Proceedings of the 1st Workshop on
Evolutionary Techniques in Planning and Scheduling

EVOPS 2013

Rome, Italy - June 11, 2013

Edited By:

Marco Baioletti, Valentina Poggioni, Marc Schoenauer, Vincent Vidal

Organizing Commitee

Program committee

Table of Contents

Towards a new generation ACO-Based Planner

M. Baioletti, R. Minciarelli, F. Paolucci, V. Poggioni

Abstract

In this paper a new generation ACO-Based Planner, called ACOPlan 2013, is described. This planner is an enhanced version of ACOPlan, a previous ACO-Based Planner (Baioletti et al. 2011), which differs from the former in the search algorithm and in the implementation, now done on top of Downwards. The experimental results, even if are not impressive, are encouraging and confirm that ACO is a suitable method to find near optimal plan for propositional planning problems.

Introduction

The basic principle of the first generation of ACO-Based planner ACOPlan, described in (Baioletti et al. 2011; 2009a; 2009b), was to use the well known *Ant Colony Optimization* metaheuristic (*ACO*) (Dorigo and Stuetzle 2004) to solve planning problems with the aim of optimizing the quality of the solution plans. The approach was based on the strong similarity between the process used by artificial ants to build solutions and the way used by state–based planners to find solution plans. Therefore, we had defined an ACO algorithm which handles a colony of planning ants with the purpose of solving planning problems by optimizing solution plans with respect to the overall plan cost.

ACO is a metaheuristic inspired by the behavior of natural ants colony which has been successfully applied to many *Combinatorial Optimization* problems. Being ACO a stochastic incomplete algorithm, there is no guarantee that optimal solutions are ever found, but in many CO problems ACO is able to find very good or near optimal solutions, sometimes being competitive with state-of-arts algorithms.

In this paper a new generation of planners based on ACO search algorithm is introduced. The most important change lies in the search process performed by the ants. In particular the ants start from the most promising states reached in the previous generations and perform a variable number of steps. A smaller number of steps L is used to enhance exploitation, while a larger number of steps B gives more importance to exploration. The parameters L and B are tuned by means of an auto-adaptive process. Other new features are described in Section 4.

The paper is structured as follows. In the two first sections a brief introduction to the metaheuristic ACO, and the previous generation of ACOPlan algorithm are described, while, in the next section the characteristics and peculiarity of new generation of ACO based algorithms are presented. The experimental results are described in Section 5, while some conclusions are drawn in Section 6.

Ant Colony Optimization

ACO is a well–known metaheuristic to tackle Combinatorial Optimization problems introduced since early 90s by Dorigo et Al. (Dorigo and Stuetzle 2004). It is inspired by the foraging behavior of natural ant colonies. When walking, natural ants leave on the ground a chemical substance called *pheromone* that other ants can smell. This stigmergic mechanism implements an ”indirect communication way” among ants, in particular when looking for the shortest path to reach food.

ACO is usually applied to optimization problems whose solutions are composed by discrete components. A Combinatorial Optimization problem is described in terms of a *solution space* S, a set of (possibly empty) *constraints* Ω and an *objective function* $f : S \to \mathbb{R}^+$ to be minimized (maximized).

The colony of artificial ants builds solutions in an incrementally way: each ant probabilistically chooses a component to add to a partial solution built so far, according to the problem constraints. The random choice is biased by the *artificial* pheromone value τ related to each component c and by a heuristic function η. Both terms evaluate the desirability of each component. The probability that an ant will choose the component c is

$$p(c) = \frac{[\tau(c)]^\alpha [\eta(c)]^\beta}{\sum_x [\tau(x)]^\alpha [\eta(x)]^\beta} \qquad (1)$$

where the sum on x ranges on all the components which can be chosen, and α and β are tuning parameters which differentiate the pheromone and heuristic contributions.

The pheromone values represent a kind of memory shared by the whole ant colony and are subject to *u*pdate and *e*vaporation. In the most applications only the best solutions are considered in the pheromone update phase: the global best solution found so far (*best–so–far*) and/or the best solution found in the current iteration (*iteration–best*). Moreover, most ACO algorithms use the following update rule (Blum 2005):

$$\tau(c) \leftarrow (1-\rho)\cdot\tau(c) + \rho \cdot \sum_{s\in\Psi_{upd}\,:\,c\in s} F(s) \qquad (2)$$

where Ψ_{upd} is the set of solutions involved in the update, F is the so called *quality function*, which is a decreasing function of the objective function f (increasing if f is to be maximized), and $\rho \in]0,1[$ is the pheromone evaporation rate. ρ is a typical ACO parameter which was introduced to avoid a premature convergence of the algorithm towards sub–optimal solutions.

The simulation of the ant colony is iterated until a satisfactory solution is found, a termination criterion is satisfied or a given number of iterations is reached.

First generation ACO based planners

According to the main features of ACO the ants–planners of the colony are stochastic and heuristic–based.

Each ant–planner executes a forward search, starting from the initial state $\mathcal{I}$ and trying to reach a state in which the goal $\mathcal{G}$ is satisfied. The solution is built step by step by adding components. At each step, the search process performs a randomized weighted selection of a solution component c which takes into account both the pheromone value $\tau(c)$ associated to the component and the heuristic value $\eta(a)$ computed for each action a executable in the current state and related to the chosen solution component. Once an action a has been selected, the current state is updated by means of the effects of a.

The construction phase stops when at least one of the following termination condition is verified

1. a solution plan is found, i.e. a state where all the goals are true is reached;
2. a dead end is met, i.e. no action is executable in the current state;
3. an upper bound L_{max} for the number of execution steps is reached.

Variants of the system was been presented and experimented, for example in (Baioletti et al. 2011; 2009a; 2009b). These variants differ in several point, for example in terms of pheromone models, evaluation functions and implementation, but they share the general algorithm presented in Fig. 1. The experimental evaluation presented in (Baioletti et al. 2011) shows that the approach was competitive and comparable with the state of the art. There was also an unsuccessful version that due to some unexpected bugs run at the IPC 2011 with terrible results.

Let $(\mathcal{I},\mathcal{G},\mathcal{A})$ be the planning problem, the optimization process is iterated for a given number N of iterations, in which a colony of n_a ants build plans with a maximum number of steps L_{max}. At each step, each ant chooses an action among the executable ones by the *ChooseAction* function that encodes the transition probability function previously described. When all ants have completed the search phase, the best plan π_{iter} of the current iteration is selected and the global best plan π_{best} is possibly updated. Finally, the pheromone values of the solution components are updated by means of the function *UpdatePheromone* that implements the updating rules 2. Relevant parameters are c_0 which denotes the initial value for the pheromone, ρ which represents the evaporation rate and σ that is a parameter of the pheromone update rule (see 3).

Algorithm 1 The algorithm ACOPlan

```
1:  $\pi_{best} \leftarrow \emptyset$
2:  InitPheromone($c_0$)
3:  for $g \leftarrow 1$ to $N$ do
4:      for $m \leftarrow 1$ to $n_a$ do
5:          $\pi_m \leftarrow \emptyset$
6:          $s \leftarrow \mathcal{I}$
7:          $A_1 \leftarrow$ executable actions in $\mathcal{I}$
8:          for $i \leftarrow 1$ to $L_{max}$ while $A_i \neq \emptyset$ and $\mathcal{G} \not\subseteq s$ do
9:              $a \leftarrow$ ChooseAction($A_i$)
10:             extend $\pi_m$ with $a$
11:             $s \leftarrow Res(s,a)$
12:             $A_{i+1} \leftarrow$ executable actions on $s$
13:         end for
14:     end for
15:     find $\pi_{iter}$
16:     update $\pi_{best}$
17:     UpdatePheromone($\pi_{best}$, $\pi_{iter}$, $\rho$, $\sigma$)
18: end for
```

Pheromone Models

The effectiveness of an ACO algorithm firstly depends on the choice of the pheromone model and its data structures. A good model should be simple to compute but enough informative to characterize the context in which an ant–planner can choose a specific action. Moreover it should allow them to distinguish the context of most successful choices from the worst ones. On the other hand the characterization of the component should not be too much detailed in order to allow the pheromone to deposit in a significant quantity.

In (Baioletti et al. 2011) several pheromone models have been proposed and empirically compared by systematic experiments.

State-State (SS): A component is defined by the current state s. This is one of the most expensive pheromone model from the space complexity point of view, because the number of possible states is exponential with respect to the problem size.

State-Action (SA): The pheromone value τ depends on the current state s and on the action a to be executed. This model is even more expensive than SS, because for each state s there can exist several actions executable (and chosen) in s. On the other hand, the pheromone values can be interpreted in terms of a preference policy: $\tau(a,s)$ represents how much it is desirable, or it has been useful, to execute a in state s.

Action-Action (AA): In this model a notion of local history is introduced: the pheromone depends both on the action a under evaluation and on the last executed action a', i.e. the pheromone is a function $\tau(a,a')$. Considering only the previous action is the simplest way in which the action choice can be directly influenced by history of previous decisions. AA allows a manageable representation and defines a sort of local first order Markov property.

Fuzzy Level-Action (FLA): The basic idea underlying this model is to associate the action under evaluation with the plan time step, i.e. the planning graph level, where it is executed. Since the limited number of levels, such approach has a more tractable space complexity with respect to the SA model. On the other hand a pure LevelAction model would present the drawback that the pheromone of an action at a time step t cannot be used in other close time steps, while it is often likely that an action desirable at certain time step, say 2, will also be desirable at close time steps, say 1 and 3. To solve this problem the FLA model which fuzzifies the LevelAction representation just described has been introduced: when pheromone is distributed over an action a executed at time step t is also spread in decreasing quantity over the same action in the close time steps, conversely the pheromone level computed for an action a to be executed at time t is computed as the weighted average of the pheromone values $\tau(a, t)$, where $t = tW, ..., t + W$ computed with the LevelAction model, where weights, i.e. the spread distribution, and the time window W are parameters of the model. The two models showing the best performances were *AA* and *FLA*.

The heuristic

The heuristic function is a key feature of ACOPlan because it directly affects the transition probability function (1) used to synthesized the solution plan. The heuristic value for a component c is defined by

$$\eta(c) = \frac{1}{h(s_c)}$$

where h is an heuristic function which evaluates the state s_c resulting from the execution of the action a_c associated to the component c in the current state.

The ACOPlan variants used both the heuristic function FF and its variant FFAC for actions costs. This function was presented in (Baioletti et al. 2011); to note that FFAC does not have static costs as in the heuristic proposed by Keyder and Geffner in (Keyder and Geffner 2008) but dynamic costs depending on the level at hand; it is similar to the heuristic function used in SAPA (Do and Kambhampati 2003) with the sum propagation for action cost aggregation.

Plan comparison and Pheromone updating

A critical point of the optimization process is the ability of comparing plans found by the colony of planner ants. This is particularly important in pheromone updating phase where the best plan of the iteration must be selected, as well as in the general ACOPlan which returns the best plan found in all the iterations.

Any comparison criteria should obviously prefer a *solution plan* to a *non solution plan*. On the other hand comparison of two *solution plans* π, π' can be easily based on actions costs. A comparison criteria cannot be easily defined when both plans π and π' are not solution plans. In this case a plan π is evaluated by a combination of the heuristic value on the best state s ever reached by π with the cost of reaching s.

The *pheromone update phase* evaporates all the pheromone values and increases the pheromone value of the components belonging to π_{iter} and π_{best} according to the formula

$$\tau(c) \leftarrow (1 - \rho) \cdot \tau(c) + \rho \cdot \Delta(c) \qquad (3)$$

where

$$\Delta(c) = \begin{cases} \sigma & \text{if } c \text{ belongs to } \pi_{iter} \\ 1 - \sigma & \text{if } c \text{ belongs to } \pi_{best} \\ 1 & \text{if } c \text{ belongs to both} \\ 0 & \text{otherwise} \end{cases}$$

and σ is usually set to $\frac{2}{3}$ to give more influence to the exploration.

Second generation ACO based planners

The new version of ACOPlan, called ACOPlan 2013, differs from the previous implementation in several points.

A first important difference is that ACOPlan 2013 is now implemented in top of Fast Downwards (Helmert 2006), as other planners, like Lama (Richter and Westphal 2010). In this way we can exploit of efficient routines for parsing the planning problem, translating into a finite domain representation, finding landmarks and computing heuristic functions. Nevertheless, our search engine is independent on the actual representation of states and only uses some Downward internal procedures, like the successor generator procedure.

The most important algorithmical difference lies in the choice of starting point and the length of the exploration performed by each ant. In the previous version of ACOPlan, each ant starts from the initial state and begins to build a plan for at most a certain number N of steps.

Since each ant always starts from the initial state, all the states close to the initial state are explored very often, thus wasting a lot of computation time. The repeated exploration of some states is a major problem in this kind of algorithms. LAMA and other similar planners avoid at all to explore a state s more than once, except when s is reachable from the initial state by a path shorter than before. In ACOPlan, a state could be explored several times, in different iteration or by different ants, and forcing the algorithm to avoid already visited states would make the search process incomplete. The use of a cache data structure, where information about already visited states are stored, only mitigates the computation efforts of re-exploring states.

In ACOPlan2013 the choice of the starting point is made in two steps.

First of all, an already found plan p, among the best-so-far and the iteration-best, is selected at random with equal probabilities. Then a state s is randomly chosen in the following way. With equal probability, s is the last state reached by p, the best state in the whole p, or the best state reached in the last 10 steps in p.

Then the ant starts its exploration from s and its initial plan is the prefix of the plan p, truncated at the step corresponding to s.

In this way, an important drawback of ACOPlan is overcome, i.e. that each generation tries to build plans starting from scratch, thus discarding many information acquired by the previous generations (except for the pheromone contribution). Indeed, in ACOPlan 2013, the ants start from the best states reached in the previous generations.

The bound N on the plan length was another major problem of ACOPlan. It is obvious that N should be large enough to allow the ant to find a plan, for instance $N \geq PL$, where PL is the length of a solution plan, which should be estimated somehow from the problem to be solved (for instance as a multiple of a relaxed plan starting from the initial state) or fixed to a large enough value.

In ACOPlan 2013, the exploration is performed using two different bounds, denoted by L (Little) and B (Big), on the number of steps from s. Usually $L < B$. The bounds L and B are used in alternation, and the sequence of generation where L is used is called "Little" phase, while in those where L is used is called "Big" phase.

In the "Little" phase, the ants explore for a small number of steps, and in this way they can exploit and continue, as much as possible, the exploration performed in the previous generation. The aim of a "little" number of steps is to make small, but progressive enhancements.

In the "Big" phase, the ants explore for a greater number of steps, thus the possibility of finding completely different solutions and the exploration of a larger part of the search space are possible.

Hence, the alternation between "Little" and "Big" phases then corresponds to having phases in which a different weight is given to the exploitation and to the exploration aptitudes.

The bounds L and B start with the value $L = 10$ and $B = 100$ and are changed in a dynamic way, making them converging to a common intermediate value.

In this way, the algorithm dynamically adapts the search process to give more importance to the exploration or to exploitation, according to the number of improvements of the heuristic function and the number of solutions found.

Another important new feature is the particular use of two heuristic functions, h_{FF} (i.e. the FF heuristics) and h_{LM} (i.e. the Landmarks count heuristics), both used also by LAMA (Richter and Westphal 2010).

In Lama each state is evaluated with both heuristic functions. In the ACOPlan approach, the simultaneous use of both of them, although possible, is a little bit problematic. While the formula for the transition probabilities can be easily extended to take into account of two heuristic functions, the major difficulty is that states (and plans) are evaluated by using heuristic values, hence the presence of two heuristic functions can cause situations of incomparable states.

Moreover, since the number of state evaluations required by ACOPlan is large, a double heuristic evaluation can be computationally heavy.

Therefore, we decide to mostly use one heuristic function and the decision is automatically taken during the search process seeing the progress obtained by using all the two heuristics for some generations each. If both heuristic functions produces comparable results, they are kept for the next sequence of generations. On the other hand, if a heuristic function h appears to work better than the other h', a "last chance" is given to h'. After this period, if h is still better than h', h' is removed and only h is used in the remaining generations.

Experiments

This section presents and discusses the results of experiments held with this new generation of ACOPlan optimizing the overall plan execution costs.

The benchmarks domain problems from the last planning competition IPC2011 () has been used in the experiments. The results presented here refers to the domains *Barman*, *Elevators*, and *Parcprinter*.

In these experiments the pheromone model *FLA* has been used.

Since the behavior of ACOPlan depends on many parameters, a preliminary phase of parameters tuning has been held by systematic tests in order to establish the ACOPlan general setting: 10 planner–ants, $\alpha = 3$, $\beta = 3$, $\rho = 0.10$. Moreover, since ACOPlan is a not deterministic planner, the results collected for each single problem instance are the median values obtained over 15 runs. The experiments were run on the EGI grid.

Comparisons have been made with LAMA 2011 (Richter and Westphal 2010) and FDss-1(Fawcett et al. 2011) respectively the winner and the runner-up planners at the Sequential Satisficing track of the last planning competition IPC2011.

For each tested domains a table showing the score obtained by each planner with respect the best cost solution (the known best cost). The score is computed as in the planner competition, that is the ratio between the solution cost and the cost of the best solution known so far, $score = cost/best\_cost$.

The tests show ACOPlan 2013 obtain comparable results with respect to the other planners, even if it is apparent that ACOPlan 2013 is not competitive with them. Anyway its results on Elevators are better than those of LAMA and FDss-1.

Conclusions and Future Works

In this paper we have described ACOPlan 2013, a new version of the planner ACOPlan, which is based on a new search algorithm and a new implementation. This new planner is then compared with some state-of-arts planners, obtaining encouraging results. Although ACOPlan 2013 is not competitive, it appears to be comparable with them and the improvements obtained with respect to the results obtained by ACOPlan are impressive both from the quality of the solutions found and the number of problems solved whith the time limit. Hence, we believe that ACO still remains a good method to solve planning problems.

Following this line, it is apparent that for this approach there is room for improvement and it can be made competitive with the state-of-art by using, for instance, learning or auto-adaptivity methods to tune the parameters of the algorithm.

Table 1: Results for plan quality on **Barman** domain

Problem	Lama-2011 score	FDss-1 score	ACO (avg) score	best sol cost
p01	0.92	0.99	0.70	279
p02	0.98	0.87	0.66	259
p03	0.85	0.99	0.75	274
p04	0.86	0.99	0.70	281
p05	0.86	0.92	0.69	297
p06	0.90	0.83	0.72	322
p07	0.98	0.99	0.63	305
p08	0.86	0.83	0.73	290
p09	0.91	0	0.58	348
p10	0.93	0.94	0.70	323
p11	0.86	0	0.70	354
p12	0.82	0.75	0.54	334
p13	0.90	1.00	0.71	396
p14	0.85	0.72	0.60	372
p15	0.96	0.99	0.64	387
p16	0.87	1	0.65	386
p17	0.87	1	0.62	383
p18	0.73	0.62	0.65	380
p19	0.90	1	0.65	387
p20	0.89	0.91	0.70	356
total	17.70	16.34	13.3	

Table 2: Results for plan quality on **Elevators** domain

Problem	Lama-2011 score	FDss-1 score	ACO (avg) score	best sol cost
p01	0.52	0.75	0.73	191
p02	0.46	0.71	0.75	417
p03	0.45	0.62	0.76	464
p04	0.92	0.75	0.59	256
p05	0.48	0.61	0.80	253
p06	0.47	0.68	0.67	513
p07	0.48	0.66	0.63	409
p08	0.38	0.68	0.58	505
p09	0.45	0.73	0.74	671
p10	0.40	0.61	0.68	602
p11	0.58	0.56	0.53	635
p12	0.48	0.53	0.57	691
p13	0.53	0.72	0.88	992
p14	0.55	0.61	0.62	804
p15	0.57	0.42	0.69	923
p16	0.50	0.62	0.53	891
p17	0.53	0.64	0.52	1066
p18	0.47	0.59	0.57	1148
p19	0.41	0.41	0.56	1417
p20	0.64	0.61	0.85	1386
total	10.28	12,52	13.25	

Table 3: Results for plan quality on **Parcprinter** domain

Problem	Lama-2011 score	FDss-1 score	ACO (avg) score	best sol cost
p01	1.00	1.00	1.00	1383121
p02	1.00	1.00	1.00	1852217
p03	0.93	1.00	0.96	2490322
p04	1.00	0.78	0.91	2754187
p05	1.00	1.00	1.00	1216462
p06	1.00	1.00	1.00	1270874
p07	0.91	1.00	0.98	2121255
p08	1.00	1.00	1.00	1681282
p09	1.00	1.00	0.97	2387265
p10	1.00	1.00	0	2021893
p11	0.86	1.00	1.00	1891203
p12	1.00	0.88	0.97	2828340
p13	0.83	0.83	0.99	3335367
p14	1.00	0.74	0.87	3119803
p15	1.00	0.74	0.85	3160821
p16	0.83	0.70	0.81	3526437
p17	0.99	1.00	0	1556448
p18	0.99	1.00	0	2376643
p19	0.98	1.00	0.98	3072626
p20	1.00	1.00	0	2308715
total	19.32	18.68	15.29	

As a future work we are also planning to study new pheromone models, by exploiting, for instance, the finite domain representation computed by Fast Downwards (Helmert 2006).

Acknowledgements

The research leading to the results presented in this paper has been possible thanks to the grid resources and services provided by the European Grid Infrastructure (EGI), the Italian Grid Infrastructure (IGI) and the National Grid Initiatives that support the Virtual Organization (VO) COMPCHEM.

References

Baioletti, M.; Milani, A.; Poggioni, V.; and Rossi, F. 2009a. An ACO approach to planning. In *Proc of the 9th European Conference on Evolutionary Computation in Combinatorial Optimisation, EVOCOP 2009*.

Baioletti, M.; Milani, A.; Poggioni, V.; and Rossi, F. 2009b. Ant search strategies for planning optimization. In *Proc of the International Conference on Planning and Scheduling, ICAPS 2009*.

Baioletti, M.; Milani, A.; Poggioni, V.; and Rossi, F. 2011. Experimental evaluation of pheromone models in acoplan. *Ann. Math. Artif. Intell.* 62(3–4):187–217.

Blum, C. 2005. Ant colony optimization: Introduction and recent trends. *Physics of Life Reviews* 2(4):353–373.

Do, M. B., and Kambhampati, S. 2003. Sapa: A multi-objective metric temporal planner. *Journal of Artificial Intelligence Research (JAIR)* 20:155–194.

Dorigo, M., and Stuetzle, T. 2004. *Ant Colony Optimization*. Cambridge, MA, USA: MIT Press,.

Fawcett, C.; Helmert, M.; Hoos, H.; Karpas, E.; Roeger, G.; and Seipp, J. 2011. Fd-autotune: Domain-specific configuration using fast downward. In *ICAPS 2011, PAL Workshop, Runner-up in learning track, IPC 2011*.

Helmert, M. 2006. The fast downward planning system. *J. Artif. Intell. Res. (JAIR)* 26:191–246.

Keyder, E., and Geffner, H. 2008. Heuristics for planning with action costs revisited. In *Proc. of ECAI 2008*, 588–592.

Richter, S., and Westphal, M. 2010. The lama planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res. (JAIR)* 39:127–177.

Pareto-Based Multiobjective AI Planning

To appear in: S. Thrun and F. Rossi, eds, *Proceedings IJCAI 2013*

Mostepha Khouadjia Marc Schoenauer
TAO Project, INRIA Saclay
LRI Paris-Sud University, Orsay, France
first.last@inria.fr

Vincent Vidal
ONERA
Toulouse, France
Vincent.Vidal@onera.fr

Johann Dréo Pierre Savéant
Thales Research & Technology
Palaiseau, France
first.last@thalesgroup.com

Abstract

Real-world problems generally involve several antagonistic objectives, like quality and cost for design problems, or makespan and cost for planning problems. The only approaches to multiobjective AI Planning rely on metrics, that can incorporate several objectives in some linear combinations, and metric sensitive planners, that are able to give different plans for different metrics, and hence to eventually approximate the Pareto front of the multiobjective problem, i.e. the set of optimal trade-offs between the antagonistic objectives. Divide-and-Evolve (DAE) is an evolutionary planner that embeds a classical planner and feeds it with a sequence of subproblems of the problem at hand. Like all Evolutionary Algorithms, DAE can be turned into a Pareto-based multiobjective solver, even though using an embedded planner that is not metric sensitive. The Pareto-based multiobjective planner MO-DAE thus avoids the drawbacks of the aggregation method. Furthermore, using YAHSP as the embedded planner, it outperforms in many cases the metric-based approach using LPG metric sensitive planner, as witnessed by experimental results on original multiobjective benchmarks built upon IPC-2011 domains.

1 Introduction

Multiobjective problems are ubiquitous in the real world, where most situations often involve at least two antagonistic objectives, such as maximizing some quality criterion (or even criteria) while minimizing some costs – and quality increase cannot be obtained without corresponding cost increase. This is true in AI planning too, as witnessed by looking at the most popular test problems that have been used in IPC competitions. Many domains have been defined in both categories of actions with cost and temporal planning: the more general problem is to minimize both the makespan (where high quality solutions correspond to small makespan values) and the cost of a given plan, while these two objectives are in general antagonistic[1].

Given two solutions A and B of such multiobjective problems, A is obviously to be preferred to B in the case when the objective values for A are all better than the objective values of B: in such case, A is said to Pareto-dominate B. However, Pareto-dominance is not a total order, and most solutions are not comparable for this relationship. The set of interest when facing a multiobjective problem is the so-called *Pareto set* of all solutions of the search space that are not dominated by any other solution: such non-dominated solutions are the best possible trade-offs between the antagonistic objectives, in that there is no way to improve on one objective without degrading at least another one. Figure 1 depicts a simple case of a two objectives AI Planning problem, and presents both the *design space*, space of solutions plans, and its projection on the *objective space*, here the (makespan$\times$cost) space (both to be minimized). The *Pareto front* (circles on the right figure) is the image of the Pareto set in the objective space.

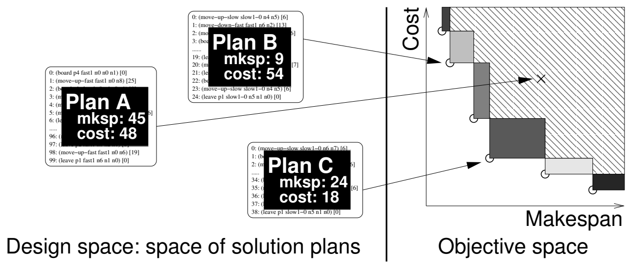


Figure 1: Design and objective spaces for a two-objectives planning problem. The hatched+grey area contains all images of possible solutions plans; this area defines the *hypervolume* of the set of circles, and the *hypervolume contribution* of each point is the grey area of the corresponding small rectangle (see Section 3). Circles are Pareto-optimal (non-dominated) solutions, aka the *Pareto front* of the problem at hand.

Sometimes, the user/decision maker might have a very precise idea of the relative losses induced by the degradation of one of the objectives with respect to the improvement of

This work was partially funded by the DESCARWIN ANR project (ANR-09-COSI-002).

[1]Though some costs might be proportional to durations in some domains.

another. It is then possible to turn the multiobjective optimization problem into a single-objective optimization problem, e.g., by optimizing some weighted sum (or any other monotonous function) of the objectives, in the so-called *aggregation method*. Any optimizer can then be used to solve the aggregated problem. However, this approach requires some a priori knowledge of the trade-off between the objectives, and/or numerous runs of the optimizer on different aggregations of the objectives. Furthermore, linear aggregation (the weighted sum case) is not able to identify the complete Pareto front in case it is not convex.

To address this multidimensional issue, Pareto-based algorithms have been designed in order to directly identify the complete Pareto front, by computing a set of approximate non-dominated solutions. This one can then be offered to the decision maker so that she/he can make an informed decision when choosing a solution. Efficient Pareto-based multiobjective algorithms have been designed using ideas from Evolutionary Algorithms, that can easily be turned into Multiobjective Evolutionary Algorithms (MOEAs) by modifying their selection process to account for Pareto dominance [Deb, 2001].

In the domain of AI planning, most works only address single-objective problems, and the very few recent approaches rely on *metric sensitive* planners to optimize metrics built as weighted sums of the objectives (more in Section 2). This paper introduces MO-DAE, the first (to the best of our knowledge) truly Pareto-based Multiobjective AI Planning System. MO-DAE is a multi-objectivization of DAE, a domain-independent satisficing planner that has been originally designed for single-objective planning [Schoenauer *et al.*, 2006; Bibaï *et al.*, 2010a], and won the IPC-2011 temporal deterministic satisficing track at ICAPS 2011. DAE uses an Evolutionary Algorithm (EA) to evolve sequences of partial states for the problem at hand, calling an embedded planner to solve in turn each subproblem of the sequence. If the embedded planner is able to compute metrics along any plan it builds, then MO-DAE will take care of the global search for the Pareto front, without the need for the embedded planner to be metric sensitive. After a brief survey of Multiobjective Evolutionary Algorithms (MOEAs, Section 3), Section 4.2 details DAE and MO-DAE.

Although there exist many single-objective planning benchmarks, thanks to the IPC competitions, none has been proposed yet for multiobjective planning. Both a specific tunable artificial benchmark, and a general method to turn some well-known IPC benchmarks into multiobjective domains are presented in Section 5. In Section 6, MO-DAE$_{\text{YAHSP}}$, the instantiation of MO-DAE$_{\text{X}}$ with YAHSP [Vidal, 2004] as the embedded planner, is validated on these instances, and compared to the metric-based approach using the metric sensitive planner LPG [Gerevini *et al.*, 2008], following Sroka and Long [2012b]. Finally, Section 7 discusses these results and sketches the directions for further research.

2 Multiobjective AI Planning

Temporal planning and numerical state variables have been formalized in PDDL2.1 [Fox and Long, 2003], as well as metric functions that allow to optimize an aggregation of some criteria based on time and numerical variables. This language has been extended in PDDL3 [Gerevini and Long, 2006] in order to express preferences and soft constraints, which de facto increase the expressivity and complexity of the metric function to be optimized. However, even though optimizing such functions is a standard way to tackle multiobjective optimization problems, its extension to Pareto-based multiobjective optimization is not straightforward and nothing had been proposed in AI Planning for that purpose until very recently. Indeed, all the literature about multi-criteria/objective planning, such as the works on the planners GRT [Refanidis and Vlahavas, 2003], SAPA [Do and Kambhampati, 2003] or LPG [Gerevini *et al.*, 2008], are concerned with optimizing an aggregation of the objectives.

The concept of metric sensitive planners has been recently defined in Sroka and Long [2012a], in order to identify planners able to give diverse solutions when faced with – possibly small – variations of the metric function. With such a planner, the Pareto front can be approximated by adequately weighting an aggregation of the objectives and running the planner several times within some time limits. An example of such a metric function is $\alpha \times \texttt{makespan} + (1-\alpha) \times \texttt{cost}$, where alpha is sampled in $[0, 1]$. The authors experimented several candidates and retained LPG which exhibited by far the best performance in terms of metric sensitivity and generated Pareto front quality [Sroka and Long, 2012b]. For this reason, this approach, named here MO-LPG, will also be used as a baseline for the validation here (Section 6). To overcome the metric insensitivity limitation of other planners, Sroka and Long [2012a] suggest to add artificial bounds on numerical variables, which gave comparable results with the other planners in comparison with MO-LPG. However, this requires significantly more engineering, as defining the bounds requires an in-depth analysis of the planning problem, while defining objective weights is straightforward.

3 Multiobjective Evolutionary Algorithms

Evolutionary Algorithms (EAs) [Eiben and Smith, 2003] are heuristic stochastic search algorithms that crudely mimic natural evolution. A population of individuals (a set of potential solutions in the search space) evolves according to two main driving forces: reproduction through blind variations (random moves in the search space) and natural selection, aka "survival of the fittest". Blind variations depend on the search space, and are usually classified into crossover operators, that involve two or more parent individuals to create one offspring, and mutation operators, that modify a single parent to create one offspring. Selection is applied to choose which parents will reproduce, and also which from the parents plus offspring will survive to the next generation. It can be deterministic or stochastic, but has to be biased toward the fittest individuals.

In the case of single-objective optimization of some objective function $\mathcal{F}$ (e.g., to be minimized), the fitness of an individual x simply is the value $\mathcal{F}(x)$. In the case of multiobjective optimization, however, Pareto-dominance is not a total order, and hence cannot be used as sole selection criterion. Several Pareto-based Multiobjective Evolutionary Algorithms (MOEAs) have thus been proposed, that use some diversity measure as a secondary criterion when Pareto-dominance cannot distinguish between individuals.

Several indicators have been proposed for comparing the results of different multiobjective optimization algorithms, i.e., that compare sets of solutions. The most popular is the *hypervolume indicator* (Figure 1), because it is the only one that has been proved to-date to be consistent with the Pareto-dominance relation. But indicators can also be used to build a fitness function for some MOEAs: the fitness of an individual (compared to the other individuals in the population) is its contribution to the indicator of the population, i.e., the difference between the indicator of the whole population and that of the population without it. Such MOEAs are called *Indicator Based Evolutionary Algorithms* (IBEA) [Zitzler and Künzli, 2004] – and IBEA$_H$, that will be used throughout this paper, is the one using the hypervolume indicator.

4 Divide-and-Evolve

4.1 Single-Objective Divide-and-Evolve

This section introduces the main principles of the satisficing planner DaE, referring to [Bibaï *et al.*, 2010a] for a comprehensive presentation. Given a planning problem $P = \langle A, O, I, G \rangle$, where A denotes the set of atoms, O the set of actions, I the initial state, and G the goal state, DaE$_X$ searches the space of sequences of partial states $(s_i)_{i\in[0,n+1]}$, with $s_0 = I$ and $s_{n+1} = G$: DaE$_X$ looks for the sequence such that the plan σ obtained by compressing subplans σ_i found by some embedded planner X as solutions of $P_i = \langle A, O, \hat{s}_i, s_{i+1} \rangle_{i\in[0,n]}$ has the best possible quality (with $\hat{s}_i$ denoting the final state reached by applying σ_{i-1} from $\hat{s}_{i-1}$). Each intermediate state $(s_i)_{i\in[1,n]}$ is first seen as a set of goals and then completed as a new initial state for the next step by simply applying the plan found to reach it. In order to reduce the number of atoms used to describe these states, DaE relies on the admissible heuristic function h^1 [Haslum and Geffner, 2000]: only the ones that are possibly true according to h^1 are considered. Furthermore, mutually exclusive atoms, which can be computed at low cost, are also forbidden in intermediate states s_i. These two rules are strictly imposed during the random initialization phase, and progressively relaxed during the search phase. The compression of subplans is required by temporal planning where actions can run concurrently: a simple concatenation would obviously not produce the minimal makespan.

Due to the weak structure of the search space (variable-length sequences of variable-length lists of atoms), Evolutionary Algorithms (EAs) have been chosen as the method of choice: EAs are metaheuristics that are flexible enough to explore such spaces, as long as they are provided with some stochastic *variation operators* (aka *move operators* in the heuristic search community) – and of course some objective function to optimize.

Variation operators in DaE are (i) a crossover operator, a straightforward adaptation of the standard one-point crossover to variable-length sequences; and (ii) different mutation operators, that modify the sequence at hand either at the sequence level, or at the state level, randomly adding or removing one item (state or atom).

The objective value is obtained by running the embedded planner on the successive subproblems. When the goal state is reached, a feasibility fitness is computed based on the compression of solution subplans, favoring quality; otherwise, an unfeasibility fitness is computed, implementing a gradient towards satisfiability (see [Bibaï *et al.*, 2010a] for details).

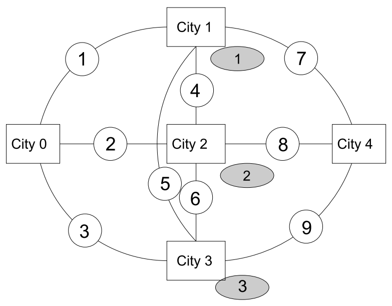
DaE can embed any existing planner, and has to-date been successful with both the optimal planner CPT [Vidal and Geffner, 2004] and the lookahead heuristic-based satisficing planner YAHSP [Vidal, 2004]. The latter has been demonstrated to outperform the former when used within DaE [Bibaï *et al.*, 2010b], so only DaE$_{\text{YAHSP}}$ has been considered in this work.

4.2 Multiobjective Divide-and-Evolve

Two modifications of DaE$_{\text{YAHSP}}$ are needed to turn it into a MOEA: (i) use some multiobjective selection (Section 3) in lieu of the single-objective tournament selection that is used in the single-objective context; (ii) use the embedded planner to compute the values of both objectives (e.g., makespan and cost). The former modification is straightforward, and several alternatives have been experimented within [Khouadjia *et al.*, 2013]. The conclusion is that $IBEA_H$ [Zitzler and Künzli, 2004] (see Section 3) performs best on instances MultiZeno (see Section 5) – and only this one will be mentioned in the following.

As explained above, the computation of the fitness is done by YAHSP – and YAHSP, like all known planners to-date, is a single-objective planner. It is nevertheless possible, since PDDL 2.1, to specify other metrics that are to be computed throughout the execution of the final plan. For the metric sensitive planners, this metric is directly used to bias the search, while some other planners, like YAHSP, simply compute it along the solution plan without interfering with the search. However, because YAHSP is both a temporal planner and a cost planner, two strategies are possible for YAHSP within MO-DaE: it can be asked to optimize only the makespan (resp. the cost), and to simply compute the cost (resp. the makespan) when executing the solution plan.

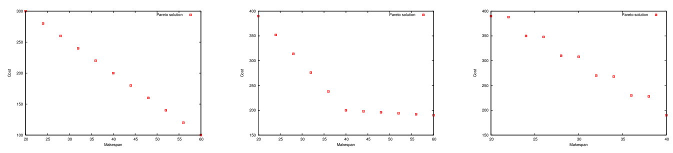
In MO-DaE$_{\text{YAHSP}}$, the choice between both strategies is governed by user-defined weights. For each individual, the actual strategy is randomly chosen according to those weights, and applied to all subproblems of the individual. Another important feature of YAHSP for the computation of the objectives of MO-DaE$_{\text{YAHSP}}$ is its stochasticity: YAHSP explores the plan space stochastically, and different runs on the same instance with different random seeds give different answers. This stochasticity, and the effect of the choice of YAHSP strategy, can be observed on Figure 4a, that represent the different objective values of the same individual obtained within MO-DaE$_{\text{YAHSP}}$ when YAHSP uses either the makespan strategy, or the cost strategy, or strategies that are independently randomly chosen for each subproblem: the bias is clear when a unique strategy is chosen for the MultiZeno9 instance, and the corresponding part of the objective space is sampled, while the hybrid strategy spreads the values in-between these two clouds of points. Note that on



Dur.	A	B	C
1	2	2	2
2	4	4	**3**
3	6	6	**4**
4	3	3	**1**
5	5	5	**2**
6	3	3	**1**
7	2	2	2
8	4	4	**3**
9	6	6	**4**

Cost			
1	30	30	30
2	20	**11**	**29**
3	10	10	10

Figure 2: Schematic view, and 3 instances, of MULTIZENO benchmark. Flight durations are attached to the possible routes (white circles), costs/risks are attached to landing in the central cities (grey circles). Three sets of values are given on the right, corresponding to Pareto fronts of Figure 3.



A - Linear B - Convex C - Concave

Figure 3: Pareto fronts for the instances from Figure 2.

some other instances (OPENSTACKS, FLOORTILE, see Section 5), the three strategies are absolutely indistinguishable (not shown here).

Some preliminary experiments were also conducted in order to try to take advantage of the stochastic nature of YAHSP evaluations, running YAHSP several times for each individual and keeping the best plan. But surprisingly, even without considering the additional CPU cost, such approach proved detrimental on the quality of the Pareto approximation: such observation had also been made when trying to hybridize EAs with local search methods (iterated YAHSP can indeed be viewed as some sort of local search) - the local search should not try to improve the solutions too early.

Final note regarding the evaluation: the strategy weights, like all parameters of MO-DAE$_{\text{YAHSP}}$, have been tuned using ParamILS (see Section 6.1), and it turned out that the optimal values for MO-DAE$_{\text{YAHSP}}$ have always been equal weights: something that was to be expected, as no objective should be preferred to the other.

5 Benchmark Domains and Instances

Two approaches were used to design multiobjective benchmark problems: first, using a highly simplified version of the well-known IPC domain ZENOTRAVEL, a simple and easy to tune domain was built, and the exact Pareto front can be easily identified for all its instances. The other approach is based on modifying IPC-2011 problems.

5.1 MULTIZENO Instances

The MULTIZENO problem domain involves cities, passengers, and planes. One plane can carry at most one passenger from one city to another (action `fly`), following an existing link of Figure 2, with corresponding flight durations (see Table). Costs are landing taxes for each of the middle cities. All instances in this work have 2 planes, all passengers are initially in `city 0`, and must reach `city 4`. The simplest non-trivial instance MULTIZENO3 has 3 passengers. In its default configuration (column A in Table), the makespan-optimal solution has a total makespan of 8, as the reader will easily find out. But all flights have to land in `city 1`, resulting in a cost of 120. The alternative route through `city 2` (resp. `city 3`) has a makespan of 16 (resp. 24), and a cost of 80 (resp. 40). By increasing the number of passengers, and modifying the flight durations and landing costs, different trade-offs are made possible. Figure 3 displays the 3 exact Pareto fronts (in the makespan $\times$ cost space) corresponding to the durations and costs of the table in Figure 2, for a total of 6 passengers (aka MULTIZENO6). Note that the second objective could also be considered as a *risk* [Khouadjia *et al.*, 2013], and the objective is then to minimize the maximum risk encountered during the execution of the plan. This variant of MULTIZENO domain will not be considered here.

5.2 Multi-Objectivization of IPC Problems

Two satisficing tracks were open at IPC-2011: sequential satisficing, i.e., sequential STRIPS planning in which actions have a cost and where the total cost is to be minimized, and temporal satisficing, where actions have a duration and can be run in parallel and where the total makespan is to be minimized[2]. Three possible ways of generating multiobjective instances have been considered. When the domains appeared in both tracks, with the same instances, and when the cost increases as the makespan decreases, a simple merge of both instances is enough. This was the case for domain ELEVATORS.

For some domains, the cost values of the cost instance did not ensure that both objectives would be antagonistic. This is the case for CREWPLANNING, FLOORTILE, and PARCPRINTER. For these instances we arbitrarily set the cost values to a maximum cost minus the value of the duration. FLOORTILE will be the typical domain from this category considered here. Finally, for the OPENSTACKS domain, the cost version has a single costly action, that penalizes the use of a new stack: such scheme is very general in scheduling applications with resources (with more resources, things get done faster, but cost more). For this domain, this cost action was simply added to the temporal domain.

6 Experiments

The goal of the following experiments is to assess the efficiency of MO-DAE$_{\text{YAHSP}}$, and its robustness with respect to some variety of planning domains and the size of the instances. The instances presented in Section 5 will be used in turn, and the performance of MO-DAE$_{\text{YAHSP}}$ will also be assessed against the baseline MO-LPG, the approach proposed in Sroka and Long [2012b] (see Section 2) which will also be evaluated. The experimental conditions will first be detailed.

[2]`www.plg.inf.uc3m.es/ipc2011-deterministic`

6.1 Parameter Tuning – Experimental Conditions

DaE$_{\text{YAHSP}}$, and even more so, MO-DaE$_{\text{YAHSP}}$, have a number of free parameters that need to be tuned in order to obtain the best possible results. It is well-known that parameter tuning can make a complete difference between failure and success for the same algorithm on the same problem. In this work, all user-defined parameters have been tuned using the framework PARAMILS [Hutter *et al.*, 2009], that handles any parameterized algorithm whose parameters can be discretized, and uses some Iterated Local Search (ILS) to explore the space of parameter configurations. Furthermore, because the goal of this work is to demonstrate the efficiency of MO-DaE to robustly find a good approximation of the Pareto front of multiobjective AI Planning problems, those parameters were tuned anew for one instance of moderate complexity in each domain (see Section 5.2), and the resulting parameter set was used for all instances of the same domain. This represents a trade-off between CPU cost and performance: there is little hope to ever find some universal parameters for DaE, that would allow DaE to obtain its best quality performance on all possible instances; on the other hand, such best quality can obviously be obtained by optimizing the parameters anew for each new instance, but at the price of a huge CPU cost. On the other hand, following Sroka and Long [2012b], within MO-LPG, LPG was ran in local-search mode, and given the same overall CPU time for its multiple restarts.

The main goal of the experiments presented here is to assess the ability of MO-DaE$_{\text{YAHSP}}$ to find good quality approximations of the Pareto front. Hence the performance of the different algorithms will be reported w.r.t. the quality of the identified Pareto front after an arbitrary CPU time of 30 min (long enough to allow the algorithms to reach some steady population, as confirmed by preliminary runs). All runs were conducted on one core of the same 24-cores server with Xeon X5650@2.67GHz processors, running Ubuntu 10.04 Lucid. DaE was implement within the PARADISEO-MOEO framework [Liefooghe *et al.*, 2007]. For all experiments, 11 independent runs were performed. All the performance assessment procedures, including the hypervolume calculations, have been achieved using the PISA performance assessment tool suite [Bleuler *et al.*, 2003].

6.2 Results on MULTIZENO Instances

Experiments have been conducted on the 3, 6 and 9 passengers versions of MULTIZENO (Section 5.1), with the Linear configuration (Figure 3-a). The MULTIZENO3 instance proved to be too easy, and both MO-DaE$_{\text{YAHSP}}$ and MO-LPG could rapidly find the complete Pareto front. For MULTIZENO6, the situation is drastically different for MO-LPG, that is only able to find a few points far away from the Pareto front (see Figure 4c). On the other hand, MO-DaE$_{\text{YAHSP}}$ perfectly identifies the complete Pareto front in all runs for the “Linear” and “Concave” cases, while 2 runs out of 11 miss one point each in the “Convex” case. Finally, when tackling MULTIZENO9 (and while MO-LPG fails to find a single feasible plan), MO-DaE$_{\text{YAHSP}}$ is able to approach the true Pareto front rather robustly, as witnessed by Figure 4b, that represents the aggregated 11 Pareto fronts of the 11 independent runs.

6.3 Results on Modified IPC-2011 Instances

Figure 5 exhibits some results on multi-objectivized IPC-2011 instances (Section 5.2) for MO-DaE$_{\text{YAHSP}}$ and MO-LPG. Indeed, because the exact Pareto front of these instances is unknown, the only possible assessment of MO-DaE$_{\text{YAHSP}}$ is by comparison to MO-LPG results.

For the ELEVATORS domain, instances 1, 5, 10 were experimented. For instance 1 (Figure 5a), MO-DaE$_{\text{YAHSP}}$ and MO-LPG find exactly the same Pareto front, but MO-LPG is unable to find any solution for instances 5 and above. On the other hand, MO-DaE$_{\text{YAHSP}}$ identifies some Pareto front, as can be seen for instance 10 on Figure 5d.

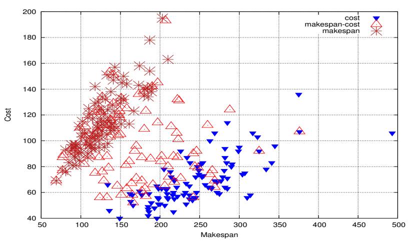
On the OPENSTACKS domain, experiments involved instances 1, 5, 10, 15 and 20. For the small instances, 5 (Figure 5b), and 1 and 10 (not shown), MO-DaE$_{\text{YAHSP}}$ clearly finds a much better Pareto front than MO-LPG. For larger instances (15 and 20, not shown), the situation is even worse for MO-LPG, that only finds very poor solutions (w.r.t. the ones found by MO-DaE$_{\text{YAHSP}}$). As an illustration of how both algorithms explore the objective space, Figures 5e and 5f show that the complete solution set computed by DaE (merge of the 11 independent approximations) is nicely distributed along the Pareto front, whereas the solutions of LPG are much more scattered in the design space.

The situation changes for FLOORTILE domain: instances 0, 3 and 4 were used, and here MO-LPG outperforms MO-DaE$_{\text{YAHSP}}$, as can be seen for instance 3 in Figure 5c. However, as the instance size increases (instance 4 and above), the gap between LPG and DaE decreases (not shown here).

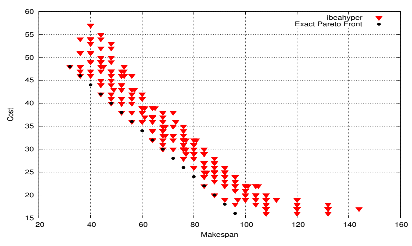
7 Discussion and Conclusion

Not all planners are metric sensitive, in the sense advocated by Sroka and Long [2012a]: the main contribution of this work is to demonstrate the ability of the MO-DaE approach to turn any planner into a multiobjective planner, provided it can reason on either objectives alone (e.g., the makespan and the cost), as has been done here with YAHSP. The resulting algorithm is a truly Pareto-based multiobjective planner, that consistently outperforms the MO-LPG metric-based approach on all instances tested here except the small FLOORTILE instances. MO-DaE$_{\text{YAHSP}}$ is able to solve much larger instances, and to find most of the time better approximations of the Pareto front than MO-LPG. More work is needed to improve even more the MO-DaE approach to multiobjective planning. But we strongly believe that the rationale underlying the original DaE is still valid, and that the decomposition-based approach that it implements will push upward the complexity of the problems that we can solve.

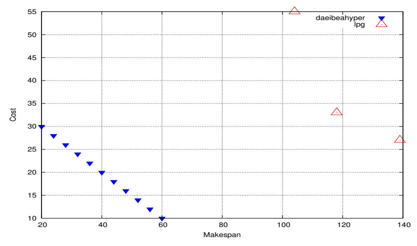
The second contribution of this work is the proposition for procedures to build multiobjective benchmark suites for AI Planning. There is no need to advocate the usefulness of large and diverse benchmark suites: in the context of single-objective optimization, advances in research (from the different versions of PDDL to the numerous powerful planners we know of today) have coevolved together with the design of the successive IPC benchmarks. Because multiobjective optimization is mandatory in order to tackle real-world problems, we strongly believe that progress in multiobjective planners



(a) Importance of YAHSP strategy

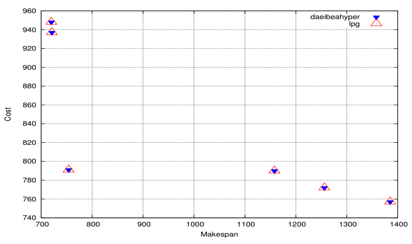


(b) MULTIZENO9: exact (●) & approx. (▼)

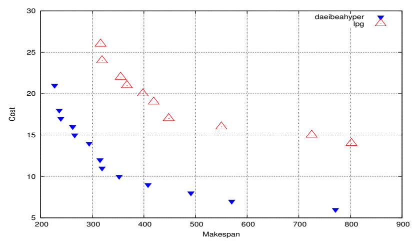


(c) MULTIZENO6: DAE (▼) vs LPG (△)

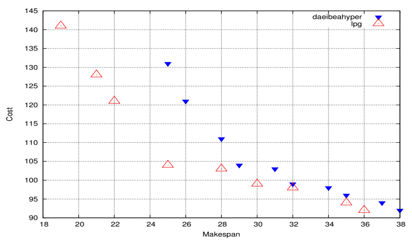
Figure 4: Experiments on MULTIZENO instances (see text for details).



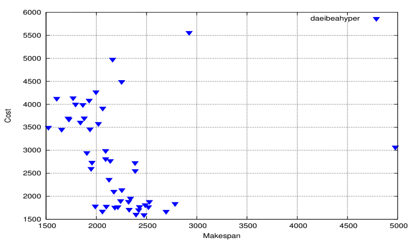
(a) ELEVATORS01:DAE (▼) vs LPG (△)



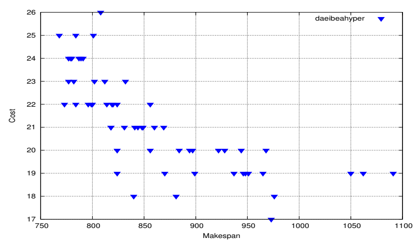
(b) OPENSTACKS5: DAE (▼) vs LPG (△)



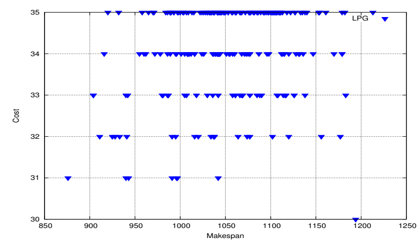
(c) FLOORTILE03: DAE (▼) vs LPG (△)



(d) ELEVATORS10: DAE



(e) OPENSTACKS15: DAE



(f) OPENSTACKS15: LPG

Figure 5: Pareto fronts identified by DAE and LPG for multi-objectivized IPC-2011 instances (a – c) and complete solution sets found by DAE (d and e) and by LPG (f), though displayed with different scales.

requires as well the design of meaningful multiobjective AI Planning benchmarks with gradual difficulties. The MULTIZENO suite detailed here is a first step in this direction: it is a tunable artificial testbed, and was shown to be able to generate interesting Pareto fronts (e.g. convex with a knee as well as non-convex). Furthermore, it has many degrees of freedom that still have not been explored: other combinations of durations and makespans, more intermediate cities, with more possible routes between them. Another possibility would have been to use the benchmarks designed by Sroka and Long [2012a; 2012b], but unfortunately the current implementation of MO-DAE$_{\text{YAHSP}}$ relies on YAHSP, which does not handle numerical state variables except for the special case of action costs. However, we believe that multiobjective instances with time and cost objectives are already challenging enough, and could enable the use or extension of many more existing planners which mainly optimize cost or time. The multi-objectivization of IPC-2011 domains is another possible route we have sketched, though more work is still required to transform the single-objective domains into "interesting" multiobjective ones, even in the favorable case where both a cost and a temporal version of the same domain already exist. When both objectives are not antagonistic enough, setting one as the inverse of the other does the trick, but the Pareto fronts remain close to linear fronts, while more interesting fronts (e.g. non-convex, "discontinuous", . . .) are necessary to test different characteristics of the planners. Furthermore, such multiobjective instances can hardly be tackled by state-of-the-art metric sensitive planners, which are among the only potential competitors as of today: beside the general difficulty of finding the proper weights depending on the objective scales, linear combinations of the objectives can only give in that case one single non-dominated plan, and other aggregations are not guaranteed to lead to points on the Pareto front. Final word on the modified IPC-2011 instances, the failure of LPG on even rather small instances suggests that we should probably have started with easiest instances (e.g., IPC-2008), since at IPC-2011 the easiest instances are significantly harder than those of IPC-2008.

References

[Bibaï *et al.*, 2010a] Jacques Bibaï, Pierre Savéant, Marc Schoenauer, and Vincent Vidal. An Evolutionary Metaheuristic Based on State Decomposition for Domain-Independent Satisficing Planning. In R. Brafman et al., editor, 20th *International Conference on Automated Planning and Scheduling (ICAPS-10)*, pages 18–25. AAAI Press, 2010.

[Bibaï *et al.*, 2010b] Jacques Bibaï, Pierre Savéant, Marc Schoenauer, and Vincent Vidal. On the Benefit of Sub-Optimality within the Divide-and-Evolve Scheme. In Peter Cowling and Peter Merz, editors, *Proc.* 10th *EvoCOP*, pages 23–34. LNCS 6022, Springer Verlag, 2010.

[Bleuler *et al.*, 2003] Stefan Bleuler, Marco Laumanns, Lothar Thiele, and Eckart Zitzler. PISA — A Platform and Programming Language Independent Interface for Search Algorithms. In *Evolutionary Multi-Criterion Optimization*, volume 2632 of *LNCS*, pages 494–508. Springer, 2003.

[Deb, 2001] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley, 2001.

[Do and Kambhampati, 2003] M.B. Do and S. Kambhampati. SAPA: A Multi-Objective Metric Temporal Planner. *J. Artif. Intell. Res. (JAIR)*, 20:155–194, 2003.

[Eiben and Smith, 2003] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer Verlag, 2003.

[Fox and Long, 2003] M. Fox and D. Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 2003.

[Gerevini and Long, 2006] A. Gerevini and D. Long. Preferences and Soft Constraints in PDDL3. In *ICAPS Workshop on Planning with Preferences and Soft Constraints*, pages 46–53, 2006.

[Gerevini *et al.*, 2008] A. Gerevini, A. Saetti, and I. Serina. An Approach to Efficient Planning with Numerical Fluents and Multi-Criteria Plan Quality. *Artificial Intelligence*, 172(8-9):899–944, 2008.

[Haslum and Geffner, 2000] P. Haslum and H. Geffner. Admissible Heuristics for Optimal Planning. In 5th *Int. Conf. on AI Planning and Scheduling (AIPS 2000)*, pages 140–149, 2000.

[Hutter *et al.*, 2009] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An Automatic Algorithm Configuration Framework. *J. Artif. Intell. Res. (JAIR)*, 36:267–306, 2009.

[Khouadjia *et al.*, 2013] M.-R. Khouadjia, M. Schoenauer, V. Vidal, J. Dréo, and P. Savéant. Multi-Objective AI Planning: Evaluating DAE-YAHSP on a Tunable Benchmark. In R. C. Purshouse et al., editor, 7th *International Conference on Evolutionary Multi-Criterion Optimization (EMO 2013)*, LNCS. Springer Verlag, 2013. To appear.

[Liefooghe *et al.*, 2007] A. Liefooghe, M. Basseur, L. Jourdan, and E.G. Talbi. ParadisEO-MOEO : A Framework for Evolutionary Multi-Objective Optimization. In *Evolutionary multi-criterion optimization*, pages 386–400. Springer, 2007.

[Refanidis and Vlahavas, 2003] I. Refanidis and I. P. Vlahavas. Multiobjective Heuristic State-Space Planning. *Artificial Intelligence*, 145(1-2):1–32, 2003.

[Schoenauer *et al.*, 2006] Marc Schoenauer, Pierre Savéant, and Vincent Vidal. Divide-and-Evolve: a New Memetic Scheme for Domain-Independent Temporal Planning. In J. Gottlieb and G. Raidl, editors, *Proc.* 6th *EvoCOP*, pages 247–260. LNCS 3906, Springer, 2006.

[Sroka and Long, 2012a] Michal Sroka and Derek Long. Exploring Metric Sensitivity of Planners for Generation of Pareto Frontiers. In K. Kersting and M. Toussaint, editors, *The Sixth "Starting Artificial Intelligence Research" Symposium (STAIRS 2012)*, pages 306–317. IOS Press, 2012.

[Sroka and Long, 2012b] Michal Sroka and Derek Long. LPG Based System for Generation of Pareto Frontiers. In P. Gregory, editor, 30th *PlanSIG Workshop*, 2012.

[Vidal and Geffner, 2004] V. Vidal and H. Geffner. Branching and Pruning: An Optimal Temporal POCL Planner Based on Constraint Programming. In *Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, pages 570–577. AAAI Press, 2004.

[Vidal, 2004] Vincent Vidal. A Lookahead Strategy for Heuristic Search Planning. In 14th *International Conference on Planning and Scheduling (ICAPS-04)*, pages 150–159. AAAI Press, 2004.

[Zitzler and Künzli, 2004] E. Zitzler and S. Künzli. Indicator-Based Selection in Multiobjective Search. In Xin Yao et al., editor, 8th *International Conference on Parallel Problem Solving from Nature (PPSN VIII)*, pages 832–842. LNCS 3242, Springer Verlag, 2004.

Scheduling Single-Armed Cluster Tools with Time Window Constraints Using Differential Evolution Algorithm

Xin Li **Richard Y. K. Fung**
Department of Systems Engineering and Engineering Management
City University of Hong Kong, 83 Tat Chee Avenue, Kowloon, Hong Kong, China
shenelee@gmail.com richard.fung@cityu.edu.hk

Abstract

To improve overall equipment efficiency (OEE) of cluster tools, a differential evolution (DE) algorithm was proposed in the paper, with residency time constraints considered in the process modules. To begin with, scheduling problem domains in cluster tools were supposed and defined, then a non-linear programming model was developed to minimize the makespan with no defective wafers. Based on the model, a scheduling algorithm using the DE algorithm was presented. The performance of the algorithm proposed was analyzed and evaluated by simulation experiments. Results indicated that the proposed algorithm was valid and practical to generate satisfied scheduling solutions.

1 Introduction

In semiconductor manufacturing fabrications, processes consist of oxidizing a wafer surface, coating photosensitive chemicals onto the surface, exposing it to a circuit image from a light source, developing and etching the circuit pattern, depositing other chemicals onto it, diffusing and implanting additional chemicals on the etched pattern, and so on. Cluster tools (Zant 2004), which are complex in the framework and prohibitively expensive in cost, have been increasingly used for most of these fabrication processes. Thus, in order to schedule and control cluster tools to improve the Overall Equipment Efficiency (OEE), developing effective and practical methods has become significant tasks to be dealt with in the semiconductor manufacturing industry.

According to SEMI Standard E21-96, a cluster tool is defined as an integrated, environmentally isolated and vacuumed manufacturing system consisting of process, transport, and cassette modules mechanically linked together. Fig. 1 shows a cluster tool with 4 process modules (PM) and a single-armed robot as the transport module (TM). There are many new challenges for this special type of integrated manufacturing systems, which are quite different with traditional manufacturing situations. Wafers are transported by the single-armed robot between process modules and /or cassette modules. At most one wafer can stay (for processing or waiting) at a processing module and the robot is able

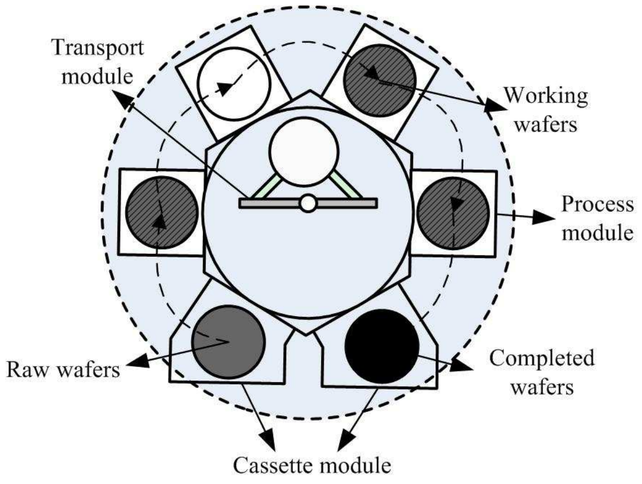


Figure 1: A single-arm cluster tool with 4 process modules

to move only one wafer at a time. There are residency time constraints for processing wafers in process modules. That means the time of each wafer in process modules cannot exceed the upper bounds allowed, i.e. maximal residency times permitted. This forms a processing time window for a wafer in each process. If the residency time constraints were violated, the wafers would become defective. Moreover, there is no buffer between adjacent process modules. As a result, after a wafer has completed its operations in a process module, including processing the wafer and staying at the process module within the residency bound after processing, the wafer must be moved to the next module immediately by the single-armed robot.

For the multi-type wafer case, the objective of scheduling discussed is to minimize the makespan of wafers in the system. The scheduling procedure to improve manufacturing performances of cluster tools can be divided into two stages. The first stage is to determine the sequence of wafers entering the cluster tool for processing, and the second one is to schedule the operation sequence of the transport single-armed robot in the cluster tool. Combinations of the two stages will give the final executable scheduling solutions. To the best of our knowledge, little work has been done on scheduling in cluster tools where the processing sequence and the operation sequence are considered simultaneously.

In this paper, a differential evolution (DE) algorithm is applied to optimize the processing sequence. A try and error-based scheduling algorithm (TE) (Zhou and Li 2012) is used to optimize the operation sequence. Hence integrating the two scheduling stages proposed in the paper differs significantly from past work.

2 Literature Review

In order to complete a wafer as early as possible, it is reasonable to start it as early as possible at the appropriate process modules. Based on this principle, Song et al. (Song, Zabinsky, and Storch 1993) suggest a heuristic algorithm called the earliest starting time heuristic (EST) in single-type job scenarios. Paul et al. (Paul, Bierwirth, and Kopfer 2007) present an approach based on defining time windows related to any activity. These time windows are continuously adjusted, giving an adaptive time window (ATW) heuristic. Hindi and Fleszar (Hindi and Fleszar 2004) propose a heuristic algorithm based on a non-standard constraints satisfaction model with routine ordering, forward checking and backtracking. Kharbeche et al. (Kharbeche et al. 2011) and Carlier et al. (Carlier et al. 2010) propose an approximate decomposition algorithm which breaks and solves the scheduling problem in two stages, i.e. a flow shop problem with additional constraints (blocking and transportation times) and a single machine problem with precedence constraints, time lags, and setup times. In order to solve these two problems, an exact branch-and-bound algorithm and a genetic algorithm are proposed.

Kim and Lee (Kim and Lee 2008) propose an extended Petri net for modeling the scheduling problems and analyze scheduling conflicts with constraints in residency time and resource capacity. Yih (Yih 1994) presents a heuristic algorithm based on trial and error methods, using the residency time of the coming wafer to eliminate resource conflicts. Chauvet et al. (Chauvet, Proth, and Wardi 2001) present a time-window based scheduling algorithm for a no-wait manufacturing system with constraints in residency processing time. However, the authors do not elaborate the time required for the transportation of jobs. Yoon and Lee (Yoon and Lee 2005) develop a real-time scheduling algorithm using a temporal constraint set to form the problem. The scheduling algorithm consists of two procedures. The first one computes feasible operations for newly inserted wafers in every process module, expressed with a temporal constraint set, and the best schedule is built using backward inference. Lee et al. (Lee, Lee, and Lee 2007) schedule a wet station with multiple-part flows and multiple-wafer handling robots using a Petri net model without considering processing time windows.

3 Methodology

3.1 Problem Formulation

The notations used to formulate the scheduling problem are listed as follows.

m: the number of process modules;

PM_i: the process module indexed by i, where $i = 1, 2, ..., m$;

PM_0: the cassette module for input wafers;

PM_{m+1}: the cassette module for output wafers;

n: the number of wafers to be processed;

W_k: the wafer indexed by k, where $k = 1, 2, ..., n$;

$\mathcal{Q}$: a processing sequence of n wafers where $\mathcal{Q} = (q_1, q_2, ..., q_h, ..., q_n)$ and q_h is the h-th wafer enters the cluster tool based on the corresponding sequence;

$f(\mathcal{Q})$: the makespan for a given processing sequence $\mathcal{Q}$ of wafers based on the TE algorithm;

$\mathcal{Q}_{RD}$: a random processing sequence;

$\mathcal{Q}_{DE}$: a processing sequence based on the DE algorithm.

The problem here is to decide a processing sequence of wafers which minimizes the makespan $T(\mathcal{Q})$. It is a permutation problem. To formulate it, 0-1 decision variables $Y_{h,k}$s are defined, i.e.

$$Y_{h,k} = \begin{cases} 1 & \text{if } q_h = k, \\ 0 & \text{otherwise,} \end{cases} \qquad \text{for all } h, k = 1, 2, ..., n.$$

It means that the wafer W_k is the h-th wafer processed in the processing sequence. $Y_{h,k}$ should satisfy the following constraints:

$$\sum_{h=1}^{n} Y_{h,k} = 1 \qquad k = 1, 2, ..., n \tag{1}$$

$$\sum_{k=1}^{n} Y_{h,k} = 1 \qquad h = 1, 2, ..., n \tag{2}$$

Equation (1) shows that the wafer W_k can enter the cluster tool to be processed at only one position in the sequence $\mathcal{Q}$. Equation (2) indicates that there is only one wafer at the h-th position in the sequence $\mathcal{Q}$. Equations (1) and (2) are the necessary and sufficient conditions to generate a feasible processing sequence.

As mentioned above, the TE algorithm is used to deal with the second scheduling stage and obtain the makespan $f(\mathcal{Q})$, under the conditions which the sequence $\mathcal{Q}$ is determined. The objective function of the scheduling problem can be described as follows:

$$\min \quad f(\mathcal{Q}) \tag{3}$$

Therefore, the scheduling problem mentioned above can be expressed as a nonlinear programming problem with (3) as the objective function, equations (1) and (2) as constraints and the TE algorithm is used to adopt the second scheduling stage.

3.2 Canonical Differential Evolution

Differential evolution algorithm is firstly proposed by Storn and Price (Storn and Price 1997) for global optimization over continuous spaces. More details can be referred to Price et al. (Price, Storn, and Lampinen 2005). It is a population based evolutionary type algorithm including initialization, mutation, crossover and selection phases. There are NP D-dimensional vectors $x_{i,g}$, where $i = 1, 2, ..., NP$, as a population (P_g) to form the g-th generation, where $g = 0, 1, 2, ...,$ i.e. $P_g = \{x_{1,g}, x_{2,g}, ..., x_{NP,g}\}$.

NP does not change during the minimization process. $x_{i,g}$ is also named as a target vector. The DE algorithm's basic processes are given in details as follows. During the initialization phase, since there is no initial intelligent information for the system, the initial population (i.e. the 0-th generation) is randomly generated according to a uniform probability distribution in the search region. During the mutation, a mutant vector is generated according to Equation (4) for each target vector $x_{i,g}$, where $i = 1, 2, ..., NP$.

$$v_{i,g+1} = x_{r_1,g} + F(x_{x_2,g} - x_{r_3,g}) \tag{4}$$

r_1, r_2 and r_3 are mutually different random indexes in $\{1, 2, ..., NP\}$. Moreover, the three random indexes are chosen to be different from the running index i. F is a real and constant amplification factor, where $F \in [0, 1]$. During the crossover phase, the trial vectors $u_{i,g+1}$ are obtained according to Equation (5) based on target vectors and mutant vectors, where $u_{i,g+1} = (u_{1,i,g+1}, u_{2,i,g+1}, ..., u_{D,i,g+1})$. $u_{j,i,g+1}$ is the j-th component of the vector $u_{i,g+1}$.

$$u_{j,i,g+1} = \begin{cases} v_{j,i,g+1} & \text{if } FR(j) \leq CR \text{ or } j = IR(i) \\ x_{j,i,g} & \text{if } FR(j) > CR \text{ or } j \neq IR(i) \end{cases}$$
$$\text{where } j = 1, 2, ..., D \tag{5}$$

$v_{j,i,g+1}$ and $x_{j,i,g}$ are the j-th component of the vectors $v_{i,g+1}$ and $x_{i,g}$ respectively. $FR(j)$ is a randomly generated float number for each j where $FR(j) \in (0, 1)$. $IR(i)$ is a randomly generated integer in the set$\{1, 2, ..., D\}$ each time. CR is the crossover constant in $[0, 1]$ determined by the user. The selection phase decides how to obtain each number of the generation $g + 1$. During this phase, each trial vector $u_{i,g+1}$ is compared to the target vector x_i, g using the given evaluation function. If the vector $u_{i,g+1}$ yields a better value, i.e. a smaller cost for minimization problem, then $x_{i,g+1}$ is set to be $u_{i,g+1}$; otherwise, the old value $x_{i,g}$ is retained to the next generation. The evaluation function given depends on special problems.

Finally, the iteration is terminated when the generation achieves the pre-set maximal generation number.

3.3 Transformation Approach

Parameters in the DE algorithm are floating-point numbers. Although DE is simple and effective, it cannot be used for combinatorial optimization problems (COPs). The sequencing scheduling problem researched in this paper is a typical COP. To solve this COP, Nearchou and Omirou's (2006) approach (NOA) is applied in this paper to correspond a floating-point vector to an integer vector which describes a sequence or permutation of the COP. For NOA, each component of the vectors $x_{j,i,g}$, $v_{j,i,g}$ and $u_{j,i,g}$ lie within the interval $[0, 1]$, i.e. $x_{j,i,g}$, $v_{j,i,g}$ and $u_{j,i,g} \in [0, 1]$. Components $z_{j,i,g}$ of an integer vector for sequence $z_{i,g}$ are mutually different integers in the set $\{1, 2, ..., D\}$. The transformation approach from $x_{i,g}$ to $z_{i,g}$ is as follows.

Step 1: The range $[0, 1]$ is divided into D equal sub-ranges. They are ordered as $SR = ([0, \frac{1}{D}), [\frac{1}{D}, \frac{2}{D}), ..., [\frac{D-1}{D}, 1])$. The h-th interval is $[\frac{h-1}{D}, \frac{h}{D})$, where $h = 1, 2, ..., D$.

Step 2: For each component $x_{j,i,g}$ of $x_{i,g}$, if $x_{j,i,g}$ lies in the h-th interval of SR, then, set $z_{j,i,g} = h$.

Step 3: Set the duplicate components of $z_{i,g}$ as 0 and then fill the components equal to 0 by selecting randomly the unused integers in the set $\{1, 2, ..., D\}$.

For example, set $D = 5$ and $x_{i,g} = (0.42, 0.57, 0.23, 0.89, 0.34)$. $SR = ([0, 0.2), [0.2, 0.4), [0.4, 0.6), [0.6, 0.8), [0.8, 1])$. For Step 2, $x_{i,g} = (3, 3, 2, 5, 2)$. In Step 3, $x_{i,g} = (3, 0, 2, 5, 0)$, and integers 1 and 4 are not used now. They are selected randomly as the components equal to 0, i.e., the 2nd and the 4th components for this example. Then, $x_{i,g} = (3, 4, 2, 5, 1)$. Further explanation can be found to Nearchou and Omirou's paper (Nearchou and Omirou 2006).

3.4 Scheduling Algorithm

Using a DE algorithm to solve the sequencing problem mentioned above, let the vector dimension D be n, representing the number of wafers. For this problem, $z_{i,g}$ is the sequence of wafers which has the same means of $\mathcal{Q}$ mentioned above. During the initialization, set $g = 0$ and each component $x_{j,i,0}$ of every vector$x_i, 0$ in P_0 is randomly generated in the interval $[0, 1]$. Each vector $x_{i,0}$ is transformed to $z_{i,0}$ according to NOA and the related makespan is calculated. The minimal makespan f_{best} and the related sequence $\mathcal{Q}_{best}$ are found. Set parameters CR, F and the maximal generation number $MaxG$.

The procedure of the DE algorithm for the sequencing problem is described as follows:

Step 1: Generate each mutant vector $v_{i,g+1}$ for the $(g+1)$-th generation according to Equation (4).

Step 2: Generate each trial vector $u_{i,g+1}$ for the $(g + 1)$-th generation according to Equation (5).

Step 3: (Selection phase)

Step 3.1: Set $i = 1$.

Step 3.2: Transform vectors $x_{i,g}$ and $u_{i,g+1}$ to $z_{i,g}$ and $z_{i,g+1}$ according to NOA respectively.

Step 3.3: If $f(z_{i,g}) < f(z_{i,g+1})$, then set $x_{i,g}$ as $x_{i,g+1}$; otherwise, set $u_{i,g+1}$ as $x_{i,g+1}$.

Step 3.4: Transform each vector $x_{i,g}$ to $z_{i,g}$ according to NOA. If $f(z_{i,g}) < f_{best}$, set $f(z_{i,g})$ as f_{best} and $z_{i,g}$ as $\mathcal{Q}_{best}$.

Step 3.5: Set $i = i + 1$. If $i < D$, go to Step 3.1.

Step 4: Set $g = g + 1$. If $g < MaxG$, go to Step 1.

Step 5: Generate the minimal makespan f_{best} and the related best sequence $\mathcal{Q}_{best}$. Complete the algorithm.

4 Simulation analysis

The performance of the proposed DE algorithm is evaluated through extensive simulation experiments. The objective of the scheduling problem in this paper is to minimize

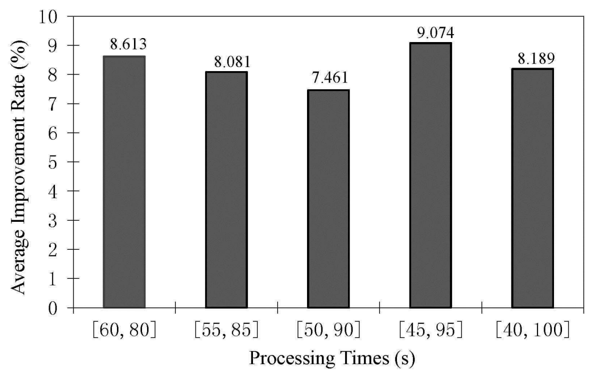


Figure 2: Improvement rates

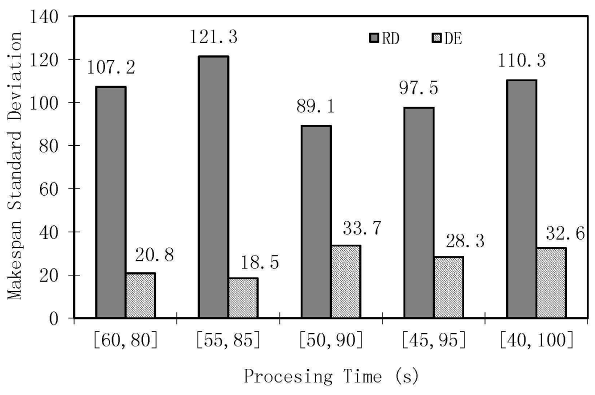


Figure 3: Standard deviations of maksespans

the makespan of a number of wafers, say 25. Makespans of randomly processing sequences are used as reasonable benchmarks for the proposed DE algorithm. The criterion called the improvement rate (R) is defined as

$$R = \frac{f(\mathcal{Q}_{RD}) - f(\mathcal{Q}_{DE})}{f(\mathcal{Q}_{RD}} \times 100\% \tag{6}$$

where $\mathcal{Q}_{RD}$ is a random processing sequence and $\mathcal{Q}_{DE}$ is a processing sequence based on the DE algorithm. As mentioned above, $f(\mathcal{Q})$ is the makespan related to the given sequence $\mathcal{Q}$ for a given instance. It is obtained based on the TE algorithm as the second scheduling algorithm. Another criterion is called makespan standard deviation. To calculate this criterion, a given instance is run 30 times. The 30 results (makespans) form a statistical sample. Then the standard deviation is calculated using statistical formula.

Set the number of wafers $n = 25$, i.e. a lot, and the number of process modules $m = 8$. Processing times and most permitted residency times are randomly generated from a uniform distribution $U[a, b]$ for each case. There are

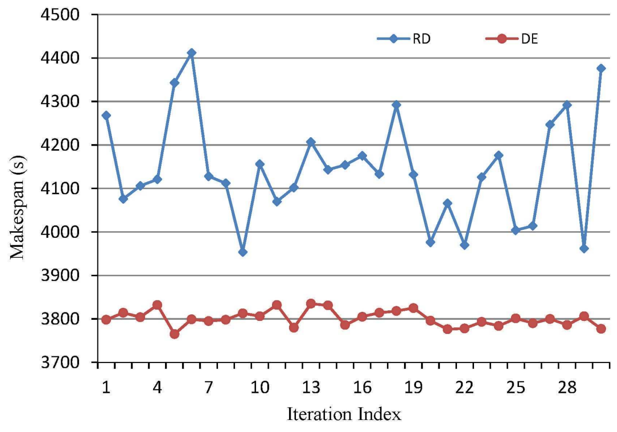


Figure 4: Makespans of a given instance

five examples where processing times are from $U[60, 80]$, $U[55, 85]$, $U[50, 90]$, $U[45, 95]$ and $U[40, 100]$ respectively. Permitted residency times are all randomly generated from $U[40, 60]$. The time unit is second(s). Robot transport time is equal to 9s. For each case, the improvement rate is the mean of 15 instances in each example.

For the DE algorithm, a population of generation $NP = 100$, maximal generation number $MaxG = 1000$ are used. The constant amplification factor $F = 0.5$ and the crossover rate $R = 0.5$.

The simulation program is coded in Visual C++ 6.0 to run on the IBM personal computer with 160GB hard disk, 1GB DDR2 memory and 2.00 GHz Intel Core 2 CPU.

Fig. 2 shows the improvement rates for the 5 cases. The smallest $R = 7.461\%$ and the largest $R = 9.074\%$. The improvement rates are between $R = 7.461\%$ and $R = 9.074\%$. It shows that the DE algorithm is effective to obtain a smaller makespan compared with the case without DE algorithm.

Since DE algorithm is a stochastic scheduling algorithm, it obtains different results (makespan) for each run of the same given instance. The makespan standard deviation provides the stability of the proposed algorithm. When the standard deviation of makespan is small, it means the algorithm always achieves a better scheduling performance in the instance. Fig. 3 shows the results of the 5 cases. For each case, an instance is selected and it is run 30 times during the simulation. The makespan standard deviations from DE algorithm are all far less than those without DE algorithm. It means that the DE algorithm is more stable in all the 5 cases. Fig. 4 shows the makespans for an instance from $U[55, 85]$. It is run 30 times. Each point is related to a makespan for a run, i.e. an iteration index. It shows that DE algorithm gives better stability.

5 Conclusions

In this paper, the DE algorithm is applied to schedule processing sequence of wafers in cluster tools. To evaluate the proposed algorithm, experiment simulation is designed and implemented. The simulation results show that: 1) the wafer processing sequence has a greater impact on scheduling performance in cluster tools. 2) An appropriate algorithm of the first scheduling stage can improve overall efficiency of cluster tools well. The DE algorithm proposed in this paper can effectively improve the system efficiency. With reductions in makespans, more stability can be obtained for diverse cases of processing time.

In future research, the DE algorithm will be tested to understand the relationship between the parameters in cases to improve the performance of the algorithm cases. The DE algorithm will also be compared with other evolution algorithms.

Acknowledgements

The work described in this paper was supported by a Research Grant from General Research Fund (GRF) of Hong Kong (RGC # CityU 113609).

References

Carlier, J.; Haouari, M.; Kharbeche, M.; and Moukrim, A. 2010. An optimization-based heuristic for the robotic cell problem. *European Journal of Operational Research* 202(2):636–645.

Chauvet, F.; Proth, J.-M.; and Wardi, Y. 2001. Scheduling no-wait production with time windows and flexible processing times. *IEEE Transactions on Robotics and Automation* 17(1):60–69.

Hindi, K. S., and Fleszar, K. 2004. A constraint propagation heuristic for the single-hoist, multiple-products scheduling problem. *Computer and Industrial Engineering* 47(1):91–101.

Kharbeche, M.; Carlier, J.; Haouari, M.; and Moukrim, A. 2011. Exact methods for the robotic cell problem. *Flexible Services and Manufacturing Journal* 23(2):242–261.

Kim, J.-H., and Lee, T.-E. 2008. Schedulability analysis of time-constrained cluster tools with bounded time variation by an extended petri-net. *IEEE Transactions on Automation Science and Engineering* 5(3):490–503.

Lee, T.-E.; Lee, H.-Y.; and Lee, S.-J. 2007. Scheduling a wet station for wafer cleaning with multiple job flows and multiple wafer-handling robots. *International Journal of Production Research* 45(3):487–507.

Nearchou, A. C., and Omirou, S. L. 2006. Differential evolution for sequencing and scheduling optimization. *Journal of Heuristics* 12(6):395–411.

Paul, H. J.; Bierwirth, C.; and Kopfer, H. 2007. A heuristic scheduling procedure for multi-item hoist production lines. *International Journal of Production Economics* 105(1):54–69.

Price, K. V.; Storn, R. M.; and Lampinen, J. A. 2005. *Differential Evolution: A Practical Approach to Global Optimization*. Natural Computing Series. Springer Berlin Heidelberg.

Song, W.; Zabinsky, Z. B.; and Storch, R. L. 1993. An algorithm for scheduling a chemical processing tank line. *Production Planning and Control* 4(4):323–332.

Storn, R., and Price, K. 1997. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization* 11(4):341–359.

Yih, Y. 1994. An algorithm for hoist scheduling problems. *International Journal of Production Research* 32(3):501–516.

Yoon, H. J., and Lee, D. Y. 2005. Online scheduling of integrated single-wafer processing tools with temporal constraints. *IEEE Transactions on Semiconductor Manufacturing* 18(3):390–398.

Zant, P. V. 2004. *Microchip Fabrication: A Practical Guide to Semiconductor Processing*. New York: McGraw-Hill, 5th edition.

Zhou, B.-h., and Li, X. 2012. Try and error-based scheduling algorithm for cluster tools of wafer fabrications with residency time constraints. *Journal of Central South University* 19(1):187–192.

Local Search in the Space of Valid Plans

Fazlul Hasan Siddiqui and Patrik Haslum
The Australian National University & NICTA Optimisation Research Group
Canberra, Australia
{firstname.lastname}@anu.edu.au

Abstract

Producing high quality plans and producing them fast are two of the main aims of automated planning. There is, however, a tension between these two goals: plans of good quality tend to be hard to find, and plans found quickly are often of poor quality. Anytime planners try to balance these objectives by producing plans of better quality over time, but current anytime planners often are not effective at making use of increasing runtime beyond the first few minutes. Local search in planning, for example, is a powerful method to quickly find plans, although their quality is usually far from optimal, as the behaviour of the search depends crucially on the topology of the search space. Nevertheless, inspired by its scalability, we hybridize the local search with bounded-cost search to apply in the space of valid plans. To perform successful local search, we make good use of plan structure by decomposing a given plan into meaningful subplans. Each subplan is optimised iteratively, and the resultant improved subplans are combined with each other, wherever possible, into an overall plan. The decomposition exploits block-structured plan deordering to identify coherent subplans resulting in "easy" subproblems for the local optimiser. Repeating the above process successively produces better quality plans, which extends the "anytime capability" of current planners – to provide continuing plan quality improvement at any time scale.

Introduction

Solution quality and solver efficiency are of particular interest in automated planning. Much progress has been made separately on those two targets, but few planners have the flexibility to be used at any point on the efficiency–quality trade-off scale. Planners that are able to find plans quickly (for example, greedy heuristic search based planners), usually find plans of poor quality. In contrast, recent advances in the design of PDDL planners have focused on plan quality as witnessed by the 6th and 7th International Planning Competition (IPC), but the planners that guarantee solution optimality, or bounded sub-optimality, do not scale up to large problems. Therefore, a gap exists between the capabilities of these two classes of planners.

Anytime planners try to strike a balance between (slow) optimal and fast (but non-optimal) planning methods, by finding an initial plan, possibly of poor quality, quickly and then continually finding better plans the more time they are given. But current anytime planners often are not effective at making use of increasing runtime beyond the first few minutes. Examples range from ARA* (Likhachev *et al.* 2003) to LAMA (Richter and Westphal 2010), where the current best approach, LAMA, is based on restarting weighted A* search with a schedule of decreasing weights. However, as the weight used in WA* decreases, it fairly quickly consumes all memory by degenerating into a plain A* search. Therefore, this method does not quite live up to the promise of continually improving plans over time.

Apart from planning, finding high quality solutions in reasonable time is also a key issue in combinatorial optimisation and Operations Research (OR) problems, where local search is extensively used for improving solution quality. The basic idea of local search is to start from an initial solution and to search for successive improvements by examining neighboring solutions. Coming back to planning, local search has also been used successfully in several recent satisficing planners, like FF (Hoffmann and Nebel 2001), LPG (Gerevini and Serina 2002), and Arvand (Nakhost *et al.* 2011). Unlike OR, local search in planning is primarily used to find a solution quickly. However, the neighborhood structure plays a crucial role in the performance of a local search method (Hoffmann 2001). In designing such methods, there is often a compromise between the size of the neighborhood to use and the computational complexity of exploring it. Large Neighborhood Search (LNS) is another local search technique that is designed to explore large promising regions of the search space. LNS treats the problem of finding a good neighbor as an optimisation problem. Theoretical and experimental studies have shown that the increase of the neighborhood size may improve the effectiveness (quality of provided solutions) of local search algorithms (Ahuja *et al.* 2007). Even though local search is incomplete, its scalability makes it superior for many large applications. Because of having complementary strengths, sometimes, local search is coupled with systematic search in combinatorial problems to provide better quality solutions.

In summary, our challenge is to design a planning system that can continue to improve the plan quality even when the current best planners stop improving. We are inspired by the power of local search, in particular, the large neighborhood search for anytime solution quality improvement, and aim to apply this in domain independent planning. The main challenge in doing this is identifying good subplans to re-optimise, and in planning, we need to do that by automatic and domain independent methods. For this, we make use of plan deordering. Standard plan deordering, however, requires unordered plan steps to be non-interfering. This limits its applicability, so that in many cases no deordering of a sequential plan is possible. To overcome this limitation, we have proposed block decomposition of plans (Siddiqui and Haslum 2012), which is a new form of plan decomposition that allows two groups of plan steps to be unordered even when their constituent steps cannot. This increased deordering of the plan makes our strategy for finding subplans more effective.

After the block decomposition, we apply a windowing heuristic over the block decomposed plan to formulate a set of windows (comprising one or more subplans, with a possible reordering of the sequential plan) for local optimisation separately using bounded-cost search. The improved subplans are recomposed into a complete plan with multiple such recompositions made if possible, resulting a new better valid plan. The process is then repeated, starting from the new best plan. This results in a local search in the space of valid plans, and we move from one valid plan to the next by replacing one or more subplans with the improved subplans. Since every successive plan is of better quality, the local search performs a hill climbing. The possibility of local optima in this case is generally low due to the large number of neighbors in the search space.

Figure 1 shows the compound anytime profile of four different planners. The plan quality improvement (as measured by the summed IPC quality score) by LAMA rises very sharply in the beginning, meaning that a small amount of additional time at this stage leads to a big improvement (mainly due to more problems solved), but then quickly flattens out. From 30 minutes to the time limit (1 hour CPU time), the improvement is less than 1%. Yet, as the figure also shows, it is possible to attain a considerably greater improvement by switching to a different method at this point.

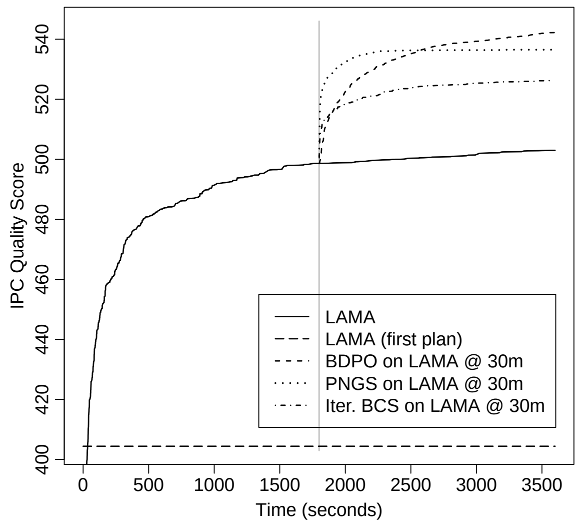


Figure 1: Total IPC quality score as a function of time for LAMA and three plan optimisation methods, on all problems used in our experiments (606 problems, of which LAMA solves 577). The score of a plan is calculated as c^{ref}/c, where c is the cost of the plan and c^{ref} the "reference cost" (best cost of all plans for the problem). Thus, a higher score reflects a lower-cost plan. *Note that the y-axis is truncated.*

Related Work

A good number of methodologies have been developed so far for plan quality improvement, most of which can be broadly classified into three categories as discussed below.

Local Search and Anytime Planning

Local search is a powerful method to address domain independent planning and has been used successfully in several satisficing planners.

LPG (Gerevini and Serina 2002), for example, uses a local search in the space of action graphs, which are particular subgraphs of the planning graph, representing partial plans (Blum and Furst 1995). The aim of using local search is to quickly find a plan. The search steps of LPG are certain graph modifications transforming an action graph into another one (corresponding to particular revisions of the partial plan). If no plan is found after a certain number of search steps, a restart is performed. It also performs a restart when a plan has been found. In this way, a sequence of plans are produced, and only the better quality plans are stored. Therefore, it behaves like an anytime process. FF (Hoffmann and Nebel 2001) is a forward chaining state space planner, guided by a heuristic that estimates goal distances by ignoring delete lists. It uses an enforced form of hill climbing, combining local and systematic search. FF also employs a pruning technique that selects a set of promising successors to each search node, and another pruning technique that cuts off branches where it appears that some goal has been achieved too early. Arvand (Nakhost *et al.* 2011) is a new addition in this category, which is built upon the FF heuristic. It uses a forward-chaining local search, and at each search step the next state is chosen from a set of random samples obtained by random walks. In comparison to the above methods, we use local search in the space of valid plans rather than the space of invalid or partial plans. They take small step in the local search having smaller neighborhood, whereas our system uses a large neighborhood search.

Anytime planning, as mentioned before, is particularly appealing when time is limited. This is because an anytime algorithm can quickly produce an initial plan, then continue to return a stream of better plans. The LAMA planner, for example, uses weighted A* with a large initial value of the weight to quickly produce an initial plan, then gradually reduces the weight, while using the cost of the best found plan for additional pruning. However, it quickly reaches a limit where its ability to find further improvements stops. Another anytime planner, Lamar (Olsen and Bryce 2011), is built upon LAMA with the addition of randomizing planning graph construction so that heuristic plateaus are less frequent.

Post-Processing for Plan Improvement

There are a good number of methods for plan quality improvement working as post-processing tools. Nakhost and Müller (2010) constructs a plan neighborhood graph (termed as PNGS), which is a subgraph of the state space of the problem, limited to a fixed distance d from states traversed by the current plan. They then search for the cheapest plan in this subgraph. If it improves on the current plan, the process is repeated around the new best plan; otherwise, the distance is increased. ITSA* (Furcy 2006) similarly explores an area of the state space near the current plan. Compared to our method these methods can be seen as using a different neighborhood, that includes only small deviations from the current plan, but anywhere along the plan. In contrast, we focus on a small section of the plan at a time, but put no restriction on how much the replacement subplan differs from the original plan section. As we will show, our method and PNG search have somewhat complementary strengths. Thus, a local search over both types of neighborhoods might be even more effective.

Ratner and Pohl (1986) use local optimisation for shortening solutions to sequential search problems. However, their approach to subproblem identification is a simple sliding window over consecutive segments of the current path. This is unlikely to find relevant subproblems for optimising plan cost in general planning problems, where the sequential plan is often an arbitrary interleaving of separate causal threads. A large majority (at least 75%) of the subproblems for which we find an improved subplan correspond to non-consecutive parts of the original plan. Likewise, Balyo, Barták and Surynek (2012) use a sliding window to minimise parallel plan length (that is, “makespan”, assuming all actions have unit duration).

The planning-by-rewriting approach (Ambite and Knoblock 2001) also uses local modifications of partially ordered plans to improve their quality. Plan modifications are defined by (domain-specific) rewrite rules, which have to be provided by the domain designer or learned from many examples of both good and bad plans. Apart from the issue of domain dependency, this technique is effective while solving many problem instances from the same domain.

Using instead a planner to solve the local improvement subproblem may be more time-consuming than applying pre-defined rules, but makes the process fully automatic. However, if we consider solving many problems from the same domain it may be possible to reduce average planning time by learning (generalised) rules from the local plan improvements we discover and using these where applicable to avoid calling the planner.

Hybridizing Planning with other Techniques

The cross fertilization of planning community with techniques from other fields like evolutionary algorithms is still quite rare. According to Schoenauer et al. (2008), Evolutionary Algorithms (EAs) can rarely efficiently solve combinatorial optimization problems on their own, i.e., without being hybridized, one way or another. The most successful of such hybridizations use Operational Research (OR) methods to locally improve any offspring that was born from EA variation operators (crossover and mutation): such algorithms have been termed “Memetic Algorithms” or “Genetic Local Search” (Merz and Freisleben 1999). Standard Memetic Algorithms use local search methods to improve the evolutionary solutions, and thus fail when the local method stops working on the complete problem. The Divide-and-Evolve approach (Bibaï *et al.* 2010) divides a complex planning task into (hopefully) easier subtasks using a state-based decomposition strategy. A satisficing planner is used to solve each subtask driven by an evolutionary algorithm to conduct the optimization process, i.e., to find solutions of good quality.

LNS, as discussed before, has been used very successfully in constraint-based approaches to hard combinatorial optimisation problems like vehicle routing with time windows (Shaw 1998) and scheduling (Godard *et al.* 2005). In this setting, a local optimisation step solves the constraint problem, using an expensive but highly optimising method, for a small set of variables, keeping remaining variables fixed at their current values. Our plan improvement approach can be viewed as applying LNS to planning. The main challenge in applying LNS is identifying good subproblems to re-optimise. Routing and scheduling solvers using LNS rely on problem-specific heuristics, based on insights into the problem and its constraint formulation, for this. For planning, we need automatic and domain-independent methods: that is what our decomposition strategy provides. Our local search is a plain hill-climbing search, always moving to a better (but not necessarily best) plan in the neighbourhood. It might be improved through use of more sophisticated metaheuristics, such as simulated annealing or restarts.

Partial-Order and Block-Deordered Plans

We use the standard STRIPS model of classical planning problems. We briefly recap some basic notions concerning partially ordered plans. For a detailed introduction, see, for example, the book by Ghallab et al. (2004).

Partially Ordered Plans

A *partially ordered plan* (POP) is a tuple $(S, \prec)$, where S is the set of plan steps and $\prec$ is a strict partial order on S; $\prec^+$ denotes the transitive closure of $\prec$. A *linearisation* of a POP is a strict total order that contains $\prec$. Each step $s \in S$, except for the initial and goal steps, is labelled by an action, $\mathrm{act}(s)$, which, as usual, has precondition (or consumed), added (or produced) and deleted (or threatened) sets of propositions. With slight abuse of terminology, we talk about the preconditions and effects of a step, meaning those of its associated action. A *causal link*, (s_i, p, s_j), records a commitment that the precondition p of step s_j is supplied by an add effect of step s_i. The link is *threatened* if there is a step s_k that deletes p such that $s_i \prec s_k \prec s_j$ is consistent. The validity of a POP can be defined in two equivalent ways: (1) a POP is valid iff every linearisation of its actions is a valid sequential plan, under the usual STRIPS execution semantics; and (2) a POP is valid if every step precondition (including the goals) is supported by an unthreatened causal link. (That is, essentially, the modal truth condition (Chapman 1987).)

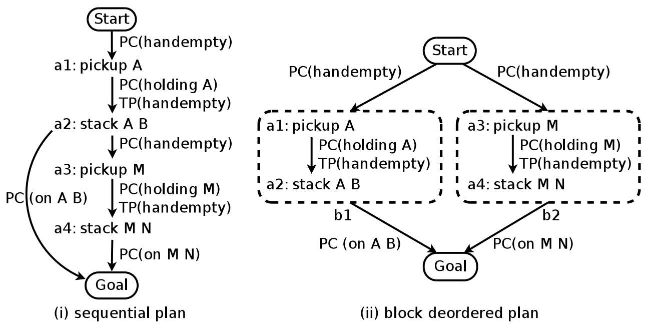


Figure 2: A sequential plan and its block deordering. Precedences are labelled with their reasons: producer–consumer (i.e., a causal link), denoted $\mathrm{PC}(p)$; threat–producer, denoted $\mathrm{TP}(p)$; and consumer–threat, denoted $\mathrm{CT}(p)$.

Block Semantics and Block Decomposition

A *block* is a part of the plan, i.e., a subset of plan steps, that must not be interleaved with steps not in the block. Unordered blocks can be executed in any order, and steps within a block may also be partially ordered.

Definition 1. *Let $(S, \prec)$ be a partially ordered plan. A* block w.r.t. $\prec$ *is a subset $b \subset S$ of steps such for any two steps $s, s' \in b$, there exists no step $s'' \in (S - b)$ such that $s \prec^+ s'' \prec^+ s'$ or $s' \prec^+ s'' \prec^+ s$.*

Blocks behave much like (non-sequential) macro actions, having preconditions, add and delete effects that can be a subset of the union of those of its constituent steps. This enables blocks to encapsulate some plan effects and preconditions, reducing interference and thus allowing more deordering. The following definition captures those preconditions and effects that are visible from outside the block, i.e., those that give rise to dependencies or interference with other parts of the plan. These are what we need to consider when deciding if two blocks can be unordered.

Definition 2. *Let $(S, \prec)$ be a partially ordered plan, where S is a set of plan steps, and b be a block. The block semantics are defined as:*

- *b adds m iff b does not have precondition m, and there is a responsible step $\hat{s} \in b$ with $m \in add(\hat{s})$, such that there is no step $s' \in b$ with $s' \not\prec \hat{s}$ that deletes m.*
- *b has the precondition m iff there is a responsible step $\hat{s} \in b$ with $m \in pre(\hat{s})$, and there is no step $s' \in b$ with an unthreatened causal link $(s', m, \hat{s})$.*
- *b deletes m iff there is a responsible step $\hat{s} \in b$ with $m \in del(\hat{s})$, and there is no step $s' \in b$ with $\hat{s} \prec s'$ that adds m.*

A *decomposition* of a plan into blocks can be recursive, i.e., a block can be wholly contained in another. However, blocks cannot be partially overlapping. The semantics of a partially ordered block decomposed plan are defined by restricting its linearisations (for which it must be valid) to those that respect the block decomposition, i.e., that do not interleave steps from disjoint blocks. Validity of such plans can be established in the same way as for POPs, by supporting each precondition of each block with an unthreated causal link.

Block Deordering

Deordering converts a sequential plan into a partially ordered plan (POP), but the conventional deordering approach restricts the deordering to only the cases where individual steps are independent and non-interfering. *Block deordering* eliminates that restriction by forming new blocks, removing ordering constraints, and possibly also adding some explicit ordering constraints that were transitively implied by the removed constraints. It enables deordering in many cases where no deordering is possible under the standard interpretation. This maximized deordering help us to cleary exhibit the plan structure.

We have proposed a block deordering procedure (Siddiqui and Haslum 2012) for automatically finding a block decomposition of a plan that maximises deordering of a partially ordered plan. This procedure works in a check-and-remove fashion: First it checks the reasons behind every necessary ordering $s_i \prec s_j$ within the current plan structure, and forms two blocks b_i and b_j with the initial element s_i and s_j respectively. Then the blocks gradually expand in opposite directions picking one after another steps from the plan structure until those reasons (and newly added reasons) behind the ordering no longer exist (due to the encapsulation within the blocks) or the expansion has reached the boundary. If no reason is left at the end, the ordering is removed as well, and this process is repeated until all the necessary orderings have been checked or the allotted time is up. As a simple example, Figure 2(i) shows a sequential plan for a small Blocksworld problem. This plan can not be deordered into a convential POP, because each plan step has a reason to be ordered after the previous. Block deordering, however, is able to break the ordering (a2 $\prec$ a3) by removing the only reason producer-consumer(handempty) based on the formation of two blocks b1 and b2 as shown in (ii). Neither of the two blocks delete or add proposition handempty (though it is a precondition of both). This removes the interference between them, and allows the two blocks to be executed in any order but without any interleaving. Therefore, the possible linearisations of the block decomposed partially ordered plan are only (a1, a2, a3, a4) and (a3, a4, a1, a2).

Our interest in block decomposition is to use it as a basis for finding subplans suitable for local optimisation. However, block deordering tends to produce blocks that localise interactions as much as possible, i.e., that are as "self-contained" as they can be, and this is useful also for local plan optimisation. The block deordering algorithm returns not just the decomposed and deordered plan, but also a justification for its correctness, by labelling ordering constraints with their reasons (causal links or threats). We exploit this when merging improvements to separate parts of a plan.

Local Optimisation

The plan optimisation algorithm (Algorithm 1) consists of three main steps, where the second and third steps are iterated within a loop. First, block deordering is applied to the

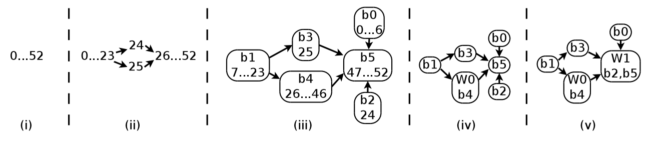


Figure 3: Illustration of the plan optimisation process: The sequential input plan (i), with cost 16, is converted into a standard POP (ii), and then a block deordered plan (iii). The first improvement (iv) replaces block 4, reducing the cost to 14. The next (v) replaces the window consisting of blocks 2 and 5, reducing the cost to 13.

input plan (π_{in}), producing a block decomposition that minimises inter-block dependencies.

Second, a set of candidate subplans, termed "windows", for optimisation, are extracted from the block deordered plan. There are a number of window formation rules, which are divided into three groups, and in each iteration the windows are extracted based on the rules from one group only. Each of those extracted windows generates a bounded-cost subproblem, which is the problem of finding a cheaper replacement for that subplan. Third, the algorithm calls a planner on each of these subproblems, with a cost bound equal to the cost of the current subplan, and a time-out. In principle, the subplanner can be any planning method that accepts a bound on plan cost; we use an iterative bounded-cost search. Whenever a better replacement subplan is found, all replacement subplans found so far (C_{sp}) are fed into a merge procedure, which tries to combine several of them to achieve a greater overall improvement. (At least one replacement subplan can always be merged, so if C_{sp} is non-empty, π_{last} is better than π_{in}.) If, at the end of the inner loop, any improvement has been found, the procedure starts over with the new best plan (π_{last}). In the recursive invocation, window formation will start with the group of rules that has been successful in the last iteration. If, on the other hand, no improvement has been found, the outer loop starts over with the window formation based on the next group of rules. If all the groups of rules have been applied and yet no improvement has been found, the time-out is doubled and the outer loop starts over from the third step, where the subplanner is tried again on each subproblem from the next group, except those known to be solved optimally already. This can be detected by comparing the subplan cost with a lower bound (obtained from an admissible heuristic, h), or by the subplanner proving the bounded-cost subproblem unsolvable. The initial time bound is 15 seconds, or $t_{\text{limit}}/|W_1 \cup W_2 \cup W_3|$, whichever is smaller. Figure 3 illustrates the process on an example problem (#2-5, data set 2-nd, from the Genome Edit Distance domain).

Window Formation

Before extracting windows, we extend blocks to cover complete non-branching subsequences of the plan. That is, if a block b_i is the only immediate predecessor of block b_j, and b_j the only immediate successor of b_i, they are merged into one block. (Note that when we talk about blocks here and in the following, these can also consist of single actions.)

Algorithm 1 Block Decomposition Plan Optimisation

```
procedure BDPO($\Gamma, \pi_{\text{in}}, t_{\text{limit}}, g$)
    Initialise $t_{\text{elapsed}} = 0$, $C_{\text{sp}} = \emptyset$, $\pi_{\text{last}} = \pi_{\text{in}}$.
    $\pi_{\text{bdp}} = \text{BLOCKDEORDER}(\pi_{\text{in}})$.
    Set time bound $t_b$ = initial time bound.
    while $C_{\text{sp}} = \emptyset$ do
        if $W_g$ not initialised then
            Set $W_g = \text{FORMWINDOWS}(\pi_{\text{bdp}}, g)$.
        for each window $(p_i, w_i, s_i) \in W_g$ do
            if $t_{\text{elapsed}} \geq t_{\text{limit}}$ then return $\pi_{\text{last}}$.
            $\Gamma^i_{\text{sub}} = \text{SUBPROBLEM}(p_i, w_i, s_i)$.
            if $h(\pi^i_{\text{sub}}) = \text{cost}(w_i)$ then
                Set $W_g = W_g \setminus \{(p_i, w_i, s_i)\}$.
                continue
            $\pi^i_{\text{sub}} = \text{BOUNDEDCOSTPLANNER}$
                    $(\Gamma^i_{\text{sub}}, \text{cost}(w_i), t_b)$.
            if $\Gamma^i_{\text{sub}}$ proven unsolvable then
                Set $W_g = W_g \setminus \{(p_i, w_i, s_i)\}$.
            else if $\pi^i_{\text{sub}} \neq$ null then
                        /* $cost(\pi^i_{sub}) \leq cost(w_i)$ */
                $C_{\text{sp}} = C_{\text{sp}} \cup \{(w_i, \pi^i_{\text{sub}})\}$.
                $\hat{\pi}_{\text{bdp}} = \text{MERGE}(C_{\text{sp}}, \pi_{\text{bdp}})$.
                if $\text{cost}(\hat{\pi}_{\text{bdp}}) < \text{cost}(\pi_{\text{last}})$ then
                    $\pi_{\text{last}} = \text{SEQUENCEPLAN}(\hat{\pi}_{\text{bdp}})$.
        if $\pi_{\text{last}} \neq \pi_{\text{in}}$ then
            return BDPO $(\Gamma, \pi_{\text{last}}, t_{\text{limit}} - t_{\text{elapsed}}, g)$.
        Set $g = (g + 1) \bmod 3$   /* next group */
        if all groups have been tried at $t_b$ then
            if $W_1 = W_2 = W_3 = \emptyset$ then return null
            Set $t_b = 2 * t_b$.
```

A window is a triple (p, w, s), where w is the set of blocks in the subplan (to be replaced), and p and s are sets of blocks to be placed before and after w, respectively. Any block that is ordered before (resp. after) a block in w must be assigned to p (resp. s), but for blocks that are not ordered w.r.t. any block in w we have a choice of assigning them to either p or s. Let $\text{Un}(b)$ be the set of blocks not ordered w.r.t. b, $\text{IP}(b)$ its immediate predecessors of b, and $\text{IS}(b)$ its immediate successors. For each block b, we generate a set of windows according to the following rules, which are categorized into three groups:

First

$w \leftarrow \{b\}, p \leftarrow \text{Un}(b)$;
$w \leftarrow \{b\}, s \leftarrow \text{Un}(b)$;
$w \leftarrow \{b\} \cup \text{IP}(b), s \leftarrow \text{Un}(b)$;
$w \leftarrow \{b\} \cup \text{IS}(b), p \leftarrow \text{Un}(b)$;
$w \leftarrow \{b\} \cup \text{Un}(b)$.

Second

$w \leftarrow \{b\} \cup \text{Un}(b) \cup \text{IP}(b)$;
$w \leftarrow \{b\} \cup \text{Un}(b) \cup \text{IS}(b)$;
$w \leftarrow \{b\} \cup \text{Un}(b) \cup \text{IP}(b) \cup \text{IS}(b)$.

Third

$w \leftarrow \{b\} \cup \text{Un}(b) \cup \text{IP}(\{b\} \cup \text{Un}(b))$;

$$w \leftarrow \{b\} \cup \mathrm{Un}(b) \cup \mathrm{IS}(\{b\} \cup \mathrm{Un}(b));$$
$$w \leftarrow \{b\} \cup \mathrm{Un}(b) \cup \mathrm{IP}(\{b\} \cup \mathrm{Un}(b)) \cup \mathrm{IS}(\{b\} \cup \mathrm{Un}(b)).$$

Applied to all blocks, these rules can produce duplicates; of course, only unique windows are kept. Each group of rules generally produces a set of windows with smaller variations of window sizes (measured by the number of actions in w), while the higher indexed groups of rules produce windows of bigger sizes compared to that of the lower indexed groups in general. In every iteration, the algorithm exploits only one group of window formation rules, rather than all the rules from different groups. Those windows are eventually processed by BDPO in order of their increasing size. If, at the end of the loop, at least one window from that group has been improved, then the complete BDPO process is restarted with the new best plan. In that case, the next window formation will be based on the rules from the last successful group. Otherwise, if there is no improvement, the process continues from formulating the next group of windows.

Grouping the window formation rules is useful mainly for two reasons: First, applying all the rules at the same time may produce a large number of windows with possibly a lot of overlap among them. Hence, most of the optimised windows will be discarded while recomposing into a single plan. Furthermore, attempting to improve the windows that contain one or more sub-windows that have been improved earlier may not be successful quite often, which leads to a waste of CPU time. Second, improvement of windows based on a group of rules renders a heuristic for selecting the group of rules for window formation in the next iteration, rather than restarting from a lower indexed group. However, too much grouping is not worthwhile either, since it forces restarting the whole procedure after only a few improvements. As a result, it does not properly make use of the block decomposed plan space. Moreover, it increases the possibility of running out of the allotted time (30 minutes in our case) without exploring the bigger-sized windows where an improvement could be possible. The current grouping of the rules is a compromise which we found working reasonably well in a broad range of domains.

Subproblem Construction and Subplanner

Each window (p, w, s) gives rise to a bounded-cost subproblem. This is the problem of finding a replacement for the part w of the plan, of cost less than $\mathrm{cost}(w)$, that can be substituted between plan parts p and s. The subproblem differs from the original problem only in its initial state and goal.

To find the initial state for the subproblem, we generate a linearisation, $a_{p_1}, \ldots, a_{p_k}$, of the actions in p, and progress the original initial state through this sequence, i.e., apply the actions in this sequence. To find the goal, we pick a linearisation, $a_{s_1}, \ldots, a_{s_m}$, of the actions in s, and regress the original goal backwards through this sequence. This ensures that for any plan $a'_1, \ldots, a'_n$ returned by the subplanner, the concatenation of the three sequences, i.e., $a_{p_1}, \ldots, a_{p_k}$, $a'_1, \ldots, a'_n, a_{s_1}, \ldots, a_{s_m}$, is a plan for the original problem.

The subplanner must return a plan of cost less than the given bound, $\mathrm{cost}(w)$. We use a simple bounded-cost greedy search, guided by the (unit-cost) FF heuristic and using an f-value based on the admissible LM-Cut heuristic (Helmert and Domshlak 2009) to prune states that cannot lead to a plan within the cost bound. It is implemented in the Fast Downward planner. The search is complete: if there is no plan within the cost bound, it will prove this by exhausting the search space, given sufficient time. Because bounded-cost search can return any plan that is within the cost bound, we iterate it: whenever a plan is found, as long as time remains, the search is restarted with the bound set to be strictly less than the cost of the new plan. This ensures we get not just an improved subplan, but the best improved subplan that the search can find within the given time limit. Also, while attempting to improve a subplan which is the strict superset of a subplan that has been improved earlier, the cost bound is set so that the improvement of the bigger window must be greater than what has already been achieved for the subwindow. This avoids finding the same improvement twice.

Constructing the subproblem from a linearisation of the block deordered plan leads to an initial state with more facts and a goal with fewer, which simplifies solving the subproblem. However, it also complicates the merging of separate improved subplans (described in the next subsection). As an alternative, we could take a "least committment" approach, taking as initially true only those facts that hold after any linearisation of p, and as goal all facts that must hold for every linearisation of s to succeed. (This is the same principle used to compute the preconditions and effects of a block, since steps in the block may be partially ordered.) However, this severely restricts the choices available to the subplanner, and thus reduces the chances of finding an improvement.

Algorithm 2 Merging Candidate Subplans

```
1: procedure MERGE($C_{\text{sp}}$, $\pi_{\text{bdp}}$)
2:     Initialise $\hat{\pi}_{\text{bdp}} = \pi_{\text{bdp}}$.
3:     Sort $C_{\text{sp}}$ by decreasing $(\mathrm{cost}(w_i) - \mathrm{cost}(\pi^i_{\text{sub}}))$.
4:     for each $(w_i, \pi^i_{\text{sub}}) \in C_{\text{sp}}$ in order do
5:         $\pi_{\text{temp}} = \text{REPLACEIFPOSS}(w_i, \pi^i_{\text{sub}}, \hat{\pi}_{\text{bdp}})$.
6:         if $\pi_{\text{temp}} \neq$ null then
7:             $\hat{\pi}_{\text{bdp}} = \pi_{\text{temp}}$
8:             $C_{\text{sp}} = C_{\text{sp}} \setminus \{(w_j, \pi^j_{\text{sub}}) \in C_{\text{sp}} \mid$
                       $w_j$ overlaps with $w_i\}$.
9:     return $\hat{\pi}_{\text{bdp}}$
```

Merging Improved Subplans

If an improved subplan for a window is found, replacing the window (i.e., the w part) of the original plan with the new subplan is always possible, by construction of the subproblem. Obviously, we gain a greater improvement if we are able to make several such replacements simultaneously. This is what subplan merging tries to do.

The set of candidate replacements (windows with improved subplans) is kept in C_{sp}. Merging all candidates is usually not possible, since windows may overlap. Merging is further complicated by the fact that each subproblem is constructed from a, potentially different, linearisation: Because of this, the replcement subplan may have additional preconditions or delete effects that the replaced window did not, or lack some of its add effects. For a single candidate, these "flaws" can always be resolved by adding more ordering constraints on the plan, but different candidates may require contradictory orderings.

Merging is done by a greedy procedure (Algorithm 2). It takes a current, block deordered, plan (π_{bdp}), and a set of candidates (C_{sp}), and sorts the candidates in decreasing order of their contribution to decreasing plan cost, i.e., the cost of the replaced plan part minus the cost of the new subplan. Each candidate is considered in turn, in this order: If the replacement is still possible, it is made, and any remaining candidates that overlap with the window are removed from further consideration. Since the first replacement is always possible, at least one most improving candidate is merged, so the new plan returned by merging has lower cost than the input plan.

MERGE maintains at all times a valid block deordered plan ($\hat{\pi}_{\text{bdp}}$), meaning that each precondition of each block (and each goal) is supported by an unthreatened causal link. Initially, this is the input plan, for which causal links (and additional ordering constraints) are computed by block deordering. The REPLACEIFPOSS subroutine takes the current plan, and returns an updated plan (which becomes the current plan), or failure if the replacement is not possible. Recall that preconditions and effects of a block are computed using least-committment semantics. This is done for both the replaced window (w_i) and the replacement subplan (π^i_{sub}), where the subplan is treated as a single block whose actions are totally ordered. For any atom in $\text{pre}(\pi^i_{\text{sub}})$ that is also in $\text{pre}(w_i)$, the existing causal link is kept; likewise, causal links from an effect in $\text{add}(w_i)$ that are are also in $\text{add}(\pi^i_{\text{sub}})$ are kept. (These links are unthreatened and consistent with the order, since the plan is valid before the replacement.) For each additional precondition of the new subplan ($p \in (\text{pre}(\pi^i_{\text{sub}}) \setminus \text{pre}(w_i))$), a new causal link must be found, and likewise for each precondition of a later block (or the goal) that was supplied by w_i but is missing from $\text{add}(\pi^i_{\text{sub}})$. Finally, π^i_{sub} may threaten some existing causal links that w_i did not; for each of these preconditions, we also try to find a new link.

The subroutine FINDCAUSALLINK takes the current block deordered plan ($\hat{\pi}_{\text{bdp}}$), the consumer block (b), which can also be the goal, and the atom (p) that the consumer requires, and performs a limited search for an unthreatened causal link to supply it. Specifically, it tries the following two options:

1. If there is a block $b' \prec^+ b$ with $p \in \text{add}(b')$, and for every threatening block (i.e., b'' with $p \in \text{del}(b'')$), either $b'' \prec b'$ or $b \prec b''$ can be added to the existing plan ordering without contradiction, then b' is chosen, and the ordering constraints necessary to resolve the threats added.

2. Otherwise, if there is a block b' with $p \in \text{add}(b')$ that is unordered w.r.t. b, and for every threatening block either $b'' \prec b'$ or $b \prec b''$ can be enforced, then b' is chosen, and the causal link and threat resolution ordering constraints added.

If these two steps cannot find one of the required causal links, the replacement fails, and the candidate is skipped by merge.

Note that some of the ordering constraints between w_i and the rest of the plan may become unnecessary when w_i is replaced with π^i_{sub}, because π^i_{sub} may not delete every atom that w_i deletes and may not have all preconditions of w_i. Even if an ordering $b \prec w_i$, for some block b, is not required after replacing w_i with π^i_{sub}, removing it may make π^i_{sub} unordered w.r.t. blocks $b' \prec b$. Each of these must be checked for potential threats, either due to atoms deleted by π^i_{sub} or by b', and new ordering constraints $b' \prec \pi^i_{\text{sub}}$ added where needed. In the same way, if an ordering $\pi^i_{\text{sub}} \prec b$ is removed, potential threats with blocks $b' \succ b$ must be checked.

Theorem 1. *If the input plan, π_{bdp} is valid, then so is the plan returned by* MERGE.

Proof sketch. This is shown by induction on the sequence of accepted replacements. Initially, the current plan is the valid input plan. It changes only when a replacement is made. A successful replacement does not invalidate the plan: All necessary causal links to and from the replacement subplan are established by REPLACEIFPOSS. Likewise, any causal links threatened by the new subplan are re-established. The remaining possibility, that the new subplan becomes unordered w.r.t. a block b' that w_i was not, and therefore π^i_{sub} threatens a causal link that w_i did not, or a causal link to or from π^i_{sub} is threatened by b', is explicitly checked for before removing any ordering constraint. □

Results

In our experiments, we have used problems from the satisficing track of the 2008 and 2011 IPC[1], data set 2-nd of the Genome Edit Distance (GED) domain (Haslum 2011), and the Alarm Processing for Power Networks (APPN) domain (Haslum and Grastien 2011). The starting point for plan improvement is the best plan found by LAMA (Richter and Westphal 2010, IPC 2011 version) in 30 minutes (called "LAMA@30" in the following). The compared methods are block decomposition plan optimisation (BDPO), plan neighbourhood graph search (PNGS), iterated bounded-cost search (IBCS) and letting LAMA continue (LAMA@60). Each is given 30 minutes and 3Gb memory per problem. IBCS uses the same bounded-cost search as local plan optimisation, applied to the whole problem.

Figure 1 shows the cumulative IPC quality score over time. Most of the plan improvement achieved by LAMA is done early: During the first 30 minutes, it reduces plan cost, from that of the first plan it finds, by an average 17.9%, but the reduction over the next 30 minutes is only 0.8%. In contrast, BDPO, starting from the plan by LAMA@30, achieves an average plan cost reduction of 8%, and PNGS 7.3%. PNGS is also fast (99.6% of its improved plans are found in the first 5 minutes), and limited mainly by memory (it runs out of memory on 86% of problems, and out of time

[1]We exclude the CyberSec domain, which our current implementation is unable to deal with. For domains that appeared in both the 2008 and 2011 IPC, we use only the instances from 2011 that were new in that year.

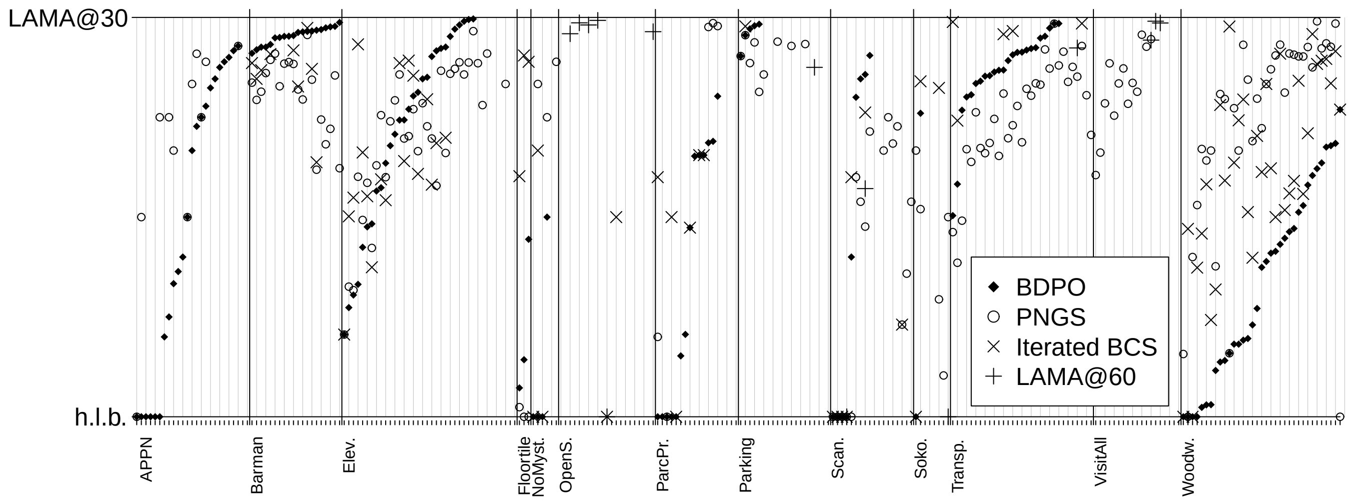


Figure 4: Cost of plans found by improvement methods applied to LAMA's best plan at 30 minutes (one problem per column) on IPC 2008 and 2011 problems. Costs are normalised to the interval between the initial plan and the highest known lower bound.

on 8.5%), while BDPO is limited mainly by time and continues to find better plans (though at a decreasing rate) even beyond 30 minutes.

Although Figure 1 depicts a convenient summary, it does not convey the complete picture: it is strongly weighted by coverage (the score taking only the first plan found by LAMA for each problem is 404.77), and, more importantly, does not tell us anything about how much plan quality can actually be improved. To this end, we compare plan costs with the highest known lower bound for each problem (obtained by a variety of methods, including several optimal planners; cf. (Haslum 2012)). For 163 out of 576 problems solved by LAMA@30, the best plan cost already matches the lower bound, which means no improvement is possible. Figure 4 shows for the remaining problems (excluding the Tidybot domain and one problem in the PegSol domain) the cost of the best plan found by the plan optimisation methods, normalised to the gap between the initial plan cost and the highest lower bound. The results clearly show strong complementarity between the methods, both across domains and in time. Only IBCS does not outperform all other methods in any domain. In the PegSol domain, all plans but one found by LAMA@30 are optimal and no method improves on the cost of the last one. In OpenStacks and VisitAll, PNGS and BDPO find very few improvements, while LAMA finds more but smaller.

Figure 5 separately depicts the plan costs for 156 problems from the GED domain, including also results by a non-optimal problem-specific algorithm (GRIMM). [2] GRIMM finds better solutions than all planners in most cases. However, BDPO and PNGS both find a better plan than GRIMM for 12 out of 156 problems, while LAMA@60 manages 6. BDPO is the best performing anytime planner in this domain finding better plans than LAMA, PNGS, and IBCS for 108,

	BDPO			PNGS			IBCS			LAMA @60		
	$=$	$<$	$\star$	$=$	$<$	$\star$	$=$	$<$	$\star$	$=$	$<$	$\star$
APPN	100	76	24	24	0	4	8	0	0	8	0	0
Barman	0	0	0	100	100	0	0	0	0	0	0	0
Elevators	42	26	0	61	45	0	29	13	0	13	0	0
Floortile	0	0	0	100	100	67	0	0	0	0	0	0
NoMystery	83	33	50	33	17	0	50	0	33	17	0	0
OpenStacks	67	0	0	67	0	0	76	5	5	95	24	5
ParcPrinter	100	50	28	28	0	6	44	0	6	22	0	0
Parking	60	0	0	95	35	0	50	0	0	55	5	0
Scanalyzer	33	0	22	100	61	28	39	0	22	17	0	6
Sokoban	50	0	12	75	38	0	50	0	12	50	12	12
Transport	3	0	0	100	97	0	0	0	0	0	0	0
VisitAll	21	0	0	84	63	0	21	0	0	37	16	0
Woodworking	97	86	11	9	3	6	6	0	6	0	0	0
GED	83	30	0	38	8	0	43	4	0	18	4	0

Table 1: For each plan improvement method, the percentage of instances where it matches the best plan ($=$); finds a plan strictly better than any other method ($<$); and finds a plan that is known to be optimal, i.e., matched by the highest lower bound ($\star$). The percentage is of instances in each domain that are solved, but not solved optimally, by LAMA@30.

[2] `http://grimm.ucsd.edu/GRIMM/`

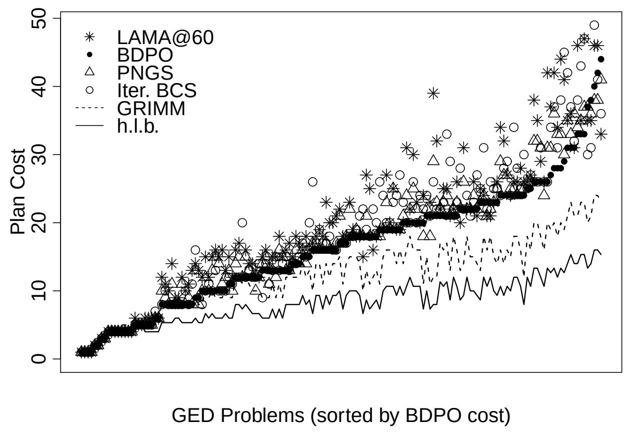


Figure 5: Plan costs (not normalised) for Genome Edit Distance problems, including a problem-specific algorithm (GRIMM).

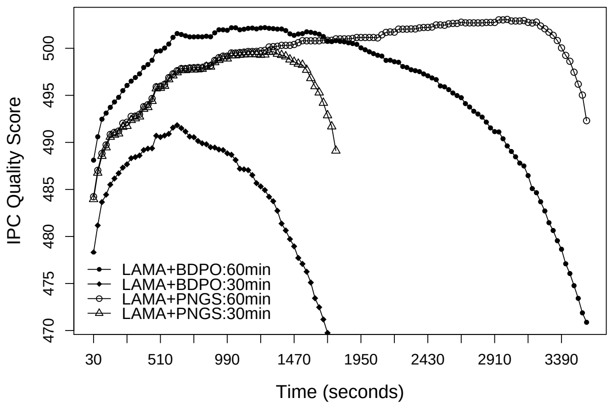


Figure 6: Impact of switching from LAMA to BDPO and LAMA to PNGS at different times during 30 and 60 minutes total runtimes.

75, and 79 problems respectively. Table 1 provides a different summary of Figures 4 and 5.

Figure 6 shows the effect of switching from LAMA to BDPO and LAMA to PNGS at different times during the total runtimes of 30 and 60 minutes. We have used problems from all the domains except Tidybot. It is clear from the figure that allocating relatively more time to BDPO than LAMA leads to a better result (as measured by the summed IPC quality score). The LAMA+BDPO combination achieves its best score when switching at around 10 and 20 minutes during the 30 and 60 minutes total runtimes, respectively. In contrast, running PNGS for relatively shorter time (less than 10 minutes) after a long run of LAMA gives the best score in their combination for both runtimes. In 30 minutes total runtime, switching to PNGS is better than switching to BDPO. However, if the total runtime extends to 1 hour, the LAMA+BDPO combination improves its best score by 10.38 units compared to its best score in 30 minutes total runtime, whereas LAMA+PNGS improves that by 3.43 units only. This suggests that the best result could be achieved by a sequential portfolio of LAMA-PNGS-BDPO, where PNGS is run for a short time, and BDPO for relatively longer time.

Conclusions

Continuing plan quality improvement is crucial in automated planning. The anytime search approach has been most successful so far to meet this goal, but also quickly reaches a limit where it becomes unable to find further improvements. Optimising a solution one small part at a time is a good idea in this respect, but the main challenge is to identify the right subproblems to re-optimise. In our experiments, we have found that at least 75% of subproblems for which we find an improvement could not have been formed by taking only sequences of consecutive actions in the input plan.

Different approaches to planning and plan optimisation have their strength at different time scales. This suggests that truly anytime planning (which provides continuing improvement of plan quality at any time scale) is best achieved by a combination of methods. PNGS, for example, can make a quick improvement over LAMA. We have shown in our experiments that extending the anytime capability of current planners with our BDPO optimisation method can further improve plans substantially, especially at a larger time scale. To achieve this, we have used a local search in the space of valid plans, where the space is formulated using a windowing heuristic over a block decomposed plan. However, the degree of block decomposition varies a lot from domain to domain. Elevator, Transport, Scanalyzer, Pegsol, and Openstack problems, for example, exhibit a high degree of decomposition, whereas Sokoban is hard and time consuming to decompose. Problems in the Woodworking and Parcprinter domains, on the other hand, can not be block-decomposed in most cases, but still exhibit good plan quality improvement by making use of our window formation and local search strategies on top of the partially ordered form.

There are many promising directions for future work. At present, we use eleven window formation rules based on the placement of predecessor, successor, and unordered blocks with respect to a candidate block. These rules only extract windows with causal chains at most three blocks long. In the experiments, we have found that different window types and sizes perform well in different domains. Therefore, dynamic rule generation for window formation based on the problem structure of different domains may perform better. We can also extend the “neighborhood” not only with larger windows but also with other types of improvement searches, like PNGS (Nakhost and Müller 2010). There is room to improve the subproblem formulation, since constructing the subproblem from different linearisations may produce different overall improvements. At present, we are using a relatively simple hill climbing search to merge the best improvements, but an alternative better merging can be done. We want to further investigate the block decomposition to get better plan structure, and are also keen to know whether adding further encapsulation into the existing blocks could produce more coherent subplans to re-optimise. We can think of better alternative heuristics to be used in bounded-cost search: The lower bound (lmcut) in bounded-cost search is not incremental, and therefore the search may spend a lot of time trying to optimise an already optimal subplan. In on-going experiments, we have found that using lazy weighted A* search (the same as used in LAMA) instead of bounded-cost greedy search in the subplanner produces much better results than our current results in most of the domains (but not all). Finally, it will be interesting to see whether our algorithm can be extended to be used during the initial planning phase to analyse the partial solution structure, and then to give a good heuristic for the remaining search by biasing the action selection towards more promising actions and away from non-promising ones.

References

R. K. Ahuja, J. Goodstein, A. Mukherjee, J. B. Orlin, and D. Sharma. A very large-scale neighborhood search algorithm for the combined through-fleet-assignment model. *INFORMS Journal on Computing*, 19(3):416–428, 2007.

J.L. Ambite and C.A. Knoblock. Planning by rewriting. *Journal of AI Research*, 15(1):207–261, 2001.

R. Barták, T. Balyo, and P. Surynek. On improving plan quality via local enhancements. In *Proc. Symposium on Combinatorial Search (SoCS'12)*, 2012.

J. Bibaï, P. Saveant, M. Schoenauer, and V. Vidal. An evolutionary metaheuristic based on state decomposition for domain-independent satisficing planning. In *Proc. 20th International Conference on Automated Planning and Scheduling (ICAPS'10)*, pages 18–25, 2010.

A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1):1636–1642, 1995.

D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

D. Furcy. Iterative tunneling search with A*. In *AAAI Workshop on Heuristic Search, Memory-Based Heuristics and their Applications*, pages 21–26, 2006.

A. Gerevini and I. Serina. LPG: A planner based on local search for planning graphs with action costs. In *Proc. 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'02)*, pages 281–290, 2002.

M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers, 2004. ISBN: 1-55860-856-7.

D. Godard, P. Laborie, and W. Nuijten. Randomized large neighborhood search for cumulative scheduling. In *Proc. 15th International Conference on Automated Planning & Scheduling (ICAPS'05)*, pages 81–89, 2005.

P. Haslum and A. Grastien. Diagnosis as planning: Two case studies. In *ICAPS'11 Scheduling and Planning Applications Workshop*, 2011.

P. Haslum. Computing genome edit distances using domain-independent planning. In *ICAPS'11 Scheduling and Planning Applications Workshop*, 2011.

P. Haslum. Incremental lower bounds for additive cost planning problems. In *Proc. 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*, pages 74–82, 2012.

Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proc. 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*, 2009.

J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of AI Research*, 14:253–302, 2001.

J. Hoffmann. Local search topology in planning benchmarks: An empirical analysis. In *Proc. 17th International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 453–458, 2001.

M. Likhachev, G. Gordon, and S Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *Proc. 17th Annual Conference on Neural Information Processing Systems (NIPS'03)*, 2003.

P. Merz and B. Freisleben. Fitness landscapes and memetic algorithm design. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 245–260. McGraw-Hill, London, 1999.

H. Nakhost and M. Müller. Action elimination and plan neighborhood graph search: Two algorithms for plan improvement. In *Proc. 20th International Conference on Automated Planning and Scheduling (ICAPS'10)*, pages 121–128, 2010.

H. Nakhost, M. Müller, R. Valenzano, and F. Xie. Arvand: the art of random walks. In *Proc. 21st International Conference on Automated Planning and Scheduling (ICAPS'11)*, pages 15–16, 2011.

A. Olsen and D. Bryce. Randward and lamar: Randomizing the ff heuristic. In *Proc. International Planning Competition (IPC'11)*, pages 55–57, 2011.

D. Ratner and I. Pohl. Joint and LPA*: Combination of approximation and search. In *Proc. National Conference on AI (AAAI'86)*, pages 173–177, 1986.

S. Richter and M. Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of AI Research*, 39:127–177, 2010.

M. Schoenauer, P. Savéant, and V. Vidal. Divide-and-evolve: a sequential hybridization strategy using evolutionary algorithms. In *Advances in Metaheuristics for Hard Optimization*, chapter 9, pages 179–198. Springer, 2008.

P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Proc. 4th International Conference on Principles and Practice of Constraint Programming (CP'98)*, pages 417–431, 1998.

F.H. Siddiqui and P. Haslum. Block-structured plan deordering. In *AI 2012: Advances in Artificial Intelligence (Proc. 25th Australasian Joint Conference*, volume 7691 of *LNAI*, pages 803–814, 2012.