Evolutionary Tile Coding: An Automated State Abstraction Algorithm for Reinforcement Learning

Stephen Lin and **Robert Wright**
Air Force Research Laboratory Information Directorate
525 Brooks Rd.
Rome, NY 13441
Stephen.Lin@rl.af.mil and Robert.Wright@rl.af.mil

Abstract

Reinforcement learning (RL) algorithms have the ability to learn optimal policies for control problems by exploring a domain's state space. Unfortunately, for most problems the size of the state space is too great for RL technologies to fully explore in order to find good policies. State abstraction is one way of reducing the size and complexity of a domain's state space in order to enable RL. In this paper we introduce a new approach for automatically deriving state abstractions called *Evolutionary Tile Coding* that uses a genetic algorithm for deriving effective tile codings. We provide an empirical analysis of the new algorithm comparing it to another adaptive tile coding method as well as fixed tile coding. Our results show that our approach is able to automatically derive effective state abstractions for two RL benchmark problems. Additionally, we present an intriguing result that shows the classical mountain car (Justin Boyan 1995) problem's state space can be reduced to just two states and still preserve the discovery of an optimal policy.

Introduction

Technological development has been driving toward more complex systems that require faster responses to events. Processes that used to require human intervention quickly grow beyond the human operator's ability to respond. Autonomous control techniques can provide responses as fast as needed to be effective, but the increasing complexity of the tasks makes the autonomous controller difficult to design. Reinforcement learning (Sutton and Barto 1998) (RL) is one approach that can be used to automate control processes and provide rich solutions that are robust in their response to new situations.

Reinforcement learning algorithms attempt to discover an optimal policy for a given domain by exploring its state space. The optimal policy, π^*, is a prescriptive function that determines the appropriate action to take in any given state that will result in the maximum aggregate reward. RL algorithms learn π^* on-line by trying various actions in the states it experiences and observing the rewards it receives. A RL algorithm must experience the consequences, r, of attempting a particular action, a, in a given state, s, a number of times before it can accurately estimate the value of taking a in s otherwise known as the Q-value, $Q(s, a)$.

A significant issue that has hindered the application of RL algorithms is the state space problem also known as the *curse of dimensionality* (Bellman 1961). Simply put, there are too many states in a problem's state space to experience and learn over for most domains. The state of a problem is defined by the features of the domain as well as the number of values the features can take. Every unique combination of feature values, if interpreted literally, can represent a unique state. Adding a new feature to the representation of the state increases the size and complexity of the state space exponentially. A means for reducing the size and complexity of state spaces is necessary for RL.

Fortunately, it has been found that in most domains much of the information that describes the state is redundant or irrelevant with regards to solving the problem. State abstraction methods have been developed to simplify state spaces by making generalizations that remove redundancies and hide unnecessary information (Sutton 1996; Li, Walsh, and Littman 2006; Wright and Gemelli 2009). In this paper we focus on one particular state abstraction approach known as tile coding (Sutton and Barto 1998).

Tile coding is a form of state abstraction for domains with continuous states spaces. It discretizes the state space into *tiles* that cover ranges of values for each feature in the state space. Every state that falls under a specific tile is treated as the same abstract state and the RL algorithm learns over the abstract state space. The effectiveness of tile coding methods depends heavily on the design of the tiling scheme. If there is insufficient resolution in a particular area of the state space the RL algorithm will not be able to find π^*. As a result the design and implementation of tile coding schemes has been a manual and time consuming process that requires significant domain expertise to be effective.

Recently, there has been work in automated tiling methods that attempt to derive an effective tiling scheme on-line (Uther and Veloso 1998; Whiteson, Taylor, and Stone 2007). In this paper we introduce a new automated tile coding algorithm called Evolutionary Tile Coding (EvoTC). EvoTC uses a genetic algorithm to derive efficient tile structures that maximizes an RL algorithm's ability to find a good policy. We compare the performance of EvoTC to competing fixed and automated tile coding methods, CMAC and Adaptive

Tile Coding. And we show that EvoTC is able to provide more efficient tile based state abstractions that should help enable RL algorithms to scale towards more complex problems.

The rest of the paper proceeds as follows. In the next section we provide background and details on the two tile coding approaches we use for comparison. We then introduce and describe EvoTC in detail. This is followed by a description of our experimental setup and results. We conclude with a discussion of the results and a summary of the conclusions we were able to make.

Background

Cerebellar Model Articulation Controller

The Cerebellar Model Articulation Controller algorithm, better known as CMAC, was introduced in (W.T., F.H., and L.G. 1990) (then called the Cerebellar Model Arithmetic Computer) as a means of providing local generalization of the state space based on how the human brain is thought to respond to stimuli (Albus 1971). This behavior allows states that are in proximity to an observed state to learn even though those states have not been observed themselves. It was chosen for our analysis because it is arguably the most popular of the tile coding methods (Sutton 1996).

CMAC partitions state spaces into a fixed set of non-overlapping tiles. Q-values that are learned from any one state in a tile are learned for all states in the tile. Partitioning the state space into many small tiles will slow learning but will improve the probability of finding optimal policies. Conversely, if the tiles are very large then Q-values will be distributed quickly across many states, but there is no guarantee that two states on opposite sides of a tile should share the same action values. In this case, each state may favor a different action, but only one action can be preferred per tile, preventing a correct policy from being found. This tradeoff is mitigated by overlapping layers of tiles to provide both coarse and fine grain generalization. Each observed state updates one tile per layer, and each of these tiles covers a different portion of the state space. The preferred action for a state is the action that maximizes the weighted sum of action values across all tiles that contain that state.

The CMAC algorithm has effectively learned a number of domains including the mountain car and single pole balance (Sutton 1996). More recently, it has been shown to suffer from some limitations on slightly more complicated problems like the double pole balance (Gomez, Schmidhuber, and Miikkulainen 2006). The main difficulty in applying CMAC is choosing a suitable way to break up the state space into tiles. If this is done inexpertly then states that do not prefer the same action can be forced to learn together if they are both confined to a single tile. This will severely slow down, if not prevent, the learning of a successful policy. A secondary concern is the memory requirements for high dimensional scenarios. The number of tiles per mapping scales exponentially in the number of input perceptions for a problem, and storing all visited tiles can quickly become unreasonable. Hashing techniques like those mentioned in (W.T., F.H., and L.G. 1990) can be used to place limits on the memory requirements of CMAC, but they effectively cause non-local generalization of learned values in the event of a hash collision, which can negatively impact policy convergence.

Adaptive Tile Coding

Adaptive tile coding (ATC) (Whiteson, Taylor, and Stone 2007) is a tile coding algorithm that automatically derives variable resolution state abstraction while learning a policy for a specific problem. It is similar to the continuous U-Tree algorithm discussed in (Uther and Veloso 1998). Both methods derive abstractions by starting with a single tile that encompasses the entire state space. Based on observations made while an RL algorithm attempts to learn over the abstract state space, "splits" are introduced. Splits divide individual tiles evenly along feature dimensions into two new abstract states. The idea is to increase the resolution only in areas of the state space where changes in action choices should be made. Splitting continues until the RL algorithm is able to solve the problem using the derived abstract state space. Determining when and where to split tiles is the only significant difference between these methods. Heuristics is used for ATC (Whiteson, Taylor, and Stone 2007) and a statistical method is used for continuous U-tree (Uther and Veloso 1998).

ATC uses two heuristics to determine first when to split and then where to split. The first heuristic keeps track of the lowest Bellman error per time step. If the lowest Bellman update fails to change for a specified consecutive number of updates, *split threshold*, then the heuristic has determined learning has stopped and it is appropriate to split a tile. Once it has been determined that it is appropriate to split the *policy criterion* heuristic determines where to split. The ATC algorithm updates the Q-values for all potential tiles in the tilings. Every time a potential tile within the current activated tile prescribes a differing action from the activated tile it updates a counter for the potential tile. ATC splits the tile with the potential tile that has the highest counter value to establish that potential tile in the tiling. This process increases the resolution of the tiling in areas where a changes in policy are likely.

In (Whiteson, Taylor, and Stone 2007) it was shown the ATC has a number of advantages over CMAC. First, the tilings are derived automatically eliminating the need to manually design and discover an effective tiling. Second, ATC was found to be faster at finding π^* than CMAC using the best found parameters for the number of tiles and tilings. The reason for the improvement is that the RL algorithm benefits from the generalization of the overly abstract state space early in the learning. As the abstract state space becomes more specific the new states are already partially learned because they retain the values learned from the more general state they were split from.

Although this approach is an improvement over fixed tile coding methods like CMAC, it suffers from a significant drawback. This approach splits the tiles in half evenly. It is highly unlikely that such a split will be positioned exactly where there is a decision point in which taking one action should be preferred over another. These methods can make

up for this by successively splitting sub-tiles until the decision point is reached. However, many unnecessary states are introduced by doing this and it will slow the RL.

Evolutionary Tile Coding

Evolutionary Tile Coding (EvoTC) is a new approach that takes flexible state space arrangement even further. Like the other adaptive tile coding approaches it starts with a single tile that encompasses the entire state space and introduces splits to increase the detail of the abstraction. The major differences are that EvoTC uses an evolutionary algorithm (Holland 1992) to determine when and where to place the splits and the splits can divide tiles unevenly. By dividing tiles unevenly EvoTC should be able to derive more efficient and effective tiling abstractions than other existing tile coding approaches.

In ATC and continuous U-tree the splits are placed in the center of tiles because it is difficult to determine exactly where the optimum split should be made. So, instead they hone in on the correct position by adding additional splits. The additional splits are unnecessary and slow learning. EvoTC is able to find better split positions by framing the problem of finding the optimal position and number of splits as an optimization problem where the performance of the RL algorithm is optimized.

EvoTC starts with an initial population of tilings to be evaluated. Each tiling is evaluated independently by pairing it with an RL algorithm that attempts to solve a problem using the tiling as a state abstraction device. The performance of the RL algorithm is considered the fitness of the tiling. Tilings that are more effective at abstracting the state space should enable the RL algorithm to perform better. After all members of the population are evaluated the fittest tilings are kept for successive generations. New tilings based on the fittest members of the previous generation are also introduced into the population for the next generation. The new tilings are generated by applying mutation operators, described later, to the current fittest members of the population. The new population is then evaluated in the same manner the previous generation was. Over the course of generations a tiling should be produced that will enable the RL algorithm to exceed a specified performance threshold and the algorithm terminates.

In the following we provide details on how the tilings are represented in the evolutionary algorithm and how the mutation operators function:

Genetic Representation of Tiles Each chromosome in EvoTC represents a single unique tiling. The chromosomes hold a tile arrangement described as a binary decision tree. The genes that make up the chromosome describe the nodes in the tree. Leaf nodes represent a current tile and hold the Q-values associated with the abstract state. Non-leaf nodes represent tile divisions and describe along what feature the division is made and its position. See figure 1 for an illustration of how the tiling is represented as a tree. The genetic representation is non-fixed to enable the tiling to become more complex as needed. The process of how the chromosome is extended is described in the *divide* mutation operator description.

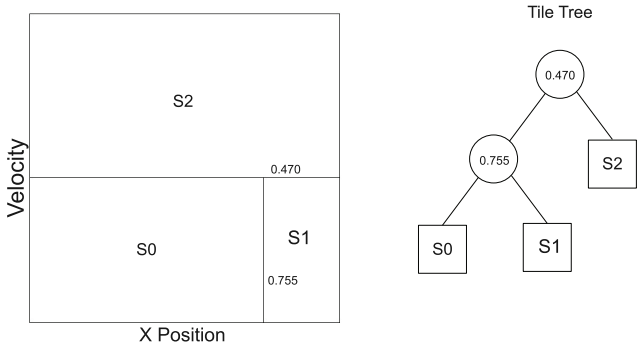

Figure 1: This figure illustrates how EvoTC represents the tile discretizations of a state space as a binary decision tree. Left: shows a sample discretization of the two dimensional state space of the mountain car problem. Right: shows the corresponding decision tree which is used to lookup the Q-values associated with the individual tiles.

Mutation Operators The key to the EvoTC is its mutation operators which make diverse tile arrangements in the search for the optimal arrangement. These mutation operators are applied, with a specified probability, to existing chromosomes in the population to make new chromosomes at the end of each generation. Two mutation operators are used in this algorithm:

- The *shift* operator moves the position of tile splits. The purpose of this mutation operator is to explore the ability of the existing tiling arrangement to properly abstract the state space. As such this operator should be activated with a higher probability than the *divide* operator which changes the structure of the tiling arrangement.

  When this mutation operator is activated it selects a number of division nodes to be modified at random. For each selected node, the position of the divide is shifted by a small amount determined by a gaussian random distribution up to within 1% of the edge of the tile. This prevents a pair of adjacent tiles from effectively becoming one tile if one of the tiles holds 0% of the state space. After the selected genes are altered, the tree is updated with the mutated genes.
- The *divide* operator introduces new splits to the tiling to add granularity to the abstract state space. It should have a relatively low probability of being activated to give the *shift* operator sufficient time to explore more general tilings.

  This operator functions by selecting a single leaf node at random to divide. The node is divided by randomly selecting a dimension to divide along and the division is placed using a random gaussian distribution over the center of the tile. Once the divide is set, new leaf nodes, and genes are created and attached to the new divide node. Finally, the Q-values for the new leaf nodes are initialized to a value that encourages exploration of the new tiles.

Experiments

We conducted an empirical comparison of CMAC, ATC, and EvoTC on two well known RL benchmark problems with continuous state spaces. The purpose of these algorithms is to reduce the size and complexity of domains' state spaces and enable a RL algorithm to discover an optimal policy for problems in those domains. We measure the effectiveness of the approaches by the number of states in the abstract state space and by the number of learning updates required by the RL algorithm to learn an optimal policy. The fewer the number of states in abstract state space the better the method's ability to effectively abstract the state space. And, the fewer the number of updates required by the RL algorithm to learn an optimal policy the better the state abstraction.

The following is a description of the benchmark problems used and our experimental setup. It should be noted that all the methods require some parameter tweaking in order to achieve their best performance. In our comparisons we used the best found parameter settings for each method. The parameters used for each method and problem are specified below. Each method was paired with the RL algorithm SARSA (Sutton and Barto 1998) to derive policies. Also, the results shown for EvoTC are representative of the median value of 25 separate runs. Because EvoTC is dependent on a stochastic search several runs with different random seeds were necessary to properly characterize its performance.

Mountain Car

The mountain car problem is a classical control RL problem in which the learner has to derive a policy to enable an automobile to escape a deep valley. The car does not have enough power to drive up the sides of the valley starting from a standing position. To get out the driver must build up enough momentum by rocking back and forth. Two continuous features, position and velocity, specify the state. At each time step the RL algorithm has to select one of three possible actions; accelerate to the left or right, or coast. A reward signal of -1 for every time step the car has not reached the goal state is provided to encourage the discovery of a policy that reaches the goal state in as few time steps as possible.

We use a problem set of 100 different starting positions and initial velocities to represent the problem domain in our experiments. The algorithms are evaluated based on the average performance over all instances in the problem set. For our problem set an optimal policy enables the car to escape the valley in average of 50 time steps.

In our experiments for CMAC we used 2 layers of tiling with 11 tiles per feature for each layer. This allows a maximum of 242 possible unique abstract states. ATC requires the *split threshold* parameter be specified. For the mountain car problem we found a value of 521 to work well. EvoTC requires the mutation probabilities be specified. For this problem values of 32% for *shift* and 5% for *divide* per tiling per generation were used. A population size of 100 was also used for each evolutionary generation.

Pole Balance

The pole balance problem models a car balacing a long pole attached on a hinge (Barto, Sutton, and Anderson 1990). The car is free to travel on a short track to keep the pole balance vertically over the car. Failure occurs if the pole falls more than 12 degrees from vertical or if the car rolls off either end of the short track. The state is represented by 4 continuous features; the position and velocity of the car, and the angle and angular velocity of the pole. There are three available actions; accelerate to the left, to the right, and to coast.

We use a problem set of 20 different initial feature values in our experiments. The goal for the algorithms to find a policy that keeps the pole balanced for at least 10^6 updates without dropping the pole or exceeding the bounds of the track.

For CMAC, the settings of 2 layers of tilings with 11 tiles per dimension of input per layer is again selected for this test for a maximum of 29282 states. The settings selected for EvoTC are 30% for *shift* and 12% for *divide*. We were unable to successfully apply ATC to this problem.

Table 1: Results for Mountain Car

	Number of Updates	Number of States
CMAC	1.22e+05	177
ATC	1.88e+05	83
EvoTC	2.00e+07	2

Results and Discussion

The results of the mountain car and pole balance are listed in Table 1. All three methods were able to converge to an optimal policy. We can see that CMAC was able to solve the mountain car problem in the fewest number of updates. This is slightly surprising because it was shown that ATC was able to outperform CMAC on this problem in (Whiteson, Taylor, and Stone 2007). We were not able to reproduce that result. However, this result is intuitive in that the fixed CMAC tile coding was tuned for this problem and was found as a result of many trial runs. ATC and EvoTC have to learn their tile abstractions and this requires some additional time and updates.

It should be noted that EvoTC is penalized by the update metric becasue all the updates required by the failed members of the population are included. Including the aggregate updates required for all the members of the population is necessary to get an accurate measure of computation time required. However, each evaluation of a tiling per generation could be done independently in parallel, which would result in a significant speed up of this algorithm.

Table 1 also shows the size of the abstract state space required for each method. CMAC only uses 177 of the potential 242 states available. EvoTC and ATC are able to solve the mountain car using substantially smaller state spaces which shows they derive much more efficient state abstractions. This suggests that they will be able to scale more effectively as the size of the state spaces increase.

The most striking result of this experiment is that EvoTC was able to derive an optimal policy using an abstract state space consisting of only 2 states. , EvoTC was consistently able to find this state abstraction during our experimentation.

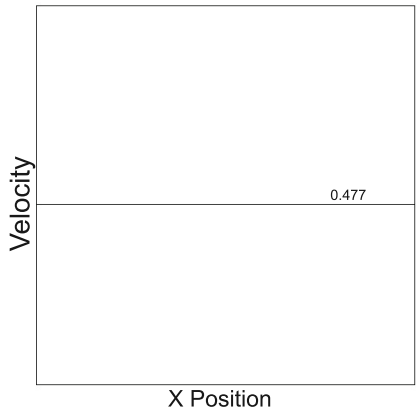

Figure 2: This figure shows how EvoTC algorithm discretized the mountain car state space

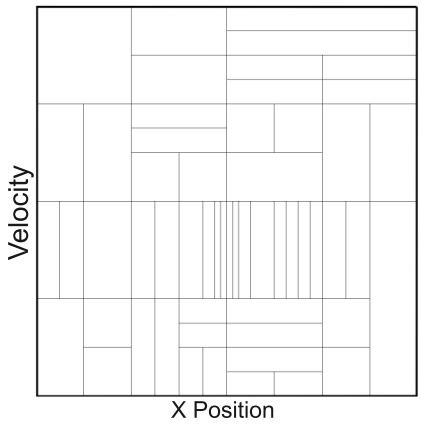

Figure 3: This figure shows how the ATC algorithm discretized the mountain car state space

The mountain car problem is one of the classic RL control problems. It is considered difficult due to its continuous state space. EvoTC simplified it to a simple two state problem which is trivial for a RL algorithm to find a policy for. Not only that, EvoTC was able to eliminate the need for an entire feature. The only split in the state space occurs at .477 of the velocity vector. There are no divisions over the position feature which means it is not relevant at all to solving the problem. This result highlights the power of automated state abstraction to find unintuitive and effective abstractions.

This experiment also shows how important the design of the abstraction can be. Figures 2 and 3 shows the abstract state spaces derived by EvoTC and ATC respectively. The abstraction derived by ATC is significantly more complex than the one derived by EvoTC and includes divisions across the position vector. ATC cannot find the same abstraction that EvoTC is able to find because it arbitrarily divides each tile evenly. As a result it had to derive a much more complex abstract state space to learn and equivalent policy.

Table 2: Results for Pole Balance

	Number of Updates	Number of States
CMAC	3.69e+08	5379
ATC	failed to converge	failed to converge
EvoTC	1.04e+09	61

Table 2 shows the results we obtained applying these methods to the pole balance problem. The pole balance problem is significantly more difficult than the mountain car problem in that it has double the number of continuous features. As such, we can see that CMAC still requires the fewest updates, but required significantly more abstract states in order to solve this problem. In our experiments we were unable to find a parameter setting that enable ATC to converge. Once again EvoTC was able to derive an abstraction with far fewer states and still allows the RL algorithm to find an optimal policy. EvoTC still required an order of magnitude more updates than CMAC, however the increase in number of updates and states required by CMAC compared to EvoTC further implies that EvoTC will scale more effectively as the size of the state space is increased.

Observations and Discussion

In our testing we found that all methods were extremely sensitive to untuned parameter settings. Slight changes to the parameter settings that work for a domain could very easily prevent these methods from converging again. This was especially true of the ATC algorithm, which required a substantial amount of trial and error to find a parameter setting that worked for the mountain car. Finding settings for CMAC and EvoTC was significantly less time consuming, but still required some trial and error.

Although EvoTC and CMAC were able to solve both benchmark problems it does not appear that either method will scale adequately as the number of features that describe the state space is increased. Both methods are tile coding based and are linear abstractions of the state space. As a result, although the abstract state spaces found by these methods are significantly smaller than the actual state space, they will still scale proportionally as the number of features is increased. It may be the case that non-linear state abstraction methods such as RL-SANE (Wright and Gemelli 2009) are necessary as the number of features are increased.

Conclusion

Real world applications have large continuous state spaces that prevent the use of RL algorithms. State abstraction methods such as tile coding are necessary in order to apply RL to non-trivial problems. Fixed tile coding algorithms such as CMAC can be effective as long as the tiling scheme is properly designed. Adaptive tile coding methods like ATC and EvoTC are appealing because they do not require manual design of the state abstraction. In this paper we introduced EvoTC and showed how it is able to abstract the state space more effectively than CMAC and ATC on two continuous state space problems.

Not only was EvoTC able to outperform CMAC and ATC in terms of abstraction power it was able to reduce the classical mountain car domain to a problem consisting of just two states. This result highlights the power and importance of automated state abstraction methods.

Although EvoTC was able to very effectively abstract the state space of the mountain car problem it does not appear that the approach will scale well as the number of features that describe the domain are increased. We believe this is due to the linear nature of the tiling abstraction. Although the tilings are gross abstractions of the state space the dimensionality of the abstract state space is the same as the original state space. In future work we will explore the derivation of non-linear state abstraction devices such as multilayered feed forward neural networks (Wright and Gemelli 2009) and examine how they scale. Non-linear state abstractions may be able to find more efficient abstractions of mutlidimensional state spaces enabling them to scale more effectively as the number of features is increased.

References

Albus, J. S. 1971. A theory of cerebellar functions. *Mathematical Biosciences* 10:25–61.

Barto, A. G.; Sutton, R. S.; and Anderson, C. W. 1990. Neuronlike adaptive elements that can solve difficult learning control problems. *Artificial neural networks: concept learning* 81–93.

Bellman, R. 1961. *Adaptive Control Processes: A Guided Tour*. Princeton University Press.

Gomez, F. J.; Schmidhuber, J.; and Miikkulainen, R. 2006. Efficient non-linear control through neuroevolution. In *ECML*, 654–662.

Holland, J. H. 1992. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press.

Justin Boyan, A. M. 1995. Generalization in reinforcement learning: Safely approximating the value function. In Lee, G. T. . D. T. . T., ed., *Neural Information Processing Systems 7*, 369–376. Cambridge, MA: The MIT Press.

Li, L.; Walsh, T. J.; and Littman, M. L. 2006. Towards a unified theory of state abstraction for mdps. In *In Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics*, 531–539.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press.

Sutton, R. 1996. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems*, volume 8, 1038–1044. MIT Press.

Uther, W. T. B., and Veloso, M. M. 1998. Tree based discretization for continuous state space reinforcement learning. In *AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, 769–774. Menlo Park, CA, USA: American Association for Artificial Intelligence.
Whiteson, S.; Taylor, M. E.; and Stone, P. 2007. Adaptive tile coding for value function approximation. Technical report, University of Texas at Austin.

Wright, R., and Gemelli, N. 2009. State aggregation for reinforcement learning using neuroevolution. In *ICAART International Conference on Agents and Artificial Intelligence*.

W.T., M. I.; F.H., G.; and L.G., K. I. 1990. Cmac: an associative neural network alternative to backpropagation. *Proceedings of the IEEE* 78(10):1561–1567.