and A is the space of legal actions. Each state $s \in S$ represents a complete configuration of the game world, including attributes such as available resources and the locations of cities and units. Each action $a \in A$ represents a joint assignment of actions to each city and each unit. At each step of the game, the agent executes an action a in state s, which changes the state to s' according to the state transition distribution $T(s'|s, a)$. This distribution incorporates both the opponent's action selection policy, and the game rules. It is not known explicitly; however, state transitions can be sampled by invoking the game code as a black-box simulator. Each state s has a *reward* $R(s) \in \mathbb{R}$ associated with it, also provided implicitly by the game code.

A *policy* $\pi(s, a)$ is a stochastic action selection strategy that gives the probability of selecting action a in state s. The *action-value function* $Q^\pi(s, a)$ is the expected final reward after executing action a in state s and then following policy π. In large MDPs, it is impractical to represent action-values by table-lookup, with a distinct value for every state and action. In such cases, the action-values can be represented by value function approximation – i.e., $Q(s, a) = f(s, a; \theta)$, where f is a differentiable function with parameter vector θ. These parameters can be updated by applying non-linear regression

[2] Civilization II was placed #3 in IGN's 2007 list of the top 100 video games of all time (http://top100.ign.com/2007)

[3] These measures of the state and action spaces are empirical estimates from our experimental setup. They are based on the average number of units, available unit actions, state attributes and the observed state attribute values.

to the final utilities. Specifically, the parameters are adjusted by gradient descent to minimize the mean-squared error between the action-value and the reward $R(s_\tau)$ at final state s_τ for every observed state s and action a.

$$\begin{aligned}\Delta\theta &= -\frac{\alpha}{2}\nabla_\theta \left[R(s_\tau) - Q(s,a)\right]^2 \\ &= \alpha \left[R(s_\tau) - Q(s,a)\right] \nabla_\theta f(s,a;\theta) \qquad (1)\end{aligned}$$

Monte-Carlo Search Monte-Carlo search is a simulation-based search paradigm for dynamically estimating the action-values from a root state s_t. This estimate is based on the results of multiple *roll-outs*, each of which samples the final reward in a simulated game that starts from s_t.[4] Specifically, in each roll-out, actions are selected according to a simulation policy π, and state transitions are sampled from the transition distribution T. At the end of the simulation, the reward $R(s_\tau)$ at the final state s_τ is measured, and the action-value function is updated accordingly. As in Monte-Carlo control [Sutton and Barto, 1998], the simulation policy may then be improved, to take account of this new information and direct the simulations towards the highest scoring regions of the state space. After n simulations have been executed, the actual action with highest final reward is selected and played, and a new search begins from root state s_{t+1}.

Monte-Carlo tree search (MCTS) [Coulom, 2006] uses a search tree to represent the action-value function. Each node of the search tree contains a single value for the state corresponding to that node. Each simulation traverses the search tree without backtracking. After each simulation, the action-values of all traversed nodes are updated to reflect the new mean reward. If all nodes of the search tree are expanded, this algorithm is equivalent to Monte-Carlo control with table-lookup, applied to the subgame starting from s_t. Monte-Carlo tree search has achieved human master-level performance in 9×9 Go [Gelly *et al.*, 2006].

In *linear Monte-Carlo search*[5] [Silver *et al.*, 2008] the value function is represented by a linear combination of features $\phi(s,a)$ and a parameter vector θ: $Q(s,a) = \theta \cdot \phi(s,a)$. The parameters are updated by applying online linear regression to the final simulation utilities via gradient descent: $\Delta\theta = \alpha \left[R(s_\tau) - Q(s,a)\right] \phi(s,a)$. This algorithm is equivalent to Monte-Carlo control with linear function approximation [Sutton and Barto, 1998], again applied to the subgame starting from s_t. Often, a linear approximation that is specialized to the current subgame can be much more accurate than a global linear approximation; it outperforms a simple Monte-Carlo tree search in 9×9 Go [Silver *et al.*, 2008].

4 Non-linear Monte-Carlo Search

Non-linear Monte-Carlo search combines Monte-Carlo search with non-linear VFA. The value function is represented by an arbitrary, smooth function approximator. At the end of each roll-out, the parameters of the function approximator are updated online by Equation 1. This produces a non-linear approximation to the current roll-out scores, specialized to the subgame starting from s_t. As in other Monte-Carlo search algorithms, the value function is used to improve the simulation policy and guide roll-outs towards the highest reward regions of the search space.

We now describe a our approach to non-linear VFA which can be applied effectively in the context of Monte-Carlo search. We represent the action-value function by the output of a multi-layer neural network. The architecture of this network incorporates a linguistic model that is designed to automatically extract and incorporate domain knowledge from game manuals.

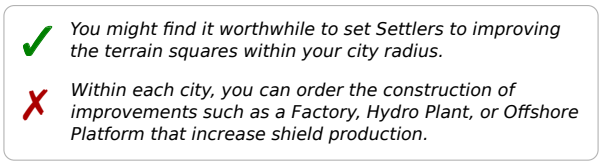

Figure 2: An example sentence relevance decision – if we are attempting to select the best action for a "Settler" unit located near a city, the first of these two sentences is the most relevant.

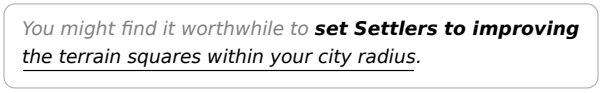

Figure 3: An example predicate labeling of a sentence: words labeled as *action description* are shown in bold, and words labeled as *state description* are underlined. The remaining *background words* are grayed out.

Incorporating Text into MC Search Consider the text shown in Figure 1, which suggests that within your city radius "settlers" should take the action "improve the terrain." One way to incorporate this kind of textual information is to enrich the action-value function with word features in addition to state and action features. However, adding all the words from a manual is unlikely to help, since only a small portion of the document is likely to be relevant for a given state (see example in Figure 2). Therefore, the linguistic model needs to identify the most relevant text segment for each state. Moreover, even when the relevant sentence is known, it is important to discriminate between words that describe *actions* from words that describe *state* (see Figure 3).

Rather than flooding the representation with all possible text features, we automatically extract, from a natural language document, the feature vector that is most effective in predicting the roll-out rewards. We learn these text features jointly with the action-value function approximation, where both processes are guided by backpropagation of error.

Network Architecture As shown in Figure 4, our architecture is a four layer neural network. The *input layer* $\vec{x}$ consists of deterministic, real-valued features $\vec{x}(s_t, a_t, d)$ of the current game state s_t, candidate action a_t, and document d.

The *second layer* consists of two sets of disjoint units $\vec{y}$ and $\vec{z}$ which encode the sentence-relevance and predicate-labeling decisions respectively. These sets of units operate as stochas-

[4] These simulated games may be played against a given heuristic game AI. In our experiments, the built-in AI is used as the opponent.

[5] Linear Monte-Carlo search is a special case of linear temporal-difference search with $\lambda = 1$.

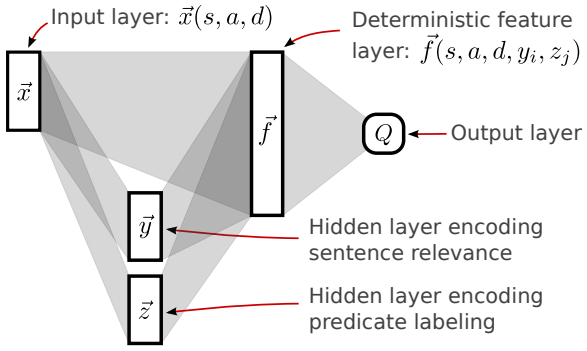

Figure 4: The structure of our network architecture. Each rectangle represents a collection of units in a layer, and the shaded trapezoids show the connections between layers. A fixed, real-valued feature function $\vec{x}(s,a,d)$ transforms the game state s, action a, and strategy document d into the input vector $\vec{x}$. The first hidden layer contains two disjoint sets of units $\vec{y}$ and $\vec{z}$ corresponding to linguistic analyses of the strategy document. These are softmax layers, where only one unit is active at any time. The units of the second hidden layer $\vec{f}(s,a,d,y_i,z_i)$ are a set of fixed real valued feature functions on s, a, d and the active units y_i and z_i of $\vec{y}$ and $\vec{z}$ respectively.

tic 1-of-n softmax selection layers [Bridle, 1990] where only a single unit is activated. The activation function for units in this layer is the standard softmax function:

$$p(y_i|\vec{x}) = e^{\vec{u}_i \cdot \vec{x}} \,/\, \sum_k e^{\vec{u}_k \cdot \vec{x}},$$

where y_i is the i^{th} hidden unit of $\vec{y}$, and $\vec{u}_i$ is the weight vector corresponding to y_i. The activation function for $\vec{z}$ is similar.

Each unit in layer $\vec{y}$ represents a single sentence from the game manual, allowing 1-of-n selection to directly model the sentence relevance decision. Similarly, each unit in $\vec{z}$ represents a unique candidate predicate label for a single word in a sentence, with all units z_i associated with the same word operating in a 1-of-n selection manner. This allows the modeling of the predicate labeling decisions for each sentence word.

The third *feature layer* $\vec{f}$ is deterministically computed given the active units y_i and z_j of the softmax layers, and the values of the input layer. Each unit in this layer corresponds to a fixed feature function $f_k(s_t,a_t,d,y_i,z_j) \in \mathbb{R}$. Finally the *output layer* encodes the action-value function $Q(s,a,d)$, which we modify to also depend on the document d, as a weighted linear combination of the units in the feature layer:

$$Q(s_t,a_t,d) = \vec{w} \cdot \vec{f}.$$

Here $\vec{w}$ is the weight vector.

Parameter Estimation Learning in our method is performed in an online fashion: at each game state s_t, the algorithm performs a *simulated* game roll-out, observes the outcome of the game, and updates the parameters $\vec{u}$ and $\vec{w}$ of the non-linear action-value function $Q(s_t,a_t,d)$. These three steps are repeated a fixed number of times at each actual game state s_t. The information from these roll-outs is used to select the actual game action. The algorithm re-learns the function approximation for every new game state s_t. This specializes the action-value function to the subgame starting from s_t.

As described in Section 3, we estimate the neural network parameters via gradient descent to minimize the mean-squared error of our function approximation. We perform this minimization by standard error backpropagation, which results in the following online updates for the output layer parameters $\vec{w}$:

$$\vec{w} \;\leftarrow\; \vec{w} + \alpha_w \,[Q - R(s_\tau)]\; \vec{f}(s,a,d,y_i,z_j), \qquad (2)$$

where α_w is a learning rate, and $Q = Q(s,a,d)$. The corresponding updates for the softmax layer parameters $\vec{u}$ are:

$$\vec{u}_i \;\leftarrow\; \vec{u}_i + \alpha_u \,[Q - R(s_\tau)]\; Q\, \vec{x}\, [1 - p(y_i|\vec{x})]. \qquad (3)$$

5 Application to Civilization II

We apply our non-linear Monte-Carlo search algorithm to the game of Civilization II, using the official game manual[6] as the source of domain knowledge.

Civilization II is a multi-player turn-based strategy game. Each grid location on the map is a tile of either land or sea, and each tile can have multiple terrain and resource attributes – for example hills, rivers, or coal deposits. Each player in the game controls a civilization consisting of multiple cities and units such as workers, explorers and archers. The player who gains control of the entire world by capturing or destroying all opposing cities and units wins the game.[7] We test our algorithms on the small size two-player game, which is set in a 1000 square map, and we play our algorithm against the built-in AI player.[8]

Strategy documents We use the official game manual of Civilization II as our source of domain knowledge, which contains 2083 sentences of average length 17 words, and has a large vocabulary of 3638 words. We use the Stanford parser [de Marneffe *et al.*, 2006] to generate the dependency parse information for the sentences of this document.

States and Actions We define the game state to be the attributes of each tile in the world map, and the attributes of each player's civilization, cities and units. The the set of legal actions for each civilization, city and unit is state dependent and is defined by the rules. In a 1000 tile Civilization II game, there are 10^{21} joint actions. To deal with this large action space, we assume that the actions of each unit and city are conditionally independent given the state. Actions of the civilization, cities and units are selected using a ϵ-greedy policy with respect to the action-value function approximation. Although our approach does not explicitly coordinate the actions of different units, it can still achieve a certain degree

[6] www.civfanatics.com/content/civ2/reference/Civ2manual.zip

[7] Civilization games can be won by diplomacy, space flight, or war. While for reasons of manageability we focus on the latter, our method can be applied to all three via a suitable reward function.

[8] The built-in AI of Civilization II is allowed bypass game rules to provide challenging opposition to human players. However, both Freeciv – the open-source reimplementation used in this work – and our algorithm are constrained to follow the rules of the game.

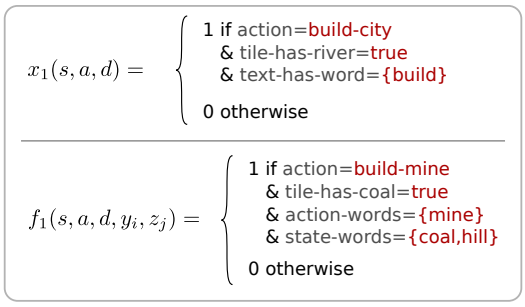

Figure 5: Some example features used by our linguistically guided method: $x_1(\cdot)$ is a feature used to identify the sentence most relevant to the current game state s, and candidate action a. This feature would be active if $a =$ "build-city", the tile in s where the action is being considered has a river, and the candidate sentence has the word "build". Feature $f_1(\cdot)$ combines state, action, and predicate information of the relevant sentence as input to the final layer.

of coordination implicitly, via the roll-outs. For example, if transport-ship A typically selects action "move to city port X" in the roll-outs, then land-unit B can adapt its action to find the best action (e.g., "board transport-ship at port X") in the context of A's usual action.[9]

Reward Function Due to the length and complexity of the game, we truncate all roll-outs after 20 steps. At the end of each roll-out we use the ratio of the game scores between the two players as the reward function.

Features We use three sets of predefined feature functions to convert the attributes of the game state, actions and text into real valued inputs to layers $\vec{y}$, $\vec{z}$ and Q of the neural network.

- *Sentence relevance features* consider attributes of the state, action and document. Some of these features compute text overlap between document words and text labels in the game state and action (e.g., feature $x_1(\cdot)$ in Figure 5).
- *Predicate labeling features* operate on the words and dependency parse information of the document. These features include the Cartesian product of the candidate predicate label with word attributes such as type, part-of-speech tag, and dependency type.
- *Output layer features* are computed based on the sentence relevance decision y_i and the predicate labeling z_j, in addition to state s and action a (e.g., feature $f_1(\cdot)$ in Figure 5).

Overall, our method computes 473,200 features, some examples of which are shown in Figure 5. The set of possible features results from taking the Cartesian product of all observed game attributes, all unique words in the game manual, and all possible predicate labels – no explicit feature selection was done.

[9]While explicit coordination is clearly desirable, it is not the focus of this paper. We leave it as an avenue for potential future work.

6 Experimental Setup

Experimental Framework We test our method on the open source implementation of Civilization II, *FreeCiv*.[10] We use a single randomly generated game start state as the initial state in all experiments. For each actual step in state s_t, 500 Monte-Carlo roll-outs are executed from state s_t, with each roll-out lasting 20 simulated steps. The action-value function is updated by Equation 2 and 3 for the first 3 steps of each roll-out. Additionally, each update step iterates 7 times over the current set of roll-outs. Each civilization, city or unit policy selects its action by an ϵ-greedy algorithm that maximizes $Q(s,a)$ with probability $1-\epsilon$ and selects a random action with probability ϵ. For all methods, all model parameters are initialized to zero.

Experiments were run on typical desktop PCs with single Intel Core i7 CPUs (4 hyper-threaded cores per CPU). All algorithms were implemented to execute 8 simulation roll-outs in parallel. In this setup, a single 100 step game runs in approximately 1.5 hours.

Baselines We compare our method against four baselines. The first, *MC Tree Search*, is a Monte-Carlo tree search algorithm, where each civilization, city and unit in the game has a separate search tree. Each node in the tree represents a unique sequence of actions for the agent, ignoring all other agents. A node's value is the average outcome of all simulations starting at that node. We use a uniform roll-out policy beyond the periphery of the tree. Among all the approaches we tested, this was the only algorithm which constructs a search tree.

The second baseline, *Linear MC*, implements a linear action-value function approximation in the Monte-Carlo search framework, and is similar to Silver et al. [2008]. The third baseline, *Non-linear MC*, uses a non-linear action-value approximation with Monte-Carlo search. This baseline's model structure is similar to our method, and is implemented as a four layer neural network. However, it is not given any documents as part of its input. These two baselines use approximately 340 and 2450 features respectively.

The final baseline, *Random-Text*, is identical to our method, except it is given a document constructed by randomly permuting the words of the original strategy guide. This ensures that the overall statistics of the document are preserved, while removing the semantic content.

Evaluation Metrics We evaluate the performance of our algorithms by their average win rate against the built-in AI of FreeCiv. The built-in AI is a hand-engineered algorithm, designed with extensive knowledge of the game to provide a challenging opponent for human players. We perform two separate evaluations on *full* and *100-step* games. The *100 step game* evaluation is computed by the percentage of games won within the first 100 game steps, treating unfinished games as a loss, averaged over 200 independent runs for each method. The *full game* evaluation is computed by the percentage of full games won, where each game is run to completion, averaged over 50 independent runs for each method.

[10]http://freeciv.wikia.com. Game version 2.2

Method	% Wins	Standard Error
MC Tree Search	0	0.7
Linear MC	17.3	2.7
Non-linear MC	26.1	3.1
Random text	40.3	3.4
Non-Linear Text MC	**53.7**	3.5

Table 1: Percentage of victories in *100-step games* while playing against the built-in game AI. Results are averaged across 200 independent game runs. All differences are statistically significant.

Method	% Wins	Standard Error
Linear MC	45.7	7.0
Non-linear MC	62.2	6.9
Non-Linear Text MC	**78.8**	5.8

Table 2: Percentage of victories in *full games* played against the built-in AI. Results are averaged across 50 games each.

7 Results

Tables 1 and 2 compare the performance of our model (*Non-Linear Text MC*) against several baselines. In both the *100-step game* and *full game* evaluations, *Non-Linear Text MC* significantly outperforms all baselines, winning 53.7% and 78.8% of games respectively. The substantial gain obtained by *Non-linear MC* over *Linear MC* further demonstrates the advantages of non-linear value function approximation. Figure 7 shows examples of feature combinations learned by *Non-linear MC*, providing some intuition for the reasons behind the method's improved performance. Surprisingly, *MC Tree Search* failed to win a single game. This illustrates the difficulties faced by search-tree based algorithms in domains with large branching factors. However, it is possible that a more sophisticated MC tree search algorithm could perform better, for example by improving the roll-out policy or controlling the exploration/exploitation trade-off [Gelly *et al.*, 2006]; or by incorporating prior knowledge or using rapid action value estimation [Gelly and Silver, 2007].

Model Complexity vs Computation Time Trade-off One potential disadvantage of non-linear models is the increase in computation time required for action-value function estimation. To explore this trade-off, we vary the number of simulation roll-outs allowed at each game step, recording the win-rate and average computation time per game. Figure 6 shows the results of this evaluation for 100, 200 and 500 roll-outs. Not surprisingly, these results show that the more complex methods have higher computational demands. However, given a limited amount of time per game step, *Non-Linear Text MC* still produces the best performance, followed by *Non-linear MC*.

The benefits of automatically extracted domain knowledge The performance gain of *Non-Linear Text MC* over *Non-Linear MC* suggests that information extracted from manuals is useful. To further validate this hypothesis, we introduce an additional baseline, *Random Text*, that mirrors the structure of the *Non-Linear Text MC* network, but is given a randomly generated document as input. The low performance of this baseline confirms our hypothesis that the algorithm does benefit from automatically extracted domain knowledge. For a comprehensive evaluation and analysis of the linguistic aspects of our method, see Branavan et al. [2011].

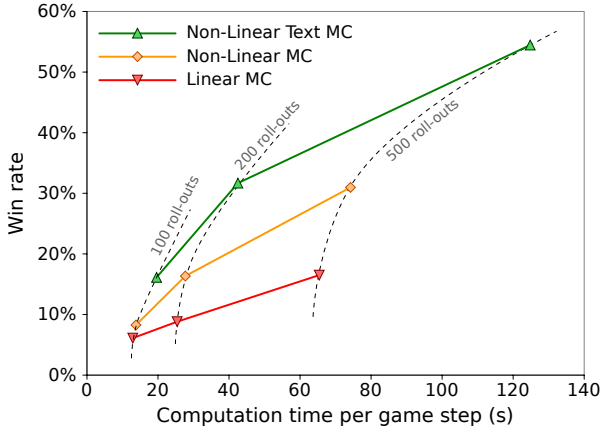

Figure 6: Win rate as a function of computation time per game step. For each Monte-Carlo search method, win rate and computation time were measured for 100, 200 and 500 roll-outs per game step, respectively.

latent variable 0	**latent variable 1**
state : grassland	state : city, grassland, river
unit : settlers	unit : workers
action : build city	action : irrigate-land

Figure 7: Examples of groups of features with high positive weights when two different latent variables are activated in the Non-Linear MC model. These positively weighted features under latent variable 0 would encourage using settlers to build cities on grassland, while the features under latent variable 1 would encourage irrigation of grassland close to both cities and rivers.

8 Conclusions and Future Work

This paper presented a new Monte-Carlo search algorithm for large sequential decision-making problems. The key innovation is to apply non-linear VFA, locally in time, to the current set of roll-outs. Our non-linear architecture leverages domain knowledge that is automatically extracted from text. Our learning algorithm simultaneously constructs a linguistic model, computes features of the game state, and estimates the value function, by backpropagation of error. We show that non-linear VFA is significantly more efficient than linear VFA, and that incorporating a linguistic model is significantly more efficient than an equivalent non-linguistic architecture. Our non-linear Monte-Carlo search algorithm wins over 78% of games against the built-in AI of Civilization II.

Our present method makes the strong assumption that units in the game operate independently of each other. While this assumption enables our method to operate tractably, it also limits the type of strategies that can be learned. For example,

due to this assumption, our model has limited unit coordination ability. It attempts to bypass this limitation by learning an *early rush* strategy—attacking and conquering the opponent early in the game before unit coordination becomes a key requirement. Modeling the value function without this assumption, while challenging, would greatly strengthen the method by allowing it to learn a broader and more robust range of game strategies.

Acknowledgments

The authors acknowledge the support of the NSF (CAREER grant IIS-0448168, grant IIS-0835652), DARPA Machine Reading Program (FA8750-09-C-0172) and the Microsoft Research New Faculty Fellowship. Thanks to Michael Collins, Tommi Jaakkola, Leslie Kaelbling, Nate Kushman, Sasha Rush, Luke Zettlemoyer, the MIT NLP group, and the IJCAI reviewers for their suggestions and comments. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors, and do not necessarily reflect the views of the funding organizations.

References

[Amato and Shani, 2010] Christopher Amato and Guy Shani. High-level reinforcement learning in strategy games. In *Proceedings of AAMAS*, pages 75–82, 2010.

[Balla and Fern, 2009] R. Balla and A. Fern. UCT for tactical assault planning in real-time strategy games. In *21st International Joint Conference on Artificial Intelligence*, 2009.

[Bergsma and Spronck, 2008] Maurice Bergsma and Pieter Spronck. Adaptive spatial reasoning for turn-based strategy games. In *Proceedings of AIIDE*, 2008.

[Billings *et al.*, 1999] Darse Billings, Lourdes Peña Castillo, Jonathan Schaeffer, and Duane Szafron. Using probabilistic knowledge and simulation to play poker. In *16th National Conference on Artificial Intelligence*, pages 697–703, 1999.

[Branavan *et al.*, 2011] S.R.K Branavan, David Silver, and Regina Barzilay. Learning to win by reading manuals in a monte-carlo framework. In *Proceedings of ACL*, 2011.

[Bridle, 1990] John S. Bridle. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. In *Advances in NIPS*, pages 211–217, 1990.

[Coulom, 2006] R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *5th International Conference on Computer and Games*, pages 72–83, 2006.

[de Marneffe *et al.*, 2006] Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. Generating typed dependency parses from phrase structure parses. In *LREC 2006*, 2006.

[Gelly and Silver, 2007] S. Gelly and D. Silver. Combining online and offline learning in UCT. In *17th International Conference on Machine Learning*, pages 273–280, 2007.

[Gelly *et al.*, 2006] S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA, 2006.

[Madeira *et al.*, 2006] Charles Madeira, Vincent Corruble, and Geber Ramalho. Designing a reinforcement learning-based adaptive ai for large-scale strategy games. In *Proceedings of AIIDE*, pages 121–123, 2006.

[Schäfer, 2008] J. Schäfer. The UCT algorithm applied to games with imperfect information. Diploma Thesis. Otto-von-Guericke-Universität Magdeburg, 2008.

[Sheppard, 2002] B. Sheppard. World-championship-caliber Scrabble. *Artificial Intelligence*, 134(1-2):241–275, 2002.

[Silver *et al.*, 2008] D. Silver, R. Sutton, and M. Müller. Sample-based learning and search with permanent and transient memories. In *25th International Conference on Machine Learning*, pages 968–975, 2008.

[Sturtevant, 2008] N. Sturtevant. An analysis of UCT in multi-player games. In *6th International Conference on Computers and Games*, pages 37–49, 2008.

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

[Tesauro and Galperin, 1996] G. Tesauro and G. Galperin. On-line policy improvement using Monte-Carlo search. In *Advances in Neural Information Processing 9*, pages 1068–1074, 1996.

[Tesauro, 1994] G. Tesauro. TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.

[Wender and Watson, 2008] Stefan Wender and Ian Watson. Using reinforcement learning for city site selection in the turn-based strategy game civilization iv. In *Proceedings of CIG*, pages 372–377, 2008.