# On Improving the Efficiency and Robustness of Table Storage Mechanisms for Tabled Evaluation

Ricardo Rocha

DCC-FC & LIACC
University of Porto, Portugal
ricroc@ncc.up.pt

**Abstract.** Most of the recent proposals in tabling technology were designed as a means to improve some practical deficiencies of current tabling execution models that reduce their applicability in particular applications. The discussion we address in this paper was also motivated by practical deficiencies we encountered, in particular, on the table storage mechanisms used for tabling support. To improve such mechanisms, we propose two new implementation techniques that make tabling models more efficient when dealing with incomplete tables and more robust when recovering memory from the table space. To validate our proposals, we have implemented them in the YapTab tabling system as an elegant extension of the original design.

## 1 Introduction

Tabling [1, 2] is a technique of resolution that overcomes some limitations of traditional Prolog models in dealing with recursion and redundant sub-computations. As a result, in the past years several alternative tabling models have been proposed [3–8] and implemented in systems like XSB, Yap, B-Prolog, ALS-Prolog and Mercury.

More recently, the increasing interest in tabling technology led to further developments and proposals that improve some practical deficiencies of current tabling execution models. In [9], Sagonas and Stuckey proposed a mechanism, named *just enough tabling*, that offers the capability to arbitrarily suspend and resume a tabled evaluation without requiring full re-computation. In [10], Saha and Ramakrishnan proposed an incremental evaluation algorithm for maintaining the freshness of tables that avoids recomputing the full set of answers when the program changes upon addition or deletion of facts/rules. In [11], Rocha *et al.* proposed the ability to support dynamic mixed-strategy evaluation of the two most successful tabling scheduling strategies, batched and local scheduling.

All these recent proposals were designed as a means to improve the performance of particular applications in key aspects of tabled evaluation like recomputation and scheduling. The discussion we address in this work was also motivated by our recent attempt of applying tabling to Inductive Logic Programming (ILP) [12]. ILP applications are very interesting for tabling because they have huge search spaces and do a lot of re-computation. In [13] we showed

that tabling is indeed a promising approach to minimize re-computation in ILP systems and that one can have impressive gains through tabling. However, we found that current tabling execution models suffer from significant limitations that reduce their applicability in many ILP applications. Analysis showed two major issues with the table storage mechanisms used for tabling support.

A first problem is *incomplete tabling*. Tabling is about storing answers for subgoals so that they can be reused when a repeated call appears. On the other hand, most ILP algorithms are interested in example satisfiability, not in the answers: query evaluation stops as soon as an answer is found. This is usually implemented by *pruning* at the Prolog level. Unfortunately, pruning over tabled computations results in *incomplete tables*: we may have found several answers but not the complete set. Thus, usually, when a repeated call appears we cannot simply trust the answers from an incomplete table because we may lose part of the computation. The simplest approach, and the one that has been implemented in most tabling systems, is to throw away incomplete tables, and restart the evaluation from scratch. In this work, we propose a more aggressive approach where, by default, we keep incomplete tables around. Whenever a call for an incomplete table appears, we first consume the answers from the table. If the table is exhausted, then we will restart the evaluation from the beginning. The main goal of this proposal is to avoid re-computation when the already stored answers are enough to evaluate a repeated call.

A second problem is *memory recovery*. When we use tabling for applications that build very many queries or that store a huge number of answers, we can build arbitrarily many or very large tables, quickly running out of memory space. In general, we will have no choice but to throw away some of the tables (ideally, the least likely to be used next). Tabling systems have not really addressed this problem. At most, they have a set of tabling primitives that the programmer can use to dynamically abolish some of the tables. However, this can be hard to use and very difficult to decide what are the potentially useless tables that should be deleted. In this work, we propose a more suitable approach for large dynamic searches, a memory management strategy based on a *least recently used* algorithm, that dynamically recovers space from the least recently used tables when the system runs out of memory.

Both proposals have been implemented in the YapTab tabling system [14] with minor changes to the original design. Preliminaries results using the April ILP system [15] showed very substantial performance gains and a substantial increase of the size of the problems that can be solved by combining ILP with tabling. Despite the fact that we used ILP as the motivation for this work, our proposals are not restricted to ILP applications and can be generalised and applied to most other applications.

The remainder of the paper is organized as follows. First, we briefly introduce some background concepts and discuss the motivation for our work. Next, we present our proposals and describe the issues involved in providing engine support for integrating them in the YapTab tabling system. We then present some experimental results and outline some conclusions.

## 2 Background and Motivation

To discuss the motivation for our work, we start by introducing some basic concepts about tabling and ILP and then we address the practical deficiencies encountered when combining them.

### 2.1 Basic Tabling Definitions

The basic idea behind tabling is straightforward: programs are evaluated by storing answers for current subgoals in a proper data space, called the *table space*. Whenever a repeated call is found, the subgoal's answers are recalled from the table instead of being re-evaluated against the program clauses. The nodes in a tabled evaluation are classified as either: *generator nodes*, corresponding to first calls to tabled subgoals; *consumer nodes*, corresponding to repeated calls to tabled subgoals; or *interior nodes*, corresponding to non-tabled subgoals. Tabling based models have four main types of operations for definite programs:

1. The *tabled subgoal call* operation is a call to a tabled subgoal. It checks if the subgoal is in the table. If so, it allocates a consumer node and starts consuming the available answers. If not, it adds a new entry to the table, and allocates a new generator node.
2. The *new answer* operation checks whether a newly found answer is already in the table, and if not, inserts the answer. Otherwise, the operation fails.
3. The *answer resolution* operation checks whether extra answers are available for a particular consumer node and, if so, consumes the next one. If no unconsumed answers are available, it *suspends* the current computation and schedules a backtracking node to continue the execution.
4. The *completion* operation determines whether a tabled subgoal is *completely evaluated*. A table is said to be *complete* when its set of stored answers represent all the conclusions that can be inferred from the set of facts and rules in the program for the subgoal call associated with the table. Otherwise, it is said to be *incomplete*. A table for a tabled subgoal is thus marked as complete when, during evaluation, it is determined that all possible resolutions have been made and, therefore, no more answers can be found.

We could delay completion until the very end of the execution. Unfortunately, doing so would also mean that we could only recover space for consumers (suspended subgoals) at the very end of the execution. Instead we shall try to achieve *incremental completion* [16] to detect whether a generator node has been fully exploited and, if so, to recover space for all its consumers. Moreover, if we call a repeated subgoal that is already completed, then we can avoid consumer node allocation and perform instead what is called a *completed table optimization* [17]. This optimization allocates a node, similar to an interior node, that will consume the set of found answers executing compiled code directly from the table data structures associated with the completed subgoal.

### 2.2 Inductive Logic Programming

The fundamental goal of an ILP system is to find a consistent and complete theory (logic program), from a set of examples and prior knowledge, the *background knowledge*, that explains all given positive examples, while being consistent with the given negative examples. Since it is not usually obvious which set of hypotheses should be picked as the theory, an ILP system must traverse the *hypotheses space* searching for a set of hypotheses (clauses) with the desired properties.

Computing the coverage of a hypothesis requires, in general, running positive and negative examples against the clause. For instance, to evaluate if the hypothesis `'theory(X):- a1(X),a2(X,Y).'` covers the example `theory(p1)`, the system executes the goal `once(a1(p1),a2(p1,Y))`. The `once/1` predicate is a primitive that prunes over the search space preventing the unnecessary search for further answers. It is defined in Prolog as `'once(Goal):- call(Goal),!.'`. Note that the ILP system is only interested in evaluating the coverage of the hypothesis, and not in finding answers for the goal.

Now assume that the previous hypothesis obtains a *good coverage*, that is, the number of positive examples covered by it is high and the number of negative examples is low. Then, it is quite possible that the system will use it to generate more specific hypotheses such as `'theory(X):- a1(X),a2(X,Y),a3(Y).'`. If the same example, `theory(p1)`, is then evaluated against this new hypothesis, goal `once(a1(p1),a2(p1,Y),a3(Y))`, part of the computation will be repeated. For data-sets with a large number of examples, we can do an arbitrarily large amount of re-computation.

### 2.3 Tabling and Inductive Logic Programming

In previous work, we have already proposed two approaches of using tabling to minimize re-computation in ILP systems [13]. The first approach is simply to table subgoals. This approach requires minimal changes to the ILP system and comes for free if using a Prolog engine with tabling support. A second approach is to table prefixes, that is, replace the conjunction of subgoals in the hypotheses with proper tabled predicates inferred during execution. If we are able to table these conjunction of subgoals, we only need to compute them once. This strategy can be recursively applied as the system generates more specific hypotheses. This idea is similar to the *query packs* technique proposed by Blockeel *et al.* [18].

However, we have found two major problems with the table storage mechanisms currently used for tabling support that reduce their applicability in many ILP applications. One of these problems is memory recovery. To recursively table conjunction of subgoals, we need to store a large number of tables, and thus, we may increase the table memory usage arbitrarily and quickly run out of memory [13]. Therefore, at some point, we need to compromise efficiency and throw away some of the tables in order to recover space. A first approach is to let the programmer dynamically control the deletion of the tables. However, this puts the burden on the ILP designer, and in the worst case may result in removing useful tables. In order to allow useful deletion without compromising efficiency,

we propose in this work a more robust approach, a memory management strategy based on a *least recently used* replacement algorithm that dynamically recovers space from the tables when the system runs out of memory.

The other problem is incomplete tabling. Consider again the evaluation of `once(a1(p1),a2(p1,Y),a3(Y))` but now with `a2/2` declared as tabled. Coverage computation with tabled evaluation works fine when examples are not covered by hypotheses. In such cases, all tabled subgoals in a clause are completed. For instance, when evaluating the goal `once(a1(p1),a2(p1,Y),a3(Y))`, if the subgoal `a3(Y)` never succeeds then, by backtracking, `a2(p1,Y)` will be completely evaluated. On the other hand, tabled evaluation can be a problem when examples are successfully covered by hypotheses. For example, if `once(a1(p1),a2(p1,Y),a3(Y))` eventually succeeds, then the `once/1` primitive will reclaim space by pruning the goal at hand. However, as `a2(p1,Y)` may still succeed with other answers for `Y`, its table entry cannot be marked as complete. Thus, when a repeated call to `a2(p1,Y)` appears, we cannot simply load answers from its incomplete table, because we may lose part of the computation. A question then arises: how can we make tabling worthwhile in an environment that potentially generates so many incomplete tables?

We first tackled this problem by taking advantage of YapTab's functionality that allows to combine different *scheduling strategies* within the same tabled evaluation [11]. Our results showed that best performance can be achieved when we evaluate some subgoals using *batched scheduling* and others using *local scheduling*. Batched scheduling is the default strategy, it schedules the program clauses in a depth-first manner as does the WAM. This strategy favors forward execution, when a new answer is found the evaluation automatically propagates the answer to solve the goal at hand. Local scheduling is an alternative strategy that tries to *force* completion before returning answers. The key idea is that whenever new answers are found, they are added to the table space, as usual, but execution fails. Answers are only returned when all program clauses for the subgoal at hand were resolved.

At first, local scheduling seems more attractive because it avoids incomplete tabling. When the `once/1` primitive prunes the search space, the tables are already completed. On the other hand, if the cost of fully generating the complete set of answers is very expensive, then the ILP system may not always benefit from it. It can happen that, after completing a subgoal, the subgoal always succeeds just by using the initial answers, making it useless to compute beforehand the full set of answers. We believe that it is very difficult to define the best strategy to evaluate each subgoal. The approach we propose in this work can be seen as a compromise between the efficiency of batched scheduling and the effectiveness of local scheduling. We want to favor forward execution in order to quickly succeed with the coverage evaluation of the hypotheses, but we also want to be able to reuse the answers already found in order to avoid re-computation.

We next describe how we extended the YapTab tabling system to be more efficient when dealing with incomplete tables and more robust when recovering memory from the table space.

## 3 Incomplete Tabling

This section describes how we extended YapTab to support incomplete tabling. The main goal of our proposal is to avoid re-computation when the answers in an incomplete table are enough to evaluate a repeated call. To support that, we thus keep incomplete tables for pruned subgoals. Then, when a repeated call to a pruned subgoal appears, we start by consuming the available answers from its incomplete table, and only if we exhaust all such answers, we restart the evaluation from the beginning. Later, if the subgoal is pruned again, then the same process is repeated until eventually the subgoal is completely evaluated.

### 3.1 Implementation Details

In YapTab, tables are implemented using *tries* as proposed in [17]. An important data structure in the table space is the *subgoal frame*. For each different tabled subgoal call, a different subgoal frame is used to store information about the subgoal. In particular, part of that information includes a pointer to where answers are stored, the `SgFr_answers` field, and a flag indicating the state of the subgoal, the `SgFr_state` field (see Fig. 1 for details).

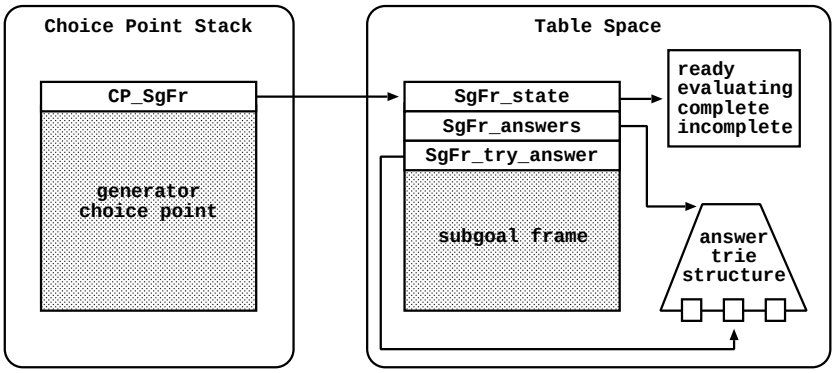


**Fig. 1.** Generator choice points and subgoal frames in YapTab

During evaluation, a subgoal frame can be in one of the following states: *ready*, i.e., without a corresponding generator in the choice point stack; *evaluating*, i.e., with a generator being evaluated; or *complete*, i.e., with the generator no longer present but with the subgoal fully evaluated. At the engine level, generator nodes are implemented as WAM choice points extended with two extra fields [11]. One of these fields, the `CP_SgFr` field, points to the associated subgoal frame in the table space.

To support incomplete tabling, we have introduced two minor changes to the subgoal frame data structure. First, a new *incomplete* state, marks the subgoals whose corresponding generators were pruned from the execution stacks. Second, when we are consuming answers from an incomplete table as a result of a repeated call to a previously pruned subgoal, a new `SgFr_try_answer` field marks the currently loaded answer (similarly to what consumer nodes have).

Handling incomplete tables also required minor changes to the tabled subgoal call operation. Figure 2 shows how we extended the `tabled_subgoal_call()` instruction to deal with incomplete tables.

```
tabled_subgoal_call(subgoal SG) {
  sg_fr = search_table_space(SG)     // sg_fr is the subgoal frame for SG
  if (SgFr_state(sg_fr) == ready) {
    gen_cp = store_generator_node(sg_fr)
    SgFr_state(sg_fr) = evaluating
    CP_AP(gen_cp) = failure_continuation_instruction()   // second clause
    goto next_instruction()
  } else if (SgFr_state(sg_fr) == evaluating) {
    cons_cp = store_consumer_node(sg_fr)
    goto answer_resolution(cons_cp)            // start consuming answers
  } else if (SgFr_state(sg_fr) == complete) {
    goto SgFr_answers(sg_fr)       // execute compiled code from the trie
  } else if (SgFr_state(sg_fr) == incomplete) {      // new block of code
    gen_cp = store_generator_node(sg_fr)
    SgFr_state(sg_fr) = evaluating
    first = get_first_answer(sg_fr)
    load_answer_from_trie(first)
    SgFr_try_answer(sg_fr) = first      // mark the current loaded answer
    CP_AP(gen_cp) = table_try_answer                  // new instruction
    goto continuation_instruction()
  }
}
```

**Fig. 2.** Pseudo-code for `tabled_subgoal_call()`

The new block of code that deals with incomplete tables is similar to the block of code that deals with first calls to tabled subgoals (ready state flag). It also stores a generator node, but instead of using the program clauses to evaluate the subgoal call, as usual, it starts by loading the first available answer from the incomplete table. The subgoal's `SgFr_try_answer` field is made to point to this first answer. A second difference is that the failure continuation pointer of the generator choice point, the `CP_AP` field, is now updated to a special `table_try_answer` instruction.

When backtracking occurs, the `table_try_answer` instruction implements a variant of the answer resolution operation (see section 2.1). Figure 3 shows the pseudo-code for it. Initially, the `table_try_answer` instruction checks if there

```
table_try_answer(generator GEN) {
  sg_fr = CP_SgFr(GEN)
  last = SgFr_try_answer(sg_fr)             // get the last loaded answer
  next = get_next_answer(last)
  if (next) {                                  // answers still available
    load_answer_from_trie(next)
    SgFr_try_answer(sg_fr) = next     // update the current loaded answer
    goto continuation_instruction()
  } else {                // restart the evaluation from the first clause
    load_compiled_code(sg_fr)               // adjust the program counter
    CP_AP(GEN) = failure_continuation_instruction()      // second clause
    goto next_instruction()
  }
}
```

**Fig. 3.** Pseudo-code for `table_try_answer()`

are more answers to be consumed, and if so, it loads the next one and updates the `SgFr_try_answer` field. When this is not the case, all available answers have been already consumed. Thus, we need to restart the computation from the beginning. The program counter is made to point to the first clause corresponding to the subgoal call at hand and the failure continuation pointer of the generator is updated to the second clause. At this point, the evaluation is in the same computational state as if we had executed a first call to the tabled subgoal call operation. The difference is that the table space for our subgoal already stores some answers.

We should remark that the use of generator nodes to implement the calls to incomplete tables is strictly necessary to keep unchanged all the remaining data structures and algorithms of the tabling engine. Note that, at the engine level, these calls are again the first representation of the subgoal in the execution stacks because the previous representation has been pruned.

### 3.2 Discussion

Let us consider again the previous ILP example and the evaluation of the goal `once(a1(p1),a2(p1,Y),a3(Y))` with predicate `a2/2` declared as tabled. Consider also that, after a long computation for `a2(p1,Y)`, we have found three answers: `Y=y1`, `Y=y2`, and `Y=y3`, and that `a3(Y)` only succeeds for `Y=y3`. Primitive `once/1` then prunes the goal at hand and `a2(p1,Y)` is marked as incomplete. Now assume that, later, the ILP system calls again `a2(p1,Y)` when evaluating a different goal, for example, `once(a2(p1,Y),a4(Y))`. If `a4(Y)` succeeds with one of the previously found answers, then no evaluation will be required for subgoal `a2(p1,Y)`. This is the typical case where we can profit from having incomplete tables. The gain in the execution time is proportional to the cost of evaluating the subgoal from the beginning until generating the proper answer.

On the other hand, if `a4(Y)` does not succeed with any of the previously found answers, then `a2(p1,Y)` will be reevaluated as a first call. This means that the answers `Y=y1`, `Y=y2` and `Y=y3` will be generated again. However, as these answers are repeated, the evaluation will fail and `a4(Y)` will not be called again for them. The evaluation will fail until a non-repeated answer is eventually found. Thus, the computation time required to evaluate `once(a2(p1,Y),a4(Y))`, either with or without the incomplete table, is then equivalent. Therefore, we may not benefit from having maintained the incomplete table, but we do not pay any cost either.

Our proposal is close to the spirit of the *just enough tabling (JET)* proposal of Sagonas and Stuckey [9]. In a nutshell, the JET proposal offers the capability to arbitrarily suspend and resume a tabled evaluation without requiring any re-computation. The basic idea is that JET copies the execution stacks corresponding to pruned subgoals to an auxiliary area in order to be able to resume them later when a repeated call appears. The authors argue that the cost of JET is linear in the number of choice points which are pruned. However, to the best of our knowledge, no practical implementation of JET was yet been done.

Compared to JET, our approach does not require an auxiliary data space, does not require any complex dependencies to maintain information about pruned subgoals, and does not introduce any overhead in the pruning process. We thus believe that the simplicity of our approach can produce comparable results to JET when applied to real applications like ILP applications.

## 4 Memory Recovery

This section describes our proposal to handle tables when the system runs out of memory. We propose a memory management strategy that automatically recovers space from the least recently used tables. Note that this proposal is completely orthogonal to the previous one, that is, we can support either or both simultaneously. In what follows, we will thus consider the case where YapTab also includes support for incomplete tabling as described in the previous section.

### 4.1 Implementation Details

In YapTab, each tabled subgoal call is represented by a different subgoal frame in the table space. Besides this representation, a subgoal can also be represented in the execution stacks. First calls to tabled subgoals or calls to previously pruned subgoals are represented by generator nodes; repeated calls to tabled subgoals are represented by consumer nodes; and calls to completed subgoals are represented by interior nodes that execute compiled code directly from the answer trie structure associated with the completed subgoal. A subgoal is said to be *active* if it is represented in the execution stacks. Otherwise, it is said to be *inactive*. Inactive subgoals are thus only represented in the table space.

A subgoal can also be in one of the following states: ready, evaluating, complete or incomplete. The ready and incomplete states correspond to situations where the subgoal is inactive, while the evaluating state corresponds to a situation where the subgoal is active. The complete state is a special case because it can correspond to both active and inactive situations. In order to be able to distinguish these two situations, we introduced a new state named *complete-active*. We use the complete-active state to mark the completed subgoals that are also active in the execution stacks, while the previous complete state is used to mark the completed subgoals that are only represented in the table space. With this simple extension, we can now use the `SgFr_state` field of the subgoal frames to decide if a subgoal is currently active or inactive.

Knowing what subgoals are active or inactive is important when the system runs out of memory. Obviously, active subgoals cannot be removed from the table space because otherwise we may lose part of the computation or produce errors. Therefore, when the system runs out of memory, we should try to recover space from the inactive subgoals. Figure 4 shows how we handle inactive subgoals in YapTab.

Subgoal frames corresponding to inactive subgoals are kept in a double linked list that is accessible by two new global registers. The `Inact_most` register points

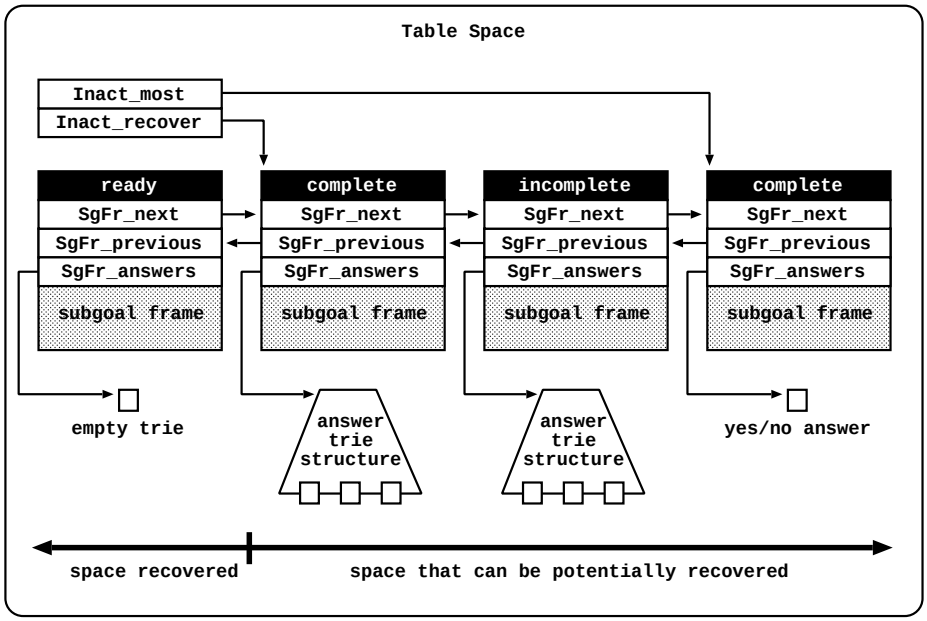


**Fig. 4.** Inactive subgoals in YapTab

to the most recently inactive subgoal frame and the `Inact_recover` register points to the least recently inactive subgoal frame from where space can be potentially recovered. Two subgoal frame fields, `SgFr_next` and `SgFr_previous`, link the list. Space from inactive subgoals is recovered as presented next in Fig. 5.

```
recover_space(structure data type STR_TYPE) {
  // STR_TYPE is the data type that we failed to allocate space for
  sg_fr = Inact_recover
  do {
    if (sg_fr == NULL)                                    // end of list
      return
    if (get_first_answer(sg_fr)) {          // subgoal frame with answers
      free_answer_trie_structure(sg_fr)                  // recover space
      SgFr_state(sg_fr) = ready                 // reset the frame state
    }
    sg_fr = SgFr_next(sg_fr)
  } while (no_space_available_for(STR_TYPE))
  Inact_recover = sg_fr                            // update recover field
}
```

**Fig. 5.** Pseudo-code for `recover_space()`

The `recover_space()` procedure is called when the system fails to allocate memory space for a specific data type, the `STR_TYPE` argument. It starts from the subgoal frame pointed by the `Inact_recover` register and then uses the `SgFr_next` field to navigate in the list of inactive subgoals until at least a page of memory is recovered. YapTab uses a page-based memory allocation scheme where each page only stores data structures of the same type, and thus, to start

using a memory page to allocate a different data structure, we first need to completely deallocate all the previous data structures from the page.

When recovering space, we only consider the subgoals that store at least one answer (completed subgoals with a yes/no answer are kept unchanged) and for these we only recover space from their answer trie structures. Through experimentation we found that, for a large number of applications, the space required by all the other table data structures is insignificant when compared with the space required by the answer trie structures (usually more than 99% of the total table space). Therefore, only sporadically, we are able to recover space from the non-answer related data structures. We thus argue that the potential benefit of recovering space from these structures does not compensate its cost.

During evaluation, an inactive subgoal can be made active again. This occurs when we execute a repeated call to an inactive subgoal. For such cases, we thus need to remove the corresponding subgoal frame from the list. On the other hand, when a subgoal turns inactive, its subgoal frame is inserted in the list as the most recently inactive frame. A subgoal turns inactive when it executes completion, it is pruned or it fails from an interior node that was executing compiled code from the answer trie structure.

This latter case can be complicated because we can have several interior nodes executing compiled code from the same answer trie. Only when the computation fails from the last (oldest) interior node should the corresponding subgoal be made inactive. To correctly implement that we use the trail stack. The call that first executes code for a completed subgoal changes the subgoal's state to complete-active and stores in the trail stack the reference to the subgoal frame. Further calls to the same subgoal (cases where the subgoal's state is now complete-active) are handled as before. Figure 6 shows how we extended the `tabled_subgoal_call()` instruction to support this.

```
tabled_subgoal_call(subgoal SG) {
  sg_fr = search_table_space(SG)     // sg_fr is the subgoal frame for SG
  if (SgFr_state(sg_fr) == ready) {
    remove_from_inactive_list(sg_fr)                                  // new
    ...
  } else if (SgFr_state(sg_fr) == evaluating) {
    ...
  } else if (SgFr_state(sg_fr) == complete) {
    remove_from_inactive_list(sg_fr)                                  // new
    SgFr_state(sg_fr) = complete-active                               // new
    trail(sg_fr)                                                      // new
    goto SgFr_answers(sg_fr)       // execute compiled code from the trie
  } else if (SgFr_state(sg_fr) == complete-active) {         // new state
    goto SgFr_answers(sg_fr)       // execute compiled code from the trie
  } else if (SgFr_state(sg_fr) == incomplete) {
    remove_from_inactive_list(sg_fr)                                  // new
    ...
  }
}
```

**Fig. 6.** Extended pseudo-code for `tabled_subgoal_call()`

When later backtracking occurs, we use the reference in the trail stack to correctly insert the subgoal in the list of inactive subgoals. This use of the trail

stack does not introduce any overhead because the YapTab engine already uses the trail to store information beyond the normal variable trailing (to control dynamic predicates, multi-assignment variables and frozen segments).

### 4.2 Discussion

With this dynamic recovery mechanism, the programmer can now rely on the effectiveness of the memory management algorithm to completely avoid the problem of deciding what potentially useless tables should be deleted. Note, however, that we can still increase the table memory space arbitrarily. This can happen if the space required by the set of active subgoals exceeds the available memory space and we are not able to recover any space from the set of inactive subgoals. A possible solution for this problem is to store data externally using, for example, a database management system. We are already studying how this can be done, that is, how we can partially move tables to database storage and efficiently load them back to the tabling engine. This idea can also be applied to inactive subgoals and, in particular, we can eventually use our memory management algorithm, not to decide what tables to delete but, to decide what tables to move to the database.

## 5 Experimental Results

To evaluate the impact of our proposals, we ran the April ILP system [15] with YapTab. The environment for our experiments was a Pentium M 1600MHz processor with 1 GByte of main memory and running the Linux kernel 2.6.11.

We first experimented our support to incomplete tabling and, for that, we used a well-known ILP data-set, the *Mutagenesis* data-set, with two different configurations that we named *Mutagen1* and *Mutagen2*. The main difference between the configurations is that the hypotheses space is searched differently. Table 1 shows the running times, in seconds, for *Mutagen1* and *Mutagen2* using four different approaches to evaluate the predicates in the background knowledge: **(i)** without tabling; **(ii)** using local scheduling; **(iii)** using batched scheduling; and **(iv)** using batched scheduling with support for incomplete tabling. The running times include the time to run the whole ILP system. During evaluation, *Mutagen1* and *Mutagen2* call respectively 1479 and 1461 different tabled subgoals and, for batched scheduling, both end with 76 incomplete tables.

Our results show that, by combining batched scheduling with incomplete tabling, we can further speed up the execution for these kind of problems. Batched scheduling allows us to favor forward execution and incomplete tabling allows us to avoid re-computation. However, for some subgoals, local scheduling can be better than batched scheduling with incomplete tabling. We can benefit from local scheduling when the cost of fully generating the complete set of answers is less than the cost of evaluating the subgoal several times as a result of several pruning operations. Better results are thus still possible if we use YapTab's flexibility that allows to intermix batched with local scheduling

within the same evaluation. However, from the programmer point of view, it is very difficult to define the subgoals to table using one or another strategy. We thus argue that our combination of batched scheduling with incomplete tabling is an excellent (and perhaps the best) compromise between simplicity and good performance.

| Tabling Mode | *Mutagen1* | *Mutagen2* |
|---|---|---|
| Without tabling | > 1 day | > 1 day |
| Local scheduling | 153.9 | 143.3 |
| Batched scheduling | 278.2 | 137.9 |
| Batched scheduling with incomplete tabling | 122.9 | 117.6 |

**Table 1.** Running times, in seconds, with and without support for incomplete tabling

We next show how we used another well-known ILP data-set, the *Carcinogenesis* data-set, to experiment with our second proposal. From our previous work on tabling conjunctions of subgoals, we selected one of the hypotheses that allocates more memory when computing its coverage against the set of examples in the *Carcinogenesis* data-set. That hypothesis is defined by a prefix that represents the conjunction of 5 tabled subgoals with a total of 20 arguments. Table 2 shows the running times in seconds (or *m.o.* for memory overflow) for computing its coverage with four different table limit sizes: 576, 384, 192 and 128 MBytes (the table limit size is defined statically when the system starts). In parentheses, it shows the number of executions of the `recover_space()` procedure.

| Tabling Mode | 576MB | 384MB | 192MB | 128MB |
|---|---|---|---|---|
| Local scheduling | 15.2 | 15.9(95) | 16.9(902) | *m.o.*(893) |
| Batched scheduling | 11.4 | 12.6(62) | 14.1(523) | *m.o.*(557) |
| Batched scheduling with incomplete tabling | 11.1 | 12.3(91) | 13.9(833) | *m.o.*(833) |

**Table 2.** Running times, in seconds, with different table limit sizes

Through experimentation, we found that this computation requires a total table space of 576 MBytes if not recovering any space, and a minimum of 160 MBytes if using our recovery mechanism (for Pentium-based architectures, YapTab allocates memory in segments of 32 MBytes). The results obtained with this particular example show that batched scheduling with incomplete tabling is again the best approach. The results also suggest that our recovery mechanism is quite effective in performing its task (for a memory reduction of 66% in table space it introduces an average overhead between 10% and 20% in the execution time). The impact of our proposal in the execution time depends, in general, on the size of the table space and on the specificity of the application being evaluated, i.e., on the number of times it may call subgoals whose tables were previously deleted by the recovery procedure.

## 6 Conclusions

In this paper, we have discussed some practical deficiencies of current tabling systems when dealing with incomplete tabling and memory recovery. Incomplete

tabling became a problem when, as a result of a pruning operation, the computational state of a tabled subgoal is removed from the execution stacks before being completed. On the other hand, memory recovery became a problem when we use tabling for applications that build very many queries or that store a huge number of answers, quickly running out of memory space.

To support incomplete tabling, we have proposed the ability to avoid recomputation by keeping incomplete tables for pruned subgoals. The typical case where we can profit from having incomplete tables is, thus, when the already stored answers are enough to evaluate repeated calls. When this is not the case, we cannot benefit from it but, on the other hand, we do not pay any cost either. To recover memory, we have proposed a memory management strategy that automatically recovers space from inactive tables when the system runs out of memory. Both proposals have been implemented in the YapTab tabling system with minor changes to the original design. To the best of our knowledge, YapTab is the first tabling system that implements support to incomplete tabling and memory recovery as discussed above. Preliminary results using the April ILP system showed very substantial performance gains and a substantial increase of the size of the problems that can be solved by combining ILP with tabling.

## Acknowledgments

We are very thankful to Nuno Fonseca for his support with the April ILP System. This work has been partially supported by Myddas (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC.

## References

1. Tamaki, H., Sato, T.: OLDT Resolution with Tabulation. In: International Conference on Logic Programming. Number 225 in LNCS, Springer-Verlag (1986) 84–98
2. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43** (1996) 20–74
3. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems **20** (1998) 586–634
4. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction. (2000) 77–87
5. Demoen, B., Sagonas, K.: CHAT: The Copy-Hybrid Approach to Tabling. Future Generation Computer Systems **16** (2000) 809–830
6. Guo, H.F., Gupta, G.: A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In: International Conference on Logic Programming. Number 2237 in LNCS, Springer-Verlag (2001) 181–196
7. Zhou, N.F., Shen, Y.D., Yuan, L.Y., You, J.H.: Implementation of a Linear Tabling Mechanism. Journal of Functional and Logic Programming **2001** (2001)

8. Somogyi, Z., Sagonas, K.: Tabling in Mercury: Design and Implementation. In: International Symposium on Practical Aspects of Declarative Languages. Number 3819 in LNCS, Springer-Verlag (2006) 150–167
9. Sagonas, K., Stuckey, P.: Just Enough Tabling. In: ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, ACM (2004) 78–89
10. Saha, D., Ramakrishnan, C.R.: Incremental Evaluation of Tabled Logic Programs. In: International Conference on Logic Programming. Number 3668 in LNCS, Springer-Verlag (2005) 235–249
11. Rocha, R., Silva, F., Santos Costa, V.: Dynamic Mixed-Strategy Evaluation of Tabled Logic Programs. In: International Conference on Logic Programming. Number 3668 in LNCS, Springer-Verlag (2005) 250–264
12. Muggleton, S.: Inductive Logic Programming. In: Conference on Algorithmic Learning Theory, Ohmsma (1990) 43–62
13. Rocha, R., Fonseca, N., Santos Costa, V.: On Applying Tabling to Inductive Logic Programming. In: European Conference on Machine Learning. Number 3720 in LNAI, Springer-Verlag (2005) 707–714
14. Rocha, R., Silva, F., Santos Costa, V.: On applying or-parallelism and tabling to logic programs. Journal of Theory and Practice of Logic Programming **5** (2005) 161–205
15. Fonseca, N.A., Silva, F., Camacho, R.: April - An Inductive Logic Programming System. In: European Conference on Logics in Artificial Intelligence. Number 4160 in LNAI, Springer-Verlag (2006) 481–484
16. Chen, W., Swift, T., Warren, D.S.: Efficient Top-Down Computation of Queries under the Well-Founded Semantics. Journal of Logic Programming **24** (1995) 161–199
17. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38** (1999) 31–54
18. Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Ramon, J., Vandecasteele, H.: Improving the Efficiency of Inductive Logic Programming Through the Use of Query Packs. Journal of Artificial Intelligence Research **16** (2002) 135–166