

Checking Graph Properties with GP

Chris Poskitt (cmp501@cs.york.ac.uk)

The University of York

PLASMA Seminar: 5th March 2009

Today's Talk

1. Motivation
2. GP Refresher
3. Testing and describing graph properties with GP
4. Examples and demonstrations
5. Conclusions and future work
6. Questions

Graphs

A **graph** G comprises

- ▶ a finite set of **nodes**, V_G
- ▶ a finite set of **edges**, E_G
- ▶ **source** and **target** functions mapping edges to nodes
 - ▶ Edges are **directed**
- ▶ labelling functions

If $V_G = \emptyset$, then G is the empty graph \emptyset .



Motivation

- ▶ Graphs are ubiquitous in computer science
- ▶ Represented at a low level in programming languages
 - ▶ Adjacency matrices
 - ▶ Adjacency lists

Adjacency Matrix Example



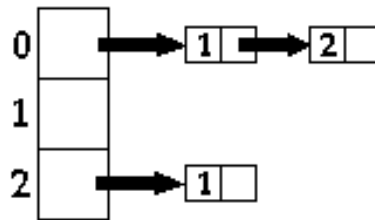
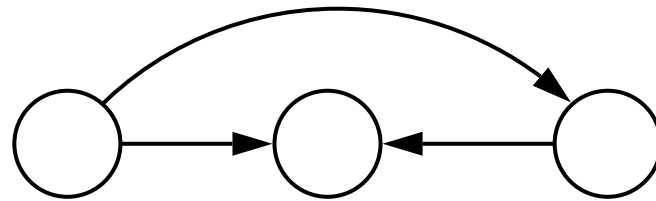
$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$A_{ij} = \text{no. edges with source } i \text{ and target } j$

$$\mathbf{L}_V = [1 \quad 2 \quad 1]$$

- ▶ Use a two dimensional array: $a[i][j]$
- ▶ ...and one dimensional arrays for labels

Adjacency Lists Example



- ▶ Array entry $a[i]$ contains a linked list of nodes adjacent to node i , where i is the source node

Trade-Offs

To make things more difficult!

Requirement	Preferred Data Structure
Huge sparse graphs	Adjacency lists
Complete or almost complete graphs	Adjacency matrices
Testing the existence of an edge	Adjacency matrices
Edge insertion and deletion	Adjacency matrices, $O(1)$ vs. $O(\text{degree}_{v_i})$
Finding the degree of a node	Adjacency lists
Traversing graphs	Adjacency lists, $\Theta(m + n)$ vs. $\Theta(n^2)$

Motivation

- ▶ Low level representations of graphs
- ▶ Performance trade-offs
- ▶ **Difficult to implement, comprehend, and verify graph algorithms that test for graph properties**

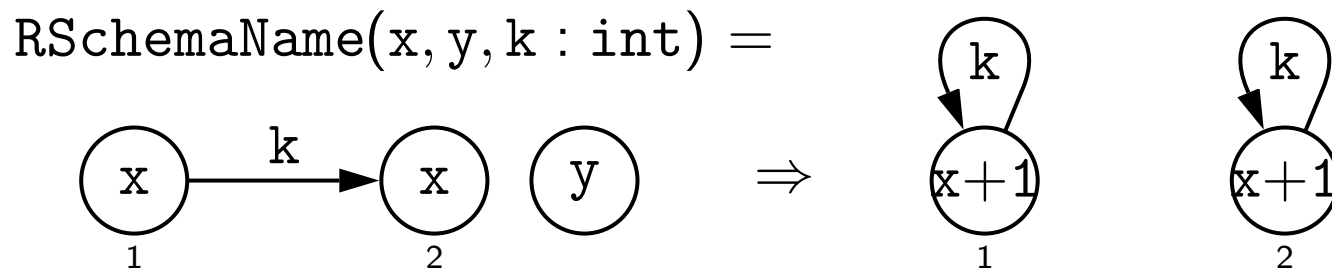
GP removes the need to work at such a low level when solving graph problems. Programmers work with the nodes and edges of the input graph directly; underlying data structures are of no concern.

Today's Talk

1. Motivation
2. GP Refresher
3. Testing and describing graph properties with GP
4. Examples and demonstrations
5. Conclusions and future work
6. Questions

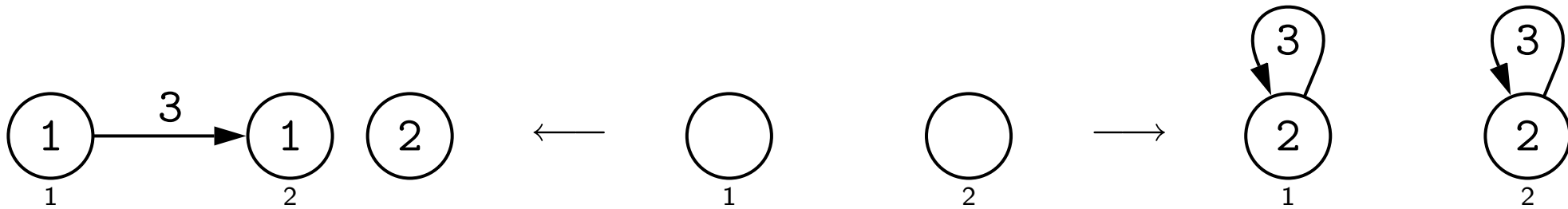
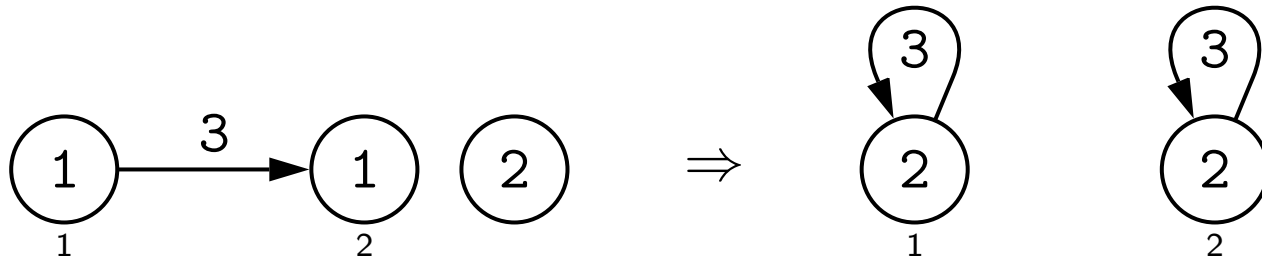
Conditional Rule Schemata

GP introduces **rule schemata**, from which rules are induced.

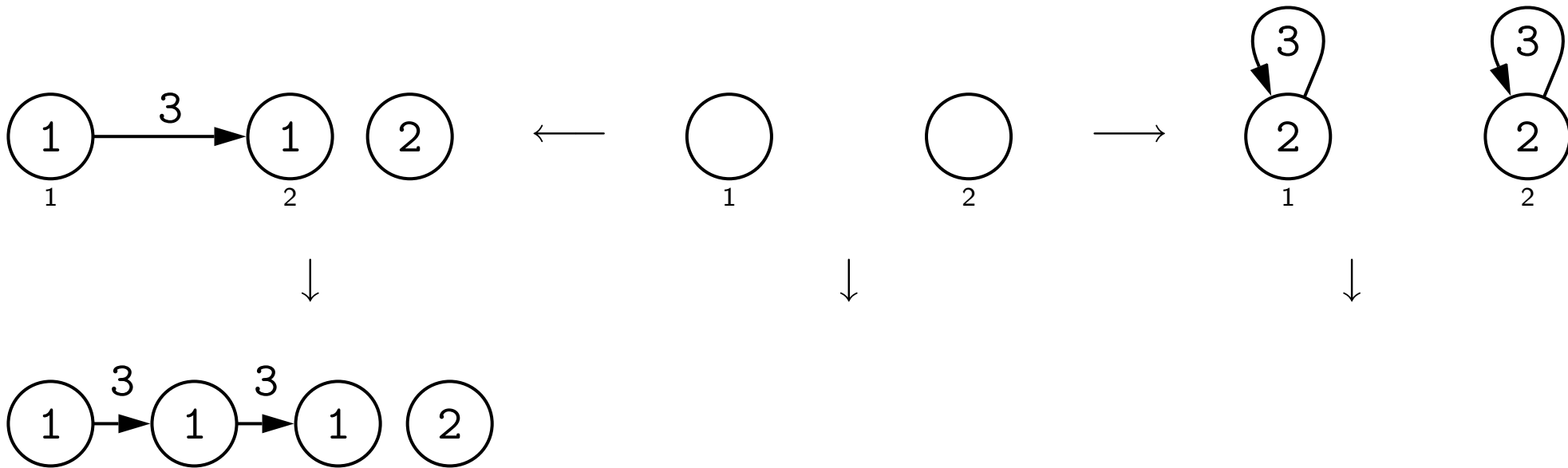


- ▶ Expressions over labels
- ▶ Conditions
 - ▶ where $x + k > y$
 - ▶ where not $\text{edge}(2, 1)$
- ▶ Rule applications are local (dangling condition)

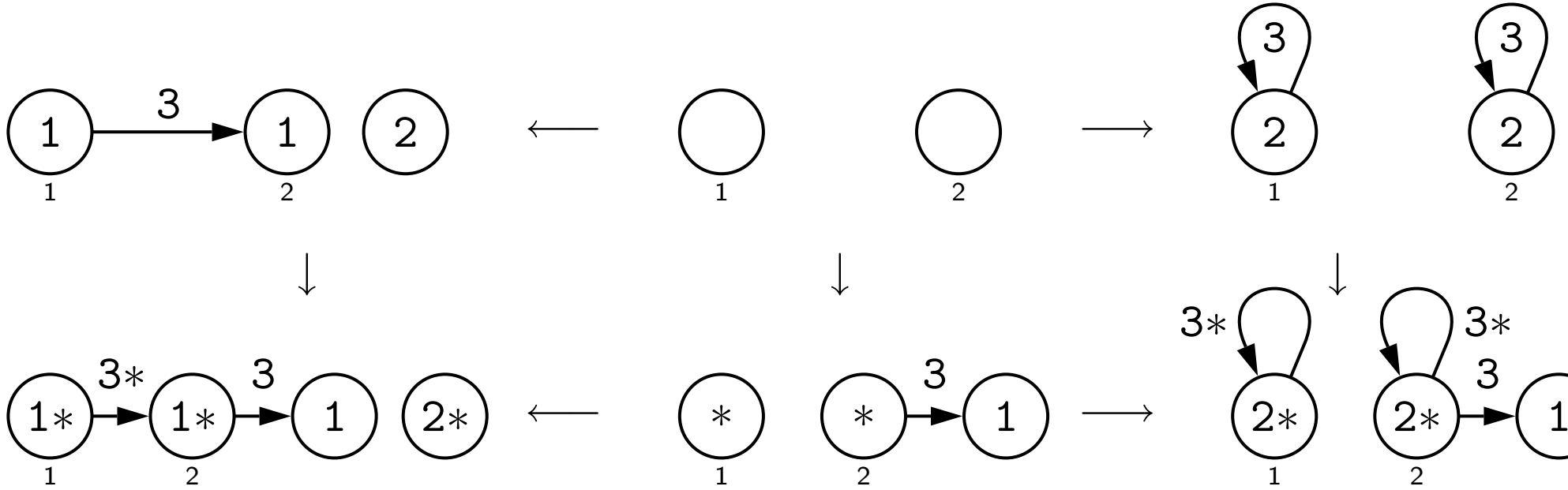
Rule (DPO Approach)



Rule (DPO Approach)



Rule (DPO Approach)



Control Constructs

- ▶ Sequential composition, ‘;’

Control Constructs

- ▶ Sequential composition, ‘;’
- ▶ Non-deterministic single-step application of a set of conditional rule schemata, e.g. $\{RS1, RS2, RS3\}$

Control Constructs

- ▶ Sequential composition, ‘;’
- ▶ Non-deterministic single-step application of a set of conditional rule schemata, e.g. $\{RS1, RS2, RS3\}$
- ▶ The as-long-as-possible looping construct, ‘!’

Control Constructs

- ▶ Sequential composition, ‘;’
- ▶ Non-deterministic single-step application of a set of conditional rule schemata, e.g. $\{RS1, RS2, RS3\}$
- ▶ The as-long-as-possible looping construct, ‘!’
- ▶ Branching constructs *if-then-else* and *try-then-else*

Control Constructs

- ▶ Sequential composition, ‘;’
- ▶ Non-deterministic single-step application of a set of conditional rule schemata, e.g. $\{RS1, RS2, RS3\}$
- ▶ The as-long-as-possible looping construct, ‘!’
- ▶ Branching constructs *if-then-else* and *try-then-else*
- ▶ The *while-do* looping construct

Control Constructs

- ▶ Sequential composition, ‘;’
- ▶ Non-deterministic single-step application of a set of conditional rule schemata, e.g. $\{RS1, RS2, RS3\}$
- ▶ The as-long-as-possible looping construct, ‘!’
- ▶ Branching constructs if-then-else and try-then-else
- ▶ The while-do looping construct

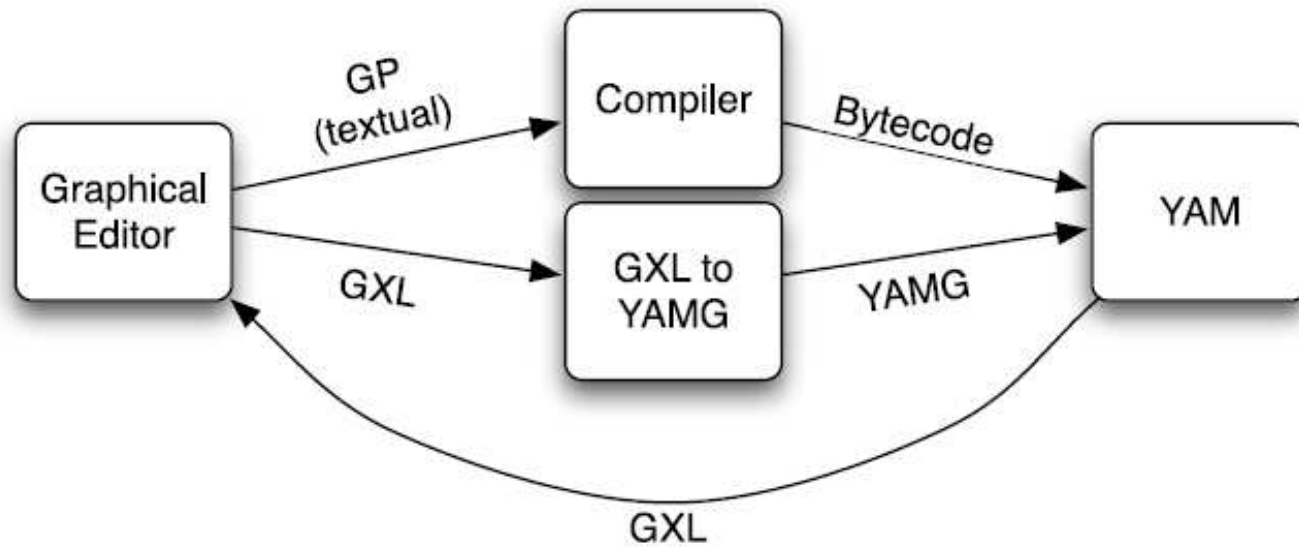
```
main = if NotComplete then No else Yes.
```

```
NotComplete = {Loop, ParallelEdges, UnconnectedNodes}.
```

if vs. try

- ▶ Powerful
- ▶ `if` hides the result of rule applications in the guard
 - ▶ Useful for **destructive** tests, e.g. trees
- ▶ `try` does not

Implementation



Checking Graph Properties with GP

- └ GP Refresher
- └ Implementation

Implementation

The screenshot shows the Graph Programs application window. The interface includes a menu bar (File, Help), a left sidebar with 'Graphs' and 'Programs' sections, and a main workspace with tabs for 'Program Editor', 'Graph Editor', 'Execution', 'Results', and 'Options'. The 'Program Editor' tab is active, showing a rule named 'T2'. The rule is defined by two diagrams: the left diagram shows a node '1: x_1' with an arrow pointing to a node '2: z', and the right diagram shows a node '1: x_1' with an arrow pointing to a node '2: z_1'. Below the diagrams are checkboxes for 'Layout', 'All Matches', and 'where:'. A 'next Label:' field contains 'a', and an 'Auto Increment' checkbox is present. A 'Program Text' area at the bottom contains the following code:

```
main = if NotConnected then No else Yes.  
NotConnected = Select; Tag!; Test.  
Tag = {T1, T2}.
```

On the right side of the workspace, there is a table with the following data:

Name	Type
z	INT
y	INT
x	INT

Implementation: Backtracking

- ▶ Optional (check box)
- ▶ Prolog-like
- ▶ Expensive
- ▶ Allows for some elegant programs
 - ▶ Hamiltonian path (later)

Implementation: Kinks

```
main = if SomeMacro then RuleSchema1 else RuleSchema2.  
SomeMacro = Rule1; Rule2; Rule3!; Test.
```

- ▶ Does not terminate on “large” input graphs,
- ▶ even with backtracking disabled

Implementation: Kinks

- ▶ Cannot use `if-then` without also `else`
- ▶ Workaround:
NullRule :

$$\emptyset \Rightarrow \emptyset$$

Implementation: Kinks

- ▶ Uses an updated semantics that drops while-do
- ▶ Can be simulated

`while C do P. ≡ (if C then P else fail)!; if C then fail else NullRule.`

- ▶ `fail?`

Implementation: Kinks

- ▶ Empty error messages (aaaarrghh!)
- ▶ Some differing notation
 - ▶ `where ~ edge 2 1` **vs.** `where not edge(2, 1)`

Today's Talk

1. Motivation
2. GP Refresher
3. Testing and describing graph properties with GP
4. Examples and demonstrations
5. Conclusions and future work
6. Questions

Describing Results

- ▶ GP takes a graph as input and returns a graph as output
- ▶ No additional variables in which to store a result (e.g. true/false indicating the existence of a property)
- ▶ Result must be **encoded** into the output graph

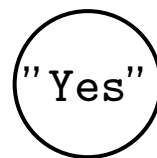
Describing Results

1. Generate a “Yes” or “No”-labelled node

Yes =

\emptyset

\Rightarrow



Describing Results

1. Generate a “Yes” or “No”-labelled node
2. Arbitrarily replace an existing label with “Yes” or “No”

Describing Results

1. Generate a “Yes” or “No”-labelled node
2. Arbitrarily replace an existing label with “Yes” or “No”
3. Return input graph with data encoded into the labels
 - ▶ Degree
 - ▶ Steps of a Hamiltonian path
 - ▶ etc.

Describing Results

1. Generate a “Yes” or “No”-labelled node
2. Arbitrarily replace an existing label with “Yes” or “No”
3. Return input graph with data encoded into the labels
 - ▶ Degree
 - ▶ Steps of a Hamiltonian path
 - ▶ etc.
4. Return single node or empty graph, following successful application of destructive rules

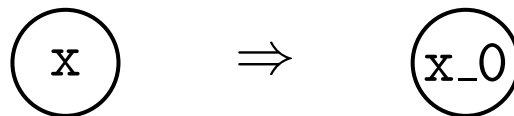
Approaching Graph Programming

- ▶ Assume integers
 - ▶ No duplication of “Yes” or “No”
- ▶ Information gathering approach
 - ▶ Store in tags
- ▶ Destructive approach
 - ▶ Some invariant holds after each reduction

Colours and Counting

- ▶ An infinite number of tags may be appended to a label.
- ▶ Of $l_t_1_t_2\dots_t_n$, l is the label and t_1 to t_n the tags.
- ▶ A single tag may represent colour
 - ▶ 0 = White (not visited)
 - ▶ 1 = Grey (being visited)
 - ▶ 2 = Black (“fully” visited, nothing more to do)
- ▶ Tags may also be used for counting
 - ▶ (Current) degree of a node
 - ▶ Sequence in a path

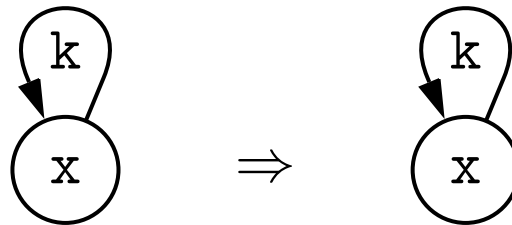
`InitNode(x : int) =`



Expecting Failure

`main = if Loop then No else Yes.`

`Loop(x, k : int) =`



- ▶ Fine in this case
- ▶ But implementation will not allow RSchema!; Test in the guard

Today's Talk

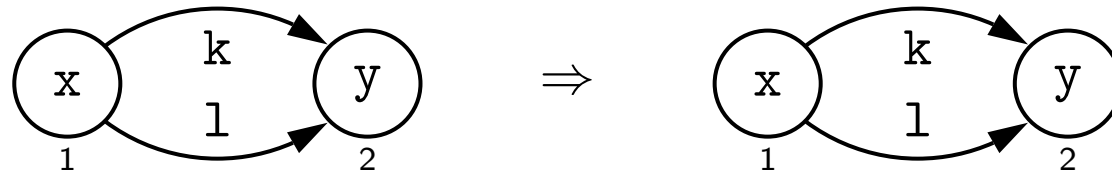
1. Motivation
2. GP Refresher
3. Testing and describing graph properties with GP
4. Examples and demonstrations
5. Conclusions and future work
6. Questions

Complete Graph

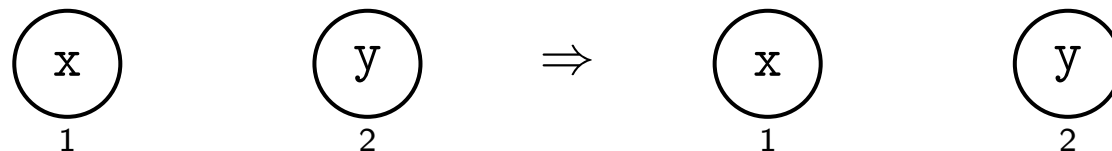
main = if NotComplete then No else Yes.

NotComplete = {Loop, ParallelEdges, UnconnectedNodes}.

ParallelEdges(x, y, k, l : int) =



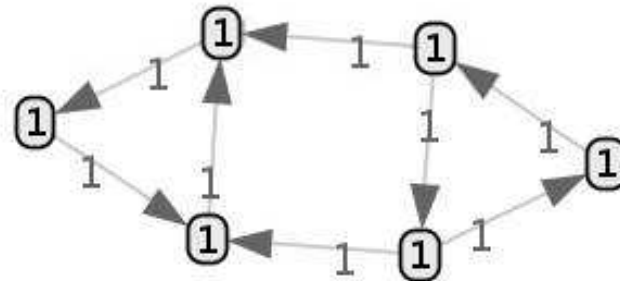
UnconnectedNodes(x, y : int) =



where not edge(1, 2)

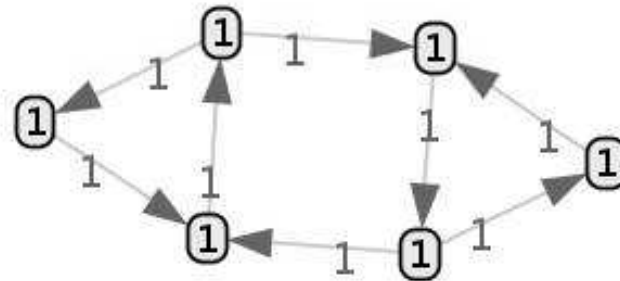
Unilaterally Connected Graph

- ▶ $\forall v_i, v_j \in V_G$, there is one of:
 1. A path from v_i to v_j
 2. A path from v_j to v_i
 3. Paths in both directions
- ▶ Unilaterally connected graphs are also weakly connected

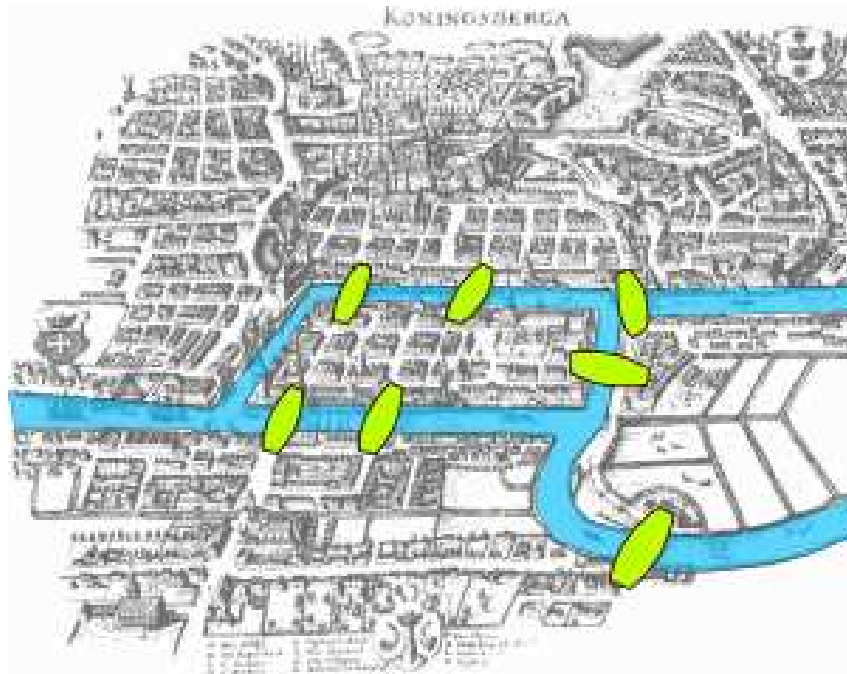


Strongly Connected Graph

- ▶ $\forall v_i, v_j \in V_G$, there is a path from v_i to v_j
- ▶ Strongly connected graphs are also unilaterally connected graphs
- ▶ Same program as `CheckUnilaterallyConnected`, except for one rule schema!



Eulerian Circuit



[Image from Wikipedia]

Eulerian Circuit

- ▶ A connected graph is **Eulerian** if every node has an even degree
- ▶ Test by reducing pairs of edges for as long as possible, and test for a remaining edge
- ▶ Computing the **Eulerian circuit** is more complicated

Eulerian Circuit

► Fleury's algorithm

1. Pick any node to start: "leading node"
2. Pick an edge connected to the node, subject to the **rule** below
3. Darken the edge
4. Move the "leading node" across the edge
5. Repeat 2-4 until returning to the initial node

RULE: Never cross a bridge of the reduced graph unless there is no other choice.

Hamiltonian Path

- ▶ Path that includes every single node in V_G exactly once
- ▶ NP-complete
- ▶ Heuristics unhelpful!
 - ▶ We do not want to weaken the “exactly once” condition
- ▶ Elegant, expensive program, making use of backtracking.

Today's Talk

1. Motivation
2. GP Refresher
3. Testing and describing graph properties with GP
4. Examples and demonstrations
5. Conclusions and future work
6. Questions

Conclusions and Further Work

- ▶ Have a go:
 - ▶ `http://www-course.cs.york.ac.uk/gra/`
 - ▶ `http://www.cs.york.ac.uk/~cmp501/gp/`
- ▶ Conclusions
- ▶ Further work
 - ▶ Verification of graph programs
 - ▶ Implementation improvements

Today's Talk

1. Motivation
2. GP Refresher
3. Testing and describing graph properties with GP
4. Examples and demonstrations
5. Conclusions and future work
6. Questions

Questions?