Slack-based Heuristics for JSSP

Heuristics in choco

Variable ordering

Value ordering

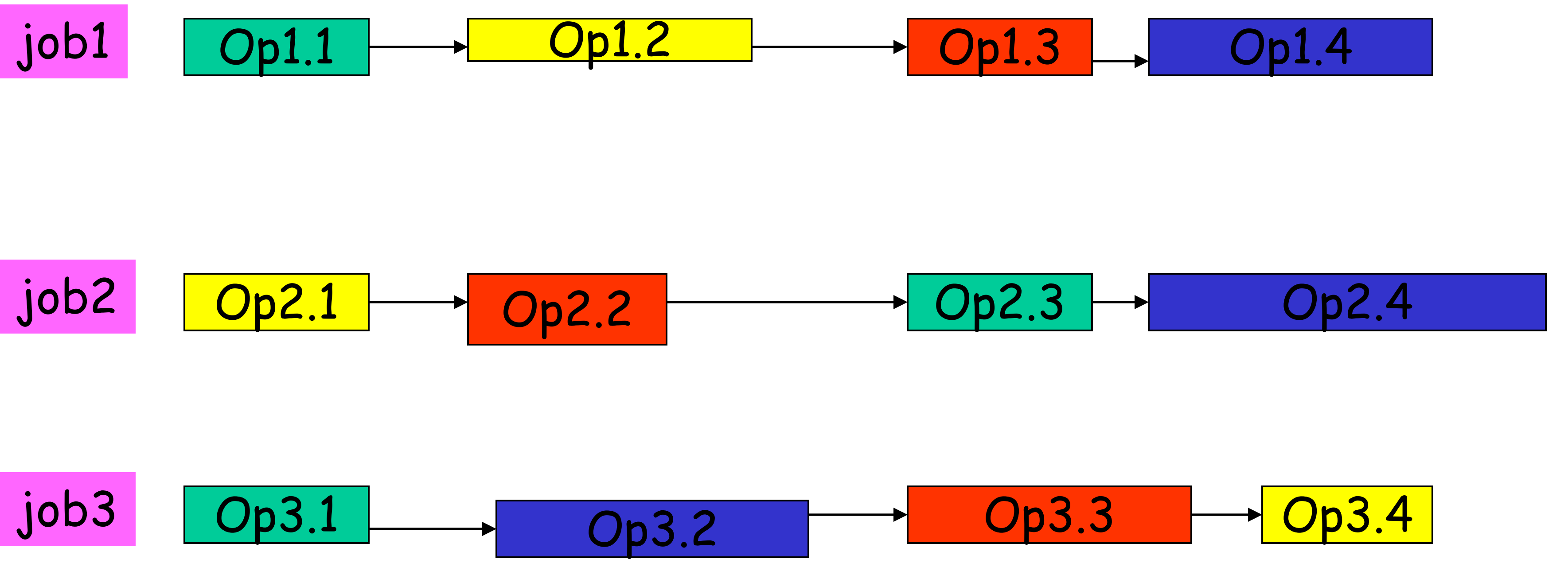
jssp refresh

We have

- a set of resources
- a set of jobs
    - a job is a sequence of operations/activities

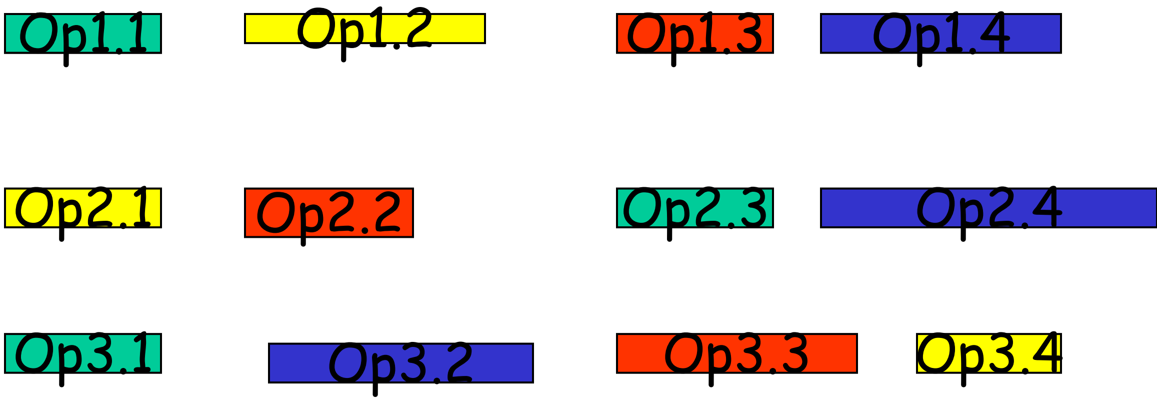
- sequence the activities on the resources

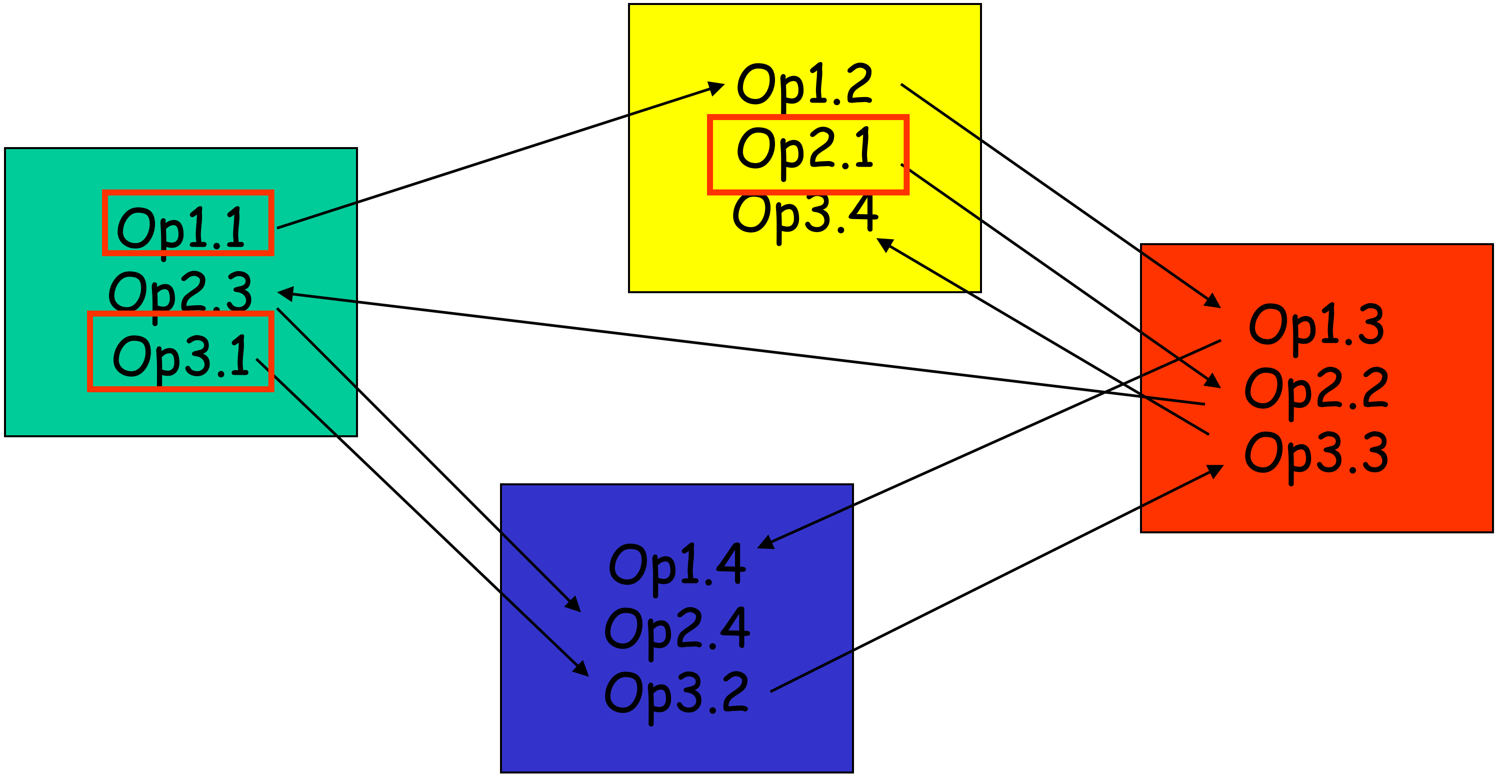
An example: 3 x 4



- We have 4 resources: green, yellow, red and blue
- a job is a sequence of operations (precedence constraints)
- each operation is executed on a resource (resource constraints)
- each resource can do one operation at a time
- the duration of an operation is the length of its box
- we have a due date, giving time windows for operations (time constraints)

An example: 3 x 4



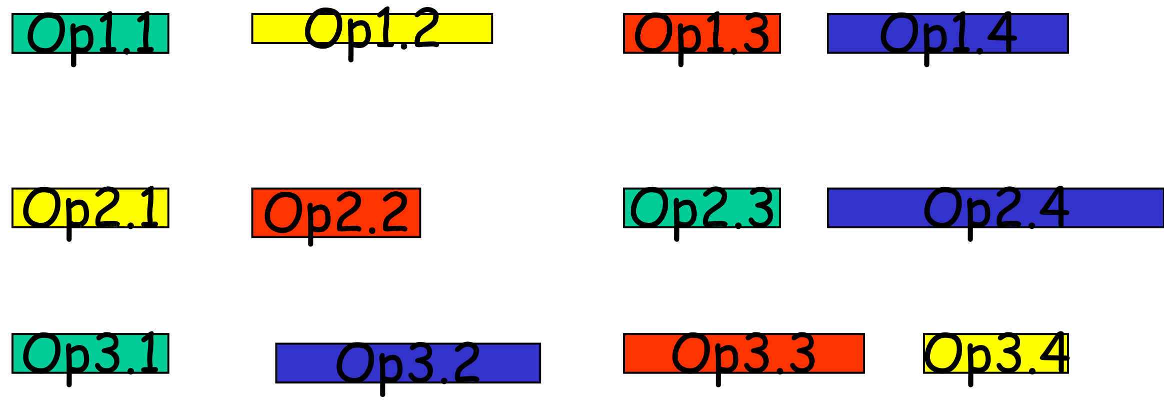


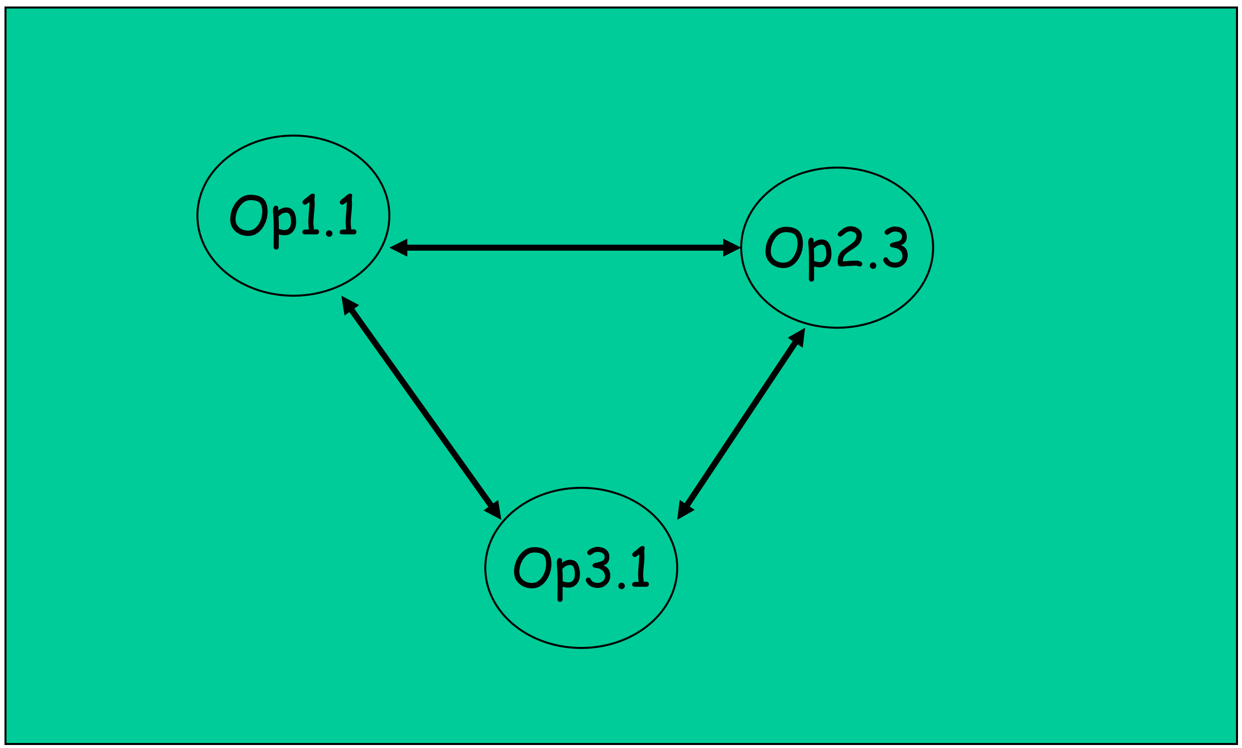
The problem

Assign a start time to each operation such that
(a) no two operations are in process on the same machine at the same time and
(b) time constraints are respected

The problem is NP complete

An example: 3 x 4

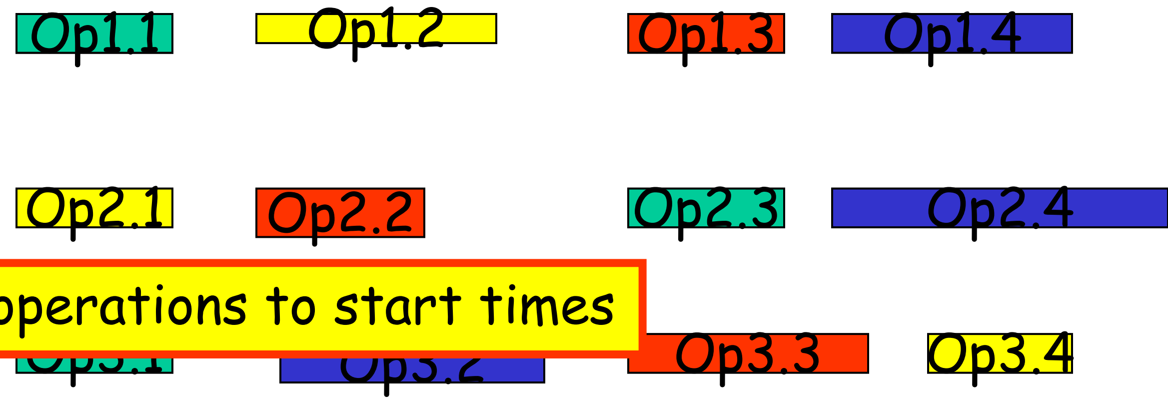




On the "green" resource, put a ***direction*** on the arrows

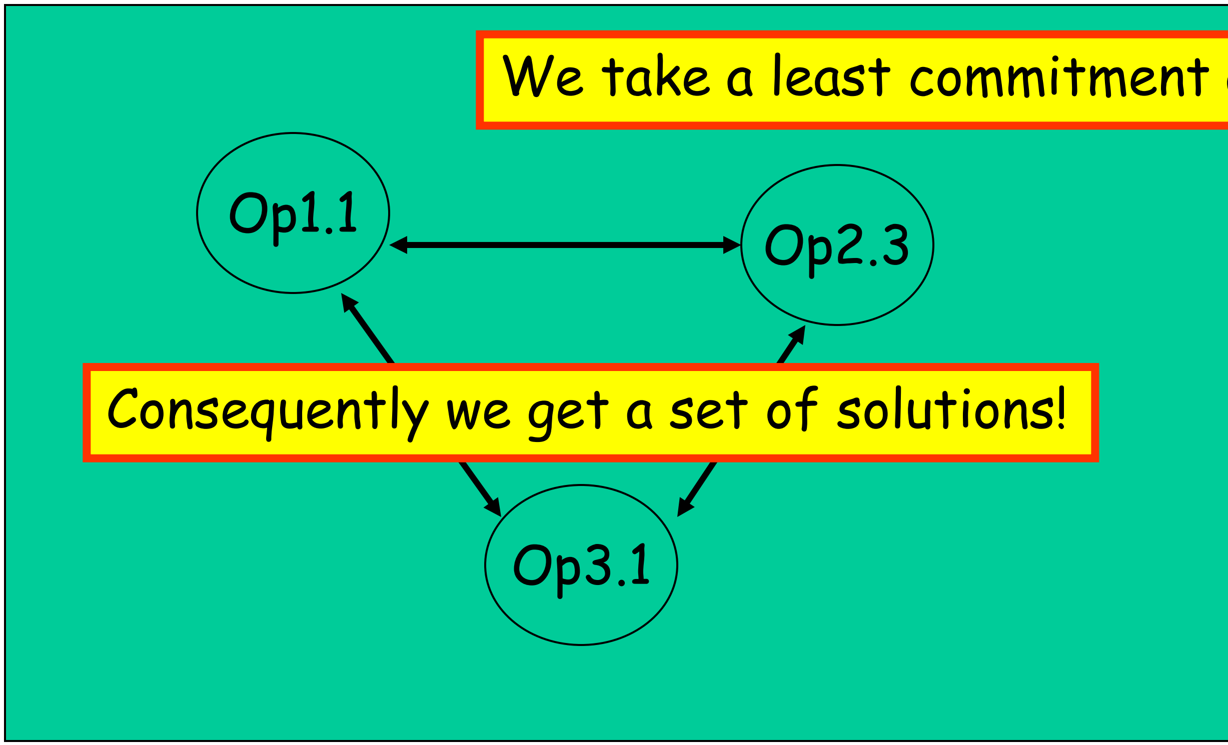
A disjunctive graph

An example: 3 x 4



We do not bind operations to start times

We take a least commitment approach



Consequently we get a set of solutions!

On the “green” resource, put a ***direction*** on the arrows

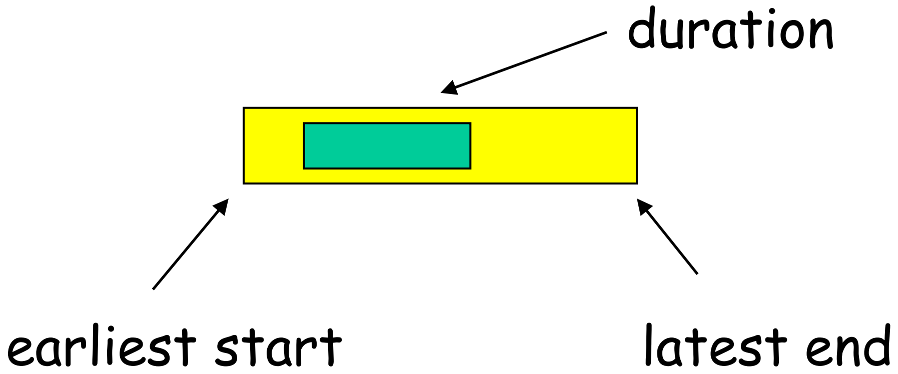
A disjunctive graph

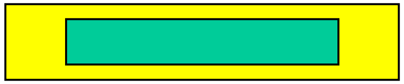
```java
Constraint before(Operation op2){
    return model.arithm(op2.start,">=",start,"+",duration);
}
```

op1.before(op2)

Picture of an operation

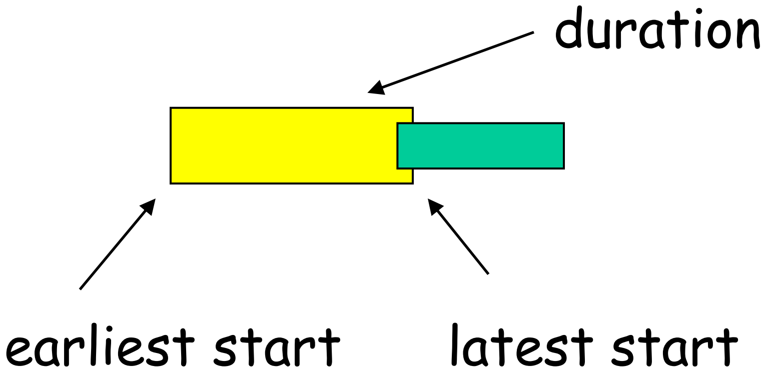
op1.before(op2)

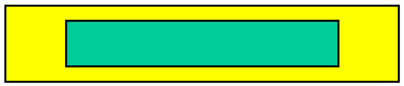




Picture of an operation

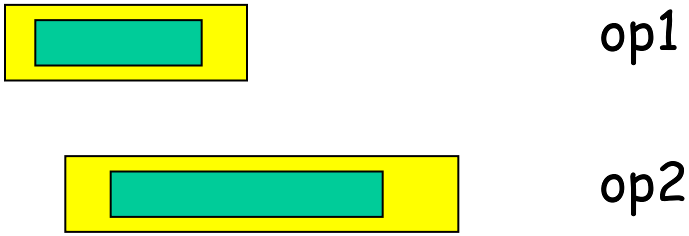
op1.before(op2)





Constrained integer variable represents start time

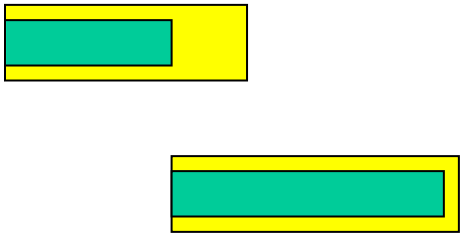
Picture of an operation



op1.before(op2) ⟶ op1.start() + op1.duration() ≤ op2.start()

Picture of an operation

op1.before(op2)



op1

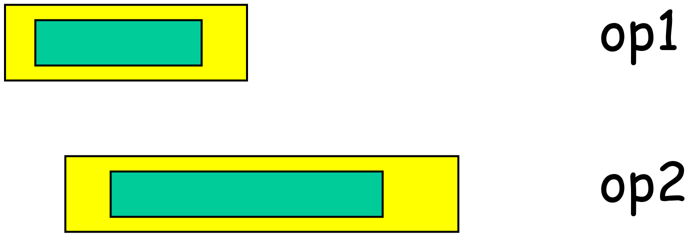
op2

propagate

op1.before(op2) ⟶ op1.start() + op1.duration() ≤ op2.start()

Picture of an operation

op1.before(op2)



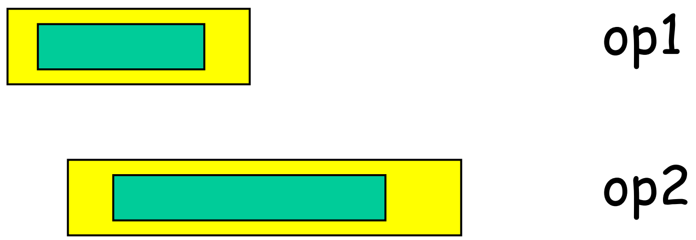
op1 and op2 cannot be in process at same time

→ op1.before(op2) ***OR*** op2.before(op1)

Not easy to propagate until
decision made (disjunction broken)

Picture of an operation

op1.before(op2)



Use a 0/1 decision variable d[i][j] as follows

$d[i][j] = 0 \rightarrow$ op[i]1.before(op[j])

$d[i][j] = 1 \rightarrow$ op[j]1.before(op[i]) heuristics

JUST ONE EXAMPLE

Stephen F. Smith

Research Professor, Robotics
Director, Intelligent Coordination and Logistics Laboratory

The Robotics Institute
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213

Email: sfs@cs.cmu.edu, Phone: (412) 268-8811, Fax: (412) 268-5569
Office: 1502E Newell & Simon Hall

ICLL Home - Windows Internet Explorer

http://www.ozone.ri.cmu.edu/

INTELLIGENT COORDINATION AND LOGISTICS LABORATORY

HOME | INFORMATION | PERSONNEL | PROJECTS | PUBLICATIONS | CONTACT

HOME

INTELLIGENT COORDINATION & LOGISTICS LABORATORY

Stephen F. Smith, Director

Problems of large-scale coordination and logistics are ubiquitous, and better solutions are becoming increasingly critical in many domains. In manufacturing, trends toward industrial globalization and constrained market focus on high value-adding products, together with new coordination concepts such as electronic marketplaces, require organizations to become more agile. Military command and control infra-structure is faced with shrinking budgets and personnel, even though current geopolitical realities demand improved capability for rapid crisis-action mission planning and deployment. The rising cost of health care places a premium on more efficient methods for administration and delivery.

CP4: Papers - Windows Internet Explorer

smith-cheng-slack-aaai93.pdf (SECURED) - Adobe Reader

File Edit View Document Tools Window Help

1 / 6 97% Find

Slack-Based Heuristics For Constraint Satisfaction Scheduling *

Stephen F. Smith
The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213
sfs@isl1.ri.cmu.edu

Cheng-Chung Cheng
The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213
ccen@isl1.ri.cmu.edu

Abstract

In this paper, we define and empirically evaluate new heuristics for solving the job shop scheduling problem with non-relaxable time windows. The hypothesis underlying our approach is that by approaching the problem as one of establishing sequencing constraints between pairs of operations requiring the same resource (as opposed to a problem of assigning start times to each operation) and by exploiting previously developed analysis techniques for limiting search through the space of possible sequencing decisions, simple, localized look-ahead techniques can yield problem solving performance comparable to currently dominating techniques that rely on more sophisticated analysis of resource contention. We define a series of attention focusing heuristics based on simple analysis of the temporal flexibility associated with different sequencing decisions, and a similarly moti-

exclusive use of a designated machine for the duration of its processing (i.e. machines have unit processing capacity). Each job has an associated ready time and a deadline, and its production must be accomplished within this interval. The problem can be extended in various ways - to include selection among designated resource alternatives for each operation, to associate multiple resource requirements (e.g. machine, operator) with operations, etc. In any case, the objective is to determine a schedule for production that satisfies all temporal and resource capacity constraints.

The job shop scheduling with non-relaxable time windows problem is known to be NP-Complete (Garey & Johnson 1979). Accordingly, the development of effective heuristic procedures for solving this constraint satisfaction problem (CSP) has been the subject of considerable previous research. This work, with few exceptions, has sought to exploit the special structure of the problem, in particular the structure of resource

start papers Microsoft Pow... CP4: Papers - ... smith-cheng-sl... 15:21

straints. The solutions generated in this way typically represent a set of feasible schedules (i.e., the sets of operation start times that remain consistent with posted sequencing constraints), as opposed to a single assignment of operation start times. In (Erschler et al. 1976, 1980) the structure of resource capacity constraints is exploited to define dominance conditions for pruning the set of feasible sequencing alternatives at each stage of the search. More recently, (Muscettola 1993) has demonstrated the utility of global resource capacity analysis techniques (similar in spirit to the approach in (Sadeh 1991)) as a focusing mechanism within this alternative search space; in this case sequencing constraints are repeatedly posted between sets of conflicting operations until resource capacity analysis indicates no further possibility of resource contention.

Like (Muscettola 1993), we believe that the inherent flexibility gained by providing sets of feasible solutions offers considerable pragmatic value over typically overconstrained fixed times solutions. The principal claim of this paper, however, is that this second formulation of the problem also provides a more convenient search space in which to operate. When the problem is cast as a search for orderings between pairs of operations vying for the same resource, we argue that it is possible to obtain the look-ahead benefits of global resource capacity analysis through the use of simpler, local analysis of the sequencing possibilities associated with unordered operation pairs. We define a series of variable ordering heuristics based on measures of temporal slack which, when integrated with the search space pruning techniques developed in (Erschler et al. 1976), are shown to yield comparable problem solving performance to contention-based heuristics at a fraction of the compu-

- **resource capacity constraints** - for any two operations i and j requiring the same resource, $st_i + p_i \leq st_j \vee st_j + p_j \leq st_i$
- **ready times and deadlines** - for each operation i of job $\mathcal{J}$, $r_{\mathcal{J}} \leq st_i$ and $st_i + p_i \leq d_{\mathcal{J}}$, where $r_{\mathcal{J}}$ and $d_{\mathcal{J}}$ are the ready time and deadline respectively associated with job $\mathcal{J}$.

While this problem representation provides a direct basis for problem solving search (and in fact has been taken as the starting point of most previous research), the problem can be alternatively formulated as one of establishing sequencing constraints between pairs of operations contending for the same resource over time. In this case, we define a decision variable $ordering_{i,j}$ for each pair of operations i and j that require the same resource, which can take on either of two values: $i \rightarrow j$ (implying the constraint $st_i + p_i \leq st_j$) and $j \rightarrow i$ (implying $st_j + p_j \leq st_i$). A solution then is a consistent assignment of values to all ordering variables. There are several potential advantages to this formulation. The advantage emphasized in this paper is that the simpler structure of the search space enables more straightforward accounting of resource capacity constraints and the use of simpler, localized analysis of current solution structure as a basis for variable and value ordering.

Our problem solving framework assumes a backtrack search procedure in which the solution is incrementally extended through the repeated selection and binding of an as yet unconstrained $ordering_{i,j}$ variable (referred to as the posting of a new precedence relation). Whenever a new precedence relation is posted, constraint propagation is performed to ensure continued tempostraints. The solutions generated in this way typically represent a set of feasible schedules (i.e., the sets of operation start times that remain consistent with posted sequencing constraints), as opposed to a single assignment of operation start times. In (Erschler et al. 1976, 1980) the structure of resource capacity constraints is exploited to define dominance conditions for pruning the set of feasible sequencing alternatives at each stage of the search. More recently, (Muscettola 1993) has demonstrated the utility of global resource capacity analysis techniques (similar in spirit to the approach in (Sadeh 1991)) as a focusing mechanism within this alternative search space; in this case sequencing constraints are repeatedly posted between sets of conflicting operations until resource capacity analysis indicates no further possibility of resource contention.

Like (Muscettola 1993), we believe that the inherent flexibility gained by providing sets of feasible solutions offers considerable pragmatic value over typically overconstrained fixed times solutions. The principal claim of this paper, however, is that this second formulation of the problem also provides a more convenient search space in which to operate. When the problem is cast as a search for orderings between pairs of operations vying for the same resource, we argue that it is possible to obtain the look-ahead benefits of global resource capacity analysis through the use of simpler, local analysis of the sequencing possibilities associated with unordered operation pairs. We define a series of variable ordering heuristics based on measures of temporal slack which, when integrated with the search space pruning techniques developed in (Erschler et al. 1976), are shown to yield comparable problem solving performance to contention-based heuristics at a fraction of the compu-

- **resource capacity constraints** - for any two operations i and j requiring the same resource, $st_i + p_i \leq st_j \vee st_j + p_j \leq st_i$
- **ready times and deadlines** - for each operation i of job $\mathcal{J}$, $r_{\mathcal{J}} \leq st_i$ and $st_i + p_i \leq d_{\mathcal{J}}$, where $r_{\mathcal{J}}$ and $d_{\mathcal{J}}$ are the ready time and deadline respectively associated with job $\mathcal{J}$.

While this problem representation provides a direct basis for problem solving search (and in fact has been taken as the starting point of most previous research), the problem can be alternatively formulated as one of establishing sequencing constraints between pairs of operations contending for the same resource over time. In this case, we define a decision variable $ordering_{i,j}$ for each pair of operations i and j that require the same resource, which can take on either of two values: $i \rightarrow j$ (implying the constraint $st_i + p_i \leq st_j$) and $j \rightarrow i$ (implying $st_j + p_j \leq st_i$). A solution then is a consistent assignment of values to all ordering variables. There are several potential advantages to this formulation. The advantage emphasized in this paper is that the simpler structure of the search space enables more straightforward accounting of resource capacity constraints and the use of simpler, localized analysis of current solution structure as a basis for variable and value ordering.

Our problem solving framework assumes a backtrack search procedure in which the solution is incrementally extended through the repeated selection and binding of an as yet unconstrained $ordering_{i,j}$ variable (referred to as the posting of a new precedence relation). Whenever a new precedence relation is posted, constraint propagation is performed to ensure continued tempoof the problem also provides a more convenient search space in which to operate. When the problem is cast as a search for orderings between pairs of operations vying for the same resource, we argue that it is possible to obtain the look-ahead benefits of global resource capacity analysis through the use of simpler, local analysis of the sequencing possibilities associated with unordered operation pairs. We define a series of variable ordering heuristics based on measures of temporal slack which, when integrated with the search space pruning techniques developed in (Erschler et al. 1976), are shown to yield comparable problem solving performance to contention-based heuristics at a fraction of the computational cost.

The remainder of the paper is organized as follows. In Section 2, we specify the problem as a CSP search for operation pair orderings, and review dominance conditions that enable search space pruning relative to this model. In Sections 3 through 5, we propose a series of variable ordering heuristics and present comparative results on a previously studied suite of 60 test problems. Finally, in Section 6, we outline current work in applying the approach to schedule optimization.

Problem Representation and Search Framework

In more precise terms, a solution to the basic job shop scheduling CSP requires a consistent assignment of values to start time variables st_i for each operation i, under the following constraints:

- **sequencing restrictions** - for every precedence relation $i \rightarrow j$ specified between operations i and j in the process plan of a given job $\mathcal{J}$, $st_i + p_i \leq st_j$, where p_i is the processing time required by operation i of job $\mathcal{J}$.

formulation. The advantage emphasized in this paper is that the simpler structure of the search space enables more straightforward accounting of resource capacity constraints and the use of simpler, localized analysis of current solution structure as a basis for variable and value ordering.

Our problem solving framework assumes a backtrack search procedure in which the solution is incrementally extended through the repeated selection and binding of an as yet unconstrained $ordering_{i,j}$ variable (referred to as the posting of a new precedence relation). Whenever a new precedence relation is posted, constraint propagation is performed to ensure continued temporal consistency and maintain current bounds on the earliest start time and latest finish time of each operation. [1] If the decision $i \rightarrow j$ is taken, for example, then est_j (the earliest start time of j) and lft_i (the latest finish time of i) are updated by

$$est_j = \max\{est_j, est_i + p_i\}, \text{ and} \quad (1)$$

$$lft_i = \min\{lft_i, lft_j - p_j\}, \quad (2)$$

and these new values are then propagated forward or backward respectively through all pre-specified and posted temporal precedence relations. If during this process, $est_k + p_k$ becomes greater than lft_k for any operation k then an inconsistent set of assignments has been detected.

As indicated at the outset, our approach to directing the search integrates a procedure previously developed by Erschler *et al*, referred to as *Constraint-based Analysis (CBA)*, which exploits dominance conditions to prune the space of possible ordering assignments. To summarize their basic idea, assume that est_i and lft_i

[1] Since we are assuming in this paper that operation processing times are fixed, we could equivalently reason in terms of earliest and latest start times.

different cases:

1. If $lft_i - est_j < p_i + p_j \leq lft_j - est_i$ then i must be scheduled before j in any feasible extension of the current ordering decisions. (case 1)
2. If $lft_j - est_i < p_i + p_j \leq lft_i - est_j$ then j must be scheduled before i in any feasible extension of the current ordering decisions. (case 2)
3. If $p_i + p_j > lft_j - est_i$ and $p_i + p_j > lft_i - est_j$ then there is no feasible schedule. (case 3)
4. If $p_i + p_j \leq lft_j - est_i$ and $p_i + p_j \leq lft_i - est_j$ then either sequencing decision is still possible. (case 4)

These dominance conditions of course provide only necessary conditions for determining a set of feasible schedules, and thus interleaved application of CBA and temporal constraint propagation yields an underspecified search procedure. What is needed to generate solutions are heuristics for resolving the undecided states specified in case 4. In this regard, previous use of CBA has emphasized fuzzy integration of sets of different scheduling rules. In (Bensana & Dubois 1988), a voting procedure based on fuzzy set theory and approximate reasoning was developed and used in conjunction with a set of fuzzy scheduling rules. In (Kerr & Walker 1989), fuzzy arithmetic together with fuzzy scheduling rules was utilized instead. Our goal, alternatively, is to investigate the effectiveness of CBA in conjunction with simple look-ahead analysis of current ordering flexibility. This leads to the search procedure that is graphically depicted in Figure 1, which we will refer to as precedence constraint posting (PCP). In the following sections, we define and evaluate a specific set of variable and value ordering heuristics.

Exploiting Estimates of Sequencing Flexibility

Intuitively, in situations where CBA leaves the search

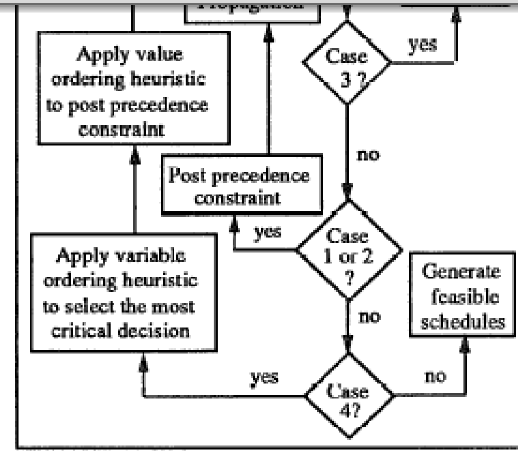


Figure 1: PCP Search Procedure

we define two measures, corresponding to the two possible decisions that might be taken. For a given pair of currently unordered operations (i, j) contending for the same resource, we define the "temporal slack remaining after sequencing i before j" as

$$slack(i \rightarrow j) = lft_j - est_i - (p_i + p_j), \qquad (3)$$

and similarly the "temporal slack remaining after sequencing j before i" as

$$slack(j \rightarrow i) = lft_i - est_j - (p_i + p_j). \qquad (4)$$

Figure 2 provides a graphic illustration of $slack(i \rightarrow j)$ and $slack(j \rightarrow i)$. Note that in either case the remaining slack is shared by both i and j. Thus, the larger the temporal slack, the greater the chance that subsequent ordering decisions involving i and j can be feasibly imposed.

Given these measures of temporal slack, we now have a basis for identifying the most constrained or "most critical" decision and for specifying an initial variable ordering heuristic. We define the ordering decision different cases:

1. If $lft_i - est_j < p_i + p_j \leq lft_j - est_i$ then i must be scheduled before j in any feasible extension of the current ordering decisions. (case 1)
2. If $lft_j - est_i < p_i + p_j \leq lft_i - est_j$ then j must be scheduled before i in any feasible extension of the current ordering decisions. (case 2)
3. If $p_i + p_j > lft_j - est_i$ and $p_i + p_j > lft_i - est_j$ then there is no feasible schedule. (case 3)
4. If $p_i + p_j \leq lft_j - est_i$ and $p_i + p_j \leq lft_i - est_j$ then either sequencing decision is still possible. (case 4)

These dominance conditions of course provide only necessary conditions for determining a set of feasible schedules, and thus interleaved application of CBA and temporal constraint propagation yields an underspecified search procedure. What is needed to generate solutions are heuristics for resolving the undecided states specified in case 4. In this regard, previous use of CBA has emphasized fuzzy integration of sets of different scheduling rules. In (Bensana & Dubois 1988), a voting procedure based on fuzzy set theory and approximate reasoning was developed and used in conjunction with a set of fuzzy scheduling rules. In (Kerr & Walker 1989), fuzzy arithmetic together with fuzzy scheduling rules was utilized instead. Our goal, alternatively, is to investigate the effectiveness of CBA in conjunction with simple look-ahead analysis of current ordering flexibility. This leads to the search procedure that is graphically depicted in Figure 1, which we will refer to as precedence constraint posting (PCP). In the following sections, we define and evaluate a specific set of variable and value ordering heuristics.

Exploiting Estimates of Sequencing Flexibility

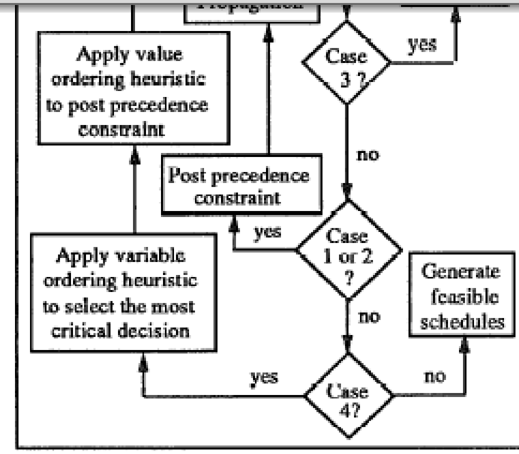


Figure 1: PCP Search Procedure

we define two measures, corresponding to the two possible decisions that might be taken. For a given pair of currently unordered operations (i, j) contending for the same resource, we define the "temporal slack remaining after sequencing i before j" as

$$slack(i \rightarrow j) = lft_j - est_i - (p_i + p_j), \quad (3)$$

and similarly the "temporal slack remaining after sequencing j before i" as

$$slack(j \rightarrow i) = lft_i - est_j - (p_i + p_j). \quad (4)$$

Figure 2 provides a graphic illustration of $slack(i \rightarrow j)$ and $slack(j \rightarrow i)$. Note that in either case the remaining slack is shared by both i and j. Thus, the larger the temporal slack, the greater the chance that subsequent ordering decisions involving i and j can be feasibly imposed.

Given these measures of temporal slack, we now have a basis for identifying the most constrained or "most critical" decision and for specifying an initial variable ordering heuristic. We define the ordering decision scheduling rules was utilized instead. Our goal, alternatively, is to investigate the effectiveness of CBA in conjunction with simple look-ahead analysis of current ordering flexibility. This leads to the search procedure that is graphically depicted in Figure 1, which we will refer to as precedence constraint posting (PCP). In the following sections, we define and evaluate a specific set of variable and value ordering heuristics.

Exploiting Estimates of Sequencing Flexibility

Intuitively, in situations where CBA leaves the search in a state with several unresolved ordering assignments (i.e., for each unordered operation pair, both ordering decisions are still feasible), we would like to focus attention on the ordering decision that is currently most constrained. Since the posting of any sequence constraint is likely to further constrain other ordering decisions that remain to be made, delaying the currently most constrained decision increases the chances of arriving at an infeasible problem solving state.

Implementation of such a variable ordering strategy requires a means of estimating the current flexibility associated with a given unresolved ordering decision. One simple indicator of flexibility is the amount of temporal slack that is retained by a given operation pair if a decision to sequence them is taken. To this end,

quencing j before i as

$$slack(j \rightarrow i) = lft_i - est_j - (p_i + p_j). \qquad (4)$$

Figure 2 provides a graphic illustration of $slack(i \rightarrow j)$ and $slack(j \rightarrow i)$. Note that in either case the remaining slack is shared by both i and j. Thus, the larger the temporal slack, the greater the chance that subsequent ordering decisions involving i and j can be feasibly imposed.

Given these measures of temporal slack, we now have a basis for identifying the most constrained or "most critical" decision and for specifying an initial variable ordering heuristic. We define the ordering decision with the **overall minimum slack**, to be the decision $ordering_{i,j}$ for which

$$\min\{slack(i \rightarrow j), slack(j \rightarrow i)\} = \min_{(u,v)}\{\min\{slack(u \rightarrow v), slack(v \rightarrow u)\}\}$$

for all unassigned $ordering_{u,v}$. Using this notion of criticality, we define a variable ordering heuristic that selects this decision at each unresolved state of the search.

With respect to the decision of which sequencing constraint to post (i.e., value assignment), we intuitively prefer the decision that leaves the search with the most degrees of freedom. Thus we post the sequencing constraint that retains the largest amount of temporal slack.

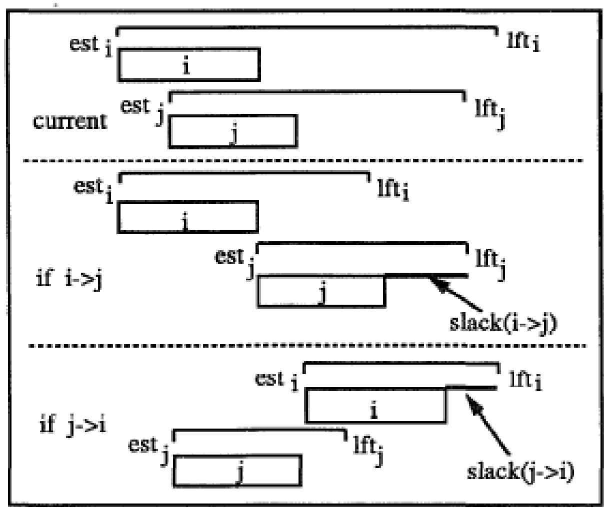


Figure 2: Slack($i \rightarrow j$) and Slack($j \rightarrow i$)

Summarizing then, our initial configuration of variable and value ordering heuristics is defined as follows:

I. **Min-Slack** variable ordering: Select the sequencing decision with the overall minimum temporal slack. Suppose this decision is $ordering_{i,j}$.

II. **Max-Slack** value ordering: choose the sequencing constraint $i \rightarrow j$ if $slack(i \rightarrow j) > slack(j \rightarrow i)$; otherwise choose $j \rightarrow i$.

A Computational Study

In this section we evaluate the performance of the above heuristics in conjunction with the PCP search

tion). Both ORR/FSS and CPS have reported very strong results on the set of scheduling problems used in this study.

As an additional point of comparison, we also include results obtained with three priority dispatch rules from the field of Operations Research: EDD, COVERT, and ATC (Vepsalainen & Morton 1987). These heuristics are frequently used and have been determined to work very well in job shop scheduling circumstances where expected job tardiness is low (as would likely be the case if a feasible solution exists).

The set of problems used in this study come from the dissertation of Sadeh (Sadeh 1991). The problem set consists of 60 randomly generated scheduling problems. Each problem contains 10 jobs and 5 resources. Each job has 5 operations. In all problems, deadlines were generated randomly within a specified range. A controlling parameter was used to generate problems in three different deadline ranges: wide (w), median (m), and tight (t). A second parameter was used to generate problems with both 1 and 2 "bottleneck" resources. Combining these two parameters, 6 different categories of scheduling problems were defined, and 10 problems were generated for each category. The problem categories were carefully defined to cover a variety of manufacturing scheduling circumstances. While each problem has at least one feasible solution, they range in difficulty from easy to hard.

The results obtained on these problems, along with those previously reported, are given in Table 1 (where problem difficulty increases from top to bottom). The number of problems solved by each approach by problem category are indicated. In the case of ORR/FSS

Incorporating Additional Search Bias

While **Min-Slack** performed quite well over the tested problem set, it does not in fact utilize all of the information provided by the temporal slack data. In particular, it relies exclusively on the smaller slack value in determining the criticality of a ordering decision $ordering_{i,j}$, and ignores any information that might be provided by the larger one.

The most common problem created by disregarding this additional value appears in a form of tie-breaking. Consider the following example. Suppose that we have two unsequenced operation pairs, one with associated temporal slack values of $(20, 3)$, and the other with values of $(4, 3)$. **Min-Slack** does not distinguish between the criticality of these two ordering decisions, since the minimum value in both cases is 3. In the event that the overall minimum slack over all candidate decisions is also 3, then **Min-Slack** will choose randomly. But, in this case sequencing the second operation pair is certainly more critical since the flexibility that will be left after the decision is made will be considerably less than the flexibility that will remain if the first unsequenced operation pair is instead chosen and sequenced.

Given this insight, we define a second variable ordering heuristic, which operates exactly as **Min-Slack** except in situations where more than one pending decision $ordering_{i,j}$ is identified as a decision with overall minimum temporal slack. In these situations, ties are broken by selecting the decision with the minimum larger temporal slack value. Applying the PCP procedure with this extended heuristic to the same suite of 60 problems yielded 57 solved problems. Although this

[2] All computation times were obtained on a Decstation 5000. Both ORR/FSS and CPS are Lisp-based systems; our procedure is implemented in C.

ues increases and decrease criticality as the slack values become more dissimilar might provide more effective search guidance.

Let us define a measure of similarity in the range [0, 1] such that for slack value pairs with identical values, the similarity value is 1 and as the distance between large and small slack values increases, the similarity value approaches 0. More precisely, we estimate the similarity between two slack values by the following ratio expression:

$$S = \frac{min\{slack(i \rightarrow j), slack(j \rightarrow i)\}}{max\{slack(i \rightarrow j), slack(j \rightarrow i)\}} \quad (5)$$

Given the definition of S and the direction of bias desired, we now define a new criticality metric, referred to as biased temporal slack, as follows:

$$Bslack(i \rightarrow j) = \frac{slack(i \rightarrow j)}{f(S)}, \quad (6)$$

where f is a monotonically increasing function.

With little intuition as to the appropriate level of bias to exert on the criticality calculation, but assuming that the level of bias should not be too great, we use $\sqrt[n]{S}, n \geq 2$, to define a set of alternatives, yielding

$$Bslack(i \rightarrow j) = \frac{slack(i \rightarrow j)}{\sqrt[n]{S}}. \quad (7)$$

By empirical reasoning, we also define a composite form of the metric with two different parameters, n_1 and n_2, as

$$Bslack(i \rightarrow j) = \frac{slack(i \rightarrow j)}{\sqrt[n_1]{S}} + \frac{slack(i \rightarrow j)}{\sqrt[n_2]{S}}. \quad (8)$$

Table 2 presents results obtained using **overall minimum Bslack** as a variable ordering criterion for different values of n in Eqn. (7) and n_1 and n_2 in Eqn. (8)

MinSlackHeuristic

MinSlackHeuristic - Notepad

File Edit Format View Help

```java
//
// Slack-Based Heuristics for Constraint Satisfaction Scheduling
// Stephen F. Smith and Cheng-Chung Cheng
// Proceedings AAAI-93
//
import org.chocosolver.solver.variables.IntVar;
import org.chocosolver.solver.search.strategy.selectors.variables.VariableSelector;

public class MinSlackHeuristic implements VariableSelector<IntVar> {

    int n;              // number of decision variables
    IntVar[] decision;  // 0/1 decision variables
    IntVar[] op1;       // start time of op1
    IntVar[] op2;       // start time of op2
    int[] duration1;    // duration of op1
    int[] duration2;    // duration of op2

    public MinSlackHeuristic(Decision[] decisions){
        n         = decisions.length;
        decision  = new IntVar[n];
        op1       = new IntVar[n];
        op2       = new IntVar[n];
        duration1 = new int[n];
        duration2 = new int[n];
        for (int i=0;i<n;i++){
            decision[i]  = decisions[i].d;
            op1[i]       = decisions[i].op_i.start;
            op2[i]       = decisions[i].op_j.start;
            duration1[i] = decisions[i].op_i.duration;
            duration2[i] = decisions[i].op_j.duration;
        }
    }

@Override
    public IntVar getVariable(IntVar[] v){
        int minSlack = Integer.MAX_VALUE;
        int slack = 0;
        IntVar minSlackVar = null;
        for (int i=0;i<n;i++)
            if (!decision[i].isInstantiated()){
                slack = slack(i);
                if (slack < minSlack){minSlackVar = decision[i]; minSlack = slack;}
            }
        return minSlackVar;
    }
    //
    // select the uninstantiated variable with minimum maximum slack
    //

    private int slack(IntVar op_i,int d_i,IntVar op_j,int d_j){
        return op_j.getUB() - Math.max(op_i.getLB() + d_i,op_j.getLB());
    }
    //
    // slack if op_i before op_j
    // i.e. slack(op_i -> op_j) in S&C AAAI-93 parlance
    //
    // NOTE: we consider earliest and latest start times whereas S&C
    //       consider earliest start and latest finish, but our calculations
    //       are exactly the same
    //

    private int slack(int i){
        return Math.max(slack(op1[i],duration1[i],op2[i],duration2[i]),
                        slack(op2[i],duration2[i],op1[i],duration1[i]));
    }
    //
    // get slack of ith decision variable, to be the largest slack
    // from either slack(op1 -> op2) or slack(op2 -> op1)
    // This differs from Smith & Cheng as they select the smaller of the two,
    // i.e. replace Math.max with Math.min
    //

}
```

Optimize

Optimize - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
import org.chocosolver.solver.Model;
import org.chocosolver.solver.Solver;
import org.chocosolver.solver.Solution;
import org.chocosolver.solver.variables.IntVar;
import org.chocosolver.solver.constraints.IIntConstraintFactory.*;
import org.chocosolver.solver.search.strategy.Search;
import org.chocosolver.solver.search.strategy.strategy.IntStrategy;
import org.chocosolver.solver.search.strategy.selectors.values.IntDomainMax;
import org.chocosolver.solver.exception.ContradictionException;

public class Optimize {

  public static void main(String[] args) throws FileNotFoundException, IOException {
      int timeLimit          = Integer.parseInt(args[1]);
      JSSP jssp              = new JSSP(args[0],9999);
      Model model            = jssp.model;
      Solver solver          = model.getSolver();
      Decision[] decisions   = jssp.getDecisions();
      int n                  = decisions.length;
      IntVar makeSpan        = jssp.getMakeSpan();

      solver.limitTime(timeLimit*1000);

      solver.setSearch(new IntStrategy(jssp.getDecisionIntVars(),new MinSlackHeuristic(jssp.getDecisions()),new IntDomainMax()));
      //
      // attach a variable & value ordering heuristic to solver
      //

      // alternative way to optimize
      // Solution solution = solver.findOptimalSolution(makeSpan,false);

      // trace optimisation
      model.setObjective(Model.MINIMIZE,makeSpan);
      while(solver.solve()) System.out.println("makespan: ["+ makeSpan.getLB() +","+ makeSpan.getUB() +"]" +
                                               "    nodes: "+ solver.getMeasures().getNodeCount());

      System.out.println("nodes: "+ solver.getMeasures().getNodeCount() +
                         "    cpu: "+ solver.getMeasures().getTimeCount());
  }
}
```

Anyway, will the variable ordering heuristic make a difference on it’s own? Admittedly the heuristic is rather expensive to run so will it reduce search effort to the point that it reduces run time (the bottom line)

Do some experiments

Table I. Lawrence jobshop scheduling instances, la01 to la15. Minimum makespan (2nd column) is posted as a constraint resulting in a decision problem. Tabulated is number of decisions (nodes), discrepancies taken, and run time in seconds. Slack-based dynamic variable and value ordering heuristics were used. Results are reported for chronological backtracking (BT). The best results between ILDS-early and ILDS-late are in **bold**.

		ILDS-early			ILDS-late			BT	
Instance	makespan	nodes	disc	time	nodes	disc	time	nodes	time
la01	666	42	0	0.05	42	0	0.05	42	0.04
la02	655	**2648**	3	0.43	5248	3	0.60	35132	1.7
la03	597	53552	6	4.1	**42345**	6	3.3	6103	0.67
la04	590	**1798**	3	0.38	2431	3	0.44	310	0.21
la05	593	91	0	0.06	91	0	0.06	91	0.05
la06	926	958	1	0.36	**306**	1	0.17	–	–
la07	890	**3660**	2	1.1	8024	2	2.5	1044950	41
la08	863	5794	1	1.6	**2409**	1	0.71	517990	20
la09	951	**760**	1	0.32	6616	1	1.6	–	–
la10	958	1045	1	0.34	**485**	1	0.20	39201106	1294
la11	1222	2090	1	2.0	**757**	1	0.79	–	–
la12	1039	36987	2	32	**22096**	2	19	–	–
la13	1150	**4117**	1	3.4	14669	1	9.4	–	–
la14	1292	**1352**	1	1.1	11142	1	6.2	399	0.13
la15	1207	111067002	4	6743	**7194189**	3	431	–	–

THEOREM 1. *The order of instantiation of variables can influence the number of probes required to find a solution.*

PROOF. We use an existence proof. Assume we have a problem with three con-

Table III. Norman Sadeh's job shop scheduling satisfaction problems. Columns E-*-* are ILDS-early. Columns L-*-* are ILDS-late. Columns *-1-* use slack-based value ordering. Columns *-0-* use the static value ordering, select 0 then select 1. Columns *-*-1 use the slack-based variable ordering. Columns *-*-0 select variables in index order. A table entry of - signifies a trivial instance solved with zero discrepancies (see Table II).

Instance	E-1-1	L-1-1	E-1-0	L-1-0	E-0-1	L-0-1	E-0-0	L-0-0
e0ddr1-1	-	-	788	1722	201	98	448	200
e0ddr1-2	-	-	-	-	267	150	3968	1427
e0ddr1-3	222	1282	4190	64855	606229	1563201	358049	438228
e0ddr1-4	-	-	222	2664	207	93	3015	2140
e0ddr1-5	-	-	-	-	-	-	743	445
e0ddr1-6	-	-	-	-	-	-	1824	2612
e0ddr1-7	-	-	231	1157	-	-	239	114
e0ddr1-8	220	801	-	-	-	-	697	158
e0ddr1-9	-	-	254	1383	-	-	698	123
e0ddr1-10	297	148	-	-	123	125	190	616
e0ddr2-1	-	-	8782	7359	1326	8997	456864	100969
e0ddr2-2	1175	1231	176616	567904	94	256	-	-
e0ddr2-3	-	-	-	-	96	96	-	-
e0ddr2-4	-	-	281	1246	-	-	174	306
e0ddr2-5	-	-	-	-	-	-	-	-
e0ddr2-6	-	-	243	3752	-	-	2386	2164
e0ddr2-7	-	-	6415	23015	-	-	1214	166
e0ddr2-8	197	224	-	-	120	76	-	-
e0ddr2-9	-	-	312	2004	5911	4226	56991	10976
e0ddr2-10	-	-	-	-	-	-	-	-
enddr1-1	-	-	-	-	-	-	9581	31246
enddr1-2	-	-	-	-	-	-	234	752
enddr1-3	-	-	-	-	1007	534	685	1458
enddr1-4	-	-	281	1694	-	-	39468	124717
enddr1-5	-	-	-	-	-	-	-	-
enddr1-6	-	-	-	-	-	-	1625	1143
enddr1-7	-	-	-	-	-	-	3179	2182
enddr1-8	161	445	308	2254	929	491	3488	8128
enddr1-9	-	-	-	-	-	-	351	212
enddr1-10	-	-	-	-	239	772	2521	13478
ewddr2-1	-	-	-	-	-	-	600	1162
ewddr2-2	-	-	247	4971	-	-	12025	3879
ewddr2-3	-	-	-	-	118	131	613	957
ewddr2-4	-	-	-	-	-	-	-	-
ewddr2-5	-	-	-	-	-	-	1946	254
ewddr2-6	-	-	655	4658	-	-	1097	211
ewddr2-7	-	-	227	4776	-	-	2882	2776
ewddr2-8	-	-	-	-	93	112	8800	2258
ewddr2-9	-	-	-	-	-	-	61419	569337
ewddr2-10	-	-	-	-	123	161	1397	661

Edge Finding

Edge-finding (EF) techniques are effective propagation techniques for resource capacity constraints that reason about the processing order of a set of activities requiring a given resource or set of resources. Some of the typical ordering relationships that EF techniques can determine are whether an activity can, cannot, or must execute before (or after) a set of activities requiring the same resource or set of resources. This in turn determines new time bounds on the start and finish times. Carlier and Pinson (1989) were responsible for some of the earliest work in this area that resulted in solving MT10, a 10×10 job shop problem that had remained unsolved for over 20 years (Muth and Thompson; 1963). Since then, there have been several variations and extensions of this work (Carlier and Pinson; 1990; Applegate and Cook; 1991; Nuijten; 1994; Baptiste and Le Pape; 1996).

The edge-finding consistency routines are invoked by specifying the EDGEFINDER= or EDGE= option in the SCHEDULE statement. Specifying EDGEFINDER=FIRST computes an upper bound on the activity finish time by detecting whether a given activity must be processed first from a set of activities requiring the same resource or set of resources. Specifying EDGEFINDER=LAST computes a lower bound on the activity start time by detecting whether a given activity must be processed last from a set of activities requiring the same resource or set of resources. Specifying EDGEFINDER=BOTH is equivalent to specifying both EDGEFINDER=FIRST and EDGEFINDER=LAST.

An extension of the edge-finding consistency routines is in determining whether an activity ***cannot*** be the first to be processed or whether an activity ***cannot*** be the last to be processed from a given set of activities requiring the same resource (set of resources). The NOTFIRST= or NF= option in the SCHEDULE statement determines whether an activity is "not first." In similar fashion, the NOTLAST= or NL= option in the SCHEDULE statement determines whether an activity is "not last."

Note: This procedure is experimental.