Int. J. Appl. Math. Comput. Sci., 2008, Vol. 18, No. 3, 409–421
DOI: 10.2478/v10006-008-0037-4

EMOTION LEARNING: SOLVING A SHORTEST PATH PROBLEM IN AN ARBITRARY DETERMINISTIC ENVIRONMENT IN LINEAR TIME WITH AN EMOTIONAL AGENT

SILVANA PETRUSEVA

Mathematics Department
Faculty of Civil Engineering, St. Cyril and Methodius University,
Partizanski odredi 24, P.O. Box 560, 1000 Skopje, Macedonia
e-mail: silvanap@unet.com.mk

The paper presents an algorithm which solves the shortest path problem in an arbitrary deterministic environment with n states with an emotional agent in linear time. The algorithm originates from an algorithm which in exponential time solves the same problem, and the agent architecture used for solving the problem is an NN-CAA architecture (neural network crossbar adaptive array). By implementing emotion learning, the linear time algorithm is obtained and the agent architecture is modified. The complexity of the algorithm without operations for initiation in general does not depend on the number of states n, but only on the length of the shortest path. Depending on the position of the goal state, the complexity can be at most $O(n)$. It can be concluded that the choice of the function which evaluates the emotional state of the agent plays a decisive role in solving the problem efficiently. That function should give as detailed information as possible about the consequences of the agent's actions, starting even from the initial state. In this way the function implements properties of human emotions.

Keywords: emotional agent, complexity, consequence programming, CAA neural network, planning.

1. Introduction

This paper considers the complexity issues of the problem of finding the shortest path in an arbitrary deterministic environment with an emotional agent. The searching method is on-line, combined with planning. Off-line search methods which first derive a plan that is then executed cannot be used to solve the path planning tasks, since the topology of the state space is initially unknown to the agent and can only be discovered by exploring, i.e., executing actions and then observing their effects. On-line search methods, also called real-time search methods, update modifiable parameters after each visit of a state and can discover certain environments goal state in polynomial time (Koenig and Simmons, 1992), while off-line methods sometimes require exponential time (Whitehead, 1991).

Whitehead (1991) analysed the search time complexity of unbiased Q-learning for problem solving tasks on a restricted class of state spaces. The results indicate that unbiased search can be expected to require time moderately exponential in the size of the state space. He proposed a learning algorithm to reduce the search. The algorithm, called *Learning with an External Critic* (LEC), is based on the idea of a mentor who watches the learner and generates immediate rewards in response to its most recent actions. This reward is then used temporarily to bias the learner's control strategy. The LEC algorithm requires time at most linear in the size of the state space and under appropriate conditions is independent of the state space size, requiring time proportional to the length of the optimal solution path. Here the research is moving toward supervised learning.

With the emotion learning algorithm proposed in this paper the agent solves the problem autonomously, using the concept of state evaluation, connecting it to the concept of feeling.

2. Emotion learning: Consequence driven systems approach

Emotion has been identified as one of the key elements of intelligence and of the adaptive nature of human beings (Damasio). The emotion based agent learning tries to

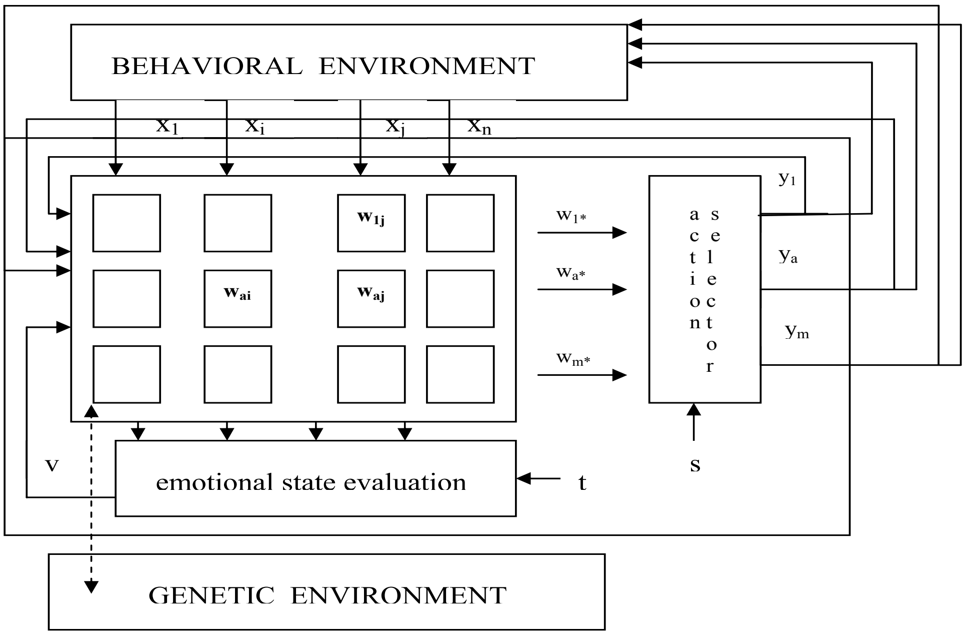

Fig. 1. CAA architecture.

develop artificial mechanisms that can play the role that the emotions play in human life. These mechanisms are called “artificial emotions”.

In the last 20 years there have been several pointers toward the issue of feelings and emotions as features needed for developing an artificial intelligent creature. Several issues in emotional agent design initiated the problem of emotion based learning in autonomous agents (Botelho and Coelho, 1998; Bozinovski, 1999a; Bozinovski 2002; Bozinovski, 2003). The problem is how to define emotion based learning and how to implement it in a useful manner. A computational model of an emotion that can be used in the learning system of an agent is presented in this paper.

This paper argues that emotion learning is a valid approach to improve the behaviour of artificial autonomous agents.

L.M. Botelho and H. Coelho (1998) argue that adaptive behaviour may be achieved through a process of emotion learning. Emotion learning refers to the improvement of the agent’s emotion process itself. Since emotion is one of the agent’s control processes, improved emotion entails improved behaviour (Botelho and Coelho, 1998).

We shall now consider how the *consequence driven systems* approach is connected with emotion learning.

Consequence driven systems theory is an attempt to understand the agent personality; it tries to find an architecture that will ground notions such as motivation, emotion, learning, disposition, anticipation, curiosity, confidence and behaviour among others, which are usually present in discussions about the agent personality (Bozinovski, 2002; 2003).

Consequence driven systems theory originated in an early reinforcement learning research effort to solve the assignment of credit problem using a neural network in 1981.

Consequence programming is understood as a subgoal-goal programming in model-free tasks. It is assumed that an agent is entering an unknown environment, looking for a known goal. The agent knows that a consequence of reaching a subgoal is a chance of reaching the goal. So, it will learn how to develop a policy of reaching a subgoal. Since there is no model, the agent will have to learn by doing, in real time. The process of reaching a subgoal driven by the fact that the goal may follow as a consequence is denoted as *consequence programming* or consequence managing (Bozinovski, 1995).

The Crossbar Adaptive Array (CAA) architecture was designed to solve the maze learning problem. The CAA architecture effort from the start was challenged by the mazes defined in the computer game *Dungeons and Dragons*, where there is one goal state but many rewarding and punishment states along the way. CAA adopted the concept of dealing with pleasant and unpleasant states, feelings and emotions (Fig. 1) (Bozinovski, 1999b). In contrast to reinforcement learning concept, CAA does not use any external reinforcement r; it uses only the current situation X as input. The CAA approach introduces the concept of state evaluation and connects it to the concept of feeling, which is used as an internal reinforcement entity (Bozinovski, 1982).

We will now describe the CAA architecture; the modified CAA architecture which is used in this paper will be explained further on.

The consequence driven systems theory assumes that the agent should always be considered as a three-environment system. The agent expresses itself in its *behavioural environment* where it behaves. It has its own *internal environment* where it synthesizes its behaviour, and it also has access to a *genetic environment* from where it receives its initial conditions for existing in its behavioural environment. The genetic and the behavioural environment are related. All the initial values are received from the genetic environment. The genetic environment is supposed to be a genotype reflection of the behavioural one. From the genetic environment CAA receives a string, denoted as a genome string or simply the CAA input genome. The input genome can have several chromosomes, but the most important one is the memory chromosome. It contains values W of the crossbar learning memory. So, if the genetic environment properly reflects the desirable or undesirable situations in the behavioural environment, then the genome string will properly inform the newborn CAA agent about the desirable (as well as undesirable) situations in the behavioural environment. Having initial emotional values of the desirable situations and using the emotional value back propagation mechanism, CAA will learn how to reach those situations. Having defined primary emotional values by the input genome, CAA learns behaviour in the environment by means of emotional value back propagation using its learning rule (Bozinovski, 1995).

One main module of the CAA architecture is its memory module (Fig. 1.), a connectionist weight matrix W. Each memory cell w_{aj} represents the emotional value toward performing action a in the situation j. It can be interpreted as a tendency, disposition, motivation (Bozinovski, 2002), expected reward, or by other terms, depending on the given problem for which CAA is used as a solution architecture. From the emotional values for performing actions in a situation k, CAA computes an emotional value v_k of being in the situation k. Having v_k, CAA considers it as a consequence of the action a performed previously in a situation j. So, it learns that performing a in j has as a consequence emotional value v_k, and on that basis it learns the desirability of performing a in j. Using crossbar computation over the crossbar elements w_{aj}, CAA performs its crossbar learning procedure, which has 4 steps:

(1) state j: choose an action in state j, $a_j = A\text{func}\{w_{*j}\}$. Let it be action a: let the environment return situation k;

(2) state k: feel the emotion being in state $k : v_k = V\text{func}\ \{w_{*k}\}$;

(3) state j: learn the emotion towards a in $j : w_{aj} = U\text{func}(v_k)$;

(4) change state: $j = k$, go to 1,

where w_{*k} means that computation considers all the possible values of the k-th column vector. The functions $A\text{func}\{\cdot\}$ and $V\text{func}\{\cdot\}$ are convenient to be defined in such a way as to assure the convergence of the above procedure. This is the so-called *on-route* acting method. The learning rule, the function $U\text{func}\{\cdot\}$ denoted as crossbar learning rule as used in CAA, is $w_{aj} = w_{aj} + v_k$, or emphasizing the rule of emotion: emotion$_{aj}$ = emotion$_{aj}$ + emotion(k), where emotion$_{aj}$ is an emotion toward performing action a in situation j, and emotion (k) is emotion being in situation k. The learning procedure described above is actually an emotion value back propagation procedure (secondary reinforcement procedure) (Bozinovski, 1982).

Another variant of the CAA method is the so-called off route acting method (or the point acting variant):

(1) state j: choose an action in state j, $a_j = A\text{func}\{w_{*j}\}$. Let it be action a: let the environment return situation k;

(2) state k: feel the emotion being in state $k : v_k = V\text{func}\ \{w_{*k}\}$;

(3) state j: learn the emotion toward a in $j : w_{aj} = U\text{func}(v_k)$;

(4) choose j; go to 1.

The on-route variant follows a route and applies the CAA learning method. In the off-route variant, the method can be applied at any state; the state can be chosen randomly or use some next-state-choosing policy (Bozinovski, 1995).

The new algorithm described in this paper uses this off-route acting method.

The CAA approach introduced learning systems that can learn without external reinforcement. In the philosophy of the CAA approach all the external reinforcement learning rules are classified into the supervised learning class. In the supervised learning system, a supervisor (e.g., a teacher) can give an external reinforcement of agent behaviour, and/or advice on how the agent should choose the future actions. In unsupervised (self supervised) systems, having no external teacher of any kind, the agent must develop an internal state evaluation system in order to compute an emotion (internal reinforcement).

3. At-subgoal-go-back algorithm

The at-subgoal-go-back algorithm (Bozinovski, 1995; Petruseva and Bozinovski, 2000) solves the problem of finding the shortest path from the starting state to the goal state in an environment with n states. The domains which are concerned are environments with n states, one of which is a starting state, one is a goal state, and some of them are undesirable and should be avoided. It is assumed that a path exists from the initial state s to the goal state g and the algorithm holds for environments which have graph representation with the following constraint: if there is a path from the initial node to some other node j, then there must be a path from that node j to the goal node. This problem is solved with the CAA agent architecture explained in Section 2.

The method which CAA uses is the backward chaining method. It has two phases: (1) search for a goal state, and (2) when the goal state is found, define the previous state as a subgoal state. The search is performed using some searching strategy, in this case random walk. When, executing a random walk, a goal state is found, a subgoal state is defined, and this subgoal becomes a new goal state. The process of moving from the starting state to the goal state is a single run (iteration, trial) through the graph. The next run starts again from the starting state, and will end in the goal state. In each run a new subgoal state is defined. The process finishes when the starting state becomes a subgoal state. That completes the solution finding process.

CAA has a random walk searching mechanism implemented as a random number generator with uniform distribution. Since there is a path to the goal state by assumption, there is a probability that the goal will be found. As the number of steps in a trial approaches infinity, the probability of finding a goal state approaches unity (Bozinovski, 1995).

The time complexity of an on-line search strongly depends upon the size and the structure of the state space, and upon a priori knowledge encoded in the agent's initial parameter values. When a priori knowledge is not available, the search is unbiased and can be exponential in time for some state spaces. Whitehead (Whitehead, 1991; 1992) showed that for some important classes of state spaces reaching the goal state for the first time, moving randomly can require a number of action executions that is exponential in the size of the state space. Because of this, for the state spaces which are concerned for this algorithm, it is assumed that the number of transitions between two states from the starting state to the goal state in every iteration is linear function of n (the number of states).

The agent starts from the starting state and should achieve the goal state, avoiding undesirable states. From each state the agent can undertake one of maximum m actions, which can lead to another state or to a certain obstacle. The agent moves in the environment randomly, and after a few iterations it learns a policy to move directly from the initial state to the goal state, avoiding undesirable states, i.e., it learns one path. After that it learns the optimal solution – the shortest path.

The initial knowledge is memorized in the matrix $W_{m\times n}$. The elements of the matrix W, $w_{ij}(i = 1,\dots,m; j = 1,\dots.n)$ give information for the states and are used for computing the actions; they are called SAE components (state – action – evolution). The elements of each column $(j = 1,\dots,n)$ give information for the states. The initial values of the elements of the column are all 0 if the state is neutral, with values -1 if the state is undesirable, and with values 1 if the state is a goal.

The learning method for the agent is defined by 3 functions whose values are computed in the current and the following state. They are (Bozinovski, 1995):

(1) The *function for computing an action* in the current state; let it be some function

$$y_j = \underset{a}{\mathrm{Afunc}}\,(w_{aj}), \quad j = 1,\dots,n; \;\; a = 1,\dots,m.$$

This function is of a neural type and is defined in the following way:

$$i = y_j = \underset{a\in A(j)}{\arg\max}\,\{w_{aj} + s_a\}\,, \qquad j = 1,\dots,n;\; a = 1,\dots,m.$$

s_a is the action modulation variable, which presents a searching strategy. The simplest searching strategy is random walk, implemented as $s = \mathit{montecarlo}[-0.5, 0.5]$, where $\mathit{montecarlo}[\mathit{interval}]$ is a random function which gives values uniformly distributed in the defined interval. With this function, the agent selects the actions randomly. Having that, NN-CAA (Neural Network CAA) will perform random walk until the SAE components receive values which will dominate the behaviour.

(2) The *function for the estimation of the internal, emotional value of being in a state*, computed in the consequence state. From the emotional values of performing the actions, CAA computes an emotional value of being in a state. The emotional value of being in a state k is computed as $v_k = \underset{i}{\mathrm{Vfunc}}\,(w_{ik})$ where $\mathrm{Vfunc}(\cdot)$ is a carefully chosen function for solving the given task. Having v_k, CAA considers it as a **consequence** of the action i performed previously in a state j. The functions $v_k,\ k = 1,\dots,n$ are computed in a "neural" fashion:

$$v_k = \mathrm{sgn}\left(\sum_{a=1}^{m} w_{ak} + T_k\right).$$

T_k is a neural threshold function (or a warning function) whose values are

$$T_k = \begin{cases} 0 & \text{if} \quad \eta_k = m, \\ \eta_k & \text{if} \quad p_k \le \eta_k < m, \\ 0 & \text{if} \quad \eta_k < p_k, \end{cases}$$

where p_k is the number of positive outcomes, η_k is the number of negative outcomes that should appear in the current state. The threshold function T plays the role of a modulator with which CAA will evaluate the states that are on the way.

(3) the *function for updating the memory*, computed in the current state. It learns that performing i in j has as a consequence the emotional value v_k, and on that basis it learns, using some function $w_{ij} = \text{Ufunc}(v_k)$. The learning rule in NN-CAA is defined by $w_{aj} = w_{aj} + v_k$; SAE components in the previous state are being updated with this rule, using the desirability of the current state.

The CAA *at-subgoal-go-back* algorithm for finding the shortest path in a stochastic environment is given in the following frame (Bozinovski, 1995):

```
CAA AT-SUBGOAL-GO-BACK ALGORITHM:
repeat
    forget the previously learned path
    define the starting state
    repeat
        from the starting state
        find a goal state moving randomly
        produce a subgoal state using the CAA learning
        method
        mark the produced subgoal state as a goal state
    until starting state becomes a goal state
    export the solution if better than the previous one
forever.
```

The experiment needs several iterations. When the goal state is reached, the previous state is defined as a *subgoal*. The goal is considered a consequence of the previous state, from which the goal is reached. Every subgoal becomes a new goal. With this algorithm, except in the first iteration, the agent does not go to the end goal state, but it starts a new iteration when it reaches a subgoal. A subgoal is a state which has a positive value for some of its elements $w[i, j]$. The process of learning finishes when the initial state becomes a subgoal. It means that at that moment the agent learnt *one path*; it learnt a policy how to move directly from the initial state to the goal state, avoiding undesirable states. Through the process of learning in every iteration, the elements w [i, j] change, so that when the initial state becomes a subgoal, they have values that enable deterministic moving from the starting to the goal state. The algorithm guarantees finding one solution, i.e., a path from the starting state to the goal. For solving the shortest path problem, another memory variable should be introduced, which will memorize the length of the shortest path, found in one *reproduction period*. The period in which the agent finds one path is called the reproductive period and, in general, in one reproductive period the agent cannot find the shortest path. After finding one path, the agent starts a new reproductive period when it learns a new path, independent of the previous solution. The variable *shortest path* is transmitted to the following agent generation in a genetic way. In this way the genetic environment is an optimisation environment which enables the memorisation of the shortest path only, in a series of reproductive periods. This optimisation period will end in finding the shortest path with probability 1. Since the solutions are continuously generated in each learning epoch, and because they are generated randomly and independently of the previous solution, then, as time approaches infinity, the process will generate possible solutions with probability 1. Among all possible solutions the best solution, the shortest path, is contained with probability 1. Since CAA will recognize and store the shortest path length, it will produce the optimal path with probability 1 (Bozinovski, 1995).

3.1. Estimation of the complexity of the at-subgoal-go-back algorithm.

Finding the shortest path with the at-subgoal-go-back algorithm is in exponential time because the agent should learn all paths from the initial state to the goal state. The number of paths from one node to another in a graph is an exponential function of n, the number of nodes in the graph. This assertion is proved with two theorems (whose proofs can be found in (Petruseva, 2006a)).

Theorem 1. *If A is the adjacency matrix for the graph G with n nodes, then the element c_{ij} of the matrix A^k shows the number of paths with length k from the node i to the node j.*

Note: This result is known, but because the author could not find the proof, the result is proven and the way it is proven is used for the estimation of the number of paths from one node to another, depending on n, the number of nodes (and m, the number of arcs from a node) in the graph. (Petruseva, 2006a).

Since the agent learns one path in every reproductive period, it is necessary to express the total number of paths from the initial state to the goal state as a function of the number of states n (and the number of actions m in every state). The following theorem shows that dependence and gives the estimation of the lower and the upper boundary for the number of paths with length s from one node to another in a graph with n nodes.

Theorem 2. *The number of paths with length s from the node i to the node j, b_s^{ij}, can be estimated as $cmk_{\min}^{s-2} \leq b_s^{ij} \leq mk_{\max}^{s-2}$, for $s \geq 2$, where c is some constant, m is the number of arcs which go out of i, $k_{\min}$ and $k_{\max}$ are the minimum and the maximum number of arcs which can* *enter a certain state in the graph (ij are indices, $s-2$ is power).*

The number of all paths B from the initial state i to the goal state g is the sum of all paths from i to g with length $d, d+1, d+2, \ldots, n-1-p$, where d is the length of the shortest path, and $n-1-p$ is the length of the longest path.

B is estimated by

$$\Omega\left[m\left(\frac{k_{\min}^{n-p-2}-k_{\min}^{d-2}}{k_{\min}-1}\right)\right].$$

S, the number of reproductive periods, is greater than or equal to B, so S can also be estimated by

$$S=\Omega\left[m\left(\frac{k_{\min}^{n-p-2}-k_{\min}^{d-2}}{k_{\min}-1}\right)\right],$$

i.e., S is an exponential function of n, the number of nodes (Petruseva, 2006a).

4. Motivation for the proposed linear time algorithm

The main motivation for the proposed linear time algorithm is the exponential time at-subgoal-go-back algorithm. In order to decrease its complexity, dynamic programming is applied, and one "intersolution"—polynomial time algorithm—is obtained with complexity $\Theta\left(n^2\right)$ (Petruseva, 2006b). This polynomial algorithm is, in fact, an immediate predecessor of the linear algorithm proposed in this paper.

The conclusion on this algorithm was that defining the function v in the subgoals in such a way as to precisely measure the consequences of the agent's actions (starting from the goal) gives drastic improvement of the speed of convergence of the algorithm. Since this function does not give information about the consequences of the agent's actions from the start state to the goal, considering the definition of the emotions, the motivation was to change learning functions so that the emotion function would measure the consequences of the agent's actions even from the start state, i.e., to implement emotion learning all the time. So the linear time algorithm was obtained, which implements breadth-first searching. In addition, it turned out that, in general, the learning time is independent of n, the number of states, and the algorithm can be applied to an arbitrary environment!

In the next sections the proposed linear time algorithm will be presented with some preliminary notions.

5. Models and planning

By a *model of the environment* we understand anything that the agent can use to predict how the environment will respond to its actions (Sutton and Barto, 1998). The model can be *completely specified* or *incompletely specified* (Bozinovski, 1995). A completely specified model assumes knowledge of all states, transition values, and transition probabilities. An incompletely specified model is one in which some of the mentioned information is missing. A special class of an incompletely specified environment is a *model-free* environment. In most cases, such an environment can be imagined as a cave in which even the number of states is unknown. A basic assumption is that when the problem solving agent finds a *target state*, it will recognize such a state. For practical purposes, a model-free environment is often given by a maximal number of possible states and a maximal number of actions per state. An agent is to enter the cave and learn by doing, from the consequences of its trials.

Models can be used to mimic or simulate experience. The model is used to *simulate* the environment and produce *simulated experience*.

The word "planning" is used in several different ways in different fields. It is used to refer to any computational process that takes a model as input and produces or improves a policy for interacting with the modelled environment.

Model → Planning → Policy

In this paper planning is viewed primarily as a search through the state space for a path to a goal. The difference between learning and planning methods is that whereas planning uses simulated experience generated by the model, learning methods use real experience generated by the environment. Learning methods require only experience as input, and in many cases they can be applied to simulated experience just as well as to real experience.

Learning and planning are deeply integrated in the sense that they share almost all the same machinery, differing only in the source of their experience (Sutton and Barto, 1998).

Peng and Williams (1992) used a combination of planning strategies based on dynamic programming with learned world models for effective learning. They proposed a prioritizing scheme in the Dyna architecture in order to improve its efficiency. They demonstrated, using simple tasks, that the use of such prioritizing schemes leads to drastic reductions in the computational effort and corresponding dramatic improvements in the performance of the learning agent.

Knowledge about the connections of the states in the environment is considered as a part of the model of the environment in this paper. The agent learns this partial model of the environment through real experience, i.e., in every successor state s it learns the first predecessor state s' and the action taken in s' which leads to s. When the agent finds the goal state, simulated experience is generated in these predecessor states (marking the shortest path). After that, in the planning phase, it moves through the optimal path using this simulated experience.

6. New emotion learning algorithm and the modified CAA architecture

The following notation is used in this paper: S denotes the finite set of states of the state space, $A(s)$ is a finite set of actions that can be executed in $s \in S$, the size of the state space is $n = |S|$. This algorithm can be applied for solving the shortest path problem in *an arbitrary* deterministic environment with n states. The assumption is that there is a path from the initial state to the goal state and the maximal number of actions which the agent can undertake in every state is the constant m $(m < n)$.

Figure 2 shows a domain which is an environment with n states, one of which is a starting state (6), one is a goal state (10), and some states are undesirable and should be avoided. The agent starts from the starting state and should learn to move along the shortest path to the goal state, avoiding undesirable states. From each state, the agent can undertake one of possible m actions, which can lead to another state or to a certain obstacle. By succ(a_i, s) we shall denote the next state which is achieved from s, or the obstacle which can appear after executing the action a_i in s.

The agent uses *direct learning (emotion learning)* based on real experience (which implements breadth-first searching), combined with planning (based on simulated experience). The agent architecture used is shown in Fig. 3. It is a modified CAA architecture explained in Section 2. The memory module W is widened with the memory elements $ps[j]$ and $pac[j]$ $(j = 1, \dots, n)$, $ps[j]$ is the element for memorizing the first predecessor state s from which the state $j$is achieved, and $pac[j]$ is the element for memorizing the action chosen in that state s, which leads to j. So, the memory elements $ps[j]$ and $pac[j]$ $(j = 1, \dots, n)$ are used for learning the partial model of the environment in the phase of model learning.

This phase is being accomplished simultaneously with the phase of direct learning. When the agent finds the goal state, simulated experience is generated using these elements. In fact, these elements are used for finding the optimal path from the starting state to the goal state, which is being marked backwards – from the goal state to the starting state. The state values $V[j]$ $(j = 1, \dots, n)$ are not being computed from the elements $w[i, j]$ in j (as in the CAA method) but from the previous value $V[s]$ in the predecessor state s.

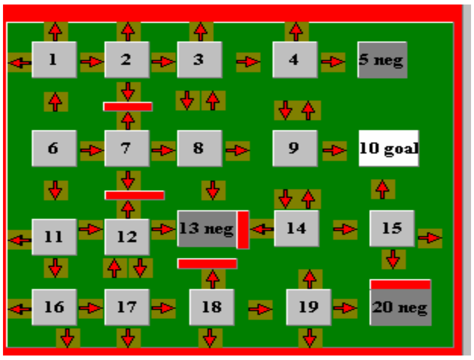

Fig. 2. Arbitrary environment.

For our new emotion learning algorithm the three learning functions explained in Section 2 are given in the following way:

1. The function **Afunc** for choosing an action in a state j is:

   (a) In the exploring phase until the goal state is found, in every state the agent chooses and explores every action from that state one by one, i.e., $y(j) = a$ $(a = 1, \dots, m)$;

   (b) When the goal is found and simulated experience is generated from the partial model, the agent uses this simulated experience in the phase of planning, moving through the optimal path, by choosing the actions in every state from the initial to the goal state with the function

$$y(j) = \arg\max_{a \in A(j)} w(a, j).$$

2. The function **Vfunc** for computing the emotional state of the agent in a state k is

$$V(k) = \begin{cases} -100 & \text{if the state } k \text{ is undesirable,} \\ V(j) + 1 & \text{otherwise (} j \text{ is the first} \\ & \text{predecessor of } k\text{).} \end{cases}$$

3. The function **Ufunc** for updating the memory elements $w[a, j]$ is given by

$$w[a, j] = \begin{cases} V[k] & \text{if the action } a \text{ in } j \text{ leads to the} \\ & \text{state } k \text{ which is not passed} \\ & \text{before} \\ -100 & \text{if the action } a \text{ in } j \text{ leads to an} \\ & \text{obstacle or to a previously} \\ & \text{learned state.} \end{cases}$$

Initially, all $V(j) = 0$, $w(a, j) = 0$ $(j = 1, \dots, n$, $a = 1, \dots, m)$, except for the undesirable states j, $w[a, j] = -1$ $(a = 1, \dots, m)$, and for the goal state g, $w[a, g] = 1$ $(a = 1, \dots, m)$.

In the exploring phase, in every state j (except for undesirable states) the agent explores every action from that state. The agent chooses the actions in the state one by one. After taking an action in j, if the successor state k is not an undesirable state or a previously passed state, the agent (1) goes in that state; (2) updates the value $V(k)$ from $V(k) = 0$ to $V(k) = V(j)+1$; (3) in k the agent memorizes (learns) the previous state **j** and the action which leads to k; (4) updates the value $w[a, j] = V[k]$ and returns to j. The value $w[a, j] = V[k]$ means that taking the action a in j the agent will learn that the shortest path from the initial state to the next state k is $V[k]$.

After exploring all actions from the initial state s, the agent has found (learned) all consequence states ss with $V(ss) = 1$ (the initial state has $V(s) = 0$). After that, the agent goes in every state ss with $V(ss) = 1$ one by one and explores their actions in the same way. So, after being in all these states (with $V(ss) = 1$ and exploring their actions), the agent finds all states ss with $V(ss) = 2$ and goes in each of these states, and so on.

When the agent finds the goal state, this exploring phase finishes and simulated experience is generated from the partial model which the agent has learned: every memory element $w[i, j]$ from the predecessor states from the goal state to the initial state, for which the state j and the action i was memorized in the phase of model learning, gets some high value (100). In this way the shortest path is marked from the goal state to the initial state. In the planning phase the agent moves through the optimal path using simulated experience.

We can consider these predecessor states which belong to the optimal path as *subgoals* which lead to the goal. The algorithm is given in Fig. 4.

The algorithm uses the so-called *off road* acting method, mentioned in Sec. 2., because the agent always chooses the states with the same state value one by one (this is also called the point acting variant). The learning procedure in every state j can be described as follows:

```
repeat
  state $j$: choose an action $a$ in state $j : y(j) = a$
  if obtained state is $k$, and if it is not previously learned
  state, then
  Begin
    state $k$: compute the emotion in the state $k$:
```

$$V(k) = \begin{cases} -100 & \text{if the state is undesirable,} \\ V(j) + 1 & \text{otherwise.} \end{cases}$$

```
    If $k$ is not undesirable state, learn the predecessor state
    $j$ and the action $a$ which leads to $k$ : $Ps[k] = j$;
    $Pac[k] = a$;
    state $j$: learn the emotion toward $a$ in $j$ : $w[a, j] =
    V(k)$;
  End;
    if the action leads to an obstacle, or if the obtained
    state is a previously learned state, then $w[a, j] =
    – 100$;
until all actions $a \in A(j)$ in the state $j$ are explored one
by one
choose $j$.
```

```
Begin
    s← initial state ;
   k=0;
   learning(s);
   while k=0 do
     begin
      successors;
      s=snew;
     end
  end.
```

Fig. 4. Emotion learning algorithm.

The state value $V(s)$ is defined as an emotional value of the agent being in the state s. In the case of solving the problem of finding the shortest path from the initial state to the goal state, $V(s)$ measures the shortest distance of the state s from the starting state.

Wittek (1995) explains that human emotions are energy complexes composed of pictures, whose elements are thoughts, words, and acting from the past. They are consequences of our previous thinking, speaking and acting (previous experience). Emotions are part of our character, personality: we have built them, and we can also unbuild them. If we become aware that the emotion which is active is not positive, we can change it sooner or later, by positive thinking, opposing a positive thoughtful complex. But if it is not the case, the emotion is a consequence from the past, from our previous way of thinking and acting. In this implementation of emotion learning, the emotional value $V(s)$ expresses the consequence of the agent's actions: taking an action from the previous state and being in the state s, the agent learns in s that the shortest path from the starting state is $V(s)$. $V(s)$ summarizes the whole acting of the agent from the initial state to the state s. So, the function of the emotion V should measure the consequence of the agent dealing and of its actions as much as possible (about the task which has to be accomplished). The algorithm is presented in Fig. 4.

The procedures *learning, successors,* simulating-*exp*, and *planning* are given below.

```
Procedure learning(s)
  begin
   $i = 0$;
    repeat
     $i = i + 1$;
     take the action $a_i$ in $s$: $y[s] = a_i$
     compute $ss = succ\,(a_i, s)$;
```

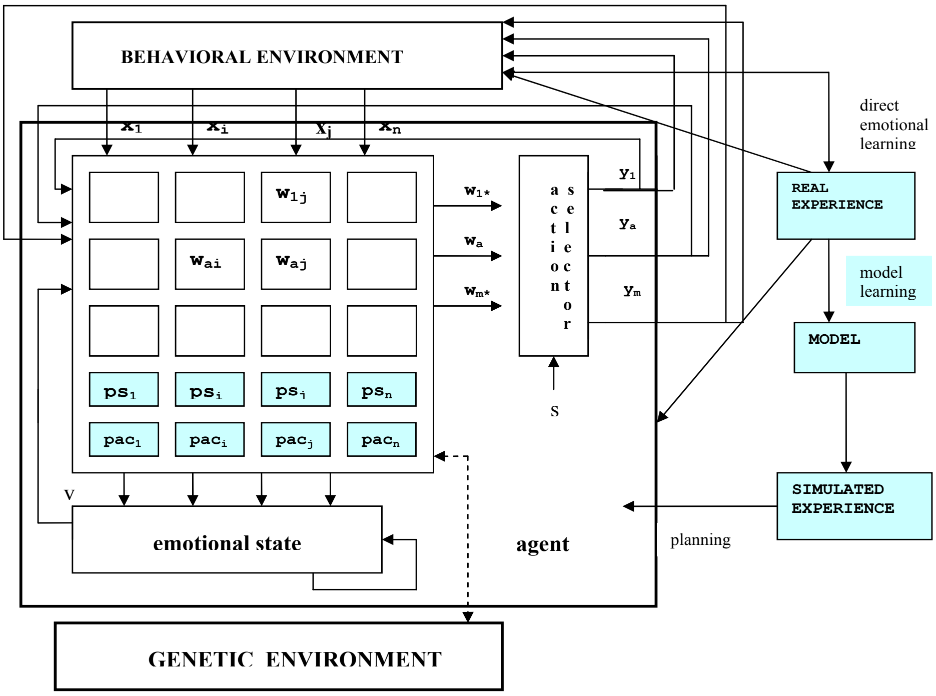

Fig. 3. Modified CAA architecture.

```
 if $ss$ is not undesirable state, obstacle or previously
  learned state, then
  begin
  go in $ss$ and learn the emotional value in $ss$:
  $V(ss) = V(s) + 1$;
  learn the predecessor state $s$ and the action $a_i$ which
  leads to $ss$ :
   $Ps(ss) = s$; $Pac(ss) = a_i$;
  update the emotional value $w[a_i, s]$ toward taking the
  action $a_i$ in $s$:
   $w[a_i, s] = V(ss)$;
  return to state $s$;
  end;
 if $ss$ is a goal state then
  begin
   $k = 1$;
   simulating-exp;
   planning;
  end;
 else
  begin
  if $ss$ is undesirable state then $V[ss] = -100$;
  if the action leads to an obstacle, previously learned,
  or undesirable state then $w(a_i, s) = -100$;
  end;
   until $k = 1$
 end.
```

```
Procedure successors
  Begin
    $\forall$ state $s1$ for which  $V(s1) = V(s) + 1$
  repeat
      choose state $s1$
      snew = s1;
      learning (snew);
  until $k = 1$
 end.
```

```
Procedure simulating-exp
  begin
   $s'$=goal state;
  repeat
     $subg = ps[s']$;
     $act = pac[s']$;
     $w[act, subg] = 100$;
     $s' = subg$;
  until $s' = startingstate$
 end.
```

```
Procedure planning
 begin
  $s = startingstate;$
   repeat
     $a = arg\max_{a_i} w(a_i, s);$
     $ss = succ(a, s);$
     $s = ss;$
   until $s = goalstate$
 end.
```

7. Estimation of the complexity of the algorithm

If we call this algorithm an *emotion-learning algorithm*, the following theorem gives the estimation of its complexity.

Theorem 3. *Without the operations for initiation, in general, the emotion-learning agent with the emotion learning algorithm finds one shortest path from the initial state to the goal state in an arbitrary deterministic environment in time which does not depend on n, the number of states, but only on the length of the shortest path (it is task dependent), but in the worst case, depending on the position of the goal state, it can be at most O (n). The only assumption is that there is a path from the initial state to the goal state, and the maximal number of actions in each state is the constant m.*

Proof. Let the environment be presented with an arbitrary directed graph G with n nodes (Fig. 5).The connection between nodes is arbitrary. Between two nodes there may be return arcs. Some arcs can lead to undesirable nodes (un) or to obstacles. Let the maximal number of arcs which can go out of a node be m (this is the graph representation for the environment in Fig. 2). The assumption is that there is a path between the starting node (S) and the goal node (G).

The agent finds one shortest path from the initial to the goal node because it first finds all nodes with shortest distance $d = 1$ from the initial node, i.e., all nodes s with $V(s) = 1$, and then, from the nodes s with $V(s) = 1$ it finds all nodes with the shortest distance $d = 2$ from the initial node, i.e., all nodes s with $V(s) = 2$ (the nodes which are not passed before). If this is not the case, i.e., if some node with $V(s) = 2$ has a distance $d = 1$ from the start node, it means that there is an arc from the initial node to that node, and this is in contrast to the assumption that from the start node all nodes with the shortest distance $d = 1$ are found. By mathematical induction, let us assume that from all nodes s with $V[s] = i - 2$ the agent finds all nodes with the shortest distance $d = i - 1$ from the initial node. Let us assume that in the next step from one node with $V[s] = i - 1$, the agent finds the goal node with the shortest distance $d = i$ from the initial node. Because there is a path from the initial to the goal node by assumption, the goal node will be reached, and $d = i$ is the shortest distance of the goal node from the initial node. If it this not the case, i.e., if there is a shorter path, then there is an arc from some node s with $V[s] = 0$, or $V[s] = 1$, or . . . $V[s] = i - 2$ to the goal node, which is in contrast to the assumption that from the previously found nodes s with $V[s] \in \{0, 1, \dots, i - 2\}$ all nodes with the shortest distance $d \in \{1, \dots, i - 1\}$ are found. When the agent finds the goal node, the exploring phase is finished, and it does not have to find all nodes with $V(s) = i$.

In every state s which the agent passes, it memorizes the *first* predecessor state and the action taken in that predecessor state which leads to s. The *next* predecessor states (together with the action) which lead to s are not memorized. Using the memory elements $ps(s)$ and $pac(s)$, the agent plans the optimal path (from the goal state to the initial state). So, the agent finds *one* shortest path from the initial state to the goal state.

For the initiation of the elements $w[i, j]$ $(i = 1, \dots, m;\ j = 1, \dots, n)$ the time needed is Θ(mn). Without operations for initiation, the time does not depend on n in a general case, but only on the shortest distance from the initial to the goal state, and in the worst case, depending on the position of the goal state, the complexity can be at most $O(n)$.

Depending on the structure of the domain and the position of the goal state, we can obtain different estimations for the complexity. Below are some examples (for which the time needed for initiation is not considered).

The worst case complexity can be estimated for the domains in Figs. 6(a) and 6(b), where the actions from every state lead to previously unexplored states and then the number of states s with $V(s) = 1$ is m, the number of states with $V(s) = 2$ is $mm = m^2$, and so on, with $V(s) = i$ is m^i (i is the shortest distance from the initial to the goal state). When the agent is in one state j, the maximal number of operations for choosing an action, going to the next state, verifying whether that state is an undesirable or a previously passed state, or an obstacle, updating the element $w[i, j]$ and the elements $prs[j]$ and $pac[j]$, can be at most a (a is a constant). Because in every state j the maximal number of actions is m and so the agent does m transitions for exploring all actions, the number of operations for exploring all actions and updating the elements $w[i, j]$ can be at most am. The total number of operations can be estimated as

$$\begin{aligned} F(n) < {} & am + am^2 + am^3 + \dots + am^{i+1} + bi \\ & + (cm + d)i, \end{aligned}$$

where the first term is the total number of operations for exploring all actions in the initial state, the second term is the total number of operations for exploring all actions in the states with $V(s) = 1$, and so on, and the term am^{i+1}

is the total number of operations for exploring all actions in the states with $V(s) = i$ (the states which have the same shortest distance from the initial state as the goal state); bi is the maximal number of operations for generating simulated experience, and $(cm + d)i$ is the maximal number of operations for moving along the optimal path in the planning phase (b and c are constants). For the domain in Fig. 6(a) many states remain unexplored. The complexity can be estimated by $O(m^{i+1})$. It depends only on the length of the shortest path i and does not depend on n.

For the domain in Fig. 6(b) all n states are explored and the total number of operations can be estimated by

$$\begin{aligned} F(n) &< am + am^2 + am^3 + \cdots + am^{i+1} + bi \\ &\quad + (cm + d)i \\ &= am(1 + m + m^2 + m^3 + \ldots m^i) + bi \\ &\quad + (cm + d)i \\ &= amn + bi + (cm + d)i. \end{aligned}$$

The term $1 + m + m^2 + m^3 + \cdots + m^i$ is the total number of the states explored, which is n.

In this case, depending on the position of the goal state, the complexity can be at most $O(n)$. In every other case where some of the actions can lead to previously explored states, the complexity can be decreased; for example, for the graph in Fig. 5, the maximal number of operations which the agent does for finding the shortest path can be estimated in the following way:

$$\begin{aligned} F(i) &< am + 3am + 4am + 3am + 2am + bi \\ &\quad + (cm + d)i \\ &= am(1 + 3 + 4 + 3 + 2) + bi + (cm + d)i \\ &< am(1 + 3 + 5 + 7 + 9) + bi + (cm + d)i \\ &= am[1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 \\ &\quad - (2 + 4 + 6 + 8)] + bi + (cm + d)i \\ &= am[9 \cdot 10/2 - 2(1 + 2 + 3 + 4)] + bi \\ &\quad + (cm + d)i \\ &= am[(2i + 1)(i + 1) - 2i(i + 1)/2] + bi \\ &\quad + (cm + d)i \\ &= am[(i + 1)(2i + 1 - i)] + bi + (cm + d)i \\ &= am(i + 1)(i + 1) + bi + (cm + d)i \\ &= am(i + 1)^2 + bi + (cm + d)i. \end{aligned}$$

The complexity does not depend on n, but only on the length of the shortest path. It can be estimated by $O(i^2)$. ■

One can conclude that the basic difference between this algorithm and the at-subgoal-go-back algorithm is in the definition of the function V for the evaluation of the inner state of the agent, because this function gives decisive information about the process of agent learning. The at-subgoal-go-back algorithm has a different choice of functions of learning, and its complexity is exponential.

If we compare these two algorithms, we may conclude the following: with the at-subgoal-go-back algorithm in every iteration the function V evaluates the state only with three values: with 1 if the state is a goal or subgoal, with 0 if it is neutral state and with -1 if the state is undesirable. The agent must pass through one path several times to learn it. It must learn all paths, compare them, and conclude which one is optimal. The function V for this algorithm changes its values even from the goal state and it does not give information about the agent's actions from the initial state to the subgoals. For learning one path, the time needed is $O(n^2)$ because the initial state becomes a subgoal after at most n iterations, and the assumption is that the number of transitions in one iteration is a linear function of n. Since the agent should learn all paths from the initial to the goal state, the time needed is an exponential function of n (Sec. 3.1).

Wittek (1995) and Glasser (1963) explain that human emotions are consequences of previous experience (thinking and acting). If we want to describe or understand our emotions properly, we should not say: "I feel good" or "I feel bad", but we should ask ourselves "Why do I feel good?" or "Why do I feel bad?", and we should look at their components, i.e., thoughts and pictures which appear when some emotion is active, as precisely as possible. In this way, we should come to the cause of our emotion, and if it is negative, we can transform it into a positive one, by positive thinking (Wittek, 1995).

In the at-subgoal-go-back algorithm the function for the evaluation of the emotion in a state, $v(s)$, is defined in the sense "good" (with the value 1), "bad" (with the value -1) and "neutral" (with the value 0). In this case this function does not give precise information about the consequences of the agent's actions: from the starting state to the goal, the states in which the agent has been have the emotional value 0, only the undesirable states which are on the road to the goal are estimated with the value -1, the other states are treated equally, i.e., as neutral. Even in the goal (subgoal) state the value of v is changed into 1 and this function has an equal value in all subgoals, i.e., all subgoals are treated equal with the value 1 regardless of their distance from the goal. In this case, we also have emotion learning (a positive emotion when the agent is in the goal (subgoal) state, negative in undesirable states and neutral in neutral states) but the evaluation is more general; the function has only three values. In this case learning is slow.

The three learning functions of the new algorithm are changed. The agent explores every action in a state and the function V gives more detailed information about the consequences of the agent's actions concerning the problem which has to be accomplished, starting even from the initial state. In this case of solving the shortest path

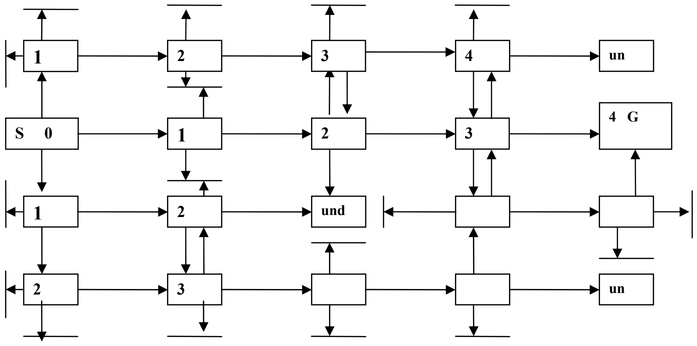

Fig. 5. Arbitrary environment.

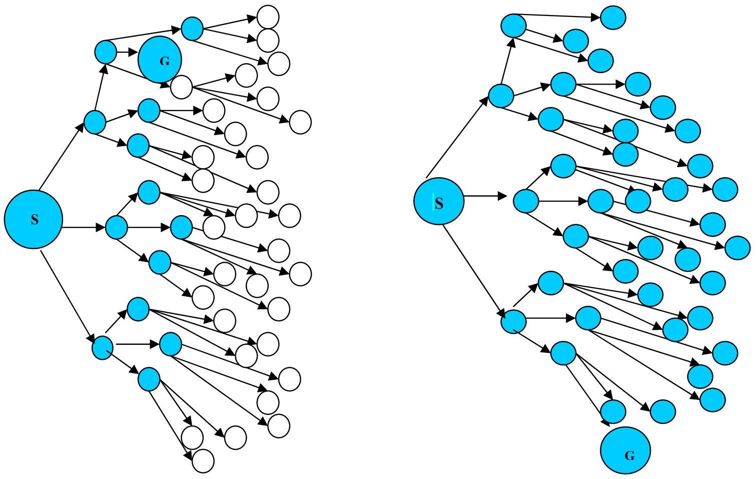

Fig. 6. (a) Domain for which the complexity can be estimated with $O(m^{i+1})$. Many states remain unexplored; (b) domain for which all states are explored and the complexity can be estimated with $O(n)$.

problem, this function measures in every state the shortest distance from the initial state. In this way, this function implements properties of human emotions. The process of learning is speedy because the agent finds the goal state in one iteration, and, in general, it does not have to explore all states. For the phase of exploring, until the agent finds the goal state, the time needed can be a polynomial or an exponential function of the length of the shortest path and in this case it does not depend on n, but in the worst case, depending on the position of the goal state (when the goal is near the end of the state space), it is at most $O(n)$. For the phase of generating simulated experience, when the shortest path is marked from the goal to the initial state, the time needed is only a linear function of the length of the shortest path i. And for the phase of planning, when the agent moves through the shortest path, the time needed is also a linear function of i.

Other differences between these two algorithms can be summarized as follows: the at-subgoal-go-back algorithm deoes not learn the model of the environment and the searching method is *on-route*, while the proposed linear time algorithm learns the partial model of the environment, the searching method is *off-route* and the algorithm can be applied in an arbitrary deterministic environment.

We can conclude that the function V which expresses the emotional, internal state of the agent should carry the information for solving the problem, i.e., it should be defined in such a way so that it gives an answer in every passed state as to what the solution of that problem in that state is. In fact, this function should measure the consequences of the agent's actions as precisely as possible, starting from the initial state.

8. Conclusion

This paper presents an algorithm which solves the shortest path problem with an emotional agent. The domains for which the algorithm can be applied are arbitrary deterministic environments with n states. The only assumption is that there is a path from the initial to the goal state. The maximal number of actions which the agent can undertake in one state is m (constant). The algorithm originates from an algorithm which in exponential time solves the same problem. By implementing emotion learning, an algorithm which implements breadth-first searching is obtained and the function which evaluates the emotional state of the agent is defined in such a way as to give as detailed information as possible about the consequences of the agent's actions starting even from the initial state. In this way, without operations for initiation, the complexity in general does not depend on n, the number of states, but it is task dependent—it depends only on the length of the shortest path. In the worst case, depending on the position of the goal state, the time needed can be at most $O(n)$.

References

Botelho L.M and Coelho H. (1998). *Adaptive agents: Emotion learning*, Association for the Advancement of Artificial Intelligence, pp. 19–24.

Bozinovski S. (1995). *Consequence Driven Systems, Teaching, Learning and Self-Learning Agent*, Gocmar Press, Bitola.

Bozinovski S. (1982). A self learning system using secondary reinforcement, *in:* E. Trappl (Ed.) *Cybernetics and Systems Research,* North-Holland Publishing Company, pp. 397–402.

Bozinovski S. and Schoell P. (1999a). *Emotions and hormones in learning*. GMD Forschungszentrum Informationstechnik GmbH, Sankt Augustin.

Bozinovski S. (1999b). Crossbar Adaptive Array: The first connectionist network that solved the delayed reinforcement learning problem, *in:* A. Dobnikar, N. Steele, D. Pearson, R. Albrecht (Eds.), *Artificial Neural Nets and Genetic Algorithms*, Springer, pp. 320–325.

Bozinovski S. (2002). Motivation and emotion in anticipatory behaviour of consequence driven systems, *Proceedings of the Workshop on Adaptive Behaviour in Anticipatory Learning Systems,* Edinburgh, Scotland, pp. 100–119.

Bozinovski S. (2003). Anticipation driven artifical personality: Building on Lewin and Loehlin, *in:* M. Butz, O. Sigaud, P. Gerard (Eds.) *Anticipatory Behaviour in Adaptive Learning Systems,* LNAI 2684, Springer-Verlag, Berlin/Heilderberg, pp. 133–150.

Glaser J. (1963). *General Psychopathology*, Narodne Novine, Zagreb, (in Croatian).

Jorgen B.J and Gutin G. (2001). *Digraphs, Theory, Algorithms and Applications*, Springer-Verlag, London.

Koenig S. and Simmons R.(1992). *Complexity Analysis of Real-Time Reinforcement Learning Applied to Finding Shortest Paths in Deterministic Domains*, Carnegie Mellon University, Pittsburgh.

Peng J. and Williams R. (1993). Efficient learning and planning with the Dyna fmework. *Proceedings of the 2nd International Conference on Simulation of Adaptive Behaviour: From Animals to Animates,* Hawaii, pp. 437–454

Petruseva S. and Bozinovski S. (2000). Consequence programming: Algorithm "at subgoal go back". *Mathematics Bulletin*, Book 24 (L), pp. 141–152.

Petruseva S. (2006a). Comparison of the efficiency of two algorithms which solve the shortest path problem with an emotional agent, *Yugoslav Journal of Operations Research*, **16**(2): 211–226

Petruseva S. (2006b). Consequence programming: Solving a shortest path problem in polynomial time using emotional learning, *International Journal of Pure and Applied Mathematics*, **29**(4): 491–520.

Sutton R. and Barto A. (1998). *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA.

Whitehead S. (1991). A complexity analysis of cooperative mechanisms in reinforcement learning, *Proceedings of AAAI*, pp. 607–613.

Whitehead S. (1992). *Reinforcement learning for the adaptive control of perception and action,* Ph.D. thesis, University of Rochester.

Wittek G. (1995). *Me, Me, Me, the Spider in the Web. The Law of Correspondence, and the Law of Projection*, Verlag DAS WORT, GmbH, Marktheidenfeld-Altfeld.

Received: 26 Feb 2007
Revised: 10 Feb 2008
Re-revised: 16 July 2008