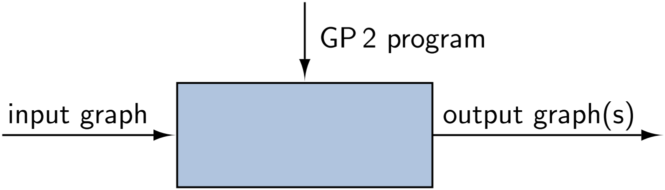
Speeding up Graph Transformation

Detlef Plump

University of York, UK

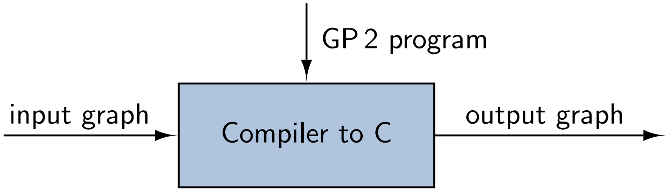
In cooperation with Ziad Ismaili Alaoui, Chris Bak, Graham Campbell, Brian Courtehoute, Mike Dodds and Jack Romö

Graph programming language GP 2



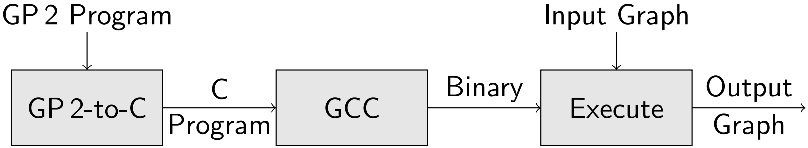
- Based on attributed graph-transformation rules
- Formal operational semantics (non-deterministic)
- Computationally complete

Graph programming language GP 2



- Based on attributed graph-transformation rules
- Formal operational semantics (non-deterministic)
- Computationally complete

GP 2-to-C Compiler



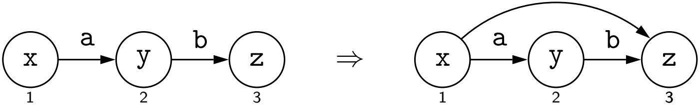
Example program: transitive closure

A graph is *transitive* if for every directed path $v \rightsquigarrow v'$ with $v \neq v'$, there is an edge $v \to v'$.

Program for computing a *transitive closure* of the input graph (smallest transitive extension):

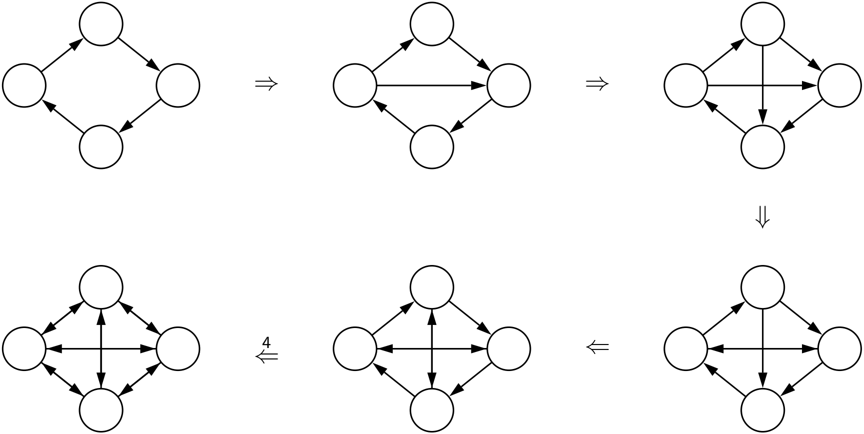
$\mathtt{Main} = \mathtt{link}!$

$\mathtt{link}(\mathrm{a}, \mathrm{b}, \mathrm{x}, \mathrm{y}, \mathrm{z} \colon \mathtt{list})$



$\mathtt{where\ not\ edge}(1, 3)$

Example program: transitive closure (cont'd)



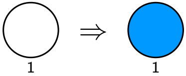
Graph transformation is slow: example

Input: A non-empty unlabelled graph G without marks.

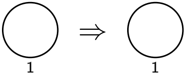
Output: Fail if and only if G is disconnected.

```
Main = init; forward!; if match then fail
```

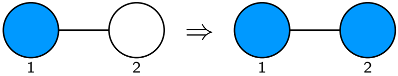
`init ()`



`match ()`



`forward ()`



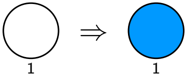
Graph transformation is slow: example

Input: A non-empty unlabelled graph G without marks.

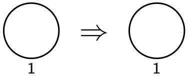
Output: Fail if and only if G is disconnected.

```
Main = init; forward!; if match then fail
```

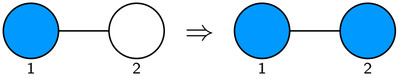
`init ()`



`match ()`



`forward ()`



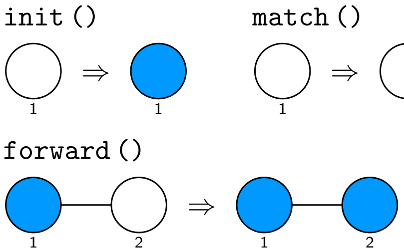
- Maximal number of rule applications: $|V|$

Graph transformation is slow: example

Input: A non-empty unlabelled graph G without marks.

Output: Fail if and only if G is disconnected.

```
Main = init; forward!; if match then fail
```



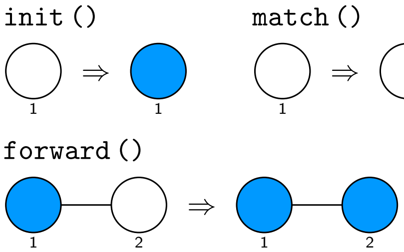
- Maximal number of rule applications: $|V|$
- Worst case time for matching `forward`: $\mathcal{O}(|V| \times |E|)$

Graph transformation is slow: example

Input: A non-empty unlabelled graph G without marks.

Output: Fail if and only if G is disconnected.

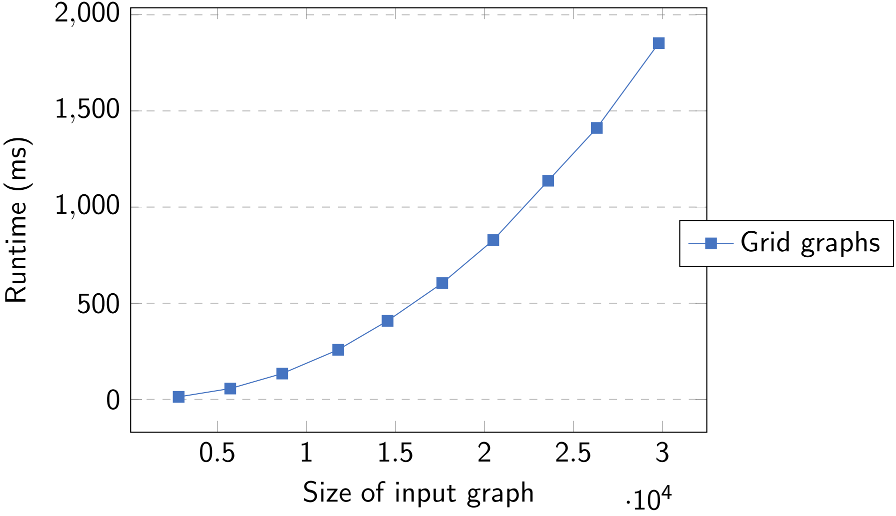
```
Main = init; forward!; if match then fail
```



- Maximal number of rule applications: $|V|$
- Worst case time for matching `forward`: $\mathcal{O}(|V| \times |E|)$
- Worst case program runtime: $\mathcal{O}(|V|^2 \times |E|)$

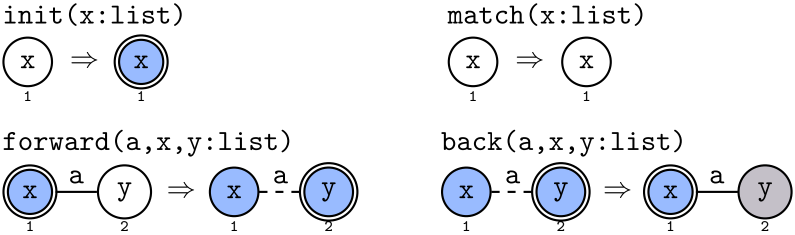
Graph transformation is slow: example

Measured runtime on square grids:



Checking connectedness with rooted rules

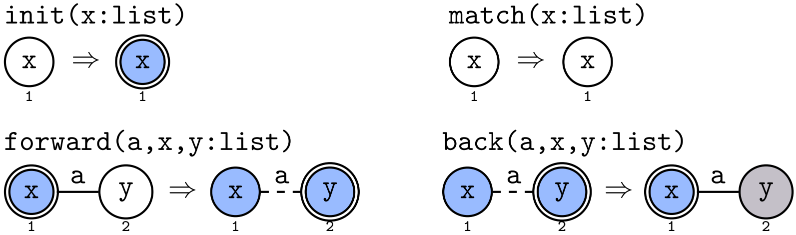
```
Main = init; DFS!; if match then fail
DFS = forward!; try back else break
```



- Rule `init` generates a unique *root node* in the host graph.
- GP 2's graph data structure includes a list of C-pointers to access roots in constant time.

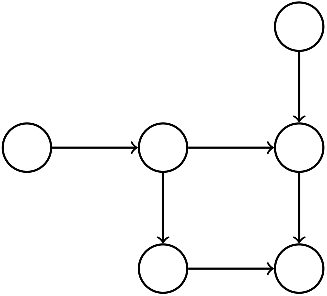
Checking connectedness with rooted rules

```
Main = init; DFS!; if match then fail
DFS = forward!; try back else break
```



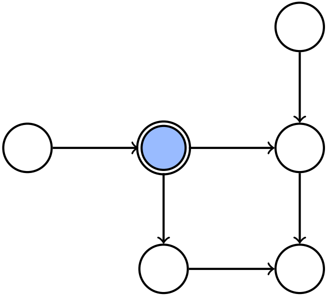
- Rule `init` generates a unique *root node* in the host graph.
- GP 2's graph data structure includes a list of C-pointers to access roots in constant time.
- Rules `forward` and `back` can be matched in constant time in graph classes of bounded node degree.

Checking connectedness with rooted rules



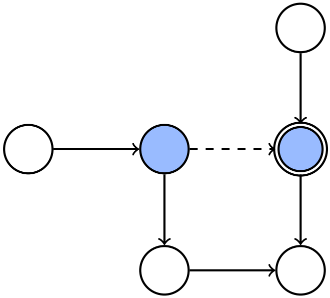
- The program implements a *depth-first search* to find all nodes connected to the root.

Checking connectedness with rooted rules



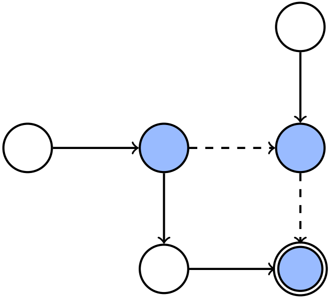
- The program implements a *depth-first search* to find all nodes connected to the root.

Checking connectedness with rooted rules



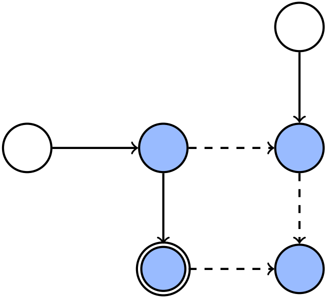
- The program implements a *depth-first search* to find all nodes connected to the root.

Checking connectedness with rooted rules



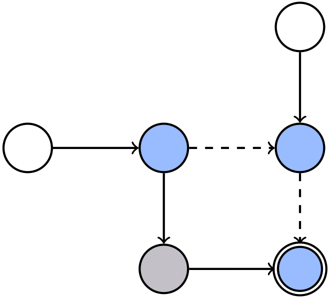
- The program implements a *depth-first search* to find all nodes connected to the root.

Checking connectedness with rooted rules



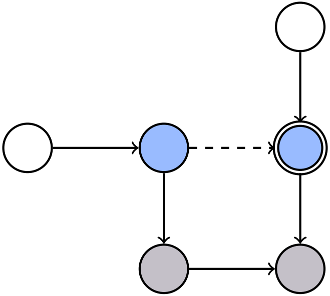
- The program implements a *depth-first search* to find all nodes connected to the root.

Checking connectedness with rooted rules



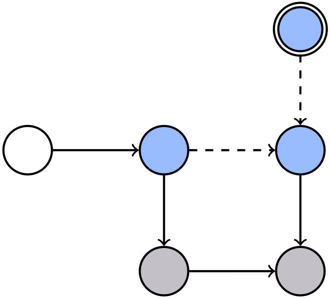
- The program implements a *depth-first search* to find all nodes connected to the root.

Checking connectedness with rooted rules



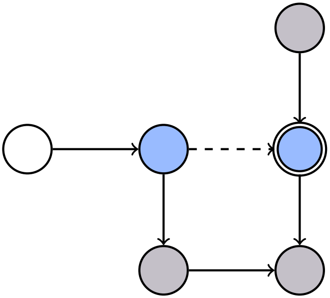
- The program implements a *depth-first search* to find all nodes connected to the root.

Checking connectedness with rooted rules



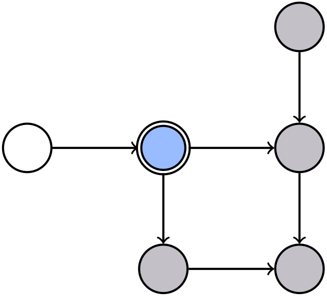
- The program implements a *depth-first search* to find all nodes connected to the root.

Checking connectedness with rooted rules



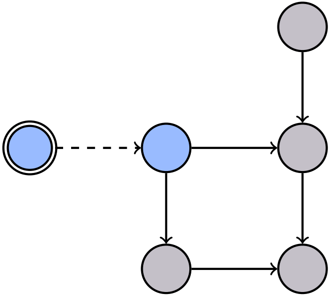
- The program implements a *depth-first search* to find all nodes connected to the root.

Checking connectedness with rooted rules



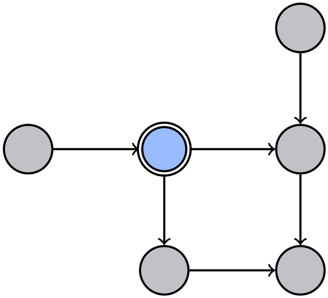
- The program implements a *depth-first search* to find all nodes connected to the root.

Checking connectedness with rooted rules



- The program implements a *depth-first search* to find all nodes connected to the root.

Checking connectedness with rooted rules



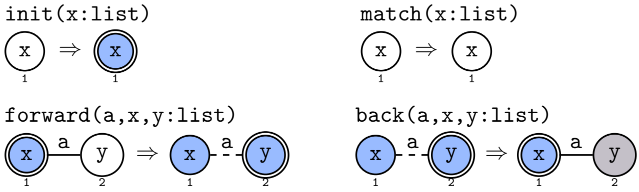
- The program implements a *depth-first search* to find all nodes connected to the root.

Checking connectedness with rooted rules

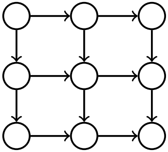
Theorem (Correctness and complexity)

1. *Given a non-empty input graph G, the program fails if and only if G is disconnected.*
2. *The program terminates in time $\mathcal{O}(|V| + |E|)$ on input graph classes of bounded node degree.*

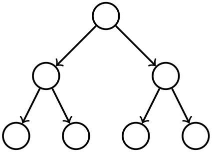
```
Main = init; DFS!; if match then fail
DFS = forward!; try back else break
```



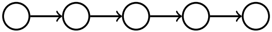
Graph classes for time measurements



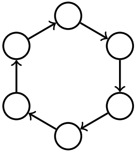
Grid graphs



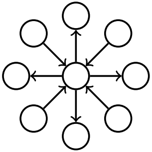
Binary trees



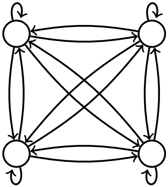
Linked lists



Cycle graphs



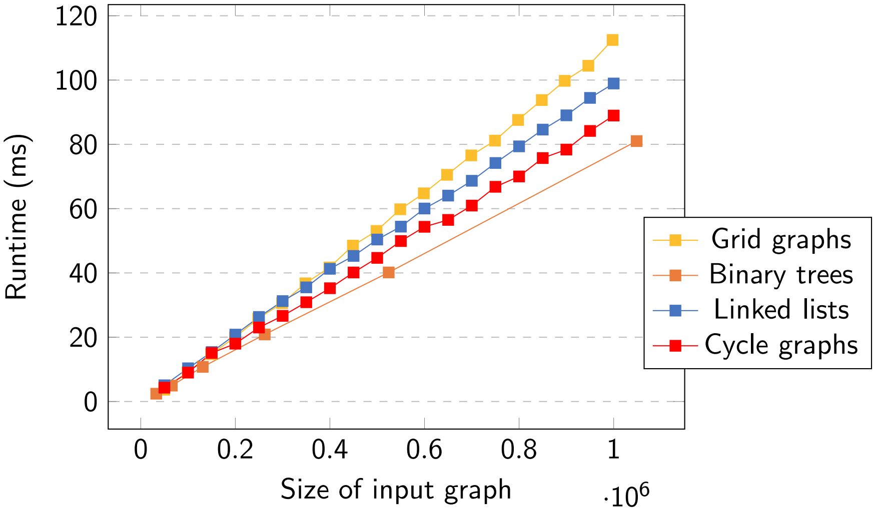
Star graphs



Complete graphs

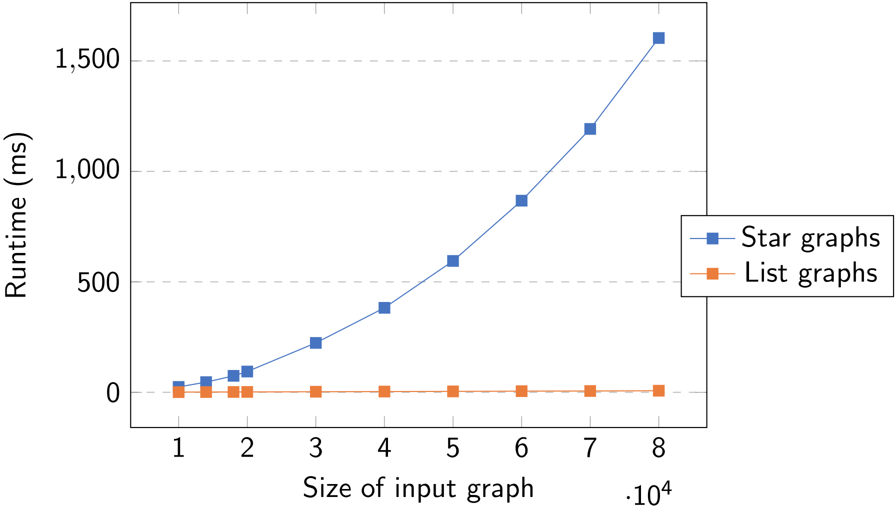
Checking connectedness with rooted rules

Measured runtime on bounded-degree graphs:

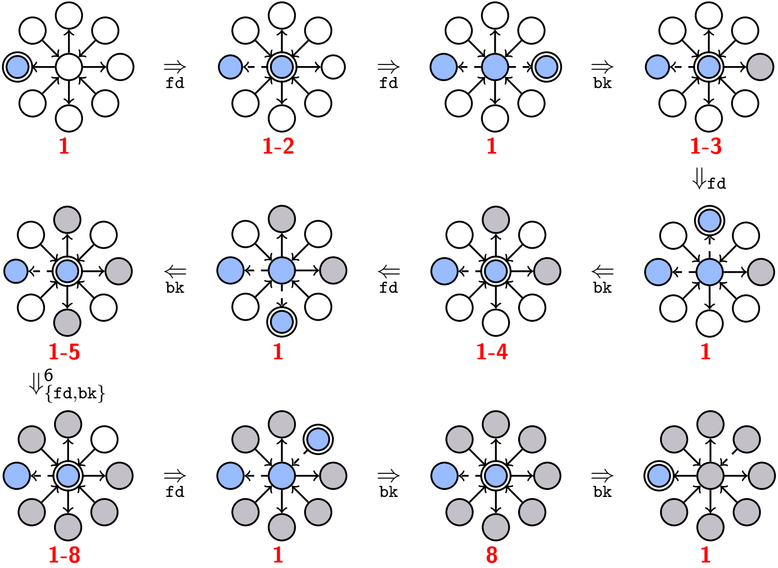


Checking connectedness with rooted rules

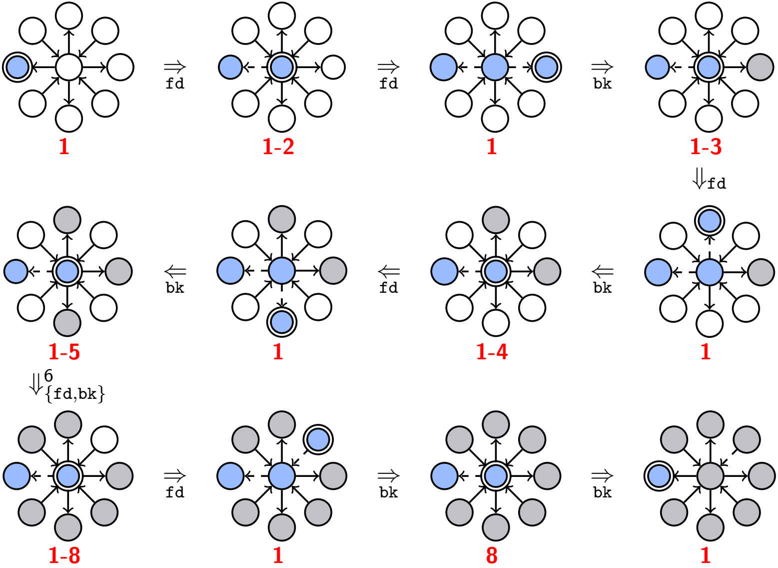
Measured runtime on star graphs:



Matching attempts with the forward rule

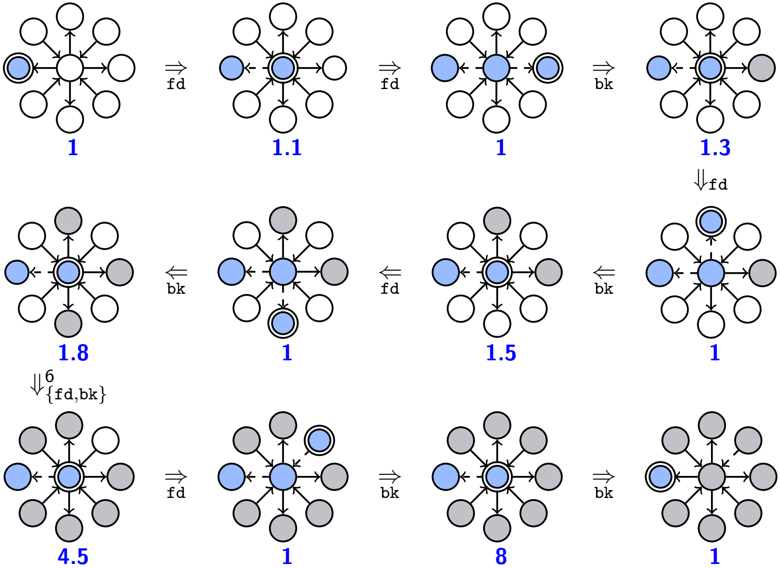


Matching attempts with the `forward` rule



Worst case: $2|E| + \sum_{i=1}^{|E|} i = \mathcal{O}(|E|^2)$

Matching attempts with the `forward` rule



Expected numbers

Improving the GP 2 graph data structure

- 2015: In Chris Bak's original graph data structure, each node contained the IDs of two inedges and two outedges. Other incident edges were placed in a dynamic array.
- 2020: In Graham Campbell's and Jack Romö's data structure, each node comes with two linked lists, one for all inedges and one for all outedges.
- 2024: Ziad Ismaili Alaoui modified the 2020 data structure by placing the edges incident with a node into 15 different lists, separated by edge marks and edge directions.

Storing incident edges

Each node v comes with a two-dimensional array holding 15 linked lists of incident edges:

	in	out	loop
unmarked	...	...	...
dashed	...	...	...
red	...	...	...
green	...	...	...
blue	...	...	...

where "..." is a linked list of edges incident with v

Storing incident edges

Each node v comes with a two-dimensional array holding 15 linked lists of incident edges:

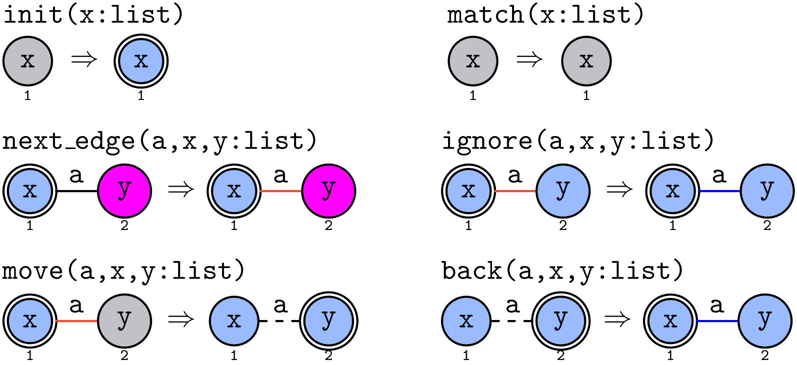
	in	out	loop
unmarked	...	...	...
dashed	...	...	...
red	...	...	...
green	...	...	...
blue	...	...	...

where "..." is a linked list of edges incident with v

- *As a consequence, finding an edge incident with a given node requires only constant time.*

Checking connectedness in linear time

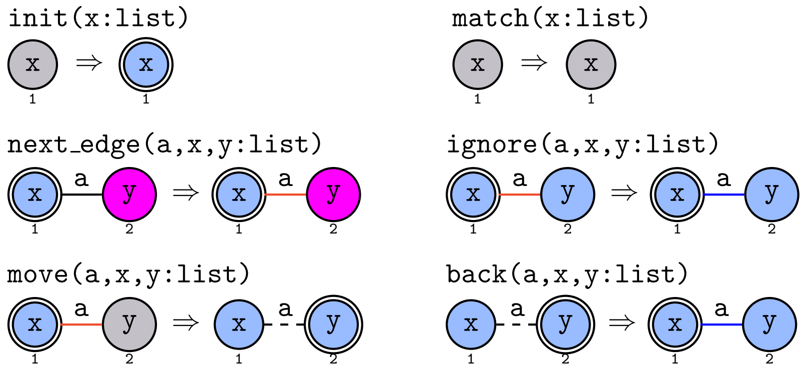
```
Main = init; DFS!; if match then fail
DFS = (next_edge; {move, ignore})!; try back else break
```



- Input graphs have grey nodes; magenta is a wildcard for marks

Checking connectedness in linear time

```
Main = init; DFS!; if match then fail
DFS = (next_edge; {move, ignore})!; try back else break
```



- Input graphs have grey nodes; magenta is a wildcard for marks
- All rules except `match` are matched in constant time.

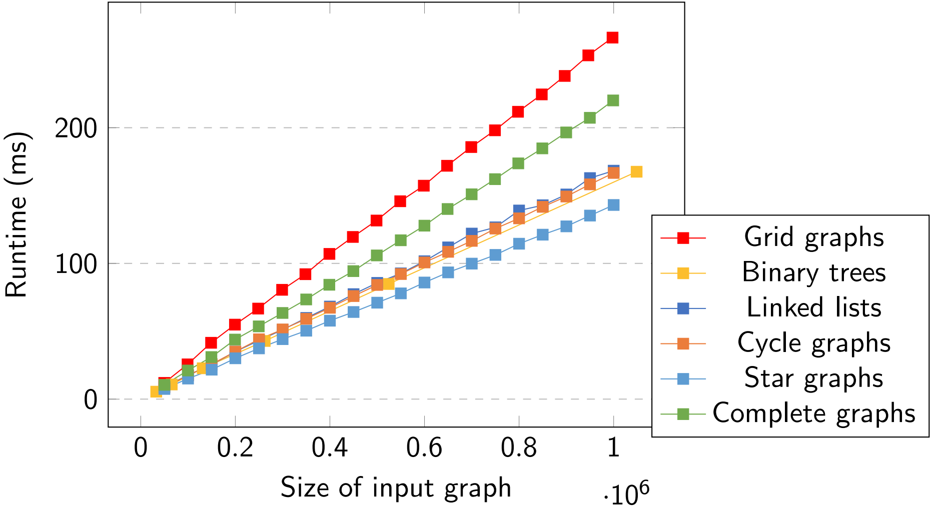
Checking connectedness in linear time

Theorem (Correctness and complexity)

1. *Given a non-empty input graph G, the program fails if and only if G is disconnected.*
2. *The program terminates in time $\mathcal{O}(|V| + |E|)$.*

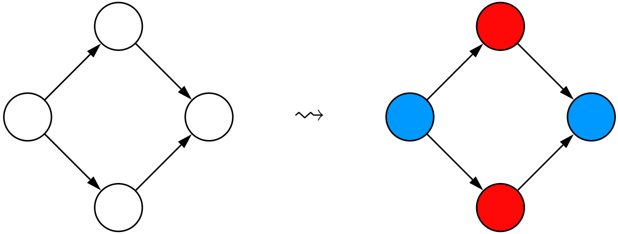
Checking connectedness in linear time

Measured runtime:



2-colouring

A *2-colouring* is an assignment $V \to \{\text{blue}, \text{red}\}$ such that each non-loop edge has end points with distinct colours.

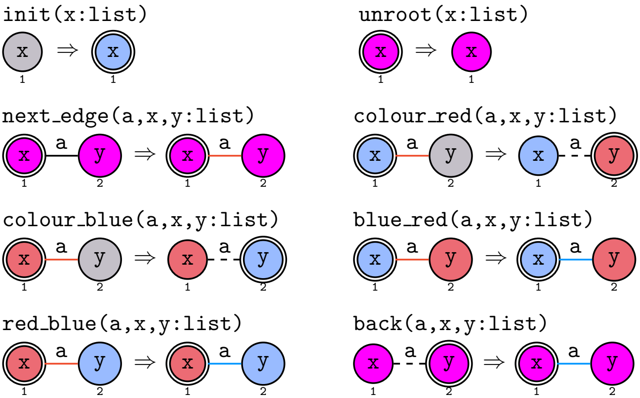


Lemma

A graph is 2-colourable if and only if it does not contain an undirected cycle of odd length ≥ 3.

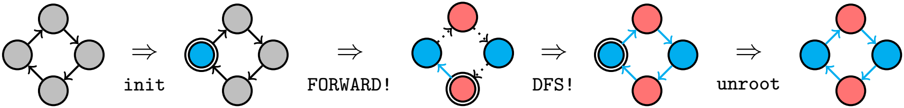
2-Colouring in linear time

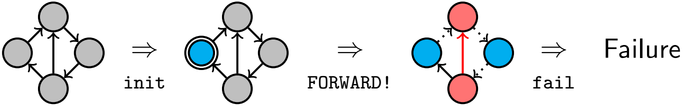
```
Main = init; DFS!; try unroot else fail
DFS = Forward!; try back else break
Forward = next_edge; try {colour_red, colour_blue, blue_red, red_blue}
else (unroot; break)
```



- Input graphs have grey nodes and are connected

2-Colouring in linear time





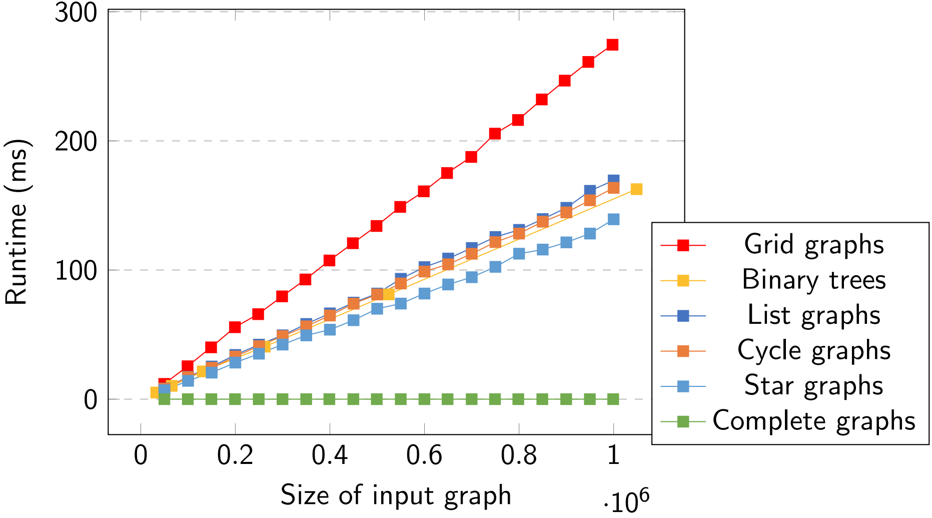
2-Colouring in linear time

Theorem (Correctness and complexity)

1. *Given a non-empty and connected input graph G, the program fails if G is not 2-colourable.*
   Otherwise, the program returns G with nodes coloured red and blue such that adjacent nodes have different colours.
2. *The program terminates in time $\mathcal{O}(|V| + |E|)$.*

2-Colouring in linear time

Measured runtime:



Recognising binary DAGs in linear time

- Directed acyclic graphs where each node has at most two outgoing edges

Recognising binary DAGs in linear time

- Directed acyclic graphs where each node has at most two outgoing edges
- Our program *reduces* input graphs by repeatedly
  - moving a root along edges in opposite direction to find a node v without incoming edges, and
  - deleting v and its ≤ 2 outedges

Recognising binary DAGs in linear time

- Directed acyclic graphs where each node has at most two outgoing edges
- Our program *reduces* input graphs by repeatedly
  - moving a root along edges in opposite direction to find a node v without incoming edges, and
  - deleting v and its ≤ 2 outedges
- The input graph is a binary DAG iff the result graph is empty

Recognising binary DAGs in linear time

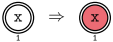
```
Main = (init; Reduce!; if flag then break)!; if flag then fail
Reduce = up!; try Delete else (set_flag; break)
```

init(x:list)

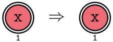
up(a,x,y:list)

where outdeg(1)<3

set_flag(x:list)

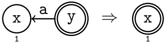


flag(x:list)

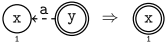


Delete = {del1, del1_d, del21, del21_d, del22, del22_d, del0}

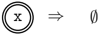
del1(a,x,y:list)



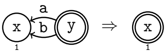
del1_d(a,x,y:list)



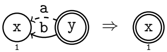
del0(x:list)



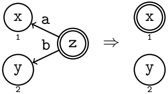
del21(a,b,x,y:list)



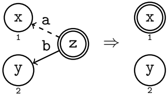
del21_d(a,b,x,y:list)



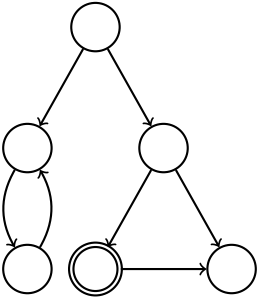
del22(a,b,x,y,z:list)



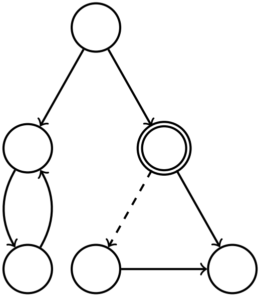
del22_d(a,b,x,y,z:list)



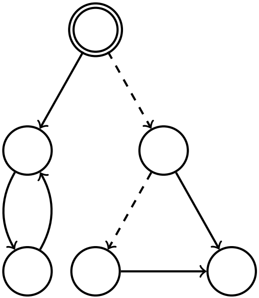
Recognising binary DAGs in linear time



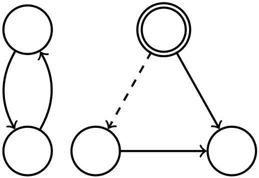
Recognising binary DAGs in linear time



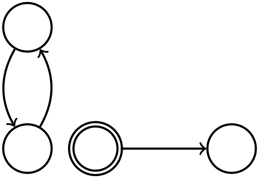
Recognising binary DAGs in linear time



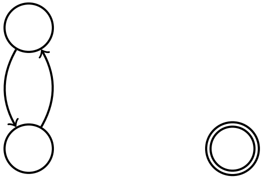
Recognising binary DAGs in linear time



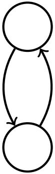
Recognising binary DAGs in linear time



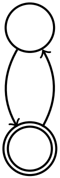
Recognising binary DAGs in linear time



Recognising binary DAGs in linear time

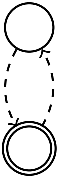


Recognising binary DAGs in linear time

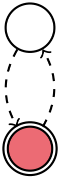


Recognising binary DAGs in linear time

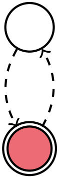
Recognising binary DAGs in linear time



Recognising binary DAGs in linear time



Recognising binary DAGs in linear time



$\Rightarrow$ Failure

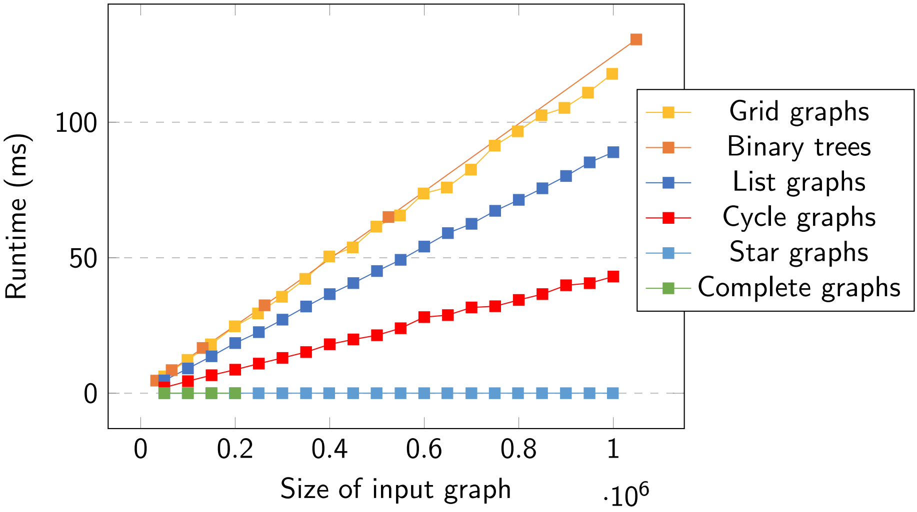
Recognising binary DAGs in linear time

Theorem (Correctness and complexity)

1. *Given an input graph G, the program fails if and only if G is not a binary DAG.*
2. *The program terminates in time $\mathcal{O}(|V| + |E|)$.*

Recognising binary DAGs in linear time

Measured runtime:



Overview: Fast GP 2 programs

Destructive	Non-destructive
Binary DAG recognition	Checking connectedness (linear time)
Tree recognition	2-Colouring (linear-time on connected graphs)
Cycle graph recognition	Topological sorting (linear-time on bounded-degree classes)
(all linear time)	Minimum spanning tree generation ($\mathcal{O}(m \log n)$ on bounded-degree classes)

Conclusion

- Rule-based graph programs allow for simple formal reasoning about correctness and complexity — compared with imperative programs.
- Programmers don't have access to the graph data structure: a reasonable price to pay for simple formal reasoning.
- Rule matching in constant time is crucial for achieving fast runtimes.
- Our case studies match the best known time bounds of imperative algorithms — sometimes under mild conditions.
- For programs such as 2-colouring, we need not assume connected input graphs if nodes are separated by marks too (work in progress).
- We speculate that all DFS-based graph algorithms can be implemented to run in linear time without extra conditions.