Agent Switch - Combining Different Strategies

Syed Hassan, Cristiana Pacheco, Hafiz Ahmed, Poppy Wright

Queen Mary, University of London, UK

s.f.hassan@se18.qmul.ac.uk, c.pacheco@qmul.ac.uk, hafiz.ahmed@se18.qmul.ac.uk, p.wright@se18.qmul.ac.uk

Abstract
This report presents the results of implementing a 'Switch' mechanism in the simplified real-time strategy game MicroRTS; the idea is that, depending on the state of the game, a bot can switch between agents (using a Finite State Machine) in order to employ the best strategy. Our algorithm is based upon modifying four methods already implemented in MicroRTS (EconomyRush, LightRush, LightDefense and WorkerRush). The strategy will be simulated by conducting game-play with UCT, NaïveMCTS, EconomyRush, LightRush, LightDefense and WorkerRush. The results produced show that creating a meta-algorithm that combines different AI behaviours, can produce an improved version of the original AIs.

1 Introduction

Evolutionary computation evolves neural networks/agents to succeed by creating and developing machine behaviour; the abstract simulation environment MicroRTS implements this idea. It is a simplified version of StarCraft, developed by Santiago Ontanon for testing, where the users control over the game units is granular. It is used to apply tree-search algorithms to Real Time Strategy games where two players controlling a group of four different types of units and two structures with different durative actions and characteristics compete simultaneously to eliminate the other within a set number of game ticks. Using a forward model, the AIs can simulate the effect of actions allowing for planning and the evaluation of potential future states to determine the most favourable next steps; this allows for game-tree search techniques to be developed more easily.
MicroRTS has been created to perform research in the development of AI and various areas of machine learning. But which technique or algorithm is the best? Some approaches are quite known specially for their involvement in frameworks such as these - deterministic which permits forward simulation [1][2]. However, a framework such as MicroRTS and known video games still use techniques which are simpler. Such as Goal Oriented Behaviours (GOAL) [6]. Which one is better? Or achieves a higher win rate?

In this project the aim is to create a player for the MicroRTS game which we will compare against different built in families of AIs, such as hard-coded strategies, Monte Carlo search and UCT-based strategies. We will compare our agent them in a two round tournament to gain an understanding of both its issues and successes in our method. We present previous work done in the area, an explanation of this framework and an explanation on the switch technique we implemented. Followed by the results and analysis of this testing and conclusions for future work from this agent/switch.

2 Literature Review

Combinatorial Multi-Armed Bandit Problem

The large branching factor problem is often represented as a Combinatorial Multi-Armed Bandit problem (a variant of the Multi-armed Bandit problem - a sequential decision problem defined over a single state) as it is a simple model for decision making [16]. Beforehand most success in approaching the large branching factor problem was found using Monte Carlo Tree Search Algorithms and the fact RTS are decision problems with combinatorial action spaces had not been dealt with directly; building upon this, a new algorithm, NaïveMCTS, was created using the Naïve sampling strategy for MCTS algorithms based on a CMAB problem [7]. The results of this initial study show that NaïveMCTS performed far better as the branching factor grew [7].

Further supporting this, the idea was developed upon while adopting linear side information. The LSI (Monte Carlo Search with linear side information) works in two phases i.e. firstly generate actions for candidates, then evaluate these candidates. Then, adopting the LSI, two instances of the scheme were proposed. The results were consistent with [7]. Additionally, the usefulness of linear side information was demonstrated as the LSI instances constantly outperformed noSI (a simplified version of the 2PHASE-CMAB) [10]. Continuing from this work, the authors went onto analyse the quality and responsiveness of the NMC and LSI algorithms. The LSI was stronger with smaller budgets and shorter lookahead, suggesting it is more robust against inaccurate evaluations of the candidate actions due to the linear side information [11].

All the previous work in this area supports/builds upon one another, developing the problem in the right directions for further research.

Switch Method

Using several methods already implemented in MicroRTS, a meta-algorithm can be created. This allows the bot to switch between agents depending on which is most helpful to the current the state of the game.

One of the three submissions to the first MicroRTS competition was StrategyTactics. This builds on strengths of PuppetSearch and the tactical performance of NaiveMCTS by splitting the search time between both algorithms. The PuppetSearch produces the actions for all units while the NaiveMCTS refines them. While competing against RandomBiased, POWorkerRush, POLightRush, NaiveMCTS, PuppetSearch and

SCV, StrategyTactics achieved the highest win ratio over all maps, because it incorporates both high-level and low-level searches. Comparatively to the hard-coded bots, which lack appropriateness, it was able to excel in non-standard situations.

A major possibility to draw from this is the fact that SCV performed better than PuppetSearch on all the open maps; this implies that splitting the search time between SCV and NaïveMCTS, similarly to StrategyTactics, could produce a superior bot [19] and that further research should be done in this area.

Partial Observability

Currently, MicroRTS does not come with any AIs that can handle partial-observability [20]. Although the default configuration of MicroRTS is deterministic and real-time, non-determinism, and partial-observability can be experimented with.

The first known attempt to deal with partial-observability using a game tree search in real time games was entered in the first MicroRTS AI competition; the BS3NaiveMCTS used the idea of determinisation to infer the whole game state using a perfect memory to record the position of all the enemy units in parts that it had observed during any point in the game [19]. This extends from the idea that the optimal strategy can be found by sampling the game state rather than considering every frame due to the fast pace of the game [17].

An issue occurred with all the bots that entered the partially-observable track when it came to larger maps; it was difficult for the bots to find each other, hence the game usually ended with a draw [19].

The main point to take from previous experiments into partial-observability is the need for scouting and exploration [19].

Non-Deterministic

Non-deterministic strategies need to be able to handle every aspect of the game and can expose choice points, which mark locations in the script where alternative actions are to be considered, to a search algorithm the user specifies [18]. Currently, when dealing with non-determinism, MicroRTS is not suitable for evaluating such AI techniques [19].

From the minimal research performed in this area, similar observation that occur in standard tracks (fully-observable and deterministic) seem to occur in non-deterministic tracks; adding determinism does not change hard-coded bots performing better in standard situations and has little effect on the main factors that determine performance (map size, map type and partial-observability) [19]. The results from the IEEE-CIG 2018 Competition further supports this with almost identical results in the standard track and the non-deterministic track [22].

3 Benchmark

Real-Time Strategy

Real-time strategy (RTS) is a subgenre of strategy video games where the game does not progress incrementally in turns [4]. RTS games allow players to position structures and units during the game which are under their control, enabling manoeuvres to combat the opponent's assets. Thus, by essentially managing and controlling resources, tactics and strategies players can dominate or destroy their components as quickly as possible [5,3]. Examples of RTS games include Dune II, Total Annihilation, Warcraft, Command & Conquer, Age of Empires, and StarCraft series.
Their large state space and significant branching factors make RTS games exceptionally difficult for AI techniques. As RTS games are played in real time this minimises the decision time to make a move, furthermore they tend to be nondeterministic and partially observable [19]. For all these reasons, human players are still superior players to AI bots and thus further research to incorporate AI into RTS games is required. Since the first call for research on RTS game AI by Buro (2003) [24], a wide range of AI techniques have been explored to play RTS games. This has also lead to the development of competitions, such as the StarCraft AI competition to stimulate AI research in RTS games [19,9].

MicroRTS
MicroRTS is a simple RTS game which was designed to facilitate the development of AI research, the primary field being in game-tree search techniques, such as Monte-Carlo variations and minimax. MicroRTS is favourable in this area in comparison to a full-fledge RTS game as it does not face the same difficulties when evolving Artificial Intelligence. These more traditional approaches involve the investment of high time and engineering, have a very large branching factor, struggle to converge when faced with large state spaces and have such a large number of actions available at a given game state that it is near impossible to evaluate the favourability of game states. Comparatively, MicroRTS is deterministic and hence permits forward simulation, allows for scaling difficulty since the map size is adjustable and is simple when used to test and modify theoretical ideas.

MicroRTS Competition
The IEEE Computational Intelligence in Games (CIG) conference first hosted the MicroRTS AI competition in 2017 [19]. It was created by Ontañón to motivate research and reduce the level of engineering in the development of AI for RTS games. The game supports the development of planning and game-tree search techniques by giving the AIs access to a forward model to simulate the effect of actions. The focus here is on other core RTS problems (large state spaces and branching factors, partial observability and non-determinism), hence agents without a forward model are excluded from the competition [8]. In order to look into these three core areas, the competition incorporates different tracks focusing on one of the problems:

1. Standard track – deterministic and fully-observable environment, focusing on real-time, durative actions which scale to larger RTS games
2. Partial-observability track – deterministic and partially-observable environment, where bots cannot see beyond their unit's view

3. Non-deterministic – non-deterministic and fully-observable environment, where actions have stochastic effects

The current winner of the EEE-CIG 2018 MicroRTS Competition in the standard track over all maps is “TiamatBot”. The bot casts action abstractions to select a subset of pure strategies from a large set of options using an evolutionary procedure. It plans in real time with the defined space using Strategy Strategy Selection (SSS) [23].

Future of Competition
Judging from the results of the competition over the two years, there is still considerable room for improvement in order to achieve bots of human-level intelligence; hard-coded bots are reasonably easily defeated by a human player and are still scoring highly.
Throughout the competition Many other AI approaches that have performed well on related areas, for example reinforcement learning, have not yet appeared in the competition [22]. In the future these addition areas could be key to finding an optimum solution.

4 Background

Finite State Machine
State Machines can be used in the development of program interactions [25]. They can simply be viewed as [26]:

- set of input events
- set of output events
- set of states
- function mapping states and input to output
- function mapping states and inputs to states
- description of the initial state

Finite State Machines are State Machines that have a limited number of possible states it can be in at one time and is used as a development tool for approaching and solving problems [25,27]. It’s purpose is to reduce an object’s behaviour into more manageable states.
The key benefits of FSMs are:

- simple to code and debug.
- low processor time
- Flexible - easily be adjusted

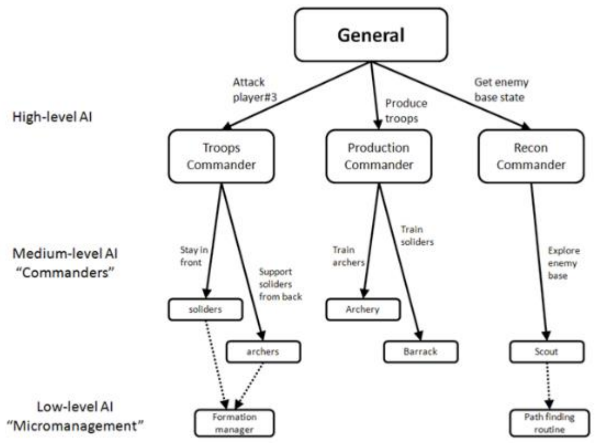
Examples and states of Finite State Machines include [27]:

- Pac-man – Evade, Chase
- Rocket – Move, TouchObject, Explode
- NPCs in RTSs – MoveToPosition, Patrol, Follow

Decision Trees

Decision trees are simple structures used to decide the optimal action from a given set of input information. Each decision is determined by the agents knowledge.
The algorithm:

1. Start at first decision (root)
2. make choice at each decision node (leaf)
3. continue till a decision cannot be made

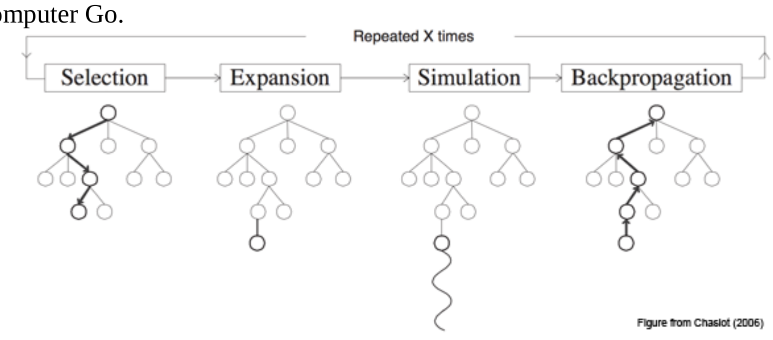


In particular, the actions in RTS games that occur are given a hierarchy in order to make real-time decisions [28]:

1. High level strategic decisions: type of unit/building to make, attack which enemy?
2. Medium level tactical decisions: deployment of a group of units.
3. Low–level micromanagement decisions: individual units actions.

Monte Carlo Tree Search

Monte Carlo Tree Search is a heuristic search algorithm that combines random simulation with the precise tree search in order to make optimal decisions. This includes move planning in combinatorial games such as MicroRTS. [29] It has the potential to be applied to a range of difficult problems, which has been proven with its success at the Computer Go.



The algorithm [29]:

1. Selection – continue to select optimal child nodes from root (node R) until a leaf (node L) is reached.
2. Expansion - create one or more child nodes from L (if it is not terminal) and select a child node (C).
3. Simulation - run a simulation from C until a result is achieved.
4. Backpropagation – use result to update the current move sequence.
Monte Carlo Tree Search is deemed beneficial over other tree searches as it is focusses its search on the more relevant areas of the tree, it is still effective with very limited knowledge of the game, search trees can be reused in the future, and implementation is very simple.

5 Techniques Implemented

An important factor to consider was which strategy is most suitable to use when the game starts After careful consideration, the EconomyRush agent was chosen mainly because it was the most balanced one compared to the rest.

EconomyRush starts off with 4 workers where 1 worker builds a base while the others collect resources, and the closest workers to the enemy, attack them. However, throughout the game, just one agent usually is not suitable to use. The current situation of the game will change throughout (i.e. the current strength and threat from the enemy in terms of workers, soldiers, distance from base and number of barracks etc.) and several techniques must be used depending on this.
In order to be able to fight the enemy in a smart way, we decided several agents were needed to be used at different times/situations, therefore, it was decided that implementing a Finite State Machine (FSM) would allow to cleverly switch between states. For this case, each state would be a different agent with particular qualities. The FSM would take into consideration factors such as number of workers, soldiers, resources, bases etc. comparing them to either the enemies. A set threshold would be picked to decide when to change between these states.
A wrapper class that models the FSM was created, consisting of an array of state concrete objects, an index to its ‘current’ state and a state transition table. An algorithm would simply compare the known information about the player and compare it to the threshold or known information about the enemy and accordingly generate a ‘message’ - which would be referring to the index in the FSM State Transition Table. The current state, and message would together determine which state to transition to next.
Unfortunately, this specific way of doing it did not seem to work on java (IntelliJ IDEA), and ended up giving errors regarding safety, therefore, it was decided that a mixture between an iterative decision tree and FSM shall be used which will terminate at the end of the game.
The states (i.e the agents) that could be changed to were as follows:

1. **EconomyRushModified:** This agent is originally coming from EconomyRush, however, it has been modified: the starting workers are now 3, the bar-

racks are built next to the base and if workers are not assigned any resources to collect, they will be sent out to attack the enemy instead of doing nothing. Bellow is the behaviour for building the barracks next to the base; followed by the workers attacking.

The building of barracks:

```
if (nbarracks == 0 && !freeWorkers.isEmpty()) {
            // build a barracks:
            if (p.getResources() >= barracksType.cost + 
resourcesUsed) {
                Unit u = freeWorkers.remove(0);
                /**
                 * Build next to base now
                 */
                buildIfNotAlreadyBuilding(u,     barrack-
sType,
                        baseXpos, baseYpos,
                        reservedPositions, p, pgs);
                resourcesUsed += barracksType.cost;
            }

        }
```

Attacking workers:

```
if (closestResource != null && closestBase != null) {
                AbstractAction aa = getAbstractAction(u);
                if (aa instanceof Harvest) {
                    Harvest h_aa = (Harvest) aa;
                    if  (h_aa.getTarget()  !=  closestRe-
source || h_aa.getBase() != closestBase) {
                        harvest(u,       closestResource,
closestBase);
                    }
                } else {
                    harvest(u,           closestResource,
closestBase);
                }
            }
            /**
             *  When  none  of  the  workers  have  a  closest
resource assigned or closest base
             * they will attack as well
             */
            else{
                    meleeUnitBehavior(u, p, gs);
                }
```

2. **SoldierRush:** This agent is originally coming from LightRush with some modifications: It will first create more light units (3) before creating ranged and then heavy. Only when it has all these unit types will it then randomise for the following trained unit. Light units are faster to create and very useful in combat, hence their priority here. Code snippet of this is bellow:

```
  int nLight = 0;
          int nRanged = 0;
          int nHeavy = 0;
          for (Unit u2 : pgs.getUnits()) {
              if (u2.getType() == lightType
                      && u.getPlayer() == p.getID()) {
                  nLight++;
              }
              if (u2.getType() == rangedType
                      && u.getPlayer() == p.getID()) {
                  nRanged++;
              }
              if (u2.getType() == heavyType
                      && u.getPlayer() == p.getID()) {
                  nHeavy++;
              }
          }

          //check if I already have some light, if not,
create.
          if   (nLight   <4   &&   p.getResources()   >=
lightType.cost) {
              train(u, lightType);
          }else if (nRanged == 0 && p.getResources() >=
rangedType.cost) {
              // check if I already have some Ranged, if
not, create
              train(u, rangedType);

          }else if (nHeavy == 0 && p.getResources() >=
heavyType.cost) {
              //check if I already have some Heavy, if
not, create.
              train(u, heavyType);
          }

          //if you already have a unit of each, randomly
select any one to generate
          if (nLight != 0 && nRanged != 0 && nHeavy != 0)
{
              int number = r.nextInt(3);
              switch (number) {
                  case 0:
                      if       (p.getResources()       >=
lightType.cost) {
                          train(u, lightType);
                      }
                      break;
                  case 1:
                      if       (p.getResources()       >=
rangedType.cost) {
                          train(u, rangedType);
                      }
                      break;
                  case 2:
                      if       (p.getResources()       >=
heavyType.cost) {
                          train(u, heavyType);
                      }
                      break;
              }
          }
```

3. **SoldierDefence:** This agent comes originally from LightDefense with some modifications as well: it will be 40% likely to create Light Soldiers, 40% Ranged Soldiers and 20% Heavy soldiers. This Agent is least likely to create heavy soldiers as they are more expensive in terms of time and resources needed. However, SoldierDefence forms a defence instead of attacking unlike SoldierRush, but SoldierDefence will also attack if the enemy gets close enough. The code for the probabilities can be found bellow:

```
public void barracksBehavior(Unit u, Player p, Physical-
GameState pgs) {
        //randomise a float and train the specific unit
        float chance = r.nextFloat();
        if    (chance<=0.40    &    p.getResources()    >=
lightType.cost)
```

```
                train(u, lightType);
            else    if    (chance>0.40    &    chance<=0.80    &
p.getResources() >= rangedType.cost)
                train(u, rangedType);
            else   if   (chance>0.80   &   p.getResources()   >=
heavyType.cost)
                train(u, heavyType);
    }
```

4. **SuicideSquad:** This final agent is selected only in a case where no bases are left, and all available workers and soldiers are sent out to attack the enemy (in other words, all-out attack). This agent is used as a last resort in the attempt to survive when resources are low, and the threat from the enemy is high. It is created from WorkerRush but with all units attacking.

The threshold for these changes regarding workers etc. would be set purely based on the size of the map, setting a fair number for the area available. Picture 1 showcases how the FSM was structured.

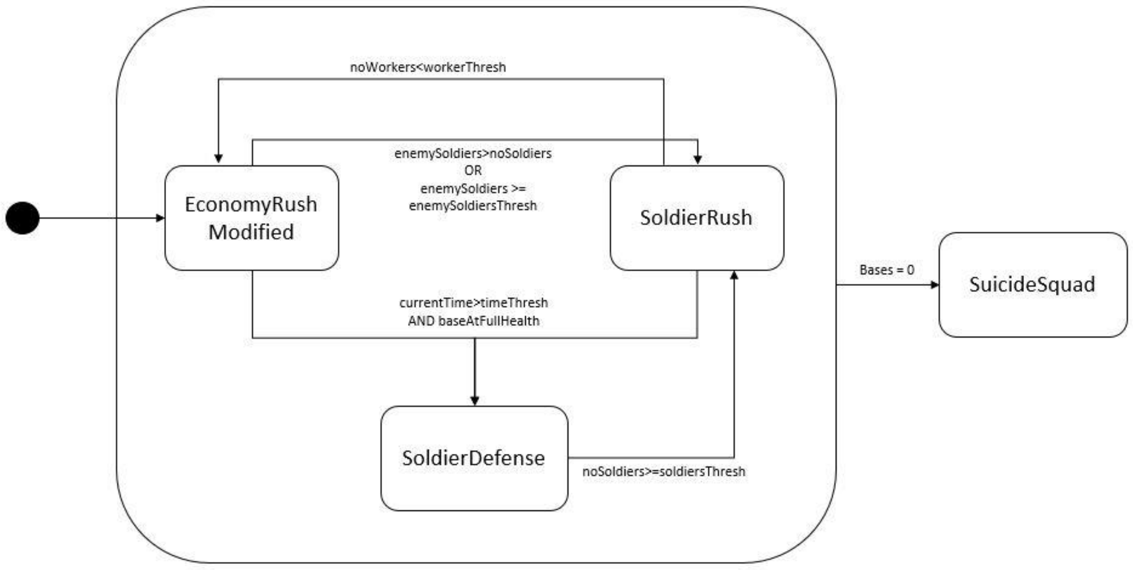


Figure 1 FSM Desgin for the switch

And the code for the change of state thus:

```
private int getState(int currentTime){
        //if we do not have bases then sends everything to at-
tack
        if (noBases == 0){
            current = 4;
        }
        else{
            switch (current) {
                case 0:
                    //when enemies have more soldiers or more
than the threshold, we attack more
                    if (enemyNoSoldiers>noSoldiers ||
                            enemyNoSoldiers   >=   enemySoldier-
sThreshold) {
                        current = 1;
                    }
                    //but if we have not been attacked for a
long time and we have more soldiers, create a defense
                    else if (currentTime>timeThreshold & baseAt-
FullHealth)
                        current = 2;
                        break;
                case 1:
                    //if we have a small amount of workers they
must have a priority
                    if (noWorkers<workerThreshold)
                        current = 0;
                    else if (currentTime>timeThreshold & baseAt-
FullHealth) //same as above
                        current = 2;
                        break;
                case 2:
                    //if we have a good amount of soldiers then
attack will all of them
                    if (noSoldiers>=soldiersThreshold){
                        current = 1;
                    }
                    break;
            }
        }
        return current;
    }
```

6 Experimental Study

To understand how good this algorithm is, a comparison had to be made between itself and other algorithms. There are many algorithms available in this framework, thus, only some – considered to be relevant – were picked. Our own algorithm is based, as previously stated, on EconomyRush, LightRush, LightDefense and WorkerRush – making them essential in this comparison. Another 2 algorithms were also introduced given their fame in the field and previous successes in other frameworks. These were NaiveMCTS and UCT. Initially, during the development, these tests were mainly run on GameVisualSimulationTest.java. This choice allowed us to carefully inspect the behaviour of the agents – both agents checked in the previous labs and the ones that came to be part of our own agent/switch – and, finally, our agent. This was also, initially, the best option for understanding strange behaviour and errors.
After, full tournaments were run with all the algorithms included. The results of these tournaments will be discussed next as they prompted a few changes in the switch to improve results. The following section also shows how the algorithm developed into the last version submitted.

7 Analysis

As stated, we developed the agent to be a switch between 4 different agents. Initially, the agents were left as they were, with only a few basic modifications (creation of all types of units for SoldiertRush and SoldierDefense).

Table 1 Tournament 1 wins

Wins\Against	QM	UCT	MCTS	ER	LR	LD	WR
QM	■	10	6	3	4	2	4
UCT	3	■	0	0	0	1	0
NaiveMCTS (MCTS)	8	16	■	11	4	3	8
EconomyRush (EC)	13	15	5	■	4	2	6
LightRush (LR)	12	15	12	12	■	10	6
LightDefense (LD)	13	14	13	14	6	■	6
WorkerRush (WR)	10	12	7	10	8	8	■

Table 2 Tournament 1 ties

Ties\Against	QM	UCT	MCTS	ER	LR	LD	WR
QM	■	3	2	0	0	1	2
UCT	3	■	0	1	1	1	4

NaiveMCTS (MCTS)	2	0		0	0	0	1
EconomyRush (EC)	0	1	0		0	0	0
LightRush (LR)	0	1	0	0		0	2
LightDefense (LD)	1	1	0	0	0		2
WorkerRush (WR)	2	4	1	0	2	2	

Table 1 and Table 2 show the results for the first tournament run and the respective wins and ties against each other. On the leftmost column we have all the algorithms in the tournament with the respective abbreviations and then how many times they've won – or tied in Table 2 – against others in the following columns. No crashes were reported or timeouts. As seen, the algorithm is performing well against UCT, but it is average with NaiveMCTS, following with worse performances against the algorithms it is based on – even if managing a few ties with them. This was surprising as it was expected to do better against its own agents – since it was, after all, a combination of these but prepared for their behaviour.
We hypothesised that, given the speed of these agents, the initial setup of workers and barrack building would have to be more efficient and faster. After all, the speed at which they start attacking is high and, upon further investigation, we realised that our agent was not coping with this mechanism. Taking this into account, we then made some changes, such as making the number of the starting workers 3 instead of 4 and building the barrack right next to the base. Followed by another tournament.

Table 3 Tournament 2 wins

Wins\Against	**QM**	**UCT**	**MCTS**	**ER**	**LR**	**LD**	**WR**
QM		15	11	8	7	4	5
UCT	1		0	2	0	0	0
NaiveMCTS (MCTS)	4	16		6	3	4	13
EconomyRush (EC)	6	14	10		4	3	6
LightRush (LR)	7	15	13	12		10	6
LightDefense (LD)	9	16	12	12	6		6
WorkerRush (WR)	10	12	3	10	8	8	

Table 4 Tournament 2 ties

Ties\Against	**QM**	**UCT**	**MCTS**	**ER**	**LR**	**LD**	**WR**
QM		0	1	2	2	3	1
UCT	0		0	0	1	0	4
NaiveMCTS (MCTS)	1	0		0	0	0	0
EconomyRush (EC)	2	0	0		0	1	0

LightRush (LR)	2	1	0	0	■	0	2
LightDefense (LD)	3	0	0	1	0	■	2
WorkerRush (WR)	1	4	0	0	2	2	■

We can see an increase of 'wins' in the tournament by our agent. It was then still winning against UCT and NaiveMCTS. It was now possible to compete with the agents it is based on as well and increasing the ties with them. Having less workers at first would not decrease the income of resources significantly and would start on the training of the units faster – allowing then a faster attack and counter attack. There might be a problem, however, from building the barracks right next to the base. Although it allows the agents to not lose as much time finding the spot and building in it, it might prove difficult for certain maps – maybe a map where that section is obstructed for example – and provide problems with mobility around the area – becomes cumbersome for some cases as well.
Finally, another change considered was related to the units themselves. Initially, we gave percentages for the units to be trained and gave higher probabilities to the less expensive ones (Light and Ranged). However, this was considered perhaps too random still – it does not make much sense to start building units that are more expensive but rather units that are less expensive and still effective. And then, build the "heavier" ones. This was the last change implemented: instead of giving the possibility of starting with heavy units, we decided it would always create at least three light units, then one ranged and one heavy. Another final tournament was run:

Table 5 Tournament 3 wins

Wins\Against	**QM**	**UCT**	**MCTS**	**ER**	**LR**	**LD**	**WR**
QM	■	14	13	9	10	8	8
UCT	2	■	0	1	0	0	1
NaiveMCTS (MCTS)	3	16	■	6	3	4	10
EconomyRush (EC)	7	15	9	■	4	4	6
LightRush (LR)	6	16	13	12	■	10	6
LightDefense (LD)	6	15	12	12	6	■	6
WorkerRush (WR)	8	11	4	10	8	8	■

Table 6 Tournament 3 ties

Ties\Against	**QM**	**UCT**	**MCTS**	**ER**	**LR**	**LD**	**WR**
QM	■	0	0	0	0	2	0
UCT	0	■	0	0	0	1	4
NaiveMCTS (MCTS)	0	0	■	1	0	0	2

EconomyRush (EC)	0	0	1		0	0	0
LightRush (LR)	0	0	0	0		0	2
LightDefense (LD)	2	1	0	0	0		2
WorkerRush (WR)	0	4	2	0	2	2	

From Table 5 and 6 we can see yet another improvement in results for our bot. In order to see the increase of won matches let's take a look at Figure 1.

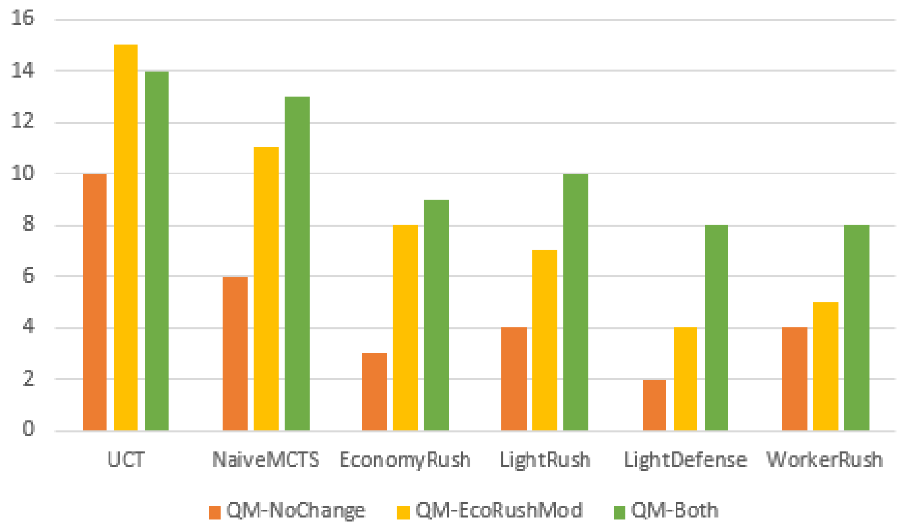


Figure 2 The number of won matches between the modified agents/switch

In this figure, we have the amount of games won from our agent against the other agents between features additions. QM-NoChange being the first one, QM-EcoRushMod the first feature added – barrack positioning and count of initial workers – and QMBoth, where the last feature was added – change on the units that are trained and amount.
We can see that overall after each modification the agent improves in its win rate. The only case where this is not shown is against UCT, however, the difference was of one game – between the agent with the change in the barracks code and the agent with both features – and this difference is not significant since the agent is still beating UCT in the clear majority of games. It also manages now to win the majority of games against EconomyRush amd LightRush (in which was heavily based on) and compete with LightDefense and WorkerRush – which tend to win against the previous, seen from the tables.
In conclusion, incorporating all units in the agents and give them a switch is not enough. It produced interesting results which could give some competition to the existing agents. However, it is very interesting to notice how simple changes such as creating one less worker at the start and changing the location of the barracks to be

closer can increase the win rate. The same applies to simply choosing which units to train first (ones that take less time). These small changes have increased the efficiency and, thus, the outcome of the games. Not every decision worked unfortunately, as stated before attempting a different way of implementing a finite state machine ended in unforeseen errors that could not be solved and randomizing units does not seem to be a good behavior either.

8 Conclusions and Future Work

In this report, we have demonstrated some of the background of our work and how we have used it as inspiration for our it. We have created an agent which acts as a switch between different agents' behaviours. It goes through several set thresholds and chooses between the most appropriate AI for that setting. We have also made some changes to the agents themselves in order to make them quicker to act and more efficient.

We believe that we have been able to demonstrate that something simple can be efficient – behaviour of professional players is complex and one strategy alone is not enough for an entire game, whichever game that may be. This work shows that a mixture between different AI behaviours, in the right circumstances, can lead to an improved version of each AI used – as some behaviours will not only be able to complement others but also counteract if playing against them. A better strategy will rely, perhaps, in how complex the different algorithms are and also how complex are the switching mechanisms between them.

However, our work is by no means complete. We have changed a few parameters such as the number of starting units, the location of the barracks, the number of each unit trained, etc. These changes, even if small in some cases, always increased the performance of the agents/switch. But many other parameters could and should be tested such as the units trained – is the number of Light units the most optimal? – or the thresholds - what are the right thresholds for the changes between agents? What happens if they are higher? Or lower? And what if the thresholds used were altogether different? Perhaps another alternative to changing into SoldierDefense through base hit points is instead focusing on the time between attacks from the enemy. Yet another possibility of changing between mostly offense and economy is also taking into account the resources left. And it is not only the thresholds that can be changed. The agents themselves could be changed – what if we create a switch that changes between more advanced algorithms such as MCTS and UCT? Or how could we combine them into the existing ones as well?

All these options present possible paths that could improve the way AI works and the combination of different strategies for the most optimal one in games.

References

1. D. Perez-Liebana, S. Samothrakis, J. Togelius, S. M. Lucas and T. Schaul, “General Video Game AI: Competition, Challenges, and Opportunities”, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)*
2. Ontañón, S., Barriga, N.A., Silva, C.R., Moraes, R.O. and Lelis, L.H., 2018. The First MicroRTS Artificial Intelligence Competition. *AI Magazine*, *39*(1).
3. Lim, C.U., Baumgarten, R. and Colton, S., 2010, April. Evolving behaviour trees for the commercial game DEFCON. In *European Conference on the Applications of Evolutionary Computation* (pp. 100-110). Springer, Berlin, Heidelberg.
4. Si, C., Pisan, Y., Tan, C.T (2014) A scouting strategy for real-time strategy games, pp. 1–8.
5. Khan,S., Kai Yang2 , Yunsheng Fu3 , Fang Lou3 , Worku Jifara1 , Feng Jiang1 , and Liu Shaohui (2018) A Competitive Combat Strategy and Tactic in RTS Games AI and StarCraft -
6. 1. Robertson, G., Watson, I.: A review of real-time strategy game AI. AI Mag. 35(4), 75–104 (2014)
7. S. Ontanon (2013) “The combinatorial multi-armed bandit problem and its application to real-time strategy games,” in Ninth Artificial Intelligence and Interactive Digital , 2013, pp. 58–64.
8. https://sites.google.com/site/micrortsaicompetition/home

9.]Santiago Ontañón (2013) "The Combinatorial Multi-Armed Bandit Problem and its Application to Real-Time Strategy Games", In AIIDE 2013
10. Shleyfman, Alexander, Antonín Komenda, and Carmel Domshlak. "On combinatorial actions and CMABs with linear side information." ECAI. 2014.
11. Komenda, Antonín, Alexander Shleyfman, and Carmel Domshlak. "On Robustness of CMAB Algorithms: Experimental Approach." Workshop on Computer Games. Springer
12. Santiago Ontañón and Michael Buro (2015) Adversarial Hierarchical-Task Network Planning for Complex Real-Time Games, in IJCAI 2015.
13. Santiago Ontañón (2016) "Experiments on Learning Action Probability Models from Replay Data in RTS Games". In AIIDE 2016 Workshop on Artificial Intelligence in Adversarial Games.
14. Santiago Ontañón (2016) "Informed Monte Carlo Tree Search for Real-Time Strategy Games", in proceedings of IEEE-CIG 2016.
15. Marius Stanescu, Nicolas A Barriga, Andy Hess, Michael Buro (2016) "Evaluating Real-Time Strategy Game States Using Convolutional Neural Networks", in proceedings of IEEE-CIG 2016.
16. Santiago Ontañón (2017) Combinatorial Multi-Armed Bandits for Real-Time Strategy Games. In Journal of Artificial Intelligence Research (JAIR)
17. Alberto Uriarte and Santiago Ontañón (2017) Single Believe State Generation for Partially Observable Real-Time Strategy Games. In Proceedings of IEEE-CIG 2017

18. NA Barriga, M Stanescu, M Buro (2017) Game Tree Search Based on Non-Deterministic Action Scripts in Real-Time Strategy Games. In IEEE Transactions on Computational Intelligence and AI in Games 2017
19. Santiago Ontañón, Nicolas A. Barriga, Cleyton R. Silva, Rubens O. Moraes, Levi H. S. Lelis (2017) The First MicroRTS Artificial Intelligence Competition
20. https://github.com/santiontanon/microrts/wiki/Artificial-Intelligence
21. http://www.planrec.org/Geib/Papers_files/PID5439751.pdf
22. https://sites.google.com/site/micrortsaicompetition/competition-results
23. https://github.com/jr9Hernandez/TiamatBot
24. Buro,M. 2003. Real-timestrategygames:anewAIresearch challenge. In Proceedings of the IJCAI, 1534–1535. Citeseer.
25. https://whatis.techtarget.com/definition/finite-state-machine
26. Bran Selic & Garth Gullekson: Real-time Object-oriented Modeling
27. http://www.ai-junkie.com/architecture/state_driven/tut_state1.html
28. Josh MaCoy and Michael Mateas. 2008. An Integrated Agent for Playing Real-Time Strategy Games by . In Proceedings of the 23rd national conference on Artificial intelligence.
29. http://mcts.ai/index.html