Dynamic Concept Model Learns Optimal Policies

Cs. Szepesvári

***Abstract*— Dynamic Concept Model (DCM) is a goal-oriented neural controller, that builds an internal representation of events and chains of events in the form of a directed graph and uses spreading activation for decision making [1]. It is shown, that a special case of DCM is equivalent to reinforcement learning (RL) and is capable of learning the optimal policy in a probabilistic world. The memory and computational requierements of both DCM and RL are analyzed and a special algorithm is introduced, that ensures intentional behavior.**

I. Introduction

Reinforcement learning is a flourishing field of neural methods. It has a firm theoretical basis and has been proven powerful in many applications. A brain model based alternative to RL has been introduced in the literature: It integrates artificial neural networks (ANN) and knowledge based (KB) systems into one unit or agent for goal oriented problem solving. The agent may possess inherited and learnt ANN and KB subsystems. The agent has and develops ANN cues to the environment for dimensionality reduction in order to ease the problem of combinatorial explosion. A dynamic concept model was forwarded that builds cue-models of the phenomena in the world, designs action sets (concepts) and make them compete in a neural stage to come to a decision. The competition was implemented in the form of activation spreading (AS) and a winner-take-all mechanism. The efficiency of the algorithm has been demonstrated for several examples, however, the optimality of the algorithm have not yet been proven in general. Here, a restriction to Markov decision problems (MDP) shall be treated making possible to show the equivalence of a special AS and RL[1]. The equivalence in this special case means, that DCM has all the advantages of RL, moreover it keeps track of more distinctions allowing faster convergence and generalization [2].

The author is with the Department of Mathematics, Attila József University of Szeged, Szeged, Hungary 6720

[1]Note, that RL and MDP are used interchangeable in the text, as RL is a special solution to MDP.

II. Markovian decision problems

The bases of the used theoratical framework is a class of stochastic optimal control problems called *Markovian Decision Problems* (MDP). Such a problem is defined in terms of a discrete-time stochastic dynamical system with finite state set

$$S = \{s_1, s_2, \ldots, s_n\}.$$

At each discrete time step t, ($t = 0, 1, 2, \ldots$) a controller observes the system's current state ($s(t)$) and generates a control ***action*** ($a(t)$), which is applied as input to the system. Actions can be choosen from a finite set A. If $s(t) = s_i$ is the observed state, and the controller generates the action $a(t) = a$, then at the next time step the system's state will be $s(t+1) = s_j$ with probability $p_{ij}(a)$. Further, it is usual to assume, that the application of action a in state s_i incurs an ***immediate cost*** $c_i(a)$.

A closed-loop ***policy*** (or simply a policy) specifies each action as a function of the observed state. Thus, such a policy is a function $\mu : S \rightarrow A$. For any policy μ there is a real-valued function, $f^\mu : S \rightarrow \mathbf{R}$, called the cost function, corresponding the policy μ. Here we define it to be the ***expected total infinite-horizon discounted cost*** that will be incurred over time given that the controller uses policy μ:

$$f^\mu(s) = E_\mu\Big[\sum_{t=0}^{\infty} \gamma^t c(t) | s(0) = s\Big], \qquad (1)$$

where γ, $0 < \gamma < 1$, is a factor used to discount future immediate costs, and E_μ is the expectation assuming the controller always uses policy μ.

The objective of the type of Markovian decision problem we consider is to find a policy that minimizis the cost of each state s as defined by Eq. (1). A policy, that achieves this objective is an ***optimal policy*** which we will denote by μ^*. Note, that there may be more than one optimal policy for the same problem, but to each optimal policy corresponds the same cost function, which is the ***optimal cost function.***

A. Reformulating goal oriented behavior

In a previous study a different formalism was used, namely goal oriented behavior [1]. Let us call the underlying problem of goal oriented begavior to be the goal oriented decision problem (GDP). It will be shown, that GDP is a special case of MDP. In GDP a finite, fixed set of goals $G \subset P(S)$ is defined. Every goal $g \in G$ is identified by the set of states in which the goal is satisfied. Now, the immediate *cost of a state*, s, is defined to be the number of goals, which are not satisfied in the given state: $c(s) = |\{g \in G : s \notin g\}|$[2]. The cost function of policy μ may be defined as

$$F^{\mu}(s) = E_{\mu}\Big[\sum_{t=0}^{\infty} \gamma^t c(s(t)) | s(0) = s\Big]. \quad (2)$$

The objective of GDP is to find a policy that minimizes the cost of each state according to Eq. (2). However, it is easy to see, that defining the cost of taking action a in state $s = s_i$ to be $c_i(a) = \sum_{j=1}^{n} p_{ij}(a)c(s_j)$, results, that for any given policy μ, equation $f^{\mu} = F^{\mu}$ will hold.

B. The optimal equation, greedy policies

A necessary and sufficient condition for a cost function f^* to be the optimal cost function is that for each state $s = s_i$ it must be true that

$$f^*(s) = \min_{a \in A}\Big[c_i(a) + \gamma \sum_{j=1}^{n} p_{ij}(a) f^*(s_j)\Big]. \quad (3)$$

This is one form of the *Bellman Optimal Equation* which can be solved for each $f^*(s)$, $s \in S$, by a dynamical programming (DP) algorithm [3]. It is a set of n (the number of states) simultaneous nonlinear equations. Let us denote the argument of function min in Eq. (3) by $Q^f(s_i, a)$: for any function $f : S \rightarrow \mathbf{R}$ and for any pair (s, a) from $S \times A$ let $Q^f(s_i, a) = c_i(a) + \gamma \sum_{j=1}^{n} p_{ij}(a) f(s_j)$. Using this function Eq. (3) may be put in the following form:

$$f^*(s) = \min_{a \in A} Q^*(s, a), \quad (4)$$

where $Q^* = Q^{f^*}$ is the optimal Q-function.

Knowledge of the optimal cost function is extremely useful: the key fact is, that the *greedy policy* underlying the optimal cost function is optimal. For any cost function f, the corresponding greedy policy is the policy μ, for which $\mu(s) = a$ if and only if $Q^f(s, a) = \min_{a' \in A} Q^f(s, a')$.

[2]Equivalently we could define the "reward" delivered to the controller when it reaches the state s to be the number of goals g, for which $s \in g$. Certainly it makes now difference.

C. Methods for finding the optimal cost function

A sutiable method for solving Eq. (3) is *Real-Time Dynamical Programming* (RTDP), which is a form of *Asynchronous Dynamic Programming* (ADP). ADP is a successive approximation method for solving the Bellman Optimal Equation. It uses a series of cost functions $f^{(k)}$ for estimating the optimal cost function f^*. For every $k = 0, 1, 2, \ldots$ let us denote by $S_k \subset S$ the set of states whose costs are *backed up* at stage k. Function $f^{(k+1)}$ is computed as follows:

$$f^{(k+1)}(s) = \begin{cases} \min_{a \in A} Q^{f^{(k)}}(s, a) & \text{if } s \in S_k; \\ f^{(k)}(s) & \text{otherwise.} \end{cases} \quad (5)$$

ADP converges to f^* provided that the cost of each state is backed up infinitely often, i.e., provided that each state is contained in an infinite number of the subsets S_k, $k = 0, 1, \ldots$. In practice, this means that whatever strategy is used to select states whose costs are to backed up, no state should ever be completely barred from selection in the future. RTDP interleaves ADP and real-time control under the following conditions: First the controller always follows a policy that is greedy with resprect to the most recent estimate of f^*. Second, the set of states, whose costs are backed up during interval t, always contains $s(t)$. There may be more then one back up stages of ADP during any interval. In the simplest case of RTDP there is only one cost back up stage during any interval, and in this sweep only the cost of state $s(t)$ is backed up. More generally, in addition to $s(t)$ the set of backed up states B_t can contain any states generated by any method, such as an exhaustive off-line search from $s(t)$, forward to some fixed search depth.

Although these choices can greatly influence the rate at which RTDP converges, it should be clear, that if it is guaranteed, that the system reaches all states infinitely many times, then this process converges to the optimal cost function, and thus the policy of controller converges to the optimal policy. There are a lot of different approaches ensuring this condition, e.g. assuming that the Markov process resulting from the use of any policy is ergodic, or using trials [3].

III. Activation spreading: Route to event-based reinforcement learning

DCM dymanically builds a doubly weighted, directed graph $G = (V, E, w, p)$. Nodes of the graph are (elementary) events of the world, i.e. elements of the set $S \times A \times S$[3]. The set of possible connections is restricted to the connections of type (v_1, v_2), where

[3]An element of $S \times A \times S$ is refered as event, since it may be viewed as the state-transition of the system resulted from perfoming an action.

$v_1 = (s, a, s')$ and $v_2 = (s', a', s'')$. Denote the set of all possible edged by E_p. The function w is a weighting on the edges, while the function v is a weighting on the nodes. The weighting w on the edge (v_1, v_2) corresponds to the relative frequency of the occurence of v_2, when v_1 has just occured. This weighting is useful for operator fusion but has no relevance here. The weighting p is relevant here as it corresponds to the probability $p_{ij}(a)$, i.e. for node $v = (s_i, a, s_j)$ $p(v) \rightarrow p_{ij}(a)$ as learning proceeds. The graph G is called to be the *internal representation* of the external world, and the process, that builds and reduces this graph may be viewed as a system identification process. Details of this process is omitted, the interested reader is referred to [1]. From now on, for the sake of simplicity we will assume, that the graph G is full (i.e. $V = S \times A \times S$, and $E = E_p$) and the weighting p is accurate.

Now, let us introduce a new real-valued function, which is defined over the nodes of graph G. Let us denote it by $R : V \rightarrow \mathbf{R}$. For the sake of simplicity from now on we will feel free to refer to the R-value of node $v = (s_i, a, s_j)$ as $R_{ij}(a)$. Function R tells the cost of events, and is defined in the terms of the cost function over the set of states: Let f be an arbitrary cost function. Now let

$$R_{ij}(a) = p_{ij}(a)(\gamma f(s_j) + c_i(a)). \qquad (6)$$

Note, that all information contained in the definition of $R(v)$ is *locally* available to the node v. Clearly

$$Q(s_i, a) = \sum_{j=1}^{n} R_{ij}(a), \qquad (7)$$

since $\sum_{j=1}^{n} p_{ij}(a) = 1$. According to this, a function $R : V \rightarrow \mathbf{R}$ is said to be optimal, if the policy, which is greedy to the Q-values defined by Eq. (7), is an optimal policy.

A. The Bellman Optimal Equation for event costs

Now, we will deduce the Bellmann Optimal Equation corresponding the function R. In order to do this, let us assume, that function f is the optimal cost function ($f = f^*$). Then for f Eq. (4) holds, and thus, according to Eq. (7)

$$f(s_j) = \min_{a \in A} \sum_k R_{jk}(a). \qquad (8)$$

Since Eq. (6) is linear in $f(s_j)$, we can rewrite it for $f(s_j)$. Putting this together with Eq. (8) yields to:

$$(R_{ij}(a)/p_{ij}(a) - c_i(a)) = \min_{a \in A} \sum_k R_{jk}(a), \qquad (9)$$

which may also be written in the form

$$R_{ij}(a) = p_{ij}(a)\Big(\gamma \min_{a \in A}\{\sum_k R_{jk}(a)\} + c_i(a)\Big). \quad (10)$$

This is the Bellman Optimal Equation for the R values. It is clear, that if a function $R : V \rightarrow \mathbf{R}$ satisfies (10), then the corresponding f function (defined by Eq. (6)) satisfies Eq. (3), and thus is the optimal cost function, which ensures, that R is also optimal. The reverse of this statement is also true: if R is optimal, then Eq. (10) holds for it.

B. RTDP as activation spreading

Eq. (10) has the same form of Eq. (3). This means, that every method which can be used for solving Eq. (3) may be used for solving Eq. (10) too. For our purposes rewrite Eq. (5) for R using Eq. (6) and (7). We get:

$$R_{ij}^{(k+1)}(a) = \begin{cases} p_{ij}(a)\Big(c_i(a) + \\ \gamma \min_{a \in A} Q^{(k)}(s_j, a)\Big) & \text{if } s \in S_k; \\ R_{ij}^{(k)}(a) & \text{otherwise.} \end{cases} \qquad (11)$$

Note, that all information needed for updating the R-value of any node is locally available in the graph, thus we can think of Eq. (11) as a special activation spreading model: The activation of node v is its R-value, and this is memorized between successful spreading trials. In more details the updating of the R-value of node v is as follows. For the sake of definiteness let us fix a node v. We equip node v with a buffer of size $|A|$, in which it can temporally store the appropriate Q-values. Now, first for any action $a \in A$, the node v will compute these Q-values: for this it receives from it's successors their R-values, and adds this value to the appropriate Q-value. Second it chooses the minimum of the computed Q-values, and then finally it accomplishes the calculations by doing the multiplications and the adding. From now on, the successive generation of the R-values will be referred as R-learning.

Now let us analyze the computational and memory requirements of the algorithm. If there are m actions, then the backing up of node v requieres $O(|\text{succ}(v)| + m)$ operations, where $\text{succ}(v)$ denotes the successors of v in graph G. Note, that $|\text{succ}(v)|$ cannot be greater then nm, since the set of possible edges is restricted to E_p. This means, that in the worst case the number of operations required for backing up the cost of a node is $O(nm)$. Assuming $m < n$, the operation requirement reduces to $O(n)$. In DP backing up a states cost requires $O(mn)$ operations. However backing up the cost of a state is a more extensive process then backing up the cost of a node, since the first is the same as backing up the costs of nodes that has common initial state. Thus the calculation requierements of DP seems to be slightly smaller[4].

Turning to the memory requirements of the process, it is apparent, that it needs a space of $O(mn^2)$ size (for storing the complete node set of the graph). This is the same value as for DP, where each state transition probability has to be stored. The Q-learning method [4] has smaller worst case memory requierements (namely it requires $O(mn)$ space for storing the Q-values), since it uses an indirect method for system identification. However, Q-learning seems to be a bit coarser and slower in convergence, then DP [3]. As Kaebling mentions, one could expect that keeping track of more distinctions the precision of cost function estimates increases [5]. Considering, that the most distinction is reached by the R-values one may hope that R-learning will be the finest method. But this has a price. Namely, for the whole graph the space needed is greater then $O(mn^2)$. Namely, if the avarage number of the number of successors of any node is s, then the space needed for the graph (with its edges) is $O(smn^2)$. It is always true, that $s \leq nm$ (since the set of possible edges is restricted to E_p), thus in the worst case, the memory requirement of the algorithm is $O(m^2n^3)$. However, in the case when edges are stored too we can use them in the cost backing up computations. In this case the number of operations required for backing up the cost of a node is only $O(s)$, in the avarage case. Note, that it may be much smaller then $O(mn)$, required by DP. The memory requirements and the number of operations required may be strongly reduced by using a well designed graph updating algorithm, like that of [1]. Such an algorithm has to ensure, that (i) events, that occur with probablity greater than zero while using the optimal policy, are included in the graph and will remain in it, and (ii) edges, whose accurate weight w is nonzero are also included in the graph, and will remain in it. A similar method ofr reducing memory requierements, is that of Korf's LRTA* algorithm [6] or the trial-based RTDP framework of [3].

Earlier it was mentioned, that details of the backing up process may help to speed up the convergence of the cost function estimates. Here we argue, that not only the speed of the system, but the behavior may be improved using a well designed backing up method.

C. A special back up method

In this specially designed back up method the backing up of costs starts from goals. Let the systems state be $s = s(t)$, and collect the set of goals $G(t)$, which are not satisfied in state s, and thus have to be satisfied: $G(t) = \{g \in G : s(t) \notin g\}$. At first, the cost of nodes in $G(t)$ and the cost of $s(t)$ are backed up[5]. Now let $S_0 = G(t)$ and let us assume that for $l \leq k$ S_l is defined. Now let

$$S_{k+1} = \{v \in V : \exists v' \in S_k, v' \in \mathrm{succ}(v)\}.$$

At stage $k+1$ the cost of nodes from S_{k+1} are backed up. The process ends at time out, or when $s(t) \in S_k$, or when the set S_k is empty. This process may be viewed as activation spreading: the source of activation is the current goal set $G(t)$, and the activation is spreaded back on the edges of the graph. The resulting behavior is clearly intentional as it backs up the costs from the goals to the systems current state, thus it uses the most recent costs values of goals. This process has an another advantage: it results in a faster convergence near to frequent goals, since the process always starts with the backing up of the costs of the goals[6]. So the resulting behavior will be more accurate for the important (frequent) goals.

IV. Conclusion

It was shown, that a special case of Dynamic Concept Model is equivalent to reinforcement learning and thus it is capable of learning the optimal policy in a probabilistic world. The dynamic nature of DCM allows the dropping of useless information and thus reduces memory requirements. In DCM it is a crucial question how to keep the important information. From the point of view of memory requirement, the worst case is when there is no information selection. In this case DCM has approximately the same memory requirement as RL. Furthermore DCM preserves information, that allows generalization, that increases information density and thus decreases further memory requirements, in a natural fashion [2]. DCM works in a similar fashion in the generalized state space, that allows to make the connection to reinforcement learning for the case of probabilistic worlds. One may ask, if the policy of the new RL is optimal. Turning the question back, one can ask, how to define generalization that keeps optimal policy.

References

[1] C. Szepesvári and A. Lőrincz, "Behavior of an adaptive AI–ANN system working with cues and competing concepts," 1993. in press.

[2] Zs.Kalmár, C. Szepesvári, and A.Lőrincz, "Generalization in an autonomus agent," in *Proceedings of ICNN'94*, 1994.

[4] Allowing, however, the use of a Q-value table can spped up the procedure to the same speed of the DP.

[5] this ensures that the cost of the current state is always backed up

[6] Here we assumed the using of a trial-based learning scheme.

[3] A. Barto, S. Bradtke, and S. Singh, "Real-time learning and control using asynchronous dynamic programming," Technical Report 91-57, Computer Science Department, University of Massachusetts, 1991.

[4] C. Watkins, *Learning from Delayed Rewards.* PhD thesis, King's College, Cambridge, 1990.

[5] D. Chapman and L. Kaelbling, "Learning from delayed reinforcement in a complex domain," TR-90-11, Teleos Research, 1990.

[6] R. Korf, "Real-time heuristic search," *Artificial Intelligence*, vol. 42, pp. 189–211, 1990.