Journal of Machine Learning Research 4 (2003) 649–682 Submitted 12/01; Revised 11/02; Published 9/03

Speedup Learning for Repair-based Search by Identifying Redundant Steps

Shaul Markovitch SHAULM@CS.TECHNION.AC.IL
Asaf Shatil SHATIL@CS.TECHNION.AC.IL
Computer Science Department
Technion, Haifa 32000, Israel

Editor: Thomas G. Dietterich

Abstract

Repair-based search algorithms start with an initial solution and attempt to improve it by iteratively applying repair operators. Such algorithms can often handle large-scale problems that may be difficult for systematic search algorithms. Nevertheless, the computational cost of solving such problems is still very high. We observed that many of the repair steps applied by such algorithms are redundant in the sense that they do not eventually contribute to finding a solution. Such redundant steps are particularly harmful in repair-based search, where each step carries high cost due to the very high branching factor typically associated with it.

Accurately identifying and avoiding such redundant steps would result in faster local search without harming the algorithm's problem-solving ability. In this paper we propose a speedup learning methodology for attaining this goal. It consists of the following steps: defining the concept of a *redundant step*; acquiring this concept during off-line learning by analyzing solution paths for training problems, tagging all the steps along the paths according to the redundancy definition and using an induction algorithm to infer a classifier based on the tagged examples; and using the acquired classifier to filter out redundant steps while solving unseen problems.

Our algorithm was empirically tested on instances of real-world employee timetabling problems (ETP). The problem solver to be improved is based on one of the best methods for solving some large ETP instances. Our results show a significant improvement in speed for test problems that are similar to the given example problems.

1. Introduction

One of the most important techniques developed in the field of Artificial Intelligence is problem solving by searching in state spaces. For some problems, the goal of the solver is to find a path from an initial state to a final state. Many search algorithms have been devised to find a low-cost or lowest-cost solution path.

In many other problems, however, such as scheduling, timetabling and VLSI design, the particular path to the goal or its cost are irrelevant. The goal in such problems is to find a solution state that satisfies given constraints or to find a best solution state according to a given optimization criterion. Such problems are solved in many cases by using repair-based methods (Zweben, 1990; Duncan, 1990; Minton et al., 1992; Schaerf, 1996; Schaerf and Meisels, 1999). These methods start with a complete suboptimal solution and attempt to improve the solution by local repair operators.

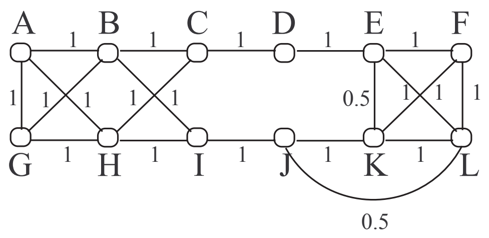

Figure 1: A TSP instance.

Examples for such methods are hill climbing, simulated annealing (Kirpatrick et al., 1983) and tabu search (Glover, 1989).

Local search algorithms can often handle large-scale problems that may be difficult for systematic search algorithms (Minton et al., 1992). Nevertheless, the computational cost of solving such problems is still very high. We observed that many of the repair steps applied by such algorithms are redundant in the sense that they do not eventually contribute to finding a solution. Such redundant steps are particularly harmful in repair-based search, where each step carries high cost due to the very high branching factor typically associated with it.

Assume, for example, that we try to solve an instance of the Traveling Salesperson Problem (TSP) by iterative repair. We start with a suboptimal tour and iteratively repair it by removing two of its edges and replacing them with the two edges that complete it into a legal tour of lower cost (Lin and Kernighan, 1973). To avoid local minima we allow also replacements that do not change the tour cost. Figure 1 shows a TSP instance. Figure 2(a) shows a sequence of steps leading to a solution generated by the above algorithm. The first step is *redundant* since removing it would have still resulted in a solution as demonstrated in Figure 2(b).

Accurately identifying and avoiding such redundant steps would result in faster local search without harming the algorithm's problem-solving ability. In this paper we propose a speedup learning methodology for attaining this goal. Our methodology is based on the following steps:

1. Defining the concept of a *redundant step*.
2. Acquiring this concept during off-line learning by analyzing solution paths for training problems, tagging all steps along the paths according to the redundancy definition, and using an induction algorithm to infer a classifier based on the tagged examples.
3. Using the acquired classifier to filter out redundant steps while solving unseen problems.

Our work builds on previous inductive speedup learning methods by Langley (1983); Mitchell, Utgoff, and Banerji (1984); Tadepalli and Natarajan (1996); and Briesemeister, Scheffer, and Wysotzki (1996). In Section 5 we explain the differences between these approaches and ours. Our algorithm was empirically tested on instances of real-world employee timetabling problems (ETP). The problem solver used as input is based on an algorithm developed by Schaerf and Meisels (1999), which is considered to be among the best methods for solving some large ETP instances. Our results show a significant improvement in speed for test problems that are similar to the given example problems.

Section 2 presents our general learning methodology. Section 3 describes the application of this methodology to ETP, including the set of features used by the induction algorithm. Section

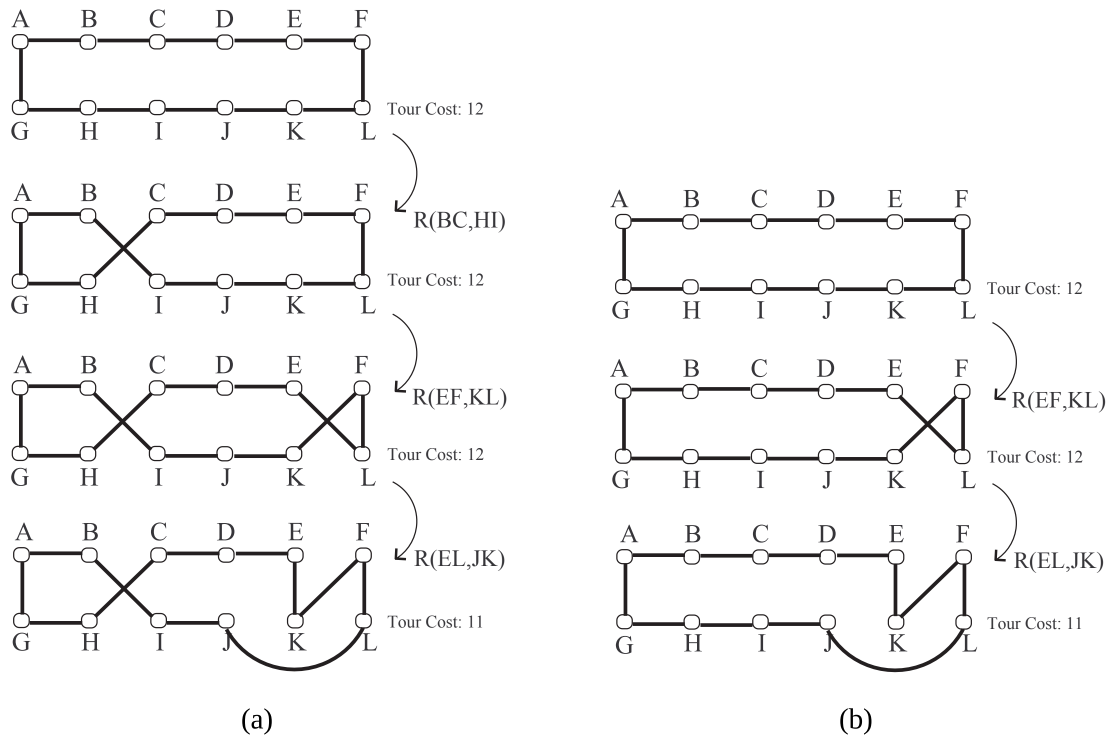

Figure 2: Applying iterative repair to a TSP instance. Figure (a) shows the sequence leading to an optimal tour. Figure (b) shows the results of applying the sequence without the first redundant operator.

4 describes the empirical evaluation of the algorithm on two real ETPs. Finally, in Section 5, we compare our approach to alternative speedup learning approaches and present our conclusions.

2. Speedup Learning for Repair Based Methods by Identifying Redundant Search Steps

In this section we give some background about local search and describe our algorithms for utilizing and learning the redundancy concept.

2.1 Local Search Algorithms

We start with a formal description of the local search approach. Let S be a set of states. Let s_0 be a start state. Let $G \subseteq S$ be a set of goal states. Let O be a finite set of operators. An operator $o \in O$ is a function from $Dom(o)$ ($Dom(o) \subseteq S$) to S. If $s \notin Dom(o)$, we define $o(s)$ to be s. Let $\sigma = o_1, \ldots, o_n$ be a sequence of operators in O. We define $\sigma(s) = o_n(\ldots o_1(s))$.

We assume that any local search technique must have a decision procedure for selecting one operator out of a given set of operators to apply in a given state. We denote this decision procedure as *select*. We can now specify more precisely the general scheme of the local search. It starts with s_0. This state is usually an approximated solution state. The *select* procedure then repeatedly selects a legal operator and applies it to the current state until a goal state (or a resource limit) is reached.

```
procedure LocalSearch($s_0$,$GOAL$,$O$)
    $s \leftarrow s_0$
    Repeat until $GOAL(s)$ do
        begin
            $O_s \leftarrow \{o \in O | s \in Dom(o)\}$ ;legal operators in $s$
            $o \leftarrow Select(O_s, s)$
            $s \leftarrow o(s)$
        end
    return $s$
```

Figure 3: A general scheme for local search.

We assume that a predicate *GOAL* that tests for membership in G is given. The formal description of the algorithm is given in Figure 3.

Many of the local search methods can be characterized by their *select* procedure. Steepest-descent hill climbing selects the operator that yields the best state with respect to a given heuristic function. Simple hill climbing selects the first operator that improves the current state. Hill-climbing methods are known to be sensitive to local minima and plateaus. Simulated annealing (Kirpatrick et al., 1983) attempts to overcome these problems by allowing selecting non-improving operators. Tabu search (Glover, 1989) avoids utilizing the last k operators used.

2.2 Local Search Using a Redundancy Filter

Local search algorithms repeatedly modify the current state in order to reach a goal state. Some of these local changes may be redundant in the sense that they do not contribute to reaching a goal state. More formally, we define a step $\langle o, s \rangle$ to be *redundant* with respect to a goal G iff for any sequence σ of operators $(o||\sigma)(s) \in G \Rightarrow \sigma(s) \in G$ ($||$ stands for the concatenation operator).

If we could identify redundant steps, we could speed up our local search by avoiding them. Figure 4 shows the local search scheme with the redundancy test incorporated. The algorithm first finds the set of operators that are legal at the current state. It then uses the redundancy predicate to filter out of this set operators that are estimated to be redundant. The *select* procedure is then applied to the reduced set. If the redundancy predicate is computationally expensive, we can switch the order of filtering. First the algorithm selects a window of best (or almost best) operators according to its *select* procedure. Then it chooses the first nonredundant operator out of this set. An example for an algorithm that uses this order of filtering is given in Section 3.

2.3 Learning the Redundancy Concept

The learning algorithm is given as input a set of training problems. It solves them and analyzes the sequences of operators that generate a solution state. It then estimates the redundancy of each step along the sequence and tags it accordingly. The tagged steps are converted to tagged feature vectors

```
procedure LocalSearch($s_0$,$GOAL$,$O$,$Redundant$)
    $s \leftarrow s_0$
    Repeat until $GOAL(s)$ do
        begin
            $O_s \leftarrow \{o \in O | s \in Dom(o)\}$ ;legal operators in $s$
            $O_u \leftarrow \{o \in O_s | \neg Redundant(o)\}$
            if $O_u$ is empty then $o \leftarrow Select(O_s, s)$
            else $o \leftarrow Select(O_u, s)$
            $s \leftarrow o(s)$
        end
    return $s$
```

Figure 4: A general scheme for local search that avoids redundant steps.

and are given to an inductive concept learner. The resulting classifer is then used as a redundancy estimator as described in Figure 4.

The algorithm in Figure 4 tests each of the legal steps for redundancy. In practice, however, we are only interested in detecting those redundant steps that might be considered useful by the problem solver. Such potentially useful steps will be selected by the *select* procedure. Steps which are selected by the *select* procedure but turn out to be redundant are called *deceptively redundant*. During the learning process, we concentrate on *deceptively redundant* steps by restricting our example set to steps along solution paths.

To estimate the redundancy of applying an operator o to a state s in a problem $p = \langle s_0, G, O \rangle$, the algorithm tests whether skipping this application would have also resulted in a goal state.[1] More formally, let $\sigma = o_0, \ldots, o_i, \ldots, o_n$ be a sequence of operators applied by the given local search algorithm to reach a solution state. Thus, $\sigma(s_0) \in G$. Let $s_i = o_0, \ldots, o_{i-1}(s_0)$. We tag $\langle s_i, o_i, p \rangle$ as *redundant* iff $o_{i+1}, \ldots, o_n(s_i) \in G$.

Note that this tagging procedure only approximates our defined concept of redundancy. We do not test whether *every* solution sequence from $o_i(s_i)$ is also a solution sequence from s_i but are rather satisfied with testing one such case. This is not as harmful as it seems since the particular solution tested is the one generated by the specific problem solver that the learner tries to improve. Moreover, it is necessary in complex domains where testing all possible solution sequences from a given state is impractical. Alternatively, we could use a limited sample of solution sequences to improve the redundancy approximation (but increase the learning cost).

The next step is to convert each tagged step to a feature vector representation. This step requires domain-specific features to be supplied by the user. In search spaces with a standard predefined structure, such as CSP spaces, it is possible to define general features. Such features will be based on the particular representation of states and operators. In Section 3.3 we will show a set of features for ETPs.

1. The method could be extended to be applied for optimization problems by testing whether skipping an operator application would have resulted in a solution with equivalent, or almost equivalent, cost.

As soon as we have a set of tagged feature vectors, we can use any of the known induction algorithms to learn the redundancy concept. Since the goal of the learning process is to speed up a problem solver, we must use a fast classifier. For example, in the experiments described in Section 4.1, we use a decision tree as our classifier. Furthermore, during classification we compute the features on demand to reduce the overall classification cost of the decision tree.

```
procedure RedundancyLearning($TrainingProblems, Solver, Features$)
    $Solver$ is a local search algorithm which returns
    a sequence to a goal state (or $\perp$ when no solution found).
    $Features$ is a sequence $f_1, \ldots, f_k$ of feature functions.
    $Examples \leftarrow \{\}$
    loop for $p = \langle s_o, GOAL, O \rangle \in TrainingProblems$
        begin
            $\sigma \leftarrow Solver(s_o, GOAL, O)$
            if $\sigma \neq \perp$
                /* analyze solution path */
                let $\sigma = o_0, \ldots, o_i, \ldots, o_n$
                loop for $j$ from 1 to $n-1$
                    begin
                        let $s_j = \langle o_0, \ldots, o_{j-1} \rangle (s_0)$
                        if $GOAL(\langle o_{j+1}, \ldots, o_n \rangle (s_j))$
                            tag $\leftarrow +$
                        else
                            tag $\leftarrow -$
                        $Examples \leftarrow Examples \bigcup \langle \langle f_1(s_j, o_j, p), \ldots, f_k(s_j, o_j, p) \rangle, tag \rangle$
                    end
        end
    Return(LearnClassifier(Examples))
```

Figure 5: The learning algorithm.

The learning algorithm is described formally in Figure 5. The algorithm assumes that the given features are functions whose domain is a step and a problem.

2.4 Generating Training Problems

The goal of a speedup learning system is to improve the potential performance of the problem solver on future problems. Most learning paradigms, such as the PAC framework (Valiant, 1984), assume that the past problems and the future problems are drawn from the same fixed distribution. This implies that the past examples are considered as a set, and the order of their appearance is ignored.

Many times, however, the order of the past examples does carry some information. In particular, for many domains, each problem is produced by a modification to the previous problem. The Dynamic Constraint Satisfaction Problems(DCSP) of Dechter and Dechter (1988) are an example of such a model. When problems are generated this way, future problems are more likely to resemble recent past problems.

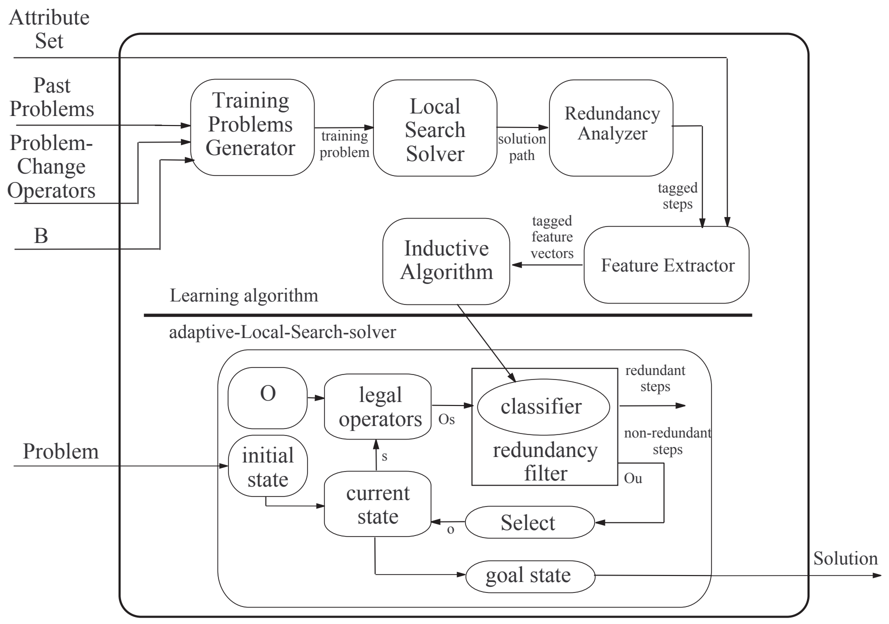

Figure 6: System description scheme.

In off-line learning, an automatic generation of training problems is often required. A reasonable approach to this task is to try and generate problems that are likely to resemble future problems. For the PAC-like problem model, it is reasonable to generate problems that are similar to the set of past problems regardless of their order. For the DCSP model described above, the next problem will most resemble the last problem. Therefore, this problem can be used as the basis for training problem generation. Assume that a set of atomic problem-change operators is given. In DCSP, for example, such operators can be implemented by the removal or addition of constraints. We define the distance d between two problems p_1 and p_2 as the minimal number of problem-change operators required to transform p_1 into p_2. We assume that at any time t, a bound B on the distance between the last problem p_t and the next problem p_{t+1} is either given or can be estimated from the sequence of externally supplied past problems $\langle p_1, \ldots, p_t \rangle$. For example, if we assume that the distance is distributed uniformly over the sequence, we can assume that

$$\max_{i=1,\ldots,t-1} \{d(p_i, p_{i+1})\} \approx \max_{i=1,\ldots,t} \{d(p_i, p_{i+1})\}.$$

Training problems can then be generated by applying to the last problem a random sequence of change operators, with length bounded by the above bound.

2.5 A General Architecture for Learning and Using a Redundancy Filter

The architecture that integrates all the above modules is illustrated in Figure 6.

3. Applying Redundancy-based Speedup Learning to ETPs

In this work we are particularly interested in speeding up the solution of ETPs. We start with a description of ETPs and continue with local search techniques used to solve such problems. We then show how to apply our algorithm to speed up these search techniques. For this we describe a set of features appropriate for ETP learning.

3.1 The Employee Timetabling Problem

Many organizations face the problem of employee timetabling, where a set of employees needs to be assigned a set of tasks in a set of fixed time shifts. The problem is typically specified by listing the set of employees with their qualifications, constraints and preferences, and the set of tasks to be fulfilled in each shift. In addition, there is commonly a set of constraints enforced by the organization. Such problems include assignment of nurses to shifts in a hospital or assignment of production workers to shifts in a factory. Related and very similar problems are school timetabling (Junginger, 1986), and examination timetabling (Carter, 1986). Schaerf (1995) gives a survey of research in this area.

We start with an exact definition of ETP, largely based on the work of Schaerf and Meisels (1999). The Employee Timetabling Problem consists of assigning employees to tasks in shifts with fixed start and end times.

Definition 1 *An* employee timetabling domain *is a triplet* $\langle E,S,T\rangle$ *where:*

- E *is a set of employees;*
- S *is a set of shifts;*
- T *is a set of tasks.*

For example, in hospital timetabling, a shift can be “Tuesday morning” and a task can be “head nurse”. The set of shifts spans over a period (a week, for example).

Definition 2 *A* timetabling assignment *is a function* $\varphi : E \times S \to T \cup \{\phi\}$.

$\varphi(e,s) = t$ means that employee e is assigned to task t in shift s. If employee e is not assigned to any task in shift s then $\varphi(e,s) = \phi$. In this case we say that e is assigned to the *null task*. Note that by the above definition an employee can be assigned to at most one task in a shift. For convenience we will represent an assignment as a two-dimensional matrix, M, where $M[e,s] = t$.

A *timetabling constraint* is a predicate over timetabling assignments. Schaerf and Meisels describe five types of such constraints:

- *A requirement constraint*. An overrequirement constraint is a triplet $\langle s,t,r_{max}\rangle$ where $s \in S$ and $t \in T$. M satisfies such a constraint iff the number of employees assigned to task t in shift s is not more than r_{max}. An underrequirement constraint is a triplet $\langle s,t,r_{min}\rangle$. M satisfies such a constraint iff the number of employees assigned to task t in shift s is at least r_{min}. For example, there must be two to four regular nurses on the Sunday evening shift.
- *An ability constraint*. This is a pair $\langle e,T_e\rangle$ where $e \in E$ and $T_e \subseteq T$. M satisfies such a constraint iff employee e is only assigned to tasks from T_e. For example, Jane can work as a nurse or as a head nurse.

- *An availability constraint.* This is a pair $\langle e, s \rangle$ where $e \in E$ and $s \in S$. M violates such a constraint iff employee e is assigned to work in shift s. For example, John cannot work on Wednesday evening.

- *A conflict constraint.* This is a triplet $\langle e, s_1, s_2 \rangle$ where $e \in E$ and $s_1, s_2 \in S$. M violates such a constraint iff employee e is assigned to work in both shifts s_1 and s_2. For example, the organization may enforce a constraint where no employee is allowed to work in two consecutive shifts. In addition, Jane is able to work on either Saturday evening or Sunday evening but not in both shifts.

- *A workload constraint.* An overload constraint is a triplet $\langle e, S_e, r_{max} \rangle$ where $e \in E$ and $S_e \subseteq S$. M satisfies such a constraint iff the number of shifts in S_e where e is assigned to work is no more than r_{max}. For example, no worker is allowed to work more than two night shifts each week. An underload constraint is a triplet $\langle e, S_e, r_{min} \rangle$ where $e \in E$ and $S_e \subseteq S$. M satisfies such a constraint iff the number of shifts in S_e where e is assigned to work is at least r_{min}.

A *timetabling problem* is a pair consisting of a domain and a set of constraints. Our goal is to find an assignment that satisfies all of the constraints. An alternative goal may be to find an assignment that optimizes some criterion such as minimizing the total wages paid to employees or equalizing the workload over the set of employees. Even et al. (1976) proved that these problems belong to the class of NP-complete problems. Despite the difficulty, many methods have been developed to cope with timetabling problems. Roughly speaking, there are two principal fields that deal with these problems: Operations Research (Burns and Carter, 1985; Nanda and Browner, 1992; Schaerf, 1995), and Artificial Intelligence (Duncan, 1990; Schaerf and Schaerf, 1995; Schaerf, 1995, 1996; Meisels et al., 1996; Meisels and Lusternik, 1997; Meisels et al., 2000, 1997; Banks et al., 1998; Schaerf and Meisels, 1999). The former includes linear programming, reduction to graph coloring, and network flow techniques. The latter includes genetic algorithms, local search, constraint satisfaction, and expert systems/rule-based approaches. In the next subsection we describe the local search technique proposed by Schaerf and Meisels. The problem solver our learning algorithm attempts to improve is based on this technique.

3.2 Local Search for ETPs

There are quite a few local search algorithms that were applied to ETPs (Schaerf and Schaerf, 1995; Schaerf, 1996; Schaerf and Meisels, 1999; Meisels and Schaerf, 2003). The problem solver targeted by our speedup learning algorithm is based on the techniques developed by Schaerf and Meisels (1999) and applied successfully to real-life ETPs.

Since the problem-solving technique is based on the local search algorithm described in Figure 3, we need only to specify the following components: the set of search states, the set of operators, the GOAL predicate, and the decision procedure *select* which chooses the next operator to apply. The rest of this section describes these components.

Each search state is a complete *timetabling assignment*. The assignment is represented, as described in the previous section, by a rectangular $m \times n$ integer-valued matrix M with a row for each employee and a column for each shift. The search space includes three types of parameterized operators:

Start assignment table

	S_1	S_2	S_3	S_4	S_5
E_1	T_1		T_2		
E_2		T_3			T_1
E_3		T_2		T_2	
E_4	T_3		T_3		T_2

Sequence of applied operators

SWAP$_{S_1,E_1,E_4}$
SWAP$_{S_5,E_2,E_4}$
DELETE$_{S_2,E_2}$
INSERT$_{S_5,E_1,T_2}$

End assignment table

	S_1	S_2	S_3	S_4	S_5
E_1	T_3		T_2		T_2
E_2					T_2
E_3		T_2		T_2	
E_4	T_1		T_3		T_1

Figure 7: The effect of applying a sequence of operators to an assignment table.

1. *Swap.*[2] This operator is instantiated by three parameters: shift s, employee e_1, and employee e_2. The instantiated operator swaps the tasks of e_1 and e_2 in shift s. We denote such an operator by SWAP$_{s,e_1,e_2}$.

2. *Insert.* This operator is instantiated by three parameters: shift s, employee e, and a non-null task t. The instantiated operator assigns task t to employee e in shift s. It is not legal to apply the operator if e is already assigned a non-null task in shift s. We denote such an operator by INSERT$_{s,e,t}$.

3. *Delete.* This operator is instantiated by two parameters: shift s and employee e. The instantiated operator assigns the null task to employee e in shift s. It is not legal to apply the operator if e is already assigned the null task in shift s. We denote such an operator by DELETE$_{s,e}$.

Figure 7 shows the results of applying a sequence of operators on an example state (timetabling assignment). Our goal is to search for a state without any violated constraints. Thus, the goal predicate checks whether the total number of violated constraints is zero. At each iteration, the *select* operator must choose one operator from the set of legal operators for the current state. Our strategy first randomly selects one of the three operator types. It then applies a heuristic function on the states resulting from applying all[3] operators of the selected type and selects the one with best heuristic value. The heuristic used is the minimum-conflict function (Minton et al., 1992), which counts the number of violated constraints. Schaerf and Meisels used a more complicated heuristic that also includes some look-ahead component and a shifting penalty scheme to dynamically determine the weight of each constraint type. We preferred the simpler heuristic because it proved to be faster on the tested domains and therefore a more challenging choice for speedup learning. Frequently, this heuristic results in a relatively large set of best operators. In such cases we use random tie-breaking. We refer to the algorithm described in this subsection as the GSHC* scheduler. The algorithm is formally described in Figure 8.

2. The original name given by Schaerf and Meisels is Replace. A swap operator is not the same as replace, but a simple extension of it.
3. Schaerf and Meisels explored various strategies for selecting a subset of the legal operators of the selected type. The strategy we employ looks at all the operators of the chosen type and corresponds to their GSHC strategy.

```
procedure LocalSearch($s_0$,$GOAL$,$O$)
    $s \leftarrow s_0$
    Repeat until $GOAL(s)$ do
        begin
            $O_s \leftarrow \{o \in O | s \in Dom(o)\}$ ;legal operators in $s$
            $O_h \leftarrow \{o \in O_s | h(o) \text{ is minimal}\}$
            $o \leftarrow RandomElement(O_h)$
            $s \leftarrow o(s)$
        end
    return $s$
```

Figure 8: The GSHC* scheduler algorithm.

3.3 Features for ETP Domains

In the previous section we described the redundancy-learning algorithm that induces a classifier based on tagged examples. For ETP, each example is a triplet $\langle M, o, p\rangle$ where M is an assignment table representing a state, o is an operator, and p is an ETP instance. In order to generalize from a training problem to future unseen problems, we convert the examples to feature vector representation. The features are usually domain-dependent. In this subsection we describe a set of features for the ETP domain. These features have been selected out of a larger group by experimenting[4] on test sets.

The features can be divided to two groups: *atomic* features and *compound* features. The atomic features can be categorized according to the parts of the example they pertain to:

- *State-problem attributes*. These are defined over a state M. They try to capture its characteristics with respect to the problem definition p. Each of the attributes of this group sums, for one constraint type, the values of its violations. For example, the *maximum-employee violations* attribute computes, for each violation of the *overrequirement* constraint, the difference between the number of employees assigned to the given task in the given shift and the maximum allowed. It then sums these differences.

  A high value for this feature may indicate that *swap* or *insert* operators are redundant since they do not reduce the number of such violations.

- *Operator-state-problem attributes*. While attributes of the previous group consider the whole assignment table M, attributes in this group consider only the part of the table defined by the instantiation of the variables of operator o. Thus, such features focus on the part of the state that may be more relevant to the specific operator considered.

  For example, assume that the operator o is $\text{INSERT}_{s,e,t}$. The *operator-maximum-employee violations* attribute sums the number of redundant employees, according to overrequirement constraints in p, who are scheduled to perform task t in shift s.

4. The experiments were performed on test sets other than those used for the final evaluation.

A positive value for this attribute may indicate that the specific insertion is redundant since there are already enough employees in the given shift assigned to perform the given task. Note that it is quite possible that the specific insert operator is *not* redundant despite a positive value for this attribute. For example, it is possible that the particular employee is underloaded and the insertion allows an overloaded employee to be freed from this shift.

- *State-operator attributes*. These attributes are defined over the state M and the operator o. These attributes attempt to characterize the relations between the state and the operator regardless of the particular problem p. For example, assume that the operator o is $\text{DELETE}_{s,e}$. The *employee workload* attribute counts the number of shifts in which employee e is scheduled to work. A low value may indicate that the employee has a low workload and that the operator may be redundant.
- *Operator-problem attributes*. These attributes are defined over the problem p and the operator o. These attributes attempt to characterize operator o with respect to problem p regardless of the particular state M. For example, assume that the operator o is $\text{INSERT}_{s,e,t}$. The *employee-workload constraint degree* attribute counts the number of workload constraints in which employee e participates.
- *Problem-base-problem attributes*. These attributes try to characterize the problem itself. Since there are too many possible features for describing an ETP, we restrict ourselves to features that describe the *differences* between the problem and the base problem.

The class of *compound attributes* combines pairs of continuous atomic attributes by arithmetic relations of their values. An example is the *attribute-i-bigger-than-attribute-k* attribute, which tests whether the value of a specific attribute is bigger than the value of another specific attribute. The complete list of attributes can be found in Appendix A.

3.4 Problem-change Operators for Generating Training ETPs

For the ETP domains used in this paper, we made several reasonable assumptions that determine the set of problem-change operators. We assume that the only type of change is a replacement of an employee by a new one. This includes changes to certain constraints of an employee. We assume that the difference between the new employee and the replaced employee is restricted as follows:

1. The set of tasks T_{new} defined by the ability constraints of the new employee is similar to that of the replaced employee (T_{old}). Specifically, we assume that either $T_{old} \subseteq T_{new}$ or $T_{new} \subseteq T_{old}$ and
$$||T_{new}| - |T_{old}|| \leq \varepsilon_q |T_{old}|.$$
This means that the new employee is either more qualified or less qualified than the replaced employee, but the difference in their qualifications is bounded. Thus, for example, such a restriction could be used to prevent replacing a nurse with an engineer.

2. The number of shifts for which the new employee is available, $|S_{new}|$, defined by the availability constraints, is similar to the number of shifts for which the replaced employee is available, $|S_{old}|$. Specifically, we assume that
$$||S_{new}| - |S_{old}|| \leq \varepsilon_a |S_{old}|.$$

```
procedure LocalSearch($s_0$,$GOAL$,$O$,redundant)
    $s \leftarrow s_0$
    Repeat until $GOAL(s)$ do
        begin
            $O_s \leftarrow \{o \in O | s \in Dom(o)\}$ ;legal operators in $s$
            $h_{min} \leftarrow min_{o \in O_s} h(o)$
            $O_h \leftarrow \{o \in O_s | h(o) \leq h_{min}(1+\varepsilon)\}$
            $o \leftarrow$ first non-redundant element in $O_h$
            if $o$ is null then $o \leftarrow RandomElement(O_h)$
            $s \leftarrow o(s)$
        end
    return $s$
```

Figure 9: The adaptive-GSHC* scheduler algorithm.

Such a restriction could be used, for example, to prevent replacing a full-time employee with a part-time employee.

We assume that the other two types of constraints, workload and conflict constraints, are based on organizational rules and are therefore not likely to be changed often. Of course, the framework presented here does not depend on these assumptions.

3.5 Utilizing the Learned Classifier for ETP

After the classifier is learned, it can be used for detecting redundant steps as described in the algorithm in Figure 4. In this algorithm, we first filter the redundant operators out of the set of legal ones and then select the best operator. This approach is reasonable if the cost of applying the redundancy classifier is very low compared to the selection process. An application of a straightforward implementation of the minimum-conflict heuristic may carry a high computational cost in the ETP domain. Our implementation, however, follows an incremental approach, which yields a very efficient heuristic computation. Since a heuristic evaluation is now much cheaper than a redundancy classification, we first find the best operators according to the heuristic and then return the first best operator that is not redundant. This version of the algorithm is described in Figure 9.

4. Experimental Evaluation

To test the effectiveness of our algorithm, we experimented with it in two real ETP domains. One, the *hospital department problem*, is timetabling of nurses in a hospital department. The other, the *factory problem*, is timetabling of production workers in a factory.[5] In the *hospital department problem*, there are 57 shifts in each week's timetable and 29 employees to be assigned to a total of 105 tasks over 7 days. The *factory problem* has 21 weekly shifts (three per day, 8 hours each)

5. The two problems are based on the problems described by Schaerf and Meisels (1999) available at `http://www.cs.bgu.ac.il/~am/`.

	hospital problem	factory problem
Employees	29	50
Shifts	57	21
Tasks	11	3
Employee Groups	11	2
Shift Groups	19	3
Required Assignments	105	243
Constraints	10766	3561

Table 1: Quantitative information about the two problem instances used for experimentation.

and 50 employees to be assigned to a total of 243 tasks. Each of these problems includes several thousands of constraints. This data is summarized in Table 1. Each of the two instances is used as a base problem from which training and testing problems are generated. The exact specification of the two instances is listed in the online Appendix.

When representing the problems as a search space according to the scheme described in Section 3.2, the computed branching factor is about $42,000$ for the *hospital department problem* and about $30,000$ for the *factory problem*. The average actual branching factor (considering only nontrivial legal operators available in each search state) is 5000 for the *hospital department problem* and 6735 for the *factory problem*.

4.1 Experimental Methodology

Most of the experiments described here consist of a learning phase and a testing phase. We assume the model described in Section 2.4, where the learner is given the last problem, and a bound B on the distance between the last problem and the next problem.

For testing we use a set of 100 random problems with distance bounded by B from the input problem. We use the same set throughout our experiments. The problem solver we use for testing is the GSHC* scheduler described in Section 3.2. Since we assume that the next problem to be solved is similar to the last problem, it is reasonable to exploit the last solution for the new problem. One of the obvious ways of doing so is by using the last solution as an initial state for the next problem. We therefore use this modified scheduler with the same initial solution for all the tests.

For training, the learner gets as input the base problem, the bound B, a set of change operators and a set of features. The learner generates training problems using the algorithm described in Section 2.4. To reduce variability, we use the same training set for all the experiments. The learner generates from each solution path a set of tagged examples as described in Figure 5. The examples are given to the $C4.5$ induction algorithm (Quinlan, 1986, 1993), which generates a decision tree classifier. This classifier is then used by the adaptive-GSHC* scheduler algorithm for solving test problems.

To test the statistical significance of our results, we use the Wilcoxon matched-pair signed-rank test (Sachs, 1982). Unlike the T test, the Wilcoxon test is distribution free. One problem in comparing the speed of problem solvers is the bias due to the time-bound imposed by the experimenter (Segre et al., 1991). To address this problem we use the approach suggested by Etzioni and Etzioni

(1994) that treats the results of such experiments as censored data. In particular, we used the most conservative approach. This approach considers as infinite the solving time of any problem in which the adaptive-GSHC* scheduler algorithm stopped due to a resource bound.

4.1.1 DEPENDENT VARIABLES

The goal of speedup learning is to improve the performance of the given problem solver. Therefore, the most important dependent variables are those measuring that improvement. In addition, to gain some insight about the learning process, we measure various aspects of the learned classifiers:

- *Speedup factor.* The ratio between the speed of the problem solver after learning (adaptive-GSHC* scheduler) and the speed before learning (GSHC* scheduler). The speed is measured by CPU time.[6]
- *Atomic operator applications*. The number of atomic operator applications, where *swap* is considered as either two or four atomic applications. This variable is independent of implementation and can therefore be used by other researchers for comparison.
- *Success rate*. The portion of the test set that is solved by the tested algorithm.
- *Accuracy on test examples*. The average accuracy of the learned concept on the test set.
- *Attribute class utility*. The contribution of each class of attributes to the performance of the learning algorithm. This variable is measured by comparing performance with and without the tested set.

4.1.2 INDEPENDENT VARIABLES

There are many independent variables that affect the performance of the learning system. For the experiments described here we look at the following variables:

- *Test distance variables*. These variables determine the degree of difference between the base problem and the test problems. To generate a test problem, a sequence of problem change operators is applied to the base problem. Such a sequence is determined by the parameters listed below (see Subsection 2.4):
  - *Operator-distance variable*. This is the length of the problem-change operator sequence (bounded by B, as mentioned in Subsection 2.4).
  - *Problem-change operator parameters*. The problem-change operator for an ETP includes two free parameters.
    - ε_q. This parameter determines the degree of difference between the qualification set of a replaced employee and that of the replacing employee. A smaller value indicates greater similarity between the two sets.
    - ε_a. This parameter determines the degree of difference between the availability set of a replaced employee and that of the replacing employee. A smaller value indicates greater similarity between the two sets.

6. We normalize the CPU time by the time required for one operator application to reduce variance due to different hardware and system load.

scheduler	Time [Seconds]		Success Rate		Atomic Operator Applications	
	with	without	with	without	with	without
GRHC* scheduler	45	191	89	0	4150	18666
GRRSHC* scheduler	88	252	26	9	16976	39280
GRSSHC* scheduler	86	263	93	92	18633	50990
GSHC* scheduler	46	296	95	76	4260	28073

Table 2: Comparision of various search techniques when applied to the *factory problem*. The comparison is for time, success rate and atomic operator applications. The *with* columns give the performance of the appropriate scheduler when the solution to the base problem is used as the initial state. The *without* columns give the performance of the appropriate scheduler that finds an initial state by some greedy procedure.

The default values for these variables are $\varepsilon_q = 0.5$ and $\varepsilon_a = 0.4$. The operator distance is uniformly chosen from $[1\%, 10\%]$. For example, for the *factory problem* this range is $[1\%, 5\%]$.

- *Training distance variables*. These variables determine the degree of difference between the base problem and the training problems. The default values for these variables are $\varepsilon_q = 0.5$ and $\varepsilon_a = 0.4$; and the operator distance is uniformly chosen from $[1\%, 10\%]$. For example, for the *factory problem* this range is $[1\%, 5\%]$.

- *Learning resources*. The most important independent variable for a learning system is the amount of resources invested in learning. The most common method of measuring learning resources in off-line learning is by counting the number of training problems. In our case this would not be a good measurement since there is a great variability in the difficulty of the training problems. Therefore we use instead the total number of tagged steps used in the classifier construction.

- *Redundancy filter selectivity*. This variable, which has the range [0,1], determines when a tree leaf should be tagged as *nonredundant*. If the portion of nonredundant examples in a leaf is above that value, the leaf is tagged as *nonredundant*; otherwise it is tagged as *redundant*. The default value of this variable is 0.5.

4.2 Performance Before Learning

To give our learning methodology a fair test, we were interested in a base algorithm that has both high success rate and high speed. The GSHC* scheduler algorithm selects from O_s the set of all legal operators. Schaerf and Meisels (1999) propose several methods for reducing the size of O_s, and therefore reducing the branching factor. We refer to the variants of the GSHC* scheduler that use these methods as follows:

- GRHC* scheduler. Select a random operator. Apply it if it improves the evaluation of the state.

- GRRSHC* scheduler. Select a random shift. Select the best operator pertaining to the selected shift.

	hospital problem	factory problem
Atomic operator applications	4500	4260
Success rate	70.0%	95.3%
Violations at end	0.59	0.05
Best operators per iteration	80	175
Solution length	2155	1774

Table 3: Comparing two test sets by applying to them the GSHC* scheduler. Each number represents the mean over the test sets.

- GRSSHC* scheduler. Select a random shift and a random employee. Select the best operator pertaining to the selected shift and employee.

Table 2 compares the performance of the four variants for the test set of the *factory problem* in terms of time,[7] success rate and atomic operator applications.

We can see that the GSHC* scheduler indeed has the best combination of speed and success rate. In addition, the table shows that it is beneficial to use a previous solution when the previous and current problems are similar. We therefore use this algorithm as our base algorithm to be improved. Table 3 shows the performance of the GSHC* scheduler on the two test sets. The table also reports the the total number of constraint violations for unsolved problems and the average number of best operators in each iteration (O_h in Figure 8).

4.3 Learning Curve

Our basic experiment tests the effect of the allocated learning resources on the utility of the adaptive-GSHC* scheduler. The learning algorithm generates solution paths for the training problems, analyzes them and generates a set of tagged examples. These examples are used by the induction algorithm to build a classifier. We allocated enough learning resources to generate $150,000$ examples.

Before using the generated classifiers, we measured their accuracy on an independent set of tagged examples and found out that it is quite accurate: 89% on the *hospital department problem* and 76% on the *factory problem*. Accurate classification, however, is not sufficient for speedup. It is quite possible that the additional cost associated with the classifier applications will outweigh its benefits. The ultimate test should therefore be the speedup gained when using the learned classifier. We run the induction algorithm 5 times - with $0\%, 25\%, 50\%, 75\%$ and 100% of the example set. Each time we compare the performance of the adaptive-GSHC* scheduler to that of the GSHC* scheduler. The learning curves showing the average speedup factor as a function of the learning resources for both domains are shown in Figure 10.

The graphs show typical learning curves with speedup factors of up to 1.5 and 3.1 for the *hospital department problem* and the *factory problem* respectively. According to the Wilcoxon test with the adjustment for censored data (as described in Subsection 4.1), the algorithm after learning (with 100% of the training data) significantly outperforms the algorithm before learning (with p=0.01).

7. The time reported is for Java running in Windows 95 on an IBM PC with a 500MHz Pentium III Processor.

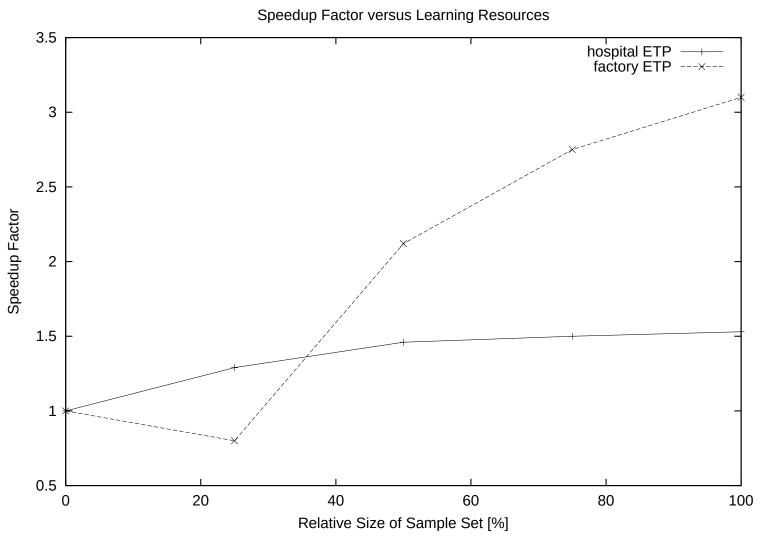

Figure 10: The effect of learning resources on the speedup factor. Each number represents the mean over the test set.

	hospital problem	factory problem
GSHC* scheduler	70%	95%
adaptive-GSHC* scheduler	71%	99%

Table 4: The success rate before and after learning.

Of course, these results would have been less impressive had the increasing speed been accompanied by reduction in the success rate of the problem solver. Table 4 shows that this is not the case. The success rate even increased slightly.

Table 5 shows the number of atomic operators before and after learning. We can see that for the *factory problem*, for example, the improvement measured by operator applications is a bit more than that represented by the speedup factor shown above (3.5 vs. 3.1). This difference is due to the overhead incurred by the classification process.

The third row of the table shows the results for the adaptive-GSHC* scheduler that, instead of using a learned classifier, uses an oracle that always gives the correct classification. (We use the same procedure utilized for tagging.) The number in this row can serve as a limit on the best possible improvement that can be achieved by our proposed algorithm.

One interesting phenomenon in the above result is that, while the accuracy for the *hospital department problem* was higher than the accuracy for the *factory problem*, the actual speedup for the *factory problem* was higher. One possible explanation is that the effect of false negatives on performance is much stronger than the effect of false positives. False negative (i.e., a redundant step that is mistakenly identified as non-redundant) classification adds the cost of a redundant operator application. False positive (a non-redundant operator application that is mistakenly identified as redundant) classification adds the cost of additional classification. Since in our domain the cost of an operator application is proportional to the branching factor and is therefore much higher than the

	hospital problem	factory problem
GSHC* scheduler	4530	4260
adaptive-GSHC* scheduler	3020	1200
adaptive-GSHC* scheduler with oracle	1440	920

Table 5: The number of atomic operator applications before and after learning. The third row shows the results for the adaptive-GSHC* scheduler with oracle.

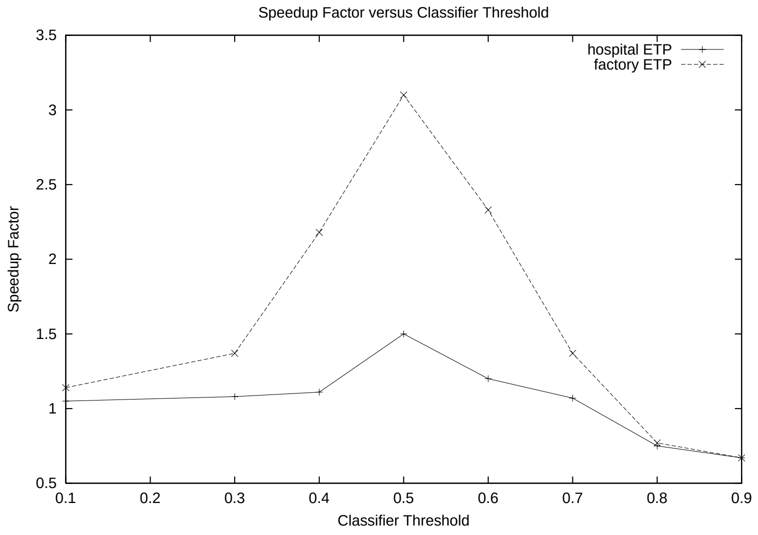

Figure 11: The effect of the filter selectivity on the speedup factor.

cost of classification, the effect of a false negative is more significant. We measured these errors and found that indeed for the *factory problem* the false negative error rate is smaller than for the *hospital department problem*– 11% versus 18%.

4.4 The Effect of the Filter Selectivity on Performance

At the end of a learning phase, the learned classifier is used to filter out steps that are estimated to be redundant. In this experiment we test the effect of the sensitivity of the filter on the system performance. Figure 11 shows the speedup factor as a function of the filter selectivity.

The graphs for the two problems show a similar behavior. A low value for the filter selectivity variable means a less selective filter. Such a filter will consider most of the steps as useful. Thus the likelihood of a redundant step passing through the filter and being selected is high. This is the reason for the lower performance with low values for the variable (values less than 0.5).

A high value for the variable means a more selective filter that estimates most of the steps to be redundant. The graphs show that for such high values the performance of the system deteriorates. This result seems counterintuitive at first since higher selectivity is presumed to identify nonredundant steps with higher accuracy. Looking at the algorithm described in Figure 9, however, reveals the reason for this phenomenon. If the filter does not find any useful step, it resorts to the default

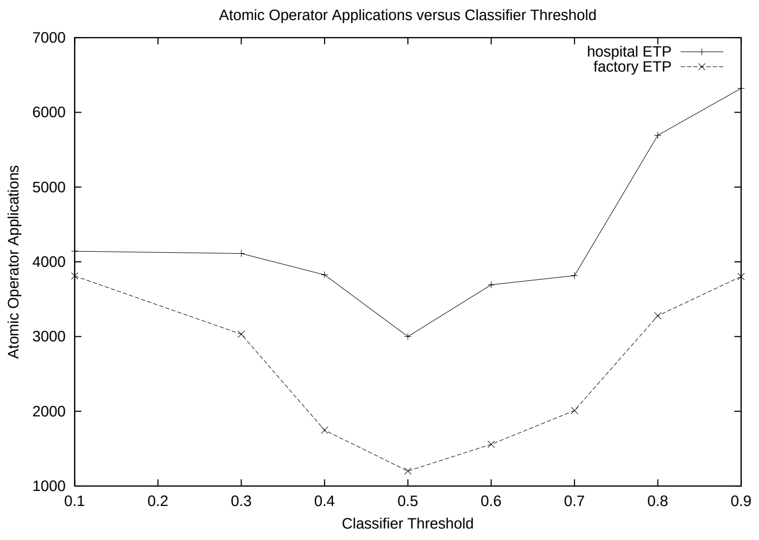

Figure 12: The effect of filter selectivity on atomic operator applications.

behavior of choosing a random step. High selectivity increases the likelihood of such incidents, which increases the chance of choosing a redundant step.

Figure 12 shows the number of atomic operator applications as a function of the filter selectivity. The graphs show patterns similar to those of the speedup factor. There is one interesting phenomenon in these graphs. Look at the graph for the *factory problem* at points 0.1 and 0.9. When the operator applications are compared, the performance for 0.9 is the same as for 0.1. However, when the same points on the speedup factor graph are compared, the performance for 0.9 is much worse.

A careful look at the algorithm (in Figure 9) will help explain this behavior. As the selectivity increases, so does the number of times that the classifier is applied. The overhead incurred by classification increases as well. Figure 13 shows the classifier overhead as a function of the filter selectivity. Indeed, while for the selectivity default value (0.5) the classification overhead is low (about 5%), for a value of 0.9 it increases to above 17%. The reason for the difference in overhead for the *factory problem* and the *hospital department problem* is that the percentage of redundant steps differs for each problem. Looking at Table 5 one can calculate the percentage of operators that are not redundant: 22% for the *factory problem* and 32% for the *hospital department problem*. Therefore, for the *factory problem*, the filter is applied more times before a nonredundant operator is found.

4.5 The Effect of the Problem Spread on the Learning Utility

The performance of a learning system usually depends on the similarity between its training and testing problems. In Section 2.4 we describe the model we assume for the input problems. The distance between subsequent input problems is assumed to be bounded by a parameter *B*. We assume that the learning algorithm is given *B* and that it uses the last problem in the sequence (the *base* problem) to generate training problems of distance bounded by *B*. The default value of *B*

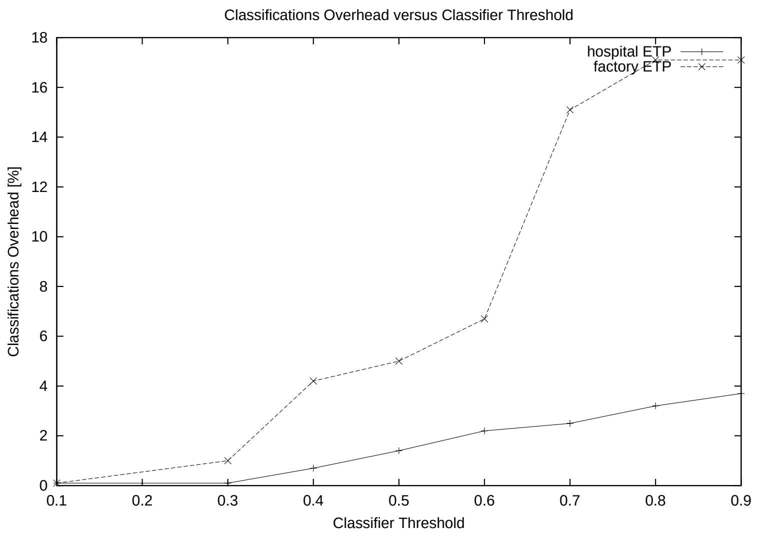

Figure 13: The effect of filter selectivity on the classifications overhead. This overhead is given by the portion of total time that is consumed by the classifications.

B	hospital problem	factory problem
3%	1.4	2.2
10%	1.5	3.1
15%	1.2	2.55

Table 6: The effect of *B* on the speedup factor.

used throughout the experiments is 10%. Note that the actual distance between training and testing problems can be up to 2*B*, since both types of problems are sampled from a circle of radius *B* around the base problem.

To test the effect of *B* on the performance of the learning system, we repeated our learning experiment with $B = 3\%$ and $B = 15\%$. The results are presented in Table 6 and Table 7 .

For $B = 15\%$, the benefit of learning decreases as expected since the testing problems are less similar to the training problems. For $B = 3\%$ we anticipated that the performance would be better since the training and testing problems are more similar. The results, however, show that the speedup factor was no better. To understand the reason for this phenomenon, recall that the problem solver starts with the solution of the base problem as its initial state. Therefore, the solver will be able to efficiently solve problems that are very similar to the base problem even without learning. Consequently, the learning process has a narrower margin for improvement.

Other parameters that affect the difference are ε_q and ε_a, which determine the similarity between a replaced employee and the replacing employee within a given operator. We tested the effect of these parameters on performance and discovered that larger values for these parameters indeed yield problems with greater differences (for the same operator distance) and therefore worse performance.

	B=3%	B=10%	B=15%
GSHC* scheduler	94.5%	95.3%	95.3%
adaptive-GSHC* scheduler	97.8%	99.5%	98.1%

Table 7: The effect of *B* on the success rate. The base problem is the *factory problem*.

attribute class	Speedup factor	Relative contribution
problem-base-problem	2.71	14%
state-problem	1.47	110%
operator-state-problem	1.90	63%
state-operator	2.06	50%
operator-problem	1.90	63%
complex	2.44	27%

Table 8: The first column gives the speedup factor obtained by using the reduced set: the full set, without any of the attributes belonging to the appropriate class. The second column gives relative contribution of the reduced set.

4.6 Attribute Class Utility

One of the most important factors in learning system performance is the set of attributes used. In this subsection, we test the contribution of the different classes of attributes to the performance of our learning algorithm. The utility of different classes of attributes is evaluated by constructing the classifiers with and without the evaluated attribute class and comparing the speedup obtained by the two classifiers.

Table 8 shows the results obtained for the *factory problem*. The right column specifies the relative contribution defined as

$$RelativeContribution(AttSet) = \frac{SF(FullSet) - SF(FullSet - AttSet)}{SF(FullSet - AttSet)} \cdot 100,$$

where $SF(A)$ is the speedup factor achieved when using attribute set A. The speedup factor of the full set for this problem is 3.1. The most important observation is that the utility of all of the classes is positive. This justifies the use of all the suggested classes of attributes.

5. Conclusions and Related Work

This paper presents a general framework for using machine learning methods to speed up a given local search problem solver. Our methodology is based on the observation that many solution steps generated by repair-based solvers are redundant in the sense that the changes they make do not contribute to the final solution. We achieve speedup by filtering out steps that are estimated to be redundant. The redundancy filter is induced based on tagged examples generated from solution paths of training problems. A step is tagged as *redundant* if skipping it would still result in a solution path.

We instantiate the above framework for employee-timetabling problems by supplying a set of step features that is used to represent examples. Our algorithm was empirically tested on two instances of a real world ETPs. While our base problem solver was quite efficient to start with, our improved problem solver was still able to speed it up by a factor of 3.1. Moreover, its problem-solving ability did not deteriorate and in fact was improved.

Like other learning approaches, our algorithm is more successful when the training problems and the testing problems are similar. We presented a method for automatic generation of training problems that are variants of the previous testing problem. This method assumes the DCSP model where each problem is produced by a modification to the previous problem. This model is true in many real-life applications. In hospital ETP, for example, a new timetable is generated weekly due to slight changes in the employee constraints. Another example is the VLSI layout problem where during the design period most layout problems are a slight variation of the previous design.

Previous work on speedup learning consists of two main approaches. One technique is to learn new *macro operators* (Korf, 1985; Iba, 1989; Markovitch and Scott, 1993; Tadepalli and Natarajan, 1996; Finkelstein and Markovitch, 1998) by composing sequences of original operators, that allow the problem solver to take "big steps" when searching. The other technique is to learn some form of control knowledge, that will help determine which action to try next (Smith, 1983; Langley, 1983; Mitchell et al., 1984; Minton, 1988; Gratch and Chien, 1996; Tadepalli and Natarajan, 1996; Briesemeister et al., 1996; Kaelbling et al., 1996; Morris et al., 1997a,b; Su et al., 2001). Control knowledge can take many forms, including *evaluation functions* (Zhang and Dietterich, 1995; Gratch and Chien, 1996; Morris et al., 1997a; Boyan and Moore, 2000) and *control rules* (Langley, 1983; Mitchell et al., 1984; Minton, 1988; Briesemeister et al., 1996; Kaelbling et al., 1996; Tadepalli and Natarajan, 1996). This knowledge, which can be used for *operator selection*, *operator rejection* or *operator ordering* (Minton, 1988), may be acquired by reinforcement learning (Zhang and Dietterich, 1995), weight adjustment (Morris et al., 1997a), linear regression (Su et al., 2001), inductive learning (Mitchell et al., 1984) and explanation-based learning (Minton, 1988).

Our method uses inductive learning to acquire rejection rules. Our algorithm uses the steps along the solution path as examples for the inductive algorithm. Several works (Langley, 1983; Mitchell et al., 1984; Tadepalli and Natarajan, 1996; Briesemeister et al., 1996) have used a similar approach for learning control rules. The most important difference between these approaches and ours is their treatment of all the solution steps as desired moves. While this approach is appropriate for general search algorithms, especially those that find low-cost solutions, it is inappropriate for local search. In local search, the solution path (which is also the search path) is usually much longer than the shortest solution. Therefore, unlike the above methods, our method considers a large percentage of the solution steps to be undesired moves.

One important issue to discuss is the conditions under which our algorithms are effective. The utility of our method is the cost saved by the filter minus the cost associated with its application. For each redundant step identified by the redundancy filter and therefore avoided by the problem solver, we save the cost of one step. This cost involves applying all the legal operators and computing the heuristic value for each of the resulting states. In repair-based search, where the branching factor is very high, this saving is very significant. The cost associated with the filtering method is the cost of applying the filter times the number of applications required until the first non-redundant operator is identified.

More formally, let L_{old} be the length of the solution sequence achieved by a standard local search algorithm. Let bf be the branching factor, c_o the cost of an operator application and c_h the cost computing the heuristic value of a state. The cost associated with one solution step is therefore $c_s = bf \cdot (c_o + c_h)$. The total search cost will therefore be $L_{old} \cdot c_s$.

Let p_u be the probability that the filter finds an operator to be non-redundant. The expected number of times of applying the filter to a sequence of operators until a non-redundant operator is found is therefore $E_f = 1/p_u$ and the total cost of using the filter per one solution step is $c_f = E_f \cdot c_c$ where c_c is the cost of one call to the classifier. Let L_{new} be the length of the solution sequence found by our algorithm. The total search cost will be $L_{new} \cdot (c_s + c_f)$. The speedup factor achieved by the adaptive method will therefore be

$$\frac{c_s}{c_s + c_f} \cdot \frac{L_{old}}{L_{new}}.$$

The right term increases as the adaptive algorithm manages to avoid more redundant steps. This depends on the density of redundant steps produced by the particular solver and the ability of the classifier to identify them. The left term increases as the overhead of the filtering *per solution step* decreases.

For example, in our factory domain, typical numbers are (roughly) $L_{old} = 1700$, $L_{new} = 500$ and $c_f \approx 0.05 c_s$. This last ratio may seem surprisingly small but note that for each solution step the classifier is applied only 3.5 times on average ($p_u = 0.28$) while c_s is relatively large due to the large branching factor ($bf = 30,000$). The speedup factor is therefore roughly 3.23 which is consistent with the empirical results achieved.

Another issue to consider is the overhead associated with learning. While this overhead may be high, its cost can be neglected in many real world applications. Take for example the hospital domain. The scheduler is typically invoked once a week with the modified constraints to design the timetable for the coming week. Thus, the period between two consecutive calls can be used for offline learning.

The work presented here builds on the above approaches in inductively acquiring control knowledge based on solution paths. However, to make it appropriate for repair-based methods, we introduce the new concept of operator redundancy together with algorithms for learning and using it. This concept allows us to identify and avoid operator applications that would be viewed by other methods as deceptively useful. In addition, we supply a well-designed set of features enabling the inductive learning of this concept in the ETP domain.

Speedup learning techniques have been tried mainly on artificial and toy domains. Recently, several works have attempted to apply such techniques to complex real-world problems. This work contributes to the field of speedup learning by extending the applicability of control-rule learning to repair-based search methods that are often used for complex real-world tasks such as ETPs.

References

D. Banks, P. van Beek, and A. Meisels. A heuristic incremental modeling approach to course timetabling. In *Proceedings of the Canadian Conference on Artificial Intelligence*, pages 16–29, 1998.

J. A. Boyan and A. W. Moore. Learning evaluation functions to improve optimization by local search. *Journal of Machine Learning Research*, 1:77–112, 2000.

L. Briesemeister, T. Scheffer, and F. Wysotzki. A concept based algorithmic model for skill acquisition. In *Proceedings of the First European Workshop on Cognitive Modeling*, 1996.

R. Burns and M. Carter. Work force size and single shift schedules with variable demands. *Management Science*, 31:599–607, 1985.

M. W. Carter. A survey of practical applications of examination timetabling algorithms. *Operations Research*, 34(2):193–202, 1986.

A. Dechter and R. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 37–42, 1988.

T. Duncan. Scheduling Problems and Constraint Logic Programming: A Simple Example and its Solution. Technical Report AIAI-TR-120, Artificial Intelligence Applications Institute, University of Edinburgh, 1990.

O. Etzioni and R. Etzioni. Statistical methods for analyzing speedup learning experiments. *Machine Learning*, 14(1):333–347, 1994.

S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal on Computing*, 5(4):691–703, 1976.

L. Finkelstein and S. Markovitch. A selective macro-learning algorithm and its application to the nxn sliding-tile puzzle. *Journal of Artificial Intelligence Research*, 8:223–263, 1998.

F. Glover. Tabu search– part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

J. Gratch and S. Chien. Adaptive problem-solving for large-scale scheduling problems: A case study. *Journal of Artificial Intelligence Research*, 4:365–396, 1996.

G. A. Iba. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3(4): 285–317, 1989.

W. Junginger. Timetabling in Germany – A survey. *Interfaces*, 16(4):66–74, 1986.

L. P. Kaelbling, M. L. Littman, and A. P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

S. Kirpatrick, C. Gelatt, Jr., and M. Vecchi. Optimization by simulated annealing. *Science*, 220: 671–680, May 1983.

R. E. Korf. *Learning to Solve Problems by Searching for Macro-Operators*. Research Notes in Artificial Intelligence. Pitman, 1985.

P. Langley. Learning effective search heuristics. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 419–421. Morgan Kaufmann, 1983.

S. Lin and B. Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.

S. Markovitch and P. D. Scott. Information filtering: Selection mechanisms in learning systems. *Machine Learning*, 10(2):113–51, 1993.

A. Meisels, E. Gudes, and G. Solotorevsky. Employee timetabling, constraint networks and knowledge-based rules: A mixed approach. *Lecture Notes in Computer Science*, 1153:93–105, 1996.

A. Meisels, E. Gudes, and G. Solotorevsky. Combining rules and constraints for employee timetabling. *International Journal of Intelligent Systems*, 12:419–439, 1997.

A. Meisels and N. Lusternik. Experiments on networks of employee timetabling problems. In *Proceedings of the 2nd International Conference on the Practice and Theory of Automated Timetabling*, pages 130–141, 1997.

A. Meisels and A. Schaerf. Modelling and solving employee timetabling problems. *Annals of Mathematics and Artificial Intelligence*, 39(1):41–59, 2003.

A. Meisels, S. E. Shimony, and G. Solotorevsky. Bayes networks for estimating the number of solutions of constraint networks. *Annals of Mathematics and Artificial Intelligence*, 28(1):169–186, 2000.

S. Minton. *Learning Search Control Knowledge: An Explanation-Based Approach.* Kluwer, 1988.

S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

T. M. Mitchell, P. E. Utgoff, and R. Banerji. Learning by experimentation: Acquiring and refining problem-solving heuristics. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 1, chapter 6, pages 163–190. Springer-Verlag, 1984.

R. A. Morris, J. L. Bresina, and S. M. Rodgers. Automatic generation of heuristics for scheduling. In M. Pollack, editor, *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 1260–1266. Morgan Kaufmann, 1997a.

R. A. Morris, J. L. Bresina, and S. M. Rodgers. Optimizing observation scheduling objectives. In *Proceedings of the 1997 NASA Workshop on Planning and Scheduling for Space*, 1997b.

R. Nanda and J. Browner. *Introduction to Employee Scheduling*. Van Nostrand Reinhold, 1992.

R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

L. Sachs. *Applied Statistics: A Handbook of Techniques*. Springer-Verlag, second edition, 1982.

A. Schaerf. A survey of automated timetabling. Technical Report CS-R9567, Centrum voor Wiskunde en Informatica (CWI), 1995.

A. Schaerf. Tabu search techniques for large high-school timetabling problems. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 363–368, 1996.

A. Schaerf and A. Meisels. Solving employee timetabling problems by generalized local search. In *Proceedings of the 6th Italian Conference on Artificial Intelligence*, pages 493–502, 1999.

A. Schaerf and M. Schaerf. Local search techniques for high school timetabling. In *Proceedings of the First International Conference on the Practice and Theory of Automated Timetabling*, pages 313–323, 1995.

A. Segre, C. Elkan, and A. Russell. A critical look at experimental evaluations of EBL. *Machine Learning*, 6:183, 1991.

S. F. Smith. Flexible learning of problem solving heuristics through adaptive search. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 421–425. Morgan Kaufmann, 1983.

L. Su, W. Buntine, R. Newton, and B. S. Peters. Learning as applied to stochastic optimization for standard-cell placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(4):516–526, April 2001.

P. Tadepalli and B. K. Natarajan. A formal framework for speedup learning from problems and solutions. *Journal of Artificial Intelligence Research*, 4:419–443, 1996.

L. G. Valiant. A theory of the learnable. *Communications of the Association for Computing Machinery*, 27(11):1134–1142, 1984.

W. Zhang and T. G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1114–1120. Morgan Kaufmann, 1995.

M. Zweben. A framework for iterative improvement search algorithms suited for constraint satisfaction problems. In *Proceedings of the AAAI-90 Workshop on Constraint Directed Reasoning*, 1990.

Appendix A. Attributes

A concept instance is a description of a problem state, an operator, and a problem definition. These variables are represented by the vector $\langle M, Op, P\rangle$, where M is the assignment-table (i.e., the problem state), P is the problem definition, and Op is the operator. Although there are three types of parameterized operators, each operator is represented by the same vector $\langle s, e_1, e_2, t\rangle$, where s indicates the shift, e_i indicates the involved employees, and t indicates the task.

The following definitions and symbols will be used in the attribute description:

- General definitions:
  - Operator (Op) symbols:
    - $Op.s$ - shift field of operator Op.
    - $Op.t$ - task field of operator Op.
    - $Op.e_j$ - employee field of operator Op.

- Let P be an employee timetabling problem. Its domain and its set of constraints are defined by the following symbols:
  - E - set of all employees.
  - S - set of all shifts.
  - T - set of all tasks.
  - SG - set of all shift groups $G_1, ..., G_s$.
  - $Rmin$ - min requirements matrix. $Rmin[S_j, T_k]$ holds the minimum number of employees that must be assigned for task T_k is shift S_j.
  - $Rmax$ - max requirements matrix. $Rmax[S_j, T_k]$ holds the maximum number of employees that are allowed to be assigned for task T_k is shift S_j.
  - $Wmin$ - min workload matrix. $Wmin[E_i, G_k]$ holds the minimum number of shifts of group G_k, in which employee E_i must be assigned.
  - $Wmax$ - max workload matrix. $Wmax[E_i, G_k]$ holds the maximum number of shifts of group G_k, in which employee E_i is allowed to be assigned.
  - C - conflict matrix. $C[S_i, S_j, E_k] = 1$ if employee E_k cannot be assigned to both shifts S_i and S_j. Otherwise, $C[S_i, S_j, E_k] = 0$.
  - Q - qualification matrix. $Q[E_i, T_k] = 1$ if employee E_i qualifies for task T_k, $Q[E_i, T_k] = 0$ otherwise.
  - A - availability matrix. $A[E_i, S_k] = 1$ if employee E_i is *available* for shift S_k and $A[E_i, S_k]$ = 0 otherwise.
- The relevant task to be used while evaluating some feature:

$$task_M(Op, j) = \begin{cases} M[Op.e_j, Op.s] & Op \text{ is a swap operator} \\ Op.t & \text{otherwise.} \end{cases}$$

- $ReqCounter_M[s, t]$
  - Meaning: the number of employees that are scheduled to perform task t during shift s.
  - Expression: $|\{e | e \in E \ \& \ (M[e, s] == t)\}|$.
- $LimitCounter_M[e, sg]$
  - Meaning: the number of shifts belonging to the shift group sg, at which employee e is scheduled to work.
  - Expression: $|\{s | s \in sg \ \& \ (M[e, s] > 0)\}|$.

- Attribute (*xxx*) symbols:
  - *c-xxx*. The attribute *xxx* is a continuous one.
  - *d-xxx*. The attribute *xxx* is a discrete one.

- Constraint violation types:
  - *maximum employee violation*: scheduling too many employees (according to *Rmax*) to perform the same task during the same shift. For example, three employees are scheduled as senior nurses on a certain night shift, although the department allows at most two senior nurses per shift.

- *minimum employee violation*: scheduling too few employees (according to *Rmin*) to perform the same task during the same shift. For example, two employees are scheduled as senior nurses on a certain night shift, although the department does not allow less than three senior nurses per shift.
- *maximum employee workload violation*: scheduling an employee to work too many times (according to *Wmax*) in a shift group. For example, a certain employee is scheduled to work three night shifts in a week, although he should not work more than two night shifts in a week.
- *minimum employee workload violation*: scheduling an employee to work too few times (according to *Wmin*) in a shift group. For example, a certain employee is scheduled to work two night shifts in a week, although he should work at least four night shifts in a week.
- *conflicting-shift violation*: scheduling an employee to work two conflicting shifts (according to *C*). For example, scheduling an employee to work two overlapping shifts.
- *qualification violation*: scheduling an employee to perform a task that he is not qualified (according to *Q*) to perform.
- *availability violation*: scheduling an employee to work a shift for which he is not available (according to *A*).

A.1 State-problem Attributes

These attributes are defined over a state M. They try to capture its characteristics with respect to the problem definition P. All of them are continuous.

1. *c-maximum-employee violations*
   - Meaning: the sum of maximum employee violations due to all the employee scheduling done in state M.
   - Expression: $\sum_{t \in T, s \in S} max\{0, ReqCounter_M[s,t] - Rmax[s,t]\}$.
2. *c-minimum-employee violations*
   - Meaning: the sum of minimum employee violations due to all the employee scheduling done in state M.
   - Expression: $\sum_{t \in T, s \in S} max\{0, Rmin[s,t] - ReqCounter_M[s,t]\}$.
3. *c-maximum-workload violations*
   - Meaning: the sum of maximum employee workload violations due to all the employee scheduling done in state M.
   - Expression: $\sum_{sg \in SG, e \in E} max\{0, LimitCounter_M[e,sg] - Wmax[e,sg])\}$.
4. *c-minimum-workload violations*
   - Meaning: the sum of minimum employee workload violations due to all the employee scheduling done in state M.

- Expression: $\sum_{sg \in SG, e \in E} max\{0, Wmin[e, sg] - LimitCounter_M[e, sg])\}$.

5. *c-conflicting-shift violations*

   - Meaning: the number of conflicting-shift violations due to all the employee scheduling done in state M.
   - Expression: $|\{\{s_1, s_2\}\,|s_1, s_2 \in S \;\&\; (M[e, s_1] > 0) \;\&\; (M[e, s_2] > 0) \;\&\; (C[s_1, s_2, e] == 1)\}|$.

6. *c-qualification violations*

   - Meaning: the number of qualification violations due to all the employee scheduling done in state M.
   - Expression: $|\{\langle e, s\rangle\,|e \in E, s \in S \;\&\; (M[e, s] > 0) \;\&\; (Q[e, M[e, s]] == 0)\}|$.

7. *c-availability violations*

   - Meaning: the number of availability violations due to all the employee scheduling done in state M.
   - Expression: $|\{\langle e, s\rangle\,|e \in E, s \in S \;\&\; (M[e, s] > 0) \;\&\; (A[e, s] == 0)\}|$.

A.2 State-operator Attributes

state-operator attributes are defined over the state M and the operator Op. These attributes attempt to characterize the relations between the state and the operator regardless of the particular problem P. The first three are continuous. The rest are binary attributes. $j \in \{1, 2\}$.

1. *c-total-employee crowding*

   - Meaning: the number of employees scheduled to work during the same shift $Op.s$.
   - Expression: $|\{e|e \in E \;\&\; (M[e, Op.s] > 0)\}|$.

2. *c-certain-employee crowding-j*

   - Meaning: the number of employees scheduled to perform the task $task_M(Op, j)$ during the shift $Op.s$.
   - Expression: $|\{e|e \in E \;\&\; (M[e, Op.s] == task_M(Op, j))\}|$.

3. *c-employee-workload-j*

   - Meaning: the number of shifts in which employee $Op.e_j$ is scheduled to work.
   - Expression: $\left|\left\{s|s \in S \;\&\; (M[Op.e_j, s] > 0)\right\}\right|$.

4. *d-trivial-swap-op-effect*

   - Meaning: check whether Op is a normal swap operator that cannot affect M (i.e., a trivial operator).
   - Expression: $((Op.e_1 \neq Op.e_2) \;\&\; (M[Op.e_1, Op.s] == M[Op.e_2, Op.s]))$.

5. *d-full-swap-op*

- Meaning: check whether Op is a normal swap operator that is also "full" (in a "full" swap operator, the switched tasks are not equal to zero).
- Expression: $((Op.e_1 \neq Op.e_2)$ & $(M[Op.e_1, Op.s] > 0)$ & $(M[Op.e_2, Op.s] > 0))$.

6. *d-replace-swap-op*

   - Meaning: check whether Op is a normal swap operator that is not "full".
   - Expression: $((Op.e_1 \neq Op.e_2)$ & $(((M[Op.e_1, Op.s] == 0)$ & $(M[Op.e_2, Op.s] > 0)) \| ((M[Op.e_1, Op.s] > 0)$ & $(M[Op.e_2, Op.s] == 0))))$.

A.3 Operator-problem Attributes

These attributes are defined over the problem P and the operator Op. They attempt to characterize operator Op with respect to problem P regardless of the particular state M. The first one is a continuous attribute. The rest are binary attributes. $j \in \{1,2\}$.

1. *c-shift-constraint degree*

   - Meaning: the number of shift groups in SG that contain the shift $Op.s$.
   - Expression: $|\{sg | sg \in SG$ & $(Op.s \in sg)\}|$.

2. *d-is-in-shifts-group-i*

   - Meaning: check whether shift group i includes the shift $Op.s$.
   - Expression: for shift group sg_i, $(Op.s \in sg_i)$.

3. *d-is-emp-j-in-employees-group-i*

   - Meaning: check if employee group-i includes the employee $Op.e_j$.
   - Expression: for employee group eg_i, $(Op.e_j \in eg_i)$.

A.4 Operator-state-problem Attributes

These attributes are defined over parts of a state M. They try to capture its characteristics with respect to the problem definition P. These parts of the state M are indicated by the instantiation of the variables of operator Op. All of the following attributes assume that $task_M(Op, j) \neq 0$. $j \in \{1,2\}$.

1. *c-operator-maximum-employee violations-j*

   - Meaning: the value of the maximum employee violation due to the (too many) employees scheduled to perform task $task_M(Op, j)$ during shift $Op.s$.
   - Expression: $max\{0, ReqCounter_M[Op.s, task_M(Op, j)] - Rmax[Op.s, task_M(Op, j)]\}$.

2. *c-operator-minimum-employee violations-j*

   - Meaning: the value of the minimum employee violation due to the (not enough) employees scheduled to perform task $task_M(Op, j)$ during shift $Op.s$.

- Expression: $max\{0, Rmin[Op.s, task_M(Op, j)] - ReqCounter_M[Op.s, task_M(Op, j)]\}$.

3. *c-operator-maximum-workload violations-j*
   - Meaning: the sum of maximum workload violations due to scheduling employee $Op.e_j$ to work (too many times) in shift groups that include the shift $Op.s$.
   - Expression: $\sum_{sg \in SG | Op.s \in sg} max\{0, LimitCounter_M[Op.e_j, sg] - Wmax[Op.e_j, sg]\}$.

4. *c-operator-minimum-workload violations-j*
   - Meaning: the sum of minimum workload violations due to scheduling employee $Op.e_j$ to work (not enough times) in shift groups that include the shift $Op.s$.
   - Expression: $\sum_{sg \in SG | Op.s \in sg} max\{0, Wmin[Op.e_j, sg] - LimitCounter_M[Op.e_j, sg]\}$.

5. *c-operator-conflicting-shift violations-j*
   - Meaning: the number of conflicting shift violations due to scheduling employee $Op.e_j$ to work at shift $Op.s$.
   - Expression: $\left|\{s | s \in S \; \& \; (s \neq Op.s) \; \& \; (M[Op.e_j, s] > 0) \; \& \; (C[s, Op.s, Op.e_j] == 1)\}\right|$.

6. *d-operator-availability violations-j*
   - Meaning: the value of the availability violation due to scheduling employee $Op.e_j$ to work at shift $Op.s$.
   - Expression: $(A[Op.e_j, Op.s] == 0)$.

7. *d-operator-qualification violations-j*
   - Meaning: the value of the qualification violation due to scheduling employee $Op.e_j$, at shift $Op.s$, to perform task $task_M(Op, j)$.
   - Expression: $(Q[Op.e_j, Op.s] == 0)$.

A.5 Problem-base-problem Attributes

These attributes are defined over the problem definition P, and may be informative only when the training model includes more than one training problem. They are restricted to characterizing the differences between this problem and the base problem. In our model, each problem is generated from a base problem by applying a sequence of replacement operators. The only possible changes are therefore in the unavailabilities and in the qualifications of some of the employees. The following attributes characterize these differences.

Assume U_{e_j} as the set of unavailabilities of employee e_j. Assume Q_{e_j} as the set of qualifications of employee e_j. In addition, $diff(A, B)$ is defined to be $(A \setminus B) \cup (B \setminus A)$.

1. *c-absolute-difference-between-unavailability-sets*
   - Meaning: size of the symmetric difference between the unavailability sets of employee $Op.e_1$ and employee $Op.e_2$.
   - Expression: $|diff(U_{e_1}, U_{e_2})|$.

2. *c-relative-difference-between-unavailability-sets*

   - Meaning: normalized version of (1).
   - Expression: $|diff\,(U_{e_1}, U_{e_2})| \,/\, (|U_{e_1}| + |U_{e_2}|)$.

3. *c-difference-between-unavailability-sets-j*

   - Meaning: normalized version of (1).
   - Expression: $|diff\,(U_{e_1}, U_{e_2})| \,/\, \left|U_{e_j}\right|$.

4. *c-absolute-difference-between-qualification-sets*

   - Meaning: size of the symmetric difference between the qualification sets of employee $Op.e_1$ and employee $Op.e_2$.
   - Expression: $|diff\,(Q_{e_1}, Q_{e_2})|$.

5. *c-relative-difference-between-qualification-sets*

   - Meaning: normalized version of (4).
   - Expression: $|diff\,(Q_{e_1}, Q_{e_2})| \,/\, (|Q_{e_1}| + |Q_{e_2}|)$.

6. *c-difference-between-qualification-sets-j*

   - Meaning: normalized version of (4).
   - Expression: $|diff\,(Q_{e_1}, Q_{e_2})| \,/\, \left|Q_{e_j}\right|$.

A.6 Compound Attributes

These attributes combine pairs of continuous atomic attributes by arithmetic relations of their values. All of them are discrete. The following set of attributes is computed for each pair $< attribute_i, attribute_k >$ of continuous attributes. There are compound attributes especially for computing the attributes of section A.5.

1. *d-attribute-i-bigger-than-attribute-k*: $(attribute_i\,(\langle M, Op, P\rangle) > attribute_k\,(\langle M, Op, P\rangle))$.

2. *d-attribute-i-equals-to-attribute-k*: $(attribute_i\,(\langle M, Op, P\rangle) == attribute_k\,(\langle M, Op, P\rangle))$.