Strategy Choosing μRTS Bot

October 30, 2018

Abstract

1 Techniques Implemented

1.1 Temporal Difference Learning

Choosing an overall strategy for the map is crucial to winning microRTS **??**. SC identifies the best from a predefined list of hardcoded strategies. Temporal Difference learning is used to assess the value of a given strategy at each timestep. Strategy Chooser uses a forward model to predict gamestate after several gameticks playing with a given strategy. The hardcoded strategies selected were WorkerRush, RangedRush, LightRush and HeavyRush. These were selected as they generally performed well, each strategy was distinct from the others and each had a map which they consistently beat the others. E.g. WorkerRush on a small map is very strong. When the forward model had no more time to simulate, or the simulation reached a terminal state, the gamestate at that point was evaluated using the SimpleSqrtEvaluationFunction3. That value was taken to be the value of the strategy being simulated.

1.2 Minor improvements to hardcoded strategies

After observing several games of SC, its performance was reasonable but any change in strategy performed poorly as the hardcoded strategies did not compliment each other. E.g. The non-workerrush strategies would send all workers to collect resources, even if moments ago they had been attacking. To make StrategyChooser more consistent, we made some alterations to the hardcoded strategies. The main improvements were: 2 workers assigned to resources, no more than one worker assigned to a resource square at a time, spare workers always sent to attack, resource workers moving to attack if approached by an enemy and barracks positioned consistently. Now that all strategies handled these elements consistently, Strategy Chooser could change strategies without undoing the advantages gained by a previous one.

1.3 Improving the Forward Modelling

The Monte Carlo Tree Search (MCTS) bot in μRTS simulates future gamestates by playing RandomBiasedAI moves against a RandomBiasedAI opponent. However, RandomBiasedAI is a very weak, non-aggressive strategy which would result in overly optomistic simulations of future gamestates. Strategy Chooser estimates the strategy of the opponent first and then runs the forward model with its candidate strategy against that opponent. To estimate the opponent's strategy the count of each of the types of units it has are multiplied by a weighting. See the pseudoCode below:

```
noWorkers,noRanged,noHeavy,noLight = getEnemyUnits();
enemyStrategies = [workerRush, rangedRush, heavyRush, lightRush]
votes[0] = 4 * noWorkers;
votes[1] = 5 * noRanged;
votes[2] = 5 * noHeavy;
votes[3] = 5 * noLight;

index = getIndexOfMax(votes)
enemyStrategy = enemyStrategies[index]
```

A smaller weighting was chosen for the number of workers because all strategies have workers so they're not a strong indicator of workerRush.

The strategy with the largest score is then considered to be the opponents strategy. Experiments were conducted with the strategy chooser considering the top 2 enemy strategies and taking an average evaluation score. This had the advantage of preparing for a shift in the enemy strategy but halved the forward model depth, which had a greater negative impact. Therefore SC only considers the most likely enemy strategy. This could be investigated again if the forward model depth can be increased.

1.4 Tuning Forward Model Depth

The time for the forward model simulation was limited by the 100ms limit given to the bot. This was then split between all the strategies Strategy Chooser was considering. There was therefore a trade-off between the simulation time and the breadth of options. Typically 100ms would allow enough time for approximately 20 steps forward in the forward model, that would then have to be split between each of the considered options. E.g. If Strategy Chooser had 4 strategies to choose then the forward model would simulate the gamestates at most 5 game ticks into the future.

Is depth the right word for the forward distance considered?

5 game ticks is not far enough to determine the long term value of a strategy. To increase this value, an inertia value was created. This inertia value, I, determines for how many gameticks the same strategy would be used before deciding upon a new one. This meant that strategies would play consistently for I gameticks. Strategy Chooser saves the progress of the forward model during this time, continuing it in the next gametick. This increases the forward

model's depth I times. The greater search depth allows strategy chooser to consider longer term effects of a decision. The inertia also results in less erratic and indecisive play because it will only switch strategies every Ith gametick.

1.5 Improving the Gamestate Evaluation

The basic evaluation function used was the SimpleSqrtEvaluationFunction3. This calculates the difference in strength of the max player compared to the min player. A positive score indicates the max player is in a stronger position. The function considers unit costs, resources and unit health but ignores unit positions. It was noticeable that many of the AIs that attempted to maximise this function were not aggressive because it is possible to increase the evaluationscore without ever attacking the enemy. This resulted in AIs that were quite passive. To encourage a more aggressive approach, we created a new evaluation function that also considers the distance of units from the enemy base and provides a parameter for tuning the aggression. The pseudocode is below:

```
function unitValue(Unit u):
   value = u.getResources * RESOURCE_IN_WORKER_WEIGHTING
   value += UNIT_BONUS_MULTIPLIER * u.getCost()*sqrt(
       u.getHitPoints()/u.getMaxHitPoints() )
   return value

function playerScore(player):
   enemyBase = player.getBase;
   maxManhattenDist = map.getWidth() + map.getHeight();

   for each unit in player.getUnits():
      score = unitValue(unit) * maxManhattenDist / (
          manhattenDistanceBetween(unit, enemyBase) +
          maxMamhattenDist )
   return score;

function evaluateGamestate(player, enemyPlayer, aggresionWeight):
   ourScore = playerScore(player);
   enemyScore = playerScore(enemyPlayer);
   return (2 * ourScore / (ourScore + aggresionWeight * enemyScore) )
       - 1;
```

This results in units that are closer to the enemy base being valued up to twice as much as units further away. This should result in a gradient to push units towards the enemy and to push enemy units away from the Strategy Chooser base.

The aggression weighting is another tunable parameter, if set high, then the evaluation function will value weakening the opposition more highly. If the aggression weight is small, it will value preservation of its own units more highly than the destruction of enemy units.

References

[1] Santiago Ontañón, Nicolas A Barriga, Cleyton R Silva, Rubens O Moraes, and Levi HS Lelis. The first microrts artificial intelligence competition. *AI Magazine*, 39(1), 2018.