Tabling for Planning

Neng-Fa Zhou[1], Roman Bartak[2], and Agostino Dovier[3]

[1] CUNY Brooklyn College & Graduate Center
[2] Charles University
[3] Univ. di Udine

Abstract.

1 Introduction

Given an initial state, a final state, and a set of possible actions, the planning problem is to find a plan that transforms the initial state to the final state. There has been a lot of interest in planning because planning is an inevitable task for building many intelligent systems such as intelligent robots. Different classes of planning problems have been studied and many approaches have been proposed [4, 13]. The classical planning problem has been a target problem for logic programming since its inception. The first logic programming language, PLANNER [6], was designed as "a language for proving theorems and manipulating models in a robot", and planning has been an important problem domain for Prolog [8]. Despite the amenability of Prolog to planning, there has been little success in applying Prolog to planning due to the looping and the state explosion problems. The planning problems that have been tackled by using Prolog are mostly toy problems, and Prolog is not recognized as a tool for planning.

Answer Set Programming (ASP) has had more successes than Prolog in solving planning problems [1, 9]. A typical ASP encoding of a planning problem divides the transition from the initial state to the final state into several frames. The first frame corresponds to the initial state and the last frame corresponds to the final state. Constraints are generated to describe the frames, ensuring that frame $t + 1$ is the result of applying one of the actions to frame t. The ASP encoding is translated into a SAT encoding and solved with a SAT solver. ASP, as a modeling language for planning, has arguably popularized the satisfiability approach to planning [7, 12].

Tabling [14, 21] can solve the looping problem in search for planning. Just like traditional STRIPS-based planners [2], a tabling-based planner treats a planning problem as a state-space search problem. During search, the planner tables all the states that have been encountered so that no state will be expanded more than once. For planning, the non-selective tabling-all approach is not feasible since there are normally an infinite number of plans for a problem. Mode-directed tabling [5, 20], which performs selective tabling based on user-supplied modes, can be used. Just like tabled model checkers [11], tabled planners also face the state explosion problem. Fortunately, with term-sharing techniques [10, 15, 19], a good representation for states that facilitates sharing can alleviate the problem. Despite that tabling has been successfully used to solve problems that were hard to solve with Prolog [16, 17], selective tabling has two weaknesses when applied to tabling: (1) no answer can be returned until the tabled predicate is completely evaluated; (2) it is impossible to handle resources and perform resource-bounded search.

This paper proposes new implementation techniques to deal with these weaknesses. To deal with the first weakness, this paper proposes a new table mode, called `nt`, that instructs the system not to table the argument. For planning, the inherited attribute values of a state, including the current partial plan, its cost, and the remaining resource amount, can be passed to the next state as an `nt` argument. Since the `nt` argument is not used in variant checking, two equivalent states will be treated as the same even if they have different attribute values. Thanks to the availability of the inherited attribute values, the planner can return a satisfactory plan as an exception without waiting for the completion of the tabled predicate. For resource-bounded planning, this paper proposes a resource-aware tabling system that fails a state immediately without expanding it if the same state has failed before due to a lack of resources and the new occurrence does not carry more resources than the old one.

The proposed techniques have been employed in the implementation of the `planner` module of Picat [18]. In order to use the `planner` module to solve a planning problem, users only need to specify the final state and the set of actions, and call the planner on an initial state to find a plan or a best plan. As users do not need to use exceptions or tabling directly, the `planner` module offers high-level abstraction for declarative description of planning problems. Furthermore, the early termination and resource-bounded search techniques can greatly improve the performance of tabled search for planning.

As examples, this paper also gives solutions in Picat to three of planning benchmarks used in the Fourth ASP Competition. For each problem, it gives a representation of the states to take advantage of the term-sharing technique in the system and a definition of the set of actions. It also gives the experimental results and compare them with that obtained by the cutting-edge ASP-based planner, Clasp. For all of the three benchmarks, the tabled planners of Picat significantly outperform the planners in Clasp.

The remainder of the paper is structured as follows. Section 2 surveys related approaches to planning. Section 3 gives an introduction to Picat's tabling system, which is used in the implementation of the `planner` module. Section 3.3 describes the two core predicates in the `planner` module and gives an example to illustrate the use of the module. Section 4 details the implementation of the two core `planner` predicates. Section 5 gives solutions for three example problems, and for each solution it also compares its performance with the solution in Clasp that was submitted to the ASP Competition. Section 6 concludes the paper.

2 Related Work on Planning

3 Tabling in Picat and the `planner` Module

Picat is a simple, and yet powerful, logic-based multi-paradigm programming language aimed for general-purpose applications. Picat is a rule-based language, in which predicates, functions, and actors are defined with pattern-matching rules. Picat incorporates many declarative language features for better productivity of software development, including explicit non-determinism, explicit unification, functions, list comprehensions, constraints, and tabling. Picat also provides imperative language constructs, such as assignments and loops, for programming everyday things. This section gives a brief introduction into Picat's tabling system. The readers are referred to the user's guide [18] for the details of Picat.

3.1 Tabling All

In Picat, in order to have all calls and answers of a predicate or function tabled, users just need to add the keyword `table` before the first rule.

The following gives an example of a tabled function and an example of a tabled predicate.

```
table
fib(0) = 1.
fib(1) = 1.
fib(N) = fib(N-1)+fib(N-2).

table
reach(X,Y) ?=> edge(X,Y).
reach(X,Y) => reach(X,Z),edge(Z,Y).
```

The function `fib` is defined with three function facts. For a function call `fib(N)`, if the argument `N` is equal to 0 or 1, then 1 is returned; otherwise, the function call is rewritten into the sum of `fib(N-1)` and `fib(N-2)`. When not tabled, the function call `fib(N)` takes exponential time in `N`. When tabled, however, it takes only linear time.

The `reach/2` predicate defines the reachability relation of a graph whose edges are given by the predicate `edge/2`. The first rule in the definition, where the head is connected to the body with the operator `?=>`, is *backtrackable*. This means that the rewriting of `reach(X,Y)` into `edge(X,Y)` is tentative. When execution backtracks over `edge(X,Y)`, the rewritting will be undone and the next rule will be applied to `reach(X,Y)`. The second rule, where the operator `=>` is used, is a non-backtrackable rule. When a non-backtrackable rule is applied to a call, the rewritting of the call to the body is a commitment, and will not be undone upon backtracking.

3.2 Selective Tabling with Table Modes

Users can also give table modes to instruct the system on what answers to table. Mode-directed selective tabling is especially useful for dynamic programming problems [5, 20]. In mode-directed tabling, a plus-sign (+) indicates input, a minus-sign (-) indicates output, `max` indicates that the corresponding variable should be maximized, `min` indicates that the corresponding variable should be minimized, and `nt` indicates that the corresponding argument is not tabled. Input arguments are assumed to be ground. Output arguments, including `min` and `max` arguments, are assumed to be variables. An argument with the mode `min` or `max` is called an *objective* argument. Only one argument can be an objective to be optimized. As an objective argument can be a compound value, this limit is not essential, and users can still specify multiple objective variables to be optimized. Only the last argument can have the mode `nt`. Again, as the `nt` argument can be a compound value, this limit is not essential, and users can pass multiple values to a tabled callee without having these values tabled. When a table mode declaration is provided, Picat tables only one answer for the same input arguments, which is optimal if an objective argument is specified.

The following example uses mode-directed tabling to find a shortest path.

```
table (+,+,-,min)
sp(X,Y,Path,W) ?=>
    Path = [(X,Y)],
    edge(X,Y,W).
sp(X,Y,Path,W) =>
    Path = [(X,Z)|Path1],
    edge(X,Z,W),
    sp(Z,Y,Path1,W1),
    W = W+W1.
```

The table mode `table(+,+,-,min)` is given before the first rule in the definition. The predicate `edge(X,Y,W)` specifies a weighted directed graph, where `W` is the weight of the edge between vertex `X` and vertex `Y`. The predicate `sp(X,Y,Path,W)` states that `Path` is a path from `X` to `Y` with the minimum weight `W`. Note that whenever the predicate `sp/4` is called, the first two arguments must always be instantiated. For each pair, the system stores only one path with the minimum weight. Also note that Picat uses pattern-matching rather than unification to test if a rule is applicable to a call. Any unifications that can bind variables in the call must be specified in the body of a rule.

The following program finds a shortest path among those with the minimum weight for each pair of vertices:

```
table (+,+,-,min).
sp(X,Y,Path,WL) ?=>
    Path = [(X,Y)],
    WL = (Wxy,1),
    edge(X,Y,Wxy).
```

```
sp(X,Y,Path,WL) =>
    Path = [(X,Z)|Path1],
    edge(X,Z,Wxz),
    sp(Z,Y,Path1,WL1),
    WL1 = (Wzy,Len1),
    WL = (Wxz+Wzy,Len1+1).
```

For each pair of vertices, the pair of variables `(W,Len)` is minimized, where `W` is the weight, and `Len` is the length of a path. The built-in function `compare_terms/2` is used to compare answers. Note that the order is important. If the term would be `(Len,W)`, then the program would find a shortest path, breaking a tie by selecting one with the minimum weight.

The `nt` mode is useful for passing global constant data, such as maps, to tabled calls. In a tabled search tree, the nodes can have attributes. Some of the attributes are *inherited*, whose values are inherited from the parent node, and some of the attributes are *synthesized*, whose values cannot be computed until the synthesized attributes of the children are computed. The `nt` mode is also useful for passing inherited attribute values down the search tree. In the implementation of the `planner` module, an `nt` argument is used to pass the current partial plan, its cost, and the available resource amount down the search tree.

3.3 The `planner` Module of Picat

The `planner` module of Picat provides predicates that can be used to solve planning problems. Given an initial state, a final state, and a set of possible actions, a planning problem is to find a plan that transforms the initial state to the final state. In order to use the `planner` module to solve a planning problem, users have to provide the following two global predicates:[4]

- `final(`*S*`)`: This predicate succeeds if *S* is a final state.
- `action(`*S*`,`*NextS*`,`*Action*`,`*ActionCost*`)`: This predicate encodes the set of actions of the planning problem. The state *S* can be transformed to *NextS* by performing *Action*. The cost of *Action* is *ActionCost*, which must be non-negative. If the plan's length is the only interest, then *ActionCost* should be 1.

The following two predicates constitute the core of the `planner` module.

- `plan(`*S*`,`*Limit*`,`*Plan*`,`*PlanCost*`)`: This predicate, if succeeds, binds *Plan* to a plan that can transform state *S* to a final state. *PlanCost* is the cost of *Plan*, which cannot exceed *Limit*, a given non-negative integer. This predicate searches for a plan by performing *resource-bounded* search. In the search tree, each node represents a state and carries attribute values including the remaining resource amount that can be used by future actions to transform

[4] A predicate or function symbol defined in a file is global if the file does not declare a module name. Global symbols can be accessed from anywhere.

```
import planner.

go =>
    S0=[s,s,s,s],
    best_plan(S0,Plan),
    writeln(Plan).

final([n,n,n,n]) => true.

action([F,F,G,C],S1,Action,ActionCost) ?=>
    Action=farmer_wolf,
    ActionCost=1,
    opposite(F,F1),
    S1=[F1,F1,G,C],
    not unsafe(S1).
action([F,W,F,C],S1,Action,ActionCost) ?=>
    Action=farmer_goat,
    ActionCost=1,
    opposite(F,F1),
    S1=[F1,W,F1,C],
    not unsafe(S1).
action([F,W,G,F],S1,Action,ActionCost) ?=>
    Action=farmer_cabbage,
    ActionCost=1,
    opposite(F,F1),
    S1=[F1,W,G,F1],
    not unsafe(S1).
action([F,W,G,C],S1,Action,ActionCost) =>
    Action=farmer_alone,
    ActionCost=1,
    opposite(F,F1),
    S1=[F1,W,G,C],
    not unsafe(S1).

opposite(n,Opp) => Op=s.
opposite(s,Opp) => Opp=n.

unsafe([F,W,G,_C]),W==G,F!==W => true.
unsafe([F,_W,G,C]),G==C,F!==G => true.
```

Fig. 1. A program for the Farmer's problem using `planner`.

the state to a final state. In resource-bounded search, a node is expanded only if the state is new and the resource amount is non-negative or the state has occurred before in an old node that had failed before due to a lack of resources but the current node carries more resources than the old one. The argument *PlanCost* is optional. If *PlanCost* is missing, then the cost of the plan is not returned. The argument *Limit* is also optional. If *Limit* is missing, then a large integer is used as the resource limit.

– `plan_unbounded(`*S*`,`*Limit*`,`*Plan*`,`*PlanCost*`)`: This predicate is the same as above except that it searches for a plan with depth-unbounded search. In depth-unbounded search, the resource limit is not taken into account during search. A new node is still expanded even if the resource limit has exceeded. The advantage of depth-unbounded search is that a failed state needs not to be re-explored when it occurs again. The arguments *PlanCost* and *Limit* are optional.

Other predicates in the `planner` module can easily be implemented by using these two predicates. For example, the `best_plan(S,Plan)` predicate uses the `plan/4` predicate to find the first plan. If such a plan is found, then it tries to find a better plan by imposing a stricter limit. This step is repeated until no better plan is found. If any plan is found, then the last plan found is returned as a best plan.

The program shown in Figure 1 solves the Farmer's problem by using the `planner` module. In this example, the length of the plan is the only interest, so the cost of each action is 1.[5]

4 The Implementation of the `planner` Module

This section gives the definitions of the two core predicates in the `planner` module: `plan/4`, and `plan_unbounded/4`. Several efforts are made to enhance the performance of the planner. First, the inherited attribute values, including the current partial plan, its cost, and the remaining resource amount, are passed down to a child node as an `nt` argument. Second, whenever a satisfactory plan that is within the resource limit is found, it is thrown as an exception, which will be caught by the first call to the tabled predicate. In this way, the plan can be returned without waiting for the tabled predicate to become complete. Third, for resource-bounded search, the linear tabling mechanism is tailored to it in such a way that a node is expanded only when the node has enough resources available.

[5] Pattern-matching requires all output unifications to be given in the bodies of rules. Pattern-matching facilitates full indexing of rules. For this example, the matching of the first argument of a call against the four-element list pattern is done only once.

4.1 Resource-bounded Search

Figure 2 gives the definition in Picat of the predicate `plan/4`. The following array is passed from a state to the next state:[6]

```
{Limit,IPlan,IPlanCost}
```

where `Limit` is the maximum amount of resources that can be consumed by future actions, `IPlan` is an inherited partial plan from the parent, and `IPlanCost` is the total cost of `IPlan`. Initially, `Limit` is the limit given by the user, `IPlan` is [], and `IPlanCost` is 0. The array is passed down as an `nt` argument. Therefore, it is not used in variant checking. Two calls to `plan_bounded_aux` are variants if their first arguments are variants.

The first rule of `plan_bounded_aux` throws (`(IPlan.reverse(),IPlanCost)`) as an exception if `final(S)` succeeds, where `IPlan.reverse()` returns the reversed list of `IPlan`.[7] The exception will be caught by the first call to the tabled predicate, which binds `Plan` and `PlanCost` of `plan/4`.

The second rule of `plan_bounded_aux` calls `action/4` to select an action. After that, it updates the remaining resource limit and checks the limit. If the updated limit is negative, then it triggers backtracking; otherwise, it continues with the tabled search by recursively calling `plan_bounded_aux` on the new state `S1`.

```
plan(S,Limit,Plan,PlanCost) =>
    IPlan = {Limit,[],0},
    catch(plan_bounded_aux(S,IPlan), (Plan,PlanCost), true).

table plan_bounded_aux(+,nt)
plan_bounded_aux(S,{Limit,IPlan,IPlanCost}),
    final(S)
=>
    throw((IPlan.reverse(), IPlanCost)).
plan_bounded_aux(S,{Limit,IPlan,IPlanCost}) =>
    action(S,S1,Action,ActionCost),
    Limit1 = Limit-ActionCost,
    Limit1 >= 0,
    plan_bounded_aux(S1,{Limit1,[Action|IPlan],IPlanCost+ActionCost}).
```

Fig. 2. The implementation of `plan/4`.

[6] In Picat, an array of size n is represented as a structure with the special functor {}/n.

[7] In Picat, the OOP notation $O.f(A_1, ..., A_n)$ is the same as $f(O, A_1, ..., A_n)$, unless O is an atom, in which case O must be a module qualifier for f.

The underlying tabling system is modified to treat `plan_bounded_aux/2` in a special way to perform resource-bounded search.[8] The tabling system [21] defines four basic operations, including *subgoal-lookup-registration*, *answer-lookup-registration*, *answer-consumption*, and *completion-checking*. Since no answer is generated for `plan_bounded_aux/2` and the same completion-checking operation can be used in case no plan is found, the only operation that needs a modification is subgoal-lookup-registration. The following gives the new definition of this operation for `plan_bounded_aux/2`.

Subgoal-lookup-registration: For a call to `plan_bounded_aux`, the system looks up the subgoal table to see if there is a call that has the same first argument.[9] If not, the system inserts the call into the subgoal table, memories the resource limit of the call, and continues with the normal execution of the program. Since the second argument of the call is not tabled, its slot is reused to memorize the resource limit. If the lookup finds that there already is a call in the table that has the same first argument, then the system checks if the call in the table is complete. If it is not complete, then the current call must be a looping descendent of the old call. In this case, the system fails the current call, triggering backtracking. If the call in the table is complete, meaning that it has failed before due to a lack of resources, then the system compares the resource limit of the current call with that of the one in the table. If the current limit is greater than the old one, then the system updates the limit and continues with the normal execution of the program. Otherwise, if the current call does not have more resources than the old one, then the system fails it.

For a call to `plan_bounded_aux`, the system fails it immediately if the same call, disregarding the second argument, has failed before due to a lack of resources and the current call does not carry more resources than the old one. In this way, the system can considerably reduce the search space.

4.2 Resource-unbounded Search

Figure 3 gives the definition in Picat of the predicate `plan_unbounded/4`. Just as in the definition of `plan/4`, an array of inherited attribute values is passed down as a non-tabled argument from a state to the next state.

The predicate `plan_unbounded_aux` performs *resource-unbounded* (also called *depth-unbounded*) tabled search. For this predicate, the linear tabling system [21] is used without any modification. For a call to `plan_unbounded_aux` with the first argument `S`, the system looks up the subgoal table to see if there is a call in the table that has the same first argument as `S`. If not, the system inserts the call into the table, ignoring the output and `nt` arguments. After registering the call, the system uses the rules to resolve the call. If `final(S)` succeeds, then

[8] In the real implementation, an internal name is used instead of `plan_bounded_aux` that cannot occur in users' programs.

[9] As the second argument is `nt`, it is not used in the lookup.

the system tables the answer (the empty plan with cost 0) for the call if it is not tabled yet. After that, the system backtracks to the latest previous call of `plan_unbounded_aux` that has an alternative action to be tried on the state. If there is no such a previous call available, then it means that the system has finished the current round of evaluation and decides whether or not the top-most looping call has completed with a fixed point. If the top-most looping call has completed, then that call succeeds or fails, depending on if it has generated an answer. Otherwise, if the top-most call has not completed, the system starts another round of evaluation of the call.

If `final(S)` fails, then the system selects an action, `Action`, to generate a new state `S1` and calls `plan_unbounded_aux` on `S1` with an array of updated attribute values. If this recursive call in the body succeeds with a plan, `Plan1`, for `S1`, then the plan, `[Action|Plan1]`, becomes a synthesized plan for `S`. Before having the synthesized plan tabled for `S`, the rule checks if an acceptable plan has been found for the initial state. If so, it throws the combined plan of the inherited and synthesized plans as an exception. The exception will be caught by the first call to `plan_unbounded_aux`, which returns the plan to the top-level predicate `plan_unbounded`. If the combined plan is not acceptable, then the system tables the answer if no answer has been tabled for the call or updates the existing tabled answer if the newly generated answer has a lower cost than the tabled one. After that, the system backtracks.

The predicate `plan_unbounded_aux` allows calls with negative resource amounts to be generated. The advantage of resource-unbounded search is that a failed state needs not to be re-explored when it occurs again. If a state has been found to be a failure after all possible actions have been tried on it, then this state will be tabled as a failed state. If a failed state occurs again during search, then it fails immediately without trying any of the actions. This can lead to a big saving of time for some problems.

Just as in the definition of `plan/4`, early termination would be impossible without access to the inherited attribute values. For a state, if the sum of the cost of the inherited plan from the parent and the cost of the synthesized plan for the state is less than or equal to the initial limit, then the combined plan is thrown as an exception.

5 Application Examples

5.1 Nomystery

Problem Description

Encoding

Performance The Picat solution solved all of the 30 instances used in the competition, while Clasp solved only 17 of the instances. Clasp was terminated

```
plan_unbounded(S,Limit,Plan,PlanCost) =>
    IPlan = {Limit,[],0},
    catch(plan_unbounded_aux(S,Plan,PlanCost,IPlan), (Plan,PlanCost), true).

table (+,-,min,nt)
plan_unbounded_aux(S,Plan,PlanCost,_),
    final(S)
=>
    Plan=[],
    PlanCost=0.
plan_unbounded_aux(S,Plan,PlanCost,{Limit,IPlan,IPlanCost}) =>
    action(S,S1,Action,ActionCost),
    Inherited1 = {Limit-ActionCost,[Action|IPlan],IPlanCost+ActionCost},
    plan_unbounded_aux(S1,Plan1,PlanCost1,Inherited1),
    Plan = [Action|Plan1],
    PlanCost = PlanCost1+ActionCost,
    (PlanCost =< Limit ->
        throw((IPlan.reverse()++Plan,IPlanCost+PlanCost))
    ;
        true
    ).
```

Fig. 3. The implementation of `plan_unbounded/4`.

by signals because of either out-of-memory or out-of-time for the other 13 instances. For the instances solved by both Picat and Clasp, Picat is more than 100 times faster than Clasp. For the Picat solution, the resource-bounded predicate, `plan/4`, was used. For this benchmark, `plan_unbounded/4` solved all the instances as efficiently as `plan/4`.

5.2 Sokoban

Problem Description

Encoding

Performance The Picat solution solved all of the 30 instances used in the competition. Clasp ran out of memory on 16 of the 30 instances. Some of the instances require plans of over 500 macro steps to solve and this poses a big challenge to Clasp's grounder because the converted SAT encodings are too large to store in memory. For instances solved by both Clasp and Picat, Picat is significantly faster than Clasp.

The Picat encoding is basically the same as the B-Prolog encoding presented in [16], but this comparison produced much favorable results for Picat because of the following two reasons: (1) the benchmark used in the competition allows more boxes than goal locations and therefore makes it hard to reverse the roles of boxes and goal locations (so called *reversed Sokoban*); (2) the improvement of the implementation, such as the introduction of hash-consing, enhances sharing and reduces memory consumption by Picat. For the Picat solution, the predicate `plan_unbounded/4` was used. When `plan/4` was used instead of `plan_unbounded/4` to perform resource-bounded search, Picat also solved all of the instances but was several times slower. This is because `plan_unbounded/4` tables many failed configurations and never explores the same failed configuration more than once. On the other hand, a configuration that is failed due to a lack of resources may need to be explored multiple times when resource-bounded search is used.

5.3 Ricochet Robots

Problem Description

Encoding

Performance This problem, which has a branching factor of 12, has been considered hard to solve. The Clasp solution is described in [3]. The Picat encoding, which defines the action predicate with only two rules, is much simpler than the Clasp encoding. It uses the resource-bounded predicate, `plan/4`. The resource-unbounded predicate, `plan_unbounded/4`, failed to solve any of the instances due to out-of-memory. The B-Prolog solution submitted to the competition uses a very complicated encoding that sacrifices completeness for memory efficiency. That encoding failed to prove the inconsistency of 17 of the 30 instances. Both Clasp and Picat solved all of the 30 instances, but Picat is several times faster than Clasp.

6 Conclusion

Acknowledgements

References

1. Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding planning problems in nonmonotonic logic programs. In *ECP*, pages 169–181, 1997.
2. Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208, 1971.
3. Martin Gebser, Holger Jost, Roland Kaminski, Philipp Obermeier, Orkunt Sabuncu, Torsten Schaub, and Marius Schneider. Ricochet robots: A transverse ASP benchmark. In *LPNMR*, 2013.
4. Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated Planning: Theory and Practice.* Morgan Kaufmann, 2004.
5. Hai-Feng Guo and Gopal Gupta. Simplifying dynamic programming via mode-directed tabling. *Softw., Pract. Exper.*, 38(1):75–94, 2008.

6. Carl Hewitt. Planner: A language for proving theorems in robots. In *IJCAI*, pages 295–302, 1969.
7. Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.
8. Robert Kowalski. *Logic for Problem Solving.* North Holland, Elsevier, 1979.
9. Vladimir Lifschitz. Answer set programming and plan generation. *Artif. Intell.*, 138(1-2):39–54, 2002.
10. Joo Raimundo and Ricardo Rocha. Global trie for subterms. In *Proceedings of CICLOPS*, 2011.
11. C.R. Ramakrishnan. Model checking with tabled logic programming. In *ALP News Letter*. ALP, 2002.
12. Jussi Rintanen. Planning as satisfiability: Heuristics. *Artif. Intell.*, 193:45–86, 2012.
13. Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (3rd Edition).* Prentice Hall, 2009.
14. David S. Warren. Memoing for logic programs. *Comm. of the ACM, Special Section on Logic Programming*, 35:93–111, 1992.
15. David S. Warren. Interning ground terms in XSB. In *Proceedings of CICLOPS*, 2013.
16. Neng-Fa Zhou and Agostino Dovier. A tabled Prolog program for solving Sokoban. *Fundam. Inform.*, 124(4):561–575, 2013.
17. Neng-Fa Zhou and Jonathan Fruhman. Toward a dynamic programming solution for the 4-peg tower of Hanoi problem with configurations. In *Proceedings of CICLOPS*, 2012.
18. Neng-Fa Zhou and Jonathan Fruhman. A User's Guide to Picat, 2013.
19. Neng-Fa Zhou and Christian Theil Have. Efficient tabling of structured data with enhanced hash-consing. *TPLP*, 12(4-5):547–563, 2012.
20. Neng-Fa Zhou, Yoshitaka Kameya, and Taisuke Sato. Mode-directed tabling for dynamic programming, machine learning, and constraint solving. In *ICTAI*, pages 213–218, 2010.
21. Neng-Fa Zhou, Taisuke Sato, and Yi-Dong Shen. Linear tabling strategies and optimizations. *TPLP*, 8(1):81–109, 2008.