<u>Situational Reinforcement Learning:</u>

Learning and combining local policies by using heuristic state preference values

S.B. Vrielink

Augustus 2006

In opdracht van:
Universiteit Twente

Opleiding:
Technische Informatica

Leerstoel:
Human Media Interaction

Beoordelingscommissie:

- Mannes Poel
- Anton Nijholt
- Zsofi Ruttkay

Abstract

This document describes an approach to reinforcement learning, called *situational reinforcement learning* (SRL). The main goal of the approach is to reduce the computational cost of learning behaviour in comparison to conventional reinforcement learning. One of the main goals of the research described in this document is to evaluate the implication of situational reinforcement learning on the computational cost of learning behaviour and on the optimality of the learned behaviour. The reduction in computational cost is mainly facilitated by decomposing the environment into smaller environments – called *situations* – and only learn behaviour – called a *policy* – for each situation. A global policy is then created by combining all learned situational policies. Each situation is based upon states that have an equal heuristic preference value. The learned behaviour of a situation will most likely direct the agent to a reachable, more favourable situation. The global policy that is created from combining the situational policies will therefore focus on continually reaching more favourable situations. The research not only evaluates the use of situational reinforcement learning as a stand-alone approach to artificial intelligence (AI) learning, but also applies the approach as an addition to conventional reinforcement learning. The method that uses SRL as a stand-alone approach will be referenced to as the *Combined* method and the method that uses it as an addition to conventional methods will be referenced to as the *Enhanced* method. Evaluation of the *Combined* method shows that the method achieves significant computational cost reductions. Unfortunately, this reduction does not come without a price and the evaluation shows that careful consideration of the heuristic function is required in order to reduce the optimality loss. The evaluation of the *Enhanced* method shows that on average, when using the modified policy iteration algorithm to learn policies, the computational cost of learning a global policy is greater than when the conventional method is solely used. I believe that the significant reduction in computational cost resulting from the use of SRL is a good incentive to perform further research on this approach.

Dit document beschrijft een *reinforcement learning* (RL) methodiek, genaamd *situational reinforcement learning* (SRL). Het hoofddoel van de methodiek is het reduceren van de benodigde berekeningen om gedrag te leren t.o.v. conventioneel RL. Één van de hoofddoelen van het onderzoek omschreven in dit document is om de implicaties van SRL te evalueren op de benodigde berekeningen om gedrag te leren en op de optimaliteit van dit geleerde gedrag. De reductie in berekeningskosten wordt voornamelijk bereikt doordat de methode de omgeving opdeelt in kleinere omgevingen – situaties genaamd – en vervolgens alleen gedrag leert voor elke situatie. Gedrag voor de globale omgeving wordt dan gecreëerd door al het situationele gedrag te combineren. Elke situatie is opgebouwd rond toestanden met gelijke voorkeurswaarden. Het geleerde gedrag binnen een enkele situatie zal de agent waarschijnlijk naar bereikbare situaties leiden met een hogere voorkeurswaarde. Het gecreëerde globale gedrag zal daarom erop gericht zijn om continue situaties te bereiken met een hogere voorkeurswaarde. Het onderzoek richt zich niet alleen op de toepassing van SRL als een alleenstaande methode om gedrag te leren, maar onderzoekt ook of de methodiek als aanvulling kan dienen voor conventioneel RL. De methode die SRL gebruikt als alleenstaande toepassing om gedrag te leren zal de *Combined* methode genoemd worden en de methode die SRL als aanvulling gebruikt zal *Enhanced* heten. De evaluatie van de *Combined* methode toont dat de methode aanzienlijke reducties in berekeningskosten teweeg brengt. Helaas komt die reductie niet zonder prijs en de evaluatie toont ook dat de voorkeurswaarden zorgvuldig gekozen dienen te worden om een groot verlies in optimaliteit te voorkomen. De evaluatie van de *Enhanced* methode toont dat gemiddeld, als modified policy iteration wordt gebruikt als

leer algoritme, de berekeningskosten om globaal gedrag te leren hoger is dan het geval zou zijn als het algoritme op de gebruikelijke manier wordt toegepast. Ik vind dat de significante reductie in berekeningscomplexiteit een goede aanleiding is om verder onderzoek te verrichten naar SRL.

Table of contents

Preface

Now that the research is nearing its end, I first and foremost wish to thank my girlfriend for taking the time to gain an understanding into my research field and providing me with ongoing motivation and criticism. Although from her perspective, the research must have been quite complicated and boring, she was never unwilling to help.

I also wish to thank Mannes Poel, my guidance teacher for the assignment, for continually keeping the research problem manageable and providing me with useful criticism. My first proposed research assignment did not only contain an entirely new approach to game AI, but also included a three-dimensional real-time multiplayer first-person shooter game that employed state-of-the-art graphics. Although the road from that daring plan to the actually performed research was a long one, it was worthwhile and educational.

Finally I wish to thank all those that participated as human players in the method evaluation for their endurance. Although the first few games were always entertaining, the relative simplicity of the game with it's teeth-grinding probabilities led to quick frustrations. I also wish to thank them for the numerous, humorous, although always erroneous, hypothesis about fixed probabilities and whatnot.

It is my hope that I get the opportunity to apply situational reinforcement learning, perhaps in a somewhat modified fashion, to a commercial computer game that I helped develop. This game will then, no doubt, be a commercial break-through… I hope.

Sander Vrielink

Introduction

In the past few years the computer gaming industry has grown considerably. Along with that growth came an increased interest in game aspects that had been previously largely ignored. Traditionally most development focused on the graphical aspect of the game, but in recent years development of the artificial intelligence (AI) in games has seen a significant growth (Darryl, 2003). The few conditional rules and predefined events that controlled most AI behaviour in the past no longer seems to meet the needs of the players. Game AI can be considered a rich field of interesting problems with often large, well defined, partially observable game environments where multiple agents have conflicting or common goals and where actions have stochastic effects. Approaches to AI originally devised to solve problems in game AI can often be fruitfully applied to conventional problems, where g*ame theory* is an excellent example (Russel & Norvig, 2003, pp. 631-641; Morris, 1994). The developed method which is explained and evaluated in this document is also devised from a game perspective, but – as will be shown – is also applicable for conventional problems.

Since game AI has seen increased interest, many different methods for creating or learning AI have been proposed. The AI in most games today still rely in some degree to a form of *finite state machines* (Gill, 1962), which often is a predefined structure that chooses actions based solely on the current state. *Search algorithms* such as A^* (Russel & Norvig, 2003) are also widely used in games, especially for path-finding (Darryl, 2003). Although there are many forms of these two methods which differ in complexity, they are still basically methods where the resulting behaviour is predefined by the developer. Other methods focus more on learning, where behaviour is not predefined but learned through experience or reinforcement. *Evolutionary algorithms* (Bakkes, Spronck & Postma, 2004, 2005) are an example of such methods, where the result of choosing an action in a certain state is evaluated and the action for that state is reconsidered accordingly. The learning process is thus performed through the evaluation of experience. Another example of a learning approach is the *neural network* (Haykin, 1999). Given a training set – which is a set of inputs and corresponding desired outputs for the network – the neural network is 'trained' to generate the desired outputs based on the inputs. If an untrained input is than presented to the network, it is most likely that the network will output a signal that corresponds to the trained input that most closely resembles the given untrained input – a sort of pattern recognition. Through the training set, the neural network learns which outputs to generate based on inputs. The last example of a learning approach to AI – and the approach adopted by the developed method – is *reinforcement learning* (Sutton & Barto, 1998). In reinforcement learning a reward structure is present that assigns rewards based on for example states or actions. The desirability of behaviour is evaluated by the rewards accumulated by that behaviour. Reinforcement learning algorithms focus on learning behaviour that maximize rewards. The approach to reinforcement learning that is developed as part of the research and that is central to the assignment will be called *situational reinforcement learning* (SRL) for reasons that will be explained later on.

Within reinforcement learning there are several ways to learn optimal behaviour. In the context of this document, only reinforcement learning algorithms that are applicable in *Markov Decision Process* (MDP) modelled environments or derivatives thereof will be considered (Russel & Norvig, 2003; Kaelbling, Littman & Cassandra, 1998; Aberdeen, 2003). One form of reinforcement learning is *dynamic programming*. According to Sutton & Barto (1998, chap. 4) "The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process". Multiple dynamic programming algorithms can be used to learn optimal behaviour for an agent, the most notable of which are *value iteration* and *policy iteration* (Mansour & Sing, 1999; Russel & Norvig, 2003; Kaelbling, 1996; Kaelbling et al., 1998; Aberdeen 2003). Other forms of reinforcement learning are the *Monte Carlo methods* (Sutton & Barto, 1998). The difference between Monte Carlo and dynamic programming is that Monte Carlo methods do not require a perfect model of the environment, but use experience gained through interaction or simulation to generate a model of the environment. *Temporal difference learning* (Sutton & Barto, 1998) is a combination of Monte Carlo and dynamic programming and tries to get the best of both. Although the situational reinforcement learning method will only be explained in detail and empirically tested for a dynamic programming algorithm – more precisely a modified version of policy iteration – an explanation will be given on how the method will work for other dynamic programming algorithms and other reinforcement learning techniques.

A problem with most dynamic programming algorithms, such as value- or policy iteration, is that finding the optimal policy – the behaviour that optimally achieves the agent's goal – is a computationally costly operation. For complex environments – and most games fall under that category – finding the optimal policy becomes an intractable problem. The two most commonly used methods of decreasing this complexity are:

- To use simpler computations that approximate the exact computations. This is for example done by the *modified policy iteration* (mPI) algorithm (Russel & Norvig, 2003; Kaelbling, 1996; Kaelbling et al., 1998; Aberdeen, 2003).
- To reduce the environment in which the learning process is performed. For example used by *hierarchical reinforcement learning* (Dietrich, 1999, 2000; Pineau, Gordon & Thun, 2003) and the *envelop method* (Russel & Tash, 1994; Gardiol & Kaelbling, 2004). Situational reinforcement learning also alters the environment in which learning is performed and as such can be seen as an alternative to such methods..

A problem with the use of an MDP modelled environment for games, is that games usually have multiple players with contradicting goals. Because the MDP environment only takes a single action set and reward function into consideration, behaviour of other agents must be modelled as being part of the environment. This considerably increases the difficulty of modelling complex behaviour of other agents. An extension of the MDP framework that tries to solve this problem is the *Markov game* framework (Littman, 1994). In a Markov game modelled environment, each agent has a corresponding action set and reward function which allows for the explicit modelling of multiple agents in the same environment.

The first goal of the assignment is to develop the situational reinforcement learning method. SRL must be applicable in MDP and Markov game modelled environments, must be able to use any dynamic programming algorithm within such environments and be able to learn policies at a lower computational cost than would be the case if the dynamic programming algorithm was applied to the environment without using SRL. Situational reinforcement learning tries to achieve this goal by decomposing the environment into smaller environments – called *situations* – and only perform the learning process for each of these local environments. A policy that spans the global environment is then created by combining all the learned local policies. When considering the goal of SRL – which is to reduce the computational cost by performing the learning process on smaller environments – the method can be seen as an alternative to methods like hierarchical reinforcement learning or the envelope method.

Situational reinforcement learning is inspired by an analogy with how humans play games: Human players often do not have the capacity like computers to foresee a game entirely from beginning to end, but they are still able to rather effectively play complex games. If the player has not foreseen the end, how can he then be rather certain that his move or planned series of moves contribute to reaching a favourable end? Various reasons exists, experience among others, but the feature that SRL is trying to exploit is the human tendency to assign heuristic values to states that indicate preference; although the human player does not see the end, his heuristics tell him that taking a certain piece of the board or making a certain move contributes to a more favourable situation. By continually trying to reach more favourable situations in such a fashion, the human player can play complex games effectively by creating rather short-term plans. A human ability that is not incorporated into SRL is the ability to use experience to alter the heuristics. Within the method, the heuristic function is a static entity given by the developer and any desired changes to this function must be done by the developer.

Quickly said, situational reinforcement learning performs the following operations:

1. Decompose the environment into unique situations. A situation is a subset of the environment which is build around states with an equal preference value according to the heuristic function, called the *inner states* of the situation. Each situation is constructed by SRL in such a fashion that is allows for the previously described human approach to game playing: it contains the states with an equal preference value – the inner states – and states with a different preference value but that are reachable through a single-transition from an inner states. These states are called the *outer states* of a situation and can be seen as goal states for that situation.
2. Learn a policy for each situation.
3. Combine the situation policies to create a policy that spans the original environment.

Within this document, the terms local and global will be frequently used. If local elements are discussed, such as a local policy, this reflects on a situation. If global elements are discussed, such as the global environment, this reflects on the original environment.

A second goal of the assignment is to put SRL into practice for a Markov game modelled environment where games of *Capture the flag* (CTF) can be played. A program that facilitates this goal is written as part of the assignment and appendix F explains this program in more detail. This game environment can then be used as a tool for the third and fourth goal of the assignment: The evaluation of SRL's implication on policy optimality and computational cost. Although the implications of the method are only empirically evaluated for one environment in which one dynamic programming algorithm is used, the results gathered from this evaluation will be used to give indications for other environments and other learning algorithms.

Besides an evaluation of using SRL on its own, a fifth goal is to evaluate the computational cost required for learning an optimal policy by using the resulting global policy of SRL as a starting policy for the modified policy iteration algorithm. This evaluation should give an indication whether SRL has a practical application as an addition to conventional reinforcement learning.

Situational reinforcement learning will be explained in the upcoming chapter: how the reward function can be used as the heuristic function which allows for a decomposition of the environment into situations, how policies can be learned for these local environments and how these policies can be combined to form a global policy. The second chapter goes into various possible applications for SRL: multiple environment modelling techniques and reinforcement learning method will be reviewed and some examples will be given of possible applications for the method. The chapter thereafter gives the method that uses the learned global policy of SRL as a starting policy for modified policy iteration on the global environment. The fourth chapter gives the evaluation method that will be used to evaluate the implications of using situational reinforcement learning. The fifth chapter gives the theoretical evaluation, based on method analysis and worst-case upper-bound complexity functions and the sixth chapter gives the empirical evaluation of the method in which SRL has been applied to the modelled CTF game environment. In the final chapters, conclusions will be drawn, a summary of this document will be given and points for future research will be mentioned. The various appendices give more detailed information on items of interest for the assignment. Within this document, the method that uses SRL as a stand-alone approach to learning behaviour will be referenced to as the *Combined* method. The method that uses the global policy of the *Combined* method as a starting policy for modified policy iteration on the global environment will be referenced to as the *Enhanced* method.

1 Situational reinforcement learning

This chapter explains situational reinforcement learning. The first paragraph gives an introduction to the approach and the second paragraph gives an explanation on it's applicability. In the paragraph thereafter, the method is given on how the environment can be decomposed into situations. After that, an elaboration is given on how local policies can be learned for each of these situations. The final paragraph explains how the local policies can be combined to form a global policy: a policy that spans the original state space.

1.1 An introduction

The inspiration for situational reinforcement learning came from an analogy with how humans play games. Two features that humans use when playing games are key to SRL:

1. Human players are often able to assign heuristic values to states of the game that indicate their overall advantage or disadvantage against the opponent. This allows human players to identify situations, which are sets of states with an equal (dis)advantageous setting, and assign preference to these situations. By trying to reach more favourable situations, which are situations with a higher heuristic value, human players can be rather certain that they are trying to win the game even if they haven't even considered the states that truly end the game. Let's take chess for example: each piece on the board can be assigned a specific value and from the amount of pieces still on the board a value can then be derived for each possible state of the game. Often just by looking at this value, a player can identify his predicament in the game.
2. Human players most often do not try to solve the entire game at once, but rather just try to improve their current situation. This allows human players to play complex games without creating a plan that spans from the beginning to the end. This human tendency can also be exemplified by chess: human players mostly focus their attention on trying to take an important piece of the opponent, instead of immediately thinking on how to manoeuvre the opponent into check-mate.

If a player has a better heuristic function – which enables him to better assess the situations in the game – and is able to plan more situations ahead – enabling him to avoid traps – then this player will probably be the victor in most games.

Conventional reinforcement learning uses a straightforward method: use no heuristic function but only assign rewards to end states and learn a policy for the entire environment at once. Although this approach results in the best possible policy, the problem is that learning an optimal policy in such a fashion for complex environments becomes intractable. To reduce the computational cost, SRL suggests the use of a more complex heuristic function that allows for situation identification. By decomposing the environment into situations and only learn optimal policies for these smaller environments, the computational cost of learning a global policy can be greatly reduced as will be shown in upcoming chapters.

There is no generic method available that can tell whether a heuristic function is correct; most of the heuristic values used in popular games are the result of decades of experience and analysis. In chess for example, the heuristic values assigned to states is almost uniformly accepted. It is the burden of the developer to devise a heuristic function.

The situational reinforcement learning approach performs – simplistically said – the following operations that will be explained in more detail in the upcoming paragraphs:

- Use a heuristic function to identify situations.

- Learn optimal policies for each situation.
- Combine the learned local policies to create a global policy.

The *Combined* method – which is SRL as a stand-alone approach to learning behaviour and is called *Combined* because it combines local policies – has the following problems, which will be elaborated and evaluated in upcoming chapters:

- The heuristic function greatly affects the optimality of the resulting policy, but what is a 'good' heuristic function?
- The reduction in computational cost is the result of learning in smaller environments, but as a result the learned policies are only optimal in their smaller environments, making the combined global policy most likely sub-optimal.

1.2 Method applicability

The *Combined* method is developed to be applicable in MDP- and Markov game modelled environments. This paragraph will give a quick summary of the MDP- and Markov game frameworks, how the heuristic function can be used therein and how this defines the applicability of the method. Appendix A as well as several studies (Littman, 1994; Russel & Norvig, 2003; Kaelbling, 1996; Kaelbling et al., 1998; Aberdeen, 2003) can give additional insight into the MDP- and Markov game framework.

A *Markov Decision Process* is a framework for modelling an environment and can be described by the tuple $\langle S, A, T, R \rangle$, where:

- *S* is a finite set of states of the world.
- *A* is a finite set of actions that can be performed by the agent.
- $T : S \times A \rightarrow \prod(S)$ is the transition function that specifies for an originating state and an action a probability distribution on resulting states. We write $T(s,a,s')$ for the probability that the agents reaches state s', given that the agent performs action *a* in state *s*.
- $R : S \times A \rightarrow \mathbf{R}$ [1] is the reward function that specifies an immediate expected reward if an agent performs an action in a state. We write $R(s,a)$ for the immediate expected reward gained by the agent if he performs action *a* in state *s*.

Summarised, the states in *S* describe the world in which the agent lives. The action set describes the possible actions at the agent's disposal. The transition function describes the dynamics of the world, meaning how the actions of the agent effect the world. The reward function describes the agent's desires. The goal of most AI learning algorithms within an MDP environment is to find the optimal policy, where a policy, $\pi : S \rightarrow A$, maps to each state in the world a single action. As such, a policy describes the behaviour of an agent. Littman (1994) describes an optimal policy in an MDP environment as "In an MDP, an optimal policy is one that maximizes the expected sum of discounted reward and is *undominated*, meaning that there is no state from which any other policy can achieve a better expected sum of discounted reward" (Littman, 1994, p. 2).

A problem with the MDP framework for the modelling of game environments is that the framework only takes a single action set and reward function into consideration, meaning that the behaviour of other agents must be modelled as being part of the environment. This

[1] Also $R : S \rightarrow \mathbf{R}$ and $R : S \times A \times S \rightarrow \mathbf{R}$ can be used, but these create no significant differences according to several studies (Russel & Norvig, 2003; Kaelbling et al., 1998).

considerably increases the difficulty of modelling complex behaviour of the other agents, which is an important aspect for effective game playing. An extension of the MDP framework that tries to solve this problem is the *Markov game* framework. In a Markov game modelled environment, each agent has a corresponding action set and reward function, allowing for the explicit modelling of multiple agents in the same environment. The Markov game framework differs from the MDP framework in the following manner:

- A collection of action sets $A_1, \cdots, A_k$ is given instead of a single actions set *A*. Each agent in the environment has a corresponding action set.
- The transition function *T* now needs to incorporate for each transition an action for each agent: $T : S \times A_1 \times \cdots \times A_k \rightarrow \prod(S)$.
- Instead of a single reward function *R*, each agent has an associated reward function: $R_i : S \times A_1 \times \cdots \times A_k \rightarrow \mathbf{R}$.

The goal of most learning algorithms in a Markov game modelled environment does not differ from the goal in an MDP modelled environment: find the optimal policy. For Markov games, where performance depends critically on the choice of opponents, this goal is somewhat more complex to achieve. Let's review this difficulty by looking at games with simultaneous turn-taking. In such games, each player must choose an action at the same time, meaning that no player knows what the other players are going to do. Because the optimal action of a player depends on the (unknown) actions of all other players, it is impossible to be certain what the optimal action is. Littman (1994) described the solution for this as "In the game theory literature, the resolution to this dilemma is to eliminate the choice and evaluate each policy with respect to the opponent that makes it look the worst" (Littman, 1994, p. 2). Simplistically put, this means that the agent assumes that the opponent is clairvoyant and will always choose the action that is worst in response to the agent's action. The agent thus evaluates each action for the worst possible outcome. This performance measure prefers conservative strategies that result in ties to more daring strategies that can results in great rewards against some opponents and low rewards to others. This is the essence of *minimax*: Behave so as to maximize your reward in the worst case (Littman, 1994).

For the assignment, we will only consider a two player zero-sum[1] Markov game with simultaneous turn-taking, unless stated otherwise, described by $\langle S, A, O, T, R \rangle$, where

- *A* is the action set of the player called the *agent* and *O* is the action set of the player called the *opponent*.
- The transition function becomes $T : S \times A \times O \rightarrow \prod(S)$, and we write $T(s, a, o, s')$ for the probability of ending in state *s'* if the agent takes action *a* and the opponent takes action *o*, both from state *s*.
- Only one reward function can suffice that one agent then tries to maximize while the other tries to minimize it. For the two-player game this becomes $R : S \times A \times O \rightarrow \mathbf{R}$ and we write $R(s, a, o)$ for the expected immediate reward if, from state *s*, the agent takes action *a* and the opponent takes action *o*. The agent tries to maximize the reward function and the opponent tries to minimize it.

As was said in the previous paragraph, the heuristic function that is used by SRL must assign heuristic values to states that represent the preference of the state. The reward function, which

[1] In a zero-sum game, the gain (or loss) of a player is exactly balanced by the losses (or gains) of the opposing player(s). It is so named because when you add up the total gains of the players and subtract the total losses then they will sum to zero.

is already present in MDP and Markov game environments, can be made to serve this goal. The reward function $R(s,a,o)$ gives immediate expected rewards based on states and actions (Kaelbling et al., 1998). Because the heuristic function should only indicate preference based on states, not on actions, SRL assumes a decomposition of the reward function into an action reward function *AR* and a state reward function *SR*:

- $AR : A \times O \to \mathbf{R}$. We write $AR(a,o)$ for the reward if the agent performs action *a* and the opponent performs action *o*.
- $SR : S \to \mathbf{R}$. We write $SR(s)$ for the reward of being in state *s*.

The *SR* function can then be used as the heuristic function that was required for the method. The assumed decomposition of the reward function $R : S \times A \times O \to \mathbf{R}$, which must still give the immediate expected rewards based on states and actions, can become:

- $R(s,a,o) = AR(a,o) + \sum_{s'} T(s,a,o,s') \cdot SR(s')$

Although any arbitrarily complex function could be used since the MDP or Markov game modelled environments do not specify the exact implementation of the reward function. The above mentioned decomposed reward function can be used for a two player zero-sum Markov game, but similar reward functions can be used for MDP environments:

- $R(s,a) = AR(a) + \sum_{s'} T(s,a,s') \cdot SR(s')$

or Markov games with more than two players, where each associated reward function must be decomposable into an action reward function and state reward function:

- $R_i(s,a_1,a_2 \cdots a_n) = AR_i(a_1,a_2 \cdots a_n) + \sum_{s'} T(s,a_1,a_2,\cdots,a_n,s') \cdot SR_i(s')$

The applicability of situational reinforcement learning depends on the environment being modelled. If a decomposition of the reward function(s) into an action reward function and a state reward function is possible, then the environment can be decomposed into situations as is described in the next paragraph and the method is applicable. Games in general are often well suited for such a decomposition because:

- Games are defined by strict rules. These rules allow for clear world dynamics, such as unambiguous probabilities for the stochastic effects of actions, and enables the modelling of most games as discrete[1] environments.
- Within games, the assignment of heuristic preference values to states comes almost naturally. For most games, expert players use their own heuristic values, possibly without consciously doing so. For games which have seen much analysis, numerical value assignment to states are almost uniformly accepted.

For the assignment, we will consider a static[2] discrete two player zero-sum game environment modelled after a game of CTF in which the players take simultaneous actions.

As was said in the previous paragraph, the *Combined* method performs three operations, which will be elaborated on in the upcoming paragraphs:

1) Decompose the global environment into local situations.
2) Learn a policy for each situation.
3) Combine the situation policies.

The applicability of the method depends entirely on the first step. If such a decomposition is possible, which is the case if the reward function can be decomposed, then situations can be created. The second step of the method operation, the learning of policies, is independent

[1] In a discrete environment, the state of the world can be represented by discrete values and a finite set of actions and states are present.

[2] In a static environment, the state of the world can only change through actions of the agent(s).

from SRL; Because each situation is created in such a way that it on itself is an MDP or Markov game environment, each learning algorithm for such environments can be used. The third step, the combining of situation policies, is developed upon situations. As long as policies from situations are being combined, this step can always be performed.

1.3 Decomposition into situations

This paragraph explains the method for decomposing an MDP-like environment into a unique set of situations. Only the two-player zero-sum Markov game environment described by $\langle S, A, O, T, R \rangle$ will be considered, but all MDP- and Markov game environments are decomposable in an analogous manner.

Let Θ be the set of situations. Each situation $\theta \in \Theta$ must be derivable from the entire Markov game environment $\langle S, A, O, T, R \rangle$ and must be a Markov game environment on its own, described by $\langle S_\theta, A_\theta, O_\theta, T_\theta, R_\theta \rangle$. Let's look at what a situation should be able to achieve: The heuristic function should enable a player to identify situations, which are sets of states with an equal (dis)advantageous setting for the player, and by doing so allow the player to restrict his learning to find a way to a more favourable situation. This means that:

- The state set S_θ should consist of all states that have an equal value according to the heuristic function, henceforth called the inner states SI_θ, and all states that have a different value according to the heuristic value but that are reachable by a single transition from the inner states, henceforth called outer states SO_θ. The inner states are the identification of the situation and the outer states are the goal states that enable the learning process to find reachable situations.
- The action sets A_θ and O_θ do not differ from the entire environment because the situations are a subset of the entire world and the available actions in the world do not change. Because the situation no longer consists of all states that were present in the global environment, the effect actions have do change but these dynamics of the world are described by the transition function.
- The transition function T_θ can be seen as having inner transitions and outer transitions. Inner transitions originate from inner states, and these transitions do not differ from the transitions if they were made in the entire environment. Outer transitions originate from outer states, and since these states can be seen as end states of a situations they will become *absorbing states*: states in which each action leads back to the state with a probability of 1.0. So outer transitions always have the same originating and resulting state and these states must be outer states of the game situation.
- The reward function R_θ does not differ from the entire environment.

Let's formalize the above mentioned requirements:

- Θ is the finite set of situations.
- S_θ is a finite set of states of the situation θ.
- A_θ and O_θ are the finite sets of actions that can respectively be performed by the agent and opponent in situation θ.
- $T_\theta : S_\theta \times A_\theta \times O_\theta \to \prod(S_\theta)$ is the transition function for situation θ that specifies for an originating situation state and an action a probability distribution on resulting

situation states. We write $T_\theta(s,a,o,s')$ for the probability that the agents reaches state s', given that the agent performs action a in state s.

- $R_\theta : S_\theta \times A_\theta \times O_\theta \to \mathbf{R}$ is the reward function for situation θ that specifies an immediate expected reward if an agent and opponent perform an action in a state. We write $R_\theta(s,a,o)$ for the immediate expected reward if the agent performs action a and the opponent performs action o in state s.
- $\forall \theta \in \Theta \bullet S_\theta \in S$: Each state set of a situation is a subset of the global state set.
- SI_θ is a finite set of inner states of the situation θ.
- SO_θ is a finite set of outer states of the situation θ.
- $\forall \theta \in \Theta \bullet S_\theta = SI_\theta \cup SO_\theta$: Each state set of a situation is the union of inner states and outer states of that situation.
- $\forall s \in S, \exists! \theta \in \Theta \bullet s \in SI_\theta$: For each state of the global environment a unique situation exists where the state is part of the inner states.
- $\forall s, s' \in S \bullet SR(s) = SR(s') \Rightarrow \exists! \theta \in \Theta \bullet s, s' \in SI_\theta$: If two states have an equal state reward, then there exists a unique situation where both states are part of the inner states.
- $\forall \theta \in \Theta, \forall s \in SI_\theta, \forall a \in A_\theta, \forall o \in O_\theta, \forall s' \in S_\theta \bullet T_\theta(s,a,o,s') = T(s,a,o,s')$: The transition function for each situation equals the transition function for the global environment if the originating state of the transition is an inner state of the situation.
- $\forall \theta \in \Theta, \forall s \in SO_\theta, \forall a \in A_\theta, \forall o \in O_\theta \bullet T_\theta(s,a,o,s) = 1.0$: The transition function for each situation specifies that each transition with an outer state as the originating state has the same outer state as the resulting state.
- $\forall \theta \in \Theta, \forall s \in SI_\theta, \forall a \in A_\theta, \forall o \in O_\theta, \forall s' \in S_\theta \bullet T_\theta(s,a,o,s') > 0 \wedge s' \notin SI_\theta \Rightarrow s' \in SO_\theta$: If a transition in a situation is possible, where an inner state of that situation is the originating state and the resulting state is not an inner state of that situation, then that resulting state is an outer state of the situation.
- $\forall \theta \in \Theta \bullet A_\theta = A \wedge O_\theta = O$: For each situation, the action set of the agent and opponent are the action set of the agent and opponent in the global environment.
- $\forall \theta \in \Theta, \forall s \in S_\theta, \forall a \in A_\theta, \forall o \in O \bullet R_\theta(s,a,o) = R(s,a,o)$: The reward function for each situation equals the reward function for the global environment.

Simplistically said, the decomposition process first identifies the unique state rewards that are present in the environment and then performs the following operations for each state reward, where for each state reward we start from the global environment:

1. Designate the states with the given state reward as being inner states.
2. Remove all transitions that do not originate from inner states.
3. Designate the reachable states which are not inner states as outer states.
4. Remove all states that are not inner or outer states.
5. Add new transitions for the outer states to make them absorbing states.

Figure 1a depicts a simple MDP environment where states are depicted by circles and possible[1] transitions are depicted by arrows. Figure 1b depicts this environment where the inner states of each situation is encircled by a dotted line. Figure 2 shows the results if the

[1] A transition is considered possible if the probability of the transition is greater than 0.

above 5 step process is used for all situations, where the inner states of a situation are still encircled by a dotted line.

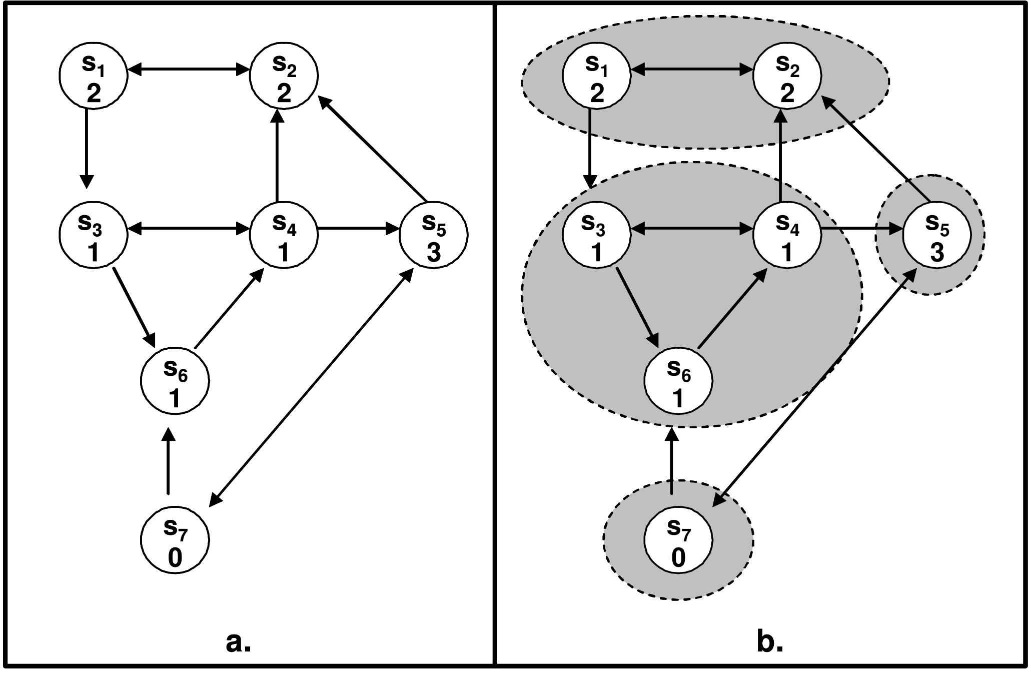


Figure 1a. example MDP environment with states, state rewards and transitions
b. The inner states of each situation encircled by a dotted line. These are <u>not</u> yet situations.

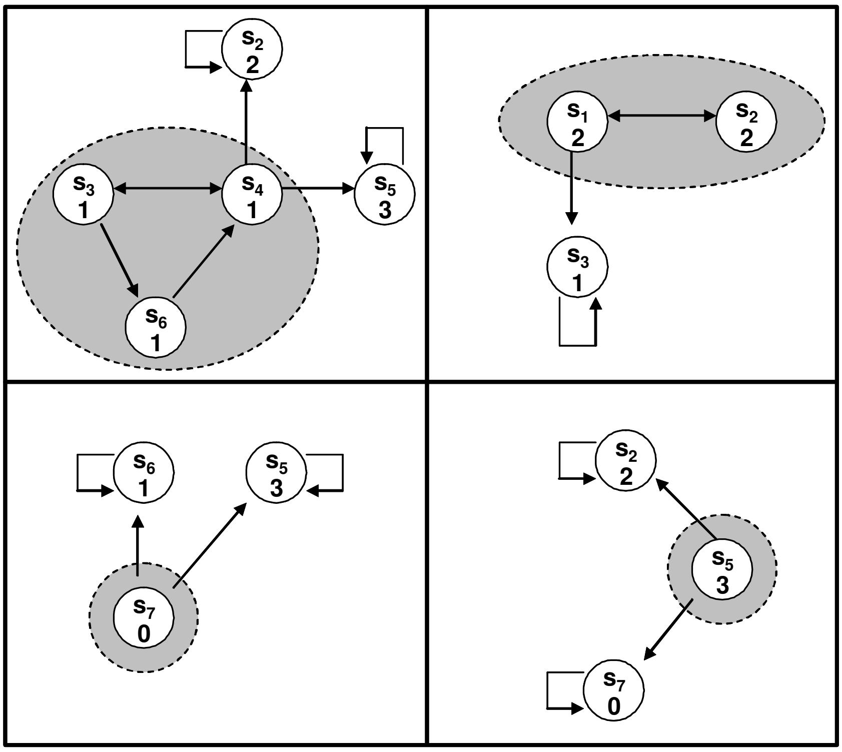


Figure 2. The four situations derived from figure 1, where inner states are encircled.

1.4 Learning local policies

Now that the process of creating situations has been explained, we turn towards the process of learning policies for these situations. A local policy π_θ is the policy belonging to situation θ that maps a single action to every state of that situation: $\pi_\theta : S_\theta \rightarrow A_\theta$. Because each situation is an MDP or Markov game environment on it's own, any policy learning algorithm suited for such environments can be used. The next chapter explains how situational reinforcement learning can be used in conjunction with different reinforcement learning algorithms.

Because the situation is created in such a fashion that each inner state has an equal state reward and the outer states are goal states, the learned policy for a game situation will most likely direct the agent to outer states with high rewards, if an outer state exists that has a higher state reward than the inner states. This conforms to a goal of the method, where the learning process should only focus on reaching more favourable situations. More on this will be explained in the evaluation.

The empirical evaluation of the *Combined* method will focus on the use of a single dynamic programming learning algorithm: modified policy iteration. The (modified) policy iteration algorithm is explained in detail in appendix B for MDP and Markov game environments, but a small introduction to the algorithm will be given here for an MDP environment. The modified policy iteration algorithms is – as it's name suggests – a modified version of the policy iteration (PI) algorithm. The method used by the policy iteration algorithm is to start off with a random policy and continually improve this policy until the optimal policy has been found. Each iteration of the PI algorithm consists of two phases: policy evaluation and policy improvement. In the policy evaluation phase, the utility of each state is recalculated by using the current policy. In the policy improvement phase, these new utility values are used to improve the policy. Let π_i be the policy after *i* iterations of PI, then the new utility of a state under policy π_i, $U_{\pi_i}(s)$, is calculated in the policy evaluation phase by solving the following equation:

$$\textbf{(1)}\ U_{\pi_i}(s) = R(s, \pi_i(s)) + \gamma \cdot \sum_{s'} T(s, \pi_i(s), s') \cdot U_{\pi_i}(s')$$

Using the new utility values, the policy can be improved by using a one-step greedy look-ahead function with respect to utility: choose the action that has the highest expected utility gain:

$$\textbf{(2)}\ \pi_{i+1}(s) = \max_a \left[R(s, a) + \gamma \cdot \sum_{s'} T(s, a, s') \cdot U_{\pi_i}(s') \right]$$

This process of policy evaluation and policy improvement is repeated until no change occurs to the policy, $\pi_{i+1} = \pi_i$. If this is the case, then the policy iteration algorithm guarantees the optimal policy is found (Kaelbling, 1996).

A problem with the previously described policy iteration algorithm, is that the computational cost of solving the linear equations in the policy evaluation phase given by **(1)** is high. For that reason, modified policy iteration was created. The idea behind modified PI, is that it might not be required to calculate the utility of each state exactly in the policy evaluation phase, but that an approximation to this exact value might yield the same results. Modified policy iteration acquires this approximation by keeping the policy fixed for *k* successive executions of the policy evaluation phase, meaning that the policy evaluation can be given by:

$$\textbf{(3)}\ U_{\pi_i}(s) \overset{k}{\leftarrow} R(s, \pi_i(s)) + \gamma \cdot \sum_{s'} T(s, \pi_i(s), s') \cdot U_{\pi_{i-1}}(s')$$

Appendix B explains what **(3)** entails in more detail. It can be shown that if *k* reaches infinity, the calculated utility values of **(3)** equals the utility values if the PI algorithm was performed, as given by **(1)**, meaning that the modified PI algorithm perfectly approximates the PI algorithm (Woodward, 2006). A problem is to find the *k* value that guarantees that level of approximation. If a *k* value is chosen to perfectly approximate the PI algorithm, then modified PI can make the same guarantees about optimality as PI. If on the other hand a *k* value is chosen that does not perfectly approximate the PI algorithm, then it is possible that a sub-optimal policy is found if the modified PI uses the same termination criteria as PI. Appendix E goes into this in more detail.

1.5 Combining local policies

The agent is now able to learn policies for each situation and the policies belonging to these situations can easily be combined to form a policy that spans the entire environment. The global policy π is created by using for each state *s*, the action specified by local policy π_θ for state *s*, where *s* is an inner state of situation θ:

- $\forall s \in S, \forall \theta \in \Theta \bullet s \in SI_\theta \Rightarrow \pi(s) = \pi_\theta(s)$

Because each state is inner state of one and only one situation, the created global policy has an action specified for each state of the entire environment. By creating the global policy in this fashion, the global policy most likely directs the agents to increasingly favourable situations, as will be explained further in the upcoming evaluation chapters.

2 Various SRL applications

In the previous chapter, the *Combined* method has been explained for a two player zero-sum Markov game and it is such an environment that is explained in appendix D and that will be used for the empirical evaluation explained in paragraph 3.2. This chapter will extend the method application to various other domains. In the first paragraph, the method application to different MDP-like environments will be elaborated. In the paragraph thereafter, something will be said about using dynamic programming algorithms. Using other reinforcement learning methods, such as *Monte Carlo* and *Temporal Difference Learning*, will be explained in the third paragraph. The final paragraph will give some examples on how the method can be applied to practical problems other than the empirically evaluated one described in appendix D.

2.1 Environments

In this paragraph a quick explanation will be given on how the method can also be applied to Markov game environments with more than two players and to partially observable MDPs (POMDPs).

The difference between a two player zero-sum Markov game and a Markov game with more than two players is that in the latter case each agent must have a corresponding reward function $R_i(s, a_1, \cdots, a_n)$. As was said in paragraph 1.2, the *Combined* method is applicable if each reward function is decomposable into an action reward and a state reward function $R_i(s, a_1, a_2 \cdots a_n) = AR_i(a_1, a_2 \cdots a_n) + \sum_{s'} T(s, a_1, a_2, \cdots, a_n, s') \cdot SR_i(s')$. When such a decomposition is possible it is also possible for any agent to decompose the environment into situations in the fashion described in paragraph 1.3. Depending on the various state reward functions, it is possible that the decomposition results in a different set of situations for each agent. Let Θ_i be the set of situations resulting from a decomposition using the state reward function SR_i, then Θ_i should be used for learning a policy for agent a_i. Because the resulting set of situations are all autonomous MDP-like environments, it is possible to learn policies for these situations in a Markov game manner.

Now let's look at POMDP environments. No detailed explanation will be given here on POMDP environments, but the papers from Kaelbling, Littman & Cassandra (1998) and Aberdeen (2003) can provide insight into the environment and solution methods for such an environment. The only difference between a MDP and POMDP environment is the fact that a POMDP environment is partially observable instead of fully observable. This means that in a POMDP environment, the agent is not certain about the state of the world, which increases the difficulty of learning optimal behaviour greatly. A POMDP modelled environment is described by the tuple $\langle S, A, T, R, \Omega, Obs \rangle$[1], where according to Kaelbling (Kaelbling et al., 1998, p. 8):

- *S*, *A*, *T* and *R* describe a Markov decision process.
- Ω is a finite set of observations the agent can experience of its world.
- $Obs : S \times A \rightarrow \prod(\Omega)$ is the observation function, which gives, for each action and resulting state, a probability distribution over possible observations (we write

[1] In most literature, the set of observations is given by *O* instead of *Obs*, but in order to avoid confusion with the opponent action set *O* used in the two player zero-sum Markov game we will use *Obs* here.

$Obs(s',a,o)$ for the probability of making observation *o* given that the agent took action *a* and landed in state *s'*.

Because a POMDP environment is only partially observable, the agent uses an internal *belief state b* that summarises its previous experience. A belief state is a probability distribution over states of the world. Because the agent does not know what the state of the world is in a partially observable environment, the agent assigns probability to states that represent the agent's belief that he is in that state: this is called the belief state.

In order to make the *Combined* method compatible with POMDP environment, the following is specified:

- Each situation $\theta \in \Theta$ is described by a tuple $\langle S_\theta, A_\theta, T_\theta, R_\theta, \Omega_\theta, Obs_\theta \rangle$.
- S_θ, A_θ, T_θ and R_θ are derived from the entire environment in exactly the same manner as described in paragraph 1.3 for an MDP environment.
- $\forall \theta \in \Theta \bullet \Omega_\theta = \Omega$: The observation set Ω_θ of each environment does not differ from the complete observation set Ω.
- The observation function for situation θ becomes $Obs_\theta : S_\theta \times A_\theta \rightarrow \prod(\Omega_\theta)$.
- $\forall \theta \in \Theta, \forall s' \in S_\theta, \forall a \in A_\theta, \forall o \in \Omega_\theta \bullet Obs_\theta(s',a,o) = Obs(s',a,o)$: The observation function for each situation equals the observation function for the entire environment for each state that is part of the situation.
- A belief state b_θ for situation θ is a probability distribution over states in situation θ.

By creating situations in this manner, all created situations are autonomous POMDP environments, just like it was in the MDP setting. Because all situations are POMDP environments, all learning methods for such environments can be used. Since a policy for an MDP environment is identical to a policy for a POMDP environment, the combination process described in paragraph 1.5 does not change.

2.2 Dynamic programming algorithms

Dynamic programming algorithms such as value iteration, policy iteration or modified policy iteration are all reinforcement learning algorithms that require a complete model of the environment in order to learn an optimal policy. Because the situations that are created as a result of using the *Combined* method are autonomous complete-model environments of their own, all dynamic programming algorithms can be used in conjunction with the *Combined* method.

2.3 Other reinforcement learning methods

Apart from the dynamic programming algorithms, which require a complete model of the environment in order to learn an optimal policy, several other reinforcement learning methods exist that do not require a complete model. Examples of such methods are the *Monte Carlo* methods, *Temporal Difference Learning* and *Q-learning*. Let's consider how the *Combined* method can be applied if only an incomplete model of the environment is available.

To apply the decomposition process of the *Combined* method in the manner described in paragraph 1.3, the method requires the following:

- The state set must be available, so that it is known which states are available to divide into situations.
- The action set must be available so that it is known which actions are available for the transitions.

- The reward function must be available so that is possible to define the inner states of each situation.
- The transition function must be available so that it is possible to define the outer states of each situation.

This means that, in order for the *Combined* method to be applied in the same manner described in chapter 1, the method requires a complete model of the environment. Because of this, the *Combined* method is in it's current form inapplicable for reinforcement learning methods that use an incomplete model of the environment.

Although it lies beyond the scope of the assignment to formally specify how the *Combined* method can be altered to become compatible with these incomplete-model methods, a short informal description of a possible way can be given. When speaking of an incomplete model of the environment, it is most often the transition function that is incomplete: the probability distribution on transitions is unknown. The other elements of the environment – the state set, action set and reward function for an MDP environment – are usually complete. If these other elements are known, it is still possible to define the amount of situations and the inner states of each situation. It would then still be possible to learn a policy for a situation by defining all states that are not inner states as outer states – which are as absorbing states. Although this would mean that the state set for each situation is still the global state set, with the only difference being the division into inner- and outer states, the computational cost of learning a policy for such a situation is still reduced in comparison to the global environment because for each situation behaviour is only learned for inner states – the outer states are absorbing states where every action has the same result – and most of the outer states will never be reached. In this fashion it is still possible to learn local policies by using any of the before-mentioned reinforcement learning algorithms.

2.4 Example applications

In this paragraph three examples will be given on how situational reinforcement learning could be applied. The first example is a brief summary of the CTF environment which is used for the empirical evaluation and fully specified in appendix D. The other examples are not fully specified and many elements are omitted for ease of understanding. These examples serve only to give insight into possible method applications. The second example illustrates how SRL could be used in a first-person shooter game and in the last example an explanation is given how SRL could be used for a conventional problem – meaning an environment that is not modelled after a game.

2.4.1 Capture the flag

The environment which is modelled after a game of CTF and specified in appendix D has the following features:

- A fully-observable environment modelled as a Markov game.
- Stochastic actions.
- Two agents with contradicting goals.
- A state set consisting of 136737 states.
- An action set consisting of 8 actions.
- Turn-based action handling with simultaneous actions.

In the CTF environment, the goal of the players is to score a pre-defined amount of points before the opponent does so by taking the flag of the opposing team and returning that flag to a specific location. By using situational reinforcement learning and the reward structure mentioned in appendix D, the learning environment does no longer encompass the entire environment – which has a large state set that results in high computational costs when learning behaviour – but is decomposed into 21 smaller and more tractable learning environments. The empirical evaluation that is explained in paragraph 4.2 uses this CTF environment as a practical application for SRL. The results given in chapter 6 show that SRL facilitated a significant reduction in computational cost in comparison to conventional reinforcement learning when learning behaviour for this CTF environment.

2.4.2 A first-person shooter

Let's consider how situational reinforcement learning could be employed for a computer controlled entity that patrols and guards a specific area in a first-person shooter (FPS) games – a reasonable playground for artificial intelligence, and one far more complicated and realistic than the environment used for the empirical evaluation. The goal of the agent is to guard an area and prevent a human player from reaching the exit in that area. Figure 3 gives a top view of this area.

This area is the agent's world. The area contains some walls, a lower and higher ground, three entry points where the human player could enter the area and one exit that the human player needs to reach. This environment can be modelled as an MDP, Markov game or POMDP environment in a similar fashion as was done in appendix D for a two player zero-sum Markov game. Although it is far more difficult to model the environment illustrated in figure 3 than it is to model the environment in appendix D, it can still be done (A first-person shooter environment will most likely require a far greater state set with time-indexed states and would require some form of real-time instead of turn-based action handling, which therefore lies beyond the scope of the assignment). Because it is possible to model the FPS environment of figure 3 as an MDP-like environment, it is also possible to use SRL to learn a policy for that environment in the manner described in chapter 1 and paragraph 2.1.

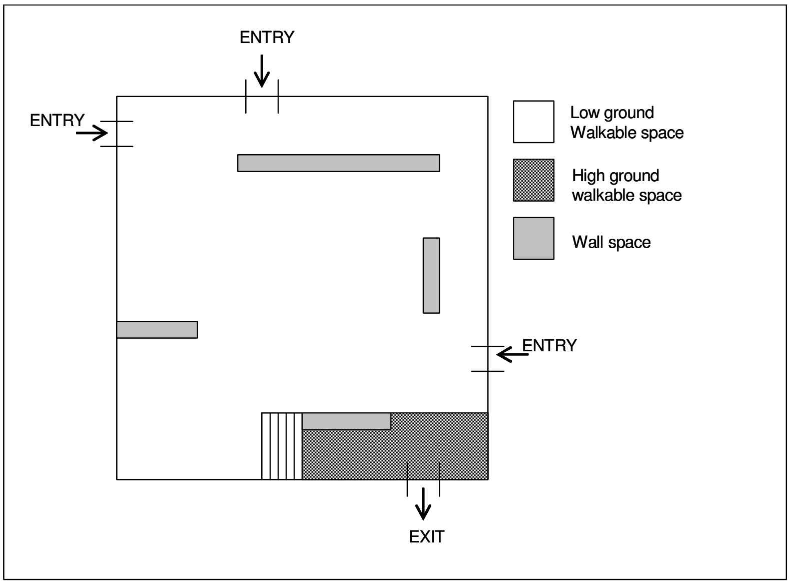


Figure 3. A top view of the example FPS world

Let's for example assume that the state reward distribution in the modelled environment of figure 3 corresponds with the following rules, where numeric preference values for the described circumstances are given in brackets:

- The agent being alive is more preferable (+0) to the agent being dead (-2).

- The human player being dead is more preferable (+2) to the human player being alive in the area (+0) which in turn is more preferable to the human player being alive at the exit (-2).
- It is more preferable (+1) for the agent to occupy higher ground than the human player than it is to be an equal ground (+0), which in turn is preferable to being on lower ground (-1).
- It is more preferable (+1) to be in a shooting position where the agent has partial cover from walls than it is to be in an uncovered shooting position (+0).

By using such a reward structure, the agent is still able to learn seemingly 'intelligent' behaviour for his area – such as: reaching a covered shooting position if the probability of reaching the covered position alive is acceptable; directly attacking the human player from an uncovered position if it isn't probable that a covered position can be reached alive; preferring high ground over low ground; avoid getting killed or letting the human player reach the exit – whilst avoiding a learning process on the entire environment. The learning of behaviour that seems intelligent to the human player at a (probable) lower computational cost than when using conventional methods was the main motivation for developing SRL. The agent behaviour can be enhanced to include team-play by modelling multiple agents with common goals in a Markov game manner or to include 'unpredictable' behaviour by using the *stochastic policy* described by Littman (Littman, 1994).

2.4.3 The taxi domain

In a paper where a method for hierarchical reinforcement learning is explained, an example environment is used to illustrate the method's workings (Dietrich, 1999, p. 9). This so called "Taxi Domain" will be used to exemplify the use of SRL in a non-game environment. Figure 4 depicts this taxi domain: a 5x5 grid world inhabited by a taxi agent with 4 distinct locations (R, G, B and Y).

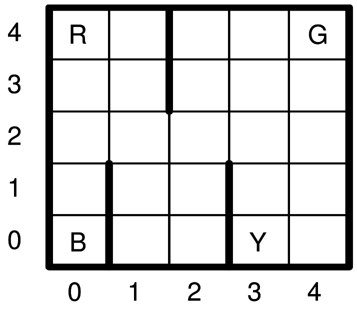


Figure 4. The Taxi Domain

There is a passenger at one of the four locations that wishes to be transported to another location. The taxi agent should move to the passenger, pick him up, drive him to the desired location and drop him off. The action set for the agent contains navigational actions, a pick-up action and a drop-off action. Based on this problem, the following very straight-forward way of distributing state rewards would be:

- +0 for all states where the passenger is not in the car and not at the location[1].
- +1 for all states where the passenger is in the car.
- +2 for all states where the passenger is not in the car and at the location he wished to be.

[1] In the Taxi domain problem described by Dietrich (1999, p. 9) it was possible for the passenger to start at the location where he wished to be. We ignore that scenario in this example.

Using situational reinforcement learning with the abovementioned state rewards would result in a decomposition of the environment into three situations. Because the situation ordering of this problem is so unambiguous[1], the resulting policy is most likely optimal. The taxi domain example shows that SRL is capable of reaching an optimal policy at reduced computational cost in comparison to conventionally using dynamic programming algorithms.

[1] There is only one possible transition between the first and second situation: when a pick-up is performed with taxi and passenger at the same location. There is also only one transition between the second and third situation: when a put-down is performed when the taxi with passenger is at the desired location.

3 Enhancing the global policy

Besides an evaluation of what the implications are on policy optimality and computational cost when using situational reinforcement learning as a stand-alone approach to learning behaviour, an alternative use for SRL will also be researched where SRL is used as an addition to conventional reinforcement learning. Because the method of learning local policies will most likely result in sub-optimal global policies, as will be discussed in upcoming chapters, the application of SRL as a starting point for learning a global optimal policy might be worthwhile.

The method of learning this global optimal policy used as part of the assignment is very straightforward: take the result of the *Combined* method and use this as a starting point for conventional reinforcement learning. This method will be referenced to as the *Enhanced* method in the remainder of this document, because the policy is enhanced to become more optimal.

What resulting information from the *Combined* method is required for the *Enhanced* method to resume learning depends on the reinforcement learning approach employed by the *Enhanced* method. If the *Enhanced* method uses for example value iteration or policy iteration, the *Combined* method's resulting global policy alone is sufficient. The value- or policy iteration algorithm can then use this policy as a starting policy for learning. For the empirical evaluation discussed in later chapters, the *Enhanced* method will use the modified policy iteration algorithm. We will not explain the algorithm in detail here, appendix B gives a detailed elaborated on how (modified) policy iteration is used in MDP and Markov game environments, but the policy evaluation phase of each learning iteration is given by **(3)** as being:

$$U_{\pi_i}(s) \overset{k}{\leftarrow} R(s,\pi_i(s)) + \gamma \cdot \sum_{s'} T(s,\pi_i(s),s') \cdot U_{\pi_{i-1}}(s').$$

As can be seen, the utility values that are being calculated by using the current policy U_{π_i} requires the utility values that were calculated by using the previous policy $U_{\pi_{i-1}}$. In order for the *Enhanced* method to resume learning by using the modified policy iteration algorithm, the global policy created by the *Combined* method alone is insufficient, since we also require the utility values that were used to learn that policy. This global utility set can be created by the *Combined* method in a similar fashion as the global policy was created: combine the local utility sets by taking from each situation the utility values for the inner states. The global utility set U_π is created by using for each state *s*, the utility value specified by local utility set U_{π_θ} for state *s*, where *s* is an inner state of situation θ:

- $\forall s \in S, \forall \theta \in \Theta \bullet s \in SI_\theta \Rightarrow U_\pi(s) = U_{\pi_\theta}(s)$

In the empirical evaluation, the modified policy iteration algorithm will use this created global utility set along with the created global policy to resume learning. The implications of using the *Enhanced* method will only be empirically evaluated for the modified policy iteration algorithm.

4 Evaluation method

This chapter elaborates on how the evaluation of the *Combined* and *Enhanced* methods will be performed. Evaluation will focus on two items:

1. The optimality of the methods resulting global policies. Because the goal of the *Enhanced* method was to learn an optimal policy, no policy optimality evaluation will be performed for that method.
2. The computational cost of executing the methods.

In the remainder of this document, the terms *learning method* and *learning algorithm* will be used frequently. In the context of this assignment, these terms entail the following:

- A learning algorithm is an algorithmic, usually mathematical, approach to learning a policy. The parameters for the algorithm are set beforehand. Examples of learning algorithms are policy iteration, modified policy iteration and value iteration.
- A learning method is a method that defined how a learning algorithm is used. It sets the parameters for the learning algorithm and employs the algorithm to learn policies. The *Combined*, *Enhanced* and hierarchical reinforcement learning methods are examples of learning methods and the evaluation will introduce the additional learning method *Complete*.

The *Combined* and *Enhanced* methods do not specify which learning algorithm should be used to learn the local policies – and it has been shown that any dynamic programming algorithm can be used – but this algorithm does influence the computational cost of learning[1] the global policies. The evaluation will therefore use one policy learning algorithm continually, unless stated otherwise: the modified policy iteration algorithm explained in paragraph 1.4 and appendix B. The results gained from the evaluation will be used as an indication for other similar learning algorithms.

The evaluation will occur theoretically and empirically in the fashion described in the upcoming paragraphs.

4.1 Theoretical evaluation

The theoretical policy optimality evaluation of the *Combined* method will occur by analysing the structure of the method. The evaluation will focus in particular on how the heuristic function, which defines the structure of the situations, affects policy optimality.

The theoretical computational cost evaluation will be performed through a worst-case computational complexity analysis of required calculations when applying the *Combined* method to MDP- and Markov game environments.

In the theoretic evaluation, a comparison will also be drawn between the *Combined* method and two methods that also reduce computational cost by learning on smaller environments: *hierarchical reinforcement learning* and the *envelop method*. Although this evaluation does not give insight into the optimality or computational implications of situational reinforcement learning, it does give insight into differences and similarities with other methods.

[1] The global policy resulting from the *Combined* method is actually a created policy and not a learned policy. Only the local policies are learned, the global policy is created from these local policies.

4.2 Empirical evaluation

The empirical evaluation is the main method of evaluation used as part of the assignment. The empirical evaluation will occur by applying the *Combined* and *Enhanced* methods to a modelled two player zero-sum Markov game environments in which games of capture the flag can be played. Appendix D gives a detailed explanation of the modelled CTF world. Computational cost evaluations will be performed by monitoring the process of policy learning and calculating a computational cost value which indicates the required amount of calculations. Policy optimality evaluations will be performed by letting the learned global policies of various methods play against each other in games of CTF.

Further along this paragraph, the terms *policy optimality* and *game performance* will be used. Let's explain the meaning these words have in the context of the assignment:

- Each policy has a certain degree op optimality. The *policy optimality* defines how well the policy achieves it's goal. The most common goal for policies is to maximize the expected reward through it's actions. With this goal, each action of the optimal policy would yields the highest possible expected reward. In the context of the two player zero-sum CTF game, the optimal agent policy maximizes expected rewards and the optimal opponent policy minimizes expected rewards.
- *Game performance* is a measurement on how a player fares in playing a game against another player. How game performance is measured as part of the assignment is explained further along this paragraph.

For the evaluation of policy optimality we will not evaluate this optimality directly, but rather evaluate the game performance of the various learned policies. If the reward function represents the player's desire to win the game, then a more optimal policy will most likely[1] win more games. Although game performance cannot be used as an exact measurement of policy optimality, it can give an indication. As part of the empirical evaluation, an indication on the policy optimality implication of the various methods will be given by monitoring the game performance of policies learned by these methods competing against each other in games of CTF.

4.2.1 The learning methods

The *Combined* and *Enhanced* methods that have been described in previous chapters will be referenced to as learning methods. For the empirical evaluation both methods will use the modified policy iteration algorithm – which is explained in paragraph 1.4 and appendix B – for learning policies. In order to evaluate the implications of these methods, the results of learning global policies with these methods must be compared to learning a global policy without such a method. This reference learning method uses the modified policy iteration in it's most basic way: learn a policy for the global environment by using modified policy iteration and start from a random policy. This method will be referenced to as the *Complete* method, because it learns a policy for the complete environment[2].

Summarised, the empirical evaluation on the effect of the learning method will use the following three different learning method which differ in the following manner:

[1] A more optimal policy does not necessarily win more game even if the reward function correctly represents the desire to win, because the element of chance plays a vital role in most games. This is also the case for the modelled CTF game.

[2] The *Enhanced* method also learns a policy for the entire environment, but it is called *Enhanced* nonetheless because of the fact that it enhances the *Combined* global policy.

1. The *Complete* method learns a global policy by using modified policy iteration algorithm on the global environment and starting from a random policy. This method can be considered as a conventional reinforcement learning approach.
2. The *Combined* method learns local policies for the situations by using modified policy iteration on each situation. The modified policy iteration algorithms start with random policies. The learned local policies are then combined to form a global policy.
3. The *Enhanced* method learns a global policy by using modified policy iteration on the global environment and starting from the global policy and utility set that were created by the *Combined* method.

4.2.2 The heuristic function

Besides an evaluation of the learning methods in general, an evaluation will also be performed on how the use of a heuristic function on itself affects policy optimality and computational cost of the learning method.

The CTF world that is described in appendix D will be referenced to as the *Standard* environment, because the reward/heuristic function used therein was the first one devised and tested. But does that reward structure correctly represents the agent's desire to win the game? In order to evaluate the effect of the reward- and heuristic function, multiple environments will be used in the empirical evaluation.

In the most basic view a game can end in three ways: a tie, a win or a lose. The reward structure that resembles this basic view only assigns rewards to states that represents these endings. The use of such a reward structure is unambiguous and correct, and such reward structures are commonly used in game AI. The use of a more complex reward function could distract a playing agent from winning the game. Since the *Combined* method requires a more complex reward structure to be able to identify multiple situations, an evaluation will be performed on how the use of such a more complex reward structure affects optimality and computational cost. For this evaluation, an environment will be used in which the reward structure only assigns non-zero rewards to states that end the game. This environment will be called the *Simple* environment, because it uses a simple reward function.

Although the reward structure used in the *Standard* environment seems correct, since higher rewards are only assigned for states that represents 'better' situations for the player, the performance of the *Combined* method in this environment was unsatisfactory. Analysis of the resulting *Combined* global policies revealed an unforeseen problem. To evaluate how different reward/heuristic functions can affect the optimality and complexity of the *Combined* method, even if those reward structures do not assign illogical rewards, a third environment is modelled in which the observed problem with the *Standard* environment is avoided. This environment is called the *Alternative* environment, because it is an alternative to the *Standard* environment.

Summarised, the empirical evaluation on the effects of the heuristic function will use the following three environments which differs in the following way:

1. The *Simple* environment uses a reward function that only assigns non-zero rewards to states in which a player is victorious: +10 for agent win states and -10 for opponent win states. Because a decomposition of the *Simple* environment into situations would

only result in one situation where policies are actually learned[1], the *Combined* and *Enhanced* learning methods are not applied to the *Simple* environment.

2. The *Standard* environment uses the reward function that is specified in appendix D for the CTF world.
3. The *Alternative* environment also uses the reward function specified in appendix D, with one difference: no rewards are assigned for the 'Dead' values of the *AS* and *OS* state variables (see appendix D for more details).

4.2.3 The policy learning algorithm

As part of the evaluation, all learning methods will use the modified policy iteration algorithm that is explained in paragraph 1.4 and appendix B. In order to evaluate the implication of the learning methods on computational cost and global policy optimality, the learning algorithms must be able to learn policies of an equal degree of optimality, preferably the optimal policy, no matter the environment in which is learned. More practically said, if the globally learned policies of the *Complete* and *Enhanced* methods are optimal then in order for a correct comparison to the *Combined* global policy, the locally learned policies of the *Combined* method should also be optimal. If we should use different optimality criteria for learned policies, then any difference in global policy optimality and corresponding computational cost could be the result of these different criteria and not the solely the result of using the *Combined* method.

In order to assure an equal degree of policy optimality, the variables of the modified policy iteration algorithm will be set in such a manner that we may assume that the optimal policy has been learned. The modified policy iteration algorithm uses two variables that are not defined by the environment, but must be set by the developer: the discount factor γ and the approximation variable *k*. Appendix E gives a detailed explanation of these variables and how they will be set as part of the assignment to assume an optimal policy. In appendix E, a new variable is also introduced: the termination value *t*. A quick explanation will now be given of these three variables and how they are set for each environment in the evaluation:

- The discount factor γ has a value between 0.0 and 1.0 and defines the weight of future rewards. Each discount factor creates a different optimal policy for the environment, so an optimal policy that is learned with a discount factor of 0.4 does not necessarily equal an optimal policy that is learned with a discount factor of 0.5. Higher discount factors means that, from the perspective of a single state, rewards of states that lie many transitions away have a greater influence. Because in games it is often better to think as far ahead as possible, a higher discount factor will in general result in policies that have a better game performance. As explained in appendix E, it is also true in generals that a higher discount factors requires more iterations to find the optimal policy, meaning that a higher discount factor results in higher computational costs if the environment is held constant. Because it can thus be said that in general the effect of a higher discount factor results in better game performance and higher computational costs, three discount factors will be used for the evaluation that should give an indication for all discount factors. The evaluation will therefore use a low discount factor of 0.1, a high discount factor of 0.9 and a discount factor in between of 0.5.
- The variable *k* of the modified policy iteration algorithm has a value of 1 or greater and defines how many times the policy remains fixed in the policy evaluation phase,

[1] The other situations only contain end states of the game, meaning that all states are absorbing states.

as explained in paragraph 1.4 and appendix B. The evaluation phase of modified PI solves the following equation:

$$U_{\pi_i}(s) \overset{k}{\leftarrow} R(s,\pi_i(s)) + \gamma \cdot \sum\nolimits_{s'} T(s,\pi_i(s),s') \cdot U_{\pi_{i-1}}(s')$$

Which can be rewritten as the following equations:

$$U_{\pi_i}(s) = U^{k}_{\pi_i}(s) | k \geq 1$$

$$U^{j}_{\pi_i}(s) = R(s,\pi_i(s)) + \gamma \cdot \sum\nolimits_{s'} T(s,\pi_i(s),s') \cdot U^{j-1}_{\pi_i}(s') | 1 \leq j \leq k$$

$$U^{0}_{\pi_i}(s) = R(s,\pi_i(s)) + \gamma \cdot \sum\nolimits_{s'} T(s,\pi_i(s),s') \cdot U_{\pi_{i-1}}(s')$$

If the value of *k* is chosen high enough, the policy evaluation phase of the modified PI yields the same result as PI would have done. In a sense, it could thus be said that *k* defines the degree of approximation that mPI has on PI. If the value of *k* is chosen high enough to make a perfect approximation of PI, then the mPI algorithm can make the same guarantees of optimality as the PI algorithm. Unfortunately, there is no exact method available that defines this value for *k* based on an environment. Because the learning environments differ between the *Combined* method and the *Complete*/*Enhanced* methods, an equal value of *k* would be unfair. For the evaluation, the minimal value of 1 is chosen for *k*.

- The termination value *t*, which is introduced in appendix E, defines the amount of unchanged policies in the policy improvement phase of policy iteration that are required for termination of the algorithm. In PI, this value is 1 and if the PI algorithm terminates after an unchanging policy in the policy improvement phase, the PI algorithm guarantees that the optimal policy is found (Kaelbling, 1996). If the previously discussed approximation value *k* in modified PI is chosen in such a fashion that leads to less then perfect approximations, it is possible that a sub-optimal policy is found if this same termination value of 1 is used for modified PI. By increasing the amount of required successive unchanging policies in the policy improvement phase, the modified PI algorithm can be enables to find the optimal policy, even if the *k* value is chosen too low. Just as with the *k* value, there is unfortunately no method available that defines a correct value of *t*. For the evaluation, the *t* values are chosen in such a fashion that we assume that the optimal policy is found, as is explained in appendix E.

Summarized, all learning methods used in the evaluation utilize the modified policy iteration algorithm to learn policies. For the evaluation, three different discount factors will be used $\gamma \in \{0.1;0.5;0.9\}$. The approximation value *k* will be set to 1 and for each learned policy the *t* value was set in such a manner that we may assume that the optimal policy was found.

4.2.4 Evaluating general performance

Although the learning methods are evaluated against each other and the global policies of the *Complete* and *Enhanced* methods are assumed optimal, it does not yet give an indication on how these policies perform in general. All learned policies are derived from a Markov game model of the CTF game, which uses *minimax* to learn policies: Behave so as to maximize your reward in the worst case (Littman, 1994). For the modelled CTF game, which has simultaneous turn-taking, this means that each agent action is evaluated against the opponent action that makes the agent action looks the worst, $\max_a \min_o [\cdots]$, which can also be seen in the modified policy iteration equations of appendix B. This way of acquiring rewards favors conservative strategies that lead to ties over daring strategies that might lead to a losing situation. Although this is one possible approach of dealing with opponent choices, it is not necessarily the best approach. To evaluate how the learned policies perform in general, two non-learning methods will also be implemented against which the learned policies must also play games of CTF:

1. A random policy creating method, referenced to as the *Random* method. The *Random* method can be considered to be the worst possible CTF player if we assume that all players intend to win the game.
2. The *Human* method, where human players must play games of CTF against policies learned by the various learning methods.

4.2.5 Evaluating computational cost

The computational cost of learning global policies when using the various learning methods will empirically be evaluated by monitoring the learning process of those methods. Appendix C explains in more detail how the complexity functions for MDP and Markov game environments are derived when using (modified) policy iteration. Unlike the theoretical evaluation of complexities, which performs a worst-case analysis for situational reinforcement learning independent of environment and policy learning algorithm by using the 'order of' $O(\cdots)$ notation, the empirical evaluation uses computational cost functions *C* that give an indication on the amount of calculations required on average to learn policies specifically for the modelled CTF game world with the modified policy iteration algorithm. The cost values *C* do not provide an exact amount, but rather an approximation to the amount of required calculations. A quick summary of these value functions *C*, which are explained in more detail in appendix C, will be given here.

The largest amount of calculations required for learning a policy for the global CTF game world – whichever of the previously mentioned learning methods are used – lie in the execution of the modified policy iteration algorithm. Although the *Combined* method also requires a decomposition of the environment into situations and a combination of local policies, the computational cost of these two operations are minute in comparison to the policy learning processes required for each situation[1]. Therefore the computational cost of executing these peripheral operations are ignored and only the computational cost of executing the modified policy iteration algorithm is considered.

Within the modified policy iteration algorithm, the computational cost of learning a global policy depends on two factors: the amount of iterations required to learn the optimal policy and the computational cost of a single iteration. For the empirical evaluation of computational cost, we are interested in the average cost required to learn a global policy. The computational cost of learning a global policy for a certain environment is not constant because the amount of iterations required to learn a policy with (modified) policy iteration depends on the initial policy, and this policy is random for both the *Complete* and *Combined* methods. Therefore an average amount of required iterations, i_{avg}, is used to calculate the cost values. This average amount of iterations will be derived from monitoring the learning process. The computational cost of a single iteration can be derived from the equations used in the policy evaluation and policy improvement phases of the modified policy iteration algorithm, which are explained in

[1] For a computation time comparison on one test system: decomposition of the environment into situations required around 7 seconds, the combining of local policies a few microseconds and the learning of a single policy takes between 3 and 30 minutes.

appendix B and are derivations of **(1)** and **(2)** mentioned in paragraph 1.4. These equations from appendix B are repeated here[1]:

$$\textbf{(4)}\ U_{\pi_i}(s) \overset{k}{\leftarrow} \min_{o}\left[R(s,\pi_i(s),o) + \gamma \cdot \sum\nolimits_{s'} T(s,\pi_i(s),o,s') \cdot U_{\pi_{i-1}}(s')\right]$$

$$\textbf{(5)}\ \pi_{i+1}(s) = \max_{a}\min_{o}\left[R(s,a,o) + \gamma \cdot \sum\nolimits_{s'} T(s,a,o,s') \cdot U_{\pi_i}(s')\right]$$

For the policy improvement phase, which uses equation **(4)**, a calculation must be performed for each state *s*. Each of these calculations must be performed *k* times and within each of these calculations all opponent actions *o* must be evaluated once. Within the equation, a summation must also be performed on all possibly resulting states when performing actions *a* and *o*. For the CTF world, there is a maximal amount of possibly resulting states of 8 when a player returns from the dead back into the game, but for almost all action-pairs there are but one or two possible resulting states. Because this amount is so small, we will for the CTF world ignore the fact that the summation takes multiple steps and view the entire calculation between brackets in **(4)** and **(5)** as a single calculation. The amount of calculations required to perform **(4)** in a single iteration can thus be described as being $n \cdot k \cdot o$, where $n = |S|$. In a similar fashion, the computational cost of **(5)** can be described as being $n \cdot a \cdot o$. By also introducing the average amount of required iterations, an indication of the average amount of calculations required to learn an optimal policy for an environment can be given by[2]:

$$\textbf{(6)}\ C = i_{avg} \cdot (n \cdot k \cdot o + n \cdot a \cdot o) = (a + k) \cdot (i_{avg} \cdot o \cdot n)$$

For the empirical evaluation of computational cost, we will not use the empirically found i_{avg} directly, but rather a corrected amount $\hat{i}_{avg}$. This corrected amount $\hat{i}_{avg}$ is the found i_{avg} decremented by the *t* value that was discussed in the previous paragraph and in appendix E. i_{avg} is decremented by the *t* value because we are interested in the amount of iterations required to learn an optimal policy, not in the amount of iterations required to be certain that we have acquired the optimal policy. Because the *t* values were chosen in such a manner that we may assume that the optimal policy has been found, the corrected amount of iterations $\hat{i}_{avg}$ represents for each environment the average minimal amount of iterations required to learn the optimal policy.

Because we are interested in the difference in average complexity values between the methods, all variables from **(6)** that remain the same <u>between</u> learning methods can be eliminated. By removing these variables *a*, *o* and *k* and also introducing the corrected amount $\hat{i}_{avg}$, **(6)** can be written as:

$$\textbf{(7)}\ C = \hat{i}_{avg} \cdot n$$

These last two variables $\hat{i}_{avg}$ and *n* may not be removed, because they differ between learning methods. If **(7)** is applied to the three methods, the following equations for average computational cost are derived, and these are used as part of the empirical evaluation:

$$\textbf{(8)}\ C_{complete} = \hat{i}_{avg,complete} \cdot n$$

[1] These equations apply to the utility maximizing agent only. For a utility minimizing agent, the variables *a* and *o* must be switched in the equation, but for the CTF game world this makes no difference since the size of *A* and *O* are equal.

[2] There is a hidden correlation between the amount of iterations and the state set size. When starting from a random policy, it is most likely that more iterations are required for larger environments.

$$\textbf{(9)}\ C_{combined} = \sum_{\theta\in\Theta}\left(\hat{i}_{avg,\theta}\cdot n_\theta\right)$$

$$\textbf{(10)}\ C_{enhanced} = C_{combined} + \left(\hat{i}_{avg,enhanced}\cdot n\right)$$

Within these three equations, the only true unknown is $\hat{i}_{avg}$ because the values of *n* and all n_θ are fixed by the CTF environment. Because the computational cost values are indicating values for comparison and not 'order of' $O(\cdots)$ notations as used in the theoretical evaluation of worst-case complexities, the constant values may not be removed. In words, they serve to indicate the cost of a single iteration of modified PI. A single iteration of modified PI has a greater cost in a larger environment, because the calculations must be performed for more states.

A *player* in the modelled CTF game world is identified by a learning method, a learning environment and a discount factor. For each unique player, 20 agent policies and 20 opponent policies will be learned, resulting in 40 policies for each player.

4.2.6 Evaluating policy optimality

The empirical evaluation of policy optimality will occur by monitoring game performance of the learned policies and using performance values as an indication of optimality. In this evaluation, we will assume that more optimal policies win more games because the used reward structures represent the desire to win the game. Game performance will be measured by letting the learned policies of the various methods play games of CTF against each other and the non-learning methods and analysing the results.

In each game, there are two players: an agent and an opponent. As is explained in appendix D, a game of CTF can end in three ways:

1. At least one of the players scored the maximum amount of points.
2. Both players choose to do nothing in two consecutive turns.
3. A to be defined amount of time has expired.

From these endings, the evaluation will identify five possible results of a single game:

1. The agent wins by being the first player to score the second point.
2. The opponent wins by being the first player to score the second point.
3. The game is a tie if both players score their second point simultaneously.
4. The game is a tie if a 'Deadlock' state is reached. Appendix D can be referenced for more details, but a 'Deadlock' state is reached if both players are alive and perform the *DoNothing* action for two consecutive turns.
5. The game is invalid if a predefined amount of time has expired. These games are declared invalid because time isn't explicitly modelled in the environment, and as such the policies haven't learned that this is a possible ending of the game. Therefore it would be unfair to force a result from these states and as such these games are declared invalid. Games in which the time has expired are usually games where the agent and opponent have reached a repetitive pattern of movement, must like a 'Deadlock' state that spans multiple states. Because the implementation of a pattern recogniser lies beyond the scope of the assignment, these games are considered invalid and will not be used in the evaluation of optimality.

Game performance will be measured by a performance value P_{p_1,p_2} and it represents the results of games played between players p_1 and p_2. A player that uses a policy learning method is identified by three variables: The learning method used (*Complete*, *Combined* or *Enhanced*), the environment in which is learned (*Standard*, *Simple* or *Alternative*), and the discount factor used (0.1, 0.5 or 0.9). A player that uses a non-learning method is identified by the used method (*Random* or *Human*). The performance value P_{p_1,p_2} represents the percentage of games more won by player p_1 against player p_2, and as such is a value between -100 and 100. A performance value P_{p_1,p_2} will be calculated for each pair of competing players as given by:

$$\textbf{(11)}\quad \begin{gathered} P_{p_1,p_2} = \frac{W_{p_1} - W_{p_2}}{Total} \cdot 100 \\ Total = W_{p_1} + W_{p_2} + T \end{gathered}$$

Where W_{p_1} is the amount of games won by player p_1, W_{p_2} is the amount of games won by player p_2 and T is the amount of games that ended in a tie. Optimality evaluation will occur by comparing these performance values.

5 Theoretical evaluation

This chapter gives the theoretical evaluation of the *Combined* and *Enhanced* methods. The first paragraph gives an evaluation of the implication that the methods have on global policy optimality and the second paragraph gives an evaluation on the computational complexity for learning the global policies. The last paragraph will give a comparison between the *Combined* method and other methods from the literature that also reduce complexity by learning on smaller environments.

5.1 Policy optimality

The optimality of the learned local policies of the *Combined* method depends on the used learning algorithms. Since most learning algorithms have as a goal to learn an optimal policy, we will only consider these optimal policies. Because the local policies are only optimal for their respective situations, the global policy that is created from combining them no longer has to be – and most likely won't be – optimal. What degree of optimality the global policy does have depends on the quality of the heuristic function and on the environment.

Let's take a look at the structure of situations: Each situations is organised in such a fashion that it consist of states that can be considered to be equally (dis)advantageous for the agent (inner states) and states that lie just outside the situation, but are inner states of another situation (outer states). Because all inner states have an equal state reward[1], there is little room for utility improvement within the situation for the agent. The outer states on the other hand are absorbing states with a different state reward. If the state reward of such an outer state is greater than the state reward of an inner state, reaching this outer state (and thus another situation) would probably lead to higher reward gains. This is especially so since, from the perspective of the local learning process, the outer state is an absorbing state which the agent never leaves. Because of this, the resulting global policy can be seen as a policy that continually tries to reach situation with a higher state reward, a very human approach to game playing. A danger with this method is that the policy can fall into a trap by short-term rewards: a local policy only takes single-transition reachable states of other situations into consideration, it does not see the rewards beyond that state. How decisive this danger is for the global optimality depends on the environment. Let's view each situation as a sub-goal of the environment; If the ordering of the sub-goals in the environment is unambiguous, and the difficulty of the environment is to find the optimal solution for each sub-goal, then the *Combined* method would probably provide a near-optimal global policy. If on the other hand there are a host of sub-goals and the difficulty of the environment is to find the optimal sequence of sub-goals, then the optimality of the resulting global policy is most likely disastrous. Chess is an example of a game in which situational reinforcement learning would most likely perform terrible, because in chess the taking of each piece can be seen as a sub-goal and the difficulty is not to take a piece, but to take the right piece at the right time. Capture the flag would be an example of a game in which SRL will likely perform better, because an unambiguous ordering of sub-goals could be to first take the flag and then return the flag, and the difficulty there would be how to best realise the taking and returning of the flag. As part of the assignment, the policy optimality will be empirically tested for the modelled CTF Markov game described in appendix D.

[1] In the *Combined* method, the state reward function is used as the heuristic function and all inner states of a situation have the same value according to the heuristic function.

5.2 Computational complexity

In this paragraph, worst-case complexity functions will be used for the theoretical evaluation of the computational complexity of the various learning methods. Appendix C gives detailed information about the upper bound worst-case complexity functions for the (modified) policy iteration algorithm in MDP- and two player zero-sum Markov game environments. All upper-bound complexity functions given in this chapter are worst-case complexities.

5.2.1 Standard policy iteration in an MDP environment

According to Kaelbling (1996, p. 15) and as explained in appendix C, the upper bound complexity of a single iteration of the policy iteration algorithm in an MDP environment is given by:

$$\textbf{(12)}\ O\left(a \cdot n^2 + n^3\right)$$

According to Mansour & Sing (1999, p. 2), the upper bound amount of iterations required to learn an optimal policy in an MDP environment when a greedy policy-iteration algorithm is used – and a greedy PI algorithm is used for the evaluation of SRL – can be given by:

$$\textbf{(13)}\ O(n)$$

The upper bound complexity of learning an optimal policy for an MDP environment by using standard policy iteration thus becomes, by combining **(12)** and **(13):**

$$\textbf{(14)}\ O\left(a \cdot n^3 + n^4\right)$$

(14) gives the worst-case complexity for the *Complete* method, so **(14)** can be rewritten as being:

$$\textbf{(15)}\ O_{complete}^{MDP}\left(a \cdot n^3 + n^4\right)$$

The upper bound complexity of the *Combined* method, is the sum of the complexities of learning policies for all situations, as given by:

$$\textbf{(16)}\ O_{combined}^{MDP}\left(\sum\nolimits_{\theta\in\Theta} O_{\theta}^{MDP}\right)$$

$$\textbf{(17)}\ O_{\theta}^{MDP}\left(a \cdot n_{\theta}^3 + n_{\theta}^4\right)$$

Now how does **(16)** compare to **(15)**? Let's begin by stating that Θ is the set of situations and $g = |\Theta|$ is the total amount of situations. How both complexity functions relates to each other depends on the amount of situations and the amount of states within each situation. If the entire environment is one situation then $O_{combined}^{MDP}\left(\sum\nolimits_{\theta\in\Theta} O_{\theta}^{MDP}\right) = O_{combined}^{MDP}\left(a \cdot n_{\theta}^3 + n_{\theta}^4\right) = O_{complete}^{MDP}\left(a \cdot n^3 + n^4\right)$. Since the complexity of **(17)** is greatest for the situation with the most states (highest n_{θ}), it is that situation that gives the greatest addition in **(16)**. The most favourable case for **(16)**, meaning the case with the lowest worst-case complexity, is where each situation has the least amount of states, meaning that all states in *S* are divided evenly among the situations in Θ. If this is the case – which is hardly ever – then the amount of states in each game situation is given by:

$$\textbf{(18)}\ |S_{\theta}| = n_{\theta} = \frac{n}{g}$$

Combining **(18)**,**(17)** and **(16)** results in:

$$\textbf{(19)}\ \dot{O}_{combined}^{MDP}\left(g \cdot \left(\left(a \cdot \frac{n^3}{g^3}\right) + \left(\frac{n^4}{g^4}\right)\right)\right) = \dot{O}_{combined}^{MDP}\left(\frac{a \cdot n^3}{g^2} + \frac{n^4}{g^3}\right)$$

As can be seen, this upper bound complexity is almost a factor g^3 smaller than the upper bound case for the *Complete* case given by **(15)**. Let's make it a little simpler: For most games, the world contains a very large amount of states and a relative small set of available actions (in the modelled CTF world for example there are around 150000 states and 8 possible actions). With *n* being such a large number and $a << n$, we can approximate **(15)** and **(17)** as being

$$\textbf{(20)}\ O^{MDP}_{complete}\left(a \cdot n^3 + n^4\right) \approx \hat{O}^{MDP}_{complete}\left(n^4\right)$$

$$\textbf{(21)}\ O^{MDP}_{\theta}\left(a \cdot n^3_{\theta} + n^4_{\theta}\right) \approx \hat{O}^{MDP}_{\theta}\left(n^4_{\theta}\right)$$

Using these complexities, **(19)** becomes:

$$\textbf{(22)}\ \dot{\hat{O}}^{MDP}_{combined}\left(g \cdot \frac{n^4}{g^4}\right) = \dot{\hat{O}}^{MDP}_{combined}\left(\frac{n^4}{g^3}\right) = \dot{\hat{O}}^{MDP}_{combined}\left(\frac{\hat{O}^{MDP}_{complete}\left(n^4\right)}{g^3}\right)$$

So, in the upper-bound case with the most favourable division of states within the situations, the complexity is approximately reduced by a factor g^3. This means that, for the worst-case in an MDP environment, **(15)** and **(16)** relate to each other in the following way:

$$\textbf{(23)}\ \frac{\hat{O}^{MDP}_{complete}\left(n^4\right)}{g^3} \le O^{MDP}_{combined}\left(\sum\nolimits_{\theta \in \Theta} O^{MDP}_{\theta}\right) \le O^{MDP}_{complete}\left(a \cdot n^3 + n^4\right)$$

In practical applications, the worst-case upper-bound case is almost never. How substantial this complexity reduction is for the modelled CTF world will be evaluated as part of the assignment.

5.2.2 Modified policy iteration in an MDP environment

Let perform the same steps as before for the modified policy iteration algorithm. The upper bound complexity of learning an optimal policy by using the modified policy iteration algorithm is, as explained in appendix C, given by:

$$\textbf{(24)}\ \tilde{O}\left(n \cdot \left(k \cdot n^2 + a \cdot n^2\right)\right) = \tilde{O}\left(\left(a + k\right) \cdot n^3\right)$$

The complexities for learning an optimal policy for respectively the global environment and a situation can thus be given by:

$$\textbf{(25)}\ \tilde{O}^{MDP}_{complete}\left(\left(a + k\right) \cdot n^3\right)$$

$$\textbf{(26)}\ \tilde{O}^{MDP}_{\theta}\left(\left(a + k\right) \cdot n^3_{\theta}\right)$$

In the most favourable case for the *Combined* method, where the states are divided evenly among the situations, the complexity can be given by:

$$\textbf{(27)}\ \dot{\tilde{O}}^{MDP}_{combined}\left(g \cdot \left(\left(a + k\right) \cdot \frac{n^3}{g^3}\right)\right) = \dot{\tilde{O}}^{MDP}_{combined}\left(\frac{\left(a + k\right) \cdot n^3}{g^2}\right) = \dot{\tilde{O}}^{MDP}_{combined}\left(\frac{\tilde{O}^{MDP}_{complete}\left(\left(a + k\right) \cdot n^3\right)}{g^2}\right)$$

So, when using the modified policy iteration, the complexity functions for the *Complete* and *Combined* method relate to each other in the following manner:

$$\textbf{(28)}\ \frac{\tilde{O}^{MDP}_{complete}\left(\left(a + k\right) \cdot n^3\right)}{g^2} \le \tilde{O}^{MDP}_{combined}\left(\sum\nolimits_{\theta \in \Theta} \tilde{O}^{MDP}_{\theta}\right) \le \tilde{O}^{MDP}_{complete}\left(\left(a + k\right) \cdot n^3\right)$$

5.2.3 Using the Markov game environment

Although the modelling of an environment as a Markov game environment does increase the complexity of learning policies[1], it does not effect the complexity reduction brought about by situational reinforcement learning. This is so because the reduction in complexity brought about by SRL is the result of the smaller state spaces; the method does not alter the use of the action sets. The Markov game environment uses multiple action sets, which increases the complexity of learning policies, but does not alter the use of the state set in policy learning. If we alter **(12)** to incorporate a second action set *O*, where $o = |O|$, the complexity would become:

$$\textbf{(29)}\ O\left(a \cdot o \cdot n^2 + o \cdot n^3\right)$$

If we performed all the previous steps for this complexity, the resulting optimal complexity reduction would still be g^3. This is also the case for even more action sets.

5.2.4 Using other policy learning algorithms

The reduction in complexity brought about by SRL is the result of using the smaller environments. The complexity reduction is directly linked to the weight of the state set size *n* in the complexity function of the used learning algorithms: the complexity reduction is greater if the state set size *n* has a greater influence in the complexity function of that learning algorithm. The reason that the complexity reduction is greater for the policy iteration algorithm than for the modified policy iteration algorithm, $g^3 > g^2$, is because the state set size *n* is of a higher order in the complexity function of the policy iteration algorithm, $O_{PI}\left(n^4\right) > O_{mPI}\left(n^3\right)$.

5.2.5 The Enhanced method

Because the methodology of the *Enhanced* method is to first perform the *Combined* method and then the *Complete* method – with the difference that the *Combined* global policy is used as a starting policy instead of a random policy – the upper bound complexity of performing the *Enhanced* method can be given by the addition of the upper bound complexities of both other methods:

$$\textbf{(30)}\ O_{enhanced}\left(O_{combined}(\cdots) + O_{complete}(\cdots)\right)$$

So, in the upper bound view the *Enhanced* method it is only a more costly method of acquiring a global optimal policy. But is this also the case for the practical application of the method? A hypothesis that will be tested for the modelled CTF world is that learning a policy for the entire environment when starting from the combined policy will take less iterations to terminate then when policy iteration starts from a random policy. This hypothesis is based on the fact that policy iteration gradually improves a policy until the optimal policy is found, so if you start with a more optimal policy it seems logical that you require less iterations to reach the desired degree of optimality. An assumption that will be made, is that the global policy resulting from the *Combined* method is more optimal than a random policy. This assumption is based on the fact that the local policies are at least optimal in their own local environments.

If the hypothesis holds, then the learning process for the entire environment when starting from the *Combined* policy would have a lower complexity than the learning process would

[1] The complexity for learning a policy in a Markov game environment roughly increases by a factor that equals the product of the sizes of the action sets. If for example *m* action sets are used where each set has a size of *o* actions, the complexity for a single iteration of standard policy iteration in that Markov game would be $O\left(o^m \cdot n^2 + o^{m-1} \cdot n^3\right)$, which is roughly is an increase in complexity of o^m.

have for the entire environment when starting from a random policy, if we don't take the complexity of learning the local policies into consideration. Because we have already demonstrated in the previous paragraphs that learning a global policy by using the *Combined* method will most likely have a lower computational cost than learning a policy for the entire environment, the question becomes whether the combination of learning local policies and enhancing the resulting combined policy has a lower computational cost than policy iteration for the entire environment from a random policy? This question will be answered in the empirical evaluation in the modelled CTF world of computational costs.

5.3 Comparison to similar methods

In this paragraph, the *Combined* method will be compared to two methods that also try to reduce computational complexity by learning in smaller environments: The *Envelope Method* (Russel & Tash, 1994; Gardiol & Kaelbling, 2004) and *Hierarchical Reinforcement Learning* (Dietrich, 1999, 2000; Pineau et al., 2003).

5.3.1 The Envelope Method

The general idea of the Envelope method is that is learning an optimal policy for the global environment is not required for good performance. Instead, the global optimal policy can be approximated by only learning an optimal policy for the states that are likely to be reached. In the Envelope method, the MDP environment for which a policy is learned is called the *envelope*, and this envelope is a sub-MDP of the global environment. Besides the required elements of an MDP environment, the envelope method required an initial world state and a method to define what rewards are assigned to states that fall outside the envelope.

An abstract view of the process of the envelope method is depicted in figure 5.

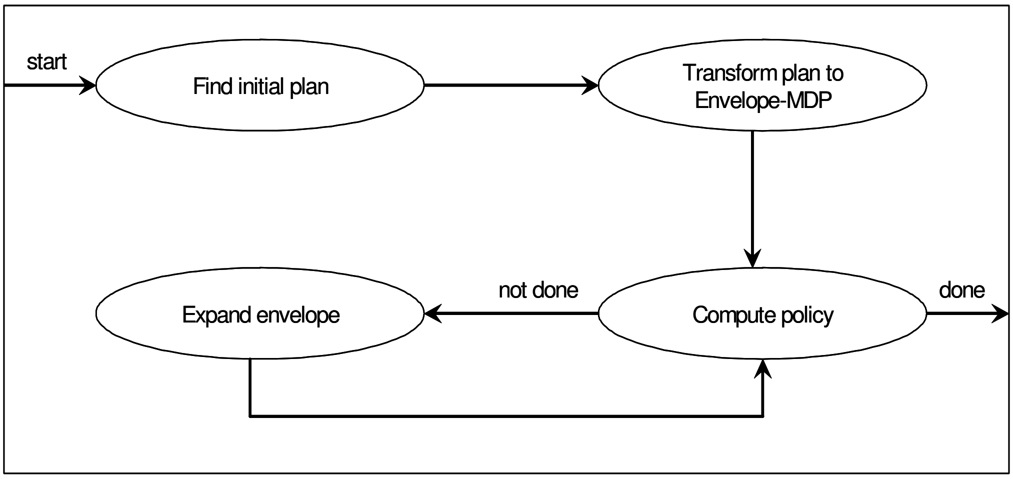


Figure 5. Abstract view of the Envelope method

The envelope method begins by finding an initial plan. In this classical planning problem, the method tries to find a series of actions that lead from initial state(s) to goal state(s). The initial states must be provided by the developer and the goal states can either be goal states of the environment or states that lie at a maximum planning depth from the initial state.

When the initial plan has been constructed, this plan is transformed to the initial envelope MDP. First, the envelope is initialized with the initial word state; then, the next state in the envelop is found by applying the plan action to the previous state; when the state containing the goal condition is reached, the set of states is complete. Transitions that initiate in an envelope state but do not land in an envelope state are redirected to a state called the *out-of-envelope* or *Out* state (Gardiol & Kaelbling, 2004).

With a new envelope constructed, a (new) policy for this envelope must be computed. This can be performed with any learning algorithm for the MDP environment, the only difference is that the rewards for states that lie outside the envelope must be assigned by a special reward function. This special reward function can for example assigns a penalty which equals an estimation of the cost of having to recover from falling out (such as having to re-plan back to the envelope) (Gardiol & Kaelbling, 2004).

After a policy has been computed, the envelope must be expanded. This phase is also called *deliberation*. Gardiol & Kaelbling (2004) described this phase as "The deliberation involves sampling from the current policy to estimate which fringe-states – states one step outside of the envelope – are likely to be reached. (…). The (…) most likely fringe states are added to the envelope." (Gardiol & Kaelbling, 2004, p. 6). When to stop deliberating depends on the criteria the developer uses; one example could be to stop deliberating when a maximum amount of fringe states have been added. With a new envelope created, a new policy is computed. This process of policy computation and envelope expansion is repeated until some termination criteria is met; for example when a fixed amount of deliberations have passed.

The differences between the Envelope method and the *Combined* method are:

- The Envelope method learns only one policy for a continually expanding envelope-MDP. The *Combined* methods learns multiple policies for static situation-MDPs.
- The Envelope method learns an optimal policy for states that are most likely to be reached, starting from an initial state, but ignores all other states for the policy. The *Combined* method learns optimal policies for situations, where situations are created based on heuristic state preference values. In the *Combined* method, no state is ignored.
- The Envelope method requires much more alteration to the MDP framework than the *Combined* method. The Envelop method requires an initial state, an initial plan (which is a planning process, not reinforcement learning), a special reward function for out-of-envelope penalties, a termination criteria when to stop a round of deliberation and a termination criteria when to stop the method itself. The *Combined* method only requires a decomposition of the reward function into a (heuristic) state reward function and an action reward function.

Both methods result in global policies that approximate the global optimal policy. The optimality of the global Envelope policy depends for a large part on the initial plan, the optimality of the global *Combined* policy depends for a large part on the heuristic function.

A possibly fruitful conjunction of both methods could be to use the envelope-expansion method on situations: Select a situation to be the initial envelope and expand the envelope with the most likely or preferable reachable situation until an end state of the game has been reached. This conjunction would allow the *Combined* method to learn further than a situation, reducing the danger of falling into traps, and would eliminate the need of initial plans, out-of-envelope penalties and special deliberation- and method termination criteria that are required for the Envelope method.

5.3.2 Hierarchical Reinforcement Learning

The general idea of Hierarchical Reinforcement Learning (HRL) is that the structure of an environment can be used to limit the amount of policies that need to be considered as well as enables the use of state abstraction. In HRL, the global MDP is decomposed into a hierarchy of smaller MDPs. The method is based on the assumption that the developer can identify useful sub-goals and defined subtasks that achieve these sub-goals (Dietrich, 1999). This discussion of HRL is based on the MAXQ Value Function Decomposition, but the discussed characteristics apply to all HRL methods.

In order to employ hierarchical reinforcement learning, the developer must identify individual subtasks that he believes are important for solving the overall task. There are various methods that specify how subtasks should be constructed. Several of these are:

- Define each subtask in terms of a fixed policy that is provided by the developer.
- Define each subtask in terms of a non-deterministic finite-state controller.
- Define each subtask in terms of a termination predicate and a local reward function. The MAXQ HRL method uses this definition.

If we consider the third method for specifying subtasks then, for each subtask, the termination states for that subtask must be defined along with the actions or other subtasks that it employs to reach its goal and a local reward function. For each sub-task an optimal policy π_i can then be learned. The hierarchical policy π, is a set containing a policy for each of the subtasks: $\pi = \{\pi_o, \cdots, \pi_n\}$. The execution of a hierarchical policy then consists of identifying the current subtask and perform the action specified in the corresponding subtask-policy for the current state.

By dividing the environment into these hierarchical blocks, each subtask-policy only needs to consider actions that are relevant for performing it's task, which eases the policy learning process. But a sub-task policy must be learned for each task, and each action of action set *A* must be employed by at least one sub-task or the action was unnecessary for the environment in the first place. The state space *S* does not change for each subtask, besides the fact that some states are considered termination states for the subtask. Because the amount of states in an environment has a greater influence in computational complexity than the amount of actions for most learning algorithms, as can be seen in appendix C, it is probable the learning a policy for each subtask involves more computations than learning a policy for the entire environment. Because of this, hierarchical reinforcement learning often makes use of state abstraction. With state abstraction, certain aspects of the state space that are irrelevant for solving the subtask are ignored. By doing this, the state space of each subtask is reduced to a subset of *S*, which reduces the computational complexity for learning a policy for that subtask even more. Dietrich (1999) states that "Perhaps the most important reason for introducing hierarchical reinforcement learning is to create opportunities for state abstraction" (Dietrich, 1999, p. 27).

The hierarchical policy that is the result of hierarchical reinforcement learning no longer has to be the global optimal policy. The optimality of the hierarchical policy critically depends on the hierarchical structure, since this structure defines which policies will be considered. Hierarchical reinforcement learning therefore tries to reach hierarchical optimality: A hierarchically optimal policy for MDP *M* is a policy that achieves the highest cumulative reward among all policies consistent with the given hierarchy (Dietrich, 1999).

The differences between HRL and the *Combined* method are:

- The HRL uses a pre-defined hierarchical structure. Within the HRL method, the problem is described as a single task which is continually decomposed into more primitive sub-tasks, where each lower-level sub-tasks can be used to achieve the goal of a higher level sub-task. Each sub-task is thus a part of a higher level task and has a specific goal. In the *Combined* method each situation represents a unique part of the original environment and no situation is part of another. Each situation has the same goal: maximize rewards. The difference is that situations are autonomous environments in which the local learning processes are unaware of the other situations, where subtasks on the other hand employ other subtasks to solve specific problems.
- The HRL views the world as a problem that must be solved. In order to solve the problem, the developer defines subtasks that solve partial-problems. In this fashion, the HRL requires from the developer a solution structure that enables the agent to solve the overall problem. The *Combined* method views the world as a living environment in which the agent should try to live as agreeable as possible. In order to determine what is agreeable, the developer defines state preference. In this fashion, the *Combined* method requires state preference values from the developer that enables the agent to identify the agreeability of situations.
- The HRL method still learns a global policy. The hierarchical structure limits the amount of possible policies to learn, but the learning process essentially still encompasses the global environment. The *Combined* method only <u>learns</u> policies for the local environments, the global policy is <u>created</u> from these local policies. The learning process of the *Combined* method no longer encompasses the global environment.
- The HRL requires different additional information to be provided by the developer. What information is required depends on the definition of a subtask. This information can consist of subtasks with termination states, local reward functions, subtask-specific-actions or complete fixed policies. The *Combined* method requires a decomposition of the reward function into a (heuristic) state reward function and an action reward function.

Both methods results in global policies that approximate the global optimal policy. The optimality of the hierarchical policy depends for a large part on the hierarchical structure, the optimality of the global *Combined* policy depends for a large part on the heuristic function. Both method also learn local policies for smaller MDPs, respectively called subtasks and situations.

A fruitful conjunction of both methods is unlikely. It might be possible to use the local reward function of subtasks to define situations within that subtask, but this would only be useful for complex subtasks and these complex subtasks could best be solved in a HRL fashion by dividing the subtask into more subtasks or use state abstraction. The other way around, where HRL is used within the *Combined* method, is also most likely ineffective: Because the situations that are created by the *Combined* method are not explicitly given by the developer, it would be a dangerous if not impossible task to define a hierarchical structure for each situation.

The action abstraction used by HRL for the subtasks might be a worthwhile addition to the *Combined* method. In specific situations, some actions are irrelevant and these could be removed from the situation. One example for the modelled CTF world would be to ignore the ‘Score’ action for all situations where the agent does not have the flag. Such an abstraction would furthermore reduce the computational complexity of learning policies.

6 Empirical evaluation

This chapter gives the results of the empirical evaluation performed as part of the assignment. The first paragraph handles the computational cost evaluation and the second paragraph gives the game performance evaluation that indicates policy optimality. The last paragraph gives a summarised comparison between the *Complete* and *Combined* method. The next chapter uses these results for conclusions and as a handle for discussions.

6.1 Computational cost

This paragraph gives the results of the empirical evaluation of computational costs as calculated in the fashion described in paragraph 3.2.5 and given by equations **(8)**, **(9)** and **(10)**. Each computational cost value is based on 40 policy learning processes that were performed in the CTF environment explained in appendix D. In the modelled CTF environment, situational reinforcement learning decomposed 1 learning problem into respectively 21 – the *Standard* environment – and 13 – the *Alternative* environment – smaller and more tractable learning problems.

The following three tables give for one of the used discount factors the calculated computational cost values for all combinations of learning method and learning environment. For the *Combined* and *Enhanced* methods, not only the exact calculated values are given, but also how this value compares to the *Complete* method in percentages. For the *Enhanced* method, this is viewed in two ways: with and without the addition of the *Combined* computational cost. The value without the *Combined* complexity illustrates whether using the resulting global policy of the *Combined* method as a starting policy for modified PI requires less iterations, which was a hypothesis explained in paragraph 4.2.5. The value with the *Combined* complexity illustrated the actual cost required for learning the global policy and gives an indication whether using the *Combined* method as a starting point is computationally worthwhile.

Because the *Combined* and *Enhanced* methods are inapplicable for the *Simple* environment, since the *Combined* method requires a more complex reward function to be able to create situations, the computational cost values for learning global policies in the *Simple* environment have only been calculated for the *Complete* method.

Discount factor 0.9		**All *C* values** $\cdot 10^7$		
		Standard	**Alternative**	**Simple**
Complete	$C_{complete} = \hat{i}_{avg,complete} \cdot n$	2,16	2,27	2,48
Combined	$C_{combined} = \sum_{\theta \in \Theta} \left(\hat{i}_{avg,\theta} \cdot n_\theta \right)$	0,80	2,10	
	Percentage with respect to *Complete* in same environment	37,0%	92,6%	
Enhanced	$\hat{C}_{enhanced} = \hat{i}_{avg,enhanced} \cdot n$	2,00	2,03	
	Percentage with respect to *Complete* in same environment	92,6%	89,4%	
	$C_{enhanced} = C_{combined} + \left(\hat{i}_{avg,enhanced} \cdot n \right)$	2,80	4,13	
	Percentage with respect to *Complete* in same environment	129,6%	182,0%	

Table 1 Computational cost values for all combinations of learning method and learning environment with discount factor 0.9

Discount factor 0.5		All *C* values $\cdot 10^6$		
		Standard	Alternative	Simple
Complete	$C_{complete} = \hat{i}_{avg,complete} \cdot n$	3,74	4,17	6,16
Combined	$C_{combined} = \sum_{\theta \in \Theta} \left(\hat{i}_{avg,\theta} \cdot n_{\theta} \right)$	2,10	3,68	
	Percentage with respect to *Complete* in same environment	56,1%	88,3%	
Enhanced	$\hat{C}_{enhanced} = \hat{i}_{avg,enhanced} \cdot n$	3,14	3,47	
	Percentage with respect to *Complete* in same environment	84,0%	83,2%	
	$C_{enhanced} = C_{combined} + \left(\hat{i}_{avg,enhanced} \cdot n \right)$	5,24	7,16	
	Percentage with respect to *Complete* in same environment	140,1%	171,5%	

Table 2 Computational cost values for all combinations of learning method and learning environment with discount factor 0.5

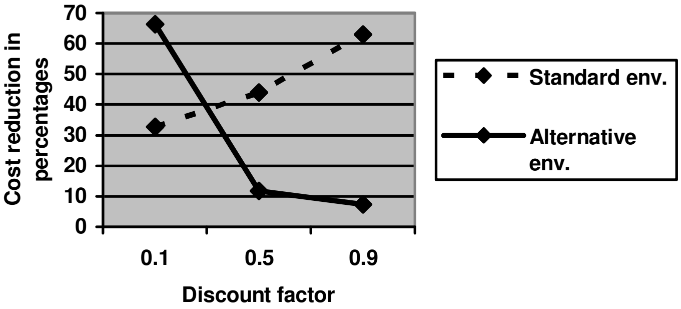
Discount factor 0.1		All *C* values $\cdot 10^6$		
		Standard	Alternative	Simple
Complete	$C_{complete} = \hat{i}_{avg,complete} \cdot n$	1,78	3,42	3,56
Combined	$C_{combined} = \sum_{\theta \in \Theta} \left(\hat{i}_{avg,\theta} \cdot n_{\theta} \right)$	1,19	1,15	
	Percentage with respect to *Complete* in same environment	67,2%	33,7%	
Enhanced	$\hat{C}_{enhanced} = \hat{i}_{avg,enhanced} \cdot n$	1,43	1,82	
	Percentage with respect to *Complete* in same environment	80,5%	53,3%	
	$C_{enhanced} = C_{combined} + \left(\hat{i}_{avg,enhanced} \cdot n \right)$	2,63	2,97	
	Percentage with respect to *Complete* in same environment	147,7%	87%	

Table 3 Computational cost values for all combinations of learning method and learning environment with discount factor 0.1

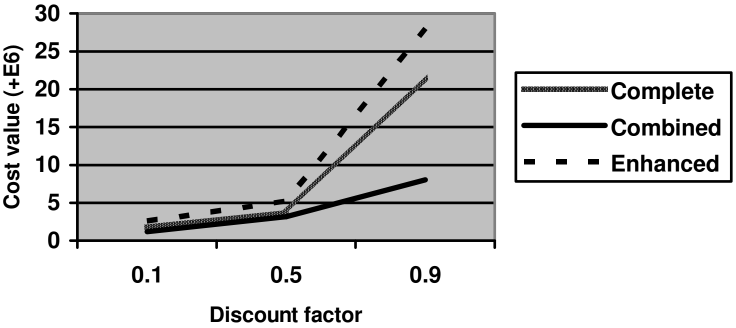
From tables 1, 2 and 3, the following observations can be made:

- Using the *Combined* method results in significant computational cost reductions in comparison to the *Complete* method. The cost reduction is, averaged over the different discount factors, 46.57% and 28.47% for respectively the *Standard* and *Alternative* environment. When also averaging over the environment, using the *Combined* method results in an average cost reduction of 37.52%.
- The effect that the discount factor has on the computational cost reduction brought about by the *Combined* method differs between the *Standard* and *Alternative* environments. Graph 1 illustrates this: for the *Standard* environment the reduction is greater for higher discount factors; for the *Alternative* environment the reduction is greater for lower discount factors. In the next chapter an explanation for this behaviour is given.

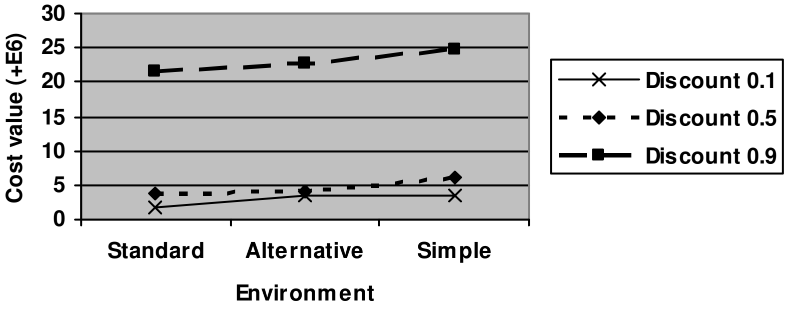
- The computational cost of using the *Enhanced* method without considering the additional cost of performing the *Combined* method is reduced in comparison to the *Complete* method. The reduction – which is brought about by starting from the learned *Combined* behaviour – is, averaged over the different discount factors, 14.3% and 24.7% for respectively the *Standard* and *Alternative* environment. When also averaging over environment, using the *Combined* method's learned information in itself results in an average cost reduction of 19.5%.
- The total computational cost of using the *Enhanced* method – meaning that the additional cost of performing the *Combined* method is also considered – is increased in comparison to the *Complete* method. The increase is, averaged over the different discount factors, 39.10% and 46.83% for respectively the *Standard* and *Alternative* environment. When also averaging over environment, using the *Enhanced* method results in an average cost increase of 42.97%. This shows that the cost reduction brought about by using the *Combined* method's resulting information is on average not significant enough to overcome the additional cost of performing the *Combined* method. One case did exist where the cost of using the *Enhanced* method showed a reduction in comparison to the *Complete* method, but this was but one case among six.
- Using a higher discount factor results in a higher computational cost. This is illustrated in graph 2 for the *Standard* environment. The same trend is observed for the *Alternative* and *Simple* environments.
- Using a more complex reward structure results in lower computational costs. When considering the complexity of the reward structure – where a more complex reward function assigns more unique state rewards - the *Standard* environment has the most complex reward function and the *Simple* environment has the least complex reward function. This is illustrated in graph 3 for the *Complete* method and the same trend is observed for the *Combined* and *Enhanced* methods.



Graph 1. Cost reduction in percentages of using the *Combined* method in comparison to the *Complete* method.



Graph 2. Computational cost values in the *Standard* environment.



Graph 3. Computational cost values for the *Complete* method.

6.2 Policy optimality

This paragraph gives the results on game performance that is used to indicate policy optimality. Each of the tables 4 to 11 gives for a certain discount factor and environment the game results of competing methods. Between all competing methods, the tables gives the percentage of games won and the percentage of games more won, along with the amount of valid[1] games played. Game performance evaluation is explained in paragraph 4.2.6 and equation **(11)** is given there to calculate the percentage of games more won. Although 2400 games were played between any two computer-controlled players in order to generate statistical significant results, this amount of games was impossible to achieve with human players. Because the amount of human test-subjects was limited, the performance values generates from these games give an indication of performance. In order to generate a more reliable indication, the human players only played games against policies learned with the discount factors of 0.1 and 0.9, not against policies learned with a discount factor of 0.5. By doing so, the games that were played could be played more often whilst it is likely that no relevant information was lost, since the results of policies learned with the 0.5 discount factor almost always lay in between the results of policies learned with the 0.1 and 0.9 discount factor.

From the results in tables 4 through 12, the following overall observations can be made:

- On overall it can be said that the *Complete* method outperformed the *Combined* method, in both head-to-head matches and in matches against non-learning methods.
- When considering the *Combined* method, policies learned in the *Alternative* environment perform considerably than policies learned in the *Standard* environment. The *Combined Alternative* policies outperformed the *Combined Standard* policies in matches against all other methods. The *Combined Alternative* policies performed nearly as well as the *Complete Alternative* policies did against the *Random* method.
- When analysing the influence of the discount factor on the *Complete* and *Combined* methods, the results show that performance of the *Complete* method decreases faster with lower discount factors than the *Combined* method. Because of this, the performance of the *Combined* method in comparison to the *Complete* method increases with lower discount factors. This can be demonstrated by the games played against human players: In *Human* versus *Complete* games, the *Complete* performance decreased significantly with lower discount factors. In *Human* versus *Combined* games, the discount factor did not influence performance all that much.

[1] A game was considered invalid if a pre-defined amount of time had expired. Because time was not modelled into the game environment, these games were considered invalid rather than tied (see paragraph 3.2.6).

- When analysing the influence of the learning environment, or more specifically the use of more complex[1] reward functions, on game performance, tables 10 through 12 show that performance depends on both the complexity of the reward function and on the used discount factor. When high discount factor are used, more complex reward functions distract the player from winning, resulting in lesser performance. When low discount factors are used, more complex reward functions can, if they correctly represent the desire to win, give a handle for the learning process to converge too, resulting in better performance. This is discussed in more detail in the next chapter.

The following more detailed observations about performance of the *Complete* and *Combined* learning methods in the *Standard* and *Alternative* learning environments can be made from tables 4 through 9:

- For all learning environments, the *Complete* method outmatches the *Combined* method. For the *Standard* environment, the *Complete* methods wins between 68,04% and 78,25% more games. The *Combined* method performs considerably better in the *Alternative* environment, where the *Complete* method wins between 41,38% and 61,04% more games. For all environments, a lower discount factor results in better performance of the *Combined* method against the *Complete* method.
- For all environments, the *Complete* method wins almost in 100% of the games from the *Random* method. All games that are not won by *Complete* end in a tie.
- In the *Standard* environment the *Combined* method never loses from the *Random* method, but only accomplishes to win between 34,63% and 39,63% games. In the *Alternative* environment, the *Combined* method performs nearly as well against the *Random* method as the *Complete* method did (almost 100% games won), the only difference being 4% more tied games.
- Performance of the *Complete* method against human players (*Human* method) depended critically on the used discount factor. If a discount factor of 0.9 was used, the *Complete* method performed better than human players, if a discount factor of 0.1 was used, the human players outmatched the *Complete* method. The results further show that *Complete* policies learned in the *Alternative* environment performed better against human players than *Complete* policies learned in the *Standard* environment.
- The *Combined* method was outperformed by the human players in both environments, although *Combined* policies learned in the *Alternative* environment performed better. The discount factor did not affect the performance of the *Combined* policies against human players, unlike the *Complete* policies.

The following observations about performance of the *Complete* method in the *Simple* environment can be made from tables 10 through 12:

- Policies learned in the *Simple* environment perform better than policies learned in the *Standard* environment, where the *Simple* policies win between 9,75% and 36,75% more games.
- Policies learned in the *Simple* environment do not necessarily perform better than policies learned in the *Alternative* environment. Although the *Simple* policies managed to win between 14,96% and 27,79% more games when high discount factors were used, the *Alternative* policies managed to win 5,25% more games when a discount

[1] When a more *complex* reward function is discussed, this refers to a reward function that assigns more unique state rewards. It does not refer to the computational complexity that was used as part of the theoretical evaluation.

factor of 0.1 was used. Lower discount factors resulted in better performance for the *Alternative* policies.

- Lower discount factors also resulted in decreased performance of the *Complete Simple* policies when playing against human players. When the highest discount factor of 0.9 was used, the *Complete* policies managed to win 45% more games. When the lowest discount factor of 0.1 was used, the human players and the *Complete Simple* policies performed equally well.
- Just as with the *Complete* policies learned in the *Standard* and *Alternative* environment, the policies learned in the *Simple* environment won almost 100% of the games against the *Random* player.

The results of games played in the *Standard* environments:

Discount factor 0.9 Standard Environment		Complete 2400 games	Combined 2400 games	Random 2400 games	Human 20 games
Complete	Percentage of games won		85,29%	98,96%	45,00%
	Percentage of games more won		78,75%	98,96%	5,00%
Combined	Percentage of games won	6,54%		39,63%	15,00%
	Percentage of games more won	-78,75%		39,54%	-45,00%

Table 4. Game performance between methods in the Standard environment with a discount factor of 0.9

Discount factor 0.5 Standard Environment		Complete 2400 games	Combined 2400 games	Random 2400 games	Human 0 games
Complete	Percentage of games won		80,00%	97,42%	
	Percentage of games more won		68,25%	97,42%	
Combined	Percentage of games won	11,75%		46,13%	
	Percentage of games more won	-68,25%		46,08%	

Table 5. Game performance between methods in the Standard environment with a discount factor of 0.5

Discount factor 0.1 Standard Environment		Complete 2400 games	Combined 2400 games	Random 2400 games	Human 20 games
Complete	Percentage of games won		80,75%	98,75%	20,00%
	Percentage of games more won		68,04%	98,75%	-50,00%
Combined	Percentage of games won	12,71%		34,63%	25,00%
	Percentage of games more won	-68,04%		34,63%	-45,00%

Table 6. Game performance between methods in the Standard environment with a discount factor of 0.1

The results of games played in the *Alternative* environment:

Discount factor 0.9 Alternative Environment		Complete 2400 games	Combined 2400 games	Random 2400 games	Human 20 games
Complete	Percentage of games won		78,04%	99,42%	65,00%
	Percentage of games more won		61,04%	99,42%	35,00%
Combined	Percentage of games won	17,00%		95,21%	30,00%
	Percentage of games more won	-61,04%		95,21%	-30,00%

Table 7. Game performance between methods in the Alternative environment with a discount factor of 0.9

Discount factor 0.5 Alternative Environment		Complete 2400 games	Combined 2400 games	Random 2400 games	Human 0 games
Complete	Percentage of games won		74,25%	99,33%	
	Percentage of games more won		51,83%	99,33%	
Combined	Percentage of games won	22,42%		96,33%	
	Percentage of games more won	-51,83%		96,33%	

Table 8. Game performance between methods in the Alternative environment with a discount factor of 0.5

Discount factor 0.1 Alternative Environment		Complete 2400 games	Combined 2400 games	Random 2400 games	Human 20 games
Complete	Percentage of games won		68,66%	99,50%	20,00%
	Percentage of games more won		41,38%	99,50%	-35,00%
Combined	Percentage of games won	27,28%		96,71%	20,00%
	Percentage of games more won	-41,38%		96,71%	-35,00%

Table 9. Game performance between methods in the Alternative environment with a discount factor of 0.1

The results of the games played in the *Simple* environment

Discount factor 0.9 vs. Simple environment		Standard 2400 games	Alternative 2400 games	Random 2400 games	Human 20 games
Simple	Percentage of games won	37,25%	51,13%	97,34%	55,00%
	Percentage of games more won	9,75%	27,79%	97,34%	45,00%

Table 10. Game performance between the *Complete* methods in the *Simple* environment and the *Complete* methods in the other environments as well as non-learning methods, with a discount factor of 0.9

Discount factor 0.5 vs. Simple Environment		Standard 2400 games	Alternative 2400 games	Random 2400 games	Human 0 games
Simple	Percentage of games won	63,42%	41,83%	98,50%	
	Percentage of games more won	36,75%	14,96%	98,50%	

Table 11. Game performance between the *Complete* methods in the *Simple* environment and the *Complete* methods in the other environments as well as non-learning methods, with a discount factor of 0.5

Discount factor 0.1 vs. Simple Environment		Standard 2400 games	Alternative 2400 games	Random 2400 games	Human 20 games
Simple	Percentage of games won	59,83%	37,88%	99,65%	30,00%
	Percentage of games more won	28,79%	-5,25%	99,65%	0,00%

Table 12. Game performance between the *Complete* methods in the *Simple* environment and the *Complete* methods in the other environments as well as non-learning methods, with a discount factor of 0.1

6.3 Comparing Combined to Complete

Table 13 is a select summary of tables 1 through 9 and gives a comparison of computational cost and game performance between the *Combined* method and the *Complete* method for both environments and all discount factors.

	Standard **environment**		***Alternative*** **environment**	
	Combined **compared to *Complete***		***Combined*** **compared to *Complete***	
Discount Factor	Computational cost	More games won	Computational cost	More games won
0.9	-63,00%	-78,75%	-7,40%	-61,04%
0.5	-43,90%	-68,25%	-11,70%	-51,83%
0.1	-32,80%	-68,04%	-66,30%	-41,38%

Table 13. Summarised comparison of complexity and game performance between Combined and Complete

7 Conclusions & Discussions

In this chapter conclusions are drawn and discussions are started based on the results of the evaluations described in the previous two chapters. The goals of the assignment – which were discussed in the introduction – can be summarised as being to:

1. Develop the situational reinforcement learning approach. This approach must be applicable in MDP and Markov game modelled environments, be able to use any policy learning algorithm within such environments and be able to learn policies at a lower complexity cost than conventional reinforcement learning.
2. Apply situational reinforcement learning to a game of CTF modelled as a Markov game environment.
3. Evaluate the implications of SRL as a stand-alone approach – the *Combined* method – on policy optimality.
4. Evaluate the implications of SRL as a stand-alone approach – the *Combined* method – on computational cost/complexity.
5. Evaluate the implications of SRL as an addition to conventional reinforcement learning methods – the *Enhanced* method – on computational cost/complexity.

As for the first goal, the situational reinforcement learning approach is applicable to all MDP and Markov game modelled environment if these environments allow for a decomposition of the reward function into a state reward function and an action reward function. It has also been shown that, with the same criterion of a decomposable reward function, the approach can be applied to POMDP environments. Situational reinforcement learning can be used in conjunction with all dynamic programming algorithms. In it's current form, SRL cannot yet be used with reinforcement learning method that use incomplete models of the environment. Situational reinforcement learning can be seen as an approach that tries to reduce the computational cost of learning behaviour by performing the learning process on smaller environment. As such, the approach is an alternative to methods like hierarchical reinforcement learning and the envelope method.

The second goal has been met for a two player zero-sum CTF game. This environment – which is described in appendix D – has been used as the empirical evaluation tool for the third, fourth and fifth goal. Small examples have also been given on how SRL could be used to learn behaviour in a first-person shooter game and in a non-game environment. The CTF environment showed how situational reinforcement learning reduced the environment – which contained 136737 reachable states – into 21 game situation. In so doing, situational reinforcement learning decomposed a single large learning problem into 21 smaller and more tractable learning problems.

The last three goals were evaluated in a theoretical and empirical fashion. We will look at the evaluation results of using SRL as a stand-alone approach to learning first – the *Combined* method. The theoretical evaluation will be discussed prior to discussion of the empirical evaluation. The evaluation results of SRL as an addition to conventional reinforcement learning – the *Enhanced* method – will be discussed thereafter in the similar order.

The theoretical evaluation showed that the reduction in computational complexity facilitated by the *Combined* method in comparison to a method that used the learning algorithm in it's most basic way called the *Complete* method, is dependent on the learning algorithm used and the amount of situations created by the heuristic function. If *g* is the amount of created situations then the worst-case complexity of the *Combined* method, $O_{combined}(\cdots)$, relates to the worst-case complexity of the *Complete* method, $O_{complete}(\cdots)$, in the following manner for respectively the policy iteration algorithm and the modified policy iteration algorithm:

- $\frac{O_{complete}(\cdots)}{g^3} \leq O_{combined}(\cdots) \leq O_{complete}(\cdots)$
- $\frac{O_{complete}(\cdots)}{g^2} \leq O_{combined}(\cdots) \leq O_{complete}(\cdots)$

The reduction in computational complexity facilitated by the *Combined* method is determined by the influence of the state set size *n* in the complexity function of the learning algorithm used: If the state set size has a higher order in the complexity function, than the reduction brought about by the *Combined* method is also greater. The reduction when using standard policy iteration is therefore theoretically greater than when using modified policy iteration, because the state set size *n* is of the order $O(n^4)$ and $O(n^3)$ for the upper bound complexity functions of respectively policy iteration and modified policy iteration.

The optimality of the global policy created by the *Combined* method is difficult to predict. The *Combined* method acquires a global policy by combining learned local policies. Because of this process, the created global policy is most likely not optimal, even if the learned local policies were. How optimal the created global policy is depends largely on the heuristic function. This dependency is created by two factors:

1. The heuristic function, being the state reward function that assigns rewards to states, defines the preference of the states. If the heuristic function does not correctly represents the goals of the agent, which for games is to win, then the policy learned from this heuristic function will not perform it's goal optimally.
2. The heuristic function defines the situations that are created and one import aspect for global policy optimality is the ambiguity of the situation ordering. If the ordering of situations is unambiguous, meaning that the state rewards of the situations alone is a sufficient indicator for preference and situations with higher state rewards will most likely lead to situations with even higher state rewards, then the resulting global policy will be near-optimal. If on the other hand the ordering of situations is ambiguous, where situations with high state rewards can lead to situations with low state rewards (i.e. short-term reward traps), then the resulting global policy can be disastrous.

Now let's look at the results of the empirical evaluation of the *Combined* method. Using the *Combined* method in the modelled CTF world always resulted in a significant reduction of computational cost, with reductions between 7.4% and 66.3% and an average reduction of 37.52%. Although the maximal-minimal cost reduction did not differ all that much between the two environments *Standard* and *Alternative*, there was a striking difference in the effect the discount factor had on cost reduction in both environments. In the *Standard* environment the reduction was greater for lower discount factors and in the *Alternative* environment the reduction was greater for higher discount factors. The explanation for this probably lies in the amount of (unique) state rewards distributed by the reward function and how this affects the relative influence of the discount factor on computational cost. In general – and as can be seen from the results – using higher discount factors result in higher computational cost, but how the discount factor and computational cost relate, depends on the environment.

Let's consider two environment on which we apply the *Complete* method. One environment will be referenced to as the *Few* environment and the reward function in this environment assigns few (unique) rewards. The reward function of the other environment assigns many rewards and is referenced to as the *Many* environment. The assumption made is that the difference in computational cost between using a high and low discount factor is smaller in the *Few* environment than in *Many* environment. This assumption is based on the fact that when a low discount factor is used in the *Many* environment, the learning process can converge relatively quickly to the various different state rewards, resulting in a lower computational cost. If a high discount factor is used in a similar environment, the algorithm does not converge to local rewards but rather to rewards further away. In a *Few* environment, the difference in computational cost between using a low and high discount factor will be smaller, because there aren't too many state rewards to converge too, resulting in more similar policies between low and high discount factors. Table 14 summarises this assumption.

Now let's consider the same *Few* and *Many* environments when we apply the *Combined* method. For this learning method, the assumption is that the difference in computational cost between low and high discount factors will be smaller when using the *Many* environment in comparison to the *Few* environment. This assumption is based on the fact that, in general, the computational cost difference between using low and high discount factors increases with the size of the environment. With a larger environment, influencing states can be farther away, resulting in a greater difference between high and low discount factors because high discount factors are influenced more by these far-away states. Because applying the *Combined* method to a *Few* environment results in less – but larger – situations than when the method is applied to a *Many* environment, the difference in computational cost between low and high discount factors will be greater for the *Few* environment. Table 15 summarises this assumption.

Using the *Complete* method	**Low discount factor**	**High discount factor**
***Many* environment**	Quick convergence because ample opportunity for local convergence.	Slow convergence because high discount factor ignores local rewards and converge to global rewards.
***Few* environment**	Slow convergence because few state rewards give little opportunity for local convergence.	Slow convergence because high discount factor converges to global rewards.

Table 14. Influence of discount factor and environment on learning convergence, and thus computational complexity, when using the *Complete* method.

Using the *Combined* method	**Low discount factor**	**High discount factor**
***Many* environment**	Quick convergence because of small situations.	Quick convergence because of small situations.
***Few* environment**	Quick convergence. Although the situation is large, the learning process can at least benefit from local convergence.	Slow convergence because of large environment and no local convergence.

Table 15. Influence of discount factor and environment on learning convergence, and thus computational complexity, when using the *Combined* method.

Within the modelled CTF world, the *Alternative* environment with it's 13 situations uses a less complex reward structure than the *Standard* environment with it's 21 situations. If we identify the *Alternative* environment as a *Few* environment and the *Standard* environment as a *Many* environment, then by using the above mentioned assumptions we can explain the difference in the implication of the discount factor between the two environments: In the *Standard* environment, the difference in computational cost between low and high discount factors when using the *Complete* method is relatively large whilst this difference is relatively small when the *Combined* method is used. Therefore when using the *Complete* method, the computational cost increases faster with higher discount factors than when the *Combined* method is used, resulting in greater reductions for the *Combined* method with higher discount factors. In the *Alternative* environment, the difference in computational cost between low and high discount factors is greater when the *Combined* method is used, resulting in lesser computational cost reduction when higher discount factors are used. For both methods, it still applies that higher discount factors means higher computational costs and that using the *Combined* method always yields a significant decrease in computational cost (averaged over discount factors and learning environment the average computational cost reduction is 37,52%).

When considering the optimality implications of using the *Combined* method, the empirical results show that the use of the *Combined* method always results in a significant optimality decrease. The game performance of the learned policies – which is used as an indication for policy optimality – shows that between 40% and 80% of games are more lost if the *Combined* method is used. As can be shown by the difference in performance of policies learned in the *Standard* and *Alternative* environments, the performance depends for a large part on the heuristic function. Analysis of the learned *Combined* policies in the *Standard* environment showed an unexpected problem that seriously hampered performance: In the *Standard* environment, killing the opponent resulted in an increase of state reward, which from the perspective of local learning processes meant a different situation. Because a dead opponent always returned to the game after two action, the situation also had to change after two action (to the situation where the opponent was alive once again) unless a situation change could be realised in less actions. In so doing, these situations where the opponent was dead became episodic situations: If the agent did not have the chance to realise another way of changing situations (such as picking up the flag or scoring a point) within two actions, the agent chose to do nothing for those two actions. This is not surprising since, from the perspective of the local learning process, always the same situation was reached, so why perform any action at all? The *Alternative* environment was devised as a solution to this problem, but the problem illustrates the danger of creating unwanted effects when using the *Combined* method. The developer should therefore carefully consider the heuristic function. One important issue for future research is to find a way, other than the *Enhanced* method, to extend the *Combined* method to overcome the short-sightedness that hampers it's optimality. Multiple solutions could be used, which are explained in more detail in chapter 9.

The empirical results further show that the *Combined* method always performs relatively better with lower discount factors. This can probably be explained through local policy convergence: When using the *Combined* method, the local learning process can only converge to inner-state rewards or single-transition reachable outer-state rewards; In either case, convergence will occur to relatively nearby states. When using a low discount factor with the *Complete* method, the policy learning algorithm will also converge to relatively nearby state-rewards. For low discount factors, the *Complete* and *Combined* methods therefore mimic each other's behaviour to some extend, resulting in policies that are more alike. When high discount factors are used, the *Complete* method can converge to rewards of states farther away but the *Combined* method is still limited to single-transition reachable outer-state rewards.

Because planning ahead is almost always favourable for games, the *Complete* method performs relatively better with higher discount factors than the *Combined* method.

When analysing the influence of the learning environment, or more specifically the use of more complex[1] reward functions on game performance, the results show that performance depends on both the complexity of the reward function and on the used discount factor. When a high discount factor is used, complex reward functions distract the agent from achieving it's goal, resulting in lesser performance. When low discount factors are used, more complex reward functions can give a handle on states for the learning process to converge too, resulting in better performance. Let's take for example a man in a room who wants to reach the exit-door to explain this. Let the discount factor represent the vision of the man, where higher discount factors means the man can see farther into the room. Let the reward function represent the amount of signposts that point to the exit-door in the room. If the man has a high discount factor and is able to see the door from his starting location, he does not need the signposts to reach the door and continually reading all the signposts distract him from running to the door, so he prefers a less complex reward function. If the man has a low discount factor and does not see the door from his starting location, the signposts can help him walk in the right direction so he prefers a more complex reward function.

From a theoretical point of view, the reduction or increase in computational cost resulting from the use of the *Enhanced* method is difficult to predict. If the *Enhanced* method uses a policy improving policy learning algorithm, such as (modified) policy iteration, then the computational cost depends largely on the optimality of the *Combined* global policy.

The empirical results show that in almost all cases, the computational cost of learning an optimal policy through the *Enhanced* method is higher than by simply using the *Complete* method. The empirical evaluation does show that it is possible to learn an optimal policy with a lower complexity cost by using the *Enhanced* method. The results show that the computational costs of performing the *Enhanced* method, without looking at the additional cost of performing the *Combined* method, is less than applying the *Complete* method. This indicates that the hypothesis from paragraph 4.2.5. that less iterations are required when starting from the *Combined* policy than when starting from a random policy is correct. This reduction in iterations increases when lower discount factors are used. This is probably also because of the previously mentioned local policy convergence: the *Combined* and *Complete* policies are more alike when lower discount are used. When the policies are more alike, it would require the *Enhanced* method less iterations to reach the same optimal policy found by the *Complete* method.

Although using the resulting *Combined* policy as a starting policy reduces the amount of iterations required to learn the optimal policy, this reduction is unfortunately most often not significant enough to overcome the added computational cost of performing the *Combined* method. The most obvious reason for this is that the global policy resulting from the *Combined* method is not optimal enough and requires too much additional iterations. Another reason, one of which the implications have not been researched, might be the use of the modified policy iteration algorithm as a policy learning algorithm for the *Enhanced* method. In this policy learning algorithm, the policy evaluation phase uses learned utility values of previous iterations. Because of this, the *Enhanced* method did not only use the *Combined*

[1] When speaking of reward function complexity, not the computational complexity is meant but rather the amount of unique state rewards that are assigned to states by the function. A more complex reward function thus assigns more unique state rewards, resulting in more situations if the *Combined* method is used.

method's resulting global policy, but also the *Combined* method's resulting global utility set, as explained in chapter 2. But this does introduce a possible problem, because the resulting utility values of a situation depends on the amount of iterations required for the local learning process of that situation. Let's explain this problem with the example in figure 6: In this figure, three states, which actually represent situations, are depicted by circles and utility values are depicted below the situation name. Figure 6a. represents the initial world, before local learning is applied by the *Combined* method to the situations. Now, let's assume that situation S_1 required 1 iteration to learn the optimal policy and situation S_3 required 6 iterations, then figure 6b represents the utility values for each situation. If a global utility set is now created, it would appear that situations S_3 is more favourable than situation S_1, since it has a higher utility value, but this is only so because situation S_3 required more iterations. Although this error will be corrected by the global learning of the *Enhanced* method, it may require additional iterations than would be the case if only the global policy was used in for example the policy iteration algorithm. Besides the fact that this problem is avoided when using this other learning algorithm, another reason why the *Enhanced* method will probably have a lower complexity cost when using another learning algorithm such as policy iteration or value iteration is because the state set size has a greater influence in the computational complexity of those algorithms, probably resulting in a greater computational cost reduction of the *Combined* method, as explained in paragraph 4.2.4.

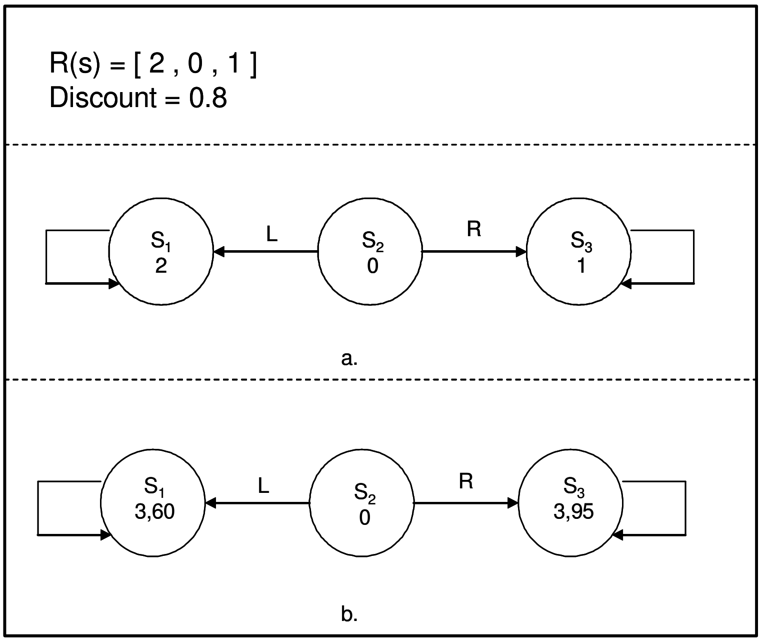


Figure 6. Three states representing situations.
a. Utility values before local learning
b. Possible utility values after local learning.

Summarised, the following conclusions can be drawn:

- Situational reinforcement learning is applicable for any environment modelled as a (PO)MDP or Markov game environment that allows for a decomposition of the reward function.
- SRL can be used in conjunction with any dynamic programming algorithm. SRL in it's current form cannot be used in conjunction with reinforcement learning methods that use an incomplete model of the environment.
- The SRL approach has been implemented and tested for a two player zero-sum Markov game environment modelled after a CTF game.

- The use of the *Combined* method results in a significant reduction of the computational cost, with an average reduction of 37.52%. This reduction depends largely on the heuristic function (more specifically the amount of situations), used discount factor and a combination of the two.
- The use of the *Combined* method also results in a significant reduction of policy optimality, with an average percentage of 51.42% more games lost. This reduction depends largely on the nature of the environment and the heuristic function (more specifically how the situations are structured within this environment).
- Whether the trade-off in computational cost and policy optimality is acceptable is a consideration for the developer. In my opinion, the optimality loss is too often too large for most practical applications of the *Combined* method, but the significant reduction in computational cost is a sufficient incentive to research possible enhancements to reduce the optimality loss.
- The use of a more complex reward function reduces computational cost and policy optimality, although the reduction in policy optimality can be reduced or even nullified if the reward function is devised in such a way that correctly represents the desire to win (or achieve a goal).
- Using the resulting *Combined* policy as a starting policy for modified policy iteration (the empirically tested *Enhanced* method) requires less iterations than a random policy would require.
- Unfortunately, the reduction in iterations is often not significant enough to overcome the added computational cost of performing the *Combined* method, although it has been shown that it is possible to achieve equal optimality with lower computational cost. It is probable that the computational cost of the *Enhanced* method is lower when the policy iteration or value iteration learning algorithms are used.

8 Summary

This document described an approach to policy learning, called situational reinforcement learning (SRL), based on a decomposition of the environment into so called *situations* by using heuristic preference values for states. Within the context of this document, the method that applies situational reinforcement learning as a stand-alone approach to learning was referenced to as the *Combined* method. The main goal of the *Combined* method was to enable policy learning at a lower computational cost than conventional reinforcement learning. This document described how SRL works, gave an implementation of the method for a game environment and described a theoretical and empirical evaluation of the method's implication on computational complexity/cost and policy optimality. Besides the *Combined* method, an explanation, implementation and evaluation was also performed on a method that used the resulting *Combined* method policy to enhance it's optimality. This method was referenced to as the *Enhanced* method, and was created to evaluate the use of SRL method as an addition to conventional reinforcement learning.

Inspiration for situational reinforcement learning came from an analogy with a common human approach to playing games: Humans tend to restrict their planning from their current situation to reaching rather short-term more favourable situations instead of planning the entire game at once. Humans also often have, especially so for games, their own heuristics that tell them which situations are favourable in comparison to others. SRL copies this behaviour by decomposing the environment into situations with equal preference according to a heuristic function. The local learning processes than only focuses on reaching more favourable situations. For the application of SRL a decomposition of the reward function is required. It is most common for the reward function *R* in such environments to assign rewards to a combination of state(s) and action(s), $R(s,a)$. SRL requires that a decomposition is possible into a state reward function $SR(s)$ and an action reward function $AR(a)$, where the reward function *R* becomes a function of the others, such as $R(s,a) = SR(s) + AR(a)$[1]. The state reward function *SR* would then be the heuristic function on which the decomposition into situations is based. Each unique reward specified by the state reward function defines a new situation. Each situation, defined by a unique state reward which we call *base reward* here, can be created from the entire environment in the following fashion:

1. Identify all states with a state reward that equals the base reward as inner states.
2. Remove all transitions that do not originate from inner states.
3. Identify the reachable states which are not inner states as outer states.
4. Remove all states that are not inner or outer states.
5. Add new transitions for the outer states to make them absorbing states.

Each situation constructed in this fashion is an MDP-like environment on it's own in which policies can be learned with conventional reinforcement learning algorithms. Because from a local learning perspective the only states with a different state reward are the outer states, the local learning process will most likely direct the agent these outer states, if they are more favourable then the inner states. A global policy is created by taking for each state the learned action from that state from the situation where that state is an inner state. By creating a global

[1] This notation does not correspond with the definition of the reward function. The reward function should "give the expected immediate reward", so a decomposition into *SR* and *AR* that corresponds in a better way to this definition would be $R(s,a) = AR(a) + \sum_{s'} T(s,a,s') \cdot SR(s')$. Because an MDP or Markov game environment does not specify how the reward function should assign rewards, this decomposition is not mandatory.

policy in this fashion, the global policy will most likely direct the agent to continually improving situations: a rather human approach to game playing.

Situational reinforcement learning is compatible with environments modelled as an MDP, Markov game or POMDP. The method can be used in conjunction with any dynamic programming algorithm, such as value iteration or (modified) policy iteration. The implications on computational cost/complexity and policy optimality when using the approach was evaluated for a Markov game environment where only modified policy iteration was used as a learning algorithm. In it's current form, situational reinforcement learning cannot be used in conjunction with reinforcement learning methods that use incomplete models of the environments, such as temporal difference learning or Q-learning.

Because the local policies learned by SRL are only optimal for their corresponding situations, the global policy that is created from combining them is most likely sub-optimal. The *Enhanced* method is an extension of the *Combined* method, that takes the resulting global policy from the *Combined* method and enhances this policy to become optimal by using conventional policy learning algorithms. In this fashion, SRL is used as an addition to conventional reinforcement learning and the question that needed answering was whether this results in lower computational costs for equal optimality in comparison to the conventional methods.

The *Combined* and *Enhanced* methods were put into practice for a two player zero-sum Markov game modelled after the game Capture the flag. The created environment is explained in appendix D and the written program used for learning and playing in that environment is explained in appendix F.

The evaluation of the *Combined* and *Enhanced* methods tried to give answers to the following questions:

- What are the implications on computational complexity/cost and on policy optimality when using the *Combined* method?
- What are the implications on computational complexity/cost and on policy optimality when using a more complex reward structure, which is a necessity for the *Combined* method?
- What are the implications on computational complexity/cost when using the *Enhanced* method? This is considered with and without the addition of the computational cost of the Combined method. Without the addition indicates whether using a non-random starting policy results in lesser costs and with the addition indicates whether using the *Enhanced* method as a whole has potential.

The evaluation was performed both theoretically and empirically, where most of the attention was given to the empirical evaluation. Although all dynamic programming algorithms for (PO)MDP or Markov game modelled environments could be used, the empirical evaluation continually used the modified policy iteration algorithm explained in appendices B and C and in several studies (Vrielink, 2005; Mansour & Singh, 1999; Russel & Norvig, 2003; Kaelbling, 1996; Kaelbling et al., 1998; Aberdeen, 2003).

An evaluation on the computational complexity implications of the *Combined* method by using worst-case complexity functions showed that the reduction in computational complexity brought about by the *Combined* method depends on the learning algorithm used and the amount of situations created. The reduction is a factor of the amount of the situations and the value of this factor is defined by the weight of the state set size in the complexity function of the used learning algorithm. If the amount of situations is given by *g* and the state set size by *n*, then for respectively the policy iteration and modified policy iteration algorithms the worst-case complexity of *Combined*, $O_{combined}(\cdots)$, relates to the standard upper-bound complexity of not using *Combined*, $O_{standard}(\cdots)$, in the following way:

- $\frac{O_{standard}(\cdots)}{g^3} \leq O_{combined}(\cdots) \leq O_{standard}(\cdots)$
- $\frac{O_{standard}(\cdots)}{g^2} \leq O_{combined}(\cdots) \leq O_{standard}(\cdots)$

The possible worst-case reduction is greater when using standard PI than when using modified PI, because the state set size *n* of both methods are respectively of the order $O(n^4)$ and $O(n^3)$.

The empirical evaluation showed that the reduction in computational cost when using the *Combined* method in the modelled environment lay between 7.4% and 66.3%, depending on used reward function, discount factor and a combination of the two. Averaged over used reward functions and discount factors, the average empirical reduction in computational cost was 37,47%.

The empirical evaluation on policy optimality was performed by using game performance as an indication to optimality. The empirical evaluation showed a danger of using the *Combined* method. Although the original environment described in appendix D, referenced to as the *Standard* environment, appeared correct in the sense that higher rewards were only given to states for which it was safe to say that they were more preferable, an unforeseen problem where an episodic situation was created devastated game performance. In direct competitions between *Combined* and *Complete* policies, where the *Complete* method can be described as being the conventional application of the policy learning algorithm, the *Combined* policies were outmatched in all cases. Between 40% and 80% of games were most lost, although performance drastically increased when the above mentioned problem was solved, referenced to as the *Alternative* environment. When only considering the *Alternative* environment, the *Combined* policies lost 51.4% more games averaged over discount factors. Against an opponent that plays random moves, the *Alternative Combined* policies performed nearly as well as the *Complete* policies. The evaluation showed that although lower discount factor resulted in lower game performance for all methods, the reduction in performance is greater for the *Complete* method than for the *Combined* method.

The empirical evaluation also showed that although using a more complex reward structure, meaning a reward structure that assigns more unique rewards, can distract an agent from winning the game, it can also assist the agent in finding an optimal policy sooner, especially for lower discount factors. So although the use of a more complex reward function does have implications on computational cost and policy optimality, it may prove to be a worthwhile endeavour if the rewards are chosen carefully by the developer.

The computational cost of using the *Enhanced* method was for all but one combination of environment and discount factor greater than the computational cost of using the *Complete* method. This indicates that although it is possible to reduce complexity cost for an equal level of optimality, it is most likely not the case if the *Enhanced* method is used in it's current form that uses the modified policy iteration algorithm. Because of a possible problem when using the global utility set – which is a requirement for modified policy iteration to continue learning – and because the reduction in computational cost brought about by the *Combined* method depends on the learning algorithm used, the *Enhanced* method may yet prove to be useful when other policy learning algorithms are used. This is then especially the case for policy learning algorithms where the computational complexity is largely dependant on state set size.

The applicability of SRL covers a broad domain – any MDP-like environment and dynamic programming algorithm can be used – and in it's current form the method could prove useful for problems with unambiguous situation-orderings, such as illustrated with the taxi domain in paragraph 2.4. In my opinion, the policy optimality loss created by the use of the *Combined* method is still too great for most practical application. I believe that the significant reduction in computational cost on the other hand is enough of an incentive to perform further research methods to reduce the optimality loss.

9 Further research

Based on the research described in this document, further research of situational reinforcement learning can be divided into two categories: additional research on the implications of the approach as described in this document and research into enhancements of the approach. The first category aims to getter a deeper insight into the pro's, con's and potentials of SRL, where the second category tries to increase the performance of SRL.

The following items are eligible for additional research into the implications of SRL:

- Perform an evaluation where a guarantee can be given that the optimal policy has been learned instead of an assumption.
- Perform an evaluation where the SRL is applied to other environments and not just to a CTF modelled Markov game environment.
- Perform an evaluation on how SRL performs against similar learning methods, such as the described Envelope method or Hierarchical Reinforcement Learning.
- Research what the implications are when the *Enhanced* method is used in conjunction with other policy learning algorithms than just modified policy iteration.
- Research how SRL can be modified to be compatible with reinforcement learning methods that use an incomplete model of the environment.

As for the enhancements to SRL, I believe that the following may prove worthwhile:

- After having learned local policies for situations, make use of this learned local information, such as learned utility values, to benefit the learning process of neighbouring situations. A possibility that might be worthwhile to research is to use the learned utility values of inner states as the utility values for the learning process of neighbouring situations where those inner states are outer states. In such a fashion, the local learning processes remain local (in contradiction to the *Enhanced* method, where the learning scope became global) whilst still incorporating learned information of other parts of the global environment.
- Make use of action abstraction within situations to ignore actions irrelevant for transitions within the situation.
- Use an Envelope kind-of method, where the envelop which starts from a certain situation is continually expanded with most-likely reached situations.
- Extend the local learning to a pre-defined 'depth' of situations. The *Combined* method described in this document would have a pre-defined depth of 1: consider only the current situation. A higher depth of for example 2 would expand the learning process to also incorporate single-transition neighbouring situations.

10 Literature

Aberdeen, D. (2003, December). *A (Revised) Survey of Approximate Methods for Solving Partially Observable Markov Decision Processes*. Retrieved July 7th, 2006, from http://users.rsise.anu.edu.au/~daa/files/papers/pomdpreview.pdf

Bakkes, S., Spronck, C., & Postma, E. (2004). TEAM: The Team-oriented Evolutionary Adaptability Mechanism. In Rauterberg, M. (Eds.), *Entertainment Computing - ICEC 2004*, volume 3166. Retrieved July 26th, 2006, from http://www.cs.unimaas.nl/p.spronck/Pubs/Team.pdf

Bakkes, S., Spronck, C., & Postma, E. (2005). Best-Response Learning of Team Behaviour in Quake III. In Aha, D.W., Muñoz-Avila, H., & Lent, M. (Eds.), *Proceedings of the 2005 IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games*. Retrieved Julyt 26th, 2006, from http://www.cse.lehigh.edu/~munoz/Publications/IJCAI05W-proceedings.pdf

Darryl, C (2003, November). Challenges for artificial intelligence in digital games. In Copier, M., & Raessens, J. (Eds.), *Digital Games Research Conference* (chap. 17). Utrecht University.

Dietrich, T.G. (1999, May). *Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition*. Retrieved July 7th, 2006, from http://arxiv.org/PS_cache/cs/pdf/9905/9905014.pdf

Dietrich, T.G (2000, July). An overview of MAXQ Hierarchical Reinforcement Learning. In Choueiry, B.Y. & Walsh, T. (Eds.), *Abstraction, Reformulation, and Approximation: 4th International Symposium* (pp. 26-44). Springer, Berlin / Heidelberg.

Gardiol, N.H., & Kaelbling, L.P. (2004). Envelope-based Planning in Relational MDPs. In *NIPS-03*. Retrieved July 7th, 2006, from http://people.csail.mit.edu/nhg/papers/nhg_lpk_nips03.pdf

Gill, A. (1962). *Introduction to the Theory of Finite-state Machines*. McGraw-Hill.

Haykin, S (1999). *Neural Networks: A Comprehensive Foundation*. Macmillan College Publishing, 2nd ed.

Kaelbling, L.P. (1996, May). Reinforcement Learning: A Survey. In *Journal of Artificial Intelligence Research 4* (pp. 237-285). Retrieved July 7th, 2006, from http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/kaelbling96a.pdf

Kaelbling, L.P., Littman, M.L., & Cassandra, A.R. (1998). Planning and Acting in Partially Observable Stochastic Domains. In *Artificial Intelligence, volume 101* (pp. 99-134). Retrieved July 7th, 2006, from http://athos.rutgers.edu/~mlittman/papers/aij98-pomdp.pdf

Littman, M.L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the Eleventh International Conference on Machine Learning* (pp. 157-163). San Francisco. Morgan Kaufmann. Retrieved July 7th, 2006, from http://www.cs.ualberta.ca/~bowling/classes/cmput608/Littman94.pdf

Mansour, Y., & Singh, S. (1999). On the Complexity of Policy Iteration. In *Uncertainty in artificial intelligence '99*. Retrieved July 7th, 2006, from www.cs.tau.ac.il/~mansour/papers/99uai.ps

Morris, P. (1994). *Introduction to game theory*. Springer.

Pineau, J., Gordon, G., & Thrun, S. (2003). Policy-contingent abstraction for robust robot control. In Meek, C. & Kjaelruff, U. (Eds.), *Proceedings of the 19th Annual Conference on Uncertainty in Artificial Intelligence*, Acapulco, Mexico. Retrieved July 7th, 2006, from http://www.cs.cmu.edu/~jpineau/files/jpineau-uai03.pdf

Russel, S., & Norvig, P. (2003). *Artificial Intelligence, A modern approach, second edition*. New Jersey: Pearson Education, Inc. (Original work published 1995).

Russel, S., & Tash, J. (1994). Control strategies for a stochastic planner. In *Proceedings of the Twelfth National Conference on Artificial Intelligence* (pp. 1079-1085). Retrieved July 7th, 2006, from http://www.cs.berkeley.edu/~russell/papers/aaai94-mdp.ps

Sutton, R.S., & Barto, A.G. (1998). *Reinforcement Learning*. MIT Press. Retrieved July 26th, 2006, from http://www.cs.ualberta.ca/~sutton/book/ebook/the-book.html

Vrielink, S.B. (2005, December). *A literature study on the MDP environment*. Retrieved July 7th, 2006, from http://members.home.nl/s.h.vrihen/School/MDP_LitStudy.doc

Woodward, R.T. (2006, June). *Markov processes and Burt & Allison 1963*. Retrieved July 7th, 2006, from http://agecon2.tamu.edu/people/faculty/woodward-richard/637/notes/09.pdf

Appendix A: Frequently used terms

This appendix contains an explanation of several frequently used terms in the document.

Dynamic programming

Dynamic programming is a form of reinforcement learning and stands for a collection of algorithms that are able to learn optimal behaviour if a complete model of the environment is given. Examples of dynamic programming algorithms are value iteration and (modified) policy iteration.

Markov Decision Process

A *Markov Decision Process* (MDP) and is a framework for modelling an environment. An MDP can be described by the tuple $\langle S, A, T, R \rangle$, where:

- *S* is a finite set of states of the world.
- *A* is a finite set of actions that can be performed by the agent.
- $T : S \times A \rightarrow \prod(S)$ is the transition function that specifies for an originating state and an action a probability distribution on resulting states. We write $T(s, a, s')$ for the probability that the agents reaches state s', given that the agent performs action *a* in state *s*.
- $R : S \times A \rightarrow \mathbf{R}$ [1] is the reward function that specifies an immediate expected reward if an agent performs an action in a state. We write $R(s, a)$ for the expected immediate reward the agent gains if he performs action *a* in state *s*.

Within an MDP environment, behaviour of the agent is most often described by a *policy* π. The policy maps to each state of the environment a single action. Within MDP environments, the most common goal for agents is to maximize the amount of rewards gained during the lifetime of the agent. With this goal, most learning algorithms try to learn an *optimal policy* that maximizes expected rewards (Russel & Norvig, 2003; Kaelbling et al., 1998; Aberdeen, 2003) elaborate on the MDP framework in more detail. Appendix B explains the process of (modified) policy iteration in an MDP framework and appendix C elaborates on complexity functions learning policies in an MDP modelled environment.

Markov game

A Markov game is an enhancement of the MDP framework to include multiple agents by explicitly modelling secondary agents. This allows for the modelling of complex behaviour of multiple agents in a single environment. Finding the optimal policy becomes somewhat more complicated than in the MDP setting because of the choice that opponents now have. Littman (1994) described this as "In the game theory literature, the solution to this dilemma is to eliminate the choice and evaluate each policy with respect to the opponent that makes it look the worst" (Littman, 1994, p. 2). This is the essence of *minimax*: Behave so as to maximize your reward in the worst case.

We will only consider the two-player game of competing agents because this simplifies the method and it is all that is required for the proposed assignment. The Markov game frameworks can then be described by the tuple $\langle S, A, O, T, R \rangle$, where:

[1] Also $R : S \rightarrow \mathbf{R}$ and $R : S \times A \times S \rightarrow \mathbf{R}$ can be used, but these create no significant differences.

- Instead of a single set of actions *A*, a collection of action sets $A_1, \cdots, A_k$ is given: an action set for each agent in the environment. Because a two-player game is considered, we will use action set *A* for the agent and action set *O* for the opponent.
- The transition function *T* now needs to incorporate for a single transition an action for each agent: $T : S \times A_1 \times \cdots \times A_k \rightarrow \prod(S)$. For the two player game this consists of $T : S \times A \times O \rightarrow \prod(S)$ and we write $T(s, a, o, s')$ for the probability of ending in state *s'* if the agent takes action *a* and the opponent takes action *o*, both from state *s*.
- Instead of a single reward function *R*, each agent has an associated reward function: $R_i : S \times A_1 \times \cdots \times A_k \rightarrow \mathbf{R}$. For a two-player zero-sum game only one reward function can suffice that one agent then tries to maximize while the other tries to minimize it. For the two-player game this becomes $R : S \times A \times O \rightarrow \mathbf{R}$ and we write $R(s, a, o)$ for the expected immediate reward if, from state *s*, the agent takes action *a* and the opponent takes action *o*.

Besides this, the Markov game framework also introduces the stochastic policy $\pi : S \rightarrow \prod(A)$. Given a state, the stochastic policy yields a probability distribution over actions. In this assignment, we will not be using stochastic policies.

Littman (1994) elaborates the Markov game framework into more detail. For an elaboration on policy iteration and complexity issues in a Markov game modelled world, respectively appendix B and C can be examined.

Policy

A policy π is a mapping from each state to a single action:

- $\pi : S \rightarrow A$ for an agent policy.
- $\pi : S \rightarrow O$ for an opponent policy.

$\pi(s)$ is written to indicate the action that policy π prescribes for state *s*. A policy must map a single action to each state in the environment.

Reinforcement learning

The name *reinforcement learning* is used for a collection of AI learning methods that use a reward structure as a means to reinforce desired behaviour. Central to reinforcement learning algorithms are the rewards distributed to the agents inhabiting the modelled environment. Optimal behaviour when using reinforcement learning is behaviour that maximizes accumulated rewards. Examples of reinforcement learning methods are dynamic programming, temporal difference learning and Q-learning.

State utility

Although there are multiple ways to define what compromises the utility of a state, the worded explanation of state utility used in the context of this document would be “The immediate expected reward for being in a state plus the expected discounted utility of the next state, assuming that the agent chooses the optimal action”. Simply put, the state utility is the reward for a state and the discounted future rewards. For a utility maximizing agent in a Markov game environment, Littman (1994) gives the utility of a state by:

$$U(s) = \max_a \min_o \left[R(s, a, o) + \gamma \cdot \sum\nolimits_{s'} T(s, a, o, s') \cdot U(s') \right]$$ (Littman, 1994, p. 3)

Appendix B: Policy Iteration

This appendix elaborates on the (modified) policy iteration algorithm for MDP- and two player zero-sum Markov game environment.

Policy iteration in an MDP environment

As was said in appendix A, an MDP environment can be described by the tuple $\langle S, A, T, R \rangle$. In such environments, the utility of a state $U(s)$ can be described by

$$\textbf{(B1)}\ U(s) = \max_a \left[R(s,a) + \gamma \cdot \sum_{s'} T(s,a,s') \cdot U(s') \right]$$

The problem most learning algorithms for MDP environments try to solve, is how to find an optimal policy π^* for the environment given this definition of state utility and the environment.

The idea behind the policy iteration algorithm, is to begin from a random policy and continually improve this policy until the optimal policy has been found. Each iteration of the PI algorithm consists of two phases: policy evaluation and policy improvement. In the policy evaluation phase, the utility of each state is recalculated by using the current policy. In the policy improvement phase, these new utility values are used to improve the policy. Let π_i be the policy after *i* iterations of PI, then the utility of a state under policy π_i, $U_{\pi_i}(s)$, is given by:

$$\textbf{(B2)}\ U_{\pi_i}(s) = R(s, \pi_i(s)) + \gamma \cdot \sum_{s'} T(s, \pi_i(s), s') \cdot U_{\pi_i}(s')$$

Using these utility values, the policy can be improved by using a one-step greedy look-ahead function with respect to utility: choose the action that has the highest expected utility gain:

$$\textbf{(B3)}\ \pi_{i+1}(s) = \max_a \left[R(s,a) + \gamma \cdot \sum_{s'} T(s,a,s') \cdot U_{\pi_i}(s') \right]$$

This process of policy evaluation and policy improvement is repeated until no change occurs to the policy, $\pi_{i+1} = \pi_i$. If this is the case, then the policy iteration algorithm guarantees the optimal policy is found (Kaelbling, 1996).

Modified policy iteration

A problem with the previously described policy iteration algorithm, is that the computational cost of solving the set linear equations that are created by **(B2)** is high. For that reason, modified policy iteration was created. The idea behind modified policy iteration, is that it might not be required to calculate the utility of each state exactly, but that an approximation to this exact value might yield the same results. Modified policy iteration acquires this approximation by keeping the policy fixed for *k* successive executions of the policy evaluation phase **(B2)**, meaning that the utility of a state under policy π_i is given by:

$$\textbf{(B4)}\ U_{\pi_i}(s) \overset{k}{\leftarrow} R(s, \pi_i(s)) + \gamma \cdot \sum_{s'} T(s, \pi_i(s), s') \cdot U_{\pi_{i-1}}(s')$$

In essence, the policy evaluation phase in the modified policy iteration algorithm consists of *k* iterations of policy evaluation, where in each iteration the previous calculated utility value is used to calculate a new utility value. **(B4)** can be rewritten as the following equations:

$$\textbf{(B5)}\ U_{\pi_i}(s) = U_{\pi_i}^k(s) | k \geq 1$$

$$\textbf{(B6)}\ U_{\pi_i}^j(s) = R(s, \pi_i(s)) + \gamma \cdot \sum_{s'} T(s, \pi_i(s), s') \cdot U_{\pi_i}^{j-1}(s') | 1 \leq j \leq k$$

$$\textbf{(B7)}\ U_{\pi_i}^0(s) = R(s, \pi_i(s)) + \gamma \cdot \sum_{s'} T(s, \pi_i(s), s') \cdot U_{\pi_{i-1}}(s')$$

It can be shown, that if *k* reaches infinity, the perfect approximation is reached, meaning that the result of **(B4)** equals the result of **(B2)** for all states (Woodward, 2006). A problem is to find the minimal *k* value that guarantees that level of approximation.

Modified policy iteration in a Markov game environment

As was said in appendix A, a two-player zero-sum Markov game environment can be described by the tuple $\langle S, A, O, T, R \rangle$. We will only consider this variant of the Markov game, but variants with more action sets or rewards functions work in an analogous way. We will also not consider the use of stochastic policies meaning that only deterministic policies are used. For all equations it must be stated that the agent is trying to maximize utility and the opponent is trying to minimize utility.

By integrating the new opponent action set *O*, the utility of a state in a Markov game environment – which for the MDP environment was given by **(B1)** – becomes respectively for the agent and opponent:

$$\textbf{(B8)}\ U(s) = \max_{a} \min_{o} \left[R(s,a,o) + \gamma \cdot \sum_{s'} T(s,a,o,s') \cdot U(s') \right]$$

$$\textbf{(B9)}\ U(s) = \min_{o} \max_{a} \left[R(s,a,o) + \gamma \cdot \sum_{s'} T(s,a,o,s') \cdot U(s') \right]$$

When using the policy iteration algorithm, the utility of a state under policy π_i - which for the MDP environment was given by **(B2)** – respectively becomes for the agent and opponent:

$$\textbf{(B10)}\ U_{\pi_i}(s) = \min_{o} \left[R(s,\pi_i(s),o) + \gamma \cdot \sum_{s'} T(s,\pi_i(s),o,s') \cdot U_{\pi_i}(s') \right]$$

$$\textbf{(B11)}\ U_{\pi_i}(s) = \max_{a} \left[R(s,a,\pi_i(s)) + \gamma \cdot \sum_{s'} T(s,a,\pi_i(s),s') \cdot U_{\pi_i}(s') \right]$$

and the policy improvement phase – **(B3)** for the MDP framework – is respectively achieved for the agent and opponent by:

$$\textbf{(B12)}\ \pi_{i+1}(s) = \max_{a} \min_{o} \left[R(s,a,o) + \gamma \cdot \sum_{s'} T(s,a,o,s') \cdot U_{\pi_i}(s') \right]$$

$$\textbf{(B13)}\ \pi_{i+1}(s) = \min_{o} \max_{a} \left[R(s,a,o) + \gamma \cdot \sum_{s'} T(s,a,o,s') \cdot U_{\pi_i}(s') \right]$$

Now, altering **(B8)** and **(B9)** to use the modified policy iteration algorithm respectively results in:

$$\textbf{(B14)}\ U_{\pi_i}(s) \overset{k}{\leftarrow} \min_{o} \left[R(s,\pi_i(s),o) + \gamma \cdot \sum_{s'} T(s,\pi_i(s),o,s') \cdot U_{\pi_{i-1}}(s') \right]$$

$$\textbf{(B15)}\ U_{\pi_i}(s) \overset{k}{\leftarrow} \max_{a} \left[R(s,a,\pi_i(s)) + \gamma \cdot \sum_{s'} T(s,a,\pi_i(s),s') \cdot U_{\pi_{i-1}}(s') \right]$$

And it are **(B12)**, **(B13)**, **(B14)** and **(B15)** that will be used to learn policies in the modelled CTF game world.

Appendix C: Computational complexity and cost

This appendix gives an elaboration on the computational complexity and cost functions that are used during the assignment. For the theoretical evaluation of the computational implications of using situational reinforcement learning we will use worst-case upper-bound computational complexity functions. For the empirical evaluation, we will use computational cost value functions that approximate the amount of required calculations. All used functions are meant to give an indication to the amount of arithmetic calculations that are required to learn a policy and the complexity functions only apply to policy iteration algorithms which use a greedy algorithm with respect to utility in the policy improvement phase.

Worst-case upper-bound computational complexities

The upper bound complexity of using a greedy policy iteration algorithm is the product of the upper bound of iterations required to learn a policy and the upper bound computational complexity of a single iteration. Although there is currently no exact upper bound known for the amount of iterations required, according to Mansour & Sing (1999) it is in practice difficult to construct an MDP in which greedy policy iteration takes more steps than the total amount of states *n* (Mansour & Sing, 1999, p. 2). Therefore we will use $O(n)$ as the worst-case upper-bound complexity for the amount of policies required to learn a policy.

But what of the upper bound complexity of a single iteration? Each iteration of policy iteration consists of two phases: policy evaluation and policy improvement. In the policy evaluation phase, a new utility is calculated for each state, by solving the following equation for each state:

$$\textbf{(C1)}\ U_\pi(s) = R(s,\pi(s)) + \gamma \cdot \sum\nolimits_{s'} \left(T(s,\pi(s),s') \cdot U_\pi(s')\right)$$

This results in *n* linear equations, which by using standard linear algebra has an upper bound complexity of $O(n^3)$.

In the policy evaluation phase, a new action is calculated for each state by using a greedy one-step look-ahead function on utility to find the best action, as given by:

$$\textbf{(C2)}\ \pi'(s) = \max_a \left[R(s,a) + \gamma \cdot \sum\nolimits_{s'} \left(T(s,a,s') \cdot U_\pi(s')\right)\right]$$

In **(C2)**, the summation $\sum_{s'}$ sums as many steps as the amount of possibly resulting states from taking action *a* in state *s*. In the worst case, the amount of possibly resulting states equals the total amount of states *n*; Because each action *a* must be evaluated a single time for each state, the upper bound complexity of the policy evaluation phase becomes $O(a \cdot n^2)$.

Add together the upper bound complexity of the policy evaluation phase and the policy improvement phase yields a total upper bound complexity for a single iterations of

$$\textbf{(C3)}\ O(a \cdot n^2 + n^3)$$

, which is confirmed by Kaelbling (1996, p. 15).

Although **(C3)** is the upper bound complexity of a single iteration of the policy iteration algorithm, the evaluated form of situational reinforcement learning uses the modified policy iteration algorithm. The only difference between policy iteration and modified policy iteration is that in the policy evaluation phase, no exact utility is calculated for each state but *k* successive approximating steps are taken for each state, where the policy remains fixed, as explained in appendix B and given by:

$$\textbf{(C4)}\ U'_\pi(s) \overset{k}{\leftarrow} R(s,\pi(s)) + \gamma \cdot \sum\nolimits_{s'} \left(T(s,\pi(s),s') \cdot U_\pi(s')\right)$$

Unlike **(C1)**, this no longer results in *n* linear equations, but simply in *n* equations. The summation still takes at worst *n* steps and for each state the equation must be calculated *k* times, resulting in an upper bound of $O(k \cdot n^2)$. Because the policy improvement phase does not change, the upper bound complexity of a single iteration of modified policy iteration becomes:

$$\textbf{(C5)}\ O(k \cdot n^2 + a \cdot n^2) = O((a+k) \cdot n^2)$$

But this is still not where we want to be. Although **(C5)** applies for modified policy iteration in an MDP framework, it does not hold for a Markov game framework. The policy evaluation and policy improvement phases of modified policy iteration in a Markov game framework are, as explained in appendix B, given by:

$$\textbf{(C6)}\ U'_\pi(s) \overset{k}{\leftarrow} \min_o \left[R(s,\pi(s),o) + \gamma \cdot \sum\nolimits_{s'} (T(s,\pi(s),o,s') \cdot U_\pi(s'))\right]$$

$$\textbf{(C7)}\ \pi'(s) = \max_a \min_o \left[R(s,a,o) + \gamma \cdot \sum\nolimits_{s'} (T(s,a,o,s') \cdot U_\pi(s'))\right]$$

(C6) differs from **(C4)** only in the opponent action *o*; Because the calculation between brackets must be evaluated for each action o, the resulting upper bound complexity for **(C6)** becomes $O(k \cdot o \cdot n^2)$. **(C7)** differs in the same way from **(C2)** and the upper bound complexity there thus becomes $O(a \cdot o \cdot n^2)$. The total upper bound complexity of a single iteration of modified policy iteration in a Markov game framework can thus be given by:

$$\textbf{(C8)}\ O(k \cdot o \cdot n^2 + a \cdot o \cdot n^2) = O((a+k) \cdot (o \cdot n^2))$$

By also taking the worst-case upper-bound amount of iterations into account, $O(n)$, the worst-case upper-bound complexity of learning an optimal policy using greedy modified policy iteration in a Markov game framework becomes:

$$\textbf{(C9)}\ O(n \cdot (a+k) \cdot (o \cdot n^2)) = O((a+k) \cdot (o \cdot n^3))$$

CTF computational cost

Besides the theoretical worst-case upper-bound complexities of the various learning methods, an indication will also be given on the difference between the methods with respect to the average computational cost required to learn a policy explicitly for the modelled CTF world when using the modified policy iteration algorithm. This difference in cost will be calculated by comparing average computational cost values *C*. These values should give an indication on the average amount of calculations required to learn a policy.

Just like with the previously described upper-bound complexity, but then averaged, the complexity value *C* = average amount of iterations * computational cost of a single iteration. The average amount of iterations is given by i_{avg} and is derived from empirically learning policies. The computational cost of a single iteration is derived from the modified policy iteration in a similar fashion as was done in **(C8)** for the upper bound case, but then without the worst-case assumption. This results in one difference with **(C8)**: In the upper bound case, the summation in **(C6)** and **(C7)** requires at most *n* steps, the worst case. For the CTF world, this is never the so: at most there are 8 possibly resulting states (when a player returns from being dead into the game), but with the bulk of the state-action pairs there are but 1 or 2 possibly resulting states. Because 8 << *n*, which lies around 150000, the amount of steps required by the summation is ignored by the computational cost value. The computational cost values that indicates the average amount of calculations required to learn an optimal policy in the CTF world or a situation therein when using the modified policy iteration algorithm thus becomes:

$$\textbf{(C10)}\ C = i_{avg} \cdot (k \cdot o \cdot n + a \cdot o \cdot n) = (a + k) \cdot (o \cdot n \cdot i_{avg})$$

For the evaluation of computational cost, we will not use the empirically found i_{avg} directly, but rather a corrected amount $\hat{i}_{avg}$. This corrected amount $\hat{i}_{avg}$ is the found i_{avg} decremented by the intuitive save *t* value that was used to assure that the optimal policy was found, as is explained in appendix E. The reason that $\hat{i}_{avg}$ is decremented by the *t* value, is because we are interested in the amount of iterations required to learn an optimal policy, not in the amount of iterations required to be certain that we have acquired the optimal policy. Because the *t* values were chosen in such a manner that we assume that the optimal policy has been found, the corrected amount of iterations $\hat{i}_{avg}$ represents for each environment the average minimal amount of iterations required to learn the optimal policy. By using the corrected average amount of iterations, it is not required to use the minimal save *t* value explained in appendix E, t_{save}, for each environment but any $t \geq t_{save}$ value can be used without influencing the evaluation complexity function.

Because we are interested in the difference in average computational cost between the methods, all variables from **(C10)** that remain the same between methods can be eliminated. By removing these variables *a*, *o* and *k* and also introducing the corrected amount $\hat{i}_{avg}$, **(C10)** can be written as:

$$\textbf{(C11)}\ C = \hat{i}_{avg} \cdot n$$

These last two variables $\hat{i}_{avg}$ and *n* may not be removed, because they differ between methods. The average amount of iterations is dependant on several factors, two of which are the initial policy and the size of the environment. If the initial policy is more optimal, then it is likely that less iterations are required because the initial policy required less ‘improvement’. If the environment has a larger state set, it is likely that more iterations are required because policy improvement influencing state rewards may come from farther states and thus require more iterations to iterate through the environment. Because the *Combined* method uses environments with smaller state sets and the *Enhanced* method uses a (probable) more optimal starting policy, the variables *n* and $\hat{i}_{avg}$ may not be removed from **(C10)**.

If **(C11)** is applied to the three methods, the following equations for average complexity are derived, and these are used as part of the empirical evaluation:

$$\textbf{(C12)}\ C_{complete} = \hat{i}_{avg,complete} \cdot n$$

$$\textbf{(C13)}\ C_{combined} = \sum_{\theta \in \Theta} \left(\hat{i}_{avg,\theta} \cdot n_{\theta}\right)$$

$$\textbf{(C14)}\ C_{enhanced} = C_{combined} + \left(\hat{i}_{avg,enhanced} \cdot n\right)$$

Within these three equations, the only true unknown is $\hat{i}_{avg}$ because the values of *n* and all n_{θ} are fixed by the CTF environment. Because the CTF average complexity values are indicating values for comparison and not ‘*order of*’ $O(\cdots)$ notations as used in the previously described upper bound complexities, these constants values may not be removed. In words, they serve to indicate the cost of a single iteration of modified PI. A single iteration of modified PI has a greater cost in a larger environment, because the calculations must be performed for more states.

Appendix D: The CTF game world

This appendix explains the various features that belong to the “Capture the flag” game. An overview of the world is given and the rules of the game will be explained. Also the variables that define the state of the world will be given as well as the available actions and some examples of possible transitions. The reward structure for the CTF world is elaborated on in the last paragraph.

A world overview

The CTF game that will be played contains two players that will compete against each other. The player will be referred to as the “Agent” and as the “Opponent”. The game environment contains the following elements:

- 8 valid locations where each player can move to. Each location is determined by two coordinates (x, y), where $(0,0)$ is the bottom left corner and $(2,2)$ the top right corner.
- 1 wall location that is inaccessible.
- 1 of the 8 valid locations is called the “Agent Flag Starting Position”.
- 1 of the 8 valid locations is called the “Opponent Flag Starting Position”.
- 1 flag belonging to the agent, called “Agent Flag”. The location of the flag is given by the state variable “Agent Flag Location”.
- 1 flag belonging to the opponent, called “Opponent Flag”. The location of the flag is given by state variable “Opponent Flag Location”.

Figure D.1 shows the world from a top view with no players placed in it.

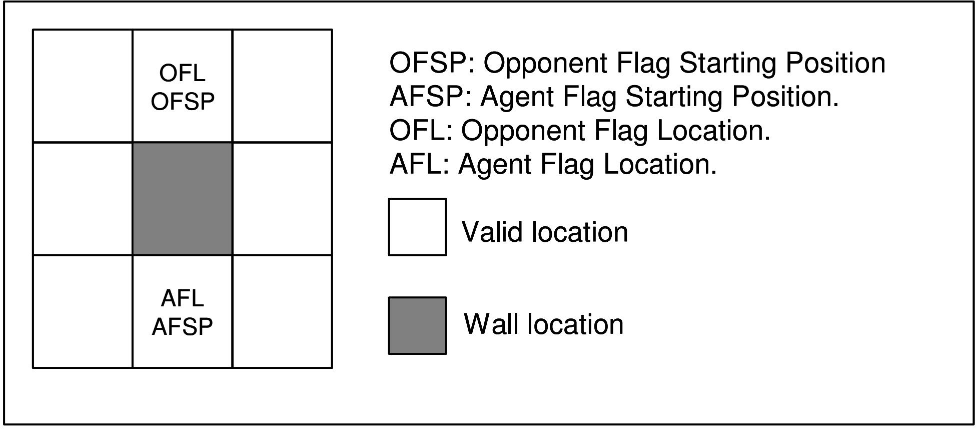

OFL
OFSP
AFL
AFSP
OFSP: Opponent Flag Starting Position
AFSP: Agent Flag Starting Position.
OFL: Opponent Flag Location.
AFL: Agent Flag Location.
Valid location
Wall location

Figure D. 1: The CTF world with no players

Rules of the game

The game has the following rules:

- The players take simultaneous actions. It is impossible for one player to take an action and for the other player not to do so. A player may choose to do nothing, but doing nothing in such a fashion is seen as an action. Any inconsistencies that may arise from this will be handled explicitly in the rules of the game.
- Each player can navigate the valid locations of the game world. An agent can move one space in any horizontal or vertical direction, if a valid location is in that direction. A player cannot move diagonally. It is allowed for multiple agents to occupy the same location.

- Each player can score a point by returning the flag of the opponent to the starting point of the agent's own flag. If a point is scored, the opposing flag is returned to it's starting location. The number of points scored in a game are recorded and define a way to end the game.
- Each player can pickup a flag, if that flag is on the same location as the player and is not already carried by the player himself or the opposing player. If the own flag is picked up in such a fashion, the flag is transported back to the starting location of the player. If it concerns the opposing flag, the player now carries the flag and the flag will thus move along with the agent.
  - If both players try to pickup the same flag simultaneously, the player to which the flag belongs has precedence.
  - If both flags are on the same location, not being carried, and a player performs a pickup, then the flag belonging to that player is always picked up first.
- Each player can attack the opposing player if the players are on the same location. If either player attacks, there must first be one of the following two possible outcomes: the agent or the opponent dies. One of the players must die. The probability of the agent dying depend on the following values:
  - The base chance of dying is 0.5.
  - If the agent/opponent holds the flag, the chance of dying is increased/decreased by 0.3.
  - If the agent/opponent performed the attack action, the change of dying is decreased/increased by 0.1.

  The chance that the opponent dies is 1 minus the chance the agent dies. For all actions, the attack has precedence. If for example, a player tries to attack, whilst the other player's action was to move away from the attacking player (which would make an attack illegal), the attack takes precedence. Should a non-attacking player remain alive, the effect of the action taken by that player still occurs.
- A player that has dies will be brought back into the game on a random location in the game after two actions of the player that is still alive, as will be explained further on with the state variables.
- The game can end in three ways:
  - At least one of the players scored the maximum amount of points.
  - Both players choose to do nothing in two consecutive turns.
  - A to be defined amount of time has expired.

State of the world

The state of the world is defined by 9 state variables $\langle AL, OL, AFL, OFL, AS, OS, AP, OP, GS \rangle$ which are:

- *AL* of "Agent Location", which represents the location of the agent. Possible values are all valid locations.
- *OL* of "Opponent Location", which represents the location of the opponent. Has the same possible values as *L*.
- *AFL* of "Agent Flag Location", which represents the location of the "Agent Flag". Possible values are all valid locations.
- *OFL* of "Opponent Flag Location", which represents the location of the "Opponent Flag". Possible values are all valid locations.
- *AS* of "Agent Status", which represents the status of the agent. Possible values are 'Normal', "Carrying Flag", 'Dead2' and 'Dead1'.

- *OS* of “Opponent Status”, which represents the status of the opponent. Has the same possible values as *AS*.
- *AP* of “Agent points”, which represent the amount of points scored by the agent. In the modelled CTF world, a maximum of 2 points can be scored by a player.
- *OP* of “Opponent points”, which represent the amount of points scored by the opponent. Has the same possible values as *AP*.
- *GS* of “Game Status”, which represent the status of the game. Possible values are ‘Normal’, ‘Idle’, ‘Deadlock’.

Modelling this world reveals a total of 1769472 unique states. When taking the actions, transitions and game rules into consideration, there is a total of 136737 reachable states.

Available actions

The action set of the agent *A* is the same as the action set for the opponent *O*. Each action set has 8 possible actions:

- *DoNothing*: The player does nothing.
- *Up*: The player moves one space up. This action is possible if there is a valid location north of the player.
- *Down*: The player moves one space down. This action is possible if there is a valid location south of the agent.
- *Left*: The player moves one space to the left. This action is possible if there is a valid location west of the player.
- *Right*: The player moves one space to the right. This action is possible if there is a valid location east of the player.
- *PickUp*: The player picks up a flag. A pickup is possible if the player and a flag are on the same location and the player is not already carrying the flag it wants to pick up. If the own flag is picked up, it is transported back to the player’s flag starting position. If the other player’s flag is picked up, the status of the player is set to “Carrying flag” and the opposing flag now moves along with the player.
- *Score*: The player scores a point. A score is possible if the player is carrying the opposing flag and is at the player’s flag starting position. After a point has been scored, the opposing flag is immediately transported back to the opposing flag starting position. After a player has scored, the points of that player is increased by one.
- *Attack*: The player attacks the other player. This action is possible if the agent and opponent are on the same locations. If an attack occurs one of the player must die (status changed to ‘Dead2’).

Transitions

The transitions and their corresponding probability that are specified by the transition function can be derived from the world states, available actions and the game rules. All actions create deterministic effects, except for the attack action and the returning of a dead player. Transitions that have not yet been described by the rules or action effects are:

- Time is not explicitly modelled in the game, and the players are not aware of the time end condition.
- All states where either of the other two end conditions are met, are absorbing states.
- For the *GS* state variables, the following holds:
    - If the *GS* variable has the ‘Normal’ value and both players are alive and both players choose to do nothing, the *GS* variables is set to ‘Idle’.

- If the *GS* variable has the ‘Idle’ value and both players choose to do nothing, the *GS* variable is set to ‘Deadlock’ and the game ends in a tie.
- In all other cases, the *GS* variable is set to ‘Normal’.

Because there are an enormous amount of possible (probability >0) transitions, we will not give them all here, but give some example transitions of interesting situations. To keep the transitions simple, the following notations are used:

- ? means that there are multiple possibilities, but that these possibilities are not really interesting for the example.
- λ means “any action”.
- As a reminder: $T(s,a,o,s')$ was written for the probability of ending in state *s'* if, from state *s*, the agent takes action *a* and the opponent takes action *o*.
- A state was given by the tuple $\langle AL, OL, AFL, OFL, AS, OS, AP, OP, GS\rangle$.

A completely out written example where both players move, and the agent is carrying the flag:

$$T\left(\begin{array}{l}((0,0),(0,2),(1,2),(0,0),Carrying\ Flag, Normal, Zero, Zero, Normal),\\ Right, Down,\\ ((1,0),(0,1),(1,2),(1,0),Carrying\ Flag, Normal, Zero, Zero, Normal)\end{array}\right)=1.0$$

An example of an attack when no player is carrying the flag:

$$T\left(\begin{array}{l}((0,0),(0,0),?,?,Normal, Normal,?,?,?),\\ Right, Attack,\\ ((1,0),(0,0),?,?,Normal, Dead2,?,?,?)\end{array}\right)=0.4$$

$$T\left(\begin{array}{l}((0,0),(0,0),?,?,Normal, Normal,?,?,?),\\ Right, Attack,\\ ((0,0),(0,0),?,?,Dead2, Normal,?,?,?)\end{array}\right)=0.6$$

Some examples of flag pickups:

$$T\left(\begin{array}{l}((0,0),(0,0),(0,0),(1,2),Normal, Normal,?,?,?),\\ PickUp, PickUp,\\ ((0,0),(0,0),(1,0),(1,2),Normal, Normal,?,?,?)\end{array}\right)=1.0$$

$$T\left(\begin{array}{l}((0,1),(0,1),(0,1),(0,1),Normal, Normal,?,?,?),\\ PickUp, PickUp,\\ ((0,1),(0,1),(1,0),(1,2),Normal, Normal,?,?,?)\end{array}\right)=1.0$$

$$T\left(\begin{array}{l}((0,0),?,?,(0,0),Normal,?,?,?,?),\\ PickUp,?,\\ ((0,0),?,?,(0,0),Carrying\ Flag,?,?,?,?)\end{array}\right)=1.0$$

An example of a score:

$$T\begin{pmatrix}((1,0),?,?,(1,0),\mathit{Carrying\ Flag},?,\mathit{Zero},?,?),\\ \mathit{Score},?,\\ ((1,0),?,?,(1,2),\mathit{Normal},?,\mathit{One},?,?)\end{pmatrix}=1.0$$

An example of successive transitions when the agent is dead:

$$T\begin{pmatrix}(?,?,?,?,\mathit{Dead}2,?,?,?,?),\\ \lambda,?,\\ (?,?,?,?,\mathit{Dead}1,?,?,?,?)\end{pmatrix}=1.0$$

$$T\begin{pmatrix}(?,?,?,?,\mathit{Dead}1,?,?,?,?),\\ \lambda,?,\\ ((0,0),?,?,?,\mathit{Normal},?,?,?,?)\end{pmatrix}=\frac{1}{8}$$

$$\vdots$$

$$T\begin{pmatrix}(?,?,?,?,\mathit{Dead}1,?,?,?,?),\\ \lambda,?,\\ ((2,2),?,?,?,\mathit{Normal},?,?,?,?)\end{pmatrix}=\frac{1}{8}$$

Rewards

Situational reinforcement learning assumes a decomposition of the reward function *R* into an action reward function *AR* and a state reward function *SR*, as was given by $R(s,a,o)=AR(a,o)+\sum_{s'}T(s,a,o,s')\cdot SR(s')$.

The action reward *AR* for the modelled CTF world is the sum of the reward of both actions and can be summarised as being:

- The actions '*DoNothing*', '*Up*', '*Down*', '*Left'*, '*Right'* and '*Attack'* if executed respectively by the agent and opponent have a reward of -0.05 and +0.05.
- The actions '*PickUp'* and '*Score'* if executed respectively by the agent and opponent have a reward of -0.02 and +0.02.

The state reward of a state in the CTF world depends on the values of the state variables. All states where the *GS* variable has the value 'DeadLock' have a state reward of 0, no matter the other variables. The state reward of all other states is the sum of the following rules:

- The value "Carrying Flag" for *AS* or *OS* has a respective reward of +2 and -2.
- The values 'Dead2' and 'Dead1' for *AS* or *OS* has a respective reward of -2 and +2.
- The reward of *AP* and *OP* is respectively 10 times and -10 times the value of *AP* and *OP* (If for example *AP* is 1, it's addition to the state reward is $1\cdot 10=10$).

Using this state reward structure, a total of 21 game situations with reachable states can be derived with the state rewards -22,-20,-18,-14,-12,-10,-8,-6,-4,-2,0,2,4,6,8,10,12,14,18,20,22.

Appendix E: Modified policy iteration variables

This appendix elaborates on the variables used in the modified policy iteration algorithm and how these values will be set for the evaluation of SRL. In the modified policy iteration algorithm, there are two variables that are not defined by the environment, but that must be set by the developer: The discount factor γ and the approximation variable *k*. First, the influence of both variables will be explained, after which the values are explained that will be used for evaluation.

Approximation value k and termination value t

The most common termination criteria for policy iteration, is to terminate if no change occurs in the policy during the policy improvement phase. The policy iteration algorithm can guarantee that the optimal policy is found if this termination criteria is reached (Kaelbling, 1996). Modified policy iteration uses the approximation variable *k* to derive a certain degree of approximation to conventional policy iteration. If *k* is chosen high enough, a perfect approximation is realised. This means that the policy evaluation phase of PI and modified PI would yield the same utility values and that the modified PI algorithm can make the same guarantee of optimality as the PI algorithm (Woodward, 2006). Unfortunately, no exact method can be given to determine the minimal value of *k* required to reach that level of approximation. It is known that *k* grows linearly in γ and that if *k* reaches infinity, the approximation becomes perfect (Woodward 2006). Should a value of *k* be chosen that leads to a less than perfect approximation, the PI algorithm could converge to a local[1] optimum of utilities, resulting in a sub-optimal policy if the same termination criteria would be used. Figure E.1 and table E.1 depicts this for a simple environment where all actions have deterministic effects, all transitions are depicted by arrows, all rewards for reaching a state are written below the state name and where we are only interested in the action for state s_2. The obvious optimal policy for this environment is $\dot{\pi} = R$. If the initial policy is the sub-optimal policy $\pi_o = L$ and the minimal $k = 1$ is chosen, the first iteration reveals no policy change because the policy iteration algorithm for state s2 converges to the local (s_1, s_2, s_3) optimal utility value of 3 instead of the global $(s_1, s_2, s_3, s_4, s_5)$optimal utility value of 10. If the conventional termination criteria would be used, the found policy would be sub-optimal. If a higher *k* value is chosen, for example 3, a better approximation is made and the optimal policy is found even if the termination criteria of one unchanging policy is used.

By increasing the required amount of iterations where the policy improvement phase yields no change, henceforth called the termination value *t*, PI is given the chance to escape local optimal policies if a value of *k* is chosen that gives a sub-optimal approximation. The example of figure E.1 and table E.1 also depict this: even if low value of *k* is chosen, if *t* is high enough the optimal policy can still be found. As with *k*, there is also no method available to determine a value of *t* that guarantees an optimal policy.

[1] The word 'local' in the appendix does not refer to a situation, but rather refers to a sub-set of the global environment.

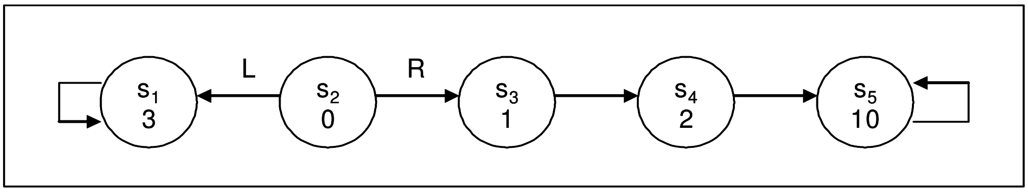


Figure E. 1 Simple example world with deterministic actions and no action rewards

Modified PI with $R(s)=[3 \mid 0 \mid 1 \mid 2 \mid 10]=U_{\pi_{-1}}$ **,** $\pi_0=L$ **and** $\gamma=0.8$	
$k=1$	$k=3$
1: $U_{\pi_0}=[5.4 \mid 2.4 \mid 2.6 \mid 10 \mid 18]$ $\pi_1=L=\pi_0$	1: $U_{\pi_0}^1=[5.4 \mid 2.4 \mid 2.6 \mid 10 \mid 18]$ $U_{\pi_0}^2=[7.3 \mid 4.3 \mid 9.0 \mid 16.4 \mid 24.4]$ $U_{\pi_0}^3=[8.9 \mid 5.9 \mid 14.1 \mid 21.5 \mid 29.5]$ $\pi_1=R\neq\pi_0$
2: $U_{\pi_1}=[7.3 \mid 4.3 \mid 9.0 \mid 16.4 \mid 24.4]$ $\pi_2=R\neq\pi_1$	2: $U_{\pi_0}^1=[10.1 \mid 11.3 \mid 18.2 \mid 25.6 \mid 33.6]$ $U_{\pi_0}^2=[11.1 \mid 14.6 \mid 21.5 \mid 28.9 \mid 36.9]$ $U_{\pi_0}^3=[11.9 \mid 17.2 \mid 24.1 \mid 31.5 \mid 39.5]$ $\pi_2=R=\pi_1$
3: $U_{\pi_2}=[8.9 \mid 7.2 \mid 14.1 \mid 21.5 \mid 29.5]$ $\pi_3=R=\pi_2$	$\vdots$

Table E.1. Modified PI performed for 3 iterations with k=1 and k=3 on MDP of figure E.1

Discount factor γ

The discount factor γ, which has a value between 0.0 and 1.0, defines the weight of future rewards. Each discount factor creates a different optimal policy in the environment. In general, higher discount factors require more iterations of modified policy iteration to terminate because:

1. In (modified) PI, the influence of the reward of a certain state into the utility of other states iterates further into the environment with each iteration. Because a higher discount factor means that the influence of future rewards is greater, the chance is also greater that such a future reward chances the policy. This means that it will generally take more iterations to find an unchanging policy with a higher discount factor.
2. Because *k* grows linearly in γ (Woodward, 2006), a higher discount factor with an equal value of *k* results in worse approximations, which in turn means that more iterations are required to find the optimal policy.
3. The number of iterations required to reach the optimal value function is polynomial in the number of states and the magnitude of the largest reward if the discount factor is held constant. However, in the worst case the number of iterations grows polynomially in $\frac{1}{(1-\gamma)}$, so the convergence rate slows considerably as the discount factor approaches 1 (Kaelbling, 1996).

For the modelled CTF world, with $t=4$ and $k=1$, this is also demonstrated in figure E.2 for the *Standard Complete* and *Simple Complete* environments.

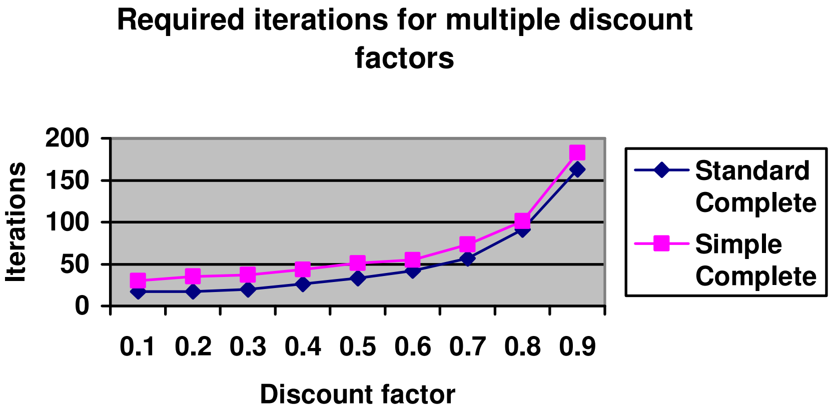


Figure E. 2. Required iterations for multiple discount factors with t=4 and k=1

Choosing the variables

A goal of the assignment is to evaluate the implications SRL on optimality and computational cost. In order to evaluate the implication of the *Combined* method on optimality by comparing the learned policies with learned policies of other methods through games of CTF, it must be made sure that the local policies learned by the *Combined* method have an equal degree of optimality as the global policies learned by the other methods. Should this not be the case, then any difference in optimality of the global policies between methods could also be the result of the initially learned local policies and not solely because of the methods. Choosing the same values of *k* for all methods would not be fair, because the value of *k* in order to obtain an equal degree of approximation is undoubtedly related to the size of the state set, and this size differs between the *Combined* method and other methods. Because there is no method available that, for a certain value of *k* and a certain environment, guarantees a certain degree of approximation, choosing a value of *k* for each environment is a problem. As was said previously, a higher value of *t* could still enable PI to find the optimal policy even if *k* is chosen to give a sub-optimal approximation. Finding a *t* value for each environment that enables the optimal policy to be found thus also ensures an equal degree of optimality. Unfortunately, there is also no method available that defines a *t* value that guarantees that the optimal policy can be found.

Because we are unable to find a method that guarantees an equal degree of optimality of the learned policies within their environments, we will use a method that assumes that the methods have found the optimal policy. We will use the *t* value to determine this optimal policy, keeping the value of *k* to be the minimal 1. By empirically testing the environments for multiple high values of *t* and analysing the longevity of found local optimal policies (iterations where no change is found), intuitive save *t* values can be found. Figure E.3 depicts this for the *Simple Complete* environment: The longest local optimal policy found lasted for 6 iterations and a *t* value of 1000 was used. When the algorithm terminated, there is one of two possibilities:

1. The optimal policy has been found.
2. A local optimal policy has been found that lasts for more than 1000 iterations.

Although no guarantees can be given that the first possibility has become reality, it seems unlikely that the second possibility happened, because the longest found other local optimal policy lasted for 6 iterations, which is far smaller than the *t* value of 1000 that is used. For each environment, intuitive save *t* values will be found and used. In words, the assumption that will be made is “A policy learned with the intuitive save t value, t_{save}, is optimal if the longest found sequence of iterations without policy change, $i_{\max}$, is no more than 5% of t_{save}"[1]. For the *Simple Complete* world of figure 6, where $i_{\max} = 6$, the learned policy would be considered optimal if the used *t* value was greater than or equal to 120.

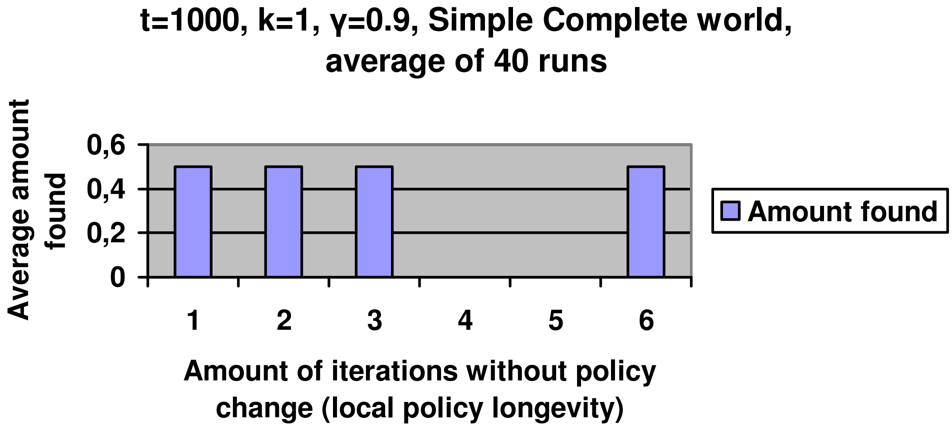


Figure E. 3 Average amount of local policies found per longevity.

Each discount factor creates a different optimal policy for the environment. Higher discount factors means that, from the perspective of a single state, rewards of states that lie farther away have a greater influence in the state utility. Because in games it is often better to think as far ahead as possible, a higher discount factor will in general result in policies that have a better game performance. As explained previously, it is also true in generals that a higher discount factors requires more iterations to find the optimal policy, meaning that a higher discount factor results in higher computational costs if the environment is held constant. Because it can thus be said that in general the effect of a higher discount factor results in better game performance and higher computational costs, three discount factors will be used for the evaluation that should give an indication for all discount factors. The evaluation will therefore use a low discount factor of 0.1, a high discount factor of 0.9 and a discount factor in between of 0.5.

[1] 5% is used because this value is often used in statistics as a border value for statistical significance

Appendix F: The developed program

This appendix elaborates on the developed program. This appendix will not go into programming details – the source code of the program can be referenced for this – and will only elaborate on the features that the program provides. The goals of the program are:

- To provide a practical application of situational reinforcement learning by modelling the game of CTF as described in appendix D.
- To use the modelled environment as a platform for the empirical evaluation as described in paragraph 4.2.

As such, the program provides the following features that will be explained in the upcoming paragraphs:

- Enable policy learning with various discount factors (0.1, 0.5 and 0.9), environments (*Standard*, *Alternative* and *Simple*) and learning methods (*Combined*, *Enhanced* and *Complete*).
- Compute average computational cost values for the learned policies as explained in paragraph 4.2.5.
- Allow learned policies to compete in games of CTF against each other, random policies and human players.

The program – which is written in the Java programming language – is present on a CD that is delivered along with this document. On this CD, the following items are present:

- A markovgame(compiled).rar file that contains the compiled version of the developed program.
- A markovgame(uncompiled).rar file that contains the uncompiled java source files of the developed program.
- A markovgame directory that contains both the compiled and uncompiled java source files of the developed program.
- A policies.rar file that contains the learned policies that were used in the empirical evaluation described in this document.
- A markovgame.jar package file that contains a compiled version on the developed program.
- A markovgame.bat file that can be used to start the program. The upcoming paragraph explain more on how to start the program.
- A version of this document.

In the upcoming paragraph an explanation is given on how the program can be started. How to access the various features of the program is explained in the paragraphs thereafter. The final paragraph elaborates on the files that are edited by the program.

Starting the program

The developed program has been written in the Java programming language. The Java runtime environment version 1.4.2 or higher is required in order to compile and start the program. In order for all features of the program to work, at least 256 MB of RAM memory must be available to the program. Besides this amount of internal memory, it might be prudent to have at least 500 MB of hard drive space available.

Because the program performs file editing – as will be explained in the last paragraph – the program must be run in an environment where it is allowed to read and write files. The program can thus not be run from the CD on which it is delivered. The entire program is written as a single Java package: the *markovgame* package. In order to start the program, one must first decide a working directory from which to run the program, let's assume this working directory to be "C:\SRL\". Besides the *markovgame.bat* file, either of the following must be copied into the working directory:

1. The markovgame directory that is present on the CD.
2. The markovgame directory that resides compressed in the markovgame(compiled).rar file on the CD.
3. The markovgame.jar file that is present on the CD.

After this is done, the markovgame.bat file can be edited accordingly your system. The file contains the following as default:

- `java -Xmx256m -cp . markovgame.CTF_MainEngine`

and this is sufficient if the java run-time environment is present in your system path and you copied either of the two directories. If the java run-time environment in not present in your system path, then the line must be edited accordingly. A possible alteration could be:

- `C:\Program Files\Java\bin\java.exe -Xmx256m -cp . markovgame.CTF_MainEngine`

If you did not copy either of the two directories, but rather the markovgame.jar file, then the class-path must be redirected to this file explicitly in the following manner:

- `java -Xmx256m -cp markovgame.jar markovgame.CTF_MainEngine`

The argument *–Xmx256m* is required in order for the java virtual machine to assign sufficient internal memory for all program features to work. The *CTF_MainEngine* class is the main class for the program and it can take two arguments. These arguments define the width and height of the frame used by the program. If you desire another dimension in pixels for your frame than the default dimension, you can alter it accordingly, for example in:

- `java -Xmx256m -cp . markovgame.CTF_MainEngine 1024 768`

After the markovgame.bat file is edited to your specific system, you can run it and the following screen should appear:

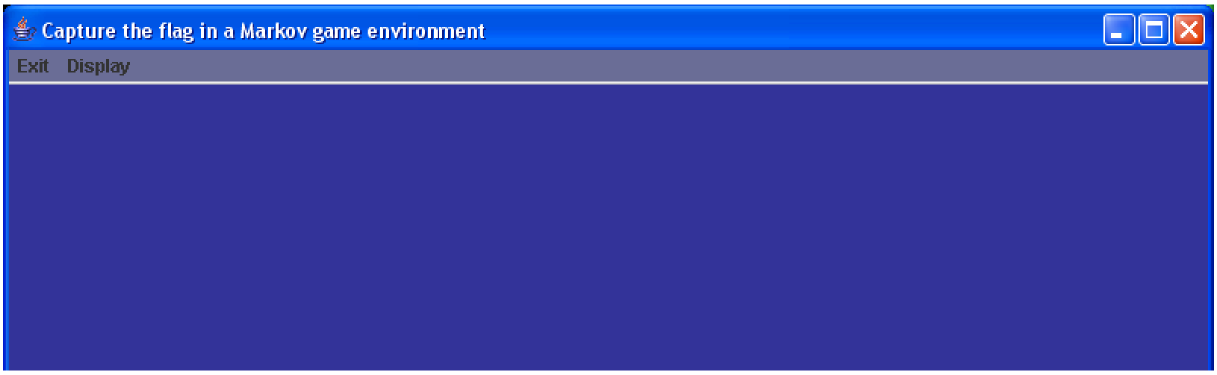


Figure F. 1. The program starting screen.

You have succeeded in starting the program. If at any time you wish to close the program, you can choose *Exit* from the menu bar or click on the ☒ in the upper-right corner of the program frame. If you wish to use a program feature, you can select *Display* from the menu bar and choose the desired feature. Each feature will be explained in the following paragraphs.

Learning policies

In order to learn policies, you can select the *Learning* item from the *Display* menu in the menu bar – as displayed in figure F.2 – which will result in the displaying of figure F.3.

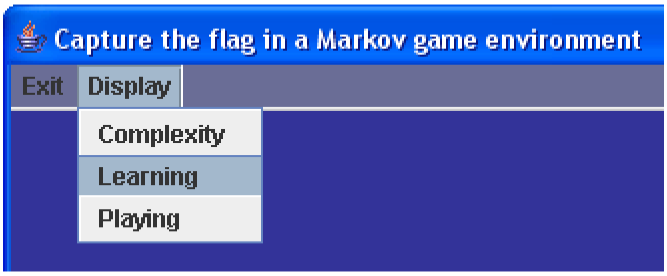
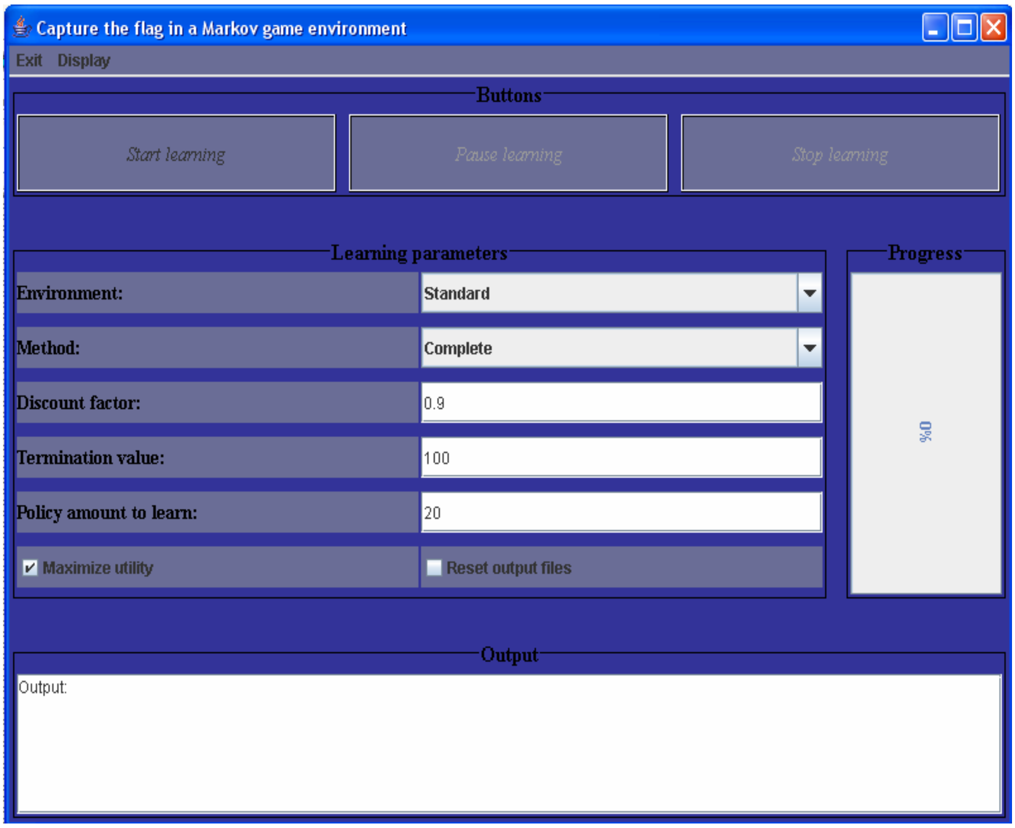


Figure F. 2. Selecting the learning feature.



Environment: Standard

Method: Complete

Discount factor: 0.9

Termination value: 100

Policy amount to learn: 20

☑ Maximize utility

□ Reset output files

Output:

Figure F. 3. The learning feature screen.

In this screen, you can input the learning parameters you desire. The following parameters must be given:

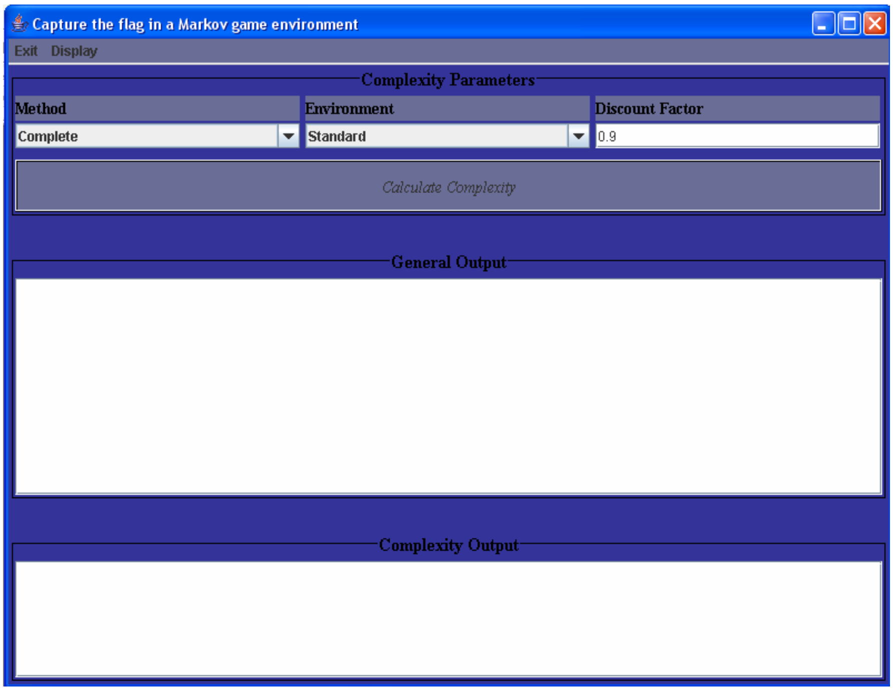
- **Environment**: you must choose for which environment a policy must be learned. This can either be *Standard*, *Alternative* or *Simple*.
- **Method**: you must choose which learning method is used to learn a policy. This can either by *Complete*, *Combined* or *Enhanced*. The *Enhanced* method can only be used if *Combined* policies with the same parameters have already been learned and written to file.
- **Discount factor**: you must choose which discount factor to use in the learning process. A discount factor must be a value between 0.00 and 1.00.
- **Termination value**: you must choose the *t* value – which is explained in appendix E. The termination value represents the amount of iterations of unchanging policies are required to terminate the learning process. If during the learning process a local optimum is found that more iterations than 5% of the termination value, then the learning process of the current policy is aborted. This is explained in appendix E as to enable the assumption that the optimal policy is found. The termination value must be a value of 1 or higher.

- **Policy amount to learn**: you must choose the amount of policies that must be learned. If all parameters are set and the learning has started, the progress bar at the right of the frame gives the percentage of policies for which learning is completed.
- **Maximizing utility**: If maximizing utility is enabled then the learned policy is a policy for the agent. If it is disabled, the learned policy is for the opponent. In order for policies to be used in game playing, both agent and opponent policies must have been learned.
- **Reset output files**: If this is enabled, then previously learned policies with the same parameters are overwritten with the newly learned policies. If it is disabled, the newly learned policies are written besides the existing ones.

After all parameters are set, the *Start Learning* button can be pushed to initiate the learning process. The *Output* field will give program output. If the *Pause learning* or *Stop learning* buttons are pushed, the corresponding action will occur after the current learning process is completed. Depending on the learning parameters and your system, it might take several minutes to half an hour for a single learning process to complete. If the learning process is paused, it can be resumed again at a later stage. If the learning process is interrupted, for example by a program shutdown or by clicking *Stop learning*, then no policy is stored to file. If the process completes, then all policies for which the optimal assumption has been met – see appendix E – are written to file.

Calculating computational cost

In order to calculate computational costs, you can select the *Complexity* item from the *Display* menu in the menu bar in a similar fashion as figure F.2. This will result in the displaying of figure F.4.



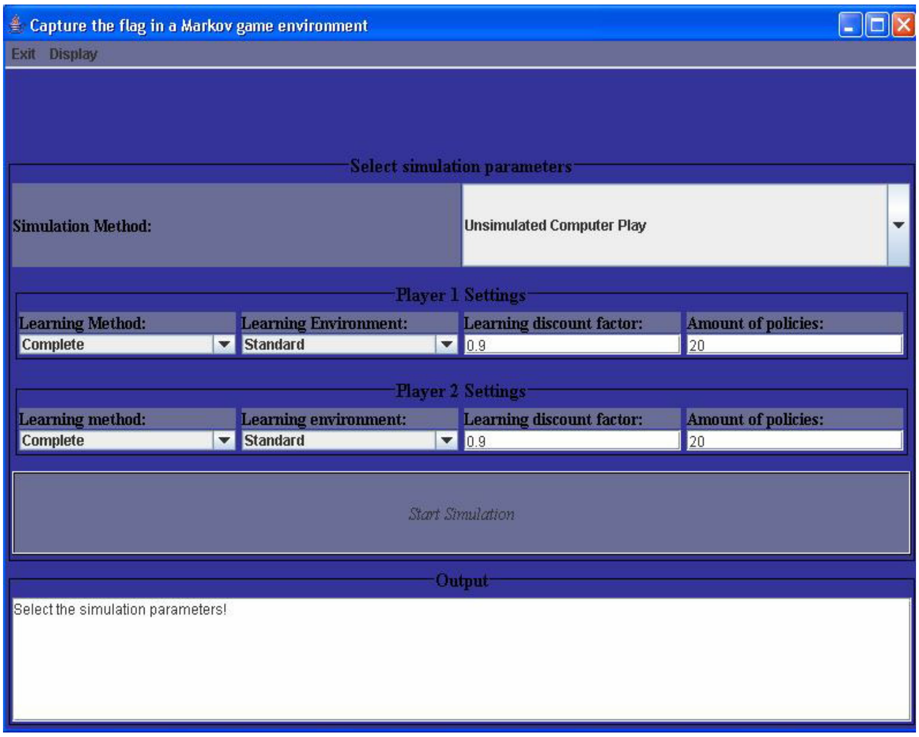
Method	Environment	Discount Factor
Complete	Standard	0.9

Figure F. 4. The computational cost feature screen.

In this screen, you must input the parameters that define the policies for which you wish to calculate the computational cost. The calculation uses <u>all</u> policies that were learned with the given parameters. In order to calculate the computational cost values, there must be at least one learned policy with the given parameters. In the *General Output* area, more detailed information is displayed of the last calculation. Each new calculation resets this output area. The *Complexity Output* area only displays the resulting computational cost values and does not reset when a new calculation is started, allow for easy comparison between multiple calculations.

Playing games

In order to play games, you can select the *Playing* item for the *Display* menu in the menu bar in a similar fashion as illustrated in figure F.2. This action will result in the displaying of figure F.5.



Select simulation parameters

Simulation Method: Unsimulated Computer Play

Player 1 Settings

Learning Method:	Learning Environment:	Learning discount factor:	Amount of policies:
Complete	Standard	0.9	20

Player 2 Settings

Learning method:	Learning environment:	Learning discount factor:	Amount of policies:
Complete	Standard	0.9	20

Output

Select the simulation parameters!

Figure F. 5. The playing feature initial screen.

In this screen you must select how you wish the game to be simulated and which players should compete. The following parameters must be set:

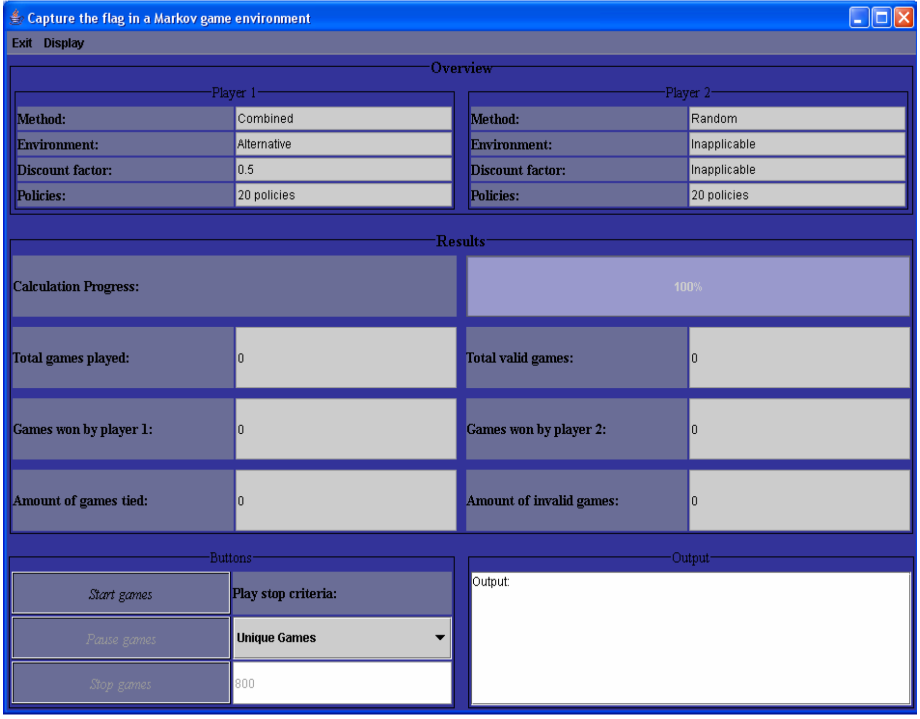
- **Simulation Method**: three ways of simulating the game are present: *Unsimulated Computer Play*, *Simulated Computer Play* and *Simulated Human Play*. In the *Unsimulated Computer Play*, each played game is not displayed but instead the desired amount of games are played in one run and only the results of the games are given. Because it is impossible for a human player to play without the game being simulated, it must be two computer controlled players competing with this simulation method. In a simulated play, each game is played turn-by-turn, meaning that a game-board is visible and each action is displayed upon this board. The difference between *Computer Play* and *Human Play* is that in the latter case one of the learning methods used must be *Human*.
- **Learning Method, Learning Environment and Learning discount factor**: these parameters do not differ from the learning feature explained in the *Learning policies* paragraph.

- **Amount of policies**: you must select the amount of different policies you wish to use for the game play. The given amount define the minimal amount of policies that must have been learned with the given parameters as agent and opponent. This means that, when you input the amount of 20, 40 policies will be read from file: 20 agent and 20 opponent policies. If insufficient policies are present, an error will be given.

When all parameters are set, you can press the *Start Simulation* button. The *Unsimulated Computer Play* and *Simulated Human Play* simulation screens will be explained next. The *Simulated Computer Play* works in an analogous fashion as *Simulated Human Play*.

Unsimulated Computer Play

If you have chosen *Unsimulated Computer Play*, then a screen similar to the one displayed in figure F.6 should appear.



Player 1		Player 2	
Method:	Combined	Method:	Random
Environment:	Alternative	Environment:	Inapplicable
Discount factor:	0.5	Discount factor:	Inapplicable
Policies:	20 policies	Policies:	20 policies

Calculation Progress:	100%		
Total games played:	0	Total valid games:	0
Games won by player 1:	0	Games won by player 2:	0
Amount of games tied:	0	Amount of invalid games:	0

Play stop criteria: Unique Games

800

Output:

Figure F. 6. Unsimulated Computer Play.

Each panel on the screen has the following purpose:

- **Player 1 and Player 2**: These panels display information about the competing players.
- **Results**: This panel displays the results of played games. The *Calculation Progress* bar indicates the progress if games are being played.
- **Buttons**: This panel is the input panel for the user. The *Start- Pause-* and *Stop games* buttons are self explanatory. The *Play stop criteria* can either be *Unique Games* or *Minimal Valid*. If *Unique Games* is chosen, then every policy of player 1 will play a single game against every policy of player 2[1], so each unique game – which is a unique combination of a player 1 and player 2 policy – is played exactly one time. If *Minimal Valid* is chosen, then the user must input a minimal amount of valid games that must be played and the two players will continue to compete against each other

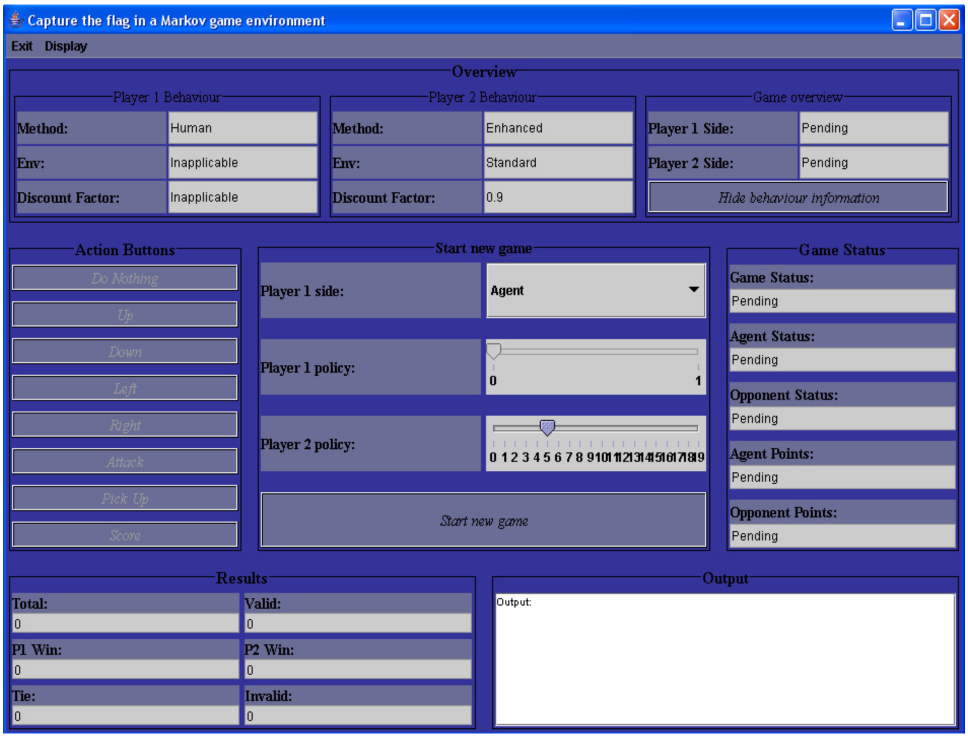
[1] Of course it is only possible to let agent policies play against opponent policies.

until the minimal amount has been reached[1]. The obvious difference between the two stop criteria is that in the latter case the user can input the minimal amount of desired valid games.

- **Output**: This panel displays the output generated by the program.

Simulated Human Play

A screen similar to figure F.7 is displayed if *Simulated Human Play* is chosen.



Player 1 Behaviour

Method: Human
Env: Inapplicable
Discount Factor: Inapplicable

Player 2 Behaviour

Method: Enhanced
Env: Standard
Discount Factor: 0.9

Game overview

Player 1 Side: Pending
Player 2 Side: Pending

Start new game

Player 1 side: Agent
Player 1 policy: 0 1
Player 2 policy: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Game Status

Game Status: Pending
Agent Status: Pending
Opponent Status: Pending
Agent Points: Pending
Opponent Points: Pending

Results

Total: 0
Valid: 0
P1 Win: 0
P2 Win: 0
Tie: 0
Invalid: 0

Output: Output:

Figure F. 7. Simulated Human Play – game selecting screen.

The frame displayed in figure F.7 is the game selecting screen where the user must input which policy will be used for the upcoming game. After a game is finished, this screen will re-appear. Let's explain all the panels that are present in this screen:

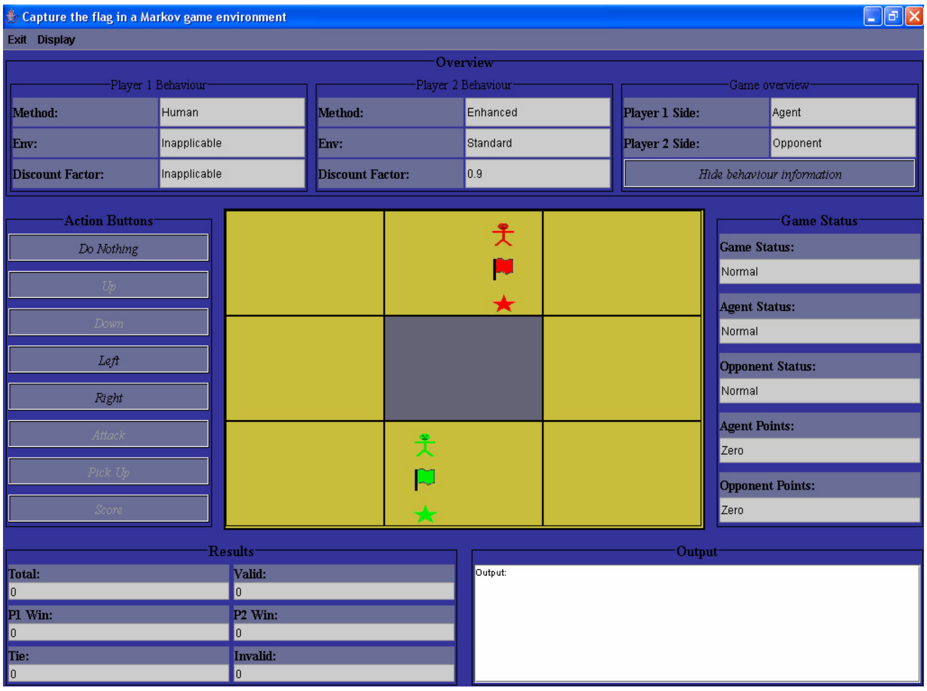
- **Player 1 Behaviour** and **Player 2 Behaviour**: These panels display information about the two players.
- **Game overview**: This panel displays which side each player is playing on and it contains the button *Hide/Show behaviour information*. This button hides or shows the behavioural information of both players displayed in the previously two mentioned panels. This option was implemented in the game to shield the human players that helped as part of the empirical evaluation from information that could influence their zeal and effort.
- **Action Buttons**: This panel contains all the possible actions that can be used by the human player. In the game selecting screen, these buttons are all disabled because no game is in progress. Once the game has started – as illustrated in figure F.8 – the buttons are enabled that represent actions that are possible for the human player.
- **Start new game**: This panel contains the input fields required to start a new game. The panel is replaced by a game board if a game is started. The user can determine

[1] In each new game, a new policy for player 1 and player 2 is used. If the given minimal valid amount is greater than the amount of possible unique games, then every policy of player 1 has played against every policy of player 2 at least once.

which player will play as agent and which policy will be used by the computer-controlled player. The policy-selecting-slider is disabled for the human player, but if *Simulated Computer Play* was chosen then both sliders would be enabled. When the parameters are set, the user can click the *Start new game* button.

- **Game Status**: This panel displays information relevant for the game in progress.
- **Results**: The results of games that have been played are displayed in this panel.
- **Output**: All relevant output generated by the game is displayed here.

Once the game has been started, the following screen appears:



Player 1 Behaviour

Method:	Human
Env:	Inapplicable
Discount Factor:	Inapplicable

Player 2 Behaviour

Method:	Enhanced
Env:	Standard
Discount Factor:	0.9

Game overview

Player 1 Side:	Agent
Player 2 Side:	Opponent

Game Status

Game Status:	Normal
Agent Status:	Normal
Opponent Status:	Normal
Agent Points:	Zero
Opponent Points:	Zero

Results

Total:	0	Valid:	0
P1 Win:	0	P2 Win:	0
Tie:	0	Invalid:	0

Output

Output:

Figure F. 8. Simulated Human Play - game in progress.

As can be seen, the *Start new game* panel is replaced by a game board. The state of the game can be seen from this board and the *Game Status* panel. On the board, the agent is represented by a green figure, the agent flag by a green flag and the agent flag starting position by a green star. The red symbols correspond with the same elements for the opponent player. The game board is automatically replaced by the *Start new game panel* when the game ends.

Program files

The following files are edited by the program when it is running. Let's assume that C:\SRL\ is the working directory:

- Immediately after starting the program, a file C:\SRL\err.log is created. All error messages generated by the program are written into this file. If at any time the program reacts strangely and the generated output of the program is insufficient, then you can check this error file to see if anything went wrong.
- If you have used the program to learn policies then those policies will be placed in the C:\SRL\policies\ directory. If you wish to use previously learned policies – for example by using the policies in the policies.rar file delivered on the CD – then those policies must be placed in this directory. Any policy is identifiable by learning environment, learning method, discount factor and whether it is maximizing or minimizing utility. As such, each policy is placed in the file that corresponds to

.\policies\<environment>\d<discount factor>\<max|min><method>.ctf, where <environment> is either *Standard*, *Alternative* or *Simple*, <discount factor> is either 0.1, 0.5 or 0.9 and <method> is either *Complete*, *Combined* or *Enhanced*. A utility maximizing policy learned in the *Alternative* environment with a discount factor of 0.5 by using the *Complete* method would thus be placed in the file .\policies\Alternative\d0.5\maxComplete.ctf.

- Besides the global policies that are placed in the previously explained files, the *Combined* method also requires a division into situation state sets and local policies. The division into state sets for an environment is placed in the .\policies\<environment>\localStateSets.ctf file, for example .\policies\Standard\localStateSets.ctf. The situational policies are written to the .\policies\<environment>\d<discount factor>\local\<max|min>local(<base reward>).ctf, where <base reward> is the heuristic state reward upon which the situation is based.