True Pareto Fronts for Multi-objective AI Planning Instances

Alexandre Quemy(✉) and Marc Schoenauer

TAO Project, INRIA Saclay and LRI - University of Paris-Sud and CNRS, Orsay, France
{alexandre.quemy,marc.schoenauer}@inria.fr

Abstract. Multi-objective AI planning suffers from a lack of benchmarks with known Pareto Fronts. A tunable benchmark generator is proposed, together with a specific solver that provably computes the true Pareto Front of the resulting instances. A wide range of Pareto Front shapes of various difficulty can be obtained by varying the parameters of the generator. The experimental performances of an actual implementation of the exact solver are demonstrated, and some large instances with remarkable Pareto Front shapes are proposed, that will hopefully become standard benchmarks of the AI planning domain.

1 Introduction

Contrary to single objective problems, Multi-Objective Problems (MOP) involve several contradictory criteria to be optimized. This distinction entails a modification of the concept of optimality itself: the optimal solution of a MOP is not a single solution but a set of solutions that represents trade-offs known as the Pareto Set. This set is made of the non-dominated points of the search space, i.e. the solutions that cannot be improved w.r.t. one objective without deteriorate at least another one. Formally, x dominates y if $\forall i \in \{1, \dots, n\},\ f_i(x) \succeq f_i(y)$ and $\exists j \in \{1, \dots, n\},\ f_j(x) \succ f_j(y)$. The projection of the Pareto Set over the objective space is called the Pareto Front.

Many benchmark suites exist for continuous multi-objective optimization (the famous ZDT [9], IHR [1], ...), for which the exact Pareto Front can be analytically computed, and with known difficulties (e.g. dimensionality, shape of the Pareto Fronts, existence of local Pareto-optima, ...). For combinatorial optimization, however, the situation is not yet so clear, and whereas there exist famous benchmark problems of all sizes, their true Pareto Fronts are only exactly known for the simplest problems (see e.g., MOCOLIB[1], offering several instances of several well-known combinatorial benchmark problems).

The benchmark suite introduced in the present work is concerned with *AI planning*: A planning domain D is defined by a set of predicates that define

[1] http://www.mcdmsociety.org/MCDMlib.html.

G. Ochoa and F. Chicano (Eds.): EvoCOP 2015, LNCS 9026, pp. 197–208, 2015.
DOI: 10.1007/978-3-319-16468-7_17

the state of the system when instantiated and a set of possible actions that can be triggered in states where their pre-conditions are satisfied, resulting in a new state. A planning problem instance $\mathcal{P}_D(I, G)$ is defined on a given planning domain D by a list of objects, used to instantiate the predicates to define the states, an initial state I and a goal state G. The aim is to come up with an optimal *feasible plan*, i.e., a set of actions that, when applied in turn to the initial state, lead the system to the goal state, and is optimal w.r.t. a given measure: the number of actions, or the total cost of the plan when actions have non-uniform costs, or the total *makespan* (total duration of the plan) when actions have durations, and can be run in parallel.

MiniZenoTravel is a simple temporal planning domain related to logistics, inspired by the well-known ZenoTravel problem introduced in the 3rd edition of the IPC series[2]. It involves cities, passengers, and planes (see e.g., Fig. 1); Planes can fly from one city to another when a link exists (on Fig. 1, the flight duration is attached to the link); Planes fly either empty, or carrying a unique passenger – and these are the only possible actions. A MiniZenoTravel instance is defined by the number of cities and the graph of the possible flights between them, a number of passengers and a number of planes. In the initial state I, all passengers and planes are in city c_I, and in the goal state G, all passengers must be in city c_G. Previous work proposed a multi-objective version of these benchmarks called MultiZenoTravel, by adding a *cost* for landing in some cities: the second objective is to minimize the *total cost* of the plan [2,8]. The latter work demonstrated that such problems could provide Pareto Fronts of various shapes and difficulties. However, the authors were only able to provide the exact Pareto Front for very small instances, due to the combinatorial explosion of the solution space.

The present work formally analyzes the MultiZenoTravel benchmarks and provides an algorithm to compute their true Pareto fronts in reasonable time, even for very large instances. Beyond providing a generic way to generate Pareto Fronts of tunable complexities for AI Planning, the proposed MultiZenoTravel benchmarks will allow testing different multi-objective optimization algorithms, from generic decomposition methods (weighted sum aggregation, Tchebycheff decomposition, Boundary Intersection approach – see e.g., [6]) to Pareto-based Evolutionary Algorithms, on complex benchmarks for which the Pareto Front is exactly known.

The paper is organized as follows: Sect. 2 formally presents the MultiZenoTravel benchmark, proving some properties of their Pareto optimal plans. Building on these properties, Sect. 3 proposes the ZenoSolver algorithm to actually derive the true Pareto Front for these instances. Sample experimental results demonstrate the diversity of Pareto Fronts that can be obtained, and gives performance measurements of its complexity on large instances.

[2] http://ipc.icaps-conference.org/.

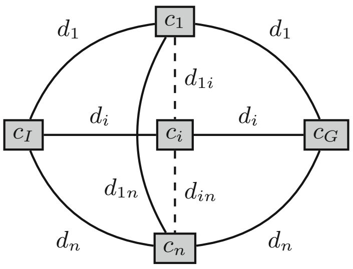


Fig. 1. A schematic view of a general MultiZenoTravel problem.

2 MultiZenoTravel Problem

2.1 Instances

Let us introduce some notations related to the planning problem briefly presented in the introduction: a MultiZenoTravel instance (Fig. 1) is defined by the following elements:

- n central cities, organized as a clique in which every node is connected to C_I and C_G, respectively the initial city and the goal city.
- $c \in (\mathbb{R}^+)^n$, where c_i is the cost for landing in C_i.
- $D \in (\mathbb{R}^+)^{n\times n}$, where D_{ij} is the flying time between C_i and C_j.
- $d^I \in (\mathbb{R}^+)^n$, where d_i^I is the flying time between C_I and C_i.
- $d^G \in (\mathbb{R}^+)^n$, where d_i^G is the flying time between C_i and C_G.
- p planes, initially in C_I, that have a capacity of an unique person.
- t persons, initially in C_I.

As said, the goal is to carry all t persons, initially in c_I, to c_G using p planes, minimizing both the makespan and the cost of the plan. In order to ease the identification of the true Pareto Front, a symmetry constraint is added: $\forall i \in [1, n], d_i^I = d_i^G$ and from thereon we will refer to a unique vector d.

Without loss of generality, all pairs (d_i, c_i) are assumed to be pairwise distinct. Otherwise, the 2 cities can be "merged" and the resulting $n - 1$ cities problem is equivalent to the original n cities problem, as there exist no city capacity constraints. Finally, we only consider cases where $t \geq p$, as the problem is otherwise trivial.

2.2 Pareto Optimal Plans

Let us make another simplifying assumption:
Assumption A1: $\forall (i, j) \in [1, n]^2, d_i + d_j < d_{ij}$[3]. Then the following holds.

[3] This might look unrealistic in real-world logistic domain. However, we hypothesize that the proposition still holds with the weaker condition that for any cities C_i, C_j, C_k (if we state the cost of C_I and C_G are respectively C_0 and C_{n+1}), $d_{ik} \leq d_{ij} + d_{jk}$ (triangle inequality).

Proposition: Pareto-optimal plans are plans where exactly $2t - p$ (possibly identical) central cities are used by a flight.

Proof: Consider a plan where a person flies from C_i to C_j. Using the same plane, the same person could fly instead from C_i to C_G, and the plane would return empty to C_j. The plan could continue unchanged from thereon: because of the hypothesis on makespans, the needed resource would be in C_j on time. Moreover, the total cost is unchanged, and the total makespan is lower or equal to the original one: the new plan thus Pareto-dominates the original one.

Iterating the same reasoning for each person, and each empty plane, we conclude that there are no flights between central cities in Pareto-optimal plans. Thus bringing the t persons from C_I to C_G will amount to carry each person through one central city: t flights will be needed from C_I to one C_i, then t flights from C_i to C_G. Finally, because planes do not need to come back from C_G in the end, only $t - p$ flights back empty will be needed, possibly through some different central cities – hence the result. $\square$

PPPs and Admissible PPPs: According to the above proposition, a Possibly Pareto-optimal Plan (PPP) is defined by 2 tuples, namely $e \in [0, n]^t$ for cities involved in eastbound flights, and $w \in [0, n]^{t-p}$ for westbound flights. Nevertheless, e and w do not hold any information about which plane will land in a particular city. This is the reason why there exists many feasible schedules, i.e., schedules that actually are feasible plans for p planes[4] using the corresponding $4t - 2p$ edges. There are at most $n^{(2t-p)}$ possible PPP but it is clear that the set of PPPs contains many redundancies, that can easily be removed by ordering the indices:

Definition: An *admissible PPP* is a pair of $E \times W$, where $E = \{e \in [1, n]^t; \forall i \in [1, t-1], d_{e_i} \geq d_{e_{i+1}}\}$ and $W = \{w \in [1, n]^{t-p}; \forall i \in [1, t-p-1], d_{w_i} \geq d_{w_{i+1}}\}$.

Number of admissible PPPs: Let K_k^m be the set of k-multicombinations (or multi-subset of size k) with elements in a set of size m. The cardinality of K_k^m is $\Gamma_k^m = \binom{m+k-1}{k}$. As E is in bijection with K_t^n, and W with K_{t-p}^n, the number of PPP is $\Gamma_t^n \Gamma_{t-p}^n$, i.e., $\binom{n+t-1}{t}\binom{n+(t-p)-1}{t-p}$.

Cost of a PPP: Given the PPP $C = (e, w) \in E \times W$, the cost of **any** plan using only the cities in e and w is uniquely defined by $\text{Cost}(C) = \sum_{e_i \in e} c_{e_i} + \sum_{w_i \in w} c_{w_i}$.

Makespan of a PPP: The makespan of a PPP is thus that of the shortest schedule that uses its $4t - 2p$ edges in a feasible way. Trivial upper and lower bounds for the shortest makespan of a PPP C are respectively $M_S(C)$, the makespan of the sequential plan (i.e., that of the plan for a single plane that would carry all persons one by one), and $M_L(C)$, the makespan of the perfect plan where none of the p planes would ever stay idle. As discussed in Sect. 3, these bounds are useful to prune the set of PPPs.

[4] Most of them are probably not Pareto-optimal, but w.r.t the previous proposition, any schedule resulting from a larger tuple e or w would be Pareto-dominated.

$$M_S(C) = 2(\sum_{e_i \in e} d_{e_i} + \sum_{w_i \in w} d_{w_i}) \qquad M_L(C) = \frac{M_S(C)}{p}$$

Greedy domination: Given two PPP C and C', C *greedily dominates* C' if $M_S(C) \leq M_L(C')$ and $Cost(C) \leq Cost(C')$.

2.3 Computing the Shortest Makespan

Flight Patterns. Clearly, within a PPP, all possible plane moves can be categorized into only 3 patterns:

P1: plane leaves C_I (non empty), flies eastward to city C_i, and goes on to C_G.
P2: plane leaves C_G (empty), flies westward to city C_i, and goes on to C_G.
P3: two planes are involved here; first plane leaves C_I (with a passenger), flies to city C_i, and goes back empty to C_I; second plane leaves C_G empty, flies to C_i, and flies back with the passenger to C_G. Note that there can be some delay between the drop-off of the person at the central city, and the arrival of the second plane.

Given a feasible plan using only the three above patterns, let α_E, α_W, and β be the numbers of effective P1, P2, and P3 patterns respectively. It is clear that β entirely determines α_E and α_W, as $\alpha_W = t-p-\beta$ and $\alpha_E = t-\beta$. Considering a PPP C, it is possible for a given β to have multiple choices for the cities involved in P3. Each choice is denoted β_{set} and the set of β_{set} the β-PowerSet.

The optimal makespan for a given admissible PPP C is the lowest makespan obtained for all $\beta_{\text{set}} \in \beta$-PowerSet. Once the optimal makespan for a couple (C, β_{set}) determined, iterating over the β-PowerSet held by C returns the optimal makespan for C. Finally, iterating the process over the set of PPP returns the Pareto Front for the considered instance.

The method to compute the optimal makespan for a particular couple (C, β_{set}) is broken down into two steps. In a first step, each β_{set} defines a subproblem without any P3 that is easy to solve. The second step is to take into account the P3 patterns in C. After detailing these two steps, we will give a constructive proof that the obtained makespan is optimal.

Step 1: Handling P3-Free PPPs. For a given $((e, w), \beta_{\text{set}})$ denote $e' = e \backslash \beta_{\text{set}}$, i.e. the tuple e from which all elements of β_{set} have been removed, and $w' = w \backslash \beta_{\text{set}}$ defined similarly. As a result, $((e', w'), \emptyset)$ is the subproblem of $((e, w), \beta_{\text{set}})$ that does not contain any P3 ($\beta' = 0$).

For a PPP with $\beta = 0$, greedy Algorithm 1 dispatches the longest flight durations first, assigning them to the available planes with shortest 'private' makespan (who have yet flown the less), ending with the one-way last flights from C_I to C_G (planes end in C_G). The algorithm returns the flight durations (D_k^1) for all planes k (to be used in the second step), and the optimal makespan for the subproblem is obviously $\max_k(D_k^1)$.

Algorithm 1. Computing the optimal makespan of PPP (e, w) when $\beta = 0$

```
$i \leftarrow 1$ ; $j \leftarrow 1$        {Indices of cities in $e$ and $w$ resp., longest durations first}
$D_k \leftarrow 0,\ k = 1, \ldots, p$        {'Private' makespan for plane $k$}
$S_k \leftarrow EAST,\ k = 1, \ldots, p$        {All planes are in $C_I$, going eastward}
while $j \leq t - p$ do
    $k \leftarrow \mathrm{ArgMin}_i(D_i)$        {Plane with shortest private makespan, in $C_I$ or $C_G$}
    if $S_k = EAST$ then
        $D_k \leftarrow D_k + 2d_{e_i}$        {From $C_I$ to $C_G$ through city $C_{e_i}$}
        $S_k \leftarrow WEST$ ; $i \leftarrow i + 1$
    else
        $D_k \leftarrow D_k + 2d_{w_j}$        {From $C_G$ to $C_I$ through city $C_{w_j}$}
        $S_k \leftarrow EAST$ ; $j \leftarrow j + 1$
    end if
end while        {Are there persons and planes left in $C_I$?}
while $i \leq t$ do
    $k \leftarrow \underset{i;S_i=EAST}{\mathrm{ArgMin}}\ (D_i)$        {Plane in $C_I$ with shortest private makespan}
    $D_k \leftarrow D_k + 2d_{e_i}$        {From $C_I$ to $C_G$ through city $C_{e_i}$}
    $S_k \leftarrow WEST$ ; $i \leftarrow i + 1$
end while
return $(D_k)_{k=1,\ldots,p}$        {All private makespans are needed for the second step}
Makespan $(e, w) = \max_{k=1,\ldots,p} \{D_k\}$
```

Step 2: Tackling Patterns P3. The second step consists in dispatching the durations of P3 patterns among the planes according to their previous flight durations $(D_k^1)_{k=1,\ldots,p}$, by sequentially assigning the longest P3 flight to the two planes with the smallest current flight durations. This can be performed greedily again, with a slightly modified version of the Algorithm 1, if we only consider the flight durations.

However, within a P3 pattern, if the plane coming from C_G lands in the central city before the person has yet arrived from C_I, it has to wait. Consequently, it is possible that the makespan of the plan is not simply the sum of the pattern durations. Indeed, the described algorithm is not taking into account the possibility of a waiting point and this is the reason why we first have to discuss the construction of a feasible plan according to the final vector of durations $(D_k^2)_{k=1,\ldots,p}$ before discussing the optimality of $\max_{k=1,\ldots,p} \{D_k^2\}$ as the makespan of the associated PPP.

Proposition: It is always possible to construct a feasible plan with the makespan returned by the two-steps method described above.

Proof by construction: Considering a P3 pattern performed by planes p_1 in C_G and p_2 in C_I through city C_i. Their schedules will look something like:

$$p_1 : C_I \rightarrow \ldots \rightarrow C_G \rightarrow \overset{\blacklozenge}{C_i} \rightarrow C_G \rightarrow \ldots \rightarrow C_G$$
$$p_2 : C_I \rightarrow \ldots \rightarrow C_I \rightarrow \overset{\lozenge}{C_i} \rightarrow C_I \rightarrow \ldots \rightarrow C_G$$

Let $\blacklozenge$ denote the time t_1 when plane p_1 arrives in C_i and $\lozenge$ the time t_2 when plane p_2 (with the person) arrives in C_i. If $t_2 > t_1$ then plane p_1 will have to wait $t_2 - t_1$ in C_i before flying back to C_G with the person. But the duration vector (D^2_k) returned by the two steps algorithm is computed assuming no waiting point. Consequently, the proposition is equivalent to assert that we can always build a plan without any waiting point.

In order to do so, for each P3, the idea is to perform the westward part as early as possible, and on the opposite, to perform the eastward part as late as possible, thus ensuring that there is no waiting time.

In order to construct such an optimal plan, we will remember the cities of every plane and every pattern during both previous steps of the algorithm. From there on, let us consider now only planes that have to perform at least one P3 pattern.

1. For each plane, select the one with the maximum number of P3 patterns to be performed. In case of tie, select the plane with longest P3 duration, or the plane with the largest current 'private' makespan.
2. Construct a partial schedule with only P1 and P2 patterns (Step 1 above).
3. For every 'not already started' P3 pattern, add its eastward part at the end of the schedule by descending order of durations.
4. For every 'already started' P3 pattern, add its westward part at the beginning of the schedule by ascending order of durations. □

Example: Considering $t = 7$, $p = 3$, $d = (2, 4, 6)$, $C = (3, 3, 2, 2, 2, 1, 1)(3, 3, 2, 1)$ and $\beta_{\text{set}} = \{3, 2, 1\}$ leads to the sub-problem $C' = (3, 2, 2, 1)(3)$ with $t' = 4$. Step 1 above gives the 'private' makespans D^1_k in the table below. Adding the P3 patterns (Step 2) gives the 'private' makespans D^2_k. The complete schedule can then be built according to the method described above.

p_i	D^1_i	D^2_i	P3
p_1	12	32	2
p_2	24	28	1
p_3	8	32	3

$$p_1 : c_I \to c_2 \to c_G \to \overset{\blacklozenge_3}{c_3} \to c_G \to \overset{\blacklozenge_2}{c_2} \to c_G \to \overset{\blacklozenge_1}{c_1} \to c_G$$
$$p_2 : c_I \to \overset{\lozenge_1}{c_1} \to c_I \to c_2 \to c_G \to c_3 \to c_I \to c_2 \to c_G$$
$$p_3 : c_I \to \overset{\lozenge_3}{c_3} \to c_I \to \overset{\lozenge_2}{c_2} \to c_I \to c_3 \to c_G$$

Hence, there is no waiting time within P3s, and the optimal makespan is 32.

Proposition: For a given PPP C and β_{set}, the algorithm returns the optimal makespan.

Proof: The incompressible time to transport all passengers, according to a given β_{set} is $T = 4 \sum\limits_{i \in \beta_{\text{set}}} d_i + 2 \sum\limits_{i \in \{e', w'\}} d_i$. A theoretical optimal plan with this pattern repartition is a plan without any waiting point for any plane. The above algorithm gives the optimal distribution of the set of times into p. Then, if a plan can be constructed with such a makespan, it is optimal for the PPP and the repartition of patterns. As it exists a method to construct such a plan, we can conclude that the algorithm is optimal for the PPP C and β_{set}. □

Complexity: Given a PPP, the worst case occurs when $w \subset e$ and $w_i \neq w_j$ if $i \neq j$. Hence, for each value of β there are $\binom{t-p}{\beta}$ possible β_{set}. As $0 \leq \beta \leq t-p$, we will perform 2^{t-p} *iterations* of the two step algorithm. A large upper-bound for the whole PPP set is hence $2^{t-p}\binom{n+t-1}{t}\binom{n+(t-p)-1}{t-p}$.

However, if an upper bound on β is given by $t-p$, a tighter upper-bound can be found as explained by the following example and due to the fact that the worst case situation for a PPP rarely occurs in the whole PPP set, the real number of iterations for a given instance is far from the above bound.

Example: Considering $C = (3,1,1)(2,1)$, the trivial upper bound is equal to two but actually, it is impossible to operate a P3 using the city C_2 since it is not in the tuple e.

3 ZenoSolver

ZenoSolver is a C++11 software dedicated to generate and exactly solve MultiZenoTravel instances. Firstly, it allows to tune every parameter in order to adjust the difficulty or to obtain different shapes of Pareto Fronts. In particular, vectors c and d are generated using two user-defined functions, f and g, such that $c_i = x_c f(i) + y_c$ and $d_i = x_d g(n-i) + y_d$, insuring that both objectives are conflicting. Second, ZenoSolver outputs the corresponding `PDDL` file[5], that can be directly used by any standard AI planner.

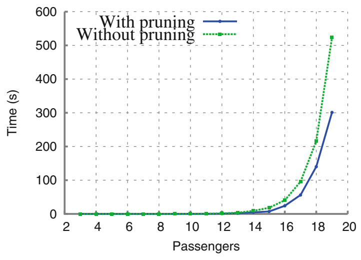
Finally, ZenoSolver computes the true Pareto Front using the algorithm described in Sect. 2, iterating over $E \times W$, storing for each value of the total cost the PPP with best makespan to date, without explicitly constructing the set of admissible PPPs. Using the Greedy domination, ZenoSolver implements a pruning method that checks if the current PPP is dominated by any other PPP already stored. As noted, the optimal makespan is lower or equal than the upper bound M_S, leading to an efficient pruning. Indeed, as PPPs are generated in an approximated increasing order [4], this avoids to iterate over the whole set to check the domination criterion.

Determining if the current PPP is dominated has complexity $O(h)$ where h is the number of different total costs. An obvious upper-bound for h is given by $(2t-p)(\max_i(c_i) - \min_i(c_i))$. However, in practice, S seems to have the same order of magnitude than the exact Pareto Front. In addition, S is the only structure kept in memory, thus, from this point of view, ZenoSolver turns out to be near-optimal regarding the memory usage (see Table 1).

3.1 Empirical Performances

Empirical complexity. The number of iterations is influenced by the number of PPPs but also by their structure. Indeed, increasing n does not significantly impact the average number of iterations per PPP since the upper-bound is 2^{t-p}.

[5] Planning Domain Definition Language, universally used now in AI Planning to describe domains and instances.



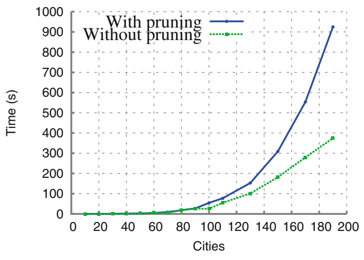
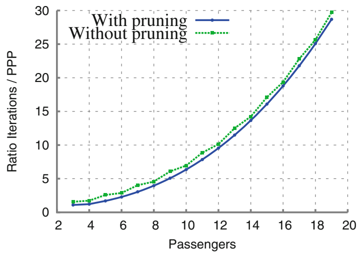


Fig. 2. Time function of t (left) or n (right) for $f(i) = g(i) = i$.



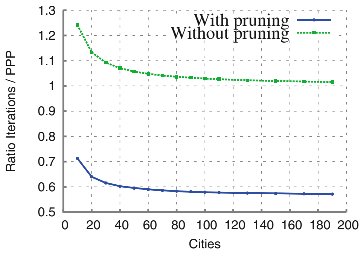
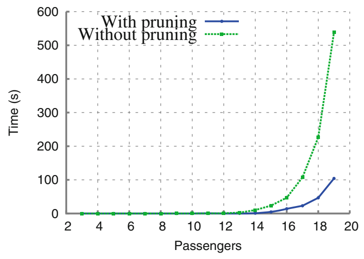


Fig. 3. Ratio of iterations over the number of PPPs, function of t (left) or n (right) for $f(i) = g(i) = i$.

On the opposite, increasing t leads to a dramatic growth of both the upper-bound and the average number of iterations per PPP. Figure 2, which displays the time vs t or n plots, confirms this remark: it requires the same CPU time for $t = 18$ than for $n = 165$.

Pruning or not pruning? The benefits of the pruning method strongly rely on the average number of iterations per PPP: Pruning becomes more efficient as t increases, as shown by Fig. 2. Furthermore, increasing n while pruning can degrade performances, even if there are less iterations than PPPs (Fig. 3 compared to Fig. 2). Note that the number of iterations follows the number of PPPs while increasing n, but explodes with t, which is in line with the previous remark.

Also, the efficiency pruning seems to be instance-dependent. Fixing n, t and p, different generating functions result in different numbers of iterations and CPU times, as demonstrated by Fig. 4. There are however some clear cases in favor of pruning, e.g. with $n = t = 9$: ZenoSolver requires 1.26×10^9 iterations and 2222 seconds without pruning. Using pruning, for $f(i) = \sqrt{i}$ and $g(i) = i$, it requires only 119000 iterations performed in 26 seconds, but 36000 iterations in 53 seconds with $f(i) = log(i)$ and $g(i) = \sqrt{i}$. In general, using concave generating functions leads to more optimistic conclusions regarding the benefits of pruning PPPs.



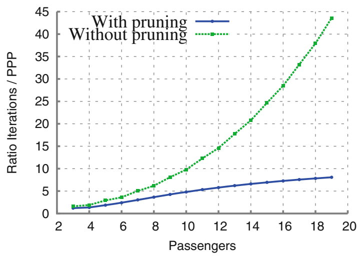


Fig. 4. Time and ratio for generating functions $f(i) = g(i) = log(i)$.

Table 1. Increasing simultaneously n and t with $f(i) = g(i) = i$.

n	t	p	PPP Size	Iterations	S Size	Front Size	Time (ms)
3	3	2	30	33	9	5	0
4	4	2	350	408	19	10	1
5	5	2	4410	6387	33	17	6
6	6	2	58212	109831	51	26	117
7	7	2	792792	1930385	73	37	2278
8	8	2	11042460	34648348	99	50	43572
9	9	2	156434850	630225670	129	65	1036772
10	10	2	2245709180	11600589455	163	82	20785211

3.2 Reference Large Instances

As mentioned in the introduction, the combinatorial multi-objective optimization domain suffers from a lack of benchmarks with a known Pareto Front but also with a *concave* or *non-regular* shapes[6].

Even if anyone can generate different instances by tuning ZenoSolver parameters to obtain the desired front shape with accuracy, we identified some large instances with totally different front shapes and complexities as displayed in Fig. 5: They could be a basic set of representative instances for MultiZeno-Travel, allowing fair comparisons between various solvers and approaches. Note that more large instances with different complexities can be found on the website of the Descarwin Project https://descarwin.lri.fr.

Table 2 gives the parameters used by ZenoSolver to build them, as well as some statistics about their complexity. The choice of the generating functions is purely empirical, guided by the fact that we would like to obtain mainly piecewise concave fronts with uneven point distributions. This is why none of these fronts is linear, though some seem to be at large scale (see detailed insets in some plots). Also note the non-uniform distribution of the points on the Instances 3, and the

[6] In the context of discrete optimization, the word "concave" seems rather abusive. However, we will call here concave parts of a Pareto front where all points are above the segment made of the two extreme points, w.r.t. the direction of optimization.

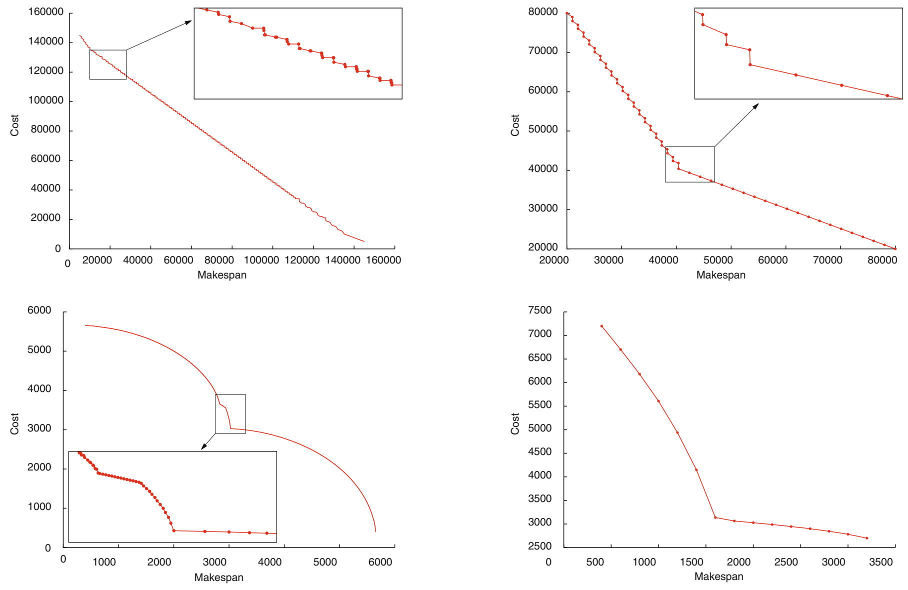


Fig. 5. True Pareto Fronts for the instances described by Table 2. Remember that these Pareto fronts are made of discrete points: the lines are visual helps to make the general shape clear.

Table 2. Large instances: parameters and generation statistics.

Inst.	n	t	p	Generating functions		Pareto#	h	PPP(k)	Iter.(k)	Time
1	20	6	2	$\frac{5}{2}i + \frac{(i \bmod 2)}{10}$	$\frac{5}{2}i + \frac{(i \bmod 2)}{10}$	409	4015	1568220	3317140	16h46
2	3	21	2			61	861	53	233	2006s
3	200	3	2	$\sqrt{i}$	$\sqrt{i}$	538	4963	270680	3906	1845s
4	8	26	25	$\sqrt{i}$	i	15	190	34176	60457	4240s

few Pareto points of the Instance 4 in spite of the complexity of this instance (26 persons), due to the small ratio $\frac{p}{t}$. The generating time strongly varies from some minutes up to hours and thus confirm dependency on the generating functions of the ZenoSolver complexity.

4 Conclusion and Perspectives

This paper has extended the MultiZenoTravel test suite in multi-objective AI planning. Furthermore, not only did we provide here a general approach to generate more complex Pareto fronts than in our previous work [2], but we also proposed here ZenoSolver, an exact solver that is provably able to exactly solve the multi-objective optimization problem (i.e., to identify the true Pareto front) for even very large instances. The complete code is publicly available at https://descarwin.lri.fr, making it easy for everyone to generate his/her own benchmark instances. However, we also provided in this paper a few typical instances that exhibit very different shapes of Pareto Fronts, for different levels of complexity.

The proposed benchmark suite opens the floor to sound comparative experiments in a combinatorial domain where, as far as we know, no ground truth (i.e., true Pareto front) existed for large instances. On-going work is concerned with using these benchmarks to compare different multi-objective optimization algorithms. Preliminary results [7] have already confirmed that Pareto-based Evolutionary Algorithms outperform the basic weighted sum aggregation in the case of complex non-convex Pareto fronts. However, deeper experiments should be made with state-of-the-art decomposition algorithms in which the different components of the decomposition cooperate (e.g., from the MOEA/D family [6]). In particular, in the AI planning domain, using these benchmarks will hopefully lead to more sound comparisons between Pareto and non-Pareto planners (see e.g., [3,5]).

References

1. Igel, C., Hansen, N., Roth, S.: Covariance matrix adaptation for multi-objective optimization. Evol. Comput. **15**(1), 1–28 (2007)
2. Khouadjia, M.R., Schoenauer, M., Vidal, V., Dréo, J., Savéant, P.: Multi-objective AI planning: evaluating DaE$_{\text{YAHSP}}$ on a tunable benchmark. In: Purshouse, R.C., Fleming, P.J., Fonseca, C.M., Greco, S., Shaw, J. (eds.) EMO 2013. LNCS, vol. 7811, pp. 36–50. Springer, Heidelberg (2013)
3. Khouadjia, M.R., Schoenauer, M., Vidal, V., Dréo, J., Savéant, P.: Pareto-based multiobjective AI planning. In: Rossi, F. (ed.) IJCAI, pp. 2321–2328. AAAI Press, Menlo Park (2013)
4. Knuth, D.E.: The Art of Computer Programming, Generating All Tuples and Permutations. Addison-Wesley, Reading (2005)
5. Sroka, M., Long, D.: Exploring metric sensitivity of planners for generation of pareto frontiers. In: Kersting, K., Toussaint, M. (eds.) 6 STAIRS, pp. 306–317. IOS Press, Amsterdam (2012)
6. Zhang, Q., Li, H.: A multi-objective evolutionary algorithm based on decomposition. IEEE Trans. Evol. Comput. **11**(6), 712–731 (2007)
7. Quemy, A., Schoenauer, M., Vidal., V., Dréo, J., Savéant, P.: Solving large multizenotravel benchmarks with divide-and-evolve. In: Daenens, C., et al. (ed.) Proceedings of LION'9. Springer (2015, To appear)
8. Schoenauer, M., Savéant, P., Vidal, V.: Divide-and-evolve: a new memetic scheme for domain-independent temporal planning. In: Gottlieb, J., Raidl, G.R. (eds.) EvoCOP 2006. LNCS, vol. 3906, pp. 247–260. Springer, Heidelberg (2006)
9. Zitzler, E., Deb, K., Thiele, L.: Comparison of multiobjective evolutionary algorithms: empirical results. Evol. Comput. **8**(2), 173–195 (2000)