Bilbo: a domain-specific language for graphs

Christopher Macca

Imperial College London

Why Graphs?

Graphs *explicitly* materialise relationships between data points

Fig 1: A portion of the London Underground map [1].

Why Not Tables?

Tables break important relationships between data points that require ad hoc joining operations to reform

Source	Target	Line
West Kensington	Earl's Court	District
Earl's Court	Gloucester Road	District
Gloucester Road	South Kensington	District
South Kensington	Knightsbridge	Piccadilly

Table 1: A tabular form of the London Underground.

Applications of Graphs

Graphs are ubiquitous in science and engineering

- Analysing social networks
- Creation and management of public transport systems
- Solving global water scarcity
- Detecting money laundering
- Use in algorithms and data structures

All benefit from an umbrella of graph theory

This motivates the need for a *productive* and *programatic* tool for working with graphs $\rightarrow$ *graph programming*

Existing Tools for Graph Programming

Existing tools for graph programming come in four formats

- Graph databases
- Graph libraries and embedded DSLs
- Graph DSLs
- Graph rewriting tools

Each format, as a consequence of desired use case, has properties that impinge on productivity

Existing Tools for Graph Programming

Tool Format	Querying	Transformation	Non-Graph Program State	Graph-Specific Syntax	Graph-Specific Semantics
Graph database	✓	✓	✗	✓	✓
Graph library	✓	✓	✓	✗	✗
Graph DSL	✓	✓	✓	✓	✓
Graph rewriting tool	✗	✓	✗	✓	✓

Table 2: Comparison of graph programming tool formats.

Existing Tools for Graph Programming

Tool Format	Querying	Transformation	Non-Graph Program State	Graph-Specific Syntax	Graph-Specific Semantics
Graph database	✓	✓	✗	✓	✓
Graph library	✓	✓	✓	✗	✗
Graph DSL	✓	✓	✓	✓	✓
Graph rewriting tool	✗	✓	✗	✓	✓

Table 2: Comparison of graph programming tool formats.

However, existing DSLs prioritise performance, not productivity

The Bilbo Language

Bilbo

- A graph DSL
- Design objectives of *expressivity* and *productivity* with graphs
- Liberates programmers from handling low-level data structures
- No unrelated language constructs

Nodes and Types

```
type City = capital, population
id = "London"
load = City(True, 100)
ldn = id::load
```

Bilbo Basics

Nodes and Types

```
type City = capital, population
id = "London"
load = City(True, 100)
ldn = id::load
```

Graphs

```
bham = "Birmingham"::City(False, 15)
man = "Manchester"::City(False, 5)

road = [ldn, <120>, bham, <90>, man]
```

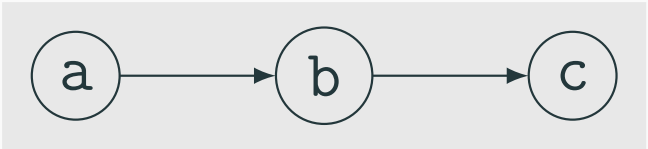
Graphs

```
1 bham = "Birmingham"::City(False, 15)
2 man = "Manchester"::City(False, 5)
3
4 road = [ldn, <120>, bham, <90>, man]
```

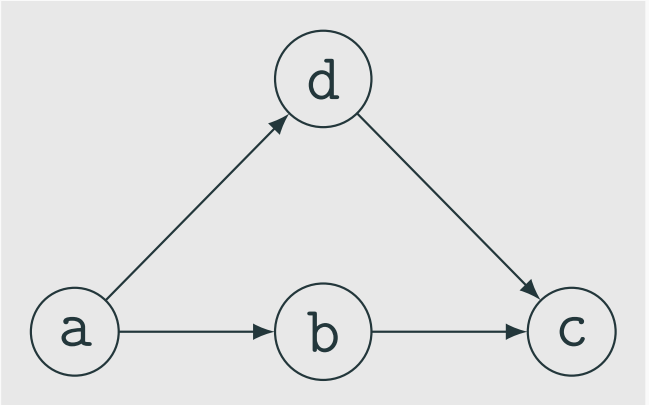
road

Combining Graphs

g

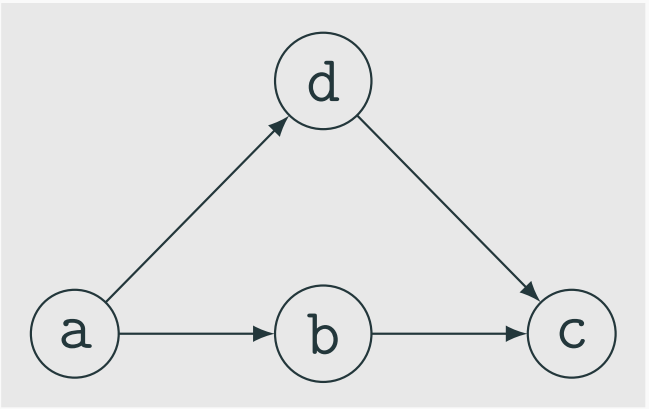


h

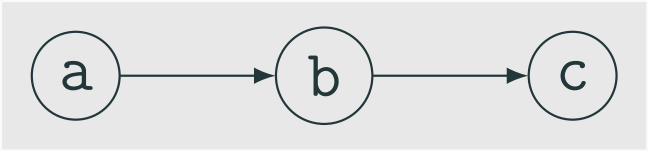


Combining Graphs

h



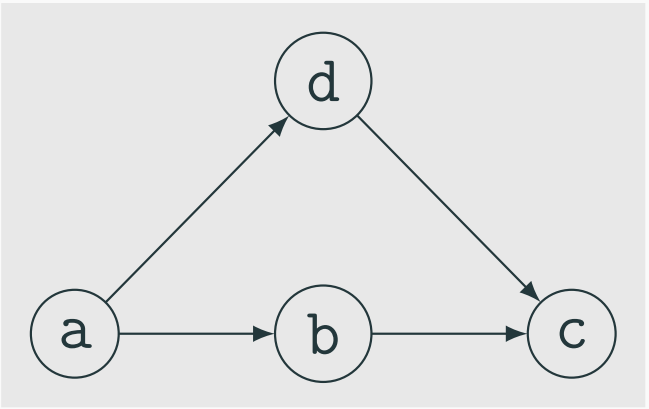
g



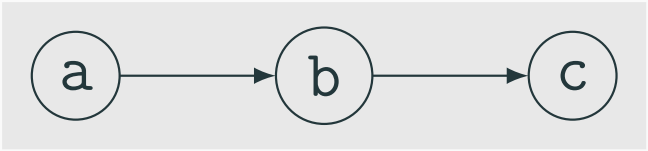

```
g = [a, >, b, >, c]
```

Combining Graphs

h



g

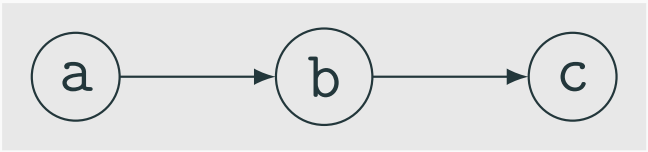



```
g = [a, >, b, >, c]
```

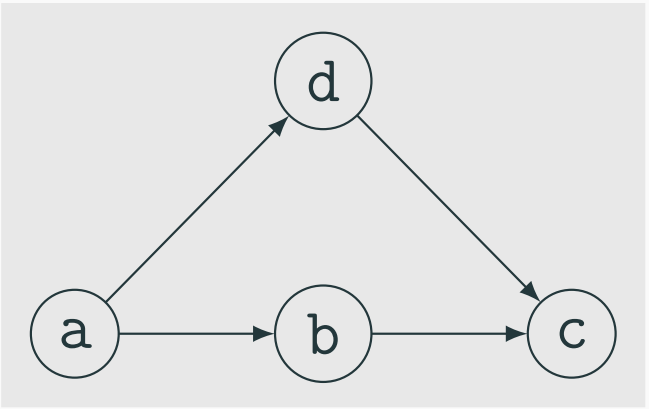
```
h = [a, >, b, >, c] + [a, >, d, >, c]
```

Combining Graphs

g



h



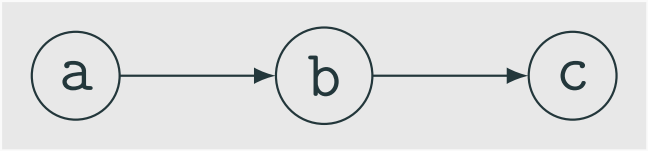

```
g = [a, >, b, >, c]
```

```
h = [a, >, b, >, c] + [a, >, d, >, c]
```

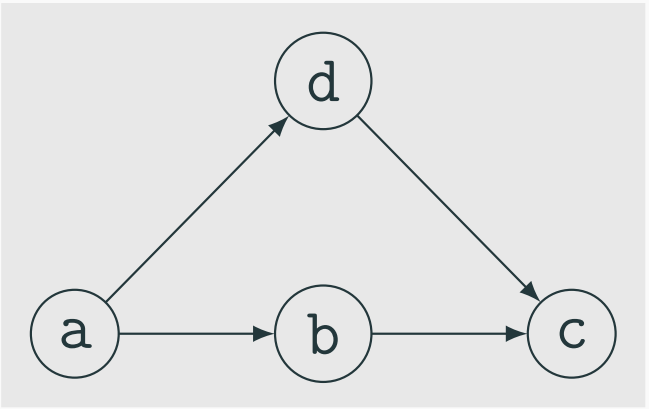
```
h = g + [a, >, d, >, c]
```

Combining Graphs

g



h




```
g = [a, >, b, >, c]
```

```
h = [a, >, b, >, c] + [a, >, d, >, c]
```

```
h = g + [a, >, d, >, c]
```

```
h = [a,>,b] + [b,>,c] + [a,>,d] + [d,>,c]
```

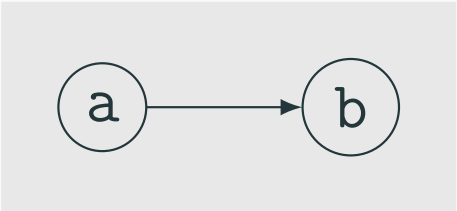
Graph Transformation

```
1 def removeEdge(g) =
2     match g
3     | [x,>,y] => become [x,y]
```

Graph Transformation

```
1 def removeEdge(g) =
2     match g
3     | [x,>,y] => become [x,y]
```

g

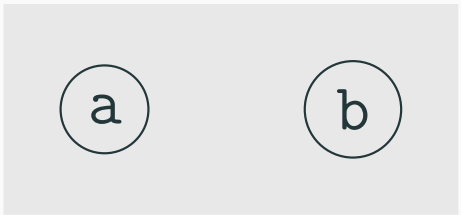



```
1 g = [a,>,b]
2 g' = g >> removeEdge
```

Graph Transformation

```
1 def removeEdge(g) =
2     match g
3     | [x,>,y] => become [x,y]
```

g’



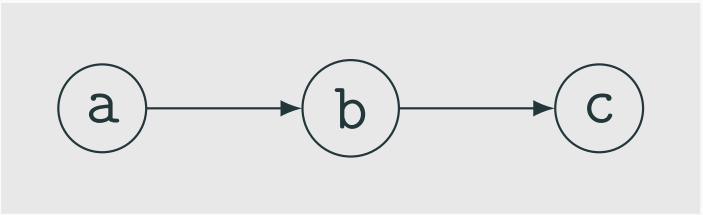

```
1 g = [a,>,b]
2 g' = g >> removeEdge
```

Nondeterminism

```
1 def removeEdge(g) =
2     match g
3     | [x,>,y] => become [x,y]
```

What if...

g

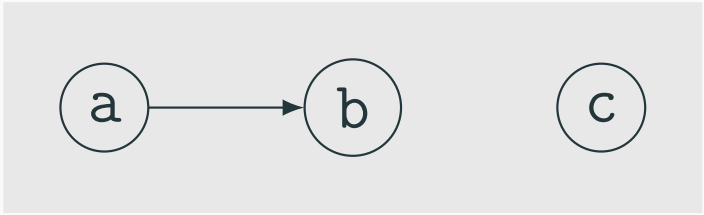



```
1 g  = [a, >, b, >, c]
2 g' = g >> removeEdge
```

Nondeterminism

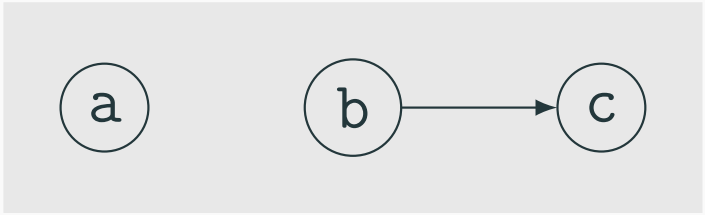
```
1 def removeEdge(g) =
2     match g
3     | [x,>,y] => become [x,y]
```

g’



or

g’

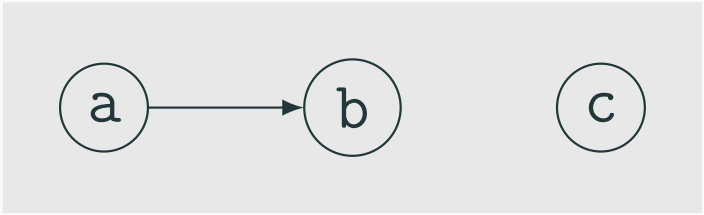



```
1 g  = [a, >, b, >, c]
2 g' = g >> removeEdge
```

Nondeterminism

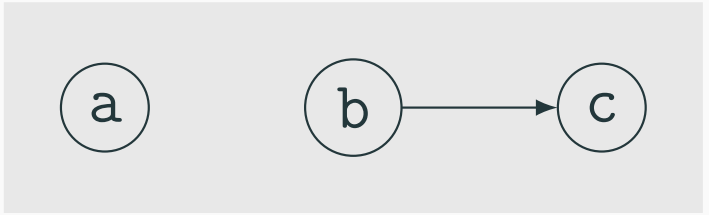
```
1 def removeEdge(g) =
2     match g
3     | [x,>,y] => become [x,y]
```

g’



and

g’

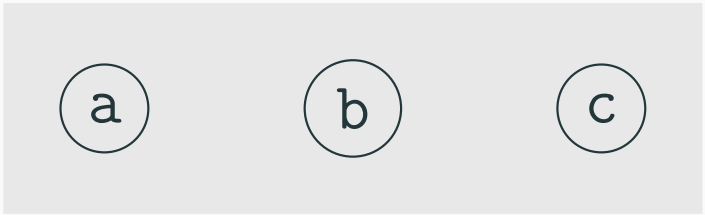



```
1 g  = [a, >, b, >, c]
2 g' = g >> removeEdge
```

Nondeterminism

```
1 def removeEdge(g) =
2     match g
3     | [x,>,y] => become [x,y]
```

g’

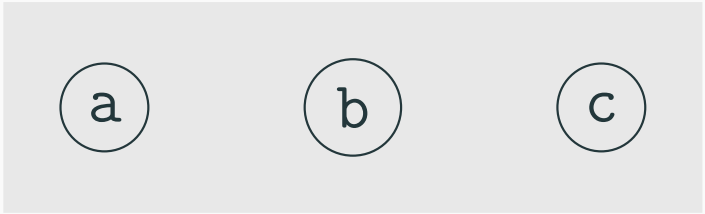



```
1 g  = [a, >, b, >, c]
2 g' = g >> removeEdge ** 2
```

Nondeterminism

```
def removeEdge(g) =
    match g
    | [x,>,y] => become [x,y]
```

g’



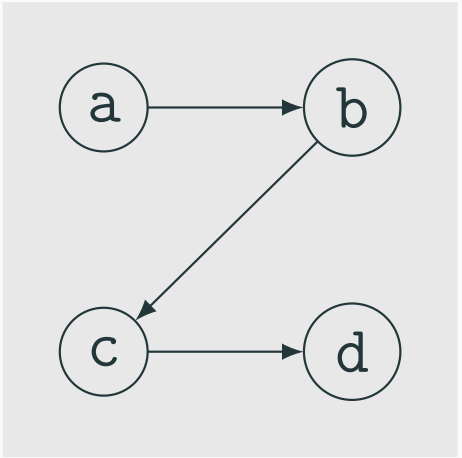

```
g  = [a, >, b, >, c]
g' = g >> removeEdge ** 2
```

What if we don’t know how many edges `g` has?

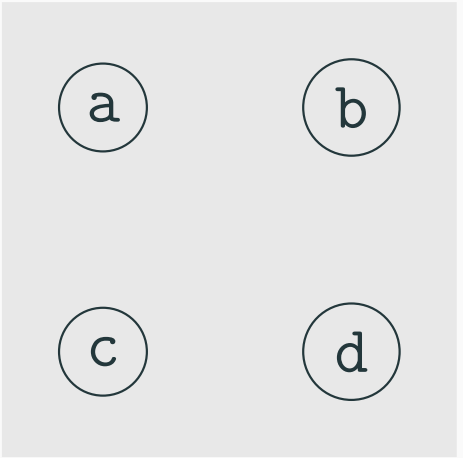
As-Long-As-Possible Application

```
1 def removeEdge(g) =
2     match g
3     | [x,>,y] => become [x,y]
```

g



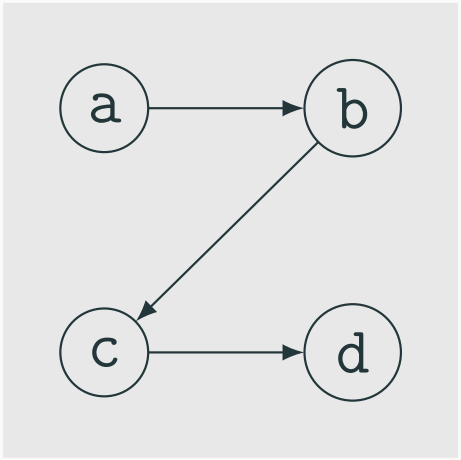
g’



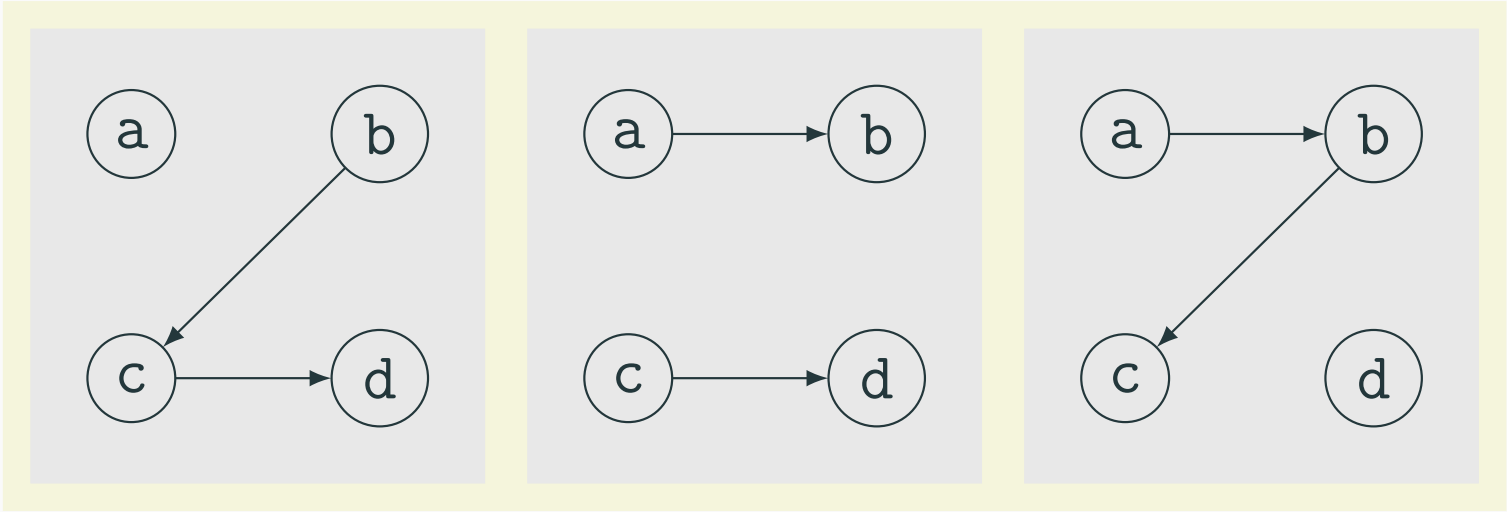

```
1 g  = [a, >, b, >, c, >, d]
2 g' = g >> removeEdge!
```

Wasted Computation

g

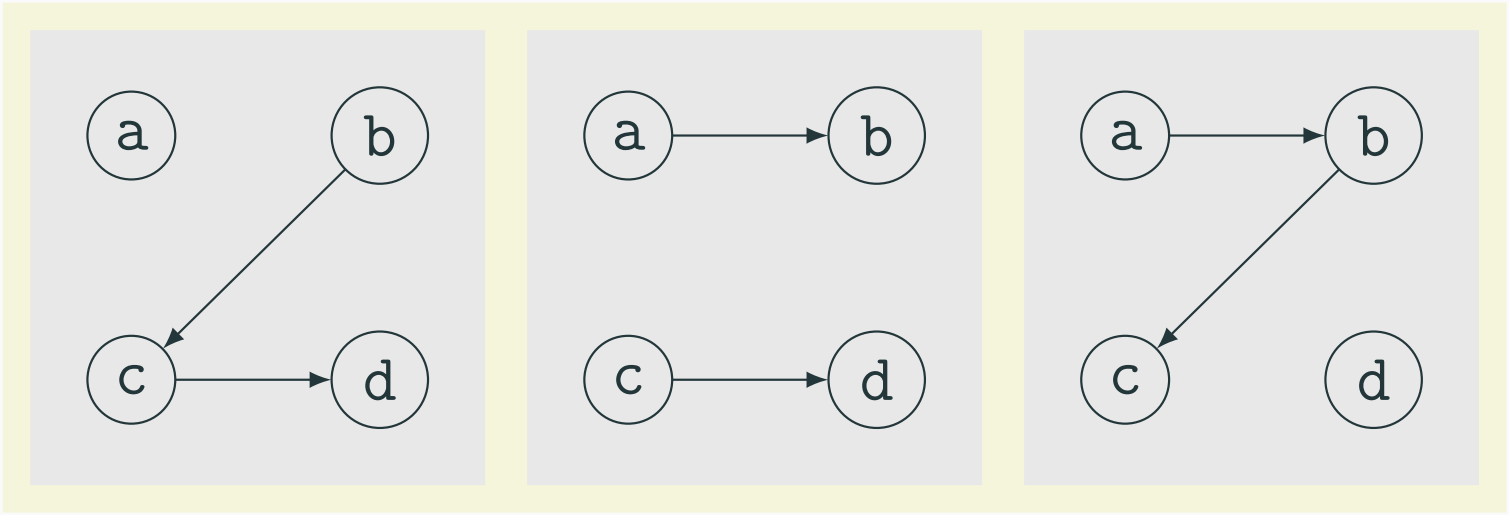


i_1



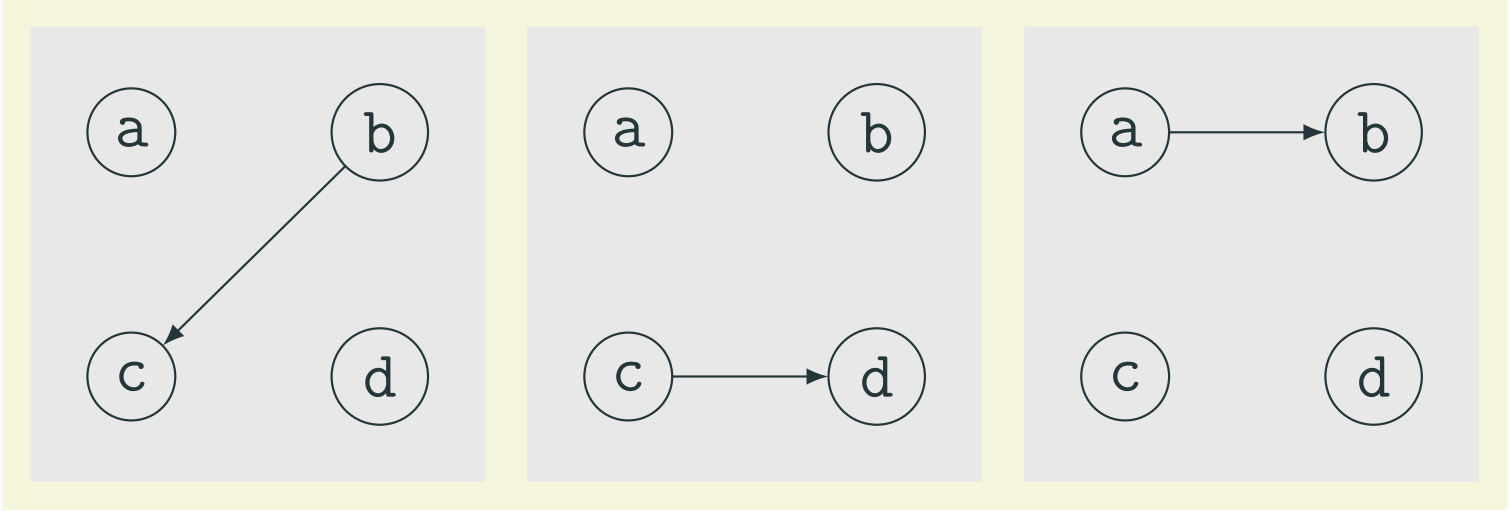
Wasted Computation

i_1



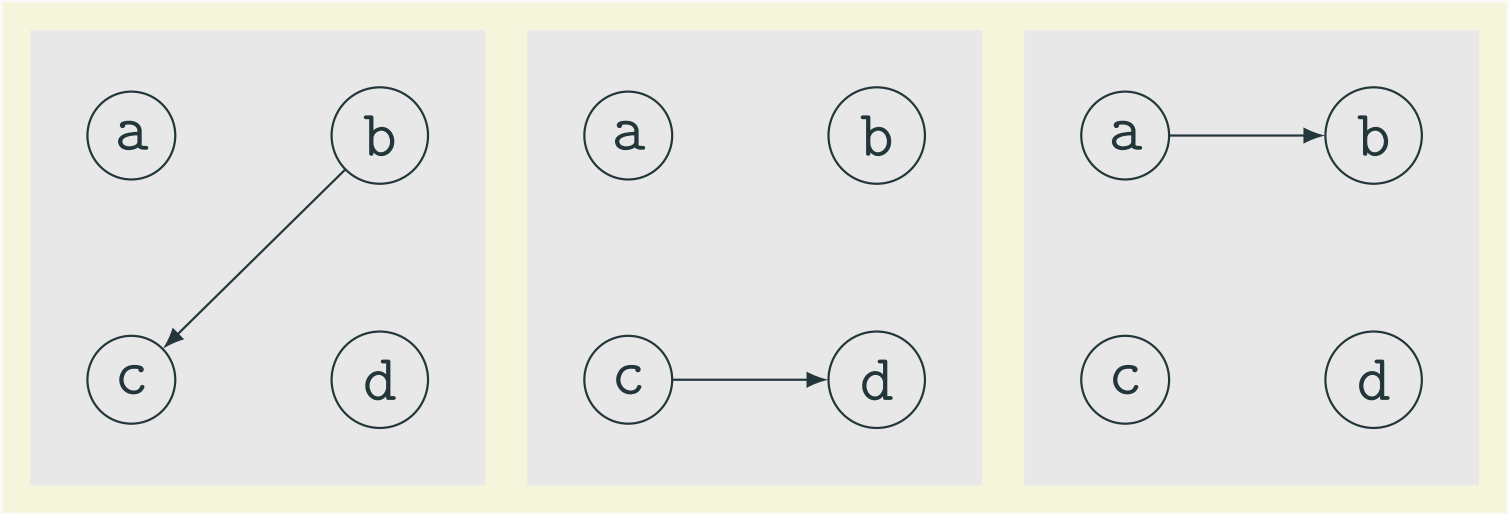
$\Downarrow$

i_2

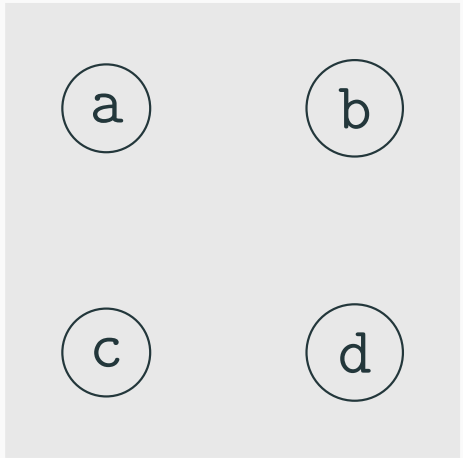


Wasted Computation

i_2



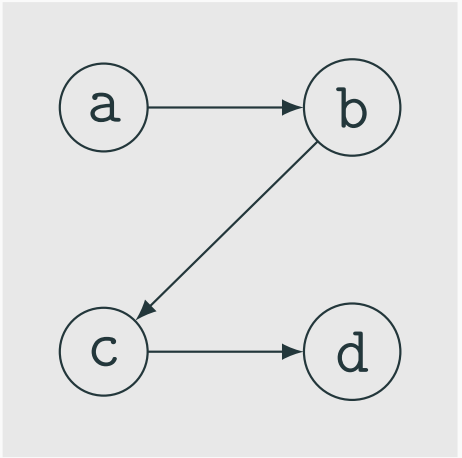
g'



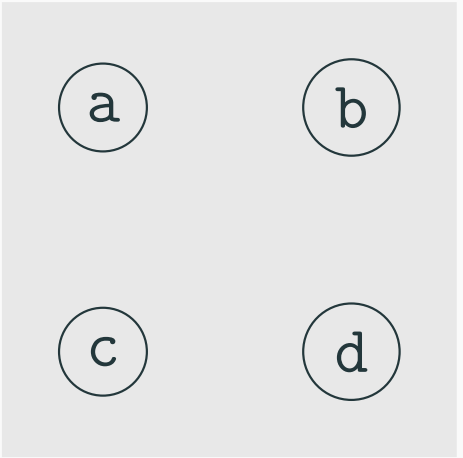
One-Match Application

```
1 def removeEdge(g) =
2     match g
3     | [x,>,y] => become [x,y]
```

g



g’




```
1 g  = [a, >, b, >, c, >, d]
2 g' = g >> $removeEdge!
```

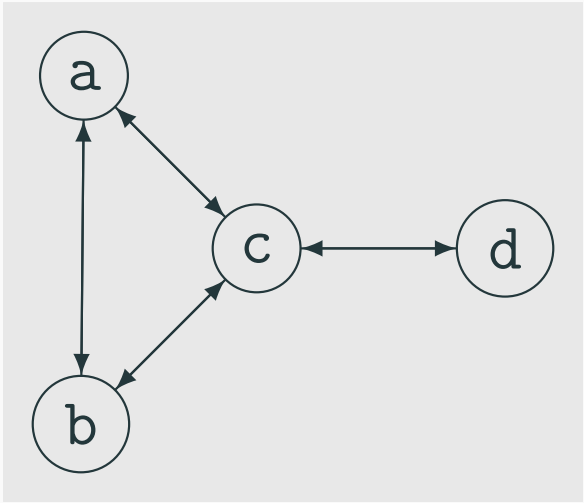
Solving Graph Problems in Bilbo

The Vertex Colouring Problem

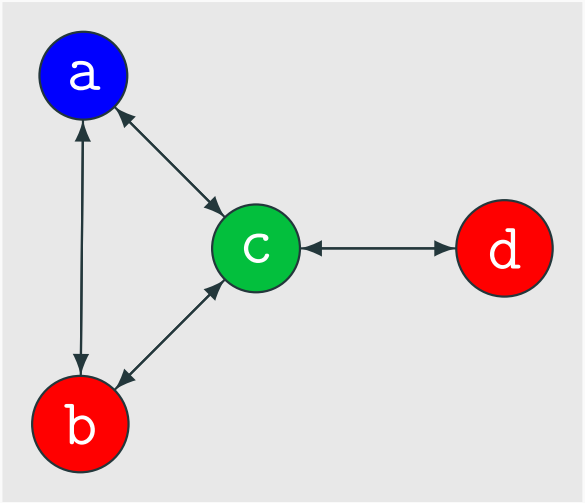
Problem

Assign a colour to each node such that no adjacent nodes have the same colour [2]

g



c1



Solving the Vertex Colouring Problem

A type for the vertex colour

```
1 type Vertex = colour
```

Solving the Vertex Colouring Problem

Transform to initialise all node loads to a `Vertex` with colour `0`

```
1 type Vertex = colour
2
3 def init(g) =
4     match g
5     | [a] where not (#a is Vertex) =>
6         become [a::Vertex(0)]
```

Solving the Vertex Colouring Problem

Transform to change colours of adjacent nodes with same colour

```
1  type Vertex = colour
2
3  def init(g) =
4      match g
5      | [a] where not (#a is Vertex) =>
6          become [a::Vertex(0)]
7
8  def changeCol(g) =
9      match g
10     | [a,>,b]
11     where (a..colour == b..colour) =>
12         b..colour = b..colour + 1
13         become [a,>,b]
```

Solving the Vertex Colouring Problem

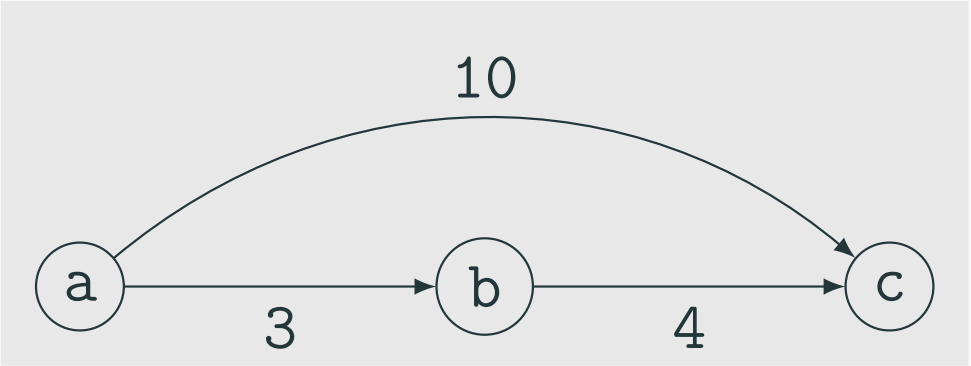
```
1  type Vertex = colour
2
3  def init(g) =
4      match g
5      | [a] where not (#a is Vertex) =>
6          become [a::Vertex(0)]
7
8  def changeCol(g) =
9      match g
10     | [a,>,b]
11     where (a..colour == b..colour) =>
12         b..colour = b..colour + 1
13         become [a,>,b]
14
15 vcolour = $init! |> $changeCol!
```

The Shortest Path Problem

Problem

Label each node with its shortest distance from a start node [2]

g

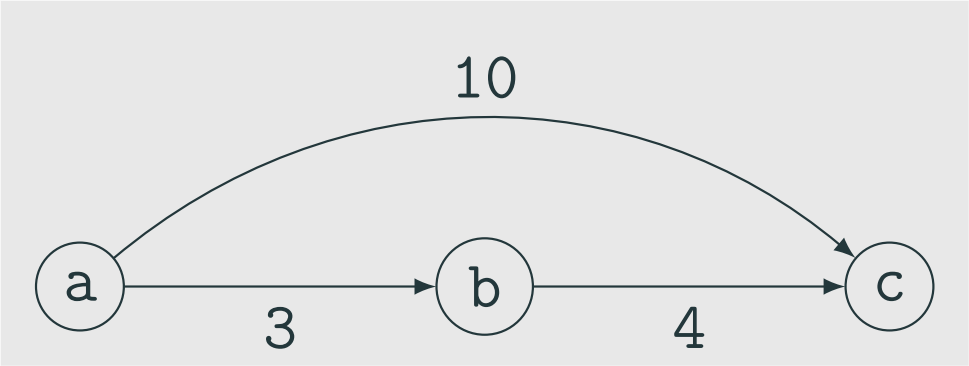


The Shortest Path Problem

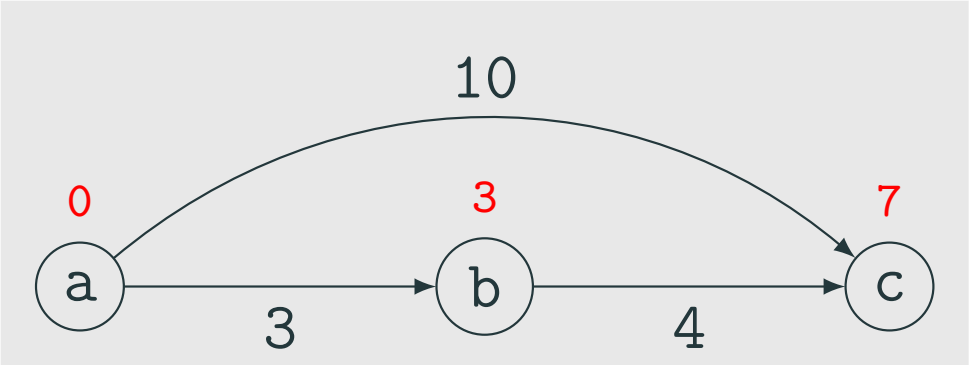
Problem

Label each node with its shortest distance from a start node [2]

g



s



Solving the Shortest Path Problem

A type for nodes to hold shortest distance value and a valid flag

```
1 type Place = dist, valid
```

For simplicity we'll assume that all node loads are of type `Place` and have `valid` set to `False`, other than the start node which has `valid` set to `True` and `dist` set to zero

Solving the Shortest Path Problem

Transform for setting initial `dist` value

```
type Place = dist, valid

def add(g) =
    match g
    | [x,w>,y]
    where x..valid and not y..valid =>
        y..valid = True
        y..dist = x..dist+w
        become [x,w>,y]
```

Solving the Shortest Path Problem

Transform for reducing `dist` values

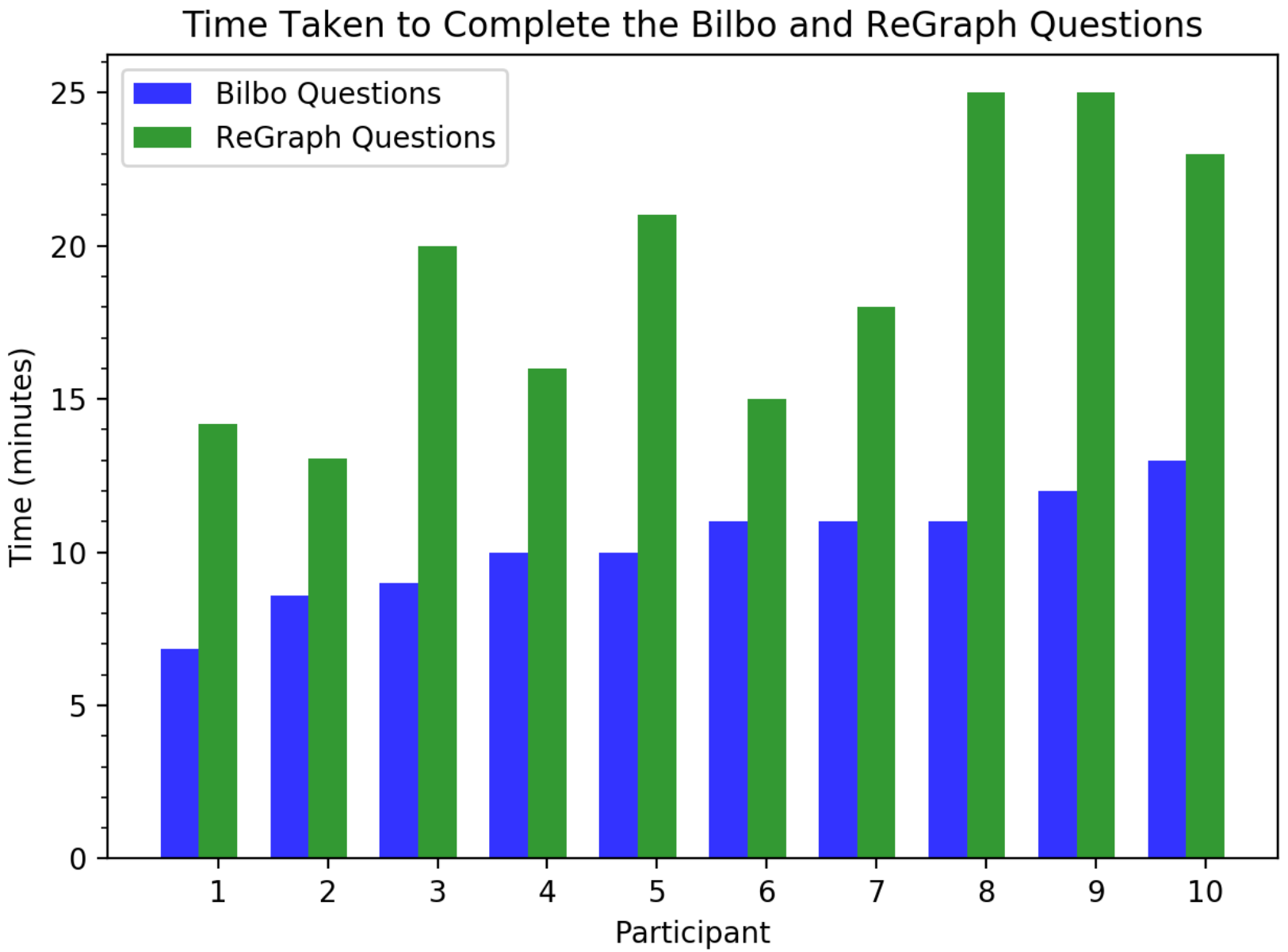
```
1  type Place = dist, valid
2
3  def add(g) =
4      match g
5      | [x,w>,y]
6      where x..valid and not y..valid =>
7          y..valid = True
8          y..dist = x..dist+w
9          become [x,w>,y]
10
11 def reduce(g) =
12     match g
13     | [x,w>,y] where y..dist > x..dist+w =>
14         y..dist = x..dist+w
15         become [x,w>,y]
```

Solving the Shortest Path Problem

```
1  type Place = dist, valid
2
3  def add(g) =
4      match g
5      | [x,w>,y]
6      where x..valid and not y..valid =>
7          y..valid = True
8          y..dist = x..dist+w
9          become [x,w>,y]
10
11 def reduce(g) =
12     match g
13     | [x,w>,y] where y..dist > x..dist+w =>
14         y..dist = x..dist+w
15         become [x,w>,y]
16
17 shortest = $add! |> $reduce!
```

User Testing

User Testing



User Testing

Comparing Bilbo to ReGraph [3], Bilbo was seen to be

- more productive (4.3 vs 2.4)
- more intuitive (4.4 vs 2.4)
- more readable (4.5 vs 2.3)

Participants said Bilbo is

- '*fantastically concise*'
- '*completely clear*'

Implementation

The Bilbo Interpreter

Bilbo has a reference interpreter written in F#

- Comes with an interactive REPL
- Cross-platform (linux, macOS and windows)
- Good test coverage
- Very easy to download and use

Conclusions

Achievements & Limitations

Achievements

- Achieves design goals of expressivity and productivity
- Can be used to solve graph problems at a high-level of abstraction
- Liberates programmers from handling low-level data structures
- All language constructs relate to graph programming

Achievements & Limitations

Achievements

- Achieves design goals of expressivity and productivity
- Can be used to solve graph problems at a high-level of abstraction
- Liberates programmers from handling low-level data structures
- All language constructs relate to graph programming

Limitations

- Subgraph isomorphism is NP-complete
- No formal language semantics

Conclusions

- Bilbo has a number of novel but useful features
- This work required an understanding of graph theory as well as language design
- Bilbo is computationally complete [4]
- Bilbo goes beyond computational completeness

Conclusions

- Bilbo has a number of novel but useful features
- This work required an understanding of graph theory as well as language design
- Bilbo is computationally complete [4]
- Bilbo goes beyond computational completeness

Future Work

- Paper submission to GCM 2020
- Continuing as open source project
- Creation of formal semantics with language reference

try Bilbo at:

`https://github.com/maccth/bilbo`

References i

- Transport for London, "Standard London Underground map - version A1, with larger type.." https://tfl.gov.uk/info-for/suppliers-and-contractors/map-sizes-and-formats.
- C. Bak, G. Faulkner, D. Plump, and C. Runciman, "A Reference Interpreter for the Graph Programming Language GP 2," *Electronic Proceedings in Theoretical Computer Science*, vol. 181, 04 2015.
- Kappa-Dev GitHub Organisation, "ReGraph." http://dev.executableknowledge.org/ReGraph/.

References ii


A. Habel and D. Plump, "Computational completeness of programming languages based on graph transformation," in *Foundations of Software Science and Computation Structures* (F. Honsell and M. Miculan, eds.), (Berlin, Heidelberg), pp. 230–245, Springer Berlin Heidelberg, 2001.