Proceedings of the 8th Workshop on
Constraint Satisfaction Techniques for Planning and
Scheduling Problems

COPLAS 2013

Rome, Italy - June 11, 2013

Edited By:

Miguel A. Salido, Roman Barták, Francesca Rossi

Organizing Commitee

Miguel A. Salido
Universidad Politécnica de Valencia, Spain

Miguel A. Salido is supported by the research project TIN2010-20976-C02-01 (Min. de Economia y Competitividad, Spain) and project PIRSES-GA-2011-294931 (FP7-PEOPLE-2011-IRSES)

Roman Barták
Charles University, Czech Republic

Roman Barták is supported by the Czech Science Foundation under the contract P202/10/1188

Francesca Rossi
University of Padova, Italy

Program committee

Federico Barber, Universidad Politecnica de Valencia, Spain
Roman Bartak, Charles University, The Czech Republic
Amedeo Cesta, ISTC-CNR, Italy
Minh Binh Do, NASA Ames Research Center, USA
Enrico Giunchiglia, Universita di Genova, Italy
Peter Jarvis, NASA Ames Research Center, USA
Michela Milano, University of Bologna, Italy
Eva Onaindia, Universidad Politecnica de Valencia, Spain
Nicola Policella, European Space Agency, Germany
Hana Rudova, Masaryk University, The Czech Republic
Francesca Rossi, University of Padova, Italy
Migual A. Salido, Universidad Politecnica Valencia, Spain
Pascal Van Hentenryck, Brown University, USA
Ramiro Varela, Universidad de Oviedo, Spain
Gerard Verfaillie, ONERA, Centre de Toulouse, France
Vincent Vidal, CRIL-IUT, France
Petr Vilim, ILOG, France
Toby Walsh, UNSW, Sydney and NICTA, Australia
Neil Yorke-Smith, American University of Beirut/SRI International, USA

Foreword

The areas of AI planning and scheduling have seen important advances thanks to the application of constraint satisfaction models and techniques. Especially solutions to real-world problems need to integrate plan synthesis capabilities with resource allocation, which can be efficiently managed by using constraint satisfaction techniques. The workshop aims at providing a forum for researchers in the field of Artificial Intelligence to discuss novel issues on planning, scheduling, constraint programming/constraint satisfaction problems (CSPs) and many other common areas that exist among them. On the whole, the workshop mainly focus on managing complex problems where planning, scheduling and constraint satisfaction must be combined and/or interrelated, which entails an enormous potential for practical applications and future research.

Miguel A. Salido, Roman Barták, Francesca Rossi
COPLAS 2013 Organizers
June 2013

Table of Contents

Constraint-based approaches for cyclic scheduling

Alessio Bonfietti, Michele Lombardi, Michela Milano
DISI, University of Bologna
{alessio.bonfietti,michele.lombardi2,michela.milano}@unibo.it

Talk Abstract

Cyclic scheduling problems arise whenever a set of non-interruptible activities subject to precedence and resource constraints must be repeatedly executed a large number of times [Hanen, 1994; Draper et al., 1999]. Cyclic Scheduling has applications in many practical domains, such as manufacturing, production systems, warehouse management, embedded systems, software compilers, and chemical plants.

From a combinatorial optimization standpoint, cyclic scheduling is the problem of assigning start times to periodic activities such that the repetition interval (i.e. the *period*) of the overall application is minimal and all the precedence relations and the resource capacities are respected.

Traditional (non-cyclic) scheduling techniques have achieved a good level of maturity in the last decade [Baptiste, Le Pape, and Nuijten, 2001], but they are not trivial to apply to cyclic scheduling. Hence two main classes of approaches have been proposed for this class of problems:

Blocked scheduling: this method is based on the assumption that the schedule is repeated after its completion time (i.e. the makespan), so that consecutive schedule repetitions do not overlap [Bhattacharyya and Sriram, 2009]. This allows the use of traditional scheduling techniques, at the cost of a considerable loss of optimality. It is possible to obtain better period values by applying a graph transformation (called *unfolding* [Parhi and Messerschmitt, 1991]) to schedule a number of consecutive iterations at once. This technique, however, may dramatically increase the graph size.

Modulo scheduling: this approach works by repeatedly fixing the period and building a restricted model [Hanen, 1994]. Optimization is carried out by performing linear or binary search on the period value. This method incurs no loss of optimality, but often results in large models and may have slow convergence.

In this talk, we will introduce cyclic scheduling problems, then briefly review some of the most important existing approaches, and finally present two main contributions:

1. A constraint approach ($CROSS^*$) that can solve cyclic scheduling problems with no blocking, no graph transformation, and no need to fix the period. The method is based on a Modular Precedence Constraint (MPC) we developed, and on an efficient dedicated search strategy [Bonfietti et al., 2011]. The approach guarantees optimality under a simplifying assumption, often reasonable in practical cases.
2. A second constraint method (*CROSS*) which makes no simplifying assumption and still does not require to repeatedly build a restricted model. The approach is based on our Global Cyclic Cumulative Constraint (GCCC) and on a restart based search strategy [Bonfietti et al., 2012].

Both the approaches adopt the Constraint Programming methodology and modular arithmetic. They proved to be able to find high quality solutions in a very short time, but they can also be pushed to run complete search (although the optimality proof takes longer). The two approaches outperformed traditional blocked- and modulo- scheduling ones in terms of speed and/or solution quality, on a number of non-trivial synthetic and industrial instances.

References

Baptiste, P.; Le Pape, C.; and Nuijten, W. 2001. *Constrains-based scheduling: applying Constraint Programming to Scheduling*. Springer.

Bhattacharyya, S. S., and Sriram, S. 2009. *Embedded Multiprocessors - Scheduling and Synchronization (Signal Processing and Communications) (2nd Edition)*. CRC Press.

Bonfietti, A.; Lombardi, M.; Benini, L.; and Milano, M. 2011. A Constraint Based Approach to Cyclic RCPSP. In *Proc of CP*, 130–144.

Bonfietti, A.; Lombardi, M.; Benini, L.; and Milano, M. 2012. Global cyclic cumulative constraint. In *CPAIOR*, 81–96.

Draper, D. L.; Jonsson, A. K.; Clements, D. P.; and Joslin, D. E. 1999. Cyclic scheduling. In *Proc. of IJCAI*, 1016–1021. Morgan Kaufmann Publishers Inc.

Hanen, C. 1994. Study of a NP-hard cyclic scheduling problem: The recurrent job-shop. *European Journal of Operational Research* 72(1):82–101.

Parhi, K. K., and Messerschmitt, D. G. 1991. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Transactions on Computers* 40(2):178–195.

Post-Optimizing Individual Activity Plans through Local Search

Anastasios Alexiadis and **Ioannis Refanidis**
Department of Applied Informatics, University of Macedonia,
Egnatia 156, 54006, Thessaloniki, Greece.
talex@java.uom.gr, yrefanid@uom.gr

Abstract

Post-optimization through local search is known to be a powerful approach for complex optimization problems. In this paper we tackle the problem of optimizing individual activity plans, i.e., plans that concern activities that one person has to accomplish independently of others, taking into account complex constraints and preferences. Recently, this problem has been addressed adequately using an adaptation of the Squeaky Wheel Optimization Framework (SWO). In this paper we demonstrate that further improvement can be achieved in the quality of the resulting plans, by coupling SWO with a post-optimization phase based on local search techniques. Particularly, we present a bundle of transformation methods to explore the neighborhood using either hill climbing or simulated annealing. We present several experiments that demonstrate an improvement on the utility of the produced plans, with respect to the seed solutions produced by SWO, of more than 6% on average, which in particular cases exceeds 20%. Of course, this improvement comes at the cost of extra time.

Introduction

Calendar applications and digital personal assistants are typically based on a series of fully specified and independent events. Each of these events is defined among others by a fixed start time, a duration (or end-time) and, potentially, a location. Furthermore, many systems also support tasks. These are individual commitments potentially having a deadline to be met (e.g., writing a paper or doing the week's shopping). Tasks are usually kept separately in task lists and are not characterized by a specific start time. In these systems, as soon as a task is dropped into the calendar, it is transformed to an event.

The need to develop intelligent automated systems for calendar management has been considered ambitious for at least three reasons. Ethnographic studies have shown that people tend to seek appropriate and contextualized assistance (Palen 1999). Moreover users would not be able to define the scheduling domain explicitly, as that would require special training for the specification of their preferences with a formal representation. Finally, the scheduler must be able to account for complex and subtle preferences and constraints. While tracking this problem is considered ambitious, intelligent assistance with time and task management has been a recognized target for AI (Myers et al. 2007) (Freed et al. 2008) (Refanidis 2007) (Refanidis and Alexiadis 2011) (Berry et al. 2011) (Bank et al. 2012).

In (Refanidis and Yorke-Smith 2010) a model is presented to treat events and tasks, denoted as activities, in a uniform way. The model supports a number of unary and binary constraints and preferences over activities. Particularly, each activity is characterized by a temporal domain, a duration range, a set of alternative locations, interruptibility, utilization, preferences over the temporal domain and the alternative durations, constraints and preferences over the way parts of an interruptible activity are scheduled in time. The model also supports binary (ordering, proximity and implication) constraints and preferences between pairs of activities. In the same work, a scheduler, based on the Squeaky Wheel Optimization (SWO) framework and coupled with domain-dependent heuristics, is employed to automatically schedule a user's individual activities. SWO is a powerful but incomplete search algorithm, so the solutions it produces are generally not optimal.

In this paper we present local-search techniques that increase the quality of SWO's plan output through post-optimization. The use of local-search methods for constraint satisfaction problems has been done before (Curran, Freuder, and Jansen 2010) (Lee et al. 2009) (Schöning 2010) (Nakhost, Hoffmann, and Müller 2010). In our work, we devised and implemented a set of transformations of valid plans, such as shifting activities, changing their durations or locations, as well as merging or splitting parts of interruptible activities. Extensive experimental results have shown constant improvement over SWO's output up to 22.7%.

This paper extends previous work (Alexiadis and Refanidis 2012), by providing additional post-processing transformations to explore the neighborhood, enhanced with a stochastic local search algorithm (that is, *simulated annealing*) to avoid local maxima. As it is shown experimentally, additional improvement on the quality of the resulting plans is achieved with the new features of the post-processing phase.

The rest of the paper is structured as follows. First, we formulate the optimization problem and illustrate the SWO-based approach. Next, we present the local-search transformations to explore the neighborhood during the post-

optimization phase, using either *hill-climbing* or *simulating annealing* (Kirkpatrick, Gelatt, and Vecchi 1983) (Ingber 1993). Subsequently, we present experimental results over a large set of problem instances. Finally, we conclude the paper and identify directions of future work.

Background

In this section we present the problem formulation, as well as the SWO approach to cope with the problem.

Problem Formulation

In previous work (Refanidis and Yorke-Smith 2010), time is considered a non-negative integer, with zero denoting the current time. A set T of N activities, $T = \{T_1, T_2, \ldots, T_N\}$, is given. For each activity $T_i \in T$, its minimum duration is denoted with d_i^{min} and its maximum duration with d_i^{max}. The decision variable p_i denotes the number of parts in which the *i-th* activity has been split, with $p_i \geq 1$. T_{ij} denotes the *j-th* part of the *i-th* activity, $1 \leq j \leq p_i$. The sum of the durations of all parts of an activity must be at least d_i^{min} and no greater than d_i^{max}.[1] For each T_{ij}, the decision variables t_{ij} and d_{ij} denote its start time and duration. The sum of all d_{ij}, for a given i, must equal d_i.[2] Non-interruptible activities are scheduled as one part.

For each T_i, we define the minimum and maximum part duration $smin_i$ and $smax_i$,[3] as well as the minimum and maximum temporal distances between every pair of parts, $dmin_i$[4] and $dmax_i$.[5]

For each activity T_i, its temporal domain is defined as a set of temporal intervals defining $D_i = [a_{i1}, b_{i1}] \cup [a_{i2}, b_{i2}] \cup \ldots \cup [a_{i,F_i}, b_{i,F_i}]$, where F_i is the number of intervals of D_i.[6]

A set of M locations, $Loc = \{L_1, L_2, \ldots, L_M\}$, as well as a two dimensional, not necessarily symmetric, matrix $Dist$ that holds the temporal distances between locations are given. Each activity T_i has a set of possible locations $Loc_i \subseteq Loc$, where its parts can be scheduled. The decision variable $l_{ij} \in Loc_i$[7] denotes the particular location where T_{ij} is scheduled.[8]

Activities may overlap in time. Each activity T_i is characterized by a utilization value, $utilization_i$.[9] At any time point, the set of activities that have been scheduled should have compatible locations (i.e., locations with no temporal distance to each other) and the sum of their utilization values should not exceed the unit.

The model supports four types of binary constraints: Ordering constraints, minimum and maximum proximity constraints and implication constraints. An ordering constraint between two activities T_i and T_j, denoted with $T_i < T_j$, implies that no part of T_j can start its execution before all parts of T_i have finished.[10] A minimum (maximum) distance binary constraint between activities T_i and T_j implies every two parts, one of T_i and another of T_j, must have a given minimum (maximum) temporal distance.[11] Finally, an implication constraint of the form $T_i \Rightarrow T_j$ implies that in order to include T_i in the plan, T_j should be included as well.[12]

Scheduling personal activities is considered a constraint optimization problem. That said, the empty schedule is a valid schedule but with low utility, thus we are interested in better schedules. There are several sources of utility. The main source concerns the activities themselves. Each activity T_i included in the schedule contributes utility $U_i(d_i)$ that depends on its allocated duration. The way T_i is scheduled by a schedule π_i within its temporal domain constitutes another source of utility, $U_i^{time}(\pi_i)$. The user can define linear and stepwise utility functions of time over the temporal domain of each activity.

Any form of hard constraint can also be considered a soft constraint that might contribute utility. So, minimum and maximum distance constraints between the parts of an interruptible activity might contribute $U_{dmin_i}(\pi_i)$ and $U_{dmax_i}(\pi_i)$ respectively. Similarly, binary preferences can be defined as well over the way pairs of activities are scheduled. Especially for ordering and proximity preferences, partial satisfaction of the preference is allowed. The Degree of Satisfaction for a partial preference p, denoted with $DoS(p)$, is defined as the ratio of the number of infinitesimal pairs of parts, one from T_i and another from T_j, for which the binary preference holds, to the total number of infinitesimal pairs of parts.

To summarize, the optimization problem is formulated as follows:

Given:

1. A set of N activities, $T = \{T_1, T_2, \ldots, T_N\}$, each one of them characterized by its duration range, duration utility profile, temporal domain, temporal domain preference function, utilization, a set of alternative locations, interruptibility property, minimum and maximum part sizes as well as required minimum and maximum part distances for interruptible activities, preferred minimum and maximum part distances and the corresponding utilities.

---

[1] $\forall T_i,\ d_i^{min} \leq d_i \leq d_i^{max}$ OR $d_i = 0$ $(C1)$

[2] $\forall T_{ij}, \sum_{j=1}^{p_i} d_{ij} = d_i$ $(C2)$

[3] $\forall T_{ij}, smin_i \leq d_{ij} \leq smax_i$ $(C3)$

[4] $\forall T_{ij}, T_{ik}\ j \neq k \Rightarrow t_{ij} + d_{ij} + dmin_i \leq t_{ik} \vee t_{ik} + d_{ik} + dmin_i \leq t_{ij}$ $(C4)$

[5] $\forall T_{ij}, T_{ik}\ j \neq k \Rightarrow t_{ij} + dmax_i \geq t_{ik} + d_{ik} \wedge t_{ik} + dmax_i \geq t_{ij} + d_{ij}$ $(C5)$

[6] $\forall T_{ij}, \exists k, 1 \leq k \leq F_i : a_{ik} \leq t_{ij} \leq b_{ik} - d_{ij}$ $(C6)$

[7] $l_{ij} \in Loc_i$ $(C7)$

[8] $\forall T_{ij}, T_{mn}, T_{ij} \neq T_{mn} \wedge (Dist(l_{ij}, l_{mn}) > 0 \vee Dist(l_{mn}, l_{ij}) > 0) \Rightarrow t_{ij} + d_{ij} + Dist(l_{ij}, l_{mn}) \leq t_{mn} \vee t_{mn} + d_{mn} + Dist(l_{mn}, l_{ij}) \leq t_{ij}$ $(C8)$

[9] $\forall t, \sum_{T_{ij}:\ t_{ij} \leq t < t_{ij} + d_{ij}} utilization \leq 1$ $(C9)$

[10] $\forall T_i, T_j, T_i < T_j \Leftrightarrow d_i > 0 \wedge d_j > 0 \Rightarrow \forall T_{ik}, T_{jl}, t_{ik} + d_{ik} \leq t_{jl}$ $(C10)$

[11] $\forall T_{ik}, T_{jl}, t_{ik} + d_{ik} + dmin_{ij} \leq t_{jl} \vee t_{jl} + d_{jl} + dmin_{ij} \leq t_{ik}$ $(C11)$

$\forall T_{ik}, T_{jl}, t_{ik} + dmax_{ij} \geq t_{jl} + d_{jl} \wedge t_{jl} + dmax_{ij} \geq t_{ik} + d_{ik}$ $(C12)$

[12] $\forall T_i, T_j, T_i \Rightarrow T_j \Leftrightarrow d_i > 0 \Rightarrow d_j > 0$ $(C13)$

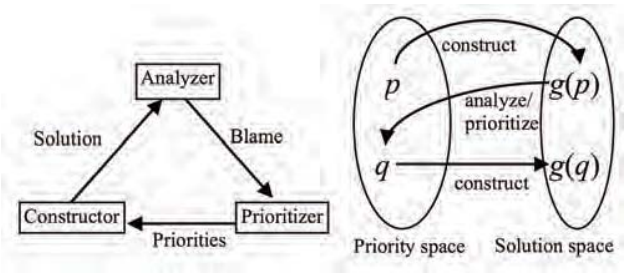


Figure 1: (a) The SWO cycle. (b) Coupled search spaces

2. A two-dimensional matrix with temporal distances between all locations.
3. A set C of binary constraints (ordering, proximity and implication) over the activities.
4. A set P of binary preferences (ordering, proximity and implication) over the activities.

Schedule

the activities in time and space, by deciding the values of their start times t_{ij}, their durations d_{ij} and their locations l_{ij}, while trying to maximize the following objective function:

$$U = \sum_{\substack{i \\ d_i \geq d_i^{min}}} (U_i(d_i) + U_i^{time}(\pi_i) + U_i^{dmin}(\pi_i) + U_i^{dmax}(\pi_i)) + \sum_{p(T_i,T_j)\in P} u_p \times DoS(p(T_i,T_j)) \quad (1)$$

subject to constraints (C1) to (C13).

The above formula calculates the total global utility of a valid plan, which directly corresponds to the *plan quality*. It is also possible to calculate a *loose upper bound* of a problem instance, by computing the sum of the maximum utilities of all preferences involved in that instance. However, since different preferences will usually contradict to each other, achieving their respective's maximum utilities simultaneously is usually impossible.

The SWO Approach

In (Refanidis and Yorke-Smith 2010), the problem is solved using the Squeaky Wheel Optimization (SWO) framework (Joslin and Clements 1999). At its core, SWO uses a Construct/Analyze/Prioritize cycle as shown in Figure 1(a). The solution is found by a greedy approach, where decisions are based on an order of the tasks determined by a priority queue. The solution is then analyzed to obtain the tasks that cannot be scheduled. Their priorities are increased, enabling the constructor to deal with them earlier on the next iteration. The cycle will be repeated till a termination condition occurs.

SWO is a fast but incomplete search procedure. The algorithm searches in two coupled spaces, as shown in Figure 1(b). These are the priority and solution spaces. Changes in the solution space are caused by changes in the priority space. Changes in the priority space occur as a result of analyzing the previous solution and using a different order of the tasks in the priority queue. A point in the solution space represents a possible solution to the problem. Small changes in the priority space can impact large ones on the solution that is generated.

SWO can easily be applied to new domains. The fact that it gives variation on the solution space makes it different than more traditional local search techniques such as WSAT (Selman, Kautz, and Cohen 1995). SWO was adapted to the Constraint Optimization Problem described in the previous section, using several domain dependent heuristics that measure the impact of the various ways that a specific activity is scheduled on the remaining ones. A solution is obtained by deciding values for the decision variables t_{ij}, d_{ij} and l_{ij}, for each T_{ij}, while trying to maximize Formula (1).

```
postprocess-swo(Activities, Best_Solution)
  New_Best_Solution ← best_neighbour(Activities,
Best_Solution)
  if U(New_Best_Solution) ≤ U(Best_Solution)
    return Best_Solution
  else
    return postprocess-swo(Activities, New_Best_Solution)
end
```

Figure 2: Hill-climbing based post-processing algorithm

Applying Local-Search Methods to Enhance SWO Output

We applied *hill-climbing* (HC) using the output of SWO as the seed value, to further enhance the solution quality. The overview of the approach is presented in Figure 2.

U denotes the objective function of formula (1) and Solution is a complete and valid assignment of values to the decision variables t_{ij}, d_{ij} and l_{ij}, for each T_{ij}. HC is initialized with the problem definition (*Activities*) and SWO's output solution as the initial *Best_Solution*.

Calculating the Best Neighbor

The *best_neighbour* function computes neighbor solutions by attempting various changes to the decision variables of the parts of activities. The algorithm always keeps the solution resulting from the change that produces the highest utility. When all the available changes have been attempted on the decision variables of every part, the algorithm chooses the neighbor solution with the best utility to continue.

For every part T_{ij} of each activity T_i, the following transformations are attempted:

Best Start Time: This transformation attempts different values for the decision variable t_{ij} of part T_{ij}, one decision variable at a time. The values attempted always belong to the temporal domain of activity T_i. For each value attempted, the constraints are checked to examine if the change is consistent with them. If it is not, it is ignored.

Changing the duration: This transformation attempts different values for the decision variable d_{ij} of part T_{ij}. The values d_{ij} are always between $smin_i$ and $smax_i$, as defined for activity T_i. Every change is checked for constraint consistency.

Decreasing the duration of a task will not result in improving the overall quality; however, if it is combined with a stochastic local search procedure like simulated annealing, interesting results arise.

Merging two parts of an activity: This transformation attempts to merge two parts of an activity into a single part. For part T_{ij}, it iterates over all other parts of activity T_i, and for every T_{ik}, $j{\neq}k$, it attempts to remove T_{ik} and move its duration to T_{ij}. T_{ik} is then removed.

For every T_{ik}, where $j{\neq}k$, the function first checks whether $d_{ij}{+}d_{ik}{\leq}smax_i$ holds. If it does not, it ignores T_{ik}. Otherwise it attempts to produce two new solutions. In the former solution, d_{ij} is increased by d_{ik}; in the latter, d_{ij} is increased by d_{ik} and t_{ij} is decreased by d_{ik}. In other words, the duration of the removed part is added either at the end or at the beginning of part T_{ij}. As above, every solution produced is checked for constraint consistency.

Transfering duration between parts of the same activity: This transformation attempts to transfer duration between two parts of the same activity. For part T_{ij}, it iterates over all other parts of activity T_i, and for every T_{ik} where $j{\neq}k$ it attempts to transfer duration from T_{ik} to T_{ij}.

For every T_{ik} (where $j{\neq}k$, $d_{ik}{>}smin_i$ and $d_{ij}{<}smax_i$), up to four neighboring solutions are computed. These are the following:

1. Move the maximum allowed duration ($trans\_d$) by setting $d_{ik} = d_{ik} - trans\_d$ and $d_{ij} = d_{ij} + trans\_d$. This moves $trans\_d$ duration from the end of T_{ik} to the end of T_{ij}.
2. Move the maximum allowed duration ($trans\_d$) by setting $d_{ik} = d_{ik} - trans\_d$, $d_{ij} = d_{ij} + trans\_d$ and $t_{ij} = t_{ij} - trans\_d$. This moves $trans\_d$ duration from the end of T_{ik} to the beginning of T_{ij}.
3. Move the maximum allowed duration ($trans\_d$) by setting $d_{ik} = d_{ik} - trans\_d$, $d_{ij} = d_{ij} + trans\_d$, $t_{ij} = t_{ij} - trans\_d$ and $t_{ik} = t_{ik} + trans\_d$. This moves $trans\_d$ duration from the beginning of T_{ik} to the beginning of T_{ij}.
4. Move the maximum allowed duration ($trans\_d$) by setting $d_{ik} = d_{ik} - trans\_d$, $d_{ij} = d_{ij} + trans\_d$ and $t_{ik} = t_{ik} + trans\_d$. This moves $trans\_d$ duration from the beginning of T_{ik} to the end of T_{ij}.

The maximum allowed duration, $trans\_d$, is calculated as $min(smax_i - d_{ij}, d_{ik} - smin_i)$. If the constraint consistency check on all the four new solutions fails, $trans\_d$ is decreased by one. This will proceed until either a new valid solution is created or $trans\_d = 0$.

Splitting a part: This transformation attempts to split a part into two parts. For part T_{ij}, where $d_{ij}{\geq}(2 \times smin_i)$ it attempts to create a new part T_{ik} where $k = p_i + 1$. The new part will have $d_{ik} = smin_i$, $l_{ik} = l_{ij}$ and the function will search for a suitable t_{ik} that succeeds on the consistency check of the new solution. If no such valid t_{ik} is found, nothing is computed.

Increasing the duration of an activity: This transformation attempts to increase the duration of a part and, consequently, of the whole activity. For part T_{ij}, where $d_{ij}{<}smax_i$, it attempts to produce up to two solutions. First, $d_{ij}+1$ is attempted. This will increase the part's duration by 1 and place the extra duration at the end of the part. On the second case, $d_{ij} + 1$ and $t_{ij} - 1$ are attempted. This will increase its duration by 1, by putting the extra duration the beginning of T_{ij}.

Swapping parts of different activities: This transformation attempts to swap two parts' start times between different activities. So, for each part T_{ij} of activity T_i, it is attempted to exchange t_{ij} with each t_{kx} of the *x-th* part of *k-th* activity.

Adding a part: This function is different from the previous ones in that it does not operate on parts but on activities, by attempting to add new parts. For an activity T_i, where:

$$\sum_{j=1}^{p_i} d_{ij} \leq (d_i^{max} - smin_i) \tag{2}$$

it is attempted to create a new part T_{ik} where $k = p_i{+}1$. The new part will have $d_{ik} = smin_i$, $l_{ik} = l_{ij}$(where $j = p_i$) and the function will search for a suitable t_{ik} that succeeds on the consistency check of the new solution. If no such valid t_{ik} is found, nothing is computed.

Adding an Activity: Similar to the previous transformation, this one is only applied on activities. If an activity is found, that was not scheduled for any reason (such as constraint violations with other activities of greater utility etc), this—and only this—transformation will be applied to that activity, as the other transformations are not defined on activities that do not have any scheduled parts. The purpose of this transformation is to include activities in the plan, that were not scheduled, by taking advantage of plan changes from the other transformations.

For an activity T_i that is not included in the current schedule, this transformation aggressively tries to schedule it by inserting parts to the timeline and within the domain of the task, starting from the earliest and proceeding to the latest time points. At each time window where a part of T_i can be inserted, it is inserted with the minimum possible duration and at the first found possible location. Due to the aggressive policy of this transformation, it might be the case that an activity will not be inserted in the current schedule, although this was possible.

Changing Locations: For every part T_{ij}, the *best_neighbour* transformation attempts to change its location and then to apply all the above methods described, except *Adding a Part* and *Adding an Activity*, which refer to activities. For every possible location $l_i \in Loc_i$ of T_{ij}, the *total travelling time* of the resulting solution is computed. Traveling time between two consequent parts, T_{ij} and T_{ul}, scheduled at l_{ij} and l_{ul}, is defined as $Dist(l_{ij}, l_{iu})$. If a suitable value for l_{ij} is found that reduces the *total travelling time*, all the above methods are re-applied on the resulting solution. The location change is then reset when we move to the next part.

This transformation allows to explore combinations of location changes with the above functions described. The logic behind the method is to search for any potential location changes that decrease the *total travelling time* of the schedule, thus giving the post-optimization algorithm more rescheduling options.

Avoiding Local Maximums

The *hill-climbing* post-processing algorithm can get stuck in local maxima. Algorithms based on *simulated annealing* (Kirkpatrick, Gelatt, and Vecchi 1983) are another type of local-search algorithms that can be applied to the above problem. Hill-climbing never makes a descending move (to states of lower global utility) and thus is not complete. In contrast a pure *random walk* is complete but extremely inefficient.

As a middle-ground to the above two extremes, we replaced hill-climbing as a post-processing algorithm with one based on *simulated annealing*. *Annealing* is the process in which one starts at a high energy state and high temperature levels, attempting to minimize the energy gradually—while lowering the temperature as well—to reach a low-energy state. Lower temperatures decrease the chances of moves to higher-energy levels (moves to worse states than the current one).

Low-energy states are reversed in our model to describe *high* global utility solutions. As a result, the simulated annealing based post-processing algorithm, presented in Figure 3, starts from the solution that results from the preceding SWO phase and gradually increases its utility. In addition, we empowered *simulated annealing* with *tabu lists*, to avoid returning to previously visited solutions. Finally, we converted all the *transformation functions* of the previous section to return all the valid neighbor solutions they calculated, instead of the best one.

There is a high probability of descending moves (to lower utility solutions), at the start of execution, which gradually decreases as the temperature variable decreases as well. This feature allows us to expand the search-space and avoid local maxima, whereas the gradual decrement allows the search algorithm to converge to a high-utility solution.

In the simulated annealing based post-processing algorithm we use the *random_neighbour* function (instead of *best_neighbour*), which returns a random neighbor solution. For increased efficiency, we define the variable lists $C_{ikt} \in C$, where each one of them holds all the valid neighbor solutions for an application of the t *transformation function*,

enhanced-postprocess-swo(Activities, Solution, Best, K, Tabu, SCHEDULE, KMAX, EMAX)

```
if K ≥ KMAX  OR  U(Best) ≥ EMAX
  return Best
NTabu ← Tabu ∪ Solution
T ← SCHEDULE[K]
C ← ∅
DO i ← 0 to ∞
  if i+ K = KMAX
      return Best
  New ← random_neighbour(Activities,Solution,C),
New ∉ NTabu, (C_ikt ∈ C) ← (C_ikt ∈ C) \ New
  ΔE ← U(New) − U(Solution)
UNTIL ΔE > 0  OR  select New with probability e^(ΔE/T)
New_K ← K + i + 1
if New > Best
  New_Best ← New
else New_Best ← Best
return enhanced-post-process-swo(Activities, New,
New_Best, New_K, NTabu, SCHEDULE, KMAX, EMAX)
end
```

Figure 3: Simulated annealing based post-processing algorithm

for an activity part T_{ik} or a whole activity T_i (in the case of *Adding a Part* or *Adding an Activity*). For activities without any scheduled parts, only *Adding an Activity* is used (which is not used for any scheduled ones). Each C_{ikt} is created dynamically when required. As an example, if the transformation *Transfer Duration* (with no location changes) is chosen for the activity part T_{ik}, the search algorithm will pop a solution out of $C_{ikTransferDuration}$. If the previous list does not exist, the transformation will be computed on the above part and $C_{ikTransferDuration}$ will be created—with all the valid neighbor solutions found. Whenever the *random_neighbour* function chooses a solution from a transformation on a part, that has already been computed, the only overhead for the algorithm will be to obtain that solution from the list. When the current solution changes, the above lists are deleted and will be recomputed on demand. The number of valid solutions, for a transformation applied on a part, is not known beforehand, thus creating the need for the above variables.

The search algorithm uses the following parameters: SCHEDULE, KMAX, EMAX and $|Tabu|$. SCHEDULE defines the cooling schedule of the *simulated annealing* search algorithm. We used a cooling schedule dependent on KMAX that is defined as: SCHEDULE[K] = SCHEDULE[K-1] $\times$ $(1 - 0.07 \times \frac{100}{\text{KMAX}})$, where SCHEDULE[1] = 0.9. KMAX (the number of iterations of the simulated annealing algorithm) was set to 2000 and EMAX (the upper bound of the problem) to the upper bound of the particular problem instance being solved. We set $|Tabu|$ (the number of past solutions kept in the tabu list) to $\frac{\text{KMAX}}{10}$.

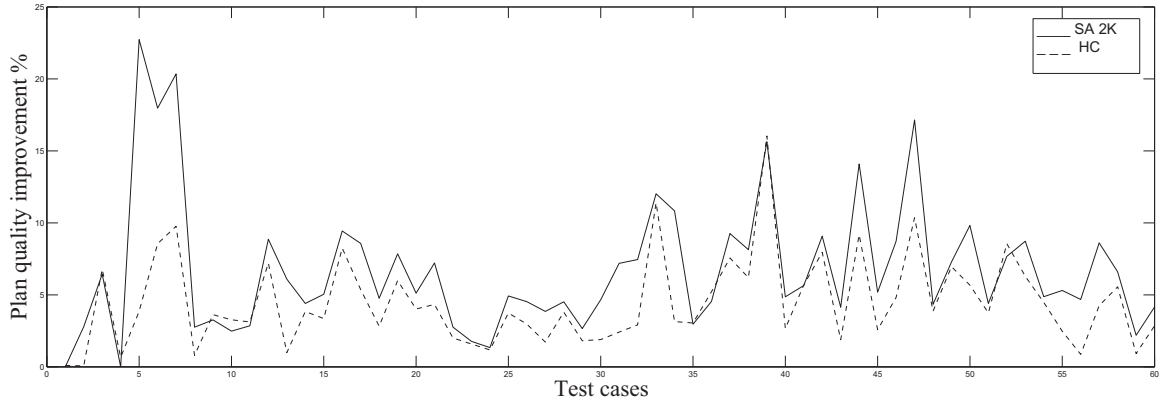


Figure 4: Improvement of plan quality over SWO. Dashed line concerns hill-climbing (HC), whereas solid line concerns simulated annealing (SA 2K) with 2,000 iterations.

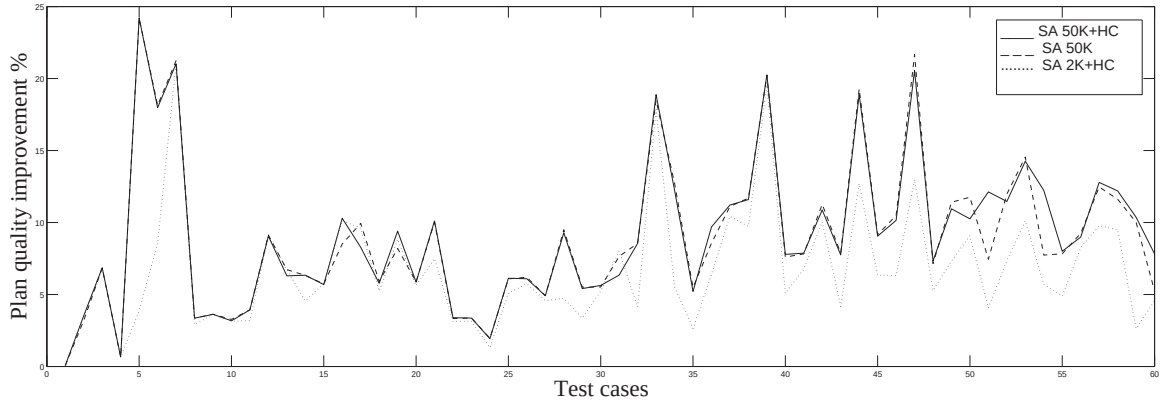


Figure 5: Improvement of plan quality over SWO for various configurations of the simulated annealing based post-processing phase. SA 50K refers to SA with 50,000 iterations. SA 50K+HC refers to SA 50K coupled with hill climbing. SA 2K+HC refers to SA with 2,000 iterations, coupled with HC.

Evaluation

We first compared the original SWO algorithm versus SWO coupled with the hill-climbing post-processing algorithm, as well as versus the SWO coupled with the simulated annealing one, on 60 test cases, ranging in size, taken from (Refanidis and Yorke-Smith 2010). The implementation of the above algorithms was done in C++ and the experiments were run on an Intel Xeon 2.66GHz processor.

The results of the first comparison are shown in Figure 4. The plot represents the plan quality percentage improvement of the two post-processing algorithms to the original SWO. The dashed line concerns the hill-climbing (HC) post-processing algorithm, whereas the solid line represents the simulated annealing post-processing algorithm with 2,000 iterations (SA 2K).

The test set (which was created for benchmarking SWO's solution quality and speed of execution) consists of five problems per number of activities, ranging from five activities to sixty, in steps of five. All the problems include activities with large temporal domains with many intervals each. A number of binary constraints and preferences were defined on them.

As we can observe from Figure 4, there is an improvement in all test cases with both HC and SA 2K. The best case was a 22.7% improvement in plan quality with SA 2K. The average was a an improvement of 4.6% for HC and 6.7% for SA 2K. As can be seen from the above results, SA 2K performed better than HC. Moreover, we can observe that SWO does not reach local maxima, though it approaches them in a satisfactory manner.

Next, we run some tests with more aggressive parameters, which are displayed in Figure 5. The dotted line represents a combination of the two algorithms, i.e. hill-climbing running on top of the resulting solution of SA (SA 2K+HC),

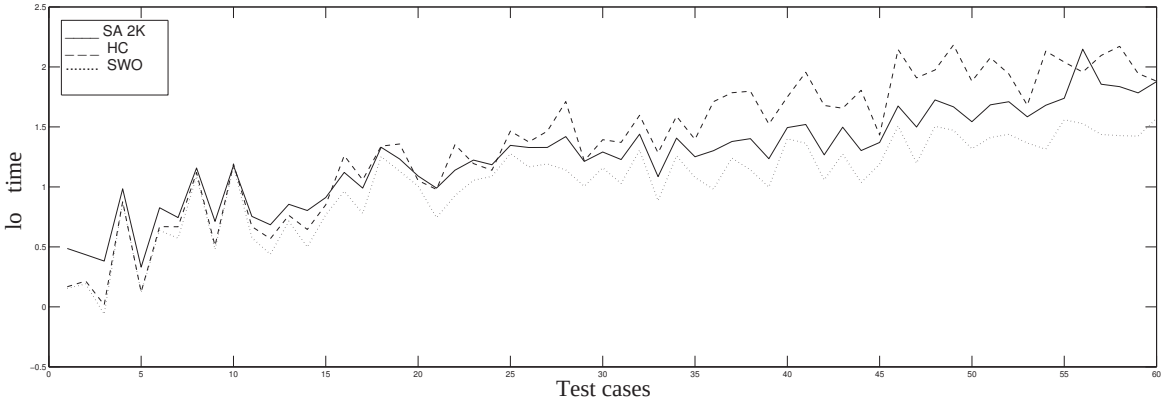


Figure 6: Execution time results for SWO, HC and SA 2K.

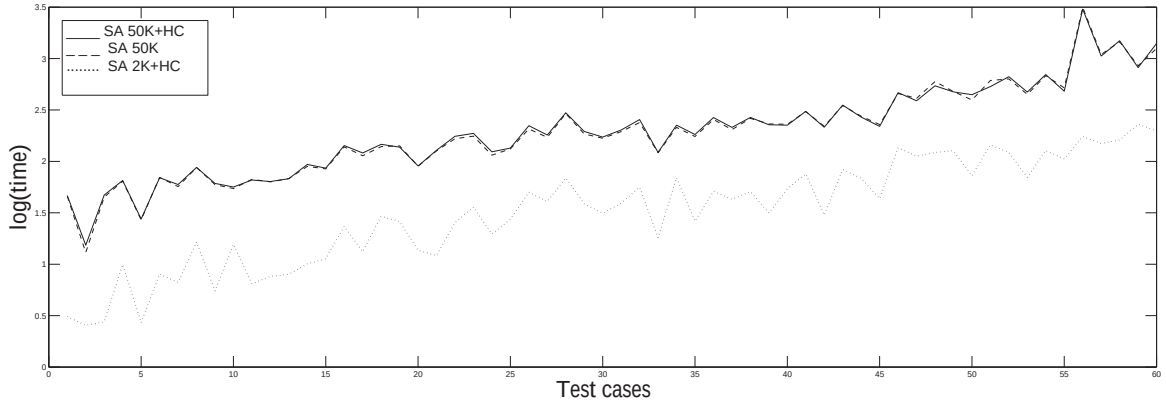


Figure 7: Execution time results for SA 50K, SA 50K+HC and SA 2K+HC.

in order to reach the nearest local maximum. The improvement over SWO was 7.03% on average. The solid line represents the combination of the two algorithms again, with a more aggressive KMAX$= 50000$ for simulated annealing (SA 50K+HC). The average improvement was 9.8%, with a best case of 24.19%. Finally, the dashed line represents the results of simulated annealing only, with KMAX$= 50000$ (SA 50K). The average improvement was 9.54% with a best case of 24.19%.

The percentage of time penalty was greater on the larger problems than the ones with fewer activities. Figure 6 compares the execution time of standard SWO to HC and SA 2K. The times given for HC and SA 2K are the combined values of the time of the main scheduler plus the post-processor. The figure is in logarithmic (base 10) scale. The dotted line represents SWO, dashed represents HC and solid represents SA 2K. The worst cases were encountered with the larger problem instances. On instances with up to 20 activities—which is the usual case for practical problems—the execution time of the three algorithms remains close. On the larger problem instances, SA 2K outperforms HC in terms of execution time, whereas simultaneously it provides better quality results as has been shown in Figure 4.

Having in mind real-world situations, we consider the time requirements acceptable, since the problems of the test set are artificially created with increased complexity (many binary constraints and preferences), thus they are more complex than typical real-world situations that are expected to involve fewer activities with less interdependencies between them.

In Figure 7 we compare the execution time of the more aggressive algorithms, SA 50K, SA 2K+HC and SA 50K+HC (combined values of scheduler plus post-processor). SA 50K and SA 50K+HC were the slowest, though their execution time is close, as the cost of the HC phase is overshadowed by the execution time of the SA with the large KMAX value. On the other hand, SA 2K+HC is 114% slower than SA 2K on average.

Finally, we compared the performance of the above algorithms to the *loose upper bound* of their respective problem

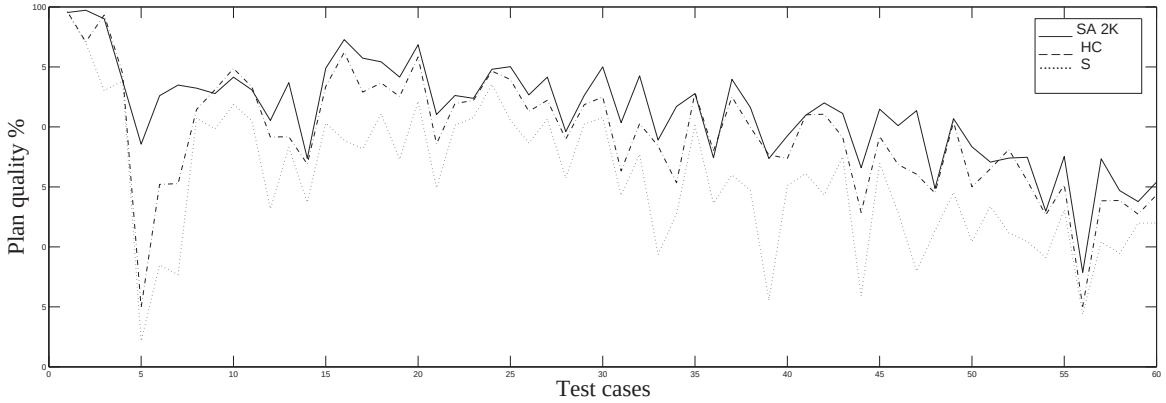


Figure 8: Quality relative to the *upper bound* for SWO, HC and SA 2K.

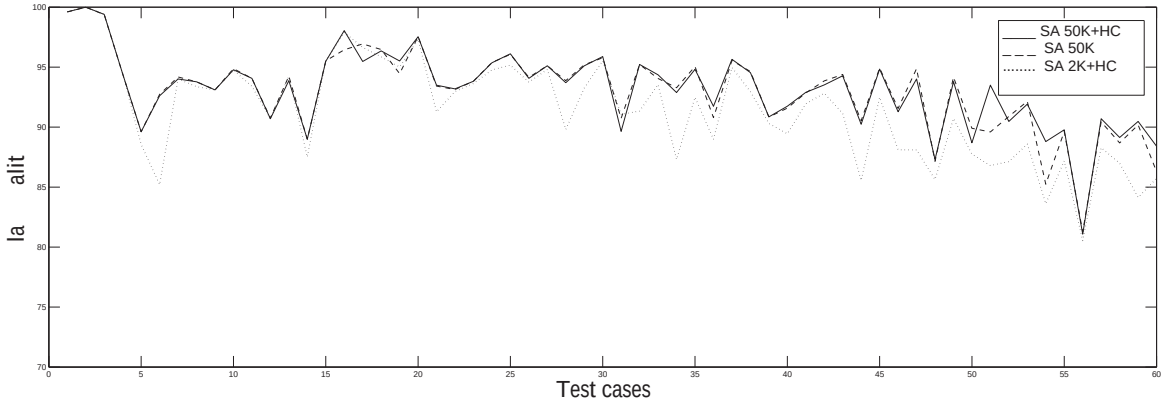


Figure 9: Quality relative to the *upper bound* for SA 50K, SA 50K+HC and SA 2K+HC.

instances. These results are shown in Figures 8 and 9. Particularly, SWO had an average plan quality of 85% to the loose upper bound, whereas HC and SA 2K had 89% and 91% respectively. Coupling SA with HC resulted in less than 1% of average improvement relative to SA, whereas increasing KMAX raised it to 93%.

The worst case for SWO was a plan quality of 72%, which was increased to 75% with HC, and to 89% with SA 2K. SA 50K+HC further raised it to 90%.

Conclusions and Future Work

In this paper we employed local search post-optimization methods to further improve the quality of personal plans with complex preferences, that were originally produced by an adaption of the Squeaky Wheel Optimization framework. Due to the complexity and interplay of the preference model, post processing seems to be a necessity to obtain locally optimal plans.

Based on our experimental results, we found two practical configurations of the proposed post-optimization modules: The first one, which is optimized for speed of execution, consists of a simulated annealing post-processing phase with a predefined moderate number of iterations. The second one, which is optimized for quality of solutions (producing high-utility local optimal solutions), couples a long simulated annealing phase with a hill-climbing one.

For the future, we are considering new enhancements to both the model and the scheduling algorithm. Concerning the model, we are working on the inclusion of joint activities, i.e., activities where many persons are involved, and on non-monotonic temporal domain preferences—for greater flexibility when specifying the user's temporal preferences. Another option that is considered is to convert the problem definition to a planning problem, with preconditions and effects, which will allow the model to be greatly enriched.

Concerning post-optimization, we are working on devising new transformations for exploring the local neighborhood, including selected combinations of the transformations presented in this paper. Using heuristics to select a subset of the transformations to consider might be of great

significance, in order to keep the whole approach efficient. Another thought we are considering is comparing the post-processing module applied on a random initial solution to SWO+SA. Moreover a formal analysis of real-world test cases would enable us to produce test data more closely related to real-world problem instances. Last of all, we will compare the individual transformations' contribution to each other in producing the final solution.

Acknowledgements

The authors are supported for this work by the European Union and the Greek Ministry of Education, Lifelong Learning and Religions, under the program "Competitiveness and Enterprising" for the areas of Macedonia-Thrace, action "Cooperation 2009", project "A Personalized System to Plan Cultural Paths (myVisitPlanner^{GR})", code: 09SYN-62-1129.

References

Alexiadis, A., and Refanidis, I. 2012. Meeting the objectives of personal activity scheduling through post-optimization. First International Workshop on Search Strategies and Non-standard Objectives (SSNOWorkshop'12), in conjunction with CPAIOR-2012, Nantes, France.

Bank, J.; Cain, Z.; Shoham, Y.; Suen, C.; and Ariely, D. 2012. Turning personal calendars into scheduling assistants. In *Proceedings of the 2012 ACM Annual Conference Extended Abstracts on Human Factors in Computing Systems Extended Abstracts*, CHI EA '12, 2667–2672. New York, NY, USA: ACM.

Berry, P. M.; Gervasio, M.; Peintner, B.; and Yorke-Smith, N. 2011. Ptime: Personalized assistance for calendaring. *ACM Trans. Intell. Syst. Technol.* 2(4):40:1–40:22.

Curran, D.; Freuder, E.; and Jansen, T. 2010. Incremental evolution of local search heuristics. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, 981–982. New York, NY, USA: ACM.

Freed, M.; Carbonell, J.; Gordon, G.; Hayes, J.; Myers, B.; Siewiorek, D.; Smith, S.; Steinfeld, A.; and Tomasic, A. 2008. Radar: a personal assistant that learns to reduce email overload. In *Proceedings of the 23rd National Conference on Artificial intelligence - Volume 3*, AAAI'08, 1287–1293. AAAI Press.

Ingber, L. 1993. Simulated annealing: Practice versus theory. *Mathematical and Computer Modelling* 18(11):29 – 57.

Joslin, D., and Clements, D. P. 1999. Squeaky wheel optimization. *J. Artif. Intell. Res. (JAIR)* 10:353–373.

Kirkpatrick, S.; Gelatt, C. D.; and Vecchi, M. P. 1983. Optimization by simulated annealing. *Science* 220:671–680.

Lee, H.-J.; Cha, S.-J.; Yu, Y.-H.; and Jo, G.-S. 2009. Large neighborhood search using constraint satisfaction techniques in vehicle routing problem. In Gao, Y., and Japkowicz, N., eds., *Advances in Artificial Intelligence*, volume 5549 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. 229–232.

Myers, K.; Berry, P.; Blythe, J.; Conley, K.; Gervasio, M.; McGuinness, D.; Morley, D.; Pfeffer, A.; Pollack, M.; and Tambe, M. 2007. An intelligent personal assistant for task and time management.

Nakhost, H.; Hoffmann, J.; and Müller, M. 2010. Improving local search for resource-constrained planning. In Felner, A., and Sturtevant, N. R., eds., *SOCS*. AAAI Press.

Palen, L. 1999. Social, individual and technological issues for groupware calendar systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: the CHI is the Limit*, CHI '99, 17–24. New York, NY, USA: ACM.

Refanidis, I., and Alexiadis, A. 2011. Deployment and evaluation of selfplanner, an automated individual task management system. *Computational Intelligence* 27(1):41–59.

Refanidis, I., and Yorke-Smith, N. 2010. A constraint-based approach to scheduling an individual's activities. *ACM TIST* 1(2):12.

Refanidis, I. 2007. Managing personal tasks with time constraints and preferences. In Boddy, M. S.; Fox, M.; and Thiébaux, S., eds., *ICAPS*, 272–279. AAAI.

Schöning, U. 2010. Comparing two stochastic local search algorithms for constraint satisfaction problems. In Ablayev, F., and Mayr, E., eds., *Computer Science – Theory and Applications*, volume 6072 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. 344–349.

Selman, B.; Kautz, H.; and Cohen, B. 1995. Local search strategies for satisfiability testing. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 521–532.

Propagation of PDDL3 Plan Constraints

P@trik Haslum
Australian National University & NICTA Optimisation Research Group
firstname.lastname@anu.edu.au

Abstract

We present a sound, though incomplete, and tractable propagation procedure for PDDL3 trajectory constraints, with the aim of providing an inexpensive unsatisfiability test for sets of such constraints. The propagator is supported by (tractable) methods that derive additional constraints from the problem description. It is applied to compute lower bounds on penalty for problems with soft trajectory constraints (preferences).

Introduction

PDDL version 3 (Gerevini et al. 2009) introduced *trajectory constraints*, a subset of linear temporal logic that can express constraints on the sequence of states visited by the execution of a plan, in addition to the constraint on the end state of the sequence that is imposed by the planning goal. Given a planning problem P and a set of trajectory constraints C, the question we must answer is, is there a plan for P whose induced state sequence satisfies C? This question is clearly as hard as deciding if there is any plan at all for P, i.e., PSPACE-complete.

However, suppose we know that P has a plan, and that we need to check not one set of trajectory constraints but a large number of different sets of constraints. This situation arises, for example, if the trajectory constraints are "soft", i.e., preferences rather than hard constraints, and we are searching for a most preferred subset that is satisfiable w.r.t. P. We should, at least in some cases, be able to infer that a constraint set C is unsatisfiable w.r.t. P without exhaustively searching through all plans for P.

This paper presents an approach to this problem, in the form of a sound but incomplete, and tractable, propagation procedure for PDDL3 trajectory constraints. That is, given a constraint set C, and some information extracted from the problem P, the propagator computes additional constraints that are implied by those given; if it finds an implied contradiction, we know that C is unsatisfiable w.r.t. P. Because the propagator reasons (mostly) not about the problem but only about constraints extracted from it, time complexity scales additively in the size of P and the number of constraint sets to be tested.

This work is motivated by a specific example of the kind of problem described above: The Rovers Qualititative-Preferences domain from the 2006 International Planning Competition. In this domain, the objective is to satisfy a maximum weight subset of preferences over trajectory constraints. For each problem there is a plan that achieves the hard goals, but the complete set of constraints can not be simultaneously satisfied. Identifying subsets of constraints that are contradictory (w.r.t. P) allows computing bounds on the minimum penalty, due to unsatisfied preferences, of any plan (Haslum 2007). My previous approach to testing satisfiability of plan constraint sets was to compile the constraints into the problem and test for unsolvability with the admissible h^m heuristic. The disadvantage of this test is that the complexity of computing h^m depends on the size of the problem (albeit only polynomially, for fixed m). As we will demonstrate, this causes the propagation-based test to scale up much better as the size of the problem grows: for the largest instances, it is three orders of magnitude faster at performing a single test. However, because the compilation-based test is able to exploit properties of the h^m heuristic to amortise computation over several tests, the difference in total runtime is only a factor 2.89 (median). On the other hand, the propagation- and compilation-based methods are complementary, in the sense that both find unsatisfiable sets that the other cannot detect. Thus, lower bounds based on the combined results of both methods are generally best.

PDDL3

PDDL3 (Gerevini et al. 2009) extends PDDL with two new features: Preferences are "soft goals", which may be either normal, final state goals or preferences over trajectory constraints. Trajectory constraints are expressed using a set of five modal operators, which may not be nested. The satisfaction of a constraint is determined by the sequence of states visited by a plan's execution. Each PDDL3 operator corresponds to a particular formula in linear temporal logic (Pnueli 1977), provided a suitable interpretation of LTL over finite state sequences (Bauer and Haslum 2010).

For ease of presentation, we consider here a standard propositional STRIPS model of planning problems, without negation. That is, a planning problem P consists of a set of propositional atoms (V), a set of actions (A), and an initial state s_0. A state (including s_0) is an assignment of truth values to the atoms in V, i.e., a propositional logic model. Each action a is described by its precondition ($\mathrm{pre}(a)$), add ($\mathrm{add}(a)$) and delete ($\mathrm{del}(a)$) effects, which are all sets of

Constraint name	φ	$\vec{s} = s_0, s_1, \ldots, s_n \models \varphi$ iff...
`(at-end` α`)`	$\mathsf{F}\alpha$	$s_n \models \alpha$
`(always` α`)`	$\mathsf{A}\alpha$	$\forall i\, s_i \models \alpha$
`(sometime` α`)`	$\mathsf{E}\alpha$	$\exists i\, s_i \models \alpha$
`(at-most-once` α`)`	$\mathsf{AMO}\alpha$	$\forall i$ if $s_i \models \alpha$ then $\exists j \geq i\, \forall i \leq k \leq j\, s_k \models \alpha$ and $\forall k > j\, s_k \not\models \alpha$
`(sometime-before` α β`)`	$\beta\,\mathsf{SB}\,\alpha$	$\forall i$ if $s_i \models \alpha$ then $\exists j < i\, s_j \models \beta$
`(sometime-after` α β`)`	$\beta\,\mathsf{SA}\,\alpha$	$\forall i$ if $s_i \models \alpha$ then $\exists j \geq i\, s_j \models \beta$
Never α	$\mathsf{N}\alpha$	$\forall i\, s_i \not\models \alpha$
Never β after α	$\beta\,\mathsf{NA}\,\alpha$	$\forall i$ if $s_i \models \alpha$ then $\forall j \geq i\, s_j \not\models \beta$

Table 1: PDDL3 and auxiliary plan constraints.

atoms, interpreted as conjunctions. The action is applicable in a state s iff $s \models \mathrm{pre}(a)$, and applying it leads to a state s' where all atoms in $\mathrm{add}(a)$ are true, all atoms in $\mathrm{del}(a) - \mathrm{add}(a)$ are false, and all other atoms retain their value from s. Note that this definition of a planning problem does not include a goal. PDDL3 has a special trajectory constraint for facts that must hold at the end of a plan execution. Thus, the standard notion of a planning goal is subsumed by the more general condition of satisfying a set of trajectory constraints, defined below.

Every sequence of actions, $\vec{a} = a_1, \ldots, a_n$, from A that is executable from the initial state induces a corresponding sequence of states, $\vec{s} = s_0, s_1, \ldots, s_n$, visited by the execution. We call this an *execution of* P. PDDL3 trajectory constraints are evaluated over state sequences. We write $\vec{s} \models \varphi$, where φ is a trajectory constraint, if $\vec{s}$ satisfies φ. We also write $P \models \varphi$ if every execution of P satisfies φ. Given a planning problem P and a set C of trajectory constraints, we say that C is satisfiable w.r.t. P iff there exists an execution of P that satisfies each constraint in C.

The PDDL3 trajectory constraints and their satisfaction conditions are summarised in Table 1. It also introduces an abbreviation for each constraint, and two auxiliary constraints which will be useful in describing the propagation algorithm. Note that PDDL3 does not allow nesting of modal operators: the formulas α and β are only allowed to be *state formulas*, i.e., Boolean formulas over V. To ensure that implication between state formulas can be decided in polynomial time, we assume that these formulas are single atoms or sets of atoms (i.e., conjunctions). This is, however, not an essential restriction of the propagation algorithm. If support for general formulas is desired, we may either give up tractability and use a complete SAT solver to decide implication, or use some sound but incomplete polynomial-time implication test. The only requirement on the test is that it is closed under transitivity; that is, if the test proves $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, it must also prove $\alpha \rightarrow \gamma$.

Note the asymmetry between the `sometime-before` and `sometime-after` constraints: $\alpha\,\mathsf{SB}\,\beta$ requires α to hold strictly before β is first achieved, while $\alpha\,\mathsf{SA}\,\beta$ requires α to hold at the same time as or after any time that β is true. The propagation procedure currently does not consider SA constraints. Some of the challenges and possibilities of making inferences from such constraints are discussed later.

In the following we will use one additional notation: $\mathsf{D}a$, where a is an action. It is read "disallowed a", and means that a must not appear in any action sequence. It is not a modal operator like in other trajectory constraints, but is implied by those in some situations. For example, if the state sequence must satisfy $\mathsf{N}\alpha$ and $\mathrm{pre}(a) \rightarrow \alpha$, then a can not be part of the corresponding action sequence. We use $\mathsf{D}a$ as a shorthand for stating that some condition that prevents the inclusion of a holds.

Inferring Constraints from the Problem

The propagation procedure works on a set of trajectory constraints, C. However, since the aim is to infer if the constraint set is inconsistent w.r.t. a planning problem, P, we extract certain information from P, which is provided as input to the propagator. This information is (mostly) expressed as additional trajectory constraints.

This is an important design decision. Since the motivation is to perform quick (in-)consistency tests on many different trajectory constraint sets, the complexity of the test should not depend (too much) on the size of P. It is acceptable to perform relatively expensive (though still tractable) computation on P to extract information that is used by the test, but not to repeat this computation for every constraint set that is tested.

Mutual Exclusion Mutual exclusion, or "mutex" for short, holds between two state formulas, α and β, iff there is no reachable state in which both of them are true, i.e., $P \models \mathsf{N}(\alpha \wedge \beta)$. We use the shorthand notation $\mathrm{mutex}(\alpha, \beta)$.

Deciding mutual exclusion in general is as hard as solving the planning problem, but there are numerous methods for computing a sound but incomplete set of mutex relations, using admissible heuristics or invariant-finding methods (e.g. Gerevini and Schubert 1998; Rintanen 2000; Helmert 2006). In the implementation of the propagator we use the pair-wise atom mutex set found by the h^2 heuristic.

Landmarks The concept of landmarks in planning was first introduced by Porteous, Sebastia and Hoffmann (2001), and have been used in many ways since. Informally, a landmark is "something that must happen at some point in any plan". Different varieties of landmarks have been defined, where the "something" is a fact, formula or set of actions.

We consider a landmark relation between state formulas: α *is a landmark of* β iff α must be achieved (strictly) before β in every execution of P. This is precisely the same as saying that $P \models \alpha\,\mathsf{SB}\,\beta$. Fact landmarks, in the usual

sense, are landmarks of the planning goal. Deciding if the landmark relation holds between arbitrary state formulas is again PSPACE-hard, but a sound approximation for single-atom state formulas can be computed in polynomial time by testing relaxed reachability of β in a problem modified by removing α from the add effects of all actions and the initial state. Note that if α is true in s_0 or β is unreachable in the original problem, $\alpha \,\mathsf{SB}\, \beta$ holds trivially. Such trivial relations are ignored.

The Never-After Relation $P \models \beta \,\mathsf{NA}\, \alpha$ iff β does not hold in, and cannot be achieved from, any reachable state where α holds. This is a kind of extended mutex relation: a normal mutex says that α and β cannot be true simultaneously, while $\beta \,\mathsf{NA}\, \alpha$ says that if α ever was true, β can never become true.

A sound but incomplete set of never-after relations between single atoms can be computed by relaxed reachability tests. Let p be an atom and s_p a state in which every atom except those that are mutex with p is true: if q is not relaxed reachable from s_p, then $P \models q \,\mathsf{NA}\, p$. This is similar to Vidal & Geffner's (2004) computation of inter-action distances, but distinguishing only the case of infinite distance (unreachability).

Conditional Constraints Relations between formulas, like landmarks and never-after, are a consequence of lack of choice. Generally, the more alternative ways there are of achieving β, the fewer α's will be landmarks of it. However, if actions are disallowed (because their preconditions or effects contradict some constraint), choices narrow, and new relations that previously did not hold may become valid. For example, suppose there are two alternative plans for getting from `A` to `B`: `(go A C)`, `(go C B)` and `(go A D)`, `(go D B)`. If, however, `(go A C)` is disallowed, there is only one way and `(at D)` becomes a landmark of `(at B)`. If both `(go A C)` and `(go A D)` are disallowed, `(at B)` becomes unreachable. This idea extends also to other types of trajectory constraints.

Definition 1 *$\langle \varphi, \overline{A} \rangle$, where φ is a trajectory constraint and $\overline{A}$ a set of actions, is a* conditional constraint *of P iff φ holds in every execution of P that does not include any action in $\overline{A}$.*

That is, if all actions in $\overline{A}$ become disallowed, then the constraint φ is satisfied by all remaining executions of P. Equivalently,

$$P \models \left(\bigwedge_{a \in \overline{A}} \mathsf{D}a \right) \rightarrow \varphi.$$

We also say that φ holds in P conditional on $\overline{A}$.

We consider two types of conditional constraints: landmarks, i.e., constraints $\alpha \,\mathsf{SB}\, \beta$, and unreachability, i.e., constraints of the form $\mathsf{N}\alpha$. The next proposition provides a method to compute a sound, but not necessarily complete, set of such conditional constraints, with single-atom state formulas, without enumerating subsets of actions. It may be that, for example, conditional never-after constraints also exist in a problem, but we currently do not have an effective method of finding them.

Proposition 2 *Let p be an atom that is false in the initial state, and $\mathrm{Adds}(p) = \{a \mid p \in \mathrm{add}(a)\}$ the set of actions that add p: $\langle \mathsf{N}p, \mathrm{Adds}(p) \rangle$ is a conditional (unreachability) constraint of P.*

Furthermore, for each $q \in \bigcup_{a \in \mathrm{Adds}(p)} \mathrm{pre}(a)$, such that q is not already a landmark of p, let $\mathrm{Reqs}(q) = \{a \mid q \in \mathrm{pre}(a)\}$ be the set of actions whose preconditions include q: $\langle q \,\mathsf{SB}\, p, \mathrm{Adds}(p) - \mathrm{Reqs}(q) \rangle$ is a conditional (landmark) constraint of P.

Proof: Since p is not initially true, some action in $\mathrm{Adds}(p)$ must take place to make it true; hence, if these actions are disallowed, $\mathsf{N}p$ must hold. Furthermore, if all actions in $\mathrm{Adds}(p) - \mathrm{Reqs}(q)$ are disallowed, all remaining actions that add p have q in their precondition. Thus q must be achieved before p. □

Enumerating pairs of propositions p and q gives directly a polynomial-time algorithm for computing conditional landmarks and unreachability, since the sets $\mathrm{Adds}(p)$ and $\mathrm{Reqs}(q)$ are immediate from action definitions.

In the example above, this algorithm will find that `(at D)` is a landmark of `(at B)` conditional on $\{$`(go C B)`$\}$, but it will not find that the same relation is also conditional on $\{$`(go A C)`$\}$.

The Propagation Algorithm

Algorithm 1 presents the main propagation algorithm. Its arguments are a set of trajectory constraints, C, and a set of conditional (landmark and unreachability) constraints, X. C is assumed to contain any non-conditional constraints (landmark and never-after relations) inferred from the problem.

The algorithm first infers state formulas that can never hold in any execution that satisfies C (lines 4–23), repeating a cycle of inferences until a fixpoint is reached, then infers state formulas that must hold, at some point, in any execution (lines 24–25). All inferences are restricted to state formulas that appear in the input. If there is a formula that must hold but cannot, a contradiction has been found. Finally, a separate check for inconsistencies with `at-most-once` constraints is done. This is detailed in Algorithm 2.

The algorithm maintains two data structures: a set D of disallowed actions and a directed graph G over the set of state formulas that combines `sometime-before` relations and implications. That is, there is an edge from α to β in G iff either $\beta \,\mathsf{SB}\, \alpha \in C$ or $\alpha \rightarrow \beta$. Note the direction of the edge in the first case: it is from the "triggering side" of the constraint, i.e., α. Both of these relations are transitive, and the first step in the fixpoint loop (lines 9–11) is to add any missing transitively implied SB relation (this assumes that implications are already transitively closed). Whenever a new constraint $\mathsf{E}\alpha$ or $\mathsf{N}\alpha$ is derived, it is propagated along SB relations and implications. Likewise, when a new SB constraint is derived, existing N constraints are propagated through it.

Algorithm 1 Trajectory Constraint Propagation

```
 1: procedure PROPAGATE(C, X)
 2:   Let F = {state formulas in C and X}.
 3:   Set D = {a | Aα ∈ C and (⋀_{p∈del(a)} ¬p) → ¬α}.
 4:   for each Aα ∈ C do
 5:     for each β ∈ F such that mutex(α, β) do
 6:       ASSERTNEVER(β)
 7:   Set G = ⟨F, {(α, β) | β SB α ∈ C or α → β}⟩.
 8:   repeat
 9:     for each (α, β), (β, γ) ∈ G do
10:       if (α, γ) ∉ G then
11:         ASSERTSB(γ, α).
12:     for each α, β ∈ F do
13:       if α SB β ∈ C and β SB α ∈ C then
14:         ASSERTNEVER(α).
15:       if α SB β ∈ C and β NA α ∈ C then
16:         ASSERTNEVER(β).
17:     for each ⟨α SB β, Ā⟩ ∈ X do
18:       if Ā ⊆ D then
19:         ASSERTSB(α, β).
20:     for each ⟨Nα, Ā⟩ ∈ X do
21:       if Ā ⊆ D then
22:         ASSERTNEVER(α).
23:   until no change.
24:   for each Eα ∈ C and Fα ∈ C do
25:     ASSERTSOMETIME(α).
26:   if ∃α such that Eα ∈ C and Nα ∈ C then
27:     return contradiction.
28:   if ∃α, β s.t. α NA β, β NA α, Eα, Eβ ∈ C then
29:     return contradiction.
30:   if not CHECKAMO(C, D) then
31:     return contradiction.
32:   return C.

33: procedure ASSERTSB(α, β)
34:   Add α SB β to C and (β, α) to G.
35:   if Nα ∈ C and Nβ ∉ C then
36:     ASSERTNEVER(β).

37: procedure ASSERTNEVER(α)
38:   Add Nα to C.
39:   for each action a do
40:     if pre(a) → α or add(a) → α then Add a to D.
41:   for each α SB β ∈ C do
42:     if Nβ ∉ C then ASSERTNEVER(β).
43:   for each β ∈ F such that β → α do
44:     if Nβ ∉ C then ASSERTNEVER(β).

45: procedure ASSERTSOMETIME(α)
46:   Add Eα to C.
47:   for each β SB α ∈ C do
48:     if Eβ ∉ C then ASSERTSOMETIME(β).
49:   for each β ∈ F such that α → β do
50:     if Eβ ∉ C then ASSERTSOMETIME(β).
```

A cyclic SB relation becomes unsatisfiable if any formula in the cycle is ever true. Similarly, a cycle between SB and NA relations implies that the trigger formula can never hold. When a state formula is proven unachievable (i.e., Nα is derived), the set of disallowed actions is updated with actions whose preconditions or add effects imply the formula (subroutine ASSERTNEVER, lines 39–40). The last step in the fixpoint loop (lines 17–22) is to check if the set of disallowed actions triggers any conditional constraint, which may result in further SB or N constraints becoming active.

Proposition 3 *If implications derived between state formulas are closed under transitivity,* PROPAGATE *is correct.*

Proof: The correctness of the CHECKAMO procedure is shown separately in Proposition 4 below. Hence, we consider only other inferences made by PROPAGATE:

Line 3: Aα and $\left(\bigwedge_{p\in\mathrm{del}(a)} \neg p\right) \rightarrow \neg\alpha$ entail Da.

If a appears in the action sequence, then the negation of every atom in $\mathrm{del}(a)$ holds in the state immediately after. If this implies $\neg\alpha$, clearly α does not hold in every state, contradicting Aα.

Lines 4–6: Aα and $\mathrm{mutex}(\alpha, \beta)$ entail Nβ. Obvious.

Lines 9–11: Since implications are already transitively closed, both edges (α, β) and (β, γ) cannot be implications; thus at least one of β SB α and γ SB β is in C. If both are, then for any state sequence $\vec{s}$ satisfying C, if $s_i \models \alpha$ there exists a $j < i$ such that $s_j \models \beta$ (otherwise $\vec{s} \not\models \beta$ SB α), and therefore there exists a $k < j$ such that $s_k \models \gamma$ (otherwise $\vec{s} \not\models \gamma$ SB β). Thus, if any state satisfies α there must be an earlier state satisfying γ. Hence $\vec{s} \models \gamma$ SB α.

Suppose β SB α is in C and $\beta \rightarrow \gamma$. If $s_i \models \alpha$, there is a $j < i$ such that $s_j \models \beta$; by the implication $s_j \models \gamma$ as well. If instead γ SB β is in C and $\alpha \rightarrow \beta$, then if $s_i \models \alpha$, then $s_i \models \beta$ by implication; thus there is a $j < i$ such that $s_j \models \gamma$.

Lines 13–14: α SB β and β SB α entail Nα. By the transitivity of SB shown above, the cyclic SB relation implies α SB α. This implies that if α is ever true, it would also have to be true in an earlier state; thus, there can be no first state in which α holds. Hence, α holds in no state. (The cycle also entails Nβ, but this is added by ASSERTNEVER.)

Lines 15–16: α SB β and β NA α entail Nβ. Suppose $\vec{s} \models \{\alpha \text{ SB } \beta, \beta \text{ NA } \alpha\}$, and that $s_i \models \beta$ for some state s_i in $\vec{s}$. There is a $j < i$ such that $s_j \models \alpha$ (otherwise $\vec{s} \not\models \alpha$ SB β). But since $\vec{s} \models \beta$ NA α, this implies $s_k \not\models \beta$ for all $k \geq j$. Hence $s_i \not\models \beta$.

Lines 17–22: If φ holds conditional on $\overline{A}$, then $\left(\bigwedge_{a\in\overline{A}} \mathrm{D}a\right) \rightarrow \varphi$ by definition.

Line 24–25: Goals that must hold in the final state (Fα) must also hold at some point in the execution.

Lines 26–27: Eα and Nα are contradictory. This is immediate from their definitions in Table 1.

Lines 28–29: α NA β and β NA α entail $\neg$E$\alpha \vee \neg$Eβ. α NA β and β NA α means that α and β are mutex, i.e., that there is no reachable state in which both are true. Thus, they cannot be achieved at the same time. If α becomes true at any point,

Algorithm 2 Checking `at-most-once` Constraints

1: **procedure** CHECKAMO(C, D)
2: Set $\mathrm{AMO\_Acts} = \emptyset$. // AMO_Acts is a set of sets
3: **for** each $\mathsf{AMO}\alpha \in C$ **do**
4: **if** $s_0 \models \alpha$ **then**
5: Add $\mathrm{ActChF}(\alpha)$ to AMO_Acts.
6: Set $D = D \cup \mathrm{ActChT}(\alpha)$.
7: **else**
8: Add $\mathrm{ActChF}(\alpha)$ and $\mathrm{ActChT}(\alpha)$ to AMO_Acts.
9: Let $\mathrm{Cands} = \{p \mid s_0 \not\models p, \exists \mathsf{E}\alpha \in C : \alpha \rightarrow p\}$.
10: Set $\mathrm{sets} = \emptyset$. // sets is a set of sets of sets
11: **for** each $p \in \mathrm{Cands}$ **do**
12: Let $\mathrm{sets}(p)$ be the smallest $\{A_1, \ldots, A_m\} \subseteq$ AMO_Acts s.t. $(\mathrm{Adds}(p) - D) \subseteq \bigcup_{i=1,\ldots,m} A_i$.
13: Add $\mathrm{sets}(p)$ to sets.
14: **for** each $S = \{A_1, \ldots, A_m\} \in \mathrm{sets}$ **do**
15: Let $R = \{p \mid \mathrm{sets}(p) = S\}$.
16: Let $G = \langle R, \{(p, q) \mid \exists a \notin D : p, q \in \mathrm{add}(a)\}\rangle$.
17: Let $R' = \mathrm{APXINDEPENDENTSET}(G)$.
18: **if** $|R'| > |S|$ **then**
19: **return** *false*.
20: **return** *true*.

β cannot be achieved later, and vice versa. Hence, at most one of $\mathsf{E}\alpha$ and $\mathsf{E}\beta$ can be satisfied.

Lines 35–36, 41–42: $\alpha\,\mathsf{SB}\,\beta$ and $\mathsf{N}\alpha$ entail $\mathsf{N}\beta$. If $s_i \models \beta$ for any i, there must be a $j < i$ such that $s_i \models \alpha$. This contradicts $\mathsf{N}\alpha$.

Lines 39–40: $\mathsf{N}\alpha$ and $\mathrm{pre}(a) \rightarrow \alpha$ entail $\mathsf{D}a$; $\mathsf{N}\alpha$ and $\mathrm{add}(a) \rightarrow \alpha$ entail $\mathsf{D}a$. If a appears in the action sequence, $\mathrm{pre}(a)$ holds in the state where it is applied and $\mathrm{add}(a)$ in the state immediately after. If either implies α, this contradicts $\mathsf{N}\alpha$.

Lines 43–44: $\mathsf{N}\alpha$ and $\beta \rightarrow \alpha$ entail $\mathsf{N}\beta$. Obvious.

Lines 47–48: $\mathsf{E}\alpha$ and $\beta\,\mathsf{SB}\,\alpha$ entail $\mathsf{E}\beta$. If $\vec{s} \models \mathsf{E}\alpha$ then $s_i \models \alpha$ for some state s_i in $\vec{s}$. Since $\vec{s} \models \beta\,\mathsf{SB}\,\alpha$, this implies that $s_j \models \beta$ for some $j < i$. Hence $\vec{s} \models \mathsf{E}\beta$.

Lines 49–50: $\mathsf{E}\alpha$ and $\alpha \rightarrow \beta$ entail $\mathsf{E}\beta$. Obvious. □

The $\mathsf{AMO}\alpha$ constraint states that α may be true in at most one contiguous subsequence of states. That is, if α is true at some point and later becomes false, it may not become true again. The procedure for checking unsatisfiability of this constraint type, shown in Algorithm 2, is based on counting. Each state formula α that appears in an AMO constraint is associated with two sets of actions: $\mathrm{ActChF}(\alpha) = \{a \mid \mathrm{pre}(a) \rightarrow \alpha, \left(\bigwedge_{p \in \mathrm{del}(a)} \neg p\right) \rightarrow \neg\alpha\}$ and $\mathrm{ActChT}(\alpha) = \{a \mid \mathrm{mutex}(\mathrm{pre}(a), \alpha), \mathrm{add}(a) \rightarrow \alpha\}$. Actions in $\mathrm{ActChF}(\alpha)$, when applied, necessarily change the value of α from true to false, and actions in $\mathrm{ActChT}(\alpha)$ change it from false to true. The $\mathsf{AMO}\alpha$ constraint implies that at most one action in each of these sets can appear in any plan. (In fact, if α is initially true, no action in $\mathrm{ActChT}(\alpha)$ can appear in the plan.) Next, we find a set of atoms that are not initially true but implied by existing E constraints, i.e., atoms that must be achieved at some point, and such that the set of still allowed actions that add each of them is covered by the union of at-most-once action sets, $\mathrm{ActChF}(\alpha)$ and $\mathrm{ActChT}(\alpha)$, where $\mathsf{AMO}\alpha \in C$. These atoms are grouped into sets whose achievers are covered by the same at-most-once action sets, and from each a subset such that no action adds two atoms in the subset is found. This amounts to solving a independent set problem over the (undirected) graph that has the atoms in the set as nodes and an edge between two atoms iff there is a, still allowed, action that adds both. Since finding a maximal independent set is NP-hard, it is solved with an approximation algorithm (Boppana and Halldórsson 1992). If the size of such a set is greater than the number of at-most-once action sets that covers its achievers, we have a contradiction.

Proposition 4 *If* CHECKAMO(C, D) *returns false, no sequence of actions satisfies* $C \cup \{\mathsf{D}a \mid a \in D\}$.

Proof: We first establish that any sequence of actions $a_1, \ldots, a_n$ satisfying $\mathsf{AMO}\alpha$:

(1) contains at most one action from $\mathrm{ActChF}(\alpha)$;
(2) contains at most one action from $\mathrm{ActChT}(\alpha)$; and
(3) if $s_0 \models \alpha$, contains no action from $\mathrm{ActChT}(\alpha)$.

Suppose a_i and a_l ($i < l$) both belong to $\mathrm{ActChF}(\alpha)$. By construction of $\mathrm{ActChF}(\alpha)$, this means $s_{i-1} \models \alpha$, $s_i \not\models \alpha$, $s_{l-1} \models \alpha$, and $s_l \not\models \alpha$. Note that $l - 1 > i$, since s_i and s_{l-1} cannot be the same state. However, since $s_{i-1} \models \alpha$, the $\mathsf{AMO}\alpha$ constraint requires that there is a $j \geq i - 1$ such that $s_k \models \alpha$ for all $i - 1 \leq k \leq j$ and $s_k \not\models \alpha$ for all $k > j$. Any choice of $j > i - 1$ violates the first condition, since $s_i \not\models \alpha$. But chosing $j = i-1$ violates the second condition, since $s_{l-1} \models \alpha$ and $l - 1 > i - 1$. This shows (1).

For (2), suppose a_i and a_l ($i < l$) both belong to $\mathrm{ActChT}(\alpha)$. Similar to the previous case, this means $s_{i-1} \not\models \alpha$, $s_i \models \alpha$, $s_{l-1} \not\models \alpha$, and $s_l \models \alpha$. By the same argument as above, this contradicts $\mathsf{AMO}\alpha$.

For (3), suppose $s_0 \models \alpha$ and that a_i belongs to $\mathrm{ActChT}(\alpha)$. This means $s_{i-1} \not\models \alpha$ and $s_i \models \alpha$. Note that $i > 1$, since s_0 and s_{i-1} cannot be the same state. Since $s_0 \models \alpha$ and $s_{i-1} \not\models \alpha$ $s_k \not\models \alpha$ must hold for all $k > i - 1$ for $\mathsf{AMO}\alpha$ to be satisfied. But this is contradicted by $s_i \models \alpha$.

Suppose the condition of the **if** statement on line 18, $|R'| > |S|$, is true. From (1) and (2) above, no more than $|S|$ actions from the set $\bigcup_{A_i \in S} A_i$ can appear in any action sequence satisfying C. Also to satisfy C, each atom in R' must be achieved. Since no action adds more than one atom in R', this means at least $|R'|$ actions that add some atom in R' must take place. But all actions that add some atom in R' and that are not disallowed are contained in $\bigcup_{A_i \in S} A_i$. Clearly, no action sequence can contain both at least $|R'|$ and at most $|S|$ actions from this set. □

Proposition 4 refers only to *sequences* of actions. Since PDDL3 constraints are evaluated over the sequence of states visited by a plan, they can, in some situations, be satisfied by a parallel plan even when not satisfiable by any sequential plan, because the parallel plan does not visit the states that occur where parallel actions are interleaved (Gerevini

et al. 2009, Section 2.4.2). However, no two actions in $\mathrm{ActChF}(\alpha)$ can occur in parallel, since they all destroy each other's preconditions. Likewise, no two actions in $\mathrm{ActChT}(\alpha)$ can take place in parallel: Since α holds after applying any action in $\mathrm{ActChT}(\alpha)$, the action must destroy the precondition of every action in $\mathrm{ActChT}(\alpha)$, as otherwise the mutex relation between α and those preconditions would not hold. Hence, contradictions found by CHECK-AMO are valid also if we consider parallel plans.

Evaluation

The trajectory constraint propagator was designed with the problem of computing lower bounds for problems in the Rovers QualitativePreferences domain in mind, so it is natural to test it in this setting. Some limitations of the current implementation (e.g., that all state formulas are atoms, and not considering `sometime-after` constraints) are also due to this particular problem set.

The method previously used to test unsatisfiability of a constraint set (Haslum 2007) was to compile the constraints into the problem, i.e., to create a modified problem P' such that any plan for P' satisfies the constraints, and check unsolvability of the resulting problem with the h^m admissible heuristic (Haslum and Geffner 2000), with $m = 1$ or 2. The compilation is also somewhat specialised for the constraints that appear in the Rovers QualitativePreferences problem set, and for use with the h^m test for unsolvability.

With both tests, unsatisfiable constraint sets are found by simply enumerating and testing subsets of constraints in the problem in order of increasing size, skipping sets that contain a subset already proven unsatisfiable. End-state goals (`at-end` constraints) are included in every test. With the compilation-based test, `sometime` constraints are treated in a special way for efficiency reasons. As a result, the two methods do not test the exact same subsets of constraints.

Compilation of Trajectory Constraints

PDDL3 trajectory constraints can be compiled away, with a polynomial increase in problem size (Gerevini et al. 2009). The compilation used in the proof of this is based on converting the constraint to an automaton, accepting exactly the state sequences that satisfy the constraint, encoding the automaton into the problem, and posing its acceptance as a goal. Variations of this compilation have been used in some planners supporting PDDL3 (e.g. Edelkamp 2006; Baier and McIlraith 2006).

The compilation we have used is simplified, mainly by restricting it to trajectory constraints in which state formulas are single atoms. It is also somewhat tailored to support inference by the h^m heuristic, which is used to detect unsolvability of the compiled problem. We assume that the constraints to be tested are not trivially satisfied or contradicted, e.g., for Ap that p is not initially false and for Ep and p SB q that p is not initially true. The compiled problem, P' is an ordinary planning problem, with an end-state goal. For each constraint, P' is modified as follows:

Fp: p becomes a goal.

Ap: Remove from P' any action with $p \in \mathrm{del}(a)$.

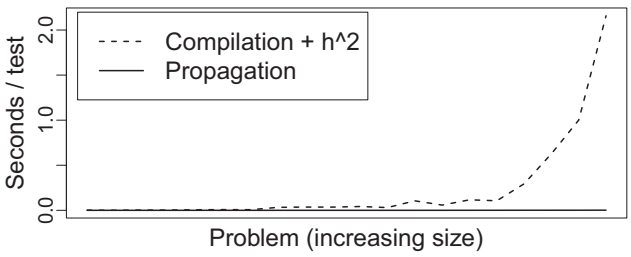


Figure 1: Time per unsatisfiability test, using compilation and the h^2 heuristic and using propagation.

Ep: Add a new atom had-p, and add had-p to the add effects of any action with $p \in \mathrm{add}(a)$. had-p becomes a goal.

p SB q: Add a new atom had-p, and add had-p to the add effects of any action with $p \in \mathrm{add}(a)$. Add had-p to the precondition of any action with $q \in \mathrm{add}(a)$.

AMOp: First, for each action that adds or deletes p, ensure the action is "toggling" w.r.t. p. That is, if the action deletes p, its precondition must include p and if it adds p its precondition must be mutex with p. This property can be enforced by splitting non-toggling actions into two cases (Hickmott et al. 2007). Next, add a new, initially true, atom first-p, and add first-p to the delete effects of any action with $p \in \mathrm{del}(a)$, and to the precondition of any action with $p \in \mathrm{add}(a)$.

Testing a constraint set C with the compilation-based method proceeds in three steps: First, the constraints are compiled to produce problem P'. Second, the h^m heuristic is computed from the initial state of this problem. Third, the problem goals are evaluated with the heuristic. If $h^m(G') = \infty$, where G' is the goal set of P', C is unsatisfiable w.r.t. P.

The first two steps are relatively time-consuming, while the last is very quick. The h^m heuristic approximates the cost of achieving a set (i.e., conjunction) of atoms of size greater than m by the cost of the most expensive subset of size m. Thus, if $|G'| > m$, h^m will only detect unsolvability if there is a m-subset of G' that is also unsolvable. This is the reason for the special treatment of `sometime` constraints: Since these are translated into end-state goals, it is not necessary to perform a separate compilation for each subset of them. Instead, all `sometime` constraints (together with some subset of other constraints) are compiled, and each subset of at most m of them (together with `at-end` goals) tested in the heuristic evaluation step.

Results

There are 20 problems in the test set. The number of soft trajectory constraints varies from 14 to 274, and tends to increase with problem size. The compilation-based test is applied to every subset of at most 2 constraints of types other than `sometime`, and every combination of one of these sets with at most m `sometime` constraints. The propagation-based test is applied to every subset of at most 3 constraints. For the largest problem that is potentially over 20 million tests. However, because sets that contain a subset already proven to be unsatisfiable are skipped, only 3.4 million tests

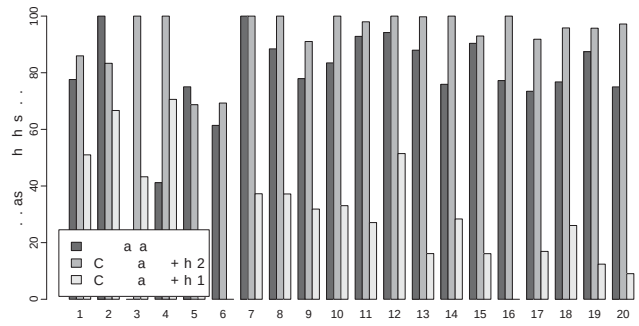


Figure 2: Lower bounds computed from unsatisfiable constraint sets found by the propagation-based and compilation-based tests, as a percentage of the highest lower bound for each problem. The highest lower bound was obtained by combining the results of both tests.

are actually needed for this problem.

The propagation-based test is faster than the compilation-based test with the h^2 heuristic, and although it does grow with problem size, it does so much more slowly. The difference ranges from about 10 times faster to over 1000 times faster on the largest problems. This is shown in Figure 1. However, because of the separate treatment of `sometime` constraints, the compilation-based method performs far fewer tests. As a result, the reduction in total runtime is only a factor that ranges from 1.85 to 4.77 (median 2.89). On two problems, the compilation-based method is faster (by a factor less than 2).

The compilation-based test with h^2 is generally more powerful than the propagation-based method: it finds more unsatisfiable constraint sets for 15 problems, and a higher lower bound for 17 problems. The lower bounds computed from the unsatisfiable sets found by the propagation-based method are, however, not far below: with the exception of two problems, they are above 75% of higher bound. There is, however, also a strong complementarity between the two methods: For 16 of the 20 problems, both methods find some unsatisfiable set that is not found by the other, and for 14 problems, the lower bound computed from the union of unsatisfiable sets found by both methods dominates the bounds produced by either method alone. Figure 2 displays this in detail. The majority of unsatisfiable constraint sets found by propagation but not with compilation and h^2 include at least one `at-most-once` constraint. This is because the CHECKAMO procedure can find contradictions implied by a more than two subgoals, as shown by the following example.

Example 1 Rovers QualitativePreferences problem #6 has, i.a., the following goals[1]:

(1) `(sent-image obj0 col)`
(2) `(sent-image obj0 LR)`
(3) `(sent-image obj1 LR)`

To achieve a `sent-image` goal, some rover must take the image (achieving `(have-image ?rover ?obj ?mode)`), and then send it. The `take-image` action, requires, among other preconditions, `(calibrated ?camera)`, which is also deleted by the action (i.e., a camera must be recalibrated before each photo), and `(supports ?camera ?mode)`.

In problem #6, there are two rovers, `rover0` and `rover1`. `rover0` has two cameras, `cam0` and `cam1`, which support both image modes; `rover1` has one camera, supporting only mode `col`. Thus, landmark analysis infers the constraints

(4) `(sometime-before (sent-image obj0 LR) (have-image rover0 obj0 LR))`
(5) `(sometime-before (sent-image obj1 LR) (have-image rover0 obj1 LR))`.

Now, consider the constraints

(6) `(at-most-once (calibrated cam0))`
(7) `(at-most-once (calibrated cam1))`
(8) `(sometime (have-image rover0 obj0 col))`.

From (2) and (4), and (3) and (5), the propagator derives

(9) `(sometime (have-image rover0 obj0 LR))`
(10) `(sometime (have-image rover0 obj1 LR))`.

Hence, the set Cands in CHECKAMO contains, i.a.,

p_1: `(have-image rover0 obj0 LR)`
p_2: `(have-image rover0 obj1 LR)`
p_3: `(have-image rover0 obj0 col)`

$A_1 = \mathrm{ActChF}(\texttt{(calibrated cam0)})$, the set of actions that change `(calibrated cam0)` from true to false, consists of all `take-image` actions using `cam0`; likewise, $A_2 = \mathrm{ActChF}(\texttt{(calibrated cam1)})$ consists of all `take-image` actions using `cam1`. Since these are the only two cameras on `rover0`, these two sets together contain all actions that add each of the three candidate atoms; thus $\mathrm{sets}(p_i) = \{A_1, A_2\}$ for $i = 1, 2, 3$. Thus, we have $S = \{A_1, A_2\}$, and $R = \{p_1, p_2, p_3\}$. Since no action adds more than one of the candidate atoms, the independent set problem is trivial (the graph has no edges), so $R' = R$. Since $|R'| = 3 > 2 = |S|$, CHECKAMO finds a contradiction.

The propagation-based test also finds a few contradictions involving more than two `sometime-before` constraints. The compilation-based test with the h^1 heuristic is fast, but also much weaker.

Discussion

The design goal for the PDDL3 propagator was to have a sound, though incomplete, test for unsatisfiability of a trajectory constraint set w.r.t. a planning problem, whose time complexity is not strongly related to the size of the problem. This led to a two-stage approach, where relevant information is extracted from the problem in a preprocessing step, and passed to the propagator in the form additional constraints. Results on problems from the IPC 2006 Rovers QualitativePreferences domain confirm that this goal has been largely met.

A limitation of the current propagator is that it makes no use of `sometime-after` constraints. This constraint stands out in that it is the only one that is not finitely satisfiable. For any constraint set C that does not include a `sometime-after` constraint, if C is satisfied by any execution of a problem P, it is also satisfied by a *finite* execution:

[1]The names of some predicates have been changed to shorten them, and to make the example easier to grasp.

Because the number of possible states is finite, in any infinite execution there is an index i such that all states that appear in the sequence appear in the finite sequence up to i, and this finite sequence satisfies all constraints in C. The `sometime-after` constraint does not have this property: If α and β are mutually exclusive state formulas, the constraint set $\{\alpha \,\mathsf{SA}\, \beta, \beta \,\mathsf{SA}\, \alpha\}$ is satisfied by a state sequence that alternates infinitely between states where α and β hold, but it is not satisfied by any finite sequence.

Like SB, the SB constraint is transitive: $\beta \,\mathsf{SA}\, \alpha$ and $\gamma \,\mathsf{SA}\, \beta$ entail $\gamma \,\mathsf{SA}\, \alpha$. It is also propagated by implication, from the right-hand side to the left. Because it is not strict, however, a cycle of SA constraints do not entail that the state formulas in the cycle can never hold. From $\alpha \,\mathsf{SA}\, \beta$ and $\beta \,\mathsf{SA}\, \alpha$ we can only infer that in any finite execution, $\alpha \wedge \beta$ must hold at some point. Applying this inference rule would mean that the set of state formulas handled by the propagation algorithm is no longer restricted to those that appear in its input; in fact, it may grow exponentially. Of course, we could limit inference to checking whether any pair of state formulas that appear in a SA-cycle are mutex.

To the best of my knowledge, this is the first approach aimed specifically at proving the unsatisfiability of PDDL3 trajectory constraints. Most planners have dealt with such constraints by compiling them away. An exception is the work of Bienvenu, Fritz and McIlraith (2006), which deals with preferences over trajectory constraints (more general than those expressible in PDDL3) using progression. Their optimistic evaluation provides a lower bound, but a rather weak one, since it assumes that any constraint that has not been irrecoverably violated by the state sequence so far will be satisfied. Baier, Bacchus and McIlraith (2009) describe an admissible heuristic, based on delete-relaxed plans, for planning with preferences. In combination with compilation it can provide lower bounds for problems with soft trajectory constraints, probably comparable to those obtained using the compilation described above with the h^1 heuristic. Resolution-based proof procedures for the full linear temporal logic have been developed in the area of formal methods (Fisher, Dixon, and Peim 2001). These methods are complete, and hence necessarily of high complexity.

Although the propagator is designed to prove unsatisfiability of trajectory constraints w.r.t. a planning problem, it could potentially also detect unsolvability of a problem without trajectory constraints, by applying it to just the constraints extracted from the problem and the problem's goal.

Acknowledgements NICTA is funded by the Australian Government represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

Baier, J., and McIlraith, S. 2006. Planning with first-order temporally extended goals using heuristic search. In *Proc. 21st National Conference on AI (AAAI'06)*.

Baier, J.; Bacchus, F.; and McIlratih, S. 2009. A heuristic search approach to planning with temporally extended preferences. *Artificial Intelligence* 173(5–6):593–618.

Bauer, A., and Haslum, P. 2010. LTL goal specifications revisited. In *Proc. 19th European Conference on Artificial Intelligence (ECAI'10)*.

Bienvenu, M.; Fritz, C.; and McIlraith, S. 2006. Planning with qualitative temporal preferences. In *Proc. 10th International Conference on Principles of Knowledge Representation and Reasoning (KR'06)*, 134–144.

Boppana, R., and Halldórsson, M. 1992. Approximating maximum independent sets by excluding subgraphs. *BIT* 32(2).

Edelkamp, S. 2006. On the compilation of plan constraints and preferences. In *Proc. of the 16th International Conference on Automated Planning and Scheduling (ICAPS'06)*, 374–377.

Fisher, M.; Dixon, C.; and Peim, M. 2001. Clausal temporal resolution. *ACM Transactions on Computational Logic* 2(1):12–56.

Gerevini, A., and Schubert, L. 1998. Inferring state constraints for domain-independent planning. In *Proc. 15th National Conference on Artificial Intelligence (AAAI'98)*, 905–912.

Gerevini, A.; Haslum, P.; Long, D.; Saetti, A.; and Dimopoulos, Y. 2009. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence* 173(5-6):619–668.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proc. 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'00)*, 140–149. AAAI Press.

Haslum, P. 2007. Quality of solutions to IPC5 benchmark problems: Preliminary results. In *ICAPS'07 Workshop on the IPC*.

Helmert, M. 2006. The Fast Downward planning system. *Journal of AI Research* 26:191–246.

Hickmott, S.; Rintanen, J.; Thiébaux, S.; and White, L. 2007. Planning via Petri net unfolding. In *Proc. 20th International Conference on Artificial Intelligence (IJCAI'07)*, 1904–1911.

Pnueli, A. 1977. The temporal logic of programs. In *Proc. of the 18th Symposium on Foundations of Computer Science (FOCS'77)*, 46–57.

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2001. On the extraction, ordering and usage of landmarks in planning. In *Proc. 6th European Conference on Planning (ECP'01)*.

Rintanen, J. 2000. An iterative algorithm for synthesizing invariants. In *Proc. 17th National Conference on Artificial Intelligence (AAAI'00)*, 806–811.

Vidal, V., and Geffner, H. 2004. Branching and pruning: An optimal temporal POCL planner based on constraint programming. In *Proc. 19th National Conference on Artificial Intelligence (AAAI'04)*, 570–577.

Decoupling the Multiagent Disjunctive Temporal Problem*

James C. Boerkoel Jr.
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology, Cambridge, MA
boerkoel@csail.mit.edu

Edmund H. Durfee
Computer Science and Engineering
University of Michigan, Ann Arbor, MI
durfee@umich.edu

Abstract

In multiagent scheduling, each agent has to schedule its activities to respect its local (internal) temporal constraints, and also to satisfy external constraints between its activities and activities of other agents. A scheduling problem is *decoupled* if each agent can independently (and thus privately, autonomously, etc.) form a solution to its local problem such that agents' combined solutions are guaranteed to satisfy all external constraints. We expand previous work that decouples multiagent scheduling problems containing strictly conjunctive temporal constraints to more general problems containing disjunctive constraints. While this raises a host of challenging issues, agents can leverage shared information as early and as often as possible to quickly adopt additional temporal constraints within their local problems that sacrifice some local scheduling flexibility in favor of decoupled, independent, and rapid local scheduling.

Motivation

Many scheduling problems can be represented using temporal constraint networks, where events are represented as variables whose domains are the possible execution times, and where constraints restrict the timings between events in terms of bounds on the differences between variables' values. Figure 1 represents one such scheduling problem involving four tasks, with constraints between the variables representing the start and end times of each task. For instance the edge from $T3^B_{ST}$ to $T3^B_{ET}$ represents the constraint $T3^B_{ET} - T3^B_{ST} \in [50, 80]$, that is, the duration of task T3 is between 50 and 80 minutes. The ***Disjunctive Temporal Problem (DTP)*** (Stergiou and Koubarakis 2000) is a general version of such problems, where constraints represent a choice among many constituent temporal difference constraints, each of which has its own bounds expressed over its own pair of timepoints. The disjunctive constraints in Figure 1 are represented with double lines, where all edges belonging to a single disjunctive constraint intersect (e.g., T1 must follow T2 by 60 minutes or precede T2 by 45).

*A version of this abstract appears at AAMAS 2013. This is a synopsis of work that appears at AAAI 2013.

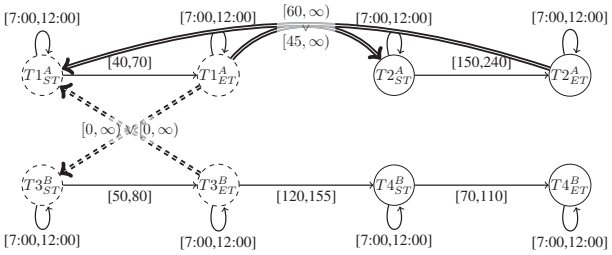


Figure 1: An example MaDTP with four tasks. $T1$ and $T2$ belong to agent A while $T3$ and $T4$ belong to agent B.

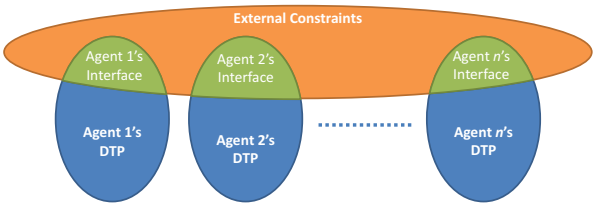


Figure 2: Overview of MaDTP structure. External constraints relate the local DTPs of different agents.

A ***consistent*** DTP is one that has a ***solution***—a scheduling of specific times to each variable that respects all constraints. There are flexibility benefits to representing DTP ***solution spaces***—sets of solutions naturally captured within the flexible ranges of times between temporal bounds—rather than a single, possibly brittle solution. The DTP is known to be an NP-hard problem, where for general DTPs with $|C|$ disjunctive temporal constraints each with k possibilities, each of the $\mathcal{O}(k^{|C|})$ possible networks of constraints must be explored in the worst case (Stergiou and Koubarakis 2000; Tsamardinos and Pollack 2003). As depicted graphically in Figure 2, a ***multiagent*** DTP (MaDTP) (Boerkoel and Durfee 2012) is one whose variables, and constraints among them, are partitioned among n agents. For example, the top and bottom rows in Figure 1 represent tasks belonging to two different agents, A and B respectively. The DTPs of different agents are constrained through ***external constraints***, represented using dashed lines.

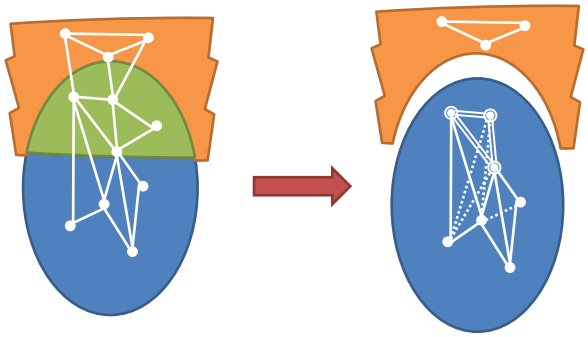


Figure 3: The objective of the MaTDP is to add new local constraints (shown as dashed lines) that render external constraints superfluous and thus encapsulate more of an agent's local problem.

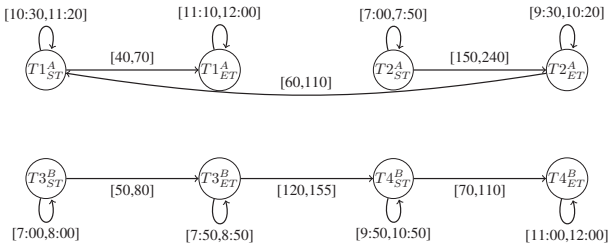


Figure 4: An example temporal decoupling of the example problem. Any combination of solutions to the top and bottom components will form a solution to the original problem in Figure 1.

While external constraints capture key relationships between different agents' activities, they also introduce *coupling* between agents' local problems. To mitigate the level of interagent coupling, we introduce a property of MaDTPs called local decomposability. An MaDTP is ***locally decomposable*** if, for any agent i, any locally consistent assignment of values to any subset of agent i's local timepoint variables can be extended to a *joint* solution. *Local* decomposability approximates full decomposability, which requires full connectivity so that any assignment of *any* subset of variables is extensible to a solution. However, to modify the completion time for a deliverable (e.g., a data analysis or query answer), an agent still needs to check that others can adjust their schedules to accommodate the change, which can trigger a further cascade of adjustments by other agents to their schedules.

Here we extend the original definition of the ***Multiagent Temporal Decoupling Problem (MaTDP)*** (Hunsberger 2002; Boerkoel and Durfee 2011; 2013b; 2013a), which was previously defined based on problems containing strictly conjunctive constraints. Agents' local DTP subproblems form a ***temporal decoupling*** of a consistent MaDTP $\mathbf{D}$ if (i) each agent's local DTP subproblem is consistent; and (ii) *any* combination of solutions to each agent's local DTP subproblem yields a joint solution to $\mathbf{D}$. As illustrated in Figure 3, the MaTDP is defined as finding, for each agent, a set of additional constraints (e.g., tighter bounds on the timings of

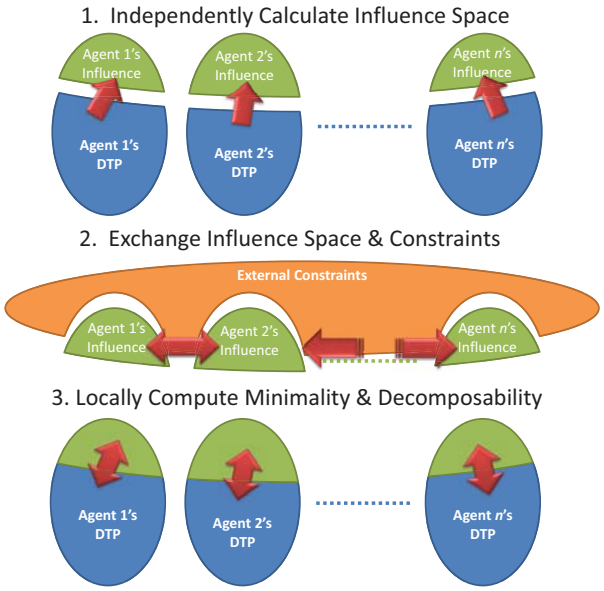


Figure 5: Graphical overview of the MaDTP-LD algorithm.

activities) that, when added to the agent's local DTP, creates a temporal decoupling of MaDTP $\mathbf{D}$. Figure 4 represents a decoupling of the MaDTP in Figure 1, where any solution to agent A's DTP in the top row can be combined with any solution to agent B's DTP in the bottom row to form a joint solution. A challenge is that finding a decoupling requires ensuring that at least one of the solutions to the MaDTP, if any exist, must survive the decoupling (Hunsberger 2002), and so is an NP-hard problem. A second challenge is that disjunctive temporal constraints involve arbitrarily many pairs of variables, and so may induce combinatorially many different network structures, making efficient representation of the set of these possible structures particularly challenging.

Influence-based Decoupling

In this section, we outline our two distributed approaches for calculating the complete and temporally independent MaDTP solution spaces, respectively.

MaDTP Local Decomposability

Our MaDTP-LD algorithm (Boerkoel and Durfee 2012) for computing the complete set of MaDTP solution spaces provides a basis for our decoupling algorithm. The key insight of MaDTP-LD is that not all local solutions qualitatively change how an agent's problem will impact other agents. Thus, instead of enumerating *all* joint component MaSTPs, an agent i can instead focus on enumerating its local component STPs that lead to distinct STP projections over its ***interface***—the portion of its local problem that is involved in external constraints. An agent's ***influence space*** (Oliehoek, Witwicki, and Kaelbling 2012) summarizes how its local constraints impact other agents so that all coordination can be limited to these smaller influence spaces. As illustrated in Figure 5, the MaDTP-LD operates in three distinct phases: (1) each agent

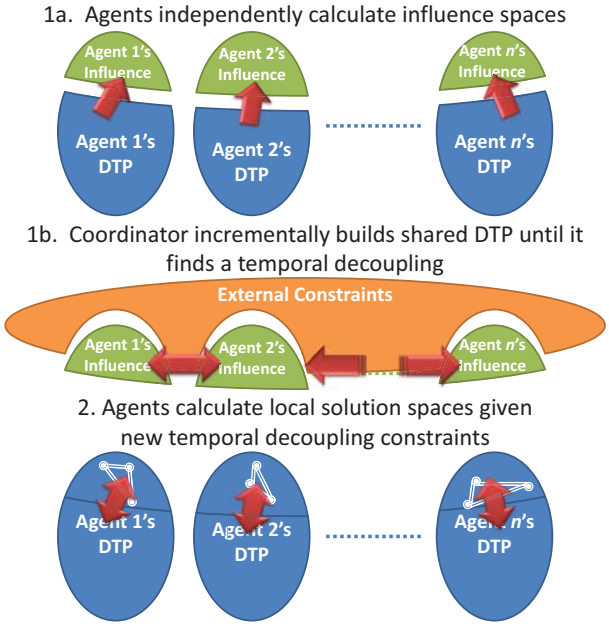


Figure 6: Graphical overview of the MaDTP-TD algorithm.

independently enumerates its influence space; (2) then agents exchange their influence spaces, incorporating the influence spaces of other agents as new local constraints; and (3) finally, each agent independently enumerates its local solution space while respecting the influence space constraints of all agents. The joint solution space is represented in a distributed fashion as a cross-product of local solution spaces and allows agents to independently manage their local solution spaces.

MaDTP Temporal Decoupling

Our MaDTP Temporal Decoupling (MaDTP-TD) algorithm (Boerkoel and Durfee 2013a; 2013b) builds on the MaDTP-LD algorithm. Our approach differs from the MaDTP-LD algorithm by incorporating information from the shared DTP as early and often as possible, rather than waiting for each agent to completely enumerate its local influence space before shared reasoning occurs. Incorporating shared information has the effect of pruning globally infeasible schedules from an agent's local search space early on and then, once a temporal decoupling as been found, short-circuiting agents' reasoning by eliminating those local schedules that are no longer consistent with respect to the new decoupling constraints. The shared DTP solution space can be thought of as the cross-product of agents' influence spaces. Thus, as agents construct their local influence spaces, they can also build the shared DTP solution space in a way that is provably sound and progressively more complete over time. Then, as soon as a solution to the shared DTP is found, it can be used to construct and install a temporal decoupling, which in turn saves computing the entire joint solution space, representing a potentially combinatorial savings. Our approach is both provably sound and complete.

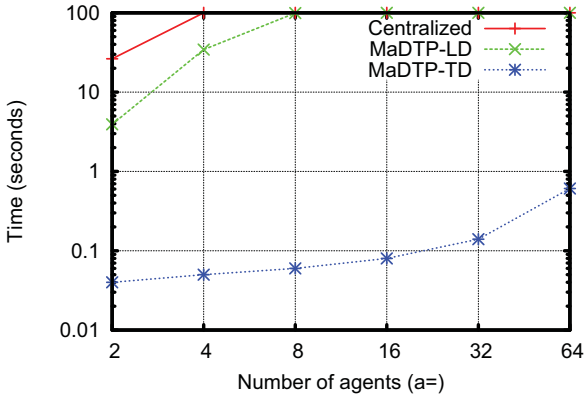


Figure 7: Runtime performance of MaDTP-LD vs. MaDTP-TD as the number of agents scales.

Comparison

We compare our new decoupling approach against our MaDTP-LD algorithm that computes the entire joint solution space, replicating our previous experimental setup (Boerkoel and Durfee 2012). We measure the maximum processing time across agents (i.e., the time the last agent completes execution) and the number of distinct, consistent local temporal constraint networks. Using a 100 second timeout on loosely-coupled problems ($p = 0.2$), Figure 7 shows that using our distributed MaDTP-LD algorithm to compute a complete, locally decomposable joint solution space runs in less orders-of-magnitude less time than a naive centralized approach that computes full joint decomposabilty, which exceeds 100 seconds for problems with more than two agents. Figure 7 also shows that using our distributed MaDTP-TD algorithm to compute a temporally decoupled solution space scales to problems involving orders-of-magnitude more agents than the MaDTP-LD algorithm. As discussed next, this is achieved by sacrificing the completeness of the joint solution space in favor of increased agent independence.

As shown in Figure 8, for problems containing just two agents, as the proportion of external constraints increases (p), our MaDTP-TD algorithm demonstrates upwards of a four orders-magnitude decrease in runtime over the complete MaTDP-LD algorithm. This is because as soon as agents find a decoupling, they immediately commence with finding only solutions that are consistent with the new decoupling constraints rather than fully enumerating the entire joint solution space. However, Figure 8 also illustrates that these gains come at the cost of limiting the completeness of local solution spaces as measured by the number of distinct consistent local temporal networks. This limits the amount of flexibility an agent has to react to scheduling disturbances.

Discussion

In conclusion, we discuss a new distributed, decoupling approach for calculating solution spaces to MaDTPs where agents independently and incrementally build their influence spaces until a valid temporal decoupling can be found. Overall, we believe the gains in runtime efficiency of our MaDTP-

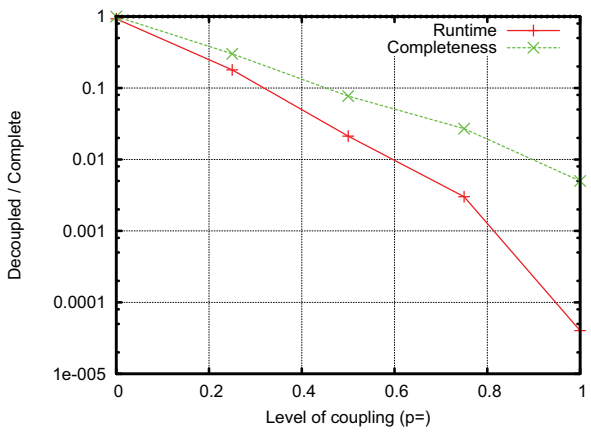


Figure 8: Relative gains in runtime compared to relative loss of completeness of the MaDTP-LD algorithm compared to MaDTP-TD algorithm as the level of interaction coupling increases.

TD algorithm over the MaDTP-LD algorithm outpace the relative sacrifices in solution space completeness—our approach solves loosely-coupled problems containing 64 agents in under a second while maintaining at least a tenth of all consistent local temporal networks, whereas the MaDTP-LD algorithm consistently exceeds 100 seconds for problems with more than four agents. In the future, we would like to investigate optimal and heuristic variants of our decoupling approach where, for example, agents produce influence spaces in a best-first manner in an attempt to guide the coordinator to a more flexible temporal decoupling in an anytime manner.

Acknowledgments

We thank the anonymous reviewers for their suggestions and Professor Julie Shah and the Interactive Robotics Group for their guidance. This work was supported, in part, by the NSF under grant IIS-0964512 and by a UM Rackham Fellowship.

References

Boerkoel, J., and Durfee, E. 2011. Distributed algorithms for solving the multiagent temporal decoupling problem. In *Proc. of AAMAS-11*, 141–148.

Boerkoel, J., and Durfee, E. 2012. A distributed approach to summarizing spaces of multiagent schedules. In *Proc. of AAAI-12*, 1742–1748.

Boerkoel, J., and Durfee, E. 2013a. Decoupling the multiagent disjunctive temporal problem. In *Proc. of AAAI-13*, To Appear.

Boerkoel, J., and Durfee, E. 2013b. Decoupling the multiagent disjunctive temporal problem (extended abstract). In *Proc. of AAMAS-13*, To Appear.

Hunsberger, L. 2002. Algorithms for a temporal decoupling problem in multi-agent planning. In *Proc. of AAAI-02*, 468–475.

Oliehoek, F. A.; Witwicki, S. J.; and Kaelbling, L. P. 2012. Influence-based abstraction for multiagent systems. In *Proc. of AAAI-12*, 1422–1428.

Stergiou, K., and Koubarakis, M. 2000. Backtracking algorithms for disjunctions of temporal constraints. *Artificial Intelligence* 120(1):81–117.

Tsamardinos, I., and Pollack, M. 2003. Efficient solution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence* 151(1-2):43–89.

A Constraint Programming Approach to Solve Scheduling Problems under Uncertainty

Laura Climent, Richard J. Wallace, Miguel A. Salido and **Federico Barber**
Instituto de Automática e Informática Industrial, Universidad Politécnica de Valencia, Spain.
Email: lcliment@dsic.upv.es, msalido@dsic.upv.es and fbarber@dsic.upv.es
Cork Constraint Computation Centre and Department of Computer Science
Western Gateway Building, University College Cork, Ireland. Email: r.wallace@4c.ucc.ie

Abstract

Many real life problems come from uncertain and dynamic environments, which means that the original problem may change over time. Therefore, the original solution may become invalid after these changes. A high percentage of approaches that deal with these problems consider the existence of knowledge about the uncertain and dynamic environment. Nevertheless, for many scheduling dynamic problems, there does not exist extra information about the future possible incidences or it is hard to obtain. In this paper, we extend and improve a CSP approach for obtaining robust solutions in order to apply it to scheduling problems and obtain robust schedules. This approach does not consider extra detailed additional information about the future possible changes. Thus, it assumes that any task of the schedule may undergo a delay. The search algorithm presented assigns values to the variables based on their number of feasible greater contiguous neighbours. They represent slack between tasks, which confers robustness to the schedule because it is able to absorb delays. We have evaluated the proposed approach with open-shop and job-shop benchmarks by using existent robustness measures in the literature that assess the number of time buffers, their duration and/or their distribution.

1 Introduction

Real life problems often evolve over time because the real environments that they come from are often uncertain and dynamic. Thus, the original solution found may no longer be valid after these changes. There are two main approaches for dealing with these situations: *reactive approaches* and *proactive approaches*. For dynamic scheduling the classification can be extended: *reactive* scheduling, *stochastic* scheduling, scheduling under *fuzziness*, *proactive* scheduling and *sensitivity* analysis (see (Herroelen and Leus 2005) for a survey).

Re-solving the problem after the loss of a solution (reactive approaches) consumes computational time. In addition, for scheduling problems there exists another disadvantage because if the new schedule is delivered late, it could cause the shutdown of the production system, the breakage of machines, the loss of the material/object in production, etc. As an alternative, proactive approaches try to avoid these drawbacks by using all the available knowledge about the possible future changes in order to avoid or minimize their effects. That is why these techniques are highly valued for dealing with problems in uncertain and dynamic environments.

For the reasons mentioned above, we focus our attention on the search for robust schedules (proactive approaches), specifically, the ability of a schedule to maximize the chances of resisting changes, since we strongly value solution loss prevention. We also consider situations where there is an added difficulty stemming from the fact that there is no detailed extra information about possible future changes that a scheduling problem may undergo. This is in contrast to most alternative approaches, which require statistics and/or probabilities of changes (for instance stochastic scheduling approaches and scheduling under fuzziness).

If no specific information is given about the dynamics of the situation, it is reasonable to assume that any task of the schedule may undergo a delay due the expected duration of an activity being exceeded or because a required resource has became unexpectedly unavailable. For instance, in scheduling problems, it is reasonable to assume that any task of the schedule may undergo a delay. In (Fu et al. 2012), the authors stated that unexpected external events such as manpower availability, weather changes, etc. lead to delays or advances in completion of activities in scheduling problems.

Example 1. In a simple scheduling problem in which the end-time of a task is 3, all the following tasks in the same job have an start-time in the interval $[3, max]$. However, if in the future there is a delay of 1 time unit in the aforesaid task, it can be assumed that the new domains of the following tasks are $[4, max]$. Figure 1 shows a schedule for this example that has become invalid due to the delay in 1 time unit of the first task T_0 (marked with a mesh) coincide with the start of the following task T_1 (starting in the 3 time unit). For this reason, in the scheduling framework, the slack between tasks confers robustness to the schedule because it is able to absorb task delays.

In this paper we extend and improve the general proactive Constraint Programming approach presented in (Climent et al. 2012) to scheduling problems. This new incorporation mainly consists on: (i) improving the efficiency of the search algorithm for dealing with scheduling problems (problems that usually have large domains) and (ii) adapting the robust-

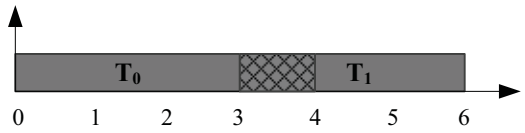


Figure 1: Invalid schedule for Example 1.

ness objective function for scheduling problems. In order to meet our main objective of finding robust schedules with slack that are able to absorb delays, the extended and improved search algorithm computes and maximizes the number of feasible greater contiguous neighbours of the values assigned to the CSP variables (for the CSPs that model the schedules).

The next section recalls some general definitions and robustness measures. Section 3 gives a brief account of earlier procedures related with our approach. Section 4 describes our CSP search algorithm for finding robust schedules. Section 5 presents a case study using a open-shop scheduling benchmark. Section 6 describes experiments with various job-shop scheduling benchmarks. Finally, Section 7 gives conclusions.

2 Technical Background

Here, we give some basic definitions that are used in the rest of the paper, following standard notations and definitions in the literature.

2.1 Constraint Programming

Constraint programming (CP) is a powerful paradigm that has being applied with success to many domains such as scheduling, planning, configuration, networks, etc.

Definition 1. A *Constraint Satisfaction Problem* (CSP) is represented as a triple $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where:

- $\mathcal{X}$ is a finite set of variables $\mathcal{X} = \{x_1, x_2, ..., x_n\}$.
- $\mathcal{D}$ is a set of domains $\mathcal{D} = \{D(x_1), D(x_2), ..., D(x_n)\}$ such that for each variable $x_i \in \mathcal{X}$, $D(x_i)$ is a set of values that the variable can take.
- $\mathcal{C}$ is a finite set of constraints $\mathcal{C} = \{C_1, C_2, ..., C_e\}$ which restrict the values that the variables can simultaneously take.

The CSP modeling of an scheduling problem usually consists of associating a variable with each start or end time of a task (in this paper we use the start time), the domains associated to each variable represent the possible time units, and therefore by means of them it is possible to fix a maximum desired makespan. Finally, the duration of the tasks and their order (if there exists) can be fixed by means of the CSP constraints.

2.2 Robustness and its measurement

In this subsection we formally explain the feature of robustness associated to solutions (and by extension to schedules) and also how to measure the robustness of the schedules.

Definition 2. The most robust solution within a set of solutions is the one with the highest likelihood of remaining a solution after any type of change.

Note that this concept has a strong dependency with respect to the assumption of the future possible changes that may occur. In addition, Definition 2 does not consider modifications in the original solution but only its resistance to future changes in the problem.

In the following we introduce several criteria for measuring scheduling robustness. There are two main factors that increase the capability of the schedule to absorb unexpected delays in its activities: the number of buffers and their duration. Since longer delays in a task subsume smaller delays in the same task, we will assume that shorter delays are more likely to occur over all the tasks. For this reason, we strongly prefer that the slack be uniformly distributed across the whole schedule. The simplest way of determining this feature is calculating the number of time buffers.

However, ideally, the slack should be as long as possible because the longer the buffers are, the longer delays are able to absorb. For this reason another straight-forward robustness measurement was proposed in (Leon, Wu, and Robert 1994) as the slack average in the schedule.

The combination of the duration of the buffers and their distribution along the schedule provides another robustness measure (R), which is a slight variant of a measure introduced in (Surico et al. 2008) that consists in maximizing the slack average (avg) and minimizing their standard deviation (std).

$$R = avg(slack) - \alpha\ std(slack) \quad (1)$$

In (Surico et al. 2008), the authors state that α should be in [0.2,0.25].

3 Limitations of earlier techniques

A high percentage of earlier proactive approaches use additional information about the uncertain and dynamic environment and usually involve probabilistic methodologies. Since our approach is searching for robust solutions for CSPs, in the following we mainly explain the limitations of other approaches of this type. For a survey about specific approaches for scheduling problems that do not involve CSP approaches see the survey (Herroelen and Leus 2005).

In one example of proactive approach that use additional information, data is gathered in the form of penalties, in which values that are no longer valid after changes in the problem are penalized (Wallace and Freuder 1998). On the other hand, in the Probabilistic CSP model (PCSP) (Fargier and Lang 1993), there exists information associated to each constraint, expressing its probability of existence.

Other techniques focus on the dynamism of the variables of the CSP. For instance, the Mixed CSP model (MCSP) (Fargier, Lang, and Schiex 1996), considers the dynamism of certain *uncontrollable* variables that can take on different values of their uncertain domains. The Uncertain CSP model (UCSP) is an extension of MCSP, whose main innovation is that it considers continuous domains (Yorke-Smith and Gervet 2009). The Stochastic CSP model (SCSP)

(Walsh 2002) assumes a probability distribution associated with the uncertain domain of each uncontrollable variable. The Branching CSP model (BCSP) considers the possible addition of variables to the current problem (Fowler and Brown 2000). For each variable, there is a gain associated with an assignment.

In most of these models as well as in a high percentage of techniques for scheduling that are not CSP approaches, it is necessary to have a list of the possible changes or the representation of uncertainty, often in the form of an associated probability distribution. As a result, these approaches cannot be used if the required information is not known. In many real problems, however, knowledge about possible further changes is either limited or nonexistent.

Previous work in the search of robust solutions for general CSPs without extra data about the future possible changes, by mean of reformulation approaches, was done in (Climent et al. 2013). However, it was not adapted and applied to scheduling problems. On the other hand, there exist model reformulation techniques that do not consider extra data about the dynamism over the schedule. Thus, they also search for schedules with slack. This is achieved by adding two variables to each original variable (the variables that represent the start time of the tasks). One variable represents the slack that is following the task and the other variable represents the sum of the slack and the original starting time. As instance, let p_i be the starting time of the task x_i. Thus, we would replace it by $p'_i = p_i + s_i$, where s_i represents the slack associated to task x_i. In addition, depending on the maximum bound of the slack desired, another constraint should be added, such as $s_i \leq k$. In this case, the delay is up to k time units. In addition, an objective function that express the goal of maximizing the total slack should be defined.

The main advantage of the approach presented in this paper over the above mentioned approaches, for whose models the variables represent slack, is that our approach can be applied when all slack-values require a consistency check. This requirement is necessary in scheduling problems where intermediate non-valid slack values are possible. Examples of this type of problem are scheduling problems with limited machine availability (see for instance (Schmidt 2000)). In these cases, some machines are unavailable in certain time intervals; for this reason, tasks that require these resources cannot be executed in such time units. We would like to note that the model reformulation techniques do not check the satisfiability of intermediate slack values.

There is an existing proactive CSP approach that does not consider detailed additional information. However, this approach does not check the satisfiability of intermediate slack values. It searches for *super-solutions* (Hebrard 2006), which are solutions that can be repaired after changes occur, with minimal changes that can be specified in advance. In Section 7 there is a comparison of the schedules obtained by our technique with the schedules obtained by this technique. This motivates the following detailed explanation of the aforementioned approach.

Definition 3. A solution is a *(a, b)-super-solution* if the loss of values of a variables at most, can be repaired by assigning other values to these variables, and changing the values of b variables at most (Hebrard 2006).

For CSPs, the main focus has been on finding (1,0)-super-solutions (because of the high computational cost of computing $b > 0$ or $a > 1$). This is one reason why we analyze this particular super-solution case in this paper. The other reason is motivated by (Verfaillie and Jussien 2005), where the authors state that a desirable objective is to "limit as much as possible changes in the produced solution", which motives the search of (a, 0)-super-solutions. In the following the (1,0)-super-solutions are further explained by a simple example.

Example 2. Let us consider the following CSP:

$x_0, x_1 \in \{1, 2, 3\}$
$C_1 : x_0 \leq x_1$

- The solution $(x_0 = 1, x_1 = 1)$ is not a (1,0)-super-solution, because if the variable x_0 loses the value 1, it is not possible to find another value for x_0 that is consistent with C_1, since $(x_0 = 2, x_1 = 1)$ and $(x_0 = 3, x_1 = 1)$ are not solutions to the problem.
- The solution $(x_0 = 1, x_1 = 2)$ is a (1,0)-super-solution, because if any variable loses its value, at least one value can be found that is compatible with the assignment of the other variable. If x_0 loses its value 1, a value of 2 can be assigned to x_0, since $(x_0 = 2, x_1 = 2)$ is solution of the problem. If x_1 loses its value 2, a value of 1 or 3 can be assigned to x_1, since $(x_0 = 1, x_1 = 1)$ and $(x_0 = 1, x_1 = 3)$ are solutions of the problem.

However, finding (1,0)-super-solutions is problematic because, (1) if there is a *backbone variable* (a variable that takes the same value in all the solutions), this ensures that there are no (1,0)-super-solutions, (2) in general, it is unusual to find (1,0)-super-solutions where all variables can be repaired. For these reasons, in (Hebrard 2006) the author also developed a *branch and bound*-based algorithm for finding solutions that are close to (1,0)-super-solutions, i.e., where the number of repairable variables is maximized (also called maximizing the (1-0)-repairability).

4 Robustness Search

In this section, we explain our main objective and search algorithm for obtaining robust schedules.

4.1 Main Objective

As stated, ideally a robust schedule should exhibit well distributed slack, so that the likelihood that it can absorb delays and still remain valid is high. To meet this objective, we extend a CSP approach that searches for robust solutions for CSPs (Climent et al. 2012) to scheduling problems. For the general CSP robust search, the authors stated that robust solutions are surrounded by feasible neighbours. Nevertheless, in scheduling problems, there is a feature inherent to the structure of the modeled CSP to be considered: the domain values represent time units. This fact implies that lower neighbour values cannot be used for repairing a variable breakage (when there is a delay in a task) because this time unit has already occurred. Thus, if there is a delaying

incident and the time point t is not available, then neither are values lower than t available. Given these particular characteristics, the desirable objective is to search for neighbours that are *greater* than the value assigned. This is illustrated bellow.

Example 3. We consider a toy scheduling problem with two tasks: T_0 and T_1. Both are of 2 time units duration and must be executed in this order. The maximum makespan allowed is 6 time units. In Figure 2 we can see the CSP model associated to the Example 3 and its solution space. The two variables X_0 and X_1 represent the start times of tasks T_0 and T_1, respectively. The domain of both variables (represented with discontinuous lines) is $[0 \dots 4]$, which preserves the maximum makespan fixed to 6 time units (the maximum start time of a task is the maximum makespan minus the duration of the aforesaid task). There exists a constraint controlling the execution order of the tasks (T_0 must start before T_1), which is $C_0 : X_1 \geq X_0 + 2$. There exist 6 solutions (black points).

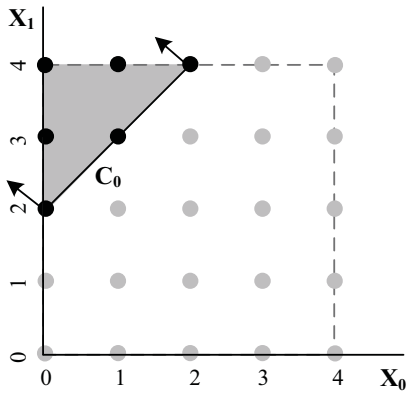
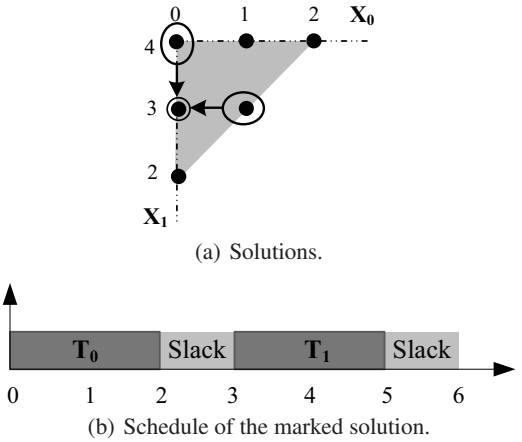


Figure 2: CSP model of Example 3 and its solutions.

If no specific information is given about the dynamic environment, which is the most robust schedule? As stated in Section 2, the greater number of time buffers and the greater their duration is, the more robust the schedule is. But how can we determine which solution of the modeled CSP meets these characteristics? The answer is obtained by calculating the greater feasible contiguous neighbours located at distance lower or equal to k from a solution. The number of greater feasible neighbours associated to each variable, corresponds to the duration of the slack that is located after the task that is represented by this variable. Thus, the slack is able to absorb a delay in the previous task as long as itself, without modifying the other tasks (robustness feature).

In the following (a) figures the greater neighbours are surrounded by a circle and connected to the marked solution. In the following (b) figures are shown the schedules equivalent to the solutions marked in (a) figures. Note that neighbours in (a) figures correspond to the slack in figures (b). For the above example, there exist 3 more robust schedules according to the criteria mentioned above. If we maximize the sum of greater neighbours values that are located at distance 1 ($k = 1$) from each value of the assignment, the solution obtained is shown in Figure 3(a). This solution has one greater neighbour for each value. On the other hand, if we maximize the sum of greater neighbours values that are located at distance $k > 1$ from each value of the assignment, the 3 solutions represented in Figures 3(a), 4(a) and 5(a) are equally robust. The computation of the sums is: $1 + 1$ (Figure 3(a)), $2 + 0$ (Figure 4(a)) and $0 + 2$ (Figure 5(a)).

At this stage, we wonder again which feature is preferable: greater number of time buffers or greater total slack duration? (if both features cannot coexist). The answer is obviously related with the type of dynamism associated to the scheduling problem. Lower k values are performing better for frequent short delays that can occur in any task. However, greater k values can obtain schedules that may face greater delays in the tasks, but these schedules probably will become invalid faced with high frequency delays (because of the lower number of tasks that have slack associated). Since in this paper we do not assume extra detailed information about the future changes, in Section 6 we analyze several ranges of k values.



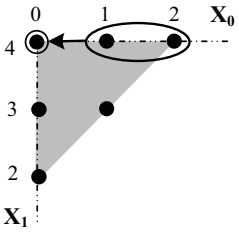
(a) Solutions.

(b) Schedule of the marked solution.

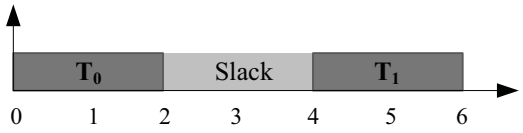
Figure 3: Schedule $S_0 = (x_0 = 0, x_1 = 3)$.

In the following we provide some notation that will be used in further formal definitions. We denote a feasible partial assignment as S, while $\mathcal{X}_S$ is the subset of variables that are involved in S. Thus, $\mathcal{X} \backslash \mathcal{X}_S$ is the set of unassigned variables in S. The value assigned to a variable x in S is denoted as $S(x)$. And $\mathcal{N}_k(x, v, S)$ is the set of feasible greater contiguous neighbour values at distance lower or equal to k from v, which is a feasible value for the variable x. Here, when we say feasible values we mean that they are also feasible with respect to S. We refer the subset $\mathcal{D}_S(x) \subseteq \mathcal{D}(x)$ that is consistent with the feasible partial assignment S. Thus, $\mathcal{N}_k(x, v, S)$ is:

$$\mathcal{N}_k(x, v, S) = \{w \in \mathcal{D}_S(x) : w > v \wedge w - v \leq k \wedge \forall j \in [1 \dots (w - v - 1)](v + j) \in \mathcal{D}_S(x)\} \quad (2)$$

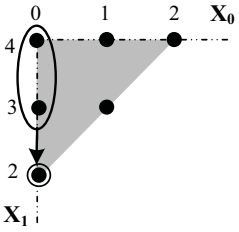


(a) Solutions.

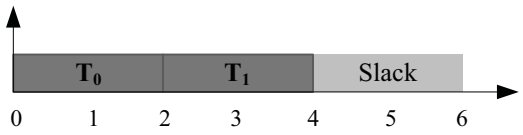


(b) Schedule of the marked solution.

Figure 4: Schedule $S_1 = (x_0 = 0, x_1 = 4)$.



(a) Solutions.



(b) Schedule of the marked solution.

Figure 5: Schedule $S_2 = (x_0 = 0, x_1 = 2)$.

The first condition checks that the feasible neighbours values are greater to the value v, the second condition checks that their distance from v is lower or equal to k. The third condition ensures that all the greater values that are closer to v than w are also feasible values (contiguity condition). If at least one of them is not, the value w does not belong to $\mathcal{N}_k(x, v, S)$. The set of feasible greater neighbours has to be contiguous because otherwise, it means that there exists a task that is making these intermediate values infeasible and therefore they do not represent a slack in the schedule.

The objective function is the total number of greater feasible contiguous neighbour values of a partial assignment S (Equation 3) and it is calculated by summing the size of $\mathcal{N}_k(x, v, S)$ (denoted $|\mathcal{N}_k(x, v, S)|$) for each variable $x \in \mathcal{X}$. If S is an incomplete assignment, the maximum $|\mathcal{N}_k(x, v, S)|$ for each $v \in \mathcal{D}_S(x)$ of the unassigned variables is calculated (upper bound). Since the maximum size of the set of neighbour values is k, it is not necessary to check all the values of $\mathcal{D}_S(x)$ in case that one has k neighbours.

$$f(S,k) = \{ \sum_{x \in \mathcal{X} \setminus \mathcal{X}_S} max\{|\mathcal{N}_k(x, v, S)|, \forall v \in \mathcal{D}_S(x)\} + \sum_{y \in \mathcal{X}_S} |\mathcal{N}_k(y, S(y), S)|\} \quad (3)$$

Next, we give a formal rationale for using the total number of greater feasible contiguous neighbours of the solution (sum of greater feasible contiguous neighbours of each value of the solution) as a measure of robustness.

For $k = 1$, each value has either zero or one neighbour. Here we can discount the case of zero neighbours because if an assignment has zero neighbours, then it must be part of a singleton domain, and it will be part of all solutions. So we need only consider values with one neighbour. In this case, a solution with a greater sum is one whose assignments have more neighbours.

Proposition 1. If we assume that having one neighbour confers greater robustness than not having any and that the probabilities of single changes are independent, then a solution with a greater neighbour-sum than another will also be more robust, and vice versa.

Note that as the number of variables in the problem increases, it becomes increasingly unlikely that a variable with an assignment having zero neighbours will be associated with the largest neighbour-sum for the remaining variables.

In the non-convex case, it is unfortunately possible for one assignment to have zero neighbours, while other assignments to the same variable have one or two. In this case, we cannot assume Proposition 1. However, as the number of variables in the problem increases, it becomes increasingly unlikely that a variable with an assignment having zero neighbours will be associated with the largest neighbour-sum for the remaining variables.

4.2 Search Algorithm

In this section we extend the search algorithm that searches for robust solutions for CSPs (Climent et al. 2012) to scheduling problems. In that paper, the authors applied the general CSP search algorithm to a toy scheduling problem (4 jobs). However, for dealing with real life problems, some efficiency techniques have to be added to the search algorithm. Furthermore, the objective function of the search algorithm presented in (Climent et al. 2012) valued lower feasible neighbours, even if as stated in Section 4.1 these values represent time that has already passed and therefore can not be considered for the robustness computing. The extended search algorithm consists of a Branch & Bound algorithm (Algorithm 1) (B&B-$\mathcal{N}_k$) that maximizes the objective function $f(S, k)$ (see Equation 3). This algorithm is an 'anytime' algorithm that prunes the branches whose objective function value is lower or equal to the maximum function value obtained (f_{Max}). We have improved it by computing the maximum possible objective function value, which is the maximum number of neighbours for each variable multiplied by the number of variables of the CSP. This value is denoted as uB (upper bound). Thus, if the objective

function value of a new solution found is equal to the upper bound, the algorithm stops, since this solution is optimal.

The most important efficiency improvement that we have adopted is the Geometric restart strategy (Walsh and others 1999) in order to reduce the repetition of failures in search due to early wrong assignments (*thrashing*). Thus, each time that the number of failures (referenced as nbF) reaches a cutoff value (referenced as C), a condition that is checked in Algorithm 3, the algorithm restarts the search from scratch, while maintaining the constraints weights stored by the $dom/wdeg$ variable selection heuristic (Boussemart et al. 2004). The value of the number-of-fails cutoff is increased geometrically in Algorithm 1 according to a scaling factor (referenced as $scale$) and a multiplicative factor (referenced as m). We have implemented two different options to carry out when a solution is found. In the first restarting option, called *restarting-completion*, when the first solution is found, the algorithm continues the search until completion (note that in this case we assign a huge number representing ∞ to the number of fails cutoff). In the second restarting option, called *restarting-scratch*, after each solution found, the algorithm restarts the search from scratch and also restarts the number of fails cutoff computation (the constraint weights remain the same). For instances whose domain size is large, this restarting option could be useful because it avoids spending a large amount of time in a specific branch. The latter happens when Algorithm 1 checks many domain values of variables located in low positions of the search tree, because the objective function of the partial assignment is better than the maximum found until this moment (f_{Max}). In this case, if there exists a time cutoff, Algorithm 1 could not analyze other branches of the tree, which may contain solutions of good quality.

Algorithm 1: B&B-$\mathcal{N}_k$: Branch and Bound algorithm

Data: $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle, k, scale, m,$ *time cutoff*
Result: $S, \mathcal{N}_k, f_{Max}$
$S \leftarrow \emptyset$; // Partial assignment
$\mathcal{X}_S \leftarrow \emptyset$; // Set of variables assigned
$\mathcal{N}_k \leftarrow \emptyset$; // Set of contiguous surrounding neighbours
$f_{Max} \leftarrow 0$; // Maximum $f(S,k)$ for the solutions
$uB \leftarrow k * |\mathcal{X}|$;
$i \leftarrow 1$;
GAC3-$\mathcal{N}_k(P, S, \mathcal{X}_S, \mathcal{N}_k, k, f_{Max})$;
repeat
  if *restarting-scratch* $\wedge$ *new solution found* **then**
    $i \leftarrow 1$;
  $C \leftarrow scale * m^i$; //number of fails cutoff
  $i \leftarrow i + 1$;
until *time cutoff* $\vee$ ***not***
MGAC3-$\mathcal{N}_k(P, S, \mathcal{X}_S, \mathcal{N}_k, k, f_{Max}, 0, C, uB)$;

The inference process is carried out by Algorithm 2 (GAC3-$\mathcal{N}_k$), which is an extension of the well known GAC3 (Mackworth 1977). Some specific notation has been included, as $Var(c)$, which it is the scope of $c \in \mathcal{C}$. The original $seekSupport$ function of GAC3 searches for a support of each domain value. However, this function has a input parameter the set of values for being analysed. Thus, if any of these values are deleted because there does not exist any consistent support with respect the partial assignment, $seekSupport$ returns *False*. This function is first called with the values of the domain of the variables (for checking if the partial assignment S is GAC3) and later with $\mathcal{N}_k$ just for assigned variables (for checking if each $\mathcal{N}_k(x, S(x), S)$ is GAC3 with respect S). In order to ensure the contiguity of the values in $\mathcal{N}_k$, Algorithm 2 checks the consistency of subsets of $\mathcal{N}_i \subseteq \mathcal{N}_k$, where i is equal to 1 initially, and it is increased in one unit until at least one of the values of $\mathcal{N}_i$ is inconsistent or until i reaches the value of k. After composing the set of contiguous neighbour values that are GAC3 with respect S, Algorithm 2 analyzes if the objective function $f(S,k)$ is greater than f_{Max}. If it is not, or S is not GAC3, returns *false*.

Algorithm 2: GAC3-$\mathcal{N}_k$: Global Arc Consistency

Data: $P, S, \mathcal{X}_S, \mathcal{N}_k, k, f_{Max}, nbF$
Result: $\mathcal{D}, \mathcal{N}_k, nbF$
$Q \leftarrow \{(x,c), \forall c \in \mathcal{C}, \forall x \in Var(c)\}$
while $Q \neq \emptyset$ **do**
  $(x,c) \leftarrow takeElement(Q)$;
  $s\mathcal{D} \leftarrow seekSupport(x, \mathcal{D}(x), c)$;
  if $|\mathcal{D}(x)| = \emptyset$ **then**
    $nbF \leftarrow nbF + 1$; // number of failures
    return *False;*
  if ***not*** $s\mathcal{D}$ **then**
    $Q \leftarrow Q \cup \{(y,c'), \forall c' \in \mathcal{C} \wedge c' \neq c \wedge \forall x, y \in Var(c') \wedge x \neq y\}$
  if $x \in \mathcal{X}_S$ **then**
    $i \leftarrow 1$;
    repeat
      update $\mathcal{N}_i(x, S(x), S)$ with Equation 2;
      $s\mathcal{N} \leftarrow seekSupport(x, \mathcal{N}_i(x, S(x), S), c)$;
      $i \leftarrow i + 1$;
    until $s\mathcal{N} = False \vee i > k$;
    $\mathcal{N}_k(x, S(x), S) \leftarrow \mathcal{N}_i(x, S(x), S)$
return $f(S,k) > f_{Max}$; // See Equation 3

For scheduling problems, whose modeled CSPs have convex domains, Bounds Arc Consistency for discrete CSPs (Lhomme 1993) is used in order to reduce the computational time. The main feature of this consistency technique is that the arc consistency in the value assignment is only applied to the bounds of each convex domain. Thus, including it in the search algorithm only affects to the $seekSupport$ function, which instead of seeking for a support for all the set of values, just checks the minimum and maximum bounds.

Algorithm 3 (MGAC3-$\mathcal{N}_k$) performs a Maintaining GAC3 process by assigning to each variable $x \in \mathcal{X}$ a new value $v \in \mathcal{D}(x)$, until the value selected is GAC3-$\mathcal{N}_k$ with respect S. Furthermore, Algorithm 3 is also responsible for updating the set of assigned variables $\mathcal{X}_S$, the partial assignment S and the maximum objective function value f_{Max} (for each solution found). Furthermore, it stores the domains and set of neighbours of all the variables before mak-

ing an assignment. Note that after a variable x is assigned, $\mathcal{D}(x)$ contains a single value that is the value assigned to x. If Algorithm 2 (GAC3-$\mathcal{N}_k$) returns *false*, then Algorithm 3 (MGAC3-$\mathcal{N}_k$) carries out the backtracking process and also restores the domains and set of neighbours of all the variables.

Algorithm 3: MGAC3-$\mathcal{N}_k$: Maintaining GAC

```
Data: P, S, X_S, N_k, k, f_Max, nbF, C, uB
Result: S, N_k, f_Max
select x ∈ X\X_S; // dom/wdeg heuristic
X_S ← X_S ∪ x;
save D and N_k;
while |D(x)| ≠ ∅ ∧ nbF < C do
    select min(v) ∈ D(x); // Lexicographical value order heuristic
    S ← S ∪ {x = v};
    D(x) ← v;
    if GAC3-N_k(P, S, X_S, N_k, k, f_Max, nbF) then
        if X_S = X then
            // New solution found
            f_Max ← f(S, k);
            if f_Max = uB then
                return True; // Best possible cost achieved
            C ← ∞; // restarting-completion
            return False; // restarting-scratch
        if MGAC3(P, S, X_S, N_k, k, f_Max, nbF, C, uB) then
            return True;
    restore D and N_k;
    S ← S\{x = v};
X_S ← X_S\x;
return False;
```

5 Case Study

In this section a case study of an scheduling benchmark is described to graphically show the robustness of schedules obtained with Algorithm 1) (B&B-$\mathcal{N}_k$). We analyze an instance derived from Taillard optimization problems (Taillard 1993): "os-taillard-4-105-0", which is a well-known problem and was used in the CSP solver competition[1].

This problem was converted to satisfaction problems by fixing the maximum makespan allowed (latest finishing time). For the analyzed instance the maximum makespan is set to 105% of the best makespan, so that it is possible to have schedules with instances of greater slack. Since the open-shop benchmark is composed of 4 machines, 4 jobs and 4 tasks per job, the resultant CSP model contains 16 variables and 48 constraints; the latter prevent two tasks from using the same machine at the same time as well as ensuring that two tasks of the same job do not overlap.

For this benchmark as well as the benchmarks analyzed in Section 6, we have used a Intel Core i5-650 Processor (3.20 Ghz) and we have fixed the time cutoff to 100 seconds. In addition, for the geometric restart, we have fixed the scale factor to 10 and the multiplicative factor to 1.5.

Figure 6 shows a non robust schedule obtained by a CSP solver. The jobs are represented on the vertical axis and time is represented on the horizontal axis. Tasks are shown in light grey with the number of task and the machine assigned to each task. The striped slack represents the natural slack produced because the earlier starting next task related with a task, is waiting to the release of a machine or to the end of another task, or by the gap between the last task of a job and the makespan. It can be observed that this schedule has only 7 natural slack after the tasks, so a delay at any other place in the schedule will invalidate the obtained solution.

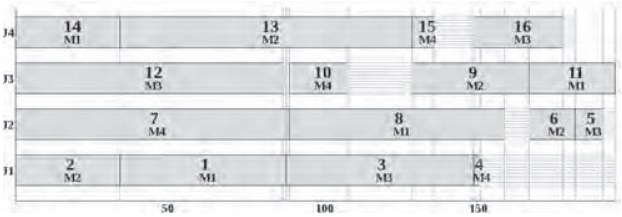


Figure 6: Non robust schedule (makespan=195).

Figure 7 shows the solution obtained for "os-taillard-4-105-0" problem by B&B-$\mathcal{N}_k$ algorithm for $k = 1$. It can be observed that it has additional instances of slack (in dark gray). These slacks are not produced because the earlier starting next task related with a task is not able to start before; for this reason, they are differentiated from the natural slack by denoting them as robust. Because of them, this optimal schedule is more robust than the previous one. It is more robust because more tasks maintain slack to handle short delays. Note that this schedule has 11 robust slacks each of 1 time unit. Therefore, if any task has a delay of 1 time unit the schedule will still be valid. The makespan of this schedule is only 2 units longer than the makespan of the schedule represented in Figure 6 (trade-off between robustness and makespan).

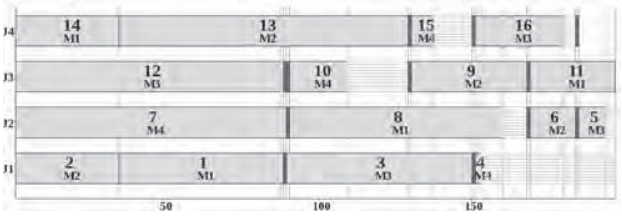


Figure 7: Schedule obtained by B&B-$\mathcal{N}_k$, $k = 1$ (11 robust slack whose sum is 11, makespan=197).

As mentioned, the duration of each robust slack obtained in the previous schedule is 1 time unit. However, by increasing the value of k, Algorithm B&B-$\mathcal{N}_k$ searches for assignments that have up to k greater feasible contiguous neighbours. As a consequence, the durations of the robust slack obtained are up to k time units. Figure 8 shows the schedule

[1] *http://www.cril.univ-artois.fr/ lecoutre/benchmarks.html*

obtained for "os-taillard-4-105-0" problem by B&B-$\mathcal{N}_k$ algorithm for $k = 3$. This schedule has the same makespan as the schedule obtained by B&B-$\mathcal{N}_k$ algorithm for $k = 1$ (see Figure 7), but the sum of robust slack is higher even though it has a smaller number of time buffers. The schedule represented in Figure 8 has 7 cases of robust slack whose sum is 18 time units. They are scattered as follows: 4 slacks of 3 time units and 3 slacks of 2 time units.

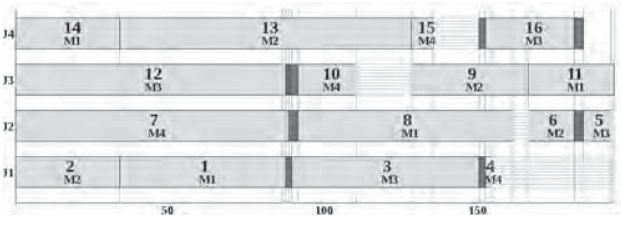


Figure 8: Schedule obtained by B&B-$\mathcal{N}_k$, $k = 3$ (7 robust slack whose sum is 18, makespan=197).

The greater the number and size of robust slacks, the higher is the probability that the rest of the schedule will remain satisfiable after changes. We have observed that schedules obtained with lower k values maximize the number of buffers even if their size is small. In contrast, the computation of higher k values tends to give priority to the sizes of the buffers and as consequence, the number of buffers obtained is sometimes lower.

6 Experimental Results

In this section, we apply the search algorithm presented in this paper to scheduling benchmarks from the literature, in order to evaluate the robustness of schedules obtained for a wide range of k values. We analyzed 5 sets of 10 job-shop CSP instances studied in (Sadeh and Fox 1996). Each instance is composed of 10 jobs of 5 tasks each and there are 5 resources. Each job has a random linear sequence of resources to visit, with the exception of the *bottleneck* resources, which are visited after a fixed number of operations (in order to further increase resource contention). The instances in the XCSP format and the XCSP parser can be found in Christophe Lecoutre's web page [2].

The solutions obtained by B&B-$\mathcal{N}_k$ algorithm for *restarting-completion* are referred as "neighbour solutions" in Figure 9. On the other hand, the solutions obtained by our technique for *restarting-scratch* are referred as "neighbour solutions(R)". We also have analyzed another proactive CSP approach in the literature: finding super-solutions (see Section 3 and (Hebrard 2006)). The main reason for choosing this technique is that it is a proactive approach that can solve general CSPs and that, like our technique, it does not consider extra information about the future possible changes (uncertainty probabilities, statistics, etc.). For calculating solutions that maximize the number of repairable values for (1,0)-super-solutions (referred as "(1,0)-super-solutions"), we modified Algorithm 1 (B&B-$\mathcal{N}_k$) by changing MGAC3-$\mathcal{N}_k$ and GAC3-$\mathcal{N}_k$ algorithms for MAC+ and GAC+ (Hebrard 2006), respectively. The value of the parameter d (duration of the delay) has been fixed to the same value than k (equally conditions). In addition, we analyzed the robustness of a simple schedule obtained by a CSP solver (referred as "simple solutions"). This analysis has not been included as an alternative to our technique obviously, but in order to detect how much natural slack a simple schedule has. In addition we have included the geometric restart and bounds consistency techniques explained in Section 4.2 in both techniques, in order to provide these computational advantages to all the approaches analyzed. For all of them also the time cutoff is fixed to 100 seconds.

Figures 9(b) show the evaluation of 'e0ddr1' benchmark. The rest of the problems sets have not been included due to space limitations but the results were similar, showing the best robustness results for the less constrained benchmarks because here it is more likely that there are buffers of long durations. It can be observed that Algorithm 1 for both restarting options outperformed both other approaches. Furthermore, the analysis of the k/d parameters shows that when these parameters have their lowest values, the number of buffers of the schedules found by our algorithm is markedly greater than for schedules obtained by the simple solver and the super-solution approach (see Figure 9(a)). However, the differences in robustness seems to be quite general. The main disadvantage of the (1,0)-super-solution technique is that it considers as repairable values those that imply a change in the start time of a task so it now follows another task sharing the same resource that it originally followed, which is not equivalent to slack in the schedule.

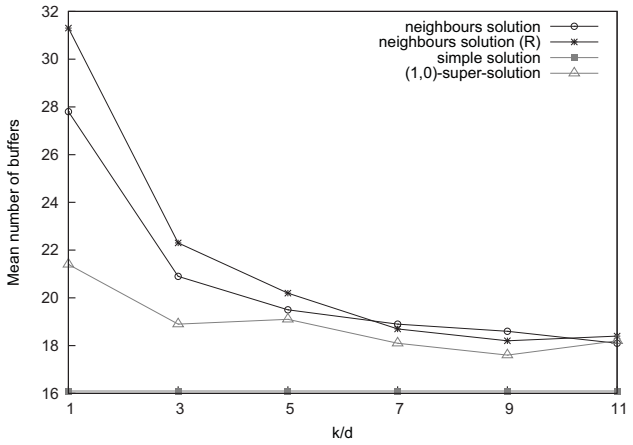
From this analysis, we can conclude that there do not exist strong differences between the two restarting options developed for Algorithm 1, restarting-completion and restarting-scratch. But the latter obtains more robust schedules for lower k values (see $k/d \in [1, 5]$ in Figure 9). For greater k values, apparently, both restarting options provide similar results.

The schedules obtained by Algorithm 1 for the lowest k values had the maximum number of buffers. On the other hand, the robustness measures are greater for the greater k values. Depending on the dynamic nature of the problem, it would be desirable to prioritize between a higher number of time buffer of short duration or a lower number of time buffer of long duration (if the two features can not be both maximized). The more information about the possible future changes and its certainty, the better robustness results we can obtain. However, even if this information is unknown, we can obtain a schedule with certain distributed slack fixing an intermediate value of k in our Algorithm 1.

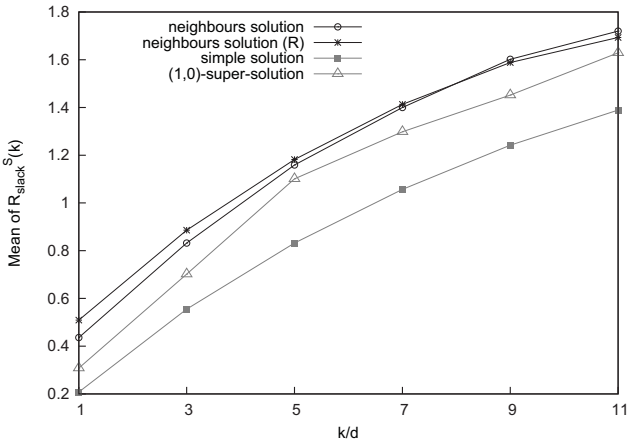
7 Conclusions

In this paper we extend and improve a CSP approach for obtaining robust solutions in order to apply it to scheduling problems and obtain robust schedules for problems that come from uncertain and dynamic environments where the information about the possible future changes is unknown. Thus, there do not exist uncertainty statistics nor probabilities about the incidences that can occur in the schedule. In this context, it is reasonable to assume that the tasks of the

[2] *http://www.cril.univ-artois.fr/ lecoutre/index.html*



(a) Mean number of time buffers



(b) Mean R^S_{slack} values

Figure 9: Robustness analysis for the e0ddr1 benchmark.

scheduling problem may undergo delays due to unexpected extensions of some activities or due to the unforeseen unavailability of some resources. For this reason, the main objective for conferring robustness consists in finding schedules that incorporate some extra slack because they are able to absorb these incidences. In order to achieve these buffer times, we search for solutions that maximize the sum of the greater feasible contiguous neighbours at distance lower or equal to k from the values of the solution. The obtained schedules have a high probability of remaining valid faced with delays in the tasks of the scheduling problem due to the incorporated slack. The number of greater feasible contiguous neighbours associated with a variable is the size of the slack located at the end of the task associated to this variable.

We have described a case study in which we applied our search algorithm to a well known open-shop benchmark. We obtained two schedules with certain robustness that only have 2 time units more of makespan than the best known makespan for the scheduling problem (trade-off between robustness and makespan). In addition, by increasing the k parameter, we greatly increased the total robust slack even though the number of time buffers was lower.

We have compared our approach for searching for robust schedules with another proactive CSP approach in the literature: super-solutions, as well as with a simple CSP solver. The evaluation has been developed with well known job-shop benchmarks. We have shown that our approach can outperform both ordinary CSP algorithms (this dramatically) and algorithms that find (1,0)-super-solutions (or maximize the number of repairable variables in case that there does not exist a (1,0)-super-solution). On the basis of these experiments, we conclude that schedules obtained with lower k values maximize the number of buffers even if their size is small. However, the computation of higher k values tends to give priority to the sizes of the slack and as consequence, the number of buffers obtained might be lower. Therefore, depending on the dynamic nature of the problem (if it is not possible to maximize both features), it would be desirable to prioritize between a higher number of small buffers or a lower number of big buffers.

Our technique can be applied to many real-world scheduling problems where there is an added difficulty in that the environment is not only dynamic but also highly uncertain, because information about the possible future changes is limited or non-existent. In these cases, we provide schedules with certain robustness even under these difficult conditions.

In the future, we plan to extend the search algorithm to handle constraint satisfaction and optimization problems (CSOP). In this case the main objective would be not only to increase the robustness of the schedule but also to decrease its makespan or another critrion. This step would allow the user to chose between the proper trade-off between these criteria, according to the scheduling problem specifications.

8 Acknowledgements

This work has been partially supported by the research projects TIN2010-20976-C02-01 (Min. de Ciencia e Innovación, Spain) and P19/08 (Min. de Fomento, Spain-FEDER), and the fellowship program FPU. We wish to thank Dr. Christophe Lecoutre and Dr. Diarmuid Grimes for their assistance.

References

Boussemart, F.; Hemery, F.; Lecoutre, C.; and Sais, L. 2004. Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-04)*, volume 16, 146.

Climent, L.; Wallace, R.; Salido, M.; and Barber, F. 2012. An algorithm for finding robust and stable solutions for constraint satisfaction problems with discrete and ordered domains. In *24th IEEE International Conference on Tools with Artificial Intelligence (ICTAI-12)*, 874–879.

Climent, L.; Wallace, R. J.; Salido, M. A.; and Barber, F. 2013. Modeling robustness in csps as weighted csps. In *Proceedings of the 10th tenth International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming (CPAIOR-13)*.

Fargier, H., and Lang, J. 1993. Uncertainty in Constraint Satisfaction Problems: a probabilistic approach. In *Proceedings of the Symbolic and Quantitative Approaches to Reasoning and Uncertainty (EC-SQARU-93)*, 97–104.

Fargier, H.; Lang, J.; and Schiex, T. 1996. Mixed Constraint Satisfaction: A framework for decision problems under incomplete knowledge. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, 175–180.

Fowler, D., and Brown, K. 2000. Branching Constraint Satisfaction Problems for solutions robust under likely changes. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP-2000)*, 500–504.

Fu, N.; Lau, H.; Varakantham, P.; and Xiao, F. 2012. Robust local search for solving rcpsp/max with durational uncertainty. *Journal of Artificial Intelligence Research* 43:43–86.

Hebrard, E. 2006. *Robust Solutions for Constraint Satisfaction and Optimisation under Uncertainty*. Ph.D. Dissertation, University of New South Wales.

Herroelen, W., and Leus, R. 2005. Project scheduling under uncertainty: Survey and research potentials. *European journal of operational research* 165(2):289–306.

Leon, V.; Wu, S.; and Robert, H. 1994. Robustness measures and robust scheduling for job shops. *IIE transactions* 26(5):32–43.

Lhomme, O. 1993. Consistency techniques for numeric CSPs. In *Proceedings of 13th the International Joint Conference on Artificial Intelligence (IJCAI-93)*, volume 13, 232–232.

Mackworth, A. 1977. On reading sketch maps. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI-77)*, 598–606.

Sadeh, N., and Fox, M. 1996. Variable and value ordering heuristics for the job shop scheduling Constraint Satisfaction Problem. *Artificial Intelligence* 86(1):1–41.

Schmidt, G. 2000. Scheduling with limited machine availability. *European Journal of Operational Research* 121(1):1–15.

Surico, M.; Kaymak, U.; Naso, D.; and Dekker, R. 2008. Hybrid meta-heuristics for robust scheduling.

Taillard, E. 1993. Benchmarks for basic scheduling problems. *European Journal of Operational Research* 64(2):278–285.

Verfaillie, G., and Jussien, N. 2005. Constraint solving in uncertain and dynamic environments: A survey. *Constraints* 10(3):253–281.

Wallace, R., and Freuder, E. 1998. Stable solutions for Dynamic Constraint Satisfaction Problems. In *Proceedings 4th International Conference on Principles and Practice of Constraint Programming (CP-98)*, 447–461.

Walsh, T., et al. 1999. Search in a small world. In *International Joint Conference on Artificial Intelligence*, volume 16, 1172–1177.

Walsh, T. 2002. Stochastic Constraint Programming. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-02)*, 111–115.

Yorke-Smith, N., and Gervet, C. 2009. Certainty closure: Reliable constraint reasoning with incomplete or erroneous data. *Journal of ACM Transactions on Computational Logic (TOCL)* 10(1):3.