GENERIC FRAMEWORK FOR MUTLI-OBJECTIVE DESIGN SPACE EXPLORATION FOR DYNAMICALLY RECONFIGURABLE SYSTEMS

by

Irina Ivanova

B.A.Sc. University of Toronto, 2005

B.Ed. University of Toronto, 2007

A Master Thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Applied Science

in the Program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2013

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.
I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Signed: _____ Date: _____
Irina Ivanova

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

Signed: _____ Date: _____
Irina Ivanova

GENERIC FRAMEWORK FOR MUTLI-OBJECTIVE DESIGN SPACE EXPLORATION FOR DYNAMICALLY RECONFIGURABLE SYSTEMS

Master of Applied Science, 2013
Irina Ivanova
Electrical and Computer Engineering Program at Ryerson University

Abstract

In recent years the Run-Time-Reconfigurable (RTR) computing systems have become the core of next generation of adaptive embedded systems. One of the major problems in this class of systems is run-time adaptation of their architecture to the dynamic workload and environmental conditions. In most cases this adaptation is considered as multi-objective optimization process which should be conducted in run-time.

Therefore, the goal of this research work was to explore the existing methods of doing multi-objective optimization and analyze their applicability for a system with potential of reconfiguration (i.e. a situation when constrains of the system can change during the course of operation). Then the development of generic framework of this optimization mechanism has to be done. This required analysis and selection of proper approach for multi-objective space exploration. The methodology based on Architecture Configuration Graph was chosen and its searching technique improved to allow faster convergence to a solution that satisfies objective constraints while optimizing specified objective. The run-time complexity analysis was done for modified methodology as well as the testing of the implemented framework to demonstrate its faster performance. The experimental results have shown the ability for run-time architecture adaptation and further utilization of the proposed framework as a core of real-time operating systems (RTOS) for dynamically reconfigurable computers.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ACG = Architecture Configuration Graph

EA = Evolutionary Algorithm (an umbrella term for a heuristic that evolves a set of solutions over simulation run)

GA = Genetic Algorithm (a more specific evolutionary algorithm: solutions are encoded in a similar fashion as chromosomes and during simulation run solutions produce better offspring solutions at each iteration)

MOP = Multi-Objective Optimization Problem

MOCO = Multi-objective Combinatorial Optimization

MOPSE = Multi-Objective Particle Swarm Optimization

PSO = Particle Swarm Optimization

SA = Simulated Annealing

SPEA = Strength Pareto Evolutionary Algorithm

TSP = Travelling Salesman Problem

VLSI = Very Large-scale Integration

I . Introduction

Motivation

The rapid progress in development of Dynamically Reconfigurable Computing Systems (DRCS) made them a core of the embedded Systems-on-Chip (SoC). These systems are based on partially reconfigurable Field Programmable Gate Array (FPGA) devices and can provide very large set of run-time reconfigurable computing resources. Therefore, the rapid synthesis of on-chip architectures optimized for the task and performance requirements became one of the most important problems in this area of research. On the other hand, it is clear that effectiveness of the DRCS directly depends on the quality of dynamic architecture-to-task adaptation. At this time the above RCS architecture synthesis and multi-objective optimization on the workload specifics is the task of system architect or designer. Existing formal methods usually consider mono-objective performance optimization or multi-objective but not run-time optimization approaches. A few techniques have been developed for performing efficient optimization of multi-parametric design methods. However, when design space reaches certain complexity level all design combinations cannot be reasonably considered. Above all, everything that has been done so far in this area has an objective of finding a set of optimal architectural solutions in a "reasonable" time, not the run-time. This is acceptable if the design space is searched only once at the design stage. However, if the design stage is repeated through the life of the system (because this system reconfigures itself to changing parameters), then the time it takes to find the new optimal solution is critical. On the other hand, the run-time architecture-to-task adaptation for high-performance computing system means self-adaptation. In other words, the decision for reconfiguration on new architecture should be done automatically without any designer's input.

Thus, the proposed research is focused on analysis and creation of a methodology and associated framework of the systems which allow run-time architecture-to-task adaptation.

Objective

The main objective of the research is to develop a framework of the system which will allow run-time multi-objective adaptation of the reconfigurable architecture on the dynamic variations of the workload and conditions of operational environment that may require adjustment of performance parameters.

The objective of this work consists of the following aspects:

1. Analyze existing approaches in design space exploration to choose the most effective one to be applied in a run-time multi-objective optimization. Analyze the notable improvements proposed for various algorithms.
2. Develop the framework for software system that provides automatic synthesis of architecture in accordance with task, specified performance constrains and available resources. As part of framework development, refine some of the algorithm details proposed in [29] and originally [29] and conduct the algorithm complexity.
3. Implement the developed framework as a prototype software system and test its functionality.
4. Analyze the effectiveness of developed framework on a series of experiments testing various aspects of run-time architecture configuration.

Original contributions and Thesis organization

1. This work intends to contribute a comprehensive review of available Design Space Exploration (DSE) methods. Specifically, the review seeks to specify which classes of problems a particular method has been applied to; what are its benefits and shortcomings in the light of its application to an unmanned reconfigurable multi-objective system and, where authors have provided enough information, what is method's computational complexity. All major improvements are to be covered and analyzed as well. This work is done in Chapter II.

2. With the results stemming from research of different Multi-Objective Optimization (MOP) solution methods, Architecture Configuration Graph (ACG) method is chosen as the most suitable one for the task of run-time multi-objective optimization expressed as set of constraints to be satisfied. Few of the proposed improvements to this method are

considered in detail and the upper bound run-time complexity is analyzed. Furthermore, since ACG method has not had run-time complexity done for all steps, the detailed run-time complexity analysis is done. Additionally, few smaller improvements (speed-ups) are proposed as well as specific algorithmic steps are presented in higher detail. This is the work at the heart of Chapter III.

3. On the base of comprehensive analysis of ACG methodology the framework of the DSE system is developed. Some constant factor improvements are offered, though not implemented, that are possible in the framework to achieve minor speed-up and possibly memory usage improvement. The framework's usage in a presence of changing objectives and constrains is also discussed. The details of implementation and design decisions are covered in Chapter VI.

4. The implemented framework's effectiveness is analyzed through a series of tests that compare exhaustive search (as an ultimate reference point), original method and the proposed modified ACG method. The test design, goals and results are presented and analyzed in Chapter V.

5. Completing the work is Chapter VI that contains a set of conclusions and outline of future work, such as possible extensions to the framework, additional research that should be carried out to further improve the ACG algorithm and its implementation.

II. Literature Review

The attempts for creating the automated decision making methods based on design space exploration have a long history. Multi-objective decision making is implemented in the areas such as high-level synthesis of computing systems; expert systems; intelligent robotic system and many others. Other areas such as economics and finances deal with multi-objective optimization too and some methods were originally developed in those areas.

During the last three decades several different approaches for design space exploration and optimization of multi-objective discrete Pareto space have been researched and developed. In this chapter the analysis of the most advanced approaches to the above problem are presented, possible areas of application are defined and the pros and cons of each are determined.

Many of the presented techniques have given a raise to numerous improvements on top of the original idea. Some of the most notable improvements are also discussed in this chapter. Best effort has been made to provide a comprehensive overview of all optimization algorithms as potentially applicable to the scope of multi-objective optimization, major methods were chosen to illuminate the trends and implications of different optimization techniques.

It has to be noted that this section reviews "extended" techniques that can be applied to MOP. Majority of techniques have started off as optimizing for one objective and initially would be extended for MOP by a simple way, such as weighted sum function. For a review of nearly all (up to the year of 2006 at least) evolutionary algorithms and their comparison, please refer to [6] or [55]. It should also be noted that the algorithms discussed below (with exception of first and last) are very often partially combined together to tackle specific problems better.

The following terms are often used when talking about optimization. These are general term paraphrasing, not formal definitions, aimed at helping to understand the literature review. The formal definitions pertaining to both continuous and discrete multi-objective optimization can be found in Chapter 2 and 3 of Multiobjective Optimization, Interactive and Evolutionary Approach [40], [9]. Also, the majority of the resource that were drawn upon to form this review contain their own formal definitions of what constitutes a multi-objective optimization problem, how to specify Pareto-optimal solution, etc.

Pareto optimal solution – a feasible solution to a multi-objective problem such that none of its parameters can be further improved without making at least one of the other parameters worse. For the purpose of multi-objective optimization, the terms optimal and Pareto-optimal are equivalent, because when trying to satisfy multiple objectives the optimal solution is such that it offers the best trade-offs between objectives, which is another way of describing Pareto optimality. Also, with the exception of exact methods, none of the heuristics promise finding even some of the Pareto-optimal solutions (except extreme cases of optimizing one objective at expense of all others), and as such a heuristic usually finds a close-to Pareto-optimal solution(s) and only sometimes truly optimal. Although very often one does not know anyway that a solution is a true Pareto-optimal solution. Therefore, for the scope of this work, these terms (optimal = Pareto-optimal = close-to optimal) are used interchangeably as applicable to each method.

Pareto dominance – one solution is said to dominate another if it is strictly better at satisfying at least one (or more) objective and as good as another (solution) at satisfying the rest of objectives. This is formally called weak dominance. However, since for the purpose of this research only this notion is used, it is called simply dominance. Obviously, not all solutions can be ordered according to Pareto dominance, as there are many pairs of solutions such that solution A is better than solution B for first objective, and solution B is better than solution A for second objective. In this case neither solution dominates the other and are said to be non-dominated.
Some algorithms reviewed below use the notion of Pareto dominance to decide which solutions that get generated during simulation are to be kept for later as being so far the superior ones.

Pareto front – a set of all non-dominated solutions. Generally, most of the algorithms seek to generate a sub-set of solutions belonging to Pareto-front or a set of solutions that are very close to it (within some set threshold).

Supported solutions – solutions that a given algorithm is capable of finding. With many iterative algorithms no matter how long they are run for some problems some points in the Pareto front will remain undiscovered.

Multi-objective optimization – is a task of finding a solution that simultaneously optimizes more than 1 objective, where each of the objectives is either maximized or minimized. Formally, as stated in [9]:

$$\left.\begin{array}{ll} maximize/minimize\ f_m(m), & where\ m = 1,2 \dots M; \\ subject\ to\ g_i(x) \geq 0, & where\ j = 1,2 \dots J; \\ h_k(x) = 0, & where\ k = 1,2 \dots K; \\ x_i^{(L)} \leq x \leq x_i^{(U)}, & where\ i = 1,2 \dots n. \end{array}\right\}$$

A solution x $\in$ R^n is a vector of *n* decision variables: x = $(x_1, x_2, \dots, x_n)^T$.

Multimodality (of a problem) – a problem has many local Pareto-optimal fronts and only 1 global. Multimodal problems are more tricky to optimize because many heuristics, once having landed in an area of a local optimum, deviate to the local best. This is why many of the below mentioned heuristics have to be so concerned with notion of diversity – that is to maintain at all times a fair probe of all parts of solution space.

Multi-objective techniques can be viewed as consisting of two parts: searching and decision making, based on that the techniques can be classified as following (as per [39]):

1. No-preference – methods that solve the problem and give a decision directly to the end user. In a sense, both search and decision making is embedded in the technique.
2. Posterior (decision making after search) – give all/some decisions on/close-to Pareto-optimal front to the end user and let the user make the decision. So, effectively, such techniques implement only the searching part.
3. Priori – certain decision is done before the commencement of the search and thus user preference is constantly applied during the search and at the end one acceptable solution is found.
4. Interactive – hybrid of posterior and priori (user or an automated a decision system is involved after every round of simulation).

II.1. Classes of Optimization Problems

Below are some of the general problems, NP-complete, and NP-hard problems that have been solved with various optimization algorithms discussed below. This list contains the problems

that have been actually mentioned in the reference literature. Theoretically, the NP-complete problems are reducible to each other in polynomial time (as shown by Richard Carp), however, they have been traditionally tackled by different methods. It should also be noted that some optimization problems require finding of one global optimum while others require a sampling of near optimal solutions points (from the human point of view). Below, the problems are classified in similar way as in [19].

Ordering Problems

- Travelling Salesman Problem (TSP) /Hamiltonian cycles (for example, network and chip routing)
- Job shop/flow shop scheduling problems (JSP) and resource constraint project scheduling problem (JCPSP), can also be called Job Sequencing
- Vehicle scheduling problem. Similar to TSP, but different methods have been applied to the two so far [2]

Subset problems:

- Knapsack Problem

Assignment Problems:

- generalized assignment problem (GAP)

Grouping Problems:

- Bin packing problem
- Graph colouring problem

II.2. Exact Solution Methods

Below is an extremely condensed mention of some of the classical methods applied to multi-objective optimization. They are covered in detail in [39].

Weighted sum method is a popular and a very simple posteriori method when objective (normalized) functions are multiplied each by its own weight value and summed together to produce one function which is then optimized by one of the classic methods such as gradient method (broadly speaking as there are many specific ways to optimize a function

by analyzing its gradient), linear programming, Newton method, etc., depending on the function type. Also, a heuristic such as simulated annealing could be used for optimization.

Obviously, this method is only as good as the weight assignment (which might need to change in dynamic environment) and not good at all when an objective function is itself a program. Also, it can be only applied to convex problems and even for those, it finds only so called "supported" solutions, i.e. those lying in the convex regions of the objective space. [17]

ε-Constraint and elastic posteriori method minimizes one of the objectives and the other objectives are converted to constraints by setting an upper bound to each of them (for the remaining objectives). It can be applied to convex, non-convex and discrete optimization problems [17].

However, at this point large-scale multi-objective optimization problem cannot be solved efficiently with weighted sum or ε-constraint method [16]. Some remedies have been suggested such as elastic constraints that were shown to be effectively applied in a practically relevant size, albeit with only 2 objectives being optimized [18].

There is also an issue noted in [16] for both ε-constraint and weighted sum methods there is no generalization for problems of 3 and over objectives.

Both methods have been applied to problems such as knapsack, spanning trees, TSP, etc. [16]

Goal programming is a fairly simple concept. It operates on idea of a goal or aspiration level, which could be inflexible and then it's a constraint or it could be flexible. Then either a weighted sum of deviations of objective functions from goals is optimized or functions are ordered by importance and optimized in that specific order (this is called lexicographic ordering). The resulting function can then be optimized by linear programming [39]. Usually, a few iterations of the above steps are made with varied weights to see if a better solution can be achieved. Obviously, there are issues of designer involvement in choosing the weights (or objective orders) well and weight adjustment process may need to be repeated a few times before achieving a solution that is acceptably close to goal levels. The benefits of goal programming are its simplicity, its ease of understanding and application, as well as its familiarity (as it has been around for a few decades) [39].

Brach-and-bound is another exact solution method capable of optimizing a multi-objective combinatorial optimization problem expressed a set of constrains. This is also a very old method, known for more than 50 years. This method is usually applied to any NP-complete problem that is combinatorial in nature. In a broad sense it works by starting to construct a tree of all possible solution combinations and on each level pruning bushes that would lead to combinations violating constrains. Out of remaining branches on current level, the more promising ones are explored first. The best solutions is kept aside and is replaced as search continues with any better solutions. As search progresses, more and more sub-bushes can be pruned as compared the current best solution. Branch-and-bound method can terminate quite fast, depending on specific values, but it can run very long and have a huge portion of design space needlessly evaluated.

Unfortunately, branch-and-bound technique, at a cost of finding one of the truly optimal solutions, results in exponential (to the power of objective space) run-time even for one objective therefore making it inapplicable for optimization problems were the power of objective space is large.

Branch-and-Bound has been applied to Bi-objective Knapsack problem [58], High-level synthesis (as part of the developed system) [15].

Branch-and-bound can be combined with interactive/iterative methods and Linear Programming (discussed below) to solve multi-objective problems fairly efficiently and accurately, as for example in [38]

Linear Programming is a method that can be applied to optimize a linear objective function (with any number of variables) subject to a set of linear constraints (or original functions may be non-linear but must be transformed into linear relationships). This method can be specifically solved by Simplex algorithm for example, and it has been first proven that any LP problem can be solved in polynomial time by ellipsoid method [27]. In practice however, Simplex method is often better, and only for some classes of LP problems it can exhibit exponential running time, while ellipsoid method is often too costly due to high degree of polynomial bound. To solve a problem with multiple objectives, one of the "downgrading" techniques mentioned above is usually applied: such as weighted-sum, or goal programming or sometimes interactive methods. Otherwise, multi-objective problem can be solved by Linear Vector Optimization and application of Benson's algorithm [35]. It should however be noted, that this source does not provide analysis of scalability or computational complexity of solving a MOLP problem by linear vector optimization and the

original paper by Benson says that algorithm is practical for relatively small scale problems. A lot of real world optimization problems that are linear in nature and allow variables to take non-integer values as a solution. For more details “Linear Programming and extensions” by George B. Dantzig provides excellent coverage.

If all or some of the variables in an LP problem may take on only integer values (IP – Integer Programming or MIP – Mixed Integer Programming), then finding the optimal solution in practice often takes exponential time. A good example of MILP is Job Shop Problem, and that of ILP – a knapsack problem. Because in the case of integer restricted variables, the methods used for Linear Programming are not applicable, the problems classified as IN or MIP are often non-linear in nature. IP and MIP and their classical solutions method, that are often problem specific, are covered in [4]. Many world problems are linear in nature, but there are perhaps even more that are integer or mixed integer programming; it is no wonder that various heurists have proliferated at solving such problems efficiently, albeit sometimes approximately. The following sub-sections cover some of those heuristics.

II.3. Iterative Algorithm - Simulated Annealing

Simulated Annealing technique is one of the oldest ones proposed (and applied in practical setting), it dates back to the 1980s. It can be briefly summarized as a heuristic that starts off at a randomly generated solution (let’s call it a current solution), then generates a next random solution which is accepted as current solution if it is better than previous one (in terms of converging to an optimum or getting closer to constrained region). With a certain probability a new solution that is worse than previous one is accepted to avoid converging to a local optimum. This probability is usually set inversely proportional to time, i.e. solutions that are worse than current one are accepted more often in the beginning of simulation.

As the solution search is repeated at the beginning the distances between the old solutions and the new ones are larger; as time passes by the distance is becoming smaller. I.e. in the beginning heuristic attempts to explore solutions far and wide and as time goes by (in simulated annealing terms, “as temperature cools down”) the locale in which random solution being generated gets smaller.

The details of original algorithm for simulated annealing are available in [30].

Simulated annealing has been applied to TSP, SAT, Knapsack, graph partitioning, set cover, etc.

There have been few papers published on applying SA to multi-objective optimization, the most interesting is perhaps [52] where improved Multiobjective Simulated Annealing (MOSA) is proposed. A composite function is not used in place of multiple objective functions. Each objective gets its own cooling schedule which illuminates favoring of certain objectives as well as need to scale them (as in a case of composite function). Also, an archive is added to store non-dominated solutions found so far (originally, simulated annealing finds only one solution, which in a case of trade-off situation is often insufficient and multiple close-to Pareto optimal solutions need to be found).

The simulation is split in 3 parts: in the first, all temperatures are set arbitrary or infinitely large and for a specified number of iterations the objectives are "studied"; in the second stage the annealing proceeds as usual for a set number of iterations and the archive is filled with non-dominated solutions; and in the thirst stage, return-to-base is implemented, which restarts annealing with the most sparsely located solutions from the archive (as will be seen below, this idea is eventually used in all evolutionary algorithms to achieve good dispersion). Authors apply their algorithm to 2 and 3 objective problems (used in other literature on optimization), but acknowledge the lack of any inference about higher dimension problems as well as feature multiple references to other works and recommendations on the multitude of configuration values. The comparison to other methods is also lacking.

Benefits of Simulated Annealing are as follows:

1. SA can be applied to optimize continuous and descrete functions
2. Solutions found by this method do not need to be encoded in any particular form (as for example is the case with genetic algorithm).
3. SA shows good performance when a function/mapping has many local optima (i.e. does not tend to get stuck in a local optima)
4. In its original form SA is very simple to implement and requires very little storage memory (since only one last best solution and configuration probabilities only need to be stored)
5. This method is generic enough that with some modifications it can be applied to a variety of optimization problems
6. It has been around for a while and has been successfully applied to a variety of real world problems, such as place and route in circuit design, resource distribution

systems, etc. A recent book “Simulated Annealing, Theory with Applications” comprises a collection of articles showing simulated annealing being applied in different fields.

Drawbacks of Simulated Annealing are as follows:

1. Perhaps the main issue with using simulated annealing for obtaining an optimal solution in a self-reconfigurable system is that this heuristic simply does not guarantee finding an optimal solution or even a good solution. This is noted for example in [15]. Just as other evolutionary algorithms it requires specifications from end user/designer that are often induced by the nature of the problem at hand: temperature cooling schedule, stopping factor (for example, reaching certain temperature), the probability with which a solution that is actually worse than current one is accepted, etc. The more complicated versions of simulated annealing have even more parameters that need to be set prior to running. Some of these parameters can be estimated during a “trial” run, but there is no guarantee that heuristic will run successfully with this set of parameters and it also adds to the running time.
2. In addition, in its pure form simulated annealing is simply too computationally expensive (in term of cost function evaluations) since it does not store any of the previously considered solutions, nor does it make an attempt to generate a new solution in an “intelligent manner” by taking in account other factors: the new solution is generated randomly and is either accepted or not as a current one. Even with utilization of some sort of memory mechanism the number of cost function evaluations may still be prohibitively high for a system that needs to find solution automatically and quick.
3. At last, SA is not particularly suited for multi-objective optimization either. To paraphrase [5] that has surveyed quite a few recent papers that applied simulated annealing for multi-objective optimization: mostly, multiple objectives are combined into a composite function that is then optimized, and there are not much other ways of applying SA to problems with multiple objectives (see above for one example of such application).

Improvements:

Some of the drawbacks mentioned above have been addressed (if merging simulated annealing with another technique of its own right could be called addressing) as highlighted below:

- The issue of potentially generating and evaluating the same solution over and over again has been mediated by introduction of tabu list / memory functionality, that stores previously considered solutions. Size of the list and which solutions to store there (the latest, the worst, the best, etc) is arbitrary specified depending on the problem
- To make annealing less greedy, new solution is accepted as long as it is within certain threshold of the previous. Threshold varies based on the number of iterations.
- Running few annealing processes in parallel, starting at different points. [30]
- More specific techniques are covered in [46]

II.4. Evolutionary Algorithm - Genetic Algorithm

Genetic Algorithm is an optimization heuristic and proceeds as follows:

A random set of valid solutions is created (i.e. the combination that represents an implementable solution, which may or may not satisfy constraints at the point when solution is created. The specific handling of constraints differs across various versions of genetic algorithm). Often additional steps are taken to make initial set to consist of unique solutions only as well as to make the set to be a uniform sampling of points in the search space.

Then, the following steps are repeated until completion:

- Mutate/crossover the solutions in the current generation to create a new (offspring) generation.
- Check all the solutions' for fitness (i.e. how close does it get to fulfilling the objectives and assign a metric based on this (there are different ways of measuring fitness) and keep the best ones for the next generation.

- The process is repeated until stopping criteria is met and in the end a set of close to optimal solutions is obtained.

Genetic Algorithms have been applied for optimization to various problems such as High level synthesis (scheduling and allocation) of datapaths in VLSI circuits, as for example in [33], TSP, Knapsack problem [54].

Benefits of Genetic Algorithms:

1. GA can be used to solve constrained and unconstrained optimization problems, and constraints can be represented by a sophisticated multi-variable function
2. GA can and have been used to solve multi-objective problems with more than just two objectives
3. GA can be applied in combinatorial optimization (where the search space is represented by a finite set of points) and in optimization of mathematical functions (where solution space is represented by infinitely many points)
4. This heuristic is capable of optimizing a problem that has many local optima (multi-modal problem).
5. This method is very good for problems that require a sampling of near optimal solution points and not just one solution.
6. It has been already successfully applied in a variety of areas, including as mentioned above, high level synthesis, and is part of Matlab's Optimization toolbox where GA's parameters such as which cross-over operator to use or population size can be specified [42]

Drawbacks of Genetic algorithms:

A. *Encoding (format) issue*

A problem solution needs to be encoded in chromosome form, before genetic algorithm could be applied. This is not always a simple task.

B. *A number of parameters issue.*

A user of genetic algorithm has to specify (or rely on an arbitrary built-in value) quite a number of parameters that control the execution of this heuristic. Below are some of the parameters that may need user input:

1. The point when evolution process stops needs to be specified. Various suggestions have been made from having the designer to specify a constant number of iteration (for example [33]) to stopping when no improvement has been reached over last arbitrary set amount of algorithm iteration [20]. Obviously, in the latter case, that amount of iteration with no improvement being made has to be set. By no means are these the only works with such suggestions and there are more other ideas in between.
2. A metric needs to be set that controls comparison of solutions for similarity and therefore to allow the maintenance of diversity of solutions (since there is no merit in keeping very similar solutions, it almost like keeping duplicates). Some examples of such metrics are: Pareto dominance rank (non-dominated solutions are preferred); the crowding distance (as used in NSGA-II algorithm, where it is actually used it in conjunction with dominance rank); sharing parameter (used in original NSGA), etc. The metric has be decided upon at some point (i.e. which flavor of GA is used) and its value has to be set, albeit with some guidelines as for example given in [10] on p. 184. Not all genetic algorithms use notion of sharing parameter, authors in [33] use random shifting to maintain diversity in the offspring.
3. The initial population size is also something that just needs to be known or determined through trails. The number mentioned in the GA related references are between 100 and 1000. Ultimately, these numbers are problem specific and require careful input from the designer and potentially rerunning the simulation a few times before satisfactory results can be obtained.
4. Probabilities for cross-over and mutation operations need to be specified. These operations are used to combine the existing solutions together to create a next generation of solutions. Again, there have been some recommendations made, but either designer has to specify these or they have to be built-in. Moreover, these probabilities (the values) also change from one problem to another.
5. The particular implementation to choose for mutation and/or crossover should also be specified. This metric is also related with specifics of chromosome encoding. There has been a few ways suggested to carry out both operators and some shown to be superior for specific situations [15].

In a presence of changing environment it is possible that when a system needs to find optimal multi-objective designs in real time, *automatically*, that built-in values will

not guarantee the convergence to optimal Pareto front at all. Consider a situation when number of resources/stations that system consists of has suddenly grown a lot (and so has the number of possible solution combinations); in that case the number of generations/iterations used before may not be enough to find optimal solution and generation/iteration parameters need to be changed. This is ok in a non-real-time design environment - designer will tweak the numbers and rerun simulations. Obviously, this is not feasible when system must reconfigure by itself and/or within specific time.

This factor (user's involvement in algorithm configuration) implies that the user that applies the algorithm on the problem set has to have quite a good knowledge of the area to choose these values well. Poorly chosen values will lead either to solutions that are not close to optimal or to extensive computational time. This, consequently, prevents use of genetic algorithms in situations when no human input is available and system needs to reconfigure itself.

C. Genetic algorithms, due to their probabilistic nature (first generation of solutions is chosen at random, and choice of mutation/cross over is probabilistic), evaluate many more solutions (at the fitness stage of the algorithm) than needed. A cost function could be quite simple, such as when computing the total area of resources, but it could be computationally expensive when computing something like scheduling time (which requires running a scheduler algorithm).

   For example, as quoted in [10] the worst-case computational complexity of NSGA-II is O (iMN^2) where N is population size and M is number of objectives, and i is number of iterations. So if population is chosen to be 100, number of objectives is 3 and iterations is 1000, then there are $3*10^5$ evaluations. This is an upper bound and on average, the algorithm will converge faster, but this large upper bound exists nonetheless. So, in a situation when design needs to be recomputed in real time, the maximum possible time needs to be allocated for system's reconfiguration when using genetic algorithm.

   The issue of evaluating more solutions that necessary has been addressed in some way by proposing a mini version for the three of the main flavours of genetic algorithms: there exists a micro-GA algorithm, a micro-MOPSO and AMGA-2 [8] [56].

D. Considering the abovementioned issues genetic algorithms are not easy ones to implement and/or they cost additional computational resources. Take for example the case when uniqueness of each solution within one generation is maintained, this adds extra running time.

Note: SPEA and SPEA2 algorithms are not covered in presented thesis specifically, because as S.Tiwari et al. claim with reference to a number of other studies, that there is not much difference in performance between SPEA and NSGA [56]. The authors themselves [60] admit that on a 3 objective test problem SPEA completely fails to come close to Pareto-optimal front and SPEAII is outperformed by NSGAII.

Improvements:

First NSGA (Non-dominated Sorting Generic Algorithm) was introduced in by Deb and Srinivas in 1994 [11] and later improved by Deb et al. in 2002. The improvement was named NSGA-II [10]. Micro-GA algorithm with a very small population size was proposed by Coello Coello et. al in [8] in 2001. AMGA [57] and an improved AMGA2 [56] was suggested as an achieve based generic algorithm by Tiwari et al. in 2008 and 2011 respectively.

Below is a summary of some of the improvements that came from original GA and that were done specifically to NSGA-II. It should be noted, that NSGA-II, out of all spinoff of genetic idea, has enjoyed the most attention (it is about 3 times more cited than all other major modifications to genetic algorithm *combined*).

1. In order to reduce computational complexity at the non-dominated sorting stage the Arena principle is suggested [53]. The computational complexity (of entire such modified NSGA-II algorithm) is claimed by authors to be O(rMN) where r is the number of non-dominated individuals among the population N and M is the number of objectives. Based on authors' experiments (comparing run-times of NSGA-II and NSGA-II with Arena principle the improvement in run-time is between 10% and 20% better than original NSGA-II. Authors used same benchmark tests as were used to test original NSGA-II. So ultimately, the improvement is a scalar factor. Dong-Feng et al., propose yet another user-controlled number to specify the stopping point for non-dominated individual selection (i.e. Arena principle is applied to choose non-

dominated individual until their number reach some user specified level). Choice of this number, as admitted by the authors, is prone to the problems mentioned above [12].

2. Ou Yang et al., propose modification to NSGA-II's crowding distance calculation to include the dummy fitness (which is equal to the number of Pareto front that an individual belongs) + the distances of all the individuals that dominate this point [43]. The authors claim that no increase in computational complexity is required to achieve this because all dummy fitness values have been already computed anyway. The paper also proposes a user set value which specifies a threshold limit of crowding distance, so that if individuals who's crowding distance falls under this limit are removed from the current population. This is done to alleviateconcentration of individuals in particular areas of the front and maintain the diversity of dispersion in current generation.
3. Ferrandi et al., though improvement of NSGA-II was not the goal of their work, do mention few modifications [20]. One of them is to force max and min individuals (i.e. individuals which have one of the objectives minimized/maximized) into the original population N (which is randomly created). They also propose is somewhat improved notion of stopping the iteration of new generations: that is not by a fixed number (albeit modifiable for each new run of the algorithm), but when over last few iteration the set of best solutions has not been improved the population size is increased 2 fold, and if with the increased size no more best solutions are found, the optimization halts. How many of the last iterations are considered is specified by the designer.
4. Coello Coello et. al proposed a micro-genetic algorithm that is based on standard GA but it uses a very small (i.e. 5 initial solutions vs. some 100 for regular GA) population [8] and an idea of reinitializing new generation from the best solutions found so far that are specifically kept aside. The authors reported significantly faster performance on 5 different optimization problems mentioned in other literature and compared their results to the same problems solved using NSGA-II and PAES. However, all problems tested included 2 objectives only and authors readily acknowledge that more studies (of their approach) and tests need to be done on problems that involve more objectives. Judging by the later papers that quote Coello Coello et. al there has not been much further development of this idea as of now.
5. AMGA is largely based on NSGA-II, it improves the calculation of crowding distance to favour solutions that are as far as possible from all other neighboring solutions. It

also maintains only unique solutions at each generation and uses an archive of previous best solutions to create new generation. Authors run their algorithm on the de-facto standard set of benchmark problems and report better performance on 2 and 3 objective problems than that of NSGA-II. AMGA2 borrows from SPEAII and micro-GA in that population is very small and a large archive of best solutions is kept. Solutions from archive (selected both to be diverse and having large non-dominance rank) and from current population are used to produce offspring. Perhaps the most interesting factor of AMGA2 is its limit on user involvement: the only specifics needed from the user are number of function evaluations and desired number of end solutions. The rest of the values are computed by algorithm based on the number of objectives, constraints, etc.; the probability of mutation, for example, is not set beforehand by the user, it depends on dominance measure of parents and is thus ever-changing, and the parent population is set to 4 times the number of objective (though no particular derivation of this factor is offered). There are also further improvements in specific crossover and mutation operand selection. The authors report the complexity of AMGA2 to be $\Theta(T A^2 \log A / P)$ where A is the size of the archive, P is the number of parents used in each iteration for offspring creation, and T is the number of objective function evaluations, it is somewhat unclear from the rest of the paper how can T be inferred about. My understanding from the test section is that T is a parameter that specifies when the whole algorithm stops. In terms of benchmark tests, AMGA2 performed better or same on most of the other benchmarks covered in relevant literature, and fared worse on 2 of the tests. It should be noted, that as with other algorithms, number of objectives studied does not surpass three. So, as authors suggest, more testing is needed to show this algorithm's true superiority.

It is interesting to note, that genetic algorithms have been specifically applied to optimization of reconfigurable systems in [20], [33].

In conclusion: genetic algorithms are very powerful tools that can be applied to a variety of different problems and fields, they are capable of searching spaces that are continuous and discrete, can handle constraints, can find multiple close-to optimal solutions at a time and can be applied to problems with more than 2 objectives. However, it is fairly expensive to run and in the end, one is not guaranteed to find the optimal solution, it may require few reruns with different parameters. So, the issues mentioned above do not make

genetic algorithms as well suited for specific situation of a system where search space often changes and where very limited time and resources are available for finding new optimal solution.

II.5. Particle Swarm Optimization and Ant Colony Optimization

[19] has excellent coverage of particle swarm optimization methods and all of its history, improvements, convergence proofs, and comparisons to some of the other optimization methods. So, current subsection of this thesis draws most of the information from this book. The facts from other sources are quoted accordingly.

Particle swarm optimization (PSO) is a fairly new development. The first PSO was proposed in 1995. And MOPSO (Multi-objective PSO) was proposed in 2002 [7]. In simple terms swarm particle and ant algorithms mimic the behavior of social insect colonies and their collective approach to solving problems.

PSO algorithm is a stochastic evolutionary algorithm, that starts off with a population (a swarm) that is "flown" through the solution space as each particle tries to find the (one) optimal solution (or get close to Pareto-front in case of multi-objective optimization). The motion through search space is controlled by the notions of direction of the flight, particle speed and inertia (note that sometimes only one of these algorithm parameters is employed) [44]

The particles start off at random positions in the design space and with some original velocity vectors. The crucial difference is that unlike a classis GA algorithm which "remembers" or "knows about" at most two generations (the parent generation and the currently constructed child generation), the ants/particles have access to global memory that reflects past experiences and current state of the population. [7] The idea is that having "collective knowledge" at their disposal, all particles deviate to one optimal solution. As a very simplistic example, one may think of foraging ants that will eventually all follow one best path chosen by the colony via iterative search.

Constrains in PSO are usually handled in one of the ways commonly used in genetic algorithms, there are no PSO specific constraint handling methods.

Originally particle swarm algorithm was proposed for finding optimum of an unconstrained one objective problem. However, this is not of a great interest for this research, therefore this section concentrates on multi-objective particle swarm optimization. Of course, objective function could be aggregated in a static manner or dynamically (when weight associated with each function changes during optimization with some frequency). Another way is to keep a global best position and a particle's personal best position with respect to each objective. The personal best position could be selected randomly or based on average over all objectives. These two ways to handle multiple objectives, however, still result in one optimal solution. To generate multiple Pareto optimal solutions achieves can be used to keep track of non-dominated solutions found so far. At last, a separate swarm can be assigned to optimize for each objective. The same issues of maintaining diversity and choosing between two non-dominated solutions need to be addressed as in genetic algorithms.

Benefits of PSO:

1. It can be applied to constrained and unconstrained optimization, and constraints can be represented by a sophisticated multi-variable function
2. PSO can be applied to multi-objective and single objective optimization
3. PSO can be applied to combinatorial optimization, especially ant algorithms
4. This method can be used in a dynamic environment (i.e when location of optimum changes in time)
5. It is well suited for optimizing multimodal functions. Generally, unimodal functions are better (more accurately and effectively) optimized by classical methods

Ant Colony Optimization (ACO) algorithms:

Ant algorithms are used to solve combinatorial optimization problems defined over discrete search space [19] where problem can be represented by a graph and reformulated as a shortest path search, such as TSP, Job Shop Scheduling, etc. A node in a graph represents a possible component in a solution and links between nodes have cost(s). A colony of ants starts off at random points and explores the solution space in parallel. Each ant reinforces positive feedback for "good path" found so far, so that other ants are more likely to include this path as part of other solution search in the future. This social communication is done via pheromone laid on the trail. The trails that get visited more often get more pheromone laid on them and thus become more attractive choice for other ants.

When a problem has two objectives, ACO could be modified to have a separate colony to optimize each objective. Each colony does optimization sequentially at each iteration of the algorithm. So the solutions found by colony A for objective A, influence starting points for the next colony B optimizing objective B. The idea is to favour the subsequent solutions that are closer to the ones already found.

Just like other evolutionary algorithms ACO needs a few specs that are problem specific and are usually inputted by the user: the number of iterations of the algorithm (a level of acceptability of solution as well as stagnation could be used as stopping factors as well), a set of probabilities or heuristics that control ants in their choice between possible paths.

Benefit of Ant algorithms:

Optimizing graph searching when there is no way that nodes could be clustered or organized to conduct a more intelligent search.

Drawbacks of particle swarm/ant algorithms:

- As with genetic algorithms, particle swarm is not guaranteed to converge to the global optimum, only to the local one (as shown in chapter 14 of [19])
- Quite a few user defined parameters that are problem specific are necessary (or the algorithm computes those itself for better or for worse)
- Ant algorithms are limited to combinatorial optimization problems, such as routing, TSP, job-shop scheduling. [6]
- PSO/ACO can fall into local optimum (stagnation) unless special measures (which are problem/function specific) are taken to make sure that this does not happen
- Ant algorithms specifically due to their path exploring nature require significant memory to operate because each time an ant constructs a path between points a and b this path has to be kept in memory

The notable improvements of MOPSO

As there is a microGA there is also micro-MOPSO [22] that relies on two archives: one to store intermediate solutions and one to store final non-dominated solutions, it also uses a very small particle population (only 5 particles) to limit the number of times that objectives functions get evaluated. Authors report testing micro-MOPSO on about half of the

benchmark functions on which NSGA-II, SPEA and others were testes, and report a better (faster) convergence to Pareto front than NSGA-II.

II.6. Divide-and-Conquer: PICO

The PICO project has been reported on by Hewlett-Packard Laboratories in [25]. Unfortunately, there are insufficient amount of detail in this article (or from a few earlier publications on the topic) to infer proper judgments on the proposed ideas from *algorithmic* point of view and therefore to offer any kind of concrete critique of the method. Below is a brief summary of some of the key points.

- A complex system is built from simpler systems (subsystems) which are realized as Pareto-optimal designs. This reduces the design space to be explored on the level of the whole system. The whole system is decomposed automatically (i.e. it is not done by designer)
- It seems that authors propose a sort of semantic filter which they call 'validity filter' by indexing (sorting) different Pareto-optimal solutions of sub-systems by parameters (restrictions specified in a form of simple equality or inequality)
- Optimization is carried out by three different agents: constructor, evaluator and space walker, that communicate to each other. The Space Walker is responsible for searching the design space, constructor – for constructing specific solution points found by the walker and the evaluator computes the various cost functions/programs associated with the system and feeds this information back to the space walker. Authors acknowledge the unsolved problem of effectively decomposing evaluator, since when a system is broken down into subsystems that are optimized the cost functions and restrictions/parameters associated with sub-system are different than those associated with the whole system.
- The heuristics that spacewalker uses are described in more detail in [51], however, it does not seem to address more than 2 conflicting objectives (area vs. performance), it mostly exploits domain specific ideas to reorganize and lead the exploration and in its main idea the walker consecutively explores the neighboring solutions of the best Pareto-solutions found so far.

II.7. Other Methods

II.7.a Markov Decision Process (MDP)

In [3] the authors have proposed an interesting concept of solving a bi-objective (execution time vs. energy consumption) design exploration of multiprocessor platform, called *'mdp'*. The problem space is modeled as a graph, where nodes of the graph are "system states" (possible configurations of resources) and the graph is created by applying one of the defined set of "actions" to a random starting state that affect the values of objective functions. An action could be doubling number of processor, or a specific number of processors. "Dual" actions (an action that exactly undoes a previously executed action) are not allowed. The graph is being created and "walked" by adhering to the Markov Decision Process.

Authors make use of an idea that in a specific design problem transformation of one solution into another affects values of objective in a problem specific way, for which lower and upper bounds can be easily estimated without actual simulation. The algorithm works by simulating a random solution point, applying all the defined actions to it (first with same probabilities, but then with more accurate ones that are adjusted as algorithm progresses). New states (solutions) are thus created, which are then each progressively used to build new solutions mostly by statistical projection vs. actual simulation. When application of action to a given solution point results in a different solution who's objective values improvement are within specified threshold then the result of action is one solutions, otherwise the resulting (estimated) range in objective functions values is partitioned into multiple solutions. If then to two of these multiple solutions two (or more) different actions can be applied this is the only time when simulation will be performed. In the case that simulation should generate values that are outside of estimated lower and upper bounds, then the simulated value is cashed and the whole process is restarted. The mdp finishes when none of the applied actions give a change greater than the above mentioned threshold, when all possible actions have been applied or when maximum number of algorithm iterations has been reached. All the simulated solutions are added to a nondominated set, so in the end mdp produces a set of Pareto nondominated points.

At the very end the mdp process is restarted with the obtained optimal solution as the initial solution to verify that the process has not been trapped in a local optimum.

The numeric experimental results do look very promising, as authors report the same (or just marginally worse) accuracy and dispersion of solution points as other methods, but at a fraction of a running time: “The three marginally more accurate algorithms require two orders of magnitude more evaluations”. However, here are some objections that are not addressed:

1. Perhaps the main drawback is that down below the objectives are aggregated into a scalarizing function, not quite as simple as weighted function, but the objectives are combined into one function with a scalar parameter. The process of scalarization is to combine multiple objective functions into one aggregated function which is then optimized. The above mentioned parameter is varied over a set of possible values and with each variation the mdp process is rerun. Obviously an intimate knowledge of the domain is required to come up with the scalarazing function, and the way to vary the scalar (in the paper it has been determined experimentally). It may not be always possible to even come up with a scalarizing function at all. In a case of dynamic environment (i.e. when parameters of design space change) it might be necessary to look for a new scalar to achieve a proper search.

2. The stage of action definition is somewhat unclear in a sense that authors point that the number of possible transformations is the dominant factor in algorithm complexity, however, they do not make any recommendations beyond just defining the transformation as a rational addition to one of the parameters of the system.

3. There is a situation when more simulations would be performed that expected: when the estimated max-min interval for a particular transformation proves to be incorrect (i.e. the value of objective function returned by actually evaluating a given combination lies outside of the estimated interval). Authors do not say by how many more extra evaluations could the whole process grow in this case. The only piece of information given is that by increasing the interval by 10% of that of estimated values, their simulation values rarely ended up outside of estimated interval. It would

be good to know if there is a robust method of arriving at this value so that it could be applied to other problems.

4. Quite a few values need to be specified by user/designer: i) value that controls the size of the graph (it's actually a decision tree, because there are no cycles); ii) value that determines granularity of exploration (how small should an interval between max and min bounds ever get; iii) a value that controls how often is probability function is updated once simulation is carried out; iv) The number of actions that can be applied to a give state. These values are mostly determined through experiments for particular domain or by designer preference.

5. The mdp algorithm does not truly scale with larger design spaces. This objection is addressed by the authors by modifying the granularity parameter, and even so, since there are no specific guidelines for such modification, it is possible that a few test runs will be needed before achieving the right balance between quality of solutions obtained at lower granularity and speed of solution search.

II.7.b Epsilon-constraint Method

Epsilon-constraint method proposed in [34] is an improvement of an earlier version. As authors demonstrate an earlier version is capable of generating the whole Pareto-front but the complexity is exponential in the problem size n for some problems. Also, original method does not scale well beyond 2 objective case, in a sense that it only finds the Pareto-optimal solutions, which when projected onto a 2 objective plane (of the first two objectives considered) match 2-objective problem solutions. The proposed improved method is a posteriori general (non-problem specific) method for optimizing multi-objective problem for which a reliable single-objective optimization algorithm or a heuristic exists. The method can run with an arbitrary number of objectives and does not require any parameter specification beyond the desired number of objective solutions.

At each iteration of the algorithm a new Pareto-optimal solution is sought.

This methods complexity is O ($k^{m-1} \cdot T$) where k is the number of desired Pareto-optimal solutions to be found, m is the number of objectives, and T is the time it takes to

carry-out single objective optimization (assuming that the same optimizer is used for all objectives).

Authors point out that their method is well applicable in the cases when good methods/heuristics exist for single-objective optimization, but do not scale well to accommodate more objectives.

As is, the method is probably, due to its relying upon external single objective optimizers is not applicable to the situations when optimization happens in a time-constrained changing environment that may affect the parameters of the optimizer. It would also warrant a modification if the objectives at hand are expressed as constraints.

II.8. ACG

The method that represents the design space as an ACG (architecture configuration graph) has been originally proposed in [29] as a solution to a specific engineering problem of high-level synthesis. For the sake of naming brevity this method is referred to as ACG method from now on. The specific details and formulae of the method are presented in the section "III. Theory". Below is a short explanation of method's workings.

This is a priori method (it produces one close-to optimal solution at its completion) that optimizes one objective and satisfies constraints of other objectives. In the remaining parts the terms "optimal" and "close-to" optimal are used interchangeably when speaking about ACG method. It can be applied to combinatorial optimization problems whereby several (conflicting) constrains need to be satisfied simultaneously and one objective function is to be optimized (maximized/minimized).

The method works by first arranging the design space as if optimizing for each objective independently, so that objective values of all variants are in the mostly monotonically increasing/decreasing. Here, a variant is simply a combination of components that form a possible solution in the design space. The sorting is done by computing the amount of influence of each resource on the objective value and then arranging the tree of design space such that the more influential the resource is, the closer it is to the root of the tree. With this arrangement the distortion of monotonic change at the edges of the bushes is

kept to the minimum possible. Also, the left and right edge of a bush bound the values in between, it is guaranteed that the left-most leaf in a bush will have a lower/same value as the right most (assuming values increase from left to right). It is also guaranteed in this arrangement that the next bush to the right will have the minimum value same or larger as minimum value of the current bush. The reverse is true for the maximum leaf and the left bush. These facts allow effective binary search through the tree in order to prune out the portion of the tree where ALL variants are known not to satisfy the requirement.

The design space arrangement process is carried out for all objectives, thus creating "views" of the design space that are sorted for particular objective and therefore can be effectively searched via binary search that follows either minimum or maximum branch of the bush (depending on constraint limiting objective from above or from below). Once the search converges, the border solution that is the closest to constraint is found and everything after it can be pruned from further consideration. This border is found for all objectives functions that are constrained. After that the objective that must be optimized is considered and the best solution is chosen and checked to see if it complies with the borders established for other objectives. If it does not, the next best solution is tried. As soon as a solution is found that satisfies all borders the process finishes and this is a solution that is returned to the user (or used by an automatically configured system).

The most expensive step is usually the evaluation of a solution. So in terms of run-time complexity we have to be mostly concerned with the number of solutions (called variants) being evaluated. As specified in [29]this number for the arrangement and border finding stages is:

$$N_{total} = S * \left[(n+2) + \log_2 \prod_{i=1}^{n} m_p\,(i) \right]$$

Where *S* is the number of objectives with constrains, *n* is the number of items and $m_p\,(i)$ is the number of available versions/copies of i^{th} item (i = 1..n).

Unlike some of the generic methods discussed above, the problems to which ACG method could be applied must have the following properties:

- Combinatorial optimization problem (the solution space is discrete)
- There is one objective that is to be maximized/minimized
- It is possible to clearly identify the minimum and the maximum variants and to compute the minimum and maximum values for all of the objectives in a reasonable time.
- The problem can be modeled as a tree structure representing all possible combinations of components and the order in which components are combined (for design space analysis), is irrelevant. So, the multi-objective TSP cannot be optimized with this method. I.e. if there is a problem of finding a shortest path while satisfying constrains or optimizing other objectives (for example, minimize vehicle wear, while satisfying constraints on distance and fuel usage). The issue here will be that ACG is modeled by a graph, not a by a tree and it requires finding the minimum and the maximum for each objective, and in the case of TSP finding a minimum for distance objective IS solving the classic TSP problem which is NP-complete.
- Other objectives are represented in form of constraints (i.e. linear inequalities/equalities). It is possible to have all objectives expressed as constrains, in which case one of the objectives can be chosen automatically to be optimized.
- For each objective function it should be possible (before the commencement of ACG method) to sort the possible versions/permutations of one item/resource in the decreasing/increasing order with a good degree of accuracy. For example, a design includes an item/resource that comes in 5 different versions. It should be possible to sort these versions according to their influence on a given objective in a reasonable time (which is problem specific).

The ACG method can be applied to problems such as Job Shop Scheduling or in general, problems that are in a form of a constrained Knapsack problem.

Benefits of ACG method:

- The ACG method is specifically designed to handle discrete constrained combinatorial problems
- ACG method can be effectively used in a situation when constraints change during the operation of the system. The search for new border variants is going to be very quick, since the design space has been already arranged and some of the variants values

previously computed. Objectives can even change from being minimized to maximized or vice versa with new satisfactory solution being found at a cost of binary search (or faster, depending on the previously done search)

- It does not require any input from user that controls that algorithm execution: in terms of stopping factor, population size or desired precision metrics, etc. All it needs is the problem specification (just as any other method would). It may potentially use a user-specified factor controlling if and for how long would algorithm search for potential stray solution in the area of arranged design space known to contain a mixture of feasible and non-feasible solutions.
- It's computational complexity's main factor is O(lg N), where N is the power of design space.
- Although in its original proposition the method produces one solution in the end, it can be easily extended to produce more Pareto-optimal solutions.
- There is no such thing as unsupported solutions for this method, all Pareto-optimal solutions can be uncovered if need be at an expense of more objective function evaluations (but still much less than exhaustive search).

Drawbacks of ACG method:

- Cannot be applied to a multi-objective unconstrained optimization that results in a set of Pareto-optimal solutions.
- Cannot be used for non-discrete problems.
- Generally, the problem to be solved has to exhibit tree-hierarchy relationship between possible solutions and solutions can be generated by permutation of solution components. So this method is largely limited to combinatorial optimization problems only.

Somewhat similar idea has appeared in [32]. Kokjazadeh et al acknowledges the idea of sorting variants by value of one of the objective functions and exploring the area of higher interest (namely, where the value of the function changes more rapidly). However, the proposed method suffers from abovementioned user-involvement (a value needs to be specified that separates the area where the function grows fast enough and where it does not; the value again could be influenced by the problem itself). Also, the authors only propose

applying their methodology on a problem with 2 objectives. If there are more, then they (objectives) are combined if they're consistent. It is unclear if authors have a suggestion of what to do with 2+ objectives that cannot be combined. [32]

After being proposed in [29] the ACG method has been also explored in [49], [47], [48] and [59]. The next section looks at these propositions in detail.

Currently known improvements to the ACG method and discussion of reported results (on improvements)

Two improvements have been proposed so far. The main contribution of [49] and [47] is a notion of a priority factor which uses partial differentiation for resources arrangement. Also, [59] suggested a fuzzy search technique to improve the search stage.

Improvements to the sorting stage

Using the priority factor to compute the order of resource of each objective allows a speed-up of a small factor, namely $N \times S$ (N – number of types of resources and S is number of objectives). To recap, [29] proposes computation of the K-factor which is done by trying for each objective each given resources as the most influential one (on top of ACG tree). So, the expense of K factor computation grows linearly with the number of resource types (not even number of versions or copies of each particular resource), and this number if in fact is partially compensated if variant values are logged in the database soon as computed. This way some of the variants computed during arrangement stage, will be reused (by querying the database) during the search stage. Also, because K-factor relies on computation of actual objective it does not require an intimate knowledge of the objective function, as is needed for priority factor. It can be clearly seen in [49] that in order to compute the priority factor for execution time model the latency is omitted and the fact that minimum clock frequency causes largest change in cycle type as number of given resource is used to derive the final formula to compute the priority factor. So, in the case the objective evaluator details are not known it would be impossible to compute priority factor as it is impossible to differentiate an unknown function. Therefore, the original ACG method's K-value is more robust and preferable as it allows to ignore specific objective function details and thus add different types of objectives at stage of the system as long as evaluator for that function is available. It

may be possible to include priority factor computation as an option in a system for the partially differentiable functions, however, it is worth noting that computation of such simple function is very fast anyway, so even computing function values for n variants is going to be fast. Perhaps, this factor is what made the authors go back to the original concept of K-value in their most recent publication [50].

Also, when results in [49] are carefully considered, it is evident that there are some omissions/errors both in the presentation of results as well as interpretation of original work [29].

The numbers of evaluated variants reported by [49] for 2 edge detectors and 2 image combiners benchmarks that were tested in original proposition differ as well as the numbers corresponding to total number of possible architecture variants. Therefore, it must be assumed that [49] has implemented the original method somewhat differently when running their tests and/or used a different edge detector and image combiner functions. In the absence of any details of author's implementation of original algorithm as well as the exact specification of number of resources, number of variants of each resource everywhere but for Wave Digital Filter (table 8 in [49]) it is rather hard to accurately interpret the presented results.

In addition, there seems to be an omission in discussion of finding the final optimal variant, the formula presented on (p.51 in [49]) accounts of evaluations necessary to determine the border variants for constrained objectives. However, there is no guarantee that the found borders do not lie within each others' pruned regions thus necessitating further search and further evaluations.

Improvements to the search stage

The work presented in [59] attempts to improve the search stage of the ACG method by using maximum and minimum values for a given objective to derive projected values for the variants in between of maximum and minimum, as if the values grow in a strictly non-decreasing manner. All variants in the design space are assigned dummy membership values calculated using the maximum and minimum objective values. Then, the membership value is calculated for the objective constraint and a variant who's membership values is the closest is chosen as a starting point of evaluation. This variant is evaluated and then the search proceeds up or down the arranged design space tree using the previously found information (i.e. variants actual values). The search terminates when two neighboring variants (in terms of

their arrangement in this particular tree) are found one to be less then constraint and other to be larger. There are actually no steps in the algorithm's pseudo code taking care of situation when two neighboring variants are such that one is exactly equal and the other is greater/less. Assumingly, search terminates at this stage too. Authors report a significant improvement in the run-time, however ultimately, in the worst case scenario the improved method does not perform better than binary search (as can be seen in the Differential Equation Solver benchmark). In fact the search performs nearly the same amount of evaluations as binary search when the arranged tree has many local optima. In addition, for such trees, the algorithm does not even find the true border. The variant that it chooses does not split the tree definitively into two regions: one where all variants satisfy the constraint, and the other where most variants do no satisfy. In the case of non-monotonic change in values of arranged tree, there cannot be a variant that definitively splits the tree in two regions. There are two border variants: one that splits off the left side of the tree where variants all guaranteed to be less or equal to the constraint; second one splits off the right side of the tree where all variants are guaranteed to be more or equal to the constraint; and there is a grey area in between where values are mixed. (Assuming the arrangement of the tree was such that objective values increase from left to right)

You may refer to AppendixAppendix A for the detailed counter example demonstrating the fact that the true border is not being found as well as that the fuzzy search is just as slow as binary (in the worst case).

From authors work it is also unclear what the course of action is if Pareto-optimal set (as determined via intersection) is empty. Assumingly, the sets of acceptable variants need to be relaxed in the hopes of finding the variants that are not included into the set, but there is no specific ways to do it or computational complexity of this course of actions provided by the authors.

Upon consulting the work used by authors to draw on the concept of fuzzy logic (The Concept of a Linguistic Variable and its Application to Approximate Reasoning-I

L. A. ZADEH) it appear that proposed concept is much more closer to the simple interpolation search [31]. As Knuth states interpolation search is a successful technique for the beginning of the search in a large sorted data set; when search range has been reduced the binary search should follow.

To summarize, if it is known (or can be somehow predicted) that the objective values in the arranged design space are strictly monotonous or design space analysis is not to be carried out to reconfigure system on the fly, then the above suggested method may be used to offer a speed up; in the case that variants are perfectly ordered the average run-time complexity (for finding border variant for one constrained objective) would be lg (lg N), where N is the total number of variants in objective space [31]. However, because this method is in the worst case is at least as time consuming as binary search, in a reconfigurable system the worst case scenario time must be allocated anyway for the reconfiguration, and that is that of binary search.

II.9. Conclusion

Although a few of the methods above have been applied specifically to the task of architecture synthesis (PICO, MDP, genetic algorithms, simulated annealing) and few other are generic enough to be applied as well, in the situation when optimization seeks to satisfy certain constrains and not necessarily to produce true Pareto point, ACG algorithm is superior due to its fast run time and the fact that it does not require additional user-specified parameters to run beyond problem specification. ACG algorithm allows the design space exploration to be done automatically and in an environment with changing constrains/objectives that do not require human designer input for algorithm adjustments. This algorithm is also very generic in a sense that it does not need any experiments on particular problem or training and can be easily applied to other combinatorial optimization problems requiring constraint satisfaction, therefore permitting to build a generic optimization framework for solving various types of problems.

Although already very efficient, the ACG algorithm could be improved slightly for certain classes of inputs. Also, original publications do not have the detailed run-time complexity analysis. These gaps are filled with the work of the next chapter.

III. Theory

III.1. Definitions

Below are some method specific definitions that are used in talking about the method in the remaining sections of this thesis (some as defined in [29]).

Resource – a notion of an abstract component of a system. Resource could have multiple copies available to be included as part of the solution, it may also have an option to be not included at all. For example, both adders and an ALUs could be part of resource list. However, there may be satisfactory solutions that only use ALU or only use adders.

Architecture Variant – a composition of resources. A variant could be referred to by its number (counting from left and only in a context of specific order of resources) or by signature (as defined below).

Border variant – is such a variant that in a context of arranged design space it separates the space in two parts. One part contains consecutive variants such that all of them satisfy some constraint and the other part contains variants that do not satisfy this constraint plus possibly some variants that do.

Signature (of a variant) – is the order of resources within the variant (generally, the order would be dictated by an objective function), i.e.: A2, B4, C0, D5. Which means that this variant is represented by 2 version/copies of resource A, 4 version/copies of resource B, etc. The order of A,B, C, D does not matter in terms of uniquely specifying a particular variant, it only matters in a the context of sorted design space allowing comparison of variants.

Mask – an order of resources for some partial design space arrangement. A mask is needed to convert one order of resources into another order of resources (i.e. converting variants between objectives). Although, all variants can be uniquely assigned a number, the order of resources for each objective needs to be maintained in order to carry out binary search and semantic filtration (which is discussed later).

Constraint - Generally, in methods such as genetic algorithms a restriction is a separate function in terms of given parameters, whereas in the scope of ACG constraint is a cut-off value on the objective function, in a form of inequality:

$f(x) \leq k$, where k is some scalar and f(x) is an objective function, x is a variant. Alternatively, the constraint can be expressed as $f(x) \geq k$.

Local arrangement –the copies/version of one resource sorted in monotonic order.

Monotonic order – is such an order of values, that when going from left to right, any two consecutive values have difference of 0 or more.

Set X (entire design space) is said to be ordered with monotonic increase (with respect to objective function f(x)) if:

$\forall x (x \in X) f(x_{i+1}) - f(x_i) = \Delta \geq 0$

And vice versa, variants can be ordered for monotonic decrease:

$\forall x (x \in X) f(x_{i+1}) - f(x_i) = \Delta \leq 0$

Note: the tree diagrams in this chapter use a simple additive function as objective unless otherwise stated, to make diagram comprehension easier. Such objective function value is computed by adding the values of resources specified in variant signature.

III.2. Design Space Arrangement

The goal in design space arrangement is to arrange the branches of the tree representing the design space to achieve the most monotonous change in values of objective function from left to right. With such arrangement a binary search can be conducted to determine the portion of the design space that satisfies objective's constraint and once such area is determined, to allow any particular variant to be quickly checked for constraint satisfaction by simply doing logical filtration on that variant's signature.

Prior to engaging in design space arrangement the variants that use maximum and minimum number of resources are calculated and compared with corresponding constraints to verify that constraints are viable.

The notion of K-value (criterion) as proposed and proven in [28] allows for accurate way of sorting the resources in the most monotonic way without evaluation of every single variant in the objective space. K-values are computed for each resource and for each objective, after which the resources are arranged in a tree fashion so that the higher the K-value of resource is the closer to the root of the tree the resource is, resulting with resource with highest K-value (for a given objective) being at the top of the tree. This partial arrangement places the more influential resources on top thereby achieving the most

monotonous change (with least disturbances at the edges of bushes). For the sake of brevity and due to the fact that no changes are proposed to this portion of the algorithm, the description of criterion value calculation is omitted here and can be consulted in details in [29] and in [28]. Design space is arranged separately for each objective, in a sense resulting in multiple views of the same space from perspective of different objectives.

As mentioned in [29] after arrangement design space can be analyzed and potentially pruned, if after certain variant objective function reaches some saturation value and subsequent addition of resources offers no improvement. This pruning is done by taking the value of maximum variant (or minimum, depending on objective) and finding the saturation point (possibly within some user specified limit) in the same way as border variant is found as described in the next section.

III.3. Selection of Border Variants

For the discussion below, without loss of generality, suppose that objective is stated as values of "x or less are acceptable" and suppose further that the design space was conventionally sorted such that values increase from left to right when representing the design space tree visually and talking about this tree. (This conventional sorting is assumed everywhere in this thesis, unless otherwise stated).

III.3.a Original border selection process

[29] and [28] propose to conduct a search in a partially arranged design space in order to prune the region that contains unacceptable variants. The remaining portion of design space will contain the acceptable variants and, possibly, some unacceptable. The latter will happen when the objective function change is not monotonic around the bush edges. As can be seen in Figure 1. Partially Arranged Design Space Tree it is possible for design space to be partially arranged and, yet, the objective growth is not strictly monotonic around bush edges.

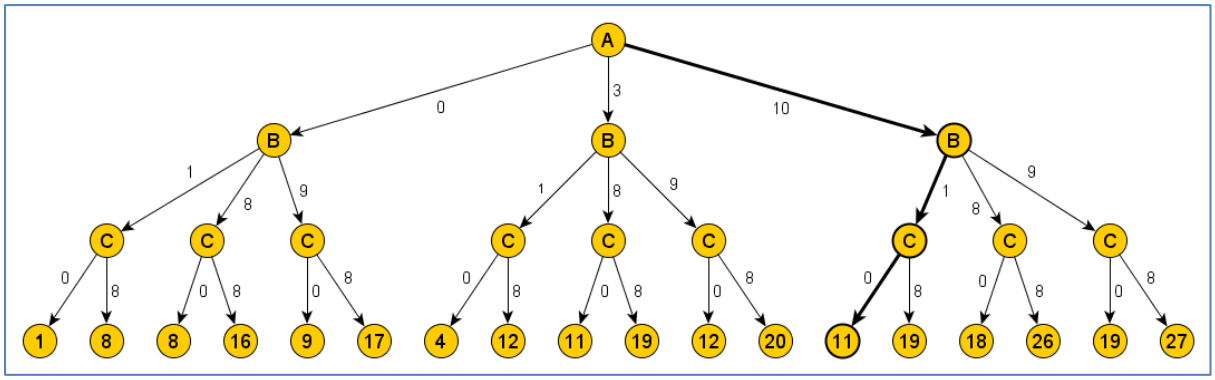


Figure 1. Partially Arranged Design Space Tree with highlighted border variant for ≤16 constraint. Everything to the right of A3B1C1 can be pruned.

The binary search is carried out by dividing each current level (for example, level of resource A) in half and then following minimum branch for the rest of the resources in case of top limiting constraint or the maximum branch in case of bottom limiting constraint. The variant is evaluated and depending on its value the search moves to the right or to the left. Each level is divided in half, until no more branches on current level are left. Division terminates when current level is the last one and division in last level has converged. Border variant shows that everything to the left (or to the right depending on constraint) can be removed from design tree, because all those variants violate the constraint.

This border selection procedure is repeated for all objectives with constraints, pruning unacceptable variants. At the final stage a variant that is optimum for one objective and satisfies constraints of all other needs to be selected. However, because the variants that remained after pruning may contain unacceptable ones, at this stage any variant even though lying within all areas limited by border variants, have to be evaluated to check if it does not happen to be the distorted variant. This process continues exhaustively trying variants from the to be optimized objective tree and evaluating all those that satisfy border variants, until a variant is found that not only lies within border variant limited area but indeed evaluates to acceptable values for all objectives.

Naturally then, when analyzing run-time complexity of this algorithm the question comes up: how long can this last stage continue in the worst case?

Example 1. Below is a simplified example showing that exhaustive search described above can be quite expensive in terms verification variant evaluations (which will have to be done for all constrained objectives).

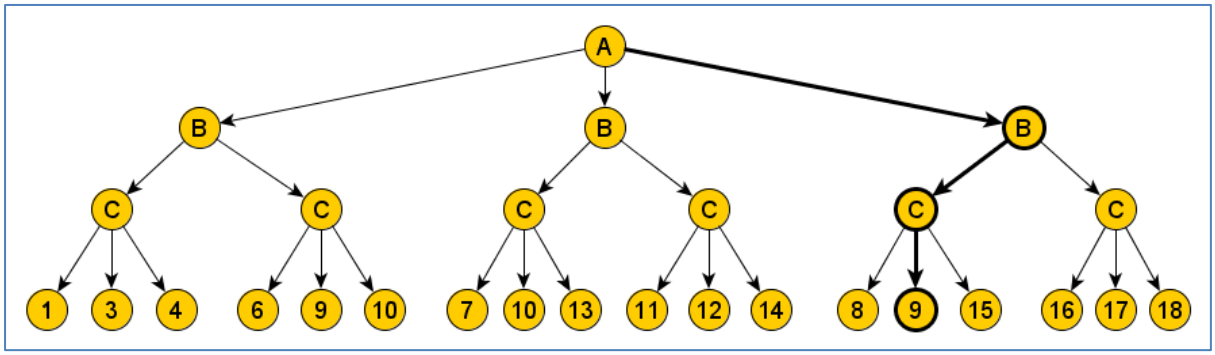


Figure 2. Partially arranged tree for objective X; border variant is A3B1C2 with constraint ≤9

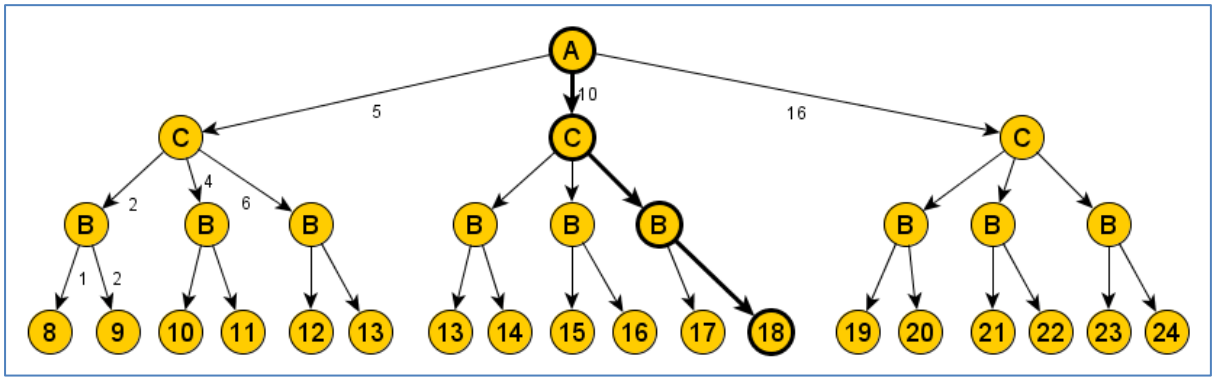


Figure 3. Partially Arranged Tree for objective Y; border variant is A2C3B2 with constraint ≤ 18. Everything to the right of A2C3B2 is pruned.

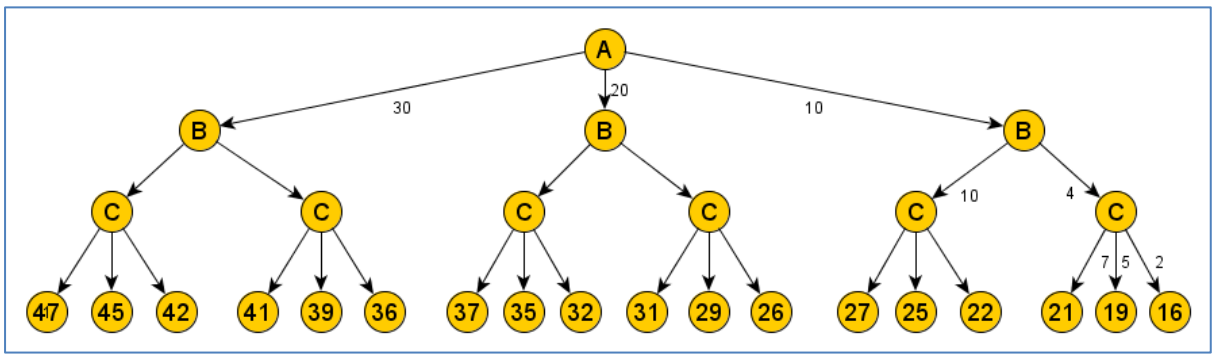


Figure 4. Partially arranged tree for objective Z, which is to be minimized

Objective X has specifically selected values, while objectives Y and Z are simple additive functions. Objective Z is to be minimized, notice that the arranged tree is the same for objective Z and X. So, the objective Z will be used to generate variants one after another to see if they fit into the other objective's borders. Variant A3B1C2 fits into border for objective X, but does not fit into border for objective Y neither does A3B1C1 because both belong to the A3 bush which is pruned entirely by constraint of objective Y.

So the variants A2B2C3, A2B2C2, A2B2C1, A2B1C3, A2B1C2, A2B1C1 will be tried exhaustively, until A2B1C1 which will finally evaluate to be within constraints for both objectives (X = 7, Y = 13, Z = 37). So, 5 extra variants were evaluated before a satisfactory one was found. Also, depending on the order in which objectives are considered, it could have happened that these 5 variants were evaluated for both objectives X and Y (if first Y was considered and then X). None of these variants would have been evaluated during the arrangement stage (only the variants A1B2C3, A3B1C3 and A3B2C1 would have been evaluated).

It is possible to have a partially arranged tree such that all variants but one (between border variant and the area where all variants are acceptable) are in violation of constraint. Here is an example:

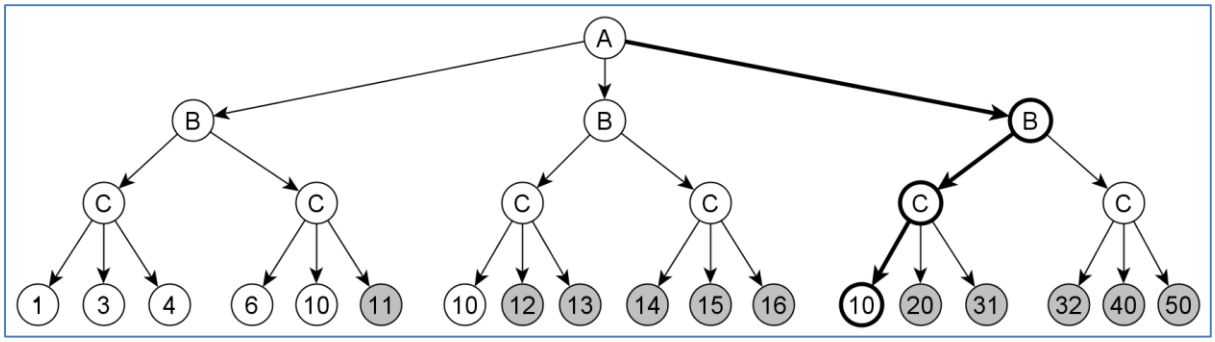


Figure 5. Only one acceptable variant between loose and tight border.

Obviously, there are many ways to grow this tree both in depth and in breadth thereby infinitely increasing the number of variants between the two with the value of 10 in above diagram. For example, the easiest way the tree can be grown by adding the number of versions that resource C has and increasing precision (the difference in objective value between adjacent variants), where *n* can be arbitrary large:

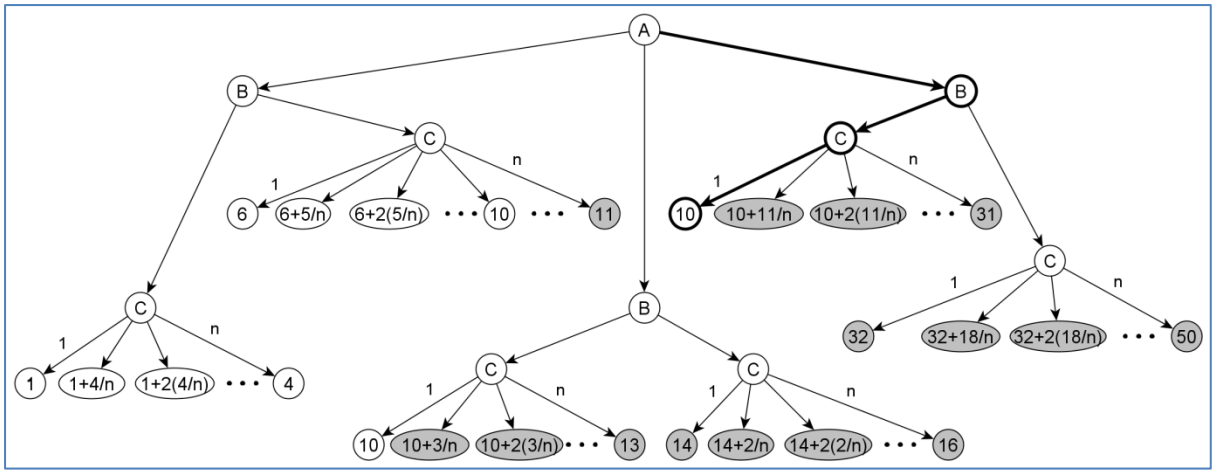


Figure 6. Growing the ACG tree, obviously, there may or not be a variant in A1B2 bush that is exactly 10, but if it would be it would be the last constraint passing variant in that bush.

Although, the above tree is artificially generated and is not a reflection of a real life situation it does convey the idea that ultimately, the number of variants that may need to be evaluated to find the final all constrains satisfying variant can be arbitrary large (at least within workable precision). This leads to:

III.3.b Problem statement

Potentially a lot of variants could be evaluated during the last stage of algorithm because finding the border variant prunes only the definitively unacceptable variants, while the remaining part of the tree may contain some unacceptable variants intermixed with acceptable and this number (of remaining unacceptable vairiants) is ultimately not bounded. It might be noted that in the previous example the variant A1B2C2 is a border below which variants are all guaranteed to satisfy the constraint. This variant would have been found if the constraint would have been set as ≥9. Clearly, in the worst case all variants before this one will be evaluated. Therefore is it necessary to determine the size of the area to be searched and to determine a more efficient means of finding the satisfactory border variant (i.e. to minimize the number of variants that are evaluated).

III.3.b Modified border variant selection procedure

It should be noted that the above described original border finding procedure is safe in a sense that none of the potentially viable variants are pruned from any of the objectives. Therefore, even if constraints are very tight, as long as there is at least one satisfactory variant it will be found.

Let's call the border variant as determined by original ACG method "loose border variant" and the border variant that leaves only the definitely acceptable variants as "tight border variant". Let's further refer to the area between tight and loose border variants as "grey area". With our conventionally partially arranged tree the variants to the left of tight border variants are all guaranteed to be less or equal to constraint and variants to the right of loose border variants are all guaranteed be greater than constraint. The only difference between finding the tight and loose border variant is the choice to follow maximum or minimum branch during binary search. Again, to reiterate, these two variants can only rise in a non-strictly monotonic design space arrangement. In the case when objective values increase strictly monotonically, the tight and loose border variants refer to the same variant that splits the design space into two areas (acceptable and not) and there is no grey area.

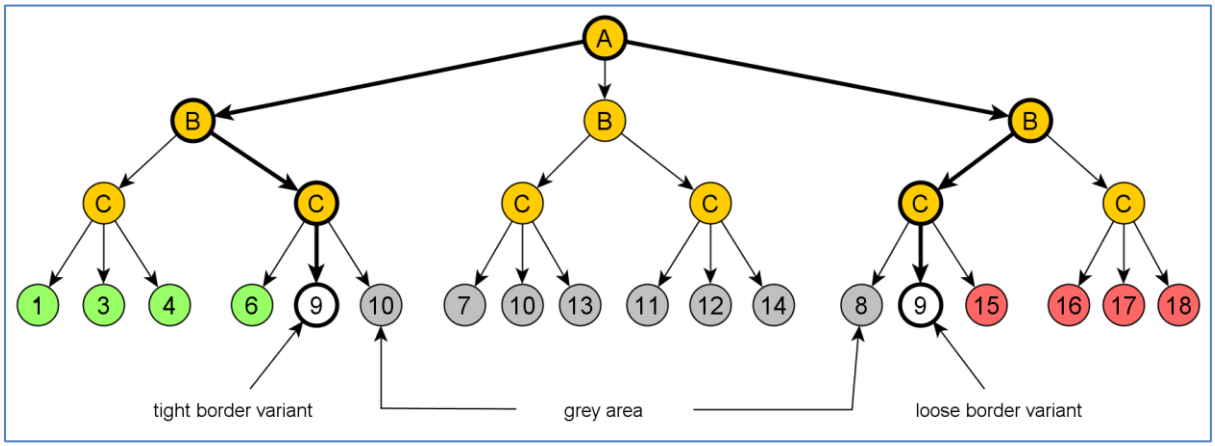


Figure 7. Tight and loose border variants and grey area

So, a modified search course is proposed. Instead of pruning design space using the loose border variant, the tight border variant could be found to determine set of acceptable variants for all objectives. This will cost as many variant evaluations as original procedure. After that searching for the optimal variant proceeds as before, except there are no verification evaluations needed, because if a variant fits (logically) within the area bounded by tight border variants of all constraints, the process can complete.

It may however happen that the areas bounded by tight border variants of different objectives, do not intersect at all. Then and only then the tree can be checked for existence of grey area, which can be further explored for missed out satisfactory variants. So the loose variant is not determined and grey area is not searched for unless needed.

III.3.d Searching grey area

There is no easy way (i.e. cheap) of checking if the grey area exists. Even though the indication of distortions is the break in monotonicity around the edges, there is no way of checking just one edge to confirm the existence of grey area. I.e. if the next bush on the top level has the minimum branch less than the constraint, then there is indeed grey area, but if it is not, there is no way to know if the grey area exists or not within the current bush. Refer to Appendix B for few sample trees with different possible distortions around the different edges.

There should be a specification in the original DSE problem of how close to true optimum should the finally chosen variant be (for the optimized objective). The general algorithm may be extended to switch into constraint loosening mode, if the areas of intersection between different objectives is smaller than certain threshold (specified by designer or end user).

1. For a given objective, use logic filtration as described in detail in Appendix C to verify for all constrained objectives, that the areas bounded by tight border variants intersect. This needs to be done only for objectives where one is constrained from below and other above. If there is at least one objective for which these areas do no intersect, then for the current objective we will be only considering grey area from now on (if it exists). This step does not involve evaluation of any variants and is therefore not expensive computationally. So, it is only for objectives for which definitively acceptable areas prove to be in contradiction with other constraints, that the grey areas need to be used.

2. So, assuming that for a given objective it was identified that if any acceptable variants exist they would lie in the grey area. As a sanity check, verify that the grey area for the current objective is at least partially within acceptable areas of other objectives, i.e. if all the variants to the right of tight border variants are not being rejected by another objective constraint. If so, then the current constrains are for sure unattainable. This

check can be performed with semantic (logic) filtration similar to that described Appendix C (i.e. see if loose border variants of two objectives define areas of corresponding ACG trees that have some overlap) and is therefore, computationally inexpensive.

3. Determine the loose border variant by conducting the binary search with reversed objective as described above except that the root level doesn't have to be divided in half, but we can start at the edge of the next bush. For this, identify the number of the top most branch of the border variant, say it's *i*. Use *i+1* to move to the next bush. Construct a variant by following the branch starting at *i+1* and all the minimum branches below. Everything to the right of this variant is bound to have the same or larger objective values. Evaluate this variant.

    a. If the value is larger or equal to constraint, then any of other acceptable values are within the same bush that contains tight border variant. So the binary search can proceed in this bush on the next level immediately.

    b. If the value is smaller than the constraint then the binary search continues dividing the top level to the right of the current variant.
    This small modification saves few evaluations in the case that the tight and loose border variants are within the same bush (on the top level).

4. Having determined the loose border variant we need to check if it is not the same as tight border variant. If it is then the last thing that can be done is the check for existence of grey area in the other objective that we are comparing with. If it does not have grey area either, then the whole process can stop as the constraints are clearly unattainable simultaneously.

5. In a case that grey area does exists, another sanity check should be done by performing semantic filtration against other objectives, to make sure that the grey area that is determined exactly now is at least partially within permitted areas for other constrains (before, in step 1 the entire area to the right of tight border variant was checked to verify partial overlap with other constrains).

6. Now the control can be passed back to final stage of algorithm that searches for optimal variant. It will proceed with exhaustively picking less and less optimal variants from the ACG tree partially arranged for the objective that is optimized.

   As each variant is picked, it can fall either within acceptable, not acceptable or grey area of the current objective's ACG tree. Here we are looking at the case when acceptable area is out of bounds, therefore, iff the variant falls exactly into grey area then we will continue looking at this variant, otherwise the next variant from the optimized objective tree needs to be picked.

   Next, the variant should be ran through semantic filtering for all other constrained objectives (besides the one currently looked at), and only if it passes all them then can the algorithm continue. Again, this operation is cheap as no variant evaluation is involved.

7. In order to search the grey area for variants that satisfy the constraint the following steps can be taken:
   a. Determine the lower border of the area to search. Take the tight border variant and move to the next smallest bush to the right and in that bush take the smallest branch. Because of the nature of binary search (as detailed above), it is known that the very next variant to the left is strictly larger than constraint (if it would not be, it would have been our border variant instead). And any and all branches to the right of the bottom most bush (containing the tight border variant) are larger than constraint (as the monotonic local arrangement is assumed). Therefore, the next variant that may (no necessarily will be but may) be less than constraint will be in the next bush to the right.

      Evaluate this lower variant and if it is more than constraint, then the entire bush containing this variant (counting from the root) can be pruned, and the bottom border can be moved to the minimum branch of the next bush (on the top level). Mind you this next variant is guaranteed to be less or equal to constraint, moreover, it would have been evaluated during loose border variant search. So, this next variant can be added to the list of acceptable variants within the grey area.

      If the lower variant evaluated to be less or equal to constraint, set is the lower border and add it to the list of acceptable variants.
   b. Determine the upper border of the area to search. With the similar logic as for lower border, the next variant that may be larger than constraint will be located at the

maximum branch of the next bush to the left. If this variant evaluates to be acceptable, add this variant to the list of acceptable variants.

8. Although the variants in the grey area may be intermixed (in terms of their objective values compared to constraint) they are however, still generally increasing from left to right. Is it therefore possible to employ binary search following the maximum or minimum branches of bushes. On average, it does not make a computation difference (i.e. number of function evaluations) to choose either maximum or minimum branch. However, from the point of running the rest of the algorithm, it would be more practical to follow the minimum branch (given our original constraint of $\leq x$), because these variants are, on average, more likely to be less than constraint.

   Take the search area determined above and divide it in half on the top level proceeding along the minimum branch for all other levels. Evaluate this variant.

   a. If the variant is larger than constraint, then everything to the right of it can be pruned on the current level.
   b. If the variant is less or equal to constraint, run it through semantic filtration for all other objectives. If it passes the filtration, evaluate this variant for optimized objective. If this value is "optimal" enough then the search can stop and this is the optimal variant that is returned as the final answer. Otherwise, continue.
   c. Save this variant and its value into the list of acceptable variants. (It is possible that original threshold of optimality needs to be relaxed and search rerun, this way the variants are not needlessly recomputed).
   d. Divide the top level to the right of current variant in half again and follow the minimum branch. Evaluate this variant.
   e. Repeat above two steps (a and b) until there is nothing left to divide on the current level.
   f. Move one level down and for each bush containing a variant from the list of acceptable variants, repeat the steps *d* and *e*.
   g. Repeat the whole cycle until the bottom level is reached.
   h. In the end the list of variants will contain all acceptable variants found above, and the additional ones can be inferred in between two variants that are exactly the same except on the lower level.

      It should be noted, that some of the variants being evaluated here would have been evaluated previously during the search for tight or loose border variants and thus

would have been saved from before (the ‘or’ is because which one would depend on the specific objective: is it less than or is it greater than, and therefore is the tight border variant on the left and loose on the right, or vice versa. To recap, the tight and loose border variants use different (max vs. min) branches in searching).

9. Each time a variant is evaluated it is tried (via semantic filtration) against other constrained objectives, before proceeding further.

It is impossible to say exactly how many values in grey area will be below the constraint and how many above. Without any assumptions about the nature of the objective function (except for the above stated assumption that with addition of resources the objective value does not decrease) it is possible to have all but one variants satisfying the constraint and vice versa. See Figure 5 for that example. That diagram is a representation of situation when all but one variants in grey area satisfy the constraint: if the original constraint was expressed as ≥10.

III.4. Semantic (Logical) filtration and Determination of the Optimal Variant

As described in [28], after the determination of tight border variants for all constrained objectives, the most optimal variant for the objective being optimized needs to be chosen such that it also satisfies all the constrains.

This stage begins with choosing the most optimal variant for objective being optimized and then checking if it fits into the acceptable areas of the designed space arranged and bordered for all other objectives. This is done without any evaluation by simply examining the variant signature against that of a border variant (described below).

If a variant does not pass a semantic check for one of the objectives, then the next variant is generated (in a scope of optimized objective) and semantic checks are repeated again for all other objectives. So the first variant that passes the semantic filters of all constrained objectives will halt the optimization process, and be returned as optimal solution. This variant would be evaluated for all objectives.

In a case that all objectives are described as constraints, it is also beneficial to have one objective to be prioritized for optimization and the same search process is done, with exception that if the border variant is reached for the prioritized objective, then the process

stops and constrains can be declared unattainable. In this case there may be potential and thus an option, to search the grey area (as described above).

When the grey area is searched, it should be done together with semantic filtration as mentioned above: soon as a feasible solution is found in a grey area it should be filtered, because it may pass all filters thus halting the need of searching the grey area further.

The semantic check is done as follows:

1. Take the optimal variant for the optimized objective, convert its signature into the proper order signature for another constrained objective O.
2. Start comparing the signature to the border going from one resource to the next in the order of this constrained objective O:
    a. If resource is strictly better (either smaller or greater, depending what objective is) than corresponding resource in border variant -> stop and pass
    b. If resource is strictly worse – fail
    c. If resource is the same, advance to the next resource.
3. Repeat for all constrained objectives.
4. It is possible to prioritize objectives to try first those that are very constrained, i.e. for which the area containing acceptable variants is small – this should cause filtration failures faster.

Actually, it is possible to potentially skip a few logical comparisons by starting not at the extreme optimum variant for optimized objective, but by looking at the border variants of other objectives and constructing a starting point that is more likely to pass. This is done as follows:

1. Start on the extreme optimal variant for optimized objective. For the sake of example, suppose the optimized objective is to be minimized (i.e. the optimum is on the left). The resource would be expressed as R1.1, R2.1,R3.1, …., Rn.1.
2. Look at all of the objectives that are constrained from below and look at the first resource (in the arranged order for that objective). Suppose that R3 is at the top of the design space tree arranged for a particular objective O and that the border variant is R3.X, RN.n, …., RM.m. So anything to the right of this variant is acceptable for objective O.

3. It is quite clear that checking variants R1.1, R2.1,R3.X-2, …., Rn.1 or .1, R2.1,R3.X-1, …., Rn.1 is pointless, because they will fail filtration at the R3. So when composing the variant in step 1 above replace corresponding resources with that at the top of the border variants for the objectives constrained from below. In this example, the initial point should be constructed as R1.1, R2.1,R3.X, …., Rn.1.
4. Further, having modified as per our example R3.1 to R3.X, we can also check if border variant allows R1.1 and if it does not (i.e. say border variant is R3.X, RN.n, …., R1.4, …, RM.m) then advance to R3.X+1, since the variant will keep failing at R1.

The above procedure can reduce the number of variants generated and then filtered by a lot, or by very little, depending on the resources' arrangement for the optimized objective and for the constrained objective currently looked at.

As with any algorithm, it is important to know the computational complexity of it both to be able to compare it to other algorithms and to accurately allocate time for algorithm completion. In the case of the on-the-run reconfigurable system, the tighter the run-time could be analyzed the more exactly the time for reconfiguration could be allocated. So, considering all of the proposed improvements the next subsection derives the formula for upper bound run-time complexity.

III.5. Computational Complexity Analysis

First of all, the run-time complexity is calculated in the number of variants being evaluated, because for many real-world objectives this is the most time consuming operation. If the specific configuration of resource is given, then to figure out for example, performance value, it is required to run a scheduler and a binder, which together assign operations to resources in appropriate to the task order. Although, of course, if all objective function are additive or similarly simple functions and design space is very large then the overhead of manipulating variants and searching for their previously calculated values will dominate over variant evaluation. Some analysis of memory requirement is presented in the next section.

Les us first define some variables:

S – number of objectives, one of which is being optimized, and the rest (s-1) have constraints to be satisfied.

n – the number of resources

m_i – number of versions/copies of resource i ($i \in 1 \ldots n$)

f_s – time it takes to evaluate a variant for objective s ($s \in S$)

The total number of variants can therefore be expressed:

$$A = \prod_{i=1}^{n} m_i$$

There are three consecutive steps as was discussed above:

a. design space arrangement,
b. border variant determination and
c. selection of the final variant that is optimal and satisfies constraints.

a. Computational complexity of design space arrangement

This stage has not been modified, so as [29] stated, this stage takes

$S * (n + 1)$ variant evaluations and allows to obtain partially arranged ACG trees for each objective

Or, in terms of specific times: $\sum_{s=1}^{S} fs * n$

b. Computational complexity of border variant

The worst case scenario needs to be analyzed for the modified border variant search and optimal variant selection procedure; all the steps that are related to border variant search are considered.

Finding a tight border variant requires computation of maximum and minimum variants followed by the binary search (out of which maximum would have been computed during arrangement stage):

$$O\ (A) = \ S * (n + 1) + S * (1 + \lceil \log_2 A \rceil \quad) + \cdots$$

It may be that all objectives have constrains, hence this operation would be carried out S times. (For now there is both n and A variables in the formula to make it shorter, in the end, the A will be replaced as per its definition).

In the worst case scenario, loose border variant will have to be found for all objectives as well (the max and min have already been computed):

$$O(A) = S * (n + 1) + 2S * (\lceil \log_2 A \rceil + 1) + \cdots$$

c. Finding the optimal variant

In the worst case scenario one objective is being optimized (which serves as a source of variants to try against other objectives) and all other objectives have the same arrangement and identical grey areas. The idea in this hypothetical situation is that each objective has grey area and for each objective there exists another objective such that permissible and pruned areas are exactly opposite (as shown in Figure 8). Of course, this does not represent any kind of situation that would arise in a real world, but because an upper bound needs to be found, the worst theoretical situation is considered.

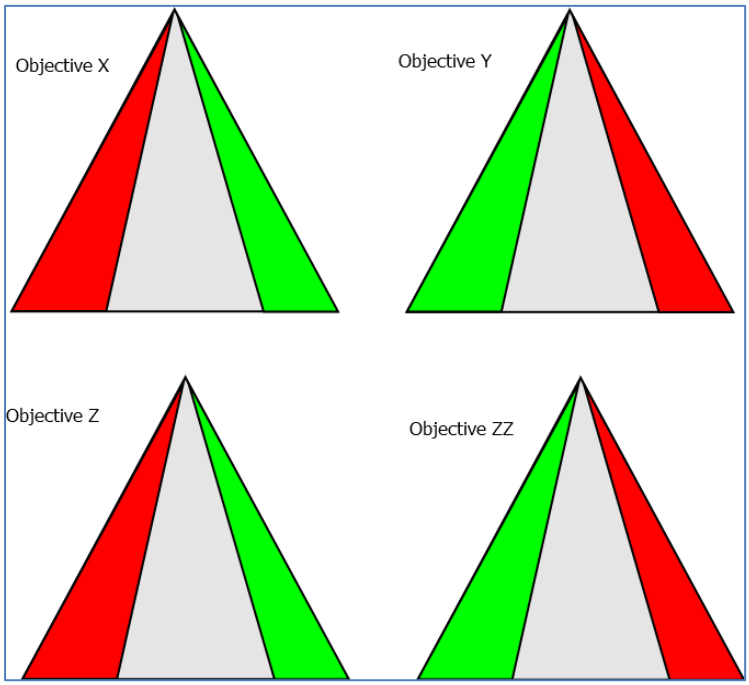


Figure 8. ACG trees with same arrangement and grey areas

So, in the worst case for all objectives the grey area is searched. Even though in the grey area the satisfactory and non-satisfactory variants are intermixed, they are still generally growing from left to right, therefore, by doing binary search on minimum or maximum branches (depending on the constraint being from top or bottom) it is possible to prune unacceptable part of the trees as the search goes on.

For example, in Figure 5 if the grey area is search, the first variant that would be considered and evaluated would be A2B2C1 with value of 13 which is greater than 10, and therefore everything to the right of and including A2B2C1 can be pruned as everything to the right is larger or equal to 13. Continuing dividing the grey area will lead to evaluation of A2B1C1 variant (which by the way would have been evaluated during search for loose border variant) and then to A2B1C2.

As was described in section "Selection of Border Variants" the idea is to repeatedly carry out search for loose border variant within the grey area and keep checking each and every one against every other objective. Since, as was proposed above, for all objectives search happens in grey area, which means that as we evaluate variants for the current objective we will evaluate them for each and every other objective (since if variant falls into grey area of another objective, it has to be evaluated to see if it satisfies constraint or not).

Clearly, in the most unfortunate case the binary search will prune only a small portion of a tree and each level from the top will be binary searched fully (as described in section III.3.d) so will be the next level and so on.

As was mentioned before, there is no easy formula of deducing the size of grey area given the number of resources and their version/number of copies, so if we will assume that grey area is the size of the entire ACG tree (which is for sure strictly more than it actually is) then it is definitive upper bound.

If there are n levels in the tree (by the number of resources) and on each level there are m_i copies/versions of nth resource (such that $= 1..n$) then the total number of evaluations required during this search is:

$$S \times (\lceil \log_2(m_n \times m_{n-1} \times m_{n-2} \times \ldots \times m_1) \rceil + \lceil \log_2(m_{n-1} \times m_{n-2} \times \ldots \times m_1) \rceil + \quad + \lceil \log_2(m_{n-2} \times \ldots \times m_1) \rceil + \ldots + \lceil \log_2 {m_1}^1 \rceil) =$$

$$S \times \sum_{i=1}^{n} \left\lceil \log_2 \prod_{j=1}^{i} m_j \right\rceil$$

Therefore, the complete formula is (if all S objectives have constraints):

$$O(n) = S \times (n+1) + 2S \times \left(\left\lceil \log_2 \prod_{i=1}^{n} m_i \right\rceil + 1 \right) + S \sum_{i=1}^{n} \left\lceil \log_2 \prod_{j=1}^{i} m_j \right\rceil$$

Or, if 1 objective is optimized and others have constraints to be satisfied:

$$O(n) = S \times (n+1) + 2(S-1) \times \left(\left\lceil \log_2 \prod_{i=1}^{n} m_i \right\rceil + 1 \right) + S \sum_{i=1}^{n} \left\lceil \log_2 \prod_{j=1}^{i} m_j \right\rceil$$

It should be noted that m_i should contains in itself the count of all possible resource versions\copies, so if a particular resource is available for example in at most quantity of 10 but may be excluded from the design at all, then, naturally, m for this resource should be 11, since all combinations need to be tried.

For example, suppose that there are 16 resources and each has 8 copies available, to be optimized for 4 objectives (one being optimized, the rest have constraints to be satisfied). Exhaustive search will make (in the absolute worst case when each variant is evaluated for each objective):

$$4 \times 8^{16} \approx 1.126 \times 10^{15}$$

evaluations whereas the above formula will result in:

$$4 \times 16 + 2 \times 3 \times (\lceil \log_2 8^{16} \rceil + 1) + 4 \times \sum_{i=1}^{16} \lceil \log_2 8^i \rceil = 132 + 294 + 1632 = 2058$$

evaluations in the worst case. In fact the actual number would be strictly less than this because, as was noted above, the grey area size was assumed to be the entire tree which it will always be strictly less than.

Therefore, the improvement is approximately 5.63×10^{11} times even in the worst case scenario when there are very few Pareto-optimal solutions in the given design space and they are located in the middle of rugged partially arranged ACG trees.

III.6. Analysis of memory requirements

Because architecture variant evaluations can be very expensive (in terms of time) whenever a variant is evaluated its value is stored. It is absolutely necessary to store a unique identifier with each variant value (i.e. a hash table with linked lists of conflicts would not work – ultimately, identifier is still needed because when a hash function sends to a particular

value how is it possible to know if this is this variant's value stored there, or some other one, who's hash value was the same?).

If a partially arranged ACG tree for some objective is chosen as a reference point and variants in this tree are numbered, then any variant can be referred to by this number provided the sorting mask is stored for each other objective. I.e. suppose the reference objective has the resources in order A, B, C, D….. (once partially arranged), but another objective has resources in the order of B, D, X, A …. So 10^{th} variant of the first objective is not the 10^{th} variant of the second. The mask can be stored in array of integers and the array needs to have n number of cells (n – number of resources). It will take few cycles to perform long division with remainders to obtain the actual variant signature from its number. Then the signature of reference objective is switched into signature of another objective, which then is converted to a number via multiplication. A signature will need an array of integers of size n.

Depending on size of design space it may or may not be possible to store variant identifies as its number in the tree. For example, in C language the largest integer is `long long int` with size of 64 bits [13], so that is enough to encode a variant number in the design space of say 20 resources that each have 16 versions ($16^{20} = 2^{80} < 64\ bit = 2^{64}$)

III.7. Conclusion

The current section fills in some gaps in description of the originally proposed method [29] and provides a specific course of action in search of the optimal variant in the partially arranged ACG trees, on an abstract enough level that the methodology can be applied to any problem satisfying specification, and not just architecture synthesis.

The specific situations explaining how the worst case scenario arises were discussed and detailed algorithms is presented for the course of action in the case that optimal variants from optimized objective fall into the area of the partially arranged ACG trees of other objectives that require verifying evaluation.

Considering these improvements it was possible to derive a more exact upper bound on the worst case scenario, where bound is expressed in terms number of objective function evaluations.

The next chapter describes details of implementation of all major proposed algorithms herein and discusses specific design choices (language, platform, data structures, etc) and their implication on performance and memory usage.

IV. Discussion of Implementation

The goal of the current work is to design a generic system that implements concepts and improvements discussed in previous chapter, such that the backbone functionality of objective space arrangement and searching is provided and the details of specific resource properties and objective functions are provided separately and independently.

Therefore, this chapter presents the details of implementation of such system, starting from language and platform choice to data structures and specific algorithmic decisions. The hope and goal of this implementation is to serve as a base for other research projects.

IV.1. Language and Platform Choice

Naturally, to accommodate a high level of abstraction and the notion of easy customization, a high-level language is the most suitable. The reason that the system is not implemented in a lower level, faster language such as C, is two-fold:

1. This is a framework designed for concept demonstration and testing purposes.
2. In a high level language is can be made fairly easy for other researchers/users to add their own modules to represent different sorts of resources and other objectives. With a lower level language in order to extend the system a much greater intimacy with implementation details would be needed, since Objective Oriented (OO) paradigm will not be there to provide an interface. The only way to avoid it would be to make would be extenders to communicate to the system via text files and plug-in executable. Representing complex structures via text file is syntactically cumbersome to say the least.

Java language was chosen for implementation due to the following factors:

- The OO paradigm allows to implement a easily extendable framework, so that custom objectives or resources could be easily added (as long as new classes representing a resource or objective implement a given interface, the existing code base would continue working).

- Java code is more readable compared to C/C++ (this is arguably a somewhat personal metric, however, Java, being a high level OO language is more descriptive, which makes it more easy and fast to read, and consequently easier to modify)
- The author has more recent experience using Java vs. C or C++
- Java is currently more widely used than C++ or C or at the very least used as much, therefore making the code base more easily usable by a larger number of other potential researchers (This is in itself a rather hard to measure metric, but there a few sources of statistical data pertaining to programming languages usage by different parameters, such as [42] and [45])
- [36] has analyzed the programming languages ratings [24] for their efficiency (how optimized their compilers are and how efficiently can code execute) vs. expressiveness drawing the conclusion that Java has been recently improved to match the performance of C++ and C quite closely and is thus sufficient for the sake of initial implementation as is the goal of the current work.
- At last, most of a MOP techniques discussed in the Chapter II have been implemented in Java. [14] is a good example, it was implemented for testing purposes along with the "classical" optimization problems, and has been quoted and compared in quite a few of the referenced papers. Although, not all previously discussed MOP techniques can be readily compared to ACG method, it is still better to illiminate any language advantage/disadvantage in any attempted comparisons.

For the similar reasons of availability and usability the system was implemented in the following environment: Windows 7, Java JDK 1.7, IntelliJ IDEA 11.

IV.2. Functional Specification

Inputs:

In order to function the system needs to following input: a list of resources (specifically, values for each of the resource corresponding to each of the objective) and objective evaluators that can compute objective value given a variant (a specific combination of resources). Finally, each objective needs to be marked with a constraint with possibly one objective being specified as minimized/maximized.

Main Components:

There are two types of resources: the one of which a number of identical copies is available and another type where a particular resource comes in different versions, or flavours (for example, a clock of different frequency, or different types of memory). Each resource at the very least needs to be specified by its type, the number of resource available (whether number of copies or versions) and the actual value/values. A resource could also be optional in the design of the system. (When a resource has versions, and each version comes in multiple copies then it has to be represented as multiple resources that come in copies). Resources could be specified by the user, or, more likely be provided as a pre-composed list. The actual input method is irrelevant for the current purpose so resources are created inside a test program.

A specific objective evaluator needs as input a variant and access to all resources in order to compute objective value. Whenever an objective is evaluated for a particular variant, this value needs to be stored, in the case it may be needed later (such as when constraint(s) change(s) and new variant needs to be found), as well as ability to do things like initial ACG tree pruning (i.e. pruning parts of tree that consists of variants nearly equal to maximum or a minimum). Therefore, a list or a database is needed to store variant values for each of the objective. All the database needs as an input is a variant and a list of values corresponding to all of the objectives. This list of course could be only partially filled if a variant has been computed for one objective, but not for the other(s).

The intent is to design an abstract extendable system that could potentially be broadened by others with custom objectives and resources by simply adding new custom classes that define a new resource or objective.

Output:

A variant (list of specific resource versions that satisfy all constraints while providing the closest to maximum or minimum for the optimized objective.

IV.3. Data Structures

The goal is to model a resource and an objective as an object to allow dealing with interfaces and obscuring specific implementation details (which could be very different between different objectives, for example) so that an abstract class can be provided for extension. The database is also modeled as an object primarily for simplicity and to allow creation of multiple databases, which can be useful at the testing stage.

There is no need to model a variant as an object because variant does not have any properties that can change (beyond its value for each objective) or be customized. In fact, it is sufficient to store either a number (in the particular tree order) as an integer (or long integer) or a signature of a variant as an array of integers.

Object representation is used only for resources and objectives, which are relatively few in numbers and "live" for the whole life-span of optimization process, whereas variants are represented with array of integers, because most of the variants are only looked at once. So the overhead associated with object creation and destruction (both in terms of processing time and memory) is kept to a minimum this way.

Resource

A notion of a resource is modeled via an abstract class of the same name that specifies a generic resource with resource name and values for each objective. An `Objective` class, which keeps a list (of partially arranged resources) is only ever concerned with `Resources` and not any more specific resource types that could are inheriting from `Resource`. `ResourceVersion` and `ResourceCopies` inherit from `Resource` and implement value storing and getting specific to when a resource comes in multiple identical copies and when resource comes in different versions. In the former case only the number of copies and one copy value is stored, where in the latter case, naturally, values of every version of the resource need to be stored. It is at this more detailed level that a function that can get resource value given a number and an objective is implemented, as it internal workings are different for different types of resources.

For example, a Clock class inherits from `Resource` and represents different clock types available for system: their specific power/unit area consumption.

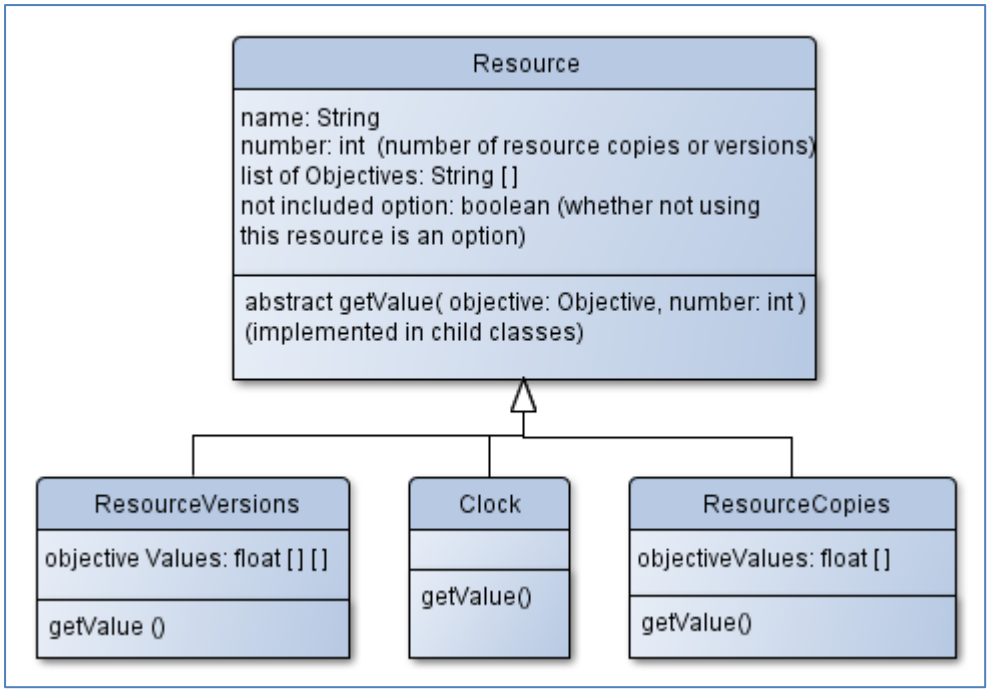


Figure 9. Resource inheritance UML diagram

Objective

The `Objective` class is where the main action of the system happens: it takes care of design space arrangement and searching. `Objective` is an abstract class that models AN objective. It leaves the details of evaluate method to be filled in by classes inheriting from it. What `Objective` does is it sorts the resources according to criterion value and finds the border variant, both of which are irrelevant to specific objective, i.e. the course of actions is the same whether the objective is cost in dollars, or performance.

Any of the inheriting classes implement the evaluate() method and add any other necessary details that are specific for a particular objective. As a very simple example: Area class does nothing except implementing the evaluate() method for computing area, but Power class also keeps track of which resource is the clock since its frequency value is necessary for computing the power objective. Power class can be supplied with a link to the actual Area objective class, and, if area is one of the objectives of the current system it can be used (both via values already computed, and by utilizing evaluate() method in Area class) to compute the power value.

The Cost objective class is a pure sample of what a cost objective function could be like: it implements discounts based on number of resources, a discount over certain total

value, adds taxes and shipping costs, for which it refers to the weight objective. The weight objective does not need to be modeled via a separate class, as it can be successfully modeled by creating another area objective where instead of areas the weights are added.

Objective class has many more methods to provide additional functionality primarily aimed to aid the analysis: such as writing a tree representing objective space to a file as a TGF graph [26] to allow examination of the tree by hand (or by multitude of third party software tools); counting evaluations, semantic filtration, traversing the objective space, etc.

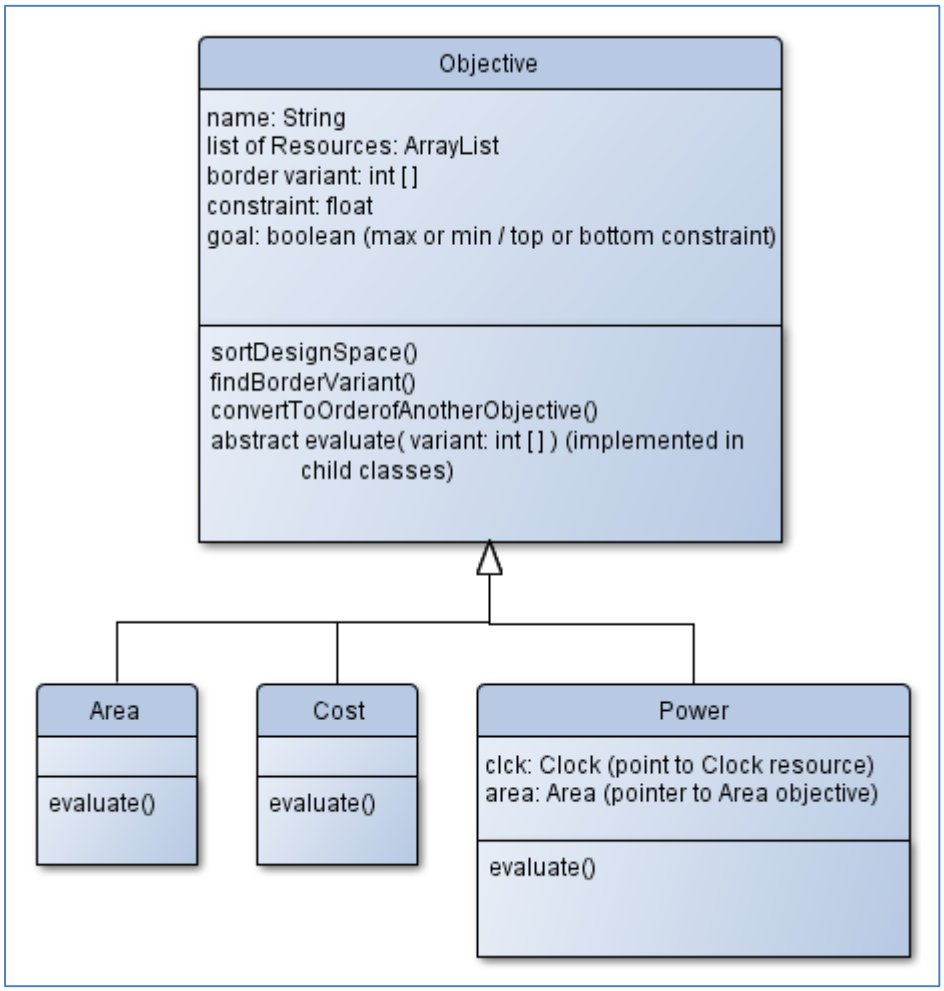


Figure 10. UML inheritance diagram of Objective class and its children

Variant

In the current implementation a variant is represented by an array of integers. It could be of course represented by a number (as counting from one of the sides of the tree). The

number is certainly taking less memory (although, if the system is potentially to optimize very large spaces it has to then work with long integer (which can store 2^{64} bits for a signed integer, thus giving an ability to represent a space with 2^{63} variants. With some manipulation, as Java does not allow unsigned variables, it is possible to claim all 64 bits for representing an unsigned variant number). A number would also make it easier to insert a variant into a database and maintain the order and it can much easier be hashed to illuminate search through database. However, when binary search of the tree happens or when semantic filtration is done it is still necessary to look at the exact signature of the variant, so conversion to and from variant number will be necessary. The amount of time the conversion takes is directly proportional to the number of resources and the number of versions of each resource as multiplication would need to be done to convert a variant signature into a number and a division with remainders would have to be done to convert a number into a signature.

Hashing will also result in quite some collisions which therefore would require a more complex structure (a linked list) than a simple table to keep the variants organized. The reason that either a very large and sparse list needs to be kept (minimizes collisions, but wastes space) or a list of smaller size (since it is known how many variants there can be checked in the ultimate worst case, as discussed above) but with more collisions that will happen because variants get picked from the whole space in a very uneven fashion (due to the nature of binary search) [21]. There might still be a way to take advantage of hashing but this would be another, separate area to explore.

Considering the above and the fact that the goal of this system is not to save every bit of memory possible, in the current implementation a variant is represented by an array of integers, where each integer *j* at position *i* indicates the number *j* of i^{th} resource (or j^{th} version of i^{th} resource) that comprise the given variant. The actual order of resources is dictated by specific objective and therefore, each objective stores with itself a mask that allows the order of resources of a given objective to be converted to the order of resources that is used in the database.

It is fairly simple to convert the database to store variant number instead of variant signature, and such change will only be localized in Repository class, since at any point in time only one variant is being considered and conversion to “signature view” is needed anyway (for semantic filtration, for when a maximum or minimum branch in tree is constructed, etc), there is absolutely no merit in converting methods in Objective class to deal with variants referred by number.

For example, if for the current objective the order of resources (after partial arrangement portion of the algorithm is carried out) is C, B, E, A, D then a variant [2,0,4,10,3] means C2,B0,E4,A10,D3 – following the convention used in previous sections.

Repository

Having the database of known variants represented by on an object allows for a few conveniences:

- easy passing of reference to the repository object to all objectives, so that each one of them would add variant values to the overall database as they get computed
- easier value fetching at the final stage of the algorithm
- all the internal implementation details are obscured from the rest of the system. The database can be kept as a list, as a hash table, as an array or connect to the external third-party database and such changes between implementation options would not affect the rest of the system.

It could be argued that per objective databases could be kept, however, there is much more value in a commonly shared database because some objectives (as the most obvious example area and power) can utilize other objectives' values during computation (i.e. when power is computed the computation is easier and faster, if an area is already known).

Mask

The order of resource which is used by variants stored in repository is be considered the master order and therefore all objectives keep a mask with them, that maps the order of resources in a given objective to the order used in the repository. A mask is just an array of integers with the same size as a variant (same number as there is the number of resources).

For example, suppose that repository orders resources in the same order that they were added (to the system): A, B, C, D, E. The example variant from the previous section (C2,B0,E4,A10,D3) would be converted to the order of repository via mask `[2,1,4,0,3]`, which is merely a map of locations. When it is applied it is interpreted as: take the value of first resource (C) and put it into position 2: `[ , ,2, , ]`; take value of second resource (B) and put it into position 1: `[ ,0,2, , ]` etc.

Resulting in masked variant `[10,0,2,3,4]`.

Naturally, while the design space of a particular objective is in the process of being arranged, the mask is being recomputed every time it is being used, because at this stage the order of resources is constantly changing. Once the space is arranged, a special Boolean flag is set indicating so and from thereafter the mask is being simply returned from the saved copy whenever requested.

IV.4. Some Important Algorithmic Details

Binary division:

When the binary search is carried out, the division is always rounded up. To avoid thus missing the branch of last division on any given level, the division actually starts at every level by setting two pointers at max and min branches and only quitting the division when they converge. Depending on result of comparison with a variant considered at any point of the division process either min or max (depending if constraint is from above or from below) is set to equal the branch of considered variant.

At any point in time only 2 variants are kept in memory: the last acceptable variant and the one that has been generated for the current check.

Finding border variant:

Before this costly stage commences, the maximum (or minimum as appropriate) value is checked against the constraint: in case the extreme variant value is beyond constraint – the process stops as constraint is unrealistic. The process also stops and sets the border equal to maximum (or minimum as appropriate) in the case that the maximum is less than constraint (or minimum is larger than constraint).

When searching for loose border variant, the goal (i.e. minimize/maximize) is reversed and the same procedure as for finding right border variant is repeated.

Maximum sizes:

Currently, repository is implemented via array of variants, thus limiting the number of variants that can be stored to maximum integer value (minus 2). The exact number would depend on Java implementation and operation system. It is however possible to change the

repository implementation to use a linked list; however then in addition to slight overhead due to using Objects for each list entry, the search will also have to be linear (and very often it will take the worst case – since each time a new variant is added, the whole list is checked to see if it was added before) while insertion will be of order 1.

The number of unique resources and the number of copies/versions of one resource is also limited by the maximum value of an integer, which is more than enough for even hypothetical optimization problem.

IV.5. Possible Constant factor improvements and corresponding costs

Below is a list of possible improvements that could provide a constant factor speed up at various stages of DSE. These are not implemented in the actual system exactly because they are providing constant factor speed-ups.

Memory: once the strict border variants has been determined and if relaxation of constraints is not0 an option, the variants that are thus definitively are unacceptable can be deleted. In the current implementation maintains a sorted list of all variants that have been computed so far. In the worst case all variants that are stored in the database will have to be put through semantic filtering for each objective, because for all, but on objective, the variants will not be in the sorted order. So, the list of computed variants will have to be walked in its entirety and each variant to be semantically filtered for each objective. Filtering can stop as soon as at least for one objective the variant is either definitely satisfying, or possibly satisfying.

Computation: if it is known that a particular objective function is of “additive” type, i.e. it behaves the same way as area function mentioned above, then the design space arrangement for that particular objective can be done by simply ordering the resources in the order of increase, without computing the actual K-values. It should however be noted that such functions are usually fast to compute anyway.

Computation and memory: If there is a resource that must be included in a project in certain quantity or configuration and there are no options of not including or using different

configuration, then such resource should not be included in the design space exploration stage. It should be appended to the variant for the purposes of computing the actual objective.

Computation and memory: as has been mentioned in [29], if it is known beforehand that two objectives are conflicting objectives, then they should be combined.

IV.6. Conclusion

In this chapter the specific details of implementation of the ACG algorithm were described, including the programming language and platform choices and the reasons for thereof; the break-down of the system into modules and the reasons of representing some entities via objects and some not. The main design decisions were presented and explained, as well as various potentially time/memory saving options were discussed along with alternative implementation paths.

It remains to test the working design space exploration system and access its ability to find an optimal variant compared to exhaustive search, as well as confirm the theoretical run-time complexity numbers. This testing and analysis stage of the present work comprises the next chapter.

V. Testing and Analysis

This chapter seeks to test the implementation of ideas proposed in Chapter III. Theoryand to confirm their viability and compare the ACG algorithm with modified search stage to the original one as well as to exhaustive search. Some additional statistics is collected along the way of testing (such as size of design space that satisfies constrains) that was not discussed in previous works related to ACG. This chapter seeks to present more information about the original ACG method as well as validate and access the proposed modifications.

V.1. Goals of Testing

The factors of interest in comparison are the number of functional evaluations, absolute running time and how close does the final answer get to the optimum determined through exhaustive search. These parameters are compared for three methods (exhaustive search, original ACG and modified ACG) in the view of overall size of design space and the size of the set of acceptable variants. All of the above is also contrasted as objective space tree grows "vertically" (i.e. number of resources increases) and horizontally (as number of versions of each resource increases. The main testing is done on partially monotonic design space, because if it known that all objectives (after design space arrangements) are strictly monotonic then it is naturally way better to use extrapolation based search as in [48] and no issues associated with searching grey area ever arise, because there is no grey area created by distortions.

V.2. Experiments Set-up

Objectives and Resources specification:

Four objectives are considered: cost, weight, area and weight, all of which are fairly simple additive functions to make exhaustive search run in a reasonable time. ResourceCopies objects are created for all resources except for Clock merely for simplicity (otherwise much more values would need to be specified by hand). The resource values for

each objective are created artificially but such that they are fairly close together thereby resulting in a not strictly monotonic partially arranged tree. During the preliminary testing (at the developing stage) it was observed that the closeness to the true optimum differs so each of the below control points of experiment is repeated for different constraints (from loose to tight).

When counting the number of variants evaluated it is the every time that an objective function is called that is considered to be an evaluation. (Retrieving previously stored value is not considered an evaluation).

Accuracy of solution for both the original and the proposed method is measured by setting the true optimum (value) found via exhaust search to be 100% and subsequently measuring the difference between true optimum objective value and those found by original method and the modified method respectively.

Exhaustive search:

This search finds the optimal variant by examining every single variant once and choosing the variant that offers the best value for the objective being optimized while satisfying all of the constraints. For 2 variants with equivalent values (of objective being optimized) the one with the overall best combination of other objective values is chosen (all other objectives are weighted equally and the differences of constrained objectives of both variants are compared). One pass is made through objective space and as a next variant gets generated, its value for the objective to be optimized is compared with the best variant saved so far. Naturally, not every variant is thus evaluated for every objective, because if a current variant does not offer improvement for optimized objective, the others are not considered, and, if it does, once an objective with constraint violation is found the further search ceases). So, the actual number of evaluations is between total number of variants and number of objectives multiplied by the total number of variants and depends on many factors: the tighter the constraints the less evaluations are done, the order in which resources are arranged and the order in which objectives are arranged.

Due to limitations on array size the results of evaluations are not stored in a repository for exhaustive search when the number of variants in objective space is larger than the maximum size of integer in Java ($\sim 2^{32}$). Because ACG method's implementation (both original and proposed modified) utilize a database to save evaluations and exhaustive search does not, it is not consistent to compare their running times. It is also beneficial to compare

how the three methods behave depending on how large is the portion of the design space that contains acceptable variants (i.e. those that satisfy the constraints), so when exhaust search runs, the count of all such variants is accumulated.

Resource combinations:

In real life there are often some resources that are included into design space as optional (either because they are such or it may not be known before space exploration if they are even necessary), so to take care of this fact ¼ of all resources in each test case are set up with an option that they can be omitted from design (so, the design space tree with 4 copies of each resource available, has a "zero" branch for some of the resources and therefore those resources have 5 branches, not 4).

One Clock resource is present (as it is used for power objective) and does not change from test case to test case.

Resources specification:

R1:	area - 10.0,	cost - 0.7,	weight - 0.05	
R2:	area - 30.0,	cost - 0.7,	weight - 0.02	
R3:	area - 100.0,	cost - 5.6,	weight - 1.25	
R4 (clock):	area - 2.0,	cost - 1.5,	weight – 1	frequency: 50
	area - 3.0,	cost - 1.2,	weight – 0.1	frequency: 100
	area - 9.0,	cost - 1.0,	weight – 0.15	frequency: 200
	area - 10.0,	cost – 0.7,	weight – 0.15	frequency: 400
R5:	area - 2.0,	cost - 0.1,	weight - 0.05	
R6:	area - 33.0,	cost - 2.45,	weight - 1.2	
R7:	area - 50.0,	cost - 0.56,	weight - 0.25	
R8:	area - 80.0,	cost - 6,	weight - 2.25	
R9:	area - 120.0,	cost - 1.9,	weight - 0.05	
R10:	area - 53.0,	cost - 4.7,	weight - 9.02	
R11:	area - 180.0,	cost - 15.6,	weight - 10.25	
R12:	area - 220.0,	cost - 3.6,	weight - 13.25	

Tests:

The specific values chosen for constraints are more or less random, except the fact that complete extremes were avoided: when constraints are very loose and thus almost entire space consists of acceptable variants, and the cases when there are very few acceptable variants (i.e. 40 valid variants in a design space of few thousand variants).

1st Set: exploring the effects of increase in number of resources. Three tests are done: 4 resources, 8 resources and 12 resources (4 copies each).
2nd Set: exploring the effects of increase in number of copies of each resource. The resource number is set to 4 and copies are varied as 4, 8,12.

Objective function specification:

To simply testing and because the actual objective function implementation does not affect the above mentioned measures to be collected, the following simple additive functions were implemented and added to the framework:

1. $Area\ (varaint: R_1.r_1, R_2.r_2, \dots, R_n.r_n) = \sum_{i=1}^{n} R_i.r_i.area;$
$where\ r_i\ is\ the\ number\ of\ copies/versions\ \ of\ i^{th}\ \ resource$

2. $Cost\ (varaint: R_1.r_1, R_2.r_2, \dots, R_n.r_n) = \begin{cases} (\sum_{i=1}^{n} R_i.r_i.cost\ ) \times 0.9,; & if\ r_i \geq 5 \\ (\sum_{i=1}^{n} R_i.r_i.cost) \times 0.95,; & if\ 5 \geq r_i > 2 \\ \sum_{i=1}^{n} R_i.r_i.cost; & if\ 2 \geq r_i \end{cases}$
$where\ r_i\ is\ the\ number\ of\ copies/versions\ \ of\ i^{th}\ \ resource$

3. $Weight\ (varaint: R_1.r_1, R_2.r_2, \dots, R_n.r_n) = \sum_{i=1}^{n} R_i.r_i.weight;$
$where\ r_i\ is\ the\ number\ of\ copies/versions\ \ of\ i^{th}\ \ resource$

4. $Power\ (varaint: R_1.r_1, R_2.r_2, \dots, R_n.r_n) = (\sum_{i=1}^{n} R_i.r_i.area) \times R_1.r_1.frequency;$
$where\ r_i\ is\ the\ number\ of copies/versions\ of\ i^{th}\ \ resource\ \ and\ R_1\ \ is\ \ the\ clock$

Test Cases specification:

Test Set 1					
4 resources 4 versions each					
Test number	Area	Cost	Weight	Power	Design Space Properties
1	≤ 200	≤ 12	≤ 3	maximize	Minimum variant: area: 132.0 cost: 8.3733 weight: 1.37 power: 6600.0 Maximum variant: area: 570.0 cost: 30.80945 weight: 5.43000 power: 228000.0
2	≤ 300	≤ 20	≥ 2.4	maximize	
3	≥ 400	≤ 24	≥ 2.4	minimize	
4	≥ 400	≥ 24	≥ 2.4	minimize	
5	≤ 400	≥ 11	minimize	≤ 150000	
8 resources 4 versions each					
Test	Area	Cost	Weight	Power	Design Space Properties
1	≤ 323	≥ 3	≤ 50	maximize	Minimum variant: area: 740.0 cost: 39.086135 weight: 36.42 power: 37000.0 Maximum variant: area: 3522.0 cost: 171.6773405 weight: 150.71
2	≤ 323	≥ 3	≤ 50	minimize	
3	≤ 500	minimize	≤ 15	≥ 200000	
4	≥ 500	minimize	≤ 15	≥ 200000	
5	≥ 500	minimize	≥ 15	≥ 200000	
12 resources 4 versions each					
Test	Area	Cost	Weight	Power	Design Space Properties
1	≤ 2500	≥ 60	≤ 120	minimize	Minimum variant: area: 740.0 cost: 39.086135 weight: 36.42 power: 37000.0 Maximum variant: area: 3522.0 cost: 171.6773405 weight: 150.71 power: 1408800.0
2	≤ 2500	≥ 60	≤ 120	maximize	
3	≤ 2500	≥ 60	≤ 100	maximize	
4	minimize	≥ 100	≤ 115	≥ 80000	
5	minimize	≥ 100	≤ 110	≥ 380000	

Table 1. Test Set 1 - Specification of test cases

Test Set 2.					
4 resources 4 versions each					
Test Number	Area	Cost	Weight	Power	Design Space Properties
1	≤ 200	≤ 12	≤ 3	maximize	Minimum variant: area: 132.0 cost: 8.3733 weight: 1.37 power: 6600.0 Maximum variant: area: 570.0 cost: 30.80945 weight: 5.43000 power: 228000.0
2	≤ 300	≤ 20	≥ 2.4	maximize	
3	≥ 400	≤ 24	≥ 2.4	minimize	
4	≥ 400	≥ 24	≥ 2.4	minimize	
5	≤ 400	≥ 11	minimize	≤ 150000	
4 resources 8 versions each					
Test	Area	Cost	Weight	Power	Design Space Properties
1	minimize	≥ 10	≤ 7	≥ 200000	Minimum variant: area: 131.0 cost: 10.4751 weight: 1.37 power: 6550.0 Maximum variant: area: 1130.0 cost: 57.46045 weight: 10.96 power: 621500.0
2	maximize	≥ 10	≤ 7	≥ 350000	
3	≤ 500	≥ 10	≤ 7	maximize	
4	≤ 850	minimize	≤ 7	≥ 350000	
5	≤ 900	maximize	≤ 8	≥ 300000	
4 resources 12 versions each					
Test	Area	Cost	Weight	Power	Design Space Properties
1	≤ 900	maximize	≤ 8	≥ 500000	Minimum variant: area: 131.0 cost: 10.4751 weight: 1.37 power: 6550.0 Maximum variant: area: 1699.6 cost: 85.7331 weight: 16.84 power: 1529640.0
2	≥ 580	maximize	≤ 8	≥ 572000	
3	minimize	≤ 50	≤ 11	≥ 572000	
4	≤ 700	≥ 26	≤ 4.6	maximize	
5	≤ 1100	≥ 25	≤ 5.5	minimize	

Table 2. Test Set 2 - Specification of test cases

For obvious reasons, the cases when all objectives are maximized/minimized or when no solution can be found by exhaust search are not presented in following results.

V.3. Results

The tests described in the previous section were carried out and the outlined measurements were taken.

Again, the number of evaluations is reflective of number of calls to AN objective function, because in all three methods remaining objectives evaluations are not carried through if the current one has violated constraint. Obviously, this number is dependent on the order in which objectives are checked for constrains.

# of resources	Test Number	# of acceptable variants	Total # of variants	# of evaluations in exhaust search	# of eval. made /w original method	Time (ms)	How close to true optimum is result from original method?	# of eval. made /w proposed method	Time (ms)	How close to true optimum is result from proposed method?
4 resources 4 versions each (1 resource is optional)										
4	1	40	1280	1,054	53	3	100.0%	54	3	100.0%
	2	2		815	54	3	100.0%	54	3	100.0%
	3	27		1,117	73	5	100.0%	65	5	100.0%
	4	95		1,185	55	3	100.0%	51	4	93.0%
	5	147		1,169	58	3	96.7%	55	3	95.5%
8 resources 4 versions each (2 resource is optional)										
8	1	161	4.096×10^5	307,530	372	35	99.7%	62	28	100.0%
	2	161		307,530	56	4	100.0%	59	5	100.0%
	3	6,720		226,043	121	5	96.8%	106	5	96.8%
	4	68,440		437,133	290	10	95.8%	105	20	54.2%
	5	8,555		354,401	120	54	84.7%	106	75	84.7%
12 resources 4 versions each (3 resource are optional)										
12	1	25,533,014	3.27×10^7	151,495,815	3,332	478	76%	122	620	100.0%
	2	25,533,014		151,527,513	138	161	100%	121	4190	100.0%
	3	19,265,838		151,503,729	26,122	76106	100%	121	9265	99.7%
	4	15,588,248		117,927,015	860,925	few hours	68%	148	8322	97.6%
	5	6,266,651		113,993,299	420,953	few hours	61%	149	3981	98.1%

Table 3. Test Set 1 - Varying the number of resources

# of resources	Test Number	# of acceptable variants	Total # of variants	# of evaluations in exhaust search	# of eval. made /w original method	Time (ms)	How close to true optimum is result from original method?	# of eval. made /w proposed method	Time (ms)	How close to true optimum is result from proposed method?
4 resources 4 versions each (1 resource is optional)										
4	1	40	1,280	1,054	53	3	100.0%	54	3	100.0%
	2	2		815	54	3	100.0%	54	3	100.0%
	3	27		1,117	73	5	100.0%	65	5	100.0%
	4	95		1,185	55	3	100.0%	51	4	93.0%
	5	147		1,169	58	3	96.7%	55	3	95.5%
4 resources 8 versions each (1 resource is optional)										
8	1	609	4608	19,334	823	37	100.00%	59	8	100.0%
	2	64		19,278	89	9	100.00%	60	11	100.0%
	3	1,497		16,931	52	3	100.00%	48	3	100.0%
	4	64		18,427	1,861	100	100.00%	73	31	87.7%
	5	267		19,405	394	23	98.26%	72	8	98.8%
4 resources 12 versions each (1 resource is optional)										
12	1	1,765	22464	74,861	1,100	101	86.65%	83	20	97.42%
	2	482		71,953	1,040	65	99.57%	82	37	94.92%
	3	1,196		81,422	6,944	590	100.00%	82	28	100.00%
	5	2,063		92,970	91	8	100.00%	68	5	100.00%

Table 4. Test Set 2 - Varying the number of copies of each resource

V.4. Analysis

It is quite clear from the results obtained that modified ACG method is faster and not only that, but it grows (with the number of variants or with the number of versions) in a more predictable fashion: very close to log of total number of variants, while the original performs significantly more evaluations as the number of resources increases as can be seen from the Table 3 and Table 4 and as the number of copies of resources increase. This is undoubtedly due to the nature of local exhaustive search that occurs at the final stage of the algorithm. The fact of potentially large number of evaluations performed by original ACG method that

happen when solving a large design space was discussed in Section III.3 and got confirmed on the scale of conducted test.

Because the original method makes a lot of evaluations (and consequently a lot of searching and inserting into the repository) when the design space is very large, the overhead from current implementation of repository becomes large as well. It was confirmed by separate test run that it is indeed the manipulation with the repository that is taking the time by running the same test with evaluation method re-computing the value instead of trying to look it up and inserting once computed. In the latter case the time is reasonable. This brings a conclusion that in real life the benefits of storing away computed values for future reference may be outweighed by the cost of manipulating database.

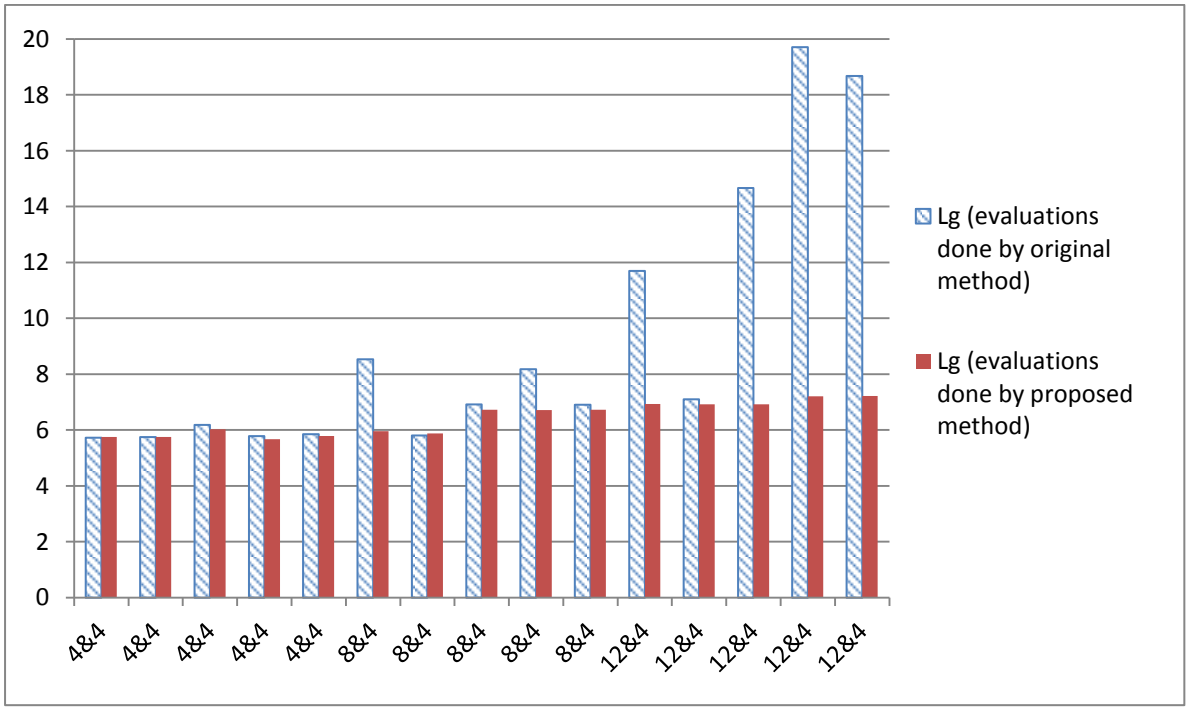


Table 5. Number of evaluations of original ACG and modified ACG as number of resources changes

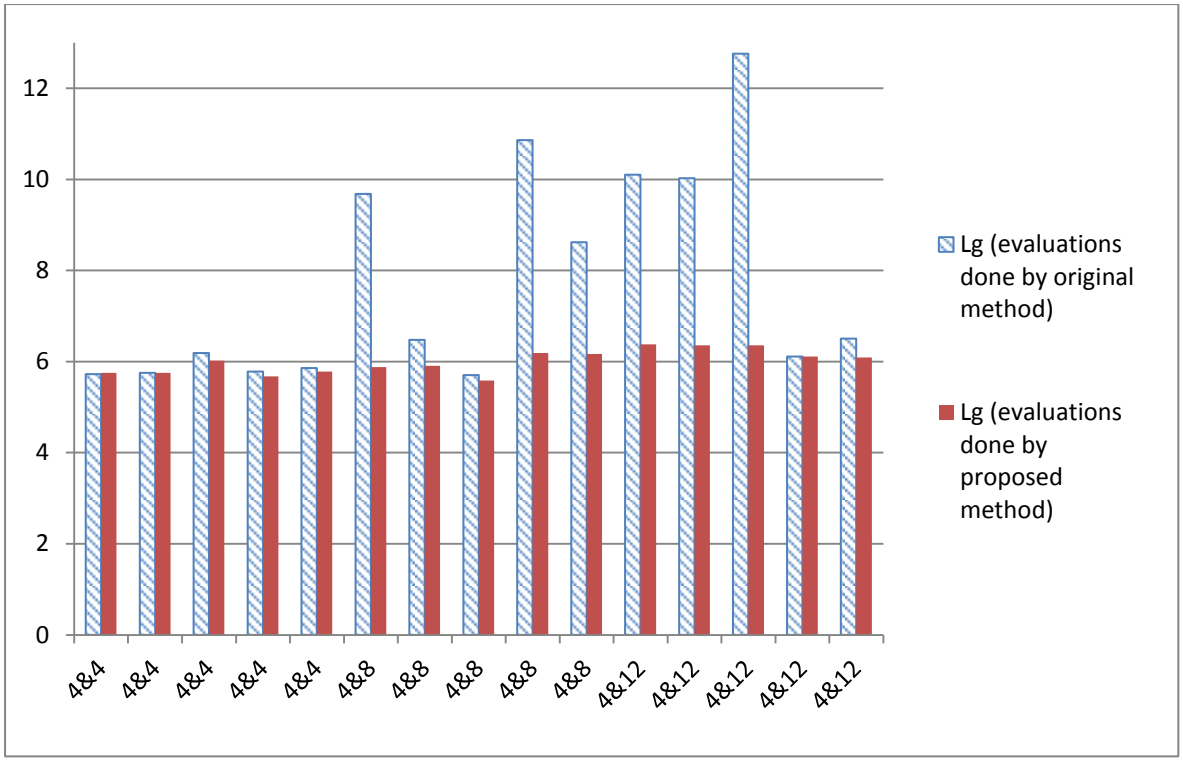


Table 6. Number of evaluations of original ACG and modified ACG as number of copies of resources changes

Accuracy of obtained solutions:

As can be seen from Table 7 and Table 8 both the original and proposed method can produce significant variation in how close is the obtained solution to the best solution. This is due to the fact that either one of the methods stops soon as a variant satisfying constraints has been found, and in a very non-monotonic space this could happen in a local maximum/minimum (opposite of objective goal). In fact, if for both test sets the average amount of variation (across all tests in a set) is pretty similar: for first test set it is 8.1% and 5.4% for original and proposed method respectively, and for second set it is 1% and 8% for original and proposed method respectively. The latter value is higher only because of one test case when the optimal value is extremely far away from the optimal value (although, all constrains are satisfied).

Therefore, at least based on the tests conducted, the proposed method does not offer any benefit of obtaining a better quality solution.

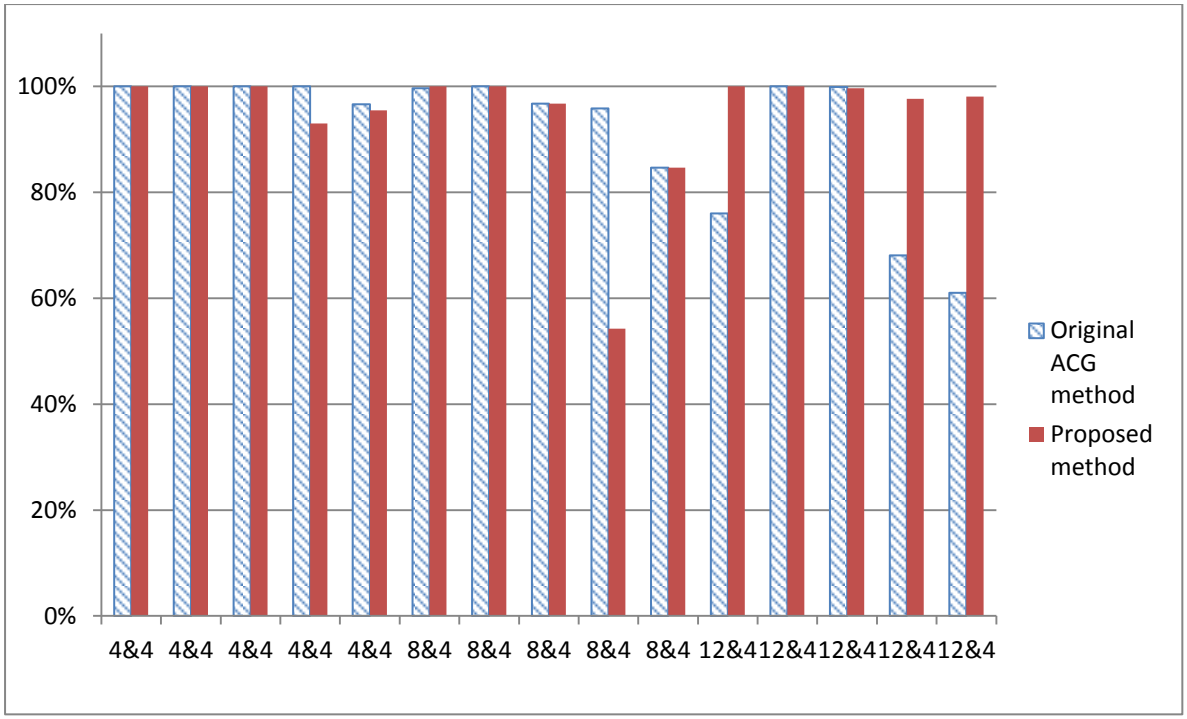


Table 7. Variation in accuracy of obtained results as number of resources grows

Table 8. Variation in accuracy of obtained results as number of copies of each resource grows

V.5. Conclusion

In this chapter the testing goals were motivated and described, as well as specific tests that were carried out. The description of the observations was done and conclusions from them drawn with the respect to the set-out testing goals, which were to compare the original ACG method and proposed modified method in terms of number of evaluations of objective functions, running time and how close does the obtained solution come to the true optimum obtained by exhaustive search. It was shown that proposed method is indeed faster (in terms of number of evaluations and run-time) but there was no evidence that it would allow to obtain more accurate solutions than original.

VI. Conclusions and Future Work

VI.1. Summary of Research

Current research work consisted of exploring the available multi-objective optimization techniques, their benefits and drawbacks, along with their area of application in a search of methodology that is simple and quick and as such can be applicable in design of systems where best solution selection is an on-going process. ACG methodology was selected as best suited for such needs. Itself and its improvements proposed by other researchers were studied, and upon this study it was found that original method has not had detailed run-time analysis carried out as well as that its final search stage was not clearly bounded. So, the current research work provides an algorithm to modify pruning stage and improve the last search stage of ACG (when the design space "view" for each objective has been sorted and areas of the design space tree that violate constrains are pruned).

Originally, the last stage carried out exhaustively and thus potentially results in a lot of needless objective function evaluations *when* this objective's design space has many local optima. The proposed modification to this stage consist of initially pruning larger areas of the design space trees down to leaving only areas that are guaranteed to satisfy a given objective thus illuminating the need for evaluations at the final stage. The pruning is rerun and relaxed only if searching with tight bounds proved fruitless. And then the grey area of the tree which contains both valid and invalid solutions is searched via binary search.

Given the more specific course of action it was possible to derive a formula for run-time complexity of the algorithm in terms of number of objective function evaluations.

The abstract framework was designed that implements the proposed algorithm. This framework is generic so that a wide variety of optimization problems (to which ACG is applicable) can be optimized with it. It allows potential users to add their own definitions of objective functions and resources.

During the experimental stage it was confirmed that proposed method is indeed faster. Since the original method was not analyzed in terms of how close does obtained solution get to the optimal that would be obtained through exhaust search. So an attempt was made to quantify the deviation from the optimal both for the original and for the proposed method.

During the experimental and results analysis stage further research areas were identified, which are outlined in detail in the next section.

VI. 2. Future Work

1. As was shown by the experimental results both the original method and the modified one can produce solutions, that though satisfy the constraints, have the optimized objective value being quite far from the optimal. In that light, it would be very beneficial to extend the method into finding a better variant by perhaps exploring the edges of the bushes around the variant to be returned as the answer. This should be especially beneficial for "wide" trees, i.e. when each resource has many variants.
2. Explore the possibility and conduct experiments of further speed up by varying the order of objectives in which they are checked during semantic filtration and searching the grey area. Perhaps by varying this order and checking first the most tightly constrained objectives the process could be made even more efficient, especially during the grey area search when more evaluations take place.
3. On a grand scale of things, the framework should be extended to have adaptation (ability to decide which constrains could be degraded. A weighted function or makers of "must not/must-have" parameters could be utilized, i.e. some constraints can be relaxed, and some cannot.
4. Go through specific optimization problems and access/modify the system to verify that currently proposed and implemented abstractions of resources and objectives could be extended to other types of objectives/resource concepts not considered in this research.
5. More testing of corner cases is needed (when there are only a few acceptable variants in objective space).
6. Test how much each of the proposed modification is contributing to run-time.
7. Experiments should be done to analyze memory usage (both for storage purposes and actual running of the program).
8. It became evident how crucial the efficient storage of computed values is. So the framework should be extended in two ways. First, there should be a decision mechanism together with plain specification by designer of when to store computed values. Depending on the size of the design space, the frequency and nature of potential system reconfiguration and time it takes to evaluation objectives, it may or not be worth it to store computed values. Obviously, if the optimization is a one-time endeavor, it is not

really beneficial, since overlap of computed variants across all objectives is quite small (there is no overlap for example between two objectives one of which is constrained from above the other which is constrained from below, because during border variant search different branches of the tree are followed, minimum and maximum respectively). For the situations when storage of variant values is needed, the different options of implementing the storage of computer values should be explored. A linked list could be a better option in some cases, since after certain number of accumulated computed values the overhead associated with keeping the database sorted is more than keeping a linked list unsorted and doing linear search.

Appendix

Appendix A. Rugged tree – fuzzy search does not find true border and takes as much as binary

Suppose that there are three resources A (3 versions), B (3 versions), C (2 versions) that are combined with the simple additive function. The weights of versions of each resource are indicated in the figure below as well as approximate membership values. The constraint is expressed as ≥19.

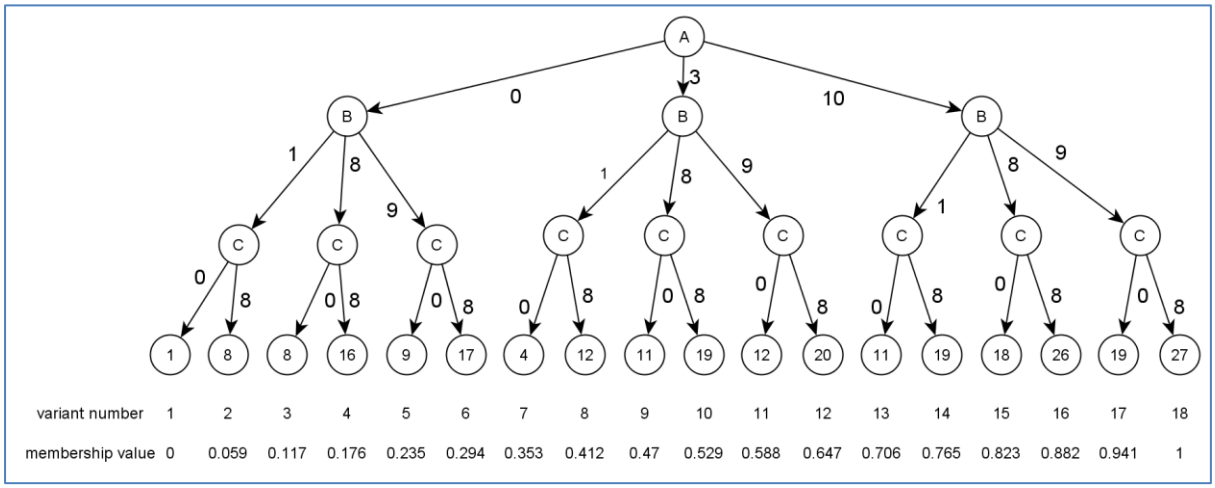


Figure 11. An ACG tree searched via fuzzy search

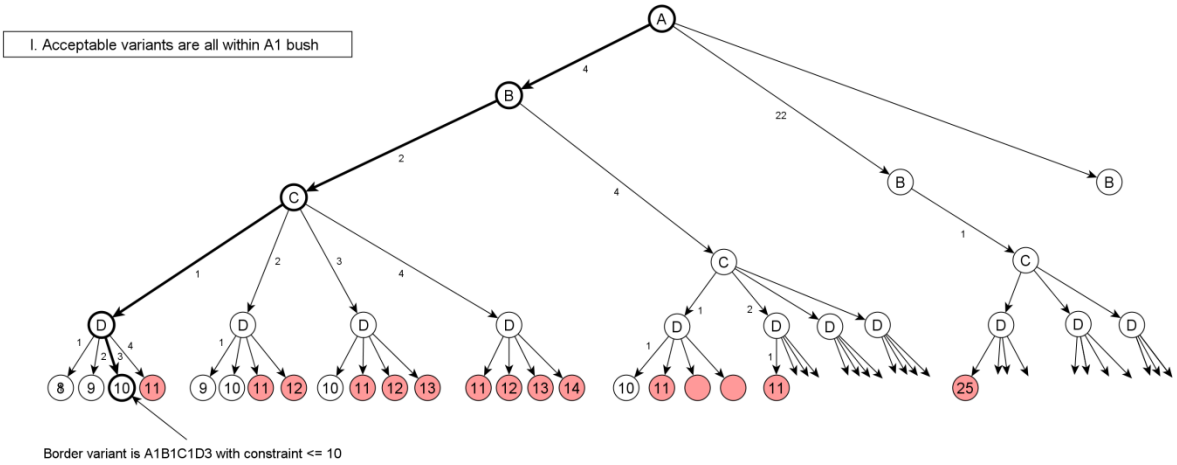
1. The initial membership value is $\frac{19-1}{27-1} \approx 0.692$
2. The closest is variant #13 (with approximate membership value 0.706). Evaluate #13: 11. 11<19 search up.
3. $\frac{1-0.706}{x-0.706} = \frac{27-11}{19-11-47}$; $x \approx 0.853$; the closest is variant #16 (0.882). Evaluate #16: 26. 26>19 search down
4. $\frac{0-0.882}{x-0.882} = \frac{1-26}{19-26}$; $x \approx 0.631$; the closest is variant #12 (0.647). Evaluate #12: 20. 20>19 search down
5. $\frac{0-0.631}{x-0.631} = \frac{1-20}{19-20}$; ≈ 0.613 ; the closest is variant #11 (0.588). Evaluate #11: 12. 12<19 search up

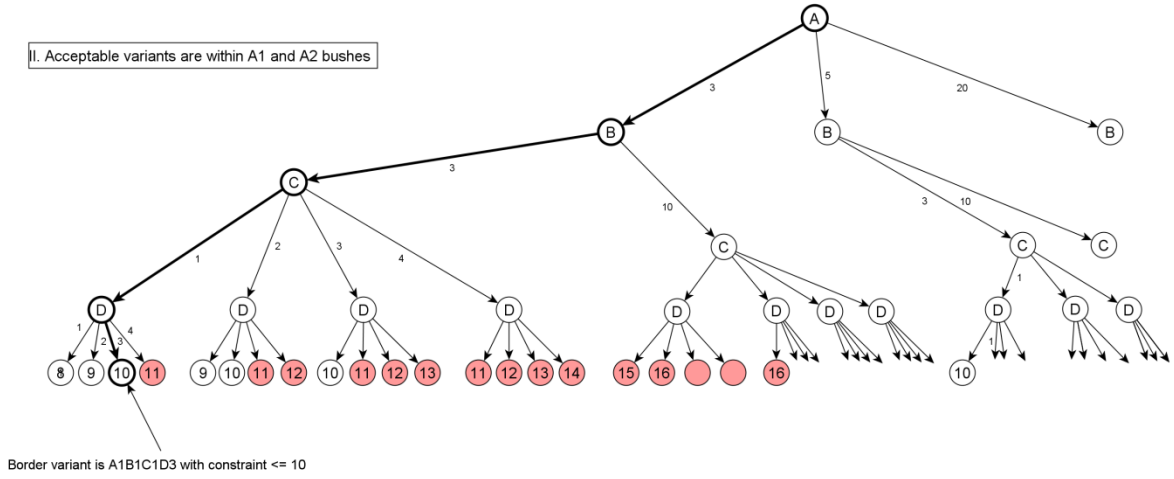
6. $\frac{1-0.588}{x-0.588} = \frac{27-12}{19-11-47}$; ≈ 0.780 ; the closest is variant #14 (0.765). Evaluate #14: 19. 19 = 19 Stop, the border variant has been found.

As can be seen above, it took 5 variant evaluations to find the "border". This is exactly the same number of variants that would be evaluated to search this tree via binary search ($\log_2 18 \approx 5$).

Notice also, that the border variant is actually #10: this variant below which there are no satisfactory variants. So this variant truly serves as border and prunes all unacceptable variants to the left of it. Whereas variant #14 prunes 2 valid variants to the left of it, and takes the same amount of evaluations to find, which, to reiterate, in the context of self-reconfigurable system will imply allocation of enough time for equivalent of binary search anyway.

Appendix B. Examples of trees to demonstrate that no specific edge check is telling enough about grey area.





Appendix C. Checking ACG trees overlap for different objectives

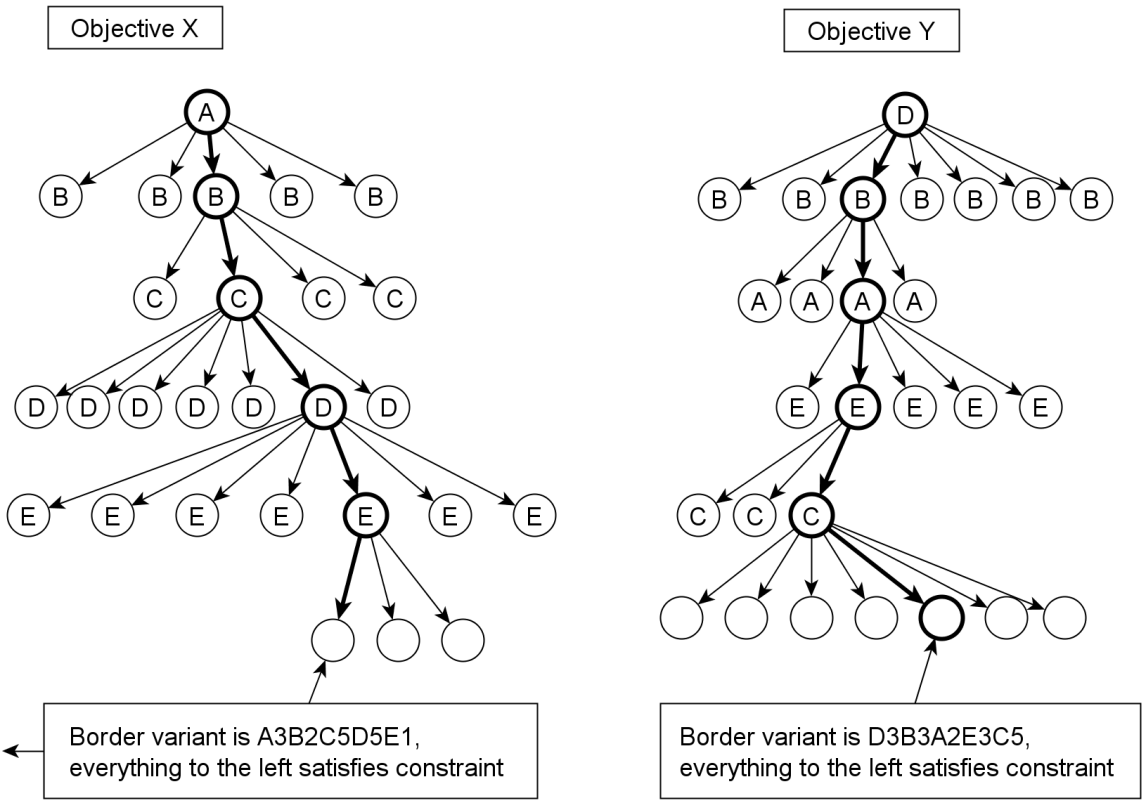
Given ACG trees have partially arranged and tight border variants found for each constraint, it is possible to check if the allowed areas of different objectives overlap or not. As long as there is partial intersection there is no need to look into grey area. So, a semantic filtration needs to be done. The algorithm proceeds as follows (assume as before left to right increasing objective function values as before):

1. If for two objectives both constraints are expressed as ≤ (or ≥) then there is for sure an intersection. DONE.

2. If objectives are expressed one as ≤ and the other as ≥, then there may or may not be intersection of the areas defined by corresponding border variants. Let's name one objective X and the other Y and, without loss of generality, assume that objective X has constraint ≤x and objective Y - ≥ y.

3. Set pointer *n* to the top resource of arranged ACG tree of objective X and pointer *m* to top resource of ACG tree for objective Y.
4. If m and n denote different resources, consider objective X (it does not matter which one is considered first).
5. If there is at least one full branch (on the current level) that is included in permissible area, then DONE. This is checked by subtracting 1 from the number of version/copies of resource m specified by the border variant. If result is not 0, then clearly, the sub-tree than stems from m.1 resource includes below all possible permutations of remaining resources, some of which will be allowed by the other border variant.

   In the following example, all variants starting at the A1 and A2 include all possible permutations of all other resources, therefore A is not the most influential resource for objective Y, there are plenty of combinations in these two bushes (starting at A1 and A2 for objective X) that include branches with D4, D5, etc.



6. If there is only one branch included in permissible area for objective X then repeat steps 4-5 for objective Y i.e. check if objective Y has at least one full branch on the current level, if so then DONE. If not, continue.

   If algorithm makes it to this point, this means that on current level both trees follow extreme branches, one the minimum and the other the maximum. There still may be a narrow area of intersection.

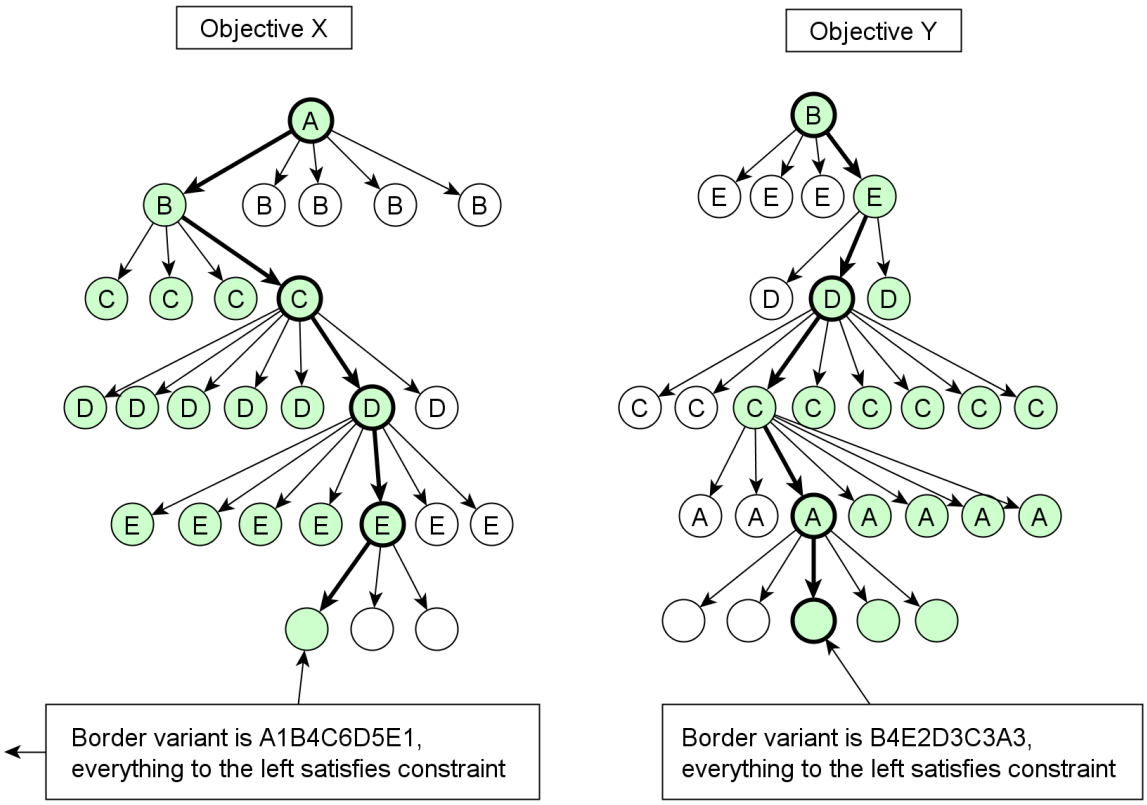
7. Move pointer m to the next level down (for objective X).

8. If now m = n check the specific number of resource for both objectives.

9. If they're not equal, FAIL.

10. If the numbers are equal, then continue looking.

    Below is an example, both top resource (A and B) lie on the borders of corresponding trees. Because in X tree the border variant follows A1**B4…** and B4 is the top resource for border variant for objective Y, there is still a possibility of overlap. If border variant for objective X would have had anything but B4 in it, it would be blocked by objective Y's constraint. In this particular example, A1B4C3D3E3 fits both objectives' constraints.

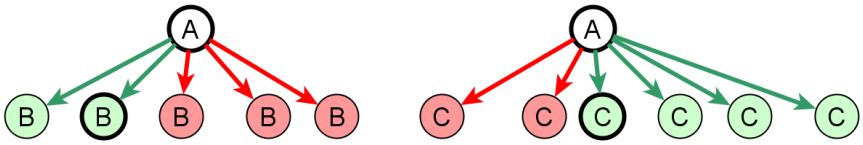


11. Advance pointer n to the next resource (in objective Y tree). Clearly, if there is at least on full bush included in permissible area, then DONE. In above example, next level for objective Y is resource E. It has a full bush stemming from E3 node, which includes all possible permutations of remaining resources (including the A1 limitation that comes from objective X).

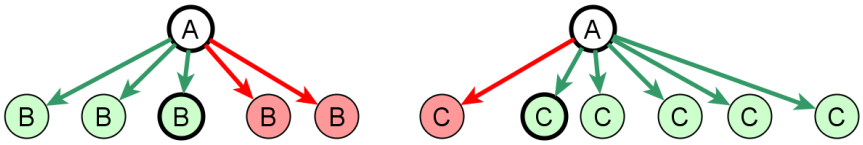
12. If no full bush is available on the current level (i.e the border variant goes through the max or min branch of current resource), then advance m pointer to next resource down and go to step 7.

13. If two current resources are the same, check the resource number: if the number for objective X is less than for objective Y, then FAIL (the areas do not intersect).
Here is an example, for one objectives A1 and A2 are permissible, but for the other only A3, A4, A5 are permissible, clearly, there is not intersection on this level and it does not matter what happened below, as one objective's constraint prunes the

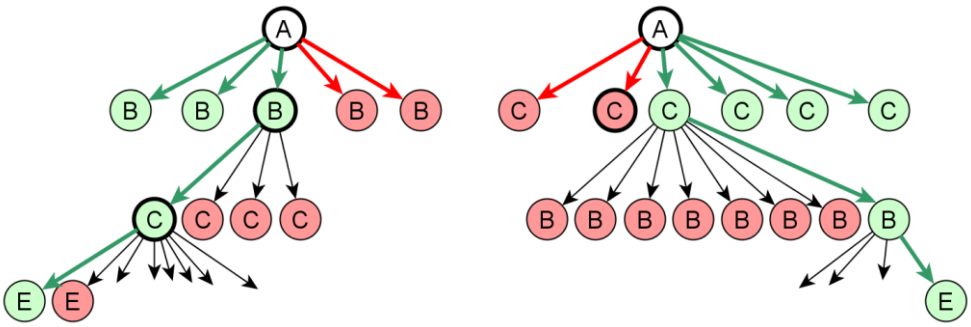
other's.



14. Else, if there is at least 2 branches overlap in both objectives permissible areas, then DONE.



15. Else, if the intersection is only one branch and keep looking (depending on remaining composition of border variants for both objectives, there may or may not be intersection. For example, here is the situation where there is no intersection:



So, to check if there is or isn't intersection further down, move pointers m and n on to next levels and go to step 3.

Bibliography

[1] Jasbir S. Arora and R. T. Marler, "Survey of multi-objective optimization methods for engineering," *Structural and Multidisciplinary Optimization*, vol. Vol. 26, No. 6, pp. 369-395, 2004.

[2] J. Christopher Beck, Patrick Prosser, and Evgeny Selensky, "Vehicle Routing and Job Shop Scheduling: What's the difference?," *Proceedings of the 13th International Conference on Artificial Intelligence Planning and Scheduling*, pp. 267-276, June 2003.

[3] G. Beltrame, L. Fossati, and D. Sciuto, "Desicion-Theoretic Design Space Exploration of Multiprocessor Platforms," *IEEE Tras. on Comput.-Aided Des. Integr. Syst.*, vol. 29, no. 7, pp. 1083-1095, July 2010.

[4] Stephen P. Bradley, Arnoldo C. Hax, and Thomas L. Magnanti, "Integer Programming," in *Applied Mathematical Programming*. Reading, MA: Addison-Wesley, 1977, ch. 9, pp. 288-304.

[5] Burcin Cakir, Fulya Altiparmak, and Berna Dengiz, "Multi-objective optimization of a stochastic assembly line balancing: A hybrid simulated annealing algorithm," *Computers & Industrial Engineering*, vol. 60, no. 3, pp. 376–384, April 2011.

[6] Carlos A Coello Coello, Gary B Lamont, and David A Van Veldhuizen, *Evolutionary algorithms for solving multi-objective problems*, 2nd ed. Boston, MA, USA: Springer, 2007.

[7] Carlos A. Coello Coello and Maximino Salazar Lecuga, "MOPSO: a proposal for multiple objective particle swarm optimization," in *Congress on Evolutionary Computation*, vol. 2, Honolulu, Hawaii, 2002, pp. 1051-1056.

[8] Carlos A. Coello Coello, Gregorio Toscano Pulido, and San Pedro Zacatenco, "A Micro-Genetic Algorithm for Multiobjective Optimization," in *Evolutionary Multi-Criterion Optimization (EMO)*, Zurich, Switzerland, 2001, pp. 126-140.

[9] Kalyanmoy Deb, "Introduction to Evolutionary Multiobjective Optimization," in *Multiobjective Optimization, Interactive and Evolutionary Approaches*, J. Branke et al., Eds.: Springer, 2008, vol. 5253, pp. 64-69.

[10] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multi-objective genetic algorithm: NSGA-II," *Transactions on Evolutionary Computation*, vol. Vol.2,

No. 6, pp. pp.182-197, April 2002.

[11] Kalyanmoy Deb and N. Srinivas, "Muiltiobjective Optimization Using Nondominated Sorting in Genetic Algorithms," *Evolutionary Computation, Vol. 2, No. 3*, pp. 221-248, 1994.

[12] Wang Dong-Feng and Xu Feng, "Multi-objective Optimization Problem with Arena Principle and NSGA-II," *Information Technology Journal*, vol. 2, no. 9, pp. 381-385, 2010.

[13] Committee Draft. (2011, April) INTERNATIONAL STANDARD ISO/IEC 9899:201x. [Online]. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf

[14] Juan J. Durillo and Antonio J. Nebro. (2012, May) jMetal - A Framework for Multi-Objective Optimization. [Online]. http://jmetal.sourceforge.net/index.html

[15] Brandon Kerry Eames, *A finite Domain Model For Design Space Exploration (PhD disscertation)*. Nashville, Tennessee: retrived from ProQuest Dissertations & Theses, 2005.

[16] Matthias Ehrgott and Xavier Gandilbleux, "Multiple objective combinatorial optimization - a tutorial," in *Multi-objective Programming and Goal Programming: Theory and Applications*, Tetsuzo Tanino, Tamaki Tanaka, and Masahiro Inuiguchi, Eds. Berlin, Germany: Sprinher-Verlag, 2003, pp. 3-7.

[17] Matthias Ehrgott and Stefan Ruzika, "An Improved Epsilon-Constraint Method for Multiobjective Programming," Mathematik, Technische Universität Kaiserslautern, pre-print 96, 2005.

[18] Matthias Ehrgott and David M. Ryan, "The method of elastic constraints for mutliobjective combinatorial optimization and its application in airline crew scheduling," in *Mutli-Objective Programming and Goal-Programming: Theory and Applications*, Tetsuzo Tanino, Tamaki Tanaka, and Masahiro Inuiguchi, Eds. Berlin, Germany: Springer-Verlag, 2003, pp. 117-122.

[19] Andries P Engelbreacht, *Fundamentals of Computational Swarm Intelligence*.: Hoboken, NJ: Wiley, 2005.

[20] Fabrizio Ferrandi, Pier Luca Lanzi, Daniele Loiacono, Christian Pilato, and Donatella Sciuto, "A Multi-Objective Genetic Algorithm for Design Space Exploration in High-Level Synthesis," in *IEEE Computer Society Annual Symposium on VLSI*, Milano, Italy, 2008, pp. 417 - 422.

[21] Michael J. Folk, Bill Zoellick, and Greg Riccardi, "Hashing & Extendible Hashing," in *File Structures: An Object-Oriented Approach with C++*, 3rd ed.: Addison-Wesley, 1998, ch. 11,12, pp. 463-480, 524-528.

[22] Juan Carlos Fuentes Cabrera and Carlos A. Coello Coello, "Micro-MOPSO: A Multi-Objective Particle Swarm Optimizer That Uses a Very Small Population Size," in *Multi-Objective Swarm Intelligent Systems*, Leandro dos Santos Coelho, Ed. Berlin, Germany: Springer-Verlag, 2010, vol. Studies in Computational Intelligence (SCI) 261, pp. 83-104.

[23] Brent Fulgham. (2012, July) The Computer Language Benchmark game. [Online]. http://shootout.alioth.debian.org/u32/benchmark.php?test=all&lang=java

[24] Brent Fulgham. (2013) The Computer Language Benchmarks Game. [Online]. http://benchmarksgame.alioth.debian.org/

[25] Vinod Kathail et al., "PICO: Automatically Desgining Custom Computers," *Computer*, vol. 35, no. 9, pp. 39-47 , September 2002.

[26] (2013, Apr.) KDE Documentation. [Online]. http://docs.kde.org/development/en/kdeedu/rocs/import-export-graphs.html

[27] Leonid G. Khachiyan, "Polynomial algorithms in linear programming," *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, no. 20, pp. 51–68, 1980 (English translation: USSR Computational Mathematics and Mathematical Physics, 20 (1980), pp. 53–72.).

[28] Lev Grigorievitch Kirischian, ""Исследование и разработка метода выбора структур вычислительных систем с перестаиваемой структурой" (Study and development of method to choose design of reconfigurable computational systems), Ph.D. thesis, Dept. of Computational Systems," *Institute of Energy*, 1984.

[29] Lev Kirischian, Valeri Kirischian, Irina Terterian, and Vadim Geurkov, "Multi-parametric optimisation of the modular computer architecture," *International Journal of Technology, Policy and Management*, vol. 6, no. 3, pp. 327-346, 2006.

[30] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671-680, May 1983.

[31] Donald E. Knuth, "Vol. 3. Sorting and Searching," in *The Art of Computer Programming*, 2nd ed. Reading, Massachusetts, Massachusetts: Addison Wesley, 1998, vol. 3, ch. 6.2.1, pp. 419-422.

[32] Ali Kokhazadeh and Omid Fatemi, "Design Space Pruning of MPSoCs Using Weighted Sub-sampling," in *18th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2011, pp. 550- 553.

[33] Vyas Krishnan and Srinivas Katkoori, "A genetic Algorithm for the Design Space Exploration of Datapaths During High-Level Synthesis," *IEEE Transations on Evolutionary Computation*, vol. 10, no. 3, pp. pp.213-229, June 2006.

[34] Marco Laumanns, Lothar Thiele, and Eckart Zitzler, ""An efficient, adaptive parameter variation scheme for metaheuristics based on the epsilon-constraint method"," *European Journal of Operational Research*, vol. 169, pp. 923-924, 2006.

[35] Mikulás Luptácik, "Multiobjective Linear Programming," in *Springer Optimization and Its Applications: Mathematical Optimization and Economic Analysis*. New York: Springer, 2010, vol. 36, ch. 8, pp. 243-269.

[36] Guillaume Marceau. (2009, May) Guillaume Marceau's blog away from home. [Online]. http://blog.gmarceau.qc.ca/2009/05/speed-size-and-dependability-of.html

[37] Silvano Martello and Paolo Toth, *Knapsack Problems*. New York: Wiley, 1990.

[38] G. Mavrotas and D. Diakoulaki, "A branch and bound algorithm for mixed zero-one multiple objective linear programming," *European Journal of Operational Research*, vol. 107, no. 3, pp. 530–541, June 1998.

[39] Kaisa Miettinen, *"A Posteriori Methods" in Nonlinear Multiobjective Optimization in International Series in Operations Research & Management*, Frederick S. Hiller, Ed. Norwell, MA: Kluwer Academic Publishers, 1999, vol. 12.

[40] Kaisa Miettinen, Francisco Ruiz, and Andrzej P. Wierzbicki, "Introduction to Multiobjective Optimization: Interactive Approaches," in *Multiobjective Optimization, Interactive and Evolutionary Approaches in Lecture Notes on Computer Science*, J. Branke et al., Eds.: Springer, 2008, vol. 5252, pp. 27-57.

[41] R.J. Mullen, D. Monekosso, S. Barman, and P. Remagnino, "A review of ant algorithms," *Expert Systems with Applications*, vol. 36, pp. 9608–9617, 2009.

[42] (2013, March) Multiobjective Genetic Algorithm Solver. [Online]. http://www.mathworks.com/products/global-optimization/description5.html

[43] Ohloh. (2013, February) Monthly commits (in lines of code, to open source projects). [Online]. https://www.ohloh.net/languages/compare

[44] J. OuYang, F. Yang, S.W. Yang, and Z.P. Nie, "The improved NSGA-II approach," *Journal of Electomagnetic Waves and Applications*, vol. 22, pp. 163-172, 2008.

[45] Bijaya Ketan Panigrahi, Yuhui Shi, and Meng-Hiot Lim, Eds., *Adaptation, Learning, and Optimization : Handbook of Swarm Intelligence (Concepts, Principles and Applications)*, 1st ed. Berlin/Heidelberg, Germany: Springer, 2011.

[46] (2011, April) Programming Language Popularity. [Online]. http://www.langpop.com/

[47] Peter Salamon, Paolo Sibani, and Richard Frost, *Facts, conjectures, and improvements for simulated annealing*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2002.

[48] Reza Sedaghat, Anirban Sengupta, and Zhipeng Zeng, "A high level synthesis design flow with novel approach for efficient design space exploration in case of multi-parametric optimization objective," *Microelectonics Reliability*, vol. 50, no. 3, pp. 424-237, 2010.

[49] Reza Sedaghat, Anirban Sengupta, and Zhipeng Zeng, "Rapid design space exploration by hybrid fuzzy search approach for optimal architecture determination of multi objective computing systems," *Microelectronics Reliability*, vol. 51, pp. 502-512, 2011.

[50] Aniban Sengupta, "A fast design space exploration based on priority factor for multi parametric optimized high level synthesis design flow," Dept. Elect.Eng, Toronto, ON, Canada, 2010.

[51] Anirban Sengupta, Reza Sedaghat, and Zhipeng Zeng, "Multi-level efficient design space exploration and architectural synthsis of an application specific processor," *Microprocessors and Microsystems*, no. 35, pp. 392-404, 2011.

[52] Greg Snider, "Spacewalker: Automated Design Space Exploration for Embedded Computer Systems," Palo Alto, CA, Tech. Rep. HPL-2001-220, Sept. 10, 2001.

[53] A. Suppapitnarm, K.A. Seffen, G.T. Parks, and P.J. Clarkson, "Simulated annealing algorithm for multiobjective optimization," *Eng.Opt.*, vol. 33, no. 1, pp. 59-85, 2000.

[54] Suqin Tang, Zixing Cai, and Jinhua Zheng, "A Fast Method of Constructing the Non-Dominated Set: Arena's Principle," in *Natural Computation, 2008. ICNC '08. Fourth International Conference on*, Jinan, 2008, pp. 391 - 395.

[55] Gamze Kilincli Taskiran, "An Improved Genetic Algorithm for Knapsack Problems," Dayton, OH, USA, M.S. Thesis 2010.

[56] Lam Thu Bui and Sameer Ricardo Alam, *Multi-objective Optimization in Computational Intellegence (Theory and Practice)*. Hershey, PA: Information Science Regerence (an imprint of IGI Global), 2008.

[57] Santosh Tiwari, Georges Fadel, and Kalyanmoy Deb, "AMGA2: improving the performance of the archive-based micro-genetic algorithm for multi-objective optimization," *Engineering Optimization*, vol. 43, no. 4, pp. 377-401, April 2011.

[58] Santosh Tiwari, George Fadel, Patrick Koch, and Kalyanmoy Deb, "AMGA: An archieve-based Micro Genetic Algorithm for Multi-objective Optimization," in *10th annual conference on Genetic and evolutionary computation*, Atlanta, GA, 2008, pp. 729-736.

[59] M. Visée, J. Teghem, M. Pirlot, and E.L. Ulungu, "Two-phases Method and Branch and Bound Procedures to Solve the Bi–objective Knapsack Problem," *Journal of Global Optimization*, vol. 12, no. 2, pp. 139-155, March 1998.

[60] Zhipeng Zeng, *"High level synthesis design flow for multi parametric optimization with hybrid hierarchical design space exploration", M.S. Thesis, Dept.Elect.Eng., Ruerson Univ.,Toronto, Ontario, Canada*., 2010.

[61] Ackart Zitzler, Marco Laumanns, and Lothar Thiele, "SPEA2: Improving the Strength Pareto Evolutionary Algorithm," Swiss Federal Institute of Technology (ETH), Zurich, Technical report 103 [Online] May 2001.