Masarykova univerzita
Fakulta informatiky

Evolutionary optimization of intrusion detection system in wireless sensor networks

Diploma thesis

Adam Saleh

Brno, spring 2008

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.

For example, setting up intrusion detection system on network of nodes so that it satisfies constrains on accuracy, battery life and memory 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, based on its parameters.

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.

We discuss calibration of the framework in chapter 3. Because evolutionary algorithms are heuristic in nature, we provide several metrics that can be used to guide the settings.

In the chapter 4 we discuss intrusion detection systems for wireless sensor networks.

In chapter 5 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.

And in chapter 6 provide a model example of optimizing such wireless sensor network application, its setup, analysis and calibration of the framework and parallel evaluation on Boinc cluster. We conclude with discussion of the results and a short comparison of used evolutionary algorithms.

2 Evolutionary Optimization

As the title states, we can afford to be quite specific in the description of our optimization problem. Usually more formal definitions of optimization problems deal with inputs and outputs in terms of words over alphabet and appropriate cost function. We thing that defining our optimization problem in terms of n-tuples of real numbers represent configuration and evaluation result of a wireless sensor network .

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 $f : X \rightarrow Z$ is the evaluation function. Without loss of generality it is assumed, that we want to minimize 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 $c : Z \rightarrow \mathbb{R}$, 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 result. 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

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.

Algorithms with strong intensification component are usually more efficient with regards to number of unique evaluations.

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.

In our case, we are constrained by 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 WSN.[7] These are modeled natural process of evolution of species. It is a family of stochastic, population-based heuristics (as opposed to deterministic, single-solution based).

2.2 Single objective evolutionary algorithms

Evolutionary algorithms are based of the fact, that natural process of evolution can be considered an algorithm solving an optimization problem of adapting a species to its environment.

2.2.1 Evolutionary algorithm

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

Step 0: Initialization: generate initial population of individuals and calculate their fitness. Population of individuals is a set of configurations. Their environment is the optimization problem.

Step 1: Variation: apply mutation and crossover parameters 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. In our case, 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.

2.3 Multi-objective evolution algorithms

Problem that we are trying to optimize is unfortunately ill-suited for single objective evolution. Objectives such as memory consumption and IDS accuracy are orthogonal, therefore it is hard to determine single criteria that would satisfactory cover both of them. A 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. We say that solution A dominates solution B, if A has no objective "worse" than B and at least one objective "better". We say that A is Pareto-optimal, if there is no other solution that would dominate A. This means that no objective of A can be improved without deterioration of another objective.

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*

1. we assume minimization

and at least one component of u is strictly smaller:

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

Definition 3. *Pareto optimality A solution $x \in S$ is Pareto optimal if for every $x' \in S$, $F(x')$ does not 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 MOP(F,S), the Pareto optimal set is defined as $\mathcal{P}^* = x \in S | \nexists x' \in S, F(x') \prec F(x)$*

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 a subset that contains only solutions that don't dominate each other a Pareto approximation.

Definition 6. *Pareto approximation For given MOP(F,S), we call set $A \subset S$ a Pareto approximation if $\forall x \in A \nexists x' \in A, F(x') \prec F(x)$*

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 harder than their single criteria counter-parts, mainly 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. One of the more popular metric is the hyper-volume indicator. We will discuss using these metrics for calibration in greater detail in chapter 3.

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

2.4.1 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 $R_i = P - \bigcup_{1 \leq j < i} \mathcal{F}_j$:

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

2.4.2 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.

We can see that 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.

2.5 Spea2

In Strength Pareto Evolutionary Algorithm[12], unlike previous NS-GAII, 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 in with the best fitness $P_t \cup \overline{P_t}$

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 $\overline{P_{t+1}}$

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.

The final result then contains just the non-dominated individuals of the archive.

2.5.1 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 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 sample 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.

2.5.2 Environmental selection

Environmental selection step depends on the number of non-dominated individuals in the union. 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.

2.5.3 Spea2+

Spea2+ a modification of Spea2, that lays additional heuristic on top of it. The main changes are:

1. there are two archives, one maintaining diversity in solution space, the other one maintaining diversity in decision space

2. mating selection for cross-over is done based on individuals neighboring each other in solution space

3. instead of binary tournament to select the mating population, it uses the whole archive of non-dominated solutions

It seems, that in some problem domains Spea2+ outperforms both NSGAII and Spea2, unfortunately, Paradiseo doesn't provide direct implementation in its current version.

2.6 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 $P_t \cup P'_t$

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:

2.6.1 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.

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

2.6.2 Sharing diversity metric

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

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.

2.6.3 MOGA+

The original NSGA algorithm had similar problem with calibration of the δ_{share} parameter. This was alleviated by NSGAII's crowding distance sorting algorithm. Because NSGA and MOGA are so similar, we have decided to add another algorithm, that would use the original convergence metric of MOGA and newer diversity metric of NSGAII. This have helped us in comparing these different algorithms.

2.7 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 sets of the same domain and outputs some quantification of a difference between the two.

Zitzler and Kunzli proposed two indicators:

2.7.1 ϵ-indicator

Additive ϵ-indicator that quantifies the minimal distance first Pareto set has to be moved in each dimension in objective space such that the second Pareto set 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.

2.7.2 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 "contained" between 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 7. *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.

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.

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.

2.8 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,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 hyper-volume 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 3.

3 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 can chose from several metrics when evaluating diversity of a Pareto approximation or its convergence to Pareto front.

3.1 Measuring convergence

IBEA algorithm provides us with two indicators, that can be used to compare Pareto approximations. We provided a third metric, that could be looked at as a different variation on ϵ_+ indicator.

3.1.1 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 hyper-volume for the rest of the set.

Listing 3.1: 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 3.2: 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.

3.1.2 Distance indicator

Distance indicator $I_{\epsilon^+} : Z \times Z \longrightarrow \mathbb{R}$ is another possible choice for indicator that IBEA algorithm uses for comparing two Pareto-approximations. I_{ϵ^+} gives the minimum distance, by which Pareto set approximation needs to be translated in some dimension in objective space, such that the other approximation is weakly dominated. If we know, that all objectives will have minimal value of 0, implementation of this metric may look like this:

Listing 3.3: Distance indicator implementation

```
def distance(o1, o2,maximum):
   return max([(v2-v1)/max for (v1,v2,max)
              in zip(o1,o2,maximum)])
```

3.1.3 Average distance indicator

Given Pareto approximation A and B it calculates the average of distances from all $a \in A$ to the nearest solution $b \in B$. This indicator is not dominance preserving, because euclidean distances will always be positive. There have been achieved good results in measuring convergence when using this indicator to compare Pareto approximation to Pareto front[7].

Listing 3.4: Distance indicator implementation

```
def distance(o1, o2, range):
  return average(
     [min([ndistance(i1,i2,range)
        for i2 in o2])
          for i1 in o1])
```

where *ndistance* calculates normalized distance between two solutions, based on the range of each objective.

3.2 Measuring diversity

Diversity metrics help us avoid Pareto approximations, that have too similar solutions. First metric we used metric based on the the spacing metric taken from Spea2, which we have discussed in greater detail in chapter 2. Spea2 uses the distance of k-th closest neighbor as a part of fitness of its individuals. For the sake of simplicity, we set $k = 1$. Instead of euclidean distance between the two neighbors, it calculates its metric as a sum of normalized differences between every objective.

Listing 3.5: Spacing indicator implementation

```
def distance_aggregate(indi,nb,range):
  sum([(i-n)/r for (i,n,r) in zip(indi,nb,range)])

def spacing(approximation,range)
   dist_indi = []
   for indi in approximation:
  neighbor_dist = [distance_aggregate(indi,nb,range)
    for nb in approximation]
  dist_indi.append(min(neighbor_dist))
   return sum(dist_indi)/len(dist_indi)
```

Similarly we can utilize the crowding distance metric from NSGAII. We calculate the value of each individual with regards to the rest of Pareto approximation and then create the compound metric by averaging the distances.

Listing 3.6: Spacing indicator implementation

```
def (approximation,range)
   split = zip(*approximation)
   aggregate = 0
   for (objective,r) in zip(split,range):
      aggregate +=sum([(prev-next)/r
        for (prev,next) in zip(
         [prev for prev in objective[:-2]],
         [nxt for nxt in objective[2:]])])
   return aggregate/len(split)
```

3.3 Parameters

When not counting algorithm specific parameters, such as sharing parameter δ_{share} of MOGA, or reference point R in I_{HD}-based IBEA, 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.

3.3.1 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.

3.3.2 Mutation and Crossover

Probability of applying mutation, or crossover directly influence how many new individual evaluations are there to be computed. By calibrating this parameter we aim to find the optimal balance between diversification and computational cost of evaluating one generation. More importantly, the implementation of mutation and crossover specify how are these new individuals generated.

Uniform mutation

Uniform mutation changes each parameter p with range $R_p$of the individual I to value uniformly chosen from $(I_p - R_p * d, I_p + R_p * d)$. Each of the parameters is mutated with probability p.

Listing 3.7: Uniform mutation example

```
d = 0.1
p = 0.5
def mutate(x):
    for i in range(len(x)):
      if random.uniform(0.0,1) > p:
          x[i] += random.uniform(-d,d)*(max_bounds[i]-
              min_bounds[i])
    return x
```

Deterministic uniform mutation

This mutation operator defines how many of the parameters be changed if given individual is to be mutated. This makes it slightly more deterministic than the standard uniform mutation.

Listing 3.8: Deterministic uniform mutation example

```
d = 0.1
k = 2
def mutate(x):
    for i in random.sample(range(len(x)),k):
        x[i] += random.uniform(-d,d)*(max_bounds[i]-
            min_bounds[i])
  return x
```

Normal mutation

Parameters are changed based on a normal distribution, with parameter $x[i]$ value set as the mean and $sigma$ setting the standard deviation.

Listing 3.9: Normal mutation example

```
sigma = 0.1

def mutate(x):
  for i in range(len(x)):
    x[i] = random.gaus(x[i],(max_bounds[i]-min_bounds[i])
        *sigma)
```

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

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

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.

3.4 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 6.

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.

3.5 Iterative calibration

To measure convergence of Pareto approximations to the Pareto optimal set, we can either specify a global reference point, with respect to which all approximations are measured, or we specify a comparison function that can provide ordering. Indicator functions from IBEA algorithm are particularly suited for this purpose. Of course, if we have $\mathbb{PF}$ of our problem available, we can use these indicators to compare approximations directly to $\mathbb{PF}$.

4 Intrusion Detection System on Wireless Sensor Network

Because we have chosen Wireless Sensor Networks, and more specifically Intrusion detection Systems as the focus of our optimization framework, before we delve into implementation details of the framework itself, we define our model optimization problem. In chapter chap:framework On this model IDS we base our experimental scenarios in chapter 6 and examples of usage of our framework in chapter 5.

4.1 Wireless sensor network

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 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.

Our model network is loosely based on the setup of physical WSN in our lab. We simulate a WSN consisting of sensor nodes equipped with CC2420 transceiver.

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

- *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* – Static topology was based on the topology of the physical WSN set up in LABAK FI MUNI.
- *Network layer* – The network layer uses static routing tree created by hand, to allow for inefficient routing of packets in between of certain nodes. This way the network contains nodes that are unintentionally dropping packets, providing a challenge for our IDS.

- *Data link layer* – Protocol CSMA-CA according to the IEEE 802.15.4 standard is used.

- *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.

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.4.1. The routing was set up so that individual nodes can eavesdrop their neighbors and that there will be some benign nodes that drop packets unintentionally.

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

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:

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

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

We want to minimize these three objective functions:

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.

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

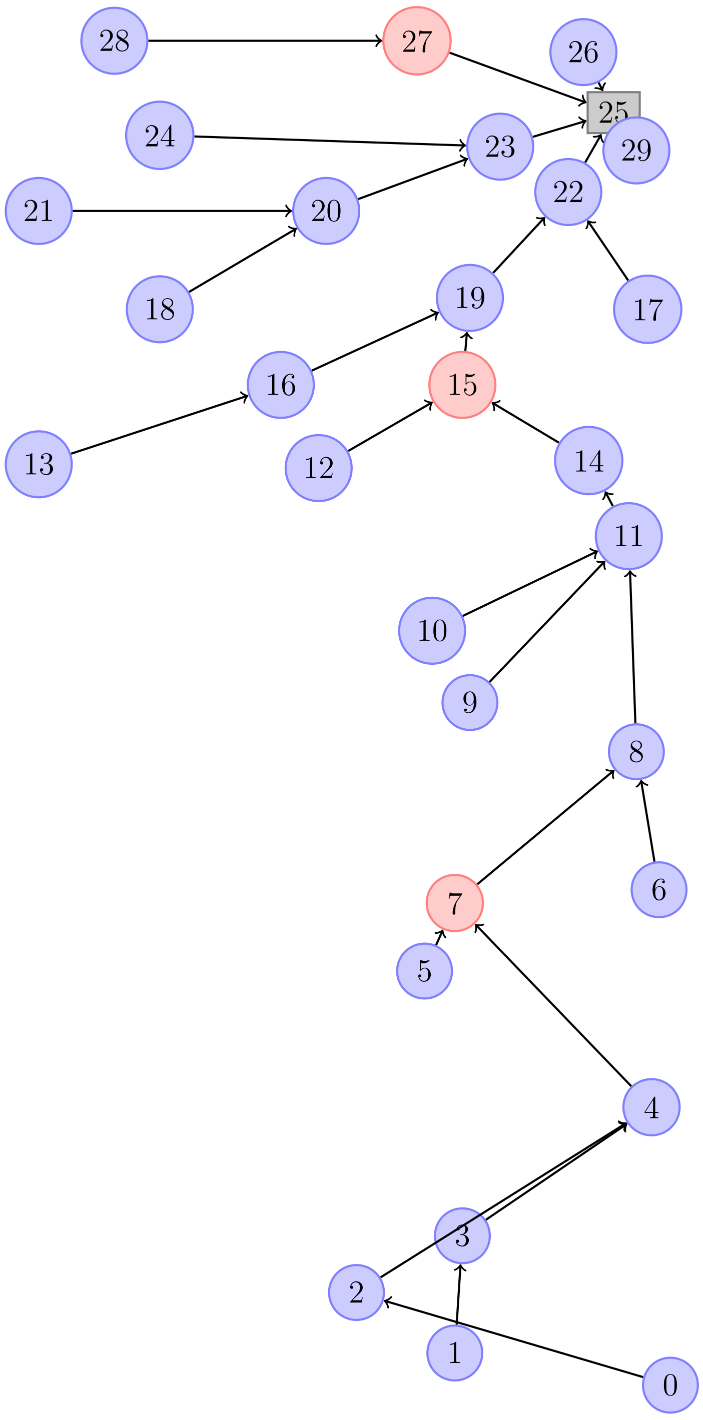

Figure 4.1: Topology of the network used in experiments,with gray base station 25 and red droppers 7, 15 and 27

5 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. Emphasis is on ease of configuration, ability to evaluate population in parallel and 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].

5.1 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 most time consuming process is the selection. This clearly is not the case with wireless sensor networks. 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.

5.1.1 Design of ParadisEO MOEA

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 5.1: 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 various classes that implement. We can showcase how this composing works on an example of a continuator:

Listing 5.2: 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.

5.1.2 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 has its value, and contains 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.

5.1.3 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 these to evaluate the children of the old population and creating the new generation.

5.1.4 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 moeoFitnessThenDiversityComparator
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.

5.1.5 Stopping criteria

Continuator operator provides the basic stopping criteria. The default implementation is based on limiting maximum number of generations[1], but in theory, continuator can use population fitness or another metric 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.

5.2 Embedded Python

Originally, to use some of Paradiseo MOEA algorithms, we would need to implement objects defining the decision space, specifying the number of objectives and evaluation function. We have decided, that scripting language would suit this task better than C++.

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.

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

5.3 Using of 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.

5.4 Entry points

While describing each entry point we try provide an illustrative source code sample, based on one of our early tests of our framework. We have optimized an earlier version of our IDS, that didn't yet include an customizable buffer. We try to find optimal settings of detection threshold and number of monitored neighbors that minimizes the false positives and false negatives ratio. While we do not discuss the results of this experiment, we provide the code as a template in the source tarball of our framework[6].

5.4.1 Input parameters

For inputs you specify their bounds, which are then used to initialize the population and to supply constraints for mutation and crossover parameters.

Listing 5.3: 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"]
```

5.4.2 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 5.4: Objectives

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

5.4.3 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 5.5: Evaluation

```python
def evaluate(monitoredNodes,forwarderThreshold):
   fname = createConfigFile(monitoredNodes,
      forwarderThreshold)
   os.system("./basicIDS -n \".;/opt/MiXiM/src;\" " +
      fname)
   fp = parseFalsePositives("./results/TestLab-0.sca")
   fn = parseFalseNegatives("./results/TestLab-0.sca")
   return (fp,fn)
```

It should be noted, that it is insufficient to specify only one parameter of the evaluation function and then treat it as a vector, all the parameters have to be specified separately.

5.4.4 Mutation and Crossover

This directly influences the intensification of heuristic. There are several popular mutation and crossover functions for individuals based on bounded vector of real numbers. To provide convenience, if mutation function is not specified, uniform mutation will be used and similarly, if no crossover function is provided, pairwise-replacement crossover will be used.

Paradiseo contains several other mutations, that can be used with individuals based on vectors of reals, we have discussed them in greater detail in chapter 3.

Again there is a small difference between our entry points for mutation and cross-over and examples used in chapter 3. Our entry points do not operate on vectors but on specific parameters, similarly to evaluation entry point.

Listing 5.6: Mutation and Crossover

```
def mutation(monitoredNodes,forwarderThreshold):
    return (random.gaus(monitoredNodes,5)),
           random.gaus(forwarderThreshold,0.1))

def crossover(nodes_1,threshold_1,nodes_2,threshold_2):
    return ((nodes_1,threshold_2),
            (nodes_2,threshold_1))
```

5.4.5 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.

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.

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.

5.5 Example problem

We have chosen the first Schaffer's bi-objective problem SCH1, to provide an example of minimal Python file.

The goal of the SCH1 problem is to minimize the following two objectives objectives of a function with single parameter x:

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

Listing 5.7: Minimal working example

```python
# -*- 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 evaluate(x):
  return (x**2,(x-2)**2)
def report(inp,rep):
  logging.info(inp+"\n"+rep)
```

5.5.1 Boinc Assisted Evolution

Because evolution is population-based heuristic, it is well suited for parallel evaluation of its individuals. ParadisEO already has two methods included, either by use of MPI protocol, or by using SMP on multi-core machine[3]. Unfortunately, SMP module of ParadisEO is not yet stable on all platforms, and MPI has specific demands on infrastructure and is usually hard to retro-fit on already written program.

Boinc on the other hand can simply utilize machines already present to form a simple computational grid. With its simple architecture consisting of Management server and several worker nodes it can be deployed in most settings. Its simple architecture prohibits cooperation between worker nodes, but that doesn't concern us while evaluating individuals.

We have decided to give the evolution algorithms ability to use a simple scheduler for creating Boinc work units, plugable from Python. In our micro-framework it consists of three functions, *send*, *completed* and *collect*.

send: this is the function, that framework uses to start evaluating a configuration. In our case it creates new work-unit on Boinc with configuration specified by input parameters.

completed: this is the function, that framework uses to check an evaluation of a particular configuration has finished. In our case, this is done by periodically polling the Boinc management server.

collect: this function has the same signature as *evaluate* function discussed previously. On receiving configuration parameters it tries to retrieve result of the work-unit in question and returns tuple of objectives we are trying to optimize.

Even though the evaluations themselves will be done in parallel, we have avoided parallel code in our framework itself. While running experiments, it is often useful to log various auxiliary data, such as number of evaluations or an average running time of an evaluation. In threaded environment, it would have lead to locking data-structures and that in turn could lead to dead-locks.

As an example we provide optimization problem that we have originally used to test our code. It contains more boiler-plate code than the academic SCH1 problem showed in listing 5.7, but is more representative of actual use case for our framework.

Listing 5.8: Example usage of boinc

```
boinc_api.setSsh("127.0.0.1")
boinc_api.setUser("xuser")
boinc_api.setPassword("password")
boinc_api.setApplication("WSNevo")

cmd = "-individual FILE_0 --cmdenv-config-name TestLab"
result = "results.zip"

def conf(nodes,buffer,treshold,packets):
    config = """
[Config TestLab]
**.node[*].nic.ids.fwdBufferSize=%s
**.node[*].nic.ids.fwdMinPacketsReceived = %s
**.node[*].nic.ids.treshold = %s
**.node[*].nic.ids.maxMonitoredNodes=%s
"""%(str(buffer),str(packets),str(treshold),str(nodes))
    return config

def send(nodes,buffer,treshold,packets):
    config = conf(nodes,buffer,treshold,minpackets)
    return boinc_api.send(config,cmd,result)

def completed(nodes,buffer,treshold,minpackets):
    config = conf(nodes,buffer,treshold,minpackets)
    return boinc_api.send(config,cmd,result)

def collect(nodes,buffer,treshold,packets):
    config = conf(nodes,bufferSize,treshold,packets)
    zip = boinc_api.collect(config,cmd,result)
    if zipfile is None:
        return ()
    td =  tempfile.mkdtemp()
    sourceZip = zipfile.ZipFile(tf.name, 'r')
    sourceZip.extractall(td)
    fp = parseFalsePositives(td+"/TestLab-0.sca")
    fn = parseFalseNegatives(td+"/TestLab-0.sca")
    return (fp,fn)
```

Most of the work is done behind the scenes by our *boinc_api* library.

6 Experiments

To test our framework we have decided to optimize a simple intrusion detection system on a model wireless sensor network based on our laboratory WSN testing setup.

6.1 Setup

We want to use the framework to optimize the four parameters of our IDS as described in 6.1. Because this experiment is aimed primarily on showcasing the functionality of our framework, we have created several optimization scenarios based on the rate of network traffic.

First we have calculated two sample exhaustive searches,with period Δ_t in which each of the nodes sends a packet to be rooted to base station set to 10 seconds and 5 seconds. Because simulation of a network with low network traffic is not as source intensive, it allowed us to gouge the problem space and run calibration on it.

6.2 Calibration

We want to compare the performance of all four algorithms, NSGAII, Spea2, MOGA and IBEA. All the algorithms will share mutation and cross-over functions:

	Name	Description	Range	Step
Parameters	p_n	Max monitored nodes	$\langle 1, 28 \rangle$	1
	p_m	Min received packets	$\langle 1, 100 \rangle$	1
	p_t	Detection threshold	$\langle 0, 1 \rangle$	0.01
	p_b	Buffer size	$\langle 0, 50 \rangle$	1
Scenario	Δ_t	Period for sending packets[1]	$\{0.1, 0.5, 1, 5, 10\}$	

Table 6.1: List of parameters to be optimized for each IDS scenario

6.2.1 Mutation function

We have used the default mutation function. It has two parameters,probability that individual will be mutated $pMut$ and relative δ neighborhood around the old value, that the new value will come from.

6.2.2 Crossover function

We have used the default cross-over function as well, with parameters $pCross$ (probability that two individuals will be crossed over) and $crossProb$ (probability that parameters of the two individuals will be swapped)

For algorithms NSGAII, Spea2 and MOGA, calibration was done on fixed number of generations and population size, with optimizing for values $popSize$, $pMut$, δ, and $pCross$:

Population size to be set to following values: $popSize \in \{50, 75, 100\}$

Mutation will be done with probability of $pCross \in \{0.01, 0.1, 0.25, 0.5\}$, with its mutation parameter $\delta \in \{0.05, 0.1, 0.2\}$

Crossover will have probability set $pCross \in \{0.01, 0.1, 0.25, 0.5\}$, with $crosProb = 0.5$

Number of generations is be set to $NGen = 200$

For IBEA we have chosen the I_{HD} indicator, that we use as a comparison metric for other algorithms as well. Because IBEA with I_{HD} requires calibration of reference point Z, we haven't used crossover operator with IBEA at all ($pCross = 0$). We know that a good reference point is dominated by all solutions, therefore we will use $Z = (f_p, f_n, m), f_p, f_n \in \{0.8, 1.1, 1.5\}, m \in \{1500, 2000, 2200\}$, to see the different behavior, when the reverence can dominate some of the individuals, when it is tightly dominated, and where it is worse by a large margin.

Because we have computed the the exhaustive search for both of the scenarios, we know the Pareto front $\mathcal{PF}$ of each their solutions. We have used these metrics, that we have already discussed in chapter 2:

ϵ_+ **indicator:** computes the smallest distance Pareto-approximation has to be translated to dominate the Pareto front.

HD indicator computes the difference in hyper-volumes between Pareto approximation and Pareto front.

Average distance from $\mathbb{PF}$ computes the average euclidean distances from between every individual of Pareto approximation and its closest neighbor from the Pareto front

Diversification computes the diversification metric of NSGAII algorithm

Spacing computes the spacing metric of Spea2 for the closest neighbor

Number of evaluations will be used to evaluate cost of running the optimization

This gives us enough data not only to decide the best algorithm for the task at hand, but to further discuss relevance of different convergence metrics.

6.3 Evolution with aid of BOINC

After comparison of problem spaces, we have chosen the best settings and algorithm. We then apply it to the three of the more resource intensive scenarios, with $\Delta_t \in \{0.1, 0.5, 1\}$ seconds. With the help of Boinc-assisted evolution we could assess whether each MOEA evolution algorithm can provide reasonable and consistent optimization of IDS on WSN.

6.4 Results

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]. `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.