The Knowledge Engineering Review, Vol. 16:1, 2001, 69–84
Printed in the United Kingdom
Copyright © 2001, Cambridge University Press

Synthesis of efficient constraint-satisfaction programs

STEPHEN J. WESTFOLD and DOUGLAS R. SMITH

Kestrel Institute, 3260 Hillview Avenue, Palo Alto, California 94304, USA
E-mail: westfold@kestrel.edu, smith@kestrel.edu

Abstract

In this paper we describe the framework we have developed in KIDS (Kestrel Interactive Development System) for generating efficient constraint satisfaction programs. We have used KIDS to synthesise global search scheduling programs that have proved to be dramatically faster than other programs running the same data. We focus on the underlying ideas that lead to this efficiency. The key to the efficiency is the reduction of the size of the search space by an effective representation of sets of possible solutions (solution spaces) that allows efficient constraint propagation and pruning at the level of solution spaces. Moving to a solution space representation involves a problem reformulation. Having found a solution to the reformulated problem, an extraction phase extracts solutions to the original problem. We show how constraints from the original problem can be automatically reformulated and specialised in order to derive efficient propagation code automatically. Our solution methods exploit the semi-lattice structure of our solution spaces.

1 Introduction

Software synthesis is the process of transforming a formal problem specification into software that is efficient and correct by construction. We have used KIDS (Kestrel Interactive Development System) over the last ten years to synthesise very efficient programs for a variety of scheduling problems (Burstein & Smith, 1996; Smith *et al.*, 1996; Smith & Westfold, 1995). The efficiency of these scheduling systems is based on the synthesis of specialised constraint management code for achieving arc-consistency. Previous systems for performing scheduling in AI (e.g. Fox *et al.*, 1989; Fox & Smith, 1984; FS Smith, 1989; FS Smith *et al.*, 1986) and operations research (Applegate & Cook, 1991; Luenberger, 1989) use constraint representations and operations that are geared for a broad class of problems, such as constraint satisfaction problems or linear programs. In contrast, synthesis techniques can derive specialised representations for constraints and related data, and also derive efficient specialised code for constraint-checking and constraint propagation. Synthesis technology allows us to specialise the code by exploiting both problem-independent knowledge (theory of linear programming, finite domains and so on) as well as problem-specific information obtained from the problem specification.

In this paper we focus on the underlying ideas that lead to the efficiency of our synthesised scheduling programs and explore their generality. Our constraint-satisfaction algorithms fall into the class of *global search* algorithms. We have explored this class in more depth elsewhere (DR Smith, 1987; DR Smith & Westfold, 1995). Here we focus on a formulation that uses points in a semi-lattice to represent solution spaces. This semi-lattice framework naturally allows for heterogeneous variables

https://doi.org/10.1017/S0269888901000029 Published online by Cambridge University Press

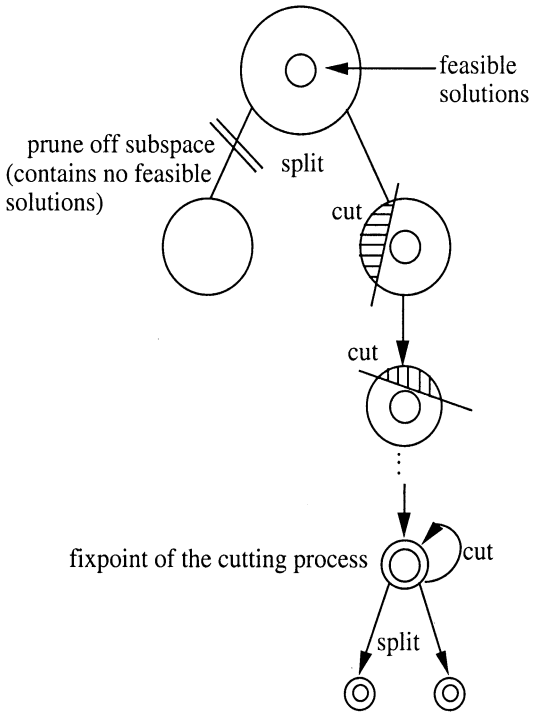


Figure 1 Pruning and constraint propagation

(variables of different types), multiple constraints on each variable, indexed variables, conditional constraints and a dynamic set of variables.

The basic idea of global search is to represent and manipulate sets of candidate solutions or *solution spaces*. The principal operations are to *extract* candidate solutions from a solution space and to *split* a space into subspaces. Derived operations include (1) *filters* which are used to eliminate spaces containing no feasible or optimal solutions and (2) *cutting constraints* that are used to eliminate non-feasible elements from a space of candidate solutions. Global search algorithms work as follows: starting from an initial space that contains all solutions to the given problem instance, the algorithm repeatedly splits spaces into subspaces (*refinements* of the space) eliminates spaces via filters and contracts spaces by propagating cutting constraints until no spaces remain to be split. The process is often described as a tree (or DAG) search in which a node represents a solution space and an arc represents the split relationship between space and subspace. The filters and constraint propagators serve to prune off branches of the tree that cannot lead to solutions. See Figure 1 which illustrates the working of pruning and constraint propagation on solution spaces.

The key to the efficiency of global search algorithms is the reduction of the size of the search space by an effective representation of solution spaces that allows constraint propagation and pruning at the level of solution spaces rather than individual concrete solutions.

There is a certain amount of freedom in the formulation of solution spaces. We have found that it is desirable to choose a formulation with a (meet) semi-lattice structure.[1] A solution space is represented by a point in the semi-lattice which is the greatest lower bound of the solution space. This structure gives you the property that a solution space for one constraint can be combined with the solution space of another constraint (using the *meet* operation – $\sqcup$) to produce the solution space for the conjunction of the two constraints.

Rehof and Mogensen (1999) have shown that satisfiability of sets of *definite* inequality constraints involving monotone functions in a finite meet semi-lattice can be decided by a linear-time algorithm

[1] This is not a limiting restriction as it is always possible to convert an arbitrary domain into a lattice by *lifting*: adding a bottom element.

https://doi.org/10.1017/S0269888901000029 Published online by Cambridge University Press

using a fixpoint iteration. Their notion of *definite inequality* generalises the logical notion of Horn clauses from the two-point Boolean lattice of the truth values to arbitrary finite semi-lattices. One indication of the power of this class is that arc consistency implies global consistency. We refer to the Rehof and Mogensen algorithm as *algorithm RM*. In CSP terms, a definite inequality problem has the desirable property that arc-consistency implies globally consistency. The algorithm RM is a (hyper)arc-consistency algorithm.

Our work can be viewed as adding disjunctive constraints to their framework.[2] The disjuncts are handled primarily by a search that incrementally generates conjunctive problems that are solved efficiently using essentially algorithm RM.

2 Framework

We use the language of first-order predicate calculus for specification. A *binary relation* on a set S is a subset of the product $S \times S$. A *function* f from a set A to a set B, written $f: A \rightarrow B$, is a subset of $A \times B$ such that for each $a \in A$ there is exactly one $b \in B$ with $(a, b) \in f$. In this case we write $f(a) = b$. A definition of f is presented as follows:

Definition $f(x) = e$

where e is an expression that may involve the variable x. If the domain of f is itself a product then f applied to the tuple (a, b) is written $f(a, b)$.

Definition. A binary relation $\sqsubseteq$ defined on a set S is a *partial order* in the set S if the following conditions hold:

$\forall(x) x \sqsubseteq x$ (**reflexivity**)

$\forall(x, y)\ (x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y)$ (**antisymmetry**)

$\forall(x, y, z)\ (x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z)$. (**transitivity**)

A function f is *monotone* with respect to the partial order if $\forall(x, y)(x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y))$.

Definition. Let $\sqsubseteq$ be a partial order on S and T a subset of S. An element p in S is a *least upper bound* of T if $\forall(x)\ (x \in \mathrm{T} \Rightarrow x \sqsubseteq p)$ and $\forall(y)\ (y \in S \wedge\ \forall(x)\ (x \in T \Rightarrow x \sqsubseteq y) \Rightarrow p \sqsubseteq y)$. Similarly an element p in S is a *greatest lower bound* of T if $\forall(x)\ (x \in T \Rightarrow p \sqsubseteq x)$ and $\forall(y)\ (y \in S \wedge\ \forall(x)\ (x \in T \Rightarrow y \sqsubseteq x) \Rightarrow y \sqsubseteq p)$.

Definition. A set S with partial order $\sqsubseteq$ is a *meet semi-lattice* iff every pair a, b in S has a least upper bound. $a \sqcup b$ (or $\sqcup(a, b)$) is defined to be the least upper bound of $\{a, b\}$. Similary, it is a *join semi-lattice* iff every pair a, b in S has a greatest lower bound, and $a \sqcap b$ is defined as this greatest lower bound. It is a *lattice* if it is both a meet semi-lattice and a join semi-lattice.

We refer to meet and join semi-lattices using the structures $(S, \sqsubseteq, \sqcup)$ and $(S, \sqsubseteq, \sqcap)$ and full lattices using the structure $(S, \sqsubseteq, \sqcup, \sqcap)$. Example lattices are $(Integer, \leq, max, min)$, $(Set, \subseteq, \cup, \cap)$, and the Boolean lattice $(\{true, false\}, \Rightarrow, \vee, \wedge)$.

Two semi-lattices $(S_1, \sqsubseteq_1, \sqcup_1)$ and $(S_2, \sqsubseteq_2, \sqcup_2)$ can be combined to make a *product semi-lattice* $(S_1 \times S_2, \sqsubseteq_p, \sqcup_p)$ where

definition $x \sqsubseteq_p y = (x.1 \sqsubseteq_1 y.1 \wedge x.2 \sqsubseteq_2 y.2)$

definition $x \sqcup_p y = (x.1 \sqcup_1 y.1, x.2 \sqcup_2 y.2)$.

A semi-lattice $(S, \sqsubseteq, \sqcup)$ or $(S, \sqsubseteq, \sqcap)$ may have *top* $(\top)$ and/or *bottom* $(\bot)$ elements, defined as follows:

$\forall(x)\ (x \in \mathrm{S} \Rightarrow x \sqsubseteq \top)$

$\forall(x)\ (x \in S \Rightarrow \bot \sqsubseteq x)$.

[2] Our work was independent of that of Rehof and Mogensen and focused on the disjunctive aspects of the problem and the synthesis of constraint-solving codes.

https://doi.org/10.1017/S0269888901000029 Published online by Cambridge University Press

The computational significance of the meet semi-lattice structure is that it allows two constraints on a variable X of the form $c_1 \sqsubseteq X$ and $c_2 \sqsubseteq X$, where c_1 and c_2 are constants in a lattice, to be replaced by the equivalent single constraint $c_3 \sqsubseteq X$ where $c_3 = c_1 \sqcup c_2$.

We follow the definitions of Rehof and Mogensen in introducing the concept of *definite inequalities*.

Definition. An inequality is called *definite* if it has the form $\tau \sqsubseteq A$, where A is an atom (a constant or a variable) and τ is a term whose functions are all monotone.

Rehof and Mogensen showed that satisfiability of a set (conjunction) of definite inequalities can be decided in linear time for a meet semi-lattice domain. For the two-point Boolean lattice this is exactly the satisfiability of propositional Horn clauses (HornSAT) problem (since Horn clauses have the form $P_1 \wedge \ldots \wedge P_n \Rightarrow Q$, which is the inequality $P_1 \sqcap \ldots \sqcap P_n \sqsubseteq Q$ in the Boolean lattice).

The problem we are considering in this paper is solving sets (conjunctions) of disjunctions of definite inequalities over meet semi-lattices. This problem is NP-complete with propositional satisfiability (SAT) as an instance.

Although the problem formulation allows any number of variables, only one is necessary, as we can replace n variables by a single variable whose value is the n-tuple of the values of the n variables. The domain of the single variable is the product semi-lattice of the n semi-lattices of the domains of the variables. This allows heterogeneous problems, where the variable domains are different, to be handled in our framework. Similarly, dynamic problems, where the number of variables constrained can increase during problem-solving, can be handled via a semi-lattice structure on dynamic maps and sequences.

For example, consider the heterogeneous problem with two semi-lattices $(S_1, \sqsubseteq_1, \sqcup_1)$ and $(S_2, \sqsubseteq_2, \sqcup_2)$, and functions $f_1 : S_1 \times S_2 \rightarrow S_1$ and $f_2 : S_1 \times S_2 \rightarrow S_2$, with the following two constraints on the variables $A_1 : S_1$ and $A_2 : S_2$.

$$f_1(A_1, A_2) \sqsubseteq_1 A_1$$

$$f_2(A_1, A_2) \sqsubseteq_2 A_2.$$

These are equivalent to

$$f_1(A.1, A.2) \sqsubseteq_1 A.1$$

$$f_2(A.1, A.2) \sqsubseteq_2 A.2$$

where $A = (A_1, A_2)$.

The transformed constraint set is not yet in definite form because the right-hand sides are not A but they can be transformed using the following equivalence for a product semi-lattice $(S_1 \times S_2 \sqsubseteq_p, \sqcup_p)$:

$$\tau \sqsubseteq_i A.i \Leftrightarrow \text{tuple-shadow}(A, i, \tau) \sqsubseteq_p A$$

where tuple-shadow(tp, i, x) is the tuple tp with the i^{th} component replaced by x.

The transformed constraint set in definite form is

$$\text{tuple-shadow}(A, 1, f_1(A.1, A.2)) \sqsubseteq_p A$$

$$\text{tuple-shadow}(A, 2, f_2(A.1, A.2)) \sqsubseteq_p A.$$

In the examples below we use the component forms of inequalities as they are simpler.

A naive method for solving our problem of a conjunction of disjunctions of definite inequalities is to distribute conjunction over disjunction to get a disjunction of conjunctions. Algorithm RM can be used to solve each conjunction of definite inequalities independently and the derived solution sets can then be unioned to create the total solution. However, this is very inefficient in general.

We now describe a global search algorithm for this problem that exploits the incremental nature of algorithm RM: the solution to a definite constraint set S can be used instead of $\perp$ as the starting point for the fixpoint iteration to find the solution to S with an extra definite constraint. Each node in the global search tree consists of a solution to a conjunctive problem plus the set of remaining disjunctions.

https://doi.org/10.1017/S0269888901000029 Published online by Cambridge University Press

The top node consists of the solution to the empty problem, $\perp$ of the semi-lattice, and the initial problem as a set of disjunctions. Children nodes are incrementally refined from their parent by choosing one remaining disjunction and creating a child node for each disjunct by adding the disjunct to the conjunctive problem solved by the parent and iterating to a fixpoint using algorithm RM.

Constraint propagation is used to eagerly reduce the size of remaining disjunct sets, pruning the space when a disjunct set becomes empty and incorporating the remaining disjunct of a unary disjunction into the current conjunctive problem. We add a $\top$ to the semi-lattice and all its components to represent the space becoming empty which means failure in the search tree. Whenever propagation leads to any component becoming $\top$, then the whole space becomes $\top$ and the branch can be pruned.

We do not discuss the choice of which disjunct set to expand and what order to explore disjuncts, as conventional considerations such as disjunct set size are applicable. In the large scheduling problems we have focused on, there are a large number of very large disjunct sets, so it is desirable to represent them procedurally rather than explicitly.

In the rest of the paper we illustrate the process of deriving a global search algorithm by applying it to an example. The steps are:

1. specify problem to be solved,
2. derive reformulation into disjunctions of definite inequalities,
3. generate code for splitting solution spaces,
4. generate constraint propagation code and
5. generate code for extracting solutions for the original problem.

All steps after the problem specification are performed automatically by the KIDS system.

3 Simple scheduling example

We describe a simple transportation scheduling problem that includes essential features found in our real scheduling problems.[3]

The input is a set of *MVR*s, where an MVR is a *MoVement Requirement* – a description of cargo that has to be moved. In this simple version an MVR includes information about when the cargo is available to be moved (*release-time*) and by when it must arrive (*due-time*).

We have a single transportation resource to be scheduled (e.g. a plane), so a schedule is a single sequence of *trips*. Each trip has a start time and a set of MVRs it has been assigned to fulfil: *sched*(*i*).trip-start is the start time of the i^{th} trip in schedule *sched*, and *sched*(*i*).trip-MVRs is the manifest of the i^{th} trip in *sched* – the MVRs assigned to that trip.

Figure 2 gives an example of a scheduling problem and a solution schedule. The release-times and due-times of the five MVRs (labelled for convenient reference *a*, *b*, *c*, *d* and *e*) to be scheduled are plotted above the time axis (the vertical axis has no meaning) and a solution schedule consisting of three trips is plotted beneath it. Trips 1 and 2 start at the earliest possible time (given their manifests) whereas trip 3 starts somewhat later than the earliest possible time. There are many similar solution schedules with slight shifts in the start times.

The specification of valid schedules is

definition **valid-schedules**(*MVRs*) =
  {*sched* | all-MVRs-scheduled(*MVRs*, *sched*)
    ∧ MVRs-ready(*sched*)
    ∧ MVRs-due(*sched*)
    ∧ trip-separations(*sched*)}.

[3] The problem is simplified to the extent that it is not NP-complete. Adding capacity bounds would be sufficient to make finding a feasible solution NP-complete.

https://doi.org/10.1017/S0269888901000029 Published online by Cambridge University Press

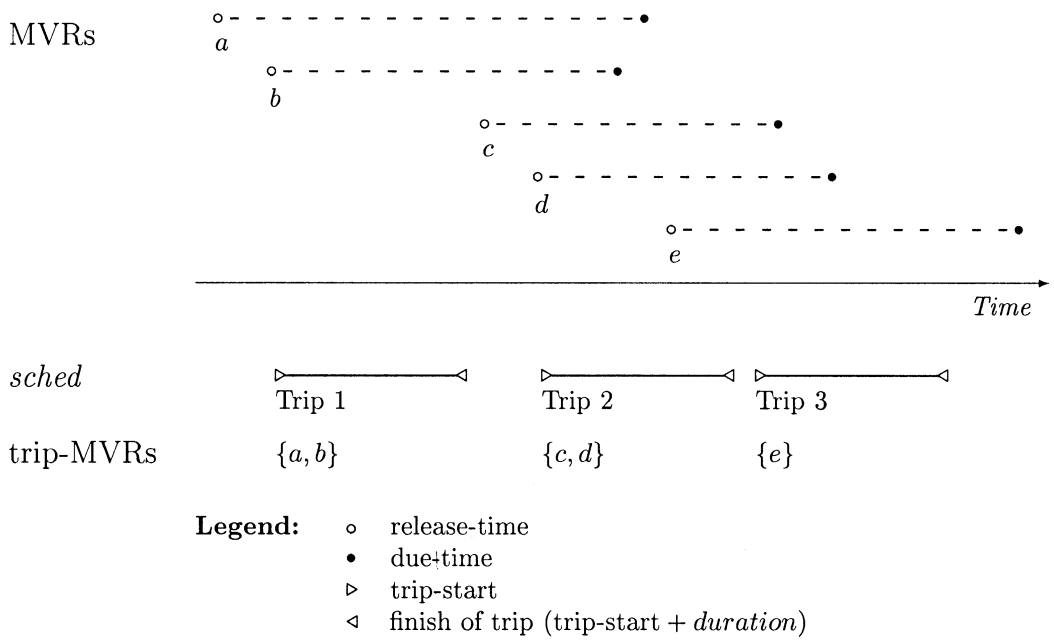


Figure 2 Scheduling problem and solution

Every MVR has to be scheduled on some trip.[4]

definition **all-MVRs-scheduled**(*MVRs*, *sched*) =
  ∀(*m*: *MVR*)
    (*m* ∈ *MVRs*
      ⇒ ∃(*i*) (*i* ∈ {1 . . . *size*(*sched*)} ∧ *m* ∈ *sched*(*i*) . trip-MVRs)).

A trip cannot start until after the release dates of all the MVRs assigned to it.

definition MVRs-ready(*sched*) =
  ∀(*i* : *Integer*, *m* : *MVR*)
    (*i* ∈ {1 . . . *size*(*sched*)} ∧ *m* ∈ *sched*(*i*) . trip-MVRs
      ⇒ release-time(*m*) ≤ *sched*(*i*) . trip-start).

A trip must complete before all the due dates of the MVRs assigned to it.

definition **MVRs-due**(*sched*) =
  ∀(*i* : *Integer*, *m* : *MVR*)
    (*i* ∈ {1 . . . *size*(*sched*)} ∧ *m* ∈ *sched*(*i*) . trip-MVRs
      ⇒ due-time(*m*) − *duration* ≥ *sched*(*i*) . trip-start).

Typically the duration is a function of the properties of a trip, but this has little effect on the derivations in this paper so we make it a constant for simplicity.

[4] This constraint does not preclude an MVR from appearing on more than one trip, but the derived algorithm does not try to make a constraint true if it is already true, so it does not put an MVR on another trip if it is already on one.

https://doi.org/10.1017/S0269888901000029 Published online by Cambridge University Press

A trip cannot begin until the previous trip has ended.

definition **trip-separations**(*sched*) =
  ∀(*i* : *Integer*)
    (*i* ∈ {1 . . . *size*(*sched*) − 1}
      ⇒ *sched*(*i*) . trip-start + *duration* ≤ *sched*(*i* + 1) . trip-start).

Again, typically there is a gap between the end of one trip and the beginning of the next, but our derivations only depend on the lack of overlap among trips.

4 Reformulation of constraints

The scheduling constraints are not disjunctive definite inequalities as specified. The most fundamental problem is that they give both upper and lower bounds on the trip-start component of trips. In other words, the full lattice structure of time is being used. We address this problem by considering the lattice as two semi-lattices, one for increasing time and the other for decreasing time. We introduce components for greatest lower bound (earliest-trip-start) and least upper bound (latest-trip-start). These are related to trip-start:

x . earliest-trip-start ≤ *x* . trip-start ≤ *x* . latest-trip-start

From these bounds, we can infer versions of the constraints that have the form of definite inequalities, giving us a reformulated problem.

Figure 3 shows the reformulated version of the problem given in Figure 2 with its solution. The Figure 2 solution can be extracted by taking the earliest start and finish times of the first two trips and a slightly later time for the third trip. The earliest-trip-start of trips 1 and 2 is the same as the release-time of MVRs *b* and *d* respectively, because of the **MVRs-ready** constraint. The earliest-trip-start of trip 3 is the same as the earliest finish time of trip 2 which is later than the release-time of MVR *e*

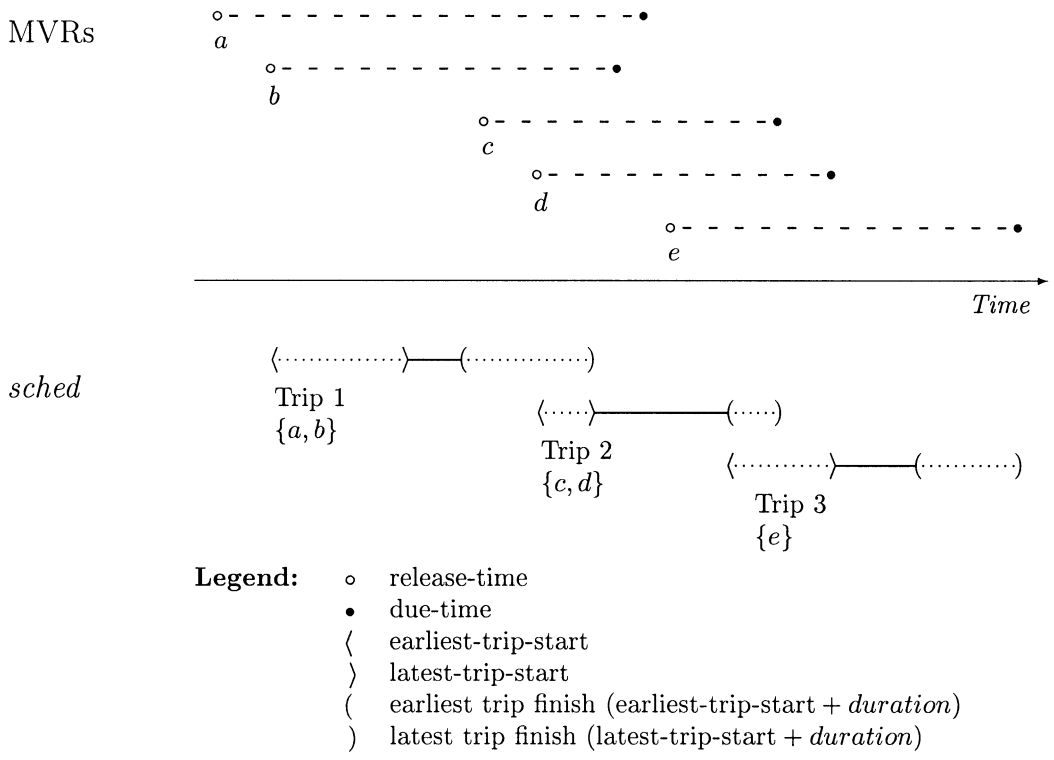


Figure 3 Reformulated scheduling problem solution

https://doi.org/10.1017/S0269888901000029 Published online by Cambridge University Press

because of the **trip-separations** constraint. Similarly, the latest trip finish time of trips 2 and 3 is the same as the due-time of MVRs *c* and *e* respectively, because of the **MVRs-due** constraint, and the latest trip finish time of trip 1 is the same as the latest-trip-start of trip 2 because of the **trip-separations** constraint. The latest-trip-start of a trip is its latest trip finish less *duration*.

The reformulated problem has only one solution for a given assignment of MVRs to trip-MVRs. All solutions to the original problem can be extracted from the solution to the reformulated problem. Not all values selected from the ranges are compatible, for example choosing the latest-trip-start for trip 1 and the earliest-trip-start for trip 2 is incompatible with the **trip-separations** constraint. However, there is at least one solution for any trip-start chosen from the range of earliest-trip-start and latest-trip-start.

Here is a simplified treatment of the inference for the definition of **MVRs-ready**. Ignoring the antecedent and incorporating the bounds on trip-start, we essentially have

$$\forall(i : \mathit{Integer},\ m : \mathit{MVR})$$
$$(\mathit{sched}(i)\,.\,\text{earliest-trip-start} \le \mathit{sched}(i)\,.\,\text{trip-start}$$
$$\Rightarrow \text{release-time}(m) \le \mathit{sched}(i)\,.\,\text{trip-start}).$$

then using the general law

$$\forall(z)(a \le z \Rightarrow b \le z) = b \le a$$

we obtain the equivalent expression

$$\forall(i : \mathit{Integer},\ m : \mathit{MVR})\ \text{release-time}(m) \le \mathit{sched}(i)\,.\,\text{earliest-trip-start}.$$

Using a generalised version of this key step, **MVRs-ready** and **MVRs-due** are easily reformulated to the equivalent definite inequalities

definition **MVRs-ready**′$(\mathit{sched}) =$
$$\forall(i : \mathit{Integer},\ m{:}\ \mathit{MVR})$$
$$(i \in \{1..\mathit{size}(\mathit{sched})\} \wedge m \in \mathit{sched}(i)\,.\,\text{trip-MVRs}$$
$$\Rightarrow \text{release-time}(m) \le \mathit{sched}(i)\,.\,\text{earliest-trip-start}).$$

definition **MVRs-due**′$(\mathit{sched}) =$
$$\forall(i : \mathit{Integer},\ m : \mathit{MVR})$$
$$(i \in \{..\mathit{size}(\mathit{sched})\} \wedge m \in \mathit{sched}(i)\,.\,\text{trip-MVRs}$$
$$\Rightarrow \text{due-time}(m) - \mathit{duration} \ge \mathit{sched}(i)\,.\,\text{latest-trip-start}).$$

The **trip-separations** constraint is reformulated in a similar manner except that we obtain two constraints because it can be used to get lower bounds on $\mathit{sched}(i+1)\,.\,\text{trip-start}$ and upper bounds on $\mathit{sched}(i)\,.\,\text{trip-start}$. Again ignoring the antecedent and incorporating the bounds, we essentially have

$$\forall(i : \mathit{Integer},\ m{:}\ \mathit{MVR})\forall(\mathit{sched} : \mathit{schedule})$$
$$(\mathit{sched}(i)\,.\,\text{earliest-trip-start} \le \mathit{sched}(i)\,.\,\text{trip-start} \le \mathit{sched}(i)\,.\,\text{latest-trip-start}$$
$$\Rightarrow \mathit{sched}(i)\,.\,\text{trip-start} + \mathit{duration} \le \mathit{sched}(i+1)\,.\,\text{trip-start})$$

then we can use monotonicity of $\le$ on either occurrence of trip-start. Here we apply monotonicity to the first occurrence:

$$\forall(i : \mathit{Integer},\ m{:}\ \mathit{MVR})\forall(\mathit{sched} : \mathit{schedule})$$
$$(\mathit{sched}(i)\,.\,\text{earliest-trip-start} \le \mathit{sched}(i)\,.\,\text{trip-start}$$
$$\Rightarrow \mathit{sched}(\text{i})\,.\,\text{trip-start} + \mathit{duration} \le \mathit{sched}(i+1)\,.\,\text{trip-start})$$

$\Rightarrow$ (monotonicity)

https://doi.org/10.1017/S0269888901000029 Published online by Cambridge University Press

$\forall(i : Integer,\ m{:}\ MVR)\forall(sched : schedule)$
$(sched(i)\,.\,\text{earliest-trip-start} \le sched(i)\,.\,\text{trip-start}$
$\Rightarrow sched(\text{i})\,.\,\text{earliest-trip-start} + duration \le sched(i+1)\,.\,\text{trip-start})$

$\Leftrightarrow$ (using the law $\forall(z)(z \le a \Rightarrow z \le b) = a \le b$)

$\forall(i : Integer,\ m{:}\ MVR)$
$sched(i)\,.\,\text{earliest-trip-start} + duration \le sched(i+1)\,.\,\text{earliest-trip-start}).$

The final result is shown below, after we do some normalisation, introducing a variable t for the τ expression of the definite inequality. This reduces the cases that need to be considered when deriving specialised constraints (as described in the next section).

definition **trip-separations-ets**$(sched) =$
$\forall(i : Integer,\ t : Time)$
$(i \in \{1..size(sched) - 1\} \wedge t = sched(i)\,.\,\text{earliest-trip-start} + duration$
$\Rightarrow t \le sched(i+1)\,.\,\text{earliest-trip-start}).$

An analogous derivation, stemming from application of monotonicity to the second occurrence of trip-start, results in

definition **trip-separations-lts**$(sched) =$
$\forall(i : Integer,\ t : Time)$
$(i \in \{1..size(sched) - 1\} \wedge t = sched(i+1)\,.\,\text{latest-trip-start} - duration$
$\Rightarrow t \ge sched(i)\,.\,\text{latest-trip-start}).$

5 Generating splitting code

The **all-MVRs-scheduled** constraint provides the disjunction that leads to the global search splitting.

$\forall(m : MVR)$
$(m \in MVRs$
$\Rightarrow \exists(i)\ (i \in \{1..size(sched)\} \wedge m \in sched(i)\,.\,\text{trip-MVRs})).$

This is not in the form of a set of disjunctions of definite inequalities. However, it is equivalent to the form

$$\bigwedge_{m \in MVRs}\ \bigvee_{i \in \{1..size(sched)\}} \{m\} \subseteq sched(i)\,.\,\text{trip-MVRs}$$

which is an indexed conjunction of disjunctions of definite inequalities in the meet semi-lattice of sets with union as meet for the trip-MVRs component of a trip.

Thus for every MVR there is a disjunction consisting of the MVR being on any one of the trips of the schedule. However, the number of trips is not known in advance. At each point in the elaboration of the search tree, the MVR could be added to an existing trip or a new trip could be created and the MVR added to it. Thus there are two kinds of incremental refinement to the solution space: adding an MVR to a trip and adding a new empty trip (to which we then add an MVR).

These two basic refinements can be expressed:

$sched'(i)\,.\,\text{trip-MVRs} = sched(i)\,.\,\text{trip-MVRs} \cup \{m\}$

$sched' = append(sched,\ trip_{\perp})$ (**append-new-trip**)

https://doi.org/10.1017/S0269888901000029 Published online by Cambridge University Press

where *sched* is the solution space of schedules before the refinement, *sched′* is the solution space of schedules after the refinement and $trip_{\perp}$ is the empty trip, the bottom of the semi-lattice of trips.[5]

To use these refinements it is convenient to have them in the form $A' = f(A)$. The second is already in this form; the first can be converted to

sched′ = seq-shadow(*sched*, *i*, (**update-trip-MVRs**)
tuple-shadow(*sched*(*i*), trip-MVRs,
sched(*i*).trip-MVRs ∪ {m}))

where seq-shadow(*S*, *i*, *x*) is equal to sequence *S* except that at index *i* its value is *x*.

6 Specialised constraint propagation code

The constraints are still not in the form of definite inequalities. Apart from **all-MVRs-scheduled** they have the form

$$\forall(x)(p(A, x) \Rightarrow x \sqsubseteq A).$$

However, this is equivalent to either of the forms

$$reduce(\sqcup, \{x \mid p(A, x)) \sqsubseteq A$$

$$\bigwedge_{x \mid p(A, x)} x \sqsubseteq A$$

which are definite inequalities provided that *p* is monotonic, which is the case for our examples.

Our derivation of propagation code works with the quantified implication form, the propagation being forward inference. After every refinement one could re-evaluate all the bounds from scratch but for efficiency it is desirable to specialise the constraint to the particular refinement. We exploit the properties that the constraint was true in the previous solution space and that we have just made an incremental refinement of this space.

We generate a procedure for each different kind of incremental refinement to the solution space (such as **append-new-trip** and **update-trip-MVRs**) and include in each procedure constraint-checking and propagation code that is specialised to the particular incremental refinement. For example, when appending a new trip to the end of the sequence of trips, the earliest start time of the new trip may be dependent on the earliest start time of the previous trip, and the latest start time of the new trip may affect the latest start time of the previous trip. Our concern in this section is how to derive such dependencies automatically.

First we describe an abstract version of the derivations. In the next subsection we give a detailed derivation of specializing a particular constraint, and then we give brief derivations of specialisation of the other constraints.

Consider an incremental refinement to *A* by some function *g*:

$A' = g(A)$ (*so* $A \sqsubseteq A'$)

and a definite constraint *C* on *A* given by

$C(A) = \forall(x)(p(A, x) \Rightarrow x \sqsubseteq A).$

We assume that the constraint *C* is true for the initial space *A* and we want it to be true in the refined space *A′*. *C*(*A*) is true if for all *x*, *p*(*A*, *x*) is false or $x \sqsubseteq A$ is true. In the latter case we have simply that $x \sqsubseteq A'$ by transitivity, so we can use *A* as an initial approximation of *A′*. To find additional values of *x* that belong in *A′* we assume the former case: we simplify *C*(*A′*) under the assumption that *p*(*A*, *x*) is false. We call this simplified *C*(*A′*) the *residual* constraint for *C* given the refinement.

[5] For completeness it is necessary also to consider the new trip being inserted into the sequence, but to reduce the search space we just consider the append case. As we schedule MVRs in order of due date, this incompleteness has not proved to be a problem in practice.

https://doi.org/10.1017/S0269888901000029 Published online by Cambridge University Press

For the common case where p is a conjunction this does not work well in practice because the negation is a disjunction which is difficult to use in simplification. Instead we treat each conjunct separately and combine the results as follows: if $p(A, x) = (p_1(A, x) \wedge p_2(A, x))$ and the residuals for $p_1(A, x)$ and $p_2(A, x)$ are $h_1(A, x)$ and $h_2(A, x)$ respectively then the full residual constraint is

$$\forall(x)(h_1(A, x) \wedge p_2(A', x) \Rightarrow x \sqsubseteq A')$$
$$\wedge \forall(x)(p_1(A', x) \wedge h_2(A, x) \Rightarrow x \sqsubseteq A').$$

If a constraint is unaffected by the refinement then its residual is *false*.

6.1 Specialising MVRs-ready′

We now use this derivation scheme to derive the residual constraint for **MVRs-ready′**:

$$\forall(i : \mathit{Integer}, m : \mathit{MVR})$$
$$(i \in \{1..\mathit{size}(\mathit{sched})\} \wedge m \in \mathit{sched}(i).\text{trip-MVRs}$$
$$\Rightarrow \text{release-time}(m) \leq \mathit{sched}(i).\text{earliest-trip-start})$$

given the **update-trip-MVRs** refinement

$$\mathit{sched}' = \text{seq-shadow}(\mathit{sched}, i_c, \quad \text{(update-trip-MVRs′)}$$
$$\text{tuple-shadow}(\mathit{sched}(i_c), \text{trip-MVRs},$$
$$\mathit{sched}(i_c).\text{trip-MVRs} \cup \{m_c\})).$$

The variables are subscripted with a c to avoid name conflict in the derivation and to emphasise that in this context they have already been bound to particular values.

The residual for the conjunct $i \in \{1..\mathit{size}(\mathit{sched})\}$ is *false* because $\mathit{size}(\mathit{sched}')$ simplifies to $\mathit{size}(\mathit{sched})$ as seq-shadow does not affect the size of the sequence.

Now we focus on the second conjunct, $m \in \mathit{sched}'(i).\text{trip-MVRs}$, simplifying it under the assumption

$$m \in \mathit{sched}(i).\text{trip-MVRs} = \mathit{false} \quad \text{(negation-assumption)}$$

We require the following rules:

$$\text{seq-shadow}(S, i, x)\ (j) = \textbf{if } i = j \textbf{ then } x \textbf{ else } S(j) \quad \text{(seq-shadow-application)}$$

$$\text{tuple-shadow}(x, f, y).\ f = y \quad \text{(tuple-shadow-deref)}$$

$$(\textbf{if } x \textbf{ then } y \textbf{ else } \mathit{false}) = (x \wedge y) \quad \text{(if-then-else-false)}$$

The simplification goes

$m \in \mathit{sched}'(i).\text{trip-MVRs}$

{substitute **update-trip-MVRs′**}

$= m \in \text{seq-shadow}(\mathit{sched}, i_c, \text{tuple-shadow}(\mathit{sched}(i_c), \text{trip-MVRs}, \mathit{sched}(i_c).\text{trip-MVRs} \cup \{m_c\}))$
$(i).\text{trip-MVRs}$

{**seq-shadow-application**}

$= m \in (\textbf{if } i = i_c \textbf{ then } \text{tuple-shadow}(\mathit{sched}(i_c), \text{trip-MVRs}, \mathit{sched}(i_c).\text{trip-MVRs} \cup \{m_c\})$
$\textbf{else } \mathit{sched}(i)).\text{trip-MVRs}$

{move **if** to top-level then use **tuple-shadow-deref**}

$= \textbf{if } i = i_c \textbf{ then } m \in \mathit{sched}(i_c).\text{trip-MVRs} \cup \{m_c\}$
$\textbf{else } m \in \mathit{sched}(i).\text{trip-MVRs}$

{**negation-assumption** and **if-then-else-false**}

https://doi.org/10.1017/S0269888901000029 Published online by Cambridge University Press

$= (i = i_c \wedge m \in sched(i_c)\,.\,\text{trip-MVRs} \cup \{m_c\})$

{distribute $\in$ over $\cup$}

$= (i = i_c \wedge (m \in sched(i_c)\,.\,\text{trip-MVRs} \vee m \in \{m_c\}))$

{**negation-assumption** and simplification}

$= (i = i_c \wedge m = m_c)$.

The full residual constraint becomes, after simplification,

$$\text{release-time}(m_c) \leq sched'(i_c)\,.\,\text{earliest-trip-start}$$

The proceduralisation of this residual is

if $\neg\text{release-time}(m_c) \leq sched'(i_c)\,.\,\text{earliest-trip-start}$
then $sched(i_c)\,.\,\text{earliest-trip-start} \leftarrow \text{release-time}(m_c)$.

Thus we have a third incremental refinement given by the equation

$sched' = \text{seq-shadow}(sched, i_c,$ (**update-earliest-start**)
$\text{tuple-shadow}(sched(i_c), \text{earliest-trip-start}, tc))$

where the refinement has been abstracted by introducing the variable *tc* for release-time(m_c).

6.2 *Specialising* ***trip-separations-ets***

Now we consider specialising the **trip-separations-ets** constraint.

$\forall(i : Integer, t : Time)$
$(i \in \{1..size(sched) - 1\} \wedge t = sched(i)\,.\,\text{earliest-trip-start} + duration$
$\Rightarrow t \leq sched(i+1)\,.\,\text{earliest-trip-start})$.

It is only affected by **append-new-trip** and **update-earliest-start**.

Consider **append-new-trip**. The residual for the first conjunct, $i \in \{1..size(sched) - 1\}$, is $i = size(sched)$. The full residual constraint becomes, after simplification,

$sched(size(sched))\,.\,\text{earliest-trip-start} + duration$
$\leq sched'(size(sched) + 1)\,.\,\text{earliest-trip-start}$

The residual for the second conjunct, $t = sched(i)\,.\,\text{earliest-trip-start} + duration$, is $i = size(sched) + 1 \wedge t = \text{trip}\perp\,.\,\text{earliest-trip-start} + duration$, but the full residual constraint simplifies to *false* because the first conjunct becomes $i \in \{1..size(sched)\}$ which is inconsistent with $i = size(sched) + 1$.

The residual from the first conjunct has the same form as **update-earliest-start** so we can reuse the same refinement procedure, although passing different arguments.

Now we consider the effect of **update-earliest-start** on **trip-separations-ets**.

The residual for the first conjunct is *false* because the size of the schedule is unaffected. The residual for the second conjunct is $i = i_c \wedge t = t_c + duration$. The full residual constraint becomes, after simplification,

$i_c < size(sched)$
$\Rightarrow t_c + duration \leq sched'(i_c + 1)\,.\,\text{earliest-trip-start})$.

https://doi.org/10.1017/S0269888901000029 Published online by Cambridge University Press

Again, except for the conditional, this has the same form as **update-earliest-start** so the same procedure can be used. The complete refinement procedure for **update-earliest-start** is as follows:

function update-earliest-start(i_c, t_c, *sched*) =
  if $t_c \leq sched(i_c)$.earliest-trip-start
    then sched % *Old bound is tighter*
  else if $\neg t_c \leq sched(i_c)$.latest-trip-start
    then $\top$ % *Fail because earliest is after latest*
  else % *Perform update and propagate*
  let (*sched′* = seq-shadow(*sched*, i_c, tuple-shadow(*sched*(i_c), earliest-trip-start, t_c)))
  if $i_c < size(sched)$
    then update-earliest-start($i_c + 1$, $t_c +$ *duration*, *sched′*)
    else *sched′*.

The specialisation of **trip-separations-lts** is similar to that of **trip-separations-ets**.

6.3 Summary of generated propagation code

The basic search routine generates new subspaces by calls to **append-new-trip** and **update-trip-MVRs**. The former has a call to **update-earliest-start** which initialises the earliest-start of the new trip based on the earliest-start of the previous trip. **Update-trip-MVRs** has calls to **update-earliest-start** and **update-latest-start** based respectively on the release-time and due-time of the MVR being added. **Update-earliest-start** is a linear recursion that propagates a change to the earliest-start of a trip to earliest-starts of subsequent trips until one does not need to be updated because the separation is already adequate. **Update-latest-start** is similar, but propagates from the latest-start of a trip to latest-starts of previous trips until one does not need to be updated.

This propagation code is very efficient, performing very few unnecessary tests. It is also space-efficient as the constraints are represented procedurally and the solution spaces are represented intensionally by bounds. We have used a depth-first propagation control structure because the dependency structure for this problem is a tree. In general, a breadth-first control structure is necessary to avoid unnecessary work when the constraint interactions are more complicated.

7 Extracting a solution

For this problem the extraction process is straightforward. The only issue is extracting valid trip-starts given earliest- and latest-trip-starts. One cannot arbitrarily choose values from the intervals to get a valid schedule. For example, choosing the latest-trip-start from one trip and the earliest-trip-start from the next trip will likely violate the **trip-separation** constraint. Always choosing the earliest-trip-starts or always choosing the latest-trip-starts gives a valid schedule. A more flexible strategy is to choose a trip-start between the earliest- and latest-trip-starts for one trip, then call the procedures **update-earliest-start** and **update-latest-start** with the chosen value, so that the consequences of the choice are propagated. Then this choose-and-propagate process can be repeated for other trips until a trip-start has been chosen for each trip.

In general, it is possible that there is no valid solution to the original problem within a non-empty valid subspace (one that satisfies the transformed problem). If a single solution is required then one must enumerate valid subspaces until one is found that has at least one extractable solution. If all solutions are required then all valid subspaces must be enumerated. If a minimal cost solution is required then a similar enumeration is necessary except we can add the constraint that the cost of a solution has to be less than the cost of the current best solution.

https://doi.org/10.1017/S0269888901000029 Published online by Cambridge University Press

8 Related work

Mackworth (1992) gives a characterisation of constraint-satisfaction problems in relation to various logical representation and reasoning systems such as Horn first-order predicate calculus and constraint logic programs. Our system does not fit within his framework. Our work, along with that of Rehof and Mogensen, suggests the addition of an extra dimension to the framework in which the Boolean lattice is generalised to an arbitrary lattice.

Our problem reformulation of replacing trip-start by upper and lower bounds on trip-start is similar to work that uses interval techniques (Benhamou *et al.*, 1994; Hyvnen, 1994; Van Hentenryck *et al*., 1995). Our framework can be seen as a generalisation of these approaches to work with arbitrary kinds of bounds.

Our model of constraint propagation generalises the concepts of cutting planes in the operations research literature (Nemhauser & Wolsey, 1988) and the forms of propagation studied in the constraint satisfaction literature (e.g. Van Hentenryck, 1989). Our use of fixed-point iteration for constraint propagation is similar to Paige's work on fixed-point iteration in RAPTS (Cai & Paige, 1989). The main differences are (1) RAPTS expects the user to supply the monotone function as part of the initial specification whereas we derive it from a more abstract statement of the problem; (2) RAPTS instantiates a straightforward iteration scheme and then performs optimisations. Such an approach would be too inefficient for scheduling applications, so we use dependence analysis to generate code that is specific to the constraint system at hand. RAPTS uses finite differencing in order to make the iteration incremental. We have incorporated this into our framework and used it in more complex scheduling problems.

9 Discussion and further work

The main focus of our work has been on the synthesis of high-performance constraint-solving codes – some of the schedulers that we have synthesised using KIDS run several orders of magnitude faster than manually written schedulers for the same problem. We believe that the speed is due to the specialised representation of constraints and the ability to optimize the propagation codes at design time. We have used KIDS to synthesise a variety of scheduling applications including ITAS (a theatre airlift scheduler) (Burstein & Smith, 1996) and the CAMPS Mission Planner (Emerson & Burstein, 1999) which plans strategic airlift missions for the Air Mobility Command at Scott AFB. The speed of the generated scheduling algorithm has allowed us to tackle very complex constraint systems. For example, the CAMPS Mission Planner involves the routine scheduling of thousands of airlift missions and the simultaneous handling of many classes of resource: aircraft and their configurations, crews and their duty days, fuel, parking capacity at ports, working and throughput capacity at ports, runway events and others. Aircraft, crews and ports each have many capacity and usage constraints that must be modelled. Every time a scheduling decision is made, it is propagated through the constraint network to decide if it entails any inconsistency. The Mission Planner is scheduled to begin operations in 2001.

We have described our work as a generalisation of various frameworks. These frameworks have been developed in more depth than ours so there are many opportunities to see how ideas developed in these frameworks can be carried over. Our focus has been mainly on scheduling algorithms. We have looked briefly at problems such as integer linear programming, but not sufficiently to make a good estimate of their potential compared to other methods. Working from such a model is unlikely to give the best results as these models capture information in a very limited lattice structure. Frequently, problems must be reformulated to get them into the form required by these general methods. Working from the original problem we may be able to capture more of its structure in lattices and so get a smaller search space. On the other hand, it may be possible to recover lattice structure information from analysis of an integer linear programming problem specification.

https://doi.org/10.1017/S0269888901000029 Published online by Cambridge University Press

Our framework is not directly applicable to incremental rescheduling if constraints are deleted as well as added. A possible way to handle the removal of a constraint is to follow dependencies of the deleted constraints to see which values may have depended on them. These values can be relaxed so that the space is large enough to find a solution when the new constraints are added.

We are interested in finding problems that have more lattice structure that our approach can exploit. Graphplan (Blum & Furst, 1997) has aroused much interest in the planning community because of its speed and its new approach to planning. Kambhampati, Lambrecht and Parker (1997) give an analysis of Graphplan, describing it as using a disjunctive representation of plans for which there is an effective global search procedure along with a non-trivial extraction phase. It would be illuminating if a derivation were possible along the lines given in this paper (see, for example, Srivastiva & Kambhampati, 1998). It would also be useful to examine other programs that solve an abstracted version of a problem before extracting concrete solutions.

Acknowledgements

This research was supported in part by DARPA and Air Force Rome Laboratories under Contracts F30602-91-C-0043, F30602-95-C-0247, and F30602-97-C-0154.

References

Applegate, D and Cook, W, 1991, "A computational study of the job-shop scheduling problem" *ORSA Journal on Computing* **3**(2) 149–156.

Benhamou, F, McAllester, D and Van Hentenryck, P, 1994, "Clp(intervals) revisited" Technical Report CS-94-18, Department of Computer Science, Brown University, April 1994.

Blum, A and Furst, M "Fast planning through planning graph analysis" *Artificial Intelligence* **90**(1) 281–300.

Burstein, MH and Smith, D, 1996, "ITAS: a portable interactive transportation scheduling tool using a search engine generated from formal specifications" *Proceedings of the Third International Conference on Artificial Intelligence Planning System* (AIPS-96) 35–44.

Cai, J and Paige, R, 1989, "Program derivation by fixed point computation" *Science of Computer Programming* **11** 197–261.

Emerson, T and Burstein, M, 1999, "Development of a constraint-based airlift scheduler by program synthesis from formal specifications" *Proceedings of the Fourteenth Automated Software Engineering Conference* 267–270.

Fox, MS, Sadeh, N and Baykan, C, 1989, "Constrained heuristic search" *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* 309–315.

Fox, MS and Smith, SF, 1984, "ISIS – a knowledge-based system for factory scheduling" *Expert Systems* **1**(1) 25–49.

Hyvönen, E, 1992, "Constraint reasoning based on interval arithmetic: the tolerance propagation approach *Artificial Intelligence* **58** 71–112; also in: E Freuder and A Mackworth (eds) *Constraint-Based Reasoning* MIT Press.

Kambhampati, S, Lambrecht, E and Parker, E, 1997, "Understanding and extending graphplan" *Proceedings of the 4th European Conference on Planning* 260–272.

Luenberger, DG, 1989, *Linear and Nonlinear Programming* Addison-Wesley Publishing Company, Inc.

Mackworth, AK, 1992, "The logic of constraint satisfaction" *Artificial Intelligence* **58** 3–20.

Nemhauser, GL and Wolsey, LA, 1988, *Integer and Combinatorial Optimization* John Wiley & Sons, Inc.

Rehof, J and Mogenson, T, 1999, "Tractable constraints in finite semi-lattices" *Science of Computer Programming* **35** 191–221.

Smith, DR, 1987, "Structure and design of global search algorithms" Technical Report KES.U.87.12, Kestrel Institute.

Smith, DR, Parra, EA and Westfold, SJ, 1996, "Synthesis of planning and scheduling software" in A Tate (ed.) *Advanced Planning Technology* AAAI Press.

Smith, DR and Westfold, SJ, 1995, "Synthesis of constraint algorithms" in V Saraswat and P Van Hentenryck (eds) *Principles and Practice of Constraint Programming* The MIT Press.

Smith, SF, 1989, "The OPIS framework for modeling manufacturing systems" Technical Report CMU-RI-TR-89-30, The Robotics Institute, Carnegie-Mellon University.

https://doi.org/10.1017/S0269888901000029 Published online by Cambridge University Press

Smith, SF, Fox, MS and Ow, PS, 1986, "Constructing and maintaining detailed production plans: investigations into the development of knowledge-based factory scheduling systems" *AI Magazine* **7**(4) 45–61.

Srivastiva, B and Kambhampati, S, 1998, "Synthesizing customizable planners from specifications" *Journal of A.I. Research* **8** 93–128; `http://www.cs.washington.edu/research/jair/volume8/srivastava98a.ps`.

Van Hentenryck, P, 1989, *Constraint Satisfaction in Logic Programming* Massachusetts Institute of Technology.

Van Hentenryck, P, McAllester, D and Kapur, D, 1995, "Solving polynomial systems using a branch and prune approach" *SIAM Journal of Numerical Analysis* 797–827.

https://doi.org/10.1017/S0269888901000029 Published online by Cambridge University Press