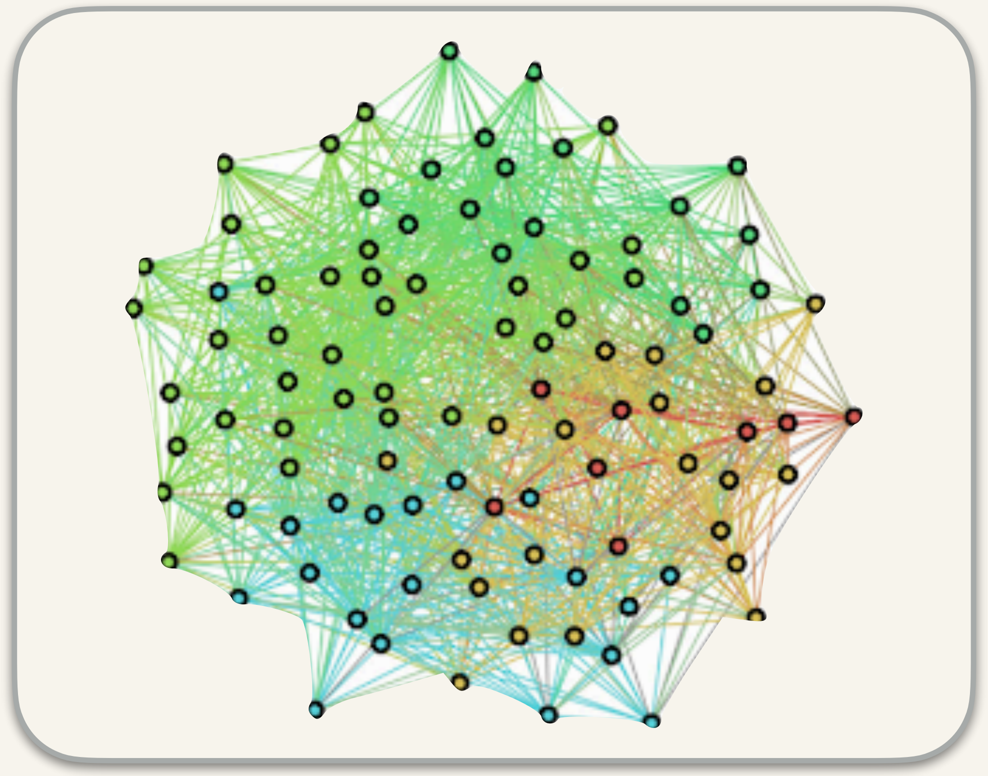
Julienne and the Graph-Based Benchmark Suite

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

Based on joint work with Guy Blelloch and Julian Shun

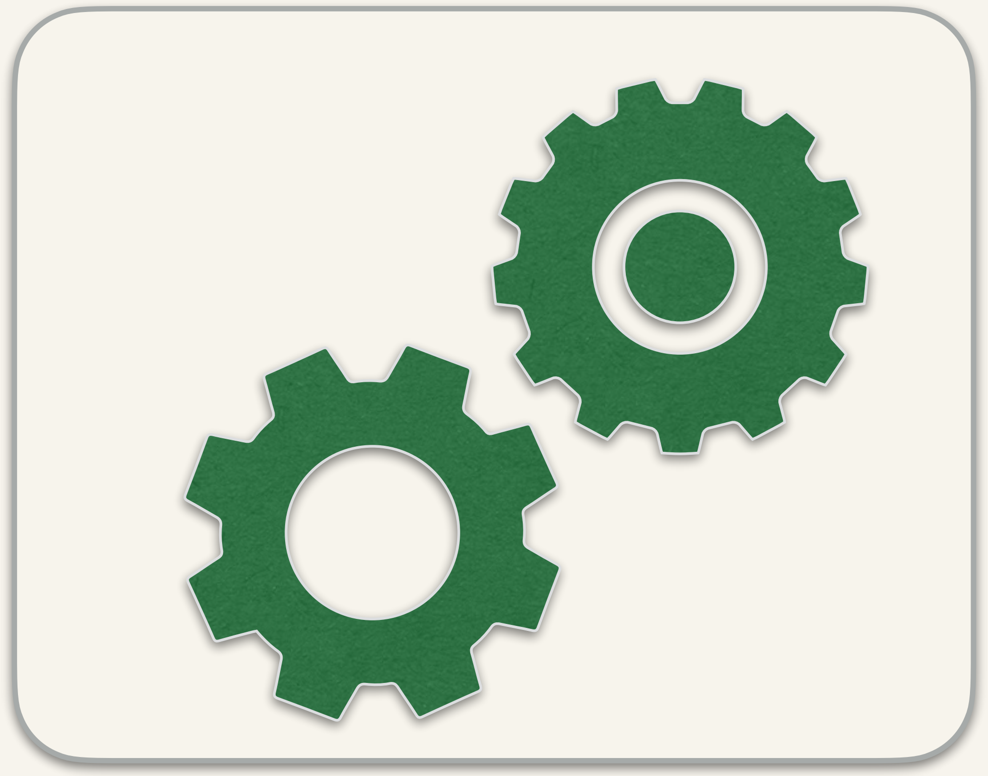
Graph Processing: algorithms and systems that enable us to analyze and understand graphs

Input Graph



- Static
- Dynamic

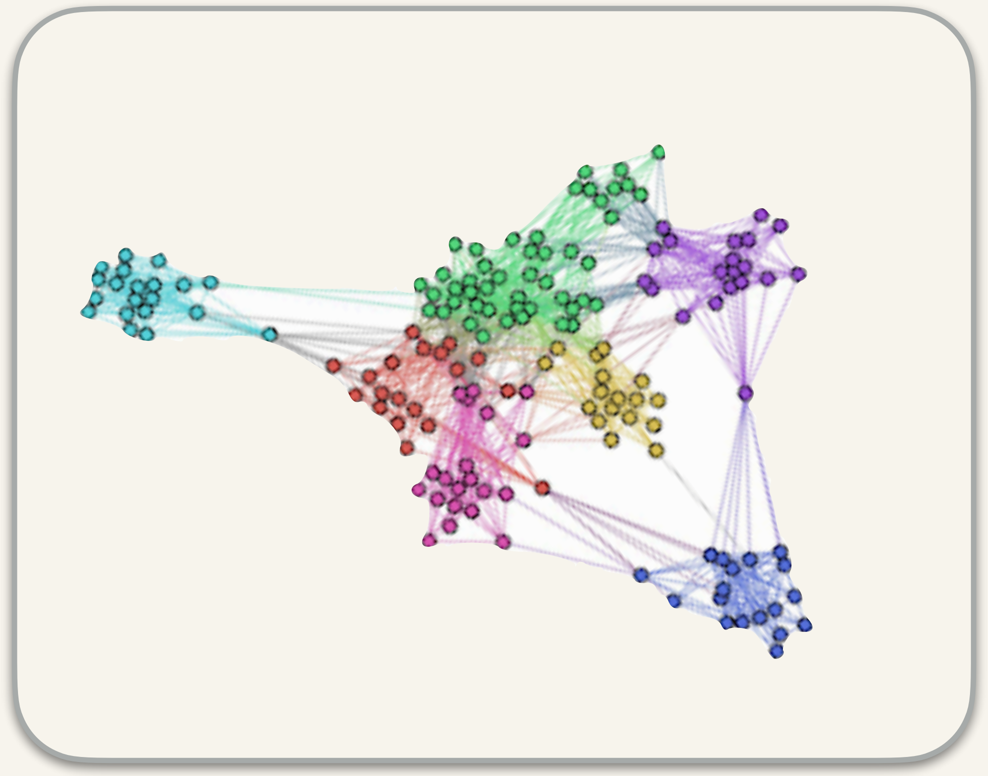
Graph Processing



Algorithms

- Connectivity
- Graph Clustering
- …
- Distance Computations
- Dense Subgraphs

Output



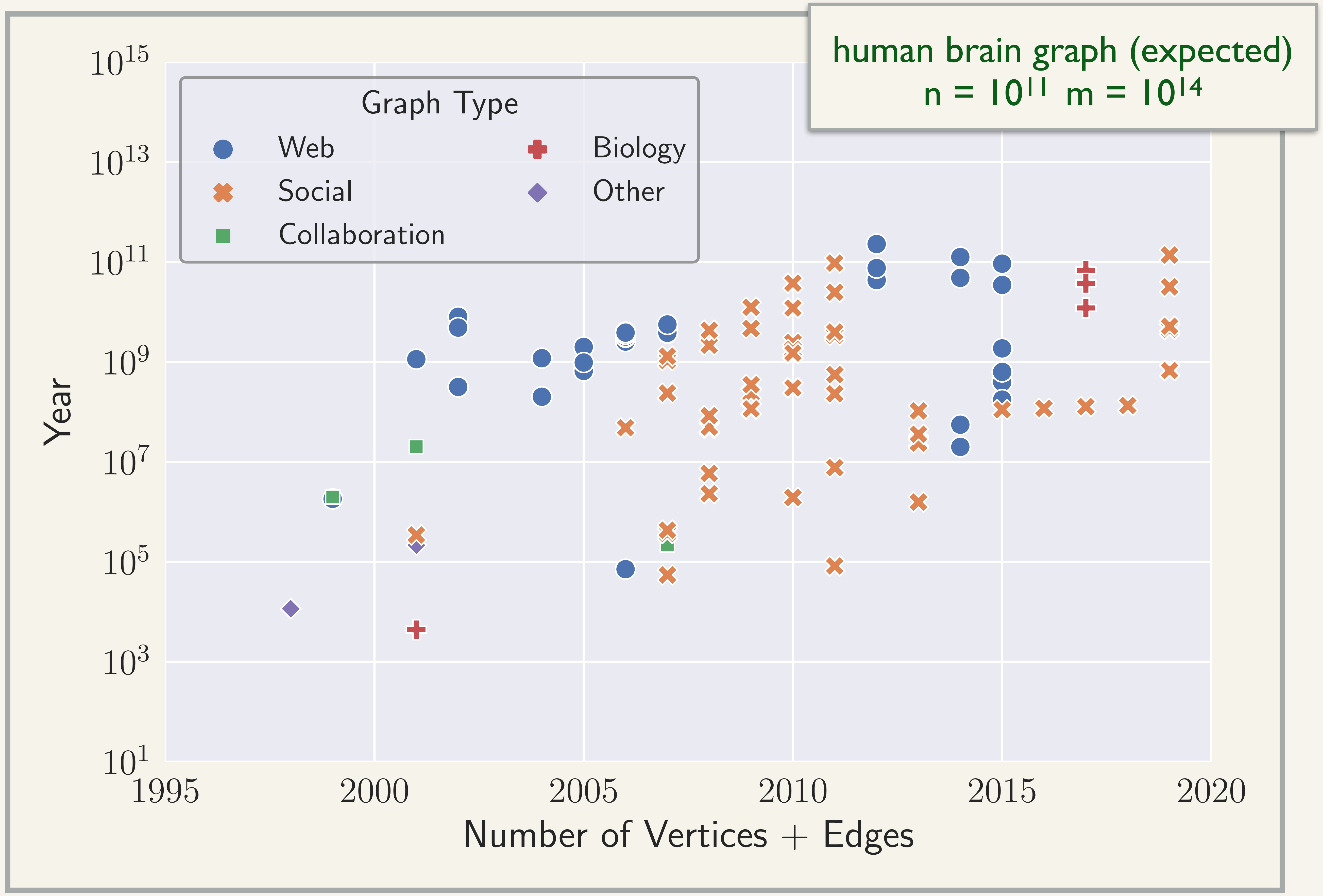
- Understanding
- Visualizations
- Graph-based features
- System-optimization

Large-Scale Graph Processing

WebDataCommons hyperlink graph

- 3.5 billion vertices and 128 billion edges
- ~1TB of memory to store
- Largest publicly available graph

"…[the 2012 graph is the] largest hyperlink graph that is available to the public outside companies such as Google, Yahoo, and Microsoft."



Year of sourcing vs total number of vertices and edges for real-world graphs from the SNAP and LAW datasets

Sources: https://snap.stanford.edu/data/, http://law.di.unimi.it/datasets.php, http://webdatacommons.org/hyperlinkgraph/

Large-Scale Graph Processing

WebDataCommons hyperlink graph

- 3.5 billion vertices and 128 billion edges
- ~1TB of me
- Largest publicly available

“…[the 2012 graph is the] largest hyperlink graph that is available to the public outside companies such as Google, Yahoo, and Microsoft.”

Parallelism is the key to processing very large graphs in a timely manner

Year of sourcing vs total number of vertices and edges for real-world graphs from the SNAP and LAW datasets

Sources: https://snap.stanford.edu/data/, http://law.di.unimi.it/datasets.php, http://webdatacommons.org/hyperlinkgraph/

Shared-Memory Parallelism

Shared-Memory Machines

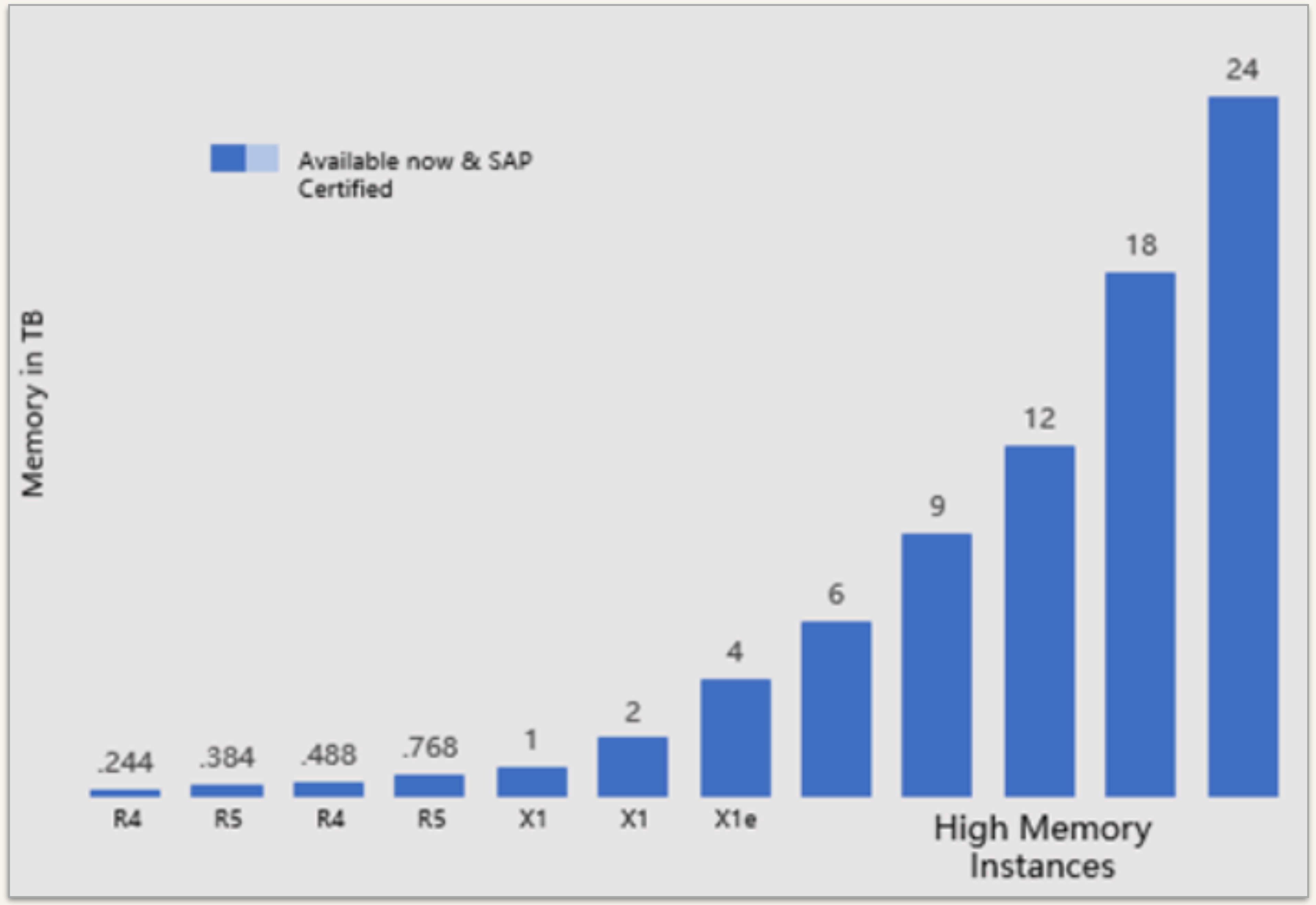
- Cost for a 1TB memory machine with 72 processors is about $20,000.

- Can rent a similar machine (96 processors and 1.5TB memory) for $11/hour on Google Cloud

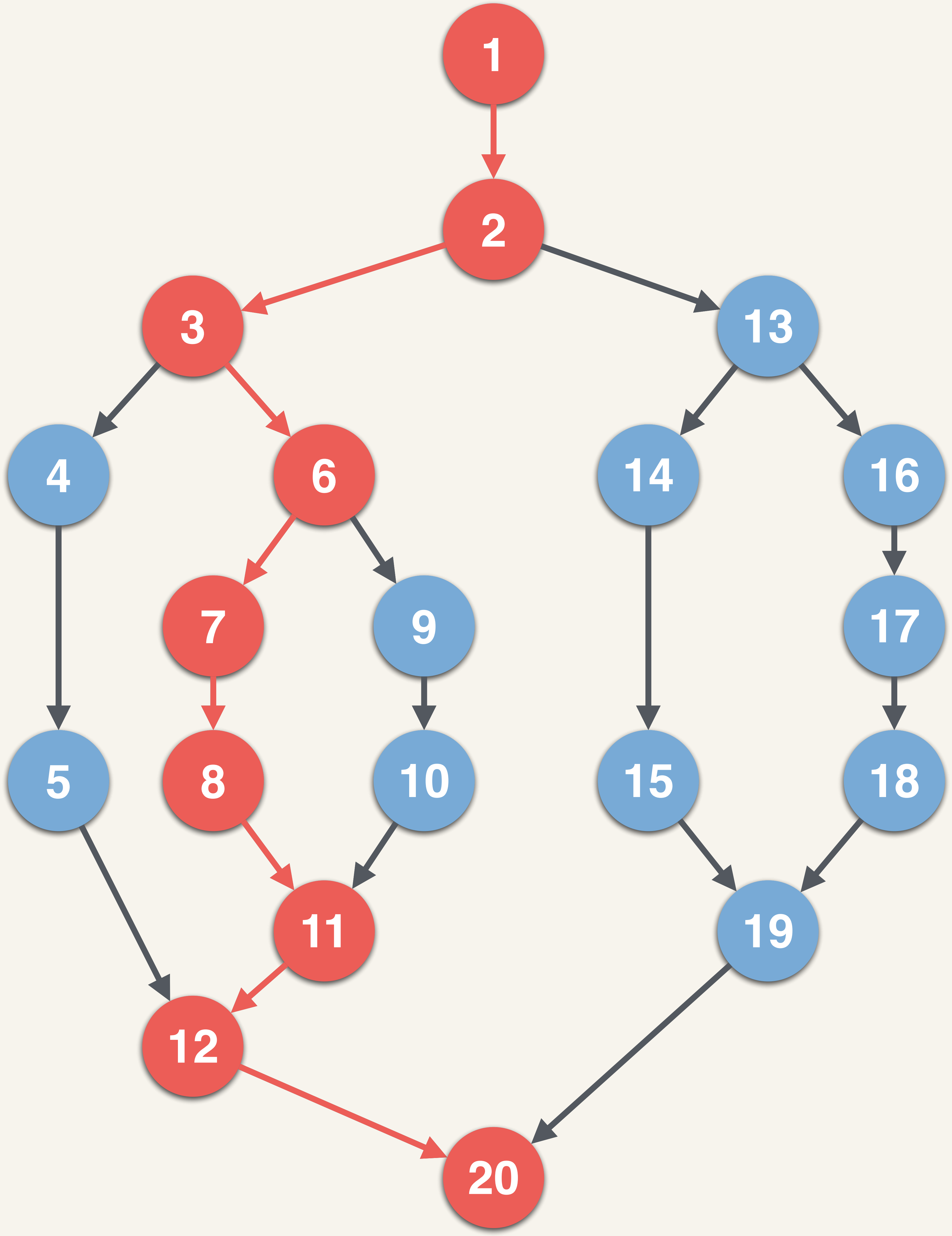
WebDataCommons Graph

- 3.5 billion vertices and 128 billion edges

A single shared-memory machine can already store the largest publicly available graph datasets, with plenty of room to spare



Work-Depth Model



Computation Graph

Work = total number of vertices in the computation graph

Depth = longest directed path in the graph (dependence length)

Running Time = Work/#Processors + O(Depth)

A *work-efficient* parallel algorithm has work that asymptotically matches that of the best sequential algorithm for the problem

Goal: work-efficient and low (polylogarithmic) depth algorithms

Theoretical Efficiency

A parallel algorithm is *theoretically-efficient* if it has good bounds on its work and depth

Why do we care about theoretical bounds?

Input-agnostic design

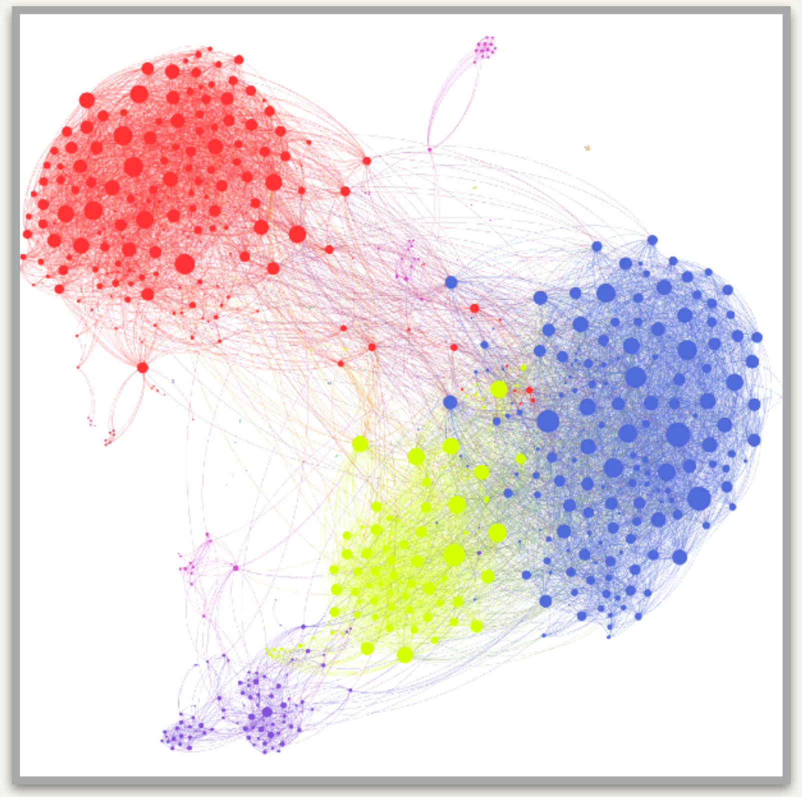
- Design codes without worrying too much about your datasets

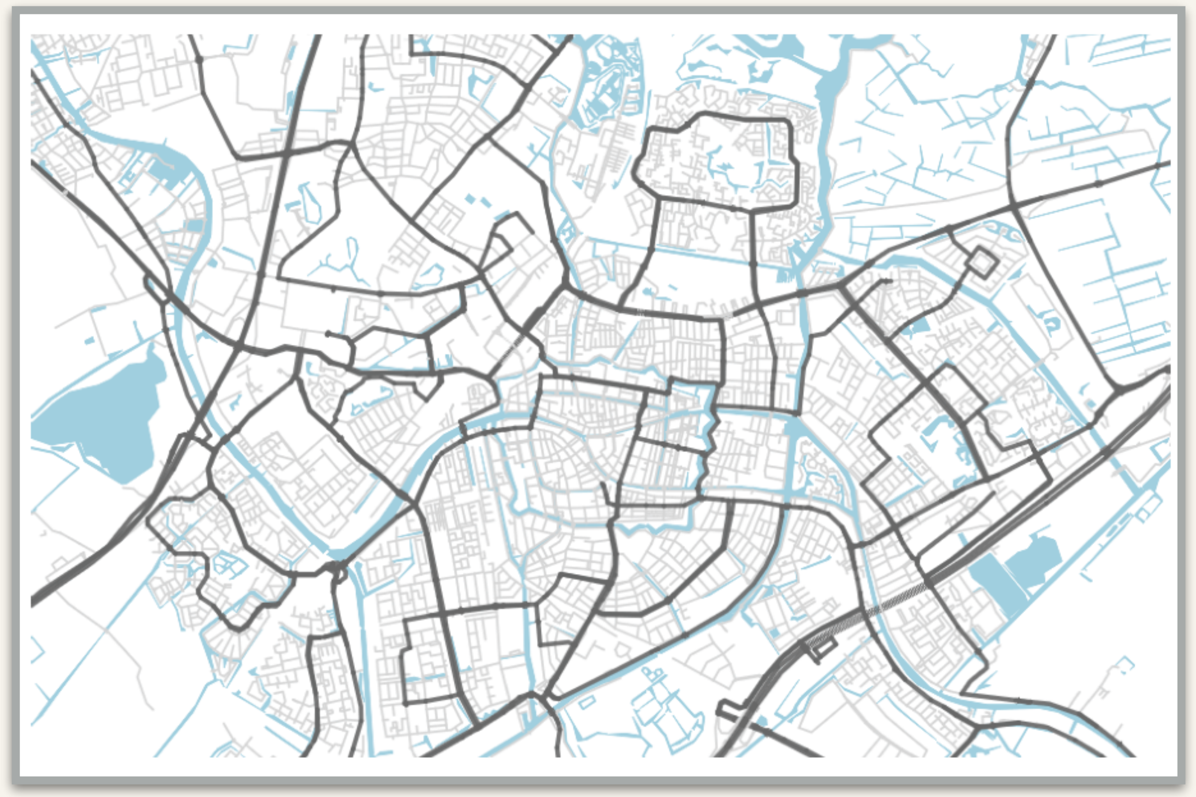
Robustness to bad inputs

- Perform well even on new classes of graphs
- Understand how they will scale on larger graphs

Work-efficiency matters in practice

- Work-efficient algorithms can be much faster than work-inefficient algorithms





Up to 9x faster using a work-efficient k-core algorithm (described in this talk)

Julienne: A Framework for Parallel Graph Algorithms using Work-efficient Bucketing [DBS'17]

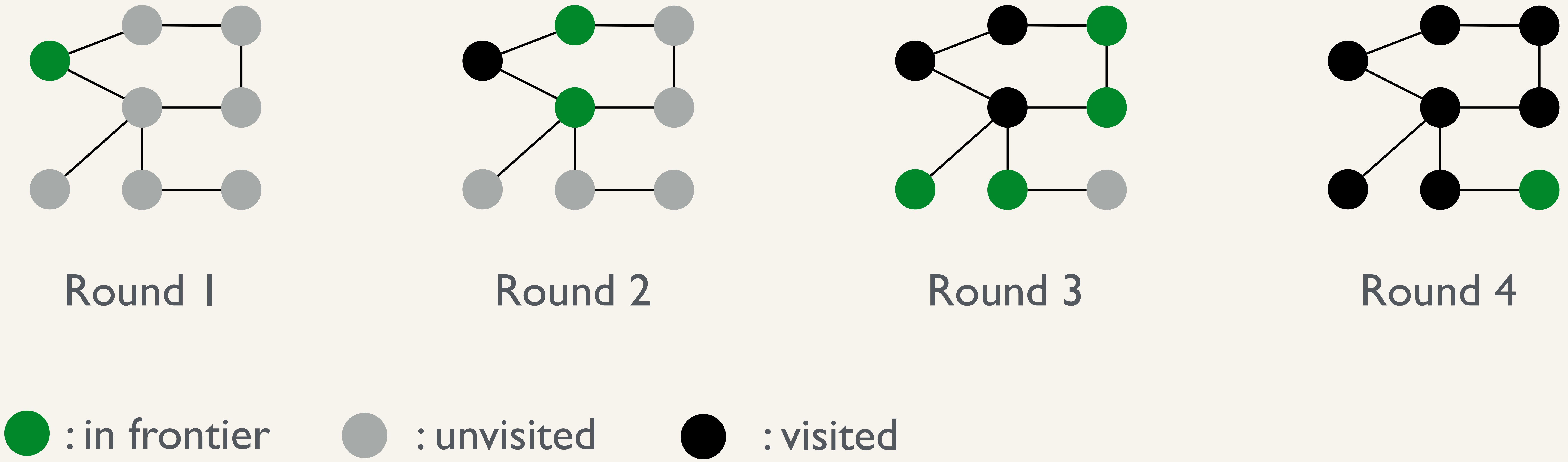
How do we design theoretically-efficient parallel graph algorithms for a certain class of *bucketing-based* problems

Frontier-Based Algorithms in Ligra

Primitives

- Frontier data-structure (vertexSubset)
- Map over vertices in a frontier (vertexMap)
- Map over out-edges of a frontier to generate new frontier (edgeMap)

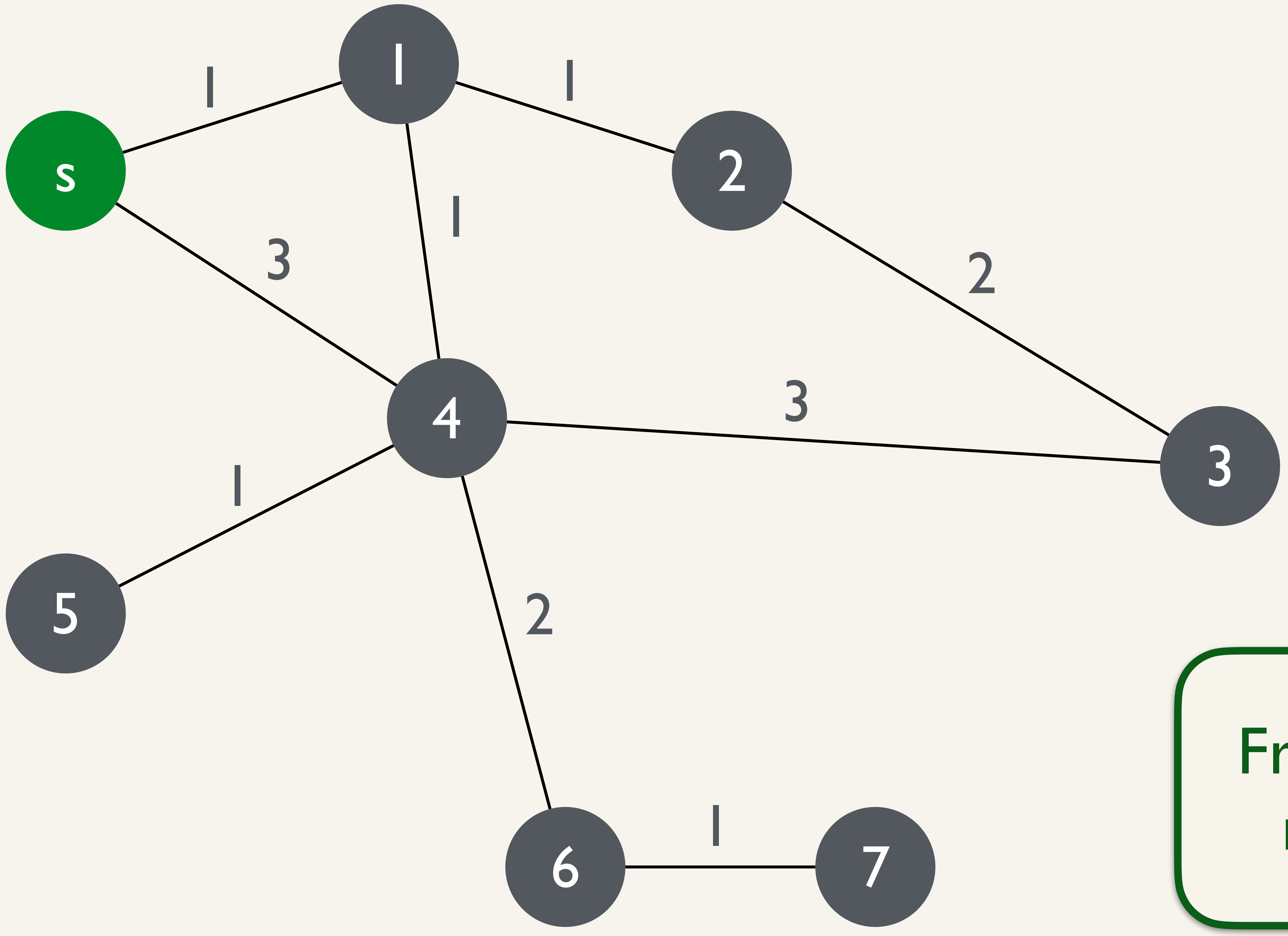
Example: Breadth-First Search



Weighted Breadth-First Search

Given: $G = (V, E, w)$ with *positive integer edge weights*, and a source **s**

Problem: Compute the shortest path distances from **s**

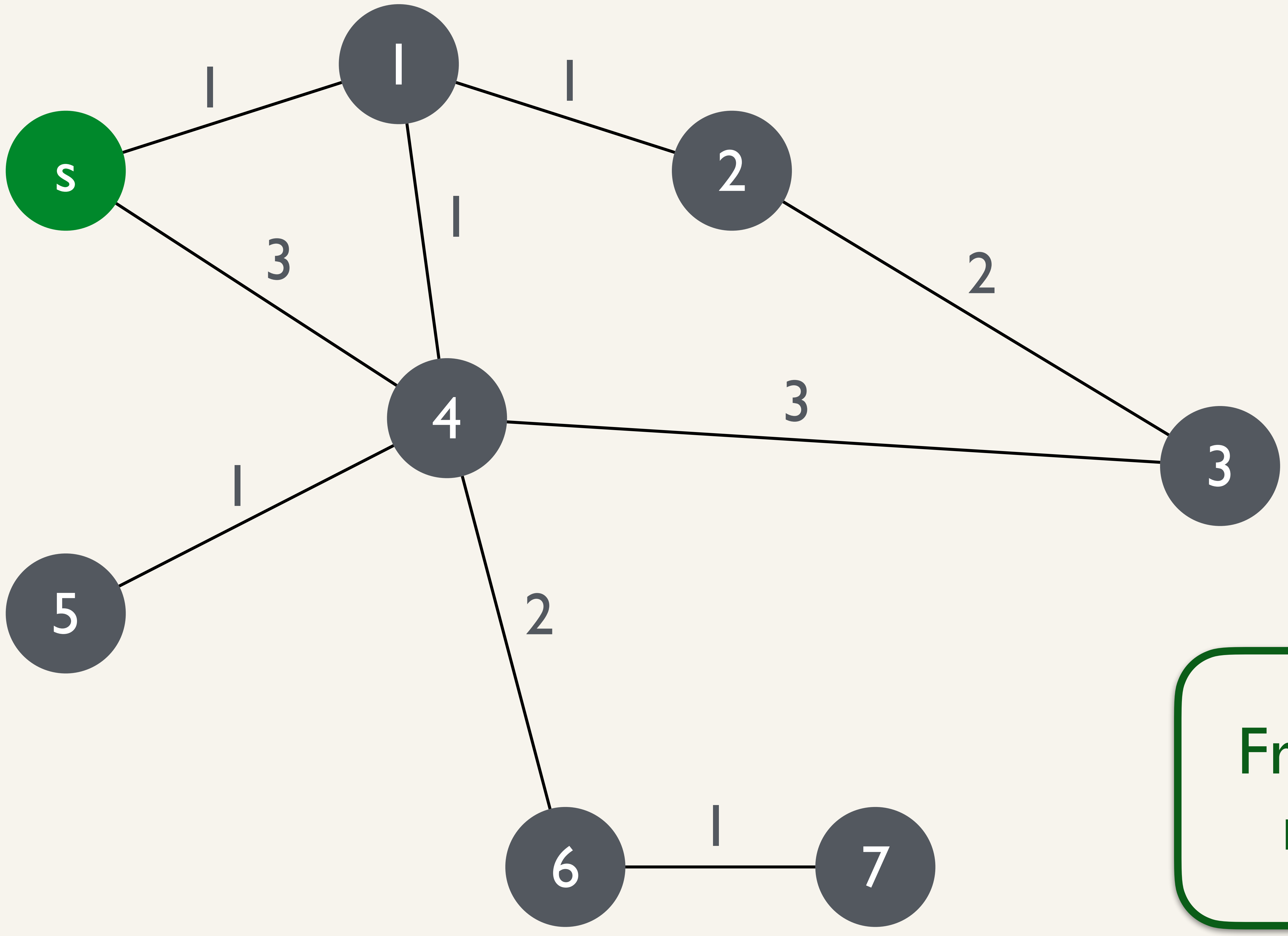


Frontier-based approach: on each step, visit all neighbors that had their distance decrease

Weighted Breadth-First Search

Given: $G = (V, E, w)$ with *positive integer edge weights*, and a source **s**

Problem: Compute the shortest path distances from **s**



Round 1

Frontier: s

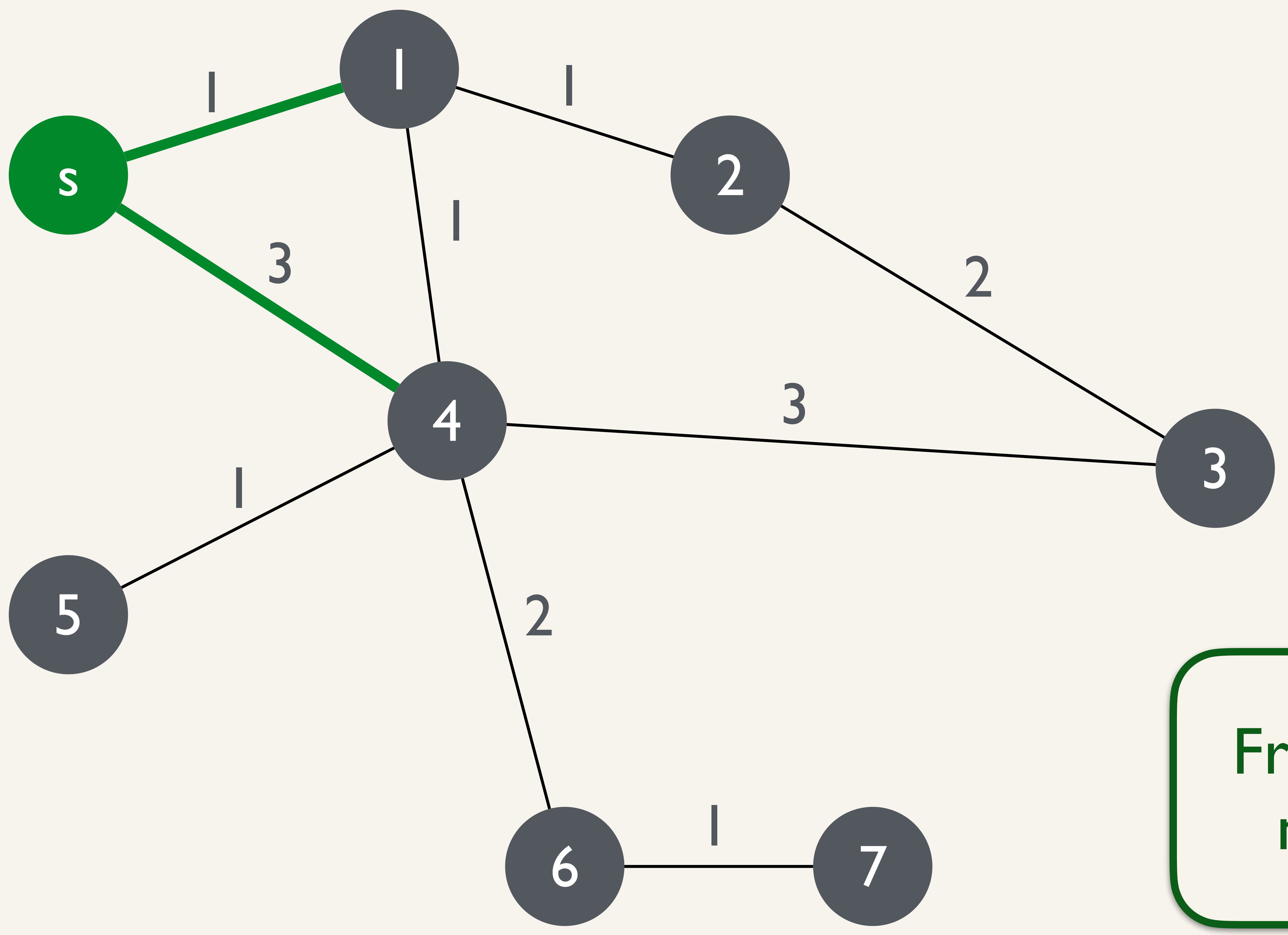
Distances: s: 0

Frontier-based approach: on each step, visit all neighbors that had their distance decrease

Weighted Breadth-First Search

Given: $G = (V, E, w)$ with *positive integer edge weights*, and a source **s**

Problem: Compute the shortest path distances from **s**



Round 1

Frontier: s

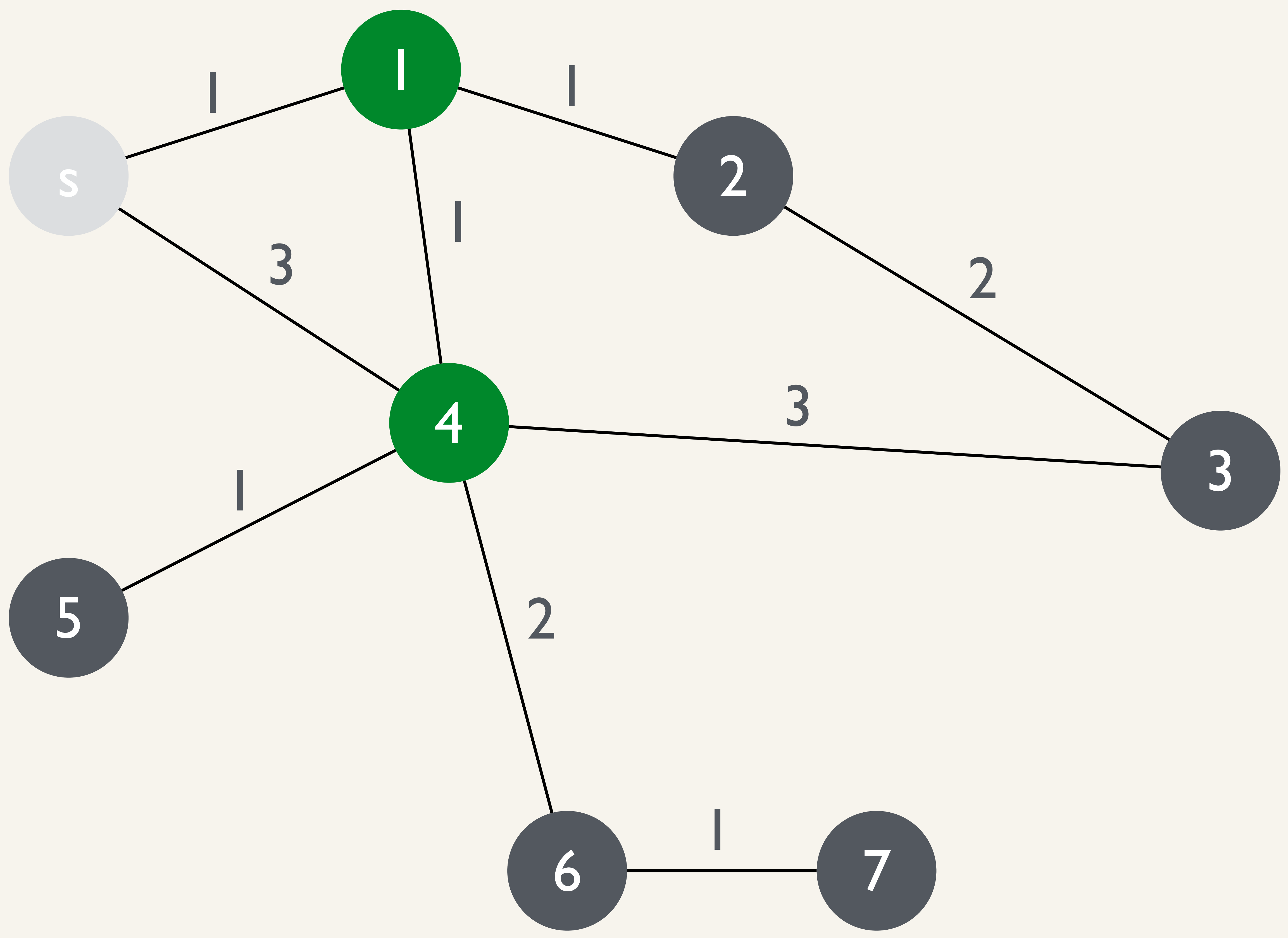
Distances: s: 0

Frontier-based approach: on each step, visit all neighbors that had their distance decrease

Weighted Breadth-First Search

Given: $G = (V, E, w)$ with *positive integer edge weights*, and a source **s**

Problem: Compute the shortest path distances from **s**



Round 2

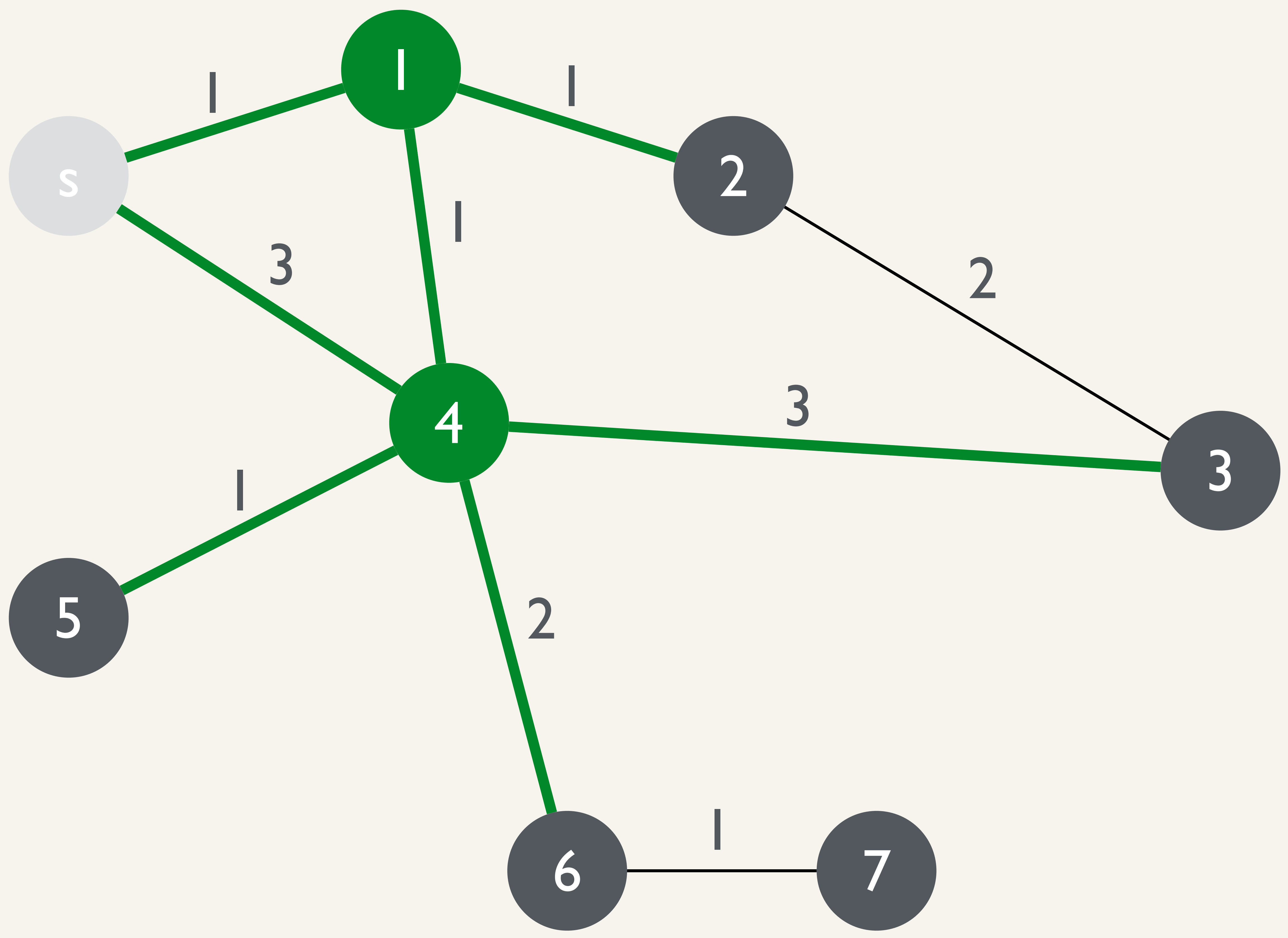
Frontier: 1 4

Distances: s: 0
1: 1
4: 3

Weighted Breadth-First Search

Given: $G = (V, E, w)$ with *positive integer edge weights*, and a source **s**

Problem: Compute the shortest path distances from **s**



Round 2

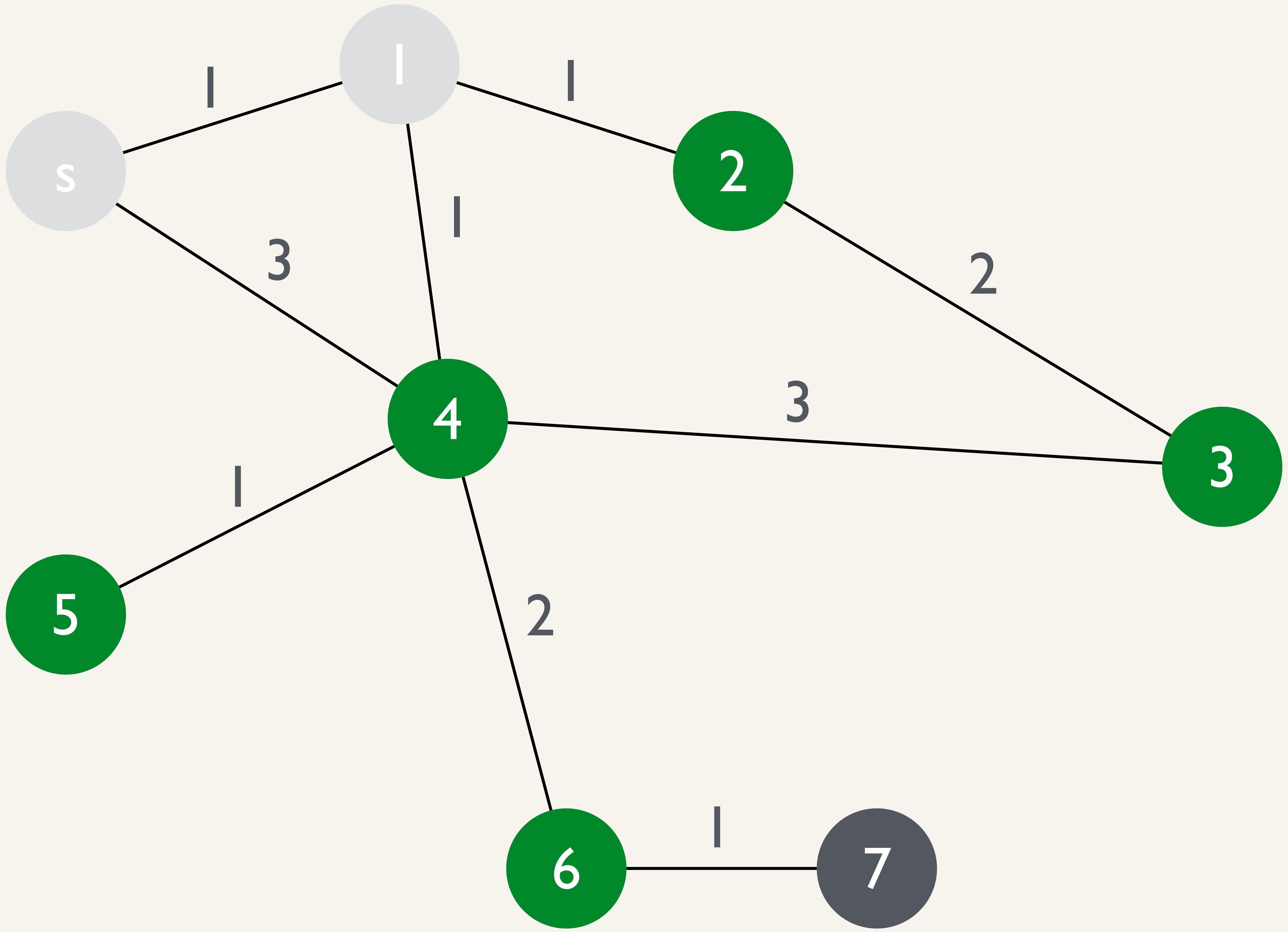
Frontier: 1 4

Distances: s: 0
1: 1
4: 3

Weighted Breadth-First Search

Given: $G = (V, E, w)$ with *positive integer edge weights*, and a source **s**

Problem: Compute the shortest path distances from **s**



Round 3

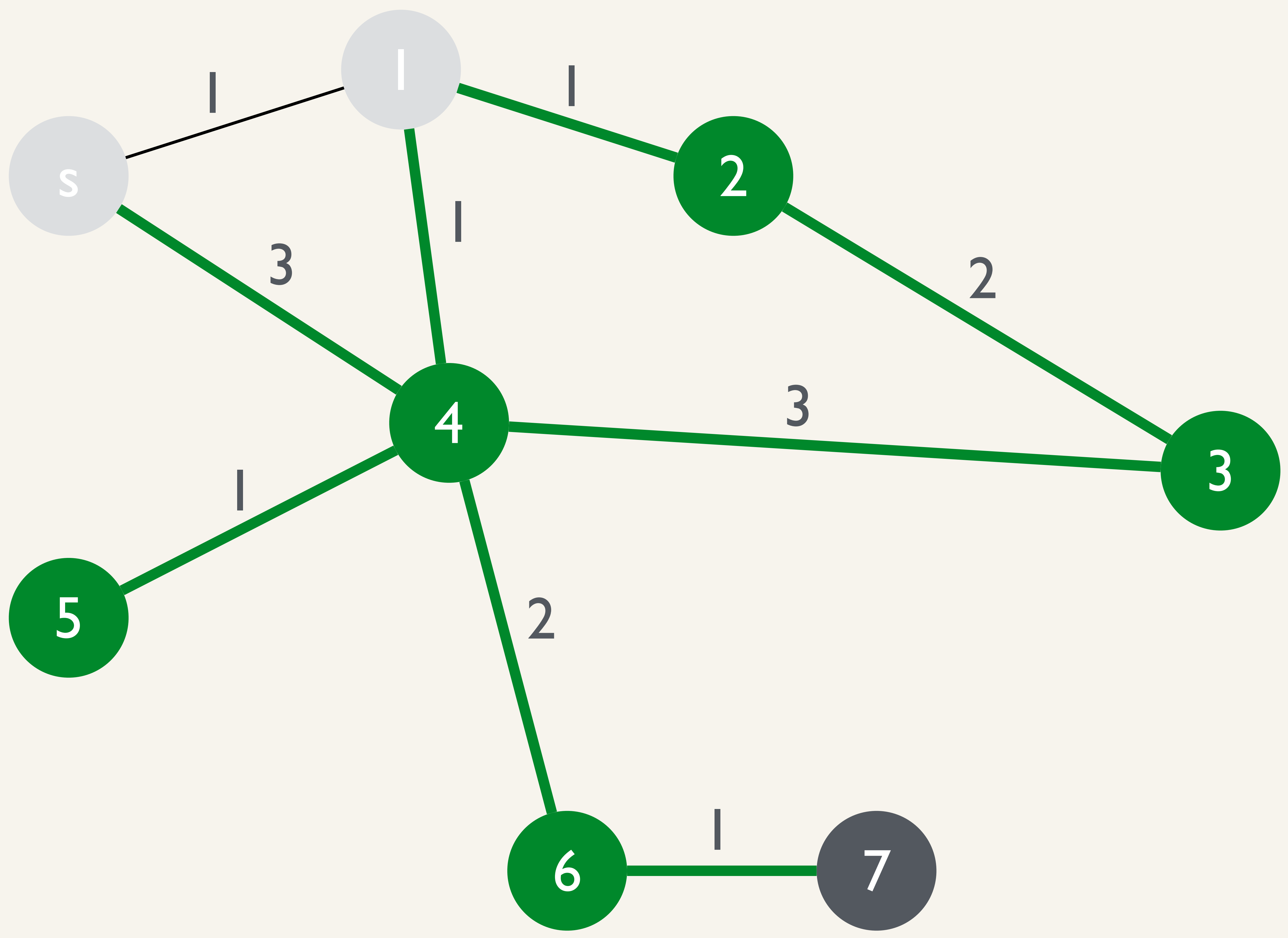
Frontier: 4 2 5 6 3

Distances: s: 0
1: 1
4: 2
6: 5

Weighted Breadth-First Search

Given: $G = (V, E, w)$ with *positive integer edge weights*, and a source **s**

Problem: Compute the shortest path distances from **s**



Round 3

Frontier: 4 2 5 6 3

Distances: s: 0
1: 1
4: 2
6: 5

Not work-efficient!

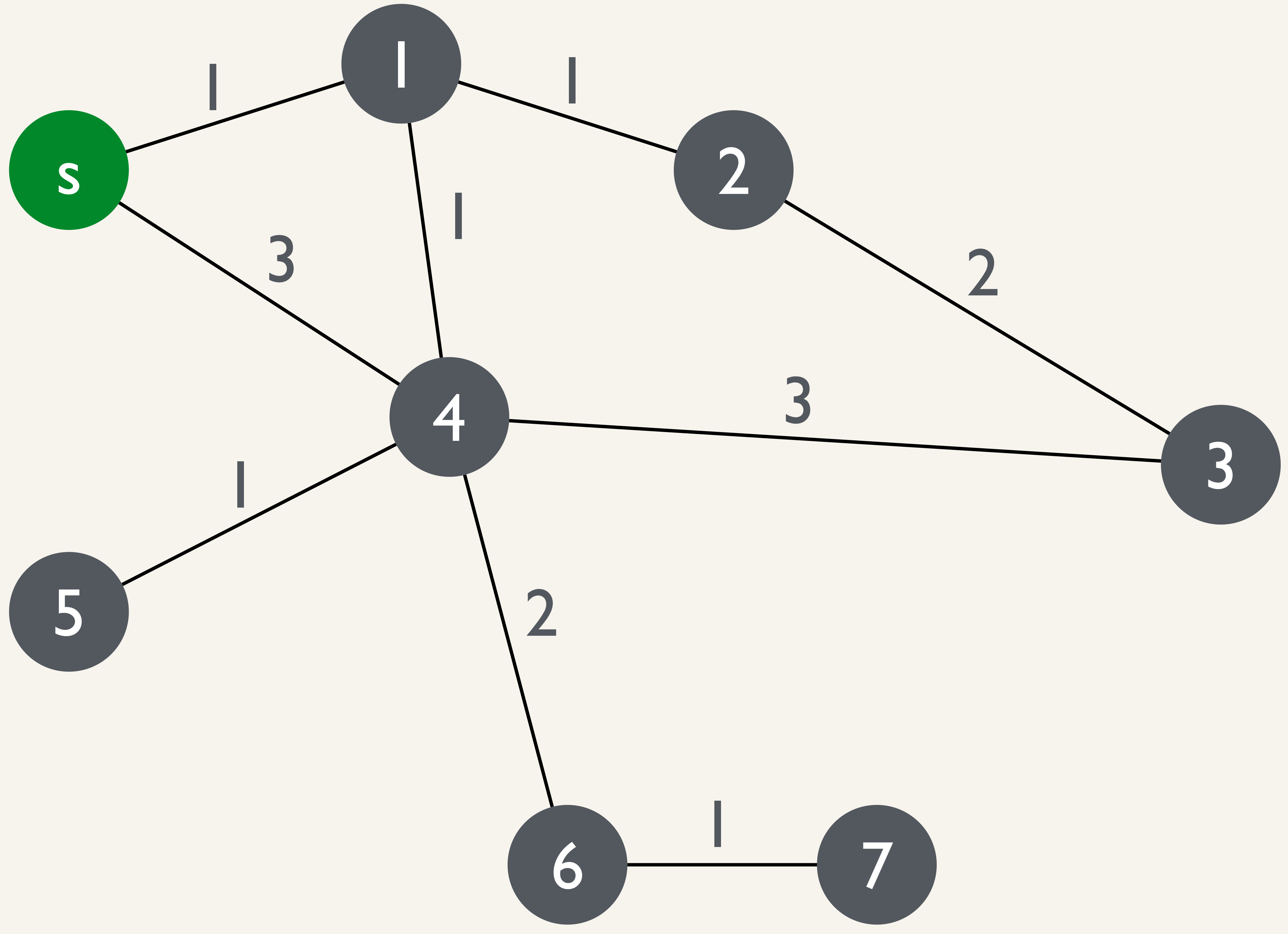
Sequential Weighted Breadth-First Search

Given: $G = (V, E, w)$ with *positive integer edge weights*, and a source **s**

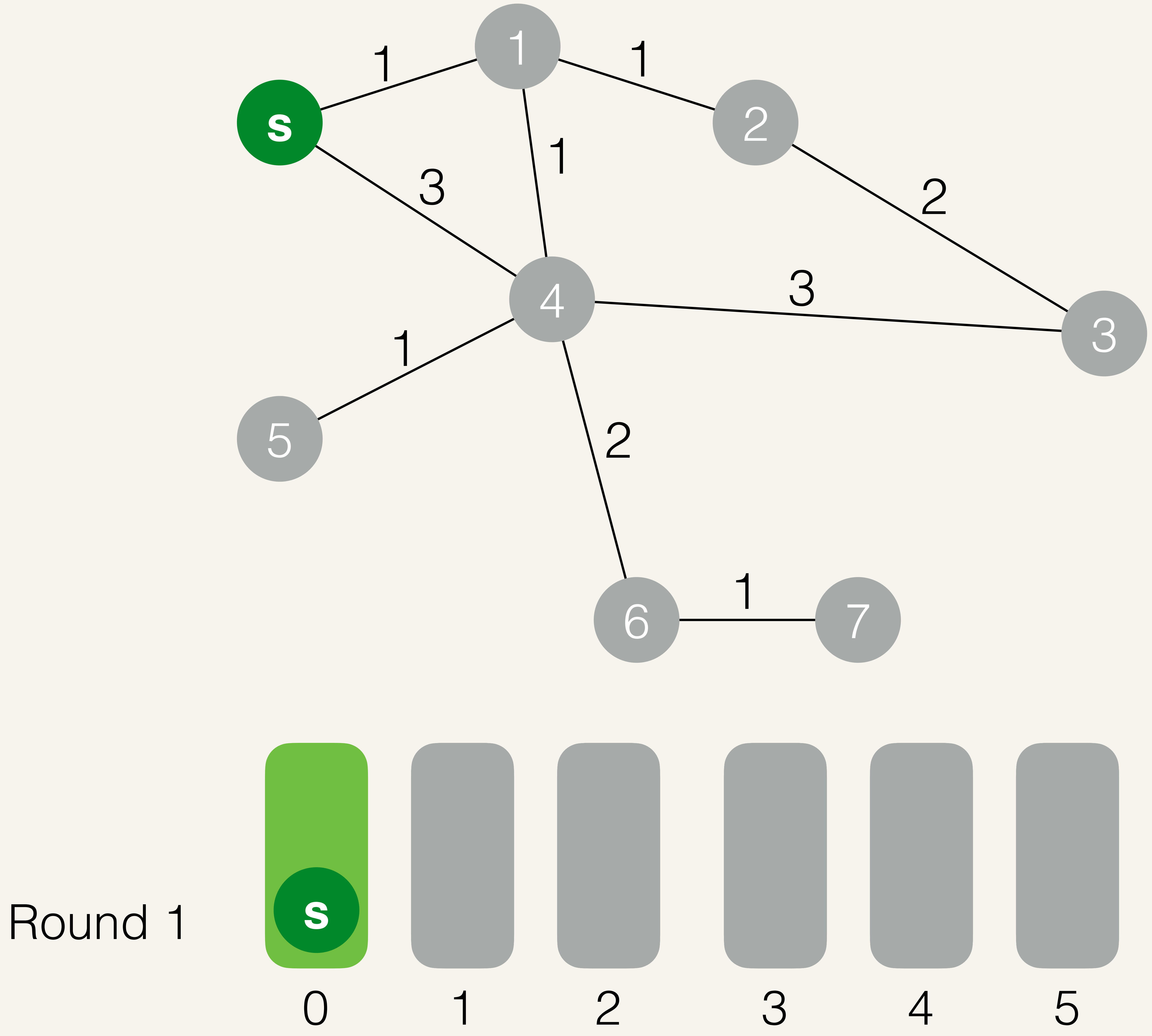
Problem: Compute the shortest path distances from **s**

Idea:

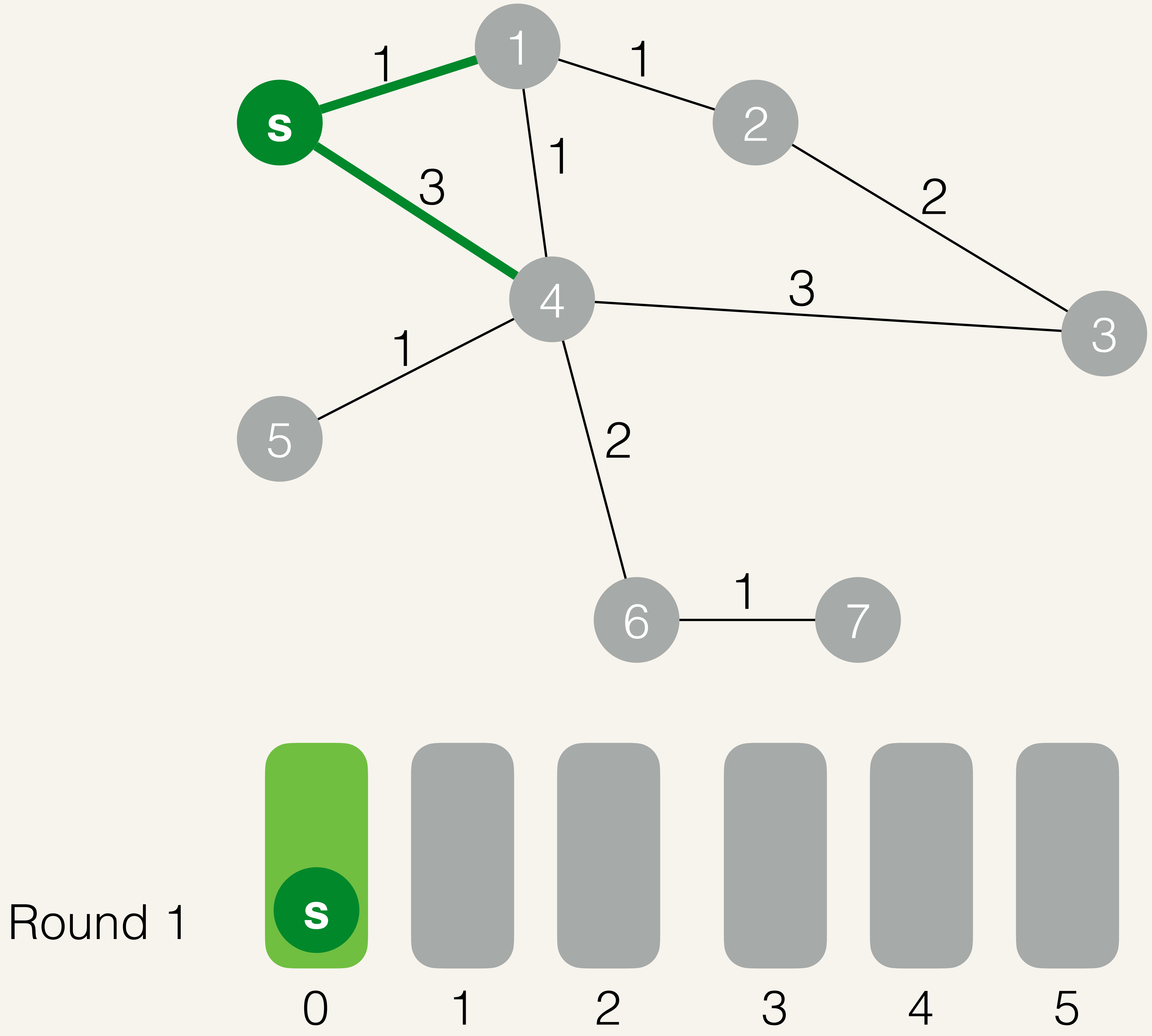
- Run Dijkstra's algorithm, but use buckets instead of a PQ
- Represent buckets using dynamic arrays
- Runs in $O(m + r_{\text{src}})$ work



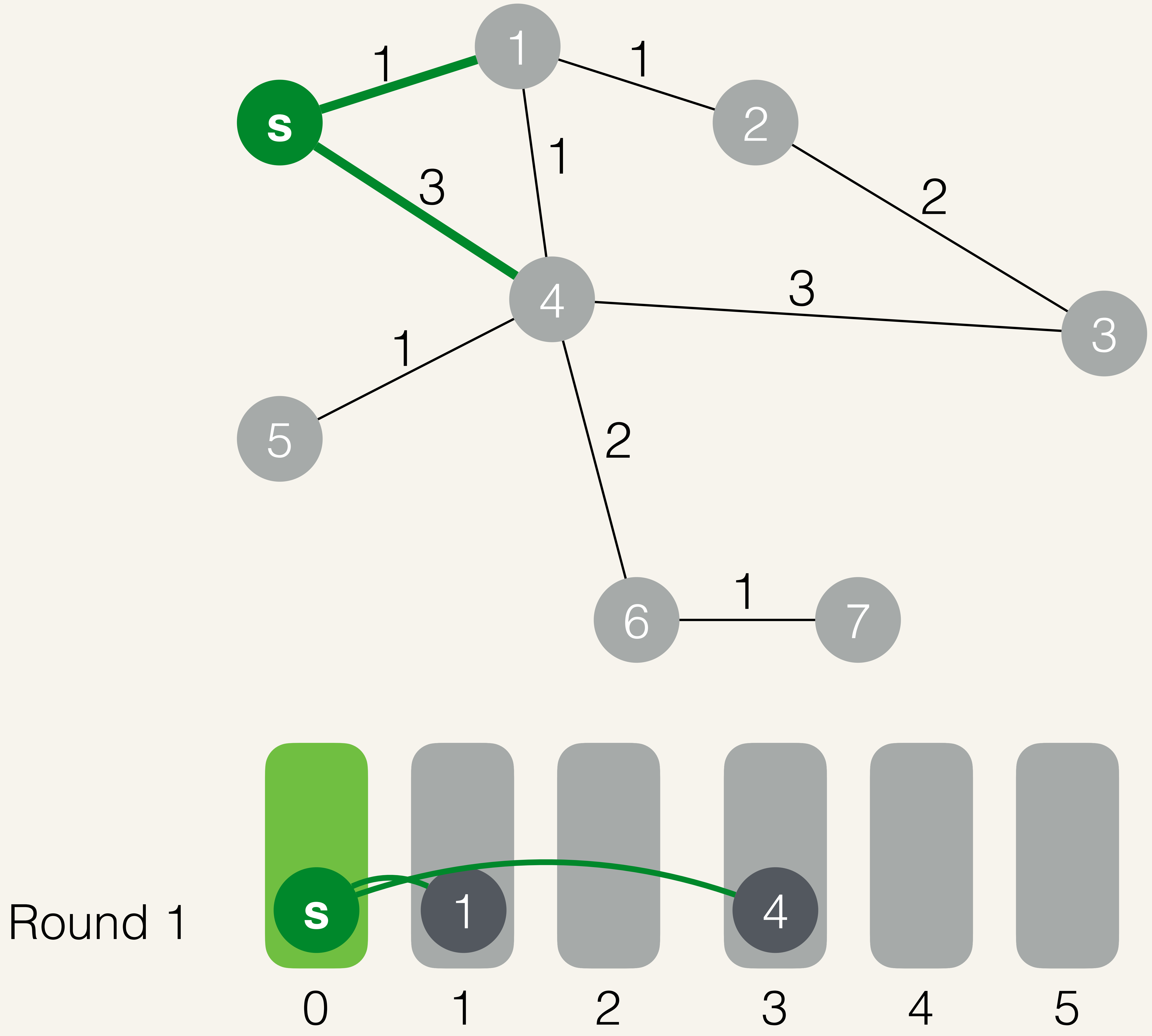
Sequential Weighted Breadth-First Search



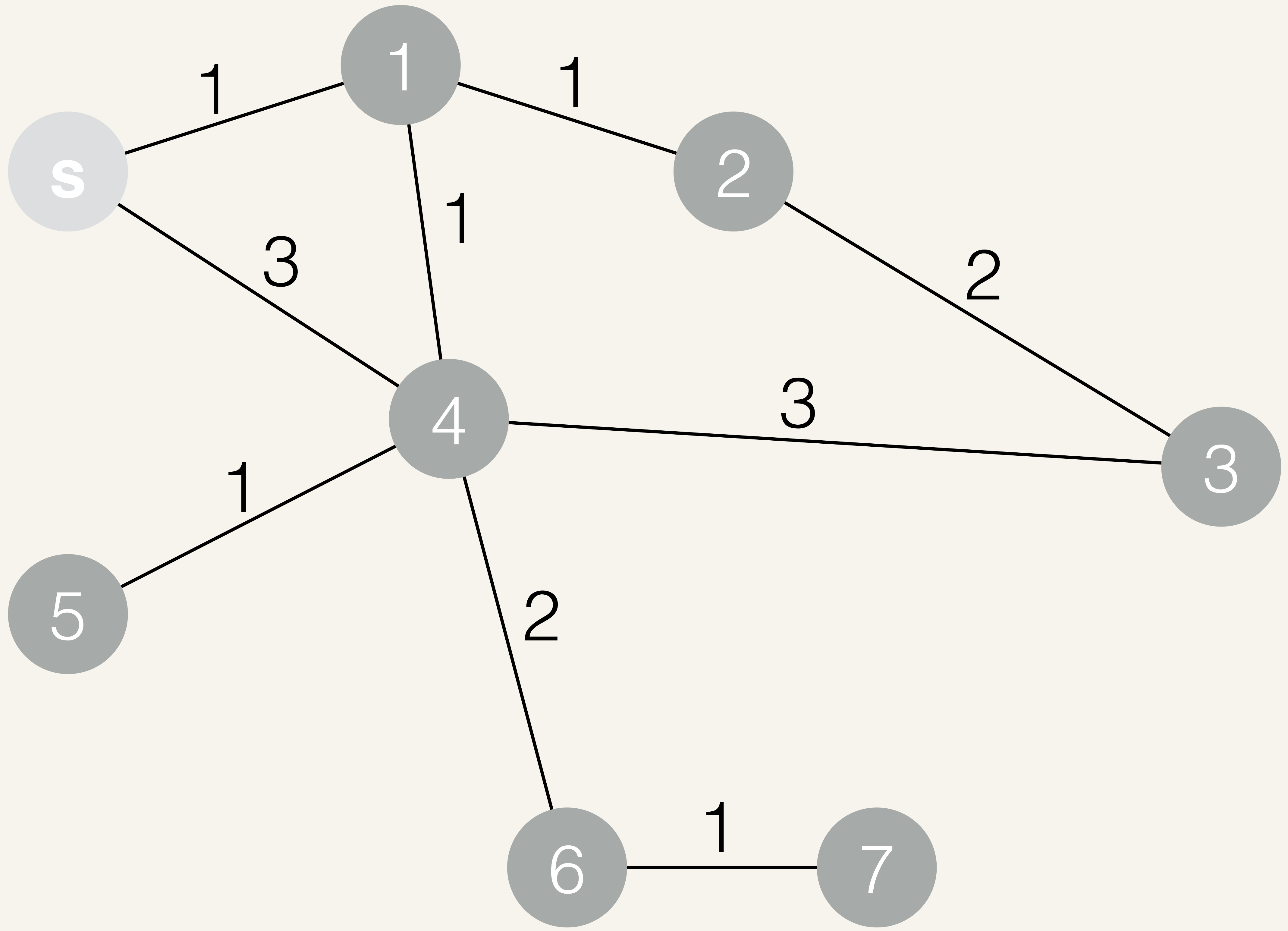
Sequential Weighted Breadth-First Search

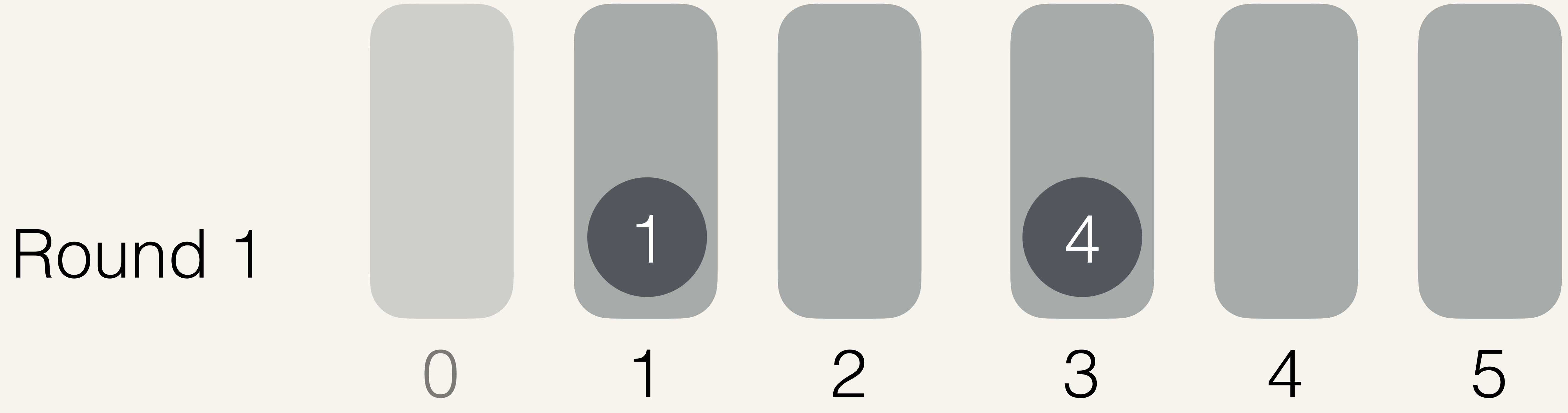


Sequential Weighted Breadth-First Search

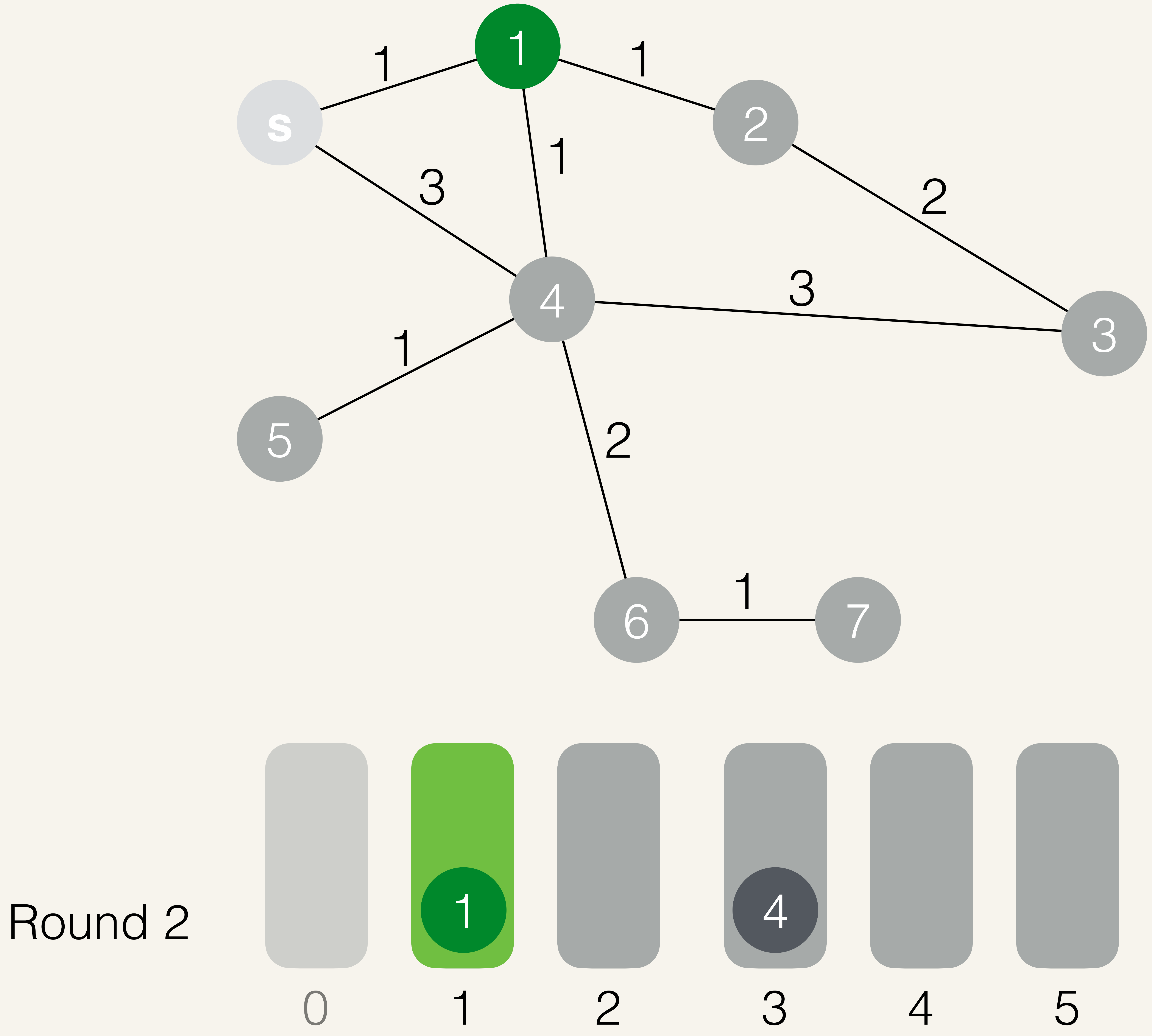


Sequential Weighted Breadth-First Search

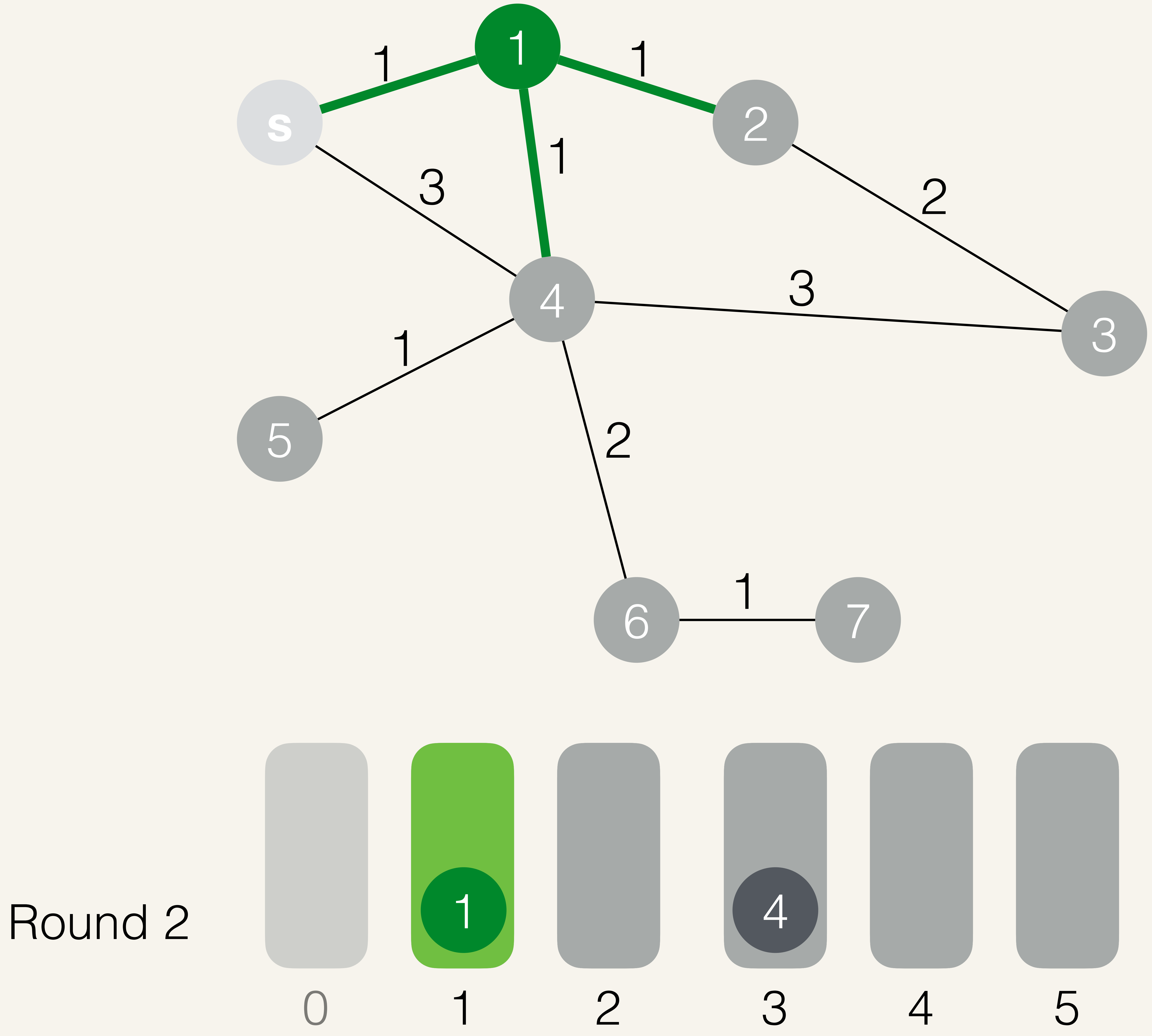




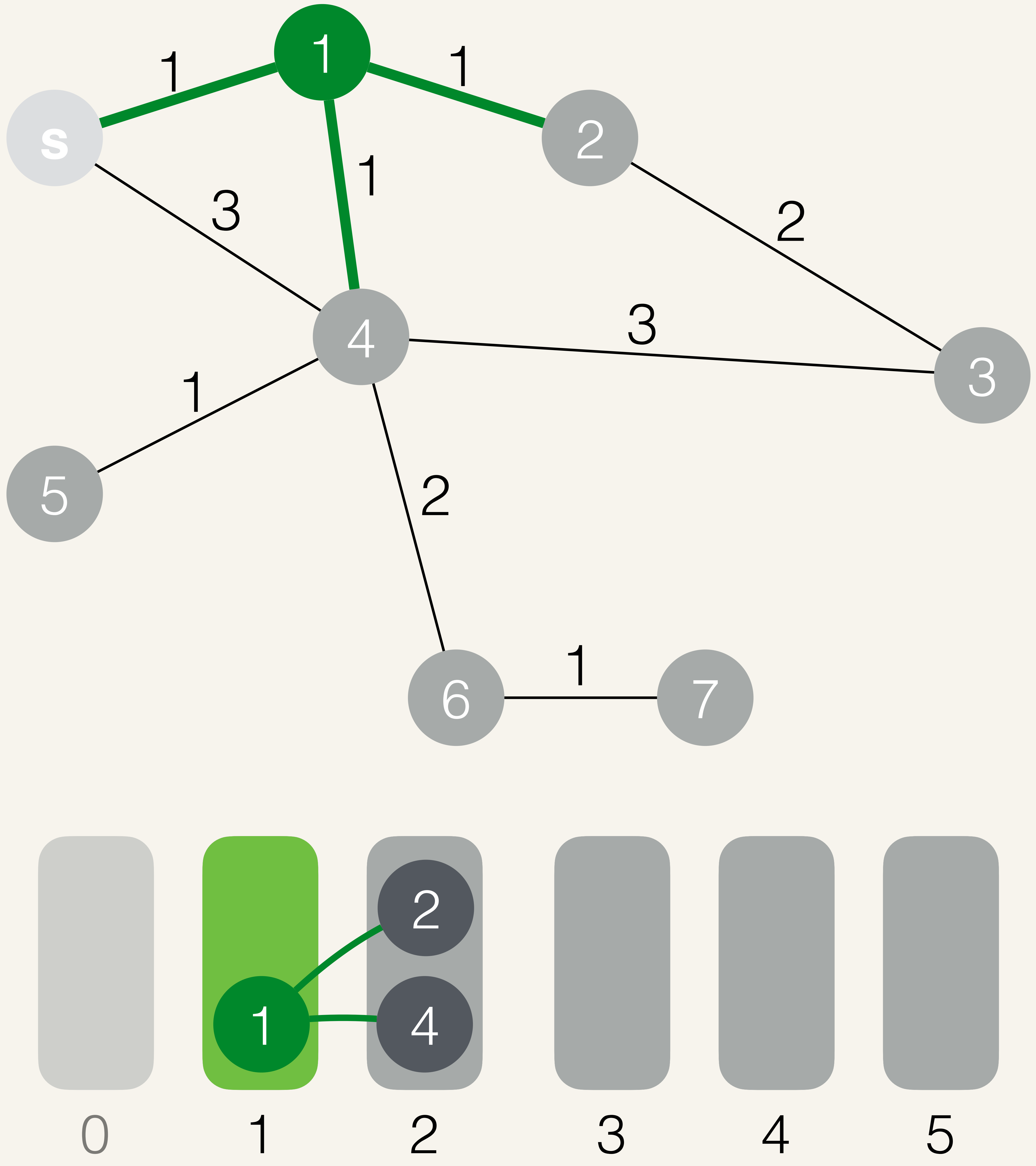
Sequential Weighted Breadth-First Search



Sequential Weighted Breadth-First Search

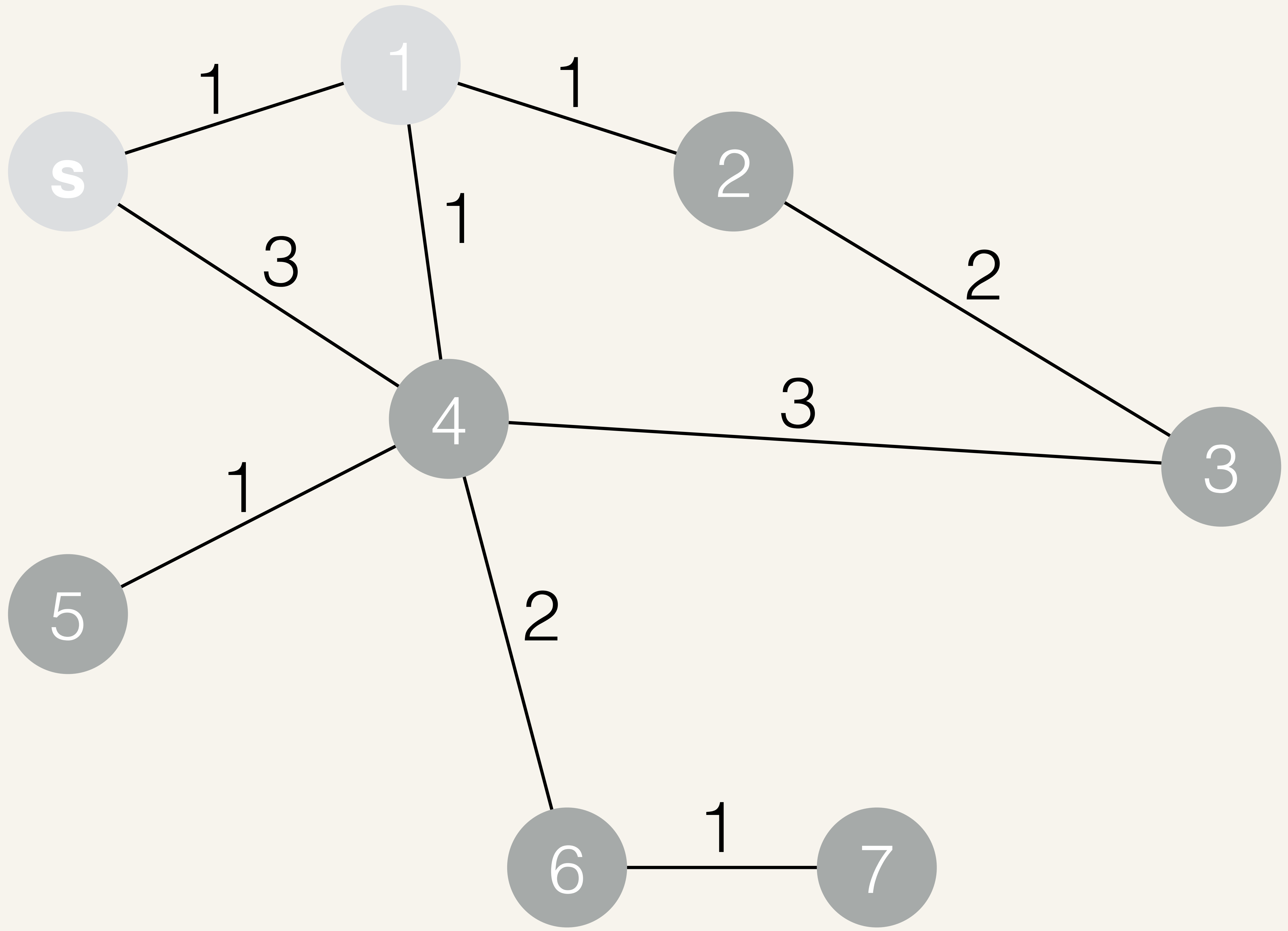


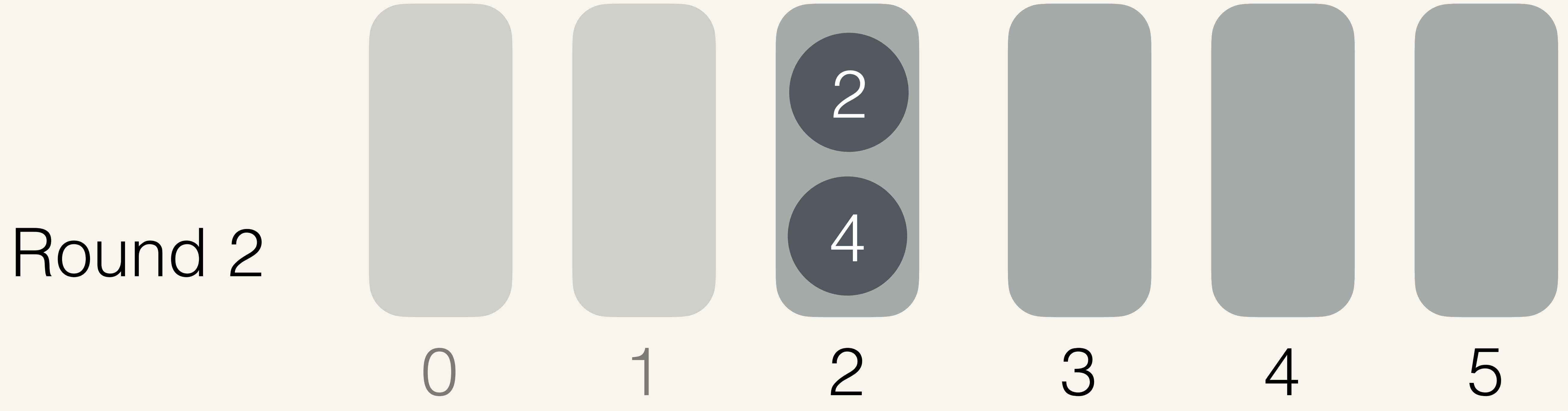
Sequential Weighted Breadth-First Search



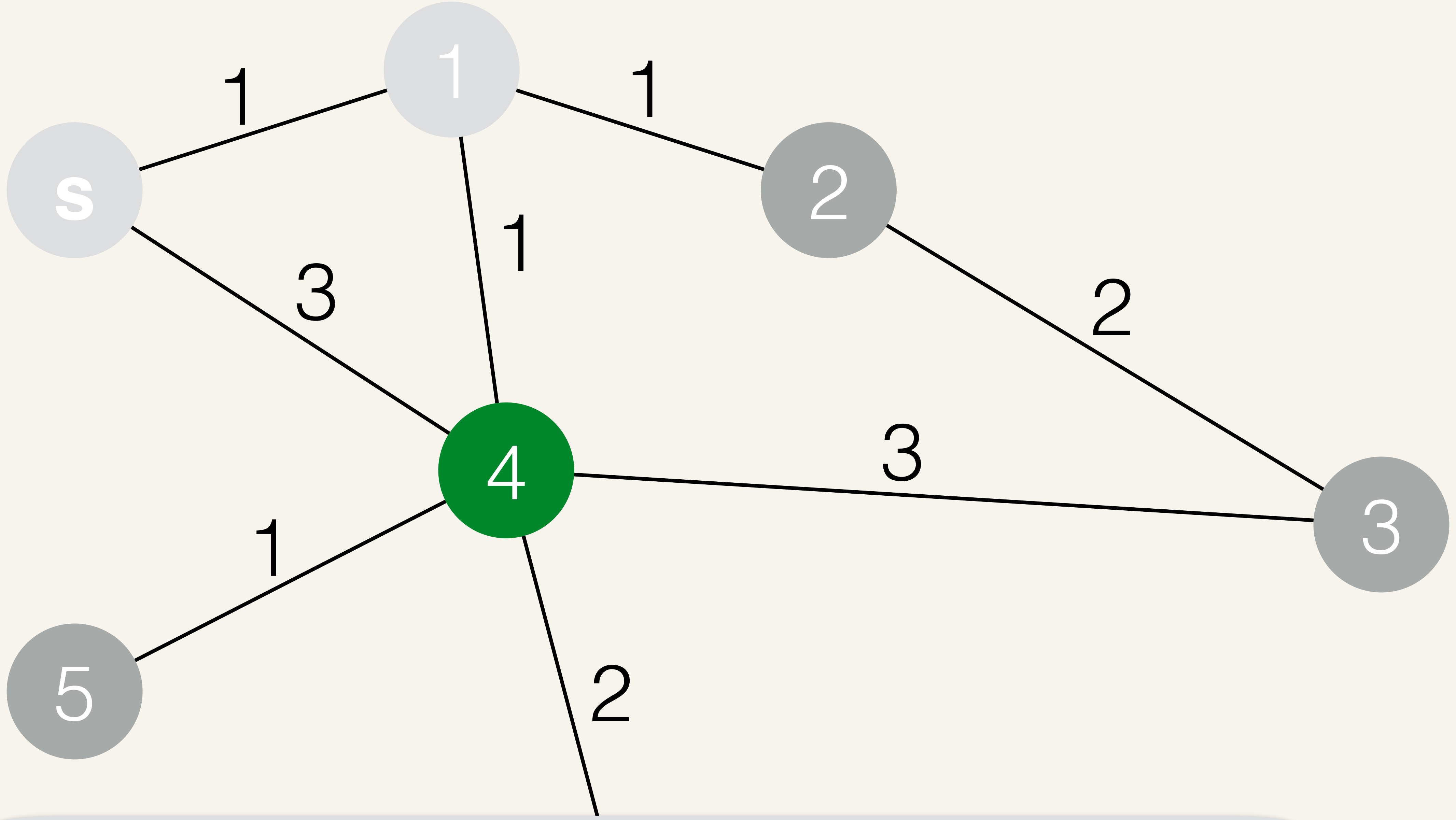
Round 2

Sequential Weighted Breadth-First Search

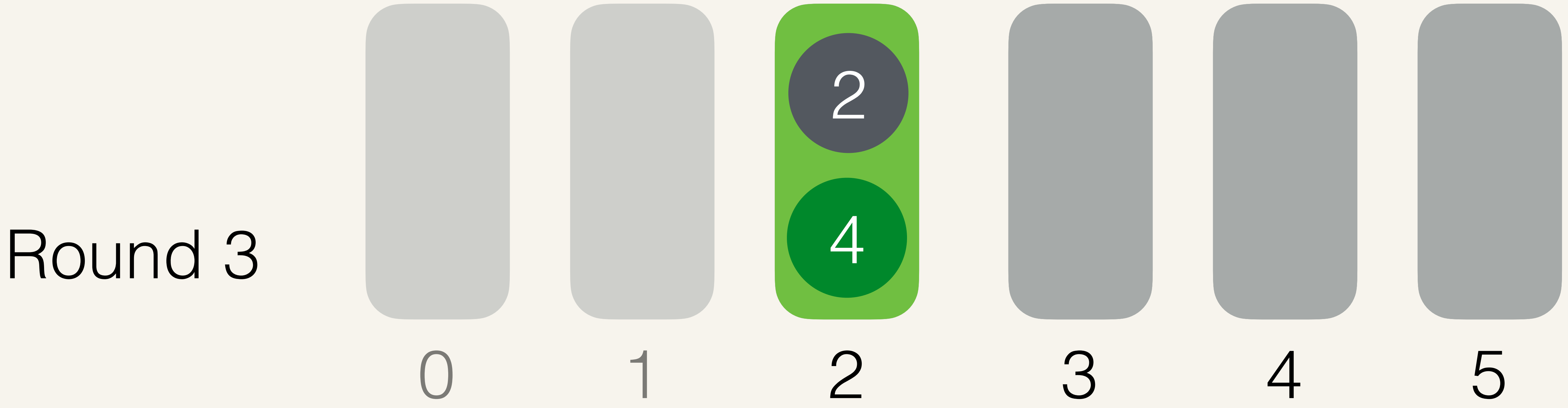




Sequential Weighted Breadth-First Search



Runs in $O(m + r_{\text{src}})$ work



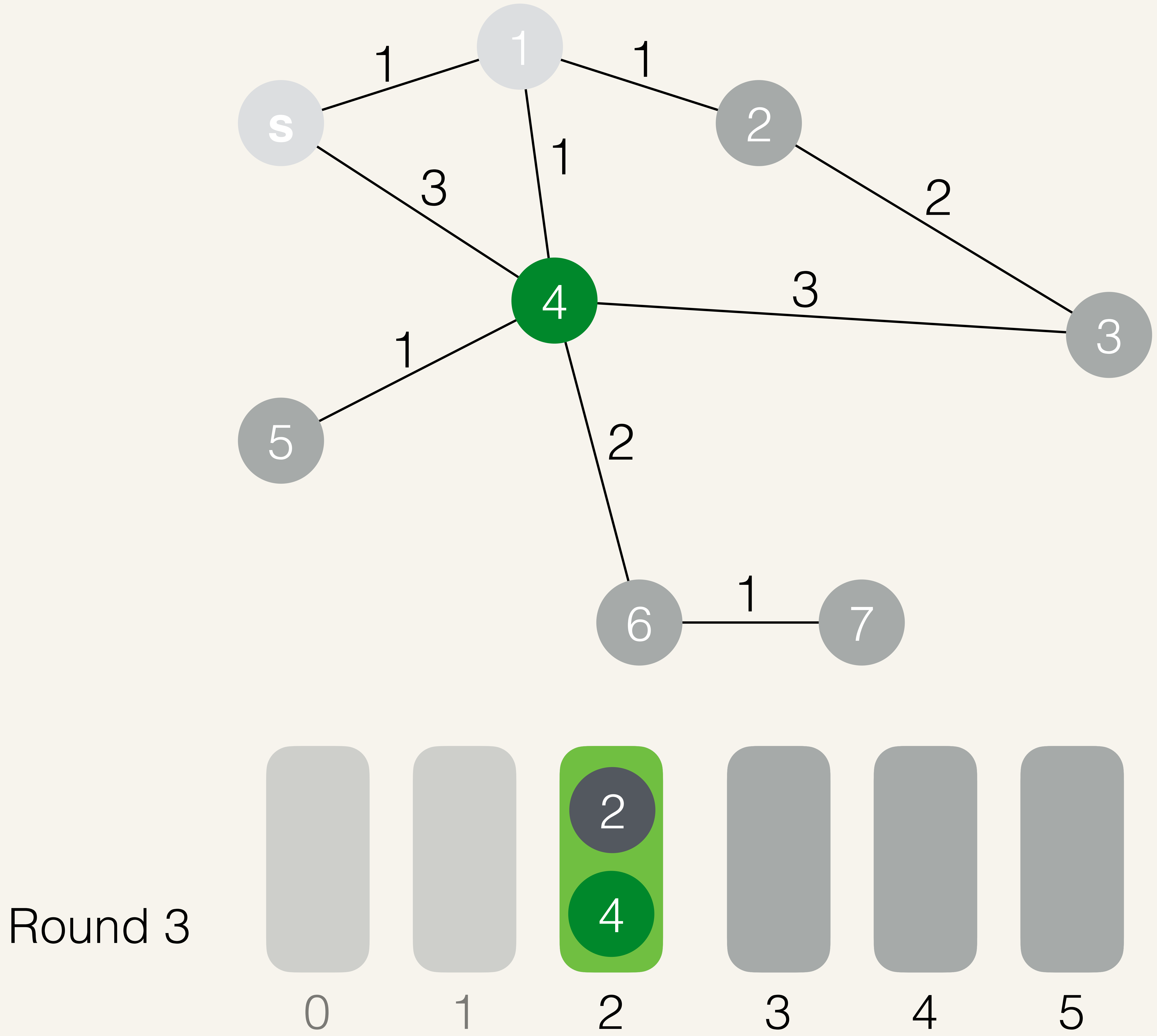
Bucketing

The algorithm uses buckets to *organize work* for future iterations

This algorithms is parallelizable

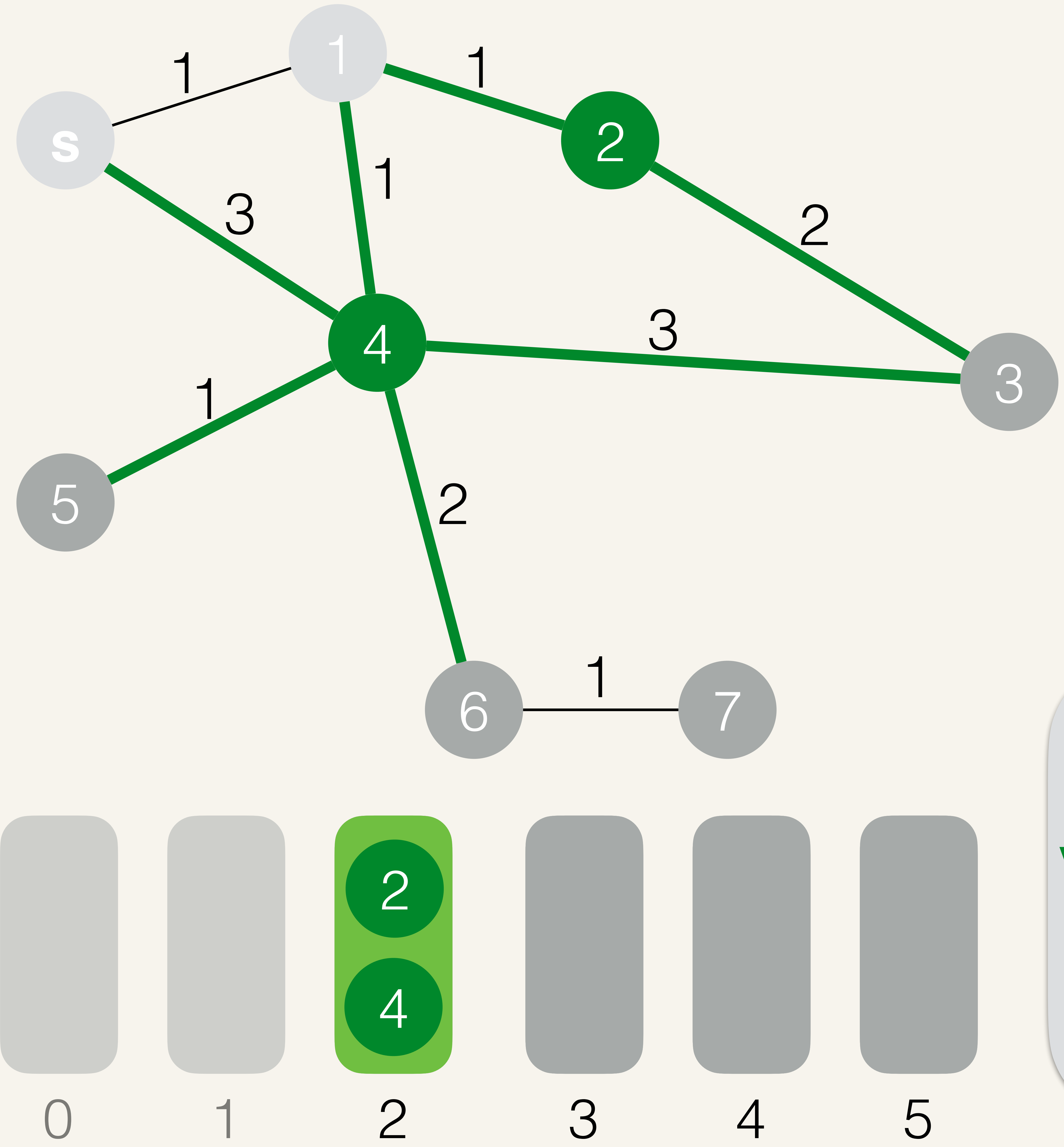
- In each step:
  1. Process all vertices in the next bucket in parallel
  2. Update buckets of neighbors in parallel

Sequential Weighted Breadth-First Search



Sequential: process vertices one by one

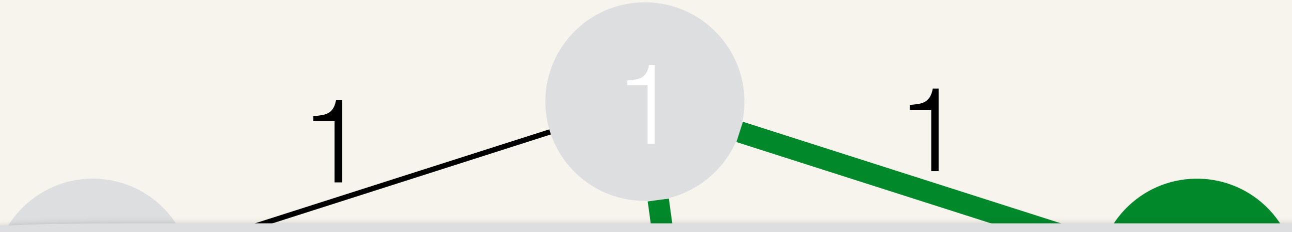
Parallel Weighted Breadth-First Search



Round 3

(1) Process vertices in the same bucket in parallel

Parallel Weighted Breadth-First Search



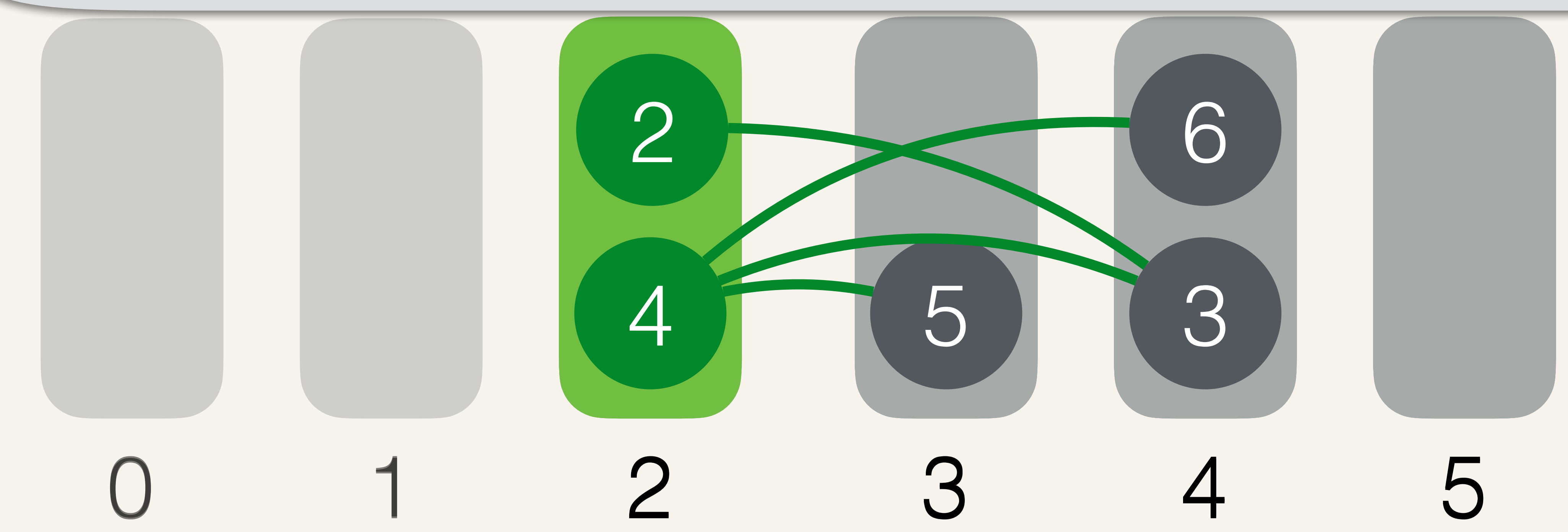
Resulting algorithm performs:

$O(m + r_{\text{src}})$ **work**

$O(r_{\text{src}} \log n)$ **depth**

(assuming efficient bucketing)

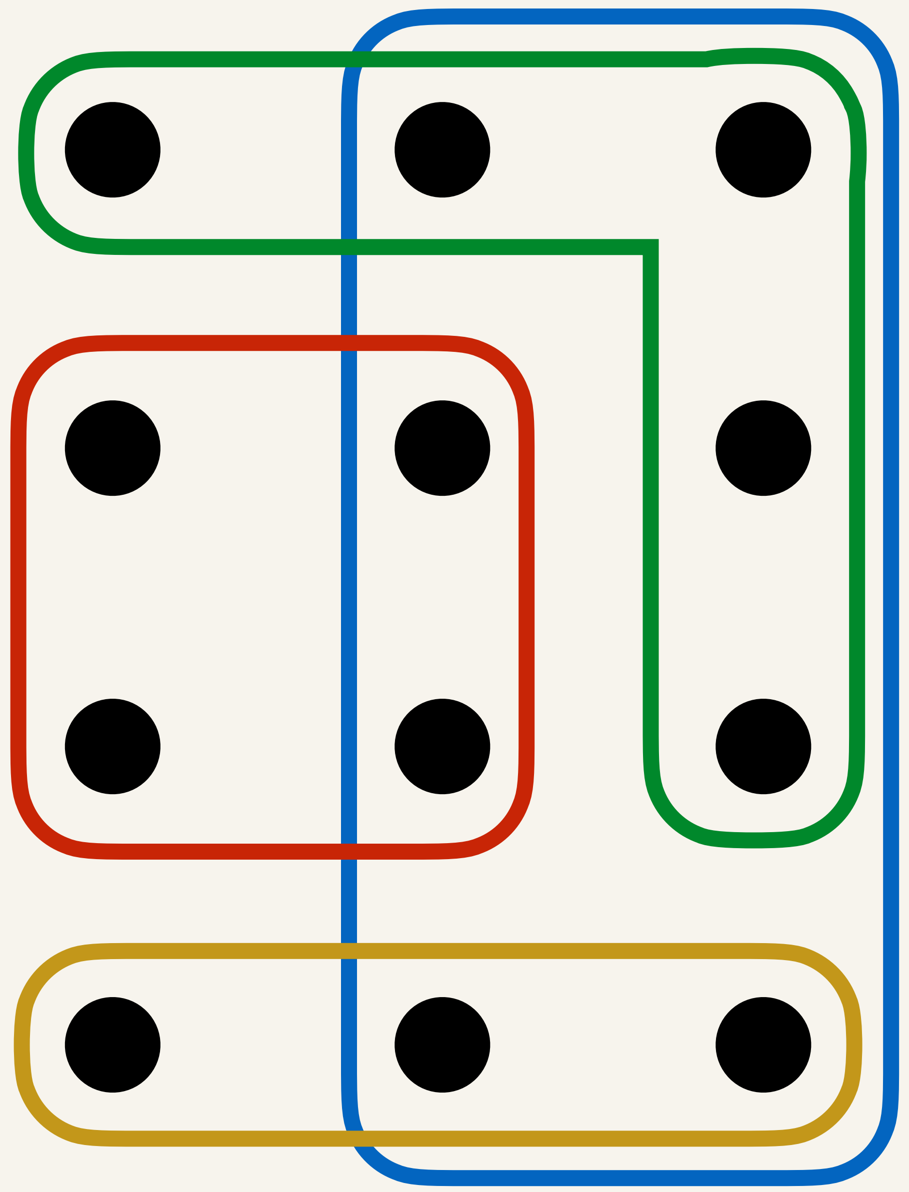
Round 3



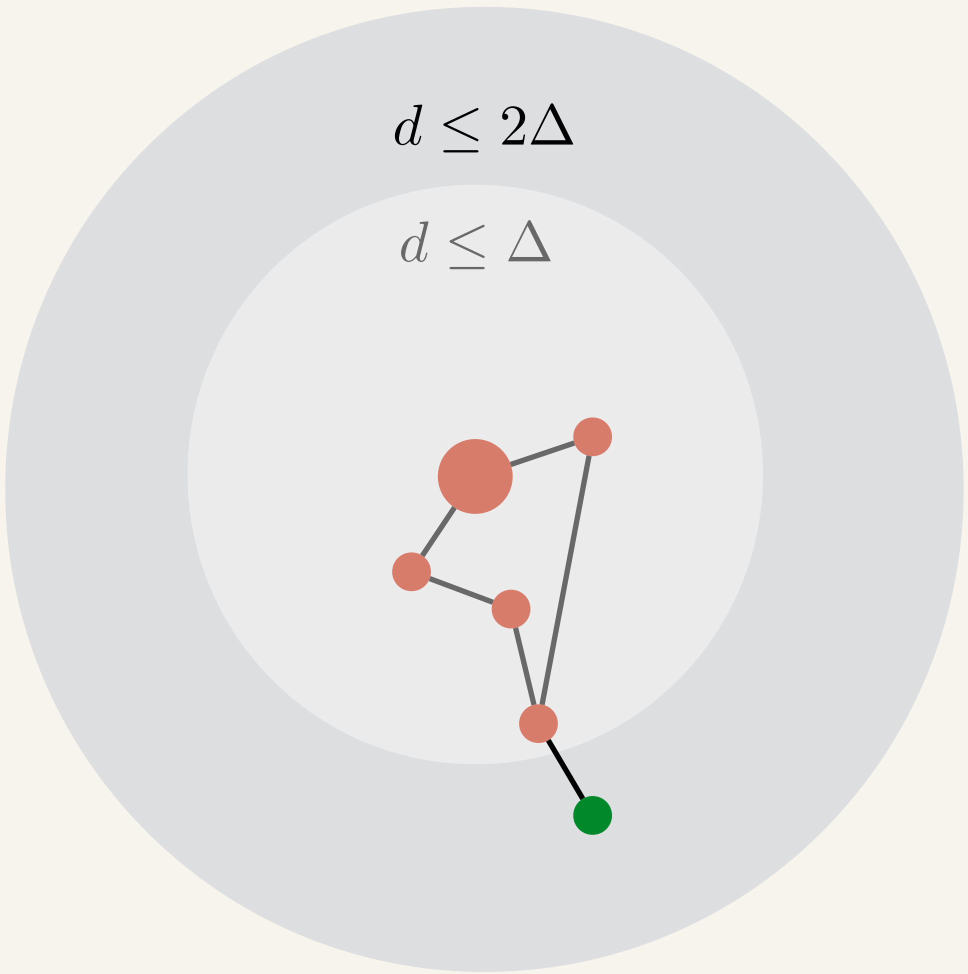
(2) Insert neighbors into buckets in parallel

Parallel Bucketing

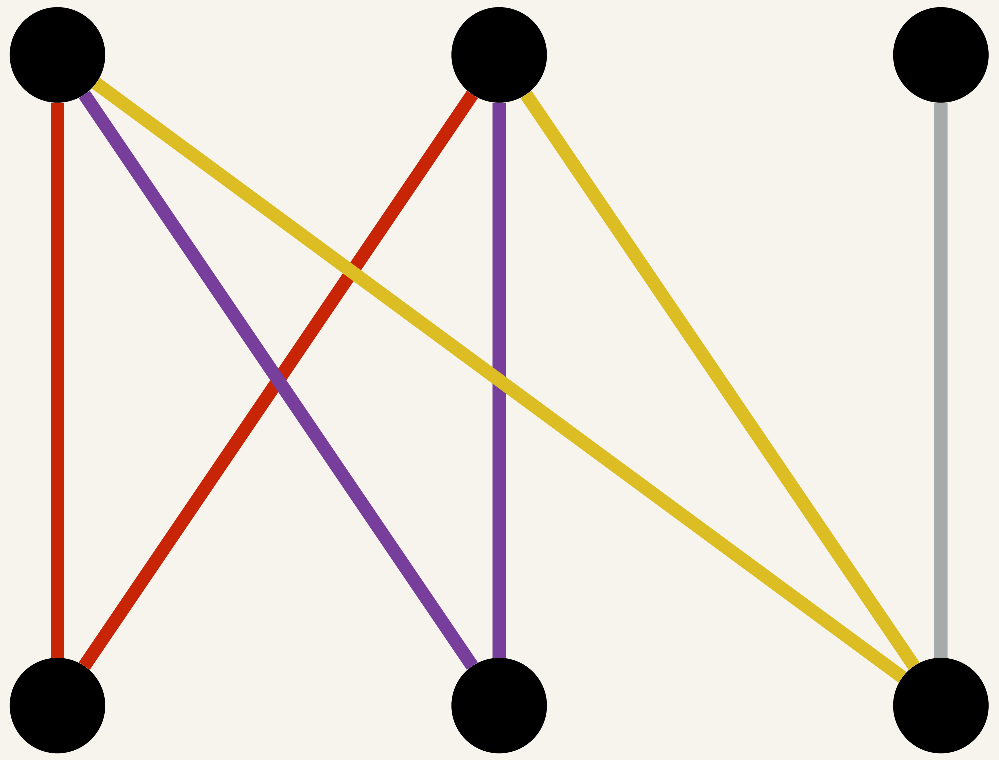
Bucketing is useful for more than just wBFS



Parallel Approximate Set Cover
[BPT'12]



Parallel Shortest Paths
[MB'03]



Parallel k-Tip Decomposition
[SS'20]

Parallel Bucketing

Bucketing is useful for more than just wBFS

Goals

- Simplify expressing algorithms using an interface
- Theoretically efficient, reusable implementation

Challenges

1. Multiple vertices insert into the same bucket in parallel
2. Possible to make work-efficient parallel implementations?

Julienne: Results

Shared memory framework for *bucketing-based algorithms*

Extend Ligra with an interface for bucketing

- Theoretical bounds for primitives
- Fast implementations of primitives

Can implement a bucketing algorithm with

- n vertices
- T total buckets
- U updates

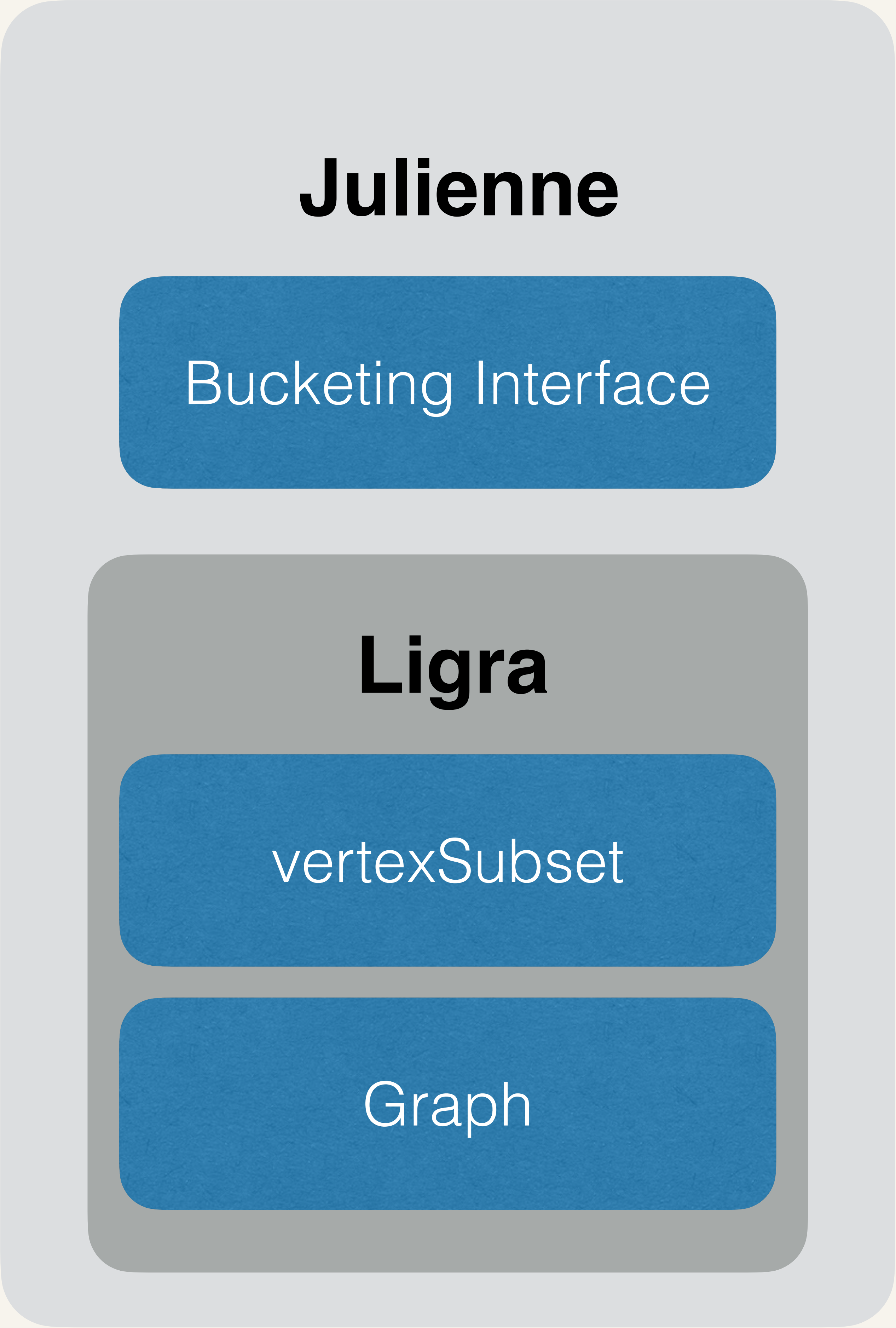
over K Update calls, and L calls to NextBucket

$O(n + T + U)$ expected work and

$O((K + L)\log n)$ depth w.h.p.

Bucketing implementation is work-efficient

Bucketing Interface



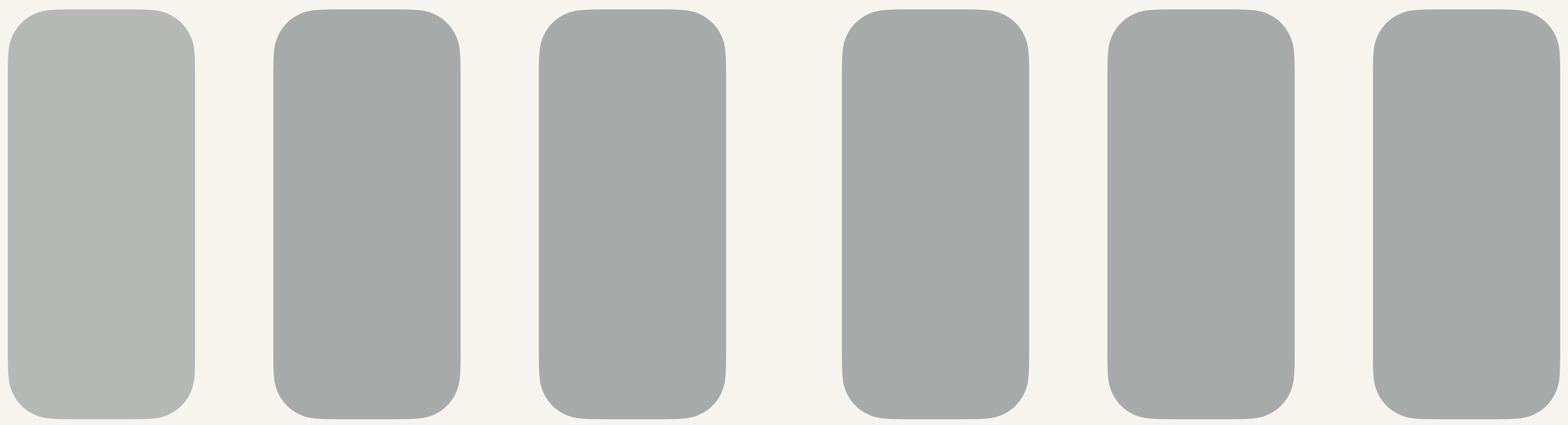
Bucketing Interface:

(1) Create bucket structure

(2) Get the next bucket (vertexSubset)

(3) Update buckets of a subset of identifiers

Bucketing Interface



MakeBuckets : buckets

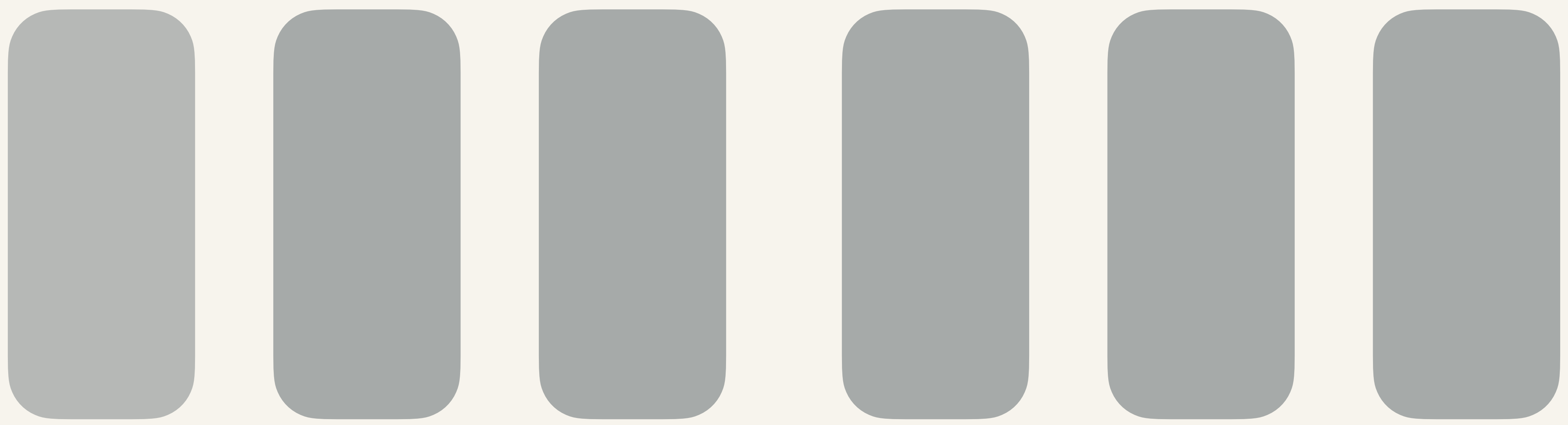
n : int

D : identifier $\rightarrow$ bucket_id

O : bucket_order

Initialize bucket structure

Bucketing Interface

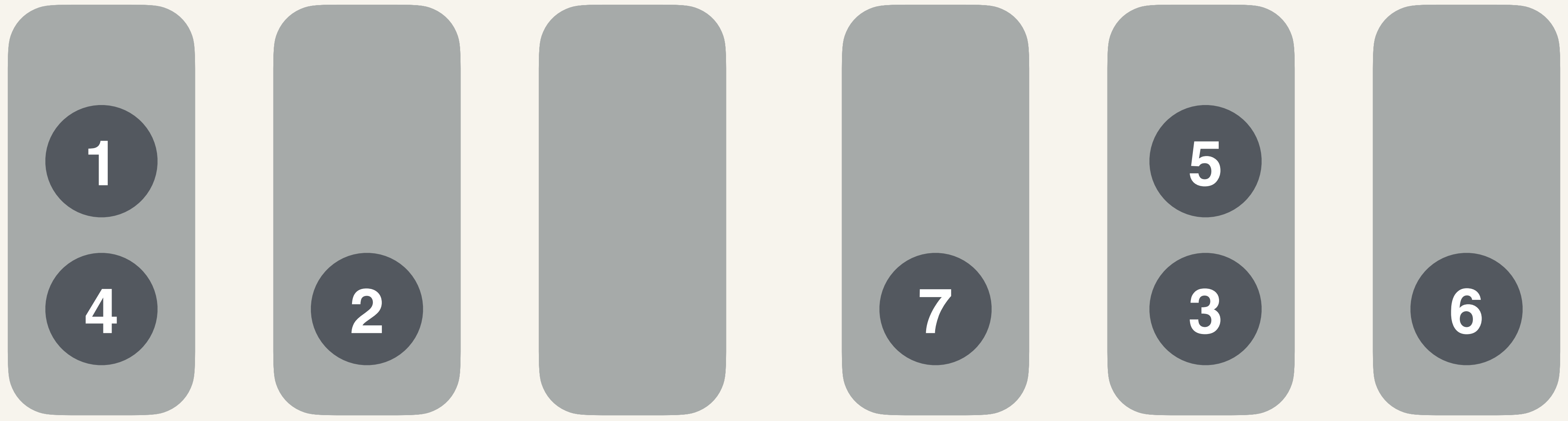


$$D(1) = 0, D(2) = 1, D(3) = 4, \ldots$$

MakeBuckets : buckets
n : int
D : identifier $\rightarrow$ bucket_id
O : bucket_order

Initialize bucket structure

Bucketing Interface

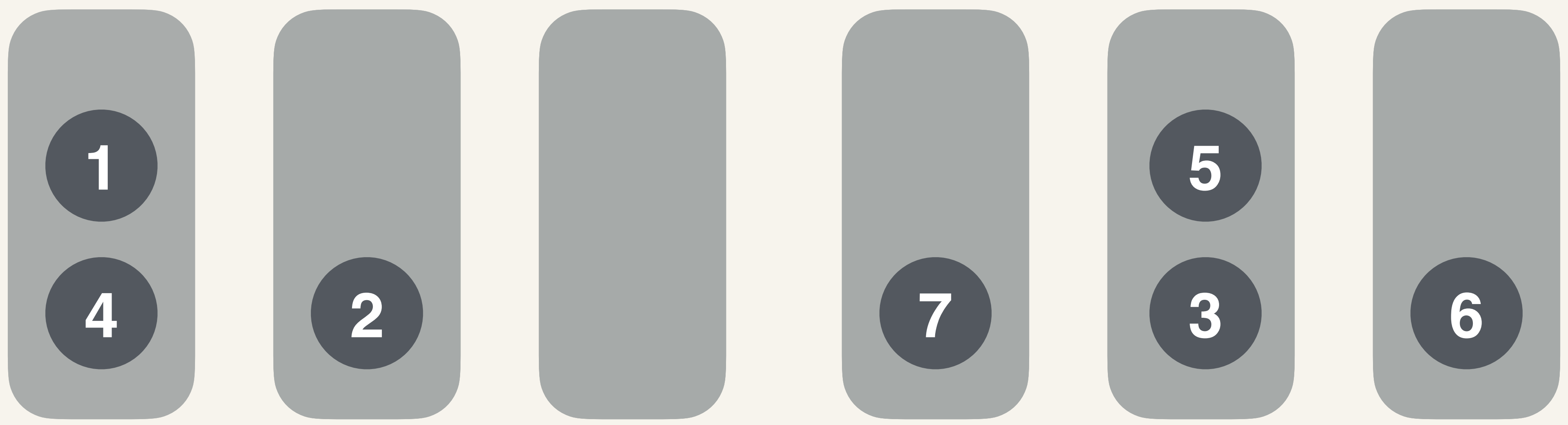


$$D(1) = 0, D(2) = 1, D(3) = 4, \ldots$$

MakeBuckets : buckets
n : int
D : identifier $\rightarrow$ bucket_id
O : bucket_order

Initialize bucket structure

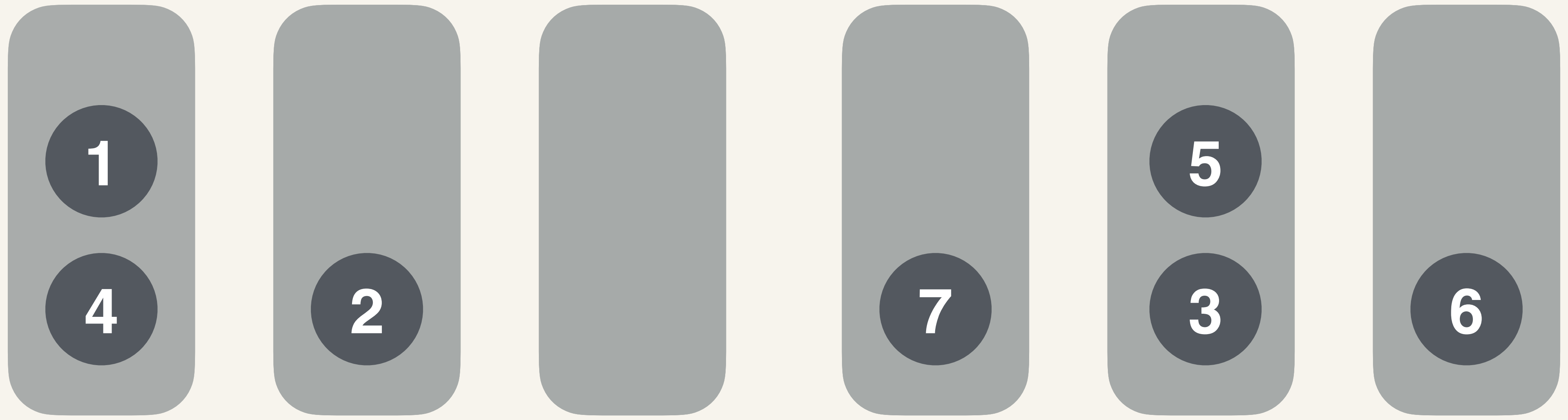
Bucketing Interface



NextBucket : bucket

Extract identifiers in the next non-empty bucket

Bucketing Interface

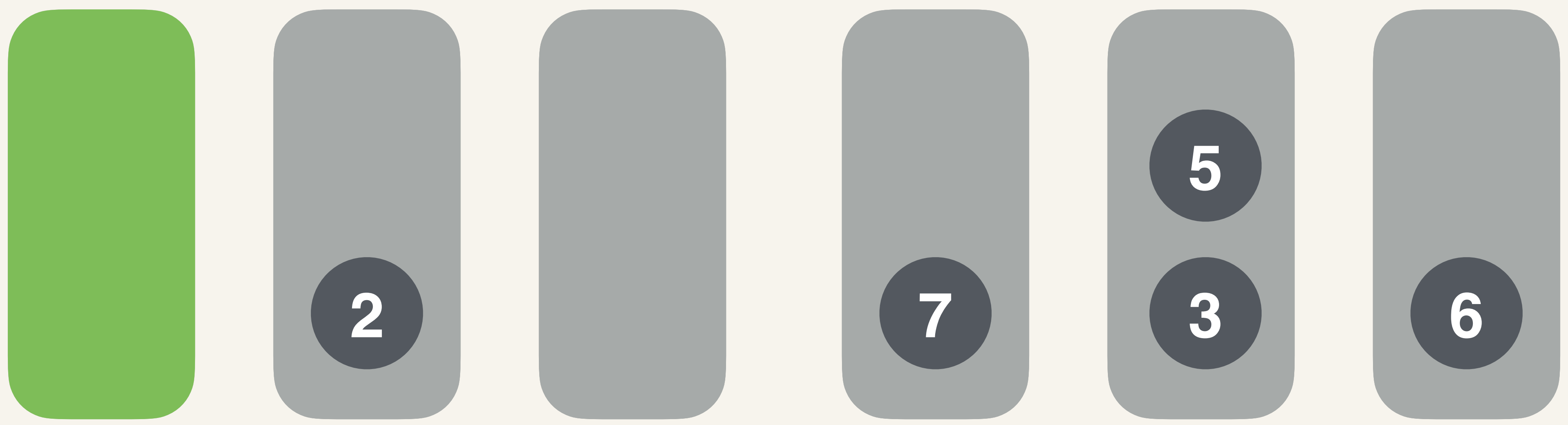


Order: increasing

NextBucket : bucket

Extract identifiers in the next non-empty bucket

Bucketing Interface

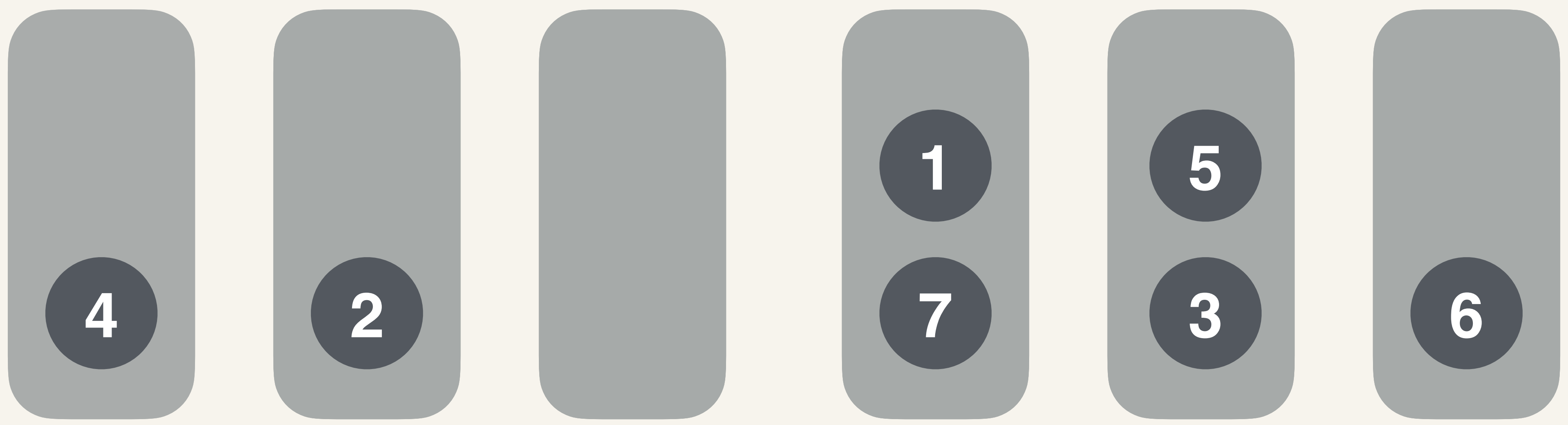


Order: increasing

NextBucket : bucket

Extract identifiers in the next non-empty bucket

Bucketing Interface



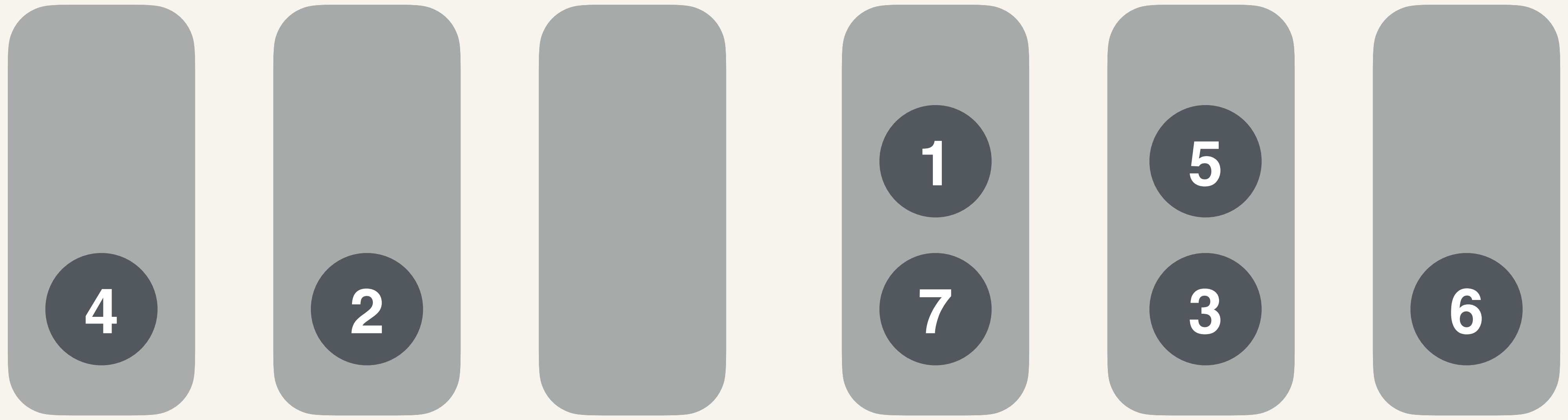
UpdateBuckets

k : int

F : int $\rightarrow$ (identifier, bucket_dest)

Update buckets for k identifiers

Bucketing Interface

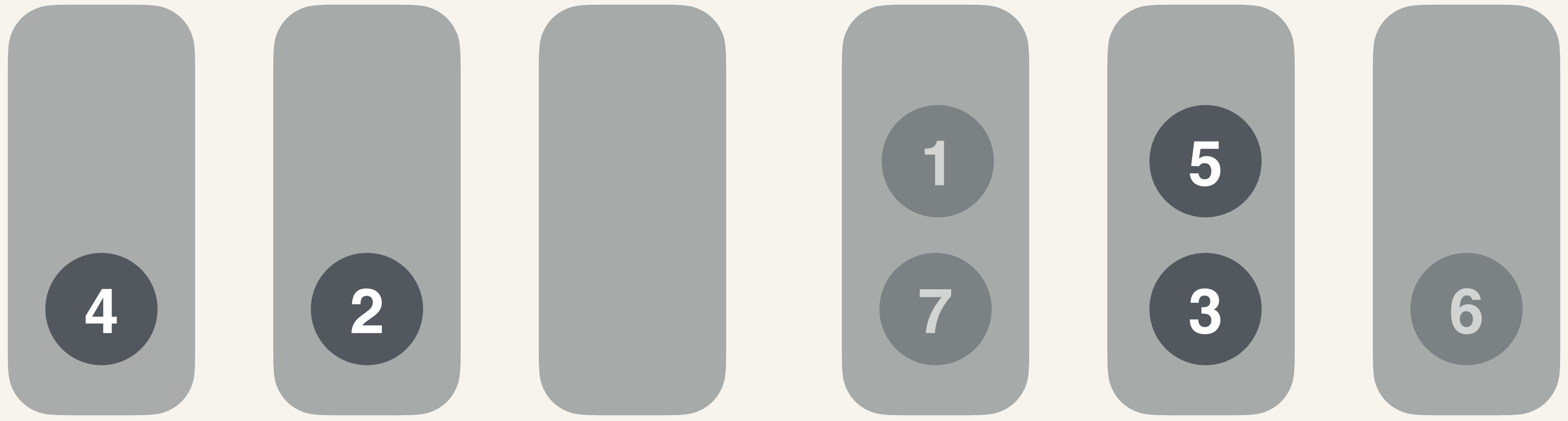


[(1,1), (7,2), (6,2)]

UpdateBuckets
k : int
F : int $\rightarrow$ (identifier, bucket_dest)

Update buckets for k identifiers

Bucketing Interface

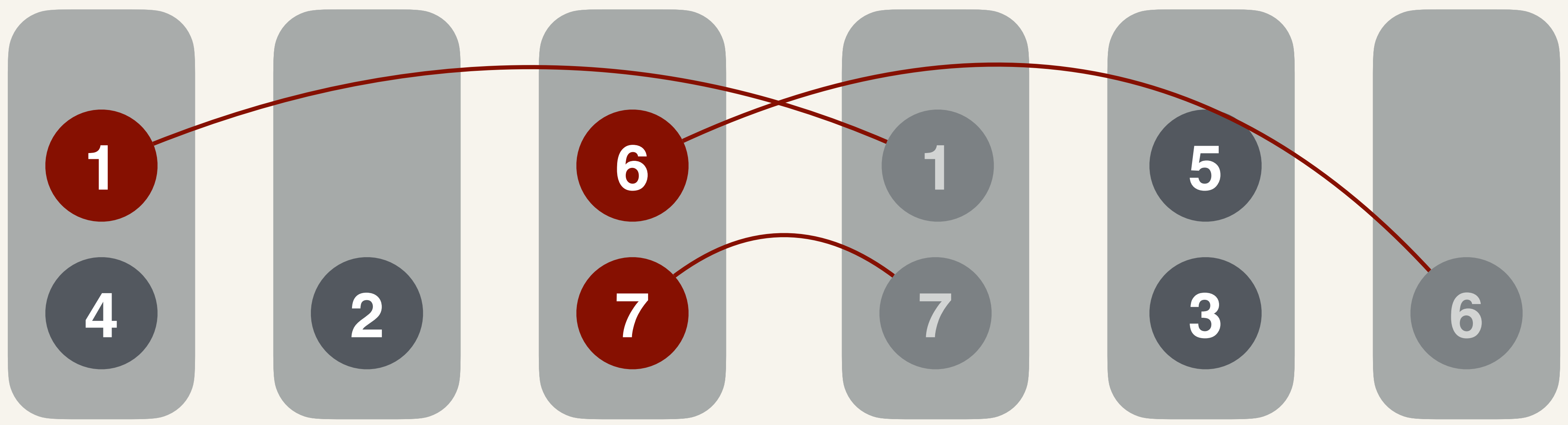


[(1,1), (7,2), (6,2)]

UpdateBuckets
k : int
F : int $\rightarrow$ (identifier, bucket_dest)

Update buckets for k identifiers

Bucketing Interface



[(1,1), (7,2), (6,2)]

UpdateBuckets
k : int
F : int $\rightarrow$ (identifier, bucket_dest)

Update buckets for k identifiers

Bucketing Interface

[(1,1), (7,2), (6,2)]

UpdateBuckets

k : int

F : int $\rightarrow$ (identifier, bucket_dest)

Update buckets for k identifiers

Sequential Bucketing

Can implement sequential bucketing with:

- n identifiers
- T total buckets
- K calls to UpdateBuckets, where each updates the ids in S_i

in $O(n + T + \sum_{i=0}^{K} |S_i|)$ work

Idea:

- Use dynamic arrays that are updated lazily

Parallel Bucketing

Can implement parallel bucketing with:

- n identifiers
- T total buckets
- K calls to UpdateBuckets, where each updates the ids in S_i
- L calls to NextBucket

in $O(n + T + \sum_{i=0}^{K} |S_i|)$ expected work and

$O((K + L) \log n)$ depth w.h.p.

Idea:

- Use dynamic arrays
- MakeBuckets: call UpdateBuckets. NextBucket: parallel filter

Parallel Bucketing

UpdateBuckets:

- Use work-efficient semisort [Gu et al. 2015]
- Given k (key, value) pairs, semisorts in $O(k)$ expected work and $O(\log k)$ depth w.h.p.

[(3,9), (4,7), …, (2,1), (1,1)]

[(2,1), (1,1), (7,1), …, (4,7), (6,7), …, (3,9)]

All ids going to bucket 1

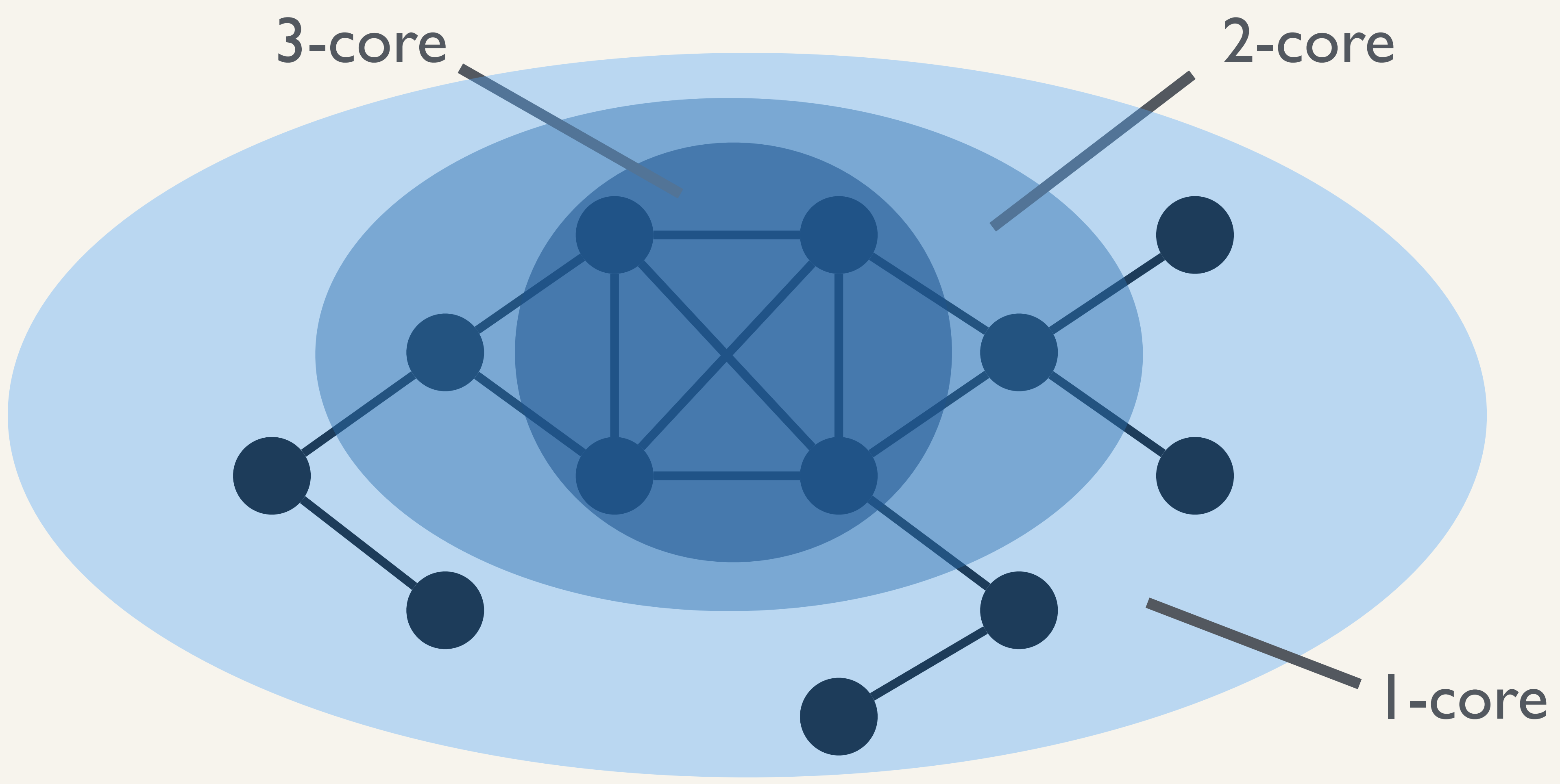
- Prefix sum to compute #ids going to each bucket
- Resize buckets and inject all ids in parallel

Can see paper for details on practical implementation and optimizations

k-Core Decomposition

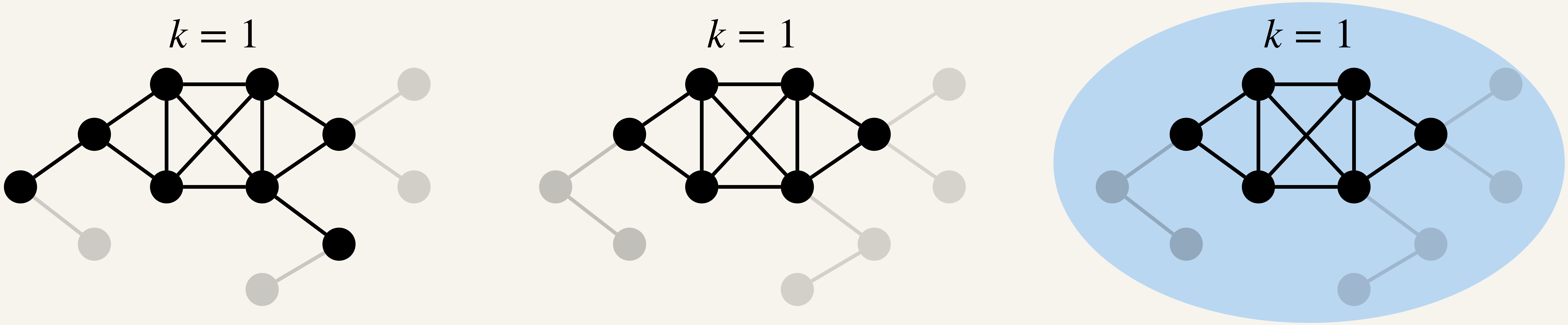
k-core : maximal connected subgraph of G where all vertices have degree at least k *within the subgraph*

coreness : largest k-core that a given vertex participates in



Widely used in network analysis tasks such as unsupervised clustering of social and biological networks

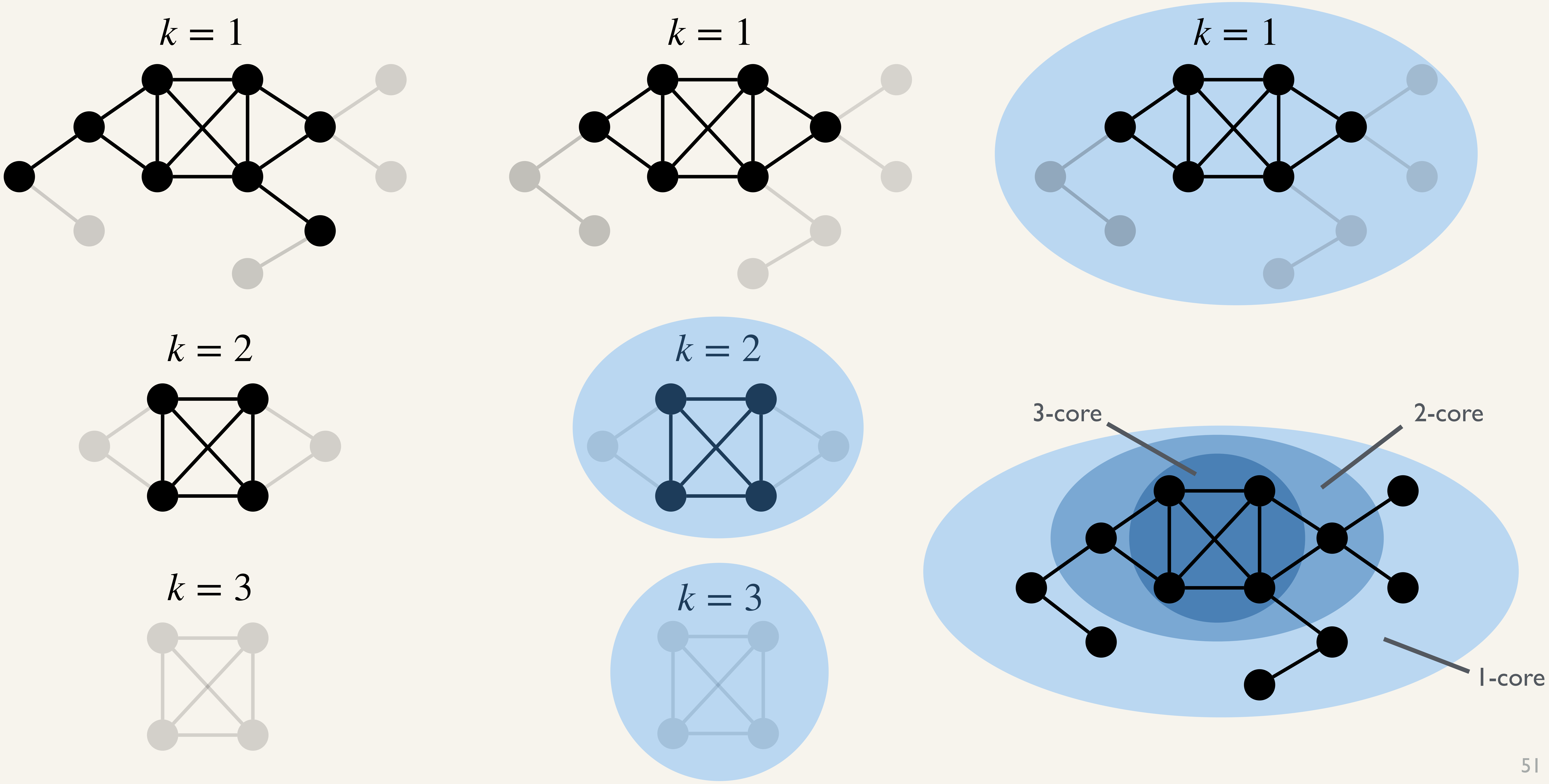
The Peeling Algorithm



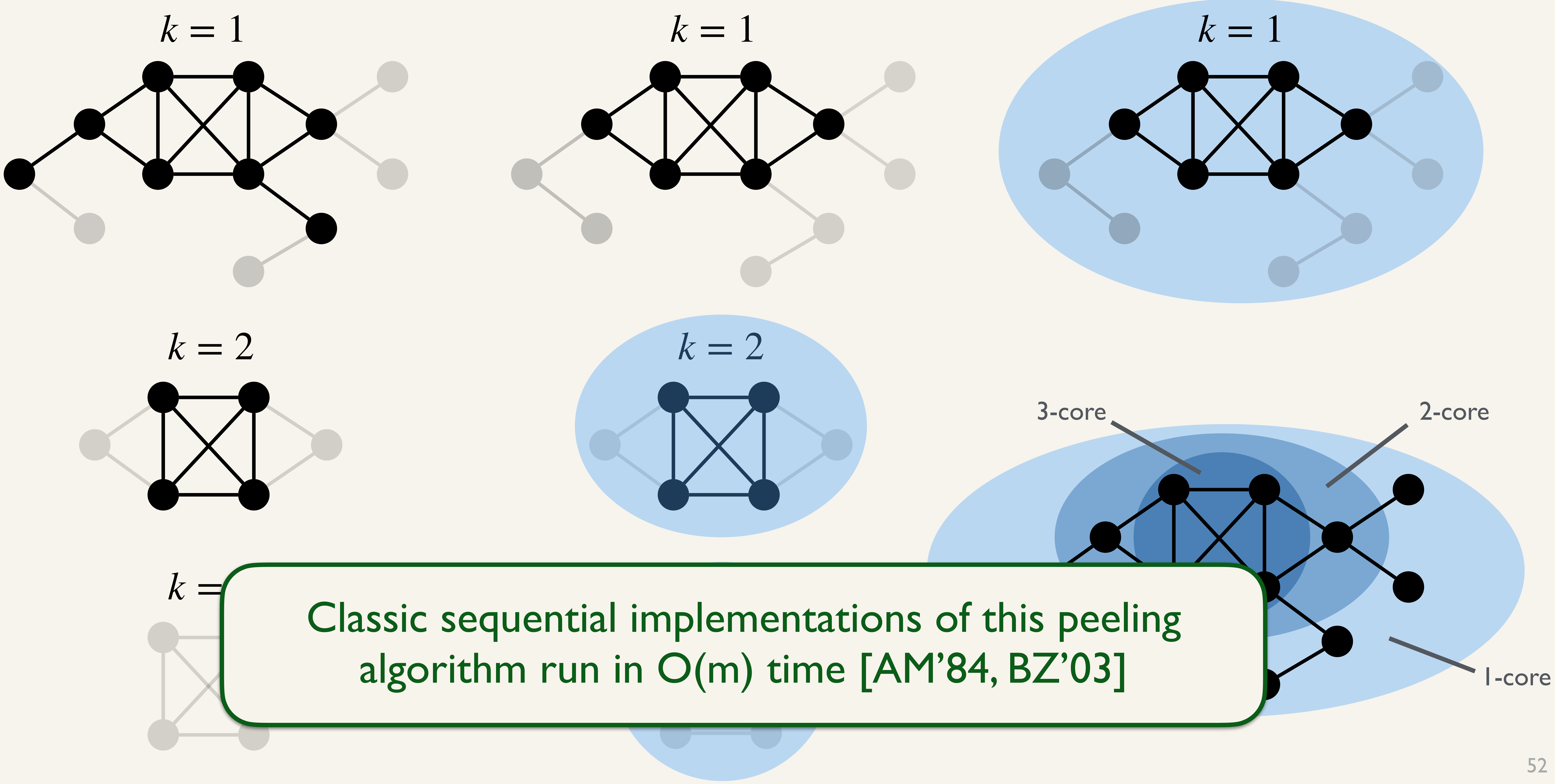
- Current degree of remaining vertices decreases as vertices are *peeled* from the graph

- Once a vertex's current degree is less than or equal to the current core number, it gets peeled

The Peeling Algorithm



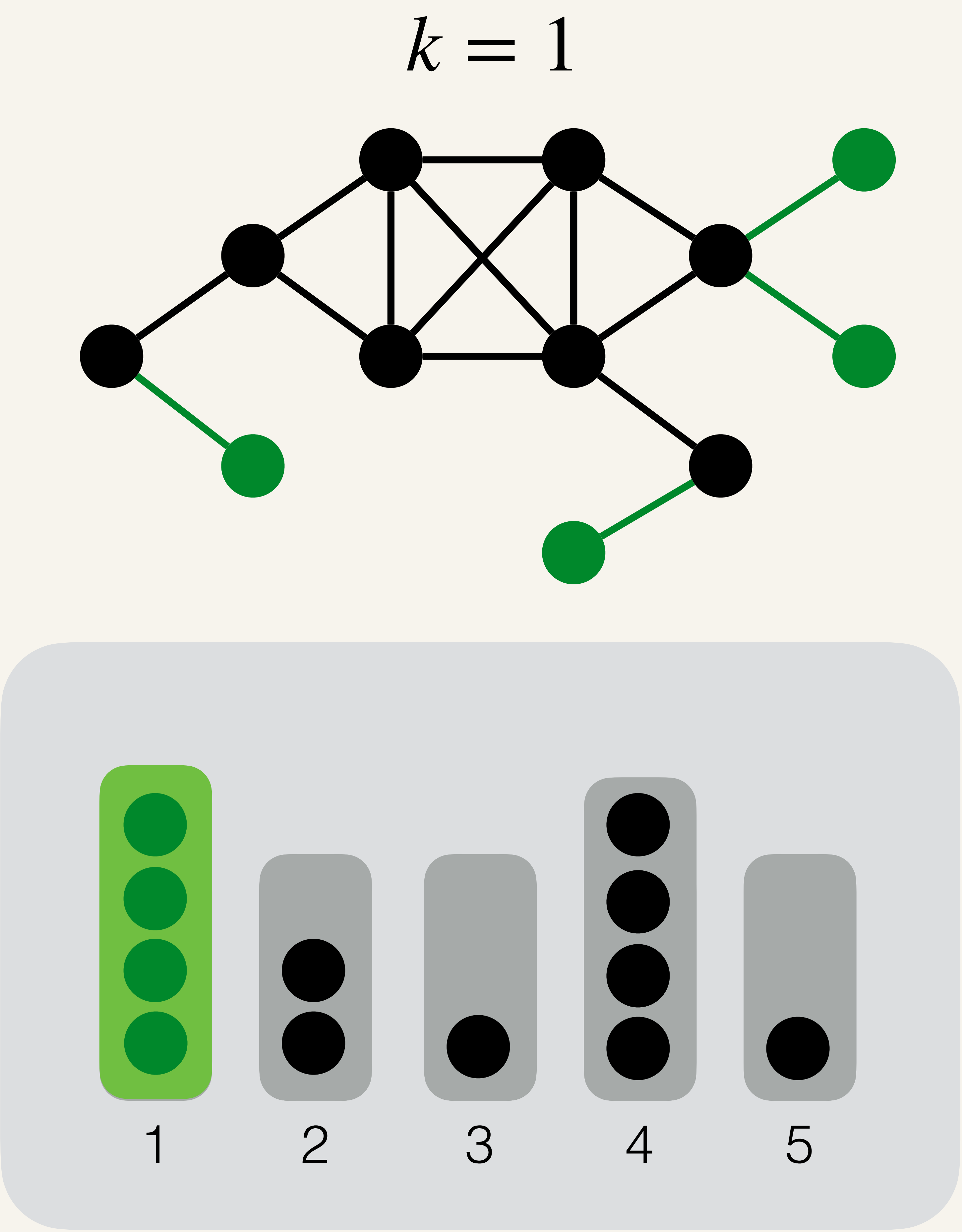
The Peeling Algorithm

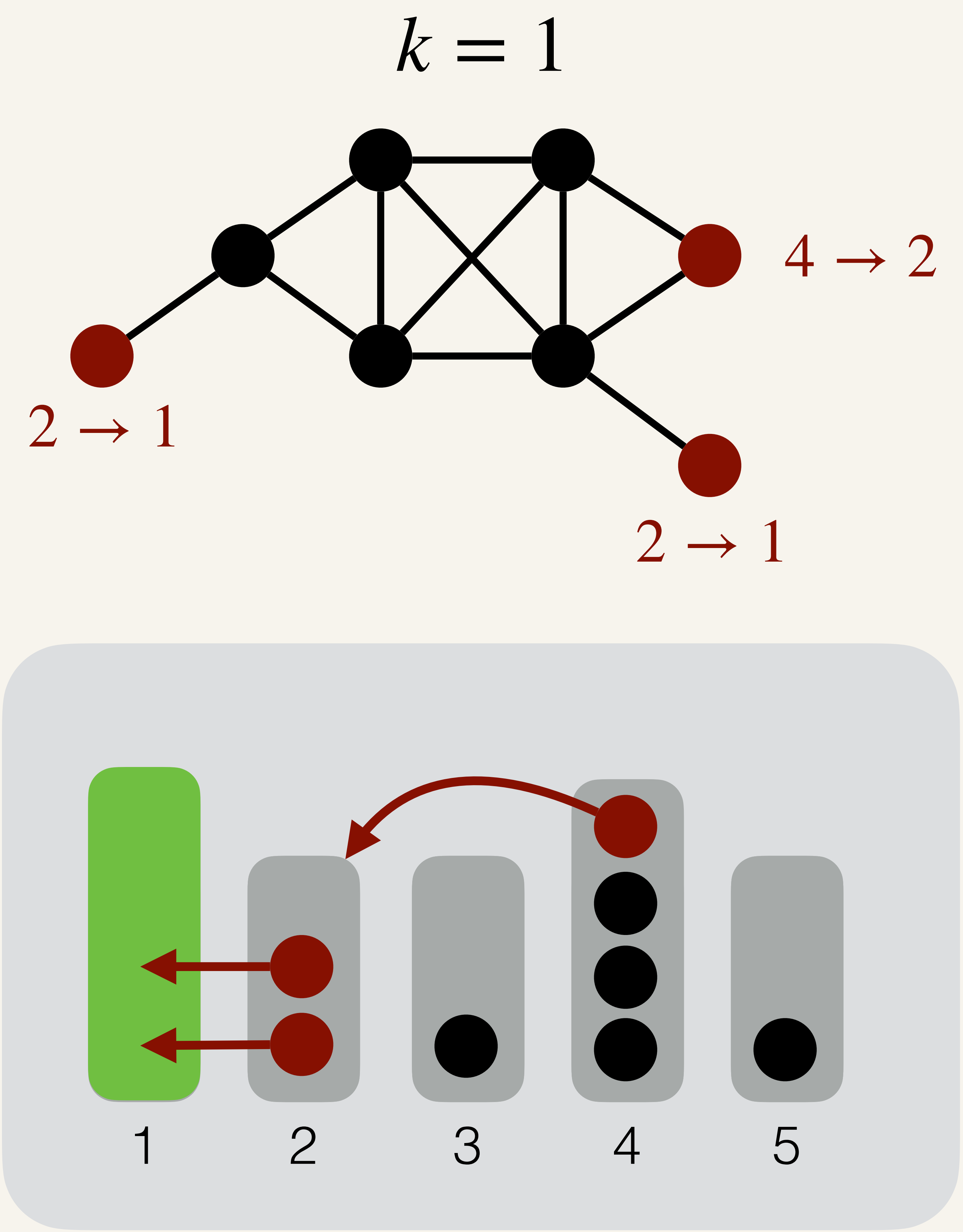


Classic sequential implementations of this peeling algorithm run in O(m) time [AM'84, BZ'03]

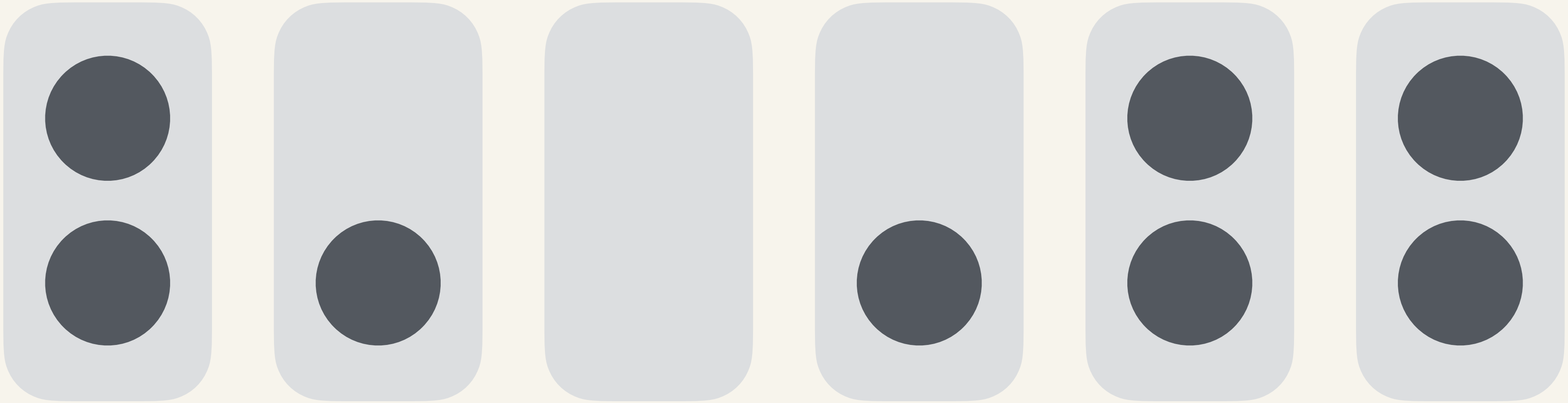
Parallel Peeling

Remove all vertices with degree less than or equal to the current core number in parallel

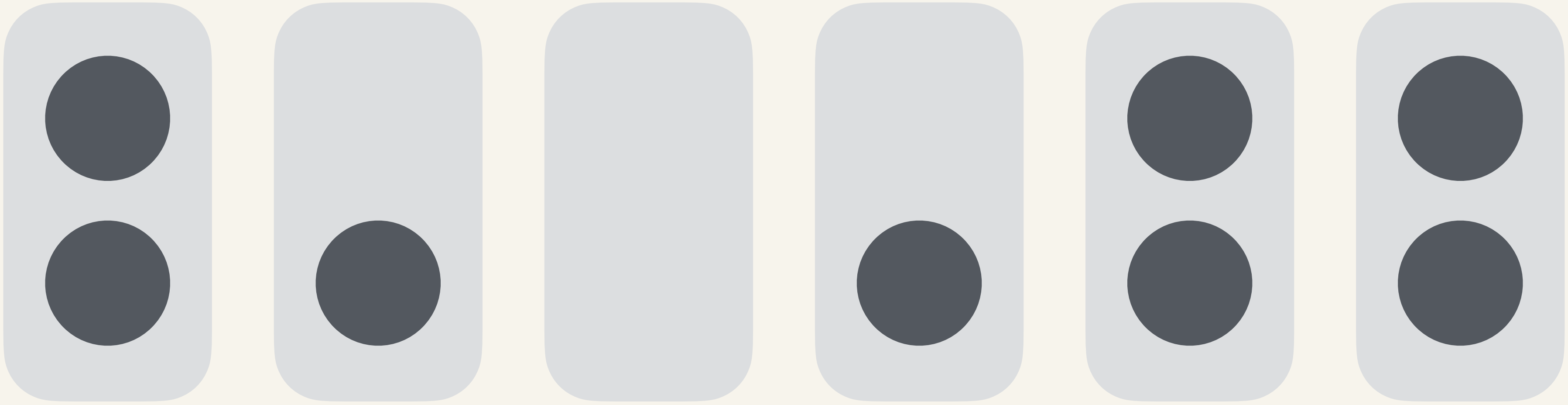




Insert vertices in bucket structure by degree

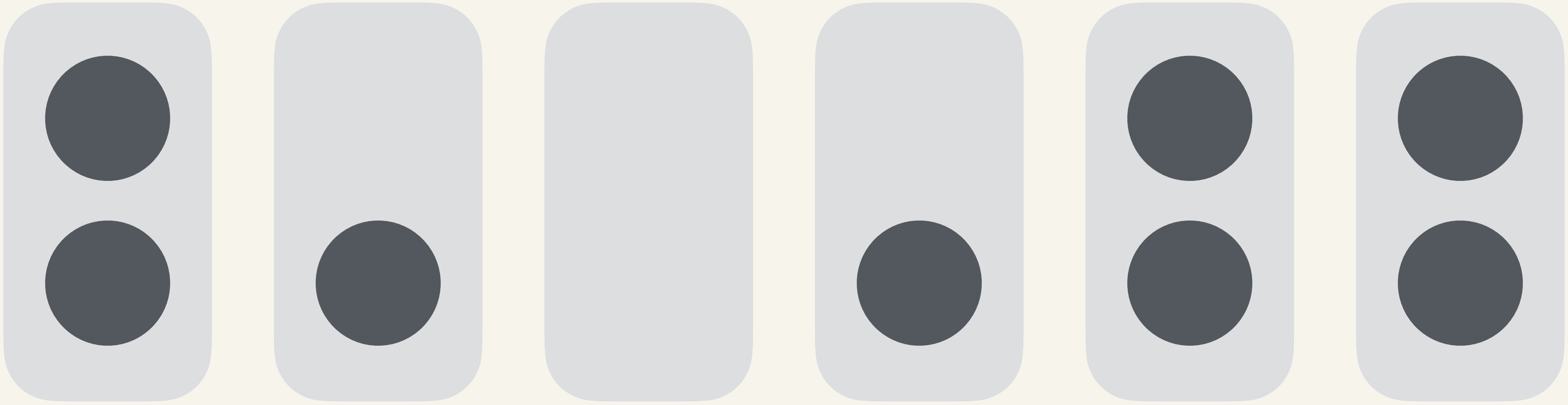


Insert vertices in bucket structure by degree



Insert vertices in bucket structure by degree

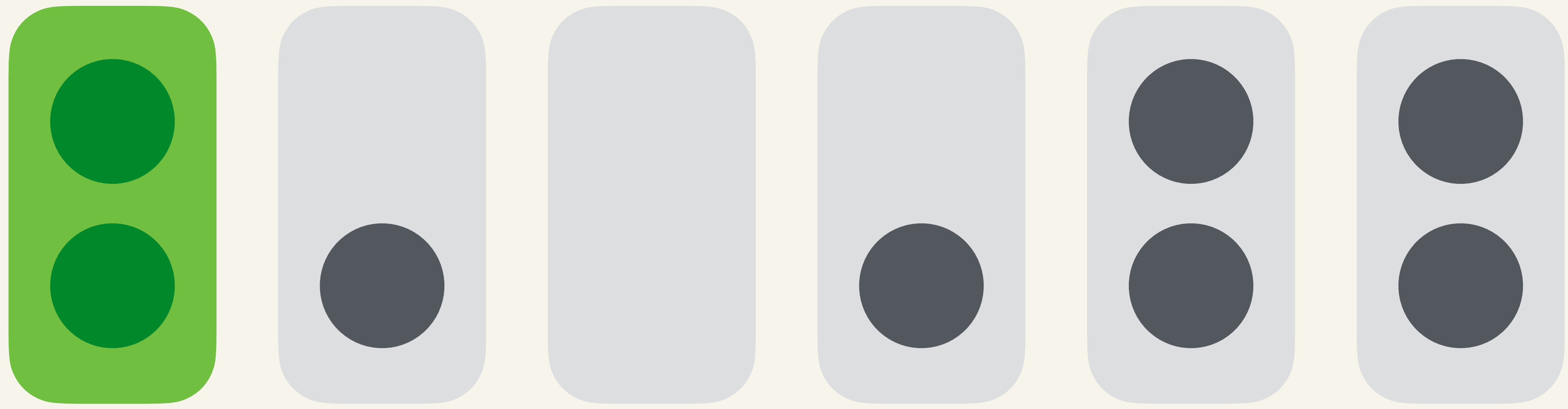
While not all vertices have been processed yet:



Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

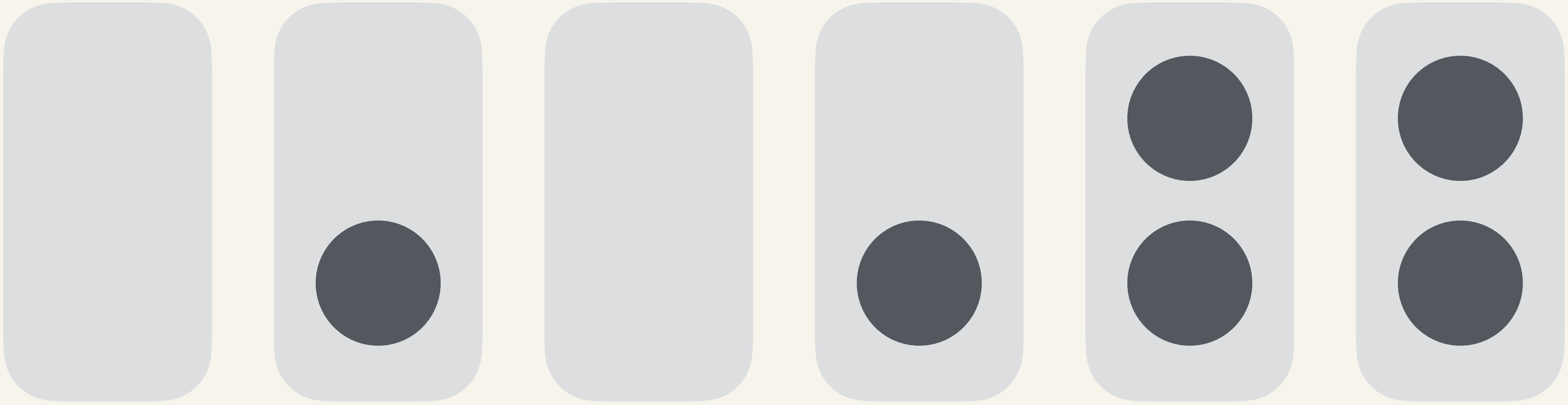
1. Extract the next bucket, set core numbers



Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

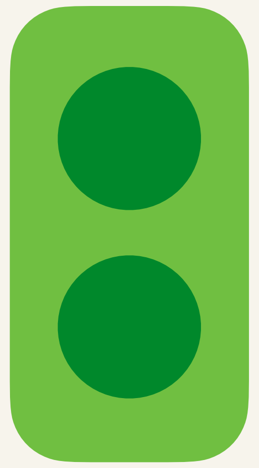
1. Extract the next bucket, set core numbers

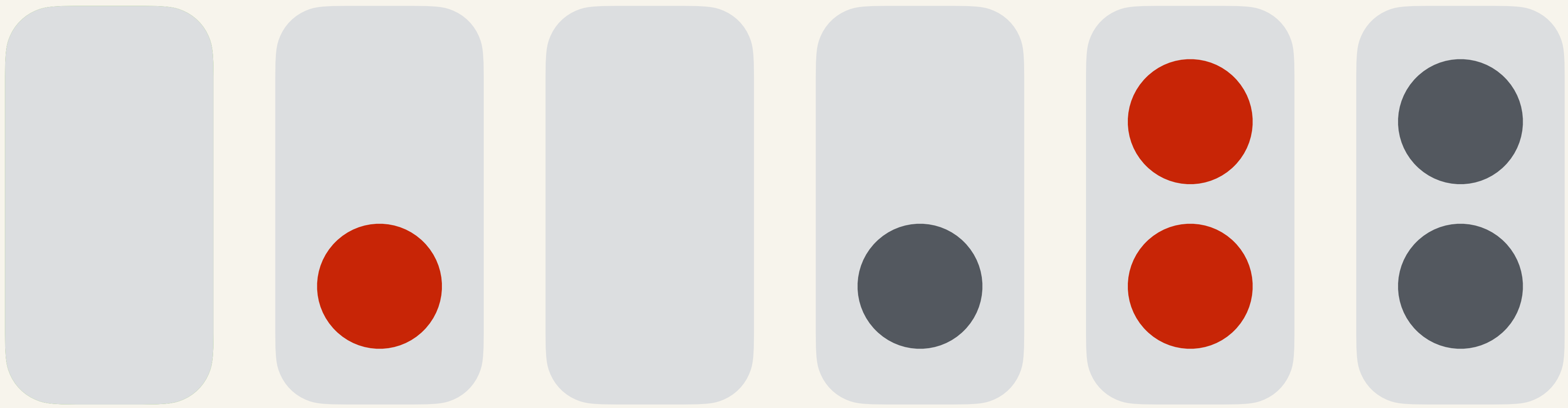


Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next bucket, set core numbers

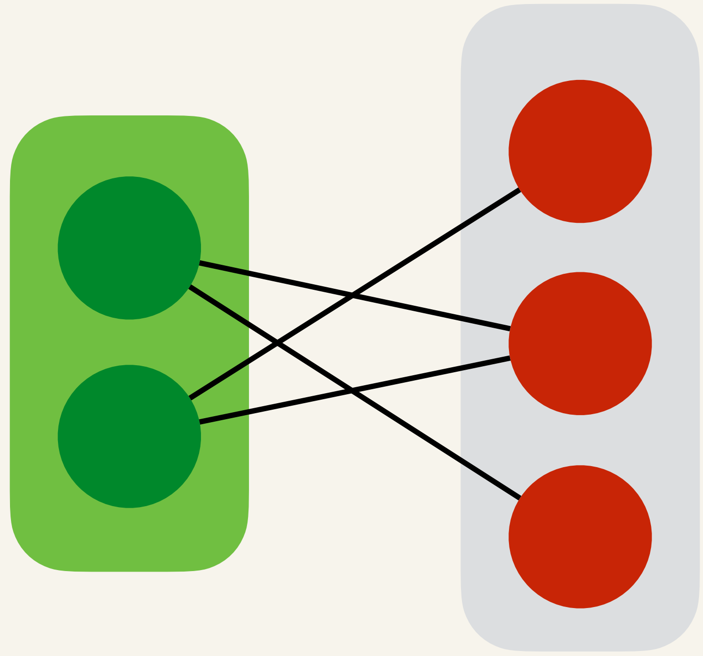


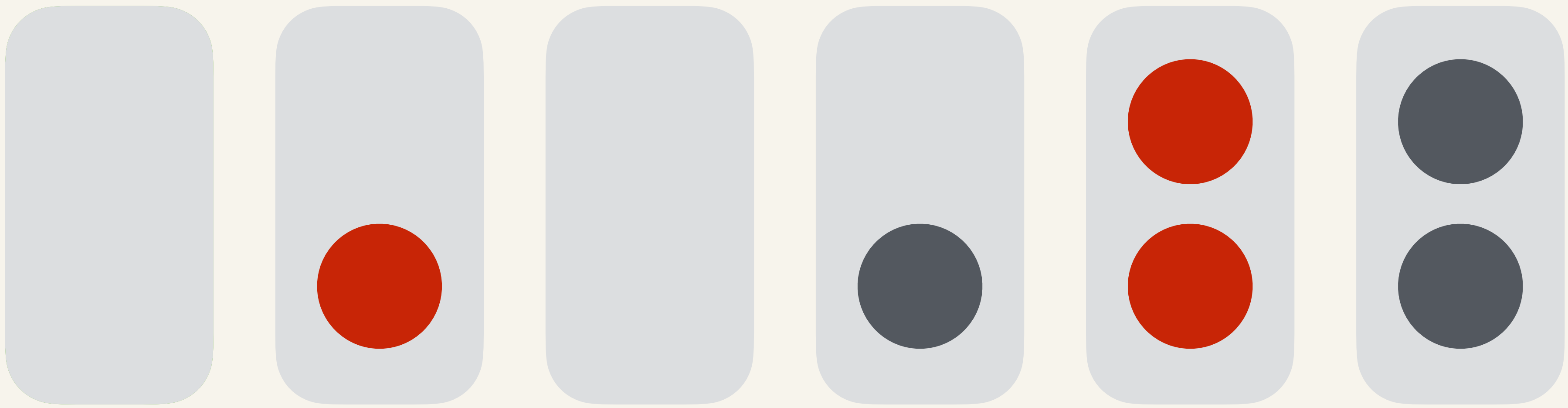


Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next bucket, set core numbers
2. Sum edges removed from each neighbor of this frontier

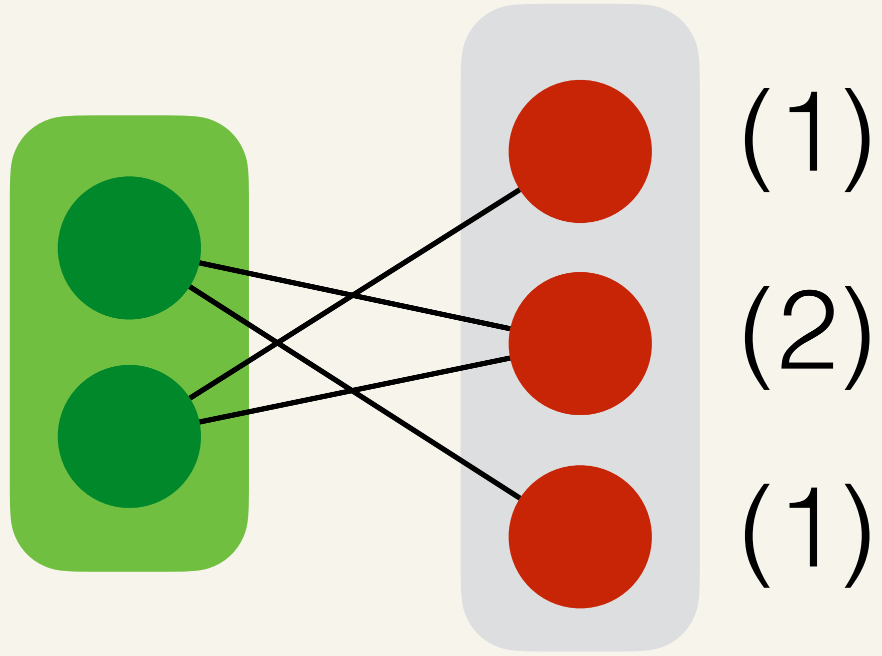


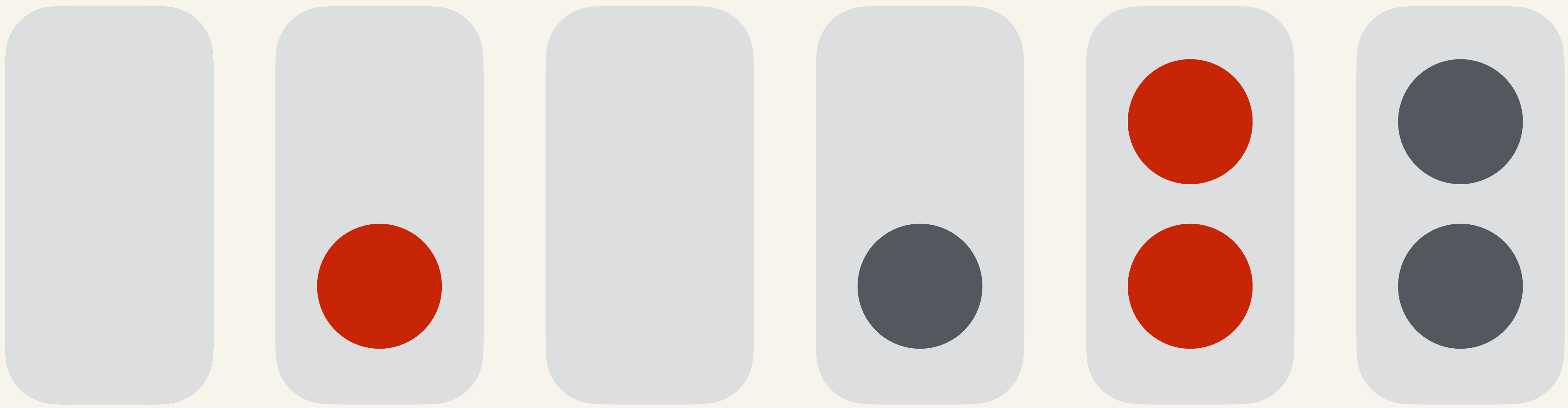


Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next bucket, set core numbers
2. Sum edges removed from each neighbor of this frontier

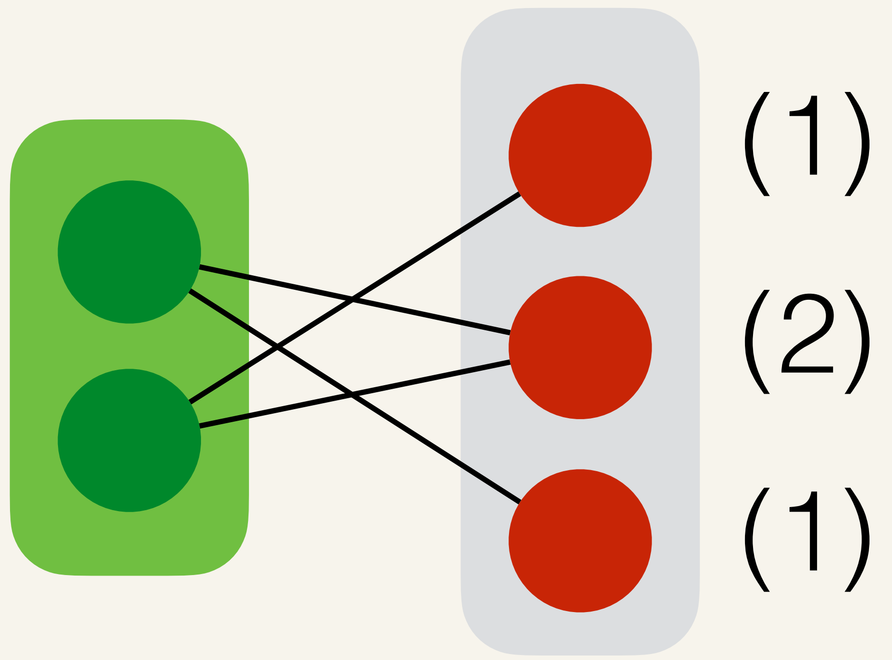


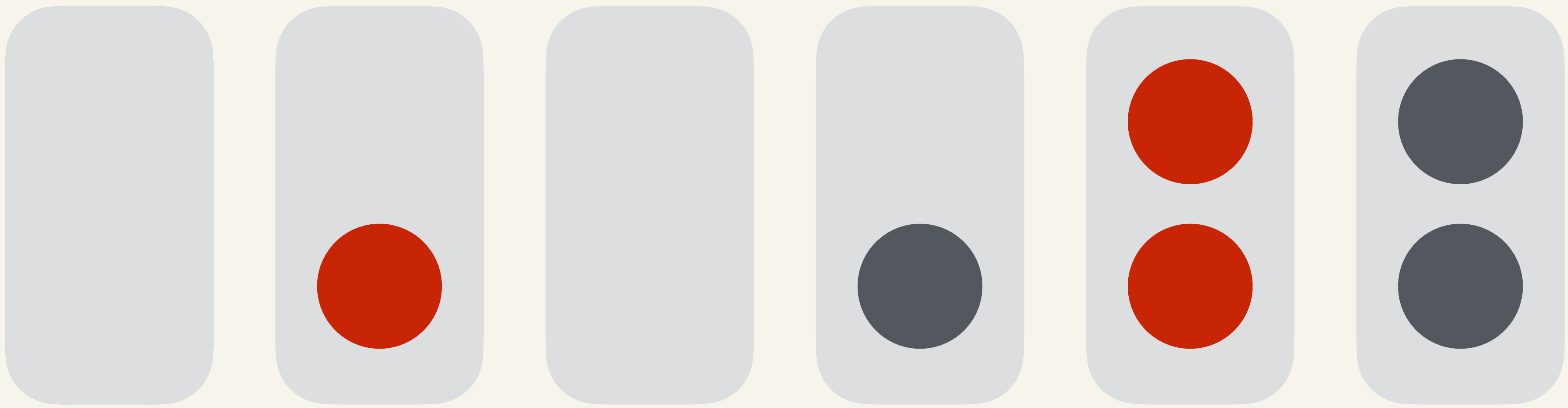


Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next bucket, set core numbers
2. Sum edges removed from each neighbor of this frontier
3. Compute the new buckets for the neighbors

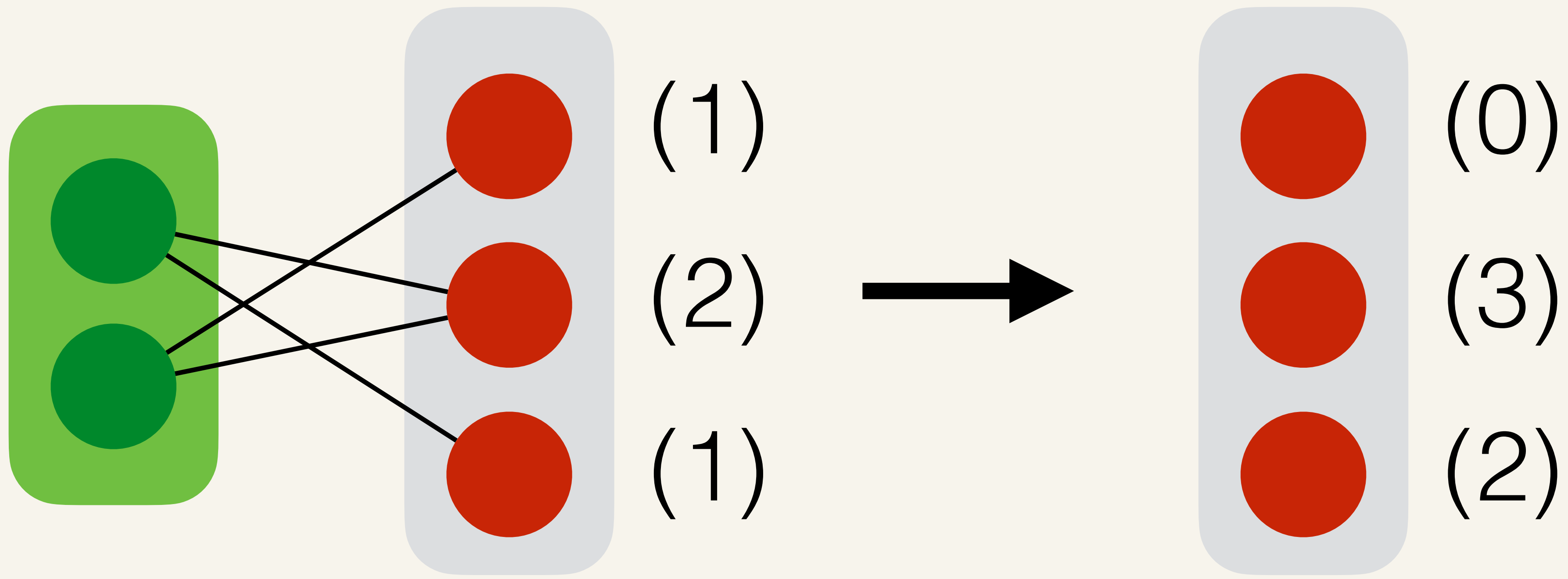


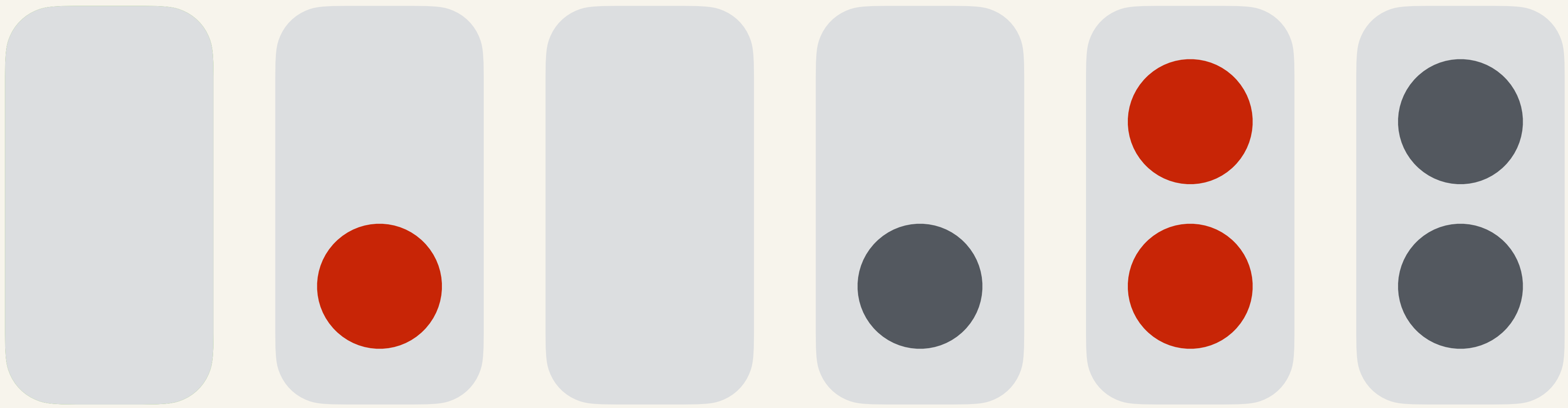


Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next bucket, set core numbers
2. Sum edges removed from each neighbor of this frontier
3. Compute the new buckets for the neighbors

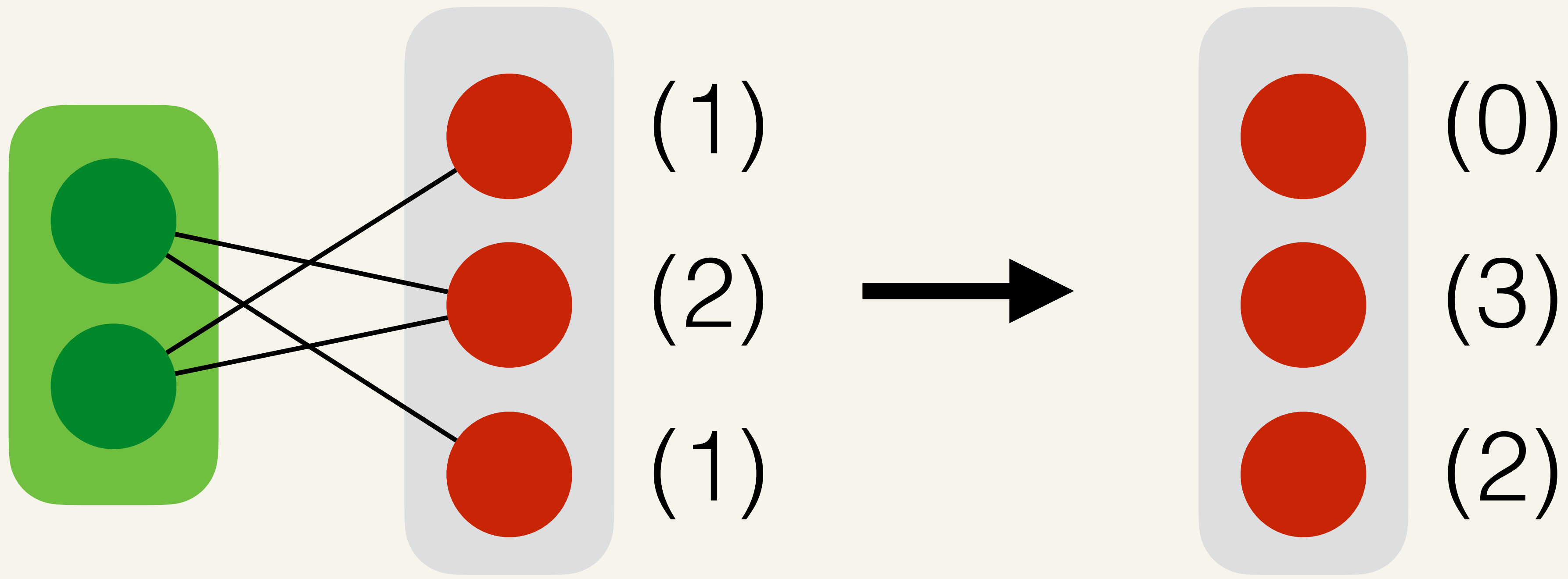


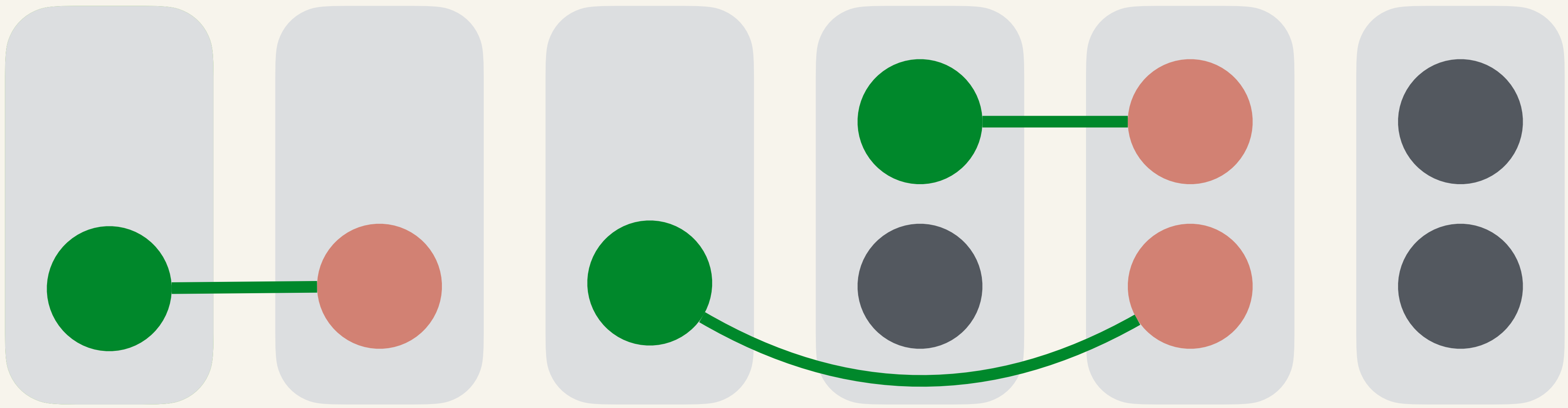


Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next bucket, set core numbers
2. Sum edges removed from each neighbor of this frontier
3. Compute the new buckets for the neighbors
4. Update the bucket structure with the (neighbors, buckets)

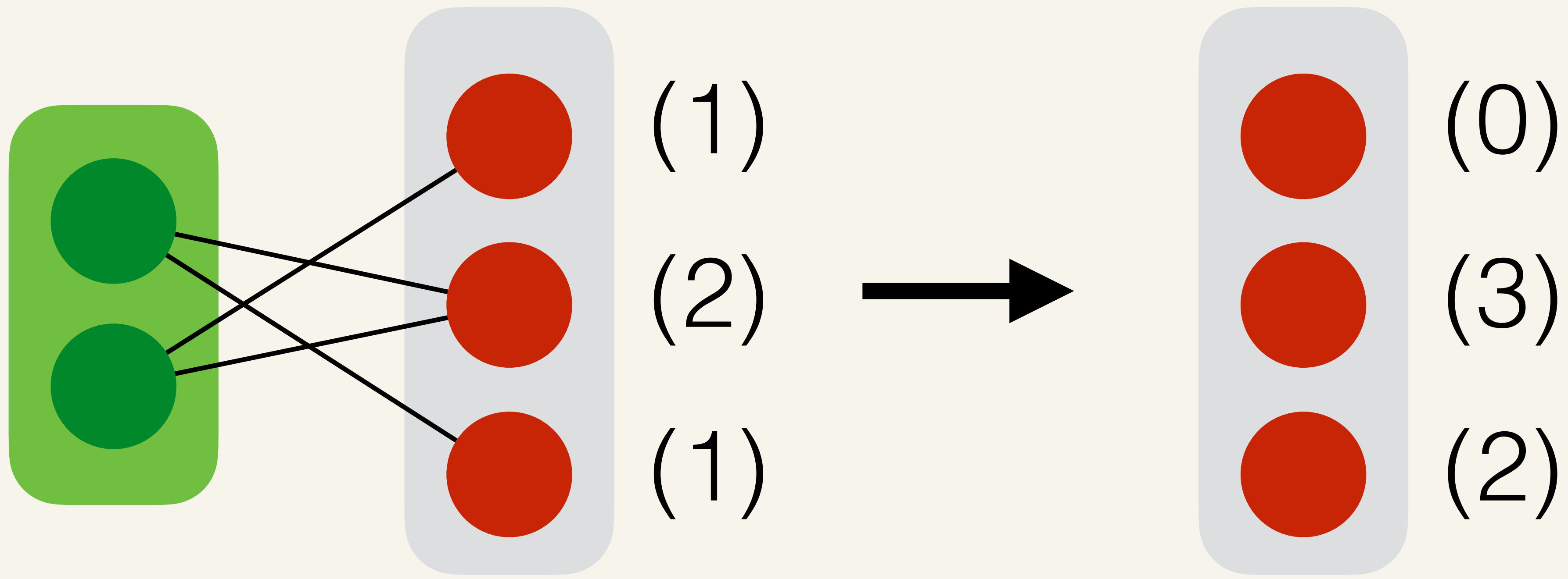


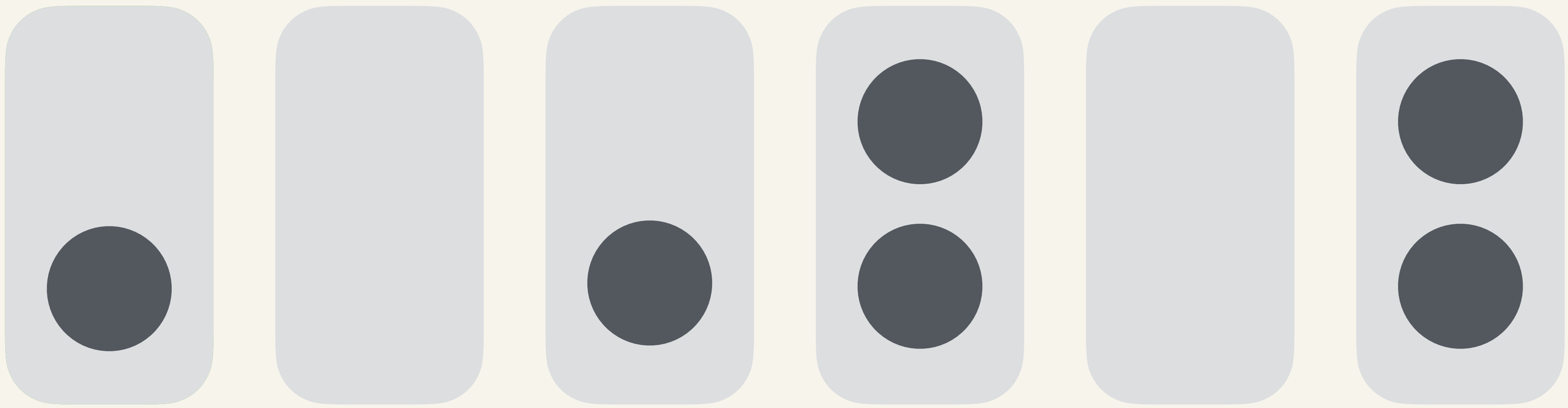


Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next bucket, set core numbers
2. Sum edges removed from each neighbor of this frontier
3. Compute the new buckets for the neighbors
4. Update the bucket structure with the (neighbors, buckets)

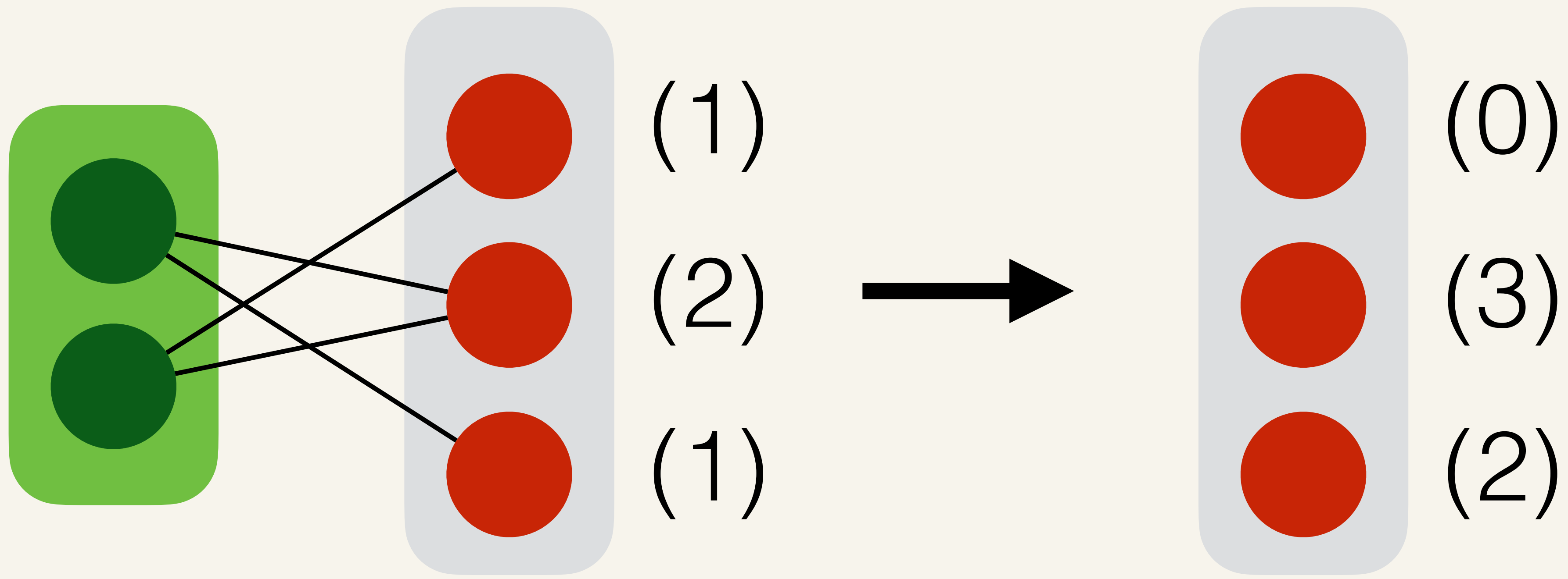


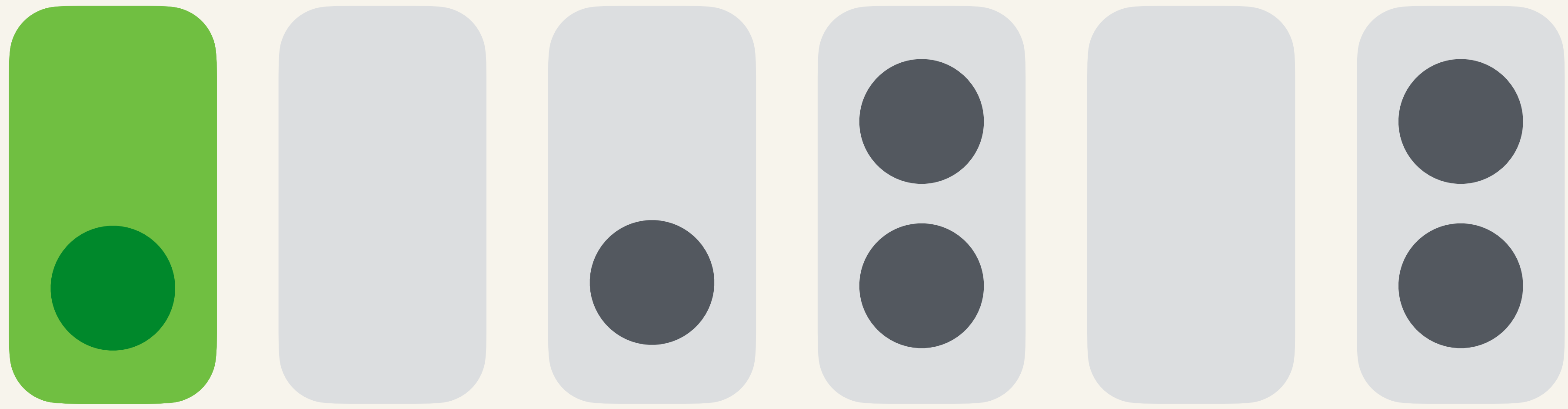


Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next bucket, set core numbers
2. Sum edges removed from each neighbor of this frontier
3. Compute the new buckets for the neighbors
4. Update the bucket structure with the (neighbors, buckets)





Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next bucket, set core numbers
2. Sum edges removed from each neighbor of this frontier
3. Compute the new buckets for the neighbors
4. Update the bucket structure with the (neighbors, buckets)

We process each edge at most once in each direction:

updates $= O(|E|)$

buckets $\leq |V|$

calls to NextBucket $= \rho$

calls to UpdateBuckets $= \rho$

Therefore the algorithm runs in:

$O(|E| + |V|)$ expected work

$O(\rho \log |V|)$ depth w.h.p.

On the largest graph we test on, $\rho = 130,728$

On 72 cores, our code finishes in a few minutes, but the work-inefficient algorithm does not terminate within 3 hours

We process each edge at most once in each direction:

updates $= O(|E|)$

buckets $\leq$

calls to Ne

calls to Up

Therefore the

On the larges

On 72 cores, ut the
work-inefficie 3 hours

Efficient peeling using Julienne

A Work-Efficient k-core Decomposition Algorithm

Julienne Algorithm in GBBS

- Actual code is under 50 lines of C++
- Parallel cost:

$O(m+n)$ expected work

$O(\rho \log n)$ depth whp

where ρ is the number of peeling rounds

Algorithm 1 k-core (Coreness)

```
Coreness[0, ..., n) := 0
procedure Coreness(G(V, E))
    vertexMap(V, fn v → Coreness[v] := d(v_i))          ▹ initialized to initial degrees
    B := makeBuckets(|V|, Coreness, increasing)         ▹ buckets processed in increasing order
    Finished := 0
    while (Finished < |V|) do
        (k, ids) := B.nextBucket()                      ▹ current core number, and vertices peeled this step
        Finished := Finished + |ids|
        condFn := fn v → return true
        applyFn := fn (v, edgesRemoved) →
            inducedD := D[v]
            if (inducedD > k) then
                newD := max(inducedD − edgesRemoved, k)
                Coreness[v] := newD
                bkt := B.getBucket(inducedD, newD)
                if (bkt ≠ nullbkt) then
                    return Some(bkt)
            return None
        Moved := nghCount(G, ids, condFn, applyFn)      ▹ Moved is an bktdest vertexSubset
        B.updateBuckets(Moved)                          ▹ update the buckets of vertices in Moved
    return Coreness
```

Our algorithm is the first work-efficient algorithm for k-core decomposition with non-trivial parallelism

Work and Depth of Algorithms in Julienne

Algorithm	Work	Depth	
k-core	$O(\|E\| + \|V\|)$	$O(\rho \log \|V\|)$	
wBFS	$O(D + \|E\|)$	$O(D \log \|V\|)$	
Delta-stepping	$O(w_\Delta)$	$O(d_\Delta \log \|V\|)$	[1]
Approx Set Cover	$O(M)$	$O(\log^3 M)$	[2]

ρ : number of rounds of parallel peeling

D : diameter

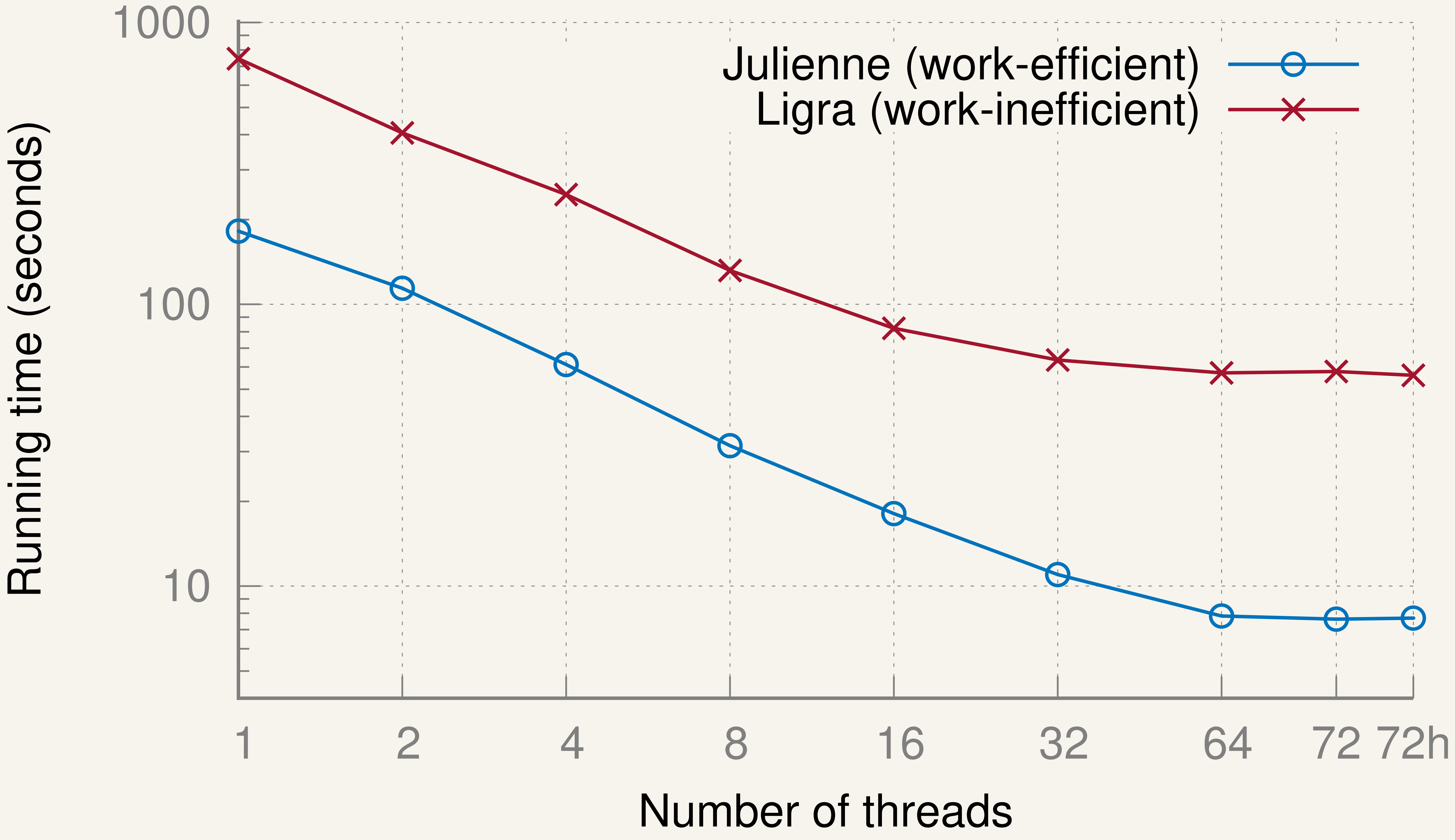
w_Δ, d_Δ : work and number of rounds of the delta-stepping algorithm

M : sum of sizes of sets

[1] Meyer, Sanders: Δ-stepping: a parallelizable shortest path algorithm

[2] Blelloch, Peng, Tangwongsan: Linear-work greedy parallel approximate set cover and variants

Experimental Results



Friendster

|V| = 121M

|E| = 3.6B

Across all inputs:

- Between 4-41x speedup over sequential peeling
- Speedups are smaller on small graphs with large ρ
- 2-9x faster than work-inefficient implementation

Experimental Results: Hyperlink Graphs

Hyperlink graphs extracted from Common Crawl Corpus

Graph	|V|	|E|	|E|(symmetrized)
HL2014	1.7B	64B	124B
HL2012	3.5B	128B	225B

- Previous analyses use supercomputers [1] or external memory [2]
- HL2012-Sym requires ~1TB of memory uncompressed

[1] Slota et al., 2015, Supercomputing for Web Graph Analytics

[2] Zheng et al., 2015, FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs

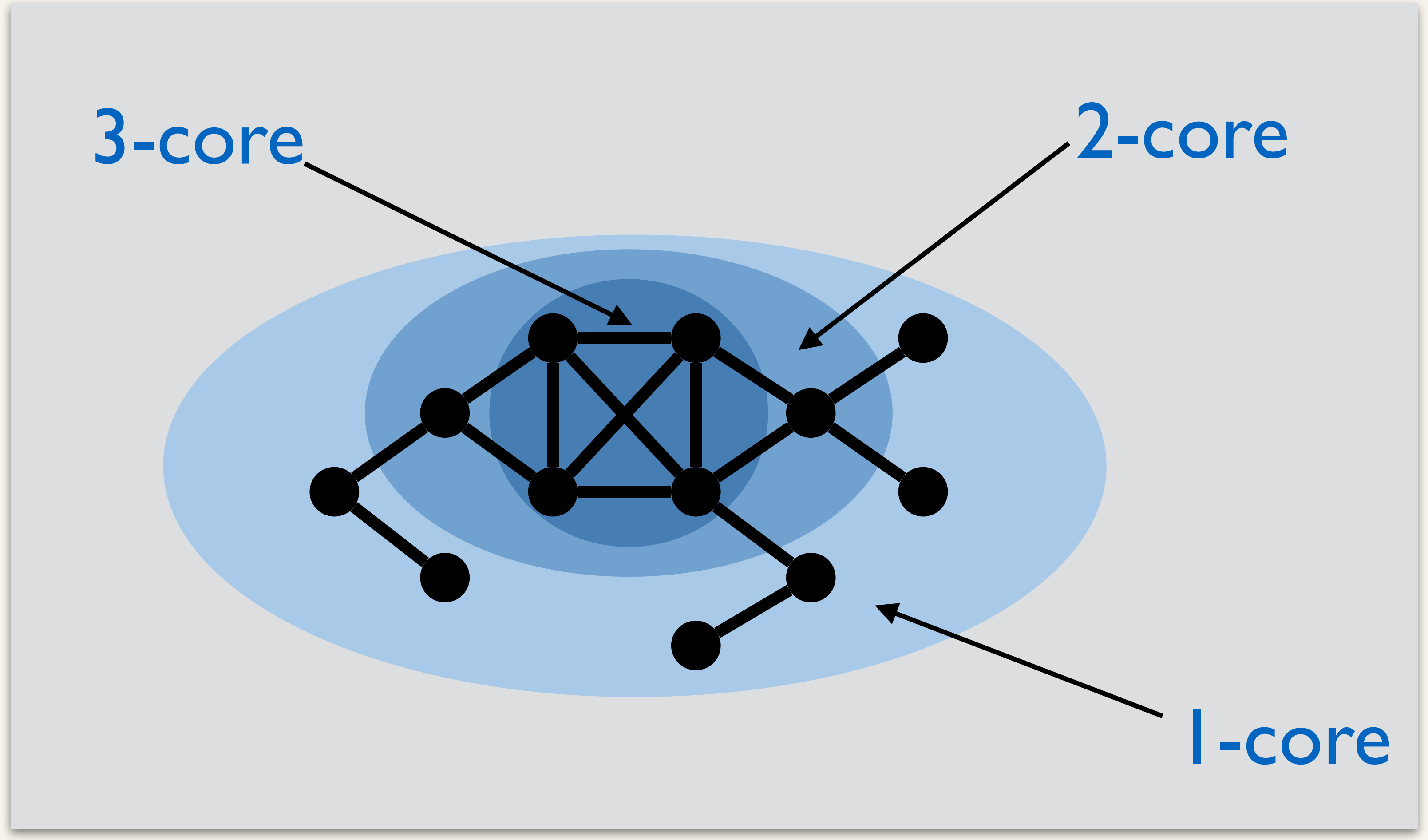
Experimental Results: Hyperlink Graphs

Graph	k-core	wBFS	Set Cover
HL2014	97.2	9.02	45.1
HL2012	206	—	104

Running time in seconds on 72 cores with hyperthreading

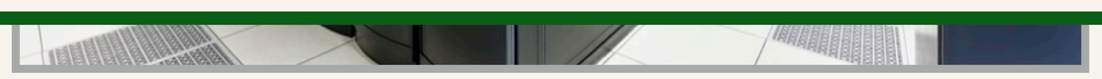
- Able to process in main-memory of 1TB machine by compressing
- 23-43x speedup across applications

k-Core Decomposition on the WebDataCommons Graph



k-core : maximal connected subgraph of G s.t. all vertices have degree at least k

	BlueWaters [SRM'16]	GBBS [**D**BS'18]
Time	363 seconds	184 seconds
Processors	8192	72
Memory	16 TB	1 TB
Quality	Approximate	Exact
Cost	Very Expensive	Highly Affordable



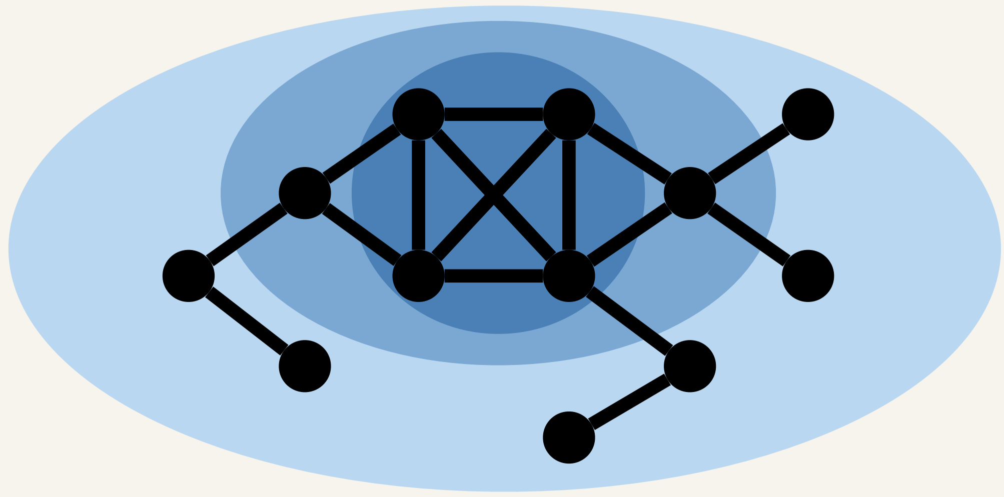
1.95x faster than the approximate distributed result by SRM'16, using 56.8x fewer hyper-threads and 16.3x less memory

Summary: Julienne

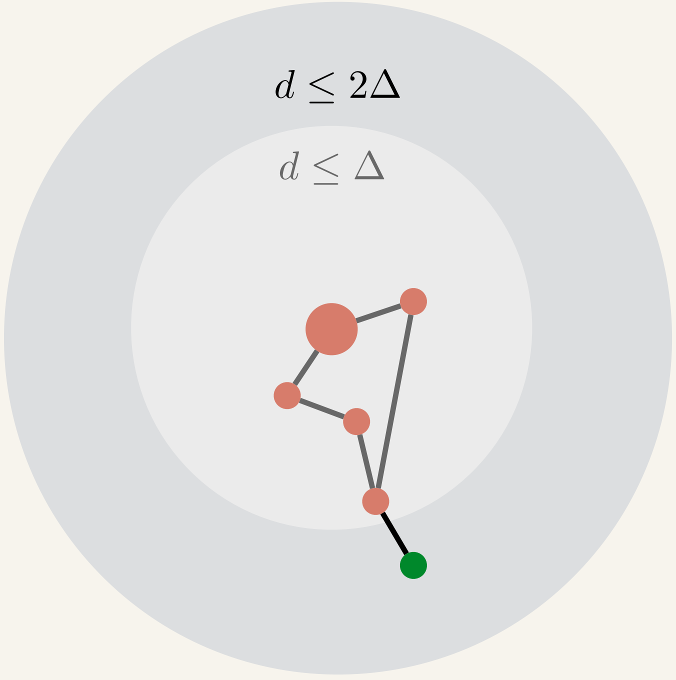
Julienne: framework for *bucketing-based algorithms*

- Codes:
  - Simple (< 100 lines each)
  - Theoretically efficient (strong bounds on work and depth)
  - Good performance in practice
  - Code included as part of the GBBS library

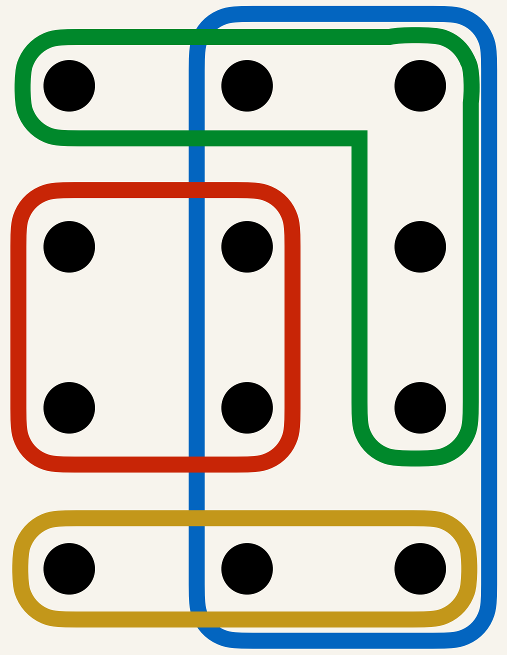
k-core



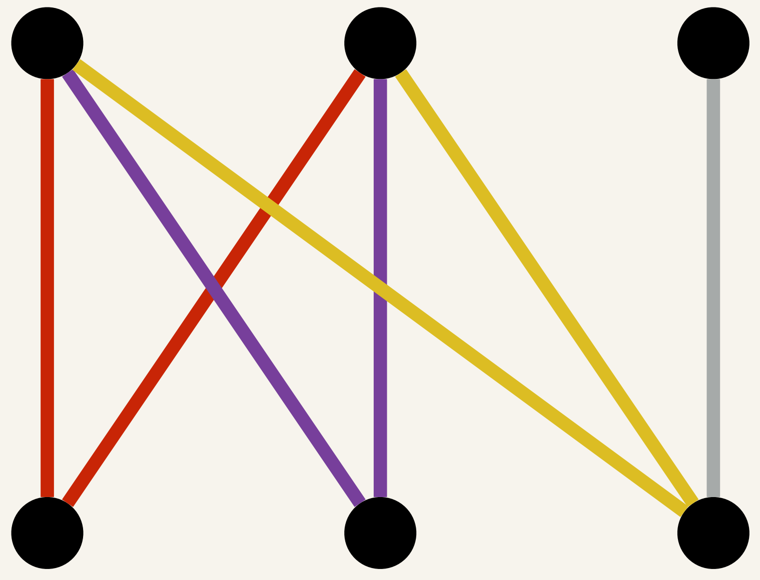
Delta-stepping wBFS



Parallel Approximate Set Cover



Parallel k-Tip Decomposition



Theoretically-Efficient Parallel Graph Algorithms can be Fast and Scalable [DBS'18]

Can we solve a broad set of fundamental graph problems on the largest graphs, affordably and quickly?

The Graph-Based Benchmark Suite (GBBS)

- Introduce a benchmark suite for graph problems with over 20 important problems
- GBBS algorithms achieve state-of-the-art results on the largest publicly available graphs

Connectivity Problems

Low-Diameter Decomposition
Connectivity
Spanning Forest
Biconnectivity
Minimum Spanning Forest
Strongly Connected Components

Subgraph Problems

k-Core Decomposition
k-Truss Decomposition
Apx. Densest Subgraph
Triangle Counting
Higher-Clique Counting

Covering Problems

Maximal Ind. Set
Maximal Matching
Apx. Set Cover
Graph Coloring

Shortest Path Problems

Breadth-First Search
Betweenness Centrality
Bellman-Ford
General Weight SSSP
Integral Weight SSSP
SS Widest Path
k-Spanner

Eigenvector Problems

PageRank
Personalized PageRank
Personalized SimRank

github.com/paralg/gbbs

Benchmarking Connectivity on WebDataCommons Graph

Benchmarks are based on I/O specifications, e.g.,

Maximal Independent Set

Input: $G(V, E)$an undirected graph

Output: $U \subseteq V$, a set of vertices such that no two vertices in U are neighbors, and all vertices in $V \setminus U$ have a neighbor in U

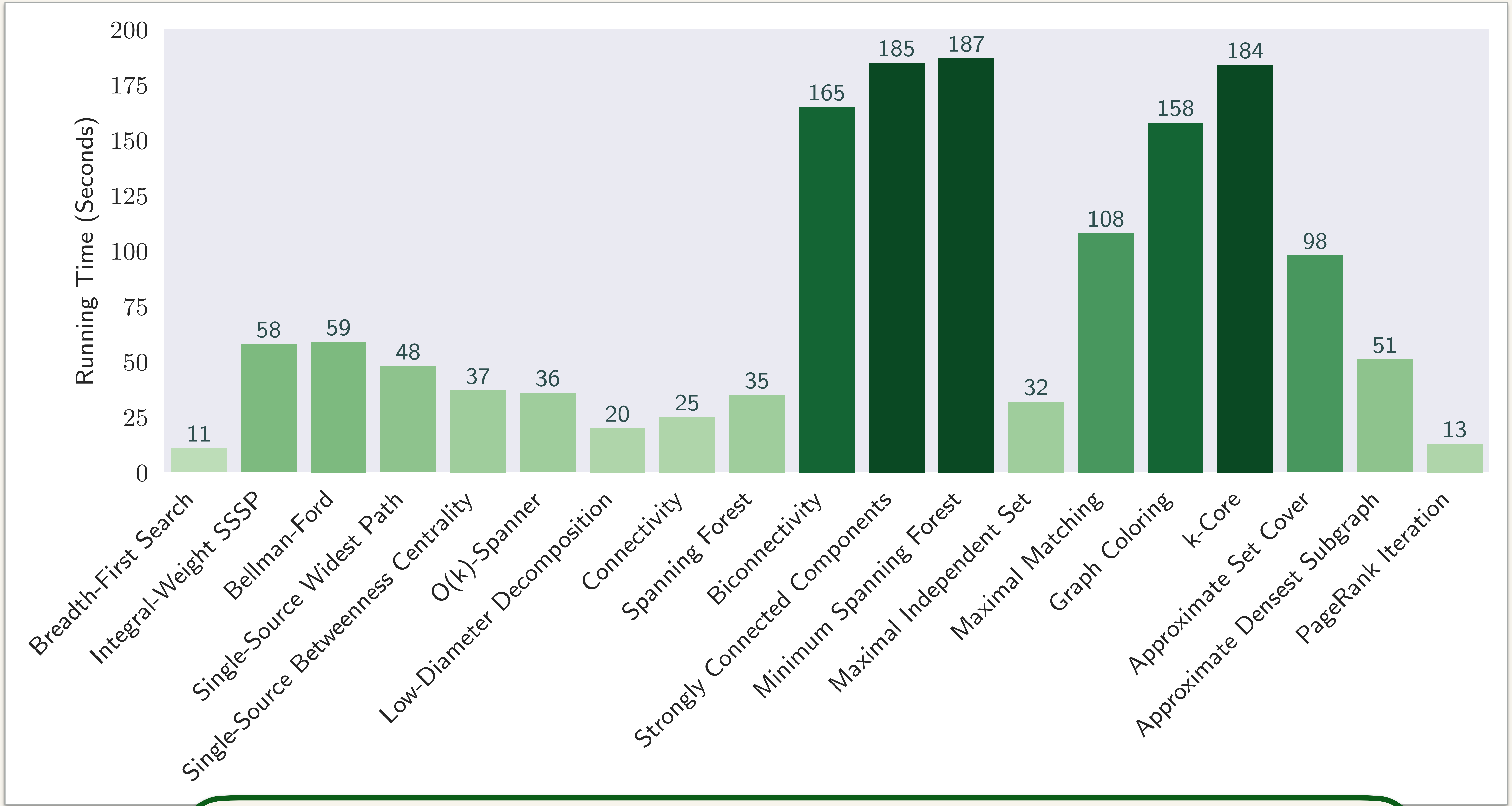
k-core (Coreness)

Input: $G(V, E)$ an undirected graph

Output: A mapping from each vertex to its coreness value (the maximum k such that the vertex is in a non-empty k-core)

I/O specification makes it easy to compare different algorithm implementations

GBBS Results on WDC Hyperlink Graph



A broad set of fundamental graph problems can be solved on a graph with over 200 billion edges in 3 minutes

Work and Depth of GBBS Results

† : in expectation   * : whp

Problem	Work	Depth
Breadth-First Search (BFS)	$O(m)$	$\tilde{O}(\text{diam}(G))$
Integral-Weight SSSP (weighted BFS)	$O(m)^{\dagger}$	$\tilde{O}(\text{diam}(G))*$
General-Weight SSSP (Bellman-Ford)	$O(\text{diam}(G) \cdot m)$	$\tilde{O}(\text{diam}(G))$
Single-Source Widest Path (Bellman-Ford)	$O(\text{diam}(G) \cdot m)$	$\tilde{O}(\text{diam}(G))$
Single-Source Betweenness Centrality (BC)	$O(m)$	$\tilde{O}(\text{diam}(G))$
O(k)-Spanner	$O(m)$	$\tilde{O}(k \log n)*$
Low-Diameter Decomposition (LDD)	$O(m)$	$O(\log^2 n)*$
Connectivity (CC)	$O(m)^{\dagger}$	$O(\log^3 n)*$
Spanning Forest	$O(m)^{\dagger}$	$O(\log^3 n)*$
Biconnectivity	$O(m)^{\dagger}$	$O(\max(\text{CC}, \text{BFS}))$
Strongly Connected Components (SCC)	$O(m \log n)^{\dagger}$	$\tilde{O}(\text{diam}(G))*$
Minimum Spanning Forest (MSF)	$O(m \log n)$	$O(\log^2 n)$
Maximal Independent Set (MIS)	$O(m)^{\dagger}$	$O(\log^2 n)*$
Maximal Matching (MM)	$O(m)^{\dagger}$	$O(\log^2 n)*$
Approximate Densest Subgraph	$O(m)$	[illegible]
PageRank Iteration	$O(n + m)$	$O(\log n)$

Main Challenge:
How do we build simple and provably-efficient implementations of these algorithms that work on the largest real-world graphs?

GBBS Library

❖ High-level graph processing interface in the lineage of *Ligra* [SB'12]

❖ Provides many useful primitives

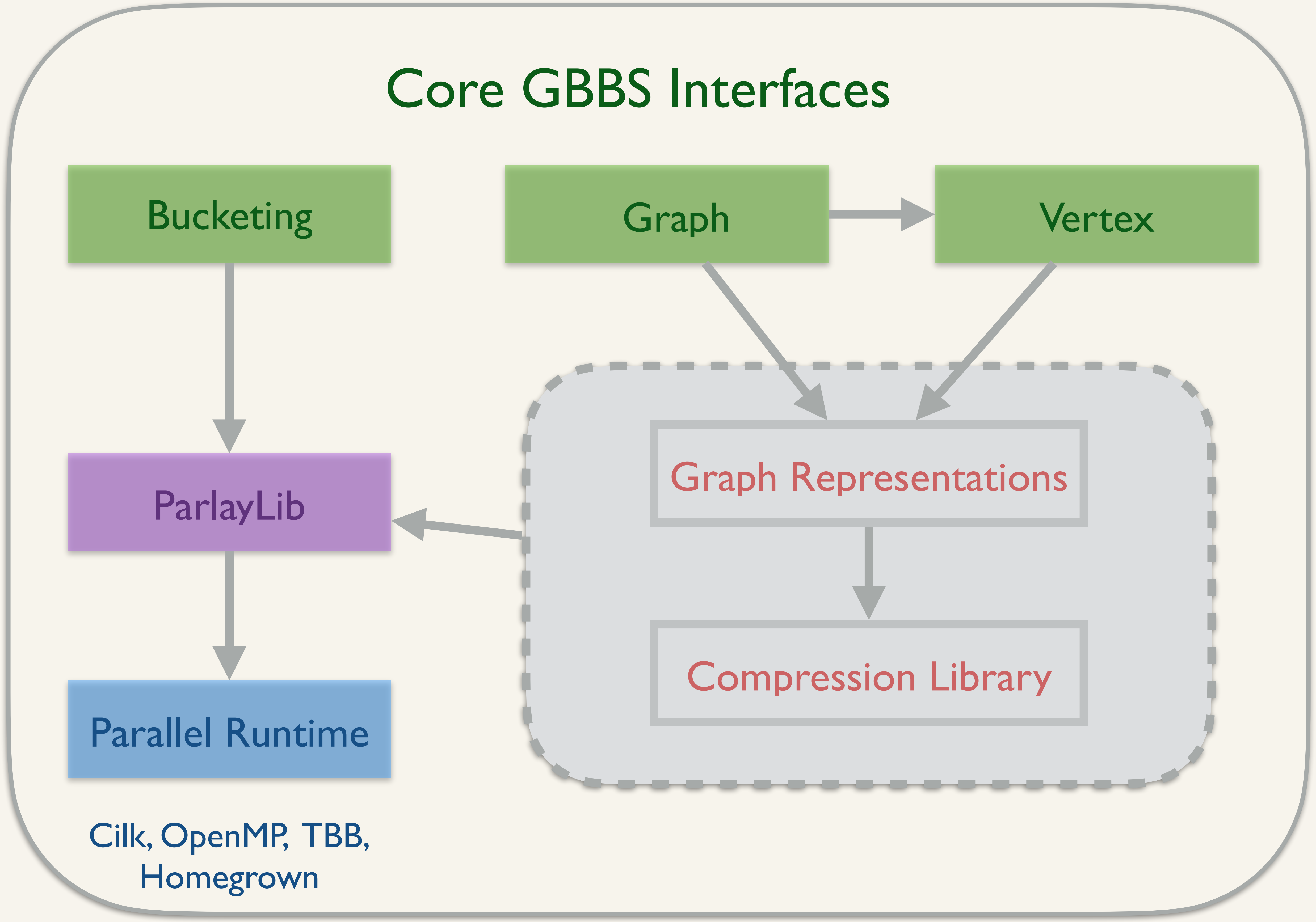
Vertex Operations

- Map
- Reduce
- Filter
- Pack
- Intersect

Graph Operations

- Filter
- Pack
- Contract

❖ Compressed graph representations based on extending Ligra+



Graph	*|V|*	*|E|*	*Size (CSR)*	*Compressed*	*Bytes/edge*
WDC Hyperlink	3.5B	128B	1080GB	446GB	1.74
WDC Hyperlink (Sym)	3.5B	225B	928 GB	351GB	1.56

Vertex Interface

	Operator	Type	Work	Depth
Neighborhood operators:	map	: (edge → void) → void	$O(\lvert N(v)\rvert)$	$O(\log n)$
	reduce	: (edge → E) $*$ E monoid → E		
	scan	: (edge → E) $*$ E monoid → E		
	count	: (edge → bool) → int		
	filter	: (edge → bool) → E seq		
	pack	: (edge → bool) → void		
	iterate	: (edge → bool) → void	$O(d_{it})$	$O(d_{it})$
	i-th	: int → edge	$O(1)$	$O(1)$
	degree	: unit → int		
	getNeighbors	: unit → nghlist		
Vertex-Vertex operators:	intersection	: (nghlist $*$ nghlist) → int	$O(l \log(h/l + 1))$	$O(\log n)$
	union	: (nghlist $*$ nghlist) → int		
	difference	: (nghlist $*$ nghlist) → int		

Provides **functional primitives** for commonly used vertex operations with good theoretical bounds on their cost

Provides functional primitives for performing whole-graph operations, and for operations that consume and produce vertexSubsets

Graph Interface

	Operator	Type	Work	Depth
Graph operators:	numVertices	: unit → int	$O(1)$	$O(1)$
	numEdges	: unit → int		
	getVertex	: int → vertex		
	filterGraph	: (edge → bool) → graph	$O(n+m)$	$O(\log n)$
	packGraph	: (edge → bool) → unit		
	extractEdges	: (edge → bool) → edge sequence		
	contractGraph	: int sequence → graph	$O(n+m)^{\dagger}$	$O(\log n)^{\ddagger}$
VertexSubset operators:	edgeMap	: vset ∗ (edge → bool) ∗ (vtxid → bool) → vset	$O\left(\sum_{u \in U} d(u)\right)$	$O(\log n)$
	edgeMapVal	: vset ∗ (edge → O option) ∗ (vtxid → bool) → O vset		
	srcReduce	: vset ∗ (edge → O) ∗ O monoid ∗ (vtxid → bool) → O vset	$O\left(\lvert U \rvert + \sum_{u \in U'} d(u)\right)$	$O(\log n)$
	srcCount	: vset ∗ (edge → bool) ∗ (vtxid → bool) → int vset		
	srcPack	: vset ∗ (edge → bool) ∗ (vtxid → bool) → int vset		
	nghReduce	: vset ∗ (edge → R) ∗ R monoid ∗ (vtxid → bool) ∗ (R → O option) → O vset	$O\left(\sum_{u \in U'} d(u)\right)^{\dagger}$	$O(\log n)^{\ddagger}$
	nghCount	: vset ∗ (edge → bool) ∗ (vtxid → bool) ∗ (int → O option) → O vset		

edgeMap [SB'13]

Inputs

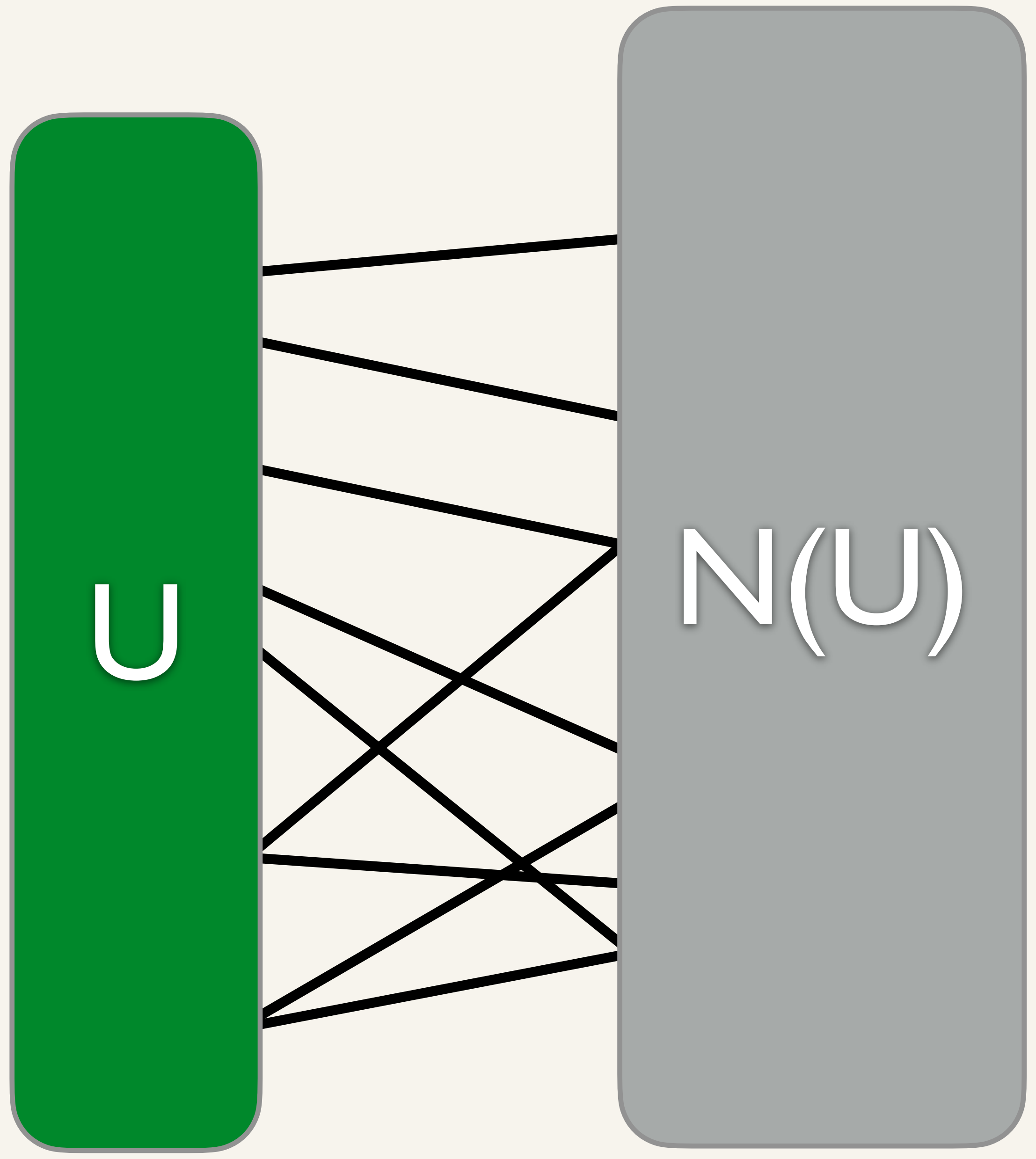
vertexSubset $\boldsymbol{U}$

Map function $\boldsymbol{F}$ **: edge $\rightarrow$ bool**

Condition function $\boldsymbol{C}$ **: vtxid $\rightarrow$ bool**

Output

vertexSubset $\boldsymbol{O}$



Consider $(u, v) \in E$ s.t. $u \in U$ and $C(v)$

If $F(u, v) =$ True return v in output, O

Operator specification doesn't insist on a particular implementation. Thus, Ligra (and GBBS) can implement direction-optimization "under the hood"

Generalizing edgeMap to Other Graph Operations

Inputs

vertexSubset U

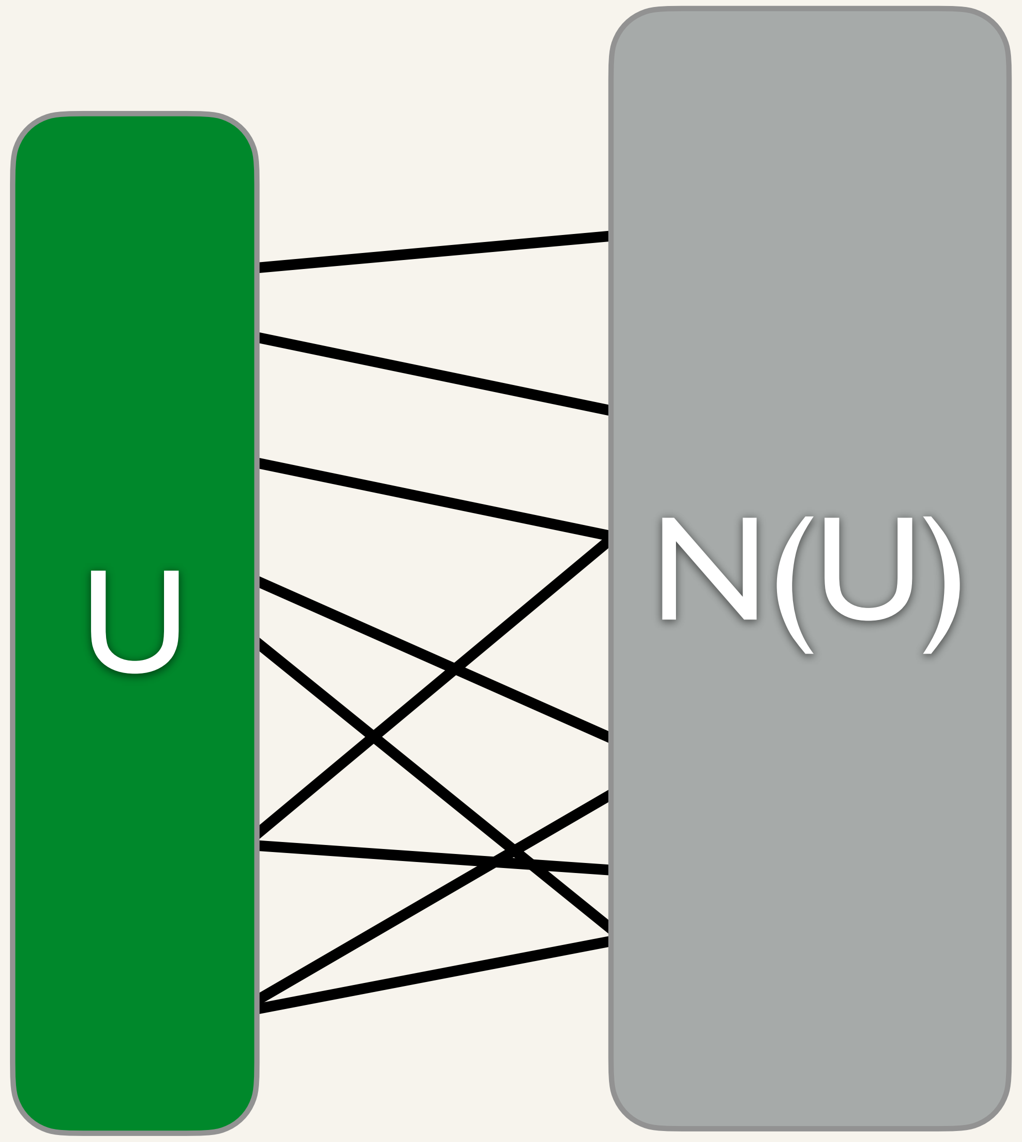
Map function F : edge → O

Combine function M : O monoid (O → O → O, identity)

Condition function C : vtxid → bool

Output

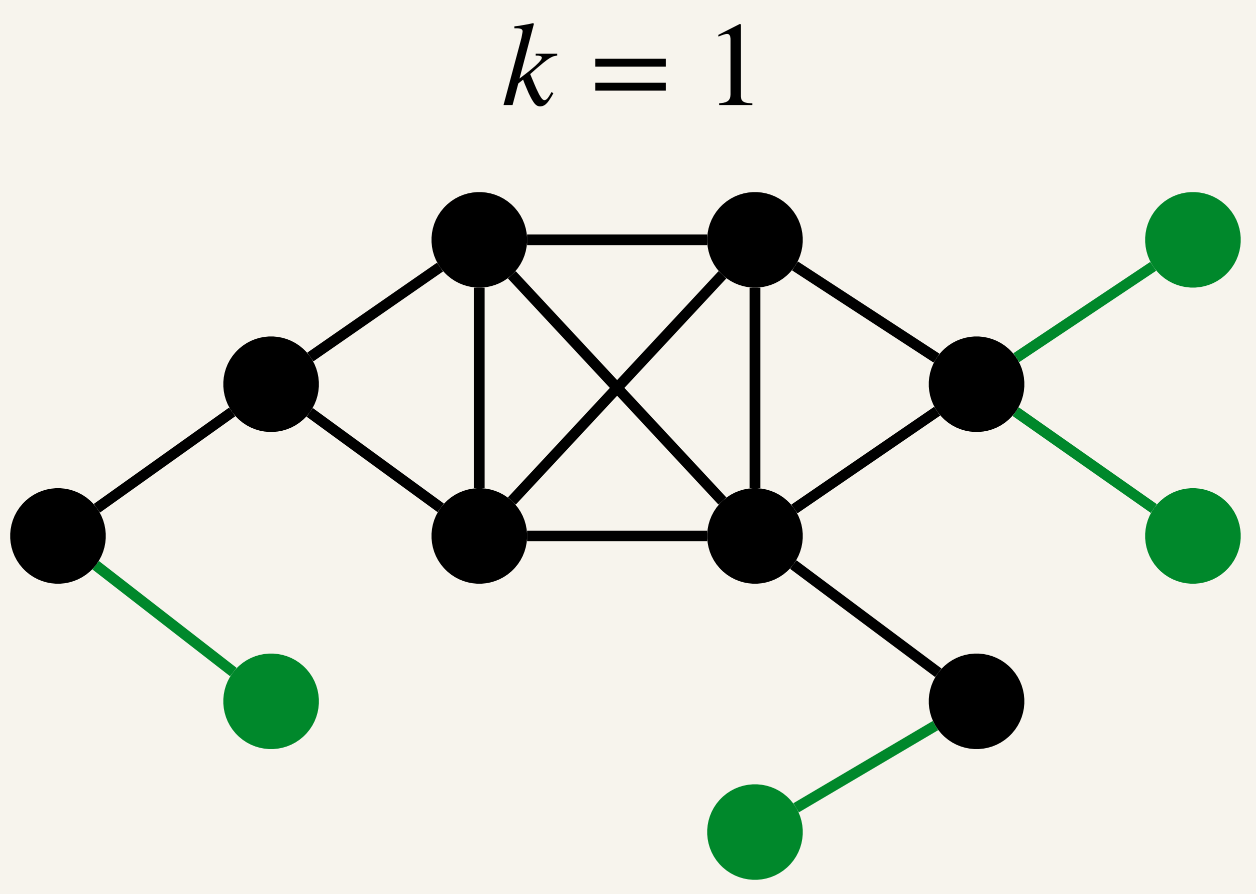
O vertexSubset R

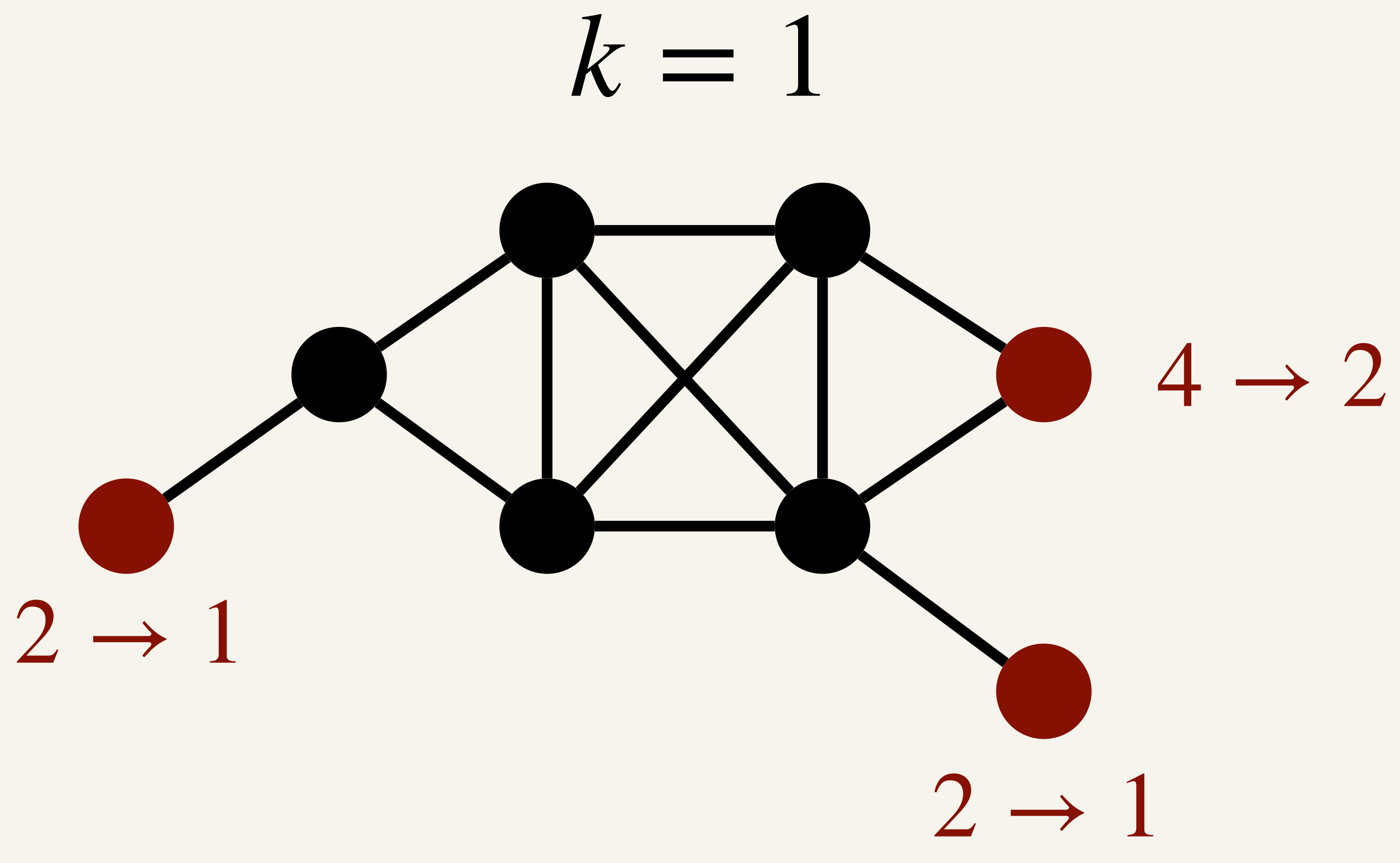


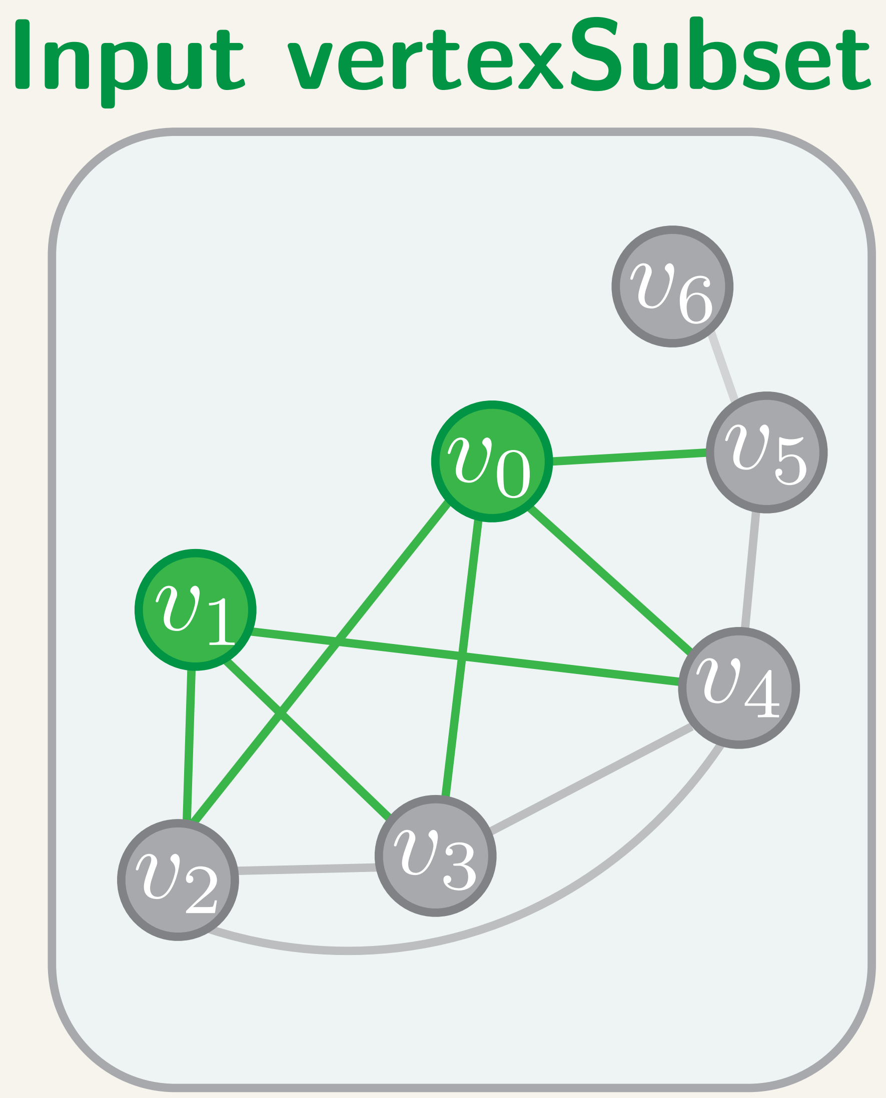
Aggregating results at the source vertices yields a **src-** version

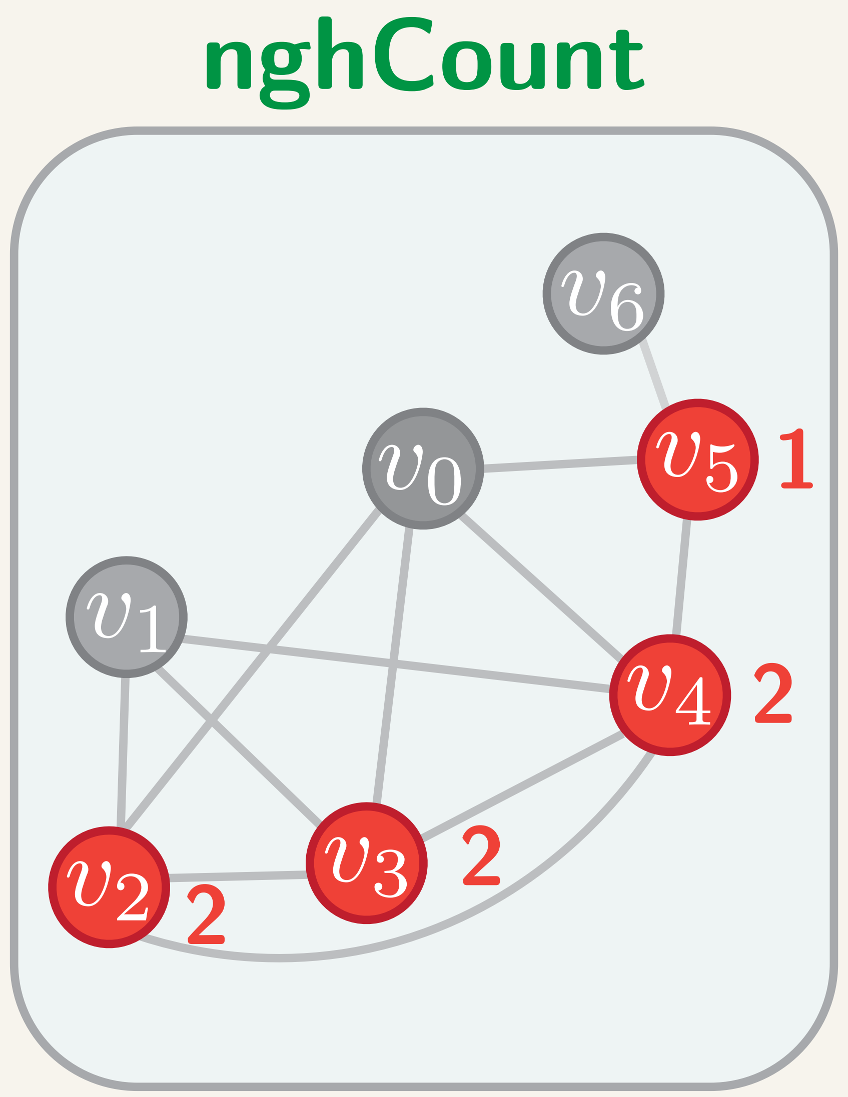
Aggregating results at the neighbor vertices yields a **ngh-** version)

Example: Updating Induced Degrees in Parallel using nghCount









Our Implementation

- We provide a provably-efficient implementation of nghCount that takes

$O\left(|U| + \sum_{u \in U} d(u) \right)$ expected work $O(\log n)$ depth *whp*

Connectivity Problems in GBBS

† : in expectation   * : whp

- Connectivity and related problems are probably the best studied problems in the parallel algorithms literature

- Practical work-efficient implementations of these problems are absent in the experimental literature

GBBS provides simple and high-level implementations of connectivity problems based on low-diameter decomposition

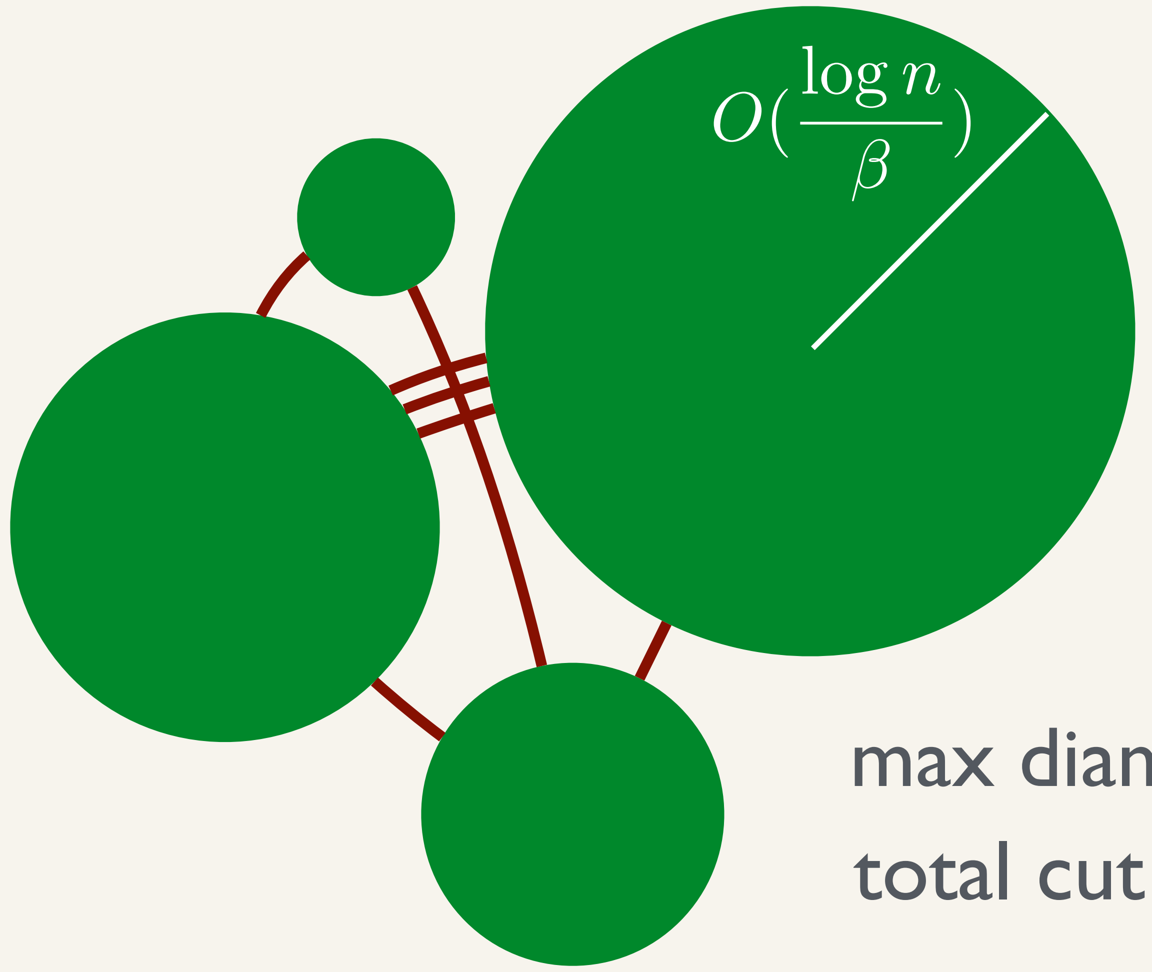
Problem	Work	Depth
Breadth-First Search (BFS)	$O(m)$	$\tilde{O}(\text{diam}(G))$
Integral-Weight SSSP (weighted BFS)	$O(m)^\dagger$	$\tilde{O}(\text{diam}(G))*$
General-Weight SSSP (Bellman-Ford)	$O(\text{diam}(G) \cdot m)$	$\tilde{O}(\text{diam}(G))$
Single-Source Widest Path (Bellman-Ford)	$O(\text{diam}(G) \cdot m)$	$\tilde{O}(\text{diam}(G))$
Single-Source Betweenness Centrality (BC)	$O(m)$	$\tilde{O}(\text{diam}(G))$
O(k)-Spanner	$O(m)$	$\tilde{O}(k \log n)*$
Low-Diameter Decomposition (LDD)	$O(m)$	$O(\log^2 n)*$
Connectivity (CC)	$O(m)^\dagger$	$O(\log^3 n)*$
Spanning Forest	$O(m)^\dagger$	$O(\log^3 n)*$
Biconnectivity	$O(m)^\dagger$	$O(\max(\text{CC}, \text{BFS}))$
Strongly Connected Components (SCC)	$O(m \log n)^\dagger$	$\tilde{O}(\text{diam}(G))*$
Minimum Spanning Forest (MSF)	$O(m \log n)$	$O(\log^2 n)$
Maximal Independent Set (MIS)	$O(m)^\dagger$	$O(\log^2 n)*$
Maximal Matching (MM)	$O(m)^\dagger$	$O(\log^2 n)*$
Graph Coloring	$O(m)$	$O(\log n + L \log \Delta)$
k-core	$O(m)^\dagger$	$O(\rho \log n)*$
Approximate Set Cover	$O(m)^\dagger$	$O(\log^3 n)*$
Triangle Counting (TC)	$O(m^{3/2})$	$O(\log n)$
Approximate Densest Subgraph	$O(m)$	$O(\log^2 n)$
PageRank Iteration	$O(n + m)$	$O(\log n)$

Connectivity Problems in GBBS using LDD

Low-Diameter Decomposition [MPX'13]

Spanning Forest [SDB'14]

Undirected Connectivity [SDB'14]



$O(m+n)$ work

$O(\log^2 n)$ depth whp

max diameter is $O(\log n/\beta)$

total cut edges $\leq \beta m$

$O(m+n)$ expected work

$O(\log^3 n)$ depth whp

Graph Contraction

Input

graph $G(V,E)$

Cluster labels F : vtxid $\rightarrow$ int

Output

graph $G'(V',E')$

Biconnectivity [SV'87]

$O(m+n)$ expected work

$O(\text{diam}(G)+\log^3 n)$ depth whp

"Hard" Problems in GBBS

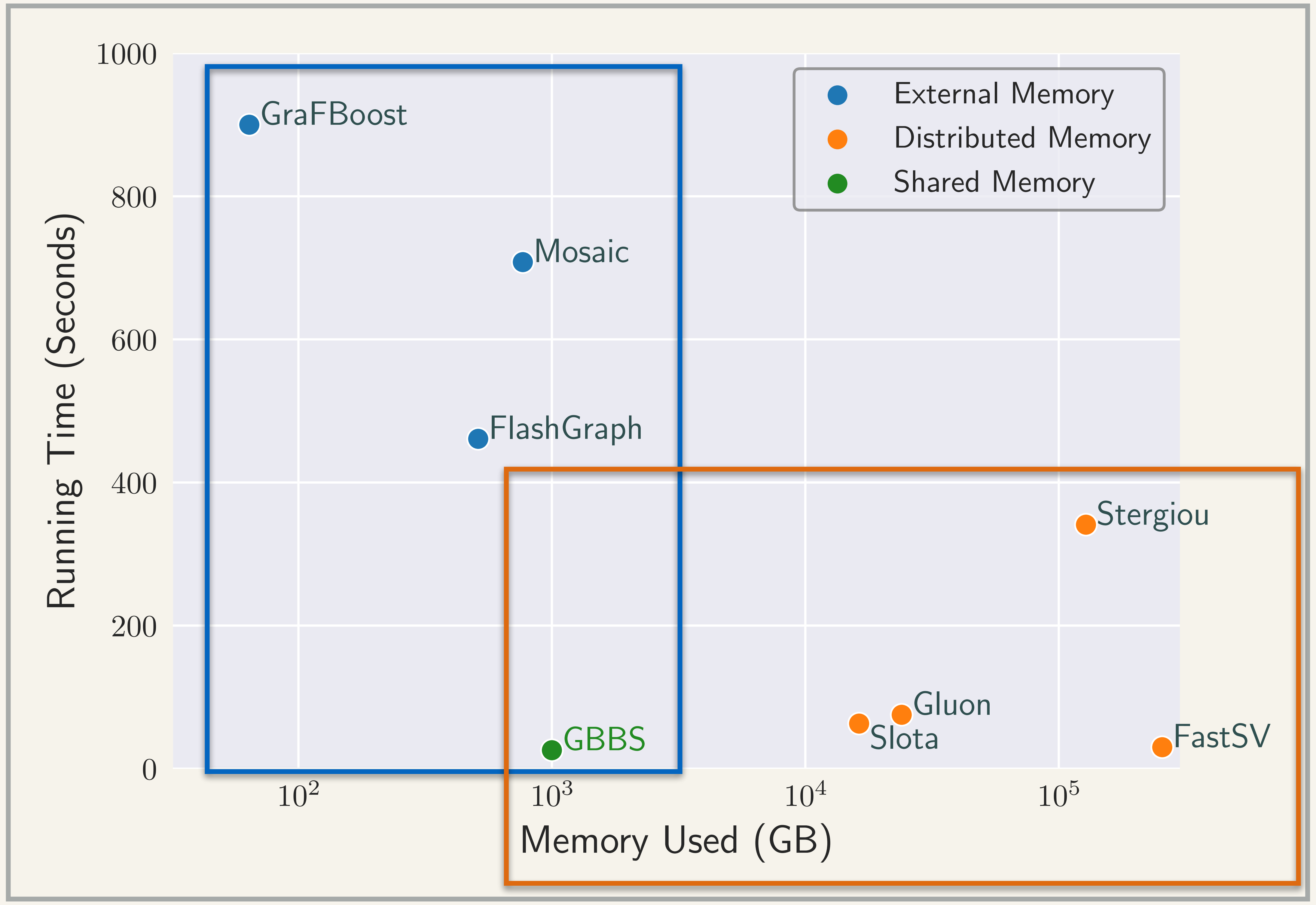
† : in expectation * : whp

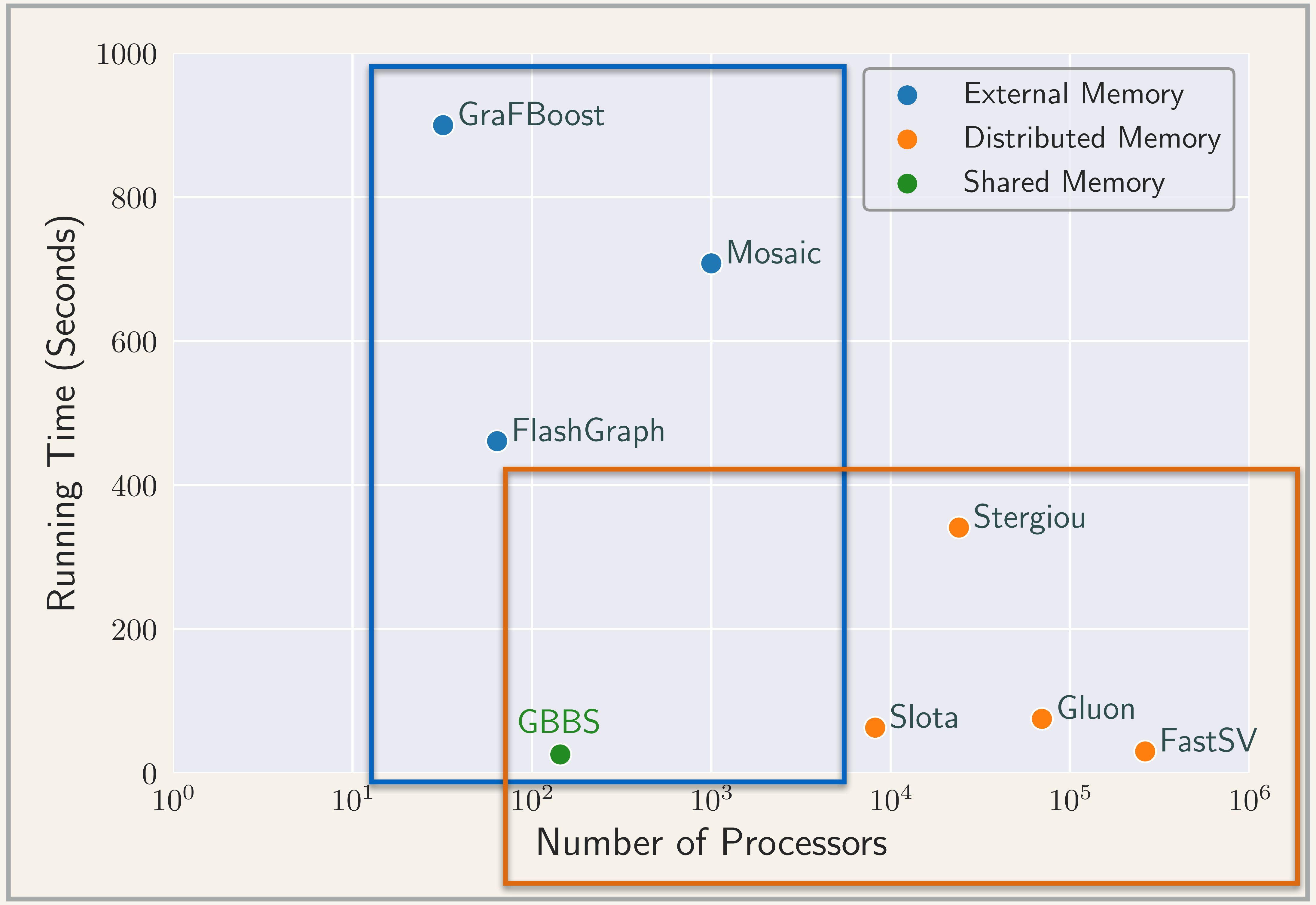
- Work-efficient, polylog depth algorithms not known for these problems
- Instead, focus on work-efficiency at the expense of parametrizing depth in terms of some other graph parameter (usually diameter)

Transitive Closure Bottleneck:
See book chapter by
Karp and Ramachandran

Problem	Work	Depth
Breadth-First Search (BFS)	$O(m)$	$\tilde{O}(\text{diam}(G))$
Integral-Weight SSSP (weighted BFS)	$O(m)^\dagger$	$\tilde{O}(\text{diam}(G))^*$
General-Weight SSSP (Bellman-Ford)	$O(\text{diam}(G) \cdot m)$	$\tilde{O}(\text{diam}(G))$
Single-Source Widest Path (Bellman-Ford)	$O(\text{diam}(G) \cdot m)$	$\tilde{O}(\text{diam}(G))$
Single-Source Betweenness Centrality (BC)	$O(m)$	$\tilde{O}(\text{diam}(G))$
O(k)-Spanner	$O(m)$	$\tilde{O}(k \log n)^*$
Low-Diameter Decomposition (LDD)	$O(m)$	$O(\log^2 n)^*$
Connectivity (CC)	$O(m)^\dagger$	$O(\log^3 n)^*$
Spanning Forest	$O(m)^\dagger$	$O(\log^3 n)^*$
Biconnectivity	$O(m)^\dagger$	$O(\max(\text{CC}, \text{BFS}))$
Strongly Connected Components (SCC)	$O(m \log n)^\dagger$	$\tilde{O}(\text{diam}(G))^*$
Minimum Spanning Forest (MSF)	$O(m \log n)$	$O(\log^2 n)$
Maximal Independent Set (MIS)	$O(m)^\dagger$	$O(\log^2 n)^*$
Maximal Matching (MM)	$O(m)^\dagger$	$O(\log^2 n)^*$
Graph Coloring	$O(m)$	$O(\log n + L \log \Delta)$
k-core	$O(m)^\dagger$	$O(\rho \log n)^*$
Approximate Set Cover	$O(m)^\dagger$	$O(\log^3 n)^*$
Triangle Counting (TC)	$O(m^{3/2})$	$O(\log n)$
Approximate Densest Subgraph	$O(m)$	$O(\log^2 n)$
PageRank Iteration	$O(n + m)$	$O(\log n)$

Case Study: Connectivity on WebDataCommons Graph





Outperform external memory results by orders of magnitude using comparable hardware.

Outperform distributed memory results using orders of magnitude less hardware.

Recent Results that use GBBS

SAGE
Semi-Asymmetric Graph Engine

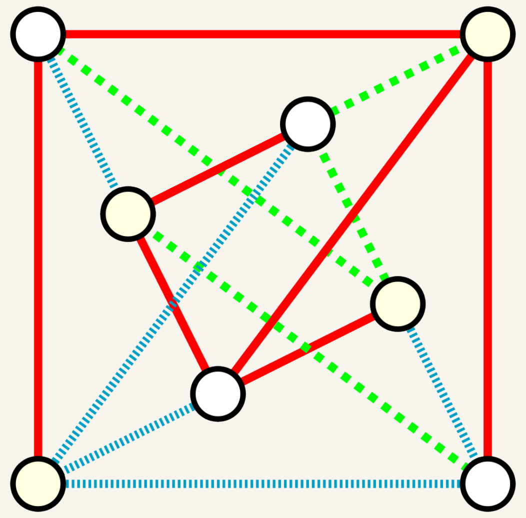
Design extensions of GBBS algorithms to a semi-asymmetric setting for NVRAM machines, and achieve state-of-the-art running times (*VLDB'20)*

with Charles McGuffey, Hongbo Kang, Yan Gu, Guy Blelloch, Phil Gibbons, and Julian Shun

ConnectIt

Framework for parallel connectivity, spanning forest, and incremental connectivity (VLDB'21)

with Changwan Hong and Julian Shun



ArbClique

Implement state-of-the-art k-clique counting (exact+approximate), and k-clique densest-subgraph algorithms in GBBS (*in submission*)

with Jessica Shi and Julian Shun

Lots of other ongoing work!

Efficient parallel graph algorithms for motifs (cycles, cliques)

Shared-memory parallel graph embedding

Parallel Graph Clustering (SCAN, Hierarchical Agglomerative Clustering)

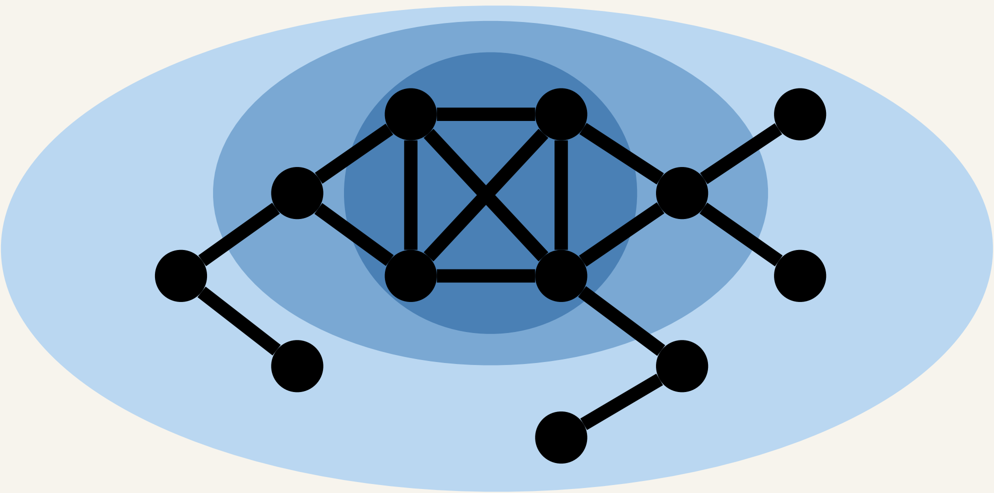
Parallel Batch-Dynamic k-Core Decomposition, Triangle Counting

Summary: Julienne

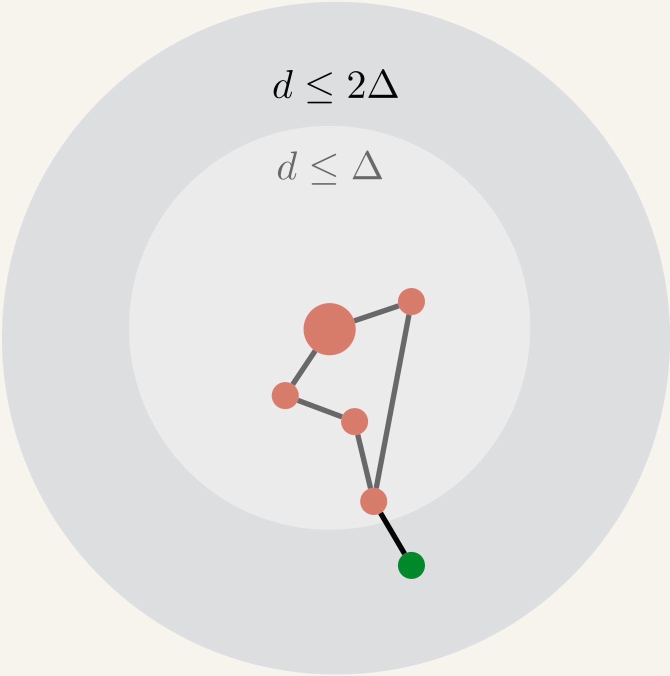
Julienne: framework for *bucketing-based algorithms*

- Codes:
  - Simple (< 100 lines each)
  - Theoretically efficient (strong bounds on work and depth)
  - Good performance in practice
  - Code included as part of the GBBS library

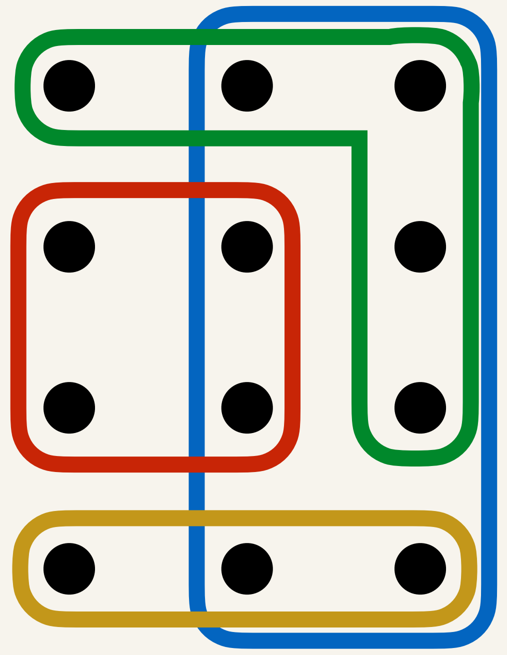
k-core



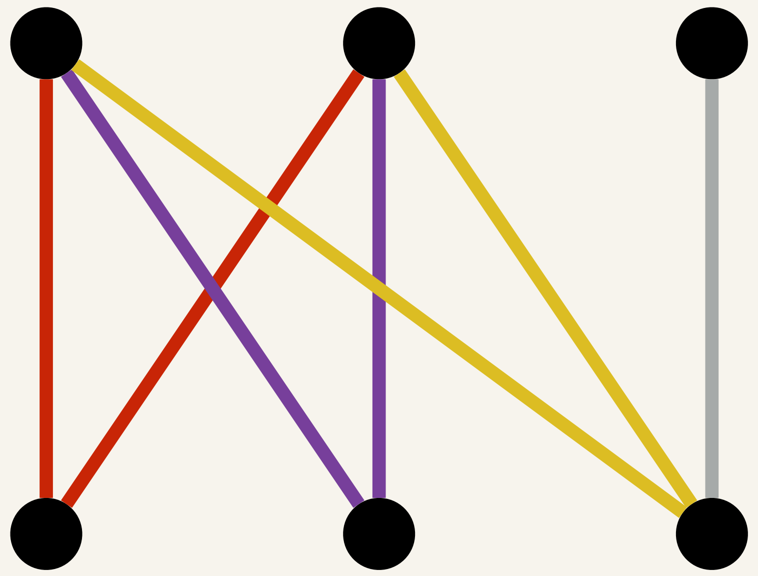
Delta-stepping wBFS



Parallel Approximate Set Cover



Parallel k-Tip Decomposition



Summary: The Graph-Based Benchmark Suite (GBBS)

- Introduced a benchmark suite for graph problems with over 20 important problems
- Introduced GBBS library, which extends Ligra, Ligra+, and Julienne to simplify expressing theoretically-efficient parallel graph algorithms
- GBBS implementations achieve state-of-the-art results on the largest publicly available graphs

Connectivity Problems

Low-Diameter Decomposition
Connectivity
Spanning Forest
Biconnectivity
Minimum Spanning Forest
Strongly Connected Components

Subgraph Problems

k-Core Decomposition
k-Truss Decomposition
Apx. Densest Subgraph
Triangle Counting
Higher-Clique Counting

Covering Problems

Maximal Ind. Set
Maximal Matching
Apx. Set Cover
Graph Coloring

Shortest Path Problems

Breadth-First Search
Betweenness Centrality
Bellman-Ford
General Weight SSSP
Integral Weight SSSP
SS Widest Path
k-Spanner

Eigenvector Problems

PageRank
Personalized PageRank
Personalized SimRank

github.com/paralg/gbbs