ConnectIt: A Framework for Static and Incremental Parallel Graph Connectivity Algorithms

Laxman Dhulipala
MIT (Postdoc)
https://ldhulipala.github.io/

Based on joint work with

Changwan Hong and Julian Shun (VLDB'21)

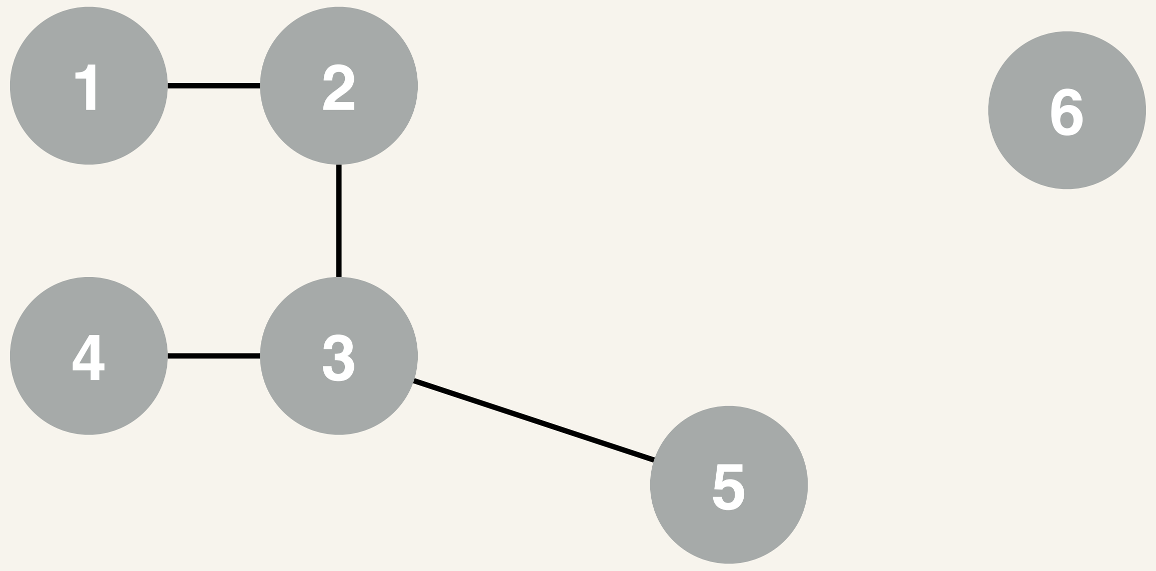
Connected Components

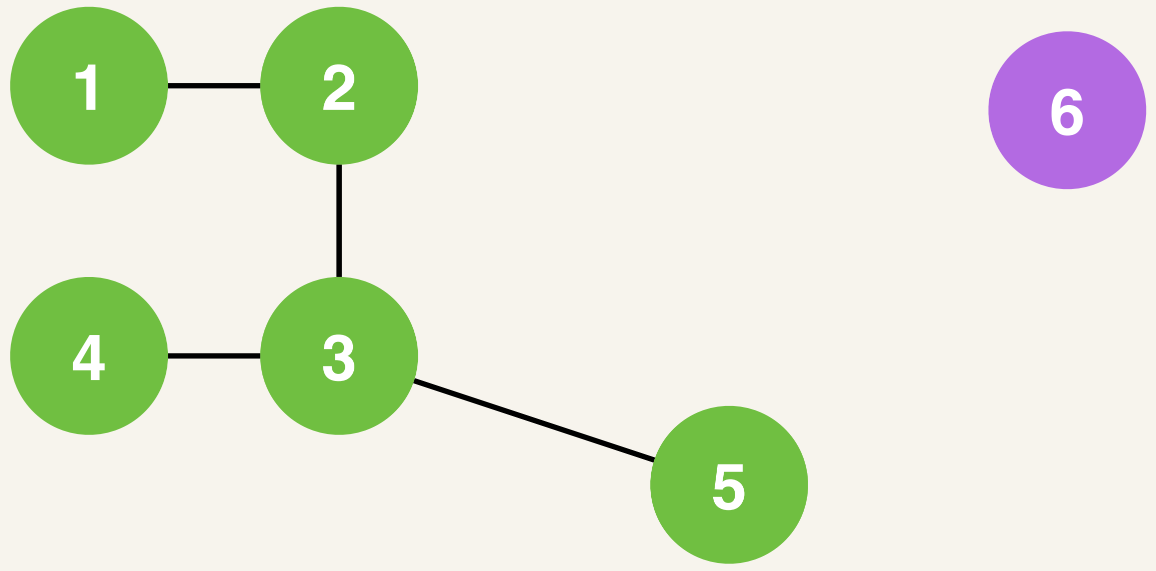
- Given a graph $G(V, E)$

  $n = |V| = \#$ vertices

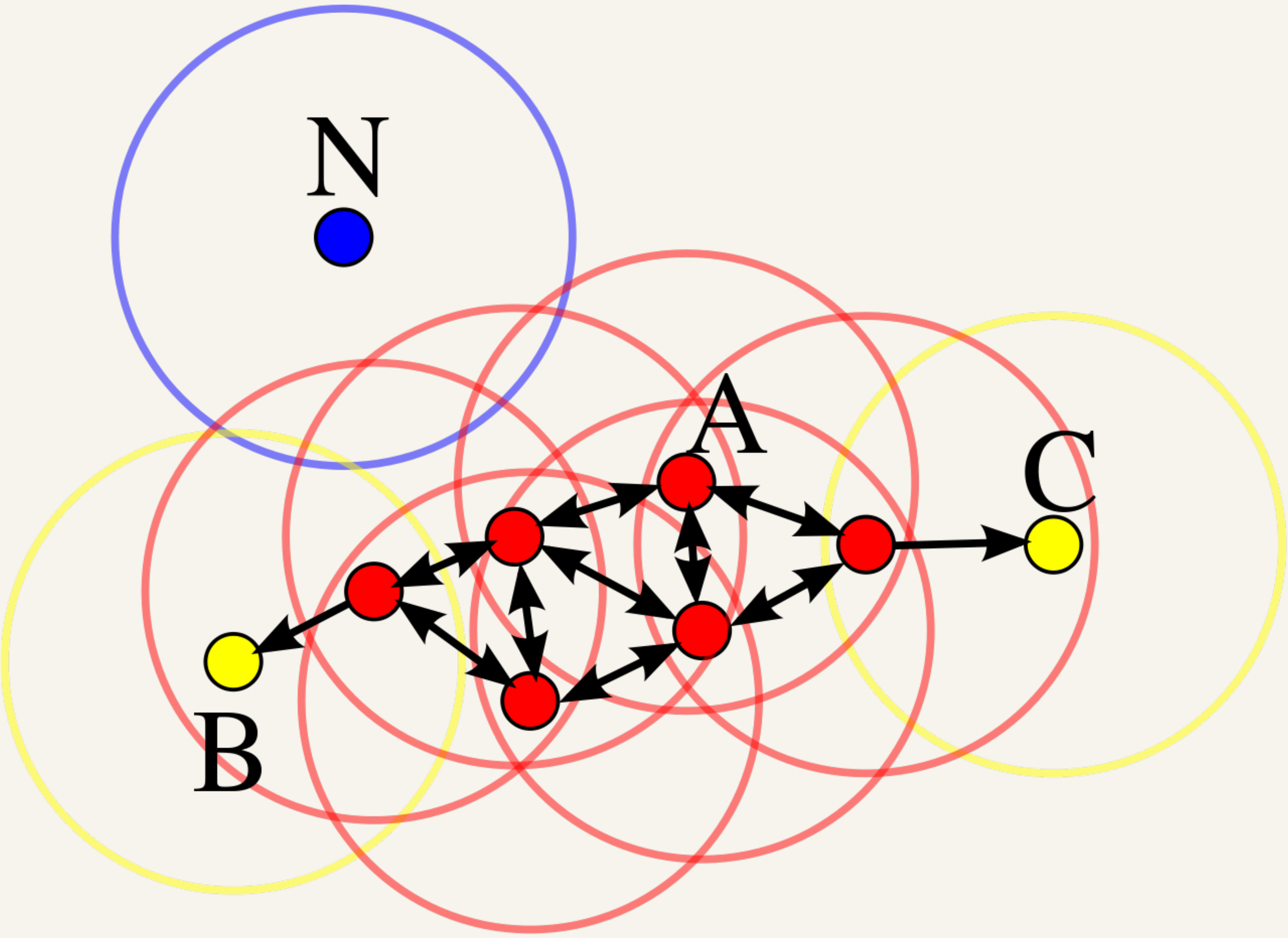
  $m = |E| = \#$ edges

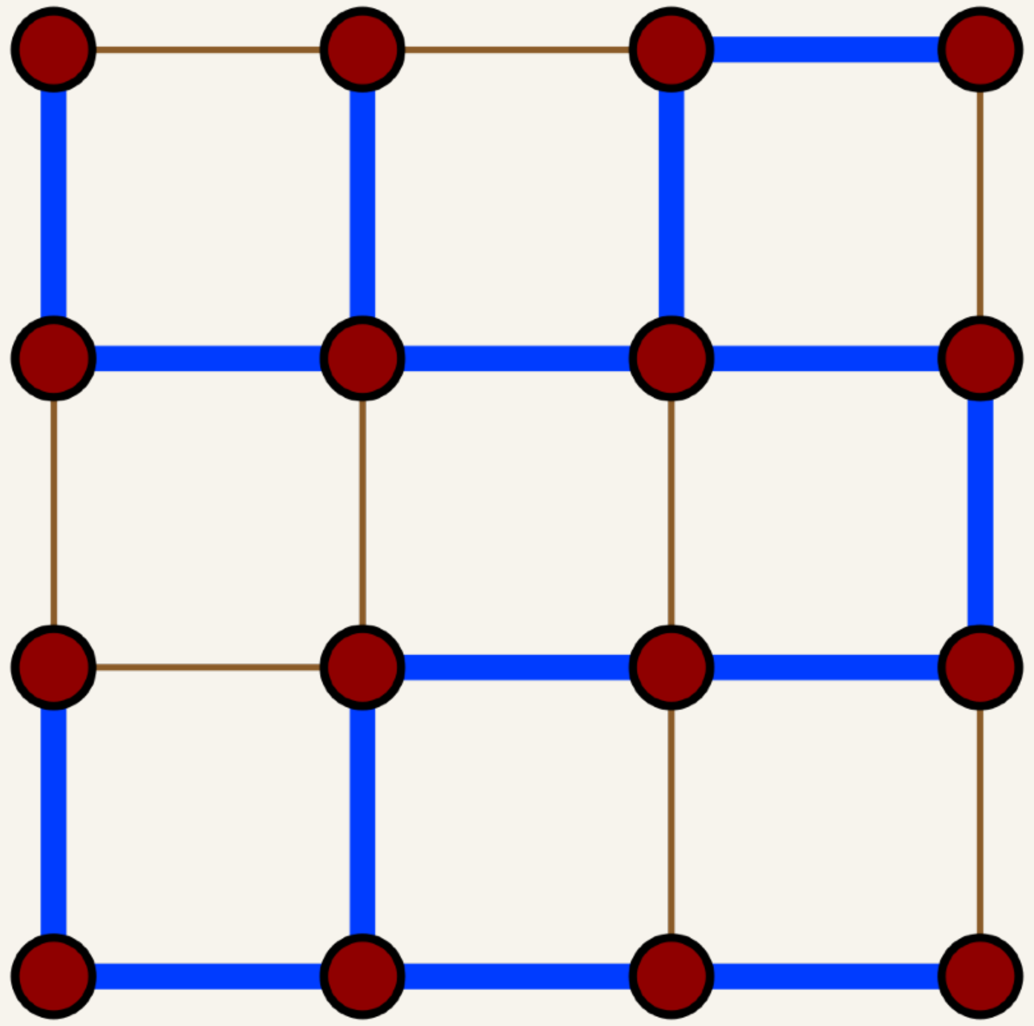
  Assign vertices labels $L(v)$ s.t. $L(u) = L(v)$ iff there is a path from u to v in G





Applications of Connected Components





Clustering

- DBSCAN
- k-Core Hierarchy
- Affinity Clustering
- …

Other Connectivity Problems

- Spanning Forest
- Biconnectivity
- Approximate Minimum Spanning Forest

Image Source: Wikipedia

Sequential Connectivity Algorithms

❖ Run Breadth-First Search or Depth-First Search:

```
labels = [-1, ..., -1]   # initialized to a null value
for i in [0, |V|):
  if labels[i] == -1:
    BFS(G, i)            # assign label i to visited vertices
return labels
```

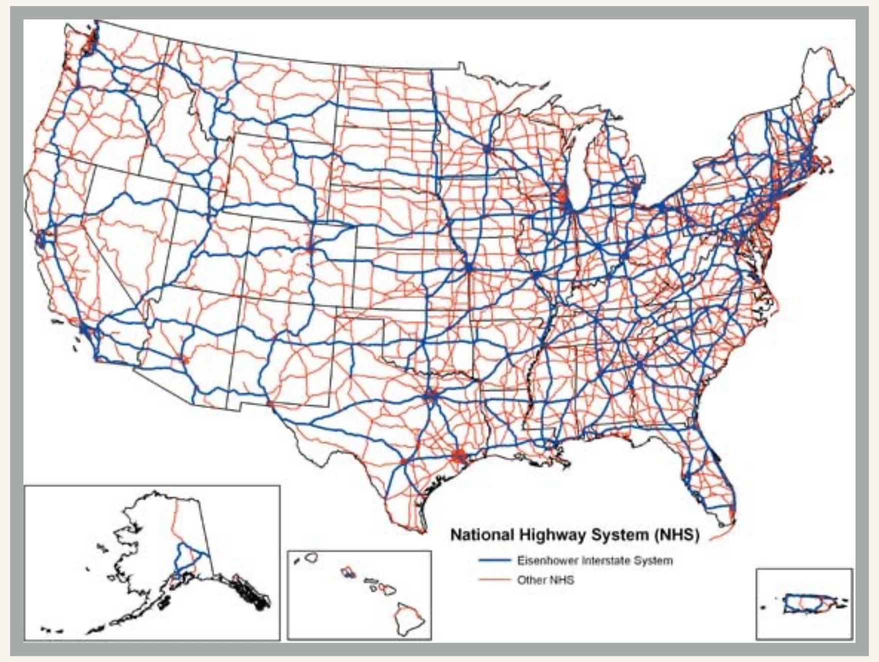
❖ Algorithms run in $O(n + m)$ time

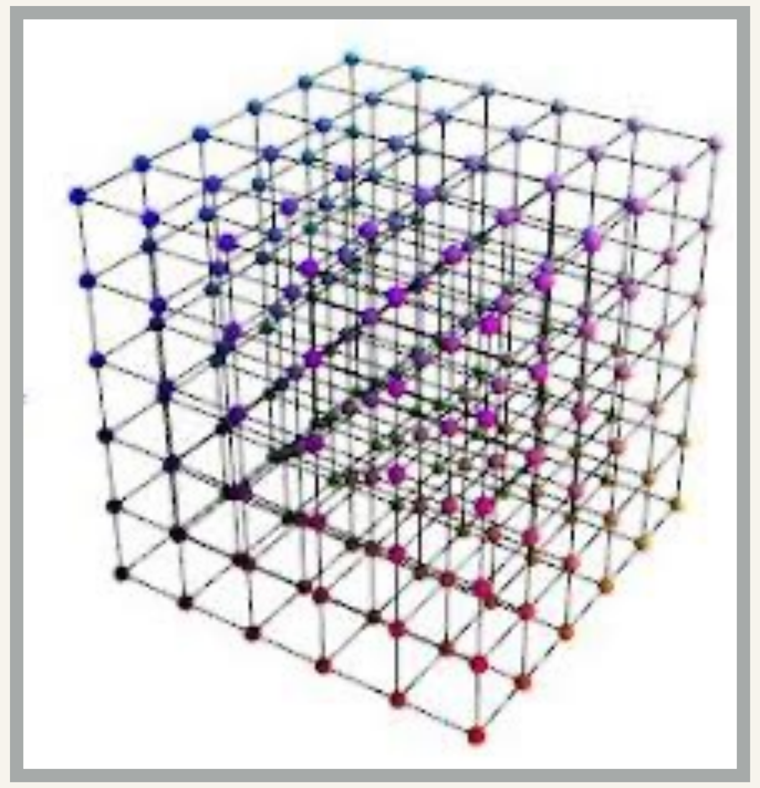
Parallel BFS for Connectivity

```
labels = [-1, ..., -1]  # initialized to a null value
for i in [0, |V|):
  if labels[i] == -1:
    ParallelBFS(G, i)   # assign label i to visited vertices
return labels
```

- ❖ Real-world graphs can have high diameter (e.g. road networks / meshes)
- ❖ Graph could also have many components

$O(m + n)$ work, $O(n)$ depth

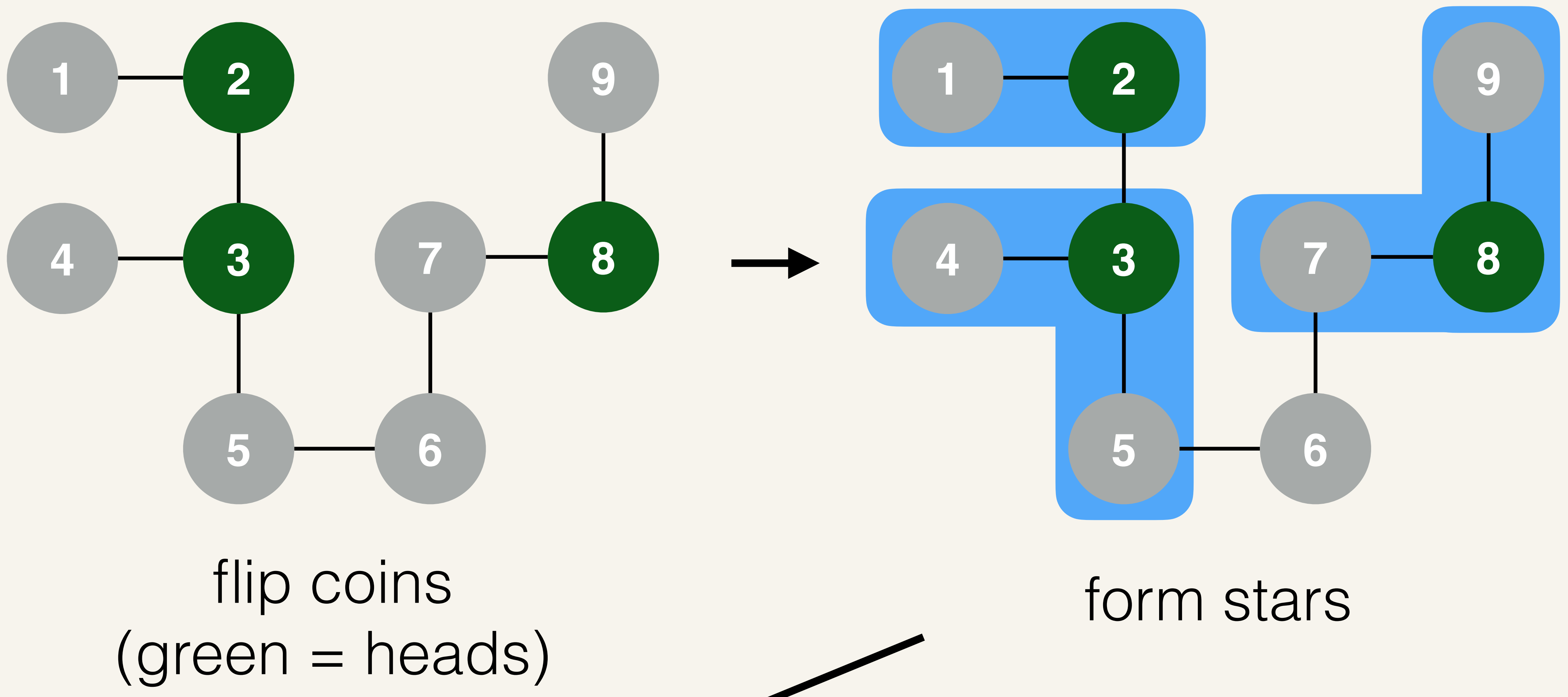


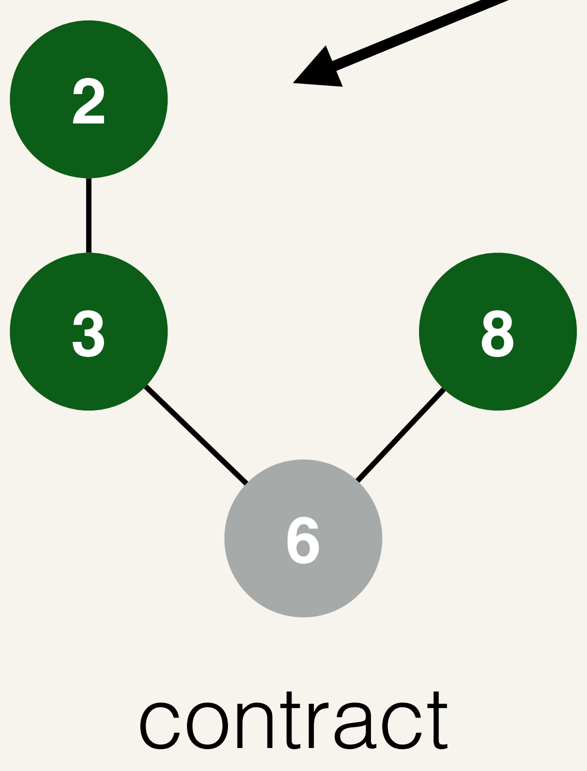


Are there low-work, polylog(n) depth connectivity algorithms?

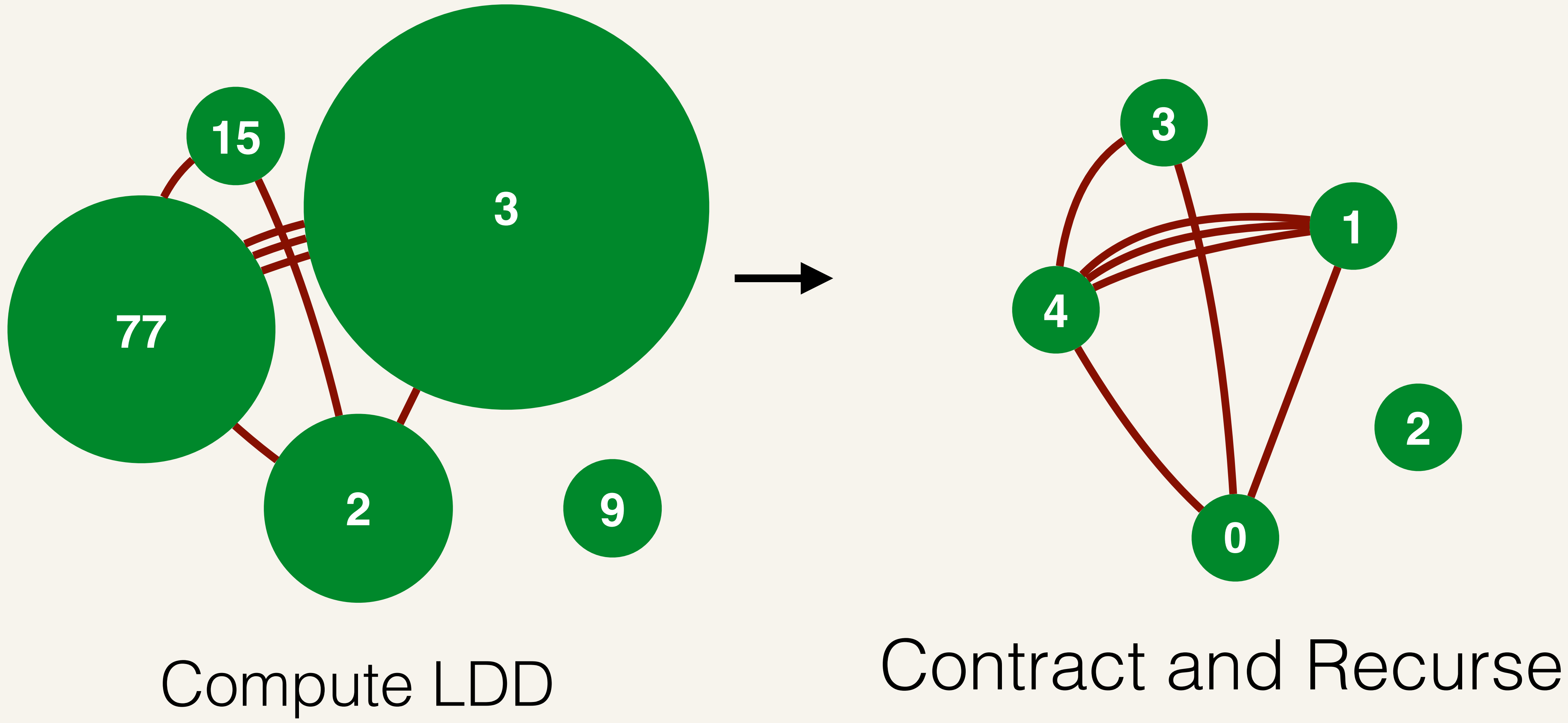
Parallel Connectivity Algorithms

Random-Mate Algorithms

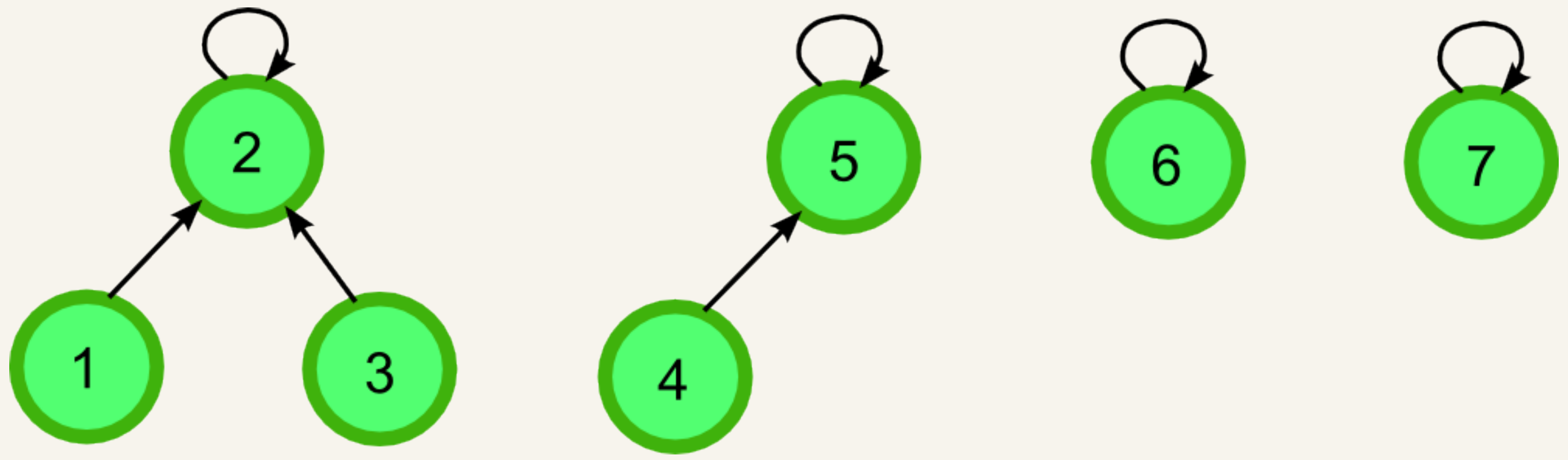




Work-Efficient Algorithms



Concurrent Union-Find



Dozens of papers on different approaches to parallel connectivity written over the past few decades!

ConnectIt: A Framework for Static and Incremental Parallel Graph Connectivity Algorithms [DHS'21]

Goal:
Explore the space of optimizations for parallel (shared-memory) graph connectivity and find the *fastest implementation of parallel connectivity*

ConnectIt Framework

Mode

- Connectivity
- Incremental Connectivity
- Spanning Forest

Sampling Options

- k-Out Sample
- BFS Sample
- LDD Sample

Finish Options

Union-Find

Jayanti-Tarjan		FindSplit
UniteRemLock	×	FindHalve
UniteRemCAS		FindCompress
…		…

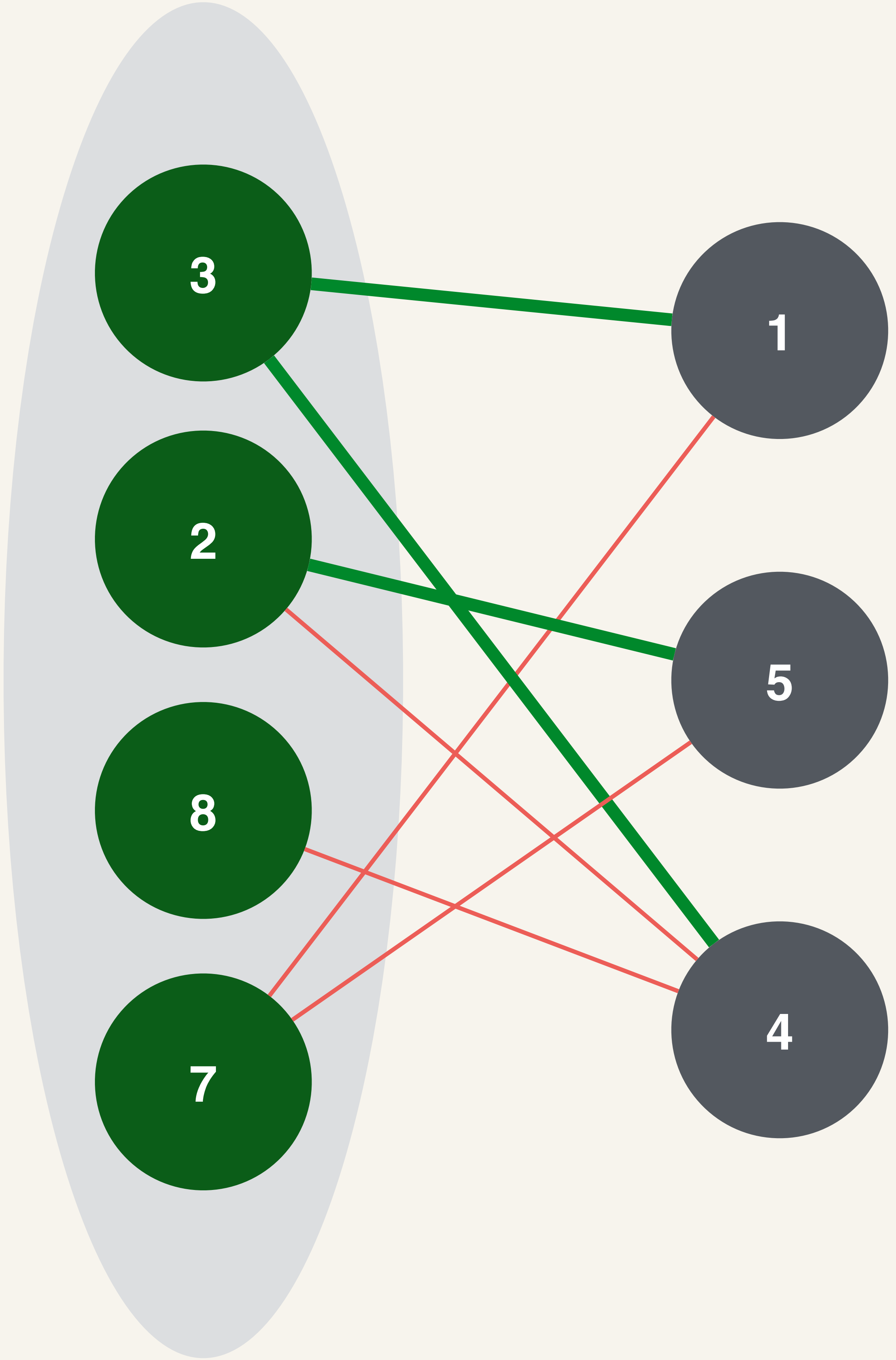
Liu-Tarjan

- Alg. P
- Alg. E
- Alg. A

Shiloach-Vishkin   Label Propagation   …

- ❖ Express several hundred different multicore implementations of connectivity, spanning forest, and incremental connectivity (most of which are new)
- ❖ Obtain *2.3x average speedup* over the fastest existing static multicore connectivity algorithms

Motivation: Direction-Optimizing BFS



Direction-optimization skips over incoming edges in dense traversals once the vertex has already been visited

```
Using direction-opt:     0.081425
Without direction-opt:   0.715358
```

(on the Twitter-Sym graph, 72 cores)

Two-Phase Execution is inspired by direction optimization. It accelerates parallel connectivity algorithms by "skipping" the traversal of certain edges

Two-Phase Execution

Sampling Phase

Compute a partial connectivity labeling while processing edges

Identify the largest component $L_{\max}$ in the partial labeling.

Finish Phase

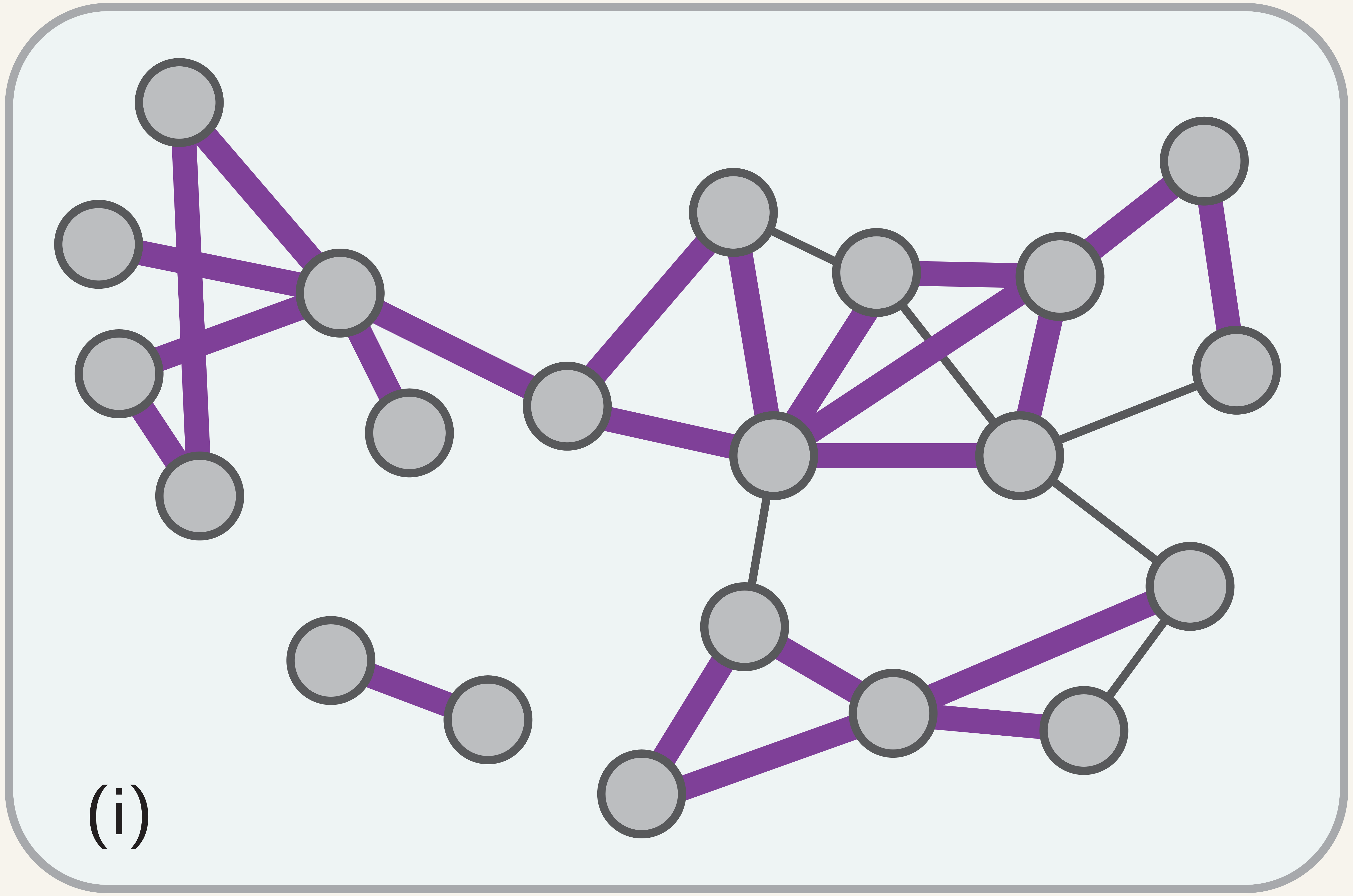
Process all vertices not in $L_{\max}$ using the given finish algorithm to compute a correct connectivity labeling.

ConnectIt: Connectivity Meta-Algorithm

```
def Connectivity(G(V,E), sample_opt, finish_opt):
  # Initialize sampling and finish algorithms
  sampling = GetSamplingAlgorithm(sample_opt)
  finish = GetFinishAlgorithm(finish_opt)

  # Initialize labels and perform sampling to
  # obtain a partial connectivity labeling.
  labels = {i -> i | i in [0, |V|)}
  labels = sampling.SampleComponents(G, labels)

  # Identify the largest (most frequent
  # component), L_max
  L_max = IdentifyFrequent(labels)

  # Compute a connectivity labeling from the partial
  # labeling using the finish algorithm.
  labels = finish.FinishComponents(G, labels, L_max)
  return labels
```

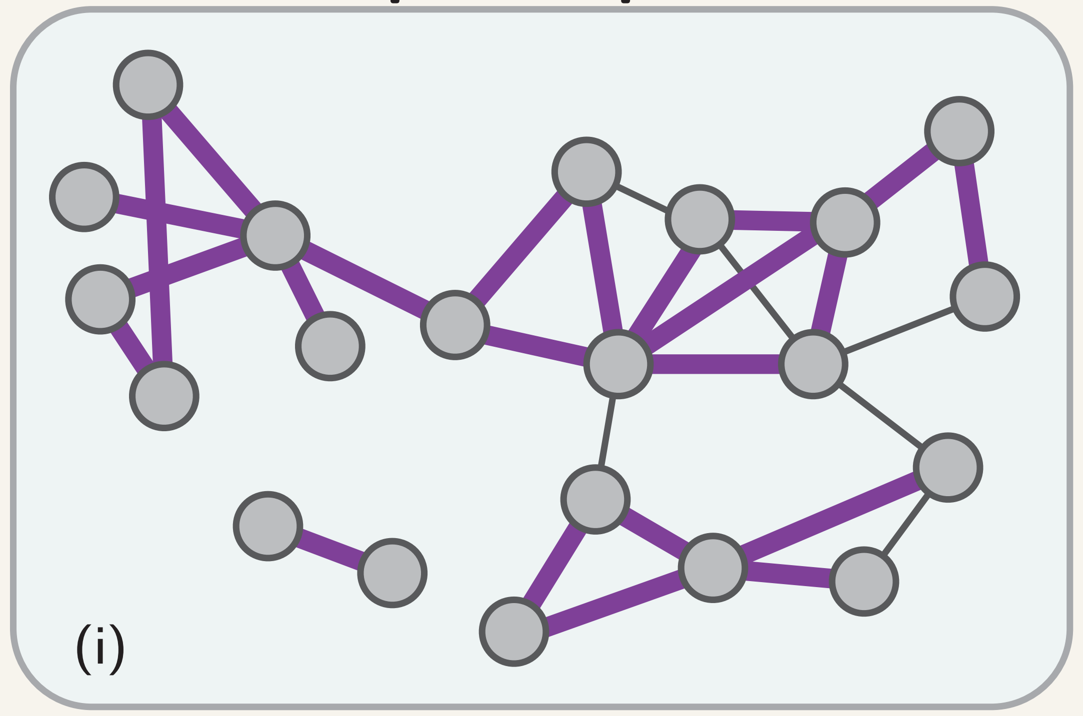
Two-Phase Execution: Example

Input Graph

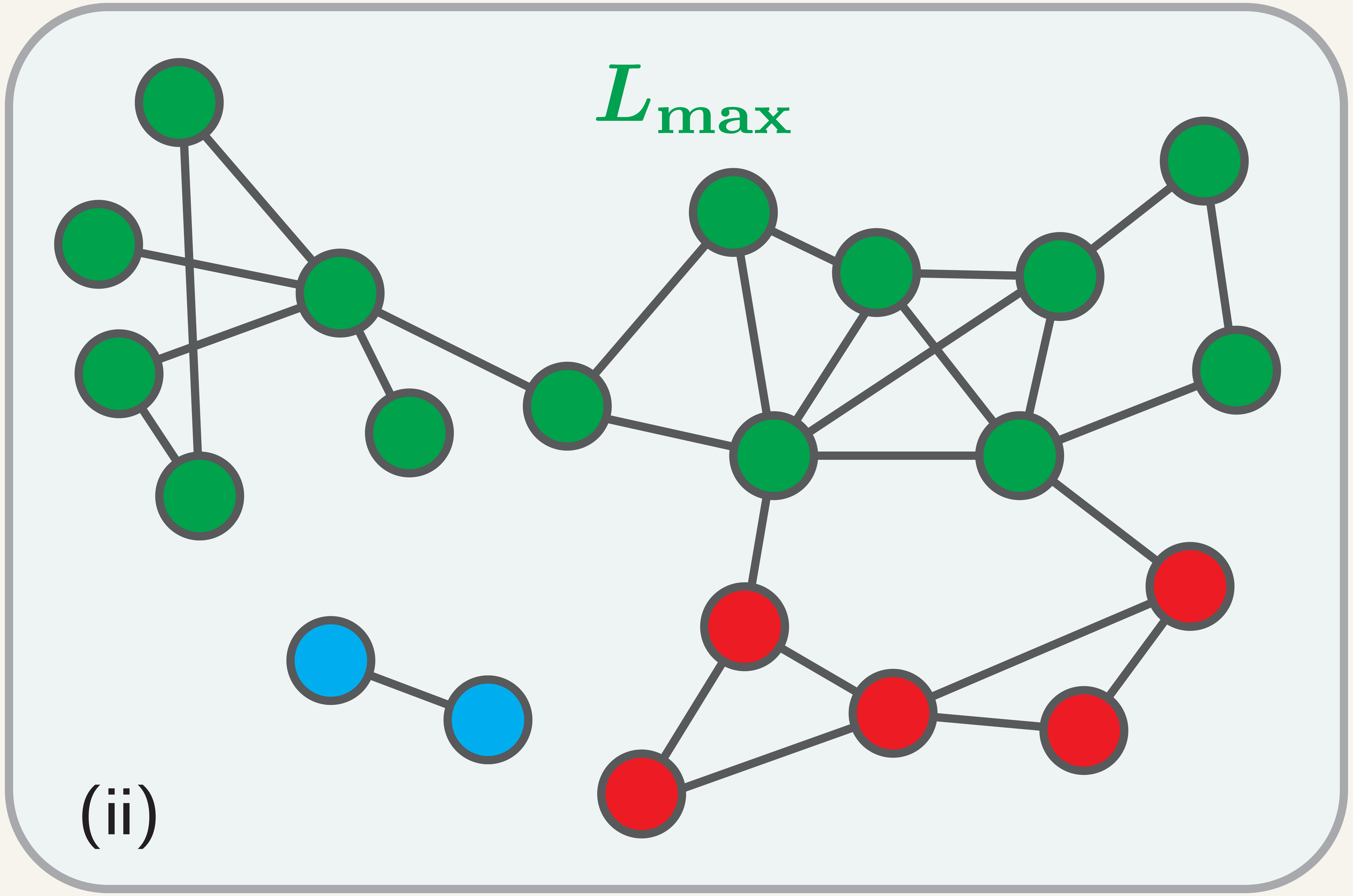


Two-Phase Execution: Example

Input Graph

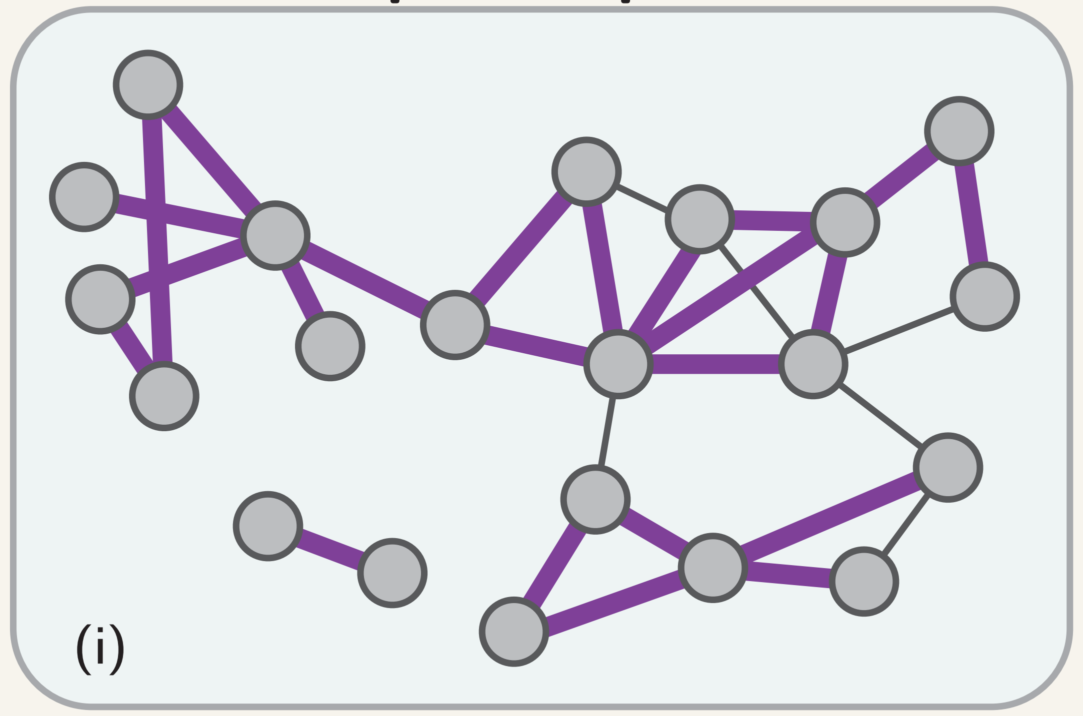


Sampled Labels

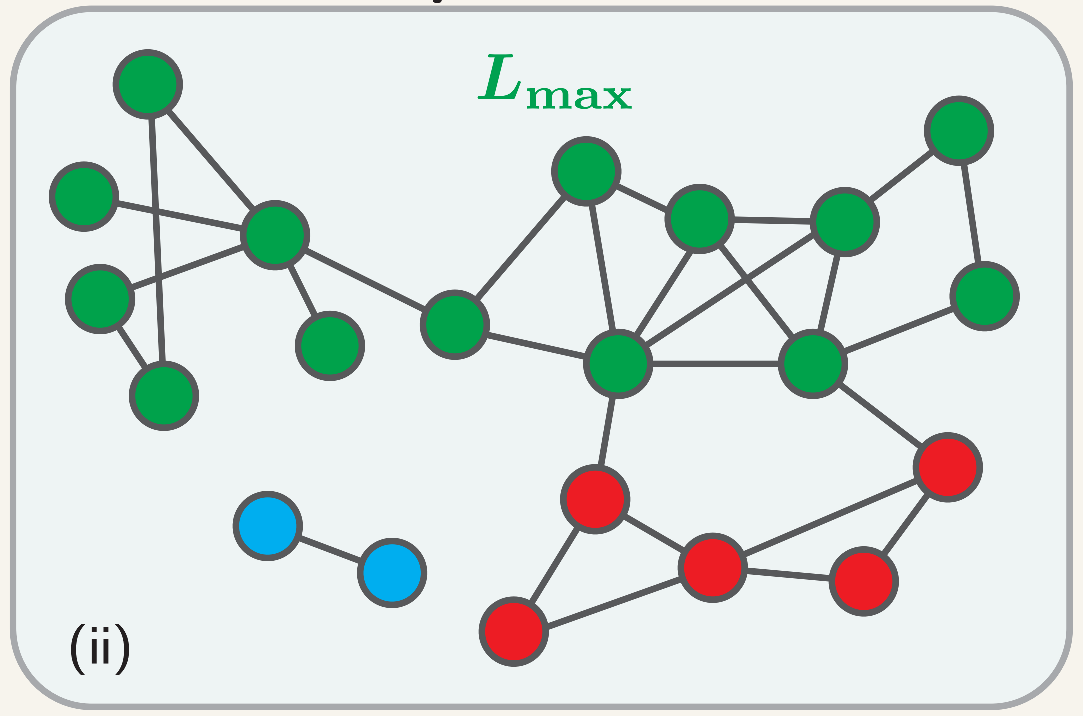


Two-Phase Execution: Example

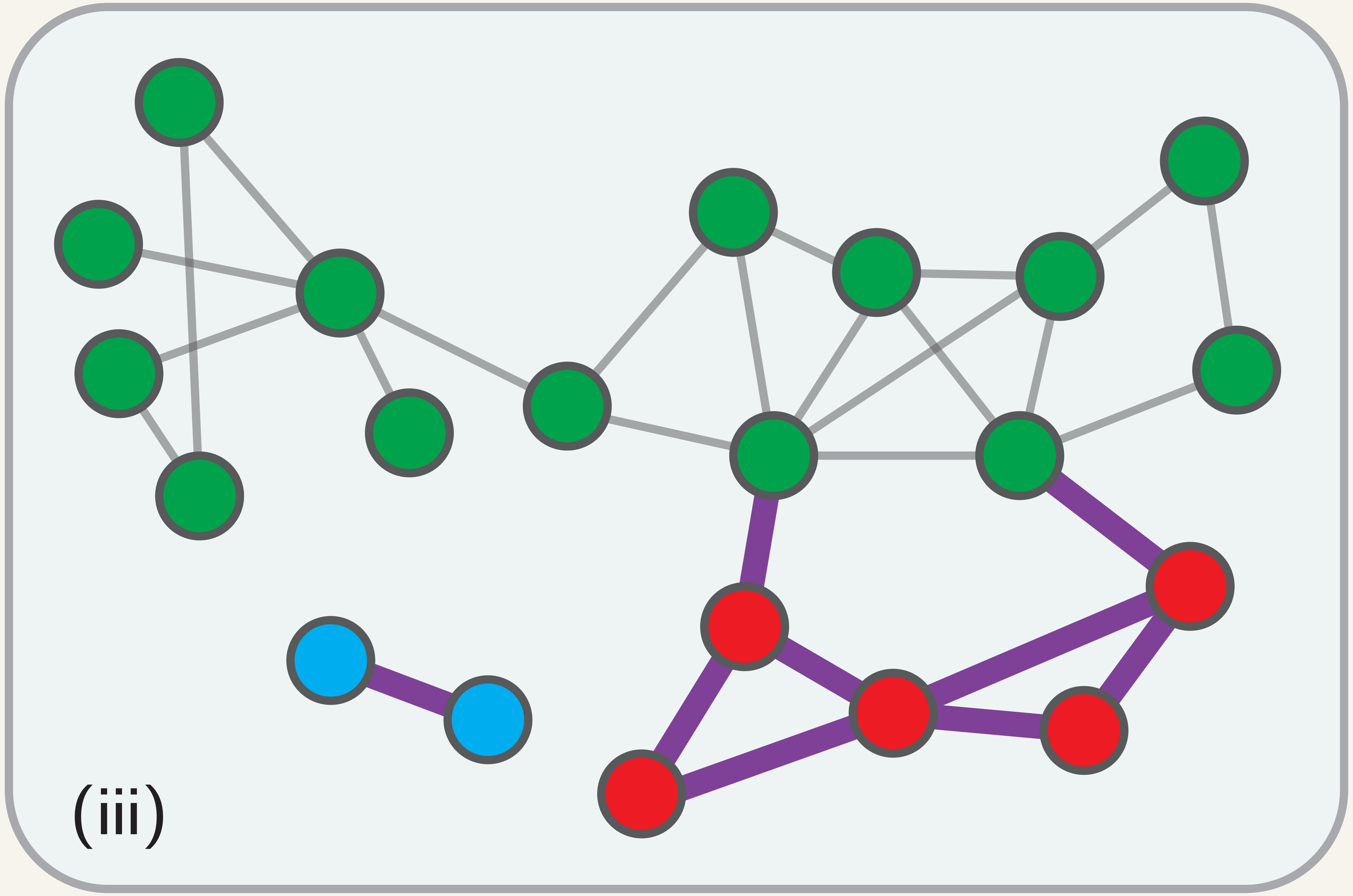
Input Graph



Sampled Labels

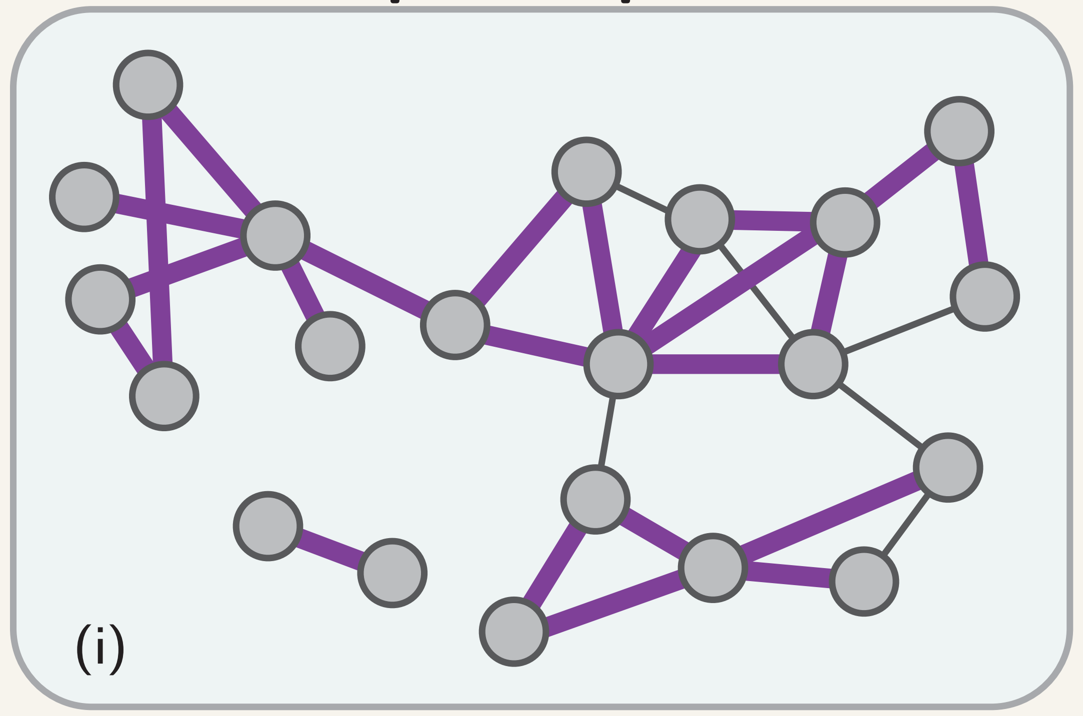


Finish Step on $v \notin L_{\max}$

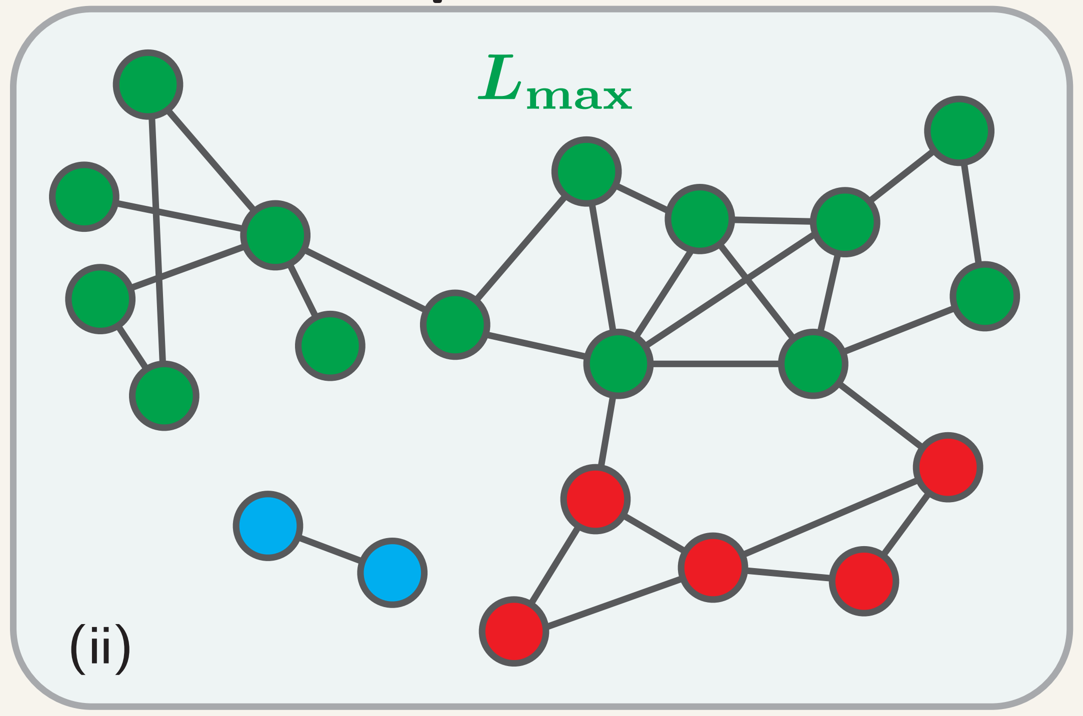


Two-Phase Execution: Example

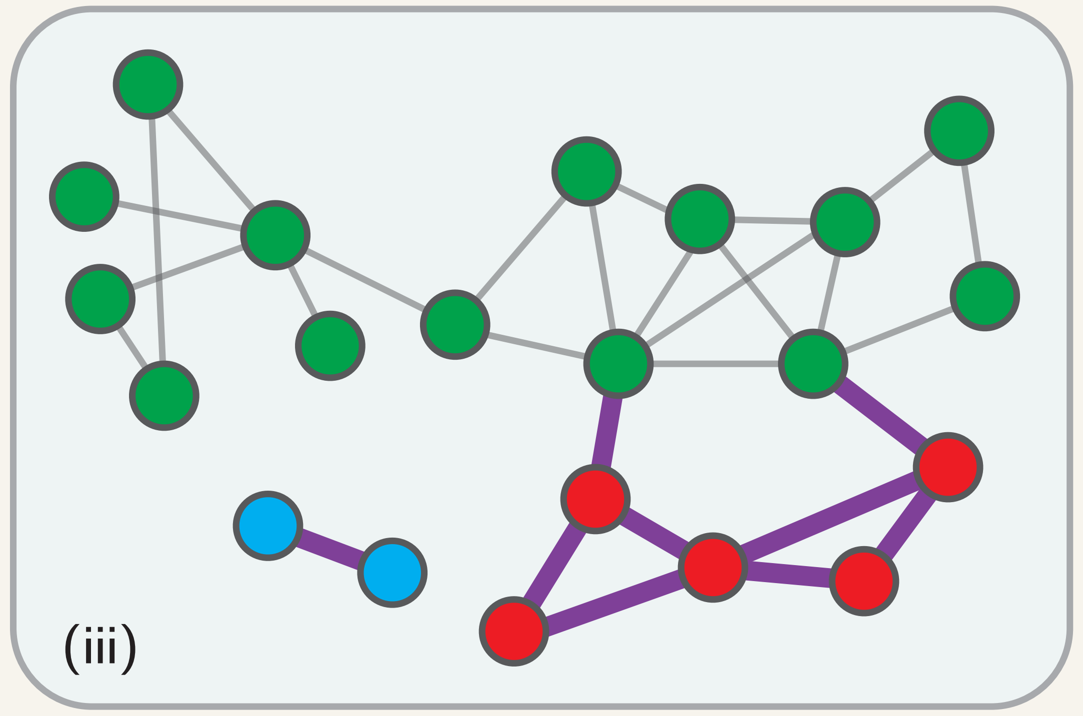
Input Graph



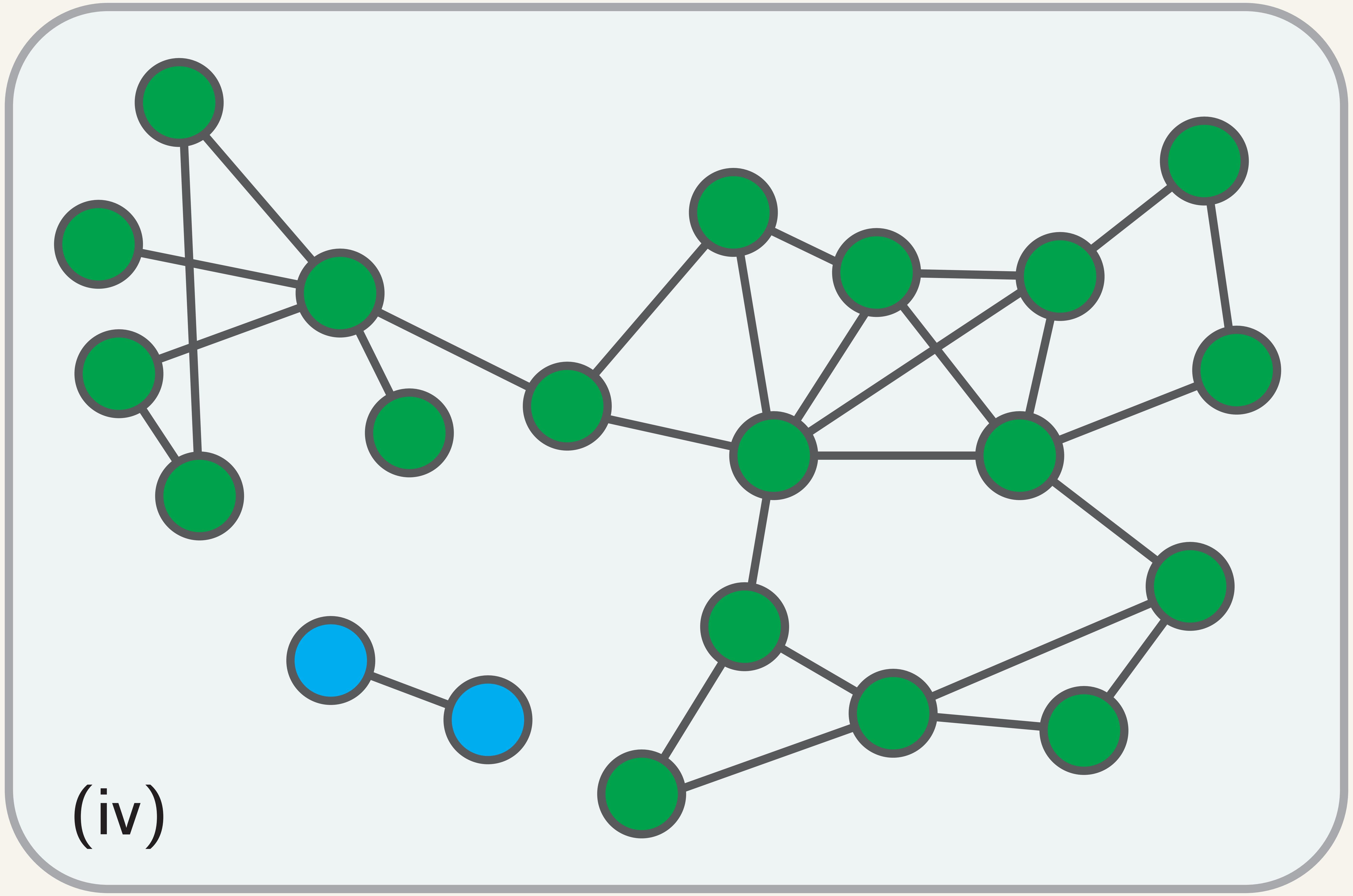
Sampled Labels



Finish Step on $v \notin L_{\max}$



Output Labeling



Properties of Sampling Methods

Connectivity Labeling

$C(u) = C(v)$ iff u and v are in the same component

Partial Connectivity Labeling

$C(u) = C(v)$ implies that u and v are in the same component

Properties of Sampling Methods

Let

$C = \text{SamplingMethod}(G)$

$C' = \text{Connectivity}(G[C])$

G[C] formed by merging all vertices v with the same label into a single vertex, and only preserving (u,v) edges s.t. C(u) and C(v) are distinct (removing duplicate edges)

A sampling method is **correct** if:

(1) $\forall v \in V,$ either $C(v) = v$ or $C(v) = r$ and $C(r) = r$

(2) $C'' = \{C'(C(v)) \mid v \in V\}$ is a connectivity labeling

Properties of Finish Methods

Let

$$C = \{i \to i \mid \forall i \in V\}$$

A connectivity algorithm is **monotone** if the algorithm updates the labels s.t. the updated labeling can be represented as the union of two trees in the previous labeling

I.e., once two vertices are in the same tree, they will always remain in the same tree.

Properties of Finish Methods

A connectivity algorithm operating on a labeling C is **linearizable monotone** if

(1) Its operations are linearizable.

(2) Every operation in the linearization order preserves monotonicity.

Composing a correct sampling method with a linearizable monotone finish algorithm yields a connectivity labeling.

Next:
Introduce several sampling and finish methods

k-Out Sampling

```
def kOutSample(G(V,E), labels, k=2):
  edges = {first edge from each vertex} U {sample k-1
           edges uniformly at random from each vertex}
  UnionFind(edges, labels)
  Fully compress the components array, in parallel
  return labels
```

Original scheme from Afforest connectivity algorithm (Sutton et al., 2018):

(1) Select the first two edges incident to each vertex (in gen. first k)

Can yield poor results depending on how vertices in the graph are ordered.

k-Out Sampling

```
def kOutSample(G(V,E), labels, k=2):
  edges = {first edge from each vertex} U {sample k-1
           edges uniformly at random from each vertex}
  UnionFind(edges, labels)
  Fully compress the components array, in parallel
  return labels
```

Theoretical motivation from Holm et al. (2019):

> Suppose each vertex of an arbitrary simple graph on n vertices chooses k random incident edges.
>
> Then the expected number of edges in the original graph connecting different connected components in the sampled subgraph is $O(n/k)$

Implies that by processing $O(nk)$ edges, only $O(n/k)$ edges need to be examined in the finish stage to compute a correct labeling.

LDD Sampling

```
def LDDSample(G(V,E), labels, beta=0.2):
  labels = LDD(G, beta)
  return labels
```

Recall theoretical guarantees of LDD:

(1) Strong diameter of each cluster is $O(\log n/\beta)$

(2) Number of intercluster edges is $O(\beta m)$ in expectation

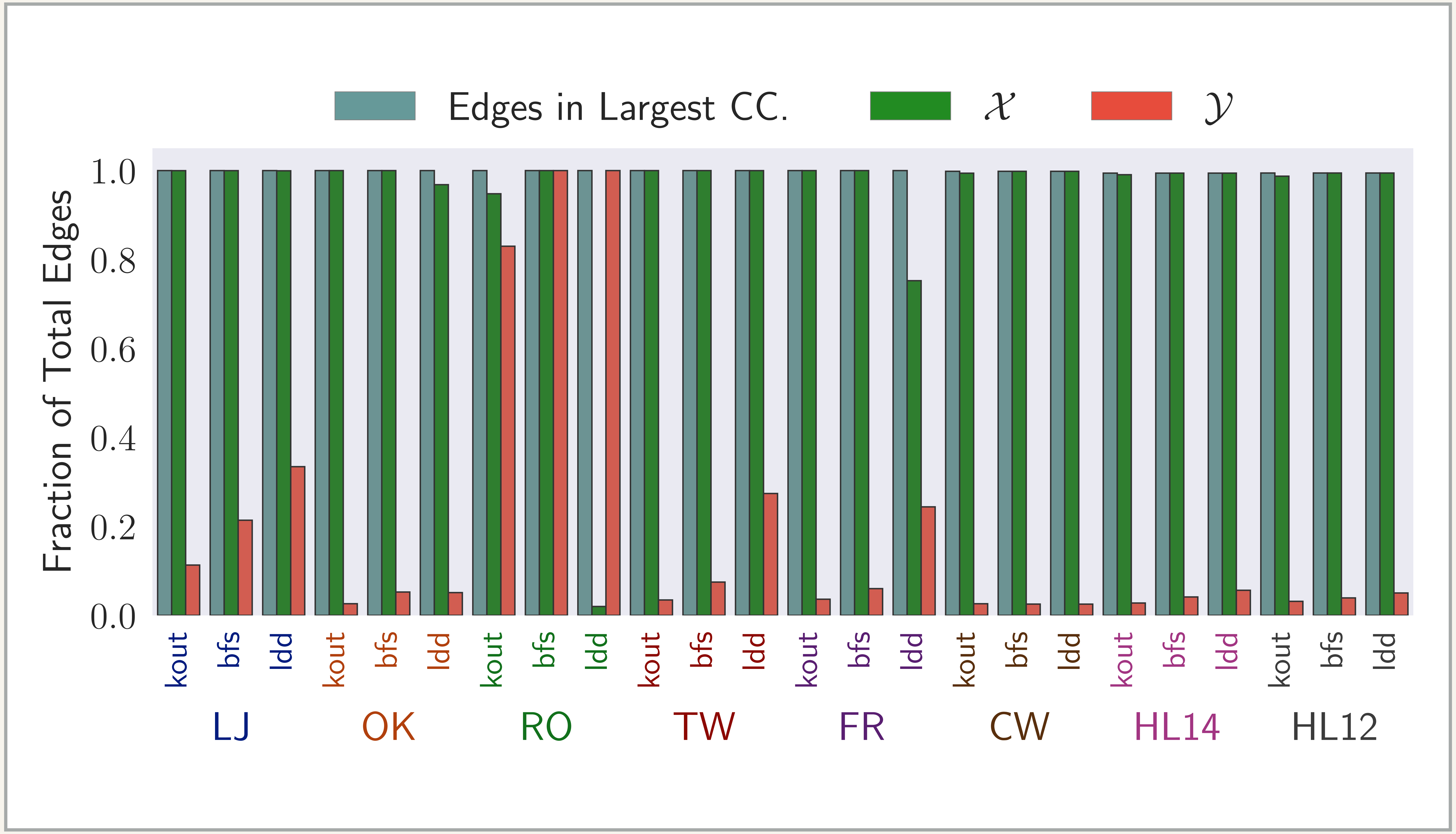
In practice, after one application of LDD, the resulting clustering often contains a single massive cluster.

BFS Sampling

```
def BFSSample(G(V,E), labels, c=5):
  for i in [0, c):
    # Run direction-optimizing BFS from random source.
    s = RandVertex()
    labels = LabelSpreadingBFS(G, s)

    # Check if BFS covered a significant fraction of the
    # vertices.
    freq = IdentifyFrequent(labels)
    if (freq makes up more than 10% of the labels) then:
      return labels

  # otherwise return identity labeling.
  return {i -> i | i in [0, |V|)}
```

Practical motivation: many real-world graphs contain a single massive (low-diameter) component which we will find with constant probability.

How do sampling strategies perform in practice?



Min-Based and Root-Based Algorithms

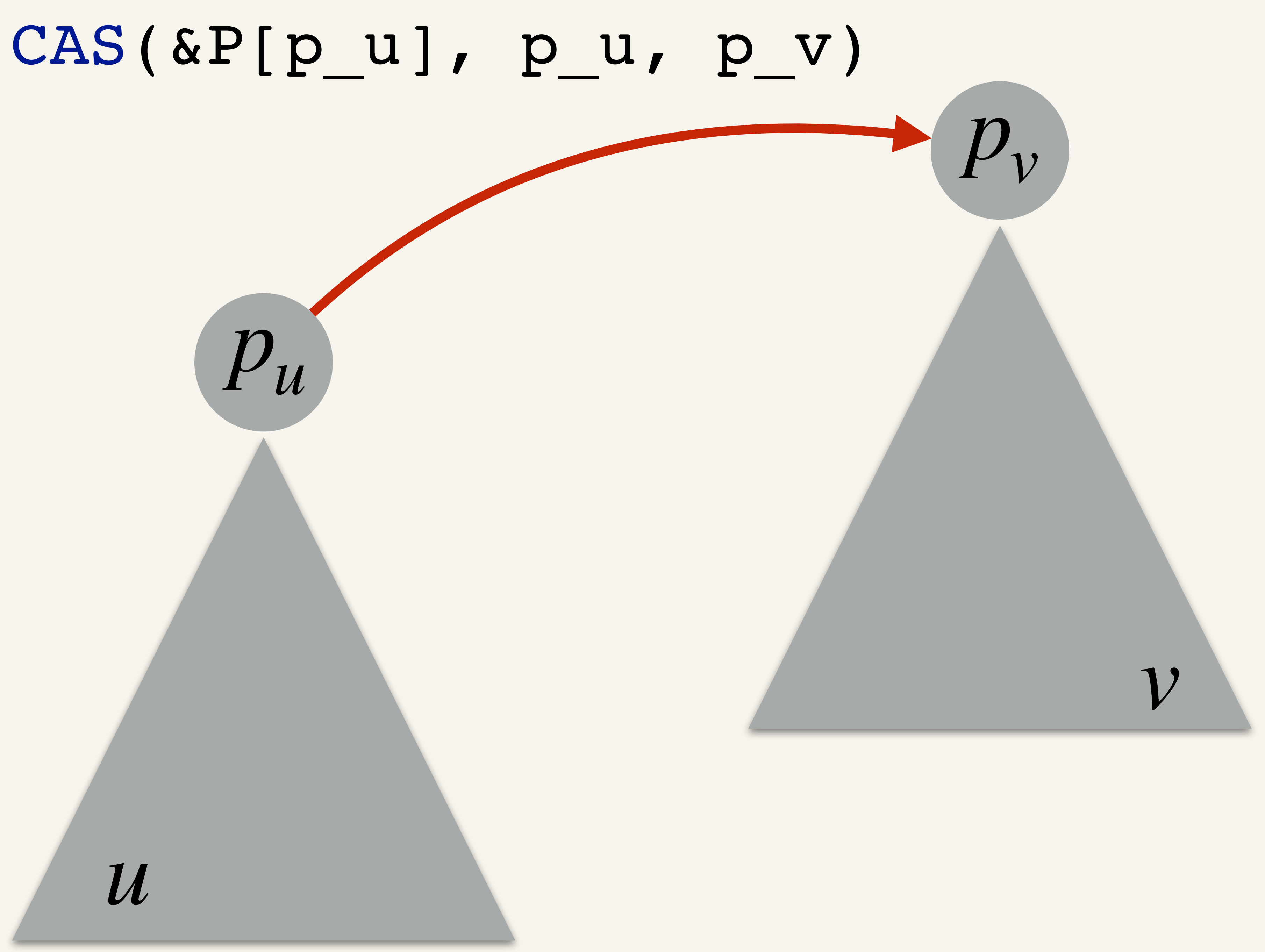
A **min-based** algorithm represents connectivity labelings as a collection of disjoint sets (similar to union-find), where all elements in a set are associated with the same label.

A min-based algorithm only updates the label of an element to a new label if the new label *is smaller than the previous label.*

A **root-based** algorithm is a special type of min-based algorithm which only links sets together by adding a link from the root of one tree to a node in another tree.

Asynchronous Union-Find: Union

```
def Union(u, v, P):
  p_u = Find(u, P)
  p_v = Find(v, P)
  while (p_u != p_v):
    if (p_u == P[p_u] and
          CAS(&P[p_u], p_u, p_v)):
      return
    p_u = Find(u, P)
    p_v = Find(v, P)
```



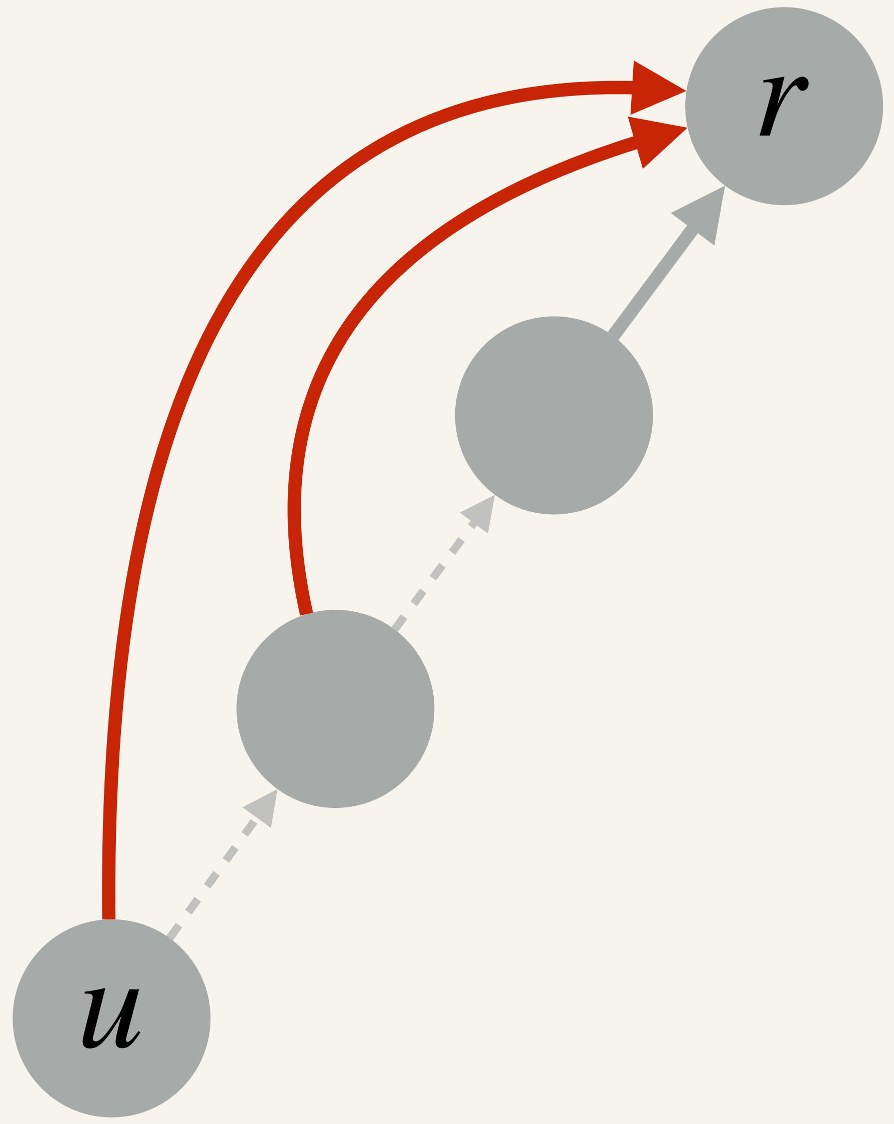
WLOG let p_u > p_v

(consistently link high to low or vice versa to prevent cycles)

Asynchronous Union-Find: Find and FindCompress

```
def FindCompress(u, P):
  # Find the root of u's tree, r. If u
  # is the root, quit.
  r = u
  if (P[r] == r):
    return r
  while (r != P[r]):
    r = P[r]

  # Make the parent of all vertices on
  # the u to r path r (or a smaller id).
  j = P[u]
  while (j > r):
    P[u] = r
    u = j
  return r
```

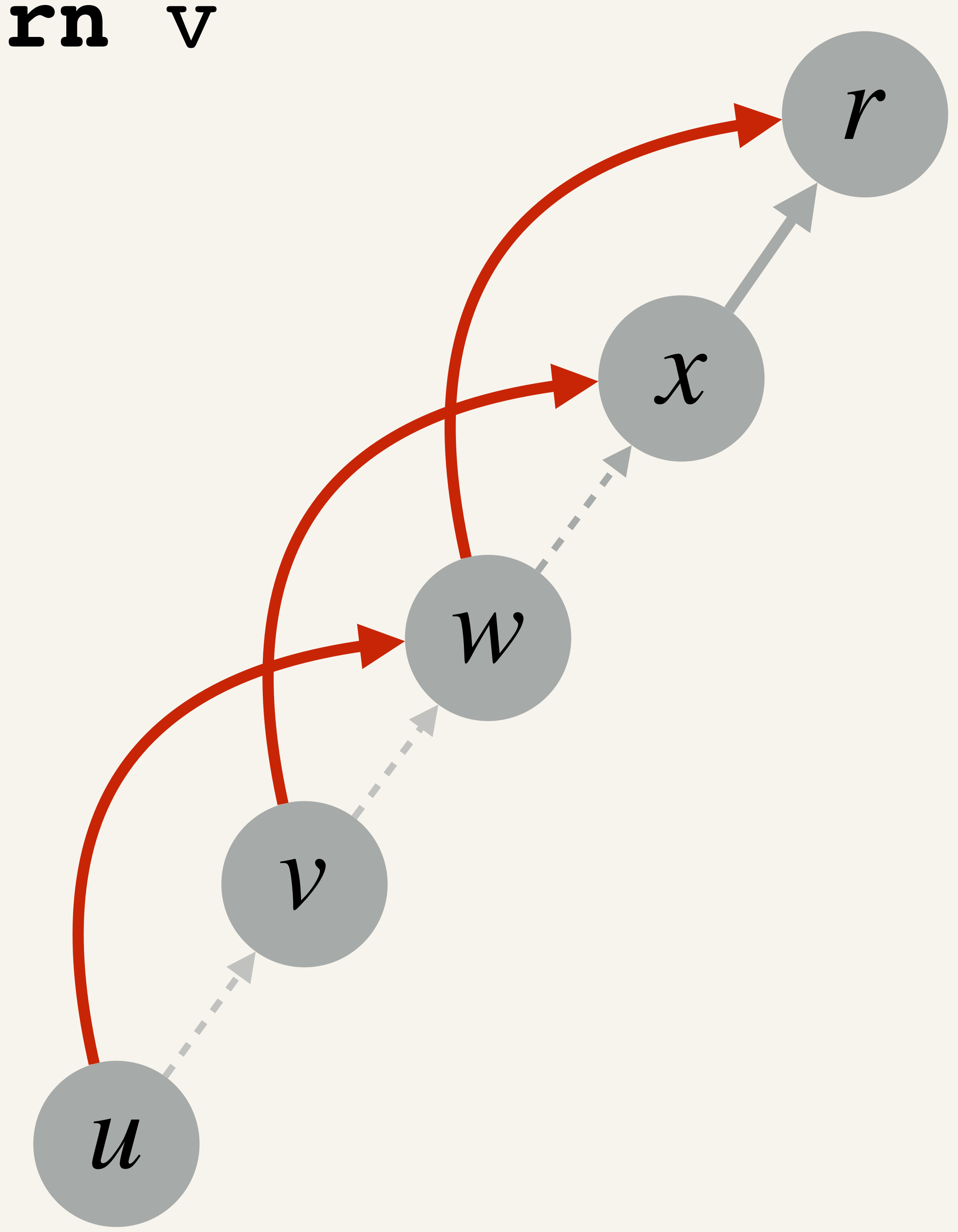
```
def FindNaive(u, P):
  v = u
  while (v != P[v]):
    v = P[v]
  return v
```



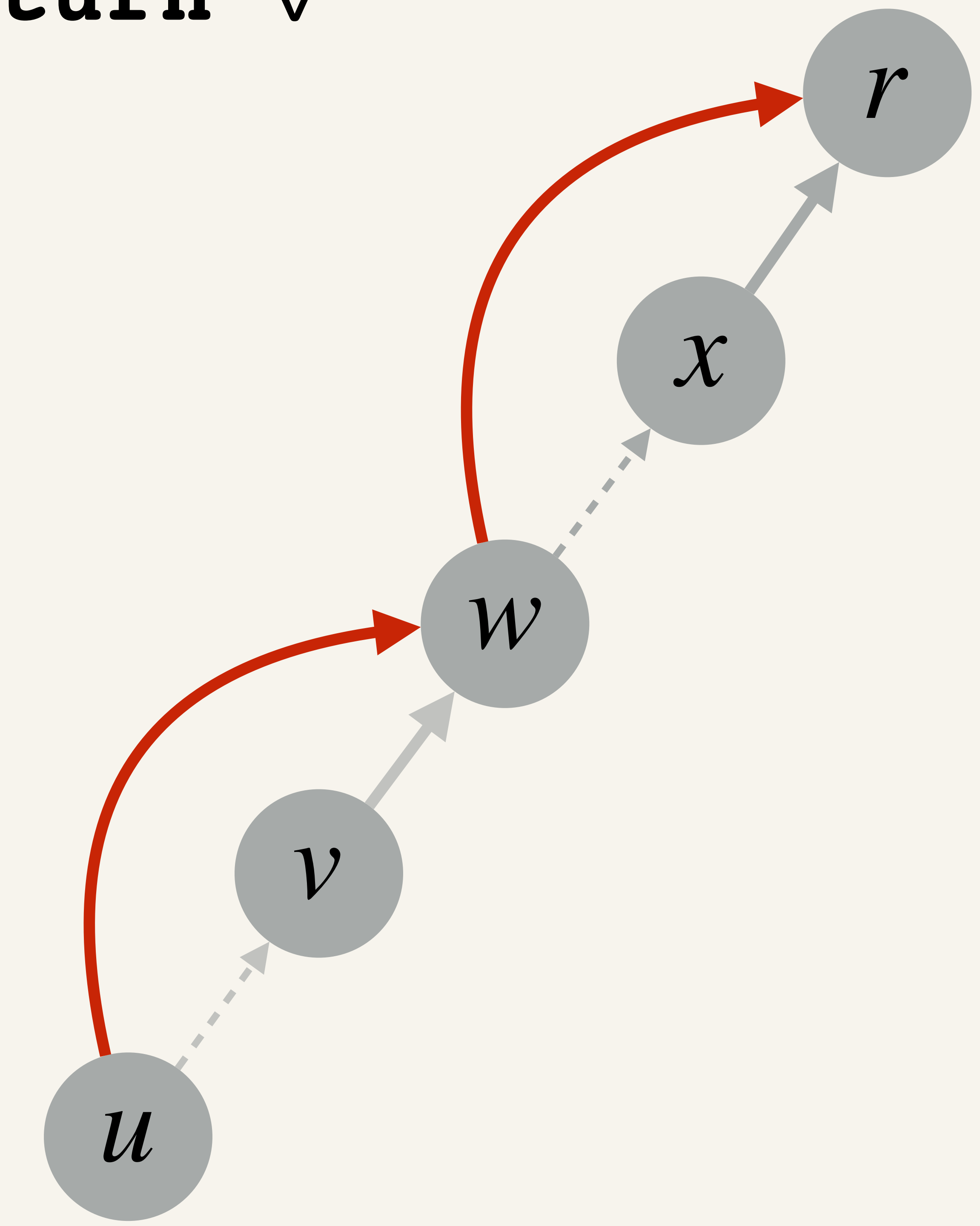
FindCompress(u, P)

Asynchronous Union-Find: Splitting and Halving

```
def FindAtomicSplit(u, P):
  v = P[u]   # parent(u)
  w = P[v]   # grandparent(u)
  while (v != w):
    CAS(&P[u], v, w)
    u = v
  return v
```




```
def FindAtomicHalve(u, P):
  v = P[u]   # parent(u)
  w = P[v]   # grandparent(u)
  while (v != w):
    CAS(&P[u], v, w)
    u = P[u]
  return v
```



Concurrent Rem's Algorithm

```
def Union(u, v, P):
  r_u = u, r_v = v
  while (P[r_u] != P[r_v]):
    # WLOG let P[r_u] > P[r_v].
    if (r_u == P[r_u] and
        CAS(&P[r_u], r_u, P[r_v])):
      # Success: linked the two trees.
      if (CompressOpt != FindNaive):
        Compress(u, P)
        Compress(v, P)
      return
    else:
      # Otherwise shorten path using splice.
      r_u = Splice(r_u, r_v, P)
```

Concurrent Rem's Algorithm: Splice Options

```
def HalveAtomicOne(u, x, P):
  v = P[u]  # parent
  w = P[v]  # grandparent
  if (u != w):
    CAS(&P[u], v, w)
  return w
```

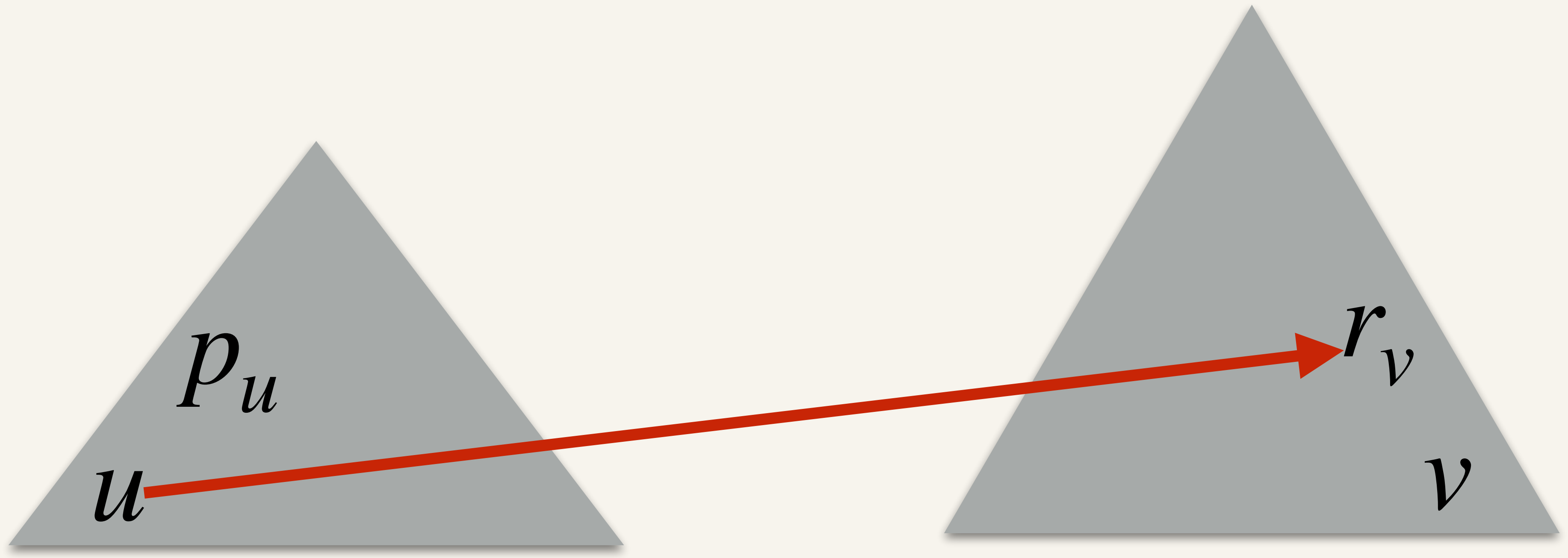
```
def SplitAtomicOne(u, x, P):
  v = P[u]  # parent
  w = P[v]  # grandparent
  if (u != w):
    CAS(&P[u], v, w)
  return v
```

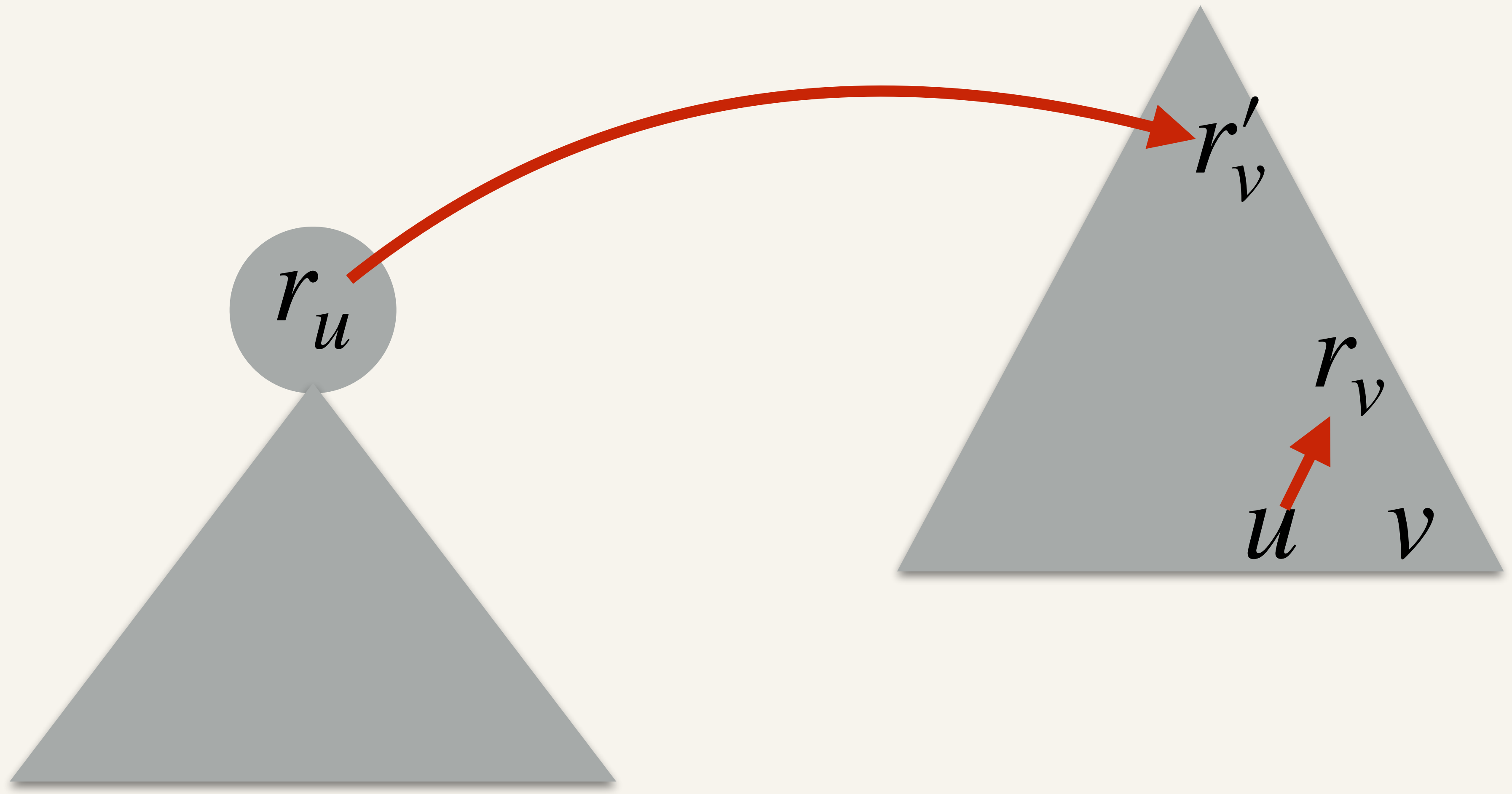
```
def SpliceAtomic(u, x, P):
  p_u = P[u]
  # Try to make u's parent x's parent which
  # could be a node in the other tree.
  CAS(&P[u], p_u, P[x])
  return p_u
```

Concurrent Rem's Algorithm: Splice Options

```
def Union(u, v, P):
  r_u = u, r_v = v
  while (P[r_u] != P[r_v]):
    # WLOG let P[r_u] > P[r_v].
    if (r_u == P[r_u] and
        CAS(&P[r_u], r_u, P[r_v])):
      # Success: linked the two trees.
      if (CompressOpt != FindNaive):
        Compress(u, P)
        Compress(v, P)
      return
    else:
      # Otherwise shorten path using splice.
      r_u = Splice(r_u, r_v, P)
```

```
def SpliceAtomic(u, x, P):
  p_u = P[u]
  # Try to make u's parent x's parent which
  # could be a node in the other tree.
  CAS(&P[u], p_u, P[x])
  return p_u
```





Other Min-Based Algorithms

Union-Find Algorithms

Jayanti-Tarjan (two-try split)

UF-Early

UF-Hooks

UF-Rem-Lock

Liu-Tarjan Algorithms

Family of min-based algorithms based on shortcutting

Shiloach-Vishkin

Label Propagation

Experiments

Dell PowerEdge R930

- 72-cores, 2-way hyper-threaded*
- 1TB of main memory
- Cost: about 20k USD

Graph Data

- Run on a collection of large real-world graphs, including largest publicly available graph (HL12)

Graph	n	m	Diam.	Num C.	Largest C.	LT-DC (s)	LT (s)
RO	23.9M	57.7M	6,809	1	23.9M	0.108	0.241
LJ	4.8M	85.7M	16	1,876	4.8M	0.101	0.226
CO	3.1M	234.4M	9	1	3.1M	0.094	0.520
TW	41.7M	2.4B	23*	1	41.7M	0.115	2.80
FR	65.6M	3.6B	32	1	65.6M	0.182	6.07
CW	978.4M	74.7B	132*	23.7M	950.5M	0.534	54.2
HL14	1.7B	124.1B	207*	129M	1.57B	1.02	101.3
HL12	3.6B	225.8B	331*	144M	3.35B	1.64	192.5

* (4 x 2.4GHz 18-core E7-8867 v4 Xeon processors)

Union-Find Comparison

	UF-JTB	UF-Rem-CAS;SpliceAtomic	UF-Rem-CAS;SplitAtomicOne	UF-Rem-CAS;HalveAtomicOne	UF-Rem-Lock;SpliceAtomic	UF-Rem-Lock;SplitAtomicOne	UF-Rem-Lock;HalveAtomicOne	UF-Early	UF-Hooks	UF-Async
FindTwoTrySplit	4.2									
FindCompress			1.3	1.3		1.9	1.9	6.6	1.6	1.7
FindHalve		1.2	1.3	1.4	1.8	1.8	1.8	6.5	1.4	3.3
FindSplit		1.3	1.4	1.4	1.7	1.7	1.7	6.3	1.4	3.3
FindNaive	5.9	1	1	1	1.5	1.5	1.5	4.8	1.5	3.3

UF-Rem-CAS with splice/split/halve and no additional compression reliably performs the best across all inputs

Comparison on WebDataCommons Hyperlink2012

System	Graph	Mem. (TB)	Threads	Nodes	Time (s)
Mosaic [72]	Hyperlink2014	0.768	1000	1	708
FlashGraph [114]	Hyperlink2012	.512	64	1	461
GBBS [32]	Hyperlink2012	1	144	1	25.8
GBBS (NVRAM) [34]	Hyperlink2012	0.376	96	1	36.2
Galois (NVRAM) [43]	Hyperlink2012	0.376	96	1	76.0
Slota et al. [99]	Hyperlink2012	16.3	8192	256	63
Stergiou et al. [101]	Hyperlink2012	128	24000	1000	341
Gluon [30]	Hyperlink2012	24	69632	256	75.3
Zhang et al. [113]	Hyperlink2012	$\geq$ 256	262,000	4096	30
ConnectIt	Hyperlink2014	1	144	1	**2.83**
	Hyperlink2012	1	144	1	**8.20**

Table 1: System configurations, including memory (terabytes), num. hyper-threads and nodes, and running times (seconds) of connectivity results on the Hyperlink graphs. The last rows show the fastest ConnectIt times. The fastest time per graph is shown in green.

- Fastest ConnectIt algorithm for HL2012 is 3.65—41.5x faster than existing distributed memory results while using orders of magnitude fewer resources

- Running time without sampling on HL2012 of our fastest algorithm is 13.9 seconds (1.69x speedup using k-Out Sampling)

Comparing No-Sampling with Sampling

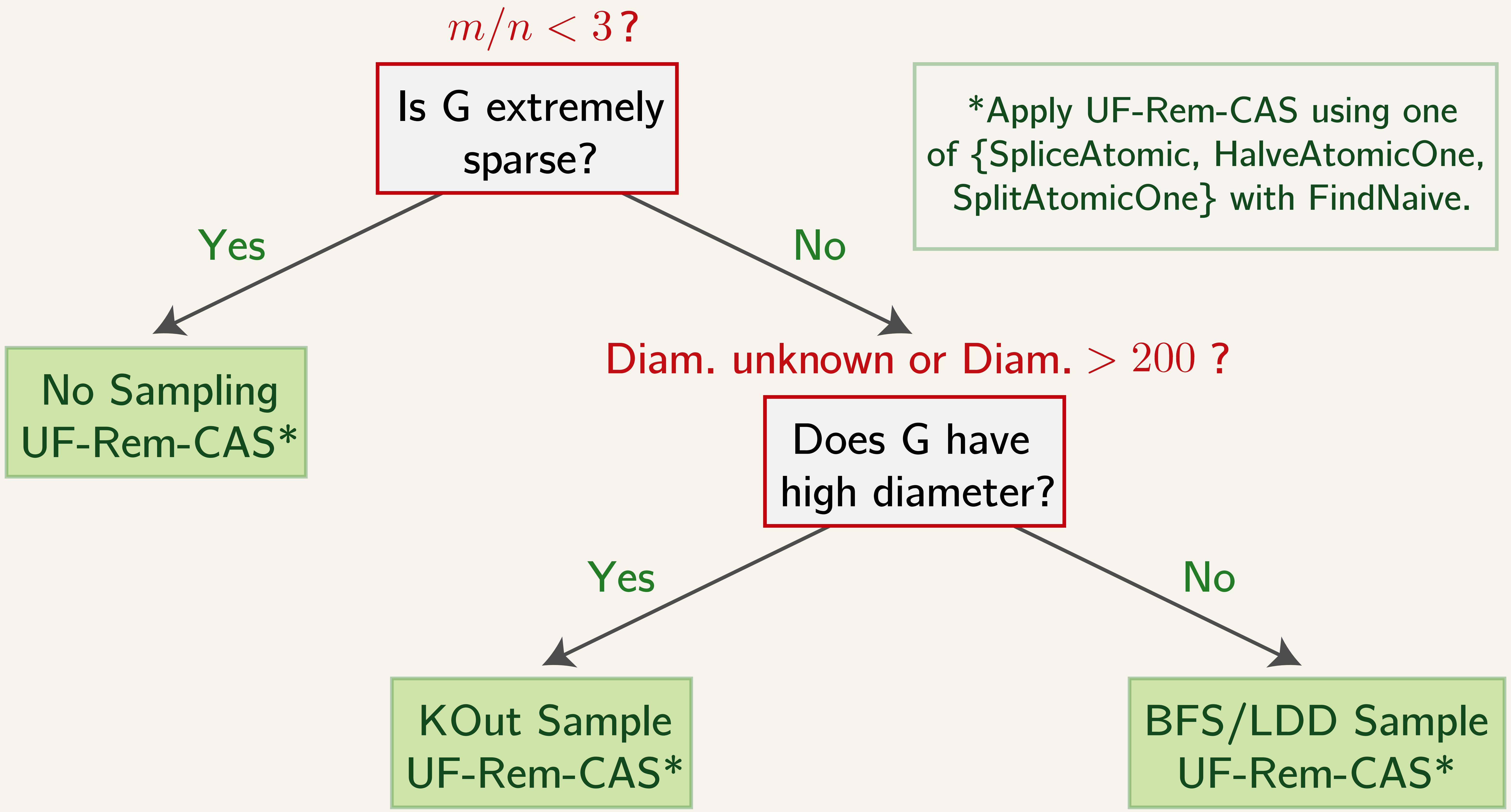
Grp.	Algorithm	RO	LJ	CO	TW	FR	CW	HL14	HL12
No Sampling	UF-Early	3.61e-2	3.48e-2	8.63e-2	2.52	1.50	59.8	17.0	32.9
	UF-Hooks	3.37e-2	1.75e-2	2.69e-2	0.390	1.17	6.05	9.37	20.0
	UF-Async	4.02e-2	2.03e-2	3.12e-2	0.426	1.21	7.92	12.2	25.5
	UF-Rem-CAS	**2.80e-2**	1.27e-2	1.91e-2	0.316	0.902	4.04	6.64	13.9
	UF-Rem-Lock	5.07e-2	1.95e-2	2.84e-2	0.437	1.23	5.64	9.20	19.3
	UF-JTB	6.90e-2	4.49e-2	8.48e-2	0.965	2.76	22.5	36.4	72.1
	Liu-Tarjan	7.40e-2	5.18e-2	6.46e-2	2.78	6.60	30.1	67.1	142
	SV	0.138	4.34e-2	5.70e-2	1.65	5.38	21.2	38.5	106
	Label-Prop	13.4	4.66e-2	6.37e-2	1.24	4.37	13.4	20.7	46.5
k-out Sampling	UF-Early	3.25e-2	9.00e-3	8.61e-3	0.117	0.227	2.28	4.77	8.94
	UF-Hooks	3.62e-2	9.18e-3	9.16e-3	0.121	0.230	2.22	3.63	8.51
	UF-Async	3.33e-2	8.97e-3	**8.56e-3**	0.117	0.228	2.21	3.60	8.49
	UF-Rem-CAS	3.43e-2	**8.96e-3**	8.62e-3	0.117	0.227	2.15	3.51	**8.20**
	UF-Rem-Lock	4.45e-2	1.13e-2	1.01e-2	0.138	0.344	2.63	4.33	9.91
	UF-JTB	3.89e-2	9.77e-3	8.80e-3	0.125	0.237	2.43	4.05	9.58
	Liu-Tarjan	6.34e-2	9.90e-3	9.18e-3	0.129	0.374	2.61	6.74	11.5
	SV	5.72e-2	9.72e-3	8.78e-2	0.124	0.237	2.70	5.03	12.5
	Label-Prop	12.6	1.02e-2	9.63e-3	0.121	0.375	2.44	4.75	9.68

Grp.	Algorithm	RO	LJ	CO	TW	FR	CW	HL14	HL12
BFS Sampling	UF-Early	2.69	1.07e-2	9.26e-3	9.42e-2	0.186	2.27	4.02	9.33
	UF-Hooks	2.65	1.09e-2	9.71e-3	9.53e-2	0.186	2.29	2.94	9.40
	UF-Async	2.69	1.08e-2	9.12e-3	9.31e-2	0.189	2.23	2.87	9.23
	UF-Rem-CAS	2.66	1.06e-2	9.19e-3	**9.24e-2**	**0.183**	2.21	**2.83**	9.11
	UF-Rem-Lock	2.67	1.13e-2	1.07e-2	0.113	0.219	2.69	3.68	10.8
	UF-JTB	2.75	1.14e-2	9.52e-3	9.80e-2	0.195	2.38	3.22	9.88
	Liu-Tarjan	2.68	1.17e-2	9.80e-3	9.61e-2	0.383	2.85	7.61	13.4
	SV	2.54	1.12e-2	9.72e-3	9.87e-2	0.196	2.59	4.13	12.2
	Label-Prop	2.58	1.19e-2	1.03e-2	9.47e-2	0.446	2.31	3.21	9.91
LDD Sampling	UF-Early	0.117	1.32e-2	8.63e-3	0.124	0.193	1.74	4.63	8.52
	UF-Hooks	0.112	1.33e-2	8.81e-3	0.127	0.197	1.75	3.58	8.46
	UF-Async	0.103	1.32e-2	8.49e-3	0.123	0.193	1.71	3.48	8.31
	UF-Rem-CAS	9.86e-2	1.29e-2	8.48e-3	0.122	0.193	**1.69**	3.46	8.28
	UF-Rem-Lock	0.126	1.54e-2	1.03e-2	0.144	0.226	2.16	4.31	9.97
	UF-JTB	0.148	1.35e-2	8.98e-3	0.131	0.202	1.85	3.84	9.13
	Liu-Tarjan	0.178	1.45e-2	8.73e-3	0.130	1.24	2.32	8.33	12.5
	SV	0.250	1.36e-2	8.81e-3	0.131	0.197	2.07	4.70	11.2
	Label-Prop	14.3	1.41e-2	8.99e-3	0.127	2.03	1.76	3.79	9.06

- Union-Find algorithms essentially always the fastest
- Sampling does not help much on very sparse graphs (avg degree in RO = 2.41)

- UF-Rem-CAS is consistently the fastest finish algorithm across all settings
- No significant difference between using SplitAtomicOne / HalveAtomicOne / SpliceAtomic

Algorithm Recommendations



- Tuning recommendations based on studying sampling performance on both real-world and synthetic networks (see paper)

Summary: ConnectIt

ConnectIt: framework for static and incremental parallel graph connectivity

- Simple to generate new combinations of sampling and finish algorithms
- Our fastest implementations of connectivity significantly outperform state-of-the-art parallel solutions
- Solutions for connectivity extend to parallel spanning forest and incremental connectivity

Code available as part of GBBS:

github.com/paralg/gbbs