Linear-Time Graph Algorithms in GP 2

Graham Campbell Brian Courtehoute Detlef Plump

Department of Computer Science
University of York

CALCO, 5 June 2019

The Programming Language GP 2

Introduction

The graph programming language GP 2 is

- based on graph transformation rules on directed graphs
- non-deterministic
- computationally complete
- equipped with a compiler generating C code

Challenge: creating linear-time programs in graph transformation languages

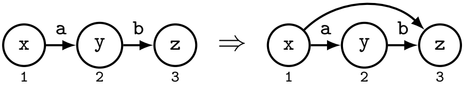
General cost of matching the left hand graph L of a rule within a host graph G:

$$\text{size}(G)^{\text{size}(L)}$$

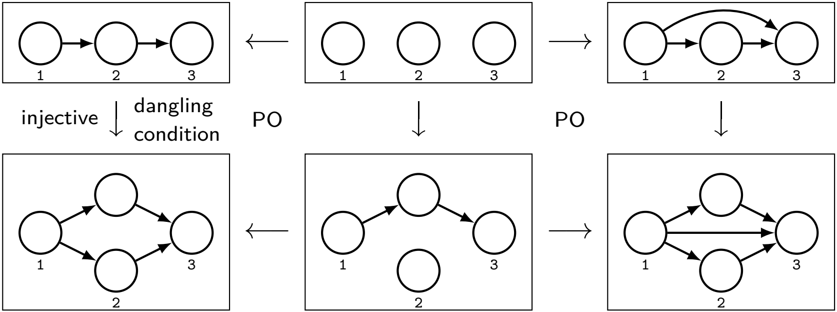
Rule Application in GP 2: Transitive Closure

```
Main = link!

link (a,b,x,y,z:list)
```




```
where not edge(1,3)
```



Tree Recognition

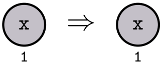
The GP 2 Program `is-tree`

Correctness

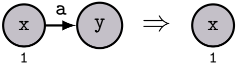
The program fails iff the input is not a non-empty tree.

```
Main = not_empty; prune!; if Check then fail
Check = {two_nodes, has_loop}
```

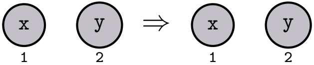
```
not_empty(a,x,y:list)
```



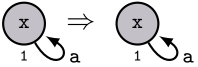

```
prune(a,x,y:list)
```



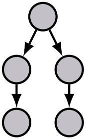

```
two_nodes(x,y:list)
```




```
has_loop(a,x:list)
```

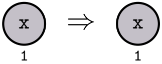


Execution Examples of is-tree

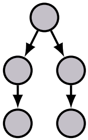


```
Main = not_empty; prune!; if Check then fail
Check = {two_nodes, has_loop}
```

```
not_empty(a,x,y:list)
```

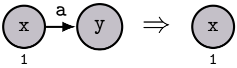


Execution Examples of `is-tree`

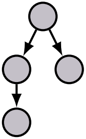


```
Main = not_empty; prune!; if Check then fail
Check = {two_nodes, has_loop}
```

```
prune(a,x,y:list)
```

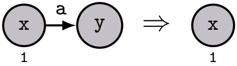


Execution Examples of `is-tree`

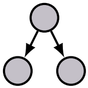


```
Main = not_empty; prune!; if Check then fail
Check = {two_nodes, has_loop}
```

```
prune(a,x,y:list)
```

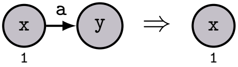


Execution Examples of is-tree

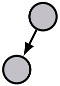


```
Main = not_empty; prune!; if Check then fail
Check = {two_nodes, has_loop}
```

```
prune(a,x,y:list)
```

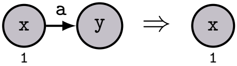


Execution Examples of `is-tree`



```
Main = not_empty; prune!; if Check then fail
Check = {two_nodes, has_loop}
```

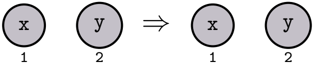
```
prune(a,x,y:list)
```



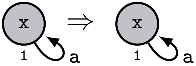
Execution Examples of `is-tree`

```
Main = not_empty; prune!; if Check then fail
Check = {two_nodes, has_loop}
```

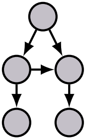
```
two_nodes(x,y:list)
```




```
has_loop(a,x:list)
```

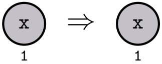


Execution Examples of is-tree

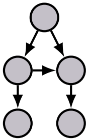


```
Main = not_empty; prune!; if Check then fail
Check = {two_nodes, has_loop}
```

not_empty(a,x,y:list)

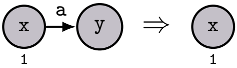


Execution Examples of is-tree

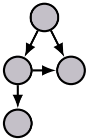


```
Main = not_empty; prune!; if Check then fail
Check = {two_nodes, has_loop}
```

```
prune(a,x,y:list)
```

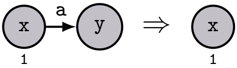


Execution Examples of is-tree

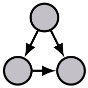


```
Main = not_empty; prune!; if Check then fail
Check = {two_nodes, has_loop}
```

```
prune(a,x,y:list)
```

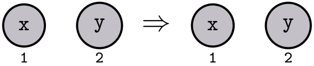


Execution Examples of `is-tree`

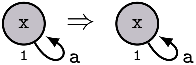


```
Main = not_empty; prune!; if Check then fail
Check = {two_nodes, has_loop}
```

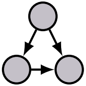
`two_nodes(x,y:list)`



`has_loop(a,x:list)`



Execution Examples of `is-tree`



```
Main = not_empty; prune!; if Check then fail
Check = {two_nodes, has_loop}
```

Time Complexity

For an input graph G with n nodes and m edges,

- `prune!` terminates after at most n applications
- `prune` is applied in $O(nm)$ time due to matching
- `is-tree` requires $O(n^2 m)$ time

Problem: Tree recognition can be done in linear time in general. Higher cost in graph programming languages due to matching.

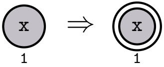
Solution: GP 2 features "rooted" nodes such as ◎ that can be accessed in constant time. The C compiler implements them as a linked list of pointers.

Trade-off: Gain of efficiency but loss of abstraction.

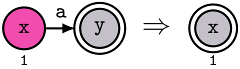
The GP 2 Program `is-tree-rooted`

```
Main = init; {prune, push}!; if {two_nodes, has_loop}
then fail
```

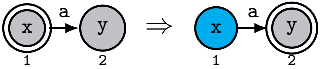
`init(x:list)`



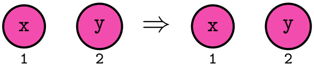
`prune(a,x,y:list)`



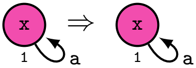
`push(a,x,y:list)`



`two_nodes(x,y:list)`

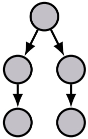


`has_loop(a,x:list)`



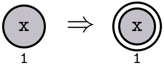
Marked nodes can only match nodes of the same colour. Magenta denotes the "any" mark wich can match any colour.

Execution Examples of `is-tree-rooted`

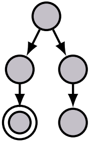


```
Main = init; {prune, push}!; if {two_nodes, has_loop}
then fail
```

```
init(x:list)
```

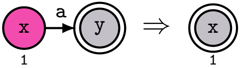


Execution Examples of `is-tree-rooted`

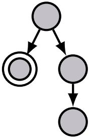


```
Main = init; {prune, push}!; if {two_nodes, has_loop}
then fail
```

```
prune(a,x,y:list)
```

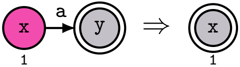


Execution Examples of `is-tree-rooted`

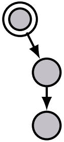


```
Main = init; {prune, push}!; if {two_nodes, has_loop}
then fail
```

```
prune(a,x,y:list)
```

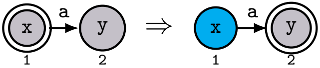


Execution Examples of `is-tree-rooted`

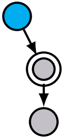


```
Main = init; {prune, push}!; if {two_nodes, has_loop}
then fail
```

```
push(a,x,y:list)
```

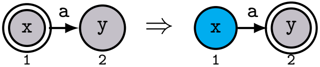


Execution Examples of `is-tree-rooted`

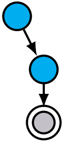


```
Main = init; {prune, push}!; if {two_nodes, has_loop}
then fail
```

push(a,x,y:list)

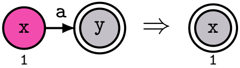


Execution Examples of `is-tree-rooted`

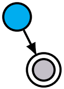


```
Main = init; {prune, push}!; if {two_nodes, has_loop}
then fail
```

```
prune(a,x,y:list)
```

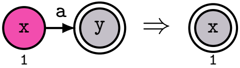


Execution Examples of `is-tree-rooted`



```
Main = init; {prune, push}!; if {two_nodes, has_loop}
then fail
```

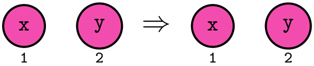
```
prune(a,x,y:list)
```



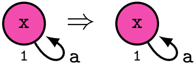
Execution Examples of `is-tree-rooted`

```
Main = init; {prune, push}!; if {two_nodes, has_loop}
then fail
```

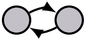
```
two_nodes(x,y:list)
```




```
has_loop(a,x:list)
```

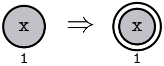


Execution Examples of `is-tree-rooted`

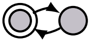


```
Main = init; {prune, push}!; if {two_nodes, has_loop}
then fail
```

```
init(x:list)
```

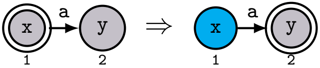


Execution Examples of `is-tree-rooted`

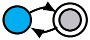


```
Main = init; {prune, push}!; if {two_nodes, has_loop}
then fail
```

```
push(a,x,y:list)
```

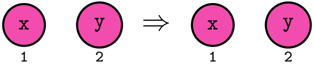


Execution Examples of is-tree-rooted

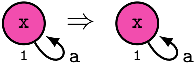


```
Main = init; {prune, push}!; if {two_nodes, has_loop}
then fail
```

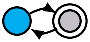
two_nodes(x,y:list)



has_loop(a,x:list)



Execution Examples of `is-tree-rooted`



```
Main = init; {prune, push}!; if {two_nodes, has_loop}
then fail
```

The Use of Roots

A rule $L \Rightarrow R$ is *fast* if

- Every connected component of L has a root.
- Other constraints on labels and conditions apply (omitted).

Theorem (Complexity of Matching Fast Rules, Bak-Plump 2012)

Rooted graph matching can be implemented to run in constant time for fast rules, provided there are upper bounds on the maximum node degree and the number of roots in host graphs.

The Use of Roots

How impactful are these constraints in practice?

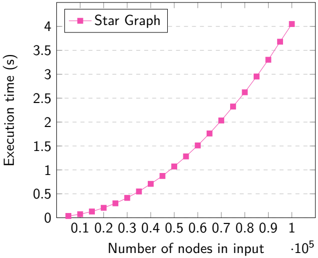
Number of roots: up to the programmer to keep it bounded

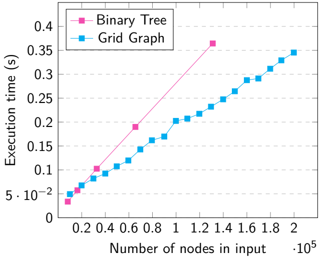
Maximum node degree: bounded in many practical applications such as traffic networks or social networks

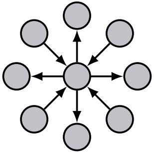
Roots

- increase control in small areas of the graph, but
- decrease the simplicity of programs.

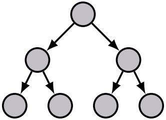
Average Execution Times of `is-tree-rooted`



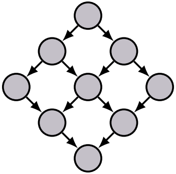




A Star Graph



A complete binary tree



A grid graph

Topological Sorting

Topological Sorting

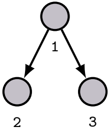
A *topological sorting* is a linear order $<$ on the nodes of a graph with no directed cycles (DAG) such that

$$\text{for each edge from } u \text{ to } v,\ u < v.$$

Idea of the program:

- Encoding the topological sorting as a stack of nodes
- Using depth-first-search (DFS) for graph traversal
- Pushing nodes onto the stack during the back step of a directed DFS
- If the program gets stuck, using an undirected DFS to find an unsorted node

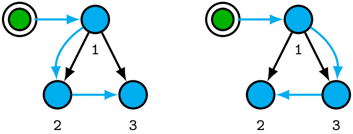
Encoding a Topological Sorting



There are two possible topological sortings on this graph:

$$1 < 2 < 3 \text{ and } 1 < 3 < 2.$$

`top-sort` may output either one of them encoded as a stack:

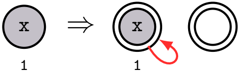


The GP 2 Program top-sort

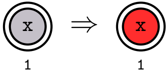
```
Main = init; SearchUnsortedNodes

SearchUnsortedNodes = ((try unsorted then SortNodes;
search_forward)!; try search_back else break)!
```

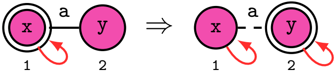
`init (x:list)`



`unsorted (x:list)`

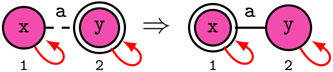


`search_forward (a,x,y:list)`



`where not edge(2,2)`

`search_back (a,x,y:list)`



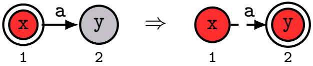
Undirected edges: notation for a non-deterministic call of rule with edge in either orientation.

Dashed edges: dashing is a mark reserved for edges.

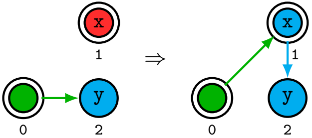
The Procedure SortNodes

```
SortNodes = (sort_forward!; try sort_back_push else (try sort_back_stack
else (try red_push else red_stack; break)))!
```

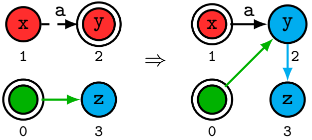
`sort_forward (a,x,y:list)`



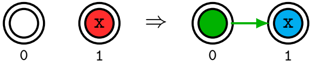
`red_push (x,y:list)`



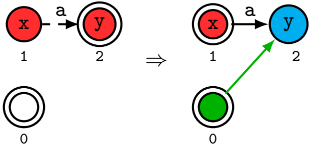
`sort_back_push (a,x,y,z:list)`



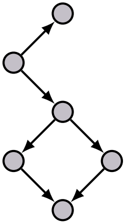
`red_stack (x:list)`



`sort_back_stack (a,x,y:list)`

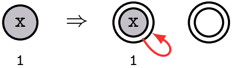


Execution Example of `top-sort`

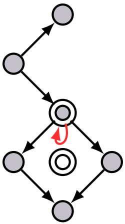


`Main = init; SearchUnsortedNodes`

`init (x:list)`

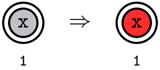


Execution Example of `top-sort`

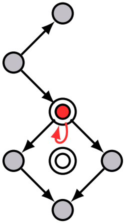


```
SearchUnsortedNodes = ((try unsorted then SortNodes;
search_forward)!; try search_back else break)!
```

`unsorted (x:list)`

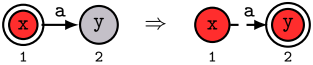


Execution Example of top-sort

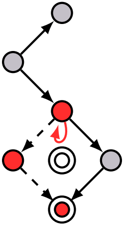


```
SortNodes = (sort_forward!; try sort_back_push else
(try sort_back_stack else (try red_push else
red_stack; break)))!
```

sort_forward (a,x,y:list)

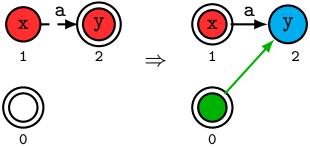


Execution Example of top-sort

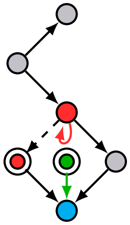


```
SortNodes = (sort_forward!; try sort_back_push else
(try sort_back_stack else (try red_push else
red_stack; break)))!
```

`sort_back_stack (a,x,y:list)`

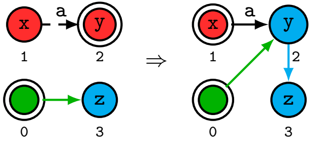


Execution Example of top-sort

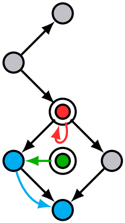


```
SortNodes = (sort_forward!; try sort_back_push else
(try sort_back_stack else (try red_push else
red_stack; break)))!
```

sort_back_push (a,x,y,z:list)

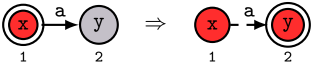


Execution Example of top-sort

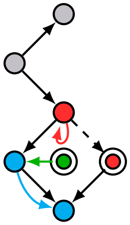


```
SortNodes = (sort_forward!; try sort_back_push else
(try sort_back_stack else (try red_push else
red_stack; break)))!
```

`sort_forward (a,x,y:list)`

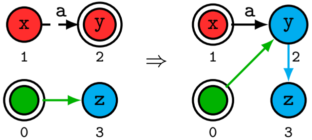


Execution Example of top-sort

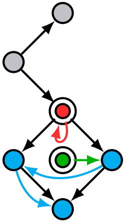


```
SortNodes = (sort_forward!; try sort_back_push else
(try sort_back_stack else (try red_push else
red_stack; break)))!
```

sort_back_push (a,x,y,z:list)

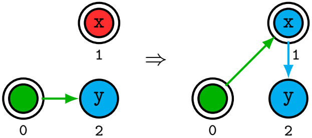


Execution Example of top-sort

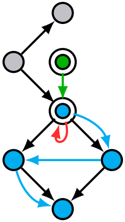


```
SortNodes = (sort_forward!; try sort_back_push else
(try sort_back_stack else (try red_push else
red_stack; break)))!
```

red_push (x,y:list)

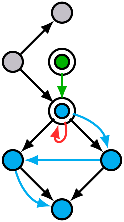


Execution Example of top-sort



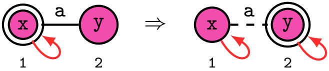
```
SortNodes = (sort_forward!; try sort_back_push else
(try sort_back_stack else (try red_push else
red_stack; break)))!
```

Execution Example of top-sort



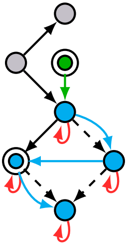
```
SearchUnsortedNodes = ((try unsorted then SortNodes;
search_forward)!; try search_back else break)!
```

search_forward (a,x,y:list)



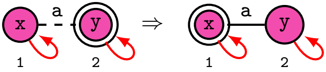
where not edge(2,2)

Execution Example of top-sort

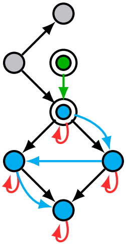


`SearchUnsortedNodes = ((try unsorted then SortNodes; search_forward)!; try search_back else break)!`

`search_back (a,x,y:list)`

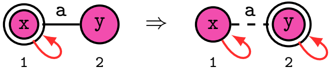


Execution Example of top-sort



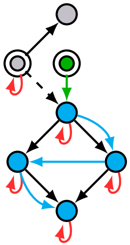
```
SearchUnsortedNodes = ((try unsorted then SortNodes;
search_forward)!; try search_back else break)!
```

search_forward (a,x,y:list)



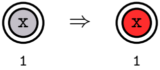
where not edge(2,2)

Execution Example of `top-sort`

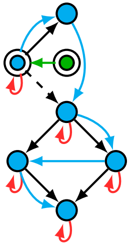


`SearchUnsortedNodes = ((try unsorted then SortNodes; search_forward)!; try search_back else break)!`

`unsorted (x:list)`

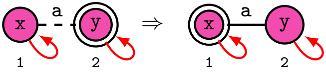


Execution Example of top-sort

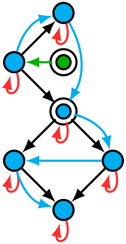


```
SearchUnsortedNodes = ((try unsorted then SortNodes;
search_forward)!; try search_back else break)!
```

search_back (a,x,y:list)

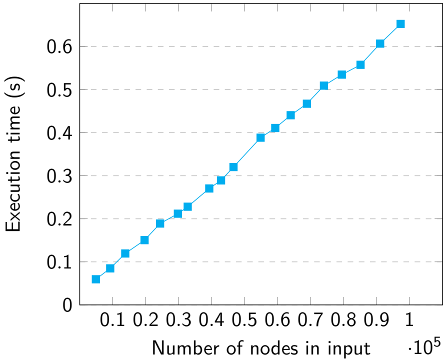


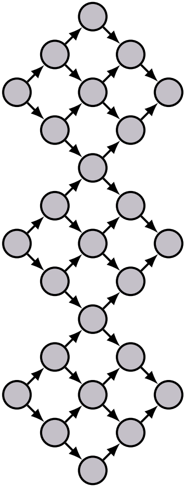
Execution Example of top-sort



```
SearchUnsortedNodes = ((try unsorted then SortNodes;
search_forward)!; try search_back else break)!
```

Performance of top-sort





A 3x3x3 grid chain graph

Conclusion

Conclusion

Rooted rules permit linear-time implementations of tree recognition and topological sorting in GP 2 for inputs of bounded degree.

Future work:

- Investigating how the implementation of data structures as part of the host graph can be used to implement more linear-time algorithms in GP 2.
- Finding a way to automate the refinement of unrooted programs by adding root nodes in order to speed up matching.
- Finding a way to implement linear-time algorithms for inputs of unbounded degree by modifying GP 2 and its implementation.