Iterative Improvement Algorithms for the Blocking Job Shop

Angelo Oddi*, Riccardo Rasconi*, Amedeo Cesta*, and Stephen F. Smith^

** ISTC-CNR, Rome, Italy* *name.surname@istc.cnr.it*

^ Robotics Institute, CMU, Pittsburgh, PA, USA *sfs@cs.cmu.edu*

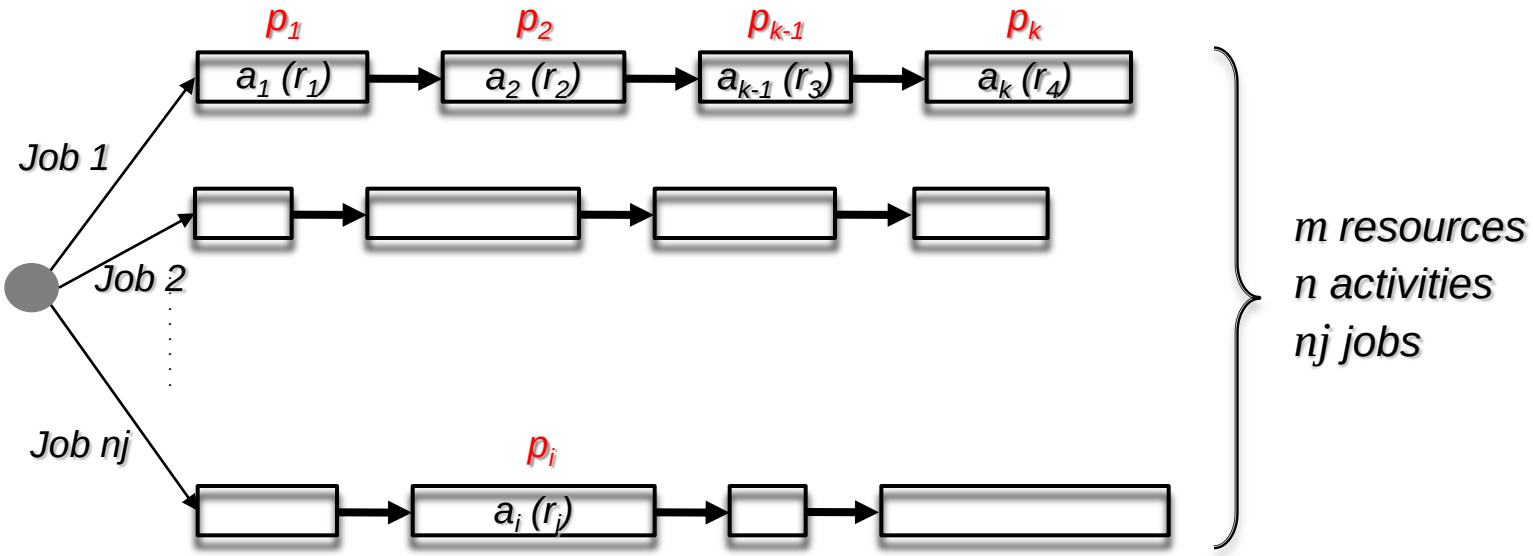
Planning & Scheduling Team

Summary

- *The BJSSP scheduling problem*
- *Modelling the BJSSP*
- *Iterative Flattening Search (IFS):*
  - *Relaxation procedures*
  - *Solving procedure*
- *Experimental evaluation*
- *Conclusions*

The scheduling problem

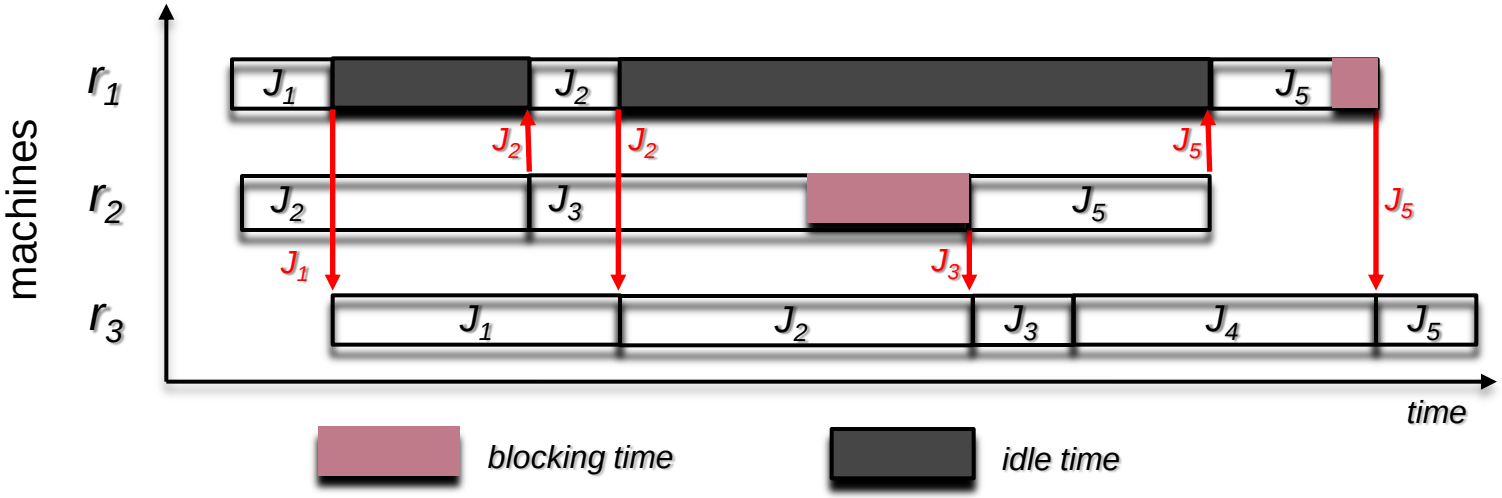
Let us start from a classical JSSP:



- *All jobs are released at time 0*
- *Each activity requires the exclusive use of a single resource*
- *Each solution imposes a total order on the subsets of activities requiring the same resource*

The scheduling problem

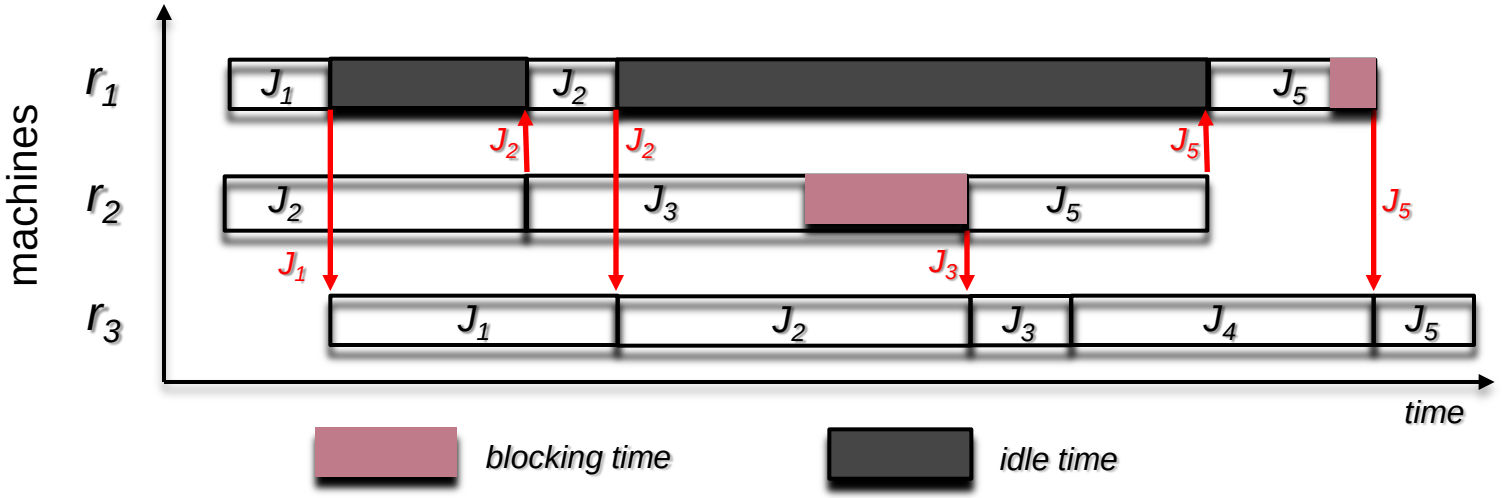
Why the Blocking JSSP differs:



- *No intermediate buffers for storing a job as it moves from one machine to another*
- *Each activity (except the last of each job) is a **blocking** activity that remains on the machine until its successor activity starts*
- *No pre-emption is allowed*

The scheduling problem

Solution: *a set $S = \{s_1, s_2, \ldots, s_n\}$ of assigned start times s_i that satisfy all previous constraints*



Objective: *let C_k be the completion time of job J_k, the goal is to minimize the makespan C_{max}*

$$C_{max} = \max_{i \le k \le nj} \{C_k\}$$

The IFS algorithm

```
iFlat (S, MaxFail)
    S* = S
    counter = 0
    while (counter <= MaxFail)
            Relax(S)
            Solve(S)
            if makespan(S) < makespan(S*)
              then S* = S
                   counter = 0
              else  counter = counter +1
    return(S*)
```

- *Solving procedure:*
  - *Precedence Constraint Posting Search (PCP)*
- *Chain-based relaxation:*
  - *Random*
  - *Slack-based*

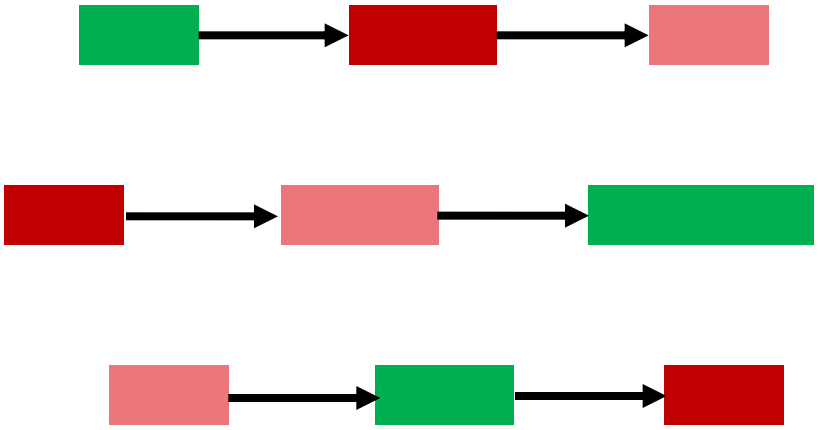
The IFS algorithm

```
iFlat (S, MaxFail)
    S* = S
    counter = 0
    while (counter <= MaxFail)
        Relax(S)
        Solve(S)
        if makespan(S) < makespan(S*)
          then
            S* = S
            counter = 0
          else  counter = counter +1
    return(S*)
```

CSP-based solving procedure

A CSP representation

*A BJSSP instance can be represented as a graph G(**A_G**, **J**, **X**) where:*

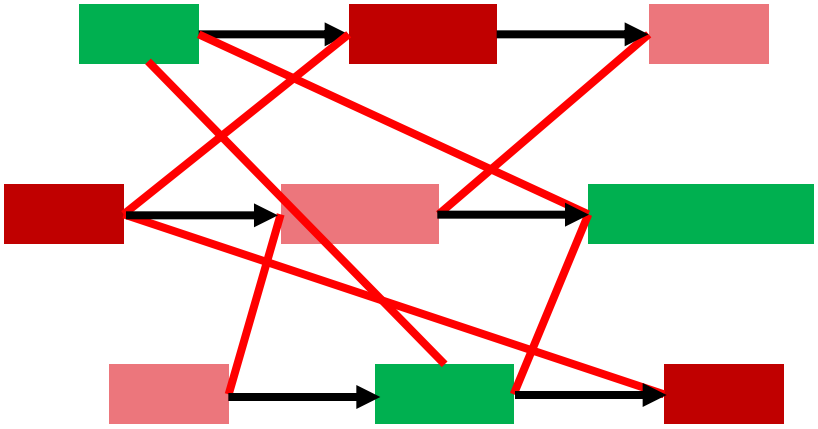


***J** is the set of **directed** edges (**a_i**, **a_j**) that represent the problem precedence constraints that exist among the activities*

Planning & Scheduling Team

A CSP representation

*A BJSSP instance can be represented as a graph G(**A_G**, **J**, **X**) where:*

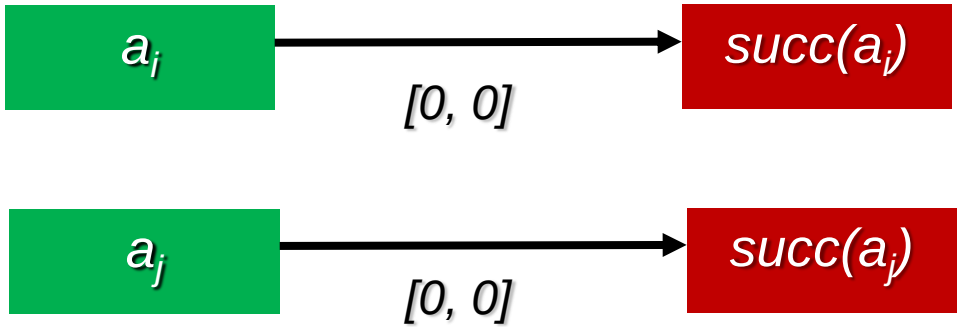


***J** is the set of **directed** edges (**a_i**, **a_j**) that represent the problem precedence constraints that exist among the activities*

X** is the set of **undirected** edges (**a_i**, **a_j**) that represent the disjunctive constraints among the activities that require the same resource, labeled with the two possible orderings between **a_i** and **a_j

A CSP representation

- *How we modelled the Blocking constraints:*



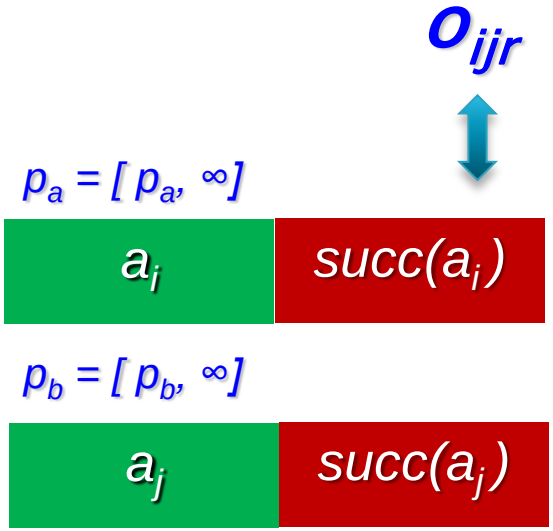
Additional BJSSP constraints:

- *for each activity a_i ($i = 1,\ldots, n_k-1$):*

$$e_i = s_{i+1}$$

A CSP representation

- *How we modelled the Blocking constraints:*



Additional BJSSP constraints:

- *for each activity a_i ($i = 1,\ldots, n_k\text{-}1$):*

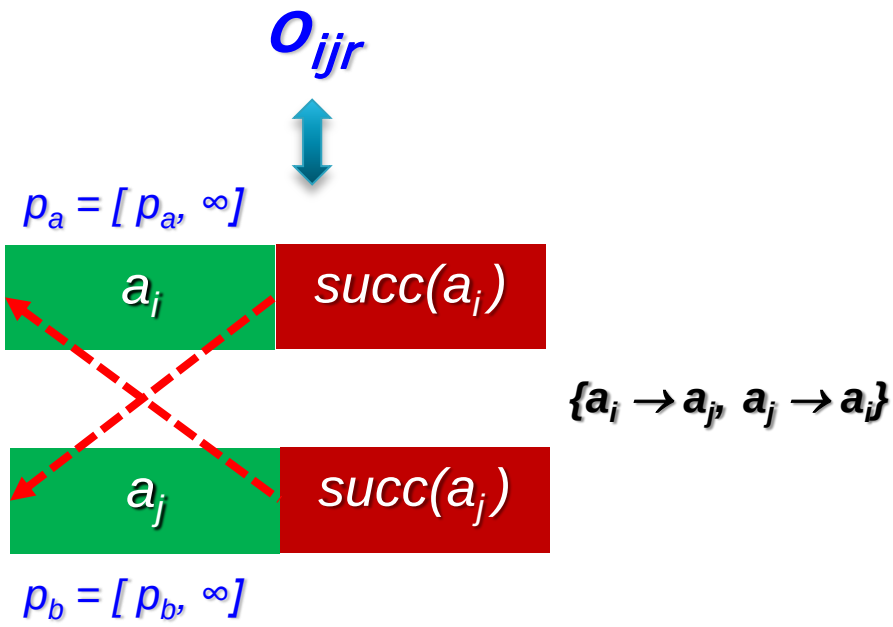
$$e_i = s_{i+1}$$

- *for each activity ai ($i = 1,\ldots, nk\text{-}1$):*

$$e_i - s_i \geq p_i$$

A CSP representation

- *In CSP terms, the following decision variables are introduced:*
  - *a variable* o_{ijr} *for each pair of activities* (a_i, a_j) *requiring the same resource* **r**, *whose domain is either* $a_j \rightarrow a_i$ *or* $a_i \rightarrow a_j$

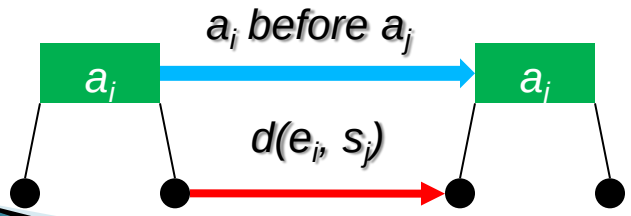


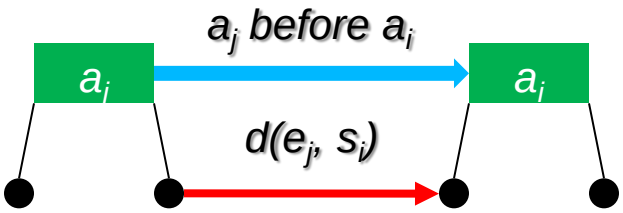
Planning & Scheduling Team

Dominance Conditions

On the basis of the temporal distance d(x,y) between the activity time points (STP representation [Dechter, Meiri and Pearl 1991]), the following propagation rules are applicable on $\boldsymbol{o_{ijr}}$*:*

1. *unsolvable conflict:* $d(e_i,s_j) < 0 \wedge d(e_j,s_i) < 0$
2. *solvable conflict:*
   a. $d(e_i,s_j) < 0 \wedge d(e_j,s_i) \geq 0 \wedge -d(s_i,e_j)$ ⟶ $\{ a_j \text{ before } a_i \}$
   b. $d(e_i,s_j) \geq 0 \wedge d(e_j,s_i) < 0 \wedge -d(s_i,e_j)$ ⟶ $\{ a_i \text{ before } a_j \}$
   c. $d(e_i,s_j) \geq 0 \wedge d(e_j,s_i) \geq 0$ ⟶ SEARCH DECISION!





A constraint-based algorithm

```
Solve(P)
    Propagate(P)
    if (one decision variable has an empty domain)
            then return(failure)
            else if (all decision variables are set)
                    then return(solution)
                    else
                        W ← Variable Ordering(P)
                        z ← Value Ordering(W)
                        set(z, W)
     Solve(P)
```

Heuristic Analysis

- *The search process is enhanced by defining proper **Variable** and **Value ordering Heuristics** based on shortest path temporal information*

- *Key ideas:*
  - ***Variable ordering** is performed according to the **Most Constrained First** principle*
  - ***Value ordering** is performed according to the **Least Constraining Value** principle*

Variable/Value ordering on o_{ijr}

- o_{ijr} *variable ordering*: *we select the best pair* (a^*_i, a^*_j) *representing the conflict with the minimum sequencing flexibility (i.e., the conflict closest to Dominance Condition 1)*

- o_{ijr} *value ordering*: *the ordering* $a^*_i \rightarrow a^*_j$ *or* $a^*_j \rightarrow a^*_i$ *that guarantees the highest amount of sequencing flexibility is selected:*

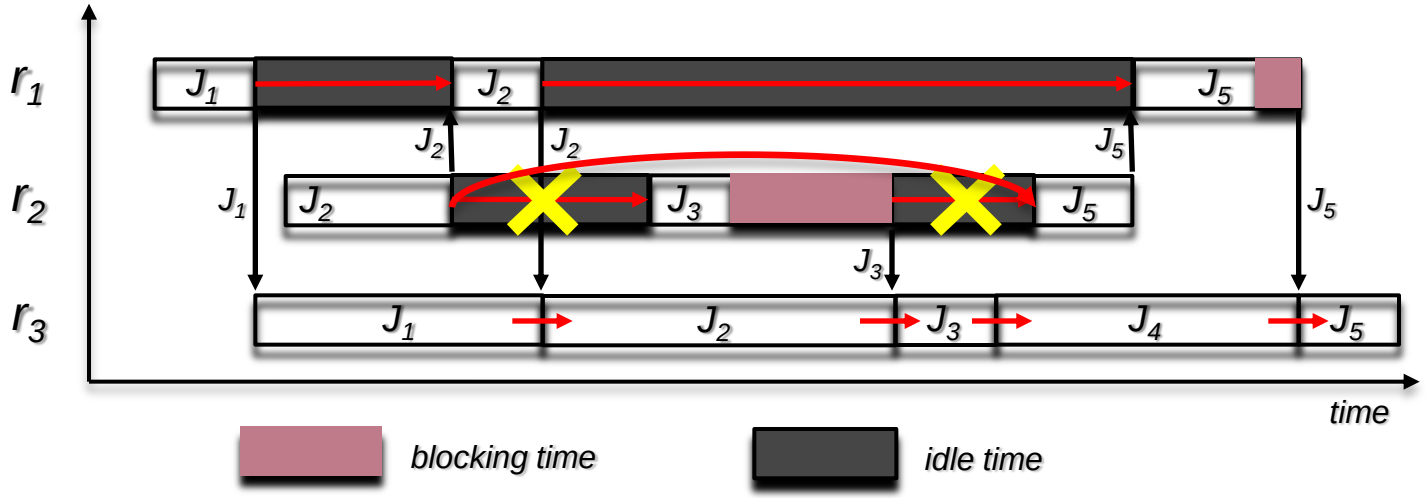
$$o_{ijr} = \begin{cases} a_i \text{ before } a_j & \text{if } d(e_i, s_j) > d(e_j, s_i) \\ a_j \text{ before } a_i & \text{otherwise} \end{cases}$$

The IFS algorithm

```
iFlat (S, MaxFail)
    S* = S
    counter = 0
    while (counter <= MaxFail)
        Relax(S)
        Solve(S)
        if makespan(S) < makespan(S*)
          then
            S* = S
            counter = 0
          else  counter = counter +1
    return(S*)
```

Chain-based Relaxation procedure

Chain-based relaxation

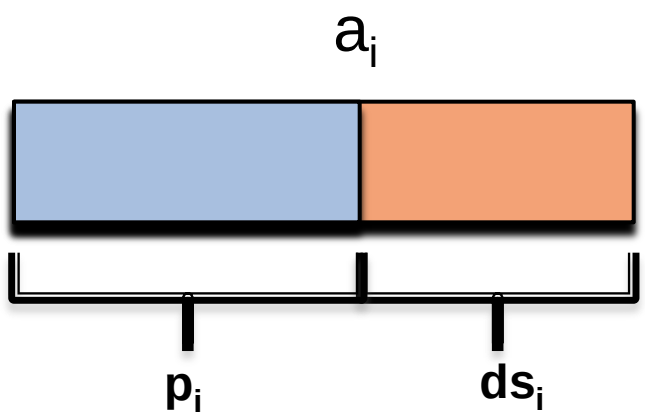


*Each **chain(r_i)** imposes a total order on a subset of problem activities requiring the same resource **r_i**. For each chain, activities are randomly removed with probability γ.*

Slack-based relaxation

- Duration slack (**ds**): how much can $\mathbf{a_i}$'s duration be extended without increasing the makespan?
- An activity $\mathbf{a_i}$ is on the critical path (CP) when $ds_i = 0$
- The smaller $\mathbf{ds_i}$ the *closest* $\mathbf{a_i}$ is to the CP
- Each activity $\mathbf{a_i}$ is removed with the following probability:

$$Pr_i = \gamma\,(1 - ds/ds_{max})$$



Empirical analysis

- *The empirical evaluation has been carried out on a set of **40** BJSSP benchmark instances from the standard **la01-la40** JSSP testbed proposed by Lawrence (Lawrence 1984)*
- *Each instance is loaded as a BJSSP as previously discussed*
- *The **40** instances are divided in sets of **5** instances each, which respectively contain **10x5**, **15x5**, **20x5**, **10x10**, **15x10**, **20x10**, **30x10** and **15x15** activities*
- *Current bests [Groeflin and Klinkert 2009; Groeflin, Pham and Burgy 2011]*
- *A time limit for each run was set to **1800** secs*
- *The algorithm has been implemented in Java and run on a AMD Phenom II X4 Quad 3.5 Ghz under Linux Ubuntu 10.4.1*

Random relaxation Vs Slack-based relaxation

Random relaxation

inst.	*best*	γ							
		0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8
	Av.C.	3K	2K	2K	2K	2K	2K	2K	1K
# impr.	**35**	**0**	**2**	**4**	**13**	**4**	**5**	**4**	**6**

# of improved solutions w.r.t. the current bests

Slack-based relaxation

inst.	*best*	γ							
		0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
	Av.C.	3K	2K	2K	2K	2K	2K	2K	2K
# impr.	**34**	**1**	**3**	**2**	**22**	**7**	**7**	**7**	**6**

CP-OPT * Performances

inst.	*best*	*cp*	*ifs*	*inst.*	*best*	*cp*	*ifs*	*inst.*	*best*	*cp*	*ifs*
la01	820	**793**	**793**	*la15*	1630	1571	**1527**	*la29*	1990	**1898**	1963
la02	793	815	**793**	*la16*	1142	1150	**1084**	*la30*	2097	2147	**2095**
la03	740	790	**715**	*la17*	977	996	**930**	*la31*	3137	**2921**	3078
la04	764	784	**743**	*la18*	1078	1135	**1026**	*la32*	3316	**3237**	3336
la05	666	**664**	**664**	*la19*	1093	1108	**1043**	*la33*	3061	**2844**	3147
la06	1180	1131	**1064**	*la20*	1154	1119	**1074**	*la34*	3146	**2848**	3125
la07	1084	1106	**1038**	*la21*	1545	1579	**1521**	*la35*	3171	**2923**	3148
la08	1125	1129	**1062**	*la22*	1458	**1379**	1425	*la36*	1919	1952	**1793**
la09	1223	1267	**1185**	*la23*	1570	**1497**	1531	*la37*	2029	**1952**	1983
la10	1203	1168	**1110**	*la24*	1546	1523	**1498**	*la38*	1828	1880	**1708**
la11	1584	1520	**1466**	*la25*	1499	1561	**1424**	*la39*	1882	1813	**1783**
la12	1391	1308	**1272**	*la26*	2125	**2035**	2045	*la40*	1925	1928	**1777**
la13	1541	1528	**1465**	*la27*	2175	2155	**2104**				
la14	1620	**1506**	1548	*la28*	2071	2062	**2027**				

- *CP-OPT obtains **25** improvements w.r.t. current bests*
- *CP-OPT obtains **11** improvements w.r.t. IFS*

*Available at
http://www-01.ibm.com/software/websphere/ilog/

Conclusions

- *In this paper we have proposed the use of Iterative Flattening Search (IFS) as a means of effectively solving BJSSP instances*
- *The proposed IFS algorithm uses a core constraint-based solving procedure which utilizes a set of propagation rules (dominance conditions) as well as a Random and Slack-based relaxation strategy*
- *The performance of the procedure has been tested on a known JSSP benchmark set (Lawrence 1984), where each JSSP instance is loaded modelling BJSSP constraints*
- *Both relaxation strategies have been tested against the best results in literature, exhibiting a significant performance improvement*
- *Very good results have also been shown using the ILOG CP-OPT solver, especially for the largest benchmark instances, where IFS is outperformed*