Modeling Ill-Structured Optimization Tasks through Cases

Kazuo Miyashita
Electrotechnical Laboratory
1-1-4, Umezono, Tsukuba, Ibaraki 305, Japan
miyasita@etl.go.jp

Katia Sycara
The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.
katia@cs.cmu.edu

Riichiro Mizoguchi
The Institute of Scientific and Industrial Research
Osaka University
8-1, Mihogaoka, Ibaragi, Osaka 567, Japan
miz@ei.sanken.osaka-u.ac.jp

Abstract

CABINS is a framework of modeling an optimization task in ill-structured domains. In such domains, neither systems nor human experts possess the exact model for guiding optimization. And the user's model of optimality is subjective and situation-dependent. CABINS optimizes a solution through iterative revision using case-based reasoning. In CABINS, task structure analysis was adopted for creating an initial model of the optimization task. Generic vocabularies found in the analysis were specialized into case feature descriptions for application problems. Extensive experimentation on job shop scheduling problems has shown that CABINS can operationalize and improve the model through the accumulation of cases.

Key words: case-based reasoning, optimization, task analysis

1 Introduction

A knowledge-based system (or an expert system) has explicit representation of knowledge in addition to the inference mechanism that operates on the knowledge to achieve the system's goal. A knowledge base represents a *model* of how domain experts approach a complicated problem in the domain. It is an *operational model* which, hopefully, exhibits some desired behavior which has been specified or observed in the real-world — in exactly the same way as a *mathematical model* attempts to mirror real-world situations.

Builders of expert systems formulate the model, first by defining a model of the behavior that they wish to understand and then corroborating and extending that model with the aid of specific examples. For example, PROTÉGÉ [26] has two interrelated phases of knowledge base construction: (1) *model building* and (2) *model extension*. When building a model, developers must first perform a requirements analysis and identify the *task* that the expert system has to perform. Then, knowledge engineers and domain experts cooperate to construct a model of the proposed system's behavior. This model generally corresponds to the developer's theory of how the experts actually solve the problem. For extending a model, the model of the intended behavior of the expert system is validated by ascertaining how well the model applies to closely related application problems.

Much of the activity involved in the first stage of model formulation, model building, entails *knowledge-level* [27] analysis, which determines (1) the *goals* for a knowledge-based system, (2) the *actions* of which the system is capable, and (3) the *knowledge* that the system can use to determine the actions that attain the goal. In recent research on AI, there is a clear consensus in favor of knowledge-level analysis and its advantages for knowledge modeling and acquisition. Chandrasekaran and his colleagues advocated the *generic task* framework [2] and identified a number of tasks of general utility (such as classification), methods for performing the tasks and the kinds of knowledge needed by the methods. Clancey proposed *heuristic classification* [5] as an abstract inference pattern for a diagnosis task by examining some expert systems such as MYCIN. McDermott developed *half-weak methods* [18], such as *cover-and-differentiate* and *propose-and-revise* methods, for solving general tasks that do not require domain specific search control knowledge. These methodologies of knowledge-level analyses have successfully been applied to the development of various expert systems.

However, the latter stage of model formulation, model extension, has no generic methodology corresponding to the knowledge-level analysis for the first stage of model formulation. It is generally believed that, though creating a knowledge model may be difficult, extending an existing model is less arduous for human experts. In other words, whereas domain experts may not be able to introspect and articulate the *process knowledge* that allows them to solve problems, these experts can easily supply the *content knowledge* that may or may not be consistent with a given model. To elicit consistent domain knowledge from human experts, several model-based knowledge acquisition tools have been developed such as MOLE [7], SALT [16], KNACK [11] and OPAL [25].

Although these model extension (knowledge acquisition) tools are powerful in allowing domain experts to make large knowledge bases without help from knowledge engineers, such tools must be strongly tied to a specific problem solving method presupposed by the tools. For example, while SALT has been proved to be useful for acquiring knowledge of the expert system, called VT, which supports design of elevator systems, SALT could not acquire effective knowledge for solving scheduling problems. The failure of SALT was caused by the fact that the *propose-and-revise* problem solving method assumed by SALT was inappropriate for the scheduling problem in spite of its structural resemblance to the design problem [32]. Hence, if a problem solving method is generic enough to be applicable to a wide variety of tasks and, at the same time, is capable of matching the nuances of particular applications, a model extension framework based upon such a problem solving method has a highly practical value.

1.1 Case-Based Reasoning

Case-based reasoning (CBR) is the problem solving paradigm where previous experiences are used to guide problem solving [13]. Cases similar to the current problem are retrieved from memory according to a similarity metric, the best case is selected from those retrieved and compared to the current problem. If needed, the precedent case is adapted to fit the current situation based on identified differences between the precedent and the current cases. CBR allows a reasoner (1) to propose solutions in domains that are not completely understood by the reasoner, (2) to evaluate solutions when no algorithmic method is available for evaluation, and (3) to interpret open-

ended and ill-defined concepts. CBR also helps a reasoner (1) take actions to avoid repeating past mistakes, and (2) focus its reasoning on important parts of a problem [14]. Owing to the above advantages, CBR has successfully been applied to many kinds of problems such as design, planning, diagnosis and instruction . Thus CBR can be regarded as an appropriate problem solving method for a large class of applications.

In terms of knowledge acquisition, CBR has a number of practically desirable features which encourage CBR applications in many domains [34]. In CBR, successful cases are stored in the case base so that they can be retrieved and re-used in the future. Failed cases are also stored so that they can warn the problem solver of potential difficulties and help recover from failures. After a problem is solved, the case base is updated with the new experience. Thus, *learning is an integral part of case-based problem solving.* Moreover, in general, it is easier for experts to collect a sufficient number of problem cases by actually solving sample problems rather than try to abstract the particulars of one or more problem cases in order to formulate a consistent rule-set. Hence, CBR has been considered as a more natural and less time consuming method of knowledge acquisition [31, 15].

It should be noted, however, that CBR is not a panacea that obviates any overhead associated with knowledge acquisition. It defines new types of knowledge acquisition tasks, i.e., definition of appropriate case features and indices. For solving these difficulties, there have been attempts of combining knowledge-level analysis and case-based reasoning by constructing a task description of case-based reasoning or using a task structure as a guideline for developing a indexing schema of the case [3, 1]. In this paper, we hypothesize that analysis of the application problem at a task-level provides a useful category of vocabularies for describing features of situations in the problem and these vocabularies can be mapped to the domain specific vocabularies for case descriptions without much effort since both vocabularies are represented at the knowledge-level. To validate our hypothesis, we present a case-based approach, implemented in the CABINS system, for formulating a model of the optimization task and using it to guide iterative solution optimization in ill-structured domains. The model of an optimization task should have the following knowledge: (1) user context-dependent preferences and (2) situation-dependent search control. In CABINS, CBR method is used for extending the optimization task model created by the domain experts and knowledge engineers based on the *task structure* analysis [4]. We

show that when coupled with the task-level analysis, the case-based reasoning method can provide strong leverage for reducing the difficulty of knowledge acquisition. To evaluate the effectiveness of our approach, extensive experiments on CABINS' performance have been done in the domain of job shop scheduling, a widely known ill-structured optimization problem.

The rest of the paper is organized as follows: Section 2 presents the characteristics and challenges of the job shop scheduling domain. Section 3 presents the model formulation approach in CABINS. Section 4 presents the overview of the case-based optimization approach in CABINS. Section 5 explains the experiment method and presents the results. Section 6 presents the concluding remarks.

2 Scheduling Problem

Scheduling assigns a set of orders over time to a set of resources with finite capacity. One of the most difficult scheduling problem classes is job shop scheduling. Job shop scheduling is a well-known NP-complete problem [8]. In job shop scheduling, each order consists of a set of activities to be scheduled according to a partial activity ordering. The dominant constraints in job shop scheduling are: temporal precedence constraints that specify the relative sequencing of activities within an order and resource capacity constraints that restrict the number of activities that can be assigned to a resource during overlapping time intervals.

The activity precedence constraints along with an order's release date and due date restrict the set of acceptable start times for each activity. The capacity constraints restrict the number of activities that can use a resource at any particular point in time and create conflicts among activities that are competing for the use of the same resource at overlapping time intervals. The goal of a scheduling system is to produce schedules that respect temporal relations and resource capacity constraints, and optimize a set of objectives, such as minimizing tardiness, minimizing WIP (work-in-process, i.e. in-process inventory) etc.

Job shop schedule optimization has been very difficult to automate for a variety of reasons.

- It is one of the most difficult NP-hard combinatorial optimization problems [8], and even domain experts are not believed to possess sufficient

heuristic knowledge for making good schedules efficiently [10].

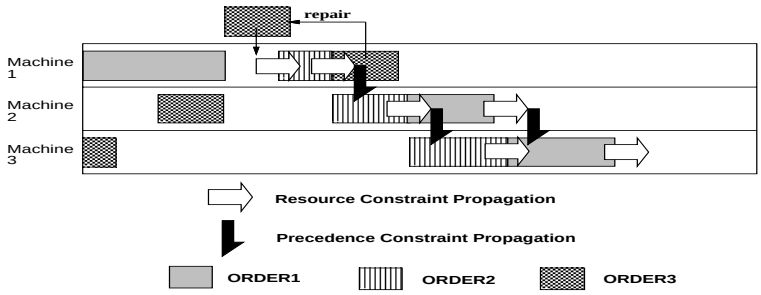

Figure 1: Example of tight constraint interactions

- Owing to the tight interactions among scheduling constraints and the often conflicting nature of optimization criteria, it is impossible to assess with any precision the extent of the required schedule revision to improve the quality of a schedule, or the impact of a scheduling decision on the global satisfaction of optimization criteria. For example, in Figure 1 moving forward the last activity of ORDER3 creates downstream cascading constraint violations. Therefore, the only way for a user or a system to evaluate the desirability of a scheduling action and assess its effects on his/her optimization preferences is to apply the action to the problem, see the resulting schedule and evaluate it in light of the evaluation preferences.

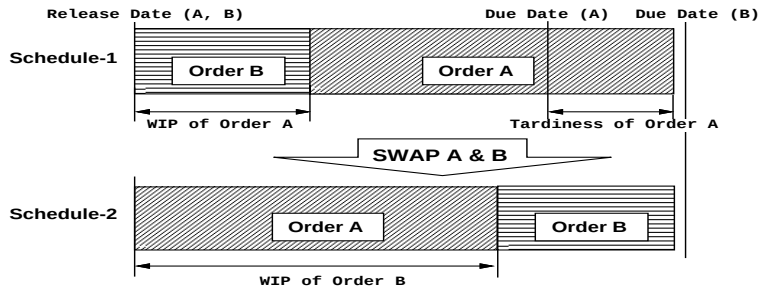

Figure 2: Example of conflicting objectives

- The evaluation itself of what is a "high quality" schedule is difficult because of the need to balance conflicting objectives and trade-offs among

them. Such trade-offs typically reflect user preferences, which are difficult to express as a cost function. For example, WIP and tardiness are not always compatible with each other. As shown in Figure 2, there are situations where a repair action can reduce tardiness, but WIP increases. Which is a better schedule depends on user preferences. A further complication is that user preferences could be implicit and context-dependent (e.g., may depend on the state of the scheduling environment at a particular time). Also, interactions among preferences and effective tradeoffs very often depend on the particular schedule produced. This means that generally a user of the scheduling system can't fully specify his/her preferences a priori before getting the scheduling results from the system. By looking over the obtained schedule results, the user often thinks of additional preferences.

In a nutshell, for real-world schedule optimization problems, we do not usually have precise evaluation criteria on the quality of schedules or effective control knowledge for finding optimal solutions efficiently.

3 Modeling the Optimization Task

The analysis of the ill-structuredness of the job shop scheduling problem in Section 2 provides insights both about the type of the problem solving methods suitable for the ill-structured optimization problem and the type of knowledge required for solving the problem. These insights are used for the definition and enhancement of a knowledge-level model for the problem.

Recently a number of uniform knowledge-level analysis frameworks for describing systems have been developed by several research groups such as MULTIS [24], KADS [35], SPARK [12] and PROTÉGÉ-II [28]. We adopt the task structure analysis [4] for building the model of the optimization task. The task structure is the tree of tasks, methods and subtasks applied recursively until it reaches the tasks that are in some sense performed directly using available knowledge. "Task" is synonymous with types of problem-solving goals: for example, optimization is a task since it characterizes a family of problems, all of which require achieving the goal of generating a solution that maximizes given evaluation criteria. "Method" is a process used to achieve the goals in the task: for example, the task of optimization can be accomplished either by constructive methods or by repair-based methods.

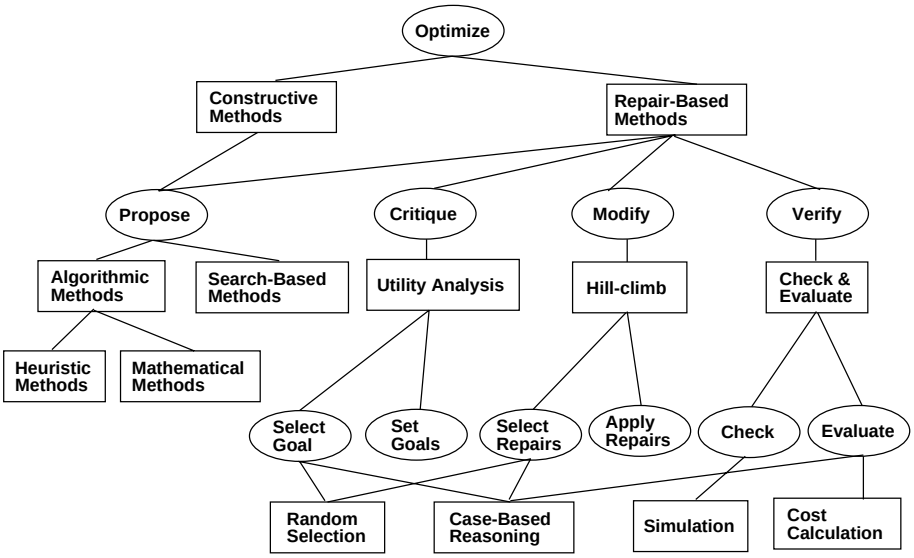

Figure 3: Task structure of optimization

Figure 3 shows the task structure for an optimization task. Since design can be considered as an of the optimization task, this task structure is constructed based on the task structure of design [3, 23]. In the task structure diagram, circles represent tasks and rectangles represent methods. This diagram is not intended to show a complete task structure for the optimization task: it, however, captures some methods and subtasks that are relevant in this paper.

The optimization task can be solved either by constructive methods or by repair-based methods. But, in ill-structured domains such as scheduling, since there is no complete domain knowledge available, the constructive methods cannot produce high quality solutions in an efficient way. The repair-based methods consist of four subtasks: propose, verify, critique and modify. For proposing a solution, two principal methods are available: algorithmic and search-based. The algorithmic methods are further categorized into two classes: heuristic and mathematical. The heuristic methods solve the problem using approximated algorithms. This methodology works only for problems with a restricted problem structure. The mathematical methods, such as linear and integer programming, can solve well-structured problems only after a precise mathematical model of the problem

has been constructed. Search-based methods, such as constraint satisfaction and branch-and-bound, search for the optimal solution in the space of partially constructed solutions with the help of domain specific search control knowledge.

In critiquing a solution, quality of the solution is analyzed based on the utility function of domain experts. If the solution is judged as acceptable, repair process is terminated with the solution. Otherwise, the sources of unacceptability in the solution are identified as repair goals. In modifying a solution, the most effective way of achieving repair goals is selected and applied to the current problem. For selection of repair goals and repair actions, one possible method is to find the most similar past experiences to the current problem situation, which suggest the plausible repair action in the current context. In ill-structured problems, the random selection method is often used for selecting a repair action, since it allows a solution to escape from local minima [29]. Because the methods of goal-setting and repair-application in Figure 3 are strongly domain-dependent, these methods have to be defined and developed by domain experts with the help from knowledge engineers.

In verifying solutions, a problem repair must be checked regarding feasibility of the result. This can be done using simulation methods, such as constraint propagation. If a feasible solution is achieved, it is then evaluated to see whether the repair improves quality of a solution by calculating an explicit cost function, or finding whether the most similar past repair results were evaluated as acceptable or not.

In CABINS, case-based reasoning is used as a method for three subtasks: evaluation, goal selection and repair selection. This is based on our hypothesis that the knowledge required for these subtasks can be acquired with small efforts by the approach shown in Section 3.1.

3.1 Case-Based Model Extension

The task structure used for model building is an analytical tool. A system that performs the task can be viewed as using some of the methods and subtasks in the task structure. Hence, it simply provides a generic vocabulary for describing how systems work. In order to formulate a model for a specific application problem, a developer of the application program needs to extend the model using a task structure.

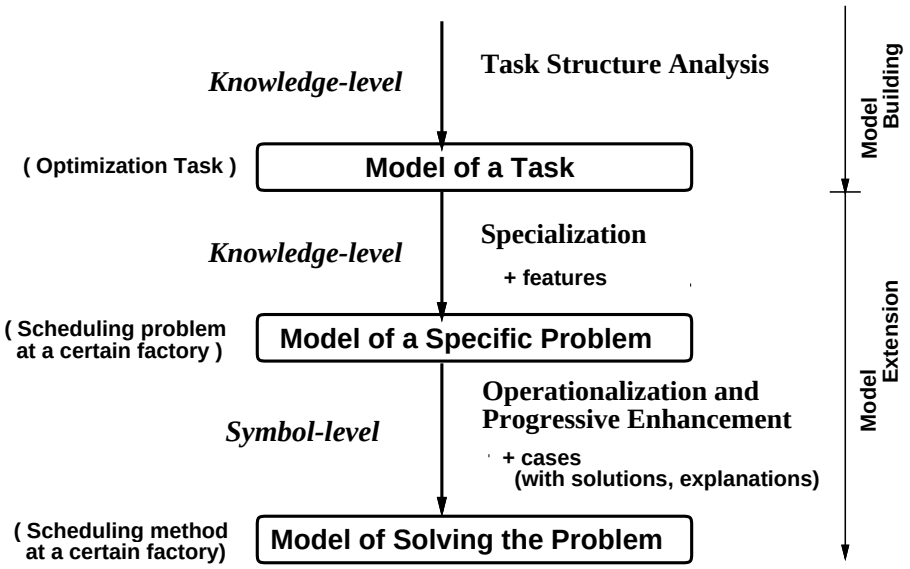

Figure 4: Model formulation in CABINS

Figure 4 shows the schematic diagram of the model formulation process in CABINS. In order to develop the CABINS system, the model of the optimization task in Figure 3 needs to be extended because several kinds of domain-dependent knowledge are required for performing the task. Since CABINS uses case-based reasoning as a principal problem solving method for the optimization task, the model extension process in CABINS is composed of the two phases: *specialization* and *operationalization and progressive enhancement*.

The first phase, specialization, is carried out at the knowledge-level. In specialization, the generic vocabulary found in the task structure analysis is transformed and refined into the vocabulary of the specific problem domain. From the CBR point of view, task structure analysis can provide the abstract case feature categories, and the specialization process maps out the feature descriptions of cases in the application domain. For example, task-level analysis suggests that a case feature category for solution quality description is necessary for the goal selection subtask, and the specialization process in the scheduling domain transforms the feature descriptions into more specific ones such as tardiness and WIP of a schedule. Because both the generic vocabulary found in the task structure analysis and the

domain-specific vocabulary to be specified at specialization are described in the knowledge-level language, specialization can be accomplished by domain experts who can enumerate the appropriate feature descriptions in the application domain with the help from a knowledge engineer who can explain the meaning of the generic vocabularies.

The second phase, operationalization and progressive enhancement, is the symbol-level process. In this phase, specific cases are accumulated by domain experts in a case base according to the case descriptions defined at the previous stage. These cases contain *content knowledge* of the domain such as the judgments and explanations by domain experts in the particular problem context. In ill-structured problems such as job shop scheduling where even human experts are not expected to have sufficient understanding of the problem, acquiring content knowledge as cases is easier than acquiring it in other forms such as rules because of the following reasons: (1) no explicit understanding of domain causality is necessary since cases implicitly map problem features to solutions, (2) knowledge does not need to be abstracted since an indexing mechanism makes abstraction of the content of cases when a case is being matched against a new problem, (3) no consistency check of knowledge needs to be done since among retrieved cases influence of inconsistent or wrong cases is weaken in the mass of "right" cases, and (4) no meta-control information over knowledge usage (such as certainty factors) needs to be described since population and relevance of the cases implicitly defines such control. Cases not only operationalize the model to be executable but also improve the capability of the operationalized model incrementally. Accumulation of cases that successfully achieved the problem goals creates the implicit model of how domain experts solve the problem and make high quality solutions. Accumulation of cases that failed in solving the problem extends the model inductively with useful search control knowledge for avoiding similar failures, which may not be recognized even by the experts. Thus, accumulation of cases can progressively enhance the problem solving capability in terms of both solution quality and problem solving efficiency.

The experimental results of CABINS in Section 5 validate the effectiveness of the above approach in job shop scheduling problems.

4 CABINS: Case-Based Optimization Approach

The considerations of the job shop scheduling problem and the model formulation method for the optimization task, make it clear that (a) an iterative revision optimization technique would be most suitable, (b) recording the user's judgments in a case base is an effective and flexible way of eliciting user optimization preferences to improve solution quality, and (c) recording successful and failed repair trials in a case base is an appropriate way of learning control knowledge to improve search efficiency. We hypothesize that these observations hold true in most of the real world ill-structured optimization problems (e.g. VLSI layout and circuit design, transportation planning).

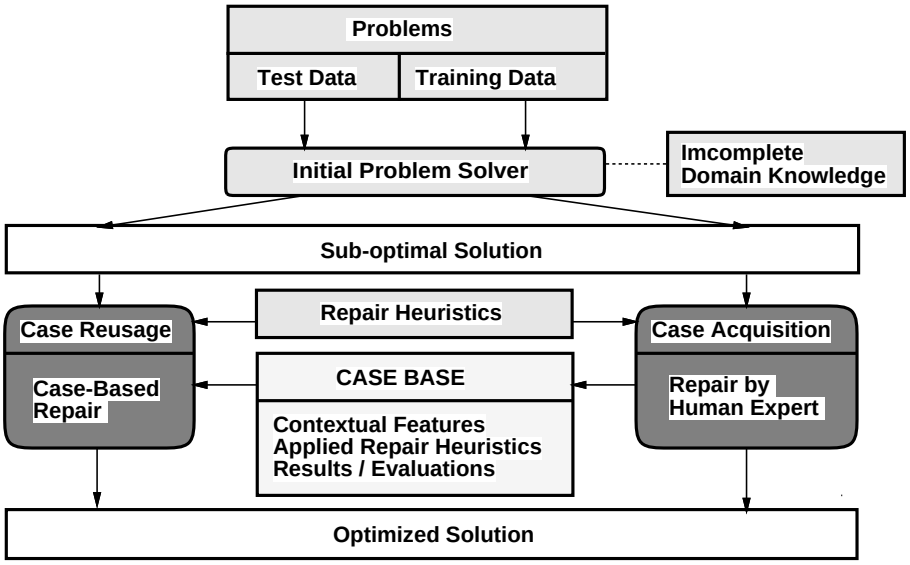

Figure 5: CABINS architecture

CABINS is a unified framework of knowledge acquisition and iterative optimization for ill-structured problems. Figure 5 shows the schematic diagram of the overall architecture of the CABINS implementation. CABINS is composed of three modules: (1) an initial solution builder, (2) an interactive repair (case acquisition) module and (3) an automated repair (case re-use) module.

To generate an initial solution, CABINS can use one of several problem solving methods (e.g. use of dispatch rules) available in CABINS [1]. But, in general any initial problem solver cannot always produce an optimal solution, because the complete knowledge of the domain and user's preferences are not available to the problem solver in ill-structured optimization problems such as job shop scheduling.

In order to compensate for the lack of these types of knowledge, CABINS gathers the following information in the form of cases through interaction with a domain expert in its training phase.

- A suggestion of which defect to be repaired: a user's selection of the most critical defect in a given solution.
- A suggestion of which repair action to apply : a user's decision on what repair action to be applied to a given solution for quality improvement.
- An evaluation of a repair result : a user's overall evaluation of a modification result.

CABINS acquires from cases a category of concepts that reflect user preferences: what combination of effects produced by an application of a particular local optimization action on a schedule constitutes an acceptable or unacceptable outcome. In CABINS, the optimization criteria are not explicitly represented as case features or in terms of a cost function, but they are implicitly and extensionally represented in the case base. CABINS learns three additional categories of concepts that reflect control knowledge for quality enhancement and efficiency improvement: (1) what aspect of a solution should be repaired, (2) what heuristic repair action to choose in a particular repair context, and (3) when to give up further repair. These concepts are recorded in the case base and are used by CABINS to guide iterative optimization and infer optimization tradeoffs in evaluating the current solution. In this way, the acquired knowledge is exploited to enhance the incomplete domain model in CABINS and improve efficiency of problem solving and quality of resulting solutions.

[1] We think CBR is an inappropriate method for generating initial schedules because owing to ill-causality of scheduling problems inherently different solutions become optimal for slightly different problems.

A basic assumption of our case-based approach is that, in spite of the ill-structuredness of the problem, the following three types of domain knowledge are available and constitute useful case features.

- Repair heuristics : a set of repair heuristics that can be applied to a problem.
- Descriptive features : attributes of a problem that describe a particular problem situation and might be useful in estimating the effects of applying repair heuristics to the problem. These features will be explained in detail in Section 4.1 for the job shop scheduling problem.
- Evaluation criteria : quantification of different aspects of the effects of applying repair heuristics to the problem. The degree of importance on these criteria is in general user- and state-dependent.

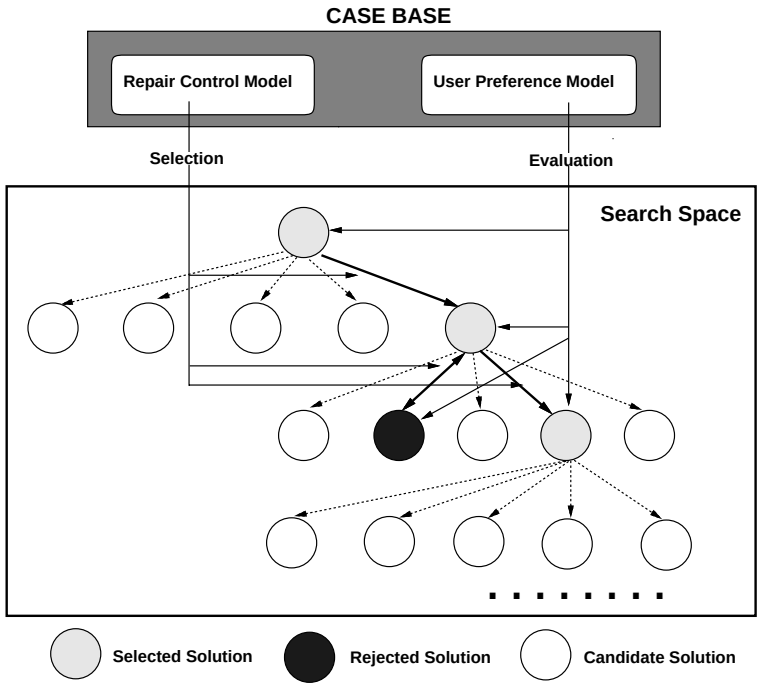

Figure 6: Search space and search control in CABINS

After enough cases have been gathered, CABINS searches for an "optimal" solution over the space of complete solutions autonomously. Figure 6

shows the schematic diagram of the search space and search control in the CABINS system. CABINS revises the current solution iteratively to improve the solution quality. For each step of the search, CABINS selects a solution among the neighbors of the current solution. The neighborhood size for the current solution (i.e. the number of potential solutions for each revision) is equal to $Number\_of\_Repair\_Actions \times Number\_of\_Repair\_Objects$. In a scheduling problem, *Repair_Actions* are several heuristics that modify the assignments of resources to activities in the schedule and *Repair_Objects* are typically the activities in the schedule. The number of revision cycles required to obtain a final solution cannot in general be predicted in advance because of tight constraint interactions in the scheduling problem. Hence the search space for a large scheduling problem can be intractably big. CABINS has the following mechanisms to control search using CBR: A *repair control model* provides search control through case-based selection of the next repair goal and action, and a *user preference model* provides search control through case-based evaluation of the result of the application of a selected repair action.

4.1 Case Representation

Corresponding to the task structure in Figure 3, CABINS has three subtasks that use CBR as a problem solving method: goal selection, repair selection and evaluation. In CABINS, repair selection is further divided into strategy selection and tactic selection. A repair goal is derived from a particular high level description of defects in the solution of a problem and their significance to a user. A way to achieve a particular repair goal is designated by selection of one of the associated repair strategies. Each repair strategy is executed by a successive application of a variety of repair tactics associated with it. A result of a tactic application is evaluated to check whether it is acceptable or not. In CABINS' repair process, all decisions (i.e. goal/strategy/tactic selection and result evaluation) are made using case-based reasoning. Hence, the content of a case must be able to represent all the decision criteria of human experts in the repair process.

Case descriptions in CABINS are defined through specialization of the task structure analysis results. Since a case in CABINS describes the application of a particular repair action to the problem, CABINS has three general types of cases corresponding to three hierarchical classes of repair actions:

goal_case, *strategy_case* and *tactic_case*. Each case type is delineated with *descriptive features*, which are heuristic approximations that reflect problem space characteristics, and a *repair history*, which records the sequence of applications of successive repair actions, the repair effects and the repair outcome. Each case type has different sets of the categories that are derived from task-level analysis and characterize the features to be described in the case. The hierarchy of repair actions and the categorization of case features give strong semantics for helping a user of CABINS understand and organize her/his expertise to be represented in a case. As a result, domain experts can easily define their own specific features and repair actions, which can be implemented by a programmer without knowledge of CABINS' internal structure and process.

Feature categories in CABINS are derived from the task structure of the optimization task (see Section 3). The task structure analysis has identified the following feature categories for each type of case:

- In a goal_case, information necessary for selecting a repair goal is stored. The repair goal is selected by identifying the most critical defect of a current problem taking into consideration the problem context. Accordingly, the goal_case has two categories of features: *Quality* and *Situation*. In Quality category, a user can define the features which constitute the evaluation of the current problem and whose unsatisfactory values cause defects. In Situation category, a user describes external factors which could influence the evaluation of the current problem. A repair action recorded in the goal_case is a repair goal. The repair goal is to select one of the most critical objectives in the current problem and sort repair targets according to the selected objective. A user's expertise captured in the goal_case is that of detecting the defects of a solution in a given problem context (i.e. quality and situation).

- A strategy_case records the information necessary to select a repair strategy. Since the repair strategy is selected based upon global characteristics of a repair target, the strategy_case has a single category for features: *Global-context*. Features in Global-context represent potential repair flexibility of the repair target as a whole. A repair action stored in the strategy_case is a repair strategy. The repair strategy is to determine the degree of allowable change by a repair and sort

changeable components of the repair target appropriately in the order of tactic applications so that ripple effects of tactic applications are minimized. The strategy_case captures user's knowledge about the tradeoffs between possible effects and allowable global disruptions by repairing the target.

- A tactic_case represents the information required for selecting a repair tactic. The repair tactic is selected based on local characteristics of each component of a current repair target. Features in the tactic_case belong to the category of *Local-context*. Local-context features reflect flexibility for revision of a repair target component within limited bounds allowed by a repair strategy. A repair action stored in the tactic_case is a repair tactic that executes revision on components of a repair target.

  Another important piece of information stored in the tactic_case is evaluation of a repair result. Features used for evaluating a repair result belong to *Repair-effect* category. Since a repair tactic is the only repair action that can make actual changes on the repair targets, the effects of the changes are evaluated by a domain expert and stored in the tactic_case with a judgment on acceptability of repair results. Thus, the tactic_case captures a user's knowledge about (a) prediction of local effects by a possible tactic application, and (b) user's preferences on the tradeoffs between favorable and unfavorable repair effects (both in local and global perspectives).

Figure 7 presents a schematic diagram showing the abstracted content of CABINS case base. The content is derived from the specialization of the task structure analysis results on the optimization task to the scheduling problem, where an order in a schedule is selected as a repair target for a strategy application. For a tactic application, each activity in the selected order is designated as a repair target. Hence, features of an order are stored as global-context in strategy_cases, and activity features are recorded as local-context in tactic_cases. Also in a tactic case, schedule quality changes are recorded as repair-effect with evaluation results by domain experts.

As will be shown in the following examples, each feature in CABINS' case is described by a set of attributes, such as `Feature`, `Value`, `Filtering`, `Importance` and `Similarity`. `Feature` attribute specifies a name of the feature to be considered. `Value` attribute records a value of the feature. Val-

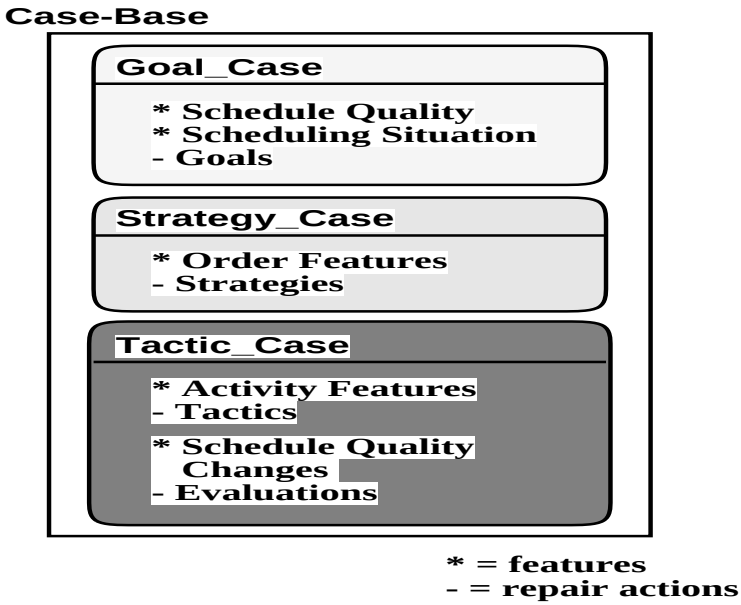

Figure 7: Case base for schedule optimization problem

ues of `Filtering`, `Importance` and `Similarity` attributes can be assigned by a user during case acquisition only when s/he likes to override the default retrieval procedure for the case. If a value of `Filtering` attribute in a feature is "ON", exact match of the feature is required for the case to be retrieved. If the value is "OFF", partial match can contribute to the case retrieval. A value of `Importance` attribute designates importance of the feature in a particular case. Thus, `Filtering` and `Importance` attributes allow the user to express the uniqueness of the feature in a particular case in different ways. These attributes can be used to represent an exceptional case in which a value of the feature in the case has a special meaning to the user. `Similarity` attribute denotes the function used to calculate feature similarity.

Figure 8 shows an example of a goal_case used in the experiments of this paper. In the example, features belonging to `Schedule_Quality` are refined into "weighted_tardiness" and "inprocess_inventory", since they are major concerns in the experiments. For features in `Scheduling_Situation`, the case has "year", "month" and "economy" (i.e. boom or depression), which are considered to affect the judgment on the appropriateness of the cur-

```
G_Case {
   Name = "exp-data/exp_0_0_5:G1";
   Schedule_Quality = (
      Slot {                                    Slot {
         Feature = weighted_tardiness;             Feature = inprocess_inventory;
         Value = 2370;                             Value = 7270;
         Filtering = OFF;                          Filtering = OFF;
         Importance = 1.0;                         Importance = 1.0;
         Similarity = NORMAL;                      Similarity = NORMAL;
      }                                         }
   )
   Scheduling_Situation = (
      Slot {                                    Slot {
         Feature = year;                           Feature = month;
         Value = 1993;                             Value = 10;
         Filtering = OFF;                          Filtering = OFF;
         Importance = 1.0;                         Importance = 1.0;
         Similarity = NORMAL;                      Similarity = NORMAL;
      }                                         }
      Slot {
         Feature = economy;
         Value = VERY_BAD;
         Filtering = ON;
         Importance = 1.0;
         Similarity = NORMAL;
      }
   )
   Goals = (
      G_Solution {
         Goal = reduce_weighted_tardiness;
         Result = FAILED;
      }
      G_Solution {
         Goal = reduce_inprocess_inventory;
         Result = SUCCEEDED;
      }
   )
}
```

Figure 8: An example of CABINS' goal_case for a schedule optimization problem

rent schedule. For example, "month" may imply several seasonal factors which influence the production planning such as more strict due-date requirements that are widely observed at the end of a year. The case shows that "reduce_inprocess_inventory" is selected as a critical goal and successfully achieved after a failure of attaining "reduce_weighted_tardiness" goal.

```
S_Case {
    Name = "exp_0_1_1:G34:GS2:O8";
    Goal = reduce_inprocess_inventory;
    Order_Features = (
        Slot {
            Feature = slack_ratio;
            Value = 0.73529;
            Filtering = OFF;
            Importance = 1.0;
            Similarity = NORMAL;
        }
        Slot {
            Feature = inventory_ratio;
            Value = 0.125;
            Filtering = OFF;
            Importance = 1.0;
            Similarity = NORMAL;
        }
        Slot {
            Feature = resource_busy_deviation;
            Value = 2.387823;
            Filtering = OFF;
            Importance = 1.0;
            Similarity = NORMAL;
        }
        Slot {
            Feature = tardiness_ratio;
            Value = 0.0;
            Filtering = OFF;
            Importance = 1.0;
            Similarity = NORMAL;
        }
        Slot {
            Feature = resource_idle_ratio;
            Value = 0.247059;
            Filtering = OFF;
            Importance = 1.0;
            Similarity = NORMAL;
        }
    )
    Strategies = (
        S_Solution {
            Strategy = shift_right_all;
            Result = SUCCEEDED;
        }
    )
}
```

Figure 9: An example of CABINS' strategy_case for a schedule optimization problem

Figure 9 is an example of a strategy_case in CABINS. In the example, `Order_Features` are refined into five features describing Global-context of an order. "Slack_ratio" is, for example, the total waiting time divided by the length of an allowable time window for completion of the order (i.e. from its release_date to due_date). High "slack_ratio" often shows a loose schedule with much repair flexibility. "Resource_busy_deviation" is the standard deviation of utilization of all the resources that activities of the order can be assigned to. High "resource_busy_deviation" indicates the presence of highly contended-for resources (bottleneck resources) which in turn makes repair less

flexible. And the example case records that "shift_right_all" strategy, which means to move *all the activities* of the order to the right on the timeline, is selected as a repair strategy and succeeded in achieving a goal.

Figure 10 is an example of a tactic_case in CABINS. `Activity_Features` include the features of an activity within limited bounds. In particular, the bound that CABINS uses is a time interval called *repair time horizon*. The repair time horizon of the activity is the time interval between the end of the activity preceding the activity in the same order and the start of the activity succeeding the activity in the same order (see Figure 11).

Associated with the repair time horizon are the features which potentially are predictive of the effectiveness of applying a particular repair tactic. "Left_slack_ratio" and "right_slack_ratio" roughly estimate the flexibility of the activity in its time horizon *without* considering resource contention. And "alternative_resources" shows the number of alternative resources to which the activity can be assigned. The other features in `Activity_Features` predict how much overall gain will be achieved by applying a corresponding repair tactic to the activity in its time horizon. For example, "imm_left_idle_ratio" predicts the possible effects of applying "slide_left" tactic to the activity.

In the example case, "jump_left" tactic, which moves the activity on the same resource as much to the left on the timeline as possible within the repair time horizon, is applied and the effects of the repair are recorded in the features of `Schedule_Quality_Changes`. Features in `Schedule_Quality_Changes` describe the impacts of a repair action application on schedule optimization objectives (e.g. tardiness, inventory). Typically these effects reflect a diverse set of objectives to be considered and heavily related to `Schedule_Quality` features in a goal_case. To be noted is that there are two perspectives in recording these effects. One is the local perspective that describes the effects that occurred to the repaired activity. The other is the global perspective, which represents the effects of a tactic application in the overall schedule. Since the effects caused by a tactic application are not determined until all the activities selected by the strategy are repaired, a human expert has to predict the acceptability of a tactic application result by considering the trade-off of current global and local repair effects. As a result, the repair effects in both perspectives need to be recorded.

`Result` is the evaluation assigned to the set of effects of a repair action and takes a value in the set ["SUCCEEDED", "FAILED"]. This judgment of a repair outcome must be made by a domain expert in the training phase and

```
T_Case {
    Name = "exp_2_1_9:G16:GS1:O1:OS1:A2";
    Goal = reduce_weighted_tardiness;
    Strategy = shift_left_all;
    Final = NO;
    Activity_Features = (
        Slot {                                          Slot {
            Feature = left_slack_ratio;                     Feature = right_slack_ratio;
            Value = 13.571429;                              Value = 0.0;
            Filtering = OFF;                                Filtering = OFF;
            Importance = 1.0;                               Importance = 1.0;
            Similarity = NORMAL;  }                         Similarity = NORMAL; }
        Slot {                                          Slot {
            Feature = imm_left_idle_ratio;                  Feature = imm_right_idle_ratio;
            Value = 25.714286;                              Value = 1885.714286;
            Filtering = OFF;                                Filtering = OFF;
            Importance = 1.0;                               Importance = 1.0;
            Similarity = NORMAL;  }                         Similarity = NORMAL; }
        Slot {                                          Slot {
            Feature = aggr_left_idle_ratio;                 Feature = aggr_right_idle_ratio;
            Value = 6242.857143;                            Value = 0.0;
            Filtering = OFF;                                Filtering = OFF;
            Importance = 1.0;                               Importance = 1.0;
            Similarity = NORMAL;  }                         Similarity = NORMAL; }
        Slot {                                          Slot {
            Feature = left_swappability;                    Feature = right_swappability;
            Value = 150.769231;                             Value = 0.0;
            Filtering = OFF;                                Filtering = OFF;
            Importance = 1.0;                               Importance = 1.0;
            Similarity = NORMAL;  }                         Similarity = NORMAL; }
         Slot {                                         Slot {
             Feature = left_alt_idle_ratio;                 Feature = right_alt_idle_ratio;
             Value = 271.428571;                             Value = 0.0;
             Filtering = OFF;                                Filtering = OFF;
             Importance = 1.0;                               Importance = 1.0;
             Simialrity = NORMAL;  }                         Similarity = NORMAL; }
          Slot {                                        Slot {
              Feature = left_alt_swappability;              Feature = right_alt_swappability;
              Value = 0.0;                                  Value = 0.0;
              Filtering = OFF;                              Filtering = OFF;
              Importance = 1.0;                             Importance = 1.0;
              Simialrity = NORMAL;   }                      Similarity = NORMAL; }
           Slot {
               Feature = alternative_resource;
               Value = 3;
               Filtering = ON;
               Importance = 1.0;
               Simialrity = NORMAL; }
    )
    Tactics = (
       T_Solution = {
          Tactic = jump_left;
          Schedule_Quality_Changes = (
              Slot {                                    Slot {
                  Feature = local_weighted_tardiness;       Feature = local_inprocess_inventory;
                  Value = 280.0;                            Value = -950.0;
                  Filtering = OFF;                          Filtering = OFF;
                  Importance = 1.0;                         Importance = 1.0;
                  Similarity = NORMAL; }                    Similarity = NORMAL; }
              Slot {                                    Slot {
                  Feature = global_weighted_tardiness;      Feature = global_inprocess_inventory;
                  Value = 0.0;                              Value = 0.0;
                  Filtering = OFF;                          Filtering = OFF;
                  Importance = 1.0;                         Importance = 1.0;
                  Similarity = NORMAL; }                    Similarity = NORMAL; }
          )
          Effect = 280.0;
          Result = SUCCEEDED; }
    )
}
```

Figure 10: An example of CABINS' tactic_case for a schedule optimization problem

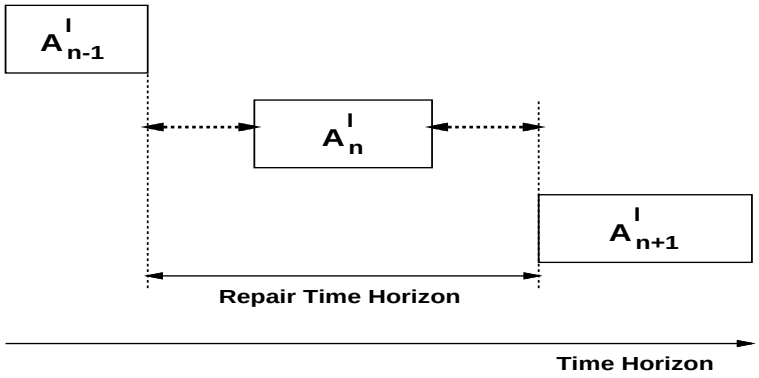

Figure 11: Repair time horizon of the activity (A_n^l)

gets recorded in the case base. A result is "SUCCEEDED" if the tradeoff involved in the set of effects for the current application of a repair action is judged acceptable. `Effect` is approximation of the aggregated effects by a repair and is determined subjectively by the user.

4.2 Case Acquisition

In CABINS, the session starts with an empty case base. A set of training problems are presented to a user who interacts with CABINS to repair the problems by hand. The interactions and the results are stored in cases with the information of the problem contexts. Through the case acquisition, the user can operationalize the model of solving schedule optimization problems.

Figure 12 shows interactions between CABINS and the user for case acquisition. In iteratively repairing a solution of a training problem, the user has to select the repair action that is deemed to be appropriate in a given particular problem situation, apply it to the problem, and evaluate the result repeatedly. User's decisions in the course of a repair along with the problem context are recorded in a case. In the selection of the above repair actions (i.e. goal, strategy and tactic), the user can assign values to the attributes of case features, such as `Filtering`, `Importance` and `Similarity` as supplemental explanations to his/her decisions.

In a schedule optimization problem, the user first selects the most urgent repair goal for a given schedule from the list of user defined goals to be achieved in the schedule. User's selection of a repair goal, overall quality

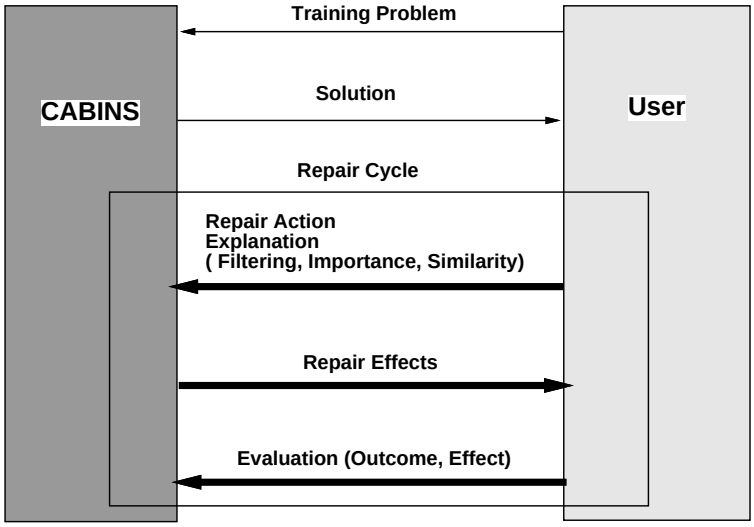

Figure 12: Interactions between CABINS and a user for case acquisition

of the current schedule and situations influencing user's scheduling decisions are recorded in a goal_case. An application of the repair goal to the current schedule produces the sorted list of orders according to significance of the defect in the given repair goal. Then, the user selects a repair strategy from a set of user defined repair strategies for repairing an order according to the sorting. User's selection of the repair strategy and global characteristics of the order constitute the content of a strategy_case. When the repair strategy is applied to the order, some activities of the order are picked up and sorted so as to avoid unnecessary computations and unbounded ripple effects. Finally, the user selects a repair tactic from a set of user defined repair tactics for repairing the first activity in the sorted queue.

A repair tactic application causes changes in the schedule by executing a repair and applying constraint propagation to resolve constraint violations. The repair tactic application may result in an infeasible schedule. An infeasible schedule will occur when constraints are propagated beyond the fixed time window boundary of any order (i.e. the time window between release date and latest due date of an order).

If the outcome of the repair tactic application is feasible, the effects of the repair are calculated and shown to the user. An effect describes the result of the repair with respect to one of the repair objectives defined as features

in `Schedule_Quality_Changes` of a tactic_case. Because of tight constraint interactions, these effects are ubiquitous in job shop scheduling and make schedule optimization extremely hard. When the application of a repair tactic produces a feasible result, the user must determine whether the resultant schedule is acceptable or not based on the calculated effects. The outcome is judged as unacceptable, if the schedule resulting from the application of the revision heuristic is feasible but the revision result does not make any improvement with respect to the user's criteria. This could happen because harmful effects might outweigh, in the user's judgment, the effected improvement. For example, if reduction of an order tardiness enforces increased utilization of low-quality machines, total cost incurred by this repair might eventually be increased, but not decreased for the user who would dislike the possible low quality of products. Therefore such a repair might be judged as unacceptable by the user. The user's judgment as to balancing favorable and unfavorable effects related to a particular optimization objective constitutes the explanations of the repair outcome. The user can supply a supplemental explanation of the judgment by assigning a value to `Effect` attribute of a repair tactic description in a tactic_case. This gives a case the additional information about to what extent the case is acceptable or unacceptable. At the end of each repair tactic application, the applied repair tactic, the effects of the repair and user judgment/explanation as to the repair outcome are recorded in a case along with the activity features.

The repair process continues until an acceptable outcome is reached, or failure is declared. Failure is declared when there is no more repair action available. The sequence of applications of successive repair actions, the effects and user's evaluation of the results are recorded in the case. In this way, a number of cases are accumulated in the case base and the model of solving the schedule optimization problem is operationalized and progressively enhanced.

4.3 Case Retrieval and Re-use

Once enough cases have been gathered, CABINS repairs sub-optimal schedules without user interaction. CABINS repairs the schedules by (1) invoking CBR with schedule quality and scheduling situation as indices to recognize schedule sub-optimalities and set a repair goal, (2) focusing on an order to be repaired based on the repair goal, (3) invoking CBR with order features as indices to decide a repair strategy, (4) focusing on an activity to be repaired

in the order, (5) invoking CBR with activity features as indices to decide the most appropriate repair tactic to be used for each activity in the order, (6) invoking CBR using the schedule quality changes as indices to evaluate the repair result, and (7) when the repair result seems unacceptable, invoking CBR with the failed tactic as an additional index to decide whether to give up or which repair tactic to use next.

In CABINS cases are retrieved using k-Nearest Neighbor method [6], and the standard formula of calculating the similarity between i-th case and the current problem is as follows [2]:

$$exp(-\sqrt{\sum_{j=1}^{N}(IM_j^i \times \frac{CF_j^i - PF_j}{SD_j})^2})$$

where IM_j^i is the importance of j-th feature of i-th case in the case base, and its value has been heuristically defined by the user. CF_j^i is the value of j-th feature of i-th case, PF_j is the value of j-th feature in the current problem, SD_j is the standard deviation of j-th feature value of all cases in the case base. Feature values are normalized by division by a standard deviation of the feature value so that features of equal salience have equal weight in the similarity function.

5 Experiments

We hypothesize that, with accumulation of cases, CABINS can (a) acquire user optimization preferences and re-use them to produce better solutions, and (b) learn control knowledge to solve problems more efficiently. In other words, through cases CABINS can incrementally improve the model for schedule optimization in terms of both solution quality and efficiency. In this section we report experimental results in the job shop schedule optimization problems to validate our hypotheses.

Our hypotheses are difficult to test since, due to the subjective and ill-defined nature of user preferences, it is not obvious how to correlate schedul-

[2] There are some varieties in the methods of similarity calculation depending on the value of `Filtering` and `Similarity` attributes in the features, but it is not relevant to the discussion in this paper. See [22] for more details.

ing results with the captured preferences or how to define quality of a schedule whose evaluation is subjective.

To address these issues, we had to devise a method to test the hypotheses in a consistent manner. To do that, it is necessary to know the optimization criterion that would be implicit in the case base, so that the experimental results can be evaluated. In the experiments reported here, we used the explicit criterion WIP+weighted tardiness to reflect the user's optimization criterion and built a rule-based reasoner (RBR) that goes through a trial-and-error repair process to optimize a schedule based on the given criteria.Since the RBR was constructed not to select the same repair action after a repair result was calculated as unacceptable, it could go through all the repair actions before giving up repairing a schedule. For each repair, the repair effects were calculated and, on this basis, since RBR had a predefined evaluation objective, it could evaluate the repair outcome in a consistent manner. Thus, RBR was used to generate a case base with about 8500 cases for the explicit optimization objective. Since RBR knows the exact objective function for evaluation, it can work as an emulator of a human scheduler, who cannot repair a schedule in the most efficient way, but can make consistent evaluations of repair results. Therefore, we used RBR not only for generating the case base for CABINS but also for making a comparison baseline for the CABINS experiments. Naturally, the objective, though known to us and RBR, is not known to CABINS and is only implicitly and indirectly reflected in an extensional way in each case base. By designing an objective into the RBR so it could be reflected in the case base we got an experimental baseline against which to evaluate the schedules generated by CABINS. We used RBR for the purpose of the controlled experiments that can clarify the usefulness and performance of CABINS. This is analogous to the controlled experiments in the psychology where the experimental situations are much simpler than what humans do but the virtue of the controlled experiments is that the experimenter can isolate and control the some variables so that meaningful conclusions can be made rather than having a confounding variables where no one can find out what is responsible for the experimental results.

We evaluated the approach on a benchmark suite of 60 job shop scheduling problems where parameters, such as number of bottlenecks, range of due dates and activity durations were varied to cover a range of job shop scheduling problem instances with the following structure. Each problem class has 10 orders of 5 operations each and 5 machines. Two parameters

were used to cover different scheduling conditions: a range parameter controlled the distribution of order due dates and release dates, and a bottleneck parameter controlled the number of bottleneck resources. Six groups of 10 problems each were randomly generated by considering three different values of the range parameter, and two values of the bottleneck configuration (1 and 2 bottleneck problems). These problems are variations of the problems originally reported in [30]. Our problem sets are, however, different in two respects: (a) we allow substitutable resources for non-bottleneck resources whereas the original problems did not, and (b) the due dates of orders in our problems are tighter by 20 percents than in the original problems.

A cross-validation method was used for the experiments. Each problem set in each class was divided in half. The training problem set was repaired by RBR to gather cases. These cases were then used for case-based repair of the validation problem set. We repeated the above process by interchanging the training and test sets. Reported results are for the validation problem sets.

In the following subsections, we present our empirical results that increasing case base size improves the performance of CABINS in terms of solution quality and problem solving efficiency. These results indicate that the accumulation of cases can progressively enhance the competence of the model of solving scheduling problems.

5.1 Case Accumulation Effects on Quality

Increase of the case base size might have beneficial or harmful effects on the CABINS performance in terms of solution quality. The possible explanations of these effects are as follows:

- Better quality solutions: After having new cases that successfully repaired the novel problems, CABINS increases possibility of improving solution quality by later re-using these cases for similar problem situations.

- Poorer quality solutions: There are two explanations that additional cases can have a deleterious effect on the quality of solutions found by CABINS. One is that incorrect cases may lead CABINS to produce poorer solutions. The other is that increase of the number of failure

cases in the case base can force CABINS to give up further exploration for better solutions even when there is a good chance to find some.

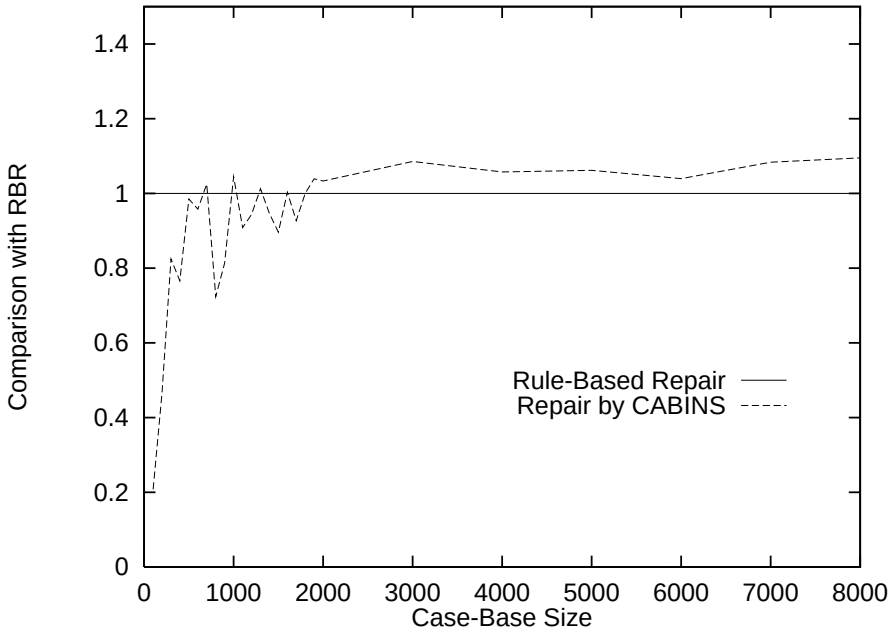

Figure 13: Effect of case base size on solution quality by CABINS

The graph in Figure 13 compares the performance of CABINS with different sized case bases in terms of solution quality. The comparison baseline is the performance of RBR that created the cases in CABINS. A value greater than 1.0 indicates that the solution produced by CABINS at the indicated case base size exceeded the solution quality produced by RBR. In the graph, the results of every 100 cases increment are plotted until the size of case base exceeds 2000 and then the results of every 1000 cases increment are plotted until 8000 are shown. To get the case bases of different sizes, an appropriate number of cases for each problem class were randomly selected and deleted from the created case base of 8000 cases.

From the graph, the schedule quality is found to improve with increased case base size. However, the marginal payoff from the increase in the case base size decreases. This can be explained partially by the fact that some number of cases (say, 2000 cases) capture well characteristics of the user

preference model that is created by the records of solution evaluations by the user in various problem contexts, and additional 1000 new cases may give much redundant information. When the size of a case base is relatively small, a new case usually adds information about a different part of the model and improves the capability of CABINS producing solutions of better quality, but it sometimes directs search to local minima which cannot be escaped until other new cases are acquired. Nevertheless the figure indicates that after collecting enough number of cases (2000 cases), CABINS does not degrade the solution quality with increase of case base size. This shows that in 2000 cases CABINS stored enough knowledge for avoiding or escaping from local minima in the search space. Therefore it is concluded that the size of a case base in CABINS can improve the solution quality only at the early stage of case accumulation. Neither deleterious nor favorable effect on solution quality can be found by further accumulation of cases. In other words, accumulation of cases can quickly improve the operational capability of the user preference model in CABINS.

5.2 Case Accumulation Effects on Efficiency

Increase of the case base size might also have beneficial or harmful effects on the CABINS performance in terms of problem solving efficiency. The possible explanations of these effects are as follows:

- More efficient problem solving: By increasing the number of failure cases, the number of different failure types that become known to CABINS increases. Hence CABINS can avoid repeating a large variety of failures, thus reducing the search time.

- Less efficient problem solving: Irrelevant cases may suggest CABINS to expand the search tree in fruitless directions.

In the experiment, we take a number of repair tactic applications as an indicator of the problem solving efficiency by CABINS because applications of repair tactics are the most time-consuming factor in CABINS' repair process.

The graph in Figure 14 compares the performance of CABINS with different sized case bases in terms of number of tactic applications. The comparison baseline is the performance of RBR. The value less than 1.0 in the graph indicates that problem solving by CABINS at the case size was more

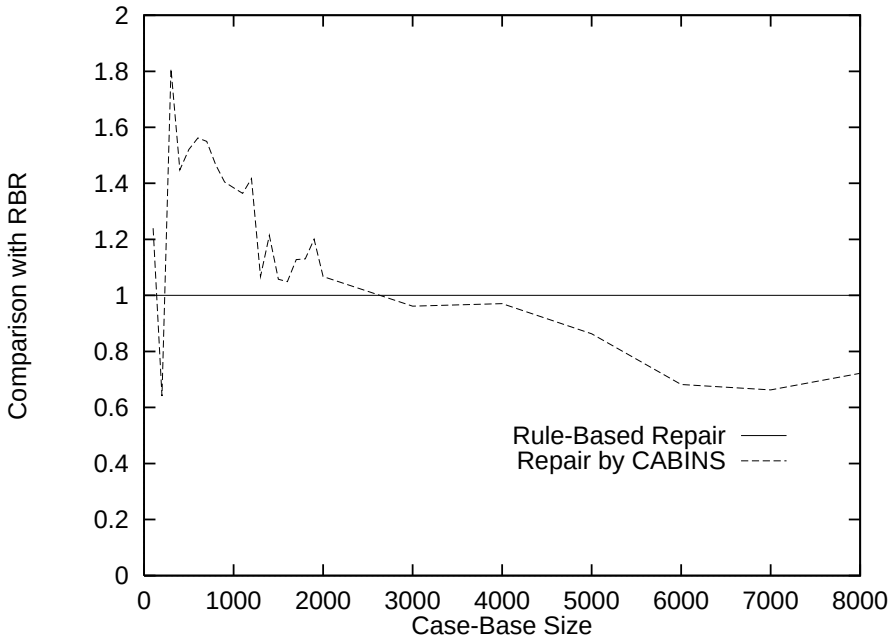

Figure 14: Effect of case base size on problem solving efficiency by CABINS

efficient than RBR. In the graph, the results of every 100 cases increment are plotted until the size of case base exceeds 2000 and then the results of every 1000 cases increment are plotted until 8000 are shown.

The graph in Figure 14 shows that CABINS improves its problem solving efficiency with the increase of case base size even after improvement of solution quality is saturated (i.e. after 2000 cases are accumulated). This result indicates that CABINS acquires effective search control knowledge for speeding-up through the accumulation of cases. The search efficiency reaches a stationary value after a large number of cases (about 6000 cases) are acquired. This can be explained by the more complicated nature of the repair control model which is created by the history of the repair goal/action selections and the repair results in the different problem situations. More diversity of cases are required to create the repair control model than the cases used for creating the user preference model. Therefore, it can be concluded that accumulation of cases can slowly but steadily improve the operational capability of the repair control model in CABINS.

6 Conclusions

We have proposed a model formulation (i.e. building and extension) method for the optimization task in ill-structured problem domains. We adopted task structure analysis as a method of initial model building and proposed a model extension technique based on case-based reasoning. We have developed a system called CABINS that implements the proposed methodology and shown that CABINS can create the user preference model and the repair control model with cases and use them to guide iterative solution optimization efficiently in ill-structured domains.

We experimentally demonstrated that, with an increase in case base size, CABINS improved the solution quality and problem solving efficiency for a benchmark suite of job shop schedule optimization problems. These results show that accumulation of cases can enhance the competence of the created models. Other job shop scheduling experiments in [33] show that CABINS outperformed the simulated annealing method [9], a well acknowledged optimization method, both in solution quality and problem solving efficiency. And the experiments in [20, 21] manifest effectiveness of the ways of exploiting failure cases in CABINS for speed-up learning in intractable optimization problems. More importantly from the modeling perspective, a user of CABINS can define domain-specific case descriptions easily by specializing the generic vocabularies found by the task-level analysis. CABINS can acquire the cases from user's interaction during the process of solution improvement, thus imposing low additional effort on the user but enhancing its problem solving capability. We think the proposed model formulation method of combining task-level analysis and case-based reasoning can provide a practical approach for solving ill-structured optimization problems.

As a limitation in the current status of our research, CABINS suffers from the *utility problem* [19] since CABINS requires more time for case matching and retrieval with increase in case base size. Although we can define the optimal case base size by monitoring the performance of CABINS for the problems in the domain [22], some knowledge filtering techniques [17] might be useful for improving efficiency of CABINS by dynamically eliminating redundant or incorrect cases in the case base.

Acknowledgment

The authors are indebted to Dr. Johan Vanwelkenhuysen at Osaka University (currently at INRIA, France) for insightful discussions on knowledge modeling.

References

[1] D. Allemang, Combining Case-Based Reasoning and Task-Specific Architectures, IEEE Expert 9, No.5 (1994) pp.24-34

[2] B. Chandrasekaran, Generic tasks as building blocks for knowledge-based systems: the diagnosis and routine design examples, The Knowledge Engineering Review 3, No.3 (1988) pp.183-210

[3] B. Chandrasekaran, Design Problem Solving: A Task Analysis, AI Magazine 11, No.4 (1990) pp.59-71

[4] B. Chandrasekaran and T.R. Johnson and J.W. Smith, Task-Structure Analysis for Knowledge Modeling, Communications of ACM 35, No.9 (1992) pp.124-137

[5] W.J. Clancey, Heuristic classification, Artificial Intelligence 27 (1985) pp.289-350

[6] B.V. Dasarathy (ed.), Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques (IEEE Computer Society Press, Los Alamos, 1990)

[7] L. Eshelman, D. Ehret and J. McDermott, MOLE: A Tenacious Knowledge-Acquisition Tool, International Journal of Man-Machine Studies 26, No.1 (1987) pp.41-54

[8] S. French, Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop (Ellis Horwood, London, 1982)

[9] D.S. Johnson, C.R. Aragon, L.A. McGeoch and C. Schevon, Optimization By Simulated Annealing: An Experimental Evaluation, Part I (Graph Partitioning, Operations Research 37, No.6 (1989) pp.865-892

[10] K. Kempf, C. LePape, S.F. Smith and B.R. Fox, Issues in the design of AI-based schedulers: Workshop report, AI Magazine 11, No.5 (1991) pp.37-46

[11] G. Klinker, KNACK: Sample-driven knowledge acquisition tool for reporting system, in: S. Marcus Ed. Automating Knowledge Acquisition for Expert Systems, Ch.5 (Kluwer Academic, Boston, 1988)

[12] G. Klinker, C. Bhola, G. Dallemagne, D. Marques and J. McDermott, Usable and reusable programming constructs, Knowledge Acquisition 3, No.2 (1991) pp.117-135

[13] J. Kolodner, R. Simpson and K. Sycara, A Process of Case-Based Reasoning in Problem Solving, Proceedings of the Ninth International Joint Conference on Artificial Intelligence (1985) pp.284-290

[14] J. Kolodner, Case-Based Reasoning (Morgan Kaufmann, San Mateo, 1993)

[15] L. Lewis and D. Minior and S. Brown, A Case-Based Reasoning Solution to the Problem of Redundant Engineering in Large Scale Manufacturing, International Journal of Expert Systems 4, No.2 (1991) pp.189-201

[16] S. Marcus and J. McDermott, SALT: A Knowledge Acquisition Tool for Propose-and-Revise Systems, Artificial Intelligence 39, No.1 (1987) pp.1-37

[17] S. Markovitch and P.D. Scott, Information Filtering: Selection Mechanisms in Learning Systems, Machine Learning 10 (1993) pp.113-151

[18] J. McDermott, Using problem-solving methods to impose structure on knowledge, Proceedings of International Workshop on Artificial Intelligence for Industrial Applications (1988) pp.7-11

[19] S. Minton, Learning Effective Search Control Knowledge: An Explanation-Based Approach (Kluwer Academic Publishers, Boston, 1988)

[20] K. Miyashita and K. Sycara, Learning Control Knowledge through Cases in Schedule Optimization Problems, Proceedings of the Tenth IEEE Conference on Artificial Intelligence for Application (1994) pp.33-39

[21] K. Miyashita and K. Sycara, Improving System Performance in Case-Based Iterative Optimization through Knowledge Filtering, Proceedings of the International Joint Conference on Artificial Intelligence (1995, to be published)

[22] K. Miyashita, A Case-Based Approach to Improve Quality and Efficiency in Ill-Structured Optimization: An Application to Job Shop Scheduling Ph.D. Dissertation (Osaka University, Osaka, 1994)

[23] R. Mizoguchi, Consideration on Design Process from a Knowledge Engineering Point of View, Journal of Japanese Society for Artificial Intelligence 7, No.2 (1992) pp.45-52 (in Japanese)

[24] R. Mizoguchi, Y. Tijerino and M. Ikeda, Task Ontology and its Use in a Task Analysis — Two-level Mediating Representation in MULTIS —, Proceedings of the Second Japanese Knowledge Acquisition for Knowledge-Based Systems Workshop (1992) pp.185-198

[25] M.A. Musen, L.M. Fagan, D.M. Combs and E.H. Shortlife, Use of a domain model to drive an interactive knowledge-editing tool, International Journal of Man-Machine Studies 12 (1987) pp.63-87

[26] M.A. Musen, Automated Support for Building and Extending Expert Models, Machine Learning 4, No.3-4 (1989) pp.347-375

[27] A. Newell, The Knowledge Level, Artificial Intelligence 18, No.1 (1982) pp.87-127

[28] A.R. Puerta, J.W. Egar, S.W. Tu and M.A. Musen, A multiple-method knowledge-acquisition shell for the automatic generation of knowledge-acquisition tools, Knowledge Acquisition 4, No.2 (1992) pp.171-196

[29] C.R. Reeves (Ed.), Modern Heuristic Techniques for Combinatorial Problems (Halsted Press, New York, 1993)

[30] N. Sadeh, Look-Ahead Techniques for Micro-Opportunistic Job Shop Scheduling, Ph.D. Dissertation (Carnegie Mellon University, Pittsburgh, 1991)

[31] E. Simoudis and J. Miller, The Application of CBR to Help Desk Applications, Proceedings of the Case-Based Reasoning Workshop (1991) pp.25-36

[32] J. Stout, G. Caplain, S. Marcus and J. McDermott, Toward automating recognition of differing problem-solving demands, International Journal of Man-Machine Studies 29, No.5 (1988) pp.599-611

[33] K. Sycara and K. Miyashita, Case-based Acquisition of User Preferences for Solution Improvement in Ill-Structured Domains, Proceedings of the Twelfth National Conference on Artificial Intelligence (1994)

[34] K. Sycara and K. Miyashita, Learning Control Knowledge through Case-Based Acquisition of User Optimization Preferences in Ill-Structured Domain, in: G. Tecuci and Y. Kodratoff (Eds.), Machine Learning and Knowledge Acquisition: Integrated Approaches (Morgan Kaufmann, San Mateo, To be published)

[35] B. Wielinga, G. Schreiber and J. Breuker, Modelling Expertise, in: G. Schreiber, B. Wiekinga and J. Breuker (Eds.), KADS: A Principled Approach to Knowledge-Based System Development, Ch.2 (Academic Press, San Diego, 1993)