The Real-Time Strategy Game Multi-Objective Build Order Problem

Jason Blackford and **Gary Lamont**
Department of Electrical and Computer Engineering, Air Force Institute of Technology
Wright-Patterson Air Force Base, Dayton, Ohio, United States of America

Abstract

In this paper we examine the build order problem in real-time strategy (RTS) games in which the objective is to optimize execution of a strategy by scheduling actions with respect to a set of subgoals. We model the build order problem as a multi-objective problem (MOP), and solutions are generated utilizing a multi-objective evolutionary algorithm (MOEA). A three dimensional solution space is presented providing a depiction of a Pareto front for the build order MOP. Results of the online strategic planning tool are provided which demonstrate that our planner out-performs an expert scripted player. This is demonstrated for an AI agent in the Spring Engine Balanced Annihilation RTS game.

Introduction

A goal of current real-time strategy (RTS) game AI research is to develop an agent that performs at the level of an expert human player. A multitude of researchers explain that there exist a set of RTS player competencies an expert must master (Cunha and Chaimowicz 2010) (Coy and Mateas 2008) (Sailer, Buro, and Lanctot 2007). These competencies include strategy execution, tactical maneuvering of units in battle, resource collection, production of units, buildings and upgrades, and scouting enemy positions. Each of these are intertwined with one another in that they introduce competing goals and resource allocation needs.

The competencies may be divided into several managers including: strategy, resource collection, production, tactics, and finally a reconnaissance manager. Each of these managers possess their own goals that compete for resources and execution. These managers can be integrated into a framework that enables these managers to interoperate as a functioning RTS game AI. One specific framework is known as a multi-scale agent (Weber et al. 2010). We adopt this approach for our investigation.

In addition to the RTS player competencies, Weber (Weber 2012) presents three capabilities unique to expert players in RTS games: estimation, adaptation, and anticipation. Estimation and anticipation relate to predicting an opponents next action based on an observed strategy the opponent is executing. Adaptation corresponds to an agent's planning mechanism in which an agent adapts its plans in real-time to cope with changes in an uncertain environment and opponent. These three capabilities enable an AI to learn from and reason about its world. Together the competencies and capabilities describe an AI framework that enables an agent to manage competing goals in an uncertain and dynamic environment.

To summarize the high level requirements for the development of an expert RTS agent, we generated the expert RTS agent pyramid in Fig. 1. The foundation of the pyramid is the RTS player competencies that are implemented with various AI algorithmic techniques. These techniques can range from complex search algorithms (Sailer, Buro, and Lanctot 2007) to scripted managers that encapsulate each competency. Above the player competencies are the expert RTS player capabilities. The capabilities rely on the algorithmic techniques utilized to implement the competencies. Collectively the competencies and capabilities enable an AI agent to handle uncertainty and manage competing goals in an RTS environment.

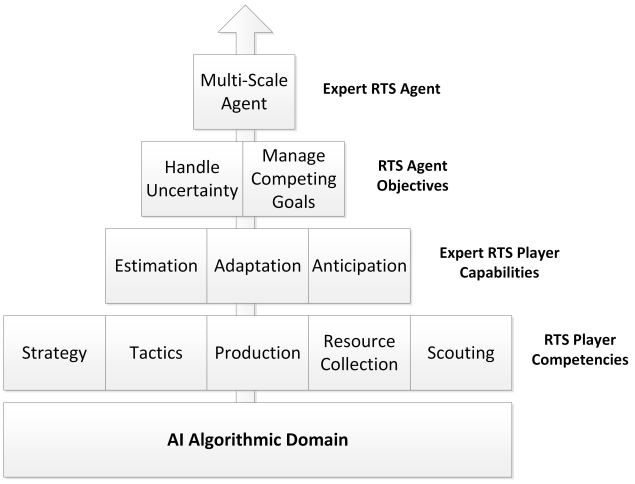

Figure 1: The expert RTS agent pyramid organizes from the bottom up what is necessary for constructing a multi-scale agent capable of playing an RTS game at an expert level.

The research problem we address in this paper is on optimizing the performance of the strategic decision-making process or build orders of an RTS agent. This investigation includes the development and analysis of an RTS agent that optimizes strategy execution in order to perform at an expert human level (Blackford 2014)[1]; therefore, we make the assumption that the build orders generated by an expert player are near-optimal. The objective of our planning tool is to produce build orders that are as good as or better than expert build orders or near-optimal. Our approach is real-time and is comparably as fast as the approach presented by Churchill and Buro in (Churchill and Buro 2012) whose depth-first search (DFS) technique utilizes around five seconds of planning time to produce build orders in Starcraft.

Background

Players of an RTS game attempt to advance the skills and technology of their cities and armies in order to over power their opponent. This technology advancement is defined in the technology tree of an RTS game. A technology tree is a large directed decision tree that determines the execution of a player's selected strategy. It establishes an ordering of actions a player must take in order to build and advance their army. This leads to the concept of build orders.

A build order is an ordered sequence of actions a player takes to execute their chosen strategy. Strategy execution is a planning problem requiring two components: first, determining a good goal-ordering to ensure expert level execution of a strategy with respect to resource allocations and time to execute the strategy; and second, issuing and executing actions in a build order to ensure expert or near-optimal player performance measured in resource allocations and time to reach a goal. With respect to providing an AI agent with the ability to rationalize across this planning problem, the first component involving goal-ordering can be determined by deriving a goal-ordering from the replays of an expert RTS player (Weber and Mateas 2009) (Ontanon et al. 2007) and then provide this to the agent via case-based reasoning (CBR). Resolving the second component is known in the RTS research community as the build order optimization (Churchill and Buro 2011) problem and is the focus of this investigation.

Build Order Methodology

This investigation frames the RTS strategic decision-making process as a build order optimization problem. The build order problem is modeled as a multi-objective problem (MOP) (Coello Coello, Lamont, and Veldhuizen 2007) with three objective functions and three constraints. The mathematical model of the strategic decision-making process is then integrated into an RTS game simulator capable of executing strategic decisions common to generic RTS games including Starcraft: Broodwars and Balanced Annihilation. The simulator is written in C++ and utilizes seven XML schema files to define any RTS game to be simulated (Blackford 2014). As build orders are passed into the simulator, it scores them based upon the three objective functions of the build order MOP. To generate build orders, an MOEA framework in conjunction with the RTS strategic decision simulator is utilized to score a population of randomly generated build order plans. To enhance the discovery of near-optimal solutions, expert solutions are injected into the population of the MOEA framework utilizing CBR (Blackford 2014). This results in an RTS strategic planning capability that is online and provides an agent with near-optimal build orders. The capabilities of the strategic planner are demonstrated in the Spring game engine with the Balanced Annihilation RTS game. Results for the ability of our model to approximate Starcraft build orders are presented in (Blackford 2014), but the focus of this paper is the RTS Spring Engine [2] Balanced Annihilation (BA) game.

Build Order as Producer/Consumer Problem

In formulating the mathematical model to approximate the build order problem, a goal programming approach was adopted (Coello Coello, Lamont, and Veldhuizen 2007). Goal programming is defining objective functions that minimize a distance to a goal. Distance is defined loosely as any quantitative metric that can be utilized to measure equivalence of a feasible solution from a multi-objective problem population to a target goal.

The conjecture that the BOO problem is a planning and scheduling problem with producer/consumer constraints is first addressed in (Wei and Sun 2009). The scheduling problem consists of two parts: Scheduling highly cumulative activities and scheduling highly disjunctive activities. Both pieces are required to minimize the makespan for executing and completing actions to reach a desired goal state from an initial state. Cumulative actions are actions that can be executed or issued concurrently (overlap) on the same resource. This applies to actions requiring volumetric resources. Disjunctive scheduling consists of pairs of actions that cannot be executed concurrently on the same resource. This applies to actions requiring unary resources. It is possible for actions to require both unary and volumetric resources; however, in the RTS domain an action most likely requires one unary resource and no more than two volumetric resources. For example, in the popular Age of Empires (AOE) games a soldier requires a barracks (unary) and food or wood and gold (volumetric resources).

Cumulative Scheduling

In cumulative scheduling, actions or activities are constrained by volumetric resources. The constraint is the cumulative producer/consumer constraint (Wei and Sun 2009). The cumulative constraint requires that the sum of the amounts of consumable resources required by a set of actions scheduled at a time t does not exceed the amount of the resource available at time t. For example, if it costs 100 food and 15 gold to produce a foot solider in an RTS game, and it costs 200 food and 100 gold to research advanced armor for the foot soldier, then to execute these actions concurrently

[1] https://code.google.com/p/online-planning-rts-agents/

[2] https://www.springrts.com/

requires a total of 300 food and 115 gold at the time of requesting execution of these activities. With respect to RTS games and volumetric resources in RTS games a slightly modified representation of the cumulative constraint (Cornelissens and Simonis 1995) can be expressed as 1:

$$\sum_{a_j \in A}^{|A|} r_j^t \leq L_t \tag{1}$$

Where t is the time of request of the set of actions contained in the action set A, and a_j is a member of the set of actions being executed at time t. The amount of a single volumetric resource (gold, food, or gas) required by an action a_j is expressed as r_j. The total amount of the volumetric resource available at time t is represented as L. Keep in mind that most actions require several resources, but this expression only captures the constraint between actions on a single volumetric resource. This constraint must be satisfied for all volumetric resources shared between actions executed at a time instance t.

Disjunctive Scheduling

This discussion is derived from the definition of a unary resource in RTS games from (Wei and Sun 2009). At first it would seem that the number of workers a player possesses can be considered a volumetric resource. For each action that requires a worker, simply assign a worker to the action. However, individual workers themselves are a resource, so it is better to divide the non-unary resource (total number of workers) into unary resources (individual workers)(Wei and Sun 2009). A non-unary resource in an RTS game is a renewable resource - this excludes consumable resources which are volumetric. In the RTS domain each non-unary renewable resource of capacity C is divided into C subsets (one capacity per subset), and each activity requiring the non-unary resource is divided into C sets and assigned one of the unary subsets. For example, if $C = 15$ workers then there exists 15 subsets of the now unary resource worker. Possible activities requiring a worker include: mining a finite amount of gold or building a structure which has a constant duration. Each one of these actions requires holding onto the unary resource for some constant duration that is generally fixed and known (derived empirically via a simulation or available data).

A popular technique for performing disjunctive scheduling is constraint satisfaction programming (CSP) (Baptiste and Pape 1996). The objective in using CSP to schedule activities under a unary resource constraint is to reduce the set of possible values for the start and end times of pairs of activities sharing the unary resource. A precedence ordering is establish for actions as follows: (Baptiste, Pape, and Ilog 1996) given two actions A and B that both require the same unary resource, schedule them according to expression 2:

$$end(A) \leq start(B) \vee end(B) \leq start(A) \tag{2}$$

This expression shows that A precedes B or B precedes A with respect to time. A solution is found when the assignment of domain start times satisfies the Boolean expression. Expression 2 is the RTS game disjunctive constraint.

Exist Constraint

A final constraint that is present in RTS games is the exist constraint. The exist constraint is defined implicitly in all RTS games and is established by the technology tree of an RTS game. It stipulates that an action A can only be executed if a unary resource R exists. This is different from the disjunctive constraint in that the action A does not require use of the resource R, only that the resource exists. For example, in Starcraft before a $Barracks$ can be constructed a $Command\ Center$ must exist. A unary resource cannot be both an exist and disjunctive constraint for a single action to be executed. The exist constraint can be formulated logically as the implication statement 3:

$$B \Rightarrow A \tag{3}$$

Where B and A are actions. This expression draws a logical implication between actions B and A (B implies A). For example, in Starcraft the unary resource $Command\ Center$ must exist before a unary resource $Barracks$ can be produced. Therefore, the action $Build\_Barracks$ is B and the action $Build\_Command\_Center$ is A. An alternative view is to bind the variables B and A to the unary resources produced from the actions $Build\_Barracks$ and $Build\_Command\_Center$. From this alternative binding, the unary resource $Barracks$ cannot exist without the unary resource $Command\ Center$. This implies that the action $Build\_Barracks$ cannot be taken without action $Build\_Command\_Center$ having already completed.

Multi-Objective Build Order Optimization

The multi-objective build order optimization problem (MOBOO) is defined by three objective functions and by the cumulative, disjunctive and exist constraints. The three objective functions are as follows:

objectives,

$$min(\sum_{i \epsilon A_G}^{|A_G|} D_i * (\{G_i\} - \{S_i\})) \tag{4}$$

$$min(\frac{1}{C_r}[\sum_{r \epsilon A_G} (A_{G_r}) - R_{S_r}]) \tag{5}$$

$$min(\sum_{i \epsilon A}^{|A|} F_i) \tag{6}$$

constraints,

$$B \Rightarrow A \tag{7}$$

$$\sum_{a_j \in A}^{|A|} r_j^t \leq L_t \tag{8}$$

$$ET(A) \leq ST(B) \vee ET(B) \leq ST(A) \tag{9}$$

The descriptions of the three objective functions are provided in the order that the functions are presented in the MOBOO MOP.

1. The first objective function, equation 4, is to minimize the summed duration of the actions i contained in the set A_G, defined by the goal state G_i, not taken by the agent in its current game state S_i. The duration of an action is captured by D_i. This is a mathematically defined objective function derived from the heuristic function description introduced by Churchill and Buro in (Churchill and Buro 2011).
2. The second objective function, equation 5, is to minimize the difference in required volumetric resources r (defined by the goal state and its action set) and available volumetric resources in the agent's current state, R_{S_r}, for all volumetric resources required by the actions in A_G. The quantity C_r is the collection rate for a resource r. Said in another way, to minimize the time required to collect the volumetric resources needed to execute the set of actions in the goal state or A_G. This is also derived from one of the heuristic functions introduced by Churchill and Buro in (Churchill and Buro 2011). Again the authors loosely defined the heuristic in words, but we have mathematically formulated our derived objective function here for modeling purposes.
3. The third objective function, equation 6, is makespan. The objective is to minimize the completion time of execution for a set of actions A. The finish time of an action i is stated as F_i.

Decision and Objective Space

Solutions in the MO-BOO decision space are action strings or build orders represented with two variable types. The first variable is an array of integers with a domain of values ranging from one to the total number of available strategic actions defined by an RTS technology tree. The decision variables contained in the integer array are initialized to a random ordering of actions. The second variable type is a binary array. This array is the same size as the integer array and each decision variable corresponds to whether or not the action defined in the integer array is to be taken. A one is used to enforce execution of an action and a zero is to signify not to take an action. As solutions are simulated in the simulator a repairing function flips the decision bits of individual actions depending on whether or not an action is feasible (passed constraints). Therefore, an initially infeasible solution becomes a feasible solution that maps to objective space. This encourages exploration, diveristy and preserves good building blocks.

The Objective space of MO-BOO is three dimensional and defined by the three objective functions. The phenotype of an action string from solution space is a single point in the three dimensional objective space. The Pareto front (Coello Coello, Lamont, and Veldhuizen 2007) is a tradeoff surface between the three objectives. The domain of the objective space is positive real numbers which means the objective space is uncountably infinite. A large objective space allows for more diversity amongst solutions.

The objective functions for each solution are calculated from an inputted goal state provided by the user. The goal state lists the desired volumetric and unary resources the player would like to reach as well as the combat units. The solutions attempt to reach this goal state and are not limited in time, but by their string length also referred to as action string length or bit length. An example of a solution mapping to objective space is provided below for clarification.

Genotype Representation: 7 1 5 7 0 1 5 7 3 6::1111101111

The genotype representation identifies actions to be executed as integers on the left-side of the colons. Each integer is an identifier for some action to be executed. The binary on the right-side of the solution representation dictates whether a corresponding integer on the left-side should be or can be executed due to constraints or solution feasibility.

The objective scores in Table 1 reveal the following about the solution selected. The first objective measured in at 120 seconds. This means that the solution had a lower bound of 120 seconds, which is the summation of the durations of the actions remaining to be taken to reach the goal state. This is the best possible time remaining if the resource constraints are satisfied. The quality of this measure or how close to this lower bound the solution is relies on the second objective which measures the required resources the plan needed in order to reach the goal. A value of 102.22 reflects that by the time the plan ended it still required 102.22 seconds to meet the volumetric requirements necessary to satisfy the goal state. Therefore, the time required for the build order to reach the goal state exceeds the lower bound established by the first objective. The third objective measures the total time or makespan of the actions completed by the plan. In the end, this plan would have reached the goal state if more decision variables or the length of the action plan was increased.

Table 1: List of Objective Measures

Objective #	Fitness Score
1	120.0
2	102.22
3	786.67

Experiments

For experimentation we utilized the Ubuntu 11.10 32-bit operating system, Spring Engine V91.0, Balanced Annihilation V7.72, Starcraft: BroodWars, Python 2.7, JMetalCPP v1.0.1. All C and C++ code is compiled with GCC 4.6.1 and/or Intel's compiler ICC. The hardware specifications are 8GB Memory, 2.80GHz Intel Core2 Duo, and NVIDIA Quadro. Matlab 2013a was utilized for analysis. We utilized the Jmetal metaheuristic algorithm suite (Nebro and Durillo 2013) to define and solve our multi-objective problem. Jmetal provides the ability to define unique solution types and is widely used throughout the metaheuristic community. We selected the NSGAII algorithm from Jmetal. More on why NSGAII was selected can be found in (Blackford 2014).

MO-BOO Pareto Fronts

To visualize the tradeoff surface of our three objective functions we plot several Pareto fronts produced by our strategic planner for the initial Balanced Annihilation (BA) state and goals defined in Table 2.

Table 2: Initial State and formulated Goals small(s) and large (L). armvp: ARM vehicle plant; armmex: metal extractor; armsolar: solar panel; armstumpy: assault tank; C.R: collection rate; Amt: amount

Resources	Initial State	Goal (S)	Goal (L)
armvp	0	-	-
armmex	0	-	-
armsolar	0	-	-
armstumpy (tanks)	0	3	6
metal C.R	1.5	-	-
metal Amt	1000	-	-
energy C.R	25	-	-
energy Amt	1000	-	-

To produce the Pareto fronts the strategic planning tool settings in Table 3 are used because they have empirically demonstrated to provide computationally fast and near-optimal results with respect to various BA strategies. These parameters tune the MOEA so that the planner is suitable for online use by an RTS agent in the BA game. The $BitLength$ parameter is the action string length or genotype size of the solutions. This relates to the maximum number of actions the planner can return for a build order. In most cases, the planner returns build orders with a fraction of the actions represented by the bit string. The time column in Table 3 is the average (over five runs) in game runtime to generate a single build order. It may be possilbe to decrease this runtime further through parallelization of the MOEA.

The parameters used for the simulator are displayed in Table 4. These parameters are derived from the Balanced Annihilation game. Other technical requirements for the simulator were obtained from technical data provided by the Balanced Annihilation game developers website [3].

The simulator implementation is designed to be a customizable RTS game strategic decision-making (build order) simulator. It is designed around the mathematical formulation of the MO-BOO problem. It strictly enforces the exist, cumulative, and disjunctive constraints, however, the objective functions can be easily modified. The simulator design supports RTS games like Age of Empires (AOE), Starcraft, and Wargus. Most RTS games can be simulated by properly defining the XML schema files (Blackford 2014).

It is important to note that not all RTS games follow the same formula. For example, the Total Annihilation and BA games are fundamentally different from games like Starcraft and AOE largely in regards to how the economies are structured. To distinguish these two types of RTS games, games like Starcraft, AOE or Wargus are labeled cumulative economy games, and games like Total Annhilation and BA are labeled non-cumulative economy games. The justification for this naming scheme comes from the cumulative constraint present in MO-BOO. Cumulative economy games must satisfy the cumulative constraint specified in MO-BOO prior to issuing and executing actions, however, non-cumulative economy games do not. In fact this constraint is absent from the MO-BOO model that approximates BA's strategic decision making process.

[3] http://www.balancedannihilation.org/

Table 3: Balanced Annihilation C++ Planner: NSGAII Parameters and planning tool execution time - execution time is not a parameter, but a performance metric. The mutation rate for all runs is 1/b and the crossover rate is 90%. Pop Size: population size; Evals: number of evaluations in thousands; Bit Len: bit length; Time: average execution time over 5 runs in seconds

Goal	Pop Size	Evals	Bit Len	Runs	Time(secs)
S	50	6k	b=60	5	6.09
S	50	6k	b=100	5	6.70
S	50	6k	b=200	5	14.32
L	50	6k	b=60	5	8.778
L	50	6k	b=100	5	9.92
L	50	6k	b=200	5	15.75

Table 4: Balanced Annihilation C++ Planner: Simulator Parameters

Metal Collect Rate	Energy Collect Rate
2.04/sec per metal extract	20/sec per solar panel

A computed Pareto front is depicted in Fig. 2. This front reveals that the larger the action string or genotype length of the solution the closer each objective function is to zero. This provides a decision surface that brings an agent closer to the goal.

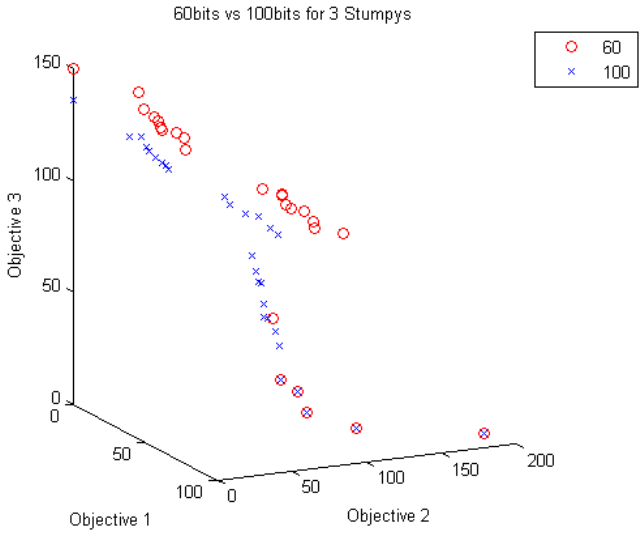

Figure 2: Comparison of the Pareto fronts produced by the parameters in Table 3 for the small goal which required only 12 actions to be reached.

For a decision maker, RTS agent or expert designer, the

best solution from the solutions presented in the Pareto fronts might be the one that minimizes objectives one and two to zero and has the smallest constant value for objective three. Objective three is measured on the vertical axis; therefore, a point in objective space lying on this axis reaches the goal, but may not be optimal with respect to makespan. Again this is a tradeoff surface. Another, decision maker may decide that the solution that is close to the goal, but executes faster than a solution lying on the vertical axis is a better choice. For example $(10, 0, 100)$ may be considered a better solution then $(0, 0, 300)$ since only ten additional seconds are required to reach the goal moving the actual makespan to 110 seconds over 300 seconds. Table 5 provides planning results for the goals presented in Table 2 across various action string lengths.

Table 5: Balanced Annihilation C++ Planner: The best build orders are marked with $*$.

Goal	Bit Length	Fitness Score	Planning Window
S	b=60	(0,0,150)	-
S	b=100	*(0,0,136)	12
S	b=200	(0,0,142)	-
L	b=60	(0,0,322)	-
L	b=100	(0,0,232)	-
L	b=200	*(0,0,198)	24

Agents for Experimentation

Agent LJD is the BA multi-scale AI developed at the Air Force Institute of Technology (AFIT) for the RTS Spring Engine BA game. We consider agent LJD to behave at an expert level with regards to strategy execution because the agent's strategy manager is scripted by an expert player. More on this agent and its design can be found in (Trapani 2012).

Agent BOO - Build Order Optimization - is our newly implemented agent. This agent utilizes our strategic planning tool to discover and execute build orders that are as good as or better than expert build orders. For experimental purposes only, Agent BOO executes any one of the strategies defined in the list that follows (Blackford 2014)(Trapani 2012):

1. **Tank Rush:** Requires building an initial economy and then producing attack waves of three stumpy tanks.
2. **Expansion:** The agent develops a large attack wave and base defenses while expanding it's volumetric resource control.
3. **Turtle:** The agent builds base defenses and infrastructure while Slowly producing large attack waves.

The strategies are manually divided into subgoals and placed into a CBR case dataset offline. During gameplay agent BOO's planner selects a case from BOO's assigned strategy's case set. The planner selects a case based upon agent BOO's current game state and passes the goals of the case to the strategic planning tool.

Throughout experimentation Agent BOO's planning window is limited to no more than fifteen actions in order to minimize planning times and enforce intermediate goal planning (Chan, Fern, and Ray 2007). A planning window as defined in (Weber, Mateas, and Jhala 2011) is the maximum number of actions an agent can execute to reach a goal state. Agent BOO is demonstrated as being capable of planning across 21 unique actions, but this is not a limit to the number of actions that our approach can plan across. These actions are derived from the first level of the Balanced Annihilation technology tree. Agent BOO is able to plan across the entire decision space captured in the first technology level. The planner can be modified to incorporate more BA RTS actions via XML schema files apart of the simulator, however, it was unnecessary for our experiments.

Tank Rush Strategy Analysis

The results presented in this section are derived from the best game out of five games each agent played in isolation. The purpose for playing in isolation is to observe which agent executes the strategies the fastest, under the assumption that an expert RTS player always executes strategies faster than a player of a lower skill level.

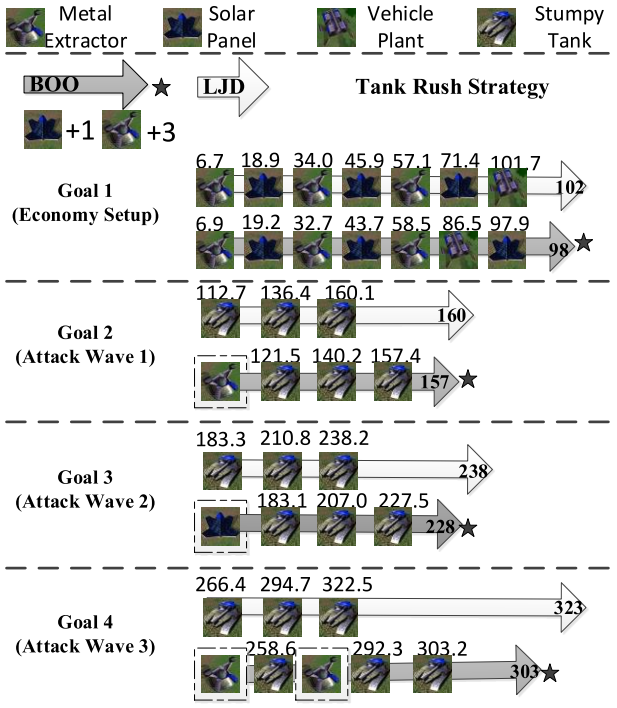

Figure 3: In game execution of the Tank Rush strategy. The data presented reflects the best game out of five games played in the Spring Engine.

From the build order timeline of the Tank Rush strategy displayed in Fig 3, it is obvious that agent BOO executes the strategy faster than agent LJD. It can be observed in Goal one that agent BOO decides to produce a vehicle plant earlier than agent LJD. This allows the vehicle plant of agent BOO to start producing the first tank of assault wave one 13 seconds earlier than agent LJD. Notice, however, in Goal

two that LJD still beats BOO in developing the first tank. This is because agent BOO decides to go and produce additional infrastructure, a metal extractor, to increase the volumetric resource collection rate in Goal two to decrease production time; whereas, agent LJD is scripted to remain with the vehicle plant and assist it in producing tanks for all attack waves. Agent BOO then returns to the vehicle plant in Goal two to assist in completing the remaining two tanks. With the increase in metal supply, agent BOO is able to complete Goal two 3 seconds faster than agent LJD. By the end of Goal three, agent BOO is now 10 seconds ahead of agent LJD in overall strategy execution. Finally, in Goal four agent BOO builds more infrastructure and ends Goal four 20 seconds ahead of agent LJD with respect to overall strategy execution. Agent BOO produces three additional metal extractors versus agent LJD, and one more solar panel versus agent LJD as noted at the top half of Fig 3.

More results are presented in Table 6. The bold text indicates the best time. The times represent when an attack wave first appears in the game state. It is clear from this table that agent BOO out-performs the expert scripted agent LJD in all strategies.

Table 6: Goal Strategy Execution Timeline. Waves (W) identify the release times of attack waves in seconds. T.R: Tank Rush; Exp: Expansion; Tur: Turtle.

Agent	Strategy	W 1	W 2	W 3	W 4
LJD	T.R	160.1	238.2	322.5	399.5
BOO	T.R	**157.4**	**227.5**	**303.2**	**353.0**
LJD	Exp	573.0	**838.8**	-	-
BOO	Exp	**519.9**	847.9	-	-
LJD	Tur	622.3	865.0	-	-
BOO	Tur	**617.5**	**833.9**	-	-

Conclusion

We provide the RTS research community with a concise mathematical model of the RTS build order problem that can be utilized to generate near-optimal build orders for cumulative and non-cumulative economy real-time strategy games including: Starcraft, Wargus, Age of Empires and Balanced Annihilation. The build order optimization problem consists of three objective functions and three constraints. In addition, we provide a unique, online multi-objective approach for solving the build order optimization problem in real-time. Our solution integrates our build order mathematical model, a simulator, an MOEA, and CBR to produce expert level build orders for an RTS agent playing against an opponent in the Spring RTS game engine in real-time.

References

Baptiste, P., and Pape, C. L. 1996. Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling. In *Scheduling, Proceedings 15th Workshop of the U.K. Planning Special Interest Group*.

Baptiste, P.; Pape, C. L. E.; and Ilog, S. A. 1996. Disjunctive Constraints for Manufacturing Scheduling: Principles and Extensions. *International Journal of Computer Integrated Manufacturing* 9(4):306–310.

Blackford, J. M. 2014. Online Build-Order Optimization for Real-Time Strategy Agents Using Multi-Objective Evolutionary Algorithms. Master's thesis, Air Force Institute of Technology.

Chan, H.; Fern, A.; and Ray, S. 2007. Extending online planning for resource production in real-time strategy games with search. *Workshop on Planning in Games, ICAPS*.

Churchill, D., and Buro, M. 2011. Build Order Optimization in StarCraft. In *7th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 14–19.

Churchill, D., and Buro, M. 2012. Incorporating Search Algorithms into RTS Game Agents. In *In Proceedings of the AIIDE Conference*, 14–19.

Coello Coello, C.; Lamont, G.; and Veldhuizen, D. 2007. *Evolutionary Algorithms for Solving Multi-objective Problems*. Springer, 2nd edition.

Cornelissens, T., and Simonis, H. 1995. Modelling Producer / Consumer Constraints. In *Workshop on Constraint Languages and Their use in Problem Modelling*.

Coy, J. M. C., and Mateas, M. 2008. An Integrated Agent for Playing Real-Time Strategy Games. In *in Proceedings of the AAAI Conf. on Artificial Intelligence*, 1313–1318.

Cunha, R., and Chaimowicz, L. 2010. An Artificial Intelligence System to Help the Player of Real-Time Strategy Games. In *Proceedings of the 2010 Brazilian Symposium on Games and Digital Entertainment*, SBGAMES '10, 71–81. Washington, DC, USA: IEEE Computer Society.

Nebro, A., and Durillo, J. 2013. jMetal 4.3 User Manual.

Ontanon, S.; Mishra, K.; Sugandh, N.; and Ram, A. 2007. Case-Based Planning and Execution for Real-Time Strategy Games. In *Case-Based Reasoning Research and Development*. Springer Berlin Heidelberg. 167–178.

Sailer, F.; Buro, M.; and Lanctot, M. 2007. Adversarial Planning Through Strategy Simulation. *2007 IEEE Symposium on Computational Intelligence and Games* 80–87.

Trapani, L. D. 2012. A Real-time Strategy Agent Framework and Strategy Classifier for Computer Generated Forces. Master's thesis, Air Force Institute of Technology.

Weber, B. G., and Mateas, M. 2009. A data mining approach to strategy prediction. *2009 IEEE Symposium on Computational Intelligence and Games* 140–147.

Weber, B. G.; Mawhorter, P.; Mateas, M.; and Jhala, A. 2010. Reactive planning idioms for multi-scale game AI. *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games* 115–122.

Weber, B.; Mateas, M.; and Jhala, A. 2011. Using Data Mining to Model Player Experience. In *In FDB Workshop on Evaluating Player Experience in Games, Bordeaux, France, ACM. ACM.*

Weber, B. G. 2012. *Integrating Learning In A Multi-Scale Agent*. Dissertation, University of California, Santa Cruz.

Wei, L., and Sun, L. 2009. Build Order Optimisation For Real-time Strategy Game. National University of Singapore.