Automated Mixed Resolution Tiling

Peter Scopes
Department of Computer Science,
University of York, UK
pds506@york.ac.uk

Daniel Kudenko
Department of Computer Science,
University of York, UK
kudenko@cs.york.ac.uk

ABSTRACT

Tile Coding (TC) is a popular and widely used form of value function approximation for Reinforcement Learning (RL). The size of tiles impacts an agent's ability to learn [10, 6]: tiles that are too large lead to performance degradation; long periods of learning causes divergence from the optimal policy; and, decreasing the exit probability of non-optimal action(s) improves performance. However, tiles that are too small do not gain any benefit from using function approximation.

Choosing a tiling that well suits an environment can require expert knowledge of TC and in depth knowledge of the environment and its transition function, neither of which may be available. Moreover, in RL it is often assumed that the transition and reward functions aren't fully known. Therefore prescribed choices for tilings are guesses at best. This motivates the creation of algorithms to determine a tiling whilst an agent is learning.

In this paper we introduce a novel algorithm, automated Mixed Resolution Tiling (AMRT), which can alter the tiles in a tiling whilst the agent is learning. We demonstrate empirically that using AMRT to alter a tiling non-uniformly during learning results in performance matching or exceeding the best fixed tiling. We also conduct an empirical study on the impact specific tile shapes can have on learning.

1. INTRODUCTION

Reinforcement Learning (RL) is a popular and widely-studied machine learning technique, where an agent learns a policy through continual interactions with the environment, based on performing actions and observing their rewards. In the basic RL formulation, in order to learn an optimal policy, an agent may need to visit each environment state and perform all possible actions in that state at least once (and often repeatedly). For this reason the speed of learning does not scale well to complex environments with large state spaces.

In order to deal with large state spaces, a popular technique employed in RL is Tile Coding (TC) [1]. In this approach, values from one or more state features are grouped into exhaustive fixed uniformtitions, called tiles. This reduces the size of the state space, and thus TC enables an agent to learn more quickly. However, with the reduction of state space granularity also comes a potential reduction in the precision and quality of the learnt policy. Moreover, in certain worst-cases learning can be severely impaired by a poor design of tiles (see for example [6] for a more formal understanding and analysis of potential TC problems).

Due to the popularity of TC in the area of RL, and because of the big impact tile design has on learning performance, the ability to design good tilings that increase the speed of learning while preserving the quality of the learned behaviour is highly important. However, tile design is a hard and not well-studied problem that often requires good knowledge and understanding of the agent environment. In cases where such a deep understanding is not available, a designer is left with trying different configurations of tiles and tilings until a satisfactory performance is reached. This is a lengthy and sometimes infeasible procedure, which could be made obsolete if an automated method for good tile design can be found. A first approach, Adaptive Tile Coding (ATC) [10], addressed this problem by starting with a few large tiles and intelligently splitting the tiles based on one of two criteria. ATC provides both further evidence that splitting tiles over time improves learning [4] and that doing so in an intelligent manner is possible and effective [7].

In previous work by Scopes and Kudenko [6] a manual method for tiling design called Mixed Resolution Tiling (MRT) was introduced. MRT is based on heuristics derived from theoretical properties of TC. Simply put, MRT recommends high resolution (small) tiles near the optimal transition path through the state space; tiles further from the optimal path should have low resolution (large) tiles, and taper from high to low resolution the further away they are from the optimal path.

In addition, one of the theoretical properties stated that increasing the probability of exiting a tile with an optimal action improves the performance of TC. However, manipulating tiles so that particular actions have a increased probability of being used on exit is hard to achieve and would need to be done a per-tile basis. One possible way of achieving this would be to alter the tile's shape.

Based on the the ideas presented above this paper performs an empirical study on the impact of specific tile shapes have on learning and then, using the conclusions of the study and ideas from MRT, proposes an automated version of the manual MRT algorithm, called Automated MRT (AMRT), which splits tiles along or near the agent's current best path from the initial tile to the goal tile whilst learning. AMRT does not require prior knowledge of the environment or transition function, rather it can devise a new tiling whilst learning in order to improve performance.

2. BACKGROUND

2.1 Reinforcement Learning (RL)

Reinforcement learning is a method where an agent learns by receiving rewards or punishments through continuous interactions with the environment [8]. The agent receives a numeric feedback relative to its actions and in time learns how to optimise its action choices. Typically reinforcement learning uses a Markov Decision Process (MDP) as a mathematical model [5].

An MDP is a tuple $\langle S, A, T, R\rangle$, where S is the state space, A is the action space, $T(s, a, s') = Pr(s'|s, a)$ is the probability that action a in state s will lead to state s', and $R(s, a, s')$ is the immediate reward r received when action a taken in state s results in a transition to state s'. The problem of solving an MDP is to find a policy (i.e., mapping from states to actions) which maximises the accumulated reward. When the environment dynamics (transition probabilities and reward function) are available, this task can be solved using dynamic programming [2].

When the environment dynamics are not available, as with most real problem domains, dynamic programming cannot be used. However, the concept of an iterative approach remains the backbone of the majority of reinforcement learning algorithms. These algorithms apply so called temporal-difference updates to propagate information about values of states, $V(s)$, or state-action pairs, $Q(s, a)$. These updates are based on the difference of the two temporally different estimates of a particular state or state-action value. The Q-Learning algorithm is such a method [9]. After each real transition, $(s, a) \rightarrow (s', r)$, in the environment, it updates state-action values by the formula:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot (r + \gamma \cdot max_a Q(s', a') - Q(s, a)) \quad (1)$$

where α is the rate of learning and γ is the discount factor. It modifies the value of taking action a in state s, when after executing this action the environment returned reward r, moved to a new state s', and if the action a' pertaining the greatest value in the new state s' was chosen.

2.2 Tile Coding (TC)

In Tile Coding (TC) [1] one or more features of the state space is exhaustively partitioned, called a *tiling*. Each partition of the tiling is called a *tile*. Only one tile from each tiling can be activated by any one state.

Figure 1 is an example of two tilings covering a 2-dimensional state space. Each dimension corresponds to a numerical feature and each point in the space corresponds to a state. As can be seen the two tiles activated in each tiling have been highlighted. The update rule is altered to reflect the use of tiles [8]:

$$\begin{aligned} Q(T(s), a) \leftarrow & Q(T(s), a)+ \\ & \alpha \cdot (r + \gamma \cdot max_a Q(T(s'), a') - Q(T(s), a)) \end{aligned}$$

Where $T(s)$ returns the tiles that are activated by the state s. TC simplifies the learning for an agent by allowing the agent to learn the tile-space rather than the state-space, which is usually much smaller in cardinality.

The basic TC algorithm can be seen in Algorithm 1. The algorithm starts by initialising the tilings and the tiles within. Then, while there is still time to learn it repeatedly resets the environment and begins an episode. During an episode the agent perceives the current state, selects and performs the action according to a exploration/exploitation policy, receives a reward, and for each tiling updates the activated tile. S is the set states in the environment, A is the set of actions, T is the tile coding function, R is the reward function, m is the number of tilings to use, n is the total number of tiles to use, π is the action selection policy, α is the learning rate, and γ is the future discount factor.

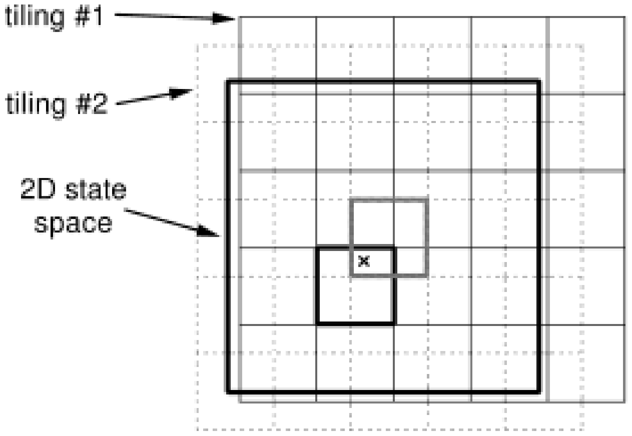


Figure 1: Multiple, overlapping grid tilings [8]

Algorithm 1 Tile-Coding($S, A, T, R, m, n, \alpha, \gamma$)

```
1: for $i \leftarrow 1$ to $m$ do
2:     Initialise tiling $i$ with $n/m$ tiles
3:     for $j \leftarrow 1$ to $n/m$ do
4:         Initialise tile $t_j$ and $Q(t_j, a) \leftarrow 0, \forall a \in A$
5: repeat
6:     reset the environment
7:     while episode not over do
8:         $s \leftarrow$ current state from $S$
9:         $a \leftarrow$ action chosen by exploration policy
10:        $s' \leftarrow$ state resulting from executing $a$ in $s$
11:        $r \leftarrow R(s, a, s')$
12:        for $i \leftarrow 1$ to $m$ do
13:            $t \leftarrow T_i(s)$
14:            $t' \leftarrow T_i(s')$
15:            $\Delta \leftarrow r + \gamma \cdot Q(t', a') - Q(t, a)$
16:            $Q(t, a) \leftarrow Q(t, a) + \frac{\alpha}{m}\Delta$
17: until time expires
```

Tile Coding, as any value function approximation method, impacts the agent's learning. This can be positive or negative; for example, TC can reduce the time required to learn a policy, or in the worst case TC can hinder learning entirely. Under the assumptions that transitions between states within a tile yield the same reward, all actions can result in a transition to a state within the same tile, computers approximate numbers, and Q-Learning or SARSA is used the following theorems hold [6]:

THEOREM 1. *Increasing the probability that an agent will do multiple, consecutive transitions on a single tile will decrease an agent's ability to stably learn a policy or to retain a learnt policy.*

THEOREM 2. *Decreasing the exit probability of non-optimal actions, or increasing the exit probability of optimal action(s), will increase an agent's ability to stably learn and retain learnt policies.*

Theorem 1 means that when a tile is too large an agent can become unable to stably learn or reliably retain a learnt policy; though when a tile is too small the agent gains no benefit from using TC. Furthermore the longer an agent is given to learn using TC the higher the probability the agent will be unable to reliably retain a learnt policy. How this impacts an agent's performance is dependant on different conditions such as the size of the tile and the position of the tile in the tile space.

Theorem 2 implies that the problems raised by Theorem 1 can be counteracted by increasing the probability exiting a tile on an optimal actions. This can be achieved by altering a tile's shape.

2.3 Automated Tiling Design

A limitation of TC is that it requires a human designer to correctly determine the tilings. While in principle tiles can be any shape or size they are usually axis-aligned rectangles. Sherstov and Stone previously demonstrated that reducing the size of tiles during learning is beneficial [7]. This is due to the generalised knowledge of the larger tiles being passed on to the smaller tiles which are then able to refine that information.

Whiteson *et al* [10] introduced Adaptive Tile Coding (ATC), an algorithm that automatically alters a single tiling during learning. When a threshold is reached ATC splits a tile from the tiling which maximises a criterion. The tile is split in half along one feature. ATC demonstrated that automated tilings can lead to faster learning and improved learnt policies. In our empirical experiments we compare ATC's performance with AMRT under the assumption that the agent does not have access to the transition function (see Section 4.2.3).

3. TILE SHAPE

Theorem 2 states that the performance of TC can be improved by increasing the probability of exiting a tile with an optimal action. There are two ways to alter the exit probabilities of a tile: changing its size or its shape. The impacts of tile size have already been shown by Scopes and Kudenko [6]. The impacts of tile shape is a significantly harder area to study due to the vast space of possible shapes. Nevertheless an algorithm that decides when and where to split tiles should also have some notion of how. Should a tile be split in half along every feature to create uniform squares (see Figure 2a), split so that one or more features gain a higher resolution (see Figure 2b), or are there specific conditions when one method is preferable other the other?

To approach these questions we have conducted an empirical study of the effects of tile shape.

3.1 Experiments & Results

3.1.1 2d-Random Walk (2d-RW)

2d-Random Walk (2d-RW) is a 2-dimensional environment where the agent must traverse from some starting point, usually the centre of the environment, to one of the goal states. There are four possible actions: *NORTH*, *EAST*, *SOUTH*, and *WEST*. The goal states are located around the outer perimeter of the environment. In Figure 3 the red circle is the starting point, any of the green squares are goal states, and the four corner squares are unreachable states. The agent receives a reward for reaching a goal state, all goal states along a particular edge will yield the same reward but the reward can vary between different edges. For example reaching any goal state to the east might yield a reward of 10 whereas any other goal state would yield 5.

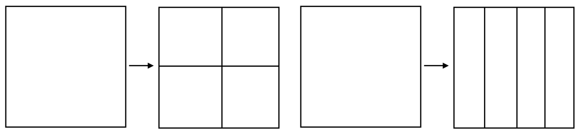

(a) A uniform split of a tile (b) A slats split of a tile

Figure 2: Examples of different ways to split tiles

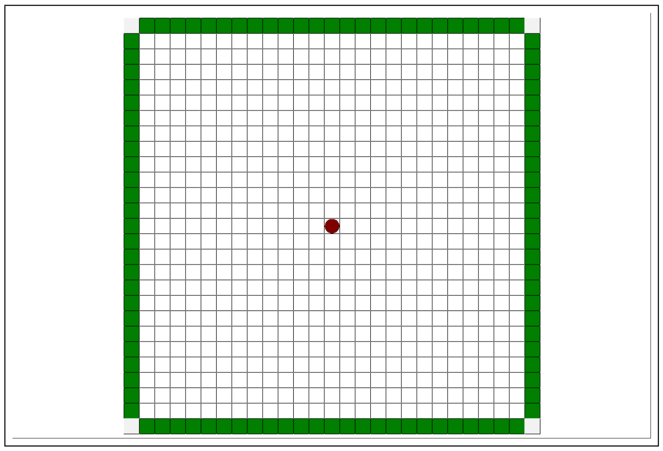
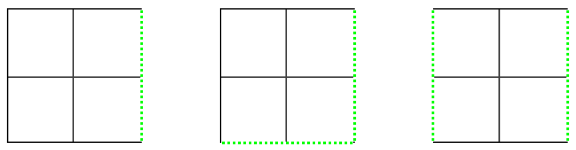


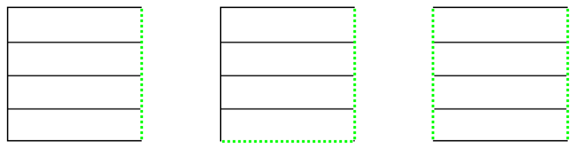
Figure 3: A 2d-Random Walk Environment (Agents moves from the red circle to a green square)

Q-learning was set up as follows: $\alpha = 0.4$ and $\gamma = 0.9$, and ϵ-Greedy exploration was used with $\epsilon = 0.4$. Tile shapes used were limited to axis-aligned rectangles. Specifically, in each experiment all the tiles in a tiling had the same shape. We compared 1 tiling of varying sizes of squares and oblongs (called *slats*) in various sizes of the environment. The reward for reaching different perimeter edges in the environment also varied. Experiments were set up so that there are unique optimal policies or multiple optimal policies. The average of 25 runs is recorded in the experiment results.

Figure 4 illustrates some of the different possible set-ups used in the empirical study. The bold, green edges indicate that the highest reward of 10 is given when the agent arrives at a state on that edge. When the agent arrives at any other edge it receives a reward of 5. The inner lines show the shape of the tiles used. The edges that yielded the highest reward were altered. For example, an instance of 2d-RW with a unique optimal policy could have the edge corresponding with the highest reward on any one of the north, east, south, or west perimeter edges.

2d-RW was used to empirically show the impact of tile shape on learning. 2d-RW can be viewed as modelling a sin-


(a) Square tiles with unique, dual adjacent, and dual opposite optimal policies respectively


(b) Slat tiles parallel to the direction of the unique, dual adjacent, and dual opposite optimal policies respectively

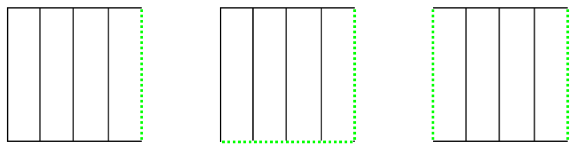

(c) Slat tiles perpendicular to the direction of the unique, dual adjacent, and dual opposite optimal policies respectively

Figure 4: Examples of different experiment set-ups

gle tile within a larger environment. Whereas in 2d-RW an episode ends when a perimeter state is reached, this would represent the agent leaving this tile in a larger environment.

3.1.2 Results of Tile Shape

Table 1 shows the mean average reward collected for the tile shape experiments. The results are separated into 4 different environment sizes: 03x03, 05x05, 09x09, and 11x11. Different tile shapes are evaluated in different instances of the 2d-RW environment, where the number of optimal policies is varied: unique, two adjacent, and two opposite (see Figure 4). We define three types of tiles: Square tiles, Parallel Tiles (slat tiles that lie parallel to the direction of movement toward the goal), and Perpendicular Tiles (slat tiles that lie perpendicular to the direction of movement toward the goal). It is worth noting in the cases where the goals are adjacent slat tiles are simultaneously parallel and perpendicular to the two goals and therefore we should expect similar performance. The rightmost column in Table 1 shows how frequently the optimal decision on alignment of slat tiles needs to be correctly guessed for slat tiles to outperform square tiles. The frequency, w, is calculated using the following expression:

$$w = \frac{MIN(r_{par}, r_{per}) - r_{sq}}{MIN(r_{par}, r_{per}) - MAX(r_{par}, r_{per})}$$

where $MIN(r_{par}, r_{per}) \neq MAX(r_{par}, r_{per})$, r_{sq} is the average reward of square tiles, and r_{per} is the average reward of the perpendicular slat tiles and r_{par} is the average reward of the parallel slat tiles and:

$$w = \frac{r_{sq}}{MAX(r_{par}, r_{per})}$$

where $MIN(r_{par}, r_{per}) = MAX(r_{par}, r_{per})$. In cases where $MIN(r_{par}, r_{per}) \geq r_{sq}$ the frequency calculation is not applicable since slat tiles perform at least as well as square tiles in the worst case.

03x03		Squares	Slats		w
			Perpendicular	Parallel	
	Unique	9.999	**10**	5.072	100%
Dual	Adjacent	**10**	**10**	**10**	n/a
	Opposite	**10**	9.999	5.132	100%

05x05		Squares	Slats		w
			Perpendicular	Parallel	
	Unique	9.015	**9.513**	5.010	89%
Dual	Adjacent	9.067	**9.493**	9.492	n/a
	Opposite	9.983	**9.998**	5.017	100%

09x09		Squares	Slats		w
			Perpendicular	Parallel	
	Unique	8.704	**8.993**	5.014	93%
Dual	Adjacent	9.003	8.995	**9.010**	53%
	Opposite	9.883	**9.979**	5.031	98%

11x11		Squares	Slats		w
			Perpendicular	Parallel	
	Unique	8.809	**8.983**	5.025	96%
Dual	Adjacent	**9.308**	9.186	9.262	160%
	Opposite	9.607	**9.917**	5.060	94%

Table 1: Average rewards of Tile Shapes on a range of environment sizes

The results of our experiments show that:

- when there is a unique optimal policy, using perpendicular slat tiles result in the quickest and most stable learning;
- when there are 2 optimal policies that are opposite to one another, using perpendicular slat tiles result in the quickest and most stable learning;
- when there are 2 optimal policies that are adjacent to one another, then no one tile shape consistently outperformed any other.

We can conclude therefore, when there is high certainty that all optimal policies have been correctly identified it is beneficial to use perpendicular slat tiles over square tiles. Whereas, when there isn't high certainty then square tiles should be used instead.

Considering that automated tiling methods use estimated values to decide which tiles to split, it is unlikely they will have high enough certainty to make slat tiles beneficial. Therefore, it is the conclusion of this study that automated tiling methods should split tiles evenly along each feature rather than giving priority to one feature over another.

4. MIXED RESOLUTION TILING

Mixed Resolution Tiling (MRT) was introduced as a manual method of devising tilings which consists of tiles of heterogeneous sizes [6]. Specifically MRT uses knowledge of optimal transition paths through the state space of an MDP; tiles with "high resolution", small tiles, should be used on or near an optimal path, "low resolution", large tiles, should be used away from the optimal paths, and that the resolution of the tiles should be tapered in between. MRT was based on the heuristics derived from theoretical properties of TC.

Automation of MRT requires an algorithm which learns about the environment splitting tiles when confident. In effect, the algorithm needs to learn a transition model of the tile space for each tiling which keeps track of its confidence. The transition model keeps track of the number of transitions from one tile to another distinct tile for each action and therefore can return the observed probability of an tile-action-tile transition. Using such a transition model a path from the start to the end tile can be determined. Furthermore, automated MRT should have a means of determining whether a tile is at risk of becoming unreliable in retaining a learnt policy. Finally, automated MRT would need fail safes to ensure that if the original tiles were too large for a given environment it could reduce the size of the tiles enough to be able to start learning.

4.1 Algorithmic Automation

Automated Mixed Resolution Tiling (AMRT), shown in Algorithm 2, is based on MRT and the standard tile coding algorithm (see Algorithm 1) AMRT adds two functions:

- AMRT-DURING which is called at every update during an episode; and
- AMRT-POST which is called after every episode.

To do this AMRT requires a constant *maxSteps*, the maximum number of steps in an episode, a new parameter c, which denotes a confidence measure, and three variables *nSteps*, the number of updates performed during the current episode, *nUpdates*, the counter for the total number of updates performed, and *tm*, a set of transition models for each tiling.

The principle behind AMRT is to ensure that tiles on or around the optimal path(s) have high resolution, all tiles are resolved *enough* for exploration, and the impact of local convergence where the Q-values on a single tile all converge toward the same value ($\frac{r}{1-\gamma}$ where $\gamma < 1$) is reduced.

AMRT ensures tiles on or around the optimal path are smaller by using a transition model of the tile space to calculate the least-cost path from an initial tile to the most recently found goal tile and splitting all tiles, and neighbouring tiles, along that path. At the end of an episode, if a goal state was reached, AMRT performs an A^* search for the least-cost path from the starting tile or the goal tile. The cost is calculated by using a function of the current Q-values and the estimated probabilities of the transitions along that path.

AMRT detects too large tiles in a number of ways: it tracks the number of repeated updates[1] on a tile and will only split tiles on or around the currently estimated optimal path whose ratio of the total number of updates to the number of repeated updates exceeds $\frac{1}{|A|}$, where $|A|$ is the number of actions (this is the SPLIT-REPEATED function in Algorithm 4, line 7). During learning, if $c \cdot maxSteps$ updates have been performed since a goal state was last entered all tiles are split (see Algorithm 3, lines 5 to 9). Finally, if at the end of an episode only the starting tile was activated on a particular tiling then all tiles in that tiling are split (see Algorithm 4, lines 10 to 14).

[1]A repeated update is any update where the state-action-state transition moves into a new state but the activated tile remains the same.

Algorithm 2 AMRT$(S, A, T, R, m, n, \pi, \alpha, \gamma, c)$

```
1: $maxSteps \leftarrow$ maximum number of steps per episode
2: $nUpdates \leftarrow 0$
3: for $i \leftarrow 1$ to $m$ do
4:     Initialise tiling $i$ with $n/m$ tiles
5:     Initialise transition model, $tm_i$
6:     for $j \leftarrow 1$ to $n/m$ do
7:         Initialise tile $t_j$ and $Q(t_j, a) \leftarrow 0, \forall a \in A$
8: repeat
9:     reset the environment
10:    $nSteps \leftarrow 0$
11:    $\bar{s} \leftarrow$ starting state from $S$
12:    while episode not over do
13:        $s \leftarrow$ current state from $S$
14:        $a \leftarrow$ action chosen by exploration policy
15:        $s' \leftarrow$ state resulting from executing $a$ in $s$
16:        $r \leftarrow R(s, a, s')$
17:        AMRT-DURING()
18:        for $i \leftarrow 1$ to $m$ do
19:            $t \leftarrow T_i(s)$
20:            $t' \leftarrow T_i(s')$
21:            $\Delta \leftarrow r + \gamma \cdot Q(t', a') - Q(t, a)$
22:            $Q(t, a) \leftarrow Q(t, a) + \frac{\alpha}{m}\Delta$
23:    AMRT-POST()
24: until time expires
```

Empirical trials indicated that when the Q-values on a single tile were all locally converging toward a non-zero value exploration could be hindered. This is caused by most exploration policies using a mixture of exploration and *exploitation*, exploiting the currently held best action a percentage of the time. The problem tile is usually located in the tile space in such a way that the tile cannot be readily exited using the currently held best action. Since the exploration policy is exploiting this action a percentage of the time, there is an overall bias for action being used. This causes the Q-values of this tile to become, in a sense, locked. Furthermore, splitting the tile would not solve the problem as the locked behaviour would be propagated to the split tiles. AMRT counteracts Q-value local convergence of all Q-values on a single tile to a single value by detecting signs of local convergence on a tile during learning and resetting the Q-values of that tile if convergence has almost occurred (see Algorithm 3, lines 13 and 14).

Algorithm 3, AMRT-DURING, specifies what to do at every step during an episode. Firstly, *nSteps* and *nUpdates* are incremented (see lines 1 and 2). Then, if the agent moved into a terminal state, *nSteps* is subtracted from *nUpdates*. This informs AMRT that the steps of this episode should not be considered in the detection of too large tiles (see lines 3 and 4). Next, if the threshold for *nUpdates* is reached ($c \cdot maxSteps$) AMRT believes that there are too large tiles

Algorithm 3 AMRT-DURING()

```
1: $nUpdates \leftarrow nUpdates + 1$
2: $nSteps \leftarrow nSteps + 1$
3: if $inTerminalState()$ then
4:     $nUpdates \leftarrow nUpdates - nSteps$
5: if $nUpdates > c \cdot maxSteps$ then
6:     for $i \leftarrow 1$ to $m$ do
7:         split all tiles in tiling $i$
8:         CLEAR($tm_i$)
9:     $nUpdates \leftarrow 0$
10: for $i \leftarrow 1$ to $m$ do
11:     if $t_i \neq t'_i$ then
12:         INCREMENT($tm_i, t_i, t'_i, a$)
13:     if ALMOST-CONVERGED($t_i$) then
14:         reset $t_i$ Q-values
```

present to properly explore the environment. Therefore all tiles are split, the transition models cleared, and *nUpdates* reset to zero see (lines 5 to 9). Finally for each tiling if the activated tile changed, the transition model is informed and if all the tile's Q-values have almost locally converged toward a single value its Q-values are reset (see lines 10 to 14).

Algorithm 4, AMRT-POST, specifies what to do after every episode. Firstly, if a goal state is reached, for each tiling an A^* search for the least-cost path from the tile activated by the initial state, $T_i(\bar{s})$, to the tile activated by the goal state, $T_i(\dot{s})$, through the transition model, tm_i, is performed. Tiles that have had fewer updates than the confidence factor c are excluded from the search (see lines 1 to 8). This means there may not be a path through the transition model from the start tile to the goal tile. If a path is found, then the neighbouring tiles, according to the transition model, tm_i, are found as well. All tiles in the union of the path and neighbours with at least $\frac{1}{|A|}$ percent, where $|A|$ is the number of actions, of repeated updates are split and those same tiles are removed from tm_i. Finally, if a goal was not reached and the whole episode was spent only within the starting tile ($|tm_i| \leq 1$), this indicates that the tiles of the tiling are too large therefore all tiles of that tiling are split and *nUpdates* is set to zero (see lines 9 to 14).

Algorithm 4 AMRT-POST()

```
1: if goal state was reached then
2:     $\dot{s} \leftarrow$ goal state reached from $S$
3:     for $i \leftarrow 1$ to $m$ do
4:         $path \leftarrow A^*(tm_i, T_i(\bar{s}), T_i(\dot{s}))$
5:         if not $empty(path)$ then
6:             $neighbours \leftarrow neighboursOf(tm_i, path)$
7:             SPLIT-REPEATED($path \cup neighbours$)
8:             REMOVE-ALL($path \cup neighbours, tm_i$)
9: else
10:     for $i \leftarrow 1$ to $m$ do
11:         if $|tm_i| \leq 1$ then
12:             split all tiles
13:             CLEAR($tm_i$)
14:             $nUpdates \leftarrow 0$
```

Following from the conclusions of the tile shape study when tiles are split, they are split in half along every feature to form square tiles, e.g. for 2 features a single tile would become 4 new tiles. Empirical trials also indicated that setting a minimum size for a tile to be allowed to split decreased the running time of AMRT. When a tile was split, counters to monitor the number of updates and repeated updates for each new tile were set to zero.

4.2 Experimentation & Results

4.2.1 Puddle World

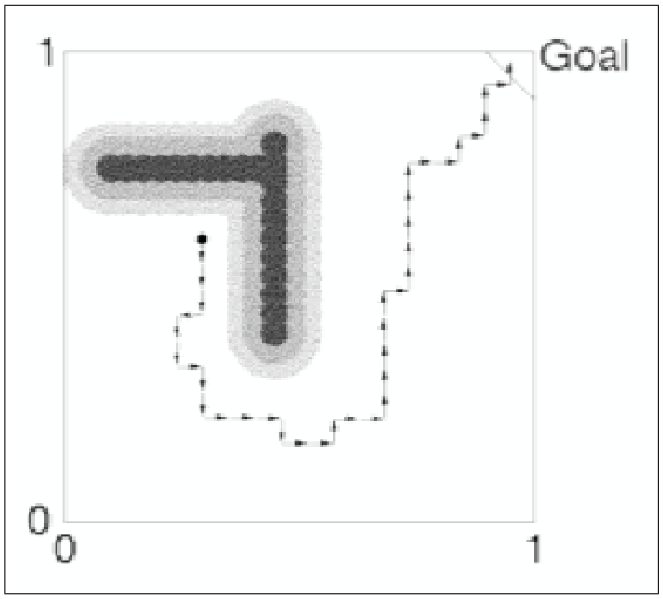


Figure 5: A Puddle World Environment (Agents move toward the goal avoiding the puddles

Puddle World is a continuous 2d navigation environment where an agent starts in the north-west and must travel to the south-east avoiding "puddles". There are four available actions: *NORTH*, *EAST*, *SOUTH*, and *WEST*. Although the movement distance is fixed the environment is continuous as Gaussian-random noise is applied to the agent's movement. There is a negative reward for each action the agent takes and it gets a reward of 0 for reaching the goal area. The agent also receives an increasingly large negative reward for being in a puddle the closer to the centre of that puddle the agent is. There are two state features: the x and y coordinate of the agent.

4.2.2 Mountain Car

Mountain Car is a continuous environment where an agent must control a vehicle stuck in a valley through forward, backward, and neutral actions. The objective is to arrive at the far right peak but this can only done if enough momentum is gained from coming down the left-hand side of the valley. The agent is accelerated toward the bottom of the valley by gravity. The agent receives a reward of -1 for every action taken and 0 for reaching the goal. There are two state features: the location and velocity of the car.

4.2.3 Results of AMRT

Puddle World and Mountain Car were used to compare the performance of AMRT against fixed uniform TC and ATC [10]. The implementation of ATC used does not require

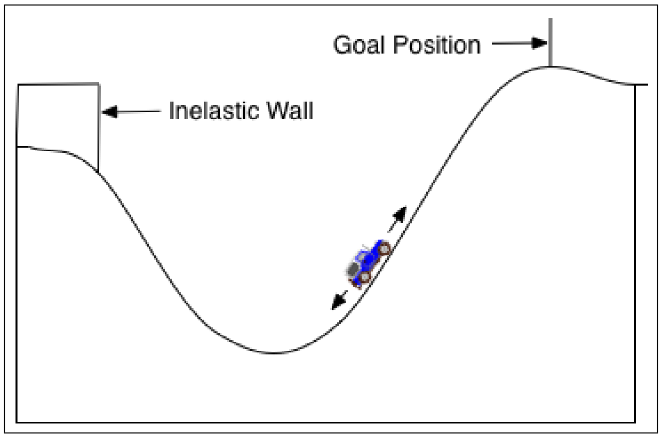


Figure 6: A Mountain Car Environment (Agents escape the valley)

the transition function. Experiments were conducted with different number of tilings, $m \in \{1, 5, 10, 25, 50\}$, and tiles per feature, $n \in \{5, 10, 15, 20, 25, 50, 100\}$, for fixed uniform TC and a selection, including the best found, have been used as comparison. Likewise, the threshold parameter, p, for ATC was trialled at $p \in \{5, 10, 15, ..., 95, 100\}$ and like the authors we found $p = 50$ a good value. For AMRT we discovered through trial and error that a consistently good value for the confidence, $c = 50$. AMRT used 1 tiling and initially started with 3 tiles per feature; AMRT was splitting was limited to having at most 100 tiles per feature.

The ATC implementation received from the authors learnt the value function for each tile rather than the Q-value. The ATC implementation also used the transition function in action selection to generate the Q-values by sampling each action once in the environment and using the sampled transition reward and state value. Furthermore, the learning algorithm sampled random states of the environment instead of playing out episodes. Finally, the policy criterion requires the transition function to gain access to the number of possible successor states. For these reasons the implementation given had to be altered to remove the need for knowledge about the transition function. These alterations included: playing out episodes, learning Q-values and using them in action selection, and modifying one of the tile splitting criterion[2].

Q-learning in the Puddle World Environment was set up as follows: $\alpha = 0.4$ and $\gamma = 0.9$, and ϵ-Greedy exploration was used with $\epsilon = 0.4$ without linear decay. Experiments were repeated 25 times and the average reward after each update is shown the the results (shown in Figure 7). The y-axis is the reward and the x-axis is the number of updates. The lightly coloured areas surrounding each line represents the standard error allowing us to be confident that these results are statistically significant. The black line is AMRT with an initial 3x3 tiling and $c = 50$, the green line is 10x10 TC with 1 tiling, the blue is a 25x25 TC with 1 tiling, and the red is ATC with $p = 50$ following the Difference criterion (the better performing of the two criteria with our implementation).

[2]Specifically, the policy criterion was altered to increment the change counter when a sub-tile's Q-values would result in a possible change in policy

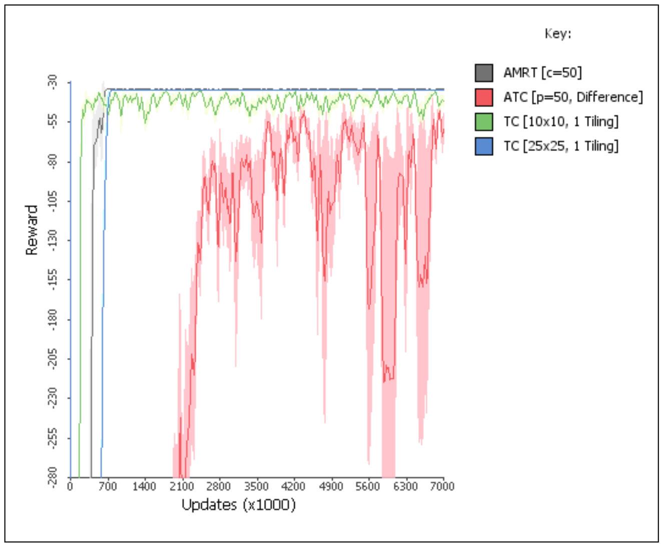


Figure 7: Puddle World Results: AMRT vs various instance of TC and ATC

We can see that AMRT is the first to reach an optimal policy. 10x10 TC with 1 tiling learns a good policy faster than AMRT but AMRT quickly overtakes. 25x25 TC with 1 tiling learns at a similar yet slower pace to AMRT but does also reach the an optimal policy. This empirically shows that in the Puddle World environment, AMRT can match the best fixed uniform tiling performance and arrives at its final policy more quickly.

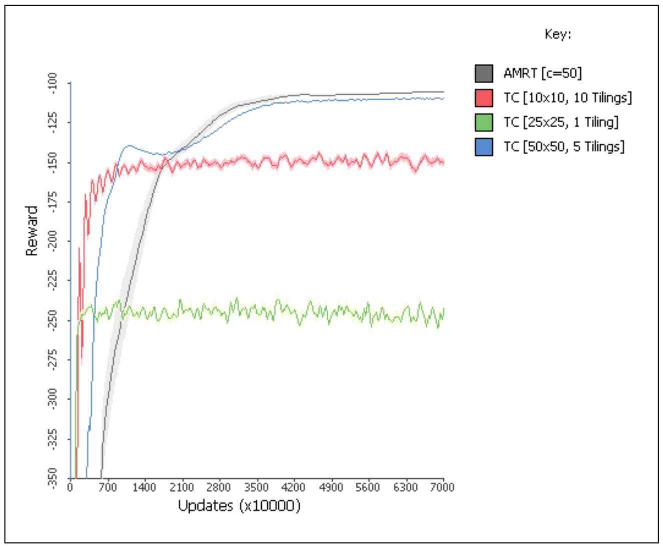


Figure 8: Mountain Car Results: AMRT vs various instances of TC

Next we performed similar experiments in the Mountain Car environment to confirm that AMRT can match or exceed the performance of fixed uniform TC in non-navigation environments.

Q-learning in the Mountain Car environment was set up as follows: $\alpha = 0.4$ and $\gamma = 0.999$, and ϵ-Greedy exploration was used with $\epsilon = 0.4$ without linear decay. Experiments were repeated 25 times and the average reward after each update is shown the the results (shown in Figure 8). The y-axis is the reward and the x-axis is the number of updates. The lightly coloured areas surrounding each line represents the standard error allowing us to be confident that these results are statistically significant. The black line is AMRT with an initial 3x3 tiling and $c = 50$, the red is 10x10 TC with 10 Tilings, the green is 25x25 TC with 1 Tiling, and the blue is 50x50 TC with 5 Tilings. ATC did not perform well enough to appear on this graph.

In a similar manner to the results of Puddle World, AMRT is the first to reach a policy with the highest reward. Again some of the TC representations learn faster in the initial stages we can see that AMRT eventually outperforms all of the present TC initialisations. This empirically shows that AMRT is not limited to navigation environments and can exceed the performance of fixed uniform TC.

The results shown above are clear confirmation that AMRT can match or exceed the performance of basic TC and ATC. AMRT achieves this without the need for trial and error of different TC set-ups or a human designer with prior knowledge of the environment. Furthermore, since AMRT uses A^* to solve a self generated sub-problem the running times of AMRT and TC were compared. At worse AMRT took less than twice as long as the fastest fixed uniform TC run and at best took less time to run. This means that AMRT doubly removes the need for trial and error of different TC set-ups or a human designer with prior knowledge of the environment.

5. CONCLUSIONS

5.1 Discussion

This paper has presented a novel algorithm AMRT, an automated version of the manual method of tiling design MRT. AMRT can outperform fixed uniform TC representations without any prior knowledge of the environment or trial and error guesses of different numbers of tilings and tiles per feature. AMRT achieves this by automatically altering tiles during learning to better match the environment. One of the appeals of TC is its typically speed of execution. AMRT is at worst takes twice as long as a run of the fastest fixed uniform Tile Coding. This removes the need for the user to design the tiling or trial an error of different set-ups.

This paper has also concluded that tile shapes can both positively and negatively impact learning. For tile shapes to be effectively used there must be some knowledge of the environment either *a priori* or learnt alongside learning a policy. Further to this, only with high confidence should slat shaped tiles be chosen over square shaped tiles at the risk of an incorrect choice leading to the performance of learning being hindered.

5.2 Future Work

The study of tile shapes in this paper is a good start but there is more work needed to be done. This paper limited the study of tile shapes to axis-aligned rectangular tiles of the same shape in each tiling. Further work could be done expanding on the study by looking into heterogeneous tile shapes, non-axis-aligned tiles, or non-rectangular shaped tiles.

Further work could also be done to AMRT by incorporating the criterion's proposed by Whiteson *et al* [10] where they use sub-tiles to help decide how a tile should be split. Also Chow and Tsitsiklis' [3] work on how to compute the tile width of a fixed uniform tiling necessary to learn an approximate optimal policy could be used to set a minimum tile size for AMRT.

REFERENCES

[1] J. S. Albus. *Brains, behavior, and robotics.* Byte books Peterborough, NH, 1981.

[2] D. P. Bertsekas. *Dynamic Programming and Optimal Control.* Athena Scientific, 3rd edition, 2007.

[3] C.-S. Chow and J. Tsitsiklis. An optimal one-way multigrid algorithm for discrete-time stochastic control. *Automatic Control, IEEE Transactions on*, 36(8):898–914, Aug 1991.

[4] R. Munos and A. Moore. Variable resolution discretization in optimal control. In *Machine Learning*, volume 49, pages 291–323, 2002.

[5] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* John Wiley and Sons, Inc., New York, NY, USA, 1994.

[6] P. Scopes and D. Kudenko. Theoretical properties and heuristics for tile coding. In *ALA Workshop, AAMAS 2014*, 2014.

[7] E. A. Sherstov and P. Stone. Function approximation via tile coding: Automating parameter choice. In *of Lecture Notes in Artificial Intelligence*, pages 194–205. Springer Verlag, 2005.

[8] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[9] C. J. C. H. Watkins. *Learning from Delayed Rewards.* PhD thesis, Cambridge University, 1989.

[10] S. Whiteson, M. E. Taylor, and P. Stone. Adaptive Tile Coding for Value Function Approximation. Technical report, University of Texas at Austin, 2007. Technical Report AI-TR-07-339.