Graph Transformation in Constant Time

Mike Dodds

miked@cs.york.ac.uk

University of York

Sales Pitch

Graph transformation is *hard*: polynomial or NP-complete depending on your point of view.

Sales Pitch

Graph transformation is *hard*: polynomial or NP-complete depending on your point of view.

It becomes much easier if we can identify *uniquely-labelled nodes* (‘roots’) in the graph.

Sales Pitch

Graph transformation is *hard*: polynomial or NP-complete depending on your point of view.

It becomes much easier if we can identify *uniquely-labelled nodes* (‘roots’) in the graph.

Rooted graph-transformation is surprisingly powerful, despite the restrictions on it.

Sales Pitch

Graph transformation is *hard*: polynomial or NP-complete depending on your point of view.

It becomes much easier if we can identify *uniquely-labelled nodes* (‘roots’) in the graph.

Rooted graph-transformation is surprisingly powerful, despite the restrictions on it.

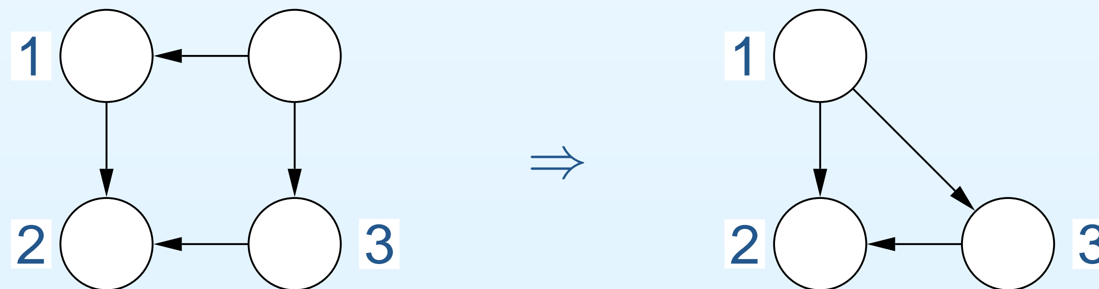
All of the following is joint work with Detlef Plump

Graph Transformation for Beginners

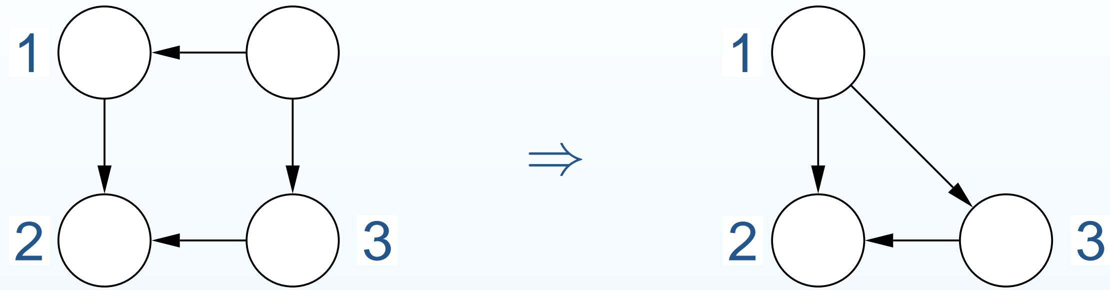
We're all familiar with string rewrite rules. We can use them to define string grammars:

$$\begin{aligned} \mathtt{S} &\Rightarrow \mathtt{aAb} \\ \mathtt{aAb} &\Rightarrow \mathtt{aaAbb} \\ \mathtt{A} &\Rightarrow \mathtt{c} \end{aligned}$$

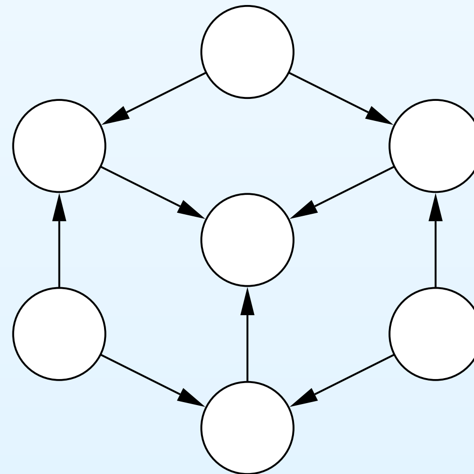
Graph-transformation rules generalise context-sensitive rewrite rules to graphs. A *graph transformation rule* consists of a left-hand side L, a right-hand side R, and an interface between the two K:



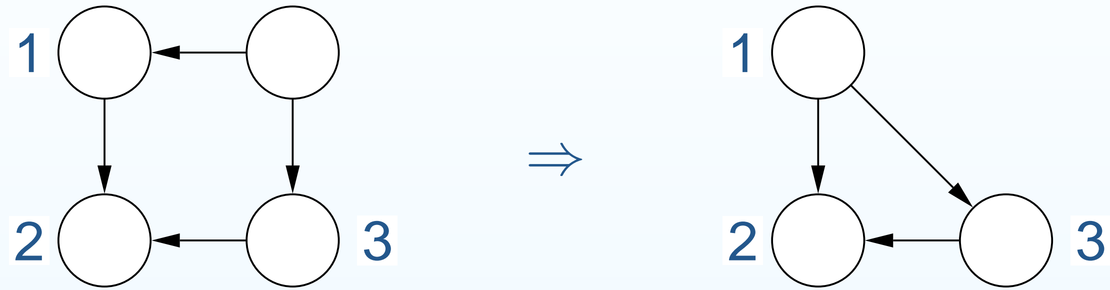
Rule Derivation



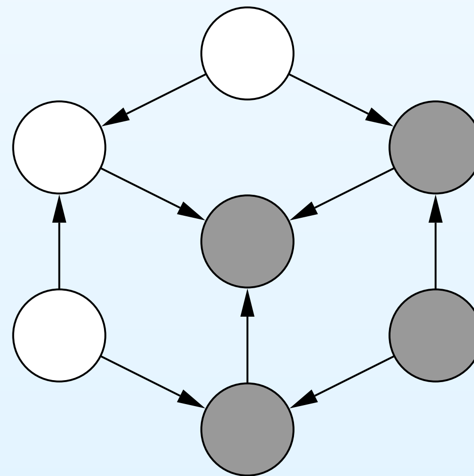
To apply the rule, let us consider a small graph:



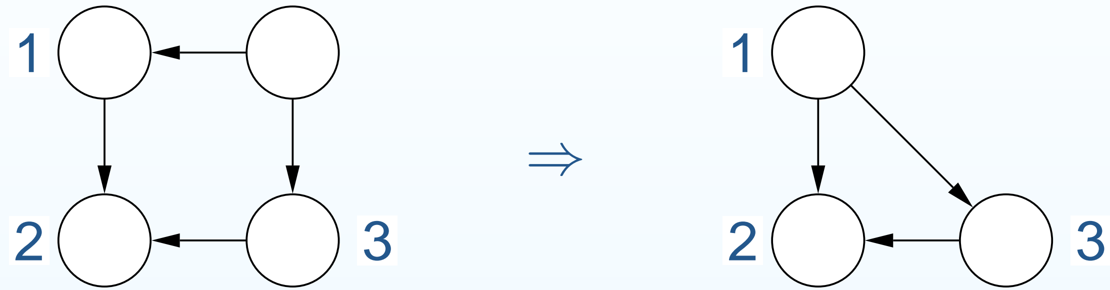
Rule Derivation



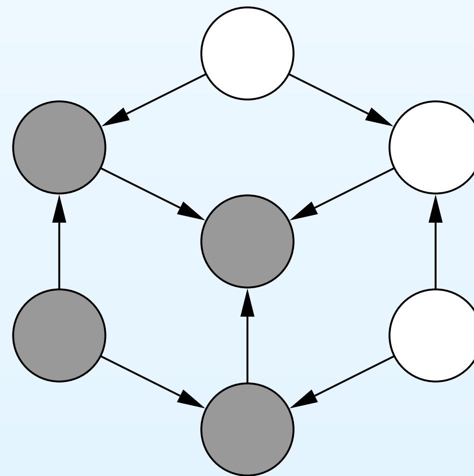
We first search for a match for the left-hand side:



Rule Derivation

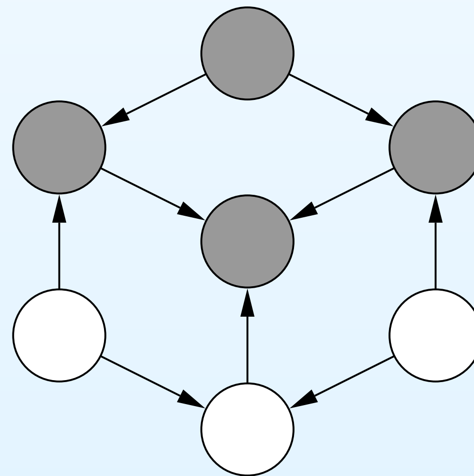


We first search for a match for the left-hand side:

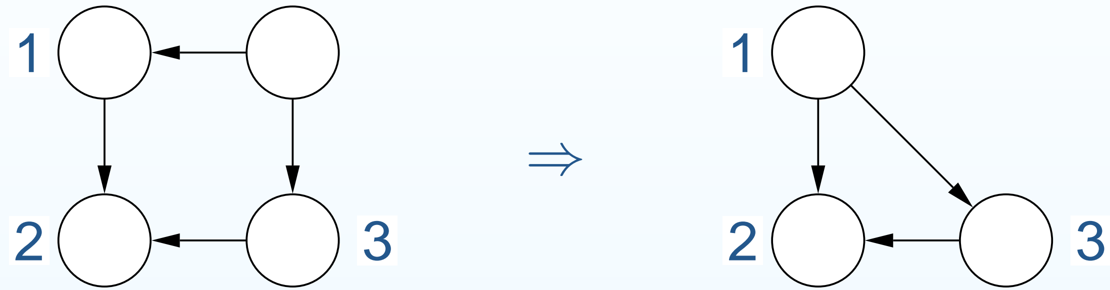


Rule Derivation

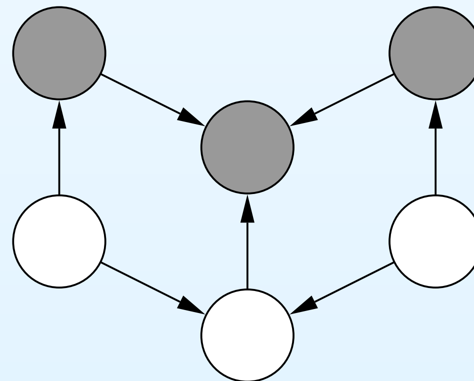
We first search for a match for the left-hand side:



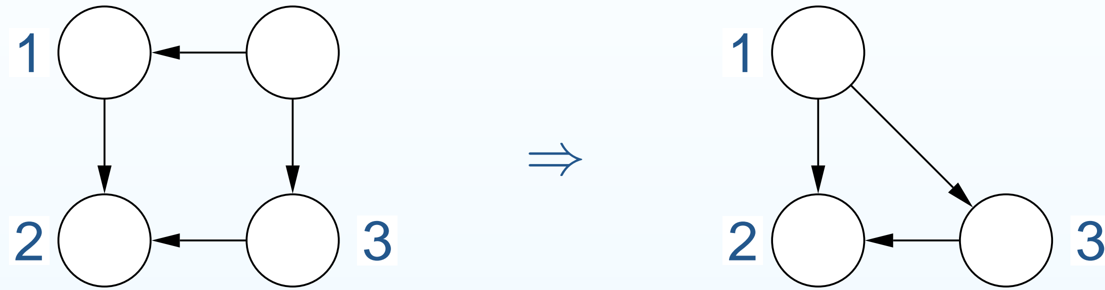
Rule Derivation



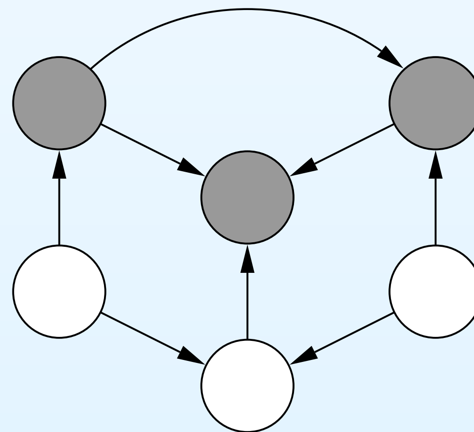
The non-interface portion is deleted:



Rule Derivation



Finally, the right-hand side is attached:



Derivation Time Complexity

Given a rule r and graph G, if r can be applied to the G resulting in graph H we write the derivation $G \Rightarrow_r H$.

Derivation is hard, because searching for a match for the left-hand side of a rule is difficult. In the worst case we have to search the entire graph for a match.

- If the rule is constant, the problem is $O(|G|^{|L|})$ – polynomial.
- With a *variable rule*, matching is equivalent to the *subgraph isomorphism problem*, which is NP-complete.

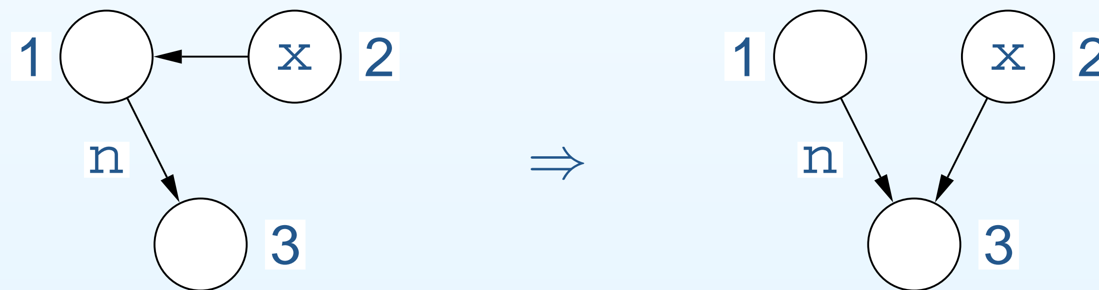
We concentrate on the fixed rule case, improving it from polynomial to constant time.

Pointer Manipulation and Graph Transformation

Pointer manipulations can be modelled as graph transformation rules. Consider this pointer assignment written in a C-like syntax:

```
x = x->n
```

We can model this as the following graph transformation rule:



Pointer Manipulation and Graph Transformation

Pointer manipulations can be modelled as graph transformation rules. Consider this pointer assignment written in a C-like syntax:

```
x = x->n
```

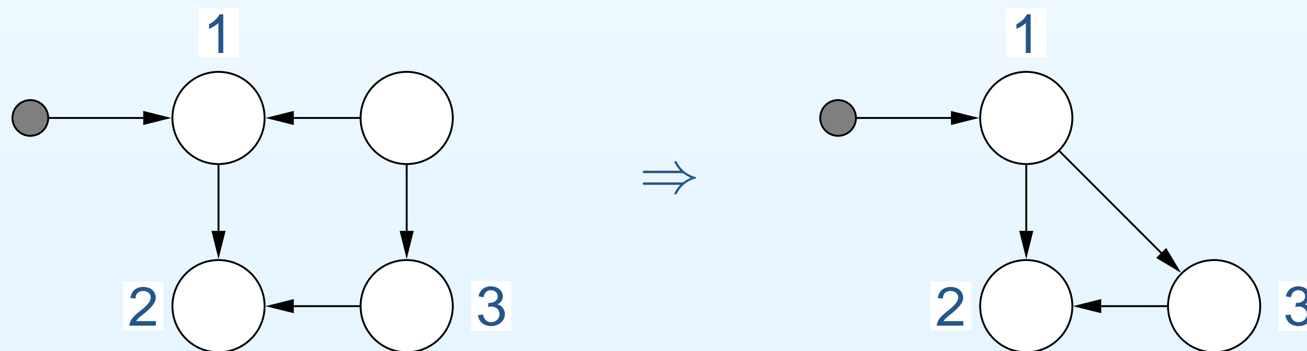
We can model this as the following graph transformation rule:

We don't need polynomial time to apply a pointer manipulation – it runs in *constant time*. What's going on?

Roots in Rewrite Rules

Answer: we know that the node labelled with ‘x’ can appear at most once in the target graph. All of the rest of the nodes can be matched deterministically by following the correct edges in the graph.

We apply this to graph rewrite rules by designating a label ϱ as the root node label which can appear at most once in the target graph. We can alter our example to make it rooted:



(Root-labelled nodes are shown as small gray nodes)

Constraints for Rooted Graph Transformation

Given a set rule r and class of graphs $\mathbb{C}$, we say that r and $\mathbb{C}$ conforms to the Rooted Graph Transformation (RGT) approach if there is an bound $b \geq 0$ and root label ϱ such that:

for rule r:

- all nodes in the left-hand side graph L are reachable from some root-labelled node

for all graphs $G \in \mathbb{C}$:

- at most one node in the G is labelled with the root label ϱ.
- the out-degree of every node in G is less than or equal to the bound b.

Constructing a Set of Matches

To apply a particular rule, we must construct a match. Formally, we construct an *injective morphism* between left-hand side L and graph G. In fact, our algorithm constructs the set of *all* matches.

0: $A_0 \Leftarrow$ morphisms only matching ϱ-labelled node
0: $E_0 \Leftarrow$ edges from root-labelled node
0: **while** all edges not matched **do**
0: pick e from E_n
0: **if** target of e has not been matched **then**
0: $A_{n+1} \Leftarrow A_n \cup$ morphisms matching e and its target
0: $E_{n+1} \Leftarrow E_n \cup$ all outgoing edges from target of e
0: **else**
0: $A_{n+1} \Leftarrow A_n \cup$ morphisms matching e
0: **end if**
0: $n \Leftarrow n + 1$
0: **end while**

Algorithm Execution Time

Each iteration of the main while-loop matches exactly one edge from the left-hand side L, so there can be at most $|E_L|$ – size of the set of left-hand side edges – iterations of the algorithm. To construct A_{n+1}, each iteration extends each morphism in A_n in at most b ways, as a result of the out-degree bound. This means that each iteration takes at worst time $b|A_n|$. This results in an overall time bound of:

$$t \leq \sum_{i=0}^{|E_L|} b^i$$

If we consider the rule and bound as fixed, both b and $|E_L|$ are constants. This results in a time complexity $O(1)$. Given a fixed rule, the other stages of application can also be completed in constant time, so we have *constant time rule derivation*.

Graph Recognition with the RGT Approach

That's all very interesting, but what can we use these restricted rules for?

Graph Recognition with the RGT Approach

That's all very interesting, but what can we use these restricted rules for?

We are interested in using the RGT approach for graph recognition. Previous work has defined a Graph Reduction System (GRS) as $\langle \Sigma, \mathcal{R}, Acc \rangle$, where :

- Σ is the GRS signature, restricting the reduction rules to certain combinations of node labels and out-edge labels.
- $\mathcal{R}$ is the set of reduction rules
- Acc is the accepting graph for the reduction system

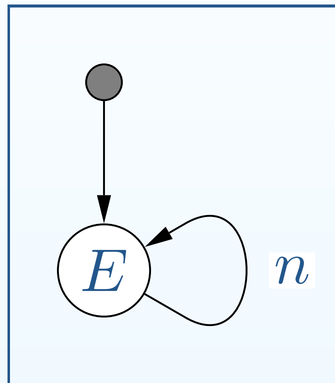
A GRS *recognises* the language $L = \{G \mid G \Rightarrow^*_{\mathcal{R}} Acc\}$, i.e a graph is a member of L iff it can be reduced to Acc.

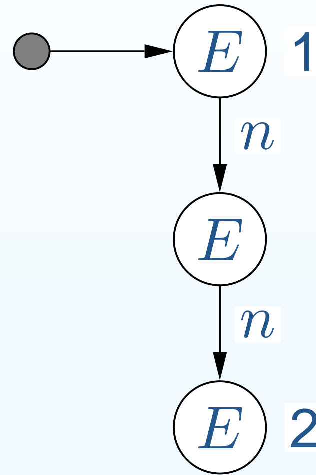
Termination of GRS

One problem with GRSs is that if we use normal graph reduction rules in $\mathcal{R}$, the best termination time-complexity we can hope for is polynomial. This is because *each rule* has a polynomial time complexity.

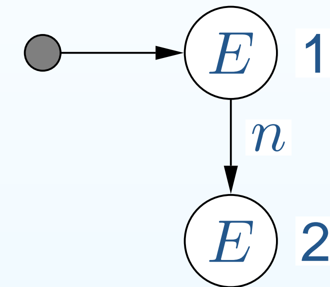
This isn't true of rules under the RGT approach, and so we can use these to produce Rooted GRSs with linear termination times.

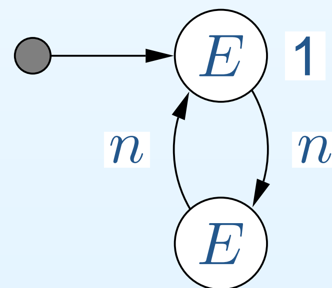
Rooted GRS for Cyclic Lists



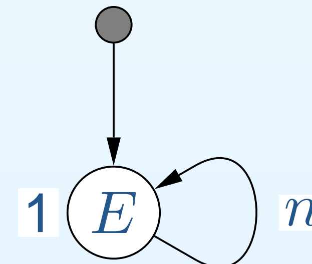


$\Rightarrow$





$\Rightarrow$



Properties of Cyclic List GRS

Termination: All rules are size-reducing, so termination will occur in at most |G| steps, where G is the reduced graph. As all of the rules are rooted, we know a step can be performed in constant time, for an overall worst-case termination time $O(|G|)$.

Completeness: We know that Acc is a member of the GRS language. We then show that every larger cyclic list can be reduced by some rule $r \in \mathcal{R}$ to give another cyclic list. As we have proved that the GRS terminates, we conclude that every cyclic list is reducible to Acc.

Soundness: We show this by deriving *from* Acc using inverse rules. We show that the cyclic list property is invariant for all r^{-1} such that $r \in \mathcal{R}$.

More Rooted GRSs

Balanced binary trees are binary trees such that all paths from the tree-root to a leaf is of the same length. We have a Rooted GRS which recognises balanced binary trees with back-pointers in linear time. It is known that balanced binary trees require context-sensitive rules to generate and recognise them.

We also have an RGRS for recognising *grid graphs* – these are rectangular graphs where each node points to its immediate rightward and downward neighbour. Once again, it is known that these graphs require context-sensitive rules to recognise and generate them.

Future Work

Look further at membership for graph languages under the RGT approach. There is quite a large body of existing literature on graph recognition. It is known, for example, that any graph language of fixed tree-width with a membership property definable in Monadic Second-Order Logic can be checked in linear time. We hope to fit the RGT approach into this wider context.

Future Work

Look further at membership for graph languages under the RGT approach. There is quite a large body of existing literature on graph recognition. It is known, for example, that any graph language of fixed tree-width with a membership property definable in Monadic Second-Order Logic can be checked in linear time. We hope to fit the RGT approach into this wider context.

Apply the RGT approach to constructing a simulation of a Turing Machine over string graphs.

Future Work

Look further at membership for graph languages under the RGT approach. There is quite a large body of existing literature on graph recognition. It is known, for example, that any graph language of fixed tree-width with a membership property definable in Monadic Second-Order Logic can be checked in linear time. We hope to fit the RGT approach into this wider context.

Apply the RGT approach to constructing a simulation of a Turing Machine over string graphs.

We are currently preparing a paper on our approach for submission to ICGT '06.