Universiteit Leiden
The Netherlands

Opleiding Informatica

Adaptive AI in game tutorials

Tijn Huiskens

Supervisor:
Dr. M. Preuss & Dr. H.J. Hoogeboom

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

13/07/2020

Abstract

Artificial Intelligence (AI) in games can be used for a variety of things. From an opponent to play against to generating an interesting world to play in. Making an enjoyable tutorial for games can be a difficult task. This thesis is about using an existing AI in the game called "MicroRTS" and transforming that into a game tutorial.

Contents

1 Introduction

Before someone can play a game, we usually need an understanding of the game rules. When playing together with another person, he or she can explain the rules to us. When playing against an computer opponent however, the computer is not able to explain the rules. This is where tutorials come into play. Tutorials explain the rules to the player, so they get an understanding of the game mechanics.

These days games are getting more and more complex. Teaching someone to play "Space Invaders" requires less effort than teaching that same person to play "StarCraft". This is because the number of actions the player can make are far more in "StarCraft" then in "Space Invaders". Because of this, making game tutorials require more effort to make. They also become more important, as it becomes more difficult to understand the various game rules without tutorial. Methods to simplify or automate the generation of tutorials is therefore getting more and more interesting.

Various research is already done on automating tutorial generation. For less complex games like "Space Invaders", tutorials can be automatically generated using "AtDELFI". More complex games like "StarCraft" have specialized tutorials, but these require effort to make. In addition, these are for persons who already have much experience in that specific game.

The idea in this research is to use a pseudo-code to transform an existing AI into an tutorial. The tutorial will not be as advanced as specialized tutorials, but more general for people who are new to that game. The existing AI already has knowledge about the specific game and using a pseudo-code can simplify the creation of tutorials. Such a pseudo-code does not yet exist, so we have to make one and find out if it works. The main question of this paper is if this approach works: "Can an adaptive AI effectively be used as a tutorial?"

After a series of experiments, we expect to have a pseudo-code fit for a tutorial. In addition, we have a bot that has this pseudo-code implemented.

1.1 The situation

When it comes to tutorials, there is a lot of variety in both the type of tutorial and how effective they are.

There are complex games with a lack of decent tutorials. For example Europa Universalis IV with only a normal match on easy settings as a tutorial. This does not explain the game mechanics, which continue changing over the course of various updates. War Game: Red Dragon has only an in-game manual describing the game, not preparing new players at all for the complexity of the game.

On the other hand there are good tutorials for less-complex games. The latest game of Super Mario has instead of a tutorial, an AI that demonstrates how to progress when the player is getting stuck on a level. ATDeLFi went a step further by generating tutorials automatically. These generated tutorials are a manual that describe the game and add pictures or demonstrations when necessary. These demonstrations can appear when the player needs that information. However, the research on ATDeLFi concluded that it only works well for less-complex games.

For players that are already experienced in a game, there are tutorials to make them even better. An example is the game "Age of Empires II", for which the gaming community made adaptive AI to teach and challenge more experienced players. Another example is the game Starcraft, where there are tutorials for specific build orders depending on what strategy the player want to use.

When it comes to AI, the "General Video Game Artificial Intelligence", short GVG-AI, is an organisation that stimulates game-AI research. Including AI that can learn to play games on its own. More detailed information on the state of AI is described in "gameaibook" [YT18].

1.2 Thesis overview

Section 1 is an introduction into tutorials and the current state of them;
Section 2 is a list of definitions used in this paper;
Section 3 discusses work that is relevant or related to this research;
Section 4 describes the general approach of this research;
Section 5 describes the experiments done, iterating until the results are enough for a conclusion;
Section 6 uses the information from previous sections to concludes this research.

2 Definitions

Below Definitions according to the dictionary [LLC95]. Except for MicroRTS, this is from their website [Ont13b]. Only the definitions used in this paper are written below.

- Tutorial
  In this context for Computers.
  A. programmed instruction provided to a user at a computer terminal, often concerning the use of a particular software package and built into that package.
  B. a manual explaining how to use a particular software package or computer system.

- Real-Time Strategy game (RTS games)
  Real-Time, of or relating to applications in which the computer must respond as rapidly as required by the user or necessitated by the process being controlled.
  Strategy, Also strategics. The science or art of combining and employing the means of war in planning and directing large military movements and operations.

- MicroRTS
  MicroRTS is a small implementation of an RTS game, designed to perform AI research. The advantage of using microRTS with respect to using a full-fledged game like Wargus or StarCraft (using BWAPI) is that microRTS is much simpler, and can be used to quickly test theoretical ideas, before moving on to full-fledged RTS games.

- Artificial Intelligence (AI)
  The capacity of a computer to perform operations analogous to learning and decision making in humans, as by an expert system, a program for CAD or CAM, or a program for the perception and recognition of shapes in computer vision systems.

- Bot
  A device or piece of software that can execute commands, reply to messages, or perform routine tasks, as online searches, either automatically or with minimal human intervention (often used in combination).

- Field of Vision (FoV)
  The entire view encompassed by the eye when it is trained in any particular direction.
  Note: in this papers context it is the entire view encompassed by the computer screen.

- Modding
  The act of rewriting programming code in a video game in order to change the appearance or performance of the software.

- Spawn point
  Of or relating to the spawning of a character or item in a video game
  (of a character or item in a video game) to originate at a fixed point in an existing game environment.

3 Related Work

Section 1 describes the situation around game tutorials and why we want a new approach. This chapter describes the relevant projects that are already out there.

3.1 Tutorial generation

To simplify the generation of tutorials, a team of scientist have made in 2018 "a fully automatic method for generating video game tutorials" [GKB+18]. Their approach is tested using the General Video Game Artificial Intelligence (GVG-AI) framework. Their system "generates a combination of textual instructions and visual videos". Part of their conclusion is that a "tutorial generator needs to have a higher level of understanding than just simple mechanics, when it comes to more complex games".

Our research will focus on existing bots as a basis for the tutorials. These already have game specific knowledge and are available before the game is released. Using this method, we aim to simplify tutorial generation for complex games.

3.2 Demonstration-based tutorials

Learning how to play a game by playing against a bot is not a new concept. AlphaGo Zero [Dee20] is a great examples of learning by playing matches instead of following a tutorial. As stated on their website: "While AlphaGo learnt the game by playing thousands of matches with amateur and professional players, AlphaGo Zero learnt by playing against itself, starting from completely random play".

AlphaGo Zero became the best in the board-game Go only by playing matches. It did not even start with a good opponent, but merely self-play against itself using random moves. Our research uses bots to let a player immediately play a "real" match instead of following a tutorial. By letting the player win and demonstrating most of the game mechanics, we aim for a better first experience with new games.

3.3 Adaptive tutorials

On July 2nd 2015, a fan of "Age of Empires II" (AoEII) under the name of "Resonance22" [res13] made an adaptive AI in AoEII [res15]. This bot is quite popular in the game, receiving the 5/5 star rating on steam with almost one thousand people rating it. It shows that in complex RTS games adaptive AI can be fun to play against. The author has made this bot adaptive, to challenge advanced players and improve their game-play. When trying this bot as a beginning player, this bot was too challenging. The author requested in his code not to edit it. However, his bot is inspiring to make a more simple variant for tutorials to beginning players.

It requires more than simply an adaptive bot to be suitable for a tutorial. What we learn from this adaptive AI is that it should not be aiming to win, unlike standard AI opponents. Also it should not advance too far ahead of the player, giving players the time they need to learn and respond. The ResonanceBot is using a rule-based system. This way the bot only performs certain actions if their criteria are met. This makes it adjust to the speed and strategies of the player. We will also use a rule-based system for our tutorial.

3.4 AI-tournaments

Research on bots is easily accessible these days. There are multiple tournaments across various game genres were bots compete against each other. A good source to start is the IEEE-CoG [iee20].

Among the listed tournaments on their website are:

- "Fighting Game AI Competition", for AI in brawler games;
- "General Video Game AI: Single-Player Learning Competition", for adaptive AI;
- "StarCraft AI Competition", for AI in the complex RTS-game "StarCraft";
- "uRTS", for AI in the minimalistic RTS-game "MicroRTS".

3.5 AI-research

MicroRTS [Ont13a] is a game focused on AI research. All main elements of an RTS game are present. The game can be modified to suit different needs. From adjusting the units to the speed at which it is played. In a typical RTS game, all units can be build from buildings, and all buildings can be build by units. This game has as its default values 2 buildings and 4 units. A base can build workers. Workers can build bases and barracks. Barracks can build light, heavy and ranged combat units.

Units or buildings can also have multiple functions. In this game workers both also harvest resources and behave like combat units. Despite being minimalistic, this game is already complex, representing RTS-games.

Both the game and the AI in this game is written in Java. Most AI are classes that extend existing classes. In the case of this research, our AI extends the "AbstractionLayerAI" class. When making AI for tournaments, they can extend the slightly different "AIWithComputationBudget" class. Using this approach, we mostly have to change the function "getAction(int player, GameState gs)" (Appendix A) and calls from this function.

4 General Approach

In section 1 we have seen the situation on tutorials. In this research we focus on simplifying tutorial generation for complex games. We aim to do this by finding an algorithm suitable to turn existing bots into tutorials. We implement a set of general rules into an existing bot. Then we check if the altered bot can serve as a tutorial.

Before we can do this, we have to decide on the following things: A game, a bot and an algorithm or pseudo-code. When we have a game, a bot and a simple algorithm or pseudo-code, we can start experimenting. We implement our algorithm and describe the challenges we encounter. Then we run a few pre-defined checks to see if it has any change of becoming a tutorial. If it passes the checks, we have players play against it and see their reaction to it. If the tutorial does not pass the checks, we take another look at the algorithm to find a solution. This makes the general approach of this research is as follows:

1. To set-up, decide what game, bot and pseudo-code to start with.
2. Then correctly implement the pseudo-code in the bot.
3. Perform basic checks on how this changed bot performs.
4. In case it does not perform promising, adjust the pseudo-code based on the bot behaviour and repeat last two steps.
5. Otherwise, check how this tutorial performs by letting both experienced and inexperienced players use it.

4.1 Set-up

4.1.1 Game

There are many games in which bots can be made. In addition to the various game types or genres, there are many different games within those genres. Choosing just any game from the store is not ideal for this research. Commercial games can have restrictions to editing existing code. That is why we are looking at open-source games. If the game is too large it can take too much time to implement, reducing the time we can spent on researching the effects. Since our aim is to research bot behaviour, we can use games dedicated to this type of research. For the open-source games there are various games and platforms dedicated to AI and research.

To get an overview of various researches in AI, we check the website of "IEEE-CoG" as mentioned in section 3.4 for current competitions. Various AI-competitions are described here including tips and tricks on how to get started. We are preferably looking for a genre with complex games were our tutorial can have most added value. On the other side, we would like a small game were we can spend more time on our research.

First we checked a fighting game competition platform. The "OpenICE" [lRU20] platform used for AI game competitions and AI research. This platform is originally made for Linux, and later supported under windows by using specific interfaces or libraries. Our research is done under Windows, and the set-up for the OpenICE platform on Windows was too time-consuming.

Then we looked as an alternative to MicroRTS, see figure 1. Similar to the OpenICE platform, this one is made and used for both AI competitions and AI research. A restriction of this platform is that it supports only the MicroRTS game. On the plus side, this is a standalone application that runs under both Windows and Linux. Since the game MicroRTS is small, complex and the set-up is simple, this is the platform of our choosing for this research.

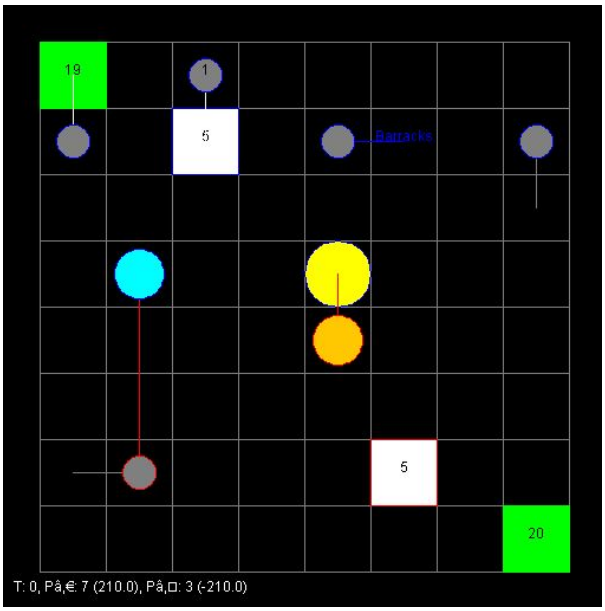

Figure 1: Demonstration of various units and actions in MicroRTS

```
159         // WORKER:
160         UnitType worker = new UnitType();
161         worker.name = "Worker";
162         worker.cost = 1;
163         worker.hp = 1;
164         switch(version) {
165             case VERSION_ORIGINAL:
166             case VERSION_ORIGINAL_FINETUNED:
167                 worker.minDamage = worker.maxDamage = 1;
168                 break;
169             case VERSION_NON_DETERMINISTIC:
170                 worker.minDamage = 0;
171                 worker.maxDamage = 2;
172                 break;
173         }
174         worker.attackRange = 1;
175         worker.produceTime = 50;
176         worker.moveTime = 10;
177         worker.attackTime = 5;
178         worker.harvestTime = 20;
179         worker.returnTime = 10;
180         worker.isResource = false;
181         worker.isStockpile = false;
182         worker.canHarvest = true;
183         worker.canMove = true;
184         worker.canAttack = true;
185         worker.sightRadius = 3;
186         addUnitType(worker);
```

Figure 2: Properties of "Worker"-units in MicroRTS

In MicroRTS, all units, structures and other in-game objects are changeable as needed for research or balancing. There is a Java file containing all units, as seen in figure 2. New units can be imported or directly added in this file. Both the player and the bots will have access to the units with their corresponding actions in the game. For this research the default units and settings are sufficient. However, this game can be scaled up in complexity by adding more units, larger maps or maybe even introducing field of vision. All units/structures/objects in this game have a value for sightRadius, hinting at the support for Field of Vision (FoV). This concept is common in RTS games and can

be interesting for follow up research. In this research however, FoV is not part of the tutorial and the game complexity is kept as low as possible to keep focus on the research.

4.1.2 Bot

Now that we have a game, we need a bot to apply our rule-based system to. Within MicroRTS, there is a wide range of selection of bots to choose from. We will take a look at some of them and see what is good enough for our research.

The first bot we see is one called "PassiveAI". As the name suggest, this bot does nothing at all. When our tutorial bot plays against the passive AI, we can monitor the behaviour of tutorial bot in case the opponent does nothing. Besides that, using this bot as starting point for our algorithm would be like writing the entire bot ourselves. We look further to the other possibilities.

Among the selection, there are multiple "defense" bots and "rush" bots. Typical for a defence bot is that the strategy is based on defending itself instead of attacking the player. The implementation of defence bots in this specific game is as follows: It builds an army and places it nearby its own base. This army is on stand-by until the player approaches the AI. Then the full army is used to defend against hostile forces and retaliate.

Typical for a "rush" strategy is to attack as quickly as possible, preventing opponents to build and develop their own army. If the opponent does not spend resources early on defences, the rush strategy will quickly take out the opponent. The difference between the the various rush bots are the type of units they use. "WorkerRush" makes only worker units, "LightRush" makes only Light units and so on for all unit types. The same applies for the variants of the defence bots.

Both rush and defence bots are decent for our research. They have all functions implemented and applied, from resource gathering to combat. The rush strategy is the opposite of what we want to accomplish. Instead of attacking as soon as possible, we want to wait until the player is ready for it. We pick a variant for the "rush" bot, called "RangedRush". This bot rushes to build ranged units and attacks with those as soon as possible. With such a difference between our starting point and goal, we can see how well our method works.

4.1.3 Pseudo-code / algorithm

By using an AI opponent to teach the game, we use teaching by demonstration. The idea is that humans learn by watching and copying behaviour of others, in this case the behaviour of our tutorial. If the AI can then play the ideal game and the player would learn from that, we can use any tournament-winning AI to teach.

However, the tutorial is supposed to be a stepping stone into the game. Inexperienced players can be discouraged to continue playing when facing an opponent that is too strong. The tutorial-AI in this research will respond at the pacing of the player. In addition, it is not attempting to win the game. The goal of the AI is demonstrating the various game concepts at a pacing the player can follow. By letting the player win in a challenging way, we prevent discouraging players from playing this game.

To do this we we are not making a "perfect" bot. Instead it follows a few rules to make it a nice teacher. When playing against ResonanceBot in Age of Empires II we realized that tutorials should adept their pacing to that of the player and they should not be capable of defeating the player. If we can make a pseudo-code around these basic rules we can see how well it performs as a tutorial.

4.2 Implementation

As the opponent of the player, our tutorial can either show moves with the aim of the player copying its behaviour, or confront the player with a situation where it can only escape by using a specific move. If these options are not sufficient or possible then there are alternatives like on-screen messages of what the player should be doing or highlighting the objects that are waiting for interaction.

Of all these teaching methods, we are only implementing to show moves with the aim of the player copying its behaviour. We want to find out whether this kind of tutorial is generally suitable for games. If we implement multiple teaching methods, we may be limiting our research to games that only support all of those methods. In our case the game MicroRTS does not provide communication between players and bots out of the box. However, assumptions are to be made with these kind of restrictions.

The assumption is made that the interface is either self-explanatory or intuitive enough for players to understand. If this is not the case, the tutorial can be extended by previously discussed methods. This is not part of our research.

What remains is the algorithm that we implement. The bots are written in Java, so we are also implementing our algorithm in Java. The exact architecture is based on the implementation of the existing bot we are altering. Details, challenges and solutions are written down during the experiments when we encounter them.

4.3 Basic check

If we were to try out every implementation we make on actual human players, we need a large group of players to involve in our experiments. Any inexperienced player would not be able to apply again as inexperienced. To reduce the amount of involved people, we apply a series of checks on every implementation we make. These checks are based on some basic tutorial rules specifically aimed at our tutorial. If all of these rules apply to our tutorial, we say it is promising enough to try out against players.

When playing tutorial or against adaptive AI in other games, we have already established some rules for our tutorial. One of those rules is to adept the tutorial pacing to that of the player. To see if our tutorial bot adepts correctly to the pacing of the player, we can put our implementation up against the "PassiveAI". This opponent is not doing anything, so our tutorial should not use any unit at all. Another check for this rule is to build a unit and see if the tutorial starts using their similar units then.

Another rule is to let the tutorial always lose. This is technically not possible to build checks for. What we can do is check if it does not win, so we alter the rule to make it not win instead. To check this we can reuse the previous situation were our tutorial was put up against the "PassiveAI".

If the tutorial did not win during that scenario, we can also verify if it does not win when the player actually has followed all of the demonstrated behaviour. To do this we have to play against our tutorial and copy all of its behaviour up to the point where we have multiple combat units. If it does not attempt to win at this point either, we can safely assume that the tutorial does not attempt to win whenever a player is playing against it.

The previous rules are easily followed by any bot that does not do anything at all. However, we would like a tutorial that teaches how the game works, including combat. Since teaching combat can conflict with the previous check, we make a separate rule to teach combat.

4.4 Adjustments

If the previously described checks fail, it means that the algorithm is not suited as a tutorial. We can then take a look at why the check(s) failed and alter the algorithm accordingly. When changing the algorithm we keep in mind that the pseudo-code is meant for various games. This means the algorithm should remain as simple and independent as possible. If it takes much effort to make a change, it is unlikely suitable for more complex games.

After changing the algorithm, we have to reapply it on our bot. This is again described and then we have to apply the checks again. Beforehand we do not know how many iterations we need and if we will ever find a suitable tutorial. What we do know is that every iteration is getting closer to the answer we seek and more data to work with.

4.5 Player check

When our tutorial did pass our checks, we can see how it performs as a tutorial. First we let some experienced players try it out. They can give useful feedback on the pacing, completeness and difficulty. Then we can try it out on inexperienced players.

We ask the person to play against the "RandomAI", our modified "RangedAI" and the existing "LightRush". The "RandomAI" does constantly a random move with all of its units and structures. The reason we ask players to play against this is to see if our tutorial does better than just randomly assigning tasks to units and structures all the time. We can also use this moment to give a basic description of the interface, since this is not part of our tutorial.

Then we ask the person to play against our modified "RangedAI". During this session we ask the players why they do the actions they take. Afterwards we ask if they enjoyed it and if they learned how the game works from this play-through.

Then we let the person play against an existing "Rush" AI to see if that could do as well. We let the players play against the "LightRush" and ask them afterwards how they enjoyed that and if they could learn the same from that bot as they learned from the modded variant. We also ask if they learned anything from either the "RandomAI" or the "LightRush" that they did not learn from our tutorial.

Once all of this is done by all players we have enough information to see if this type of tutorial-bot has potential in games. This way we can find out if that is an effective way to replace or supplement (non-)existing tutorials in games. As a result we end up with some rules to help transform certain existing AI into tutorials.

5 Experiments

In this section we implement an pseudo-code into a bot and see how it performs as a tutorial. The idea behind it is to find out if a pseudo code translated into existing bots can make decent tutorials.

Tutorials in RTS games can teach building, using and the strengths/weaknesses of units and buildings. Strategies are usually not part of a tutorial. More advanced guides exist for that. If the controls are self-explanatory, this would make a full tutorial.

In real-time games, the state of the game field is constantly changing. An AI can constantly monitor this game-state. For example, at a given time it can check each object in the game and extract information from it. While doing this, the real-time goes on. However, computers can do this faster than that players can make actions. Because of this, it looks like the AI continuously knows the game-state. We use this and apply a rule-based system, to respond to player actions. By counting the amount of units the player has, we can make if-statements to decide when to teach something.

Let us start with the situation where the player has 0 of a certain unit. Then it is useless to teach how to use these units. The player is unable to do that from that game-state to begin with. On the other hand, if the player has made multiple of those units, it is useless to teach how to build them. The player has likely done that already in order to get multiple of those units. A solution to this is to base it on the amount the player has of these units or buildings.

The first attempt is described by the following pseudo code:

Algorithm 1: Tutorial-bot initial

```
Initialize x as amount of actions to teach for each individual unit/structure;
Initialize pCount as the amount of player units and buildings, sorted by type;
forall unit/building type do
    if pCount < x then
        Demonstrate building this unit/building type;
    else if pCount = x then
        Demonstrate using this unit/building type;
    else if pCount > x then
        Demonstrate countering this unit/building type;
end
```

The value of x is a variable based on the number of actions that we want to explicitly teach for that specific unit type. This includes actions like combat, harvesting and building. It excludes actions like movement, as this is demonstrated implicitly as part of other actions. The idea behind this is that we need at most 1 unit for every action we want to teach. It is possible that a single unit can teach multiple actions. This value can be tweaked for each unit or building when trying out the tutorial.

The value of *pCount* is unique for each type of unit and building. During the **forall** *unit/building type* it loops through each unit and building type, using the unique *pCount* for that specific unit or building.

5.1 Implementation algorithm 1

What action to take is defined in the function "getAction" (Appendix A). This function already sorts all units and buildings by type, calling different functions based on their type. In other words, the **forall** loop in our pseudo code is already implemented in the existing AI. This is no coincidence, as AI usually checks the current game-state. That is the only way the AI can make decisions for the next move. That it is already sorted by type is no coincidence either. The pseudo code does not state how it is sorted. In this case the AI sorts on *bases*, *barracks*, *melee units* and *workers*. It could have been a different sorting. The flexibility of our pseudo code takes this into account.

Since we are re-using an existing AI instead of writing a new one, it is good to take a look at the existing definitions. For example, workers are defined as units that can harvest. In this specific case there is only 1 type of unit capable of doing this. If there are units added that can harvest, we can take another look at this call. Another aspect is that *melee units* is marked as one unit type, even though there are 3 units in this category. Other AI can have different definitions, changing the implementation of this pseudo code while keeping the result similar. Given our situation, most changes take place within "baseBehavior"(Appendix B), "barracksBehavior"(Appendix B), "meleeUnitBehavior"(Appendix C) and "workersBehavior"(Appendix C).

baseBehavior
The base-behaviour is responsible for building units. This means it can demonstrate building units, so we should implement the "**if** $pCount < x$" rule. We also want to demonstrate how to use this building, however this behaviour is superseded by the earlier build demonstration. Because of this, we do not have to implement this statement as it would simply build units again. Countering this building is also not needed, as this rule only applies for combat. This building has no combat behaviour. In different games buildings can have different actions or combat capabilities. It seems that the pseudo-code takes this into account. For this specific game of MicroRTS we only have to implement 1 rule in the function "baseBehavior".

In order to count all of the player units, we have copied and altered the for-loop from function *unitBehaviour*. Within function *baseBehavior*, there already is an if-statement to check if the building should build units. The if-statement is extended to contain our "$pCount < x$" as well. For x we have chosen the value of 3. This is based on the worker-actions we do like to teach: Harvesting, Building and Combat.

barracksBehavior
Just like bases, the barracks-behaviour is also responsible for building units. This means we only have to implement the "**if** $pCount < x$" rule. Differences are the value of x, which we set to 1 for combat units. These units cannot build or harvest, only engage in combat. Also there are 3 types of units that the barracks can build, so we have to count all 3 unit types beforehand. Overall, a similar implementation applies to the barracks as for the base.

meleeUnitBehavior
When taking a look at this function, it is responsible for all non-harvesting combat units. In this specific setting, all units within this function are only capable of combat. Not only melee combat as suggested by the function name, but the implementation is the same for ranged and melee units.

Units that can harvest and fight are taken care of in another function. When a unit is added that can build and fight, but not harvest, a new function should be made for that. It is no coincidence that such units do not exist in this game. If it were to exist, the AI already had a separated function to handle those units.

Since all units within this function cannot build other units or buildings, we do not have to implement the "**if** $pCount < x$" rule. However, since the units can still be used, the "**else if** $pCount = x$" will be implemented. This rule is implemented by letting a unit attack the same unit as the player has. The player always has this unit, because that is the condition for this statement to activate.

In addition to using this unit, a player should get a basic understanding of the strengths and weaknesses of these units. We will implement the "**else if** $pCount > x$" rule as well to teach this. The idea behind this is that if a player has "too many" of a unit type, the AI will demonstrate that units weakness. This is implemented by hard-coding a counter-unit for each combat unit the player has.

Since this is a tutorial for beginners, only basic understanding of strengths and weaknesses have to be taught. For example, if the player has $> x$ light units, the AI will attack the nearest light unit with a heavy unit. The AI always has at least one heavy unit, because it will demonstrate building 1 even if the player has 0. Exceptions are when the counter unit is "locked" by other means. For example, the counter unit is made by other buildings that are not available. This should be taken into consideration when and if assigning counter-units.

workersBehavior
We have seen the implementation of "**if** $pCount < x$" in the base and barracks behaviour, both with different values of x. After that we have implemented the "**elseif** $pCount = x$" and "**elseif** $pCount > x$" in the combat units behaviour. Now we will see all these functions come together in the workers-behaviour function. Workers can build buildings, harvest resources and engage in combat.

The implementation for building behaviour is the same for units and buildings. First we count the number of relevant objects the player has. In this case the number of barracks and bases. Then we implement the if-statement where workers build barracks or bases if the player has $< x$. The value of x is 1 here. This is because these specific buildings can only build.

Next the implementation for using units. We have already implemented unit building behaviour. What remains is the harvesting and combat behaviour. In this case we can make the value for x either 2 or 3. For only teaching harvesting and combat behaviour 2 may seem sufficient. However, 1 unit is already occupied in building. So we set this to 3, since we want to teach both harvesting and combat behaviour in addition to building.

With the implementation for using units behind us, we can focus on countering these units. Workers are the weakest units in the game. Any unit besides the worker itself can serve as a counter. However, all other units are "locked" behind other requirements. In addition, the existing implementation separates workers behaviour from combat units behaviour. Because of this, we will simply counter the redundant amount of player workers with our own. Choosing the value of 3 for x again, we implement the last if-statement.

5.2 Observation algorithm 1

To observe the pseudo-code, we will eventually let players try it out and get their feedback. Before doing that, we have a few checks too see if the tutorials are promising. We apply a set of 4 scenarios to verify that the pseudo-code behaves as expected.

The first check is to verify the tutorials pacing. According to the pseudo-code, it should not use units until the player has those specific units. This can be verified by respectively not having any unit at all and building units that the AI has and can use. In other words, make no units at all and make a basic unit.

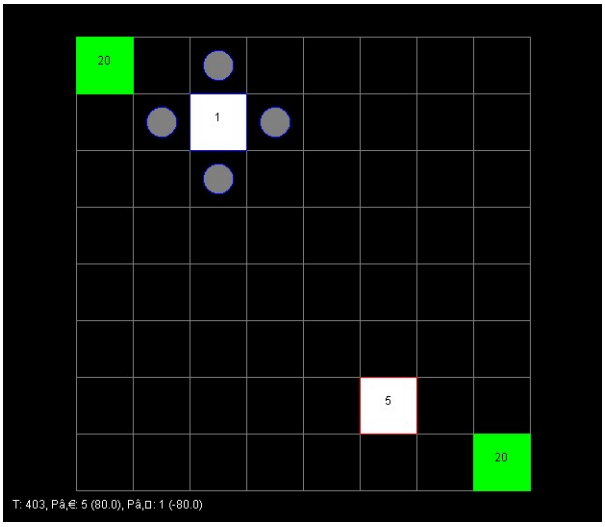

Figure 3: Algorithm 1 (blue) is not using units until the player (red) has similar units

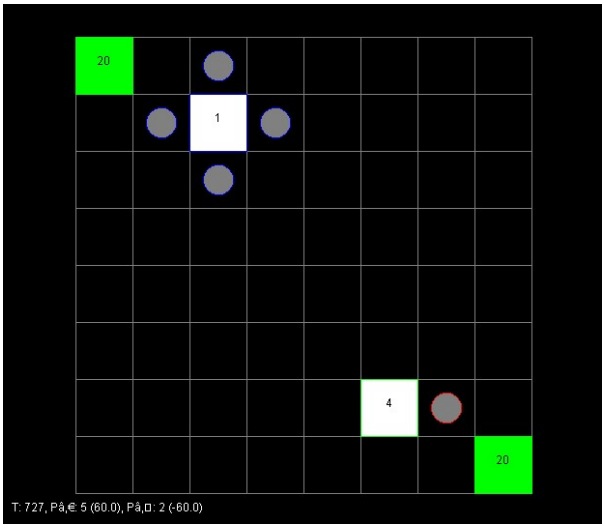

Figure 4: Algorithm 1 (blue) is supposed to use units now that the player (red) has similar units, but this does not happen

Another check is to see if the tutorial will always lose. This is technically not possible to check. However, we have already verified it will not win if the player does nothing in the previous scenario. We add the situation where the player becomes a treat by building combat units. If the tutorial still makes no attempt at winning the game we say it passed this check.

Last check is when the player makes multiple of the same unit and see if the tutorial can counter these. Since worker units have a special place among counters, we use an actual combat unit for this. It does not really matter which of the 3 combat units, for they all have a hard-coded counter

unit. In the screen-shot we can see that the player made 3 "light" units, in addition to 1 heavy and 1 ranged unit. We except to see the tutorial attacks the light units, however this is not happening. The tutorial does not even build combat units.

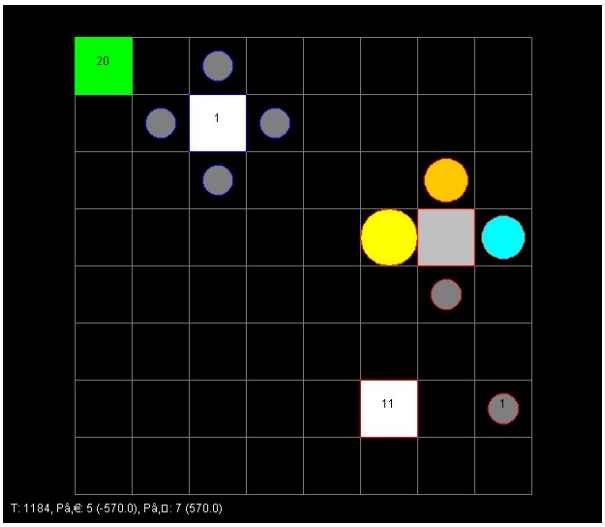

Figure 5: Algorithm 1 (blue) does not attempt to win

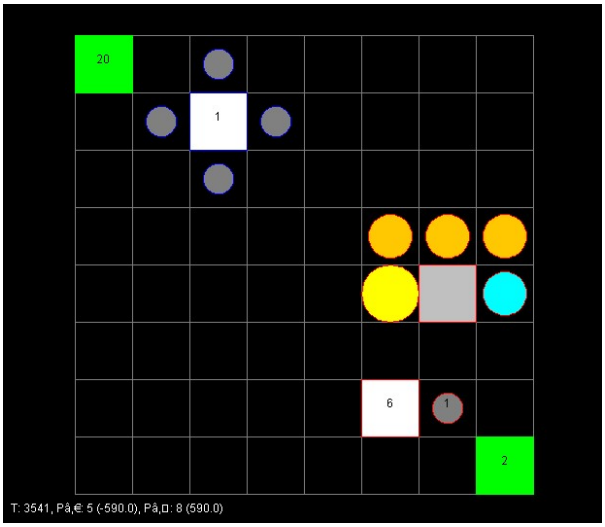

Figure 6: Algorithm 1 (blue) does not even make a single combat unit in this situation, despite the player (red) having multiple

5.3 Alteration algorithm 1

When looking at the pacing, figure 3, function base-behaviour was making 4 worker units. It did not wait for the player to build something and did not limit to x. We want to slightly change the pseudo-code to have max 1 unit more than the player, of each unit type. This makes the base-behaviour adept to the pacing of the player. This also prevents the Tutorial from making more than x units of a type. The new if-statement is: **if** $pCount < x$ **and** $pCount >= aiCount$

When we look at figure 4, the tutorial did not demonstrate the use of units. The tutorial did not use units, because the player never made x amount of workers. We can alter the statement to check if the player simply has 1 unit. This prevents the tutorial from waiting until the player has x units. Whenever the player has a unit, the tutorial will demonstrate using that unit. The new if-statement is: **if** $pCount = 1$. The idea behind this, is that we can teach how to use any unit as soon as the player has that unit.

When looking at figure 5, we see that it did not attempt to win at all. This part of the behaviour is what we expected to see, so no changes are required on this part.

The last check, seen at figure 6, implies that the tutorial does not counter. However, the tutorial never reached a situation in where it can counter, since it lacked the necessary units and buildings. For now, we will keep the counter function as it is.

Algorithm 2: Tutorial-bot building-behaviour update

```
Initialize x as amount of actions to teach for each individual unit/structure;
Initialize pCount as amount of player units and buildings, sorted by type;
Initialize aiCount as amount of AI units and buildings, sorted by type;
forall unit/building type do
    if pCount < x and pCount >= aiCount then
        Demonstrate building this unit/building type;
    else if pCount = 1 then
        Demonstrate using this unit/building type;
    else if pCount > x then
        Demonstrate countering this unit/building type;
end
```

5.4 Implementation algorithm 2

The changes of the previous experiment are in the "build" statement and the "use" statement. The functions to alter for the "build" statement are baseBehavior, barracksBehavior and workersBehavior. The functions to alter for the "use" statement are meleeUnitBehavior and workersBehavior.

baseBehavior
The if-statement was extended in experiment 1 to contain our "$pCount < x$". In this 2nd run, we add the before mentioned **and** $pCount >= aiCount$. This does mean counting another variable. The implementation of this count is similar to the $pCount$ variable. By copying and editing that part of the code, this is little effort to implement. The expected outcome of this change is that it will build only 1 unit (of each type) ahead of the player and not make more than x units of each type.

barracksBehavior
For the function of barracksBehaviour, the changes are more of the same. Just like baseBehavior, counting the number of AI units and then adding the $pCount >= aiCount$ to the if-statements. Having 3 different unit types, we have to implement this 3 times within this function. In our code, we named the variables and implemented the statements respectively $pLight >= aiLight$ $pHeavy >= aiHeavy$ and $pRanged >= aiRanged$.

meleeUnitBehavior
This function has sorted units that can only use combat. This means, the moment a player has made these units, we can immediately demonstrate the combat behaviour. In the first experiment we have not seen the effect of the $pCount = x$ with $x = 1$ rule on combat units, because a barracks was never build. We do like to see the behaviour of this rule before changing it. Coincidently, the new rule $pCount = 1$ has the same implementation. We can see the effect of the previous rule as well as the new rule with this function. We will not change the $pCount = 1$ for now, since we already had $x = 1$.

workersBehavior
Unlike the combat units, we have already seen that the multi-functional worker units do not teach

what they should. We apply the changed algorithm here to see if it is an improvement. First we apply the altered rule for building, then the rule for using it.

For the building rule, we have already implemented the altered rule in the baseBehavior and barracksBehavior functions. We apply the same changes in this workersBehavior function. Adding the $pCount >= aiCount$ to the existing if-statement for building behaviour.

The changed rule for using units is only a change in the value of x. Using this unit for building is still not under this rule, for that is covered in the first rule. So this will show how to fight and how to harvest. The rule for countering is left unchanged. The observation will show how this works out.

5.5 Observation algorithm 2

Using the same set of 4 scenarios, we take a look at the behaviour of algorithm 2.

The first check is still to verify the tutorials pacing. We expect to build only if the player has an equal amount of that unit. So it can build 1 unit when both player and AI have none. Then it will wait building until the player has 1 unit. At this time it can build another one, if the value of x allows for it.

What we see in the first scenario, see figure 7, is that it will not build more than 1 unit as long as the player does not have one. It also will not use the unit until the player has one too. This is what we expected, so this part of the algorithm seems correct so far.

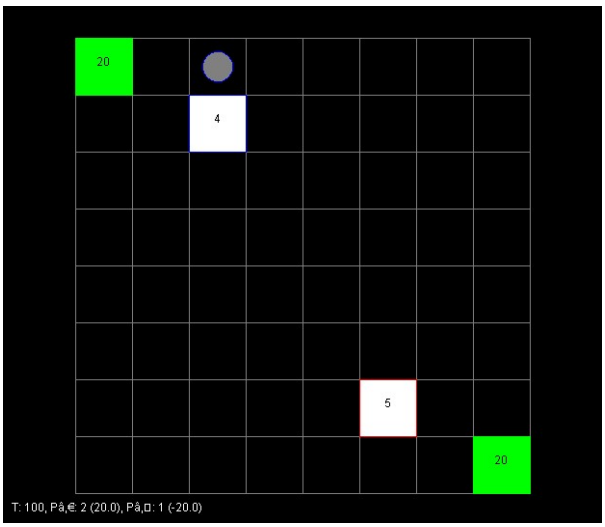

Figure 7: Algorithm 2 (blue) is not using units until the player (red) has similar units

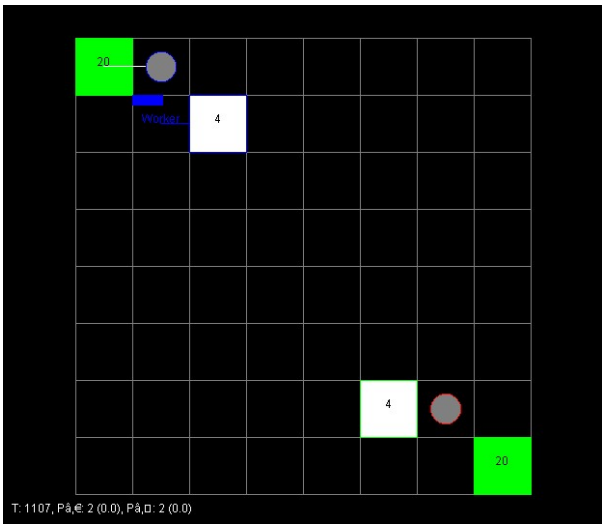

Figure 8: Algorithm 2 (blue) is using units now that the player (red) has similar units

When the player does make a unit, see figures 8, 9 and 10, the tutorial starts building another one and demonstrates the various uses. Harvesting, building and combat behaviour are all demonstrated. When both the player and the tutorial loses a worker, the tutorial does not make another one until the player has made one. Also, it does not fight again until the player has at least one unit.

Noticeably, the tutorial does continue harvesting. This is likely because of the game mechanics, where units keep harvesting until set to another task. Having 2 units, these demonstrations can feel a bit fast-paced. It is also not very nice to start attacking the moment the player has a single unit. This is something to find out by having humans play against it.

Example of possible changes are excluding combat from the use-rule as well. Just like the build action is not part of the use-rule. We can then add the combat behaviour to the counter-rule. For now these ideas are not implemented until we get feedback from human play-testers, besides ourselves.

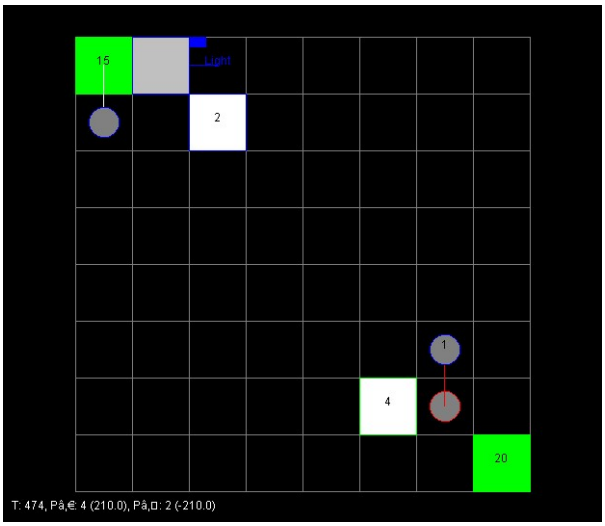

Figure 9: Algorithm 2 (blue) has demonstrated building a barracks and currently demonstrates combat

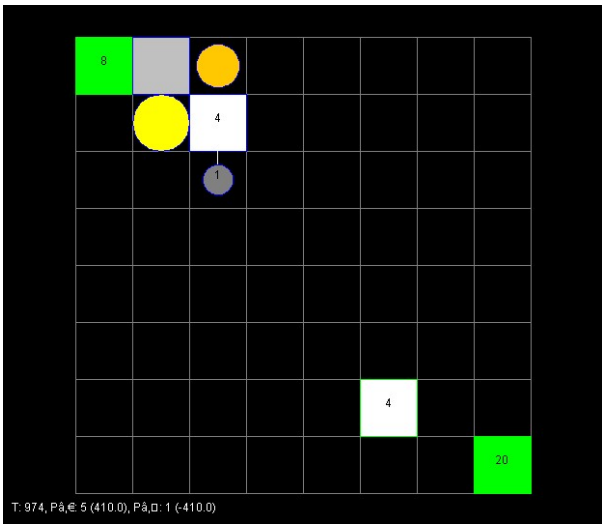

Figure 10: Algorithm 2 (blue) has destroyed worker units of the player (red), but does not finish the game

In an attempt to reach the next scenario, we find out in how difficult it is to reach with the tutorial continually attacking our workers. As seen in figure 11, it took all of our resources (the green square with a number on it) to reach the next scenario. The tutorial did not make an attempt to win. However, just like the first experiment the tutorial did not have any combat units to fight with. When having more than x worker units, the tutorial did "counter" by attacking the surplus workers with its own. As can be seen in figure 12, this stopped when the player had x amount of units. In that situation, it stopped using and building units. In other words, the tutorial stopped when the player had made at least four worker units and at least one of each combat unit. Another thing we noticed is that at one point the tutorial could not build combat units, because the barracks was "locked-in" by its own worker units. This can be seen in figure 10.

5.6 Alteration algorithm 2

The algorithm has three rules: for building, for using units excluding building but including combat, and for countering. We take a look at these rules in order.

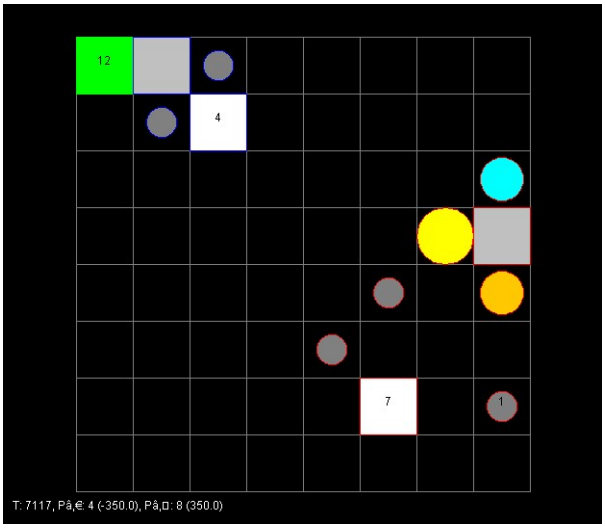

Figure 11: It cost most our resources and much effort to build a barracks and combat units

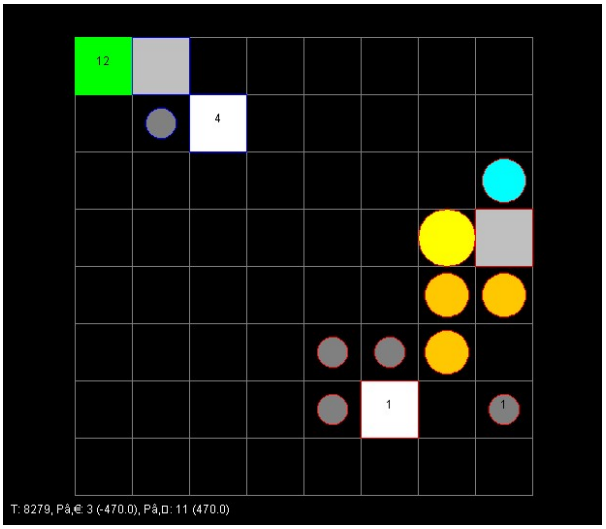

Figure 12: Algorithm 2 (blue) is no longer building units when the player has made x amount of those unit

When taking a look at the building behaviour, it stopped when the player reached x units. This means that the building rule prevents the tutorial from functioning once the player has reached certain criteria. We can change the $pCount < x$ into $aiCount < x$. The second part of the building-rule remains unchanged, for this makes it adapt the pacing to that of the player. The first part still prevents the tutorial from making too many units, but no longer stops producing at all when the player has reached that amount. The new if-statement is: **if** $aiCount < x$ **and** $pCount >= aiCount$.

The combat rule is the same as using a units, while they should be used in different situations. By separating this, the rules can be tailored better to the specific game or situation. What we do, is making the rules for a single action only. Changing the countering rule into the combat rule, can let us tailor both the use-rule and combat-rule to their respective situation.

Countering will be replaced by combat, since the combat rule is bringing a lot of difficulties. After we find a suitable solution for combat, we can always decide to add another rule specifically for more advanced concepts like countering. With these changes we have a new algorithm 3.

5.7 Implementation algorithm 3

There is a slight change in the building behaviour and a more major change in the use and combat behaviour.

baseBehavior
The baseBehavior function only has building behaviour, so a simple change of variable pCount into aiCount is all that is necessary.

Algorithm 3: Tutorial-bot rule-separation update

```
Initialize x as amount of actions to teach for each individual unit/structure;
Initialize pCount as amount of player units and buildings, sorted by type;
Initialize aiCount as amount of AI units and buildings, sorted by type;
forall unit/building type do
    if aiCount < x and pCount >= aiCount then
        Demonstrate building with this unit/building type;
    else if pCount = 1 then
        Demonstrate using this unit/building type, excluding building and combat;
    else if pCount > x then
        Demonstrate combat with this unit/building type;
end
```

barracksBehavior
Similiar to baseBehavior, the only change here is a single variable in the if-statement.

meleeUnitBehavior
Interestingly, the meleeUnitBehavior function becomes more like the original function. Without countering rules we can remove half of the changes we made earlier. The only thing remaining is the combat part, with the rule changed to that of the previously countering behaviour.

workersBehavior
In the workersBehavior function, we had combat behaviour that activated by the same rules as using these units to harvest. The if-statement is now changed to that of the new combat behaviour.

5.8 Observation algorithm 3

This is the last time we use the four checks, before revealing our tutorial bot to players.

During the first check, our bot shows no changed behaviour compared to the previous algorithm. It builds a worker and waits for the player before using it.

When the player does make a worker, the bot starts building another one. Using both workers the tutorial demonstrates building a barracks and harvesting resources. It does attempt to take away the player resources in case the player does nothing at all. Both described behaviours can be seen in Figures 13 and 14.

The continues attempt to harvest stops the moment a second worker is build by the player. This does not stop previous harvesting activities. Behaviour like this can be dependent on the used bots and the exact implementation. The hard value of $x = 1$ in the use function does prevent teaching usage when the player quickly builds multiple units. This is shown in Figure 16, where a barracks is never build.

Since there is no more countering in the rules, the last check is about attacking instead. According to these rules, the tutorial may only attack if there are multiple units of a type. This behaviour can be shown in Figures 17 and 18.

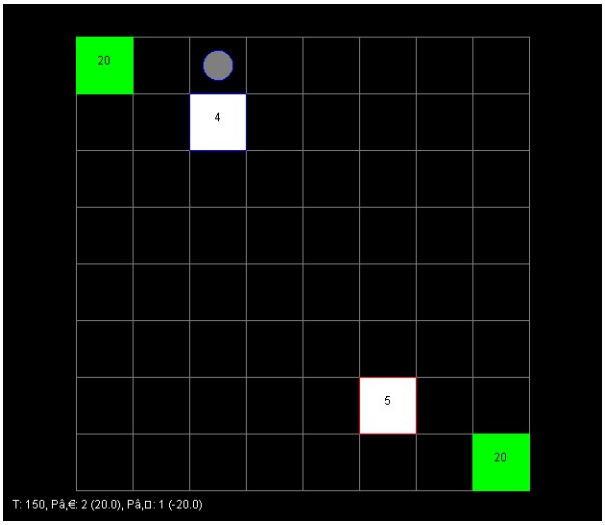

Figure 13: Algorithm 3 (blue) waiting with unit actions until the player (red) has made that unit.

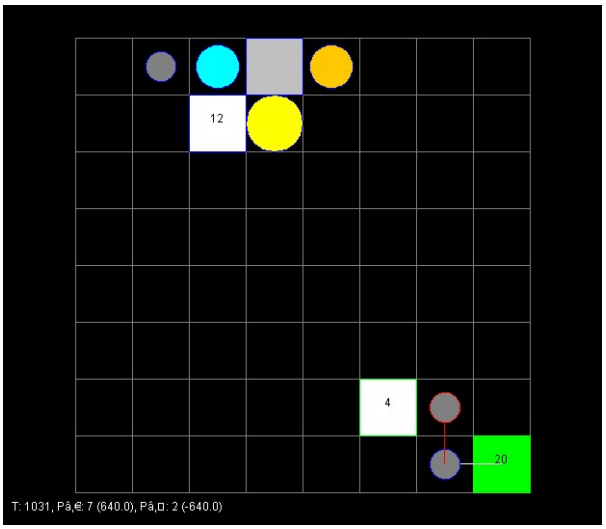

Figure 14: Algorithm 3 (blue) attempting to take resources from the player (red)

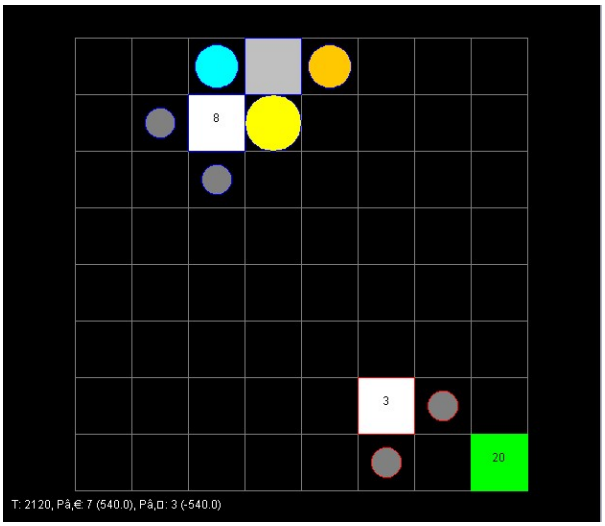

Figure 15: Algorithm 3 (blue) not stealing resources when the player (red) makes another worker in time

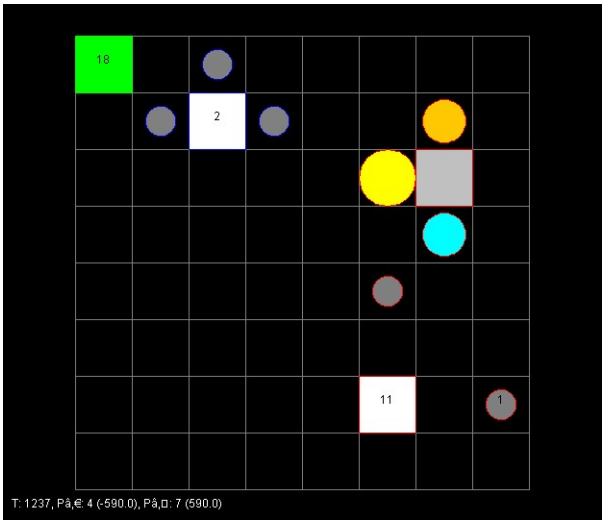

Figure 16: Algorithm 3 (blue) does not make a barracks when the player (red) makes another worker too fast

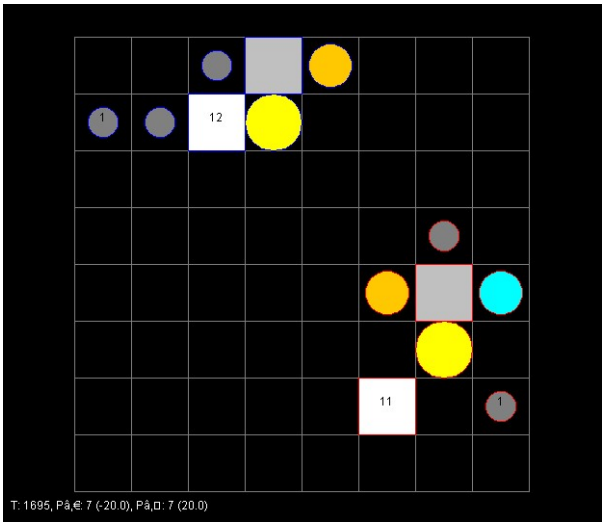

Figure 17: Algorithm 3 (blue) does not attack when the player (red) has only one of a unit type

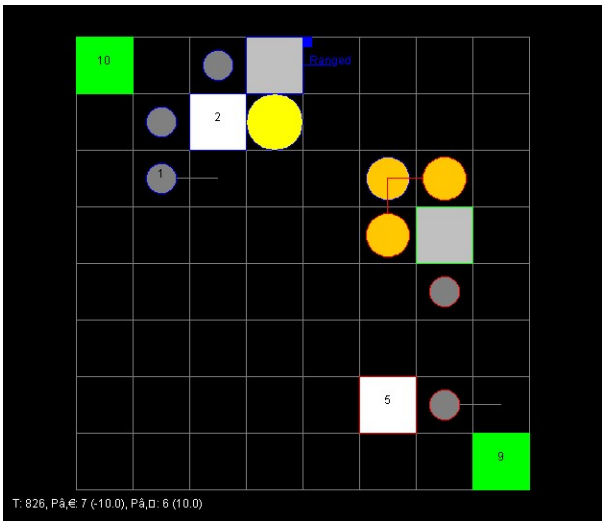

Figure 18: Algorithm 3 (blue) does attack when the player (red) has multiple of a unit type

5.9 Player feedback algorithm 3

Algorithm 3 seems promising as a tutorial-maker, so we let a player play against it to get some first impressions. Checks by players are different than the previous checks.

First we explain a chosen person what type of game MicroRTS is and let him play against our tutorial. While the person is playing, we ask what he/she is doing or trying to do during the game. Whenever this person does not know how to continue with the tutorial, we give tips or hints on what to do next. We use this method until the person has finished the tutorial.

Then we let the same person play against respectively a bot using only random moves and a bot similar to what our own tutorial originated from. When these matches are done we ask what matches were enjoyed most, what was learned from those matches and what information it would rather learn during the tutorial. This is done as a semi-structured interview, where the person has the freedom to tell its own experience and feedback.

We have found an advanced gamer, known to various RTS-games, willing to try out our tutorial. Not knowing about the game MicroRTS, we let him play against our tutorial and get some first impressions. The goal of this approach is to get some early feedback. In addition with our own findings in the earlier checks, we find out if the tutorial is good enough to try out on non-experienced gamers.

The first questions that arose that were not specifically about the tutorial, but on the working of the interface. We explained how the interface worked, since the tutorial is incapable of doing so in this game. We already had this as a limitation of MicroRTS. In other games we may be able to let the tutorial communicate with the player. Despite this, a simple explanation of the interface beforehand solves this issue for this specific game.

Other feedback was in the set-up or scenario of the tutorial game. We set the first location as the tutorial (blue, left-top corner), and the second location as the player (red, right bottom corner). Intuitively the player has the first spot (blue, left top corner), so there was a bit of confusion here. Interestingly, our tutorial is unique in that the player can choose what spawn point it has. Despite this feature, it did not help locate the starting location when it was set to something less intuitively. In addition to the aforementioned explanation, we will set up the starting position more intuitively.

The overall impression about the tutorial was a positive one. In general, the teaching by demonstration seems to work quite good. The player expects the tutorial to show what to do next and copies that behaviour. It is easily recognized that the tutorial is always one step ahead and then waiting for the player to take action.

There was one point during the tutorial where the tutorial stopped demonstrating the next actions. This occurred when the tutorial was supposed to teach combat behaviour. The algorithms requires more than x of a unit type, before demonstrating combat behaviour. The player does not intuitively make multiple of the same unit within this game. The player expected the tutorial to attack and was waiting for instructions or demonstrations by the tutorial.

For completeness we also write down two bugs that are in this version of our algorithm. These may not be directly related to our algorithm. They are related to chooses we made when implementing it. The player noticed and mentioned these bugs as well.

The first bug is that the tutorial stops harvesting when the player makes a second worker too soon. When an action is demonstrated, it should continue doing so until the action is either completed or impossible to finish. It is still a good idea to keep in mind that the value of $x = 1$ in the algorithm can cause these sort of problems. Changing the hard value into a variable range can make our algorithm more general applicable.

The second bug was noticed that when the player attacked units of the tutorial-bot, those units did not fight back. Having this behaviour disabled makes the game too easy and less realistic. Any unit, both from the player and the bot, should automatically fire at opponents within firing range. Our tutorial has this unwanted behaviour inherited from the bot it came from. Interestingly, there are games where this kind of unit behaviour is an actual option. Using that option in this tutorial seems to have a negative effect on the user experience.

Our tutorial is set-up like any other match against bots, so it is possible to let other bots play the tutorial as well. This is unintended, but appreciated behaviour. When the player could not win against a specific bot, it let the tutorial play against that bot to observe and learn.

5.10 Final changes

We have seen that the general impression was positive, however there are still certain things that need to change. The changes do not seem to be in the algorithm itself, besides tweaking some variables. Since these are not directly related to the algorithm we have so far, another approach is necessary. First we take a look at the bugs, their causes and solutions. Then we discuss the possibilities and consequences of the suggested changes.

Since we have mostly applied our algorithm in an existing AI, the bugs are likely existing AI behaviour. The worker that was harvesting stopped that task to build a building. This makes sense when there is only one worker available, as is the case in the original AI. Since our tutorial makes multiple workers, we needed to change the order in which the workers were assigned tasks. This will let the harvesting unit continue harvesting, while another worker is building. Moving the code order to assign harvesting before building, does the job in this case. If there is a scenario where the player has enough resources from start to let a worker build, the existing code order would be fine. This would result in the first worker building instead of harvesting. In the default starting scenario there is not enough resources to build, so we have to start with harvesting.

The other bug is that units did not fight whenever a 'hostile' unit is within firing range. This required a bit more research into the code. It appears any unit of the player that has no action, is given an 'idle' action. When a unit is idle it will not move, but it automatically fires at 'hostile' units within firing range. The existing AI should have this behaviour as well whenever a unit has no action given. This behaviour is not implemented in the existing AI, possibly because they have their own combat rules. These combat rules do not prevent units not getting actions, however it does mask this behaviour. This behaviour can come to light with help of the 'PassiveAI'. If a scenario is made where 'PassiveAI' has units from the start, those units do not fight either. Instead of being 'idle', they have no orders at all.

The tutorial itself was a positive experience up until teaching combat. The combat demonstration was too late, so the value of x should be different for combat behaviour. This makes sense, since using combat had a value of 1 in the earlier algorithms. The higher value was based on teaching how to counter, a functionality we removed from the tutorial.

We can also tweak the variables and see what effect that has. It seems the player does not makes more than 2 worker units, since by that time it is shown how to make a barracks and combat units. We can lower the amount x for worker units to build. We choose the value two, since the tutorial seems efficient in re-using the worker that builds for combat purpose. Any worker above this amount has little to none added value in our tutorial. However, with less resources spent on basic worker units, more can be spent on combat units. This increased the difficulty of the tutorial slightly.

Concerning the variable for worker combat behaviour, we tried setting this to > 1. The effect of this is that the player is attacked even before it has the change to build a barracks with it's second worker. In addition, this attack early on can cause the tutorial to attack the player harvesting worker. This is definitely unwanted behaviour, because that unit cannot fight back while harvesting. This can even result in a loss for the player, so the variable should definitely be higher than one.

Alternatively, we set the worker combat value to > 2. Our first impression is that this behaviour is not demonstrated, since players are not taught to make > 2 units. However, playing styles differ between players, so some people do make > 2 workers. The effect of changing this variable is to decide how soon something is taught.

Concerning the combat values, the suggestion was to make it either demonstrate combat only once or such that the tutorial always loses the combat. That last suggestion is a reversed countering

that we used in our previous algorithms. We can simply choose a weak unit, for example a worker, and fight with that.

After some short experiments it turns out that the tutorial becomes either too easy or too difficult by implementing the previously discussed reverse countering. A problem here is that there is no clear balance between the units. For example, one 'heavy' is as strong as two 'light' units, which is a nice two to one balance. A ranged unit on the other hand, can be taken out by a single heavy or light unit. It cannot be taken out by a singe worker unit, since this unit is taken out before it reaches the ranged unit. Ranged units can also take out multiple heavy or light units, depending on their behaviour.

Light units can be taken out by two ranged units or four worker units. Four worker units is too much for the tutorial, as the tutorial would spend most of its resources on worker units to attack the player light units. This results in the tutorial experience being too easy and less fun. Sending ranged units to take out the light unit makes the tutorial too difficult. If the player reacts immediately, a light unit can take out two ranged units. However, if the player does not react swiftly enough, the ranged units can take out the light unit and all other units within its range. Attacking with the ranged unit in general makes the tutorial too difficult for beginning players.

The other suggestion was to make the tutorial attack only once. This implementation can differ depending on how the game is programmed. It does slightly change the algorithm, which we will see at the results. However, in this game it is as simple as making a class variable that the tutorial can use. For bots in "Age of Empires II", there is a specific call to make rules happen only once. Implementing this is actually less work than finding out strength relationship between units.

Algorithm 4: Tutorial-bot limiting-behaviour update

```
Initialize x as amount of actions to teach for each individual unit/structure;
Initialize pCount as amount of player units and buildings, sorted by type;
Initialize aiCount as amount of AI units and buildings, sorted by type;
forall unit/building type do
    if aiCount < x and pCount >= aiCount then
        Demonstrate building with this unit/building type;
    else if pCount = 1 then
        Demonstrate using this unit/building type, excluding building and combat;
    else if pCount > x then
        Demonstrate combat with this unit/building type, execute only once for each unit
         type;
end
```

5.11 Final feedback

To see if this implementation has the desired effect, we test this version first by the same person who gave the earlier feedback. However, this time we only ask to play against our tutorial. As

an experienced gamer in RTS games and previous experience in MicroRTS, this check is focused around the new combat behaviour. The other fixes were positively welcomed, but those are not the main focus as it does not change anything in the algorithm.

After making combat units the tutorial instantly starts attacking. This is the behaviour we expected, yet the initial feedback is that is feels too fast. As soon as the player makes a combat unit, it is destroyed by the tutorial before the player can use that unit.

Now it is time to let more people play against our tutorial. There are slight changes in set-up compared to the previous player check. Based on the feedback, we let players first play against the bot using only random moves. During this game, we can explain how the interface works and observe how much they learn from a bot that only makes random moves. Then we let the player play against our tutorial and finally play against a "rush" bot similar to what our tutorial originates from.

The next person to play against our tutorial is experienced in RTS-games as well. This person does not know MicroRTS yet. Feedback from this person is similar to that of the previous person. A positive impression on learning how to use units and build units and structures. On the downside, it spend half of its total resources on a building, not knowing the value of that building. In this case, the player already started with that building. Players start the game with this building, called the 'Base'. So it made another 'Base', only to realize afterwards it costs half of its total resources and had no added value. Concerning the combat, this person expected to learn more about how tough units and buildings were. For example when playing a regular match, after our tutorial, it did not know how much damage units did to buildings. Also, most feedback was again on the lack of a good interface. Communication from the tutorial to the player seemed less relevant. A few lines saying what units/structures belong to the player and what the goal of the game is may be enough.

How this is experienced by beginning players is something to find out by letting players without any experience in this game genre play this tutorial. After letting someone play who is new to this game genre, this tutorial is seems difficult. This person also copied the tutorial behaviour up to the combat part. After that, it wasted resources on more buildings instead of combat units, and lost all of its units eventually. Having the first set of units destroyed instantly by the opponent was very confusing and gave a disadvantage to the player. This person did not understand why the units suddenly disappeared, because it happened so fast. The overall impression of the tutorial was a negative one. This makes sense since it was too difficult and it result in a tie, with all of the player units and resources depleted.

Our tutorial is played by experienced RTS-gamers who liked it and less experienced non-RTS gamers for who it is too difficult. The next person who plays this tutorial, is someone without any gaming experience at all. We can expect this person to have a negative experience and the tutorial being way too difficult. Unlike the previous persons who played our tutorial, this person who has never gamed before does not copy the tutorial behaviour. Since everything in the game is new information, teaching by demonstration does not work in this scenario.

Based on the overall feedbacks, it seems that this tutorial is well suited for players who have experience in similar games. With the current rules, it is still too difficult for less experienced

gamers, but there is potential for that group as well. For the last group, non-gamers, this research is not very usable. Other teaching methods than demonstration based are highly recommended if the goal is to attract people have have rarely gamed before.

6 Conclusions

In this research we have seen multiple iterations of a simple set of rules that can turn an existing bot into a tutorial. The first set of rules were so strict that it felt like scripted tutorials. The only way to progress in that tutorial was to do exactly as the tutorial demonstrated.

In the second attempt, progress was more based on player behaviour. This caused unexpected behaviour and little control over the tutorial. Because of this, we decided that every type of action needs its own rule.

Then we encountered unexpected behaviour caused by the bot we started with. This demonstrates how the bot to start with can affect performance. Instead of default behaviour of the bot units, we choose to use the basic behaviour of the player units. After implementing this, we reached an algorithm that was generally well-received by players who already had experience in the RTS-game genre.

With all this information, we can formulate an answer to the question: "Can an adaptive AI effectively be used as a tutorial?". The answer depends on what the audience of the tutorial is. We have not looked at specialised tutorials for players that already know the specific game. For players that did not know the game yet, the success of the tutorial seems related to their experience in similar games. If aiming for players with a lot or at least some amount of experience in similar games, this method of making a tutorial seems promising. For players that usually do not play games and have no experience similar games, another approach is recommended as a tutorial.

6.1 Results

For players who are new to this RTS-game genre, the tutorial was too difficult. We have seen that whenever the player makes a generally bad move, the tutorial does not copy that and increases its relative strength compared to the player. Experienced players can overcome this extra challenge and make few of these moves. For beginning players the opposite applies. They make more generally bad moves and increases the difficulty with that.

We have found rules that work well for demonstrating building and harvesting behaviour to players with experience in similar games. Rules for teaching combat are more difficult to find, possibly because it involved multiple unit types with varying strength ratios. The combat-rules that we did find can be a good start for tutorials in similar games, and then optimized for that specific game.

The algorithm and specific rules we used, are general enough to be directly applicable in similar RTS-games. This can improve the first impression of experienced-players for the game, when a more traditional tutorial is considered too easy. The variables used in this experiment can be altered and should be optimized for each unit type. In addition, the scenario, or settings, should be chosen such that players can easily observe the tutorial. For example, enabling "Fog of War" would prevent players from observing the tutorial.

In other game genres, simple rules like those found here can be applied as well. For example in a brawler game, separate rules can be made for movement, offensive and defensive moves. Building does not apply there, and combat may need to be more specified.

In racing games, it can be more important to focus on giving the opponent the same disadvantages as the player. For example, whenever a player becomes off-track, the tutorial can get off-track as well. The final algorithm in this research does not adept to players giving themselves negative advantages. However this is a welcome addition to the algorithm, as revealed by inexperienced players playing it.

The general impression is that players who have experience in similar games enjoyed the tutorial and learned from it. The less experience someone has in this game-genre, the less it performs in the tutorial and the less effect this learning method has. Someone who had no gaming experience did not watch the demonstrated behaviour at all, because everything in the game was a new experience. When asked, non-experienced players did mention that playing against the tutorial was a better experience than playing against other bots. This can be explained by the pacing, since the tutorial only advances when the player does.

6.2 Limitations

During this research we have seen the effects of this tutorial on a very small scale. Some limitations were already visible, while others can occur when implementing this in more complex games.

The biggest limitation we have seen is the need for a good self-explaining interface. All of the persons who played our tutorial needed additional information on the interface. There are ways in which the tutorial can explain the interface. Those possibilities can vary from game to game.

In the case of our tutorial in MicroRTS, we did not have the ability to chat with the player. Telling the player what button to press or what the general objective is can improve the tutorial. Neither did we have the capabilities to implement visual aid like arrows pointing what button to click. MicroRTS clearly demonstrates some limitations of this tutorial. In other games, for example "Age of Empires II", tutorials do have the possibility of sending text messages to players.

Limitations that count for any game are in the controls. When teaching by demonstration, as is the dominant strategy in our research, the tutorial needs the same capabilities as the player. The tutorial can only demonstrate how to act in certain situations if the tutorial can reach the same situation and the player can make the same actions as the tutorial.

An example of this limitation is in the game "Space Invaders". The opponent controls multiple aliens with the goal of reaching the other side. Players on the other hand, control only a single unit and has no means of reaching the opposite side. Instead they have to prevent the opponent from reaching the other side.

Another more general limitation of teaching by demonstration is that the player has to copy demonstrated behaviour. We have already seen that this does not work for players that are new to the game-genre. These players do not copy the tutorial behaviour and do not advance on their own either. By not advancing during the tutorial, they cannot finish it. A timer may be set to prevent players from getting stuck during the tutorial, but for now this is an unsolved limitation.

6.3 Further Research

When we take into account that this type of tutorial is more popular by experienced players, it may go further than teaching the basics. During this research only a small focus has been on implementing strategies. The potential in combination with this tutorial is nevertheless worthwhile for future research.

We have also seen that some rules are situational, making them optional in the tutorial. In its final version, the tutorial only teaches worker combat when the player makes a sufficient amount of worker units. This can be expanded for example by this strategy used by bots where only worker units are used to win. This can add more value to more experienced players, while it is not taught to less-experienced players.

Games where the opponents do have similar controls are often games where players can play against each other. Examples are strategy games and brawler games, but also sports, racing and boardgames can potentially use this kind of tutorial. Expanding this type of tutorial to those game genres and more complex games is also good for future research.

Since we already see a difference in tutorial-completion based on experience, we can measure this. Then we can use those numbers to bias the tutorial and maybe even select completely different rules based on that. In such a way, experienced players can learn multiple times by playing against the same tutorial. This approach does require more rules, to be found in future research. If we can find more general rules that can be applied across various games, we can expect to see more of this type of tutorial.

Measuring statistics of players during this type of tutorial is also useful for developers. Where players can be given an optimal difficulty setting based on their performance against the tutorial, developers can use aggregated information to improve their game. Finding out how to use this type of tutorial to measure statistics and how to use them are open for future research.

References

[Dee20] DeepMind. Alphago zero. https://www.deepmind.com/research/case-studies/alphago-the-story-so-far, 2020.

[GKB+18] Michael Cerny Green, Ahmed Khalifa, Gabriella A.B. Barros, Tiago Machado, Andy Nealen, and Julian Togelius. Atdelfi: Automatically designing legible, full instructions for games. In *Foundations of Digital Games 2018 (FDG18)*, New York, NY, USA, 2018. ACM.

[iee20] ieeecog. ieeecog. https://ieee-cog.org/, 2020.

[LLC95] MultiMedia LLC. Dictionary.com. https://www.dictionary.com/, 1995.

[lRU20] Intelligent Computer Entertainment lab Ritsumeikan University. openice. http://www.ice.ci.ritsumei.ac.jp/˜ ftgaic/, 2020.

[Ont13a] Santiago Ontañón. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *AIIDE 2013*, pages 58–64. Association for the Advancement of ArtificialIntelligence, 2013.

[Ont13b] Santiago Ontañón. microrts. https://github.com/santiontanon/microrts, 2013.

[res13] resonance22. Resonance22 profile. https://steamcommunity.com/id/resonancesteam, 2013.

[res15] resonance22. Resonancebot. https://steamcommunity.com/sharedfiles/filedetails/?id=473358292, 2015.

[YT18] Georgios N. Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. Springer, 2018. http://gameaibook.org.

A Original RangedRush PlayerAction behaviour

```
    public PlayerAction getAction(int player, GameState gs) {
        PhysicalGameState pgs = gs.getPhysicalGameState();
        Player p = gs.getPlayer(player);
//        System.out.println("LightRushAI for player " + player +
//                " (cycle " + gs.getTime() + ")");

        // behavior of bases:
        for (Unit u : pgs.getUnits()) {
            if (u.getType() == baseType
                    && u.getPlayer() == player
                    && gs.getActionAssignment(u) == null) {
                baseBehavior(u, p, pgs);
            }
        }

        // behavior of barracks:
        for (Unit u : pgs.getUnits()) {
            if (u.getType() == barracksType
                    && u.getPlayer() == player
                    && gs.getActionAssignment(u) == null) {
                barracksBehavior(u, p, pgs);
            }
        }

        // behavior of melee units:
        for (Unit u : pgs.getUnits()) {
            if (u.getType().canAttack && !u.getType().canHarvest
                    && u.getPlayer() == player
                    && gs.getActionAssignment(u) == null) {
                meleeUnitBehavior(u, p, gs);
            }
        }

        // behavior of workers:
        List<Unit> workers = new LinkedList<>();
        for (Unit u : pgs.getUnits()) {
            if (u.getType().canHarvest
                    && u.getPlayer() == player) {
                workers.add(u);
            }
        }
        workersBehavior(workers, p, pgs);
```

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

B Original RangedRush Building behaviour

```
    }

    public void baseBehavior(Unit u, Player p, PhysicalGameState pgs) {
        int nworkers = 0;
        for (Unit u2 : pgs.getUnits()) {
            if (u2.getType() == workerType
                    && u2.getPlayer() == p.getID()) {
                nworkers++;
            }
        }
        if (nworkers < 1 && p.getResources() >= workerType.cost) {
            train(u, workerType);
        }
    }

    public void barracksBehavior(Unit u, Player p, PhysicalGameState pgs) {
        if (p.getResources() >= rangedType.cost) {
            train(u, rangedType);
        }
```

C Original RangedRush Unit behaviour

```
    public void meleeUnitBehavior(Unit u, Player p, GameState gs) {
        PhysicalGameState pgs = gs.getPhysicalGameState();
        Unit closestEnemy = null;
        int closestDistance = 0;
        for (Unit u2 : pgs.getUnits()) {
            if (u2.getPlayer() >= 0 && u2.getPlayer() != p.getID()) {
                int d = Math.abs(u2.getX() - u.getX()) + Math.abs(u2.getY() - u.getY());
                if (closestEnemy == null || d < closestDistance) {
                    closestEnemy = u2;
                    closestDistance = d;
                }
            }
        }
        if (closestEnemy != null) {
//            System.out.println("LightRushAI.meleeUnitBehavior: " +
                u + " attacks " + closestEnemy);w
            attack(u, closestEnemy);
```

```
    }
}

public void workersBehavior(List<Unit> workers, Player p, PhysicalGameState pgs) {
    int nbases = 0;
    int nbarracks = 0;

    int resourcesUsed = 0;
    List<Unit> freeWorkers = new LinkedList<>(workers);

    if (workers.isEmpty()) {
        return;
    }

    for (Unit u2 : pgs.getUnits()) {
        if (u2.getType() == baseType
                && u2.getPlayer() == p.getID()) {
            nbases++;
        }
        if (u2.getType() == barracksType
                && u2.getPlayer() == p.getID()) {
            nbarracks++;
        }
    }

    List<Integer> reservedPositions = new LinkedList<>();
    if (nbases == 0 && !freeWorkers.isEmpty()) {
        // build a base:
        if (p.getResources() >= baseType.cost + resourcesUsed) {
            Unit u = freeWorkers.remove(0);
            buildIfNotAlreadyBuilding(
                u,baseType,u.getX(),u.getY(),reservedPositions,p,pgs);
            resourcesUsed += baseType.cost;
        }
    }

    if (nbarracks == 0 && !freeWorkers.isEmpty()) {
        // build a barracks:
        if (p.getResources() >= barracksType.cost + resourcesUsed) {
            Unit u = freeWorkers.remove(0);
            buildIfNotAlreadyBuilding(
                u,barracksType,u.getX(),u.getY(),reservedPositions,p,pgs);
            resourcesUsed += barracksType.cost;
        }
    }
```

```
        // harvest with all the free workers:
        for (Unit u : freeWorkers) {
            Unit closestBase = null;
            Unit closestResource = null;
            int closestDistance = 0;
            for (Unit u2 : pgs.getUnits()) {
                if (u2.getType().isResource) {
                    int d = Math.abs(u2.getX() - u.getX()) +
                        Math.abs(u2.getY() - u.getY());
                    if (closestResource == null || d < closestDistance) {
                        closestResource = u2;
                        closestDistance = d;
                    }
                }
            }
            closestDistance = 0;
            for (Unit u2 : pgs.getUnits()) {
                if (u2.getType().isStockpile && u2.getPlayer()==p.getID()) {
                    int d = Math.abs(u2.getX() - u.getX()) +
                        Math.abs(u2.getY() - u.getY());
                    if (closestBase == null || d < closestDistance) {
                        closestBase = u2;
                        closestDistance = d;
                    }
                }
            }
            if (closestResource != null && closestBase != null) {
                AbstractAction aa = getAbstractAction(u);
                if (aa instanceof Harvest) {
                    Harvest h_aa = (Harvest)aa;
                    if (h_aa.target != closestResource || h_aa.base!=closestBase)
                        harvest(u, closestResource, closestBase);
                } else {
                    harvest(u, closestResource, closestBase);
                }
            }
        }
    }
```