BABEȘ-BOLYAI UNIVERSITY CLUJ-NAPOCA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

SPECIALIZATION COMPUTER SCIENCE

DIPLOMA THESIS

ARTIFICAL INTELLIGENCE IN REAL-TIME STRATEGY GAMES

Supervisor  
Conf. Univ. Dr. Crăciun Florin

Author

Kerezsi Krisztian Alex

2016

Contents

[1. Introduction 2](#_Toc452778978)

[1.1 Real-Time Strategy Games 2](#_Toc452778979)

[1.2 Motivation 4](#_Toc452778980)

[1.3 RTS AI Competitions 4](#_Toc452778981)

[2. Application Programming Interface (BWAPI) 6](#_Toc452778982)

[2.1 API Capabilities 6](#_Toc452778983)

[3. RTS Sub-Problems 7](#_Toc452778984)

[3.1 Strategy 7](#_Toc452778985)

[3.1.1 Learning and data collecting 8](#_Toc452778986)

[3.1.2 Enemy modeling 8](#_Toc452778987)

[3.1.3 Strategic stance 9](#_Toc452778988)

[3.1.4 Army composition 9](#_Toc452778989)

[3.1.5 Build order 10](#_Toc452778990)

[3.2 Tactics 10](#_Toc452778991)

[3.1.6 Scouting 11](#_Toc452778992)

[3.1.7 Combat timing 12](#_Toc452778993)

[3.1.8 Building placement 12](#_Toc452778994)

[3.2 Reactive control 13](#_Toc452778995)

[3.2.1 Unit micro 14](#_Toc452778996)

[3.2.2 Multi-agent pathfinding 14](#_Toc452778997)

[4. KBot 17](#_Toc452778998)

[4.1 Application structure 17](#_Toc452778999)

[4.1.1 Unit layer abstraction 17](#_Toc452779000)

[4.1.2 Action layer abstraction 19](#_Toc452779001)

[4.1.3 Coordinator layer abstraction 21](#_Toc452779002)

[4.2 Frame management 29](#_Toc452779003)

[4.3 Performance optimization and method profiling 31](#_Toc452779004)

[4.4 Map and construction analysis 33](#_Toc452779005)

[5. Bibliography 34](#_Toc452779006)

# Introduction

Introduced in the early 1990, real-time strategy games have recently become popular for artificial intelligence research. In the past couple of years significant advancements have been made in RTS games with contribution from many fields within computer science and engineering. Motivation for RTS AI has also grown rapidly with the emergence of competition such as Google AI Challenge, StarCraft AI Competitions (organized by SSCAI and AIIDE).

## Real-Time Strategy Games

Real-Time strategy games can be classified as strategic video games which simulate military warfare on various scales. Players assume the role of a military commander in charge of a group of forces which must build an economy, construct bases (buildings and defenses), and create a combat army (train units and research new technologies/abilities in order to increase the capabilities for their units) and defeat the enemies by taking control of their command centers or destroying their armies and bases. RTS games vary in size and complexity. However they share some common traits from the traditional strategy games:

**Real-Time**: RTS games are played in real-time, meaning that each player can perform several actions as fast as the game is executed (an average of 30 frames per second), and the game progresses normally even if no actions are given. This is unlike traditional games like ***chess*** or ***go***where players have several minutes to perform an action, and the game cannot progress until a player has acted or the turn’s time has reached the permitted limit. For example StarCraft runs on 24 frames/second, in other words, a player can perform an action once every 42 milliseconds.

**Simultaneous Moves**: Arises when players have to make their strategy choices simultaneously, without knowing the strategies that have been chosen by the other player(s). Additionally, these actions may be durative, in other words, they require some time to complete.

**Imperfect information:** Players in RTS cannot see their opponent’s units and actions unless they are actively scouting them. Typically, a map is initially covered by a fog-of-war (a layer covering the maps locations that are not explored or the player has no vision over it).

**Non-Determinism**: Some RTS games have non-determinism in their actions. This characteristic refers to a random event occurring while performing a constant action. For example, in StarCraft, units that are positioned on different heights, have a small change of missing their target.

**Multi-Unit Control**: RTS games allow users to control multiple units at the same time, with each able to be given individual actions. This means that at any given state there may be an exponential number of possible actions with respect to the number of units the player controls.

**Complexity**: The complexity of RTS games is much higher than traditional games, in terms of state space size, the number of actions can be performed at any time step, and the number of actions required to reach the end of a game. For example the number of possible states in ***chess*** is approximately 1050, Go has around 10170, while StarCraft has shown to have at least 101000 as a lower bound [8].

## Motivation

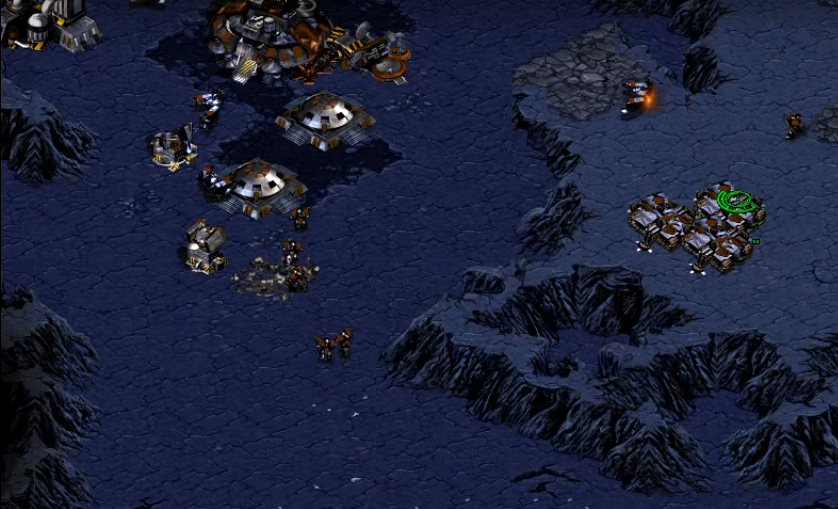
When **StarCraft** was released in 1998, it captured the video game world in a way never seen before, with millions of players playing competitively over LAN and on Blizzard’s battle.net servers. The game has been sold in millions of copies and it became one of the most popular games in the world. In South Korea, so many peoples were playing StarCraft that the Ministry of Culture, Sports and Tourism formed KeSPA, the Korean e-Sports association to manage and promote the professional play of the game in their country. StarCraft has been played professionally in Korea and around the world ever since, with millions of dollars in prize money being awarded annually. With such an established industry and competitive RTS gaming we can motivate to develop better agents that can play against players not only for competitive purposes but also to enhance the player’s multi-tasking abilities.

**Creating better AI agents:** Recent advances in were made in the gaming industry for traditional games. AI agents have been created that are capable of defeating human world champions at several games like: *Chess*, *Go*, *Texas* *Holdem* poker. [1] Competitions, whether it is Human vs. Human, Human vs. AI or AI vs. AI, has always been a motivating factor to research on because it combines elements from mathematics, multi-unit control and prediction.

In my opinion, artificial intelligence will provide a key aspect to our future not only in the gaming industry, but also in other domains such as: medicine, biology, chemistry, image processing and many other domains we can think of.

## RTS AI Competitions

Recently AI competitions became popular due to their challenging properties, with the goal of beating a professional human player at popular RTS games such as StarCraft. These competitions had several categories focusing on sub-problems in RTS games such as providing a good build order and resource management in order to survive attacks and avoid resource exhaustion [2].



# Application Programming Interface (BWAPI)

The Brood War Application Programming Interface is a free and open source C++/Java framework that is used to interact with the popular real-time strategy game **StarCraft: BroodWar**. Using BWAPI, students, researchers and hobbyist can create Artificial Intelligence agents that can play the game without any human interaction [3].

BWAPI reveals the visible parts of the game state to AI modules by default. Information on units that have gone back into the fog of war is denied to the AI. This enables programmers to write competitive non-cheating AI’s that must plan and operate under partial information conditions. BWAPI also denies user input by default, ensuring the user cannot take control of the game units while the AI is playing. These defaults can be changed for flexibility unless enforced by the Tournament Module.

## API Capabilities

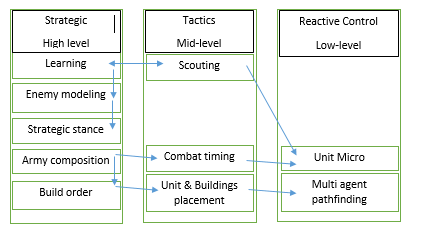
* Write competitive AI’s for StarCraft by controlling individual units.
* Read all relevant aspects of the game state.
* Analyze replays frame-by-frame, and extract trends, build orders, and common strategies.
* Get comprehensive information on the unit types, upgrades, technologies, weapons and more.
* Study and research real-time AI algorithms in a robust commercial RTS environment

# RTS Sub-Problems

Real time strategy games are incredibly complex, even for professional player. In order to manage the games complexity we break it into sub-problems. The sub-problems are not necessarily independent. This approach requires a level of abstraction in order to be trackable by humans. Researchers have adapted the divide and conquer strategy in order to treat each sub-problem and reduce the overall complexity.

We can categorize the RTS sub-problems based on the time scale that we deal with and the level of abstraction the problem requires in order be approachable. These categories are **Strategy**, **Tactics** and **Reactive Control**.

These categories mimic a military command hierarchy, both in terms of command as well as information processing.



## Strategy

At the highest level of abstraction comes the strategy and corresponds to the decision making of the agent/player in an RTS game. Each decision is crucial because it can influence the games odds in the favor of the player that takes appropriate decisions in appropriate times. In order to get a better overview of the applied strategy in RTS games, I’ve pointed out some of the main keys on which I will be focusing on:

* Learning and data collecting
* Enemy modeling
* Strategic stance
* Army composition
* Build order

## Learning and data collecting

In order to act, a player/agent needs to collect as much information as possible from the map and from the enemy players regarding their buildings, army and positioning. Examples of this game data collected can be rules, unit, properties actions etc. In RTS games, players who have more information about resource locations and enemies, have a significantly higher change to win the game by bringing the appropriate decisions for the states in which they are in.

In order to collect data and process it, a player usually sacrifices one or more of his units, sending them to explore and find where the enemy’s bases are located. It is important to know that that the more time with respect to the space explored is obtained, the higher changes are for the player to win.

## Enemy modeling

In the beginning, the players have no vision or information about their enemies, but as the game progresses, more and more information is obtained such as race, buildings, number of units or map positioning. A player must attempt to learn as much as possible to get a better predictions of what the next actions might be. An example of a proper enemy modeling method would be the hidden Markov model (HMM) in which the system being modeled is assumed to be a Markov process with unobserved (hidden) states [9]. In simpler Markov models (like Markov chain), the state is directly visible to the observer, and therefore the state transition probabilities are the only parameters. In a hidden Markov model, the state is not directly visible, but the output, dependent on the state, is visible. HMM’s are also known for their application in temporal pattern recognition such as speech handwriting, gesture recognition etc.

## Strategic stance

The player’s strategic stance determines the style of the play in an RTS game which corresponds to a balance between aggression and economic state. When a player has a large number of resources, his units production is at an exponential rate, he may adapt an offensive style whereas a player with low economy and small army might adapt a defensive stance in order to hold up to the enemy forces. There are various styles that can be adapted also depending on the race and positioning of the player.

## Army composition

Army composition is decided by the strategic stance, with special consideration to the predicted opponent’s army composition. Each unit type in an RTS game has its own unique properties such as attack style, attack range, movement speed, movement type, hit points and mana points. These complex interactions between units make an army composition difficult. Most of the time, the enemy’s stance determines the army’s composition.



## Build order

Once the army composition has been chosen, the army units must be built by the player using one of his workers to build structures that allow creating supply and units using the gathered resources. The sequence of actions taken to arrive at a given set of goal units is called build order. Build order can be described as a resource allocation problem which features concurrent actions. As important as it is for developing the strategy, build orders often get hardcoded due to their efficiency issues. An alternative for hardcoding would be to implement a priority based system where each build would have assigned a priority value. Based on this value, the agent would know what action follows the current one.

## 3.2 Tactics

One step down in the abstraction level, we can find the tactics category, which is responsible for obtaining strategic goals. Tactics are specially focused on actions that take a small amount of time to perfume and obtain. For human players tactics involve much more flexibility due to the quick issuing commands that can be done during the game over units, builds and other elements that are involved in the gameplay process.



## Scouting

The act of gathering information in RTS games is recognized as **Scouting**. Games like StarCraft have their maps covered by the fog of war, disallowing the player to predict or act with respect to his opponent’s strategy. In order to get information about the map, the enemy positioning and build progression, a random worker is assigned to explore the map. Each unit has a different range view, and enemy units can be seen only in the vision radius of the players units. Some units are invisible (Ghosts, Dark Templar or any burrowed Zerg unit), so they can be detected only with special equipment or detector units. For example the Zerg’s Overlord can detect any close invisible units [4].

## Combat timing

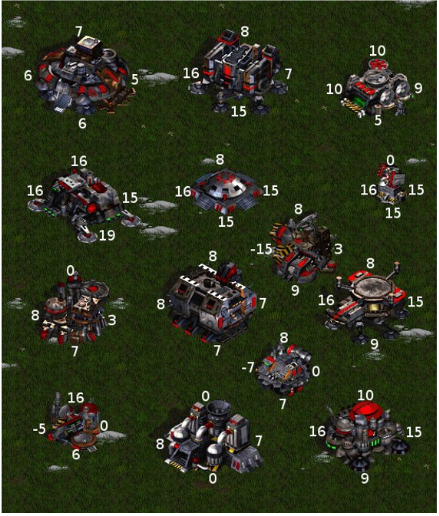
Once an army has been built, the player must decide where and when to land the next attack. In RTS games, attack timings are very important due to the rock-paper-scissors nature of army composition (i.e. the unknown state or the lack of information regarding the enemy’s army determines a random decision making system where a player has to make an assumption and act accordingly without knowing any environment data).

Most combat related data are focused on low-level abstraction, rather than the more abstract task of planning when and where squads should attack in RTS games.

## Building placement

After deciding what the next required building is, the player should consider placing the building in such a way that does not block the players own units in moving, and increases the difficulty for the enemy to invade the base. At the moment, creating a perfect build placement is rather difficult to obtain because the base should fit and take advantage of its current environment. For example ***Chokepoints*** are tight passing areas that allow players to form defensive strategies by blocking them and creating strong anti-ground structures to defend the entrance in the base. Therefore a good building placement would be as near as possible to the chokepoint.

A player must also consider the width and height of a building. Some building also require special placement (Example: Refineries require geysers in order to perform the build). Each building has defined its own length measured in tiles. A tile is a 32 x 32 pixel space. A building cannot be built on occupied tiles or on tiles that have units walking or taking any kind of action on them. One must always consider to check the tiles availability whenever planning the next build.



*The following pictures describes the Terran and Protos building structures (width and height)*

## Reactive control

Reactive control problems involve creating unit actions that involve achieving unit specific goals (Example: *Scouting the enemy base or engaging an invasion over a group of enemy units*). We can list two types of unit controls:

* Unit Micro
* Multi-agent path finding

## Unit micro

Unit micro refers to the unit’s action management mainly in combat performed by the player on a frame. World champion at StarCraft (Jaedong) once said that he can perform 400+ actions/minute and that unit micro lead him in becoming world champion at Broodwar.

There are numerous successful strategies that apply unit micro. For example a Liquipedia article [6] explains the advantages of mutalisk micro on harassing the enemy units. It prevents the targeting AI to pick different mutalisk every time the units move out and come back in the range of the defending towers.

## Multi-agent pathfinding

Pathﬁnding and terrain analysis are an important aspect of most video games, and is especially so in RTS games where a player may control a high number of units at once. RTS pathﬁnding typically consists of guiding multiple units on a 2-D map, with units having various properties such as size, speed, and acceleration. In most games, pathﬁnding focuses on shortest-path optimization, whereas RTS games may involve more complex optimizations involving unit damage, keeping units in formations, or avoiding enemy vision.

The default path finding algorithm implemented in StarCraft is based on the popular A\* path finding algorithm and is executed every time a unit or a group of units receive a moving/attack/construct action.

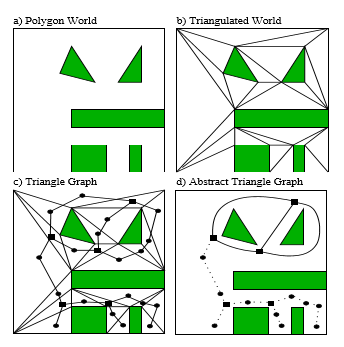
Due to the large computational needs of A\*, it has been modiﬁed in many ways in order to produce pathﬁnding systems for real-time games. Hagelback [7] combined potential ﬁelds with A\* pathﬁnding as a means of unit navigation in StarCraft, concluding that it was preferable to naive A\* for StarCraft navigation.



Later there have been other researches regarding pathfinding and a new method called **triangulation method** has been discovered that is much faster and reliable than Hagelback method dividing the map in triangles and iterating.

Given an environment represented with a polygonal description — one that has barriers between traversable terrain and obstacles described as line segments (see on the figure bellow) — we wish to create a triangulation. This is done by inserting edges between these line segments endpoints until all spaces are divided into triangles [5].

In its simplest form, pathﬁnding in a constrained triangulation is done by hopping from triangle to triangle



*How to reduce triangulation. Graph is constructed from polygonal world description*

In its simplest form, pathﬁnding in a constrained triangulation is done by hopping from triangle to triangle. This presents a challenge because we do not know the path the object will take through each triangle.

# KBot

KBot is a StarCraft AI agent created with the purpose of analyzing better methods to optimize agents by improving cyclical computations and asynchronous frame management. Also the agent uses data collection and action ordering systems to perform several tasks and receive info from the environmental output. With the gathered data, a processing is made in order to cache de important outputs and use them in the upcoming frames.

## Application structure

In order to create accessibility in the whole project, the bot had to be arranged on different abstraction levels. Each level is handled on a separate frame and provides full access to all other related levels.

There are 3 abstraction layers:

* + Unit layer abstraction
  + Action layer abstraction
  + Coordinator layer abstraction

## 4.1.1 Unit layer abstraction

Unit layer abstraction provides control over certain units and provides the movement, training, attack, and gathering actions. Unit layer abstractions are the lowest level due to their direct actions over the controlled units. There are multiple units that can be controlled in game. Kbot takes advantage of “Terran Marine” units and “Terran SCV”. Each unit has multiple properties that define their frame state and are used as input data for the communication layers.

A unit is defined as any game entity that can be targeted. Therefore buildings are also considered to be units.

An example of Unit layer abstraction is provided by the “**Terran supply depot**”. In the following code section I will shortly describe the behavior of a Unit controller:

// Each class is created as a singleton. They implement BaseClass in

// order to get full access of the game input data. Unit layers run once

// at half of a second and update on each frame in order to keep // consistency of the data we are using.

public class TerranSupplyDepot **extends** BaseClass **implements** IBuilding **{**

private UnitType \_buildingType **=** UnitType**.**Terran\_Supply\_Depot**;**

private static TerranSupplyDepot \_instance**;**

//Unit Controllers keep an overall track of when a unit is beeing

//added

private int count **=** 0**;**

//A primary method for the building units is the should build method

//The method provides a boolean value weather the deapot is required

//or not

@Override

public boolean shouldBuild**()**

//Can build method caches the building and sets an awaiting

//for a builder worker to pick it up

@Override

public boolean canBuild()

//checks if any building action has already been queued

public boolean isAlreadyQueued()

//provides a check for currently building units of this type

public boolean isBuilding()

//force build for this type of units ( usefull when requiring //imidiate usage of an unit

@Override

public void forceBuild**()**

//Provides the number of units of this type available for the player

@Override

public int getNumberOfThisType**()**

//Get buildings that are not completed

@Override

public List**<**Unit**>** getBuildingsOfThisTypeNotCompleted**()**

//Updates the current number of buildings of this type

@Override

public void updateCount**()**

//Gets the objects of this type

@Override

public List**<**Unit**>** getBuildingsOfThisType**()**

//provides the count of my units of this type

public int getCount**()**

//sets the count for the counter

public void setCount**(**int count**)**

**}**

## 4.1.2 Action layer abstraction

Action layer abstraction is a formal action handler that decides how the actions should be executed. After processing the input data, a corresponding action is generated .An action describes a behavior of a given unit. For example in order to give a build order for a worker, the abstract unit layer must cache a request for executing the build. After the caching has been completed, the application will wait for the SCV to free up and give the building command.

Actions can be organized on two categories:

* Single execution actions
* Chained executed actions



**Single executed actions** execute a single command after passing pre and post conditions. For example in order to build a new supply depot, we should consider the possibility of own or enemy units already occupying the required tiles, or the resources do not match the required price for the building. Therefore the action is not executed until all conditions pass. After passing conditions have met, the action queue will remove all terminated actions allowing units to perform other actions. An example of simple executed actions is the build action:

public class BuildAction **extends** Action**{**

private UnitType buildingTypeNeeded**;**

private TilePosition position**;**

private Boolean isConstructionFinished**;**

private boolean actionStarted**;**

private boolean isSpecialBuilding**;**

public BuildAction**(**Unit unit**,** UnitType buildingTypeNeeded**,** TilePosition position **,** Action nextAction**,** boolean isSpecialB**)**

// If construction has not started, we check for pre and post

// conditions in order to create a reliable real time system of data

// checking .

// After the building has been built, mark action as executed

@Override

public void executeActions**()** **{**

**if(**isConstructionFinished **!=** **null** **&&** isConstructionFinished **==** **false)** **{**

**if(!**unit**.**isConstructing**()** **&&** **!**unit**.**isMoving**())** **{**

isConstructionFinished **=** **true;**

**this.**setActionExecuted**(true);**

**}**

**}**

**else** **{**

**if** **(**isPreconditionPassed**()** **&&** isConditionPassed**())** **{**

**if** **(!**unit**.**isConstructing**()** **&&** **!**actionStarted**)** **{**

**if(this.**unit**.**build**(**buildingTypeNeeded**,** position**))**

**this.**actionStarted **=** **true;**

**else**

Painter**.**getInstance**().**paintPosition**(**position**);**

**}else** **{**

isConstructionFinished **=** **false;**

**}**

**}**

**}**

**}**

@Override

public boolean isConditionPassed**()**

**}**

public UnitType getUnitType**()**

**}**

**Chained executed actions** perform multiple commands after passing pre and post conditions. The action chain is built as a linked list where each action is linked to the previously required action. I.E if we need a build action after a movement action then the chain will be built as:

***New*** *BuildAction* ***(****parameters,* ***new*** *MovemenActiont****(****x, y, null));*

Notice the null value at the end of the chain. Each chain is terminated by a null value, and the validity of pre and post condition are checked only for non-executed actions.

Let Y be a dependent state of an action X. If Y has been executed, then checking is no longer required. This assumption can be proven using Boolean algebra. If X is assumed to have a valid state (preconditions && post conditions = true) that means that the implication holds only if the resulting state for Y is true.

Therefore left hand side (((true -> X) -> Y) -> Z) remains true after evaluation for any value.

## 4.1.3 Coordinator layer abstraction

Coordinator layers provide component management for different tasks that need to be accomplished. They include the highest level of data manipulation that the agent uses. Each coordinator uses the **runCoordinator** method that executes the managing logic for a specific category. They can be accessed from anywhere from the project and can also act as services for other coordinators. Kbot includes the following coordinators:

* Builder coordinator
* Workers coordinator
* Scout coordinator
* Supply coordinator
* Attack coordinator
* Strategy coordinator

**Builder coordinator** provides building coordination technique for managing build queues. If the queue contains a registered unit, the coordinator pick a random builder from the list of builders and searches for a valid tile that can be used. Tile searching algorithms use circular fashion tile processing and create maps and commands for units to proceed with the building phase.

An important notice is that not all buildings require a free tile to build on. These buildings are categorized as special buildings. Refineries for example require geysers to be allow to be built.

public void runCoordinator**(){**

//Get available building from queue

Building nextBuilding **=** BuildOrder

**.**getInstance**()**

**.**peekNextBuilding**();**

**if(**BuildOrder**.**getInstance**().**canNextBuildingBeBuilt**())** **{**

// gat available builder base

Unit scv **=** WorkerCoordinator

**.**getInstance**().**getAvailableBuilder**(**0**);**

// peek next building

Building building **=** nextBuilding**;**

**if(**scv **!=** **null)** **{**

**if** **(**building **!=** **null** **&&** **!**building**.**isBuilding**())** **{**

// check if special builing then create action

// for searching special building tiles

**if** **(**building**.**isSpecialBuilding**())** **{**

TilePosition tile **=** BuildLogicsCoordinator

**.**getInstance**()**

**.**determineSpecialBuildPosition**(**building**.**getUnitType**());**

// if tile has been found

// then enqueue next building action

**if** **(**tile **!=** **null)** **{**

BuildOrder**.**getInstance**().**getNextBuilding**();** ActionQueue**.**getInstance**().**enqueueAction**(**

**new** BuildAction**(**scv**,** building**.**getUnitType**(),** tile**,** **null,** **true));**

**}**

**}**

**else** **{**

// Clean painting tile for profiling

// purposes only.

Painter**.**getInstance**().**clearTiles**();**

// determine a legit bulding position

TilePosition tile **=** BuildLogicsCoordinator

**.**getInstance**()**

**.**determineBuildPosition**(**building**.**getUnitType**());**

// if tile has been found then pop actual

// building from stack

**if** **(**tile **!=** **null)** **{**

BuildOrder

**.**getInstance**()**

**.**getNextBuilding**();**

// then enqueue next building action

ActionQueue

**.**getInstance**()**

**.**enqueueAction**(new** BuildAction**(**scv**,** building**.**getUnitType**(),** tile**,** **null,** **false));**

**}**

**}**

**}**

**}**

**}**

**}**

**Workers coordinator** provides coordination over SCV units. SCV units can have multiple roles in the game. They can build, gather minerals, gather gas, and scout on enemy units. The main role of this coordinator is to make sure that on each check frame, all SCV have a specific job and there is a continuous flow of work. When creating a new unit, the worker coordinator checks if the unit is idle or not and assigns a role to it based on numerical and statistical check for ensuring a correct management for the players resources.

Workers cannot build if the amount of resources does not meet the required building or unit price. Therefore an average of 10 miner workers and 3 gas extractor workers are required in order to maintain the economy at stabile level.

public void runWorkers**(){**

// get all available command centers and check that each SCV unit on // each command center has a role and perfome an update in case they

// are not doing their required work

ArrayList**<**Unit**>** commandCenters **=** ListUtils**.**getMyCommandCenters**();**

// if we have command centers than we should manage our units

// perform an update check

**if(**commandCenters**.**size**()** **>** 0**)** **{**

**for** **(**int i **=** 0**;** i **<** commandCenters**.**size**();** i**++)** **{**

updateWorkers**(**i**,** commandCenters**);**

Unit closestMineral **=** **null;**

Unit closestGasExtractor **=** **null;**

// get specific command centers tile position

// and get nearest workers

TilePosition tile **=** commandCenters

**.**get**(**i**)**

**.**getTilePosition**();**

List**<**Unit**>** workers **=** ListUtils

**.**getNearestUnitsTo**(**tile**,** UnitType**.**Terran\_SCV**,** Requirements**.**SEARCH\_RANGE\_WORKERS**);**

// iterate workers and check if workers belong to any

// of the available categories ( miner, extractor ,

// builder, or scout) and if not assign a role to him

**for** **(**Unit worker **:** workers**)** **{**

**if** **(!**isWorkerInAnyCategory**(**i**,**worker**))** **{**

**if** **(this.**areMinerWorkersRequired**(**i**))** **{**

**if** **(**closestMineral **==** **null)** **{**

closestMineral **=** ListUtils

**.**getNearestNeutralUnitsTo**(**tile**,** UnitType**.**Resource\_Mineral\_Field**,** Requirements**.**SEARCH\_RANGE\_WORKERS**)**

**.**get**(**0**);**

**}**

sendToGatherMinerals**(**worker**,** closestMineral**,** i**);**

**}** **else**

**if** **(this.**areGasWorkersRequired**(**i**))** **{**

**if** **(**closestGasExtractor **==** **null)** **{**

closestGasExtractor **=** ListUtils

**.**getClosestUnit**(**\_self**.**getUnits**(),** commandCenters**.**get**(**i**),** UnitType**.**Terran\_Refinery**);**

**}**

sendToGatherGas**(**worker**,** closestGasExtractor**,** i**);**

**}**

**}**

**}**

**}**

**}**

**}**

**Scout coordinator** provides control over a group of scouting units (mainly SCV units) and creates control strategy for exploring where the enemy base is located, being one of the most crucial coordinators in the game. The scouts gather much more than location info. They also gather unit count info, how powerful the enemy units are and what types of strategies should the agent apply in order to resist and counter attack the enemy forces.



public void runCoordinator**(){**

// Prepare action queue

ActionQueue queue **=** ActionQueue**.**getInstance**();**

// Prepare scout for action

scout **=** worckerCoordI**.**getScout**();**

// if there are available explored locations

**if(**availableLocations**.**size**()** **>** 0 **&&** availableLocations**.**get**(**0**).**y **==** **true)**

// remove from the exploring list

availableLocations**.**remove**(**0**);**

**else**

// check if the scout isn’t already checking other places

// send the scout to gather informations about the following location

**if(**scout **!=** **null** **&&** scout**.**exists**()** **&&** **!**queue**.**isActionQueued**(**Signatures**.**SCOUTING\_SIGNATURE**)){**

**if(**availableLocations**.**size**()** **>** 0 **){**

// get first available location and enqueue

// a scouting action

Tuple**<**BaseLocation**,**Boolean**>** first **=** availableLocations**.**get**(**0**);**

queue**.**enqueueAction**(new** ScoutingAction**(**scout**,** first**));**

**}**

**}**

**}**

**Supply coordinator** ensures that there are enough units to perform several tasks related to the economy of the player. On each frame check, the coordinator performs a query if there are any command centers that require working units. If the evaluation’s result ends with a requirement, then the supply coordinator sends a training request to the command center in need of working units.

Also, computations are done regarding the current state of the economy. A coordinator cannot perform requests when the economy reaches to a dangerous state. If so, an await will happen until the economy reaches to a stable level again.

public void runCoordinator**(){**

// prepare command centers for check

List**<**Unit**>** supplyBuilders **=** ListUtils**.**getMyCommandCenters**();**

// prepare worker coordinator instance for query

WorkerCoordinator wCoordInstance **=** WorkerCoordinator

**.**getInstance**();**

// prepare resource coordinator for query

ResourceCoordinator resCInstance **=** ResourceCoordinator

**.**getInstance**();**

// iterate all the command centers and check if miners are

// required or gas workers are required or builder workers are

// required.

**for** **(**int i **=** 0**;** i **<** supplyBuilders**.**size**();** i**++)** **{**

**if(**wCoordInstance**.**areMinerWorkersRequired**(**i**)**

**||** wCoordInstance**.**areGasWorkersRequired**(**i**)**

**||** wCoordInstance**.**areBuildersRequired**(**i**)){**

// check if resources available permit picking a builder

// worker from the current list of scv units

**if(**resCInstance**.**getMyResources**().**getMinerals**()** **>** **(**100**\*** **(**wCoordInstance**.**getBuilders**(**i**).**size**()** **+** 1**))**

**&&** wCoordInstance**.**getBuilders**(**i**).**size**()** **<** Requirements**.**MAX\_NR\_BASE\_BUILDERS**)** **{**

populateBuilders**(**supplyBuilders**,**i**);**

**}**

// virtual supplies represent the number of supplies

// currently queued + nr of phisical supplies owned

// if no training is perfomed, train a new scv unit

**if(**getVirtualSupplyUsed**()** **<** getTotalSupply**()** **-** Requirements**.**ALERT\_SUPPLY\_ZONE **&&** **!**supplyBuilders**.**get**(**i**).**isTraining**())** **{**

supplyBuilders**.**get**(**i**).**train**(**UnitType**.**Terran\_SCV**);**

**}**

**}**

**}**

**}**

**Attack coordinator** is responsible for creating an attack army that is capable of defeating the enemy force using the gathered data from the scouts and the past events. The main attack formation is composed of a large number of Terran Marines that burst into attack once the odds are in the favor the agent. On each frame check, the coordinator creates a query on the resource coordinator and supply coordinator that validates if there are enough resources and supplies to queue a new training. If the supply zone is under the alert supply zone than the training will be disabled until a new supply depot is constructed. If the performed check evaluates as required, than the coordinator iterates through all the attack unit providers (in this case the Terran Barracks), and starts training units.

Attack units can also have the role of defending the workers and the base from any harassment.

public void runCoordinator**(){**

// update army to take action on any possible threat

updateArmy**();**

// if barracks exist then start training units

**if(**TerranBarracks**.**getInstance**().**getCount**()** **>** 0**){**

// prepare barracks for iteration

List**<**Unit**>** availableBarracks **=** TerranBarracks

**.**getInstance**()**

**.**getBarracks**();**

// get available barracks and check if units can

// be trained at the current baracks

// also make checks if current economy permits

// marines and barracks have an idle state

**for** **(**Unit brk**:** availableBarracks**)** **{**

**if(**brk**.**canTrain**(**UnitType**.**Terran\_Marine**)** **&&**

canAffordUnit**(**UnitType**.**Terran\_Marine**)** **&&**

**!**brk**.**isTraining**() &&**

// supply zone should not exceed critical supply zone

SupplyCoordinator**.**getInstance**().**getUsableSupply**()** **>**

Requirements**.**ALERT\_SUPPLY\_ZONE**){**

//start the actual marine training

brk**.**train**(**UnitType**.**Terran\_Marine**);**

**}**

**}**

**}**

**}**

**Attack strategy coordinator** is responsible for managing the current army by means of movement, situational tactics, and approach strategies. Currently the agent’s stance is defensive and built on a safe winning algorithm by swarming with a high number of units at once, leaving the enemy player without options.

The agent can be reprogrammed to adapt an aggressive strategy by launching a high number of attacks with small groups of units reducing the economy growth of the enemy player, hence leaving no other possibility than to surrender.

Target selection is an important part to handle because one has to decide what type of units should be targeted first in order to keep alive as many units as possible. In every group there are high damage with slow attacking rate units and low damage with fast attacking rate units. So depending on your army size a targeting strategy needs to be settled before the actual attack begins.

public void runCoordinator**(){**

// gather army an prepare to launch an attack

List**<**Unit**>** myArmy **=** AttackCoordinator

**.**getInstance**()**

**.**getArmy**();**

// check if attack has been lauched than prevent units from executing

// other actions unless the battle odds require so

attackLaunched **=** **!(**myArmy**.**size**()** **>=** Requirements**.**MIN\_ARMY\_ATTACK\_SIZE

**&&** **(**enemyTarget **==** **null** **||** **(**enemyTarget **!=** **null** **&&** **!**enemyTarget**.**exists**())** **||** myArmy**.**get**(**0**).**isIdle**()));**

// check attack condition to begin when the attack is not launched

// and the minimum attack size is passed the required team size

**if(!**attackLaunched **&&** **(** myArmy**.**size**()** **>=**

Requirements**.**MIN\_ARMY\_ATTACK\_SIZE**)){**

// get the closest unit that represents the highest threat for the

// team

enemyTarget **=** **!**attackLaunched **?**

ListUtils**.**getClosestNonBuildingUnits**(** myArmy**.**get**(**0**)** **,**\_game**.**enemy**().**getUnits**()):** enemyTarget**;**

An easy unit selection strategy is the direct harassment described below:

// if the enemy target is found and it exists and

// is first attack action on the following target then

// send each unit to attack the specified enemy target

**if(**enemyTarget **!=** **null** **&&** enemyTarget**.**exists**()){**

attackLaunched **=** **true;**

// iterate through all units and proceed with the

// attack action

**for** **(**Unit u**:** myArmy**)** **{**

u**.**attack**(**enemyTarget**);**

**}**

**}**

**else{**

// if enemy target is dead or turned invisible then

// select another targetable unit that presents a high

// threat for the team

enemyTarget **=** ListUtils**.**

getClosestNonBuildingUnits**(**myArmy**.**get**(**0**),**\_game**.**enemy**()**

**.**getUnits**());**

**}**

**}**

**else** **if(**attackLaunched **&&** **(** myArmy**.**size**()** **>=** 0 **)){**

// if attack is successfully accomplished and target has

// been eliminated than mark attack flag to false

**if(**enemyTarget **!=** **null** **&&** **!**enemyTarget**.**exists**())**

attackLaunched **=** **false;**

**}**

**}**

## 4.2 Frame management

During the execution, a high amount of computation needs to be processed on each frame, creating a performance issue in communication between the game and the agent, sometimes leading to execution break or the death of the process. In order to prevent this events from happening, a frame based division approach is required.

Each frame has assigned a module to execute in order to keep computations as low as possible. Starcraft runs at 30 frames / second, hence providing possibility of managing 30 modules on each second in order to increase the overall performance and to keep reliability in means of information analysis.



Dividing as much as possible on sub problems and grouping them in modules that execute on separate frames creates a difficulty in managing data in real-time.

In order to prevent a data from being managed incorrectly or losing important parts from it, a mechanism must be implemented that will allow updating modules with the intended data or locking other data packages that must only be manage on a specific frame without any updates.

That is why I have implemented a flag based system that allows the application to internally cache and manage its own flags and data, based on passing between modules. In this way, if a module’s data is being used in 3 other modules that require an information regarding a specific state than the module that requires the latest information will subscribe for an update. Finally the data is updated and cached only when it is necessary leaving other dependent modules to use information from the parent module’s cache.

An example of data handling with flags would be the following: Assume we have 3 modules named X, Y, and Z. Module X uses a method provided by module Y and a method provided by module Z, but in order to provide a value they both perform computations on a set of values (named *alpha*) queried from the game’s core. Module Y’s method executes first so makes the first query and caches the set, notifying modules X and Y, that it has already requested the values for *alpha* so they no longer have to query for the same set, but instead it can directly provide them the latest values. This way we can avoid unnecessary computation between modules.

If current state of data is not so important, updates can be performed once at 15 frames. Data will not be accurate but it will avoid multiple querying on the currently processed frame.

## 4.3 Performance optimization and method profiling

In order to optimize the overall performance for the agent, a continuous time measurement was required for each module. Any coordinator’s run method contains two timestamps. By subtracting the first timestamp from the second one, we get the exact time of execution for the selected method.

On the right of the screen, a list of bars are displayed that represent the actual performance of the application. The bars can have the colors: red, white or the combination of red and white. If the bar is completely red, it means that, the method is not slow compared to the other modules functionalities so, the method could be optimized.

If the bar is white, it means that the method belongs local optima and has a linear evolution with respect to the rest of the application.

Looking at the picture below, we can notice that the worker coordinator, the supply coordinator and the building analyzer are being the most active module performing the highest number of computation due to the frame checks performed at the beginning of the game.



An overview on the created profiler:

// general puprose class used to measure the code’s

// overall perfomance. This class is used only for debugging

// purposes and code code profiling and testing.

public class CodeProfiler **{**

private static HashMap**<**String**,** Long**>** aspectsStart **=** **new** HashMap**<>();**

private static HashMap**<**String**,** Double**>** aspectsLength **=** **new** HashMap**<>();**

// method used to start measuring a module by caching

// a start time before the method is executed

public static void startMeasuring**(**String title**)** **{**

measureAspect**(**title**);**

**}**

// method used when starting a new measure

private static void measureAspect**(**String title**)** **{**

aspectsStart**.**put**(**title**,** now**());**

**}**

// returns the time in nanoseconds

private static long now**()** **{**

**return** System**.**nanoTime**();**

**}**

// method to finish measuring a module caching the length between

// the start of the measuring and the current moment

public static void endMeasuring**(**String title**)** **{**

long measured **=** now**()** **-** aspectsStart**.**get**(**title**);**

**if** **(!**aspectsLength**.**containsKey**(**title**))** **{**

aspectsLength**.**put**(**title**,** **(**double**)** measured**);**

**}** **else** **{**

aspectsLength**.**put**(**title**,** aspectsLength**.**get**(**title**)** **\*** 0.6 **+** measured **\*** 0.4**);**

**}**

**}**

// returns the total time consumption for a specific module

// this method is mainly used by the painter class

public static HashMap**<**String**,** Double**>** getAspectsTimeConsumption**()** **{**

**return** aspectsLength**;**

**}**

**}**

## 4.4 Map and construction analysis

In order to get a valid building position, the agent must perform an analysis over the current environment and determine if there are any collisions with other units or buildings, and checks if the environment permits construction on the requested tiles.

The agent also has to take in consideration that the building must be as close as possible to each other in such a way that it will not block the path for own units and will create difficulty for enemy units to perform an attack.

To manage all of these conditions, I selected a matrix (group) composed of X\*Y tiles. Each group is duplicated on the X axis and Y axis to perform a circular check .For each group of matrix the builder checks if he can perform a building on the specified tile. If not, the algorithm continues until it reaches outside of its searching scope.

Building algorithms are performed by the Building logics coordinator.



# 5. Bibliography

[1] Anna Bogdanowicz, Advances in AI making games smarter. http://theinstitute.ieee.org/technology-focus/technology-topic/advances-in-artificial-intelligence-make-games-smarter571

[2] David Churchill, StarCraft AI competitions (AIIDE). https://webdocs.cs.ualberta.ca/~cdavid/starcraftaicomp/history.shtml

[3] Adam Heinermann, Brood War API

https://github.com/bwapi/bwapi

[4] Blizzard Entertainment general info, Overlord overview

http://starcraft.wikia.com/wiki/Overlord

[5] Douglas Demyen and Michael Buro. Eﬃcient triangulation-based pathﬁnding.

https://skatgame.net/mburo/ps/tra.pdf

[6] JulyZerg , Mutalisk Harassment: http://wiki.teamliquid.net/starcraft/Mutalisk\_Harassment#Grouping

[7] Johan Hagelback. Potential-ﬁeld based navigation in StarCraft. In CIG (IEEE), 2012.

https://www.researchgate.net/publication/261280564\_Potential-field\_based\_navigation\_in\_StarCraft

[8] Santiago Ontano´n, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game AI research and competition in StarCraft. TCIAIG, 2013. Available at: https://hal.inria.fr/file/index/docid/871001/filename/survey.pdf

[9] Zoubin Ghahramani, Gatsby Computational Neuroscience Unit, University College London, An introduction to Hidden Markov Models and Bayesian Networks

http://mlg.eng.cam.ac.uk/zoubin/papers/ijprai.pdf