Efficient Support for Incomplete and Complete Tables in the YapTab Tabling System

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

CICLOPS 2006, Seattle, Washington, USA, August 2006

Motivation

- This work was motivated by our recent attempt of **applying tabling to Inductive Logic Programming (ILP)** [Rocha *et al.*, ECML'05].
- ILP applications are an excellent case study for tabling because they have **huge search spaces** and do a **lot of re-computation**.
- In particular, in this work we focus on the table space and how to **efficiently handle incomplete and complete tables**.

Tabling and ILP

- Tabling is about storing answers for subgoals so that they can be reused when a repeated call appears.
- On the other hand, ILP systems are interested in evaluating hypotheses, and not in finding answers for goals. This is usually implemented by **pruning** at the Prolog level.

Tabling and ILP

- Tabling is about storing answers for subgoals so that they can be reused when a repeated call appears.
- On the other hand, ILP systems are interested in evaluating hypotheses, and not in finding answers for goals. This is usually implemented by **pruning** at the Prolog level.
- For instance, to evaluate if the hypothesis

  theory(X):- a1(X), a2(X,Y), a3(Y).

  covers the example **theory(p1)** an ILP system executes the goal

  once(a1(p1), a2(p1,Y), a3(Y)).

- The **once/1** primitive prunes over the search space preventing the unnecessary search for further answers. It is usually defined as

  once(Goal):- call(Goal), !.

Tabling and ILP: Incomplete Tables

- Consider now that **a2/2** is a tabled predicate and that our goal succeeds

  once(a1(p1), a2(p1,Y), a3(Y)).

  a2(p1,Y) will be removed from the execution stacks before being completed.

Tabling and ILP: Incomplete Tables

- Consider now that **a2/2** is a tabled predicate and that our goal succeeds

  once(a1(p1), a2(p1,Y), a3(Y)).

  a2(p1,Y) will be removed from the execution stacks before being completed.
- Thus, when a repeated call to **a2(p1,Y)** appears, we cannot simply trust the answers from its table, because we may loose part of the computation.

Tabling and ILP: Incomplete Tables

- Consider now that **a2/2** is a tabled predicate and that our goal succeeds

  once(a1(p1), a2(p1,Y), a3(Y)).

  a2(p1,Y) will be removed from the execution stacks before being completed.
- Thus, when a repeated call to **a2(p1,Y)** appears, we cannot simply trust the answers from its table, because we may loose part of the computation.
- A common approach is to **throw away** incomplete tables and restart the evaluation from the beginning when a repeated call appears.

Tabling and ILP: Incomplete Tables

- How can we make tabling worthwhile in an environment that potentially generates so many incomplete tables?

Tabling and ILP: Incomplete Tables

- How can we make tabling worthwhile in an environment that potentially generates so many incomplete tables?
- We first studied this problem by using YapTab's functionality that allows to combine **batched with local scheduling** [Rocha *et al.*, ICLP'05].

  Our results showed best performance when we evaluated some subgoals using batched scheduling and others using local scheduling.

  The problem is that from the programmer's point of view it is very difficult to define beforehand the subgoals to table using one or another strategy.

Incomplete Tables: Our Approach

- **Main Goals**
  - **Reuse the already found answers** in order to avoid re-computation.
  - **Favor forward execution** in order to quickly succeed with the evaluation of the hypotheses.

Incomplete Tables: Our Approach

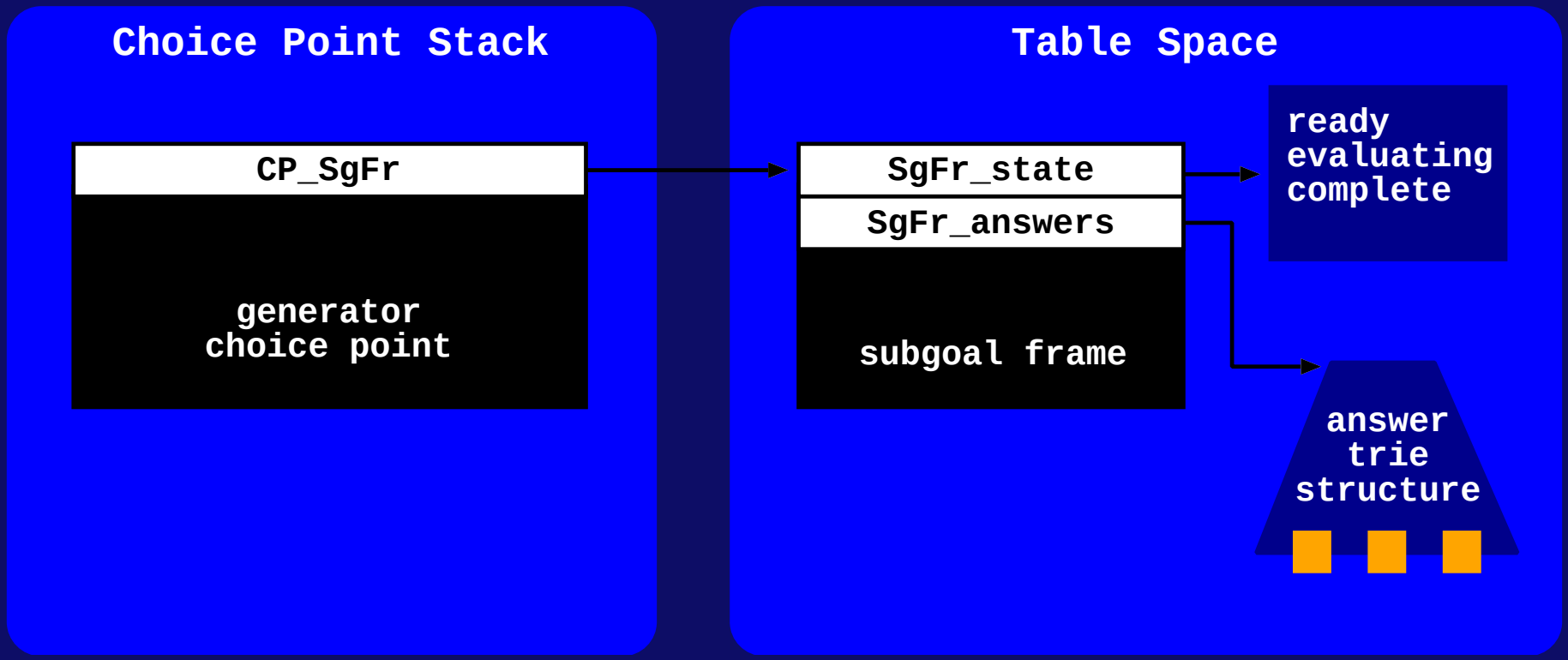
Main Goals

- **Reuse the already found answers** in order to avoid re-computation.
- **Favor forward execution** in order to quickly succeed with the evaluation of the hypotheses.

Basic Idea

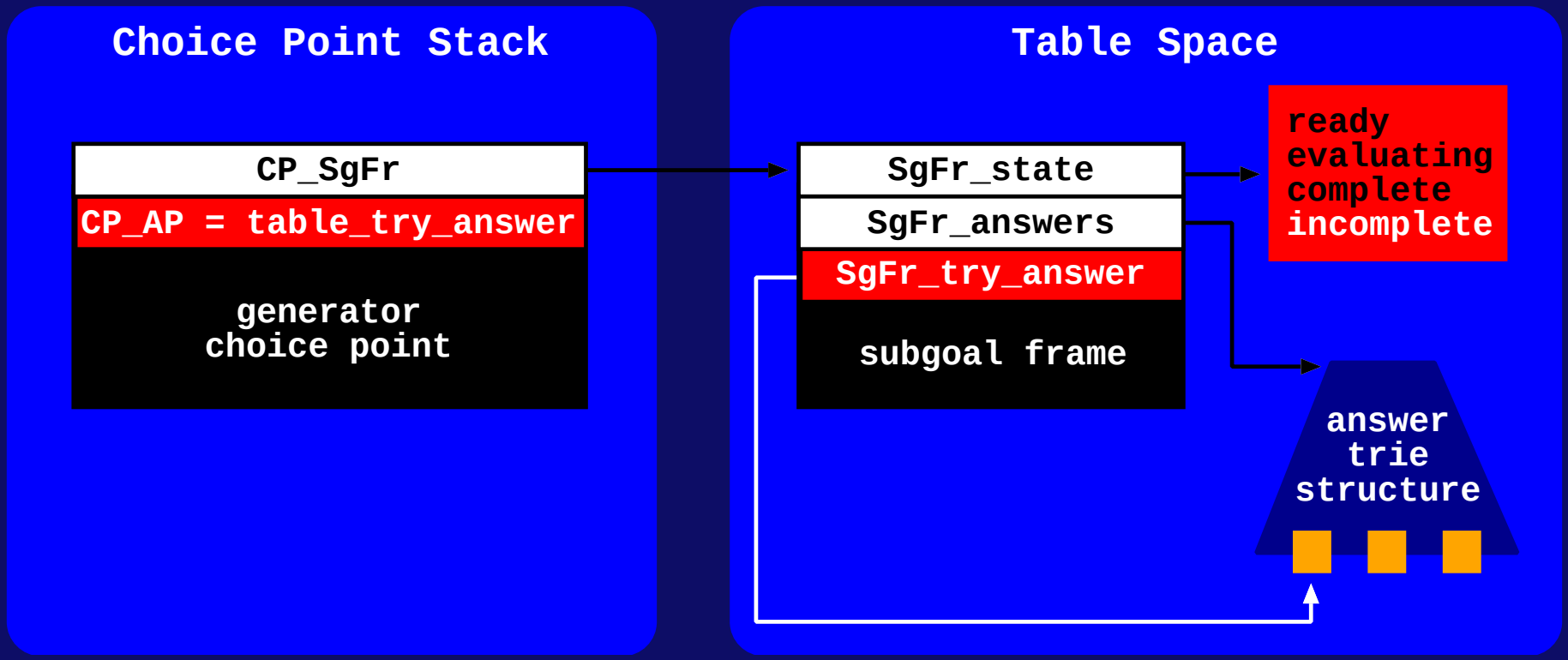
- By default, we keep incomplete tables for pruned subgoals.
- Then, when a repeated call appears, we start by consuming the available answers from its incomplete table.
- If the table is exhausted, then we restart the evaluation from the beginning.
- Later, if the subgoal is pruned again, then the same process is repeated until eventually the subgoal be completely evaluated.

Incomplete Tables: Implementation



- **YapTab's Original Design**
  - The **CP_SgFr** field points to the corresponding subgoal frame.
  - The **SgFr_state** field indicates the state of the subgoal.
  - The **SgFr_answers** field points to where answers are stored.

Incomplete Tables: Implementation



- **YapTab's Extensions**
  - A new **table_try_answer** pseudo-instruction.
  - A new **incomplete** state.
  - A new **SgFr_try_answer** field marks the currently loaded answer.

Incomplete Tables: Implementation

```
tabled_subgoal_call(subgoal SG) {
  sg_fr = search_table_space(SG)          // get subgoal frame for SG
  if (SgFr_state(sg_fr) == ready) {
    ...
  } else if (SgFr_state(sg_fr) == evaluating) {
    ...
  } else if (SgFr_state(sg_fr) == complete) {
    ...
  } else if (SgFr_state(sg_fr) == incomplete) {          // new block
    gen_cp = store_generator_node(sg_fr)
    CP_AP(gen_cp) = table_try_answer       // new pseudo-instruction
    SgFr_state(sg_fr) = evaluating
    first = get_first_answer(sg_fr)
    load_answer(first)
    SgFr_try_answer(sg_fr) = first         // mark the loaded answer
  }
}
```

Incomplete Tables: Implementation

```
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(next)
    SgFr_try_answer(sg_fr) = next      // update the loaded answer
  } else {          // restart the evaluation from the first clause
    PREG = get_wam_code(sg_fr)       // PREG is the program counter
    CP_AP(GEN) = failure_continuation(PREG)        // second clause
  }
}
```

Incomplete Tables: Discussion

- Now assume that **a2(p1,Y)** is called again when evaluating a different goal **once(a2(p1,Y), a4(Y)).**

Incomplete Tables: Discussion

- Now assume that **a2(p1,Y)** is called again when evaluating a different goal

  once(a2(p1,Y), a4(Y)).

- If **a4(Y)** succeeds with one of the previously found answers for **a2(p1,Y)**, then we take advantage of having maintained the incomplete table for **a2(p1,Y)**.

Incomplete Tables: Discussion

- Now assume that **a2(p1,Y)** is called again when evaluating a different goal

  once(a2(p1,Y), a4(Y)).

- If **a4(Y)** succeeds with one of the previously found answers for **a2(p1,Y)**, then we take advantage of having maintained the incomplete table for **a2(p1,Y)**.
- Otherwise, **a2(p1,Y)** will be reevaluated as a first call. This means that the evaluation will fail for **a2(p1,Y)** until a non-repeated answer is eventually found.

  We may not benefit from having maintained the incomplete table, but we do not pay any cost either, because the computation time required to evaluate the goal, with or without the incomplete table, is equivalent.

Tabling and ILP: Complete Tables

- When we use tabling for applications that build **very many or very large tables**, we can quickly **run out of memory**.
- A common approach is to have a set of 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.

Complete Tables: Our Approach

- **Basic Idea**
  - 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.

Complete Tables: Implementation

- **Active/Inactive Tabled Subgoals**
  - A tabled subgoal is said to be **active** if it is represented in the execution stacks.
  - Otherwise, it is said to be **inactive**. Inactive subgoals are only represented in the table space.

Complete Tables: Implementation

- **Active/Inactive Tabled Subgoals**
  - A tabled subgoal is said to be **active** if it is represented in the execution stacks.
  - Otherwise, it is said to be **inactive**. Inactive subgoals are only represented in the table space.
- Knowing what subgoals are active or inactive is important when the system runs out of memory.
  - We should try to recover space from the inactive subgoals.

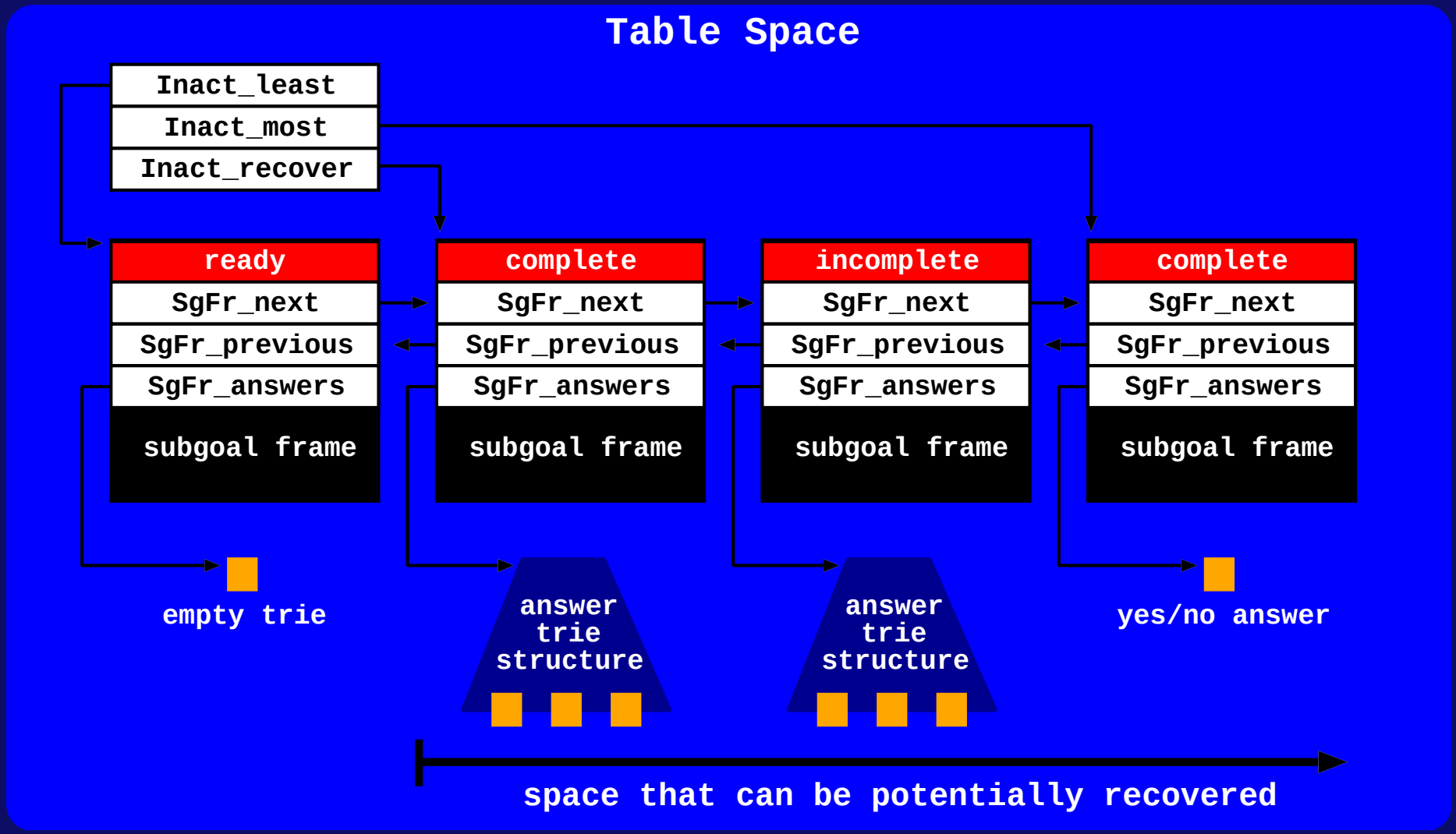
Complete Tables: Implementation

- **Subgoal's States**
  - Ready → Inactive
  - Evaluating → Active
  - **Complete → Active/Inactive**
  - Incomplete → Inactive

Complete Tables: Implementation

- **Subgoal's States**
  - Ready → Inactive
  - Evaluating → Active
  - **Complete → Active/Inactive**
  - Incomplete → Inactive
- **YapTab's Extension**
  - **Complete → Inactive**
  - **Complete-Active → Active**
- With this simple extension, we can use the **SgFr_state** field of the subgoal frames to decide if a subgoal is active or inactive.

Complete Tables: Implementation



Complete Tables: Implementation

- **Inactive → Active**
  - We execute a first call to a non-completed subgoal (→ **evaluating**).
  - We execute a first call to a completed subgoal (→ **complete-active**).
- **Active → Inactive**
  - The subgoal completes (→ **complete**).
  - The subgoal is pruned (→ **incomplete**).
  - We have consumed all answers from a completed subgoal and there is no other node consuming answers from it (→ **complete**). To implement that we use the **trail stack**.

Preliminary Results

Tabling Mode	Config1	Config2
Without tabling	$>$ 1 day	$>$ 1 day
Local scheduling	153.9	143.3
Batched scheduling	278.2	137.9
Batched scheduling with incomplete tables	**122.9**	**117.6**

Running times (in seconds) for the Mutagenesis data-set

- The running times include the time to run the whole ILP system.
- Config1 and Config2 call respectively 1479 and 1461 different tabled subgoals and, for batched scheduling, both end with 76 incomplete tables.

Preliminary Results

Tabling Mode	576MB	384MB	192MB
Local scheduling	15.2	15.9(95)	16.9(902)
Batched scheduling	11.4	12.6(62)	14.1(523)
Batched scheduling with incomplete tables	**11.1**	**12.3(91)**	**13.9(833)**

Running times and number of recovering operations for the Carcinogenesis data-set

- This data-set requires a total table space of 576 MBytes if not recovering any space, and a minimum of 160 MBytes if using our recovering mechanism.
- For a memory reduction of 66% in table space, our recovering mechanism introduces an average overhead between 10% and 20% in the execution time.

Conclusions

- We have discussed some practical deficiencies of current tabling systems when dealing with incomplete and complete tables.
- Our proposals have been implemented in the YapTab tabling system with minor changes to the original design.
- Preliminaries 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.
- The problems and proposals presented in this work are not restricted to ILP applications and can be generalised and applied to any other application.