Stochastic Search and Planning for Maximization of Resource Production in RTS Games

Thiago F. Naves and Carlos R. Lopes
Faculty of Computing, Federal University of Uberlândia, Uberlândia, Minas Gerais, Brasil

Abstract—*RTS games are an important field of research in Artificial Intelligence Planning. In these games we have to deal with features that represent challenges for Planning such as time constraints, numerical effects and actions with a large number of preconditions. RTS games are characterized by two important phases. The first one has to do with gathering resources and developing an army. In the second phase the resources produced in the first phase are used in battles against enemies. Thus, the first phase is vital for success in the game and the power of the army developed directly reflects in the chances of victory. Our research is focused on the first phase. In order to maximize resource production we developed algorithms based on Simulated Annealing and classical planning. As a result we have obtained a signification improvement of the strength of the army.*

Keywords: Real-Time Strategy Games, Resources, Actions, Goal, Search, Planning.

1. Introduction

Real-time strategy (RTS) games are one of the most popular categories of computer games. Titles like "Starcraft 2" and "World of Warcraft" are two representative examples of this category. These games have different character classes and resources, which are employed in battles. Battles can be waged by a human player or a computer.

It is possible to identify two phases in a RTS game. First, there is an initial period in which each player starts the game with some units and/or buildings and develops his army via resource production. In the next phase military campaigns take place. In this phase the resources gathered earlier are employed for offense and defense. Therefore, the stage of resource production is vital to succeed in this game.

The resources in RTS games are all kinds of raw materials, basic construction, military units and civilization. For obtaining a desired set of resources is necessary to carry out actions. In general we have three sorts of actions related to resources: actions that produce resources, actions that consume resources and actions that collect resources.

Once we conceive a sequence of actions (a plan) that produces a desired set of resources (a goal), we carry out this plan to take the game from an initial state of resources to another final state in which the desired set of resources is achieved. To identify what resources to be achieved (a goal) is an important step to elaborate a plan of action. The choice of an adequate goal has a direct impact on the strength of the army, which is important in the development phase. Thus, in the specification of a goal it is necessary to maximize the amount of resources to be achieved in order to raise the strength of the army. In this paper we describe our approach to achieve goals that maximize production of resources.

The domain used for this work is StarCraft, which is considered the game with the highest number of constraints on planning tasks [1]. In our approach we have to generate an initial plan of action to achieve an arbitrary amount of resources. In order to do this, we developed an sequential planner based on the STRIPS language [2]. The initial plan is the basis for exploring new plans in order to find the one that has largest amount of resources.

In this paper, we focus on the task of finding the goal that maximizes resource production based on an initial plan of action, as we stated earlier. Once we have an initial plan, we use Simulated Annealing [3] (SA) to maximize the amount of resources to be achieved. However, SA is not only used for goal specification. In the process of figuring out a goal specification, SA is also used to produce a plan to achieve that goal. Our work is motivated by the results obtained in [4] and by gaps in the approaches of [5] and [6] with respect to the choice of goals to be achieved during the production of resources. For the correct functioning of SA, techniques were introduced to adjust its parameters and validate the plans generated during the search. The results obtained are encouraging and show the efficiency of the method to find good solutions.

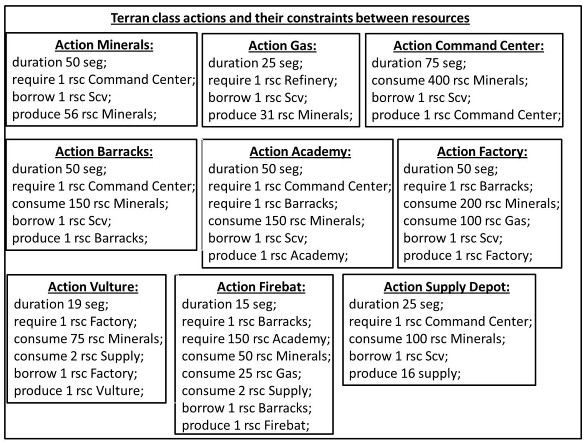
We understand that this research is interesting for the AI planning field because it deals with a number of challenging problems such as concurrent activities and real-time constraints. Most work in RTS games do not consider the choice of goals within the game. Usually goals for resource production have a random amount of resources to be achieved. Thus, this research can go toward a new approach to resource production, and can be extended to other applications.

The remainder of this paper is organized as follows. Section 2 lists the characteristics of the problem. In section 3 related work is presented and discussed. Section **??** explains about the domain of production resources. Section 4 describes techniques and algorithms for adjusting the parameters of SA in RTS games. Section 5 presents the consistency checker that works with SA. Section 6 discusses the results of the experiments. In section 7 we make final considerations about our approach.

2. Characterization of the problem

The planning problem in RTS games is to find a sequence of actions that leads the game to a goal state that achieves a certain amount of resources. This process must be efficient. In general, the search for efficiency is related to the time that was spent in the execution of the plan of actions (makespan). However, in our approach we seek for actions that increase the amount of resources that raise the army power of a player. This is achieved by introducing changes into a given plan of action in order to increase the resources to be produced, especially the military units, without violating the preconditions between the actions that will be used to build these resources.

To execute any action you must ensure that its predecessor resources are available. The predecessor word can be understood as a precondition to perform an action and the creation of the resource as a effect of its execution. In this paper actions and resources have the same name. To tell them apart we use the prefix *act* to indicate actions and the prefix *rsc* when referring to resources. Resources can be labeled in one of the following categories: $Require$, $Borrow$, $Produce$ and $Consume$. Figure 1 shows the precedence relationship among some of the main resources of the domain. These resources are based on StarCraft, which has three different character classes: Zerg, Protoss and Terran. This work focuses on the resources of the class Terran.



Terran class actions and their constraints between resources

Action Minerals:
duration 50 seg;
require 1 rsc Command Center;
borrow 1 rsc Scv;
produce 56 rsc Minerals;

Action Gas:
duration 25 seg;
require 1 rsc Refinery;
borrow 1 rsc Scv;
produce 31 rsc Minerals;

Action Command Center:
duration 75 seg;
consume 400 rsc Minerals;
borrow 1 rsc Scv;
produce 1 rsc Command Center;

Action Barracks:
duration 50 seg;
require 1 rsc Command Center;
consume 150 rsc Minerals;
borrow 1 rsc Scv;
produce 1 rsc Barracks;

Action Academy:
duration 50 seg;
require 1 rsc Command Center;
require 1 rsc Barracks;
consume 150 rsc Minerals;
borrow 1 rsc Scv;
produce 1 rsc Academy;

Action Factory:
duration 50 seg;
require 1 rsc Barracks;
consume 200 rsc Minerals;
consume 100 rsc Gas;
borrow 1 rsc Scv;
produce 1 rsc Factory;

Action Vulture:
duration 19 seg;
require 1 rsc Factory;
consume 75 rsc Minerals;
consume 2 rsc Supply;
borrow 1 rsc Factory;
produce 1 rsc Vulture;

Action Firebat:
duration 15 seg;
require 1 rsc Barracks;
require 150 rsc Academy;
consume 50 rsc Minerals;
consume 25 rsc Gas;
consume 2 rsc Supply;
borrow 1 rsc Barracks;
produce 1 rsc Firebat;

Action Supply Depot:
duration 25 seg;
require 1 rsc Command Center;
consume 100 rsc Minerals;
borrow 1 rsc Scv;
produce 16 supply;

Fig. 1: Some resources of the class Terran and their dependencies.

For example, according to Figure 1 to execute an action that produces a $Firebat$ resource you must have a $Barracks$ and $Academy$ available, a certain amount of $Minerals$ and Gas is also required. The Firebat is a military resource. Thus, the action $Firebat$: $Require$ *(1 rsc Academy, 1 rsc Barracks)*, $borrow$ *(1 rsc Barracks)*, $consume$ *(50 rsc Minerals, 25 rsc Gas)* and $produces$ *(1 rsc Firebat)*. The $Barracks$ will be $borrow$, i.e, it is not possible to perform another action until *act firebat* is finished. The time to build a $Firebat$ is 15 seconds. This situation describes the challenges that surround the planning of actions.

In our approach each plan of action has a time limit, army points, feasible and unfeasible actions. The time limit constrains the maximum value of makespan of the plan of action. The army points are used to evaluate the strength of the plan in relation to its military power. Each resource has a value that defines its ability to fight the opponent. Feasible actions in a plan are those actions that can be carried out, i.e, they have all their preconditions satisfied within the current planning. Unfeasible actions are those that can not be performed in the plan and are waiting for their preconditions to be satisfied at some stage of planning. Unfeasible actions do not consume resources of planning and do not contribute for the evaluation of the army points of the plan.

Returning to the example of the $Firebat$, suppose that a goal with a time limit of 700 seconds is *(1 rsc Firebat, 1 rsc Marine)*. A plan for this goal can be achieved by the following actions: *(10 act Minerals, 1 act Gas, 1 act Refinery, 1 act Barrack, 1 act Academy, 1 act Firebat, 1 act Marine)*. In a plan every action produces a resource with its respective name, i.e, *10 act Minerals* correspond to ten individual executions of action *Minerals*. The plan contains 28 army points and a makespan of 680 seconds. Among the possible operations that can be made in this plan, consider replacing an action $Minerals$ by a new action $Firebat$. In this case the action $Minerals$ contributes to make available the resource $Barracks$. In this way $Barracks$ becomes unfeasible within the plan due to lack of $Minerals$ and consequently the other resources such as $Academy$, $Firebat$ and $Marine$ will also be because $Barracks$ is a precondition for those resources.

With all the actions that became unfeasible the plan now has 0 army points. Now, suppose if instead of removing $Minerals$ it was requested to include a $Marine$ action in place of a Gas action. This exchange is feasible, because $Marine$ has just as preconditions *(1 rsc Minerals, 1 rsc Barracks)* and these are already available at the time. The action $Firebat$ becomes unfeasible and the duration of the plan (makespan) drops to 670 sec. If in the next operation an action Gas is inserted into the plan the action $Firebat$ action would become feasible. In this way a plan is established with 40 army points and 695 sec of makespan. The plan that is found is better than the initial one. Reducing the makespan of the goal is not one of the objectives of this work. However,due to the characteristics of changes in the plan this might happen.

Results, as the example presented above, are motivators for this research.

3. Related Work

There is little research in maximizing goals of resources production in RTS games. One of the reasons is the complexity involved in searching and managing the state space, which makes difficult to attend the real-time constraints.

The work developed by [4] remains our approach in a certain sense. He also uses Simulated Annealing to explore the state space of the StrarCraft in order to balance the different classes in the game by checking the similarity of plans in each class. In our work Simulated Annealing is used to determine a goal to be achieved that maximizes resources production, given the current state of resources in the game and a time limit for completion of actions.

[7] developed a linear planner to generate a plan of action given an initial state and a goal for production of resources, which is defined without the use of any explicit criterion. The generated plan is scheduled in order to reduce the makespan. Our work also makes use of a sequential planner. However, our approach deals with dynamic goals for resources production. This is necessary in order to find out a goal that maximizes the strength of the army. To the contrary, [7] works with a goal with fixed and unchangeable resources to be achieved.

Work developed by [6] has the same objective as the one pursued by [7]. These approaches differ on use of the planning and scheduling algorithms. [6] developed their approach using Partial Order Planning and SLA* for scheduling. [6] also works with a goal with fixed and unchangeable resources to be achieved and not can be adapted to our problem.

We also surveyed some existent planners that could be applied. [8] and [9] were considered for our problem at hand, but both have a different approach and would have to be modified to adapt to our goal. In short, most of the approaches surveyed have different focus and the techniques used are not efficient for the domain that we are exploring.

4. The use of Simulated Annealing to Maximize Resource Production

Simulated Annealing is a meta-heuristic which belongs to the class of local search algorithms [3]. SA was chosen due to the robustness and possibility to be combined with other techniques. it was also the algorithm that achieved the best results among those used during the experiments. SA takes as input a possible solution of the problem, in our case a plan of action developed by a sequential planner. The mechanism that generates neighbors will perform operations of exchange and replacement of actions in the plan. It should be noted that these operations generate new plans of action.

In SA a possible solution is evaluated through the use of an objective function. In our approach the objective function is responsible for counting the army points existent in each plan of action. In this way it is possible to find out which states have the highest attack strength. A state that is better evaluated that the current state becomes the current state. Even if the new plan generated is not chosen as the current solution there is still a probability of its acceptance. Algorithm 1 shows the pseudocode of the Simulated Annealing algorithm.

Algorithm 1 SA($G_{initial}$, P, M, α)

```
1: $G_{current} \leftarrow G_{initial}$
2: $T_0 \leftarrow InitialTemp()$
3: while $T_0 > 0$ do
4:     $i \leftarrow 1$
5:     $nSuccess \leftarrow 0$
6:     while $i < M$ do
7:         $G_{neighbor} \leftarrow NewNeighbor(G_{current})$
8:         $\Delta r \leftarrow Evaluation(G_{current})$ - $Evaluation(G_{neighbor})$
9:         if $\Delta r < 0$ or $Random() < PrAcceptance()$ then
10:            $G_{current} \leftarrow G_{neighbor}$
11:            $nSuccess \leftarrow nSuccess + 1$
12:        end if
13:        $i \leftarrow i + 1$
14:    end while
15:    if $nSuccess \geq P$ then
16:        $T_0 \leftarrow \alpha * T_0$
17:    end if
18: end while
19: return $G_{current}$
```

The parameters of SA were adjusted to the domain of RTS games. In the next subsections we describe in details the algorithm, techniques were developed and parameters used.

4.1 Initial Solution

The algorithm takes as input a initial solution (plan of action) and performs operations on its actions. The initial solution $G_{initial}$ is generated by the sequential planner (it will be briefly explained in the next subsection) and it is a plan with the actions ordered by their starting time. The resources that should be achieved in the initial plan are chosen randomly and the makespan is constrained to the time limit established for the goal. Time limits vary from 3 to 5 minutes. The idea is to make SA work with varied plans, which avoids that the algorithm becomes biased to a specific value of the objective function.

4.2 Sequential Planner

The planning algorithm takes a goal to be achieved a list of resources available , which contains all the resources available to the player as the amount of minerals or units, and a time limit for the execution of the plan to be generated The planning algorithm was built upon STRIPS[2]. In this way, based on the input data, a linear plan of actions is generated to achieve a goal.

Depending on the available time, given by the time limit parameter, the planner may be called more than once, each time with a random goal. The algorithm takes only one resource as a goal at a time because the idea is to build an initial plan of action without setting a fixed amount of resources. In each iteration a plan of action is achieved and appended to a previous plan obtained in early iterations. This is done until the time limit is reached by the resulting plan. The algorithm takes this behavior because it is necessary to build a plan of action without exceeding the time limit.

While a resource is being planned is possible to check whether the plan under development is not exceeding this time. If an action exceeds the time limit, the planning is stopped and the plan built so far is considered a valid plan and returned. Generate a plan based on a time limit is a way to build the plan based on the behavior of a human player, which does not specify an amount of resources to their goal, but builds as many resources as he can in a interval of time.

As an example suppose that it is necessary to produce a plan to achieve the resource *Marine* with time limit of 150 sec. based on the following available resources: (*(4 Scv, 50 Minerals)*. Our planner is capable of producing the following plan: (*(3 act Minerals, 1 act Barracks, 1 act Marine)*. Therefore, in order to produce *Marine*, it is necessary to execute three actions *Minerals*, one action *Barracks* and one action *Marine*.

4.3 The Cooling Scheme

The initial temperature T_0 ($InitialTemp()$, line 2) is achieved based on the average number of actions in the plan of action, which is given as input to the algorithm. With the time limit is possible to generate plans with a small number of actions. Our objective is to reproduce a typical behavior of a human player. This is achieved by establishing an initial temperature of about five times the quantity of actions in the plan. This is a appropriate value for the domain.

For cooling, a value close to a few tenths of a percent of the initial temperature is good. A logarithmic function can make the algorithm become very slow, which is an inappropriate behavior for an RTS game environment.

We developed an adaptive method of reduction, where the temperature T_{i+1} drops by a factor α (line 19) if for a given number of iterations P the new plans are always accepted. The alpha value is low and close to 1. For instance, 0.97 is a good value to be defined by alpha. P must be large enough to explore the possibilities of operations on the plans. Based on experiments we found that values between 15 and 20 represent a good choice for P.

$$T_{i+1} = \begin{cases} T_i & \text{if the number of plans accepted } < P. \\ \alpha . T_i & \text{if the number of plans accepted } = P. \end{cases}$$

The probability of a new plan to be accepted if it is not better evaluated than the current plan is given by the equation 1.

$$1 - 0.5 \text{ tanh } [2\Delta r(v_i - v_{i+1}T_{i+1}] \qquad (1)$$

Where v_i is the feasible number of actions that the plan has in the corresponding iteration. The variable Δr is a factor derivation and is obtained from the difference between the values of the objective function (army points) of the current plan and new plan generated (line 11 Algorithm 1). Finally this value is divided by temperature. Thus, the algorithm can behave differently depending on the temperature value.

The function responsible for calculating the probability is $PrAcceptance()$ (line 9). Its value must be greater than the value generated by the function $Random()$ (line 9), which produces a random value between 0 and 1. The parameter M determines how many neighbors will be generated before the temperature decreasing (line 6). Its value should enable a good search in the space of states given the temperature of the algorithm.

4.4 The mechanism that generates neighbors

There are several ways to generate a new neighbor plan $G_{neighbor}$ from the current solution $G_{current}$ in Simulated Annealing. This generation is the essential for the SA get a good conversion, which is the way the plan is modified to be compatible with the characteristics of RTS games. Next we describe four possible mechanisms for acomplishing this task.

A. Two actions are randomly selected in the plane and switched places, one assuming the position of other.

B. An action of the plan is randomly chosen and replaced by a new action that is that is randomly chosen among all available actions in the game.

C. One of the following moves is randomly chosen: Randomly select two actions within the plan and switch their positions, or replace a action chosen randomly within the plan by a new action randomly selected among all available actions in the game.

D. The same mechanism as above, but the new action that will replace the other will be randomly chosen among the available actions in the plan.

The mechanism A is useful when the actions of the plan are known. Thus, the initial plan must be developed with a high value in its objective function to get good permutations. This makes the algorithm with an tendentious implementation, a situation we want to avoid in this work. The mechanism B can return satisfactory results, due to the insertion of new actions in the plan, exploring well the possibility of new resources. But C is the mechanism that returns the best results through a combination of two exploration strategies. The mechanism D has the same problem as A. It needs quality guarantee in the initial plan to achieve good results.

When a neighbor plan is generated with function $NewNeighbor(G)$ (line 8), the consistency checker is used to manage and evaluate the changes made in the goal. The algorithm for checking consistency will be further discussed in the following sections.

5. Consistency Checker

In planning when changes are made in a plan of action it is necessary to see how these changes affect it, especially when the dependency relationship is as strong as in the case of StarCraft. A simple insertion of an action may result in the inviability of many others. However, this makes the plan suitable for the introduction of new actions. This relationship is given by the loss and addition of resources and needs to be checked effectively to not harm the planning. Therefore, we developed a checker capable of dealing with changes in a plan, verifying its consistency, ordering and managing the resources.

The consistency checker is used when SA calls the mechanism for generating neighbors. Algorithm 2 shows the pseudo-code of the checker. The algorithm takes the plan of action $Actions$, a list $NewActions$ with the new action to be inserted or two actions that will be switched, and the list of resources available R_{avaib}. Initially is checked whether any action becomes unfeasible due the operation to be carried out in the plan, either to replace one action or exchange actions of place. This checking is performed by function $checkPlan(Actions)$.

Unfeasible Actions are placed in the list $actionsUnf$. When at least one action is inserted into the $actionsUnf$, the algorithm can find the others which will also become unfeasible. This is done by going through the graph of precedence of the actions of the plan. If any of these has a resource marked as unfeasible in the predecessor list, it will become unfeasible as well.

The method $Update(Action, R_{avaib})$ (line 8 of algorithm 2) is used for check what resources will no longer exist and which will be available in the plan to be used by other actions, when the list of actions unfeasible insert some action. This means that if a resource *ScienceFacility* becomes unfeasible it does the same with *CovertOps* and *NuclearSilo*, but in return it releases 88 seconds and provides 250 *Minerals* and 300 *Gas*. These resources enable the introduction of new actions or make others that were unfeasible in the plan become feasible again.

The amount of resources that are released do not always covers the loss of actions that become unfeasible in the plan because these actions may be contributing directly to maximizing the objective function. To help the algorithm to make a decision a variable penalty is used. For each action that becomes unfeasible in the plan a μ value is added to it. Thus, the more unfeasible actions appear greater will be

Algorithm 2 Consistency($Actions$, $NewActions$, R_{avaib})

```
1: $actionsUnf \leftarrow checkPlan(NewActions)$
2: for all Action $Inf \in actionsUnf$ do
3:    for all Action $Act \in Actions$ do
4:       if $Act.Predecessors = Inf$ then
5:          $Act.feasible \leftarrow false$
6:          $actionsUnf.push(Act)$
7:          $penalty$ += $\mu$
8:          $Update(Act, R_{avaib})$
9:       end if
10:   end for
11: end for
12: $adaptPlan(NewActions, R_{avaib})$
13: $continue \leftarrow true$
14: $Actions\ Act$
15: while $continue = true$ do
16:    if $Act \leftarrow checkPred(actionsUnf, R_{avaib})$ then
17:       $Act.feasible \leftarrow true$
18:       $actionsUnf.erase(Act)$
19:       $penalty$ -= $\mu$
20:       $Update(Act, R_{avaib})$
21:    else
22:       $continue \leftarrow false$
23:    end if
24: end while
25: return $Actions$
```

the penalty on the plan when it is evaluated. The value of μ should be small and its value is based on the formula:

$$\mu = 0.1 \text{ - } n/2 \tag{2}$$

where n is the number of actions in the plan.

With unfeasible actions already established and available resources defined, the function $adaptPlan(NewActions, R_{avaib})$ (line 2) is called. This function is responsible for changing the current plan, either by insertion or switch of actions. This function also rearranges the scheduling of actions in the plan, due to the changes made by the operation performed.

In the last stage the consistency checker verifies that after the changes in the plan some unfeasible action can become feasible again. The algorithm goes through the list of unfeasible actions and for each one checks whether its predecessors are available. This is done by function $checkPred(Act, actionsUnf, R_{avaib})$ (line 16). When an action becomes feasible again it is removed from the list of unfeasible actions, the value of the penalty is decreased and the function $Update$ (line 20) is again used to update the resources available. If the function $checkPred$ is called and no action is feasible then the algorithm exits the loop. However, if the opposite happens the algorithm is repeated because an action that is now feasible can make others to become feasible.

Finally, the algorithm returns the new plan for SA that evaluates the objective function and decides whether it will be the new solution. The following is an application example of the consistency checker.

5.1 Example of application

To illustrate the behavior of the checker suppose we have a initial plan of action that achieve the resource *Firebat* as depicted in Figure 2. The initial state of the plan is (13, 0, 42, 6, 6, 16, 565). These plan values correspond to the number of actions, number of actions unfeasible, amount of minerals available, amount of gas available, supply used, army points and the makespan of the plan. The time limit is 600 sec.

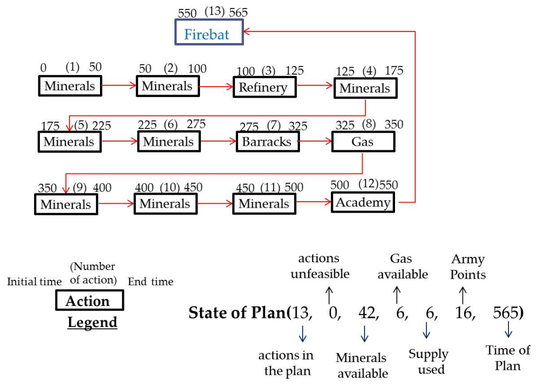


Fig. 2: An plan of action to achieve the resource Firebat.

For instance, suppose that in the first run of SA an action *Marine* substitutes the *Minerals* of number 10. Thus, the *act Minerals* is placed on the list of unfeasible actions and this action is the predecessor of *act Academy*, this also becomes unfeasible and with it the *act Firebat*. At this point the plan state is (13, 2, 186, 31, 4, 0, 465). The algorithm now inserts the new action and updates the resources it will consume. It is feasible based on available resources and the state of the plan is changed to (13, 2, 136, 31, 6, 12, 480). The algorithm cannot enable either of the two unfeasible actions, as the *act Marine* that entered does not contribute for these actions. Although there is enough *rsc Gas* and *rsc Minerals* for *act Firebat*, it may not be feasible as the *act Academy*, its predecessor continues unfeasible. The plan is then returned to SA that can opt for it, although it has fewer army points than the previous.

With the plan accepted suppose that SA indicates replacement of the action *Gas* of number 8 by another action *Marine*. *Gas* is the predecessor of *act Firebat*, but since this is already unfeasible the removal of *Gas* does not affect the plan. Now the plan state is (13, 3, 136, 0, 6, 12, 455). Although the *act Minerals* predecessor of *act Marine* are in the plan, it is in a forward position, i.e, the *act Marine* is being done prior to its predecessor. The new action goes unfeasible into the plan. No action becomes feasible in the next review and the final state of this plan is the same.

Now in SA suppose that occurs an switch of positions between the actions *Minerals* of number 11 and *Marine* of number 8. In this operation the actions remain feasible, not being necessary to check whether any action will be unfeasible. After the switching of actions the algorithm detects that *act Marine* has all its predecessors being executed before it and becomes feasible. The plan is now status (13, 2, 86, 0, 8, 24, 505) with 24 points army. The goal found is better than the original and the makespan is also reduced from 565 to 505. Although, reducing the makespan is not a goal yet explored in this work.

6. Experiments and Discussion of Results

The experiments were conducted on a computer intel core i7 1.6 GHz CPU with 4 GB of ram running on a windows operating system. The API Bwapi [10] allows to control Starcraft units and get information such as number of cycles and time of the game. It was used as an interface for the experiments with a human player and for performance evaluation of SA.

Table 1 contains comparison results. In the tests, we worked with different time limits for the makespan of the plans. We consider the planner as real-time because it can have smaller runtime (CPU time) than the makespan of the goals that it finds. In fact, this strategy was successfully used in the experiments, where the SA was able to find and maximize a future goal while the current goal was being executed in the game. The values that appear under the SA and the human player are the army points in the final plan of action obtained by both.

Table 1: Results of the test between SA and a human player

Time limit	Human player	SA	Makespan of SA	Runtime of SA
200 sec.	12	12	196	35 sec.
400 sec.	31	**34**	392	80 sec.
600 sec.	**52**	47	586	205 sec.
800 sec.	102	**109**	800	386 sec.
1000 sec.	141	**146**	990	555 sec.
1200 sec.	**204**	199	1189	696 sec.
1400 sec.	**241**	236	1388	812 sec.

Regarding the results SA proved to be able to compete with a human player, in some cases surpassing him. The most promising results were in the goals with medium time limit, where game tactics are not yet a determining factor for success in a direct confrontation, and production of more military resource can be crucial to the victory. The time limit imposed for the tests is high because the plans do not have their actions parallelized. This will be implemented in further work.

Table 2 shows the performance of SA. For goals with time limit medium the results are good, with 60% over 95% of

Table 2: Results of performance tests of SA

Time Limit	450 sec.	750 sec.	1050 sec.
Optimum value for army points	41	92	156
Number of runs over 95% of optimum	60%	40%	50%
Number of runs over 90% of optimum	30%	40%	40%
Average value	34	79	144
Number of Runs	10	10	10

optimal value. The best solution for plans with medium limit of time is known because many players share their rankings on the Internet informing these valuesâĂŃâĂŃ. For larger goals the results are encouraging. Although, with higher time limits (1050 sec.) the average of the results was better than with shorter (750 sec.). This happens because in larger plans is easier to validate actions due to the presence of more resources available.

The algorithm returns good results, but not always the optimal solution. This happens because in our approach, the SA was adapted to return a solution compatible with characteristics of real-time performance. StarCraft has a search space of exponential order, and SA running without the techniques that we have built could take hours to return the optimal solution.

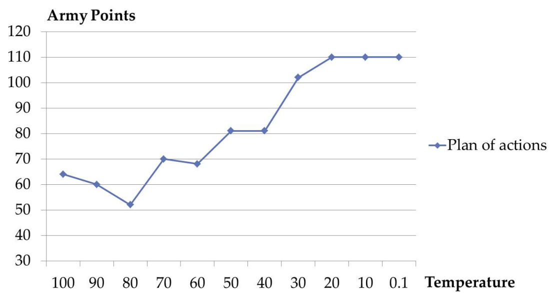


Fig. 3: Convergence of Simulated Annealing.

Figure 4 shows the convergence of the SA in a execution with time limit of 900 sec. At lower temperatures it accepts few new plans and keep those with the highest army points, a desired behavior for the SA. For this implementation the action plan given as input to the SA had 0 army points, demonstrating that even with bad entries the algorithm achieved good convergence.

7. Conclusion and Future Work

In this paper we propose the use of Simulated Annealing to maximize resource production in an RTS game. Research in this area is recent and the proposed approach is little explored by existing work. Our research is divided into two parts: The use of a sequential planner to generate one plan of action without fixed resources with time limit; and use of Simulated Annealing to maximize the resources of that plan of action by increasing the strength of its army.

Analyzing the results, the goals achieved always have an army points greater than the initial plan of action, which shows the convergence of the algorithm. The techniques developed to manage the planning contributed effectively in the quality of the results, and can be used in any domain of real-time games.

As future work we intend to explore interesting scheduling algorithms in order to reduce the makespan of the plans and achieve results with more quality. Also, We are investigating other features of StarCraft that can be used in the objective function evaluation. Based on our experience with Simulated Annealing, we also understand that other techniques such as bio-inspired algorithms may be explored.

References

[1] D. Churchil and M. Buro, "Build order optimization in starcraft," *AIIDE 2011: AI AND INTERACTIVE DIGITAL ENTERTAINMENT CONFERENCE*, 2011.

[2] R. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artif. Intell.*, vol. 2, no. 3/4, pp. 189–208, 1971.

[3] E. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimisation and Neural Computing.* John Wiley e Sons, Inc. New York, NY, USA 1989, 1989.

[4] T. Fayard, "Using a planner to balance real time strategy video game," *Workshop on Planning in Games, ICAPS 2007*, 2005.

[5] H. Chan, A. Fern, S. Ray, C. Ventura, and N. Wilson, "Extending online planning for resource production in real-time strategy games with search," *Workshop on Planning in Games ICAPS*, 2008.

[6] A. Branquinho, C. R. Lopes, and T. F. Naves, "Using search and learning for production of resources in rts games," *The 2011 International Conference on Artificial Intelligence*, 2011.

[7] H. Chan, A. Fern, S. Ray, N. Wilson, and C. Ventura, "Online planning for resource production in real-time strategy games," in *ICAPS*, 2007.

[8] M. B. Do and S. Kambhampati, "Sapa: A scalable multi-objective metric temporal planner," *Journal of AI Research 20:155Ű194*, 2003.

[9] A. Gerevini, A.; Saetti and I. Serina, "Planning through stochastic local search and temporal action graphs in lpg," *Journal of Artificial Intelligence Research 20:239-230*, 2003.

[10] Bwapi, *BWAPI - An API for interacting with Starcraft : Broodwar*, 2011. [Online]. Available: http://code.google.com/p/bwapi/