USING VALUE ITERATION TO SOLVE SEQUENTIAL DECISION PROBLEMS IN GAMES

Thomas Hartley, Quasim Mehdi, Norman Gough
The Research Institute in Advanced Technologies (RIATec)
School of Computing and Information Technology
University Of Wolverhampton, UK, WV1 1EL
E-mail: T.Hartley2@wlv.ac.uk

KEYWORDS

Value iteration, artificial intelligence (AI), AI in computer games.

ABSTRACT

Solving sequential decision problems in computer games, such as non-player character (NPC) navigation, can be quite a complex task. Current games tend to rely on scripts and finite state machines (FSM) to control AI opponents. These approaches however have shortcomings; as a result academic AI techniques may be a more desirable solution to solve these types of problems. This paper describes the process of applying the value iteration algorithm to an AI engine, which can be applied to a computer game. We also introduce a new stopping criterion called game value iteration, which has been designed for use in 2D real time computer games and we discuss results from experiments conducted on the AI engine. We also outline our conclusions which state that the value iteration and the newly introduced game value iteration algorithms can be successfully applied to intelligent NPC behaviour in computer games; however there are certain problems, such as execution speed, which need to be addressed when dealing with real time games.

INTRODUCTION

Whilst playing computer games online against human opponents, it became apparent that it was a more interesting playing experience, than that of playing against non-player characters (NPCs). The human opponents were more difficult to anticipate and were more challenging, in comparison to their NPC counterparts. As a result, we tend only to play the single player aspect of a computer game a handful of times before we feel the game's gameplay becomes predictable and easy to beat.

This is backed up by Jonathan Schaeffer (2001 in Spronck *et al.*, 2003) who states that the general dissatisfaction of game players with the current levels of AI for computer controlled opponents makes them prefer human controlled opponents. Currently commercial computer game AI is almost exclusively controlled by complex "manually-designed scripts" (Spronck *et al.*, 2002). This can result in poor AI or "Artificial Stupidity" (Schaeffer, 2001 in Spronck *et al.*, 2002).

The predictability and any "holes" within a scripted computer game can then be exploited by the human player (Spronck *et al.*, 2002). The game industry is however constantly involved in employing more sophisticated techniques for NPCs (Kellis, 2002), especially in light of the increase in personal PC power, which enables more time to be spent processing AI. Recent games, such as Black & White (Lionhead, 2001) use learning techniques to create unpredictable and unscripted actions. However most games still do rely on scripts and would benefit from an improvement in their AI.

These observations formed the basis of a research project into the field of AI and computer game AI. The aims of this project were to research computer games in order to shed light on where computer game AI can be poor and to research AI techniques to see if they might be able to be used to improve a computer game's AI. The objectives of the project were the delivery of a computer game AI tool that demonstrated how an AI technique could be implemented as an AI engine and a computer game that demonstrated the engine. This paper demonstrates how Markov decision processes can be applied to a computer game AI engine, with the intention of showing that this technique will be a useful alternative to scripted approaches. This paper covers the implementation of the AI engine; the implementation of the computer game will be covered in our next paper.

MARKOV DECISION PROCESSES

Markov decision processes (MDPs) are a mathematical framework for modelling sequential decision tasks / problems (Bonet, 2002) under uncertainty. According to Russell and Norvig, (1995), Kristensen (1996) and Pashenkova and Rish (1996) early work conducted on the subject was by R. Bellman (1957) and R. A. Howard (1960).

The technique works by splitting an environment into a set of states. An NPC moves from one state to another until a terminal state is reached. All information about each state in the environment is fully accessible to the NPC. Each state transition is independent of the previous environment states or agent actions (Kaelbling and Littman, 1996). An NPC observes the current state of the environment and chooses an action. Nondeterministic effects of actions are described by the set of transition probabilities (Pashenkova and Rish, 1996). These transition probabilities or a transition model (Russell and Norvig, 1995) are a set of probabilities associated with the possible transitions between states after any given action (Russell and Norvig, 1995). For example the probability of moving in one direction could be 0.8, but there is a chance of moving right or left, each at a probability of 0.1. There is a reward value for each state (or cell) in the environment. This value gives an immediate reward for being in a specific state.

A policy is a complete mapping from states to actions (Russell and Norvig, 1995). A policy is like a plan, because it is generated ahead of time, but unlike a plan it's not a sequence of actions the NPC must take, it is an action that an NPC can take in all states (Yousof, 2002). The goal of MDPs is to find an optimal policy, which maximises the expected

utility of each state (Pashenkova and Rish, 1996). The utility is the value or usefulness of each state. Movement between states can be made by moving to the state with the maximum expected utility (MEU).

In order to determine an optimal policy, algorithms for learning to behave in MDP environments have to be used (Kaelbling and Littman, 1996). There are two algorithms that are most commonly used to determine an optimal policy, however other algorithms have been developed, such as the Modified Policy Iteration (MPI) algorithm (Puterman and Shin, 1978) and the Combined Value-Policy Iteration (CVPI) algorithm (Pashenkova and Rish, 1996).

The two most commonly used algorithms for determining an optimal policy have a foundation and take inspiration from Dynamic Programming (Kaelbling and Littman, 1996) which is also a technique for solving sequential decision problems. In addition problems with delayed reinforcement are well modelled as MDPs (Kaelbling and Littman, 1996). There are many algorithms in the area of reinforcement learning (For example: Q learning) that address MDP problems (Mitchell, 1997), in fact understanding Finite MDPs are all you need to understand 90% of modern reinforcement learning (Sutton and Barto, 2000).

The two most commonly used algorithms are value iteration (Bellman, 1957) and policy iteration (Howard, 1960). The value iteration (VI) algorithm is an iterative process, which calculates the utility of each state, which is then used to select an optimal action (Russell and Norvig, 1995). The iteration process stops when the utility values converge. Convergence occurs when utilities in two successive iterations are close enough (Pashenkova and Rish, 1996). The degree of closeness can be defined by a threshold value. This process was however, observed to be inefficient, because the policy often becomes optimal long before the utility estimates reach convergence (Russell and Norvig, 1995). Because of this another way of finding an optimal policy was suggested. It is called policy iteration.

The policy iteration (PI) algorithm generates an initial policy, which usually involves taking the rewards of states as their utilities (Pashenkova and Rish, 1996). It then calculates the utilities of each state, given that policy (Russell and Norvig, 1995). This is called value determination (Pashenkova and Rish, 1996; Russell and Norvig, 1995). It then updates the policy at each state using the new utilities. This is called policy improvement (Pashenkova and Rish, 1996). This process is repeated until the policy stabilises. The process of value determination in policy iteration is achieved by a system of linear equations (Pashenkova and Rish, 1996).

This works well in small state spaces, but in larger state spaces this system is not efficient. However arguments have been made that promote each approach as being better for large problems (Kaelbling and Littman, 1996). This is where other algorithms such as modified policy iteration (MPI) can be used to improve the process. Modified policy iteration was introduced by Puterman and Shin (1978). In modified policy iteration, value determination is similar to value iteration, with the difference being that utilities are determined for a fixed policy, not for all possible actions in each state (Pashenkova and Rish, 1996). The problem with this process is that the number of iterations of the value determination process is not determined. Pashenkova and Rish (1996) state that Puterman (1994) proposed the following options that could be used to solve this problem. Firstly, simply use a fixed number of iterations, secondly choose the number of iterations according to a predefined pattern and thirdly use the same process as value iteration.

COMPUTER GAMES & APPLICATIONS

There are many different types of commercial computer games available today; these include Real Time Strategy (RTS) games, sims games, God games and First person shooters (FPS) (Tozour, 2002). The AI in these and other type of games could possibly benefit from MDPs.

The most obvious computer game application for MDPs is a grid world navigation example, where the game world is split into a grid, which an NPC uses to navigate from one location to another. This example can be found in most literature on the subject including Russell and Norvig (1995) and Mitchell (1997). The task of moving NPCs in these types of game is in essence a sequential decision problem. This is exactly what the MDPs framework solves. This use of MDPs could be applied to RTS, FPS or 2D platform games. Other applications of MDPs include decision-making and planning. For this work we propose to apply MDPs to NPC movement in a 2D style game, such as Pac-man (Namco, 1980). We have chosen this type of game because it operates in real time and offers plenty of scope to explore the different features of MDPs.

DEVELOPMENT

In this section we present the development of the VI algorithm as an AI engine for use in real time 2D style computer games. The VI algorithm was implemented with a convergence threshold as the stopping criterion. However we also looked into creating our own stopping criterion, which was based around VI and designed for speed and use in real time computer games.

Value iteration using convergence as a stopping criterion is designed to find the optimal policy. However a less than optimal policy is acceptable in computer games if it speeds up processing time and still allows the NPC to reach its goal in an appropriate and acceptable manner. We have developed a new stopping criterion, which is as simple and quick as possible, but which still should achieve a workable policy. We call the new stopping criterion “Game Value Iteration” (GVI) and it works as follows: we simply wait for each state to have been affected by the home state at least once. This is achieved by checking if the number of states, with utilities that are equal to or less than 0 (zero) are the same after 2 successive iterations. All non-goal states have a reward (cost), which is slightly negative depending on their environment property (i.e. land, water etc.). Since utilities initially equal rewards, a state’s utility will be negative until it has been affected by the positive influence of the home state.

As a result the number of cells with negative utilities will decrease after each iteration. However some states may always retain a negative utility, because they have larger negative rewards due to their environment property and they may be surrounded by states with similar environment properties. Consequently when the number of states with negative utilities stays the same for 2 successive iterations we can say that the states are not optimal, but they should be good enough for a workable policy to exist, which the NPC can use to navigate the map. Before this point it is likely that no workable policy for the entire environment would exist. This stopping criterion assumes rewards can only be negative and there is a positive terminal state which is equal to 1. Also note that, checking if a state's utility is greater than 0 is not required for the terminal states, because their utilities never change. When each state has been affected by the home state at least once we can say that the states are not optimal, but they should be good enough for a workable policy to exist, which the NPC can use to navigate the map.

An AI engine program was developed in Microsoft Visual Basic in conjunction with the AI engine. This program contained the AI engine itself and an environment to test the engine. The environment consisted of a top down view, just like a 2D style game and was made up of a 10x10 grid of cells, each cell in the grid having different properties associated with it. For example a cell could have a land, wall or water property. Figure 2 shows an example of how the grid based environment would look. Figure 2 is based on an example of this type of environment found in Russell and Norvig (1995).

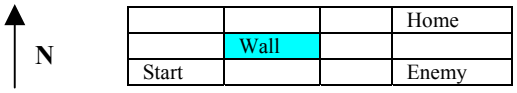


Figure 1: Example of the grid based environment.

The properties of an environment are used by the NPC (i.e. AI engine) to affect the reward value for each cell. For example water could mean slower movement for the NPC, so by giving cells with the water property an additional negative reward value (i.e. –0.02) it will mean that the reward for being in that cell is slightly less than cells with no water property. When the utility value of each cell is created the utility values of cells with the water property will be less than those with no water property. So when an NPC makes a choice of which cell to move to it will be less likely to move to the cell that has the water property.

The NPC will be able to move in one of four directions North, East, South, or West, which will supposedly move the NPC one cell in the intended direction, but only with a certain amount of probability (Pashenkova and Rish, 1996), such as 0.8. However this will depend on the obstacles in the grid such as a wall or the edge of the grid.

The NPC will begin in a start state, which can be any cell in the grid, except the enemy cell or home cell. The terminal states, where the simulation ends, are the home and the enemy states. In 2D style game the home state for the NPCs will be the human player. The home terminal state is the positive terminal state for the NPC and the enemy terminal state is the negative terminal state, which the NPC will avoid.

IMPLEMENTATION

This section covers how MDPs were implemented as an AI engine. As stated above the utility value of each cell in the grid (game environment) was determined by using the value iteration algorithm. We used two different stopping criteria: utility convergence and our new stopping criterion, called game value iteration, to ensure that each cell in the grid creates a usable policy for the NPC.

When the utility values for each cell are initialised they are initialised to the reward value of each cell. Each non-goal state always has a slightly negative reward on top of any cell property rewards. The cell(s) containing the enemy will have a reward value of –1 and the cell containing the home (or goal) will have a reward value of +1, regardless of the cell's other properties.

A schematic description of the GVI algorithm is given below. The value iteration algorithm is implemented exactly as it is in Russell and Norvig (1995). The GVI algorithm is based on this algorithm.

```
function GAME VALUE-ITERATION(M, R) returns a utility function
    inputs:          M, a transition model (or transition probabilities)
                     R, a reward function on states
    local variables: U, a utility function, initially identical to R
                     U1, a utility function, initially identical to R
                     AllStatesChanged, an all states changed flag initially equal to false
                     NumStatesBelowZero, stores the number of states below zero
                     LastNumStatesBelowZero, stores the last number of states below zero

    repeat
        for each state i do
            U1 [i] ← R[i] + max_a Σ_j M^a_ij U[j]
        end

        U ← U1
        LastNumStatesBelowZero = NumStatesBelowZero

        for each state i do
            if U(i) <= 0 then
                if U(i) != AnyTerminalStates then
                    NumStatesBelowZero = NumStatesBelowZero + 1
                end if
            end if
        end

        if LastNumStatesBelowZero = NumStatesBelowZero then
            AllStatesChanged = true
        end if

    until AllStatesChanged = true
return U
```

The step in the schematic description above, where the utility values are determined is the first *for* loop just after the *repeat* statement. The equation in that loop can also be seen below.

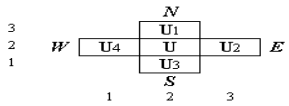
$$U1\ [i] \leftarrow R[i] + \max_a \sum_j M_{ij}^a \quad U[j]$$

Where *U1[i]* is the new utility value estimate for a cell in the grid and *R[i]* is the reward value. max_a is select the utility that returns the maximum value. *i* is the index of all cells in the grid and *j* is the index of the number of cells surrounding *i* (i.e. possible moves, north, south, east, west). *M* is the transition model (the probability of moving in a certain direction) and *U* is the current utilities.

Given the value iteration equation above, the utilities for each state can be determined, and given the fixed policy of maximising expected utilities, an NPC will be able to make a move in any state. No matter what the outcome of any action

is, the NPC will always know where to move next, by selecting the cell that has the highest expected utility. Next we are going to show an example of how the equation will work in practice. It demonstrates for one iteration how the utility value for one cell in the grid will be determined.

Key
U = Utility.
P = Probability.



PA = 0.8.
PB = 0.1.
PC = 0.1.
PD = 0.0.

To work out the utility of cell 2,2 the following will be conducted:

Action N = PA * U1 + PB * U2 + PC * U4 + PD * U3.

Action E = PA * U2 + PB * U3 + PC * U1 + PD * U4.

Action S = PA * U3 + PB * U4 + PC * U2 + PD * U1.

Action W = PA * U4 + PB * U1 + PC * U3 + PD * U2.

U = Reward + The action that returns the maximum value.

This process is repeated for every cell in the grid, except for the enemy's cell(s), the home cell and any wall cells. If the utility is being calculated for the cell next to a wall or a cell on the edge of the grid, there will be no possible move in those directions. If this occurs, then the utility value of the cell whose utility is being calculated will be used. One iteration is complete when every cell has been visited once. The process is repeated until the stopping criterion is met.

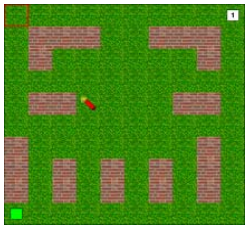
EXPERIMENTAL RESULTS

Many different experiments were conducted on the AI engine through the AI engine program. The results of these experiments were used to help implement a computer game and to validate our work. The parameters that were varied in the experiments included the configuration of the maps (i.e. locations of obstacles and goal states) and the reward values associated with cell properties.

However the results discussed here mainly look at determining the appropriate threshold value for VI, determining whether the GVI algorithm works in practice and comparing each algorithm's performance. In our experiments an NPC was setup to learn what action to take in each cell by using the VI algorithm. Tables 1 and 2 show some of the results of this work and screenshots of the test maps used to produce the results in those tables.

For all experiments the following things were kept the same: there were two goal states, +1 (home) and –1 (enemy), and there was a cost of -0.0000001 for all non-goal states. The probability of moving in the intended direction was 0.8 and the size of the game world was 10x10.

The HD column in tables 1 and 2 stands for hamming distance between the generated policy and the optimal policy. The optimal policy is the policy obtained by running the algorithm with the same initial data and maximum precision (Pashenkova and Rish, 1996). The use of hamming to determine the difference between a policy and an optimal policy is based on that used in Pashenkova and Rish (1996).



Convergence Threshold	No. of Iterations to Convergence	Agent Successfully Navigated Map	No. of steps Agent took To Navigate Map	HD
1.00	1	No	N/A	70
0.5	4	No	N/A	61
0.25	8	No	N/A	39
0.125	12	No	N/A	11
0.0625	15	No	N/A	2
0.031250	19	Yes	18	0
0.015625	21	Yes	18	0
0.007812	23	Yes	18	0
0.003906	24	Yes	18	0
0.001953	25	Yes	18	0
0.000977	26	Yes	18	0
0.000488	27	Yes	18	0
0.000244	28	Yes	18	0
0.000122	29	Yes	18	0
0.000061	30	Yes	18	0
0.000031	30	Yes	18	0
0.000015	31	Yes	18	0
0.000008	32	Yes	18	0
0.000000	59	Yes	18	-
GVI	**18**	**Yes**	**18**	**1**

Table 1: The environment map and the results produced from experiments conducted on the map.

Convergence Threshold	No. of Iterations to Convergence	Agent Successfully Navigated Map	No. of steps Agent took To Navigate Map	HD
1.00	1	No	N/A	27
0.5	4	No	N/A	27
0.25	7	No	N/A	27
0.125	10	No	N/A	26
0.0625	14	No	N/A	22
0.031250	19	No	N/A	15
0.015625	20	No	N/A	12
0.007812	22	No	N/A	14
0.003906	27	Yes	32	9
0.001953	34	Yes	36	3
0.000977	40	Yes	36	0
0.000488	46	Yes	36	0
0.000244	50	Yes	36	0
0.000122	53	Yes	36	0
0.000061	56	Yes	36	0
0.000031	60	Yes	36	0
0.000015	63	Yes	36	0
0.000008	66	Yes	36	0
0.000000	210	Yes	36	-
GVI	**32**	**Yes**	**36**	**4**

Table 2: The environment map and the results produced from experiments conducted on the map.

The maps used for the experiments above attempt to represent a maze like world that you would expect to see in 2D style games. However we also experimented with simpler and more complex maps. Tables 1 and 2 show that the largest threshold, which produces an optimal policy, is 0.031250 (Table 1). However this threshold does not produce an optimal policy in Table 2. This shows that the utility thresholds, which produce an optimal policy, vary from map to map. In general we observed that as map complexity increased, they required more iterations and smaller thresholds to achieve workable and optimal policies. This could cause problems in computer games because maps are constantly changing and vary from level to level. As a result it's reasonable to say that a conservative threshold would have to be used to ensure that a map always converged to an optimal or near optimal policy. Tables 1 and 2 also show that the utility values for the VI algorithm converge after the policy has converged. This result is consistent with previous work in the area, such as Pashenkova and Rish (1996) and Russell and Norvig, (1995) and is a recognised issue with this algorithm.

From tables 1 and 2 we can see that the GVI algorithm produces a workable policy that is less than optimal, but

converges at a low number of iterations. This means the algorithm should on average be quicker to run than VI because larger numbers of iterations require more processing time. Also a benefit of this algorithm is that it automatically adapts to the complexity of the game world, so it should always produce the best policy it can, without running any unnecessary iterations. The algorithm should always produce a workable policy, but it will not necessarily be the optimal policy. In our experiments above this seems to matter very little, because the hamming distance is very small, but on a large map (E.g. 20x20 or 40x40) this difference might become significant.

We also conducted experiments on the reward values of cells to see how they affect an agent's movement. These experiments showed that the affect a negative reward would have on an NPC depended on how optimal the policy was. If a zero threshold was used with VI, a small negative value (i.e. -0.02) for the cell property water would be enough to affect the NPCs behaviour so it would be likely to avoid water until it was necessary so go through it. However for less optimal policies this value would need to be slightly bigger to have a similar affect (i.e. -0.06). This affect is just like the one discussed in the paragraph above. Because the policy is not optimal the water (or enemy's) effect on the game environment is lessened. Also it is worth noting that if the negative rewards are increased by too much this can also cause problems, because they can have too great an affect on the cell's utility which can prevent the GVI algorithm from converging to a workable set of utilities.

DISCUSSION

The experiments conducted on the AI engine program have shown that MDPs using both VI and the newly introduced GVI algorithms can be used to create intelligent NPC behaviour. The movement produced by the AI engine appeared to the authors to be less scripted and deterministic than that in researched 2D style computer games. The AI engine also offers interesting environment features through creative use of reward values. This could make the MDPs AI engine interesting to computer game players and the computer game industry, because it offers a different approach to solving the problem of AI in 2D style games.

The MDP AI engine with VI and GVI as an AI tool for NPC navigation offers game developers a different approach to applying AI to 2D style games. However from the results of this work and our observations we can see that there are limitations with this technique that need to be researched further. Firstly, even though the VI algorithm works in our AI engine (which has just 1 NPC), it is very processor intensive. The GVI algorithm does overcome this problem; however this algorithm would need to be tested further to prove its usefulness. Secondly, the experiments conducted here were only on 10x10 grids. This size grid is quite small for a game environment, so experiments would need to be conducted on larger grids to determine if the VI and GVI algorithms can execute quickly enough and the less than optimal policy for GVI is still viable. The hamming distance between the GVI algorithm policies and the optimal polices was quite small in our experiments; however it could be a lot larger in bigger game environments.

CONCLUSIONS AND FUTURE WORK

This paper has shown that Markov decision processes using Value iteration and the newly introduced GVI algorithm can be successfully applied to an AI computer game engine. The development of the AI engine and the experiments conducted on the AI engine allowed a greater understanding of this approach and the problems involved, in relation to computer games.

There is plenty of scope for further work in this area. Firstly we intend to apply the AI engine to a 2D real time computer game to determine if the technique can operate successfully in this domain. Secondly we plan to extend the size of the game environments and confirm that the use of a less than optimal policy still produces a viable solution in larger environments.

REFERENCES

Bellman R. (1957) *Dynamic Programming* Princeton University Press, Princeton, New Jersey.

Bonet B. (2002) An e-Optimal Grid-Based Algorithm for Partially Observable Markov Decision Processes. *in* Proc. 19th Int. Conf. On Machine Learning. Sydney, Australia, 2002. Morgan Kaufmann. Pages 51-58.

Howard R. (1960). *Dynamic Programming and Markov Processes*. Cambridge, MA: The MIT Press.

Kaelbling L. and Littman, M. (1996) Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, vol. 4, pp. 237-285.

Kellis E. (2002). An Evaluation of the Scientific Potential of Evolutionary Artificial Life God-Games: Considering an Example Model for Experiments and Justification. MSc. Thesis, University of Sussex.

Kristensen A. (1996), Textbook notes of herd management: Dynamic programming and Markov decision processes <http://www.prodstyr.ihh.kvl.dk/pdf/notat49.pdf> (accessed 24 April 2003).

Lionhead Studios / Electronic Arts (2001) Black & White. <http://www.eagames.com/>.

Mitchell T. (1997). *Machine Learning*, McGraw Hill: New York.

Namco (1980), Pac-man. <http://www.namco.co.uk/>.

Pashenkova E. and Rish I. (1996) Value iteration and Policy iteration algorithms for Markov decision problem. <http://citeseer.nj.nec.com/cache/papers/cs/12181/ftp:zSzzSzftp.ics.uci.eduzSzpubzSzCSP-repositoryzSzpaperszSzmdp_report.pdf/value-iteration-and-policy.pdf> (accessed 23 April 2003).

Puterman M. (1994) Markov decision processes: discrete stochastic dynamic programming. New York: John Wiley & Sons.

Puterman M. and Shin M. (1978) Modified policy iteration algorithms for discounted Markov decision processes. Management Science, 24:1127-1137.

Russell S. and Norvig P. (1995). *Artificial Intelligence A modern Approach*, Prentice-Hall: New York.

Spronck P., Sprinkhuizen-Kuyper I. and Postma E. (2002). Evolving Improved Opponent Intelligence. *GAME-ON 2002 3rd International Conference on Intelligent Games and Simulation* (eds. Quasim Medhi, Norman Gough and Marc Cavazza), pp. 94-98.

Spronck P., Sprinkhuizen-Kuyper I. and Postma E. (2003). Online Adaptation of Game Opponent AI in Simulation and in Practice. *GAME-ON 2003 4th International Conference on Intelligent Games and Simulation* (eds. Quasim Medhi, Norman Gough and Stephane Natkin), pp. 93-100.

Sutton R. and Baro A. (2000) *Reinforcement Learning An Introduction*. London: The MIT Press.

Tozour P. (2002) The Evolution of Game AI in Steve Rabin (ed) *AI Game Programming Wisdom*, Charles River Media, pp. 3-15.

Yousof S. (2002) MDP Presentation CS594 Automated Optimal Decision Making, <http://www.cs.uic.edu/~piotr/cs594/ Sohail.ppt> (accessed 27 April 2003).