SSST (State-Space Search Tool): A Probabilistic Forward-Chained Expert System Shell with Full Backtracking

Zeki O. Bayram

Computer Engineering Department
Boğaziçi University
Bebek 80815/Istanbul-Turkey
internet: bayram@boun.edu.tr

Abstract

We describe a probabilistic backtracking forward-chained expert system shell that performs a best-first search of the state-space consisting of working memory states. The state space that needs to be traversed can be narrowed significantly through tactful use of the **context mechanism**. **Fail conditions** detect forbidden working memory states and cause immediate backtracking. Heuristic information about which rules should have higher priority are encoded in the rules at the granularity level of **condition elements** in the form of **importance factors**. Facts in the working memory have associated with them **confidence factors,** which allows the representation of uncertain information. The paradigm that results as the combination of this specific set of features permits declarative specification of the state space required for the solution of many kinds of scheduling problems and other kinds of problems requiring intelligent search of the state space with possible backtracking.

Keywords: Expert system shell, forward chaining, backtracking, context, heuristic, intelligent search, inference

1. Introduction

Forward chaining inference engines start with an **initial state** of the world (as described by the initial contents of the *working memory*), and seek to reach a **goal state** through repeated application of a certain set of transformations, usually specified in the form of if-then rules [[5],[7]]. The deficiency of most current forward chaining expert system shells is that they allow a **single line of reasoning** (also called **hill climbing**). If, upon reaching a certain state the system cannot proceed any further (possibly because a wrong choice was made earlier in the selection of which rule to fire), the system stops without finding a solution. The main reason for this deficiency is efficiency: saving the *choice points* at every iteration certainly consumes a lot of space.

State Space Search Tool (SSST) is an expert system shell with an inference engine that does allow backtracking. In SSST the efficiency problem is dealt with in three ways: (1) by making the search space small through a **context mechanism**, (2) by careful

pruning of branches in the search tree that cannot possibly lead to a solution using **fail conditions**, and (3) by guiding the search through heuristic information encoded in the rules at a very fine level of granularity through **importance factors**. Fail conditions are predicates on the working memory which describe forbidden working memory states. If a fail condition is satisfied (made true) by the current working memory, immediate backtracking is initiated to explore other branches of the search space. As such, fail conditions help to specify constraints on the working memory states. Through clever use of these three mechanisms, the search space can be cut down dramatically and full backtracking becomes practical.

SSST programs are also declarative in the sense that **(production) rules** describe relationships that may exist among working memory states, **fail conditions** describe forbidden memory states, and **success conditions** (called end_goal in SSST syntax) describe **final** working memory states beyond which no further inferencing needs to be made, since the working memory already contains a solution. After the initial working memory is given, all the rest is up the system.

SSST is not complete in a theoretical sense, in that a line of reasoning may be infinite, with no solutions on the path, while at the same time a solution may exist on another unexplored branch. However, a theoretically complete system requires a full breadth-first search of the state space which is an impractical proposition in terms of time efficiency. It is by a similar efficiency consideration that the Prolog language [[6]] also uses a depth-first strategy, even though it is incomplete for SLD resolution, the operational semantics of Prolog.

In the remainder of this paper we describe the knowledge representation, execution model of SSST and give one (due to lack of space) example of its use. We then compare SSST with some other expert system shells in its class and conclude with a summary and further research directions.

2. Knowledge Representation in SSST

2.1 Working Memory Elements

Knowledge is represented in the form of facts in the working memory. Before a fact can be added to the working memory, its template must be made known to the system through the **literalize** command. Suppose we wish to place information about a car into the working memory. Assume that a car has the attributes **owner**, **color** and **age**. First, the command

```
?- literalize(car(owner,color,age)).
```

is given to the system. This needs to be done only once. Then, to place information about a car into working memory, the following command is given.

```
?- make( car(0.5, owner george, color green, age 5)).
```

This means that we are 50% confident that there is a car whose color is green, age is 5 and is owned by george. It is possible to leave some attributes unspecified: they are taken to be **nil**. If the working memory does not contain an entry for a given fact, its confidence factor is taken to be -1.0.

2.2 Production Memory

The production memory consists of a set of productions. Each production is an **if-then rule** of the form

```
rule( rule_name,
      list_of_contexts_in_which_the_rule_is_active,
      condition_element1 and
      condition_element2 and
          ........
      condition_elementn
       -->
      right_hand_side_action1 and
      right_hand_side_action2 and
          .........
      right_hand_side_actionm ).
```

2.2.1 Left-Hand-Side Conditions

Each condition element (CE) is one of

- An **SSST goal** which is meant to match a fact in the working memory. The template of the goal should have been by a literalize command. An example of an SSST goal is: `car(1,owner mary).` The first argument to `car` is the **importance factor**. Such a goal is true if it has an importance factor IF, it matches a working memory element with confidence factor CF, and IF*CF>0. An SSST goal usually has logical variables in them. For example, in `car(1,owner X, color red)`, X is a logical variable. Logical variables are bound to terms in the working memory through simple **first order unification**. If the variable occurs anywhere else in the rule, either on the left hand side or right hand side, the value it is originally bound to is substituted for all the other occurrences. Logical variables always start with a capital letter, as in Prolog systems.
- `evaluate(importance_factor, goal)` where `goal` is a goal to be satisfied by the underlying CLP(R) interpreter. This kind of CE evaluates to true if the goal succeeds and the `importance_factor` > 0, or if it fails and `importance_factor` < 0.
- `one_true(importance_factor,` $\texttt{condition_element}_1$ `and` $\texttt{condition_element}_2$ `and ...).` This is like the **or** operator. It is true only if at least one of its components is true and its importance_factor > 0, or if none of its components is true and its importance_factor < 0. Note that the syntactic construct "and" used here is nothing more than a connector, and does not have the "logical and" meaning.

- `all_true( importance_factor, condition_element`$_1$ `and condition_element`$_2$ `and ...)`. This is the **and** operator. It is true if all its components are true and its importance_factor > 0, or not all of its components are true and its importance_factor < 0. Again note that "and" is used only as a connector.
- `not_true(importance_factor, condition_element`$_1$ `and condition_element`$_2$ `and ...)`. This is the **not** operator. It is false if all its components are true and its importance_factor > 0, or there is no condition under which all its components can be true, but its importance_factor is negative. It is true otherwise. The "and" here can really be seen as the "logical and".

2.2.2 Right-Hand-Side Actions

The following is the list of right hand side actions.

- `remove(index)` where index is the position on the left hand side of the CE to be removed. It causes the removal of the fact which matched the CE from the working memory.
- `make(CF, w_m_e)`. Causes the addition to the working memory element `w_m_e` to the working memory with confidence factor CF.
- `modify(index, CF, attribute_name`$_1$ `attribute_value`$_1$`, attribute_name`$_2$ `attribute_value`$_2$`,....)`. Causes the modification of the working memory element matched by the CE on the left whose position is *index* on the left hand side.
- `evaluate(goal)` where `goal` is a goal to be satisfied by the underlying CLP(R) interpreter.
- `add_context(a_context_name)` which adds `a_context_name` to the current list of active contexts
- `remove_context(a_context_name)` which removes `a_context_name` from the current list of active contexts.

Right hand side actions are usually grouped together using the "and" syntactic construct, which again has no relation to the "logical and".

Note that the language of SSST is very small in terms of the primitives. This is because all facilities of the underlying CLP(R) interpreter are available to be called upon both on the left hand side and right hand side of the rules (for example, input, output operations etc.).

2.3 Initializing the Working Memory and Active Contexts

Before facts can be added to the working memory, their templates must be declared to the system through the literalize command. Also, before contexts can be added to the active contexts list, they must be declared by the "context" command. As an example of a template declaration, we may have:

```
?- literalize(pref(student,course)).
```

This declares the predicate "pref" to have two attributes, "student" and "course". Note the "?-" which means this is treated as a CLP(R) goal to be evaluated when the input file is being read. Next, in order to place an initial fact into the working memory, we use the **make** command, as shown below:

```
?- make( pref(1.0, student sema, course cmpe150)).
?- make( pref(0.9, student sema, course cmpe420)).
?- make( pref(0.8, student sema, course cmpe220)).
```

The above code is part of a full example given later on. Here, we are declaring that student "sema" has a preference for course "cmpe150" with a desire as strong as "1.0", has a preference for course "cmpe420" with a desire as strong as "0.9", and has a preference for course "cmpe220" with a desire as strong as "0.8". The conflict resolution mechanism of SSST will guarantee that if there is a rule which matches all three facts, the instantiation obtained using the first fact will have higher precedence over the instantiation obtained using the second fact, and similarly for second and third facts. Again notice the use of "?-".

The following code shows how to declare contexts and initialize the active context list.

```
context(left).
context(right).
?- add_context(left).
```

2.4 Success Conditions

A **success condition** is a predicate on the working memory elements. If the current working memory satisfies a success condition, the inference engine stops, prints the contents of the working memory, asks for whether another answer is required, and if the answer is positive, it backtracks. Otherwise, no further inferencing is done. Syntactically, an end condition is similar to a left hand side condition and is written as **end_goal(name, context_list, a_condition)**. An example of an end condition is given below.

```
end_goal( e1, [right],
        id(1, cannibals_left 0, missionaries_left 0, boat right)).
```

2.5 Fail Conditions

A fail condition is like a consistency check on the working memory. It says that a working memory under which the condition holds true is invalid. A fail condition, if satisfied by the working memory, causes immediate backtracking. Syntactically, it is written as **fail_condition(name, context_list, a_condition)**. An example of a fail condition is given below:

```
fail_condition( f1, [left,right],
                id(1, cannibals_left C, missionaries_left M) and
                    evaluate(1,C>M) and
                    evaluate(1,M>0)).
```

3. Execution Model of SSST

3.1 Inference Algorithm

In Figure 1 we give the execution model of SSST. This model performs a full search of the state-space, **backtracking** if necessary to find a solution. A choice point for backtracking occurs where a rule selection is made from the conflict set for application to the current working memory and the conflict set contains other rules that apply to the same working memory. Rule selection is done according to a measure (to be described shortly) of how *well* each rule in the conflict set matches the current working memory. First, the rule with highest score is selected, and upon backtracking, other rules in the conflict set are selected, in decreasing order of their score.

```
function  INFER ( CS1,WM1,CTX1) : boolean
Begin
  For each rule R in CS1 do
  begin
     (WM2,CTX2) :=  apply(WM1,R,CTX1);
      if WM2 satisfies any end condition, print WM2 and return(TRUE);
      if WM2 satisfies any fail condition then
                continue with the next rule in CS1;
     CS2 := match (WM2, CTX2);
      if  INFER (CS2, WM2, CTX2) then return(TRUE);
  end_for;
  return(FALSE);
End;

Begin main
          Working_memory := initialize_with_facts();
          Context := Form_initial_set_of_contexts();
          Conflict_set :=  match(Working_memory, Context);
          return INFER( Conflict_set, Working_memory, Context);
End main;
```

Figure 1: Abstract Model of Execution of SSST

Success conditions permit the description of the conditions under which computation may stop, and **fail conditions** specify unallowed working memory states, causing immediate backtracking. The algorithm is recursive, with the **working memory**, **conflict set** and **set of active contexts** being arguments to the INFER function.

The **context mechanism** allows the constraining of the state space, resulting in more efficient execution (both in terms of speed and space), since a rule is applicable only if at least member of its context list is active at the time of the match operation. **apply(...)** takes as arguments a working memory, a rule and a context list and returns a pair: an updated working memory and an updated context list. This function applies its rule argument to its working memory argument, and updates the working memory and context

list according to the actions specified in the rule. **match(...)** matches only the rules that are active in the current context list against the current working memory, returning a new conflict set of instantiated rules.

For the sake of simplicity, the algorithm has been presented at a very high level of abstraction, and without the techniques used to avoid iteration over the rule base and working memory inside the **match(...)** function.

3.2 Conflict Resolution

The **match** phase in forward chaining causes a confidence factor to be associated with each condition element (CE) on the left hand side of a rule (which is the confidence factor of the working memory element that unified with the condition element). Since each CE already had an importance factor (IF) associated with it, we can then compute, for each rule matched against the working memory, a score, denoting how well the rule matched the working memory. That score is given by $\sum_i CF_i * IF_i$, where i is the index of each condition element (its relative position on the left hand side). An instantiated rule is placed in the conflict set only if **each** CE in the rule has **positive** CF*IF and if $\sum_i CF_i * IF_i$ is greater than some threshold, usually zero. The rules are then ranked in the conflict set according to their scores, those with highest scores being on top, with rules with higher scores taking precedence over rules with lower scores. Upon backtracking, all rules in the conflict are tried. This is an implementation of the "best-first" conflict resolution strategy with backtracking.

4. An Example SSST Program: Assignment of Teaching Assistants to Courses Problem

This is a problem that surfaced in the Computer Engineering Department of Bogazici University. A fixed number of teaching assistants are to be assigned to a fixed number of computer courses as assistants. Each assistant fills out a form of preferences, listing his/her preference from highest to lowest. For each course the number of teaching assistant needed is fixed. Below we give a solution to this problem in SSST. As a simplification, we assume that there are only three students, "duygu," "sema" and "cenk." Also we include only three preferences for each student. The algorithm cycles through the students, trying to give each student their first choice, then their second choice etc.

First we declare a context called "all" and make it active. In this example, only one context is used.

```
context(all).
?- add_context(all).
```

Next, we declare the templates for facts in the working memory.

```
?- literalize(pref(student,course)).
?- literalize(assigned(student,course)).
?- literalize(requires(course,no_students)).
?- literalize(current_student(student)).
?- literalize(info(student,class,remaining)).
```

We then initialize the working memory with facts regarding students and their preferences. For example, "sema" is a Ph.D. student and will assist in at most 2 courses.

```
?- make( info(1, student sema, class phd, remaining 2)).
?- make( info(1, student cenk, class masters,remaining 2)).
?- make( info(1, student duygu, class phd, remaining 2)).
```

Next we add the preferences of students. Note how "confidence factors" are used to specify preferences.

```
?- make( pref(1.0, student sema, course cmpe150)).
?- make( pref(0.9, student sema, course cmpe420)).
?- make( pref(0.8, student sema, course cmpe220)).

?- make( pref(1.0, student cenk, course cmpe420)).
?- make( pref(0.9, student cenk, course cmpe520)).
?- make( pref(0.8, student cenk, course cmpe350)).

?- make( pref(1.0, student duygu, course cmpe220)).
?- make( pref(0.9, student duygu, course cmpe520)).
?- make( pref(0.8, student duygu, course cmpe350)).
```

Then we add information regarding how many teaching assistants are required for each course. We start allocating courses to student with "sema".

```
?- make( requires(1, course cmpe150, no_students 2)).
?- make( requires(1, course cmpe220, no_students 2)).
?- make( requires(1, course cmpe350, no_students 1)).
?- make( current_student(1, student sema)).
```

We place an order in which students will be considered for allocation of courses. We shall cycle around "sema," "cenk," "duygu" and then back to "sema". This order is established using CLP(R) facts rather than SSST facts, since confidence factors are not needed here.

```
next(sema, cenk).
next(cenk,duygu).
next(duygu,sema).
```

Now we come to the rules. **r1** says that if the current student S has not been assigned its maximum number of courses, he has a preference for a course C and the course C still requires student assistants, assign the student S to the course C, remove the course preference which has been assigned, decrease by one the number of students required for the course C, decrease by one the number of courses the student can be assigned to (since one assignment has just been made), and make the next student the current one.

```
rule( r1, [all],
      current_student(1, student S)  and
      info(1, student S, remaining R) and
      evaluate(1, R>0) and
      pref(1, student S, course C) and
      requires(1, course C, no_students N) and
      evaluate(1, N>0) and
      assigned(-1, student S, course C) and
      evaluate(1, next( S, S2))
      -->
      remove(4) and
      make( assigned(1, student S, course C)) and
      modify(5, 1.0, no_students (N-1)) and
      modify(1, 1.0, student S2) and
      modify(2, 1.0, remaining (R-1)) ).
```

In rule **r1a**, if the current student has already been assigned its total maximum number of courses, and there is a student who has not been assigned its maximum number of courses, go on to the next student in hopes of finding a solution.

```
rule( r1a, [all],
      current_student(1, student S)  and
      info(1, student S, remaining 0) and
      info(1, student S3, remaining R) and
      evaluate(1, R>0) and
      evaluate(1, next( S, S2))
      -->
      modify(1, 1.0,  student S2)).
```

Rule **r1b** is similar to rule **r1a**, except we take no notice of the student's preference. Even though both this rule and rule r1 will usually match the same working memory, because of the conflict resolution strategy, rule r1 will always take precedence, and **r1b** will apply only upon backtracking, if a solution cannot be found by taking into account students desires. This is the guarantee that at least a solution will be found, except when no solution is theoretically possible, i.e. when the total number of assignments possible is less then to total number of required assignments.

```
rule( r1b, [all],
      current_student(1, student S)  and
      info(1, student S, remaining R) and
      evaluate(1, R>0) and
      requires(1, course C, no_students N) and
      evaluate(1, N>0) and
      assigned(-1, student S, course C) and
      evaluate(1, next( S, S2))
      -->
      make( assigned(1, student S, course C)) and
      modify(4, 1.0, no_students (N-1)) and
      modify(1, 1.0, student S2) and
      modify(2, 1.0, remaining (R-1)) ).
```

Finally, we have the success condition which evaluates to true when all courses have been assigned their required number of students.

```
end_goal(g1,[all],not_true(1, requires(1, course C, no_students N) and
                           evaluate(1, N>0))).
```

One solution found by SSST is given below.

```
assigned(1, student cenk, course cmpe150)
assigned(1, student sema, course cmpe220)
assigned(1, student duygu, course cmpe220)
assigned(1, student cenk, course cmpe350)
assigned(1, student sema, course cmpe150)
```

Note in this solution how the problem has been specified declaratively, the number of rules required is very small, preferences of students have been easily incorporated into rules to guide the system in conflict resolution so that students' preferences are honored as much as possible, and the presence of the end condition simplified the rules actually used during forward chaining.

5. Related Work

There is an abundance of Expert System Shells with varying features. These include object-oriented extensions to the basic paradigm of knowledge representation, context mechanisms for rule partitioning, focus of attention mechanisms including varying rule priorities at run time, two way integration with other software systems, such as programming languages and database systems, truth maintenance and graphical development tools among others. [[1],[3],[4],[5],[7]] contain detailed surveys of various currently available Expert System Shells and their features. None of these tools surveyed has the specific combination of features which allows a full search of the state-space, declarative specification of the problem in terms of rules, fail conditions and success conditions, and incorporation of heuristic information inside rules at the granularity level of condition elements.

6. Conclusions and Further Work

We presented a probabilistic forward chaining expert system shell with full backtracking. Knowledge representation is declarative in the form of if-then rules, fail conditions and success conditions. Heuristics about rule priorities are specified at the granularity of condition elements. Working memory elements have confidence factors associated with them, which permits representation of uncertain knowledge. The paradigm of computation that results from the combination of features present in SSST make it most suitable for the precise specification and efficient solution of problems requiring possible backtracking in the search of state-space for their solution. Scheduling problems are good examples of such problems.

Future work on SSST includes improving the efficiency of the interpreter through optimizations of its various components, as well as addition of object oriented functionality to replace the current system of templates and facts.

REFERENCES

[1] Kuru,S., Akin, L., Bayram, Z. Boðaziçi University, Computer Engineering Department Technical Report. *The Evaluation of Real Time AI Tools*, 1994

[2] Stylianou,C.A., Smith, D.R., Madey,G.R. *An Empirical Model For the Evaluation and Selection of Expert System Shells*, 1993

[3] Gevarter,W.B. *The Nature and Evaluation of Commercial Expert System Building Tools*. Computer Magazine, May 1987, pp 24-41

[4] Mettrey, W. *A Comparative Evaluation of Expert System Tools*, IEEE Computer, February 1991, pp. 19-31

[5] Alty, J.L., Coombs,M.J. *Expert Systems,*NCC Publications, 1984

[6] Clocksin,W.F., Mellish,C.S., *Programming in Prolog*, Springer-Verlag, 1981

[7] Hayes-Roth,F., Waterman A.D., Lenat,D.B. (editors) *Building Expert Systems,* Addison Wesley, 1983