Job shop scheduling by constraint satisfication

Eindhoven University of Technology

Department of Mathematics and Computing Science

Job Shop Scheduling by Constraint Satisfaction

by

W.P.M. Nuijten, E.H.L. Aarts, D.A.A. van Erp Taalman Kip
and
K.M. van Hee 93/39

Computing Science Note 93/39
Eindhoven, November 1993

Copies can be ordered from:
Mrs. M. Philips
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

Job Shop Scheduling by Constraint Satisfaction

W.P.M. Nuijten [1] E.H.L. Aarts [1,2] D.A.A. van Erp Taalman Kip [1,3] K.M. van Hee [1]

[1] Eindhoven University of Technology, Department of Mathematics and Computing Science, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
[2] Philips Research Laboratories, P.O. Box 80000, 5600 JA Eindhoven, The Netherlands
[3] RIKS, P.O. Box 463, 6200 AL Maastricht, The Netherlands

Abstract

We present a new algorithm for the job shop scheduling problem based on constraint satisfaction techniques. Novel features of our algorithm are its extensive consistency checking and randomized variable and value selection. An empirical performance analysis is presented in which our algorithm is compared with the best-known approximation algorithms for job shop scheduling. We argue that our algorithm performs comparable to these algorithms.

1 Introduction

We are concerned with a problem in deterministic machine scheduling known as the Job Shop Scheduling Problem (JSSP) [French, 1982]. Informally, the problem can be stated as follows. Given are a set of jobs and a set of machines. Each job consists of a set of operations that must be processed in a given order. Furthermore, each operation is given an integer processing time and a machine by which it has to be executed. Once an operation is started, it is executed without interruption and a machine can execute at most one operation at a time. A schedule assigns a start time to each operation. Basically, the problem is to find for each machine an ordering of the operations that have to be executed by it. In the optimization variant of the JSSP one is asked to find a schedule that minimizes the maximum completion time of the operations. Here, we follow the feasibility variant of the JSSP. To this end a fixed overall deadline is introduced and the problem is to find a schedule, if it exists, that meets this overall deadline. The reason to consider this variant is that it corresponds to the general formulation of constraint satisfaction problems [Montanari, 1974]. It should be noted that any algorithm that solves the feasibility variant can be used to solve the optimization variant with a binary search in a number of calls to this algorithm which is bounded by $\log(Np_{\max})$, where N denotes the number of operations and $p_{\max}$ the maximum processing time; see also [Garey & Johnson, 1979].

This paper is one in a series of papers in which we report on our investigations of the potentials of constraint satisfaction techniques for scheduling. The general objective of our research is to investigate solution methods that are capable of solving a wide range of scheduling problems, but that also perform satisfactory on special cases. Here, the method under consideration is constraint satisfaction and in this paper we concentrate on the JSSP as a special case. This choice is motivated by the fact that this is a well-investigated problem for which a number of high quality solution methods exist that are compared on the basis of a well-known benchmark set of problem instances.

The organization of the paper is as follows. In Section 2 we discuss constraint satisfaction after which, in Section 3, we give a formal definition of the JSSP. Section 4 presents our constraint satisfaction approach to the JSSP. Section 5 presents the computational results and a performance comparison with other algorithms. Section 6 concludes the paper with a discussion of the obtained results.

2 Constraint satisfaction

2.1 The constraint satisfaction problem

An instance of the Constraint Satisfaction Problem (CSP) [Montanari, 1974] involves a set of variables, a domain for each variable specifying the values to which it may be assigned, and a set of constraints on the variables. The constraints define which combinations of domain values are allowed and which are not. One is asked to assign values to variables such that all constraints simultaneously are satisfied. The obvious difference with an optimization problem is the absence of an optimization criterion; the problem is to find a feasible solution instead of a best solution. Modeling the instances of a problem as instances of the CSP is a rather general and flexible technique and a large class of problems can be treated as such. Before giving a definition of the CSP, we formally define *domains*, *constraints* and *assignments*.

Definition 2.1. A *domain* is a finite set, and its elements are called *values*. □

The size of a domain D is $q|D|$ where q is the maximum number of bits needed to encode any value in the domain.

Definition 2.2. Given is a collection of domains $D = \{D_1, \ldots, D_p\}$. A *constraint* $c : D_1 \times \ldots \times D_p \rightarrow \{\top, \bot\}$ on D defines which combinations of values from the domains satisfy the constraint. The set of all constraints on D is denoted by $\mathcal{C}(D)$. □

A constraint c on $D = \{D_1, \ldots, D_p\}$ is called *polynomially computable*, if for all $s \in D_1 \times \ldots \times D_p$, $c(s)$ can be computed within a time bounded by a polynomial in the sum of the sizes of the domains. We consider only polynomially computable constraints. Below, we treat the constraints of Definition 2.2 as predicates, e.g., $c(s)$ is used instead of $c(s) = \top$ and $\neg c(s)$ is used instead of $c(s) = \bot$.

Definition 2.3. Given are a set of variables $X = \{x_1, \ldots, x_p\}$, and for each variable x_i a domain $D(x_i)$ specifying the values to which it may be assigned. An *assignment* $s \in D(x_1) \times \ldots \times D(x_p)$ specifies for each $x_i \in X$ a value in its domain, i.e., $s(x_i)$ denotes the value of variable x_i. The set of all assignments is denoted by $\mathcal{A}(X, D)$. □

Definition 2.4. An instance of the *Constraint Satisfaction Problem* (CSP) is a triple $\langle X, D, C \rangle$, where $X = \{x_1, \ldots, x_p\}$ is a set of variables, $D = \{D(x_1), \ldots, D(x_p)\}$, gives for each variable x_i a domain $D(x_i)$, for which the size is polynomial in p, and $C = \{c_1, \ldots, c_q\} \subseteq \mathcal{C}(D)$ is a set of polynomially computable constraints on D, such that q is polynomial in p. The question is whether there is an assignment $s \in D(x_1) \times \ldots \times D(x_p)$ such that $\forall_{1 \leq i \leq q}\, c_i(s)$. Such an assignment is called a *solution*. □

The size of an instance of the CSP is characterized by the number of variables, i.e., each instance of the CSP can be coded in size bounded by a polynomial in p, where p is the number of variables. This follows from the sizes of the domains being polynomial in p, the number of constraints being polynomial in p, and each constraint being polynomially computable, which implies that there exists an encoding that is polynomial in the sum of the sizes of the domains, which in turn is polynomial in p.

Theorem 2.1. *The CSP is NP-complete.*

Proof. The CSP is in NP, as for a given assignment s all constraints can be checked in polynomial time. A straightforward polynomial time transformation from SATISFIABILITY [Garey & Johnson, 1979] concludes the proof. $\square$

The constraints of Definition 2.2 may involve all variables. However, one often only considers *unary* and *binary* constraints, which are concerned with one and two variables, respectively. For simplicity reasons, we assume that at most one unary constraint for each variable x is given, denoted by c_x, and that at most one binary constraint relates variable x to variable x', denoted by $c_{xx'}$. c_x can be represented as a Boolean function over $D(x)$, i.e., $c_x : D(x) \to \{\top, \bot\}$. $c_{xx'}$ can be represented by a Boolean function over the Cartesian product of $D(x)$ and $D(x')$, i.e., $c_{xx'} : D(x) \times D(x') \to \{\top, \bot\}$. A unary constraint c_x can be viewed as a binary constraint c_{xx} such that, for $v \in D(x)$, $c_x(v) \Leftrightarrow c_{xx}(v, v)$. Therefore, often only binary constraints are considered. Nudel [1983] proofs that this can be done without loss of generality. An instance of the CSP with only binary constraints is called a *binary* CSP instance. We, furthermore, assume that binary constraints are *symmetric*, i.e., $c_{xx'}(v, v') = c_{x'x}(v', v)$ for $v \in D(x)$ and $v' \in D(x')$. With a binary CSP instance, a *constraint graph* G can be associated, with a node set given by the set of variables, and directed arcs (x, x') for each constraint $c_{xx'}$. In [Dechter & Pearl, 1988], some classes of binary CSP instances are identified that can be solved in polynomial time. One of these classes is the class of instances in which the constraint graph is a tree. Such instances can be solved in $O(pd^2)$, where p is the number of variables and d is the size of the largest domain. For a more elaborate discussion on well-solvable special cases we refer to [Dechter & Pearl, 1988].

2.2 Solving the CSP

The question of solving the CSP has received much attention, especially in the field of artificial intelligence [Nadel, 1989]. We divide the approaches into two classes, viz., *domain independent* and *domain dependent*. Domain independent approaches do not use specific conditions on the variables, their values or the corresponding constraints. As is often done in artificial intelligence, the emphasis is put on common properties of problems. The performance of such general approaches for specific subclasses of problems, however, is often questionable. Therefore, an increasing number of constraint satisfaction researchers employ a domain dependent approach and try to improve performance by exploiting the structure of the problem. Such an approach is often said to be typical of operations research. We use techniques that find their origins in both artificial intelligence and operations research and as a result, we are able to combine the best of both worlds in the sense that the resulting approach is general but still capable of exploiting problem specific structure, leading to improved performance characteristics.

Most approaches to solve the CSP are based on tree search algorithms. In order to treat these algorithms properly, we need to introduce some definitions.

Definition 2.5. A *search state* of an instance $I = \langle X, D, C\rangle$ of the CSP is a pair $\langle \Pi, \delta\rangle$, where $\Pi \subseteq C(D)$ is a set of *posted constraints* and for each variable $x \in X$, $\delta(x) \subseteq D(x)$ gives the *current domain*. $\square$

Definition 2.6. A *search tree* of an instance I of the CSP is a minimal connected directed acyclic graph $\langle V, E\rangle$, where V is a set of search states of I. $\square$

In a search state for each variable the current domain is administrated, being the values of each variable that are still under consideration. Furthermore, a search state records which constraints are posted, thus recording which decisions led to this search state. The node set of a search tree consists of a set of search states. Going from one node to another is done by posting an extra constraint. At any node in the search tree a limited number of constraints may be posted, defining the edges of the tree. Initially, i.e., in the root of a search tree, the search state always is $\langle \emptyset, D \rangle$, representing that no constraints are posted and the current domains are equal to $D(x)$ for all variables $x \in X$.

A search state represents a set of solutions that satisfy both the original constraints and the posted constraints and that, for each variable $x \in X$, have a value from $\delta(x)$.

Definition 2.7. Let $I = \langle X, D, C \rangle$ be an instance of the CSP, and let $\sigma = \langle \Pi, \delta \rangle$ be a search state of I. The *solution set* of σ is the set $\mathcal{S}(\sigma) = \{s \in \mathcal{A}(X, D) \mid \forall_{c \in C}\, c(s) \wedge \forall_{c \in \Pi}\, c(s)\}$. □

Observe that all assignments in the solution set of any search state of an instance I, also are solutions of I.

Each time a constraint is posted, some values of variables may become *inconsistent*. Let $\sigma = \langle \Pi, \delta \rangle$ be a search state. A value $v \in \delta(x)$ for a variable x is called inconsistent if no solution in $\mathcal{S}(\sigma)$ exists that includes the assignment of v to x. Then, this value can be removed from the current domain of x, without losing any solutions. More formally a search state $\sigma' = \langle \Pi, \delta' \rangle$, where $\delta'(x) = \delta(x) \backslash \{v\}$ and for all $x' \neq x$, $\delta(x') = \delta'(x')$, can be obtained such that $\mathcal{S}(\sigma) = \mathcal{S}(\sigma')$. The resulting search state σ' is called *solution equivalent* with σ. The following definition formally defines this property.

Definition 2.8. Let $I = \langle X, D, C \rangle$ be an instance of the CSP. A *search state* $\sigma = \langle \Pi, \delta \rangle$ of I is called *solution equivalent* with a search state $\sigma' = \langle \Pi, \delta' \rangle$ if for all $x \in X$, $\delta(x) \subseteq \delta'(x)$ and $\mathcal{S}(\sigma) = \mathcal{S}(\sigma')$. □

The process of removing inconsistent values is usually called *consistency checking.* Consistency checking transforms one search state into a solution equivalent one. It is highly unlikely that all inconsistent values can be removed in polynomial time, as this would imply that starting with a search state $\sigma = \langle \Pi, \delta \rangle$, we could obtain in polynomial time a solution equivalent search state $\sigma' = \langle \Pi, \delta' \rangle$ such that for all $x \in X$ and for all $v \in \delta(x)$, there exists an $s \in \mathcal{S}(\sigma')$ such that $s(x) = v$. By iteratively choosing any value of the current domain of any variable and then removing all inconsistent values, we would have an $O(pZ)$ algorithm solving the CSP, where Z is the time complexity of the algorithm removing all inconsistent values. If Z is polynomial, so is this solution method, which contradicts with the CSP being NP-complete and the assumption that $\mathrm{P} \neq \mathrm{NP}$.

Once a search state is found such that $|\delta(x)| = 1$, for all $x \in X$, a solution is found and the instance is solved. If a search state is reached where the current domain of any variable is empty, i.e., there exists an $x \in X$ such that $\delta(x) = \emptyset$, we say that a *dead end* occurs.

Definition 2.9. Let $I = \langle X, D, C \rangle$ be an instance of the CSP. A search state $\sigma = \langle \Pi, \delta \rangle$ is called a *dead end* if there exists an $x \in X$ such that $\delta(x) = \emptyset$. □

If a dead end is reached it is proven that no solution exists that satisfies all the original constraints together with the posted constraints. Then the tree search algorithm has to *backtrack*, i.e., undo certain decisions and try alternatives for them. The search typically stops if a solution is found, or if all alternative decisions in the root of the tree have been tried without success. In the latter case, the instance is said to be *infeasible.*

We focus on algorithms where decisions are the selection of one value for a variable. Selecting a value $v \in \delta(x)$ for a variable x can be seen as posting a unary constraint c_x

6

such that $c_x(v') \Leftrightarrow v' = v$, for all $v' \in D(x)$. Of course only those variables $x \in X$ are regarded with $|\delta(x)| > 1$. In general, constraint satisfaction tree search algorithms that employ abovementioned strategy, can be described by the following framework.

```
while not solved and not infeasible do
    check consistency
    if a dead end is detected then
        backtrack
    else
        select variable
        select value for variable
    endif
endwhile
```

So, in each iteration, a variable is selected and assigned a value in its current domain. Subsequently, *consistency checking* is used to eliminate values that are inconsistent with the assignments made so far. Variable and value selection heuristics try to prevent the search from getting stuck in a dead end. If, however, the search does get stuck in a dead end, backtracking is needed to escape from it. These three basic components, viz., *consistency checking*, *variable and value selection* and *dead end handling*, are treated below in more detail.

Consistency checking. Each time a variable is assigned a value, *inconsistent* values of the unassigned variables are removed. A value $v \in \delta(x)$ for a variable x is called inconsistent if no solution exists that includes the assignment of v to x in addition to the assignments made so far. Important forms of consistency are *node*, *arc* and *path consistency*, also called 1-, 2-, and 3-consistency, respectively. Mackworth [1977] gives a description of all three forms of consistency. We only address node and arc consistency.

Node consistency refers to the consistency of unary constraints. A constraint c_x is node consistent if $c_x(v)$ holds for all values $v \in \delta(x)$. Node consistency is easily achieved by deleting all values that do not satisfy the unary constraints. It suffices to do this only at the beginning of the search. Using unary constraints can therefore be eliminated by including appropriate redefinitions of the domains, i.e., by introducing new domains $D'(x) = \{v \in D(x) \mid c_x(v)\}$.

Arc consistency refers to the consistency of binary constraints. The idea of arc consistency is the following. If a value v of a variable x is inconsistent with all values of variable x', i.e., $\forall_{v' \in \delta(x')} \neg c_{xx'}(v, v')$, then no solution exists that includes the assignment of v to x in addition to the assignments so far, and thus v can be eliminated from $\delta(x)$. If, however, for all values $v \in \delta(x)$, at least one value in $\delta(x')$ is consistent with it, $c_{xx'}$ is called arc consistent.

Definition 2.10. Let x and x' be variables with current domains $\delta(x)$ and $\delta(x')$ respectively. Furthermore, let $c_{xx'}$ be the constraint defined on x and x'. $c_{xx'}$ is said to be *arc consistent* with respect to $\delta(x)$ and $\delta(x')$ if and only if

$$\forall_{v \in \delta(x)} \exists_{v' \in \delta(x')} c_{xx'}(v, v').$$

□

If all constraints of a binary CSP are arc consistent, we say that *full* arc consistency is achieved. Arc consistency algorithms have a long history; they originate from the Waltz filtering algorithm [Waltz, 1972] and have been refined several times [Mackworth, 1977], resulting in the time optimal algorithm AC-4 of Mohr & Henderson [1986]. AC-4 runs in $O(ed^2)$, where e is the number of binary constraints and d is the size of the largest domain. Van Hentenryck, Deville & Teng [1992] introduce AC-5 which generalizes both AC-3 [Mackworth, 1977] and AC-4. Moreover, AC-5 can be instantiated to an $O(ed)$ algorithm achieving arc consistency for functional, anti-functional and monotonic constraints. Mackworth & Freuder [1993] give an overview of the history of arc consistency algorithms.

Variable and value selection. The selection of a next variable and its value is done by *variable* and *value selection heuristics*, respectively. In its simplest form both the variables and their values are chosen in a predefined order. More sophisticated techniques dynamically try to select a variable which maximally constrains the rest of the variables. The idea of these variable selection heuristics is that one starts with *critical variables*, i.e., variables that are difficult to instantiate. A critical variable is one that is expected to cause backtracking, namely one whose remaining possible values are expected to conflict with the remaining possible values of other variables. Furthermore, so-called *least constraining* value selection heuristics are used that select a value that leaves as many options as possible open to the remaining uninstantiated variables. A least constraining value is one that is expected to participate in many solutions to the CSP. Naturally, both application domain dependent and independent variable and value selection heuristics have been developed. In Section 3 some variable and value selection heuristics in the field of scheduling are given. We mention two examples of domain independent variable selection heuristics, viz., selecting the variable with smallest number of remaining values; see [Bitner & Reingold, 1975; Haralick & Elliott, 1980; Purdom, 1983], and selecting the variable with maximal degree in the constraint graph [Freuder, 1982]. A domain independent value selection heuristic is Advised BackTracking (ABT) [Dechter & Pearl, 1988] that estimates for each value of the selected variable the number of solutions it participates in by counting the solutions to an instance that corresponds to a tree-like relaxation of the original constraint graph.

Dead end handling. If a dead end is reached it is derived that no solution exists that satisfies all original constraints and all posted constraints. As a result at least one posted constraint should be undone and an alternative constraint should be tried. In our case of assigning values to variables, this means at least one assignment has to be replaced by another. In its most simple form undoing constraints is done by means of *chronological backtracking*, which consists of undoing the last constraint and posting another constraint. Chronological backtracking, however, often results in exhaustively searching a subtree in which no solution can be found. This is because often the combination of several earlier constraints causes the algorithm to get stuck in such a subtree. The main problem of escaping from a dead end, is to decide which posted constraints to undo. Besides chronological backtracking, more sophisticated procedures, so called *intelligent backtracking* procedures, are developed to escape from a dead end. As examples of intelligent backtracking procedures, we mention *backjumping* and *dependency directed backtracking.* Like chronological backtracking, backjumping [Nadel, 1989] undoes the posted constraints in the reversed order they were posted, but in addition to the last posted constraint it undoes those constraints that did not contribute to the dead end, up to the last posted constraint that did contribute to the dead end. This improves the performance of the tree search. Another way of improving the performance is by recording the reasons for the dead end in the form of new constraints, so that the same conflicts will not arise again in the later search. *Dependency directed backtracking* [Stallman & Sussman, 1977] incorporates both backjumping and constraint recording. We refer to [Dechter & Pearl, 1988] for a short overview of intelligent backtracking procedures.

Besides these tree search approaches, some other solution methods have been proposed for the CSP. We mention the method of Minton, Johnston, Philips & Laird [1992] which is based upon the work of Adorf & Johnston [1990]. Instead of constructing a solution by extending consistent partial assignments as is done by the tree search algorithms, Minton, Johnston, Philips & Laird [1992] propose to assign a tentative value to every variable, resulting in a possible inconsistent full assignment, and to modify this assignment by choosing a variable with a conflicting assignment and replacing it with an assignment that minimizes the number of remaining conflicts, until all conflicts are resolved. The basis of this approach is reformulating the CSP as an optimization problem, and then solving it by a local search procedure. It this way, all local search procedures may be used to solve the problem. Little effort has been done to do so however.

3 The job shop scheduling problem

First, we present the feasibility variant of the JSSP and then give the straightforward relation with the CSP, i.e., we show that the JSSP can be seen as a special case of the CSP.

Definition 3.1. An instance of the *Job Shop Scheduling Problem* (JSSP) consists of a set $\mathcal{O}$ of N operations, a set $\mathcal{J}$ of n jobs, a set $\mathcal{M}$ of m machines, an overall deadline $D \in \mathbf{Z}^+$, a function $J : \mathcal{O} \to \mathcal{J}$ giving for each operation the job to which it belongs, a function $M : \mathcal{O} \to \mathcal{M}$ giving for each operation the machine on which it must be processed, a function $p : \mathcal{O} \to \mathbf{Z}^+$ giving the processing time of each operation, and a binary relation $\prec$ decomposing $\mathcal{O}$ into chains, such that every chain corresponds to a job. A *schedule* is a function $s : \mathcal{O} \to \mathbf{Z}_0^+$ giving the start times of the operations. The problem is to find a schedule s such that for all $o, o' \in \mathcal{O}$:

(i) $s(o) \geq 0$,

(ii) $s(o) + p(o) \leq D$,

(iii) $s(o) + p(o) \leq s(o')$, if $o \prec o'$, and

(iv) $s(o) + p(o) \leq s(o')$ or $s(o') + p(o') \leq s(o)$, if $M(o) = M(o')$ and $o \neq o'$.

□

Constraint (i) of Definition 3.1 implies nonnegative start times, constraint (ii) implies that all operations must be completed before D, constraint (iii) accounts for the binary relation, i.e., if $o \prec o'$, then o' cannot start before o is finished, and constraint (iv) implies that no two operations can be processed on the same machine at the same time. Constraints defined on operations that are related by $\prec$ are called *precedence constraints.* Constraints defined on operations that must be scheduled on the same machine are called *capacity constraints.*

That the JSSP is a special case of the CSP can easily be seen as follows. For each operation $o \in \mathcal{O}$, a variable is introduced, i.e., $\mathcal{O}$ is used as the set of variables. Furthermore, for each operation the domain $D(o) = [0, D - p(o) + 1]$ is defined, specifying the start times for it, and the set of binary constraints is defined as follows. Let $o, o' \in \mathcal{O}$,

$o \neq o'$, $t \in D(o)$, and $t' \in D(o')$, then

$$c_{oo'}(t,t') \Leftrightarrow \begin{cases} t + p(o) \leq t' & \text{if } o \prec o', \\ t' + p(o') \leq t & \text{if } o' \prec o, \\ t + p(o) \leq t' \ \vee \ t' + p(o') \leq t & \text{if } M(o) = M(o'),\ o \not\prec o', \text{ and } o' \not\prec o, \end{cases}$$

Note that a binary constraint defined on the variables of operations o and o' is denoted as $c_{oo'}$. As for each operation a variable is introduced, we often use "variable" and "operation" interchangeably. The same holds for "value" and "start time", as values correspond to start times.

The decision variant of the JSSP has been proved to be NP-complete [Garey, Johnson & Sethi, 1976]. Over the years many algorithms have been designed to handle the optimization variant of the JSSP. Roughly speaking, these algorithms can be divided into two classes, viz. optimization and approximation algorithms. Most optimization algorithms for the JSSP are based on branch & bound techniques; see for instance Carlier & Pinson [1989], Applegate & Cook [1991], and Brucker, Jurisch & Sievers [1992]. Among the most successful approximation algorithms we mention the shifting bottleneck procedure of Adams, Balas & Zawack [1988], simulated annealing [Van Laarhoven, Aarts & Lenstra, 1992; Matsuo, Suh & Sullivan, 1988], genetic local search [Aarts, Van Laarhoven, Lenstra & Ulder, 1992; Dorndorf & Pesch, 1992] and tabu search [Dell'Amico & Trubian, 1993].

Most attention to the feasibility variant of the JSSP is paid by the field of constraint satisfaction. Most constraint satisfaction algorithms for scheduling are based on tree search algorithms, which construct a solution of a given problem instance by assigning the start times to the operations one by one; we only mention [Keng & Yun, 1989] and [Sadeh, 1991]. Recently, Muscettola [1993] and Smith & Cheng [1993] have used a different formulation of the JSSP as the one we presented. They follow the well-known observation that the actual problem of the JSSP is determining machine orderings, i.e., orderings of the operations on the machines. Novel features are however the variable and value ordering heuristics, defining which operations to order and how. Minton, Johnston, Philips & Laird [1992], see Section 2.2, also test their so-called repair heuristic on scheduling.

Although several applications of constraint satisfaction techniques to scheduling have been reported in the literature, few comparative studies are known. One example of such a study is given by Sadeh [1991] who compares his variable and value orderings with those of Keng & Yun [1989], and with the general variable ordering of Purdom [1983] and the general value ordering of Dechter & Pearl [1988]. Sadeh has carried out his performance analysis on a specific set of randomly generated instances and he concludes that his orderings perform better than the other approaches used in the comparison. We implemented Sadeh's variable and value orderings and tested them using the well-known benchmark set of problem instances of [Lawrence, 1984] and [Fisher & Thompson, 1963]. We found that Sadeh's algorithm performs poorly on these instances; see [Nuijten, Aarts, Van Erp Taalman Kip & Van Hee, 1993]. In their turn Muscettola [1993] and Smith & Cheng [1993] compare their approaches to the one of Sadeh and come up with slightly better results. Muscettola [1993] also implemented the approach of [Minton, Johnston, Philips & Laird, 1992], and he reports quite poor results, even on the relatively easy instances from [Sadeh, 1991].

4 A new constraint satisfaction approach

In this section our approach is presented which is based on tree search algorithms. In Section 4.1 the consistency checking algorithms we use are discussed. Section 4.2 presents the way operations and start times are selected. Section 4.3 discusses the way dead ends are handled.

4.1 Consistency checking

Arc consistency. In Definition 2.10 arc consistency is defined for the general binary CSP. The straightforward translation to the JSSP leads to the following. Let $o, o' \in \mathcal{O}$, then constraint $c_{oo'}$ is arc consistent if and only if for all $t \in \delta(o)$, there exists a $t' \in \delta(o')$ such that $c_{oo'}(t, t')$, where $\delta(o)$ and $\delta(o')$ are the current domains of o and o', respectively. Before discussing arc consistency of the constraints of the JSSP in detail, we introduce the following notations. For each operation $o \in \mathcal{O}$ we have

- $\delta(o)$: the current domain,
- $\text{est}(o) = \min\{t \mid t \in \delta(o)\}$: the earliest possible start time,
- $\text{ect}(o) = \min\{t + p(o) \mid t \in \delta(o)\}$: the earliest possible completion time,
- $\text{lst}(o) = \max\{t \mid t \in \delta(o)\}$: the latest possible start time,
- $\text{lct}(o) = \max\{t + p(o) \mid t \in \delta(o)\}$: the latest possible completion time.

Let $o, o' \in \mathcal{O}$ and $o \prec o'$, then $c_{oo'}(t, t') \Leftrightarrow t + p(o) \leq t'$; see Section 3. The following theorem is concerned with arc consistency of precedence constraints.

Theorem 4.1. *Let $o, o' \in \mathcal{O}$ and $o \prec o'$. Then (i) $c_{oo'}$ is arc consistent with respect to $\delta(o)$ and $\delta(o')$, if and only if* $\text{lct}(o) \leq \text{lst}(o')$ *and (ii) $c_{o'o}$ is arc consistent with respect to $\delta(o)$ and $\delta(o')$, if and only if* $\text{ect}(o) \leq \text{est}(o')$.

Proof. (i) If $\text{lct}(o) \leq \text{lst}(o')$, then for all $t \in \delta(o)$, $t + p(o) \leq \text{lst}(o')$. (ii) If $\text{ect}(o) \leq \text{est}(o')$, then for all $t' \in \delta(o')$, $\text{ect}(o) \leq t'$. □

Arc consistency of $c_{oo'}$ and $c_{o'o}$ easily can be achieved by deleting from $\delta(o)$ all start times larger than $\text{lst}(o') - p(o)$ and by deleting from $\delta(o')$ all start times smaller than $\text{ect}(o)$.

Let $o, o' \in \mathcal{O}, o \neq o', M(o) = M(o'), o \not\prec o'$, and $o' \not\prec o$, then $c_{oo'}(t, t') \Leftrightarrow t + p(o) \leq t' \lor t' + p(o') \leq t$; see Section 3. To illustrate how capacity constraints can be made arc consistent, we discuss the following example.

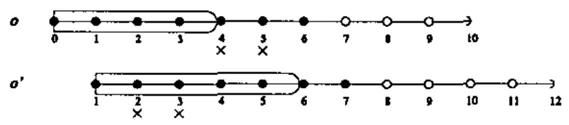

Figure 1: Two operations with $M(o) = M(o')$.

Let $\text{est}(o) = 0, \text{lst}(o) = 6, p(o) = 4$ and $\text{est}(o') = 1, \text{lst}(o') = 7, p(o') = 5$. In Figure 1, possible start times are denoted by a '•' and times at which an operation cannot be started although the operation may be processed at that time are denoted by a 'o'. Furthermore the processing time is indicated by a bar starting at the earliest start time. The '×'s indicate inconsistent start times. In this case these inconsistent start times have to be removed to achieve arc consistency of $c_{oo'}$ and $c_{o'o}$. As an example we state that o cannot be started at time 4, as this makes scheduling o' impossible. In general, an operation o cannot be started at any time in the open interval $(\text{lst}(o') - p(o), \text{ect}(o'))$ for any o' that is to be scheduled on the same machine. Therefore, $c_{oo'}$ is arc consistent if and only if $\delta(o) \cap (\text{lst}(o') - p(o), \text{ect}(o')) = \emptyset$, which leads to the following theorem.

Theorem 4.2. *Let $o, o' \in \mathcal{O}, o \neq o', M(o) = M(o')$, $o \not\prec o'$, and $o' \prec o$. Then $c_{oo'}$ is arc consistent with respect to $\delta(o)$ and $\delta(o')$, if and only if $\delta(o) \cap (\text{lst}(o') - p(o), \text{ect}(o')) = \emptyset$.*

Proof. If o is started at time $\text{lst}(o') - p(o) + 1$ or later, o' cannot be scheduled after o. Furthermore, if o is started at $\text{ect}(o') - 1$ or earlier, o' cannot be scheduled before o. From this follows that starting o at any time in the open interval $(\text{lst}(o') - p(o), \text{ect}(o'))$, which implies that o' cannot be scheduled before nor after o. □

Arc consistency of $c_{oo'}$ can be achieved by deleting the start times in $(\text{lst}(o') - p(o), \text{ect}(o'))$ from $\delta(o)$. Similarly, arc consistency of $c_{o'o}$ can be achieved by deleting the start times in $(\text{lst}(o) - p(o'), \text{ect}(o))$ from $\delta(o')$.

We now discuss the complexity of achieving arc consistency for a constraint $c_{oo'}$. In [Mackworth, 1977], the well-known procedure REVISE is introduced that achieves arc consistency for a general binary constraint $c_{xx'}$ on the variables x and x' in $O(|\delta(x)||\delta(x')|)$ running time, by deleting all values from $\delta(x)$ for which no value in $\delta(x')$ exists that satisfies $c_{xx'}$. REVISE is used in AC-3 which is an $O(ed^3)$ algorithm [Mackworth & Freuder, 1985] for achieving full arc consistency, where e is the number of binary constraints and d is the size of the largest domain. AC-3 can be used as a component of the consistency checking procedure mentioned in the framework of Section 2.2. When applying REVISE, it is needed that all values of the variables are administrated. However, we choose not to represent the consistent start times by specifying them individually, but by specifying the consistent intervals to which they belong, resulting in the representation of a domain $\delta(o)$ by a number of disjunctive intervals, i.e., $\delta(o) = [a_1, b_1] \cup \ldots \cup [a_x, b_x]$, with $a_1 \leq b_1 < a_2 \leq \ldots < a_x \leq b_x$. From Theorem 4.2 follows that achieving arc consistency for a constraint may lead to the deletion of at most one interval from one of the domains of the operations between which the constraint is defined. This may lead to the splitting of at most one interval $[a_i, b_i]$ into two new intervals. This implies that each domain can be represented by a list of at most $| \mathcal{O}_{M(o)} |$ intervals, where $\mathcal{O}_{M(o)}$ is the set of operations that are to be scheduled on machine $M(o)$. From this it follows that achieving arc consistency for $c_{oo'}$ can be done in $O(\mathcal{O}_{M(o)})$, both for precedence and capacity constraints. We thus can devise our own procedure achieving arc consistency, with $O(z)$ running time, where $z = \max_{o \in \mathcal{O}} |\mathcal{O}_{M(o)}|$. If we incorporate this procedure in AC-3, an $O(zde)$ full arc consistency algorithm results, referred to as AC-3a. AC-4 of Mohr & Henderson [1986] achieves full arc consistency in $O(ed^2)$ running time. AC-4 however also has an $O(ed^2)$ space complexity, which is $O(N^2d^2)$, whereas AC-3a has an $O(e + Nz)$ space complexity, which is $O(N^2)$. This leads to the following theorem.

Theorem 4.3. *Full arc consistency of a search state of the JSSP, can be obtained in $O(zde)$ running time and $O(e+Nz)$ space complexity, where e is the number of constraints, d is the size of the largest domain, and $z = \max_{o \in \mathcal{O}} |\mathcal{O}_{M(o)}|$.* □

Beyond arc consistency. In addition to achieving full arc consistency we use a technique due to Carlier & Pinson [1989], which determines whether an operation is to be scheduled before or after a specific subset of operations on the same machine. This technique can be used to reduce the domains of the unassigned operations, and thus can be used in a consistency algorithm. We illustrate the abovementioned technique by means of the following example.

Figure 2 shows three operations o, o', and o''. If o is scheduled on time 0, it is easy to see that it is impossible to schedule both o' and o''; the same holds for scheduling o on times $1 - 7$. What actually can be deduced is that o should be scheduled after the other two

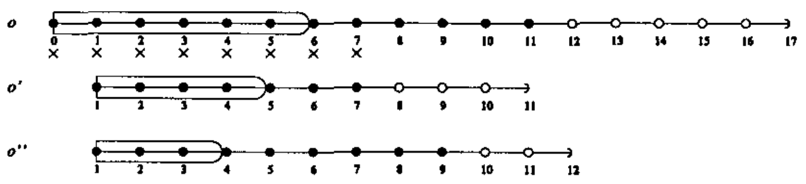
Figure 2: Three operations with $M(o) = M(o') = M(o'')$.

operations. Before continuing, we define the following abbreviations, for $S \subseteq \mathcal{O}$.

- $e(S) = \min_{o \in S} \text{est}(o)$
- $C(S) = \max_{o \in S} \text{lct}(o)$
- $P(S) = \sum_{o \in S} p(o)$

The following theorem provides the conditions under which one operation o is to be scheduled after a set S of operations on the same machine.

Theorem 4.4. *Let $o \in \mathcal{O}$ and $S \subset \mathcal{O}$, such that $o \notin S$ and $\forall_{o' \in S}\, M(o') = M(o)$. If*

$$e(S \cup \{o\}) + P(S \cup \{o\}) > C(S),$$

then no schedule exists in which o precedes any of the operations in S.

Proof. Suppose all operations are scheduled and o precedes an operation of S. If $o'' \in S$ is the operation that is scheduled last, then $s(o'') + p(o'') \geq e(S \cup \{o\}) + P(S \cup \{o\}) > C(S) \geq \text{lct}(o'')$ which leads to a contradiction. □

If no schedule exists in which an operation o precedes any of the operations in set S, est(o) can be set to $\max_{S' \subseteq S} e(S') + P(S')$. Formally, $\max_{S' \subseteq S} e(S') + P(S') = \infty$ as $\emptyset \subseteq S$ and $e(\emptyset) = \infty$ and $P(\emptyset) = 0$. We therefore, from now on, assume that $S' \neq \emptyset$. By considering all $S \subseteq \mathcal{O}_{M(o)}$, we come up with

$$L(o) = \max_{S \subseteq \mathcal{O}_{M(o)} | \alpha} \max_{S' \subseteq S} e(S') + P(S'),$$

$$\alpha \Leftrightarrow o \notin S \ \wedge \ e(S \cup \{o\}) + P(S \cup \{o\}) > C(S),$$

as a lower bound for the earliest start time of operation o. The following theorem provides the conditions under which one operation o is to be scheduled before a set S of operations on the same machine.

Theorem 4.5. *Let $o \in \mathcal{O}$ and $S \subset \mathcal{O}$, such that $o \notin S$ and $\forall_{o' \in S}\, M(o') = M(o)$. If*

$$C(S \cup \{o\}) - P(S \cup \{o\}) < e(S),$$

then no schedule exists in which o succeeds any of the operations in S.

Proof. Similar to the proof of Theorem 4.4 □

Consequently, if no schedule exists in which an operation o succeeds any of the operations in set S, lst(o) can be set to $\min_{S' \subseteq S} C(S') - P(S' \cup \{o\})$. If we do this for all sets $S \subseteq \mathcal{O}_{M(o)}$, we come up with

$$U(o) = \min_{S \subseteq \mathcal{O}_{M(o)} | \alpha} \min_{S' \subseteq S} C(S') - P(S' \cup \{o\}),$$

$$\alpha \Leftrightarrow o \notin S \ \wedge \ C(S \cup \{o\}) - P(S \cup \{o\}) < e(S),$$

as an upper bound for the latest start time of operation o.

Let $\mu \in \mathcal{M}$ be a machine and let T be the set of operations that are to be scheduled on μ. In [Carlier & Pinson, 1990] an $O(|T|^2)$ algorithm is given that for all $o \in T$ calculates $L(o)$

and $U(o)$. We derive a more concise $O(|T|^2)$ algorithm to do the same. We present the derivation of the algorithm that calculates all $L(o)$; a similar derivation of the algorithm that calculates all $U(o)$ is omitted.

In our algorithm we use the following two arrays:

- A contains all operations of T in ascending order of earliest start times.
- B contains all operations of T in ascending order of latest completion times.

These arrays can be constructed in $O(|T|\log|T|)$ time. A_x is used to denote the x-th element of array A; A_o^{-1} is used to denote the place of o in A, i.e., $A_o^{-1} = x$ if $A_x = o$.

For all A_x, with $1 \le x \le |T|$, we calculate

$$L(A_x) = \max_{S \subseteq T | \alpha} \max_{S' \subseteq S} e(S') + P(S'),$$

$$\alpha \Leftrightarrow A_x \notin S \ \wedge \ e(S \cup \{A_x\}) + P(S \cup \{A_x\}) > C(S).$$

We use L_x as a shorthand notation for $L(A_x)$. Lemma 4.1 is used to rewrite L_x.

Lemma 4.1. *Let $S \subseteq T$ and $o \in T$. If*

$$\forall_{U \subseteq T}\, e(U) + P(U) \le C(U), \tag{1}$$

$$e(S \cup \{o\}) + P(S \cup \{o\}) > C(S), \tag{2}$$

then

$$\mathrm{lct}(o) > C(S) \Leftrightarrow o \notin S.$$

Proof. Suppose $o \notin S \wedge \mathrm{lct}(o) \le C(S)$. Then with (2) $e(S \cup \{o\}) + P(S \cup \{o\}) > C(S \cup \{o\})$ holds and therefore with (1) a contradiction follows, hence $o \notin S \Rightarrow \mathrm{lct}(o) > C(S)$. Trivially $\mathrm{lct}(o) > C(S) \Rightarrow o \notin S$ holds. □

If (1) is not true a dead end is detected. Below, we describe an $O(|T|^2)$-algorithm to verify (1). Now we assume that (1) is true, so we may use Lemma 4.1 to rewrite L_x.

$$L_x = \max_{S \subseteq T | \alpha} \max_{S' \subseteq S} e(S') + P(S')$$

$$\alpha \Leftrightarrow \mathrm{lct}(A_x) > C(S) \ \wedge \ e(S \cup \{A_x\}) + P(S \cup \{A_x\}) > C(S)$$

In order to calculate L_x we iteratively consider for each $1 \le y \le |T|$ all subsets of T for which $C(S) = \mathrm{lct}(B_y)$. It is clear that by doing so we consider all subsets of T. Therefore, the following holds

$$L_x = \max_{1 \le y \le |T|} L_{x,y},$$

$$L_{x,y} = \max_{S \subseteq T | \alpha} \max_{S' \subseteq S} e(S') + P(S'),$$

$$\alpha \Leftrightarrow C(S) = \mathrm{lct}(B_y) \ \wedge \ \mathrm{lct}(A_x) > \mathrm{lct}(B_y) \ \wedge \ e(S \cup \{A_x\}) + P(S \cup \{A_x\}) > \mathrm{lct}(B_y).$$

However, for reasons displayed below we choose an equivalent definition of $L_{x,y}$.

$$L_{x,y} = \max_{S \subseteq T | \alpha} \max_{S' \subseteq S} e(S') + P(S')$$

$$\alpha \Leftrightarrow C(S) \le \mathrm{lct}(B_y) \ \wedge \ \mathrm{lct}(A_x) > \mathrm{lct}(B_y) \ \wedge \ e(S \cup \{A_x\}) + P(S \cup \{A_x\}) > \mathrm{lct}(B_y)$$

Remark that $\mathrm{lct}(B_y) = \mathrm{lct}(B_{y+1}) \Rightarrow L_{x,y} = L_{x,y+1}$.

From the previous definitions we obtain a preliminary version of the algorithm. Notice that we first calculate $L_{x,1}$ for all $1 \le x \le |T|$, than $L_{x,2}$ for all $1 \le x \le |T|$ etc., which is the reversed order of how we presented it.

```
for $y := 1$ to $|T|$ do
    if $y = |T| \lor \mathrm{lct}(B_y) \neq \mathrm{lct}(B_{y+1})$ then
        for $x := 1$ to $|T|$ do
            $\mathrm{est}(A_x) := \mathrm{est}(A_x)$ max $L_{x,y}$;
        endfor
    endif
endfor
```

Next, we must calculate the value of $L_{x,y}$. Remark that if $\mathrm{lct}(A_x) \leq \mathrm{lct}(B_y)$, $L_{x,y} = -\infty$, and thus no improvement of the earliest start time will result. In the sequel we, therefore, assume that $\mathrm{lct}(A_x) > \mathrm{lct}(B_y)$, which results in the redefinition of $L_{x,y}$ as follows.

$$L_{x,y} = \max_{S \subseteq T | \alpha} \max_{S' \subseteq S} e(S') + P(S')$$

$$\alpha \Leftrightarrow C(S) \leq \mathrm{lct}(B_y) \;\land\; e(S \cup \{A_x\}) + P(S \cup \{A_x\}) > \mathrm{lct}(B_y)$$

We now are going to treat subsets S of which at least one operation is placed before A_x in array A and subsets S of which all operations are placed after A_x in array A, separately. This leads to

$$\begin{aligned}
L_{x,y} &= E_{x,y} \text{ max } F_{x,y}, \\
E_{x,y} &= \max_{S \subseteq T | \varepsilon(S)} \max_{S' \subseteq S} e(S') + P(S'), \\
\varepsilon(S) &\Leftrightarrow C(S) \leq \mathrm{lct}(B_y) \;\land\; \forall_{o \in S}\, A_o^{-1} > x \;\land\; \mathrm{est}(A_x) + P(S) + p(A_x) > \mathrm{lct}(B_y), \\
F_{x,y} &= \max_{S \subseteq T | \varphi(S)} \max_{S' \subseteq S} e(S') + P(S'), \\
\varphi(S) &\Leftrightarrow C(S) \leq \mathrm{lct}(B_y) \;\land\; \exists_{o \in S}\, A_o^{-1} < x \;\land\; e(S) + P(S) + p(A_x) > \mathrm{lct}(B_y).
\end{aligned}$$

Both $\varepsilon(S)$ and $\varphi(S)$ are frequently used below.

We first concentrate on the calculation of $E_{x,y}$. The set

$$S_{x,y} = \{A_i \mid x \leq i \;\land\; \mathrm{lct}(A_i) \leq \mathrm{lct}(B_y)\},$$

contains those operations of T that are not placed before A_x in array A and that are to be scheduled before $\mathrm{lct}(B_y)$. Let $S \subseteq T$, such that $\varepsilon(S)$ holds. Then, trivially $S \subseteq S_{x,y} \;\land\; \varepsilon(S_{x,y})$. With

$$\max_{S' \subseteq S_{x,y}} e(S') + P(S') \geq \max_{S' \subseteq S} e(S') + P(S'),$$

this leads to the observation that

$$\neg\varepsilon(S_{x,y}) \Rightarrow \forall_{S \subseteq T}\, \neg\varepsilon(S),$$

which implies that $E_{x,y} = -\infty$, and

$$\varepsilon(S_{x,y}) \Rightarrow E_{x,y} = \max_{S' \subseteq S_{x,y}} e(S') + P(S').$$

With

$$G_{x,y} = \max_{S' \subseteq S_{x,y}} e(S') + P(S'),$$

this leads to

$$E_{x,y} = \begin{cases} G_{x,y} & \text{if } \mathrm{est}(A_x) + P(S_{x,y}) + p(A_x) > \mathrm{lct}(B_y) \\ -\infty & \text{otherwise.} \end{cases}$$

We use an array G to administrate $G_{i,y}$ for all $1 \leq i \leq |T|$. From

$$\begin{aligned}
G_{i,y} &= \max_{i' \in \{i,\ldots,|T|\} | \mathrm{lct}(A_{i'}) \leq \mathrm{lct}(B_y)} \mathrm{est}(A_{i'}) + P(S_{i',y}), \\
G_{|T|+1,y} &= -\infty,
\end{aligned}$$

$$
\begin{aligned}
G_{i,y} &= \begin{cases} G_{i+1,y} \ \max\ \mathrm{est}(A_i) + P(S_{i,y}) & \text{if } \mathrm{lct}(A_i) \le \mathrm{lct}(B_y) \\ G_{i+1,y} & \text{otherwise,} \end{cases} \\
P(S_{|T|+1,y}) &= 0, \\
P(S_{i,y}) &= \begin{cases} P(S_{i+1,y}) + p(A_i) & \text{if } \mathrm{lct}(A_i) \le \mathrm{lct}(B_y) \\ P(S_{i+1,y}) & \text{otherwise,} \end{cases}
\end{aligned}
$$

we derive the following part of the algorithm that calculates $G_{i,y}$ for all $1 \le i \le |T|$. Here, g is the variable containing $G_{i,y}$ and P is the variable containing $P(S_{i,y})$.

```
$P := 0$; $g := -\infty$
for $i := |T|$ down to 1 do
    if $\mathrm{lct}(A_i) \le \mathrm{lct}(B_y)$ then
        $P := P + p(A_i)$;
        $g := g$ max $\mathrm{est}(A_i) + P$;
    endif
    $G_i := g$;
endfor
```

Next, we turn to the calculation of $F_{x,y}$. Let $S \subseteq T$, such that $\varphi(S)$ holds. Furthermore, let $x' < x$ be such that $A_{x'} \in S \ \wedge\ \forall_{x''<x'} A_{x''} \notin S$. Then, trivially $S \subseteq S_{x',y} \ \wedge\ \varphi(S_{x',y})$. From

$$\max_{S' \subseteq S_{x',y}} e(S') + P(S') \ge \max_{S' \subseteq S} e(S') + P(S'),$$

by using the definition of $G_{x,y}$ and $e(S_{x',y}) = \mathrm{est}(A_{x'})$, we derive

$$F_{x,y} = \max_{x' \in \{1,\ldots,x-1\} | \mathrm{est}(A_{x'}) + P(S_{x',y}) + p(A_x) > \mathrm{lct}(B_y)} G_{x',y}.$$

We define

$$H_{x,y} = \max_{x' \in \{1,\ldots,x-1\} | \mathrm{lct}(A_{x'}) \le \mathrm{lct}(B_y)} \mathrm{est}(A_{x'}) + P(S_{x',y}).$$

Let $z \in \{1, \ldots, x-1\}$ be the index for which $\mathrm{est}(A_z) + P(S_{z,y})$ is maximal, i.e.,

$$\mathrm{lct}(A_z) \le \mathrm{lct}(B_y) \ \wedge\ \mathrm{est}(A_z) + P(S_{z,y}) = H_{x,y}.$$

If $H_{x,y} + p(A_x) \le \mathrm{lct}(B_y)$, then trivially $F_{x,y} = -\infty$. If $H_{x,y} + p(A_x) = \mathrm{est}(A_z) + P(S_{z,y}) + p(A_x) > \mathrm{lct}(B_y)$, then $\varphi(S_{z,y})$ holds, thus $F_{x,y} \ge G_{z,y}$. From Lemma 4.2 we then derive that

$$G_{z,y} = \max_{1 \le x' < x} G_{x',y},$$

which implies that $F_{x,y} = G_{z,y}$.

Lemma 4.2. *Let $1 \le x, y \le |T|$ and $z, z' < x$. Then*

$$\mathrm{est}(A_z) + P(S_{z,y}) \ge \mathrm{est}(A_{z'}) + P(S_{z',y}) \Rightarrow G_{z,y} \ge G_{z',y}.$$

Proof. If $z' \ge z$, trivial. If $z' < z$, then suppose $\mathrm{est}(A_z) + P(S_{z,y}) \ge \mathrm{est}(A_{z'}) + P(S_{z',y}) \ \wedge\ G_{z,y} < G_{z',y}$. Without loss of generality we assume that z' is the largest number smaller than z for which this holds. Then $G_{z',y} = \mathrm{est}(A_{z'}) + P(S_{z',y}) > G_{z,y}$, which together with $G_{z,y} \ge \mathrm{est}(A_z) + P(S_{z,y})$ leads to a contradiction. Hence, $\mathrm{est}(A_z) + P(S_{z,y}) \ge \mathrm{est}(A_{z'}) + P(S_{z',y}) \Rightarrow G_{z,y} \ge G_{z',y}$. □

From the definition of $G_{x,y}$, we derive that $G_{z,y} = G_{1,y}$, which leads to

$$F_{x,y} = \begin{cases} G_{1,y} & \text{if } H_{x,y} + p(A_x) > \mathrm{lct}(B_y) \\ -\infty & \text{otherwise.} \end{cases}$$

The values of $H_{x,y}$ are described by

$$H_{0,y} = -\infty,$$

$$H_{x,y} = \begin{cases} H_{x-1,y} \max \operatorname{est}(A_x) + P(S_{x,y}) & \text{if } \operatorname{lct}(A_x) \leq \operatorname{lct}(B_y) \\ H_{x-1,y} & \text{otherwise.} \end{cases}$$

This leads the algorithm of Figure 3, for calculating lower bounds for the earliest start times of all operations in T. Obviously, the algorithm is $O(|T|^2)$, as the outer loop and both inner loops are of $O(|T|)$.

```
for $y := 1$ to $|T|$ do
    if $y = |T| \lor \operatorname{lct}(B_y) \neq \operatorname{lct}(B_{y+1})$ then
        $P := 0$; $g := -\infty$
        for $i := |T|$ down to 1 do
            if $\operatorname{lct}(A_i) \leq \operatorname{lct}(B_y)$ then
                $P := P + p(A_i)$;
                $g := g$ max $\operatorname{est}(A_i) + P$;
            endif
            $G_i := g$;
        endfor
        $H := -\infty$;
        for $x := 1$ to $|T|$ do
            if $\operatorname{lct}(A_x) > \operatorname{lct}(B_y)$ then
                if $\operatorname{est}(A_x) + P + p(A_x) > \operatorname{lct}(B_y)$ then
                    $\operatorname{est}(A_x) := \operatorname{est}(A_x)$ max $G_x$;
                endif
                if $H + p(A_x) > \operatorname{lct}(B_y)$ then
                    $\operatorname{est}(A_x) := \operatorname{est}(A_x)$ max $G_1$;
                endif
            else
                $H := H$ max $\operatorname{est}(A_x) + P$;
                $P := P - p(A_x)$;
            endif
        endfor
    endif
endfor
```

Figure 3: An $O(|T|^2)$ algorithm for calculating lower bounds for the earliest start times of operations in T.

The algorithm for finding upper bounds for the latest completion time can be developed analogous to the algorithm for finding lower bounds for the earliest start time of operations. The last issue we need to address is the checking of (1) of Lemma 4.1. If (1) does not hold, i.e.,

$$\exists_{S \subseteq T}\, e(S) + P(S) > C(S),$$

a dead end occurs. This is equivalent with

$$\exists_{x \in \{1,\ldots,|T|\}}\, \exists_{y \in \{1,\ldots,|T|\}}\, e(S_{x,y}) + P(S_{x,y}) > C(S_{x,y}),$$

which, in turn, is equivalent with

$$\exists_{x \in \{1,\ldots,|T|\}}\, \exists_{y \in \{1,\ldots,|T|\}}\, G_{x,y} > \operatorname{lct}(B_y).$$

From this the algorithm of Figure 4, to check (1) of Lemma 4.1, follows.

```
for $y := 1$ to $|T|$ do
    if $y = |T| \lor \mathrm{lct}(B_y) \neq \mathrm{lct}(B_{y+1})$ then
        $P := 0$; $g := -\infty$
        for $x := |T|$ down to 1 do
            if $\mathrm{lct}(A_x) \leq \mathrm{lct}(B_y)$ then
                $P := P + p(A_x)$;
                $g := g$ max $\mathrm{est}(A_x) + P$;
            endif
            if $g > \mathrm{lct}(B_y)$ then "report a dead end" endif
        endfor
    endif
endfor
```

Figure 4: An $O(|T|^2)$ algorithm for identifying dead ends.

What we did not use. In this section we discuss two ways of identifying inconsistent start times that we did not use for reasons explained in Section 4.2. We, however, do mention them as they can be used in combination with other ways of operation and start time selection.

The following example illustrates situations in which it is impossible to derive precedence relations between an operation o and a set of operations S, but in which it only can be derived that o is to be scheduled before or after all operations in S. This is a simple extension we made on the basis of Theorem 4.4 and 4.5. Figure 5 shows such a situation.

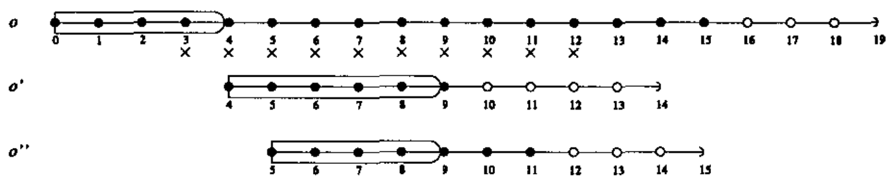

Figure 5: Three operations with $M(o) = M(o') = M(o'')$.

It easily can be seen that operation o cannot be started on $3-12$, as this makes scheduling both o' and o'' impossible. This leads to the following theorem.

Theorem 4.6. *Let $o \in \mathcal{O}$ and $S \subset \mathcal{O}$, such that $o \notin S$ and $\forall_{o' \in S}\, M(o') = M(o)$. If*

$$e(S) + P(S \cup \{o\}) > C(S),$$

then no schedule exists in which there are two operations $o', o'' \in S$ such that o' precedes o and o precedes o''.

Proof. Similar to the proof of Theorem 4.4. □

If we find that no schedule exists in which there are two operations $o', o'' \in S$ such that o' precedes o and o precedes o'', the open interval $(C(S) - P(S \cup \{o\}), e(S) + P(S))$ is deleted from $\delta(o)$.

The second additional way of identifying inconsistent start times is illustrated as follows. Regard the following search state as depicted in Figure 6. We have three operations o, o', and o'', all with processing time 3, earliest start time 0 and latest start time 6. This search state is fully arc consistent and no precedence relations can be derived. There exists, however, no schedule in which o is started on start time 1, as this implies that neither of the two other operations can be scheduled before o and that they cannot be scheduled both after o. Similarly, 4 is an inconsistent start time for o as this implies that neither of the two other operations can be scheduled after o and that they cannot be scheduled both before o. In Figure 6 all inconsistent start times are marked by '×'.

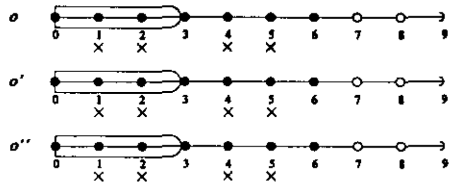

Figure 6: Three operations with $M(o) = M(o') = M(o'')$.

In order to treat the general case of this way of finding inconsistent start times, we need to introduce several notations.

- $\text{ect}(n, T) = \min_{U \subseteq T \mid |U|=n} e(U) + P(U)$ denotes the earliest completion time of any n operations of the set T.
- $\text{lst}(n, T) = \max_{U \subseteq T \mid |U|=n} C(U) - P(U)$ denotes the latest start time of any n operations of the set T.

Note that $\text{ect}(n, T)$ is actually a lower bound on the earliest completion time of any n operations. We remark that any other lower bound can be used in the sequel. Similarly, $\text{lst}(n, T)$ is an upper bound on the latest start time of any n operations, and any other upper bound can be used in the sequel.

Theorem 4.7. *Let $o \in \mathcal{O}$ and $S \subset \mathcal{O}$, such that $o \notin S$ and $\forall_{o' \in S}\, M(o') = M(o)$. Then operation o cannot be started on any value in the set given by*

$$\bigcup_{i=0}^{|S|-1} (\text{lst}(|S| - i, S) - p(o), \text{ect}(i + 1, S)).$$

Proof. Let $1 \leq i \leq |S|$ and suppose o is started on or after $\text{lst}(|S| - i, S) - p(o) + 1$, resulting in a completion time larger than $\text{lst}(|S| - i, S)$. As $\text{lst}(|S| - i, S)$ is the latest starting time of any $|S| - i$ operations from S, this implies that at most $|S| - i - 1$ operations can be scheduled after o. Now suppose o is started on or before $\text{ect}(i + 1, S) - 1$. As $\text{ect}(i + 1, S)$ is the first completion time of any $i + 1$ operations from S this implies that at most i operations can be scheduled before o. Therefore starting o on any $t \in [\text{lst}(|S| - i, S) - p(o) + 1, \text{ect}(i + 1, S) - 1]$ implies that at most $|S| - i - 1 + i = |S| - 1$ operations can be scheduled. □

4.2 Operation and start time selection

Based on the observation that every schedule can be transformed into a left justified schedule in which the operations are scheduled as early as possible while preserving the precedence constraints and the machine orderings, we decided to use operation and start time selection that together only construct left justified schedules. For reasons explained in Section 4.3 we furthermore introduce randomization. For operation selection we first determine the earliest minimal completion time of any unscheduled operation and then randomly select one operation that can be started before this completion time. The start time selection consists of selecting the earliest possible start time of this operation. The combination of this operation selection and start time selection results in the construction of only left justified schedules. Recall that the ways of identifying inconsistent start times based on Theorem 4.6 and Theorem 4.7 may result in the deletion of start times strictly larger than the earliest start time of an operation and strictly smaller than the latest completion time of an operation. By using these particular operation and start time selection we cannot benefit from such deletions and therefore we do not use aforementioned ways of identifying inconsistent start times. Another advantage of our way of operation and start time selection is that the number of possible decisions in each node of the search tree is drastically reduced. We remark that results of experiments we did with more sophisticated variable and value selection heuristics were quite disappointing; see [Van Erp Taalman Kip, 1993; Nuijten, Aarts, Van Erp Taalman Kip & Van Hee, 1993].

4.3 Dead end handling

As said before, once a tree search algorithm gets stuck in a dead end, some facility is needed to escape from it. We employ an approach that consists of two parts. First we try to solve the instance at hand by simply using chronological backtracking. However, if this does not lead to a solution after a reasonable number of backtrack steps, we use the most rigorous escape facility that is conceivable, namely the complete restart of the search. A problem then is to direct the search along a path different from the ones followed previously. Inspired by recent successes of randomized and other non-systematic search methods, like simulated annealing and genetic local search, we combine restarting the search with a randomized selection of a next operation and its start time as is described in Section 4.2. In this way the probability of following the same search path more than once is very small since the number of possible paths usually is very large. Furthermore, we use that if, by way of backtracking, it is derived that operation o cannot be scheduled on time t, it is implied that it cannot be started on any time in the interval $[t+1, \min_{o' \in O_{M(o)} \setminus \{o\}} \mathrm{ect}(o'))$. In this way we actually focus on determining machine orderings.

5 Computational results

We compared the performance of the following algorithms: the tabu search algorithm from [Dell'Amico & Trubian, 1993], referred to as TS, the simulated annealing algorithm from [Van Laarhoven, Aarts & Lenstra, 1992], referred to as SA, BOTTLE-8 from [Applegate & Cook, 1991], referred to as BOT, and our randomized constraint satisfaction algorithm, referred to as RCS. TS and SA are known to be the best approximation algorithms for the JSSP among a number of local search algorithms, including simulated annealing, tabu search, threshold accepting and genetic local search; see also [Aarts, Van Laarhoven, Lenstra & Ulder, 1992; Dorndorf & Pesch, 1992; Matsuo, Suh & Sullivan, 1988]. We included BOT in the comparison as it is a high-quality approximation algorithm tailored to the JSSP. BOTTLE-8 is the BOTTLE-X procedure of [Applegate & Cook, 1991] with X = 8. BOTTLE-X is an extension of the shifting bottleneck procedure of Adams, Balas & Zawack [1988]. We choose to use $X = 8$ as for smaller X the procedure finds relatively poor solutions using little computation time and for larger X the computation times become too large.

The comparison is carried out for a set of 40 instances of the JSSP due to [Lawrence,

Instance	LB/UB	n	m	TS		SA		BOT		RCS	
				%	t	%	t	%	t	%	t
F1	666	10	5	0	0.1	0	17.6	0	5.9	0	1.3
F2	655	10	5	0	18.8	0	16.7	1.06	7.8	0	12.8
F3	597	10	5	0	21.6	1.51	18.4	1.34	9.8	0	1.0
F4	590	10	5	0	32.2	0	17.3	0	12.5	0	1.7
F5	593	10	5	0	0.3	0	16.9	0	4.3	0	0.7
G1	926	15	5	0	0.3	0	40.9	0	8.5	0	1.5
G2	890	15	5	0	0.6	0	53.7	0	10.0	0	1.9
G3	863	15	5	0	0.3	0	41.7	0	9.5	0	2.1
G4	951	15	5	0	0.2	0	40.4	0	7.0	0	1.6
G5	958	15	5	0	0.2	0	34.7	0	7.6	0	1.5
H1	1222	20	5	0	0.4	0	89.6	0	11.8	0	3.2
H2	1039	20	5	0	0.2	0	93.6	0	12.7	0	3.5
H3	1150	20	5	0	0.4	0	80.6	0	9.9	0	3.1
H4	1292	20	5	0	0.4	0	66.0	0	10.7	0	2.5
H5	1207	20	5	0	1.2	0	105.1	0	17.7	0	16.6
A1	945	10	10	0	97.4	1.16	98.0	2.65	1912.6	0	194.8
A2	784	10	10	0	21.7	0.13	102.8	0.13	3018.4	0	2.5
A3	848	10	10	0	63.1	1.53	96.1	0	1606.4	0	325.3
A4	842	10	10	0	103.8	0.71	118.6	0	7605.0	0.12	235.2
A5	902	10	10	0	71.7	0	95.3	0.55	8680.6	0.55	67.9
B1	1040/1047	15	10	0.76	198.8	2.21	284.4	2.12	9674.0	2.79	862.5
B2	927	15	10	0.64	191.4	1.19	309.0	0.86	6182.9	1.08	594.3
B3	1032	15	10	0	1.8	0	299.0	0	16.9	0	99.2
B4	935	15	10	0.64	181.8	1.82	299.7	0.64	9682.6	0.75	522.0
B5	977	15	10	0.20	191.7	1.54	304.7	1.23	6049.4	0.41	1033.3
C1	1218	20	10	0	22.1	0	620.3	0	284.1	0	1477.4
C2	1235/1236	20	10	0.56	254.2	2.75	647.9	1.86	19238.9	4.05	1156.2
C3	1216	20	10	0	186.4	0.66	622.0	0.66	12034.2	0	650.5
C4	1120/1160	20	10	5.53	281.3	8.75	629.7	6.96	25119.9	9.91	667.1
C5	1355	20	10	0	10.4	0	565.1	0	11.1	0	169.3
D1	1784	30	10	0	2.1	0	216.7	0	20.8	0	41.9
D2	1850	30	10	0	2.2	0	250.3	0	18.3	0	92.9
D3	1719	30	10	0	1.8	0	268.6	0	13.1	0	111.6
D4	1721	30	10	0	5.1	0	269.4	0	19.6	0	291.6
D5	1888	30	10	0	1.3	0	238.3	0	15.8	0	77.9
I1	1268	15	15	0.78	238.4	1.97	763.7	2.44	5095.8	1.89	1338.1
I2	1397	15	15	0.85	242.2	2.57	755.3	1.65	5196.1	1.00	772.0
I3	1184/1196	15	15	1.60	256.6	2.62	782.9	6.76	471.8	7.94	1149.4
I4	1233	15	15	0.72	237.8	1.22	823.7	0.97	1761.9	0.41	1011.7
I5	1222	15	15	0.90	236.6	0.98	767.6	2.29	2274.2	2.05	841.5
FISH06	55	6	6	0	6.6	0	7.4	0	23.4	0	0.5
FISH10	930	10	10	0.53	156.6	2.26	111.3	0.86	199.3	0	255.8
FISH20	1165	20	5	0	260.2	1.63	121.1	3.86	49.0	0	10.7

Table 1: Performance comparison of tabu search (TS), simulated annealing (SA), BOTTLE-8 (BOT), and randomized constraint satisfaction (RCS) for different instances of the JSSP. See text for explanation of the symbols.

1984] and three instances due to [Fisher & Thompson, 1963]. The last three instances include the notorious 10-jobs 10-machines instance that has defied solution to optimality for more than twenty years. For every optimization instance mentioned above, we devised a number of constraint satisfaction instances by setting the overall deadline to the smallest integer larger than respectively $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\%$ above the minimal makespan. The tests for RCS and BOT were performed on a SPARC-station ELC. Van Laarhoven, Aarts & Lenstra [1992] did their tests for SA on a VAX-785. We divided the running times they report by 7 to get corresponding running times on a SPARC-station ELC. Dell'Amico & Trubian [1993] did their tests for TS on a PC 386.

Let N_I be the number of operations of an instance I. For RCS we choose to restart if the number of chronological backtrack steps exceeded $0.2 * N_I$. In Table 1 we give the results when allowing a maximum of 500 restarts (RCS). To reduce actual running times the restarts can performed in parallel. We used 10 SPARC-stations ELC and distributed the restarts over the various workstations until one of them found a solution. Evidently, this leads to a linear speed up. In order to keep the comparisons fair, we reported the accumulated running times of all 10 SPARC-stations. However, when using K processors, with K smaller than the number of restarts, running times obtained on one processor can be divided by K, to get the approximate resulting running times on K processors. For SA we choose to use the results obtained with the decrement parameter $\delta = 0.01$. Better results can be obtained with smaller values of δ, leading to increased running times; see [Van Laarhoven, Aarts & Lenstra, 1992]. This phenomenon of spending more time to get better results can also be observed with our algorithm; we can spend more time by simply allowing more restarts. In [Nuijten, Aarts, Van Erp Taalman Kip & Van Hee, 1993] some experiments are reported that show this behavior. For BOT we did the experiments as they are described in the text of [Applegate & Cook, 1991]. The results reported by them were obtained with a small variation of the algorithm.

The column LB/UB of Table 1 gives for each instance the minimal makespan if known. Otherwise, the best lower bound and upper bound found up to now are reported. The columns n and m give the number of jobs and the number of machines, respectively. For all instances, the number of operations of each job equals the number of machines. Furthermore, Table 1 shows for each of the algorithms the deviation from the lower bound for which a solution was found in terms of percentage (%), together with the running time in seconds (t). For TS, SA, and RCS the deviations from the lower bound are the best from a set of five runs using different seeds for the random number generator. The running times reported are averages over these five runs.

It can be observed that our algorithm performs well. From the 43 instances, 30 were solved to optimality, including the notorious FISH10 instance. For most other instances near-optimal solutions were found. Comparing RCS to the other algorithms leads to the conclusion that RCS is outperformed by TS, but that RCS performs comparable to SA and BOT. From the 14 instances that were not solved to optimality by both TS and RCS, TS outperforms RCS on 12. The only two instances for which RCS finds a better solution are I4 and FISH10. From the 20 instances that were not solved to optimality by both SA and RCS, SA outperforms RCS on 6, whereas RCS does better on 14 instances. If a same analysis is done for BOT and RCS, we come up with 7 instances for BOT and 12 for RCS.

6 Conclusion

We presented a randomized constraint satisfaction algorithm for the JSSP. We showed through an empirical performance analysis that our algorithm performs well, i.e., high quality solutions can be obtained within moderate running times. As our algorithm basically consists of a number of independent restarts, its effectiveness can be improved by allowing more restarts, this at the cost of larger running times. Another advantage of the independency of the restarts is that our algorithm can be easily run in parallel with a linear speed up.

As mentioned in Section 1, this paper is one in a series of papers in which we report on our investigations of the potentials of constraint satisfaction techniques for scheduling. Forthcoming papers will deal with three generalizations on the basic JSSP, viz., the introduction of

- machine alternatives, i.e., an operation may be processed on any one of a number of alternative machines thus adding the problem of finding a machine assignment for each operation,
- multiple capacity machines, i.e., machines may be able to execute more than one operation simultaneously, and
- operations that are executed on several machines simultaneously.

It is our general believe that constraint satisfaction techniques can be successfully applied to handle specific scheduling problems such as the JSSP, since it allows to incorporate specific problem structures which improve the quality of the obtained solutions. On the other hand constraint satisfaction techniques are sufficiently open to deal with more general problems and their constraints, which we hope to demonstrate in our forthcoming papers. The combination of being a satisfactory approach for job shop scheduling and the possibility of extending it to solve more general scheduling problems, makes our approach an interesting one to study.

References

AARTS, E.H.L., P.J.M. VAN LAARHOVEN, J.K. LENSTRA, AND N.J.L. ULDER [1992], *Job Shop Scheduling by Local Search*, Memorandum COSOR 92-29, Eindhoven University of Technology.

ADAMS, J., E. BALAS, AND D. ZAWACK [1988], The shifting bottleneck procedure for job shop scheduling, *Management Science* **34**, 391–401.

ADORF, H.-M., AND M.D. JOHNSTON [1990], A discrete stochastic neural network algortihm for constraint satisfaction problems, *Proc. International Joint Conference on Neural Networks*, 17–21.

APPLEGATE, D., AND W. COOK [1991], A computational study of the job-shop scheduling problem, *ORSA Journal on Computing* **3**, 149–156.

BITNER, J.R., AND E.M. REINGOLD [1975], Backtracking programming techniques, *Communications of the ACM* **11**, 651–656.

BRUCKER, P., B. JURISCH, AND B. SIEVERS [1992], *A Branch & Bound Algorithm for the Job-Shop Scheduling Problem*, Working document, University of Osnabrück, Germany.

CARLIER, J., AND E. PINSON [1989], An algorithm for solving the job-shop problem, *Management Science* **35**, 164–176.

CARLIER, J., AND E. PINSON [1990], A practical use of Jackson's preemptive schedule for solving the job shop problem, *Annals of Operations Research* **26**, 269–287.

DECHTER, R., AND J. PEARL [1988], Network-based heuristics for constraint-satisfaction problems, *Artificial Intelligence* **34**, 1–38.

DELL'AMICO, M., AND M. TRUBIAN [1993], Applying tabu-search to the job-shop scheduling problem, *Annals of Operations Research* **41**, 231–252.

DORNDORF, U., AND E. PESCH [1992], *Evolution Based Learning in a Job Shop Environment*, Working document, University of Limburg, the Netherlands.

ERP TAALMAN KIP, D.A.A. VAN [1993], Some constraint satisfaction algorithms for the generalized job shop scheduling problem, Master's thesis, Eindhoven University of Technology.

FISHER, H., AND G.L. THOMPSON [1963], Probabilistic learning combinations of local job-shop scheduling rules, in: J.F. Muth and G.L. Thompson (eds.), *Industrial Scheduling*, Prentice Hall.

FRENCH, S. [1982], *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*, Wiley & Sons.

FREUDER, E.C. [1982], A sufficient condition of backtrack-free search, *Journal ACM* **29**, 24–32.

GAREY, M.R., AND D.S. JOHNSON [1979], *Computers and Intractability; A Guide to the Theory of NP-completeness*, W.H. Freeman and Company, New York.

GAREY, M.R., D.S. JOHNSON, AND R. SETHI [1976], The complexity of flowshop and jobshop scheduling, *Mathematics of Operations Research* **1**, 117–129.

HARALICK, R.M., AND G.L. ELLIOTT [1980], Increasing tree search efficiency for constraint satisfaction problems, *Artificial Intelligence* **14**, 263–313.

HENTENRYCK, P. VAN, Y. DEVILLE, AND C.-M. TENG [1992], A generic arc-consistency algorithm and its specializations, *Artificial Intelligence* **57**, 291–321.

KENG, N., AND D. YUN [1989], A planning/scheduling methodology for the constrained resource problem, *Proc. 11th International Joint Conference on Artificial Intelligence*, 998–1003.

LAARHOVEN, P.J.M. VAN, E.H.L. AARTS, AND J.K. LENSTRA [1992], Job shop scheduling by simulated annealing, *Operations Research* **40**, 113–125.

LAWRENCE, S. [1984], *Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques (Supplement)*, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh.

MACKWORTH, A.K. [1977], Consistency in networks of relations, *Artificial Intelligence* **8**, 99–118.

MACKWORTH, A.K., AND E.C. FREUDER [1985], The complexity of some polynomial network consistency algorithms for constraint satisfaction problems, *Artificial Intelligence* **25**, 65–74.

MACKWORTH, A.K., AND E.C. FREUDER [1993], The complexity of constraint satisfaction revisited, *Artificial Intelligence* **59**, 57–62.

MATSUO, H., C.J. SUH, AND R.S. SULLIVAN [1988], *A Controlled Search Simulated Annealing Method for the General Job Shop Scheduling Problem*, Working Paper 03-04-88, Graduate School of Business, University of Texas at Austin, Austin, USA.

MINTON, S., M.D. JOHNSTON, A.B. PHILIPS, AND P. LAIRD [1992], Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems, *Artificial Intelligence* **58**, 161–205.

MOHR, R., AND T.C. HENDERSON [1986], Arc and path consistency revisited, *Artificial Intelligence* **28**, 225–233.

MONTANARI, U. [1974], Networks of constraints: Fundamental properties and applications to picture processing, *Information Sciences* **7**, 95 – 132.

MUSCETTOLA, N. [1993], Scheduling by iterative partition of bottleneck conflicts., *Proc. 9th IEEE Conference on AI Applications*, 49–55.

NADEL, B.A. [1989], Constraint satisfaction algorithms, *Computational Intelligence* **5**, 188–224.

NUDEL, B. [1983], Consistent-labeling problems and their algorithms: Expected-complexities and theory-based heuristics, *Artificial Intelligence* **21**, 135–178.

NUIJTEN, W.P.M., E.H.L. AARTS, D.A.A. VAN ERP TAALMAN KIP, AND K.M. VAN HEE [1993], Randomized constraint satisfaction for job shop scheduling, *IJCAI '93 Workshop on Knowledge-Based Production Planning, Scheduling and Control.*

PURDOM, JR., P.W. [1983], Search rearrangement backtracking and polynomial average time, *Artificial Intelligence* **21**, 117–133.

SADEH, N. [1991], *Look-ahead Techniques for Micro-opportunistic Job Shop Scheduling*, Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

SMITH, S.F., AND C.-C. CHENG [1993], Slack-based heuristics for constraint satisfaction, *Proc. 11th National Conference on Artificial Intelligence.*

STALLMAN, R.M., AND G.J. SUSSMAN [1977], Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis., *Artificial Intelligence* **9**, 135–196.

WALTZ, D.L. [1972], *Generating Semantic Descriptions from Drawings of Scenes with Shadows*, Tech. Rept. AI271, MIT, Cambridge, MA.

9

91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.
91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.
91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.

In this series appeared:

91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.
91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.

92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.
92/04	J.P.H.W.v.d.Eijnde	Program derivation in acyclic graphs and related problems, p. 90.
92/05	J.P.H.W.v.d.Eijnde	Conservative fixpoint functions on a graph, p. 25.
92/06	J.C.M. Baeten J.A. Bergstra	Discrete time process algebra, p.45.
92/07	R.P. Nederpelt	The fine-structure of lambda calculus, p. 110.
92/08	R.P. Nederpelt F. Kamareddine	On stepwise explicit substitution, p. 30.
92/09	R.C. Backhouse	Calculating the Warshall/Floyd path algorithm, p. 14.
92/10	P.M.P. Rambags	Composition and decomposition in a CPN model, p. 55.
92/11	R.C. Backhouse J.S.C.P.v.d.Woude	Demonic operators and monotype factors, p. 29.
92/12	F. Kamareddine	Set theory and nominalisation, Part I, p.26.
92/13	F. Kamareddine	Set theory and nominalisation, Part II, p.22.
92/14	J.C.M. Baeten	The total order assumption, p. 10.
92/15	F. Kamareddine	A system at the cross-roads of functional and logic programming, p.36.
92/16	R.R. Seljée	Integrity checking in deductive databases; an exposition, p.32.
92/17	W.M.P. van der Aalst	Interval timed coloured Petri nets and their analysis, p. 20.
92/18	R.Nederpelt F. Kamareddine	A unified approach to Type Theory through a refined lambda-calculus, p. 30.
92/19	J.C.M.Baeten J.A.Bergstra S.A.Smolka	Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.
92/20	F.Kamareddine	Are Types for Natural Language? P. 32.
92/21	F.Kamareddine	Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.

92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottum-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for Fω, p. 15.
92/26	T.H.W.Beelen W.J.J.Stut P.A.C.Verkoulen	A modelling method using MOVIE and SimCon/ExSpect, p. 15.
92/27	B. Watson G. Zwaan	A taxonomy of keyword pattern matching algorithms, p. 50.
93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.
93/14	J.C.M. Baeten J.A. Bergstra	On Sequential Composition, Action Prefixes and Process Prefix, p. 21.

93/15	J.C.M. Baeten J.A. Bergstra R.N. Bol	A Real-Time Process Logic, p. 31.
93/16	H. Schepers J. Hooman	A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
93/17	D. Alstein P. van der Stok	Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
93/18	C. Verhoef	A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
93/19	G-J. Houben	The Design of an Online Help Facility for ExSpect, p.21.
93/20	F.S. de Boer	A Process Algebra of Concurrent Constraint Programming, p. 15.
93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correctness?, p. 24.
93/22	E. Poll	A Typechecker for Bijective Pure Type Systems, p. 28.
93/23	E. de Kogel	Relational Algebra and Equational Proofs, p. 23.
93/24	E. Poll and Paula Severi	Pure Type Systems with Definitions, p. 38.
93/25	H. Schepers and R. Gerth	A Compositional Proof Theory for Fault Tolerant Real-Time Distributed Systems, p. 31.
93/26	W.M.P. van der Aalst	Multi-dimensional Petri nets, p. 25.
93/27	T. Kloks and D. Kratsch	Finding all minimal separators of a graph, p. 11.
93/28	F. Kamareddine and R. Nederpelt	A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.
93/29	R. Post and P. De Bra	GOLD, a Graph Oriented Language for Databases, p. 42.
93/30	J. Deogun T. Kloks D. Kratsch H. Müller	On Vertex Ranking for Permutation and Other Graphs, p. 11.
93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.
93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.

7

93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.

[illegible]