A Real Time Learning Agent

Gabriela Şerban

Department of Computer Science
"Babeş-Bolyai" University, Cluj-Napoca, Romania
gabis@cs.ubbcluj.ro

Abstract. It is well known that search has a long and distinguished history in Artificial Intelligence, since all Artificial Intelligence problems require some sort of searching [7]. Search algorithms are useful for problem solving by intelligent (single or multiple) agents. In this paper we propose an original algorithm which extends the Learning Real-Time A* (LRTA*) algorithm [1], used for solving path-finding problems. This algorithm preserves the characteristic of LRTA* (a *real-time* search algorithm), providing a better exploration of the search space. Using this algorithm, we design an Agent for solving a path-finding problem (searching a maze).

Keywords: Search, Agents, Learning.

1 Introduction

One class of problems addressed by search algorithms is the class of path-finding problems. Given a set of states (configurations), an initial state and a goal (final) state, the objective in a path-finding problem is to find a path (sequence of moves) from an initial configuration to a goal configuration.

In single-agent problem solving, the question is [7] that an agent is assumed to have *limited rationality*, so, the computational ability of an agent is usually limited. Therefore, the agent must do a limited amount of computations using only partial information on the problem.

The A* algorithm ([2]), a standard search algorithm, extends the wavefront of explored states from the initial state and chooses the most promising state within the whole wavefront. In this case, at each step, the global knowledge of the problem is required, that is why the computational complexity is considerable. So, the task is to solve the problem by accumulating local computations for each node in the graph (the search problem). These local computations can be executed concurrently (the execution order can be arbitrary), so, the problem could be solved both by single and multiple agents.

2 Path-Finding Problem

A path-finding problem consists of the following components [7]:

- a set of nodes, each representing a state;
- a set of directed links, each representing an operator available to a problem solving agent (each link is weighted with a positive number representing the cost of applying the operator - called *distance*);
- a unique node called the *start node*;
- a set of nodes, each of which represents a goal state.

We call the nodes that have directed links from node i *neighbors* of node i.

The problem is to find a path from the initial state to a goal state. In the followings we will refer to the problem of finding an optimal (shortest) path from the initial state to a goal state (we call the shortest path the path having the shortest distance to goal).

Notational conventions used in the followings are:

- *h(s)* - the shortest distance from node s to goal nodes;
- *h'(s)* - the estimated distance from node s to goal nodes;
- *k(s,s')* - the distance (cost of the link) between s and s'.

3 Learning Real-Time A*

When only one agent is solving a path-finding problem, it is not always possible to perform local computations for all nodes (for example, autonomous robots may not have enough time for planning and should interleave planning and execution). That is why the agent must selectively execute the computations for certain nodes. The problem is which node should choose the agent.

A way is to choose the current node were the agent is located. The agent updates the *h'* value of the current node, and then moves to the best neighboring node. This procedure is repeated until the agent reaches a goal state. The method is called the Learning Real-Time A* algorithm [1].

The algorithm is described in Figure 1.

---

1. Calculate $f(j) = k(i,j) + h'(j)$ for each neighbor j of the current node i
2. **Update:** Update the estimate of node i as follows:
$$h'(i) := min_j f(j) \tag{1}$$
3. **Action selection:** Move to the neighbor j that has the minimum $f(j)$ value.

Fig. 1. The Learning Real-Time A* algorithm.

One characteristic of the algorithm is that the agent determines the next action in a constant time. That is why this algorithm is called an *on-line*, *real-time* search algorithm.

The function that gives the initial values of h' is called a *heuristic function*. A heuristic function is called *admissible* if it never overestimates (in the worst case, the condition could be satisfied by setting all estimates to 0).

In LRTA*, the updating procedures are performed only for the nodes that the agent actually visits. The following characteristic is known [1]:

- In a finite number of nodes with positive link costs, in which there exists a path from every node to a goal node, and starting with non-negative admissible initial estimates, LRTA* is *complete*, i.e., it will eventually reach a goal node.

Since LRTA* never overestimates [7], it *learns* the optimal solution through repeated trials. In this case, the values learned by LRTA* will eventually converge to their actual distances along every optimal path to the goal node.

4 A Real-Time Learning Algorithm (RTL)

In fact, the behavior of the agent in the given environment can be seen as a Markov decision process. Regarding LRTA* there are two problems:

1. in order to avoid recursion in cyclic graphs, it should be retained the nodes that have been already visited (with the corresponding values of *h'*). There-

fore, the space complexity grows with the total number of states in the search space;
2. what happens in some *plateau* situations - states in which, let us say, exists more successor (neighbor) states with the same minimum value for *h'* (the choice of the next action is nondeterministic).

In the followings, we propose an algorithm (RTL) which is an extension of the LRTA* algorithm, having some alternatives of solving the above presented problems. We mention that the algorithm preserves the completeness of LRTA*.

The proposed solutions for the problems (1) and (2) are:

1. we keep a track of the visited nodes, but we do not retain the values of *h'* for each node;
2. in order to choose the next action in a given state, the agent determines the set of states S (which were not visited by the agent) having a minimum value for *h'*. If S is empty, the training fails, otherwise, the agent chooses a random state from S as a successor state (this allows a better exploration of the search space).

The idea of the algorithm (based on LRTA*) is the following:

– through repeated trials (training episodes), the agent tries some paths (possible optimal) to a goal state, and retains the shortest one;
– the number of trials is selected by the user;
– after a training trial there are two possibilities:
  • the agent reaches a goal state; in this case the agent retains the path and it's cost;
  • the learning process fails (the agent does not reach the final state, because it was blocked).
– for avoiding cycles in the search space, the agent will not choose a state that was visited before, only if it has a single alternative (it was blocked) and it must return to the formerly visited state.

We make the following notations and assumptions:

– $S = \{s_1, \cdots, s_n\}$ - the set of states;
– $si \in S$ - the initial state;
– G - the set of goal states;
– $A = \{a_1, \cdots, a_m\}$ - the set of actions that could be executed by the agent;
– we assume that the state transitions are deterministic - a given action in a given state transitions to a single successor state (the Markov Model is not hidden [8]);
– with the former assumption, the transitions between states (and their costs') could be retained as a function $env : S\text{x}A\text{x}N \rightarrow S$ - if $s, s' \in S$, $a \in A$ and $c \in N$ so that if the agent takes the action a in the state s he reaches the state s' with the cost c, then $s' = env(s, a, c)$;
– we will say that the state s' is the *neighbor* of the state s iff $\exists a \in A$ and $c \in N$ so that $s' = env(s, a, c)$;

- *h'(s)* - the estimated distance from state s to a goal node;
- we will say that the *cost* of the path $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \xrightarrow{a_{k-1}} s_k$ is $C = \sum_{i=1}^{k-1} c_i$, where $s_{i+1} = env(s_i, a_i, c_i)$ for all $i = 1, \cdots, k-1$.

The algorithm

The algorithm consists in a repeated update of the estimated values of the states, until the agent reaches a goal state (in fact a training sequence). The training is repeated for a given number of trials.

The algorithm is shown in Figure 2.

```
Repeat until the number of trials was exceeded or until the correct solution was found

– Training:
  1. Initialization:
     • $sc$ (the current state):= $si$ (the initial state)
     • calculate the estimation of the current state $h'(sc)$
  2. Iteration:
     Repeat until ($sc \in G$) or (the agent was blocked) or (the number of visited
     states exceeds a maximum value)
     (a) Update:
         • for each state $s'$ neighbor of $sc$ the agent calculates the estimation of
           the shortest distance from $s'$ to a goal state

               $f(s') = c + h'(s'), s' = env(s, a, c)$          (2)

         • the agent determines the set of states $M = \{s_1^", \cdots s_k^"\}$ so that for all
           $j = 1, \cdots, k$

               $s_j^" = argmin_{s'} \{f(s') \mid \exists a \in A, c \in N \; so \; that \; s' = env(s, a, c)\}$     (3)

     (b) Action selection:
         • if $k = 1$ (the agent has a single alternative to continue) then the agent
           moves in the state $s_1^"$;
         • otherwise the agent determines from the set $M$ a subset $M'$ of states
           that were not visited in the current training sequence and chooses
           randomly a state from $M'$.
```

Fig. 2. The Real-Time Learning (RTL) algorithm.

We have to mention that:

- we considered that if the agent finds in several trials the same optimal solution, then it is very probable that the solution is the correct one, and the training process stops;
- the training process during one trial is a linear one;
- the agent determines the next action in a real-time;

- the space complexity is reduced (there are retained only the states from the optimal path).

As in the LRTA* algorithm, if the heuristic function (the initial values of *h'*) is *admissible* (never overestimates the true value -$h'(s) <= h(s)$ for all $s \in S$-), then we can easily prove that the RTL algorithm is complete, i.e, it will eventually reach the goal [4] and $h'(s)$ will eventually converge to the true value $h(s)$ [6].

5 An Agent for Searching a Maze

5.1 General Presentation

The application is written in Borland C and implements the behavior of an Intelligent Agent (a robotic agent), whose purpose is coming out from a maze on a shortest path, using the algorithm described in the previous section (RTL).

We assume that:

- the maze has a rectangular form; in some positions there are obstacles; the agent starts in a given state and it tries to reach a final (goal) state, avoiding the obstacles;
- in a certain position on the maze the agent could move in four directions: north, south, east, west (there are four possible actions);
- the cost of executing an action (move in one direction) is 1;
- as a heuristic function (initial values for $h'(s)$) we have chosen the Manhattan distance to the goal (it is obvious that this heuristic function is admissible), which assures the completeness of the algorithm.

In fact it is a kind of semi-supervised learning, because the agent starts with an initial knowledge (the heuristic function) , so it has an informed behavior). In the worst case, if the values of the heuristic function are 0, then the learning is unsupervised, but the behavior of the agent becomes uninformed.

5.2 The Agent's Design

For implementing the algorithm, we will represent the following structures:

- a *State* from the environment;
- the *Environment* (as a linked list of *States*);
- a *Node* from the optimal path (the current *State* and the estimation *h'* of the current state);
- the optimal path from a training sequence (as a linked list of *Nodes*).

The basis classes used for implementing the agent's behavior are the followings:

- **IElement**: defines an interface for an element. This is an abstract class having two pure virtual methods:

  - for converting the member data of an element into a string;
  - a destructor for the member data.
- **CNode**: defines the structure of a *Node* from the optimal path. This class implements (inherits) the interface *IElement*, having (besides the methods from the interface) it's own methods for:
  - setting components (the current state, the estimation of the current state);
  - accessing components.
- **CState**: defines the structure of a *State* from the environment. This class implements (inherits) the interface *IElement*, having (besides the methods from the interface) it's own methods for:
  - setting components (the current position on the maze, the value of a state);
  - accessing components;
  - calculating the estimation *h'* of the state;
  - verifying if the state is accessible (contains or not an obstacle).
- **CList**: defines the structure of a linked list, with a generic element (a pointer to *IElement*) as information of the nodes. The main methods of the class are for:
  - adding elements;
  - accessing elements;
  - updating elements.
- **CEnvironment**: defines the structure of the agent's environment (it depends on the concrete problem - in our example the environment is a rectangular maze). The private member data of this class are:
  - **m**: the environment, represented as a linked list (*CList*) of states (*CState*);
  - **si**: the initial state of the agent (is a *CState*);
  - **sf**: the final state from the environment (is a *CState*);
  - **l**, **c**: the dimensions of the environment (number of rows and columns).

  The main methods of the class are for:
  - reading the environment from an input stream;
  - setting and accessing components;
  - verifying the neighborhood of two states in the environment.
- **Agent**: the main class of the application, which implements the agent's behavior and the learning algorithm.
  The private member data of this class are:
  - **m**: the agent's environment (is a *CEnvironment*);
  - **l**: the list of *Nodes* used for retaining the optimal path in the current training sequence (is a *CList*);

  The public methods of the agent are the followings:
  - **readEnvironment**: reads the information about the environment from an input stream ;
  - **writeEnvironment**: writes the information about the environment in an output stream ;
  - **learning**: is the main method of the agent; implements the RTL algorithm.

Besides the public methods, the agent has some private methods used in the method **learning**.

We notice that all the representations of data structures are linked, which means that there are no limitations for the structures' length (number of states).

5.3 Experimental Results

For our experiment, we considered the environment shown in Figure 3. The state marked with 1 represents the initial state of the agent, the state marked with 2 represents the final state and the states filled with black contains obstacles (which the agent should avoid).

We repeat the experiment four times, because of the random character of the action selection mechanism. The results after the experiments are shown in Table 1, 2, 3, 4 (in a solution the agent determines the moving direction from the current state).

We notice that, in average, after 8 episodes, the agent finds the optimal path to the final state.

The graphical representation of the training during the four experiments is shown in Figure 4.

Table 1. First experiment

Number of episodes	8
The optimal solution	East North North East North North East East East North
Episode	Number of steps until the final state was reached
1	10
2	16
3	10
4	10
5	18
6	12
7	14
8	10

6 Conclusions and Further Work

The algorithm described in this paper is very general, could be applied in any problem which goal is to find an optimal solution in a search space (a path-finding problem).

On the other hand, the application is designed in a way which allows us to model (with a few modifications) any environment and any behavior of an agent.

Further work is planned to be done in the following directions:

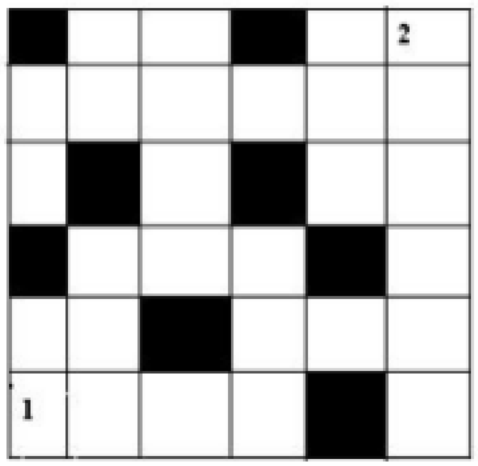

Fig. 3. The agent's environment

Table 2. Second experiment

Number of episodes	6
The optimal solution	East North North East North North East East East North
Episode	Number of steps until the final state was reached
1	16
2	10
3	14
4	10
5	10
6	10

Table 3. Third experiment

Number of episodes	14
The optimal solution	East North North East North North East East East North
Episode	Number of steps until the final state was reached
1	18
2	10
3	12
4	10
5	16
6	10
7	12
8	14
9	16
10	28
11	14
12	16
13	14
14	10

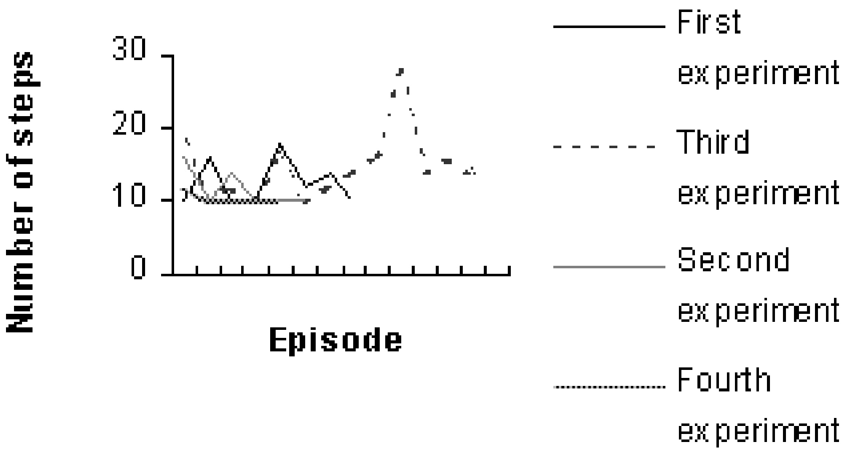

Fig. 4. The number of steps/episode during the training processes

Table 4. Fourth experiment

Number of episodes	5
The optimal solution	East East East North East East North North North North
Episode	Number of steps until the final state was reached
1	12
2	10
3	12
4	10
5	10

- to analyze what happens if the transitions between states are nondeterministic (the environment is a Hidden Markov Model [8]);
- to use probabilistic action selection mechanisms (ϵ-Greedy, SoftMax [5]);
- to combine the RTL algorithm with other classical path-finding algorithms (RTA*);
- in which way the agent could deduce the heuristic function from the interaction with it's environment (a kind of reinforcement learning);
- to develop the algorithm for solving path-finding problems with multiple agents.

References

1. Korf, R., E.: Real-time heuristic search. Artificial Intelligence (1990)
2. Korf, R., E.: Search. Encyclopedia of Artificial Intelligence, Wiley-Interscience Publication, New York (1992)
3. Russell, S.J., Norvig, P.: Artificial intelligence. A modern approach. Prentice-Hall International (1995)
4. Ishida, T., Korf, R., E.: A moving target search. A real-time search for changing goals. IEEE Transaction on Pattern Analysis and Machine Intelligence (1995)
5. Sutton, R., Barto, A., G.: Reinforcement learning. The MIT Press, Cambridge, England (1998)
6. Shimbo, M., Ishida T.: On the convergence of real-time search. Journal of Japanese Society for Artificial Intelligence (1998)
7. Weiss, G.: Multiagent systems - A Modern Approach to Distributed Artificial Intelligence, The MIT Press, Cambridge, Massachusetts, London (1999)
8. Serban, G.: Training Hidden Markov Models - a Method for Training Intelligent Agents, Proceedings of the Second International Workshop of Central and Eastern Europe on Multi-Agent Systems, Krakow, Poland (2001) 267–276