Incremental Computation and Visualisation of Three-Dimensional Pareto Frontiers

Gustavo Martins
University of Coimbra, Portugal

October, 2016

Abstract

Multi-objective optimisation has been extensively studied and used for a long time, having practical applications in almost every field there is, from Economics and Finance to Logistics and Engineering, among many others. As result, many methods, techniques and algorithms have been developed over the years in order to solve the most diverse problems, such as resource management, network and systems design, or ..., to name a few. However, there is a lack of tools that allow a graphical visualisation of the solutions produced by these methods, especially for problems with three or more objectives. In this work, a method to compute the facets of a three-dimensional Pareto frontier, which is based on a well-known algorithm for finding the optimal solutions of a set, is proposed. Additionally, an incremental approach to the frontier computation, allowing for a reasonably fast update of the frontier, is presented. Further applications for this method are also explored.

Keywords: Computational Geometry; Multi-Objective Optimisation; Pareto Frontier; Data Visualisation.

1 Introduction

Optimal solutions Multi-objective optimisation is an area of decision making where two or more objective functions — usually conflicting — are evaluated and optimised simultaneously, and which purpose is to either minimise or maximise each of the objectives. If these happen to be of conflicting nature, usually there is no single solution that optimises all the objective functions at once. Instead, the result of such optimisation is a set of *optimal solutions*, i.e. solutions that cannot be improved on one objective (dimension) without deteriorating at least one of the others. These optimal solutions are also known as *non-dominated solutions*, which, as the name might suggest, are solutions that are not dominated by any other, i.e. there is no other solution that is equally-good or better on all objectives.

Definition 1: Dominance [4] Considering minimisation on all d dimensions, a solution a is said to *dominate* solution b (denoted by $a \succ b$) if

$$\forall i : a_i \leq b_i \wedge \exists i : a_i < b_i, i \in \{1, \ldots, d\}$$

Likewise, a solution c is said to be *dominated* by a set S when

$$\exists s \in S : c \prec s$$

Pareto frontier The set of optimal solutions, also called *Pareto set*, use of the frontier, algorithms to compute the frontier (kung)

Definition 2: Optima [1] Given a set of solutions $S = \{s_1, \ldots, s_m \in \mathbb{R}^d\}$, the *set of optimal solutions* of S is

$$\text{optima } S = \{s \in S : \nexists r \in S, r \succ s\}$$

Definition 3: Optimal solution set [1] A set of solutions S such that optima $S = S$ is called a *optimal/non-dominated solution set*.

Visualisation visualisation can help the user understand how the objectives interact with each other, and make the best decision to which solution to choose. how improving one objective is related to deteriorating the others. talk about the trade-off curve. For a two-dimensional space there are several tools that provide a good visualisation

TODO: Slightly describe the EAF and the AFT as further applications [1, 3]

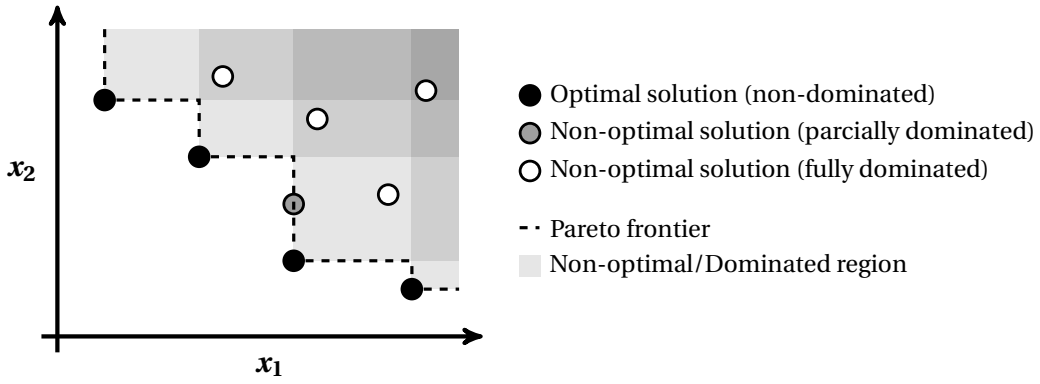

FIGURE 1: Two-dimensional Pareto frontier
Consider minimisation for both objective functions — $f_1(x) = x_1$ and $f_2(x) = x_2$.

Note Throughout the rest of the document, the solutions will be referred to in terms of optimality and dominance, whenever possible. In case it is not possible, e.g. when comparing the values of the solutions, please assume minimisation on all three dimensions.

Document structure This document is organised as follows. Section 2 presents in detail a well-known method to find the optimal solutions on a three-dimensional space. The method serves as a structural frame for the computation of the facets of the Pareto frontier, proposed in Section 3. Section 4 describes an incremental approach to the frontier computation, allowing the frontier to be updated when necessary — both insertion and removal — or even to be constructed one solution at a time, which could be used alongside the optimisation process to display the frontier in real-time.

2 Find the optimal solutions of a set

Algorithm On of the most well-known algorithms to find the set of optimal solutions was first proposed by Kung in 1975 [2], and even though it was presented as an algorithm to find the maxima of a set, it can be applied to find both the maxima and minima by using the definition of dominance. The methodology is to divide the problem into several sub-problems in the lower dimensional space — allowing the algorithm to be applied to any number of dimensions. The algorithm achieves that by pre-sorting the solutions by one of the dimensions. That way, a solution is only optimal if its projection onto the other dimensions is also optimal — given that the previous solutions in the sorting list dominate the current solution in the dimension by which the sorting is performed.

ALGORITHM 1. Optimal solution set [2, 4, 1]

Input: S_i - set of solutions in $\mathbb{R}^d$; d - number of dimensions (objectives)
Output: S_o - set of optimal solutions of S_i

1: **function** OPTIMA(S_i, d)
2: $Q \leftarrow$ SORT(S_i, d) ▷ Queue containing S sorted by coordinate d
3: $S_o \leftarrow \emptyset$
4: $S^* \leftarrow \emptyset$ ▷ Set of optimal solutions, of dimension $d-1$
5: **while** $Q \neq \emptyset$ **do**
6: $s \leftarrow Q$.POP() ▷ Extract element from the queue
7: $s^* \leftarrow$ PROJECT(s, $d-1$) ▷ Projection of s onto the first $d-1$ dimensions
8: **if** $s^* \not\prec S^*$ **then**
9: $S_o \leftarrow S_o \cup \{s\}$
10: $S^* \leftarrow$ OPTIMA($S^* \cup \{s^*\}$, $d-1$)
11: **return** S_o

When d reaches the value 1, there is no need to perform lines 3–10. As replacement, the following suffices: $S_o \leftarrow \{Q.\text{POP}()\}$.

Three-dimensional implementation The first step of the algorithm consists of sorting the solutions by the following order: x_1, x_2, x_3 — any sorting algorithm with logarithmic time complexity is a valid choice. Given the order, the solutions projection contains the second and third dimensions — x_2 and x_3. The more complex operations are the optimality verification and the

update of the lower dimensional space — lines 8 and 10 of the Algorithm 1. This can be achieved by using a balanced binary tree, called *projection tree*, that stores its elements by the following order: x_2, x_3. That way, verifying if a solution is optimal or not consists of inserting the solution in the projection tree and comparing it with its predecessor in the tree. Since the predecessor was inserted first in the projection tree and the fact that it is the predecessor implies that it has an equal or lower value of x_1 and x_2, respectively, than the current solution. Thus, the current solution is optimal only if it has a lower x_3 value than its predecessor. In the case that the solution is optimal, the remaining operation is to update the projection tree, which can be done by iterating through the successors in the tree, removing them in the process, until one that has a lower x_3 value is reached.

Time complexity Given the initial sorting of the solutions by a particular coordinates, the algorithm starts with the complexity $\mathcal{O}(n \log n)$. Also, each point is inserted in the projection tree even before verifying its optimality, which adds another $\mathcal{O}(n \log n)$ to the overall complexity. Verifying a solution's optimality implies accessing the predecessor in the projection tree, adding $\mathcal{O}(n)$ — given that the binary tree is threaded; otherwise, $\mathcal{O}(n \log n)$. Finally, if the solution is not optimal, it must be removed from the tree. Otherwise, the now dominated projected solutions need to be removed from the tree — these are the successors, accessible in $\mathcal{O}(1)$ time each. However, it is known that a solution is inserted only once in the tree, and therefore, removed at most once, either in the first or in the second scenario — it can stay in the projection tree until the end, and not be removed. Therefore, the complexity increases another $\mathcal{O}(n \log n)$, totalling a complexity of $\mathcal{O}(3 \times n \log n + n) = \underline{\mathcal{O}(n \log n)}$.

Space complexity Beyond the set that stores the solutions, the only auxiliary storage necessary is the projection tree, containing at most the total number of solutions in the set — in the case they are all optimal. Therefore, the space complexity of the algorithm is $\underline{\mathcal{O}(n)}$.

3 Compute the Pareto frontier

Representation visible adjacencies, i.e. adjacencies that are not between triangles with the same orientation

3.1 Special case: Equal coordinates

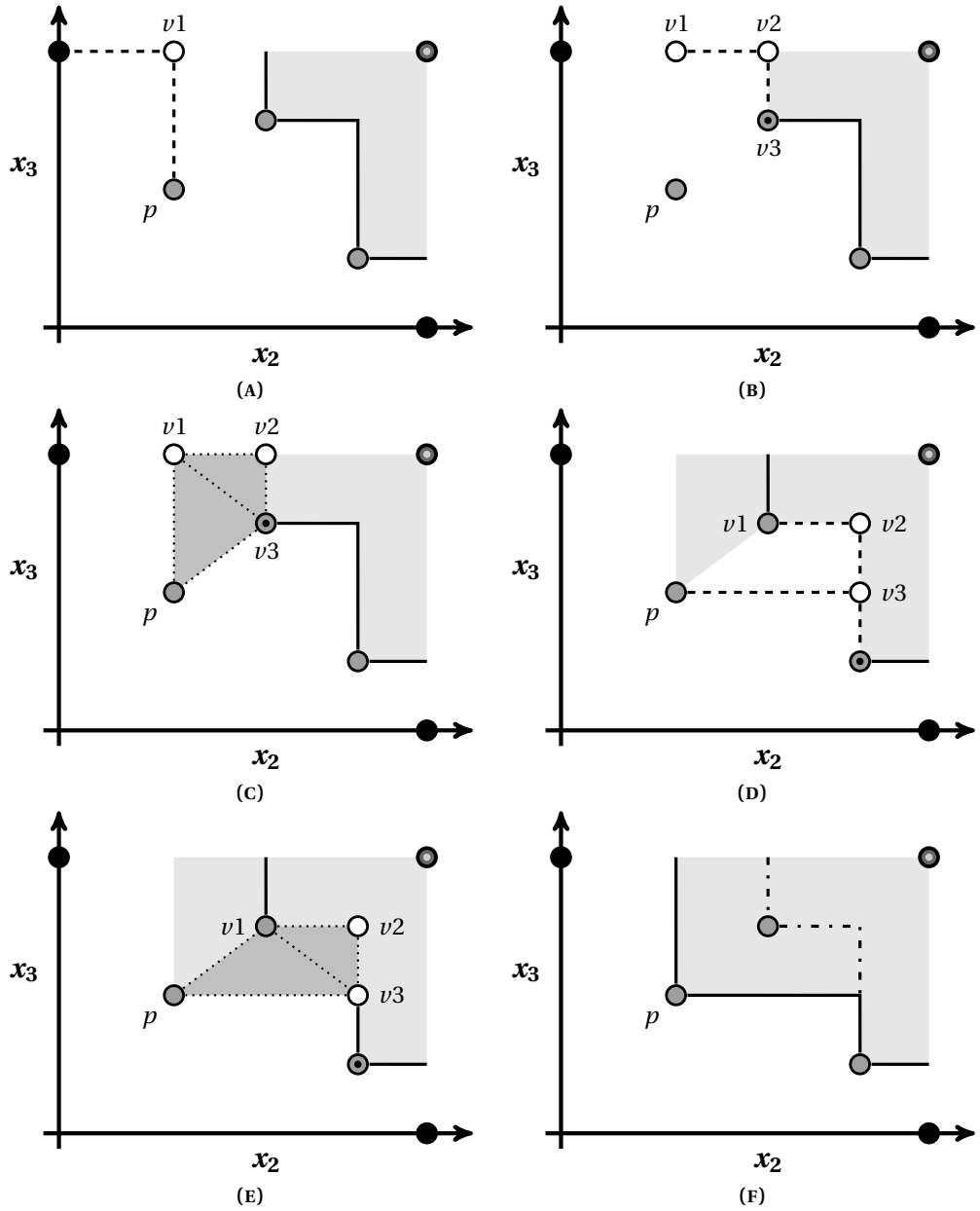

FIGURE 2: Two-dimensional Pareto frontier
Consider minimisation for both objective functions — $f_1(x) = x_1$ and $f_2(x) = x_2$.

save

Complexity

4 Incremental approach

save_break

5 Conclusions and Further developments

TODO: Practical application to the AFT [1, 3]

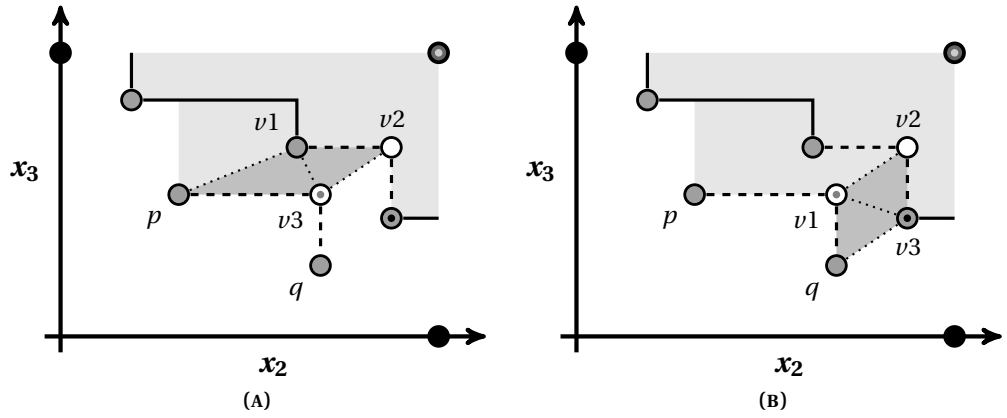

FIGURE 3: Two-dimensional Pareto frontier
Consider minimisation for both objective functions — $f_1(x) = x_1$ and $f_2(x) = x_2$.

References

[1] C. M. Fonseca, A. P. Guerreiro, M. López-Ibáñez, and L. Paquete. On the computation of the empirical attainment function. In *International Conference on Evolutionary Multi-Criterion Optimization*, pages 106–120. Springer, 2011.

[2] H.-T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM (JACM)*, 22(4):469–476, 1975.

[3] M. López-Ibánez, L. Paquete, and T. Stützle. Exploratory analysis of stochastic local search algorithms in biobjective optimization. In *Experimental methods for the analysis of optimization algorithms*, pages 209–222. Springer, 2010.

[4] F. P. Preparata and M. Shamos. *Computational geometry: An Introduction*. Springer Science & Business Media, 1985.

[5] M. Teschner, B. Heidelberger, M. Müller, D. Pomerantes, and M. H. Gross. Optimized spatial hashing for collision detection of deformable objects. In *VMV*, volume 3, pages 47–54, 2003.

ALGORITHM 2. Facet computation

Global space: P - projection tree; V - vertices; T - triangles
Input: p - point to be evaluated; q - next point to be evaluated; S - saved vertices
Output: none

1: **function** FACET(p, q, S)
2: P.INSERT(p) ▷ Insert p in the projection tree
3: $c \leftarrow P$.PRED(p) ▷ Get p's predecessor in the projection tree
4: **if** $c_3 \leq p_3$ **then** ▷ If p is not optimal
5: P.REMOVE(p) ▷ Remove p from the tree
6: **return** ▷ Exit function
7: **if** $S \neq \emptyset$ **then** ▷ If vertices were saved from the previous facet (special case)
8: $v1 \leftarrow S_1$ ▷ Recover vertex $v1$ (was saved as $v3$)
9: **else** ▷ General case
10: **if** P.SUCC(p)$_2 = p_2$ **then** ▷ If p's successor is partially dominated by p
11: $c \leftarrow P$.SUCC(p) ▷ Edge intersects the successor and not the predecessor
12: P.REMOVE(c) ▷ Remove p's successor from the tree
13: $v1 \leftarrow V$.CREATE(p_1, p_2, c_3) ▷ Define vertex $v1$
14: **loop** ▷ Iterate through p's successors
15: $c \leftarrow P$.SUCC(p) ▷ Get p's successor in the projection tree
16: **if** $S \neq \emptyset$ **then** ▷ If vertices were saved from the previous facet (special case)
17: $v2 \leftarrow S_2$ ▷ Recover vertex $v2$ (was $v2$ before)
18: $S \leftarrow \emptyset$ ▷ Clear saved vertices from special case
19: **else** ▷ General case
20: $v2 \leftarrow V$.CREATE(p_1, c_2, $v1_3$) ▷ Define vertex $v2$
21: **if** $c_3 \leq p_3$ **then** ▷ If p's successor is not fully dominated by p
22: **break** ▷ Break cycle (edge intersection)
23: $v3 \leftarrow V$.CREATE(p_1, c_2, c_3) ▷ Define vertex $v3$
24: T.CREATE($v1$, p, $v3$) ▷ Create triangle [$v1$-p-$v3$]
25: T.CREATE($v1$, $v2$, $v3$) ▷ Create triangle [$v1$-$v2$-$v3$]
26: P.REMOVE(c) ▷ Remove p's successor from the tree (it is fully dominated)
27: $v1 \leftarrow v3$ ▷ Vertex $v3$ will be the new vertex $v1$
28: **if** $q \neq \emptyset$ **and** $q_1 = p_1$ **and** $q_2 < c_2$ **and** $q_3 < p_3$ **then** ▷ Special case (p and q intersect in non-dominated space)
29: $v3 \leftarrow V$.CREATE(p_1, q_2, p_3) ▷ Define vertex $v3$ (intersection of p and q)
30: $S \leftarrow \{v3, v2\}$ ▷ Save vertices $v3$ and $v2$ (will be $v1$ and $v2$, respectively)
31: **else** ▷ General case
32: $v3 \leftarrow V$.CREATE(p_1, c_2, p_3) ▷ Define vertex $v3$
33: **if** $p_3 = c_3$ **then** ▷ If p's successor is partially dominated by p
34: P.REMOVE(c) ▷ Remove p's successor from the tree
35: T.CREATE($v1$, p, $v3$) ▷ Create triangle [$v1$-p-$v3$]
36: T.CREATE($v1$, $v2$, $v3$) ▷ Create triangle [$v1$-$v2$-$v3$]

The projection tree sorts its elements by the following order: x_2, x_3, x_1.
Function P.INSERT(a) inserts point a into the projection tree (P) and function P.REMOVE(a) removes point a from the tree.
Function P.PRED(a) returns the predecessor of a in the tree, and function P.SUCC(a) returns the successor of a in the tree.
Function V.CREATE(a, b, c) creates a vertex with the coordinate values of a, b and c. In the case there is an existing vertex with the same values, the function returns its reference instead. One can efficiently achieve this by using a hash table. The hash function is defined as HASH(a, b, c) = $a.p_1$ **xor** $b.p_2$ **xor** $c.p_3$, where p_1, p_2 and p_3 are large prime numbers: 73856093, 19349663, 83492791, respectively [5].
Function T.CREATE(a, b, c) creates a triangle formed by the vertices a, b and c.