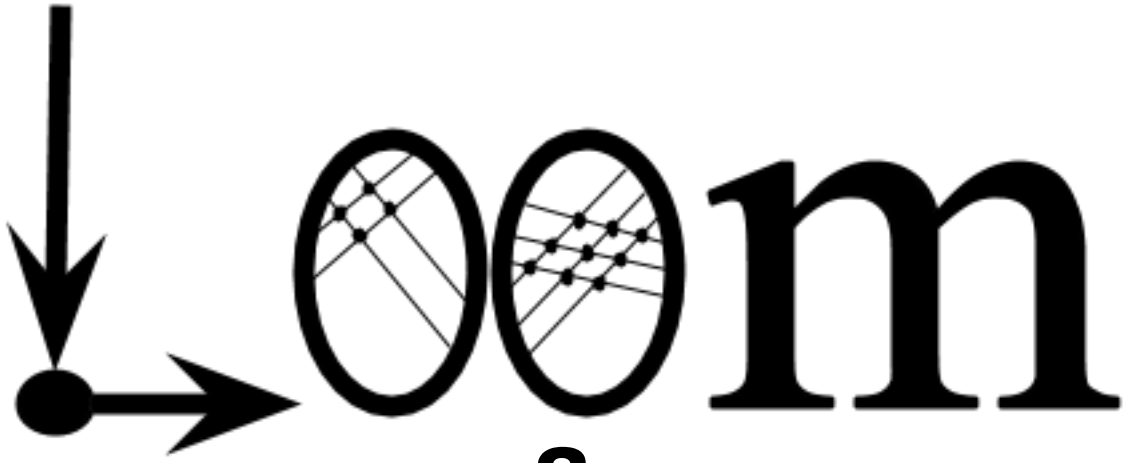


&

Functional Graphs in Clojure

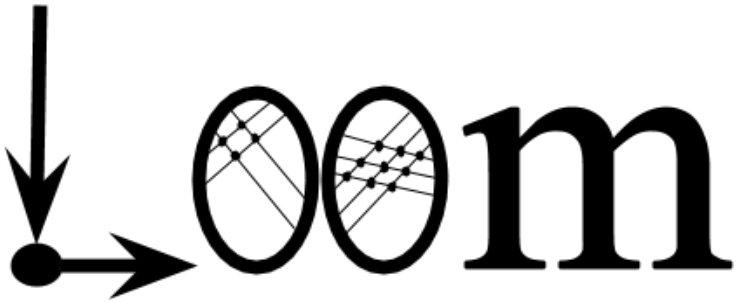
Aysylu Greenberg

@aysylu22

Google

Aysylu Greenberg

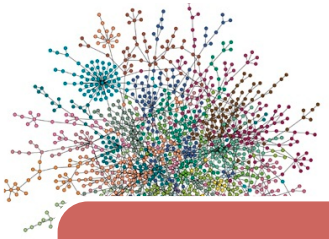
Google



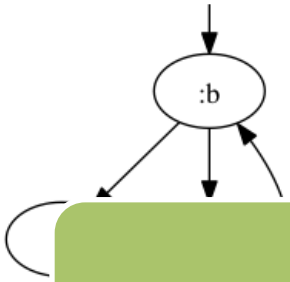
github.com/aysylu/loom

@aysylu22

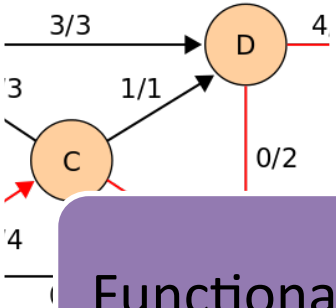
In this Talk



A Moment of Graph Theory



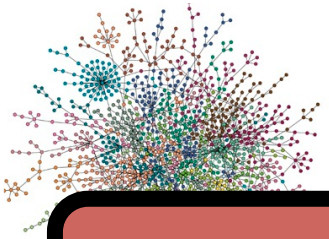
Graph API Design



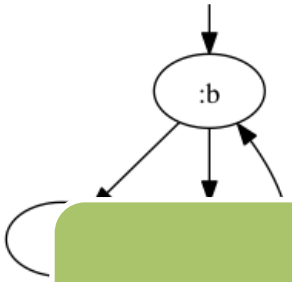
Functional Graph Algorithms

Loom Overview

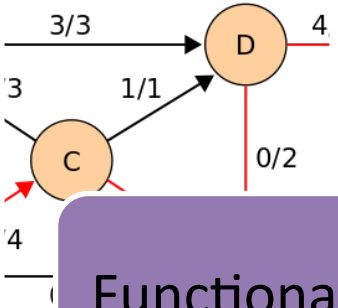
In this Talk



A Moment of Graph Theory



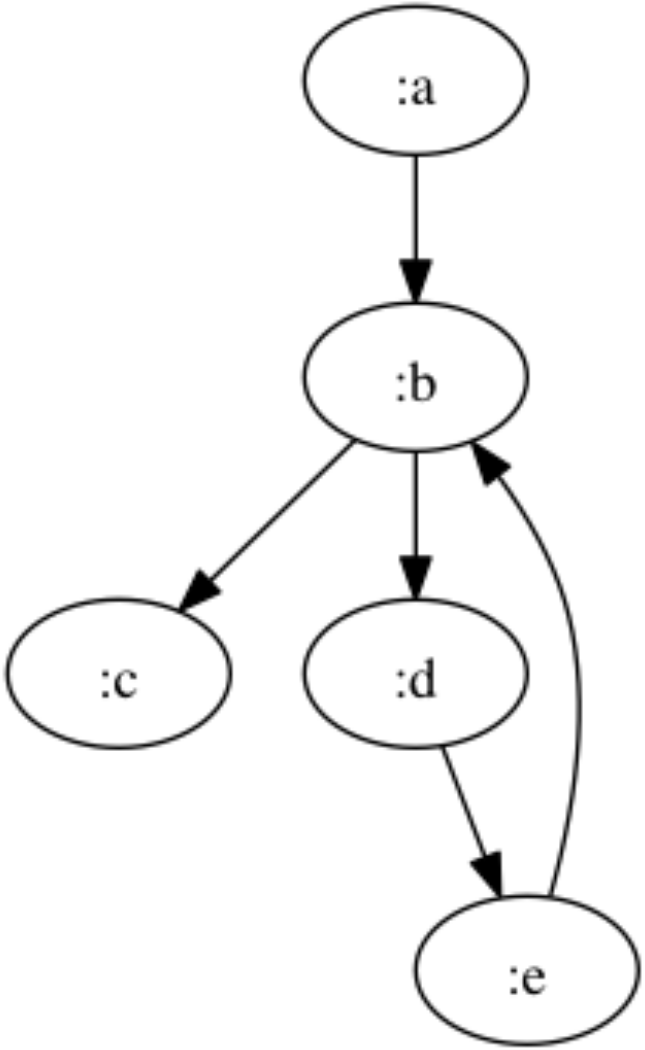
Graph API Design



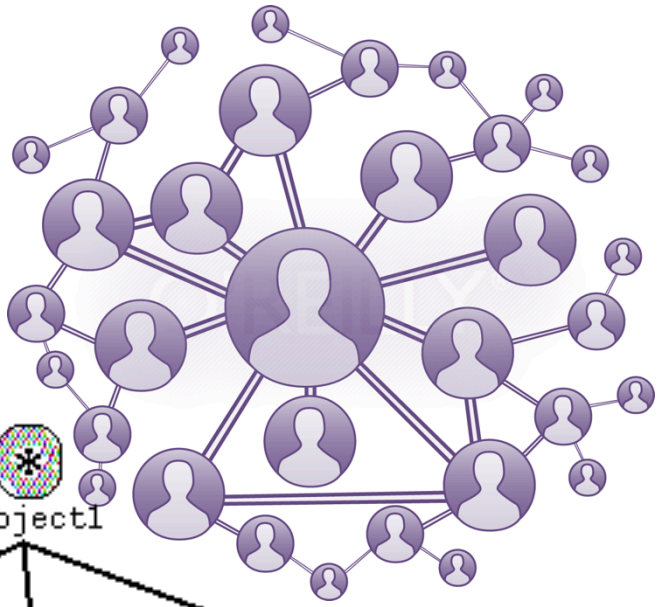
Functional Graph Algorithms

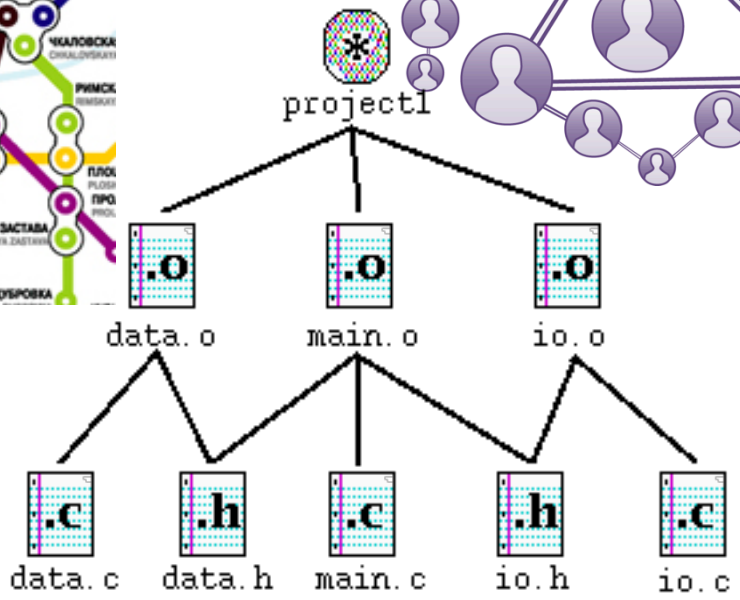
Loom Overview

Graphs

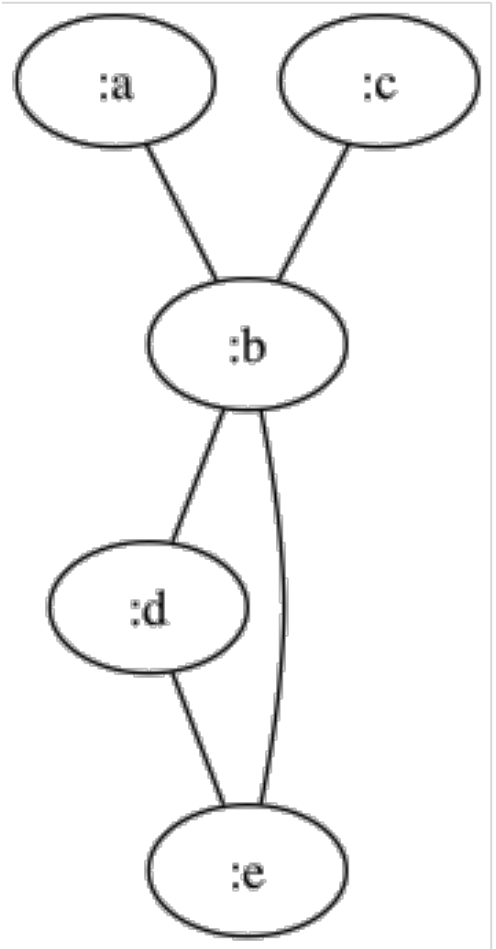


Graphs in Real Life

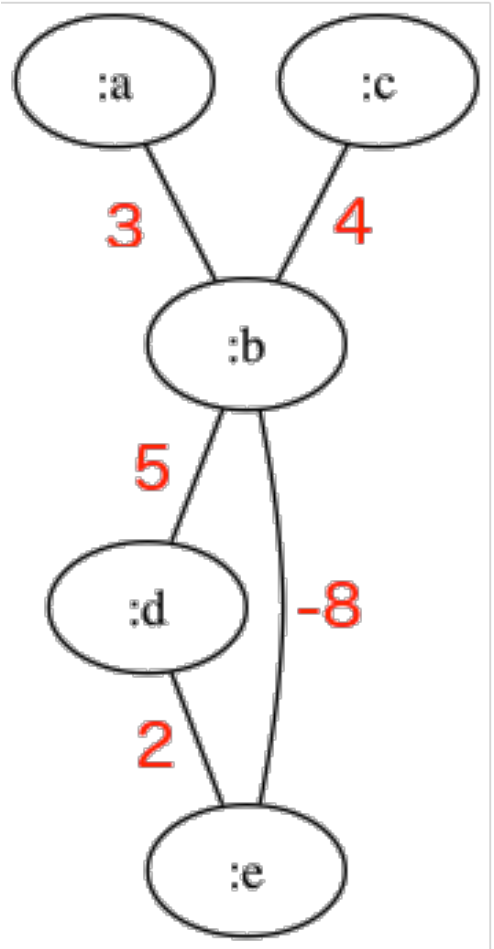




Types of Graphs

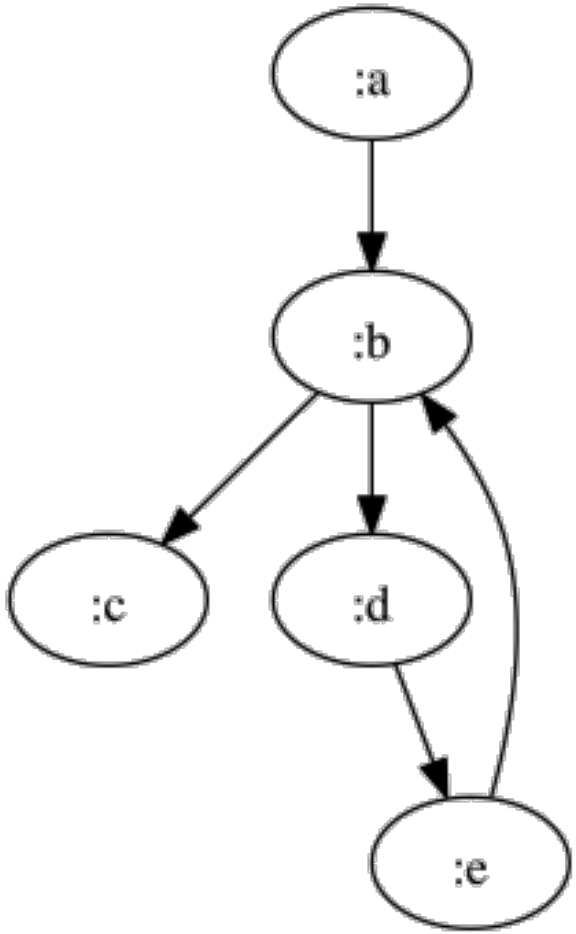


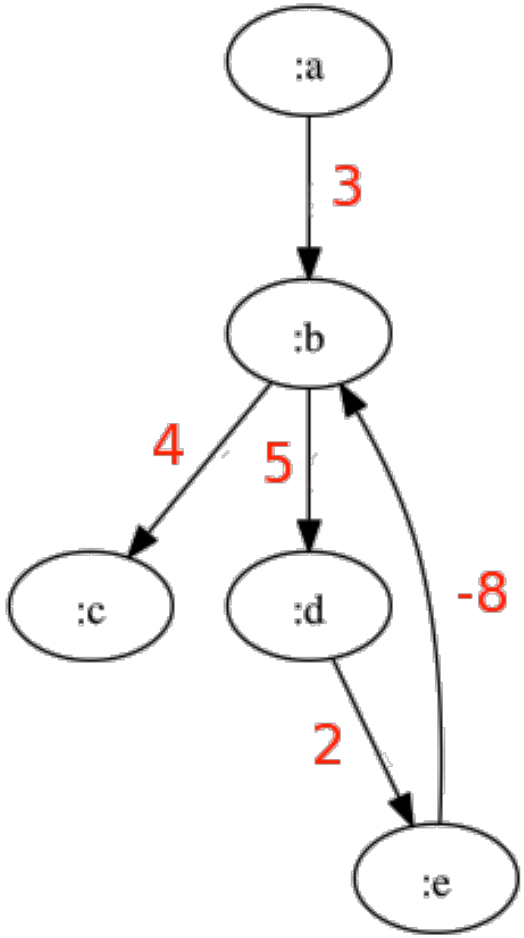
Simple Graph



Weighted Graph

Types of Graphs

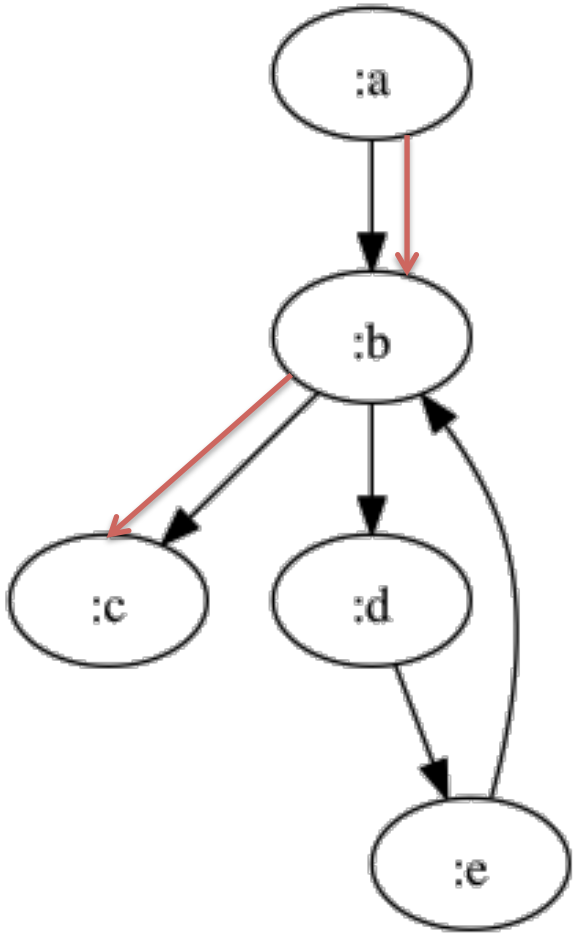




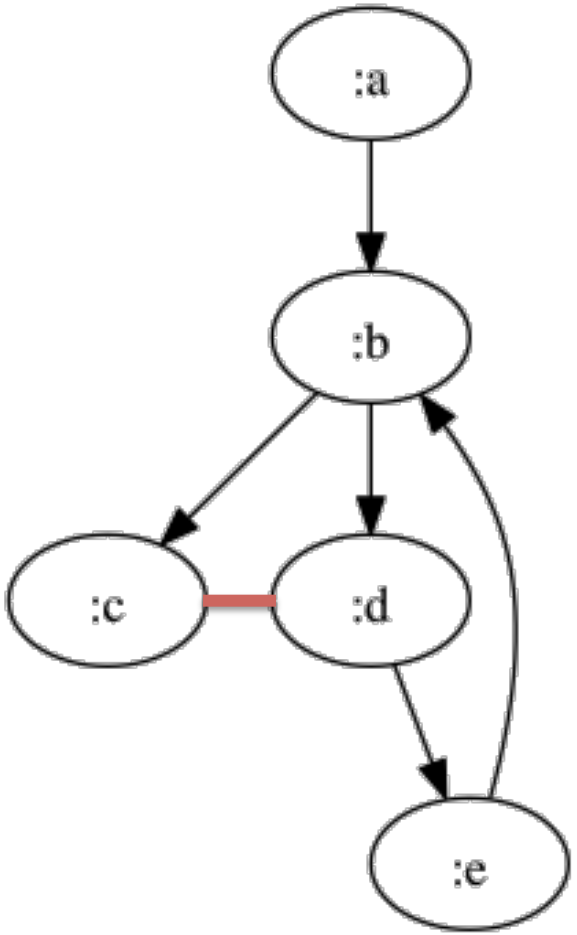
Directed Graph (Digraph)

Weighted Digraph

Types of Graphs

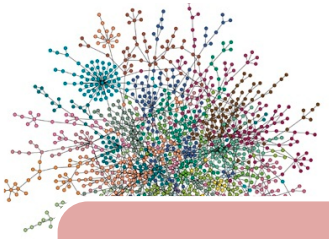


Multigraph

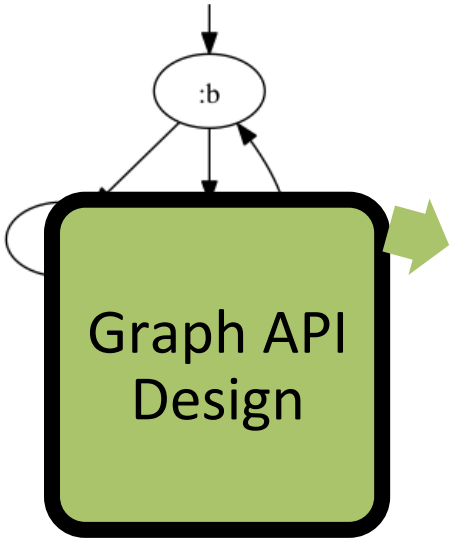


Mixed Graph

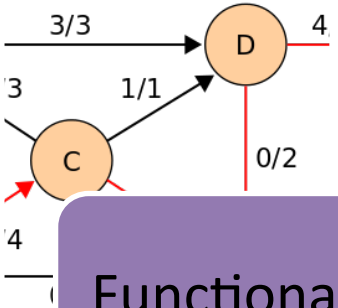
In this Talk



A Moment of Graph Theory



Graph API Design



Functional Graph Algorithms

Loom Overview

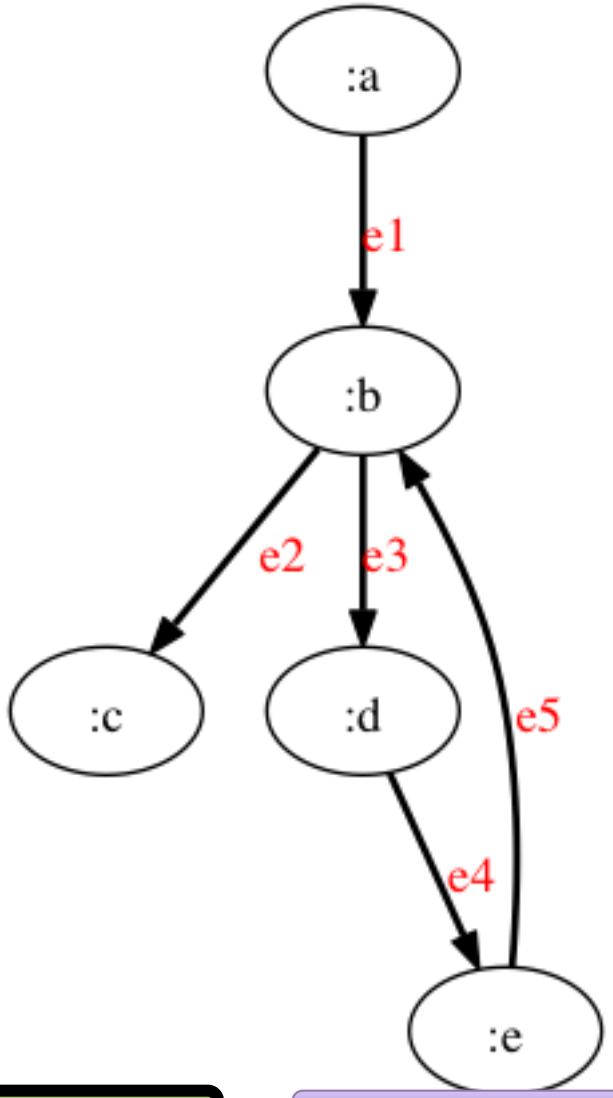
Graphs in OO World

```
class Node {
  string data;
}

class Edge {
  Node node1;
  Node node2;
}

class Graph {
  List<Node> nodes;
  List<Edge> edges;
}
```

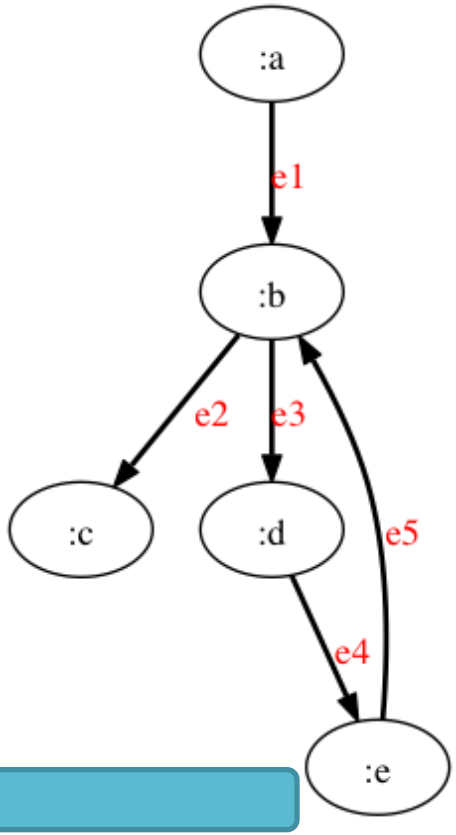
Graph Representations



Graph Representations

- Adjacency List

  {A: [B], B: [C, D], C: [], …}

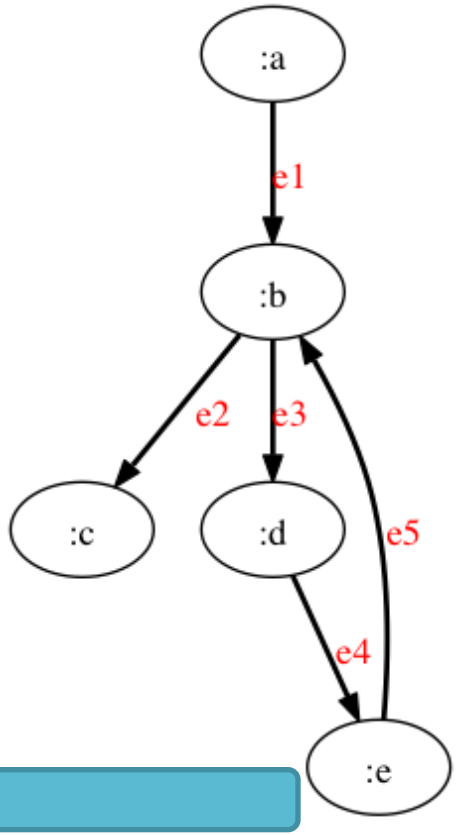


Graph Representations

- Adjacency List
- Incidence List

{A: [e1], B: [e2, e3], …,

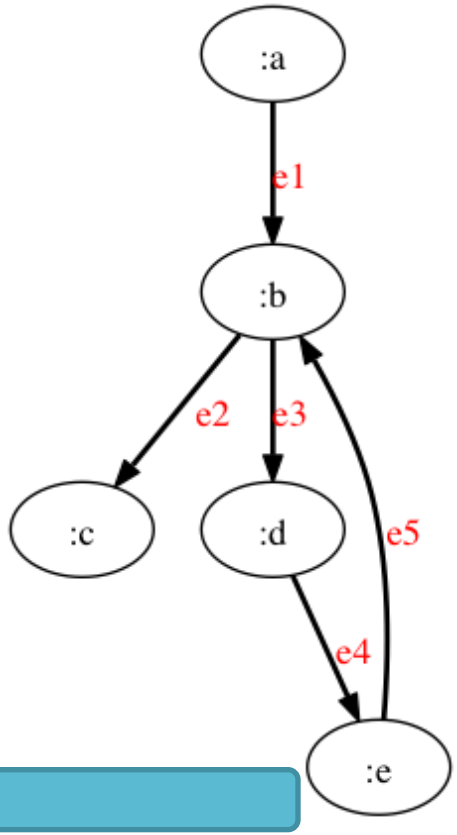
e1: [A, B], e2: [B, C], …}



Graph Representations

- Adjacency List
- Incidence List
- Adjacency Matrix

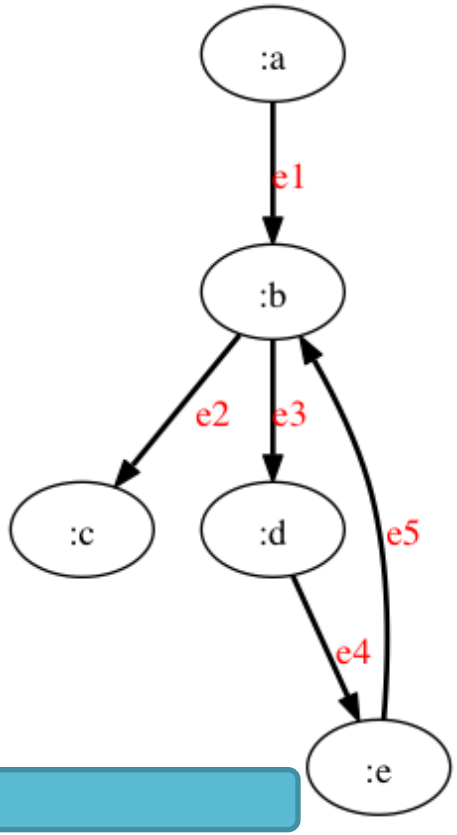
Source	Destination			
	A	B	…	E
A	0	1	…	∞
B	∞	0	…	∞
…	…	…	…	…
E	∞	1	…	0



Graph Representations

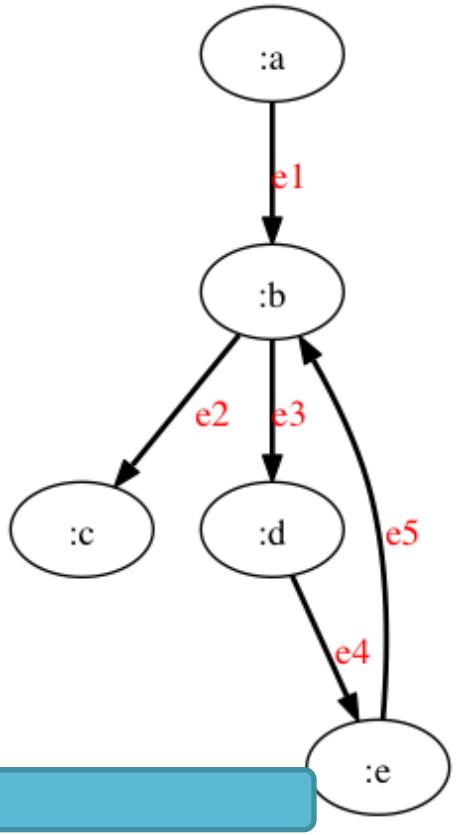
- Adjacency List
- Incidence List
- Adjacency Matrix
- Incidence Matrix

Nodes	Edges			
	e1	e2	..	e5
A	S	N	…	N
B	D	S	…	D
…	…	…	…	…
E	N	N	…	S



Graph Representations

- Adjacency List sparse graphs
- Incidence List
- Adjacency Matrix dense graphs
- Incidence Matrix



Graphs in FP World

- Structs, interfaces & implementations
- Flexible representation
- Immutable graphs

Graphs in Clojure: Graph

```
(defprotocol Graph
  (nodes [g])
  (edges [g])
  (has-node? [g node])
  (has-edge? [g n1 n2])
  (successors [g node])
  (out-degree [g node])
  (out-edges [g node]))
```

Graphs in Clojure: Digraph

```
(defprotocol Digraph
  (predecessors [g node])
  (in-degree [g node])
  (in-edges  [g node])
  (transpose [g]))
```

Graphs in Clojure: WeightedGraph

```
(defprotocol WeightedGraph
  (weight [g n1 n2]))
```

Graphs in Clojure: EditableGraph

```
(defprotocol EditableGraph
  (add-nodes* [g nodes])
  (add-edges* [g edges])
  (remove-nodes* [g nodes])
  (remove-edges* [g edges])
  (remove-all [g]))
```

Graphs in Clojure: Complex Graphs

How to create basic editable digraph?

```
(defrecord BasicEditableDigraph
  [nodes succs preds])
```

Graphs in Clojure: Complex Graphs

How to create basic editable digraph?

```
(defrecord BasicEditableDigraph
  [nodes succs preds])
(extend BasicEditableDigraph
  Graph …
  Digraph …
  EditableGraph …)
```

Graphs in Clojure: Complex Graphs

How to create basic editable digraph?

```
(defrecord BasicEditableDigraph
  [nodes succs preds])
(extend BasicEditableDigraph
  Graph
    (nodes [g])
    (edges [g])
    (has-node? [g node])
    (has-edge? [g n1 n2])
    (successors [g] [g node])
    (out-degree [g node])
)
```

Protocol definition
(not valid code!)

Graphs in Clojure: Complex Graphs

How to create basic editable digraph?

```
(defrecord BasicEditableDigraph
  [nodes succs preds])
(extend BasicEditableDigraph
  Graph default-graph-impl
  Digraph …
  EditableGraph …)
```

Graphs in Clojure: Complex Graphs

```
(def default-graph-impl
 {:edges
  (fn [g]
    (for [n1 (nodes g)
          e (out-edges g n1)]
      e))
  :nodes (fn [g] ...)
  ...})
```

```
(nodes [g])
(edges [g])
(has-node? [g node])
(has-edge? [g n1
n2])
(successors [g] [g
node])
(out-degree [g
node])
```




Graphs in Clojure: Complex Graphs

How to create basic editable digraph?

```
(defrecord BasicEditableDigraph
  [nodes succs preds])
(extend BasicEditableDigraph
  Graph default-graph-impl
  Digraph …
  EditableGraph …)
```

Graphs in Clojure: Complex Graphs

How to create basic editable digraph?

```
(defrecord BasicEditableDigraph
  [nodes succs preds])
(extend BasicEditableDigraph
  Graph default-graph-impl
  Digraph default-digraph-impl
  EditableGraph …)
```

Graphs in Clojure: Complex Graphs

```
(def default-digraph-impl
  {:transpose
   (fn [g]
     (assoc g
            :succs (:preds g)
            :preds (:succs g)))
   …})
```

```
(predecessors [g]
              [g node])
(in-degree [g node])
(transpose [g])
```




Graphs in Clojure: Complex Graphs

How to create basic editable digraph?

```
(defrecord BasicEditableDigraph
  [nodes succs preds])
(extend BasicEditableDigraph
  Graph default-graph-impl
  Digraph default-digraph-impl
  EditableGraph …)
```

Graphs in Clojure: Complex Graphs

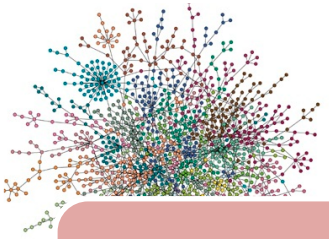
How to create basic editable digraph?

```
(defrecord BasicEditableDigraph
  [nodes succs preds])
(extend BasicEditableDigraph
  Graph default-graph-impl
  Digraph default-digraph-impl
  EditableGraph editable-graph-impl)
```

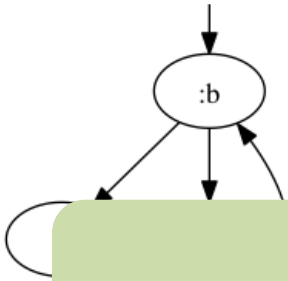
Graphs in FP World

- Structs, interfaces & implementations
- Flexible representation
- Immutable graphs

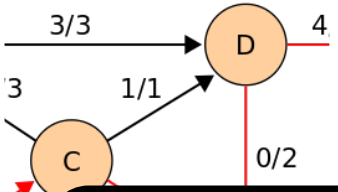
In this Talk



A Moment of Graph Theory



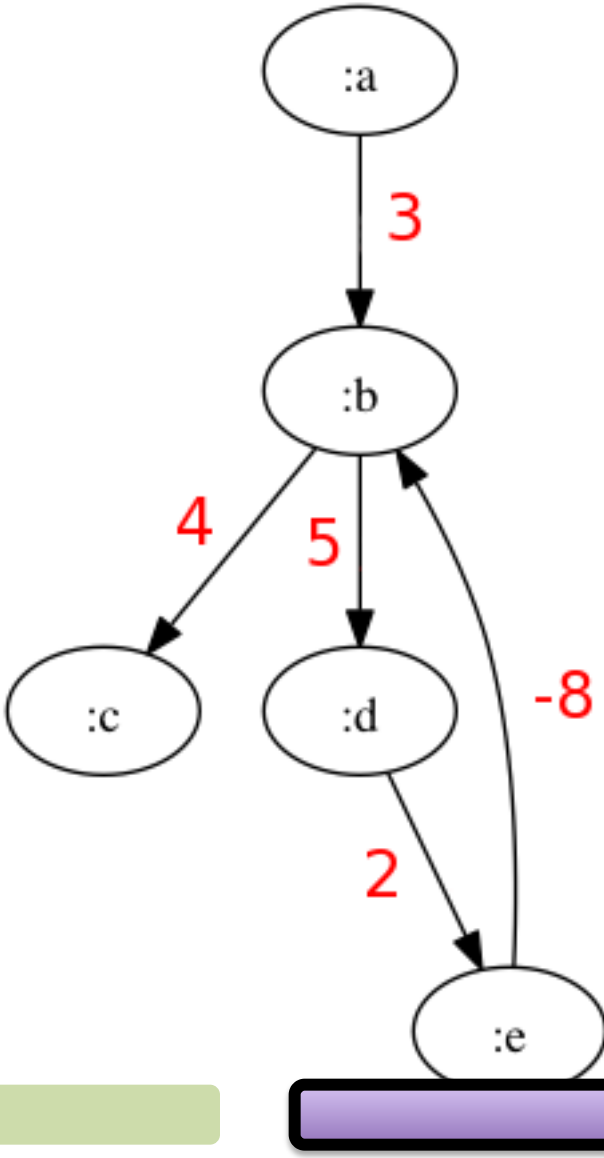
Graph API Design



Functional Graph Algorithms

Loom Overview

Functional Graph Algorithms: Bellman-Ford



Functional Graph Algorithms: Bellman-Ford

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```

CLRS *Introduction to Algorithms*

Functional Graph Algorithms: Bellman-Ford

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2       v.d = ∞
3       v.π = NIL
4  s.d = 0
```

CLRS *Introduction to Algorithms*

Functional Graph Algorithms: Bellman-Ford

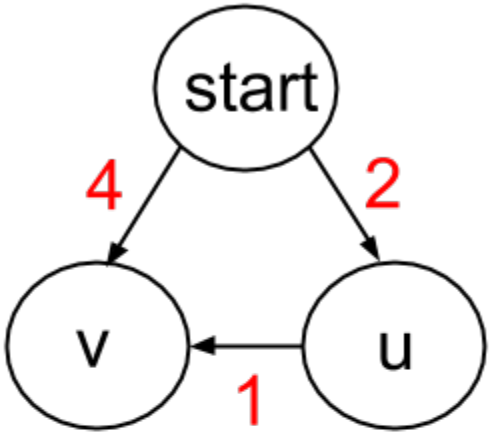
```
(defn- init-estimates
  [graph start]
  (let [nodes (disj (nodes graph) start)
        costs (interleave nodes(repeat ∞))
        paths (interleave nodes (repeat nil))]
    [(apply assoc {start 0} costs)
     (apply assoc {start nil} paths)]))
```

INITIALIZE-SINGLE-SOURCE(G, s)

1 **for** each vertex $v \in G.V$
2 $\quad v.d = \infty$
3 $\quad v.\pi = \text{NIL}$
4 $s.d = 0$

Functional Graph Algorithms: Bellman-Ford

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```




```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2       v.d = u.d + w(u, v)
3       v.π = u
```

Functional Graph Algorithms: Bellman-Ford

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2       v.d = u.d + w(u, v)
3       v.π = u
```

Functional Graph Algorithms: Bellman-Ford

```
(defn- can-relax-edge?
  [[u v :as edge] edge-cost costs]
  (let [vd (get costs v)
        ud (get costs u)
        sum (+ ud edge-cost)]
    (> vd sum)))
```

RELAX(u, v, w)

1 **if** $v.d > u.d + w(u, v)$

2 $\quad v.d = u.d + w(u, v)$

3 $\quad v.\pi = u$

Functional Graph Algorithms: Bellman-Ford

```
(defn- relax-edge
  [[u v :as edge]
   uvcost
   [costs paths :as estimates]]
  (let [ud (get costs u)
        sum (+ ud uvcost)]
    (if (can-relax-edge? edge uvcost costs)
      [(assoc costs v sum)
       (assoc paths v u)]
      estimates)))
```

RELAX(u, v, w)

1 **if** $v.d > u.d + w(u, v)$

2 $\quad v.d = u.d + w(u, v)$

3 $\quad v.\pi = u$

Functional Graph Algorithms: Bellman-Ford

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

Functional Graph Algorithms: Bellman-Ford

```
(defn- relax-edges
  [g start estimates]
  (reduce (fn [estimates [u v :as edge]]
            (relax-edge
              edge
              (weight g u v)
              estimates))
          estimates
          (edges g)))
```

BELLMAN-FORD(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 **for** $i = 1$ **to** $|G.V| - 1$
3 **for each** edge $(u, v) \in G.E$
4 RELAX(u, v, w)
5 **for each** edge $(u, v) \in G.E$
6 **if** $v.d > u.d + w(u, v)$
7 **return** FALSE
8 **return** TRUE

Functional Graph Algorithms: Bellman-Ford

```
(defn bellman-ford
  [g start]
  (let [initial-estimates (init-estimates g start)
        ;relax-edges is calculated for all edges V-1 times
        [costs paths] (reduce (fn [estimates _] (relax-edges g start estimates))
                              initial-estimates
                              (-> g nodes count dec range))
        edges (edges g)]
    (if (some (fn [[u v :as edge]] (can-relax-edge? edge (wt g u v) costs))
              edges)
      false
      [costs
       (->> (keys paths)
            ;remove vertices that are unreachable from source
            (remove #(= Double/POSITIVE_INFINITY (get costs %)))
            (reduce
              (fn [final-paths v]
                (assoc final-paths v
                   ; follows the parent pointers
                   ; to construct path from source to node v
                   (loop [node v path ()]
                         (if node
                           (recur (get paths node) (cons node path))
                           path))))
              {}))])))
```


- Function composition
- Functions operate on helper data structures
- Higher order functions
- ***reduce*** to iterate over elements

Functional Graph Algorithms: Bellman-Ford

```
(defn bellman-ford
 [g start]
  (let [initial-estimates (init-estimates g start)
        ;relax-edges is calculated for all edges V-1 times
        [costs paths]
        (reduce
         (fn [estimates _] (relax-edges g start estimates))
         initial-estimates
        (-> g nodes count dec range))
         edges (edges g)]
```

BELLMAN-FORD(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 **for** $i = 1$ **to** $|G.V| - 1$
3 **for** each edge $(u, v) \in G.E$
4 RELAX(u, v, w)
5 **for** each edge $(u, v) \in G.E$
6 **if** $v.d > u.d + w(u, v)$
7 **return** FALSE
8 **return** TRUE

Functional Graph Algorithms: Bellman-Ford

```
(if (some (fn [[u v :as edge]]
              (can-relax-edge? edge
                               (weight g u v)
                               costs))
          edges)
    false
 ;;; return paths))
```

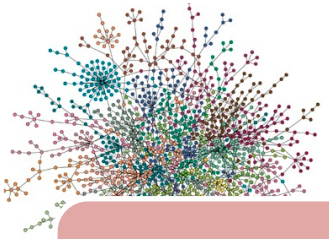
BELLMAN-FORD(G, w, s)

```
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

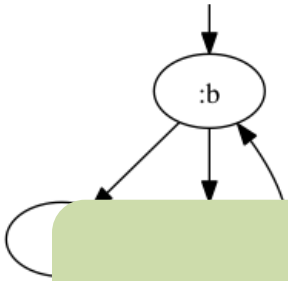
Functional Graph Algorithms

- Everything is a function
- Functions on internal representation
- Every function returns new representation

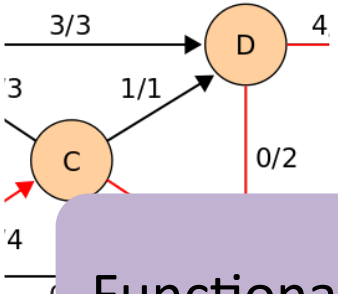
In this Talk



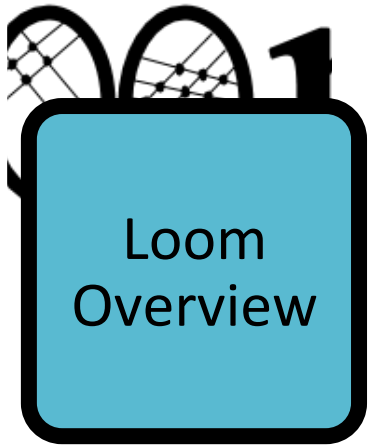
A Moment of Graph Theory



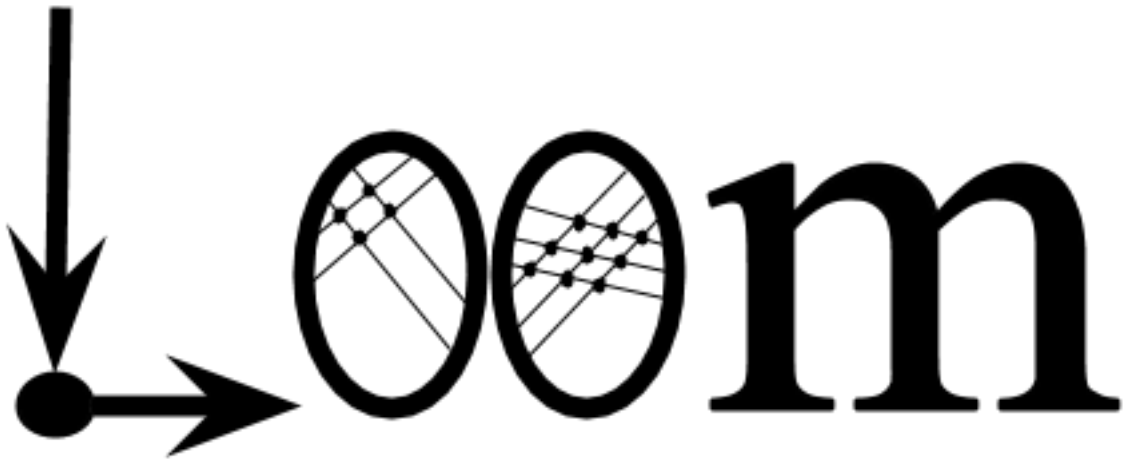
Graph API Design



Functional Graph Algorithms



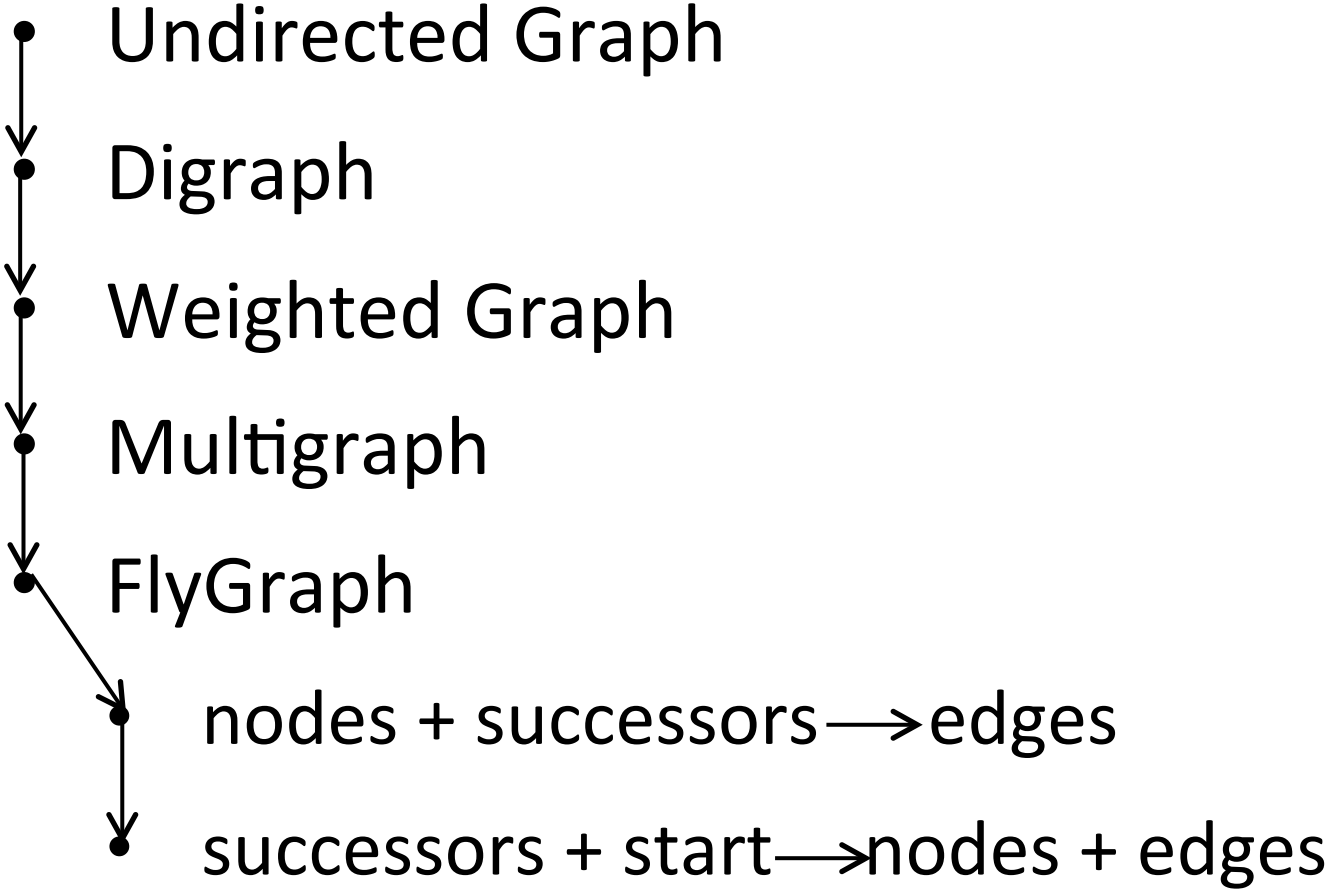
Loom Overview



Graph Algorithms + Visualization

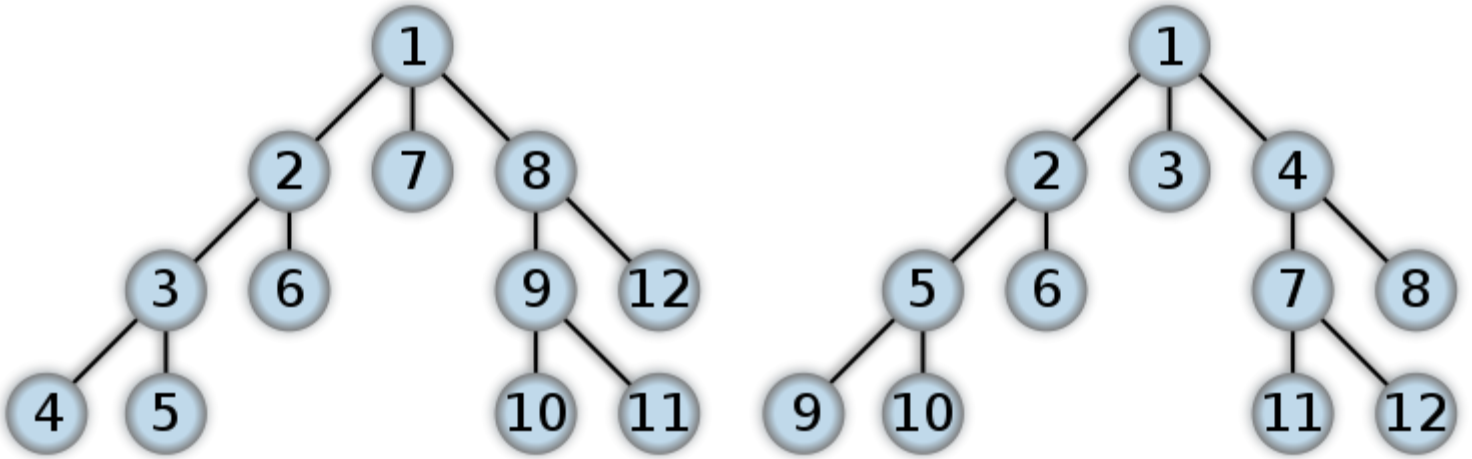
github.com/aysylu/loom

Loom Overview: Supported Graphs



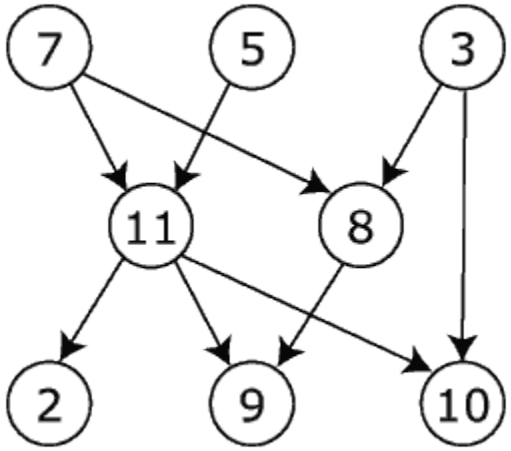
Loom Overview: Graph Algorithms

- DFS/BFS (+ bidirectional)



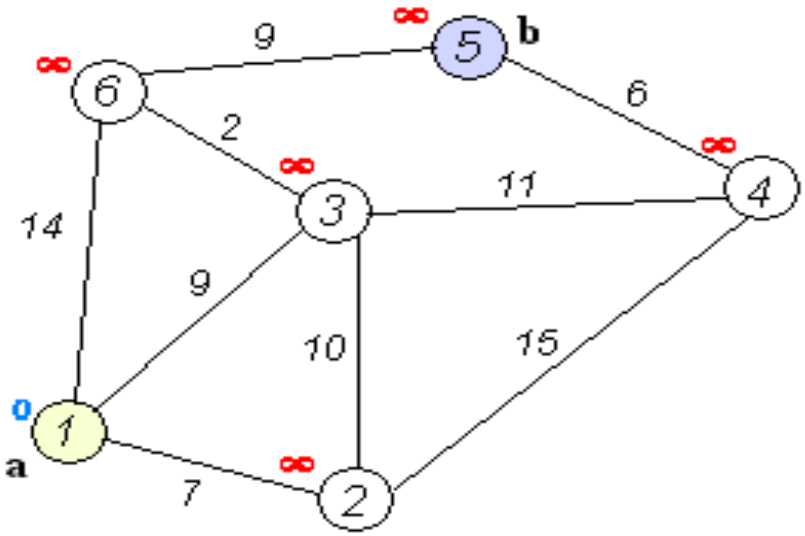
Loom Overview: Graph Algorithms

- DFS/BFS (+ bidirectional)
- Topological Sort



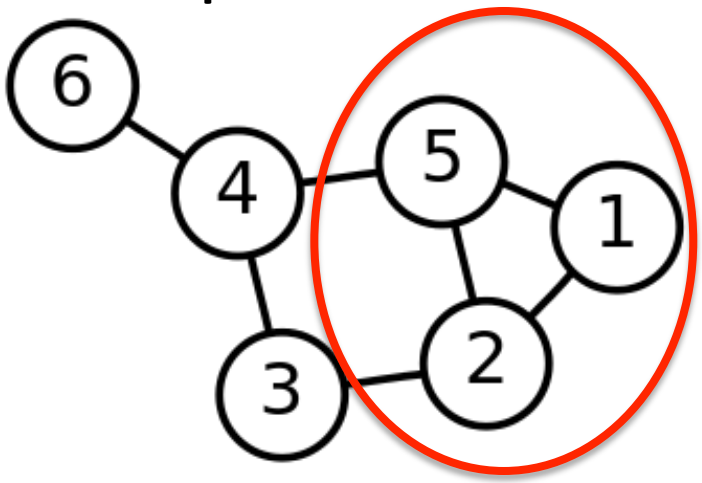
Loom Overview: Graph Algorithms

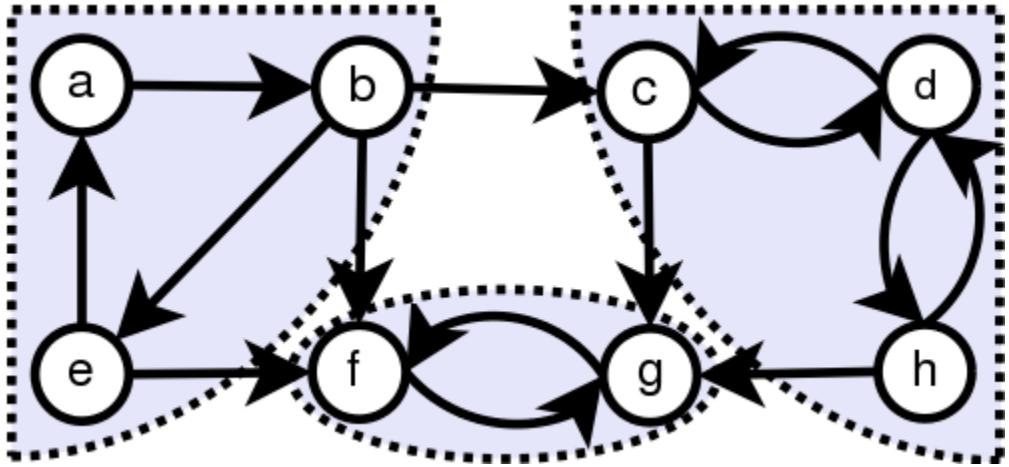
- DFS/BFS (+ bidirectional)
- Topological Sort
- Shortest Path (Dijkstra, Bellman-Ford, A*, Johnson's)



Loom Overview: Graph Algorithms

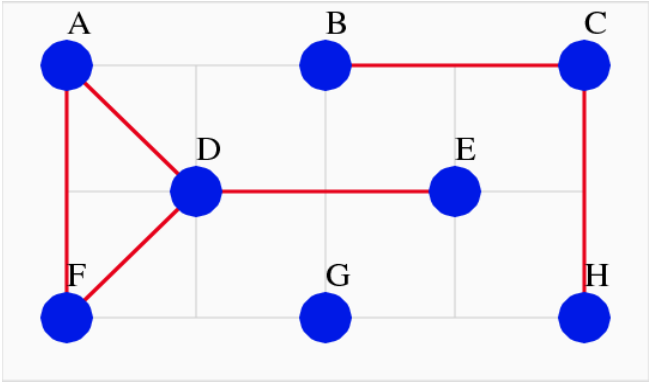
- DFS/BFS (+ bidirectional)
- Topological Sort
- Shortest Path (Dijkstra, Bellman-Ford, A*, Johnson's)
- Cliques & SCC (Bron-Kerbosch, Kosaraju)





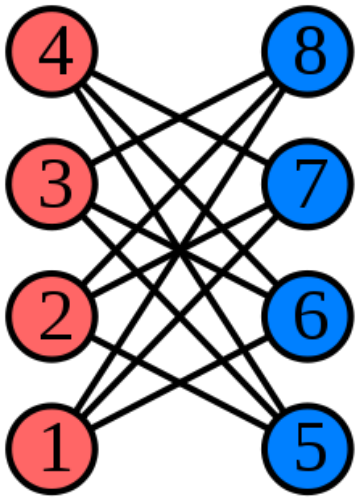
Loom Overview: Graph Algorithms

- DFS/BFS (+ bidirectional)
- Topological Sort
- Shortest Path (Dijkstra, Bellman-Ford, A*, Johnson's)
- Cliques & SCC (Bron-Kerbosch, Kosaraju)
- Density (edges/nodes)
- Loner Nodes



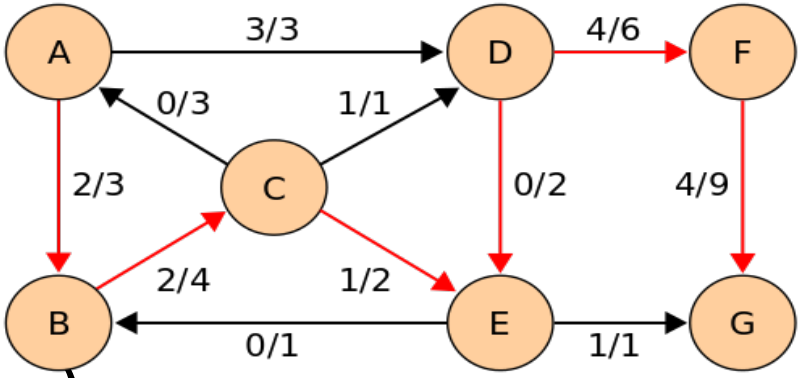
Loom Overview: Graph Algorithms

- DFS/BFS (+ bidirectional)
- Topological Sort
- Shortest Path (Dijkstra, Bellman-Ford, A*, Johnson's)
- Cliques & SCC (Bron-Kerbosch, Kosaraju)
- Density (edges/nodes)
- Loner Nodes
- Greedy & Two Coloring



Loom Overview: Graph Algorithms

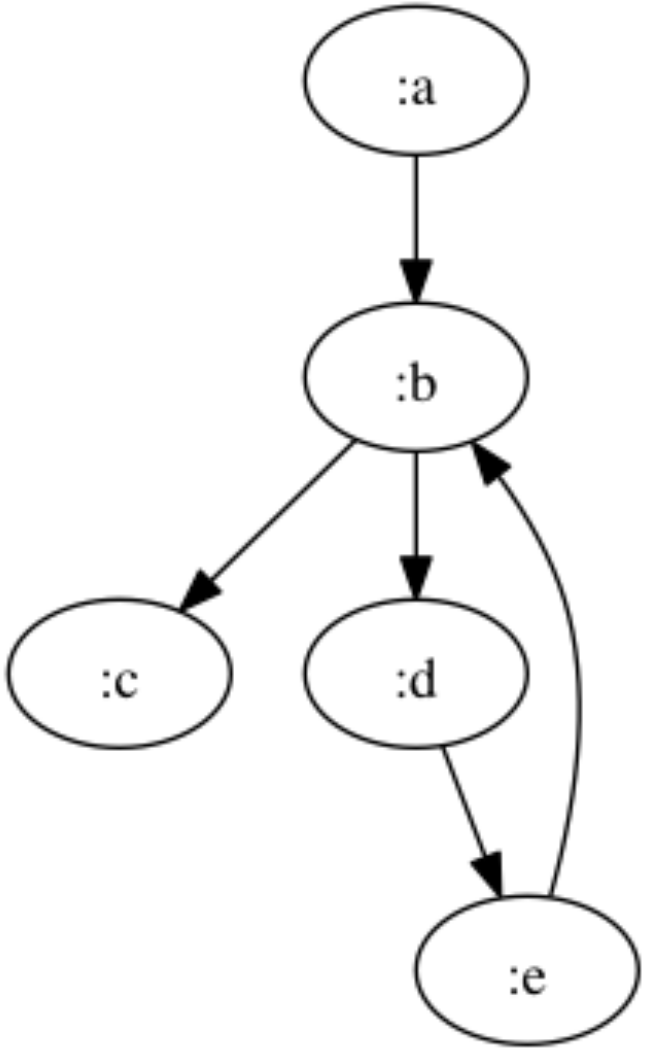
- DFS/BFS (+ bidirectional)
- Topological Sort
- Shortest Path (Dijkstra, Bellman-Ford, A*, Johnson's)
- Cliques & SCC (Bron-Kerbosch, Kosaraju)
- Density (edges/nodes)
- Loner Nodes
- Greedy & Two Coloring
- Max-Flow (Edmonds-Karp)

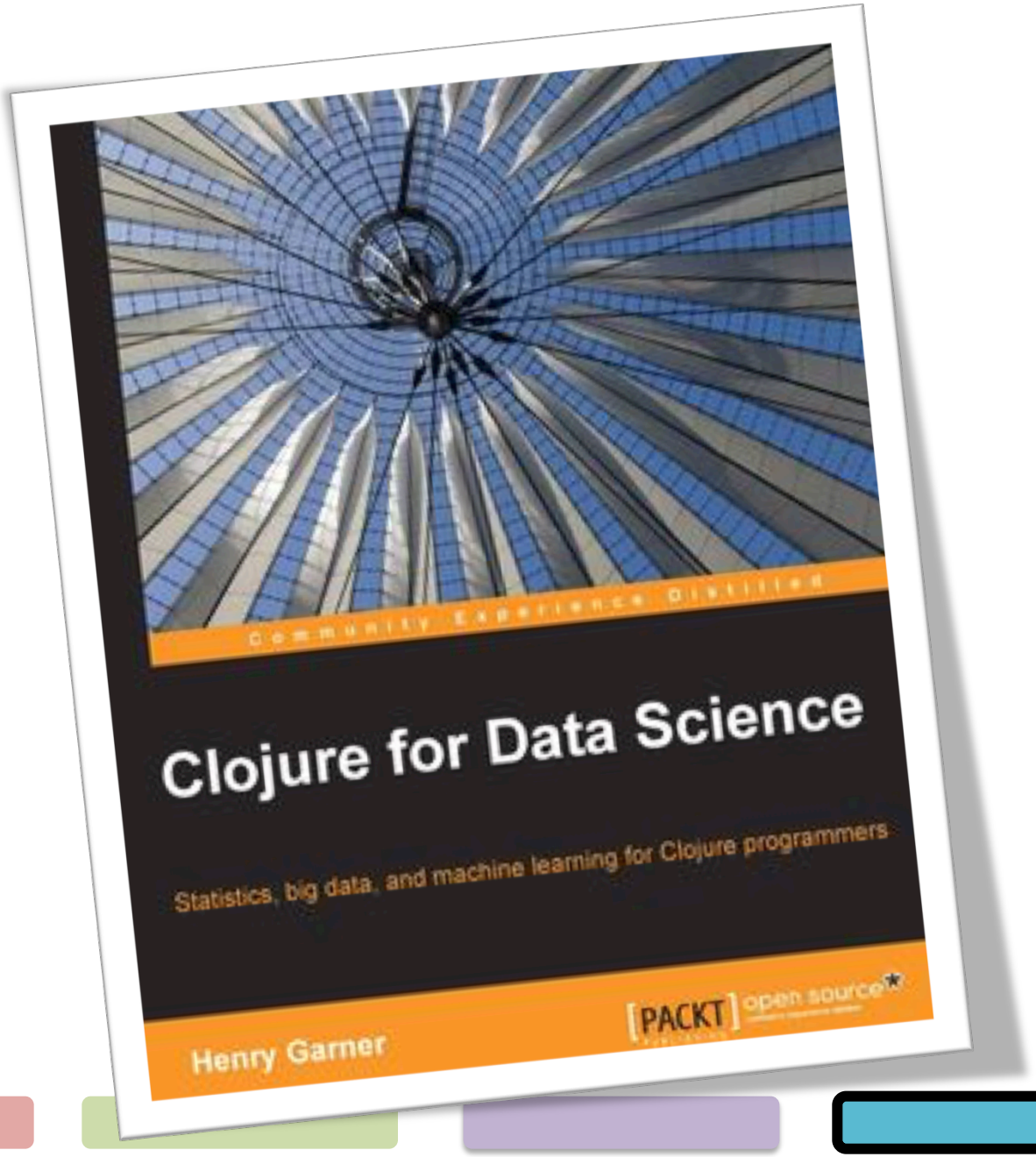


Functional Graph Algorithms: loom.alg-generic

- No knowledge of graph representation
- Requires only successors
  + start for DFS/BFS, topological sort, Dijkstra
  + end for BFS path

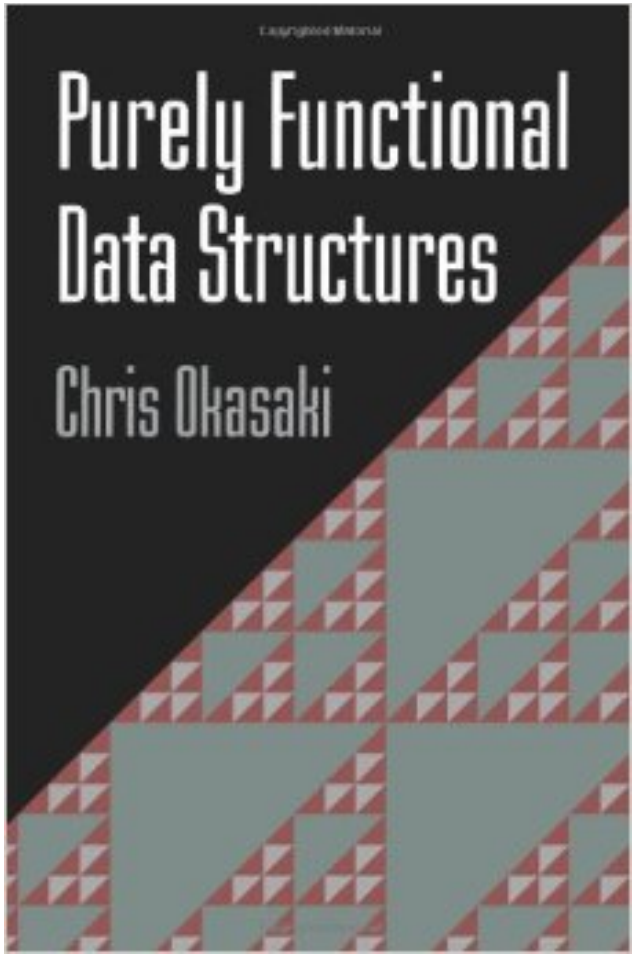
Loom Overview: Visualizing Graphs




Community Experience Distilled
Clojure for Data Science
Statistics, big data, and machine learning for Clojure programmers
Henry Garner
[PACKT] open source

WRAP UP: FUNCTIONAL GRAPHS

Functional Data Structures



Takeaway #1

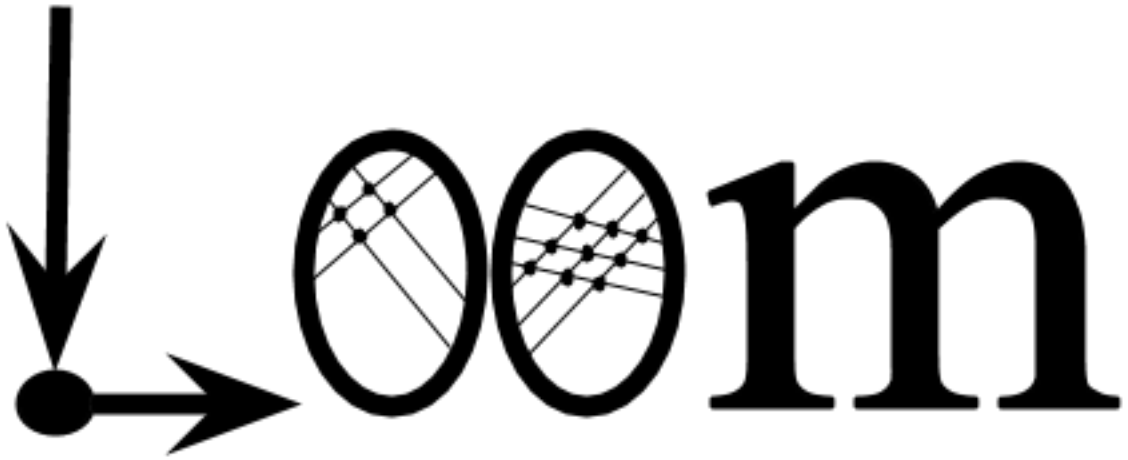
DATA & FUNCTIONS SEPARATED

Takeaway #2

FLEXIBLE REPRESENTATION

Takeaway #3

IMMUTABLE GRAPHS



Takeaway #4

USE LOOM

Later for reddit

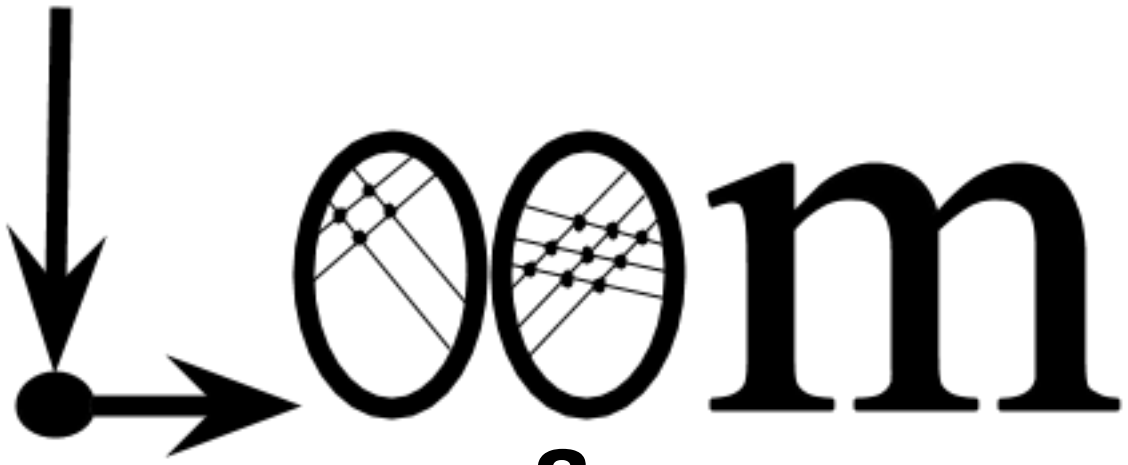
Loom is super great

My takeaway from this is that, if you have a problem that may require a graph, loom is super easy to use and effective. You build a graph, then you query the graph, and that's all there is to it. If you have such a problem, I encourage you to try to use loom to solve it. And let me know how that goes, too!

https://adambard.com/blog/building-a-suggestion-engine-for-redditlater/

Thanks to Loom contributors

- John Szakmeister
- Robert Lachlan
- Stephen Kockentiedt
- Alex Engelberg
- Kevin Downey
- Aaron Brooks
- Mark Engelberg
- Daniel Shapero
- Daniel Gregoire
- Tom Hancock
- Horst Duchêne
- Ashton Kemerling
- Justin Kramer
- Jony Hudson
- Guru Devanla
- Joshua Davey
- Sung Pae
- François Rey
- Georgi Shopov
- Matt Revelle
- Zack Maril
- Daniel Compton
- Reid D McKenzie
- ZTO
- Aysylu Greenberg
- Paul Snyder



&

Functional Graphs in Clojure

Aysylu Greenberg

@aysylu22

Google