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 solution, 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* or *Pareto frontier*, is the result of a typical multi-objective optimisation. Finding the set of optimal solutions of a problem can be extremely useful to the decision maker, since it restricts the possible choices to a small set, allowing for a more efficient look at the existent trade-offs.

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 The graphic visualisation of the Pareto frontier can play an important role in the process of decision making. One of the things that facilitates is the understanding of how the different objectives interact with each other, i.e. the effect that the improvement on one of the objectives has on the others. For a two-dimensional problem, the visualisation of a Pareto frontier, which is often called *trade-off curve*, is somewhat trivial. However, for a three-dimensional problem, the computation of the Pareto frontier requires a more complex method. In this work, one possible method to its computation is proposed.

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

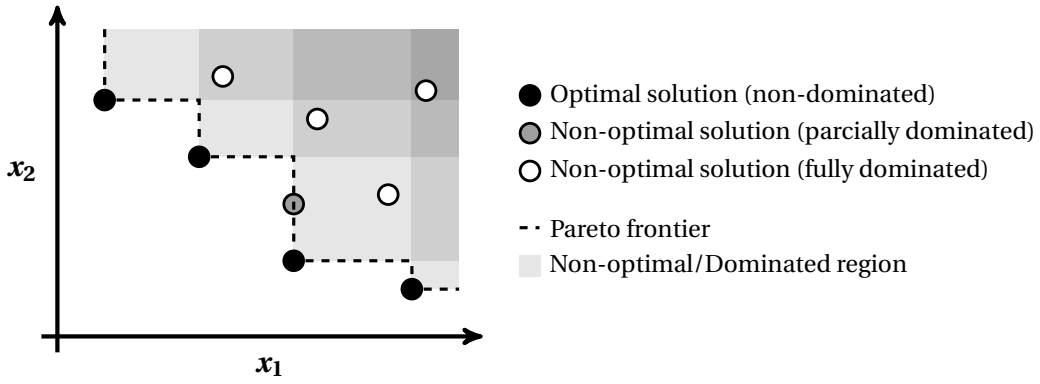

FIGURE 1: Two-dimensional Pareto frontier
Consider minimisation on both objective functions — $f_1(x) = x_1$ and $f_2(x) = x_2$.

Note *Throughout the rest of the document, the solutions will be referred to in terms of optimality and dominance whenever possible. In the 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 of use alongside the optimisation process to display the frontier in real-time.

2 Find the optimal solutions of a set

Algorithm One 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 optimal only 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 Algorithm 1. This can be achieved by using a balanced binary tree, called *projection tree*, that stores its elements in 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 dimension, the algorithm starts with $\mathcal{O}(n \log n)$ time complexity. 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 must 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 A three-dimensional Pareto frontier can be defined as three groups of *facets*, each group with its own orientation, perpendicular to one of the three axes. Each of these facets is generated by (in most of the cases) one solution, hereafter designated as a *point*. The intersections between the points are designated as *vertices*. A facet can be divided into several *triangles*, formed by points and vertices. The reasons for representing each of the facets using triangles are the following: any non-curved shape can be represented as a set of one or more triangles; also, being the triangle the most basic shape that can have internal area, any 3D graphics visualisation tool should provide with a way to display triangles.

Methodology The algorithm for computing the three-dimensional Pareto frontier uses the method for finding the optima as an underlying structure. However, using the method only once would produce only one of the three groups of facets. Therefore, three *dimensional sweeps* of the points are required to represent the whole frontier, with each sweep using a different sorting order for the points: $\{x_1, x_2, x_3\}$, $\{x_2, x_3, x_1\}$ and $\{x_3, x_1, x_2\}$ — with the respective projection trees as $\{x_2, x_3, x_1\}$, $\{x_3, x_1, x_2\}$ and $\{x_1, x_2, x_3\}$.

Reference point and sentinels To define the limits of the frontier the concept of a *reference point* was employed. The reference point is defined by the maximum values of the set of points on the three dimensions. Additionally, a small margin is added to the values both for visualisation purposes and to circumvent eventual complications with computer's numerical precision. Using the reference point, one can create two *sentinel points*, designed to be at the extremes of the projection tree, ensuring that a point always has a predecessor and a successor, thus easing the implementation of the algorithm. The sentinels assume the values of the reference point in one of the two dimensions and negative infinity on the other, so there is no point beyond it, e.g. $(r_2, -\infty)$ and $(-\infty, r_3)$, where r is the reference point. The third dimension is irrelevant, since the sentinels are used only in the projection tree.

Algorithm *Consider the first dimensional sweep for the following description.* If a point p is non-optimal then no facets are to be calculated, and the processing function can be terminated. If the point is optimal, the first step of the algorithm is to find vertex $v1$, which is the intersection of p and its predecessor in the projection tree, or its successor in the case that this has the same x_2 value as p. Next, the loop that iterates through the successors of p in the projection tree is performed, so to find vertices $v2$ and $v3$. Vertex $v3$ is either the successor of p or, in the case that the successor has an equal or lower x_3 value, their intersection — being the latter the point at which the loop is terminated. Vertex $v2$ is the intersection of vertices $v1$ and $v3$. Between cycles, vertex $v3$ becomes vertex $v1$. The triangles that are calculated during each cycle are formed by the points/vertices $[v1, p, v3]$ and $[v1, v2, v3]$.

Special case: facet sharing During the computation of the facets of the frontier, there is one case that requires special attention. This occurs when two optimal points share the same value in one of the dimensions and the edges that originate from them intersect each other in a non-dominated region. If this happens, both points are responsible for the generation of

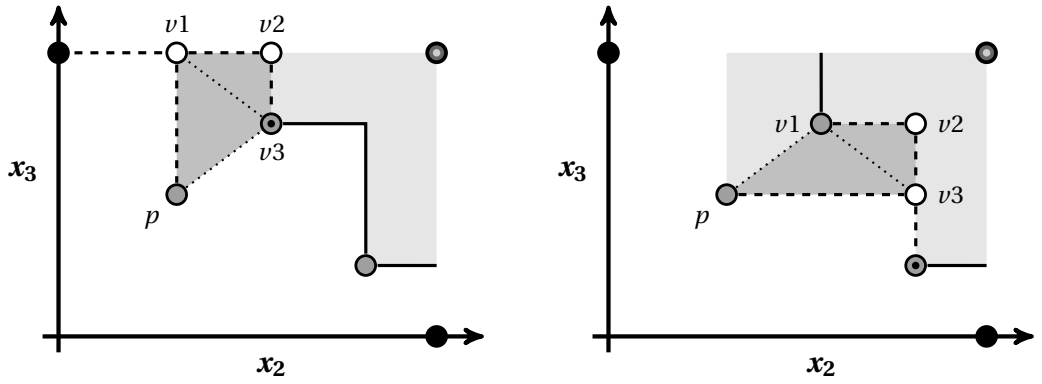

FIGURE 2: Facet computation
Triangles are formed by the points/vertices $[v1, p, v3]$ and $[v1, v2, v3]$.
Vertex $v3$ becomes $v1$ for the next cycle.

different sections of the same facet, hence the name *facet sharing*. Upon detection of the existence of this special case, the solution is to create the vertex $v3$ at the intersection of the two points and save vertices $v2$ and $v3$ from the processing of one point to be used as vertices $v2$ and $v1$, respectively, by the processing of the next point. The special handling of this case is not strictly necessary for the correct representation of the facet, however, it is adopted for two reasons: doing so avoids the creation of one additional vertex per special case — in the location where the vertex $v3$ would normally be; also, in the case that one wants to find and highlight the edges of the frontier, i.e. adjacencies between vertices and/or points that are not between triangles with the same orientation, the handling of this special case would be necessary nonetheless.

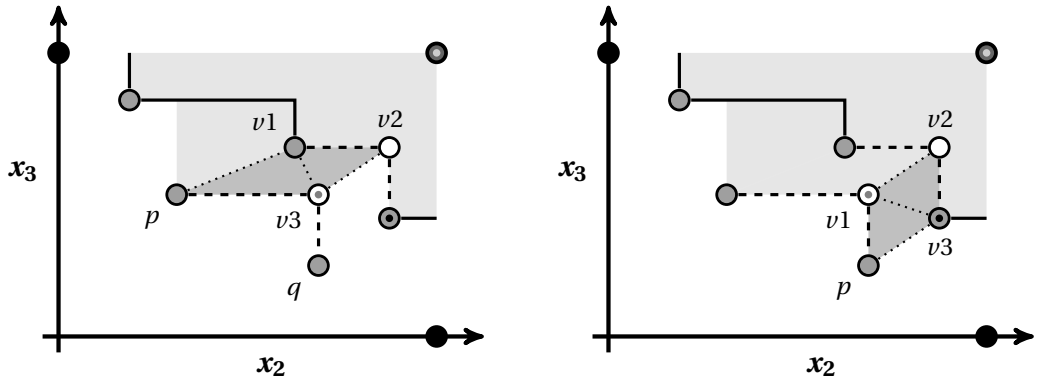

FIGURE 3: Facet computation special case: facet sharing
Vertices $v2$ and $v3$ are saved from one point to be used as $v2$ and $v1$ by the next point.
Triangles are still formed by the points/vertices $[v1, p, v3]$ and $[v1, v2, v3]$.

Complexity In addition to the method for finding the optima, which defines the structure for the computation of the frontier, the other operations that are performed in this algorithm are the creation of vertices and triangles, whose number is in relation to the number of optimal solutions. Since both these operations perform in $\mathcal{O}(1)$ time, the overall time complexity of the algorithm is maintained, $\underline{\mathcal{O}(n \log n)}$. Space-wise, the projection tree is still the only auxiliary storage necessary — if the sorting function can be modified in accordance to the performing dimensional sweep it requires $\mathcal{O}(n)$ space; otherwise, $\mathcal{O}(3n)$. Therefore, the space complexity is also maintained, $\underline{\mathcal{O}(n)}$.

4 Incremental approach

Methodology The objective of an incremental approach to the frontier computation is to be able to perform modifications — adding or removing points — without having to recalculate every facet. The implementation consists of knowing whether to (re)compute the facets or to just update the projection tree — much like the optima algorithm — for each point of the set for each dimensional sweep. Also, whenever the points that have been scheduled for addition and removal are finished processing, the algorithm can be terminated, since the rest of the facets will remain intact. Another optimisation to be considered is whenever the points scheduled for removal are at the end of the sorting list for a particular dimension. If this is

the case, it would be enough to remove the facets generated by these points and translate the vertices from the old limits of the frontier to the new limits. Likewise, whenever the frontier's span increases — addition of points beyond the limits —, the vertices at the limits can be translated to the new limits, possibly avoiding some facet recalculation.

Points scheduled for addition and removal Between frontier computations, any number of points can be scheduled for addition or removal. The incremental approach recalculates facets only when these points influence the computation, i.e. when they are present in the projection tree. When iterating through the points' predecessors and successors in the projection tree, the points scheduled for addition are treated as regular points, while the ones scheduled for removal are ignored. The latter are still verified and removed from the projection tree when dominated.

Advantages and disadvantages The main *advantage* of an incremental approach is that not every facet needs to be recalculated, reducing not only processing time but also memory management. Still, the projection tree needs to be updated. In the worst case scenario, one has to process every single point, inserting them in the projection tree, only to compute the facets generated by a single point at the end. However, it is still a better choice than to recalculate the whole frontier from scratch. The main *disadvantage* is memory consumption. In order to efficiently add and remove points from the frontier, it is advisable to store the set of points in a order-based structure, like a binary tree, so to avoid sorting the entire set whenever the frontier is to be updated. Note that three structures are necessary, one for each dimension. Furthermore, if one desires to perform the optimisations described at the end of the paragraph "Methodology", the same needs to be done for the vertices, so that those at the limits are easily accessible.

Algorithm conditions:
The dimensional sweep can be terminated when none of the following conditions proves true:

- Projection tree contains points scheduled for addition and/or removal
- Points scheduled to be added and/or removed are still to be processed
- Facet sharing (vertices have been saved)
- Facet sharing break

When deciding whether to (re)compute the facet (1) or to just insert the point in the projection tree (2), the following verifications must be performed (in this order):

- Point is scheduled for removal → (2)
- Point is scheduled for addition, or ...
  Projection tree contains points scheduled for addition and/or removal, or ...
  Point has the same sweep dimension value as the next point scheduled to be added or removed, or ...
  Facet sharing (vertices have been saved), or ...
  Facet sharing break → (1)
- Otherwise → (2)

Special case: facet sharing break Due to the incremental nature of the method, a new special case must be considered — mentioned above as *facet sharing break*. This occurs when the projection of a new point c dominates or is equal to the intersection of two already existing points, p and q, that share a facet. When this happens, the facets of p and q need to be recalculated. However, since they no longer share the same facet and c is partially dominated by p, and therefore removed from the projection tree, there would be no reason to recompute the facets of q. To solve this problem, a verification at the end of the facet computation routine is necessary, activating a flag variable to force the recalculation of the facets of q. The verification — placed between lines 33 and 34 of Algorithm 2 — is as follows:

$$c.\text{STATE}() = 1 \textbf{ and } q_1 = p_1 \textbf{ and } p.\text{STATE}() = 0 \textbf{ and } q.\text{STATE}() = 0$$

where STATE() returns 1 if the respective point is scheduled for addition, -1 if it is scheduled for removal, and 0 otherwise.

Complexity Neither the time nor the space complexities of the algorithm are changed by the incremental approach. The only thing to note is the increase in memory usage, already described in the paragraph "Advantages and disadvantages".

5 Conclusions and Further developments

? In this work a method for computing a three-dimensional Pareto frontier is proposed. Additionally, an incremental approach to the frontier computation allows for a fast update of the latter. Furthermore, since it is built on top of a well-known

algorithm for finding the optimal solutions of a set, it can perform both functions. For the same reason, the implementation of the method is simplified.

TODO: Practical application - AFT [1, 3]

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 processed; q - next point to be processed; 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** ▷ Special case (facet sharing)
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** ▷ Special case (facet sharing)
17: $v2 \leftarrow S_2$ ▷ Recover vertex $v2$ (was saved as $v2$)
18: $S \leftarrow \emptyset$ ▷ Clear saved vertices
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 $v3$ becomes 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 (facet sharing)
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 become $v1$ and $v2$)
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 in 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 that there is an existing vertex with the same values, the function returns the existing vertex 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 and 83492791, respectively [5].
Function T.CREATE(a, b, c) creates a triangle formed by the points/vertices a, b and c.