Progressive Solutions: A Simple but Efficient Dominance Rule for Practical RCPSP

András Kovács[1,2] and József Váncza[1]

[1]Computer and Automation Research Institute,
Hungarian Academy of Sciences
[2] Cork Constraint Computation Centre,
University College Cork, Ireland
{akovacs,vancza}@sztaki.hu

Abstract. This paper addresses the solution of practical resource-constrained project scheduling problems (RCPSP). We point out that such problems often contain many, in a sense similar projects, and this characteristic can be exploited well to improve the performance of current constraint-based solvers on these problems. For that purpose, we define the straightforward but generic notion of progressive solution, in which the order of corresponding tasks of similar projects is deduced a priori. We prove that the search space can be reduced to progressive solutions. Computational experiments on two different sets of industrial problem instances are also presented.

1 Introduction

The practical value of constraint-based scheduling hinges both on the representation power of the models and the efficiency of the solution techniques. Solution performance, in turn, depends on whether the solver can recognize and take advantage of the structural properties of the problem at hand.

Generic models, though different (like flow shop, job shop, resource-constrained project scheduling, etc.), hide some eventual structural properties of specific real-life problem instances. Making such properties explicit by adding extra features to the model is an option, but it comes together also with specialized solution techniques. Just to the contrary, in this paper we suggest to detect and exploit some hidden structural properties within the boundaries of a generic model. We introduce the simple notion of *progressive pairs* to characterize similar patterns of activities of a scheduling problem. Similarity will be defined both in terms of temporal relations and resource requirements. Typically, progressive pairs are inherent in practical discrete manufacturing problems where products or components of similar/same type are produced in parallel, by using the same technology and a common pool of resources.

We take the classical model of resource-constrained project scheduling problem (RCPSP) [3], and demonstrate our approach on the objective of minimizing

the makespan.[1] When detecting and exploiting progressive pairs, we rely on no extra domain-specific information.

The results presented here are based on our previous works that suggested the application of *consistency preserving* transformations to exploit some structural properties of constraint programs [7]. Earlier experiments with a combination of symmetry breaking techniques and so-called freely completable solutions convinced us that the performance of generic constraint-based methods can considerably be improved on *practical* problem instances. Now we take a more general approach to rule out dominated solutions by constraints added *before* the search process.

In the sequel we give an overview of relevant works related to symmetry breaking and the application of dominance rules. Following the definition of the RCPSP model, Sect. 4 presents the idea of progressive solutions together with the basic definitions, theorems and proofs. Sect. 5 describes how we detect this structural property among the projects in an RCPSP instance, while Sect. 6 summarizes the results of our experiments on two industrial data sets. Finally, conclusions are drawn.

2 Related Work

Recently, considerable efforts have been made to explore various classes of consistency preserving transformations in constraint programming. These transformations reduce the search space while ensuring that at least one (optimal) solution remains, if the original problem was solvable. Hence, they essentially extend the traditional toolbox of constraint programmers that mostly consists of equivalence preserving transformations. Such transformations – like constraint propagation or shaving – guarantee that the original and the transformed problems have exactly the same set of solutions.

The most intensively studied branch of consistency preserving transformations is doubtlessly *symmetry breaking*. *Symmetry* is a bijective function f defined on the bindings of the variables such that for each variable binding α, $f(\alpha)$ is a solution iff α is a solution, too. Breaking this symmetry means excluding all but one of the symmetric equivalents. The foremost of all symmetry breaking techniques is the addition of symmetry breaking constraints to the model *before search*. More sophisticated methods, such as the Symmetry Breaking During Search (also called Symmetry Excluding Search) and the Symmetry Breaking via Dominance Detection prune symmetric branches of the search tree *during search*. All of these general frameworks require an explicit declaration of the symmetries in the form of symmetry functions or a dominance checker. See [11] for a recent overview of symmetry breaking techniques.

[1] Note that while this objective is often criticized by practitioners, it really helps to squeeze a given amount of work into a pre-defined time frame. In a hierarchical production planning and scheduling setting, where the primary goal of scheduling is to generate an executable solution that complies with a segment of the production plan, makespan is a useful criterion [13].

A wider class of consistency preserving transformations is constituted by the *dominance rules*. They define properties of a problem that must be satisfied by at least one of its (optimal) solutions. By now, little work has been done to apply dominance rules in constraint programming. The recent paper [12] calls the attention to the application of dominance rules and defines novel dominance rules for three different problems.

At the same time, dominance rules are widely used in operations research and project scheduling. For instance, dominance rules of different strength and computational complexity are described for RCPSP with the criteria of minimal makespan in [3]. Dominance rules, as well as methods for the insertion of redundant precedence constraints are proposed for the problem of minimizing the number of late jobs on a single machine in [2].

3 Notations

Below, we define progressive solutions for resource-constrained project scheduling problems with the criterion of minimizing makespan, and give an outlook on possible extensions at the end of the paper. Hence, let T denote a set of non-preemptive *tasks*. Each task $t \in T$ has a fixed *duration* d_t, and requires ρ_t^r units of each renewable cumulative *resource* $r \in R$ during the whole length of its execution. The number of available units of resource r at a time, i.e., the *capacity* of r is denoted by q_r. Tasks can be connected by end-to-start *precedence constraints* $(t_1 \rightarrow t_2)$ that state that task t_1 must end before the start of task t_2. We assume that there is no directed circle in the graph of precedences.

Then, the objective is to find non-negative start times $start_t$ for the task $t \in T$, such that all precedence and resource constraints are satisfied, and the *makespan*, i.e., the maximum of the end times $end_t = start_t + d_t$ is minimal.

Although they are not allowed in the original problem definition, we will use the notion of *start-to-start precedences* as well, denoted by $(t_1 \dashrightarrow t_2)$, meaning that $start_{t_1} \leq start_{t_2}$. For brevity, we call the maximal sets of tasks connected by precedence constraints *projects*.

4 Progressive Solutions of Scheduling Problems

Factories often produce several pieces of the same product, or products belonging to the same product family during their short-term scheduling horizon. As a consequence, their detailed scheduling problems may include many, in a sense similar projects. This chapter is devoted to show that in such cases, a valid ordering of tasks belonging to similar projects can be deduced by off-line inference. These investigations will allow us to insert precedence constraints in the constraint-based model of the scheduling problem *before search*, and hence, to reduce the search space.

4.1 The Underlying Idea

As a simple example, suppose that two identical projects, P and Q are to be executed within the scheduling horizon (besides arbitrary other projects). By identical, we mean that for each index i, tasks p_i and q_i in Fig. 1.a. (and in all subsequent figures) have equal durations and resource requirements. Now, it is easy to see that if there exists a solution to this scheduling problem, then there exists one in which each task of P precedes its corresponding task in Q.

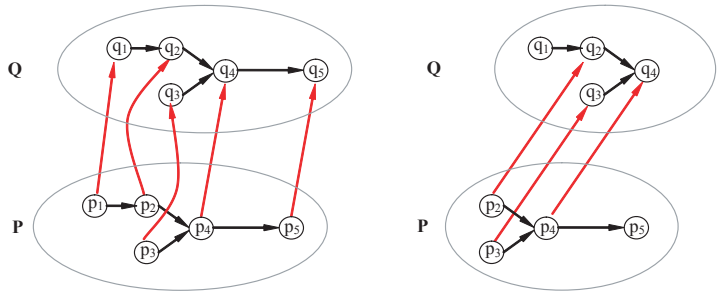

Fig. 1. a.) and b.) Examples of similar projects in the scheduling problem.

Now, assume that some tasks belonging to project P have already been executed before the start of the current scheduling horizon, hence, there is no match of q_1 in P. Similarly, some tasks of Q suffice to be done after the end of the horizon, resulting in no corresponding task for p_5 in Q. Again, tasks of P can precede tasks of Q, see Fig. 1.b. The third example in Fig. 2.a. depicts a case where P and Q are different members of the same product family. P requires an additional finishing operation (p_5) that has no match in Q, while there is an extra component built in Q (q_6, q_7, q_8). Finally, Fig. 2.b. has a theoretical significance, since it shows an example where the inferred precedence constraints form directed circles between projects, but not between tasks.

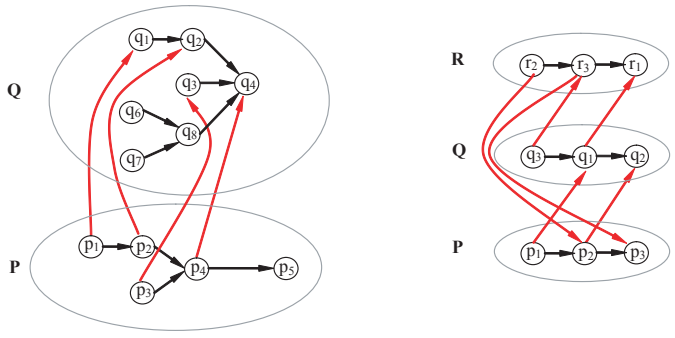

Fig. 2. a.) and b.) Two more examples of similar projects in the scheduling problem.

Below, we formally define our notion of similarity between projects and present how all this makes possible the reduction of the solution space.

4.2 Progressive Solutions

Definition 1 *Two sets of tasks P and Q are defined isomorphic, and will be denoted by $P \equiv Q$, iff there exists a bijection $\beta : P \leftrightarrow Q$ such that for each pair of tasks $p \in P$ and $q \in Q$*

$$\begin{aligned} &\beta(p,q) &&\Rightarrow\ \forall r \in R : \rho_p^r = \rho_q^r \ \wedge\ d_p = d_q, \text{ and} \\ &\beta(p_1,q_1) \wedge \beta(p_2,q_2) &&\Rightarrow\ (p_1 \rightarrow p_2) \Leftrightarrow (q_1 \rightarrow q_2). \end{aligned}$$

Definition 2 *Given two projects P and Q, we call them a progressive pair iff there exists a $P^* \subseteq P$ and a $Q^* \subseteq Q$ such that $P^* \equiv Q^*$, and there are no incoming precedences to P^* and no outgoing precedences from Q^*. This relation will be denoted by $P \rightrightarrows Q$ (see Fig. 3).*

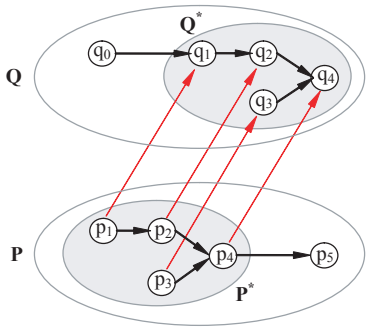

Fig. 3. The progressive pair $P \rightrightarrows Q$.

Furthermore, to avoid the ambiguous situations where $P \rightrightarrows Q$ and $P \leftrightarrows Q$ hold simultaneously for two isomorphic projects $P \equiv Q$, we label the projects by unique identifiers $L(.)$. Now, we say that two isomorphic projects constitute a progressive pair $P \rightrightarrows Q$ only if $L(P) < L(Q)$.

Definition 3 *A solution of a scheduling problem is called progressive, iff for each progressive pair $P \rightrightarrows Q$, the execution of P precedes Q, in the formal sense that for each pair of tasks $p \in P^*$ and $q \in Q^*$ such that $\beta(p,q)$, $p \dashrightarrow q$ holds. We will refer to this type of start-to-start precedence constraints as* progressive constraints.

Note that if at least one of the resources required by p and q is unary, then $(p \dashrightarrow q) \Leftrightarrow (p \rightarrow q)$.

Theorem 1 *If an RCPSP problem has a solution, then it also has a progressive solution with minimal makespan.*

We start the proof by the following simple lemma.

Lemma 1 *Given an RCPSP problem with no directed circles of precedence constraints, the insertion of progressive precedence constraints does not create a directed circle of (end-to-start and start-to-start) precedences between the tasks.*

Proof: Let us label the tasks $t \in T$ by $l(t) = |Pred(t)| - |Succ(t)|$, where $Pred(t)$ and $Succ(t)$ are the sets of predecessors and successors (direct and indirect) of t in the original problem, respectively. Notice that $l(t_1) < l(t_2)$ holds for all precedences $(t_1 \rightarrow t_2)$ in the original problem, and $l(t_1) \leq l(t_2)$ for all the inserted progressive constraints $(t_1 \dashrightarrow t_2)$.

Now, let us assume that there is a directed circle of precedences C. According to the above, C consists of progressive constraints only, with $l(t_1) = l(t_2)$. Then, by the definition of the progressive pairs, $Pred(t_1) \equiv Pred(t_2)$ and $Succ(t_1) \equiv Succ(t_2)$. This also implies that all the projects traversed by C are isomorphic. It is a contradiction, because by definition $L(P) < L(Q)$ must hold for each subsequent pair of projects P and Q traversed by C. □

Now, we prove Theorem 1 by an algorithm that departs from an arbitrary optimal solution, and through iteratively swapping pairs of tasks, generates a progressive solution with the same makespan. In each step of the algorithm, a progressive pair $P \rightrightarrows Q$ is selected, such that some of the progressive constraints between P and Q are violated in the actual schedule S. Then, the algorithm computes a modified schedule S' by swapping all the pairs of tasks in P and Q which violate the progressive constraints as follows.

$$\forall p \in P, q \in Q : \beta(p,q) \wedge start_p^S > start_q^S \quad \Rightarrow \quad \begin{array}{l} start_p^{S'} = start_q^S, \text{ and} \\ start_q^{S'} = start_p^S. \end{array}$$

For all other tasks $t \in T$, $start_t^{S'} = start_t^S$.

Lemma 2 *S' is feasible, and its makespan equals the makespan of S.*

Proof: All resource capacity constraints are satisfied in S', because only pairs of tasks with equal durations and resource requirements were swapped. In order to show that precedence constraints $p_1 \rightarrow p_2$, where $p_1, p_2 \in P^*$, cannot be violated in S' either, we introduce q_1 and q_2 to denote the two tasks in Q for which $\beta(p_1, q_1)$ and $\beta(p_2, q_2)$ hold. Then,

- if neither the pair (p_1, q_1), nor (p_2, q_2) were swapped, then the start times of p_1 and p_2 are unchanged in S' w.r.t. S, and S is feasible;
- If the pair (p_1, q_1) was swapped, but (p_2, q_2) not, then $end_{p_1}^{S'} = end_{q_1}^S < end_{p_1}^S \leq start_{p_2}^S = start_{p_2}^{S'}$;
- If the pair (p_2, q_2) was swapped, but (p_1, q_1) not, then $end_{p_1}^{S'} = end_{p_1}^S \leq end_{q_1}^S \leq start_{q_2}^S = start_{p_2}^{S'}$;

– If both (p_1, q_1) and (p_2, q_2) were swapped, then
$end^{S'}_{p_1} = end^{S}_{q_1} \leq start^{S}_{q_2} = start^{S'}_{p_2}$.

Precedence constraints pointing from P^* to $P \setminus P^*$ and those within $P \setminus P^*$ are also satisfied, because only tasks of P^* were moved earlier, and tasks of Q^* later in the schedule. The proof is analogous for precedence constraints in Q, and trivial for the precedence constraints between tasks of $T \setminus (P \cup Q)$, because those tasks were not moved. □

The above step is iterated until there are no more progressive constraints violated.

Proof of Theorem 1: The algorithm halts when it has found a progressive schedule. According to Lemma 2, this schedule is feasible, and has an optimal makespan. Furthermore, this is reached in finitely many steps, because the algorithm performs a brick sort over the tasks, according to the partial ordering defined by the progressive constraints. □

5 Computing the Progressive Pairs

Computing the progressive pairs in essence requires a pairwise comparison of the projects, and checking whether they have appropriate isomorphic subsets of tasks. The computational efficiency of these algorithms is of special importance, because no polynomial-time algorithm is known for deciding whether two general graphs are isomorphic [5]. Furthermore, we did not find a generic way to decrease the number of necessary isomorphism tests below $O(n^2)$.

Despite all this, our experiments suggest that in practical cases, PPs can be computed fast enough. On the one hand, efficient graph isomorphism algorithms are known for many classes of graphs, including trees [1, pp. 84–86], planar graphs, and graphs of bounded valence [4, 5, 8]. The algorithms also return a matching. In fact, the graph structures we encountered in our recent industrial applications belonged to the class of in-trees, in the most general case. On the other hand, often some kind of meta-knowledge about the projects (drawing numbers of parts and assemblies, product family codes, etc.) can be exploited, too.

In our pilot system, we assumed that the precedence graphs of projects form in-trees. For each pair of projects P and Q, and for each task $p \in P$, we took the sub-tree P^p of the precedence tree of P rooted at the node corresponding to p. We checked if P^p is isomorphic with a sub-tree of Q also containing the root of Q. Clearly, a positive answer of the isomorphism test means that a progressive pair $P \rightrightarrows Q$ has been found with $P^* \equiv P^p$ (see procedures `ComputeProgressivePairs` and `CheckIfProgressive` in Fig. 4.). We implemented a simple depth-first search to perform the isomorphism tests (procedure `TryMatching`). Note that `TryMatching` considers only the precedence constraints present in the original problem formulation, but not the progressive constraints inserted beforehand.

Although this algorithm has an exponential worst-case complexity, it proved efficient enough for even the largest practical instances we tackled (see the next section for details). This was possible because the vast majority of the isomorphism tests could return false immediately, due to the different durations or resource requirements of tasks in the roots of the examined sub-trees.

```
1 PROCEDURE ComputeProgressivePairs()
2   FORALL project P
3     FORALL project Q : Q ≠ P
4       CheckIfProgressive(P, Q)

5 PROCEDURE CheckIfProgressive(P, Q)
6   q := last task of Q
7   FORALL task p ∈ P, ordered by the increasing distance of p
-          from the root of P
8     IF NOT ((p is the last task in P) AND (L(P) > L(Q))) THEN
9       IF p ≡ q THEN
10        M := TryMatching(P^p, Q, {< p, q >})
11        IF M ≠ ∅ THEN
12          Add progressive pair P ⇉ Q with matching M
13          RETURN

14 PROCEDURE TryMatching(P, Q, M)
15  p, p0 := Select a pair of task from P such that (p → p0),
-           p has no match in M, but p0 has a match in M
16  IF there is no such p, p0
17    RETURN M
18  q0 := Match of p0 in M
19  FORALL q : (q → q0) AND q has no match in M
20    IF p ≡ q THEN
21      M' := TryMatching(P, Q, M ∪ {< p, q >})
22      IF M' ≠ ∅
23        RETURN M'
24  RETURN ∅
```

Fig. 4. An algorithm for computing the progressive pairs when projects form in-trees.

6 Experiments

We performed computational experiments on two different sets of industrial problems with two purposes. First, to estimate to what extent the tasks in a scheduling problem can be ordered by off-line inference, and second, to measure how much the inferred precedence constraints can speed up the solution process.

Our first set of data derives from an industrial partner that manufactures mechanical parts of high complexity for the energy industry. Their products can be ordered into four product families. Members of the same family share a similar structure, but differ in various parameters. The overall number of different end products is ca. 40, but this number may grow in the future. A project, aimed at the fabrication of one end product, consists of up to a few hundred tasks. Since the bill of materials of the products are tree-structured, the precedence relations within a project also form an in-tree. Tasks require one unit of a machine resource (unary or cumulative) and one unit of a human resource (cumulative) for their execution. There are altogether ca. 100 different resources in the plant. For more details on this scheduling problem, the readers are referred to [6].

The other set of problem instanced originates from the ILOG MascLib library [10]. MascLib contains industrial and generated benchmarks classified according to the complexity of the scheduling model. For our experiments, we used the *No-Calendar General Shop* (NCGS) problem class. Although the authors of the library suggest the usage of more realistic criteria – combinations of non-performance, earliness/tardiness, setup and mode costs – we simplified these instances to standard job-shop problems and minimized makespan. Therefore, we disregarded the due times of the tasks and the option of not performing them, while preserving their durations, resource requirements, and the precedence relations. The library contains 26 NCGS instances, but 13 of them differ only in the tardiness and non-performance costs from the others, and 7 others do not contain progressive pairs at all – we believe these were the generated instances. Hence, we performed experiments on the 6 remaining instances.[2]

As the first step of the experiments, we detected the progressive pairs in the problem instances. The results are presented in Table 1. The first group of rows stands for the instances from the industrial partner, while the second group for the NCGS instances. Columns *Tasks*, *Projects*, and *Resources* give information about the size of the problem instance, while column *PPs* indicate the number of progressive pairs of projects found. *EtS* and *StS* displays the number of inferred end-to-start and start-to-start progressive precedence constraints. The last two columns contain the *order strength (OS)* [9] without (OS^-) and with (OS^+) the inferred constraints. OS was calculated as the number of precedence constraints within one resource, divided by the number of task pairs competing for a resource, i.e.,

$$OS = \frac{\sum_{r \in R} |Prec_r|}{\sum_{r \in R} \frac{|T_r|(|T_r|-1)}{2}},$$

[2] We also experimented with a third set of data that came from the automotive industry. These were job-shop problems with ca. 50 unary resources and hundreds of projects, each containing at most 6 sequentially ordered tasks. Progressive pairs could be found in these instances as well, but the resource loads were so unbalanced that it was easy to find optimal solutions even without the progressive constraints.

where $T_r = |t \in T : \ \rho_t^r \geq 1|$ and $Prec_r = \{(t_1 \rightarrow t_2) \ or \ (t_1 \dashrightarrow t_2) : \ t_1, t_2 \in T_r\}$.[3] Hence, OS is 0 when there are no ordering constraints within the resources, and 1 if the tasks are completely ordered. The results show that in all the instances, the inferred progressive constraints could considerably reduce the search space. In the case of the NCGS instances, all the inserted precedence constraints were of the end-to-start type, since these problems contained unary machines only. The time needed to find the progressive pairs did not exceed 1 second even for the largest problem instances.

	Tasks	Projects	Resources	PPs	EtS	StS	OS^-	OS^+
p1	3511	97	95	308	17605	864	0.011	0.090
p2	2767	80	95	242	10674	641	0.014	0.093
p3	1470	70	95	132	2729	273	0.030	0.105
p4	1753	80	95	148	2833	225	0.025	0.077
p5	2472	89	95	196	3389	339	0.017	0.050
p6	2570	91	95	181	3653	323	0.019	0.058
p7	1133	70	95	134	1495	212	0.068	0.227
p8	769	68	95	122	293	248	0.052	0.084
p9	1620	85	95	160	2723	299	0.027	0.094
p10	1677	71	95	156	491	348	0.024	0.033
p11	1471	69	95	129	337	277	0.026	0.034
p12	585	71	95	143	232	221	0.032	0.069
p13	1786	83	95	187	2918	353	0.024	0.082
p14	1240	72	95	220	1570	230	0.067	0.201
p15	947	45	95	92	1223	97	0.088	0.269
NCGS_21	60	16	5	39	147	-	0.000	0.377
NCGS_31	75	19	5	42	162	-	0.000	0.277
NCGS_54	260	45	10	476	1783	-	0.007	0.399
NCGS_55	260	45	10	588	1921	-	0.007	0.427
NCGS_75	1250	41	30	40	1600	-	0.042	0.128
NCGS_81	2500	72	30	302	12210	-	0.022	0.184

Table 1. Order strength without and with progressive constraints.

In order to measure the effect of the inferred ordering decisions on algorithm performance, we fed these instances into ILOG Scheduler 5.1. We used ILOG's default *branch-and-bound* search with the *setting times* branching strategy and the *edge-finding* algorithm for the propagation of resource constraints. For all instances, the solution process was stopped when the optimality of a solution was proven, or after 600 seconds passed without improvement. In the latter case, we computed a lower bound by pure constraint propagation. The tests were run on a 1.6 GHz Pentium IV computer under Windows 2000 operating system.

The results achieved without and with the presence of the progressive constraints are shown in Table 2, where each row stands for one problem instance.

[3] Including all the edges in the transitive closure of the precedence graph.

UB and LB stand for the best found upper and lower bounds, respectively. *Error* was calculated as $(UB - LB)/LB$, while '-' denotes optimality. Values displayed in columns *Nodes* and *Time* were measured only until the best solution was found. The solver often generated significantly more search nodes until the timeout, but displaying those figures in the table would not be informative.

The figures show that progressive constraints facilitated both finding better solutions and proving tighter lower bounds. While improved lower bounds and better pruning is an evident outcome of a tighter formulation, the presence of the progressive constraints also had a positive effect on the branching heuristic. This is clearly shown by better first solutions for 10 of the 21 problem instances. All in all, the addition of the progressive constraints decreased the gap between the solutions found and the lower bounds by 60% on average, and made possible finding optimal solutions for two previously unsolvable instances (p5 and NCGS_31).

At the same time, the presence of progressive constraints had a negative impact on the solution process in the case of two instances: p9, where equivalent solutions were found, but with less search without the progressive constraints,

	Tasks	Without PP					With PP				
		UB	LB	Error (%)	Nodes	Time (sec)	UB	LB	Error (%)	Nodes	Time (sec)
p1	3511	372	341	9.09	3511	651	345	342	0.88	3511	646
p2	2767	289	247	17.00	2767	29	251	248	1.21	2767	29
p3	1470	276	276	-	1470	11	276	276	-	1470	15
p4	1753	252	252	-	19719	1891	252	252	-	7347	553
p5	2472	290	276	5.07	2472	21	276	276	-	2472	36
p6	2570	269	230	16.96	2570	25	254	230	10.43	2570	23
p7	1133	254	254	-	5686	148	254	254	-	1133	8
p8	769	264	264	-	2309	31	264	264	-	770	2
p9	1620	343	334	2.69	1620	9	343	334	2.69	3240	328
p10	1677	304	284	7.04	1677	9	295	284	3.87	6924	406
p11	1471	320	307	4.23	3450	286	310	307	0.98	4469	273
p12	585	358	349	2.58	585	1	356	349	2.01	585	1
p13	1786	370	366	1.09	4073	483	373	366	1.91	5409	475
p14	1240	383	373	2.68	5504	147	376	373	0.80	4885	541
p15	947	233	222	4.95	5232	107	232	222	4.50	3267	77
NCGS_21	60	2872	2799	2.61	9304	0	2872	2854	0.63	13469	0
NCGS_31	75	3412	3339	2.19	9552	0	3348	3348	-	10235	1
NCGS_54	260	1105	1105	-	260	0	1105	1105	-	260	0
NCGS_55	260	975	975	-	260	0	975	975	-	260	0
NCGS_75	1250	1164	1028	13.23	1200404	1310	1122	1044	7.47	17649	82
NCGS_81	2500	2220	1902	16.72	5956417	20479	2014	1902	5.89	5001	234

Table 2. Effect on algorithm performance.

and p13, where better solution could be constructed without them.[4] This effect is caused by the adverse interaction of the inserted constraints with the search strategy, an unfavorable phenomenon well known from the literature of symmetry breaking [11].

7 Conclusions

In this work we focused on the solution efficiency of constraint-based scheduling on practical RCPSP instances. For that purpose, we suggested a method for detecting progressive pairs, and transforming the original problem into a tighter constrained formulation by the application of a novel dominance rule. It was proven that the proposed transformation preserves the consistency of the original problem.

Our hypothesis was that practical scheduling problems do have components with inherent temporal and resource-related similarities. The experiments confirmed that the simple but generic notion of progressive solutions is appropriate to capture this structural property. Further on, applying progressive constraints made the scheduling problems almost in each case easier to solve.

The proposed method naturally extends to richer scheduling models, including earliest start and latest finish times, setup times, or various other criteria, such as the minimization of tardiness costs. Finally, one has still to investigate if the harmful interaction of the progressive constraints and the search heuristic can be eliminated, likewise it is done by advanced techniques of symmetry breaking.

Acknowledgement

This research has been supported by the grants NKFP 2/010/2004 and OTKA T046509.

References

1. A.V. Aho, J.E. Hopcroft, and J.D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.
2. Ph. Baptiste, L. Peridy, and E. Pinson. A Branch and Bound to Minimize the Number of Late Jobs on a Single Machine with Release Time Constraints. European Journal of Operational Research, 144(1), pp. 1–11, 2003.
3. E.L. Demeulemeester and W.S. Herroelen. Project Scheduling: A Research Handbook. Kluwer Academic Publishers, 2002.
4. J.E. Hopcroft and R.E. Tarjan. A V^2 Algorithm for Determining Isomorphism of Planar Graphs. Information Processing Letters 1, pp. 32–34, 1971.

[4] There is an apparent disproportion between search nodes and time for p9 and p10: the number of nodes doubles while time multiplies by ca. 40. This is due to the fast processing of nodes until a first solution is found (where the number of nodes equals the number of tasks), and heavier computation later, with a valid upper bound.

5. B. Jenner, J. Köbler, P. McKenzie, and J. Torán. Completeness Results for Graph Isomorphism. Journal of Computer and System Sciences 66(3), pp. 549–566, 2003.
6. A. Kovács. Novel Models and Algorithms for Integrated Production Planning and Scheduling. PhD Thesis, Budapest University of Technology and Economics, 2005. http://www.sztaki.hu/~akovacs/thesis/
7. A. Kovács and J. Váncza. Completable Partial Solutions in Constraint Programming and Constraint-based Scheduling. In Proc. of the 10th International Conference on Principles and Practice of Constraint Programming, Springer LNCS 3258, pp. 332–346, 2004.
8. E. Luks. Isomorphism of Bounded Valence Can Be Tested in Polynomial Time. Journal of Computer and System Sciences 25, pp. 42–46, 1982.
9. A.A. Mastor. An Experimental and Comparative Evaluation of Production Line Balancing Techniques. Management Science 16, pp. 728–746, 1970.
10. W. Nuijten, T. Bousonville, F. Focacci, D. Godard, and C. Le Pape. Towards an Industrial Manufacturing Scheduling Problem and Test Bed. In Proc. of the 9th Int. Conf. on Project Management and Scheduling, pp. 162–165, 2004.
11. K.E. Petrie and B.M. Smith. Comparison of Symmetry Breaking Methods in Constraint Programming. In Proc. of the 5th International Workshop on Symmetry and Constraint Satisfaction Problems, 2005.
12. S.D. Prestwich and J.C. Beck. Exploiting Dominance in Three Symmetric Problems. In Proc. of the 4th International Workshop on Symmetry and Constraint Satisfaction Problems, pp. 63–70, 2004.
13. J. Váncza, T. Kis, and A. Kovács. Aggregation – The Key to Integrating Production Planning and Scheduling. CIRP Annals – Manufacturing Technology 53(1), pp. 377–380, 2004.