Mode-Directed Tabling for Dynamic Programming, Machine Learning, and Constraint Solving

Neng-Fa Zhou
CUNY Brooklyn College & Graduate Center
http://www.sci.brooklyn.cuny.edu/˜ zhou/

Yoshitaka Kameya
Tokyo Institute of Technology
http://sato-www.cs.titech.ac.jp/kameya/

Taisuke Sato
Tokyo Institute of Technology
http://sato-www.cs.titech.ac.jp/sato/

***Abstract*—The abstract goes here.**

I. Introduction

Tabling [12], [13], [15] has been found increasingly important for not only helping beginners write workable declarative programs but also developing real-world applications such as natural language processing, model checking, and machine learning applications. The idea of tabling is to memorize answers to tabled calls and use the answers to resolve subsequent variant or subsumed calls. This idea resembles the dynamic programming idea of reusing solutions to overlapping sub-problems and, naturally, tabling is amenable to dynamic programming problems. In a traditional tabling system, all the arguments of a tabled call are used in variant or subsumption checking and all answers are tabled for a tabled predicate. This non-selective approach is problematic for many dynamic programming problems such as those that require computation of aggregates. The aggregate predicates as provided by XSB [11] can be used but they require tabling all possible solutions. Mode-directed tabling [7] has been proposed to remedy this weakness. The idea of mode-directed tabling is to use table modes to control what arguments should be used in variant checking and how answers should be tabled.

In this paper, we extend the table mode declaration proposed in [7] and show several application examples. Our table mode declaration takes the form of

```
:-table p(M1,...,Mn):C.
```

where `p/n` is a predicate symbol, `C` (called a *cardinality limit*) is an integer which limits the number of answers to be tabled, and each `Mi` (i=1,...,n) is a mode which can be `min`, `max`, `+` (input), `-` (output), or `nt` (non-tabled). An argument with the mode `min` or `max` is assumed to be output. The system uses only input arguments in variant checking, disregarding all output arguments and arguments with the `nt` mode. After an answer is produced, the system tables it unconditionally if the cardinality limit is not exceeded yet. When the cardinality limit has been reached, however, the system tables the answer only if it is better than some existing answer in terms of the argument with the `min` or `max` mode. No arguments of answers with mode `nt` are tabled.

Our table mode declaration has the following two new features: first, a mode-directed tabled predicate can have multiple answers produced, and therefore, the relation from the inputs to the outputs is not required to be a function; and second, the mode `nt` is new, which allows arguments to be disregarded not only in variant checking but also in answer tabling.

Our extended table mode declaration was motivated by the need to scale up the PRISM system [9], [10] for handling large data sets. For a given set of possibly incomplete observed data, PRISM constructs the explanation graph using tabling and estimates the probability distributions by conducting EM learning on the graph. For many real-world applications, the explanation graphs are too large to be completely stored. Mode-directed tabling allows for construction of partial explanation graphs. We'll give an example to illustrate the use of the table mode declaration in PRISM.

The new mode `nt` is suitable for describing non-discriminating [2] arguments that have no relationship with input arguments. For example, for the `append` predicate that appends two given lists, the mode can be declared as `append(+,nt,-)`, which disregards the second argument in tabling. In this way, table space can be significantly saved. We'll show how the mode `nt` can be used in performing constraint checking for an ASP program.

The table mode declaration has been implemented in B-Prolog [14] and all the examples presented in this paper have been tested. This paper is structured as follows: Section 2 details the semantics of table modes; Section 3 gives solutions to a dynamic programming problem, called *hydraulic planning*, which was one of the benchmarks used for the Second ASP solver competition; Section 4 presents a PRISM example for a data mining application; Section 5 shows how mode-directed tabling can be used to evaluate an ASP program for the Hamilton cycle problem; and Section 6 concludes the paper.

II. Mode-Directed Tabling

Mode-directed tabling amounts to using table modes to instruct the system on how to table subgoals and their answers. This section describes how to declare table modes in B-Prolog. Mode-directed tabling is orthogonal to tabling approaches (e.g., suspension-based SLG [8] or iteration-based linear tabling [15]), subgoal testing methods (e.g., variant testing or subsumption testing), and answer returning strategies (local or batched [3], also called lazy or eager [15]). Therefore, table modes can be introduced into other tabling systems.

A table mode declaration takes the following form:

```
:-table p(M1,...,Mn):C.
```

where `p/n` is a predicate symbol, `C`, called a *cardinality limit*, is an integer which limits the number of answers to be tabled for `p/n`, and each `Mi` (i=1,...,n) is a mode which can be `min`, `max`, `+`, `-`, or `nt`. When `C` is 1, it can be ommitted together with the preceeding ':'. For each predicate, only one table mode declaration can be given. In the current implementation in B-Prolog, only one argument in a tabled predicate can have the mode `min` or `max`. Because an optimized argument can be a compound term and the built-in `@<`/2 is used to select better answers for compound terms, this restriction is not essential.

The mode `+` is called *input*, `-` *output*, `nt` *non-tabled*, `min` *minimized*, and `max` *maximized*. An argument with the mode `min` or `max` is also called *optimized*. An optimized argument is assumed to be output. The system uses only input arguments in variant checking of tabled subgoals and ignores all other arguments. Notice that a table mode does not tell the instantiation state of an argument. An input argument can be variable and an output argument can be ground.

A mode declaration not only instructs on what arguments are used in variant checking, it also guides the system in tabling answers. After an answer of a tabled predicate is produced, the system tables it unconditionally if the cardinality limit is not exceeded yet. When the cardinality limit has been reached, however, the system tables the answer only if it is better than some existing answer in terms of the optimized argument. If no argument is optimized, all new answers are discarded once the cardinality limit has been reached. When an answer is tabled, no arguments with the mode `nt` are tabled.

Non-tabled arguments normally are used to pass global data to a tabled predicate that are never to be updated. They are global to a predicate in the sense that all subgoals of the predicate can have access to them and they should not be instantiated if they are variables. Non-tabled arguments can contain attributed variables and hence in general cannot be simulated using global variables. Even in cases where non-tabled arguments can be represented as global variables, they are much faster to access than global variables. Note that an argument should not be declared non-tabled if it is dependent on the input arguments or some output argument is dependent on it.

For a tabled program, the same query may return different answers under control of different modes. Consider the following predicate:

```
p(1,1).
p(1,2).
p(1,3).
p(2,3).
```

and the query `p(1,X)`. The following gives different answers under control of different modes.

```
% :-table p(+,-).
?-p(1,X).
X=1;
no

% :-table p(+,-):2.
?-p(1,X).
X=1;
X=2;
no

% :-table p(+,min):2.
?-p(1,X).
X=1;
X=2;
no

% :-table p(+,max):2.
?-p(1,X).
X=3;
X=2;
no

% :-table p(nt,-):3.
?-p(1,X).
X=1;
X=2;
X=3;
no
```

III. Mode-Directed Tabling for Dynamic Programming

The traditional non-selective approach to tabling is problematic for many dynamic programming optimization problems such as those that require computation of aggregates. For example, for the shortest path problem, there may be a huge number of paths between two nodes, and it does not make sense to table all the paths first and then find a shortest one. Mode-directed tabling allows tabling only intermediate results that are useful for finding the final result. In this section, we present solutions to hydraulic system planning, a dynamic programming problem used as one of the benchmarks in the second ASP solver competition.[1]

A. Hydraulic system planning

A simplified version of the hydraulic system on a space shuttle consists of a directed graph, G, such that:

- Nodes of this graph are labeled as tanks,jets, or junctions.
- Every link between two nodes is labeled by a valve.
- There are no paths in G between any two tanks.
- For every jet there always is a path in G from a tank to this jet.

Tanks can be full or empty. Valves can be open or closed. Some of the valves can be stuck in the close position. A state of G is specified by the set of full tanks, the set of open valves, and the set of valves stuck in the closed position.

[1]http://www.cs.kuleuven.be/ dtai/events/ASP-competition/index.shtml

A node of G is called pressurized in a state if it is a full tank or if there exists a path from some full tank of G to this node such that all the valves on the edges of this path are open.

We assume that in a state a shuttle controller can open a valve corresponding to a directed link $<N_1, N_2>$ only if N_1 is pressurized and not stuck.

Given a graph G together with an initial state and a jet J, the problem is to help a shuttle controller find a shortest sequential plan to pressurize J.

B. A solution

The following shows a tabled program for finding a shortest plan to pressurize a node.

```
:-table pressurize(+,-,min).
pressurize(Node,Plan,Len):-
    full(Node),!, Plan=[],Len=0.
pressurize(Node,[Valve|Plan],Len):-
    link(AnotherNode,Node,Valve),
    \+ stuck(Valve),
    pressurize(AnotherNode,Plan,Len1),
    Len is Len1+1.
```

A node is pressurized if it is a full tank or it is linked to a pressurized node with a link that is not stuck. The following predicates are given to represent the graph and its attributes:

- `link(`N_1`, `N_2`, `V`)`: V is the valve on the pipe connecting node N_1 and N_2.
- `full(`T`)`: tank T is full. A tank is empty if it is not mentioned to be full explicitly.
- `stuck(`V`)`: valve V is stuck. A valve is not stuck if it is not mentioned to be stuck explicitly.

A call `pressurize(Node,Plan,Len)` pressurizes `Node` with a plan `Plan` of length `Len`. The table mode `pressurize(+,-,min)` indicates that for each node only one plan with the shortest length is compputed.

C. Hydraulic system planning with leaking valves

Mow let's consider a variant of the problem where some of the valves are leaking. Given a graph G together with an initial state of G and a jet J, a shuttle controller needs to find a shortest plan among those using the least number of leaking valves.

D. A solution to the variant problem

The following give a tabled program for solving the problem.

```
:-table pressurize(+,-,min).
pressurize(Node,Plan,(Leaks,Len)):-
    full(Node),!,
    Plan=[],Leaks=0,Len=0.
pressurize(Node,[Valve|Plan],(Leaks,Len)):-
    link(AnotherNode,Node,Valve),
    \+ stuck(Valve),
    pressurize(AnotherNode,Plan,Pair),
    Pair=(Leaks1,Len1),
    Len is Len1+1,
    (leaking(Valve)->
        Leaks is Leaks1+1;
        Leaks is Leaks1).
```

The subgoal `leaking(Valve)` is true if `Valve` is leaking. A valve is assume to be not leaking if it is not mentioned to be leaking explicitly. For each plan, two attribute values are computed: `Leaks` is the number of leaking valves involved in the plan and `Len` is length of the plan. Because only one argument can be optimized in a tabled predicate in the current implementation in B-Prolog, these two values are combined into a pair `(Leaks,Len)`. Note that the ordering of the constituents is important. If the pair were `(Len,Leaks)`, then the plan returned would have the least number of leaking valves among the shortest plans.

IV. Mode-Directed Tabling for Constraint Solving

It is well known that tabling is useful in top-down evaluation of logic programs with stratified negation, and it is also known that tabling can be used in top-down computation of well-founded semantics of logic programs with non-stratified negation [1]. Answer set programming (ASP) gives a lgoic program with non-stratified negation a different semantics called stable model or answer-set semantics [5]. ASP has become another constraint language for modeling and solving combinatorial search problems.

For a logic program with non-stratified negation, the computation of an answer set requires iterations of guessing, propagation, and testing, wheather the program is gounded into a propsitional one solved by a SAT solver (e.g., Clasp [4]) or compiled into a constraint program solved by propagation (e.g., NPDatalog [6]). Guessing or labeling amounts to giving a truth value to an atom, propagation entails to derive the truth values of atoms, and testing means to guarantee that certain atoms can be derived to be true.

Consider the following program for finding a Hamilton cycle in a given directed graph:[2]

```
{c(X,Y)} :- e(X,Y).

:- 2 {c(X,Y) : e(X,Y)}, v(X).
:- 2 {c(X,Y) : e(X,Y)}, v(Y).

r(X) :- c(0,X), v(X).
r(Y) :- c(X,Y), r(X), e(X,Y).

:- not r(X), v(X).
```

The given graph is represented by two relations: `v(X)` means `X` is a node, and `e(X,Y)` means that `Y` is connected to `X`. The first rule is a choice rule which constrains the relation `c/2` to be a sub-relation of `e/2`. The second and third rules ensure that no node has two or more incoming or outgoing arcs in the relation `c/2`. The relation `r/1` together with the

[2]http://en.wikipedia.org/wiki/Answer_set_programming

constraint (last the rule) below its definition ensure that every node is reachable. Obviously, a relation `c/2` that satisifies all the constraints forms a Hamilton cycle.

We use two sets, called `IN` and `OUT`, to denote the set of atoms of `c/2` that are currently known to be true and false, respectively. In the beginning, both sets are empty. Atoms are added into these sets either through labeling or propagation. The relation `r/1` can be encoded as the following tabled predicate:

```
:-table reach(+,nt).
reach(X,OUT):-
      not_member(c(0,X),OUT).
reach(X,OUT):-
      e(X,Y),
      not_member(c(X,Y),OUT),
      reach(Y,OUT).
```

The suhgoal `not_member(Elm,Set)` succeeds if `Elm` is not an element in `Set`. Tuples in a set are indexed so that the test fast is fast.

References

[1] Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.

[2] Henning Christiansen and John P. Gallagher. Non-discriminating arguments and their uses. In *ICLP*, pages 55–69, 2009.

[3] J. Freire, T. Swift, and D. S. Warren. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. *Journal of Functional and Logic Programming*, 1998.

[4] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A User's Guid to gringo, clasp, clingo and iclingo. Technical report, University of Potsdam, 2008.

[5] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080, 1988.

[6] Sergio Greco, Cristian Molinaro, Irina Trubitsyna, and Ester Zumpano. NP Datalog: a Logic Language for Expressing NP Search and Optimization Problems. *Theory and Practice of Logic Programming: Special Issue on Constraint Handling Rules*, 10(2):125–166, 2010.

[7] Hai-Feng Guo and Gopal Gupta. Simplifying dynamic programming via mode-directed tabling. *Softw., Pract. Exper.*, 38(1):75–94, 2008.

[8] Konstantinos Sagonas and Terrance Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, 1998.

[9] Taisuke Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, pages 391–454, 2001.

[10] Taisuke Sato, Yoshitaka Kameya, and et. al. The prism user's manual, March 2010. `http://www.mi.cs.titech.ac.jp/prism/`.

[11] Terrance Swift, David S. Warren, et al. The XSB Programmer's Manual: version 3.2, vols. 1 and 2, 2009. http://xsb.sf.net.

[12] Hisao Tamaki and Taisuke Sato. OLD resolution with tabulation. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 84–98, 1986.

[13] D. S. Warren. Memoing for logic programs. *Comm. of the ACM, Special Section on Logic Programming*, 35:93–111, 1992.

[14] Neng-Fa Zhou. B-Prolog users manual, version 7.4, 2010. http://www.probp.com/.

[15] Neng-Fa Zhou, Taisuke Sato, and Yi-Dong Shen. Linear tabling strategies and optimizations. *Theory and Practice of Logic Programming (TPLP)*, 8(1):81–109, 2008.