# A First Experimental Evaluation of Search Plan Driven Graph Pattern Matching

Gernot Veit Batz, Moritz Kroll, and Rubino Geiß

Universität Karlsruhe (TH), 76131 Karlsruhe, Germany
`batz@ira.uka.de`, {`moritz,rubino`}`@ipd.info.uni-karlsruhe.de`

**Abstract.** With graph pattern matching the field of graph transformation (GT) includes an NP-complete subtask. But for real-life applications it is essential that graph pattern matching is performed as fast as possible. This challenge has been attacked by the approach of search plan driven, host-graph-sensitive (also known as model-sensitive) graph pattern matching. To our knowledge no experimental evaluation of this approach has been published yet. We performed first experiments regarding the runtime performance using the well-known GT benchmark introduced by Varró et al. as well as an example from compiler construction. Moreover we present an improved cost model and heuristics for search plans and their generation.

**Keywords.** graph transformation, graph pattern matching, subgraph isomorphism problem, search plan driven, host-graph-sensitive, model-sensitive, heuristic optimization, experiment

## 1 Introduction

In graph transformation (GT) [1] declarative rules are used to specify the alteration of graphical structures. The application of those transformations requires *graph pattern matching*[1] which is an NP-complete problem (see Garey and Johnson [2], problem GT48). However, real-life applications demand that transformation steps are done within a reasonable amount of time. For this reason efficient graph pattern matching is one of the important issues in GT.

This challenge has been attacked by the heuristically optimizing approach of *search plan driven, host-graph-sensitive*[2] graph pattern matching [3–7]. The key idea of this method is to represent possible matching strategies by so-called *search plans*, which are sequences of *primitive matching operations* dealing with single graph elements. A cost model assigns costs to all matching operations and search plans. This makes search plan generation an optimization problem and allows the generation of matching strategies at runtime depending on the current *host graph*. The required statistical information about the host graph can

[1] Also known as "subgraph matching" or the "subgraph isomorphism problem".
[2] In the following, we omit the term "host-graph-sensitive" (which is also referred to as "model-sensitive") for conciseness.

be obtained by an analysis of the host graph in linear time. The actual graph pattern matching is performed by *executing* the generated search plan.

However, no experimental evaluation of this approach has been published yet. In this paper we present a first experimental case study on search plan driven graph pattern matching using our GT tool GrGen.NET [8–11] (see Section 3). As test cases we utilize the well-known GT benchmark invented by Varró et al. [12] (see Section 3.1) as well as an example taken from compiler construction (see Section 3.2).

Additionally we present an improved version of cost model and heuristics (Section 2): The old cost model and heuristics only consider the backtracking that might occur during the execution of the primitive operations. The new cost model and heuristics deal with the effort raised by the operations themselves as well. So, in our experiments we evaluate the old cost model and heuristics as well as the new ones. At least for our two test cases it becomes apparent that

– the execution times of the possible search plans vary greatly (otherwise, there would be no room for optimization at all),
– the new cost model yields a reasonable picture of the real execution times,
– the search plans generated by the improved heuristics are quite good,
– the old cost model and heuristics perform partly worse (but still tolerable).

## 2 Search Plan Driven Graph Pattern Matching

In this section we give a brief introduction to search plan driven graph pattern matching. Other and partly more detailed discussions that describe different flavors of the approach have been provided by Batz [4], Varró et al. [5], and Horváth et al. [7]. Here, we present an improved version of the approach, which has not been published before.

Given a GT rule $L \rightsquigarrow R$ and a host graph $H$, we want to find a *match* (i.e. an occurrence) of $L$ in $H$. We do this in three major steps:

1. To obtain the statistical information needed by the cost model we perform an analysis of the host graph $H$ (this might be avoided by using domain-specific knowledge).
2. Using the information provided by the analysis, we generate a search plan $P$ of preferably low cost.
3. We perform the actual graph pattern matching by executing $P$.

### 2.1 Search Plans and their Execution

**Primitive Matching Operations.** A search plan $P = \langle o_1, \ldots, o_k \rangle$ is a sequence of primitive matching operations $o_i$, which are atomic search actions binding exactly one pattern element to an appropriate[3] unbound element of the host graph. We distinguish five kinds of primitive matching operations:

[3] In this context "appropriate" means that nodes are bound to nodes, edges to edges, and that the types of the participants are compatible.

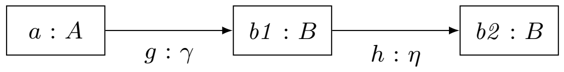

**Fig. 1.** A simple pattern graph.

1. A *lookup operation* $\mathsf{lkp}(x)$ binds the pattern element $x$ to *any* appropriate element of the host graph.
2. An *incoming edge operation* $\mathsf{in}(v, e)$ binds the pattern edge $e$, that must be an incoming edge of the already bound pattern node $v$, to an appropriate unbound edge of the host graph, that is incoming on the current host graph partner of $v$.
3. An *outgoing edge operation* $\mathsf{out}(v, e)$ works analogously to $\mathsf{in}(v, e)$ but deals with an *outgoing* edge.
4. A *get source operation* $\mathsf{src}(e)$ binds the source node of an already bound pattern edge $e$ to the source node of the current host graph partner of $e$.
5. A *get target operation* $\mathsf{tgt}(e)$ works analogously to $\mathsf{src}(e)$ but deals with the *target* node.

**Search Plans.** Now we are able to give an exact definition of search plans: A sequence $P = \langle o_1, \ldots, o_k \rangle$ of primitive matching operations is called a *valid search plan* for $L$ if the following two conditions hold:

1. Every element of the pattern graph $L$ is treated exactly once.
2. If an operation $o_i$ requires, that a pattern element is already bound, then this element must be bound by one of the preceding operations $o_1, \ldots, o_{i-1}$.

Consider, for example, the small pattern graph shown in Figure 1. Then the operation sequences

$$P_1 := \big\langle \mathsf{lkp}(b2), \mathsf{in}(b2, h), \mathsf{src}(h), \mathsf{in}(b1, g), \mathsf{src}(g) \big\rangle\,,$$
$$P_2 := \big\langle \mathsf{lkp}(h), \mathsf{tgt}(h), \mathsf{src}(h), \mathsf{lkp}(a), \mathsf{out}(a, g) \big\rangle$$

are valid search plans for this pattern graph.

**Executing a Search Plan.** Given a valid search plan $P = \langle o_1, \ldots, o_k \rangle$, we can perform the actual graph pattern matching by *executing* $P$. This means, that a match of $L$ in $H$ is stepwise constructed by executing one primitive operation of $P$ after another. Whenever a primitive operation $o_i$ is executed successfully, the current partial match is extended by a new binding of the pattern element the operation $o_i$ refers to. When the whole search plan $P$ is executed successfully, a full match of $L$ in $H$ is found.

When we execute a primitive operation that deals with a pattern element $x$, it might happen that $x$ can be bound to more than one appropriate element of the host graph. In this case we choose only one of the possible elements. The alternatives can be processed later by backtracking, if necessary. This, of course, may lead to an execution time that is exponential in the number of pattern elements.

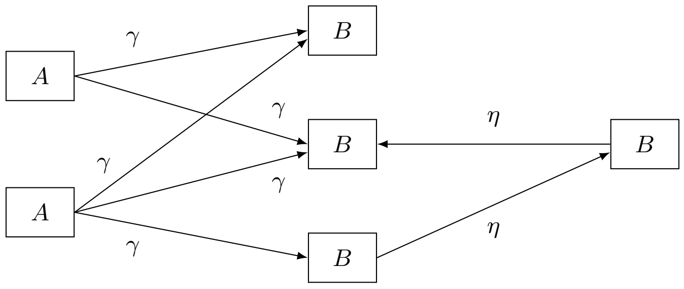

**Fig. 2.** A simple host graph.

**Implicit Checks.** Consider, for example, the following two operation sequences, which are—according to the above definition—both valid search plans:

$$P_3 := \big\langle \mathsf{lkp}(b2), \mathsf{in}(b2, h), \mathsf{lkp}(b1), \mathsf{lkp}(a), \mathsf{lkp}(g) \big\rangle$$
$$P_4 := \big\langle \mathsf{lkp}(a), \mathsf{lkp}(b2), \mathsf{out}(a, g), \mathsf{in}(b2, h), \mathsf{src}(h) \big\rangle$$

While executing $P_3$ the operation $\mathsf{lkp}(b1)$ is performed *after* $h$ is already bound by a previous operation. As a consequence, the only remaining choice to bind *b1* is the source node of the current host graph partner of $h$. But as a lookup operation chooses *any* appropriate element, a chosen node is probably not that very same source node. Thus, an appropriate check must be performed implicitly during the execution of a lookup operation. This applies analogously to the operation $\mathsf{src}(h)$ in $P_4$ as well as to edge lookups like $\mathsf{lkp}(g)$ in $P_3$. More precisely, if a primitive operation binds a pattern element $x$ to a partner $y$ in $H$, it must be checked implicitly, whether the current partners of all already bound pattern elements incident to $x$ are correctly linked to $y$.[4]

## 2.2 The Cost Model

As exponential execution time is a consequence of backtracking, the cost model must assign a higher cost to a search plan whose execution requires more backtracking. But in some cases the running time of an operation (without considering backtracking) may be even more siginficant.

**The Cost of Primitive Matching Operations.** For a primitive operation $o$ we define two different cost measures $b(o)$ and $t(o)$ that estimate the *backtracking* and the execution *time* raised by $o$ itself, respectively. Backtracking is caused

[4] Of course, $P_3$ and $P_4$ are quite stupid search plans. But allowing such awkward situations and performing implicit checks simplifies the theory a lot in the sense that the set of valid operation selections directly corresponds to the set of directed spanning trees (DSTs) in the plan graph and in the sense that all enumerations of such a DST are valid search plans (see Section 2.3).

by primitive operations that involve multiple possible bindings. Accordingly, $b(o)$ estimates the number of choices that are possible during the execution of $o$. To predict the time spent during the execution of $o$ itself, $t(o)$ estimates the number of host graph elements processed during the execution of $o$. Both measures should reflect the underlying data structures, of course. In this paper we define $b$ and $t$ in a way that they are suited to the implementation provided by our tool GrGen.NET. Note that in the following $b(o) \leq t(o)$ always holds for any operation $o$ by definition.

Before going on we define some notation first: For any pattern element $x$ let $\#_x$ be the number of elements in $H$ having a type compatible with $x$. For a pattern edge $e$, let $M_e$ be the number of edges in $H$ having a compatible edge type as well as compatibly typed source and target nodes. Lastly, let $S_e$ and $T_e$ be the number of edges in $H$ having a compatible edge type as well as a compatibly typed source node or target node, respectively (thus, in case of $S_e$ and $T_e$ "the other" node may have an incompatible type). Of course, $\#_e \geq S_e \geq M_e$ and $\#_e \geq T_e \geq M_e$ always holds.

*Node Lookups.* Consider an operation $\mathsf{lkp}(v)$ for a pattern node $v$. Then the number of choices raised by $\mathsf{lkp}(v)$ as well as the number of host graph elements processed during the execution of $\mathsf{lkp}(v)$ is at most $\#_v$. Accordingly, we assign $t(\mathsf{lkp}(v)) := b(\mathsf{lkp}(v)) := \#_v$ as costs. For the pattern graph of Figure 1 and the host graph of Figure 2 we have $t(\mathsf{lkp}(b1)) = b(\mathsf{lkp}(b1)) = 4$, for example.

*Edge Lookups.* An edge lookup $\mathsf{lkp}(e)$ for a pattern edge $e$ works analogously to a node lookup $\mathsf{lkp}(v)$. But sooner or later, the source and target node of $e$ must also be matched. This further reduces the number of possible choices for $e$. For example, if an edge $f$ in $H$ has a compatible edge type, but an incompatible typed source node, an appropriate operation $\mathsf{lkp}(e)$ will be executed successfully but a subsequent operation $\mathsf{src}(e)$ will fail. So, we assign $t(\mathsf{lkp}(e)) := \#_e$ and $b(\mathsf{lkp}(e)) := M_e$. As a consequence $\mathsf{src}$ and $\mathsf{tgt}$ should occur as early as possible in a search plan of course.

*Incoming and Outgoing Edge Operations.* We define $b(\mathsf{out}(v, e)) := M_e/\#_v$, which is the average number of appropriate incident to the current host graph partner of $v$. In contrast, we set $t(\mathsf{out}(v, e)) := S_e/\#_v$, which is the average number of outgoing edges incident to the current host graph partner regardless whether they are appropriate or not. Analogously, we assign $b(\mathsf{in}(v, e)) := M_e/\#_v$ and $t(\mathsf{in}(v, e)) := T_e/\#_v$. So, for the pattern graph of Figure 1 and the host graph of Figure 2 we have $b(\mathsf{in}(b1, g)) = 5/4 = 1.25$ and $t(\mathsf{in}(b1, g)) = 7/4 = 1.75$, for example.

*Get Source and Get Target Operations.* Primitive Operations of the form $\mathsf{src}(e)$ and $\mathsf{tgt}(e)$ do *not* raise multiple choices during the matching process and process exactly one graph element during their execution. So, we assign $t(\mathsf{src}(e)) := t(\mathsf{tgt}(e)) := b(\mathsf{src}(e)) := b(\mathsf{tgt}(e)) := 1$ as costs.

**The Cost of a Search Plan.** The cost model should estimate the time needed for the execution of a search plan $P = \langle o_1, \ldots, o_k \rangle$. So, we use the cost function

$$t(P) := t_1 + b_1 t_2 + b_1 b_2 t_3 + \cdots + b_1 \cdots b_{k-1} t_k \tag{1}$$

with $b_i := b(o_i)$ and $t_i := t(o_i)$. This is in fact a weighted sum of the estimated execution times $t_1, \ldots, t_k$ of the operations $o_1, \ldots, o_k$. The weights $b_1 \cdots b_{i-1}$ with $2 \leq i \leq k$ multiplicatively accumulate the estimated backtracking that arises during the matching process. So, under the assumption that $b_1, \ldots, b_{i-1}, t_i$ are stochastically independent[5] for $1 < i \leq k$, the expected execution time is in $O(t(P))$. In the context of our second experiment (see Section 3.2) the cost function $t(P)$ does a far better job than the "traditional" cost function $b(P)$ that is defined as

$$b(P) := b_1 + b_1 b_2 + \cdots + b_1 \cdots b_k \tag{2}$$

and only estimates the backtracking raised by $P$. So, we claim that with the cost function $t(P)$ we have an improved cost model.

## 2.3 Generating a Search Plan

As we do not know efficient algorithms that construct search plans that are optimal according to $t(P)$ or $b(P)$, we have to settle for heuristic methods. Here, we consider two heuristic methods which we call BacktrackingOnly and BacktrackingLookup. The first of the two only minimizes the backtracking, which actually addresses the cost function $b(P)$. The second tries to minimize the backtracking *and* the execution time of the operations themselves and, hence, addresses the cost function $t(P)$. However, the second one works exactly as the first one except for a little modification. So, we firstly describe the BacktrackingOnly heuristics and characterize the modifications done by BacktrackingLookup afterwards.

## 2.4 The BacktrackingOnly Heuristics

The BacktrackingOnly heuristics works in two phases: the *operation selection* and the *operation ordering*. But before these two phases can be performed, we have to generate a structure that we call *plan graph* first.

**The Plan Graph.** For a pattern graph $L$ the plan graph $\tilde{L}$ is defined as follows:

1. For every element $x$ of $L$ the plan graph $\tilde{L}$ contains a node $v_x$ representing $x$. All these nodes are labeled with the name of the pattern element they represent. Additionally, $\tilde{L}$ contains a special *root* node.
2. For every element $x$ of $L$ the plan graph $\tilde{L}$ contains an edge $f_x$ that leads from the root node to the node that represents $x$ in $\tilde{L}$. It is labeled with $\mathsf{lkp}$ and represents the operation $\mathsf{lkp}(x)$.

[5] Stochastic independence might not hold, but it provides a good intuition of what is happening.

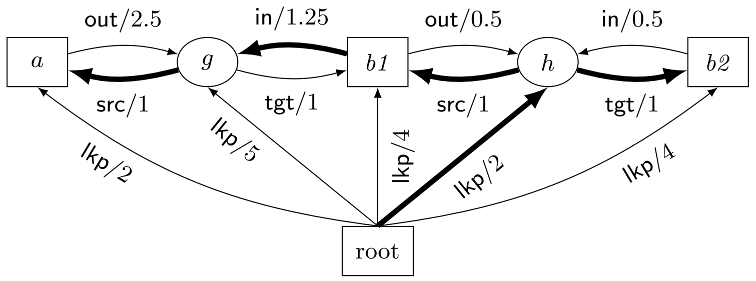

**Fig. 3.** The plan graph for the pattern graph of Figure 1 with estimated backtracking costs induced by the host graph of Figure 2. A multiplicative MDST is denoted by thick edges. It has a total cost of $2 * 1 * 1 * 1.25 * 1 = 2.5$.

3. For every edge $e$ in $L$ let $v_e$ be the node representing $e$ in $\tilde{L}$. Then $\tilde{L}$ contains four further edges incident to $v_e$:
   - An edge labeled with $\mathsf{tgt}$ leading from $v_e$ to the node representing the target node $t$ of $e$ and an edge in reverse direction labeled with $\mathsf{in}$. These edges represent the operations $\mathsf{tgt}(e)$ and $\mathsf{in}(t, e)$, respectively.
   - An edge labeled with $\mathsf{src}$ leading from $v_e$ to the node representing the source node $s$ of $e$ and an edge in reverse direction labeled with $\mathsf{out}$. These edges represent the operations $\mathsf{src}(e)$ and $\mathsf{out}(s, e)$, respectively.

As every edge of the plan graph represents a primitive matching operation $o$, we can assign the estimated backtracking $b(o)$ as a cost to the corresponding edge. In this way the plan graph becomes a weighted directed graph. Figure 3 shows the plan graph for the pattern graph of Figure 1 while the edge weights represent the estimated backtracking induced by the host graph of Figure 2.

**Operation Selection.** Looking at the cost function $b(P)$ in equation (2) we see that $b_1 b_2 \cdots b_k$ is the most significant term. By minimizing this term, the BacktrackingOnly heuristics tries to minimize $b(P)$. As every cost $b_i$ appears in $b_1 b_2 \cdots b_k$ exactly once, this corresponds to finding a *minimal operation selection*[6] $S = \{o_1, \ldots, o_k\}$. This is not a trivial task but luckily there is a one-to-one correspondence between the set of valid operation selections and the set of directed spanning trees (DSTs) in the plan graph[7]. Moreover, the corresponding DST of a *minimal* operation selection is just a *minimum* directed spanning tree (MDST) according to multiplicatively[8] computed total costs. Such an MDST can be computed in polynomial time by the Edmonds/Chu-Liu algorithm [13, 14]. In Figure 3 an example of a multiplicative MDST is denoted by thick edges.

[6] Minimal in terms of the backtracking.

[7] This has been shown by Batz [4].

[8] Typically, the cost of a MDST is computed in terms of a sum, but it can also be computed in terms of a product. As logarithms are strictly increasing functions from $\mathbb{R}_{>0}$ to $\mathbb{R}$ and $\log(ab) = \log a + \log b$ holds, the computation of a multiplicative MDST can be reduced to the computation of an additive MDST using the logarithm.

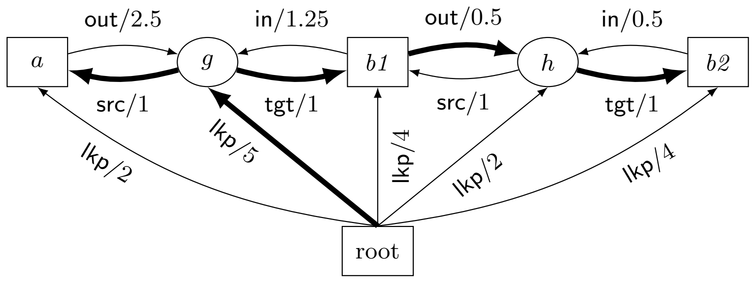

**Fig. 4.** The same plan graph as the one of Figure 3. But here another multiplicative MDST which involves edge costs between 0 and 1 is shown. Its total cost is computed by $5 * 1 * 1 * 0.5 * 1 = 2.5$.

**Operation Ordering.** Having found a minimal operation selection $S$ we have to build a valid search plan from the operations in $S$. This can be done very simply by traversing[9] the corresponding MDST starting from the root node. During the traversal we successively emit the operation represented by each edge. All operation sequences generated in this way are valid search plans.

However, we do not only want *some* valid search plan but a "good" one. Therefore, we traverse the plan graph in a *best-first* manner. That means that we prefer edges of minimal cost. To understand this, look at equation (2). Obviously an operation has more impact on the overall cost the earlier it occurs in a search plan. So, we place the cheap operations possibly early and the expensive ones possibly late. Moreover, all operations of kind src and tgt are placed *as soon as possible*. This is because src and tgt operations rise costs next to no expense but a pretended match might be exposed earlier.

**Non-Forgetful Operation Selection.** As the cost of an operation selection $S = \{o_1, \dots, o_k\}$ we could assign the product $b_1 b_2 \cdots b_k$. However, this would make the selection process "forgetful": As a plan graph may contain edge costs between 0 and 1 and as the total cost of a DST is computed in terms of a product[10] each such edge *reduces* the total cost of a DST. As a result, information about intense intermediate backtracking may be destroyed. Consider for example, the MDST shown in Figure 4: Though the cost $b(\mathsf{lkp}(g)) = 5$ indicates that intense backtracking may arise, the cost $b(\mathsf{out}(\mathit{b1}, h)) = 0.5$ taints the result as $5 * 0.5 = 2.5$ holds. So, we assign

$$\mathrm{BO}_*(S) := \prod_{o \in S} \max\{1, b(o)\} \tag{3}$$

[9] A traversal of a directed graph is an enumeration of its edges where an edge must not be visited unless its source node is a given start node or the target node of an already visited edge.

[10] In the context of the corresponding, additive MDST problem, we obtain by logarithmizing, a multiplicative edge cost between 0 and 1 becomes a negative edge cost.

as the cost of an operation selection $S$. If we use this "non-forgetful" cost function, the DST shown in Figure 4 has no longer minimal cost.

Varró et al. [5] use a different cost function for the operation selection, namely

$$\mathrm{BO}_{+}(S) := \sum_{o \in S} b(o)\,. \tag{4}$$

Obviously, $\mathrm{BO}_{+}$ is also a "non-forgetful" cost function minimizing backtracking. But using $\mathrm{BO}_{+}$ the operation selection does not directly minimize the product $b_1 \cdots b_k$ anymore, which might distort the result. On the other hand, by $\mathrm{BO}_{+}$ costs between 0 and 1 are *not* lifted to 1. In this way the first-fail principle is included in the operation selection, which is the pro of this technique.

So, there are two variants of the BacktrackingOnly heuristics: The first one uses the function $\mathrm{BO}_{*}$ on the operation selection and the second one $\mathrm{BO}_{+}$ instead.

### 2.5 The BacktrackingLookup Heuristics

Unlike BacktrackingOnly, the BacktrackingLookup heuristics not only minimizes backtracking but also deals with estimated execution times. This is achieved only by a little modification of BacktrackingOnly: Go back to the plan graph as defined in Section 2.4 and consider all edges that correspond to an operation $o := \mathsf{lkp}(e)$ for a pattern edge $e$. For all these edges replace the cost $b(o)$ by $t(o)$. Having done this we proceed exactly as in case of BacktrackingOnly with cost function $\mathrm{BO}_{*}$. So, the operation selection of BacktrackingLookup minimizes a cost function $\mathrm{BL}(S)$ which is a modified version of $\mathrm{BO}_{*}(S)$: For edge lookups we minimize the estimated execution time instead of the backtracking. At least for our second test case (see Section 3.2) this is a real step forward. For our first test case (see Section 3.1) both methods behave similar well.

## 3 Experimental Results

The essential idea of our experiments is to generate *all* search plans possible for a pattern graph $L$ while measuring the execution time for each search plan. This way we experimentally validate our cost model and our heuristics. Additionally we consider the heuristics proposed by Varró. As test cases we use the well-known STS Mutex benchmark introduced by Varró et al. [12] (see Section 3.1) as well as an example taken from compiler construction (see Section 3.2). All measurements are carried out using our GT tool GrGen.NET [9–11], which implements the BacktrackingLookup heuristics described in Section 2, as well as our benchmarking tool spBench which is included in the 1.3 release of GrGen.NET. The underlying platform is an AMD Athlon XP 3000+ with 1 GByte main memory that runs with Windows XP and .NET 2.0.

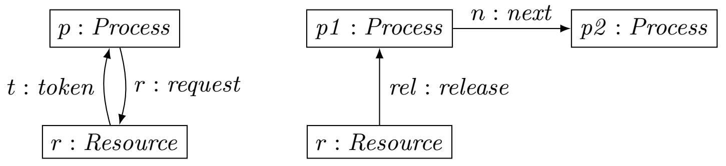

**Fig. 5.** The pattern graph of the *takeRule* (left) and the *giveRule* (right).

**Displaying the Results.** In each diagram we relate the different cost measures ($b(P)$, $t(P)$, $\mathrm{BO}_*(S)$, $\mathrm{BO}_+(S)$, and $\mathrm{BL}(S)$) with the execution time each. On the horizontal axis we display the cost, on the vertical axis the detected execution time. Note that we use a logarithmic scale for both axes; for technical reasons both axis show the logarithmized values.

However, some points occur in the data set very often. Under these circumstances scatter plots loose very much of their expressiveness. For this reason, we do not chart every single point. Instead, we divide the plane into hexagons and draw only those hexagons that contain at least one point. The more points a hexagon contains, the darker we draw it.

**Goal.** We are primarily interested in the quality of a heuristics: It is good, if and only if plans with low costs actually have low execution times, i.e., we want to see that the leftmost points in the diagram have low execution times. Besides this primary goal it would be nice to have a good overall correlation between costs and execution time—both for a heuristics and for a cost model.

### 3.1 First Experiment: The STS Mutex Benchmark

**Background.** The *STS Mutex benchmark* models a mutual exclusion scenario of $N$ processes trying to access a single resource (in this experiment we choose $N = 10,000$). The $N$ processes are represented by $N$ nodes of type *Process*, which are connected by $N$ edges of type *next* such that the processes form a ring. The single resource is represented by a node of type *Resource*. This structure is built first by certain rules. Afterwards other rules insert and delete edges of different types all over the graph. More details can be found in a technical report by Varró et al. [12].

**Experimental Setup.** In this experiment we perform an exhaustive exploration of all search plans only for the *takeRule* and the *giveRule*. Figure 5 shows the pattern graphs of these rules.

In a first pass we execute the whole benchmark for every search plan that is possible for the *takeRule* and measure the execution time with a timeout of 10 seconds each. For the *giveRule* we use a fixed search plan, namely the one provided by the heuristics. The results are displayed in Figure 6. The plot on the right, which relates $\mathrm{BL}(S)$ with the execution time, shows the desired behavior:

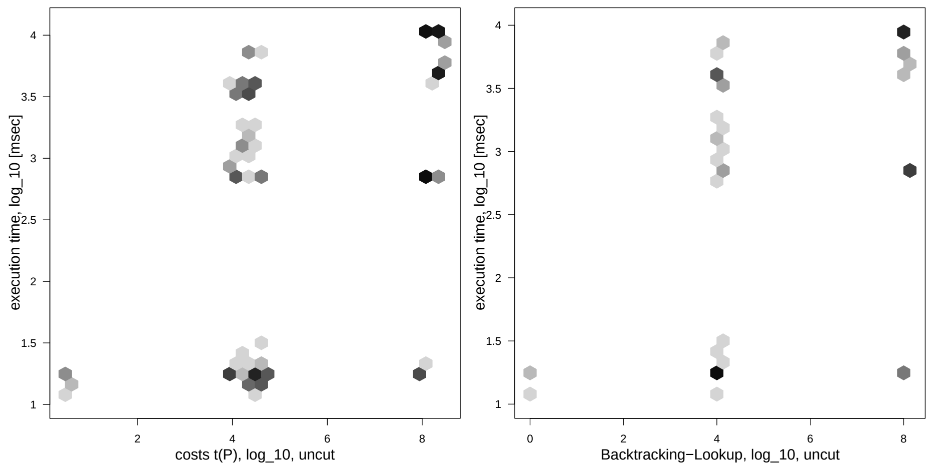

**Fig. 6.** Plotted results for all search plans possible for the *takeRule*.

Points in the far left have low execution time. Hence, BACKTRACKINGLOOKUP chooses one of the most fast executing plans possible. In the plot on the left we see that the actual cost function $t(P)$ shows a little more diversified behavior than $\mathrm{BL}(S)$. However, due to the very simple structure of the STS benchmark the figures only show a simple level-wise distribution.

In a second pass we performed the analogous experiment by checking all possible search plans for the *giveRule*. The resulting plots are similar and hence omitted.

## 3.2 Second Experiment: Finding Loop Counters

**Background.** In compiler construction internal intermediate representations (IRs) are used for programs. Modern IRs are graph based and represent programs as dependency or flow graphs. In this experiment we use an adoption of an IR called FIRM [15] for our GT tool GrGen.NET. FIRM uses dependency graphs, and fulfills the so-called *SSA*[11] property. IRs with this property represent programs in a way that each variable has exactly one occurrence as a left-hand-side of an assignment. This requires the so-called *Φ-operation* that models alternative dataflows as they occur in the context of conditions and loops, for example.

**Experimental Setup.** The essential idea of this second experiment is to find loop counters in C programs. As a host graph we use the IR graph of the C function `RenderTiles()` in `RenderWorld.c` (revision 1328) taken from the open source project JAGGED ALLIANCE 2 – STRACCIATELLA [16]. The function `RenderTiles()` has 1,418 lines of code; the corresponding IR graph consists of

[11] SSA stands for *static single assignment*.

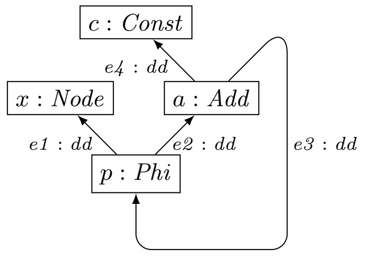

**Fig. 7.** A pattern graph that specifies data dependencies typical for a loop counter.

4,705 nodes and 16,714 edges. As a pattern graph we use the one shown in Figure 7, which shows a pattern typical for a loop counter. To understand this, consider the following code fragment written in C:

```
int i;
for(i = x; i < 100; i = i + 3) { /*...*/ }
```

The initial value `x` of the loop counter `i` is represented by the pattern node $x$. The pattern node $c$ represents the constant used to increase `i`, which is `3` in the example. The pattern node $a$ of type *Add* denotes the operator '`+`' in `i = i + 3`. The loop counter `i` has no corresponding node in the pattern graph[12]. The pattern node $p$ of type *Phi* denotes the $\Phi$-operation mentioned above. It is needed because a loop induces alternative control paths: When the loop is entered for the first time, `i` has the initial value `x`. This is what the edge *e1* stands for. At the beginning of each subsequent iteration the loop counter `i` is incremented by $c$, which is expressed by the edge *e2*.

The pattern graph used in this second experiment has far too many search plans to execute them all. In fact, there are exactly 4,441,472 search plans. With a timeout of, for example, 5 seconds (and under the assumption that most search plans need 5 seconds or more) the execution of the search plans would take about 250 days (without the compilation time needed for the search plans) which is far too long of course. However, we reduce the set of search plans dramatically by restricting ourselves to search plans where `src` and `tgt` operations are scheduled as soon as possible—every sensible search plan should be like that (see Section 2.3). Moreover, we only consider search plans with a limited number of lookup operations. Table 1 shows the number of search plans with at most $N$ lookup operations for $N \in \{1, 2, \ldots, 8\}$—as the pattern graph has eight elements, at most eight lookups are possible.

In this experiment we choose $N \leq 2$ and perform an exhaustive search for *all* matches for each of the 3,424 search plans (repeated a 50 times each). The timeout in this second experiment is also 10 seconds. The results are displayed in Figure 8. The two plots at the top show the behavior of $b(P)$ and $t(P)$. Both cost functions exhibit a visible correlation between costs and execution time.

[12] In graph based SSA, variables have no explicit representation in general.

| $N$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| all SPs | 1,408 | 56,448 | 460,544 | 1,592,192 | 3,084,032 | 4,078,592 | 4,401,152 | 4,441,472 |
| ASAP SPs | 160 | 3,424 | 25,000 | 83,312 | 152,464 | 192,768 | 204,216 | 205,488 |

**Table 1.** Number of search plans with at most $N$ lookup operations for the pattern graph of Figure 7. The middle row contains the numbers of *all* such search plans, the lower row the numbers when `src` and `tgt` operations appear *as soon as possible.*

However, $b(P)$ reveals a great weakness: Several search plans with low cost have comparatively high execution time. At this regard $t(P)$ is much better. This is also reflected by the corresponding heuristic methods (see the two plots in the middle): In the plot on the left (BacktrackingOnly) there are leftmost points with comparatively high execution times, too. In contrast, the plot on the right (BacktrackingLookup) shows the desired behavior that minimal cost corresponds to quite low execution times.

The two plots at the bottom are included mainly for completeness. The left of the two shows the variant of BacktrackingOnly that uses Varró's cost function $\text{BO}_+(S)$. It shows a similar behavior as $\text{BO}_*(S)$ and—this is most important—those undesirable leftmost points with high execution times are also present. The bottom right plot actually shows the same as the plot directly above—with only one difference: In all plots of this second test case we omitted the points belonging to search plans that yielded a time-out (i.e., that needed 10 seconds or more). In this single plot we did not. It shows that $\text{BL}(S)$ does really not assign low costs to very slow search plans.

## 4 Related Work

The concept of search plans is not new in GT. It has already been used by Zündorf [17] in context of the early GT tool Progres. But, although he defines a sophisticated cost model, the actual search plan generation works with a rather coarse grained cost model. Moreover, the cost of a primitive operation is derived from assumptions as well as static connection assertions and not from the current host graph. Also, his approach is greedy except for the choice of the start node.

To our knowledge Dörr [18] was the first in GT who suggested a preparatory analysis of the present host graph for bunches of appropriate edges to prevent backtracking. He also suggests an approach to operation selection that is based on the computation of a DST. However, Dörr does not use a cost model. For this reason he is only able to generate a linear time search plan or no search plan at all. Moreover, lookup operations are only allowed as the first operation of a search plan. Lookups of edges are not supported at all.

Search plan driven, host-graph-sensitive graph pattern matching has originally been presented independently by Batz [3] as well as by Varró et al. [5]. Varró et al. coined the term *model-sensitive search plans* to emphasize that the search plan is generated depending on the present host graph. However, there are application domains, where the term "model" is not common. So, we propose

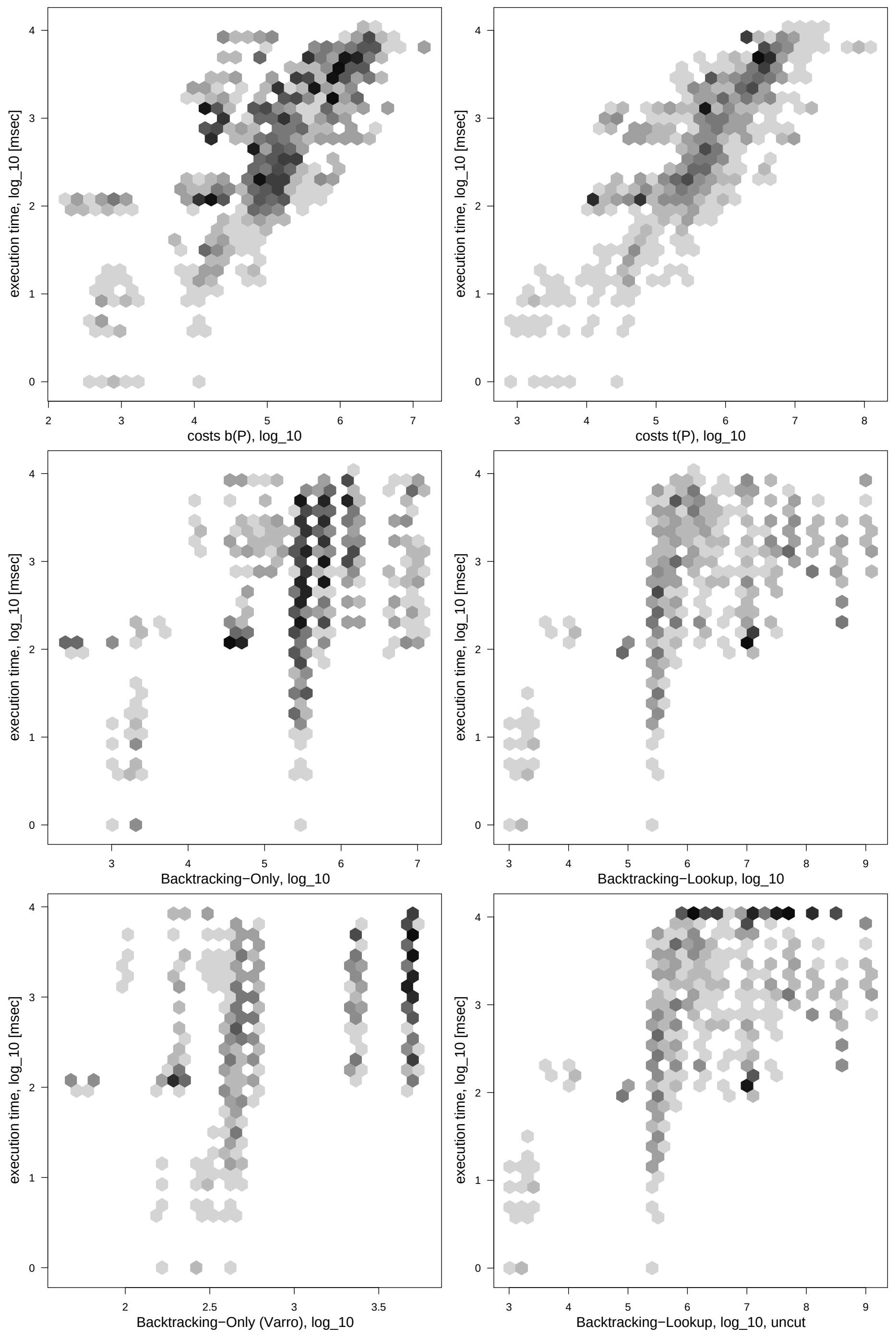

**Fig. 8.** Plotted results for our second test case "Finding Loop Counters".

the more general term "host-graph-sensitive". When search plan driven graph pattern matching arose, it did not include lookup operations for edges. This has been suggested a little later by Geiß, Batz et al. [6, 4]. The latter paper also contains a proof of the one-to-one correspondence between the set of operation selections and the set of DSTs of the plan graph.

Recently, Horvath et al. [7] suggested a generalization of plan graphs as they are defined here. Here, binding constraints on in and out operations are expressed by the direction of the corresponding edges in the plan graph. Horvath et al., in contrast, use so called adornments. These are annotations that relate binding constraints with costs. In this way an additional kind of primitive operations can be handled: It enables a more direct binding of a pattern edge that connects already bound pattern nodes. The authors announce that their approach will be implemented in the next release of VIATRA2 which is a GT based framework for model transformations.

An implementation of the BACKTRACKINGLOOKUP heuristics as described in Section 2 is included in our GT tool GRGEN.NET [9–11].

## 5 Conclusions

In this paper we presented a first experimental evaluation of search plan driven graph pattern matching (using our GT tool GRGEN.NET) as well as an improved cost model and heuristics. As test cases we used the well-known GT benchmark introduced by Varró et al. as well as an example taken from compiler construction. At least for the two test cases it became apparent that

– the execution times raised by the possible search plans vary greatly, so there is room for optimizations,
– the improved cost model reasonably reflects the real execution times,
– the search plans generated by the improved heuristics are quite good,
– the old cost model and heuristics perform partly worse.

For the future it is interesting whether better heuristic methods for the generation of search plans can be developed and how we can deal with NACs[13]. Moreover, the development of more GT benchmarks—particularly with bigger pattern graphs—would be highly desirable.

*Acknowledgements.* We want to thank all the students and researchers at IPD Goos for the creative atmosphere. Especially we want to thank Christoph Mallon. Also we want to thank Jakob Blomer, Edgar Jakumeit, Jens Müller, and Katja Weisshaupt who helped us on the preparation of this work. Moreover we thank our Professors Gerhard Goos and Peter Sanders for supporting this work. Last but not least, we want to thank the anonymous reviewers for their useful hints and suggestions.

[13] A negative application condition (NAC) is a graph associated with the pattern graph. An appropriate occurrence of a NAC in $H$ prevents the application of a GT rule.

## References

1. Heckel, R.: Graph Transformation in a Nutshell. In Bézivin, J., Heckel, R., eds.: Language Engineering for Model-Driven Software Development. Volume 04101 of Dagstuhl Seminar Proceedings., Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2004)
2. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1990)
3. Batz, G.V.: Graphersetzung für eine Zwischendarstellung im Übersetzerbau. Master's thesis, Universität Karlsruhe (2005)
4. Batz, G.V.: An Optimization Technique for Subgraph Matching Strategies. Technical Report 2006-7, Universität Karlsruhe, Fakultät für Informatik (April 2006)
5. Varró, G., Varró, D., Friedl, K.: Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans. In Karsai, G., Taentzer, G., eds.: Proc. of Int. Workshop on Graph and Model Transformation (GraMoT'05). Volume 152 of ENTCS., Tallinn, Estonia, Elsevier (September 2005) 191–205
6. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.M.: GrGen: A Fast SPO-Based Graph Rewriting Tool. In Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G., eds.: ICGT 2006. Volume 4178 of LNCS., Springer (2006) 383 – 397
7. Ákos Horváth, G.V., Varró, D.: Generic Search Plans for Matching Advanced Graph Patterns. In: Proc. of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007), Braga, Portugal, Electornic Communications of the EASST (2007) 57–68
8. Geiß, R., Kroll, M.: GrGen.NET: A Fast, Expressive, and General Purpose Graph Rewrite Tool. In: this volume
9. Kroll, M.: GrGen.NET: Portierung und Erweiterung des Graphersetzungssystems GrGen (May 2007) Studienarbeit, IPD Goos, Universität Karlsruhe.
10. Geiß, R.: GrGen.NET homepage. http://www.grgen.net (February 2008)
11. Blomer, J., Geiß, R.: The GrGen.NET User Manual. Technical Report 2007-5, Universität Karlsruhe, IPD Goos (July 2007)
12. Varró, G., Schürr, A., Varró, D.: Benchmarking for Graph Transformation. Technical report, Department of Computer Science and Information Theory, Budapest University of Technology and Economics (March 2005)
13. Edmonds, J.: Optimum Branchings. J. Res. Natl. Bureau Standards **71B** (1967) 233–240
14. Chu, Y.J., Liu, T.H.: On the shortest arborescence of a directed graph. Science Sinica **14** (1965) 1396–1400
15. Trapp, M., Lindenmaier, G., Boesler, B.: Documentation of the Intermediate Representation FIRM. Technical Report 1999-14, Universität Karlsruhe, Fakultät für Informatik (December 1999)
16. Mallon, C., Gantert, W.C.: Jagged Alliance 2 - Stracciatella (A port of the game Jagged Alliance 2 using SDL). http://ja2.dragonriders.de/ (October 2007)
17. Zündorf, A.: Graph Pattern Matching in PROGRES. In: 5th. Int. Workshop on Graph-Grammars and their Application to Computer Science. Volume 1073 of LNCS., Springer (1996) 454–468
18. Dörr, H.: Efficient Graph Rewriting and its Implementation. Volume 922 of LNCS. Springer (1995)