Using Behaviour Trees and back chaining for Starcraft AI

GROUP 3

Marco Schouten 1998-05-22 schouten@kth.se

Justin Salér 1992-12-07 justinr@kth.se

Abstract

AI and games go hand in hand and the Real-time Strategy game Starcraft is a great playground for testing different AI paradigms. In this project, we have tested the ability of behaviour trees and back chaining for controlling a starcraft AI system. Our AI competed against five other groups, with a similar mission, and we were able to score adequate results. Our AI system was able to defeat two of the teams but lost to two others. We used a single behaviour tree for controlling our entire solution, separably into a building tree and a combat tree. Although the use of a behaviour tree led to a reactive solution, the lack of modularity, the ability to distribute logic and tasks between different components, served as our system's greatest weakness. Finally, after consulting with a Starcraft professional, we went with a strategy called "Dragoon rush", which required a greater level of combat micromanagement, than what could be easily implemented with Behaviour Trees.

1 Introduction

Video games are for many people their first encounter with artificial intelligence (AI). In video games, AI is often used for controlling enemies, who have more information and a smaller range of available actions than a human player. Due to these attributes, these AI systems does not need to be very advanced. Creating an AI system that would be able to, with the same limited information and a vast array of possible actions as a human player, defeat a player would be more complex. This is however possible with state of the art AI techniques.

The idea of creating machines and computer capable of beating humans in games with their own rules is not a new one. For as long as we had digital computers, researchers have been attempting to teach them to defeat humans in games, such as chess. As computer and video games have become more popular, so has to create AIs that can defeat humans. Examples of video games, which have been researched extensively in the AI field, is DOTA and Starcraft.

We have created an AI bot, for Starcraft using behaviour trees (BTs). Behaviour trees are an AI paradigm, famous for their adaptability and dynamicity, which is commonly used in robotics, but also video games. The bot competed against four other teams with the following rules:

- All groups must play Protoss. [1]
- Play only the map Fortress [2]
- StarterBot folder within UAlbertaBot is the basline for the code. [3]
- BWEM library for map analysis.
- teams will play twice against each other, once at home, once out.

After conversing with a Starcraft professional we went with a quick strategy, called ”Dragoon Rush”. We were able to obtain adequate results, ending up in a third place out of five. Some of our strengths were a strong algorithm for scouting, the action to obtain the location and development of the opposite team, and a good focus fire combat system. However, the strengths of the behaviour tree were overwhelmed by its weakness, the unnecessary overlay. Additionally, a single behaviour tree was used for all the units, without any individual control systems for the units. A multi-agent system approach

[1]Protoss Race: https://liquipedia.net/starcraft/Protoss
[2]Map Fortress: https://liquipedia.net/starcraft/Fortress
[3]UAlbertaBot https://github.com/davechurchill/ualbertabot/wiki

where each unit had its control system would have yielded better results, but was impossible with a behaviour tree due to computation limitations.

1.1 Contribution

In this project, we implemented the BOT strategy through the use of Behavior Trees and the BWAPI library. Behaviour trees were used both micromanagement and macro-management. A simple version of back chaining was implemented, where when input a unit or building type, the program could recursively create a fitting PPA (post-condition pre-condition action) BT, with subtrees for gathering mineral or gas and building required units, building or supply sources.

Using behaviour trees for starcraft AI has been investigated previously, but not to the same extent. In the article by Larsen et. al [S. et al., 2010], it was limited to combat macro-management. In our project, BTs were used both for macro and micromanagement, and both for building and combat. We also pioneer by implementing back chaining in our solution.

1. The BTs were coded in C++. The implemented node types were action, condition, sequence, fallback and parallel.
2. Additionally, within the BT actions and conditions we took advantage of the BWAPI library to get information about the map and the state of the game (e.g. getting a list of enemy visible units).
3. The BT which was ticked every frame was separated into two sub-tree, with a parallel at the root. One of the sub-trees was used for controlling the building and the other for combat.

.

1.2 Outline

In Section 2 we explore the available literature for BOTs that plays Starcraft to provide an overview of the pros and cons of different approaches. In Section 3 we explain the details of our implementations. In Section 4 we analyse the results of our findings. In Section 5, we give a conclusion on our results and what could be improved.

2 Related work

Below here we investigate the available literature.

2.1 Behaviour Trees

Behaviour Trees [Colledanchise and Ögren, 2018] are an efficient technique that decomposes the behaviour of agents in a set of tasks organised in a tree-like structure. The core strength of BT is its flexibility. It is easy to add new features to or to make changes to the behaviour of an agent. This is due to the structure of the tree that is traversed by Ticks [4]. For these reasons, BT is preferred for developing such AI compared to older approaches like Finite State Machines. [Ho and Komura, 2011].

2.2 Prof D. Churchill Video Lectures

Prof. Dave Churchill has dedicated over 10 years in the development of Starcraft BOT and BWAPI, a library that works as interface between C++ Scripts and Blizzard's Game engine. He has uploaded some video tutorials to aid others to better understand the workings of Starcraft BOT. In particular he had three videos, the first describing the installation process [Churchill, 2021a], the second about general Stracraft strategy and game rules [Churchill, 2021b] and the third [Churchill, 2021c] providing more details and example scripts about describing how test the interface with the Bot. [5]

Starcraft, as in most RTS games, starts the game by gathering resources (essential for building units), build buildings and combat enemies. The game is won whenever all the enemy buildings are destroyed. In the game there are three main categories of strategies that are effective against each other in a rock-paper-scissor manner: Rush > Expand > Defend > Rush.

- Rush, is about going aggressively and trying to attack the enemy as early as possible.
- Defend, builds defensive units (e.g. towers) and strategically place buildings in order to create choke points that lead to a numerical advantage with respect to fighting units.
- Expand, is about building an expansion as fast as possible in order to win the long-term game, focusing on maximising the economy gains.

Rush wins against Expand because while the expanding player invests resources in building buildings and more workers, the rushing player has a

[4] A Tick is a process of visiting the next node in the tree. Each Tick returns Success, Running or Failure

[5] Video lectures links: (a) https://youtu.be/lSmkDjFm3Tw , (b) https://youtu.be/czhNqUxmLks , (c) https://youtu.be/FEEkO6__GKw

numerical advantage of combat units, hence much greater combat power. Defend wins against Attack because, in terms of economy, defensive units are cheaper hence giving the defensive player a slight edge. Additionally, the defender's new units spawn directly in the base (the zone subject of attack) while the attacker's units need to wait the travel time. For this reasons, the defending player tends to have more units, hence stronger fighting power. Expand wins against Defend because long term they would out-resource the defending player.

Build order is another critical concept for RTS games: it determines the sequence of buildings actions that the player performs at the beginning of the game. This determines the strategy the player is going for (e.g. Rush) and is very important as, before scouting, player A has no information whatsoever about player B. Having a solid Build order may lead to a huge advantage, however, once scouting takes place, the other player can counter react and specialize its build order to save its game.

In the third video, He presents the basic structure for the StarterBOT project, namely how the main.CPP works, additionally, he shows some common examples about basics scripts. In a nutshell, StarterBOT is based on Event-driven programming, a paradigm where the flow of the program is decided by events like user inputs such as key presses or mouse clicks. The OnFrame Method is the core of the program and is called every frame and runs up to 1 second, (it has a timeout upper limit). But there are other useful events such as OnCreate, which is triggered once a unit is created, or onStart to initialize the variables or settings.

2.3 Gathering

In Starcraft, only one probe (the Protoss gatherer unit) can gather mineral from a mineral field at a time. On the map "Fortress", there are only eight mineral fields. This means that when more probes should gather minerals than there are fields some kind of queue system is required.

With the built-in system, a worker trying to gather from an occupied mineral field will enter an idle state where it will wait for free fields. But as shown by Rooijackers M. and M. H. M. Winands, the people behind LetaBot, three times winners of SCCAI, this solution is not optimal [Rooijackers and Winands, 2017]. They suggest a cooperative approach with a smarter queue where all the gatherers are queueing for all the available mineral fields. Although we didn't implement any advanced strategies for gathering, the knowledge of the weakness of the built-in strategy helped us better understand mineral gathering and actively avoid idle workers and gathering bugs.

2.4 Reinforcement Learning

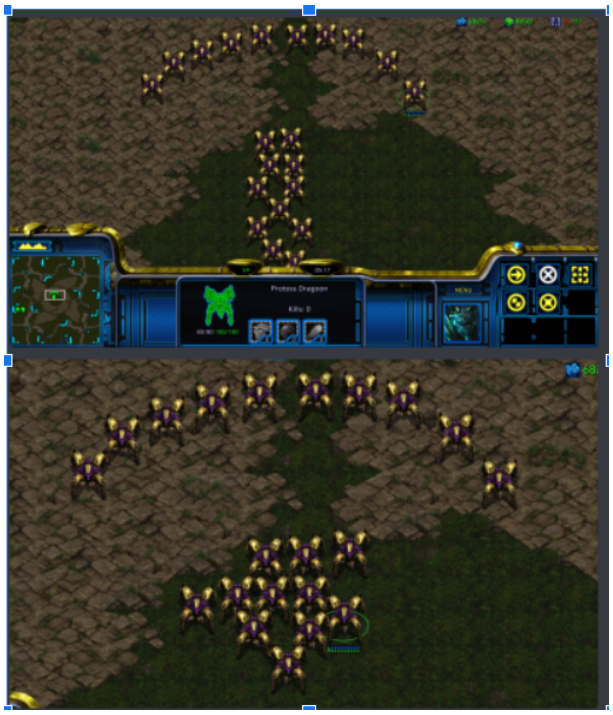
Deep multi-agent reinforcement learning (RL) is a fast-developing research area that can deal with a particularly challenging class of problem: cooperative multi-agent learning. Each agent must adapt its behaviour according to the limited and incomplete information under decentralization constraint. A major issue that arises in this problems is that there is no standardized benchmark to test this system beyond naive toy examples like grid worlds. [Samvelyan et al., 2019] tested RL on the popular Starcraft game. Each agent (within a given team) shared the same reward function and has an action space between 7 or 70 actions. Performances of this RL algorithm are quite good and seems the right way forward, given sufficient hardware equipment.

2.5 Discussion with Starcraft Pro: DragOn[NaS]

We had the pleasure to talk with Alex Empey, (alias Dragon[NaS]) a Starcraft Professional Player [6]. We discussed "Build Orders", the initial sequence of steps to execute the same way every game, in other words, the "opening". This is particularly important because [Empey, 2021] at the beginning of the game, you have no information about the opponent, and you are forced to take blind decisions which will have a huge impact on the resources, unit amounts and position you have at a given time of the game. He argued that teaching dragoons as a priority are the most common and standard way to play PVP. There are several ways to do it, depending on how safe you want to play. For example, a more aggressive version of this build order consists of sending an early scout (around 10 supply) and skip 1 defensive Zealot to have a faster Dragons. However, this decision makes your base vulnerable to early harassment by the enemy units. About scouting he said that it is worth it, so you can try and react to your opponents opening build. There are also different schools of thought on how late to scout, generally it's with a probe between 10supply and 15supply. You Sooner is safer, later is more economic. You can also scout with a dragoon and not use a probe at all which would be very greedy.

He also argued that having a circular line of Dragoons around chokepoints will make you win even when outnumbered by a few units.

[6]Alex DragOn Empey is a Protoss player from Canada, he was the founder and leader of the North American Squad, and leader of Team Canada since 2012. He also coached CPL seasons 1-4, winning 1st place in seasons 1 and 2. https://liquipedia.net/starcraft/DragOn



3 Proposed method

Our solution which was coded in C++ uses Behaviour Trees together with BWAPI. A single behaviour tree is used both for combat and building. Conditions and actions were created to fit the BT paradigm. Some additional logic is also added handled by an "Information Manager", responsible for keeping track of game-wide information.

3.1 Behaviour Trees

The behaviour tree has five different types of nodes: action, condition, parallel, fallback and sequence. These nodes were then created together as a tree.

Parallels, fallbacks and sequence are composite nodes which will tick their children nodes based on their result. A parallel always ticks all their nodes, fallbacks tick their nodes until one of them returns "success" and sequence tick their nodes until one of them returns "failure" or "running".

Actions and conditions have no children. A condition immediately returns "success" or "failure" based on the current state and action will attempt to perform an action and return "success" or "failure" if it can complete it and "running" if it can't.

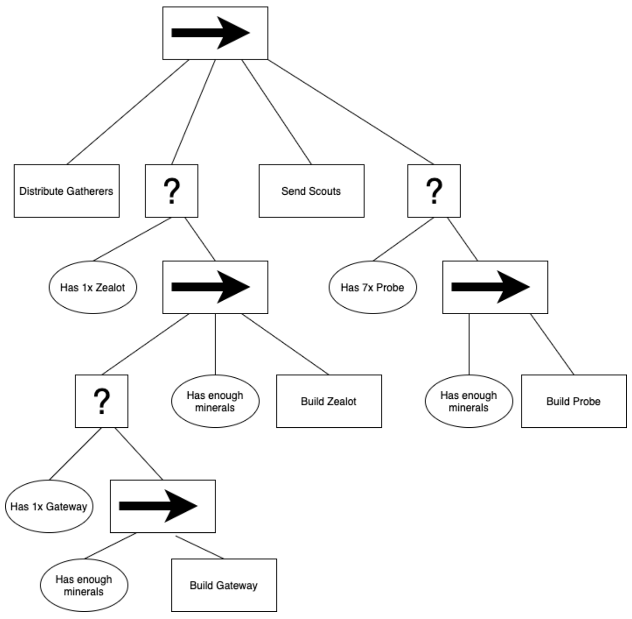


Figure 1: An example tree for building, created by backchaining. The arrow marked blocks are sequence and the blocks with question marks are fallback. The round blocks with text are conditions and the square ones are actions.

To separate the building logic from the combat logic, the tree was constructed so that it had a parallel at its root, with one child for building and one child for combat.

3.2 Building

Despite the name, the Building tree handles not only building buildings but also scouting, building units, ordering upgrades and gathering. An example can be seen in figure 1 The root of the building tree is a sequence, with most of its children being PPA BTs returning success whether a building, upgrade or unit existed, and if not they would try to build them. Additionally, the tree has a gathering action, triggered every time to ensure that we do not have any idle workers, and a scout action that always returns success, but is only ticked when enough buildings and units are created. The Building Tree was created to ensure that the build order was followed, and that scouting and gas gathering happened at the correct time relative to the build order.

Building buildings and units and research upgrades use PPA BTs, which could be created with back chaining. By giving a simple command such as "Build Dragoon" or "Research Longer Range", the back chaining algorithm was able to create a fitting PPA BT. When a PPA was ticked, it would construct any pre-required units or buildings, gather required material and then create or research the requested building, unit or upgrade.

Using behaviour trees for the building had a strong weakness. Since it takes time to create a building or research an upgrade, if the conditions returned success first only when an order was completed, a waiting time would be added. The fix to that was for the conditions to account for units/buildings/upgrades in progress.

But this didn't mitigate the error completely. The default StarterBot code didn't use queuing, a feature in Starcraft. This led to that every time we needed more units than we had factories, buildings that could build the unit, the bot stopped the build order. This was circumvented by enabling the use of queueing, and as long as we had enough units in a queue, proceed to the next step in the building BT. This however led to the suboptimal unit creation when a new factory was completed while queueing units. E.g. we have scheduled five dragoons at one gateway, the dragoon factory, and then we build two more gateways. In this scenario we want to spread the dragoon creation between the gateways. An algorithm was added to do that every time a new factory was built. This was however not optimal either, since it required order cancellations and refunding resources is not instantaneous.

3.3 Combat and Scouting

Scouting is a component of our solution responsible to gather information about the enemy units, builds, and movements to better react and counter its strategy. Our scouting consists of selecting a worker, and make it move to each of the starter spawning locations in a circle. Then once he finds about enemy Nexus location he saves it within a Singleton Information Manager Class that stores all sorts of information.

The combat component is responsible for micro-managing the units to make them fight with the best odds of success. Our combat system consists of selecting the best enemy unit according to a formula evaluation which takes into account the distance, firing power and health of enemy units. Once a target is selected, our platoon of ranged units (Dragoons) will attack with a single target focused fire on that enemy. In this way, we took advantage of the strengths of ranged units.

Our strategy consisted of building a platoon of 8 Dragoons and then sending them to move them close to the enemy base, where they could spot enemy dangerous units to attack them.

4 Experimental results

In this section, we will comment on the best results for each problem, and discuss how our methods can be improved.

4.1 Experimental setup

We ran our experiments in Starcraft Broodwar 1.6.4. To be able to use Starcraft Broodwar, we also used an external program called Chaos Launcher. Chaos Launcher aided us with using the BWAPI library to connect our bot with the game. The bot was written in C++ using Visual Studio. The behaviour tree logic was created from scratch using standard C and C++ libraries.

4.2 Analysis of Outcome

As seen in table 1 (we are group 3), we scored third place out of five teams. We were not the only team using behaviour trees, with group 6 using it for their micro-management. Neither of the top two teams, Group 2 and Group 5, however, used it at all.

Albeit both we and group 6 used BTs, they used them only for micro-management. A common trend between the different teams, including the winning team, is using a lot of different managers for different parts. Although our solution is compact and thus easy to understand, a more modular approach could have been implemented to handle each aspect.

Where we stuck with "Dragoon Rush", both of the stronger teams had a grander plan, which included several different offensive units and upgrades, as well as base expansion. Although this approach is more naive in competitive play, being more similar to an intermediate player, it proved very strong against our bots.

But they didn't win by chance. Both their macro-management and micro-management bested us. Their combat management made them win battles with equally sized armies. Although our units had a strong single-target strategy, their ranged units had to kite. The winning team, Group 5, had implemented both individual retreats when a unit needs to recharge their shields they back of, and collaborative retreat, avoiding battles where they are weaker.

Also in macro-management, both Group 2 and Group 5 had a high level of flexibility in their strategies. For example Group 2 had several different build orders and build plans which varied depending on the current game state.

	Wins	Losses
1 (G5)	8	0
2 (G2)	6	2
3 (G1)	**4**	**4**
4 (G10)	2	6
5 (G16)	0	8

Table 1: Results from the final. We scored third. Each team faced off against each other twice.

5 Summary and Conclusions

Using behaviour trees for Starcraft was able to get apt results, but using a single behaviour tree for the entire system is suboptimal. Additionally, several of our systems, such as combat and building could have been improved, which might have been easier if avoiding the BT paradigm. Our simple strategy, "dragoon rush", while a great choice for pro-play, required a level of combat micromanagement which we didn't have.

Controlling every unit continuously with the same overarching BT, led to a large loss in modularity. Modularity, the ability to distribute logic and tasks between several sub-systems, is one of the greatest advantages of AI players over human players. An AI can control every unit separately, using a multi-agent approach, which was harder to manage with our BT approach. Lastly, the StarterBot which we built our solution upon was using a listener API to the game, where it was notified every time a change occurred. Using this listener API together with unit-specific state machines could have reduced the number of times our AI would need to send actions through the BWAPI.

Albeit Dragoon "rush" is a great counter strategy towards zealot-heavy ones, it requires some kind of flanking or kiting micromanagement. Without this, you get the opposite effect where our dragoons got overwhelmed by the zealots and easily destroyed. Zealots are also faster to build, which made us easily overwhelmed when the enemy decided to build a lot of zealots early on.

A more flexible approach where we choose what to build on the current situation would have been preferred. This was also one of the initial motivations behind the use of BTs. Continuously evaluating the game state and select the currently best choice of units, upgrades and buildings, could have led to better results. This would have also included expansion to other bases, something that was not implemented since it was not required for the "dragoon rush" plan.

References

[Churchill, 2021a] Churchill, D. (2021a). *Video Lectures about Starcraft and BWAPI*, UAlbertaBot - Installation Instructions and Tutorial.

[Churchill, 2021b] Churchill, D. (2021b). *Video Lectures about Starcraft and BWAPI*, Introduction to Starcraft, Strategy, and Bot AI Programming.

[Churchill, 2021c] Churchill, D. (2021c). *Video Lectures about Starcraft and BWAPI*, STARTcraft - Complete Beginner Starcraft: Broodwar AI Programming Tutorial with C++ / BWAPI.

[Colledanchise and Ögren, 2018] Colledanchise, M. and Ögren, P. (2018). *Behavior trees in robotics and AI: An introduction.* CRC Press.

[Empey, 2021] Empey, A. (2021). Dragon[nas]. *Discussion over text messages.*

[Ho and Komura, 2011] Ho, E. S. and Komura, T. (2011). A finite state machine based on topology coordinates for wrestling games. *Computer Animation and Virtual Worlds*, 22(5):435–443.

[Rooijackers and Winands, 2017] Rooijackers, M. L. and Winands, M. H. (2017). Resource-gathering algorithms in the game of starcraft. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 264–271. IEEE.

[S. et al., 2010] S., L., J., G., K., S.-J., T., O., and L., H. P. (2010). Applying behavior trees to starcraft ai. Aalborg University.

[Samvelyan et al., 2019] Samvelyan, M., Rashid, T., de Witt, C. S., Farquhar, G., Nardelli, N., Rudner, T. G. J., Hung, C.-M., Torr, P. H. S., Foerster, J., and Whiteson, S. (2019). The starcraft multi-agent challenge.