

**An Execution Mechanism for
Combining Query Packs and
Once-Transformations**

Remko Tronçon

Henk Vandecasteele

Jan Struyf

Bart Demoen

Gerda Janssens

Report CW 362, October 2003

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

An Execution Mechanism for Combining Query Packs and Once-Transformations

Remko Tronçon

Henk Vandecasteele

Jan Struyf

Bart Demoen

Gerda Janssens

Report CW 362, October 2003

Department of Computer Science, K.U.Leuven

Abstract

In Inductive Logic Programming (ILP), several techniques have been introduced to improve the efficiency of query execution. One such technique is query pack execution. A set of queries with a common prefix, as it is generated by the refinement operator of a typical ILP system, can be executed faster after it is converted into a tree structure called a query pack. On the other hand, query transformations improve the efficiency of executing a single query by transforming it into a different form that is more efficient to execute. Combining query packs with query transformations is difficult because a transformation may have a negative effect on the structure of the pack. The once-transformation is an important query transformation, and can improve the efficiency of query execution by several orders of magnitude. In this work, we extend query pack execution in such a way that it is able to handle queries produced by the once-transformation. We do this in the context of ilProlog, a high performance Prolog system with specific extensions for supporting ILP systems. We evaluate our approach on both artificial domains and real world ILP applications.

An Execution Mechanism for Combining Query Packs and Once-Transformations

Remko Tronçon Henk Vandecasteele Jan Struyf Bart Demoen
Gerda Janssens
{remko, henkv, jan, bmd, gerda}@cs.kuleuven.ac.be
Dept. of Computer Science, K.U. Leuven, Belgium

Abstract

In Inductive Logic Programming (ILP), several techniques have been introduced to improve the efficiency of query execution. One such technique is query pack execution. A set of queries with a common prefix, as it is generated by the refinement operator of a typical ILP system, can be executed faster after it is converted into a tree structure called a query pack. On the other hand, query transformations improve the efficiency of executing a single query by transforming it into a different form that is more efficient to execute. Combining query packs with query transformations is difficult because a transformation may have a negative effect on the structure of the pack. The once-transformation is an important query transformation, and can improve the efficiency of query execution by several orders of magnitude. In this work, we extend query pack execution in such a way that it is able to handle queries produced by the once-transformation. We do this in the context of *ilProlog*, a high performance Prolog system with specific extensions for supporting ILP systems. We evaluate our approach on both artificial domains and real world ILP applications.

1 Introduction

Efficiency is an important issue in machine learning and data mining. This is especially true for Inductive Logic Programming (ILP) and relational data mining systems, which induce models based on input data that is stored in a relational format. Because the data is represented in a more expressive form, induction becomes more complex: larger spaces have to be searched, and evaluating the performance of a candidate hypothesis is computationally more expensive. In order to cope with this increased complexity, several techniques have been introduced that make the induction task more efficient ([4] presents an overview).

One such technique is *query pack* execution [3]. In ILP, each hypothesis is represented by a query in first order logic. The set of all possible queries in the hypothesis language can be structured as a lattice and finding a suitable query corresponds to searching through the lattice. Most ILP algorithms search from general to specific: in each node of the search, the current query is refined into several candidate queries by adding one or more goals. The performance of each candidate must be evaluated on the input data, i.e. each candidate query must be executed on each example. Because the candidates are similar (they all share the current query as common prefix) there will be much redundancy in this evaluation step. Structuring the set of queries in a tree structure (a query pack) is a way to remove this redundancy. In practical applications, query packs can speedup induction by an order of magnitude or more.

Transforming queries [6, 10] is another approach for improving the efficiency of first order induction. A query-transformation replaces a query by a form that is more efficient to execute. As an example, the *once-transformation* recursively wraps sets of goals that are independent (given that certain variables are ground) in *onces*. A *once* is a special construct that prunes away unnecessary computations during query

execution. For some ILP applications the once-transformation can make query execution several orders of magnitude faster [6].

Combining query packs with query transformations is difficult because the transformation has a negative effect on the structure of the pack. Since the transformation alters the form of some queries (e.g. by reordering goals or by introducing onces), it will be difficult to find a common part that can be exploited by using query pack execution. In this work, we extend query pack execution in such a way that it is able to handle queries produced by the once-transformation. This is accomplished by inserting special control instructions, which we will call *activate* and *deactivate*, in the pack. A pack that contains these instructions will be called an *activate/deactivate-pack* or *adpack* for short. With some preliminary experiments, we show that adpack execution performs better than using either only the once-transformation or only the query packs.

This paper is organized as follows. In Section 2 we briefly review the once-transformation and query pack execution, and describe the intuition behind adpacks. In Section 3, we illustrate this intuition with a small example. Section 4 gives the formal definition of an adpack. Section 5 describes how an adpack can be constructed based on a set of once-transformed queries. In Section 6 we define how an adpack can be executed on a training example. This is illustrated with an elaborated example in Section 7. In Section 8 an implementation of adpack execution in a Prolog engine is discussed. Section 9 presents preliminary experiments on both artificial and real world data sets. Finally, in Section 10 we summarize the main conclusions and future work.

2 Intuition

In this section we describe the once-transformation, query packs and adpacks, which generalize both the once-transformation and query packs. We don't give formal definitions. Instead we try to explain the intuition behind each technique.

Consider a set of queries representing the hypotheses that have to be evaluated during an inductive learning process. We want to optimize the execution of this set (where a query is actually a conjunction of atoms or predicate calls).

A first observation is that we want to find out whether a query succeeds or fails. Thus, once the query has succeeded, the execution does not have to backtrack to any of the unexplored alternatives for the calls of the query. This can be realized easily by adding a cut at the end of every query in the set. Figure 1a represents such a set of cut-terminated queries (with the arguments of the goals left out for the sake of simplicity).

Secondly, as these queries are hypotheses generated in a systematic way by the inductive learner, the queries begin with the same predicate calls, but they have different endings. To avoid the repeated execution of the common part, the notion of *query packs* [3] with their own execution mechanism was introduced. The common prefix of the conjunction is executed once, and a so-called pack-or node groups the alternative endings. The scope of the terminating cut goes only up to the closest pack-or node: the other alternatives of the pack-or node might still require backtracking into the upper part. Once a query succeeds and its terminating cut is executed, we try to remove this query from the pack. However, since parts of the query might be shared with other (still unsuccessful) queries, initially only the part up to the closest pack-or node will be removed. Later, when all the sibling alternatives in that pack-or have also succeeded, the removal propagates upwards to the parent pack-or, eventually arriving in the root pack-or. In other words: the cut is blocked in the pack-or until all other branches in the pack-or are successful. Transforming the queries from Figure 1a results in the query pack of Figure 1b. We omit the terminating cut at the end of each clause in this representation to avoid confusion with a real cut.

A third observation is that a conjunction of predicate calls can have parts whose execution does not affect the execution of the remaining part. For example, consider the first query of Figure 1a, this time with

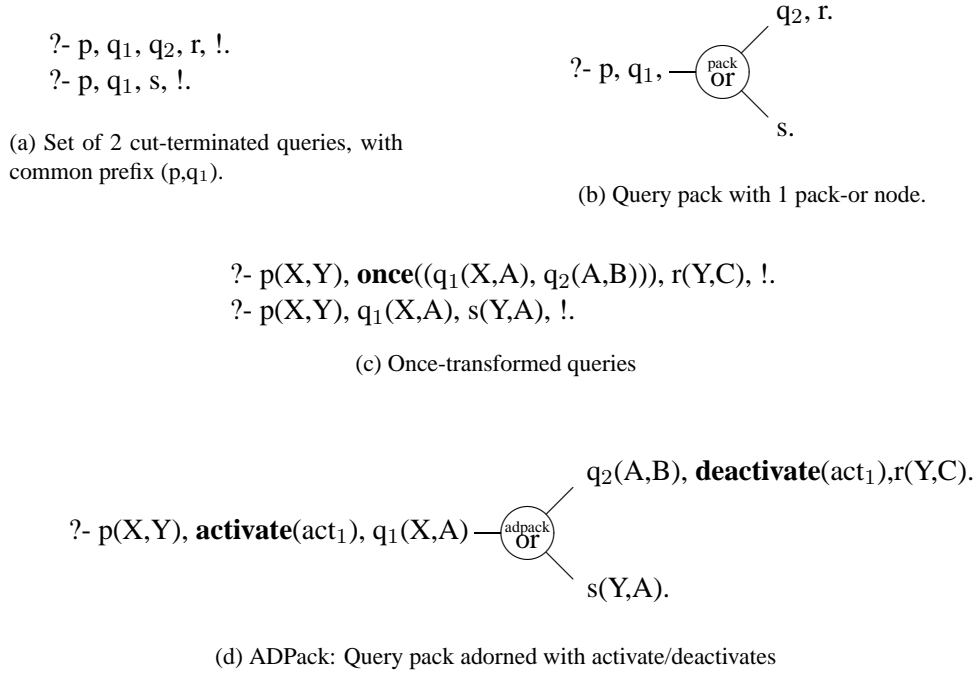


Figure 1: Different transformations on a set of queries

arguments to the goals:

$$?- p(X,Y), q_1(X,A), q_2(A,B), r(Y,C), !.$$

Suppose that the call $p(X,Y)$ grounds both the variables X and Y , then the q_i calls do not affect the r call. In other words, if the call to $r(Y,C)$ fails, we should not backtrack to a q_i call, but to $p(X,Y)$. When $(q_1(X,A), q_2(A,B))$ succeeds, all backtracking over the q_i calls should be avoided until the execution backtracks to $p(X,Y)$, as then new values for X and Y can be found. We want a cut with a limited scope: this can be obtained by the *once* construct. The above query becomes:

$$?- p(X,Y), \mathbf{once}((q_1(X,A), q_2(A,B))), r(Y,C), !.$$

The argument of the *once* is a conjunction of calls (possibly nested *onces*), such that when the last call of the conjunction succeeds, all the alternatives for calls in this conjunction are removed. This transformation is called *once-transformation*, and is discussed in depth in [6]¹. Query transformations such as these are orthogonal to query packs.

The contribution of this paper is that we propose a new execution mechanism that supports the combination of the query packs with the *once-transformation* in order to minimize the amount of backtracking as much as possible. Therefore we introduce the notion of an *adpack*. Consider the *once-transformed* queries of Figure 1c, where both queries have the common prefix $(p(X,Y), q_1(X,A))$, but only the first one has an additional *once*. The *adpack* is obtained by first constructing a query pack as before without taking into account the *onces*, and next indicating the scope of the *once* by using corresponding *activate* and *deactivate*

¹Note that the *once-transformation* uses information about how the predicates ground their arguments. In our experiments, all calls ground all their arguments, so we assume this scenario throughout this paper.

p(a,b).	q ₁ (a,d).	q ₁ (a,f).	q ₂ (d,e).	q ₂ (e,e).	s(b,e).
p(a,c).	q ₁ (a,e).	q ₁ (b,d).	q ₂ (d,f).	r(c,f).	

Figure 2: An example fact database.

goals (in Section 5 we describe a slightly different approach, which is suitable for a once-transformation that reorders goals). This results in the adpack shown in Figure 1d. A deactivate goal indicates that all alternatives up to the corresponding activate goal are not relevant for finding solutions for the remainder of the branch. The adpack-or node again blocks terminating cuts until all siblings have success. It also blocks the once as long as backtracking into the scope of the once is needed for other siblings. Consider Figure 1d. The deactivate for $\text{once}((q_1(X,A), q_2(A,B)))$ first only removes alternatives for q_2 as the other branch with s still needs to be able to backtrack to q_1 . When r fails, we backtrack to the closest adpack-or node. As long as s fails, we will further explore the alternatives for q_1 , but without considering the q_2, r ending due to the blocked once. Note that it is as if the q_2, r branch is removed from the pack. This removal is not definite and will be undone as soon as the execution backtracks to a call before the once. When s succeeds, its terminating cut is blocked by the q_2, r branch, but the once can now have its full scope and the remaining alternatives for q_1 will be removed. When we finally backtrack to p , the q_1, q_2, r branch will again be executed (and the temporary removal is abolished).

To summarize we can say that a terminating cut permanently removes a branch from the adpack, whereas a deactivate only temporarily removes a branch from the adpack.

3 Example

We will now illustrate in more detail the execution of adpacks on a small example. Consider the adpack from Figure 1d, and suppose we want to apply it to the fact database seen in Figure 2.

Forward execution starts by executing $p(X, Y)$, thus binding X to a and Y to b . We come across the activate goal indicating the start of the once from the first query, arriving in $q_1(X, A)$ which binds A to d . Now, the adpack splits off in 2 branches; we select the first branch (corresponding to the rest of the first query) and leave behind a choicepoint. After $q_2(A, B)$ binds B to e , the deactivate delimiting the once from the first query is encountered. This means that, for the rest of the query, it is unnecessary to backtrack over all goals between the activate and the deactivate, which are exactly the goals of the once from the first query. This would mean that we can cut away all choicepoints of these goals. However, since $q_1(X, A)$ is a shared goal between the first and the second query, we cannot cut away its choicepoints, since they can be relevant for the second query. Therefore, we only cut away the choicepoints of the goals up to the adpack-or node. A second thing that has to be done is reassure that whenever execution chooses an alternative for $q_1(X, A)$, it should not enter this branch (since $q_1(X, A)$ is part of the once, and its alternatives are not relevant for this branch). Therefore, we mark the branch as ‘closed’ (or in other words, we temporarily remove the branch from the adpack). The last goal $r(Y, C)$ has no solution, so this query fails and backtracks to the last choicepoint, which is the one of the adpack-or. The second branch is now chosen, which fails immediately.

Back in the adpack-or, all the branches have been unsuccessfully tried, so we have to backtrack to a choicepoint. Since $q_1(X, A)$ still has alternatives, and since these alternatives can be relevant for the second branch, we backtrack to $q_1(X, A)$. Forward execution restarts at $q_1(X, A)$, now binding A to e , and arriving in the adpack-or. Since the first branch is still closed (indeed, we have not yet backtracked outside the once, so any alternative chosen is not relevant for this branch), we can only enter the second branch. This time, $s(Y, A)$ succeeds, and thus this branch is marked as successful (meaning that the second query was successful).

After the success of the second branch, execution backtracks to the adpack-or node, and since all branches have been tried and there are still unsuccessful ones, we again have to backtrack. Normally execution would backtrack to the goal $q_1(X, A)$ which still has alternatives, but its alternatives are not relevant for any query of the adpack since the second branch is already successful, and the only choicepoints relevant for the first query are those before the activate (i.e. before the once). Our new execution mechanism avoids backtracking to $q_1(X, A)$, and backtracks directly to the most recent choicepoint before the once, namely $p(X, Y)$. After forward execution binds X to a and Y to c , the activate of the once from the first query is encountered. Since this means that execution has backtracked outside the once, execution should be allowed to enter the first branch of the adpack-or again, and so the first branch is opened again. The rest of the execution is straightforward: A is bound to d , the first branch is entered, B is bound to e , $r(Y, C)$ succeeds, and execution backtracks to the adpack-or. All branches of the adpack-or are successful, so any other backtracking is unnecessary, and the execution of the adpack finishes.

In this example, we can see where it pays off to have the activate/deactivates in the pack:

- When $r(Y, C)$ fails the first time, $q_2(A, B)$ would still have alternatives if the deactivate hadn't cut them away, and they would be tried in vain.
- When execution backtracked to $q_1(X, A)$, the first branch would have been executed again if the deactivate hadn't closed it, while this execution is known to be unsuccessful in advance.
- After the success of the second branch, execution would have normally backtracked to the last remaining alternative of $q_1(X, A)$, and then trying the first branch again. But adpack execution skipped this choicepoint since it was still inside the once, and was therefore not useful for the first branch.

4 Definition of an ADPack

An adpack is a tree defined as `<adpack>` in the following definition:

```

<adpack> := <subgoal> <adpack-or> | <subgoal>
<adpack-or> := 2 or more <adpack>
<subgoal> := <literal> | activate(<id>) | deactivate(<id>) |
            ( <subgoal> , <subgoal> )
<literal> is a Prolog-call
<id> is a natural number

```

The adpack should also satisfy the following restrictions:

- All activate/1 and deactivate/1 literals occur in pairs. Such a pair shares a unique identification number as their argument.
- For each activate/deactivate pair, both literals should always be a part of the same path from the root of the adpack to a leaf with the activate preceding the deactivate, but they cannot be located on the same branch (i.e. they should be separated by at least one adpack-or).
- If a deactivate a comes before a deactivate b on the path from the root of the adpack to a leaf, then the activate of a comes after the activate of b on that path.

```

function construct_adpack( $Q$ )
   $P := \text{empty\_adpack}$ 
  for each query  $q \in Q$ 
     $q_1 := \text{once-transform}(q)$ 
     $q_2 := \text{create\_adquery}(q_1)$ 
    merge_in_adpack( $q_2, P$ )
  post_process_adpack( $P$ )
  return  $P$ 

```

Figure 3: An algorithm for constructing an adpack, given a set of queries Q , produced by the refinement operator of the ILP system.

5 Construction of ADPacks

In this section, we show how an adpack can be constructed given a set of queries produced by the ILP system.

Constructing an adpack can be done as follows: iterate over the set of queries, transform each query with the once-transformation and merge the transformed query in an accumulator adpack. The `construct_adpack` algorithm (Figure 3) implements this idea. It takes as input a set of queries Q and starts with an empty adpack P . In each iteration of the main loop, it takes a query q from Q and transforms it using the once-transformation into q_1 . Then it converts q_1 to an intermediate structure q_2 and finally adds q_2 to the accumulator pack P . After all queries have been transformed and added to P , a post-processing step makes the adpack suitable for compilation and execution (Section 6).

We now describe each step in more detail. The algorithm starts with the first input query. It transforms this query using the once-transformation. We continue the example from the previous section. After the transformation step we obtain the following query.

$$?- p(X,Y), \text{once}(q_1(X,A), q_2(A,B)), r(Y,C).$$

The next step is to convert this query into an intermediate structure, which can later on be merged in the accumulator pack. The conversion step replaces each `once/1` with an `activate/deactivate` pair. An `activate` that precedes a certain goal is stored together with that goal. If the `onces` are nested, it is possible that several `activates` occur in a sequence. In that case the complete sequence is stored with the corresponding next goal. This idea of storing `activates` with a goal will make the merge step less complex. The conversion algorithm also assigns a unique identifier to each `activate/deactivate` pair. For the example query we obtain the following intermediate structure. Before each goal, we show the (possibly empty) set of preceding `activates`.

$$\{\}p(X,Y), \{\text{activate}(\text{act}_1)\}q_1(X,A), \{\}q_2(A,B), \{\text{deactivate}(\text{act}_1)\}, \{\}r(Y,C).$$

In a final step, the converted query is merged in the accumulator pack. The merge algorithm (Figure 4) searches for the largest possible prefix of the new query that can be shared with the other queries that are already in the pack. To implement this search, the algorithm uses two iterators i_1 and i_2 . Iterator i_1 points to a goal g_1 of the accumulator adpack and iterator i_2 points to a goal g_2 of the new query. Initially g_1 is the first goal of the pack and g_2 the first goal of the query. As long as g_1 and g_2 are equal, the two iterators are moved to the next goal. If the goals differ, a new adpack-or with two branches is created and remaining goals of i_1 are moved to the first branch and those of i_2 the second branch. If i_1 arrives at an adpack-or, the merge algorithm searches for a branch that starts with a goal equal to g_2 . If such a branch exists, i_1

```

function merge_in_adpack( $q, P$ )
   $i_1 = \text{iterator on } P, i_2 = \text{iterator on } q$ 
  while not at_end( $i_1$ ) do
     $g_2 = \text{first goal of } i_2$ 
    if at_adpack-or( $i_1$ ) then
      if  $b = \text{find_suitable_branch}(\text{adpack-or}(i_1), g_2)$  then  $i_1 = b$ 
      else  $i_1 = \text{create new branch in adpack-or}(i_1)$ 
    else
       $g_1 = \text{first goal of } i_1$ 
      if  $g_1 = g_2$  then
         $\text{activates}(g_1) = \text{activates}(g_1) \cup \text{activates}(g_2)$ 
         $\text{increment}(i_1), \text{increment}(i_2)$ 
      else if  $g_1 = \text{deactivate}(\text{ID1}) \wedge g_2 = \text{deactivate}(\text{ID2}) \wedge$ 
         $\text{activates\_at\_same\_goal}(\text{ID1}, \text{ID2})$  then
         $\text{remove\_from\_adpack}(\text{activate}(\text{ID2}), P)$ 
         $\text{increment}(i_1), \text{increment}(i_2)$ 
      else
         $\text{create adpack-or } o \text{ with two branches at position } i_1$ 
         $\text{append\_query}(\text{branch}(1, o), i_1)$ 
         $i_1 = \text{branch}(2, o)$ 
   $\text{append\_query}(i_1, i_2)$ 

```

Figure 4: An algorithm for merging a query in an adpack accumulator.

continues along that branch. If no suitable branch can be found, a new branch is added to the adpack-or and the remaining goals of i_2 are moved to that branch.

Merging the first query in an empty adpack is trivial: the adpack is overwritten by the new query. Consider an adpack that already contains our first example query and suppose we add the following query (continuing the example from Section 2).

$$\{\}p(X,Y), \{\}q_1(X,A), \{\}s(Y,A).$$

The merge_in_pack algorithm will follow the path through the adpack until it arrives at goal $g_1 = q_2(A,B)$ in the pack and goal $g_2 = s(Y,A)$ in the new query. These goals are different, so it creates a new adpack-or and moves $q_2(A,B)$ to the first branch and $s(Y,A)$ to the second one. The resulting adpack (after post-processing) is the one shown in Figure 1d.

We now describe how activates are handled. If g_1 and g_2 are equal, the algorithm adds the set of activates from g_2 to the set of activates stored with g_1 in the adpack. Inserting these new activates in the pack is allowed because they do not alter the execution of other queries from the pack that pass through g_2 . This is because an activate does not influence query-execution, it only opens a path and marks a useful backtrack point (Section 6).

If a deactivate is reached on the path through the adpack or on the new query, a new adpack-or is created (or a new branch is added to an existing one). Deactivates can not be inserted in an existing adpack because they change query-execution. If a deactivate would be inserted in a pack after a goal g , then alternatives for g would be pruned, which is incorrect for the other queries that pass through g . There is one exception to this rule: if a deactivate d_1 is reached in the adpack at the point where a deactivate d_2 occurs in the new

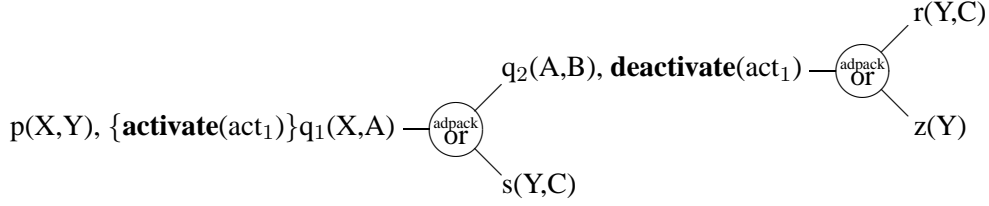


Figure 5: The resulting adpack before the post-processing step.

query. In this case the deactivate can be shared if the corresponding activates occur before the same (shared) goal². If this happens, the algorithm removes the activate that belongs to d_2 from the adpack and continues the merge process with the goals that come after d_1 and d_2 , i.e., it does not include the activate/deactivate pair from the new query because it can reuse an existing activate/deactivate pair from the pack. Suppose that we need to add the following query as third query to the adpack.

$$\{\}p(X,Y), \{\text{activate}(\text{act}_2)\}q_1(X,A), \{\}q_2(A,B), \text{deactivate}(\text{act}_2), \{\}z(Y).$$

The algorithm moves through the adpack until it arrives at the adpack-or. There it selects the first branch because that branch starts with $q_2(A,B)$, which also occurs in the new query. After that it arrives at $\text{deactivate}(\text{act}_1)$ in the pack and $\text{deactivate}(\text{act}_2)$ in the new query. Because activates act_1 and act_2 both occur before the same shared goal $q_1(X,A)$, the algorithm removes $\text{activate}(\text{act}_2)$ from the pack, shares $\text{deactivate}(\text{act}_1)$ with the new query and continues in the pack at $r(Y,C)$ and in the new query at $z(Y)$. Because these goals differ, it creates a new adpack-or with two branches. The resulting accumulator adpack is shown in Figure 5.

As said before, if iterator i_1 arrives at an adpack-or, it must select a suitable branch as a starting point for continuing the merge. At this point, the algorithm must also consider the case with two deactivates for which the corresponding activates are stored at the same goal, i.e., if g_2 is a deactivate then the algorithm will search for a branch that starts with a deactivate for which the corresponding activate is stored at the same goal as the activate that belongs to g_2 .

After all queries are merged in the pack, a final post-processing step is necessary. This post-processing step converts the accumulator adpack to an adpack that can be used as input for the compiler. Sequences of activates, which are stored at certain goals in the accumulator adpack, are inserted as regular activate/1 goals in the final adpack. Furthermore, activate/deactivate pairs that do not have an adpack-or between them are converted back into regular ones. Such a situation can occur, for example, if two queries have a common prefix containing a once are merged in an adpack. This also happened in the example above, but there it was not possible to create a regular once because the first goal after the activate was shared with a third query. Note that converting activate/deactivate pairs into ones can be tricky because the order of activates that occur before the same goal is lost in the adpack representation (because activates are stored in sets). A possible solution is to keep activates sorted such that activate act_1 comes before activate act_2 if $\text{offset}(\text{deactivate}(\text{act}_1)) > \text{offset}(\text{deactivate}(\text{act}_2))$, with $\text{offset}(g)$ the number of goals that precede g in the adpack.

Note that the adpack construction algorithm is general in the sense that it also can be used in combination with other query transformations. The only requirement is that the output of the transformation is a valid

²This can be implemented efficiently in Prolog by adding an identifier to each goal, represented by a variable. A deactivate gets the identifier of the goal that stores its corresponding activate. During the merge process, if two goals are equal, their identifiers are unified. If the goal-identifiers stored in two deactivates are equal, then the corresponding activates come before the same goal.

query, which may contain onces (or cuts, which can always be converted to onces). Of course the resulting adpack will only be efficient if the set of input queries have a common prefix. In that sense it maybe difficult to use adpacks in combination with a transformation that reorders goals. The most extreme example of such a transformation is the reordering transformation described in [10], but also the cut- and once-transformations reorder goals to some extent when they construct independent sets of goals. This makes it interesting to compare two versions of the once-transformation: one that reorders goals and one that does not.

6 Execution mechanism for ADPacks

We now describe the execution mechanism for adpacks in detail.

During the execution of an adpack, we will remember for each branch whether it is open or closed (temporarily disabled), and whether the branch is successful. Initially, every branch is open and unsuccessful.

The **forward execution** of an adpack consists in executing the goals on the branches as normal, except for the two special goals *activate* and *deactivate*. In these particular cases, the following actions have to be taken:

- **activate(*Id*)**

Let *DeactBranch* be the branch containing the deactivate corresponding to this activate (i.e. the deactivate with the same *Id*). If *DeactBranch* is still unsuccessful, do the following:

1. If this activate/deactivate pair has been deactivated before, reactivate it and open all branches on the path from the current branch to *DeactBranch*.
2. Remember the current choicepoint, and associate it with this activate.

Finally, continue execution.

- **deactivate(*Id*)**

Register deactivation of this activate/deactivate pair, cut away all choicepoints on this branch, close the current branch, and continue execution.

When execution reaches an adpack-or, the set of children that have to be executed is determined. This is exactly the set of child branches which are marked open, but are still unsuccessful. A child of this set is chosen, it is removed from the set, a choicepoint is created (which enables backtracking such that the remainder of the set can be executed), and the selected child is executed.

Finally, when the end of a query (a ‘leaf’ of the adpack) is reached, success for the current branch is registered, all its choicepoints are cut away, and execution backtracks to the parent adpack-or node.

Normally, when **backtracking** occurs, the only thing that has to be done is restoring a previous state by a.o. undoing bindings and to select the next alternative to be tried. However, when backtracking to an adpack-or, more complicated actions have to be taken. We distinguish the following 4 situations (in order) when backtracking to an adpack-or, with their corresponding actions:

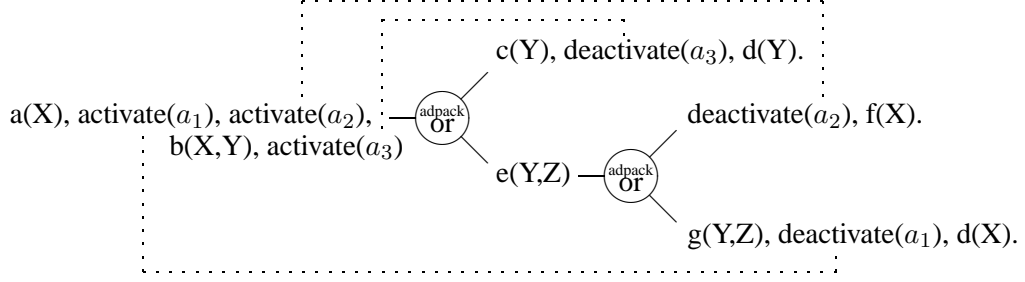
1. *There is still a child branch that has to be tried:* Remove the untried branch from the set of children which have to be tried, and execute it.
2. *All branches are successful:* Mark success of the parent branch, cut away all its choicepoints, and backtrack further.

?- a(X), b(X,Y), once(c(Y)), d(Y).
 ?- a(X), once((b(X,Y), e(Y,Z))), f(X).
 ?- a(X), once((b(X,Y), e(Y,Z), g(Y,Z))), d(X).

(a) Set of 3 once-transformed queries

a(1). a(2).
 b(1,1). b(1,2).
 b(2,1). c(1).
 c(2). d(1).
 e(1,1). e(2,1).
 f(2). g(2,1).

(b) Example program



(c) ADPack of the 3 queries

Figure 6: Elaborated Example

3. *All branches are closed or successful:* First, the most recent relevant choicepoint has to be computed. This is exactly the corresponding choicepoint (saved in the forward execution) of the last activate on the parent branch that was deactivated in the past, and corresponds to a closed, unsuccessful branch. If no such activate is found, the parent adpack-or is taken as choicepoint.

If the most recent relevant choicepoint is the parent adpack-or, close the parent branch, cut away all its choicepoints, and backtrack to the parent adpack-or. Otherwise, cut up to the computed choicepoint, and backtrack to it.

4. *There is still an open, unsuccessful branch:* Since this branch has been tried before, it has failed without deactivation or success. No special actions have to be taken here: backtrack to the previous choicepoint.

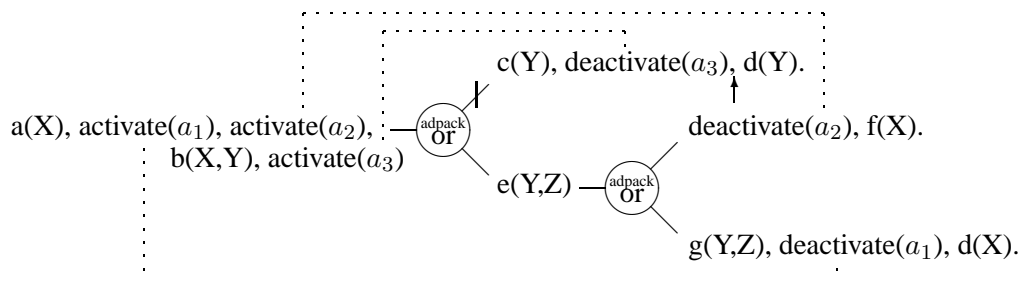
For more details on each different step, we refer to the high-level Prolog implementation in Appendix A, which follows the given specification very close.

7 Elaborated Example

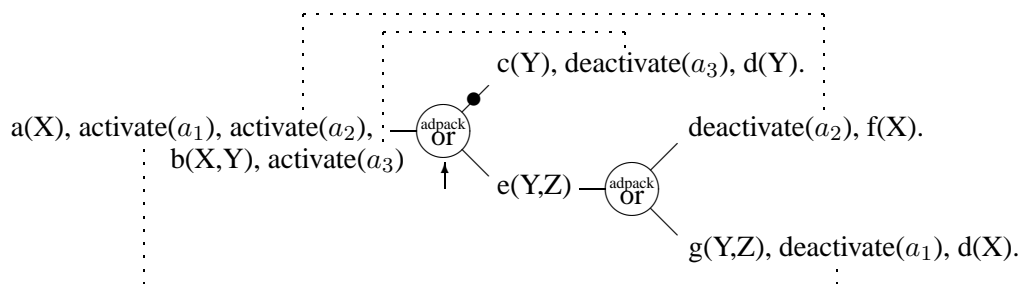
We will now illustrate the execution of adpacks on a larger example covering most aspects of the execution. The example adpack and the fact database on which we will run the pack are given in Figure 6.

Executing the first branch of the adpack binds the variables X and Y to 1 (due to the calls to *a* and *b*). Since the branches containing the deactivate of a_1 , a_2 and a_3 are initially unsuccessful, and since all branches are initially open, the 3 activate goals on the branch will do nothing but remember the last choicepoint, being the choicepoint of *a* for the first two activates, and the choicepoint of *b* for the third activate. At the end of the first branch, the first adpack-or is reached, and its first child is chosen for

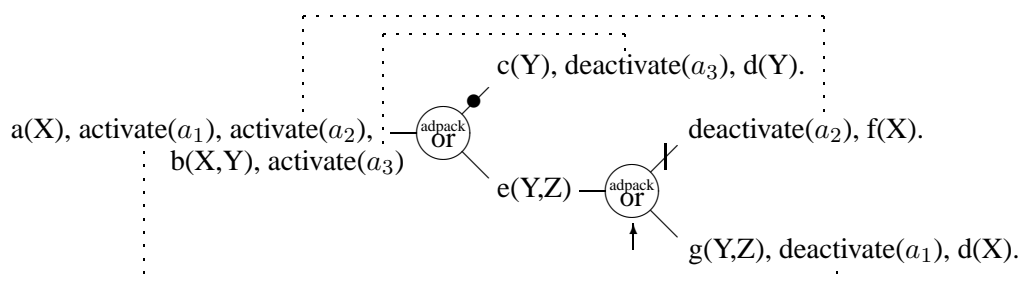
execution. After executing the first alternative of c , $deactivate(a_3)$ is encountered, representing the end of the once. This means that the choicepoints on the current branch (in this case only the choicepoint belonging to c) have to be cut away, and that the current branch has to be marked as closed, such that execution won't enter this branch until $activate(a_3)$ (denoting the begin of the once) is executed again. The adpack is now in the following state:



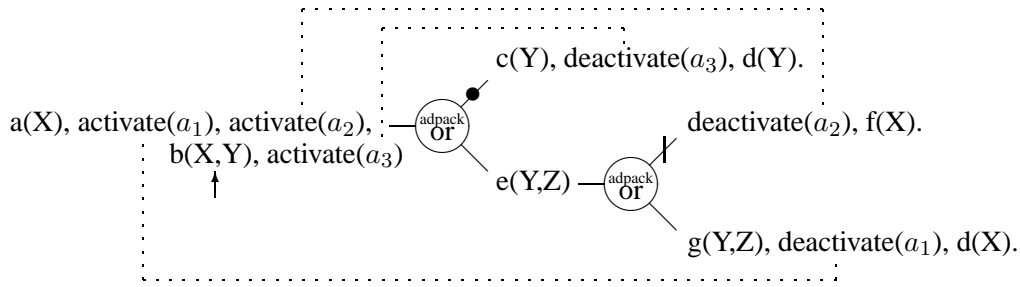
(a bar denotes that a branch is closed, and the arrow indicates the current point of execution). After the succeeding call to d , the end of the query is reached, and so the branch is marked as successful (denoted by a dot), and execution backtracks to the parent adpack-or:



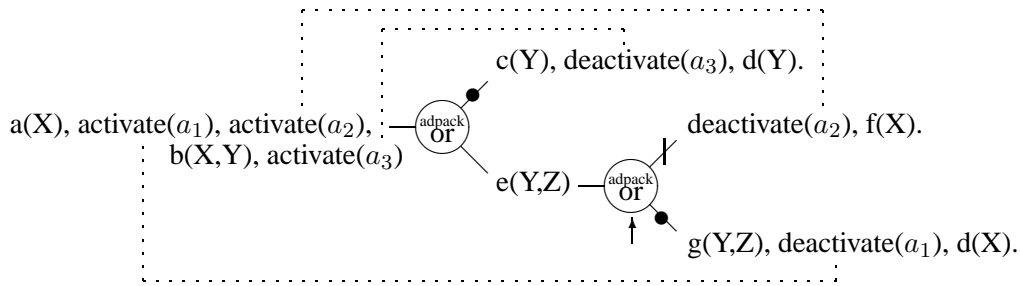
The following branch is executed, resulting in a successful call to e (thus binding Z to 1) and encountering the next adpack-or. Again, the first branch of the latter is executed, causing the branch to be closed immediately due to the $deactivate$. The next call to f fails, causing execution to backtrack back to the parent adpack-or:



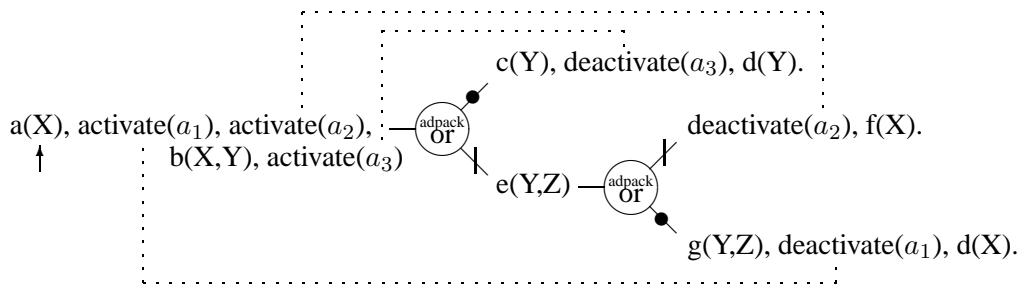
Execution of the last branch will fail immediately due to the call to g , returning execution to the parent adpack-or. Since all branches of the current adpack-or have been tried, and not all of them are closed, normal backtracking occurs, returning the execution in the parent adpack-or (since e has no alternatives). There, the same situation arises, resulting in backtracking to the nearest choicepoint, $b(X, Y)$:



Forward execution now binds Y to 2, and then arrives in the first adpack-or again. Now, only the second branch is still open for execution (since the first branch is already successful), and is therefore executed. After calling e (again binding Z to 1), we arrive in another adpack-or with one open branch. Indeed, the first branch is still closed, because execution didn't backtrack over $activate(a_2)$ (the begin of the once) yet, so the alternative solutions are not interesting for this branch. Entering the only open branch, execution now reaches the end of the branch, and so it is marked as successful:

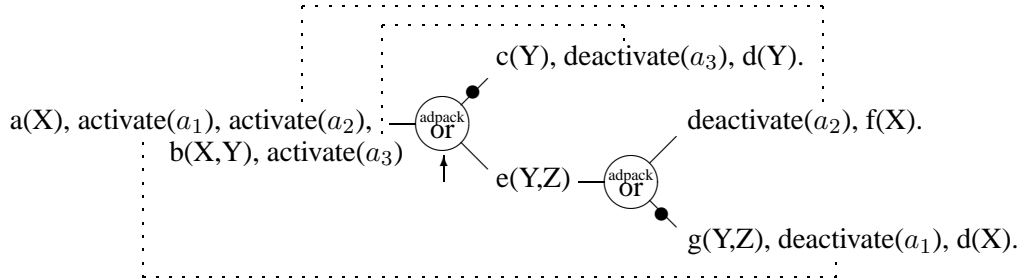


At this point, execution is in an adpack-or where all branches are closed (or successful). Now, the system should backtrack to a point that would cause an activate to be executed that opens one of the children branches of this adpack-or. Since there is no such backtrackpoint on the parent branch, we close the latter too, and look higher up, in the parent adpack-or, for such a backtrackpoint. There, again a relevant choicepoint is searched on the parent branch. Since the deactivates of a_3 and a_1 both lie on a branch that is already successful, it isn't interesting to reactivate these activates. However, a_2 still corresponds to an unsuccessful branch, so backtracking to the choicepoint before its activate will reopen the path to that branch. Therefore, execution backtracks to the call to $a(X)$ (the most recent choicepoint before the most recent relevant activate, a_1):

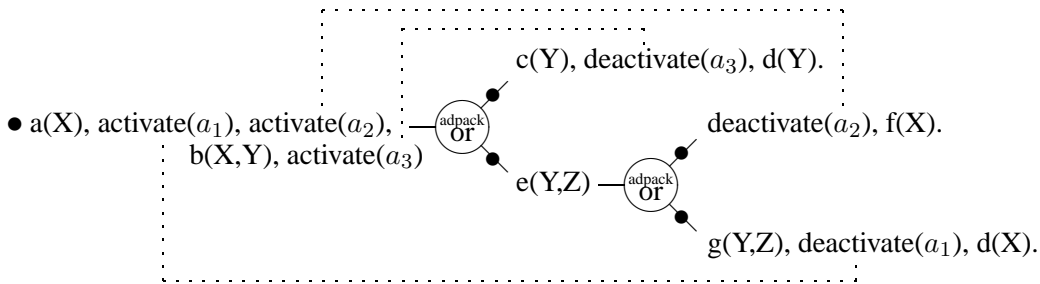


Now, forward execution restarts, binding X to 2, executing the activate of a_1 (which does nothing since it corresponds to a successful branch), then arriving in the activate of a_2 . This activate opens all branches from

the corresponding deactivate up to the current branch. Indeed, this activate corresponds to the begin of a once, and since we backtracked to a point before the once, backtracking inside the once is allowed again. After binding Y to 1 and executing the activate of a_3 (which also does nothing), execution arrives in the top-level adpack-or:



Execution follows the single path of open branches, finally reaching the end of the last unsuccessful branch, which is now marked as successful. Since all child branches of the adpack-or are successful, its parent branch is marked for success also, and execution arrives in the top-level adpack-or. All the latter's children are also successful now, and therefore the top-level branch is marked for success, ending execution of this adpack:



8 Implementation

The results from [3] indicate that, to benefit from a special execution mechanism such as that for query packs, it has to be implemented in the system at hand. Therefore, just as for query packs, we compile an adpack to specialized WAM bytecode [11].

8.1 Compiling

Each adpack that is to be executed is handed to the compiler, which compiles it to WAM-bytecode. This code will make use of 6 new WAM instructions: `adpack_start`, which initializes all the data structures; `adpack_try`, which will do the forward execution of an adpack-or; `adpack_retry`, which will handle the backtracking to an adpack-or; `adpack_activate` and `adpack_deactivate`, to activate and deactivate parts of the adpack; `adpack_success`, to register the success of a branch of an adpack-or. The implementation of these instructions is described in 8.3. Besides code, the compiler also generates information on the structure of the adpack. We omit the details of storing this information together with the code. With both the code and knowledge of the structure, the system has enough information to load the code into memory, initialize all data structures, and execute the adpack.

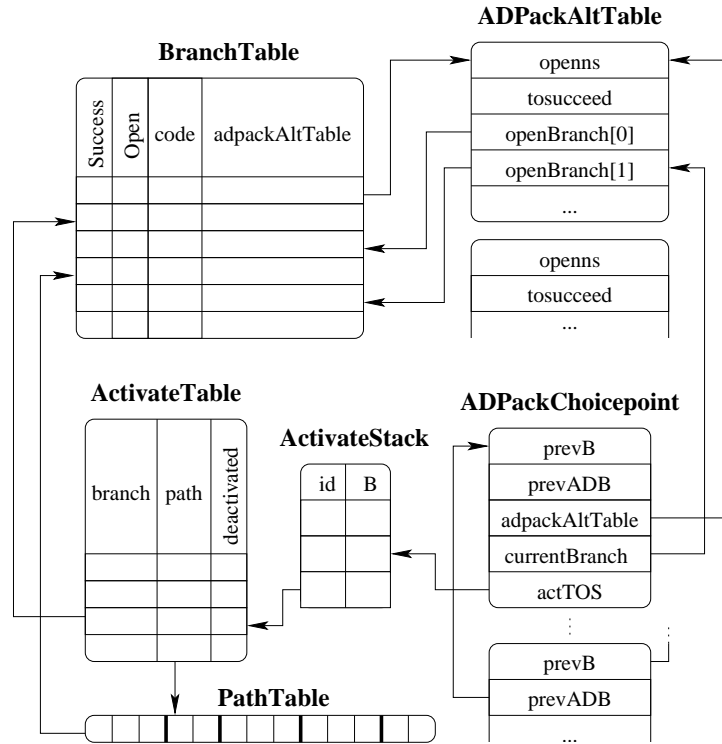


Figure 7: Overview of the data structures in the implementation of adpack execution

8.2 Data structures

To execute an adpack, we make use of the data structures depicted in Figure 7. These data structures are designed for executing adpacks as efficient as possible. The description of these data structures is as follows:

- **BranchTable:** This global table contains for each branch in the adpack the following information: a boolean indicating whether this branch was completed successfully (*success*, initially false); a boolean indicating whether the branch can be entered for execution. (*open*, initially true); the location of the code block for this branch (*code*); and finally a reference to the alternatives table of the adpack-or to which this branch belongs (*adpackAltTable*).
- **ADPackAltTable:** Each adpack-or has a corresponding ADPackAltTable. This table contains the number of branches which are still unsuccessful (*tosucceed*), the number of open, non-successful branches (*openns*), and for each of those open, non-successful branches an entry in *openBranch* containing its own index in the BranchTable.
- **ActivateTable:** This global table contains for each activate/deactivate pair a boolean saying whether the deactivate has been triggered (*deactivated*), the branch on which the deactivate resides (*branch*), and a reference to a path in the PathTable (*path*). The row on which a deactivate is located in the ActivateTable is exactly the unique identification number of the activate/deactivate pair (as described in 4).
- **PathTable:** This table contains for each activate/deactivate pair a sequence of branches describing the path from the activate to the deactivate. Each path is referenced from an entry in the ActivateTable.

- **ActivateStack:** Each time an activate is performed, the unique identification number of the activate/deactivate pair to which this activate belongs (*id*) is pushed onto the ActivateStack, together with the most recent choicepoint at the time the activate is triggered (*B*).
- **ADPackChoicepoint:** Whenever an adpack-or is executed, an ADPackChoicepoint is created, containing a reference to the previous choicepoint (*prevB*), a reference to the previous ADPackChoicepoint (*prevADB*), a reference to the ADPackAltTable of this adpack-or (*adpackAltTable*), the index of the currently executed branch in the openBranch table of the corresponding ADPackAltTable (*currentBranch*, initially 0), and the top of the ActivateStack at the time the execution of the adpack-or started (*actTOS*). The pointer to the code to be executed upon backtracking to this choicepoint will always point to the `adpack_retry` instruction.

Finally, we impose an extra constraint on the unique identification number associated with each activate/deactivate pairs: for each path from the root to the leaf of the adpack, the identification numbers of the activates on that path should increase. Numbering the activate/deactivate pairs by depth-first traversal satisfies this condition.

8.3 Executing the ADPack Instructions

In this section, we present the details of every WAM instruction.

8.3.1 `adpack_start` `<#adpack-or>` `<#branches>` `<#activates>` `<data>`

`adpack_start` indicates that an adpack will be executed. This instruction has to reset all data structures, using information on the total number of disjunctions (`<#adpack-or>`), the total number of branches in the adpack (`<#branches>`), the total number of activate/deactivate pairs (`<#activates>`), and the location of all data structures (`<data>`). The instruction should ensure that:

- All AltTables have all their children in `openBranch[]`, and that `opens` and `tosucceed` are initialized to the total number of children.
- The open flag of all branches in the BranchTable is set to true, and the success flag to false.
- The ActivateStack is emptied.
- The `deactivated` flag of all activate/deactivate pairs in the ActivateTable is set to false.
- A dummy ADPackChoicepoint is put on the choicepoint stack.

The implementation of this instruction is omitted, as it can be easily reconstructed.

8.3.2 `adpack_activate` `<id>`

This instruction activates the activate/deactivate pair `<id>`, opens the path from the activate to the deactivate, and updates the ActivateStack.

```
if (!BranchTable[*ActivateTable[id].path]->success) {
  if (ActivateTable[id].deactivated) {
    ActivateTable[id].deactivated = false;

    /* Open path */
```

```

    if (!(br=&BranchTable[ActivateTable[id].branch])->open) {
        /* Open first branch */
        br->open = true;
        br->adpackAltTable->openBranch[br->adpackAltTable->
            openns++] = ActivateTable[id].branch

        /* Open rest of path */
        path = ActivateTable[id].path;
        while (*path && !(br=&BranchTable[*path])->open) {
            br->open = true;
            br->adpackAltTable->openBranch[
                br->adpackAltTable->openns++] = *path;
            path++;
        }
    }
}

/* Pop choicepoints of more recent activates */
while ((ActivateStack-1)->id >= id)
    ActivateStack--;

/* Push most recent choicepoint */
ActivateStack->id = id;
ActivateStack->B = B;
ActivateStack++;
}

```

8.3.3 adpack_deactivate <id>

This instruction registers deactivation of an activate/deactivate pair, and closes the current branch. The only argument of `deactivate` is the unique number of the activate/deactivate pair.

```

/* Register deactivation */
BranchTable[ADB->adpackAltTable->
    openBranches[ADB->currentBranch]].open = false;
ActivateTable[id].deactivated = true;

/* Cut up to previous ADPackChoicepoint */
B = ADB;

```

Note that `ADB` is a register containing a reference to the previous `ADPackChoicepoint`.

8.3.4 adpack_success

This instruction denotes the end of a branch, and corresponds to the terminating cut from 2. It registers success of the current branch, and backtracks.

```

/* Register success & decrement #tosucceed */
BranchTable[ADB->adpackAltTable->

```

```

    openBranches[ADB->currentBranch]].success = true;
ADB->adpackAltTable->tosucceed--;

/* Cut up to previous ADPackChoicepoint */
B = ADB;

/* Backtrack */
fail;

```

8.3.5 adpack_try <id>

The `adpack_try` instruction corresponds to the part from Section 6 describing the forward execution of an `adpack-or`. This instruction puts an `ADPackChoicepoint` on the `choicepoint-stack`, using the first argument of the instruction to lookup the `adpackAltTable` corresponding to the `ADPack` that is going to be executed. The `currentBranch` field is initialized to 0, the `actTOS` field is set to the current top of the `ActivateStack`, and the address of the `adpack_retry` instruction is used as alternative for the choicepoint. Finally, the program counter is set to the code for the first branch.

```

/* If there are no open nodes, backtrack */
if (adpackAltTable->openns =< 0)
    fail;

/* Make new choicepoint */
prevB = B
B++;
B->prevB = prevB;
B->prevADB = ADB;
B->adpackAltTable = (address of AdpackAltTable #id);
B->currentBranch = 0;
B->actTOS = /* top of */ ActivateStack;
B->alt = @adpack_retry;
ADB = B;

/* Set program counter */
PC = BranchTable[ADB->adpackAltTable->openBranch[0]].code;

```

8.3.6 adpack_retry

This instruction handles backtracking to an `adpack-or`. Using the information stored in the current `ADPackChoicepoint`, it is determined which action to take. The 4 cases correspond to the ones in Section 6.

```

tbl = ADB->adpackAltTable;

/* Temporarily remove branch if necessary & set current branch */
if (BranchTable[tbl->openBranches[ADB->currentBranch]].success
    || !BranchTable[tbl->openBranches[ADB->currentBranch]].open) {
    tbl->openns--;
    tbl->openBranches[ADB->currentBranch] =
        tbl->openBranches[tbl->openns]
}

```

```

}
else {
    ADB->currentBranch++;
}

/* Reset stack */
/* top of */ ActivateStack = ADB->actTOS;
if (ADB->currentBranch < tbl->openns) {           /* Case 1 */
    /* Try another branch */
    PC = BranchTable[tbl->openBranches[tbl->currentBranch]].code;
}
else {
    ADB = ADB->prevADB;

    if (ADB->adpackAltTable->tosucceed == 0) {     /* Case 2 */
        /* Mark branch as successful */
        goto adpack_success;
    }
    else if (ADB->adpackAltTable->openns == 0) {   /* Case 3 */
        /* Pop until a non-successful deactivated branch */
        while (ActivateStack > ADB->actTOS) {
            ActivateStack--; // Pop element
            if (ActivateTable[ActivateStack->id].deactivated
                && ! BranchTable[ADB->adpackAltTable->openBranch[
                    ADB->currentBranch]].success) {
                /* Cut & backtrack */
                B = ActivateStack->B;
                fail;
            }
        }

        /* Close branch */
        BranchTable[ADB->adpackAltTable->openBranches[
            ADB->currentBranch]].open = false;
        B = ADB;
        fail;
    }
    else {                                       /* Case 4 */
        B = B->prevB;
        fail;
    }
}
}

```

8.4 Example

Compiling the example from Figure 6, would result in the following WAM code:

```

1 adpack_start 2 4 3 0x800121      17 adpack_success
2 allocate 5                        18 putpval Y3 Y1
3 putpvar Y2 A1                    19 putpval Y4 Y2
4 call a/1                          20 call e/2
5 adpack_activate 0                21 adpack_try 2
6 adpack_activate 1                22 adpack_desactivate 1
7 putpval Y2 A1                    23 putpval Y2 A1
8 putpvar Y3 A2                    24 call f/1
9 call b/2                          25 adpack_success
10 adpack_activate 2               26 putpval Y3 A1
11 adpack_try 1                     27 putpval Y4 A2
12 putpval Y3 A1                   28 call g/2
13 call c/1                        29 adpack_desactivate 0
14 adpack_desactivate 2            30 putpval Y2 A1
15 putpval Y3 A1                   31 call d/1
16 call d/1                         32 adpack_success

```

Notice that the `adpack_retry` instruction is not explicitly present in the code. It is stored only once somewhere in the data memory, since the exact location of the instruction does not matter (it gets all its information from the current `ADPackChoicepoint`).

8.5 Optimizations

In practical ILP applications, activates often come in batches. Consider the following part of an `adpack` coming from real-life queries:

$$\dots p(X), activate(a1), activate(a2), \dots, activate(a100), q(X, Y) \dots$$

After `p(X)` has succeeded for the first time, 100 activates are executed, and `q(X,Y)` is called. If the latter fails, another alternative for `p(X)` will be tried, and the first activate will pop all the later activates one by one from the stack. In practice, such a scenario happens frequently. The occurrences of these batches of activates cause inefficiency in both time (popping the stack one by one, plus a different emulator cycle for each activate) and space (all activate records on the stack will have the same choicepoint field).

To solve these problems, we introduce a new instruction `adpack_activate_range`, which has as its two arguments the begin and the end of the activate id range. This of course imposes the extra constraint on the compiler to have a subsequent numbering on the activates.

Additionally, we will now store variable-length records on the `ActivateStack`, where each record has a backtrackpoint, a pointer to the previous record, and a variable list of activate id's which belong to this record. Popping the stack in `adpack_activate` and `adpack_activate_range` can now be simplified to checking the last element of each record, and popping the whole record if this id is more recent than the id of the current activate. Pushing elements on this stack can be done in two ways:

- Each time an activate instruction has to put something on the stack, it checks if the choicepoint of the top element is the same as the current choicepoint. If so, the activate instructions add the new activate (or in the case of `adpack_activate_range`, batch of activates) to the current record. Otherwise, a new record is started and pushed onto the `ActivateStack`.
- `adpack_activate` always creates a new record on the `ActivateStack` containing 1 activate, and `adpack_activate_range` stores its batch of activates in one record. In this case, we trust the

compiler to determine statically which activate instructions can be grouped together. This means that there will be no overhead due to extending records on the stack, and all the dynamic checks will be omitted. The downside of this approach is that there could be groups of activates that can be grouped together on the stack (because they have they share the same most recent choicepoint), but that this couldn't be determined statically.

We will measure the impact of these optimizations in Section 9. The concrete changes to the implementation and the implementation of `adpack_activate_range` is omitted.

9 Experiments

The aim of our experiments is to gain more insight in the performance of adpack execution. We compare the following settings: regular query execution 'Query', once-transformed query execution 'Once', query pack execution 'Pack' and adpack execution 'ADPack'.

The experiments are performed with the ILP system TILDE [2], a first order decision tree learner available in the ACE system [1]. ACE is supported by ilProlog as described above. We run the experiments on a Linux Intel P4 1.8GHz system with 512Mb RAM.

Figure 8 shows the results of running TILDE in the different settings on three data sets: Mutagenesis [9] (230 examples), Carcinogenesis [8] (330 examples) and a version of Bongard [5] (10000 examples). For each setting, the graph shows the query execution time 'Execute', the compilation time 'Compile' (only relevant for query packs and adpacks – the other settings use meta-calls) and the transformation time 'Transform' (only relevant for once-transformed queries and adpacks). All bars are relative to the total time for 'Queries', which is shown, together with the number of refinement steps and the total number of queries evaluated, above its corresponding bar. Above the other bars, the speedup in execution time over 'Queries' is shown. The experiments are performed for 3 values of TILDE's lookahead parameter (the number of goals that is added in each refinement step).

In all experiments, query pack execution performs better than queries and the speedup increases as lookahead increases. Similar results were obtained in [3]. The once-transformed queries yield a significant improvement in execution time for the Carcinogenesis and Bongard data sets (without lookahead), but the improvement is smaller than that reported in [6] ($> 100\times$ for Carcinogenesis). A possible explanation is that the greedy search strategy of TILDE avoids the highly non-determinate queries for which the once-transformation performs best. In [6], the ILP system Aleph [7] was used, which implements a branch-and-bound search strategy. The performance of the once-transformed queries is less good for higher lookahead values. Possibly, the goals added by lookahead share variables with goals from the start of the query (the language bias allows this), which has a negative effect on the once-transformation. More detailed experiments are necessary to confirm this.

The new adpack query execution mechanism performs better than all other settings (execution time), but is close to query packs in most cases (especially when the once-transform does not perform well). The best improvement over query packs is obtained on Carcinogenesis ($\pm 10\times$). For some applications (e.g., Mutagenesis with lookahead ≥ 1), the higher transformation³ and compilation times of the adpack version can make regular query pack execution the best choice (best total time). However, the proportion of time spent during transformation and compilation decreases as the number of examples increases (cfr. Bongard data set with 10000 examples – the transformation and compilation time are not visible on the graph).

³The transformation time for adpacks is higher than that of the once-transformed queries because the queries must also be merged in the adpack (cfr. Figure 3).

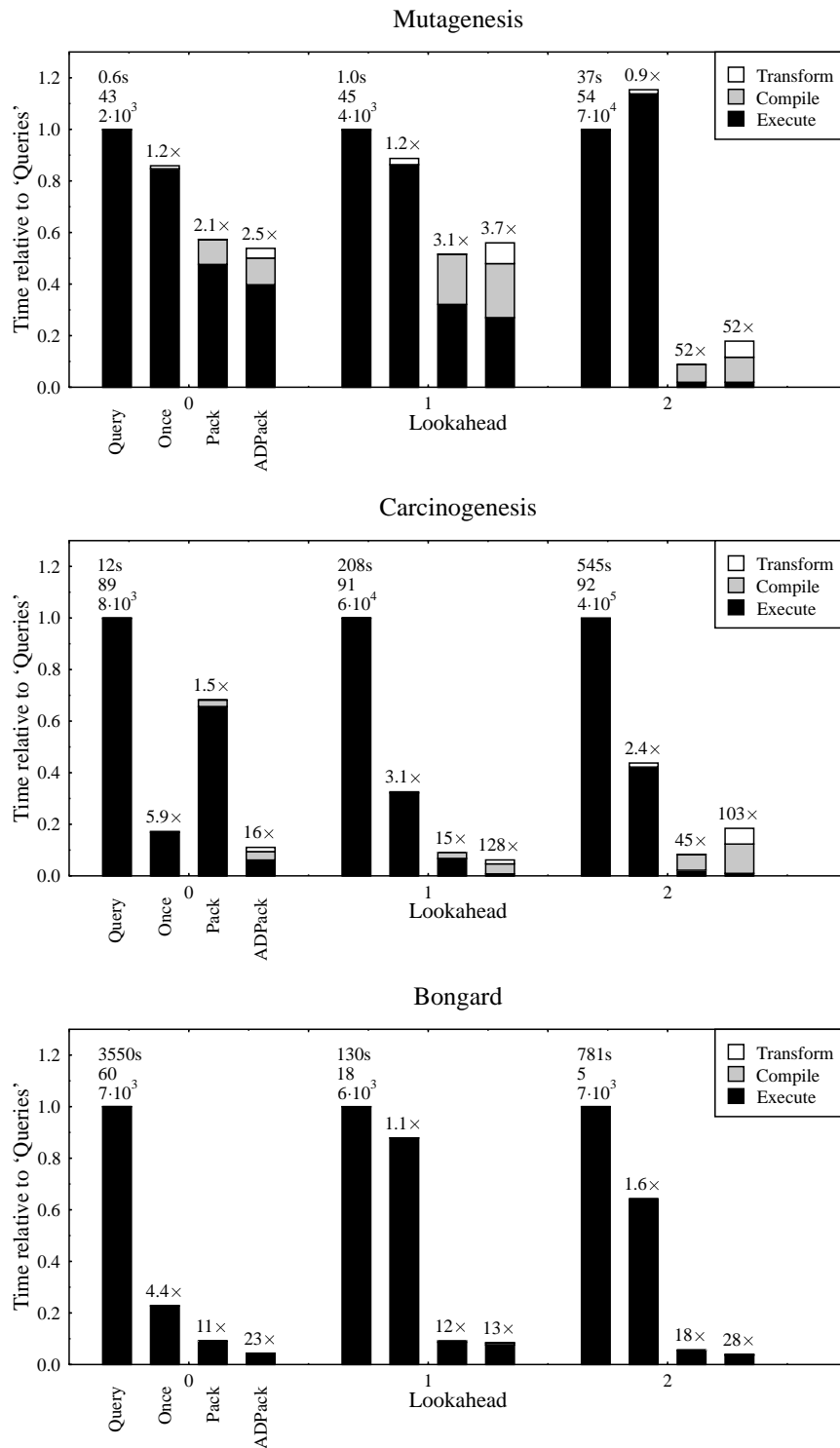


Figure 8: Experiments on the data sets Mutagenesis, Carcinogenesis and Bongard.

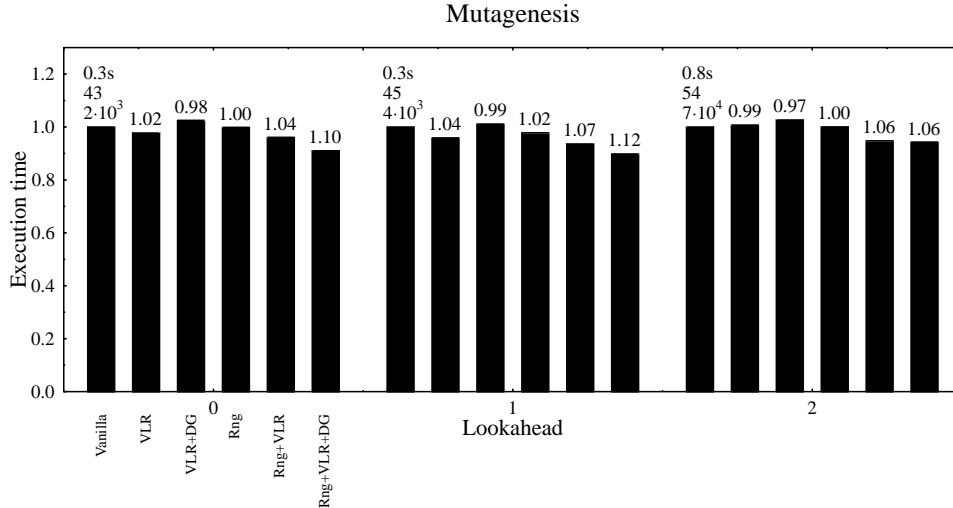


Figure 9: Effect of various optimizations on the execution time

As a final experiment, we compare the impact of the optimizations introduced in Section 8.5. Figure 9 shows the execution time for 6 different versions of the ADPpack execution mechanism on the Mutagenesis dataset, where each of the versions is formed by using a combination of following changes (each described in more detail in 8.5):

- **Variable-length records (VLR):** Store variable-length records on the ActivateStack. This not a real optimization, just a change of data structure. This change is needed for the following optimizations.
- **Dynamic Grouping (DG):** Accumulate activates with the same corresponding choicepoint in the top record of the ActivateStack. Only start a new record if the choicepoint of the activate (or group of activates) is different than the one of the record on top of the ActivateStack.
- **Use `activate_range` (Rng):** let the compiler statically group subsequent activates together, and generate `activate_range` instructions for these groups.

As can be seen in Figure 9, the optimizations do not introduce significant changes in execution time. Variable-length records seem to be a slightly more efficient representation for the ActivateStack. The extra checks needed for dynamic grouping are not compensated in execution time if activates aren't grouped together using `adpack_range`. This is due to the fact that when activates are statically grouped together, the check only has to be performed once for each group of activates, instead of for each activate separately. Introducing `activate_range` does not improve execution time in itself, but combined with dynamic grouping it results in the best execution time.

10 Conclusion & Future Work

The preliminary experiments from Section 9 show that the evaluation of queries benefits from the new execution mechanism, compared to using only query packs or query transformations. There are some open questions regarding the once-transformation: the effect of the ILP system and search strategy that is used and the impact of the lookahead parameter. Further work will include experiments designed to answer these questions. Another aspect which has to be investigated is to what extent the reordering of literals done by

the once-transformation has an influence on the execution time. Finally, the once-transformation was only one of many query-transformations applicable to ILP queries. The combination of query packs with the other transformations is another interesting topic of investigation.

References

- [1] The ACE data mining system. <http://www.cs.kuleuven.ac.be/~dtai/ACE/>.
- [2] H. Blockeel and L. De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, June 1998.
- [3] H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Improving the efficiency of Inductive Logic Programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135–166, 2002.
- [4] H. Blockeel and M. Sebag. Scalability and efficiency in multi-relational data mining. *SIGKDD Explorations*, 5(1), 2003. To Appear.
- [5] L. De Raedt and W. Van Laer. Inductive constraint logic. In K. P. Jantke, T. Shinohara, and T. Zeugmann, editors, *Proceedings of the Sixth International Workshop on Algorithmic Learning Theory*, volume 997 of *Lecture Notes in Artificial Intelligence*, pages 80–94. Springer-Verlag, 1995.
- [6] V. Santos Costa, A. Srinivasan, R. Camacho, H. Blockeel, B. Demoen, G. Janssens, J. Struyf, H. Vandecasteele, and W. Van Laer. Query transformations for improving the efficiency of ILP systems. *Journal of Machine Learning Research*, 4(Aug.):465–491, 2003.
- [7] A. Srinivasan. The Aleph manual. <http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/>.
- [8] A. Srinivasan, R. King, and D. Bristol. An assessment of ILP-assisted models for toxicology and the PTE-3 experiment. In *Proceedings of the Ninth International Workshop on Inductive Logic Programming*, volume 1634 of *Lecture Notes in Artificial Intelligence*, pages 291–302. Springer-Verlag, 1999.
- [9] A. Srinivasan, S. Muggleton, M. Sternberg, and R. King. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, 85(1,2):277–299, 1996.
- [10] J. Struyf and H. Blockeel. Query optimization in inductive logic programming by reordering literals. In *Proceedings of the 13th International Conference on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2003. To appear.
- [11] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI, 1983.

A Meta-interpreter for ADPacks

```

% Adpack Example:
%
% {br0} a(X), activate(a1), / {br1} c(Y), deactivate(a3), d(Y).
% activate(a2), b(X,Y), activate(a3) \ {br3} deactivate(a2),
% f(X).
% {br2} e(Y,Z) \
% {br4} g(Y,Z), deactivate(a1), d(X).
%
test_adpack :-
  adpack_execute(
    [a(X), activate(a1), activate(a2), b(X,Y), activate(a3), adpack_or([
      branch(br1,[c(Y), deactivate(a3), d(Y)]),
      branch(br2,[e(Y,Z), adpack_or([
        branch(br3,[deactivate(a2), f(X)]),
        branch(br4,[g(Y,Z), deactivate(a1), d(X)]))]
    ])]
  ),
  write('Success: '), (success(S), write(S), write(' '), fail ; nl).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% NOTES:
% * The following predicates are used to manipulate the data structures,
% and are assumed to be predefined:
% - init_datastructures/1: initializes all data structures
% - set_open/2, set_success/1: sets the 'open' and 'success' flag of a branch
% - open/1, success/1: checks the 'open' and 'success' flag of a branch
% - set_deactivated/2: sets the 'deactivated' flag of an activate/deactivate pair
% - d_branch/2: the branch on which a deactivate is located
% - parent/2: retrieves the parent branch of a branch
% These predicates are easy to implement using assert/retract.
% * The predicate '_$savecp'(B) is a builtin predicate to retrieve the most
% recent choicepoint, '_$cutto'(B) cuts away all choicepoints up to B.

adpack_execute(Pack) :-
  init_datastructures(Pack), % Initialize all data structures
  '_$savecp'(B), % Get top choicepoint
  adpack_execute(Pack,br0,[],B). % Start executing
adpack_execute(_).

adpack_execute([activate(Id)|Gs], CurrentBranch, Activates, ParentCutp) :- !,
  d_branch(Id,DeactBranch), % Get edge containing deactivate
  (\+ success(DeactBranch) -> % If corresponding edge is unsuccessful
  (deactivated(Id) -> % If deactivated flag is set ...
  open_path(DeactBranch,CurrentBranch), % Open path from end to cur.
  set_deactivated(Id,false) % Clear deactivated flag
  ;
  true
  ),
  '_$savecp'(B), % Get current choicepoint
  NActivates = [act(Id,B)|Activates] % Remember current choicepoint
  ;
  NActivates = Activates
),
adpack_execute(Gs,CurrentBranch,NActivates,ParentCutp).

adpack_execute([deactivate(Id)|Gs], CurrentBranch, Activates, ParentCutp) :- !,
  set_deactivated(Id,true), % Set deactivated flag
  set_open(CurrentBranch,false), % Close current edge
  '_$cutto'(ParentCutp), % Cut up to parent adpack_or
  adpack_execute(Gs,CurrentBranch,Activates,ParentCutp).

adpack_execute([], CurrentBranch, _Activates, ParentCutp) :- !,
  set_success(CurrentBranch), % Register success

```

```

    '$_scutto'(ParentCutp),          % Cut up to parent adpack_or
    fail.                            % Backtrack

adpack_execute([adpack_or(Branches)], CurrentBranch, Activates, ParentCutp) :- !,
(
    %% Forward execution %%
    member(branch(Branch,Goals), Branches), % Get a child edge
    \+ success(Branch), open(Branch),      % If edge is unsuccessful and open ...
    '$_savecp'(B),                          % Get current choicepoint
    adpack_execute(Goals,Branch,[],B)      % Execute edge
;
    %% Backtracking %%
    ( all_success(Branches) ->           % If all children are succesful ...
      set_success(CurrentBranch),        % Set success flag of parent branch
      '$_scutto'(ParentCutp)            % Cut up to parent adpack_or
    ;
      \+ contains_open_branch(Branches), % If all edges are closed ...
      % Compute most recent relevant cutpoint on current edge
      determine_cutpoint(Activates,ParentCutp,CP),
      (CP == ParentCutp ->              % If cutpoint is parent adpack_or ...
        set_open(CurrentBranch,false)   % Close edge
      ;
        '$_scutto'(CP)                  % Cut up to parent
      )
    ),
    fail                                 % Backtrack
).

adpack_execute([G|Gs], CurrentBranch, Activates, ParentCutp) :-
    call(G),                             % Execute goal
    adpack_execute(Gs,CurrentBranch,Activates,ParentCutp).

% Opens all branches in a path until an already open branch is found or
% the begin branch is reached
open_path(BeginBranch, BeginBranch) :- !.
open_path(CurrentBranch, BeginBranch) :-
    ( open(CurrentBranch) ->
      true
    ;
      set_open(CurrentBranch,true),
      parent(CurrentBranch,ParentBranch),
      open_path(ParentBranch,BeginBranch)
    ).

% Succeeds if all branches in the list are successful
all_success([]).
all_success([branch(Branch,_)|Branches]) :-
    success(Branch),
    all_success(Branches).

% Succeeds if the list of branches contains an open, unsuccessful branch
contains_open_branch(Branches) :-
    member(branch(Branch,_),Branches),
    \+ success(Branch),
    open(Branch).

% Computes the most recent useful cutpoint to backtrack to
determine_cutpoint([], ParentCutp, ParentCutp).
determine_cutpoint([act(Id,B)|Acts], ParentCutp, Cutp) :-
    d_branch(Id,E),
    ((deactivated(Id), \+ success(E)) ->
      Cutp = B
    ;
      determine_cutpoint(Acts,ParentCutp,Cutp)
    ).

```