Computer generated multi-objective robot designs

[1719961] Sebastiaan Saarloos

Abstract

Solving most real-world engineering problems are complex decision making processes, where multiple problems must be addressed. In this paper we are going to discuss some of these challenges and provide an algorithm which we solve the multi-objective problem with.

1 Introduction

One of the challenges faced by a Computer Engineer building robotic platforms is finding the optimal position of sensors (modules) on the robotic platform. When designing robotic platforms, for instance for rescue robots, it could be quite difficult to place modules and sensors in such a way that they do not influence each other. For an engineer developing an algorithm that can design such a robot, while be compliant to the requirements and constraints of the client and maintaining compatibility between sensors described earlier could be quite difficult, but might be solvable with an algorithm that takes all these factors in account.

In an earlier project, a group of students designed a placement algorithm to determine the optimal position of modules on rescue robots. The problem of that was solved was by automatically placing modules according to their mass. The solution of that was proposed back then could place the modules according to the mass of the modules in a discrete space and was based at a single-objective strategy, where multiple objectives were added together into one single fitness value. In this paper we try to improve the proposals.

When trying to create such a robot it is difficult to create good result based on one single utility function biased by the engineer who created the utility function and also prevents the great set of possible good and suitable solutions for the problem. In real life situations many people that are not completely known with the design considerations when creating the algorithm are concerned with choosing the right solution. When choosing a single-objective strategy only one single solution is presented, and might be based on some engineering bias, while a multi-objective strategy presents a lot of acceptable solutions. If designed correctly the decision making process of the algorithm can be completely transparent it can be checked by external people and real humans can make the choice from the set of best solutions provided by the algorithm

In this paper, we propose a new algorithm, based on Local Search, to solve this issue. Local Search (LS) is a proven that is researched a lot for this purpose

and has extensions that makes it useful for the continuous domain and Pareto optimisation.

In section 2 we describe the case we use to develop an algorithm for, we continue in 3 to explain problems with multi-objective optimisation. In section 4 we propose the algorithm and discuss some useful implementation details in pseudo-code. In the 5 section we analyse the results of the algorithm and in section 6 we present the conclusions and directions for future research.

2 Case description

As described in the introduction section we are trying to build a modular rescue robot. The robot contains multiple modules that have multiple functionalities. These modules should be placed in such a manner that the robot as a whole is usable and feasible for the task it should performing. The Modular robot case has a strong resemblance to the box-packing problem as described by Scheithauer (1992). In subsection 2.1 we are going to discuss the concept of modules and its properties and continue in subsection 2.3 to explain the requirements and constraints that are based of these properties.

2.1 Modules

As discussed earlier, a rescue-robot is build by combining multiple modules. These modules have some common properties that define each module. All these properties will be defined in this subsection.

Position The position property gives the placement of a module inside the robot. The position is defined by a three dimensional vector $\vec{p} \in \{ a \in \mathbb{R} \mid 0 \leq a \}^3$. For readability, we denote the x-, y- and z- values of the position to p_x, p_y and p_z.

Size The size property of a module is defined as a 3 dimensional vector, $\{ s \in \mathbb{R} \mid 0 \leq s \}^3$. All dimensions should be higher than 0, but lower than the size. For readability, we denote the $width$-, $height$- and $length$- values of the position to s_{width}, s_{height} and s_{length}.

The size, and position combined can be used to define the space that is occupied by the module $\{ m \in \mathbb{R} \mid \}^3$

ßKijk ff of er een betere definitie is om he maximum te definieren

Mass The mass property of a module is defined as a single scalar, $\{ m \in \mathbb{R} \mid 0 \leq m \}$. The unit of the mass is irrelevant, but the unit must be equal for all modules.

Orientation The orientation of a module is the rotation of the module over its x and z axis in the radian set, defined by $\{ r \in \mathbb{R} \mid 0 \leq r \leq 2\pi \}$ and transforms the coordinates of each position inside the Module.

Functionality The functional requrements of a modules is described by the functionality property of a module. In our case we only focus on two kinds of functionalities, *generic* functionality, with no special requirements and constraints, and *camera*, with a front-facing free line of sight. The functionality properties are defined by the functionality set, which is defined as following;

$\mathbb{F}unctionality = \{ generic, camera \}$

2.2 Solution

A solution can be described as a collection of modules. The module has a fitness determined by it's fitness functions, described in subsection 2.3. The result of each fitness function is described as a value between 0 and 1 ($\{ x \in \mathbb{R} \mid 0 \leq x \leq 1 \}$) and are stored in a vector.

2.3 Requirements and constraints

The placement of a module has multiple constraints. A constraint must be met, otherwise a solution is not valid.

As discussed earlier are modules an entity that performs a given task, but for its functioning might be dependent of the placement of other modules. For example, camera modules can be placed on a robot to observe its surroundings. The viewing angle of a camera should not be blocked by other modules, so other modules should be placed outside of the viewing-angle of the camera, or the camera should be placed where there are no other modules in its viewing angle. To be able to calculate, it is necessary the Requirements and Constraints mathematically, such that the definition is as precise as possible and can be implemented in code more easily.

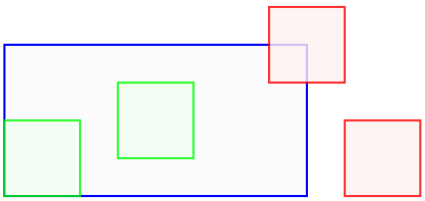

Fig. 1: Two Dimensional Representation of the Space constraint

Space constraints While designing a robot the client may have space constraint. This means that the rescue robot may not exceed a certain size. For the algorithm this is defined by a s_{height}, a s_{width} and a s_length. All modules must fit inside the the space derived from these dimensions. A two dimensional representation of this constraint is given by by figure 1, where the blue box is a representation of the space that might be used for placing modules. Green boxes represent modules that are placed inside the constrained space, and red modules outside the constrained space.

The space is defined by a set from the origin to the width height and length,

and can be defined mathematically as following

$$\begin{aligned}\mathbb{S}olutionSpace = \{\, x \in \mathbb{R} \mid 0 \leq x \leq width \,\} \\ \times \{\, y \in \mathbb{R} \mid 0 \leq y \leq width \,\} \\ \times \{\, z \in \mathbb{R} \mid 0 \leq z \leq length \,\}\end{aligned} \quad (1)$$

The ModuleSpace can be defined by a subset of Space

$$\mathbb{M}oduleSpace = \{\, x \in \mathbb{R} \mid \vec{M}_x \leq x \leq (\vec{M}_x + \vec{M}_w) \,\} \quad (2)$$

The constraint is defined as following;

$$\forall \vec{P} \in \text{solutionSpace}(\vec{S^L}) \wedge \vec{S_i^P} \wedge (0.5\vec{S_i^s}) \in \text{solutionSpace}(\vec{Sol}_L) \quad (3)$$

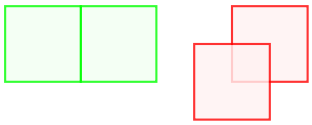

Fig. 2: Two Dimensional Representation of the Collision Constraint

Collission constraint For a solution that is physically possible the modules may not collide with each other, this means that modules may not share any three-dimensional space. This fenomene is visually described by figure 2 The space of a module is defined by equation 4 and the collision is described by equation 5.

$$\begin{aligned}\text{moduleSpace}(\vec{P}, \vec{S}) = \{\, x \in \mathbb{R} \mid \vec{P}_x \leq x \leq \vec{P}_x + 0.5\vec{S}_x \,\} \\ \times \{\, y \in \mathbb{R} \mid 0 \leq y \leq \vec{P}_y + 0.5\vec{S}_y \,\} \\ \times \{\, z \in \mathbb{R} \mid 0 \leq x \leq \vec{S}_z + 0.5\vec{S}_z \,\}\end{aligned} \quad (4)$$

$$\text{collides}(lhsModule, rhsModule) = \text{moduleSpace}(moduleBox) \cap \text{moduleSpace}(rhsModule) \neq \{\ \} \quad (5)$$

Where we consider;

- $\vec{Cam}$ the module that is determined as a camera.
- $\vec{S}$ The solution without the module assigned to $\vec{Cam}$.
- $\vec{Cam}_\alpha$ The view angle of the camera.
- $\vec{Cam}_o$ The orientation of a module compared to the x axis.
- $\vec{Cam}_c$ The centre position of the camera module, defined as $\vec{Cam}_p + 0.5\vec{Cam}_s$

2.4 Requirements

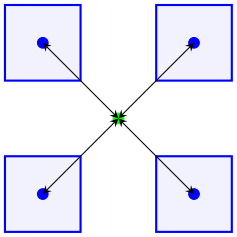

Fig. 3: Two Dimensional Representation of the Intermodular Distance

Inter-modular distance The inter-modular distance is used to pack the modules packed as tight as possible together. This is achieved by calculating the average position of all the modules (described in equation 6), and then calculating the average distance to this average position (described in equation 7). This all is visually represented in figure 3, where the blue boxes are modules. The green dot the average position, and the lines between the distance that will be used for the average..

$$\text{clusterCenter}(\vec{P}, \vec{S}) = \frac{\sum_{i=0}^{n} \vec{P}_i + (0.5\vec{S}_i)}{n} \tag{6}$$

$$\text{averageClusterDistance}(S^{\vec{Position}}), S^{\vec{Size}}) = \frac{\sum_{i=0}^{n} |\text{clusterCenter}(\vec{P}_i + -(0.5\vec{S}_i))|}{n} \tag{7}$$

Stability requirement The robot should should be stable in terms of tilting. That means that if there is a force at applied at the robot in the x or z axis, the robot may not tumble. This is the case when the centre of mass is positioned

$$\text{centerOfMass}(\vec{P}, \vec{S}, \vec{M}) = \frac{\sum_{i=0}^{n} (\overrightarrow{P_i} + \overrightarrow{S_i} \cdot 0.5) \cdot \overrightarrow{M_i}}{\sum_{i=0}^{n} \vec{M}_i} \tag{8}$$

$$\text{centerOfMassDistance}(\vec{P}, \vec{S}, \vec{M}, \vec{P_{ideal}}, \vec{L}) = \frac{|\text{centerOfMass}(\vec{P}, \vec{S}, \vec{M}) - \vec{P_{ideal}}|}{|\vec{L}|} \tag{9}$$

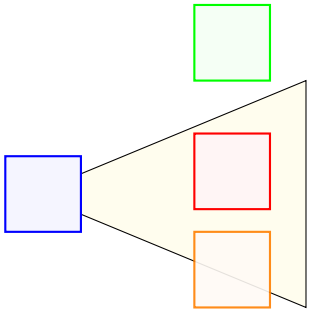

Fig. 4: Two Dimensional Representation of the Free line of sight requirement

Free field sight for Camera Modules For modules that are are defined as camera, a free field of sight is necessary. The free field of sight. The field of sight is the space in front of the camera module where module placement should be avoided. A two dimensional visual representation of the constraint is given in figure 4. The blue box is the camera module, the yellow triangle the field of sight. The Green, Orange and Red boxes are placed modules. The green module is placed outside the field of sight, which is the ideal condition. The Orange module is placed partially inside the field of sight, and is not ideal. The red one is completely inside the field of sight and should be avoided completely.

$$\text{countInSight}(\vec{Cam}, \vec{Sol}) = \sum_{i=0}^{n} \begin{cases} (\text{spherical}(\vec{C_i} - \vec{Cam_c}) - [0, \vec{Cam_o}, 0]) \in \mathbb{C}amera\mathbb{S}ight(\vec{B_\alpha}) & 1 \\ otherwise & 0 \end{cases} \quad (10)$$

where

3 Continuous Multi Objective Optimisation

ßLeg uit wat de eventuele andere probleemrichtingen zijn buiten de scope van het huidige onderzoek.

In the previous section we discussed the engineering problem we try to solve automatically, and the requirements and constraints to design such a robot.

In the previous section we discussed some engineering objectives while designing a rescue-robot. Achieving the best possible solution for each objective is can be achieved by using common existing optimisation algorithms, for example Local Search and Genetic Algorithms. In these scenarios you optimise for one single objective without accounting for any other objectives. The problem with this approach is that you are optimising for one single objective at a time, while not accounting for any other solution, as a result you get very a very good solution for a single objective, while never reaching an optimum for any other objective. Especially when other objectives might be opposing to the objective the algorithm optimised for.

Pareto Domination and Pareto Front To solve the problem with comparing multiple solutions over multiple objectives, we propose to use Pareto Domination to indicate if a solution is more optimal that some other solutions. Instead of single objective optimisation, where one solution is proposed as approximation of an optimal solution, according to Vamplew et al. (2011) and Paquete et al. (2004) Multi-Objective Optimisation using Pareto Domination this creates a set of optimal solutions, where each solution in the set is not dominated by any other solution. The definition whether a solution dominates another solution is given by equation 11. The equation describes Pareto domination. The operator receives two arguments, the fitness values of the left hand side (lhs) solution and the fitness values of the right hand side (rhs). Index-wise all fitness values of the lhs must be greater or equal to the values at the same index in rhs and at least one in lhs must be greater than the value at the same index in rhs.

$$\text{dominates}(\vec{lhs}, \vec{rhs}) = (\exists (fc) \in (\vec{lhs} \times \vec{rhs}) \mid f > c) \wedge (\forall (f, c) \in (\vec{lhs} \times \vec{rhs}) \mid f \geq c) \tag{11}$$

Where;

- $\vec{lhs}$ is the fitness vector of the new solution.
- $\vec{rhs}$ is the fitness vector of the comparing comparing solution.

Feasible solutions As described in Case-Description section there are some constraints to the engineering problem, that must be met. This has consequences to the Pareto Set as described in the previous paragraph. If all Objectives are threaded equal, some non-feasible solutions may dominate feasible solutions. This has the consequence that a feasible optimal solution might not end-up in the approximation of the Pareto front, while the non-feasible solution might end up in the Pareto Front. Removing non-feasible solutions from the Pareto-set might decrease the success of the algorithm, because some non-feasible solutions that dominate feasible solutions might become feasible after further optimisation.

Continuous Optimisation Traditional Optimisation Techniques try to find an optimal solution for a given problem at a given interval, the discrete domain. This means that all solutions are defined at a defined interval, which implies that the size of the set of possible solutions is defined. For the continuous domain the interval between solutions approaches 0, which implies that there are infinite solutions. According to Igel et al. (2007) we can we can modify the step-size, according to the success-rate of the previous iteration and the success rate and step-size of the current iteration to determine the step-size of the offspring iterations. According to the paper we its might be a efficient solution to select dominating solutions to increase the iteration length in neighbourhoods where there is a high probability of success, while in neighbourhoods with a low probability of success there will be less samples.

$$\text{Discrete}(min, max) = \{\, x \in \mathbb{I} \mid min \leq x \leq max \,\} \tag{12}$$

Where "Descrete" is the function that returns a set describing the descrete domain between a given minimum min and maximum max .

$$\text{Continuous}(min, max) = \{\, x \in \mathbb{R} \mid min \leq x \leq max \,\} \tag{13}$$

Where "Continuous" is the function that returns a set describing the continuous domain between a given minimum "min" and maximum "max".

Based on existing literature, an algorithm should be able to solve the case wee described. The greatest issue that still needs to be resolved is the difference between Requirements and Constraints and how they dominate each other.

4 Continuous Pareto Local Search

In this section we are going to propose an algorithm that can solve the Modular Rescue-Robot case as described by section 2. First we are going to propose the

general algorithm and then we are going to the archive clean operator and the step-size operator.

Based on the engineering challenges described in section 3 we propose the Continious Pareto Local-Search (C-PLS) algorithm. The Algorithm described by algorithm 1 solves the problem with Multi-Objective Optimisation in a Continuous domain, while accounting for the difference in importance of the objectives. We propose a version of the Pareto Local-Search that uses the step size operator of $1+\lambda$ CMA-ES and modified the Pareto Dominance operator by to accommodate for the difference of importance of objectives.

4.1 Algorithm description

The algorithm is, as discussed earlier, based on Pareto Local Search. Specific for the Case we applied the algorithm for, it contains one outer loop that runs

the algorithm continuously, and 3 inner loops to iterate over the Cartesian axis.

Algorithm 1: C-PLS algorithm

Input: InitialSolution
Result: ParetoFront based on valid solutions
$initialStepsize \leftarrow 1$;
$\vec{limits} \leftarrow InitialSolution^{l}$
while $|\mathbb{A}rchive \setminus \mathbb{V}isited| > 0$ **do**
  `/* Cartesian product of X, Y, and Z coordinates between 0 and limit with step-size defined by stepSize */`
  $\mathbb{C}oordinates \leftarrow \{ (stepSize \cdot x, stepSize \cdot y, stepSize \cdot z) \in \mathbb{N}^3 \mid 0 \leq (x, y, z) \leq \vec{limits} \}$;
  $currentSolution \leftarrow randomPickSolutionFromArchive(Archive)$;
  $iterations \leftarrow 0$;
  $dominated \leftarrow 0$
  for $module \in solution$ **do**
    for $\vec{coord} \in \mathbb{C}oordinates$ **do**
      $iterations \leftarrow iterations + 1$
      $currentSolution^{position}_{module} \leftarrow \vec{coord}$;
      $valid \leftarrow$ isValid($currentSolution$);
      $fitness \leftarrow$ calculateFitness($currentSolution$);
      if dominatesArchive($currentSolution, \mathbb{A}rchive$) **then**
        $dominated \leftarrow dominated + 1$
        $currentSolution_{stepsize} \leftarrow -1$;
        $\mathbb{A}rchive \leftarrow \mathbb{A}rchive \cup currentSolution$
      end
    end
  end
  for $solution \in \mathbb{A} \setminus \approx \sqsupset \gtrapprox$ **do**
    $newStepsize \leftarrow$ stepsize($currentSolution$, $\frac{dominated}{iterations}$);
    if $solution_{stepsize} = -1$ **then**
      $solution_{stepsize} \leftarrow newStepsize$
    end
  end
  $\mathbb{A}chive \leftarrow$ cleanArchive($\mathbb{A}chive$);
end
return $\mathbb{A}rchive \cap \mathbb{V}alid$;

Where;

- $initialStepsize$ is the interval between the coordinates at the cartasian axis.
- $\vec{limits}$ is the limits that the solution may occupy.

y

4.2 Pareto Domination

To determine whether one solution dominates the other solution, the following implementation of Pareto domination is used.

Algorithm 2: The function domination operator.

Input: $lhsSolution$
Input: $rhsSolution$
Result: Boolean. If true the lhsSolution dominates the rhsSolution. If false, the lhsSolution does not dominate the rhsSolution
function `dominates`($lhsSolution$, $rhsSolution$)**:**
  $lhsFitness \leftarrow lhsSolution_{fitness}$;
  $rhsFitness \leftarrow rhsFitness_{fitness}$;
  $lhsValid \leftarrow lhsSolution \in \mathbb{V}alid$;
  $rhsValid \leftarrow rhsSolution \in \mathbb{V}alid$;
  for $i = 0; i < \text{len}(lhsFitness); i++$ **do**
    if $lhsFitness_{\geq} rhsFitness_i$ **then**
      return false;
    end
  end
  for $i = 0; i < \text{len}(lhsFitness); i++$ **do**
    if $lhsFitness_{>} rhsFitness_i$ **then**
      return true;
    end
  end
  return false;

4.3 Archive cleaning

To improve the memory efficiency of the algorithm, we propose a Archive Cleaning Operator, as described by algorithm 3. The cleaning operator removes solutions that are considered obsolete. The solutions are considered obsolete when the solution is dominated by a valid solution, or when the solution is not valid

and visited.

Algorithm 3: Cleaning the archive by removing all items that are dominated by a valid solution

Input: Archive
Result: CleanedArchive
$shouldRemove \leftarrow \{\ \}$;
for $rhsSolution \in archive$ **do**
  for $lhsSolution \in archive$ **do**
    if $lhsSolution = rhsSolution$ **then**
      continue;
    end
    if dominates($lhsSolution, rhsSolution$)
     $\wedge(lhsSolution_{valid} = rhsSolution_{valid} \vee lhsSolution_{valid})$ **then**
      $shouldRemove \leftarrow rhs \cup shouldRemove$
    end
  end
  return $archive \cap shouldRemove$;
end

The archive-clean operator compares all solutions with any other solution in the Archive. If a solution dominates another solution and the are both valid or only the left hand side is valid, then the right hand side will be removed

4.4 Step-size

To enable the algorithm to provide solutions in the continuous domain, a fixed step-size is not desirable. To solve the problem a dynamic step-size operator is provided. We tried solving this by using a Evolutionary Strategy Step-Size operator proposed by (Rechenberg, 1973) and according to (Kern et al., 2004) provides a good chance to get good results.

Algorithm 4: The $1+\lambda$ CMA-ES step-size operator, based on (Hansen & Ostermeier, 2001) and (Igel et al., 2007)

Input: σ_{prev} /*Stepsize of */
Input: $C_p = 0.5$ /*learning rate*/
Result: New stepsize for all ofsprings
$successrate \leftarrow (1 - C_p)$;
$\sigma \leftarrow \sigma \cdot \exp(\frac{1}{d}\frac{p_{succ}+c_p p_{succ}}{1-p_t argetSuccess})$;
return σ

Where;

- σ is the step-size of the C-PLS algorithm.
- σ_{prev} is the step-size of the parent iteration.

5 Analysis

In this section we are going to analyse a test run of the algorithm. First we explain how we tested the algorithm, after that we analyse the success-rate of

the algorithm, the quality of solutions and the memory usage.

We ran the algorithm described by 4 around 3000 iterations, and after every iteration we saved the archive. We tested the memory efficiency of the algorithm, which means the solutions that are in the discovered optimum in the archive versus the total amount of solutions in the archive. We also tested the combined fitness by measuring the euclidean distance of all finesses, assuming all fitness functions are threaded equal and the average fitness and maximal fitness values of all fitness functions of all solutions inside the archive. The implementation of the algorithm, written in Rust, can be found at our GitHub repository [1].

Fitness performance To see wether how the algorithm performs from a fitness perspective, it is important to review the statistics of all individual objectives. In figure 5 the average output of all objective fitness functions is plotted seperately.

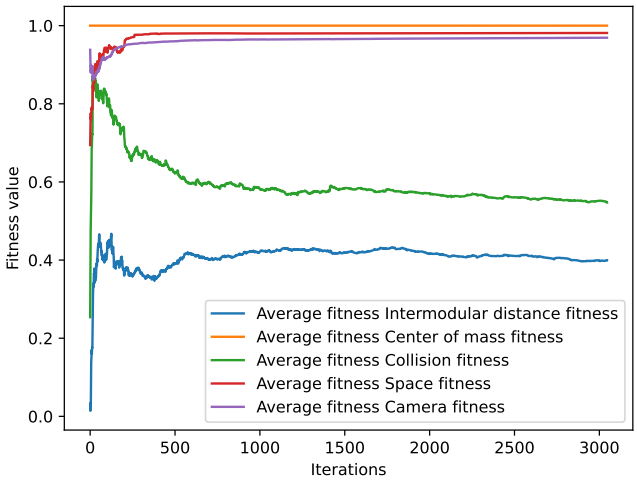

Fig. 5: The average value per iteration of all fitness functions independently

At start we can see a lot of fluctuation in the average of the inter-modular distance fitness and the collision fitness, and a steep improvement of the other objectives. The Center of Mass, Space and Camera objectives are all considered stable after 500 iterations. There is still a slight increase in the Space and Camera objectives.

What is more interesting are the Collision and Inter-modular Distance objectives. These keep, comparing to the other objectives fluctuating. This is interesting because we gave the Collision objective a higher priority to make it part of the Constrained Objectives, and the Inter-modular distance is part of the Requirement Objectives and from a space point of view they are competing objectives.

Combined Objective Fitness To view the performance of the Pareto front we try to combine the Objective Fitnesss through euclidean distance. Considering

[1] https://github.com/sebastiaan1997/researchComputer-generated-multi-objective-robot-designs

that some finesses might be of greater importance than others, this might not be the most ideal way for comparing objectives. In figure 6 we plotted the average euclidean distance, the median euclidean distance and the highest euclidean distance.

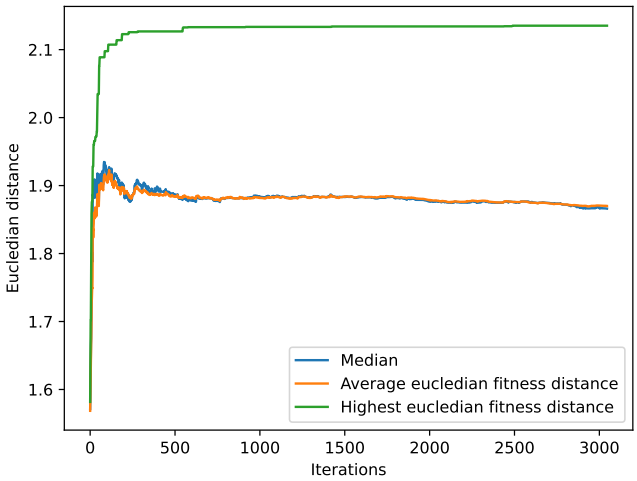

Fig. 6: The median average and highest euclidean distance of all fitness functions per iteration of the C-PLS algorithm.

Here we see until about 600 iterations a lot of change in the values. The highest euclidean distance keeps improving and the average and median euclidean distances are fluctuating a lot. After 600 iterations the trend stagnates. The highest euclidean fitness has a slight improvement over the rest of the test, and the average and median euclidean fitnesses are degrade slightly. This might indicate that after 600 iterations the algorithm reached a local optimum, that approximates the Pareto-Front.

Archive size The archive size represents the amount of solutions that are stored in in memory. We consider the size of the archive a good indication of the memory usage. In figure 7 we plotted the size of the archive per iteration for over 3200 iterations, we also plotted the amount of visited solutions of the archive and the amount of solutions that we consider valid.

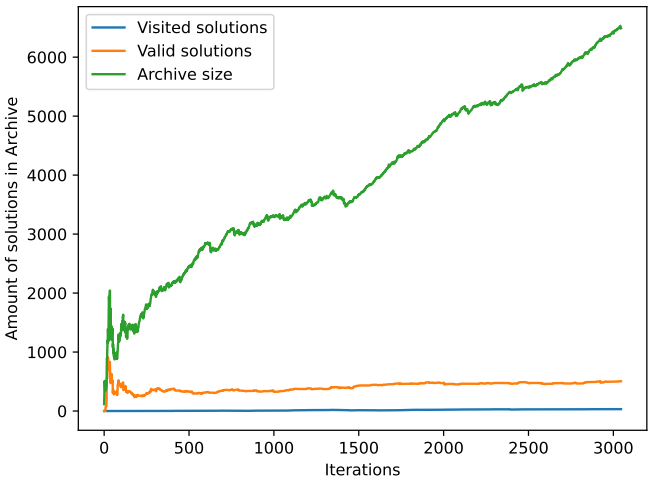

Fig. 7: The size of the archive, compared to the visited solutions and valid solutions inside the archive.

The chart shows a almost linear increase of the amount of solutions inside the archive, while the amount of valid solutions increases fairly slowly. We see regularly a small decline in the size of the archive. In these spots the algorithm found a superior dominating solution, and a lot of dominated solutions are removed from the archive.

Considering the earlier discussed fitness charts of the archive it is interesting that the amount of Solutions in the archive keeps growing, while the amount of valid solutions does not increase much.

It might be interesting to investigate another clean operator to improve memory performance, but the fitness performance of the algorithm can be considered useful for the purpose it is build for.

6 Conclusion

As we showed earlier in the paper we solved a Multi-Objective problem using an Pareto Local Search algorithm in the continuous domain. Based on the analysis given in section 5 the provided algorithm is able to provide valid solutions that satisfy the engineering case we used to test the algorithm.

According to the highest euclidean fitness distance, and the visual representation of that result, the algorithm was able to generate high quality solutions.

The algorithm was able to generate around 200 valid solutions after 600 iterations and was able to double that after 3000 iterations, at cost of high memory consumption according to the Archive size, that grew to above 6000 solutions.

Future work To increase the memory performance of the algorithm it might be interesting to investigate better types of memory cleaning.

We expect that our proposed algorithm could be adapted for multiple cases. For example, we expect that it could be used in gate-assignments at airports, but also designing a warehouse, where items should be stored in an efficient way to provide an optimal.

References

Hansen, N., & Ostermeier, A. (2001). Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, *9*(2), 159–195.

Igel, C., Hansen, N., & Roth, S. (2007). Covariance matrix adaptation for multi-objective optimization. *Evolutionary computation*, *15*(1), 1–28.

Kern, S., Müller, S. D., Hansen, N., Büche, D., Ocenasek, J., & Koumoutsakos, P. (2004). Learning probability distributions in continuous evolutionary algorithms–a comparative review. *Natural Computing*, *3*(1), 77–112.

Paquete, L., Chiarandini, M., & Stützle, T. (2004). Pareto local optimum sets in the biobjective traveling salesman problem: An experimental study. *Metaheuristics for multiobjective optimisation* (pp. 177–199). Springer.

Rechenberg, I. (1973). Evolutionsstrategie—optimierung technischer systeme nach prinzipien der biologischen information.

Scheithauer, G. (1992). Algorithms for the container loading problem. *Operations research proceedings 1991* (pp. 445–452). Springer.

Vamplew, P., Dazeley, R., Berry, A., Issabekov, R., & Dekker, E. (2011). Empirical evaluation methods for multiobjective reinforcement learning algorithms. *Machine learning*, *84*(1), 51–80.