TU

Institut für Automation
Abt. für Automatisierungssysteme

Technische
Universität
Wien

Projektbericht Nr. 183/1-30
Jänner 1992

ROOMS
A Case-Based System for Scheduling Study-Courses

A. Bezirgan, J. Dorn

Ausschnitt aus: Salvador Dali, "Die Beständigkeit der Erinnerung"

ROOMS

A Case-Based System for Scheduling Study-Courses

Atilla Bezirgan,
Technical University Vienna,
Institute for Automation,
Treitlstraße 3/183/1,
A-1040 Vienna
Fax ++43-1-563260
Phone: ++43-1-58801-8184
email: bezirgan@eiaut1.tuwien.ac.at

Jürgen Dorn,
Technical University Vienna,
Christian Doppler Laboratory for Expert Systems,
Paniglgasse 16
A-1040 Vienna
Fax ++43-1-5055304
Phone: ++43-1-58801-6127
email: dorn@vexpert.dbai.tuwien.ac.at

Topic: Scheduling

Category: long

ROOMS
A Case-Based System for Scheduling Study-Courses

Atilla Bezirgan, Jürgen Dorn

Abstract

Case-based reasoning has become a promising concept for a lot of applications where traditional reasoning techniques fail because of complexity and knowledge elicitation problems. Although, the problems that motivate case-based reasoning exist in scheduling too, no serious attempt has yet been made to show its benefits for this domain.

We use a simplified scheduling problem to show how case-based reasoning can be applied to cope with complexity due to constraint interaction in scheduling. By means of an existing experimental system that was implemented in Prolog we demonstrate how scheduling heuristics can be learned by explanation-based techniques. Although our system has only a very small case base, we show how it will grow with further problem solving experience.

1 Introduction

Scheduling is the allocation of resources over a period of time in order to accomplish some jobs and it differs from planning in that it does not select the actions that are necessary to accomplish a job. Resources are typically those materials, tools, and individuals which are to be consumed or used by an operation. Each resource is associated with a formal specification of its characteristics and capabilities. In production process scheduling a job is usually identified with a product which has to meet certain quality requirements [3]. However, in other domains like scheduling of crews [12], school classes [10], or vehicles routing [13] a job cannot be identified with such a product.

With each job a process plan may be given that contains the necessary operations, a set of explicit constraints concerning the sequence of operations, and a set of resource requirements. When a number of jobs has to be executed together, the composition of their resource requirements imposes additional constraints on the sequence of operations which prohibit simultaneous use of nonsharable resources.

In scheduling valid sequences of operations are determined by explicit constraints in the plans and implicit constraints imposed by the availability of the resources. The task of scheduling jobs and resources is difficult for at least two reasons. First, we have to deal with the combinatorial complexity due to multiple ways of job accomplishment [4]. Second, conflicting objectives may hinder the definition of an undisputed optimality measure [15].

First attempts in operations research to overcome these problems were based on formal mathematical models [3]. Models were developed for a wide range of tasks, but each was restricted to applications with small numbers of jobs and resources. Beside the problem of complexity the effort required to develop a model for a new scheduling task is very high.

As a consequence, many researchers propose heuristics to reduce the complexity. These are mainly rules of thumb, that can be represented easily in form of production rules as described in [1]. However, the problem with these heuristics is, that usually no generally accepted domain theory or formal model exists. Human experts learn them by practising and often they are not aware of any formal model of their knowledge. In expert system projects it is the most difficult problem to elicitate this knowledge [6]. Additionally, the knowledge elicitation process will never be complete and therefore expert systems will react strange on the boundaries of their knowledge. A further problem is the difficulty to maintain these rule-based systems.

Deep modeling of domains were proposed to overcome these problems. The knowledge in these models is generic and every task may be solved using a causal theory about the domain.

With object-oriented representation [5] the modeling of jobs, resources, and their characteristics is supported and constraint-based reasoning can be used to represent and process constraints between jobs and resources as proposed in [2]. These techniques are good in describing physical knowledge about an application and they are easy to maintain. Nevertheless, the complexity in search for a schedule is not reduced by them. Heuristics are still necessary.

Case-Based Reasoning (CBR) [14] combines advantages from heuristical and deep modeling. It promises a solution for the knowledge elicitation and maintenance problems, because it uses old cases to solve actual problems. Cases contain at least a description of a problem and its solution. One of the main issues in CBR is the possibility to use analogous cases to find a solution for a problem that was never solved before by the system. In order to achieve such a functionality old cases are stored in a structured way in a case base. Cases are attributed with their characteristics and links between the cases are established to represent similarities between them. To solve a problem, the case-based system (CBS) looks for the most promising old case. Usually, some features of this case must be adapted to the actual problem. Then the system tries to solve the problem with this adapted case. In contrast to approaches with deep modeling the evaluation component must not search exhaustively for a solution, but applies the causal domain theory only to perform a simulation. If this evaluation is not successful the system explains the failure and repairs the plan by performing an exhaustive search in the causal model. The repaired plan and the explanation of the failure are stored in the case base. The latter is used to anticipate failures in the future.

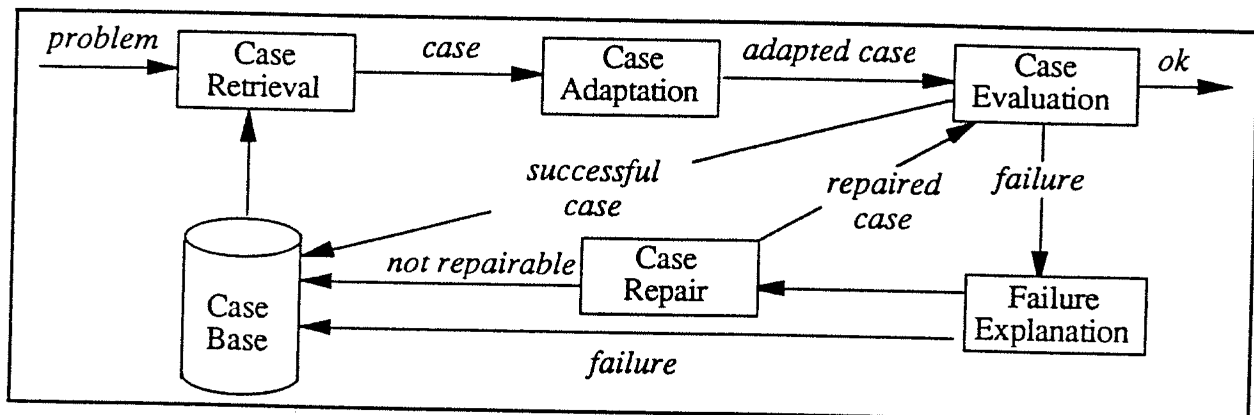

Figure 1: The General Structure of a CBS

The application of CBR in domains like legal reasoning and diagnosis in medicine appears straight forward, because these domains are often based on cases. When a lawyer or a doctor is confronted with a new case, he uses his experience gained from previous cases.

In scheduling theory the concept of cases is unknown and therefore the adaptation of CBR appears more difficult. On the other hand, we believe that human schedulers use something like cases to overcome the inherent complexity in scheduling. For example, a human scheduler may have successfully scheduled several important jobs in the morning. Other important jobs that he scheduled in the afternoon failed. Without using any causal theory of the domain he will schedule the next important job in the morning by using his experience.

We have developed a CBS that shows what cases can be in scheduling and how the different components of a CBS could be constructed and interact. We discuss how such a CBS is to be used to overcome existing scheduling problems. Our CBS operates in the domain of study-course scheduling and is written in Prolog.

We are not aware of any comparable work in the literature although there are some people trying CBR for scheduling. SMARTplan [9] is announced, but no results are known to us. Some systems like TRUCKER [11] have some scheduling flavor, but emphasize planning. Clavier [7] as the only serious candidate is not comparable either since a very specialized domain is used where cases seem obvious.

2 The Study-Course Scheduling Problem

Our problem comes from a simplified view of organizing study-courses. The organizer has at his disposal a number of lecturers and course rooms. In scheduling study-courses he has to take into consideration constraints on the sizes of and the equipment in the rooms and on the abilities and the availability of the lecturers. There are five main entities in our domain:

1. We use a bounded, linear, discrete **time** model with a one hour granularity and a fixed zero point. Intervals are represented by sorted lists of time points and may have holes in them, e.g. the zero point being 1.1.1992 00:00, the time point 61 represents 3.1.1992 [13:00–14:00) and [61, 62, 64, 65] is a valid interval.
2. There is a fixed number of course **rooms**, each with a unique name, a fixed capacity and some equipment like a projector or a blackboard, e.g. room(r1, 50, [projector, video]). Each room has a utilization rate defined as the ratio of the number of hours it has been used in a given interval to the interval length.
3. There is a fixed number of **lecturers**, each with a unique name. Each lecturer can teach certain subjects and has a calendar indicating his availability, e.g. lecturer(amanda), canTeach(amanda, [math, networking]), hasTime(amanda, [4, 5, 6, 7,...]). Lecturers also have utilization rates defined like the ones for rooms except that the number of hours of availability within the given interval is used as divisor.
4. A **courses' list** contains all the courses already scheduled. Each course entry has a unique name, the scheduled time, the lecturer and the room used, e.g. [course(math1_92, [4, 5], amanda, r1), course(programming_92, [5, 6], john, r3)].
5. A **requirement** is a triple of constraints that the schedule of a course has to satisfy. There are time, lecturer, and room constraints, e.g. [tdur(Time, 2), leq(Lect, amanda), rmcapbetween(Room, 20, MAXCAP)], where tdur(Time, 2) means that we have a two hour course, leq(Lect, amanda) means that the lecturer has to be amanda and rmcapbetween(Room, 20, MAXCAP) means that the capacity of the room should be at least 20.

An entry in the requirements' list contains the name of the course for which these are the requirements, the requirement triple and an optional list of search strategies to be used, e.g. "minimally utilized lecturer first". The requirements' list contains an entry for the current problem and a history of all the requirements used in constructing the courses' list. This will be shown to be needed to do certain analysis on scheduling decisions made earlier.

The constraints used in a requirement are pure conjunctions of constraint primitives. These primitives for the three constraint classes are given below.

Please note that in the descriptions given below output parameters have an initial upper case letter and input parameters have an initial lower case letter.

Time:
- tdur(TimeInterval, dur): The 'TimeInterval' should contain 'dur' time points.
- tbetween(TimeInterval, lowerBound, upperBound): All time points in the 'TimeInterval' should lie between 'lowerBound' and 'upperBound'.
- tclosed(TimeInterval): The 'TimeInterval' should not contain holes.
- trepeat(TimeInterval, inTimeInterval, firstTimeInterval, stepTime): The 'TimeInterval' should contain the 'firstTimeInterval ' and every time point at a distance of a natural multitude of 'stepTime' from any point in the 'firstTimeInterval' within the 'inTimeInterval'.

Lecturer: leq(Lecturer, lecturer): The 'Lecturer' should be 'lecturer'.
canTeach(Lecturer, subject): The 'Lecturer' should be able to teach the 'subject'.

Room: rmeq(Room, room): The 'Room' should be 'room'.
rmcapbetween(Room, capLower, capUpper): The capacity of the 'Room' should lie between 'capLower' and 'capUpper'.
rmequip(Room, equipmentList): The 'Room' should have all the equipment in the 'equipmentList' in it.

The domain is described in a subset of Prolog. It contains all the primitive predicates mentioned above and some other supporting primitive predicates. A conjunction of these primitives may be used to build clauses for compound predicates. Prolog variable binding is used in parameter passing and a predicate may have several clauses. There is no 'cut' and no negation primitive in the language. Usually no recursion is used at this level of description. All language primitives are free of side effects.

The task is to find a schedule for a not yet scheduled course so that

- the requirement for it is satisfied,
- the chosen room is not used for two different courses at the same time,
- the chosen lecturer is not scheduled for two different courses at the same time and
- he is available at the scheduled time.

The main sources of complexity are

- interactions of constraints on valid schedules. Having chosen lecturer and time there may be no room available.
- influence of a scheduling session on future sessions. Choosing an unnecessarily large room may make it impossible to schedule a course in future for which this room would be needed.

These interactions rule out exhaustive search in the space spanned by the dimensions of room, lecturer, and time as a permanent solution since:

- the search space is to large (O(dim(room) * dim(lecturer) * dim(time_interval))),
- constraint interactions often lead to invalid schedules forcing the search to be continued (The problem is not decomposable along the three dimensions),
- there is no simple way of optimizing the search process (In our case this is just an assumption. We plan to implement an optimized non-case-based solution for comparison purposes),

moreover

- exhaustive search doesn't solve the problem of the system influencing itself.

Removing one of the first three characteristics above would lead to an improved performance of the search process. We try to remove the third property by using domain knowledge for optimization. The fourth characteristic is dealt with by applying and learning search heuristics which we call strategies.

Please note that we have a well-defined problem and a complete and consistent domain theory. For now this is the class of problems we consider. We do not deal with problems like that of open-textured descriptions or incomplete domain theories. Moreover, we are satisfied by finding any solution and do not look for a good one in any sense.

3 The Case-Based Approach

The main idea in dealing with complexity due to constraint interaction is to avoid exhaustive search by partitioning the search space along dimensions derived from the domain theory. Each part in such a partition is called a concept. The partitioning we have in mind has the following characteristics.

- The concepts are not known or not constructible at the time of system implementation, e.g. because there are too many to be considered manually. So dynamic concept acquisition is needed.
- Either the solutions of all problems belonging to one concept are easily transformable into each other or there is a distinct problem-solution-pair from which the solutions of all problems belonging to the concept can be easily derived (adaptability criterion).
- Problems for which a solution can be found by the same specific way of applying domain knowledge, i.e. having the same causal configuration belong to one concept.

We use a CBS to achieve such a partitioning. Each case represents a point in the problem space to which a solution has been calculated by exhaustive search. Hereby a case contains at least the description of a problem and its solution. Given a new problem the retrieval component delivers a case from which the adaptation component can most probably generate a solution. The adaptation component performs local search and simple transformations on the old solution in the retrieved case [8] to propose a solution to the new problem. The proposed solution is evaluated to see if it really is a solution to the new problem. If it is, we are finished. Otherwise, the explanation component does a causal analysis to find the reason why the proposed solution failed. The explanation usually consists of some features of the problem predicting the failure and a constraint, whose violation led to the failure. It then finds by exhaustive search a valid solution to the new problem and alters the case-base so that future cases with the same problem do not lead to an exhaustive search. It does the latter by using the features in the explanation as an index. It also puts the violated constraint into the new case to guide local search in adaptation using this case in future. Note that in our system the repair component is replaced by a component doing exhaustive search.

From the above description you can see that the concepts to be acquired are implicitly defined by the adaptation component. Those problems to which a solution can be constructed by the adaptation component from a certain case build a concept. For the further discussion we introduce a relation r(X, X') which holds between two cases X and X' if and only if X can be adapted to become X', i.e. r(X, X') = adapt(X, Y, X', Y') and $Y \in$ solution(X) and $Y' \in$ solution(X') and (X, Y) $\in$ case-base. A concept is then defined by $c(X) = \{X' \mid r(X, X')\}$. The language to be used in concept formation is the causal language of the explanation component.

Some requirements for the CBS are:

- The frequency of concept reuse should be exploited to improve efficiency and save space, i.e. cases used often should be found faster (improvement through exercise) and cases not used frequently should be abolished (forgetting when not used).
- After having been used to solve a certain finite set of problems the system should be able to adapt successfully for every further problem for a given fixed domain.
- Requirements on the retrieval component:
  - If $X \in c(X_{old})$ then retrieve(X, X_{old}).
  - The retrieval should be fast.
- Requirements on the relation R and the adaptation component.
  - The system should learn from its own problem solving experience. Specially no problem should be solved twice by exhaustive search. This is tantamount to demanding reflexivity of the relation R.

- Other characteristics of the relation R and hence of the adaptation component are desirable but hard to guarantee. Having symmetry and transitivity would lead to equivalence classes of problems and to disjunct concepts. To guarantee these characteristics one would have to redefine R and impose some tough restrictions on the domain and the system. We did not do this. Thus in our system the concepts are overlapping and the acquired concepts are dependent on the sequence of the problems the system is confronted with.
- The average number of elements in a concept should be large (average concept size = adaptation breadth big) and the total number of concepts should be small. This is to keep the maximum size of the case-base small. This is also accomplished if there are many concepts but only a small subset is required for a certain application environment.
- The adaptation should be fast.
- The old solution should be changed as little as possible to avoid side-effects.

- Requirements on the evaluation component:
  - Should support failure explanation by delivering an evaluation trace.
  - The evaluation should be fast.
- Requirements on the explanation component:
  - To enable the retrieval component to be fast the explanations should preferably be based on superficial features of problem descriptions using derived features only when necessary.
  - Given $X' \notin c(X_{old})$ and retrieve(X', X_{old}) let $p(X)$ be a predicate characterizing features predicting the failure. Then $p(X)$ should be such that $(\forall X: X \in c(X_{old}) \rightarrow \neg p(X))$ and $p(X')$ (demanding instead of $p(X')$ $(\forall X: X \in c(X') \rightarrow p(X))$ would lead to concept discriminating explanations. However, since we have overlapping concepts such explanations cannot be built).
  - The explanation may be slow.

The self-influencing problem is solved to some extent by analyzing earlier cases to learn scheduling strategies for avoiding problems due to self-influencing in future. This is a form of explanation-based learning. By this mechanism strategies such as "smallest room first" can be learned, after being unable to schedule a course because a room of the required size is occupied by another course for which a smaller room would have been sufficient and available.

4 ROOMS

A case-based system has been designed that satisfies the above requirements. The components of this system are described below.

The **case-base** is a structure for organizing a set of cases. ROOMS case-base is organized as a binary tree, where the internal nodes contain clauses and the leaves contain cases. Arcs between nodes are marked by true resp. false. The heads of the clauses have problem descriptions as parameters, i.e. 'courseList', 'requirementList'. The Prolog representation is as follows. cb(nodeContent, trueBranch, falseBranch) where nodeContent is either a clause or a case. The predicates in the nodes are pure conjunctions of negated and non-negated primitives defined by the explanation component. Initially the case-base is empty. The first solution is constructed by exhaustive search and the first non-empty case-base is a tree consisting of one leaf node containing the first case.

A case contains a scheduled course, the problem description consisting of courses' list and requirements' list belonging to the scheduled course, a list of the cases successfully adapted or adaptable from this case or the number of such cases and the age of the last such case, a list of the strategies used in finding the solution, e.g. "minimally utilized lecturer first" or "smallest room first", and a list of constraints, e.g. "choose only available lecturers" or "if the courses for a lecturer are subsequent choose only near rooms". The cases are given sequential case num-

bers which reflect the age of a case. A case is represented by case(caseNo, courseList, requirementList, resultCourse, supportingCaseList, strategyList, constraintList) where

1. 'caseNo' is as explained above. This is used in "forgetting when not used",
2. 'courseList' and 'requirementList' represent the problem,
3. 'resultCourse' represents the solution to the problem,
4. 'supportingCaseList' represents a set of problem-solution pairs adapted or adaptable successfully from the the current case. This is needed to make it possible to check if a new concept subsumes an old one and can hence replace it. It further serves the purposes of "forgetting when not used". If this list gets very large all supporting cases are removed and a count of the supporting cases plus the caseNo of the last supported case is used instead. Cases in this list do not have a supportingCaseList component.
5. strategyList is as explained above. Some strategies are predefined and may be used in the problem description, e.g. "minimal utilization", others are learned by the system while explaining failures, e.g. "smallest room first".
6. constraintList is as explained above. It is created by the failure explanation component and used by the adaptation component to guide local search.

Note that the calendars of the lecturers are taken to be fixed predicates to demonstrate the different treatment of fixed and variable constraints. These could be included in the problem description and hence in the cases too.

Each case ever processed by the system is contained at most in one place in the case-base.
An example case:

```
case(112, [...],
   [[c140, (tbetween(Time, 18, 29), tdur(Time, 3), tclosed(Time),
     canTeach(Lecturer, math),
     rmcapbetween(Room, 160, MAXCAP)),
     [min(utilization(Lecturer))]],...],
   course(c140, [20, 21, 22], amanda, r3), [], [], []).
```

Figure 2: A Sample Case

The **retrieval component** retrieves for a given problem description a case from the case base. The retrieval is done by a binary search algorithm. In each internal node the predicate found there is evaluated using the current problem description as the actual parameter. These predicates are constructed by the explanation component. Two problem descriptions both satisfying such a predicate are similar in some sense (avoiding the same problems, being syntactically similar in structure and parameters,...). Depending on the result of the evaluation the arc marked true resp. false is traversed to get to the next node. The process terminates when a leaf node is reached. The leaf node contains the case that is passed on to the adaptation component. For example given an appropriate courses' list (e.g. the same as above) and the requirements' list [[c231, [tbetween(Time, 25, 30), tdur(Time, 2), canTeach(Lecturer, networking), rmcapbetween(Room, 100, MAXCAP), []],...] case 112 given above could be selected for adaptation.

The **adaptation component** does a fast and mostly syntactic analysis of the new problem situation and of its differences to the old one. It has rules for changing the old solution depending on this analysis. The analysis is concerned with the structure of the time-, lecturer- and room-requirements and with the values of parameters of primitives (structural and parametric adaptation). The adaptation rules are not guaranteed to generate a valid solution to the new problem. Fig. 3 shows some of these rules.

ROOM-RULE 4:
IF there are rmmincap and rmequip primitives in the requirement AND
(the room-requirement is not a structural subset of the old requirement OR
the parameters of a primitive in the new and old requirement are not equal)
THEN
IF the room in the old solution fulfils the room-requirements
THEN use it as the new room
ELSE use any room as the new room

TIME-RULE 7:
IF there is a tbetween primitive in both the new and the old requirement
THEN move the old case temporally to the new time (add the difference of the lower bounds of the new and old tbetween to all time points in the old case) before using any other rules

Figure 3: Adaptation Rules

If there are any search strategies in the requirements' list or in the old case a local search for the relevant parameter is performed using these strategies. If by this a strategy conflict arises the strategy in the requirements' list is used. If there are any constraints in the old case these are used in local search, too. If there is any parameter value satisfying the constraints, it is found and used. Otherwise any value is used causing a failure later on and thus triggering the learning mechanism. Since the search is local this is a fast process.

For example case 112 above could be used by the adaptation component to find a schedule for the course c231 above. The TIME-RULE 7 would be applied first. One rule saying that if the old lecturer can teach the required subject than he should be selected for the new solution is then applied. After several such adaptations we could get the proposed solution course(c231, [27, 28], amanda, r3).

The **evaluation component** checks all the constraints on the components of a solution. It contains the domain theory written in a subset of Prolog and processed by a meta-interpreter. Given an instantiated solution variable this works as a constraint checker. The explanation component calls the same meta-interpreter with an uninstantiated solution variable to do exhaustive search for solutions. The meta-interpreter also returns derivation paths which are needed for failure explanation. If the evaluation is successful the case is asserted into the case-base as a supporting case.

The **explanation component** is used to process cases for which adaptation failed. This is recognized by the evaluation component. First a solution to the problem is found by exhaustive search. Then the old case and its supporting cases are tested to check if they are adaptable from the new case. If all of them are adaptable then the old case is replaced by the new one and the old case becomes a supporting case. Of course this is only possible if the supporting cases have not yet been replaced by their number to save space in the case-base.

If such a concept subsumption is not possible a set of features that lead to the failure is looked for. In order to find these features a causal model of the scheduling process is used. The node in the case-base containing the old case is then split in two, the node itself becoming an internal node. In this internal node a predicate is asserted that checks a given problem description to find out if it has the features that lead to the failure. The appropriate successor node will contain the old case the other the new case. A violated constraint found responsible for the failure is also put into the new case.

If in the first step a solution cannot be found even with exhaustive search, then a trial is made to develop a search strategy. This is done by looking at possible mistakes made in past scheduling sessions that lead to the current failure. If a strategy can be developed it is asserted into the strategy list of the old case in the case-base which is found to be the culprit. If no stra-

tegy is found, no change is made. In any way, since no solution is found to the new problem an error message is issued and the case-base is left unchanged otherwise.

The case-base is cleaned up regularly, e.g. every time after processing a fixed number of cases. Cases with relatively few supporting cases and cases which have not been used for a very long time are deleted from the case-base. This is done by removing the appropriate leaf. The sibling tree of the leaf is moved up one node deleting the internal father node of the leaf. Another cleanup operation is the replacement of supporting cases by their number and the caseNo of the last supported case. This is done when a case has too many supporting cases.

5 Conclusion

We have developed a CBS for a simplified scheduling problem that demonstrates how case-based reasoning can facilitate scheduling. The system is implemented partially at the time of writing this paper and will probably be completely implemented by publication time. The case and case-base representations, the retrieval component, and the evaluation component are fully implemented, the adaptation component is partially implemented.

Besides the implementation we have shown how cases may look like in scheduling. There are other scheduling models in which cases would look different. For example, in many applications it would be necessary to regard requirements for a set of jobs simultaneously.

When such a simplified scheduling system like ours is used, it will soon reach its limits. For example, if a lecturer is scheduled for two subsequent courses in different rooms, a problem may occur that the distance between both is too far. In such a case a new concept "distance of rooms" must be integrated in our domain model. Then the system can learn again new cases, constraints, and strategies for this extended domain model.

Our system is able to learn new scheduling strategies and constraints. It overcomes thereby problems that occur in traditional expert systems during knowledge elicitation. If such a CBS is used for a new application, only the causal domain theory must modeled. Heuristics to improve the solution process are learned automatically by the system. Moreover, the system only learns heuristics that are necessary and singular solutions are thrown away if they are not used for a long time.

In contrast to a general CBS our system is restricted in some aspects. If the CBS should be applied to a domain such as crew scheduling, it must be extended in a broader sense, because concepts in our adaptation component are not generic enough to deal with groups of resources (crews). Another problem that cannot be solved by our system is the deletion and movement of courses that were scheduled earlier. To solve the latter problem we have to use experience over a lot of old cases.

One feature that appears very important to us for the retrieval component is the use of fuzzy logic to represent similarities between cases. Since we use discrete discrimination of cases, we do not have a measure how similar two cases are. Another aspect that we will investigate is how this retrieval component may be implemented on a commercial data base system. A further not yet studied aspect in the CBS are user interfaces for these systems.

Last but not least an important task will be to make performance tests which compare solutions implemented in a rule-based system with a solution with our CBS. Beside the advantage of easier knowledge acquisition, we hope that we also gain the advantage of finding solutions faster for most given requests.

6 References

[1] G. Bruno, E. Antonio, and P. Laface. A Rule-Based System to Schedule Production. IEEE Computer, Vol. 19, pp 32–39, 1986.

[2] Mark S. Fox. Constraint-Directed Search: A Case Study of Job-Shop Scheduling. Pitman, London, 1987.

[3] S. French. Sequencing and Scheduling: An Introduction to the Mathematics of Job Shop. Chichester, Ellis Horwood, 1982.

[4] Michael R. Garey and David S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness.Freeman and Co., 1979.

[5] Tim J. Grant. An Object-Oriented Approach to AI Planning and Scheduling. Proceedings of the Seminar: Expert Systems and Optimization in Process Control, Uxbridge, UK, pp 11–27, 1985.

[6] Frederick Hayes-Roth, Donald A. Waterman, Douglas B. Lenat.
Building Expert Systems: A Tutorial. Addison Wesley, 1983.

[7] Daniel Hennessy and David Hinkle. Initial Results from Clavier: A Case-Based Autoclave Loading Assistant. Procceedings of the DARPA Case-Based Reasoning Workshop, pp 225-232, 1991.

[8] Janet Kolodner, Christopher Riesbeck. CBR Tutorial: MA2. 11th International Joint Conference on Artificial Intelligence, Detroit, Michigan, 1989.

[9] Phyllis Koton. SMARTplan: A Case-Based Resource Allocation and Scheduling System. Procceedings of the DARPA Case-Based Reasoning Workshop, pp 285-289, 1989.

[10] Mark Mallett. Sorting out Schedules.
BYTE, No. Dec, pp 263 – 268, 1991.

[11] Mitchell Marks, Kristian J. Hammond and Tim Converse. Planning in an Open World: A Pluralistic Approach. Procceedings of the DARPA Case-Based Reasoning Workshop, pp 285-289, 1988.

[12] Ernesto M. Morgado, and João P. Martins.
Scheduling and Managing Crew in the Portuguese Railways.
Proceedings of the World Congress on Expert Systems, pp 377–384, 1991.

[13] Jean-Yves Potvin, Guy Lapalme and Jean-Marc Rousseau.
Integration of AI and OR Techniques for Computer-Aided Algorithmic Design in the Vehicle Routing Domain.
Journal of the Operational Research Society, Vol. 41, No. 6, pp 517–525, 1990.

[14] Christopher Riesbeck, Roger C. Schank. Inside Case-Based Reasoning. Lawrence Erlbaum, 1989.

[15] Stephen F. Smith, Mark S. Fox and Peng Si Ow.
Constructing and Maintaining Detailed Construction Plans: Investigations into the Development of Knowledge-Based Factory Scheduling Systems.
AI Magazine 7(4) Fall, pp 45–61, 1986.