A decision support systems shell in Prolog

van Hee, K.M.; Nuijten, W.P.M.

Published: 01/01/1990

Document Version
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

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

A DECISION SUPPORT SYSTEMS SHELL IN PROLOG

by

K.M. van Hee
W.P.M. Nuijten

NFI 11.90/01

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

Juli 1990

K.M. van Hee
Eindhoven University of Technology
Faculty of Mathematics and Computing Science
P.O. Box 513, 5600 MB Eindhoven, the Netherlands

W.P.M. Nuijten
Eindhoven University of Technology
Faculty of Mathematics and Computing Science
P.O. Box 513, 5600 MB Eindhoven, the Netherlands

A Decision Support Systems Shell in Prolog

K.M. van Hee and W.P.M. Nuijten
Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

ABSTRACT
Many operational planning problems can be considered as searching for an element in a finite set. Classical approaches lead to sophisticated combinatorial optimization algorithms that exploit a lot of the structure of the decision situation. Often these systems are not easy to adapt to new constraints on solutions imposed by the decision makers.
Our approach is to use general search methods that are easy to adapt in case of new constraints. In general they give less good solutions but are more robust for new constraints. A general-purpose search shell based on these search methods is sketched using Prolog. As an illustration two examples are given: the travelling salesman problem and precedence constrained scheduling.

1. Introduction

We use a well-known paradigm of systems theory cf. Checkland (1981) where there are two communicating systems, one to be controlled, called target or object system, and a control system, often called the information system. The target system sends status information to the information system and the information system sends decisions to the target system. The term 'decision' is used for a unit of information. A decision might be a simple control action that influences the target system for a short period of time, or a set or sequence of such actions that controls the target system for a longer period of time. An example of the last case is a production schedule or a route plan. There are at least two protocols for the communication between target and information system: one where the information system sends periodically new decisions to the target system and one where the information system only sends new decisions if the observed state of the target system satisfies some condition. It is clear that in

general two successive decisions are dependent. For instance if the decision is a production schedule the next decision might be an adaptation of the foregoing one. Although decision making is a process, we consider only the production of one decision.
A decision support system (DSS) is an automated part of an information system that assists users in making decisions by:
- performing data management functions,
- evaluating the effects of user proposed decisions,
- generating decisions that satisfy some user defined conditions.

The evaluation function can be divided into two parts: verification whether the user proposed decision is allowed and a computation of quality measures for allowable decisions.
The generation function may also be split into several parts varying from stepwise development of a decision, where the user may choose in each step from a finite set of computed alternatives, to a fully automatically computed decision.
Our definition of a DSS is more restrictive than others, cf. for instance Keen and Scott Morton (1978). As advocated in Eiben and van Hee (1990) a DSS should be developed in the evolutionary style. This means that first the data management functions are developed, then the verification function, then the other evaluation functions, then the stepwise generation support and finally the automatic generation of decisions. In this approach the users experience with the system evolves simultaneously with the functionality of the system.
Many DSS's for operational planning are based on sophisticated algorithms that rely on specific properties of the decision situation. For instance the simplex algorithm requires the object function and all constraints to be linear, algorithms for job shop scheduling may require that for each task there is only one possible machine available etc.. In many practical situations however the requirements of these algorithms are often not satisfied. Even if they are satisfied at the time the DSS was developed, they may be unsatisfied later due to changes in the decision situation. For DSS's for tactical or strategic decision making this phenomenon is not so severe. In these cases a DSS often considers a simplification of the real situation and the proposed decisions are of a rather global nature such as the capacity of storage space or the number of vehicles. In operational decision making however it is unacceptable that a DSS suggests an unexecutable decision or a decision that does not exploit all possibilities the decision maker has.
Although specialized algorithms may find optimal solutions in case all requirements are fulfilled we advocate to use a more robust approach that sometimes gives less good decisions but that guarantees the proposed decisions to be executable. In Van Hee and Lapinski (1989) an abstract machine for decision making that could easily be adapted to new restrictions in the decision situation is presented. In Eiben and van Hee (1990) a method for obtaining robust DSS's based on graph search methods is introduced. The method was described for resource constrained project scheduling. Here we generalize the method a bit and describe how it can be implemented in Prolog to obtain a DSS shell. The shell can be used to create specific DSS's for specific situations. If the situation changes one can easily generate a modified DSS.
In the shell there is generic knowledge: this knowledge or code will be part of all systems composed with the shell. This part has to be defined by the designer of the shell. Then there is a part of the knowledge that is fixed for a problem type, this is called domain knowledge. This part has to be defined by the DSS-expert. The third

part of knowledge is dependable on the instance of the decision situation and can be defined by the user or decision maker (instance knowledge). This part is not restricted to factual knowledge as usually is the case but it also concerns rules such as restrictions on decisions and goals to be met.
The paper is organized as follows. In section 2 a method for the construction of DSS's is sketched. In section 3 a Prolog implementation of the generic knowledge of the shell is given. In section 4 we consider two problem types to illustrate how the domain knowledge can be defined. Finally in section 5 we sketch the architecture of the shell.

2. Method for the construction of DSS

In this section a method for the construction of DSS's is sketched.
The first step to be made by a DSS-expert is defining what the set of candidate solutions is, i.e. the set C in which a solution is searched. Observe that 'candidate' is a wider definition than 'solution': a solution is a candidate satisfying certain conditions. Searching is performed on the basis of a row of candidates. This list is called the *search state* of the search process. The next step is the definition of a binary relation N on the candidates. This relation is called the neighbourhood relation. Only candidates that are neighbours of candidates already in the search state are considered. So our search methods are local search methods (cf. Papadimitriou and Steiglitz (1982)). Requirements for a neighbourhood could be:
- a neighbourhood should not be too large,
- a neighbourhood should be easy to generate,
- from a starting point every candidate should be reachable through the neighbourhood relation.

Next a goal is defined, i.e. a predicate on the candidates defining which candidates are solutions. Also some additional constraints on candidates can be defined. Formally these additional constraints are represented by a predicate *feasible* on the candidates (a candidate satisfying the constraints is called feasible). These goal- and feasibility predicates should be definable by the user (instance knowledge). The feasibility predicate is used to filter the neighbours and eliminating all non-feasible neighbours. So in effect by using the feasibility predicate search can be restricted to any subset of the set of candidates. We remark that the use of a feasibility predicate often is recommended as in general the set of candidates is far too large. As the feasibility predicate is easily redefined it gives our search methods a great deal of flexibility.
A *search step* is a transition of the current search state to a new search state. This is done by:
- selecting a candidate p from the current search state,
- generating all neighbours of p,
- eliminating all non-feasible neighbours,
- updating the search state from the old search state and the feasible neighbours, i.e. determining which candidates are to be maintained in the new search state and in what order.

The first step is standardized: always select the first candidate.
A search method is an iteration process of search steps. The search process continues until a solution is found. Formally this can be described as follows.

Let $S = C^*$ (the set of all finite rows from C) be the search state space and $T : S \rightarrow S$ the transition function (defining the search steps). We remark that for the sake of convenience we sometimes treat elements of S (rows) as sets. With $s \in S$ it holds that T(s) is a permutation of a subset of $s \cup \{c \in C \mid \exists\, d \in s : (d,c) \in N \wedge \text{feasible}(c)\}$,
i.e. T(s) is a permutation of a subset of all candidates in search state s joined with all feasible neighbours of all candidates in s. A search method can be described by a search function $F : S \rightarrow S$ which is defined as follows, with $s \in S$:

$F(s) = F(T(s))$ if $\forall\, p \in s : \neg\, \text{solution}(p)$,
$F(s) = s$ if $\exists\, p \in s : \text{solution}(p)$.

Let A be defined by $A = \{ s \in S \mid \exists\, p \in s : \text{solution}(p) \}$, let $T^n(s)$ be defined by $T^n(s) = T(T^{n-1}(s))$ and $T^0(s) = s$ for $n \in \mathbb{N}$, and let s_0 be some starting search state in S.

In case for all $n \in \mathbb{N}$, $T^n(s_0) \notin A$ holds, $F(s_0)$ is undefined. In that case, since S is finite, the sequence $T^n(s_0)$ has to be periodically, i.e. for some $p \in \mathbb{N}$ and all $n \in \mathbb{N}$:

$T^{n+p}(s_0) = T^n(s_0)$.

However if F is defined for s_0 then $F(s_0)$ is totally specified by T.

3. A Prolog implementation of the generic knowledge

In this section a Prolog implementation of the generic part is given. A background knowledge of Prolog is therefore presumed (cf. Schnupp and Bernhard (1986)). As stated in section 1 we are developing DSS's based on graph search methods. A component of a graph search method is the search strategy. At the end of this section some examples of search strategies are given. By treating the generic knowledge components of the domain knowledge and the instance knowledge will be encountered. We state that in general a search method consists of an initialization step followed by zero or more search steps. We define the *state* of a graph search method to be a list of candidates that could be used for further searching (see section 2). The foregoing is reflected by the Prolog-rule:

```
solve :-
   initialize(State!),
   search_loop(State?).
```

The ? (input) and ! (output) notation is borrowed from the specification language Z (cf. Spivey (1989)). Suffixing a variable with ? means that the variable needs to have a value when the predicate is invoked. Suffixing a variable with ! means that if all input variables have a value a new value for this output variable is calculated. The notation has no further meaning: State? and State! both address the same variable.

The predicate *initialize* produces an initial list of candidates and is to be defined by the user (instance knowledge).
The predicate *search_loop* performs zero or more search steps. Searching is stopped if at least one solution is encountered in the present search state. If so all solutions in the search state are displayed. If no solution is encountered a search step is performed yielding a new search state and the search process continues with that new search state.

```
search_loop(State?) : -
   goal(State?),
   display_solutions(State?).

search_loop(State?) : -
   search_step(State?, NewState!),
   search_loop(NewState?).
```

Notice that:
- *goal* checks whether there is a solution in *State*,
- *display_solutions* displays all solutions in *State*.

The predicates *goal* and *display_solutions* use the predicates *solution* and *display*. The predicate *solution* defines which candidates are solutions by checking whether the value of some evaluation function on the candidates is below a certain boundary. Without loss of generality we assume the better a candidate is the less its value is. Observe that a special form of the *goal* predicate is used here. We state the general format for *solution* to be:

```
solution(Candidate?) : -
   eval(Candidate?, Value!),
   Value <= boundary.
```

where
- *eval* is an user defined evaluation function on the candidates,
- boundary is a user defined constant.

The predicate *display* displays a candidate and is to be defined by the DSS-expert (domain knowledge).

```
goal([Cand | Cands]?) : -
   solution(Cand?).

goal([Cand | Cands]?) : -
   not(solution(Cand?)),
   goal(Cands?).

display_solutions([]).

display_solutions([Cand | Cands]?) : -
   not(solution(Cand?)),
   display_solutions(Cands?).
```

```
display_solutions([Cand | Cands]?) :-
    solution(Cand?),
    display(Cand?),
    display_solutions(Cands?).
```

We remark that the *goal* predicate can be expanded with a parameter valued with the current time. Herewith we enable the user to control the computation time of the search process. We suppose that after the search process is stopped because the time limit was exceeded the user decides whether to continue the search process and if so may update the current search state (e.g. delete, change or produce new candidates). In that way an interactive DSS is yielded. For the sake of convenience we omit the elaboration of this idea.

The predicate *search_step* calculates a new search state form the old one by
- selecting the first candidate in the present search state for further search,
- generating all neighbours of the selected candidate by means of the predicate *generate* to be defined by the DSS-expert (domain knowledge),
- removing all non-feasible neighbours by means of the predicate *filter*,
- finally calculating a new search state from the feasible neighbours and the old search state by means of the predicate *update*.

```
search_step([Cand | State ]?, NewState!) :-
    generate (Cand?, Neighbours!),
    filter (Neighbours?, FeasibleNeighbours!),
    update (FeasibleNeighbours?, [Cand | State]?, NewState!).
```

Note that both *generate* and *filter* may produce an empty list of candidates.

The predicate *filter* uses the user defined predicate *feasible* which defines the feasible candidates (instance knowledge). This gives the problem solving method its flexibility (see section 2).

```
filter([],[]).

filter([X | Y]?, [X | Z]!) :-
    feasible(X?),
    filter(Y?, Z!).

filter([X | Y]?, Z!) :-
    not(feasible(X?)),
    filter(Y?, Z!).
```

The predicate *update* defines the search strategy. There is a library of '*update*-predicates' all defining a search strategy of which the user may choose one (see end of this section).

In conclusion the following components are to be defined by the DSS-expert (domain knowledge):
- a neighbourhood relation of candidates: *generate*.
- a procedure for presenting a solution: *display*.

The user has the following components to control the search process:
- a procedure for making an initial set of candidates: *initialize*,
- a predicate defining the feasible candidates: *feasible*,
- an evaluation function *eval* on candidates together with a boundary,
- a search strategy (an *update* predicate).

Above made partitioning is arbitrary. By shifting more components to be defined by the DSS-expert the level of abstraction is enhanced. By shifting more components to be defined by the user flexibility is gained, but the user has to have more domain knowledge.
A so far untreated aspect of instance knowledge is background data. This data is used in the other predicates. We here state that the DSS-expert should define a format for all data and possibly provide a tool for data manipulation.

Next we define five '*update*-predicates' each describing a search strategy (cf. Pearl (1971) for the first four strategies). An *update* predicate determines which candidates from the current search state and the feasible neighbours are to be maintained in the next search state and in what order.

Depth-first search

By appending all feasible neighbours to the front end of the old search state after deleting the first candidate and taking the result as the new search state depth-first search is yielded. The predicate *append* is a system predicate putting its first argument in front of its second argument obtaining its last argument.

```
update(FeasibleNeighbours?, [Cand | State]?, NewState!) :-
   append(FeasibleNeighbours?, State?, NewState!).
```

Observe that as the list of feasible neighbours can be empty backtracking is enabled.

Breadth-first search

By appending all feasible neighbours to the end of the old search state after deleting the first candidate and taking the result as the new search state breadth-first search is yielded.

```
update(FeasibleNeighbours?, [Cand | State]?, NewState!) :-
   append(State?, FeasibleNeighbours?, NewState!).
```

Best-first search

By first appending all feasible neighbours to the old search state after deleting the first candidate and then sorting all candidates in the resulting search state best-first search is yielded. Sorting is done according to the *eval* predicate: candidates with the lowest

'*eval*-values' (the best candidates) are placed at the beginning of the search state.

```
update(FeasibleNeighbours?, [Cand | State]?, NewState!) :-
   append(FeasibleNeighbours?, State?, TempState!),
   sort(TempState?, NewState!).
```

Hill climbing

By first sorting all feasible neighbours and then appending the sorted neighbours to the front end of the old search state after deleting the first candidate hill climbing is yielded. Here sorting is also done according to the *eval* predicate (see best-first search).

```
update(FeasibleNeighbours?, [Cand | State]?, NewState!) :-
   sort(FeasibleNeighbours?, SortedNeighbours!),
   append(SortedNeighbours?, State?, NewState!).
```

Simulated Annealing

The following definition of *update* yields simulated annealing cf. Aarts and Korst (1989). Simulated annealing is a randomized search strategy where the search state always contains exactly one candidate. Randomly a neighbour is generated. Let *valn* be the value of this neighbour and *valc* the value of the original candidate. Whenever the neighbour is not worse (*valn* ≤ *valc*) it is accepted as the next candidate with which the search is continued. If the neighbour is worse (*valn* > *valc*) it is accepted with chance

$$\exp\left(\frac{valc - valn}{c}\right)$$

where $c \in \mathbb{R}^+$ is a constant. If the neighbour is not accepted the process continues by again randomly generating a neighbour etc.

Here the *update* predicate is an invokement of the predicate *sima*.

```
update(FeasibleNeighbours?, [Cand]?, [NewCand]!) :-
   sima(FeasibleNeighbours?, Cand?, seed?, NewCand!).
```

where *seed* is a randomly obtained integer (we assume the availability of a random number generator) used for the randomization of simulated annealing.
The predicate *sima* uses the predicates:
- *length(A,L)* where *L* becomes the length of list *A* (a system predicate),
- *take(P,L,E)* where *E* becomes the element in list *L* on place *P*.

Observe that the in *sima* used functions *exp* and / are yet to be defined (as predicates !). This assumes the availability of real arithmetic. There are Prolog versions with real arithmetic. If one does not have such a version, real arithmetic can be simulated by integers. We remark that a, b, c en m are constants.

```
sima(FeasibleNeighbours?, Cand?, Seed?, New!) :-
   length(FeasibleNeighbours?, L!),
   Place is ((a * Seed + b) mod L) + 1,
   take(Place, FeasibleNeighbours?, New!),
   eval(New?, Rn!),
   eval(Cand?, Rc!),
   Seed1 is (a * Seed + b) mod m,
   exp((Rc - Rn) / c) >= Seed1 / m.

sima(FeasibleNeighbours?, Cand?, Seed?, NewCand!) :-
   length(FeasibleNeighbours?, L!),
   Place is ((a * Seed + b) mod L) + 1,
   take(Place, FeasibleNeighbours?, NewTry!),
   eval(NewTry?, Rn!),
   eval(Cand?, Rc!),
   Seed1 is (a * Seed + b) mod m,
   exp((Rc - Rn) / c) < Seed1 / m,
   NewSeed is (a * Seed1 + b) mod m,
   sima(FeasibleNeighbours?, Cand?, NewSeed?, NewCand!).

take(1, [Cand | Cands]?, Cand!).

take(Place?, [Cand | Cands]?, CandRes!) :-
   Place > 1,
   PlaceMin1 is Place - 1,
   take(PlaceMin1?, Cands?, CandRes!).
```

4. Examples of domain and instance knowledge

In this section the predicates that (in section 3) were left to be defined by the DSS-expert and the user are defined for the Travelling Salesman Problem (TSP) cf. Lawler, Lenstra, Rinnooy Kan and Shmoys (1985) and Precedence Constrained Scheduling (PCS) cf. Garey and Johnson (1978).

4.1 The Travelling Salesman Problem

We start with a mathematical description of the problem type. Let
- C be a finite set of cities and $N = |C|$.
- dist : $C \times C \longrightarrow \mathbb{R}_0^+$ be a function denoting the distance between cities.

A route is a bijection r : {1,..,N} $\longrightarrow$ C : a permutation of all cities. Let R be the set of all routes.

The function length : R $\longrightarrow \mathbb{R}$ is defined for every r $\in$ R:

$$\text{length}(r) = \sum_{i=1}^{N-1} \text{dist}(r(1),r(i+1)) + \text{dist}(r(N),r(1))$$

denoting the length of a route.

A route with its length below a certain boundary is what we are looking for.

In Prolog the function *dist* is represented by a set of facts *dist(c1,c2,d)* where *d* is the distance from *c1* to *c2*.
The next question is the representation of routes in Prolog. We decide to represent a route by a list.
The neighbourhood of a route is chosen to be all routes generated by switching two adjacent cities (with exception of switching the first and the last city). Remark that we assume a route to be given. This implies that initially at least one route must be present in the search state. The predicate *generate* uses the predicate *hgenerate*. The latter has two input arguments:
- the first is a begin part of the original route,
- the second is a end part of the original route.

It holds that by appending the begin part at the front end of the end part the original route is yielded. A neighbour is produced by switching the first two cities of the end part and then appending the begin part at the front end of the thus generated list. Furthermore the first city of the end part is appended at the end of the begin part yielding the new begin part. The same city is then deleted from the end part.

```
generate(Route?, Neighbours!) :-
   hgenerate(Route?,[],Neighbours!).

hgenerate(_,[City]?,[]).

hgenerate(BeginPart?, [City1, City2 | EndPart]?, [Neighbour | Res]!) :-
   append(BeginPart?, [City2, City1 | EndPart]?, Neighbour!),
   append(BeginPart?, [City1]?, BeginPartNew!),
   hgenerate(BeginPartNew?, [City2 | EndPart]?, Res!).
```

We remark that this is not the only possible neighbourhood or even the best possible. Another example of a TSP neighbourhood is one containing all 2-changes of cities and not just the 2-changes of adjacent cities.

After this 'expert part' (domain knowledge) the components to be defined by the user are treated. We evaluate a route by its length (the predicate *last* picks the last element of a list):

```
eval([First | Tail]?, Value!) :-
   last(Tail?, Last!),
   dist(Last?, First?, X!),
   heval([First | Tail]?, Dist),
   Value is Dist + X.
```

```
heval([City]?, 0!).

heval([City1, City2 | Tail]?, Dist!) :-
    heval([City2 | Tail]?, DistRest!).
    dist(City1?, City2?, X!),
    Dist is DistRest + X,
```

Any *initialize* predicate yielding a route will do.
The user can state that all routes are feasible. This would result in the following definition of *feasible*:

```
feasible(Route?).
```

Of course any of the treated search strategies can be chosen.

4.2 Precedence Constrained Scheduling

Again we start with a mathematical description. Let
- M be a finite set of machines,
- J be a finite set of jobs,
- able $\subseteq M \times J$ such that $(m,j) \in$ able means machine m can perform job j,
- pre $\subseteq J \times J$ such that $(j',j) \in$ pre denotes that j' should be completed before j is started (j' is a predecessor of j),
- dur : able $\longrightarrow \mathbb{R}_0^+$ such that dur(m,j) denotes the amount of time needed by machine m to perform job j.

In a schedule jobs are attributed by machines and beginning times, that is a schedule is a pair (m,b) of partial functions, where
- dom(b) = dom(m),
- $m : J \rightharpoonup M$ is such that $\forall j \in \mathrm{dom}(m) : (m(j),j) \in$ able,
- $b : J \rightharpoonup \mathbb{R}_0^+$ is such that
  - all jobs have their predecessors ready:
    $\forall j \in \mathrm{dom}(b)\ \forall j' \in J : (j',j) \in \mathrm{pre} \Rightarrow j' \in \mathrm{dom}(b) \wedge b(j') + \mathrm{dur}(m(j'),j') \leq b(j)$,
  - the processing of two jobs on the same machine can not overlap:
    $\forall j,j' \in \mathrm{dom}(b) :$
    $m(j) = m(j') \wedge j \neq j' \Rightarrow b(j) \geq b(j') + \mathrm{dur}(m(j'),j') \vee b(j') \geq b(j) + \mathrm{dur}(m(j),j)$

Let S be the set of all schedules.

The function $c : S \longrightarrow \mathbb{R}_0^+$ is defined for every $(m,b) \in S$:

$$c((m,b)) = \max \{\ b(j) + \mathrm{dur}(m(j),j) \mid j \in \mathrm{dom}(m)\ \},$$

denoting the completion time of a schedule.

A complete schedule is a schedule (m, b) ∈ S where all jobs are scheduled, i.e. dom(m) = dom(b) = J. A complete schedule with its completion time under a certain boundary is what we are looking for.

In Prolog the set of jobs is represented by the fact *jobs(joblist)* where *joblist* is a list of all jobs. The set of machines is represented by the fact *machines(machinelist)* where *machinelist* is a list of all machines. The set *able* is represented by a set of facts *able(job, machinelist)* where *machinelist* is a list of all machines on which *job* can be performed. The set *pre* is represented by a set of facts *pre(job, preds)* where *preds* is a list of all predecessors of *job*. The function *dur* is represented by a set of facts *dur(mach, job, d)* where *d* is the duration of *job* on machine *mach*. All this is background knowledge.
We represent schedules by a list of operations. An operation is a term *operation(job, mach, bt, ct)* where *mach* is the machine on which *job* is performed, *bt* is the beginning time and *ct* the completion time of *job*.
We state a job on a machine only to be added to a schedule with its beginning time equal to the maximum of:
- the maximum completion time of any job on the same machine,
- the maximum completion time of any predecessor of the job.

The neighbourhood of a schedule is chosen to be those schedules generated by extending the schedule with one job on a machine that has the earliest possible beginning time among the beginning times of all job-machine combinations. Note that if more jobs can be started at the same time more neighbours will be generated. To generate all neighbours we first determine which unscheduled jobs have all predecessors scheduled. Next we determine which operations have the earliest beginning time. After that we construct the neighbours by appending all found operations to the original schedule each yielding a new schedule. All this is reflected by the following definition of *generate*:

```
generate(Schedule?, Neighbours!) :-
    all_preds_sched(Schedule?, Jobs!),
    earliest_operations(Schedule?, Jobs?, OperList!),
    make_neigh(Schedule?, OperList?, Neighbours!).
```

where
1) *all_preds_sched* determines which unscheduled jobs have all predecessors scheduled,
2) *earliest_operations* determines operations for all above found jobs who have the earliest possible beginning time,
3) *make_neigh* constructs the neighbours by appending all found operations to the original schedule each yielding a new schedule.

ad 1)

The predicate *all_preds_sched* uses the predicate *hall_preds_sched(S,J,N)* where *N* becomes a list of unscheduled jobs from job list *J* that have all predecessors scheduled

in schedule *S* (see Appendix for the definition of *hall_preds_sched*).

```
all_preds_sched(Schedule?, NoPreds!) : -
   jobs(AllJobs),
   hall_preds_sched(Schedule?, AllJobs?, NoPreds!).
```

ad 2)

The predicate *earliest_operations* uses the predicates
- *earliest_machs(S,J,M,O)* where *O* becomes a list of operations denoting on which machines from machine list *M* job *J* can be started as early as possible given schedule *S*.
- *new_operationlist(O1,O2,O3)* where *O3* becomes a list of the earliest possible operations in *O1* and *O2* (all operations in *O1* have the same beginning time and so do all operations in *O2*) (see Appendix for the definition of *new_operationlist*).

```
earliest_operations(Schedule?, [Job]?, OperList!) : -
   able(Job?, Machs!),
   earliest_machs(Schedule?, Job?, Machs?, OperList!).

earliest_operations(Schedule?, [Job | Jobs]?, OperList!) : -
   earliest_operations(Schedule?, Jobs?, OperList1!),
   earliest_operations(Schedule?, [Job]?, OperList2!),
   new_operationlist(OperList1?, OperList2?, OperList!).
```

The predicate *earliest_machs* uses the predicate *make_operation(S,J,M,O)* where *O* becomes a list of one operation *(J,M,BT,CT)* with
- *BT* the earliest possible beginning time after the last job on machine *M* and after the last predecessor of job *J*,
- *CT* is the completion time of job *J* (*CT* = *BT* + *dur(M,J)*).

```
earliest_machs(Schedule?, Job?, [], []).

earliest_machs(Schedule?, Job?, [Mach | Machs]?, OperList!) : -
   earliest_machs(Schedule?, Job?, Machs?, OperList1!),
   make_operation(Schedule?, Job?, Mach?, OperList2!),
   new_operationlist(OperList1?, OperList2?, OperList!).
```

The predicate *make_operation* uses the predicates
- *comptime_mach(S,M,T)* where *T* becomes the maximum completion time of any job on machine *M* in schedule *S* (see Appendix for the definition of *comptime_mach*),
- *comptime_preds(S,J,T)* where *T* becomes the maximum completion time of any predecessor of job *J* in schedule *S* (see Appendix for the definition of *comptime_preds*),
- *max(A,B,C)* where *C* becomes the maximum of *A* and *B* (not elaborated).

```
make_operation(Schedule?, Job?, Mach?, Oper!) : -
   comptime_mach(Schedule?, Mach?, Time1!),
   comptime_preds(Schedule?, Job?, Time2!),
   max(Time1?, Time2?, Begintime!),
   dur(Mach?, Job?, Dur!),
   Comptime is Begintime + Dur,
   Oper = [operation(Job, Mach, Begintime, Comptime)].
```

ad 3)

The predicate *make_neigh* is rather straightforward.

```
make_neigh(Schedule?, [], []).

make_neigh(Schedule?, [Oper | OperList]?, [[Oper | Schedule] | Neighbours]!) : -
   make_neigh(Schedule?, OperList?, Neighbours!).
```

As mentioned before a complete schedule with its completion time under a certain boundary is what we are looking for. We evaluate a schedule by its completion time plus a penalty. The penalty of a schedule is the sum of the maximum duration of all unscheduled jobs. Observe the penalty for a complete schedule to be equal to zero, so complete schedules are evaluated strictly by their completion time.

```
eval(Schedule?, Value!) : -
   penalty(Schedule?, Pen!),
   comptime(Schedule?, CT!),
   Value is Pen + CT.
```

The predicate *comptime(S,T)* calculates the completion time *T* of a schedule *S* (i.e. the maximum completion time of any scheduled job).

```
comptime([], 0).

comptime([operation(_,_,_,Time1) | OperList]?, Value!) : -
   comptime(OperList?, Time2!),
   max(Time1?, Time2?, Value!).
```

The penalty of a schedule is calculated by first determining which jobs are unscheduled. This is done by deleting the scheduled jobs from a list of all jobs. From the unscheduled jobs the sum of all maximum durations is calculated obtaining the penalty. The predicate *penalty(S,P)* uses the predicates

- *schjobs(S,SJ)* where *SJ* becomes a list of all scheduled jobs in schedule *S*.
- *unschjobs(AJ,SJ,UJ)* where *UJ* becomes a list of all jobs in job list *AJ* that are not in job list *SJ*.
- *totaldur(UJ,P)* where *P* becomes the sum of all the maximum durations of the jobs in *UJ*.

(see Appendix for the definition of *schjobs*, *unschjobs* and *totaldur*)

```
penalty(Schedule?, Pen!) :-
    schjobs(Schedule?, SchedJobs!),
    jobs(Alljobs!),
    unschjobs(Alljobs?, SchedJobs?, UnSchedJobs!),
    totaldur(UnSchedJobs?, Pen!).
```

We choose the following definition of the *initialize* predicate yielding a list containing just the empty schedule:

```
initialize([[]]).
```

Of course any of the treated search strategies can be chosen.

5. The Shell

The shell is in fact a tool box to construct domain specific DSS's. Up to now we sketched only a small part of the tool kit. For instance we did not pay attention to the user interface of a DSS, which might take a lot of lines of code. The tool kit has the following components, besides a Prolog interpreter:

- **generic knowledge base**: containing the predicate definitions of section 3.
- **domain knowledge definition facility**: to define predicates like generate, display, eval and feasible. Furthermore this facility enables the expert to define the predicate names of the facts of the instance knowledge to be defined by the user. In addition to these predicate names the expert should define the arity of these predicates, the types of the parameters and possibly some predicates checking the consistency of the facts.
- **instance data entry facility**: to enter facts, using the above mentioned names, arities, types and consistency checks.
- **control facility**: to compose a specific search method by selection of a search strategy, an evaluation predicate and a feasibility predicate. Furthermore this facility should allow the user to change parameters during the search process. It depends on the expertise of the user if he is able to define evaluation or feasibility predicates. An unexperienced user should only select these predicates from the domain knowledge base. The predicate names of facts are considered as domain knowledge here. In case of the TSP only *dist* was such a predicate. In the case of PCS *jobs*, *machines*, *able*, *pre* and *dur* were such predicates. It is clear that only facts of the form *able(j, [m1,m2])* are allowed if *j* is a job and *m1* and *m2* are machines. Hence here a consistency check to guarantee referential integrity is required. In fact the shell should have all standard database management functions.

Appendix

In this appendix some in section 4 used predicates are elaborated.

The predicate *hall_preds_sched(S,J,N)* calculates a list *N* of unscheduled jobs from job list *J* that have all predecessors scheduled in schedule *S*. In the definition of *hall_preds_sched* the following predicates are used:
- *member(X,Y)* a system predicate checking whether *X* is an element of *Y*,
- *allmember(X,Y)* checking whether *X* is a subset of *Y*.

```
hall_preds_sched(Schedule?, [], []).

hall_preds_sched(Schedule?, [Job | Jobs]?, NoPreds!) :-
   member(operation(Job,_,_,_), Schedule?),
   hall_preds_sched(Schedule?, Jobs?, NoPreds!).

hall_preds_sched(Schedule?, [Job | Jobs]?, [Job | NoPreds]!) :-
   not (member(operation(Job,_,_,_), Schedule?)),
   pre(Job?, Preds!),
   allmember(Preds?, Schedule?),
   hall_preds_sched(Schedule?, Jobs?, NoPreds!).

hall_preds_sched(Schedule?, [Job | Jobs]?, NoPreds!) :-
   not (member(operation(Job,_,_,_), Schedule?)),
   pre(Job?, Preds!),
   not (allmember(Preds?, Schedule?)),
   hall_preds_sched(Schedule?, Jobs?, NoPreds!).

allmember([], Schedule?).

allmember([Pred | Preds], Schedule?) :-
   member(operation(Pred,_,_,_), Schedule?),
   allmember(Preds?, Schedule?).
```

The predicate *new_operationlist(O1,O2,O3)* calculates which operations in operation lists *O1* and *O2* have the earliest beginning times resulting in *O3* and uses the predicate *time(O,T)* where *T* becomes the beginning time of the first operation in *O* (*O* not empty).

```
new_operationlist([], OperList?, OperList!).

new_operationlist(OperList?, [], OperList!).

new_operationlist(OperList1?, OperList2?, OperList1!) :-
   time(OperList1?, Time1!),
   time(OperList2?, Time2!),
   Time1 < Time2.
```

```
new_operationlist(OperList1?, OperList2?, OperList2!) :-
   time(OperList1?, Time1!),
   time(OperList2?, Time2!),
   Time1 > Time2.

new_operationlist(OperList1?, OperList2?, OperList!) :-
   time(OperList1?, Time1!),
   time(OperList2?, Time2!),
   Time1 = Time2,
   append(OperList1?, OperList2?, OperList!).

time([operation(_,_,T,_) | OperList], T).
```

The predicate *comptime_mach(S,M,T)* calculates the maximum completion time *T* of any job on machine *M* in schedule *S*.

```
comptime_mach([], Mach?, 0).

comptime_mach([operation(_,Mach1,_,_) | OperList]?, Mach2?, Time!) :-
   not (Mach1 = Mach2),
   comptime_mach(OperList?, Mach?, Time!).

comptime_mach([operation(_,Mach,_,Time1) | OperList]?, Mach?, Time!) :-
   comptime_mach(OperList?, Mach?, Time2!),
   max(Time1?, Time2?, Time!).
```

The predicate *comptime_preds(S,J,T)* calculates the maximum completion time *T* of any predecessor of job *J* in schedule *S*.

```
comptime_preds(Schedule?, Job?, Time!) :-
   pre(Job?, Preds!),
   hcomptime_preds(Schedule?, Preds?, Time!).

hcomptime_preds(Schedule?, [Pred]?, Time!) :-
   member(operation(Pred,_,_,Time), Schedule?).

hcomptime_preds(Schedule?, [Pred | Preds]?, Time!) :-
   hcomptime_preds(Schedule?, Preds?, Time1!),
   member(operation(Pred,_,_,Time2), Schedule?),
   max(Time1?, Time2?, Time!).
```

The predicate *schjobs(S,J)* calculates a list *J* of all scheduled jobs in schedule *S*.

```
schjobs([],[]).

schjobs([operation(Job,_,_,_) | Y]?, [Job? | X!]) :-
   schjobs(Y?, X!).
```

The predicate *unschjobs(AJ,SJ,UJ)* calculates a list *UJ* of unscheduled jobs from the list *AJ* of all jobs and a list of *SJ* of scheduled jobs.

```
unschjobs([], Schjobs?, [])

unschjobs([X | Y]?, Schjobs?, UnSchedJobs!) :-
   member(X?, Schjobs?),
   unschjobs(Y?, Schjobs?, UnSchedJobs!).

unschjobs([X | Y]?, Schjobs?, [X | UnSchedJobs]!) :-
   not(member(X?, Schjobs?)),
   unschjobs(Y?, Schjobs?, UnSchedJobs!).
```

The predicate *totaldur(JL,P)* calculates the sum *P* of all maximum durations of all jobs in job list *J* and uses the predicate *maxdur(J,T)* where *T* becomes the maximum duration of job *J* on any machine.

```
totaldur([], 0).

totaldur([Job | Jobs]?, Pen!) :-
   maxdur(Job?, M!),
   totaldur(Jobs?, Pen1!),
   Pen is M + Pen1.
```

The predicate *maxdur(J,M)* calculates the maximum duration *M* of job *J* and uses the predicates

- *durlist(J,ML,DL)* where *DL* becomes a list of the durations of job *J* on all machines in machinelist *M*,
- *maxlist(DL,M)* where *M* becomes the maximum in list *DL*.

```
maxdur(Job?, M!) :-
   able(Job?,MachList!),
   durlist(Job?,MachList?,DurList!),
   maxlist(DurList?, M!).

durlist(Job?, [], []).

durlist(Job?, [Mach | MachList]?, [Dur | DurList]!) :-
   dur(Mach?, Job?, Dur!),
   durlist(Job?, MachList?, DurList!).

maxlist([], 0).

maxlist([X | Y]?, Z!) :-
   maxlist(Y?, R!),
   max(X?, R?, Z!).
```

References

Aarts, E.H.L. and Korst, J., Simulated annealing and Boltzmann machines: a stochastic approach to combinatorial optimization and neural computing, Wiley, 1989.

Checkland, P., Systems Thinking and Systems Practice, Wiley, 1981.

Eiben, A.E. and van Hee, K.M.: Knowledge Representation and Search Methods for Decision Support Systems, in: Gaul, W. and Schader, M., Ed., Data, Expert Knowledge and Decisions, Springer-Verlag, 1990.

Garey, M.R. and Johnson, D.S., Computers and Intractability: A Guide to the Theory of NP-completeness, Freeman and Co, 1978.

van Hee, K.M. and Lapinski, A., OR and AI approaches to decision support, Decision Support Systems 4 (1989), pp 447-459.

Keen, P.G.W. and Scott Morton, M.S., Decision support systems: an organized perspective, Addison-Wesley, 1978.

Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G. and Shmoys, D.B., The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization, Wiley, 1985.

Papadimitriou, C.H. and Stieglitz, K., Combinatorial optimization: algorithms and complexity, Prentice-Hall, 1982.

Pearl, J.R., Artificial intelligence: the heuristic programming approach, McGraw-Hill, 1971.

Schnupp, P. and Bernhard L.W., Productive Prolog Programming, Prentice Hall, 1986.

Spivey, M., The Z notation, Prentice Hall, 1989.