ECS7002P Assignment 1 Report - Group G

Bo Peng, Yunyao Liu, Jiageng Li

November 1, 2018

Abstract

This paper presents the outcome of Group G of the microRTS group project. The microRTS is a full-observed real-time strategy game which is build on grid map within different scale and condition. The aim of our group work is to construct a battle bot with specific strategy which can control the units on the map automatically, based on the current state of the game, make the real-time decision and command and assign the action to the units to beat the opponent on the map finally. This report will introduce the background knowledge of the game, interpret the technical details of the bot we created and analyse the testing result respectively.

1 Introduction

In the past few years, significant progress has been made in stronger abstract game playing agents, the artificial intelligence (AI) controller of game agent is much more stronger to be able to beat the top professional human players such as Alpha Go, while the Real-Time Strategy (RTS) game is different and more complicated from abstract game as chess, the actions that different game agents assigned could be simultaneous, in some situations the player only have the partial vision and size of game state and game area, due to the game state changes at per game time, the available and possible order actions of each game units are limited in different game state. Every imperceptible change of game state could result in a totally different game ending. Because of those factors that is difficult and complex for artificial intelligence host to judge and make decision of the game, human players are usually stronger than the AI bots.

The bot that our group created is based on a simple type of RTS game which called microRTS, microRTS is a two-player game build on several different grid map, each player of the game could control units and assign actions to them for the final goal of destroying each units of the opponent. In microRTS, the basic elements include resource, building and moving units, the resource is the initial immovable units on the map, which could be harvested to producing new units. The building units include base and barrack

which are fixed on the map, base and resource are both the initial units of the game, base could produce worker and barrack could produce other attack units. The moving units include worker, light, heavy and ranged, worker can only be produced by base and barrack produced other three types of units. The duty of worker is to harvest resource and build new buildings, worker does have the attack ability but the damage is low. Other three attack units which produced by barrack divided into light, heavy and ranged. As the name suggests, light has high movement speed and considerable damage points, heavy has the highest damage point and hit point but the lowest movement speed, ranged has lower hit points but the maximum attack range.

Although the microRTS has the reduced units and simpler game state than other complex RTS game as StarCraft, the decision space is still huge. The game map is rectangular grid-based map of arbitrary size, each grid represent a movement unit. The initial game state is variant that resource and base could be anywhere, sometimes the resource might separate the map to two parts, each side set up a player, which means the player could not get to their opponent directly. In some initial map the resource is lack, some there is an built-in barrack near the base. While those are just the complexity of the initial state, if the initial game state is set as the root of the decision tree, each change of the game state will increase the branch of the decision tree with the game goes on. In order to defeat the opponent, it is necessary to build a bot strategy that adapts the volatile situation. The aim of our group is to construct such a bot strategy which is strong enough to beat the opponent of variant game state, the bot we create using the decision tree to judge the game state and make actions. Through the test, the bot we create is able to defeat all the built-in bot on four given testing map.

2 Literature Review

Previous research in μRTS has focused on areas such as Monte Carlo tree search (MCTS); adversarial search algorithms that perform search at some level of abstraction, such as Puppet Search or adversarial HTN planning; and even deep learning for RTS games[4].

In 2017, there was a AI competition that hosted at the IEEE Computational Intelligence in Games (CIG) within the base of microRTS, the goal of that competition is to search for a better way of artificial intelligence techniques to handle Real-Time Strategy games. There were three submissions that received by 2017 competition: StrategyTactics[1], Strategy Creation through Voting (SCV)[5], and BS3NaÖveMCTS[6]. The StrategyTactics uses the strategic strength of PuppetSearch and tactical performance of NaÖveMCTS to achieve algorithms which performs well of making low-level

decisions of complex game state with higher branching factors or long-term actions consequence. SCV uses a set of built-in strategies to create different and new strategies to adapt changeable game state. BS3Na0̈veMCTS is an extension of Na0̈veMCTS to deal with the situation that when the map is partial observed in RTS games. However, the bot strategy that our group built draws on the advantages of the first two approaches, using decision tree to judge the game state and implement a portfolio build-in strategy for different situations.

3 Benchmark

Our tactical bot extends built-in class 'WorkerRushPlusPlus', the reason will be demonstrated later in details, we override some methods and changed the 'PlayerAction' behaviour, at last we would test our bot with built-in strategies to measure the effectiveness.

There were few difficulties during building our intelligent bot, the most awkward one is the path-finding problem. In our pre-defined behaviour tree, the bot has to judge whether there is an available road to destination, there was an existing method of path-finding, this method bases on a moveable unit to find the way, but in the map 'NoWhereToRun9x8', there is only one base at beginning, which always leads to the wrong judgement of path-finding state and the bot behaviour did not meet our expectation. So we rewrote the path-finding method in our code, considering the road between two coordinates existing or not. Then the problem was fixed.

Another problem appeared during the test of map 'NoWhereToRun9x8'. In our strategy, the barrack would keep training attack units until the road to enemy is clear, while the barrack would crashed when it is surrounded by units. So we changed the rules of units 'Standby', make them would move to nearest grid to make the way when they 'notice' they are near the barrack or other unit.

One of the most important factor of win the battle is finance. In this game it represents the harvesting efficient, a large amount of workers is not necessarily the best to the speed of obtaining resource. In some situations too many workers might block the road, sometimes producing workers would overuse resource of training attack units. So a balance relation between worker amount and game state is necessary. In our strategy, we consider the minimum value between base amount times 2 and resource unit amount times 1.5 as worker amount. But some times the resource unit might at corner or surrounded by other resource, which could leads to wrong judgement and produce redundant workers then reduce efficiency. We solved this problem by using the path-finding method we rewrote before to judge whether the worker is able to get to the resource first to count the available resource, then all the workers are on their duty.

4 Background

4.1 Decision Tree

A decision tree is a decision support tool that uses a tree-like graph or model of decisions and their possible consequences. A diagram of decision tree structure is shown as Figure 1.

A decision tree is composed of decision nodes, start of the first decision (its root). There will be at least two decision options after each judgement. Decisions are made by internal knowledge of agents, for example parameters or constraints.

At the beginning of a decision tree from the root, the tree will start to decide which option to make next step by estimating current state value, and continues making decisions and choices at each node until it meets a termination or there are no more decisions to make in this branch of tree.

As it shown in Figure 1. 'Shell weight' is the decision criterion which is incorporated in each node until it reaches the termination or last state.

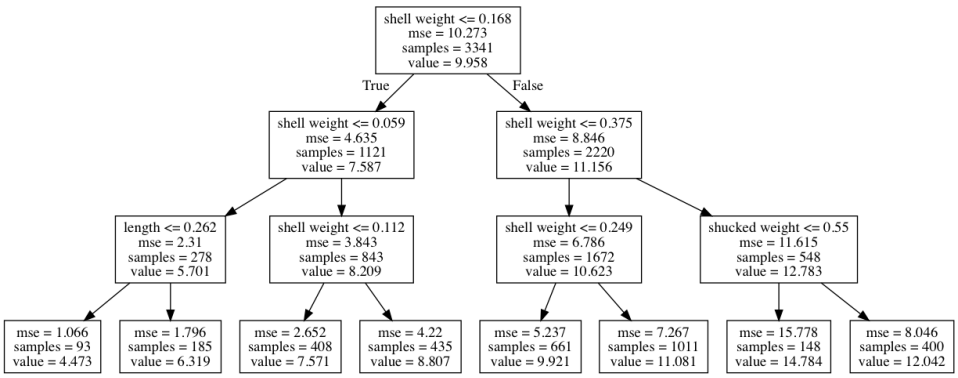

Figure 1: Decision Tree

Decision tree can also control the behavior of the system in a simplifier way, by making decisions in script, we can control the leaf node where the system will reach finally, as it shown in Figure 2, which is a decision tree of script applied to RTS game[2], it controls the flow of the game state: At the beginning game state will reach 'Gather Resources' leaf since that all game starts with one defensive buildings. After gathering enough resources, which will lead to 'Build defenses' leaf, it starts to 'Expand' its base number. After enough bases was settled, the decision tree will lead the system to take some offensive actions, first is to 'Train Soldiers'. As we can see from Figure 2, according to decision criterion, 'Attack' is final action that the game system will take.

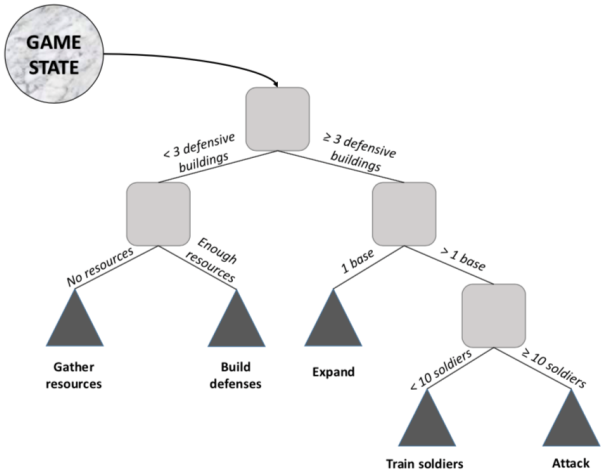

Figure 2: Game Tree Search[2]

4.2 Behavior Tree

A behavior tree is a model of plan execution that is graphically represented as a tree[3]. A simple structure of Behavior Tree is shown as Figure 3. A Behavior Tree is a mathematical model, which consists of actions and nodes. Behavior Trees can control the order and priority of actions by using different types of nodes and can represent clearly a framework of a program. A Behavior Tree is often composed of 4 types of typical nodes, all of them act different functions operating the tree. The four main types of nodes are illustrated as below.

Composite Node

Composite Node includes Selector Nodes, Sequence Nodes and Parallel Nodes.

A Selector node will act as an OR operand when used in a Behavior Tree, when it begins, all child nodes of this Selector node will be executed iteratively, if one of them is successfully executed, it will return a TRUE Boolean value, if not, a False Boolean value will be returned and passed back to Selector node;

A Sequence Node can act as an AND operand when it is used in a Behavior Tree, when it begins, all child nodes of this Sequence node will be executed iteratively, only if all child nodes are executed successfully, it will return a TRUE Boolean value and passed it to Sequence Node;

Parallel Nodes provide a method to execute many nodes simultaneously.

Decorator Node

When Decorator Node is added in a Behavior Tree, it adds some extra actions to the return value received from its child nodes and then pass the new value to the root.

Condition Node

Act in Behavior Tree very simple, it will return a TRUE Boolean value only if the condition is satisfied, very similar to nodes in a Decision Tree nothing but a Condition Node requires a return value.

Action Node

Node that can specifically complete an action or a movement, an Action Node also requires return value.

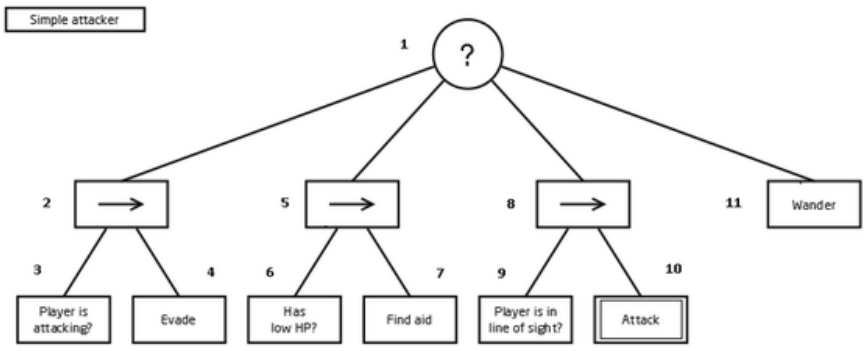

Figure 3: Behavior Tree[3]

4.3 Breadth-First Search (BFS)

Breadth-First Search is an algorithm that uses brute way to search all nodes first before it takes any action of exploring, so it is an uninformed or blind algorithm of searching and does not know any goal information.

Breadth-First Search is very simple, very easy to implement in algorithm but it requires lots of memory occupation since it explores all nodes until find a termination or a goal.

5 Techniques Implemented

The main parts of our controller are shown as below, I will introduce them combined with our code to you including techniques they used and how they work respectively. Some illustration is shown as code comments.

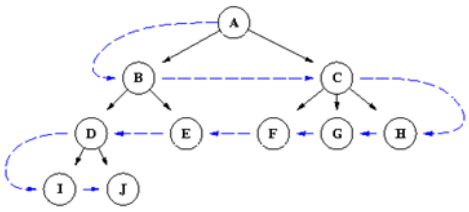

Figure 4: BFS

5.1 BFS Path Finding Strategy

In our controller, we used Breadth-First Search to get to our target from a start point in range.

```
 /*
    BFS path finding from (startX, startY) to (targetX, targetY) in range.
    Methods in the PathFinding.class require a unit, this method does not.
 */
protected boolean PathExistsInRange(int startX, int startY, int targetX,
        int targetY, int range, PhysicalGameState pgs) {
    boolean visited[][] = new boolean[pgs.getWidth()][pgs.getHeight()];
    for (int i = 0; i < pgs.getWidth(); ++i)
        for (int j = 0; j < pgs.getHeight(); ++j)
            visited[i][j] = false;
    for (Unit u : pgs.getUnits())
        if (u.getType().isResource)
            visited[u.getX()][u.getY()] = true;
    int queX[] = new int[pgs.getWidth() * pgs.getHeight() + 1],
        queY[] = new int[pgs.getWidth() * pgs.getHeight() + 1];
    int head = 0, tail = 1;
    queX[0] = startX;
    queY[0] = startY;
    visited[startX][startY] = true;
    while (head < tail) {
        for (int i = 0; i < 4; ++i) {
            int nextX = queX[head] + conjX[i],
                nextY = queY[head] + conjY[i];
            if (nextX >= 0 && nextX < pgs.getWidth() && nextY >= 0 &&
                    nextY < pgs.getHeight() && !visited[nextX][nextY]) {
                if (ManhattanDistance(nextX, nextY, targetX, targetY) <= range)
                    return true;
                queX[tail] = nextX;
                queY[tail] = nextY;
                visited[nextX][nextY] = true;
                ++tail;
            }
        }
        ++head;
```

```
    }
    return false;
}
```

5.2 Melee Units Standby Strategy

This method called 'MelleeStandby', it has the functions that: Attack enemies near the melee unit; Try not to block the way, blocking way can cause game crush; And a standby function. I will introduce this function below, some illustration is shown as code comments.

```
protected void MeleeStandby(Unit melee, List<Unit> enemies, PhysicalGameState pgs) {
    boolean noEnemyAround = true;
    for (Unit enemy : enemies)
        // attack nearby enemies
        if (ManhattanDistance(melee, enemy) <= 1 + max(melee.getType().attackRange,
                enemy.getType().attackRange)) {
            attack(melee, enemy);
            noEnemyAround = false;
            break;
        }
    if (noEnemyAround) {
        int emptyDirection[] = {-1, -1, -1, -1};
        int emptyDirectionNum = 0;
        boolean conjWithBarrack = false;
        for (int i = 0; i < 4; ++i) {
            int X = melee.getX() + conjX[i];
            int Y = melee.getY() + conjY[i];
            if (X >= 0 && X < pgs.getWidth() && Y >= 0 && Y <= pgs.getHeight()) {
                Unit u = pgs.getUnitAt(melee.getX() + conjX[i], melee.getY() + conjY[i]);
                if (u == null)
                    emptyDirection[emptyDirectionNum++] = i;
                else if (u.getType() == barrackType)
                    conjWithBarrack = true;
            }
        }
        // try not to block the way (especially the barrack)
        if (emptyDirectionNum > 0 && (emptyDirectionNum < 3 || conjWithBarrack)) {
            int direction = r.nextInt(emptyDirectionNum);
            move(melee, melee.getX() + conjX[direction], melee.getY() + conjY[direction]);
        }
    }
}
```

5.3 When and Where to Build Bases

'BuildBase' method, which builds a new base near a resource that no bases around it.

```
protected void BuildBase(Unit worker, List<Unit> freeResource, Player p,
        PhysicalGameState pgs) {
    int X = -1, Y = -1;
    for (Unit target : freeResource) {
        // find an empty position
        for (int i = 0; i < 16; ++i) {
            X = target.getX() + roundX[i];
            Y = target.getY() + roundY[i];
            if (X >= 0 && Y >= 0 && X < pgs.getWidth() && Y < pgs.getHeight() &&
                    pgs.getUnitAt(X, Y) == null) {
                break;
            } else {
                X = -1;
                Y = -1;
            }
        }
        if (X >= 0)
            break;
    }

    List<Integer> reservedPositions = new LinkedList<>();
    if (X >= 0) {
        // build new base
        buildIfNotAlreadyBuilding(worker, baseType, X, Y, reservedPositions, p, pgs);
        resourceUsed += baseType.cost;

        // remove resources nearby the new base from the freeResource list
        LinkedList<Unit> removingResource = new LinkedList<>();
        for (Unit resource : freeResource)
            if (ManhattanDistance(resource, X, Y) <= resourceRange)
                removingResource.add(resource);
        for (Unit resource : removingResource)
            freeResource.remove(resource);
    }
}
```

5.4 Train Workers

'BasesBehavior' method: train workers if the current number of workers is lower than 'workerLimit', otherwise do nothing, this method acts as a Decorator in our system, which will determine what type of worker to train depends on parameter value.

```
protected void BasesBehavior(List<Unit> baseList, List<Unit> workerList,
        int resourceNum, Player p, GameState gs) {
    int workerLimit = min((int)(resourceNum * workersForEachResource),
            baseList.size() * workersInEachBase);
    for (Unit base : baseList)
        if (gs.getActionAssignment(base) == null && workerList.size() < workerLimit &&
```

```
                p.getResources() - resourceUsed >= workerType.cost) {
            train(base, workerType);
            --workerLimit;
            resourceUsed += workerType.cost;
        }
    }
```

5.5 Workers Behavior

'WorkersBehavior' method. This method combines Decision Tree and Behaviour Tree to realize these functions:

1. Try to build new bases if there are free resources;
2. Try to build new barracks if the current number of resources is equal to or higher than 'newBarracksThreshold';
3. Harvest resources with no more than 'workerLimit workers';
4. After 1, 2, 3, workers with no mission act as melee units.

```
protected void WorkersBehavior(List<Unit> baseList, List<Unit> workerList,
        List<Unit> ourResource, List<Unit> freeResource, List<Unit> meleeList,
        int barrackNum, Player p, GameState gs) {
    PhysicalGameState pgs = gs.getPhysicalGameState();
    int workerLimit = min((int)(ourResource.size() * workersForEachResource),
            baseList.size() * workersInEachBase);
    for (Unit worker : workerList)
        if (gs.getActionAssignment(worker) == null) {
            // build new base
            if (!freeResource.isEmpty() && p.getResources() - resourceUsed
                    >= baseType.cost)
                BuildBase(worker, freeResource, p, pgs);
            // build new barrack
            else if ((barrackNum == 0 || p.getResources() - resourceUsed
                    >= newBarracksThreshold) &&
                    p.getResources() - resourceUsed >= barrackType.cost)
                BuildBarrack(worker, baseList, p, pgs);
            else if (workerLimit > 0) { // Harvest
                WorkerHarvest(worker, baseList, ourResource);
                --workerLimit;
            } else {    // free workers as melee units
                meleeList.add(worker);
            }
        }
}
```

5.6 Statistics and Summary

'getAction' method. Also combines Decision Tree and Behavior Tree to determine units' actions.

```java
public PlayerAction getAction(int player, GameState gs) {
    PhysicalGameState pgs = gs.getPhysicalGameState();
    Player p = gs.getPlayer(player);
    PlayerAction pa = new PlayerAction();

    int enemyPlayer = 1 - player;
    LinkedList<Unit>
        baseList = new LinkedList<>(),
        barrackList = new LinkedList<>(),
        workerList = new LinkedList<>(),
        meleeList = new LinkedList<>(),
        enemyMovableUnits = new LinkedList<>(),
        enemyBuildings = new LinkedList<>(),
        enemyAroundBase = new LinkedList<>();

    // Statistics
    for (Unit u : pgs.getUnits())
        if (u.getPlayer() == player)
            if (u.getType() == baseType)
                baseList.add(u);
            else if (u.getType() == barrackType)
                barrackList.add(u);
            else if (u.getType() == workerType)
                workerList.add(u);
            else
                meleeList.add(u);
        else if (u.getPlayer() == enemyPlayer)
            if (u.getType().canMove)
                enemyMovableUnits.add(u);
            else
                enemyBuildings.add(u);

    // Find enemies that are around our bases
    List<Unit> ourBuildings = new LinkedList<>(baseList);
    ourBuildings.addAll(barrackList);
    for (Unit enemy : enemyMovableUnits)
        for (Unit base : baseList)
            if (ManhattanDistance(enemy, base) <= baseAlarmDistance) {
                enemyAroundBase.add(enemy);
                break;
            }

    if (enemyAroundBase.isEmpty()) {    // no enemy around our bases
        ArrayList<Unit> enemyUnits = new ArrayList<>(enemyMovableUnits);
        enemyUnits.addAll(enemyBuildings);

        // find if there is a path to enemies
        boolean pathToEnemyExists = false;
        if (!enemyUnits.isEmpty() && !baseList.isEmpty())
```

```
50              pathToEnemyExists = PathExistsInRange(
51                      baseList.getFirst().getX(), baseList.getFirst().getY(),
52                      enemyUnits.get(0).getX(), enemyUnits.get(0).getY(), 1, pgs);
53
54          // find resources that are around our bases and around no bases
55          LinkedList<Unit> freeResource = new LinkedList<>();
56          LinkedList<Unit> ourResource = new LinkedList<>();
57          for (Unit u : pgs.getUnits())
58              if (u.getType().isResource) {
59                  boolean baseAroundResource = false;
60                  for (Unit v : pgs.getUnitsAround(u.getX(), u.getY(), resourceRange))
61                      if (v.getType() == baseType) {
62                          baseAroundResource = true;
63                          if (v.getPlayer() == player &&
64                                  PathExistsInRange(v.getX(), v.getY(),
65                                                    u.getX(), u.getY(), 1, pgs))
66                              ourResource.add(u);
67                          break;
68                      }
69                  if (!baseAroundResource)
70                      freeResource.add(u);
71              }
72
73          if (pathToEnemyExists) {  // Exist a path to enemy
74              if (barrackList.isEmpty()) {
75                  return super.getAction(player, gs);
76              } else {
77                  resourceUsed = 0;
78                  BasesBehavior(baseList, workerList, ourResource.size(), p, gs);
79                  WorkersBehavior(baseList, workerList, ourResource, freeResource,
80                          meleeList, barrackList.size(), p, gs);
81                  BarracksBehavior(barrackList, meleeList, 8, 2, 3, p);
82
83                  // Melee Behavior
84                  if (ourResource.isEmpty() || meleeList.size() >= attackThreshold)
85                      AttackClosestEnemy(meleeList, enemyUnits);
86                  else
87                      for (Unit melee : meleeList)
88                          MeleeStandby(melee, enemyUnits, pgs);
89              }
90          } else {  // No path to enemy
91              resourceUsed = 0;
92              BasesBehavior(baseList, workerList, ourResource.size(), p, gs);
93              WorkersBehavior(baseList, workerList, ourResource, freeResource,
94                      meleeList, barrackList.size(), p, gs);
95              BarracksBehavior(barrackList, meleeList, 0, 0, 1, p);
96
97              // Melee Behavior
98              for (Unit melee : meleeList)
```

```
                MeleeStandby(melee, enemyUnits, pgs);
            }
        } else {    // have enemy around our bases
            AttackClosestEnemy(meleeList, enemyAroundBase);
            AttackClosestEnemy(workerList, enemyAroundBase);
        }

        return translateActions(player, gs);
    }
```

6 Experimental Study

Before we design the behaviour tree of our intelligent battle bot, we have conducted several test between the built-in strategies to decide which strategy is better within specific game state and customize our unique strategy for the bot to achieve better performance on the battle field. The built-in bot we have tested consist of WorkerRush (WR), WorkerDefence (WD), WorkerRushPlusPlus (WR++), HeavyRush (HR), HeavyDefence (HD), LightRush (LR), LightDefence (LD), RangedRush (RR), RangedDefence (RD) and SimpleEconomicRush (SER). First we test the map of 'basesworkers16x16', the competition was set between different Rush strategies(exclude Defence strategies), the testing result is shown in the Table 1.

Table 1: Test results on 'basesworkers16x16'. '1' represents row unit wins and '0' represents column unit wins

vs	WR	WR++	HR	LR	RR	SER
WR	-	1	0	0	0	0
WR++	0	-	0	0	0	0
HR	1	1	-	0	0	1
LR	1	1	1	-	0	1
RR	1	1	1	1	-	1
SER	1	1	0	0	0	-

From the Table 1, we noticed that on this battle map the built-in bot with 'WorkerRushPlusPlus' strategy presents the highest combat effectiveness. The difference between strategy 'WorkRush' and 'WorkRushPlusPlus' is the harvest efficient, result in 'WorkerRushPlusPlus' perform better than 'WorkRush' on this map. The testing result is similar on the map 'baseworkers24x24H', the reason we speculate is although the size of battle map is expanded, the transformation is not enough to turn the table. Through the observation of fighting procedure, we noticed that the scale and initial condition such as the number of resource and whether there exists barracks near the base at the begin of the map are major cause of why 'WorkerRush-

PlusPlus' perform so good. Even though worker is definitely not the best choice to fight, worker still has the attack ability, with the advantages of high movement speed and low production cost of resource and time, a flood of workers could be an army with destructive force in the map with small scale, with the observation of battle between 'WorkerRush' and strategies which have priority to build barracks when there is no barrack such as 'SimpleEconomicRush', the workers always get to the opponent's buildings and cause damage before the opponent's barrack construction complete. The situation changed in a map of larger scale, if the worker failed to destroy opponent's barracks or bases before the first training of attack units as heavy, light or ranged finished, the situation is reversed, 'WorkerRush' bot would be defeated finally. From the above, the strategy 'WorkerRushPlusPlus' is invoked by our tactical bot after the judgement of map scale and barracks state, if map size is not large enough and no barracks at beginning, do 'WorkerRushPlusPlus'.

We run the test between 'Rush' and 'Defence' strategies, the situation did not change significantly of the same type strategy and the 'Rush' bots performed better, so the 'Defence' strategies were abandoned.

Then we switched map to 'TwoBasesBarracks16x16' and run the test, the result is shown on the Table 2.

Table 2: Test results on 'TwoBasesBarracks16x16'. '1' represents row unit wins and '0' represents column unit wins

vs	WR	WR++	HR	LR	RR	SER
WR	-	1	1	1	1	0
WR++	0	-	1	1	1	0
HR	0	0	-	0	0	0
LR	0	0	1	-	0	0
RR	0	0	1	1	-	0
SER	1	1	1	1	1	-

From the result we noticed as prediction, the 'Worker' strategies were utterly routed. The reason is apparently, there are two initial barracks of each player on the map, only training worker is not enough to beat enemy. On this map, we noticed that 'HeavyRush' is the strongest bot, which means heavy unit is stronger than other two types attack units. But there is a strange phenomenon that the strategy 'SimpleEconomicRush' failed to beat any bot include 'Worker' bot, which is unexpected. Though analysing the code structure and battle procedure, we discovered that one rule of 'SimpleEconomicRush' is that the worker amount depends on the base amount, which is 3 times of base amount. On the map 'TwoBasesBarracks16x16', there exist two bases of each player, leads to the bot of 'SimpleEconomicRush' would create 6 workers, finally such amount of workers have possi-

bilities to block the road between base and resource, then the whole action gets stuck and bot crashed. In order to solve this problem and test the real strength of 'SimpleEconomicRush', we run the test on previous map which only have one base and no barracks for each player and get rid of 'Worker' strategies(The reason why eliminating 'Worker' is that create a condition that each player has barracks, but 'Worker' strategies are incapable to build barracks), the situation is changed, 'SimpleEconomicRush' become the best bot for beating any other 'Rush' bot. Through analysing, the strength of 'SimpleEconomicRush' including the higher harvesting efficient and variant attack units portfolio. So when building our tactical bot, under the condition of existing barracks on map, we simulated the behaviour of 'SimpleEconomicRush' and fixed the relation between worker amount and base amount to avoid stuck and crash problems.

The final map we tested is 'NoWhereToRun9x8'. In this map, the resource split the map into two parts, each player located in one side, which means the moveable unit is unable to reach opponent's units or buildings directly. The only way to get through the path is waiting for the resource unit is exhausted. The testing result is shown at the Table 3.

Table 3: Test results on 'NoWhereToRun9x8'. '1' represents row unit wins and '0' represents column unit wins

vs	WR	WR++	HR	LR	RR	SER
WR	-	1	1	1	1	1
WR++	0	-	1	1	1	1
HR	0	0	-	1	1	1
LR	0	0	0	-	1	1
RR	0	0	0	0	-	1
SER	0	0	0	0	0	-

From the result we noticed that 'SimpleEconomicRush' still perform best. But there was one thing caught our attention, the ranged units shown unexpected battle effectiveness on this map, ranged units have the largest attack range, which make them able to ignore the terrain to cause damage. Because the resource divide the map into two parts, heavy and light units is unable to get close to ranged but ranged could attack them. So when we design our strategy, we set up a rule that when the path.

7 Analysis

I want to introduce our development process using a flowchart combined with illustration.

The first thing we considered about our strategy is the search algorithm,

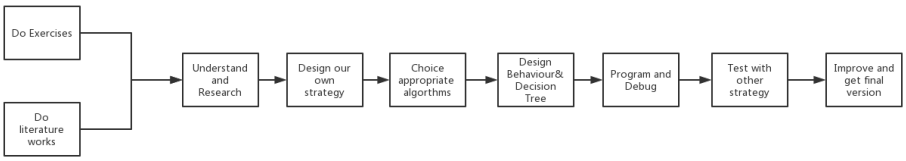

Figure 5: Flow Chart of Development Process

taking that all possible point positions are not that many into consideration, it will not require much memory space than using DFS. Also using BFS pathfinding strategy is much quicker. Also, we did not use any informed search algorithms so that we can diminish computation time and react quicker, since all informed algorithms will need more time to calculate the best combination of actions.

So the main goal of implementing BFS algorithm is to reduce the computation time of search based on acceptable non-optimal algorithms, and arrange actions as fast as possible instead of using search algorithms to calculate combination of actions to find optimal moves. By controlling the choices points of each unit we created in uRTS, we can meet the given time constraints and results in a quicker reaction and a more simplified AI system. Assuming that we are creating a quickly reactive system in uRTS game. In this case we will allow scripts to expose only a few carefully chosen choice points, if at all, resulting in fast searches that may sometimes miss optimal moves, but generally produce acceptable action sequences quickly[2]. So, our bot is based on decision tree which can minimize the effect of RTS constraints so that the bot can gener-ate sequence of actions very quickly. Meanwhile, to maximize our advantage of 'quickly generated action sequence', we guarantee that every decision made will lead to a certain action.

We compared all hard-coded strategies, including WorkerRush, LightRush, Heavy-Rush and RangedRush, and then tested them against each other, we found that WorkerRush is quickest rush strategy and it remain a high wining rate among all strategies, so we decided to improve it by using our own Tree. When there is no more branch of tree, algorithm will turn to another state or branch to continue executing decisions. Decisions of each node are made by value of parameters in our program, decision tree is quite fast and easy to implement in deciding which action an unit should do during the configuration of our agent.

The Tree is shown as Figure 6.

As you can see from the Figure 6, our Tree is more like a Decision Tree combined with a Behaviour Tree, some nodes represent decision conditions and others represent actions. The required returned value depends on what kind of action the nodes take and the unit condition.

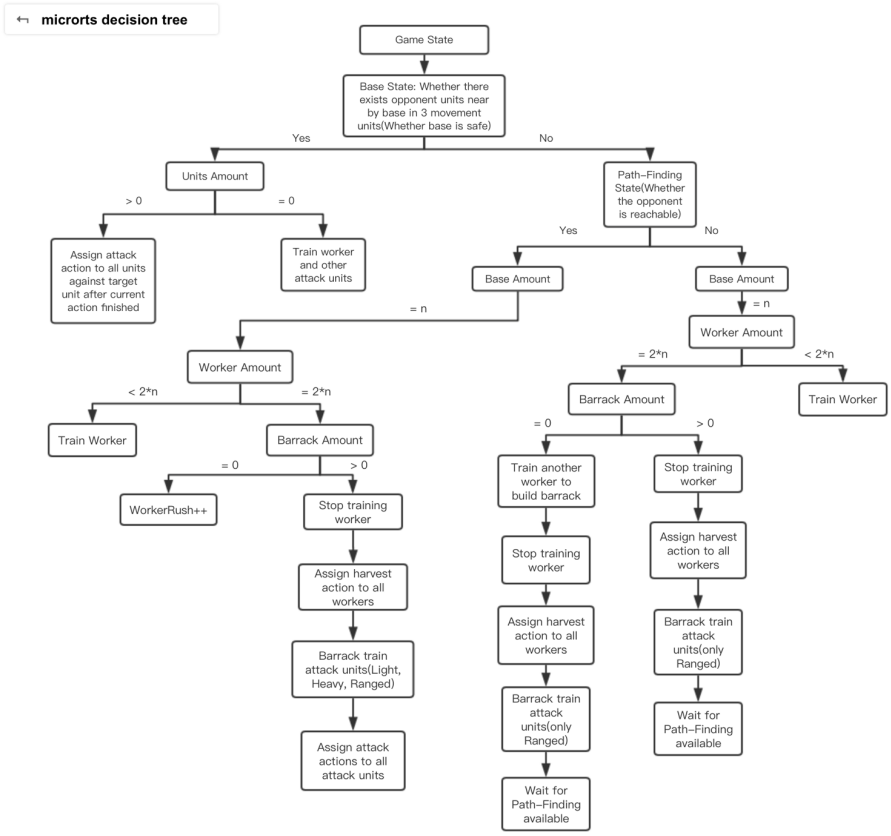

Figure 6: Decision Tree of Group G

At the beginning node (root) of the tree, we will consider the security of our base, if there are any enemy units around it within 3 movements, if the base is not safe, the system will assign units to attack enemies until they all dead. Otherwise, when the base is safe (There are more than 3 movements between our base), the system will do options depends on different types of map.

So, when the system confirm base is in a safe condition, the system will decide which map we are using now according to the Barrack Amount and if opponents are reach-able (Opponents are not reachable in the map with resources blocked in the middle). There are four different types of maps, so we decide different strategies to reply to them.

The first two types of maps are very similar but has different size and resource, so we directly train worker units until the number fits our condition, then control all workers to attack.

When it comes with map with Barracks, we will assign harvest action to all workers and train all kinds of attack units. Then we assign attack

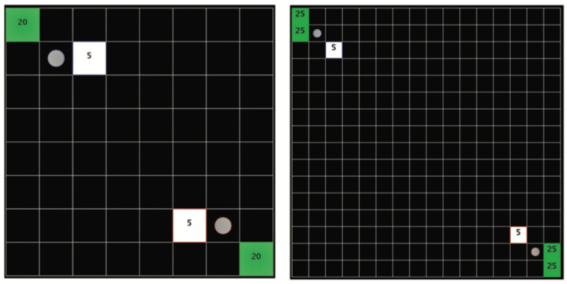

Figure 7: First Two Maps

actions to units.

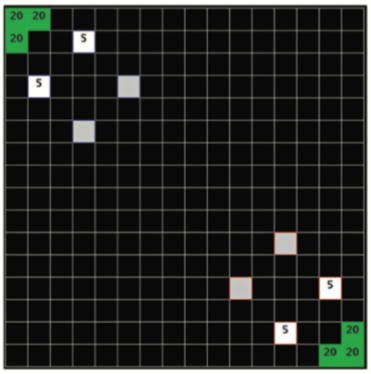

Figure 8: Map with Initial Barracks

Last is the type of map that opponents are not reachable at the beginning, at this situation we will execute a different strategy: First we train workers and let them build Barracks, after that let workers harvest to get resources. Then build Ranged Attack Units to attack. In this way, our system can do it best to train Ranged Attack Units, which has huge advantage in this map.

References

[1] Nicolas A Barriga, Marius Stanescu, and Michael Buro. Combining strategic learning and tactical search in real-time strategy games. *arXiv preprint arXiv:1709.03480*, 2017.

[2] Nicolas A Barriga, Marius Stanescu, and Michael Buro. Game tree search based on nondeterministic action scripts in real-time strategy games. *IEEE Transactions on Games*, 10(1):69–77, 2018.

[3] Ryan Marcotte and Howard J Hamilton. Behavior trees for modelling artificial intelligence in games: A tutorial. *The Computer Games Journal*, 6(3):171–184, 2017.

[4] 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.

[5] Cleyton R Silva, Rubens O Moraes, Levi HS Lelis, and Kobi Gal. Strategy generation for multi-unit real-time games via voting. *IEEE Transactions on Games*, 2018.

[6] Alberto Uriarte and Santiago Ontanón. Automatic learning of combat models for rts games. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.