A Path-Repair Algorithm for Solving Scheduling Problems

Chia-Lin Hsieh¹
(Received November 21, 2003; Revised December 8, 2003; Accepted December 29, 2003)

Abstract

Search algorithms for solving CSP (Constraint Satisfaction Problems) usually fall into one of two main families: local search algorithms and systematic algorithms. Both families have their own advantages. Designing hybrid approaches seem promising since those advantages may be combined into a single approach. In this paper, we propose a new hybrid technique which performs an overall local search over partial assignments instead of complete assignments, and uses filtering and conflict-based techniques to efficiently guide the search and maintain local consistency. In the case of inconsistency, it proposes explanations to retract only those constraints that are really involved in the contradiction. This new technique benefits from both classical approaches: a priori pruning of the search from filtering-based search and possible repair of early mistakes from local search. We focus on a specific version of local search technique called best-first-repair which determines, upon the constraint inconsistency, a pertinent choice point to repair the search path to restore the satisfiability. Experiments done on open-shop scheduling problems have shown with the best highly specialized algorithms.

Keywords: Constraint Satisfaction Problems; Explanation-Based Search; Local Repair; Hybrid Search Algorithm.

1. Introduction

Many industrial and engineering problems can be modeled in the form of constraint satisfaction problems (CSPs). A classic CSP involves a set of variables each with an associated domain of possible values and a set of constraints over those variables. Algorithms for solving CSP usually fall into one of two main families: systematic and local search algorithms.

Systematic algorithms [1-2] for CSP typically explore a search tree which is based on the proposed values for each of the variables of the solved problem. Such search algorithms start from an empty variable assignment that is extended until obtaining a

¹ Department of Industrial Management, Aletheia University, Tamsui, Taipei, Taiwan, R.O.C.
complete assignment that satisfies all the constraints in the problem. Backtracking occurs when a dead-end (i.e. the domain of a variable is emptied) is reached. The biggest problem of such backtracking-based search algorithms is that they are typically cursed with early mistakes in the search. That is, a wrong variable value can cause a whole sub-tree to be explored with no success.

Local search algorithms (e.g. min-conflict [3], GSAT [4], tabu search [5]) perform an incomplete exploration of the search space by repairing infeasible complete assignments. They do not suffer from the early-mistake problem. Local search algorithms are capable of following a local gradient in the search space. They may be far more efficient than systematic algorithms to find the first solution. For optimization problem, they can reach a better quality in certain specific time frame. Unfortunately, they can not guarantee to find a solution, and may be unable to find one. Thus, they are not the panacea.

In this paper, we present a hybrid approach to perform an overall local search, and using systematic search to select a candidate neighbor or to prune the search space. Our main goal is to show that systematic algorithms can be exploited for local search algorithms and can also greatly improve their behavior. This leads us to propose a generic search technique over CSP called path-repair.

The basic idea is that the algorithm starts with a partial solution which is the result of partial assignments. It first applies a filtering technique. When no inconsistency is detected, the algorithm adds a decision constraint that extends the current solution, and the search continues. When a dead-end is reached, we know that there exists an incompatibility between the decision constraints so far. The algorithm tries to resolve the conflicts between the decision constraints. The current requirements may be achieved by maintaining useful information during search in order to realize more efficient resolutions. Efficient failure analysis is required to identify the minimal conflict set. This new incomplete technique is studied through solving open-shop scheduling problems. We compare it against highly specialized algorithms and study its behavior while varying the values of its parameters.

The paper is organized as follows. In section 2, related definition is introduced. Next section, a path-repair algorithm mixed up with certain local-repair heuristic which serves to identify a decision constraint to recover inconsistency is represented. After presenting relations with previous works in section 4, a case study of the path-repair algorithm on open-shop scheduling problems is discussed in section 5.
Results for the implementation on open-shop benchmark problems are given in section 6. Finally, conclusion and future works are presented.

2. Preliminaries

A CSP is a couple \(<V, C>\), where \(V = \{v_1, v_2, ..., v_n\}\) is a set of variables and \(C = \{c_1, c_2, ..., c_m\}\) is a set of constraints. For a given \(k\)-ary constraint on \(k\) variables \(\{v_1, v_2, ..., v_k\}\) is a logical formula that defines the allowed combinations of values for the variables \(\{v_1, v_2, ..., v_k\}\). And \(\hat{K}\) is the logical conjunction of the constraints \(C\) in the form \(K : \hat{K} = (c_1 \land ... \land c_k)\). A constraint can be given in extension or in intension. Domains of the variables are handled as unary constraints.

Classical CSP problem solving simultaneously involves filtering algorithms \textit{a priori} to prune the search space and an enumeration mechanism to overcome filtering incompleteness. Filtering algorithms (e.g. arc-consistency filtering [6] for binary CSP over finite domains) are typically used at each node of the search tree developed by systematic algorithms for solving CSPs. They are required not only to perform a pre-processing to exploit all dependent constraints but also keeps track of changes for future use. When a filtering step causing an empty domain (a dead-end [7]), classical explorations based on standard backtracking would require an optimal step to obtain the best solution. After such a filtering step, three situations may arise:

(1) The domain of a variable becomes empty: there is no feasible solution;
(2) All the domains are reduced to a singleton: those values assigned to their respective variables providing a feasible solution for the problem;
(3) There exists at least one domain which contains two values or more: the search has not yet been successful. In a classical approach, it would time for enumeration through a backtracking-based mechanism.

In a more general way, a filtering algorithm \(\Phi\) applied to a set \(C\) of constraints returns a new set \(C' = \Phi(C)\) such that \(C \subseteq C'\). Moreover, for any filtering algorithm \(\Phi\) applied to a set \(C\) of constraints of a given CSP, there exists a function \(\text{Inference}\) to interprets the results of the filtering algorithm for the overall search algorithm. When applied to \(C' = \Phi(C)\), the function \(\text{Inference}\) answers:

\[2\] We consider domain reduction as additional constraints.
The function *Inference* typically has a low computational cost. Its aim is to make explicit use of some properties that depend on the filtering algorithm. A function *Inference* can be identified in many other filtering or pruning algorithms. For example in integer linear programming, the aim is to find an optimal integer solution. This can be achieved by using the simplex method over the real numbers. If there is no real solution or if the real optimum has only integer values, then an *Inference* function would return *noSolution* or *Solution* respectively.

3. A Path-Repair Search

The idea of the proposed *Path-Repair* search is simple. Such an algorithm works upon a partial instantiation of the variables using filtering techniques to remove unsupported values from the domain of the variables by a set of enumerating constraints upon the variables of the problem. The partial instantiation under consideration is defined by a set of decision constraints on the variables of the problem. Such decision constraints define a *path* in the search tree.

3.1 Principles of the Path-Repair Search

The principle of the *Path-Repair* algorithm is shown in Figure 1. The search starts with an initial path. It may range from an empty path that defines a complete assignment. The main loop first checks the consistency of the search.
A Path-Repair Algorithm for Solving Scheduling Problems

Procedure Path-Repair(C)
Begin
\[ P := \text{initial path}; \]
Do while true
\[ C' = f(C \cup C_p); \]
Swith Deduction(C');
Case noSolution:
\[ k \leftarrow \text{explanation for the failure}; \]
Best-First-Repair(\( P, k, \xi \));
Case Solution:
Return (C');
Default:
\[ P \leftarrow \text{Extend}(P, \xi); \]
EndSwith
Until false;
End;

Figure 1 The Path-Repair Algorithm

The parameter \( C \) of the Path-Repair algorithm is the set of constraints to be solved. \( P \) is defined as a path in the search tree. At each node of that path, a decision constraint is added. \( C_p \) contains the set of added decision constraints while moving along \( P \). A filtering algorithm \( f \) is then applied on a set of constraints \( C \) which will return a new set \( C' = f(C \cup C_p) \) such that \( C \subseteq C' \). There is one thing to be considered is that we consider domain deduction as addition of redundant constraints\(^3\).

Moreover, for any filtering algorithm \( f \) applied on the set \( C \) of constraints, the function Deduction is then called over \( C' \). Three contradictions may occur:

- \( \text{Deduction}(C') = \text{Solution} \); a solution is found. The algorithm terminates and returns \( C' \);
- \( \text{Deduction}(C') = \text{Extend} \); the algorithm tries to extend the current path \( P \) by adding a decision constraint. For that purpose, a function \( \text{Extend}(P, \xi) \) is called to add a constraint to \( P \). Parameter \( \xi \) can be used to store a context. We will discuss its meaning later.
- \( \text{Deduction}(C') = \text{noSolution} \); \( C \cup C_p \) is inconsistent. We will say that the path \( P \) is a dead-end. That is, the path \( P \) can’t be deepened. In this case, a

\(^3\) We call those constraints deduction constraints.
Local-Repair algorithm will try to repair the current path through the function
Best-First-Repair(\(P, k, \xi\)). As \(\xi\), parameter \(k\) will be discussed later (in section 3.2).

The function Deduction has typically a low computational cost. Its aim is to make
explicit use of some properties that depends on the filtering algorithm been used. The
Path-Repair algorithm appears here as a search method that generates partial search
trees and decides when the repair should happen. It also uses filtering techniques to
prune the search tree. The key components are the repair function Neighbor and the
extension function Extend.

3.2 Explanations in Path-Repair Search

The proposed method relies on the key concept of explanation\(^4\). An explanation
is a set of constraints whose conjunction leads to a contradiction. It is an explanation of
the failure. Explanation is kept when the variable is updated during enumeration and is
contained in the “trail” of the domain modification. When a dead-end occurs, this
information associated with the variable terms as an explanation, which will be pushed
into a conflict set \(\xi\) to trigger a repair procedure.

**Definition 1 (Explanation)** An explanation \(k\) for a set of constraints \(C\) and a path
\(P\), is a subset of \(C_P\) such that \(C \land k \Rightarrow \text{false}\).

**Definition 2 (Conflict Set)** A conflict set \(\xi\) is a list structure to keep all generated
explanations during search.

As long as constraints in an explanation \(k\) remain in a given path \(P\), the path
will remain inconsistent. We observe that at least one constraint has to be retracted from
explanation \(k\) to make path \(P\) consistent. Therefore, the removal of some constraints
from the conflict set \(\xi\) will give a solvable constraint system.

Explanations are provided by the filtering algorithm as soon as it can prove that no
solution exists in the subsequent paths from the current partial path. Suppose that for
each value or set of values \(a_i\) removed from the domain of variable \(v\), a set of
decision constraints \(k_i \subseteq C_p\) is given. Thus, \(k_i\) is called a removal explanation for \(a_i\)
and is such that \(C \land k_i \Rightarrow v \neq a_i\). If so, \(k = \bigcup_i k_i\) is an explanation since no value for

\(^4\) It is similar to the terms eliminating explanations [8] and no-goods in [9]
v is allowed by the union of $a_i$. Therefore, in order to compute explanations, it is required to compute an explanation for each domain reduction (i.e. removals of values) when encountering a dead-end during the search. Actually, a contradiction is identified at the dead-end variable. Since all the value removals have an explanation, the union of the value removal for each value of the empties domain is obviously a contradiction explanation. Value removals are the consequences of the filtering algorithm. Therefore, explanations can be easily computed by using a “trailing” mechanism within the filtering algorithm to memorize the reason why a removal is performed [10].

There is another condition. When all alternatives to the current node have been unsuccessfully tested, the set of constraints results in contradiction. The path to the current node in the search tree is itself an explanation for the contradiction. Let $C_e$ be a contradiction explanation. Let $r$ be the most recent node included in $C_e$. Ginsberg [8] proposed an intelligent backtracking mechanism that concluded: “During the search, any backtracking to any node between $r$ and the current node is useless since all the decisions made in $C_e$ remain contradictory”. The contradiction explanation enables the search directly jump back towards $r$ which saves more or less works. Thus, the set $C_e \setminus \{r\}$ is then the explanation for the non-selection of the path to node $r$. No improvement will be made unless all the possibilities on node $r$ have been tested. For our point of view, intelligent backtracking mechanism can be used to trace back to the last solution when contradiction is occurred during the search.

### 3.3 Best-First-Repair in Path-Repair Search

In this section, we present our generic Best-First-Repair function in the Path-Repair algorithm. The method performs no backtracks but jumps.

#### 3.3.1 Principles of the Best-First-Repair Function

The Best-First-Repair function is proposed to restore inconsistency. Explanations reveal some useful information for failure analysis. Obviously, a strict subset of them may provide much more precise information but cost more on computation. Thus, no minimal set of explanations will be computed in practice. Like a local search algorithm, the principle of our search is based on the selection of neighbors in which many heuristics may be used.

We have defined a neighbor path $P'$ of a path $P$ according to one or more explanations $k$ of its conflict set $\xi$, where $P'$ and $P$ have at least one constraint
different from \( k \). Indeed, a more precise neighbor can be computed. Let \( c \) be a constraint to be retracted from \( C_p \). We observe that as long as all the constraints in \( k \) but \( c \) remain in the path, \( c \) can never be satisfied. Thus, the negation of \( c \) should be added in the new path. In this way, all possible neighborhood for an inconsistent path \( C_p \) can be found. That is to say, a possible neighborhood for an inconsistent path \( C_p \) is made from one explanation \( k \subset C_p \) of its conflict set \( \xi \) by negating one constraint in \( k \).

Once we choose a neighbor path \( P' \) instead of \( P \), the exploration performs no backtracking but jumping among the search tree. The exploration traverses in the same level of the tree with minimal costs. This jump avoids as much thrashing as possible and cuts branches at each identification of an explanation.

Let us consider the following example. Let \( P \) is the path \((c_a, c_b, c_c, c_d)\) with an explanation \((c_a, c_b, c_c)\). The neighbor paths of \( P \) can be defined by three sets \((c_a', c_b, c_c, c_d)\), \((c_a, c_b', c_c, c_d)\) and \((c_a, c_b, c_c', c_d)\). Suppose the path \((c_a', c_b, c_c, c_d)\) is chosen. The result can be illustrated from Figure 2. The search will jump from path \( P \) to path \( P' \) such that the subsequent of tree will be kept unchanged.

Figure 2 Best-First-Repair Search Between Neighbors

Now, there remains to specify which neighbor should be chosen among the defined neighbors. The degree of freedom for the choice of the negation constraint in \( k \) enables the use of heuristics. The Best-First-Repair function in Path-Repair search enables to choose proper neighbors efficiently.
3.3.2 A Best-First-Repair Heuristic for Failure Analysis

The main purpose of failure analysis is to retract a *near-minimal* set of constraints to restore *satisfiability* (not *optimality*). A simple *Best-First-Repair* heuristic strategy is used to consider the most constraining constraints in the conflict set $\xi$. The heuristic approach is greedy since it attempts to minimize the number of updates after each constraint deduction. However, if there is a tie between them, we choose the one with the most recent backtrack point (*i.e.* the set with the later introduction order) for it leads to more complete procedure than native backtracking.

The *Best-First-Repair* Heuristic is shown in Figure 3. The function $\text{Min\_Conflict}(\xi)$ checks all the explanations in conflict set $\xi$ and tries to find one decision constraint in $k$ such that negating this decision makes the decision set compatible with all the explanations. When several decision constraints can be negated, we use the following heuristic, called *weight-explanation* heuristic to choose the most constraining decision constraint $c$ in the conflict set. In the heuristic, a weight $r$ is associated with each decision constraint. The weight remembers the number of times that the decision has appeared in any explanation. Once an explanation is found, the weight of the decision constraints is increased by $1/\gamma$ where $\gamma$ is the arity of the explanation.

![Figure 3 Best-First-Repair Search](image)

```plaintext
Function Best-First-Repair( P, k, \xi )
/* precondition: k is an explanation of P */
add k to the conflict set;
repeat
  c = Min\_Conflict(\xi);
  P' = Neighbor( P, c);
  if C_{P'} covers all explanations in \xi
     return P';
until \xi = \emptyset;
return;
```

The function $\text{Neighbor}(P,c)$ negates the chosen constraint $c$ with the greatest weight, when negated, makes the new decision set compatible with all the conflicts in $\xi$. 
Suppose such a constraint does not exist, we may consider to extend the neighbors. That is, another negation should be triggered. The algorithm terminates when the conflict set $\xi$ becomes empty or a neighbor path that covers all the explanations has been found.

A simple example is illustrated as below. Suppose we have a conflict set $\xi = \{k_1 = \{c_1, c_2, c_4\}, k_2 = \{c_2, c_4\}, k_3 = \{c_1, c_3\}\}$, where the index of $c_i$ denotes the inserting order and the elements $k_1, k_2, k_3$ of $\xi$ denote explanations found from the enumeration. Suppose the function $\text{Min}\_\text{Conflict}(\xi)$ first returns $c_1$ and jumps to path $P_1$ which cannot cover $k_2$. Thus, another neighbor of $P_1$ should be extended and another repair is triggered until the condition is satisfied. Finally, once $c_4$ has been found, the inconsistency can be restored. Actually, the set $\{c_1, c_2\}$ or $\{c_1, c_4\}$ are the possible candidates in the conflict set $\xi$ to be retracted. However, $\{c_1, c_4\}$ will be considered (for $c_4$ is inserted after $c_2$) in our approach.

We have observed that this strategy still leaves some choices unspecified and makes the search incomplete so that it doesn't compute an optimal (minimal) subset. But it does have the effect of localizing the conflicts to a subset of constraints easily determined from filtering procedure. This algorithm will terminate because:

- Only new explanations are introduced and the $\text{Neighbor}$ function returns a new path which has never been traversed;
- The number of existing paths is finite ($2^c$, actually, where $c$ is the number constraints in the problems).

4. Related Works

The combination of systematic search and local repair heuristic is the main spirit of the path-repair algorithm, which is similar to many other existing works. Among them, Ginsberg and McAllester [11] has probably been the most influential by highlighting the relationships between local search and systematic search. The search is guided by the use of explanations to make it systematic. In the same spirit are [7,9,12]. Path-repair algorithm is a generic algorithm which generalizes not only some non-systematic algorithms but also some systematic ones.

The algorithm proposed in [13] can be most be seen as an instance of the path-repair algorithm where the decision constraints are instantiations; there is no...
A Path-Repair Algorithm for Solving Scheduling Problems

propagation and no pruning; and it does not make use of conflicts, neither in the neighbor function nor in the extend function. The common idea is to extend a partial instantiation when it is consistent and to perform a local change when the partial solution reaches a dead-end.

The use of explanations are computed by the filtering algorithm is a well-known techniques that has been used with different variations for different combinations of filtering algorithms with systematic search algorithms [14-18]. Nevertheless, the version of the path-repair algorithm is the first one in combination with a local search algorithm.

5. A Case Study: Solving Open Shop Scheduling Problems

In order to verify our idea, the proposed algorithm is now implemented on an open-shop scheduling problem. Our main idea is to record an explanation for each domain reduction during the search.

5.1 Open-Shop Scheduling Problems

Classical scheduling shop problems for which a set \( J \) of \( n \) jobs consisting each of \( m \) tasks (operations) must be scheduled on a set \( M \) of \( m \) machines can be considered as a constraint satisfaction problem (CSP) upon intervals. It can also be considered as an optimization problem for building non-preemptive schedules for minimal makespan\(^6\). Variables are the starting dates of the tasks. Bounds thus represent the least feasible starting time and the least feasible ending time of the associated task. One of those problems is called the open-shop problem [19]. For this kind of problems, tasks for a given job may be processed in any order, but only one at a time.

Few branch and bound methods for open-shop scheduling problems have been proposed so far. One of the best one [20] consists, in each node, in fixing disjunctions (adding new tentative precedence constraints) on the critical path of a heuristic solution. This method uses immediate selections [21-22] in each node in order to fix additional disjunctions by propagation. The open-shop scheduling problem is NP-hard for \( \min(m, n) \geq 3 \). This problem looking quite simple to describe is really hard to solve optimally. The instances of size 6*6 variables remain unsolved so far.

\(^6\) Ending time of the last task.
5.2 Path-Repair Search for Open-Shop Scheduling Problems

A revised Path-Repair approach has been tried on the open-shop problems (see Figure 4). The open-shop problems we consider are optimization problems. This requires a main loop that calls the function path-repair until improvement is no longer possible. Improvements are generated by adding a constraint that specifies that the makespan is less than the current best solution found. The Intelligent-Backtrack function also plays an important role to efficiently improve the search due to the use of explanations.

```
Procedure Min-MakeSpan(C)
Begin
    P ← initial path;
    UB ← ∞; /* UB = Upper Bound */
    LastSolution ← false;
    Repeat
        C ← C ∪ {makespan < UB}
        Solution ← Path-Repair(C);
        Switch Deduction(C) on node r
        Case noSolution:
            exp(r) ← explanation for the failure;
            LastSolution ← Intelligent-Backtrack(P, exp(r));
            return LastSolution;
        Case Solution:
            UB ← value of makespan in solution;
            LastSolution ← Solution;
        EndSwitch
    Until false;
End;
```

Figure 4 Revised Path-Repair Search for Open-Shop Scheduling Problems

Our variables are the starting dates \( v_i \) of each job \( i \) in the open-shop problem. The domain associated to each variable \( v_i \) is the interval \([r_i, f_i]\), where \( r_i \) is the release time of job \( i \) and \( f_i \) is its latest start time. As the value of \( r_i \) (respect to \( f_i \))
increases (respect to decreases) only during the search, once a \( r_i \) or a \( f_i \) is modified, all the nodes whose conjunction is responsible for the modification will be recorded.

The explanations for \( r_i \) and \( f_i \) is the same as the one of \( r_i \) an \( q_i \), since 
\[ q_i = UB - f_i - p_i . \]
In order to be consistent with immediate selection method [21], modifications of \( r_i \) (head) and \( q_i \) (tail) will be considered. There are two kinds of explanations to be considered: fixed disjunctions and heads\(^7\).

1. Explanations for Fixed Disjunctions

\(\text{Branching:}\)

Let us assume that a disjunction \( i \to j \) is fixed in a new node \( r \) in the search tree by a branching scheme. If the precedence constraint \( i \to j \) doesn’t exist in the current node \( r \), then the explanation of the arc \((i,j)\) will be \( \exp(i, j) \leftarrow \{r\} \). If \( i \to j \) already exists, the explanation will be the same.

\(\text{Immediate Selection on Disjunctions:}\)

Let’s consider two jobs \( i \) and \( j \) in the same machine. If 
\[ r_i + p_j + p_i + q_i \geq UB \]
immediate selections on disjunctions fix the disjunction \( i \to j \). If the precedence constraint \( i \to j \) doesn’t exist, the explanation of the arc \((i,j)\) will be \( \exp(i, j) \leftarrow \exp(r_j) \cup \exp(q_i) \). If the arc \((i,j)\) already exists, the explanation will be the same.

2. Explanations for Heads

If two jobs \( i \) and \( j \) are processed on the same machine and 
\[ r_j + p_j + p_i + q_i > UB \]
then the precedence constraint \( i \to j \) is added. Also, the head of job \( i \) cannot be started until \( \max(r_j, r_i + p_i) \). If \( r_j \) is adjusted because of the precedence constraint \( i \to j \), the explanation of the new value of \( r_j \) will become \( \exp(r_j) \leftarrow \exp(r_j) \cup \exp(i, j) \cup \exp(r_i) \). If \( r_j \) is not modified, its explanation will be the same.

These explanations are used when contradiction occurs. There exists a job \( i \) for which 
\[ r_i + p_j + q_i \geq UB \]. In this case, the domain of the variable \( v_i \) becomes empty. This contradiction is due to the last modification of \( r_i \) or \( q_i \). The intelligent

\(^7\) Since \( q_i \) is the symmetric node of \( r_i \), we record explanation for the heads \( r_i \) (w.r.t. the tails \( q_i \)) as the lower (w.r.t. upper) bound of the variable value only.
backtracking will lead to the most recent node \( (i.e. \exp(r_i) \cup \exp(q_i)) \) which is responsible for the modification of either \( r_i \) or \( q_i \).

Since all children of a given node \( r \) have been explored unsuccessfully, node \( r \) is no longer valid and backtracking is unavoidable. Therefore, all the children have to be rejected due to a computable reason and a relevant backtracking point is obtained to the most recent node in the union of the explanations excluding the node \( r \).

5.3 Complexity Issue of the Algorithm

This technique has non-exponential spatial complexity since the approach is related to the number of disjunctions of the problem. Indeed, all explanations are restricted to polynomial size by keeping only their upper and lower bounds for the variables and for each node in the current path in the search tree. Since each job is disjunctive with all the jobs operated in the same machine, the number of disjunctions gives the maximum depth of the search tree in the worse case. Thus, the space needed to record all explanations (for heads, tails and disjunctions) is then \( O(m^2n^2(m + n)^3) \).

As regards to time complexity, explanations for the modifications of a head or a tail require making the union of several explanations during constraint propagation and immediate selections. Those unions involve a polynomial number of sets of polynomial size and therefore take polynomial time.

6. Evaluation Results

To verify our approach, the implementation on two series of reference problems is performed:

1. Taillard’s problems [23]: these benchmarks are consisting of 4 series of 10 problems of size 4×4, 5×5, 7×7, and 10×10.

2. Brucker et al. problems [20]: 52 problems of size 3×3 to 8×8. Those problems are characterized by a common lower bound value 1000.

The path-repair algorithm (referred to as \( PR \) in the resulting tables) is compared with the best published solving techniques for the open-shop problem:
For Taillard’s instances, our results are compared with one highly specialized tabu search problems for solving *open-shop* problems (referred to as *TB* in the resulting tables).

For all the instances, our results are compared with a genetic algorithm introduced in [24] (referred to as *GA* in the resulting tables) which gives good results on all those problems.

### Table 1 Results on Taillard’s problems

<table>
<thead>
<tr>
<th>Series</th>
<th>TB</th>
<th>GA</th>
<th>PR</th>
</tr>
</thead>
<tbody>
<tr>
<td>4x4</td>
<td>(*)</td>
<td>0.31/1.84(8)</td>
<td>0/0(10) (◊)</td>
</tr>
<tr>
<td>5x5</td>
<td>(*)</td>
<td>1.26/3.72(1)</td>
<td>0/0(10) (◊)</td>
</tr>
<tr>
<td>7x7</td>
<td>0.75/1.71(2)</td>
<td>0.41/0.95(4)</td>
<td>0.44/1.92(6) (◊)</td>
</tr>
<tr>
<td>10x10</td>
<td>0.73/1.67(1)</td>
<td>0/0 (10)(◊)</td>
<td>2.01/3.15(0)</td>
</tr>
</tbody>
</table>

(*) no results shown of this series in the paper
(◊) best results for this series

### Table 2 Results on Brucker et al. problems

<table>
<thead>
<tr>
<th>Series</th>
<th>GA</th>
<th>PR</th>
</tr>
</thead>
<tbody>
<tr>
<td>3x3</td>
<td>0/0 (8)</td>
<td>0/0(8) (◊)</td>
</tr>
<tr>
<td>(8 pbs)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4x4</td>
<td>0/0(9)</td>
<td>0/0(8) (◊)</td>
</tr>
<tr>
<td>(8 pbs)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5x5</td>
<td>0.36/2.07 (6)</td>
<td>0/0(9) (◊)</td>
</tr>
<tr>
<td>(8 pbs)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6x6</td>
<td>0.92/2.26(3)</td>
<td>0.71/3.5 (6) (◊)</td>
</tr>
<tr>
<td>(8 pbs)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7x7</td>
<td>3.82/8.2 (6)</td>
<td>4.4/11.5 (5)</td>
</tr>
<tr>
<td>(8 pbs)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8x8</td>
<td>3.1/7.5 (8)</td>
<td>4.95/11.8 (◊)</td>
</tr>
<tr>
<td>(8 pbs)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(◊) better results for this series

Table 1 presents the computational results on Taillard’s problems. Table 2 presents the computational results on Brucker et al. instances. Results are both presented in the following format: “average deviation from the optimal value”/“maximum deviation from the optimal value” (“number of optimally solved instances”). Exceptions occurred for 7x7 and 8x8 problems on Brucker et al. instances for which the deviation is computed from the lower bound value 1000.

As regards to execution time, CPU times are not available for these instances in [20, 23, 24]. Thus, there is no way to compare their performance. Just for an idea, for Taillard’s problems, path-repair results 10x10 average CPU time is 15 hours, and for 7x7 average CPU time is about 2 hours. For Brucker et al. instances, 10x10 average
CPU time is 3 to 4 hours and for size less than 8×8, average CPU time is less than 4 minutes.

The results obtained on the two sets of problems are interesting because they show that path-repair is a competitive algorithm compared with other techniques. Compared to Taillard’s instances, path-repair gives comparable results by the genetic algorithm (in [24]) shows the best results for 10×10 problems. On Brucker et al. instances, path-repair is far better than the genetic algorithm (in [24]) on small instances but the latter becomes better on larger problems.

Such good behavior of path-repair is quite promising because our implementation remains general and complete compared to other specialized algorithms. In the open-shop problems, this is probably due to the intelligent backtracking search in the algorithm. Intelligent backtracking may start by partially solving a sub-problem and then go to another one. Solve it again and then continue to solve the first sub-problem. In cases where it has to go backtrack to choices in the first part of its work, the search spaces of the two sub-problems are multiplied. Our path-repair, thanks to its use of explanations, can identify independent sub-problems and stay in a sub-problem until it has been solved. The heuristic of conflict resolution seems good as well. This is another benefit from the use of explanations.

7. Conclusion and Further Works

In this paper, we provide here a generic algorithm for solving CSP, the path-repair algorithm. The two main points of the path-repair algorithm is: its ability both to perform a local search and to prune the search space and its explanation-based heuristic. The concepts of explanation have been proved useful to implement the algorithm. Explanation allows considering the relevant neighbor selections for determining backtrack points to make the search efficient. In our previous work, such mechanism had been successfully applied to an over-constrained problem [25].

Experiments with a revised path-repair algorithm have proved its adaptation to a branch and bound method over open-shop scheduling problems. Complexity results are described. Several well known hard instances of the open-shop scheduling problems have been solved for the first time by our algorithm and competed well with the well-known specialized algorithms. This was quite promising, since the implementation is general and the search is complete.
In the future, our research will focus on the following issues:

- In our implementation for open-shop scheduling problems, the conflict set we compute is not minimal. A method presented in [26] can be used to compute minimal conflict set in reasonable time. As more precise conflicts may greatly improve the efficiency of the path-repair implementation. Such a conflict-detection method deserves further experimentation.
- Different local-search heuristics on the path-repair algorithm should be applied to evaluate the different performances in order to find a better combination.
- Experiments of the path-repair algorithm over other applications rather than scheduling problems should be performed.

**Reference**