Resource Production in StarCraft Based on Local Search and Planning

Thiago F. Naves(✉) and Carlos R. Lopes

Faculty of Computing, Federal University of Uberlândia, Uberlândia, Brazil
{tfnaves,crlopes}@ufu.br

Abstract. This paper describes a approach for resource production in real-time strategy games (RTS Games). RTS games is a research area that presents interesting challenges for planning of concurrent actions, satisfaction of preconditions and resource management. Rather than working with fixed goals for resource production, we aim at achieving goals that maximize real-time resource production. The approach uses the Simulated Annealing (SA) algorithm as a search tool for deriving resource production goals. The authors have also developed a planning system that works in conjunction with SA to operate properly in a real-time environment. Analysis of performance compared to human and bot players corroborate in confirming the efficiency of our approach and the claims we have made.

Keywords: Real-time strategy games · Actions · Resources · Planning · Search

1 Introduction

Determining which resources should be built into a game is one of the main challenges in RTS Games. Along with the choice of resources, it is necessary to automate the planning of actions that build resources, satisfy constraints between actions and manage parallelism of actions. All these planning issues must be made under uncertainty, since the opponent is hidden over fog of war. These features are commonly found in problems that are relevant to the artificial intelligence planning area. Based on this information, we have explored ideas from the automated planning area for determining resource production goals as well as a plan of actions that achieves them.

The resources in RTS games originate from many sources of raw materials, basic construction, military units and civilization. The resources are used in attack and defense against enemies. In fact, there is an initial stage of the game in which players spend time aiming at developing the maximum amount of resources. During this time there are no direct confrontations and players who achieve better quality in resource production have advantage in the confrontation phase. Thus, resource production is a vital task for success in the game.

We have developed an approach that maximizes resource production based on goals established at runtime. Subsequently, it is possible to determine a plan of actions and their scheduling that leads the game from an initial resource state to a goal state that contains the new resources produced to improve the army. The plan of actions contains all the necessary actions to achieve the goal. A goal is described by the resources that this can achieve. Actually, as it is described later, the plan of actions by itself describes the resources or goals that should be produced in order to maximize the armed force of a player.

For this, the Simulated Annealing algorithm (SA) [1] is used along with planners and checkers that have been developed so that the approach is able to operate within the RTS games domain. The SA performs a search process, where an initial plan of actions goes through several changes in order to maximize the quality of the resource production and increase the force of the army that will be generated.

The domain used in our work is StarCraft, which is considered a game with a high number of constraints in planning tasks [6]. The work has been developed to solve a common deficiency in research involving RTS games, which is the choice of which resource production goals to strive for. For instance, [2,5,6] present planning and scheduling algorithms to determine which actions should be carried out to reach a resource production goal from an initial resource state. However, the establishment of a goal is not based on any explicit criterion that assures quality in its planning. Most probably the specification of another goal might bring better results. Different from previous work our approach does not take into account a fixed goal for resource production.

In our approach an initial plan of actions is generated. In order to reach this, a planner system called POPlan has been developed, which is able to plan and schedule the necessary actions to achieve an initial amount of resources, which may change at runtime. The initial plan is the basis for exploring new plans in order to find the one that has the greatest number of resources. In conjunction, a consistency checker called SHELRChecker was developed. SHELRChecker validates and manages the changes made during the search for new plans of actions. By doing this we claim that action planning based on a fixed goal is not necessary, but what is important is a specification of a policy that maximizes resource production.

The algorithms shown in this paper can also be used to build tools to assist players, such as an interface to help in choosing a goal and which resources should be built at given moments of the match. This work is a result of our efforts in developing action planning algorithms for resource production in RTS games. In this paper we focus on describing the planning system. Results of experiments and performance compared to human players and bot players that compete in StarCraft competitions corroborate in confirming the efficiency of our approach and the claims made in this work.

The remainder of the paper is organized as follows. Section 2 presents the characterization of the problem. Section 3 describes related work. In Sect. 4 there is a brief description of the use of SA in the approach proposed here. The description of POPlan and its operation is in Sect. 5. One finds in Sect. 6 a description of the consistency checker SHELRChecker. The experiments and discussion of the results are found in Sect. 7 and finally the conclusion in Sect. 8.

2 Characterization of the Problem

RTS games are strategy games in which military decisions and actions occur in a real-time environment [3]. 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 an effect of its execution. Resources can be placed into one of the following categories: *Require*, *Borrow*, *Produce* and *Consume*. The resources are based on StarCraft game, which has three different character classes: Zerg, Protoss and Terran. This work focuses on the resources of the class Terran.

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 is spent in the execution of the plan of action (makespan). However, in our approach we seek 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.

In our approach each plan of action has a time limit, army points, feasible and unfeasible actions. The time limit is the deadline of the plan of action, i.e., the maximum makespan value that a plan has to perform their actions. The time limit works as a due date for any plan of action, which cannot be exceeded. 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. The sum of this value for each resource present in a plan corresponds to its army points. 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 cannot be performed in the plan and are waiting for its preconditions to be satisfied at some stage of planning. Unfeasible actions do not consume planning resources and do not contribute to the evaluation of the army points of the plan.

As an example of operations in a plan, suppose that a time limit of 170 s was imposed to find a goal and the initial plan was set to perform *1 Firebat*. In this case, the completed plan of action has the following actions: *(8 collect-minerals, 1 collect-gas, 1 build-refinery, 1 build-barrack, 1 build-academy, 1 build-firebat)* with makespan equals to 160 and 16 army points. There are several operations that could be made in this plan and that would leave unfeasible actions. For example, exchange one of the actions *collect-minerals* that is a precondition of *Barracks* with any other action would leave the resource unfeasible. In consequence the resources *Firebat* and *Academy* would also be unfeasible, since these have *Barracks* as one of their preconditions. In this way, the plan would have 0 army points.

Some specific combinations of operations can make the plan increase its resource production and consequently its army strength. For example, suppose that a *build-marine* action is inserted in place of the *collect-gas* action. The new action is feasible in the plan, because it uses the *Minerals* left by the action *build-firebat*, which became unfeasible due to lack of *Gas*. Now, if in the next two operations the actions of *collect-minerals* and *collect-gas* are inserted in the plan, the resource *Firebat* once again becomes feasible. This happens because the action of minerals would be executed by a *Scv* resource which is idle and the scheduling process allocates it as early as possible in the plan. The gas action would be performed at the same time as a result of the scheduling process. Thus, the final goal now has 28 army points without exceeding the time limit, still having the same makespan of 160 s. The state of resources when starting a match in StarCraft is *4 Scv, 1 CommandCenter*.

3 Related Work

There is little research in maximizing goals of resource production in RTS games. Some of the reasons are: the complexity involved in searching and managing the state space, planning and decision making under uncertainty and the resource management under real-time constraints.

Our approach is based on the work developed by [8]. It also uses SA to explore the state space of the StrarCraft. But in this case to balance the different classes in the game by checking the similarity of plans in the each class. In our work Simulated Annealing is used to determine a goal to be achieved that maximizes resource production, given the current state of resources in the game and a time limit for the completion of actions.

In [5] a linear planner was developed 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 now uses a planner system with scheduling. Our approach deals with dynamic goals for resource production. This is necessary in order to figure out 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 that pursued by [5]. Their 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 cannot be adapted to our problem.

The work of [6] presents an approach based on the FF [11] algorithm which uses enhanced hill-climbing and a delete-relaxation based heuristic on to develop a plan of action that achieves a specific goal in the shortest possible time (minimum makespan). The goals to be used are expert goals, recovered from a database created from analyses of games already played. In this approach, heuristics and efficiency strategies to reduce the search space and achieve better results. In this work the amount of goals and the number of produced resources are limited those goals that are at the database, at some point of the game a more appropriate goal maybe cannot be considered, by not being at the database or not produce the correct amount of resources. The approach proposed in this paper aims to select and build goals through plans of actions without resource constraints and various possibilities of productions.

The works of [14,18] uses an approach with Case Based Reasoning (CBR) and Goal Driven Autonomy (GDA). CBR is used to select expert goals from a database and GDA allows a goal to be discarded during its execution and a new goal is chosen that takes into account the game situation. This approach has similarities with that proposed in this article, especially in relation to planning goals within the game, however this approach also has a limit on the number of goals that are considered, which are present in a database.

The work of [12] also uses expert goals to select a goal in RTS games. In this, an analysis is made on already played matches and goals are scored according to characteristics of current state of the game, such as, the amount of remaining units and amount of downed enemies. To select a goal within the game is used genetic algorithm (GA), which is responsible for selecting a goal according to its score and characteristics that better satisfy the current objective of the player. The strategy of choosing goals from characteristics of the state of the game is interesting, but the approach proposed here considers a wider range of possible goals and seeks to find this without the help of prior information or expert goals.

We also surveyed some already existing planners that could be applied. Approaches described in [7,10,13] 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 objectives and the techniques used are not efficient for the domain that we are exploring.

4 Simulated Annealing in the Maximization of Resource Production

Simulated Annealing is a meta-heuristic which belongs to the class of local search algorithms [1]. SA was chosen due to the robustness and possibility of it being combined with other techniques. It was also the algorithm that achieved the best results during the initial tests that were completed to develop this approach. SA takes as input a possible solution to the problem, in our case a plan of action that was developed by POPlan, which is going to be described in the next section. 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. The following will briefly describe some of the main features of the algorithm and the adjustments made for use in the RTS games environment. More details of the functioning of the SA are available in [16].

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 that exist in each plan of action. Another attribute present in resources could be used to evaluate the plan, but thinking about the game as a whole the attack force is a good value to be maximized. In this way it is possible to find out which states have the highest attack strength. A state that has a better evaluation than 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.

There are several ways to generate a new neighbor plan from the current solution in SA. The operation used in the approach is the following: we choose randomly between select two actions within the plan and switch their positions, or replace an action for another randomly chosen. This mechanism yields nice results. When a neighbor plan is generated, algorithm SHELRChecker is used to manage and evaluate the changes made in the goal.

5 POPlan

The POPlan is a system that uses the concept of partial order planning (POP) [15] to generate a plan of action and perform the scheduling task during this process. Two reasons explain why a partial order planner is required. The first reason is the fact that SA need a plan as input data that enables the search for solutions with parallelism of actions. Another reason has to do with finding an integrated solution that reduces time during the process of planning and scheduling.

The system architecture is composed of two levels. At one level we have the planner component responsible for plan generation. At another level we have the scheduling of the actions that make the generated plan. These two levels interweave their activities and are depicted in Fig. 1. With this architecture, the system has the concept of strong coupling [9] where planning and scheduling problems are reduced to a uniform representation. In Fig. 1 the domain element represents the environment of RTS games where the planning is conducted, the restrictions represent the preconditions and precedences of the actions of a specific RTS game and the processes of planning and scheduling use the previous information in their executions. The restrictions satisfaction process is made defining the actions and causal links between them and the resources available, which are used to build a certain resource production goal. During the scheduling it is checked if any of the links established in the previous process became invalid, i.e., if any link has become inconsistent. For example, if during the process of creating links an action has to have a link to a $Minerals$ resource, it means that this action will use it at a certain time. However, if this $Minerals$ is used by another action that will be scheduled first it may cause inconsistency. It is necessary to reconstruct this link with other $Minerals$ resource available.

With the POPlan architecture if any inconsistency is found in the scheduling step it is possible to check and satisfy it in the planning stage that is being performed intercalated with that. Thus, it is not necessary to interrupt the process for such a verification. In this coupling, the planner is essential for successfully achieving this representation, because the causal links defined between the actions in the planning stage have the temporal order of constraints among the actions, which also assists in the production and use of resources. With that, the scheduling step only figures the best time for the action to be performed, leaving planning tasks to be used in the next stage.

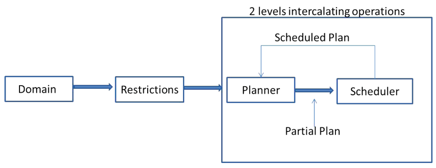

Fig. 1. Representation of the strong coupling intercalating POP and scheduling.

Algorithm 1. POPlan($Plan$, R_{goal}, T_{limit}))

```
1: $Plan \leftarrow LinkBuild(Plan, R_{goal})$
2: for each Action $Act \in Plan$ do
3:   if $Act.unity = false$ then
4:     $times \leftarrow Quantity(Act)$
5:   end if
6:   while $i < times$ do
7:     for each Action $ActRs \in Plan$ do
8:       if $ActRs = Act.rbase$ then
9:         $Schedule(Plan, ActRs, Act)$
10:      end if
11:    end for
12:    $contr \leftarrow Constructs(Plan, Act)$
13:    if $contr = false$ then
14:      $Exit()$
15:    end if
16:    $i \leftarrow i + 1$
17:  end while
18: end for
19: return $Plan$
```

Algorithm 1 describes the pseudo-code of the POPlan.

The POPlan receives as parameters the list where the actions that will make the plan will be inserted ($Plan$), the resource that is the current goal (R_{goal}) and the time limit for completion of the plan of action T_{limit}. At the beginning of the operation, the planner calls the method ($LinkBuild()$) (line 1), responsible for defining all the actions that will make the plan and its respective causal links.

Using the concept of POP, all actions that make up the plan have causal links. It is possible to separate those links into two types. The first is the precedence link, where an action that has such a link is necessary for the execution of another action that receives the second type of link, which is the condition link. For example, an action *collect-minerals* with a link to an action *build-factory* means that the action of collecting mineral precedes the execution of the action that builds a $Factory$. This action in turn has a link indicating that the action of collecting minerals is a condition for its execution. Thus, when an action has some of its attributes changed, it becomes easier to find out which other actions should change due to this occurrence. Each action with a link is located in lists, precedence links is in the list *link* and condition link in *linkReb* list (in Algorithm 2).

Algorithm 2 receives as parameters the plan of action ($Plan$) and the resource production goal (R_{goal}). Algorithm 2 shows the pseudo-code of the LinkBuild method.

Algorithm 2. LinkBuild($Plan$, R_{goal})

```
1: $Plan \leftarrow Extract(R_{goal})$
2: for each Action $Act \in Plan$ do
3:     if $Act.finished = false$ then
4:         for each Resource $Rsc \in Act.Pred$ do
5:             Action $Actn$
6:             $exist \leftarrow CheckPred(Act, Rsc, Actn)$
7:             if $exist = false$ then
8:                 $Act.linkReb.push(Actn)$
9:                 $Actn.link.push(Act)$
10:                $Plan.push(Actn)$
11:                $Plan \leftarrow LinkBuild(Plan, R_{goal})$
12:            else
13:                $Act.linkReb.push(Actn)$
14:                $Actn.link.push(Act)$
15:            end if
16:        end for
17:        $Act.finished = true$
18:    end if
19: end for
20: return $Plan$
```

Algorithm 2 inserts the first action in the plan and from its preconditions produces the remaining necessary actions. When an action is selected, it is checked to see if all its preconditions have been satisfied on the plan through the attribute exist (line 3). This checking is necessary due to the fact that when an action is placed in the plan to satisfy certain preconditions, the preconditions of this action that have just been entered are immediately checked (line 11). So it is possible that an action of the plan that will be checked has had its preconditions already satisfied.

Between lines 3 and 4 Algorithm 2 checks for each action of the plan to see if its precondition resources are present in the plan or must be inserted into it. This check is made by $CheckPred()$ (line 6), which checks whether there is an action in the plan that can satisfy the precondition or if it is necessary to insert a new action. If a new action is necessary, the value false is assigned to the variable $exist$ and the algorithm creates the links between the action and its precondition, putting the new action in the plan and calls itself to enter the preconditions of the new action (lines 7 and 11). If the action that satisfies the precondition already exists, then only the necessary links between these actions are created (lines 12–14).

The function $CheckPred()$ (line 6) receives the variable $Actn$ as a parameter and assigns it to either the new action that will be inserted into the plan or the action that is already there and can satisfy the necessary precondition. The algorithm validates the attribute, which exists for each action and that already has all its preconditions satisfied. At the end, the plan containing the actions and the links between them is returned to the POPlan to continue its execution. The plan that Algorithm 2 builds and returns to POPlan contains only the actions necessary to reach the resource goal production and the respective links between them. The starting and ending times of each action will be defined later by the scheduler, which does not need to perform any specific planning task, since the actions are already configured.

The POPlan after setting up the plan with the necessary actions and links through *LinkBuild*(), starts to go through the actions of the plan. Algorithm 1 between lines 2 and 5 verifies to see if the action is not the type of production unit, i.e., whether the action is the mineral or gas type, but only resources that are not of unit type. The function *Quantity*() (line 4) is called to check how many times the action should be executed. For example, if the action *build-minerals* has a link to a *build-academy* that requires 150 *Minerals*, three executions of the same action are necessary because each execution generates 50 *Minerals*.

After setting the starting time of the action through the scheduling process, the planner will now finish setting its attributes, again using the planning stage. The method *Constructs*() (line 12) is responsible for this task, by setting the starting and ending execution time of the action, inserting new actions of renewable resources if necessary and, modifying some link action if necessary due to the scheduling that can change these links. This procedure also checks there is any threat held in these (according to the concept of POP), and checks to see if the plan time limit has not been exceeded. The final plan that is built by the planner, contains the actions that reach a certain amount of resources.

6 SHELRChecker the Consistency Checker

With the search for goals toward plans of actions that have quality in army points and scheduled actions, verification and validation of new plans becomes even more important. The scheduled consistency checker, SHELRChecker, is used in this step. Algorithm 3 shows its pseudo-code.

Algorithm 3. SHELRChecker(*Plan*, *opt*, T_{limit})

```
1: $Invib(ActsInv, opt)$
2: for each Action $Act \in ActsInv$ do
3:     if $Act.feasi = true$ then
4:         for each Action $Actl \in Act.link$ do
5:             $UndoL(Actl, ActsInv)$
6:         end for
7:         for each Action $Actl \in Act.Clink$ do
8:             $ReleaL(Actl, ActsInv)$
9:         end for
10:    end if
11: end for
12: $Rearrange(Plan, ActsInv, opt)$
13: while $cont = true$ do
14:     for each Action $Act \in ActsInv$ do
15:         $feasible \leftarrow Gather(Plan, Act)$
16:         if $feasible = true$ then
17:             for each Action $ActRs \in Plan$ do
18:                 if $ActRs = Act.rBase$ then
19:                     $Schedule(Plan, ActRs, Act)$
20:                     $cont \leftarrow Constructs(Plan, Act)$
21:                 end if
22:             end for
23:         end if
24:     end for
25: end while
26: return $Plan$
```

The Algorithm 3, receives as parameters the current solution of the SA (*Plan*), the operation that will be performed in the plan *opt* and the time limit of the solution to be generated (T_{limit}). Initially the method $Invib()$ is called (line 1). It is responsible for checking what actions will be unfeasible due to the operation. The method places the first actions in the unfeasible action list *ActsInv*, which is necessary for the algorithm to find out which others will also become unfeasible.

The algorithm between lines 2 and 11 finds the other actions that will be unfeasible. For each one that still has the attribute ($feasi$) valid (line 3), i.e., every action that has just been added to the list of unfeasible, the algorithm calls the methods responsible for removing its links and finding out which other actions will also be unfeasible. The method ($UndoL()$) (line 5) is called to put each resource that receives an action link that was unfeasible (*Act*) within the list of unfeasible actions. This is carried out as (*Act*) which is a precondition of such actions, which will no longer run. For example, if an action *build-refinery* becomes unfeasible this means that the *Refinery* does not exist in the game anymore, so all actions *collect-gas* receiving a link of this resource will also become unfeasible.

The method $ReleaL()$ (line 8) is called just after $UndoL()$, it is responsible for releasing the actions that were serving as a precondition for the action that became unfeasible, thus eliminating the links from other actions that came from it. This frees the resources used by the action to be used by other actions. After the list of unfeasible actions has been completely filled, the algorithm calls the $Rearrange()$ (line 12). It performs the operation on the plan so that it becomes a new neighbor plan. From line 12 onwards, the algorithm focuses on what actions can become feasible again, configures its attributes and schedules the action. The function *Gather* (line 15) is used with every unfeasible action. It seeks the preconditions of the action that is being checked *Act*, to see if the available actions can be used to execute it.

When the preconditions of an action are satisfied within the plan, the algorithm schedules the action and changes its attributes, as they become feasible. A search for base resources of the action (line 17) is made to define which among those available would execute the action. When a base resource is found the scheduling algorithm is called $Schedule()$ (line 19). The algorithm is the same as used in the planner. It performs the schedule for each feasible action. Finally, the function $Constructs()$ (line 20) is called to change the attributes of the action that became feasible again. It sets up the execution times of actions, builds all the links between them and their preconditions, removes the action from the list of unfeasible and checks to see if the time limit has not been reached. If this time has been reached the verifier stops the process returning the plan with the last action that became feasible as new solution for SA.

7 Experiments and Discussion of Results

The experiments were conducted by using the StarCraft game environment. It was possible through the use of the BWAPI (BroodWar API) [4], which allows us to test the developed algorithms within the game and collect the results. We compare our results against human players and a bot from competitions. We ran our experiments using a computer equipped with an Intel Core i7 1.73 GHz with 8 GB RAM on Windows 7 system.

The procedures used in the experiments are: $SA(S)$, $S(SS)$, $Player(E)$ and BT. $SA(S)$ refers to our approach, with SA plus POPlan and SHELRChecker. $SA(SS)$ also refers to our approach in conjunction with the subgoals strategy, which will be presented during the experiments. $Player(E)$ is a human player experience level in the game. To be classified as an experienced player a 5-year experience with StarCraft was considered. BT is the bot called BTHAI [17], who participated in the 2014 AIIDE StarCraft Competition[1].

In all tests, the used approaches tried to achieve the best goal with resource production, i.e., a goal focused on produce resources that allow a player to advance against the enemy bases and overcome it in a confrontation. These goals produce various types of resources that increase the amount of army points. Produce resources with ability to fight enemys leads to the production of other high level types of resources present in the game, since its are required to enable more powerful resources. In fact, a feature observed in various players when played matches of the game is the trend to produce offensive resources along the match and only produce other types of resources when these are needed to enable the coming of new resources that will be used to attack and resist to the enemy's army. Thus, the approaches will be compared according the amount of resources that can produce within a time limit. Different time limits will be imposed on each test and approaches should try to achieve the best goal without exceeding this time limit (Table 1).

Table 1. Results of Experiment 1

Time limit	Player(E)	SA(S)	Makespan of SA(S)	Runtime of **SA(S)**
150 s	32	32	148 s	28.8 s
250 s	95	**98**	250 s	42.6 s
350 s	**151**	132	350 s	63.2 s
450 s	**207**	188	599 s	96.8 s
550 s	**254**	210	547 s	129.2 s
650 s	**369**	298	645 s	164.5 s
750 s	**437**	307	746 s	199.1 s

In Experiment 1, we conduct a performance test between the $SA(S)$ and $Player(E)$. $Player(E)$ was instructed to use as a strategy goals that produce resources he would use to form a base to confront and defend against enemy. For each time limit, we considered five executions and collected the average

[1] StarCraft Competition is a competition between Starcraft bots, this competition is an event that is part of AIIDE (AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment).

values for army points, makespan, and runtime. $SA(S)$ was able to defeat the experienced player in one occasion, tying in another one and losing in five others. One can see that $SA(S)$ achieves its best results when searching for goals with average time intervals between 150 and 350 s. This happens because in these time intervals the number of actions that the approach needs to manage is not as large as those at longer intervals, which helps the algorithm to achieve better performance (Table 2).

Table 2. Results of Experiment 2

SA(S)			SA(SS)		
Army points	Makespan	Runtime	Army points	Makespan	Runtime
242	596 s	148.6 s	**299**	599 s	100.1 *s*
245	594 s	149.9 s	**303**	600 s	101.4 *s*
241	592 s	148.5 s	**297**	600 s	98.8 *s*
242	590 s	147.1 s	**302**	597 s	99.4 *s*
243	593 s	151.5 s	**297**	600 s	98.2 *s*
243	596 s	148.8 s	**312**	598 s	97.7 *s*
242	598 s	150.9 s	**301**	600 s	100.1 *s*
245	588 s	149.4 s	**301**	598 s	98.3 *s*
237	594 s	149.7 s	**297**	600 s	99.6 *s*
240	593 s	151.2 s	**307**	598 s	97.1 *s*

In order to use our approach under real-time constraints we used a strategy that takes into account the fact the best results have been achieved considering time intervals that range from 150 to 300 s. This strategy consists of decomposing a resource production goal to be achieved in time intervals superior to 300 s into smaller resource production subgoals to be achieved in time intervals in which the algorithm can find the best results. As soon as a subgoal is achieved its plan of actions is carried out. While executing the actions that satisfies a subgoal, the algorithm keeps working in order to figure out a plan of action that satisfies the next subgoal, and so on. For example, suppose a 600 s time limit for resource production. In this case, the algorithm splits the resource production goal for this time interval into three resource production subgoals considering a 200 s time limit for each one. Whenever a goal is divided into subgoals, these have time intervals between 150 and 200 s.

Experiment 2 demonstrates the performance of the subgoals strategy. In this experiment, the $SA(S)$ and $SA(SS)$ (that uses the subgoals strategy) were executed ten times to find goals with time limits of 600 s. For comparison, the left column of the table contains the algorithm with its normal execution, where on the right hand column of the table, the strategy of dividing the main goal into subgoals was used. The algorithm divided the goal of 600 s into 3 subgoals of 200 s.

Through an analysis of the results in Experiment 2, the algorithm using subgoals in most cases found better solutions in a shorter runtime. This is due to subgoals containing smaller quantity of actions, which facilitates algorithm operations during the management of actions that become unfeasible due to changes in the plan. This feature also allows the algorithm to perform more modifications in the plans. In this way, the algorithm can exploit goals with better quality and with more army points. By using a single goal, the number of actions in the plan is high and causes the generated solutions to become complex to manage. In the subgoals strategy, when a subgoal finishes its execution we have as a result resources which are used in achieving the next subgoal.

Table 3. Results of Experiment 3

Player(E)		SA(SS)	
Army points	Number of military resources	Army points	Number of military resources
181	9	174	9
182	8	**184**	12
180	9	172	9
184	10	178	11
181	9	**182**	11
178	8	**184**	12
182	10	173	11
184	10	174	9
184	10	148	6
183	9	177	10

Table 3 presents the results of Experiment 3. A comparison is made again using $Player(E)$, only this time with $SA(SS)$. Ten executions with the time limit of 400 s were made. The $Player(E)$ obtained seven wins over $SA(SS)$ with respect to the army points. The $SA(SS)$ goals were able to produce more resources than the $Player(E)$ in most experiments. This occurred at times when the algorithm was able to produce more military resources than the human player, due to the approach of using subgoals strategy. Again, this approach generates solutions that have more resources with lower production cost. This happens because $SA(SS)$ schedules their actions in order to produce more resources. Due to space limitations, the description of the scheduling algorithm is going to appear in an upcoming paper. $Player(E)$ frequently concentrated on producing more powerful resources, but production was realized on a small scale due to the time limit imposed.

In Experiment 4 a comparison between the $SA(SS)$ and BT bot player is made. This was chosen because it is an agent with open code and also by the use of the resources of Terran class in Starcraft, the same used by our approach.

To accomplish this experiment tests were performed to determine the maximum time that *BT* was able to produce resources without using their resources on direct attacks within the game. This was considered the time that both approaches would have to perform their resource production. At the end, the quality of both in terms of army points was evaluated. *BT* was chosen for having an offensive resource production strategy, similar to that described at the beginning of this section. During each experiments execution the bot concentrated only on achieving goals that increase its ability to confront and be confronted by enemies. The time was set in 270 s and ten executions were carried out with this time.

With the results described in Table 4, $SA(SS)$ was able to overcome the BT in most of the executions. The source code of BT has not been changed since this could alter its characteristics and resource selection strategies. In most of the executions BT concentrated in producing Firebat (Terran resource) units, resource which has a good cost benefit. However, this strategy has a maximum amount of resources that can be built in a time interval and thus the values of army points achieved by BT were often similar. $SA(SS)$ does a search for the best combination of resources that maximize production and raise the power of the army produced. During program execution, $SA(SS)$ produces different combination of resources within the time interval and the best one is chosen as a goal along with its plan of actions.

Table 4. Results of Experiment 4

SA(SS)	BT	
Army points	Army points	Execution
109	85	**1**
85	**89**	**2**
106	84	**3**
109	85	**4**
104	82	**5**
106	85	**6**
108	89	**7**
109	84	**8**
106	85	**9**
108	82	**10**

Experiment 5 shows performance analysis of $SA(SS)$. Table 5 describes the average results of 15 runs with three different time intervals. For goals with an interval of 250 s, the amount of near-optimum solutions is large. This result is highlighted by the average army points (93) whose value is close to the optimal value. At intervals of 400 s good results were obtained. In the 600 s intervals, a

Table 5. Results of Experiment 5 using SA(SS)

Time limit	250 s	400 s	600 s
Optimum value of army points	100	184	312
Number of runs over 95% of optimum	84%	41%	37%
Number of runs over 90% of optimum	6%	49%	44%
Average value of ***runtime***	32.2 s	64.6 s	99.1 s
Average value of army point	93	167	281
Number of runs	15	15	15

drop in performance begins to be presented. The results point to the effectiveness of this approach in the search for goals in shorter time intervals, in which the algorithm can obtain best results. This feature also reinforces the use of the subgoals strategy.

8 Conclusion and Future Work

In the previous sections we described our approach to resource production that aims to find goals with quality in RTS games. The approach uses Simulated Annealing, which from an initial plan of action figures a new plan that maximizes the production of resources within a given time limit.

Experiments allowed us to conclude that our approach was able to compete with an experienced human player and beat a mid-level player, in terms of quality in the production of resources. With respect to performance, experiments that make use of smaller time limits produce results close to the optimal value and in some cases the optimal plan of actions.

As a future work we are going to investigate whether other local search algorithms might improve the results already achieved. We also want to investigate other strategies that can be used to reduce the complexity in the management of actions during the generation of new plans. In the experiments we compare the performance of our approach with just one bot player (BT). Others AI players are going to be used for comparison. In the near future our goal is the management of resources to be used in tactics for for attack and defense in conjunction with the approach already developed in order to build a complete player agent.

Acknowledgments. This research is supported in part by the Coordination for the Improvement of Higher Education Personnel (CAPES), Research Foundation of the State of Minas Gerais (FAPEMIG) and Faculty of Computing (FACOM) from Federal University of Uberlândia (UFU).

References

1. Aarts, E., Korst, J.: Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimisation and Neural Computing. Wiley, New York (1989)

2. Branquinho, A., Lopes, C.R., Naves, T.F.: Using search and learning for production of resources in RTS games. In: The 2011 International Conference on Artificial Intelligence (2011)
3. Buro, M.: Call for AI research in RTS games. American Association for Artificial Intelligence (AAAI) (2004)
4. BWAPI: BWAPI - an API for interacting with starcraft: broodwar. http://code.google.com/p/bwapi/. Accessed 09 Apr 2015
5. Chan, H., Fern, A., Ray, S., Wilson, N., Ventura, C.: Online planning for resource production in real-time strategy games. In: ICAPS (2007)
6. Churchil, D., Buro, M.: Build order optimization in starcraft. In: AI and Interactive Digital Entertainment Conference, AIIDE 2011 (2011)
7. Do, M.B., Kambhampati, S.: Sapa: a scalable multi-objective metric temporal planner. J. AI Res. **20**, 63–75 (2003)
8. Fayard, T.: Using a planner to balance real time strategy video game. In: Workshop on Planning in Games, ICAPS 2007 (2007)
9. Fink, E.: Changes of Problem Representation: Theory and Experiments. Springer, Heidelberg (2003)
10. Gerevini, A., Saetti, A., Serina, I.: Planning through stochastic local search and temporal action graphs in LPG. J. Artif. Intell. Res. **20**, 239–290 (2003)
11. Hoffmann, J., Nebel, B.: FF: the fast-forward planning system. J. Artif. Intell. Res., 253–302 (2001)
12. Jay, Y., Nick, H.: Evolutionary learning of goal priorities in a real-time strategy game. In: Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (2012)
13. Kecici, S., Talay, S.S.: TLplan-C: an extended temporal planner for modeling continuous change. In: The International Conference on Automated Planning and Scheduling (ICAPS) (2010)
14. Klenk, M., Molineaux, M., Aha, D.: Goal-driven autonomy for responding to unexpected events in strategy simulations. Comput. Intell., 187–203 (2013)
15. Minton, S., Bresina, J., Drummond, M.: Total-order and partial-order planning: a comparative analysis. J. Artif. Intell. Res. **2**, 227–262 (1994)
16. Naves, T.F., Lopes, C.R.: Maximization of the resource production in RTS games through stochastic search and planning. In: IEEE International Conference on Systems, Man and Cybernetics - SMC, pp. 2241–2246 (2012)
17. Ontanon, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., Preuss, M.: A survey of real-time strategy game AI research and competition in starcraft. IEEE Trans. Comput. Intell. AI Games (2013)
18. Weber, B.G., Mawhorter, P., Mateas, M., Jhala, A.: Reactive planning idioms for multi-scale game AI. In: Proceedings of IEEE CIG (2010)