A Heuristic for the Subgraph Isomorphism Problem in Executing PROGRES

Albert Zündorf[1]

Department of Computer Science III
Aachen University of Technology
Ahornstraße 55, 52074 Aachen, Germany

e-mail: albert@i3.informatik.rwth-aachen.de
phone: ++49/+241/80-21312 fax: ++49/+241/80-21329

Abstract

The work reported here is part of the PROGRES (**PRO**grammed **G**raph **R**ewriting **S**ystems) project. PROGRES is a very high level multi paradigm language for the specification of complex structured data types and their operations. The data structures are modelled as **di**rected, **a**ttributed, **n**ode and **e**dge labelled graphs (diane graphs). The basic programming constructs of PROGRES are graph rewriting rules (productions and tests) and derived relations on nodes (paths and restrictions). These basic operations may be combined to build partly imperative, partly rule based, complex graph transformations by special control structures which regard the nondeterministic nature of graph rewriting rules.

In order to use PROGRES not only as a specification language but also for building rapid prototypes and even for the derivation of final implementations, we have to be able to execute PROGRES specifications efficiently. The central problem in executing a graph rewriting system is to match the left-hand sides of the executed graph rewriting rules to their isomorphic images within the current work graph. In this paper we propose a heuristic solution for this subgraph isomorphism problem when executing PROGRES.

1 Introduction

Graph-like data structures are said to be not 'efficient' since many algorithms and problems for graphs like finding a minimal graph colouring, finding a k-factor, or finding a minimal absorbing node set are known to be NP-complete (cf. [Gould 88, Volk 91]). Also, graph grammars and graph rewriting systems seem to be 'inefficient' since executing a graph rewriting step involves the problem of matching the left-hand side of a graph rewriting rule (in the following called production) to a subgraph of the current working graph in order to replace it by the right-hand side of the rule. This means we have to solve the (sub-) graph isomorphism problem which is known to be NP-complete in the size of the left-hand side of the applied production. In general, a naive algorithm for executing a graph rewriting step has the *time complexity* $O(P * N^{L})$ where P is the number of available productions, N is the number of nodes in the

1. Supported by *Deutsche Forschungsgemeinschaft.*

current work graph, and L is the maximum size of a left-hand side of the system (cf. [BuGlTr 91]). Although being polynomial, this complexity is acceptable at most for a rapid prototype. But, for a final implementation of e.g. balanced, sorted, binary trees only insert and delete operations with the time complexity $O(\log N)$ are acceptable.

Although we knew about these complexity problems, the advantages of *graphs* as a general purpose data structure and of *graph grammars* as a well-suited theory for modelling operations on graphs convinced us to use them as basic concepts at our department (cf. [Nagl 79]). We built GRAS (**gra**ph **s**torage), a nonstandard database management system especially for diane graphs (cf. [LewSchü 88, KiSchWe 92]). We use GRAS as the central data management and data integration component in many software development environments within the framework of the IPSEN project (**i**ntegrated **p**roject **s**upport **en**vironments, cf. [EnJaSchä 88]). Furthermore, we developed our own specification language PROGRES (cf. [Schürr 91, Schürr 91b, ZünSchü 92]) which is based on graph rewriting rules as basic operations, and we use this language to specify the operational behaviour of our IPSEN tools since 1980 (cf. [EngSchä 85]). In the beginning, we used PROGRES in a pencil and paper manner. But in 1988 we started to build an IPSEN-like development environment for PROGRES now including a partly textual, partly graphical, structure oriented editor for PROGRES specifications and an incrementally working analyzer, performing a huge number of consistency checks while editing a specification.

Since 1991 we are working on an *execution component* for the PROGRES environment. At first, we only wanted to build an interpreter and compiler for PROGRES in order to gain rapid prototypes for (a subset of) a specified system and thus to be able to verify the behaviour of the specified data structures and operations. Meanwhile we are convinced that it is possible to derive an efficient implementation directly from its PROGRES specification. This derivation process can be mechanized completely, only in some cases efficiency hints from the user in the form of pragmas may be needed.

Naturally, the generated system can only be as good as its specification. Poor and inefficient specifications will lead to poor implementations. All we hope we will be able to guarantee is that all information contained in a specification is used and that the generated code for a graph rewriting rule is of the same time complexity and nearly as good as a hand coded implementation of this graph rewriting rule. Especially, the generated code should not introduce any efficiency drawbacks caused by the use of a graph grammar based language for its derivation.

To achieve these goals we have to reconsider the complexity $O(P * N^L)$ for performing a graph rewriting step for the language PROGRES. The factor P results from testing all productions of a graph grammar for applicability. In PROGRES, we use programmed graph rewriting steps, i.e. the user defines which graph rewriting rule (in PROGRES called *production*) may be applied in the next step by means of control structures, (cf. [ZünSchü 92]). So, in PROGRES P is usually a very small number and in many cases even equal to 1. The factor N^L results from the subgraph isomorphism problem. A naive implementation computes every possible mappings of the L nodes of the left-hand side of a production to the N nodes of the current graph. Then, for every matching, correct adjacency is tested until all required conditions hold. Once a subgraph isomorphic to the left-hand side is found, the replacement step simply has costs in the size of the nodes and edges which have to be removed, created, or redirected. These costs are determined by the modification to be performed (and by the data structure used to store the graph) and will arise in a hand coded solution, too. So the central problem that remains for the execution of PROGRES is the problem of efficiently matching the left-hand side of a production to a subgraph of the current work graph: the *subgraph isomorphism problem*.

We address this problem for PROGRES by several special indexing methods within our graph storage system GRAS, by special new language elements and pragmas allowing access to these indexing schemes, and by a heuristic algorithm for computing an ‘optimal’ GRAS query plan for the left-hand side of a production. These techniques and especially the resulting algorithm are outlined in this paper.

In chapter 2 we introduce a sample specification for balanced, threaded, binary sorting trees. In chapter 3 we define the set of all possible GRAS query sequences that match the left-hand side of a given production. So, in chapter 4, we only have to define a metric for such GRAS query sequences in order to choose the optimal one for generating a good implementation for a production.

2 The language PROGRES

For an informal introduction to PROGRES, we will use a small specification of *balanced, threaded binary trees*. This well-known data structure can be used for search and sorting problems with variable data. We use this data structure as an example in order to show that its PROGRES specification has the same time complexity as an implementation in any other language e.g. like C.

For the definition of basic operations on diane graphs, PROGRES provides its users with means for the construction of *graph rewriting rules* (productions and tests) and *derived relations* on nodes (paths and restrictions).

Figure 1 shows an example tree in the form of a directed, attributed, node- and edge-labelled graph (diane graph). We now try to insert a new element 9 into this tree. This will be done by the production call TBTInsertLeft(Anchor, 9). Figure 2 shows the declaration of the production TBTInsertLeft. Applying this production to the graph in Figure 1 works as follows:

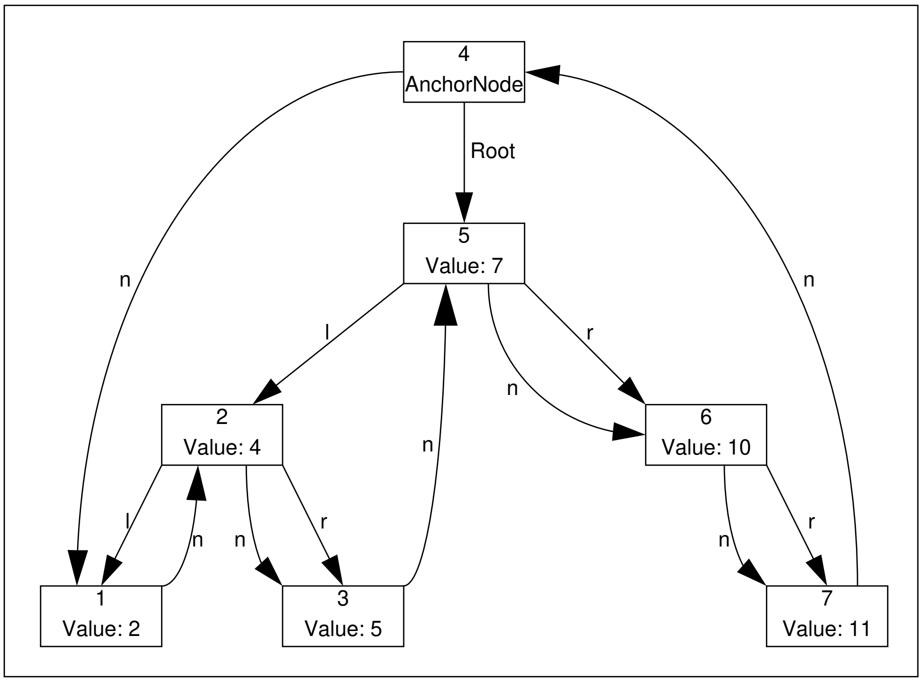
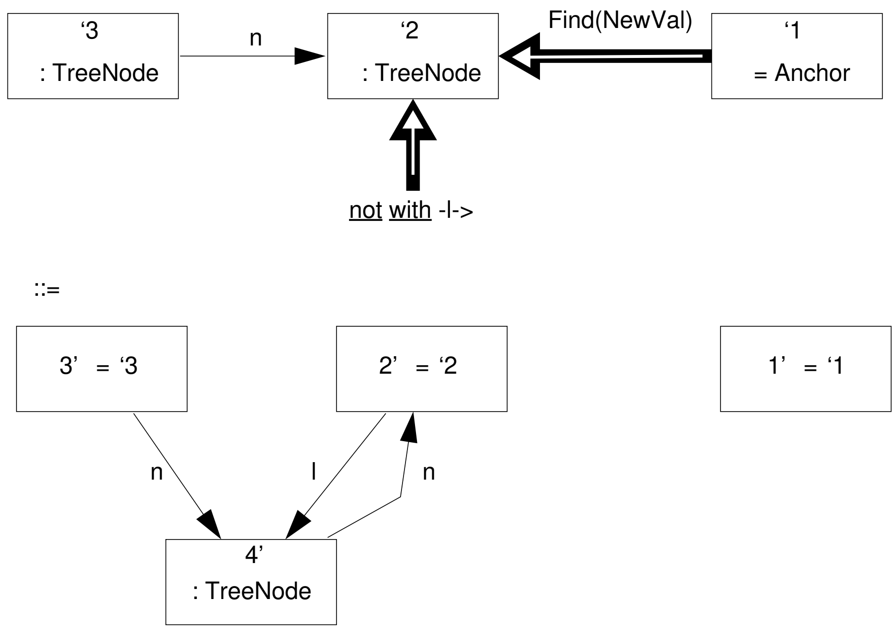


Figure 1: An example threaded binary tree

```
node class Node
  intrinsic Value : integer ;
end;
node type AnchorNode : Node end;
node type TreeNode : Node end;
edge type Root : Node [0:1] -> Node [0:1];
edge type l : Node [0:1] -> Node [0:1];
edge type r : Node [0:1] -> Node [0:1];
edge type n : Node [0:1] -> Node [0:1];
production TBTInsertLeft ( Anchor : Node ; NewVal : integer ) =
```




```
  condition ('3. Value < NewVal) and (NewVal < '2. Value);
  transfer 4'.Value := NewVal;
end;
path Find ( NewVal : integer ) : Node -> Node [0:1]
=
  -Root-> & { valid (NewVal < self. Value) & -l->
            | valid (NewVal > self. Value) & -r-> }
end;
```

Figure 2: Parts of the specification of TBTrees

1. First of all, a (partial) subgraph of the current work graph which is isomorphic to the graph on the left-hand side of the production is determined, i.e. a subgraph consisting of three nodes, one of type AnchorNode ('1) and two nodes of type TreeNode ('2 and '3). The nodes matched by '2 and '3 have to be connected by an edge labelled n which builds the thread. The restriction "not with -l->" to node '2 ensures that the matched node not already has an outgoing edge labelled l connecting it to a left son. The first parameter of TBTInsertLeft, Anchor, is used to pass the internal reference for the node to be matched by '1 (in our example in Figure 1 this is the key number 4)[1]. The assignment "= Anchor" within node '1 directly matches this node to the node with the corresponding key in our data

1. This internal key is returned by the production TBTCreate as out parameter.

structure. From node ‘1, the derived relation (in PROGRES called *path*) “Find(NewVal)” leads to node ‘2 (i.e. the matched nodes 4 and 6 in Figure 1). The path Find is also defined in Figure 2. From the start node (which is an implicit parameter to the path) first an edge labelled Root is traversed. This part of the path is concatenated by & to an iteration enclosed by { }. The iteration consists of two alternative parts. If the parameter NewVal is less than the Value attribute of the current node (denoted by self), and an outgoing edge labelled l exists, then this edge is traversed. Else (|), if NewVal is greater than the current value and an r-edge exists, we walk down to the right son. This is iterated until all (i.e. both) alternatives fail. Then the current start node of the iteration is the computed result[1]. So the path Find(9) in our example leads from the Anchor node 4 via the Root edge to node 5 and via the second alternative to the final node 6, the insertion point for our new tree element. Finally, the attribute condition “‘3.Value < NewVal < ‘2.Value” ensures that we really found the correct insertion place[2].

2. When a matching subgraph of the current work graph has been selected, this subgraph is removed from the work graph and the graph on the right-hand side of the production is inserted. A node inscription like “2’ = ‘2” designates an identical replacement of a node by itself. This means that 2’ is the same node as ‘2 and all its attribute values and all incoming or outgoing edges remain unchanged unless they are explicitly mentioned within the production. So the production removes only the n-edge connecting the nodes that are matched against ‘3 and ‘2. A new node of type TreeNode is created and connected to the remaining graph by two n-edges and the l-edge. Finally the transfer instruction is executed, assigning NewVal to the Value-attribute of the newly created node.

One possible result graph of an application of TBTInsertLeft(Anchor, 9) to the graph in Figure 1 is shown in Figure 3. We hope that this short introduction to PROGRES suffices to give an impression of the language and its operations as well as the task to execute a PROGRES production. The rest of the paper concentrates on matching the left-hand side of a production to an isomorphic image within the current work graph.

3 The search plan space

The most general strategy for matching the left-hand side of a production works as follows: First, we compute the sets of candidates for all left side nodes (by querying our database for the sets of all nodes of the corresponding types). Then, for every left side node we choose a current candidate. Finally, we check our choices for fullfilling all remaining edge, path and attribute requirements of the left-hand side. If any of these requirements fails, we have to try another combination of candidates for the left-side nodes. This is repeated until all requirements are met (and an matching subgraph is found) or until all combinations failed (the searched subgraph doesn´t exist).

While this standard search strategy could be used for any production, it is not very efficient since it does not utilize any of the advanced querying possiblities and indexing schemes of our underlying database management system GRAS. Often, it is possible to determine a left-side

1. Note that, in general, traversing an edge may yield a set of nodes and the iteration will be considered for each element of this set, so the iteration computes a set of nodes as result. Note also that cyclic iterations are detected and terminated automatically (cf. [Schürr 91]).
2. This attribute condition is added only for demonstration purposes, since in a correct structured search tree, the construction of our left-hand side and of the path Find already ensure the correct insertion point.

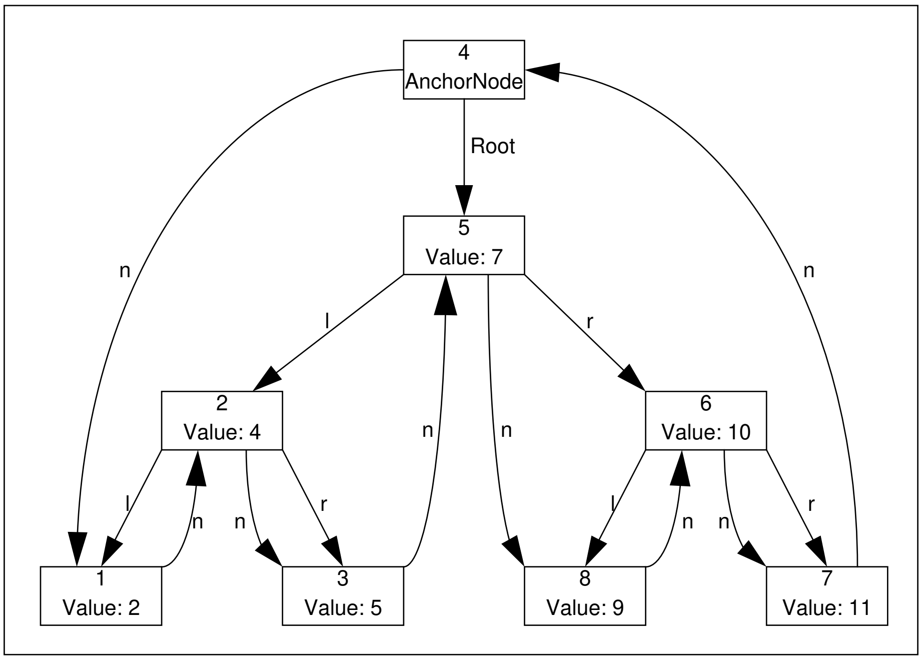


Figure 3: The resulting tree of applying TBTInsertLeft(Anchor, 9) to the tree of Figure 1

node by evaluating it´s initial assignment. Once a candidate is chosen for a left-side node, its neighbours could easily be determined by traversing the corresponding left-side edges. Inspecting attribute conditions and other restrictions as early as possible we often could avoid the costs of query actions for non matching canditate combinations. In general there is a huge number of alternative action sequences that could be used to match the left-hand side of a given graph rewriting rule. So, the central problem in executing a graph rewriting rule is to find a sequence of basic query actions that matches the left-hand side with a good or optimal efficiency.

In this chapter we first model the left-hand side of a production as a diane graph. Then we consider possible query actions for matching the different elements of a left-hand side and their interdependencies. Therefore, we enrich the diane graph built for the left-hand side of a production by *action nodes* (representing the query actions) which are connected to the parts of the left-hand side graph they are dealing with. Once we have built the graph modelling all elements of the left-hand side of our production and all query actions that may be used to match these elements, this *action graph* will be used to derive a sequence of query actions that match the whole left-hand side to (one of) its isomorphic image(s) within the current work graph as efficiently as possible.

Within the PROGRES environment, a production is stored as an abstract syntax tree enriched by context sensitive edges comprising symbol table information (cf. [Lewe 88, Koss 92]). For our purposes a simpler representation suffices, reducing the structure of left-hand side elements to a single node with a text attribute containing the corresponding specification part.

The example production TBTInsertLeft is represented by the *left-hand side graph* shown in Figure 4. The left-hand side of TBTInsertLeft is modelled by one node for every node to be matched by the left-hand side, one node for every initial assignment, restriction and attribute condition, and one node for every required edge and path. The interdependencies of the left-hand side elements are represented by edges of the types:

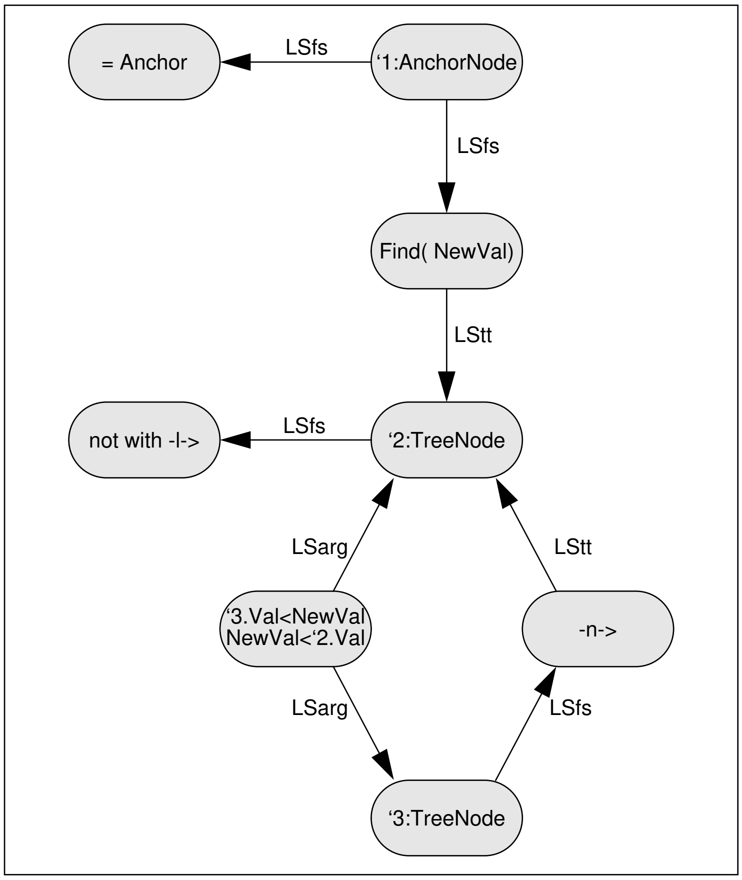


Figure 4: The left-hand side of TBTinsertLeft as diane graph

1. LSfs : left side node -> left side edge, path , or restriction.
2. LStt : left side edge or path -> left side node
3. LSarg :left side restriction or attribute condition -> left side node

LSfs-edges lead **f**rom a left side node to a left side edge, path, or restriction that uses it as **s**ource node. LStt-edges lead from left side edges or paths **t**o their **t**arget nodes. LSarg-edges lead from a left side restriction or attribute condition to left side nodes that are used as **arg**uments. E. g., in Figure 4 the node " '2 : TreeNode" is

- target of a LStt-edge from the path "Find(NewVal)",
- source of a LSfs-edge leading to the node "-n->",
- source of a LSfs-edge leading to the restriction "not with -l->",
- target of a LSarg-edge from the attribut condition " '3.Val<NewVal and NewVal<'2.Val", which uses '2 as an argument.

An element of a left-hand side may be matched to an element of the current work graph by executing several query actions. We now model each possible query action by an *action* node. Such an action node will be connected to the left side elements it **m**atches by m-edges and to the left side nodes it **r**equires to be already matched by r-edges. We could build these query action nodes by the means of PROGRES productions applied to the graph of Figure 4. But in order to avoid the confusions arising by such a bootstrap step we use a slightly modified notation (but with the same semantics).

In Figure 5 we start with a simple left side restriction. For a restriction just a TestRestriction

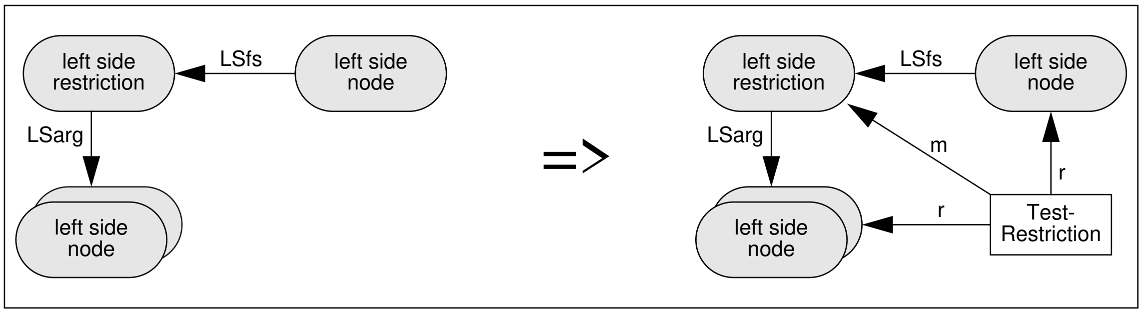


Figure 5: Action for left side restrictions

action is generated. Although the restriction may contain a complex application condition which may cause several query actions to be performed, only one action node is used here for simplicity reasons. This action node matches the left side restriction (note the m-edge) and requires the left side node that is source for the restriction (see the r-edge). In general, a restriction may use any other left side node as an additional parameter. In Figure 5, the LSarg-edge leads from the restriction to a double box representing the (possibly empty) set of all left side nodes that are needed as arguments. The TestRestriction action is connected to all elements of this set by an r-edge modelling that these nodes are required by the test action.

By applying the rule of Figure 5 to the node "not with -l->" node of Figure 4 a TestRestriction node is generated, with one[1] r-edge leading to "'2:TreeNode" and with an m-edge leading to "not with -l->", cf. Figure 11.

For a left side attribute condition nearly the same graph rewriting rule is used. Instead of a TestRestriction action a TestAttCond action is created. Since an attribute condition has no incoming LSfs-edge the corresponding left side node is omitted, cf. Figure 5. By applying this

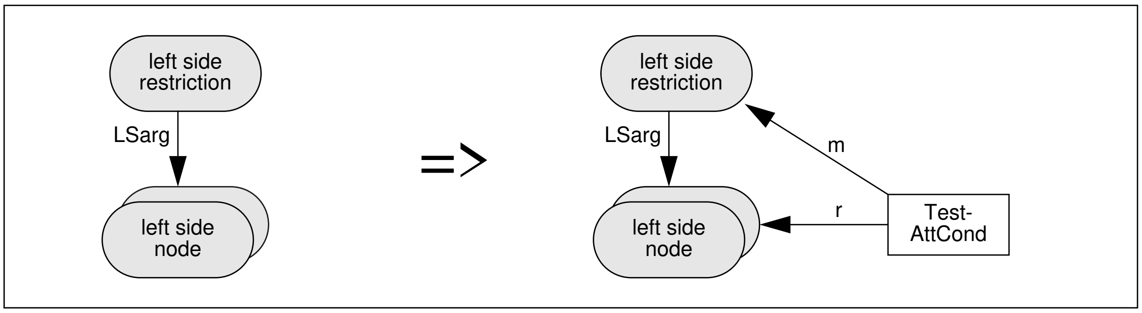


Figure 6: Action for left side attribute conditions

rule to the attribute condition "'3.Val<NewVal and NewVal<2'.Val", a TestAttCond node is generated, with an m-edge matching the attribute condition, and with two r-edges leading to the nodes "'2:TreeNode" and "'3:TreeNode", which are used (as arguments) within the attribtute condition (cf. Figure 10).

Initial assignments like "= Anchor" can not only be used to verify a matching attempt for a left side node but also to directly compute this match. So, in Figure 7, we refine the query plan for restrictions in the case of such left side *node expressions*. We generate two new query action nodes: one for testing the initial assignment and one for matching the left side node directly by getting the resulting key from evaluating the expression. Note, that the GetNodeExpr action

1. Note, "not with -l->" has no outgoing LSarg-edges.

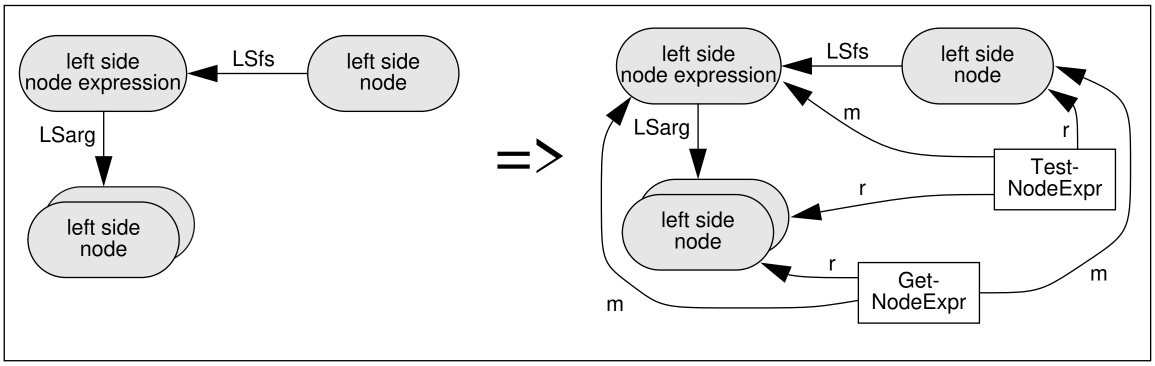


Figure 7: Action for left side node expressions

matches both left side elements, the node and the corresponding expression (cf. node "= Anchor" in Figure 10).

A left side edge may be matched by three different actions. If source and target nodes are already matched, we only have to test the existence of the required edge. On the other hand, if only one of the connected left side nodes already has been found until now, we could traverse the edge in forward or reverse direction using the actions GetTargets or GetSources respectively. Cf. Figure 8 and node "-n->" of Figure 11.

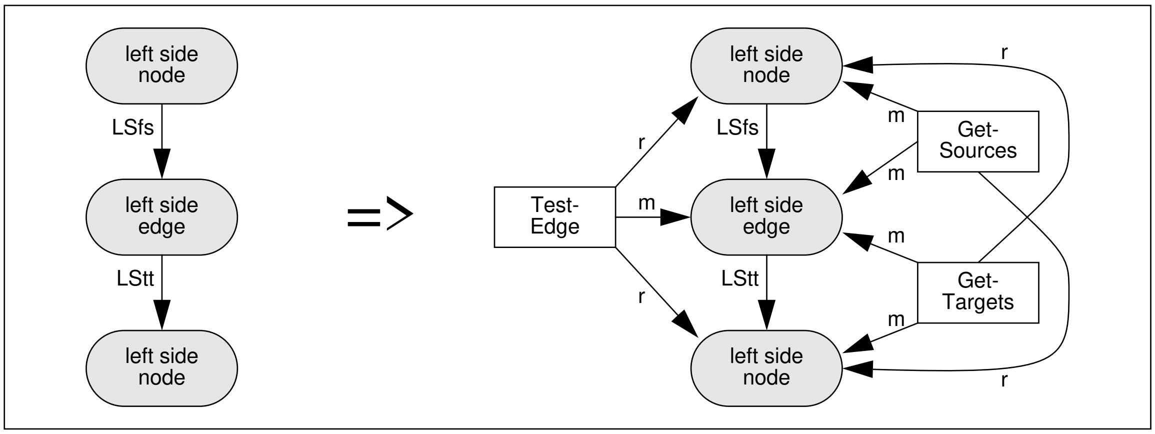


Figure 8: Action for left side edges

The next operation for our action graph generates a GetPath and a TestPath action for left side paths. Note that there is no GetSourcesOfPath action available since, unlike edges, we are not able to traverse paths against their declared direction. So, left side paths are handled like left side edges, just omitting the GetSources action and using the path actions instead of the edge actions, cf. node "Find(NewVal)" of Figure 11.

Finally, in Figure 10, the most general action that can be used to match a left side node is modelled. We are always able to match a left side node by querying for all nodes of a certain type. This query action should be used only as a last resort, since in general it computes a very large set of nodes. Note that the GetInstances action always computes a set of nodes as result. Other Get actions like the GetTargets action for traversing an edge in general also compute a set of nodes. But, considering our example of Figure 2 again, we see that for all nodes of the tree there exists at most one outgoing edge of a given edge type. Thus, traversing e.g. an l-edge will compute at most one node. It is very important for our matching strategy to be able to divide the Get actions into *set valued* actions, which compute a set of nodes, and into *element valued* actions that compute at most one node. Thus we augmented the declarations of edge types (and paths) in PROGRES by an optional cardinality assertion. The declarations

```
edge_type E1 : Node -> Node [0:1] ;
edge_type E2 : Node -> Node [1:1] ;
edge_type E3 : Node -> Node [0:n] ;
edge_type E4 : Node -> Node [1:n] ;
```

ensure that for every node in the specified graph holds that there exists at most one outgoing E1-edge, exactly one outgoing E2-edge, any number of E3-edges, and at least one E4-edge. The same cardinality assertions may be made for the source types, cf. Figure 2.

To be able to make use of this cardinality information for our matching strategies, we introduce an additional left side *candidates* node for every left side node, modelling the results of set valued actions. A candidates node represents a set of candidates for matching the corresponding left side node. A GetSelection action is used to loop through the set of candidates. The original left side node represents the actual choice for matching this left side node to a node of the current work graph. Thus, the GetSelection action requires the candidates node and matches the corresponding left side node representing the current selection.

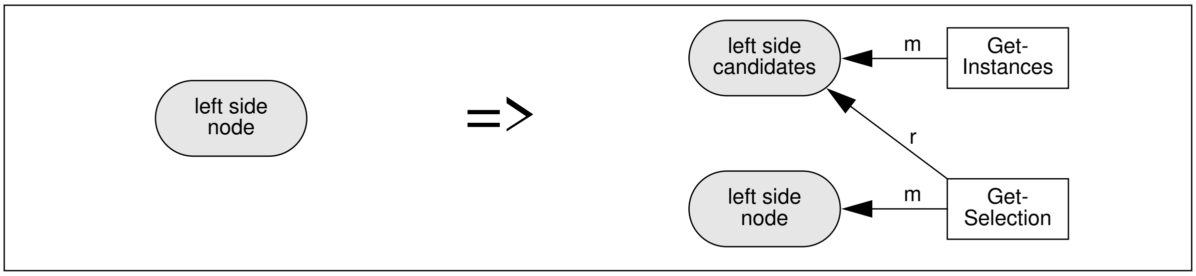


Figure 9: Actions for left side nodes

At this point we have to refine all previous actions to reflect the distinction of set and element valued Get actions. Set valued actions will match only the candidates node, while element valued actions directly match both the candidates and the corresponding left side node. Instead, we use the action of Figure 10 to (1) remove all m-edges connecting a set valued Get action with the original left side node, and (2) to insert new m-edges connecting the set valued and the element valued Get actions to the candidates node.

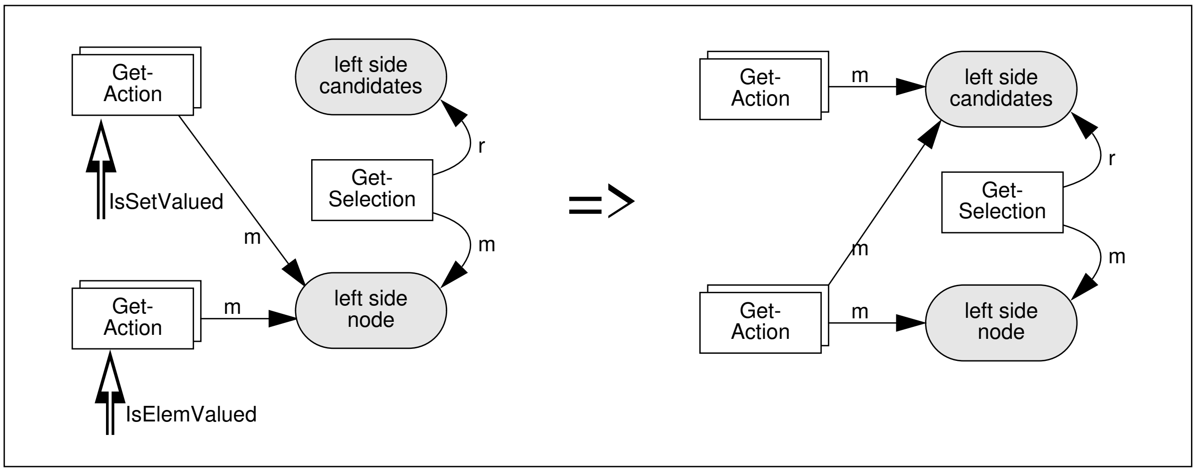


Figure 10: Splitting set and element valued Get actions

Applying all these operations (using the one of Figure 10 as the last) to the graph of Figure 4 leads to the action graph for TBTInsertLeft shown in Figure 11. Note that in our example speci-

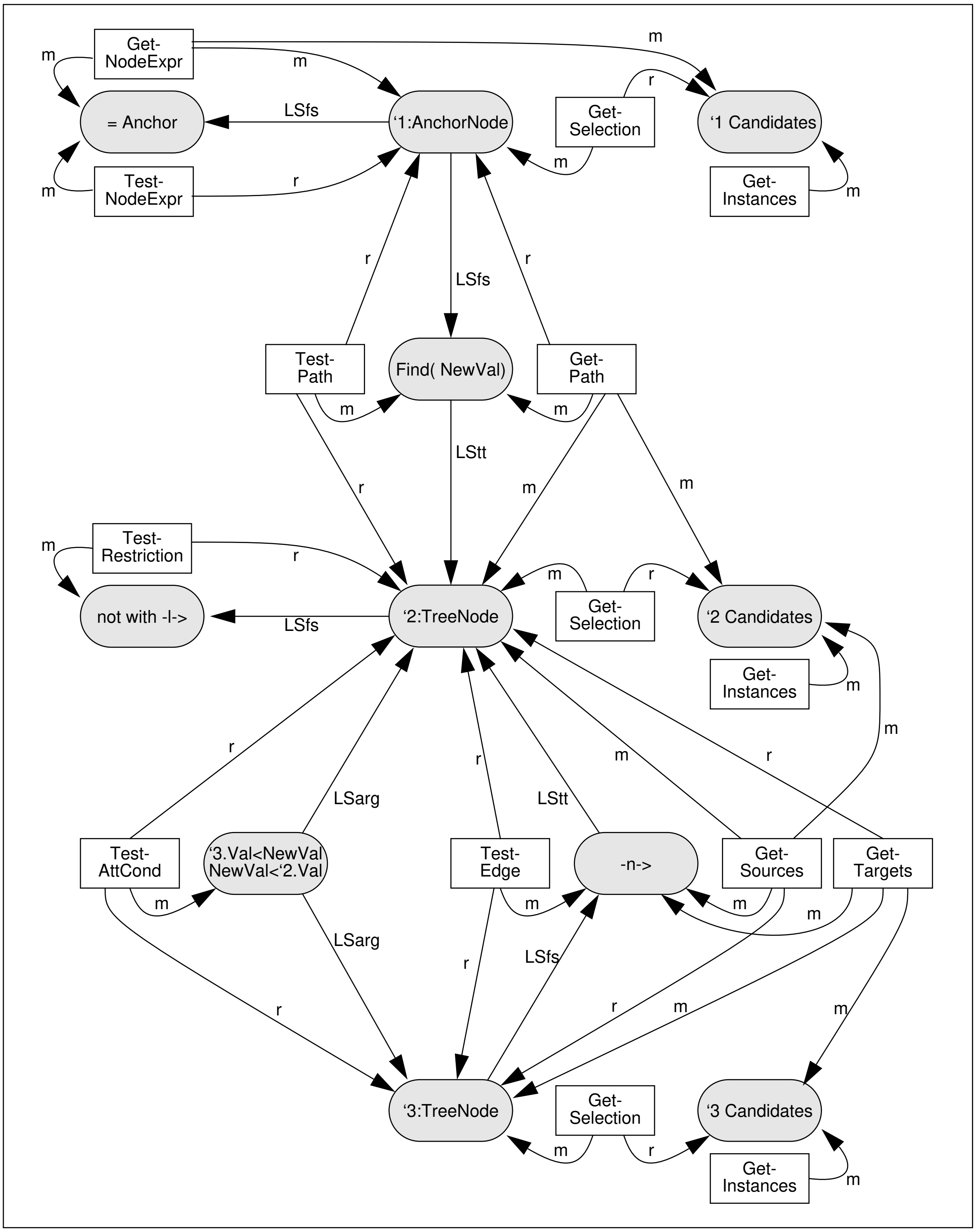


Figure 11: The action graph for TBTInsertLeft

fication of Figure 2, all edges are declared as of type Node [0:1] for both directions. Thus the corresponding Get actions are element valued. The same holds for the GetPath and for the GetNodeExpression actions within the example. All these Get actions match both the candidates and the left side node.

Thus prepared, we are now able to define some central issues of our algorithm formally:

Definition 1:

Let P be a production of specification Spec:

1. We call the graph AG_P built by the operations described within this chapter the *action graph* of production P.
2. We call $L_P := \{ l \mid l$ is a left side element node within $AG_P \}$ the *left side elements* of P.
3. We call $AS_P := \{ a \mid a$ is a query action node within $AG_P \}$ the *action set* of P.
4. We call $AL_P := a_1, \ldots, a_n$ where all $a_i \in AS_P$ an *action list* for P.
5. We call a *enabled by* AL_P if and only if
   $\forall\ l \in L_P$ where l is required by a holds : $\exists\ b \in AL_P$ such that b matches l.
6. We call $SP_P := a_1, \ldots, a_n$ (where all $a_i \in AS_P$) a *search plan* for P if and only if
   $\forall\ l \in L_P$ holds: $\exists\ a \in SP_P$ such that l is matched by a and
   $\forall\ a_i \in SP_P$ holds: a_i is enabled by $a_1, \ldots, a_{i-1}$
7. We call $PLANS_P := \{ sp \mid sp$ is a search plan for P $\}$ the *search plan space* for P.

Proposition 1:

$\forall$ P where P is a production of specification Spec holds:
$PLANS_P \neq \emptyset$

For every production there exists at least one search plan.

Proof:

Idea: First, all left side nodes may be matched to the work graph by the corresponding GetInstances and GetSelection actions. Once all left side nodes are matched, the remaining left side elements, like edges, restrictions, and attribute conditions, are applied by the corresponding test actions.

Let GI_P be an action list of all GetInstances nodes of AG_P,
let GS_P be an action list of all GetSelection actions of AG_P,
let TA_P be an action list of all Test actions of AG_P.

We call $SSP_P := GI_P \circ GS_P \circ TA_P$ a *standard search plan* for P.

SSP_P is a search plan since:

1. By construction of SSP_P and AG_P, all left side nodes and candidates are matched by GI_P and GS_P. Additionally, for every remaining left side restriction, attribute condition, node expression, and path, there exists a test action that exactly matches this left side element. Since these test actions are in TA_P, part one of definition 1.6 holds.

2. Since a GetInstances action requires no left side element, the empty set suffices to enable GI_P. Since GI_P matches all left side candidates, it suffices to enable the GetSelection actions of GS_P. Thereby, all left side nodes of AG_P are matched. As the test actions of TA_P may require only left side nodes, TA_P is enabled by $GI_P \circ GS_P$.

Thus, by definition 1.6, SSP_P is a search plan and by definition 1.7 $SSP_P \in PLANS_P$, q.e.d.

In general, the search plan space for a production P offers a wide range of possibilities for the organization of the subgraph search. In the next chapter we propose our heuristics for choosing an almost optimal one for the efficient execution of PROGRES.

4 The heuristics

In order to find an optimal search plan SP_P for a production P, we now define a function $Costs : PLANS_P \rightarrow \mathbf{N}$ that computes the estimated costs of a search plan. Then we define a search plan to be optimal if it has minimal costs within $PLANS_P$.

The principles we followed for calculating the estimated costs of a search plan are the same principles as known from relational database query languages. But compared to a relational database query, in PROGRES we have much more freedom in choosing query actions and search plans as shown in chapter 3.

Thus, the following points should be considered for the construction of a search plan:

1. The most important principle is to avoid the action GetInstances which computes the set of all nodes of a certain type. In general, the size of this set is proportional to the size of the current work graph and the whole database has to be inspected to compute it.
2. We try to keep the size of our candidate sets as small as possible, since we will have to iterate over all their elements by the GetSelection query. E.g. if a left side node n is target of several edges and / or paths that are declared to have multiple targets, then the corresponding Get actions all match the candidates node for n. So, if we select two or more Get actions we can intersect the result sets and thereby minimize the candidates set.
3. We try to avoid the candidates sets at all which is possible using element valued edges and paths. Then, additional set valued edges and paths are matched more efficiently by their test actions. This avoids the GetSelection action.
4. We prefer the usage of edges and paths that are declared to be partial, i.e that may compute an empty result. An empty result of a Get action terminates a wrong matching attempt. An early termination of a match attempt (perhaps within a GetSelection loop) avoids the costs of the following actions for this wrong selection. This technique is known as first-fail principle in constraint satisfaction systems (cf. [HaraElli 80, Henten 89]).
5. We try to choose test actions which apply restrictions and attribute conditions as early as possible, again in order to terminate wrong match attempts early. According to the first-fail principle we try to match nodes early that enable many test actions. A wrong match attempt for a node with many restrictions will fail with a (hopefully) higher probability, thus avoiding further useless computations based on such a wrong match.
6. Naturally, we prefer simple Test and Get actions like testing or traversing a single edge (perhaps with a single result) to complex actions like calling a path with a complex body like the path Find in our example.

Figure 12 shows a table of basic GRAS query actions and their estimated costs. These costs are absolutely heuristic. We choose these costs by (1) estimating the time complexity of the corresponding GRAS query. (2) We consider the estimated probability of an empty result for the actions. If the action probably fails, it detects a wrong match and keeps the costs of the whole search low. (3) The most important factor is the expected result size. Note that all Test actions are of lower costs than the Get actions, that set valued Get actions have ten times the costs of

action	costs
TestAttCond	1
TestEdge [0:1]	2
TestEdge [1:1]	3
TestEdge [0:n]	4
TestEdge [1:n]	5
TestRestriction	6

action	costs
TestPath [0:1]	10
TestPath [1:1]	12
TestPath [0:n]	14
TestPath [1:n]	16
GetNodeExpr [*:1]	20
GetEdges [*:1]	22

action	costs
GetPath [*:1]	24
GetNodeExpr [*:n]	200
GetEdges [*:n]	220
GetPath [*:n]	240
GetSelection	500
GetInstances	1000

Figure 12: Estimated costs of basic query actions

the corresponding element valued actions, and that GetInstances actions are really expensive since we expect a huge result set.

The estimated costs in Figure 12 are not very realistic. E.g. the GetPath action is estimated to have about 10 times the costs of a single edge. These costs do not differentiate between very simple paths which may follow only a small number of edges and really complex paths that e.g. call several other complex paths or contain iterations or other expensive operations. In general, the costs of an action may differ between different specifications, between different edge types or paths within one specification, and naturally for different current graphs for the same edge type or path. We will reconsider this aspect at the end of this chapter.

Now we are able to define the costs of a search plan just as the sum of the costs of its actions:

Definition 2:

Let $SP_P := a_1, \ldots, a_n$ be a search plan for P

We define the function Costs : $PLANS_P \rightarrow \mathbf{N}$ by:

$$Costs(SP_P) := \sum Costs(a_i)$$

We call SP_P an *optimal* search plan if it has minimal costs within $PLANS_P$.

This cost function is quite simple, but together with the algorithm proposed below it already leads to quite good computational behaviour of the generated code for the subgraph search. We will discuss refinements of our cost function at the end of this chapter.

In order to avoid the computational overhead to determine the set of all possible search plans we propose a greedy algorithm constructing a search plan just by choosing the cheapest enabled action until all left side elements are matched, cf. Figure 13.

Only in the case of all cheapest enabled actions being GetInstances operations, the algorithm considers all possible alternatives and chooses the one that leads to the cheapest following costs. If e.g. the production TBTInsertLeft of Figure 2 would not contain the “<= Anchor” restric-

```
PROCEDURE PlanSearch( ag : ActionGraph; sp : ActionList) : ActionList;
BEGIN
  IF ( ∃ l where l is not matched by sp) THEN
     EnabledActions := { a | a is enabled by sp };
     UsefullEnabledActions := { a ∈ EnabledActions | a matches an element not matched by sp }
     MinCostActionSet := { a ∈ UsefullEnabledActions | a has minimal costs within UsefullEnabledActions }
     min_a := GetAnyElem( MinCostActionSet );
     MinCostActionSet := MinCostActionSet \ min_a;
     MinPlan := PlanSearch( ag, sp ∘ min_a );
     MinCosts := Costs( MinPlan );
     IF ( min_a is a GetInstances or GetSelection action) THEN
        WHILE ( NOT MinCostActionSet is empty ) DO
           min_a := GetAnyElem( MinCostActionSet );
           MinCostActionSet := MinCostActionSet \ min_a;
           NewPlan := PlanSearch( ag, sp ∘ min_a );
           IF ( Costs( NewPlan ) < MinCosts ) THEN
              MinPlan := NewPlan;
              MinCosts := Costs( MinPlan );
           END (* if *);
        END (* while *);
     END (* if *);
  END (* if *)
  RETURN MinPlan;
END PlanSearch;
```

Figure 13: Search plan algorithm

tion, the only possibility to start the subgraph search would be a GetInstances action. But if we would select the GetInstances action for one of the TreeNodes first we would have to use a second GetInstances action to match the AnchorNode node, since the path Find can be traversed in forward direction only. Considering all three left side nodes of TBTInsertLeft as start points for the subgraph search, our algorithm would determine that the best search plans would always start with the AnchorNode node because then only one of the expensive GetInstances actions is necessary.

Applying our algorithm to the original TBTInsertLeft production of Figure 2 using the action graph of Figure 11 works as follows:

1. At the beginning only the GetInstances actions for all nodes and the GetNodeExpr action for the AnchorNode node are enabled. Thus, our algorithm chooses the GetNodeExpr action matching the node expression, the anchor node, and its candidates node.
2. Now the GetInstances actions for the tree nodes and the GetPath action are enabled, where the latter one is chosen as cheapest. This matches the left side path, the tree node ‘2, and its candidate node.
3. At this point the GetInstances action for node ‘3, the GetTarget action for the n-edge, and the TestRestriction action for “<= not with -l->” are enabled. According to the costs table in Figure 12, TestRestriction is chosen, matching exactly the left side restriction.
4. Now, the GetTargets action for the n-edge is chosen, matching the left side edge, the node ‘3, and its corresponding candidates node.
5. Now the TestAttCond action is enabled, matching the last required left side element and the algorithm terminates.

The resulting search plan for the example production is optimal with respect to our cost model and looks quite reasonable. It uses the Anchor parameter as entry to the search and finds the other left side nodes by calling the path Find and traversing the n-edge. As proposed by our heuristics, all restrictions are applied as early as possible, terminating wrong match attempts early. Additionally, the computed search plan makes use of the cardinality informations for the left side path and edge and uses element valued actions avoiding the expensive GetSelection operation.

The production TBTInsertLeft is quite simple; determining a good search plan is pretty obvious. But more complex productions with many parallel edges and paths are handled satisfying by our algorithm. We applied it to several existing specifications (mainly of IPSEN tools) that were built without knowledge of this execution strategy. In most cases, good search plans were computed. The most problematic GetInstances action is avoided whenever possible, and the heuristic principles proposed within this chapter are respected. In some cases, little modifications of the productions lead to much better search plans; in some cases, a complex search plan was necessary due to the complexity of the modelled problem (we were not able to construct a better one by hand), and in some cases, we found that bad search plans were caused by a bad structured graph model.

On the other hand, the considered specifications were made by software engineers experienced in the design of complex data structures. They specified the operations on their data models in a way that an efficient execution is derived easily. So we plan to use PROGRES within a course for graduate students to gain more experience with our algorithm. We also plan to refine our cost model by offering the user means to redefine the estimated costs of basic query actions, e.g. for different edge types or paths[1] or even for different applications of these specification elements. We also could refine the Costs function in order to regard e.g. the number of left side elements matched by an action or regarding that the GetSelection actions iterate (parts) of the following search actions so that the costs of these iterated actions should have more weight for the overall costs. Additionally, we might consider all possible search plans instead of using our greedy algorithm, which in general needs less computational expense. Depending on the experiences we will make, we may have to refine our greedy algorithm for some other special situations. Eventually, we could ask the user to propose a search plan.

We will conclude this chapter by analyzing the computational complexity of the generated search plan for our example. Main memory implementations of diane graphs using adjacency lists can achieve a constant time complexity for primitive operations (cf. [MeNä 89]). It is possible to map the query actions used in this approach to the operations of such an implementation. Then, the complexity of the query actions is proportional to the size of the computed result. An empty result may cause the same costs as a result of size 1.

The search plan we computed for production TBTInsertLeft starts with the action GetNodeExpr. This leads to costs 1, since the key of the matched node is passed by Anchor which is element valued. The path Find iteratively traverses the tree structure down to the required insertion point. Within each iteration only element valued edges are traversed, leading to constant costs and thus to the complexity $O(\log N)$ for a balanced tree with N elements. The node '3 is matched by traversing an n-edge in reverse direction, and all remaining tests also require constant expense. The rewrite operation removes one n-edge, creates one new node and three new edges, and sets the value of one attribute. This is done by a constant number of query actions. Thus, the costs of the path Find dominate and the whole operation is of complexity $O(\log N)$. This is

1. Knowing the costs of traversing an edge more exactly we could try to analyze a path implementation in order to determine the costs of this path.

exactly the complexity that is determined by the specified problem. For this example, no efficiency drawbacks arise from using a graph grammar based operation for its implementation.

As already mentioned, currently the execution of PROGRES is based on GRAS. In GRAS, traversing an edge (as the other basic GRAS actions) has logarithmic costs in the size of the current graph (cf. [KiSchWe 92]). These additional costs are compensated since GRAS is a general purpose database system for diane graphs storing its data on disk and offering persistency, error recovery, and undo / redo capabilities[1]. Within the IPSEN project the GRAS based execution of PROGRES is well-suited because GRAS is already used as the central data management component. Since in other contexts a main memory implementation is preferred, we are considering to realize such a facility. It shall be possible to choose wether the generated code for a specification is based on GRAS or on a main memory implementation of diane graphs.

5 Conclusions

While there is a large amount of work done in the theory of graph grammars, there exist only a few systems for executing them. An early approach is the one of Göttler in [Gö 88]. Göttlers graph rewriting language includes rewriting rules and control structures of comparable comprehensiveness to those of PROGRES (cf. [ZünSchü 92]). His implementation is able to execute a graph rewriting system. But, the problem of efficiently finding a match for the left-hand side of an applied rule is transferred to the user. He has to provide a function that computes the desired match. This function may use some standard search functions provided by the system. But, the problem of finding a good matching strategy in order to gain an efficient implementation is the task of the programmer.

Today, the GraphEd system of [Hims 89] includes a facility for the execution of *context free* graph grammers, i.e., the left-hand side of rewrite rules consist of single nodes only. GraphEd originally is an editor for graphs. Within this editor the graph grammer part of the system is intended as a macro mechanism. So, the user selects a node and then apply a rewrite rule on it. The system includes no subgraph searching facilities.

The agg system is a editor for hierarchical graphs and meta graphs. It is developed on the base of the GaG system of [Beyer 91]. The agg system includes graph grammer features according to the algebraic graph grammar approach including new devolopments like the single pushout approach, cf. [EhKoLö 91]. In agg one can draw graph rewriting rules. Such a rule is applied by specifying graphically the homomorphism between the left-hand side of the executed rule and the subgraph to be replaced. An automatic subgraph searching facility is planed.

A completely different approach is described in [BuGlTr 91]. This approach is suitable especially for graph rewriting systems without control structures. In such kinds of graph grammars at every moment a large number of graph rewriting rules is considered for application. [BuGlTr 91] adapted the idea of the RETE matching algorithm which is used for forward chaining (string) rewriting systems. This algorithm considers all left-hand sides of all rules in parallel. During the execution of a rewrite step for every basic graph changing operation (e.g., inserting or deleting an edge / node) its influence to the applicability of a rule in this part of the graph is computed and stored incrementally. So, after a rewriting step the set of all possible matches for all rules is already computed. This algorithm requires an large amount of book-

1. These undo / redo capabilities are used to execute the nondeterministic control structures of PROGRES (cf. [ZünSchü 92])

keeping information that payes, if for every rewrite step the set of all applicable productions have to be considered e.g. in order to choose the next rule with respect to some priority remarks.

The algorithm proposed within this paper makes use of a number of special features of PROGRES, facilitating the subgraph isomorphism problem in order to achieve our goal: execution of PROGRES without efficiency drawbacks caused by the use of graph rewriting rules as basic operations. Due to our current experiences, this goal is met by our solution for the specifications of data structures and tools within the context of the IPSEN project. To meet the requirements of other application areas and to be prepared for future developments, we plan to enhance our algorithm in several directions as discussed in chapter 4.

We think that our results could be applied to other graph rewriting systems, too. The approach is suitable especially if

1. huge work graphs are considered and only small subgraphs have to be matched (for testing two huge graphs for isomorphism, other algorithms are more appropriate, cf. [Gould 88]),
2. if the work graph is stored in a data structure that supports special indexing schemes for efficient access to different graph elements that should be used within the matching task,
3. and if at any time only a small number of graph rewriting rules must be considered (like in PROGRES where we use control structures to direct the choice of an applicable production). If a large number of rules is considered in every graph rewriting step e.g. the algorithm of [BuGlTr 91] has advantages.

So far we have implemented a compiler (and an interpreter) for PROGRES that generates code using the standard search plan for the left-hand side of productions. Thus, we are able to compile and execute PROGRES specifications for rapid prototyping purposes. We just started the implementation of the algorithm proposed within this paper to gain efficient execution code for PROGRES specifications. The compiler / interpreter is part of the PROGRES environment that also comprises a structure oriented, textual / graphical editor, an incremental analyzer, and a graphical browser for the work graph. The whole system consists of 260 KLOC in common basic modules and about 70 KLOC within PROGRES specific modules.

To conclude, we hope we were able to show that it is possible to execute PROGRES specifications without any efficiency losses caused by using graphs as general purpose data structures and graph rewriting rules for modelling the operations on these graphs.

References

[Beyer 91] Beyer M.: GAG: Ein graphischer Editor für algebraische Graphgrammatiksysteme; Diplomarbeit, TU Berlin (1991)

[BuGlTr 91] Bunke H., Glauser T., Tran T.-H.: An efficient implementation of graph grammars based on the RETE matching algorithm; in [EhKrRo 91], 174-189 (1991)

[EhKoLö 91] Ehrig H., Korff M., Löwe M.: Tutorial Introduction to the Algebraic Approach of Graph Grammars Based on Double and Single Pushouts; in [EhKrRo 91], 24-37 (1991)

[EhKrRo 91] Ehrig H., Kreowski H.-J., Rozenberg G. (Eds.): Proc. 4th Int. Workshop on Graph-Grammars and their Application to Computer Science; LNCS 532, Springer-Verlag, Berlin (1991)

[EnJaSchä 88] Engels G., Janning Th., Schäfer W.: A Highly Integrated Tool Set for Program Development Support; Proc. ACM SIGSMALL Conf., 1-10, ACM Press (1988)

[EngSchä 85] Engels G., Schäfer W.: Graph Grammar Engineering: A Method Used for the Development of an Integrated Programming Support Environment; in Ehrig et al.(Eds.): Proc. TAPSOFT '85; LNCS 186, 179-193, Springer-Verlag, Berlin (1985)

[Gould 88] Gould R.: GRAPH THEORY; The Benjamin / Cummings Publishing Company, Menlo Park, California (1988)

[HaraElli 80] Haralick R.M., Elliot G.L.: Increasing Tree Search Efficiency for Constraint Satisfaction Problems; Artificial Intelligence, Vol. 14, 263-313 (1980)

[Henten 89] Van Hentenryck P.: Constraint Satisfaction in Logic Programming; MIT Press, London (1989)

[Hims 89] Himsolt M.: GraphEd: An Interactive Graph Editor; STACS 89, LNCS 349, Springer-Verlag, Berlin (1989)

[LeReJa 93] Lee H.-Y., Reid Th. F., Jarzabek St. (eds): CASE '93 Proc. 6th Int. Workshop on Computer-Aided Software Enginneering; IEEE Computer Society Press (1993)

[Lewe 88] Lewerentz C.: Extended Programming in the Large in a Software Development Environment; in: Proc. 3rd. Int. ACM SIGPLAN/SIGSOFT Symp. on Practical Software Engineering Environments; SIGSOFT Notes 13/5, 173-182, acm press (1988)

[KiSchWe 92] Kiesel N., Schürr A., Westfechtel B.: Design and Evaluation of GRAS, a Graph-Oriented Database System for Engineering Applications; in: [LeReJa 93], 272-286 (1992)

[Koss 92] Kossing P.: Modellierung abstrakter Syntaxgraphen zu normierten EBNF's; Master's Thesis, RWTH Aachen (1992)

[MeNä 89] Mehlhorn K., Näher S.: LEDA, a Library of Efficient Data Types and Algorithms; Technical Report, TR A 04/89, FB10, Universität des Saarlands, Saarbrücken (1989)

[Nagl 79] Nagl M.: Graph-Grammatiken; Vieweg Verlag (1979)

[SchmiBer 92] Schmidt G., Berghammer R. (Eds.): Graph-Theoretic Concepts in Computer Science; WG '91, LNCS 570, Springer-Verlag, Berlin(1992)

[Volk 91] Volkmann L.: Graphen und Digraphen: Eine Einführung in die Graphentheorie; Springer-Verlag, Berlin (1991)

[Schürr 91] Schürr A.: Operationales Spezifizieren mit programmierten Graphersetzungssystemen; Ph.D. Thesis, RWTH Aachen, Deutscher Universitäts Verlag, Wiesbaden (1991)

[Schürr 91b] Schürr A.: PROGRESS: A VHL-Language Based on Graph Grammars; in: [EhKrRo 91], 641-659 (1991)

[ZünSchü 92] Zündorf A., Schürr A.: Nondeterministic Control Structures for Graph Rewriting Systems; in: [SchmiBer 92], 48-62 (1992)