, , 1–30 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Probe Backtrack Search for Minimal Perturbation in Dynamic Scheduling

HANI EL SAKKOUT & MARK WALLACE {hhe,mgw}@icparc.ic.ac.uk
IC-Parc, Imperial College, London

Editor: Claude Le Pape, Christopher Beck, Andrew Davenport

Abstract.
This paper describes an algorithm designed to minimally reconfigure schedules in response to a changing environment. External factors have caused an existing schedule to become invalid, perhaps due to the withdrawal of resources, or because of changes to the set of scheduled activities. The total shift in the start and end times of already scheduled activities should be kept to a minimum. This optimization requirement may be captured using a linear optimization function over linear constraints. However, the disjunctive nature of the resource constraints impairs traditional mathematical programming approaches. The *unimodular probing* algorithm interleaves constraint programming and linear programming. The linear programming solver handles only a controlled subset of the problem constraints, to guarantee that the values returned are discrete. Using *probe backtracking*, a complete, repair-based method for search, these values are simply integrated into constraint programming. *Unimodular probing* is compared with alternatives on a set of dynamic scheduling benchmarks, demonstrating its effectiveness.

In the final discussion, we conjecture that analogous *probe backtracking* strategies may obtain performance improvements over conventional backtrack algorithms for a broad range of constraint satisfaction and optimization problems.

Keywords: Scheduling, Constraint Satisfaction, Constraint Programming

1. Introduction

1.1. *Overview*

This paper considers a dynamic variant of the classic scheduling problem. The aim is to restore consistency in a schedule disturbed by resource or activity changes. To reduce disruption, the new schedule should keep changes to a minimum.

We consider only non-preemptive scheduling problems, where activities are not interruptible. We also restrict our study to reusable resources, which become available again for allocation once activities terminate (e.g. machines and containers). Dynamic variants of these NP-hard scheduling problems are theoretically and commercially significant. Our research into this subject was initially motivated by the need for an effective solution strategy to a commercial aircraft scheduling problem.

The kernel resource feasibility problem (KRFP), is introduced here to capture this scheduling problem and many others. The KRFP generalizes most scheduling benchmarks, including job-shop and resource constrained project scheduling. Minimal perturbation algorithms are compared in this paper on dynamic KRFPs.

The strategy proposed, *unimodular probing*, solves scheduling problems using a very close integration of constraint programming search with linear programming optimization. It is a complete tree search algorithm that, at each search node, implements a repair step. The repair is to the *super-optimal* assignment (i.e. better with respect to the optimization function than the *feasible* optimal, but only partially consistent) that is generated by the linear optimizer. The constraint subset handled by the linear optimizer is chosen so that its solutions are always discrete (integer), facilitating their integration into constraint programming search and enabling the application of the algorithm to discrete scheduling problems.

The strength of the strategy is demonstrated by comparisons on random benchmark tests. The tests are designed to capture the essence of the commercial scheduling application that motivated our research. Furthermore, the type of schedule disturbance inflicted in these tests (newly created resource contention) is the crux of dynamic scheduling.

The strategy's effectiveness raises the more general issue of perturbation minimization in constraint satisfaction problems, and suggests that the complete repair-based search strategy utilized, *probe backtracking*, may be useful even when the cost function precludes the use of linear programming. In fact, probe backtracking can be used to support constraint satisfaction methods on problems without an optimization requirement. The paper remarks on this important issue.

1.2. Paper Structure

Section 2 starts with a toy example, and discusses the commercial importance of dynamic scheduling. The features of a minimal perturbation problem and a KRFP are then detailed in Sect. 3. The following section outlines a constraint model for the KRFP. A particular minimal perturbation variant of the KRFP, the resource utilization problem (RUP), is introduced in Sect. 5. This is used to illustrate and compare the algorithms throughout the paper.

The second part of the paper describes a range of algorithms and tests them. Section 6 studies the temporal sub-problem of the RUP and methods for solving it. The following section introduces the algorithms for minimal perturbation. Unimodular probing is compared empirically with more traditional constraint programming algorithms, and with mixed integer programming search in Sect. 8. Section 9 details where it sits in the literature. The final section discusses the utility of unimodular probing and probe backtracking in general.

2. Motivation for Dynamic Scheduling

2.1. A toy example

EXAMPLE: Figure 1 shows a toy dynamic scheduling problem. In the upper segment of the figure, three activities (A, B and C) are scheduled. The three utilize identical resources. Two temporal constraints, relating several of the start and end times of the activities, are satisfied in the current schedule.

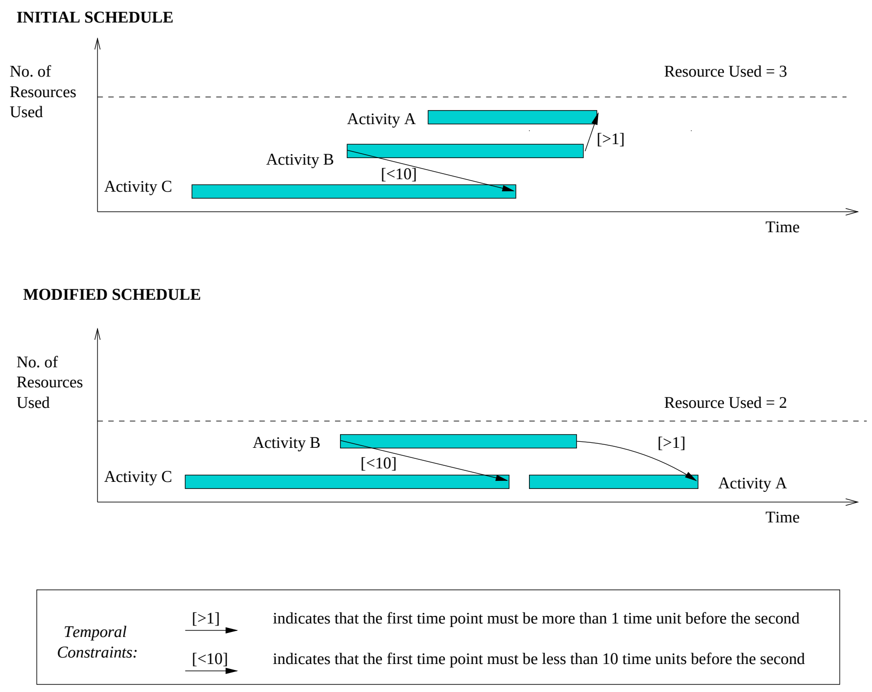

Figure 1. Example of a schedule modified to reduce resources

In the second segment the figure shows the result of rearrangements in response to a reduction the number of available resources. The modified schedule remains consistent with respect to the temporal constraints of the problem.

□

This simple example shows the essence of dynamic scheduling. In this paper, the aim is to find a modified schedule that "minimally" differs from a schedule that has become inconsistent.

2.2. Commercial Application

There is a strong commercial demand for techniques that manage better the process of schedule development. At the beginning of the scheduling process schedule requirements are often vague, becoming well-defined only as the time of execution draws closer. Initial schedules are often created only to be overtaken by newly emerging detail and unforeseen events. Moreover, organizations exhibit inertia; changes must be agreed with all affected parties, and activities and resources may have attributes obstructing immediate re-allocation. Disruption must be minimized as the schedule evolves.

Some research has focused on improving schedule **robustness** (see for instance [23] and [18]). The goal was to avoid changes to the working schedule by generating robust schedules (or high level scheduling decisions) that are likely to remain valid after minor changes to the problem definition. Nevertheless, major changes usually necessitate modifications to even very robust schedules. Other research has investigated **dynamic scheduling**, where schedules are restored to feasibility when change is inevitable (e.g. reactive scheduling in ISIS [13] and Micro-Boss [32]). [5] surveys research into robustness, dynamic scheduling and other related areas.

In this paper we focus on dynamic scheduling, formulated as an optimization problem: in addition to restoring feasibility, the new schedule must differ minimally from the previous schedule.

Algorithms for dynamic scheduling deal with three forms of change resulting in schedule inconsistency. These are outlined below [with examples in brackets].

Activity changes. Requests for new or extended activities may lead to resource contention and schedule re-arrangements. In long-term scheduling, activities may be introduced to improve schedule efficiency, and make better use of under-utilized resources [temporarily leasing out cranes in construction scheduling]. In short term scheduling, new activities are introduced as they arise [just in time orders in outsourced manufacturing], sometimes even during schedule execution [emergency services scheduling]. The durations of activities may be extended or their levels of resource usage increased for similar reasons.

Resource changes. Resource reductions also introduce schedule inconsistency. They can also be requested over the long-term to reduce schedule costs [aircraft utilization problems]. Shorter term rescheduling is often in response to resource failures [out-of-service machines in job shop scheduling] that limit resource availability over all or part of the schedule horizon.

Temporal changes. The simplest form of temporal change is a contraction of the schedule horizon. Other temporal changes can also result in schedule inconsistency, and may occur in the long term [bus services re-arranged for regularity] or the short term [downstream effects of missed activity deadlines in manufacturing].

While the unimodular probing strategy was originally developed to tackle resource reductions in aircraft utilization problems, the algorithms described handle all the above kinds of change, as detailed in Sect. 5.

3. Problem Definitions

3.1. The Minimal Perturbation Problem

Minimal perturbation problems are introduced to formalize the requirement for minimally disruptive rescheduling. They are particularly useful for scheduling problems where activities should be kept close to times when they are most needed (e.g.

rush hour trips in bus scheduling). Minimal perturbation problems correspond to the points where a solution (schedule) must change because the problem has been redefined. An old problem and a solution to it are given, as well as the required changes to the problem definition. Perturbation is modeled by a function that measures the difference between the old and new solutions. New solutions should minimize perturbation as measured by this function.

The scheduling problem is represented here as a *constraint satisfaction problem* (CSP). A CSP Θ is a 3-tuple (V, D, C) where V is a set of n variables $\{v_1, .., v_n\}$; D is a function mapping the variables to n corresponding domains of values $d_1, .., d_n$; and C a set of constraints (relations) on subsets of the variables in V. Let α be a complete assignment, mapping each variable $v_i \in V$ to a value in its corresponding domain d_i. α is a solution to Θ if and only if each constraint in C holds under α.

Problem redefinitions are captured as *dynamic CSPs* (DCSPs). A DCSP is a sequence of CSPs, where each one differs from its predecessor by constraint additions or deletions [6]. Let us consider the case of a CSP Θ transformed into a new CSP Θ'. The transformation is captured in terms of two sets C_{del} and C_{add}, respectively standing for the set of constraints about to be removed from, and added to the constraints C of Θ (denoted C_Θ).

An optimal solution to a *minimal perturbation problem* is a solution to the altered CSP that it is "minimally different" to a previously satisfactory solution.

Definition 1. **Minimal Perturbation Problem**

A *minimal perturbation problem* Π is a 5-tuple $(\Theta, \alpha_\Theta, C_{\text{del}}, C_{\text{add}}, \delta)$ where:

- Θ is a CSP;
- α_Θ is a solution to Θ;
- C_{del}, C_{add} are constraint removal and addition sets;
- δ is a function that evaluates the difference between two complete assignments. It is used to measure perturbation.

A complete assignment is a *solution* to Π if and only if it is a solution to $\Theta' = (V, D, C_{\Theta'})$, where $C_{\Theta'} = (C_\Theta \setminus C_{\text{del}}) \cup C_{\text{add}}$. A solution α_Π is *optimal* if and only if $\delta(\alpha_\Pi, \alpha_\Theta)$ is minimal.

3.2. Introduction to the KRFP

The kernel resource feasibility problem (KRFP) is a generic model for scheduling. It also represents the widest class of problems to which the algorithms presented may be applied without extension.

In the KRFP the goal is to fix the start and end times of activities such that quantities of available resource are not over-allocated. The decision variant of the KRFP is an NP-complete problem (it is an extension of the NP-complete resource constrained scheduling problem [15]). The KRFP allows the expression of variable duration activities and arbitrary metric constraints relating activity start and end times. The metric constraints can capture many of the complex temporal requirements that exist in real-world scheduling problems.

The KRFP is in fact the scheduling "kernel" of the resource feasibility problem (RFP) of planning [9]. Unlike the RFP, the KRFP does not allow the representation of several resource pools of the same type, which means that every activity is committed to a single resource or resource pool. This restriction avoids the introduction of resource assignment sub-problems into the KRFP. Nevertheless, the algorithms described in this paper may be extended with resource assignment procedures to create algorithms for the full RFP.[1]

Many well-studied scheduling problems, which were in demonstrated [10] to be RFP instances, happen also to be KRFP instances, including:

- the traditional scheduling benchmark, job shop scheduling [14]
- multiple capacity job shop scheduling [27]
- ship loading [1, 22]
- bridge building [2]

We shall describe the three components of the KRFP (activities, resources and time constraints) before giving a full problem definition. In the following, variable symbols are italicized (e.g. $quantity_i$) while constant symbols are not (e.g. r_i).

3.3. Activities of the KRFP

In the KRFP, the activities are represented as a set of n atomic activities $\mathcal{A} = \{A_1, ..A_n\}$, such that each activity:

1. requires only one type of resource;
2. is non-interruptible but possibly has a duration that may vary;
3. and uses a quantity of resource that does not change throughout its duration.

Each atomic activity $A_i \in \mathcal{A}$ consists of five elements:

- r_i, the name of the required resource type;
- $area_i$, the resource area variable of A_i;
- $quantity_i$, the variable representing the used quantity of resource r_i;
- s_i, the *discrete* start point variable of A_i;
- and e_i, the *discrete* end point variable of A_i.

A constraint $area_i = quantity_i \times (e_i - s_i)$ relates $area_i$ and $quantity_i$ to the duration of the activity. While both $area_i$ and $quantity_i$ are variables, in most problems at least one will be fixed. A fixed $quantity_i$ and a variable $area_i$ expresses a variable-duration activity such as the "block move" of blocks world planning. A fixed $area_i$ and a free $quantity_i$ represents an *elastic* activity, which trades-off duration against resource usage.

Any other constraints relating the start and end times s_i and e_i should be included in the set of temporal constraints, $\mathcal{TC}$ (Sect. 3.5). For example, for an activity of fixed duration d, the single constraint $e_i = s_i + \mathrm{d}$ is added to $\mathcal{TC}$.

Complex activities that involve a sequence of sub-activities or use more than one resource must be modeled by several atomic activities. For instance, a complex activity using two resources $\mathrm{r}_1, \mathrm{r}_2$ is represented by two atomic activities. Temporal constraints that equate the start and end variables of these two atomic activities are then included in the temporal constraint set $\mathcal{TC}$.

3.4. The resources of the KRFP

The KRFP resources are reusable, i.e. are available for re-use after activities release them. Warehouses and machines are examples of reusable resources.

KRFP resources are described by:

- A set $\mathcal{R}$ of resource types.
- a function $quantity : \mathcal{R} \to \mathbb{N}$ specifying the number of interchangeable resources or the resource capacity for each resource type $r \in \mathcal{R}$.[2]

Let $\mathcal{A}_\mathrm{r}$ be the set of activities requiring a resource $\mathrm{r} \in \mathcal{R}$, given by $\mathcal{A}_\mathrm{r} = \{A_i : A_i \in \mathcal{A} \wedge \mathrm{r}_i = \mathrm{r}\}$. Let $\mathrm{T}_{\max}$ be the limit of the scheduling horizon. The following constraint ensures that the resource quantities are sufficient to satisfy the activities:

$$\forall r \in \mathcal{R}, \forall t \in \{0..\mathrm{T}_{\max}\} : quantity(r) \geq \sum_{A_i \in \mathcal{A}_r \wedge s_i \leq t < e_i} quantity_i$$

3.5. The Temporal Constraints of the KRFP

$\mathcal{TC}$ is the set of all KRFP temporal constraints. $\mathcal{TC}$ consists only of constraints in the following forms:

- u R c (bounding constraints)
- u R v $\pm$ c (distance constraints)

where $\mathrm{R} \in \{ =, <, >, \leq, \geq \}, u, v \in \bigcup_{A_i \in \mathcal{A}} \{s_i, e_i\}$, and $\mathrm{c} \in \mathbb{N}$

These forms generate tractable temporal sub-problems that correspond to simple temporal problems (STPs) [8].

The full KRFP combines the activities, resources and temporal constraints. The decision problem variant of the KRFP is given below.

Definition 2. **Kernel Resource Feasibility Problem** (KRFP)

INSTANCE: A 5-tuple $(\mathcal{A}, \mathcal{R}, \mathit{quantity}, \mathcal{TC}, \mathrm{T}_{\max})$:

- A set $\mathcal{A}$ of n activities $A_1, \ldots, A_n$. Each activity A_i is described as a 5-tuple $(\mathrm{r}_i, \mathit{area}_i, \mathit{quantity}_i, s_i, e_i)$.
- A set of $\mathcal{R}$ of m resource types $\mathrm{r}_1..\mathrm{r}_m$.
- The quantity of resource function $\mathit{quantity} : \mathcal{R} \rightarrow \mathbb{N}$.
- A set $\mathcal{TC}$ of temporal constraints of the form u R c or u R v $\pm$ c, where: $\mathrm{R} \in \{ =, <, >, \leq, \geq \}$, $u, v \in \bigcup_{A_i \in \mathcal{A}} \{s_i, e_i\}$, and $\mathrm{c} \in \mathbb{N}$
- The latest point of the scheduling horizon $\mathrm{T}_{\max} \in \mathbb{N}$.

QUESTION: Is there is an assignment of values to the variables in $\mathcal{A}$ such that the following constraints are satisfied?

- The activity constraints :

$$\forall A_i \in \mathcal{A} : \qquad \mathit{area}_i = \mathit{quantity}_i \times (e_i - s_i) \tag{1}$$

- The temporal constraints in the set $\mathcal{TC}$, including the constraints relating the start and end times of each activity, as well as the constraints enforcing the schedule horizon $\{0 \leq s_i \leq \mathrm{T}_{\max}, 0 \leq e_i \leq \mathrm{T}_{\max} : A_i \in \mathcal{A}\}$.
- Let $\forall r \in \mathcal{R}$: $\mathcal{A}_\mathrm{r} = \{A_i : A_i \in \mathcal{A} \wedge \mathrm{r}_i = \mathrm{r}\}$. The resource constraints:

$$\forall r \in \mathcal{R}, \forall t \in \{0..\mathrm{T}_{\max}\} : \qquad \mathit{quantity}(r) \geq \sum_{A_i \in \mathcal{A}_r \wedge s_i \leq t < e_i} \mathit{quantity}_i \tag{2}$$

4. Representing the KRFP as a Constraint Satisfaction Problem

This section gives a CSP representation of the KRFP. While the activity constraints of Eqn. 1 and the temporal constraints of $\mathcal{TC}$ can be represented directly as CSP constraints, the resource overlap constraints of Eqn. 2 would impact efficiency, because a CSP constraint would be required for each time point.

The model presented here is based on [9]. A resource quantity variable Q_{r} is introduced for each $\mathrm{r} \in \mathcal{R}$. The resource quantity variable Q_{r} corresponds to the quantity used of that resource (the maximum quantity used over the schedule horizon). It is sufficient to count the quantities used at activity start times, since these are the points where increases in usage occur. For each activity $A_i \in \mathcal{A}_{\mathrm{r}}$, a variable $Q_{\mathrm{r}s_i}$ counts the quantity of resource r used at its start time s_i. Q_{r} is the maximum of these quantities $Q_{\mathrm{r}} = \mathrm{MAX}\{Q_{\mathrm{r}s_i} : A_i \in \mathcal{A}_{\mathrm{r}}\}$.

$Q_{\mathrm{r}s_i}$ variables are defined in terms of Booleans. A Boolean $B_{s_iA_j}$ is introduced for each pair of activities $A_i, A_j \in \mathcal{A}_{\mathrm{r}}$. It is set when activity A_j overlaps with s_i.

$$\forall r \in \mathcal{R}, \forall\ A_i, A_j \in \mathcal{A}_r : B_{s_iA_j} = \left\{ \begin{array}{l} 1 \text{ iff } s_j \leq s_i \wedge s_i < e_j \\ 0 \text{ otherwise} \end{array} \right\} \tag{3}$$

When linked with the corresponding $Q_{\mathrm{r}s_i}$, these Boolean variables link temporal and resource reasoning. The following constraints are applied:

$$\forall r \in \mathcal{R}, \forall A_i \in \mathcal{A}_{\mathrm{r}} : \qquad Q_{rs_i} = \sum_{A_j \in \mathcal{A}_r} B_{s_iA_j} \cdot \mathit{quantity}_j \tag{4}$$

Each $Q_{\mathrm{r}s_i}$ variable on the LHS (left hand side) of one of the constraints of Eqn. 4 is bounded by the maximum resource quantity $\mathit{quantity}(\mathrm{r})$, via the constraints $Q_{\mathrm{r}s_i} \leq Q_{\mathrm{r}}$ and $Q_{\mathrm{r}} \leq \mathit{quantity}(\mathrm{r})$. While it is possible to dispense with Q_{r} and directly impose the constraints $Q_{\mathrm{r}s_i} \leq \mathit{quantity}(\mathrm{r})$, the Q_{r} variables will allow us to easily modify the KRFP in the next section, by adding new constraints on them.

5. A Minimal Perturbation Variant of the KRFP

This section introduces the resource utilization problem (RUP), chosen as an example of KRFP-based minimal perturbation problems. The algorithms of the paper are tested and compared on the RUP.

Section 2 described the kinds of changes that frequently arise in dynamic scheduling. These changes may cause the existing schedule to violate the following types of constraint:

- Type (1): the activity constraints of Eqn. 1
- Type (2): the temporal constraints in $\mathcal{TC}$
- Type (3): the resource constraints of Eqn. 2, modeled by Eqns. 3 & 4

All the algorithms described here deal with change by solving the modified KRFP from scratch. The initial schedule impacts their search only via the minimal perturbation requirement that is captured in the optimization function. *Thus any changes that result in a new KRFP can be handled.* Activity changes will cause the set of activities or the activity constraints to change; reductions in resource availability for all or part of the schedule horizon can result in changes to the resource quantities (all the horizon) or the introduction of new, dummy activities (part of it); and, lastly, any temporal changes cause modifications to the temporal constraint set.

While the algorithms handle arbitrary KRFP changes, they address only a special class of perturbation functions, specifically those that can be expressed linearly (as a linear function over linear constraints). For example, a count of activities shifted in time cannot be expressed linearly, but a weighted sum of time shifts can. Nevertheless, the algorithms utilize search strategies that can be replicated in analogous algorithms tackling other perturbation functions. The wider applicability of probe backtracking in particular will be discussed later in Sect. 10.

Section 6.3 will show that for the optimization functions addressed, schedules with violations of constraints of Type (1) and Type (2) can be cheaply transformed into minimally perturbed schedules with zero or more violations of Type (3) only. Only computationally difficult problems involving newly created resource contention involve violations of Type (3) after the transformation.

The RUP is of interest because it implements changes that always cause newly created resource contention (i.e. Type (3) violations). In the RUP, there is only one resource type, and activities require only one unit of that resource. The only allowed change to the problem is a reduction in the quantity of the resource. The RUP's perturbation function measures *temporal disruption*, defined to be the total time shift, i.e. the total change (usually in minutes) to the start and end times of activities. This is a linearly expressed function, as will be shown in Sect. 6. The toy example shown in Fig. 1 involved an RUP-style change.

Definition 3. **Resource Utilization Problem (RUP)**

A *resource utilization problem* is defined to be a minimal perturbation problem $(\Theta, \alpha_\Theta, C_{\text{del}}, C_{\text{add}}, \delta)$ where:

- Θ is any KRFP $(\mathcal{A}, \mathcal{R}, \mathit{quantity}, \mathcal{TC}, \mathrm{T}_{\max})$ modeled as a CSP, such that:
  - there is only one resource type r ($\mathcal{R} = \{\mathrm{r}\}$) and Q_{r} is the resource quantity variable for that resource type
  - $\forall A_i \in \mathcal{A} : \mathit{quantity}_i = 1$
- α_Θ is a solution to Θ
- $C_{\text{del}} = \emptyset, C_{\text{add}} = \{Q_{\mathrm{r}} \leq \mathrm{c}\}$, such that $\mathrm{c} < \mathit{quantity}(\mathrm{r})$
- $\delta(\alpha_{\Theta'}, \alpha_\Theta) = \sum\limits_{u \in \bigcup\limits_{A_i \in \mathcal{A}} \{s_i, e_i\}} |\alpha_{\Theta'}(u) - \alpha_\Theta(u)|$

The RUP is an interesting model in its own right. It may be used to capture minimal perturbation requirements in scheduling environments where resource reductions occur due to schedule efficiency improvements or resource failures. The following sections will show how the algorithms can solve minimal perturbation variants of the KRFP with particular emphasis on the RUP.

6. Properties of the Temporal Sub-Problem

With one exception (viz. mixed integer programming (MIP)), the algorithms compared in this paper will conduct search in two, interleaved phases. In the resource feasibility phase, the algorithms relieve resource contention by ordering temporal variables. In the temporal optimization phase, the algorithms find values for the temporal variables that are optimal, *subject to the orderings decided in the resource phase.* The two phases are interleaved differently in the algorithms, such that they all eventually find the problem's global optimum.

Let us focus initially on the temporal optimization phase. It was noted that the allowed forms of the temporal constraints TC in a KRFP constitute a tractable sub-problem (Sect. 3.5). In this section, the temporal constraints, together with the constraints associated with the RUP perturbation function, are shown to have a property known as *total unimodularity* (TU). This property additionally means that linear solvers such as Primal or Dual Simplex will return optimal solutions that are *discrete* [16].[3]

6.1. The temporal constraints $\mathcal{TC}$ and the TU property.

TU problems are well known in the mathematical programming community, and are often encountered as components of combinatorial optimization problems. Ordinary network flow problems, such as *transportation*, *assignment* and *minimum cost network flow*, have TU.

Linear programming (LP) solvers express the problem's variables and linear inequality constraints using a constraint matrix. The necessary conditions for TU to hold in the matrix are not given here, but a more practical set of sufficient conditions are relayed instead [17].

The constraint matrix may be configured in two ways. In the first, rows represent constraints, and columns represent variables, with the element e_{ij} of the matrix denoting the coefficient of a variable j in a particular constraint i. In the dual configuration, the reverse holds, with rows corresponding to variables and columns to constraints.

A set of sufficient conditions in a problem or its dual

1. All variable coefficients are 0, 1, or -1, and all constants integer.
2. Two or less nonzero coefficients appear in each row.
3. The columns of the matrix can be partitioned into two subsets S_1 and S_2 such that
   (A) If a row contains two nonzero coefficients with the same sign, one element is in each of the subsets.
   (B) If a row contains two nonzero elements of opposite sign, both elements are in the same subset.

The temporal constraints of the KRFP satisfy the sufficient conditions if we partition the variables (columns) into a subset S_1 containing all the temporal variables, and another S_2 containing none. Thus the temporal constraints of the RUP (and the KRFP) constitute a TU set.

6.2. *The optimization function constraints*

Often a subproblem satisfies the conditions guaranteeing the TU property but *link constraints* violate them. Link constraints are not genuine problem constraints because they do not constrain the set of feasible solutions. They are used to link variables in the TU set with variables in the optimization function. The RUP optimization function uses link constraints.

The RUP perturbation function δ was defined to be the sum, over the temporal variables, of absolute change. In LP, this function is modeled by introducing a variable d_x for each temporal variable x. d_x corresponds to the absolute change in x and is given by $d_x = |x - \mathrm{c}|$, where c represents the value given to x in the previous solution. This non-linear expression is captured by placing d_x in the optimization function and adding the following linear constraints:

$$d_x \geq x - \mathrm{c} \tag{5}$$
$$d_x \geq \mathrm{c} - x \tag{6}$$

The constraints of Eqn. 6 violate the sufficient conditions for TU in the matrix. Instead, we outline a proof that TU is preserved on the incremental addition of a new variable d_x and the two constraints of Eqns. 5 & 6.

The Primal and Dual Simplex algorithms work by obtaining solutions at the vertices of the solution polyhedron, as defined by the linear inequalities of the problem. The TU property guarantees that the solutions at the vertices are discrete. The proof demonstrates that solutions at the new vertices created by the addition of this constraint pair are also discrete. The proof is then extended by induction.

Let E be a TU set. We extend E to E' by adding a new variable d_x and the two constraints in Eqns. 5 & 6. d_x adds a new dimension to the polyhedron, and cuts it along the hyper-planes $d_x = x - \mathrm{c}$ and $d_x = \mathrm{c} - x$. All existing vertices are projected in the d_x dimension onto these two hyper-planes. At the projected vertices d_x takes the value $|x - \mathrm{c}|$, which is discrete because by induction x is. In addition, new vertices are added where the two hyper-planes intersect at $x = \mathrm{c}$. But the constraint $x = \mathrm{c}$ satisfies the sufficient conditions for TU as described above, which guarantees that the vertices on its hyper-plane are discrete. Thus the extended set E' also has TU.

This completes the demonstration that the temporal optimization phase can be solved efficiently, since cheap linear solving procedures will guarantee to return globally optimal, discrete solutions.

6.3. *Translating schedule violations into resource violations*

It is now possible to affirm the statement in Sect. 5 that schedules with violations of Type (1) (activity constraints) and Type (2) (temporal constraints) may be transformed efficiently into violations of Type (3) (resource constraints), such that the newly transformed schedule is not more perturbed than any feasible schedule. This new schedule is described as being *super-optimal*; a super-optimal assignment is a partially consistent assignment that is at least as good, with respect to the optimization function, as any solution (i.e. fully consistent assignment).

The simple case, Type (2) violation, is addressed first. It was shown that the temporal and optimization function constraints constitute a TU set. Because of the TU property, a linear solver will produce an assignment satisfying the temporal constraints while minimizing differences to the existing (temporally inconsistent) solution. The assignment however may still violate constraint Types (1) and (3).

To transform violations of the activity constraints – Type (1) – additional steps are required before and after the linear solver is invoked. The activity constraints $area_i = quantity_i \times (e_i - s_i)$ have four variables only two of which appear in the optimization function, namely e_i and s_i. These two variables must be minimally constrained so that no optimal solutions are ruled out. This is achieved by the imposition of two constraints $\mathrm{LB}(area_i/quantity_i) \leq (e_i - s_i)$ and $(e_i - s_i) \leq \mathrm{UB}(area_i/quantity_i)$, where LB an UB are functions returning respectively the integer lower and upper bounds of an expression. These two constraints can be expressed in the form required for inclusion in $\mathcal{TC}$, the temporal constraint set. The linear solver then returns a discrete optimum satisfying $\mathcal{TC}$. The final step is to re-satisfy the activity constraints using a polynomial time procedure that assigns appropriate values to $area_i$ and $quantity_i$. Any remaining violations in the resulting super-optimal schedule will be of Type (3).

7. A Suite of Algorithms for Minimal Temporal Perturbation

Let us turn our attention now to the resource feasibility phase of the algorithms. Given the efficient algorithms for temporal optimization these are the critical algorithmic component.

Unlike a conventional backtrack search procedure using conventional variable and value ordering heuristics, the algorithms conduct search in the resource feasibility phase around the problem's constraints. The aim of the search is to reduce *contention* in the constraints; a constraint is said to be subject to contention when the domains of its variables are such that some combinations of values could violate it. For some constraints, such as the resource constraints described in Eqn. 4, contention can be measured and the search can be directed towards regions where contention is reduced.

Recall that in the RUP the variable $Q_{\mathrm{r}s_i}$ corresponds to the quantity of resource r used at the start of activity A_i (Sect. 4). In the RUP all activities use a single unit of resource, and Eqn. 4 simplifies to:

$$\forall\, A_i \in \mathcal{A} : \qquad Q_{\mathrm{r}s_i} = \sum_{A_j \in \mathcal{A}} B_{s_i A_j} \tag{7}$$

Notice that, at the start of search, on the left hand side (LHS) of Eqn. 7, $Q_{\mathrm{r}s_i}$ will have been bounded by the RUP's *reduced* resource capacity c (c $< quantity(\mathrm{r})$) via $Q_{\mathrm{r}s_i} \leq Q_{\mathrm{r}}$ and the newly added constraint $Q_{\mathrm{r}} \leq$ c (Def. 3). However, if left unconstrained the RHS sum expression can freely take a range of values depending on the current domains of the overlap Booleans (which depend on the domains and orderings of variables in the temporal sub-problem).

Contention exists in the resource constraints described Eqn. 7 only when the RHS upper bound exceeds the LHS upper bound. Moreover, the degree of contention is the given by $max(0, UB(RHS) - UB(LHS))$, where UB is a function returning the upper bound of an expression according to the domains of the variables in it. By continually monitoring the LHS and RHS upper bounds during search, the search procedure may order constraints with contention according to the degree of contention in them. Once a constraint is selected for contention relief, a backtrackable search decision imposes a new temporal constraint that forces apart temporal variables, setting some RHS Boolean $B_{s_i A_j}$ to 0. This decision may be backtracked later if inconsistency is detected, or if a solution has been found but a better one is sought by a branch and bound optimization process.

For the general KRFP, where the constraints of Eqn. 4 do not simplify to Eqn. 7, a minor extension is required to allow search to relieve contention using a further method. Recall that elastic activities have variable resource usage. Where variable resource usage occurs on the RHS, the search procedure can alternatively lower $UB(RHS)$ by forcing apart the temporal variables s_j, e_j of an elastic A_j (to impose a new minimum duration and obtain a reduction in the upper bound of $quantity_j$).

7.1. *The Constraint Backtrack algorithms (CB)*

The first set of algorithms are based on the resource feasibility algorithm of [29] extended here with temporal optimization. The resource feasibility phase is designated constraint_backtrack (CB).

Fig. 2 gives the pseudo-code for the first stage of CB (constraint_backtrack).[4] The procedure is initialized with the parameter MonitoredConstrs set to be the constraints where contention must be relieved (the constraints of Eqn. 7 in the RUP). The procedure implements an ordinary depth-first search using calls to PushConstrStore and PopConstrStore. PushConstrStore pushes a decision represented as a constraint onto the constraint store stack and triggers local consistency propagation. PopConstrStore undoes a decision and the propagation that followed it.

7.1.1. Constraint filtration and selection. In algorithm CB, the procedure constraint_filter returns the constraints that are subject to contention. The constraints filtered out have no impact on resource feasibility in this search branch, so they are excluded from consideration. When no more contention constraints exist, the resource feasibility phase may pass control over to the temporal optimization phase (subsequent backtracking may return search again to the resource feasibility phase).

```
 1.  begin constraint_backtrack(MonitoredConstrs)
 2.      ContentionConstrs := constraint_filter(MonitoredConstrs);
 3.      if ContentionConstrs == $\emptyset$
 4.          then return TRUE
 5.          else begin
 6.                  Constr := select_constraint(ContentionConstrs);
 7.                  Decision:= select_decision(Constr);
 8.                  if PushConstrStore(Decision)
 9.                      then if constraint_backtrack(MonitoredConstrs)
10.                          then return TRUE;
11.                  PopConstrStore;
12.                  if (PushConstrStore not(Decision))
13.                      then if constraint_backtrack(MonitoredConstrs)
14.                          then return TRUE;
15.                  PopConstrStore;
16.                  return FALSE;
17.              end;
18.  end constraint_backtrack
```

Figure 2. constraint_backtrack (CB): An initial algorithm for the resource feasibility phase

The procedure select_constraint selects from the filtered constraint set (ContentionConstraints) the constraint Constr with the greatest potential for conflict – viz. the greatest value for the RHS upper bound minus the LHS upper bound. This value is always positive for ContentionConstraints.

7.1.2. Selecting a decision. At this point the objective is to reduce contention in Constr. Suppose that Constr constrains the resource quantity $Q_{\mathrm{r}s_i}$ at the start of activity A_i. As described, in the RUP contention is reduced by imposing a new temporal ordering constraint, setting some RHS $B_{s_iA_j} = 0$. The new temporal ordering constraint is chosen using a least commitment heuristic designed to select the one with the smallest impact, in order to maximize the chance of obtaining feasible solutions. The heuristic in select_decision assesses, for each RHS $B_{s_iA_j}$ that is not yet fixed, the ordering constraints that could be used to set $B_{s_iA_j} = 0$ (namely $s_i < s_j$ and $e_j \leq s_i$). If a potential ordering constraint is compatible with the variable domains, the local impact of imposing the constraint is calculated, in terms of the number of values pruned from the domains of s_i, s_j and e_j. Decision is assigned to the ordering constraint with the smallest local impact.

On backtracking the ordering constraint of Decision is revoked and replaced by its negation not(Decision) (as expected not$(x < y)$ is simply $x \geq y$ and not$(x \leq y)$

is $x > y$). If the negation also fails, it is revoked and the algorithm backtracks to an earlier decision.

7.1.3. Constraint propagation. Once a decision is imposed it is pushed into the constraint store by PushConstrStore. The algorithm applies arc-B consistency propagation on the arithmetic constraints [24]. The decision is propagated through interplay between these resource and temporal constraints. Propagating the decision may result in confirmed overlaps elsewhere (e.g. $B_{s_i A_k} = 1$ for some k).

In addition to arc-B consistency a look-ahead check is applied. This check is found to be important for the KRFP model, which relies on computing overlap at activity start times only. After some Boolean $B_{s_i A_j}$ has been set to 0 by the latest search decision, a resource usage profile is built for the time horizon interval which activity A_i must span (from $UB(s_i)$ to $LB(e_i)$). The interval only exists when $UB(s_i) < LB(e_i)$. Other activities are assumed to take minimal spans over this interval. Failure is signaled and a backtrack instigated when resource usage exceeds the capacity at any point in the interval.

7.1.4. Increasing propagation in the algorithm: (CB+LP) and (CB+LP+Edge) As the results of Sect. 8 will show, the CB algorithm is relatively ineffective at obtaining the optimum and proof of optimality. The propagation methods described above fail to sufficiently prune search. In particular, the optimization function is not used to reduce the search space; the implications of individual search decisions for cost are not discovered until most variables are fixed.

Cost bounds can dramatically reduce the search space of the optimization process by pruning search branches that lead to worse solutions than those already found. The algorithm CB was extended to a new algorithm CB+LP applying the linear solver during the resource feasibility phase. The linear relaxation of the problem is solved after every search decision to obtain a lower bound on the cost in the resulting sub-problem.

The algorithm was also extended to deploy the cubic edge-finder for capacitated resources [28]. This algorithm is denoted CB+LP+Edge and also achieves improvements in performance. Edge is a more powerful but more costly lookahead check than that described in Sect. 7.1.3.

Nevertheless, even the improved search spaces of CB+LP and CB+LP+Edge remain large. For improved performance, it is necessary to reduce the number of decisions made in the resource feasibility phase, avoiding whenever possible modifications to the old solution.

7.2. The Probe Backtrack algorithms (PB)

Probe backtracking is an extended form of backtracking, where the backtrack search component is supported by lookahead procedures (*probe generators*) which generate potentially good assignments (*probes*).

The origins of the generic probe backtracking algorithm are described in Sect. 9. As in several previous algorithms, probe backtracking associates each unassigned variable not only with a domain of values, but also with a tentative value. In probe backtracking the tentative value is not necessarily adhered to and may be changed by the probe generator as the backtrack search progresses. The purpose of the probe generator is to direct the backtrack search component and limit the size of its search space. This is achieved by focusing the backtrack search only on regions where the probe violates constraints. Unlike many other repair-based algorithms, probe backtracking algorithms are in general complete.

Figure 3 demonstrates the extension of the constraint_backtrack algorithm to perform a probe driven search. The extensions have been highlighted in the figure. The first change is the introduction of obtain_probe_assignment at the start of the procedure. This represents a call to the probe generator, which obtains the probe used to focus the search decisions at this search node. The only condition that must be guaranteed in a probe generator is that the assignments it produces must satisfy variables' domain constraints, updated to reflect the decisions imposed by backtrack search algorithm. In practice the other condition necessary for good algorithm behavior is that the probe generator produces good quality assignments, preferably super-optimal with respect to the optimization function (where one exists).

```
begin probe_backtrack(MonitoredConstrs)
    Assignment := obtain_probe_assignment;
    ViolatedContentionConstrs := constraint_filter(MonitoredConstrs,Assignment);
    if ViolatedContentionConstrs == ∅
        then return TRUE
        else begin
                Constr := select_constraint(ViolatedContentionConstrs,Assignment);
                Decision := select_decision(Constr,Assignment);
                if PushConstrStore(Decision)
                    then if probe_backtrack(MonitoredConstrs)
                        then return TRUE;
                PopConstrStore;
                if (PushConstrStore not(Decision))
                    then if probe_backtrack(MonitoredConstrs)
                        then return TRUE;
                PopConstrStore;
                return FALSE;
            end;
end probe_backtrack
```

Figure 3. probe_backtrack (PB): A probe backtracking algorithm for the resource feasibility phase

The other highlighted changes to the algorithm reflect the fact that the assignments returned by the probe generator are now inputs to the constraint_filter, select_constraint and select_decision procedures. These are examined in the context of the RUP below.

7.2.1. Constraint filtration and selection. It is now possible to increase the number of constraints removed by the filtration process. As with the CB algorithms, any constraints that are not subject to contention are filtered out. In addition, if the tentative values of the probe returned by the probe generator satisfy a constraint, then it is filtered out, even though it may be subject to contention.

EXAMPLE: Let $X[D\mathrm{v}]$ represent a variable X with a domain D and a probe value v. In probe backtracking, ViolatedContentionConstrs would not include the constraint

$$Q_{\mathrm{rs}_i}[\{0..1\}1] = B_{s_iA_j}[\{0,1\}0] + B_{s_iA_k} : [\{0,1\}0] + B_{s_iA_l}[\{0,1\}1]$$

even though it is in contention, because it is satisfied in the probe assignment. □

While constraints satisfied by the current probe may at some later point become violated in a future probe, for now it is assumed that the current probe will lead to a solution. Clearly, the procedure may terminate immediately if all constraints are satisfied by the current probe, since it is a solution.

7.2.2. Selecting a decision. In decision selection it is also possible to use the assignment to order the possible choices. The constraint Constr will have been violated by the probe assignment. Decisions should be selected that force new probes to obtain assignments that are closer to satisfying Constr. Selecting decisions that allow the probe generator to return the same probe assignment would not progress the search.

In the RUP, Constr constrains some resource quantity Q_{rs_i} at the start of activity A_i. As in CB contention reduction is achieved by forcing apart two temporal variables (i.e. imposing a new temporal ordering constraint) setting some RHS $B_{s_iA_j} = 0$ in the process. However, in PB the heuristic prefers to reset a $B_{s_iA_j}$ that has a value of 1 under the current probe assignment. This heuristic operates like a filter because such variables always exist in a violated constraint.

Subject to this primary heuristic, a secondary heuristic is used to order the decision constraints. Two secondary heuristics were considered. The first is the least commitment heuristic used in the CB algorithm. Unless indicated otherwise, this is the secondary heuristic applied in the PB algorithms studied.

The other secondary heuristic relies on estimates of the impact of a new ordering constraint on the assignment returned by the probe. The following example illustrates how impact is measured.

EXAMPLE: Suppose that the ordering constraint $e_j \leq s_i$ is being considered for selection. Currently $e_j > s_i$ according to the probe assignment otherwise $e_j \leq s_i$

would have been removed from consideration by the primary heuristic. The alternative secondary heuristic estimates the minimum change that must be undergone by the two variables, and is given by $probe(e_j) - probe(s_i)$, where $probe$ returns the tentative value of an unassigned variable.

□

In the alternative secondary heuristic, the ordering constraint with the smallest change estimate is chosen as the value for Decision. Algorithms deploying the alternative secondary heuristic are suffixed by +Alt (e.g. PB(Unimod)+Edge+Alt).

7.2.3. The algorithm PB(Old) The first version of a probe backtracking algorithm, PB(Old), creates a probe generator closely based on the old solution. The old solution is of course only partially consistent with the new problem constraints. On each invocation of obtain_probe_assignment, the probe values for the temporal variables are moved to lie within their new domains, pruned by the local consistency propagation of earlier search decisions. The values for the Boolean variables are then made to correspond with the values chosen for the temporal variables.

While very simple the above pure probe backtracking algorithm is incomplete. The probe values may not satisfy all the temporal constraints, and completeness is lost because they are not monitored for violation (the MonitoredConstrs parameter includes only resource constraints).

The incomplete algorithm is restored to completeness by introducing an intermediate phase between the probe backtracking and temporal optimization phases, to relieve any remaining contention from resource constraints filtered out by the probe. In conjunction with LP propagation on the relaxed problem, the resulting complete algorithm, PB(Old)+LP, is an improvement over the CB algorithms (see Sect. 8). This is because it focuses search to some degree on the old solution.

Nevertheless, the algorithm is not ideal; the probes generated in the probe backtracking phase may be of limited relevance as they violate a number of temporal constraints that increases as new search decisions are made. Though the tentative values for the remaining variables are still super-optimal, they may violate so many temporal constraints that they no longer provide accurate information about feasible optimal solutions.

7.2.4. The unimodular probing algorithm PB(Unimod) This observation motivates a probe backtracking algorithm that continually updates the probe assignment during search to satisfy the temporal constraints. The global impact on cost of choices made during search are clarified by finding the LP optimum satisfying the temporal constraints posted so far — the linear relaxations of any other constraints are *not* included in the LP set to avoid discreteness violations. The discrete values at the LP optimum are installed as the new probe values for the variables. Since only violations of resource constraints occur in this probe, it achieves a high level of consistency, and therefore is a better approximation of the problem's optimal solutions. The probe backtracking algorithm described is a *unimodular probing* algorithm [11, 12].

At each search step the unimodular probing algorithm applies LP to the up to date TU set $\mathcal{TC}'$. The discrete, super-optimal assignments returned by LP, called unimodular probes, satisfy the constraints in the TU set. As with PB(Old), PB(Unimod) focuses on the resource constraints that are violated by the probe. If the probe violates any resource constraints, the algorithm selects a violated constraint, and rules out the present unimodular probe by imposing a new temporal ordering constraint (which is a new TU constraint added to the constraints in $\mathcal{TC}'$). Hence search decisions add at least one new TU constraint to the LP set at each search node. Moreover, after a decision has been posted, the local consistency methods applied may derive further unary and binary TU constraints.

When no more resource constraint violations remain, in contrast to the probe of PB(Old), the PB(Unimod) probe is a full solution. This solution is optimal subject to the temporal orderings imposed so far, and no further search needs to take place in this search branch. Backtracking can therefore be immediately initiated in a branch and bound search for the global optimal.

To summarize, PB(Unimod) extends backtrack search by using LP not only as a propagation method (detecting failure through cost bounds), but also as the basis for variable and value selection heuristics in discrete problems such as the KRFP. Most importantly, the repair strategy limits the number of decisions made by the backtrack search component by avoiding any more search decisions in a search branch where the conflict set has become empty. An empty conflict set is proof that the probe is both feasible *and optimal given the search decisions taken.* Local consistency techniques such as arc-B and edge finding are orthogonal and contribute to the efficient handling of the disjunctive constraints that are out of LP's scope.

8. Performance Comparisons

Unimodular probing was found to significantly outperform rival algorithms on the large scale commercial application that originally motivated this research. Unfortunately because the data is commercially sensitive these problems are not publicly available. Instead, the algorithms have been tested on a set of randomly-generated RUP benchmarks. The commercial application problems involve a far greater number of activities but are comparable nevertheless because structured temporal constraint networks reduce their combinatorics. The results found for the random problems appear to closely reflect our commercial experience.[5] Details of the benchmarks have been made available on the Internet for future comparisons.[6]

Each benchmark is generated by randomly creating a set of activities for a single resource pool over a restricted schedule horizon. Locally feasible, binary temporal constraints are posted such that the expected number of constraints on a temporal variable is given by E_{density}, where E_{density} is a parameter to the problem generator. While activities have a nominal duration, this is not fixed since activities are allowed to shrink by a given amount.

For every benchmark file a number of different problems may be constructed. One parameter is the maximum allowed time shift, which imposes lower and upper bounds on the temporal variables. In the problems tested bounds on acceptable time shifts are uniformly applied to the temporal variables (e.g. activity starts and ends must shift by no more than 50 minutes). The second parameter is the desired reduction in the number of resources. Greater reductions tighten the problem and cause greater perturbation.

Two problem sets were attempted. In both the activity durations were nominally 100 minutes and the maximum allowed shrinkage 10. The total schedule horizon was 985 minutes with a time granularity of 5 minutes. All algorithms were tested on 300MHz Pentiums, with a timeout of 2500 CPU seconds and a spaceout of 150MB.

In problem set 1, 1800 problems were searched by five algorithms.[7] The following parameters were varied: no. of activities ({20,30}); E_{density} ({0,0.5,1}); maximum time shift ({20,50,100}); and number of resource reductions ({-1,-2}).

Four of the algorithms compared on this problem set have been described in this paper: CB,CB+LP,PB(Old)+LP, PB(Unimod). The fifth was a commercial mixed integer programming package (MIP), CPLEX. The same problem formulation was used, with the problem Booleans declared as integer. It should be noted that there exists a large body of literature on specialized MIP problem formulations that take advantage of specific problem characteristics (e.g. [31]). However we are not aware of any formulations that closely match the RUP or the KRFP more generally. The default CPLEX search settings were used; other settings were attempted on a representative sample of 200 problems for both the node selection and variable selection strategies with minor or no improvement (3% in the best case). Experienced MIP practitioners confirm our belief that the default CPLEX settings are often among the best for MIP search.

The results are shown in Table 1. The first column gives the percentage of problems where a timeout or spaceout occurred. The second gives the percentage for which the algorithm successfully proved a solution to be optimal. The final column gives the percentage of problems proved to be infeasible.

Table 1. Problem set 1: Search Results

Algorithm	Time/Spaceouts	Proofs(Optimal)	Proofs(Infeasible)
CB	50.8%	26.3%	22.8%
CB+LP	27.3%	50.4%	22.2%
PB(Old)+LP	22.2%	53.1%	24.7%
PB(Unimod)	**7.8%**	**66.2%**	**26.0%**
MIP	24.0%	57.7%	18.3%

The results show an improvement that corresponds with the progression of algorithms from CB (the worst) to PB(Unimod) (the best). MIP appears to hold out relatively well, albeit more so on proving optimality than on proving problem infeasibility. Figure 4 shows the percentages of time/spaceouts for the algorithms distributed over problem size (20/30 activities). The graph indicates that the computational advantage of PB(Unimod) scales as the number of activities increases.

Since the first four algorithms ran on the ECL^{i}PS^e platform and shared much of their code, time/spaceouts are a good means of comparison. The fifth, MIP, is a commercial package developed for efficient linear programming. To obtain a

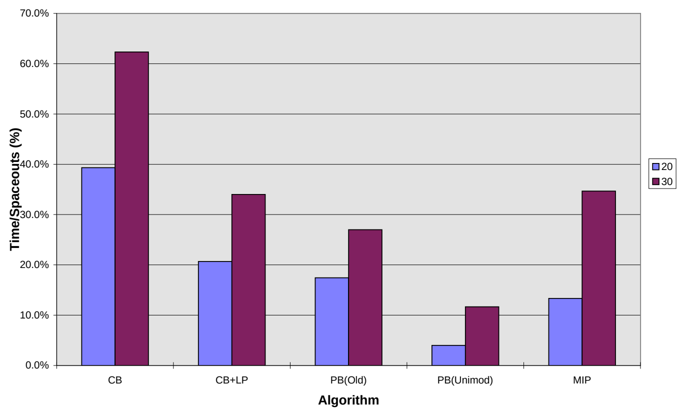

Figure 4. Problem set 1: Time/Spaceout Percentages for 20 and 30 Task Problems

fairer assessment, the two algorithms MIP and PB(Unimod) were compared by the computing the average number of times they invoked the linear solver, as well as the average CPU time, for the problems in set 1 that were solved to optimality by *both*. Table 2 gives the averages together with 90th percentiles.

Table 2. Problem set 1, solved to optimality by both : MIP vs. PB(Unimod)

Algorithm	LP Nodes Avg. / 90th percentile	CPU time (s) Avg. / 90th percentile
PB(Unimod)	**583 / 1148**	**50 / 92**
MIP	25636 / 72757	173 / 591

The results show that PB(Unimod) dramatically reduces the number of LP search nodes required to obtain an optimal solution. However, the specialized MIP package manages many more Simplex calls per second than the PB(Unimod) implementation (which has a relatively inefficient interface to Simplex). Nevertheless, the reduction in the number of LP Nodes is translated into a significant CPU time advantage.

Another series of experiments was run in problem set 2, which was composed of 1000 problems. Greater ranges were chosen for time shift and resource reduction to investigate the effects of larger perturbations and tighter problems. The following parameters were applied: no. of activities ({20}); E_{density} ({0.5}); maximum time shift ({20,50,100,500,1000}); and number of resource reductions ({-1,-3,-5,-7}).

The five algorithms tested were CB+LP, CB+LP+Edge, PB(Unimod), PB(Unimod) +Edge,PB(Unimod)+Edge+Alt, and MIP. The results show that both CB+LP and PB(Unimod) are improved by the addition of edge finding propagation because it

Table 3. Problem Set 2: Search Results

Algorithm	Time/Spaceouts	Proofs(Optimal)	Proofs(Infeasible)
CB+LP	21.5%	22.8%	55.7%
CB+LP+Edge	17.4%	23.0%	59.6%
PB(Unimod)	16.4%	26.4%	57.2%
PB(Unimod)+Edge	13.1%	26.5%	60.4%
PB(Unimod)+Edge+Alt	**12.2%**	**27.6%**	**60.2%**
MIP	27.6%	21.6%	50.8%

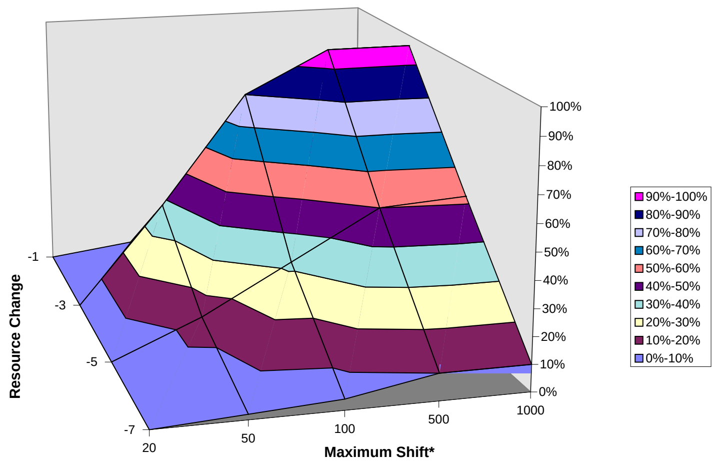

Figure 5. Problem Set 2: MIP Time/Spaceouts versus Resource Reduction and Max. Time Shift[8]

expedites proofs of infeasibility. This indicates that the addition of propagation methods in general will yield similar benefits to both types of algorithm. A performance improvement was also noted with the use of the secondary heuristic Alt.

Figures 5, 6 and 7 show the MIP, CB+LP+Edge and PB(Unimod)+Edge+Alt time/spaceouts distributed over the maximum time shift and resource reduction parameters. The graphs show the performance gains of the latter scale well with the hardness of the problem.

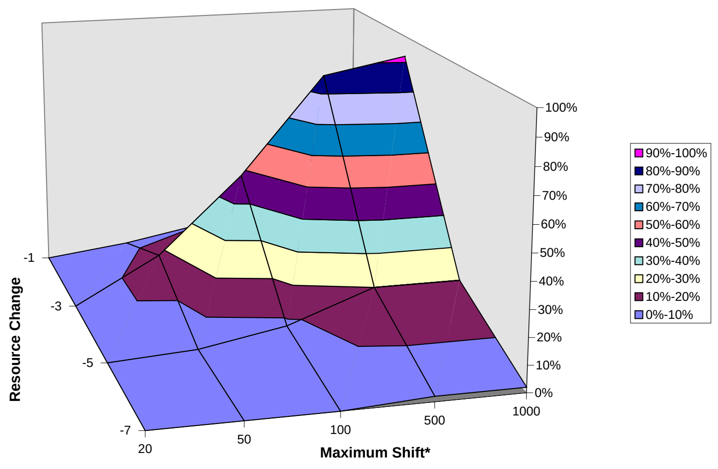

Figure 6. Problem set 2: CB+LP+Edge Time/Spaceouts versus Resource Reduction and Max. Time Shift

9. Background

9.1. Probe Backtracking (PB)

Probe backtracking aids backtrack search by utilizing a forward probing method in addition to the more conventional forward local consistency filtration algorithms. This concept is a generalization and a synthesis of a number of ideas that have existed implicitly in earlier constraint and mathematical programming algorithms:

Static Probe Backtracking *Driving and scoping backtrack search by the conflicts in an initial partially consistent assignment.* Early methods used heuristic measures to detect problem bottlenecks and areas of constrainedness (e.g. [32] in scheduling). Minton et al. proposed a disciplined approach based on using an initial, partially feasible, complete assignment in conjunction with the min-conflicts heuristic [26]. The initial assignment was produced using a polynomial-time procedure that tended to reduce the number of constraint violations. One of the algorithms presented was a backtrack algorithm that repaired regions of constraint violation in the complete assignment. Newer methods also employed the notion of partially consistent, tentative assignments to uninstantiated variables [33, 34]. These methods also started from a partially consistent assignment and reduced constraint violations until global feasibility was obtained. [30] used probabilistic analysis to show that a random probe assignment obtained before search improves

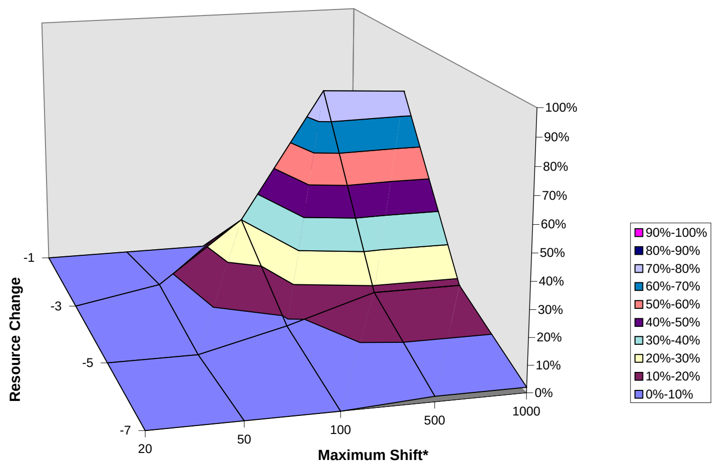

Figure 7. Problem set 2: PB(Unimod)+Edge+Alt Time/Spaceouts versus Resource Reduction and Max. Time Shift

the average case performance of backtracking (the term *probe* used here is from [30]).

Dynamic Probe Backtracking *Utilizing an independent algorithmic component to update the probe assignment after each search decision.* By this definition, the algorithms referred to above are not dynamic. This kind of dynamicity is to be found implicitly in mathematical programming techniques such as Mixed Integer Programming (MIP) that conduct search by repairing up-to-date linear programming probes.

Easy Sub-Problem Probes *Partitioning of the problem requirements into hard and easy sub-problems for handling by separate algorithmic components.* An early constraint programming example is [7], which solves an easy sub-problem and uses elicited information to guide a search of the whole problem. The easy sub-problems in this case are backtrack-free portions of the constraint graph. Another example is given by mathematical programming methods that use a problem's linear relaxation as an easy sub-problem.

Super-Optimal Probes (optimization only) *Optimizing by repairing probes that are super-optimal with respect to the optimization function.* Again this idea exists in MIP and other methods applying linear programming to relaxed problems.

9.2. Unimodular Probing

Probe backtracking restricted to unimodular probing is closely related to MIP. In MIP however, only one form of disjunctive constraint is allowed (the integrality constraint). MIP also differs from unimodular probing because it applies the linear solver to a linear relaxation of the problem constraints. Newer LP-based frameworks allow the expression of other disjunctive constraints [3, 19]. However they assume the linear solver is applied to the linear relaxation.

By limiting the linear solver to only sets of constraints having TU unimodular probing enables discrete problems to be addressed more efficiently. Resource scheduling problems are often fully discrete because of temporal granularity; for example, the commercial resource utilization problem that was investigated by the authors required flights to be fixed with a 5 minute resolution. Approximations obtained by rounding non-integer LP solutions to the linear relaxation lead to inconsistency and/or sub-optimality.

10. Discussion

Minimal perturbation variants of the KRFP show up very clearly the deficiencies of constraint programming (CP) and mathematical programming. CP algorithms are suited to the satisfaction of disjunctive constraints, but local heuristics and consistency methods are generally not effective at global optimization. In mathematical programming, the optimization function is central to the search, but disjunctive constraints are hard to satisfy.

The results of Sect. 8 show the unimodular probing strategy is capable of proving optimality for many problems where the constraint programming based alternatives fail to do so. The dramatic reduction in the number of LP solver calls by unimodular probing as opposed to MIP is also interesting.

Two issues arise from the results. First, the reasons for unimodular probing's improved performance on dynamic versions of the KRFP and the problem classes where similar improvements are likely, and second, the utility of probe backtracking in general. While these issues are by no means resolved here, some direction is given to the discussion.

10.1. Unimodular probing performance and applicability

The reasons for improvement over ordinary constraint programming search are more or less clear. Probe backtracking PB focuses CP search on a dynamically changing, high-quality but only partially consistent assignment. Traditionally, specialized local heuristics are constructed by the constraint programmer to reflect the optimization criterion. However, the unimodular probes have greater information content, because they take a global view of the optimization function, while simultaneously taking into account a subset of the problem constraints.

When compared to mathematical programming methods in general and MIP search in particular, unimodular probing appears to offer several advantages. First, local consistency methods are applied in unimodular probing, achieving a comparative reduction in the search space. Second, unimodular probing's search ordering heuristics are clearly better than MIP's. MIP sees all violations in terms of non-integrality in the problem's many integer variables, and thus typically faces a large set of very similar violations during search. Unimodular probing sees instead fewer, more heterogeneous violations of the problem's n-ary constraints. Moreover, the non-discrete values returned by LP for the relaxed problem in MIP have a limited meaning in a discrete context, and MIP heuristics based on them may not lead to well informed search choices. In unimodular probing, the probes give a relatively high level view of the sources of violation, enabling the use of meaningful heuristics for bottleneck prioritization and repair selection (the AI approach).

In terms of applicability, unimodular probing can be applied to discrete problems, with a linear optimization function and a TU constraint subset that includes the optimization variables. A newly reported result demonstrates that unimodular probing can yield performance benefits on other problems.[9]

10.2. Probe backtracking as a general strategy

The abstract probe backtracking procedure PB can lead to concrete strategies for many problems unrelated to dynamic scheduling or minimal perturbation. Unfortunately, the constraint programmer is faced with the potentially difficult task of constructing a fast and effective probe generation procedure, satisfying as many constraints as possible, and producing good, preferably super-optimal assignments (if an optimization requirement is present). However there are a number of cases where it seems probe backtracking may be especially worth the effort.

10.2.1. Problem decomposition If a problem has an easy or tractable sub-problem, probe solutions to it can aid search. The goal is to allow the backtrack algorithm to focus on solving hard sub-problems while maximizing the chance of success by making search decisions that maintain feasibility in the tractable part.

10.2.2. Algorithm hybridization Probe backtracking is useful when a specialized algorithm is well suited to solving a sub-problem. Section 9 remarked that mathematical programming methods integrate linear solvers with backtrack search in this way. Probe backtracking can be used to hybridize other solvers that generate partially consistent assignments. Local search algorithms such as hill climbing, simulated annealing, and genetic algorithms, conventionally viewed as substitutes for backtrack search, can act in support of it. This is achievable by configuring probe backtracking to apply search decisions that may be communicated to and enforced by the local search solver.

10.2.3. Optimization It seems that probing may be particularly suited to minimal perturbation optimization since the values of the minimal perturbation probe will tend to satisfy many of the problem constraints (minimal perturbation probes generally preserve feasible parts of the previous solution). Nevertheless, probing can be generally effective for optimization when probes provide a global and accurate focus on the optimization function during search.

The property of super-optimality in the probe is necessary for the completeness of repair-based optimization algorithms. However, even probes that return just good (not necessarily super-optimal) assignments may still be used to direct search. For hard problems, sub-optimal probes sacrifice theoretical completeness in exchange for improved average-case performance.

10.3. Conclusion

A probe backtracking strategy has been shown to be more effective than available alternatives for minimizing perturbation in dynamic scheduling. We conjecture that analogous probe backtracking strategies will be important for improving the performance of backtrack algorithms on many other constraint satisfaction and optimization problems.

Acknowledgments

We thank Thomas Richards for his valuable contributions to this paper, Eric Jacquet-Lagrèze and Denis Montaud for their expert advice on MIP search, and Thomas Schiex, Claude Le Pape and Andrew Davenport for their helpful comments on early versions. We would also like to express our gratitude to Joachim Schimpf and Stefano Novello for continually extending ECL^{i}PS^e in response to our requests, and finally to the reviewers for their valuable and detailed comments.

Notes

1. Bin packing algorithms may be used for resource assignment. Many surveys have been made for bin-packing, see for example [4].
2. $\mathbb{N}$ is the set of natural numbers.
3. Note that the Simplex methods are not polynomial time algorithms, even though the problem they address is tractable. It is possible to use polynomial time algorithms [21, 20], but Simplex is more widely used because it remains more efficient in the average case.
4. In the pseudo-code some peripheral data structures and parameters have been removed to improve legibility.
5. In fact, the deeper understanding of algorithm performance issues gained from these experimental results has led to some improvements in the commercial algorithm.
6. http://www.icparc.ic.ac.uk/ ~hhe/rfp_benchmarks.html
7. Problem sets 1 and 2 are both based on the schedule instances 1-50 on the web-site.
8. (*) Maximum shift scale sampled at irregular intervals.
9. Results of good performance on a chemical processing problem with a just-in-time optimization function have recently been presented [25].

References

1. Abderrahmane Aggoun and Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Premières Journées Francophones sur la Programation en Logique*, 1992.
2. M. Bartusch, R.H. Möhring, and F.J. Radermacher. Scheduling project networks with resource constraints and time windows. *Annals of Operations Research*, 16:201–240, 1988.
3. Henri Beringer and Bruno de Backer. Satisfiability of boolean formulas over linear constraints. In *IJCAI-93*, pages 296–301, Chambéry, France, August 1993.
4. E.G. Coffman Jr., M.R. Garey, and D.S.Johnson. Approximation algorithms for bin-packing — an updated survey. In G. Ausiello, M. Lucertini, and P. Serafini, editors, *Algorithm Design for Computer System Design*, pages 49–106. Springer, 1984.
5. A. Davenport. Managing uncertainty in scheduling: a survey. Working Draft, 1998.
6. R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proc. of AAAI-88*, pages 37–42, 1988.
7. R. Dechter and J. Pearl. Network-based heuristics for Constraint Satisfaction Problems. *Artificial Intelligence*, 34, 1988.
8. Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.
9. Amin El-Kholy and Barry Richards. Temporal and resource reasoning in planning: The *parc*PLAN approach. In *Proc. of the 11th European Conference on Artificial Intelligence, ECAI-96*, pages 614–618, Budapest, Hungary, 1996.
10. Amin O. El-Kholy. *Resource Feasibility in Planning.* PhD thesis, Imperial College, University of London, 1996.
11. Hani El Sakkout, Tom Richards, and Mark Wallace. Unimodular probing for minimal perturbance in dynamic resource feasibility problems. In *Proc. of the CP97 workshop on Dynamic Constraint Satisfaction*, 1997.
12. Hani El Sakkout, Tom Richards, and Mark Wallace. Minimal perturbation in dynamic scheduling. In *Proc. of the 13th European Conference on Artificial Intelligence, ECAI-98*, Brighton, UK, 1998.
13. M.S. Fox. *Constraint-directed search: a case study of job-shop scheduling.* Morgan Kaufmann Publishers Inc., 1987.
14. Simon French. *Sequencing and Scheduling: An introduction to the Mathematics of the Job-Shop.* Ellis Horwood, England, 1982.
15. Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Bell Telephone Laboratories, Inc., 1979.
16. Robert S. Garfinkel and George L. Nemhauser. *Integer Programming.* John Wiley & Sons, 1972.
17. I. Heller and C.B.Tompkins. An extension of a theorem of Dantzig's. In Kuhn and Tucker, editors, *Linear Inequalities and Related Systems*, pages 247–254. Princeton University Press, 1956.
18. D.W. Hildum. *Flexibility in a Knowledge-Based System for Solving Dynamic Resource-Constrained Scheduling Problems.* PhD thesis, Dept. of Computer Science, University of Massachusetts, Amherst, 1994.
19. J. N. Hooker and M.A.Osorio. Mixed logical/linear programming. *Discrete Applied Mathematics (to appear)*, 1996. Electronic copy available from first author's home page: http://www.gsia.cmu.edu/afs/andrew.cmu.edu/gsia/jh38/papers.html.
20. N. Karmarkar. A new polynomial-time algrithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.
21. L.G. Khachian. A polynomial algorithm in linear programming. *Soviet Math. Dokl.*, 20(1):191–194, 1979.
22. Claude Le Pape, Philipe Couronné, Didier Vergamini, and Vincent Gosselin. Time-versus-capacity compromises in project scheduling. In *Proc. of the 13th Workshop of the UK Planning and Scheduling SIG*, Glasgow, Scotland, 1994.
23. V.J. Leon, S.D. Wu, and R.H.Storer. Robustness measures and robust scheduling for job shop. *IIE Transactions*, 26(5):32–43, 1994.

24. Olivier Lhomme. Consistency techniques for numeric csps. In *Proc. of the 13th International Joint Conference on Artificial Intelligence, IJCAI-93*, pages 232–238, Chambéry, France, 1993.
25. Vassilios Liatsos. Short term scheduling. Presentation at the DIMACS Workshop on Constraint Programming and Large-Scale Discrete Optimization, Rutgers University, NJ, Sept. 1998.
26. Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.
27. W. Nuijten and E. Aarts. Constraint satisfaction for multiple capacitated job shop scheduling. In *Proc. of the 11th European Conference on Artificial Intelligence, ECAI-94*. John Wiley & Sons, Ltd., 1994.
28. Wim Nuijten. *Time and Resource Constrained Scheduling: A constraint satisfaction approach.* PhD thesis, Eindhoven University of Technology, 1994.
29. Dionysios Pothos. A constraint-based approach to the british airways schedule re-timing problem. Technical Report 97/04-01, IC-Parc, Imperial College, 1997.
30. Paul Walton Purdom, Jr. and G. Neil Haven. Probe order backtracking. *Siam Journal of Computing*, 26:456–483, 1997.
31. M. Queyranne and Y. Wang. Single-machine scheduling polyhedra with precedence constraints. *Mathematics of Operations Research*, pages 1–20, 1991.
32. N. Sadeh. Micro-opportunistic scheduling: The micro-boss factory scheduler. In M. Zweben and M. Fox, editors, *Intelligent Scheduling*, chapter 4, pages 99–136. Morgan Kaufman, 1994.
33. Gérard Verfaillie and Thomas Schiex. Solution reuse in dynamic constraint satisfaction problems. In *AAAI-94*, pages 307–312, Seattle, WA, August 1994.
34. Makoto Yokoo. Weak-commitment search for solving constraint satisfaction problems. In *AAAI-94*, pages 313–318, Seattle, WA, August 1994.