Adapting Reinforcement Learning to Tetris

Submitted in partial fulfilment
of the requirements of the degree
Bachelor of Science (Honours)
of Rhodes University

Donald Carr
Department of Computer Science
Rhodes University
Grahamstown 6139, South Africa
g02c0108@campus.ru.ac.za

November 7, 2005

Abstract

This paper discusses the application of reinforcement learning to Tetris. Tetris and reinforcement learning are both introduced and defined, and relevent research is discussed. An agent based on existing research is implemented and investigated. A reduced representation of the Tetris state space is then developed, and several new agents are implemented around this state space. The implemented agents all display successful learning, and show proficiency within certain conditions.

ACM Classification System (1998) I.2.8 Problem Solving, Control Methods, and Search

Acknowledgements

My sincere thanks to Philip Sterne for his continued patience and general enthusiasm towards the complexities of reinforcement learning. I hope it survives the completion of my thesis. My heartfelt appreciation to Leah Wanta for her support, rambunctious jibing and dedicated proofreading. Thanks to Microsoft, Telkom, Thrip, Comverse, Verso and Business Connexion, whose funding made investigating computationally heavy artificial algorithms as painless as possible. My thanks to the Rhodes University Computer Science Department, who offer an incredible honours programme and whose staff welcomed questions and gave their time generously. And finally, my thanks to the Rhodes University Physics Department for the insight into numerical methods and control methods they have given me.

Contents

1 Introduction 6

2 Related Work 8

2.1 Tetris 8

2.2 Mathematical foundations of Tetris 10

2.3 Solving NP-Complete problems 11

2.4 Reinforcement learning 12

2.4.1 The value function 12

2.4.2 Eligibility traces 14

2.4.3 Exploration 15

2.4.4 Existing applications 15

2.4.5 Large state space successes 16

2.4.6 Tetris related reinforcement learning 17

2.5 Conclusion 21

3 Design 22

3.1 Redesigning the Tetris state space 22

3.2 The structure of a reinforcement learning agent 26

3.2.1 The discovery of transitions 27

3.2.2 The calculation of an index 27

3.2.3 Correcting for multiple subwells 28

3.2.4 Exploring amongst transitions 29

3.2.5 Updating the value function 30

3.3 Application design 30

3.4 Conclusion 32

4 The Melax-defined player 33
4.1 Melax Tetris . . . 33
4.2 Initial results . . . 34
4.3 Mirror symmetry . . . 35
4.4 Different exploration policies . . . 36
4.4.1 Optimistic exploration . . . 36
4.4.2 Standard exploration . . . 37
4.4.3 Conclusion . . . 38
4.5 Reinforcement learning constants . . . 39
4.6 Sarsa(λ) agent . . . 40
4.7 Conclusion . . . 42

5 Contour Tetris 43
5.1 Initial considerations . . . 43
5.2 TD(0) agent . . . 44
5.3 Sarsa(λ) agent . . . 47
5.4 Conclusion . . . 48

6 Full Tetris 51
6.1 Extending the contour agent to the full game . . . 51
6.2 Training subwell size vs performance . . . 54
6.3 Extension to the full Tetris well . . . 55
6.4 Extension to the full Tetris tetromino set . . . 56
6.5 Extension to the full Tetris game . . . 56
6.6 Conclusion . . . 56

7 Concluding remarks 58
7.1 Possible extentions . . . 58

List of Figures

2.1 Tetris game in progress 8

2.2 The range of complete Tetris pieces as defined by Fahey [2003] 9

2.3 Melax's reduced tetrominoes 18

2.4 Melax's results as taken from Melax [1998] 19

2.5 Bdolah and Livnat's results as taken from Bdolah and Livnat [2000] 19

3.1 The full Tetris well 22

3.2 Height-based Tetris well 23

3.3 Height difference-based Tetris well 24

3.4 Capped height difference-based Tetris well 24

3.5 Capped height difference-based Tetris subwells 25

3.6 Mirror identical states 25

3.7 Dividing of well into subwells 29

3.8 Class diagram of reinforcement leaning oriented Tetris 31

4.1 Our Melax-defined agent 34

4.2 Results for the Melax-defined agent 34

4.3 Performance impact of mirror symmetry 35

4.4 Exploring methods in conjunction with optimistic exploration 37

4.5 Exploitation of knowledge 37

4.6 Exploring methods without optimistic exploration 38

4.7 Exploitation of knowledge 38

4.8 Impact of α on learning 39

4.9 Performance of Sarsa(0) agent 40

5.1 TD(0) agent in well of width four 45

5.2 TD(0) agent in well of width five 45

5.3 Contrasting hole inclusion 47
5.4 Sarsa agent in well of width four 48
5.5 Sarsa agent at time of termination 49
5.6 Sarsa agent in well of width five 50

6.1 Agent trained in subwell of width four playing in well of width five 52
6.2 Agent trained in subwell of width five playing in well of width ten 52
6.3 Comparison of extendability of subwell sizes with reduced tetrominoes 54
6.4 Comparison of extendability of subwell sizes with standard tetrominoes 54
6.5 TD(0) agent in full well with reduced tetrominoes 55
6.6 Sarsa(λ) agent in full well with reduced tetrominoes 56
6.7 TD(0) agent in reduced well with full tetrominoes 56
6.8 Sarsa(λ) agent in reduced well with full tetrominoes 57
6.9 Sarsa(λ) agent playing full Tetris 57

List of Tables

2.1 Melax's results for reduced Tetris 18
2.2 Relational regression results [Driessens, 2004] 21

Chapter 1

Introduction

Reinforcement learning is a branch of artificial intelligence that focuses on achieving the learning process in the course of a digital agent's lifespan. This entails giving the agent the ability to perceive its circumstances, giving it a memory of previous events and rewarding it on account of its actions in the context of a predefined reward policy. The drawback of traditional reinforcement learning is that there is an exponential increase in the agent's storage and exploration-time requirements for a linear increase in the dimensions of a problem. This has restricted the adoption of reinforcement learning in many domains.

Tetris is a well established game that was created in 1985 by Alexey Pajitnov and has been thoroughly investigated by both the mathematics and artificial intelligence communities. Although conceptually simple, it is NP-complete [Breukelaar et al., 2004] and any formalised optimal strategy would be incredibly contentious.

We seek to successfully apply reinforcement learning to Tetris, which is a challenge due to the size of the Tetris game. Our approach involves simplifying the description of the Tetris game rather than adopting specialised reinforcement learning approaches. This involves extracting the core information required to function intelligently from the full problem description. The identification of a general means of reducing problem descriptions may remove the existing complexity-related restrictions and lead to the wider application of reinforcement learning to sophisticated problems.

Chapter 2 introduces Tetris and places it in its mathematical context. The application of reinforcement learning to Tetris is then justified before delving deeper into the theory behind reinforcement learning. This is followed by a discussion of relevent research. In Chapter 3, the reduction of the Tetris state space to a contour-state description, the design of the reinforcement learning agent and the design of the full application are discussed. Chapter 4 discusses the implementation and investigation of an agent covered in the related work and explores different aspects of reinforcement learning. In Chapter 5, we implement agents that utilise the contour state description developed in Chapter 3, and we investigate their performance and characteristics. We then discuss the extension of the contour players to the complete game in Chapter 6 and summarise the investigation in Chapter 7.

Chapter 2

Related Work

In this chapter we introduce a formal specification of Tetris which defines our problem domain. We then look beyond the raw specification and discuss the established mathematical traits of Tetris. We justify the adoption of reinforcement learning for solving Tetris, introduce the theory that we use throughout the paper and discuss related research in the field of reinforcement learning. We end off with a review of previous attempts to apply reinforcement learning to Tetris.

2.1 Tetris

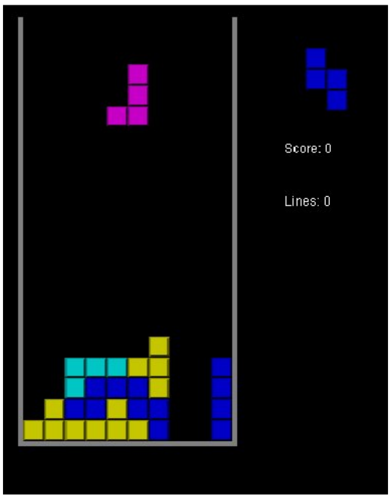


Figure 2.1: Tetris game in progress

Tetris, shown in figure 2.1, is so well established that its name has basically lent itself to the entire genre of puzzle games. All variations have a range of different tetrominoes (see Figure

2.2 for the standard set), which are each defined by a static arrangement of square blocks. These tetrominoes can be rotated and translated in the absence of obstructions.

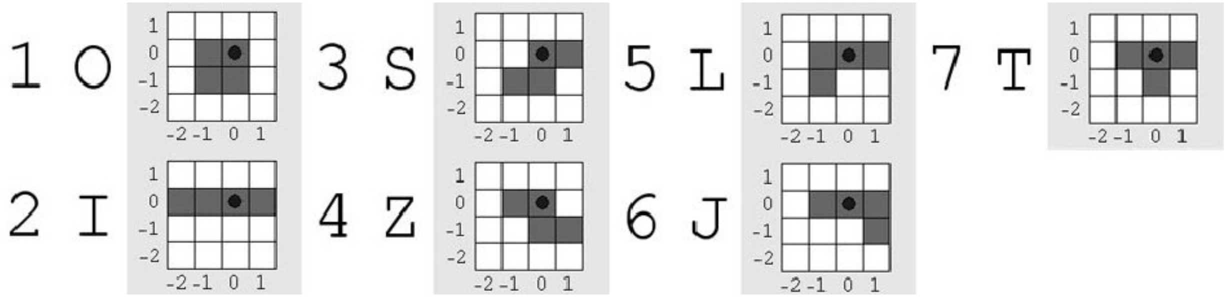


Figure 2.2: The range of complete Tetris pieces as defined by Fahey [2003]

A single tetromino is selected by the game and appears in the top centre block of a discrete well. The tetromino descends at a fixed rate, which is determined by the current difficultly level, until it meets an obstruction. The tetromino is set in place if the contact still exists in the descent step following initial contact with an obstruction. If in being fixed it completes a row, the row is completely removed and the entire well's contents above the deleted row are shifted downwards one row.

Many different artificial intelligence approaches have been applied to Tetris, and in order to remove implementation discrepancies when comparing algorithms, guidelines defining the Tetris game must be adopted. The agent, given the successful application of reinforcement learning, will therefore achieve results which will be directly comparable with those attained by other implementations following the same specifications. The standards set forth by Fahey [2003] were chosen, as there is a fair amount of existing artificial intelligence research associated with them and they seem reasonable, intuitive and comprehensive.

Formal Tetris Specification [Fahey, 2003]

- Tetris has a board with dimensions 10 by 20
- Tetris has seven distinct pieces (see figure 2.2)
- The current game piece is drawn from a uniform distribution of these seven pieces
- Points are awarded for each block that is landed (not for completing rows)
- The player scores the most points possible for each piece by executing a drop before one or more free-fall iterations transpire

- The game has ten different difficultly settings, which determine the period of free-fall iterations, and are applied as row completion passes certain thresholds

We strictly adopt the first three specifications, but the remaining specifications entail minor adjustments in implementation and only warrant adoption when the agent is contrasted against existing results. We remove the time-based descension of the tetromino since it as optimistic consideration that can be adopted at a later date in necessary.

Artificial intelligence approaches to Tetris are categorised according to the information supplied to the artificial agent. An agent that is solely aware of the current tetromino is referred to as a one-piece algorithm, while an agent that is also informed about the next tetromino is referred to as a two-piece algorithm. Any agent informed of the next tetromino is given an obvious advantage as it can plan its current tetromino placement accordingly. This restricts the comparison of algorithms to other algorithms within the same category.

2.2 Mathematical foundations of Tetris

It has been mathematically proven [Brzustowski, 1992, Burgiel, 1997] that it is possible to generate a sequence of tetrominoes that will guarantee the eventual termination of any game of Tetris played in a well of width 2(2n+1), with n being any integer. This is most readily achieved by sending alternating Z and S pieces (see figure 2.2) into the well, which leads to the gradual accumulation of persistent blocks and eventually the termination of the game [Brzustowski, 1992]. The implication of this is: even if the agent were to play a flawless game of Tetris, the series of tetrominoes that guarantee termination of the game are statistically inevitable after a long enough time (infinite period).

The number of rows completed by a good Tetris player will follow an exponential distribution [Fahey, 2003], owing to the stochastic nature of the game. Some Tetris games are harder than others due to the pieces drawn and the order in which they are delivered, and the resulting performance spectrum can be mistaken for erratic behaviour on the part of the player.

Tetris has been proven to be NP-complete [Breukelaar et al., 2004]. The implication of this is that it is computationally impossible to linearly search the entire policy space and select an ideal action. This justifies the use of approximating techniques like reinforcement learning in trying to determine the optimal policy.

One of the assumptions reinforcement learning requires is that the environment has the Markov property [Sutton and Barto, 2002]. Tetris satisfies this requirement, as all the relevant information required to make an optimal decision is represented in the state at any instant in time. Rephrased, there is no historical momentum to the current state of the system, and any future occurrence is therefore entirely dependent on the current state of the system. If we are handed control of a Tetris game at any point, we are as equipped to play from that point as we would be had we played up until that point.

2.3 Solving NP-Complete problems

Attractive solutions to problems outside of the computational range of linear search methods can be discovered by emulating biological processes. Two such approaches are genetic algorithms and reinforcement learning.

Genetic algorithms search directly in the solution (policy) space of a problem, breeding solutions amongst the fittest individuals in order to approach an optimal solution. Reinforcement learning yields an environment to an agent that is subsequently left to explore for itself. The agent gets feedback directly from the environment in the form of rewards or penalties, and continuously updates its value function towards the optimal policy. Both methods ideally converge on the best policy[McLean, 2001], although their different routes gear them towards distinct problems.

Reinforcement learning offers a higher resolution than genetic algorithms, as genetic algorithms select optimal candidates at the population level, while reinforcement learning selects optimal actions at an individual level [McLean, 2001]. Every action taken under reinforcement learning is judged and driven towards the optimal action in that state, whereas in contrast, genetic algorithms reward complete genetic strains regardless of the behaviour of individual genes within the previous episode. Reinforcement learning also differs from genetic algorithms by indirectly adjusting its policy through the updating of its value function, rather then introducing random variations directly to its policy and relying on the agent chancing upon a superior policy.

A great deal of information is conveyed in the course of a Tetris game, and reinforcement learning would enable the agent to capture this information and adapt accordingly. This would also enable a directed real-time adjustment of the agent's policy, rather than a global adjustment at the end of the game. Another consideration is that as the agent's performance improves, the number of rows completed in a game increases and the lifespan of the agent increases. This does not impact negatively on the reinforcement learning agent as it learns with every move, but has an increasingly large impact on the rate of improvement of the genetic algorithm agent, since it learns with the end of each game. These traits indicate that reinforcement learning is better suited to solving Tetris than genetic algorithms.

2.4 Reinforcement learning

Reinforcement learning can be applied in non-deterministic environments, where taking a certain action within the context of a state does not necessary lead to the same reward or state transition. It does, however, require that the environment be stationary and that the probabilities of getting a certain reward or transitioning to a certain state remain the same [Kaelbling et al., 1996]. [1] Tetris is a stochastic game due to the random drawing of the next block. However, it is stationary as the game description remains constant and it therefore satisfies this requirement.

Reinforcement learning defines an approach to solving problems rather than offering a rigid specification to be followed through to implementation. It is defined in terms of an agent's interaction with an environment. The agent's perception of the environment is encapsulated in a value function which spans every state in which the environment can exist and associates a value with each of these states. This value function is updated upon receiving feedback from the environment as defined by the reward function. The reward function is statically declared outside of the influence of the agent at the outset of a problem and steers the development of the value function. Rewards can be either positive or negative, and encourage or discourage the agent accordingly. The agent follows a policy, derived from the value function by the exploration policy, that maps states to actions and therefore dictates the behaviour of the agent [Sutton and Barto, 2002].

The goal of the agent is to maximise its long-term reward. Its initial behaviour is driven purely by trial and error, but as the agent starts to form an accurate impression of the values of the states, it becomes increasingly important for it to strike a balance between the exploration of unknown states and the exploitation of valuable states [Sutton and Barto, 2002].

When a goal has been reached the reward function supplies a reward to the agent. The value associated with the originating state is adjusted accordingly and this reward is gradually backed up throughout the value function in the wake of the following transitions [Sutton and Barto, 2002].

2.4.1 The value function

Reinforcement learning is a broad field of research and it encompasses a range of well investigated learning approaches suited to specific problem domains. In this paper we adopt TD(0) and

[1] For completeness it should be noted that reinforcement learning can be extended to problems that are non-stationary on condition that they change slowly. This is achieved by emphasising the importance of recent experiences over historical ones, and allows the agent to continue learning in slowly varying tasks.

Sarsa(λ), which both promise distinct benefits. TD(0) is the simplest reinforcement learning approach and utilises a state-value table which associates a value with every state in the state space. Sarsa(λ) extends this by utilising a state-action table which associates a value with every action leaving every state. This increases the storage and exploration-time requires of the agent but also improves the resolution afforded the agent. The values in both tables are indicative of the long-term rewards associated with the respective entries. These entries are states for the TD(0) agent and are updated through the use of equation 2.1. In Sarsa(λ) these entries represent state-action pairs and these values are updated using equation 2.2

$$V(s_t) \leftarrow V(s_t) + \alpha(r_t + \gamma V(s_{t+1}) - V(s_t)) \tag{2.1}$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{2.2}$$

$V(s_t)$ refers to the value of the current state, $V(s_{t+1})$ refers to the value of the next state and r_t refers to the reward received in transitioning from the current state to the next state. Similarly, $Q(s_t, a_t)$ refers to the value of the selected action within the current state, and $Q(s_{t+1}, a_{t+1})$ refers to the value of the action, as chosen by the policy, taken in the destination state.

Equation 2.1 and 2.2 state that the value associated with the current state is equal to the current value and a correction factor. This factor is the current value subtracted from the sum of the observed reward and discounted future rewards. The value function therefore incrementally converges on a long-term optimal solution.

When evaluating the value of a transition, the TD(0) includes the reward associated with the transition in its value considerations. This is distinctly different from Sarsa which only considers the value of the state-action pair when evaluating transitions, and has implications discussed in chapter 4.

The α and γ terms dictate the behaviour of the update function :

- The α term determines the weighting of the current adjustment in relation to all previous adjustments. A large constant α gives recent value function adjustments a larger influence than historical adjustments. In order to guarantee convergence, this α value must be kept relatively small, and this can be further guaranteed by having the α value diminish over the course of an agent's training.

- The γ factor determines the extent to which future rewards affect the current value estimation. The larger the γ term, the greater the significance attributed to future rewards. This approach of "backing up" the values is an example of temporal-difference learning [Sutton and Barto, 2002] and is a way of propagating information about future rewards backwards through the value function.

The value function does not necessarily have to take the form of a table. The value function can be seen as a mathematical function that takes the originating state as input and produces the state with the highest predicted value as output. Rather than storing the values in distinct positions within a table, this information is stored in the behaviour of the function. This approach removes the storage requirements of the agent at the expense of encrypting the state information in a convoluted function, and is not explored further in this thesis.

2.4.2 Eligibility traces

Learning can be accelerated through the use of eligibility traces that update all state-action pairs following every transition. This is achieved by assigning a unity weighting to state-action pairs when they are visited and discounting this weighting on every subsequent iteration. These weighting terms allow for recently visited states to adjust their values towards those of destination states by adding localised correction terms. Each localised correction factor is calculated by multiplying the current correction factor by the local weighting factor, and these localised corrections are used to update each respective state-action pair.

This process in shown in equation 2.3.

$$\begin{aligned}
\delta &= r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \\
elig(s_t, a_t) &= 1 \\
\text{for all s,a :} & \\
Q(s,a) &\leftarrow Q(s,a) + \alpha elig(s,a)\delta \\
elig(s,a) &= \gamma\lambda elig(s,a)
\end{aligned}$$

where Q(s,a), α and γ refer to the quantities defined in 2.4.1. elig(s,a) is the weighting associated with a particular state-action pair, λ is the discounting factor used to adjust the weighting following every update and δ is the current correction factor.

Eligibility traces can be equally validly applied within the TD approach, resulting in TD(λ). However, they are not used in conjunction with the TD methods investigated in this thesis.

2.4.3 Exploration

The agent can possess one of a variety of exploration policies. With a purely greedy policy, the agent will always select the state transition believed to offer the greatest long-term reward. Although this will immediately benefit the agent, it may well fail to find the ideal policy in the long run. With an ϵ-greedy method, the agent will select the best state transition the majority of the time and take exploratory transitions the rest of the time. The frequency of these exploratory transitions is determined by the value of ϵ utilised by the policy. It is possible to vary ϵ, in order to have an initially open-minded agent that gains confidence in its value function as its experience increases over time. One problem inherent in the ϵ-greedy approach is that the agent explores indiscriminately and is as likely to explore an obviously unattractive avenue as it is to explore a promising one. An alternative approach is offered by the Softmax approach shown in equation 2.3. This approach associates a probability of selection with every possible transition, which is proportional to the the predicted value of that transition. This encourages the agent to explore promising-looking transitions more thoroughly than less promising ones.

$$P = \frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(b)/\tau}} \tag{2.3}$$

The degree to which the estimated value affects the probability of selection is dictated by the τ term, which is referred to as the temperature. For large temperatures the state transitions become almost equiprobable, while at low temperatures the respective probabilities of selection become more sensitive to value differences between the states. In the limit as temperature goes to zero, the Softmax policy reduces to the greedy policy.

2.4.4 Existing applications

Reinforcement learning performs very well in small domains, and by using the insight offered by Sutton and Barto [2002], it is fairly simple to create an agent that plays simple games like Tic-Tac-Toe or Blackjack successfully. It has been successfully applied to many sophisticated problems such as :

- Packet routing in dynamically changing networks [Boyan and Littman, 1994]
- Robotic control [Kimura et al., 1997]
- Acrobot [Sutton and Barto, 2002]

- Chess [Baxter et al., 1998]

Bellman is cited [Sutton and Barto, 2002] as stating that reinforcement learning suffers from the ”curse of dimensionality”. This refers to the exponential increase in the complexity of the system as the number of elements in it increases linearly. This tendency is responsible for reinforcement learning having relatively few successes in large state-space domains [Sutton et al., 2005]. These successes include :

- RoboCup-Soccer Keep-Away [Sutton et al., 2005]
- Backgammon [Tesauro, 1995]
- Elevator control [Crites and Barto, 1996]

2.4.5 Large state space successes

TD-Gammon

Tesauro [1995] used reinforcement learning to train a neural network to play backgammon. The program was so successful that its first implementation (Version 0.0) had abilities equal to Tesauro’s well established Neurogammon [2] [Tesauro, 1995]. More noteworthy is that by Version 2.1, TD-Gammon is regarded as playing at a level extremely close to that of the world’s best human players and has influenced the way in which expert backgammon players play[Tesauro, 1995]. The unbiased exploration of possible moves and reliance on performance rather then established wisdom leads, in some circumstances, to TD-gammon adopting non-intuitive policies superior to those utilised by humans [Tesauro, 1995].

Backgammon is estimated to have a state space larger then 10^{20}. This state space was reduced by the use of a neural network organised in a multilayer perception architecture. Temporal difference learning with eligibility traces was responsible for updating the weighting functions on the neural network as the game progressed. Another benefit associated with using reinforcement learning methods rather than pure supervised learning methods was that TD-gammon could be (and was) trained against itself [Tesauro, 1995].

[2] Neurogammon was a neural network backgammon player, trained on a database of recorded expert games, who convincingly won the 1989 International Computer Olympiad Backgammon Championship.

RoboCup-Soccer Keep-Away

Sutton et al. [2005] managed to successfully train reinforcement learning agents to complete a subtask of full soccer which involved a team of agents — all learning independently — keeping a ball away from their opponents.

This implementation overcame many difficulties, such as having multiple independent agents functioning with delayed rewards and, most importantly, functioning in a large state space. The state space problem was resolved by using linear tile-coding (CMAC) function approximation to reduce the state space to a more feasible size [Sutton et al., 2005].

2.4.6 Tetris related reinforcement learning

We found three existing extentions of reinforcement learning to Tetris that all implement one-piece methods.

Reduced Tetris

Melax [1998] applied reinforcement learning to a greatly reduced version of the Tetris game. His Tetris game had a well with a width of six, an infinite height and the greatly reduced piece set shown in figure 2.3. The length of the game was dictated by the number of tetrominoes attributed to the game, which was set at ten thousand. Although the height of the Tetris well was infinite in theory, the active layer in which blocks were placed was two blocks high, and any placement above this level resulted in the lower layers being discarded until the structure had a height of two. The game kept a record of the number of discarded rows and this was used as a score for the performance of the agent. This approach to scoring resulted in better performance corresponding to a lower score.

The two block active height prevented the agent from lowering the block structure and completing rows that it initially failed to complete. This differs from traditional Tetris where a player can complete a previously unfilled row after reducing the well structure and re-exposing it. Since the pieces are drawn stochastically and unfilled rows form an immutable blemish on the performance evaluation of the agent, this introduces a random aspect to the results.

The agent was implemented using TD(0) and was punished a hundred points for every level it introduced above the working height of the well.

Melax's agent achieved significant learning, as shown in table 2.1. These results are reflected in figure 2.4.

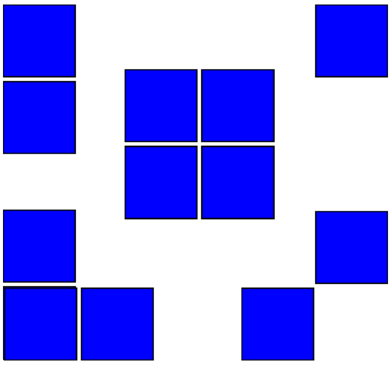


Figure 2.3: Melax's reduced tetrominoes

Game	Height
1	1485
2	1166
4	1032
8	902
16	837
32	644
64	395
128	303
256	289

Table 2.1: Melax's results for reduced Tetris

As the number of games increased, the agent learned how to minimise the total height of the pieces in the well and therefore maximised its long-term reward.

One possible problem with this implementation is that by defining rewards for sub-goals, such as increasing the working height, the agent is discouraged from a range of policies that may include the optimal policy. Melax effectively steered the development of the agent's policy towards reducing the height following every transition, rather than placing the blocks optimally and reducing the height of the final structure. Keeping the height of the game low leads to the completion of rows as the agent builds horizontally, and this might actually be the ideal policy for the agent to adopt. However, the optimal policy may never be discovered. This reward structure is ideal in the context of Melax's well description, as the historical well information beneath the working height is discarded and immediate rewards are therefore dominant. What bears consideration is that, with a different well representation, reinforcement learning might achieve a superior optimal policy, and these results reflect on the well representation rather than the reinforcement methods.

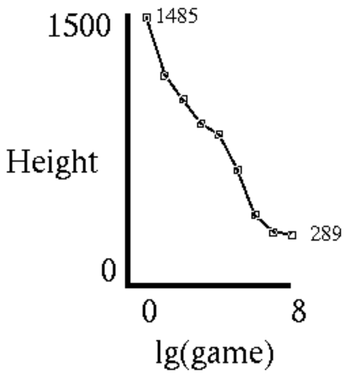
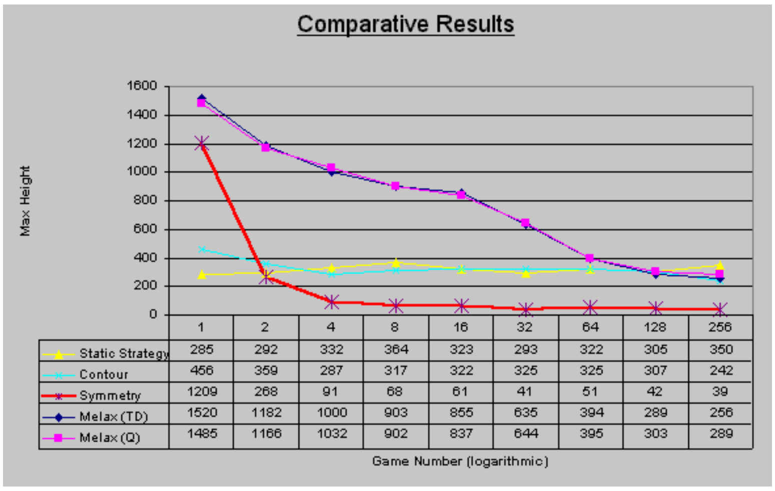


Figure 2.4: Melax's results as taken from Melax [1998]

Mirror Symmetry

Melax's approach was adopted and extended by Bdolah and Livnat [2000], who investigated different reinforcement learning algorithms and introduced state space optimisations. The state space was reduced through two distinct approaches. In the first approach, subsurface information was discarded leaving only the contour of the game for consideration. This approach was further divided into either considering the contour differences as simply being positive or negative or reducing the height differences to a small spectrum. The second state space reduction made use of mirror symmetry within Melax's well in order to reduce the number of different states.



	1	2	4	8	16	32	64	128	256
Static Strategy	285	292	332	364	323	293	322	305	350
Contour	456	359	287	317	322	325	325	307	242
Symmetry	1209	268	91	68	61	41	51	42	39
Melax (TD)	1520	1182	1000	903	855	635	394	289	256
Melax (Q)	1485	1166	1032	902	837	644	395	303	289

Figure 2.5: Bdolah and Livnat's results as taken from Bdolah and Livnat [2000]

Both optimisations appear to have greatly improved the performance of the agent, judging by the chart shown in figure 2.4.6. There are, however, some troubling aspects to these results.

The mirror symmetry results are far superior to the results achieved by any other method. This optimisation effectively ignored one reflection of duplicated states, and thus should have sped up the learning process while converging on the same solution. The accelerated learning is evident, but the results indicate that the mirror symmetry actually led to the adoption of a distinctly different policy to that adopted by the original agent. This means that the value function must have converged on a different set of values, which negates the original assumption that the values are identical and necessarily redundant. These results are suspicious, and our own investigation, discussed in chapter 4.1, reveals the predicted increase in learning speed while converging on the original optimal policy.

The contour approach extended the perceptive ability of the agent and maintained information below the original two layer structure. This enabled the agent to continually reduce the well structure over the course of the game. The results in figure seem to indicate incredibly fast learning, and by the end of the first game, the agent settles on a policy that produces a result far superior to the original Melax result.

Despite the dubious results associated with the mirror symmetry optimisation, it is a sound suggestion that is perfectly legitimate in the Tetris game defined by Melax. This optimisation is equally legitimate in the full game, since every tetromino in the standard tetromino set is mirrored within the set. Incorporating this in the reduction of our final state space would roughly halve the number of states required in describing the Tetris well.

Relational reinforcement learning

Relational reinforcement learning was applied to the full Tetris problem by Driessens [2004]. Relational reinforcement learning differs from traditional methods in its structuring of the value function. Rather than storing every possible state in a table, the relationship between the elements in the environment is utilised in developing a reduced state space. This state information is then stored in a decision tree structure.

Driessens approached the problem with three separate relational regression methods [Driessens, 2004] which he developed over the course of his thesis. The first of these regression methods had already proven itself with the successful extension of reinforcement learning to Digger[3].

Driessens results for full Tetris are shown in table 2.2 .

The RRL-RIB agent reached its optimal policy within fifty training games. In the four hundred and fifty subsequent training games this policy was not improved upon. The RRL-KBR

[3] Another arcade game with a large state space

Regression method	Learning games	Average completed rows
RRL-TG	5000	10
RRL-RIB	50	12
RRL-KBR	10-20	30-40

Table 2.2: Relational regression results [Driessens, 2004]

agent reached a better policy earlier than the other regression methods. It then rather unexpectedly unlearned its policy after a further twenty to thirty games.

Since this is actually a full implementation of Tetris, its results can be compared against other one-piece artificial intelligence methods. The best hand coded competitor completes an average of six hundred and fifty thousand rows per game and the best dynamic agent, utilising genetic algorithms, completes an average of seventy four thousand rows per game[Fahey, 2003]. Driessens results are not impressive in light of the competition and are very poor even by human standards.

Driessens attributes the poor functionality to Q-learning, stipulating that Q-learning requires a good estimate of the future rewards in order to function properly and that the stochastic nature of Tetris severely limits the accuracy of these estimates. Since his regression methods were derived from Q-learning, this inadequacy impacted all of his methods. Q-learning in known to be unstable [Sutton et al., 2005, Thrun and Schwartz, 1993] when incorporated in function approximation, and this could certainly have contributed to the poor performance reflected in the above results.

Despite the final results of Driessens's agent, the idea of exploiting the internal relationships present within the Tetris well as a means of reducing the state space is an attractive one.

2.5 Conclusion

This chapter associated the project with a Tetris standard and contextualised the reduction of the Tetris state description performed in the next chapter. Previous Tetris implementations suggested possible state space optimisations such as mirror symmetry and relational information. Melax's agent offers credible results for comparison and a clean specification that was implemented as an initial reinforcement learning experiment and is discussed in Chapter 4. The reinforcement learning concepts covered in this chapter are integral to the implementation of the agent and were directly incorporated into the design of the agent as shown in the next chapter.

Chapter 3

Design

In this chapter we reduce the state space of Tetris by adopting assumptions and discuss the possible consequences of each assumption. We then design the reinforcement learning agent and consider the processes it requires. We end the chapter by considering the structure of the whole application.

3.1 Redesigning the Tetris state space

Traditional reinforcement learning uses a tabular value function which associates a value with every state. The primary design consideration is how the Tetris state space can be reduced without discarding pertinent information. Since the full Tetris well, shown in figure 3.1, has dimensions twenty blocks deep by ten blocks wide, there are two hundred block positions in the well that can be either occupied or empty.

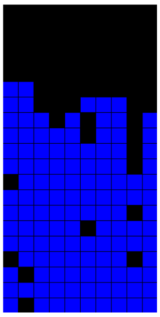


Figure 3.1: The full Tetris well

$$\text{State Space} \;=\; 2^{200}$$

This is an unwieldy number and since a value would have to be associated with each state, this representation is completely infeasible. We choose to stick with traditional reinforcement learning and introduce reductions in the tabular description of the environment by considering the game from a human perspective and by adopting the mirror symmetric optimisations suggested by Bdolah and Livnat [2000].

Assumption 1

The position of every block on screen is not a consideration that is factored into every move by a human player. We only consider the contour of the well when making decisions. We limit ourselves to merely considering the height of each column.

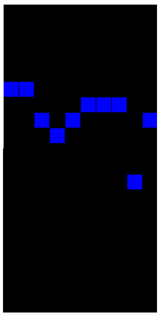


Figure 3.2: Height-based Tetris well

$$\text{State Space} \;=\; 20^{10} \approx 2^{43}$$

Assumption 2

The height of each column is fairly irrelevant except perhaps when the height of a column starts to approach the top of the well. The importance lies in the relationship between successive columns, rather than in their isolated heights.

$$\text{State Space} \;=\; 20^{9} \approx 2^{39}$$

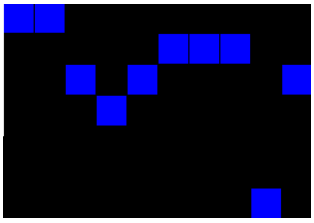


Figure 3.3: Height difference-based Tetris well

Assumption 3

Beyond a certain point, height differences between subsequent columns are indistinguishable. A human will not adopt different tactics when the height difference between two columns advances from nineteen to twenty. We could either cap the maximum height differences or start separating the heights into fuzzy sets as the height differences increase past certain thresholds. We cap the maximum height difference between wells at ± 3, and truncate all height differences outside of this range down to ± 3. The agent will therefore generalise for any height difference greater than three. Since only the straight tetromino can span a height difference of three, and this tetromino can span any height difference, this assumption seems fair to make.

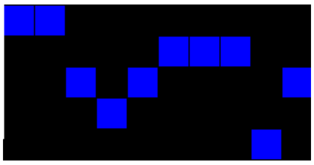


Figure 3.4: Capped height difference-based Tetris well

$$\text{State Space} \quad = \quad 7^9 \approx 2^{25}$$

Assumption 4

The largest tetromino is four blocks wide. At any point in placing the tetromino, the value of the placement can be considered in the context of a subwell of width four. These subwells could then be extended to the full well by tiling them across the extent of the full well.

$$\text{State Space} \quad = \quad 7^3 = 343 \approx 2^8$$

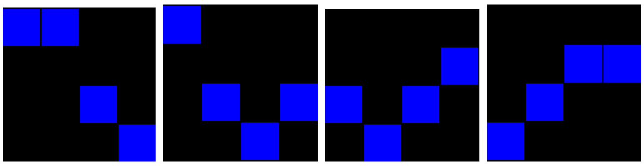


Figure 3.5: Capped height difference-based Tetris subwells

Assumption 5

Since the game is stochastic and the tetrominoes are uniformly selected from the tetromino set, the value of the well should be no different to its mirror image.

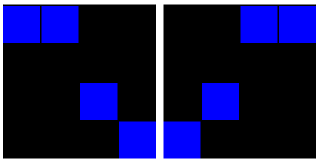


Figure 3.6: Mirror identical states

$$\text{State Space} \quad = 175$$

Repercussions

We now have a much reduced state space, which we hope will neither limit the agent nor appreciably steer its policy. We subsequently refer to any agent that functions within this state space as a contour agent. The implications of the assumptions adopted above should be considered before we proceed towards implementing agents that adopt this state representation.

- Assumption 1 discards all information about the subsurface structure of the well. The initial representation can be perceived to store the location of every hole in the structure. The agent will therefore be oblivious to any differences between transforming to a said contour and the same contour with spaces beneath the surface. The existing holes are not important, but we may wish to include a penalty for the holes introduced during a state transition.

- Assumption 2 introduces no obvious evils.

- Assumption 3 removes the agent's ability to distinguish between extremes in height differences. It is reasonable to assume that the agent will have to utilise states with large height differences at some point in following the optimal policy, and that the broadly generalised values stored in these positions will convey no meaningful information.
- Assumption 4 removes the global context in which the agent functions and restricts its view to each individual transition. There is a distinct difference between this assumption and the others, and that is that this reduction not only reduces the information available to the agent but completely separates the agent's perception of the game from reality. We need to dynamically re-establish the context in which the agent is functioning and this is dealt with in section 3.2.3. Having a well with similar dimensions to the tetromino set might hamper the development of a realistic policy, and there is a possibility that a wider subwell might be ideal.
- Assumption 5 reduces the state space in a non-trivial fashion. The mirrored states are still allocated space but are never explored, removing the computational burden they represent.

3.2 The structure of a reinforcement learning agent

We set out to create an agent that functions within the reduced state representation developed above. We decided to implement a one-piece algorithm and the agent can therefore only consider the tetromino currently allocated to it in the course of each move. The agent's behaviour can be separated into distinct processes.

1. Discover transitions
   (a) Calculate index
2. Correct for multiple subwells (Note: only used in extending the well)
3. Choose amongst transitions using exploration policy
4. Update value function

Each of these processes is now discussed in depth in the following subsections.

3.2.1 The discovery of transitions

This method discovers the transitions available from the current state. The agent makes use of a conceptual game that exists purely for its benefit, isolating any conceptual manipulations from the full game. The agent copies the block formation from the real Tetris well into its conceptual well before performing each of the possible transitions with the current tetromino. The well structure resulting from each transition is used by the indexing function to calculate the appropriate index into the value function. Each transition is stored as an object that contains the number of translations and rotations, the resulting reward and the value of the resulting state. Every unique transition is added to a list of possible transitions.

3.2.2 The calculation of an index

An indexing function in used to associate every Tetris state with a unique position in the value table. The indexing function is completely dependent on the state representation of the game and therefore differs between the Melax-defined agent developed in Chapter 4 and the agent we develop in Chapter 5.

For the Melax-defined agent, the well is viewed as a 12 block array. The first position is assigned a value of one, and this value is doubled for each successive block position. If the position is occupied, the value of that position is added to an accumulator. This enables us to calculate a unique index for every state.

For our contour agent, the well is viewed as a block array one block narrower than the well the agent is playing in. Each of these positions can have a value between 0 and 6, corresponding to a height difference of -3 and 3 accordingly. The first position is assigned an intrinsic value of 1. The intrinsic value of each subsequent position is determined by multiplying the previous position's intrinsic value by 7. Multiplying the values in the block array by the intrinsic value of the array position and summing the resulting products yields a unique index. This method is not limited to height differences of 7 and is readily expandable, with the inherent values of the block positions increasing as x^n, where x is the number of height differences and n the relative row position.

The index adjustment due to mirror symmetry

Mirror symmetry is factored into both of the above indexing functions by calculating the index values from both directions. The smallest value is then always selected. The unused array positions are still issued storage space but are never explored, and therefore introduce no exploratory overhead.

The index adjustment due to further extentions in the game description

The core indexing function solves the problem of representing the state space. Expanding the agent's perception beyond the state is handled by introducing an adjustment to the previous calculated index. This index adjustment function adds further weighted terms to the existing index. The weighting used is always the full size of the previous state description. Sarsa for instance requires a state-action table. The state index is adjusted to include the tetromino type, position of placement and number of rotations. Each enhancement adds the existing index to the new consideration multiplied by the existing state space. This approach facilitates the continued expansion of the state description.

None of the Sarsa agents considered within this thesis utilise the mirror symmetry optimisation, due to the late discovery of incompatibilities between our mirror optimisation approach and our method of extending the state-value table to a state-action table.

3.2.3 Correcting for multiple subwells

Any agent that utilises the state space developed in the last section will imagine the full well to be four blocks wide by twenty blocks high. The agent believes it is playing in a well of width four while in actuality it can be playing in a well of any width. This highlights the distinction between the full Tetris game and the conceptual Tetris game. The conceptual game's well is defined by the contour-state description while the full Tetris description is unrestrained. The separation between reality and conceptual reality removes the independence of the agent, and introduces a clear distinction between the training and running of the agent. The agent must be trained in a Tetris well with the same dimensions as its conceptual well. However, it can be run in any well with equal or greater width than its conceptual well.

We solved the disparity between the full Tetris well and the contour Tetris well by breaking the full well into overlapping subwells. This is shown in figure 3.7. The single global transition shown in figure 3.7 corresponds to several distinct subtransitions within different subwells.

In discovering possible transitions, the agent previously iterated through every possible positional placement and every orientation of the current tetromino within these placements. In extending the subwell representation to the full game, the agent had to iterate through all the subwells comprising the full well and discover the values of the subtransitions in these subwells.

The transitions in the full well are therefore related to subtransitions by a transition adjust-

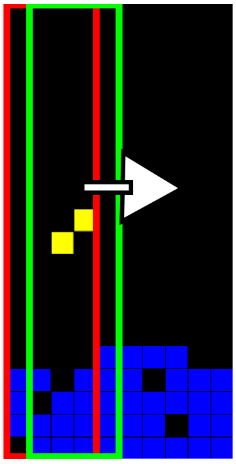


Figure 3.7: Dividing of well into subwells

ment function which accepts the complete range of subtransitions and returns a list containing a single value for every unique transition in the full well. There were two distinct approaches to selecting global transitions out of all the subtransitions. The subtransitions occupying the same physical description in the full well can be averaged, leading to a single transition which reflects the value of the transition across all the subwells and gives a broad overview of the value of the transition. Another approach is to retain the largest transition value from all the subwells, disregarding all the other transitions and placing the emphasis on selecting the optimal action. These approaches were both tested and are discussed in Chapter 6.1.

3.2.4 Exploring amongst transitions

ϵ-greedy, greedy and softmax are implemented as competing exploration policies, and any of them can be implemented alongside optimistic exploration if desired.

Greedy and softmax are implemented as distinct policies. However, ϵ-greedy is a deviation from the greedy policy and is implemented within the greedy method by giving the agent a fixed probability of choosing randomly amongst the available transitions. There are therefore two methods that accept a range of possible transitions and return a single transition.

Optimistic exploration is achieved by initialising the value function with values slightly larger than the largest anticipated value. This value is easily determined when dealing with purely negative rewards, since all states will have negative values and therefore zero is an optimistic value. When dealing with positive rewards, the easiest approach is to set the agent to explore for a large period of time, look at the predicted values and adopt a value slightly larger than the largest value for the starting values.

3.2.5 Updating the value function

The transition selected by the exploration policy is taken in the full game. The results of this transition are used in updating the agent's value function.

The update functions for the TD(0) and Sarsa(λ) agents are shown by equation 2.1 and 2.3, respectively. The two approaches require different state representation and data manipulations, and this disparity is grounds for implementing distinct agents.

The update method accepts the current index, destination index and a reward. In the Sarsa(λ) agent this reward is interpreted as the reward associated with taking the action from the current state. In TD(0), the reward can be interpreted as the reward received in transitioning from the current state to the destination state.

In implementing the Sarsa(λ) agent, the state-value table had to be extended to contain every transition off each state. The number of transitions is dictated by the number of different tetrominoes, the width of the well and the number of different orientations. The size of the state-action space is calculated by multiplying the number of transitions by the original number of states. This drastically increases the state space of the game and may introduce a great deal of redundant information depending on the tetromino set used. Fixing this would require including tetromino-set specific considerations in our Tetris implementation and would remove the generic nature of the implementation. We choose to retain the redundancy as, although it introduces additional training time, it should not impact the final values converged on by the value function.

Replacement eligibility traces are used in the Sarsa(λ) update method.

3.3 Application design

We designed a Tetris game from first principles in order to have complete control over the structure of the game and familiarise ourselves with the required methods.

The application can be readily divided up into the following logical classes.

- The game window
- The Tetris game
- The tetromino set
- The tetromino object
- The Tetris player

The interaction between these components is shown in figure 3.8

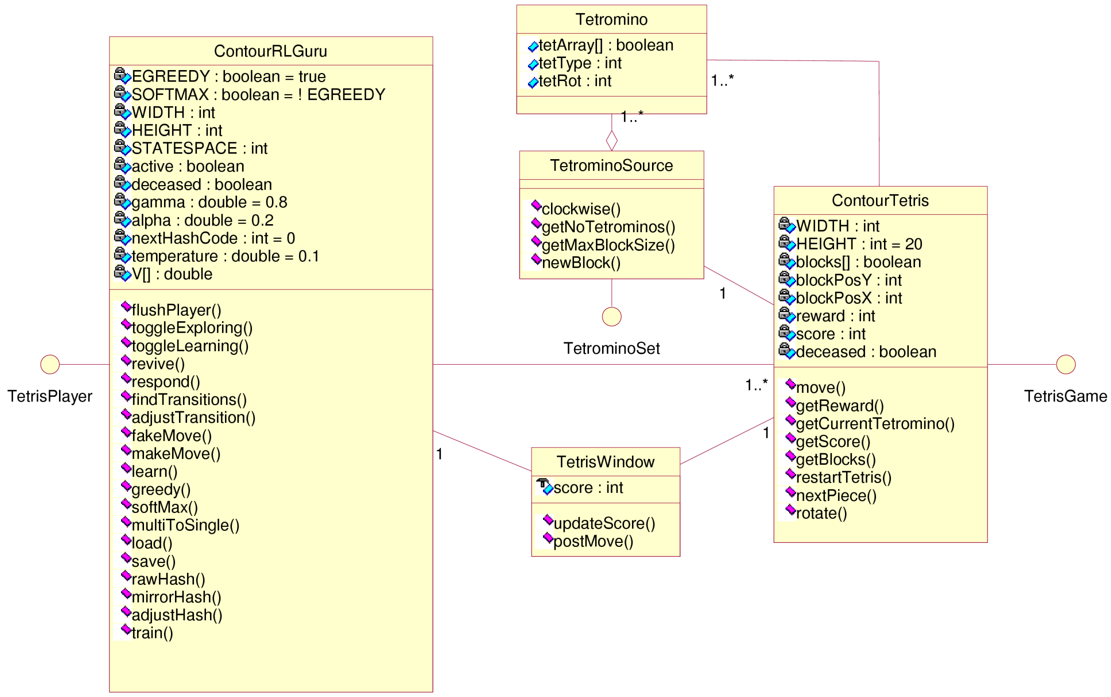


Figure 3.8: Class diagram of reinforcement leaning oriented Tetris

The game window acts as the interface between the user and the Tetris game, and is responsible for displaying all information regarding the system. The Tetris game is the self-contained core of the program and is responsible for managing the game and yielding information on the state of the game. The Tetris game and agent are both instantiated in the game window and the agent is passed a reference to the game upon instantiation. Exclusive control is switched between either of these two control sources, and the game is therefore oblivious to the source of its instructions. Including human interaction enables the Tetris implementation to be checked. The tetromino set is instantiated within the Tetris game and is responsible for the definition and rotational manipulation of the tetrominoes. All tetromino transitions which occur within the well are checked and performed in the Tetris game. The tetromino object is common to all methods and is a simple structure encapsulating the traits of a tetromino. This object is oblivious to the definitions dictated by the tetromino source and is used by all the classes.

The Tetris game, tetromino set and the artificial agent are all objects that will need to change in the course of the investigation. This is simplified if these three objects implement generic interfaces which allow us to utilise polymorphism. This structuring allows for seamless swapping between different game definitions, tetromino sets and artificial agents. The differing game definitions are largely restricted to variations in the dimension of the well, whereas tetromino-set definitions are unrestricted and any tetromino collection can be created. As long as the agent implements the correct interface, the theory guiding the actions of the agent can subscribe to any artificial intelligence method. This follows the reasoning, outlined by the strategy design pattern [Gamma et al., 1998], that competing algorithms should implement a common interface and therefore be seamlessly interchangeable.

We would expect any object-orientated Tetris game to deal with a large number of tetromino objects, and the performance penalty introduced by instantiating large numbers of simple objects warrants consideration. This is addressed by conventional design patterns and corresponds to a fly-weight design pattern, as discussed in Gamma et al. [1998]. Rather than having the game continually recreating individual tetrominoes within the set of available tetrominoes, it is preferable to create every possible tetromino once and subsequently pass out a reference to the relevant tetromino. This optimisation is piece-specific and is implemented in the tetromino source class by instantiating a two dimensional array of tetrominoes corresponding to the range of orientations for every tetromino. The first time a tetromino is assigned or uniquely rotated it is created within this array structure, and a pointer is returned following all subsequent requests for this orientation of the tetromino.

3.4 Conclusion

This chapter detailed the design of the contour state representation we extend to our agents, and discuss in Chapter 5. The structure of the agent and the methods required by reinforcement learning were discussed in detail. We then ended the chapter by discussing the design requirements of a flexible Tetris application. The agent and application considerations discussed in this chapter were used in implementing the agents discussed throughout the rest of this thesis.

Chapter 4

The Melax-defined player

In this chapter we discuss the implemention of our first reinforcement learning agent according to Melax's specifications. We investigate the TD(0) value function, the constants utilised in it and a range of exploration methods. We also investigate the impact of the mirror symmetry optimisation and the performance of a Sarsa(0) agent.

4.1 Melax Tetris

Melax clearly defined the conditions under which his player functioned:

- Well has infinite height
- Well has a width of 6 blocks
- The game has a working height of 2 blocks — any blocks below this are discarded
- The game is limited to 10 000 tetrominoes
- The game uses a reduced tetromino set (see figure 2.3)
- The alpha and gamma values are defined as 0.02 and 0.8, respectively

Optimistic exploration was employed and the initial value estimates for each state were set to zero, which is optimistic when receiving purely negative rewards. The performance of the agent is gauged by the final height of the tetromino structure in the well. The lower the structure height, the more tightly the tetrominoes are packed and the better the performance of the agent.

We followed the Melax specifications and implemented the agent shown in figure 4.1.

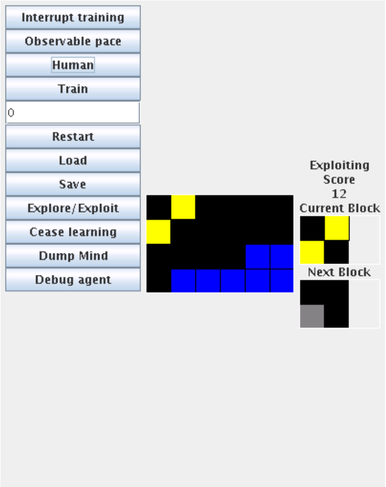


Figure 4.1: Our Melax-defined agent

4.2 Initial results

The agent was set to exploit its knowledge from the outset (greedy policy) and therefore had to rely on the optimistic initialisation values to drive its exploration. The results are shown in figure 4.2 and contrast the performance of our reinforcement learning player against other standards.

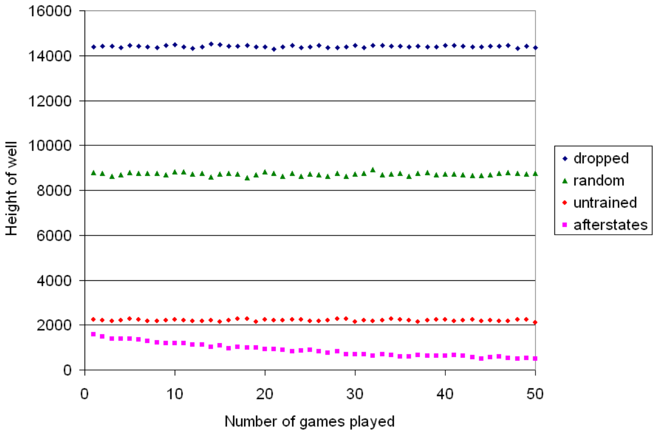


Figure 4.2: Results for the Melax-defined agent

The first standard is the maximum height of the structure which is determined by piling up tetrominoes without undertaking any translation or rotation. The second standard is the height of the structure when the blocks are randomly placed. The final standard is the performance of the agent when learning is disabled before any training has occurred, and the agent is set to exploit its knowledge. It is apparent that the untrained agent has insight into the placement of blocks, and this is supplied by the real time consideration of transitional reward performed by TD agents.

The agent starts at the level of the untrained agent and steadily improves. The eventual height of the well is roughly a quarter of the height of the untrained agent's well.

4.3 Mirror symmetry

We proceeded to implement the mirror symmetric optimisation of the Melax state space identified by Bdolah and Livnat [2000]. Figure 4.3 shows the improvement in the agent's performance when the mirror symmetry optimisation is included in the state description. Since all the mirror symmetrical states reference the same position in the table, these values are updated twice as often as they were before the optimisation. This leads to the tabular values converging on the correct values more rapidly. This is clearly reflected in the graphed results, and it is important to note that the final policy arrived at is the same in both circumstances. This vindicates the assumption that mirror identical states should have the same value associated with them and experimentally reinforces the scepticism shown in Chapter 2 when interpreting the results of Bdolah and Livnat [2000].

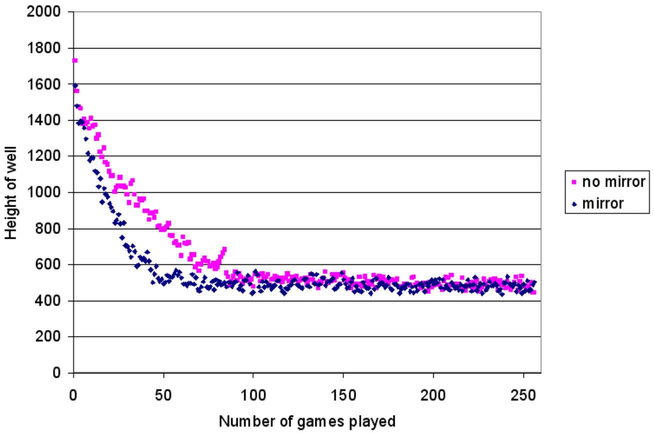


Figure 4.3: Performance impact of mirror symmetry

The adoption of the mirror symmetry optimisation results in the agent reaching its optimal policy after about fifty games, while the unoptimised agent takes about ninety games to reach its optimal policy.

4.4 Different exploration policies

We used our Melax agent to compare the performance of the different exploration methods. These exploration methods have several important characteristics. The rate of learning, the final policy achieved and the variation in the performance of the final policy are all characteristics we are interested in. These traits are not independently influenced by the exploration policies, but are also influenced by the theoretical constants discussed in Chapter 4.5.

The rate of learning becomes increasingly important as the state space of problems grow, as there is a threshold to experimental patience that the optimal policy may lie beyond. The final policy determines the final performance of the agent and is of obvious importance. Fluctuations in the performance of the final policy are less concerning, as they still allow the agent to identify the optimal policy and can be reduced through the adjustment of the aforementioned theoretical constants.

There is a distinction between the performance of the agent during exploration and the performance of the agent when exploiting its final policy. We therefore need to compare both sets of data in conjunction in order to extract a fair comparison. The agent is trained for a hundred games using each exploration policy and is then set to play fifty games with learning disabled and the agent set to exploit its knowledge. The different exploration methods are also separately tested in conjunction with optimistic exploration.

The ϵ-greedy method is set to explore 5 percent of the time. Setting the Softmax parameters introduces difficulties as the initial temperature, rate of change of temperature and final temperature must all be configured.

4.4.1 Optimistic exploration

It is apparent from figure 4.5 that the Softmax and ϵ-greedy methods settle on a similar policy. The greedy approach leads the agent to adopting a final policy that is superior to either of the other adopted policies. Figure 4.4 shows the greedy approach experiencing the most rapidly learning. This behaviour is predicted when the greedy policy is used in conjunction with optimistic exploration, since the other approaches explore amongst the unrealistic values rather then simply reducing them to discover the least unrealistic values. It is evident from figure 4.4 that all three agents converge on their final policies within forty games, but fail to improve their policies in the remaining sixty games. All three agents initially show exponential learning as they reduce their optimistic values to realistic ones, and consequently start utilising meaningful value estimations.

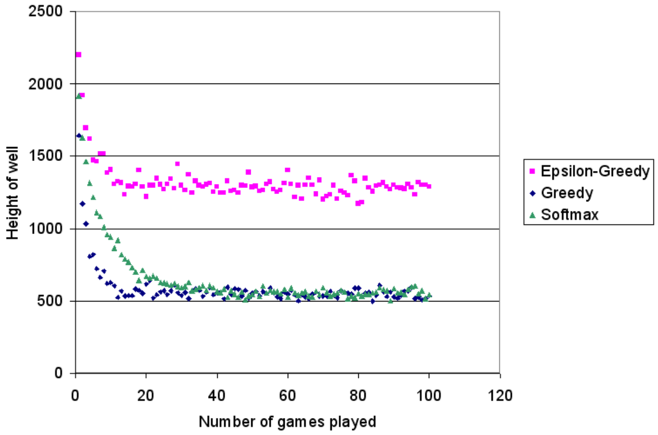


Figure 4.4: Exploring methods in conjunction with optimistic exploration

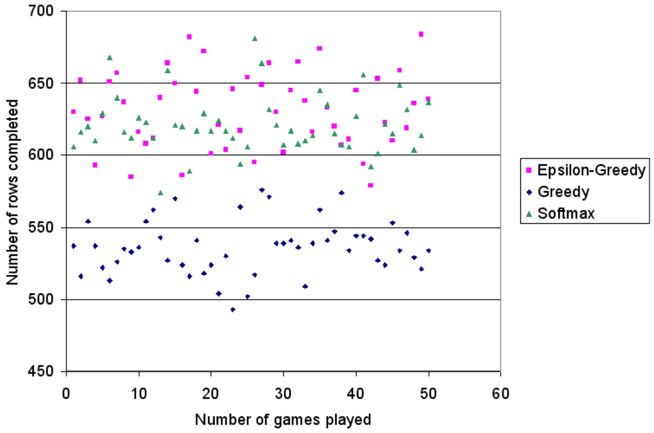


Figure 4.5: Exploitation of knowledge

4.4.2 Standard exploration

The value table of the previous optimistic agent had a minimum value greater then -150 after settling on the optimal policy. The starting values of the non-optimistic agent were therefore set to -150, and the exploration methods were then tested.

Figure 4.6 shows all three agents achieving far better initial results then they did in figure 4.4 due to the absence of misleading optimistic values. The ϵ-greedy agent experiences a brief spate of learning in the first few games and stumbles upon nothing useful in the remaining period. The greedy player stumbles onto a sub-optimal policy after less then 5 games, and then shows no further improvement following this. The initial height of the Softmax agent's well is due to indiscriminate exploration on the part of the Softmax agent. The Softmax player shows incredibly

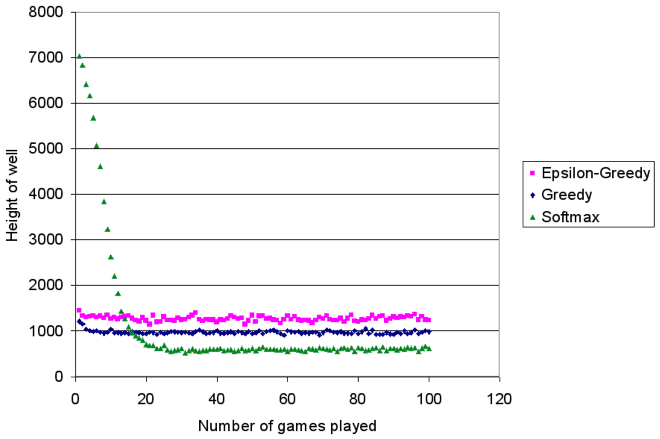


Figure 4.6: Exploring methods without optimistic exploration

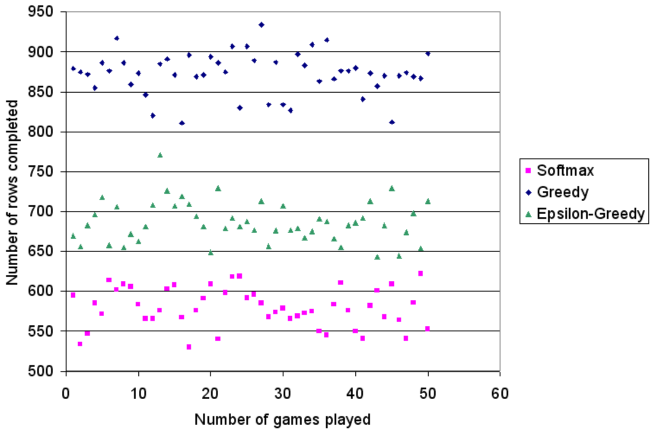


Figure 4.7: Exploitation of knowledge

rapid learning and settles on the optimal policy within 30 games.

The performance of the agents' final policies is shown in figure 4.7. The greedy approach results in the agent discovering the least rewarding policy, the ϵ-greedy approach results in the discovery of a better policy and the Softmax approach discovers the the best policy.

4.4.3 Conclusion

The Softmax approach proves its worth both with and without the aid of optimistic learning. The drawback with the Softmax approach is that although the results look to have changed the least in the comparison, it required separate implementations for the optimistic and non-optimistic testcases. Achieving ideal exploration requires discovering the optimal starting temperature, change in temperature and rate of change of temperature. The agent requires the full training period to explore between adjustments to the parameters, and as the state space gets larger the ideal values get increasingly vague and intangible. As the state space grows and the agent takes longer to explore, the iterative discovery of ideal learning parameters becomes increasingly time consuming.

The combination of optimistic learning and the greedy approach results in the agent learning incredibly rapidly and settling on the optimal policy. An added benefit is that the learning method requires no adjustments following a change in state space and the only quantity that has to be established in each investigation is the optimistic initialisation value.

Due to the convenient and desirable qualities of optimistic learning with the greedy policy, we utilised this exploration method in all future investigations.

4.5 Reinforcement learning constants

Melax found that the optimal policy was discovered with a γ term of 0.8 and an α term of 0.02. Through experimentation we discovered that increasing the α term to 0.2 almost increases the speed of learning by an order of magnitude as shown in figure 4.8. Adjusting the γ term has far less tangible results, and since this shifts the relationship between successive states it can have subtle consequences.

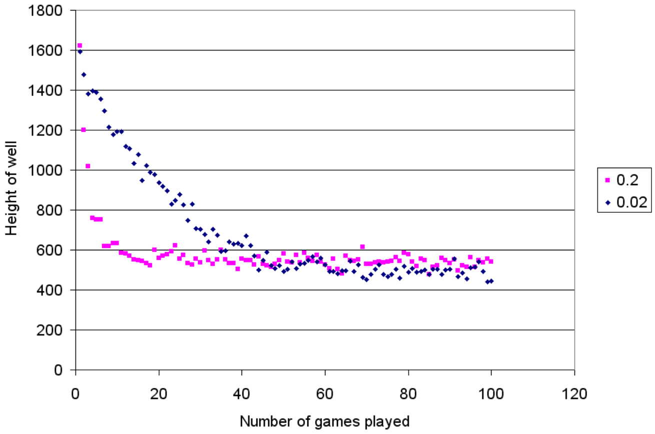


Figure 4.8: Impact of α on learning

An α value of 0.2 results in the agent settling on the optimal policy within ten games, while an α value of 0.02 results in the agent settling on the optimal policy after approximately fifty games. The agent with an α value of 0.02 eventually converges to a superior optimal policy, but the improvement in performance is small in proportion to the overall learning exhibited and the increase in training time is a serious drawback.

In all subsequent investigations we maintained an α value of 0.2 and a γ value of 0.8 unless otherwise stated.

4.6 Sarsa(λ) agent

Melax mentions having implemented a Sarsa agent without eligibility traces (Sarsa(0)) and reports no appreciable difference in the results achieved by the agent.

In adopting Sarsa(0) we have to expand the state space. Melax's reduced piece set contains five different tetrominoes, the well is six blocks wide and any grid based game is restricted to four orientations. There are therefore approximately $5 \times 6 \times 4$ transitions off each state and the size of the state space is increased from 4096 states (2^{12}) to 491520 states (approximately 2^{19}). The impact of this is that the Sarsa(0) agent takes an incredible amount of time to train as the results in figure 4.9 show.

As mentioned earlier, this Sarsa agent does not utilise mirror symmetry.

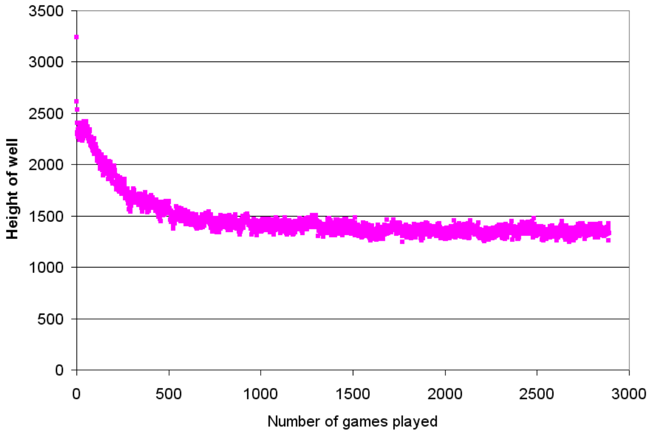


Figure 4.9: Performance of Sarsa(0) agent

A large amount of this exploration time is wasted exploring redundant states, since only the L shaped Melax tetromino has four unique orientations, while all the others only have two unique orientations. There is further storage wastage as only the single block tetromino can move across the full span of the well regardless of the orientation of the block.

After learning plateaus off the agent is capable of constructing a well structure around one thousand four hundred rows high. These results are far worse then those achieved by the TD(0) agent. The TD(0) agent only takes fifty games to find the optimal policy in contrast to the Sarsa(0) agent who takes around eight hundred games to discover a sub-optimal policy. This increase in training time is to be expected with the expansion of the state space and the absence of mirror symmetry, but the poor final results of the Sarsa(0) are not easily explained. Observing both agents play does not reveal any obvious short sightedness on the part of the Sarsa(0) player.

The Sarsa(0) agent has all the state information available to the TD(0) agent, and increased perception since it develops a value for every action. What is more disturbing is that in Chapter 5 the same agent, functioning within a different Tetris definition and updating its value function with the aid of eligibility traces, completely outperforms the competing TD(0) agent.

As mentioned at the introduction of TD(0), this approach includes the current transitional reward in evaluating the value of a transition. The Sarsa agent lacks this real time aid and this is only possible source of ignorance we can identify. The approaches are quite distinct, with the Sarsa(0) relying purely on its state-action table while the TD(0) agent uses its value table as a supplement to real time information. This task may simply be suited towards the latter approach. It is also important to note that any agent that is initialised with a uniform value function, and factors transitional rewards in its value appraisal, will choose the transitions associated with immediate reward. This leads the agent to explore these transitions to a larger extent than the surrounding transitions, and may well direct the agent towards the optimal policy for this game. Although the Sarsa(0) agent does not consider the transitional rewards at run time, it should still be utilising these rewards in the updating of its state-action table and should therefore be aware of them. It is possible that the agent was not trained for long enough, and this leads to the final consideration.

Given enough training time, the Sarsa(0) agent might have experienced a discontinuous jump in performance and discovered the optimal policy that the TD(0) agent is using. This type of performance jump is evident in later agents, and the sheer size of the state space may have prevented the agent from chancing onto this within a reasonable period of time(one full day of training). It is important to note that the relationship between the increase in state space and the increase in training time is nonlinear. In converging on an accurate value estimation, the agent moves both towards and away from the correct value. Increasing the size of the state space a hundred fold therefore has an indeterminable impact on the agent's learning time.

4.7 Conclusion

In this chapter we discussed the results achieved by our Melax-defined reinforcement learning player and discussed the results of further experimentation with the value function constants, mirror symmetry, exploration methods and a Sarsa(0) agent. The results determined in this chapter set many experimental parameters used in subsequent investigations, such as the value of α and the use of the optimistic-greedy exploration policy.

Chapter 5

Contour Tetris

The implementation of the Melax agent supplied a great deal of insight in applying reinforcement learning to Tetris, which we used in implementing two agents using the state representation designed in Chapter 3.1. These agents use TD(0) and Sarsa(λ) respectively and we discuss the performance of our agents with the intention of extending them to the full well.

5.1 Initial considerations

We continued to use the Melax tetromino set, shown in figure 2.3, in the implementing of our contour agents. It was not apparent how large a performance penalty would be incurred in changing sets, but initial results acquired with the standard tetromino set made comparison of the competing algorithms difficult. We wanted to see if learning was achievable, and the reduced tetromino set was utilised as a means of exaggerating the learning in order to compare results. We assumed that if learning was attained, it would be retained when using different tetromino sets, although almost certainly to a lesser extent.

In creating our agents, we had to decide on a reward policy. Initially we wanted to lead our agents as little as possible and afford them the freedom to determine their own policies. Melax's punishing of the agent for increasing the height seemed too prescriptive, and we initially opted to rather reward the agent for simply completing rows. This met with initial success when used with the TD(0) agent however the Sarsa(λ) agent performed poorly due to the sparsity of rewards.

After a great deal of experimentation we adopted a Melax like reward scheme for the Sarsa(λ) agent. We settled on a reward scheme where the agent is punished one hundred points for each block in height the well is increased and forty points for every hole that is covered in taking a transition. These final penalties were arrived at through trial and error. The covered hole penalty is limited to a maximum of three covered holes in order to prevent the formation of persistent vallies in the well. Without this restriction these vallies become too formidable to be contained and therefor lead to the eventual termination of the agent. The agent cannot perceive the number of holes introduced in a transition with our state representation, and this information is stored in the value associated with a transition leaving a state.

All comparison charts plot the performance of the learning agent alongside the performance of an untrained non-learning agent. This indicates the degree of learning achieved and allows us to view the performance of the agent in the context of performance of the untrained agent. This consideration is maintained throughout this thesis as we consider it important to draw a distinction between the inherent performance of the agent and the performance of the agent due to learning.

Early in the investigation it became apparent that the tactics utilised in a well of width four might differ from those required for a wider well, and we therefore also investigated the performance of the agents in wells of width five. Extending the wells beyond a width of five leads to state spaces that are unreasonably large and we therefore operate within these limits.

For the sake of conciseness the agent trained in a well of width four is referred to as four and the agent trained a well of width five is referred to as fiver for the rest of the thesis. These titles are then prefixed with the reinforcement method to unique identify the agents.

5.2 TD(0) agent

The performance of the TD(0) agent is shown for wells of width four and five in figures 5.1 and 5.2, respectively.

The TD(0) agent performs well in a reduced well four blocks wide. After nineteen games the performance jumps two orders of magnitude and the agent proceeds to complete over a hundred thousand rows the majority of the time, and occasionally manages to complete almost a million rows.

This explosive jump in performance could be due to two factors. The most obvious is that the agent may have discovered a significant series of improvements to its policy. What is less obvious is that the agent learns after every single move, so the longer the agent survives the more the agent learns and a self sustaining cycle of survival and learning is achieved.

If the untrained TD(0) agent has its learning ability disabled, and is instructed to exploit its knowledge, it still completes rows and takes transitions that offer it immediate reward. This is due to the TD(0) agent's consideration of the value of the transition in conjunction with the

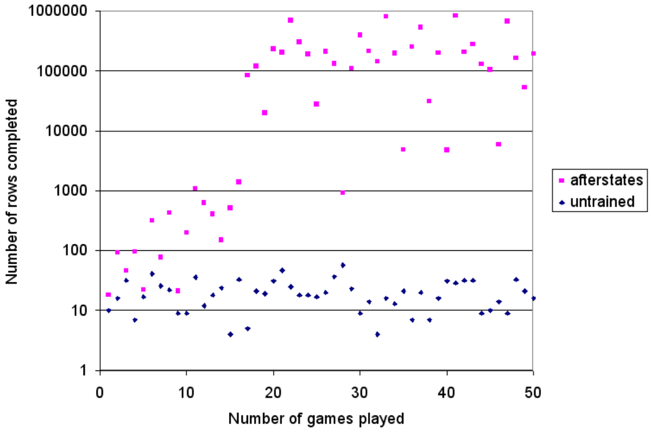


Figure 5.1: TD(0) agent in well of width four

value of the destination state. If all the values in the value function are initialised to a common value, the agent will always take the transition that offers it immediate reward. This is the factor responsible for the high baseline plotted alongside every TD(0) chart.

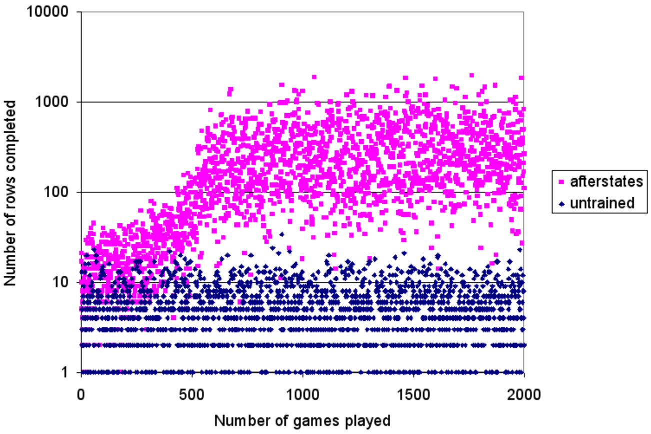


Figure 5.2: TD(0) agent in well of width five

As the well width is increased the performance of the TD(0) agent deteriorates drastically. With a well width of five the agent takes about seven hundred games to discover the optimal policy, which results in the agent completing around three hundred rows per game.

The reasons for the sudden drop in performance become obvious when watching TD fiver play. Due to the contour representation designed in Chapter 3, the agent is oblivious to the presence of holes and introduces them liberally in trying to achieve immediate rewards. These tactics are also used by TD four. The difference in performance between the agents is therefore explained by the increasing importance of covered holes as the well width is extended. Narrow wells experience a high turnover of rows, and covered holes are therefor rapidly re-exposed and have very little impact on the agent. As the well width increases the number of pieces required to complete a row increases and the lifespan of a covered holes is extended. It is important to realise that as the well widens the importance of a single hole increases as this is all that is required in order to render a row uncompletable. This is also exacerbated by the fact that the wider the well, the more blocks that are required to complete a row and the greater the likelihood of introducing a covered hole. The reward function adopted for the TD(0) agent could also be contributing to the decline in performance, since as the rate of row completion drops the rewards get increasingly sparse and the agent receives increasingly little feedback from the reward function. The net effect is that increasing the width of the well by a single block can have serious implications for the performance of the TD(0) agent.

TD four's results look deceptively impressive, as the agents displays obvious learning. What the agent actually learns is how to exploit the description of the well in order to receive the maximum amount of reward. The agent achieves this by placing the tetrominoes across the well at their widest orientation, which almost completely bridges the width of the well and almost guarantees the attainment of reward in the subsequent move. This explains the agents strong preference for laying tetrominoes down horizontally, and it is this tendency that introduces covered holes.

These tactics are not what we were trying to achieve, and will obviously perform exceedingly poorly in a well the width of the full Tetris game. These results reveal two dangers surrounding the reward function. The first is that the agent will learn to do whatever is required in order to get rewards. Rewarding the agent for completing rows therefore results in an agent that completes rows, rather then an agent that builds solid structures or minimises height. The TD agent did exactly what was demanded of it and became incredibly qualified at dropping the tetrominoes in order to complete rows within a narrow structure. This did not correspond to the behaviour we envisaged, but corresponded to the set task. The second danger lies in rewarding a varying task. Rather then having a problem description with a set reward and a variable state space, we actually implemented an agent that experienced a decrease in the frequency of rewards, with an increase in the size of the state space.

We tried to correct for the behaviour of the TD(0) agents by adopting a real time penalty for introducing covered holes. This created the problem of weighting the hole penalty against the row completion reward. The problem was further aggravated by the fact that introducing holes does not have uniform impact. In some circumstances introducing a hole has little impact, and in others it can completely negate the value of the transition. This is best shown with the L block from the standard tetromino set and is shown in figure 5.3.

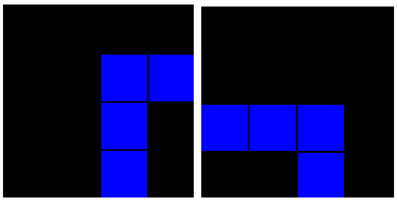


Figure 5.3: Contrasting hole inclusion

The introduction of two vertical holes has a far more damaging effect than introducing two horizontal holes as it effects two separate rows. In introducing the two horizontal holes the L block has also extended across three quarters of the reduced well, which places the agent in a likely position to receive reward in the following transition.

Punishing a complete contour because a hole was introduced in one of the transitions leaving it makes little sense, and the introduction of the covered hole penalty actually had a negative impact on the agent playing in the well of width four. The tactics of the agent did not perceivably change but made there was a undeniable decreased in the number of rows completed per game. The agent in the well of width five started to avoid introducing holes, but did not improve appreciably.

5.3 Sarsa(λ) agent

The performance of the Sarsa(λ) agent is shown for wells of width four and five in figures 5.4 and 5.6.

The Sarsa(λ) agent performs incredibly in the four block wide well as shown in figure 5.4. Although the Sarsa(λ) agent takes longer to train then the TD(0) player, the performance explosion that occurs at around seventy six games is unparallelled in this thesis. The agent was left to play for a day and had to be forcibly terminated in order to end the game and free up the computer it was running on. Figure 5.5 shows the game at the time of termination

The agent had completed twelve million rows and had an almost empty well as shown in figure 5.5.

Figure 5.6 shows the Sarsa(λ) agent, functioning in a well of width five, steadily improving over the course of its lifespan. After around seven hundred games the rate of improvement increases and the agent reaches the optimal policy at around nine hundred games. At this point the agent is capable of completing around ten thousand rows per game.

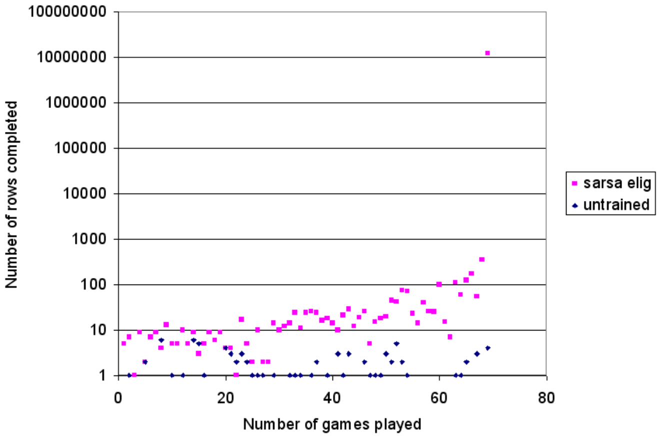


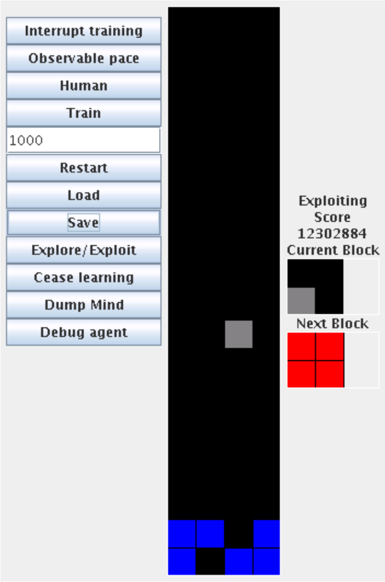
Figure 5.4: Sarsa agent in well of width four

The Sarsa(λ) agent displays very different in-game tactics to the TD(0) agent. The player places blocks intelligently, and creates a solid well structure. This is due to the fact that the agent is aware of introducing holes, and the associated penalty is incorporated in the relative state-action transition. This representation is very intuitive, since the same transition with the same tetromino off the same contour will always introduce an identical number of covered holes.

The drop in performance that becomes evident in the altering of the well size may be due to qualities associated with the well rather short comings on the part of the agent. There is no mathematical proof guaranteeing the termination of the Tetris game when using the reduced tetromino set and there are no obvious combinations of tetrominoes that would be terminal in the well of width four. It is possible that the game using the reduced piece set and having width four could be played indefinitely. On the other hand, a long sequence of square tetrominoes in the well of width five would completely overwhelm the agent. This may well account for the wider agents inability to play indefinitely like the agent functioning in the well of width four.

5.4 Conclusion

We implemented TD(0) and Sarsa(λ) agents. These agents used the state representation developed in Chapter 3.1 , and therefore trained in wells of width four. The agents both achieved impressive results within these wells. However, it was evident from watching the agents play that the tactics they had adopted could not be expected to produce results of the same calibre in the full well. The agents were then trained in a well of width five. The resulting TD(0) agent showed a drastic drop in performance and no improvement in its tactics. The Sarsa(λ) performed



1000

Exploiting
Score
12302884

Figure 5.5: Sarsa agent at time of termination

convincingly in the well of width five, and showed the most promise in tackling the full Tetris game.

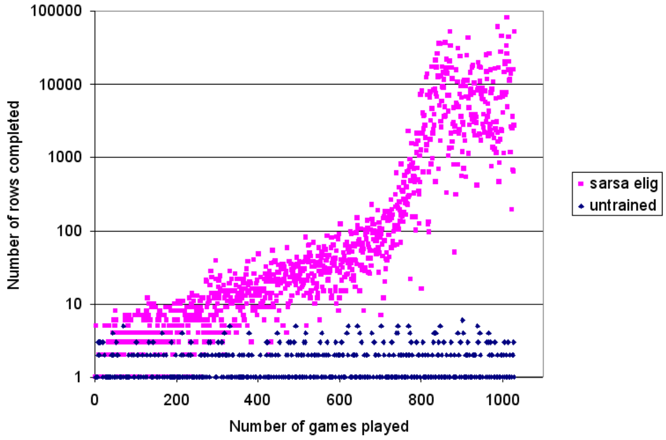


Figure 5.6: Sarsa agent in well of width five

Chapter 6

Full Tetris

In this chapter we extend our contour agents to play the full Tetris game. There are three distinct components to this task, and they each warrant separate attention. The first consideration is the extension of the reduced game to the full game as discussed in Chapter 3.2.3. The standard Tetris tetromino set (shown in figure 2.2) has to be adopted, and we discuss the impact this has on our agents' learning. We then test the performance of our original agents when functioning within the full Tetris well and using the reduced blockset. Finally, we adopt both modifications and extend our agents to the full game.

The results are a great deal noisier then the previous chapters results and consequently have to be averaged across a large number of games. We average the results across twenty five games unless otherwise stated.

6.1 Extending the contour agent to the full game

The two competing methods we suggested for translating the subtransitions into transitions were investigated using the Sarsa(λ) agent. We also investigated the validity of the extension approach for two separate extensions to ensure that our results were consistent. The agents used in this investigation were fully trained and had settled on their optimal policies as shown in figures 5.4 and 5.6, respectively. Learning was then disabled and the agents were set to exploit their knowledge.

Figure 6.1 shows Sarsa(λ) four, playing in a full well of width 5. This extention allows the direct comparison of the extended agent's results against Sarsa(λ) four's results shown in figure 5.4 and Sarsa(λ) fiver's results shown in 5.6. We had hoped that the extension method would retain the original performance of Sarsa(λ) four or achieve the performance of Sarsa(λ) fiver.

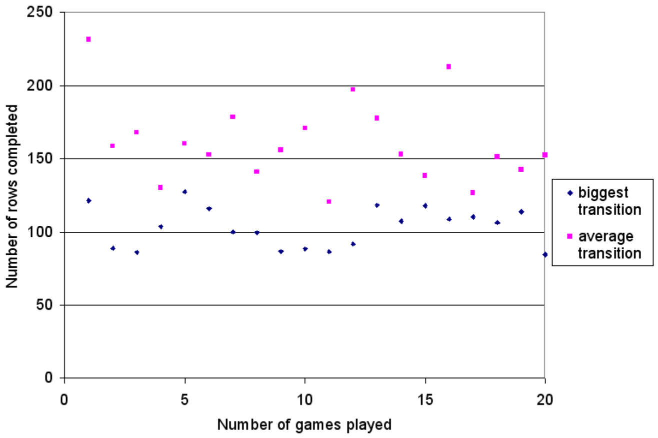


Figure 6.1: Agent trained in subwell of width four playing in well of width five

It is painfully evident that this is not the case, and that the extended agent lacks the abilities of either of its target agents.

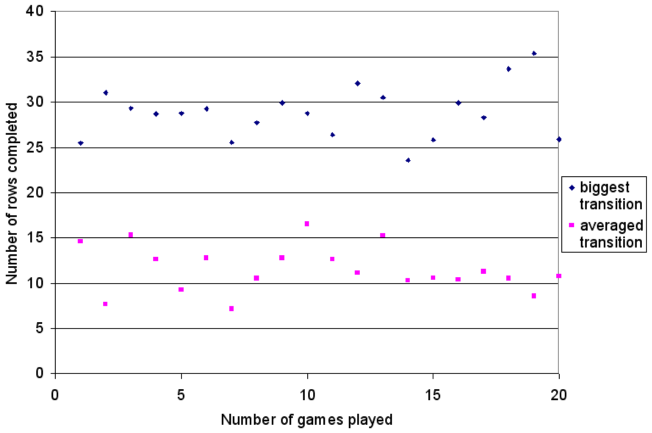


Figure 6.2: Agent trained in subwell of width five playing in well of width ten

Figure 6.2 shows the results of Sarsa(λ) fiver being extended to play in a well of width ten. As the difference in width between the conceptual well and the full well increases, we would expect the agent's value function to become increasingly inaccurate and this is shown in comparing figure 6.1 and figure 6.2. This may be due to the fact that we are effectively using extrapolation to extend the ideal policy from the reduced well to the full well, and the further we extrapolate the less accurate the results obtained become. This may also be due to the fact that as the difference in width between the conceptual and full well increases the original subtransitions are increasingly manipulated or viewed in isolation, corresponding to the averaging method or biggest transition method, respectively.

Sarsa(λ) fiver still exhibits ability, although any comparrison with its original results makes the extended results seem numerically unacceptable. The agent performs erraticly when using the averaging approach, and is as inclined to build upwards as it is to build sideways, and after initially selecting a direction it continues along this path. This resulted in the agent achieving zero rows regularly, and completing a flood of rows alternatively. When the agent uses the biggest transition method it placed blocks intelligently, and its original tactics are still identifiable. The agent occasionally makes a bad decision and introduces a covered hole, but still packs the well tightly and intelligently the majority of the time.

It is clear from figure 6.1 that the averaging of the subtransitions offers a distinct advantage over taking the largest subtransition alone, as the averaging method performs consistently better. The exact opposite result is shown in figure 6.2, where the selection of the biggest subtransitions inspires the best performance on the part of the agent. There is no obvious reason why the different transition methods perfom better in certain well extensions. It is possible that in extensions where the conceptual well is of a similar size to the full well, the averaging approach will perform best as the averaging process has little effect on the subtransitions. In circumstances where the full well is far larger then the conceptual well, the averaging method will possibly result in meaningless transitions, and the selection of the biggest transition will retain a single meaningful transition that offers the best chance of intelligent placement. This would explain the apparantly conflicting results shown above, and is supported by the behaviour of the agent in both extensions.

In extending the well we deliberately avoided manipulating the value function or introducing external terms as a means of improving the performance of the agent. The justification behind the adoption of our extension method was that if the agent is offered a range of transitions, taking the best transition will lead the agent to build across the well. This relied on the assumption that the agent would build across the well and try to maintain a set of valuable contours across the full well rather then cycle through all the possible contours within a single subwell.

The extension we are actually interested in is the extenion of the contour space to the full game, and this is a transition from a small conceptual well to a large full well. We therefore adopt the biggest transition approach in our subsequent investigations.

6.2 Training subwell size vs performance

The size of the subwell the agent was trained in was predicted to have an impact on the performance of the agent operating in the full well. This extendability of different width subwells was investigated using the Sarsa(λ) agent and the results are shown in figures 6.3 and 6.4.

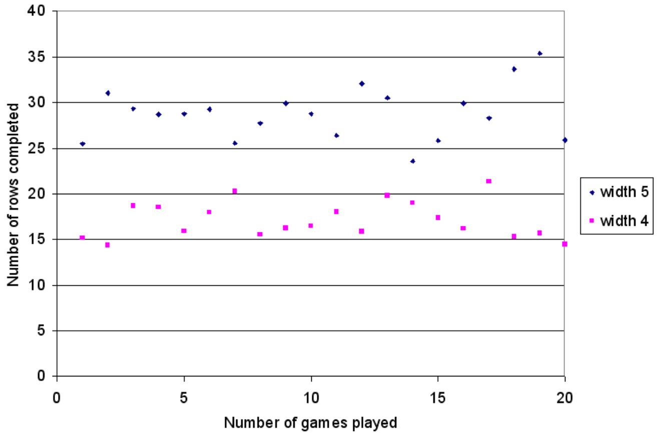


Figure 6.3: Comparison of extendability of subwell sizes with reduced tetrominoes

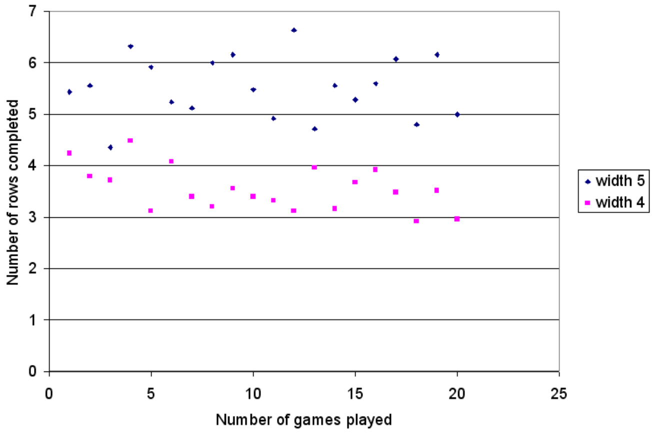


Figure 6.4: Comparison of extendability of subwell sizes with standard tetrominoes

Both investigations revealed a distinct advantage to Sarsa(λ) fiver.

Although the final state space design was chosen to represent a well four blocks wide, agents trained in narrow wells developed a policy which favoured easy returns. In narrow wells there is a rapid turnover of rows since two tetrominoes on average complete a row. Introducing holes therefore has minor implications as rows have a short lifespan and the obstructing layer is likely to removed in the ensuing block placements. This performance does not scale out to the full representation of the game, where a single introduced hole can devalue a series of successive compact placements and therefore have long term implications.

By extending the width of the reduced well the agent trains within a broader well and tends towards a policy that reflects the policy required in playing the game. We narrowed the well in order to reduce the state space, and every extra unit of width increases the state space by a factor of seven for our contour representation. We therefore seek to strike a balance between increasing the state space and giving the agent access to the environment we want him to eventually solve.

The final agents adopted were therefore all trained in a well of width five.

6.3 Extension to the full Tetris well

The performance of Sarsa(λ) fiver and TD(0) fiver are reflected in figures 6.3 and 6.3. Learning was disabled, both Sarsa(λ) fiver and TD(0) fiver were set to exploit their knowledge and then one hundred games were played by each agent.

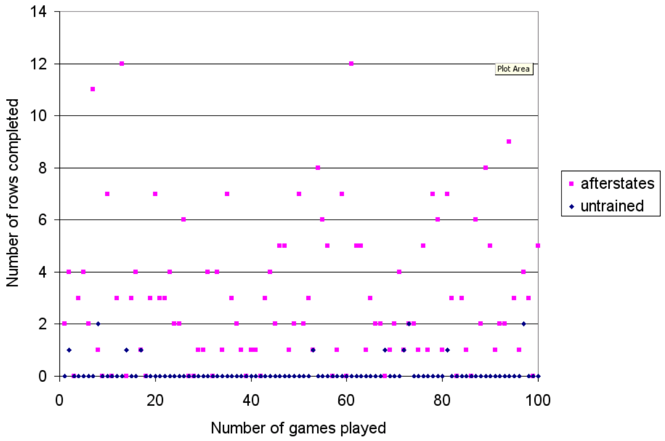


Figure 6.5: TD(0) agent in full well with reduced tetrominoes

Both figure 6.6 and show a great deal of scatter, which is a direct result of our extension method. The results shown for TD(0) are fairly unimpressive, but this was anticipated following the behaviour of TD(0) fiver in Chapter 5. Considering the limited success we have had with our extension method, the results for Sarsa(λ) fiver are fairly impressive. Sarsa(λ) fiver achieves an average of 30 rows while TD(0) achieves an average of 4 rows.

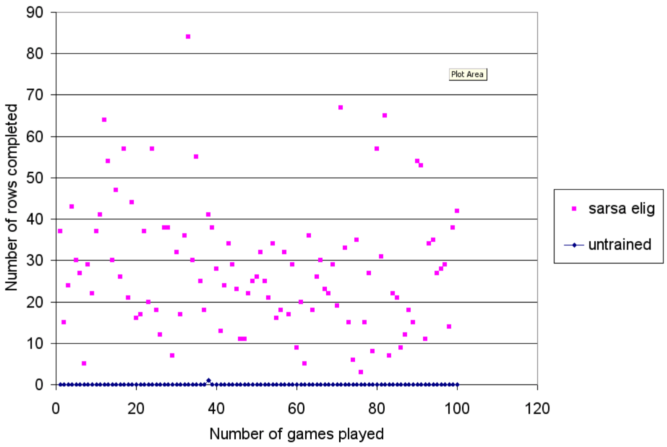


Figure 6.6: Sarsa(λ) agent in full well with reduced tetrominoes

6.4 Extension to the full Tetris tetromino set

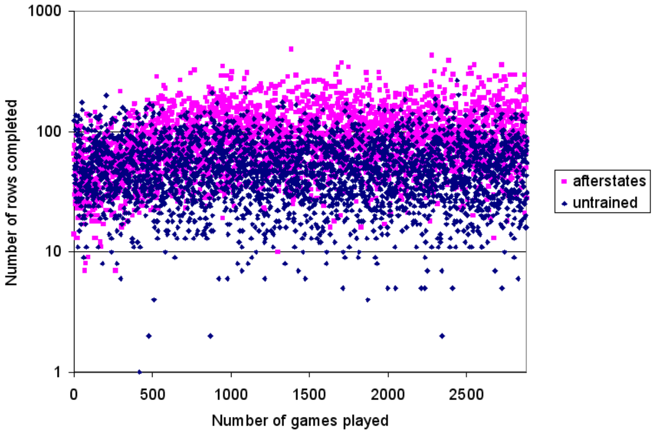


Figure 6.7: TD(0) agent in reduced well with full tetrominoes

6.5 Extension to the full Tetris game

6.6 Conclusion

In this chapter we extended our contour agents to play in the full Tetris well.

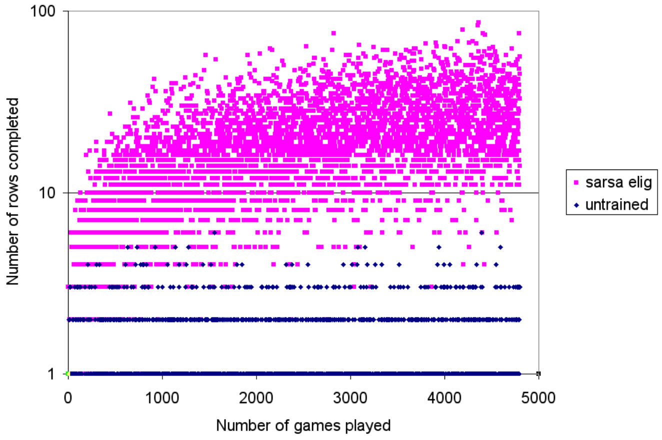


Figure 6.8: Sarsa(λ) agent in reduced well with full tetrominoes

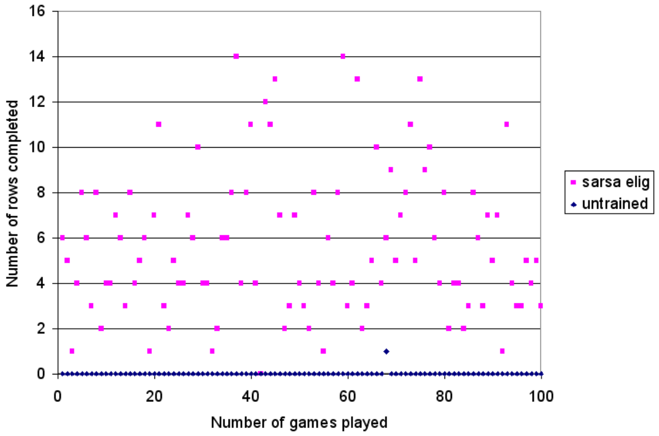


Figure 6.9: Sarsa(λ) agent playing full Tetris

Chapter 7

Concluding remarks

The TD(0) agent achieved incredible results when playing within the original subwell. These results were unrealistically high, and dropped off sharply as the well was increased and real time rewards became more intermittent. The TD(0) representation neglected the importance of different transitions.

The Sarsa(λ) player achieved incredible results within the original subwell defined in the state space reduction. The performance was still impressive as the well width was increased, although the required training time greatly increased.

The reducing of the state representation had a greater impact on the TD(0) approach, since this approach depends purely on the accuracy of the state information. The extended resolution afforded the Sarsa(λ) player enabled it to associate a value with the act of transitioning off a state.

7.1 Possible extentions

The contour agents performed well in the narrow well. The Sarsa(λ) agent in particular showed a great deal of promise, and the extension to the full well did not seem to do the agent justice. It is tempting to try to re-achieve the earlier performance by introducing real time modifications to the data, but this would start to shift the agent away from reinforcement learning and weaken the influence of its value function. Our reduced contour space assumed the existence of

Softmax seems like an incredibly useful approach to exploring the state space of an agent. The results achieved in Chapter 4, and undocumented performance achieved with the contour players all indicate that it is a very sensible approach to the exploration problem. It is, however, exceedingly difficult to implement and requires a continual refinement of the parameters until the desired behaviour is achieved. Redefining the method in terms of tangible values, such as minimum exploration rate for the poorest transition, which in term set the required parameters would provide the power of Softmax without the implementation overhead. This could well be exceedingly difficult but would be an incredibly interesting extension.

Bibliography

Jonathan Baxter, Andrew Trigdell, and Lex Weaver. Knightcap: a chess program that learns by combining TD(λ) with game-tree search. In *Proc. 15th International Conf. on Machine Learning*, pages 28–36. Morgan Kaufmann, San Francisco, CA, 1998. URL `http://citeseer.ist.psu.edu/baxter98knightcap.html`.

Yael Bdolah and Dror Livnat. Reinforcement learning playing tetris. 2000. URL `http://www.math.tau.ac.il/~mansour/rl-course/student\_proj/livnat/tetri%s.html`.

Justin A. Boyan and Michael L. Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6, pages 671–678. Morgan Kaufmann Publishers, Inc., 1994. URL `http://citeseer.ist.psu.edu/boyan94packet.html`.

Ron Breukelaar, Erik D. Demaine, Susan Hohenberger, Hendrik Jan Hoogeboom, Walter A. Kosters, and David Liben-Nowell. Tetris is hard, even to approximate. *International Journal of Computational Geometry & Applications*, 14:1-2:41, 2004. URL `http://theory.csail.mit.edu/~dln/papers/tetris/tetris.pdf`.

John Brzustowski. Can you win at tetris? Master's thesis, University of British Columbia, 1992.

Heidi Burgiel. How to lose at tetris. *Mathematical Gazette*, 81:491:194–200, 1997. URL `http://www.findarticles.com/p/articles/mi\_qa3773/is\_199803/ai\_n87851%30`.

R. H. Crites and A. G. Barto. Improving elevator performance using reinforcement learning. In D. S. Touretzky, M. C. Mozer, and M. E. H., editors, *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*, pages 1017–1023, Cambridge, MA, 1996. MIT Press.

Kurt Driessens. *Relational Reinforcement Learning*. PhD thesis, Catholic University of Leuven, 2004. URL `http://www.cs.kuleuven.ac.be/~kurtd/PhD/`.

Colin P. Fahey. Tetris specifications & world records, 2003. URL `http://www.colinfahey.com/2003jan\_tetris/2003jan\_tetris.htm`.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, Inc., 1998.

Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996. URL `http://citeseer.ist.psu.edu/kaelbling96reinforcement.html`.

H. Kimura, K. Miyazaki, and S. Kobayashi. Reinforcement learning in POMDPs with function approximation. In *Proc. 14th International Conf. on Machine Learning*, pages 152–160. Morgan Kaufmann, San Francisco, CA, 1997. URL `http://www.fe.dis.titech.ac.jp/~gen/robot/robodemo.html`.

Clinton Brett McLean. Design, evaluation and comparrison of evolution and reinforcement learning models. Master's thesis, Rhodes University, 2001.

Stan Melax. Reinforcement learning tetris example. 1998. URL `http://www.melax.com/tetris/`.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, 2002. URL `http://www.cs.ualberta.ca/sutton/book/ebook/index.html`.

Richard S. Sutton, Gregory Kuhlmann, and Peter Stone. Reinforement learning for robocup-soccer keepaway, 2005.

Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38 (3), 1995. URL `http://www.research.ibm.com/massive/tdl.html`.

Sebastian Thrun and Anton Schwartz. Issues in Using Function Approximation for Reinforcement Learning. In M. Mozer, P. Smolensky, D. Touretzky, J. Elman, and A. Weigend, editors, *Proceedings of the 1993 Connectionist Models Summer School*, Hillsdale, NJ, 1993. Lawrence Erlbaum. URL `http://citeseer.ist.psu.edu/thrun93issues.html`.