Not final version
Cat. title differs.

Constraint Satisfaction for Resource Management using ATMS: a Timetable Design Support System.

Luis Montero

MSc Student

Department of Artificial Intelligence
University of Edinburgh
28th July 1989

Abstract

Truth Maintenance Systems (TMS) have turned out to be very useful for many kinds of constraint satisfaction problems, for example qualitative reasoning or scheduling. A particularly difficult constraint satisfaction problem, very well known by course organisers in universities is the arrangement of lectures according to teachers, students and department constraints and preferences, so that the problem is solved and everyone is pleased.

The proposal of this project was due to both the interest in knowing how to solve such a problem, and the fact that a version of de Kleer ATMS, a very advanced and efficient TMS system, had been built by Peter Ross, and was available in Edinburgh PROLOG.

This thesis first outlines some of the reasons why an ATMS is useful for a timetabling problem, how it is used together with PROLOG, in order to produce a system for solving that problem, and how that system works.

Acknowledgements

I have received technical help and encouragement from my two supervisors, Tim Smithers and Dave Robertson, as well as Graeme Ritchie, whose expertise and up to date information about previous real timetables data was very useful. Peter Ross was very kind to solve my queries about using his ATMS implementation, that was absolutely necessary for this project. Ken Currie and many other people, both in A.I.A.I. and D.A.I., especially my MSc colleagues, gave me useful information and help.

Far from the department, I would like to thank the Stevenson Foundation, for the Scholarship which provided me with economic support to attend the MSc course, and do this work. My family and friends, either here or in Spain, and my colleagues in my previous job and University, who supported me in many bad moments, and encouraged me to follow an "Artificial Intelligence" career, have played a better role in this work than they can imagine. Thanks very much to all them.

Table of Contents

Chapter 1

Introduction

The problem of constructing a Timetable Design Support System, is just a particular example of a more wide category known as "resource management problems". In this case, the "resources" are teachers, students, subjects, rooms, times, and so on, and their management must lead to a timetable where lectures are set. A big research effort is being made in Artificial Intelligence in order to find a right way to solve "Resource management problems" as they appear everywhere, in design and manufacturing, transport, banking, economic models, and many other.

This project was very suitable to attempt, since, at least, *the goal was clear* (although the rest of the work had to be done from scratch) and "common sense" had to be used rather than "specialised expertise". In addition, I was well aware of the problem, since I was attending a course with many constraints, and information was readily available.

The proposal of the project was due to Tim Smithers, who thought of it as a way of using ATMS for a problem completely different from the usual ones in the Edinburgh Design System (EDS) project, more oriented to engineering design. I found in it a very good opportunity of learning about "resource management problem solving", and using Truth Maintenance Systems that I had found very interesting while studying them in the module "Knowledge Representation and Inference II".

1.1 Brief History

In the initial proposal, the first step was trying to design a "tutorials" timetable system that would lead, at the end of the project to a "lectures" timetable system. Later, in a meeting with Graeme Ritchie, on 2th May 1989, another proposal about constructing an "Exams timetable design system" was made. Since all these problems were really related (there were only differences about where most constraints are: students, teachers, rooms), I found that attempting everyone of them was a waste of time. I'd better concentrate on solving ~~solve~~ the final goal of the project: the "lectures timetable", define clearly ~~very well~~ this particular case, learn about it, and then any more general? ~~other~~ system could be attempted either by me or by others. Fortunately, as it was an open project, designed to experiment, all those details were left to me and I had the opportunity to learn and practice a lot with it.

The first step was looking for literature. I began with the three references mentioned in Tim Smithers' proposal: [Rich 78] [deKleer 84] [Ross 87], which were very useful since my knowledge about advanced problem solving techniques and ATMS was rather limited. However, as the project was taking form, I realised that other information, mainly technical reports about real applications related to my project would be very welcome. I began looking for references about it, either in proceedings of conferences, journals, books, or asking everyone who could give me any information about it, inside and outside the University.

Unfortunately, I could not find in the literature any technical report about using ATMS in this kind of project. What I found were some articles about applications to qualitative reasoning or scheduling, that I will describe it in more detail in the literature review chapter.

Another thing I did was to take advantage of seminars and courses about related subjects (e.g.: scheduling), given at the University of Edinburgh and contact the people who had been involved or interested in ATMS and related projects. This will also be described in the following chapter, together with the literature review.

Anyway, most of these contacts were too late in time, hence I had to start my project much earlier. What I did, since the beginning of April 1989 was to put my early ideas into practice. One thing I had clear in my mind is that the Timetable Design Support System needed two phases:

1. Finding automatically an initial solution, as optimal as possible, according to the initial constraints.

2. Allow the user to make modifications to

    - The initial solution Timetable
    - The input Constraints

    and even the possibility of resolving the problem automatically.

HINT: This second phase could be repeated as many times as desired.

I considered the first phase NECESSARY, as hundreds of constraints appear in this kind of problems. Therefore, I discarded the approach used in EDS (only second phase, without automatic "resolving": everything is left to the user), as it could make it completely useless. Therefore, I started attempting the first phase, to construct an "initial problem solver". I realised I had some problems:

- My lack of experience in using special "problem solving techniques" and implementing them in PROLOG (I will describe some of them while commenting [Rich 78] in the Literature Review).

- My lack of experience in working with Peter Ross' ATMS inside PROLOG.

- The fact that the complete Timetable problem, even if not very well defined, at this moment, seemed to be very complex to face it initially and, in addition, real data was not available yet.

Therefore, I decided to attempt a classical well-known, well-defined problem, whose solutions were known: The Geoffrey Marnell's problem, proposed by Peter Ross for a tutorial in the module "Knowledge Representation and Inference I". In addition, I had solved it a different way, using LISP, and therefore, I could be more sensitive to improvements in efficiency, and goodness of the problem solver algorithm.

The problem is as follows (I included assumption numbers, as I was using ATMS, as well):

```
Mr. Craft, Mr. Skill, Mr. Art and Mr. Wood are four schoolteachers.
Although each teaches classes in exactly two subjects,
Assumption 201: only one of the four teaches mathematics.

In addition:
Assumption 202: a) three of them teach English
Assumption 203: b) two teach Science
Assumption 204: c) two teach History
Assumption 205: d) Peter does not teach English
Assumption 206: e) Both Simon and Mr. Skill teach History
Assumption 207: f) Steven teaches Science
Assumption 208: g) Mr. Craft does not teach any of the subjects
                   that Charles or Mr. Art teaches.

What is the full name of each teacher, and
who teaches what subjects?
```

I found some techniques, used in PROLOG module, and studied in [Rich 78], like "Best-First-Search using Agenda" or "tree processing", very useful for this problem. I began working on it, on 13th April 1989, and gave a demonstration of the final version on 25th April 1989. I discovered some things in it:

- If there is a clear definition of the objects in the system, and information is used properly, so that impossible situations can be discarded in advance, the complexity of the problem is vastly reduced. This problem, whose possible combinations are over 300,000, was solved using only 21 steps (467 if all possible solutions are required, which implies a complete exploration of the tree). Of course I had to use an AD HOC strategy (it was another reason to face only "lecture timetabling" in the future and define very well the problem).

- A tree structure was very useful, and I planned to include it in the real problem, as a powerful debugging tool, as will be seen.

- ATMS caused some problems, as I was using an early version. When I got the right one, I could see that it worked properly in this problem, the way I had expected, and realised that the structure I had in mind for the timetable system was possible. I will describe this in Chapter 4.

I will not describe my "Geoffrey Marnell's problem solver" in more detail, as it is not the subject of this project.

After this, it was time to define clearly the problem, in the following areas:

- Constraints used in the real Timetabling problem

- Database Scheme, including description of ATMS nodes and assumptions, the debugging tree, the history of a session, and other information options in the system

- Problem solver, divided in two phases:

  1. "First phase": Initial Problem Solver
  2. "Second phase": design of a system to maintain consistency in the information after every possible combination of changes, which include:
     - addition of constraints
     - deletion of constraints (necessary in the system, although undesirable for ATMS, as will be seen)
     - movements of lectures in the timetable (in order to allow the user performing changes in it himself)

     so that "automatic *resolve*" option could restore a valid state, with a solution, after any set of changes. This option had to be implemented so that new changes could be added later and further "resolve"s would restore a valid state.

     Other useful options, like "load/save" were desirable.

- User Interface: windows, menus, use of the mouse, a nice presentation of the Timetable and user information, to facilitate its use.

These points will be commented in chapters 3, 4, 5 and 6, respectively. Chapter 2 is dedicated to the literature review and foundations, that provided me with the adequate background to attempt the project.

Chapter 2

Literature Review and Foundations of the Project

I would like to explain in this chapter the foundations of Problem Solving and ATMS, proposed improvements to ATMS and other implementation subjects, and how it is used in real applications. Even though it is actually a long chapter, it is not possible to cover all points in detail, since I found wide information, but quite diverse (and often not very related with the project), mainly about "improvements" and "applications".

Therefore, I will cover in more detail the foundations of ATMS. There will be a section to explain how Peter Ross' ATMS works, that will be easily understandable after the foundations section. Another section will cover the meetings and seminars that gave me some background to go on. I hope that this chapter will be useful, not only for making the project clear to the reader, but to let future MSc students know how I managed to get information, how to select it for a completely new problem, and all the difficulties I found.

2.1 Problem Solving Techniques

A picture of artificial intelligence from a "Problem Solving" point of view, is given in [Rich 78], mainly in the three first chapters, and the second part of the book, dedicated to "Advanced Problem Solving Techniques".

This book gave me the first ideas about how to face the project. Some of the techniques proposed in it, like the use of Agendas in "Best-First-Search" were used in the system. On the other hand, the use of ATMS and the fact that the project was too specific, made other techniques proposed in the book useless (A* and AO* algorithms, Planning, ..., as will be seen), but even so, I consider it a very good beginners guide for problem solving.

2.2 Foundations of TMSs and ATMS

The first Truth Maintenance Systems appeared in the late 70s as a way to make problem solvers work more efficiently, reducing the search space. As deKleer says in [deKleer 86a], there were two problems:

- How can this space be searched efficiently, or

  How can maximum information be transferred from one point in the space to another.

- How, conceptually, should the problem solver be organized

TMSs were proposed as a solution since in many problems,

> For most tasks, there is a great deal of similarity among the points of the search space. As a consequence efficiency can be gained by carrying results obtained in one region of the search space into other regions

[deKleer 86a]

The proposed Problem Solver Architecture should consist of two parts:

- The Problem Solver, containing the set of rules governing the problem

- The TMS, which records the current state of the search, with coherency and exhaustivity.

```
------------  Justifications  ------------
| Problem  |  ------------->  |          |
|          |                  |   T M S  |
| Solver   |  <-------------  |          |
------------      Beliefs     ------------
```

The tasks of a TMS are ([deKleer 86a] for more detail):

1. Function as a cache for all the inferences ever made

2. Allow the Problem Solver to make non-monotonic justifications (i.e.: Unless there is evidence on the contrary, infer A).

3. Ensure that the database is contradiction free.

One of the first proposed TMS was the "Justifications-based TMS" by Doyle, explained in [Doyle 79]. This system is based on the enhancement of an "unique" solution, represented by a set of nodes, through the justifications asserted by the problem solver, after performing the tests. Nodes are IN if they belong to the solution, OUT otherwise. When a solution turns inconsistent, because of any of its IN nodes, *Dependency Directed Backtracking* is used:

Dependency Directed Backtracking may be defined as "adding justifications to remove contradiction". The idea is to "jump" directly to the level (in the search tree) where the discovered "bad" node was set, discard the selection, and obtain a coherent environment by adding new nodes and justifications. The usual, simple, alternative is *Chronological Backtracking*, which consists of performing a pure "depth first search" strategy (not useful for Doyle's TMS, of course), as used in PROLOG, for instance. Both styles will be ~~commented~~ referred to often in this thesis.

There were many problems with Doyle's TMS (see [deKleer 86a]), but the two most important were:

- *The single state problem*: Only one solution (IN) is explored, and only its justifications are found, even if other solutions are more suitable.
- Dependency Directed Backtracking, as used in TMS, is cumbersome and computationally expensive.

Many solutions were proposed in the 80s. The most successful has been the Assumption-Based Truth Maintenance System (ATMS) proposed in [deKleer 84] and formally explained in [deKleer 86a]. Its origins are the problems that de Kleer found while using Doyle's TMS in Qualitative Reasoning problems. I will explain its foundations as briefly and clear as possible, so that the rest of the thesis is understandable. I will start with the notation of its elements, as shown in [deKleer 86a]. More information can be found in that article, but [deKleer *et al* 87] and [Forbus 87] are the best tutorials I could find to learn ATMS, and they are strongly recommended to any beginner.

- *Node* corresponds to a problem-solver "datum". There is a special "nogood" (false) node, as well.
- *Justification*: a Horn clause of the form

  $$X_1 \wedge X_2 \wedge \cdots \Rightarrow n$$

  where X1, X2, ... are the *antecedent* nodes and n is the *consequent node.* A problem solver description of the justification is called "informant". n may be "nogood", which means that X1, X2, ... lead to a contradiction.
- *assumption*: A special kind of node that can only be antecedent (to say it exactly, it is justified only by itself).
- *environment*: A set (conjunction) of assumptions. Node n "holds" environment E if, according to the current set of justifications J, the following is true:

$$E, J \vdash n$$

in terms of propositional calculus. If n is "nogood", E is said to be *inconsistent*

- *context*: A set of assumptions plus all nodes derivable from them.
- *label* of a node n is any set of environments associated with n. The label is "minimal" if no environment in it is a superset of any other.
- A *Basic Data Structure* for a node contains a problem solver "datum", its label, and justifications related to it.

$$\gamma_{datum} = < datum, label, justifications > .$$

The differencies between this scheme and Peter Ross' ATMS will be clear in next section.

I will now end this short description by clarifying how information is propagated in ATMS. The justification shown above,

$$X_1 \wedge X_2 \wedge \cdots \Rightarrow n$$

automatically "implies" justifications involving antecedent nodes (and assumptions) of justifications of X1, X2, ..., and consequent nodes of justifications of n, in terms of propositional calculus (many implementations to do so are proposed. The easiest one involves going back to the environments). Therefore, we can be sure that the whole ATMS environment is propositionally consistent.

This way, the problem solver using ATMS avoids the two problems mentioned in Doyle's TMS

- There are many possible solutions to attempt, depending on the problem solver strategy, since ATMS may keep track of everything (real non-monotonicity).

- The propagation scheme is not cumbersome at all, it is reliable, it is efficient, and it is all the system has to do, since the search strategy may be left to the problem solver, and chosen depending on the problem (this will be discussed in next section, anyway).

Some final Comments:

- Deletion of assumptions and justifications is undesirable (as de Kleer says) and, in some implementations, like Peter Ross' one, impossible, so that the problem solver MUST include a way to cover these cases (e.g.: change - deletion - of constraints), if they are going to happen. I included this in my system.

- The choice of the search strategy is more flexible here. I will explain the one I chose, but other alternatives are proposed in chapter 6. In any case, some "extensions" to ATMS propose a backtracking mechanism embedded in it. This and other ideas will be commented in section 2.3. We now look at Peter Ross' ATMS implementation.

2.3 Peter Ross' ATMS

This section is extracted from Peter Ross' paper [Ross 87] about his implementation of ATMS, in order to make clear how it works. It is based on the description in [deKleer 86a].

"The paper describes what is essentially a record-keeping system to show how deductions depend on sets of initial assumptions, without presuming those assumptions to be either true or false. Only the justifications given to the ATMS, of the form

$$A_1 \wedge A_2 \wedge A_3 \wedge \cdots \Rightarrow C$$

need be true (the A_i are the *antecedents*, the C is the *consequent*).

The later of de Kleer's papers ([deKleer 86b] and [deKleer 86c], commented in next sections) complicate the overall picture by trying to build a proper formal logic into the ATMS. For the purposes of experiment and research, it seemed better to implement the simple system and leave the inferential details out of the ATMS, as de Kleer first advocated. The initial version has been coupled to NIP, a version of Edinburgh Prolog.

2.3.1 The basics

The vocabulary is basically that suggested by de Kleer:

node: the internal representation of a datum, that is, something that can figure as an antecedent or consequent.

assumption: a "foundation" node, so to speak. The truth or falsehood of all other nodes ultimately rests on the truth or falsehood of the assumptions.

justification: essentially, a Horn clause:

$$A_1 \wedge A_2 \wedge \cdots \Rightarrow C$$

showing how the truth of one node depends on a conjunction of others.

environment: a set of assumptions. The ATMS's job is to maintain records of all the consistent, minimal environments in which each node holds.

label: the sound and complete set of environments in which a node holds.

context: the set of all nodes which hold in a given environment. If the set includes the 'false' node, the context (and that environment) is called "inconsistent". According to first-order logic, such a context should logically contain all nodes; however, the ATMS cannot create justifications for itself.

2.3.2 The NIP interface

In the initial implementation, node identifiers must be integers. They can be arbitrary, except that zero is predeclared to be the 'false' node.

The following predicates are provided:

atms_setup(+Desired,–Granted)

This initialises the ATMS; all other predicates will fail with a warning message if this has not been done. Purely for reasons of laziness, you are required to give an upper bound on the number of assumptions you will create. The predicate returns the bound you have been granted; this is your desired number rounded up to the nearest multiple of the number of bits in a word on your machine (a constant which is calculated at system compilation time rather than being user-declared - so you don't need to know it).

atms_assumption(+NodeID)

This creates a new node, and specially marks it as being an assumption. Internally, a unique bit position is assigned for it in the bit sets that represent environments. The predicate fails, with a warning message, if that *NodeID* is already known to the system, or if you have already created the maximum number of assumptions granted by the initialisation routine.

atms_node(+NodeID)

This creates a new node, with an initially empty label. The predicate fails, with a warning message, if the *NodeID* is already known to the system. There is no built-in limit on the number of nodes you can create.

atms_justification(+ConsequentID,+ListOfAntecedentIDs)

This notifies the system of a justification (actually it need not be new) and precipitates an internal flurry of label adjustments. The predicate fails if any of the antecedent IDs is zero or is equal to the consequent ID. The latter would not actually upset the system, but it is presumed that the user might appreciate this defense against carelessness. The system can be told of a set of inconsistent nodes (and thus, implicitly or explicitly, of sets of inconsistent assumptions) by giving zero as the consequent ID. Looping justifications, such as two nodes implying each other or anything more complicated, are allowable and often useful. The internal label adjustment process is nevertheless guaranteed to terminate.

atms_see_node(+NodeID)

This prints out useful information about that node, for your own program debugging purposes. The predicate fails if the *NodeID* is unknown.

atms_env_data

This prints out useful information about all known environments. The inconsistent environments are called 'nogoods'; these form the label of the 'false' node.

atms_get_envs(+ListOfNodeIDs,–ListOfListsOfAssumptions)

Given a list of nodes, this returns a list of lists of assumptions representing all the environments in which the given nodes collectively hold. Thus, to determine the label of a node, give a list containing only that node as first argument. The predicate fails with a warning message if any of the specified nodes is unknown.

atms_get_context(+ListOfAssumptions,–Consistency,–ListOfNodeIDs)

Given a non-empty list of assumptions, this returns the atom 'consistent' or 'inconsistent' as appropriate, and a list of all the nodes that hold in that environment. This means that 'inconsistent' is returned if and only if the list contains the number zero, indicating that the 'false' node is one of those that appears to hold. The predicate fails with a warning message if any of the given IDs does not refer to a known assumption, or if the list is empty.

atms_debug(+Integer)

Debugging predicate.

As a point of use, notice that any assumption can be permanently invalidated by giving it, by itself, as a justification of the 'false' node. Also, if you want to find the minimal consistent subsets of a set of assumptions (this being dependent on the current set of justifications), just use *atms_get_context/3* to find the context and then use *atms_get_envs/2* to find the minimal consistent subsets.

If you want to find maximal contexts – that is, sets of nodes which are as large as possible without including anything inconsistent – then there is a simple procedure to follow. It is easily implemented in Prolog:

- Obtain the label of the false node by

  ```
  atms_get_envs(0, FalseNodeLabel)
  ```

  In general this will be a list of 'nogoods'.

- Obtain a list of all the assumption nodes in existence. Presumably, since the Prolog program created them all explicitly, it will have the necessary information already.

- Generate a maximal context by omitting one element of each 'nogood' from the list of all assumptions, and then find the context in which the remaining assumptions hold by using

  ```
  atms_get_context(PrunedSetOfAssumptions, _, MaxContext)
  ```

- Backtrack as necessary to find each way of deleting one element of each nogood from the full set of assumptions, to get other maximal contexts.

2.3.3 Limitations

You cannot retract a justification, although you can add an extra assumption node to each justification which you would take to mean " this justification is valid"; you can then track down which nodes depend on which justifications by looking for those special assumptions in the labels.

You cannot read back what the set of justifications is. You cannot read back what the set of tenable assumptions is. At the moment, your inference engine must do the latter two, although it would be very easy to add them".

2.4 Extensions to ATMS

This is a short section about suggested improvements to ATMS that could make projects like mine easier in the future.

As explained in last section, last paragraph, one of the main attempts to enhance ATMS has been to construct a backtracking mechanism inside, since many applications would take advantage of it, although the "raw" version oriented to qualitative reasoning did not require it. I was interested in this problem since such a mechanism could be included in the system I was designing. However, the different proposals ([deKleer & Williams 86], [Smith 88]) were rather different and none of them especially designed for my system.

A large set of interesting enhancements to the "raw" ATMS are proposed by de Kleer himself in [deKleer 86b], but they were not included in Peter Ross' ATMS, as he clearly explains, and were not very useful for my project, either. However, it is a very interesting article for people involved in ATMS on a long-term basis.

I have found some small articles about more "clever" enhancements, like "labelling algorithms" or "Massively Parallel ATMS" (~~that~~ which de Kleer found very difficult, in the beginning), and so on. Some of them are [Koff *et al* 88], [deKleer 88], [Forbus & deKleer 88], [Dixon & deKleer 88], but many other appear in AI proceedings, of recent years, since the interest is growing. They may be very interesting on a long-term basis; mainly, those related with "parallelism", in relation with "parallel logic programming", as will be suggested in chapter 6.

2.5 ATMS applications to Problem Solving

This is a subject where I would have wanted to find references more close to my problem, and of a more technical orientation, but I could not find them. The first article I found about it was [deKleer 86c], a very complete "guide" to the overall use of ATMS in problem-solving, but very few explanatory examples. Something similar can be said about [AIAI 87]. [Smithers 85] and [Smithers *et al* 89] are also good explanatory guides about AI in design and manufacturing, but no details about the potential role of ATMS in them is explained in detail.

The only article I found about a real application, whose main goal was the use of ATMS, is [Arlabosse *et al* 88]. Since Heriot-Watt University was involved in it, I tried to contact any person in that University who could know about the project, and I failed. Anyway, the project commented in the article was about Qualitative Reasoning (not very close to my project), and I believe that the authors were probably more interested in showing what they did rather than how they did it (probably oriented to more experienced people). Therefore, I could not take much advantage of it.

the lack of articles on applications made me try other sources of information (meetings, seminars, as said in the "Brief history"), that will be commented in next section.

2.6 Meetings and Seminars

I shall first describe three sources of knowledge which I used about scheduling systems:

1. Interview with Mark S. Fox (Director of the Intelligence Systems Laboratory, The Robotics Institute, Carnegie Mellon University)

   knowing that Mark S. Fox was giving some seminars about AI-scheduling research at CMU, in the department of Artificial Intelligence, University of Edinburgh, on 11th and 12th April 1989, and that he is one of the best researchers in this area (designer of ISIS system in 1980), I decided to attend those seminars ands ask him about references for my project. He sent me, in May, some brochures about technical publications and projects being developed in his Institute. Unfortunately they were not directly related with my project, but with scheduling. In addition I did not find a direct relation with the use of a TMS, therefore I could not take much advantage of them.

2. Project Planning Workshop

There was a course organised by the department of Meteorology, about MSc projects management, on Monday 8th May 1989, and Tuesday 9th May 1989, that I decided to attend, encouraged by Graeme Ritchie. I found it really more oriented to BIG projects involving many people than MSc projects, but it happened to be very useful for my work since it gave me practical knowledge about using scheduling techniques (critical paths, Gantt charts, PERT charts, and so on) that allowed me understand better the articles I found about scheduling and, indirectly, clarified my ideas about "resources" and constraints in my project.

3. Iain Buchanan's seminar about a Distributed Asynchronous Scheduler (DAS)

   I attended this seminar, on 18th May 1989, about a very advanced AI-scheduling system that is being constructed in the Turing Institute, Strathclyde University (Glasgow). No technical details were provided, but I was told how ATMS was used in it, and some advantages DAS had over ISIS and other similar systems in the 80s.

I will describe now two main contacts I had about ATMS:

1. Karl Millington, one of the most experienced researchers in EDS project, told me about the way ATMS is used in it. In a meeting on 10th May 1989, and a joint EDS and EdCAAD meeting on 30th May 1989, he provided me with information about it that I can resume in two points:

   - EDS is an architecture designed to support many kinds of design problems, not necessarily very well defined. Therefore, it is oriented to SUPPORT systems, where the user has to describe almost everything and solve the problem him/herself with the help of the system.
   - The assumptions are considered as reasoning justifications, more than "pure logic axioms", and justifications had to be interpreted like "rules" in an Expert System or "rules of thumb" in common sense, to justify nodes. It is the natural choice for EDS kind of problems.

   However, since the goal of my project was quite well-defined, and I did not require a "general purpose" architecture, I attempted a more "PURE

LOGIC" style in it. As a result, something more than a mere SUPPORT system was constructed, as will be shown.

2. Ken Currie, one of the heads of the Planning group in AIAI, gave me some articles about ATMS for problem solving on 17th May 1989, and even proposed that I should update Peter Ross' ATMS with newly proposed techniques oriented to use it in a "dependency directed backtracking" style, as will be explained later (either as part of the MSc thesis or as a PhD subject). Since those techniques were not necessary for my system (as I was using another scheme that will be explained later), and they lay far away from the project objectives set by Tim Smithers, I discarded that idea. Anyway, he gave me some of the articles commented above ([AIAI 87], [Smith 88]), useful for background reading at that moment.

Chapter 3

Detailed Description of the problem: Constraints

I would like to describe here an explanation of the problem, trying to make clear the kind of constraints existing in it, together with a small introduction to their handling in the system, that will be more clearly explained later in 'Hierarchical Plan' subsection, and related ones.

The final version has been designed to arrange lectures in a scheme similar to the one existing in the MSc in Information Technology - Knowledge Based Systems, whose characteristics can be described as follows.

- Lectures usually last for one hour, beginning at "o'clock" times (exceptions can be handled easily, anyway), same time every week. They can be arranged from Monday to Friday, from 9:00 until 17:00. Anyway, those restrictions are to be defined by the user, as facts of the form:
  - days([mon,tue,wed,thu,fri]).
  - hours([9,10,11,12,13,14,15,16]).
- Subjects for lectures may have, *either a defined number of lectures to be arranged* according to constraints and preferences (usually, 2 or 3), *or only fixed lectures.* This leads to two different kinds of constraints:
  1. subjlectures(Subject,Number).

2. fix($Subject$,[Day_1,$Hour_1$],$ConstraintNumber_1$).

   ...

   fix($Subject$,[Day_n,$Hour_n$],$ConstraintNumber_n$).

depending on the status *fixed vs. non-fixed* of *Subject. Number* is the number of lectures to be arranged for *Subject. ConstraintNumbers* (not used in Subjlectures) are constraint identifier numbers to be handled by ATMS, as in the other constraints that will appear. The status of a Subject may be changed, in both directions, during a session, as will be shown.

- The list of Subjects MUST be ordered as follows:

  - All fixed Subjects MUST appear first.
  - Other subjects MUST have any *subjlectures* predicate in the database, and they SHOULD be ordered MOST RESTRICTED FIRST.

    This is strongly recommended by DeKleer in [deKleer 86c] about applications of ATMS Problem Solving to Scheduling, and it is a common sense rule, anyway, in order to avoid backtracking, so that the *hierarchical plan* can work properly, as will be shown later

  Therefore, the final result should be:

  - subjects([$Subjectfix_1$,...,$Subjectfix_f$,$Subjectnonfix_1$, ... ,$Subjectnonfix_n$])

    Where subindex "1 ... n" means "more ... less" constrained subjects.

- There is a relation *Subject-Room* which implies that *Subject* lectures CAN be given at *Room.* The corresponding input form is:

  - lectroom($Subject$,$Room_1$,$ConstraintNumber_1$).

    ...

    lectroom($Subject$,$Room_n$,$ConstraintNumber_n$).

- Incompatibilities *Subject-Subject* : Two different subjects attended by the same student cannot have lectures at the same time. They cannot even follow each other if they are in distant rooms (such as Kings Buildings vs. Main University Area in Edinburgh). This leads to two different kinds of constraints:

1. nonsimult($Subject_1$,$Subject_2$,$ConstraintNumber$).
2. nonfollow($Subject_1$,$Subject_2$,$ConstraintNumber$).

where the order of subjects is not relevant.

- Incompatibilities *Subject-Time* : Some Subjects cannot be taught at some times, as a consequence of:

  1. Departmental decisions for all subjects (e.g.: No lectures at 13:00)
  2. Lecturer constraints (e.g.: Lectures for other courses at the same time)
  3. Rooms constraints (e.g.: atlt2, the only big enough room for some subjects is available only few times a week)
  4. Many other

  The first (1) and the other (2, 3, 4) are included respectively, as

  – notpos(all,[Day,Hour],ConstraintNumber)

  – notpos(Subject,[Day,Hour],ConstraintNumber)

- Preferences *Subject-Time*: Some *times* are particularly *bad* or *very bad* for lectures (although they can be used if no other choice), as a consequence of:

  1. Departmental decisions for all subjects (e.g.: Seminars on Wednesday afternoons)
  2. Lecturer preferences (e.g.: Mornings reserved for departmental projects)
  3. Many other

  They are included, respectively, as

  – bad(all,[Day,Hour]).

  – verybad(all,[Day,Hour]).

  – bad(Subject,[Day,Hour]).

- verybad(Subject,[Day,Hour]).

No constraint number is used. In fact, they show preferences rather than constraints.

Other kind of preferences are due to the *days between lectures for the same subject*. A regular distribution of lectures along the week is more desirable than cramping all them in the same day. As most subjects have two or three lectures, a distance of 3 days between days was considered excellent; 2 or 4 days, good; 1 day, bad; and 0 days (more than one lecture in the same day), very bad. I chose "3" in order to avoid penalizing tuesdays and thursdays (happens if 2 or 4 are chosen) and even wednesdays (happens if 4 is chosen), BUT it may be changed, because it is set by the user as an asserted input fact:

optdifdays(3).

All these preferences are used to compose AGENDAS of suitable times for a subject, ordered by priority. It will be explained later on.

- The maximum number of allowed assumptions is also entered by the user in the input file. I chose 4096, a quite big number, as the problem is very complex for a real timetable (HINT: $4096 = 2^{12}$).

  maxassumptnumber(4096).

- Some numbers are entered in the input, in order to establish the initial numbers for tree nodes, normal nodes, non-constraints assumptions and added constraint assumptions. so on:

  - firsttreenode(10000).
  - firstnode(20000).
  - firstas(30000).
  - consnumber(40000).

- and an aditional constraint number is entered in order to send ATMS inconsistencies due to lectures for different subjects at the same time, same room (first "consnumber": 40000):

nonsimultsamelecture(40000).

- One more predicate will show if we allow backtracking at previous levels ("backtrack") or not ("nobacktrack"). It will be explained in chapter 4.

There are two different kinds of "constraints" in the info shown above:

- Items that do not change in a session. They form what I call a "defaults" file.

- Items that may change during a session. The form the real constraints file.

Therefore, a defaults constraint file should have the following kind of information.

```
days([mon,tue,wed,thu,fri]).
hours([9,10,11,12,13,14,15,16,17]).
optdifdays(3).
maxassumptnumber(4096).
firsttreenode(10000).
firstnode(20000).
firstas(30000).
consnumber(40000).
nonsimultsamelecture(40000).
nobacktrack.
```

and an input constraint file should have the following kind of information.

```
subjects([...]).
```

```
subjlectures(Subject,Number).

fix(Subject,[Day,Hour],ConstraintNumber).

lectroom(Subject,Room,ConstraintNumber).

nonsimult(Subject1,Subject2,ConstraintNumber).

nonfollow(Subject1,Subject2,ConstraintNumber).

notpos(Subject,[Day,Hour],ConstraintNumber).

bad(Subject,[Day,Hour]).

verybad(Subject,[Day,Hour]).
```

The next Chapter shows a description of the Database system used in the system to handle the manipulation of these constraints in an ATMS environment in order to solve the problem, and other data structures used in the system to help the user.

Chapter 4

Objects and Structure of the problem: Data description

4.1 Formal Definition of Timetable and Lectures

I shall now clarify the main concept in this problem: *the Timetable.*

Definition: A *Timetable* is a set of *lectures.*

Therefore, I need to define lecture first. A lecture can be defined as an *object* whose *slots* should include, at least:

- Subject
- Time
- Room

But, since:

- Time is [Day,Hour] in our definition,
- lectures are represented in the ATMS database by nodes, and they need a number,
- PROLOG is not an Object Oriented Programming language, and it is more suitable for list processing,

the final chosen representation was a list as follows:

Lecture = [NodeNumber,Subject,Day,Hour,Room]

And, consequently, a Timetable is a list whose elements are lectures:

Timetable = $[Lecture_1, \ldots, Lecture_n]$.

The timetable is initially empty, until the problem solver fills it with *consistent* lectures until an initial solution is found (if possible) and further modifications with the *user options* update it. This involves a lot of searching, ATMS work, and alterations in the environment, whose tracks have to be kept in an adequate manner. Since the resolution of Geoffrey Marnell's problem, I had in mind to keep a detailed structure in the system, in the following aspects:

1. A *database* for assumptions, nodes and justifications, compatible with the above mentioned "Very Purist and Logically Consistent" scheme of justifications in ATMS environment, so that everything could be tested, and nothing could go wrong, even after many changes.

2. A *Proof Tree* showing how the problem was solved, every time the user requires *solve* or *resolve* options to find a solution automatically, so that a kind of *debug* was possible.

3. The *History* of the session: all changes, both in timetables and constraints, that the user made the program perform, so that keeping track of everything was possible for the user, so that he/she could recover a previous state, see how changes affected the timetable, and so on.

4. An *INFO* utility, using ATMS info, that shows what things fail at every moment, since changes in constraints and timetables are allowed after the initial solution is found.

Each of these will now be described separately:

4.2 ATMS Database and Justifications Scheme

In Peter Ross' ATMS, nodes and Assumptions identifiers are numbers, and no other semantic information about their meanings can be included. Therefore, a database containing that semantic information and relating it with the nodes and assumptions numbers in ATMS was required in PROLOG environment. I shall now describe this database:

4.2.1 Assumptions

There are two kind of used assumptions in the system:

1. Constraints: *fix, lectroom, nonsimult, nonfollow, notpos* , whose assumption number comes from the input constraint file. E.g.:

   - fix(spc,[tue,16],1010)

     means that "Constraint assumption 1010 supports the fact that a lecture for spc MUST be given on tue at 16".

     16.00 hours.

   - lectroom(kri2,atlt2,2002)

     means that "constraint assumption 2002 supports the fact that a lecture for kri2 MAY be given in atlt2".

   - nonsimult(spc,kri2,3003)

     means that "constraint assumption 3003 supports the fact that lectures for spc and kri2 CANNOT be given at the same time".

   NOTE: I will use these 3 constraints as part of the example in next sections.

2. "generate and test" assumptions: assumptions created while solving the problem, for non-fixed subjects. E.g.:

   - assumpt([30001,kri2,tue,16]).

     means that "assumption 10001 supports the fact that a lecture for kri2 MAY be given on tue at 16" (Clearly, non-monotonic reasoning, since it can be falsified later).

4.2.2 Nodes

The nodes information is kept in the database in a similar way as "generate and test" assumptions. There are two different kinds of nodes depending on their representations (that will be mentioned as "noderep"s from now on): they may include room information or not:

1. node([20010,spc,tue,16]).

   means that "node 20010 represents a lecture for spc (Software for Parallel Computers) given on tue at 16".

   Another node of the same style that will be used in next sections is:

   node([20001,kri2,tue,16]).

2. node([20002,kri2,tue,16,atlt2]).

   means that "node 20002 represents a lecture for kri2 being given on tue at 16, at atlt2 room"

   other nodes of the same style that will be used in next sections examples are:

   node([20011,spc,tue,16,kb]).

   node([20013,spc,thu,14,kb]).

4.2.3 Justifications Scheme

Let's see how nodes justifications are entered in ATMS:

1. The first one corresponds to a FIXED lecture (if we follow the examples in the subsections before). Therefore, the following justification is entered:

   $$1010 \Rightarrow 20010$$

   or, graphically:

```
      fix(spc,[tue,16],1010).

                 |
                 v

     node([20010,spc,tue,16]).
```

2. The second one includes a "room" information, therefore, it needs two steps:

   (a) Justification of the same node without room information, say node([20001,kri2,tue,16]).
   Since kri2 does not have fixed lectures, we use the "generate and test" assumption
   assumpt([30001,kri2,tue,16]).
   Therefore, the following justification is entered in ATMS:

   $$30001 \Rightarrow 20001$$

   (b) Justification of the node using (a) plus "rooms constraints" [1]. We look at the corresponding lectroom predicate "facts"
   lectroom(kri2,atlt2,2002).
   Therefore, the following justification is entered in ATMS:

   $$20001, 2002 \Rightarrow 20002$$

   More graphically, the scheme is as follows:

[1] In the creation of a new node with room information, the inconsistency with constraint "nonsharedrooms(40000)", and eventual nodes for "other lectures at the same time, same room", are sent to ATMS, so that the system does not have to worry later about it

```
      assumpt([30001,kri2,tue,16]).

                       |
                       v

        node([20001,kri2,tue,16]).    lectroom(kri2,atlt2,2002).

                                   |        |
                                   v        v

                     node([20002,kri2,tue,16,atlt2]).
```

If the problem solver discovers that there is also an existing node

node([20010,spc,tue,16])

in the timetable, and a constraint

nonsimult(spc,kri2,3003)

in the constraints input file, we have to establish the incompatibility (with node 20001, as the problem is independent of the room). Therefore, the following justification is set:

$$20010, 20001, 3003 \Rightarrow 0(nogoodnode)$$

✓

or

```
   node([20001,kri2,tue,16]).   nonsimult(spc,kri2,3003).

                              |     |  node([20010,spc,tue,16]).
                              |     | /
                              v     v/
                              nogood
```

"nonfollow" and "notpos" would be treated the same way. I will describe how the problem solver looks for such inconsistencies later.

HINT: *This justifications style is the heart of the system*

4.3 "Debugging Tree" utility

Now that the way assumptions and nodes are used is clear. I will describe the tree structure and a new category of nodes: the *tree-nodes.* A "toy description" of the problem solver, following the example before is included, in order to describe the tree structure.

Let's suppose that I only have to set two lectures for spc, fixed at [tue,16], and [thu,14], and one lecture for kri2 (non-fixed). Our toy problem solver would do the following:

enter [20011,spc,tue,16,kb] in the timetable. In addition to the operations described in the section before, this would also mean the creation of a *tree-node at level 1, say 10001*, and a new justification is entered:

$$20011 \Rightarrow 10001$$

Next time, the second fixed lecture for kri2 on [tue,16] is entered:

[20013,spc,thu,14,kb].

A tree-node is created at the second level, say 30002, and a new justification is entered:

$$10001, 20013 \Rightarrow 10002$$

Next time, an attempt to put a lecture for kri2 on [tue,16] is done:

[20001,kri2,tue,16].

A tree-node is created at the third level, say 30004. The problem solver discovers that there was a "nonsimult" conflict (see page before). Therefore, two justifications are set:

$$10002, 20001 \Rightarrow 10004$$

$$20010, 20001, 3003 \Rightarrow 0$$

Therefore, "backtracking" will produce a successful new time for it, say [20004,kri2,wed,16,atlt2].

A new tree-node, say 10005, is created, at the third level, and a justification is set:

$$10002, 20004 \Rightarrow 10005$$

And the problem is solved.

The solution timetable is

```
[[20011,spc,tue,16,kb],[20013,spc,thu,14,kb],[20004,kri2,wed,16,atlt2]]
```

"10005" is the SOLUTION NODE, and, from an ATMS point of view, it accumulates the Information about all the nodes existing in the timetable (20011, 20013, 20004).

The solution path from the "top" (level 0) is [30001,30002,30005].

The tree is as follows:

```
                          top
                           |
              30001: [20011,spc,tue,16,kb]
                           |
              30002: [20013,spc,thu,14,kb]
                          / \
30004: [20001,kri2,tue,16]   30005: [20004,kri2,wed,16,atlt2]
            |                                 |
FAIL: ''nonsimult" with 20010, 3003        SOLUTION
```

Since in a real timetable there are many levels with many more nodes, such a graphic representation is not possible, but the system would keep it in a list representation and show it in the following way.

```
[30000,top]
 [30001,[20011,spc,tue,16,kb]]
  [30002,[20013,spc,thu,14,kb]]
   [30004,[20001,kri2,tue,16]]
   fail: [20010,20001,3003]

   [30005,[20004,kri2,wed,16,atlt2]]
   SOLUTION!: [30001,30002,30005]
```

This tree is a powerful debugging tool, since it allows an expert user to see how and why things were done, and think about small changes, perhaps in constraints (to improve the solution), and even in the program.

A tree is produced and kept in the environment for the initial solution and for every time the user makes the system *resolve* the problem automatically, after some changes. Each *resolve subtree* uses its solution node as its identifier. There is always an active tree. The default one is the *solve* one (whose identifier is *tree*). Any *resolve* subtree can be activated, looking for its identifier (using *History* options, explained in next section), and using "puttree(Identifier)"

Some options to look at the tree are included, quite similar to the options offered in the OYSTER system, used by the Mathematical Reasoning Group:

`snapshot(Filename).`	`- to obtain a snapshot of the tree in Filename`
`display.`	`- to see the actual level and children`
`down(N).`	`- to move to the Nth child (N = Node number)`
`up.`	`- to move to the previous level`
`top.`	`- to move to the top level`
`solution.`	`- to move to the solution leaf`

4.4 "History of the session" utility

Usually, a session for solving a Timetable problem may be very long, since many changes can be introduced by the user until he/she is satisfied with the final result. Keeping track of the history of a session may be useful for several reasons:

- Debugging purposes.
- Watching and recovering previous states if something undesirable happens after some changes.
- Keeping a record of the exploration carried out.

Every change, either in the timetable or the constraints makes the system create a new "tree-node" which will act as the identifier of the change performed ("tree-node" name is used as an extension of the function performed for subtrees, explained before). Let's see the history after a change to the example in the section before:

Let's suppose that the user adds a constraint, say

notpos(kri2,[wed,16],4004).

then, the environment has changed, since lecture

[20004,kri2,wed,16,atlt2]

is no longer good in the timetable.

A "tree-node" identifier is created, say 30006, And a new fact (the change) has entered the history.

If he/she uses now the *resolve option*, a new node, say

[20006,kri2,fri,16,atlt2]

would enter the timetable replacing 20004. Then, a "tree-node", say

10007

would be created, and a justification entered (30007 is justified by the nodes in the timetable):

$$20011, 20013, 20006 \Rightarrow 10007$$

And a new fact has entered the history. The whole history is now:

```
[[solve,10005,tree,no,[]],
 [change,10006,[add,notpos,wed,16,4004],no,[4004,20004]],
 [resolve,10007,subtree,no,[]]
]
```

where "no" means that such options were possible ("yes" otherwise), and "[]" means that there were no failing constraints with the timetable at that moment, while [4004,20004] means that 4004 constraint was not satisfied because of the lecture 20004.

If I ask for the timetable, only the last one is shown, of course. In order to keep previous states, all historical timetables are kept in the Prolog database, as follows:

```
historyTimetable(change,10007,[[20011,spc,tue,16,kb],
                               [20013,spc,thu,14,kb],
                               [20006,kri2,fri,16,atlt2]]).
```

Some options are available to see the history:

- *history_info*: Will show the history of the session, from the first "solve" up to now. Solution nodes numbers after every change are shown. They can be used in order to see past timetables or consult resolve "subtrees" as shown in the section before

- *showtimetable(SolNode)*: Shows the past timetable corresponding to SolNode solution node. If no argument is entered, the last one is shown

- *snapshottimetables(Filename)*: Creates a snapshot of the history of the session, together with the history of the timetables

- *numbers_info(Number)*: Shows the "meaning" of an ATMS number (e.g.: 20011 means a lecture for spc given on tue at 16 in kb; 1010 is the number of the constraint fix(spc,[tue,16],1010); 10007 is a treenode, and so on)

4.5 Problem INFO utility

After a solution for the timetable problem is found, the resulting timetable will match all the constraints. But, constraints additions and deletions are allowed later, as well as moving lectures in the timetable. It is even possible that *no initial solution was found*, so that an approximate partially incorrect solution was set. In all these cases, the consistency of the timetable with constraints may be lost, as we saw in the previous section. Therefore, I included an INFO utility in order to let the user know what is corrupted after any change. It is also useful for the system, since the resolve option may be required later, and will use such information.

Provided that we have a very good TMS tool (ATMS keeps track of everything), a good way to find the things that fail in the system, could be simple: seeing which ATMS environments leading to 0 (nogood) affect nodes in the actual timetable (solution node). However, it is not enough, since some constraints may have been deleted, and ATMS do not allow the retraction of justifications or the deletion of assumptions. De Kleer says in [deKleer 86a], that such retractions and deletions should be a bad idea. In fact they are not necessary: I constructed a way of handling an INFO mechanism, avoiding retractions. I consider two kind of constraints:

- active constraints

- deleted constraints [2]

and the algorithm is:

1. find the failing constraints in the present situation (testing the solnode against active constraints with ATMS functions).

2. if there are none, we have finished; otherwise, go to 3.

3. find all environments leading to 0 (nogood node).

4. ignore those environments where "deleted constraints" appear.

5. select those environments where failing constraints appear.

6. find the contexts corresponding to any one of them

7. in each context, find the nodes that include room information which make the constraint fail and appear in the actual timetable.

8. select the lowest priority one (advise the user to remove it and take it as a candidate for removing, if "resolve" option required later).

Steps 1 ... 7 only require ATMS processing, and, 8 only needs a small database search. Therefore, the process is efficient, and it works.

The obtained info is shown after any change, and may also be obtained under user request with *user_info* command which will make a nice presentation of the unsatisfied constraint assumptions, nodes which make them fail, and *system advice* about which lectures to remove in order to solve the problem (if there is, in fact, any failure).

Next chapter will show how all this database information is used in the problem solver and every option that the system provides.

[2] No direct relation with IN or OUT nodes in a Justification based TMS. This is only a fast way of discarding old constraints in ATMS

Chapter 5

The System: Problem Solver and User Options

I think that the best way to attempt this chapter is to follow the way the system was initially designed: First, a description of the Initial Problem Solver (*solve* option). Second, the User options (adding, deleting constraints, moving lectures in the timetable, resolve the timetable, load and save constraints or environments, and so on). The human interface and some implementation details will also be covered in this chapter.

5.1 The Solve option: Hierarchical Plan

When I had to face the implementation of the initial problem solver, given some constraints, there were several points to cover:

- Solve the problem, whenever possible.
- If it was not possible, find the best partial solution
- Try to find an almost optimum solution according to preferences
- Avoid unnecessary backtracking
- Integrate it in the whole system, where further changes are possible

- Create the debug tree,

- Create the ATMS environment

And I had some choices about problem solving strategy:

1. A* or AO* heuristic search

2. Operations Research techniques

3. No search at all (or very few): Only support system

4. PLANNING techniques

5. Best-First-Search using an Agenda

and the decisions were:

1. had to be discarded since no suitable heuristic function was found. Anyway, these algorithms are not very likely for this problem.

2. had to be discarded: it is useful for related items, like scheduling, but not in this case.

3. had to be discarded, for the reasons shown in the introduction (first phase was necessary) and the discussion with Karl Millington about EDS.

What I did was a mixture of 4 and 5: I used a Hierarchical Plan, such that every level works in a "best-first-search using agenda" way. It does not mean that I am using a *planner*: Since the problem was clearly defined, I just designed a fixed Plan, that will be described in detail in the following subsections:

5.1.1 Hierarchy of Subjects, definition of levels

There is a hierarchy of subjects as has been described in chapter 3 (a sequence in which they are considered). Each lecture for a subject defines a different level, as seen in chapter 2 examples before. These levels are fixed or not, depending on the status of their corresponding subjects. For each level, attempts are made to set a lecture, either fixed or not.

5.1.2 Fixed Levels

At a fixed level there are only two choices: Either the time set in the "fix" constraint is good or not, according to constraints (the trees at this levels are "one father, one son"). In both cases, the lecture is entered, but if it is not good, error is reported, both to the ATMS and the user, so that he/she may change it later.

5.1.3 Non-fixed Levels: Best-First-Search

in Non-fixed Levels, a *Best-First-Search* strategy is used, where the *BEST* depends on *preferences* plus "notpos" constraints (useful in order to discard unsuitable times). An ordered *agenda* is used to do so.

An agenda, as described in search algorithms is a list where next step choices are kept, so that we attempt them until, eventually, one of the choices leads to a successful end, or there are no more choices in the agenda.

In this *plan*, the "successful end" at each level, is to set one lecture that keeps the constraints satisfied.

an agenda, at any non-fixed level, in this system is a list as follows

$[Pri_1/[Day_1,Hour_1]], \ldots,[Pri_n/[Day_n,Hour_n]]]$

Where Pri_j is the Priority that $[Day_j,Hour_j]$ has according to preferences.

A low number means Higher priority in this system, since it allows the use of *setof* automatic increasing ordering. Therefore, lower "Pri" numbers correspond to the most suitable times and are set first in the Agenda.

The best-first-search at each level is very simple: Once the agenda is created, *Chronological backtracking* is applied according to that agenda, until a "successful end" is reached. This way, we ~~guarantee that~~ are guaranteed to choose the "best" *time* that matched the constraints, and is compatible with other lectures at previous levels~~, is chosen~~.

Some questions have to be answered, however:

- How are the agendas created. Which are the criteria?

- How do we check that a suitable time is consistent with the constraints?
- How do we arrange rooms information, not included in the agenda?
- What happens if no time in the agenda is valid?

These questions will be answered in next sections.

5.1.4 Agendas

In this subsection I will describe the *Agendas Production System.* As explained in the introductory chapter, it is based on the preferences found in the constraints file (plus the "notpos" constraints, that, directly, discard a time for a lecture), plus the difference in days between lectures for the same subjects.

For every non-fixed subject, where m lectures will be set there is a "first" level. At this level, every time gets its priority value:

"notpos" affected times get "11" (a number greater than 10 will discard them)

"verybad" affected times get "2"

"bad" affected times get "1"

other (good) times get "0"

therefore, "good" times are set first, then "bad", then "verybad", then "notpos". This is the *primitive agenda.*

At each other level, for each remaining time in the agenda, the minimum difference in days with the other lectures set for the same subject, is calculated, and *added* to the corresponding priorities in the *primitive agenda.* The resulting agenda is used at that level. It will be made clear in an example, in 5.1.8.

5.1.5 Testing Mechanism

For every selected time, we know that "notpos" constraints cannot affect it, since it was tested while constructing the agenda. Therefore, "nonsimult" and "nonfollow" constraints must be tested, as well as rooms possible problems. The latter will be explained in the following subsection.

In order to test "nonsimult" and "nonfollow" constraints, we have to perform some steps:

- looking in the timetable for the lectures set at the same time ("Then") and adjacent ones ("Sides").
- for each lecture in "Then", extract the subject, and test if a "nonsimult" constraint affects both that subject and the one we try to introduce. In that case, discard the "Time". Otherwise:
- for each lecture in "Sides", extract the subject, and test if a "nonfollow" constraint affects both that subject and the one we try to introduce. In that case, discard the "Time". Otherwise: test rooms problems.
- At the same time, send all the information to ATMS.

5.1.6 Rooms Rearrangement

The first thing the system has to do in order to find a room for the lecture is looking at the available rooms ("lectroom(Subject,Room,_)"). If there is one available room that is not busy at the same time ("Then" is tested again), it is chosen, and the lecture is definitively set. Otherwise, a method can be used in order to save effort, that will be explained in the following paragraphs:

The Hierarchical approach to the problem means that no lectures are set at any level, before the previous level has been completely set. Therefore, modifications to previously set lectures, in order to avoid inconsistencies are not allowed. But *rooms* are an *exception*:

According to "timetable experts" like Graeme Ritchie, Rooms are a "minor problem" in lectures arrangement (although they should not be in exams arrangement). Therefore, if the change of room for a previous lecture can avoid backtracking, it is done (even altering the tree structure, as alterations are very small).

The idea is try to rearrange the rooms for lectures in "Then", until one of the rooms available for the new subject stays free (It may or may not succeed, but such an attempt is reasonable). It will be clear after the example shown in 5.1.8.

5.1.7 Failures: Backtracking at upper levels

One question arises when it is not possible to set a lecture at a non-fixed level (which means that the plan fails). What do we do then? There are two natural choices:

- Ask the user what time does he/she want the lecture to be (arbitrarily) and assert it, sending all information about problems that it will produce, to ATMS.

- perform backtracking at upper levels (what I call *undesirable backtracking*) until one branch leads to a solution.

This system allows both of them, which are selected in the defaults input file, as shown before, including either "nobacktrack" or "backtrack".

In my opinion, the first option is the natural choice, since most times we do not know *a priori* if there is really a solution, and, anyway, from the point of view of the user and the application, it is better to solve the problem as soon as possible, even with faults, see what happens and modify it.

The second option is the "purist" one, of course, and it uses chronological backtracking, as well (I will explain in 5.1.9 why Dependency Directed Backtracking is avoided). But it is not the most practical one and should be used only if we know, *a priori* that there is at least one solution, and we are not in a hurry: I made a test altering the real MSc timetable constraints so that the *second* non-fixed subject had no suitable times. More than 50 undesirable backtrackings were needed until the system finished exploration and realised that there was no solution. If this problem happens at a lower level, there would be thousands of backtrackings. Following the "purist" style, this option stops in those situations, showing the longest partial solution found and telling the user to exit and modify the constraints.

The example in the next subsection shows a backtracking option, in order to allow the reader to understand the whole process.

5.1.8 Example

It is the moment to show a more complex example, to see all those things together. Some abbreviatures are used for subjects (*databas* means "database systems", *matreas* means "mathematical reasoning" and *nlg* means "natural language processing") and rooms (*a1* and *a2* are arbitrary names). A "_" symbol is used instead of constraint numbers, as they will be irrelevant in the example (I omit a description of how nodes are produced and justified by assumptions, to simplify the scheme). The constraints are shown below:

```
days([mon,tue,wed,thu,fri]).

hours([10]).

...

subjects([databas,matreas,nlg]).

fix(databas,[mon,10],_).

lectroom(databas,a1,_).

lectroom(databas,a2,_).

subjlectures(matreas,2).

lectroom(matreas,a1,_).

subjlectures(nlg,1).

lectroom(nlg,a1,_).

nonsimult(databas,nlg,_).

notpos(nlg,[tue,10],_).
notpos(nlg,[wed,10],_).

notpos(nlg,[fri,10],_).
```

Plus usual additional information. Then the hierarchical plan would react as follows:

- Level 1: fixed lecture is set

  timetable = [[21002,databas,mon,10,a1]]

  treenode = 11001

  $$21002 \Rightarrow 11001$$

- Level 2: no preferences:

  Agenda = [0/[mon,10],0/[tue,10],0/[wed,10],0/[thu,10],0/[fri,10]]

  [mon,10] is picked up. The only room allowed (a1) creates conflicts with "databas", Therefore the previous lecture is changed, as explained before

  timetable = [[21003,databas,mon,10,a2], [21005,matreas,mon,10,a1]],

  treenode = 11002

  the tree structure is partially broken (31002 cannot inherit from 31001, since a previous node has changed). Therefore, the justification is:

  $$21003, 21005 \Rightarrow 11002$$

- Level 3: Optimum difference between days = 3:

  New Agenda = [0/[thu,10],1/[wed,10],1/[fri,10],2/[tue,10]]

  [thu,10] is picked up

  timetable = [[21003,databas,mon,10,a2], [21005,matreas,mon,10,a1], [21007,matreas,thu,10,a1]]

  treenode = 11003

  $$11002, 21007 \Rightarrow 11003$$

- Level 4: lecture affected by "notpos"

  Agenda = [0/[mon,10],0/[thu,10],11/[tue,10],11/[wed,10],11/[fri,10]]

  (11 (greater than 10) means not possible)

  [mon,10] picked up. Not possible because of "nonsimult(databas,nlg,_). Therefore we perform backtracking at the same level (*not-undesirable backtracking*).

  [thu,10] picked up. Rooms arrangement with existing lecture for matreas is not possible.

  Other times are not suitable. Therefore, we can perform backtracking at levels before (*undesirable backtracking*), if we want to find a complete solution.

- Level 3: [thu,10] discarded.

  New Agenda = [1/[wed,10],1/[fri,10],2/[tue,10]]

  [wed,10] is picked up

  timetable = [[21003,databas,mon,10,a2], [21005,matreas,mon,10,a1],

  [21017,matreas,wed,10,a1],

  treenode = 11007

  $$31002, 21017 \Rightarrow 31007$$

- Level 4: retried: Same Agenda

  Agenda = [0/[mon,10],0/[thu,10],11/[tue,10],11/[wed,10],11/[fri,10]]

  [mon,10] is picked up, and discarded again

  [thu,10] is picked up. Now there is no problem:

  timetable = [[21003,databas,mon,10,a2], [21005,matreas,mon,10,a1],

  [21017,matreas,wed,10,a1], [21027,nlg,thu,10,a1]]

  treenode = 11008

  $$11007, 21027 \Rightarrow 11008$$

  solution node: 11008

5.1.9 Comments

This example illustrates several important principles:

1. The fact that *matreas* is higher in hierarchy than *nlg* (which is more constrained) produces undesirable backtracking and creates problems. "Most constrained first" rule should be applied.

2. The possibility of changing previous rooms, even altering a little the tree structure, is very useful. However, is not easily extensible to other cases (as a kind of "dependency directed backtracking") for several reasons:

   - It may be completely useless: We could try to modify "databas" lecture, (level 1), when we discover, at level 4, that nlg does not have a place, but [mon,10] would be good, if level 1 was altered. However it would not work, since databas is fixed.

   - Even if it was useful (using a complicated algorithm) we would be altering the hierarchy and preference rules, so that the solution would not be optimal.

   - Even if the solution was reasonably good, the tree structure would be totally spoiled and no further debugging would be possible.

   - Even if we keep track of all those "dirty tricks", such an algorithm would be very difficult (room management was not easy at all, and it was a very small problem), and probably not-complete (it would not solve all possible cases).

   - De Kleer shows in his paper [deKleer 86c] the convenience to use an ATMS with chronological backtracking (PROLOG style), instead of dependency directed backtracking (after all, this is not a justifications TMS, and there are not really IN or OUT nodes, although a similarity between "nodes in the timetable" and IN nodes exist).

   Anyway, the task of finding a *complete* and *efficient* dependency directed backtracking mechanism would be very useful, and it is left as an open task to be commented in chapter 6.

3. backtrack-nobacktrack

   "nobacktrack" option would have asked the user where to set "nlg" lecture, in the fourth level. Let's suppose the user chooses the same time ([thu,10]). The system would tell the user the list of incompatibilities (with lectures *matreas* at [thu,10]) and the user could change this one by him/herself, with the same final result.

   It is true that in this case, "backtrack" has proved to be a little better, but, what happens if there are 9 levels between *matreas* and *nlg*? Chronological backtracking would perform thousands of unnecessary operations, while dependency directed backtracking would have the problems shown above. I believe that "nobacktrack" is more convenient in such a case.

4. Database problems

   Database search is widely used in this hierarchical plan. Consider the previous example:

   - At level 1, "fix" predicate facts are searched in the Prolog database to see if any of them affects databas.
   - At all levels, "lectroom" predicate facts are searched as well
   - At level 2 and 4, "bad", "verybad" and "notpos" are tested to form the agenda.
   - At all levels, "nonsimult" and "nonfollow" are tested to see if a time is suitable for a lecture.
   - Every time a new node is created we have to:
     - check that it did not exist before (level 4 on [thu,10] is the same node both times)
     - check if an assumption exists that can justify it

   Otherwise, we would produce a proliferation of nodes and assumptions that would make future database searching more difficult, and, *worst of all, we could not be able to use the ATMS properly.* Let's see why: If we have 3 different

   "assumpt([N,matreas,thu,10])" (or "node([N,matreas,thu,10])") where "different" means "having different Ns" (e.g.: 30001, 30002, 30003) and we discover inconsistencies with other nodes or constraints, *which one of the three do we use to send the ATMS as the fault? All them?* (then we are doing extra work instead of saving it). Otherwise, it is impossible to give sound information to the user.

Uniqueness is necessary and so is database search in this case.

Now that we are aware of the necessity of much database searching, improving the default handling of databases search by Prolog could be a good idea, but:

- A design of an ideal database to handle it may be a problem as hard as the whole system itself.
- Prolog is not a very suitable language for designing kinds of database system other than the normal one.
- The program would be less readable if an *Ad Hoc* database system is used.

A good solution avoiding these problems could be to use *Parallel Logic Programming.* It will be commented in chapter 6.

It is now time to comment all other options. The main principles I applied here were:

- All kinds of constraints could be added and removed
- Subjects could be entered and removed
- Subjects could change their status (fixed - non-fixed)
- Lectures in the timetable could be moved, in time and/or room
- Options were reasonably "orthogonal" in the sense that every (reasonable) change the user wants to do is possible, even if it involves many small changes. It does not mean that all those possible complex changes are available in only one step.

- These options together with “resolve” can be done in any order, as many times as required, without corrupting the ATMS or other information in the system: It must be always reliable.

- For any change that produces problems, all possible inconsistencies are sent to the ATMS, not only the first found one (necessary if we want “resolve” option to work properly).

- If the user wants, he/she do not have to create an input constraints file, but enter all constraints from scratch before “solve” option.

In the following sections we shall see how these principles are implemented by the different options provided in the system.

5.2 Addition of constraints

I will describe in this section the information that can be added to the database system, and how it affects the environment. The following subsections cover each possible addition.

5.2.1 Subject

This option allows a new subject to enter the system. In this case, the user must enter also its position in the hierarchy (if it is not the first one), and the rooms available for it (at least one). If addition is done before any attempt to “solve” the problem, the user interprets it as pure “data entry”, and the user must tell the system if lectures are fixed or not, and, depending on it, either what times are fixed for it, or how many lectures it will have, depending on the answer, but they are not entered in the timetable (“solve” will do). If “solve” option was used before, the new subject is supposed to be non-fixed with 0 lectures, so that further additions of “fix” or “subjlectures” (shown below), will set its right lectures.

5.2.2 Lectroom

A new room for a subject may enter the system if it was not there. It does not affect the timetable or the ATMS environment.

5.2.3 Nonsimult

A new constraint "nonsimult(Subject1,Subject2,N)" may enter the system, if it was not there. In that case, all possible problems caused by this addition are found and reported to ATMS.

5.2.4 Nonfollow

As previous item.

5.2.5 Notpos

A new constraint "notpos(Subject,[Day,Hour],N)" may enter the system, if it was not there. In this case, any eventual "bad" or "verybad" "constraint" affecting *Subject* and *[Day,Hour]* is removed, and, if there is any lecture for *Subject* on *[Day,Hour]*, the problem is reported to ATMS.

5.2.6 Bad

A new "constraint" (preferences information) "bad(Subject,[Day,Hour])" may enter the system if it was not there. Any eventual "verybad" or "notpos" constraint affecting *Subject* and *[Day,Hour]* is removed (in the case of "notpos", it would involve "deleting" a constraint, and, consequently, setting its constraint number as "deleted").

5.2.7 Verybad

As previous item.

5.2.8 fix

The addition of a constraint "fix(Subject,[Day,Hour],N) is forbidden if no "solve" option has been performed, as fixed lectures have been already entered while introducing the new *Subject.*

Otherwise, it is allowed, and it involves:

- The removal of all lectures for that subject in the timetable.

- The deletion of all existing "fix" constraints for it, if it was fixed.

- The deletion of "subjlectures(Subject,N)", otherwise, and, consequently, a change of status: $non-fixed \rightarrow fixed$.
  non-fixed

- A new position in the hierarchy, which the user must decide.

And all new lectures are entered. The reader may find it more natural adding and deleting fixed lectures one by one, but I did it this way for two reasons

- This way, it can be used to change the status of a lecture in only one step.

- A change of a fixed lecture is a serious problem (in fact is is very uncommon in real timetables), since it affects changes, not only in the timetable, but in the *constraints*, situation that "move" option cannot handle (only changes of room are allowed for fixed lectures with "move" option, since they do not involve change of constraints), and this is the best way of managing this case in only one step, and making it clear.

5.2.9 Subjlectures

The addition of a constraint "subjlectures(Subject,N)" is also forbidden, if no "solve" option has been used, as the number of lectures has already been set while entering the new subject.

Otherwise, it is allowed, and it involves:

- If Subject was not fixed (so that a "subjlectures(Subject,OldN)" is still in the system), there are three possible cases:

  1. If N is greater than OldN, it means the addition of N-OldN lectures, done *automatically*. This is affected by the default option "backtrack"/"nobacktrack", explained before.
  2. If N is less than OldN, it means the deletion of the OldN-N lectures chosen by the user.
  3. If N is OldN, nothing is done.

- If Subject was fixed, it involves the removal of all lectures for that subject in the timetable, and the deletion of all "fix" constraints for it. A new position in the hierarchy is decided by the user, and N lectures are set as explained in the previous item, case 1. This means, of course, a change of status: $fixed \rightarrow non-fixed$.

non-fixed (Use \! in TeX mathmode to move symbols closer together)

5.3 Deletion of constraints

I will describe in this section the information that can be removed from the database system, and how it affects the environment. The following subsections cover each possible deletion.

5.3.1 Subject

This option allows a *Subject* to be removed from the system. It involves:

- The elimination of *Subject* from the list of subjects
- The removal of all lectures for *Subject* in the timetable
- The deletion of all "constraints" and "preferences" (lectroom, nonsimult, nonfollow, notpos, bad, verybad, fix, subjlectures) which include that *Subject* as an argument.

5.3.2 Lectroom

A new room for a Subject may abandon the system only if it is not being used for that *Subject* in the timetable. It does not affect the timetable or the ATMS environment.

5.3.3 Nonsimult

A new constraint "nonsimult(Subject1,Subject2,N)" can always be deleted, if it exists. Its number N is set as "deleted". It may produce eventual changes in ATMS environment.

5.3.4 Nonfollow

As previous item.

5.3.5 Notpos

As previous item.

5.3.6 Bad

As previous item, but no constraint number exists, and it never produces any change in the ATMS environment.

5.3.7 Verybad

As previous item.

5.3.8 Fix and Subjlectures

"fix" and "subjlectures" cannot be deleted, as explained earlier, except by adding "subjlectures" or "fix", respectively.

As can be seen, this option is very simple. Its only complications will arise in "ATMS INFO" option, as a consequence of the changes in the sets of "active" and "deleted" constraints, as shown in chapter 4.

5.4 Movement of lectures in the timetable

As said before, fixed lectures can only change in room, but not in time. All other lectures can change also in time. When such a change is made, the following is done:

1. Look for the lectures in the timetable corresponding to the destination time (Then), and adjacent lectures (Sides)

2. If there is a lecture in "Then" for the same subject, the change is not done (This is the only forbidden case).

3. If there is a problem like the following:

   - Conflict of the new lecture with "notpos"
   - Conflict of the new lecture with "nonsimult" and any other lecture in "Then"
   - Conflict of the new lecture with "nonfollow" and any other lecture in "Sides"

   then, the system looks for all other possible problems (many of them can happen at the same time), and sends inconsistencies to ATMS, so that the *Info* will tell us all problems.

Why is all this checking done? Let's come back to the example in 5.1.8., with one more subject (*assemb*), and the following additional constraints:

- notpos(assemb,[mon,10],_).

- nonsimult(databas,assemb,_).

If we try to move one lecture for assemb to [mon,10], the system would realise first the inconsistency with

"notpos(assemb,[mon,10],_)".

Let's suppose that the system does not look for other inconsistencies, although, as can be seen, there is another one, say, *I2*, with

"[21003,databas,mon,10,a2]", and

"nonsimult(databas,assemb,_)".

If the user decides later to remove the constraint

"notpos(assemb,[mon,10],_)",

then the system will tell him/her that the asserted lecture for "assemb" is O.K. (an eventual "remove" option would not even affect it). *However*, it is still inconsistent because of *I2*, but ATMS was not told so.

Such a bad situation does not happen in this system. I hope that the effort in looking for all inconsistencies in order to make the system reliable is now clear.

5.5 The Resolve option

The resolve option has been designed to lead the timetable to a solution that follows the constraints, again, after some changes have been performed. The main requirement for the user at the moment it is needed, is usually *to do it as soon as possible*. Otherwise, he/she'd better save the constraints and "solve" again from scratch (say, option *O2*).

O2 would probably lead to something very close to the "best solution", if the hierarchy has been established well, as it would follow the plan. On the other hand, it is slow if many lectures have to be set in the timetable. It would be similar to "destroying our house and building it again, just because our washbasin was broken", which is not very clever.

What the system does, instead, is

1. look at the list of candidates for removing (see 4.5, step 8 of the algorithm), except for eventual fixed lectures.

2. remove them from the timetable

3. fill the timetable with, as many lectures as we have removed, for the same subjects, in a strategy similar to the "solve" one.

This process does not follow strictly the hierarchy (old lectures of lower levels that are not candidates for removal, will stay in their places, even if "higher level" lectures could be set in their place), but I do not consider this to be important. On the other hand, it is reasonably fast, which is what we want.

As we are following step 3 (in section 4.5) before the "solve" strategy, the default choice "backtrack"-"nobacktrack" will affect us. In the first case, the problem may be left unsolved and unfinished, and we may be told to attempt option *O2*. In the second case, we may be told to choose any place for every lecture that has not a suitable "hole" in the timetable. In these cases where the problem remains unsolved, the user will have to help the system to solve the problem himself, looking at constraints, other bothering lectures in the timetable, and so on.

I consider "resolve" utility absolutely necessary, in order to avoid the user having the problems shown in the previous paragraph whenever he/she makes a change. This way, he will seldom have such a hard task.

5.6 Loading and Saving information

In my opinion, those options are fundamental, since it is not common to establish a definitive timetable in only one session: constraints change from day to day, and new updates may be necessary. This implies the need for *saving/loading a whole session* (ATMS environment, timetables, trees, history, etc).

I found also the need to *save/load only the constraints*. First, when an unexperienced user faces a problem like this, he/she is probably more interested in creating an input file of constraints to be loaded by the system than entering the subjects and constraints one by one. Second, after an unsuccessful *resolve* attempt, or a solution that does not satisfy us after many changes, we may want to save only the constraints, and try *solve* again from scratch, forgetting the whole environment.

In both cases, I made the predicates for loading and saving, so that the resulting files were *Prolog programs (actually, sets of predicate facts)*, for two reasons:

- It makes the *loading process* easy, since Prolog programs can be easily reconsulted in a Prolog environment (of course!) like the one I am using.

- It makes the file more *understandable* and even *modifiable* by the user, so that he can even create the initial constraints file from scratch, as shown above.

The task was not very difficult, but many predicates were needed in order to format all the different predicate facts and write them in a Prolog file style. I had to take some care, as well, in order to make the system reliable, in the sense that after saving, abandoning the session, starting again and loading the session before (environment), the state is exactly the same.

The use of *load* option is possible only in an "empty environment". More exactly, an environment where *solve* has not been used yet. It is done in order to maintain the reliability of the system: the actual constraints *must* be according to the actual environment (ATMS, history, trees, etc), and both of them *must* be according to the assumptions, nodes and justifications numeration. I think this was the best way to achieve this goal.

As can be seen, loading and saving an environment is a little slow, as a consequence of the big amount of information that has to be put, but it is just as fast or slow as loading or saving any Prolog program of the same size. Any improvement at this point should be done on the implementation of Prolog, rather than on the system.

5.7 Human-Computer Interaction: The Graphic Interface

I would like to begin this section describing how the system is loaded, and how to start a session, since it will make clear why such a graphic interface was chosen. It will also allow any potential user to get started with the system.

The final version of the system is designed to work on a SUN-3 machine, under "suntools" environment, so that the first UNIX (SUN) command the user should use, once we are in the right directory, should be:

suntools [-i] (-i for black screen: to avoid premature blindness!)

Appendix F shows the content of the ".suntools" file that should be in the *home* directory in order to use the system (also designed for small bold characters). In this case, some windows appear. The most important of them, is a large *cmdtool* window that will be one of our working windows. We use this window to write a new command:

```
prolog -U200 -L1024 -C1024
```

where -U200 is due to Peter Ross' program "prolog.ini", needed for ATMS system, and -L1024 -C1024 have proved to be convenient for working with programs wider than 100 KBytes, as the one in this system. Then, the user should write (now inside PROLOG):

```
[-'start.pl'].
```

(I decided not to include it in "prolog.ini" file, since it would have meant modifying Peter Ross' file).

While loading start.pl, the main "Help" info will appear on the screen, while the other programs are loaded, out of sight fromthe user.

When everything is loaded, and that window has disappeared, the user should start by typing

```
template.
```

And a graphic Main window with a menu of options appears to the right. This window and menu, are consequences of the *graphic interface* which I used, as will be described just now.

The interface in this system is based on Richard Tobin/Peter Ross "Simple NIP-Suntools Interface", ~~that~~ to which Del Cornali, an MSc colleague, ~~gently explained~~ gave me a gentle introduction. to ~~me how to~~ use. It does not contain many options, but it was quite nice and easy to use. The main options I used were the graphic windows (with the possibility of copying images to them) and the menus, to be set inside windows, in order to select the desired system options.

At different moments, depending on the options chosen different menus appear, in order to select the day, hour, room, subject, yes/no, and many other options that the user is given. The system gets back to the menu before (or the main menu) in case of bad entries (or pressing the mouse out of the menu, for instance). It serves to the reliability of the system and to facilitate the user a way to "escape" from options wrongly selected that he /she does not want to continue.

The fact that the "Simple NIP-Suntools Interface" does not allow the use of text windows for editing, made necessary the use of the above mentioned "cmdtool", as the other main window in the system, used for most of the output information from the system, and for input, when more that a simple "option selection" is needed (e.g.: file names, new subjects or rooms added to the system, etc). The use of "cmdtool" instead of "shelltool" will be explained below.

One of the most important needs of a system like this is a nice presentation of the timetable, which allows the user to see how lectures are set, just by a simple sight. A whole section of the program is dedicated to that task. It had some key requirements:

- To be a *text* presentation, instead of graphic, because a "snapshot" of the timetable in a file would be required.

- To give, at least, a small concession to graphics, to make possible the simple sight.

- To keep as much information as possible in few space: Use of abbreviations for subjects and rooms, and putting days and hours in a purpouse-built grid, designed for weeks of five working-days, as usual.

Even so, the size of the timetable may be big, as can be seen in appendix I, where some "layouts" of a session are shown. In addition, the *info* option shown together with the timetable after any *solve*, *resolve* or *change*, may be more than a page in length if many constraints happen to fail, so that it may "hide" the timetable. Therefore, a scrolling option was needed, which "cmdtool" provides, and "shelltool" does not.

The eventual use of editors for, say, new created constraints files, may make necessary the use of a "shelltool" window, therefore, there is a big one just under the "cmdtool" one. All the user has to do is hide the "cmdtool" window, and the other will appear. If more things are needed, the user still has all other suntool and sunview options. If any of this things is to be done, it is better to *exit* the system (but not Prolog) for a while, because, otherwise, it is waiting for a response in the menu, and it will not allow the user to do anything. Once the user has finished and wants to recover the environment, all he has to do is use the command:

```
mainmenu.
```

and he/she may continue the session (no save/load needed at all).

More information about the "Simple NIP-Suntools Interface", may be obtained from Richard Tobin, of A.I.A.I. and Peter Ross, od D.A.I., as well as all NIP and sunview manuals in the department.

5.8 Implementation Comments

Up to now, I have covered mainly the theory behind the program, the options I considered useful (and why), and how the user would use them in order to take advantage of the system. I have intentionally left all implementation details apart (e.g.: use of assertions and retractions in Prolog, the meaning of predicates facts, some of them used as "variables" or "database records", goodness of algorithms, etc). My main interest has been to describe the system as simply and clearly as possible, avoiding details about programming, even if, after this, it is not so clear how more than *190 KBytes* were necessary to implement it. I believe this is the best way to describe my system, and I will not describe implementation details here, either.

However, there are several appendices with the complete listings of the programs. The predicates are thoroughly commented and adequately ordered so that reading it may allow another person to follow, understand, and hopefully improve it.

Chapter 6

Further Research

I believe that this project may encourage future students and other people interested in the subject, to study it, find potential improvements, and try to implement them. I have found some areas in which it might be clearly improved, and I would have been very keen on attempting such improvements myself if I had more time.

Some of the potential improvements have been mentioned, or, at least, their need have been clear after the description in this thesis. They will be shown in the following sections:

6.1 Database system: Parallel Logic Programming

As shown in 5.1.9., I suggest the use of parallel logic programming in a problem like this: Since most database problems in this system are about searching for the "fact" that matches an expression: e.g.:

"node(20001,X,Y,Z,T)."

and most times only one does, the ideal software would be a parallel logic programming language with a kind of OR parallelism: either *Concurrent Prolog* style, or a language with committed choices (since most times, only one "fact" matches the expression), like *Parlog.*

The ideal hardware device would be a multiprocessor system with shared memory. In this case, using OR Parallelism, the speed of search would be multiplied by the number of processors, since no communications conflicts may arise.

In the moment of writing this thesis, there is no such ideal combination Software-Hardware. Anyway, as soon as I knew that a Sequent machine was available for use in the department of Artificial Intelligence, and Edinburgh Prolog, a version of C (needed for Peter Ross' ATMS), and even a subset of Parlog were suitable for it, I asked for an account in order to make some tests myself, even if it was not the ideal device, in order to find some potential improvements.

However, the use of Sequent machine seems to be at an early stage, and I had a large number of problems with it. This, together with the lack of time forced me to abandon the idea. I leave it as an open field for further research in this area.

6.2 A general Timetables Architecture

As shown before, many kinds of timetables may exist, and not only "tutorials", "lectures", or "exams", such as timetables for trains, planes, manufacturing plants, and so on. The kinds of constraints seem completely different, but, perhaps, they can be defined at a more abstract, architectural level: What do they all have in common?. Is it possible to define a common "shell" and even develop a kind of program, able to create a tool for each kind of timetable (meta-program)?

This was an idea I had in mind from the beginning (in fact, following it or not, was one of the main decisions in the system), but it would have been a bad direction of work, since I had not enough experience in solving this kind of problem using ATMS, and the lack of time would have condemned it to failure. In addition, if I wanted something to work efficiently by taking advantage of all details, it had to be done in an *Ad Hoc* way, and it was.

However, having finished this version, it could be a good moment to attempt this. I leave it also as an open task for other, after looking at my project results. I believe that such a "Timetabling" general scheme would be as useful and important as similar efforts done on "Qualitative Reasoning" or "Scheduling". I believe that such a meta-program would be very useful.

6.3 Dependency Directed Backtracking

As explained in chapter 5, dependency directed backtracking was not the right choice in the scheme I constructed, for many reasons. However, a generalization of the problem, as suggested in section before, may make it more desirable than the way I followed. It would be very interesting a research study about:

- constructing an efficient timetabling "dependency directed backtracking" algorithm
- applying it properly to a system like this, while avoiding the problems mentioned in chapter 5

This line of research could even be followed in relation to the future development of EDS project.

6.4 Efficient chronological backtracking using ATMS

As explained in chapter 5, one of the reasons to reject any *dependency directed backtracking* strategy was due to the fact that both Prolog and ATMS were more suitable for a *chronological backtracking* approach. However, since this system is more oriented to avoid backtracking at levels before (undesirable backtracking), no one of them is really used if our choice is "nobacktrack", as explained before.

The fact is that *it is extremely easy* to provide a mechanism to use information kept inside ATMS at previous attempts in order to avoid "re-inventing the wheel" later. In the example shown in 5.1.8 ("backtrack" option), the second time we test [mon,10] at level 4, we'd better test if ATMS has rejected it before (for any reason), instead of searching for problems again. This would produce an

ATMS "supported" *chronological backtracking*, easy to follow in a system like this (Prolog + ATMS), and, perhaps, as efficient as *dependency directed backtracking*, with no one of its disadvantages).

I designed such a mechanism (less than 20 lines) but I did not include it because of the problems created by "undesirable backtracking", and I decided to *recommend* "nobacktracking" option, where such a mechanism is not needed. However, if point 4 of this "further research" list is covered, this option would be very useful, as an alternative to point 5. Perhaps a comparison of both strategies would be a very intertesting project itself, using my project as an initial scheme for ideas.

6.5 Data Input: An "implementation independent" Front-End

If we look at the system from an academic point of view, it is computationally very "purist", quite efficient and can solve almost any "lectures" problem which is formulated adequately. However, from the point of view of a potential user, the input of data is, perhaps, not very nice, as Graeme Ritchie made me realise, for some reasons:

- The user would prefer to enter the list of teachers and students and their "human" constraints, so that the system may deduce "lectures" and "subjects" constraints, instead of calculating them himself (e.g.: list of teachers constraints that lead to "notpos" times for their subjects, and rules like "two subjects are *nonsimult* if and only if there exists a student that takes both of them", and so on).

- The user would like to enter constraints in a positive or negative - disjunctive or conjunctive *normal form* (DNF/CNF) style, depending on the teacher/subject constraints. The way the constraints are entered corresponds to a conjunction on negative clauses, which is not always the best (e.g.: a subject that has only three suitable times, should not force the user to enter 37 "notpos" (negative-conjunctive CNF) constraints: he could write 3 positive-disjunctive DNF suitable times, and let the system do the rest).

- The user may not know Prolog and may not like, either to create a Prolog file of constraints, or write them one by one in the system, the other option provided.

However, while the points covered in previous sections are important, this one is of little theoretical interest to AI researchers. An user interface is a very easy and simple facility which any programmer may provide, but very *controversial*, since it is not clear what the user really likes as a presentation, and *time consuming*, even if it does not involve Artificial Intelligence at all. I preferred to separate completely the origin of constraints (students-teachers) and the real lectures constraints problem.

I agree, anyway, that it would be very important in a real system and it should be done in this case, if it would move to a commercial system.

6.6 Combined Options

There is another proposal related to the previous one: to take advantage of the orthogonality of the options in the system, in order to produce combined options so that the user does not have to "think" (e.g.: the user would say "I do not like this lecture here. Where else may I put it": Equivalent to "add notpos" plus "resolve". It could be even iterated to see all possible places, include more hints, ...).

I think that, as in the previous extension, this is easy, but controversial and time consuming. It should be done if destinated to a commercial system.

Chapter 7

Conclusions

This project is the result of the initial objectives in the project, plus some facts that were revealed later, and some decisions I had to take:

- I had to choose between a basic "Support" system or a system that really *solve* the problem and, in addition, give additional support to the user. I found the latter as the right choice, even if it was more hard work.

- I had to choose between a global architecture for timetables or an *Ad Hoc* case to be studied deeply. I chose the second one, as it allowed me to construct the kind of system explained in the previous item.

- I had to choose between an abstract study of how to construct such a system, advantages and disadvantages, or to construct a real system, able to be used by a real user, with concessions to presentation, information to the user, and so on. I found the second one as the most natural one.

The final system is the result of these decisions, together with the ~~lack of~~ limited time in which to do the project (21 weeks), and the need of a compromise between the development of the different sections in the work: Dedicating more time to some of them would have probably spoiled drastically other areas.

I will summarise now the main points about what has been achieved in the system:

1. The *Hierarchical Plan* designed to solve the problem happened to be a good decision, since it really solves the problem and it is very efficient,

as can be seen while using the system. It also allowed the use of "search trees" as debugging tools, and many other facilities for the user, as shown before. I believe that the nightmare produced by a "Brute Force" search with backtracking, does not resist a comparison with it.

2. The set of user options:

   - adding and deleting constraints,
   - moving lectures,
   - resolving the timetable at any moment,
   - saving and loading either only constraints, or a complete session,

   and the other user utilities, such as:

   - search trees,
   - history of a session,
   - user information about the status at any moment,

   together with the graphics interface using

   - windows,
   - menus, and
   - a nice layout of the timetables,

   form a complete and reliable system not only as an interesting toy example, but to be used in *real problems.*

3. An ATMS has proved to be a very good tool to cope with "resource management" problems, and, if used properly, gives *complete* and *reliable* information to be used in Problem Solving.

This project was itself another kind of "resource management" problem (where I was the main resource), that I had to attempt with my "natural intelligence". The work described in this thesis is the result.

Conclusion now much better.

Bibliography

[AIAI 87] Knowledge Based Planning Systems Group AIAI. *Truth Maintenance Systems.* Technical Report, Artificial Intelligence Applications Institute, June 1987.

[Arlabosse *et al* 88] Francois Arlabosse, Bruno Jean-Bart, Nathalie Porte, and Beatrice deRavinel. An efficient problem solving architecture using atms, tested on a non-toy case study. *AI Communications Vol. 1 No. 4, pp. 6-15*, December 1988.

[deKleer & Williams 86] Johan deKleer and Brian C. Williams. Back to backtracking: controlling the atms. *Proceedings of the National Conference on Artificial Intelligence, Philadelphia, pp 910-917*, May 1986.

[deKleer 84] Johan deKleer. Choices without backtracking. *Proceedings of the National Conference on Artificial Intelligence, Austin, Texas, pp 79-85*, August 1984.

[deKleer 86a] Johan deKleer. An assumption-based tms. *Artificial Intelligence No. 28, pp. 127-162*, May 1986.

[deKleer 86b] Johan deKleer. Extending the atms. *Artificial Intelligence No. 28, pp. 163-196*, May 1986.

[deKleer 86c] Johan deKleer. Problem solving with the atms. *Artificial Intelligence No. 28, pp. 197-224*, May 1986.

[deKleer 88] Johan deKleer. A general labelling algorithm for assumption-based truth maintenance. *Proceedings of the National Conference on Artificial Intelligence, pp 188-192*, July 1988.

[deKleer *et al* 87] Johan deKleer, Kenneth D. Forbus, and Brian C. Williams. Truth maintenance systems. *National Conference on Artificial Intelligence, Tutorial No: TA4*, July 1987.

[Dixon & deKleer 88] Michael Dixon and Johan deKleer. Massively parallel assumption-based truth maintenance. *Proceedings of the National Conference on Artificial Intelligence, pp 199-204*, July 1988.

[Doyle 79] J. Doyle. A truth maintenance system. *Artificial intelligence No. 12, pp. 231-272*, 1979.

[Forbus & deKleer 88] Kenneth D. Forbus and Johan deKleer. Focusing the atms. *Proceedings of the National Conference on Artificial Intelligence, pp 193-198*, July 1988.

[Forbus 87] Kenneth D. Forbus. Building problem solvers: program notes on truth maintenance systems. *National Conference on Artificial Intelligence, Tutorial No: TA4*, July 1987.

[Koff *et al* 88] Caroline N. Koff, Nicholas S. Flann, and Thomas G. Dietterich. An efficient atms for equivalence relations. *Proceedings of the National Conference on Artificial Intelligence, pp 182-187*, July 1988.

[Rich 78] Elaine Rich. *Artificial Intelligence.* McGraw-Hill, 1978.

[Ross 87] Peter Ross. *A simple ATMS.* DAI Software Paper No. 4, Department of Artificial Intelligence, University of Edinburgh, February 1987.

[Smith 88] Barbara M. Smith. Forward checking, the atms and search reduction. *Reason Maintenance Systems and their Applications, Edited by Smith and Kelleher. Ellis-Horwood*, 1988.

[Smithers 85] Tim Smithers. The alvey large scale demonstrator project 'design to product'. *Artificial Intelligence in Manufacturing, Key to Integration?, North-Holland (reprinted in 1987)*, November 1985.

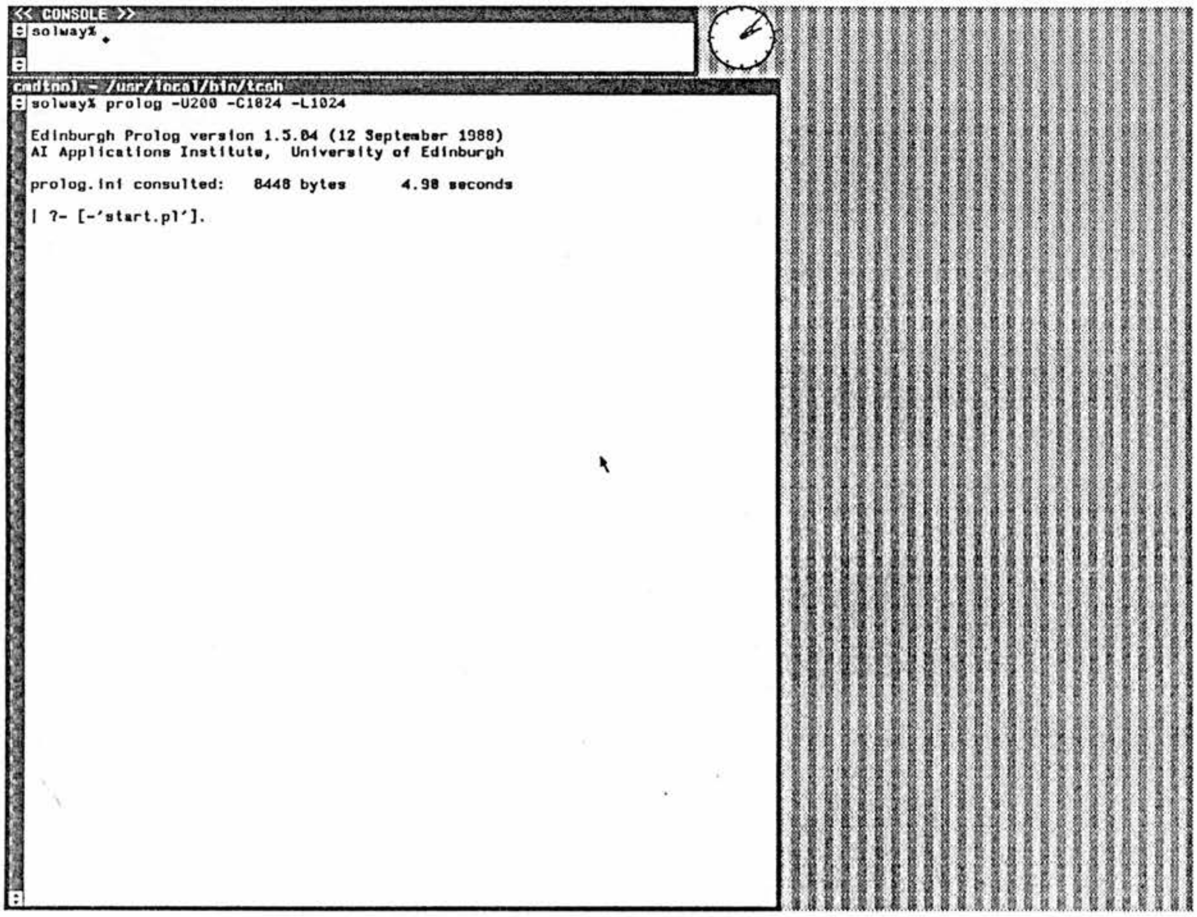
[Smithers *et al* 89] Tim Smithers, Alistair Conkie, Jim Doheny, Brian Logan, and Karl Millington. Design as intelligent behaviour. an ai in design research programme. *For submission to the Design Theme of the Fourth International Conference on Applications of Artificial Intelligence in Engineering, Cambridge, England*, July 1989.

Appendix A

System Screens

This appendix contains 12 screens, as a layout of a whole, small session. They will appear beginning in next page (screen pages are numbered 1 to 12, to simplify the search).

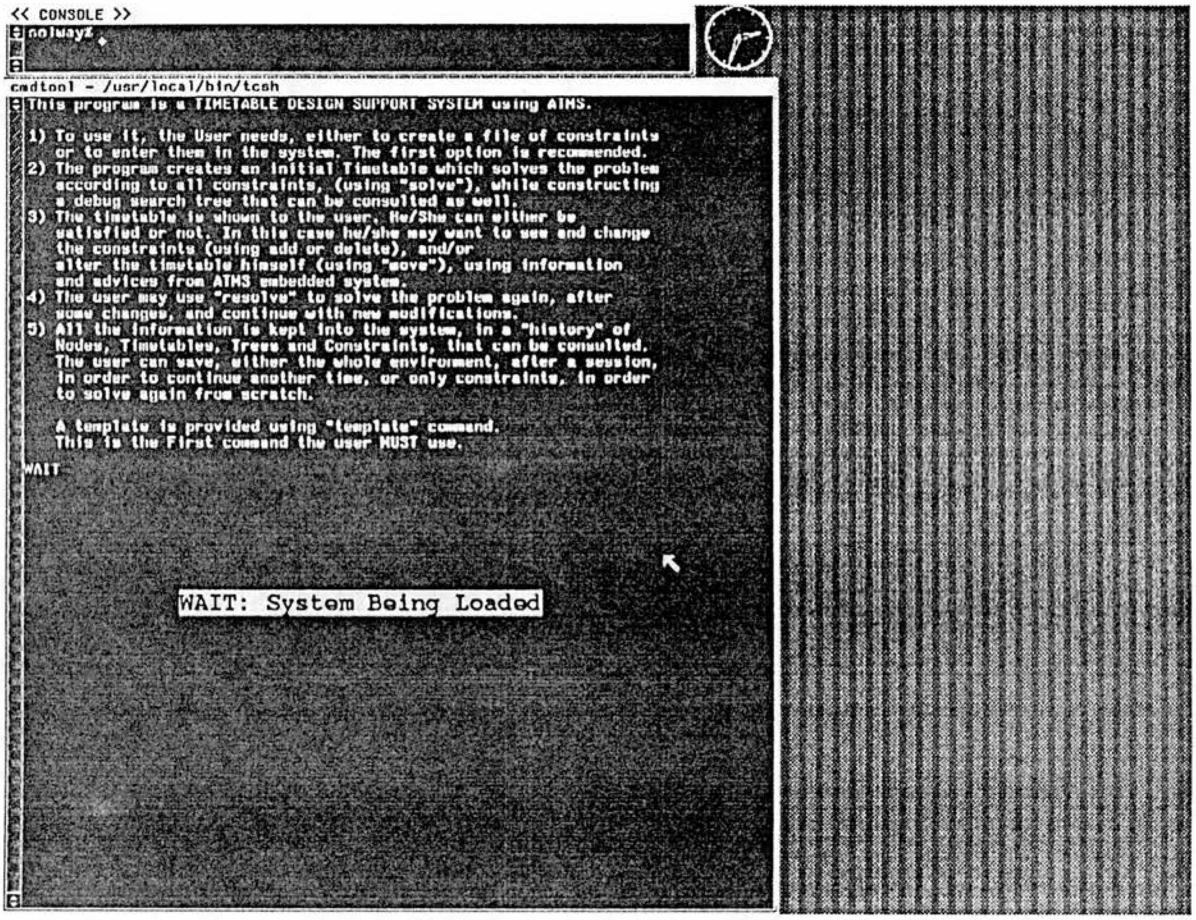
The first screen (1) serves to orient the user about how to begin. The display shows parameters for working under Prolog, and the way to get started with the system, reconsulting the file 'start.pl'



```
solway%
```

```
solway% prolog -U200 -C1024 -L1024

Edinburgh Prolog version 1.5.04 (12 September 1988)
AI Applications Institute,  University of Edinburgh

prolog.ini consulted:   8448 bytes      4.98 seconds

| ?- [-'start.pl'].
```

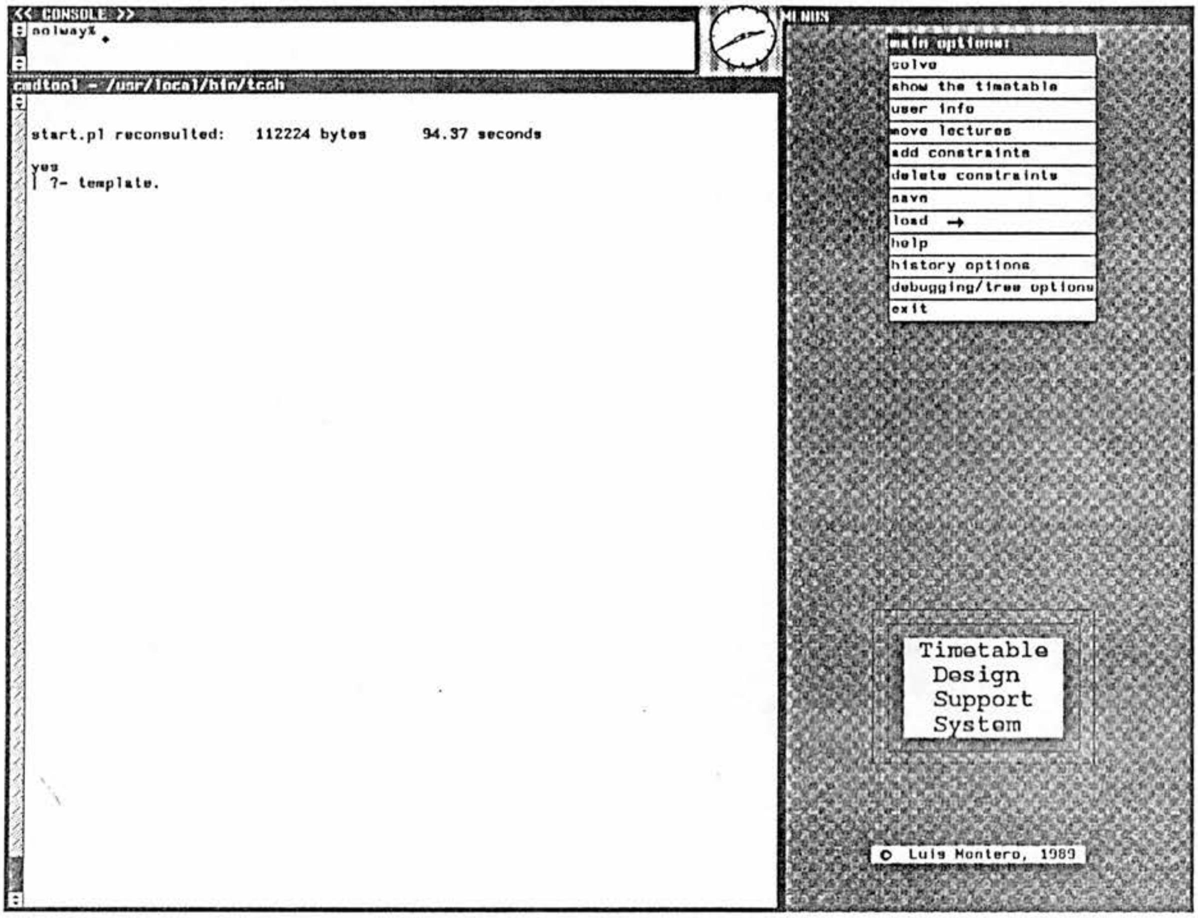
Next screen (2) shows the next step, when the system is reconsulting the other system files, and how the screen is seen if working on black background colour (strongly reccomended for working in the terminal, as eyes will suffer much less).



```
This program is a TIMETABLE DESIGN SUPPORT SYSTEM using ATMS.

1) To use it, the User needs, either to create a file of constraints
   or to enter them in the system. The first option is recommended.
2) The program creates an initial Timetable which solves the problem
   according to all constraints, (using "solve"), while constructing
   a debug search tree that can be consulted as well.
3) The timetable is shown to the user. He/She can either be
   satisfied or not. In this case he/she may want to see and change
   the constraints (using add or delete), and/or
   alter the timetable himself (using "move"), using information
   and advices from ATMS embedded system.
4) The user may use "resolve" to solve the problem again, after
   some changes, and continue with new modifications.
5) All the information is kept into the system, in a "history" of
   Nodes, Timetables, Trees and Constraints, that can be consulted.
   The user can save, either the whole enviroment, after a session,
   in order to continue another time, or only constraints, in order
   to solve again from scratch.

   A template is provided using "template" command.
   This is the First command the user MUST use.

WAIT
```

```
WAIT: System Being Loaded
```

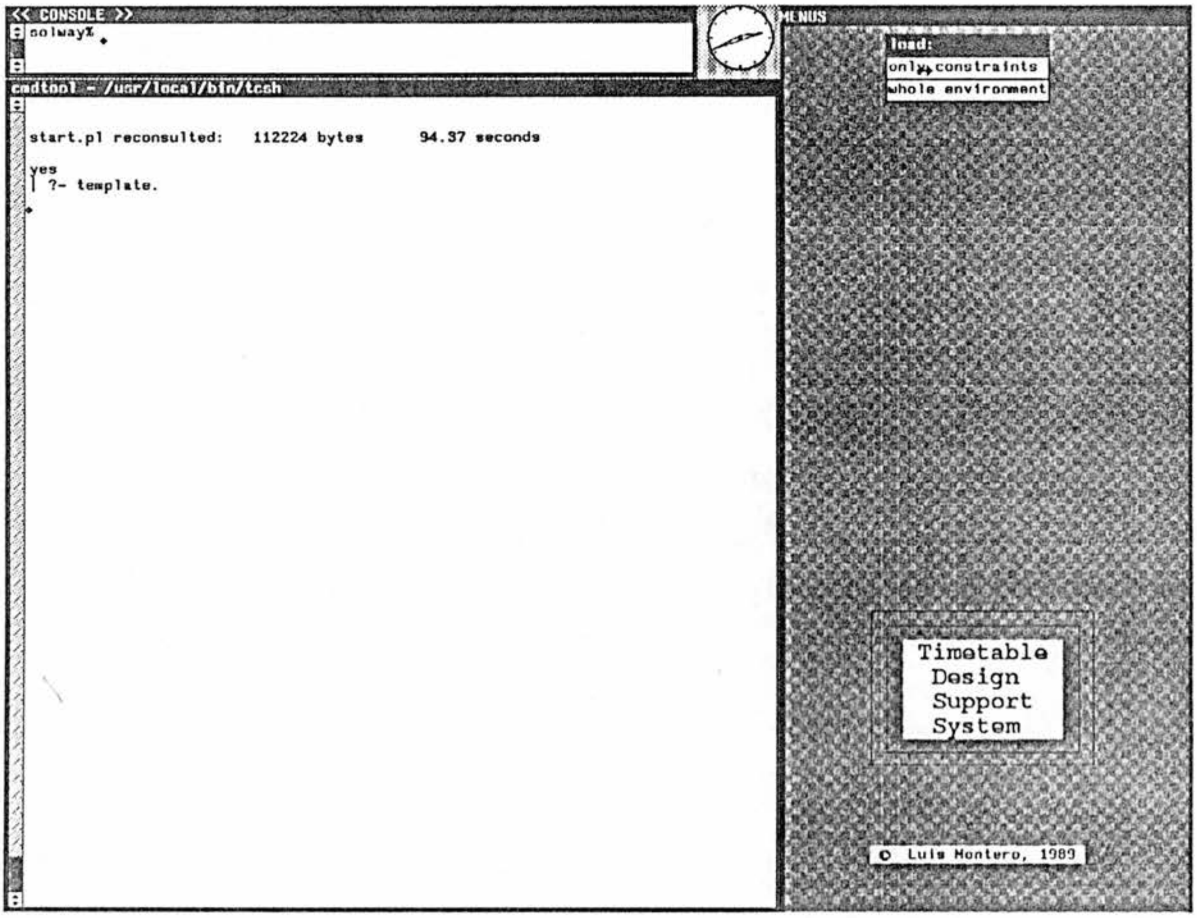
Next screen (3) shows the main menu, once the user has entered "template". All choices can be seen here, though some of them lead to new menus of options. We start by loading (load option) the file containing the constraints.



```
noIway%
```

```
start.pl reconsulted:   112224 bytes      94.37 seconds

yes
| ?- template.
```

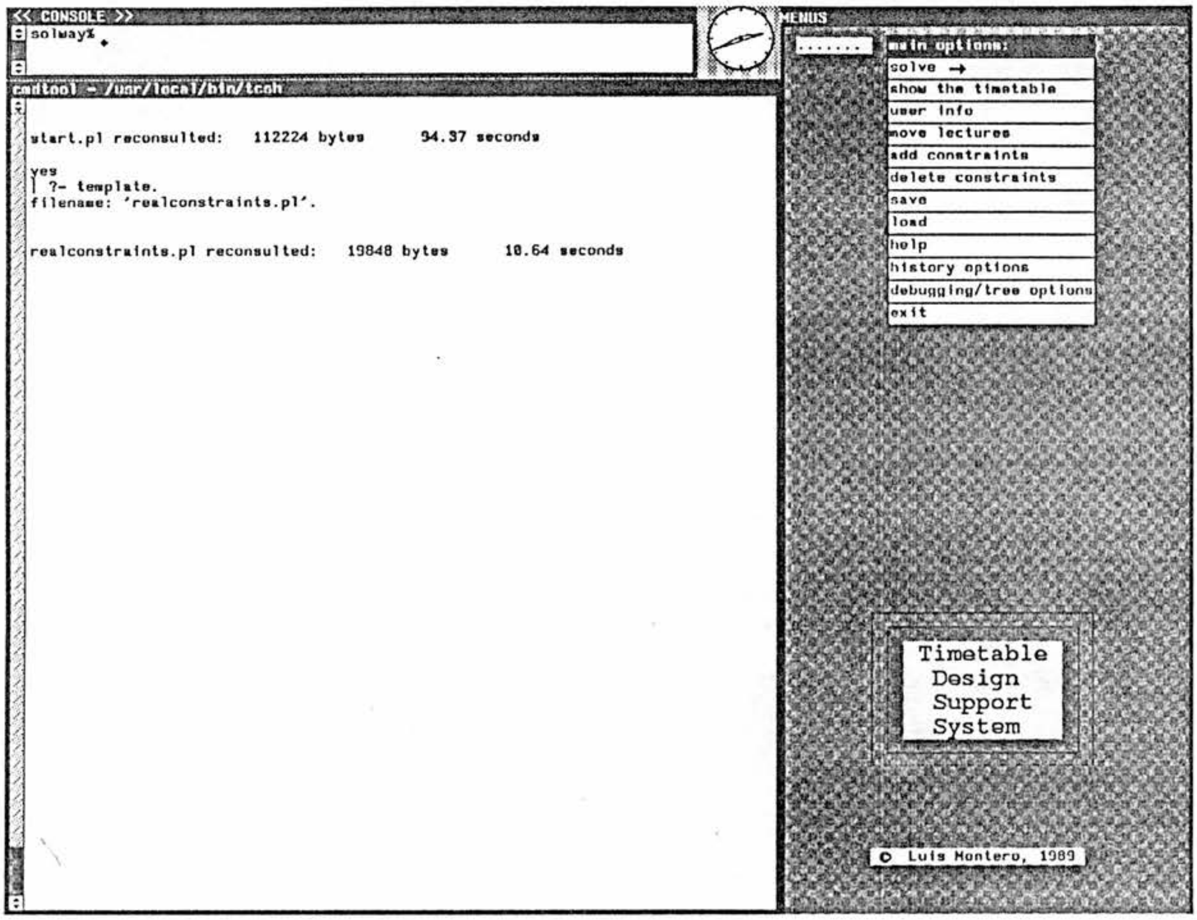
Next screen (4) shows the two choices offered to the user: load (or save, if that was the option selected before) either constraints alone or the whole environment after a session. Saving the whole session at the end is a good practice, so that, in the future, the user can recover the saved session and continue with it. In this case, we are starting from scratch, so that only constraints have to be loaded.



```
solway%
```

```
start.pl reconsulted:   112224 bytes      94.37 seconds

yes
| ?- template.
```

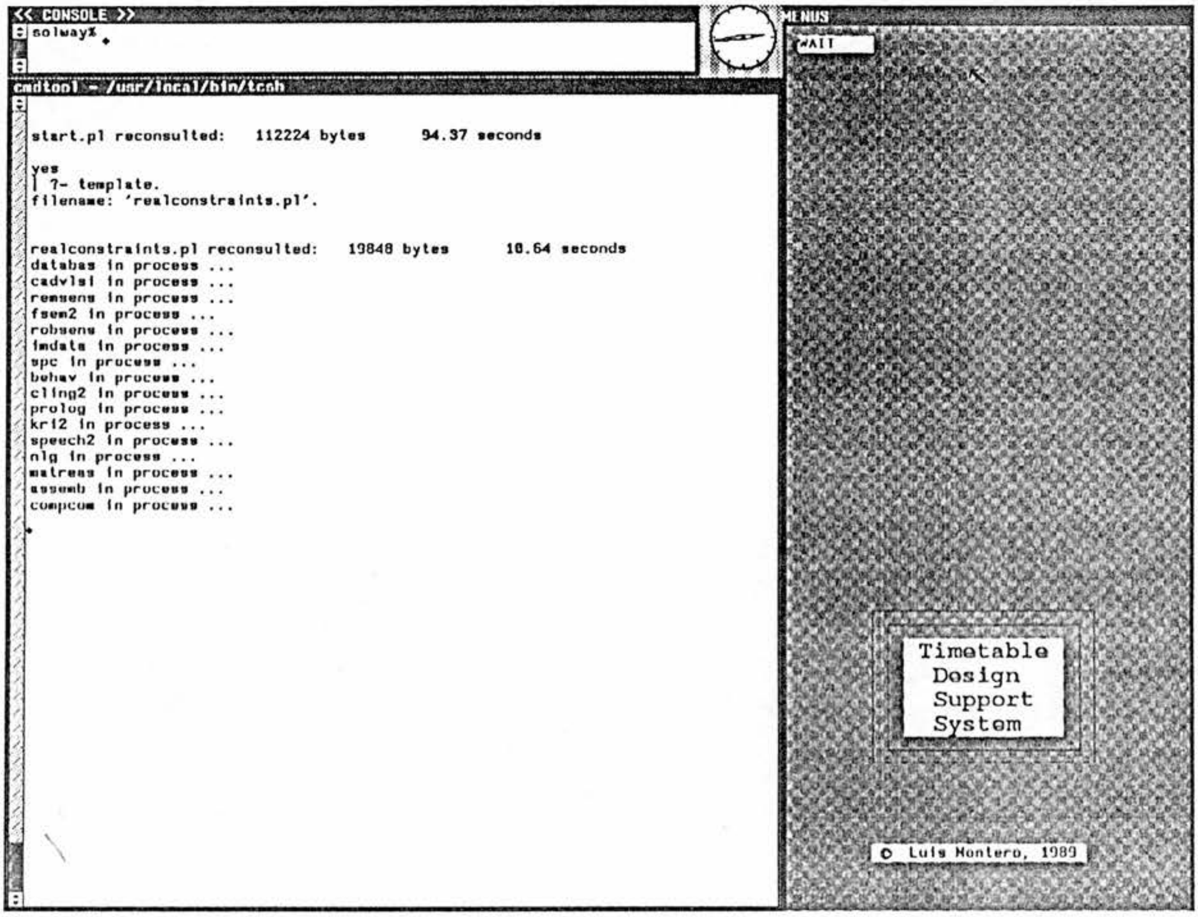
Next screen (5) shows how the file has been loaded. Now we select the "solve" option in order to find an initial solution.



```
solway%
```

```
start.pl reconsulted:   112224 bytes      34.37 seconds

yes
| ?- template.
filename: 'realconstraints.pl'.


realconstraints.pl reconsulted:   19848 bytes      10.64 seconds
```

Next screen (6) shows how "solve" works, setting the lectures for the respective subjects, in the right order (hierarchy).

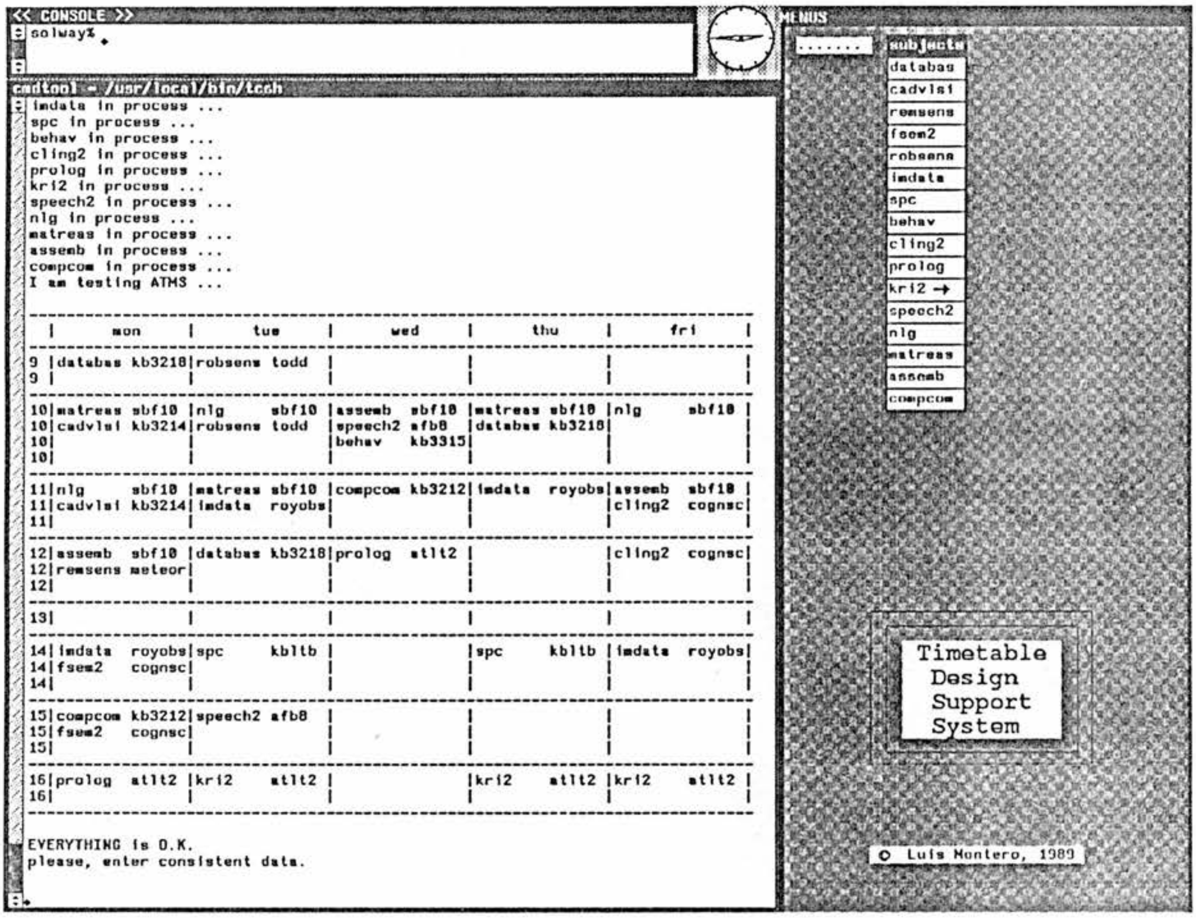


<< CONSOLE >>
cmdtool - /usr/local/bin/tcsh
MENUS
WAIT
Timetable Design Support System
© Luis Montero, 1989

```
solway%
```

```
start.pl reconsulted:   112224 bytes      94.37 seconds

yes
| ?- template.
filename: 'realconstraints.pl'.


realconstraints.pl reconsulted:   19848 bytes      10.64 seconds
databas in process ...
cadvlsi in process ...
remsens in process ...
fsem2 in process ...
robsens in process ...
imdata in process ...
spc in process ...
behav in process ...
cling2 in process ...
prolog in process ...
kri2 in process ...
speech2 in process ...
nlg in process ...
matreas in process ...
assemb in process ...
compcom in process ...
```

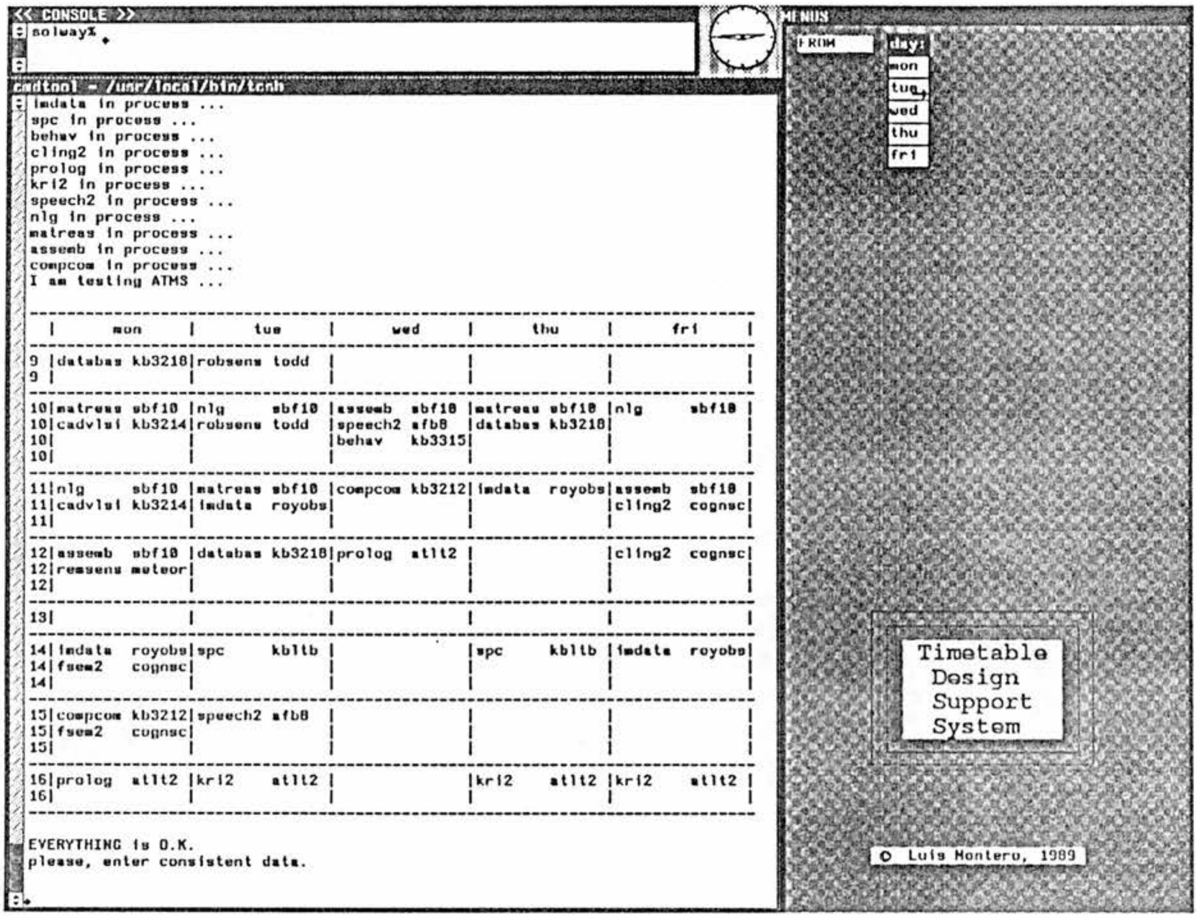
Next screen (7) shows the result after "solve" is done, and the timetable obtained. If the user decides, then, to move a lecture he/she does not like (e.g.: the one for "kri2" on tuesday), all he/she has to do is select that option from the menu, as shown in screen 3. The user will be told to enter the name of the subject, of course.



```
solway%
```

```
imdata in process ...
spc in process ...
behav in process ...
cling2 in process ...
prolog in process ...
kri2 in process ...
speech2 in process ...
nlg in process ...
matreas in process ...
assemb in process ...
compcom in process ...
I am testing ATMS ...

---------------------------------------------------------------------------
  |     mon      |     tue      |     wed      |     thu      |     fri      |
---------------------------------------------------------------------------
9 |databas kb3218|robsens todd  |              |              |              |
9 |              |              |              |              |              |
---------------------------------------------------------------------------
10|matreas sbf10 |nlg     sbf10 |assemb  sbf10 |matreas sbf10 |nlg     sbf10 |
10|cadvlsi kb3214|robsens todd  |speech2 afb8  |databas kb3218|              |
10|              |              |behav   kb3315|              |              |
10|              |              |              |              |              |
---------------------------------------------------------------------------
11|nlg     sbf10 |matreas sbf10 |compcom kb3212|imdata  royobs|assemb  sbf10 |
11|cadvlsi kb3214|imdata  royobs|              |              |cling2  cognsc|
11|              |              |              |              |              |
---------------------------------------------------------------------------
12|assemb  sbf10 |databas kb3218|prolog  atlt2 |              |cling2  cognsc|
12|remsens meteor|              |              |              |              |
12|              |              |              |              |              |
---------------------------------------------------------------------------
13|              |              |              |              |              |
---------------------------------------------------------------------------
14|imdata  royobs|spc     kbltb |              |spc     kbltb |imdata  royobs|
14|fsem2   cognsc|              |              |              |              |
14|              |              |              |              |              |
---------------------------------------------------------------------------
15|compcom kb3212|speech2 afb8  |              |              |              |
15|fsem2   cognsc|              |              |              |              |
15|              |              |              |              |              |
---------------------------------------------------------------------------
16|prolog  atlt2 |kri2    atlt2 |              |kri2    atlt2 |kri2    atlt2 |
16|              |              |              |              |              |
---------------------------------------------------------------------------

EVERYTHING is O.K.
please, enter consistent data.
```

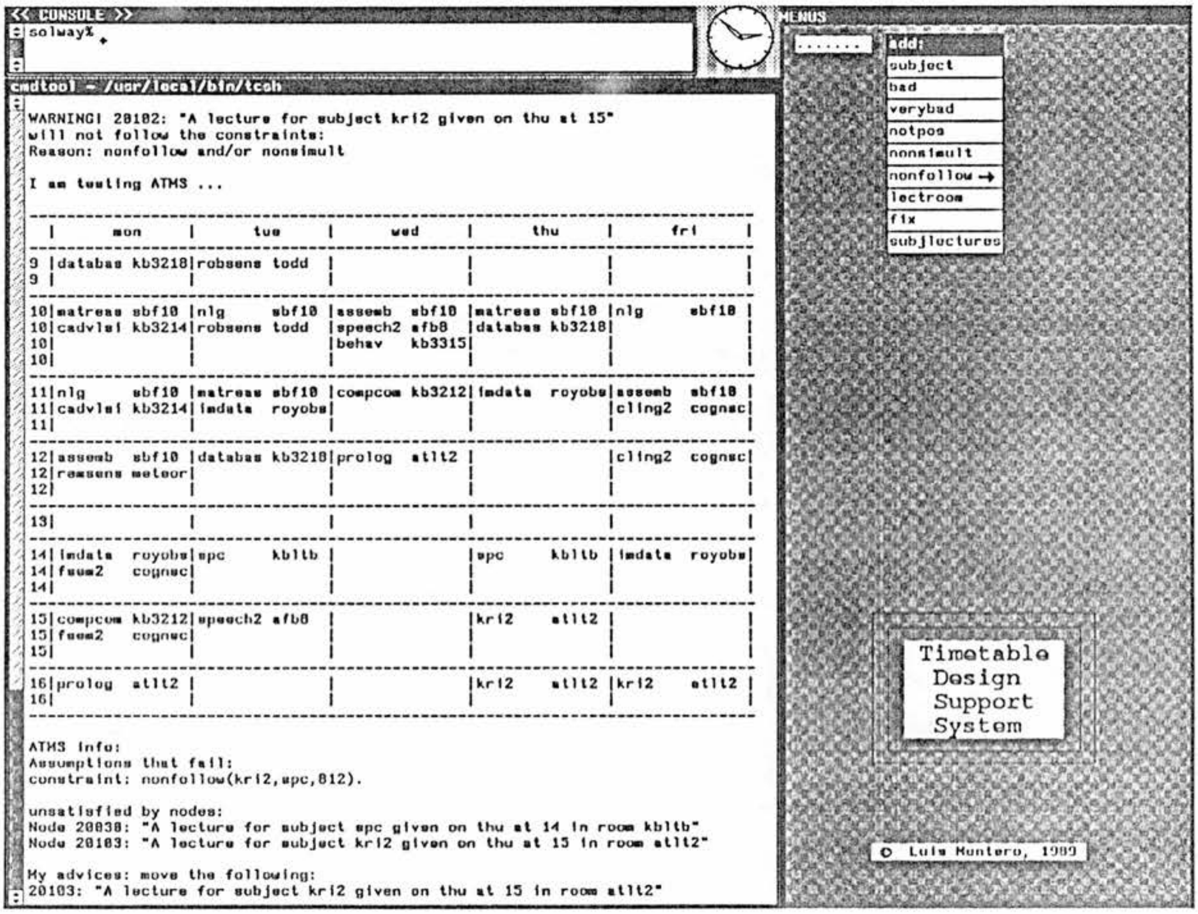
Next screen (8) shows the next step, when the user is told to enter the original time and room, and the destination ones, so that new menus, as the one shown above, will appear.



```
solway%
```

```
imdata in process ...
spc in process ...
behav in process ...
cling2 in process ...
prolog in process ...
kri2 in process ...
speech2 in process ...
nlg in process ...
matress in process ...
assemb in process ...
compcom in process ...
I am testing ATMS ...

------------------------------------------------------------------------------
  |     mon      |     tue      |     wed      |     thu      |     fri      |
------------------------------------------------------------------------------
9 |databas kb3218|robsens todd  |              |              |              |
9 |              |              |              |              |              |
------------------------------------------------------------------------------
10|matress sbf10 |nlg     sbf10 |assemb  sbf10 |matress sbf10 |nlg     sbf10 |
10|cadvlsi kb3214|robsens todd  |speech2 afb8  |databas kb3218|              |
10|              |              |behav   kb3315|              |              |
10|              |              |              |              |              |
------------------------------------------------------------------------------
11|nlg     sbf10 |matress sbf10 |compcom kb3212|imdata  royobs|assemb  sbf10 |
11|cadvlsi kb3214|imdata  royobs|              |              |cling2  cognsc|
11|              |              |              |              |              |
------------------------------------------------------------------------------
12|assemb  sbf10 |databas kb3218|prolog  atlt2 |              |cling2  cognsc|
12|remsens meteor|              |              |              |              |
12|              |              |              |              |              |
------------------------------------------------------------------------------
13|              |              |              |              |              |
------------------------------------------------------------------------------
14|imdata  royobs|spc     kbltb |              |spc     kbltb |imdata  royobs|
14|fsem2   cognsc|              |              |              |              |
14|              |              |              |              |              |
------------------------------------------------------------------------------
15|compcom kb3212|speech2 afb8  |              |              |              |
15|fsem2   cognsc|              |              |              |              |
15|              |              |              |              |              |
------------------------------------------------------------------------------
16|prolog  atlt2 |kri2    atlt2 |              |kri2    atlt2 |kri2    atlt2 |
16|              |              |              |              |              |
------------------------------------------------------------------------------

EVERYTHING is O.K.
please, enter consistent data.
```

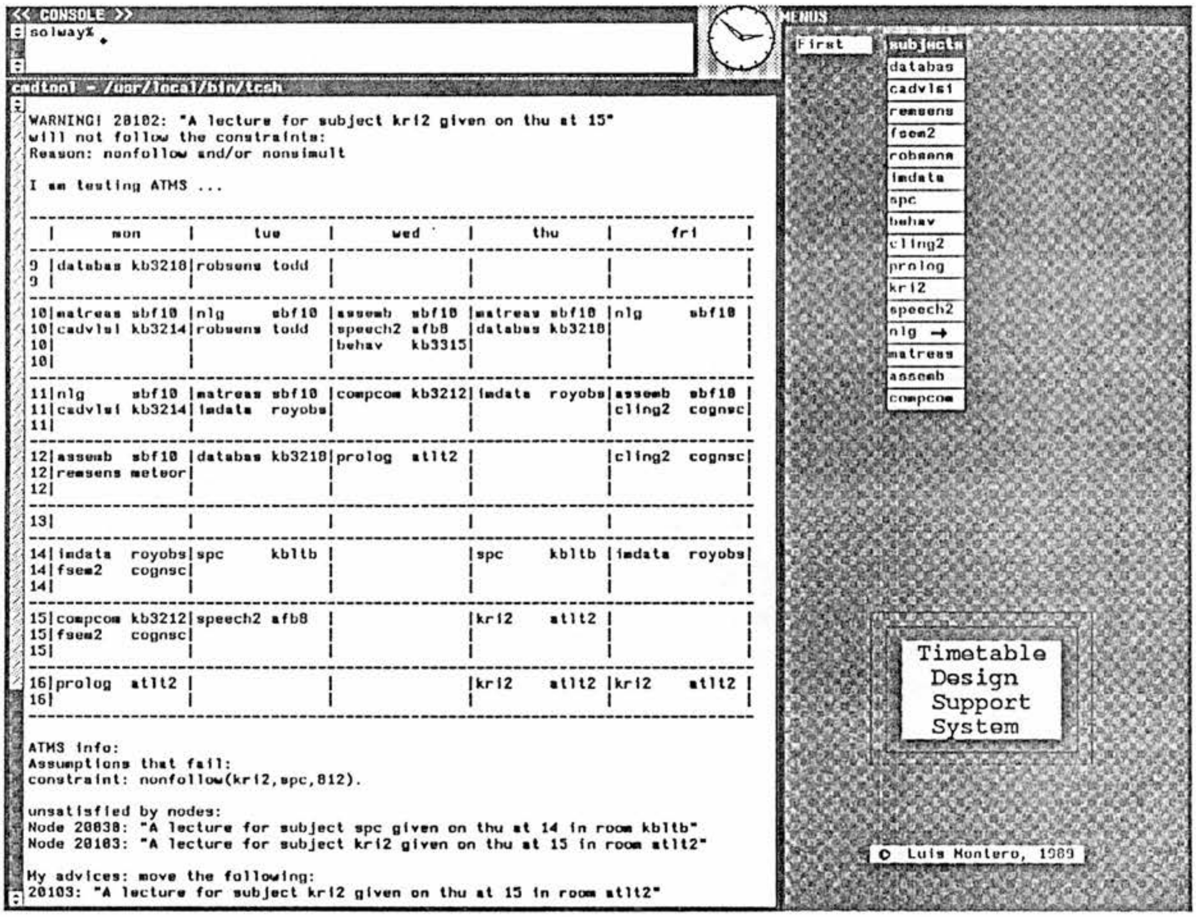
Next screen (9) shows the result after the movement has been done. The system checks if that change affects the consistency (it does!). The new timetable is shown, as well as the failing constraints, and some advice about what to do, if the user wants to solve the problem him/herself. Let's suppose that the user decides, then, to add a new constraint (e.g.: natural language - nlg - and mathematical reasoning - matreas - should not follow each other). He/she will have to select "add constraints" option from the main menu, as shown in screen 3, and a new menu of options, as the one shown below, will appear. The user will select "nonfollow".



```
solway%
```

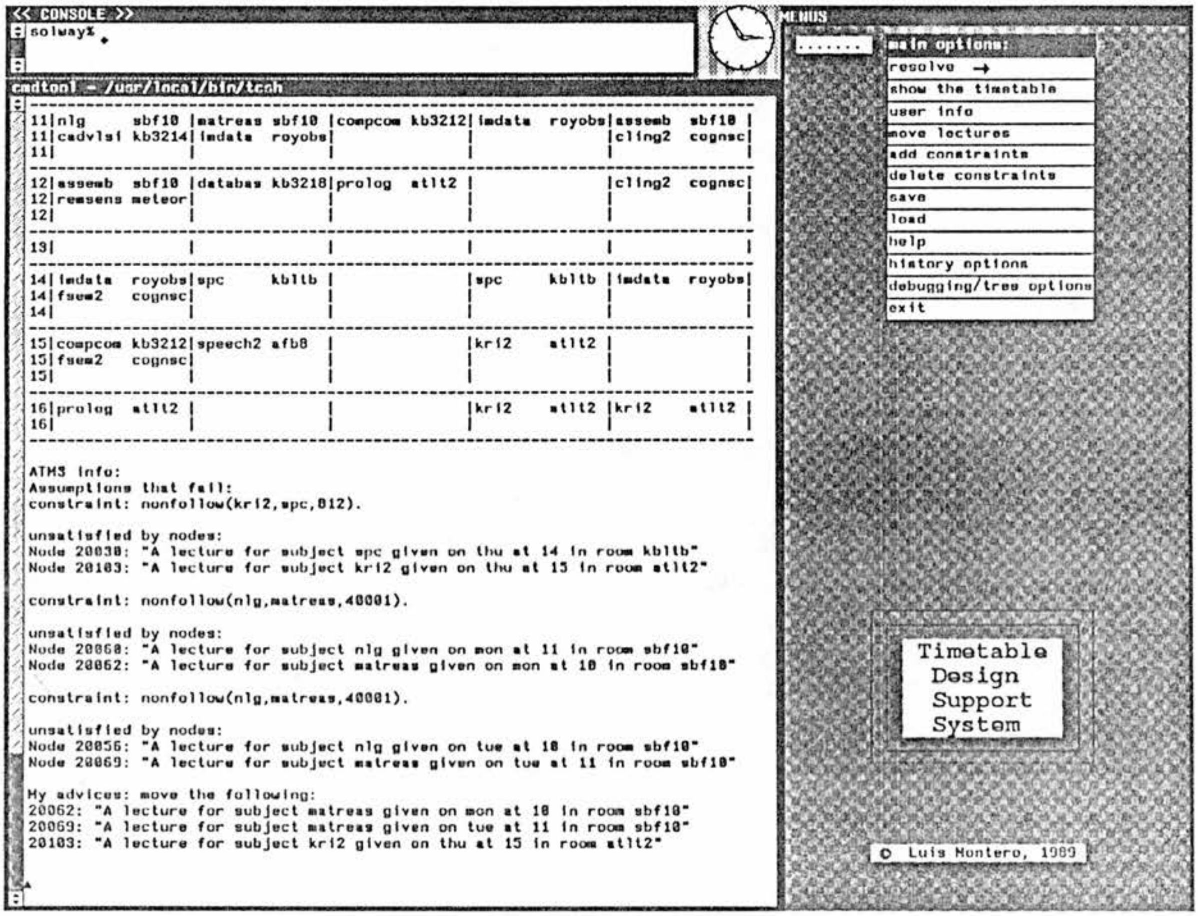
```
WARNING! 20182: "A lecture for subject kri2 given on thu at 15"
will not follow the constraints:
Reason: nonfollow and/or nonsimult

I am testing ATMS ...

-------------------------------------------------------------------------
  |      mon      |      tue      |      wed      |      thu      |      fri      |
-------------------------------------------------------------------------
9 |databas kb3218|robsens todd  |               |               |               |
9 |               |               |               |               |               |
-------------------------------------------------------------------------
10|matreas sbf10 |nlg     sbf10 |assemb  sbf10 |matreas sbf10 |nlg     sbf10 |
10|cadvlsi kb3214|robsens todd  |speech2 sfb8  |databas kb3218|               |
10|               |               |behav   kb3315|               |               |
10|               |               |               |               |               |
-------------------------------------------------------------------------
11|nlg     sbf10 |matreas sbf10 |compcom kb3212|imdata  royobs|assemb  sbf10 |
11|cadvlsi kb3214|imdata  royobs|               |               |cling2  cognsc|
11|               |               |               |               |               |
-------------------------------------------------------------------------
12|assemb  sbf10 |databas kb3218|prolog  atlt2 |               |cling2  cognsc|
12|remsens meteor|               |               |               |               |
12|               |               |               |               |               |
-------------------------------------------------------------------------
13|               |               |               |               |               |
-------------------------------------------------------------------------
14|imdata  royobs|spc     kbltb |               |spc     kbltb |imdata  royobs|
14|fsum2   cognsc|               |               |               |               |
14|               |               |               |               |               |
-------------------------------------------------------------------------
15|compcom kb3212|speech2 sfb8  |               |kri2    atlt2 |               |
15|fsem2   cognsc|               |               |               |               |
15|               |               |               |               |               |
-------------------------------------------------------------------------
16|prolog  atlt2 |               |               |kri2    atlt2 |kri2    atlt2 |
16|               |               |               |               |               |
-------------------------------------------------------------------------

ATMS info:
Assumptions that fail:
constraint: nonfollow(kri2,spc,812).

unsatisfied by nodes:
Node 20038: "A lecture for subject spc given on thu at 14 in room kbltb"
Node 20183: "A lecture for subject kri2 given on thu at 15 in room atlt2"

My advices: move the following:
20103: "A lecture for subject kri2 given on thu at 15 in room atlt2"
```

Next screen (10) shows again the fact that the system tells the user to enter the "First" and "Second" subject he/she wants to include in the new constraint. Other constraints may require different data, in the same way, unless they are new "names" (e.g.: new subjects or rooms), when entering them in the left window may be necessary. In this case, "nlg" subject is selected from the menu.



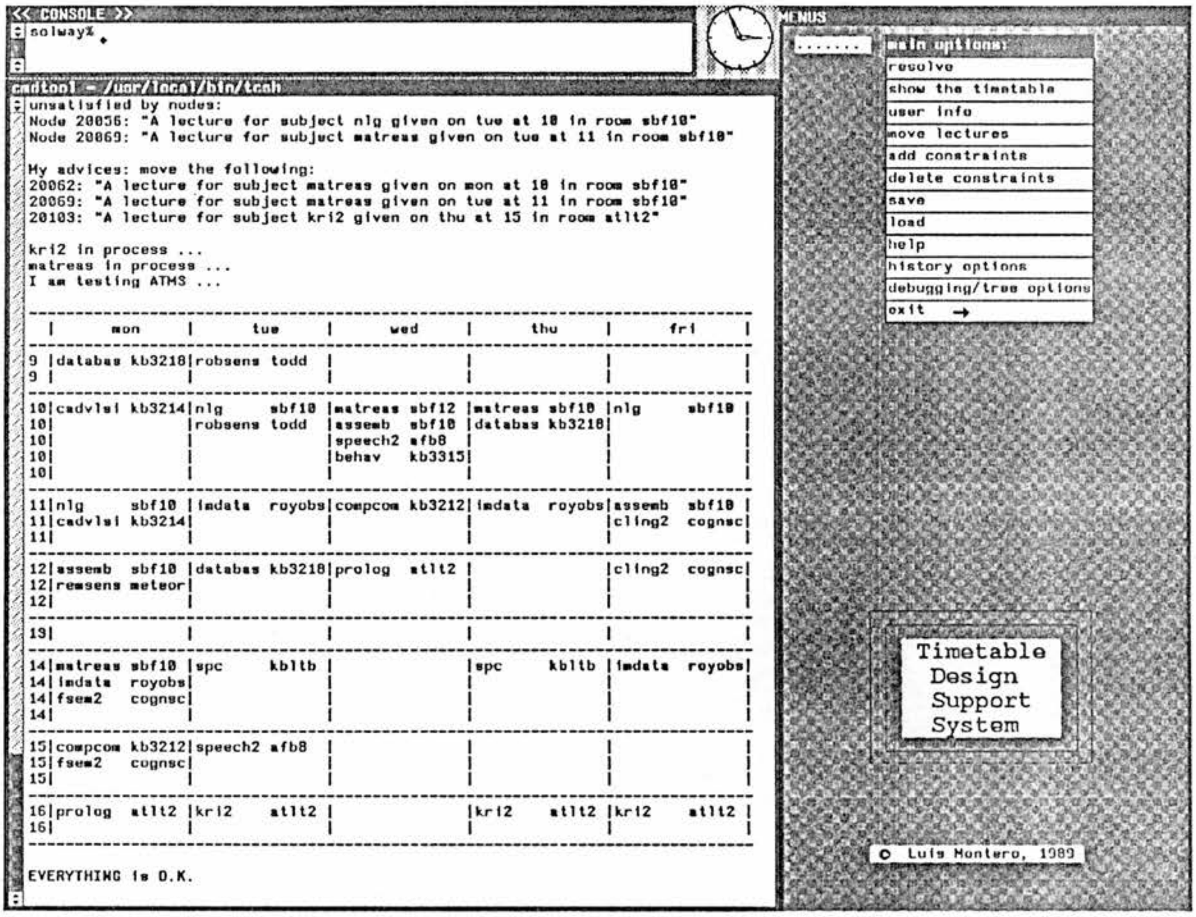
```
WARNING! 20102: "A lecture for subject kri2 given on thu at 15"
will not follow the constraints:
Reason: nonfollow and/or nonsimult

I am testing ATMS ...

---------------------------------------------------------------------------
  |       mon      |       tue      |       wed      |       thu      |       fri      |
---------------------------------------------------------------------------
9 |databas kb3210|robsens todd   |                |                |                |
9 |               |                |                |                |                |
---------------------------------------------------------------------------
10|matreas sbf10 |nlg     sbf10  |assemb  sbf10  |matreas sbf10  |nlg     sbf10  |
10|cadvlsi kb3214|robsens todd   |speech2 afb8   |databas kb3210|                |
10|               |                |behav   kb3315|                |                |
10|               |                |                |                |                |
---------------------------------------------------------------------------
11|nlg     sbf10 |matreas sbf10  |compcom kb3212|imdata  royobs|assemb  sbf10  |
11|cadvlsi kb3214|imdata  royobs|                |                |cling2  cognsc|
11|               |                |                |                |                |
---------------------------------------------------------------------------
12|assemb  sbf10 |databas kb3210|prolog  atlt2  |                |cling2  cognsc|
12|remsens meteor|                |                |                |                |
12|               |                |                |                |                |
---------------------------------------------------------------------------
13|               |                |                |                |                |
---------------------------------------------------------------------------
14|imdata  royobs|spc     kbltb  |                |spc     kbltb  |imdata  royobs|
14|fsem2   cognsc|                |                |                |                |
14|               |                |                |                |                |
---------------------------------------------------------------------------
15|compcom kb3212|speech2 afb8   |                |kri2    atlt2  |                |
15|fsem2   cognsc|                |                |                |                |
15|               |                |                |                |                |
---------------------------------------------------------------------------
16|prolog  atlt2 |                |                |kri2    atlt2  |kri2    atlt2  |
16|               |                |                |                |                |
---------------------------------------------------------------------------

ATMS info:
Assumptions that fail:
constraint: nonfollow(kri2,spc,012).

unsatisfied by nodes:
Node 20038: "A lecture for subject spc given on thu at 14 in room kbltb"
Node 20103: "A lecture for subject kri2 given on thu at 15 in room atlt2"

My advices: move the following:
20103: "A lecture for subject kri2 given on thu at 15 in room atlt2"
```

Next screen (11) shows the result after the last addition: timetable, failing constraints and lectures affected, and the system's advice ("scrolling" may be necessary if the user wants to look at the complete timetable, as can be seen). The user realises that there are many things to do and decides to select the "resolve" option, so that the system will solve it automatically again.



```
------------------------------------------------------------------------------
11|nlg     sbf10 |matreas sbf10 |compcom kb3212|imdata  royobs|assemb  sbf10 |
11|cadvlsi kb3214|imdata  royobs|              |              |cling2  cognsc|
11|              |              |              |              |              |
------------------------------------------------------------------------------
12|assemb  sbf10 |databas kb3218|prolog  atlt2 |              |cling2  cognsc|
12|remsens meteor|              |              |              |              |
12|              |              |              |              |              |
------------------------------------------------------------------------------
13|              |              |              |              |              |
------------------------------------------------------------------------------
14|imdata  royobs|spc     kbltb |              |spc     kbltb |imdata  royobs|
14|fsem2   cognsc|              |              |              |              |
14|              |              |              |              |              |
------------------------------------------------------------------------------
15|compcom kb3212|speech2 afb8  |              |kri2    atlt2 |              |
15|fsem2   cognsc|              |              |              |              |
15|              |              |              |              |              |
------------------------------------------------------------------------------
16|prolog  atlt2 |              |              |kri2    atlt2 |kri2    atlt2 |
16|              |              |              |              |              |
------------------------------------------------------------------------------

ATMS info:
Assumptions that fail:
constraint: nonfollow(kri2,spc,012).

unsatisfied by nodes:
Node 20030: "A lecture for subject spc given on thu at 14 in room kbltb"
Node 20103: "A lecture for subject kri2 given on thu at 15 in room atlt2"

constraint: nonfollow(nlg,matreas,40001).

unsatisfied by nodes:
Node 20060: "A lecture for subject nlg given on mon at 11 in room sbf10"
Node 20062: "A lecture for subject matreas given on mon at 10 in room sbf10"

constraint: nonfollow(nlg,matreas,40001).

unsatisfied by nodes:
Node 20056: "A lecture for subject nlg given on tue at 10 in room sbf10"
Node 20069: "A lecture for subject matreas given on tue at 11 in room sbf10"

My advices: move the following:
20062: "A lecture for subject matreas given on mon at 10 in room sbf10"
20069: "A lecture for subject matreas given on tue at 11 in room sbf10"
20103: "A lecture for subject kri2 given on thu at 15 in room atlt2"
```

And next screen (12) shows the new arrangement of lectures afterwards. The user may save the environment, if he/she wishes, as shown before. To exit the system, the user might use "exit" option and confirm (yes/no).



```
solway%
```

```
unsatisfied by nodes:
Node 20056: "A lecture for subject nlg given on tue at 10 in room sbf10"
Node 20069: "A lecture for subject matreas given on tue at 11 in room sbf10"

My advices: move the following:
20062: "A lecture for subject matreas given on mon at 10 in room sbf10"
20069: "A lecture for subject matreas given on tue at 11 in room sbf10"
20103: "A lecture for subject kri2 given on thu at 15 in room atlt2"

kri2 in process ...
matreas in process ...
I am testing ATMS ...

-------------------------------------------------------------------------------
  |     mon      |     tue      |     wed      |     thu      |     fri      |
-------------------------------------------------------------------------------
9 |databas kb3218|robsens todd  |              |              |              |
9 |              |              |              |              |              |
-------------------------------------------------------------------------------
10|cadvlsi kb3214|nlg     sbf10 |matreas sbf12 |matreas sbf10 |nlg     sbf10 |
10|              |robsens todd  |assemb  sbf10 |databas kb3218|              |
10|              |              |speech2 afb8  |              |              |
10|              |              |behav   kb3315|              |              |
10|              |              |              |              |              |
-------------------------------------------------------------------------------
11|nlg     sbf10 |imdata  royobs|compcom kb3212|imdata  royobs|assemb  sbf10 |
11|cadvlsi kb3214|              |              |              |cling2  cognsc|
11|              |              |              |              |              |
-------------------------------------------------------------------------------
12|assemb  sbf10 |databas kb3218|prolog  atlt2 |              |cling2  cognsc|
12|remsens meteor|              |              |              |              |
12|              |              |              |              |              |
-------------------------------------------------------------------------------
13|              |              |              |              |              |
-------------------------------------------------------------------------------
14|matreas sbf10 |spc     kbltb |              |spc     kbltb |imdata  royobs|
14|imdata  royobs|              |              |              |              |
14|fsem2   cognsc|              |              |              |              |
14|              |              |              |              |              |
-------------------------------------------------------------------------------
15|compcom kb3212|speech2 afb8  |              |              |              |
15|fsem2   cognsc|              |              |              |              |
15|              |              |              |              |              |
-------------------------------------------------------------------------------
16|prolog  atlt2 |kri2    atlt2 |              |kri2    atlt2 |kri2    atlt2 |
16|              |              |              |              |              |
-------------------------------------------------------------------------------

EVERYTHING is O.K.
```

I hope this appendix will simplify the work to any person attempting to use the system. Good luck!

Appendix B

System Files: start.pl

This appendix contains the text of the program that loads the main program, shows the main menu and initialises the system in order to work. The listing is shown in next page, in a format of two pages in one.

```
/*
File:    start.pl
Author:  Luis Montero, MSc Student (lmg@forth, lmg@aipna)
Purpose: Starting program for the Timetable Design Support System using ATMS.
*/

% ============================================================================
/*
PREDICATE: help
ARGUMENTS: NONE
COMMENTS:  Succeeds after showing information about the system
*/
% ============================================================================


help :-
 write('This program is a TIMETABLE DESIGN SUPPORT SYSTEM using ATMS.'),
 nl,
 nl,
 write('1) To use it, the User needs, either to create a file of constraints'),
 nl,
 write('   or to enter them in the system. The first option is recommended.'),
 nl,
 write('2) The program creates an initial Timetable which solves the problem'),
 nl,
 write('   according to all constraints, (using "solve"), while constructing'),
 nl,
 write('   a debug search tree that can be consulted as well.'),
 nl,
 write('3) The timetable is shown to the user. He/She can either be'),
 nl,
 write('   satisfied or not. In this case he/she may want to see and change'),
 nl,
 write('   the constraints (using add or delete), and/or'),
 nl,
 write('   alter the timetable himself (using "move"), using information'),
 nl,
 write('   and advices from ATMS embedded system.'),
 nl,
 write('4) The user may use "resolve" to solve the problem again, after'),
 nl,
 write('   some changes, and continue with new modifications.'),
 nl,
 write('5) All the information is kept into the system, in a "history" of'),
 nl,
 write('   Nodes, Timetables, Trees and Constraints, that can be consulted.'),
 nl,
 write('   The user can save, either the whole environment, after a session,'),
 nl,
 write('   in order to continue another time, or only constraints, in order'),
 nl,
 write('   to solve again from scratch.'),
 nl,
 nl,
 write('   A template is provided using "template" command.'),
 nl,
 write('   This is the First command the user MUST use.'),
 nl.


% ============================================================================
/*
PREDICATE: start
ARGUMENTS: NONE
COMMENTS:  Succeeds after asserting initial values in prolog database,
           coming from constraints or solution file, for efficiency reasons,
           in order to begin a new session
*/
% ============================================================================

start :-                          % Some predicates are in "newtimetable.pl"
 maxassumptnumber(MAN),
 putsetup(MAN),                   % look at "ATMS DATABASE" section
 days(Days),
 createtranslate(Days,1),         % look at "SOLVE OPTION" section
 hours(Hours),
 join(Days,Hours,Dayshours),      % look at "SOLVE OPTION" section
 asserta(dayshours(Dayshours)),
 asserta(subjects([])),
 asserta(timetable([])),
 asserta(unsolved([])),
 asserta(tracks([])).


:- [-'graph.pl'],
   shell("clear"),
   help,
   startserver,
   makeimage(1152,300,Image),
   readimagebinary(Image,'wait.lmg'),
   makewindow('WAIT',450,19,718,300,Waitwindow),
   copy(Waitwindow,0,0,1152,300,set,Image,260,0),
   asserta(notallowed(resolve)),
   [-'atms.pl'],
   [-'addons.pl'],
   [-'newtimetable.pl'],
   [-'defaults.pl'],
   start,
   shell("clear"),
   killwindow(Waitwindow).
```

Appendix C

System Files: addons.pl

This appendix contains the text of a program containing some "general purpose predicates widely used in the main program of this system. The listing starts in next page, in a format of two pages in one, with page numbers starting with 1.

```
/*
File:     addons.pl
Author:   Luis Montero, MSc Student (lmg@forth, lmg@aipna)
Purpose:  set of useful predicates frequently used in applications
*/

% ===========================================================================
/*
PREDICATE: member(X,+Y)
ARGUMENTS: X, anything
           Y, list      (input)
COMMENTS:  Succeeds if X is member of Y. It is valid either if
           X is bound (test), or not (find elements of Y)
*/
% ===========================================================================

member(X,[X|_]).

member(X,[_|T]) :-
 member(X,T).


% ===========================================================================
/*
PREDICATE: memberchk(+X,+Y)
ARGUMENTS: X, anything
           Y, list      (input)
COMMENTS:  Succeeds if X is member of Y. It has a RED CUT
*/
% ===========================================================================

memberchk(X,[X|_]) :-
 !.

memberchk(X,[_|T]) :-
 memberchk(X,T).


% ===========================================================================
/*
PREDICATE: membercdr(X,+Y)
ARGUMENTS: X, anything
           Y, list of pairs (input)
COMMENTS:  Succeeds if X is 'cdr' of a member of Y. It is valid either if
           X is bound (test) or not (find 'cdr's of elements in Y)
*/
% ===========================================================================

membercdr(X,[_/X|_]).

membercdr(X,[_|T]) :-
 membercdr(X,T).


% ===========================================================================
/*
PREDICATE: remove(+L,+X,?Y)
ARGUMENTS: L, list (Set) (input)
           X, anything
           Y, list       (output)
COMMENTS:  Succeeds after instantiating Y to the list L after removing
           the only appearance of X in it.
*/
% ===========================================================================
```

```
remove([],_,[]).

remove([X|T],X,T) :-
 !.


remove([Y|T],X,[Y|T1]) :-
 remove(T,X,T1).


% ===========================================================================
/*
PREDICATE: bagremove(+L,+X,?Y)
ARGUMENTS: L, list      (input)
           X, anything
           Y, list      (output)
COMMENTS:  Succeeds after instantiating Y to the list L after removing
           all appearances of X in it.
*/
% ===========================================================================

bagremove([],_,[]).

bagremove([X|T],X,T1) :-
 !,
 bagremove(T,X,T1).

bagremove([Y|T],X,[Y|T1]) :-
 bagremove(T,X,T1).


% ===========================================================================
/*
PREDICATE: recremove(+L,+X,?Y)
ARGUMENTS: L, list      (input)
           X, list      (input)
           Y, list      (output)
COMMENTS:  Succeeds after instantiating Y to the list L after removing
           all appearances of members of X in it.
*/
% ===========================================================================

recremove([],_,[]).

recremove([Z|T],X,T1) :-
 memberchk(Z,X),
 !,
 recremove(T,X,T1).

recremove([Y|T],X,[Y|T1]) :-
 recremove(T,X,T1).


% ===========================================================================
/*
PREDICATE: item(+Y,+Z,?T)
ARGUMENTS: Y, list
           Z, number
           T, anything
COMMENTS:  Succeeds after instantiating T to the Zth element in the
           list Y, or testing if T is the Zth element in Y
*/
% ===========================================================================
item([X|_],1,X) :-
 !.

item([_|Y],N,X) :-
 N > 1,
 N1 is N - 1,
 item(Y,N1,X).


% ===========================================================================
/*
PREDICATE: position(+X,+Y,?Z)
ARGUMENTS: X, anything
           Y, list
           Z, number
COMMENTS:  Succeeds after instantiating Z to 0 if X is not in Y, in the
           first level, or to the position it has in Y, otherwise.
           It is also valid for finding the element Z in the list Y.
*/
% ===========================================================================

position(_,[],0).

position(X,[X|_],1).

position(X,[W|Y],N) :-
 X \== W,
 position(X,Y,N1),
 ((N1 \== 0,
   N is N1 + 1)
 ;
  (N1 = 0,
   N is 0)
 ).


% ===========================================================================
/*
PREDICATE: dif(+S,?D,+N)
ARGUMENTS: S, integer
           D,  integer
           N,  integer
COMMENTS:  Succeeds after instantiating D to the numbers whose difference with
           S is N, either upper or lower.
*/
% ===========================================================================

dif(S,D,N) :-
 D is S + N.

dif(S,D,N) :-
 D is S - N.


% ===========================================================================
/*
PREDICATE: abs(+S,?N)
ARGUMENTS: S, integer
           N,  integer
COMMENTS:  Succeeds after instantiating N to the absolute value of S
*/
% ===========================================================================
```

```
abs(S,S) :-
 S > 0,
 !.

abs(S,N) :-
 N is 0 - S.


% ===========================================================================
/*
PREDICATE: between(+D,+First,+Last)
ARGUMENTS: D,      integer
           First, integer
           Last,  integer
COMMENTS:  Succeeds if D is between First and Last (First <= Last) or it is
           one of them
*/
% ===========================================================================

between(D,First,Last) :-
 D >= First,
 D =< Last.


% ===========================================================================
/*
PREDICATE: putinpos(+Value,+Place,+State,?State1)
ARGUMENTS: Value,  integer
           Place,  integer
           State,  list
           State1, list
COMMENTS:  Succeeds after instantiating State1 to the list result of setting the
           content of position 'Place' in State to 'Value'
*/
% ===========================================================================

putinpos(N,1,[_|Y],[N|Y]) :-
 !.

putinpos(N,M,[X|Y],[X|Z]) :-
 M1 is M - 1,
 moveaux(N,M1,Y,Z).


% ===========================================================================
/*
PREDICATE: reverse(+X,?Y)
ARGUMENTS: X, list
           Y, list
COMMENTS:  Succeeds after instantiating Y to the reversed X list.
           (efficient version, using reverse/3)
*/
% ===========================================================================

reverse(L,R) :-
 reverse(L,[],R).

reverse([],X,X).

reverse([X|Y],Sofar,Z) :-
 reverse(Y,[X|Sofar],Z).


% ===========================================================================
/*
PREDICATE: intersection(+X,+Y,?Z)
ARGUMENTS: X, list
           Y, list
           Z, list
COMMENTS:  Succeeds after instantiating Z to the intersection of
           X and Y, if both of them are sets
*/
% ==============================================================================

intersection(X,Y,Z) :-
 length(X,LX),
 length(Y,LY),
 LY < LX,
 !,
 intersection(Y,X,Z).

intersection([],_,[]).

intersection([E1|E2],Set,[E1|Rest]) :-
 memberchk(E1,Set),
 !,
 intersection(E2,Set,Rest).

intersection([_|E2],Set,I) :-
 intersection(E2,Set,I).


% ==============================================================================
/*
PREDICATE: union(+X,+Y,?Z)
ARGUMENTS: X, list
           Y, list
           Z, list
COMMENTS:  Succeeds after instantiating Z to the union set of
           X and Y, if both of them are sets
*/
% ==============================================================================

union(X,Y,Z) :-
 conc(X,Y,XY),
 setof2(I,member(I,XY),Z).


% ==============================================================================
/*
PREDICATE: conc(+X,+Y,?Z)
ARGUMENTS: X, list
           Y, list
           Z, list
COMMENTS:  Succeeds when Z is the concatenation of X and Y
*/
% ==============================================================================

conc([],L,L).

conc([X|L1],L2,[X|L3]) :-
 conc(L1,L2,L3).


% ==============================================================================
/*
PREDICATE: drawlist(+L1)
ARGUMENTS: L1, list
```

```
COMMENTS:  Succeeds after writing the list L1, one member a line
*/
% ==============================================================================

drawlist([]) :-
 nl.

drawlist([H|T]) :-
 nl,
 write(H),
 drawlist(T).


% ==============================================================================
/*
PREDICATE: setof2(+X,+Y,?Z)
ARGUMENTS: X, anything
           Y, predicate
           Z, list
COMMENTS:  like 'setof', but in the cases where 'setof' would fail,
           'setof2' instantiates Z to []
*/
% ==============================================================================

setof2(X,Y,Z) :-
 setof(X,Y,Z),
 !.

setof2(_,_,[]).


% ==============================================================================
/*
PREDICATE: bagof2(+X,+Y,?Z)
ARGUMENTS: X, anything
           Y, predicate
           Z, list
COMMENTS:  like 'bagof', but in the cases where 'setof' would fail,
           'bagof2' instantiates Z to []
*/
% ==============================================================================

bagof2(X,Y,Z) :-
 bagof(X,Y,Z),
 !.

bagof2(_,_,[]).


% ==============================================================================
/*
PREDICATE: sum(+X,+Y,?Z)
ARGUMENTS: X, list
           Y, list
           Z, list
COMMENTS:  Succeeds after instantiating Z to the list whose elements are sums
           of elements in X and the corresponding elements in Y
*/
% ==============================================================================

sum([],[],[]).

sum([H1|T1],[H2|T2],[H3|T3]) :-
 H3 is H1 + H2,
 sum(T1,T2,T3).


% ===========================================================================
/*
PREDICATE: times(+Item,+Bag,?List)
ARGUMENTS: Item, anything
           Bag,  list
           List, list
COMMENTS:  Succeeds after instantiating List to the list of "lists
           whose first element is Item and whose second
           element is the number of times it appears in Bag"
*/
% ===========================================================================

times(_,[],0).

times(H,[H|T],N) :-
 !,
 times(H,T,N1),
 N is N1 + 1.

times(H,[_|T],N) :-
 times(H,T,N).


% ===========================================================================
/*
PREDICATE: writelist(+List)
ARGUMENTS: List, list
COMMENTS:  Succeeds after writing each element in List in a
           different line
*/
% ===========================================================================

writelist([]).

writelist([H|T]) :-
 write(H),
 nl,
 writelist(T).


% ===========================================================================
/*
PREDICATE: spaces(+N,?Spaceslist)
ARGUMENTS: N,      integer
           Spaces, list
COMMENTS:  Succeeds after instantiating Spaces to a list of N 'space' ASCII
           numbers
*/
% ===========================================================================

spaces(0,[]) :-
 !.

spaces(N,[32|Spaces]) :-
 N1 is N - 1,
 spaces(N1,Spaces).


% ===========================================================================
/*
PREDICATE: zero(+N,?List)
ARGUMENTS: N, integer
           List, list
COMMENTS:  Succeeds after instantiating List to a list of N '0's
*/
% ===========================================================================

zero(0,[]) :-
 !.

zero(N,[0|X]) :-
 N1 is N-1,
 zero(N1,X).


% ===========================================================================
/*
PREDICATE: allmembers(+List,+Set)
ARGUMENTS: List, list
           Set,  list
COMMENTS:  Succeeds if all members of List are in Set
*/
% ===========================================================================

allmembers([],_).

allmembers([H|T],Set) :-
 member(H,Set),
 allmembers(T,Set).
```

Appendix D

System Files: newtimetable.pl

This appendix contains the text of the main program of the system. The listing starts in next page, in a format of two pages in one, with page numbers starting with 1.

```
/*
File:    newtimetable.pl
Author:  Luis Montero, MSc Student (lmg@forth, lmg@aipna)
Purpose: A Timetable Design Support System using ATMS.

GENERAL OVERVIEW:
This is the main Prolog code file for the TIMETABLE DESIGN SUPPORT SYSTEM
using ATMS. The predicates have been grouped together is SECTIONS, in order
to facilitate reading by people diferent from the author. Those sections are:

- HELP MENUS
- ATMS DATABASE
- DEBUG-TREE OPTIONS
- SOLVE OPTION
- SOLVE OPTION: FIXED LECTURES
- SOLVE OPTION: NON-FIXED LECTURES
- TESTS
- ATMS INFORMATION AND HISTORY
- SAVE AND LOAD OPTIONS
- RESOLVE OPTION
- DELETE OPTION
- ADD OPTION
- MOVE OPTION
- SHOW TIMETABLES
- GRAPHIC INTERFACE

*/

% ****************************************************************************
% *                                                                          *
% *                              HELP MENUS                                  *
% *                                                                          *
% ****************************************************************************


% ============================================================================
/*
PREDICATE: help_tree
ARGUMENTS: NONE
COMMENTS:  Succeeds after showing information about the environment and
           options that can be used to process the debug-tree
*/
% ============================================================================

help_tree :-
 nl,
 write('The history of the solution found after "solve" or "resolve" commands'),
 nl,
 write(' (the proof), including failures and solutions, is kept in a tree.'),
 nl,
 write('There is an active tree. The default one is the "solve" one (tree)'),
 nl,
 write('You can activate a "resolve" subtree, looking for its "SolNode"'),
 nl,
 write('solution node, in the "history", and doing "puttree(SolNode)"'),
 nl,
 nl,
 write('COMMANDS to process and see the tree'),
 nl,
 nl,
 write('snapshot(Filename). - to obtain a snapshot of the tree in Filename'),
 nl,
 nl,
 write('display.            - to see the actual level and children'),
 nl,
 nl,
 write('down(N).            - to move to the Nth child (N = Node number)'),
 nl,
 nl,
 write('up.                 - to move to the previous level'),
 nl,
 nl,
 write('top.                - to move to the top level'),
 nl,
 nl,
 write('solution.           - to move to the solution leaf'),
 nl.


% ============================================================================
/*
PREDICATE: help_cons
ARGUMENTS: NONE
COMMENTS:  Succeeds after showing information about the kind of constraints
           that can be used in the system, and how they should appear in the
           constraints file
*/
% ============================================================================

help_cons :-
 nl,
 write('The following Kinds of facts are used in a constraints file:'),
 nl,
 nl,
 write('subjects    - fixed first. Other subjects shhould be ordered'),
 nl,
 write('              most constrained first'),
 nl,
 nl,
 write('subjlectures  - number of lectures per subject, if not fixed'),
 nl,
 nl,
 write('The following constraints must include a "constraint assumption"'),
 nl,
 write('number, unless the opposite is said. Numbers must be unique'),
 nl,
 nl,
 write('nonsimult   - subjects that cannot have lectures at the same time'),
 nl,
 write('nonfollow   - subjects that cannot have subsequent lectures (e.g.:'),
 nl,
 write('              lectures at KB/MAIN CAMPUS taken by the same person)'),
 nl,
 write('lectroom    - pairs subject-room suitable for it'),
 nl,
 write('fix         - fixed times per subject, if fixed lectures'),
 nl,
 write('bad,        - "bad" or "very bad" times for each subject, and for'),
 nl,
 write('verybad       "all" subjects in general. NO CONSTRAINT ASSUMPTION'),
 nl,
 write('              NUMBER USED'),
 nl,
 write('notpos      - "not possible" times for each subject, and "all"'),
 nl,
 nl,
 write('OTHER times than bad, verybad or notpos, are "good"'),
 nl,
 nl,
 write('For more info, edit the file "realconstraints.txt"'),
 nl.


% ==========================================================================
/*
PREDICATE: help_user
ARGUMENTS: NONE
COMMENTS:  Succeeds after telling the user some information about
           history-information options
*/
% ==========================================================================

help_user :-
 nl,
 write('If you want INFORMATION about the environment, you have some options:'),
 nl,
 nl,
 write('user_info - will show you the unsatisfied constraint assumptions, and'),
 nl,
 write('            nodes which make them fail'),
 nl,
 write('            and "system advices" about which lectures to remove to'),
 nl,
 write('            solve the problem, if there is, in fact, any failure'),
 nl,
 nl,
 write('showtimetable - shows the actual timetable'),
 nl,
 nl,
 write('history_info - will show the history of the session, from the first'),
 nl,
 write('            "solve" up to now. Solution nodes numbers after every'),
 nl,
 write('            change are shown. so that you can use them in order'),
 nl,
 write('            to see past timetables or consult resolve "subtrees"'),
 nl,
 nl,
 write('showtimetable(SolNode) - Shows the past timetable coresponding to'),
 nl,
 write('            Solnode solution node'),
 nl,
 nl,
 write('snapshottimetables(Filename) - creates a snapshot of the history'),
 nl,
 write('            of the session, together with the history of the'),
 nl,
 write('            timetables'),
 nl,
 nl,
 write('numbers - to see the meanings of the numbers in the history or tree'),
 nl,
 nl,
 nl.


% ==========================================================================
/*
PREDICATE: help_add_delete
ARGUMENTS: NONE
COMMENTS:  Succeeds after telling the user some information about
           add-delete and move options
*/
% ==========================================================================

help_add_delete :-
 nl,
 write('If you want to PERFORM CHANGES IN the TIMETABLE, use "move"'),
 nl,
 nl,
 write('move   - Allows you to change a lecture in Day, Hour and/or Room,'),
 nl,
 nl,
 nl,
 write('If you want to PERFORM CHANGES IN CONSTRAINTS, use "add" or "delete":'),
 nl,
 nl,
 write('You can add and delete:'),
 nl,
 nl,
 write('NATURAL CHANGES:'),
 nl,
 write('"Subject"'),
 nl,
 write('"bad" constraints'),
 nl,
 write('"verybad" constraints'),
 nl,
 write('"notpos" constraints'),
 nl,
 write('"nonsimult" constraints'),
 nl,
 write('"nonfollow" constraints'),
 nl,
 write('"lectroom" constraints'),
 nl,
 nl,
 write('If such changes are possible. The only side effect they produce is,'),
 nl,
 write('eventually, to change the ATMS environment (desirable side effect,'),
 nl,
 write('of course), so that the state of the problem may pass from "solved"'),
 nl,
 write('to "unsolved" or viceversa.'),
 nl,
 nl,
 write('CHANGES TO ALTER THE STATUS OF LECTURES FOR A SUBJECT'),
 nl,
 write('"Subjlectures"'),
 nl,
 write(' depending on the previous status, either "subjlectures" or ALL "fix"'),
 nl,
 write(' constraints of the subject entered are deleted (if any), and ALL'),
 nl,
 write(' lectures are removed (they lose their priority). In any case,'),
 nl,
 write(' the timetable is updated, according to the number of lectures'),
 nl,
 write(' required, and a new "subjlectures" constraint is created'),
 nl,
 nl,
 write('"fix"'),
 nl,
 write(' depending on the previous status, either "subjlectures" or ALL "fix"'),
 nl,
 write(' constraints of the subject entered are deleted (if any), and ALL'),
 nl,
 write(' lectures are removed, and replaced by the'),
 nl,
 write(' new entered ones, as new "fix" constraints are created'),
 nl,
 nl,
 write(' use "add" and "delete" commands for more details'),
 nl.



% ***************************************************************************
% *                                                                         *
% *                              ATMS DATABASE                              *
% *                                                                         *
% ***************************************************************************


% ===========================================================================
/*
PREDICATE: putsetup(+N)
ARGUMENTS: N, integer
COMMENTS:  Succeeds after setting up atms for N assumptions
*/
% ===========================================================================

putsetup(N) :-
 atms_setup(N,_).


% ===========================================================================
/*
PREDICATE: putnode(+Node)
ARGUMENTS: Node, integer (node number)
COMMENTS:  Succeeds after sending Node to ATMS
           and keeping track in 'nodenumber'
*/
% ===========================================================================

putnode(Node) :-
 atms_node(Node),
 asserta(nodenumber(Node)).


% ===========================================================================
/*
PREDICATE: puttreenode(+Node)
ARGUMENTS: Node, integer (node number)
COMMENTS:  Succeeds after sending Node to ATMS
           and keeping track in 'treenode'
*/
% ===========================================================================

puttreenode(Node) :-
 atms_node(Node),
 asserta(treenode(Node)).


% ===========================================================================
/*
PREDICATE: putassumpt(+As)
ARGUMENTS: As, integer (assumption number)
COMMENTS:  Succeeds after sending As to ATMS
           and keeping track in 'assumptnumber'
*/
% ===========================================================================

putassumpt(As) :-
 atms_assumption(As),
 asserta(assumptnumber(As)).


% ===========================================================================
/*
PREDICATE: putconstraint(+As)
ARGUMENTS: As, integer (assumption number)
COMMENTS:  Succeeds after sending As to ATMS
           and keeping track in 'constraintnumber'
*/
% ===========================================================================

putconstraint(As) :-
 atms_assumption(As),
 asserta(constraintnumber(As)).


% ===========================================================================
/*
PREDICATE: putjustification(+Node,+List)
ARGUMENTS: Node, integer (node number)
COMMENTS:  Succeeds after sending the justification to ATMS
           and keeping track in 'justification'. The option of doing so only
           if such justification was not yet in the ATMS database, was
           removed for efficiency reasons, but, if used, would need the
           rule shown above.

% ===========================================================================

putjustification(Node,List) :-
 justification([Node|List]),
 !.

*/

putjustification(Node,List) :-
 asserta(justification([Node|List])),
 atms_justification(Node,List).


% ===========================================================================
/*
PREDICATE: sendjusttoatms(+L)
ARGUMENTS: L, list (of assumptions)
COMMENTS:  Succeeds after sending all justifications representations in L
           to ATMS
*/
% ===========================================================================

sendjusttoatms([]).

sendjusttoatms([[Node|Just]|RT]) :-
 atms_justification(Node,Just),
 sendjusttoatms(RT).


% ===========================================================================
/*
```

```
PREDICATE: sendnodestoatms(+L)
ARGUMENTS: L, list (of assumptions)
COMMENTS:  Succeeds after sending all assumptions in L to ATMS
*/
% ======================================================================

sendnodestoatms([]).

sendnodestoatms([RH|RT]) :-
 atms_node(RH),
 sendnodestoatms(RT).


% ======================================================================
/*
PREDICATE: sendastoatms(+L)
ARGUMENTS: L, list (of assumptions)
COMMENTS:  Succeeds after sending all assumptions in L to ATMS
*/
% ======================================================================

sendastoatms([]).

sendastoatms([RH|RT]) :-
 atms_assumption(RH),
 sendastoatms(RT).


% ======================================================================
/*
PREDICATE: sendconstoatms(+L)
ARGUMENTS: L, list (of assumptions)
COMMENTS:  Succeeds after sending all assumptions in L to ATMS
           and keeping track in 'constraintnumber'
*/
% ======================================================================

sendconstoatms([]).

sendconstoatms([RH|RT]) :-
 putconstraint(RH),
 sendconstoatms(RT).


% ======================================================================
/*
PREDICATE: sendconstraints
ARGUMENTS: NONE
COMMENTS:  Succeeds after sending all initial constraints to ATMS Database
*/
% ======================================================================

sendconstraints :-
 setof2(N,initconstraint(N),Aslist),
 sendconstoatms(Aslist).


% ======================================================================
/*
PREDICATE: findconstraint(+N,?Pred,?Arglist)
ARGUMENTS: N,       integer
           Pred,    atom (predicate name)
           Arglsit, list (of Pred arguments)
COMMENTS:  Succeeds after finding the predicate name and arguments
           corresponding to the constraint whose asumption number is N
*/
% ======================================================================

findconstraint(N,nonsharedrooms,[N]) :-
 nonsharedrooms(N).

findconstraint(N,nonsimult,[Arg1,Arg2,N]) :-
 nonsimult(Arg1,Arg2,N).

findconstraint(N,nonfollow,[Arg1,Arg2,N]) :-
 nonfollow(Arg1,Arg2,N).

findconstraint(N,lectroom,[Arg1,Arg2,N]) :-
 lectroom(Arg1,Arg2,N).

findconstraint(N,fix,[Arg1,Arg2,N]) :-
 fix(Arg1,Arg2,N).

findconstraint(N,notpos,[Arg1,Arg2,N]) :-
 notpos(Arg1,Arg2,N).


% ======================================================================
/*
PREDICATE: initconstraint(?N)
ARGUMENTS: N, integer
COMMENTS:  Succeeds if N is the number of any constraint
*/
% ======================================================================

initconstraint(N) :-
 findconstraint(N,_,_).


% ======================================================================
/*
PREDICATE: constraints(?Aslist)
ARGUMENTS: Aslist, List of Assumption numbers
COMMENTS:  Succeeds after instantiating Aslist to the list of existing
           constraints in the environment
*/
% ======================================================================

constraints(AsList) :-
 setof2(N,constraintnumber(N),AsList).


% ======================================================================
/*
PREDICATE: db(+A,+Node)
ARGUMENTS: A,     noderep [Node,Subject,Day,Hour [,Room] ]
           Node, integer (node number), the same Node in A
COMMENTS:  Tests is such node exists in database (in this case, Assert Node
           to such node number; otherwise, it creates a new one. In this
           case, the eventual inconsistencies with existing nodes for
           same day, hour and room are sent to ATMS)
*/
% ======================================================================

db(A,_) :-
 node(A),
 !.
```

```
% creates a new one (noderep without room)

db(A,Node) :-
 A = [_,_,_,_],
 nodenumber(LastNode),
 !,
 Node is LastNode + 1,
 putnode(Node),
 asserta(node(A)).


% creates a new one (noderep with room: inconsistencies to ATMS via
% recsharedrooms2 predicate

db(A,Node) :-
 A = [_,_,Day,Hour,Room],
 nodenumber(LastNode),
 !,
 Node is LastNode + 1,
 putnode(Node),
 bagof2(X,nodeDayHour(X,Day,Hour,Room),Bag),
 recsharedrooms2(Node,Bag),
 asserta(node(A)).


% creates the first one

db(A,Node) :-
 firstnode(FirstNode),
 Node is FirstNode + 1,
 putnode(Node),
 asserta(node(A)).


% ============================================================================
/*
PREDICATE: recsharedrooms2(+Node,+Nodelist)
ARGUMENTS: Node,      node number
           Nodelist, list of nodes numbers
COMMENTS:  Succeeds after sending ATMS all inconsistencies between Node and
           elements in Nodelist: Nodes that hold another lecture at the
           same time, same room, which is impossible.
*/
% ============================================================================

recsharedrooms2(_,[]).

recsharedrooms2(Node,[Node|T]) :-
 !,
 recsharedrooms2(Node,T).

recsharedrooms2(Node,[H|T]) :-
 recsharedrooms3(Node,H),
 recsharedrooms2(Node,T).


% ============================================================================
/*
PREDICATE: recsharedrooms3(+Node1,+Node2)
ARGUMENTS: Node1, node number
           Node2, node number
COMMENTS:  Succeeds after sending ATMS the inconsistency [Node1,Node2,As]
           where As is the assumption meaning that a room cannot hold two
           different lectures at the same time (it is impossible).
*/
% ============================================================================

recsharedrooms3(Node1,Node2) :-
 nonsharedrooms(As),
 putjustification(0,[Node1,Node2,As]).


% ============================================================================
/*
PREDICATE: nodeDayHour(?Node,+Day,+Hour,+Room)
ARGUMENTS: Node, node number
           Day,  day representation
           Hour, hour representation
           Room, room representation
COMMENTS:  Succeeds after instantiating Node to the number of a node whose
           Day, Hour and Room coincide with the arguments
*/
% ============================================================================

nodeDayHour(X,Day,Hour,Room) :-
 node([X,_,Day,Hour,Room]).


% ============================================================================
/*
PREDICATE: dba(+A,+As)
ARGUMENTS: A,     noderep [As,Subject,Day,Hour]
           As,    integer (node number), the same As in A
COMMENTS:  Tests is such assumption exists in database (in this case, Asserts
           As to such assumption number; otherwise, it creates a new one.
*/
% ============================================================================

dba(A,_) :-
 assumpt(A),
 !.


% creates a new one

dba(A,As) :-
 A = [_,_,_,_],
 assumptnumber(LastAs),
 !,
 As is LastAs + 1,
 putassumpt(As),
 asserta(assumpt(A)).


% creates the first one

dba(A,As) :-
 firstas(FirstAs),
 As is FirstAs + 1,
 putassumpt(As),
 asserta(assumpt(A)).


% ============================================================================
/*
PREDICATE: dbtreenode(+TreeNode)
ARGUMENTS: TreeNode, integer (node number)
```

```
COMMENTS:  produces a new tree node
*/
% ==========================================================================

% creates a new one

dbtreenode(TreeNode) :-
 treenode(TN),
 !,
 TreeNode is TN + 1,
 puttreenode(TreeNode).


% creates the first one

dbtreenode(TreeNode) :-
 firsttreenode(TN),
 TreeNode is TN + 1,
 puttreenode(TreeNode).


% ==========================================================================
/*
PREDICATE: dbass(+Nodedb)
ARGUMENTS: Nodedb, noderep. In this case: [Node,Subject,Day,Hour]
COMMENTS:  Succeeds after creating a new assumption with the same content
           as Nodedb, which justifies it
*/
% ==========================================================================

dbass(Nodedb) :-
 Nodedb = [Node,Subject,Day,Hour],
 Asdb = [As,Subject,Day,Hour],
 dba(Asdb,As),
 putjustification(Node,[As]).


% ==========================================================================
/*
PREDICATE: dbnode(+Nodedb)
ARGUMENTS: Nodedb, noderep. In this case: [Node,Subject,Day,Hour,Room]
COMMENTS:  Succeeds after finding the corresponding [Node2,Subject,Day,Hour]
           noderep, the corresponding 'lectroom(Subject,Room,As)' constraint
           assumption, and sending the justification to ATMS
*/
% ==========================================================================

dbnode(Nodedb) :-
 Nodedb = [Node,Subject,Day,Hour,Room],
 Nodedb2 = [Node2,Subject,Day,Hour],
 node(Nodedb2),
 lectroom(Subject,Room,As),
 putjustification(Node,[Node2,As]).


% ==========================================================================
/*
PREDICATE: removeallnodes(+Subject)
ARGUMENTS: Subject, subject representation
COMMENTS:  Succeeds after removing all nodes that include Subject, and
           asserting corresponding "oldnodes"
*/
% ==========================================================================
```

```
removeallnodes(Subject) :-
 Nodedb = [_,Subject|_],
 retract(node(Nodedb)),
 !,
 asserta(oldnode(Nodedb)),
 removeallnodes(Subject).

removeallnodes(_).


% ==========================================================================
/*
PREDICATE: retractallnlectures
ARGUMENTS: NONE
COMMENTS:  Succeeds after removing all nlectures in the database
*/
% ==========================================================================

retractallnlectures :-
 retract(nlectures(_,_)),
 !,
 retractallnlectures.

retractallnlectures.


% ==========================================================================
/*
PREDICATE: retractallnodes
ARGUMENTS: NONE
COMMENTS:  Succeeds after removing all oldnodes in the database
*/
% ==========================================================================

retractalloldnodes :-
 retract(oldnode(_)),
 !,
 retractalloldnodes.

retractalloldnodes.


% ==========================================================================
/*
PREDICATE: retractallbad(+Subject)
ARGUMENTS: Subject, subject representation
COMMENTS:  Succeeds after removing all "bad"s in the database
           that include Subject
*/
% ==========================================================================

retractallbad(Subject) :-
 bad(Subject,_),
 !,
 retract(bad(Subject,_)),
 retractallbad(Subject).

retractallbad(_).


% ==========================================================================
/*
PREDICATE: retractallverybad(+Subject)
ARGUMENTS: Subject, subject representation
```

```
COMMENTS:  Succeeds after removing all "verybad"s in the database
           that include Subject
*/
% ============================================================================

retractallverybad(Subject) :-
 verybad(Subject,_),
 !,
 retract(verybad(Subject,_)),
 retractallverybad(Subject).

retractallverybad(_).


% ============================================================================
/*
PREDICATE: retractallnotpos(+Subject)
ARGUMENTS: Subject, subject representation
COMMENTS:  Succeeds after removing all "notpos"s in the database
           that include Subject
*/
% ============================================================================

retractallnotpos(Subject) :-
 notpos(Subject,_,As),
 !,
 asserta(deleted(As)),
 retract(constraintnumber(As)),
 retract(notpos(Subject,_,As)),
 retractallnotpos(Subject).

retractallnotpos(_).


% ============================================================================
/*
PREDICATE: retractallnonsimult(+Subject)
ARGUMENTS: Subject, subject representation
COMMENTS:  Succeeds after removing all "nonsimult"s in the database
           that include Subject
*/
% ============================================================================

retractallnonsimult(Subject) :-
 nonsimult(Subject1,Subject2,As),
 member(Subject,[Subject1,Subject2]),
 !,
 asserta(deleted(As)),
 retract(constraintnumber(As)),
 retract(nonsimult(Subject1,Subject2,As)),
 retractallnonsimult(Subject).

retractallnonsimult(_).


% ============================================================================
/*
PREDICATE: retractallnonfollow(+Subject)
ARGUMENTS: Subject, subject representation
COMMENTS:  Succeeds after removing all "nonfollow"s in the database
           that include Subject
*/
% ============================================================================

retractallnonfollow(Subject) :-
 nonfollow(Subject1,Subject2,As),
 member(Subject,[Subject1,Subject2]),
 !,
 asserta(deleted(As)),
 retract(constraintnumber(As)),
 retract(nonfollow(Subject1,Subject2,As)),
 retractallnonfollow(Subject).

retractallnonfollow(_).


% ============================================================================
/*
PREDICATE: retractallfix(+Subject)
ARGUMENTS: Subject, subject representation
COMMENTS:  Succeeds after removing all "fix"s in the database
           that include Subject
*/
% ============================================================================

retractallfix(Subject) :-
 fix(Subject,_,As),
 !,
 asserta(deleted(As)),
 retract(constraintnumber(As)),
 retract(fix(Subject,_,As)),
 retractallfix(Subject).

retractallfix(_).


% ============================================================================
/*
PREDICATE: retractallfixed
ARGUMENTS: Subject, subject representation
COMMENTS:  Succeeds after removing all "fixed"s in the database
           that include Subject
*/
% ============================================================================

retractallfixed :-
 retract(fixed(_,_)),
 !,
 retractallfixed.

retractallfixed.


% ============================================================================
/*
PREDICATE: retractalllectroom(+Subject)
ARGUMENTS: Subject, subject representation
COMMENTS:  Succeeds after removing all "lectroom"s in the database
           that include Subject
*/
% ============================================================================

retractalllectroom(Subject) :-
 lectroom(Subject,_,As),
 !,
 asserta(deleted(As)),
 retract(constraintnumber(As)),
 retract(lectroom(Subject,_,As)),
 retractalllectroom(Subject).

retractalllectroom(_).


% ******************************************************************************
% *                                                                            *
% *                            DEBUG-TREE OPTIONS                              *
% *                                                                            *
% ******************************************************************************


% ==============================================================================
/*
PREDICATE: snapshot(+Filename)
ARGUMENTS: Filename, atom
COMMENTS:  Succeds after producing a snapshot of the solution tree in the file
           Filename
*/
% ==============================================================================

snapshot(Filename) :-
 tell(Filename),
 ((activetree(tree),
   tree(Tree))
 ;
  (activetree(In),
   subtree(In,Tree))
 ),
 drawtree(Tree,0),
 told.


% ==============================================================================
/*
PREDICATE: drawtree(+T,+N)
ARGUMENTS: T, list (tree)
           N, integer
COMMENTS:  Succeeds after drawing a repreesentation of T , where N is the
           left margin
*/
% ==============================================================================

drawtree([],_) :-
 !,
 nl.

drawtree(A,N) :-
 A = [Node|_],
 integer(Node),
 !,
 spaces(N,Spaceslist),
 name(Spaces,Spaceslist),
 write(Spaces),
 write(A),
 nl.

drawtree([fail|T],N) :-
 !,
 spaces(N,Spaceslist),
 name(Spaces,Spaceslist),
 write(Spaces),
 write('fail: '),
```

```
 write(T),
 nl.

drawtree([solution|T],N) :-
 !,
 spaces(N,Spaceslist),
 name(Spaces,Spaceslist),
 write(Spaces),
 write('SOLUTION!: '),
 write(T),
 nl.

drawtree([H|T],N) :-
 !,
 N1 is N + 1,
 drawtree(H,N1),
 drawtree(T,N).

drawtree(_,_) :-
 write('DRAW PROBLEMS'),
 nl.


% ==============================================================================
/*
PREDICATE: display
ARGUMENTS: NONE
COMMENTS:  Succeeds after showing the top two levels in the actual position
           of the solution tree
*/
% ==============================================================================

display :-
 nl,
 now(L),
 ((activetree(tree),
  tree(Tree))
 ;
  (activetree(In),
   subtree(In,Tree))
 ),
 gotolevel(Tree,L,[H|T]),
 write(' '),
 write(H),
 nl,
 disp2(T),
 nl.


% ==============================================================================
/*
PREDICATE: gotolevel(+Tree,+List,?LastTree)
ARGUMENTS: Tree,     list
           List,     list
           LastTree, list
COMMENTS:  Suceeds after instantiating LastTree to the tree obtained going
           down across Tree, following List path
*/
% ==============================================================================

gotolevel(Tree,[],Tree).

gotolevel([_|TT],[LH|LT],LastTree) :-
 loc(LH,TT,NewTree),
 gotolevel(NewTree,LT,LastTree).

% ============================================================================
/*
PREDICATE: loc(+LH,+Children,?Newtree)
ARGUMENTS: LH,        integer (node)
           Children,  list (of 'brother trees')
           Newtree,   list (tree)
COMMENTS:  Succeeds after instantiating 'Newtree' to the tree from 'Children'
           list whose node number is LH
*/
% ============================================================================

loc(LH,[TH|_],TH) :-
 TH = [[LH|_]|_],
 !.

loc(LH,[_|TT],TH) :-
 loc(LH,TT,TH).


% ============================================================================
/*
PREDICATE: disp2(+Children)
ARGUMENTS: Children, list of brother trees or 'fail/solution' terminal node
COMMENTS:  Succeeds after writing the first level of each item in Children
*/
% ============================================================================

disp2([]).

disp2([H|T]) :-
 write('  '),
 disp3(H),
 nl,
 disp2(T).


% ============================================================================
/*
PREDICATE: disp3(+Item)
ARGUMENTS: Item, list of brother trees or 'fail/solution' terminal node
COMMENTS:  Succeeds after writing the first level of each item in Children
           (disp2 predicate)
*/
% ============================================================================

disp3([fail|T]) :-
 !,
 write('fail: '),
 write(T).

disp3([solution|T]) :-
 !,
 write('SOLUTION!: '),
 write(T).

disp3([H|_]) :-
 write(H).


% ============================================================================
/*
```

```
PREDICATE: terminal(+Tree)
ARGUMENTS: +Tree          list (Tree representation - leaf)
COMMENTS:  Succeeds if Tree is a 'fail' or 'solution' node, fails otherwise
*/
% ============================================================================

terminal([solution|_]).

terminal([fail|_]).


% ============================================================================
/*
PREDICATE: down(+N)
ARGUMENTS: N, integer
COMMENTS:  Succeeds after setting the position in the solution tree to N
           (changing 'now' value), if N is one of the children in the
           previous position
*/
% ============================================================================

down(N) :-
 nl,
 now(L),
 ((activetree(tree),
  tree(Tree))
 ;
  (activetree(In),
   subtree(In,Tree))
 ),
 gotolevel(Tree,L,[_|T]),
 loc(N,T,_),
 now(A),
 conc(A,[N],A1),
 retract(now(A)),
 asserta(now(A1)).


% ============================================================================
/*
PREDICATE: up
ARGUMENTS: NONE
COMMENTS:  Succeeds after jumping up a position in the solution tree
*/
% ============================================================================

up :-
 now(A),
 reverse(A,[_|T]),
 reverse(T,A1),
 retract(now(A)),
 asserta(now(A1)).


% ============================================================================
/*
PREDICATE: top
ARGUMENTS: NONE
COMMENTS:  Succeeds after jumping to the top of the solution tree
           (setting 'now' to []).
*/
% ============================================================================

top :-
```

```
 retract(now(_)),
 asserta(now([])).


% ===========================================================================
/*
PREDICATE: solution
ARGUMENTS: NONE
COMMENTS:  Succeeds after jumping to the total or partial solution
*/
% ===========================================================================

solution :-
 ((activetree(tree),
   treesol(L))
  ;
  (activetree(In),
   subsol(In,L))
 ),
 retract(now(_)),
 asserta(now(L)).


% ===========================================================================
/*
PREDICATE: puttree(+In)
ARGUMENTS: In, 'tree' or integer
COMMENTS:  Succeeds after setting a new active tree for exploration
*/
% ===========================================================================

puttree(In) :-
 retract(activetree(_)),
 retract(now(_)),
 asserta(activetree(In)),
 asserta(now([])).


% ***************************************************************************
% *                                                                         *
% *                             SOLVE OPTION                                *
% *                                                                         *
% ***************************************************************************


% ===========================================================================
/*
PREDICATE: solve
ARGUMENTS: NONE
COMMENTS:  Succeeds after finding a solution for the timetable problem,
           according to the set of constraints. The solution is then
           asserted, as well as the search tree, in order to be seen by
           the user later. The possible failure in the search (backtrack
           option used) is asserted,
           as well, in order to let the system know that there is no solution
           (in this case, the longest partial solution has been asserted
           as 'partial', and the corresponding branch in the Tree, as 'part')
*/
% ===========================================================================


solve :-
 notallowed(solve),
 !,
 write('option not available now'),
 nl,
 fail.

solve :-
 subjects([]),
 !,
 write('no subjects to set'),
 nl,
 fail.

solve :-
 solvestart,
 set(Tree,Timetable,Fail),
 asserta(notallowed(solve)),
 retractallnlectures,
 retractallfixed,
 asserta(tree(Tree)),
 ((Fail = yes,
   !,
   takepart(L),
   asserta(treesol(L)),
   reverse(L,[Solnode|_]),
   asserta(solnode(Solnode)),
   retract(timetable(_)),
   asserta(timetable(PartTimetable)),
   findinconsist(BadAs),
   retract(unsolved(_)),
   asserta(unsolved(BadAs)),
   retract(tracks(_)),
   setof2(X,findtracks(BadAs,X),Tracks),
   asserta(tracks(Tracks)),
   historyTimetable(backtrack,Solnode,PartTimetable),
   showtimetable,
   write('UNSATISFIABLE CONSTRAINTS. I found a partial solution. Change the'),
   nl,
   write('constraints, and repair it. You should EXIT the session now'),
   nl,
   user_info)
  ;
  (retract(notallowed(resolve)),
   sol(L),
   retract(sol(L)),
   asserta(treesol(L)),
   reverse(L,[Solnode|_]),
   asserta(solnode(Solnode)),
   retract(timetable(_)),
   asserta(timetable(Timetable)),
   findinconsist(BadAs),
   retract(unsolved(_)),
   asserta(unsolved(BadAs)),
   retract(tracks(_)),
   setof2(X,findtracks(BadAs,X),Tracks),
   asserta(tracks(Tracks)),
   showtimetable,
   user_info)
 ),
 ((part(_),
   retract(part(_)))
  ;
  (true)
 ),
 asserta(history([[solve,Solnode,tree,Fail,BadAs]])),
 asserta(historyTimetable(solve,Solnode,Timetable)),
 asserta(activetree(tree)),
 asserta(now([])).


% ============================================================================
/*
PREDICATE: recnlectures
ARGUMENTS: List, list
COMMENTS:  Succeeds after asserting all 'nlectures' predicates
           corresponding to List.
*/
% ============================================================================

recnlectures([]).

recnlectures([[Subject,N]|List]) :-
 asserta(nlectures(Subject,N)),
 recnlectures(List).


% ============================================================================
/*
PREDICATE: solvestart
ARGUMENTS: NONE
COMMENTS:  Succeeds after asserting initial values in order to start "solve"
           option
*/
% ============================================================================

solvestart :-
 sendconstraints,
 subjects(Subjects),
 createfixed(Subjects),
 createlectrooms(Subjects),
 bagof2(X1,subjlecturesarglist(X1),List1),
 recnlectures(List1).


% ============================================================================
/*
PREDICATE: createtranslate(+Days,+N)
ARGUMENTS: Days, list (of days representations)
           N,    integer
COMMENTS:  Succeeds after asserting translate(Day,Nday), for all Day,
           member of Days, where Nday is the position that it holds in Days.
*/
% ============================================================================

createtranslate([],_).

createtranslate([H|T],N) :-
 asserta(translate(H,N)),
 N1 is N + 1,
 createtranslate(T,N1).


% ============================================================================
/*
PREDICATE: join(+Days,+Hours,?Dayshours)
ARGUMENTS: Days,      list (of days representations)
           Hours,     list (of hours representations)
           DaysHours, list (of pairs [day,hour])
COMMENTS:  Succeeds after instantiating DaysHours to the list of all pairs
           [Day,Hour] such that Day is in Days and Hour is in Hours
*/
% ============================================================================

join(_,[],[]).

join(L1,[H2|T2],L3) :-
 subjoin(L1,H2,H3),
 join(L1,T2,T3),
 conc(H3,T3,L3).


% ============================================================================
/*
PREDICATE: subjoin(+Days,+Hour,?Dayshour)
ARGUMENTS: Days,     list (of days representations)
           Hour,     hour representations
           DaysHour, list (of pairs [day,hour])
COMMENTS:  Succeeds after instantiating DaysHour to the list of all pairs
           [Day,Hour] such that Day is in Days
*/
% ============================================================================

subjoin([],_,[]).

subjoin([H1|T1],L2,[[H1,L2]|T3]) :-
 subjoin(T1,L2,T3).


% ============================================================================
/*
PREDICATE: createlectrooms(+Subjects)
ARGUMENTS: Subjects, list (of Subjects representations)
COMMENTS:  Succeeds after asserting lectrooms(Subject,Rooms), for all Subject
           member of Subjects, where Rooms is the set of Room elements such
           that 'lectroom(Subject,Room,_)'
*/
% ============================================================================

createlectrooms([]).

createlectrooms([HSubjects|TSubjects]) :-
 setof2(X,lectr(HSubjects,X),Rooms),
 asserta(lectrooms(HSubjects,Rooms)),
 createlectrooms(TSubjects).


% ============================================================================
/*
PREDICATE: lectr(?X,?Y)
ARGUMENTS: X, Subject representation
           Y, room representation
COMMENTS:  Succeeds if there exists a Z such that lectroom(X,Y,Z)
*/
% ============================================================================

lectr(X,Y) :-
 lectroom(X,Y,_).


% ============================================================================
/*
PREDICATE: createfixed(+Subjects)
ARGUMENTS: Subjects, list (of Subjects representations)
```

```
COMMENTS:  Succeeds after asserting fixed(Subject,Times), for all Subject
           member of Subjects, that have fixed times (Times is the set of
           Time elements such that 'fix(Subject,Time,_)', while CHECKING
           That fixed lectures are first, and all other are affected by
           "subjlectures"
*/
% ============================================================================

createfixed([]) :-
 !.

createfixed([HSubjects|TSubjects]) :-
 setof(X,fx(HSubjects,X),Times),
 !,
 asserta(fixed(HSubjects,Times)),
 createfixed(TSubjects).

createfixed(Subjects) :-
 createfixed2(Subjects).


% ============================================================================
/*
PREDICATE: createfixed2(+Subjects)
ARGUMENTS: Subjects, list (of Subjects representations)
COMMENTS:  Succeeds after performing the TEST shown in createfixed predicate
           above
*/
% ============================================================================

createfixed2([]).

createfixed2([HSubjects|TSubjects]) :-
 subjlectures(HSubjects,N),
 integer(N),
 N >= 0,
 !,
 createfixed2(TSubjects).

createfixed2([HSubjects|_]) :-
 setof(X,fx(HSubjects,X),_),
 !,
 write('WARNING: Subjects list not well ordered. Look at '),
 write(HSubjects),
 nl,
 fail.

createfixed2([HSubjects|_]) :-
 write('WARNING: Subject without number of lectures to set. Look at '),
 write(HSubjects),
 nl,
 fail.


% ============================================================================
/*
PREDICATE: fx(?X,?Y)
ARGUMENTS: X, Subject representation
           Y, room representation
COMMENTS:  Succeeds if there exists a Z such that fix(X,Y,Z)
*/
% ============================================================================

fx(X,Y) :-
```

```
 fix(X,Y,_).


% ============================================================================
/*
PREDICATE: set(?Tree,?Timetable,?Fail)
ARGUMENTS: Tree,      list (tree representation)
           Timetable, list (of nodereps)
           Fail,      yes or no
COMMENTS:  Succeeds after instantiating Timetable to the timetable solution,
           Tree to its search tree, and Fail to 'yes' or 'no', depending
           on the existance of a solution (see 'solve' comments)
*/
% ============================================================================

set([[In,tree]|Brothers],Timetable,Fail) :-
 firsttreenode(In),
 subjects([H|T]),
 settimetable(H,T,[],[],Timetable,Brothers,Fail).


% ============================================================================
/*
PREDICATE: settimetable(+Subject,+Restsubjects,+Nodelist,+Timetable,
                        ?Newtimetable,?Brothers,?Fail)
ARGUMENTS: Subject,      subject representation
           Restsubjects, list (of subjects representations)
           Nodelist,     list (of integers - nodes numbers)
           Timetable,    list (of nodereps)
           Newtimetable, list (of nodereps)
           Brothers,     list (of trees representations)
           Fail,         yes or no
COMMENTS:  Succeeds after instantiating Newtimetable to the timetable solution,
           corresponding to Subject (and Restsubjects, using recursion),
           depending on the actual state of Timetable (according to the
           previous subjects. Brothers will hold the list of trees
           corresponding to Subject-Restsubjects, and Fail is 'yes' or 'no',
           depending on the existance of a solution. Nodelist, input value,
           holds the path, from the top, up to our actual position in the
           tree search. It is necessary to set dependencies between nodes
           and asuumptions, and to assert the final solution (or partial).

           There are two cases for settimetable:
           - lectures FIXED in time (Therefore, Brothers can only contain
             one Tree: if a fixed lecture is not possible, there is no
             possible backtracking, but failure is reported at once)
           - lectures NOT FIXED in time. Then, an agenda of suitable times,
             ordered by priority (preferences) is used for best-first-search.
*/
% ============================================================================

settimetable(Subject,Restsubjects,Nodelist,Timetable,
             Newtimetable,[Tree],Fail) :-
 fixed(Subject,Times),
 !,
 write(Subject),
 write(' in process ...'),
 nl,
 fixedinsert(Subject,Restsubjects,Times,Nodelist,Timetable,
             Newtimetable,[Tree],Fail),
 !.

settimetable(Subject,Restsubjects,Nodelist,Timetable,
             Newtimetable,Brothers,Fail) :-
 write(Subject),
 write(' in process ...'),
 nl,
 dayshours(Dayshours),
 setof2(X,dayshourspri(Subject,Dayshours,X),Agenda),
 length(Agenda,L),
 zero(L,Pribefore),
 nlectures(Subject,N),
 agendainsert(Agenda,N,[],Pribefore,Subject,Restsubjects,Nodelist,Timetable,
              Newtimetable,Brothers,Fail).


% ****************************************************************************
% *                                                                          *
% *                          SOLVE OPTION: FIXED                             *
% *                                                                          *
% ****************************************************************************


% ============================================================================
/*
PREDICATE: fixedinsert(+Subject,+Restsubjects,+Times,+Nodelist,+Timetable,
                       ?Newtimetable,?Brothers,?Fail)
ARGUMENTS: Subject,       subject representation
           Restsubjects,  list (of subjects representations)
           Nodelist,      list (of integers - nodes numbers)
           Times,         list (of times representations)
           Timetable,     list (of nodereps)
           Newtimetable,  list (of nodereps)
           Brothers,      list (of trees representations)
           Fail,          yes or no
COMMENTS:  Succeeds after instantiating Newtimetable to the timetable solution,
           corresponding to Subject (and Restsubjects, using recursion),
           depending on the actual state of Timetable (according to the
           previous subjects. Brothers will hold the list of trees
           corresponding to Subject-Restsubjects, and Fail is 'no', only if
           there is no solution (backtracking optrion). Nodelist, input value,
           holds the path, from the top, up to our actual position in the
           tree search. It is necessary to set dependencies between nodes
           and asuumptions, and to assert the final solution (or partial).
           This predicate covers the case when subject has its lectures FIXED
           Fixed holds the list of such lectures). Therefore Brothers can only
           contain one Tree: if a fixed lecture is not possible, there is no
           possible backtracking, but failure is reported at once.

           However, The double recursion fixedinsert-settimetable and
           settimetable-agendainsert, makes it go to next subjects levels
           not neccessarily with fixed lectures. Therefore many 'Children'
           can appear at lower levels.
*/
% ============================================================================

fixedinsert(Subject,Restsubjects,[HTimes|TTimes],Nodelist,Timetable,
            Newtimetable,[Tree],Fail) :-
 HTimes = [Day,Hour],
 Nodedb = [Node,Subject,Day,Hour],
 db(Nodedb,Node),
 fix(Subject,[Day,Hour],As),
 putjustification(Node,[As]),
 dbtreenode(TreeNode),
 lookfor(Timetable,Day,Hour,Before,Then,After,Sides),
 ((notpos(Subject,[Day,Hour],As2),
   !,
   Fail1 = yes,
   Reason = notpos,
   Tree1 = [fail,As,As2],
   putjustification(0,[As,As2]))
  ;
  (test(Nodedb,Then,Sides,Then1,Tree1,HThen,Fail1),
   Reason = 'nonfollow and/or nonsimult')
 ),
 ((Fail1 = yes,
   !,
   assertpartial(Nodelist,Timetable),
   erasew('WAIT'),
   warning(Nodedb,Reason),
   write('It will be set, provided that you will change it later'),
   nl,
   write('Choose a Room for it'),
   nl,
   idwindow(menu,Menuwindow),
   lectroomsmenu(Menuwindow,Subject,Room),
   putwindow('WAIT'),
   Nodedb2 = [Node2,Subject,Day,Hour,Room],
   recsubtest(Nodedb,Then,Sides,_,_,_,_),
   db(Nodedb2,Node2),
   dbnode(Nodedb2),
   HThen2 = Nodedb2,
   Then2 = [Nodedb2|Then])
  ;
  (HThen = [Node2|_],
   HThen2 = HThen,
   Then2 = Then1)
 ),
 testok(Nodelist,Before,Then2,After,HThen2,Node2,TreeNode,Tree1,
        Timetable1,Father),
 fixedinsert(Subject,Restsubjects,TTimes,[TreeNode|Nodelist],Timetable1,
             Newtimetable,Children,Fail),
 Tree = [Father|Children].


% no more lectures to set -> go to next subject

fixedinsert(_,[HSubjects|TSubjects],[],Nodelist,Timetable,
            NewTimetable,Children,Fail) :-
 settimetable(HSubjects,TSubjects,Nodelist,Timetable,
              NewTimetable,Children,Fail).


% last subject, no more lectures to set -> solution

fixedinsert(_,[],[],Nodelist,Timetable,
            Timetable,[[solution|RevNodelist]],no) :-
 reverse(Nodelist,RevNodelist),
 asserta(sol(RevNodelist)).


% ****************************************************************************
% *                                                                          *
% *                        SOLVE OPTION: NON-FIXED                           *
% *                                                                          *
% ****************************************************************************


% ============================================================================
/*
```

```
PREDICATE: agendainsert(+Agenda,+N,+Daysbefore,+Pribefore,
                        +Subject,+Restsubjects,+Nodelist,+Timetable,
                        ?Newtimetable,?Brothers,?Fail)
ARGUMENTS: Agenda,       list (of Pri/Time)
           N,            integer
           Daysbefore,   list (of Ndays - days numbers)
           Pribefore,    list of integers (priorities of differences between
                         days at level before)
           Subject,      subject representation
           Restsubjects, list (of subjects representations)
           Nodelist,     list (of integers - nodes numbers)
           Timetable,    list (of nodereps)
           Newtimetable, list (of nodereps)
           Brothers,     list (of trees representations)
           Fail,         yes or no
COMMENTS:  Succeeds after instantiating Newtimetable to the timetable solution,
           corresponding to Subject (and Restsubjects, using recursion),
           depending on the actual state of Timetable (according to the
           previous subjects. Brothers will hold the list of trees
           corresponding to Subject-Restsubjects, and Fail is 'yes' or 'no',
           depending on finding a solution or not ("backtrack" case). Nodelist,
           input value, holds the path, from the top, up to our actual position
           in the tree search. It is necessary to set dependencies between nodes
           and asuumptions, and to assert the final solution.

           The strategy of agendainsert depends on the default option chosen:
           - "backtrack" means that "undesirable backtracking at levels before
             is allowed. This choice means that the user believes that there
             is a solution, and doesn't care how it is found. Therefore a
             failure in finding any solution after all effort will be reported
             to him in order to exit prolog and change the constraints.
           - "nobacktrack" means that we follow strictly the plan. If it does
             not work at a level, a lecture is set arbitrarily, where the user
             wants, and we follow It is the natural and MOST LOGICAL choice.

           An agenda of suitable times, ordered by priority (preferences) is
           used for best-first-search. Priority depends on:
           1) global ("all") preferences of times in the course (good, bad,
              verybad, notpos: "not possible")
           2) same kind of preferences for each particular subject.
           3) minimum difference in days with other lectures of same subject
              (avoidance of lectures in the same day, ...)

           In order to consider 3), Two arguments are used: Daysbefore, list
           of previous days numbers for the same subject, and Pribefore, list
           of priorities of type 3) the previous time (in order to remove
           it next time: the 'minimum difference' is completely different with
           0, 1, or 2 days in Daysbefore; therefore, we must recalculate it
           again, and remove (SUBSTRACT), Pribefore. 'nextagenda' predicate,
           shown later is committed to do it.

           Three TESTS are done, which lead to different processes (brackets
           and semicolons)
           1) Is our newly created tree node affected by 'notpos' (Pri>10)?
           2) Does 'test' predicate fail? (nonfollow/nonsimult/rooms/same time)
           3) Does 'agendainsert' fail in all lower levels?
           Knowing this, it is easier to follow the structure.
*/
% ============================================================================

agendainsert([Pri/[Hour,Nday]|TAgenda],N,Daysbefore,Pribefore,
             Subject,Restsubjects,Nodelist,Timetable,
             Newtimetable,Brothers,Fail) :-
 N > 0,
 translate(Day,Nday),
 Nodedb = [Node,Subject,Day,Hour],
 db(Nodedb,Node),
 dbass(Nodedb),
 dbtreenode(TreeNode),
 Pribefore = [_|TPribefore],
 ((Pri > 10, % any notpos
   !,
   notpos(Any,[Day,Hour],As),
   member(Any,[all,Subject]),
   notposfailed(Nodedb,TreeNode,[fail,As,Node],Bigbrother),
   choicebacktrack(N,Subject,Restsubjects,Nodelist,Timetable,Bigbrother,
                   Newtimetable,Brothers,Fail))
 ;
  (lookfor(Timetable,Day,Hour,Before,Then,After,Sides),
   test(Nodedb,Then,Sides,Then1,Tree1,HThen,Fail1),
   !,
   ((Fail1 = yes,
     !,
     testfailed(Nodelist,Nodedb,Node,TreeNode,Tree1,Bigbrother,_),
     agendainsert(TAgenda,N,Daysbefore,TPribefore,
                  Subject,Restsubjects,Nodelist,Timetable,
                  Newtimetable,Restbrothers,Fail),
     Brothers = [Bigbrother|Restbrothers])
   ;
    (HThen = [Node2|_],
     testok(Nodelist,Before,Then1,After,HThen,Node2,TreeNode,Tree1,
            Timetable1,Father),
     N1 is N - 1,
     ((N1 = 0,
       !)
     ;
      (nextagenda([Nday|Daysbefore],Pribefore,TAgenda,NewAgenda,Priafter))
     ),
     agendainsert(NewAgenda,N1,[Nday|Daysbefore],Priafter,
                  Subject,Restsubjects,[TreeNode|Nodelist],Timetable1,
                  Newtimetable1,Children,Fail2),
     Bigbrother = [Father|Children],
     ((Fail2 = no,
       Fail = no,
       Newtimetable = Newtimetable1,
       Brothers = [Bigbrother])
     ;

% BACKTRACKING: Next will only happen if 'backtracking' option

      (agendainsert(TAgenda,N,Daysbefore,TPribefore,
                    Subject,Restsubjects,Nodelist,Timetable,
                    Newtimetable,Restbrothers,Fail),
       Brothers = [Bigbrother|Restbrothers])
     )
    )
   )
  )
 ),
 !.


% empty agenda:

agendainsert([],N,_,_,
             Subject,Restsubjects,Nodelist,Timetable,
             Newtimetable,Brothers,Fail) :-
 Bigbrother = [0,'empty agenda'],
 N > 0,
 choicebacktrack(N,Subject,Restsubjects,Nodelist,Timetable,Bigbrother,
                Newtimetable,Brothers,Fail).


% no more lectures to set -> go to next subject

agendainsert(_,0,_,_,
             _,[HSubjects|TSubjects],Nodelist,Timetable,
             Newtimetable,Brothers,Fail) :-
 settimetable(HSubjects,TSubjects,Nodelist,Timetable,
              Newtimetable,Brothers,Fail).


% last subject, no more lectures to set -> solution

agendainsert(_,0,_,_,
             _,[],Nodelist,Timetable,
             Timetable,[[solution|RevNodelist]],no) :-
 reverse(Nodelist,RevNodelist),
 asserta(sol(RevNodelist)).


% ==========================================================================
/*
PREDICATE: choicebacktrack(+N,+Subject,+Restsubjects,
                           +Nodelist,+Timetable,+Bigbrother
                           ?Newtimetable,?Brothers,?Fail)
ARGUMENTS: N,            integer
           Subject,      subject representation
           Restsubjects, list (of subjects representations)
           Nodelist,     list (of integers - nodes numbers)
           Timetable,    list (of nodereps)
           Bigbrother,   list (tree representation)
           Newtimetable, list (of nodereps)
           Brothers,     list (of trees representations)
           Fail,         yes or no
COMMENTS:  Succeeds after instantiating Newtimetable to the timetable solution,
           corresponding to Subject (and Restsubjects, using recursion),
           in the case where no more choices were found in the agenda.
           depending on the actual state of Timetable (according to the
           previous subjects. Brothers will hold the list of trees
           corresponding to Subject-Restsubjects, and Fail is 'yes' or 'no',
           depending on finding a solution or not ("backtrack" case). Nodelist,
           input value, holds the path, from the top, up to our actual position
           in the tree search. It is necessary to set dependencies between nodes
           and asuumptions, and to assert the final solution.

           The strategy depends on the default option chosen:
           - "backtrack" means that "undesirable backtracking at levels before
             is allowed. This choice means that the user believes that there
             is a solution, and doesn't care how it is found. Therefore a
             failure in finding any solution after all effort will be reported
             to him in order to exit prolog and change the constraints.
           - "nobacktrack" means that we follow strictly the plan. If it does
             not work at a level, a lecture is set arbitrarily, where the user
             wants, and we follow It is the natural and MOST LOGICAL choice.
*/
% ==========================================================================

choicebacktrack(N,Subject,Restsubjects,Nodelist,Timetable,Bigbrother,
               Newtimetable,Brothers,Fail) :-
 assertpartial(Nodelist,Timetable),
 nl,
```

```
 ((backtrack,
   !,
   write('BACKTRACK'),
   nl,
   Newtimetable = Timetable,
   Restbrothers = [[fail,no_more_choices]],
   Brothers = [Bigbrother|Restbrothers],
   Fail = yes)
  ;
  (nobacktrack,
   !,
   erasew('WAIT'),
   write(N),
   write(' lecture(s) for subject '),
   write(Subject),
   nl,
   write('cannot be set following the plan'),
   nl,
   write('Choose arbitrarily times for it(them)'),
   nl,
   readtimes(N,Subject,Timetable,[],_,Timetable1,Nodedblist),
   putwindow('WAIT'),
   dbtreenode(TreeNode),
   extract(Timetable1,Nodes),
   putjustification(TreeNode,Nodes),
   Father = [TreeNode|Nodedblist],
   agendainsert(_,0,_,_,
                Subject,Restsubjects,[TreeNode|Nodelist],Timetable1,
                Newtimetable,Children,_),
   Smallbrother = [Father|Children],
   Brothers = [Bigbrother,Smallbrother],
   Fail = no)
 ).


% ==========================================================================
/*
PREDICATE: assertpartial(+Nodelist,+Timetable)
ARGUMENTS: Nodelist,     list (of integers - nodes numbers)
           Timetable,    list (of nodereps)
COMMENTS:  Succeeds after asserting a new partial solution (both the
           partial Timetable and the Nodelist path) only if it is the
           longest one, up to now.
*/
% ==========================================================================

assertpartial(Nodelist,_) :-
 takepart(X),
 length(X,P),
 length(Nodelist,N),
 P >= N.

assertpartial(Nodelist,Timetable) :-
 ((Nodelist = [Partnode|_])
  ;
  (Partnode = 'top')
 ),
 asserta(historyTimetable(backtrack,Partnode,Timetable)),
 reverse(Nodelist,RevNodelist),
 ((part(_),
   retract(part(_)))
  ;
  (true)
 ),
 asserta(part(RevNodelist)).


% ============================================================================
/*
PREDICATE: takepart(?X)
ARGUMENTS: X, anything
COMMENTS:  Succeeds after instantiating X to the FIRST item such that
           part(X)
*/
% ============================================================================

takepart(X) :-
 part(X),
 !.


% ============================================================================
/*
PREDICATE: dayshourspri(+Subject,+Dayshours,?PriHourNday)
ARGUMENTS: Subject, subject representation
           Dayshours,   list (of Times in form [Day,Hour])
           PriHourNday, Pair Pri/Time, in form [Hour,Nday], Nday: day number
COMMENTS:  Succeeds after instantiating Pri to the associated priority
           for [Subject-Day-Hour], according to 'punct' criteria (No
           difference in days involved here)
*/
% ============================================================================

dayshourspri(Subject,Dayshours,Pri/HN) :-
 member(DH,Dayshours),
 DH = [Day,Hour],
 translate(Day,Nday),
 HN = [Hour,Nday],
 punct(all,DH,Pri1),
 punct(Subject,DH,Pri2),
 Pri is Pri1 + Pri2.


% ============================================================================
/*
PREDICATE: punct(+Subject,+DH,?N)
ARGUMENTS: Subject, subject representation
           DH,      Time in form [Day,Hour]
           N,       integer
COMMENTS:  Succeeds after instantiating N to the priority associated with
           the state of the Time in constraints file (bad,verybad,notpos,none)
*/
% ============================================================================

punct(Subject,DH,1) :-
 bad(Subject,DH),
 !.

punct(Subject,DH,2) :-
 verybad(Subject,DH),
 !.

punct(Subject,DH,11) :-
 notpos(Subject,DH,_),
 !.

punct(_,_,0).
```

```
% ============================================================================
/*
PREDICATE: nextagenda(+Daysbefore,+Pribefore,+Agenda,?NextAgenda,?Priafter)
ARGUMENTS: Daysbefore,   list of integers (days numbers)
           Pribefore,    list of integers
           Agenda,       list of Pri/Time
           NextAgenda,   list of Pri/Time
           Priafter,     list of integers
COMMENTS:  Succeeds after instantiating Nextagenda to the next agenda,
           and Priafter to the list containing the Next Pribefore
           elements), according to Daysbefore, Pribefore and Agenda.
*/
% ============================================================================

nextagenda(Daysbefore,Pribefore,Agenda,Nextagenda,Priafter) :-
 recnextagenda(Daysbefore,Pribefore,Agenda,T1),
 sort(T1,T2),
 separate(T2,Nextagenda,Priafter).


% ============================================================================
/*
PREDICATE: recnextagenda(+Daysbefore,+Pribefore,+Agenda,?T1)
ARGUMENTS: Daysbefore,   list of integers (days numbers)
           Pribefore,    list of integers
           Agenda,       list of Pri/Time
           T1,           list of Pri/Time/Sub
COMMENTS:  Succeeds after instantiating T1 to the list whose elements are
           pairs Pri/Time (Next agenda elements) - Sub (Next Pribefore
           elements), according to Daysbefore, Pribefore and Agenda.
*/
% ============================================================================

recnextagenda(_,_,[],[]).

recnextagenda(Daysbefore,[Substract|TSubstract],[Pri/[Hour,Nday]|T],
              [Pri1/[Hour,Nday]/Add|T1]) :-
 setof(X,daydist(Nday,Daysbefore,X),[Mindis|_]),
 optdifdays(N),
 Dif is Mindis - N,
 abs(Dif,Add),
 Mid is Pri + Add,
 Pri1 is Mid - Substract,
 recnextagenda(Daysbefore,TSubstract,T,T1).



% ============================================================================
/*
PREDICATE: daydist(+NDay,+DS,?Dis)
ARGUMENTS: NDay,   integer (day number)
           DS,     list of integers (days numbers)
           Dis,    integer
COMMENTS:  Succeeds after instantiating Dis to the absolute difference between
           Nday and an element of DS
*/
% ============================================================================

daydist(Day,DS,Dis) :-
 member(D,DS),
 distance(Day,D,Dis).
```

```
% ===========================================================================
/*
PREDICATE: distance(+D1,+D2,?Dis)
ARGUMENTS: D1,      integer
           D2,      integer
           Dis,     integer
COMMENTS:  Succeeds after instantiating Dis to the absolute difference between
           D1 and D2
*/
% ===========================================================================

distance(D,D,0) :-
 !.

distance(D1,D2,Dis) :-
 P is D1 - D2,
 abs(P,Dis).


% ===========================================================================
/*
PREDICATE: separate(+T,?NextAgenda,?Priafter)
ARGUMENTS: T,            list of Pri/Time/Sub
           Agenda,       list of Pri/Time
           Pribefore,    list of integers
COMMENTS:  Succeeds after instantiating NextAgenda to the list whose elements
           are pairs Pri/Time (Next agenda elements) and Pribefore, to the
           list whose elements are Sub (integers), according to T.
*/
% ===========================================================================

separate([],[],[]).

separate([A1/A2/A3|T],[A1/A2|T1],[A3|T2]) :-
 separate(T,T1,T2).



% ***************************************************************************
% *                                                                         *
% *                                TESTS                                    *
% *                                                                         *
% ***************************************************************************


% ===========================================================================
/*
PREDICATE: lookfor(+Timetable,+Day,+Hour,?Before,?Then,?After,?Sides)
ARGUMENTS: Timetable, list (of nodereps)
           Day,       day representation
           Hour,      hour representation
           Before,    list (of nodereps)
           Then,      list (of nodereps)
           After,     list (of nodereps)
           Sides,     list (of nodereps)
COMMENTS:  Succeeds after instantiating Before, Then, After and Sides to the
           elements of the Timetable: before, the same, after, and adjacent
           respect the input Day and Hour.
*/
% ===========================================================================

lookfor([],_,_,[],[],[],[]).
```

```
lookfor([H|T],Day,Hour,Before,Then,After,Sides) :-
 ((H = [_,_,Day,Hour1|_],
   !,
   ((Hour2 is Hour - 1,
     Hour2 = Hour1,
     !,
     Sides = [H|Sides1],
     Before = [H|Before1],
     lookfor(T,Day,Hour,Before1,Then,After,Sides1))
   ;
    (Hour1 < Hour,
     Before = [H|Before1],
     lookfor(T,Day,Hour,Before1,Then,After,Sides))
   ;
    (Hour1 = Hour,
     Before = [],
     Then = [H|Then1],
     lookfor(T,Day,Hour,[],Then1,After,Sides))
   ;
    (Hour2 is Hour + 1,
     Hour2 = Hour1,
     !,
     Sides = [H|Sides1],
     Before = [],
     Then = [],
     After = [H|After1],
     lookfor(T,Day,Hour,[],[],After1,Sides1))
   ;
    (Hour1 > Hour,
     Sides  = [],
     Before = [],
     Then   = [],
     After  = [H|T])
   )
  )
 ;
  (H = [_,_,Day1|_],
   translate(Day1,D1),
   translate(Day,D),
   ((D1 < D,
     Before = [H|Before1],
     lookfor(T,Day,Hour,Before1,Then,After,Sides))
   ;
    (D1 > D,
     Sides  = [],
     Before = [],
     Then   = [],
     After  = [H|T]))
  )
 ).


% ===========================================================================
/*
PREDICATE: test(+Nodedb,+Then,+Sides,?Then1,?Tree,?HThen,?Fail)
ARGUMENTS: Nodedb,         noderep
           Then,           list (of nodereps)
           Sides,          list (of nodereps)
           Then1,          list (of nodereps)
           Tree,           tree representation ([] or information leaf)
           HThen,          noderep
           Fail,           yes or no
COMMENTS:  Succeeds after instantiating Then1 to the 'Then' ("lookfor") list
           after including the 'noderep' information for the new node (HThen),
           obtained from Nodedb, after testing the conditions shown below.
           if no failure is found, Fail is set to no, and Tree can hold
           either [], or 'changes' information, depending on if we have changed
           rooms or not
           If a failure is found, Then1 is Then, Fail is set to yes, and
           Tree holds 'failure' information.

           The Test consists of:
           - possible nonsimult crashes (Then used)
           - possible nonfollow crashes (Sides used)
             (Both of them tested in subtest)
           - possible rooms conflict. As rooms conflict is supposed not to
             be serious (but have to be solved), we don't backtrack, BUT
             have to 'rearrangerooms'. If it is possible, I consider that
             the tree node is the same as before, though including the
             'changes' information (Much more practical and efficient than
             backtracking). Otherwise, the failure is reported and backtracking
             is done.
             (tested in rearrangerooms, via subtestchoices)
*/

% empty 'Then and Sides' case (No failure is possible)

test([_,Subject,Day,Hour|_],[],[],[HThen],[],HThen,no) :-
 !,
 lectrooms(Subject,[Room|_]),
 HThen = [Node,Subject,Day,Hour,Room],
 db(HThen,Node),
 dbnode(HThen).


% general case

test(Nodedb,Then,Sides,Then1,Tree,HThen,Fail) :-
 subtest(Nodedb,Then,Sides,Roomsbefore,Node,As,Failsub),
 subtestchoices(Nodedb,Then,Roomsbefore,Node,As,Failsub,
                Then1,Tree,HThen,Fail).


% ===========================================================================
/*
PREDICATE: subtest(+Nodedb,+Then,+Sides,?Roomsbefore,?Node,?As,?Fail)
ARGUMENTS: Nodedb,        noderep
           Then,          list (of nodereps)
           Sides,         list (of nodereps)
           Roomsbefore,   list (of rooms representations)
           Node           integer (node number)
           As             integer (constraint assumption number)
           Fail,          yes or no
COMMENTS:  Succeeds after testing:
           - possible lecures for the same subject at the same time
           - possible nonsimult crashes (Then used)
           - possible nonfollow crashes (Sides used)

           if no failure is found, Fail is set to no, and Roomsbefore to
           the list of rooms in then (possible conflicts) that will be
           passed to subtestchoices
           If a failure is found, Fail is set to yes, and Node and As are
           instantiated to the nodes that, together with our created node
           lead to a contradiction, or to 0, if "same time"
*/
% ===========================================================================

subtest(_,[],[],[],_,_,no).
```

```
subtest(Nodedb,[],[HSides|TSides],[],Node,As,Fail) :-
 Nodedb = [_,Subject1,_,_],
 HSides = [_,Subject,Day,Hour,_],
 ((nonfollowing(Subject,Subject1,As),
   !,
   node([Node,Subject,Day,Hour]),
   Fail = yes)
 ;
  (subtest(Nodedb,[],TSides,[],Node,As,Fail))
 ).

subtest(Nodedb,[HThen|TThen],Sides,[Room|Roomsbefore],Node,As,Fail) :-
 Nodedb = [_,Subject1,_,_],
 HThen = [_,Subject,Day,Hour,Room],
 ((Subject = Subject1,
   !,
   Node = 0,
   As = 0,
   Fail = yes)
 ;
  (nonsimultaneous(Subject,Subject1,As),
   !,
   node([Node,Subject,Day,Hour]),
   Fail = yes)
 ;
  (subtest(Nodedb,TThen,Sides,Roomsbefore,Node,As,Fail))
 ).


% ===========================================================================
/*
PREDICATE: recsubtest(+Nodedb,+Then,+Sides,?Roomsbefore,?Node,?As,?Fail)
ARGUMENTS: Nodedb,        noderep
           Then,          list (of nodereps)
           Sides,         list (of nodereps)
           Roomsbefore,   list (of rooms representations)
           Node           0 or 1
           As             0 or 1
           Fail,          yes or no
COMMENTS:  Succeeds after testing (IN THIS CASE: ALL!):
           - possible lecures for the same subject at the same time
           - possible nonsimult crashes (Then used)
           - possible nonfollow crashes (Sides used)

           if no failure is found, Fail is set to no, and Roomsbefore to
           the list of rooms in then (possible conflicts) that will be
           passed to subtestchoices
           If a failure is found, Fail is set to yes, and Node and As are
           instantiated to 0 if "same time", 1 otherwise. justifications are
           sent to ATMS about Nodes and assumptions that, together with our
           created node, lead to a contradiction.
*/
% ===========================================================================

recsubtest(_,[],[],[],1,1,no).

recsubtest(Nodedb,[],[HSides|TSides],[],NewNode,NewAs,Fail) :-
 Nodedb = [Lastnode,Subject1,_,_],
 HSides = [_,Subject,Day,Hour,_],
 ((nonfollowing(Subject,Subject1,As),
   !,
   node([Node,Subject,Day,Hour]),
   putjustification(0,[Lastnode,Node,As]),
   Fail = yes)
  ;
  (Fail = NewFail)
 ),
 recsubtest(Nodedb,[],TSides,[],NewNode,NewAs,NewFail).

recsubtest(Nodedb,[HThen|TThen],Sides,[Room|Roomsbefore],NewNode,NewAs,Fail) :-
 Nodedb = [Lastnode,Subject1,_,_],
 HThen = [_,Subject,Day,Hour,Room],
 ((Subject = Subject1,
   !,
   NewNode = 0,
   NewAs = 0,
   Fail = yes)
  ;
  (nonsimultaneous(Subject,Subject1,As),
   !,
   node([Node,Subject,Day,Hour]),
   putjustification(0,[Lastnode,Node,As]),
   Fail = yes,
   New2Node = NewNode,
   New2As = NewAs)
  ;
  (Fail = NewFail,
   New2Node = NewNode,
   New2As = NewAs)
 ),
 recsubtest(Nodedb,TThen,Sides,Roomsbefore,New2Node,New2As,NewFail).


% ==========================================================================
/*
PREDICATE: nonsimultaneous(?X,?Y,?Z)
ARGUMENTS: X, subject representation
           Y, subject representation
           Z, integer: constraint assumption number
COMMENTS:  Succeeds if nonsimult(X,Y,Z) or nonsimult(Y,X,Z)
*/
% ==========================================================================

nonsimultaneous(X,Y,Z) :-
 nonsimult(X,Y,Z).

nonsimultaneous(X,Y,Z) :-
 nonsimult(Y,X,Z).


% ==========================================================================
/*
PREDICATE: nonfollowing(?X,?Y,?Z)
ARGUMENTS: X, subject representation
           Y, subject representation
           Z, integer: constraint assumption number
COMMENTS:  Succeeds if nonfollow(X,Y,Z) or nonfollow(Y,X,Z)
*/
% ==========================================================================

nonfollowing(X,Y,Z) :-
 nonfollow(X,Y,Z).

nonfollowing(X,Y,Z) :-
 nonfollow(Y,X,Z).
```

```
% ==========================================================================
/*
PREDICATE: subtestchoices(+Nodedb,+Then,+Roomsbefore,+Node,+As,+Failsub,
                          ?Then1,?Tree,?HThen,?Fail)
ARGUMENTS: Nodedb,        noderep
           Then,          list (of votereps)
           Roomsbefore,   list (of rooms representations)
           Node,          integer (node number)
           As,            integer (constraint assumption number)
           Failsub,       yes or no
           Then1,         list (of votereps)
           Tree,          tree representation ([] or information leaf)
           HThen,         noderep
           Fail,          yes or no
COMMENTS:  Succeeds after instantiating Then1 to the 'Then' list after
           including the 'noderep' information for the new node (HThen),
           obtained from Nodedb, after testing the condition shown below.
           Failsub holds the result of 'subtest'.
           if no failure is found, Fail is set to no, and Tree can hold
           either [], or 'changes' information, depending on if we have changed
           rooms or not
           If a failure is found, Then1 is Then, Fail is set to yes, and
           Tree holds 'failure' information.

           The Test consists of:
           - possible rooms conflict. As rooms conflict is supposed not to
             be serious (but have to be solved), we don't backtrack, BUT
             have to 'rearrangerooms'. If it is possible, I consider that
             the tree node is the same as before, though including the
             'changes' information (Much more practical and efficient than
             backtracking). Otherwise, the failure is reported and backtracking
             is done.
             (tested in rearrangerooms)
*/
% ==========================================================================

% Failsub = yes, because the same subject was in Then

subtestchoices([Lastnode|_],_,_,0,0,yes,
               _,[fail,Lastnode,0,0],_,yes) :-
 !.

% Failsub = yes, because of recsubtest

subtestchoices([Lastnode|_],_,_,1,1,yes,
               _,[fail,Lastnode,1,1],_,yes) :-
 !.


% Failsub = yes, because of constraint As violation.

subtestchoices([Lastnode|_],_,_,Node,As,yes,
               _,[fail,Lastnode,Node,As],_,yes) :-
 putjustification(0,[Lastnode,Node,As]).


% Failsub = no

subtestchoices(Nodedb,Then,Roomsbefore,_,_,no,
               Then1,Tree,HThen,Fail) :-
 Nodedb = [_,Subject|_],
 lectrooms(Subject,Rooms),
 membertest(Rooms,Roomsbefore,Nodedb,Nodedblist,Fail1),
 ((Fail1 = no,
   Nodedblist = [HThen|_],
   Then1 = [HThen|Then],
   Fail = no,
   Tree = [])
 ;
  (rearrangerooms([Nodedb|Then],[],Then1,Fail2),
   ((Fail2 = no,
     Then1 = [HThen|Thenprev],
     Fail = no,
     Tree = [changed_rooms,Then,'will_be:',Thenprev])
    ;
    (Fail = yes,
     Tree = [fail,not_possible_rooms_arrangement])
   )
  )
 ).


% ==========================================================================
/*
PREDICATE: recsubtestchoices(+Nodedb,+Timetable,+Then,+Sides,
                             ?Newtimetable,?HThen,?Fail)
ARGUMENTS: Nodedb,        noderep
           Timetable,     list of noderep
           Then,          list of noderep
           Sides,         list of noderep
           Newtimetable,  list of noderep
           HThen,         noderep
           Fail,          yes or no
COMMENTS:  Succeeds after performing 'nonsimult' and 'nonfollow' tests,
           via 'recsubtest'. If they are successful, an attempt to arrange
           room for Nodedb is done. If 'same room'
           conflicts' arise, Fail is yes. If any other conflict arise, Fail is
           no, but adequate messages are shown.
           Otherwise Newtimetable holds the new timetable after setting the
           new noderep (HThen), and Fail is no. NO ASSERTION OTHER THAN
           'ATMS RELATED' IS DONE.
*/
% ==========================================================================

recsubtestchoices(Nodedb,Timetable,Then,Sides,
                  Newtimetable,HThen,no) :-
 Nodedb = [Node1,Subject,Day,Hour],
 recsubtest(Nodedb,Then,Sides,Roomsbefore,Node,As,Subfail),
 subtestchoices(Nodedb,Then,Roomsbefore,Node,As,Subfail,Then1,_,HThen,ChFail),
 ((Subfail = yes,
   warning(Nodedb,'nonfollow and/or nonsimult'),
   lectroom(Subject,Room,As2),
   Nodedb2 = [Node2,Subject,Day,Hour,Room],
   db(Nodedb2,Node2),
   putjustification(Node2,[Node1,As2]),
   Then2 = [Nodedb2|Then])
 ;
  (Subfail = no,
   ((ChFail = yes,
     warning(Nodedb,'no rooms arrangement with other lectures'),
     nl,
     lectroom(Subject,Room,As2),
     Nodedb2 = [Node2,Subject,Day,Hour,Room],
     db(Nodedb2,Node2),
     putjustification(Node2,[Node1,As2]),
     Then2 = [Nodedb2|Then])
    ;
    (ChFail = no,
     Then2 = Then1)
   )
  )
 ),
 lookfor(Timetable,Day,Hour,Before,_,After,_),
 conc(Then2,After,After1),
 conc(Before,After1,Newtimetable).


% ==========================================================================
/*
PREDICATE: rearrangerooms(+Then,+Roomsbefore,?Then1,?Fail)
ARGUMENTS: Then,         list (of nodereps)
           Roomsbefore,  list (of rooms representations)
           Then1,        list (of nodereps)
           Fail,         yes or no
COMMENTS:  Succeeds after instantiating Then1 to the result of finding a new
           arrangement of rooms for Then1, BUT without repetitions of rooms.
           The predicate uses recursion and double recursion with
           recarrange. If that instantiation is possible, Fail is 'no'.
           If it is not possible, Fail is 'yes'.
*/
% ==========================================================================

rearrangerooms([],_,[],no).

rearrangerooms([HThen|TThen],Roomsbefore,[HThen1|TThen1],Fail) :-
 HThen = [_,Subject|_],
 lectrooms(Subject,Rooms),
 membertest(Rooms,Roomsbefore,HThen,Nodedblist,Memberfail),
 ((Memberfail = no,
   recarrange(TThen,Roomsbefore,Nodedblist,HThen1,TThen1,Fail))
 ;
  (Fail = yes)
 ).


% ==========================================================================
/*
PREDICATE: recarrange(+Then,+Roomsbefore,+Nodedblist,?HThen1,?TThen1,?Fail)
ARGUMENTS: Then,         list (of nodereps)
           Roomsbefore,  list (of rooms representations)
           Nodedblist,   list (of nodereps)
           HThen1,       noderep
           TThen1,       list (of nodereps)
           Fail,         yes or no
COMMENTS:  Succeeds after instantiating HThen1 and TThen1 to the legal values
           such that Then1 = [HThen1|TThen1] is equivalent to
           Then = [HThen|TThen], BUT without repetitions of rooms.
           The predicate uses recursion and double recursion with
           rearrangerooms. HThen is picked up from Nodedblist.
           If that instantiation is possible, Fail is 'no'. If it is not
           possible, whatever 'HThen' we choose, Fail is 'yes'.
*/
% ==========================================================================

recarrange(_,_,[],_,_,yes).

recarrange(TThen,Roomsbefore,[HThen|TNodedblist],HThen1,TThen1,Fail) :-
 HThen = [_,_,_,_,Room1],
 rearrangerooms(TThen,[Room1|Roomsbefore],TThen1,Fail1),
 ((Fail1 = no,
   Fail = no,
   HThen1 = HThen)
 ;
  (recarrange(TThen,Roomsbefore,TNodedblist,HThen1,TThen1,Fail))
 ).


% ============================================================================
/*
PREDICATE: membertest(+Rooms,+Roomsbefore,+Nodedb,?Nodedblist,?Fail)
ARGUMENTS: Rooms,       list (of rooms representations)
           Roomsbefore, list (of rooms representations)
           Nodedb,      noderep
           Nodedblist,  list (of nodereps)
           Fail,        yes or no
COMMENTS:  Succeeds after testing which rooms in Rooms are not in Roomsbefore,
           and instantiating Nodedblist to the list of the
           new created nodes corresponding to such those rooms. Fail is 'yes'
           if there are not such rooms, 'no' otherwise.
*/
% ============================================================================

membertest([],_,_,[],yes) :-
 !.

membertest([HRooms|TRooms],Roomsbefore,Nodedb,Nodedblist,Fail) :-
 Nodedb = [_,Subject,Day,Hour|_],
 HNodedblist = [Node1,Subject,Day,Hour,HRooms],
 db(HNodedblist,Node1),
 dbnode(HNodedblist),
 ((member(HRooms,Roomsbefore),
   !,
   membertest(TRooms,Roomsbefore,Nodedb,Nodedblist,Fail)
 ;
  (membertest(TRooms,Roomsbefore,Nodedb,TNodedblist,_),
   Nodedblist = [HNodedblist|TNodedblist],
   Fail = no))
 ).


% ============================================================================
/*
PREDICATE: notposfailed(+Nodedb,+TreeNode,+Tree1,?Tree)
ARGUMENTS: Nodedb,        list (noderep)
           Treenode,      integer (node number)
           Tree1,         list (tree representation: information leaf)
           Tree,          list (tree representation)
COMMENTS:  Succeeds after instantiating Tree to the list with the right
           information according to Treenode, Nodedb and Tree1,
           and sending ATMS the failure (it covers 'notpos' failures)
*/
% ============================================================================

notposfailed(Nodedb,TreeNode,Tree1,[[TreeNode,Nodedb]|[Tree1]]) :-
 Tree1 = [fail|FailAsNode],
 putjustification(0,FailAsNode).


% ============================================================================
/*
PREDICATE: testok(+Nodelist,+Before,+Then1,+After,
                  +Nodedb,+Node,+TreeNode,+Tree1,
                  ?Timetable1,?Tree)
ARGUMENTS: Nodelist,      list (of integers - nodes numbers)
           Before,        list (of nodereps)
           Then1,         list (of nodereps)
           After,         list (of nodereps)
           Nodedb,        noderep
           Node,          integer (node number)
           Treenode,      integer (node number)
           Tree1,         list (tree representation: information leaf)
           Timetable1,    list (of nodereps)
           Tree,          list (tree representation)
COMMENTS:  Succeeds after instantiating Tree to the list with the right
           information according to Treenode, Nodedb and Tree1,
           and sending ATMS the justification of the tree
           node, via treejustify
*/
% ============================================================================

testok(Nodelist,Before,Then1,After,Nodedb,Node,TreeNode,Tree1,
      Timetable1,[TreeNode|[Nodedb|Tree1]]) :-
 conc(Then1,After,After1),
 conc(Before,After1,Timetable1),
 ((Tree1 = [changed_rooms|_],
   !,
   treejustify(_,Timetable1,Node,TreeNode,yes))
 ;
  (treejustify(Nodelist,_,Node,TreeNode,no))
 ).


% ============================================================================
/*
PREDICATE: testfailed(+Nodelist,+Nodedb,+Node,+TreeNode,+Tree1,?Tree,?Fail)
ARGUMENTS: Nodelist,      list (of integers - nodes numbers)
           Nodedb,        list (noderep)
           Node,          integer (node number)
           Treenode,      integer (node number)
           Tree1,         list (tree representation: information leaf)
           Tree,          list (tree representation)
           Fail,          yes or no
COMMENTS:  Succeeds after instantiating Tree to the list with the right
           information according to Treenode, Nodedb and Tree1, setting
           Fail to yes, and sending ATMS the justification of the tree
           node, via treejustify
*/
% ============================================================================

testfailed(Nodelist,Nodedb,Node,TreeNode,Tree1,
           [[TreeNode,Nodedb]|[Tree1]],yes) :-
 treejustify(Nodelist,_,Node,TreeNode,no).


% ============================================================================
/*
PREDICATE: treejustify(+Nodelist,+Timetable,+As,+Node,+Changedrooms)
ARGUMENTS: Nodelist,      list (of integers - nodes numbers)
           Nodedblist,    list (of nodereps)
           Node,          integer (node number)
           Treenode,      integer (node number)
           Changedrooms, yes or no
COMMENTS:  If Changedrooms is no, it succeeds after sending ATMS the
           justification [In,Node] -> Treenode, where In is the first node in
           Nodelist - the tree father - (Nodelist is empty only for the
           first node).
           If Changedrooms is yes, it succeeds after sending ATMS the
           justification [Node|Nodelist2] -> Node, where Nodelist2 is the list
           of numbers of the nodes that appear in Timetable
*/
```

```
% ==========================================================================

treejustify(_,Timetable,Node,Treenode,yes) :-
 extract(Timetable,Nodelist2),
 putjustification(Treenode,[Node|Nodelist2]).

treejustify([In|_],_,Node,Treenode,no) :-
 putjustification(Treenode,[In,Node]).

treejustify([],_,Node,Treenode,no) :-
 putjustification(Treenode,[Node]).


% **************************************************************************
% *                                                                        *
% *                      ATMS INFORMATION-HISTORY                          *
% *                                                                        *
% **************************************************************************


% ==========================================================================
/*
PREDICATE: findinconsist(?BadAsNodes)
ARGUMENTS: BadAsNodes, list of lists of [Assumption] plus node numbers
COMMENTS:  Succeeds after instantiating Tracks to the list of 'tracks'
           recorder in the environment, and BadAsNodes to the list of "lists,
           whose head is an unitary list containing an unsatisfied constraint
           (assumption) number, and whose tail is the list of timetable
           positions (nodes numbers) that make it unsatisfiable"
*/
% ==========================================================================

findinconsist(BadAsNodes) :-
 write('I am testing ATMS ...'),
 nl,
 solnode(Solnode),
 constraints(Constraints),
 find_bad_as(Solnode,Constraints,BadAs),
 atms_get_envs([0],BadEnvs),
 timetable(Timetable),
 extract(Timetable,Nodes),
 find_bad_asnodes(BadAs,BadEnvs,Nodes,BadAsNodes),
 nl.


% ==========================================================================
/*
PREDICATE: extract(+Nodedblist,?Nodelist)
ARGUMENTS: Nodedblist,   list (of nodereps)
           Nodelist,     list (of integers - nodes numbers)
COMMENTS:  Succeeds after instantiating Nodelist to the list of numbers of
           the nodes that appear in Nodedblist
*/
% ==========================================================================

extract([],[]).

extract([[Node|_]|T],[Node|T1]) :-
 extract(T,T1).


% ==========================================================================
/*
PREDICATE: find_bad_as(+Solnode,+Constraints,?BadAs)
ARGUMENTS: Solnode,     node number
           Constraints, list of constraints (assumptions) numbers
           BadAs,       list of constraints (assumptions) numbers
COMMENTS:  Succeeds after instantiating BadAs to the list of actual
           constraints that are not satisfied by the solution (solnode),
           according to ATMS information
*/
% ==========================================================================

find_bad_as(_,[],[]).

find_bad_as(Solnode,[As|Aslist],[As|BadAs]) :-
 atms_get_envs([Solnode,As],[]),
 !,
 find_bad_as(Solnode,Aslist,BadAs).

find_bad_as(Solnode,[_|Aslist],BadAs) :-
 find_bad_as(Solnode,Aslist,BadAs).


% ==========================================================================
/*
PREDICATE: find_bad_asnodes(+BadAs,+BadEnvs,+Nodes,?BadAsNodes)
ARGUMENTS: BadAs,      list of assumptions numbers
           BadEnvs,    list of environments (lists of Assumptions)
           Nodes,      list of nodes numbers
           BadAsNodes, list of lists of [Assumption] plus node numbers
COMMENTS:  Succeeds after instantiating BadAsNodes to the list of "lists,
           whose head is an unitary list containing a constraint from BadAs
           (assumption) number, and whose tail is the list of timetable
           positions (nodes numbers) that make it unsatisfiable"
*/
% ==========================================================================

find_bad_asnodes([],_,_,[]).

find_bad_asnodes([H|T],BadEnvs,Nodes,BadAsNodes) :-
 find_bad_nodes(H,BadEnvs,Nodes,HB),
 find_bad_asnodes(T,BadEnvs,Nodes,TB),
 conc(HB,TB,BadAsNodes).


% ==========================================================================
/*
PREDICATE: find_bad_nodes(+As,+BadEnvs,+Nodes,?BadAsNodes)
ARGUMENTS: As,         assumption number
           BadEnvs,    list of environments (lists of Assumptions)
           Nodes,      list of nodes numbers
           BadAsNodes, list of lists of [Assumption] plus node numbers
COMMENTS:  Succeeds after instantiating BadAsNodes to the list of "lists,
           whose head is an unitary list containing As, unsatisfied constraint
           (assumption) number, and whose tail is the list of timetable
           positions (nodes numbers) that make it unsatisfiable"
*/
% ==========================================================================

find_bad_nodes(_,[],_,[]).

find_bad_nodes(As,[H|T],Nodes,BadAsNodes) :-
 setof2(X,testAsEnvs(As,H,Nodes,X),HB),
 find_bad_nodes(As,T,Nodes,TB),
 conc(HB,TB,BadAsNodes).
```

```
% ==========================================================================
/*
PREDICATE: testAsEnvs(+As,+Env,+Nodes,?BadAsNodes)
ARGUMENTS: As,          assumption number
           Env,         list of environments (lists of Assumptions)
           Nodes,       list of nodes numbers
           AsNodes,     list of [Assumption] plus node numbers
COMMENTS:  Succeeds after instantiating AsNodes to the list
           whose head is an unitary list containing As, unsatisfied constraint
           (assumption) number, and whose tail is the list of timetable
           positions (nodes numbers) that make it unsatisfiable
*/
% ==========================================================================

testAsEnvs(As,Env,Nodes,[As|BadNodes]) :-
 memberchk(As,Env),
 testenvs(Env),
 atms_get_context(Env,inconsistent,Context),
 setof(X,fivetypenode(Context,Nodes,X),BadNodes),
 ((notpos(_,_,As),
   !,
   BadNodes = [_])
 ;
  (nonsharedrooms(As),
   !,
   BadNodes = [Node1,Node2],
   node([Node1,_,_,_,Room]),
   node([Node2,_,_,_,Room]))
 ;
  (BadNodes = [_,_])
 ).
% allmembers(BadNodes,Nodes).


% ==========================================================================
/*
PREDICATE: testenvs(+Aslist)
ARGUMENTS: Aslist, list of assumptions numbers
COMMENTS:  Succeeds if no assumption in Aslist have been deleted
*/
% ==========================================================================

testenvs([]).

testenvs([H|T]) :-
 \+ deleted(H),
 testenvs(T).


% ==========================================================================
/*
PREDICATE: fivetypenode(+Context,+Nodes,?Solnode)
ARGUMENTS: Context,     list of environments (lists of Assumptions and Nodes)
           Nodes,       list of nodes numbers
           Solnode,     node number
COMMENTS:  Succeeds after instantiating Solnode to the node number of a node
           that includes room information (length = 5) and, either it is in
           Context, or it is direct consequence of a node in Context. In
           addition it must be in Nodes (timetable nodes)
*/
% ==========================================================================

fivetypenode(Context,Nodes,Solnode) :-
 member(Node,Context),
 node([Node|T]),
 ((T = [_,_,_,_],
   Solnode = Node)
 ;
  (T = [Subject,Day,Hour],
   node([Solnode,Subject,Day,Hour,_]))
 ),
 memberchk(Solnode,Nodes).


% ==========================================================================
/*
PREDICATE: user_info
ARGUMENTS: NONE
COMMENTS:  tells the user what is wrong between the actual timetable and
           constraints, from the information existing in ATMS, plus some
           advices about what to do, according to the 'tracks' recorded
           while modifying the timetable and constraints.
*/
% ==========================================================================

user_info :-
 unsolved(BadAsNodes),
 tracks(Tracks),
 ((BadAsNodes = [],
   !,
   nl,
   write('EVERYTHING is O.K.'),
   nl)
 ;
  (nl,
   write('ATMS info:'),
   nl,
   write('Assumptions that fail:'),
   nl,
   writeasnodeslist(BadAsNodes),
   write('My advices: move the following:'),
   nl,
   writeinfolist(Tracks),
   nl)
 ).

% ==========================================================================
/*
PREDICATE: findtracks(+BadAsNodes,?Nodedb)
ARGUMENTS: BadAsNodes, list [As|Nodes]
           Nodedb,     nodelist
COMMENTS:  succeeds if Nodedb is the "track" corresponding to the Nodes in
           BadAsNodes (see next predicate)
*/
% ==========================================================================

findtracks(BadAsNodes,Nodedb) :-
 member([_|Nodelist],BadAsNodes),
 track(Nodelist,Nodedb).


% ==========================================================================
/*
PREDICATE: track(+Nodes,?Nodedb)
ARGUMENTS: Nodes,       list of nodes numbers
           Nodedb,      nodelist
COMMENTS:  succeeds if Nodedb is the lowest level lecture from Nodes
*/
```

```
% ===========================================================================

track([Node],[Node|Rest]) :-
 !,
 node([Node|Rest]).

track([Node1|Nodelist],Nodedb) :-
 node([Node1,Subject1|Rest1]),
 track(Nodelist,[Node2,Subject2|Rest2]),
 subjects(Subjects),
 position(Subject1,Subjects,P1),
 position(Subject2,Subjects,P2),
 ((P1 > P2,
   !,
   Nodedb = [Node1,Subject1|Rest1])
 ;
  (Nodedb = [Node2,Subject2|Rest2])
 ).


% ===========================================================================
/*
PREDICATE: writeasnodeslist(+AsNodes)
ARGUMENTS: AsNodes, list of constraint (assumption) number plus nodes numbers
COMMENTS:  shows the unsatisfied constraint with the nodes that make it fail
*/
% ===========================================================================

writeasnodeslist([]).

writeasnodeslist([[As|Nodes]|BadAsNodes]) :-
 write('constraint: '),
 showconstraint(As),
 write('unsatisfied by nodes:'),
 nl,
 writenodelist(Nodes),
 nl,
 writeasnodeslist(BadAsNodes).


% ===========================================================================
/*
PREDICATE: showconstraint
ARGUMENTS: N, integer
COMMENTS:  shows the constraint whose asumption number is N
*/
% ===========================================================================

showconstraint(N) :-
 findconstraint(N,Pred,Arglist),
 writeassumptarglist(Pred,[Arglist]).


% ===========================================================================
/*
PREDICATE: history_info
ARGUMENTS: NONE
COMMENTS:  shows the history of the problem
*/
% ===========================================================================

history_info :-
 history(History),
 explainhistory,
 writelist(History).


% ===========================================================================
/*
PREDICATE: explainhistory
ARGUMENTS: NONE
COMMENTS:  explains the meaning of every action in the history of the problem
*/
% ===========================================================================

explainhistory :-
 nl,
 write('for each event in the history, you will be given a list whose'),
 nl,
 write('elements are: [Event,Node,Desc,Failed,Bad], where'),
 nl,
 write('Event  - may be either "solve" or "resolve" or "change"'),
 nl,
 write('Node   - is the solution node after the change'),
 nl,
 write('Desc   - is the description of the event. E.g. in case of "change"'),
 nl,
 write('         Desc will say if it was "move", "add" or "delete", plus'),
 nl,
 write('         subjects, days, hours, rooms, ... affected'),
 nl,
 write('Failed - is "yes" if the change was not possible, "no" otherwise'),
 nl,
 write('Bad    - is [] if problem is solved, and the list of unsatisfied'),
 nl,
 write('         constraints (and responsible nodes) numbers, otherwise'),
 nl.


% ===========================================================================
/*
PREDICATE: node_info(+N)
ARGUMENTS: N, integer
COMMENTS:  tells the user what the node N means, if such a node exists,
           either if it belongs to Timetable or not. It is also valid
           for non-constraint assumptions
*/
% ===========================================================================

node_info(N) :-
 Nodedb = [N|_],
 !,
 node(Nodedb),
 write('Node '),
 nodedb_info(Nodedb).

node_info(N) :-
 Nodedb = [N|_],
 !,
 assumpt(Nodedb),
 write('Assumption '),
 nodedb_info(Nodedb).

node_info(N) :-
 treenode(N),
 write(N),
 write(': is a Tree Node'),
 nl.
```

```
node_info(N) :-
 write(N),
 write(': is not in the actual database'),
 nl.


% ==========================================================================
/*
PREDICATE: number_info(+Number)
ARGUMENTS: Number, integer
COMMENTS:  tells the user what the node N means, if such a node exists,
           either if it belongs to Timetable or not. It is also valid
           for all kind of assumptions
*/
% ==========================================================================

number_info(Number) :-
 showconstraint(Number),
 !,
 write('It is a constraint'),
 nl.

number_info(Number) :-
 node_info(Number),
 nl.


% ==========================================================================
/*
PREDICATE: nodedb_info(+Nodedb)
ARGUMENTS: Nodedb, noderep
COMMENTS:  tells the user what the noderep Nodedb means, if such a node exists,
           either if it belongs to Timetable or not. It is also valid
           for non-constraint assumptions
*/
% ==========================================================================

nodedb_info(Nodedb) :-
 Nodedb = [N,Subject,Day,Hour|Tail],
 write(N),
 write(': "A lecture for subject '),
 write(Subject),
 write(' given on '),
 write(Day),
 write(' at '),
 write(Hour),
 ((Tail = [Room],
   write(' in room '),
   write(Room)
  )
 ;
  (Tail = [])
 ),
 write('"'),
 nl.


% ==========================================================================
/*
PREDICATE: writenodelist(+List)
ARGUMENTS: List, list of nodes numbers
COMMENTS:  tells the user what the node means, for each node in List
           either if it belongs to Timetable or not. It is also valid
           for non-constraint assumptions
*/
% ==========================================================================

writenodelist([]).

writenodelist([H|T]) :-
 node_info(H),
 writenodelist(T).


% ==========================================================================
/*
PREDICATE: writeinfolist(+List)
ARGUMENTS: List, list of nodereps
COMMENTS:  tells the user what the node means, for each node in List
           either if it belongs to Timetable or not. It is also valid
           for non-constraint assumptions
*/
% ==========================================================================

writeinfolist([]).

writeinfolist([H|T]) :-
 nodedb_info(H),
 writeinfolist(T).


% **************************************************************************
% *                                                                        *
% *                        LOAD AND SAVE OPTIONS                           *
% *                                                                        *
% **************************************************************************


% ==========================================================================
/*
PREDICATE: load_c(+CF)
ARGUMENTS: CF, file name
COMMENTS:  Succeeds after loading the constraints from the file NCF
*/
% ==========================================================================

load_c(CF) :-
 putwindow('WAIT'),
 [-CF],
 erasew(_).


% ==========================================================================
/*
PREDICATE: load_s(+CF)
ARGUMENTS: CF, file name
COMMENTS:  Succeeds after loading the constraints plus the whole
           environment from the file NCF
*/
% ==========================================================================

load_s(SF) :-
 putwindow('WAIT'),
 [-SF],
 subjects(Subjects),
 createlectrooms(Subjects),
 setof2(D,deleted(D),SetD),
 sendastoatms(SetD),
 setof2(W,constraintnumber(W),Set0),
 sendastoatms(Set0),
 setof2(X,assumptnumber(X),Set1),
 sendastoatms(Set1),
 setof2(Y,nodenumber(Y),Set2),
 sendnodestoatms(Set2),
 setof2(T,treenode(T),Set4),
 sendnodestoatms(Set4),
 setof2(Z,justification(Z),Set3),
 sendjusttoatms(Set3),
 asserta(activetree(tree)),
 asserta(now([])),
 ((notallowed(solve),
   idmenu(main,Mainmenu),          % Just for the user interface
   setmenuitem(Mainmenu,1,resolve))
 ;
  (true)
 ),
 erasew(_).


% ===========================================================================
/*
PREDICATE: save_c(+NCF)
ARGUMENTS: NCF, file name
COMMENTS:  Succeeds after writing the actual constraints in the file NCF
*/
% ===========================================================================

save_c(NCF) :-
 putwindow('WAIT'),
 tell(NCF),
 readall,
 told,
 erasew(_).


% ===========================================================================
/*
PREDICATE: save_s(+NCF)
ARGUMENTS: NCF, file name
COMMENTS:  Succeeds after writing the actual constraints, plus the actual
           environment in the file NCF, in order to recover it later.
*/
% ===========================================================================

save_s(NCF) :-
 ((notallowed(solve),
   !)
 ;
  (write('There is no environment yet. Only constraints'),
   nl,
   fail)
 ),
 putwindow('WAIT'),
 tell(NCF),
 readall,
 nl,
 nl,
 timetable(T),
 writeassumpt(timetable,T),
 nl,
```

```
 nl,
 tree(Tree),
 writeassumpt(tree,Tree),
 nl,
 nl,
 treesol(Sol),
 writeassumpt(treesol,Sol),
 nl,
 nl,
 ((part(Partsol),
   !,
   writeassumpt(part,Partsol),
   nl,
   nl)
 ;
  (true)
 ),
 bagof2(X,subtreearglist(X),Listtree),
 writeassumptarglist(subtree,Listtree),
 nl,
 nl,
 bagof2(Y,subsolarglist(Y),Listsol),
 writeassumptarglist(subsol,Listsol),
 nl,
 nl,
 bagof2([Not],notallowed(Not),Listnot),
 writeassumptarglist(notallowed,Listnot),
 nl,
 nl,
 solnode(Solnode),
 writeassumpt(solnode,Solnode),
 nl,
 nl,
 history(History),
 writeassumpt(history,History),
 nl,
 nl,
 unsolved(BadAs),
 writeassumpt(unsolved,BadAs),
 nl,
 nl,
 bagof2([Z0],treenode(Z0),List0),
 writeassumptarglist(treenode,List0),
 nl,
 nl,
 bagof2([Z1],nodenumber(Z1),List1),
 writeassumptarglist(nodenumber,List1),
 nl,
 nl,
 bagof2([Z2],assumptnumber(Z2),List2),
 writeassumptarglist(assumptnumber,List2),
 nl,
 nl,
 bagof2([Z3],constraintnumber(Z3),List3),
 writeassumptarglist(constraintnumber,List3),
 nl,
 nl,
 bagof2([Z31],newconsnumber(Z31),List31),
 writeassumptarglist(newconsnumber,List31),
 nl,
 nl,
 bagof2([Z32],deleted(Z32),List32),
 writeassumptarglist(deleted,List32),
 nl,
 nl,
 bagof2([Z4],justification(Z4),List4),
 writeassumptarglist(justification,List4),
 nl,
 nl,
 bagof2([Z5],node(Z5),List5),
 writeassumptarglist(node,List5),
 nl,
 nl,
 bagof2([Z6],assumpt(Z6),List6),
 writeassumptarglist(assumpt,List6),
 nl,
 nl,
 bagof2(Z8,historyTimetablearglist(Z8),List8),
 writeassumptarglist(historyTimetable,List8),
 told,
 erasew(_).


% ==========================================================================
/*
PREDICATE: readall
ARGUMENTS: NONE
COMMENTS:  suceeds after writing the actual constraints in the output device
*/
% ==========================================================================

readall :-
 subjects(Subjects),
 writeassumpt(subjects,Subjects),
 nl,
 nl,
 bagof2(X1,subjlecturesarglist(X1),List1),
 writeassumptarglist(subjlectures,List1),
 nl,
 nl,
 bagof2(X2,nonsimultarglist(X2),List2),
 writeassumptarglist(nonsimult,List2),
 nl,
 nl,
 bagof2(X3,nonfollowarglist(X3),List3),
 writeassumptarglist(nonfollow,List3),
 nl,
 nl,
 bagof2(X4,lectroomarglist(X4),List4),
 writeassumptarglist(lectroom,List4),
 nl,
 nl,
 bagof2(X5,fixarglist(X5),List5),
 writeassumptarglist(fix,List5),
 nl,
 nl,
 bagof2(X6,badarglist(X6),List6),
 writeassumptarglist(bad,List6),
 nl,
 nl,
 bagof2(X7,verybadarglist(X7),List7),
 writeassumptarglist(verybad,List7),
 nl,
 nl,
 bagof2(X8,notposarglist(X8),List8),
 writeassumptarglist(notpos,List8).
```

```
% ==========================================================================
/*
PREDICATE: writeassumpt(+Pred,+H)
ARGUMENTS: Pred, atom (predicate identifier)
           H,    predicate argument
COMMENTS:  Succeeds after writing in a 'prolog' manner the unary predicate
           pred whose only argument is H.
*/
% ==========================================================================

writeassumpt(Pred,H) :-
 write(Pred),
 write('('),
 write(H),
 write(').'),
 nl,
 nl.


% ==========================================================================
/*
PREDICATE: writeassumptarglist(+Pred,+List)
ARGUMENTS: Pred, atom (predicate identifier)
           List, list of lists of arguments
COMMENTS:  Succeeds after writing in a 'prolog' manner the unary predicate
           pred with all the different lists of arguments in List (valid
           for n-ary predicates and any number of facts)
*/
% ==========================================================================

writeassumptarglist(_,[]).

writeassumptarglist(Pred,[H|T]) :-
 write(Pred),
 write('('),
 writearg(H),
 write(').'),
 nl,
 nl,
 writeassumptarglist(Pred,T).


% ==========================================================================
/*
PREDICATE: writearg(+Arglist)
ARGUMENTS: Arglist, list of arguments
COMMENTS:  Succeeds after writing Arglist in a 'prolog' manner
*/
% ==========================================================================

writearg([H]) :-
 !,
 write(H).

writearg([H|T]) :-
 write(H),
 write(','),
 writearg(T).


% ==========================================================================
/*
PREDICATE: historyTimetablearglist(?Arglist)
ARGUMENTS: Arglist, list of arguments
```

```
COMMENTS:  Succeeds after instantiating Arglist to the list of arguments
           [A1,...,An] such that historyTimetable(A1,...,An)
*/
% ===========================================================================

historyTimetablearglist([Change,In,Subtree]) :-
 historyTimetable(Change,In,Subtree).


% ===========================================================================
/*
PREDICATE: subtreearglist(?Arglist)
ARGUMENTS: Arglist, list of arguments
COMMENTS:  Succeeds after instantiating Arglist to the list of arguments
           [A1,...,An] such that subtree(A1,...,An)
*/
% ===========================================================================

subtreearglist([In,Subtree]) :-
 subtree(In,Subtree).


% ===========================================================================
/*
PREDICATE: subsolarglist(?Arglist)
ARGUMENTS: Arglist, list of arguments
COMMENTS:  Succeeds after instantiating Arglist to the list of arguments
           [A1,...,An] such that subsol(A1,...,An)
*/
% ===========================================================================

subsolarglist([In,Sol]) :-
 subsol(In,Sol).


% ===========================================================================
/*
PREDICATE: subjlecturesarglist(?Arglist)
ARGUMENTS: Arglist, list of arguments
COMMENTS:  Succeeds after instantiating Arglist to the list of arguments
           [A1,...,An] such that subjlectures(A1,...,An)
*/
% ===========================================================================

subjlecturesarglist([Subject,N]) :-
 subjlectures(Subject,N).


% ===========================================================================
/*
PREDICATE: nonsimultarglist(?Arglist)
ARGUMENTS: Arglist, list of arguments
COMMENTS:  Succeeds after instantiating Arglist to the list of arguments
           [A1,...,An] such that nonsimult(A1,...,An)
*/
% ===========================================================================

nonsimultarglist([Subject1,Subject2,As]) :-
 nonsimult(Subject1,Subject2,As).


% ===========================================================================
/*
PREDICATE: nonfollowarglist(?Arglist)
ARGUMENTS: Arglist, list of arguments
COMMENTS:  Succeeds after instantiating Arglist to the list of arguments
           [A1,...,An] such that nonfollow(A1,...,An)
*/
% ===========================================================================

nonfollowarglist([Subject1,Subject2,As]) :-
 nonfollow(Subject1,Subject2,As).


% ===========================================================================
/*
PREDICATE: lectroomarglist(?Arglist)
ARGUMENTS: Arglist, list of arguments
COMMENTS:  Succeeds after instantiating Arglist to the list of arguments
           [A1,...,An] such that lectroom(A1,...,An)
*/
% ===========================================================================

lectroomarglist([Subject,Room,As]) :-
 lectroom(Subject,Room,As).


% ===========================================================================
/*
PREDICATE: fixarglist(?Arglist)
ARGUMENTS: Arglist, list of arguments
COMMENTS:  Succeeds after instantiating Arglist to the list of arguments
           [A1,...,An] such that fix(A1,...,An)
*/
% ===========================================================================

fixarglist([Subject,Time,As]) :-
 fix(Subject,Time,As).


% ===========================================================================
/*
PREDICATE: badarglist(?Arglist)
ARGUMENTS: Arglist, list of arguments
COMMENTS:  Succeeds after instantiating Arglist to the list of arguments
           [A1,...,An] such that bad(A1,...,An)
*/
% ===========================================================================

badarglist([Subject,Time]) :-
 bad(Subject,Time).


% ===========================================================================
/*
PREDICATE: verybadarglist(?Arglist)
ARGUMENTS: Arglist, list of arguments
COMMENTS:  Succeeds after instantiating Arglist to the list of arguments
           [A1,...,An] such that verybad(A1,...,An)
*/
% ===========================================================================

verybadarglist([Subject,Time]) :-
 verybad(Subject,Time).


% ===========================================================================
/*
```

```
PREDICATE: notposarglist(?Arglist)
ARGUMENTS: Arglist, list of arguments
COMMENTS:  Succeeds after instantiating Arglist to the list of arguments
           [A1,...,An] such that notpos(A1,...,An)
*/
% ===========================================================================

notposarglist([Subject,Time,As]) :-
 notpos(Subject,Time,As).



% ***************************************************************************
% *                                                                         *
% *                          RESOLVE OPTION                                 *
% *                                                                         *
% ***************************************************************************



% ===========================================================================
/*
PREDICATE: resolve
ARGUMENTS: NONE
COMMENTS:  Succeeds after removing the lectures in 'track's (inconsistent
           with the constraints) from the timetable, introducing new ones
           in it, automatically, creating a subtree for the new state, a
           solution path for it (subsol), and updating 'unsolved' and 'history'
*/
% ===========================================================================

resolve :-
 notallowed(resolve),
 !,
 write('option not available now'),
 nl,
 fail.

resolve :-
 tracks(InitTracks),
 fixremove(InitTracks,Tracks),
 Tracks = [_|_],
 timetable(Timetable),
 recremove(Timetable,Tracks,Timetable1),
 setof2(Z,inThen(Z,Tracks),SubjectsBefore),
 bagof2(Z,inThen(Z,Tracks),Bagbefore),
 subjects(Subjects),
 ordnonfix(SubjectsBefore,Subjects,OrdSubjects),
 times2(Bagbefore,OrdSubjects,OrdSubjectsTimes),
 recnlectures(OrdSubjectsTimes),
 subresolve(OrdSubjects,Timetable1,Newtimetable,Brothers,Fail),
 ((Fail = no,
   sol(L),
   retract(sol(L)))
 ;
  (Fail = yes,
   takepart(L))
 ),
 ((part(_),
   !,
   retract(part(_)))
 ;
  (true)
 ),
 retract(solnode(_)),
 dbtreenode(SolNode),
 extract(Newtimetable,Nodes),
 putjustification(SolNode,Nodes),
 asserta(solnode(SolNode)),
 findinconsist(BadAs),
 asserta(subsol(SolNode,L)),
 firsttreenode(In),
 asserta(subtree(SolNode,[[In,subtree]|Brothers])),
 retract(unsolved(_)),
 asserta(unsolved(BadAs)),
 retract(tracks(_)),
 setof2(X,findtracks(BadAs,X),NewTracks),
 asserta(tracks(NewTracks)),
 history(List),
 conc(List,[[resolve,SolNode,subtree,Fail,BadAs]],Newlist),
 retract(history(List)),
 asserta(history(Newlist)),
 asserta(historyTimetable(resolve,SolNode,Newtimetable)),
 showtimetable,
 user_info,
 ((BadAs = [],
   !)
 ;
  (write('Sorry. There is nothing else I can do.'),
   nl)
 ).

resolve :-
 write('Sorry. No non-fixed lectures bothering the constraints.'),
 nl.


% ===========================================================================
/*
PREDICATE: subresolve(+OrdSubjects,+Timetable,?Newtimetable,?Brothers,?Fail)
ARGUMENTS: OrdSubjects,  list (of subjects representations)
           Timetable,    list (of nodereps)
           Newtimetable, list (of nodereps)
           Brothers,     list (of trees representations)
           Fail,         yes or no
COMMENTS:  Succeeds after updating the timetable with the lectures from
           'tracks', now in a right place. If it is not possible, a
           message is produced in that sense
*/
% ===========================================================================

subresolve([HSubjects|TSubjects],Timetable,Newtimetable,Brothers,Fail) :-
 settimetable(HSubjects,TSubjects,[],Timetable,
              Newtimetable,Brothers,Fail),
 ((Fail = no,
   retract(timetable(_)),
   asserta(timetable(Newtimetable)),
   retractallnlectures)
 ;
  (Fail = yes,
   spoiled)
 ).


% ===========================================================================
/*
PREDICATE: inThen(?Subject,+List)
ARGUMENTS: Subject,      subject representation
           List,         list of noderep
```

```
COMMENTS:  Succeeds after instantiating subject to any subject in any noderep
           in List
*/
% ============================================================================

inThen(Subject,[[_,Subject|_]|_]).

inThen(Subject,[_|Then]) :-
 inThen(Subject,Then).


% ============================================================================
/*
PREDICATE: fixremove(+Nodedblist,?Nodedblist2)
ARGUMENTS: Nodedblist,  list of nodereps
           Nodedblist2, list of nodereps
COMMENTS:  Succeeds after instantiating Nodedblist2 to the list of nodereps
           from Nodedblist that are not fixed
*/
% ============================================================================

fixremove([],[]).

fixremove([[_,Subject|_]|IT],T) :-
 fix(Subject,_,_),
 !,
 fixremove(IT,T).

fixremove([H|IT],[H|T]) :-
 fixremove(IT,T).


% ============================================================================
/*
PREDICATE: ordnonfix(+SubjectsBefore,+Subjects,+OrdSubjects)
ARGUMENTS: SubjectsBefore, list of subjects representation
           Subjects,       list of subjects representation
           OrdSubjects,    list of subjects representation
COMMENTS:  Succeeds after instantiating OrdSubjects to the list of subjects
           from SubjectsBefore, in the order they appear in Subjects
*/
% ============================================================================

ordnonfix(_,[],[]).

ordnonfix(SubjectsBefore,[HSubjects|TSubjects],
           [HSubjects|SubjectsAfter]) :-
 memberchk(HSubjects,SubjectsBefore),
 !,
 ordnonfix(SubjectsBefore,TSubjects,SubjectsAfter).

ordnonfix(SubjectsBefore,[_|TSubjects],SubjectsAfter) :-
 ordnonfix(SubjectsBefore,TSubjects,SubjectsAfter).


% ============================================================================
/*
PREDICATE: times2(+Bag,+Set,?Settimes)
ARGUMENTS: Bag,      list
           Set,      list
           Settimes, list
COMMENTS:  Succeeds after instantiating Settimes to the list of "lists
           whose first element is an element in Set and whose second
           element is the number of times it appears in Bag"
*/
% ============================================================================

times2(_,[],[]).

times2(Bag,[HS|TS],[[HS,N]|TST]) :-
 times(HS,Bag,N),
 times2(Bag,TS,TST).


% ============================================================================
/*
PREDICATE: spoiled
ARGUMENTS: NONE
COMMENTS:  Succeeds after giving a message when no solution has been found
           with "resolve" option, after some - controversial - changes have
           been made.
*/
% ============================================================================

spoiled :-
 write('No solution has been found. Either'),
 nl,
 write(' (1) the constraints are unsatisfiable, or'),
 nl,
 write(' (2) some lectures are bothering "resolve" option strategy,'),
 nl,
 write('     and, either "solve" option can solve it, or yourself: changing'),
 nl,
 write('the constraints and timetable. To use "solve" option, save the'),
 nl,
 write('constraints (save_c), exit PROLOG, and "solve" again with them'),
 nl,
 write('from scratch'),
 nl,
 nl.



% ****************************************************************************
% *                                                                          *
% *                            DELETE OPTION                                 *
% *                                                                          *
% ****************************************************************************


% ============================================================================
/*
PREDICATE: delete(lectroom,[+Subject,+Room,+As],?Fail)
ARGUMENTS: lectroom, bound atom
           ONE LIST OF:
            Subject, subject representation
            Room,       room representation
            As,      integer
           Fail, yes or no
COMMENTS:  Succeeds after deleting a 'lectroom' constraint in the environment,
           if such a constraint exists, and it it does not cause problems to
           the timetable
*/
% ============================================================================

delete(lectroom,[Subject,Room,_],yes) :-
 timetable(Timetable),
 Nodedb = [_,Subject,_,_,Room],
 memberchk(Nodedb,Timetable),
 !,
 shout(Nodedb,'must be moved to other room first').

delete(lectroom,[Subject,Room,As],no) :-
 retract(lectroom(Subject,Room,As)),
 asserta(deleted(As)),
 retract(constraintnumber(As)),
 lectrooms(Subject,Rooms),
 remove(Rooms,Room,NewRooms),
 retract(lectrooms(Subject,Rooms)),
 asserta(lectrooms(Subject,NewRooms)).


% =========================================================================
/*
PREDICATE: delete(subject,[+Subject],?Fail)
ARGUMENTS: subject, bound atom
           ONE LIST OF:
            Subject, subject representation
           Fail, yes or no
COMMENTS:  Succeeds after deleting a subject in the timetable, and everything
           related with it
*/
% =========================================================================

delete(subject,[Subject],no) :-
 putwindow('WAIT'),
 timetable(Timetable),
 subjremove(Timetable,Subject,NewTimetable),
 retract(timetable(Timetable)),
 asserta(timetable(NewTimetable)),
 subjects(Subjects),
 remove(Subjects,Subject,NewSubjects),
 retract(subjects(Subjects)),
 asserta(subjects(NewSubjects)),
 lectrooms(Subject,Rooms),
 retractalllectroom(Subject),
 retract(lectrooms(Subject,Rooms)),
 ((fix(Subject,_,_),
   !,
   retractallfix(Subject))
 ;
  (retract(subjlectures(Subject,_)))
 ),
 retractallbad(Subject),
 retractallverybad(Subject),
 retractallnotpos(Subject),
 retractallnonsimult(Subject),
 retractallnonfollow(Subject),
 retractalloldnodes,
 erasew(_).

% =========================================================================
/*
PREDICATE: subjremove(+Timetable,+Subjects,?NewTimetable)
ARGUMENTS: Timetable,    list of noderep
           Subject,      subject representation
           NewTimetable, list of noderep
COMMENTS:  Succeeds after instantiating NewTimetable to the list of nodereps
           that are in timetable whose subject is not Subject

*/
```

```
% =========================================================================

subjremove([],Subject,[]) :-
 removeallnodes(Subject).

subjremove([Nodedb|T],Subject,NT) :-
 Nodedb = [_,Subject|_],
 !,
 retract(node(Nodedb)),
 asserta(oldnode(Nodedb)),
 subjremove(T,Subject,NT).

subjremove([H|T],Subject,[H|NT]) :-
 subjremove(T,Subject,NT).


% =========================================================================
/*
PREDICATE: delete(bad,[+Subject,+Day,+Hour],Fail)
ARGUMENTS: bad,           bound atom
           ONE LIST OF:
            Subject,      subject representation
            Day,              day representation
            Hour,            hour representation
           Fail, yes or no
COMMENTS:  Succeeds after deleting a 'bad' constraint according to the
           arguments above, if such a constraint exists
*/
% =========================================================================

delete(bad,[Subject,Day,Hour],no) :-
 retract(bad(Subject,[Day,Hour])).


% =========================================================================
/*
PREDICATE: delete(verybad,[+Subject,+Day,+Hour],?Fail)
ARGUMENTS: verybad,       bound atom
           ONE LIST OF:
            Subject,      subject representation
            Day,              day representation
            Hour,            hour representation
           Fail, yes or no
COMMENTS:  Succeeds after deleting a 'verybad' constraint according to the
           arguments above, if such a constraint exists
*/
% =========================================================================

delete(verybad,[Subject,Day,Hour],no) :-
 retract(verybad(Subject,[Day,Hour])).


% =========================================================================
/*
PREDICATE: delete(notpos,[+Subject,+Day,+Hour,+As],?Fail)
ARGUMENTS: notpos,        bound atom
           ONE LIST OF:
            Subject,      subject representation
            Day,              day representation
            Hour,            hour representation
            As,           integer
           Fail, yes or no
COMMENTS:  Succeeds after deleting a 'bad' constraint according to the
           arguments above, if such a constraint exists
```

```
*/
% ===========================================================================

delete(notpos,[Subject,Day,Hour,As],no) :-
 asserta(deleted(As)),
 retract(constraintnumber(As)),
 retract(notpos(Subject,[Day,Hour],As)).


% ===========================================================================
/*
PREDICATE: delete(nonfollow,[+Subject1,+Subject2,+As])
ARGUMENTS: nonfollow,    bound atom
           ONE LIST OF:
            Subject1,     subject representation
            Subject2,     subject representation
            As,           integer
           Fail, yes or no
COMMENTS:  Succeeds after deleting a 'nonfollow' constraint according to the
           arguments above, if such a constraint exists
*/
% ===========================================================================

delete(nonfollow,[Subject1,Subject2,As],no) :-
 asserta(deleted(As)),
 retract(constraintnumber(As)),
 retract(nonfollow(Subject1,Subject2,As)).


% ===========================================================================
/*
PREDICATE: delete(nonsimult,[+Subject1,+Subject2,+As])
ARGUMENTS: nonsimult,    bound atom
           ONE LIST OF:
            Subject1,     subject representation
            Subject2,     subject representation
            As,           integer
           Fail, yes or no
COMMENTS:  Succeeds after deleting a 'nonsimult' constraint according to the
           arguments above, if such a constraint exists
*/
% ===========================================================================

delete(nonsimult,[Subject1,Subject2,As],no) :-
 asserta(deleted(As)),
 retract(constraintnumber(As)),
 retract(nonsimult(Subject1,Subject2,As)).



% ***************************************************************************
% *                                                                         *
% *                              ADD OPTION                                 *
% *                                                                         *
% ***************************************************************************


% ===========================================================================
/*
PREDICATE: add(subject,[+Subject],?Fail)
ARGUMENTS: subject, bound atom
           ONE LIST OF:
            Subject, subject representation
           Fail, yes or no
```

```
COMMENTS:  Succeeds after introducing a new subject in the timetable, asking
           for the number of lectures, if they are fixed or not, and
           introducing them.
*/
% ===========================================================================

add(subject,[Subject],no) :-
 repeat,
 write('number of rooms available: '),
 read(R),
 integer(R),
 nl,
 !,
 readrooms(R,Subject,Rooms),
 asserta(lectrooms(Subject,Rooms)),
 ((notallowed(solve),
   !,
   write('I assume it is non-fixed with 0 lectures (timetable not modified)'),
   nl,
   write('You should add the constraints for it, and then, either "fix" or'),
   nl,
   write('"subjlectures" information to update the timetable with it'),
   nl,
   asserta(subjlectures(Subject,0)))
 ;
  (repeat,
   write('number of lectures: '),
   read(N),
   integer(N),
   nl,
   !,
   putwindow('fixed? '),
   idwindow(menu,Menuwindow),
   idmenu(yesno,Yesnomenu),
   showmenu(Menuwindow,Yesnomenu,F),
   erasew('fixed? '),
   ((F = 1,
     nl,
     !,
     readlist(N,DH,Aslist),
     recassertfix(Subject,DH,Aslist))
   ;
    (F = 2,
     nl,
     asserta(subjlectures(Subject,N)))
   ),
   write('no change has been performed in the timetable, but information has'),
   nl,
   write('been entered for further "solve"'),
   nl
  )
 ),
 subjects(Subjects),
 retract(subjects(Subjects)),
 ((Subjects = [],
   !,
   OrdSubjects = [Subject])
 ;
  (reorder(Subject,Subjects,OrdSubjects))
 ),
 asserta(subjects(OrdSubjects)).


% ===========================================================================
/*
PREDICATE: fixDHAs(+Subject,+List)
ARGUMENTS: Subject,   subject representation
           List,      list [[Day,Hour],List]
COMMENTS:  No comments: It is clear enough
*/
% ========================================================================

fixDHAs(Subject,[[Day,Hour],As]) :-
 fix(Subject,[Day,Hour],As).


% ========================================================================
/*
PREDICATE: sepfix(+List1,?List2,?List3)
ARGUMENTS: List1, list of lists [X,Y]
           List2, list
           List3, list
COMMENTS:  Succeeds after instantiating List2 and List3 to the list of
           X and Y from List1, respectively
*/
% ========================================================================

sepfix([],[],[]).

sepfix([[X,Y]|T],[X|T1],[Y|T2]) :-
 sepfix(T,T1,T2).


% ========================================================================
/*
PREDICATE: add(bad,[+Subject,+Day,+Hour],?Fail)
ARGUMENTS: bad,            bound atom
           ONE LIST OF:
            Subject,      subject representation
            Day,               day representation
            Hour,            hour representation
           Fail, yes or no
COMMENTS:  Succeeds after adding a 'bad' constraint according to the
           arguments above, removing eventual 'verybad' or 'notpos'
           related constraints
*/
% ========================================================================

add(bad,[Subject,Day,Hour],no) :-
 retractbadnotpos(Subject,Day,Hour),
 asserta(bad(Subject,[Day,Hour])).


% ========================================================================
/*
PREDICATE: add(verybad,[+Subject,+Day,+Hour],?Fail)
ARGUMENTS: verybad,        bound atom
           ONE LIST OF:
            Subject,      subject representation
            Day,               day representation
            Hour,            hour representation
           Fail, yes or no
COMMENTS:  Succeeds after adding a 'verybad' constraint according to the
           arguments above, removing eventual 'bad' or 'notpos'
           related constraints
*/
% ========================================================================
```

```
add(verybad,[Subject,Day,Hour],no) :-
 retractbadnotpos(Subject,Day,Hour),
 asserta(verybad(Subject,[Day,Hour])).


% ========================================================================
/*
PREDICATE: add(notpos,[+Subject,+Day,+Hour,+As],?Fail)
ARGUMENTS: notpos,         bound atom
           ONE LIST OF:
            Subject,      subject representation
            Day,               day representation
            Hour,            hour representation
            As,           integer
           Fail, yes or no
COMMENTS:  Succeeds after adding a 'notpos' constraint according to the
           arguments above, removing eventual 'bad' or 'verybad' conflicts,
           and testing if a node inconsistent with it is in Timetable.
           If it is fixed, notpos condition is not included. Otherwise,
           we keep track of that node and send info to ATMS.
*/
% ========================================================================

add(notpos,[Subject,Day,Hour,As],no) :-
 retractbadnotpos(Subject,Day,Hour),
 altconstraint(As),
 asserta(notpos(Subject,[Day,Hour],As)),
 ((node([Node,Subject,Day,Hour]),
   !,
   putjustification(0,[Node,As]))
 ;
   (true)
 ).


% ========================================================================
/*
PREDICATE: retractbadnotpos(+Subject,+Day,+Hour)
ARGUMENTS: Subject,      subject representation
           Day,               day representation
           Hour,            hour representation
COMMENTS:  Succeeds after removing all 'bad', 'verybad' or 'notpos'
           constraints according to the arguments above
*/
% ========================================================================

retractbadnotpos(Subject,Day,Hour) :-
 bad(Subject,[Day,Hour]),
 !,
 retract(bad(Subject,[Day,Hour])),
 retractbadnotpos(Subject,Day,Hour).

retractbadnotpos(Subject,Day,Hour) :-
 verybad(Subject,[Day,Hour]),
 !,
 retract(verybad(Subject,[Day,Hour])),
 retractbadnotpos(Subject,Day,Hour).

retractbadnotpos(Subject,Day,Hour) :-
 notpos(Subject,[Day,Hour],As),
 !,
 asserta(deleted(As)),
 retract(constraintnumber(As)),
 retract(notpos(Subject,[Day,Hour],As)),
 retractbadnotpos(Subject,Day,Hour).

retractbadnotpos(_,_,_).


% ==========================================================================
/*
PREDICATE: add(nonfollow,[+Subject1,+Subject2,+As])
ARGUMENTS: nonfollow,     bound atom
           ONE LIST OF:
            Subject1,      subject representation
            Subject2,      subject representation
            As,            integer
           Fail, yes or no
COMMENTS:  Succeeds after adding a 'nonfollow' constraint according to the
           arguments above
*/
% ==========================================================================

add(nonfollow,[Subject1,Subject2,As],no) :-
 asserta(nonfollow(Subject1,Subject2,As)),
 altconstraint(As),
 timetable(Timetable),
 find(nonfollow,Subject1,Subject2,As,Timetable).


% ==========================================================================
/*
PREDICATE: add(nonsimult,[+Subject1,+Subject2,+As])
ARGUMENTS: nonsimult,     bound atom
           ONE LIST OF:
            Subject1,      subject representation
            Subject2,      subject representation
            As,            integer
           Fail, yes or no
COMMENTS:  Succeeds after adding a 'nonsimult' constraint according to the
           arguments above
*/
% ==========================================================================

add(nonsimult,[Subject1,Subject2,As],no) :-
 asserta(nonsimult(Subject1,Subject2,As)),
 altconstraint(As),
 timetable(Timetable),
 find(nonsimult,Subject1,Subject2,As,Timetable).


% ==========================================================================
/*
PREDICATE: find(+Choice,+Subject1,+Subject2,+As,+Timetable)
ARGUMENTS: Choice,        nonsimult or nonfollow
           Subject1,      subject representation
           Subject2,      subject representation
           As,            integer
           Timetable,     list of noderep
COMMENTS:  performs, either nonfollow or nonsimult changes, depending on
           Choice: looks for nodes affected by the new nonfollow or nonsimult
           constraint, and updates tracks and ATMS according to it.
*/
% ==========================================================================

find(Choice,Subject1,Subject2,As,Timetable) :-
   Nodedb1 = [Node1,Subject1,Day,Hour1,_],
   Nodedb2 = [Node2,Subject2,Day,_,_],
```

```
   member(Nodedb1,Timetable),
   lookfor(Timetable,Day,Hour1,_,Then,After,Sides),
   ((Choice = nonfollow,
     member(Nodedb2,Sides))
   ;
    (Choice = nonsimult,
     member(Nodedb2,Then))
   ),
   putjustification(0,[Node1,Node2,As]),
   find(Choice,Subject1,Subject2,As,After).

find(_,_,_,_,_).


% ==========================================================================
/*
PREDICATE: add(lectroom,[+Subject,+Room,+As],?Fail)
ARGUMENTS: notpos,        bound atom
           ONE LIST OF:
            Subject,       subject representation
            Room,             room representation
            As,            integer
           Fail, yes or no
COMMENTS:  Succeeds after adding a 'lectroom' constraint according to the
           arguments above,
*/
% ==========================================================================

add(lectroom,[Subject,Room,As],no) :-
 altconstraint(As),
 asserta(lectroom(Subject,Room,As)),
 lectrooms(Subject,Rooms),
 retract(lectrooms(Subject,Rooms)),
 asserta(lectrooms(Subject,[Room|Rooms])).


% ==========================================================================
/*
PREDICATE: add(fix,[+Subject],?Fail)
ARGUMENTS: fix,           bound atom
           ONE LIST OF:
            Subject,       subject representation
           Fail, yes or no
COMMENTS:  Succeeds after performing the changes to introduce fixed lectures
           for Subject. All previous lectures of subject are removed, and,
           if they were not fixed, subjlectures constraint is removed, as
           well: USED FOR MOVING NON-FIXED -> FIXED.
*/
% ==========================================================================

add(fix,[Subject],Fail) :-
 fix(Subject,_,_),
 !,
 write('number of lectures: '),
 read(N),
 nl,
 readlist(N,DH,Aslist),
 setof2(X,fixDHAs(Subject,X),DHAslist),
 subjects(Subjects),
 retract(subjects(Subjects)),
 remove(Subjects,Subject,NextSubjects),
 reorder(Subject,NextSubjects,OrdSubjects),
 asserta(subjects(OrdSubjects)),
 timetable(Timetable),
 subjremove(Timetable,Subject,NewTimetable),
 retractallfix(Subject),
 rectestfix(Subject,NewTimetable,DH,Aslist,LastTimetable,Fail),
 ((Fail = yes,
   !,
   setof2(Y,oldnode(Y),Set),
   removeallnodes(Subject),
   recassertnode(Set),
   sepfix(DHAslist,DH1,Aslist1),
   recassertfix(Subject,DH1,Aslist1))
 ;
  (retract(timetable(Timetable)),
   asserta(timetable(LastTimetable)),
   recassertfix(Subject,DH,Aslist))
 ),
 retractalloldnodes.


add(fix,[Subject],Fail) :-
 subjlectures(Subject,OldN),
 write('number of lectures: '),
 read(N),
 nl,
 readlist(N,DH,Aslist),
 subjects(Subjects),
 retract(subjects(Subjects)),
 remove(Subjects,Subject,NextSubjects),
 reorder(Subject,NextSubjects,OrdSubjects),
 asserta(subjects(OrdSubjects)),
 timetable(Timetable),
 subjremove(Timetable,Subject,NewTimetable),
 retract(subjlectures(Subject,OldN)),
 rectestfix(Subject,NewTimetable,DH,Aslist,LastTimetable,Fail),
 ((Fail = yes,
   !,
   setof2(Y,oldnode(Y),Set),
   removeallnodes(Subject),
   recassertnode(Set),
   asserta(subjlectures(Subject,OldN)))
 ;
  (retract(timetable(Timetable)),
   asserta(timetable(LastTimetable)),
   recassertfix(Subject,DH,Aslist))
 ),
 retractalloldnodes.

% ========================================================================
/*
PREDICATE: rectestfix(+Subject,+Timetable,+DH,+Aslist,?LastTimetable,?Fail)
ARGUMENTS: Subject,       subject representation
           Timetable,     list of noderep
           DH,            list of [Day,Hour]
           Aslist,        list of assumption numbers
           LastTimetable, list of noderep
           Fail,          yes or no
COMMENTS:  Succeeds after instantiating LastTimetable to the result of adding
           to Timetable the new fixed lectures   (whose data are in Subject, DH
           and Aslist). Many TESTS must be done: first, 'notpos', and then,
           'nonsimult', 'nonfollow' and 'rooms conflicts'. LAST THREE TESTS
           are performed, against all existing lectures that may
           create conflicts.
           If There is a Failure, Fail is yes. If no failure is
           found in any new fixed lecture, Fail is no. NO ASSERTION OTHER THAN
                   'ATMS RELATED' IS DONE.
*/
% ========================================================================

rectestfix(_,Timetable,[],[],Timetable,no).

rectestfix(Subject,Timetable,[[Day,Hour]|DH],[As|Aslist],LastTimetable,Fail) :-
 Nodedb = [Node,Subject,Day,Hour],
 db(Nodedb,Node),
 putjustification(Node,[As]),
 ((notpos(Subject,[Day,Hour],As2),
   !,
   putjustification(0,[As,As2]),
   warning(['new node',Subject,Day,Hour,As],'notpos'),
   Failrec = no)
 ;
  (true)
 ),
 lookfor(Timetable,Day,Hour,_,Then,_,Sides),
 recsubtestchoices(Nodedb,Timetable,Then,Sides,
                   Nexttimetable,_,Failrec),
 ((Failrec = no,
   !,
   Fail = Newfail)
 ;
  (Fail = yes)
 ),
 rectestfix(Subject,Nexttimetable,DH,Aslist,LastTimetable,Newfail).


% ========================================================================
/*
PREDICATE: recassertfix(+Subject,+DH,+Aslist)
ARGUMENTS: Subject, subject representation
           DH,      list of [Day,Hour]
           Aslist,  list of As, Assumption numbers
COMMENTS:  Succeeds after asserting 'fix' (Subject,[Day,Hour],As) for every
           member of DH and Aslist, respectively.
*/
% ========================================================================

recassertfix(_,[],[]).

recassertfix(Subject,[[Day,Hour]|DH],[As|Aslist]) :-
 asserta(fix(Subject,[Day,Hour],As)),
 recassertfix(Subject,DH,Aslist).


% ========================================================================
/*
PREDICATE: recassertnode(+List)
ARGUMENTS: List,    subject representation
COMMENTS:  Succeeds after asserting 'node' for every
           member of List
*/
% ========================================================================

recassertnode([]).

recassertnode([H|T]) :-
 asserta(node(H)),
 recassertnode(T).
```

```
% ==========================================================================
/*
PREDICATE: add(subjlectures,[+Subject,+N],?Fail)
ARGUMENTS: subjlectures, bound atom
           ONE LIST OF:
            Subject,      subject representation
            N,            integer
           Fail, yes or no
COMMENTS:  Succeeds after asserting subjlectures(Subject,N), and performing
           the adequate changes in the timetable, according to this. If
           subject lectures were fixed, they are NOT FIXED ANY MORE, and
           all related 'fix' facts dissappear from the database.
           USED FOR MOVING FIXED -> NON-FIXED. If they
           were not fixed, but the new number N of lectures is different,
           either new lectures are entered or old ones are deleted (see
           'deletelectures' predicate down below) depending on N and 'Old N'
*/
% ==========================================================================

add(subjlectures,[Subject,N],Fail) :-
 setof(X,fixDHAs(Subject,X),DHAslist),
 !,
 retractallfix(Subject),
 retract(subjects(Subjects)),
 remove(Subjects,Subject,NextSubjects),
 reorder(Subject,NextSubjects,OrdSubjects),
 asserta(subjects(OrdSubjects)),
 timetable(Timetable),
 subjremove(Timetable,Subject,NewTimetable),
 asserta(subjlectures(Subject,0)),
 subjlectadd(Subject,0,N,NewTimetable,Fail),
 ((Fail = yes,
   !,
   setof2(Y,oldnode(Y),Set),
   removeallnodes(Subject),
   recassertnode(Set),
   sepfix(DHAslist,DH,Aslist),
   recassertfix(Subject,DH,Aslist))
 ;
  (write('WARNING: lectures of this subject are NOT FIXED ANY MORE ...'),
   nl,
   write('They have been rearranged according to the actual database'),
   nl)
 ),
 retractalloldnodes.


add(subjlectures,[Subject,N],Fail) :-
 subjlectures(Subject,OldN),
 !,
 timetable(Timetable),
 subjlectadd(Subject,OldN,N,Timetable,Fail).


% ==========================================================================
/*
PREDICATE: subjlectadd(+Subject,+OldN,+N,+Timetable,?Fail)
ARGUMENTS: Subject,    subject representation
           OldN,       integer
           N,          integer
           Timetable,  list of noderep
           Fail,       yes or no
COMMENTS:  Succeeds after performing the adequate changes in the timetable,
           knowing that there were OldN lectures of Subject in it, and there
           must be N. Three cases appear: adding new lectures, removing
           existing ones or performing no change.
*/
% ==========================================================================

subjlectadd(Subject,OldN,N,Timetable,Fail) :-
 NewN is N - OldN,
 ((NewN = 0,
   Fail = no,
   nl)
 ;
  (NewN < 0,
   Fail = no,
   deletelectures(NewN,Subject,Timetable,Newtimetable),
   retract(timetable(_)),
   asserta(timetable(Newtimetable)))
 ;
  (NewN > 0,
   asserta(nlectures(Subject,NewN)),
   settimetable(Subject,[],[],Timetable,
                Newtimetable,_,Fail),
   ((Fail = yes,
     spoiled)
   ;
    (Fail = no,
     retract(timetable(_)),
     asserta(timetable(Newtimetable)),
     retract(nlectures(Subject,_)),
     retract(subjlectures(Subject,_)),
     asserta(subjlectures(Subject,N)))
   )
  )
 ).


% ==========================================================================
/*
PREDICATE: deletelectures(+NewN,+Subject,+Timetable,?Newtimetable)
ARGUMENTS: NewN,          integer (negative difference between N and OldN)
           Subject,       subject representation
           Timetable,     list of noderep
           Newtimetable,  list of noderep
COMMENTS:  Succeeds after performing the work for the second case in
           'sublectadd' predicate before: asking which lectures to release,
           and removing from Timetable, leaving the result in NewTimetable.
*/
% ==========================================================================

deletelectures(0,_,Timetable,Timetable).

deletelectures(N,Subject,Timetable,NewTimetable) :-
 N1 is N + 1,
 idwindow(menu,Menuwindow),
 repeat,
 write('Which lecture do you want to delete: '),
 nl,
 daysmenu(Menuwindow,Day),
 hoursmenu(Menuwindow,Hour),
 lookfor(Timetable,Day,Hour,Before,Then,After,_),
 Nodedb = [_,Subject,Day,Hour,_],
 ((memberchk(Nodedb,Then),
   !)
 ;
```

```
  (write('No such lecture. Try again'),
   nl,
   fail)
 ),
 !,
 remove(Then,Nodedb,Then1),
 conc(Then1,After,After1),
 conc(Before,After1,Timetable1),
 deletelectures(N1,Subject,Timetable1,NewTimetable).


% ****************************************************************************
% *                                                                          *
% *                               MOVE OPTION                                *
% *                                                                          *
% ****************************************************************************


% ============================================================================
/*
PREDICATE: move([+Subject,+Day,+Hour,+Room,+Day2,+Hour2,+Room2],?Fail)
ARGUMENTS: ONE LIST OF:
            Subject, subject representation
            Day,         day representation
            Hour,       hour representation
            Room,       room representation
            Day2,        day representation
            Hour2,      hour representation
            Room2,      room representation
           Fail, yes or no
COMMENTS:  Succeeds after setting a lecture for Subject
           at Day2,Hour2,Room2, if no 'same room'
           constraints avoid it (otherwise, the error is reported and the
           timetable is not modified.
           Possible room conflicts are sent to 'tracks' and ATMS, so that a
           further consultation will allow the user see the mistake
           Any attempt to change FIXED lectures (except change of room)
           or non-existing lectures,
           has been detected previously and avoided as well: input validity
           test is done in the template environment
*/
% ============================================================================

move([Subject,Day,Hour,Room,Day,Hour,Room2],no) :-
 !,
 timetable(Timetable),
 Nodedb = [_,Subject,Day,Hour,Room],
 Nodedb2 = [Node2,Subject,Day,Hour],
 db(Nodedb2,Node2),
 Nodedb3 = [Node3,Subject,Day,Hour,Room2],
 db(Nodedb3,Node3),
 dbnode(Nodedb3),
 lookfor(Timetable,Day,Hour,Before,Then,After,_),
 remove(Then,Nodedb,Then1),
 conc([Nodedb3|Then1],After,After1),
 conc(Before,After1,Timetable1),
 retract(timetable(Timetable)),
 asserta(timetable(Timetable1)).

move([Subject,Day,Hour,Room,Day2,Hour2,Room2],Fail) :-
 timetable(Timetable),
 Nodedb = [_,Subject,Day,Hour,Room],
 remove(Timetable,Nodedb,Timetable2),
```

```
 moveaux(Timetable2,Subject,Day2,Hour2,Room2,Fail).


% ============================================================================
/*
PREDICATE: moveaux(+Timetable2,+Subject,+Day2,+Hour2,+Room2,?Fail)
ARGUMENTS: Timetable2, list of nodereps
           Subject,    subject representation
           Day2,       day representation
           Hour2,      hour representation
           Room2,      room representation
           Fail,       yes or no
COMMENTS:  Succeeds after setting a lecture for Subject
           at Day2,Hour2,Room2, if no 'same room'
           constraints avoid it (otherwise, the error is reported and the
           timetable is not modified.
           Possible room conflicts are sent to 'tracks' and ATMS, so that a
           further consultation will allow the user see the mistake
*/
% ============================================================================

moveaux(Timetable2,Subject,Day2,Hour2,Room2,Fail) :-
 Nodedb2 = [Node2,Subject,Day2,Hour2],
 db(Nodedb2,Node2),
 dbass(Nodedb2),
 Nodedb3 = [Node3,Subject,Day2,Hour2,Room2],
 db(Nodedb3,Node3),
 dbnode(Nodedb3),
 lookfor(Timetable2,Day2,Hour2,Before,Then,After,Sides),
 ((notpos(Subject,[Day2,Hour2],As),
   putjustification(0,[Node2,As]),
   warning(Nodedb3,'notpos'))
  ;
  (true)
 ),
 recsubtest(Nodedb2,Then,Sides,_,Nodesub,Assub,Subfail),
 ((Subfail = yes,
   !,
   ((Nodesub = 0,
     Assub = 0,
     !,
     shout(Nodedb2,'same subject, same time'),
     Fail = yes)
    ;
    (warning(Nodedb2,'nonfollow and/or nonsimult'),
     Fail = no)
   )
  )
  ;
  (Fail = no)
 ),
 ((Fail = no,
   !,
   nl,
   retract(timetable(_)),
   conc([Nodedb3|Then],After,After1),
   conc(Before,After1,Timetable1),
   asserta(timetable(Timetable1)))
  ;
  (true)
 ).


% ============================================================================
/*
PREDICATE: shout(+Nodedb,+Message)
ARGUMENTS: Nodedb,  noderep
           Message, string
COMMENTS:  Succeeds after writing an explanation why Nodedb is not
           in accordance with the timetable and constraints
*/
% ============================================================================

shout(Nodedb,Message) :-
 nodedb_info(Nodedb),
 write(' has produced a failure:'),
 nl,
 write('Reason: '),
 write(Message),
 nl,
 write('CHANGE NOT ASSERTED'),
 nl.


% ============================================================================
/*
PREDICATE: warning(+Nodedb,+Message)
ARGUMENTS: Nodedb,  noderep
           Message, string
COMMENTS:  Succeeds after writing an explanation why Nodedb is not
           in accordance with the timetable and constraints
*/
% ============================================================================

warning(Nodedb,Message) :-
 write('WARNING! '),
 nodedb_info(Nodedb),
 write('will not follow the constraints:'),
 nl,
 write('Reason: '),
 write(Message),
 nl.



% ****************************************************************************
% *                                                                          *
% *                          SHOW TIMETABLES                                 *
% *                                                                          *
% ****************************************************************************


% ============================================================================
/*
PREDICATE: maxchar(+Item,?N)
ARGUMENTS: Item, 'subject' or 'room',
           N,    integer,
COMMENTS:  Succeeds after instantiating N to the number of characters reserved
           for writing 'Item' elements in the layout.
*/
% ============================================================================

maxchar(subject,7).

maxchar(room,6).


% ============================================================================
```

```
/*
PREDICATE: shtimetable(+Timetable)
ARGUMENTS: Timetable, list of nodereps
COMMENTS:  Succeeds after producing a layout of Timetable
*/
% ============================================================================

shtimetable(Timetable) :-
 hours([Minhour|_]),
 Soonhour is Minhour - 2,
 dayshours(DH),
 days([_|Days]),
 daystimetables(Days,Soonhour,Timetable,Timetables),
 writeheadings,
 writetimetable(DH,Timetables,[],[],_,_).


% ============================================================================
/*
PREDICATE: showtimetable
ARGUMENTS: NONE
COMMENTS:  Succeeds after producing a layout of the actual Timetable
*/
% ============================================================================

showtimetable :-
 timetable(Timetable),
 shtimetable(Timetable).


% ============================================================================
/*
PREDICATE: showtimetable(+SolNode)
ARGUMENTS: SolNode, node number
COMMENTS:  Succeeds after producing a layout of 'SolNode' Timetable
*/
% ============================================================================

showtimetable(Node) :-
 historyTimetable(Action,Node,Timetable),
 write('This was the timetable after '),
 write(Action),
 write(' had been done'),
 nl,
 shtimetable(Timetable).


% ============================================================================
/*
PREDICATE: snaptimetable(+Timetable,+File)
ARGUMENTS: Timetable, list of nodereps
           File,      file name
COMMENTS:  Succeeds after producing a layout of Timetable in the file File
*/
% ============================================================================

snaptimetable(Timetable,File) :-
 tell(File),
 showtimetable(Timetable),
 told.


% ============================================================================
/*
```

```
PREDICATE: snaptimetable(+File)
ARGUMENTS: Timetable, list of nodereps
           File,      file name
COMMENTS:  Succeeds after producing a layout of Timetable in the file File
*/
% ============================================================================

snaptimetable(File) :-
 timetable(Timetable),
 tell(File),
 showtimetable(Timetable),
 told.


% ============================================================================
/*
PREDICATE: snapshottimetables(+File)
ARGUMENTS: File,      file name
COMMENTS:  Succeeds after producing a layout of all "historical" Timetables
           in the file File
*/
% ============================================================================

snapshottimetables(File) :-
 history(History),
 tell(File),
 write('TIMETABLES HISTORY'),
 nl,
 nl,
 explainhistory,
 recsnaphisttimetable(History),
 told.


% ============================================================================
/*
PREDICATE: recsnaphisttimetable(+History)
ARGUMENTS: History, list
COMMENTS:  Succeeds after writing the corresponding timetable for each Event
           in History
*/
% ============================================================================

recsnaphisttimetable([]).

recsnaphisttimetable([H|T]) :-
 H = [Action,Node|_],
 nl,
 write(H),
 nl,
 nl,
 historyTimetable(Action,Node,Timetable),
 shtimetable(Timetable),
 nl,
 recsnaphisttimetable(T).


% ============================================================================
/*
PREDICATE: daystimetables(+Days,+Soonhour,+Timetable,?Timetables)
ARGUMENTS: Days,        list of days representations
           Soonhour,    integer
           Timetable,   list of noderep
           Timetables,  list of lists of noderep
COMMENTS:  Succeeds after instantiating Timetables to the list of
           timetables for each different day (useful for 'showtimetable')
*/
% ============================================================================

daystimetables([HDays|TDays],Soonhour,Timetable,[Before|TTimetables]) :-
 lookfor(Timetable,HDays,Soonhour,Before,_,After,_),
 daystimetables(TDays,Soonhour,After,TTimetables).

daystimetables([],_,Timetable,[Timetable]).


% ============================================================================
/*
PREDICATE: writetimetable(+DH,+Timetables,+Any,+PastDayshours,?NT,?Next)
ARGUMENTS: DH,             list of [Day,Hour]
           Timetables,     list of lists of noderep
           Any,            list of 0's and 1's
           PastDayshours,  list of [Day,Hour]
           NT,             list of noderep
           Next,           list of noderep
COMMENTS:  This is a 'clever and tricky' predicate that succeeds after
           producing the timetable layout. The way it works is looking
           at the same HOUR for every day, 'again and again' until no
           more lectures at this HOUR are left, before attempting the
           following. Hints:
           - Any holds 1 for any lecture found at a particular day (same HOUR)
             in the 'round before', 0 for non-found lectures. At the end of
             the days (Timetables = []), We test if Any has a 1 (then, we
             continue with the same HOUR: new round), with the 'next'
             timetables. Otherwise, we jump to the following hour.
           - PastDayshours keeps [Day,Hour] of the previous round, in order
             to recover it, in case of attempting same HOUR again.
           - The 'next' timetables information is recorded in 'NT', and
             passed to 'Next', when Any is empty again (new line, new round).
           - 'Next' is then, passed recursively as the new 'Timetables'
             information
*/
% ============================================================================

writetimetable([[Day,Hour]|TDH],[HTime|TTime],Any,PastDayshours,NT,Next) :-
 NT = [NHTime|NTTime],
 maxchar(subject,MCS),
 maxchar(room,MCR),
 ((Any = [],
   writeright(Hour,2),
   write('|'),
   Next = NT)
 ;
  (true)
 ),
 ((HTime = [[_,Subject,Day,Hour,Room]|THTime],
   !,
   NHTime = THTime,
   writeright(Subject,MCS),
   write(' '),
   writeright(Room,MCR),
   write('|'),
   Any2 = [1|Any])
 ;
  (NHTime = HTime,
   writeright(' ',MCS),
   write(' '),
   writeright(' ',MCR),
   write('|'),
   Any2 = [0|Any])
 ),
 writetimetable(TDH,TTime,Any2,[[Day,Hour]|PastDayshours],NTTime,Next).

writetimetable(TDH,[],Any,PastDayshours,[],Next) :-
 nl,
 ((memberchk(1,Any),
   !,
   reverse(PastDayshours,HDH),
   conc(HDH,TDH,DH),
   writetimetable(DH,Next,[],[],_,_))
 ;
  (writedash,
   writetimetable(TDH,Next,[],[],_,_))
 ).

writetimetable([],_,[],[],[],[]).


% ============================================================================
/*
PREDICATE: writeheadings
ARGUMENTS: NONE
COMMENTS:  Succeeds after writing the heading of the timetable layout
*/
% ============================================================================

writeheadings :-
 writedash,
 write('   |'),
 days(Days),
 writedays(Days),
 nl,
 writedash.


% ============================================================================
/*
PREDICATE: writedays(+Days)
ARGUMENTS: Days, list of days representations
COMMENTS:  Succeeds after writing the days in a right format, in the heading
           of the timetable layout
*/
% ============================================================================

writedays([]).

writedays([HDays|TDays]) :-
 maxchar(subject,MCS),
 maxchar(room,MCR),
 spaces(MCR,Spaceslist),
 name(Spaces,Spaceslist),
 write(Spaces),
 writeright(HDays,MCS),
 write(' |'),
 writedays(TDays).


% ============================================================================
/*
PREDICATE: writedash
ARGUMENTS: NONE
COMMENTS:  Succeeds after writing a dashes line, to appear in
           the timetable layout
*/
% ============================================================================

writedash :-
 write(----------------------------------------),
 write(----------------------------------------),
 nl.


% ============================================================================
/*
PREDICATE: writeright(+X,+N)
ARGUMENTS: X, atom
           N, integer
COMMENTS:  Succeeds after writing X with N characters (either truncating it
           or ading blanks).
*/
% ============================================================================

writeright(X,N) :-
 name(X,List),
 formatword(List,N,Formlist),
 name(Y,Formlist),
 write(Y).


% ============================================================================
/*
PREDICATE: formatword(+List,+N,?Formlist)
ARGUMENTS: List,     list
           N,        integer
           Formlist, list
COMMENTS:  Succeeds after instantiating Formlist to the list representing
           List 'name' with N positions, in the sense expressed in
           'writeright'
*/
% ============================================================================

formatword(List,N,Formlist) :-
 length(List,L),
 Dif is N - L,
 ((Dif > 0,
   !,
   spaces(Dif,Spaces),
   conc(List,Spaces,Formlist))
 ;
  (pickup(List,N,Formlist))
 ).


% ============================================================================
/*
PREDICATE: pickup(+List,+N,?Formlist)
ARGUMENTS: List,     list
           N,        integer
           Formlist, list
COMMENTS:  Succeeds after instantiating Formlist to the list containing the
           first N items in List
*/
% ============================================================================

pickup(_,0,[]) :-
 !.
```

```
pickup([H|List],N,[H|Formlist]) :-
 N1 is N - 1,
 pickup(List,N1,Formlist).



% ***************************************************************************
% *                                                                         *
% *                           GRAPHIC INTERFACE                             *
% *                                                                         *
% ***************************************************************************


% ===========================================================================
/*
PREDICATE: template
ARGUMENTS: NONE
COMMENTS:  Succeeds after creating the initial windows, images and menus
           that will be used in the "user-interface", and starting the
           session
*/
% ===========================================================================

template :-
 makeimage(414,900,Image),
 readimagebinary(Image,'paint.lmg'),
 makewindow('MENUS',0,750,394,900,Menuwindow),
 copy(Menuwindow,0,0,414,900,set,Image,260,10),
 asserta(idwindow(menu,Menuwindow)),
 days(Days),
 hours(Hours),
 atomize(Hours,Atomhours),
 makemenu(['yes/no:',yes,no],Yesnomenu),
 asserta(idmenu(yesno,Yesnomenu)),
 makemenu(['day:'|Days],Daysmenu),
 asserta(idmenu(days,Daysmenu)),
 makemenu(['hour:'|Atomhours],Hoursmenu),
 asserta(idmenu(hours,Hoursmenu)),
 makemenu(['main options:','solve','show the timetable','user info',
  'move lectures','add constraints','delete constraints','save','load',
  'help','history options','debugging/tree options','exit'],Mainmenu),
 asserta(idmenu(main,Mainmenu)),
 makemenu(['add:',subject,bad,verybad,notpos,nonsimult,nonfollow,lectroom,
  fix,subjlectures],Addmenu),
 asserta(idmenu(add,Addmenu)),
 makemenu(['delete:',subject,bad,verybad,notpos,nonsimult,nonfollow,lectroom],
  Deletemenu),
 asserta(idmenu(delete,Deletemenu)),
 makemenu(['save:','only constraints','whole environment'],Savemenu),
 asserta(idmenu(save,Savemenu)),
 makemenu(['load:','only constraints','whole environment'],Loadmenu),
 asserta(idmenu(load,Loadmenu)),
 makemenu(['help:','main','move/add/delete','constraints','tree',
          'history/info'],Helpmenu),
 asserta(idmenu(help,Helpmenu)),
 makemenu(['history:','show the history','snapshot of timetables history',
  'see a particular timetable','meaning of a number'],Histmenu),
 asserta(idmenu(hist,Histmenu)),
 makemenu(['tree:',puttree,snapshot,display,down,up,top,solution],
          Treemenu),
 asserta(idmenu(tree,Treemenu)),
 mainmenu.
```

```
% ===========================================================================
/*
PREDICATE: atomize(+NList,?List)
ARGUMENTS: Nlist, list of numbers
           List, list of atoms
COMMENTS:  Succeeds after instantiating List to the list whose elements are
           the atoms corresponding to numbers in Nlist (adding a space to
           the left), as it is necessary in this menus utility
*/
% ===========================================================================

atomize([],[]).

atomize([NH|NT],[AH|AT]) :-
 name(NH,List),
 name(AH,[32|List]),
 atomize(NT,AT).


% ===========================================================================
/*
PREDICATE: mainmenu
ARGUMENTS: NONE
COMMENTS:  Succeeds after setting the Main Menu, waiting for a correct
           response and reacting as required.
*/
% ===========================================================================

mainmenu :-
 idwindow(menu,Menuwindow),
 idmenu(main,Mainmenu),
 repeat,
 showmenu(Menuwindow,Mainmenu,N),
 choosemainmenu(N),
 N = 12,
 !.


% ===========================================================================
/*
PREDICATE: putwindow(+X)
ARGUMENTS: X, string
COMMENTS:  Succeeds after writing the string, formatted to 8 characters,
           in a predefined position of the Menus Window
*/
% ===========================================================================

putwindow(X) :-
 idwindow(menu,Menuwindow),
 name(X,List),
 formatword(List,8,Formlist),
 name(Y,Formlist),
 drawtext(Menuwindow,12,22,set,Y).


% ===========================================================================
/*
PREDICATE: erase
ARGUMENTS: X, string
COMMENTS:  Succeeds after writing seven dots and a blank,
           in a predefined position of the Menus Window
*/
% ===========================================================================
erasew(_) :-
 idwindow(menu,Menuwindow),
 drawtext(Menuwindow,12,22,set,'....... ').


% ===========================================================================
/*
PREDICATE: choosemainmenu(?N)
ARGUMENTS: N, integer
COMMENTS:  Succeeds after having reacted according to the selected option N.
*/
% ===========================================================================

choosemainmenu(1) :-
 notallowed(solve),
 putwindow('WAIT'),
 resolve,
 erasew('WAIT'),
 !.

choosemainmenu(1) :-
 \+ notallowed(solve),
 putwindow('WAIT'),
 solve,
 erasew('WAIT'),
 idmenu(main,Mainmenu),
 setmenuitem(Mainmenu,1,resolve),
 !.

choosemainmenu(2) :-
 showtimetable,
 !.

choosemainmenu(3) :-
 user_info,
 !.

choosemainmenu(4) :-
 move,
 !.

choosemainmenu(5) :-
 add,
 !.

choosemainmenu(6) :-
 delete,
 !.

choosemainmenu(7) :-
 savemenu,
 !.

choosemainmenu(8) :-
 ((notallowed(solve),
   !,
   write('It is undesirable to load over the actual environment'),
   nl,
   write('If you need it, exit prolog, restart the system, and then, load'),
   nl,
   fail)
 ;
  (true)
```

```
 ),
 loadmenu,
 !.

choosemainmenu(9) :-
 helpmenu,
 !.

choosemainmenu(10) :-
 historymenu,
 !.

choosemainmenu(11) :-
 treemenu,
 !.

choosemainmenu(12) :-
 putwindow('sure?'),
 idwindow(menu,Menuwindow),
 idmenu(yesno,Yesnomenu),
 showmenu(Menuwindow,Yesnomenu,N),
 erasew('sure?'),
 !,
 N = 1.


% ===========================================================================
/*
PREDICATE: savemenu
ARGUMENTS: NONE
COMMENTS:  Succeeds after setting the Save Menu, waiting for a correct
           response and reacting as required.
*/
% ===========================================================================

savemenu :-
 idwindow(menu,Menuwindow),
 idmenu(save,Savemenu),
 repeat,
 showmenu(Menuwindow,Savemenu,N),
 choosesavemenu(N),
 !.


% ===========================================================================
/*
PREDICATE: choosesavemenu(?N)
ARGUMENTS: N, integer
COMMENTS:  Succeeds after having reacted according to the selected option N.
*/
% ===========================================================================

choosesavemenu(0).

choosesavemenu(1) :-
 write('filename: '),
 read(Filename),
 nl,
 save_c(Filename).

choosesavemenu(2) :-
 write('filename: '),
 read(Filename),
 nl,
 save_s(Filename).


% ==========================================================================
/*
PREDICATE: loadmenu
ARGUMENTS: NONE
COMMENTS:  Succeeds after setting the Load Menu, waiting for a correct
           response and reacting as required.
*/
% ==========================================================================

loadmenu :-
 idwindow(menu,Menuwindow),
 idmenu(load,Loadmenu),
 repeat,
 showmenu(Menuwindow,Loadmenu,N),
 chooseloadmenu(N),
 !.


% ==========================================================================
/*
PREDICATE: chooseloadmenu(?N)
ARGUMENTS: N, integer
COMMENTS:  Succeeds after having reacted according to the selected option N.
*/
% ==========================================================================

chooseloadmenu(0).

chooseloadmenu(1) :-
 write('filename: '),
 read(Filename),
 nl,
 load_c(Filename).

chooseloadmenu(2) :-
 write('filename: '),
 read(Filename),
 nl,
 load_s(Filename).


% ==========================================================================
/*
PREDICATE: helpmenu
ARGUMENTS: NONE
COMMENTS:  Succeeds after setting the Help Menu, waiting for a correct
           response and reacting as required.
*/
% ==========================================================================

helpmenu :-
 idwindow(menu,Menuwindow),
 idmenu(help,Helpmenu),
 repeat,
 showmenu(Menuwindow,Helpmenu,N),
 choosehelpmenu(N),
 !.


% ==========================================================================
/*
```

```
PREDICATE: choosehelpmenu(?N)
ARGUMENTS: N, integer
COMMENTS:  Succeeds after having reacted according to the selected option N.
*/
% ==========================================================================

choosehelpmenu(0).

choosehelpmenu(1) :-
 help.

choosehelpmenu(2) :-
 help_add_delete.

choosehelpmenu(3) :-
 help_cons.

choosehelpmenu(4) :-
 help_tree.

choosehelpmenu(5) :-
 help_user.


% ==========================================================================
/*
PREDICATE: historymenu
ARGUMENTS: NONE
COMMENTS:  Succeeds after setting the History Menu, waiting for a correct
           response and reacting as required.
*/
% ==========================================================================

historymenu :-
 ((history(_),
   !)
  ;
   (write('history will not begin until "solve"'),
    fail)
 ),
 idwindow(menu,Menuwindow),
 idmenu(hist,Historymenu),
 repeat,
 showmenu(Menuwindow,Historymenu,N),
 choosehistorymenu(N),
 !.


% ==========================================================================
/*
PREDICATE: choosehistoryemenu(?N)
ARGUMENTS: N, integer
COMMENTS:  Succeeds after having reacted according to the selected option N.
*/
% ==========================================================================

choosehistorymenu(0).

choosehistorymenu(1) :-
 history_info.

choosehistorymenu(2) :-
 write('filename: '),
 read(File),
 nl,
 snapshottimetables(File).

choosehistorymenu(3) :-
 write('identifier node: '),
 read(Solnode),
 nl,
 showtimetable(Solnode).

choosehistorymenu(4) :-
 write('assumption or node number: '),
 read(Number),
 nl,
 number_info(Number).


% ============================================================================
/*
PREDICATE: treemenu
ARGUMENTS: NONE
COMMENTS:  Succeeds after setting the Tree Menu, waiting for a correct
           response and reacting as required.
*/
% ============================================================================

treemenu :-
 ((tree(_),
   !)
 ;
  (write('there is no tree yet'),
   fail)
 ),
 idwindow(menu,Menuwindow),
 idmenu(tree,Treemenu),
 repeat,
 showmenu(Menuwindow,Treemenu,N),
 choosetreemenu(N),
 !.


% ============================================================================
/*
PREDICATE: choosetreemenu(?N)
ARGUMENTS: N, integer
COMMENTS:  Succeeds after having reacted according to the selected option N.
*/
% ============================================================================

choosetreemenu(0).

choosetreemenu(1) :-
 write('identifier node: '),
 read(Solnode),
 nl,
 puttree(Solnode).

choosetreemenu(2) :-
 write('filename: '),
 read(File),
 nl,
 snapshot(File).

choosetreemenu(3) :-
 display.
```

```
choosetreemenu(4) :-
 write('identifier child node: '),
 read(Node),
 nl,
 down(Node).

choosetreemenu(5) :-
 up.

choosetreemenu(6) :-
 top.

choosetreemenu(7) :-
 solution.


% ============================================================================
/*
PREDICATE: subjectsmenu(+Menuwindow,?Subject)
ARGUMENTS: Menuwindow, id atom
           Subject,    subject representation
COMMENTS:  Succeeds after setting the actual Subjects Menu on Menuwindow
           window, waiting for a correct response and selecting Subject
           according to it
*/
% ============================================================================

subjectsmenu(Menuwindow,Subject) :-
 subjects(Subjects),
 dbmenu(subjects,Subjects,Subjectsmenu),
 showmenu(Menuwindow,Subjectsmenu,N),
 item(Subjects,N,Subject).


% ============================================================================
/*
PREDICATE: reordersubjectsmenu(+Menuwindow,+Subjects,?N)
ARGUMENTS: Menuwindow, id atom
           Subjects,   list of subjects
COMMENTS:  Succeeds after setting the Subjects Menu on Menuwindow
           window, waiting for a correct response and setting N to the
           corresponding number
*/
% ============================================================================

reordersubjectsmenu(Menuwindow,Subjects,N) :-
 dbmenu(subjects,Subjects,Subjectsmenu),
 showmenu(Menuwindow,Subjectsmenu,N).


% ============================================================================
/*
PREDICATE: dbmenu(+Title,+Opt,?Id)
ARGUMENTS: Title, atom
           Opt,   list of atoms
           Id,    id atom
COMMENTS:  Succeeds after setting Id to the corresponding menu identifier
           for [Title|Opt] menu, if one exists; a new created one, otherwise
*/
% ============================================================================

dbmenu(X,Y,Z) :-
 idmenu(X,Y,Z),
 !.

dbmenu(X,Y,Z) :-
 makemenu([X|Y],Z),
 asserta(idmenu(X,Y,Z)).


% ===========================================================================
/*
PREDICATE: lectroomsmenu(+Menuwindow,+Subject,?Room)
ARGUMENTS: Menuwindow, id atom
           Subject,    subject representation
           Room,       room representation
COMMENTS:  Succeeds after setting the Rooms Menu corresponding to Subject on
           Menuwindow window, waiting for a correct response and selecting Room
           according to it
*/
% ===========================================================================

lectroomsmenu(Menuwindow,Subject,Room) :-
 lectrooms(Subject,Rooms),
 dbmenu(rooms,Rooms,Lectroomsmenu),
 showmenu(Menuwindow,Lectroomsmenu,N),
 item(Rooms,N,Room).


% ===========================================================================
/*
PREDICATE: daysmenu(+Menuwindow,?Day)
ARGUMENTS: Menuwindow, id atom
           Day,        subject representation
COMMENTS:  Succeeds after setting the Days Menu on Menuwindow
           window, waiting for a correct response and selecting Day
           according to it
*/
% ===========================================================================

daysmenu(Menuwindow,Day) :-
 days(Days),
 idmenu(days,Daysmenu),
 showmenu(Menuwindow,Daysmenu,N),
 item(Days,N,Day).


% ===========================================================================
/*
PREDICATE: hoursmenu(+Menuwindow,?Hour)
ARGUMENTS: Menuwindow, id atom
           Hour,       subject representation
COMMENTS:  Succeeds after setting the Hours Menu on Menuwindow
           window, waiting for a correct response and selecting Hour
           according to it
*/
% ===========================================================================

hoursmenu(Menuwindow,Hour) :-
 hours(Hours),
 idmenu(hours,Hoursmenu),
 showmenu(Menuwindow,Hoursmenu,N),
 item(Hours,N,Hour).


% ===========================================================================
/*
```

```
PREDICATE: delete
ARGUMENTS: NONE
COMMENTS:  Succeeds after producing a template in order to delete constraints
           from the environment, and performing such deletions, if no
           serious inconsistency is found
*/
% ===========================================================================

delete :-
 ((subjects([]),
   !,
   write('no subjects'),
   nl,
   fail)
  ;
  (true)
 ),
 idwindow(menu,Menuwindow),
 idmenu(delete,Deletemenu),
 repeat,
 write('please, enter consistent data.'),
 nl,
 nl,
 showmenu(Menuwindow,Deletemenu,Num),
 ((Num = 1,
   Opt = subject,
   subjectsmenu(Menuwindow,Subject),
   Lastaction = [Subject])
  ;
  (Num = 2,
   Opt = bad,
   subjectsmenu(Menuwindow,Subject),
   daysmenu(Menuwindow,Day),
   hoursmenu(Menuwindow,Hour),
   bad(Subject,[Day,Hour]),
   Lastaction = [Subject,Day,Hour])
  ;
  (Num = 3,
   Opt = bad,
   subjectsmenu(Menuwindow,Subject),
   daysmenu(Menuwindow,Day),
   hoursmenu(Menuwindow,Hour),
   verybad(Subject,[Day,Hour]),
   Lastaction = [Subject,Day,Hour])
  ;
  (Num = 4,
   Opt = notpos,
   subjectsmenu(Menuwindow,Subject),
   daysmenu(Menuwindow,Day),
   hoursmenu(Menuwindow,Hour),
   notpos(Subject,[Day,Hour],As),
   Lastaction = [Subject,Day,Hour,As])
  ;
  (Num = 5,
   Opt = nonsimult,
   putwindow('First '),
   subjectsmenu(Menuwindow,Subject1),
   putwindow('Second '),
   subjectsmenu(Menuwindow,Subject2),
   erasew('Second '),
   ((nonsimult(Subject1,Subject2,As),
     !,
     Lastaction = [Subject1,Subject2,As])
    ;
    (nonsimult(Subject2,Subject1,As),
     !,
     Lastaction = [Subject2,Subject1,As])
   ;
    (fail)
   )
  )
 ;
  (Num = 6,
   Opt = nonfollow,
   putwindow('First '),
   subjectsmenu(Menuwindow,Subject1),
   putwindow('Second '),
   subjectsmenu(Menuwindow,Subject2),
   erasew('Second '),
   ((nonfollow(Subject1,Subject2,As),
     !,
     Lastaction = [Subject1,Subject2,As])
   ;
    (nonfollow(Subject2,Subject1,As),
     !,
     Lastaction = [Subject2,Subject1,As])
   ;
    (fail)
   )
  )
 ;
  (Num = 7,
   Opt = lectroom,
   subjectsmenu(Menuwindow,Subject),
   lectroomsmenu(Menuwindow,Subject,Room),
   lectroom(Subject,Room,As),
   Lastaction = [Subject,Room,As])
 ;
  (Num = 0)
 ),
 !,
 Num > 0,
 delete(Opt,Lastaction,Changefail),
 newsolnode([delete,Opt|Lastaction],Changefail).


testnew(N) :-
 constraintnumber(N),
 !,
 write(N),
 write(' is the number of an existing constraint'),
 nl,
 fail.

testnew(N) :-
 deleted(N),
 !,
 write(N),
 write(' is the number of an old constraint. You cannot use it'),
 nl,
 fail.

testnew(N) :-
 assumptnumber(N),
 !,
 write(N),
 write(' is the number of an existing assumption'),
 nl,
```

```
 fail.

testnew(N) :-
 nodenumber(N),
 !,
 write(N),
 write(' is the number of an existing node'),
 nl,
 fail.

testnew(N) :-
 treenodenumber(N),
 !,
 write(N),
 write(' is the number of an existing tree node'),
 nl,
 fail.

testnew(_).


% ============================================================================
/*
PREDICATE: add
ARGUMENTS: NONE
COMMENTS:  Succeeds after producing a template in order to add constraints
           to the environment, and performing such aditions, if no
           serious inconsistency is found
*/
% ============================================================================

add :-
 idwindow(menu,Menuwindow),
 idmenu(add,Addmenu),
 repeat,
 write('please, enter consistent data.'),
 nl,
 nl,
 showmenu(Menuwindow,Addmenu,Num),
 ((Num = 1,
   Opt = subject,
   write('subject: '),
   read(Subject),
   nl,
   subjects(Subjects),
   ((memberchk(Subject,Subjects),
     !,
     write('That subject was already in the database'),
     nl,
     fail)
   ;
    (true)
   ),
   Lastaction = [Subject])
 ;
  (Num = 2,
   Opt = bad,
   ((subjects([]),
     !,
     write('no subjects'),
     nl,
     fail)
   ;
    (true)
  ),
  subjectsmenu(Menuwindow,Subject),
  daysmenu(Menuwindow,Day),
  hoursmenu(Menuwindow,Hour),
  ((bad(Subject,[Day,Hour]),
    !,
    write('That fact was already in the database'),
    nl,
    fail)
  ;
   (true)
  ),
  Lastaction = [Subject,Day,Hour])
;
 (Num = 3,
  Opt = verybad,
  ((subjects([]),
    !,
    write('no subjects'),
    nl,
    fail)
  ;
   (true)
  ),
  subjectsmenu(Menuwindow,Subject),
  daysmenu(Menuwindow,Day),
  hoursmenu(Menuwindow,Hour),
  ((verybad(Subject,[Day,Hour]),
    !,
    write('That fact was already in the database'),
    nl,
    fail)
  ;
   (true)
  ),
  Lastaction = [Subject,Day,Hour])
;
 (Num = 4,
  Opt = notpos,
  ((subjects([]),
    !,
    write('no subjects'),
    nl,
    fail)
  ;
   (true)
  ),
  subjectsmenu(Menuwindow,Subject),
  daysmenu(Menuwindow,Day),
  hoursmenu(Menuwindow,Hour),
  newconsnumber(LastAs),
  !,
  As is LastAs + 1,
  ((notpos(Subject,[Day,Hour],As2),
    !,
    write('That constraint is already in the database, with number '),
    write(As2),
    nl,
    fail)
  ;
   (asserta(newconsnumber(As)))
  ),
  Lastaction = [Subject,Day,Hour,As])
;
 (Num = 5,
  Opt = nonsimult,
  ((subjects([]),
    !,
    write('no subjects'),
    nl,
    fail)
  ;
   (true)
  ),
  putwindow('First '),
  subjectsmenu(Menuwindow,Subject1),
  putwindow('Second '),
  subjectsmenu(Menuwindow,Subject2),
  erasew('Second '),
  newconsnumber(LastAs),
  !,
  As is LastAs + 1,
  ((nonsimultaneous(Subject1,Subject2,As2),
    !,
    write('That constraint is already in the database, with number '),
    write(As2),
    nl,
    fail)
  ;
   (asserta(newconsnumber(As)))
  ),
  Lastaction = [Subject1,Subject2,As])
;
 (Num = 6,
  Opt = nonfollow,
  ((subjects([]),
    !,
    write('no subjects'),
    nl,
    fail)
  ;
   (true)
  ),
  putwindow('First '),
  subjectsmenu(Menuwindow,Subject1),
  putwindow('Second '),
  subjectsmenu(Menuwindow,Subject2),
  erasew('Second '),
  newconsnumber(LastAs),
  !,
  As is LastAs + 1,
  ((nonfollowing(Subject1,Subject2,As2),
    !,
    write('That constraint is already in the database, with number '),
    write(As2),
    nl,
    fail)
  ;
   (asserta(newconsnumber(As)))
  ),
  Lastaction = [Subject1,Subject2,As])
;
 (Num = 7,
  Opt = lectroom,
  ((subjects([]),
    !,
    write('no subjects'),
    nl,
     fail)
   ;
    (true)
   ),
   subjectsmenu(Menuwindow,Subject),
   write('Room: '),
   read(Room),
   nl,
   newconsnumber(LastAs),
   !,
   As is LastAs + 1,
   ((testroom(Subject,Room),
     write('this room was already in the database'),
     nl,
     fail)
   ;
    (asserta(newconsnumber(As)))
   ),
   Lastaction = [Subject,Room,As])
 ;
  (Num = 8,
   Opt = fix,
   ((notallowed(solve),
     !)
   ;
    (write('option available only after "solve". Save the constraints'),
     nl,
     write('in a file, and modify it, if you need this option just now'),
     nl,
     fail)
   ),
   subjectsmenu(Menuwindow,Subject),
   Lastaction = [Subject])
 ;
  (Num = 9,
   Opt = subjlectures,
   ((notallowed(solve),
     !)
   ;
    (write('option available only after "solve". Save the constraints'),
     nl,
     write('in a file, and modify it, if you need this option just now'),
     nl,
     fail)
   ),
   subjectsmenu(Menuwindow,Subject),
   write('Number of lectures: '),
   read(N),
   nl,
   integer(N),
   N >= 0,
   ((subjlectures(Subject,N),
     !,
     write('That fact is already in the database'),
     nl,
     fail)
   ;
    (true)
   ),
   Lastaction = [Subject,N])
 ;
  (Num = 0)
),
!,
 Num > 0,
 add(Opt,Lastaction,Changefail),
 ((sol(_),
   retract(sol(_)))
 ;
  (true)
 ),
 newsolnode([add,Opt|Lastaction],Changefail).


% ============================================================================
/*
PREDICATE: move
ARGUMENTS: NONE
COMMENTS:  Succeeds after producing a template in order to move lectures
           in the timetable, and performing such changes, if no
           serious inconsistency is found
*/
% ============================================================================

move :-
 timetable([]),
 !,
 write('there is nothing to move'),
 nl.

move :-
 idwindow(menu,Menuwindow),
 repeat,
 write('please, enter consistent data.'),
 nl,
 nl,
 subjectsmenu(Menuwindow,Subject),
 putwindow('FROM '),
 daysmenu(Menuwindow,Day),
 hoursmenu(Menuwindow,Hour),
 lectroomsmenu(Menuwindow,Subject,Room),
 putwindow('TO '),
 daysmenu(Menuwindow,Day2),
 hoursmenu(Menuwindow,Hour2),
 lectroomsmenu(Menuwindow,Subject,Room2),
 erasew('TO '),
 Lastaction = [Subject,Day,Hour,Room,Day2,Hour2,Room2],
 timetable(Timetable),
 memberchk([_,Subject,Day,Hour,Room],Timetable),
 ((Day = Day2,
   Hour = Hour2,
   !,
   ((Room = Room2,
     write('THERE is NO CHANGE'),
     nl,
     !,
     fail)
   ;
    (true)   % change of room, same time, always allowed
   )
  )
 ;
  (fix(Subject,_,_),
   !,
   write('FIXED LECTURE'),
   nl,
   fail)
 ;
```

```
   (true)
  ),
  !,
  move(Lastaction,Changefail),
  newsolnode([move|Lastaction],Changefail).


% ============================================================================
/*
PREDICATE: newsolnode(+Lastaction)
ARGUMENTS: Lastaction, list of information
COMMENTS:  Succeeds after updating the environment, when a change (move, add
           or delete) has been introduced by the user:
           - Creating a node to hold the new timetable state
           - introducing the change in the 'history', to be seen later
           - Looking at the state of the problem (solved or not) according
             to 'tracks' and ATMS information, and asserting it.
*/
% ============================================================================

newsolnode(_,_) :-
 timetable([]),
 !.

newsolnode(Lastaction,ChangeFail) :-
 timetable(Timetable),
 dbtreenode(TreeNode),
 extract(Timetable,Nodelist),
 putjustification(TreeNode,Nodelist),
 retract(solnode(_)),
 asserta(solnode(TreeNode)),
 ((ChangeFail = yes,
   !,
   unsolved(BadAs))
 ;
  (Lastaction = [_,lectroom|_],
   !,
   unsolved(BadAs))
 ;
  (Lastaction = [delete,bad|_],
   !,
   unsolved(BadAs))
 ;
  (Lastaction = [delete,verybad|_],
   !,
   unsolved(BadAs))
 ;
  (unsolved([]),
   Lastaction = [delete|_],
   !,
   BadAs = [])
 ;
  (findinconsist(BadAs),
   retract(unsolved(_)),
   asserta(unsolved(BadAs)),
   retract(tracks(_)),
   setof2(X,findtracks(BadAs,X),Tracks),
   asserta(tracks(Tracks)))
 ),
 history(List),
 conc(List,[[change,TreeNode,Lastaction,ChangeFail,BadAs]],Newlist),
 retract(history(List)),
 asserta(history(Newlist)),
 asserta(historyTimetable(change,TreeNode,Timetable)),
```

```
 showtimetable,
 user_info.


% ============================================================================
/*
PREDICATE: readtimes(+N,+Subject,+Timetable,+Nodedblist,
                     ?DH,?NewTimetable,?NewNodedblist)
ARGUMENTS: N,              integer
           Subject,        subject representation
           Timetable,      list of nodereps
           Nodedblist,     list of nodereps
           DH,             list of [Day,Hour]
           NewTimetable,   list of nodereps
           NewNodedblist,  list of nodereps
COMMENTS:  Succeeds after getting N days, hours and rooms from
           the user and instantiating them to DH, and updating Newtimetable
           and NewNodedblist according to the new lectures generated.
           Eventual inconsistencies are sent to ATMS
*/
% ============================================================================

readtimes(0,_,Timetable,Nodedblist,[],Timetable,Nodedblist).

readtimes(N,Subject,Timetable,Nodedblist,
          [[Day,Hour]|DH],Newtimetable,Newnodedblist) :-
 idwindow(menu,Menuwindow),
 N > 0,
 N1 is N - 1,
 readtimes(N1,Subject,Timetable,Nodedblist,
           DH,Midtimetable,Midnodedblist),
 atomize([N],[M]),
 putwindow(M),
 repeat,
 daysmenu(Menuwindow,Day),
 hoursmenu(Menuwindow,Hour),
 Nodedb = [Node,Subject,Day,Hour],
 db(Nodedb,Node),
 dbass(Nodedb),
 ((notpos(Subject,[Day,Hour],As),
   putjustification(0,[Node,As]))
 ;
  (true)
 ),
 ((notpos(all,[Day,Hour],As2),
   putjustification(0,[Node,As2]))
 ;
  (true)
 ),
 Nodedb2 = [Node2,Subject,Day,Hour,Room],
 ((memberchk([Day,Hour],DH),
   write('same lecture, same time. Try again'),
   nl,
   fail)
 ;
  (memberchk(Nodedb2,Timetable),
   write('same lecture, same time. Try again'),
   nl,
   fail)
 ;
  (lectroomsmenu(Menuwindow,Subject,Room))
 ),
 lookfor(Midtimetable,Day,Hour,Before,Then,After,Sides),
 recsubtest(Nodedb,Then,Sides,_,_,_,_),
 db(Nodedb2,Node2),
 dbnode(Nodedb2),
 Newnodedblist = [Nodedb2|Midnodedblist],
 conc([Nodedb2|Then],After,After1),
 conc(Before,After1,Newtimetable),
 erasew(_).


% ========================================================================
/*
PREDICATE: readlist(+N,?DH,?Aslist)
ARGUMENTS: N,       integer
           DH,      list of [Day,Hour]
           Aslist,  list of As, Assumption numbers
COMMENTS:  Succeeds after getting N days, hours from
           the user and new Assumption numbers from the database system, and
           instantiating them to DH and Aslist, as required.
*/
% ========================================================================

readlist(0,[],[]).

readlist(N,[[Day,Hour]|DH],[As|Aslist]) :-
 idwindow(menu,Menuwindow),
 N > 0,
 N1 is N - 1,
 readlist(N1,DH,Aslist),
 atomize([N],[M]),
 putwindow(M),
 repeat,
 daysmenu(Menuwindow,Day),
 hoursmenu(Menuwindow,Hour),
 ((memberchk([Day,Hour],DH),
   write('same lecture, same time. Try again'),
   nl,
   fail)
 ;
  (true)
 ),
 newconsnumber(LastAs),
 !,
 As is LastAs + 1,
 asserta(newconsnumber(As)),
 altconstraint(As),
 erasew(_).


altconstraint(As) :-
 notallowed(solve),
 !,
 putconstraint(As).

altconstraint(_).


% ========================================================================
/*
PREDICATE: readrooms(+N,+Subject,?Rooms)
ARGUMENTS: N,       integer
           Subject, subject representation
           Rooms,   list of rooms
COMMENTS:  Succeeds after getting N 'Room's and 'As's, assumption numbers from
           the user and asserting lectroom(Subject,Room,As), according to
           the input data
*/
% ========================================================================

readrooms(0,_,[]).

readrooms(N,Subject,[Room|Rooms]) :-
 N > 0,
 N1 is N - 1,
 nl,
 write('Room: '),
 read(Room),
 nl,
 repeat,
 newconsnumber(LastAs),
 !,
 As is LastAs + 1,
 asserta(newconsnumber(As)),
 altconstraint(As),
 asserta(lectroom(Subject,Room,As)),
 readrooms(N1,Subject,Rooms).


% ========================================================================
/*
PREDICATE: reorder(+Subject,+Subjects,?OrdSubjects)
ARGUMENTS: Subject,     subject representation
           Subjects,    list of subjects representation
           OrdSubjects, list of subjects representation
COMMENTS:  Succeeds after instantiating OrdSubjects to the list with the
           same elements as Subjects, PLUS Subject, in the specified order
*/
% ========================================================================

reorder(Subject,Subjects,OrdSubjects) :-
 write('Remember that FIXED Subjects MUST BE FIRST'),
 nl,
 write('Which is the Subject that you want to precede '),
 write(Subject),
 write('?'),
 nl,
 write('If First, Press the mouse out of the Menu'),
 nl,
 putwindow('Previous'),
 idwindow(menu,Menuwindow),
 reordersubjectsmenu(Menuwindow,Subjects,N),
 erasew('Previous'),
 reorderaux(Subject,Subjects,N,OrdSubjects).


% ========================================================================
/*
PREDICATE: reorderaux(+Subject,+Subjects,+N,?OrdSubjects)
ARGUMENTS: Subject,     subject representation
           Subjects,    list of subjects representation
           N,           integer
           OrdSubjects, list of subjects representation
COMMENTS:  Succeeds after instantiating OrdSubjects to the list with the
           same elements as Subjects, PLUS Subject after the Nth position
*/
% ========================================================================

reorderaux(Subject,Subjects,0,[Subject|Subjects]) :-
 !.
```

```
reorderaux(Subject,[H|T],N,[H|OT]) :-
 N1 is N - 1,
 reorderaux(Subject,T,N1,OT).


% ============================================================================
/*
PREDICATE: testday(+Day)
ARGUMENTS: Day, days representation
COMMENTS:  Succeeds if input data Day is correct
*/
% ============================================================================

testday(Day) :-
 days(Days),
 memberchk(Day,Days).


% ============================================================================
/*
PREDICATE: testhour(+Hour)
ARGUMENTS: Hour, hour representation
COMMENTS:  Succeeds if input data Hour is correct
*/
% ============================================================================

testhour(Hour) :-
 hours(Hours),
 memberchk(Hour,Hours).


% ============================================================================
/*
PREDICATE: testsubject(+Subject)
ARGUMENTS: Subject, subject representation
COMMENTS:  Succeeds if input data Subject is correct
*/
% ============================================================================

testsubject(Subject) :-
 subjects(Subjects),
 memberchk(Subject,Subjects).


% ============================================================================
/*
PREDICATE: testroom(+Subject,+Room)
ARGUMENTS: Subject, subject representation
           Room,    room representation
COMMENTS:  Succeeds if any 'lectroom(Subject,Room,_)' exists
*/
% ============================================================================

testroom(Subject,Room) :-
 lectroom(Subject,Room,_).
```

Appendix E

Constraints Files: realconstraints.pl

This appendix contains the file where the real constraints for the timetable, according to the "MSc in Information Technology" course information. This serves as a sample of a "constraints file" and as a layout of how the system saves the "actual constraints" in a file (except for comments with '/*', '*/' and '%', of course). The same layout, but including more predicates would be the one for a "whole environment" file.

The listing starts in next page, in a format of two pages in one, with page numbers starting with 1.

```
/*
File:    defaults.pl
Author:  Luis Montero, MSc Student (lmg@forth, lmg@aipna)
Purpose: defaults for the Timetable Design Support System using ATMS.
*/

days([mon,tue,wed,thu,fri]).


hours([9,10,11,12,13,14,15,16]).


optdifdays(3).


maxassumptnumber(4096).


firsttreenode(10000).


firstnode(20000).


firstas(30000).


newconsnumber(40000).


nonsharedrooms(40000).


nobacktrack.
```

```
/*
File:    realconstraints.pl
Author:  Luis Montero, MSc Student (lmg@forth, lmg@aipna)
Purpose: Constraints for the Timetable Design Support System using ATMS.
*/

subjects([databas,cadvlsi,remsens,fsem2,robsens,imdata,spc,behav,cling2,
          prolog,kri2,speech2,nlg,matreas,assemb,compcom]).


subjlectures(prolog,2).
subjlectures(kri2,3).
subjlectures(speech2,2).
subjlectures(nlg,3).
subjlectures(matreas,3).
subjlectures(assemb,3).
subjlectures(compcom,2).

nonsimult(prolog,kri2,900).
nonsimult(prolog,nlg,901).
nonsimult(kri2,nlg,902).
nonsimult(prolog,matreas,903).
nonsimult(kri2,matreas,904).
nonsimult(nlg,matreas,905).
nonsimult(prolog,fsem2,906).
nonsimult(kri2,fsem2,907).
nonsimult(nlg,fsem2,908).
nonsimult(kri2,cling2,909).
nonsimult(fsem2,cling2,910).
nonsimult(prolog,spc,911).
nonsimult(kri2,spc,912).
nonsimult(nlg,spc,913).
nonsimult(prolog,assemb,914).
nonsimult(prolog,robsens,915).
nonsimult(assemb,robsens,916).
nonsimult(prolog,databas,917).
nonsimult(assemb,databas,918).
nonsimult(robsens,databas,919).
nonsimult(kri2,databas,920).
nonsimult(nlg,databas,921).
nonsimult(matreas,spc,922).
nonsimult(matreas,cling2,923).
nonsimult(kri2,assemb,924).
nonsimult(assemb,spc,925).
nonsimult(kri2,robsens,926).
nonsimult(prolog,speech2,927).
nonsimult(kri2,speech2,928).
nonsimult(fsem2,speech2,929).
nonsimult(kri2,compcom,930).
nonsimult(cling2,compcom,931).
nonsimult(cling2,databas,932).
nonsimult(databas,compcom,933).
nonsimult(compcom,cadvlsi,934).
nonsimult(compcom,remsens,935).
nonsimult(compcom,robsens,936).
nonsimult(compcom,imdata,937).
nonsimult(compcom,spc,938).
nonsimult(compcom,behav,939).


nonfollow(prolog,spc,811).
nonfollow(kri2,spc,812).
nonfollow(nlg,spc,813).
nonfollow(prolog,robsens,815).
nonfollow(assemb,robsens,816).
nonfollow(prolog,databas,817).
nonfollow(assemb,databas,818).
nonfollow(kri2,databas,820).
nonfollow(nlg,databas,821).
nonfollow(matreas,spc,822).
nonfollow(assemb,spc,825).

nonfollow(kri2,robsens,826).

nonfollow(kri2,compcom,830).

nonfollow(cling2,compcom,831).

nonfollow(cling2,databas,832).

nonfollow(cling2,compcom,833).


lectroom(prolog,atlt2,201).

lectroom(kri2,atlt2,211).

lectroom(speech2,afb8,261).

lectroom(nlg,sbf10,221).

lectroom(matreas,sbf10,231).

lectroom(matreas,sbf12,232).

lectroom(assemb,sbf10,241).

lectroom(compcom,kbltb,251).

lectroom(compcom,kb3212,252).

lectroom(databas,kb3218,101).

lectroom(cadvlsi,kb3214,111).

lectroom(spc,kbltb,121).

lectroom(behav,kb3315,131).

lectroom(robsens,todd,141).

lectroom(imdata,royobs,151).

lectroom(fsem2,cognsci,161).

lectroom(cling2,cognsci,171).

lectroom(remsens,meteor,181).


fix(databas,[mon,9],301).

fix(databas,[tue,12],302).

fix(databas,[thu,10],303).

fix(cadvlsi,[mon,10],311).

fix(cadvlsi,[mon,11],312).

fix(remsens,[mon,12],381).

fix(imdata,[mon,14],351).

fix(imdata,[tue,11],352).

fix(imdata,[thu,11],353).

fix(imdata,[fri,14],354).

fix(fsem2,[mon,14],361).

fix(fsem2,[mon,15],362).

fix(robsens,[tue,9],341).

fix(robsens,[tue,10],342).

fix(spc,[tue,14],321).

fix(spc,[thu,14],322).

fix(behav,[wed,10],331).

fix(cling2,[fri,11],371).

fix(cling2,[fri,12],372).


bad(all,[mon,9]).

bad(all,[tue,9]).

bad(all,[wed,9]).

bad(all,[thu,9]).

bad(all,[fri,9]).

bad(all,[wed,14]).

bad(all,[wed,15]).

bad(all,[wed,16]).

bad(all,[fri,14]).

bad(all,[fri,15]).

bad(all,[fri,16]).


notpos(all,[mon,13],1001).

notpos(all,[tue,13],1002).

notpos(all,[wed,13],1003).

notpos(all,[thu,13],1004).

notpos(all,[fri,13],1005).


% DAVE REASONS :

notpos(prolog,[mon,14],1101).

notpos(prolog,[mon,15],1102).
notpos(prolog,[tue,9],1103).

notpos(prolog,[thu,9],1104).

notpos(prolog,[thu,15],1105).

notpos(prolog,[fri,9],1106).

notpos(prolog,[fri,14],1107).

% ATLT2 REASONS:

notpos(prolog,[mon,9],1108).

notpos(prolog,[mon,10],1109).

notpos(prolog,[mon,11],1110).

notpos(prolog,[mon,12],1111).

notpos(prolog,[tue,10],1112).

notpos(prolog,[tue,11],1113).

notpos(prolog,[tue,12],1114).

bad(prolog,[tue,14]).

notpos(prolog,[wed,9],1115).

notpos(prolog,[wed,10],1116).

notpos(prolog,[wed,11],1117).

bad(prolog,[wed,14]).

bad(prolog,[wed,15]).

notpos(prolog,[thu,10],1118).

notpos(prolog,[thu,11],1119).

bad(prolog,[thu,12]).

bad(prolog,[thu,14]).

bad(prolog,[fri,10]).

notpos(prolog,[fri,11],1120).

notpos(prolog,[fri,12],1121).

notpos(prolog,[fri,14],1122).

notpos(prolog,[fri,15],1123).

% TIM REASONS:

notpos(kri2,[mon,14],1201).

notpos(kri2,[mon,15],1202).

notpos(kri2,[mon,16],1203).

notpos(kri2,[tue,9],1204).

notpos(kri2,[tue,10],1205).

notpos(kri2,[tue,11],1206).

notpos(kri2,[tue,12],1207).

notpos(kri2,[wed,9],1208).

notpos(kri2,[wed,10],1209).

notpos(kri2,[wed,11],1210).

notpos(kri2,[wed,12],1211).

notpos(kri2,[wed,14],1212).

notpos(kri2,[wed,15],1213).

notpos(kri2,[wed,16],1214).

notpos(kri2,[thu,9],1215).

notpos(kri2,[thu,10],1216).

notpos(kri2,[thu,11],1217).

notpos(kri2,[thu,12],1218).

notpos(kri2,[fri,9],1219).

notpos(kri2,[fri,10],1220).

notpos(kri2,[fri,11],1221).

notpos(kri2,[fri,12],1222).

% ATLT2 REASONS:

bad(kri2,[tue,14]).

bad(kri2,[thu,14]).

bad(kri2,[thu,15]).

notpos(kri2,[mon,9],1223).

notpos(kri2,[mon,10],1224).

notpos(kri2,[mon,11],1225).

notpos(kri2,[mon,12],1226).

notpos(kri2,[fri,14],1227).

notpos(kri2,[fri,15],1228).
```

```
% SPEECH

notpos(speech2,[mon,9],1301).
notpos(speech2,[mon,10],1302).
notpos(speech2,[mon,11],1303).
notpos(speech2,[mon,12],1304).
notpos(speech2,[mon,14],1305).
notpos(speech2,[mon,15],1306).
notpos(speech2,[mon,16],1307).
notpos(speech2,[tue,9],1308).
notpos(speech2,[tue,10],1309).
notpos(speech2,[tue,11],1310).
notpos(speech2,[tue,12],1311).
notpos(speech2,[tue,14],1312).
bad(speech2,[wed,9]).
bad(speech2,[wed,10]).
notpos(speech2,[wed,11],1313).
notpos(speech2,[wed,12],1314).
notpos(speech2,[wed,16],1315).
notpos(speech2,[thu,9],1316).
notpos(speech2,[thu,10],1317).
notpos(speech2,[thu,11],1318).
notpos(speech2,[thu,12],1319).
notpos(speech2,[thu,14],1320).
notpos(speech2,[thu,15],1321).
notpos(speech2,[thu,16],1322).
notpos(speech2,[fri,9],1323).
notpos(speech2,[fri,10],1324).
notpos(speech2,[fri,11],1325).
notpos(speech2,[fri,12],1326).
notpos(speech2,[fri,14],1327).
notpos(speech2,[fri,15],1328).
notpos(speech2,[fri,16],1329).
```

```
% NLG

notpos(nlg,[tue,11],1401).
notpos(nlg,[tue,12],1402).
notpos(nlg,[fri,14],1403).
notpos(nlg,[fri,15],1404).
notpos(nlg,[fri,16],1405).


% MATREAS

bad(matreas,[tue,14]).
bad(matreas,[tue,15]).
bad(matreas,[wed,14]).
bad(matreas,[wed,15]).
bad(matreas,[fri,14]).
bad(matreas,[fri,15]).


% ASSEMB

notpos(assemb,[tue,11],1601).
notpos(assemb,[tue,12],1602).
notpos(assemb,[tue,14],1603).
```

Appendix F

Suntool File: .suntools

This appendix shows the content of the file '.suntools' neccessary to obtain the layout seen in the "screendump"s, assuming the use of "small bold" characters. The third line has been "cut" into two, as it is longer than 80 characters.

```
shelltool  -Wp    0  72 -Ws 730 828 -WP  128   0
cmdtool    -Wp    0  72 -Ws 750 828 -WP   64   0
cmdtool    -Wp    0   0 -Ws 670  71 -WP    0   0 -Wl "<< CONSOLE >>"
 -WL console -C
clock      -Wp  497  32 -Ws 218  39 -WP  680   0 -Wi
```