Build-order Optimization In Real-time Strategy Games Using Evolutionary Techniques

Matthieu Macret
School of Interactive Arts and Technology
Simon Fraser University, Surrey, B.C., Canada V3T 0A3
mmacret@sfu.ca

ABSTRACT

As did board games like Chess in the past, real-time strategy games are becoming popular test applications for real-time AI research. In this paper we approach the real-time planning problem by considering build-order optimization in real-time strategy games. This problem class can be formulated as a resource accumulation and allocation problem where an agent has to decide which objects to produce at what time in order to meet one of several goals: either maximizing the number of produced objects in a given time period or producing a certain number of objects as fast as possible. We propose a system using genetic programming which optimizes build orders in the commercial real time strategy game Starcraft©.

Categories and Subject Descriptors

I.2.1 [**Applications and Expert System**]: Games

General Terms

Algorithms, Measurement, Performance, Experimentation.

Keywords

Real-time strategy games, optimization, genetic programming.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Conference'10, Month 1–2, 2010, City, State, Country.
Copyright 2010 ACM 1-58113-000-0/00/0010…$10.00.

1. INTRODUCTION

Video games are an excellent test bed for research in AI. A majority of games demand real-time AI that must be capable of challenging and entertaining the player.

Real-time Strategy (RTS) games in particular are an interesting domain for evaluating AI techniques [1], because they provide several challenges for building game AI. The decision complexity of RTS games is huge [2], and requires simultaneous reasoning about both strategic and tactical goals. Research in RTS game AI can be divided into two branches: higher-level and lower-level AI. Lower-level AI refers to the behaviors of single units and small groups of units that are given commands. Higher-level AI refers to management of resources, decisions on what to build, and strategic decisions such as sending units into battle.

One of the focuses of strategic play in RTS games is a build order. A build order defines the sequence in which buildings are constructed, units are produced and technologies are researched. Build orders target a specific strategy, such as rushing or timing attacks. Rush strategies attempt to overwhelm an opponent with inexpensive combat units early in the game, while timing attacks engage opponents based on a trigger, such as completing an upgrade. Given the size of the decision space in RTS [2], optimizing build-orders is not trivial.

We present a system based on a genetic approach for optimizing build orders. The system is implemented with the Evolutionary Framework OpenBeagle and uses a Starcraft© mechanics anthology and a rules-based simulator to evaluate the fitness function of the different individuals.

2. Related work

The problem of build-order optimization was first discussed by Kovarsky and Buro [3]. They present a formal formulation of this problem using the Planning Domain Definition Language (PDDL) [4]. This language was developed to standardize planning domain and problem description in order to enable different planners to compete against one another in international planning competitions.

The RTS technology tree that specifies the relationships between units, buildings, and resources defines this domain.

They situate this problem only on the initial game phase where there is no or little interaction with the opponent.

They consider only optimization in terms of time and resources used.

The specification of the solution of the problem stays close to the game and can be easily understood by a player or interpreted by a game engine. Indeed, the build-order is expressed by a sequence of actions to execute in the game, for example, "*create-worker, create-marine*" (see Figure 1).

They emphasize that this build-order optimization problem is very challenging for automated planning systems because, contrary to common planning domain, systems have to deal with unit creation and destruction and they have to decide what actions can be executed.

Zhan Wei and Wee Sun [5] use a constraint-based solver to deal with this problem. In order to represent the planning domain, they translate game rules such as RTS technology tree and the availability of resources into constraints using producer/consumer constraints, cumulative constraints and disjunctive constraints.

The input of their system is a game state represented by a list of units and buildings the user wants. They reason about execution time of build order in order to select the fastest to achieve the targeted goal.

The advantage of their system is that it is supposed to work in real-time. Thus, build orders can be recalculate in real-time during the game.

However, it is not easy to express the problem in term of constraints and, given the size of the decision space, it is too restrictive to consider only one solution by couple (actual game state, targeted game state).

This solver is compared to other solvers such as the greedy solver, round-robin solver or branch and bound solver. It shows good performances on simple cases. However it only look at one part of the optimization problem, which is optimizing *time execution*.

Weber and Mateas propose an other method for selecting build orders. They are using case-based reasoning. In order to build their case library, they analyse games between different scripted bots. Each case is represented by a starting game state and a build order. They design a system to generalize these cases using the conceptual neighborhood technique. Then, these generalized cases can be recalled during the game depending of the current game state.

Since they are reasoning about cases, the optimization performance is directly dependant of the quality and the size of the case library. Specifying cases that would be relevant for optimizing build order is not trivial.

Weber and Ontanon[7] propose a system to extract automatically cases from replays in order to build an efficient case library. To achieve this task, they develop a Goal ontology for RTS. Actions in traces extracted from the replays are broken into cases and label with a goal.

The system was evaluated by collecting several StarCraft replays and converting them into cases usable in a case-based planner. It was able to set up a resource infrastructure and begin expanding the tech tree, the system is currently unable to defeat the built in AI of StarCraft, which performs a strong rush strategy.

Ponsen et al [8] propose another system to produce a case library. They are using genetic algorithms to evolve build orders. In their system, a gene is an action in the game, for example, *Build a town hall*. These genes are combined to make up chromosomes which are actual build orders. During the system's evolution, genetic operator are applied on the BO population such as cross-over, mutation, substitution and selection. Selection is done by evaluating directly the build orders in the game. The BO are implementing in a bot which has to play against the standard game AI. The fitness function is proportional to the performance of the build order in this game.

The best build orders resulting of their system were able to defeat the standard game AI. However, evaluating individuals making them play against another bots takes a lot of time and makes the best individual only able to win against the bots it was trained to play against.

3. GENETIC PROGRAMMING FOR BUILD ORDER OPTIMIZATION

In this section, we describe how genetic programming can be applied to address the build order optimization problem in Starcraft. Moreover, genetic approaches showed that they are able to generate unexpected solutions [8], which is very promising in RTS where unexpected build orders could surprise the opponent and give a certain advantage to the player or bot using them.

Starcraft human gamers can also take advantage of this system to help them work out new strategies.

3.1 Starcraft Simulator

In order to evaluate build orders, we develop a simulator of the Starcraft's mechanics. This simulator takes as input an initial game state and a build order and gives the evolution in the time of the game state when the BO is applied.

It is implemented using a rule-based system.

3.1.1 Game State and build order

In our system, a game state is described by:

- The time since the beginning of the game,
- The player's resources: crystals and gas.
- The lists of units and building that belong to the player but also the upgrades he has developed.
- Some statistics such as global defense score or global attack score.

A build order is a sequence of actions to perform in the game. A example of Build order is: *build a SCV*, *build a Command Center*, *Build a Barrack*, *Build a Marine*, *Build a Marine*.

3.1.2 Starcraft gameplay

In Starcraft, the player has to perform four tasks concurrently:

- Gathering resources
- Building her base
- Building her army
- Manage her army in order to destroy the enemy.

The rules of the game are mainly constraints on the construction of units or building:

A building needs a working unit (SCV) to be build, a fixed quantity of resources and time and can have other buildings as prerequisites.

Units have also a resources and time cost and need an available building to be produced. Some of these constraints can be summarized in the tech tree in Figure 1. For instance, an Academy needs a Barracks to be built or a Missile Turret needs a Engineering Bay.

Units and buildings also have attributes such as hit points, armor score or attack score.

All these data are needed to check if a build order is well formatted and to interpret it in terms of new game state and statistics.

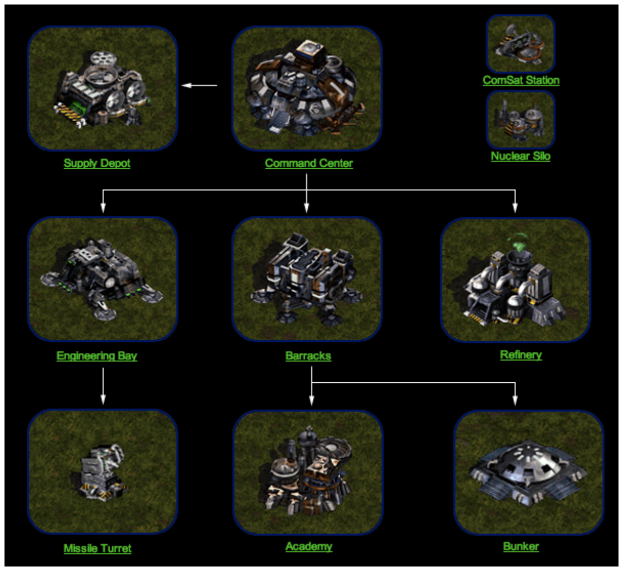


Figure 1: Terran Tech Tree

3.1.3 Rule-based system

We use CLIPS (C Language Integrated Production System) to implement our rule-based system. CLIPS is a public domain software tool for building expert systems. The first version were developed in 1985 at NASA. CLIPS is one the most widely used expert tool because it is fast, efficient and free.

We choose to use this technology to build our simulator because we needed speed and robustness. Indeed, this simulator is designed to be part of our genetic programming system. It will be used to evaluate each individual of the population. As the population could be huge (thousands of individuals), evaluating one individual has to be as fast as possible in order to reduce the execution time of the whole system.

The game state, in our system, is represented by facts. Each building or unit is represented by three facts:

- (waiting name nbr) which shows the number of building/units which are busy building an other unit/building.
- (available name nbr) which shows the number of building/units which are available to build an other unit/building.
- (name boolean) which shows if the player owns at least one building/unit of this type.

Each upgrade is only represented by the last type of facts described above.

There is also other fact to represent:

- resources. (resources name nbr)
- size of the population. (supply nbr)
- time. (time nbr)

The game mechanics are implemented with rules. For each building/units/upgrades, two rules are provided:

- a rule to put the order in queue (see Figure 9),
- a rule to actually execute the order and add the building/unit/upgrade to the fact list (see Figure 10),

For example, the order “build a marine” has to be simulated. A fact BuildMarine is added to the system. All the prerequisites concerning this order are checked. (ex: resources, type of building needed…) If they are validated, the first rule is fired and a fact “*waiting buildMarine timeToBuild”* is added to the system. The second rule would only fire when the time of the system would be equal to *timeToBuild*.

Finally, a last rule to update the time is implemented in the system.

Figure 2 shows how a whole build order is simulated. Each order of the build order is added to the system and the time is updated until the maximum time specified by the user. At each time, a line representing the game state is added to a game state file.

This file is a speadsheet. Figure 8 shows its structures.

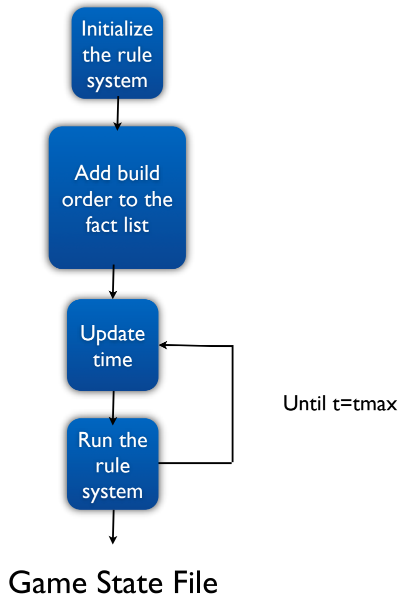


Figure 2: Build order simulator principle

3.2 Evolutionary system

3.2.1 Open Beagle Framework

We use the C++ Open Beagle Framework to implement our genetic algorithm. Open BEAGLE is made to be extensible, to allow the user to implement his own algorithm, with little efforts and minimal code writing. Open BEAGLE was designed with the objectives of providing an Evolutionary Computing framework that is generic, user friendly, portable, efficient, robust and free.

To implement our system using this framework, we have to specify only three elements:

- The primitives and the terminals,
- The genetic operators,
- The fitness function.

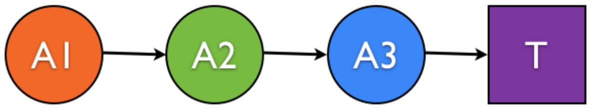


Figure 3: Example of genotype.

3.2.2 *Primitives and terminals*

The primitives are the elements that are composing the genotype of each individual and a terminal is an element that is ending a genotype.

In our system, we define a primitive as an action in the game. For example, *Building a marine* is a primitive. (A1 in Figure 3)

We only use one kind of terminal that represents the end of the build order. (T in Figure 3)

The resulting genotype of an individual is a build order.

3.2.3 *Genetic operators*

We use standard genetic operators such as selection, crossover, substitution and mutation.

3.2.4 *Evaluator*

The fitness function is used to evaluate each individual in order to keep only the best individuals in the next generation.

In our system, we implement a Build Order Converter that converts a genotype in a build order to be used as an input of the simulator described in section 3.1.

The Evaluator object contains a Starcraft simulator and a fitness function. The evaluator takes as input the converted genotype as well as the initial game state. The fitness function is designed to specify the optimization criteria. Figure 4 summarizes this architecture.

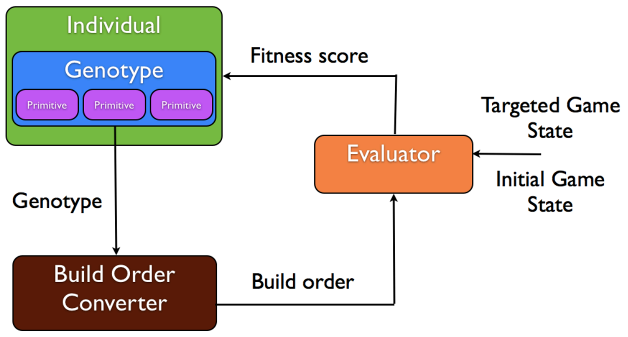


Figure 4: Genetic approach principles

The fitness function aims to evaluate several features of the individuals that can be selected among the followings:

- The distance between the targeted game state and the game state reached by the converted build order,
- The time to execute the build order in game,
- The resources used by the build order,
- The global attack score or the global defense score.

This list is not exhaustive and can be extending with other criteria.

The design of the fitness function is directly linked to the choice of the optimization criteria.

For example, if the goal is to optimize a build order to have 10 marines at t=20, an example of good fitness function would be:

$$\frac{Number\quad of\quad marines(t=20)}{10}$$

More complex optimization criteria can be taken into account. For example:

- Maximizing resources,
- Have at least 10 marines
- Have a barrack at t=44s
- Minimizing the number of order

In this case, a example of fitness function would be:

$$\frac{1}{Nbr\quad of\quad order(t=t\max)}\times\frac{Nbr\quad of\quad marines(t=t\max)}{10}$$
$$\times resources(t=t\max)\times Nbr\quad of\quad barracks(t=44)$$

The use of the fitness function makes very convenient the specification of the optimization criteria. Indeed, only the fitness function has to be modified when different optimization criteria are chosen.

The genetic programming system aims to maximize the fitness function. Thus, the resulting individuals are solutions of the build order optimization problem as designed in the fitness function.

4. DISCUSSION

As proof of concept, we conduce a small experiment. A very simple optimization problem is defined: *Have 10 SCV at t=300s.*

The Starcraft simulator is limited to 4 actions:

- Build a Supply Depot,
- Build a SCV,
- Build a Marine,
- Build a barrack.

Concerning the genetic programming parameters, the population size is limited to 200, the number of generation to 50 and the maximum size of a Build Order to 25 actions.

The fitness function is defined as follows:

$$\frac{Number\quad of\quad marines(t=300)}{10}$$

Graph shows the evolution of the mean of fitness score with the generations. The fitness score is increasing with the generations to reach a floor around 1 after 9 generations. It shows that the BOs after generation 9 are optimized according to our criterion (ie there is more than 10 SCV after applying the BO).

A peak of 1,8 is even observed at generation 5, which means that the optimization was even overtaken with 18 SCV!

The best Build Order of the evolution is the following:

- build a SCV,
- build a Supply Depot,
- build a SCV,
- build a Supply Depot,
- build a SCV (7 times)

A very interesting thing to notice is that in this build order, there is only two types of actions (BuildSCV and buildSupplyDepot). Thus, the system was actually able to learn that:

- it is necessary to build a supply depot early in the game to overcome the limit of eight units in the population,
- building a barrack or a marine is useless.

Contrary to other systems where the game or replays are used to evaluate the BO, we are reasoning directly about the game mechanics. It makes the optimization more objective than in Ponsen's system[8] for example, where the BOs are optimized against a specific AI opponent.

Using a genetic approach makes our system able to optimize a build order to reach a targeted game state in terms of several criteria such as time, resources or others statistics (attack score, defense score...) and not only for one or two features like in other system [2,5,7,8]

The principal difficulty here is to define the good fitness function in order to represent the optimization criteria.

Future works will be to experiment with the whole rule system and a more complex fitness function. An user interface to easily set up the fitness function and therefore the optimization criteria will also be developed.

Moreover, this tool has potential to explore computer creativity in real time strategy game. Indeed, designing original build orders is a creative task and gives the advantage to the player who uses them to surprise his opponents. A lot of Build orders are already well known by the Starcraft community, such as *roach rushes* or the *fortress* build orders. A large library of Starcraft replays are available on internet and could be analyzed to extract already known build orders. The resulting build order library could be used in combination with the genetic-based system presented here to find original but also competitive new creative build orders.

Finally, the build orders generated by our system have also potential to be used as cases in a learning case-based system like Darmok2[7] to play Starcraft.

.

5. REFERENCES

[1] Buro, M.: Real-Time Strategy Games: A New AI Research Challenge. In: *Proceedings of the International Joint Conference on Artificial Intelligence*. (2003) 1534–1535

[2] Aha, D.W., Molineaux, M., Ponsen, M.: Learning to Win: Case-Based Plan Selection in a Real-Time Strategy Game. *Lecture notes in computer science 3620 (2005) 5–20*

[3] Kovarsky, A., and Buro, M. 2006. A First Look at Build-Order Optimization in Real-Time Strategy Games. In *Proceedings of the GameOn Conference*, 18–22.

[4] McDermott D. and Committee A., 1998. PDDL – the planning domain definition language. In *Technical Report*.

[5] Wei, L.Z. and Sun, L.W. *Build Order Optimisation For Real-time Strategy Game.*

[6] Weber, B., Mateas, M.: Case-Based Reasoning for Build Order in Real-Time Strategy Games. In: *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference*, AAAI Press (2009) 106–111.

[7] Ben G. Weber and Santiago Ontañón (2010) Using Automated Replay Annotation for Case-Based Planning in Games. In *ICCBR 2010 workshop on CBR for Computer Games*, pp. 15 - 24.

[8] Ponsen, M. and Munoz-Avila, H. and Spronck, P. and Aha, D.W. 2006. Automatically generating game tactics via evolutionary learning. In *AI Magazine*.

APPENDIX

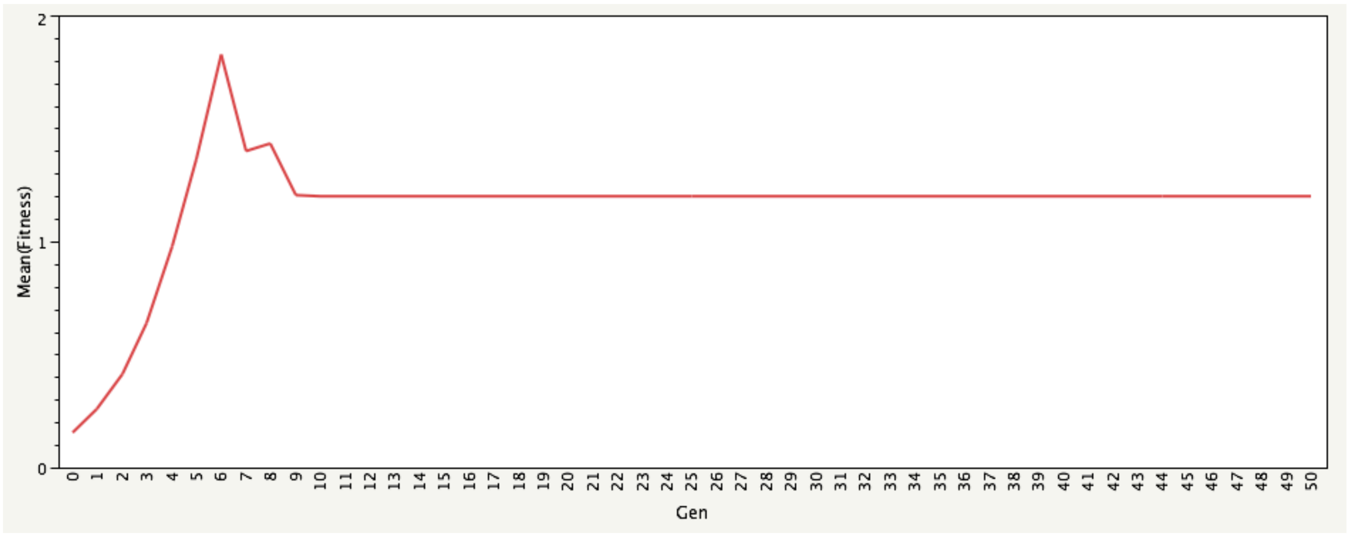
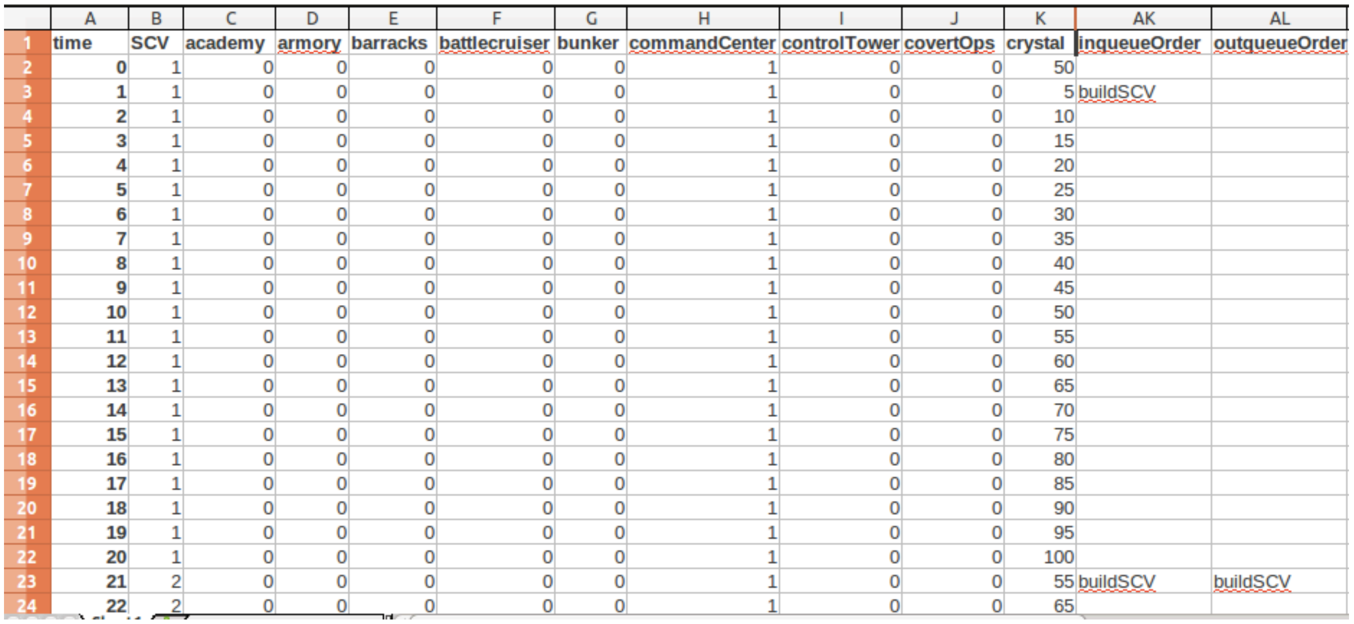
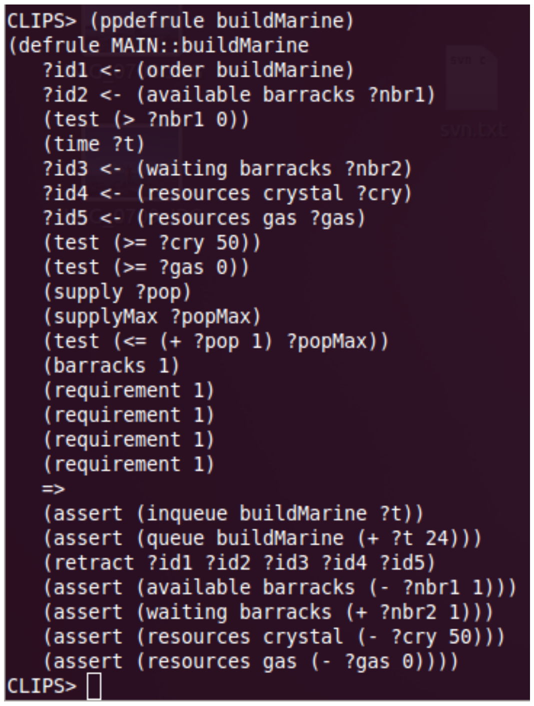


Figure 7: Mean of the fitness score in function of the time



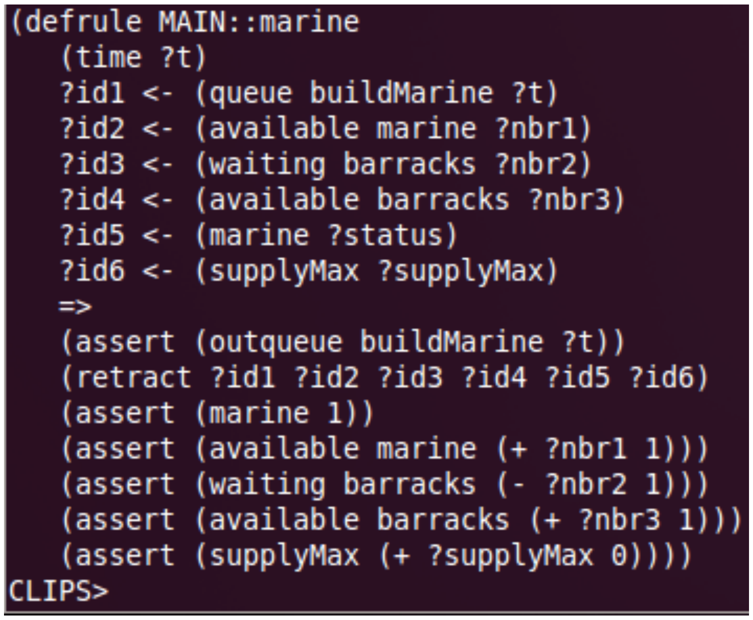
	A	B	C	D	E	F	G	H	I	J	K	AK	AL
1	time	SCV	academy	armory	barracks	battlecruiser	bunker	commandCenter	controlTower	covertOps	crystal	inqueueOrder	outqueueOrder
2	0	1	0	0	0	0	0	1	0	0	50		
3	1	1	0	0	0	0	0	1	0	0	5	buildSCV	
4	2	1	0	0	0	0	0	1	0	0	10		
5	3	1	0	0	0	0	0	1	0	0	15		
6	4	1	0	0	0	0	0	1	0	0	20		
7	5	1	0	0	0	0	0	1	0	0	25		
8	6	1	0	0	0	0	0	1	0	0	30		
9	7	1	0	0	0	0	0	1	0	0	35		
10	8	1	0	0	0	0	0	1	0	0	40		
11	9	1	0	0	0	0	0	1	0	0	45		
12	10	1	0	0	0	0	0	1	0	0	50		
13	11	1	0	0	0	0	0	1	0	0	55		
14	12	1	0	0	0	0	0	1	0	0	60		
15	13	1	0	0	0	0	0	1	0	0	65		
16	14	1	0	0	0	0	0	1	0	0	70		
17	15	1	0	0	0	0	0	1	0	0	75		
18	16	1	0	0	0	0	0	1	0	0	80		
19	17	1	0	0	0	0	0	1	0	0	85		
20	18	1	0	0	0	0	0	1	0	0	90		
21	19	1	0	0	0	0	0	1	0	0	95		
22	20	1	0	0	0	0	0	1	0	0	100		
23	21	2	0	0	0	0	0	1	0	0	55	buildSCV	buildSCV
24	22	2	0	0	0	0	0	1	0	0	65		

Figure 8: Game state file example



```
CLIPS> (ppdefrule buildMarine)
(defrule MAIN::buildMarine
   ?id1 <- (order buildMarine)
   ?id2 <- (available barracks ?nbr1)
   (test (> ?nbr1 0))
   (time ?t)
   ?id3 <- (waiting barracks ?nbr2)
   ?id4 <- (resources crystal ?cry)
   ?id5 <- (resources gas ?gas)
   (test (>= ?cry 50))
   (test (>= ?gas 0))
   (supply ?pop)
   (supplyMax ?popMax)
   (test (<= (+ ?pop 1) ?popMax))
   (barracks 1)
   (requirement 1)
   (requirement 1)
   (requirement 1)
   (requirement 1)
   =>
   (assert (inqueue buildMarine ?t))
   (assert (queue buildMarine (+ ?t 24)))
   (retract ?id1 ?id2 ?id3 ?id4 ?id5)
   (assert (available barracks (- ?nbr1 1)))
   (assert (waiting barracks (+ ?nbr2 1)))
   (assert (resources crystal (- ?cry 50)))
   (assert (resources gas (- ?gas 0))))
CLIPS>
```

Figure 9: Example of rules - Build a marine



```
(defrule MAIN::marine
   (time ?t)
   ?id1 <- (queue buildMarine ?t)
   ?id2 <- (available marine ?nbr1)
   ?id3 <- (waiting barracks ?nbr2)
   ?id4 <- (available barracks ?nbr3)
   ?id5 <- (marine ?status)
   ?id6 <- (supplyMax ?supplyMax)
   =>
   (assert (outqueue buildMarine ?t))
   (retract ?id1 ?id2 ?id3 ?id4 ?id5 ?id6)
   (assert (marine 1))
   (assert (available marine (+ ?nbr1 1)))
   (assert (waiting barracks (- ?nbr2 1)))
   (assert (available barracks (+ ?nbr3 1)))
   (assert (supplyMax (+ ?supplyMax 0))))
CLIPS>
```

Figure 10: Example of rules - Add a marine