Reducing state space exploration in reinforcement learning problems by rapid identification of initial solutions and progressive improvement of them

Kary FRÄMLING
Department of Computer Science
Helsinki University of Technology
P.O. Box 5400, FIN-02015 HUT
FINLAND
Kary.Framling@hut.fi http://www.cs.hut.fi/~framling

Abstract: Most existing reinforcement learning methods require exhaustive state space exploration before converging towards a problem solution. Various generalization techniques have been used to reduce the need for exhaustive exploration, but for problems like maze route finding these techniques are not easily applicable. This paper presents an approach that makes it possible to reduce the need for state space exploration by rapidly identifying a "usable" solution. Concepts of short- and long term working memory then make it possible to continue exploring and find better or optimal solutions.

Key-Words: Reinforcement learning, Trajectory sampling, Temporal difference, Working memory, Maze route finding

1 Introduction

The work presented in this paper started from the idea to develop an artificial neural net (ANN) model that would do problem solving and learning in similar ways as humans and animals do. The model would also correspond to some very rough-level ideas and knowledge about how the brain operates, i.e. activations and connections between different areas of the brain and notions of short- and long term working memory.

Animal problem solving mainly seems to be based on *trial and learning*. The success or failure of a trial modifies behavior in the "right" direction after some number of trials, where "some number" is in the range one (e.g. learning how to turn on the radio from "power" button) to infinity (e.g. learning how to grab things, which is a life-long adaptation procedure).

Such behavior is currently studied mainly in the scientific research area called *reinforcement learning* (RL). RL methods have been successfully applied to many problems where more "conventional" methods are difficult to use due to factors like lacking data about the environment, which forces the neural net to *explore* its environment and learn interactively. Exploring is a procedure where the *agent* (see for instance [2] for a discussion on the meaning of the term “agent”) has to take actions without a priori knowledge about how good or bad the action is, which may be known only much later when the goal is reached or when the task failed.

The RL problem used in this paper, maze route finding, is commonly used in psychological studies of animal learning and behavior [3]. Animals have to explore the maze and construct an internal model of the maze in order to reach the goal. The more maze runs the animal performs, the quicker it goes to the goal since solutions get better memorized.

Maze route finding is not a very complicated problem to solve with many existing methods as pointed out in section 2 of this paper, where the problem setup is explained. Therefore, the ANN solution presented in section 3 is not unique by being the first one able to solve the problem. It does, however, solve the problem in a new way, which needs significantly less initial exploration than existing methods. Initial exploration runs are mainly shortened by the SLAP (Set Lowest Action Priority) reinforcement presented. Identified solutions may be further reinforced by temporal difference (TD) methods [5].

Even though TD learning can be used as an element of the methods described in the paper, the methods presented here do not use classical notions of *value functions* for indicating how good a *state* or an *action* are for reaching the goal. Instead notions of *short term working memory* and *long term working memory* are used for selecting appropriate actions at each state. Short term working memory exists only during the problem solving, while previous problem-solving instances are stored in long term working memory. This memory organization gives new possibilities to balance between exploration of a new environment and/or new solutions on one hand, and exploitation of existing knowledge on the other hand.

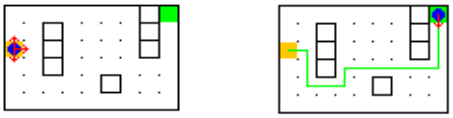
2 Problem Formulation

Sutton and Barto use a maze like the one in Fig 1 in chapter 9 of their 1998 book [7]. The discussion that follows on the advantages and disadvantages of existing RL methods is principally based on this book concerning symbols, equations and method descriptions.

An agent is positioned at the starting point inside the maze and has to find a route to the goal point. Each maze position corresponds to a state, where the agent selects one

action of four, i.e. going north, south, east or west, unless some of these are not possible. Each state is uniquely identified in a table-lookup manner.

Initially, the agent has no prior knowledge about the maze, so it has no idea of what action should be taken at different states. Therefore it chooses an action randomly the first time it comes to a previously unvisited state, without knowing if it is a good one or not. If it was a bad one, the agent ends up in a dead end and has to walk back and try another direction. Coming back to a state already visited is also a bad sign since the agent is walking around in circles.



a b

Fig 1. Grid problem. a) Agent is shown in start position and goal position in upper right corner. b) One of the optimal solutions.

2.1 Symbolic methods

This maze route finding problem is easy to solve with a classical depth-first search through an inference tree representing all possible solutions [1]. The root of the tree is the starting state. The root has links to all *next states* that can be reached from it, which again have links to all their next states. The inference tree can be recursively constructed where leafs of the three are indicated by one of three cases:

1. Goal reached.
2. Dead-end, i.e. state with no next-states.
3. Circuit detected, i.e. coming back to a previously encountered state.

Depth-first search can explore the tree until a solution is found, which can be memorized. If the goal is to find the optimal path, breadth-first search [1] or complete exploration of the whole tree can be used.

Depth-first and breadth-first search become unfeasible when the search tree grows bigger due to a great number of states or a great number of links (actions). Heuristic approaches are often used to overcome these problems. They make it possible to concentrate only on "interesting" parts of the search tree by associating numerical values with each tree node or each link in the search tree, which indicate the "goodness" of that node or that link. Heuristic values can be given directly, calculated or obtained by learning. Reinforcement learning is one way of learning these values.

2.2 Reinforcement learning principles

In RL, heuristic estimates correspond to the notion of value functions, which are either *state-values* (i.e. value of a state in the search tree) or *action-values* (i.e. value of a link/action in the search tree). In the maze problem, value functions should be adjusted so that "good" actions, i.e. those leading to the goal as quickly as possible are selected.

One possible RL approach to the maze problem using state values would be to randomly select actions until the goal is reached, which forms one *episode*. During the episode, a reward of –1 is given for all state transitions except the one leading to the goal state. Then the value of a state $s \in S$ (set of possible states) for a given episode can be defined formally as

$$V^{\pi}(s) = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^{k} r_{t+k+1} \middle| s_t = s\right\}, \qquad (1)$$

where $V^{\pi}(s)$ is the state value that corresponds to the expected return when starting in s and following *policy* π thereafter [7]. A policy is the "rule" being used for selecting actions, which can be random selection as assumed here or some other rule. So, for Markov Decision Processes (MDP), $E_{\pi}\{\}$ denotes the expected value given that the agent follows policy π. The value of the terminal state, if any, is always zero. γ^{k} is a discounting factor that is less than or equal to one and determines to what degree future rewards affect the value of state s. When the number of episodes using random policy approaches infinity, the average state value over all episodes converges to the actual state-value for policy π.

Once state values have converged to correct values, states which are "closer" to the goal will have higher state values than states that are further away. If the policy is then changed to *greedy exploration*, i.e. always taking the action that leads to the next state with the highest state value, then the agent will automatically follow the optimal path. Unfortunately, random initial exploration is too time consuming to be useful in practical problems. The usual way to treat this case is to use *ε-greedy exploration*, where actions are selected greedily with probability (1 - ε), while random action selection is used with probability ε. Another version of ε-greedy exploration called *softmax* is sometimes used. Softmax selects actions leading to high state values with a higher probability than actions leading to low state values, instead of using random action selection.

When ε-greedy exploration and –1 reward on every state transition is used for the grid world of Fig 1, all state values can be initialized to 0 or small random values. During exploration, states that have not been visited or that have been visited less than others will have higher state values than more frequently visited ones. Therefore ε-greedy exploration will by definition tend to exhaustively explore the whole state space, so initial episodes are very long. Convergence towards correct state values also requires a great number of episodes, so this approach is not usable for bigger problems.

2.3 Monte-Carlo methods

Another possibility is to only give positive reward at the end of an episode and zero reward for all intermediate transitions. Monte-Carlo Policy Evaluation [7] is one possibility for propagating the reward backwards through the state history of one episode. If a reward of +1 is given for reaching the goal, +1 is added to the "return values" of all states appearing in the episode. The state-value of a state is then the average return value over all episodes. Using ε-greedy exploration, state values eventually converge to the optimal policy, even though guaranteed convergence has not yet been formally proved according to [7].

For the maze problem used in this paper, generating episodes using a random policy requires an average of 1700 steps. Even with TD methods studied in the next section, nearly 30 episodes is required before convergence towards a solution occurs, so for Monte-Carlo simulation the number of episodes needed is probably over 100. This would mean over 170 000 steps, which is very slow compared to all other methods treated later in this paper.

2.4 Temporal difference learning and TD(λ)

Monte-Carlo policy evaluation requires successfully completed episodes in order to learn. Therefore it quickly becomes too slow in order to be usable for most applications since it might require a very big number of episodes before starting to select better actions than using a random policy. Solving this problem is one of the main issues in so called *bootstrapping* methods, like those based on *temporal-difference* (TD) learning [5]. Bootstrapping signifies that state- or action value updates occur at every state transition based on actual reward, but also on the difference between the current state value and the state value of the next state according to:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \quad (2)$$

, which is known as the *TD(0)* method. The more advanced *TD(λ)* algorithm, of which TD(0) is an instance, is currently the most used bootstrapping method. TD(λ) uses a notion of *eligibility trace*, which λ refers to. An eligibility trace signifies using the state/action history of each episode for propagating rewards backwards, just like in Monte-Carlo methods. Associating an eligibility trace value with each state, which is usually increased by one (*accumulating eligibility trace*) every time the state is encountered during an episode, creates the trace. λ is a *trace decay* parameter, which together with γ determines how fast the eligibility trace disappears for each state. For an accumulating eligibility trace, a state's eligibility trace value $e_t(s)$ at time t is calculated by:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases} \quad (3)$$

Experience has shown that TD methods generally converge much faster to the optimal solution than do Monte-Carlo methods [7]. Using a model of the environment that is constructed during exploration, can further accelerate convergence as for Dyna agents [6]. In Dyna agents, the model memorizes which states are reached by what actions for each state/action pair encountered, so TD learning can be used for updating value functions both during interaction with the environment and without interaction with the environment.

For the maze in Fig 1, Sutton and Barto have compared convergence times between direct reinforcement learning and Dyna agent learning [7]. Since both of these use random exploration on the first run, the first episode lasted for about 1700 steps. Direct RL needed about 30 episodes before converging to the optimal path of 14 steps, while the best Dyna agent found it after about five episodes. However, both methods stay in eternal oscillation between 14 steps and 16 steps due to ε-greedy exploration that regularly puts the agent off the optimal path.

The main shortage of these techniques is that they have a very long first exploration run, during which they go through most states numerous times (54 states and 1700 steps => ~32 visits per state). For a simple maze like the one in Fig 1 this is not a big problem, but the length of the initial exploration run can be expected to grow exponentially as the number of states increases. These long exploration runs are due to the need of current methods to first explore the whole state space in order to converge towards an optimal solution.

Exploration of the entire state space is impossible to use in most practical applications reported like backgammon [8], which has approximately 10^{20} states. However, many states correspond to similar game situations, for which similar moves are appropriate. Therefore learning results for one state can be applied to numerous other states too if there is a way to identify similar states based on state descriptions instead of treating each state as a separate case.

Many artificial neural networks are capable of such generalization, where actions learned for one state description are automatically applied to similar states even though these states would never have been encountered before. Also, in a game like backgammon, most states have a very small or zero probability of occurring in a real game, so they do not need to be learned.

However, in a maze problem this approach does not seem to be applicable since there are no general rules that could be learned based on a general description of possible states. There are 16 different states depending on possible directions, but there is no generally applicable rule for what action is appropriate for each type of state, so the problem of excessively long initial exploration remains. The solution proposed in this paper rapidly finds and memorizes at least one usable solution using minimal exploration efforts and then explores towards the optimal solution.

3 Problem Solution

One of the initial ideas of the work presented here was to maintain a link with animal and human problem solving and the brain. This is why the reinforcement learning methods presented here use an artificial neural net (ANN) model even though they could probably also be implemented in other ways. In this "brain inspired" ANN, neurons are either *stimulus* or *action neurons*, which seems more appropriate than speaking about inputs and outputs of the neural net. In the maze solving problem, each state corresponds to one stimulus neuron and each possible action to one action neuron.

When the ANN agent enters a completely unknown maze, it only has four action neurons which correspond to the four possible actions, but it has no stimulus neurons. Stimulus neurons are created and connected to action neurons for every state encountered for the first time during an episode. When a new stimulus neuron is created, the weights of its connections to action neurons are initialized to small random values for instance in the interval]0,1]. Since these stimuli and their connection weights are created during one episode and exist only until the episode is finished, they are here called *short term working memory*. Once an episode is finished, both short term stimuli and connection weights can be copied as instances in *long term working memory*.

3.1 ANN architecture

The purpose of long term working memory is to be able to solve the same problem more efficiently in the future. When a stimulus is activated in short term working memory, we can suppose that the corresponding stimulus instances in long term working memory are also activated to a certain degree. Since long term working memory instances are connected to action neurons, they affect what action is selected. Actions are selected according to the *winner-takes-all* principle, where the action neuron with the biggest activation value wins. Activation values of action neurons are calculated according to:

$$a_n = \sum_{i=1}^{nstim} stw_{i,n} * s_i + \sum_{j=1}^{nltm} \sum_{i=1}^{nstim} \alpha * ltw_{j,i,n} * s_i \qquad (4)$$

where a_n is the activation value of action neuron n, $stw_{i,n}$ is the connection weight from stimulus neuron i to action neuron n, s_i is the current activation value of stimulus neuron i, $ltw_{j,i,n}$ is the connection weight for long term working memory instance j and stimulus i to action neuron n, *nstim* is the number of stimulus neurons and *nltm* is the number of instances in long term memory. α is a weighting parameter that adjusts to what degree stimulus activations in short term working memory cause activation of corresponding stimuli in long term working memory. α can also be considered as a parameter that adjusts the influence of past experiences on action selection. Since short term working memory connection weights are always initialized to random values when a state is encountered for the first time during an episode, adjusting the α parameter offers an alternative to ε-greedy exploration and softmax for balancing between exploration and exploitation.

Equation (4) can be rewritten in the form

$$a_n = \sum_{i=1}^{nstim} stw_{i,n} s_i + \alpha \sum_{i=1}^{nstim} s_i \sum_{j=1}^{nltm} ltw_{j,i,n} \qquad (5)$$

, which shows that long term working memory can be implemented as a vector of sums of stored connection weights, which makes it possible to implement the proposed model in a computation- and memory efficient way. Only two connection weight matrices are needed, one for short term working memory weights and the other one for long term working memory weights. The short term working memory matrix is of size (number of actions)*(number of states encountered during current episode). The long term working memory matrix is of size (number of actions)*(number of states ever encountered). Straight matrix multiplication and addition is enough to perform the needed calculations.

3.2 Search for "usable" initial solution

Exploration and exploitation happen simultaneously, which one is predominant depends on the value of α and on the number of instances in long term working memory. Long term working memory is initially empty for a completely unexplored maze. Therefore action selection according to equation (4) is random the first time a new state is encountered because the new stimulus neuron created in short term working memory has random initial connection weights.

If a state already encountered during the same episode is visited again, it is either due to coming back from a dead end or going around in circles. In both cases it would be unwise to take the same action as the previous time in the same state. In order to know what action was taken the previous time, it is sufficient to evaluate equation (4) for the current state and see which action wins. The winning action is punished according to the new *Set Lowest Action Priority* principle, shortly SLAP. The "slapped" action is punished by decreasing its weights enough to make it become the least activated among possible actions the next time we are in the same state. This is done according to the formula:

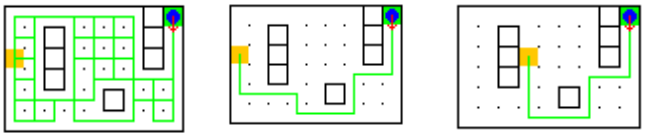
$$\Delta stw_{i,n} = -(a_n - a_{\min}) stw_{i,n} s_i / a_n \qquad (6)$$

where a_n is the activation value of the slapped neuron and a_{min} is the new activation desired, obtained by taking the lowest action activation among possible actions (possible directions) and subtracting a small ratio of it.

Slapping is not only used for punishing actions that lead to dead ends and circuits, slapping is also applied to the direction the agent comes from directly after entering a

state. Otherwise the probability that the agent would go back in the same direction as it came from would be as high as taking a new direction. The goal of SLAP is therefore mainly to make the exploration go to the goal as quickly as possible with minimal exploration effort. Sutton and Barto [7] call this principle *trajectory sampling* and show for a simple problem that this technique greatly reduces computation time compared to exhaustive search, especially for problems with a great number of states. For the sample run in Fig 2, the first episode took only 104 steps and still directly gives the rather good solution of 16 steps on the second episode (14 is the optimal solution). This result can be considered excellent compared to the 1700 steps reported for TD and Dyna-Q in [7] for the first episode, not to mention that they need up to 30 episodes before reaching a 16-step solution.

The last actions used for each state are implicitly stored in short term working memory weights by SLAP reinforcements. Therefore an exploitation run that uses these weights will directly follow the shortest path discovered as in Fig 2b. This is also true if the agent starts from some other state encountered during exploration than the initial starting state as in Fig 2c.



a b c

Fig 2. a) First episode, 104 steps, b) second episode, 16 steps, c) different starting point than initial one, 13 steps.

Once an episode is finished, short term working memory weights can be copied as an instance in long term working memory. This can either be done directly or after an additional reinforcement has been applied. This is implemented by doing a "replay" of all stimuli activations (states) and rewarding the winning actions by increasing the value of the connection weight between the stimulus and the winning action by a *final reward* value. Dispatching the final reward in this way is actually very similar to TD(λ) with $\gamma = 1$. The main difference is that the eligibility trace is not stored anywhere, it is reconstructed instead. Even though no formal proof is shown here for the similarity with TD(λ), it can still be assumed that TD(λ) methods could be used for propagating the final reward backwards as well. Therefore most existing experience and knowledge about TD methods could be applicable concerning convergence, calculation complexity etc.

3.2 Search for optimal solution

Since only a part of the state space is usually visited during the initial exploration and only a part of the possible actions in different states are used, the initially identified solution has a high probability of being sub optimal. This is also the case in Fig 2, where the optimal solution would be 14 steps. However, the optimal solution is very difficult to find since it has a much smaller probability of occurring during random exploration than other solutions. At least two possibilities exist in order to find the optimal solution:

1. Letting several neural net agents search for a solution and see which agent found the best one.
2. Using low α and/or low final reward at the end of episodes and letting the same agent do a great number of exploration runs. This could also be combined with ε-greedy and softmax exploration.

The first possibility might seem to be rather wasteful, but since initial exploration only requires an average of 115 steps, there can still be 15 agents exploring before reaching the 1700 steps used by the initial run with TD(λ) in [7]. Classical TD(λ) (without model as in Dyna-Q) apparently needs far over 10000 exploration steps before finding a path requiring only 16 steps, but then quite rapidly finds the optimal path with 14 steps. The experimental probability of an agent using random policy to find the optimal path is 0.007, which means that it takes about 143 agents on the average before the optimal path is found. Therefore an average of 16445 (143*115) steps are needed for SLAP agents to find the optimal solution. This number seems to be approximately the same as for TD(λ), but the huge advantage of SLAP agents over TD(λ) is that they find a rather good solution already after one episode and about 115 exploration steps. Such a solution can be directly usable, so further exploration can be deferred to when there is spare time to do it. This also corresponds rather well to human behavior – first find a "usable" solution and be curious about other solutions when there is time for it.

Table 1 shows the number of steps needed for the first and the second episodes of ten sample SLAP agents. After each episode, the agents received a final reward of one at the end of the episode before storing the solution as an instance in long term working memory. All agents had $\alpha = 1$. For 30 sample agents, the longest initial episode took 336 steps and the shortest took 26 steps. The total number of initial episode steps for the 30 agents was 3460 and the optimal solution of 14 steps was found by one of these. Even the worst second episode solution requiring 26 steps could be usable in many applications.

Table 1. Exploration steps for first episode versus second episode for ten different agents.

Run #	1	2	3	4	5	6	7	8	9	10
Episode 1	44	174	66	148	26	110	136	218	40	168
Episode 2	22	18	14	24	22	18	18	16	20	16

The second possibility to find the optimal path is to use the same agent all the time and let it gradually improve.

This possibility has so far only been studied for the case of using low α values and a final reward of one. However, only episodes that are shorter than any previous episode are stored as instances in long term working memory, which means that episodes tend to get shorter as the number of episodes increases. When using $\alpha = 0.01$, the path followed became stable after an average of about 500 episodes and a total of about 15000 steps. All agents that discovered the 14 step solution at least once (about one agent out of five) eventually converged to that solution, while the others converged to a solution of 16 steps. Convergence could certainly be made much quicker in several ways. One way would be to use adaptive final reward values, where a reward counter would count the total amount of final rewards given and then give a bigger final reward than this amount for better solutions, thus slightly overriding all previous solutions. Unfortunately, despite its simplicity, this method has not been tested yet.

Adjusting the values of α, the final episode reward and the interval for random initial weights of new stimuli in short term working memory determine the balance between random exploration and greedy exploration. But if the solutions found during the first episodes are too far from the optimal solution, these parameters are not sufficient for converging to the optimal solution. Using ε-greedy exploration should solve this problem since it would introduce stochastic behavior. Testing this is one of the first issues of future research. Future research will also focus on comparing existing RL methods and those proposed in this paper for other mazes and for other kinds of problems. It would be especially interesting to extend the approach to problems requiring generalization for different states based on state descriptions. One such problem is the minefield navigation problem treated in [4], which is more general than well-known cases like backgammon [8] that require a great amount of domain knowledge. In the minefield navigation problem there are no states, only continuous-valued state descriptions, where the number of stimuli is constant while the degree of activation of stimuli changes. All calculations used in this paper are applicable to this kind of stimuli, but they will certainly need to be further developed in order to solve this kind of problem.

4 Conclusion

This paper presents how initial exploration runs in reinforcement learning can be significantly shortened. This is achieved by the SLAP reinforcement learning principle, which makes the agent avoid coming back to states already visited. SLAP also has the side effect of memorizing the shortest path found during an episode in the weights of the neural net model presented here, thus finding "usable" solutions with minimal exploration. Since "usable" solutions are found very quickly, it becomes feasible to let multiple agents do simultaneous exploration and retain the best ones. Letting these agents communicate and exchange their information would be an interesting topic for future research since that could further reduce exploration time.

Notions of short- and long term memory presented offer agents a possibility to maintain a balance between previously found solutions and searching for even better solutions. This gives agents a much more "human like" behavior than do existing RL methods, i.e. first finding a usable solution and then being curious enough to improve the solution when there is time for it. Most current RL methods first exhaustively explore the whole state space several times and then converge towards an optimal solution, which is definitely not how a human individual finds a new way to navigate through a town, for instance.

Methods presented here are still at an early stage of research, so a lot of work remains before their position in the research area of reinforcement learning can be established. The results presented in this paper should still give a clear indication that the methods developed give several big advantages compared to existing methods. If similar results are obtained for other problems and problem domains, reinforcement learning could probably be used in many new application areas where they are not yet feasible due to excessive exploration times.

References:

[1] Genesereth, M.R., Nilsson, N.J., *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann Publishers, 1987.

[2] Jennings, N.R., Sycara, K., Woolridge, M., A Roadmap of Agent Research and Development, *Autonomous Agents and Multi-Agent Systems*, Vol. 1, No. 1, 1998, pp. 3-38.

[3] Louie, K., Wilson, M.A., Temporally Structured Replay of Awake Hippocampal Ensemble Activity during Rapid Eye Movement Sleep, *Neuron*, Vol. 29, No. 1, 2001, pp. 145-156.

[4] Sun, R., Merrill, E., Peterson, T., From Implicit Skills to Explicit Knowledge: A Bottom-Up Model of Skill Learning, *Cognitive Science*, Vol. 25, No. 2, 2001.

[5] Sutton, R.S., Learning to predict by the method of temporal differences, *Machine Learning*, Vol. 3, 1988, pp. 9-44.

[6] Sutton, R.S., Integrated architectures for learning, planning, and reacting based on approximating dynamic programming, in *Proceedings of the Seventh International Conference on Machine Learning*, Morgan Kaufmann Publishers, 1990.

[7] Sutton, R.S., Barto, A.G., *Reinforcement Learning*, A Bradford Book, MIT Press, Cambridge, MA, 1998.

[8] Tesauro, G.J., Temporal difference learning and TD-Gammon, *Communications of the ACM*, Vol. 38, 1995, pp. 58-68.