Masarykova univerzita
Fakulta informatiky

Evolutionary optimization of intrusion detection system in wireless sensor networks

Diploma thesis

Adam Saleh

Brno, December 2013

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Adam Saleh

Advisor: Andryi Stetsko, Ph.D.

Acknowledgement

I would like to thank my supervisor ...

Abstract

Keywords

Wsn,ids,Spea2,NSGAII

Contents

1 Introduction

Wireless sensor networks are relatively new concept describing a wirelessly communicating network of battery powered nodes, that contain some computational capacity and are able to gather information with attached sensors. Current research hints at applications in monitoring wild-life and gathering information in otherwise unattainable places. They are an interesting topic for security research because of their possible military and intelligence gathering applications, their distributed nature and the constrains on their hardware.

In this work we concern ourselves with the problem of setting up an intrusion detection system on network of nodes so that it satisfies constrains on accuracy, battery life and memory. This can be a time consuming process. With the usage of simulation frameworks, such as MiXiM, effort spent on optimization is directly proportional to invested computational resources, with realistic simulations taking up to several hours of processor time.

Achieving high accuracy and low memory footprint are in many cases inversely proportional. This means that the network operator has to chose a trade-off between desired objectives. We believe that this process of optimization and evaluation of different trade-offs could be greatly improved by using multi-criteria algorithms. Multi-criteria algorithms instead of a single solution return a set of candidates covering different trade-offs between the objectives. This allows for better evaluation of said objectives.

Evolutionary algorithms help immensely with reducing demands for computational resources while optimizing problems. Because evolutionary optimization works by iterating evaluations and modification of a set of candidates, algorithms have been successfully modified to solve multi-optimization problems.

We explain the basic ideas behind evolutionary heuristics in chapter 2,as well as their applicability on concepts of multi-criteria optimization in terms of Pareto optimality, as well as the main principles of three state of the art multi-criteria evolutionary algorithms.

In chapter 3 we introduce our black-box micro-framework that is based on Paradiseo evolutionary framework. Because our framework is partially written in Paradiseo, we cover some of its design decisions in

grater detail. Then we have provided a walk through of all the configuration entry points of our micro-framework, complete with examples in Python.

In the chapter 4 we discuss intrusion detection systems for wireless sensor networks. We introduce one such IDS that we use as a model problem to optimize with our evolutionary framework.

We discuss calibration of the evolutionary framework in chapter 5. Because evolutionary algorithms are heuristic in nature, we provide several metrics that can be used to guide the settings. Based on a large precomputed problem dataset we then try to find the most robust default setup of our framework.

And in chapter **??** provide a model example of optimizing such wireless sensor network application,using parallel evaluation on Boinc cluster.

2 Evolutionary Optimization

TODO decision space, Objective space, parameters,objective vector, evaluation

Definition 1 (Optimization problem)**.** *We define optimization problem as a triplet* (X, Z, f)*, where* $X \subseteq \mathbb{R}^n$ *is the decision space,* $Z \subseteq \mathbb{R}^m$ *is the objective space and* $f : X \to Z$ *is the evaluation function. Without loss of generality it is assumed, that we want to minimize each objective.*

This definitions allows the evaluation function f to output incomparable objective vectors, which is a key concept for multi-criteria optimization. Optimization problem can be easily made compatible with single-criteria algorithms by defining cost-function $c : Z \to \mathbb{R}$, that creates an aggregated metric from the objective vector.

Because of the nature of wireless communication, where interference in shared medium has to be taken into account, it is hard to predict how will a change in configuration impact the quality of transmission. Therefore thorough and expensive simulation is required for every configuration that we would want to evaluate. This renders the usage of simple exhaustive search for optimization purposes impractical. Optimization problems that have problem space that is expensive to search, are good fit for heuristic approaches [9].

2.1 Basic principles of meta-heuristics

Because of their nature, heuristic approaches are hard to evaluate and reason about, therefore they are often considered to be a method of last resort. Decision to use heuristics lazily on problems where more deterministic approaches (such as using a SAT-solver or linear programming) would be more efficient may become a costly one. Heuristic approaches are usually considered when we need to solve our problem in order of magnitude faster than possible using conventional approaches, while sacrificing guarantees on optimality.

Most of the heuristics used are variants on search through problem space, with different approaches to avoid undesirable local optima. There are two basic principles that are used when designing a meta-heuristic[9]:

2.1.1 Diversification

An algorithm should explore as much of the search space as possible. This principle is called diversification.

Random search can be considered an extreme application of diversification, where in every round we generate next solution randomly without using any information from previous (good) solutions. Less extreme example is the family of population based heuristics, where in each iteration we are trying to improve a set of candidates, called a population.

Problem spaces with many local optima are less likely to cause problems for algorithms that have strong diversification component.

2.1.2 Intensification

An algorithm should exploit information from the solutions it previously found. This principle is called intensification, because it covers the notion of trying to intensify a good solution to find a better one.

Local search can be considered an extreme application of intensification, where next instance to evaluate is always selected deterministically, based on the previous solution. Local search and its variants belong to a family of single-solution based heuristics, where in each iteration we are basing our search of the problem-space of single solution.

2.2 Evolutionary heuristics

In our experiments, we were constrained by a number of configuration evaluations we are able to perform in a given time frame. We are focusing on evolutionary heuristics, because they seem to have reasonable performance with regard of optimizing wireless sensor networks.[7] These are modeled on natural process of evolution of species. It is a family of stochastic, population-based heuristics (as opposed to deterministic, single-solution based).

2.3 Single objective evolutionary algorithms

Evolutionary algorithms are based on the fact that natural process of evolution can be considered an algorithm solving an optimization

problem of adapting a species to its environment.

2.3.1 Evolutionary algorithm

Basic principles behind evolutionary algorithms can be shown on this template[9]:

Step 0: Initialization: generate initial population of individuals. Each individual encodes a potential solution to our optimization problem. Then we use evaluation function to calculate their fitness.

Step 1: Variation: apply mutation and crossover funcions on the individuals to generate new offsprings. Variation is there to explore the search space, where:

Mutation of individual usually consists of randomly selecting some configuration from its neighborhood as its offspring.

Crossover usually consists of creating an offspring by permutation of two individuals.

This step provides diversification for evolution heuristic.

Step 2: Fitness assignment: calculate the fitness values of the offspring in population

Step 3: Environmental selection: select the best individuals of a population to "survive" to the next generation. Environmental selection chooses individuals based on their fitness score. This should provide convergence to configurations that are more optimal. This step provides intensification for evolutionary heuristic.

Step 4: Termination: check if termination criterion holds (usually based on number of generations), if it does, return the result and stop, if it doesn't, increment the generation counter and proceed with step 1.

In algorithms, there are variations on this template, for example, instead of first creating the new generation and then selecting the best individuals for next iteration, *environmental selection* can precede *variation*. In this case selection creates a subset of population called mating

pool. Variation generates the rest of the population using only individuals from mating pool. Some algorithms use this kind of mating selection as an implicit step of variation, with environmental selection applied later on.

The template is then realized into an actual algorithm by defining all the steps. For example, in our experiment, initialization is done by uniformly selecting configurations from decision space, selection and variation steps are selected by the framework based on the evolutionary algorithm.

Crossover and mutation functions on the other hand are to be calibrated for each optimization problem on case by case basis.

2.4 Multi-objective evolution algorithms

Problem that we are trying to optimize is unfortunately ill-suited for single objective evolution. Objectives such as minimizing memory consumption while maximizing IDS accuracy are orthogonal, therefore the usual solution is to provide some sort of weighted average. In this case,because output of our algorithm would be only a single solution, we would need to know emphasis on different criteria before we run our optimization.

Better approach is to recognize multi-objective nature of our problem, and use algorithms that return a set of candidates covering different trade-offs between the objectives.

Multi objective algorithms are based on idea of Pareto optimality and domination.

Definition 2. *Pareto dominance. An objective vector* $u = (u_1, \cdots, u_n)$ *is said to dominate* $v = (v_1, \cdots, v_n)$ *(denoted by* $u \prec v$*) if and only if no component of* v *is smaller*[1] *than the corresponding component of* u *and at least one component of* u *is strictly smaller:*

$$\forall i \in \langle 1, n \rangle : u_i \leq v_i \wedge \exists i \in \langle 1, n \rangle : u_i < v_i$$

Definition 3. *Pareto optimality. A solution* x *in decision space* S *is Pareto optimal if for every* $x' \in S$*,resulting evaluation* $F(x')$ *does not*

1. we assume minimization

dominate $F(x)$, that is

$$\forall x' \in S, F(x) \not\prec F(x')$$

Subset of solutions where no solution from the super-set dominate the one in the set is called Pareto optimal set.

Definition 4. *Pareto optimal set For given multi-objective problem MOP(F,S) and given decision space S and evaluation function F, the Pareto optimal set is defined as*

$$\mathcal{P}^* \subseteq S, \forall x \in \mathcal{P}^* \nexists x' \in \mathcal{P}^*, F(x') \prec F(x)$$

Pareto front is the largest Pareto optimal set in the solution space. It is the set of all non-dominated solutions the solution space.

Definition 5. *Pareto front For given MOP(F,S), the Pareto optimal set is defined as $\mathcal{PF} = \{x \in S | \nexists x' \in S, F(x') \prec F(x)\}$*

We will call any Pareto optimal set that is not a Pareto front a Pareto approximation.

Multi-objective heuristics then try to obtain best approximation of Pareto front. To gouge how good an approximation of Pareto front is, we usually use two criteria, first to measure convergence to Pareto optimal front and second to measure diversity in found Pareto set.

Calibration of multi objective evolution algorithms is hard,because it is hard to reason about the convergence of output without having the real Pareto optimal front in the first place. Pareto optimal sets usually aren't directly comparable, but there are several metrics that can be used to compare them, some of them used in following algorithms.

2.5 NSGAII

Second generation of the non-dominated sorting genetic algorithm[2] is currently the most popular multi-objective heuristic. It improved on its previous iteration in several ways, most notable is the exclusion of sharing parameter δ_{share}, that previously needed to be specified to maintain good diversity in final population.

Step 0: Initialization: Generate an initial population P_0

Step 1: Fitness assignment: calculate the fitness values of individuals P_0

Step 2: Variation: Generate temporary population P'_t by applying crossover and mutation operators on P_t

Step 3: Fitness assignment: calculate the fitness of P'_t

Step 4: Non-dominated sort: sort the $P_t \cup P'_t$, first by domination rank, second by crowding metric

Step 5: Environmental selection: create P_{t+1} as the first N individuals of the sorted $P_t \cup P'_t$

Step 6: Termination: If $t \geq T$, or other stopping criterion is satisfied, return non-dominated individuals from P_{t+1}, else increment t and continue with step 2.

We separate step 4, the non-dominated sort into two phases:

2.5.1 Non-dominated sorting

Every individual is assigned a rank based on order of non-dominated front it belongs to. The first front is the Pareto front of the population:

$$\mathcal{F}_1 = x \in P| \nexists x' \in P, x' \prec x$$

Other can be defined recursively as the Pareto front of the remaining population not containing previous fronts $R_i = P - \bigcup_{1 \leq j < i} \mathcal{F}_j$:

$$\mathcal{F}_1 = x \in R_i| \nexists x' \in R_i, x' \prec x$$

2.5.2 Crowding distance sorting

Individuals in the same rank are then sorted by their crowding distance. First crowding distance per objective of each individual is calculated. Aggregated crowding distance of the individual is then the sum through all the objectives.

Let $\mathcal{P}_m$ be a non-dominated set ordered by objective m, and $\mathcal{P}_m[i]$ the value of objective m of individual i, then we can recursively. define individuals crowding distance C_i:

$$C_i = \sum_m C_{i_m}$$

, where

$$C_{i_m} = \frac{\mathcal{P}_m[i+1] - \mathcal{P}_m[i-1]}{f_m^{max} - f_m^{min}}$$

It has to be noted, that for aggregation to work, each C_{i_m} is normalized to be a fraction between 0 and 1.

First sorting helps with general convergence to real Pareto front, while second improves diversity in the non-dominated part of the population. To get the result we just need to extract non-dominated results from the final population.

2.6 Spea2

Strength Pareto Evolutionary Algorithm[12], unlike previous NSGAII, stores non-dominated individuals in archive separate from the rest of the population. Therefore, after new population is generated, new contents of archive are determined in two passes, first using procedure to ensure convergence and then procedure to ensure diversity.

Let us denote:

P_t to be the population in generation t, $\overline{P_t}$ the corresponding archive of non-dominated individuals, N the population size,$\overline{N}$ the archive size and T the maximum number of generations.

The overall algorithm then is as follows:

Step 1: Initialization: Generate an initial population P_0 and create the empty archive $\overline{P_0}$, set $t = 0$

Step 2: Fitness assignment: calculate the fitness values of individuals in $P_t \cup \overline{P_t}$

Step 3: Environmental selection: Copy all non-dominated individuals in $P_t \cup \overline{P_t}$ to $\overline{P_{t+1}}$. If size of $\overline{P_{t+1}}$ exceeds $\overline{N}$, then remove exceeding individuals with worst fitness, otherwise, if size of $\overline{P_{t+1}}$ is less than $\overline{N}$, fill $\overline{P_{t+1}}$ with dominated individuals with the best fitness values $P_t \cup \overline{P_t}$

Step 3: Termination: if $t \geq T$ or another stopping criterion is satisfied, then stop and return the the non-dominated individuals from $\overline{P_{t+1}}$ as a result

Step 5: Mating selection: generate the mating pool M

Step 6: Variation: apply crossover and mutation operators to the mating pool and set P_{t+1} to the resulting population. Increment generation counter and go to Step 1.

2.6.1 Fitness assignment

Fitness of an individual is an aggregated metric composed of strength of individual and density estimation. Strength of an individual i is the number of individuals from $P_t \cup \overline{P_t}$ it dominates:

$$S(i) = |\{j | j \in P_t \cup \overline{P_t} \wedge i \prec j\}|$$

Raw fitness of i is then aggregated from all strengths of individuals that dominate i:

$$R(i) = \sum_{j \in P_t \cup \overline{P_t}, j \prec i} S(j)$$

To distinguish between individuals that do not dominate each other algorithm uses a density metric based on the distance of k-th nearest neighbor of i, denoted δ_i^k. As a common setting, k equal to the square root of the population size is used. Thus density is defined by:

$$D(i) = \frac{1}{\delta_i^k + 2}$$

Because of the two added in the denominator $0 < D(i) < 1$. Aggregate fitness of an individual is then the sum of its raw fitness and density:

$$F(i) = R(i) + D(i)$$

Because the raw fitness has integer values, sorting individuals by the fitness metric yields similar results to the two pass sorting of NSGAII.

2.6.2 Environmental selection

Environmental selection step depends on the number of non-dominated individuals in the $P_t \cup \overline{P_t}$. This is the set of individuals with fitness lower than one:

$$P'_{t+1} = \{i | i \in P_t \cup \overline{P_t} \wedge F(i) < 1\}$$

If $|P'_{t+1}| < \overline{N}$ it is sufficient to set $\overline{P_t + 1}$ to contain the best $\overline{N}$ individuals from $P_t \cup \overline{P_t}$ based on the fitness.

If $|P'_{t+1}| > \overline{N}$ a trimming procedure is used that is based on the density in the P'_{t+1} set, as opposed to $P_t \cup \overline{P_t}$ used for fitness metric.

Iteratively, individual that has minimum distance to another individual is chosen to be removed. If there are several individuals with minimum distance, tie is broken by considering their second smallest distances and so forth.

2.7 MOGA

Multi objective genetic algorithm is the oldest algorithm we have included. It originated the template used by NSGA and later on NSGAII. The main difference is in the calculation of convergence and diversity metrics.

Step 0: Initialization: generate an initial population P_0

Step 1: Fitness assignment: calculate the fitness values of individuals P_0

Step 2: Variation: generate temporary population P'_t by applying crossover and mutation operators on P_t

Step 3: Domination ranking: for each individual count number of solutions that dominate it

Step 4: Diversity assignment: is calculated for each individual based on number of individuals in its neighborhood

Step 5: Environmental selection: create P_{t+1} as the first N individuals of the sorted $P_t \cup P'_t$. Sorting is based primarily on domination ranking, with diversity assigment helping to order individuals with equal rank.

Step 6: Termination: If $t \geq T$, or other stopping criterion is satisfied, return non-dominated individuals from P_{t+1}, else increment t and continue with step 2.

Convergence metric in step 3 and diversity preserving metric in step 4 warrant a closer look:

2.7.1 Domination ranking

Similarly to NSGAII, population is first sorted into ranks based on the number of solutions that dominate given individual. While in NSGAII the individual was sorted into a rank based on a layer it belonged to, in MOGA each individual is directly assigned the count of individuals that dominate it.

$$f(i) = |\{j | j \in P \wedge j \succ i\}|$$

2.7.2 Sharing diversity metric

This metric counts how crowded is the space around the select individual, based on the δ_{share} parameter.

If individual i has neighbor j at euclidean distance $|i - j|$, it adds to compound sharing metric by

$$Sh(|i-j|) = \begin{cases} 1 - (\frac{d}{\delta_{share}})^{\alpha} & \text{if} d < \delta_{share} \\ 0 & \text{else} \end{cases}$$

For each individual we sum the sharing coefficients of those that share its domination ranking.

$$S(i) = \sum_{j \in P \wedge f(j) = f(i) \wedge i \neq j} Sh(|i-j|)$$

As we can see, only neighbors closer than δ_{share} can make the fitness of i worse. Value of δ_{share} therefore has great influence on the resulting

Pareto-approximation. Because of this complexity that δ_{share} brings to configuration of the algorithm, MOGA is rarely used.

2.8 Ibea

Both NSGAII and Spea2 differ mostly in their approach to characterize the convergence to Pareto front and diversity. Indicator-Based evolutionary algorithm [11] takes more abstract approach, based on a concept of binary quality indicators. Indicator is a function, that takes two Pareto optimal sets and outputs some quantification of a difference between the two.

Zitzler and Kunzli proposed two indicators that are discussed below:

2.8.1 ϵ-indicator

Additive ϵ-indicator quantifies the minimal distance the Pareto set A has to be moved in each dimension in objective space such that the second Pareto set B is weakly dominated.Formal definition:

$$I_{\epsilon^+}(A,B) = min_\epsilon\{\epsilon|\forall x_1 \in A, \forall x_2 \in B, \forall m \in M : P_m[x_i] + \epsilon < P_m[x_2]\}$$

If A dominates B, resulting indicator will be negative.

2.8.2 Hyper-volume based indicator

Hyper-volume-distance indicator, that quantifies how much volume is dominated by the first Pareto set but not dominated by the second, with respect to predefined reference point Z. Hyper-volume $H(A)$ of a Pareto approximation A is the total volume of n-dimensional space that is enclosed by the individual results and the reference point Z. That is, hyper-volume of a set is the total volume of space dominated by the sets individuals.

Hyper-volume indicator then compares two Pareto approximations:

$$I_{HD}(A,B) = \begin{cases} H(B) - H(A) & \text{if} \forall x_1 \in A, \forall x_2 \in B, x_2 \prec x_1 \\ H(B \cup A) - H(A) & \text{else} \end{cases}$$

Hyper-volume is considered to be the best single value indicator of how good a Pareto approximation is, because it aggregates both convergence and diversity metrics. Primarily it is a convergence metric, though if the Pareto front of the solution space is is linear in nature, hyper-volume indicator will lead to optimal diversity of approximations as well. [1]

Both of these indicators are dominance preserving.

Definition 6. *A binary quality indicator is denoted as dominance preserving if* $\forall x_1, x_2, x_3 \in Z$*:*

(i) $x_1 \prec x_2 \Rightarrow I(x_1, x_2) < I(x_2, x_1)$*, and*

(ii) $x_1 \prec x_2 \Rightarrow I(x_3, x_1) > I(x_3, x_2)$

Because any single individual can be considered a Pareto set of size one, IBEA uses given indicator to quantify fitness of an individual:

$$F(x_1) = \sum x_2 \in P - x_1 a(I(x_1, x_2), k)$$

, where $a(i, k) = -e^{-\frac{i}{k}}$ is function that amplifies fitness of dominating individuals.

2.8.3 IBEA algorithm template

Algorithm then goes as follows:

Step 1: Initialization: Generate an initial population P_0, set $t = 0$

Step 2: Fitness assignment: calculate the fitness values of individuals in P_t based on the chosen indicator

Step 3: Environmental selection: iterate the following steps until $|P_t| < N$:

1. find $x \in P_t$ that has minimal fitness and remove it from P_t
2. recalculate the fitness of the remaining population,

Step 3: Termination: if $t \geq T$ or another stopping criterion is satisfied, then stop and return the result as the non-dominated individuals from P_{t+1}

Step 5: Mating selection: generate the temporary mating pool Ma with binary tournament selection on P

Step 6: Variation: apply crossover and mutation operators to the mating pool and set P_{t+1} to the resulting population. Increment generation counter and go to Step 2.

TODO

Paradiseo implementation of IBEA uses adaptive scaling of the amplification function parameter, therefore calibration of the k is not as important. On the other hand, choosing appropriate indicator is crucial.

2.9 Choosing the algorithm

All of these algorithms give some guarantees on convergence to Pareto front. Two of them are incremental improvements on their predecessor.

MOGA is the oldest algorithm of the four, but while it uses deprecated diversity metric, its convergence metric can not be entirely dismissed as inferior to the rest.

Right now, NSGAII is considered to be the standard benchmark for MOEA algorithms,both Spea2 and IBEA are being compared against it in their originating papers . It is the oldest of the three and therefore the most widely used one. Because there are no algorithm specific variables, it has the easiest calibration.

While Spea2 is newer, it doesn't mean it is better in every instance. It has to be noted, that even comparison between these two algorithms with regards to optimizing IDS for WSN came out inconclusive.[7] On the other hand, several papers claim they achieved better result with Spea2 than NSGAII. Similarly to NSGAII it doesn't have any algorithm specific configuration.

Of the more interesting concepts that IBEA uses is the hypervolume distance indicator, which might provide an interesting alternative to heuristics used in more popular NSGA and SPEA. Unfortunately experimental results show[11], that I_{HD} based IBEA is very sensitive to miscalibration of reference point Z. If provided with good reference point, it consistently outperformed the other tho algorithms, but optimal value of Z varied greatly based on the problem set. We will discuss the hyper-volume as a metric of Pareto set convergence in

chapter 5.

3 Our Framework

Using multi-criteria evolutionary algorithms is a viable and efficient way of optimizing wireless sensor networks[7]. Therefore we have decided to create a micro-framework, that would simplify the setup of evolutionary optimization for a particular problem. Our micro-framework combines the Paradiseo framework and system for distributed computation Boinc with Python as a glue language. In our experiments we then use this evolutionary micro-framework to optimize wireless sensor network simulated by MiXiM package for Omnet++ simulation framework. Because one of the results of our work is the amalgamation of evolutionary framework and the WSN framework, in this chapter we concern ourselves with the evolutionary parts of our framework only.

Emphasis is on:

- ease of configuration
- ability to evaluate population in parallel
- having good facilities for analyzing the result.

We provide the tarball of our framework containing sources, sample configuration and binaries for the Linux operating system[6].

3.1 Paradiseo MOEO

ParadisEO is a highly configurable framework for metaheuristics written in C++. We have chosen it because of its modularity and good library of multi-criteria algorithms.[3]

Out of the seven algorithms provided in Paradiseo, we include four of them: MOGA, NSGAII, SPEA2 and IBEA. We do not include NSGA and SPEA algorithms, because in general, they have worse performance than their successors.[2][12]

SEEA, or simple elitist evolutionary algorithm is best suited for problems with inexpensive evaluation functions, where the selection is order of magnitude more resource intensive process than evaluation of population. This is usually not the case with wireless sensor networks where it is common for evaluation of a single individual to take several

minutes. Should for any reason be an algorithm not included in our micro-framework better suited than the ones that are, it is easy to add one. Even if the algorithm we'd wanted to use wasn't present in Paradiseo framework, due to its white-box nature and modular design and large library of integrated metrics it could be easily added.

3.1.1 Design of ParadisEO MOEo

ParadisEO heavily relies on meta-templating features of C++ and operator overloading. The resulting framework is then more composition than inheritance based. This results in elegant, but sometimes deceptive syntax, where for example `popEval(empty_pop, _pop);` might look like a function call, but in reality it is using the `operator()` method of previously declared `popEval` object. For this reason we include a short overview of the frameworks design.

Every multi-objective evolutionary algorithm in Paradiseo extends the moeoEA class. The implementation of such algorithm is best explained on a moeoNSGAII class, probably the most popular multi-objective algorithm. Because moeoEA implements the unary functor interface, to define new algorithm we could change the definition of the the operator().

Listing 3.1: NSGAII algorithm in Paradiseo

```
virtual void operator () (eoPop <MOEOT> &_pop){
    eoPop<MOEOT> offspring, empty_pop;
    popEval(empty_pop, _pop); // a first eval of _pop
    fitnessAssignment(_pop);
    diversityAssignment(_pop);
    do{
        breed (_pop, offspring);
        popEval (_pop, offspring);
        replace (_pop, offspring);
    }while (continuator (_pop));
}
```

As you can see, the algorithm itself is a template defined by implementation of the fitnessAssigment,diversityAssignment, breed, popEval, replace and continuator functor object and eoPop population object. Therefore most of the implementation itself is done in defining which class implement a particular member of the algorithm. We can

showcase how this composing works on an example of a continuator:

Listing 3.2: Object composition in Paradiseo

```
/** a continuator based on the number of generations*/
eoGenContinue < MOEOT > defaultGenContinuator;
/** stopping criteria */
eoContinue < MOEOT > & continuator;

moeoNSGAII (unsigned int _maxGen, ...):
defaultGenContinuator(_maxGen), continuator(
    defaultGenContinuator)
```

If we use constructor that ends evaluation after certain number of generation, it first initializes defaultGenContinuator with number of generations, and then passes it as a continuator to use.

3.1.2 Population object

Population in Paradiseo is in its core an ordered vector of individuals. Because our individuals are represented by vectors of real numbers, we will describe the moeoRealVector implementation. Each individual contains its value and variables for fitness and diversity metrics. It should be noted, that moeoRealVector doesn't contain any bounds on specific parameters of individuals configuration. In Paradiseo, we define boundaries externally, with specification in initialization function and crossover and mutation functions.

3.1.3 Fitness and diversity assignment

Class moeoDominanceDepthFitnessAssignment is used to sort the population into non-dominated ranks, by updating the fitness variable of each individual. Class moeoFrontByFrontCrowdingDiversityAssignment is similarly used to set the diversity variable. The replace operator uses instances these classes to evaluate the offsprings of the population and creating the new generation.

3.1.4 Creation of new generation

Breed operator is a standard eoBreed class that gets supplied with transform-object that aggregates mutation and cross-over functions.

Similarly, the popEval function just evaluates all the new individuals in populations and assigns them objective values. The specifics of the algorithm are hidden in the replace operator of moeoElitistReplacement class:

1. recalculate fitness and diversity
2. sort the population with standard moeoFitnessThenDiversity-Comparator
3. remove exceeding population

As you can see, this implementation slightly differs from the sketch of algorithm provided in chapter 2, namely that the NSGAII in paper[2] never exceeded the population size.

3.1.5 Stopping criteria

Continuator operator provides the basic stopping criteria. The default implementation is based on limiting maximum number of generations[1], but custom continuator could use a metric that agregates fitness of individuals in population to decide whether to proceed with another step. This feature is often utilized in evolutionary implementation of decision problem algorithms, because it can help evaluate whether population contains enough information to carry out the decision. In multi-criteria optimization problems we usually don't have a definition of "good enough" solution, but if we had, we could use it to further limit the number of expensive evaluations.

3.2 Example problem

We have chosen the first Schaffer's bi-objective problem SCH1, to provide a walkthrough for a minimal C++ file. The example itself is taken from ParadisEO multi-objective evolution tutorial [**?**]. While this is a somewhat contrived example, it was used in comparing different MOEA in several papers and it is succint enough to fit on one page.

1. As hinted by the class-name *defaultGenContinuator*

The goal of the SCH1 problem is to minimize the following two objectives objectives of a function with single parameter x, where $x \in R : 0 \leq x \leq 2$:

$$f(x) = [x^2, (x-2)^2]$$

3.2.1 Setting desired objectives

First we need to set our objectives to be a vector of two real numbers, that we both want to minimize. Both minimizing function expects index of the inquired objective on its input, and returns whether we want this objective either minimized. As is customary in C family of languages, indexing is zero-based. By default every objective that is not minimizing, will be maximizing.

Listing 3.3: Objectective initialization

```
class Sch1ObjectiveVectorTraits : public
    moeoObjectiveVectorTraits {
public:
     static bool minimizing (int i) {
         return true;
     }
     static unsigned int nObjectives () {
         return 2;
     }
};

// objective vector of real values
typedef moeoRealObjectiveVector <
    Sch1ObjectiveVectorTraits > Sch1ObjectiveVector;
```

As we can see our objective vector is moeoRealObjectiveVector with constraints of Sch1ObjectiveVectorTraits specified by templating.

3.2.2 Definition of individual

We want to define our individual as a vector of real numbers as well. It has to be noted, that individual in ParadisEO holds the objective vector as well, with accompanying constraints, as we can see from inclusion of Sch1ObjectiveVector in template definition. Evaluation of individual is

then done by reading it as a vector of parameters and then modifying its objective vector to reflect the application of evaluation function.

Listing 3.4: Individual initialization

```
// multi-objective evolving object for the Sch1 problem
class Sch1 : public moeoRealVector < Sch1ObjectiveVector
   > {
public:
    Sch1() : moeoRealVector < Sch1ObjectiveVector > () {}
};
```

Second thing to note about individuals definition is that it doesn't set the parameter bounds. Individuals parameter boundaries are set by population object it resides in.

3.2.3 Evaluation of the population

In this step we slightly differ from the official ParadisEO tutorial, where only evaluation function of a single individual is defined. In our case, we evaluate the whole population of offsprings at once. Defining our own population evaluation allows us later down the line to evaluate whole population in paralel with relative ease.

Listing 3.5: Individual initialization

```
template<class EOT>
class Sch1PopEval : public eoPopEvalFunc<Sch1> {
public:
  /** Ctor: set value of embedded eoEvalFunc */
  Sch1PopEval() {}

  /** Do the job: simple loop over the offspring */
  void operator()(eoPop<EOT> & _parents, eoPop<EOT> &
     offspring) {
      (void)_parents;
      for(int i=0;i<offspring.size();i++){
  if (offspring[i].invalidObjectiveVector()) {
      Sch1ObjectiveVector objVec;
            double x = offspring[i][0];
            objVec[0] = x * x;
            objVec[1] = (x - 2.0) * (x - 2.0);
            offspring[i].objectiveVector(objVec);
  }
```

```
    }
};
```

3.2.4 Initializing and running the evolution

In previous sections we have specified the nature of our problem. Now we put it together to create a runable heuristic solver. Because this is just to serve as an example, we hardcode several parameters of our evolutionary algorithm.

Listing 3.6: Individual initialization

```
// set individuals bounds
eoRealVectorBounds bounds (1, 0.0, 2.0);
//setting probability of crossing over two parameters
eoRealUXover <Evolve> xover(0.25);
//setting the scope and probability of mutation of a
    parameter
eoUniformMutation < Evolve > mutation (bounds,0.01,0.25);

eoRealInitBounded < Sch1 > init (bounds);
// set population size to 100
eoPop < Sch1 > pop (100, init);
// set number of generations to 100
eoGenContinue < Sch1 > defaultGenContinuator(100);
//set probability of applying mutation (and crossover)
//to each offspring in a new generation to 0.35 (and
    0.25)
eoSGAGenOp < Sch1 > defaultSGAGenOp(xover, 0.25, mutation
    , 0.5);

PyPopEval< Sch1 > popEval;
```

With all the parameters set, we can initialize the NSGAII algorithm, run it, and print out the results. Changing the algorithm is simple, because the way continuator, populaiton evaluation and offspring generation is setup is shared across all of the algorithms. Only difference would be in supplying some algorithm specific parameter, like the indicator for IBEA, or the secondary archive for SPEA2.

Listing 3.7: Individual initialization

```
moeoNSGAII < Evolve > nsgaII (defaultGenContinuator,
   popEval, defaultSGAGenOp);
nsgaII (pop);
moeoUnboundedArchive < Sch1 > arch;
arch(pop);
arch.sortedPrintOn (cout);
```

3.2.5 Mutation and Crossover

One of the settings we need to elaborate on further are the initialization of mutation and crossover for the evolution algorithms. This choice directly influences intensification and diversification of heuristic. Having defined the individual as a vector of reals limits our choice in mutation and crossover functions. This means we could affor to chose only one representative for each, the `eoUniformMutation` and `eoRealUXovereoRealUXover`. These two functions are passed to `eoSGAGenOp` that is used in the algorithm to generate ofsprings of the old generation. Constructor of `eoSGAGenOp` has four parameters, namely the crossover function, the probability that this function will be applied to a particular pair of individuals in population, the mutation function and again the probability that this function will be applied to a particular individual. When we used `eoSGAGenOp < Sch1 > defaultSGAGenOp(xover, 0.6, mutation, 0.6);`, new generation will have approximately 60% individuals created by crossover application and 60% by applying mutation. In this instance there has to be at leas 20% of offsprings that were created by application of both.

3.2.6 Mutation

Function we are using,`eoUniformMutation` has three parameters in constructor, first the `bounds` object that sets parameter boundaries, second the `\epsilon` parameter that informs mutation of the scope of change it is allowed to do and fourth `p_change`, the probability of changing particular parameter. Abridged implementation is showed in listing **??**.

Listing 3.8: Individual mutation

```
bool operator()(EOT& _eo){
  bool hasChanged=false;
  for (unsigned lieu=0; lieu<_eo.size(); lieu++)
```

```
      if (rng.flip(p_change){
        // check the bounds
        double emin = _eo[lieu]-epsilon*bounds.range(i);
        double emax = _eo[lieu]+epsilon*bounds.range(i);
        emin = std::max(bounds.minimum(lieu), emin);
        emax = std::min(bounds.maximum(lieu), emax);
        _eo[lieu] = emin + (emax-emin)*rng.uniform();
        hasChanged = true;
       }
    return hasChanged;
  }
```

3.2.7 Crossover

Crossover application recombines two individual to create two new individuals. `eoRealUXovereoRealUXover` jas single parameter in constructor and that is the `preference` probability. This crossover function simply exchanges some of the individual's parameters among them based on the value of `preference`. This means that with `preference =0` or `preference=1` both offsping will be the same as parrents, with `preference=0.05` first offspring will have on average half of parameters from first parent and rest of parameters from second parent. The second offspring will have inherited the parameters inversely. Abridged implementation is showed in listing **??**.

Listing 3.9: Individuals crossover

```
bool operator()(EOT& _eo1, EOT& _eo2){
bool changed = false;
for (unsigned int i=0; i<_eo1.size(); i++) {
   if (rng.flip(preference))
     if (_eo1[i] != _eo2[i]){
       double tmp = _eo1[i];
       _eo1[i]=_eo2[i];
       _eo2[i] = tmp;
       changed = true;
     }
 }
return changed;
}
```

3.3 Embedded Python

After some experimentation wit ParadisEO framework in similar fashion we descibed in previous section, we have realized, that reusing the C++ source code is relatively cumbersome. Each change required recompilation, and that necessitated working build environment. We have decided, that a combination of a portable and scripting language would suit this task better.

We chose Python for these reasons:

1. It is an efficient glue language. Most often we want to optimize already written application with minimal changes. Python is well suited for running external applications, specifying their inputs and parsing their outputs.

2. It has good facilities for statistical analysis. That allows us to include analysis of every optimization run into the executable itself. Having automatically generated experimental log has proven invaluable especially in calibration of algorithms for our specific problem.

3. It is easily integrated with C++. We strive to make the integration with ParadisEO simple and extensible.

3.4 Using our framework

Our micro-framework consists of a single executable and a configuration file in Python. Configuration itself consists of defining selected entry points, in particular the number and bounds of input parameters, the number of objectives to optimize, definitions of cross-over and mutation functions and either the definition of evaluation function, or functions for scheduling evaluations in parallel on Boinc. We provide reporting function entry-point as well, to automatically collect and analyze results of the algorithm run.

3.4.1 Example problem in python

We have implemented this by changing most of the entry points for the evolutionary algorithm to be defined in python. These include the def-

inition of the number of objectives, number of parameters, parameter bounds and population evaluation function. To further simplify configuration, we asume, that all the objectives are to be minimized and that both objectives and parameters are vectors of real numbers.

Listing 3.10: Minimal working example

```
# -*- coding: utf-8 -*-
import logging
logging.basicConfig(filename='evolve.log')

#minimizing everything
def nObjectives():
  return 2
def nParams():
  return 1
def minimumBounds(i):
  return 0.0
def maximumBounds(i):
  return 2.0

def popeval(pop):
  return [[x[0]**2,(x[0]-2)**2] for x in pop]
def report(inp,rep):
  logging.info(inp+"\n"+rep)
```

You can recognize most of the entrypoints counterparts by comparing them to example C++ code in previous section. Only entrypoint that has no direct equivalent is the `report` function. This function is called after the algorithm ended and recieves the command-line parameters and the results of the evolution as well. It serves as somewhat more refined `arch.sortedPrintOn (cout);` from the previous example.

3.4.2 Command line arguments

One of the things that was missing in the minimal python example was the choice of algorithm, and the definition of mutation and crossover functions, size of populaiton and the number of generation. These are defined by supplying command line parameters to the binary. Assuming that we have saved the example as a file "objectives.py", and we have it in the same folder as our binary named "evolve", we execute the evolution with this command:

Listing 3.11: Minimal working example

```
./evolve  --pAlgo=nsgaII --mutEpsilon=0.1 --popSize=50 --
    maxGen=200 --pMut=1 --mutProb=0.01 --pCross=0.01 --
    crossProb=0.5
```

TODO

3.5 Entry points

Now we describe each entry-point in more detail.

3.5.1 Input parameters

For inputs it is necessary to specify their bounds, which are then used to initialize the population and to supply constraints for mutation and crossover parameters. This is done by specifiing three functions, nParams, minimumBounds and maximumBounds. The nParams function has no inputs and just returns the number of parameneters for the evaluation function. The minimumBounds is a function that recieves the index of the parameter of the individual.

Listing 3.12: Inputs

```
def nParams():
  return 2 #specifying two parameters

td=[{"min":1    , "max":28}, # bounds of first
    {"min":0.01 , "max":1.0:}]# bounds of second

# specifying entry points
def minimumBounds(i):
  return td[i]["min"]
def maximumBounds(i):
  return td[i]["max"]
```

3.5.2 Objectives

Because we focus on multi-objective evolution, you need to specify number of objectives. To simplify configuration,minimization of all the objectives is hard-coded.

It is advisable to keep the number of objectives limited, because number of solutions on the Pareto optimal dramatically increase with new objectives. At minimum, Pareto front of $(n+1)$ objective problem will contain all the solutions of a Pareto front of n-objective problem. [9]

Listing 3.13: Objectives

```
def nObjectives():
  return 3
```

3.5.3 Evaluation function

To simplify configuration, inputs and outputs of evaluation function are always an n-tuple of floating point numbers. We believe this is general enough, and that most WSN optimization problems can be fitted to this constraint. Only problem might pose translation of combinatorial problems to continuous ones, fortunately so far we have been quite successful with using just a simple rounding techniques. In addition their convergence may provide valuable insight to structure of the problem.

Because the function is specified in Python we can easily run external binaries.

Listing 3.14: Evaluation

```
def popeval(pop):
  return map(lambda x:[x[0]**2,(x[0]-2)**2],pop)
```

3.5.4 Parallel evaluation of population

Because we recieve the whole population at once (as opposed to evaluating every individual separatly), there is nothing to stop us from evaluating the individuals in parallel. Python has a wide array of tools and libraries allowing for paralelization. Probably simplest approach is to import map function from multiprocessing module, that uses threadpool to utilize all of the processor cores to paralelize the process of mapping a function over an iterable.

Listing 3.15: Evaluation

```
import multiprocessing
def popeval(pop):
  pool = multiprocessing.Pool()
  out = pool.map(lambda x:[x[0]**2,(x[0]-2)**2],pop)
  pool.close()
  return out
```

3.5.5 Reporting

Reporting function takes two arguments, both strings. First string is the `char * args` string passed to the executable. Second string is the console printout or the Paradiseo result-archive. Both of these are strings in their nature. While integrating Python with C++ by passing strings is not particularly elegant, we have chosen this particular implementation, because it is easiest to extend and debug. In example below we parse the result of an evolution run and store it automatically in sqlite database.

Listing 3.16: Evaluation

```
def report(inp,rep):
    db = sqlite3.connect("results.db")
    cur = db.cursor()
    cur.execute('CREATE TABLE IF NOT EXISTS reportres (
       result1 REAL,result2 REAL,x REAL)')
    lines2 = rep.split("\n")
    for line in lines2:
        vals = line.split()
        if len(vals) > 1:
            result1 = float(vals[0])
            result2 = float(vals[1])
            x = float(vals[3])
            cur.execute('INSERT OR IGNORE INTO reportres
               (result1,result2,x) VALUES (?,?,?)',(
               result1,result2,x))
```

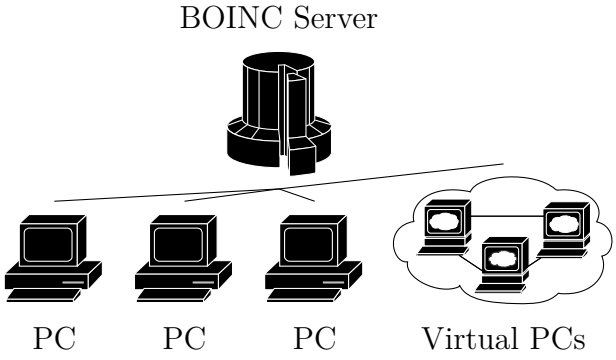
This way, we could integrate the creation of experimental log directly into our executable, computing different . This helps especially with calibration of evolution algorithm parameters. We provide examples of different metrics and indicators that help with evaluation of result in next section.

3.6 Boinc integration

Because we had access to a reasonably sized BOINC cluster we wrote a simple library that allows us to schedule evaluation of individuals on Boinc nodes. BOINC project was originaly concieved as an opensource middleware for distributed computations, where voluneers could allow different organizations to use their idle processor time. Seti@home initiative, that searches for traces of extra terestrial inteligence, is probably the most famous of the projects using BOINC for volunteer computing.

The BOINC cluster we had access to was used to better utilize idle desktops in Masaryk University computer labs. On figure 3.1 we show a high level overview with BOINC server as an orchestrator that has several desktop PCs connected as worker nodes. In case the computation that is scheduled is too resource intensive, administrator is able to add a batch of virtual machines to serve as aditional nodes. This way we had access to around 50 processors to run our experiments on.

Figure 3.1: Lab setup of BOINC cluster



BOINC server has the role of a scheduler while clients are just a simple worker nodes. We submit a bundle with an executable that defines application that we want to run on several workers at once. Scheduling a workunit then consists of specifying configuration files, command-line options and a relative path to a result file. After several work-units were sheduled, server pushes the work-units (and if necessary the executable as well) to idle clients. BOINC architecture allows for several modes of returning results from clients. Client can send partial results back to server periodically, server can verify if result satisfies arbitrary condi-

tions by using custom verificator module, or merge a batch results into one by using custom asimilator module. We have used the default verificator accepting every work-unit, that succesfully produced a result file and default asimilator that doesn't merge results.

This means we could create a simple, single file, python library[2] [6], that schedules evaluation of every individual in generation on BOINC, then polls boinc server untill every related work-unit has been completed and finally collects all the results. Listing 3.18 showcases how this implementation of popeval looks like.

Listing 3.17: Boinc Evaluation

```
def popeval(pop):
  connection = connect()
  for individual in pop:
    schedule(br,individual)
  while not all(completed(br,individual) for individual
     in pop):
    time.sleep(1)
  out = [collect(br,individual) for individual in pop]
  disconnect(connection)
  return out
```

To explain each of the calls, let us presume, that we want to parelilze our SCH1 problem on BOINC. To do this we would need a win32 executable, that accepts x on commandline and saves a file results.txt containg on two separate lines x^2 and $(x-2)^2$. We would upload our executable creating a ne boinc application with hypothetical application identifier 1337. With this presumtption in place we can now define the `schedule`, `completed` and `collect` functions.

3.6.1 Schedule

Scheduling function outsorces the bulk of the work to `create_wu`, that accepts five arguments that are necessary to create a workunit (and the connection object). While we believe parameter names are self-explanatory, we have to note the importance of naming a work-unit correctly. All of the utility functions use work-unit name to certain extent, therefore it is important that relationship between individuals

2. in file boincmechanized.py

parameters and name of the evaluating work-unit are strictly one-to-one.

Listing 3.18: Boinc Evaluation

```
def schedule(connection,individual):
  x = individual[0]
  name = "boinc_sch1_%s_x"%(str(x))
  config_file = ""
  command_line = " %s "%(str(x))
  result_file = "results.txt"
  return create_wu(connection,name,"1337",config_file,
     command_line,result_file)
```

3.6.2 Completed

Testing if the work-unit has been completed is based solely on its name.

Listing 3.19: Boinc Evaluation

```
def completed(connection,individual):
    x = individual[0]
    name = "boinc_sch1_%s_x"%(str(x))
    return wu_completed(br, name)
```

3.6.3 Collect

Similarily to `wu_completed`, `wu_collect` requires work-units name. It returns a temporary file object, that has the contents of the work-units result file.

Listing 3.20: Boinc Evaluation

```
def completed(connection,individual):
    x = individual[0]
    name = "boinc_sch1_%s_x"%(str(x))
    tf = wu_collect(br,name)
    result = tf.readlines()
    return [float(result[0]),float(result[1])]
```

3.6.4 Connection

Because the BOINC server we had access to didin't have any sort of programable interface for remote access aviable, we have hardcoded the credential information into the library file itself. On the other hand, the library itself is compact and the schedule-completed-collect template is easy to appropriate. One could easily imagine a local virtualization cluster or remote cloud provider instead of BOINC.

4 Intrusion Detection System for Wireless Sensor Network

So far we have been discussing our multi-criteria evolutionary optimization from the theoretical and implementation points of view. What was absent were any concrete optimization scenarios, that might utilize our framework[1].

Our ultimate goal is to have a workflow for optimizing intrusion detection systems in wireless sensor networks. This means we want to have a good example network with a simple and configurable IDS to try to optimize with our framework.

4.1 Wireless sensor network settings

Implementation of WSN was reused from [8], with just a slight modifications to decouple it from previous optimization framework. It uses MiXiM simulator based on the OMNeT++ platform. MiXiM network provides complex communication models, including capabilities for precise simulation of interference on physical layer and various energy consumption models. Simulation this precise is often costly enough to warrant the usage of evolutionary heuristics.

We simulate a WSN consisting of sensor nodes equipped with CC2420 transceiver.

The settings of different simulation models are taken verbatim from[7], except for the network topology:

- *Wireless channel model* – An open changing environment is simulated using the *log-normal shadowing* model [4] The pass loss exponent was set up to 2 (outdoor environment).
- *Network topology and routing* – We discuss the network topology and routing in a later section.
- *Data link layer* – Protocol CSMA-CA according to the IEEE 802.15.4 standard is used.

1. Disregarding the SCH1 example.

- *Physical layer* – The radio model represents the CC2420 transceiver that is compliant to the IEEE 802.15.4 standard used by MICAz and TelosB sensor nodes. The sending power is set up to -25 dBm (0.00316227766017 mW) for all sensor nodes.

4.2 Intrusion Detection System

Our nodes implement a simple IDS capable of detecting nodes that seem to be intentionally dropping packets. If we look at the wireless network stack of our node, IDS is part of the medium access control sublayer. There it can eavesdrop on neighboring nodes and check whether they behave accordingly. For each neighbor it updates the number of packets received and the number of packets forwarded.

We wanted to use IDS that is simple, but highly configurable. We have these four parameters to optimize:

1. number of monitored nodes p_n
2. size of buffer for eavesdropped packets p_b
3. threshold for minimal number of received packets for a node to be considered malicious p_m
4. threshold for ratio between forwarded and received packets for a node to be considered to be malicious p_t

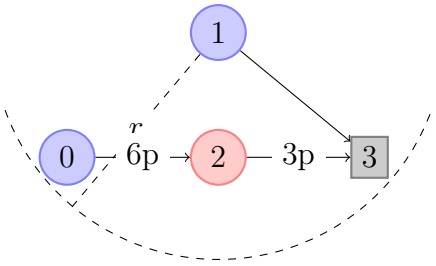
Lets ilustrate this on a small example in figure 4.1 from the point of view of node 1. We presume that node 1 can reliably eavesdrop on anything in range r. Node 2 is malicious and is dropping half of the packets that it is supposed to route from node 0 to base-station 3.

At the end of simulation, based on a preset threshold and ratio between forwarded and relieved packed it decides whether node is malicious or benign. More specifically, the set of malicious neighbors of a node i is:

$$M_i = \{n \in N_i | r_{in} \geq p_m \wedge \frac{f_{in}}{r_{in}} \geq p_t\}$$

And the set of benign nodes is:

Figure 4.1: IDS ilustration



$$B_i = \{n \in N | r_{in} < p_m \vee \frac{f_{in}}{r_{in}} < p_t\}$$

You can see, that both sets are disjoint, and their union is the set of neighbors N_i.

We can see, that based on the values of p_n p_b p_m and p_t, node 1 from our example either asumes all nodes in its neighborhood are benign, or corectly identifies the malicious dropper.

To provide meaningfull objectives for minimization we agregate results across all the nodes. This gives us three objectives:

False negative ratio that calculates for each malicious node the percentage of its neighbors that claimed it is benign and returns the average.

$$fn(x) = \frac{1}{|M|} * \sum_{m_k \in M} \frac{|\{i | m_k \in B_i\}|}{|\{i | m_k \in N_i\}|}$$

False positive ratio that calculates for each benign node the percentage of its neighbors that claimed it is malicious and returns the average.

$$fn(x) = \frac{1}{|B|} * \sum_{b_k \in B} \frac{|\{i | b_k \in M_i\}|}{|\{i | m_i \in N_i\}|}$$

Memory footprint that aggregates the impact of number of monitored nodes and the size of buffer on the memory consumption. We know, that IDS requires 8 bits of space for overy node it is monitoring and 16 bit for every packet stored in buffer.

$$m(x) = 8 * p_n + 16 * p_b$$

For the evolution framework, our simulated network of nodes with IDS is just a black-box function, with four input parameters and a triplet of numbers as an output.

4.3 Network topology

We have loosely based our topology on the wireless sensor network of the Masaryks university laboratory of applied cryptography.

Test application running on each of the nodes periodically sends a packet containing arbitrary information through the network to a base station. We have marked some of the nodes as droppers, so that instead of forwarding all of the packets through the network, they drop certain percentage. In our case, dropping ratio was set to 0.5. Network has a static routing tree, as described in Fig.4.2.

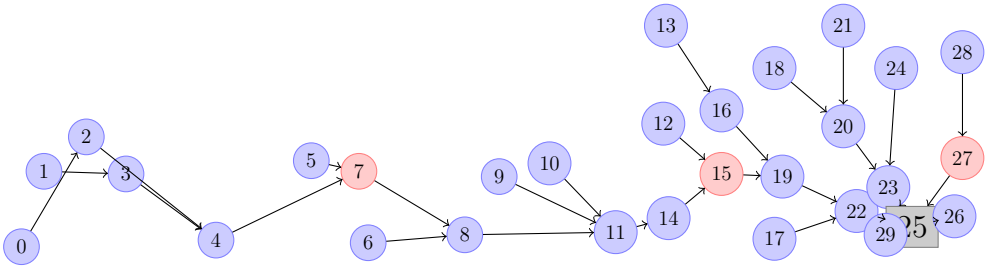
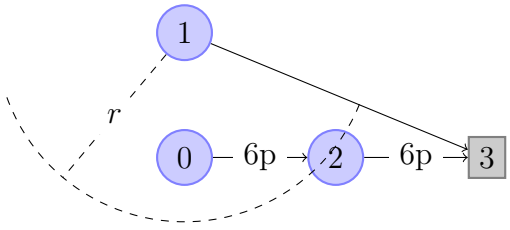


Figure 4.2: Topology of the network used in experiments,with gray base station 25 and red droppers 7, 15 and 27

Figure 4.3: Unreliable IDS



The network layer uses static routing tree created by hand, to allow for inefficient eavesdroping of packets in between of certain nodes. The

situation we are trying to attain is better explained on simpler example in figure 4.3. Because node 2 is at the edge of the reliable range for eavesdroping for node 1, nodee 1 might percieve node 2 as malicious, even if node 2 forwarded all of its packets. This way excessively high number of neighbors that every node monitors p_n would lead not only to high memory usage, but to higher false positives as well.

5 Calibration

We can think of calibration on an evolutionary algorithm as a meta-optimization problem with two objectives, maximizing the fitness of result and minimizing the number of evaluations. With single-objective algorithms, optimizing for these two objectives is easy, fitness is represented by a single value. With multi-objective problems, we usually have two values describing fitness, one for convergence, second for diversity. In all of the previously described algorithms convergence is used as the primary fitness metric, diversity is used to provide ordering among individuals that are not dominating each other.

While measuring number of evaluation is straight-forward, we could chose from several metrics when evaluating diversity of a Pareto approximation or its convergence to Pareto front. IBEA algorithm provides us with two indicators, that can be used to compare Pareto approximations. To simplify our effort we have chosen the hypervolume indicator, because it combines to a certain degree both the convergence and diversity aspects of comparing pareto approximations.

5.1 Hyper-volume indicator

One of the more popular indicators in current research is measuring the hyper-volume of the space dominated by the resulting Pareto approximation. Hyper-volume indicator is based on idea of calculating the volume of objective-space that given Pareto approximation dominates, but is dominated by some reference point R. This works with assumption that the results in objective-space are spread homogeneously, and that we can specify a reasonable reference point. A good reference point is a solution that is dominated by all the other individuals. [1]. Because we usually don't know how will this individual looks like, by guessing this reference point we are expressing certain biases (or expectations) on the shape of the solution space as well.

Hyper-volume gained popularity as a research topic in part because of challenges in its implementation. Early algorithms were based purely on the inclusion-exclusion principle[10]. Several more efficient implementations were proposed since then, our is based on QHV algorithm[5], that uses divide and conquer to calculate hyper-volume more efficiently.

Our algorithm is significantly simplified. For each individual it calculates the hyper-volume it covers and then subtracts parts of the volume that are dominated by rest of the set. Then recursively calculates hypervolume for the rest of the set.

Listing 5.1: Hypervolume indicator implementation

```
def inclHV(p,R):
  return reduce(lambda x, y: x*y,
    [r-i for (i,r) in zip(p,R)])

def exclHV(p,front,R):
  volume = inclHV(p,R)
  dom = dominatedbit(p,front,R)
  return volume - dom

def hv(front,R):
  if front == []:
    return 0
  f = sorted(front,cmp=improves_last)
  return exclHV(f[0],f[1:],R) + hv(f[1:],R)
```

Sorting in *hv* function ensures, that we process individuals in order. Function *dominatedbit* calculates volume of the space enclosed between p and R, that is dominated by the rest of the $front$. First it iterates over *front* and for each individual i calculates the closest intersection between space dominated by i and p. Because we assume minimization, this can be achieved by simply selecting the maximum from each objective. Then it filters out dominated intersection and returns the hyper-volume.

Listing 5.2: Calculation of intersecting hypervolumes

```
def dominatedbit(p0,front,R):
  out = []
  for p1 in front:
    p3 = [max(i0,i1) for (i0,i1) in zip(p0,p1)]
    if p3!=p0 and not p3 in out:
      out.append(p3)
  return hv(nondominated(out),R)
```

Original algorithm[5] uses memoization to further improve the performance. Because we do not use this indicator as a part of evolution

algorithm itself, but only to evaluate fitness of final results, we have deemed further optimization unnecessary.

5.2 Parameters

When not counting algorithm specific parameters, such as sharing parameter δ_{share} of MOGA, there are four parameters that can be used to calibrate evolutionary algorithms: *population size*, *number of generations*, *probability of mutation* and *probability of cross-over*. In addition, implementations of mutation and cross-over operators usually provide additional parameters to specify the amount of change mutated (or crossed-over) individual undertakes.

5.2.1 Population size and Number of generations

These two parameters set the boundaries on minimal and maximal number of evaluations. Lets mark the size of population N and number of generations G Because the whole initial population needs to be evaluated, number of evaluations will at least the size of populations. Because at each new generation, at most N new individuals are generated, therefore in one optimization run, at most $N \times G$ evaluations.

With these mutation operators, the value of parameter d[1] can have significant impact on the speed of convergence, because it specifies the bounds of neighborhood where the individual can mutate. In certain situations we might consider experimenting with the implementation of the mutation operator, especially if we have had some additional domain knowledge that we could use to choose the mutated individual beyond normalized randomization on its parameters.

5.3 Calibration with a search sample

Because the main reason we use multi-objective algorithms is to avoid the need to do the expensive computation of true Pareto front, we could calibrate it on a easier variant of the problem. Underlying assumption is, that the particular variant of the problem will have similar solution space. We will show such an example in chapter **??**.

1. or *sigma* in the case of normal mutation

Computing an exhaustive search for a subset of our problem can help in several ways:

- we can check our assumptions about the shape and homogeneity of solution space
- we can use the Pareto front as a reference point
- for any approximation, we can easily compute how many individuals it dominates and how many individuals dominate is

This is probably the most expensive, but on the other hand the most precise way to estimate, how will an evolutionary algorithm behave in a particular problem domain.

Bibliography

[1] Anne Auger, Johannes Bader, Dimo Brockhoff, and Eckart Zitzler. Theory of the hypervolume indicator: optimal μ-distributions and the choice of the reference point. In *Proceedings of the tenth ACM SIGEVO workshop on Foundations of genetic algorithms*, pages 87–102. ACM, 2009.

[2] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2):182–197, 2002.

[3] Arnaud Liefooghe, Matthieu Basseur, Laetitia Jourdan, and El-Ghazali Talbi. Paradiseo-moeo: A framework for evolutionary multi-objective optimization. In *Evolutionary multi-criterion optimization*, pages 386–400. Springer, 2007.

[4] T. Rappaport. *Wireless Communications: Principles and Practice*. Prentice Hall PTR, 2nd edition, 2001.

[5] Luís Russo and Alexandre P Francisco. Quick hypervolume. *arXiv preprint arXiv:1207.4598*, 2012.

[6] Adam Saleh. Source code for micro-framework based on Paradiseo – webpage. [NOT YET], 2013. `http://doesntyetexist/`.

[7] Martin Stehlik, Adam Saleh, Andriy Stetsko, and Vashek Matyash. Multi-objective optimization of intrusion detection systems for wireless sensor networks, 2013.

[8] A. Stetsko. *On intrusion detection in wireless sensor networks*. PhD thesis, Masaryk University, 2012.

[9] El-Ghazali Talbi. *Metaheuristics: from design to implementation*, volume 74. Wiley, 2009.

[10] Jin Wu and Shapour Azarm. Metrics for quality assessment of a multiobjective design optimization solution set. *Journal of Mechanical Design*, 123:18, 2001.

[11] Eckart Zitzler and Simon Künzli. Indicator-based selection in multiobjective search. In *Parallel Problem Solving from Nature-PPSN VIII*, pages 832–842. Springer, 2004.

[12] Eckart Zitzler, Marco Laumanns, Lothar Thiele, Eckart Zitzler, Eckart Zitzler, Lothar Thiele, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm, 2001.