'PLATE' : a decision support system for resource-constrained project scheduling problems

Editors: prof.dr. K.M. van Hee
prof.dr. H.G. Sol

'PLATE'; A DECISION SUPPORT SYSTEM FOR RESOURCE-CONSTRAINED PROJECT SCHEDULING PROBLEMS

by

A. Jansen
L. Klieb
C. Noorlander
G. Wolf

NFI 11.90/03

EINDHOVEN UNIVERSITY OF TECHNOLOGY
F. du Buisson
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN

Augustus 1990

A. Jansen
Eindhoven University of Technology
Faculty of Mathematics and Computing Science
P.O. Box 513, 5600 MB Eindhoven, the Netherlands

L. Klieb
Eindhoven University of Technology
Faculty of Mathematics and Computing Science
P.O. Box 513, 5600 MB Eindhoven, the Netherlands

C. Noorlander
Eindhoven University of Technology
Faculty of Mathematics and Computing Science
P.O. Box 513, 5600 MB Eindhoven, the Netherlands

G. Wolf
Eindhoven University of Technology
Faculty of Mathematics and Computing Science
P.O. Box 513, 5600 MB Eindhoven, the Netherlands

"PLATE": A Decision Support System for Resource-Constrained Project Scheduling Problems

A. Jansen, L. Klieb, C. Noorlander, G. Wolf

Department of Mathematics and Computing Science,
Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

(date: august 17, 1990, preliminary version)

Abstract: PLATE is an interactive decision-support system for resource-constrained scheduling problems, developed at the Eindhoven University of Technology in the Netherlands in the scope of an international exercise, coordinated by IIASA, the International Institute for Applied Systems Analysis. Attention is not only paid to the (mathematical) problem of finding algorithms for generating good schedules in reasonable time, but also to the functionality and architecture of the whole system, especially the user interface.

1. Introduction

In this article we describe a decision support system, developed in the context of an international exercise, coordinated by IIASA, the International Institute of Applied Systems Analysis [1]. The participants, research groups from several countries, each developed a DSS for a representative class of scheduling problems, according to several conditions specified by IIASA. The purpose of the exercise is to acquire knowledge and experience in methods for designing decision support systems by means of the independent development of these systems for one and the same planning situation and their final evaluation. This paper reports the experiences of the computer science team of the Eindhoven University of Technology.

The planning situation, resource constrained project scheduling, can be characterized by a set of tasks, each to be processed by a set of resources in a certain time interval. No resource can be allocated to two tasks at the same time and a task has to be performed without interruption. For each task there is a release time, a deadline and a due date. Resources are available only at certain time intervals. Waiting times between two tasks can be specified by means of generalized precedence constraints. A task can be performed only by specific resource sets, each with a specific processing time.

A schedule consists of a set of allocations, triples (T,Rs,I), with T a task, Rs a resource set and I a time-interval. The constraints mentioned above define the feasibility of schedules. We call a schedule complete if every task is allocated, otherwise the schedule is partial.

To measure the quality of a schedule several criteria are possible. Most criteria are related to earliness or tardiness of tasks with respect to their due dates. The overall criterion can be specified by the user by parameterisation of several weighting factors [1].

In this article we pay attention not only to the (mathematical) problem of finding algorithms for generating good schedules in reasonable time, but also to the functionality and architecture of the whole system, especially the user interface.

Starting point is the idea that the planning process is made up of two essential components: automatic planning and manual planning.

Within the automatic planning process a schedule is generated by the system, either from scratch, or from a partial schedule. Because of the fact that there is no efficient algorithm for solving the general RCPS-problem, which is NP-hard, we use an approximation method, based on heuristics, to generate a schedule, satisfying as many constraints as possible and having a rather good (although not optimal) measure of quality. The user can tune the planning algorithm by means of several parameters.

Within the manual planning process the user is more involved. The user interface, the software component responsible for interaction between user and system, is comparable with an electronic planning board. A graphical representation of the schedule as Gantt diagram is displayed to show the allocation of tasks to resources (y-axis) over time (x-axis), including all characteristic values of the schedule. Infeasible schedules are not forbidden, but violations of constraints are visible to the user by means of special colouring. The system supports the management of primitive scheduling-operations, i.e. the user can insert, delete and shift task allocations in time by means of function-keys. It is very important for the user to see immediately the consequences of decisions. In this respect the system acts like a planning editor.

Within the whole planning process both automatic and manual planning may be used, in arbitrary order and frequency. In realistic situations we start with an empty or partial schedule. By automatic planning we can generate a basic schedule, that should be completed/corrected by manual planning.

Section 2 presents a formal description of the problem. In section 3 we give an function description of the system by means of dataflow diagrams and functional decomposition. Also some implementational aspects are discussed. In section 4 we explain algorithms and heuristics for the automatic planner. Also some test results are given. In section 5 we describe the manual planning component and the user interface. Finally in section 6 we discuss some problems and conclusions.

2. Summary of the scheduling problem

In this section we give a short, formal summary of the decision situation considered. For the original problem definition we refer to [1].

2.1. The kernel data

The **kernel** data of the decision situation are given by a 9-tuple $(R,E,T,a,b,F,P,\alpha,\beta)$, where:

- R is a non-empty, finite, set of **resources**.
- $E \in R \rightarrow \wp(\mathbb{R}x\mathbb{R})$ is a function defining **availabilities**: $(a,b) \in E(r) \Leftrightarrow$ "Resource $r \in R$ is available during time interval [a,b]". For arbitrary $(a,b) \in E(r)$ and $(c,d) \in E(r)$ with $(a,b) \neq (c,d)$ we require that $(a,b) \cap (c,d) = \emptyset$.
- T is a non-empty, finite, set of **tasks**.
- $a \in T \rightarrow \mathbb{R}$ defines **release times**: a(t) is the time at which task t becomes available for processing.
- $b \in T \rightarrow \mathbb{R}$ defines **deadlines**: b(t) is the time at which task t must have been processed (completely).
- $F \in T \rightarrow \wp(\wp(R))$ defines **feasible resourcesets**: processing of task $t \in T$ is only possible with resourcesets in F(t).
- $P \in \{(t,f) \mid t \in T \wedge f \in F(t)\} \rightarrow \mathbb{R}^+$ defines **processing times**: P(t,f) is the time needed to process task $t \in T$ with (feasible) resourceset $f \in F(t)$.
- $\alpha \in TxT \xrightarrow{p} \mathbb{R}$ is a partial function defining **minimal required waiting times**: $\alpha(t,u)$ is the minimal required time between the completion time of task t and the starting time of task u for $(t,u) \in dom(\alpha)$.
- $\beta \in TxT \xrightarrow{p} \mathbb{R}$ defines **maximal allowed waiting times**: $\beta(t,u)$ is the maximal allowed time between the completion time of task t and the starting time of task u for $(t,u) \in dom(\beta)$.

Our objective is the construction of a **schedule**, which is a 3-tuple (G,S,C), where:
- $G \in T \xrightarrow{}_{p} \wp(R)$ is a **resource schedule**. We require $G(t) \in F(t)$ for $t \in dom(G)$. G(t) gives the resourceset with which task $t \in dom(G)$ will be processed.
- $S,C \in T \xrightarrow{}_{p} \mathbb{R}$ are **time schedules**. S(t) gives the starting time of (the processing of) task $t \in dom(S)$, and C(t) gives the completion time of task $t \in dom(C)$.

A resource- or time schedule is **complete** iff its domain is equal to T. A schedule (G,S,C) is **complete** iff G,S and C are complete. A schedule (G_1,S_1,C_1) is an **extension** of schedule (G_2,S_2,C_2) iff $G_2 \subseteq G_1 \wedge S_2 \subseteq S_1 \wedge C_2 \subseteq C_1$.

2.2. The kernel constraints

We now define the **kernel** constraints which a schedule should satisfy. They are formulated as predicates which return true for an arbitrary complete schedule $\mathcal{F}=(G,S,C)$ iff the corresponding constraint is satisfied.

- BorderOk($\mathcal{F}$) := "Every task is processed between its release time and deadline"
  := $(\forall t: t \in T: a(t) \leq S(t) \wedge C(t) \leq b(t))$.
- MinWaitOk($\mathcal{F}$) := "The schedule satisfies the minimal required waiting times"
  := $(\forall t,u:(t,u) \in dom(\alpha): \alpha(t,u) \leq S(u)-C(t))$.
- MaxWaitOk($\mathcal{F}$) := "The schedule satisfies the maximal allowed waiting times"
  := $(\forall t,u:(t,u) \in dom(\beta): S(u)-C(t) \leq \beta(t,u))$.
- TimesOk($\mathcal{F}$) := "The completion time of each task is equal to its starting time plus its processing time"
  := $(\forall t: t \in T: C(t)=S(t)+P(t,G(t)))$.
- ResOk($\mathcal{F}$) := "No resource can be allocated to 2 tasks at the same time"
  := $(\forall t,u:t,u \in T \wedge G(t) \cap G(u) \neq \emptyset: C(t) \leq S(u) \vee C(u) \leq S(t))$.
- AvailOk($\mathcal{F}$) := "Every resource is available when it is used"
  := $(\forall t: t \in T: \forall r: r \in G(t): \exists a,b:(a,b) \in E(r): a \leq S(t) \wedge C(t) \leq b)$.

A complete schedule $\mathcal{F}$ is **feasible** iff it satisfies all 6 kernel constraints.

2.3. The optimality function

Finally, we define an optimality value for each feasible schedule.
The **optimality** data of the decision situation are given by a 5-tuple (Q,d,u,v,w), where:
- $Q \in \prod(T)$ defines a set of projects. Observe that a project is simply a set of tasks, and that every task belongs to exactly one project (Q is a **partition** of the taskset[1]).
- $d \in T \cup Q \rightarrow \mathbb{R}$ defines **duedates** for the projects and tasks. We say that a task or project $t \in T \cup Q$ has been processed **optimally** iff its completion time[2] is equal to its duedate.
- $u \in (\mathbb{R}_0^+)^8$ is a row of 8 weights.
- $v \in T \cup Q \rightarrow \mathbb{R}_0^+$ defines **earliness weights** for the tasks and projects.
- $w \in T \cup Q \rightarrow \mathbb{R}_0^+$ defines **tardiness weights** for the tasks and projects.

The optimality value of a schedule is equal to the weighed (via u) summation of the maximal and total weighed earlinesses and tardinesses of the projects and tasks (8 components in total). We define help values U(i:0≤i<8) and $CP \in Q \rightarrow \mathbb{R}$ for feasible schedule $\mathcal{F}=(G,S,C)$ as follows:
- $CP \in Q \rightarrow \mathbb{R}$ where $(q \in Q)$ $CP(q):=(\mathbf{MAX}t: t \in q: C(t))$
  CP(q) is the completion time of project q.

---

[1] Q is a partition of set T, notation $Q \in \prod(T)$, iff Q is a set of sets with:
- $T=(\cup E: E \in Q: E)$.
- $(\forall E,F: E,F \in Q \wedge E \neq F: E \cap F=\emptyset)$.

[2] The completion time of a project is equal to the maximum of the completion times of all tasks in that project.

- U(0) := "the maximal weighed earliness of the tasks"
  := (**MAX**t:t∈T:v(t)·max(d(t)-C(t),0))
- U(1) := "the maximal weighed earliness of the projects"
  := (**MAX**q:q∈Q:v(q)·max(d(q)-C(q),0))
- U(2) := "the maximal weighed tardiness of the tasks"
  := (**MAX**t:t∈T:w(t)·max(C(t)-d(t),0))
- U(3) := "the maximal weighed tardiness of the projects"
  := (**MAX**q:q∈Q:w(q)·max(C(q)-d(q),0))
- U(4) := "the total weighed earliness of the tasks"
  := (**SUM**t:t∈T:v(t)·max(d(t)-C(t),0))
- U(5) := "the total weighed earliness of the projects"
  := (**SUM**q:q∈Q:v(q)·max(d(q)-C(q),0))
- U(6) := "the total weighed tardiness of the tasks"
  := (**SUM**t:t∈T:w(t)·max(C(t)-d(t),0))
- U(7) := "the total weighed tardiness of the projects"
  := (**SUM**q:q∈Q:w(q)·max(C(q)-d(q),0))

Now we are able to give the optimality value $Z(\mathcal{F})$ of $\mathcal{F}$:

- $Z(\mathcal{F})$:= (**SUM**i:0≤i<8:u(i)·U(i))

A feasible schedule $\mathcal{F}$ is **optimal** iff it has a minimal optimality value (the minimum of the optimality values of all feasible schedules).

Now, our **scheduling problem** looks as follows. Given (constant) kernel data (R,E,T,a,b,F,P,α,β), optimality data (Q,d,u,v,w) and a starting schedule $PS=(G_p,S_p,C_p)$: Construct an optimal schedule $\mathcal{F}=(G,S,C)$ which is an extension of PS.

Remark: Mathematically it is not necessary to introduce the availabilities of resources in our problem explicitly, as they can be modelled with dummy tasks. Nevertheless we have chosen to keep them in our definition because it is an important concept which is needed later on.

3. Functional design and implementation

To describe the boundaries between the system and its external environment we use a context diagram (see fig. 1), which is in fact a dataflow diagram of level zero.

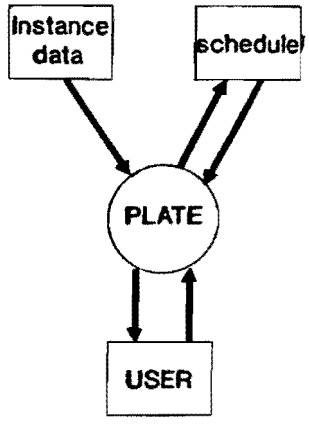


Fig.1 Context diagram

We have in fact two permanent data stores used by the system: the problem-instance data and the schedules. The problem instance data, containing the specifications of tasks and resources, including all constraints [1], are used only for input. The schedules, containing the effective allocations of resource sets and time intervals to tasks, are used for both input and output (existing schedules may be updated). As usual for an interactive system, the user is also involved to influence the planning process.

Within PLATE several processes can be distinguished, which are specified in the dataflow diagram of level one (see fig. 2). There is are a compiler and a decompiler, to convert ascii files (both instance data and schedules) to binary files and vice versa. There are two planning processes, the automatic and the manual planner.

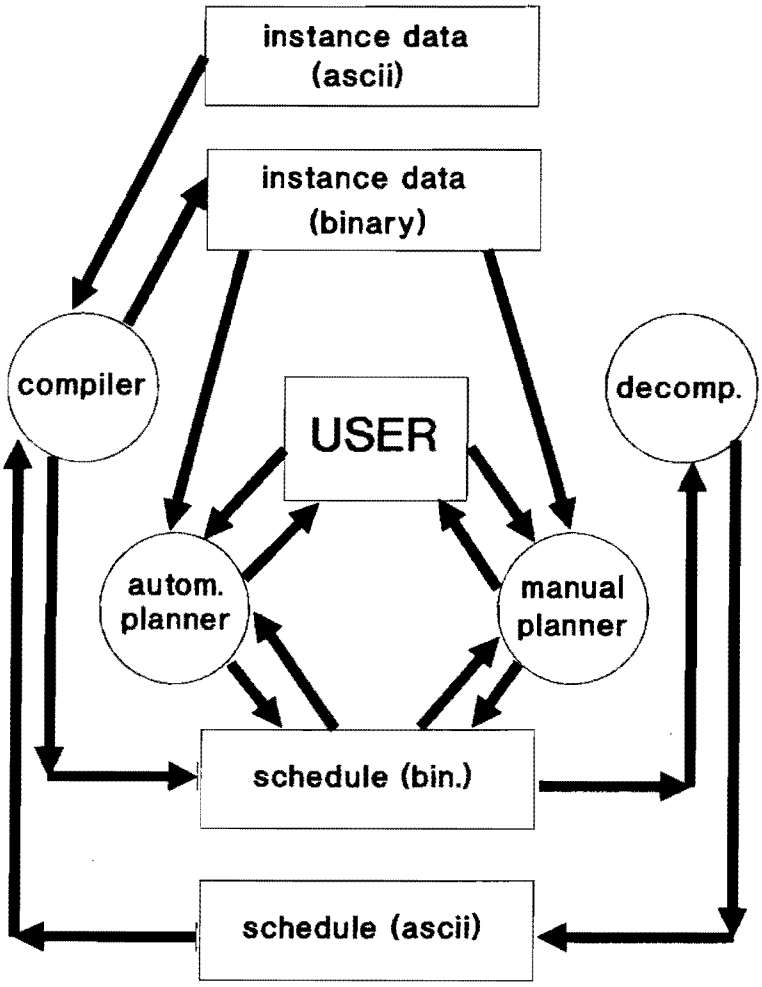


Fig. 2 Dataflow diagram

The user input is driven by menus and function keys, preventing the user from giving infeasible commands. The hierarchical menustructure corresponds pretty well with the functional decomposition of PLATE (see fig. 3).

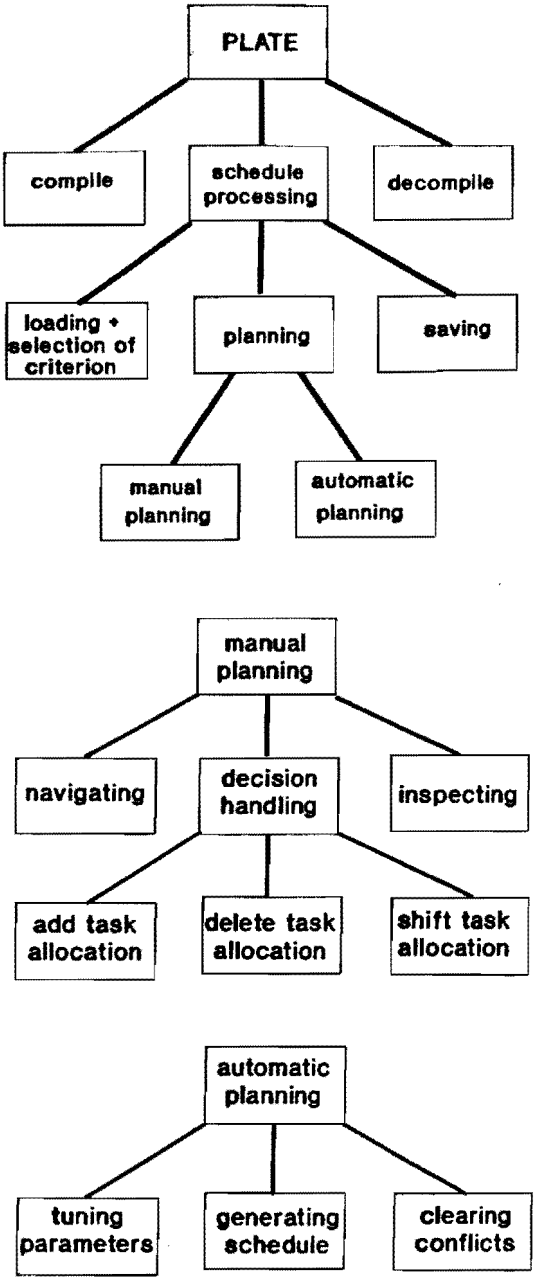


Fig.3 Functional decomposition

Before the effective scheduling process can be started, the problem instance data and the schedules should be converted to binary files by a simple compiler. Schedules created or updated (in binary form) can be decompiled afterwards back to ascii files.

Before starting the planning process, the user should load the (compiled) instance data and a schedule, which may exist or may be empty. Moreover the user should select a criterion, to measure the quality of the schedule during the planning process. Afterwards the user may decide to save the schedule or not.

The planning process does consist of both automatic and manual planning. Both processes may be performed in arbitrary order and frequency.

During automatic planning a complete schedule will be generated by the system, although some constraints may be violated. The user has the possibility to clear the schedule from faulty planned tasks and iterate the generation to get better results. Moreover he can tune the planning algorithm by means of several parameters (see section 4).

During manual planning two groups of functions are available to the user. First there are functions for navigation and inspection. With these we can scroll and zoom the planning-board, and inspect all kinds of relevant information. Secondly there is a group of decision handling functions. With these we can add task allocations to the schedule, remove them from the schedule or shift task-allocations in time (see section 5).

PLATE is written in Turbo Pascal (Borland), version 5.0, using the database toolbox and a software package for linear programming. It is made up of approximately 20.000 lines of source code, for a larger part written by students. The system runs on an IBM-PC with a CGA colour monitor under MS-DOS version 3.3.

4. Automatic planning

In this section, we globally describe the approximation method used in the automatic planning component of our DSS. Most parts are taken from [2].

4.1. Some definitions

First, we simplify constraint AvailOk by introducing a function **ResSetAvail**$\in \wp(R)\rightarrow\wp(\mathbb{R}x\mathbb{R})$ which gives the time segments during which an arbitrary resourceset is available, so for $W\in\wp(R)$ holds:

$(\cup a,b:(a,b)\in ResSetAvail(W):[a,b])=(\cap r:r\in W:(\cup a,b:(a,b)\in E(r):[a,b]))$

The algorithm used for calculating ResSetAvail(W) given E is very easy and will not be further elaborated here. Using this definition of ResSetAvail, we may rewrite constraint AvailOk as:

$AvailOk(\mathcal{F}) := (\forall t:t\in T:\exists a,b:(a,b)\in ResSetAvail(G(t)):a\leq S(t) \wedge C(t)\leq b)$

In the future we will use this alternative definition.

When we now observe constraints AvailOk and ResOk which a feasible schedule $\mathcal{F}=(G,S,C)$ must satisfy, we see that:

(1) Every task t will be processed in exactly one segment of ResSetAvail(G(t)). This means that two functions $X,Y\in T\rightarrow\mathbb{R}$ can be found for $\mathcal{F}$, where [X(t),Y(t)] represents the time segment in which task t will be processed: $(\forall t:t\in T:(X(t),Y(t))\in ResSetAvail(G(t)) \wedge X(t)\leq S(t) \wedge C(t)\leq Y(t))$.
Formalisation gives:
- We define a **segment schedule** for resource schedule G as a pair (X,Y), where:
  - $X,Y\in T\rightarrow_p\mathbb{R} \wedge dom(X)=dom(Y) \wedge dom(X)\subseteq dom(G)$
  - $(\forall t:t\in dom(X):(X(t),Y(t))\in ResSetAvail(G(t)))$
- A segment schedule (X,Y) is **complete**, notation FullSeg(G,X,Y), iff dom(X)=T.
- A complete schedule $\mathcal{F}=(G,S,C)$ **satisfies** segment schedule (X,Y) for G, notation $SegsOk(\mathcal{F},X,Y)$ iff $(\forall t:t\in dom(X):X(t)\leq S(t) \wedge C(t)\leq Y(t))$, which means that all tasks in $\mathcal{F}$ must be planned in the time segments given by (X,Y).

An important property of segment schedules is the following. For an arbitrary complete schedule $\mathcal{F}=(G,S,C)$ and a segment schedule (X,Y) for G holds:

$FullSeg(G,X,Y) \wedge SegsOk(\mathcal{F},X,Y) \Rightarrow AvailOk(\mathcal{F})$.

(2) For every pair (t,u) of tasks where $G(t)\cap G(u)\neq\emptyset$ we see that $S(t)\geq C(u)$ or $C(t)\leq S(u)$, so t and u must be processed in some strict order. Therefore, we can define a derivate set $O\subseteq TxT$ for $\mathcal{F}$, which represents the imposed orders for such conflicting taskpairs:

$(t,u) \in O \Leftrightarrow (G(t) \cap G(u) \neq \emptyset \wedge C(t) \leq S(u))$

Formalisation gives:

- We define a **sequence schedule** for resource schedule G as a set $O \subseteq TxT$, where: $(\forall t,u:(t,u) \in O: t,u \in dom(G) \wedge (u,t) \notin O \wedge G(t) \cap G(u) \neq \emptyset)$
- A sequence schedule is **complete**, notation FullOrder(G,O), iff: $(\forall t,u:t,u \in T \wedge G(t) \cap G(u) \neq \emptyset: (t,u) \in O \vee (u,t) \in O)$, which means that O contains an order for *precisely* all conflicting taskpairs.
- A complete schedule $\mathcal{F}=(G,S,C)$ **satisfies** sequence schedule O for G, notation OrderOk($\mathcal{F}$,O), iff $(\forall t,u:(t,u) \in O: C(t) \leq S(u))$, which means that for every order $(t,u) \in O$ holds that t will be completely processed before u in $\mathcal{F}$.

An important property of sequence schedules is the following. For an arbitrary complete schedule $\mathcal{F}=(G,S,C)$ and a sequence schedule O for G holds:

FullOrder(G,O) $\wedge$ OrderOk($\mathcal{F}$,O) $\Rightarrow$ ResOk($\mathcal{F}$).

We see that every feasible schedule $\mathcal{F}=(G,S,C)$ satisfies exactly one segment schedule (X,Y) and exactly one sequence schedule O, i.e.:

- $(X,Y):(\forall t:t \in T:(X(t),Y(t)) \in ResSetAvail(G(t)) \wedge X(t) \leq S(t) \wedge C(t) \leq Y(t))$
- $O=\{(t,u) \mid t,u \in T \wedge G(t) \cap G(u) \neq \emptyset \wedge C(t) \leq S(u)\}$

The reverse is of course not true: There can be many (but also zero!) feasible schedules $\mathcal{F}=(G,S,C)$ which satisfy an arbitrary segment schedule and sequence schedule for G.

4.2. Finding an optimal solution

Using segment- and sequence schedules, we can give an algorithm which finds an optimal solution for the scheduling problem in finite time. First, we consider a subproblem. Suppose given are:

- A complete resource schedule $G \supseteq G_p$.
- A complete segment schedule (X,Y) for G.
- A complete sequence schedule O for G,

and we are searching for the optimal schedule (G,S,C) which satisfies (X,Y) and O. The problem is calculating time schedules S and C which satisfy the following rest constraints (while minimizing Z)[3]:

- $(\forall t:t \in T:a(t) \leq S(t) \wedge C(t) \leq b(t))$ (BorderOk)
- $(\forall t,u:(t,u) \in dom(\alpha):\alpha(t,u) \leq S(u)-C(t))$ (MinWaitOk)
- $(\forall t,u:(t,u) \in dom(\beta):S(u)-C(t) \leq \beta(t,u))$ (MaxWaitOk)
- $(\forall t:t \in T:C(t)=S(t)+P(t,G(t)))$ (TimesOk)
- $(\forall t:t \in T:X(t) \leq S(t) \wedge C(t) \leq Y(t))$ (SegsOk)
- $(\forall t,u:(t,u) \in O:C(t) \leq S(u))$ (OrderOk)
- $(\forall t:t \in dom(S_p):S(t)=S_p(t))$ (Extension requirement)
- $(\forall t:t \in dom(C_p):C(t)=C_p(t))$ (Extension requirement)

We observe that these constraints are all **linear** (in)equalities in variables S and C. This subproblem can therefore be solved with one of the well-known algorithms (e.g. simplex-method) for linear programming problems using Z as target-function[4]. Since we know that every feasible schedule $\mathcal{F}$ consists of a complete resource schedule G and that it satisfies a complete segment schedule and sequence schedule, it is easy (yet time-consuming) to find an optimal schedule by traversing all possible combinations of resource-, segment- and sequence schedules and calculating time schedules by solving the mentioned subproblem using an L.P.-algorithm. It is evident that this process walks

---

[3] According to the 2 properties of segment- and sequence schedules, we may leave out constraints ResOk and AvailOk as these are implied by the fact that $\mathcal{F}$ should satisfy the complete (X,Y) and O.

[4] In spite of the max- and min-components in the target function, we can write Z as a linear function by adding some (linear) constraints (see e.g. [5], pp. 14-21).

through all feasible, and therefore also all optimal, schedules. The algorithm runs in finite time because there are only a finite number of (complete) resource-, segment- and sequence schedules. The algorithm becomes:

```
function OptSolve():𝔹 x (T→℘(R)) x (T→ℝ) x (T→ℝ) x ℝ
{ this function returns a 5-tuple (b,G,S,C,v), where:
    b ⇒ "the scheduling problem has feasible solutions (schedules)" ∧
        "(G_opt,S_opt,C_opt) is an optimal schedule" ∧
        "v=Z(G_opt,S_opt,C_opt) is the optimality value of (G_opt,S_opt,C_opt)" ∧
  ¬b ⇒ "the scheduling problem has no feasible solutions"
}
var G_opt:T→℘(R); S_opt,C_opt:T→ℝ; val:ℝ; solve:𝔹;
begin
 val:=+∞;
 for all G:G∈T→℘(R) ∧ (∀t:t∈T:G(t)∈F(t)) do
  for all X,Y:FullSeg(G,X,Y) do
   for all O:FullOrder(G,O) do
    (S,C,solve):=LpSolve(G,X,Y,O);
    if solve ∧ Z(G,S,C)<val then G_opt,S_opt,C_opt,val:=G,S,C,Z(G,S,C) fi
   od
  od
 od;
 { val=+∞ ⇒ "The problem has no feasible solutions" ∧
   val≠+∞ ⇒ "(G_opt,S_opt,C_opt) is an optimal schedule with value val"
 }
 return(val≠+∞,G_opt,S_opt,C_opt,val)
end;
```

Here we assume the availability of a linear programming problem solver "LpSolve" which calculates an optimal solution for the earlier mentioned sub-problem.

This algorithm is unusable for all but the very simplest problem instances because of the combinatorial explosion of the number of possible combinations of resource-, segment- and sequence schedules. Therefore, we have to invent an approximation method, which is the subject of paragraph 4.4. First, however, we will reformulate our scheduling problem into a form which reflects the problem in a more natural way.

4.3. Reformulation of the scheduling problem

We start with the introduction of the concept **extended schedule**. This is a 6-tuple (G,X,Y,O,S,C), where:
- G is a resource schedule.
- (X,Y) is a segment schedule for G.
- O is a sequence schedule for G.
- S and C are starting- and completion time schedules.

An extended schedule is **complete** iff G, (X,Y), O, S and C are complete.

Next, we extend the definition of 4 kernel constraints, and introduce 2 new constraints (for extended schedules). Let $\mathcal{F}$=(G,X,Y,O,S,C) be an arbitrary complete extended schedule. We define:
- BorderOk($\mathcal{F}$):=(∀t:t∈T:a(t)≤S(t) ∧ C(t)≤b(t))
- MinWaitOk($\mathcal{F}$):=(∀t,u:(t,u)∈dom(α):α(t,u)≤S(u)-C(t))
- MaxWaitOk($\mathcal{F}$):=(∀t,u:(t,u)∈dom(β):S(u)-C(t)≤β(t,u))

- TimesOk($\mathcal{F}$):=($\forall$t:t$\in$T:C(t)=S(t)+P(t,G(t)))
- SegsOk($\mathcal{F}$):=($\forall$t:t$\in$T:X(t)$\leq$S(t) $\wedge$ C(t)$\leq$Y(t))
- OrderOk($\mathcal{F}$):=($\forall$t,u:(t,u)$\in$O:C(t)$\leq$S(u))

A complete extended schedule $\mathcal{F}$=(G,X,Y,O,S,C) is **feasible** iff it satisfies these 6 constraints. The optimality value Z($\mathcal{F}$) of a feasible extended schedule $\mathcal{F}$=(G,X,Y,O,S,C) is defined as Z(G,S,C). A feasible schedule $\mathcal{F}$ is **optimal** if it has minimal value Z($\mathcal{F}$). An extended schedule $\mathcal{F}_1$ is an **extension** of extended schedule $\mathcal{F}_2$ iff all subschedules in $\mathcal{F}_1$ are a superset of the corresponding subschedules in $\mathcal{F}_2$ (analogous to the extension of an ordinary schedule).

We see:

(1) When $\mathcal{F}$=(G,X,Y,O,S,C) is a feasible extended schedule, then (G,S,C) is a feasible 'normal' schedule. When $\mathcal{F}$ is optimal, (G,S,C) is optimal.

(2) When (G,S,C) is a feasible schedule, then $\mathcal{F}$=(G,X,Y,O,S,C) is a feasible extended schedule. When (G,S,C) is optimal, $\mathcal{F}$ is optimal by choosing (X,Y) and O as mentioned on the end of paragraph 4.1.

Hence: For every feasible/optimal schedule (G,S,C) we can find a feasible/optimal **extended** schedule (G,X,Y,O,S,C) and vice-versa. The following reformulated scheduling problem is therefore functionally equal to the original one: Given kernel data (R,E,T,a,b,F,P,α,β), optimality data (Q,d,u,v,w) and an extended start schedule (G_p,X_p,Y_p,O_p,S_p,C_p): Construct an optimal extended schedule $\mathcal{F}$=(G,X,Y,O,S,C) which is an extension of the extended start schedule.

This specification will be used throughout the rest of this section. It highlights more of the real scheduling problems, which are the determination of a resource-, segment- and sequence schedule. Now, we have a formulation of the problem which contains all these relevant concepts. In the rest of this section when we write 'schedule', we mean 'extended schedule' unless stated otherwise.

4.4. A 4-phase approximation method

Our problem is finding a method which generates a good complete schedule $\mathcal{F}$=(G,X,Y,O,S,C) which is an extension of start schedule (G_p,X_p,Y_p,O_p,S_p,C_p) within reasonable time limits. As indicated in the previous paragraph, it is not acceptable to walk through all possible combinations of resource-, segment- and sequence schedules. We propose, as a first approximation step, to determine the schedule in 4 separated phases:

- Phase 1 (resource scheduler): Determine resource schedule G.
- Phase 2 (segment scheduler): Determine segment schedule (X,Y) for G.
- Phase 3 (sequence scheduler): Determine sequence schedule O for G.
- Phase 4 (time scheduler): Determine starting- and completion time schedules S and C.

From the following reasons it follows that these 4 phases should be performed in the order given:

- Only when the resourceset for a task is known, we can plan that task in a segment, because the possible segments follow from that resourceset.
- When tasks are already planned in segments, many sequences for conflicting task pairs are already implicitly given by the segment schedule. Our sequencing task will therefore be much easier (it has less work to do, and all tasks are already planned in a time segment, which makes the decision situation for the sequencer less difficult.)
- We can only determine time schedules via L.P. when complete resource-, segment- and sequence schedules are available.

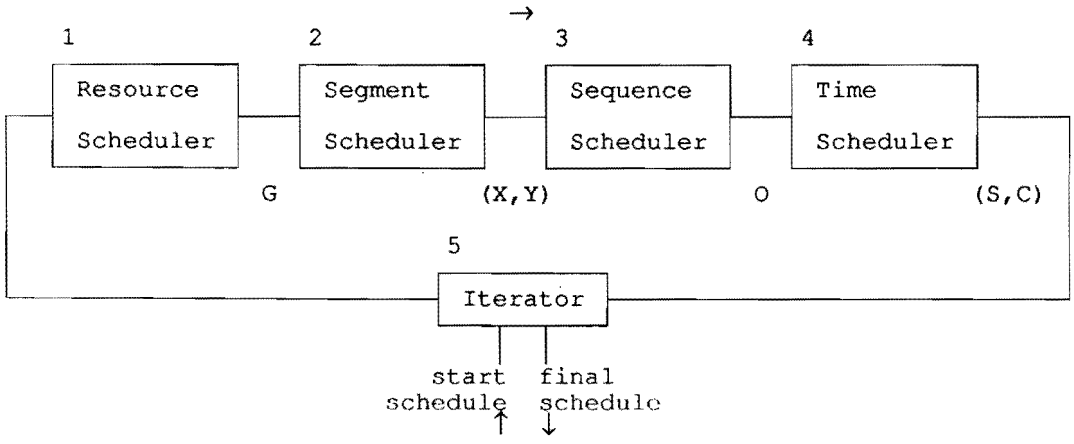
Our proposed approach has, among others, the advantage that we can handle the 4 phases more or less isolated (a kind of divide and conquer strategy). We now have to develop 4 relatively simple schedulers instead of 1 very complex one. Note though that there are still some problems to be solved. The most important ones are:

- We have to invent criteria which indicate when (combinations of) resource-, segment- and sequence schedules are **good** (can be part of feasible and/or optimal schedules). In the first 3 phases we then have to search for (sub)schedules which satisfy these criteria. In general we need evaluation functions which assign values to arbitrary schedules. These values should be a measure for the quality of such a schedule.
- In the first 3 phases it is not possible to walk through all possibilities for the resource-, segment- and sequence schedules since these sets are much too large for practical problem instances (yet much smaller than the set of all combinations of these schedules). We see that the reduction of the problem size via the 4-phase method is not enough: We need a second approximation method for the first 3 scheduling phases.

In the next 2 paragraphs we will shortly discuss these 2 problems.

A possible disadvantage, apart from the non-optimality, is the following. It may happen that in a certain phase a scheduling decision is taken which appears to be bad (i.e. leads to a bad schedule) in a later phase. When we hold on to our strict 4-phase method, it is not possible anymore to correct such an error. It is therefore necessary to make our method more flexible. One easy way of doing so is introducing a separate fifth phase ('Iterator') which removes any 'bad' parts from a complete schedule (keeps only the good parts) and reruns the 4 phases until an acceptable schedule is obtained or until the user finds the resulting schedule good enough. Note that the removal of bad parts may also be done by hand. Of course, more advanced methods are possible, e.g. removing errors **during** the scheduling process, and not afterwards as proposed.

Graphically, we can depict our scheduling system as follows:



When phase 1 starts with a begin schedule $(G_p,X_p,Y_p,O_p,S_p,C_p)$, after scheduling phase 1 we have obtained a schedule (G,X_p,Y_p,O_p,S_p,C_p); after phase 2 we have a schedule (G,X,Y,O_p,S_p,C_p) etc. till (G,X,Y,O,S,C) after phase 4 (the begin schedule is extended to a complete schedule). The iterator (phase 5) then removes possible bad parts from this (complete) schedule and reruns the 4 phases with a new begin schedule, or it stops and returns the complete schedule as result.

Remark: Our scheduling system makes global decisions in the beginning, and detailed decisions in the end: The later the phase, the smaller the freedom for the placement of tasks (the possible time segments in which tasks may be planned will be reduced continuously during the scheduling process).

4.5. Evaluation of schedules

In this paragraph we shortly consider **evaluation functions**, which assign values to arbitrary schedules.

These values should indicate if such a schedule is extendible to a **good feasible** schedule. These functions will be used by the search method which will be described in the next paragraph. A search process will walk through many schedules which will be rejected or selected as part of our ultimate schedule. This selection or rejection is based upon the values of the evaluation functions. They will therefore be very important for our results. These functions are in fact the **intelligence** of the scheduling system. Some important aspects of schedules are:

- Feasibility (schedules can contain errors).
- Occupation rate of the resource(set)s.
- The optimality (distances to duedates).
- Future 'planability' of tasks (freedom of placement, processing time etc.).

For each of these (and other) aspects of a schedule, an evaluation function should be available. Then, every schedule has a number of evaluation values, each of which is a measure for the quality of that schedule. These values will then be combined to one value by a weighted summation. This ultimate value is a measure for the overall quality of that schedule (e.g. the lower this value, the better the schedule).

We have implemented a number of fairly general evaluation functions in our prototype which consider the aspects mentioned above. The user is able to enter weights for the different functions. This is an essential property, as different problem instances need different weights for obtaining a good solution (the importance of a certain evaluation function depends on the problem type. Sometimes the occupation rate is important, sometimes the optimality etc.)

4.6. Searching

The first 3 scheduling problems (resource-, segment- and sequence scheduler) are very similar, as they can all be formulated as a so-called 'unstructured search problem':

definition
An **unstructured search problem** (USP) is a pair $(S,\leq)$, where:
- S is the **search space** (an arbitrary, non-empty, finite set)
- $\leq \in S \times S \rightarrow \mathbb{B}$ is the **compare function** (transitive and reflexive: $\leq$ is a so-called pre-order). It is used to distinguish between elements of the search space.

A **solution** for a USP $(S,\leq)$ is an element $s \in S$. An **optimal solution** is a solution s satisfying $(\forall t: t \in S: s \leq t)$.
□

Our first 3 scheduling problems can be formulated as USPs by choosing S as the set of resource-, segment- resp. sequence schedules. The compare function follows from the evaluation values of schedules using the earlier mentioned weighed combination of those values.

The elements of the search space have a special structure for our 3 problems. This observation leads to a more specific search problem, the so-called 'allocation problem':

definition
An **allocation problem** (AP) is a 4-tuple $(V,A,f,\leq)$, where:
- V is an arbitrary, non-empty, finite set of **objects**.
- A is an arbitrary, non-empty, finite set of **allocations**.
- $f \in V \rightarrow \wp(A)$ is a function which gives the possible allocations for each object. We require that every object has at least one possible allocation, so $(\forall v: v \in V: f(v) \neq \emptyset)$.

Now, we define the **search space** for AP $(V,A,f,\leq)$ as the set $S := \{s \mid s \in V \rightarrow A \wedge (\forall v: v \in V: s(v) \in f(v))\}$.
- $\leq \in S \times S \rightarrow \mathbb{B}$ is the **compare function** (transitive and reflexive).

A **solution** of an AP $(V,A,f,\leq)$ is a function (schedule!) $s \in V \rightarrow A$ where every object v has a possible

allocation s(v). Hence, such a solution is an element of the search space S for (V,A,f,≤). An **optimal solution** is a solution s satisfying (∀t:t∈S:s≤t).
□

Note: every AP belongs also to the class of USPs: (S,≤) is the USP corresponding to AP (V,A,f,≤) with search space S.

The following choices for V,A and f show how our 3 scheduling problems can be modelled as an allocation problem.
- resource scheduler: V=T\dom(G_p); A=℘(R); f={(t,u)|t∈V ∧ u=F(t)}
- segment scheduler: V=T\dom(X_p); A=ℝxℝ; f={(t,u)|t∈V ∧ u=ResSetAvail(G(t))}, where G is an earlier determined complete resource schedule.
- sequence scheduler: V={ {t,u}|t,u∈V ∧ G(t)∩G(u)≠∅ ∧ (t,u)∉O_p ∧ (u,t)∉O_p}; A=T; f={(p,p)|p∈V}, where G is an earlier determined complete resource schedule. Here, an object is a unordered pair of tasks. An allocation for such a pair is the task which is processed first.

Now, we have to invent a search method which doesn't walk through all elements of the search space, but yet finds a good solution. In our implementation we have choosen Greedy Search (simple and easy), but other methods are certainly possible. Greedy search works as follows for an allocation problem (V,A,f,≤) with search space S and so-called extended search space U:={t|t⊆s ∧ s∈S}. We start with an empty schedule s:=∅ which will be stepwise extended to a complete schedule s∈S. In every step we add one object v with allocation a∈f(v) to s. When determining this allocation a for v, we consider a certain small **local environment** of objects O on which v has much influence[5]. The algorithm works as follows:

```
procedure GreedySearch;
var s:U; v:V; O:℘(V); a:A;
begin
  s:=∅;
  while dom(s)≠V { not all objects allocated } do
    v:=GetObject(s);          { Select an object v with no allocation, so v∈V\dom(s) }
    O:=GetEnviron(v,s);       { Select a (small) environment O⊆V\(dom(s)∪{v}) of objects
                                with no allocation where v has much influence on }
    a:=FullSearch(s,v,O);     { Evaluate all schedules s'∈U with s'⊇s and dom(s')=
                                dom(s)∪O∪{v}. Let t be the best of these schedules, then
                                select a=t(v) }
    Alloc(v,a)                { Add pair (v,a) to s }
  od
  { s is the obtained (complete) schedule }
end;
```

Some notes concerning GetObject and GetEnviron:
- GetObject: This function determines the order in which the objects will be processed. We have chosen to walk the time-axis from the left to the right (w.r.t. to deadline): most urgent job first. Of course, there are other possibilities.
- GetEnviron: The maximum number of allowed objects in O will be given by a user-parameter. This determines the speed of the search process (the number of schedules which have to be processed in step FullSearch). In our implementation we have choosen a very simple way of placing objects in O: the objects which are closest to v on the time-axis will be selected first. Of course, much more

[5] This means that the possibility of finding a good allocation for an object in O strongly depends upon the allocation made for v.

difficult (and better) selection methods are possible.

4.7. Test results

In this paragraph we will mention some interesting first results of our prototype. This will be done by discussing our 3 most important test cases.

4.7.1. A (fictive) photo development- and printing company

This has been our primary test case (100 tasks, 9 resources) as it is a very difficult planning problem which contains all aspects of resource-constrained project scheduling. For the ASCII-problem specification and a feasible schedule of this case we refer to the appendix of [4]. This fictive problem has been developed by us to have a good test case at hand.

A close look at the problem and feasible schedule reveals that the problem is indeed very difficult because many resources will be almost fully occupied (100%) in a feasible schedule and relatively many conflicting task pairs will occur. The automatic scheduler should therefore distribute the resources very smoothly over the tasks and time to obtain a schedule with little errors.

Another aspect which complicates planning is that strong minimal- and maximal waiting time constraints are posed between many tasks. Every scheduling decision taken for a task(pair) has therefore relatively much influence on the future 'planability' of other tasks.

It is clear that we have to strive for a schedule which contains as few errors (wrongly planned tasks) as possible because a very small fraction of the complete schedules is also feasible. The optimality criterion is therefore not very important in this case.

Our best result achieved so far is a schedule where 6 tasks have been wrongly planned. This is achieved using a search depth of 2 in the first 3 scheduling phases and 'playing' a little with the weights for the evaluation functions (tuning). It appears, and was also to be expected, that especially the evaluation functions which control the occupation of resources are important, which means that they should have relatively large weights.

Obtaining the result takes 2 iteration steps: After the first step 8 tasks are in error. Removing these tasks and re-running the schedulers results in the schedule with 6 errors. Further iterations give no more improvements. Computing the schedule requires approximately 10 minutes computing time on a 10 Mhz. IBM-AT compatible computer (all tests have been run on this machine). We finally note that 3 of the error tasks can be planned correctly easily by hand using the manual planner. The other 3 require probably extensive changes in the schedule.

4.7.2. A 6*6*6 Job shop

Our next test case was the 6*6*6 Job-shop problem which appears in [6]. We note that this is a pure sequencing problem, so only the performance of phase 3 of the scheduler will be tested with this case. Also we note that our evaluation functions are not especially written for this kind of relatively simple structured problems. They should be able to cope also with more complex problems. Nevertheless, a Job-shop can be a good test case to see whether our fairly general evaluation functions are able to cope reasonably with such specific problems. We finally note that finding a feasible schedule is not difficult. The problem is finding an optimal schedule (all tasks processed as early as possible).

This reasonable small case contains 6 projects each consisting of 6 tasks. These 36 tasks must be processed with 6 resources (machines). These resources are already allocated to the tasks, and the problem consists of determining an order for 90 conflicting task pairs. A known optimum is 55 [6].

In the current prototype implementation we can use 3 evaluation functions for the sequencer:

1. Feasibility. This function indicates (via a non-negative value) whether the current schedule can be extended to a feasible schedule. The higher the value, the more errors the schedule contains. A

value of zero indicate no errors so far.

2. Freedom. The value of this function is a measure for the freedom with which tasks can be planned in the future (distance between the borders for the possible starting- and completion times of the tasks).
3. Duedate distance. The value of this function is a measure for the value of the optimality function Z for a feasible schedule which is an extension of the schedule considered.

The feasibility function has a fixed weight of 1e20 (10^{20}) so that it dominates the other functions. In the tests we have used 5 different combinations of weights for the other 2 functions. Our results are given in the following table:

Search depth:	1	2	3	4
(mean) scheduling time:	0:10	0:32	1:25	3:40
Parameter combination (freedom, duedate)				
(1e10,1)	61	61	60	60
(10,1)	55	61	60	60
(5,1)	55	61	60	60
(1,1)	55	60	60	55
(1,1e10)	55	55	60	60

We see that certain good choices for the weights of the evaluation functions lead to the optimal solution. This is a promising result.

4.7.3. A 10*10*10 Job shop

This is the well-known notorious 10*10*10 Job-shop problem, also taken from [6]. The same remarks made for the 6*6*6 problem apply here. This problem requires the calculation of a sequence for 450 task pairs. A known optimum is 930 ([3], page 49).

Our results are:

Search depth:	1	2	3
(mean) scheduling time:	1:46	6:55	21:06
Parameter combination (freedom, duedate)			
(1e10,1)	1134	1088	1166
(10,1)	1106	1088	1151
(5,1)	1106	1088	-
(1,1)	1282	1158	-
(1,1e10)	1243	1318	-

The best result we obtained is 1088 at search depth 2. This is reasonable (distance 159 to the optimum, approx. 17%) knowing that our evaluation functions are not especially written for Job-shops and that this problem is a very difficult one. Of course, specially developed algorithms for such problems obtain better results, see e.g. [3], page 49. We expect that implementing problem specific evaluation functions might improve our results.

4.8. Some remarks on the automatic scheduler

An important conclusion which may be drawn from our (though limited) experiments is that a larger search depth does not necessarily lead to better results (see e.g. the tables in the previous 2 sub-paragraphs. Also our experiments with the case in paragraph 4.7.1 indicate this). A reasonable explanation for this fact is that the search depths which can be selected are not large enough to have a predictable influence on the results (because of the time limits). A choice for another small search depth may have a positive or negative effect because it leads to other, but not necessarily better results. The search depth can therefore be considered as a random effect.

An important fact is that experimenting with the evaluation functions and weights can lead to consistently better results (see again the tables in the previous paragraphs). It therefore seems reasonable to direct further research to the area of evaluation functions and the systematic finding of good parameters/weights for them. Another option is implementing an alternative (e.g. genetic) search algorithm.

All in all we are reasonably satisfied with our first results. We expect that further research might improve our results considerably. Among others, much work has to be done on the following topics:

- Finding good evaluation functions and weights for certain problem types. One very interesting point is building a library of evaluation functions from which the system automatically extracts the important ones, given the problem it has to solve (different problem types often require also different evaluation functions for obtaining good results).
- Other search techniques.
- Better ways of combining the 4 phases (other iterators).

We feel that placing tasks in segments is essentially different from sequencing because segment scheduling is a more global task which requires other evaluation functions than the sequencer. This is one of the reasons why we have kept resource availabilities in our problem definition (see also the remark at the end of paragraph 2.3) and made 2 different schedulers for segment- and sequence scheduling instead of one which handles both.

5. User interface and manual planning.

With respect to the user interface we can distinguish output- and input aspects, i.e. the screens and the user actions.

During manual planning the screen looks like an electronic planning-board (see fig. 4). A graphical representation of the schedule as Gantt diagram is displayed to show the allocation of tasks to resources (y-axis) over time (x-axis).

Because of the restrictions in screen sizes, it is important to show only the most relevant information in a rather compact and surveyable way. In this aspect graphical information is preferable to alpha-numeric information.

While planning we should steadily be aware of the quality of the schedule. Not only the value of the selected criterion should be visible and always up-to-date, but also other characteristic values, like a measure of completeness, the percentage of incorrectly planned tasks, etc.

It is not forbidden to work with infeasible schedules. Most of the constraints are weak, i.e. the system does not guarantee the feasibility of the schedule with respect to these constraints. However violations of weak constraints are visualised by means of special colouring. So the user is aware of the measure

of infeasibility.

With respect to the userfriendliness, it is important that user actions should be performed with minimal user effort and a minimal chance of giving wrong commands. So all user input is driven by menus and function keys. The depth of the menu structure is kept as small as possible.

During manual planning two groups of functions are available to the user. First there are functions for navigation and inspection. With these we can scroll and zoom the planboard and inspect all kinds of relevant information, like special characteristic values, but also the problem-instance data.

Secondly there is a group of decision handling functions. With these we can add task allocations to the schedule, remove them from the schedule, or shift task allocations in time.

To add a task allocation to the schedule the user may first choose an unplanned task and then select a feasible resourceset; he may also start by choosing an arbitrary resourceset and then select an unplanned task, which is feasible. If there is a time interval, where all resources of the selected set are available for task execution, the system will propose the first interval. The user can accept the interval and specify the exact starting time or can ask for the next appropriate time-interval. However, if there is no suitable time interval, the user can still specify a starting date. Before allocating the task, the system checks against any violations of constraints, like precedence, release times, deadlines, but also time overlapping. In the case of any infeasibility the user will be notified and it is up to the user to effect the decision.

The removal of a task is much simpler. The user moves the cursor to the relevant task at the planning-board before pressing the relevant function key.

A task allocation can be shifted in time by removing the task temporarily from the planning board and putting it in a list of "semi-planned" tasks (the resourcesets remain allocated, but not the time intervals). A task can be replanned by selecting a task from that list and by moving the cursor along the time axis.

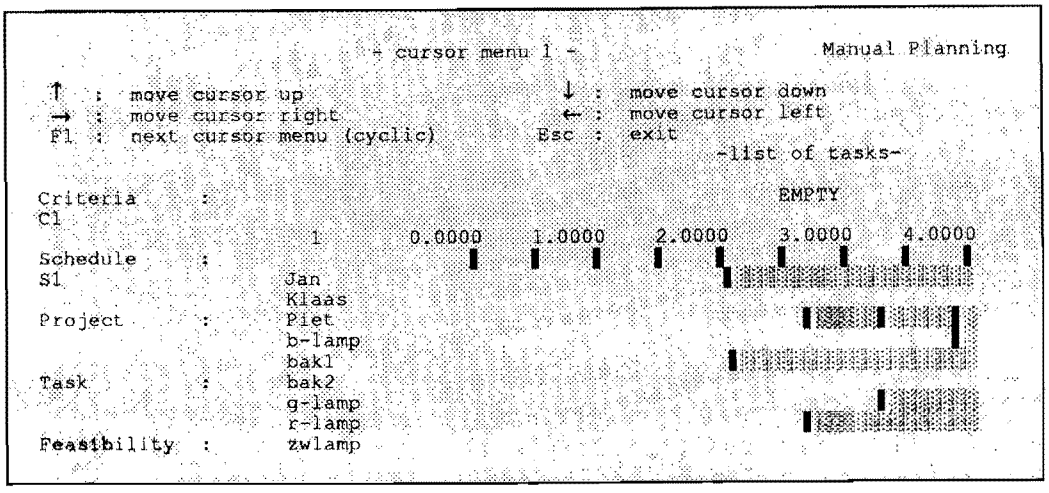


Fig. 4: The electronic planning board

6. Conclusions and future research.

One of the biggest problems we encountered was the inflexibility of the software and its sensibility to change requests. This is of course a general software engineering problem, not only related to our project, but decision support systems seem to suffer from it extremely. A minor change in the underlying model, or in the limiting constraints, can have the consequence of redesigning the whole system.

The reason for that can be found in the fact that conventional DSS-systems are dominated by the algorithmic aspect. Mostly mathematicians working in the field of operations research were involved in the development. Although the algorithmic aspect may be very interesting from the mathematical point of view, the software dealing with it plays an inferior role related to other software components like the management of decisions and the user interface.

The application of techniques from the field of artificial intelligence may be a step in the right direction. More general and flexible search methods can be used for a huge class of planning problems [8]. The idea of an expert system shell and a rule base with the flexibility to add and change complex structured rules and metarules can be applied to constraints and search methods in a planning situation [7].

Anyway we should be aware that by developing a DSS-system we are dealing in the first place with a software engineering problem. Perhaps we should intend to build optimal software to produce practicable schedules, instead of trying to produce optimal schedules with impracticable software.

Acknowledgements

We like to thank Lida van de Bent, Jan Böinck, Clen Lubbers, Frans van der Meeren and Reinoud van Dommelen for their contributions to the programming of the system, and Ad Aerts and Kees van Hee for many valuable suggestions. Furthermore we thank Wim Keulemans for making available the L.P.-software.

Literature

[1] J.M. Anthonisse, K.M. van Hee, J.K. Lenstra (1987). "Resource-Constrained Project Scheduling:an International Exercise in DSS Development", Note OS-N8702, Centre for Mathematics and Computer Science, Amsterdam.

[2] A. Jansen (1990). "Een aanpak van resource-constrained project scheduling problemen", Master's Thesis, Eindhoven University of Technology, Eindhoven. (in Dutch)

[3] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys (1989). "Sequencing and Scheduling:Algorithms and Complexity", Report BS-R8909, Centre for Mathematics and Computer Science, Amsterdam.

[4] "User manual of PLATE: a Decision Support System for resource-constrained project scheduling problems", Eindhoven University of Technology, Eindhoven (1990).

[5] K.G. Murty (1983). "Linear programming", John Wiley & Sons.

[6] H. Fisher, G.L. Thompson (1963). "Probabilistic Learning Combinations of Local Job-shop Scheduling Rules", in: J.F. Muth, G.L. Thompson (eds.), "Industrial Scheduling", Prentice Hall (pp. 225-251).

[7] A.E. Eiben, K.M. van Hee (1990). "Knowledge representation and Search Methods for Decision Support Systems", in: W. Gaul, M. Schader (eds.), "Data, Expert Knowledge and Decisions", Computer and System Science Series, Springer-Verlag.
[8] K.M. van Hee, A. Lapinski (1989). "OR and AI approaches to decision support", Decision Support Systems 4, pp. 447-459.

Appendix: Notational conventions

In this appendix some of our, possibly not completely standard, notational conventions will be explained.
Let P and Q be arbitrary boolean predicates in n variables. Let R be an expression in n variables with a set as result type. Let S be an expression in n variables with a number as result type. We define:
- $(\forall a_1,..,a_n{:}P(a_1,..,a_n){:}Q(a_1,..,a_n))$ is a boolean predicate saying that for all variables $a_1,..,a_n$ satisfying P also Q holds.
- $(\exists a_1,..,a_n{:}P(a_1,..,a_n){:}Q(a_1,..,a_n))$ is a boolean predicate saying that there exist variables $a_1,..,a_n$ satisfying P for which also Q holds.
- $(\cup a_1,..,a_n{:}P(a_1,..,a_n){:}R(a_1,..,a_n))$ is an expression giving the union of all sets $R(a_1,..,a_n)$ where variables $a_1,..,a_n$ satisfy P.
- $(\cap a_1,..,a_n{:}P(a_1,..,a_n){:}R(a_1,..,a_n))$ is an expression giving the intersection of all sets $R(a_1,..,a_n)$ where variables $a_1,..,a_n$ satisfy P.
- $(\mathbf{MAX}a_1,..,a_n{:}P(a_1,..,a_n){:}S(a_1,..,a_n))$ is an expression giving the maximum of all values $S(a_1,..,a_n)$ where variables $a_1,..,a_n$ satisfy P.
- $(\mathbf{SUM}a_1,..,a_n{:}P(a_1,..,a_n){:}S(a_1,..,a_n))$ is an expression giving the sum of all values $S(a_1,..,a_n)$ where variables $a_1,..,a_n$ satisfy P.

Let A and B be arbitrary finite sets, We define:
- $A\rightarrow B$ is the set of all **functions** from A to B. As usual, a function is simply a set of pairs. $f=\{(a_1,b_1),..,(a_n,b_n)\}$ is a function from A to B ($f\in A\rightarrow B$) iff:
  0. $(\forall i,j{:}1\le i<j\le n{:}a_i\neq a_j)$
  1. $\{a_1,..,a_n\}=A$
  2. $\{b_1,..,b_n\}\subseteq B$
- $A\xrightarrow[p]{}B$ is the set of all **partial functions** from A to B. The set of pairs $f=\{(a_1,b_1),..,(a_n,b_n)\}$ is a partial function from A to B ($f\in A\xrightarrow[p]{}B$) iff:
  0. $(\forall i,j{:}1\le i<j\le n{:}a_i\neq a_j)$
  1. $\{a_1,..,a_n\}\subseteq A$
  2. $\{b_1,..,b_n\}\subseteq B$
- The **domain** dom(f) of a (partial) function f is defined by $dom(f):=\{a\mid\exists b{:}true{:}(a,b)\in f\}$.
- The **powerset** $\wp(A)$ of set A is defined by $\wp(A):=\{C\mid C\subseteq A\}$.
- $\mathbb{R}$ denotes the set of all real numbers. $\mathbb{R}^+$ denotes the set of all positive real numbers. $\mathbb{R}_0^+$ denotes the set of all non-negative real numbers. $\mathbb{B}$ denotes the set of booleans.

Let TP be an arbitrary type (set of values). Let a be an arbitrary non-negative integer. We define:
- TP^a denotes the set of all rows with a elements of type TP. Let q be an arbitrary row with a elements, then is q(n) defined as the n-th element of the row (counting starts from 0, so $0\le n<a$).

Example: Let $q:=<1.5, 2, 0, 2.25, 3>\in(\mathbb{R}_0^+)^5$, then q(0)=1.5 and q(2)=0.