Using a Planner to Balance Real Time Strategy Video Game

Thierry Fayard

Centre National d'Etude Spatiale
18 av ed Belin 31400 Toulouse
Thierry.fayard@cnes.fr

Abstract

Real time strategy video games often include species able to fight each other. For the author to balance species parameters in order to make the game interesting can be a very difficult task, currently involving many players in long tests phases. In this paper we show how we can emulate average to good players with a planner based on simulated annealing algorithm. In such a game, and during the developing phase only, the players have to find the best sequence of activities to build the strongest army. Formally, it is maximizing an objective function within a given time. This planning problem is subject to temporal constraints and constraints for renewable, non renewable and cumulative resources. The cumulative resources are depleted or replenished over time depending on the player's choice. A general real-time strategy game model is presented. We choose Starcraft as an example where we compare our planner with human players. Then we discuss how our planner is able to help to balance species.

Introduction

To be able to balance species in a real time strategy (RTS) video game, we need first to be able to emulate very good players in such games. This paper describes a simulated annealing (SA) procedure to solve a resource constrained planning problem, as found in RTS. In the 1990's, computer capabilities made possible this kind of RTS video game, so called because time seems to flow continuously for the player, who has to manage a part of a simulated world, while the computer or another player controls another part of that world. 'Dune', 'Warcraft', 'Red Alert', 'Age of Empires', 'Command and conquer' and 'Starcraft' are some of the most well known RTS video games. In all of them the player has to choose one species, then to collect resources, build a big army and fight the enemy. Species can be very different from each other, but they must be similar in power, otherwise everybody will choose the most powerful one. In the early phase of such a game, each player gathers resources and develops an army, prior to any kind of interaction with other players. We have interest only in the developing phase and we don't consider any interaction between players in this paper. The player may execute a variable amount of activities, in an order chosen by him only, under resource and precedence constraints. This results in an army, along with some production facilities, which will determine the ability to achieve victory.

A practical application is developed to illustrate the efficiency of the SA algorithm and to compare the results with some of the best possible sequences of activities which experienced players already know. The comparison between what the software provides and what an experienced player does gives us confidence in our results.

This paper is organized as follows. First we detail the modeling of an RTS game and the given constraints in Section 2. We discuss implementing simulated annealing to optimize the objective function in Section 3. We present in Section 4 some key points of the software that we specifically developed for 'Starcraft', a very well known RTS game. Finally in Section 5 we show some results and explain how developers, can take advantage of such methods to balance their game. Section 6 is the concluding section of the paper.

A model for real-time strategy games

What is a real-time strategy game ?

In most real-time strategy games, competition based on resource gathering and killing enemies takes place on a two-dimensional map. On the map there are units, buildings and some collectable materials. Units and buildings may belong either to you or to your enemies, and you can control only your own. Most materials on the map are available for all, in fixed quantities or infinite, as long as you are able to collect them. They can be wood, food or mine, depending on the context of the game. A unit is an entity able to move on the map. It can be a soldier, a tank, an aircraft, a resource collector or any moving device. Units are mostly fighters : they are able to destroy other units or buildings. They can be specialized in ground attack (short - or long-range), air attack or resource

gathering. The buildings are used to produce units, upgrade units, allow materials storage, or develop specific capabilities which enable production of some kind. Materials are required to produce a unit, a building or an upgrade, and they must be available at the time you decide to produce it. They have to be collected constantly by the players. There are many options to win the game, but to destroy all enemy units and buildings is the most common. If you play against a human, you need an army at least about the same size as his one to achieve victory, and in addition, you also need to constantly produce new units to replace lost ones.

In short, we can distinguish two phases in such a game : a 'development phase' and a 'fighting phase'. Each player starts the game with some units and/or buildings, then he develops his army, this is the 'development phase'. When he feels strong enough, he starts to move his fighter units to engage the enemy, this is the 'fighting phase'. Although these two phases can be partly simultaneous, we only consider the 'development phase' as an independent whole in this paper. In practice, this limitation is of no consequence for the player for two reasons. First, most of the time there is no interaction at all between players in the beginning, because they start on distant places on the map. The time taken by a player to move his army from one place to another will be used by his adversary to produce a much stronger army, and this effect is particularly relevant at the beginning. Second, once the player has chosen a tactic, the set of activities to be planned doesn't depend on the decision taken by his adversary. The player has only to guess the most appropriate tactic with his limited knowledge of the opponent's intention. In brief we will, here, only consider a single-agent planning problem and omit the multi-agent theories.

The general planning problem in RTS

During the 'development phase', playing means executing one of the four type of activities and nothing else :

- sending a collector unit to materials
- building something
- producing units
- upgrading units.

Collecting materials is an automatic process which usually doesn't need the player's attention, as for other types of activities. Building something can start as soon as the required resources and the materials are available. The same situation applies for producing units and upgrades. It ends after a given time which may differ also due to the state of the game. For example, some upgrades may reduce the duration of building. In some games, one can use more than one unit to speed up the building process.

Any activity can be performed many times while resources are available and the dependence-feasible graph is respected. After the start, activities can run concurrently. Activities can be performed in alternative modes which differ with regard to processing time or resource requirements.

In the present context, a play is a sequence of activities (SOA) ordered by their starting dates. Then the planner has to find the best SOA which will optimize an objective function within a given time.

The resources

We will define three types of resources in a way which is convenient for this problem.

(i) The renewable resources : most units or buildings are renewable resources. For example a building can be used to produce units, when the process ends the building is available for another task. It is the same for units, when they end the construction of a building they are available for another task. Unlike classical RCPSP, Resource Constraint Project Scheduling Problem (Bouleimen, Lecocq 1998) the number of such resources is not fixed at all, and depend on the activities already processed by the player.

(ii) The non renewable resources : In most RTS games, there is only one resource of this type, which is used to limit the total population of units. For example, it can be the total number of farms you control which determines your maximum population limit. That resource is a non renewable one since, once it is affected, it cannot come in use for something else. Again here, the number of such resources is not fixed, and depends on the activities already processed by the player.

(iii) The resource reservoirs (also called cumulative resources) : In most real time strategy games, material has to be collected on the map. It can be gold, food, wood and so on. Sometimes, the total available amount is limited, for example a gold mine may come to an end. Sometimes it is not, for example, a forest may provide wood forever. We will consider that they are always available, which is the case at the start of the game. These materials are stocked in reservoirs, generally there is one reservoir for each type of material, but their capacity has no limit. These resource reservoirs are filled over time with materials at a speed which depends on the number of resources allocated by the player to that task. Materials are collected by all collector units, except when they have something else to do, such as building something. When more than one resource reservoir is present, the player can balance the number of units attributed to each one. A good assumption for the game model is to state that collectors are always collecting materials when they don't perform a different known activity. A special building is sometimes needed to be able to collect peculiar materials, for example a farm to collect food. To increase the collecting speed, you just need to create more collecting units. The collecting speed can also be improved if you upgrade some units, or if you develop

some special abilities, which of course comes at a cost. Sometimes the maximum collecting speed cannot exceed a threshold given by the game state. The key parameter for materials is the time needed to collect them. Resource reservoirs are depleted by most activities done by the player, such as producing a unit or a building.

Most activities need resources and materials in an amount which may vary, depending on the state of the game. All three type of resources, renewable, non-renewable and materials in resource reservoirs have to be available when an activity starts. As one can see, the production model for resources can be relatively complex and is specific to each game.

The dependence-feasible graph

In RTS game, it is not possible to perform any activity at any time. You have to follow a precedence constraint directed a-cyclic graph to construct buildings, units or upgrades. We will call such graph a dependence-feasible graph. For example, in the 'Starcraft' game, you can't build an airport if you don't have a factory, you need barracks to produce a soldier, and so on. Often you have more than one dependence link. In some cases, you can cluster two units to make a bigger one, or you can transform some units into buildings. Again, this makes the model quite complicated.

Time constraint

Each activity, such as constructing a building or producing a unit, takes some time to be completed. An activity cannot be stopped before the end, it can be cancelled in the game, but we won't consider that case.

Most of the time, to produce a unit, you need a specific building, which has to be available at the start of the activity. For example, to produce a marine you need barracks for 20 seconds, meanwhile the barracks cannot be used for something else. Since building can produce units only one by one, time is also a strong constraint. Buildings are also needed to perform upgrades, but they are usually effective for all units when finished.

The rule is that, to perform an activity, you have to wait for the resources to be available. An activity is thus feasible when :

· the dependence graph is respected, or will be respected when some previous activities come to an end
· needed renewable resources are ready, or will be ready when some previous activities comes to an end
· needed non renewable resource are available, or under production
· needed materials are available in resource reservoir, or at least one unit is able to collect them.

When an activity is feasible, it may take some time before it actually starts, because materials may have to be collected, a building under construction may have to be finished first, or a building needed is already doing something which has to be terminated first. Waiting for resources to be available introduces lags. This decreases the efficiency of the project, a good player will thus try to minimize delays.

The player goal

To win the game, you must eradicate your opponents, and this is the purpose of most units. Each has some hit points and some attack points. The more hit points it has, the tougher it is, the more attack points it has, the more hit points it can get from each enemy shot. Units may also have some defense points to reduce enemy attacks, some air attack points to attack air units, and so on. Some advanced units have special abilities which make them efficient, but these peculiarities may safely be ignored. We will not discuss all subtleties of these games in this paper and we will only consider that attack point is a good thing to maximize. The sum of attack points of all units is the army strength.

Considering that there are three ways to optimize a plan, or a sequence of activities (SOA) according to the player goal :
· The army strength is given and we want the SOA which allows to achieve this strength in the shortest possible time. Although it is a usual RCPSP, it is not a usual situation for the player.
· The number of activities is fixed and the objective function is the army strength. This is not a good choice, because it does not take into account the duration to achieve the SOA.
· The total duration is given and the objective function is the army strength obtainable within that duration. This is a usual situation for the player.

We choose that third option, because players frequently decide on a given duration without attack, it can be 5 to 20 minutes. A sequence optimized for 5 minutes will be very different from one optimized for 10 minutes, because in the latter you need to spend more on production equipments, and you don't have time for that in the first case. To take this issue into account, we simply evaluate the army strength obtained with all the activities in the sequence which end before the specified date. The objective function will be that army strength.

The use of the simulated annealing algorithm for planning optimization

Simulated annealing (SA) is a meta-heuristic which belongs to the local search algorithm class (Kirkpatrick, Gellat, Vecchi, 1983) (Aarts, Korst, 1989). We have chosen SA because it seems to be simpler to use than Genetic algorithm or taboo search, but it would be nice to try also these meta-heuristics. SA is an iterative process

with only one current solution and one neighborhood, in which we select the next solution. Solutions are a Sequence Of Activities (SOA) in our case. The sequence is a mere starting-time ordered activity list. The algorithm starts by evaluating an initial SOA, which is the first parameter to choose, and an SOA near the initial SOA called the neighbor. By "evaluate" we mean compute the objective function for that SOA with the game model. Then SA may replace the initial SOA by this neighbor with a probability which is a function of the evaluation difference and the “temperature”, or the time already spent in the search. Then SA looks again for a new neighbor and so on.

We have tuned the different SA parameters in the software developed for this type of game. Though this was made specifically for the 'Starcraft' game, these parameters should also be appropriate for similar games. The simulated annealing algorithm is described below :

The initial SOA

Choose the initial sequence of activities. Usually even a beginner has an idea of how to play, aberrant sequences can be avoided. But in fact, we have seen that even with a very bad start, the algorithm is able to find a good outcome in a medium size problem.

The cooling scheme

Choose the initial 'temperature' parameter To . Roughly a value of some tenths of a percent of the objective function of the initial state is appropriate. The descent profile of the temperature function A logarithmic descent should give the absolute optimum, but it is very slow. We implemented an adaptive descent profile. The temperature will decrease by a factor μ if, for a given number M of iterations, the new sequence is always accepted. The value μ is near and below 1. For example, 0.98 works well. M should be large enough, in our case one thousand to some thousands are good values.

$$T_{k+1} = \begin{cases} T_k & \text{if the number of consecutive accepted transitions} < M \\ \mu . T_k & \text{if the number of consecutive accepted transitions} = M \end{cases}$$

The probability of acceptance is given as follows :

$$0.5 - 0.5 \tanh [2m (c_k - c_{k+1} - T_{k+1})]$$

where c_k is the cost function of iteration k, and m is the slope. The slope m can be related to T in the following way : $m=m_o/T_k$, to give more significance to the slope for large values of T.

The stop criterion

There are two stop criteria. First, the process is stopped when the objective function doesn't make any significant progress for a given number N of iterations. Second, the process is stopped when the total number of iterations exceeds a given value.

The neighborhood-generation mechanism

There are many ways to choose the neighborhood definition. Some serve different purposes for the user. We describe here four of them :

(i) Two activities are randomly selected in the SOA and switched.

(ii) One activity is randomly selected in the SOA and replaced by a randomly selected activity among all available ones.

(iii) randomly choose one of the following moves : switch two randomly selected activities inside the sequence or, replace one randomly selected activity by another randomly selected one among all available activities. The probability of switching should be much higher than the one of replacing.

(iv) The same as the previous one, except that the replacement of an action is done by choosing a new activity inside the sequence.

Neighborhood (i) is usable when you know exactly what actions have to be selected. The initial SOA has to be appropriate for the aim. The result will be a better permutation. Neighborhood (ii) can give some results, but it is (iii) which gives by far the best results. Option (iv) needs also some knowledge from the user since the initial SOA has to contain all needed activities appropriate for the aim, only the number and the positions of each type of activity vary.

The objective function

As mentioned above, the objective function can be the plain sum of all attack points (or any desirable characteristic) of units or buildings obtained, after all activities have been performed, before a given date. The question arises of the SOA which are not feasible, and probably most of them are not feasible because of the structure of the graph of the precedence relation constraint. The first idea is to give a zero value to the function in that case, but it is not very efficient because too many SOA are rejected, and it may be difficult to get out of some local maxima. It is also possible to reduce the objective function by a weight depending on the number of violations. In fact, we got very good results by just skipping infeasible activities in the SOA, while evaluating the objective function. When too many activities are infeasible, the feasible activities are not able to give good results, in this way it reacts as a good weight. If you have only a few infeasible activities in a good SOA, this one can stay and then evolve favorably later. It was a great improvement of the algorithm. The slight drawback is that it increases the

number of activities in the SOA at the beginning of the search.

The computational experiment

The application was coded and compiled with the Builder C++. The tests were carried out on a PC AMD Athlon-3200+ with 2 Gigabytes memory under MS Windows XP. We preferred to program the software completely, rather than to use commercial tools. Nevertheless every parameters of the model are accessible in a single file. So, the costs, the durations of construction or production of unit, the tree of dependence are modifiable simply.

We chose to illustrate our optimizer with the game 'Starcraft', a Blizzard company title. This game was ranked at the top by some game magazines in years 1996-2000, because of its game play, its depth and its variety, among other reasons. Compared to those of other real time strategy game, the 'Starcraft' developing phase is very rich.

A straightforward way to compute the model is to follow time, as 'Starcraft' itself was computed. As time goes on you update the state of the game. It is easy and safe to compute such model. The problem is that it is very demanding in processing time. Especially when the time step is tiny, you spend all your time updating. The way we choose to compute the model is based on events. For each new activity, we find which events make it possible. This method seems to be more than twenty times faster than the method which follows time.

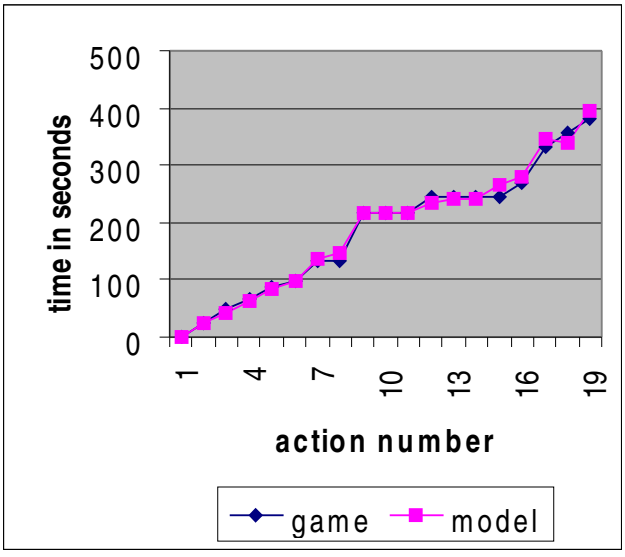

Figure I
The model fits the game well. Differences may be seen, just as between 2 human players, but there are no drift.

To simplify the model, the distribution of resource collectors is fixed at three for gas and the others are for mines, a typical situation, at the start of the game. The resource-production functions are mostly linear with time and with the number of resource collectors, and is weighted with some tabulated values. Non linear terms are to represent efficiency loss when the number of these units exceeds fifteen. Of course, when resource collectors are doing something else, such as building, this is taken into account.

Some model parameters, such as traveling duration, were tuned to fit a human player for various SOA. When the behavior of the model is compared to the actual game on a different set of SOA, one can find some small oscillating differences, but there is no long term drift, as we can see in Figure 1.

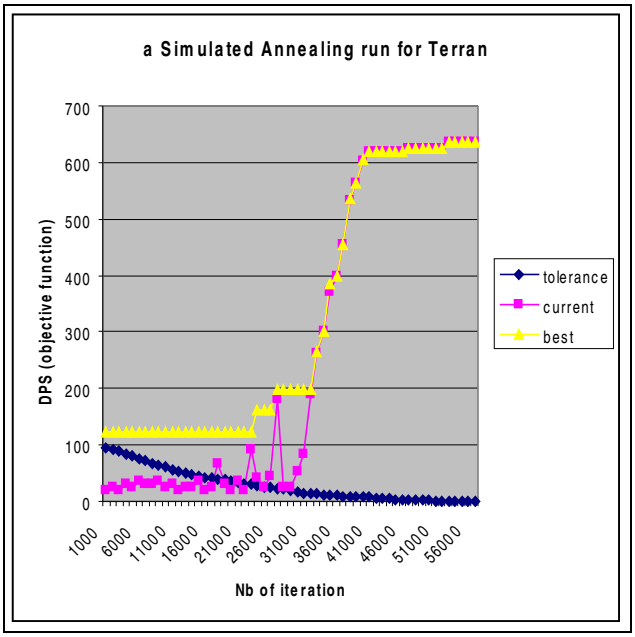

Figure II
Here one can see how DPS progress while temperature decrease with iteration process

The model validation versus human players

The general behavior of our planner is good with Terran species, average with Protoss and unsatisfactory with Zerg. Differences arise from the model, for example Protoss have less way to achieve good results since the number of units is lower at the beginning and the dependence's tree is longer. For the Zerg the non renewable use of a unit to build structure makes the problem more difficult. This is because that unit is used also to fill reservoir resource. It may happen that the reservoir is empty, so it is not possible to buy a unit able to fill the reservoir. This problem is easily solved by adding sometime that unit at the start of the sequence. But globally simulated annealing works fine as we will see. We have compared the obtained solutions

with our planner versus those given by an expert in Starcraft, and they match well.

We will separate analysis on short term solutions in which we plan about 30 activities and longer term solutions, with up to 130 activities.

For sequences of actions with a short duration, the planner converges to the best solution after a few trials. For short durations, the best solution is known because millions players tested it and it is available for the community on many internet sites. Such solutions are called rush play, because it is a way to win very quickly against a non prepared player, or a player who intends to prepare a long term attack. That's why players sometimes decide in advance to avoid rush.

For longer problems, differences may arise between experienced players and our planner. The solutions are not usually published. Mainly because after some time in the game, the key factor to win won't be only the development phase but also a good scouting of the enemy intention, to help the player in choosing a good tactic. Also some other factors (such as speed's player or agility) interfere, but this is out of our consideration. Anyway an expert player tried our solutions for longer problems and found them very satisfactory. We are not able to claim that we got the global optimum, but we can say that the best solution we found is a very good solution and even our expert is not able to find a better one. We then after compare the found solutions to our best known to determine the planner reliability.

Number of actions Project duration	27 actions 450 sec	78 actions 800 sec	106 actions 1200 sec
Problem size	10^{38}	10^{111}	10^{151}
Optimum value for ground attack	224	824	1348
number of runs over 95% of optimum	45%	10%	10%
number of runs over 90% of optimum	45%	13.3%	30%
Average value	180	656	1207
Standard deviation	49.5	99.5	69.0
Relative deviation	18.6%	15.1%	5.7%

Table I
Performances of our planner for Terran sequences of actions

As one can see, Table I shows that the reliability of the planner is very good for small projects. Indeed in one run you have 55% chances of being under 95% of optimum, but after 10 runs this probability drops to 3 over 100000 $(\#55/100)^{10}$. For medium-size or large problem the results are encouraging.

Surprisingly, the relative deviation decreases when the number of activities goes up. This is due to the game itself. You have more ways to achieve a good result when you have time to perform numerous activities. Consequently, the stochastic search finds its way near the optimum more easily.

Let's analyze the best run for a search optimizing a 1200-second ground attack. If you know enough about that game you can see that it found a very good solution with 3 marines, 15 firebats, 8 vultures and 31 tanks. The total strength is 1348, done in 106 actions. The buildings are one set of barracks and three factory shops. Vultures probably have the best quality-price ratio with 20 attack points for low cost 75 mines only, so one wonders why the solution also gives firebats. The answer is that, to create vultures, you need a factory which needs barracks. So, if you have barracks, it is good to produce firebats. One can also wonder why we get tanks. The answer is more subtle because the cost is heavy, 150 for mine, 100 for gas and 50 seconds for time, but we have to consider that supply building is another constraint in the game. Indeed, a vulture and a tank both consume a fourth of a supply which takes 100 mines and, above all, time to be built. Therefore, since gas production is due to the presence of factory, it is good to produce some tanks too. The play as usual starts with 4 resource collectors and a base. Then the beginning of the found sequence is as follows :

```
5 resource collectors
1 supply            // you need one supply for 8 units
1 refinery          // to get gas as soon as possible
5 resource collectors
1 barrack
1 resource collector
1 supply
2 resource collectors  // all of them are created as soon as
                       //possible
1 factory
1 resource collector
1 factory
1 factory shop      // for tank production
1 tank,…                    // then only it starts to produce
                    fighting //units along with supply
```

That sequence was tested with a human A, against an other good human player B, on a special map with no relief at all to avoid tactic influences. Player A had to follow the computer sequence of action. All players had to wait 1200 seconds before attacking in a unique massive battle. Player

B lost because his army strength was only 1284 compared to 1348 for player A, mainly because one factory shop was missing in his SOA.

Balancing the game's parameters

The game parameters are mainly : costs in resources both for buildings or units, production duration, damage per second (DPS) and/or hit points. The idea is that two species are balanced if the best found sequences of actions give similar DPS in a given time. To avoid singularities, in place of using only the best SOA, we use the average DPS of the sequences found in a given radius from the best one. For example if the planner give 1000 SOA, may be 50 will have a DPS at 90% of the best one, then we take the average DPS of these 50 SOA. We do the same for each species and we compare the average DPS.

We can compare Terran and Protoss production curves as shown in figures III and IV. They show obtained DPS with 20 runs sequences terminated at the time indicated below. Durations have only relative signification. The Protoss species seems to be more efficient, especially with best players. Actually it is not true since the game is well balanced. But this first approach doesn't consider the range which can be very large for Terran tanks. Units Protoss found by the planner are always hand-to-hand fighters (Zealot and Dark Templar) and they are more tough. Obviously, a different objective function have to be found to represent effectiveness.

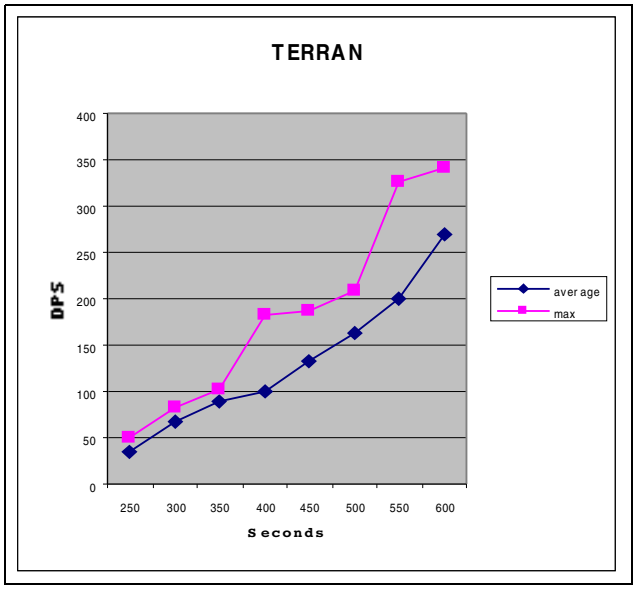

Figure III

The developing curve for Terran obtained from our planner for best and average simulated players. Two steps appears for bests due to firebat production and tanks production. Average players are also able to produce these units but later than best players.

But it is interesting to notice that the curves reflect reality quite well. For the Terran we can see that a good player is able to produce firebats or tanks about 50 seconds before an average player and it gives good advantage for a while. Protoss's curve tells that the gap between good and average players is large.

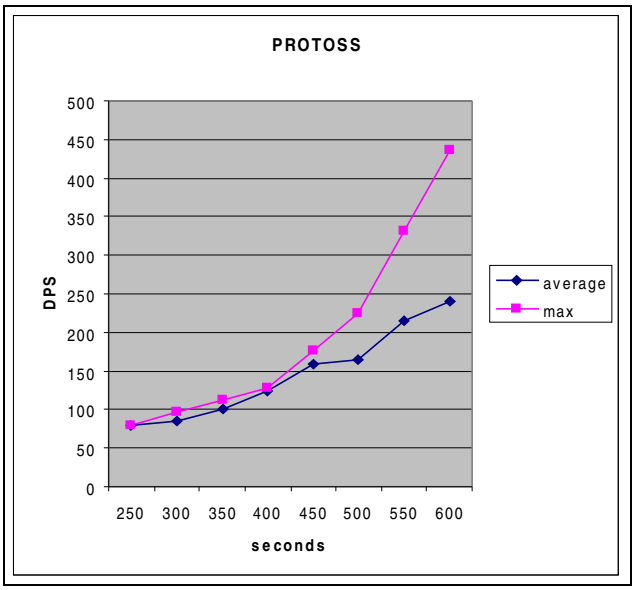

Figure IV

The developing curve for Protoss shows a steep end due to Dark Templar units which are available after 450 seconds and are very efficient. They have also stealth capability, but they are weak. One can see also that the gap between the best and the average player is much higher than Terran's one. May be this shows that our planner is able to find out that Protoss are more difficult to play than Terran, which is well admitted.

If we change manually some parameters such as production duration or costs in resources we can see well the result in the curves, and there is much to say. Some parameters, such as the cost of resources collectors, are very efficient for short duration sequences. And for long duration sequences the unit's cost are preponderant as expected.

So we can see that our planner is able to give back some known fact on Starcraft game. We suppose that it may be of some help for developers who want to simulate various kind of players quickly.

Conclusion

The 'Starcraft' planning problem seems to be an easy one, especially for Terran. May be the shape of the objective function is smooth near the optimum. Since 'Starcraft' is a game, the complexity of the problem is limited, this is

probably why the results are so encouraging. Genetic algorithms and tabu search should also be tested. We are quite confident that adaptation to other real-time strategy games will give similar results.

We have briefly shown that game designers may use such techniques to balance their games. Of course, human players will not be replaced by optimization software, but the latter can accelerate this part of the game development. Because if the game is not balanced in terms of production capability for different species, it will be hardly balanced at all. Of course, one have to consider more precisely cost function, plain DPS is not probably the best criteria. May be a proportion of hit points, defense points, or the range, has to be added to the DPS in that function. In addition to that, we can think of models able to include dynamic (speed) and/or geometric properties (range, zone effect…) for fight's simulation. But the problem description and the model will then be more delicate by far.

Some other extensions in our software can be anticipated. Giving the ability to free some parameters and fix others and make stochastic search at a double level, one for the sequence of action as we already do and one based on the gap between species. Probably, all these tracks can be usefully investigated.

References

Kirkpatrick, S. and Gelatt, C. D. and Vecchi, M. P. 1983. Optimization by Simulated Annealing, *Science* 4598, vol 220, 671-680.

Aarts, E.H.L. and Korst, J.H.M. 1989. Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimisation and Neural Computing. *Wiley*, Chichester

Bouleimen, K. and Lecocq, H. 1998. A new efficient simulated annealing algorithm for the RCPSP and its multiple mode version, in : *Proceedings of the 6th International Workshop on Project Management and Scheduling,* Istanbul.