Impact-Based Search in Constraint-based Scheduling

Armin Wolf
Fraunhofer FIRST, Kekulstr. 7, 12489 Berlin
armin.wolf@first.fraunhofer.de

Abstract: A novel adaptation of impact-based search strategies for constraint-based resource scheduling is presented. Search based on impacts applies a general purpose search strategy originally from Linear Integer Programming and recently adapted to Constraint Programming. To my knowledge it is shown for the first time that this strategy is properly applicable to constraint-based scheduling and performs well on the class of job-shop scheduling problems. Evidence is given empirically by comparison with a problem-specific and a random strategy.

1 Introduction

In Constraint Programming (CP) one of the essentials to be successful in problem solving is the ability to design an according search strategy. Mostly, there are problem specific strategies, especially in constraint-based scheduling. Scheduling of activities on resources are in general NP-hard problems (cf. [BIPN01]), especially if they must be scheduled optimally with respect to an objective function like their minimal make-span. Nevertheless, beyond polynomial algorithms for pruning the search space (e.g. [BIPN01, BC01, Vil04, Vil07, Wol03]) specialized, highly sophisticated search strategies (cf. [BL00, Vil05, Wol04, Wol05]) have been applied successfully. In contrast to these special-purpose approaches, this article follows the idea presented in [Ref04]: a general-purpose, impact-based search strategy. However, in [Ref04] this strategy inspired from Linear Integer Programming is applied to problems like multi knapsack and magic square problems where the variables’ domain are rather small. Due to the use of the impacts of variable-value assignments the strategy requires approximations for problems with larger variables’ domains as shown in [Ref04], e.g. for Latin square completion problems.

Within this paper a novel approach is presented such that impact-based search is applicable and performs well in constraint-based scheduling where the variables have large or even huge domains. Here, the impacts are independent from variable-value assignments because they are computed for order decisions.

2 Search Strategies for Scheduling Problems

The considered scheduling problems are defined by a set activities and a set of resources. An activity $t$ is non-interruptible, has a non-empty set of potential start times $S_t$, i.e. a
finite integer set which is the domain of its variable start time \( s(t) \). Furthermore, it has a fixed duration \( d(t) \), i.e. a positive integer value. Due to lack of space, the following considerations are restricted to non-preemptive single-resource scheduling problems: each activity is related to exactly one resource which will be required exclusively during the activity’s processing. Furthermore, the activities might be related by additional temporal constraints like “before”, “after”, “together” etc. Thus, given a finite set of activities \( T = \{ t_1, \ldots, t_n \} \) with at least two elements \( (n \geq 2) \), the considered scheduling problem is to find a solution, i.e. some start times \( s(t_1) \in S_{t_1}, \ldots, s(t_n) \in S_{t_n} \) such that either \( s(t_i) + d(t_i) \leq s(t_j) \) or \( s(t_j) + d(t_j) \leq s(t_i) \) holds for \( 1 \leq i < j \leq n \).

In Constraint Programming (CP) the usual approach to solve such a scheduling problem consists of two steps:

1. **model** the scheduling problem as a constraint satisfaction problem (CSP) with an appropriate set of constraints.
2. **search** for a solution of the CSP by using the pruning algorithms implemented for the constraints.

The CSP modeling will be mainly based on some global constraints serializing the activities on a single-resource. Additionally some order constraints might state some temporal conditions between the activities. Then in CP a **depth-first backtracking tree-search** is usually applied to solve this problem. At each level of the search tree decisions are made partitioning the actual search space: Given the currently valid constraint conjunction \( C \), decisions \( e_1, \ldots, e_k \) have to be taken such that \( (e_1 \lor \ldots \lor e_k) \land C \leftrightarrow C, \quad C \not\rightarrow (C \land e_i) \) and \( e_i \land e_j \leftrightarrow \text{false} \) hold for \( 1 \leq i < j \leq k \). Then, the search will select and state one decision \( e_i \), i.e. \( C \land e_i \). If pruning detects no inconsistency search continues at the next level. Otherwise **backtracking** is performed: Another not yet taken decision is selected if possible; otherwise the previous level is considered recursively. For instance, **labeling** fits in this pattern. There, the decisions on each level in the search tree are the assignments of different values to a variable. For example, given \( C \equiv (X > 0 \land X < 4 \land X \in [-2, 2]) \) then the decisions \( e_1 \equiv (X = 1), e_2 \equiv (X = 2) \) satisfy the previously stated requirements, e.g. \( (e_1 \lor e_2) \land C \leftrightarrow C \), i.e. no solution will be lost during the search.

In general, the efficiency of such a tree-based search strongly depends on the choices and selections made as well as their order in the traversed search tree. In general, the choice/selection orders are variable/value orders: At each level a not yet considered constrained variable \( X \) (e.g. a start time) with its current domain of values \( \{v_1, \ldots, v_k\} \) (e.g. the actual potential start times) is selected. Then, selecting one of these values, say \( v_j \), determines the decision \( X = v_j \).

Thus, there are three recommended general principles to reduce the search effort:

1. make decisions (e.g. for a variable) which maximally restrict the search space
2. select a decision (e.g. a variable’s value) maximizing the number of possibilities
3. make good choices at the top of the search tree.
For single-resource scheduling, especially for job-shop scheduling it is in general sufficient to determine a linear order of the activities on each single-resource. Thus at each level of the search tree two “unordered” activities \( p \) and \( q \) have to be selected and either \( p \) “before” \( q \), i.e. \( s(p) + d(p) \leq s(q) \), or vice-versa \( q \) “before” \( p \), i.e. \( s(q) + d(q) \leq s(p) \) has to be stated. Thus, these principles have to be adapted properly to the partial ordering of the activities.

A simple heuristic that addresses the first principle in job-shop scheduling is the consideration of the resource with highest demand first. Here, the demand is the ratio of the sum of durations and the difference between the latest possible end time and earliest start time of all its activities. Then sort all activities on the current resource such that their slack is not decreasing. Here, the slack is the ratio of the difference between the earliest and latest start time of this activity and the activity’s duration. Then, the pairs of the first and second activity, the first and third, etc. will be considered for their partial ordering (cf. [Wol05]). Sorting could be either performed static before the search or updated during search. In the following, another approach is presented. It is based on impacts (cf. [Ref04]) addressing all three principles. However, it is generalized for order decisions. Again, the impact of a decision means the reduction of the search space due to the pruning triggered by this decision.

In general, the Cartesian product of all potential start variables \( P = |S_{t_1}| \times \cdots |S_{t_n}| \) before and after a decision is a good estimation of the size of the search space. By convention let \( P' \) denote this Cartesian product after a decision if \( P \) denotes this product before any decision.

For another estimation the number of not detectable preferences [Vil04, Vil07] between any two activities \( p \) and \( q \) are computed before and after a decision. \( N \) resp. \( N' \) denotes this number before/after a decision. Any two activities have no detectable preferences if neither \( \min(S_p) + d(p) > \max(S_q) \) nor \( \min(S_q) + d(q) > \max(S_p) \) holds, i.e. such pairs of activities are unordered.

Considering both estimations, the overall impact of a partial ordering of a previously unordered pair \( p, q \) is the normed weighted sum

\[
I(p, q) = I(s(p) + d(p) \leq s(q)) = \alpha(1 - 2^{N'-N}) + \beta(1 - P'/P)
\]

where \( \alpha + \beta = 1 \) and \( \alpha, \beta \geq 0 \) holds. Here, the ratio \( 2^{N'-N} \in (0, 1] \) is a measure for the reduction of the potential order decisions and the ratio \( P'/P \in (0, 1] \) is a measure for the reduction of the potential start times value. The smaller the ratio \( R \) the greater the reduction and thus the impact \( (1 - R) \) of a decision. These impacts are computed before any search for all \( O(n^2) \) unordered pairs addressing the last principle: The pairs are sorted with respect to these initial impacts in decreasing order. While searching, i.e. establishing different partial orders, the corresponding impacts \( I_1, \ldots, I_d \) according to the currently considered CSP are computed. On this basis their average value

\[
\overline{I}(p, q) = \overline{I}(s(p) + d(p) \leq s(q)) = 1/d \sum_{i=1}^{d} I_i(s(p) + d(p) \leq s(q))
\]
is used to follow the first two principles: A pair $p, q$ of unordered activities to be ordered next is the one having maximal $\bar{T}(p, q) + \bar{T}(q, p)$. Ties are broken on the impact of the choice of such a pair: the one that will have maximal impact at the current level in the search tree. For such a pair $p, q$ the next decision is selected such that $I(p, q)$ resp. $I(q, p)$ will be minimal, i.e. some “look ahead” is performed.

Finally, the impact-based search restarts at top level still remembering the already computed impacts. For restarting the simple approach suggested in [Ref04] is used: For the first run of the search at most $3n(n - 1)/2$ choices are possible to find a solution. If no solution was found for this cut-off, this value is increased by multiplying it with $1.4142$ (approx. $\sqrt{2}$) before search is restarted. This increase guarantees that the search process is complete, i.e. it will find a solution if there is any and will prove inconsistency if there is none.

3 Experiments

Well-known benchmark instances of job-shop scheduling (JSS) problems are chosen to compare different search strategies: the $10 \times 10$ (LA16–20) and the $15 \times 10$ (LA21–25) instances introduced in [Law84]. Three different search strategies are compared on these resource constrained project scheduling instances: the previously introduced impact-based search with $\alpha = \beta = 0.5$, the sorting search presented in [Wol05] and restated in the previous section as well as a random search that sorts the pairs to be ordered randomly restarting in the same manner as the impact-based search (cf. previous section).

All search procedures perform in two phases sufficient for the considered benchmark problems: After the linear orders on the single-resources are established due to search, the start times of the activities are labeled with their earliest possible times without search. The different search strategies are applied to the problem instances twice: (1) restricting the make-span to be less than its minimal value proving the optimality of the minimal make-span and (2) restricting the make-span to be equal to its minimal value finding an optimal solution.

All search strategies as well as the problem models are realized in our Java constraint solving library firstCS [Wol06]. The constraint models consists of two types of constraints: SingleResource for each resource serializing the corresponding activities and Before for the linear orders of the activities within each job. Before constraints are also used for ordering unordered activities during search.

The comparison of the search strategies is performed under Windows XP, SP2 on a PC Pentium 4, 2.99 GHz, with 2 GByte RAM running Sun Java 1.6. The results are presented in Table 1. Fields without any entry reflect the fact that ongoing unsuccessful search was interrupted after one hour. The results for the LA21 instance are omitted because each strategy neither proves optimality nor finds an optimal solution within one hour runtime. No search was required to prove the optimality of the LA23 instance: initial pruning detects the inconsistency of the considered CSP. Best results are highlighted in boldface. Comparing the numbers shows that impact-based search performs well for the proofs of
optimality: In 75% of all non-trivial cases it performs best according to the number of performed backtracks (# backtracks) and choices made (# choices) as well as runtime (in msec.). Further, impact-based search performs also rather well for finding an optimal solution, i.e. a schedule. However, the problem-specific adapted sorting search performs a bit better. Random search is in both cases the least performing strategy.

<table>
<thead>
<tr>
<th>JSS instance</th>
<th>LA16</th>
<th>LA17</th>
<th>LA18</th>
<th>LA19</th>
<th>LA20</th>
<th>LA22</th>
<th>LA23</th>
<th>LA24</th>
<th>LA25</th>
</tr>
</thead>
<tbody>
<tr>
<td>min. make-span</td>
<td>945</td>
<td>784</td>
<td>848</td>
<td>842</td>
<td>902</td>
<td>927</td>
<td>1032</td>
<td>935</td>
<td>977</td>
</tr>
</tbody>
</table>

proof of optimality: (I)mport-based, (S)orting, (R)andom

| # backtracks(I) | 537  | 47   | 483  | 9429 | 671  | 633  | 0    | 75458 | 1924905 |
| # choices(I)    | 536  | 46   | 482  | 9479 | 670  | 632  | 0    | 75584 | 1925304 |
| time [msec.](I) | 594  | 375  | 672  | 6579 | 766  | 1657 | 0    | 97815 | 2120822 |
| # backtracks(S) | 1355 | 7    | 5289 | 19901| 3773 | 295  | 0    | 130405| —     |
| # choices(S)    | 1354 | 6    | 5288 | 19000| 3772 | 294  | 0    | 130404| —     |
| time [msec.](S) | 594  | 32   | 2156 | 8844 | 1953 | 313  | 0    | 108676| —     |
| # backtracks(R) | 415756 | 313 | 97039 | 2813405| 1900578 | —    | 0    | —     | —     |
| # choices(R)    | 4158462 | 312 | 97423 | 2814188| 1901317 | —    | 0    | —     | —     |
| time [msec.](R) | 1287777 | 203 | 34095 | 983631 | 721090 | > 1h | 0    | > 1h  | > 1h  |

finding an optimal solution: (I)mport-based, (S)orting, (R)andom

| # backtracks(I) | 93   | 2    | 366  | 6812 | 496  | 1850 | 1252 | 50611 | 640820 |
| # choices(I)    | 129  | 33   | 396  | 6873 | 523  | 1910 | 1337 | 50781 | 641239 |
| time [msec.](I) | 297  | 281  | 848  | 842  | 902  | 3093 | 2266 | 65052 | 721608 |
| # backtracks(S) | 123  | 1    | 3087 | 17245| 469  | 550  | 48   | 36151 | —     |
| # choices(S)    | 156  | 28   | 3128 | 17285| 503  | 595  | 95   | 36208 | —     |
| time [msec.](S) | 78   | 31   | 1297 | 17610| 266  | 609  | 156  | 27486 | > 1h  |
| # backtracks(R) | 46785 | 15   | 229754 | 198568 | 351237 | —    | —    | —     | —     |
| # choices(R)    | 47087 | 53   | 230312 | 199024 | 351826 | —    | —    | —     | —     |
| time [msec.](R) | 1439 | 31   | 78956 | 73628 | 126036 | > 1h | > 1h | > 1h  | > 1h  |

Table 1: Benchmark results for some LA job-shop scheduling instances

4 Conclusion

A novel adaptation of impact-based search strategies for constraint-based resource scheduling is presented. It is shown that this strategy – properly applied to constraint-based scheduling – performs well on the class of job-shop scheduling problems: evidence is given empirically by comparison with a problem-specific and a random strategy. The encouraging results will motivate some future work on fine tuning, e.g. of the parameters \( \alpha \) and \( \beta \), based on more exhaustive experiments especially on other job-shop scheduling instances.

References

[BC01] Nicolas Beldiceanu and Mats Carlsson. Sweep as a Generic Pruning Technique Applied to the Non-overlapping Rectangles Constraint. In Toby Walsh, editor, *Principles and


