

BLEKINGE TEKNISKA HÖGSKOLA

# Adaptive Goal Oriented Action Planning for RTS Games

by

Matteus Magnusson   Tobias Hall

A thesis submitted in partial fulfillment for the  
degree of Bachelor

in the  
Department of Computer Science and Communication

June 2010

# Declaration of Authorship

Matteus Magnusson and Tobias Hall declare that this thesis titled, “Adaptive Goal Oriented Action Planning for RTS Games” and the work presented in it are our own. We confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this College.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this College or any other institution, this has been clearly stated.
- Where we have consulted the published work of others, this is always clearly attributed.
- Where we quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely our own work.
- We have acknowledged all main sources of help.
- Where the thesis is based on work done by ourselves jointly with others, we have made clear exactly what was done by others and what we have contributed ourselves.

*“You must construct additional pylons.”*

—StarCraft®<sup>1</sup>

---

<sup>1</sup>StarCraft is a registered trademark of Blizzard Entertainment, Inc, in the U.S. and other countries.

BLEKINGE TEKNISKA HÖGSKOLA

# *Abstract*

Department of Computer Science and Communication

Bachelor thesis

by [Matteus Magnusson](#) [Tobias Hall](#)

This thesis describes the architecture of an adaptive goal-oriented AI system that can be used for Real-Time Strategy games. The system is at the end tested against a single opponent on three different maps with different sizes to test the ability of the AI opposed to the 'standard' Finite State Machines and the likes in Real-Time Strategy games.

The system consists of a task handler agent that manages all the active and halted tasks. A task is either low-level; used for ordering units, or high-level that can form advanced strategies. The General forms plans that are most advantageous at the moment. For creating effective units against the opponent a priority system is used; where the unit priorities are calculated dynamically.

# *Acknowledgements*

We would like to thank the creators of the Spring engine which has done a good job implementing an engine that supports the modification community. The creators of Evolution RTS who has done a so far great job with the mod, and are always quick to answer our questions.

A special thanks to our primary supervisor during the development of this thesis: Johan Hagelbäck who always helped us with short notice and was a great support with his knowledge in RTS games. Our secondary supervisor and examiner Stefan Johansson that helped us with the more specific thesis layout.

To Steve Rabin who is the editor of the AI Game Programming Wisdom-series and the authors who contributed with their articles to him; these were very useful when designing the system. To all those who read this thesis and contributed with finding errors in the text and sending feedback to us.

Finally, we want to thank our families that has supported us in bad and good times.

# Contents

<b>Declaration of Authorship</b>	<b>i</b>
<b>Abstract</b>	<b>iii</b>
<b>Acknowledgements</b>	<b>iv</b>
<b>List of Figures</b>	<b>vii</b>
<b>List of Tables</b>	<b>viii</b>
<b>Listings</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Brief Description</b>	<b>3</b>
2.1 Architecture of AI Ice . . . . .	3
2.2 The Engine and Mod That AI Ice Runs On . . . . .	3
<b>3 Detailed Description</b>	<b>4</b>
3.1 Mechanics of Evolution RTS . . . . .	4
3.1.1 Armor System . . . . .	4
3.2 Mechanics of AI Ice System . . . . .	5
3.2.1 Spring Wrapper . . . . .	5
3.2.2 Units . . . . .	6
3.2.3 Enemy Information . . . . .	7
3.2.4 Extraction point mapping . . . . .	8
3.2.5 Priority System . . . . .	8
3.2.6 Task System . . . . .	14
<b>4 Result</b>	<b>21</b>
4.1 Bot Tests . . . . .	21
4.1.1 RAI . . . . .	21
<b>5 Discussion</b>	<b>22</b>

<b>6 Conclusion and Future Work</b>	<b>24</b>
6.1 Conclusion . . . . .	24
6.1.1 Project/Architecture conclusion . . . . .	24
6.1.2 Things we would've done differently . . . . .	24
6.2 Future work . . . . .	25
6.2.1 Improvements to the current system . . . . .	25
6.2.2 New Features . . . . .	26
<b>A AI Ice Architecture</b>	<b>28</b>
A.1 Interaction Between Agents . . . . .	28
A.2 Task System . . . . .	28
A.3 Complete Documentation . . . . .	29
<b>Bibliography</b>	<b>30</b>
<b>Index</b>	<b>31</b>

# List of Figures

3.1	Evolution RTS damage multiplier chart . . . . .	5
3.2	Path of the roaming scout . . . . .	19
A.1	Collaboration diagram between agents in AI Ice. . . . .	28
A.2	Task system class diagram. . . . .	29



# List of Tables

3.1	Energy increment per minute over time . . . . .	13
3.2	Energy income over time . . . . .	13
4.1	Al Ice vs. RAI . . . . .	21

# Listings

3.1	Damage Type Priority . . . . .	9
3.2	Damage Type Priority Scaling . . . . .	10
3.3	Armor Type Priority . . . . .	11
3.4	Energy Increment Over Time . . . . .	12

# Introduction

Today there exists a large amount of games; and many of the larger games has some AI implemented in it. There exists many different kinds of AI; first person shooter, sports, board games, etc. If you're reading this you have probably played a game where you thought that the AI knew what you did even though it could not possibly see you—maybe it went straight to the house you resided in, or had some almighty power—more resources than it could possibly get under that time, even if it would play perfect in every sense. Some of these can easily be spotted whereas some are only spotted by the most skilled players that have played the game a lot. The current now flows towards minimizing the cheating for AI in games; however this often complicates problems—solutions a lot, making the game more CPU hungry.

Good for us then that more and more of the CPU is used by the AI system. Physics will probably move to the graphics card in an early future—although physic support on the graphics card already exists, far from all games has support for it and if a game has the support the user still needs an extra graphics card. When the time is right the AI will hopefully get a decent amount of CPU to do some fancy work and act all natural without cheating, in fact it's hard to act as a human if it can cheat.

Real-Time Strategy games are however still behind in this area, most AIs cheat and are statically written through state-machines and the likes. A newer approach is to base the AI on a goal oriented architecture. The AI will then select what goals are most beneficial and execute them. Most notably it gets a less static behaviour if implemented correctly. To be able to turn off the cheating a good resource handling is needed; which could be implemented through a goal meaning it would handle itself once implemented correctly. Some sort of scouting also needs to be done and then keep the information about the enemies' units, preferable where they are located. The scouting can be implemented using goals again, and the enemy information could be used with some sort of terrain analysis.

In our AI, Al Ice, we're going to implement a simple goal-oriented system<sup>1</sup> with an adaptive behaviour—meaning it will create tasks that counter the enemies' behaviour. E.g. creating units that are good against the enemies' armada. The intention is to show that an adaptive goal oriented architecture can compete against the more traditional architecture (finite state machine system). The method of proving or disproving

---

<sup>1</sup>We call it a task system; task and goal generally being the same thing.

this hypothesis is to let the different systems play against each other and monitor the outcome.

## Brief Description

### 2.1 Architecture of Al Ice

The architecture consists of the *General* that is the mastermind in Al Ice. He forms plans; practically the same as creating various tasks<sup>1</sup> that are supposed to lead it into a victorious defeat of the opponent. To do this it fetches information that has been collected about the enemy during game play to later decide, create, and run tasks with highest priority. Al Ice will always try to construct units that counter the opponents weapon and armor types.

Except the general and tasks there is *a)* an agent for handling the enemy information that is updated when a unit enters line-of-sight. The agent also updates the enemy information about each second, such as the current position of the unit; *b)* a task unit handler which handles all our task units—a task unit can simply be described as a wrapper between a unit from the spring engine and a set of tasks bound to the unit; *c)* extraction point mapping where information about the extraction points are held—if a point belongs to us, the enemy or is free; and *d)* the task handler that takes care of all the active and halted tasks. It also binds and unbinds the connection with task units and splits the load over several frames making it more CPU friendly.

### 2.2 The Engine and Mod That Al Ice Runs On

Al Ice is implemented for the Spring engine [1] with the modification *Evolution RTS* [2]. Spring is only an engine and does not come with a game; therefore a separate 'game', also called mod, needs to be downloaded. There are several games available and one of these is Evolution RTS that Al Ice is especially built for. An important side note is that Al Ice does not cheat in any sense, no extra resource, disabled fog of war<sup>2</sup>, or any other kind of unfair advantages.

---

<sup>1</sup>A task always wants to fulfill its goal—whether it is to move to a certain location with a unit or attack with a group of units.

<sup>2</sup>Fog of war being a 'fog' that lies over the battle making it impossible to see enemy units beyond our units' sight distance; thus disabled fog of war makes a player or AI able to see everything that happens on the map

## Detailed Description

### 3.1 Mechanics of Evolution RTS

Evolution RTS [2] is an open source game modification created for the Spring engine. It's a Real-Time Strategy game featuring both micro and macro-management with intense battles, real time meaning it's not turn based and that all the players can issue orders at any time.

The resource gathering system is automated; i.e. you construct resource gathering buildings that collect the resources automatically, giving you a constant stream of income. There are two different kinds of resources; energy and metal. Energy can be produced with three different buildings: solar collector, fusion reactor and geothermal power plant. Metal can only be produced with two buildings: metal extractor and metal maker. The metal extractor building has to be built on top of special sites on the map, these sites are called "extraction points" and are very important for map control and metal income, since they produce much more metal than the metal maker building. Metal is used to construct additional units whereas energy is needed when units fire their weapons—some buildings also consume energy that is needed for their use; shield generator is one kind of building that always consumes energy.

To construct units the player needs to have factories. There are several different kinds of units in Evolution RTS: Flying, amphibious, all-terrain<sup>1</sup>, and ground units. As a side note, Al Ice does not use any amphibious units.

#### 3.1.1 Armor System

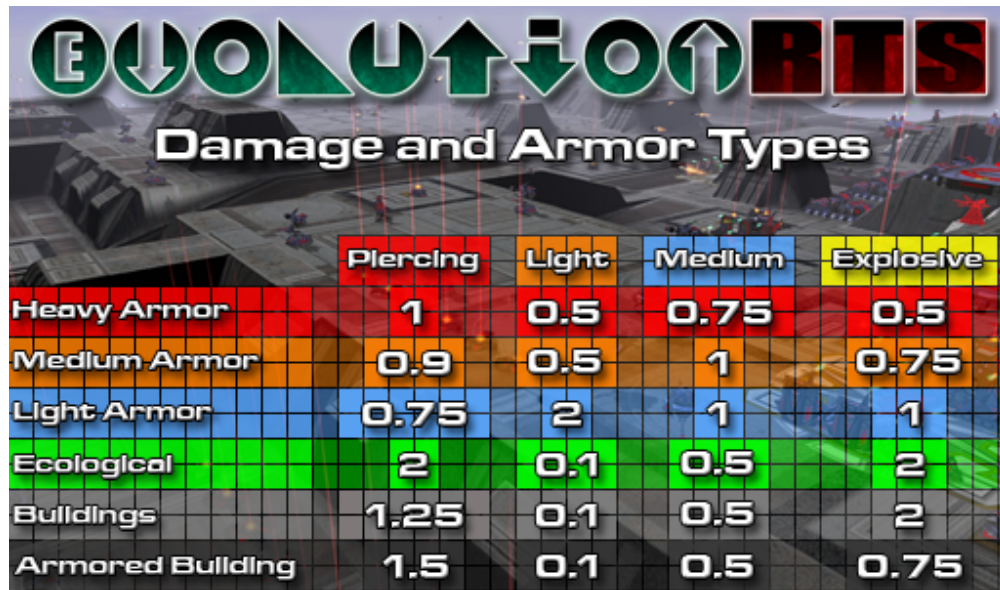
Evolution RTS uses an armor system [3] that simplifies the damage calculation between weapons and unit armor; both for balancing units, although this is not our area, and generating a priority for what units to create which Al Ice does; see 3.2.5 for how the priority system works.

The damage a unit will do against another unit is calculated by adding its weapon damage times the damage multiplier. The damage multiplier is calculated by the armor type the unit who gets hit has and the damage type of weapon that fired against the hit

---

<sup>1</sup>Can travel on any terrain, even over cliff edges.

FIGURE 3.1: Evolution RTS damage multiplier chart



The image shows a damage multiplier chart for the game Evolution RTS. At the top, there are several icons: a green 'E' in a circle, a green arrow pointing down, a green circle, a green arrow pointing up, a green arrow pointing down, a green circle, a green arrow pointing up, and the letters 'RTS' in a red box. Below these icons is the title 'Damage and Armor Types'. The chart is a table with armor types as rows and weapon types as columns. The background of the table is a grid of colored squares: red for Heavy Armor, orange for Medium Armor, blue for Light Armor, green for Ecological, grey for Buildings, and dark grey for Armored Building.

	Piercing	Light	Medium	Explosive
Heavy Armor	1	0.5	0.75	0.5
Medium Armor	0.9	0.5	1	0.75
Light Armor	0.75	2	1	1
Ecological	2	0.1	0.5	2
Buildings	1.25	0.1	0.5	2
Armored Building	1.5	0.1	0.5	0.75

unit. Each unit has a specified armor and weapon type; provided the unit has a weapon at all. The damage multiplier chart is shown in figure 3.1

## 3.2 Mechanics of AI Ice System

In the following sections the most important mechanics of AI Ice are described. Spring Interface; how the interaction and events are handled. General; the mastermind of AI Ice. Units; how we handle our own units. Enemy information; how enemy information is gathered, stored, and used. Extraction point mapping; how we handle the extraction points on the map, i.e. which are free, under our control, and under enemy control. The task system; how tasks work, how they are handled, and the connection with units. In addition to the task system the tasks that are central in AI Ice are described; the scouting tasks, and the attack task.

### 3.2.1 Spring Wrapper

Details about the current game can be extracted from this. However it is only possible to get the information that is either available from start or that the AI can see; meaning that enemy information can only be extracted from those units that we can see at the moment. If we want to save some information that needs to be done internally by the AI and is exactly what 3.2.3 Enemy Information and 3.2.4 Extraction point mapping does.

Available features from the start are the resource positions; i.e. where metal extraction points can be located, how large the map is, our start position, the available units, etc.

## Events

Everything of the AI is handled through events, whereas the *update* is the most notable event where a major part of AI Ice gets updated—the event sends the current frame of the game<sup>2</sup>. Apart from the update event some more standard events exist; when a unit or enemy is created, destroyed, finished, and when an enemy enters and leaves line-of-sight. To listen to these events other agents need to implement the appropriate handler, that has approximately the same name as the event. It should then add itself as an observer to the Spring Wrapper and will then listen to all the events it implements.

If an agent wants to listen to a specific unit's or enemy's event it will need to implement specific handlers for unit events or enemy events respectively. To start listening to the events the agents should add itself as an observer for all the units it wants to listen to.

This system makes it very easy to listen to events and only to listen to those that are relevant for the agent.

### 3.2.2 Units

All the units that are available in Evolution RTS are grouped with some useful attributes by AI Ice—these are not the units that are currently in play but units that can be built or summoned in one way or another. It's important to note that a unit in Spring can be both a mobile unit and a building; in fact everything that can be 'marked' is a unit, even trees.

**unitDef** The unit's definition used by the Spring Engine.

**unitName** The name of the unit; not the human-readable name but the 'definition'.

**canAttackAir** If the unit has the ability to attack air; for simpler checking instead of checking its weapons.

**armorType** The armor type that the unit has. Can be one of the armor types shown in figure 3.1

**damageType** The damage type that the unit has; provided that the unit has a weapon at all. Can be one of the damage types shown in figure 3.1

**build** True if the unit should be used in the building priority; i.e. if AI Ice should use it or not.

**groups** The groups that the unit belongs to. The groups are described more thoroughly in 3.2.2 Groups.

The unitName, canAttackAir, build, and groups are set in the source-code while the others values are set during the initialization. The canAttackAir flag would ideally be set in the initialization, but that information was rather hard to extract; therefore the canAttackAir is now statically defined in the code.

---

<sup>2</sup>Spring always simulates a frame rate of 30 frames per second; meaning that the game time can be extracted from this.



## Groups

Units are grouped by their primary use; the groups are static and set in the source code. There are currently nine groups—however, more can be added easily to the units. Most units are only in one of the nine groups except the different builders.

**Attack force** Units belonging to the attack force are mobile units that are able to attack. Both airborne units as ground units belong to this group.

**Armored building** Armored buildings consist of buildings that are able to attack.

**Builder** All units that can build something belongs to this group. This group is primarily used to build new builders for the unit with the highest priority, and as a wrapper for all the different builders; if there is some information that needs to be extracted from them. Builders should also belong to another group; *factory* or *mobile builders*, which is mainly used for the priority generation. See 3.2.5 for more information about the priority system.

**Factory** A group for all the factories in the game; i.e. builders that are buildings. Because a factory is a builder it should also belong to the builder group.

**Mobile builder** Used for all the mobile builders in the game. As with factories, mobile builders should also belong to the builder group.

**Economic** These are the units (buildings) that produces either metal or energy.

**Scout** Units that are dedicated to scouting should be a part of this group. In Evolution RTS there is one airborne unit which has this as it's speciality; wide line-of-sight and because it's airborne its speed is much faster than ground units. There is also a rather fast amphibious scout; however as Al Ice doesn't use amphibious units this scout isn't used.

**Utility building** Consists of all the buildings that gives the player, in this case Al Ice, with more strategic options; like radar tower, radar-jammer, and nuclear missile silo. This group isn't used in v0.3 of Al Ice.

**Healer** Healers are those specially assigned to heal (repair) the damaged units. In Evolution RTS all mobile builders can heal, but only the ORBs are designated healers. This group isn't used in v0.3 of Al Ice.

### 3.2.3 Enemy Information

Al Ice needs to know how many enemy units there are, otherwise the priority calculation will be ineffective. It also needs to know the positions of the enemy units to be able to order attacks. To meet these demands Al Ice has a uses an agent 'Sighted Enemies' who's job is to handle and update all enemy unit information. Since Al Ice does not cheat it would loose information about the enemy unit if it moves out of sight. That's why the last known enemy position needs to be saved. SightedEnemies works a lot like extraction point mapping and updates itself automatically—it also updates the positions of sighted enemies regularly, since units can change positions, which metal extraction

points can't. Various information can be extracted from Sighted Enemies; a list of all enemy units—ordered by armor and damage type if wanted, how much armor and DPS the enemy has grouped by armor and damage type respectively, and the closest enemy to a specified point mainly when ordering attacks.

### 3.2.4 Extraction point mapping

Al Ice needs to save all the information the scouting tasks collect, information about which extraction points are free for the taking and which points are occupied by an enemy. To save this information the system listens to events from the Spring engine, such events as: enemy enters line of sight, enemy destroyed, unit created and unit destroyed. This way Al Ice will automatically update the information about an extraction point as any unit, not only a scouting unit, passes by and saves it to a vector. The *Extraction Point Map* agent has methods for calculating and returning the position of the closest extraction point by a specified owner. This allows the General to expand Al Ice's resource income and to plan future attacks on enemy outpost's.

### 3.2.5 Priority System

Instead of assigning priorities to task, with the exception of a task priority for a task unit 3.2.6, the priorities are assigned to the available units. The system is mostly based on other articles [4, 5] for generating priorities, grouping them, and how to balance them in a more isolated manner.

The units available in the priority system are only those considered at the moment; usually being all that Al Ice are specified to build—which is done statically in the code. See 3.2.2 on how the units and groups are handled. The priority system will only generate<sup>3</sup> new priorities when necessary; i.e. when there is an available builder. It checks for available builders a few times each second, thus not making the system generating new priorities when an available builder which can't build any of the units in the current priority list exists.

The priorities are divided into groups; made for the simplicity of tweaking priorities with the use of specific multipliers, minimum, and/or maximum values for a group. How the specific group priorities are generated is described in more detail under each group's paragraph.

When generating priorities for units, some will have no or a low priority; those under this threshold will never make it into the priority list. This functionality is useful when building a specific unit, or if all units from a group should be prohibited at all costs. It's currently used with the *economics only* period, which is run just after the *initial build sequence* is completed. In this time all other groups are set to 0 in priority—this makes the available builders to only build economic buildings and boost our income in the beginning of the game. If an available builder can't build any units in the generated priority list it is skipped.

In the first stage of the priority generation it generates the basic information about the attack force and armored buildings priority. These are described more thoroughly in their respective paragraph. The second stage iterates through all the units that is

---

<sup>3</sup>generate and calculate has the same meaning and are used interchangeably.

taken into consideration when generating the priorities. Since all units belong to at least one group, it's then up to that group to generate the priority for the unit. As always some units need special treatment—it's the group's responsibility to handle the special algorithm needed for the priority generation of the unit. To handle 'special' units the group simply checks if a unit needs any special treatment using an if-else if-else statement which contains the units definition name. An example of this could be the airborne builder, which is the only builder from the mobile builder group that will get another priority than the minimum builder priority—the number of airborne builders will slowly, but linearly, increase during the game. If the unit doesn't need any special treatment it will default into the else statement and use the default priority generation for the group. If a unit happens to belong to more than one group it will only keep the highest generated priority. After the priority is generated it is either added to the priority list or discarded if the priority is too low, as described earlier. The third and final stage consists of sorting the priority list and then adjusting the builder priorities; described in the builder group paragraph.

### Group Priority Generation

**Attack Force** The main calculation of the attack force priority is done in the first stage of the priority generation. The general idea is that the priorities generated are based on what armor type is good against them; i.e. what armor type the enemy can deal least damage to, depending on their current units, and what damage type is good against them; i.e. what damage type our units should have that will make them destroy all the enemy units in the least time. Only units that can do us harm will be considered into the calculation; i.e. the enemies' attack forces and armored buildings. How the priorities are generated are described in more detail below.

**Damage type priority generation** First of all the priority for all damage types are generated. The pseudo-code describes how the priority is generated and is described in listing 3.1. The `enemyArmorTypeHealth(armorType)` returns the total seen health

LISTING 3.1: Damage Type Priority

```
Map<String , Double> tempDamageHealth;
double maxHealth = Double.MIN_VALUE;
for (all damage types) {
    double health = 0.0;
    for (all armor types) {
        health += enemyArmorTypeHealth(armorType) / damageMultiplier(
            armorType, damageType);
    }
    if (health > maxHealth) {
        maxHealth = health;
    }
    tempDamageHealth.put(damageType, health);
}
```

that the enemies have of the specified armor type. `damageMultiplier(armorType, damageType)` returns exactly what it says; the damage multiplier between the specified

armor and damage type. See 3.1.1 on how the Armor System works. The value you get from this algorithm is how much health the enemy would have had if the DPS<sup>4</sup> isn't changed depending on the armor type. E.g. if the unit has a default health of 1000, and our damage type's effectiveness is 0.5 then it would be the same as making the effectiveness 1.0, keeping the DPS value, and divide the health with the effectiveness increasing the health to 2000. As can be seen this will generate higher health if the enemy damage type is ineffective against the armor type—thus the damage type that has lowest total health is the damage type that should be prioritized.

To keep the priority in reasonable ranges it needs to be scaled, but also inverted since the damage type with lowest health should get the highest priority. The pseudo-code for this is described in listing 3.2.

LISTING 3.2: Damage Type Priority Scaling

```
double scaleFactor = PRIORITY_MAX - PRIORITY_MIN;
for (healths in tempDamageHealth) {
    double scaledPriority = scaleFactor * health / maxHealth;
    // Invert the priority
    scaledPriority = scaleFactor - scaledPriority + PRIORITY_MIN;

    for (all armor types) {
        insertIntoMatrix(armorType, damageType, scaledPriority);
    }
}
```

First the priority, actually the health, gets scaled between 0.0 and scaleFactor. E.g. if PRIORITY\_MAX is 20 and PRIORITY\_MIN is 10 then scaleFactor will be equal to 10. The priority is then inverted; note, to get the highest priority (20) the health needs to be 0 thus the priorities will be linearly scaled. Then the scaled priority is added to the matrix for all armor types. E.g. if you think of the matrix as columns and rows the scaled priority is added column-wise for each damage type. Then the scaled priority from the armor calculation will be multiplied in row-wise.

**Armor type priority generation** It's now time to generate armor type priorities from the enemies' damage types, that is which type of units that will withstand longest if all enemy units would target randomly on our units. The concept is really the same as the damage type priority generation, see listing 3.3. However there are some subtle differences described below.

The for-loops has now exchanged places; i.e. instead of `for (all damage types)` being the outer loop the `for (all armor types)` is now the outer loop. The `dps` is how much total DPS the enemy can do on the current armor type. Thus the armor type with the lowest DPS should be prioritized. As with the damage type priority the value needs to be scaled and inverted. The algorithm is just about the same as the damage type priority scaling and will therefore not be described in further details.

**Iterating through attack force units** In the second stage all units belonging to the attack force group will be iterated through it's method. This method simply fetches

---

<sup>4</sup>Damage per second.

LISTING 3.3: Armor Type Priority

```

Map<String, Double> tempArmorTypeDps;
double maxDps = Double.MIN_VALUE;
// Start with armor types instead of damage types
for (all armor types) {
    double dps = 0.0;
    for (all damage types) {
        dps += enemyDamageTypeDps(damageType) * damageMultiplier(
            armorType, damageType);
    }
    if (maxDps < dps) {
        maxDps = dps;
    }
    tempArmorTypeDps.put(armorType, dps);
}

```

the priority from the matrix, generated in stage one. E.g. If a unit has the armor type light and damage type explosive the method fetches the information from the row light armor and column explosive damage.

A bonus priority is also given to anti-air units depending on the how much total health flying units the enemies have. There are better ways to implement this and one approach is described in [6.2.1 More accurate unit priority](#).

The method also decreases the unit's priority with the amount of units of that type we already have in game. This is used to generate some more variety to the units we create. As with the bonus priority for flying units, there are better approaches that can be implemented, also described in [6.2.1 More accurate unit priority](#).

**Armored Buildings** The generation of armored buildings priority is in general the same as with attack force, but has a much simpler algorithm. The enemies' DPS against our armor type doesn't need to be taken into account since all armored buildings have *armored building armor*. For the calculation of our preferred damage type only the enemies' attack forces are needed. The rest of the priority generation is the same as for the attack force priority generation; decrement of armored units we already have, and a bonus priority for the amount of flying units the enemies have.

**Builders** Builders only get their priority adjusted in the third face. However a builder should always be in another group, either *mobile builder* or *factory* and can thus get a 'normal' priority in the second stage. These are described in the paragraphs below.

In the third stage of the priority generation the system checks if it has a builder that can build the highest priority unit—the builder doesn't have to be available at the moment. If no builder exists that can build the highest priority unit a builder unit that can create the highest priority unit will acquire the *FORCE\_BUILD* priority making it top priority to build that builder unit. This would have the effect of building the builder that then can build the actually unit with the highest priority.

Another occasion when a builder will acquire the *FORCE\_BUILD* priority is when the metal has come above a certain threshold specified in the source code. When it

comes over this threshold it's rather safe to assume that our income is being wasted, i.e we use less resources than we produce.

**Mobile Builders and Factories** Of all the mobile builders only the airborne builder has a special algorithm, the rest will default to the minimum builder priority. You may ask why only the airborne builder uses an algorithm; the airborne unit exceeds in speed, both as in travel speed and in performance since a path doesn't have to be calculated for the unit when it moves. The amount of airborne builders Al Ice will have increases linearly over time.

As with the airborne builder, the number of factories also increases linearly over time.

**Economics** The general idea is that Al Ice should increase it's energy income exponentially in the beginning until a threshold is hit. The same goes for metal with the difference that the income should increase linearly instead of exponentially.

**Energy** The energy income increment increases exponentially, however there is a threshold of the increment. When the increment value hits the threshold it will stay at that value. This is easiest described through pseudo-code and an example, which is found in listing 3.4 and an example in table 3.1.

LISTING 3.4: Energy Increment Over Time

```
double expIncrement = INCREMENT_EXP * GAME_TIME;
double currentIncome = getIncome(Energy);
double increment = expIncrement + INCREMENT_START;

// Ceil the increment to the max value (threshold)
if (increment > INCREMENT_THRESHOLD) {
    increment = INCREMENT_THRESHOLD;
}

double shouldHave = increment * GAME_TIME;
double diffIncome = shouldHave - currentIncome;

// Apply fusion priority
priority = FUSION_PRIORITY * diffIncome / FUSION_INCOME;
```

Most of the variables are self describing; however there are some hidden ambiguous information to some of them. *GAME\_TIME* is the elapsed minutes since the game was started. *currentIncome* includes the income of those economic buildings that are being built at the moment. At the start of the algorithm the exponential increment is calculated and added to the increment along with the initial increment; this value get capped to the maximum value, *INCREMENT\_THRESHOLD*, if it exceeds it. Next the difference between the current income and the amount of income that Al Ice should have is calculated. It is then used to set a priority for the Fusion Reactor, which is the

only energy producing building Al Ice can build<sup>5</sup>. *FUSION\_INCOME* is the amount of income a single Fusion Reactor produces.

Table 3.1 displays the increment over time with different *INCREMENT\_EXP* values. *INCREMENT\_START* is set to 5.0 and *INCREMENT\_THRESHOLD* is set to 20.0. Table 3.2 shows a similar table which displays how much income Al Ice should have after the a certain amount of time.

TABLE 3.1: Energy increment per minute over time

		Minutes			
		5	10	15	20
Exp	0.25	6.25	7.5	8.75	10
	0.5	7.5	10.0	12.5	15.0
	1.0	10.0	15.0	20.0	20.0
	1.5	12.5	20.0	20.0	20.0

TABLE 3.2: Energy income over time

		Minutes			
		5	10	15	20
Exp	0.25	31.25	75.0	131.25	200.0
	0.5	37.5	100.0	187.5	300.0
	1.0	50.0	150.0	300.0	400.0
	1.5	62.5	200.0	300.0	400.0

As can be seen in these tables are that the increment and income grows very fast, even for low exponential values.

**Metal** The metal income increases linearly over time. Since there are two ways of producing metal; through metal extractors, the preferred way, or through metal maker, which converts energy into metal. Metal extractors needs to be built on specific spots on the map thus limiting the number of metal extractors one can build. For 1) there are a limited number of extraction points; and 2) enemies will also want those extraction points making them an intensive spot for battles. Metal makers can be placed anywhere but draws a huge amount of energy; 20 energy or 2 fusion reactors. Thus building metal extractors when extraction points are available would be best.

Al Ice always tries to own a certain amount of percentage of all extraction points found on the map. Before it has these amounts of extraction points metal extractors will get a higher building priority. After Al Ice has occupied the extraction points it will still try to build metal extraction until there are less than x free and the percentage of the free spots are less than y; x and y are specified in the source code. Note that both

<sup>5</sup>There are two other buildings that can produce energy. Fusion reactors have the benefit of producing a lot of energy on a small area and can be built anywhere.



have to be true if the priority for metal extractors should be zero. This helps Al Ice cope with the difference between huge and small maps. E.g. If a map only has 10 metal extractors it would probably be good to get a fourth metal extraction point but not the fifth. This can be achieved by setting  $x$  to 10 and  $y$  to 25%. Now if the enemy has 4 extraction points and we have 3, 30% would then be free, but only 3 extraction points. Thus we decide to expand our income to the 4th extraction point. Now only 20% is free making the statement false. In a large map where there might be up to 100 extraction points Al Ice would stop when only 9 extraction points are free. This could be seen as the percentage value of 10%—it still builds when 10% is free, thus it stops at 9%. In a short summary, if it's a large map it would use the fixed value and in a small map the percentage value would be used.

When a metal extractor can't be built due to the algorithm above, metal maker will instead take the place of producing metal. This is also why the energy increment isn't linear because more energy is needed later in the game, both because of units but also because Al Ice tries to create metal makers instead of metal extractors.

**Storage** Storage buildings aren't used as much as human players use them. Al Ice usually always have the current metal near zero—if it doesn't it will build a new builder that makes it go to the bottom again. Thus the only reason for creating storage is energy. Energy isn't used all the time and some times it's needed more than others making storage buildings the perfect buffer for energy. The priority of storage is increased by the number of the current energy income.

**Utility Buildings and Healers** Buildings and healers aren't implemented in Al Ice v0.3. The buildings only enhance the strategic options that can be used in Al Ice but aren't a part of this thesis. The same goes for healers.

**Scouts** Al Ice always wants to scout; thus making the priority of scouts very high, in fact if Al Ice doesn't have *SCOUTS\_MIN* it will force the build of a scout using *FORCE\_BUILD* priority value. That's really all about the scout priority.

### 3.2.6 Task System

The task system consists of a task handler; which handles all the active and halted tasks—task units; which is easily described as a wrapper for a unit and its tasks—the task base class; all tasks derive from this class—task observers; task observers implement the *ITaskObserver* interface.

The task system is based on several articles about various task systems found throughout the *AI Game Programming Wisdom* series added with our own spice—where inspired it will be cited.

#### Tasks

A task is in general the same thing as a goal in our implementation; i.e. a task wants to fulfill it's goal—a move to task will try to move the unit to the specific location. There are different kinds of tasks; these are described in 3.2.6. All tasks derive from the task



base class that consists of several abstract methods which the derived classes need to implement.

**execute() : Status**

Executes the task; usually this is where an AICCommand is sent to the Spring Engine if it hasn't been sent already. The execute method is mainly based on the system described in [6]. The method also returns a status depending on if the task is running, done either by successfully completing or failed by some means.

**EXECUTED\_SUCCESSFULLY** When a task has executed successfully, but isn't done.

**COMPLETED\_SUCCESSFULLY** The task has completed the task successfully.

**FAILED\_CLEANLY** One of the return statuses when a task has failed. This status is returned when the task hasn't changed the environment; i.e. when no resources has been used, both resources as in metal and energy, and units. E.g. for an attack task this would be before a move or an attack command was issued. Another example is when a unit should be built but the builder died before the command was issued.

**UNEXPECTED\_ERROR** This happens when a task has changed the environment and then fails; i.e. when resources has been used on the task.

**halt()**

Halts the task; if the task has sub-tasks these also get halted. Note that the task itself needs to handle this and that all method calls to halt only should be done through the task handler's `haltTask(...)`. If the task is bound to a unit it will issue the *stop command* to the unit.

**resume()**

Resumes the task; as with halt, if the task has sub-tasks it will need to resume these manually. This method can be seen as a restart method for the task.

**Task types** The tasks can be split into two groups; 1. *Simple task* which only consists of the task alone; 2. *Composite task* which can be composed of simple tasks and/or composite tasks.

## Task Unit

The task unit can easiest be described as a wrapper of a Spring unit and tasks. A task unit can have three tasks directly bound to it; i.e. the tasks it's currently executing. A unit is not considered free if it isn't currently executing a task, although there is one exception to this.

**Task priority** Tasks are ordered into three priorities; High, medium, and low. Low is treated as an idle task; meaning that if a task unit has an idle task and no medium and high tasks it will be treated as free. These priorities are sufficient for the current use—adding more priorities are easy since they are a part of an enumeration; thus it's

'only' to add an enumerate and increase it's size. A similar approach can be seen in [7] where units have three different goal queues; a long-term goal queue, that can roughly be compared to the idle task; the reactive goal queue with a medium priority task; and lastly the immediate goal queue with a high priority task.

**High-level task** A task unit can also have a high-level goal bound to it. This will have the effect of always making the task unit busy. E.g. this functionality is used in the attack task making task units busy. If it wasn't used, units would be free when waiting for others to catch up in the regroup or move to destination position.

**Setting tasks** If the task unit has a task with higher priority when a new task is added, the newly added task will halt immediately before it's execute is run. Likewise if the task unit has a lower priority the lower priority task will get halted. If a task is inserted and there already exists another task with that priority an error will occur and the method for setting a task will return false.

**Removing tasks** Task units implement the `ITaskObserver` interface so that they automatically remove tasks when they are completed. This makes the system very easy to use; however to bind a task to a task unit the appropriate run method in the task handler needs to be invoked. If a task is completed it first searches through the tasks in the priority task list; if it's found the task will be removed and a lower priority task will be resumed if no higher task priority is found. Note that a higher task priority will only exist if the task was forcibly removed, i.e. it didn't complete or fail naturally. If the task wasn't found in the ordinary task it 'must' be a high-level task; if it were the high-level task it will remove it, else it will just log that a severe error has occurred.

### The Task Handler Agent

The task handler manages all the active and halted tasks; finished tasks are automatically removed and their respective observers are called. A task can only have one task observer that isn't a task unit. If a task is bound to one or more task units they are also added to the observer list. The task handler supports latent execution meaning that it runs the task over multiple frames, it also runs one iteration through all the tasks over multiple frames.

**Task grouping** In addition to the standard task type, they can be grouped by another category; how they interact with task units. *a)* When a task interacts with exactly one unit, i.e. when the task is a simple task. Note that just because the task is a simple task it doesn't mean that it interacts with a unit—an *abstract task* is an example of this; *b)* a high-level task for one or more units. This should be the task that directly handles the sub-tasks that interact with the units; and *c)* tasks that has no direct interaction with units. This includes abstract tasks which are simple tasks but with no unit bound to it—e.g. a task that gets completed when we have a certain amount of units available, and high-level tasks which handle other high-level tasks.

With this it's easy to manage the connection between a task and units. Either there is a direct connection, an indirect connection, or no connection. Which leads on to the next topic; adding and handling tasks and thus the task unit.

**Adding a task to the task handler** When a task gets added to the task list tasks are added to the task handler through three different types of run methods; two of whom can bind a task to a task unit. These methods are described just below.

**run(Task, ITaskObserver)**

Adds a task to the task handler with an optional task observer. Does not bind a task unit to the task.

**run(Task, ITaskObserver, TaskUnit, TaskPriority)**

Adds the task to the task handler and binds a task unit to it with a specific task priority. If the task unit already has a task with that priority the method will return false and no task is added. When the task is finished the task unit's `onTaskFinished(Task, Result)` will be invoked and the task will be removed from the task unit.

**run(Task, ITaskObserver, LinkedList<TaskUnit>, TaskPriority)**

Except adding the task to the task handler it binds the task as a high-level task to the units in the list. If the task unit already has a high-level task the method will return false and no task will be added. When the task is finished all the task units' `onTaskFinished(Task, Result)` will be invoked and the high-level task will be removed from all task units. The task priority isn't used at the moment, but is intended to be used when task units can have several high-level goals.

As can be seen all methods have two arguments in common; the task and the optional task observer. It should also be noted that a task only allows one task observer, if you exclude the task units which handles themselves. It is implemented in this way to lower the coupling between classes and keep the hierarchy intact; the task should only have one, or no owner.

### **ITaskObsever interface**

To listen to tasks that finish a class needs to implement the *ITaskObserver* interface. When a task is completed, either successfully or has failed by some means the task observers are called. The arguments are the task itself so that the observer can determine which task that was completed if it listens to more than one, and the result of the task being one of 'Statuses' described in [3.2.6 Tasks](#).

### **Central tasks**

These tasks are the marrow of Al Ice; without these Al Ice would probably stand still and do nothing.

**Move close to ...** Does as it sounds. Moves a task unit to a specified location using either a calculated path (for ground units) or a direct location (for flying units).

It returns as completed when the unit has moved close to the destination; this can be specified by an input parameter 'radius' to the task when creating it. If the unit somehow fails to move to the destination, it dies or gets stuck, the task will return that some unexpected error has occurred.

**Build unit by unit** Builds a unit by a unit. The task might seem rather simple but there are some mechanics that are implemented to get this to work as would be expected. The simple fall-through is described below.

1. Checks that the builder really can build the unit; if it can't it will fail cleanly.
2. A listener is added for the unit so that it can listen to when it creates a new building and save that unit.
3. A build spot is calculated; This makes sure that we don't build where we stand, because it takes some extra time to move from this area first. It also checks so that the building position isn't on an extraction point rendering it unusable.
4. Waits until the unit has been built and then returns completed successfully.

The task always checks if the unit that is being built isn't null; provided that the builder has started to build a unit. There is also one more exception that needs to be handled; when the builder becomes idle before the task has finished building—this usually mean that the unit failed to move to the specified position, or could not build on the position.

**Initial build sequence** The initial build sequence is a task that can contain several tasks. Each of the tasks in it's list will be executed in a sequence. Al Ice only uses this task at the start of the game to construct all fundamental buildings and units, but the task can be used in other ways e.g. you can't build a tank before a tank factory has been constructed.

**Scouting** Al Ice has an abstract scout task that is derived from the main Task class. The scout task handles and executes the move tasks but gets the target destination from it's derived classes. Al Ice currently has three different kinds of scouting tasks implemented: random, roaming and extraction point.

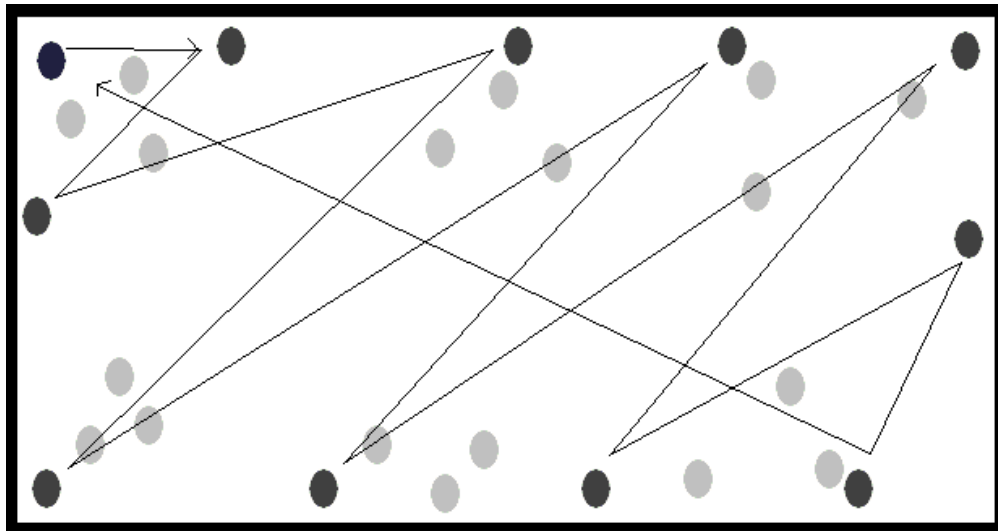
A scout task will be finished if all destinations have been visited and the unit has returned to base, or if the scout unit has been destroyed.

Al Ice depends on the information these scouting tasks provides since Al Ice does not cheat. That's why it always has a scout task running. A big down side with this method is that many scouts might get destroyed which means that additional scouts will have to be constructed.

**Random** The random scout task just generates a number of random positions on the map. It was primarily implemented to test the scouting system and it's components, Al Ice does not currently execute this task.

**Roaming** The roaming scout task is used to scout the entire map, it uses an algorithm to find a route that will make the unit see almost the entire map. But there is one great disadvantage with this scouting method; it takes even a fast moving unit a very long time to finish the scouting route. The path of the scout can be seen in figure 3.2 where the black dots are the calculated positions and the start dot is our start position. The distance between the calculated positions are the same for all sides. The number of dots per side is calculated by using the appropriate length of the side, height or width, and dividing it with the preferred distance between the dots.

FIGURE 3.2: Path of the roaming scout



**Extraction point** The extraction point scouting task is used to scout all the metal extraction points on the map, the scout unit will visit the closest non visited extraction point until every extraction point has been visited. This task gives us very valuable information, just like the roaming scout task, but it's much faster to execute and therefore gives results faster which benefits our priority system. However if the extraction points aren't scattered in a favorably way this scout task will not explore the whole map leaving black undiscovered areas.

**Attack tasks** Al Ice does not have a complete set of attack tasks. For v0.3 of Al Ice two versions of attacks have been implemented. A ground attack that attacks ground targets, and an air attack that attacks air units. Both of these tasks derive from the base attack that is an abstract task class.

**Attack Target** Attack target is a simple task when assigning a unit to attack a single target. The unit will hunt down the target as long as it is alive and can be seen by Al Ice. If it's killed it will return a successful completion whereas it will return an unexpected error when the target escaped from our line-of-sight.

**Base attack** The abstract task class that 'all' attack tasks derive from. The base attack task needs a set of attacking units and optional healers if the attack group should

have designated healers to it. However the healers aren't used in the current version, but are implemented as a stub feature.

When the base attack is run it starts off with regrouping all the unit to the group's center position. If this isn't done the units could be scattered around in the map and would not attack as a mass. As soon as all units has reached the regroup position it will find a target destination to go to. This destination is found using the abstract method `getTargetLocation()` which the derived classes implements; thus the derived class can find a target destination. E.g. it can be an outpost, flying unit etc. When the target destination is reached and cleared of enemies it will try to find a new target destination—this will keep going until all the attacking units are dead.

Whenever an enemy unit is close to the groups center position it will assign attack tasks to attack the enemy for all attacking units. To get a close enemy it uses the second abstract method `getCloseEnemy()` that the derived classes implements; this makes that the derived class has total control of deciding which enemy to attack. After the enemy has died or escaped the units returns to their previous task; regroup or move to target destination.

**Ground attack** The ground attack that derives from base attack has one primary strategy; destroy the enemies metal extractors if some exists (if we have seen one lately). If it can't find any metal extractors it will target the nearest enemy ground unit. The `getCloseEnemy()` simply returns the closest enemy ground unit to the group's center position that is inside a certain radius. The task takes a set of attackers as input parameters and optional healers, which is what the base attack task needs. The General decides how large the attack force should be.

**Air attack** The air attack only attacks air targets, being that air units are in minority of the units the number of air attacks will probably not be high if the enemy doesn't use a huge amount of air units. `getTargetLocation()` returns the last seen position of an air unit and `getCloseEnemy` returns the closest enemy air unit that is inside the range specified by a status. As with the ground attack task it needs an attack force and the General decides how large the attack force should be.

# Chapter 4

## Result

The support for Evolution RTS is rather low and there were some bugs with the Shard<sup>1</sup> AI; thus we could not test AI Ice against it. However AI Ice was tested against RAI v0.601, which is a bot that supports multiple mods. The 'only' available documentation on RAI is the actual source code[8] with almost no comments. This makes it hard to describe and understand why RAI uses different strategies from time to time.

The test was performed on three different maps; 1) CaisRevenge is a smaller, but not symmetric map; 2) DeltaSiegeDry which is medium sized symmetric map and is a very popular map on-line; and 3) DeltaSiegeDryRevX\_v3 is actually the DeltaSiegeDry map but mirror at the bottom making doubling the size of the map. To get a fairly good and equal result the test was run 10 times on each map where the AIs changed start position after 5 times. The fastest, longest, and medium time is calculated, where these values only applies when AI Ice won. The time format is minutes:seconds.

### 4.1 Bot Tests

#### 4.1.1 RAI

The test results between AI Ice and RAI v0.601. As can be seen in table 4.1 AI Ice won

TABLE 4.1: AI Ice vs. RAI

	AI Ice	RAI	%	Medium	Fastest	Longest
CaisRevenge	10	0	100%	16:16	09:10	25:00
DeltaSiegeDry	10	0	100%	20:27	08:57	31:55
DeltaSiegeDryX_v3	9	1	90%	23:13	08:32	38:21

all times except one match on the big map 'DeltaSiegeDryRevX\_v3' where AI Ice lost after half an hour. This sums up to approximately 97% chance that AI Ice will win over RAI.

---

<sup>1</sup>Another thesis bot currently developed.

# Chapter 5

## Discussion

Al Ice did splendid against the RAI v0.601, 97% chance of winning is not bad. On the map that Al Ice lost RAI quickly built outwards towards Al Ice and cornered it. Since Al Ice only has a simple strategic implemented it wouldn't just move past the first 'block', there actually was a free path to a whole other area, and continue to expand but was instead trapped. A simple terrain analysis would've helped it overcome this situation.

Since RAI really plays randomly there were one time when a game ended just after about eight and a half minutes on the big map. RAI simply didn't expand at all thus it was just for Al Ice to storm the base—which didn't have any units because of the low metal income when it didn't expand.

Another one unofficial test was run on an extremely small map—Al Ice lost. It probably lost due to the fact that it's main strategy is to attack with mobile units and not to attack with defensive buildings, whereas RAI built towers that could attack a long distance (the whole map) which destroyed our resources making us an even easier target.

Another important fact to mention is that the RAI AI is using a map cheat, which Al Ice isn't; this means that RAI got an unfair advantage, but this only shows the effectiveness of the adaptive priority calculation. RAI could also have a simple terrain analysis that can be read from the change-log[8]. How advanced it is isn't mentioned.

When playing as a human against Al Ice it's also fairly easy to win if you know how it acts. One strategy that works is to get a great deal of extraction points on the map and build towers around those that are closest to Al Ice—its highest attack goal is the closest extraction point. Thus it will leave the other extraction points in peace and you can then start building up a good tower defense. You can then either continue expanding the base through building towers and using an tower attack strategy or creating units and attacking with them.

The second strategy is to defend a smaller area while building nukes, since Al Ice has no defense against those at all. When you have 2-3 nukes you can start building a smaller attack group and attack while launching the 3-4 nukes you will have by that time.

When the Shard AI gets operational one could have it battle Al Ice and see how well the system does against the official AI of Evolution RTS. The Shard AI uses a map cheat, so it already has an advantage—but it will nevertheless be fun to test how it



competes against it, even though Al Ice isn't advanced at all.

## Conclusion and Future Work

### 6.1 Conclusion

An adaptive AI that doesn't cheat can beat a 'static' AI that does. However care must be taken when adjusting the priorities, and it can be rather hard to adjust the priorities to work for 'all' situations. With a better placement of buildings and attack algorithms it would probably be overwhelming for a human player to keep up with the micromanagement which would certainly be implemented in a real product.

#### 6.1.1 Project/Architecture conclusion

Creating a good architecture that is maintainable and flexible from the beginning and plan for extra functionality helped us enormously. This made it very easy for us to extend the functionality of many components when we needed. If a new feature was added a change was often only require in one or two classes, seldom more than two. See [A](#) for a class diagram of the architecture.

We have both gained enormous experience working with this project; we understand a large amount of how the Spring API works, how much work and how hard it is to generate priorities for units—especially attacking units, how a task system can be implemented and used which isn't specific for the RTS genre, and a lot of other AI and non-AI related topics.

#### 6.1.2 Things we would've done differently

In fact there aren't much that we wouldn't do the same. However there is one area that Al Ice isn't very good of, performance. It uses a lot more of CPU than other bots, that is mainly because most parts of the system isn't really optimized at all. However that wasn't a part of the problem, and with more time some of the performance problems when running with a faster speed could have been fixed. Playing as a human against Al Ice in normal speed wasn't any problem though.

## 6.2 Future work

There are many parts of AI Ice that can either be improved, added, or replaced entirely by something else. You can probably find several features that can be a base for thesis work.

### 6.2.1 Improvements to the current system

#### Priority System

**Optimized priority generation** Instead of generating priorities for all the units when we have an available builder it would be better to only generate priorities for the units that the available builder can build.

**More accurate unit priority** The current priority system does not take into account our own DPS into the calculation of the damage type in the attack force priority, and armored buildings priority generation. The effect of this would be that we create many units that are good against the enemy's armor type, and when we have enough it would start to create units that are good against the second most used armor type. To get a more variety the units priority currently decreases with the number of units that we already have. However this will not make other units which share the same damage type get lower priority which is the desired effect.

For anti-air units there could be some sort of threshold. I.e. AI Ice should always try to have a certain amount of anti-air units, even when no air units have been spotted. This base amount of anti-air units should get higher priority if an air unit has been seen. When the threshold is passed the calculation is done as now; anti-air units get a bonus priority depending on the size of the enemies' air units.

Another approach is to insert a new armor type, flying, and a new damage type, anti-air. The damage multipliers for the all other damage types will be skipped in the priority calculation. The effect of this would make anti-air units have the lowest armor type health if they only consider the air units making them highest priority to create. However this functionality needs to be implemented with the improvement discussed above where our own DPS is taken into account. If it isn't taken into account it will make AI Ice to literally spit out anti-air units.

**Better economics priority over time** Currently AI Ice will always try to increase it's current income linearly for metal and almost linearly for energy. However this algorithm has one major flaw, when an enemy destroys lots of resource buildings it will most certain make the AI to create resource buildings it top priority rather than creating a balanced amount of defensive and recourse buildings.

#### Task system and tasks

**More than one high-level task per task unit** Instead of only having one high-level task task units could have the ability to have different high-level tasks with priorities and a set of 'normal' tasks for each high-level task. This would have the similar effect of stacking tasks. E.g. a task-unit could have a high-level task 'attack' with a normal

priority; the active tasks being MoveCloseTo (normal priority), and AttackTarget (high priority). Maybe the enemy decides to attack and our unit is close to the attacking position; making the General assign a new high-level task 'defend' with high priority to the unit; this would push the current stack of 'normal' tasks and a new empty set would be available where the unit would get a new MoveCloseTo task and AttackTarget task.

### Base attack

**'Checkpoints' to the target destination in base attack** The base attack task could be improved by having several regroup positions on the way to the target destination—now the units are scattered in a fine line due to the fact that units have different moving speed.

**Advanced grouping** Instead of sharing the same attack target for all attacking units it could be split into the type of the damage type the unit has; creating one list for each kind. Then changing the `getcloseEnemy()` to `getCloseEnemy(damageType)`; this would make the derived classes able to chose different targets, i.e. those that are effective, for the different kinds of damage types—making Al Ice destroy targets quicker.

The intention was that the attack target task should look for the best suitable enemy; e.g. if there are several enemy units in attack range, look for an enemy with an armor type that is weak to the units damage type. This way Al Ice would destroy it's enemies much faster and with minimal loss.

## 6.2.2 New Features

### Better path finding

Al Ice path finding and movement system relies on the path finding implement in Spring which is not advance at all; it only has a path finding for the terrain, but not the buildings placed on the map. Units using the internal Spring path finding can easily get stuck, if they try to move in the same direction they will still be stuck and might make more units get stuck. By implementing an overlay on the base path finding to include building avoidance, i.e. not try to move straight through buildings, many functions would function better.

### Advanced terrain analysis

An implementation of advanced terrain analysis would probably improve the probability to beat the enemy. Flanking positions is one good example that would be useful, but most of all is probably where to build defensive (armored) buildings and regular buildings. A test could be conducted to battle this version of Al Ice and how important terrain analysis is.

### Dynamic unit priority

One could implement a dynamic unit priority. Dynamic meaning a unit priority system that would work on every modification based on the Spring engine. This will probably make the AI more attractive because of the popularity of the other mods.

### **Task priority and planning**

Right now Al Ice does not plan or care about the future of the game, all priority calculation is just for the current moment and the General does only try to use all the resources at all time. If one implemented a task priority and a planner for the General. The planner could also be implemented without a pre-generated task priority, it would just analyze the outcome of executing the task and then adding a priority to the task.

### **Lua support**

With Lua support it would be easy for more advanced 'users' to implement more functionality into Al Ice, mostly through tasks. With a planning system it would be relatively easy create new tasks and add them in the set used by Al Ice. It would also make the user able to customize its behaviour by maybe removing or replacing tasks.

### **Various team play features**

As of now Al Ice will act as a stubborn child if a player or an AI plays with it; it wants everything for itself. A new feature could be the ability to play with others in the same team. Three different approaches could be used; One for interacting with other team-members that also are Al Ice, i.e. the two AI-players would act more or less as one; another approach when playing with other AIs probably that the other AI is the 'master', i.e. when it attacks Al Ice attacks; and the last approach when interacting with humans. The use of chat would be a great idea of interaction, whereas the player could assign the AI to do specific things; maybe build an big air attack group, or defend the base. It would also be possible to make Al Ice come up with plans itself; how to attack, what units to build, etc.

The other aspect of team play is the enemy; how many players are there in a team and are there many teams? A good question would then be who to attack and could be implemented with the use of goals.

### **Playing at the same level as the opponent**

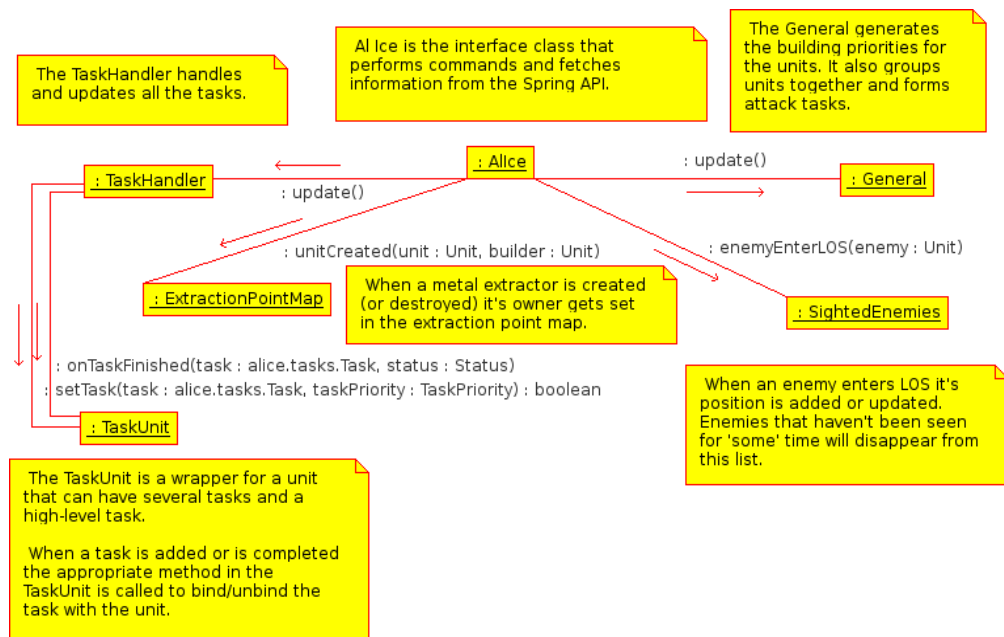
Instead of always trying to do it's best the AI could be made more stupid when the opponent isn't as good with the goal to create a 50-50% success rate. Creating an AI like this would make it more fun to play against, and as you get better it will also.

# Appendix A

## AI Ice Architecture

### A.1 Interaction Between Agents

FIGURE A.1: Collaboration diagram between agents in AI Ice.

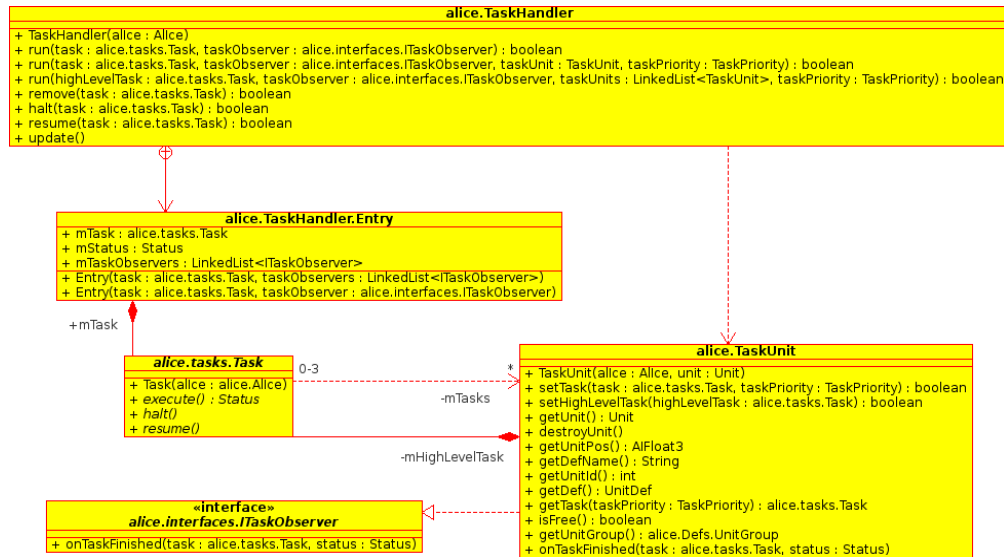


Most of the system is described in comments inside the figure. To avoid making the diagram more cluttered by text and connections than it already is, only the most important methods are covered.

### A.2 Task System

Figure A.2 shows the central parts of the task system. Some of the functionality has been described in 3.2.6, thus this figure is mostly for enlighten and gather all the functionality in one place.

FIGURE A.2: Task system class diagram.



### A.3 Complete Documentation

For a complete documentation on the API we refer to the [javadoc](#) documentation. If the page is off-line or isn't available it could have been moved, please email one of the authors [Matteus Magnusson](#) [Tobias Hall](#).

# Bibliography

- [1] Spring homepage, May 2010. URL <http://springrts.com/>.
- [2] Evolution RTS, May 2010. URL <http://www.evolutionrts.info/>.
- [3] Forboding Angel. How does the new armorclass system work?, April 2010. URL <http://www.evolutionrts.info/?p=446>.
- [4] Kevin Dill. Getting started with decision making and control systems. In Steve Rabin, editor, *AI Game Programming Wisdom 3*, pages 321–330. Charles River Media, 2006.
- [5] Kevin Dill. Embracing declarative ai with a goal-based approach. In Steve Rabin, editor, *AI Game Programming Wisdom 4*, pages 229–238. Charles River Media, 2008.
- [6] Alex J. Champandard. Getting started with decision making and control systems. In Steve Rabin, editor, *AI Game Programming Wisdom 4*, pages 257–264. Charles River Media, 2008.
- [7] Brett Laming. The marpo methodology: Planning and orders. In Steve Rabin, editor, *AI Game Programming Wisdom 4*, pages 239–255. Charles River Media, 2008.
- [8] RAI source code, May 2010. URL <http://springrts.com/wiki/AI:RAI>.



# Index

- architecture, 24, 28
- armor system, 4
- armor type, 4, 10
- attack force, 7, 9
- base attack, 19
- base attacks, 26
- builder, 7, 11
  - factory, *see* building-factory
  - mobile, 7, 12
- building
  - armored, 7, 11
  - economic, 7, 12
  - factory, 7, 12
  - utility, 7, 14
- command
  - stop, 15
- damage type, 4, 9
- economics only, 8
- Evolution RTS, 3
- extraction point
  - map, 7
- General, 3, 8, 20
- ground attack, 20
- groups, 7
- healer, 7
- initial build sequence, 8, 18
- priority
  - armored building, 11
  - attack force, 9
  - builder, 11
  - energy, 12
  - factory, 12
  - healer, 14
  - metal, 13
  - mobile builder, 12
  - scout, 14
  - storage, 14
  - utility building, 14
- priority system, 8
- scout, 7, 14
- sighted enemies, 7
- task, 14, 25
  - abstract, 16
  - composite, 15
  - execute(), 15
  - halt(), 15
  - priority, 15
  - resume(), 15
  - simple, 15
  - type, 15
- task observer, 17
- task system, 25
- task unit, 15