Künstl Intell (2013) 27:221–233
DOI 10.1007/s13218-013-0263-2

TECHNICAL CONTRIBUTION

A Multi-objective Genetic Algorithm for Build Order Optimization in StarCraft II

Harald Köstler · Björn Gmeiner

Received: 28 February 2013 / Accepted: 31 May 2013 / Published online: 29 June 2013

Abstract This article presents a modified version of the multi-objective genetic algorithm NSGA II in order to find optimal opening strategies in the real-time strategy game StarCraft II. Based on an event-driven simulator capable of performing an accurate estimate of in-game build-times the quality of different build lists can be judged. These build lists are used as chromosomes within the genetic algorithm. Procedural constraints e.g. given by the Tech-Tree or other game mechanisms, are implicitly encoded into them. Typical goals are to find the build list producing most units of one or more certain types up to a certain time (Rush) or to produce one unit as early as possible (Tech-Push). Here, the number of entries in a build list varies and the objective values have in contrast to the search space a very small diversity. We introduce our game simulator including its graphical user interface, the modifications necessary to fit the genetic algorithm to our problem, test our algorithm on different Tech-Pushes and Rushes for all three races, and validate it with empirical data of expert StarCraft II players.

Keywords Genetic algorithm · Multi-objective optimization · NSGA II · Starcraft II

1 Introduction

Electronic sports, i.e. competitive play of video games, attracts more and more people, and StarCraft® II: Wings of Liberty™ published by Blizzard Entertainment[1] is one of the most popular real-time strategy games[2] currently. It is the successor of the first StarCraft game that has already been published in 1998. Typically, during a match in the BattleNet,[3] up to eight human players in two teams compete with each other. Each player selects one out of three possible races (Terran, Protoss, or Zerg) that all have different types of units. A unit can be thought of as a single soldier with properties like health, attack strength, or movement speed. A player can control his units in real-time, i.e. let them move forward or let them attack enemy units. Structures are buildings, where units can be produced in or unit properties can be improved via upgrades. Each player starts with a single base and some worker units in order to harvest resources, and then spends them on structures, units, and upgrades.

The goal is to use the units to destroy all structures in the enemy base. A match can be divided up into three different phases: early, mid, and late game. In the first phase, the players try to build up their main base and decide for a certain opening strategy, where one can focus either on fast expansion to increase later income, developing straightly new technology to build more advanced units, or quickly build as many early units as possible in order to rush the enemy, i.e. to attack as soon as possible. All possible units and structures of the three races and their technological requirements are described in the StarCraft 2 Tech-Tree.[4] Thus, the Tech-Tree lists for all structures, which units they can produce,

H. Köstler (✉) · B. Gmeiner
Cauerstr. 11, 91058 Erlangen, Germany
e-mail: harald.koestler@fau.de

B. Gmeiner
e-mail: bjoern.gmeiner@cs.fau.de

[1] ©2010 Blizzard Entertainment, Inc. All rights reserved. Wings of Liberty is a trademark, and StarCraft and Blizzard Entertainment are trademarks or registered trademarks of Blizzard Entertainment, Inc. in the U.S. and/or other countries.

[2] http://blizzard.com/en-us/games/sc2/.

[3] http://eu.battle.net/de/.

[4] http://us.battle.net/sc2/en/game/race/terran|zerg|protoss/techtree/wol.

and which other structures are required before one is able to build a certain structure.

In general, the best opening strategy depends on the map, the race, and also the opponents strategy. In mid game there are first real-time battles between different players, the type of produced units typically has to be adapted to the opponents strategy, and the players expand, i.e. build up additional bases in order to improve income. The winner of a battle depends of course on the number and type of units a player has, but also on his skills to move the units over the map and to choose appropriate target units for an attack (micro-management). The optimal build order thus provides an advantage in the battle, but is no guarantee to win. Towards the end in the late game phase all players are usually trying to take out enemy bases or defend their own bases having either a mix of different unit types or a large number of units of a certain kind. A whole game takes typically between 10 and 45 minutes.

Our goal in this paper is to evaluate the best opening strategies (build orders) for the early game phase under certain assumptions:

- the races of all players are known,
- the player is not interrupted by other actions done by other players,
- the player wants to either build one certain type of advanced unit as fast as possible (Tech-Push), or he wants to attack at a certain time with as many units of one or more types as possible (Rush).

We develop a tool to help players to decide which build order they can choose based on their opponents and time and unit restrictions. In order to determine quickly the feasibility of a certain build order and the exact build times of the contained objects, we have implemented a simulator for StarCraft 2 build orders. For short games up to 2–3 minutes it is then possible to use exhaustive search to find the optimal one with respect to a given strategy directly. Due to practical reasons, we neglect micro-management of the units because it heavily depends on the player's skills and therefore cannot predict the chances to win a game choosing a certain strategy accurately. Yet, we provide the income and value of the army to give a rough estimate of the quality of the defined strategies.

For longer games a comparison of the achieved results by random search with strategy lists provided by the community convinced us, that we were far off an optimal solution. Therefore, we implement a genetic algorithm (see e.g. [2, 4, 12, 15]) to find a good strategy.

To optimize several objective functions, we adopted the non-dominated sorting-based multi-objective algorithm II (NSGA-II) by [5, 6]. Other well-known multi-objective optimization evolutionary algorithms can be found e.g. in [9, 16, 18, 28].

Especially for the first StarCraft version from 1998 there exist a number of publications that deal with different aspects of the game. In [25] a multi-objective genetic algorithm [1] that is a descendant of the NSGA-II is used to optimize StarCraft maps with respect to fair resource and player placement.

An important issue in games is to improve the artificial intelligence (AI). In [23] the behavior of the AI is controlled by a neural network. The focus is here on micro-management, i.e. the movement and attacking strategy of units. Further AI approaches are found e.g. in [8, 21, 22, 27].

Close to our contribution that extends [11] is the work [3]. They present a depth-first branch and bound algorithm in order to optimize build orders for the first StarCraft version. Their goal was to perform the optimization in real-time and to incorporate it into the AI. The algorithm plans for a certain time duration the next actions in the build list starting from the current game state described by former actions and current resources. They prune the search space via respecting prerequisites on actions imposed by the Tech-Tree and available resources. Additionally, domain specific knowledge learned from game replays is used to insert certain fixed actions in the planned build list for specific game states. A Bayesian model for predicting build orders from observations, which parameters are also learned from game replays is introduced in [24]. It can also be incorporated in an AI and allows for simple and dynamic build order adaption to the opponent player strategy.

In contrast to these two approaches our genetic algorithm for the new StarCraft version aims for a (global) optimum. This enables us to provide build orders for human players offline, where we fix the player strategy independently of other players. Our algorithm is more suitable especially for opening strategies, where usually no interactions with other players are happening and one is interested in a global optimum of the objective values. With the Bayesian model, build orders with objective values that differ considerably from these found in the learning set cannot be predicted. The depth-first branch and bound algorithm is also not able to take into account the full build list for longer games.

Our paper is structured as follows: Sect. 2 briefly describes different stages and all constraints of the simulator (forward simulation) that is able to compute approximately the in-game build times of objects. Section 3 introduces the single-objective and multi-objective genetic algorithms for computing the optimal build orders. Features and implementation details of the simulator toolbox in C# and an efficient, parallel version in C++ are discussed in Sect. 4. The results for Rushes and Tech-Pushes in Sect. 5 show that our genetic algorithm is able to find reasonable build orders for all three races.

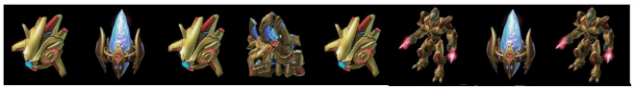

Fig. 1 Build list specifying the build order (from left to right) that consists of three worker units, two Pylon structures, one Gateway structure, and two Zealot units

2 Forward Simulation

Since it is required to test a large number of possible build lists and compute the corresponding build times, an efficient forward simulation is required to obtain an efficient optimization algorithm and reduce its overall computational time. Altogether the number of possible build lists increases exponentially with their size.

The simulator works with a list of players. Each single player assembles his build list either from a prescribed strategy build list or chooses all building actions in a random way. The build list consists of actions like producing structures, units, upgrades or do special operations like Chrono-Boost. Chrono-Boost is a Protoss race specific ability of a structure to accelerate its unit production. Figure 1 shows an exemplary build list for the Protoss race. During the simulation the resources, i.e. minerals and Vespin gas, are updated in a time-discrete manner each second in-game time, the remaining actions are event-driven.

A player chooses its next action in advance from his build list. If an action can be performed depends on two kinds of requirements: *Weak requirements* are delaying the start of the planned action. Weak requirements are that the player has enough resources (minerals and Vespin gas) if his income is larger than zero, or that there is a free slot in a production facility to produce a unit if the player has already constructed the necessary structure. In contrast *hard requirements* have to be fulfilled, otherwise the simulation will not proceed and the build list becomes invalid. Invalid build lists are not considered further in the later optimization process. Hard requirements are given by the Tech-Tree and the maximal available units that can be built, what depends on the number of bases and special structures, e.g. on the number of Pylons for the Protoss race. When all requirements are fulfilled the player starts the action and it is put in an active action list. After completion, the action is moved from the active action list to a finished action list.

Apart from dependencies due to the Tech-Tree the following race-dependent special rules have to be taken into account:

- Terran and Protoss can only build one unit per production structure at the same time.
- Terran can attach upgrades to structures. This enables for example the construction of new unit types in the upgraded structure or provides an additional production place.
- Terran can improve its minerals income by attaching an Orbital command update that costs 150 minerals to their main base structure and then using its energy to create a so-called MULE (Mobile Utility Lunar Excavator). MULEs are special worker units that exist only for 90 s in-game time, but harvest about six times more minerals than a usual worker unit.
- Zerg have no production structures, but instead obtain every 15 s a new larva from the main base structure that can be morphed into one of the different unit types.
- Zerg create a structure via morphing a worker into it. Thus the worker count has to be reduced by one, whenever a structure is built.
- The Zerg Queen unit is able to inject four additional larvae every 40 seconds and thus can accelerate the Zerg unit production.
- Unlike for other actions, it is not enough to decide, whether a Protoss' Chrono-Boost should be activated or not. An additional decision is required to determine on which type of structure it shall be applied. If there are more structures of one type, we decided to apply the following rules: Structures producing something are preferred, and secondly the structure with the next expiring Chrono-Boost is chosen.

During a forward simulation, we enter four different stages in each second:

1. *Check of possible actions*: If the next action is random, all possible actions are collected. If a strategy list is given, the next action is taken from the list. Additionally it is determined, if one has to wait due to a weak requirement.
2. *Resources* like mineral, Vespin gas, larvae count, and energy (required e.g. for Chrono-Boost) are updated.
3. *Start a production*: If an action is possible (i.e. no waiting state), it is put to the active action list. For Protoss units, a free place in a suitable production facility is found and assigned. Further new objects are generated for the units, structures, or upgrades.
4. *Check active action list*: Produced objects are removed from the action list and associated operations like increase of income, updating maximum number of units, or clearing of production slots are done.

3 Optimization

Finding the optimal build list is a discrete optimization problem since the list has a finite size (typically between 10–30 entries for early game simulations) and each entry is one of about 5–30 different discrete values representing the object type. The whole Tech-Tree for each race contains about 15 structure and 15 unit types. Even for short simulation times of 2–3 in-game minutes, where most of the object types cannot be built because of Tech-Tree prerequisites, we end up

with $5^{10} \approx 10^7$ possible forward simulations, and thus we cannot check all of them and select the best one. Note that even for the first entry in the build list there are already four different possibilities that have no prerequisites, and once the first structures are completed the number of possibilities increases. For more complex build lists, which easily involve several different units, six different structures, four Chrono-Boost types, and 30 actions (around 6 minutes in-game time), we have no chance to explore the whole search tree to find the optimum.

Alternatives are heuristics, for example branch and bound algorithms or genetic algorithms. Both options seem to be reasonable to us, we chose the latter because of the large search space. Of course, one can prune the search space as in [3] for a branch and bound algorithm, but then the found optimum can be far away from the global optimum. Additionally, we did not want to make too much use of build lists from game replays, such that our algorithm could also be applied to new strategy games in order to balance for example different races.

In general, the applied optimization algorithm needs to be tailored to three different issues:

- the *search space is extremely large* (e.g. 30^{30} possible candidates) in contrast to a *very small objective space* (e.g. one integer between zero and ten),
- *hard constraints* have to be fulfilled, and
- the number of actions is not known in advance for a certain strategy list.

In the following, we explain how we try to cope with these difficulties within single- and multi-objective genetic algorithms.

3.1 Single-objective Optimization

Genetic algorithms are search heuristics inspired by natural evolution. Genetic algorithms use populations of strings (chromosomes), which encode candidate solutions (individuals). Initially, all individuals are generated randomly. In a our genetic algorithm, the population undergoes the following steps during one generation:

1. *Forward simulation*: For each individual of the population, a forward simulation is performed with the inputs encoded by its chromosomes.
2. *Fitness evaluation*: Results of the forward simulations are evaluated as a fitness value that measures the quality of the chromosomes. The fitness of an individual is just the number of units of one certain type that can be produced during forward simulation.
3. *Selection*: The best few individuals of each generation are directly accepted for the next generation. Individuals with a higher fitness are more likely to be selected for the next generation, others are withdrawn.
4. *Reproduction*: New individuals are generated by the combination of two (or more) selected individuals. Parts of both parents' chromosomes are mixed together.
5. *Mutation*: Some genes (single entries of the string) of the new chromosomes are randomly changed.

Next we discuss in detail the single components of our genetic algorithm. Its objective function is to find the maximum number of units of one certain type that can be produced up to a certain in-game time.

Encoding Besides a variety of applications, genetic algorithms are a suitable choice in scheduling problems [26]. In principle, one of our build lists has similar features as a schedule. But we are not only interested in an optimal arrangement of given actions, but also to decide which actions and how many of them to put into the schedule. In addition, the amount of available resources depends on the number of workers for minerals and Vespin gas, and production facilities.

For our application it is thus a quite natural choice to use build lists as individuals (see Fig. 1). One entry of a build list is equal to one action, which leads to *integer encoding* of the chromosomes. The length of the chromosomes depends on how many actions can be performed within the simulated time and differs potentially for different individuals in the population. In general, "good" build lists tend to be longer. That is due to the reason that it is often beneficial to perform more actions during the simulation time. To give an idea, for five minutes in-game time a forward simulation typical needs string lengths of about 20 actions and more.

Initialization For the initial population we consider random individuals, which are created in such a way that all hard requirements are fulfilled for all actions. Thus, the hard requirements, which can be considered as constraints, are directly fulfilled by the encoding procedure of the build lists. This ensures that the forward simulation will not run into a dead end meaning that nothing will happen any more until the simulation time ends. Invalid strings are not able to provide a good genetic pool. Additionally, the user can insert custom build lists to the initial population. In the game community there exist large number of build lists provided by expert players and encoded similar to our build lists. These lists can help and accelerate the optimization process to converge towards an optimal solution.

Selection For selection all individuals are sorted according to their fitness. If two individuals are the same one of them is removed from the population. Among all solutions with highest fitness, we keep the following best candidates directly for the next generation:

- the candidate with the *most workers*,

Table 1 Comparison between random search and different genetic algorithms. Ten experiments were done for each case and the maximum number of Stalkers of the best of them is listed. We stop the genetic algorithm after 50 generations

Algorithm	Generations	Individuals/Generation	No. of Stalkers
Random search	1	10 000	3
	1	100 000	4
	1	1 000 000	4
Genetic	50	10 000	5
Genetic (distance-based selection)	50	10 000	6
Genetic (distance-based selection, mutation)	50	10 000	7

– the candidates with the *most production structures* (of each kind),
– and the candidate that *would earliest built an objective unit* after the finished simulation time.

Furthermore, we apply either fitness-proportional or truncate selection. In case of truncate selection only the best 1–10 % individuals of the whole population are chosen. This very high selection pressure showed up to be necessary to achieve good results. The reason might be that we have a very large search space compared to the objective space as mentioned before. In our simulation, the objectives are small integers (usually below 10), which means many solutions have the same objective value.

Reproduction To find new individuals we implemented an adapted uniform crossover. Starting from the first gene of both parent individuals, each gene is randomly chosen, if the choice does not violate any hard requirement. Otherwise the other gene is chosen. Although it is not guaranteed that a valid child is generated, this procedure significantly increases the chance. If two individuals have different length the shorter one is filled with arbitrary genes such that it stays valid.

We choose the parent individuals similar to binary tournament selection from the mating pool. Therefore, four individuals are chosen randomly. Then the distances between all possible pairs of these individuals are calculated. The pair with the largest distance is selected. We define a *distance* d, which is similar to a Hamming distance, of two individuals a and b with a string length of l by

$$d = \sum_{i=1}^{l} (l - i) \cdot (a_i - b_i). \tag{1}$$

The difference of two genes of the build lists $a_i - b_i$ is zero, if the genes are equal, and one otherwise. We decrease the effect of different genes w.r.t. the position linearly, and implicitly to the passed simulation time, because the variation of genes at the beginning of the string, i.e. at an early time stage, has a larger effect on the simulation than variation of genes at the end. The underlying idea is a similar to e.g. [19], but is placed before the reproduction in a computationally cheap way. We denote this special type of selection for the reproduction as *distance-based selection*, which helps to retain the diversity. If it is not used, the solutions equalize during a few generations due to our aggressive selection described above.

We also tried to compare strings with other distance metrics, like calculating a dice's coefficient [7] by bigram or trigram, or compare solutions based on the numbers of each object type. However, the first was less effective, the latter similar effective, when sorting and selecting the individuals in non-domination fronts, but much more expensive.

Mutation We apply a mutation operator to copies of the best solutions instead to individuals obtained by crossover, since mutating the individuals after reproduction, as done usually, had nearly no effect. 20 % of the new population pool is filled with such mutated solutions. This is similar to a step of tabu search [10, 20] without storing already visited solution. Mutated individuals are deleted after their evaluation, if they are not a best solution of the next generation.

Mutation by flipping two adjacent genes, changing genes, as well as inserting and deleting random genes was tested. The latter performed best and was able to improve the solution, when the population has already converged. One reason for this is that we shift the genes when inserting and deleting and thus the build times of many objects within the resulting individual change. This increases diversity. The mutation rate was set to 10 % for inserting and deleting, each. Such high mutation rates were shown to be rather effective for small population sizes [14]. One difference is that we do not perform a reproduction before. Especially for TechPushes our mutation is extremely helpful.

Comparison of Components and Parameters In a first test setup for our genetic algorithm we try to figure out how many Stalker units can be built in the first 400 seconds of the game. Within this setup, ten experiments were done for each of the following cases and the best of those runs is chosen. A simple random search within all possible build lists in the first row of Table 1 shows that one is able to find up to four of at least seven possible Stalkers in 10^6 runs. Here, whenever an action was started, we randomly choose a new possible action and wait until this action can be performed. The second row in Table 1 lists the improvement

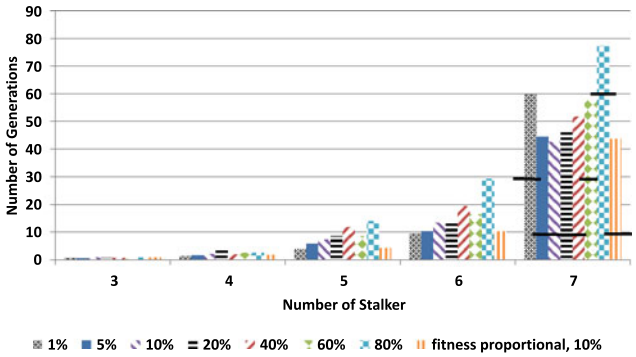

Fig. 2 Achievable number of Stalker units after a certain number of generations with varying selection parameters

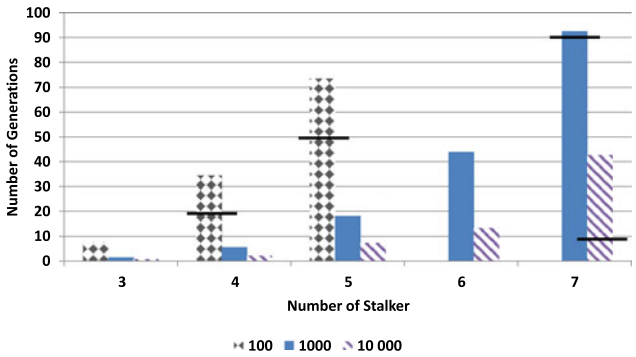

Fig. 3 Achievable number of Stalker units after a certain number of generations with varying number of individuals per generation

using a plain implementation of a genetic algorithm, with random initialization, forward simulation, fitness evaluation, truncate selection, and reproduction. The other two variants with distance-based selection and mutation illustrate the effects of both extensions as discussed in Sect. 3.1. For all variants we tuned the optimal selection pressure between 1–5 % manually. The complete genetic algorithm adds runtime overhead of roughly 30 % compared to a random search for a population of 10 000 individuals.

Next, we perform a parameter study for the different components. For the non-varying parameter we use a population size of 10 000 individuals, truncation selection choosing the best 10 %, add/remove mutation, and stop after 100 generations. The number of Stalkers of ten experiments are averaged. Figures 2, 3, and 4 depict the (arithmetic) average number of generations g_{av} over ten runs to achieve a certain number of Stalker units. If the displayed number of Stalker units could not be accomplished in an experiment after 100 generations, we count them like a run with 100 generations, this means that

$$g_{av} = \frac{1}{10}\left(\sum_{i=1}^{10} \min(100, g_{i,s})\right), \tag{2}$$

where $g_{i,s}$ is the number of generations it takes in experiment i to achieve s Stalkers. If there is a contribution of

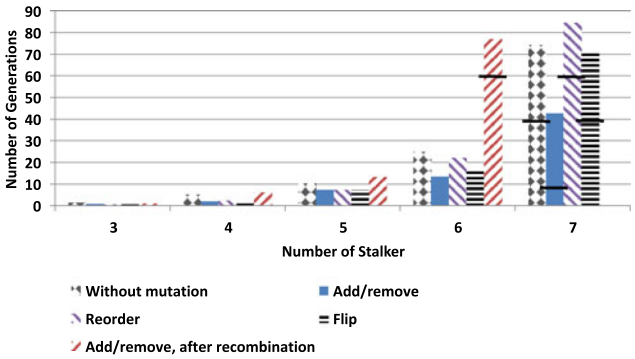

Fig. 4 Achievable number of Stalker units after a certain number of generations with varying mutation operators

failed experiments (where $g_{i,s} > 100$) to g_{av}, we incorporate a horizontal line that splits the bar into two parts. The part of the bar below the horizontal line g_{100} shows the contribution of failed experiments to g_{av}, the part above the horizontal line the contribution of successful experiments.

Consequently, the average number of generations g_{av}^{suc} when neglecting the failed experiments can be calculated by

$$g_{av}^{suc} = \frac{10}{10 - 0.1 g_{100}}(g_{av} - g_{100}). \tag{3}$$

The first bar with a bar in Fig. 4 is e.g.:

$$g_{av}^{suc} = \frac{10}{10 - 0.1 \cdot 60}(77 - 60) = 42.5. \tag{4}$$

A selection of 10 % turns out to be optimal with add/remove mutation. The mutation rate after recombination, as it is usually done in genetic algorithms, was set down to 1 % for Fig. 4 but shows a negative effect. Figure 3 suggests that an improved performance is achieved by relatively high population sizes. We choose these optimal parameters for our further experiments.

3.2 Multi-objective Optimization

In case of more than one objective we apply an adapted NSGA II algorithm. Here, the population is sorted by non-domination into Pareto fronts. An individual dominates another one, if all of its objective values are equal or better, and at least one of them is better than the objective values of the dominated one. Each individual of the second front is dominated by at least one individual of the first front. Any individual of the third front is dominated by at least one individual of the first or second front, and so on. A crowding distance (see below) is assigned to each individual, which quantifies the distance to its neighbors.

We apply the genetic operators as described above. The parents for recombination are selected based on the front and crowding distance combined with binary tournament selection.

However, with standard NSGA-II one encounters the following problems:

Algorithm 1 Penalize-distance (I)

1: $l = |I|$ {number of individuals in I}
{Initialize distance $I[i]_{distance}$ via NSGA-II crowding-distance-assignment algorithm}
2:
3: **for** $m = 0 \ldots (|m| - 1)$ **do**
4: $sort(I, m)$ {stable sorting using all different objective tuples}
5: **end for**
6: $\sigma = 0$ {initialize a penalty parameter σ}
7:
8: **for** $i = 1 \ldots (l - 1)$ **do**
9:
10: **if for each** m: $(I[i-1].m == I[i].m)$ **then**
11: $\sigma = \sigma + 1$ {increase the penalty for non-changing objective tuples}
12: **else**
13: $\sigma = 0$ {reset the penalty, when objective tuple changed}
14: **end if**
15: $I[i]_{distance} = I[i]_{distance} - |m| \cdot \sigma$ {apply the penalty to the distance}
16: **end for**

1. one does not obtain sufficient diversity preservation by the crowding distance assignment,
2. the maximal population size N is limited by the non-dominated sorting algorithm, which still has a complexity of $O(N^2)$.

The first issue arises from the fact that there are usually only a very small number of different objective values and many equal tuples. This is not taken into account explicitly by NSGA-II. To relax the second problem, the non-domination sorting is not applied to all individuals, but only to lists of individuals with the same objective tuples.

Crowding Distance The crowding distance calculation assumes that each individual represents a different objective tuple in our problem. Unfortunately, e.g. by the sorting processes, it is even likely that all chosen individuals have an equal objective tuple. To penalize individuals with equal objective tuples, we extend the NSGA-II crowding-distance-assignment algorithm, by calling Algorithm 1 afterwards. I is a list of individuals in one front. An objective tuple m corresponds to one or more individuals of a front.

If several equal objective tuples exists, each additional individual gets a growing negative distance. Please note, Algorithm 1 requires a stable sorting. Instead of lines 3–5, one can sort the list of individuals I according to the objective tuples once. An applied penalty is always stronger than a later added (positive) crowding-distance, apart from objective boundary points.

Non-domination Sorting In a first step all solutions with an equal objective tuple are collected in one list. Then, these lists are sorted instead of the single individuals by a non-domination sorting algorithm. In a typical setup (e.g. the experiment in the next section) one has to sort around 50 lists instead of several thousand single individuals. This modification allowed us to increase the population size beyond around 5000 individuals. For problems without such small objective spaces, we refer to a more general approach [17]. Large population sizes are especially important for multi-objective optimization, since we optimize a whole Pareto front of individuals instead of a single individual. In addition to the first front, the best individual of each front is taken for selection of the best. This is quite cheap, since there are not many different objective values.

4 Implementation Details

4.1 Software Features

A major goal was to develop a toolbox, where users can easily adapt parameters of the forward simulation and optimization process. Without programming background or knowledge in optimization users should be able to create and optimize their builds. The implementation of the toolbox was done in the C# programming language using Microsoft Visual Studio.[5] Here, main focus was put on productivity and modern object oriented software design, but not on most efficient code. A graphical user interface (see Fig. 5) is provided to guide the user through the optimization process and visualize the output. The user can for example reduce the search space by selecting only a subset of the Tech-Tree to be considered in build lists or adjust the simulation time and the objective function set. Experienced users can also modify the optimization strategy and the parameters for the genetic algorithm. Apart from some special mechanisms (e.g. Chrono-Boost), properties like costs or Tech-Tree requirements of all structures, units, and upgrades can be adjusted via input text files.

4.2 Parallelization

In order to speed up the optimization process we also ported our code to C++ and parallelized it using MPI (Message Passing Interface) [13] provided by Microsoft HPC Pack 2008 SDK. The single-objective genetic algorithm can be parallelized easily. Instead of one population of size N on one single compute process we have P different populations of size N/P, which we call colonies, on P compute

[5] http://www.microsoft.com/germany/visualstudio/.

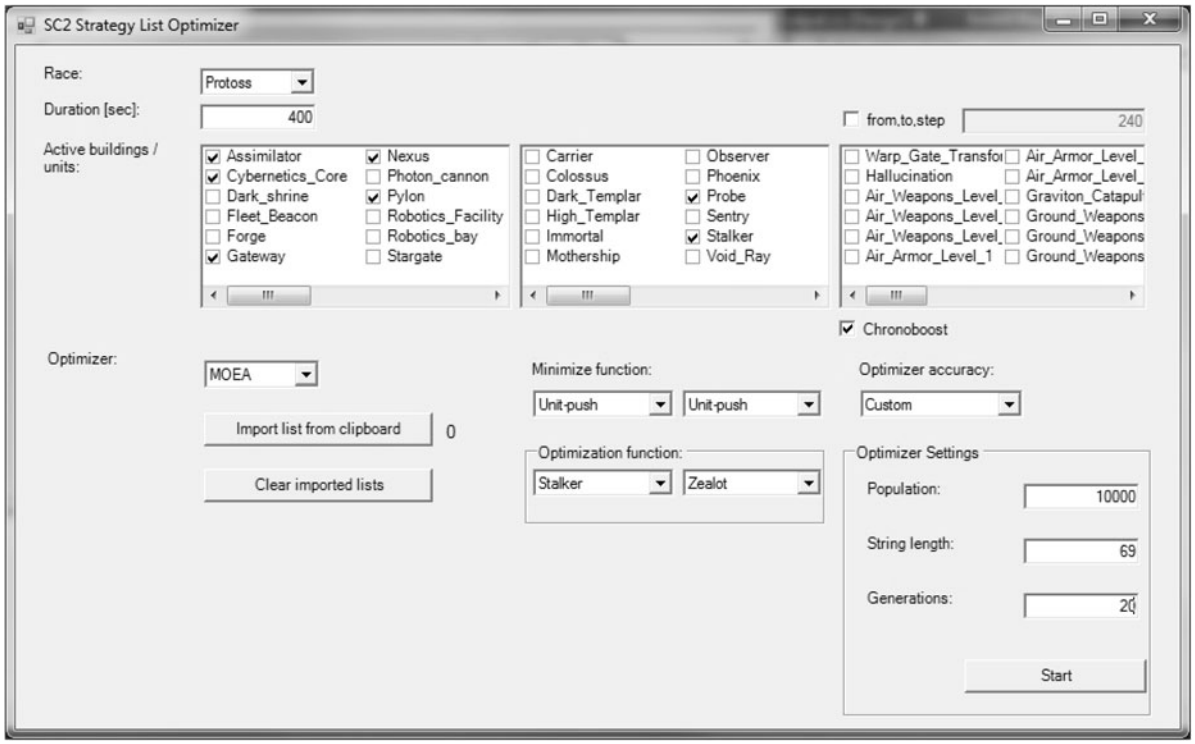

Fig. 5 Graphical user interface of toolbox. With the shown settings a Protoss Rush (Unit-Push) aiming for a maximum number of Zealots and Stalkers at 400 seconds in-game time can be optimized by our multi-objective evolutionary algorithm (MOEA)

processes. Forward simulation, fitness evaluation, reproduction, mutation, and selection of the best can be done completely locally without any communication between different compute processes. For evaluating the global fitness of all colonies, the maximum local fitness is sent to all processes via an *MPI_Allreduce* once per generation. Then, each process communicates to all other processes its current local average fitness via *MPI_Allgather*. After that, all processes with low average fitness compared to the overall global maximum fitness add a fixed number of individuals to their colony, which they receive from randomly chosen processes with relatively high average fitness. Here, we use nonblocking communication via *MPI_Isend* and *MPI_Recv*. This strategy of exchanging the best global individuals leads to similar local average fitness after a few generations.

The main advantage of the parallel version is the run-time reduction. For example in Sect. 3.1 the run-time for one generation of the genetic algorithm with full features (last row of Table 1) was for our C# code about 1.1 s on one of four cores of an Intel Core i7-3770 at 3.4 GHz. The C++ code took for the same settings about 0.8 s on one core, 0.4 s on two cores, 0.26 s on four cores with one thread per core, and 0.21 s on four cores with two threads per core.

The parallel version of the multi-objective genetic algorithm does not achieve the expected benefit up to now, i.e. it requires more generations than the serial version, so we do not discuss it here in detail.

5 Experimental Results

For the results produced within our toolbox it is assumed for the simulator that for all builds, one worker does not earn minerals, but is reserved for building. If an Assimilator, a Refinery, or an Extractor (the structure type of Protoss, Terran, and Zerg for mining Vespin gas) is finished, automatically three of the workers are assigned to it in order to mine Vespin gas. Furthermore, we assume due to in-game measurements an average rate of 0.714 minerals or 0.65 Vespin gas per worker and second.

5.1 Tech-Pushes

As an example for a Tech-Push, we search the earliest possible time to build a Void Ray unit. Note that Tech-Pushes tend to have only few genes compared to the game time, since a good economy with e.g. many workers is not so much important as for Rushes. The obtained optimal build list is shown in Table 2.

A population of 10 000 individuals over 200 generations and a maximal simulated time of seven minutes turned out to be sufficient. The longest chain of dependencies (Pylon, Gateway, Cybernetics Core, Stargate, Void Ray) gives one theoretical lower bound for the time, which is exactly 4:00 minutes. The remaining objects (Probes, i.e. workers, and Assimilators) have to be built to deliver enough resources.

Table 2 Optimized build order for Void Ray Tech-Push. The times (mm:ss) denote the finishing times of the corresponding object type

Probe	Probe	Probe	Probe	Pylon	Gateway	Probe	Probe
0:17	0:35	0:52	1:09	1:30	2:36	1:56	2:13
Assimilator	Assimilator	Cybernetics Core	Probe	Stargate	3 × Boost of Stargate		Void Ray
2:29	2:40	3:27	3:00	4:28	4:29		**5:08**

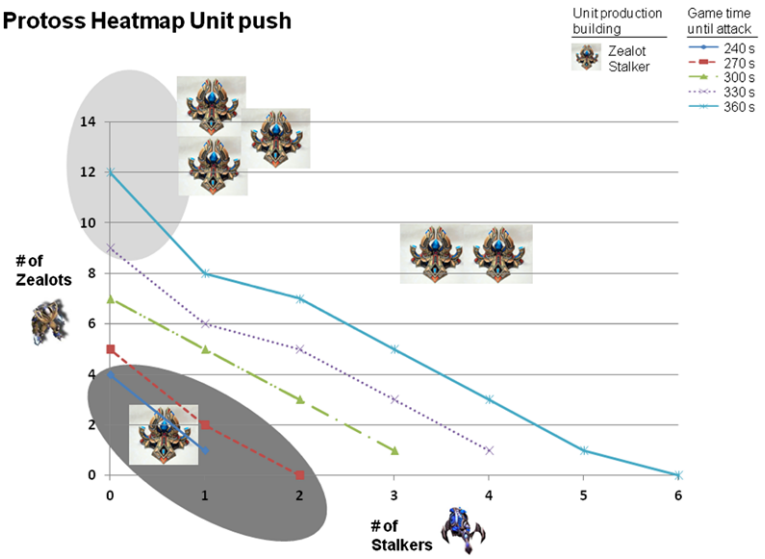

Fig. 6 Heatmap of the production of Zealot and Stalker units. The displayed regions in the map mark the number of necessary production facilities

The lack of resources is responsible for the impossibility to build the target unit in a direct way. Some individuals also try to accelerate the production of workers at the beginning with a Chrono-Boost, but this did not lead to a better minimal time to solution.

5.2 Rushes

In a Rush, players try to get as many as units of one or more different types as possible in a prescribed time. We optimize for two different unit types and apply the multi-objective genetic algorithm for Rushes in the following. Among the best build lists that were found, those which include most workers are presented. We also played some build lists in order to validate our forward simulation and cross-checked, if the results are reasonable via comparing them to build lists used by experienced Starcraft II players.

Protoss We start with the Protoss race. A multi-objective optimization run with 40 000 individuals and 10 000 generations was done in order to get as close as possible to the Pareto-optimal front. Increasing the number of generations by a factor of 10 compared to [11], we were able to improve few points in Figs. 6 and 7. Although the population seems to be converged after 10–20 generations, even after several hundreds of generations we are occasionally able to improve the best solution due to our mutation operator described above. One compute core of the Intel Core i7-3770 needs several hours for one optimization run.

Figure 6 depicts the number of produced Stalkers and Zealots after 240–360 seconds. One Zealot costs 100 minerals and does not require Vespin gas and consequently no Assimilator structures. Hence, the builds without Stalkers include no Assimilators. One Stalker costs 125 minerals and 50 Vespin gas. The fronts with 330 and 360 seconds require two Assimilators for the two builds having most Stalkers, all remaining lists suggest one Assimilator. Zealots and Stalkers are produced in Gateways. Their optimal number is a trade-off between their costs and the additionally provided production capacities. The number of produced workers (Probes) lies between 2 and 14. It depends on the simulated time, but not on the produced unit combination in a clear trend. Outstanding points, like $(\mathit{Zealots}, \mathit{Stalkers}) = (12, 0), (1, 5), (0, 6)$ are much more difficult to find for the genetic algorithm, since they can be reached only very few seconds before the limiting time. Only an extremely small number of candidates allow those objective values.

The two solutions with zero and one Stalkers at 279 seconds are obtained after a few generations. However, it is very difficult to find an individual producing two Stalkers only. One reason is that the solution with two Zealots, one Stalker, and one additional Stalker in production that has

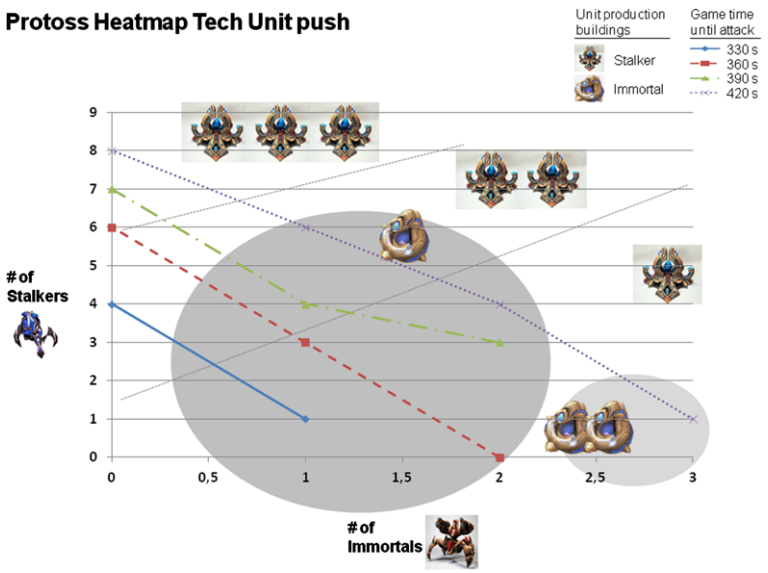

Fig. 7 Heatmap of the production of Immortal and Stalker units. The displayed regions in the map mark the number of necessary production facilities

Table 3 Maximum number of Stalkers, Zealots, and Immortals that can be produced within 360 s and their costs. For the army value we add all required resources and weight minerals and Vespin gas 1:2

# Zealots	# Stalkers	# Immortals	Minerals	Vespin gas	Army value
12	0	0	1200	0	1200
5	3	0	875	150	1175
0	6	0	750	300	**1350**
0	3	1	625	250	1125
0	0	2	500	200	900

not been finished is much farther than the candidate without Zealots, one Stalker, and an additional Stalker in production. However, the latter variant is not a member of the first front. Therefore, we take the best solutions of all fronts, which we can easily afford due to the few different objective values.

Figure 7 presents the number of produced Stalkers and Immortals after 330–420 seconds. One Immortal costs 250 minerals and 100 Vespin gas and is constructed in a Robotics Facility. The lists of the more advanced builds have around 40 genes and up to 12 build options per gene and thus are much longer than in the last example. A Tech-Push showed that the first Immortal can be produced after 325 seconds. Note that one requires at least one Gateway and a Cybernetics Core in every build list, since they are prerequisites for a Robotics Facility.

For players it is now interesting to compare various opening strategies for a given time. Exemplary in Table 3 the best computed builds at time 360 s are summarized. It suggest that a good in-game strategy is to attack the enemy after 360 s with six Stalkers. This seems to be reasonable, because Zealots can be built quite early in the game and are most effective when attacking at 240–300 s, and Immortals have more prerequisites and can be used more effectively in later game stages. Of course, the computation of the army value is not very significant for real games. Here, much depends on micro-management of the army. But the computed optimal build lists can help to improve resource macro-management and plan the Rush timing in the early game stage.

Zerg For Zerg, the technological analog to Zealots and Stalkers are Zergling and Roaches. Figure 8 visualizes the results for Rush times with them between 240–360 seconds. For the multi-objective optimization we choose 40 000 individuals and run 500 generations. Since one can only build Zergling pairs, we in fact count the number of Zergling pairs in our graph, and one of them costs 50 minerals, one Roach 75 minerals and 25 Vespin gas. An interesting observation is that one is able to rediscover standard strategies like the seven Roaches Rush (point $(Zergling, Roaches) = (0, 7)$) at 300 s in-game time. Note that Zerg build orders tend to be longer than these of the other two races, because overall more units can be produced. One reason is that Zerg units are relatively cheap and another that a Queen unit can increase unit production by injecting larvae. In contrast to the other two races units are not produced in structures but morphed from larva units. When the game starts, each Zerg player has three larvae, a new larva is injected automatically from the main base structure every 15 s.

The army values for a Rush at 360 s are shown in Table 4. Similar to Table 3 for the Protoss race the best strategy

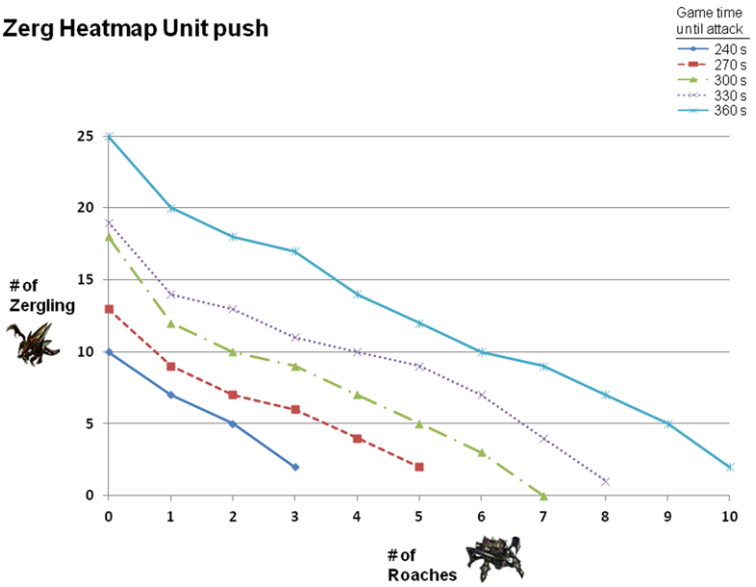

Fig. 8 Heatmap of the production of Zergling and Roach units

Table 4 Maximum number of Zerglings and Roaches that can be produced within 360 s and their costs. For the army value we add all required resources and weight minerals and Vespin gas 1:2

# Zergling	# Roaches	Minerals	Vespin gas	Army value
25	0	1250	0	1250
18	2	1050	50	1150
10	6	950	150	1250
5	9	925	225	**1375**
2	10	850	250	1350

seems to be a mixed Zergling and Roaches built. Even the overall maximal army value of 1375 is close to the Protoss result of 1350, what points out the good balancing of the different races. However, the unit count is higher for Zerg.

Terran As an example for a Terran Rush we choose Marines and Marauder units, since they their required Tech-level is comparable to Zealots and Stalkers. One Marine costs 50 minerals and one Marauder 100 minerals and 25 Vespin gas. In Fig. 9 army size results for Rush times between 240–360 seconds of a combined Marine-Marauder production are depicted. For the multi-objective optimization we choose 40 000 individuals and run 500 generations. If one looks at the army values for example at time 360 s (see Table 5), the maximal army value is lower than for Protoss and Zerg. A possible reason is that both Zerg and Protoss make use of their special abilities Queen larva injection and Chrono-Boost within the found optimal build lists. In contrast, The found optimal Terran build lists for shorter simulation times than 360 s do not attach an Orbital Command to the Terran main base structure and thus are not able to create MULEs for an improved minerals income. An explanation for that could be that for shorter in-game times it is more beneficial to build additional workers instead of the Orbital Command update in the main base structure. Additionally, the number of production facilities is a limiting factor for Terran for a pure Marines Rush. Marines are cheaper than Zealots (ratio 1:2), but the build times are not shorter in the same way (ratio 25:38). Thus, in general a Terran requires more production facilities, but the costs for them are the same as for Protoss. However, for the game this does not mean necessarily a disadvantage for the Terran, since Marines have in contrast to Zealots a ranged attack and as mentioned before it depends on unit micro-management within combats how valuable an army really is.

6 Conclusions and Future Work

We presented a multi-objective genetic algorithm to find optimal build orders for players of real-time strategy games and especially Starcraft II. Most problem dependent algorithmic adjustments help to cope with large search spaces (e.g. 10^{30} possible candidates) in contrast to very small objective space (e.g. one integer between zero and ten). We applied the optimization to all three Starcraft II races with up to 450 seconds simulation time. The resulting build orders of achievable units should be a valuable tool both for new and experienced players and help to testify and improve their build strategies.

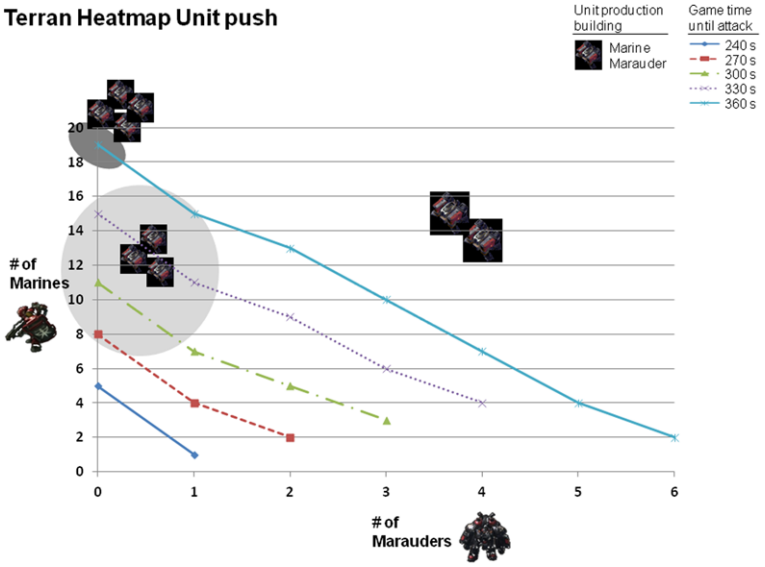

Fig. 9 Heatmap of the production of Marine and Marauder units. The displayed regions in the map mark the number of necessary production facilities

Table 5 Maximum number of Marines and Marauders that can be produced within 360 s and their costs. For the army value we add all required resources and weight minerals and Vespin gas 1:2

# Marines	# Marauders	Minerals	Vespin gas	Army value
19	0	950	0	950
15	1	850	25	900
7	4	750	100	950
4	5	700	125	950
2	6	700	150	**1000**

Currently, we are working on the parallelization of the multi-objective genetic algorithm in order to reduce the long optimization times that can be several hours on one CPU core. Related to that is the plan to include our algorithm in a StarCraft II AI. Here, one does not need to search the full space each time step but one can restart the optimization at a certain in-game time with the current state of the AI. It is then also useful to compare our approach to the branch and bound algorithm from [3] adapted to StarCraft II. Our multi-objective genetic algorithm has to be tuned more for the three different races, while we are now quite satisfied with the Protoss results, the results for Terran and Zerg could perhaps still be improved. Finally, we will update our toolbox to be able to optimize build orders also for the upcoming Starcraft II: Heart of the Swarm.

References

1. Beume N, Naujoks B, Emmerich M (2007) Sms-emoa: multiobjective selection based on dominated hypervolume. Eur J Oper Res 181(3):1653–1669
2. Cantú-Paz E (1998) A survey of parallel genetic algorithms. Calc. Parallèles 10(2):141–171
3. Churchill D, Buro M (2011) Build order optimization in Starcraft. In: Proceedings of AIIDE, pp 14–19
4. Davis L, Mitchell M (1991) Handbook of genetic algorithms. Van Nostrand Reinhold, New York
5. Deb K, Goel T (2001) Controlled elitist non-dominated sorting genetic algorithms for better convergence. In: Evolutionary multi-criterion optimization. Springer, Berlin, pp 67–81
6. Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multiobjective genetic algorithm: Nsga-ii. IEEE Trans Evol Comput 6(2):182–197
7. Dice L (1945) Measures of the amount of ecologic association between species. Ecology 26(3):297–302
8. Fernández-Ares A, Mora A, Merelo J, García-Sánchez P, Fernandes C (2011) Optimizing player behavior in a real-time strategy game using evolutionary algorithms. In: 2011 IEEE congress on evolutionary computation (CEC). IEEE, New York, pp 2017–2024
9. Fonseca C, Fleming P (1993) Genetic algorithms for multiobjective optimization: formulation discussion and generalization. In: Proceedings of the 5th international conference on genetic algorithms. Morgan Kaufmann, San Francisco, pp 416–423
10. Glover F, Laguna M (1998) Tabu search, vol 1. Springer, Berlin
11. Gmeiner B, Donnert G, Köstler H (2012) Optimizing opening strategies in a real-time strategy game by a multi-objective genetic algorithm. In: Research and development in intelligent systems, vol XXIX, pp 361–374
12. Goldberg D (1989) Genetic algorithms in search, optimization, and machine learning. Addison-Wesley, Reading
13. Gropp W, Lusk E, Skjellum A (1999) Using MPI: portable parallel programming with the message passing interface, vol 1. MIT Press, Cambridge
14. Haupt R (2000) Optimum population size and mutation rate for a simple real genetic algorithm that optimizes array factors. In: Antennas and propagation society international symposium, vol 2. IEEE, New York, pp 1034–1037. doi:10.1109/APS.2000.875398

15. Haupt R, Haupt S, Wiley J (2004) Practical genetic algorithms. Wiley Online Library
16. Horn J, Nafpliotis N, Goldberg D (1994) A niched Pareto genetic algorithm for multiobjective optimization. In: Proceedings of the first IEEE conference on evolutionary computation. IEEE world congress on computational intelligence. IEEE, New York, pp 82–87
17. Jensen M (2003) Reducing the run-time complexity of multiobjective eas: the Nsga-ii and other algorithms. IEEE Trans Evol Comput 7(5):503–515
18. Knowles J, Corne D (2000) Approximating the nondominated front using the Pareto archived evolution strategy. Evol Comput 8(2):149–172
19. Mauldin M (1984) Maintaining diversity in genetic search. In: Proceedings of the national conference on artificial intelligence, vol 247, p 250
20. Misevicius A (2004) An improved hybrid genetic algorithm: new results for the quadratic assignment problem. Knowl-Based Syst 17(2):65–73
21. Othman N, Decraene J, Cai W, Hu N, Low MYH, Gouaillard A (2012) Simulation-based optimization of Starcraft tactical AI through evolutionary computation. In: 2012 IEEE conference on computational intelligence and games (CIG). IEEE, New York, pp 394–401
22. Rathe EA, Svendsen JB (2012) Micromanagement in Starcraft using potential fields tuned with a multi-objective genetic algorithm. Ph.D. thesis, Norwegian University of Science and Technology
23. Shantia A, Begue E, Wiering M (2011) Connectionist reinforcement learning for intelligent unit micro management in Starcraft. In: The 2011 international joint conference on neural networks (IJCNN). IEEE, New York, pp 1794–1801
24. Synnaeve G, Bessiere P et al (2011) A Bayesian model for plan recognition in rts games applied to Starcraft. In: Proceedings of AIIDE, pp 79–84
25. Togelius J, Preuss M, Beume N, Wessing S, Hagelbäck J, Yannakakis GN (2010) Multiobjective exploration of the Starcraft map space. In: Proceedings of the IEEE conference on computational intelligence and games (CIG), pp 265–272
26. Wall M (1996) A genetic algorithm for resource-constrained scheduling. Ph.D. thesis, Massachusetts Institute of Technology
27. Wang Z, Nguyen KQ, Thawonmas R, Rinaldo F (2012) Monte-Carlo planning for unit control in Starcraft. In: 2012 IEEE 1st global conference on consumer electronics (GCCE). IEEE, New York, pp 263–264
28. Zitzler E, Thiele L (1999) Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. IEEE Trans Evol Comput 3(4):257–271