2012 IEEE International Conference on Systems, Man, and Cybernetics
October 14-17, 2012, COEX, Seoul, Korea

Maximization of the Resource Production in RTS Games through Stochastic Search and Planning

Thiago F. Naves, Carlos R. Lopes
(tfnaves@mestrado.ufu.br) (crlopes@facom.ufu.br)
Faculty of Computing
Federal University of Uberlândia
Avenue João Naves de Ávila 2121, Uberlândia (MG), Brazil, ZIP 38400-902

***Abstract*—RTS games are an important field of research in Artificial Intelligence Planning. These games have many challenges for planning. 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 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. This work focuses on the choice of goals to be achieved during the game. To do this, we developed an approach for maximization of production resources based on stochastic search and planning. The results show the effectiveness of our approach in finding goals that increase the strength of the player army**

***Index Terms*—Real-Time Strategy Games, Resources, Goals, Search, Planning.**

I. Introduction

Real-time strategy (RTS) games are one of the most popular categories of computer games. ”Starcraft 2” is a representative game title 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 it 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 are 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 [6]. 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 [9]. 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 [1] (SA) to maximize the amount of resources to be achieved. However, our approach is not only used to specify the goal. We developed a consistency checker capable of handling and and validating the changes made in the plan. Thus, our algorithm also produces a plan that achieves the goal found during the search. Our work is motivated by the results obtained in [8] and by gaps in the approaches of [4] and [2] with respect to the choice of goals to be achieved during the production of resources. 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 works 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 II lists the characteristics of the problem. In section III related work is presented and discussed. Section IV shows the sequential planner, which is an important component in our approach. Section V presents the consistency checker that works with SA. Section VI discusses the results of the experiments. In section VII we make final considerations about our approach.

II. 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

978-1-4673-1714-6/12/$31.00 ©2012 IEEE

certain amount of resources. This process must be efficient. In general, the search for efficiency is related to the time that is 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.

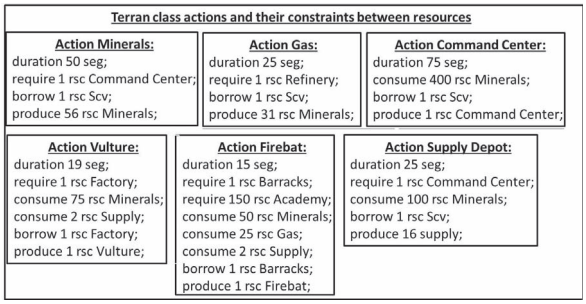

Fig. 1. Some resources of the Terran class and their dependencies.

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 $Firebat$ involves: $Require$ *(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 (sec). 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 planning resources and do not contribute for the evaluation of the army points of the plan.

Returning to the $Firebat$ example, 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 $Minerals$ with a new $Firebat$ action. 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 unfeasible 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 $Marine$ just has 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 in the example presented above, are motivators for this research.

III. 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 it difficult to attend the real-time constraints.

The work developed by [8] remains as our approach in a certain sense. It 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 the completion of actions.

[5] 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 discover a goal that maximizes

the strength of the army. To the contrary, [5] works with a goal with fixed and unchangeable resources to be achieved.

Work developed by [2] has the same objective as the one pursued by [5]. These approaches differ in the use of the planning and scheduling algorithms. [2] developed their approach using Partial Order Planning and SLA* for scheduling. [2] also works with a goal with fixed and unchangeable resources to be achieved and can not be adapted to our problem.

We also surveyed some existent planners that could be applied. [7] and [10] were considered for our problem in 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 focuses and the techniques used are not efficient for the domain that we are exploring.

IV. Planning Architecture

In this section, the sequential planner will be described, which is responsible for generating an initial plan to be passed on to the SA algorithm.

The Sequential Planner is responsible for developing plans of actions that lead the game of a certain resource state to another with more resources. Algorithm 1 shows a version of our planner. Our approach does not work with a goal that has a fixed and unchanging number of resources, but with a plan of developed action based on a time limit for the execution of their actions in the game. The planner was developed based on means-ends analysis, a search technique used in the STRIPS planner [9].

Algorithm 1 Sequential Planner(R_{goal}, R_{avaib}, T_{limit})

```
1:  Plan ← Extract(R_goal)
2:  for each Predecessor R_predecessor ∈ Plan do
3:    if Predecessors(R_predecessor, R_avaib) then
4:      R_predecessor.exist ← true
5:      Plan.push(R_predecessor)
6:    else
7:      Plan.push(R_predecessor)
8:    end if
9:  end for
10: for each Action Act ∈ Plan do
11:   if Act.bTime + actuaTime <= T_limit then
12:     if Act.exist = false then
13:       checkingLegality(act, R_avaib)
14:       Update(R_avaib, Act)
15:     else
16:       checkingLegality(act, R_avaib)
17:     end if
18:   else
19:     Exit()
20:   end if
21: end for
22: return Plan
```

The algorithm takes a goal to be achieved R_{goal}, which is a unique resource that will be produced, a list of resources available R_{avaib}, which contains all the resources available to the player such as the amount of minerals or units, and gets the time limit for the plan T_{limit}. The algorithm takes only one resource at a time because the idea is to build an initial plan of action without setting a fixed amount of resources and that it can be a basis for exploration of new plans.

In the first step of the algorithm the function $Extract(R_{goal})$ takes the resource to be developed and puts its respective production action in the plan and then a search among the resource predecessors of the initial action is made. Figure 2 shows the order in which the predecessors of an action are visited and placed in the plan. When a predecessor already exists between resources available ($Predecessors(R_{predecessor}, R_{avaib})$), its corresponding action is added to plan $Plan$ and marked as true. This means that the resource already exists. Otherwise, its production is necessary.

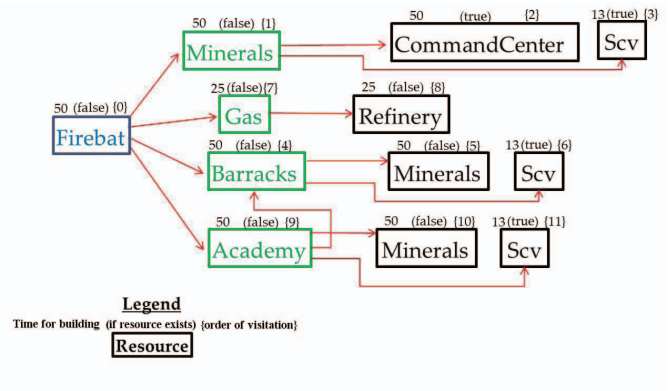

Fig. 2. Precedence graph of the resource Firebat.

In the second part of the algorithm the validation of actions occurs in the plan that was built in the previous step. An action Act consists of an tuple $(b, i, e, c, Predecessors)$ where b is the time it takes for the action to be built, i is the initial time when it began to be built, and e is the end time of the action when it is built, c is the cost of action between minerals, gas and supply and $Predecessors$ is the list of predecessor resources of the action.

Initially, the validation of an action is done by checking if the time it takes to be produced will not exceed the time limit (line 11). $ActualTime$ is a variable that measures the actual time of planning based on current actions that are in production or already produced. The method $checkingLegality$ is responsible for verifying the legality of an action by checking that the predecessors are available, updating the time of the game and setting the attributes of the action (its starting and ending time). Even if the resource already exists, the method is called, because it is necessary to verify the legality of the action and set the parameters of the resource.

The method $Update(R_{avaib}, Action)$ updates the list of resources available with the action that is created in the current iteration (line 12). If the resource already exists the method is not called because this resource will not consume $Minerals$ or Gas and it is already set. If the action exceeds the time

limit established to plan the method $Exit()$ stops planning and returns the current plan.

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 at a time as a goal 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 T_{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.

In this article emphasis is not given to the development of SA. The focus in this work is the construction of the planner and its consistency checker. The latter is responsible for validating all the operations that SA makes in the plan and ensures that at the end of the algorithm the plan of actions to achieve the goal is valid. Thus, our approach can give full support to an agent that controls the game or assistance to a human player during the game.

V. 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 in order not to 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 pseudocode of the checker. There are several ways to generate a new neighbor plan in Simulated Annealing. The way the plan is modified needs to be compatible with the characteristics of RTS games. The mechanism of operation used for this task is described next. One of the following moves is randomly chosen: Randomly select two actions within the plan and switch their positions, or replace an action chosen randomly within the plan by a new action randomly selected among all available actions in the game. This mechanism returns good results through a combination of two exploration strategies.

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 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 is checked. 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

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

```
1:  actionsUnf ← checkPlan(NewActions)
2:  for all Action Inf ∈ actionsUnf do
3:    for all Action Act ∈ Actions do
4:      if Act.Predecessors = Inf then
5:        Act.feasible ← false
6:        penalty += μ
7:        Update(Act, R_avaib)
8:      end if
9:    end for
10: end for
11: adaptPlan(NewActions, R_avaib)
12: continue ← true
13: Actions Act
14: while continue = true do
15:   if Act ← checkPred(actionsUnf, R_avaib) then
16:     Act.feasible ← true
17:     penalty -= μ
18:     Update(Act, R_avaib)
19:   else
20:     continue ← false
21:   end if
22: end while
23: return Actions
```

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 to 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 unfeasible actions receives 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 cover 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 (line 6) value is added to it. Thus, the more unfeasible actions appear, greater will be the penalty on the plan when it is evaluated. The value of μ should be small and its value is based on the formula:

$$\mu = 0.1 \text{ - } n/2 \tag{1}$$

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 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.

A. Example of application

To illustrate the behavior of the checker suppose we have a initial plan of action that achieves the resource *Firebat* as depicted in Figure 3. 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.

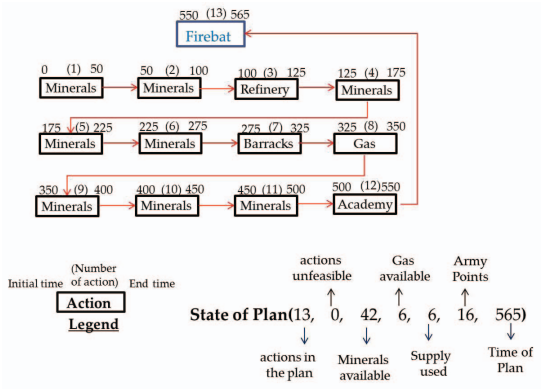

Fig. 3. A plan of action to achieve the resource Firebat.

For instance, suppose that in the first run of SA an action *Marine* substitutes the action *Minerals* of number 10. Thus, the *act Minerals* is placed on the list of unfeasible actions and as 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. The action *Marine* is feasible based on available resources. Consequently, 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 to become feasible. Although there is enough *rsc Gas* and *rsc Minerals*, the *act Firebat* is not feasible because the *act Academy* (its predecessor) remains 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* is 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 into the plan as unfeasible . No action becomes feasible in the next review and the final state of this plan remains the same.

Now in SA suppose that a switch of positions occurs between the actions *Minerals* of number 11 and *Marine* of number 8. In this operation the actions remain feasible. It is not necessary to check whether any action will become 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 now has status (13, 2, 86, 0, 8, 24, 505) with 24 army points. The goal found is better than the original and the makespan is also reduced from 565 to 505. Reducing makespan is not a goal to be explored in our work yet.

VI. 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 [3] allows the control of Starcraft units and get information such as the 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.

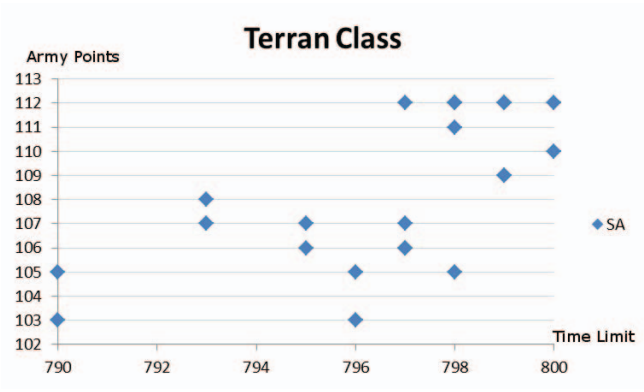

Fig. 4. Results of Simulated Annealing in different executions.

Figure 4 shows the results obtained in 20 runs of SA along with the consistency checker. Different plans generated by the sequential planner were used for each run. The time limit (makespan) imposed for plans in this experiment was 800 sec, and the best result with this limit was a plan with 112 army points. The best solution for plans with medium time limit is known because many players share their rankings on the Internet. We also collect these values through tests. The

algorithm was able to find goals with the best result in some of the runs. In others it was able to achieve plans close to the best result.

Table I 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 a 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 algorithm 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 I
RESULTS OF THE TEST BETWEEN SA AND A HUMAN PLAYER

Time limit	Human player	SA	Makespan of SA	Runtime of SA
100 sec.	8	8	97	21 sec.
250 sec.	21	21	248	44 sec.
450 sec.	35	**39**	450	89 sec.
600 sec.	**54**	51	597	202 sec.
750 sec.	**90**	85	749	341 sec.
900 sec.	124	**130**	896	483 sec.
1050 sec.	170	**172**	1048	585 sec.
1300 sec.	**224**	221	1300	735 sec.
1450 sec.	**263**	258	1449	832 sec.

Regarding the results, our approach proved to be able to compete with a human player, in some cases surpassing him. 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 II
RESULTS OF PERFORMANCE TESTS OF SA

Time Limit	300 sec.	650 sec.	950 sec.
Optimum value for army points	27	68	139
Number of runs over 95% of optimum	68%	46%	56%
Number of runs over 90% of optimum	32%	48%	43%
Average value	25	63	135
Average CPU runtime	58	225	502
Number of Runs	15	15	15

Table II shows the performance of our approach. The results are good, two bands of the time limit with more than 50% of executions are close to the best result.We obtained several goals close to 95% better result, and the runtime of the algorithm was maintained an expected average.

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.

VII. 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 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 to the quality of the results, and can be used in any domain of real-time games.

As a 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 might be explored.

ACKNOWLEDGMENTS

This research is supported in part by Research Foundation of the State of Minas Gerais (FAPEMIG) and Faculty of Computing (FACOM) from Federal University of Uberlândia.

REFERENCES

[1] 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.

[2] 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.

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

[4] 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.

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

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

[7] M. B. Do and S. Kambhampati, "Sapa: A scalable multi-objective metric temporal planner," *Journal of AI Research 20:155194*, 2003.

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

[9] 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.

[10] 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.