Julienne: A Framework for Parallel Graph Algorithms using Work-efficient Bucketing

Laxman Dhulipala

Joint work with Guy Blelloch and Julian Shun

SPAA 2017

Giant graph datasets

Graph	|V|	|E| (symmetrized)
com-Orkut	3M	234M
Twitter	41M	1.46B
Friendster	124M	3.61B
Hyperlink2012-Host	101M	2.04B
Facebook (2011)	721M	68.4B
Hyperlink2014	1.7B	124B
Hyperlink2012	3.5B	225B
Facebook (2017)	> 2B	> 300B
Google (2017)	?	?

● : Publicly available graphs used in our experiments

● : Private graph datasets

Traditional approaches

One possible way to solve large graph problems:

- Hand-write MPI/OpenMP/Cilk codes
- Run a powerful machine or a large cluster

Traditional approaches

One possible way to solve large graph problems:

- Hand-write MPI/OpenMP/Cilk codes
- Run a powerful machine or a large cluster

Benefits

- Good performance
- Can hand-code custom optimizations

Traditional approaches

One possible way to solve large graph problems:

- Hand-write MPI/OpenMP/Cilk codes
- Run a powerful machine or a large cluster

Benefits

- Good performance
- Can hand-code custom optimizations

Downsides

- Usually require a lot of code
- Need lots of expertise to write and understand codes
- Not everyone has a supercomputer

Graph processing frameworks

High level goals

- Simple set of primitives (interface)
- Implementations easy to write and understand
- Algorithms can handle very large graphs

Graph processing frameworks

High level goals

- Simple set of primitives (interface)
- Implementations easy to write and understand
- Algorithms can handle very large graphs

Ex: Pregel, GraphLab, Ligra, GraphX, GraphChi…

Graph processing frameworks

High level goals

- Simple set of primitives (interface)
- Implementations easy to write and understand
- Algorithms can handle very large graphs

Ex: Pregel, GraphLab, Ligra, GraphX, GraphChi…

Additional goals:

- Primitives have theoretical guarantees
- Support common optimizations “under the hood”
- Implementations competitive with non-framework codes

Graph processing frameworks

High level goals

- Simple set of primitives (interface)
- Implementations easy to write and understand
- Algorithms can handle very large graphs

Ex: Pregel, GraphLab, Ligra, GraphX, GraphChi…

Additional goals:

- Primitives have theoretical guarantees
- Support common optimizations “under the hood”
- Implementations competitive with non-framework codes

Our goals:

- All of the above on a single *affordable shared memory machine*

An “affordable” machine

An “affordable” machine

Dell PowerEdge R930

- 72-cores (4 x 2.4GHz 18-core E7-8867 v4 Xeon processors)
- 1TB of main memory
- Costs less than a mid-range BMW

An “affordable” machine

Dell PowerEdge R930

- 72-cores (4 x 2.4GHz 18-core E7-8867 v4 Xeon processors)
- 1TB of main memory
- Costs less than a mid-range BMW

Ligra

Shared memory graph processing framework [1]

[1] Shun and Blelloch, 2013, Ligra: A Lightweight Graph Processing Framework for Shared Memory

[2] Shun, Dhulipala and Blelloch, 2013, Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+

Ligra

Shared memory graph processing framework [1]

Benefits

- Designed to express frontier-based algorithms
- Primitives and implementations have theoretical guarantees
- Optimizations (direction-optimizing, compression [2])
- Implementations are simple to write and understand
- Competitive with hand-tuned codes

[1] Shun and Blelloch, 2013, Ligra: A Lightweight Graph Processing Framework for Shared Memory

[2] Shun, Dhulipala and Blelloch, 2013, Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+

Ligra

Shared memory graph processing framework [1]

Benefits

- Designed to express frontier-based algorithms
- Primitives and implementations have theoretical guarantees
- Optimizations (direction-optimizing, compression [2])
- Implementations are simple to write and understand
  - Competitive with hand-tuned codes

Downsides

- Some algorithms may not be efficiently implementable

[1] Shun and Blelloch, 2013, Ligra: A Lightweight Graph Processing Framework for Shared Memory

[2] Shun, Dhulipala and Blelloch, 2013, Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+

Ligra

Shared memory graph processing framework [1]

Benefits

- Designed to express frontier-based algorithms
- Primitives and implementations have theoretical guarantees
- Optimizations (direction-optimizing, compression [2])
- Implementations are simple to write and understand
- Competitive with hand-tuned codes

Downsides

- Some algorithms may not be efficiently implementable

This work: Made Ligra codes run on the largest publicly available graphs on a single machine

[1] Shun and Blelloch, 2013, Ligra: A Lightweight Graph Processing Framework for Shared Memory

[2] Shun, Dhulipala and Blelloch, 2013, Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+

Ligra: Frontier-based algorithms

Primitives

- Frontier data-structure (vertexSubset)
- Map over vertices in a frontier
- Map over out-edges of a frontier

Ligra: Frontier-based algorithms

Primitives

- Frontier data-structure (vertexSubset)
- Map over vertices in a frontier
- Map over out-edges of a frontier

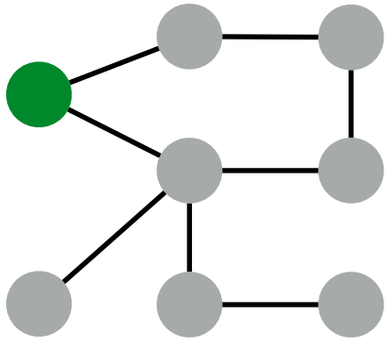
Example: Breadth-First Search

Ligra: Frontier-based algorithms

Primitives

- Frontier data-structure (vertexSubset)
- Map over vertices in a frontier
- Map over out-edges of a frontier

Example: Breadth-First Search



Round 1

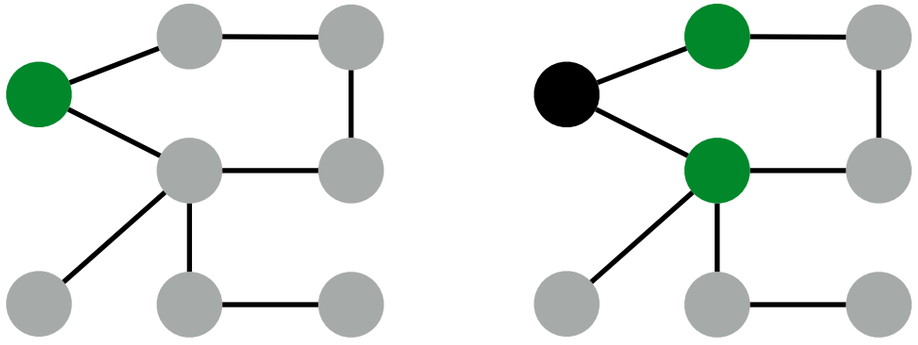
: in frontier : unvisited : visited

Ligra: Frontier-based algorithms

Primitives

- Frontier data-structure (vertexSubset)
- Map over vertices in a frontier
- Map over out-edges of a frontier

Example: Breadth-First Search



Round 1

Round 2

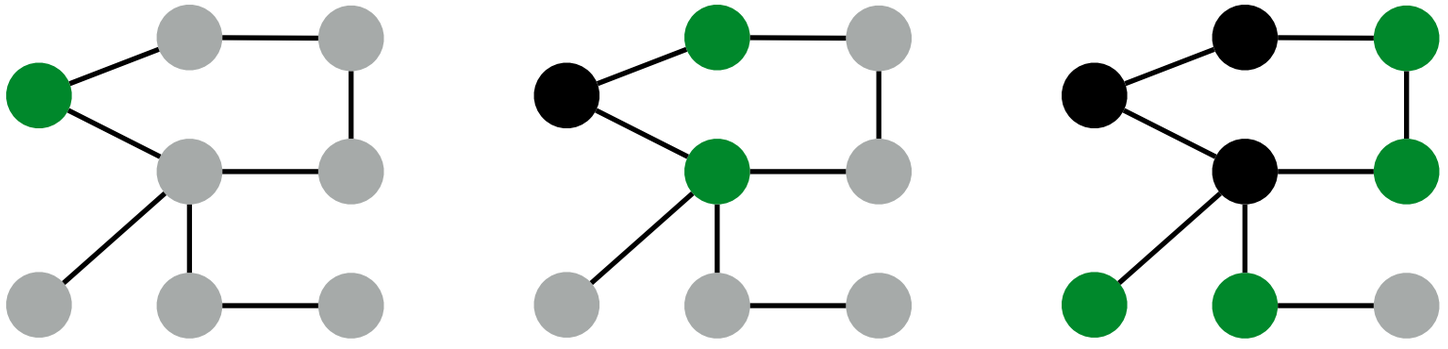
: in frontier : unvisited : visited

Ligra: Frontier-based algorithms

Primitives

- Frontier data-structure (vertexSubset)
- Map over vertices in a frontier
- Map over out-edges of a frontier

Example: Breadth-First Search

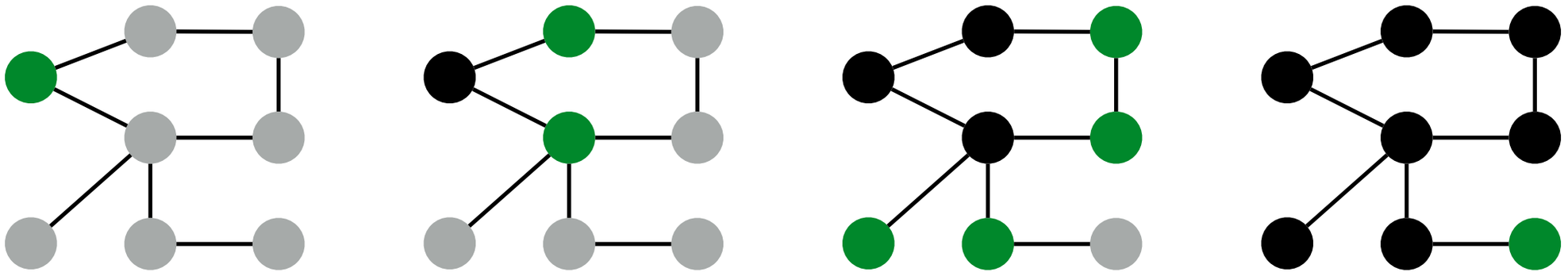


Ligra: Frontier-based algorithms

Primitives

- Frontier data-structure (vertexSubset)
- Map over vertices in a frontier
- Map over out-edges of a frontier

Example: Breadth-First Search

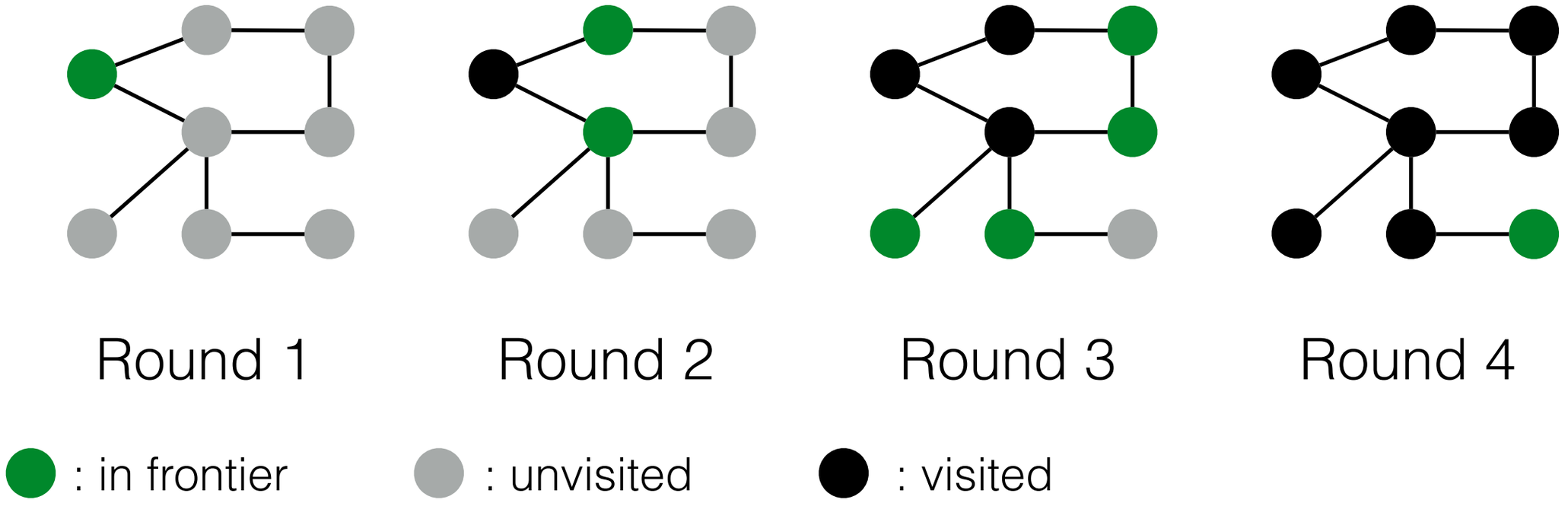


Ligra: Frontier-based algorithms

Primitives

- Frontier data-structure (vertexSubset)
- Map over vertices in a frontier
- Map over out-edges of a frontier

Example: Breadth-First Search

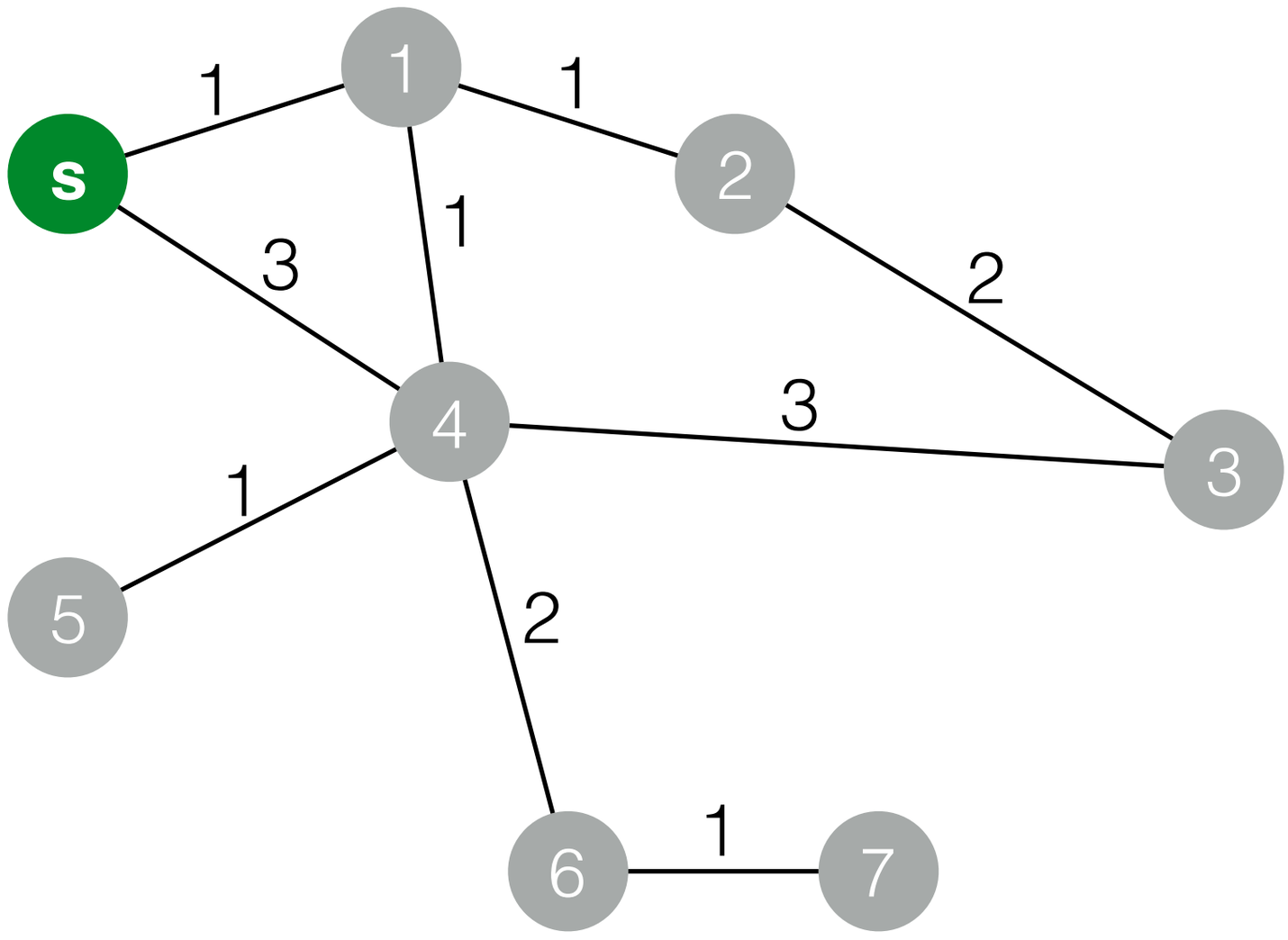


Some useful graph algorithms cannot be efficiently implemented in frontier-based frameworks

Example: Weighted Breadth-First Search

Given: $G = (V, E, w)$ with *positive integer edge weights*, $\mathsf{s} \subseteq V$

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

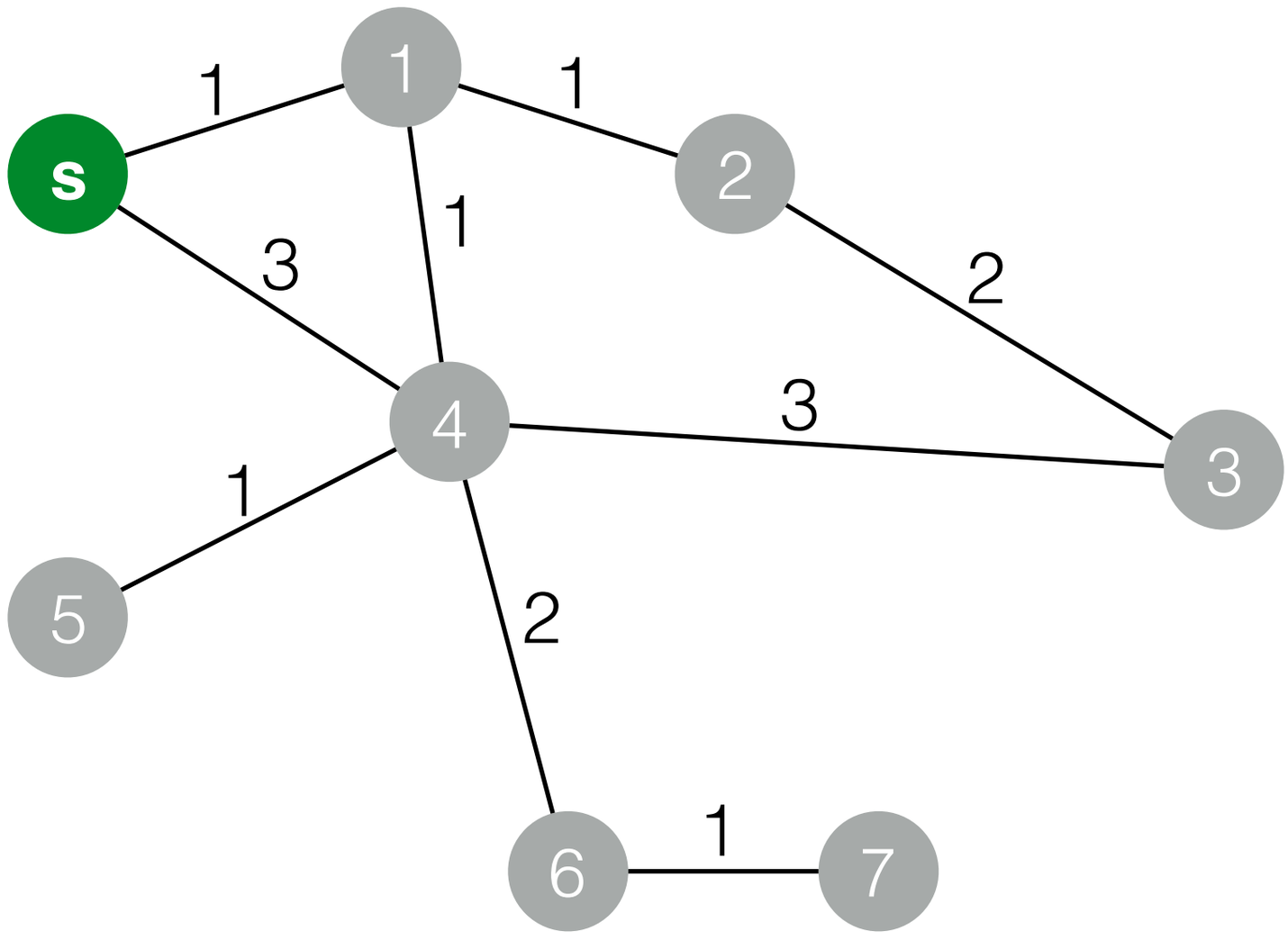


Example: Weighted Breadth-First Search

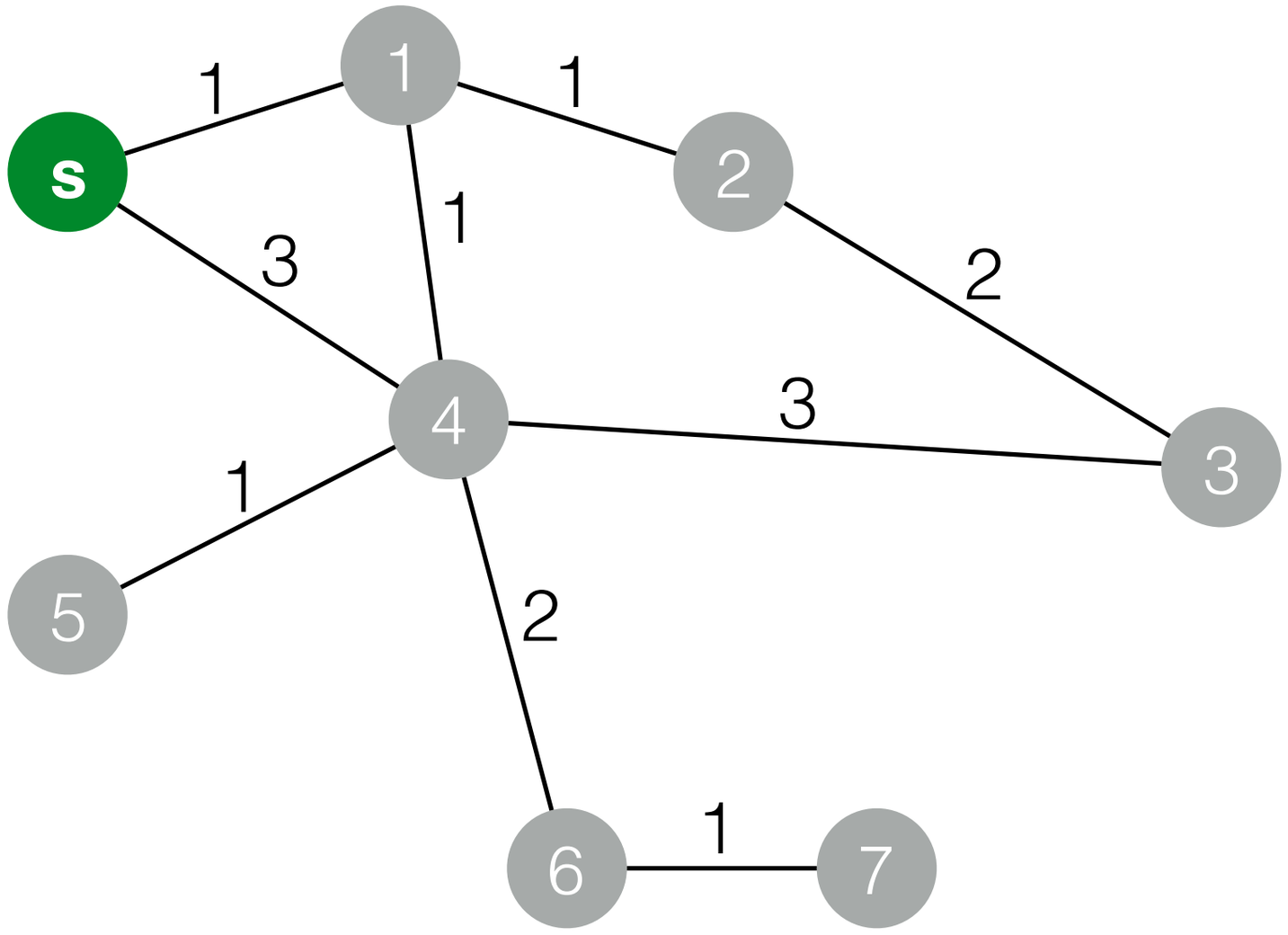
Given: $G = (V, E, w)$ with *positive integer edge weights*, $\mathsf{s} \subseteq V$

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

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



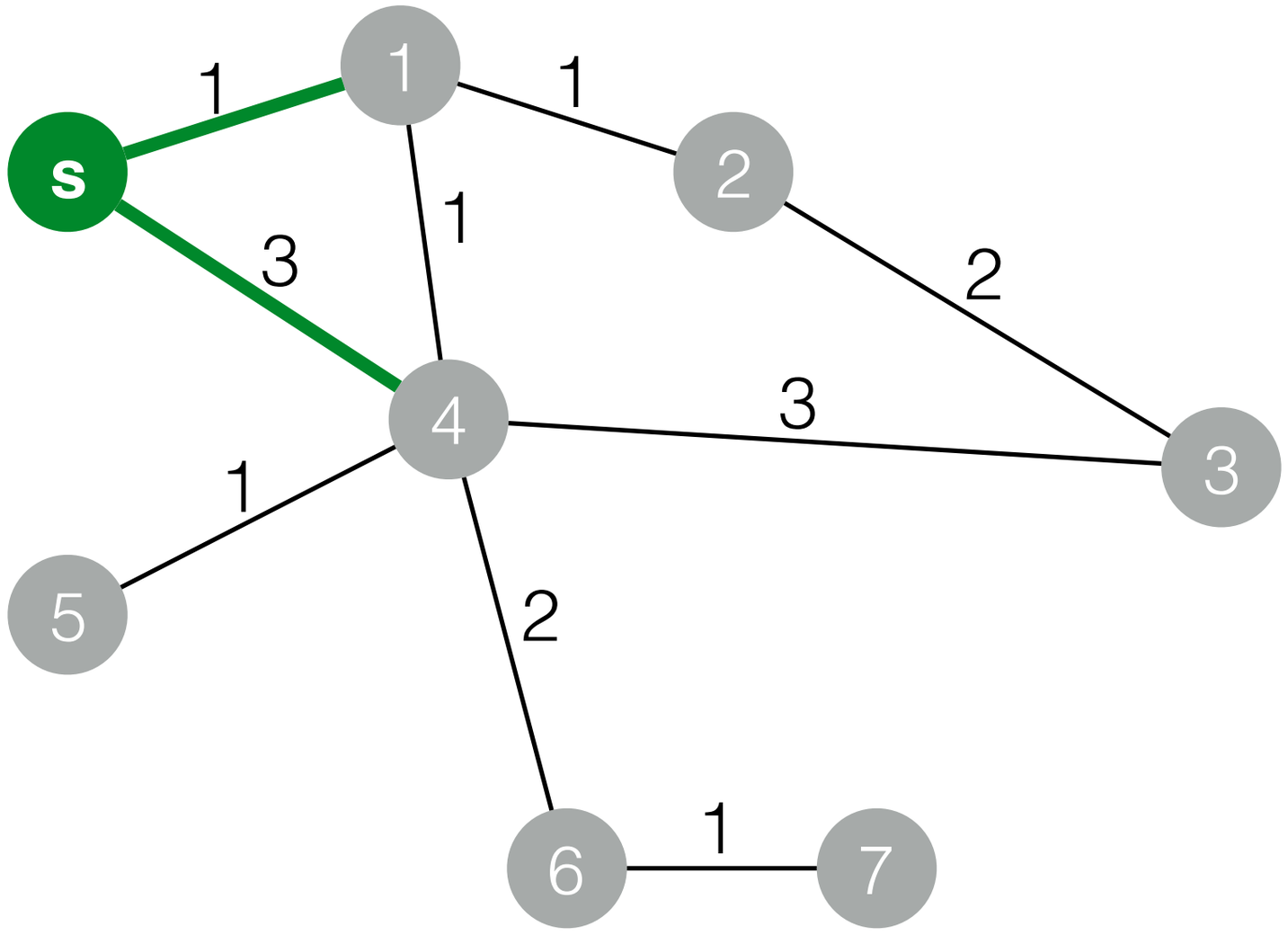
Example: Weighted Breadth-First Search



Frontier: s

Round 1

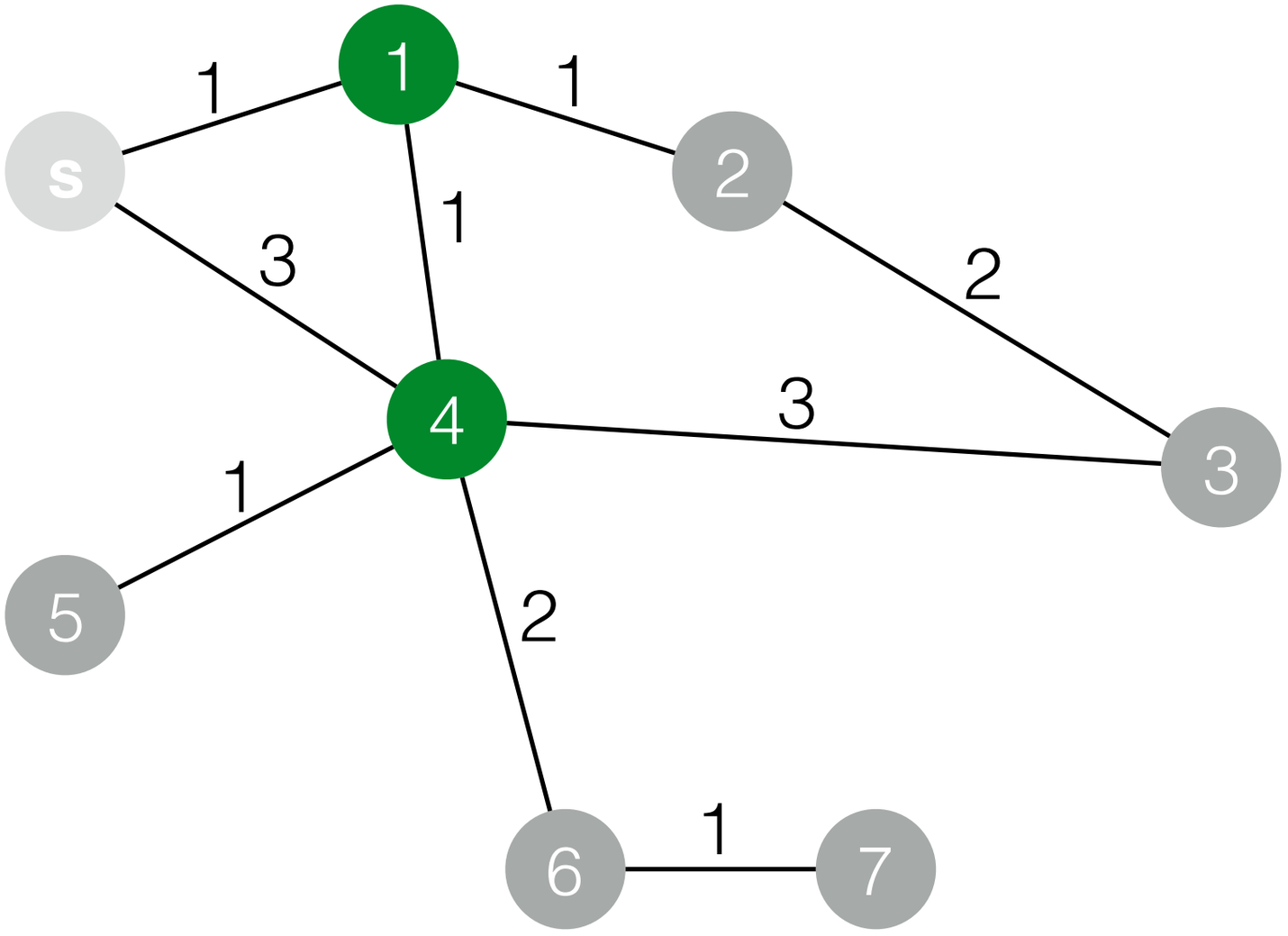
Example: Weighted Breadth-First Search



Frontier: s

Round 1

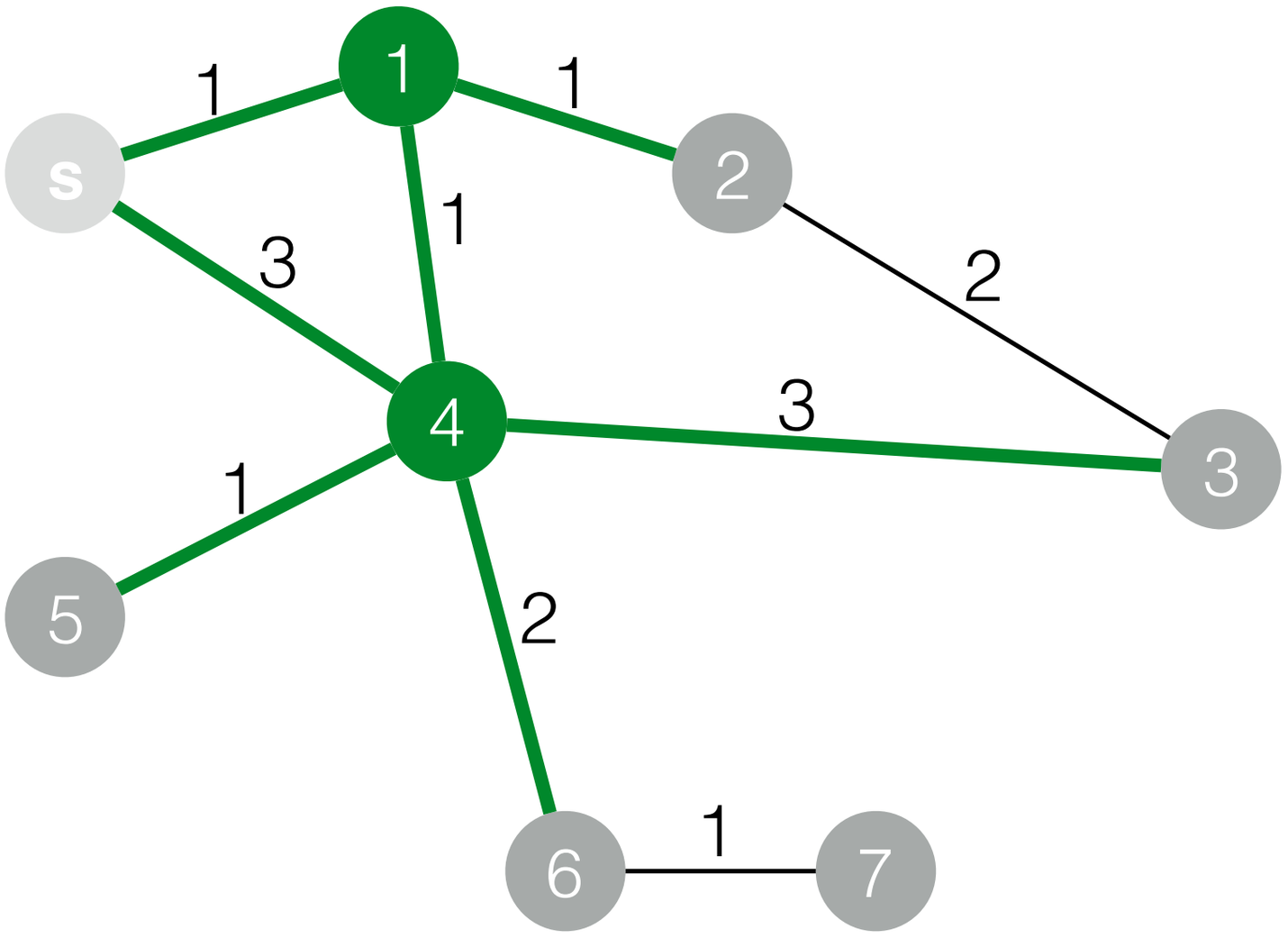
Example: Weighted Breadth-First Search



Frontier: 1 4

Round 2

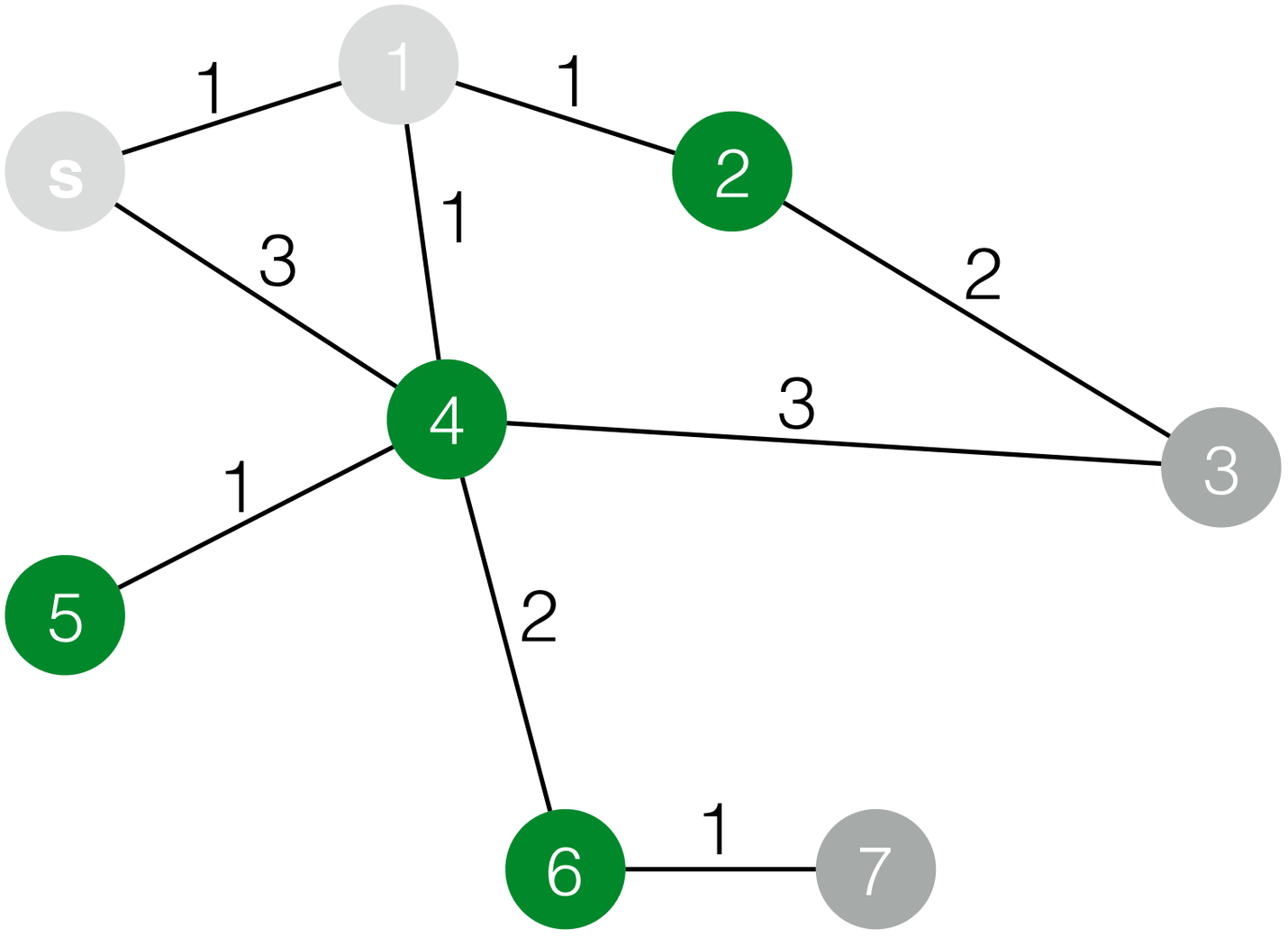
Example: Weighted Breadth-First Search



Frontier: 1 4

Round 2

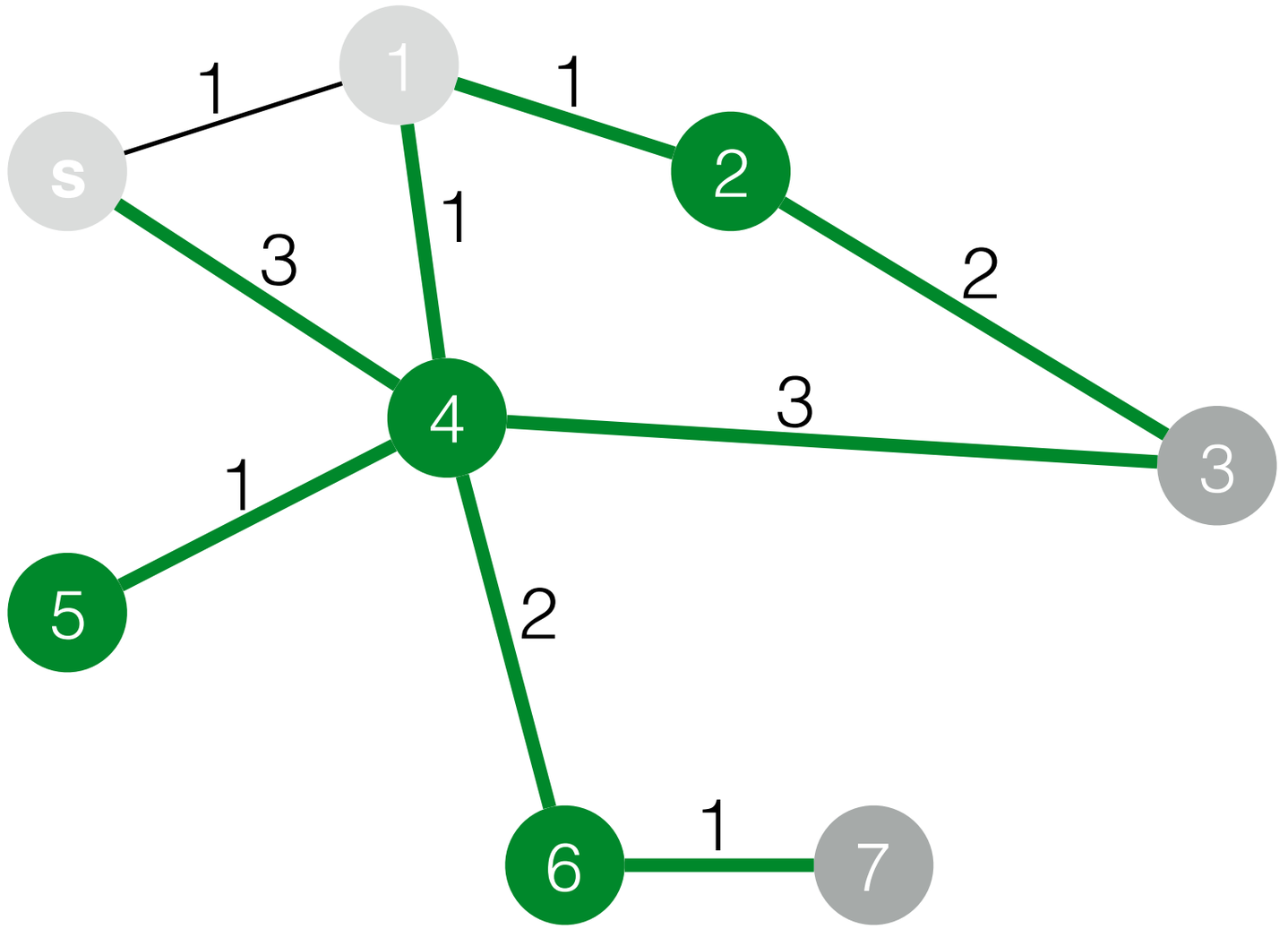
Example: Weighted Breadth-First Search



Frontier: 2 4 5 6

Round 3

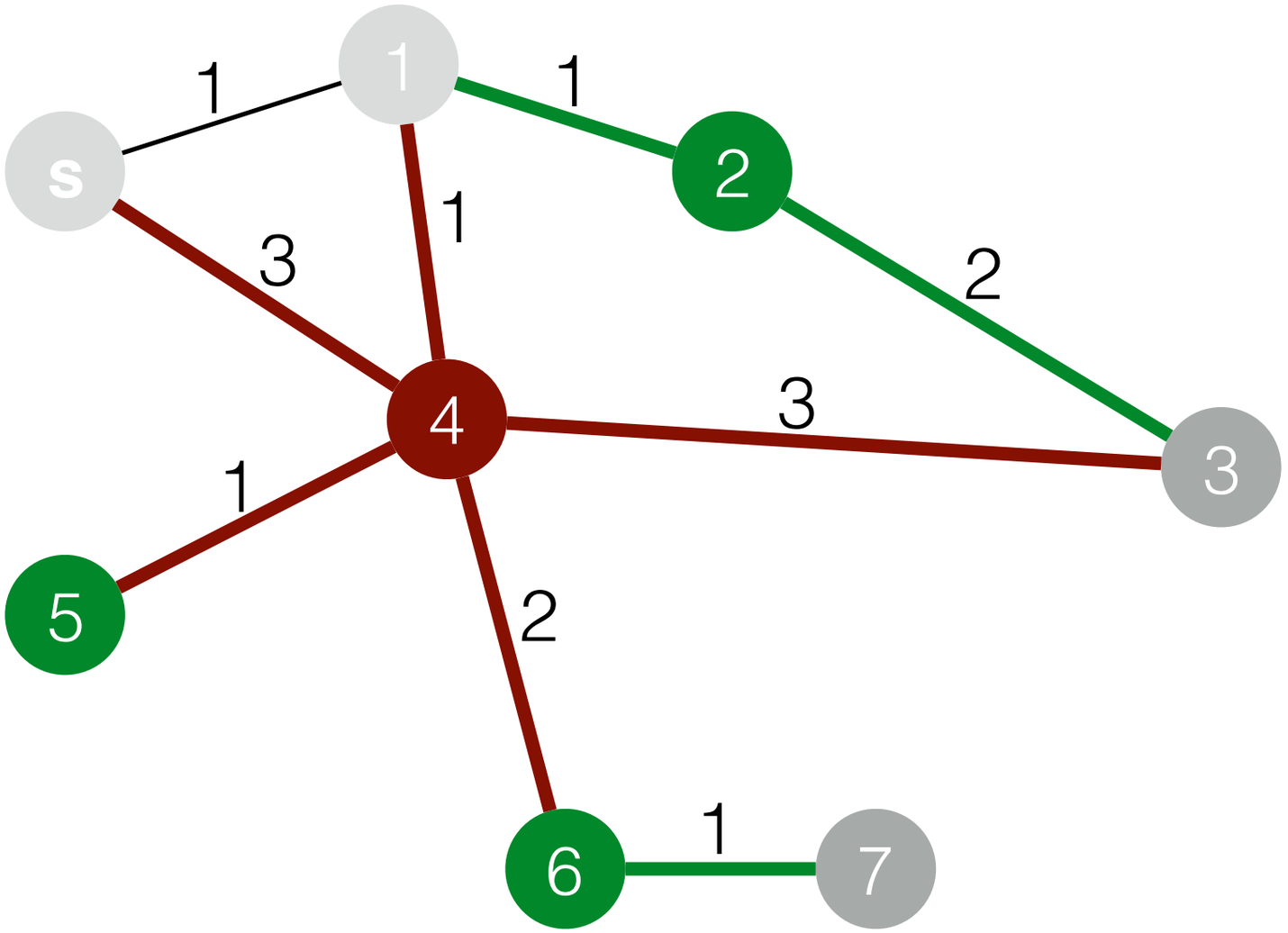
Example: Weighted Breadth-First Search



Frontier: 2 4 5 6

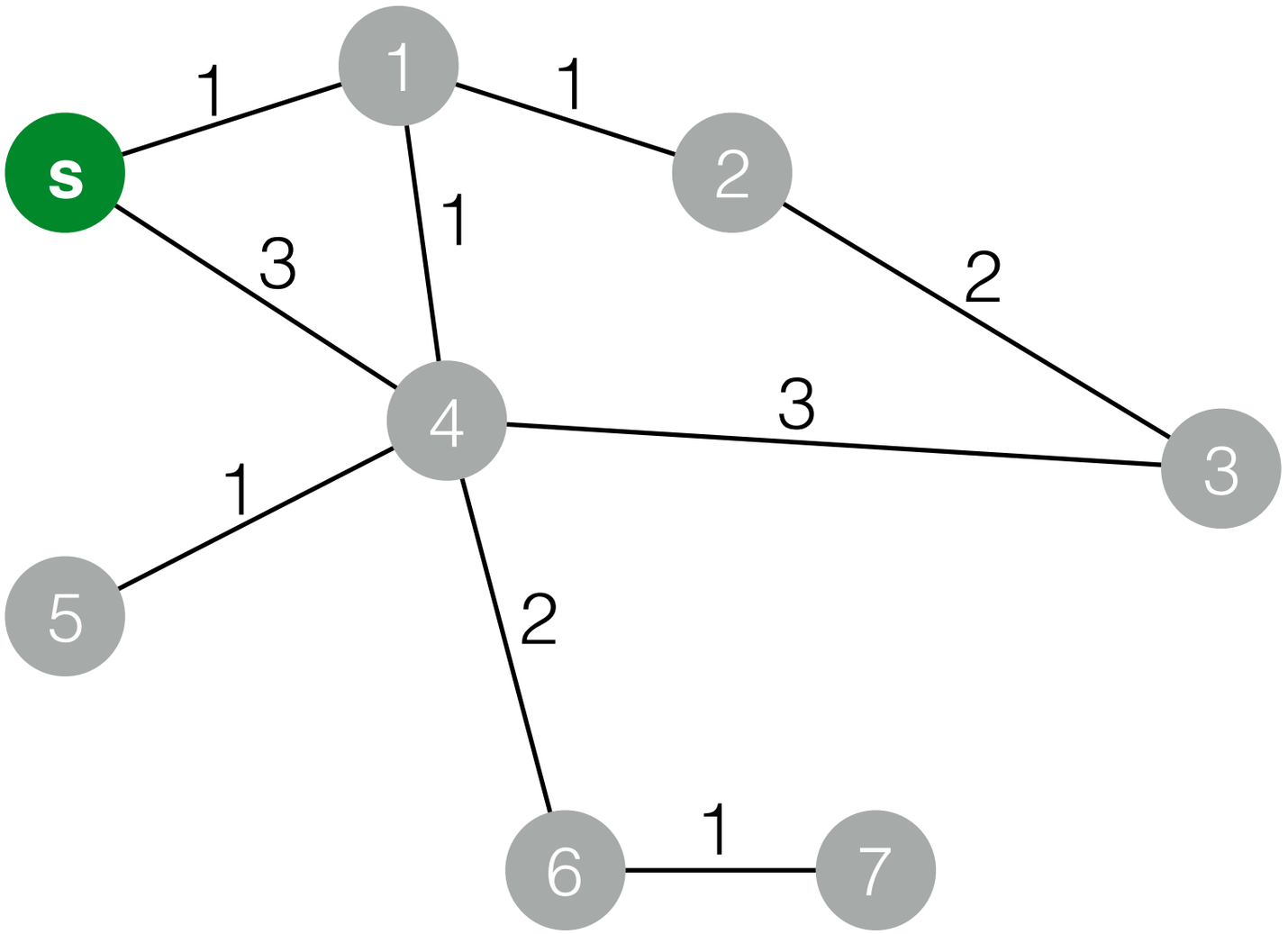
Round 3

Example: Weighted Breadth-First Search



Not work-efficient!

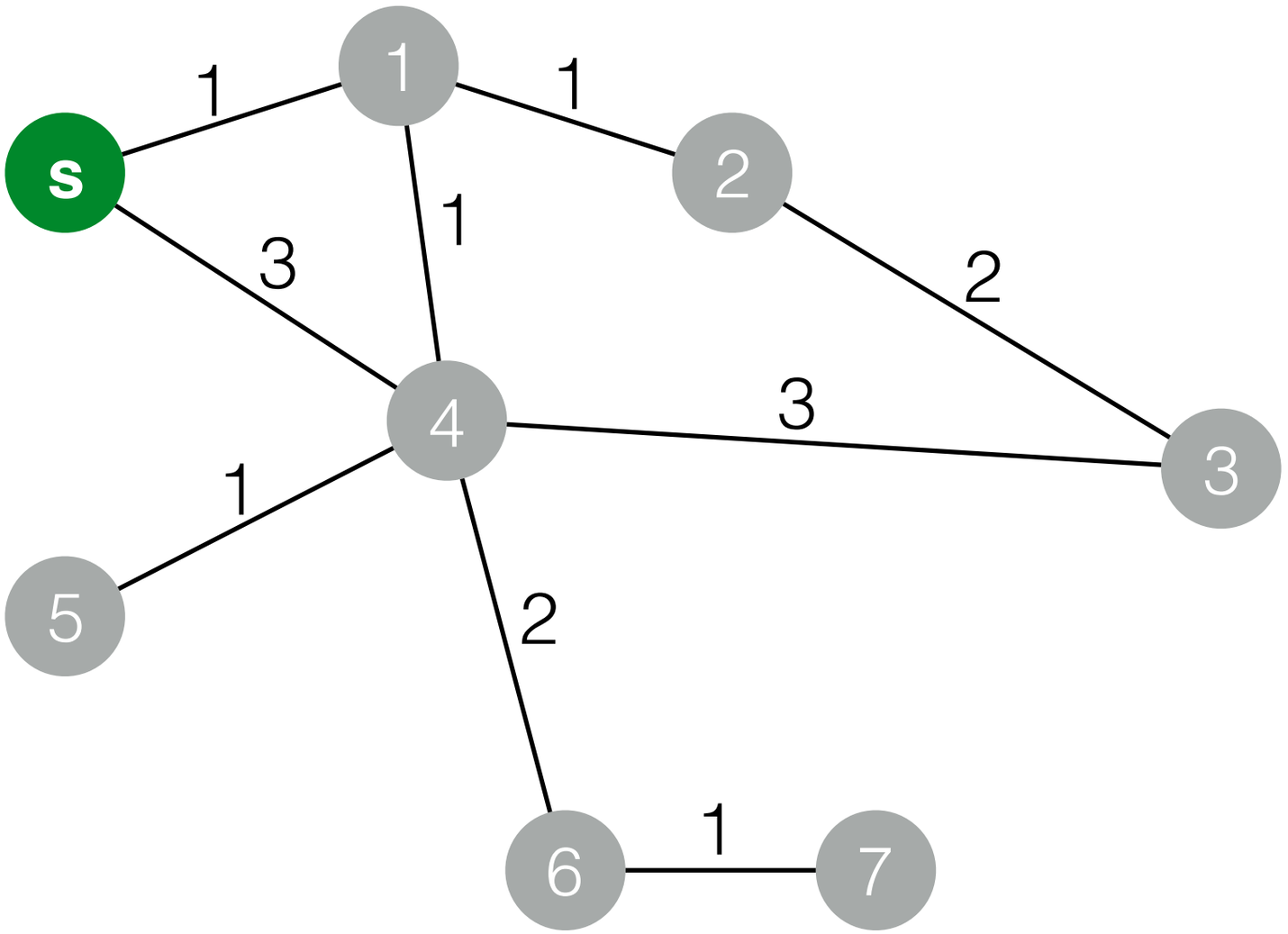
Sequential Weighted Breadth-First Search

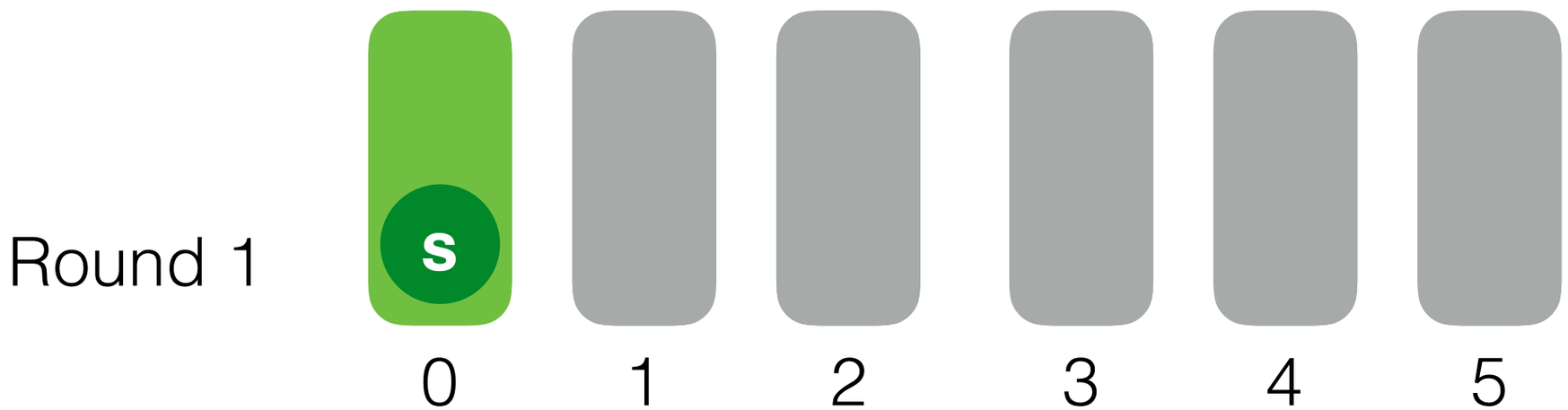


Idea:

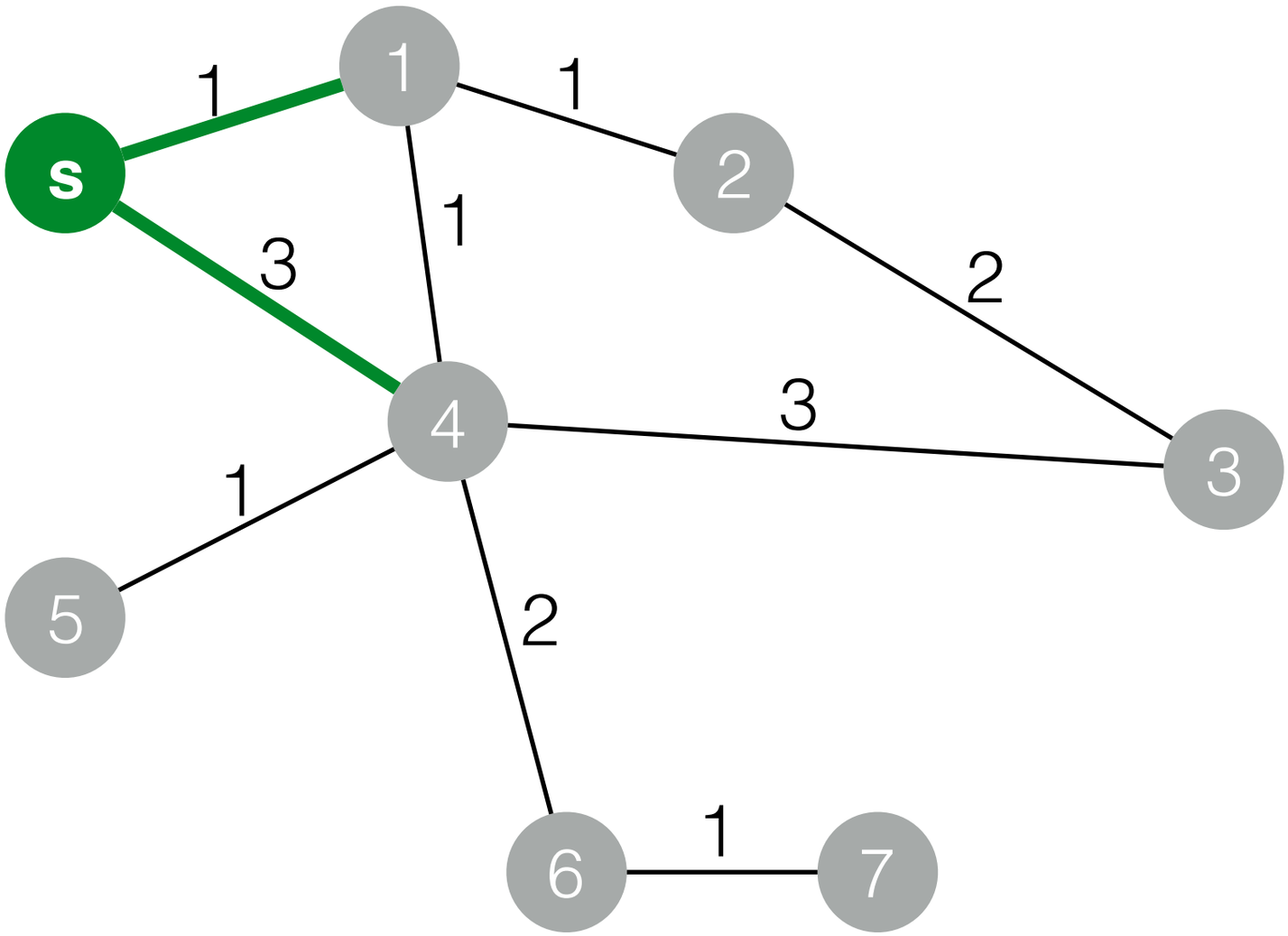
- Run Dijkstra's algorithm, but use *buckets* instead of a PQ
- Represent buckets using dynamic arrays
- Simple, efficient implementation running in $O(D + |E|)$ work

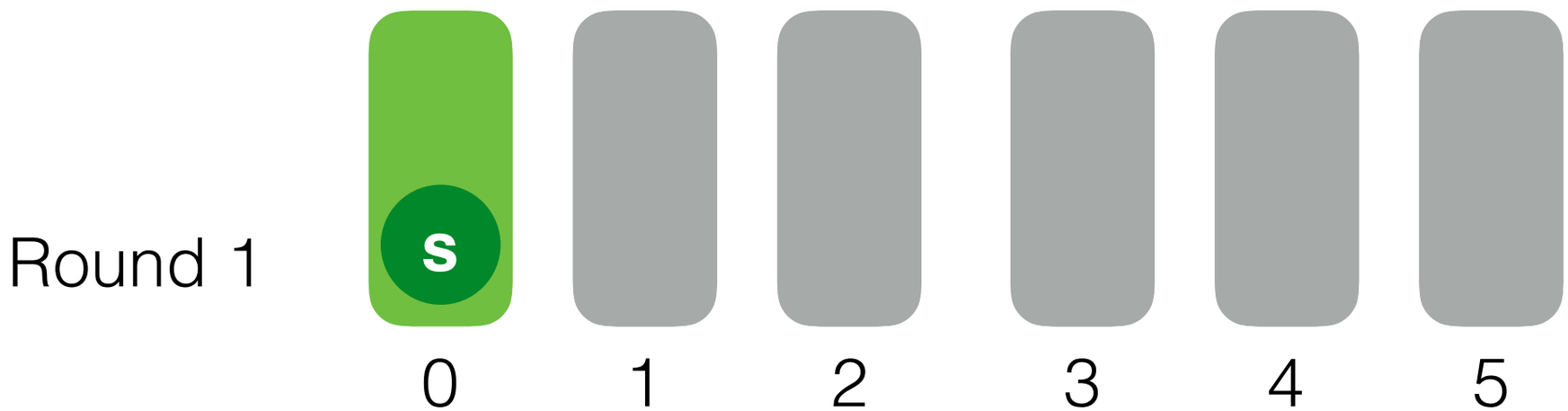
Sequential Weighted Breadth-First Search



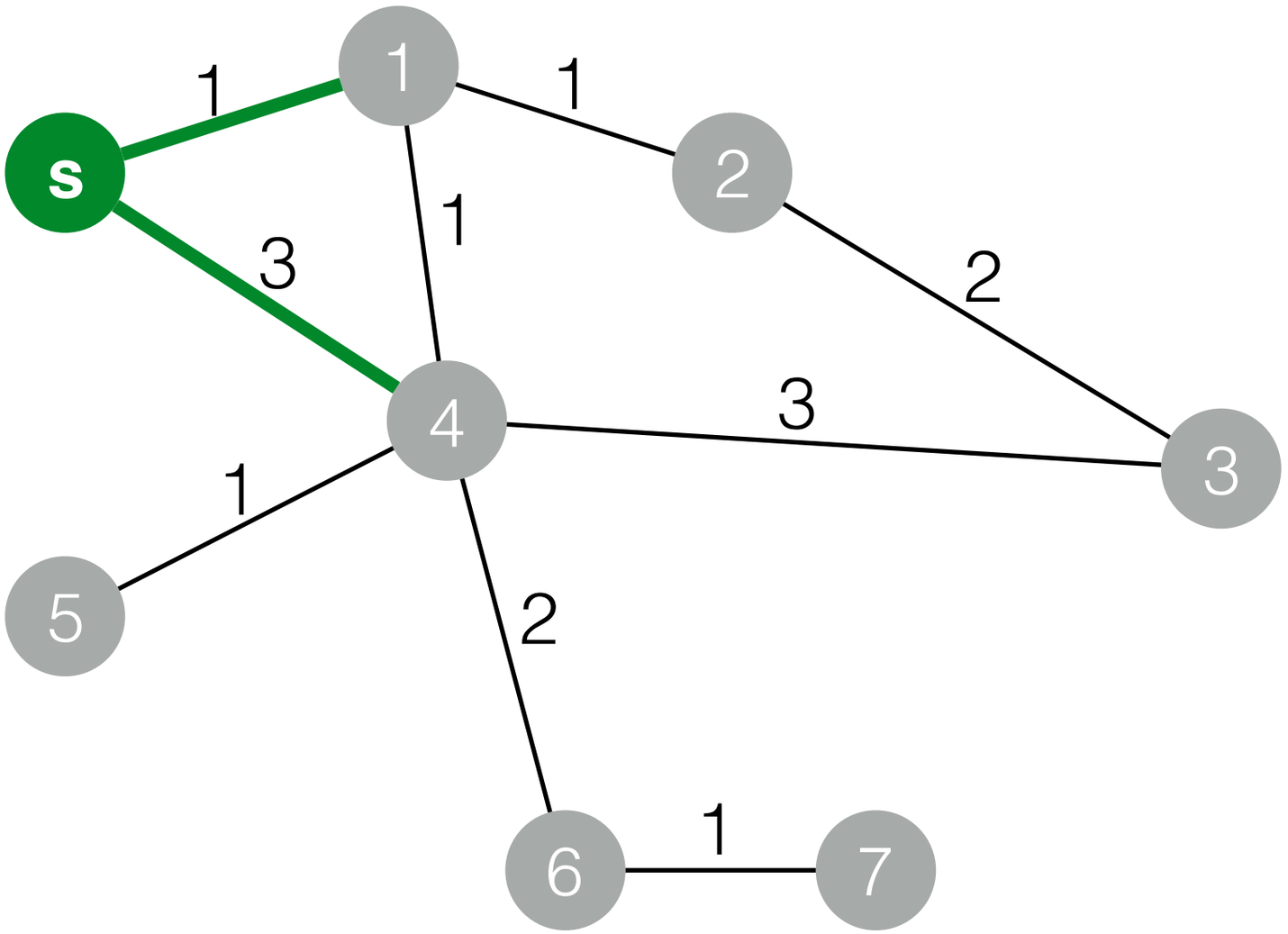


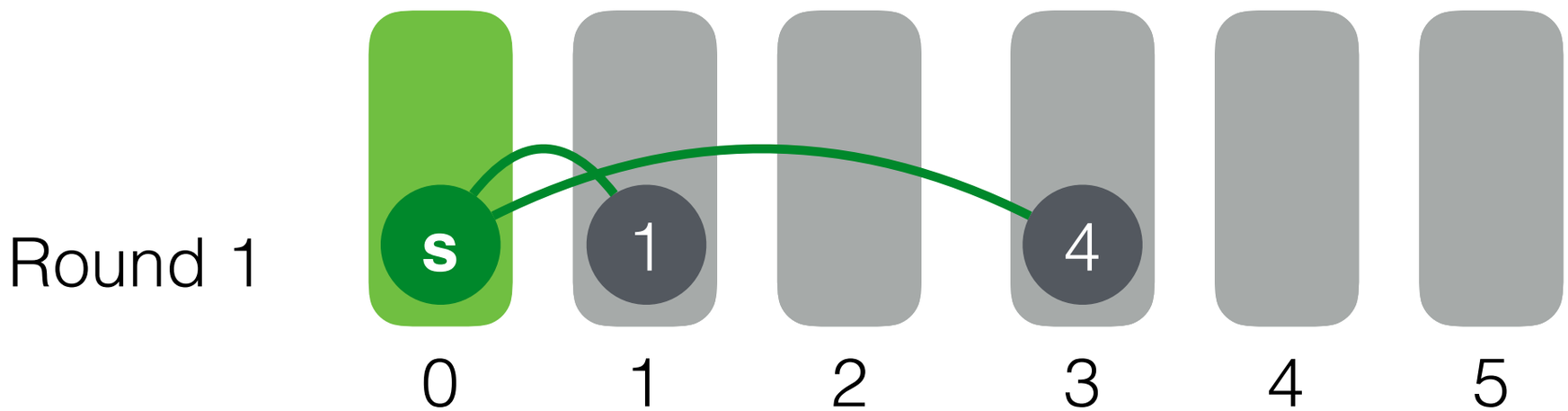
Sequential Weighted Breadth-First Search



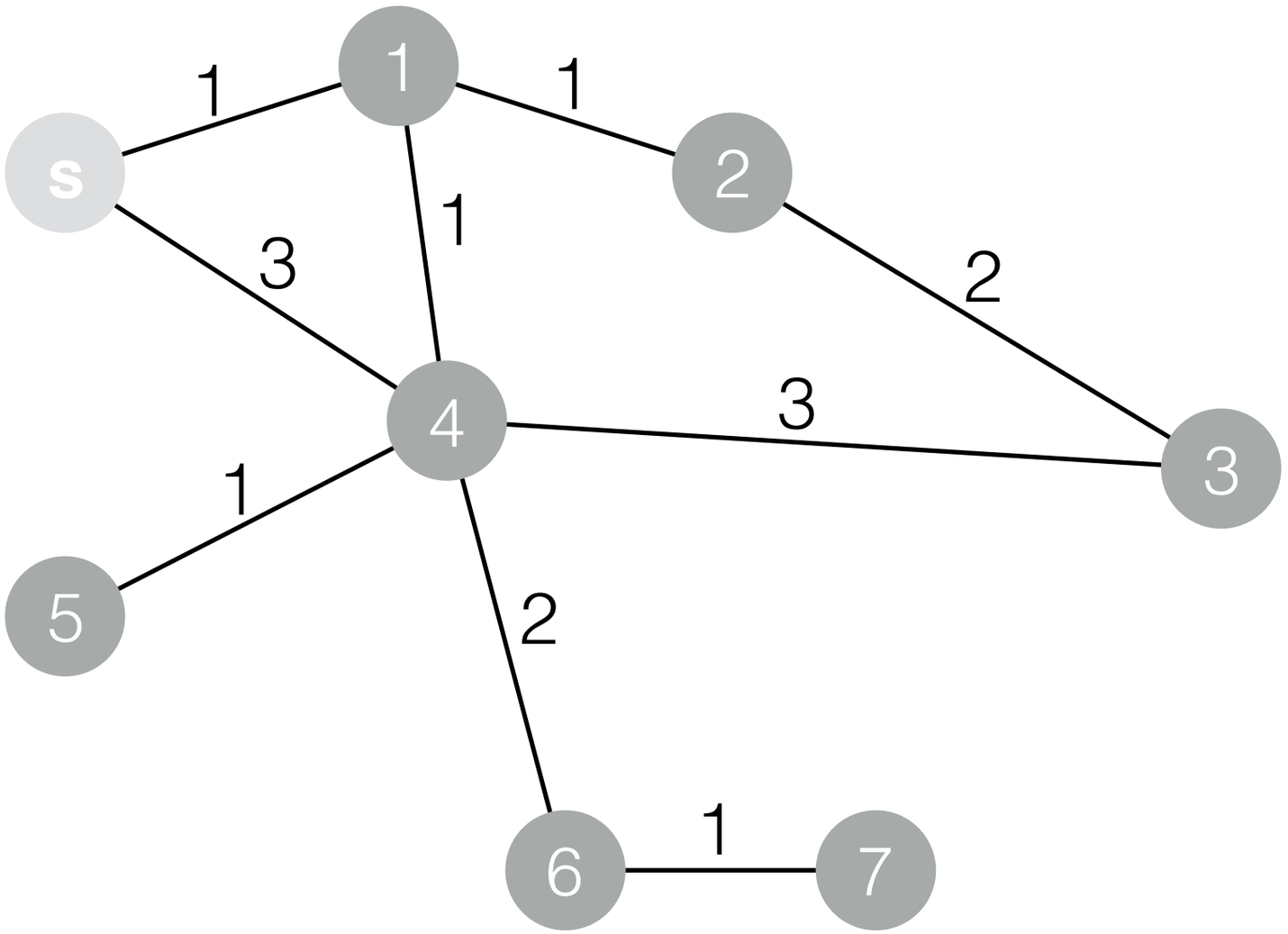


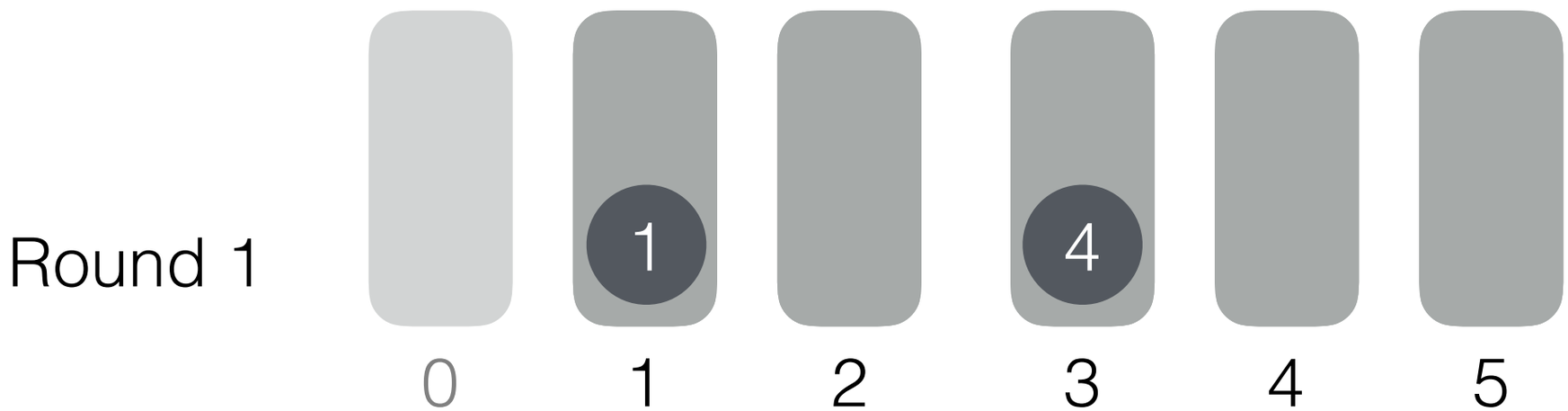
Sequential Weighted Breadth-First Search



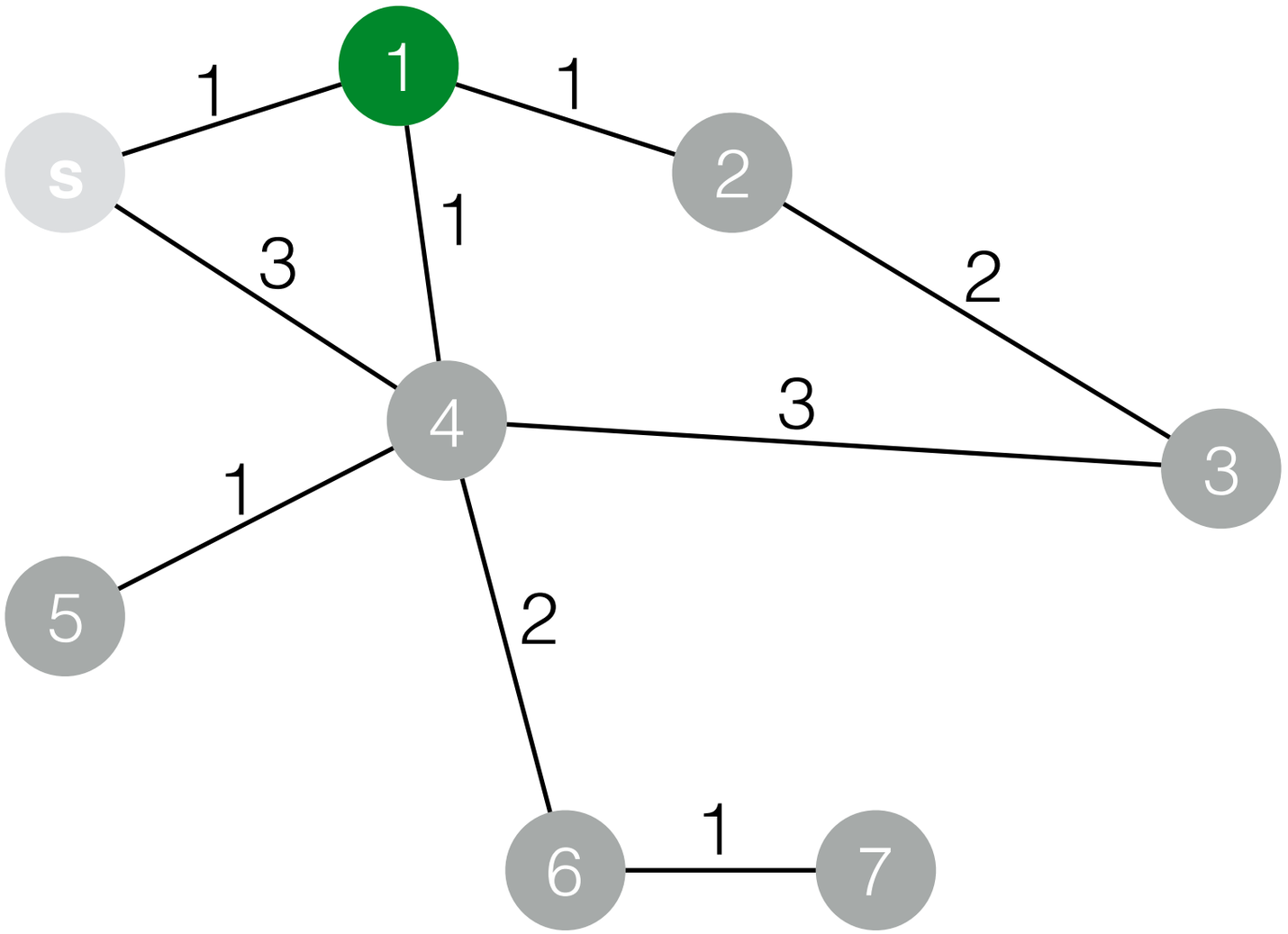


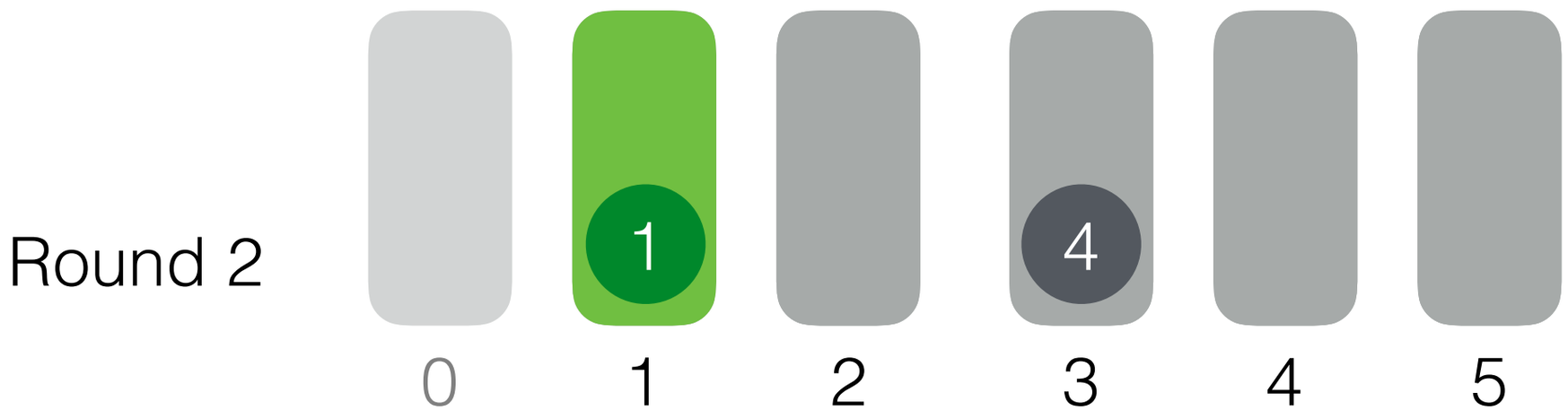
Sequential Weighted Breadth-First Search



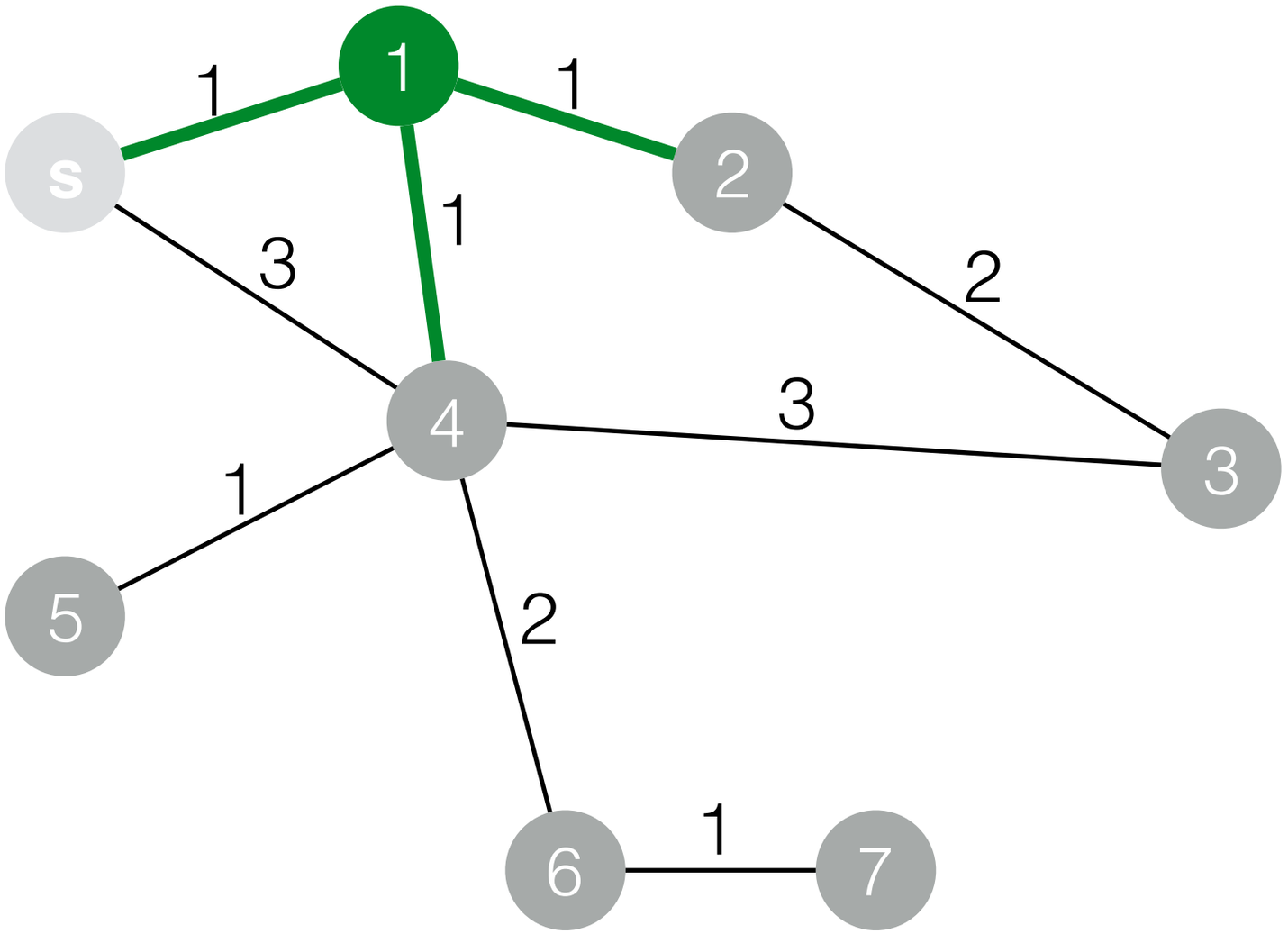


Sequential Weighted Breadth-First Search

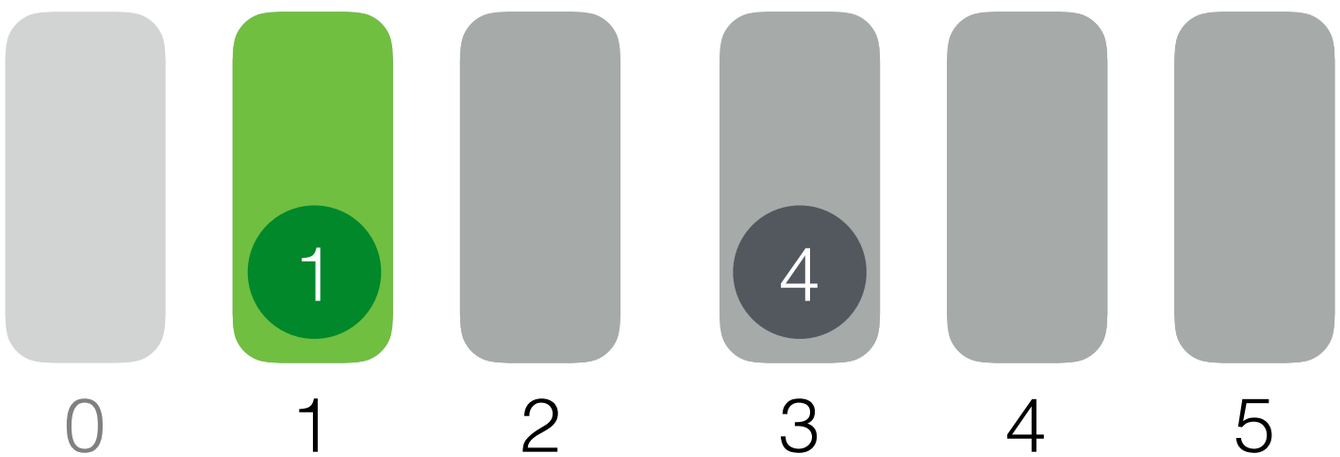




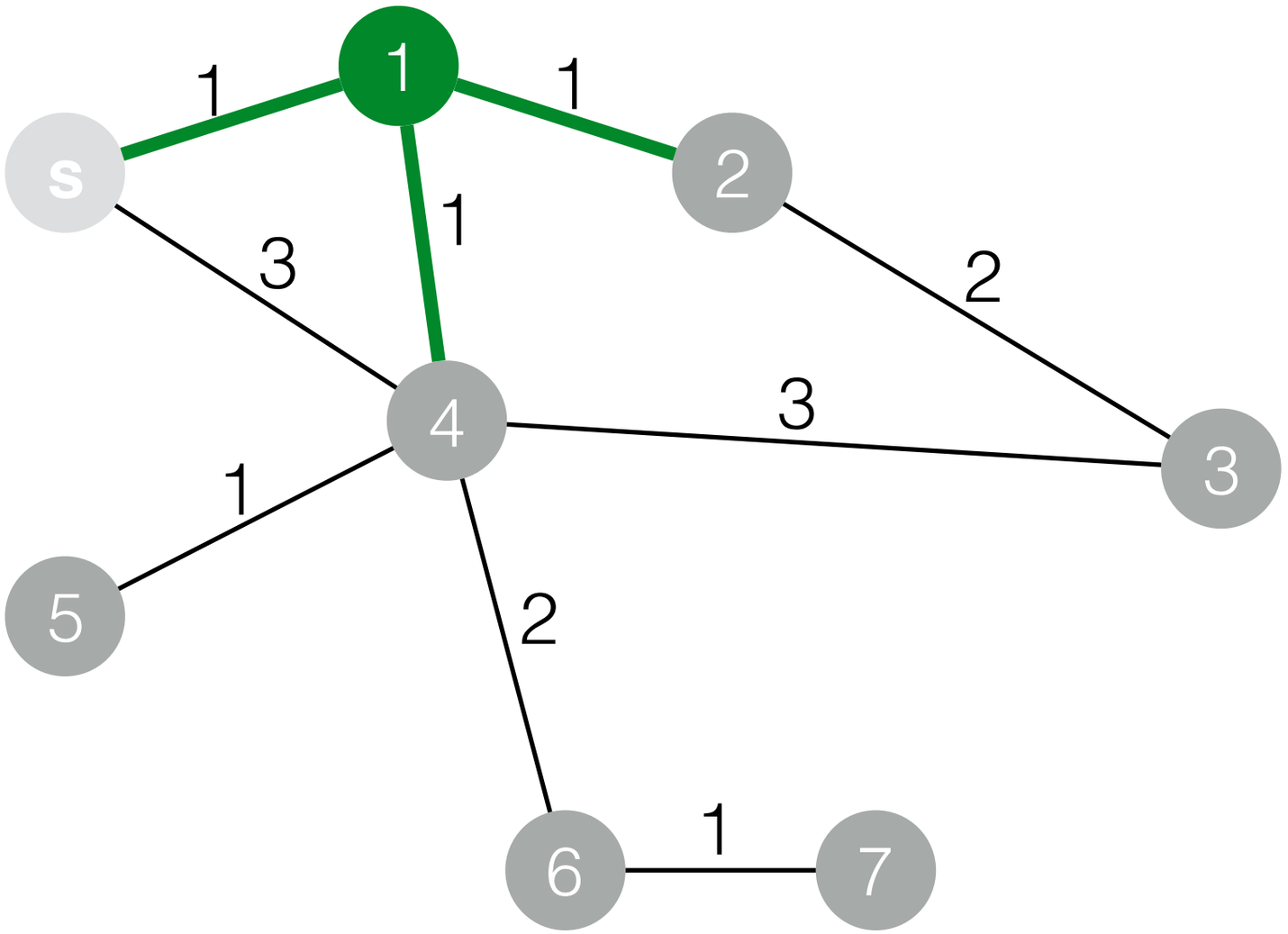
Sequential Weighted Breadth-First Search



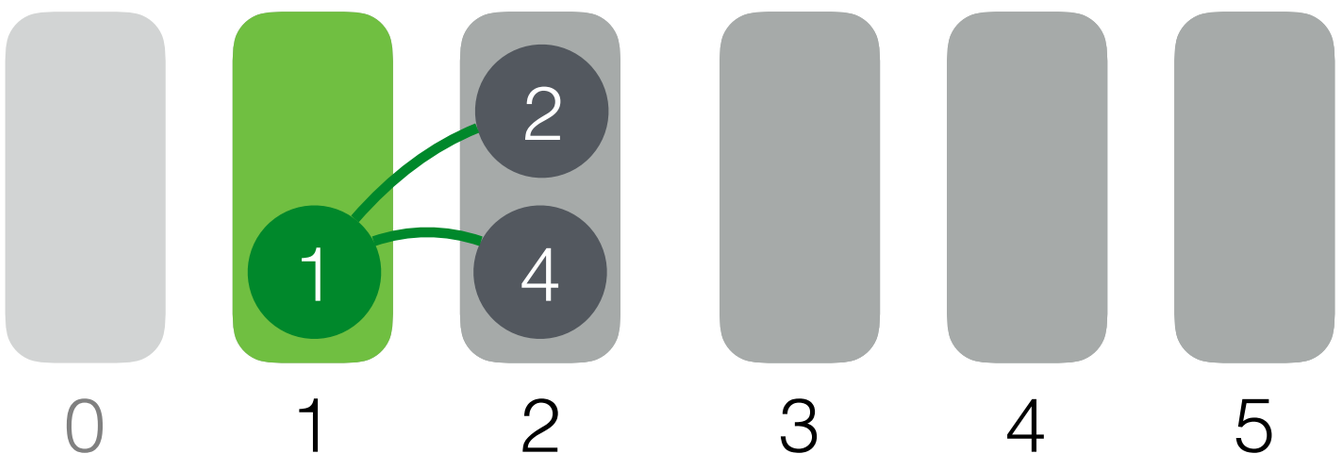
Round 2



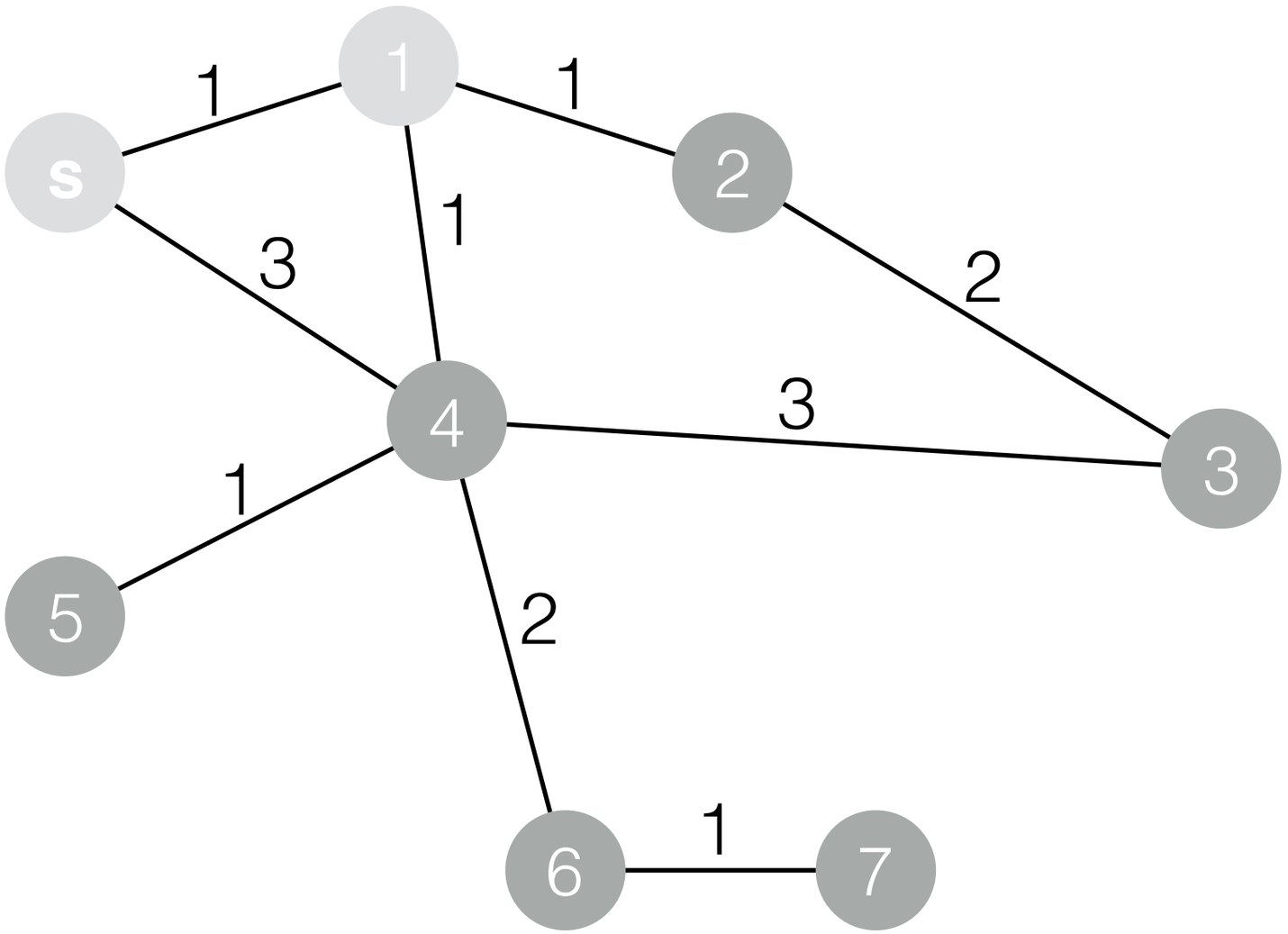
Sequential Weighted Breadth-First Search



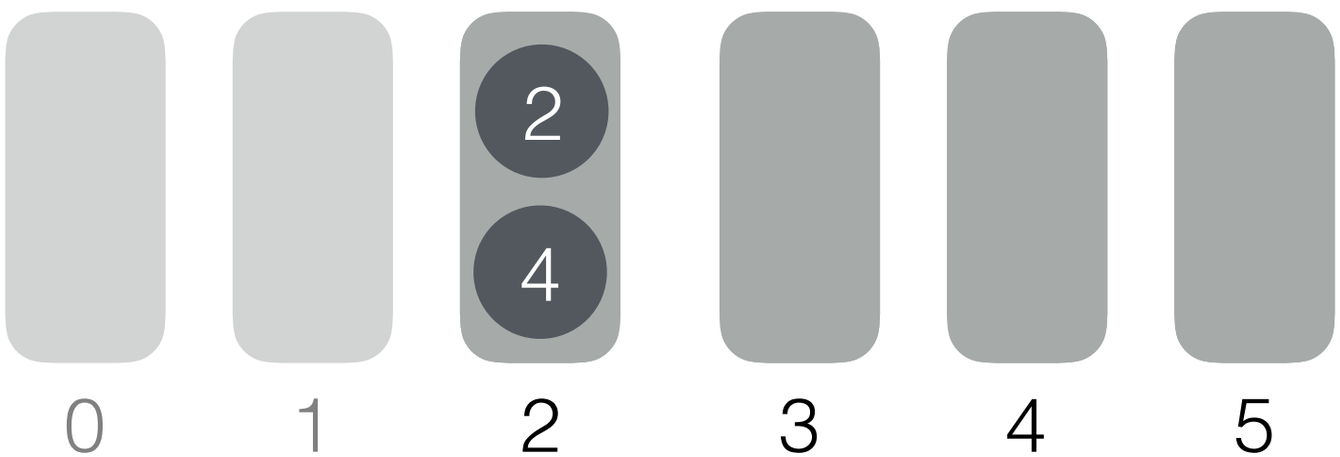
Round 2



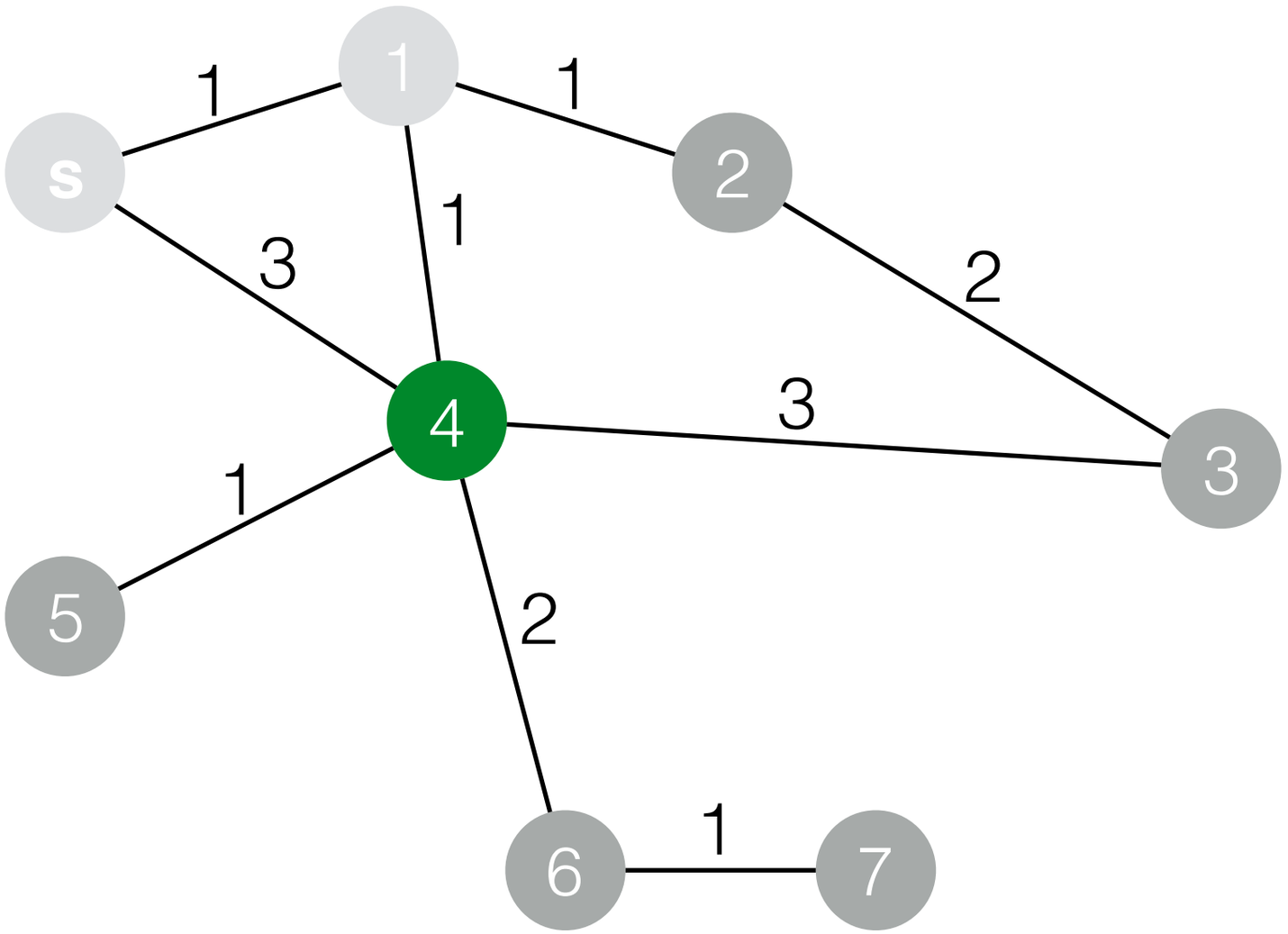
Sequential Weighted Breadth-First Search



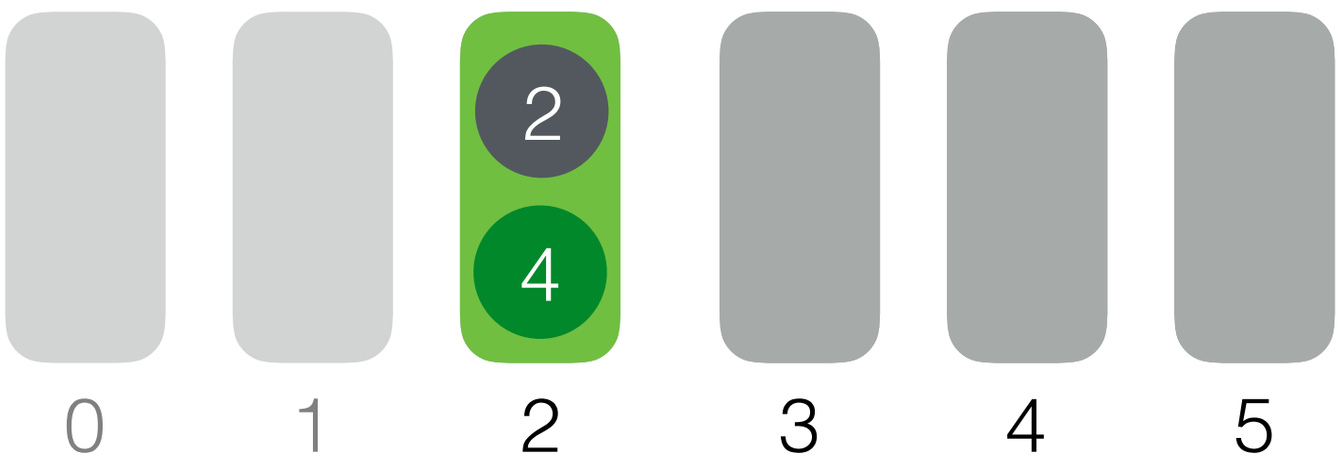
Round 2



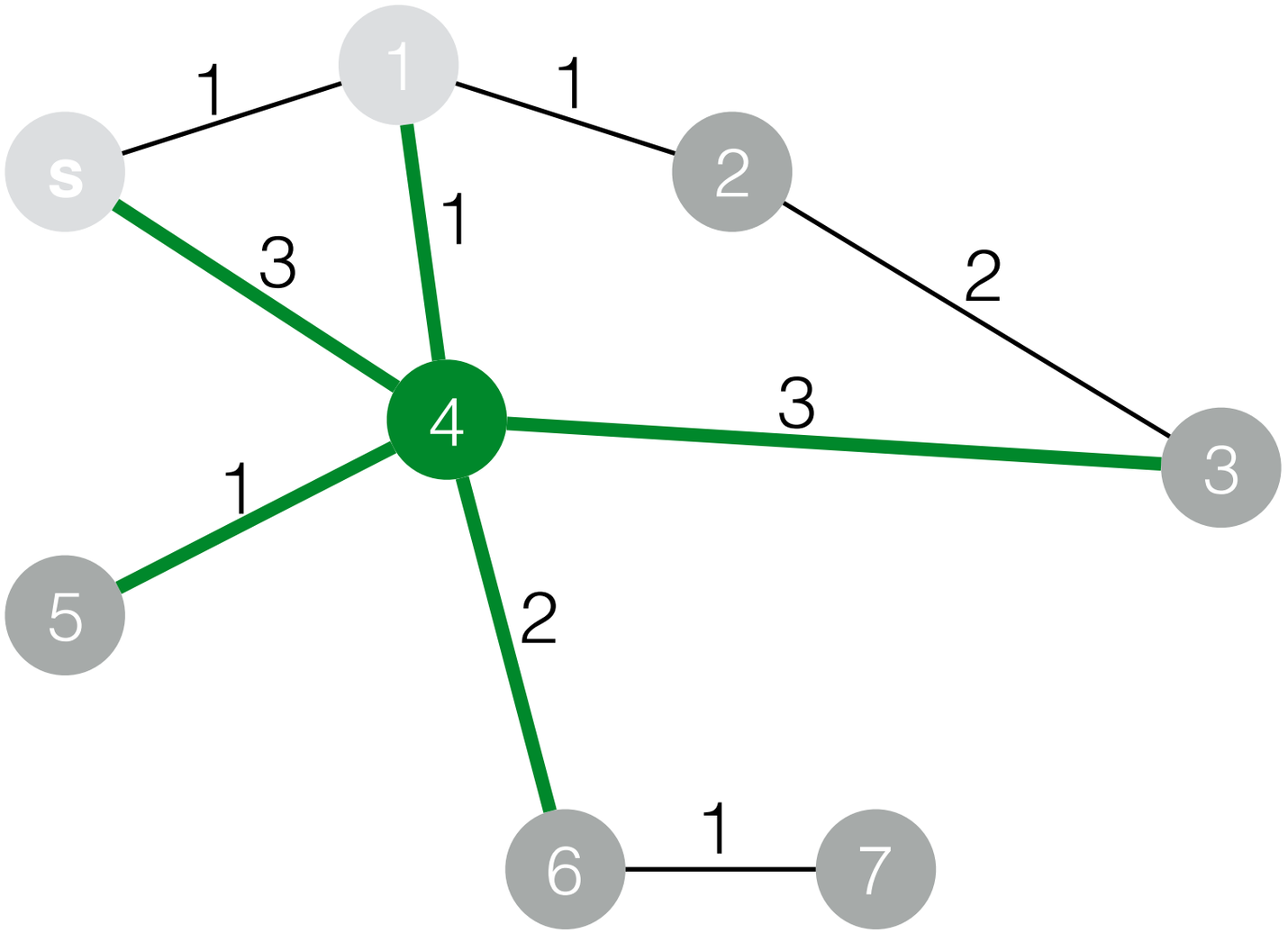
Sequential Weighted Breadth-First Search



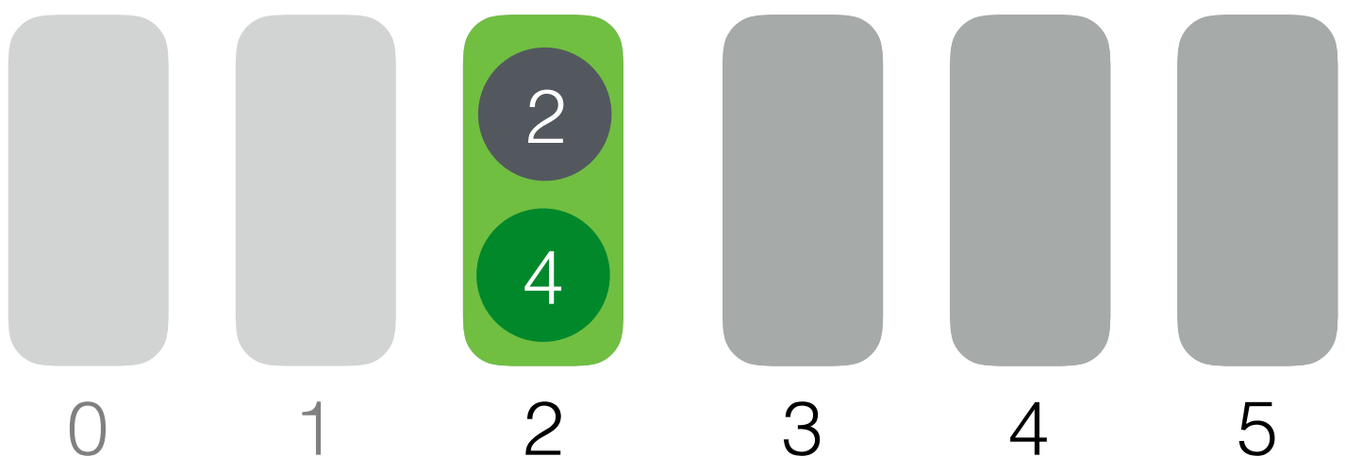
Round 3



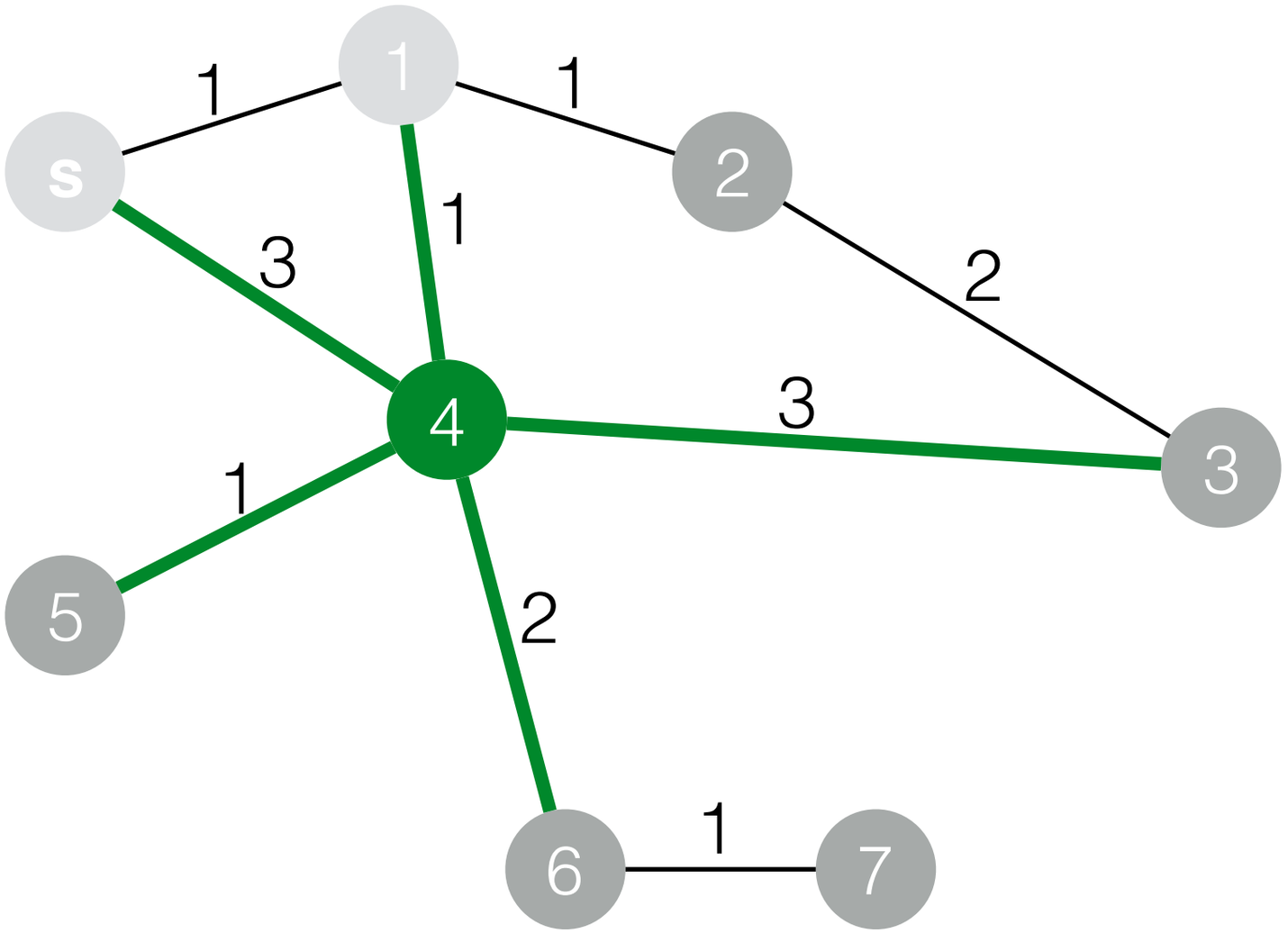
Sequential Weighted Breadth-First Search



Round 3



Sequential Weighted Breadth-First Search



$O(D + |E|)$ work where D is the graph diameter

Roun

0 1 2 3 4 5

Bucketing

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

Bucketing

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

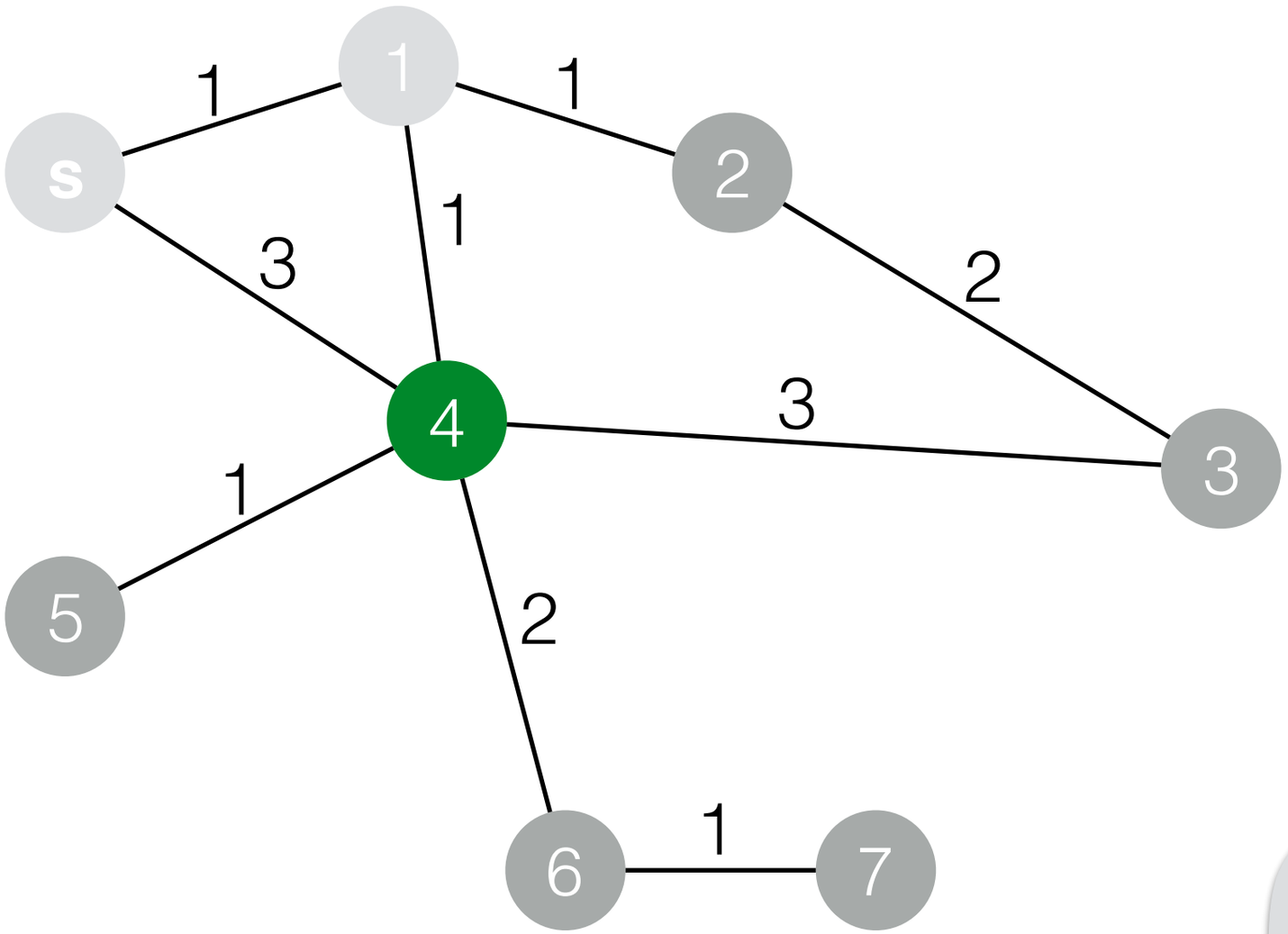
Bucketing

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

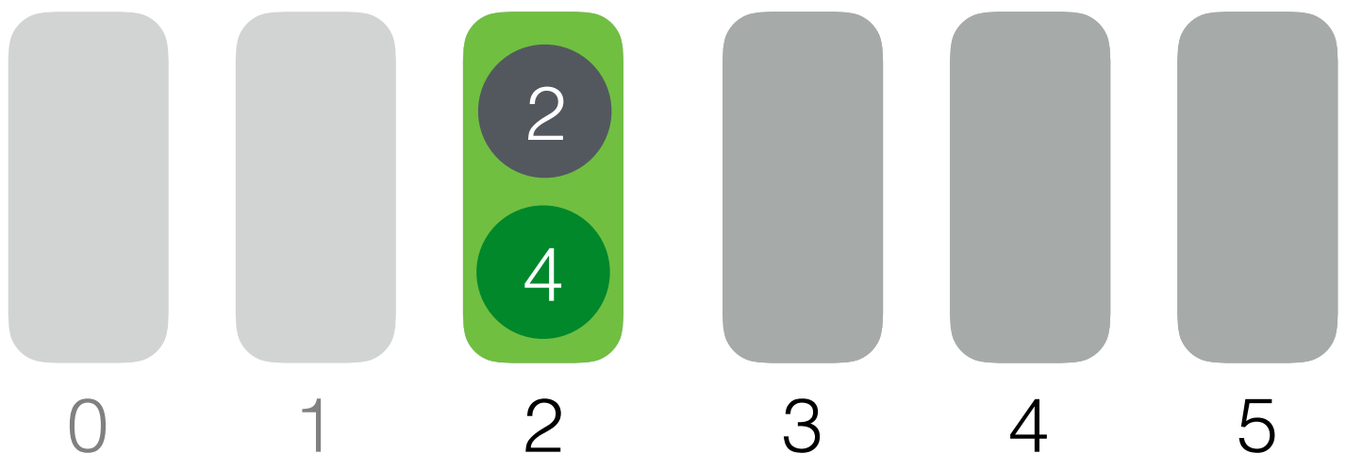
This algorithms is actually parallelizable

- 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

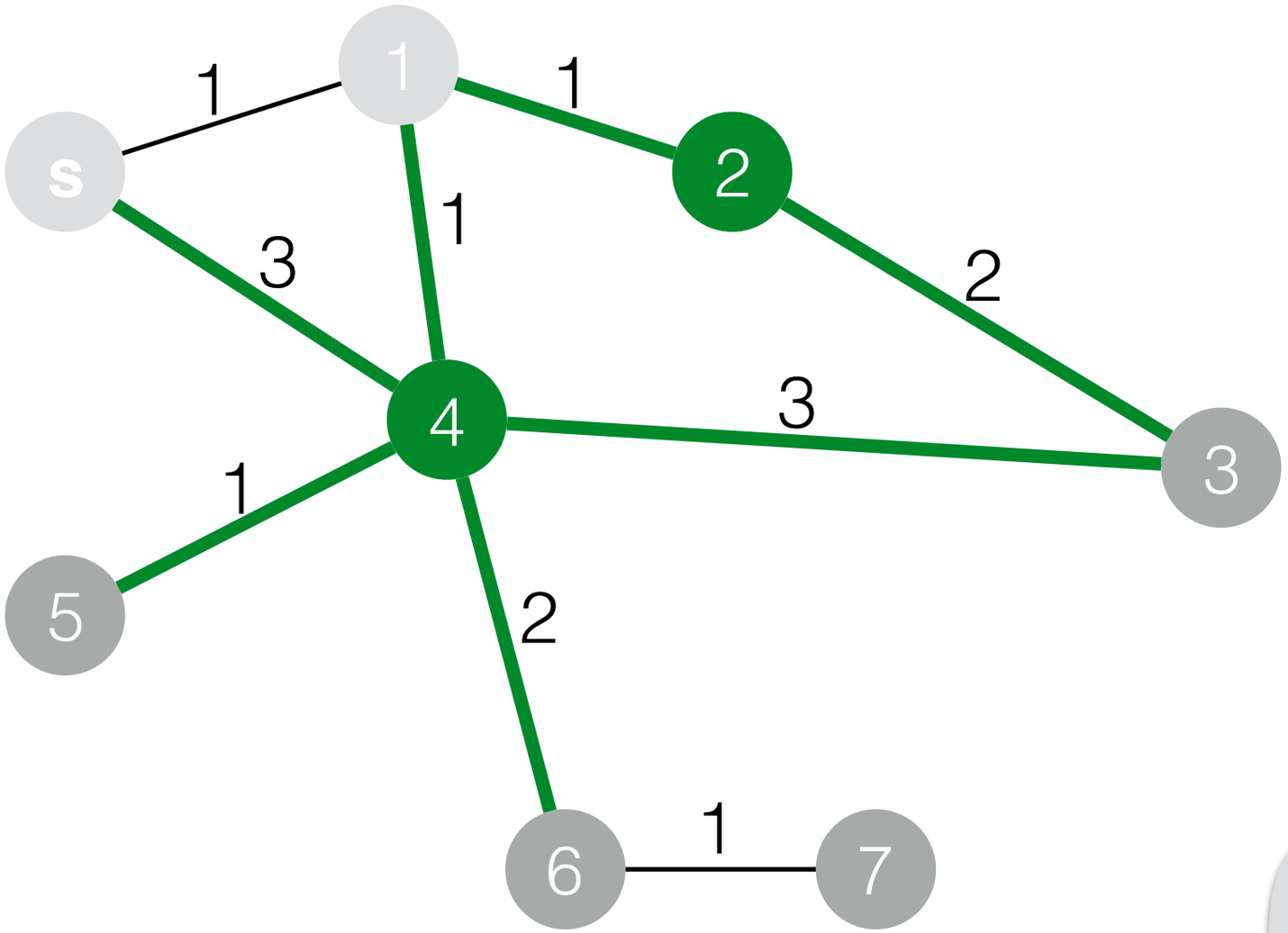


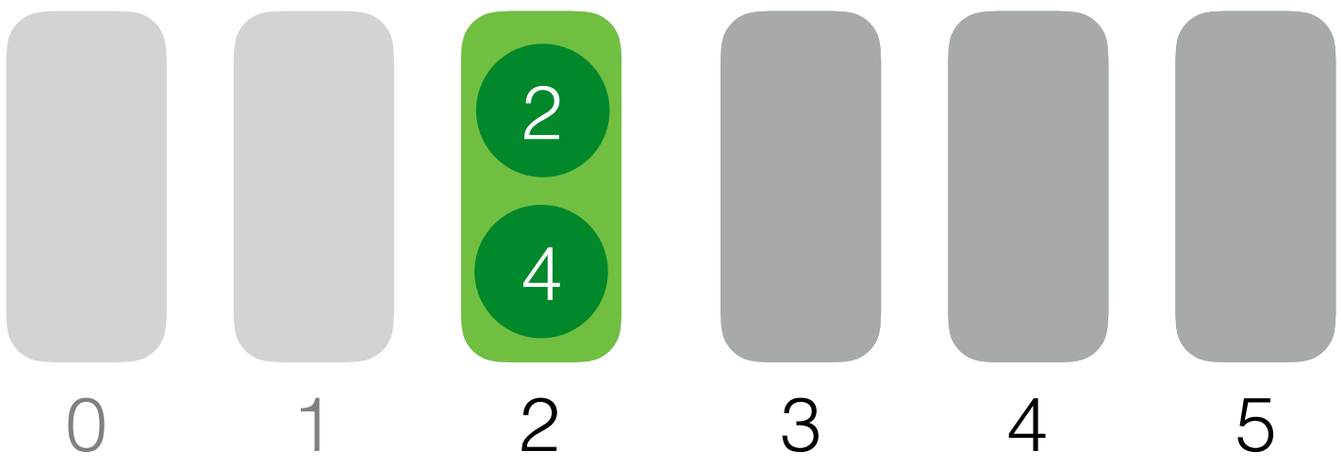
Round 3



Sequential: process vertices one by one

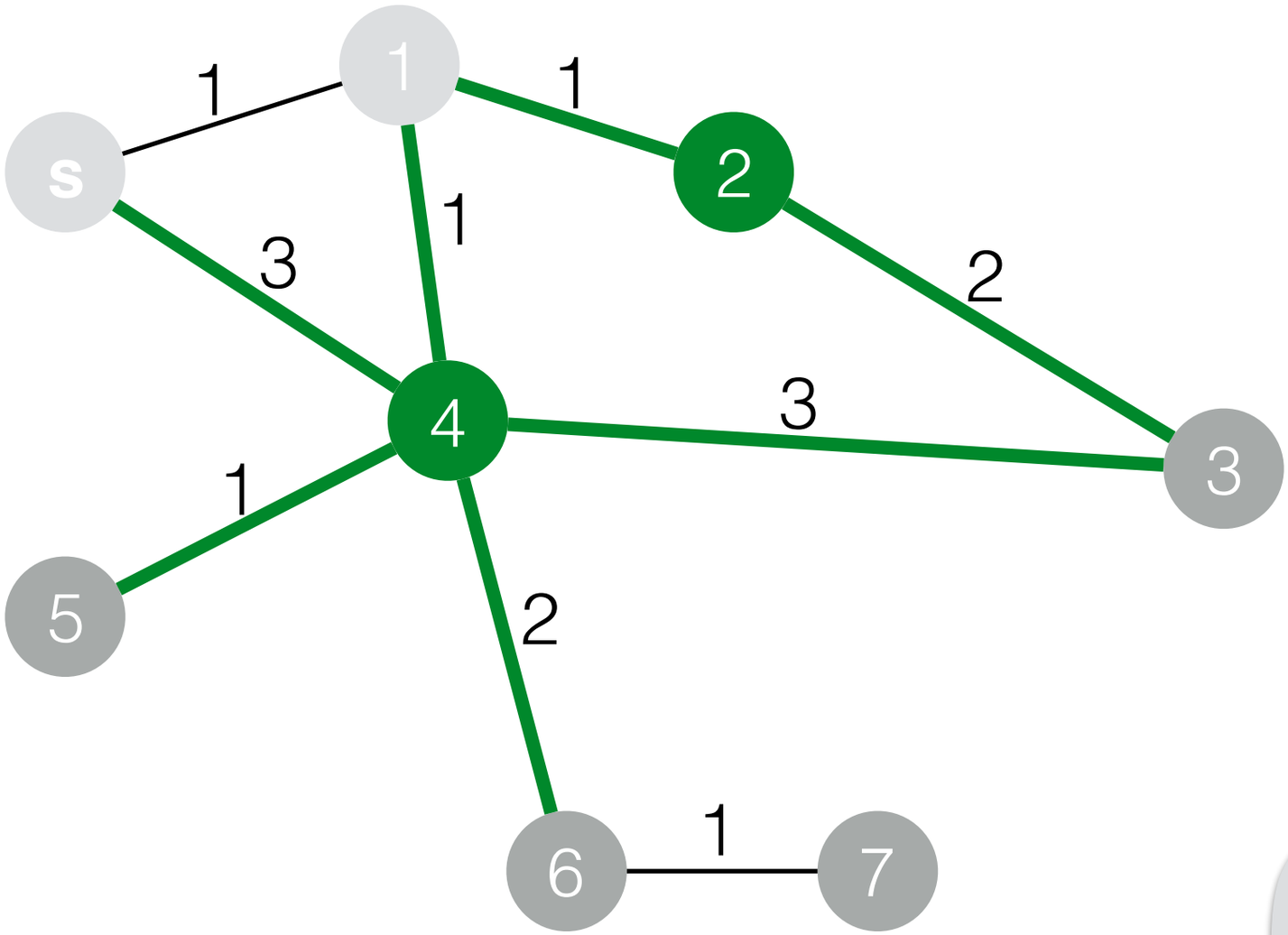
Parallel Weighted Breadth-First Search



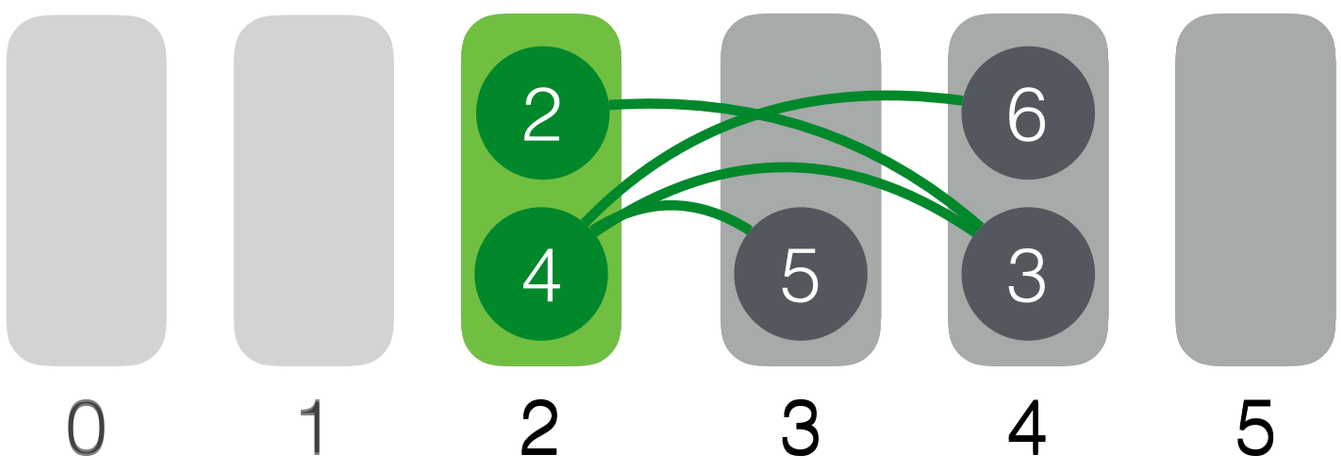


(1) Process vertices in the same bucket in parallel

Parallel Weighted Breadth-First Search



Round 3



(2) Insert neighbors into buckets in parallel

Parallel Weighted Breadth-First Search

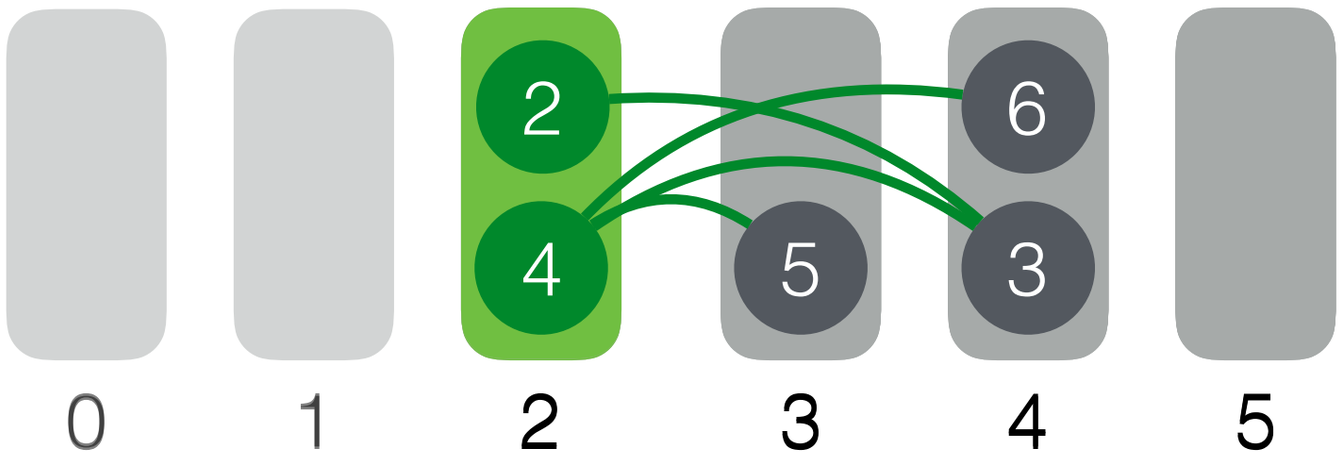
Resulting algorithm performs:

$O(D + |E|)$ **work**

$O(D \log |V|)$ **depth**

(assuming efficient bucketing)

Round 3



(2) Insert neighbors into buckets in parallel

Parallel bucketing

Bucketing is useful for more than just wBFS

- k-core (coreness)
- Delta-Stepping
- Parallel Approximate Set Cover

Parallel bucketing

Bucketing is useful for more than just wBFS

- k-core (coreness)
- Delta-Stepping
- Parallel Approximate Set Cover

Goals

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

Parallel bucketing

Bucketing is useful for more than just wBFS

- k-core (coreness)
- Delta-Stepping
- Parallel Approximate Set Cover

Goals

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

Difficulties

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

Results: Julienne

Shared memory framework for *bucketing-based algorithms*

Results: Julienne

Shared memory framework for *bucketing-based algorithms*

Extend Ligra with an interface for bucketing

- Theoretical bounds for primitives
- Fast implementations of primitives

Results: Julienne

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.

Results: Julienne

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 U

Bucketing implementation is work-efficient

Results: Julienne

Work-efficient implementations of 4 bucketing-based algorithms:

- k-core (coreness)
- Weighted Breadth-First Search
- Delta-Stepping
- Parallel Approximate Set Cover

Results: Julienne

Work-efficient implementations of 4 bucketing-based algorithms:

- k-core (coreness)
- Weighted Breadth-First Search
- Delta-Stepping
- Parallel Approximate Set Cover

Codes are simple

- All implementations < 100 LoC

Results: Julienne

Work-efficient implementations of 4 bucketing-based algorithms:

- k-core (coreness)
- Weighted Breadth-First Search
- Delta-Stepping
- Parallel Approximate Set Cover

Codes are simple

- All implementations < 100 LoC

Codes competitive with, or outperform existing implementations

Results: Julienne

Work-efficient implementations of 4 bucketing-based algorithms:

- k-core (coreness)
- Weighted Breadth-First Search
- Delta-Stepping
- Parallel Approximate Set Cover

Codes are simple

- All implementations < 100 LoC

Codes competitive with, or outperform existing implementations

First work-efficient k-core algorithm with non-trivial parallelism

Results: Julienne

Work-efficient implementations of 4 bucketing-based algorithms:

- k-core (coreness)
- Weighted Breadth-First Search
- Delta-Stepping
- Parallel Approximate Set Cover

Codes are simple

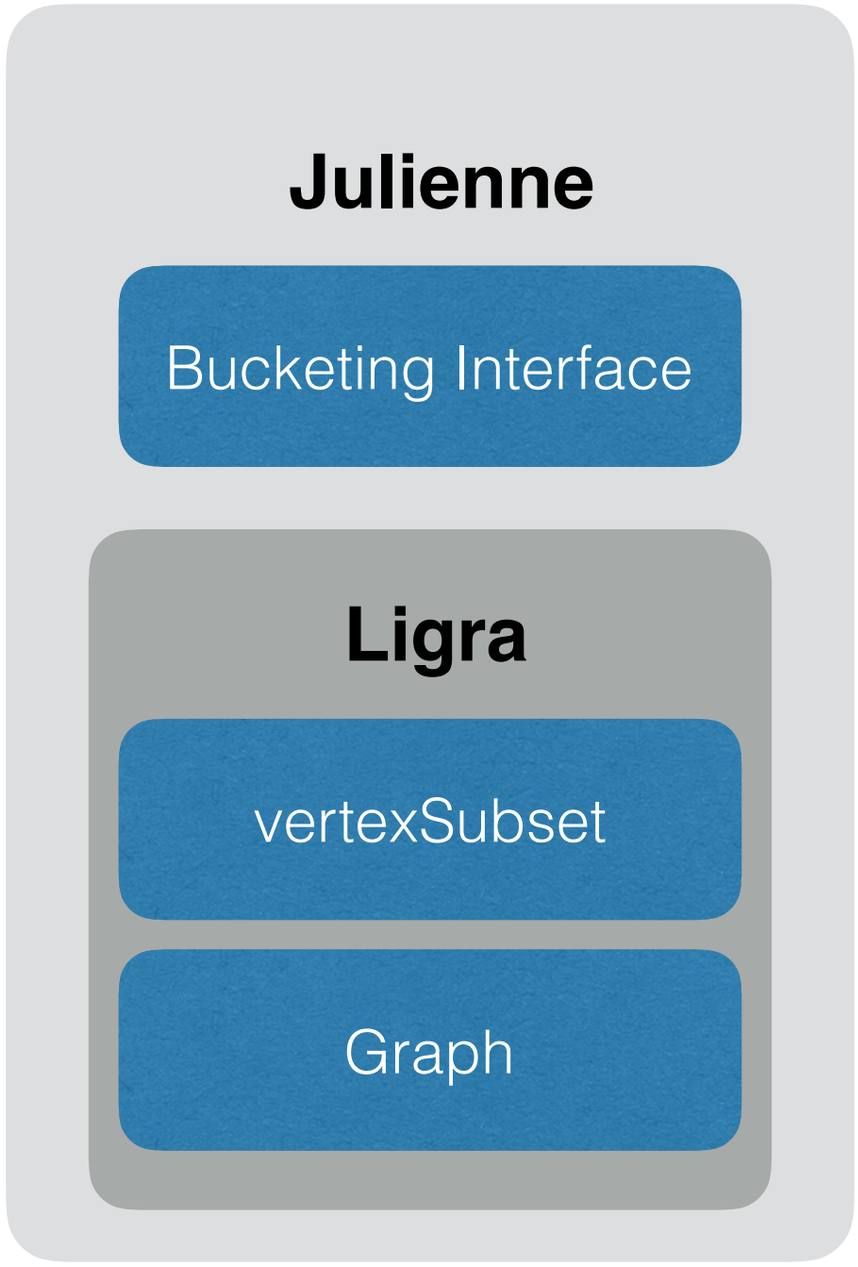
- All implementations < 100 LoC

Codes competitive with, or outperform existing implementations

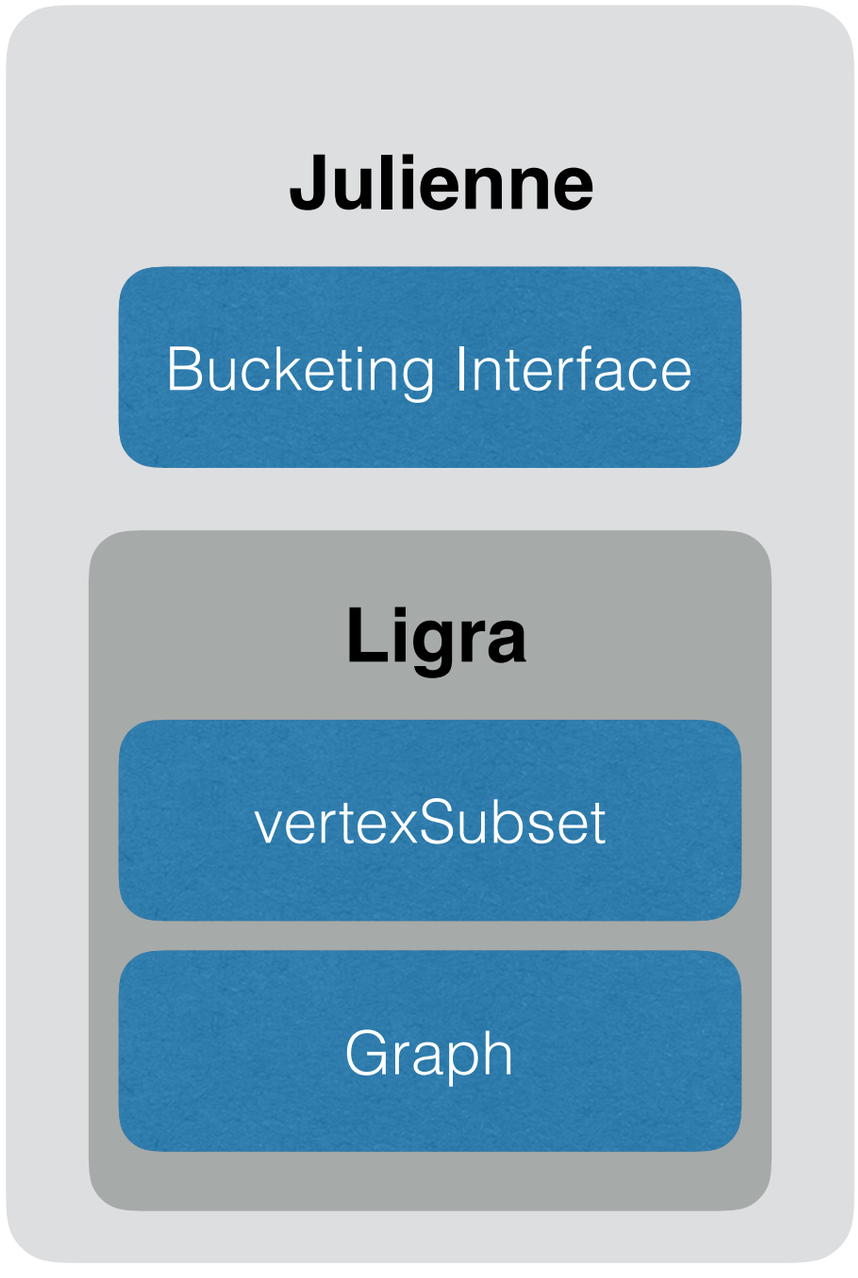
First work-efficient k-core algorithm with non-trivial parallelism

Compute k-cores of largest publicly available graph (~200B edges) in ~3 minutes and approximate set-cover in ~2 minutes

Julienne: Interface



Julienne: Interface



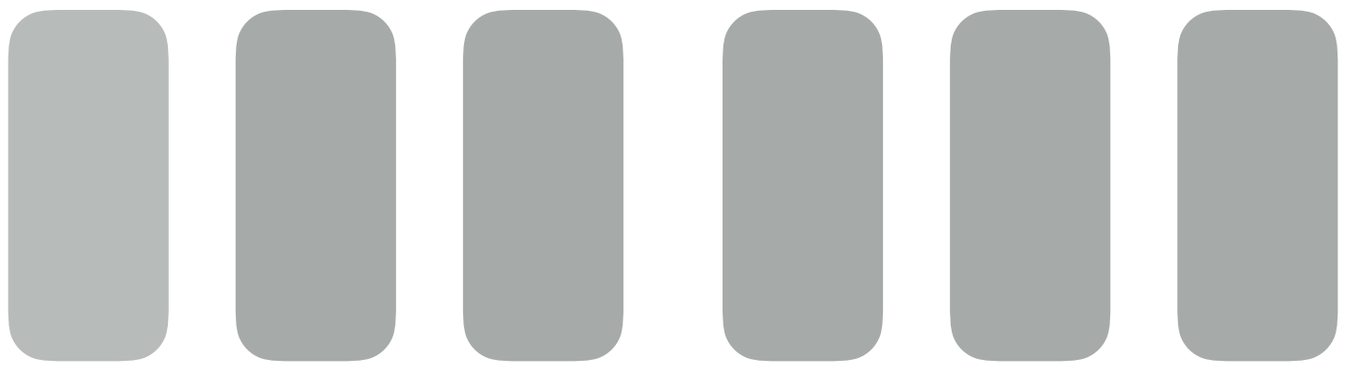
Bucketing Interface:

(1) Create bucket structure

(2) Get the next bucket (vertexSubset)

(3) Update buckets of a subset of identifiers

Julienne: Interface



MakeBuckets : buckets

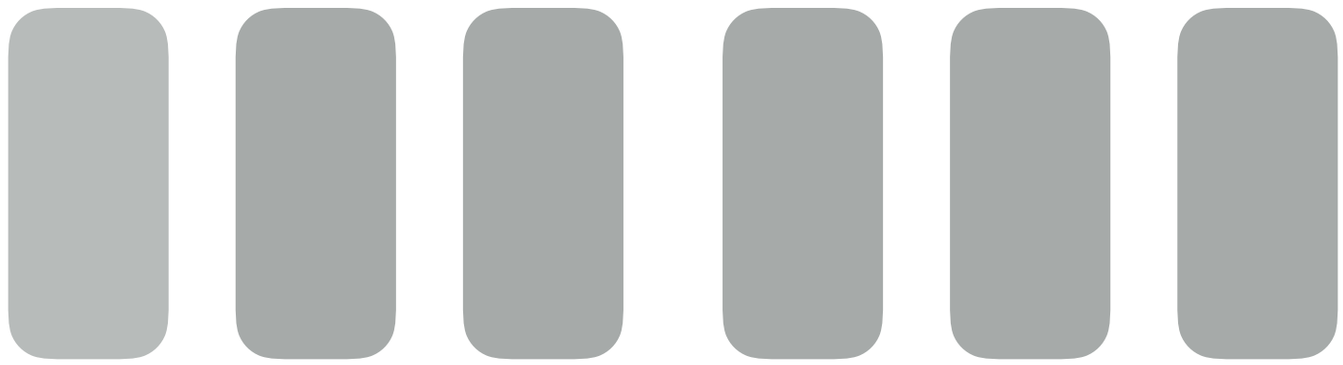
n : int

D : identifier $\rightarrow$ bucket_id

O : bucket_order

Initialize bucket structure

Julienne: Interface



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

MakeBuckets : buckets

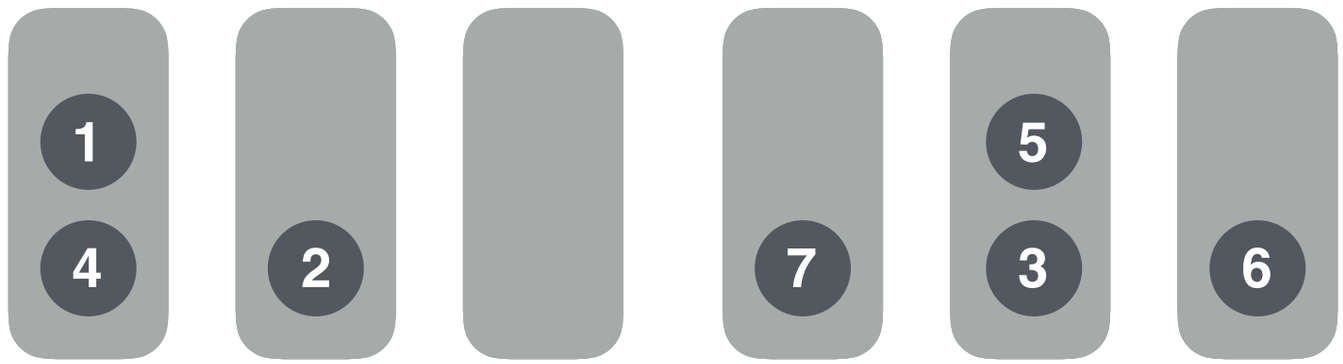
n : int

D : identifier $\rightarrow$ bucket_id

O : bucket_order

Initialize bucket structure

Julienne: Interface

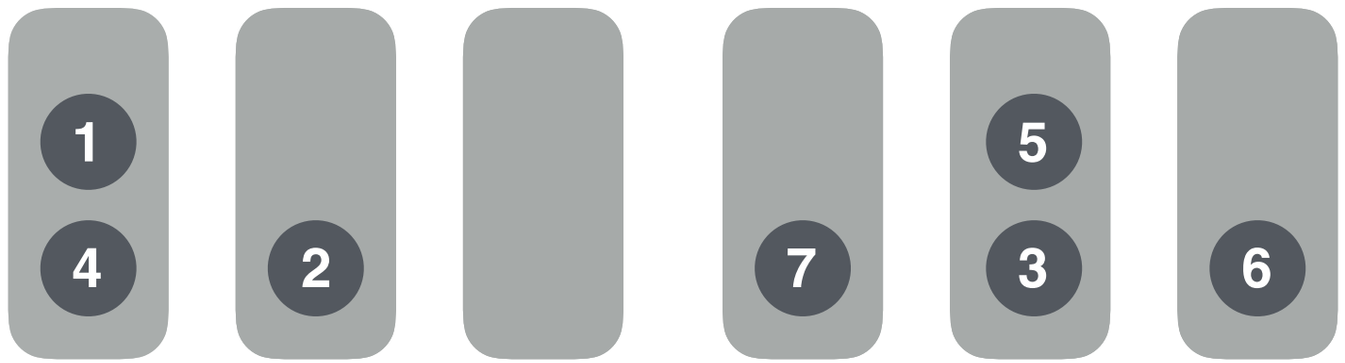


$$D(1) = 0, D(2) = 1, D(3) = 4, \ldots$$

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

Initialize bucket structure

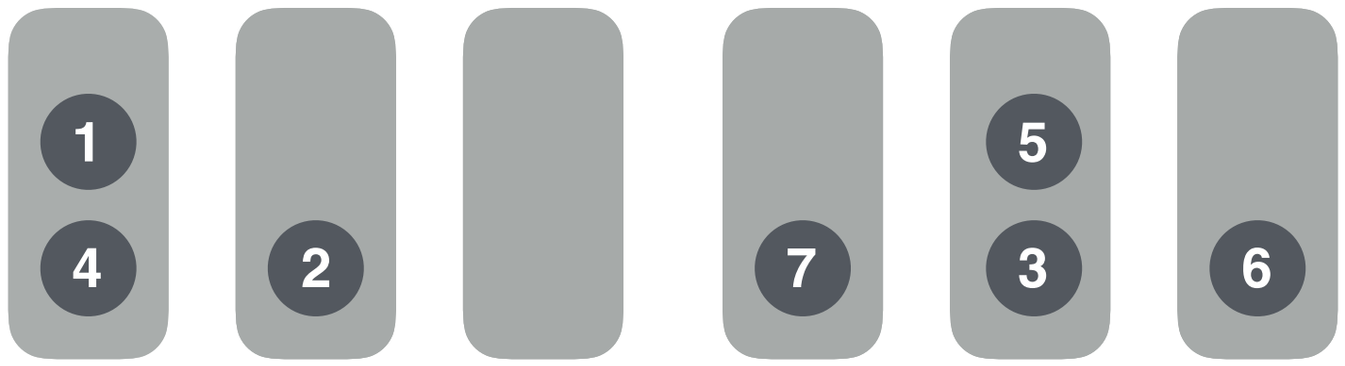
Julienne: Interface



NextBucket : bucket

Extract identifiers in the next non-empty bucket

Julienne: Interface

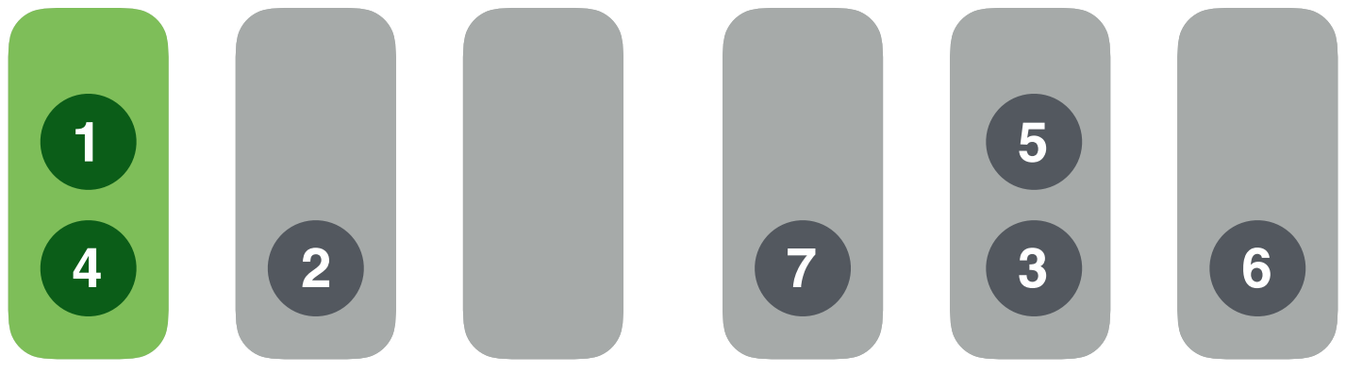


Order: increasing

NextBucket : bucket

Extract identifiers in the next non-empty bucket

Julienne: Interface

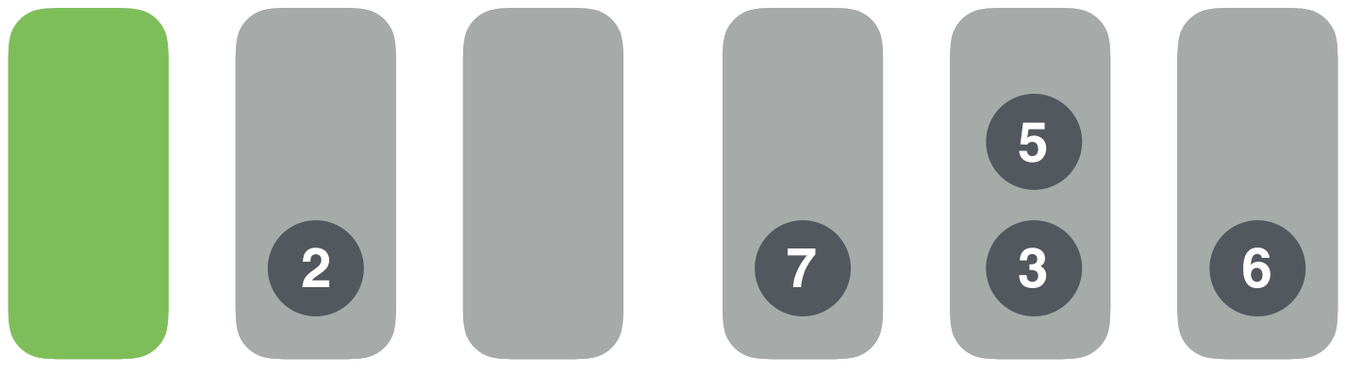


Order: increasing

NextBucket : bucket

Extract identifiers in the next non-empty bucket

Julienne: Interface

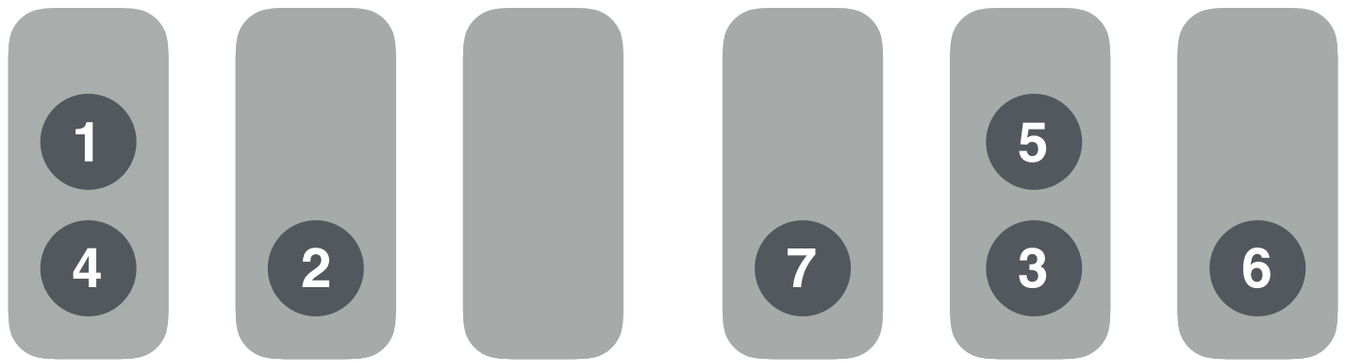


Order: increasing

NextBucket : bucket

Extract identifiers in the next non-empty bucket

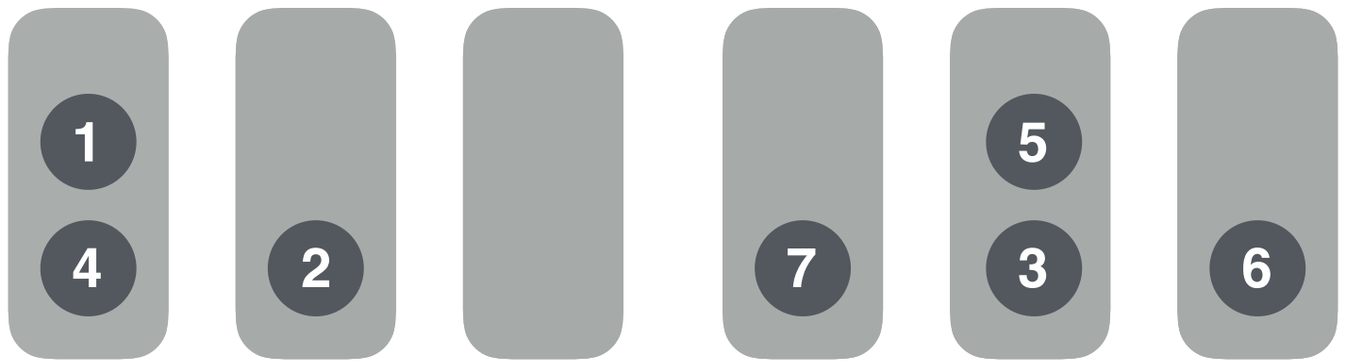
Julienne: Interface



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

Update buckets for k identifiers

Julienne: Interface

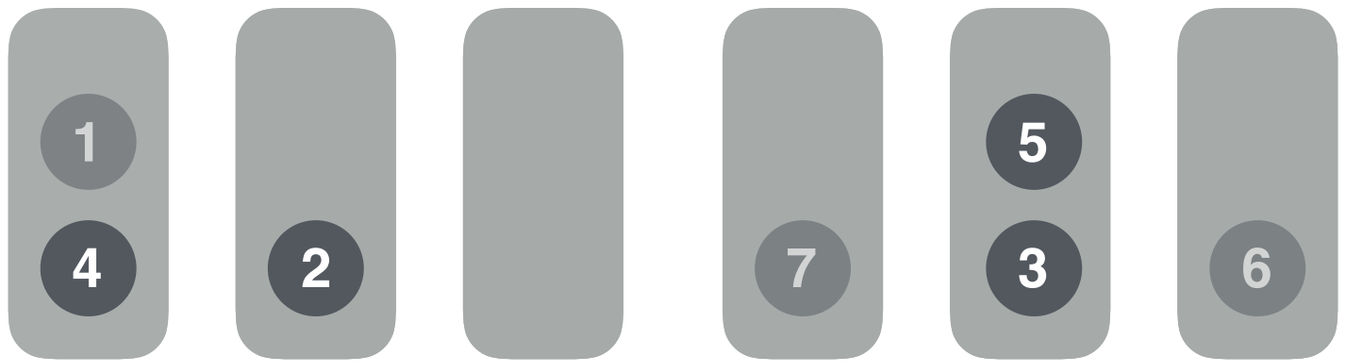


[(1,3), (7,2), (6,2)]

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

Update buckets for k identifiers

Julienne: Interface

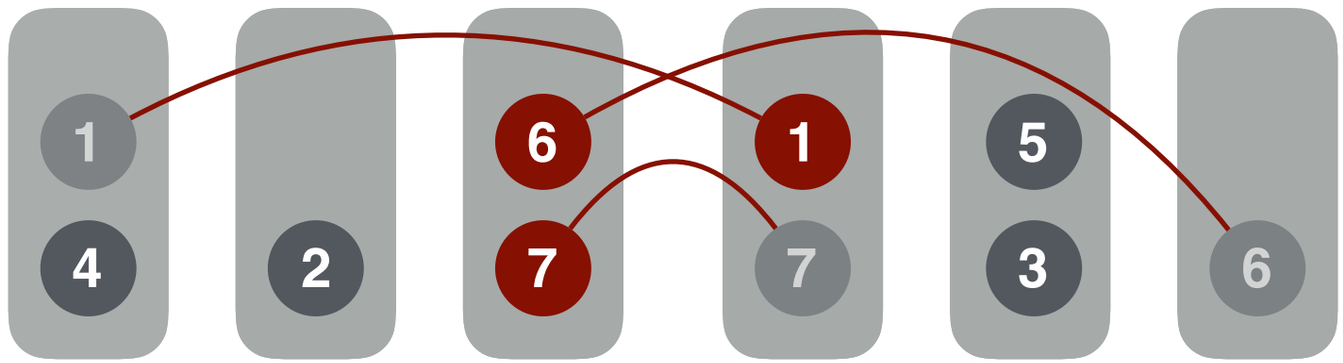


[(1,3), (7,2), (6,2)]

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

Update buckets for k identifiers

Julienne: Interface



[(1,3), (7,2), (6,2)]

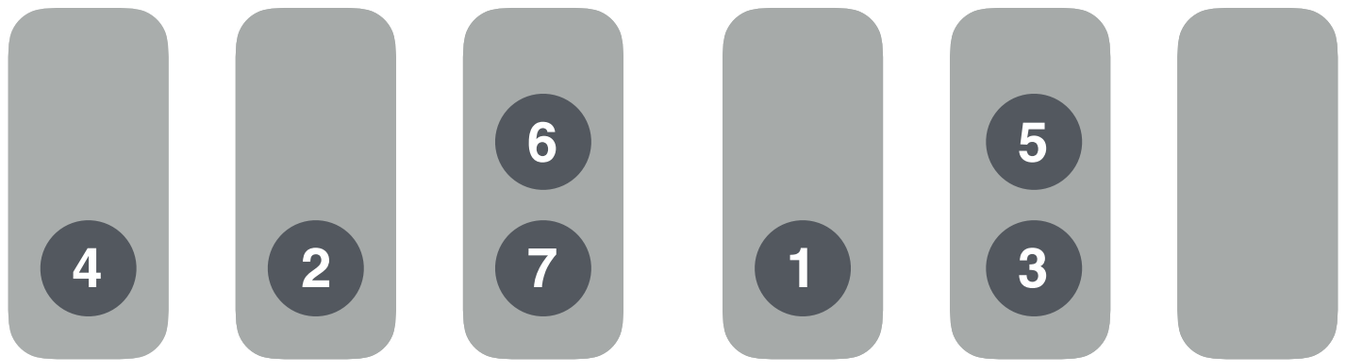
UpdateBuckets

k : int

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

Update buckets for k identifiers

Julienne: Interface



[(1,3), (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

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

Implementation:

- Use dynamic arrays
- Update 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.

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.

Implementation:

- 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.

Parallel Bucketing

UpdateBuckets:

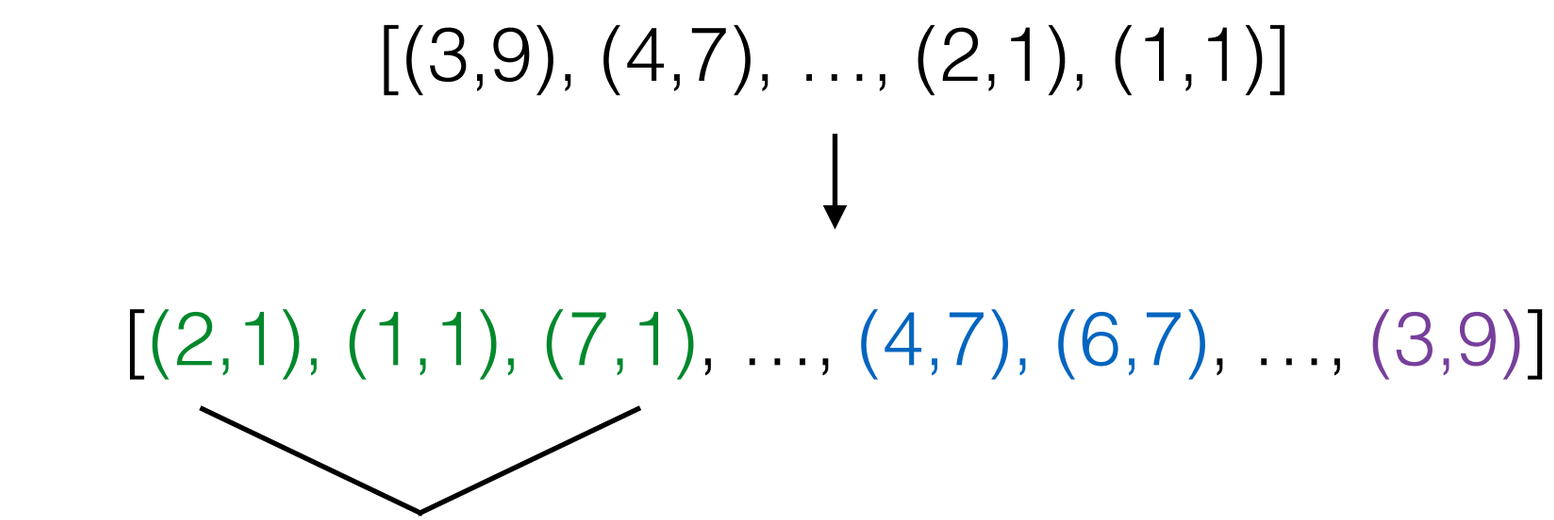
- Use work-efficient semisort [Gu et al. 2015]
- Given k (key, value) pairs, semisorts in $O(k)$ expected work and $O(\log k)$ depth w.h.p.

[(3,9), (4,7), …, (2,1), (1,1)]

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.

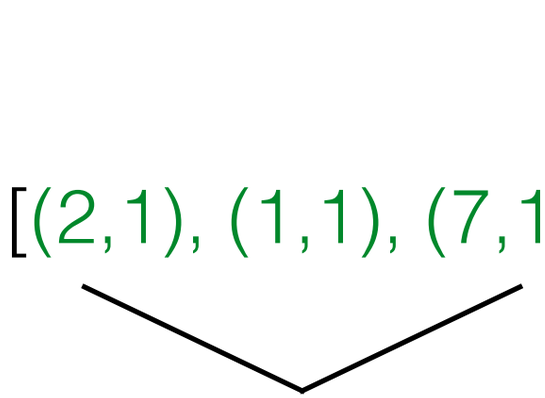


Parallel Bucketing

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)]

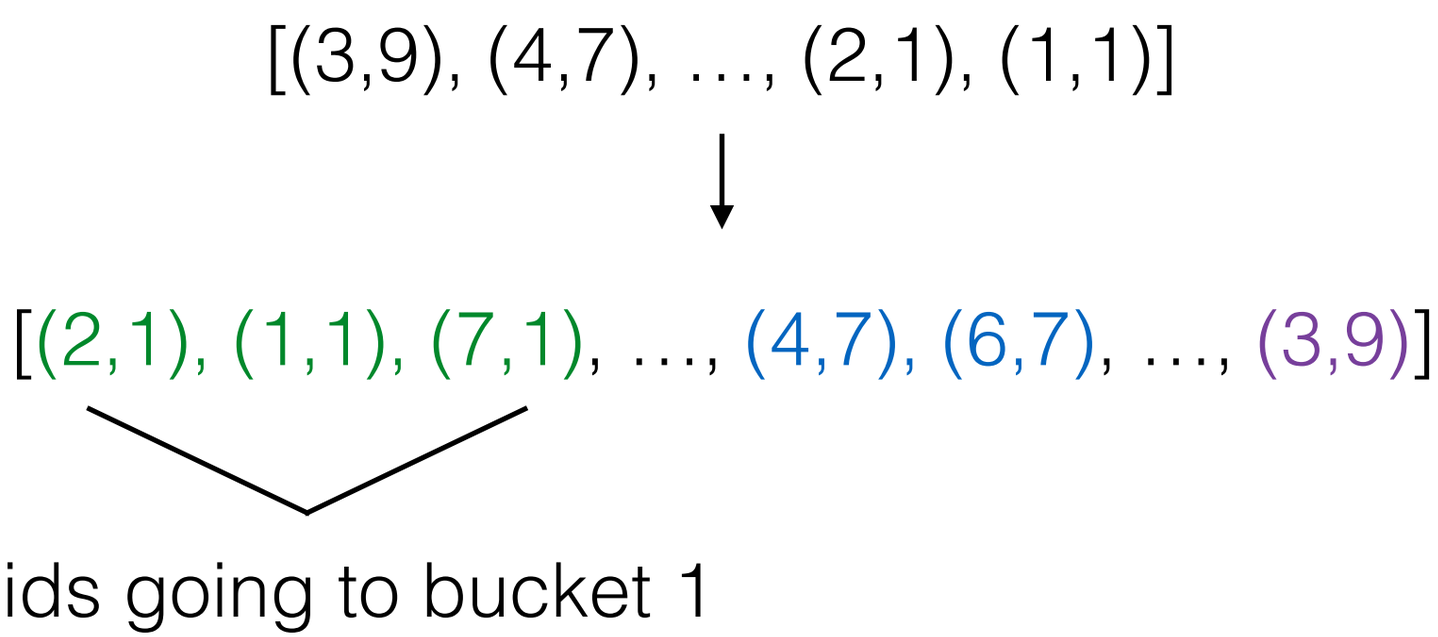
All ids going to bucket 1

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

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.



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

Please see paper for details on practical implementation and optimizations

Example: k-core and coreness

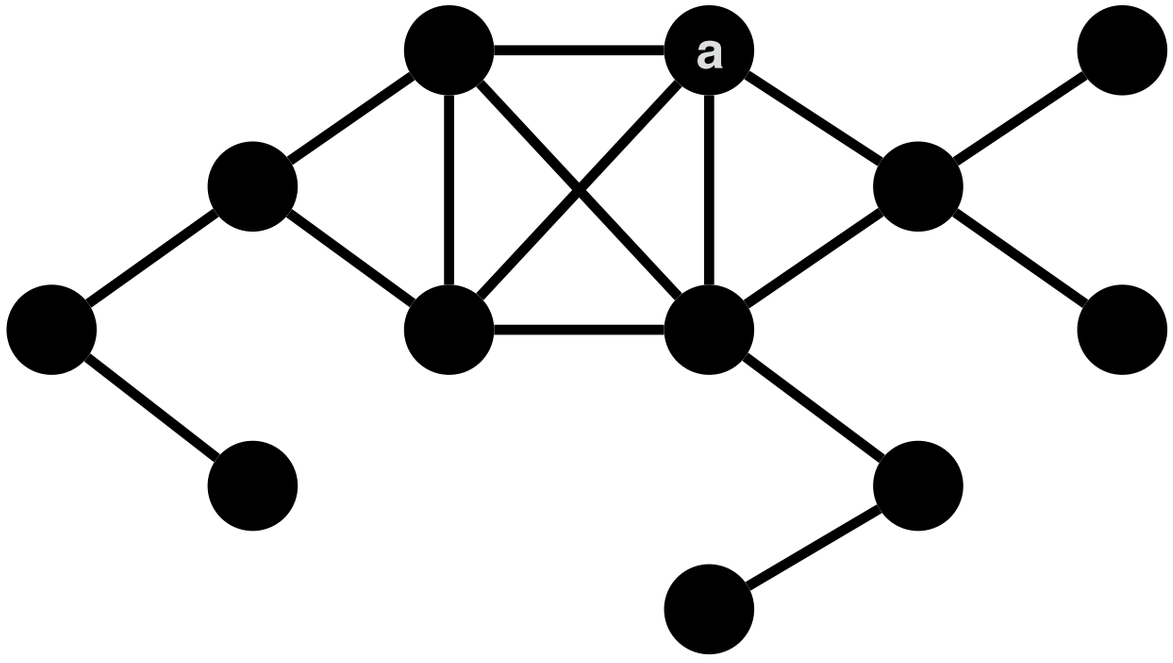
k-core : maximal connected subgraph of G s.t. all vertices have degree $\geq k$

$\lambda(v)$: largest k-core that v participates in

Example: k-core and coreness

k-core : maximal connected subgraph of G s.t. all vertices have degree $\geq k$

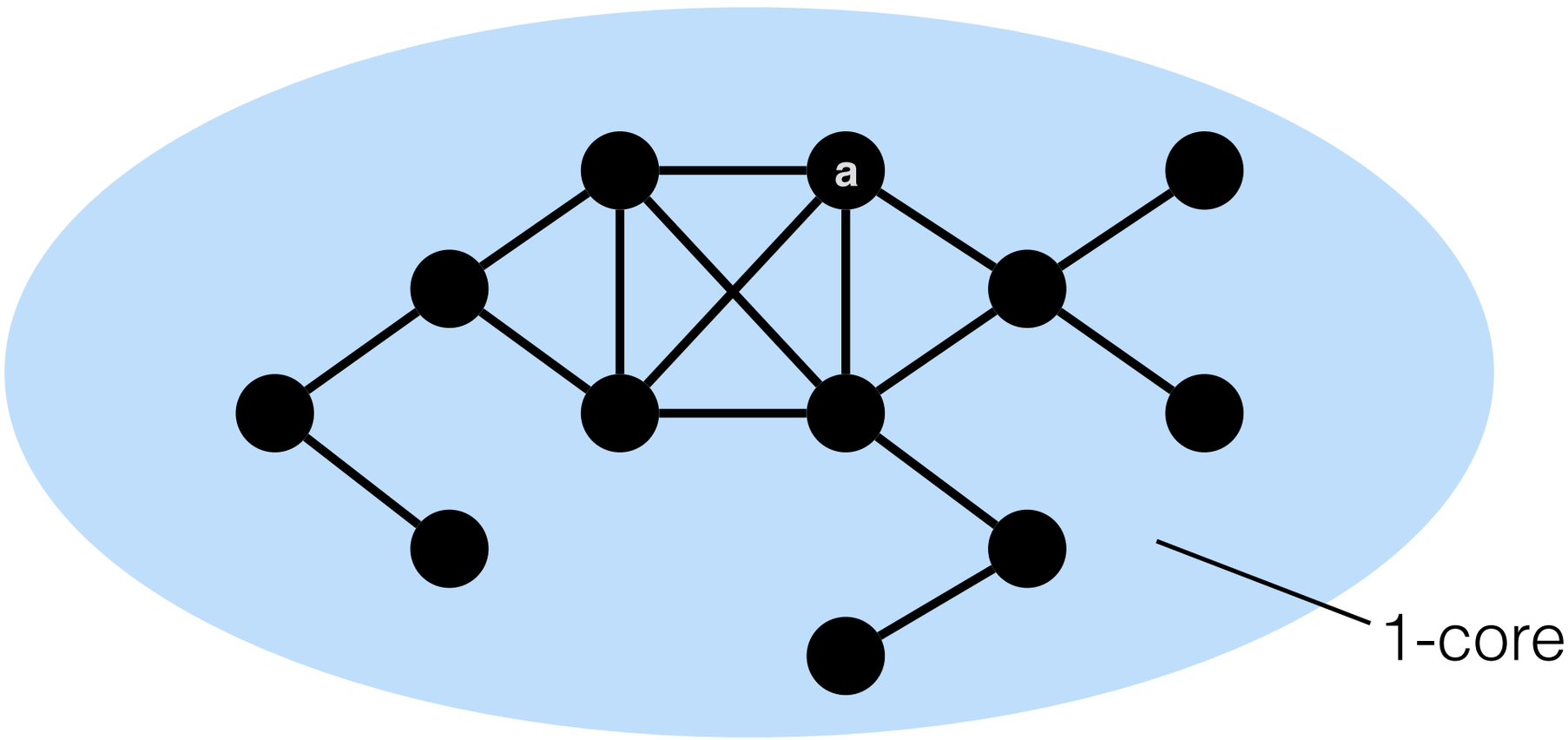
$\lambda(v)$: largest k-core that v participates in



Example: k-core and coreness

k-core : maximal connected subgraph of G s.t. all vertices have degree $\geq k$

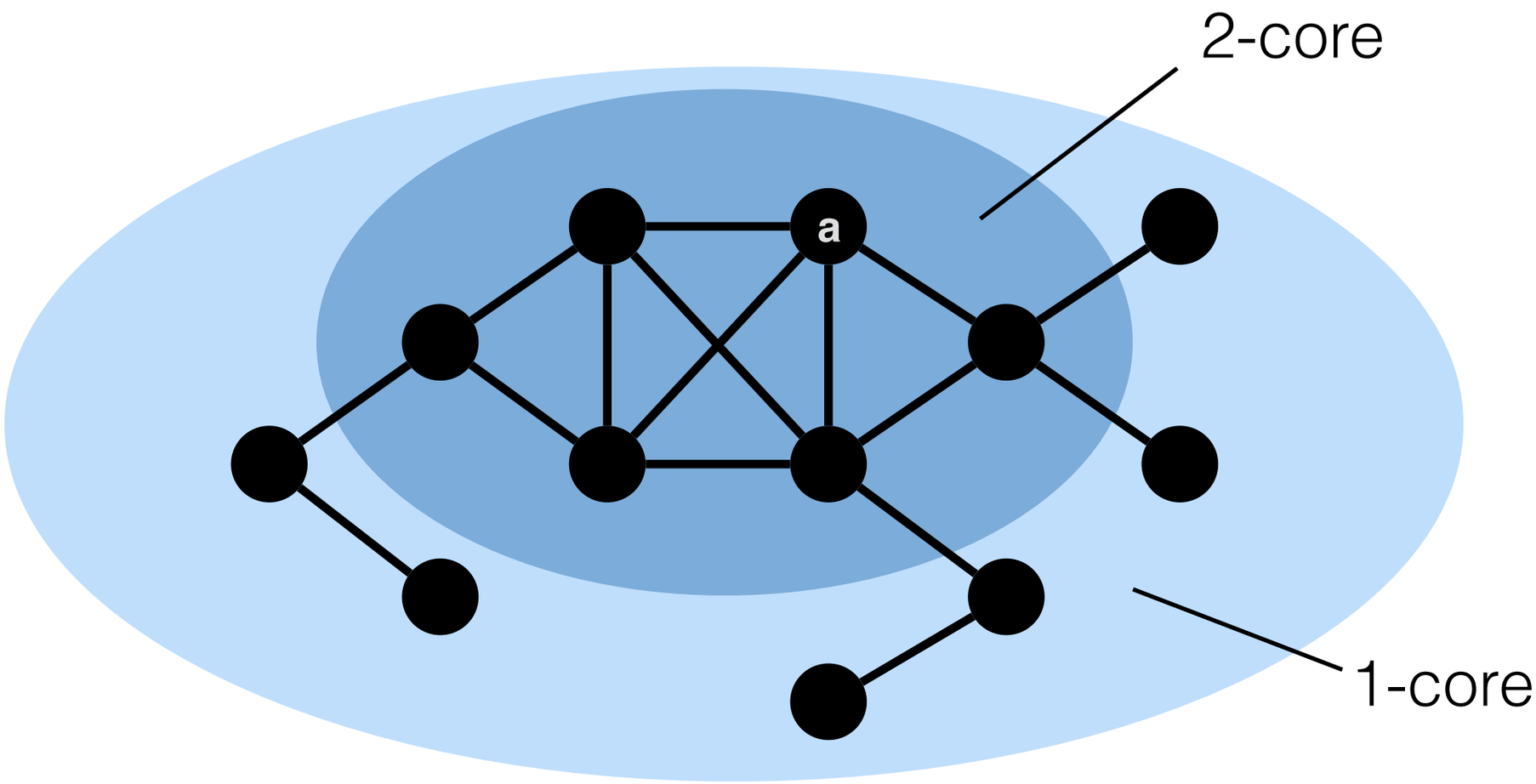
$\lambda(v)$: largest k-core that v participates in



Example: k-core and coreness

k-core : maximal connected subgraph of G s.t. all vertices have degree $\geq k$

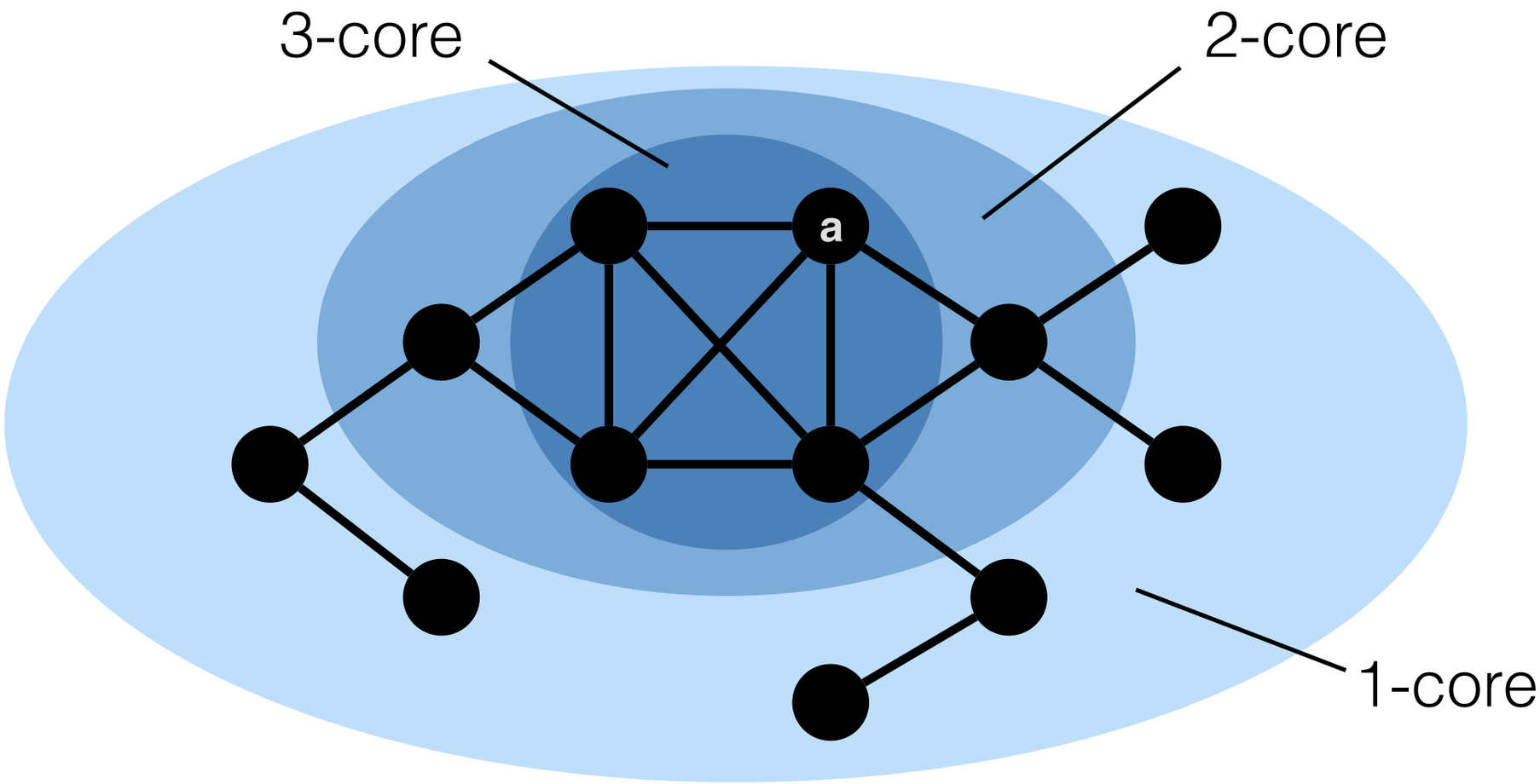
$\lambda(v)$: largest k-core that v participates in



Example: k-core and coreness

k-core : maximal connected subgraph of G s.t. all vertices have degree $\geq k$

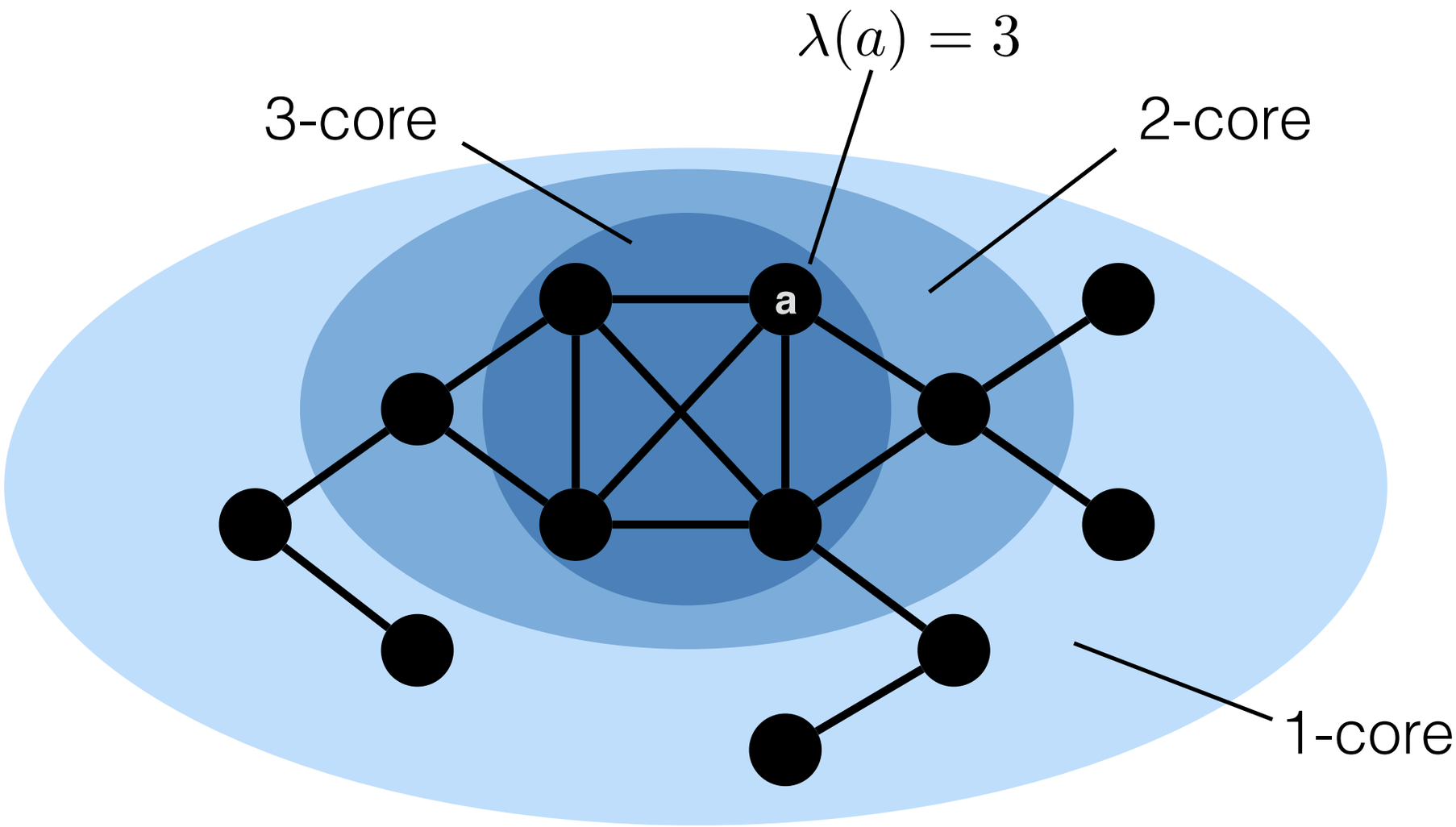
$\lambda(v)$: largest k-core that v participates in



Example: k-core and coreness

k-core : maximal connected subgraph of G s.t. all vertices have degree $\geq k$

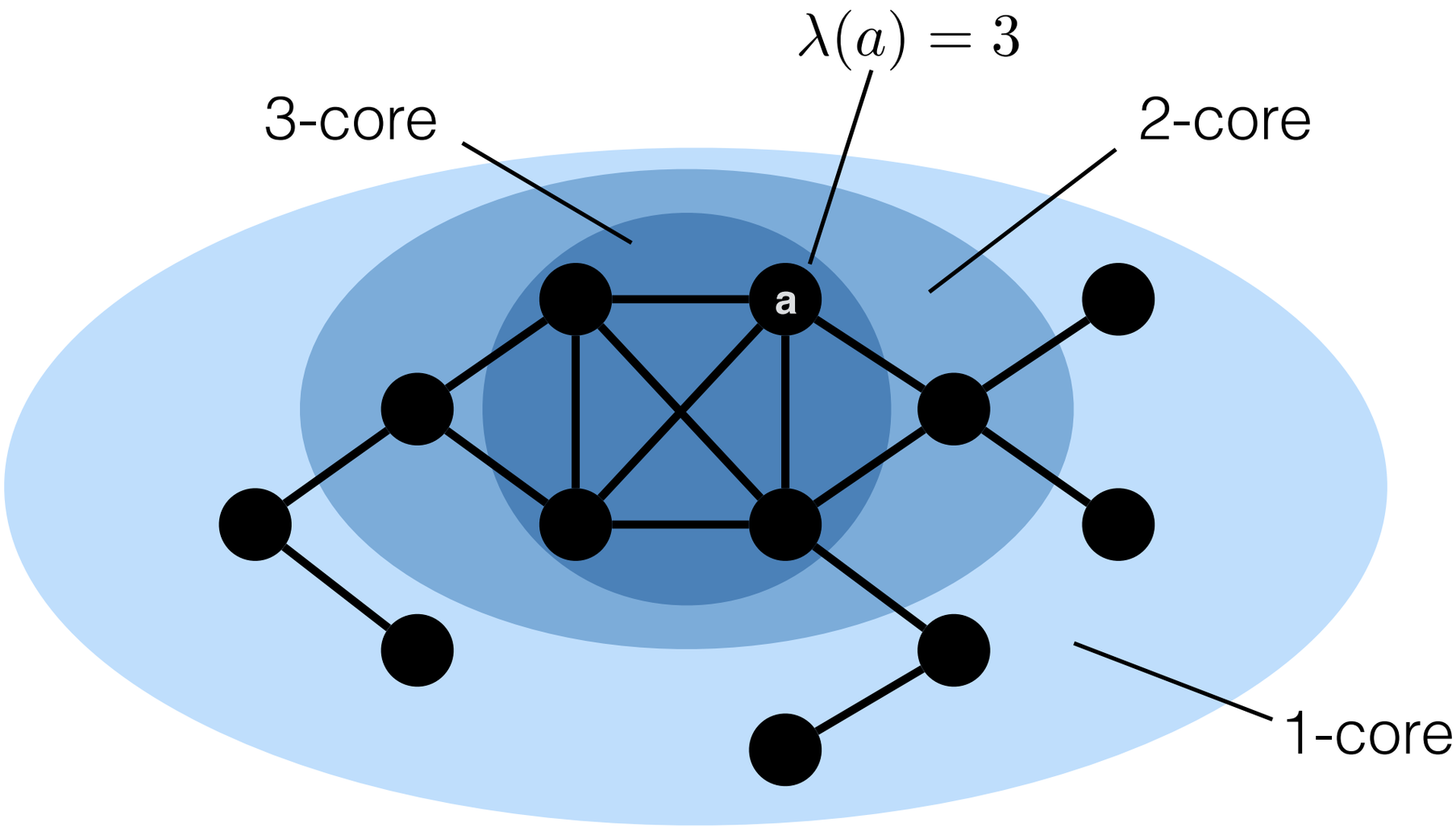
$\lambda(v)$: largest k-core that v participates in



Example: k-core and coreness

k-core : maximal connected subgraph of G s.t. all vertices have degree $\geq k$

$\lambda(v)$: largest k-core that v participates in



Can efficiently compute k-cores after computing coreness

k-core and Coreness

Sequential Peeling:

- Bucket sort vertices by degree
- Remove the minimum degree vertex, set its core number
    - Update the buckets of its neighbors

k-core and Coreness

Sequential Peeling:

- Bucket sort vertices by degree
- Remove the minimum degree vertex, set its core number
  - Update the buckets of its neighbors

Each vertex and edge is processed exactly once:

$$W = O(|E| + |V|)$$

k-core and Coreness

Sequential Peeling:

- Bucket sort vertices by degree
- Remove the minimum degree vertex, set its core number
  - Update the buckets of its neighbors

Each vertex and edge is processed exactly once:

$$W = O(|E| + |V|)$$

Existing parallel algorithms:

- Scan all remaining vertices when computing each core

k-core and Coreness

Sequential Peeling:

- Bucket sort vertices by degree
- Remove the minimum degree vertex, set its core number
  - Update the buckets of its neighbors

Each vertex and edge is processed exactly once:

$$W = O(|E| + |V|)$$

Existing parallel algorithms:

- Scan all remaining vertices when computing each core

$\rho =$ number of peeling steps done by the parallel algorithm

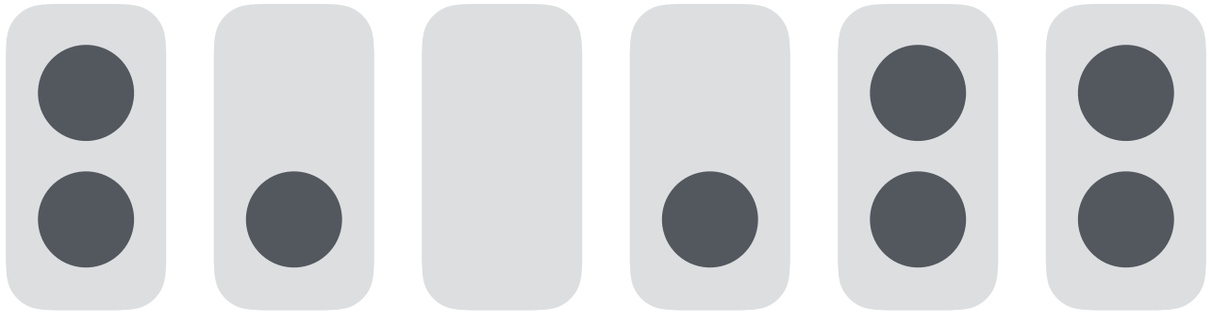
$$W = O(|E| + \rho|V|)$$

$$D = O(\rho \log |V|)$$

Work-efficient Peeling

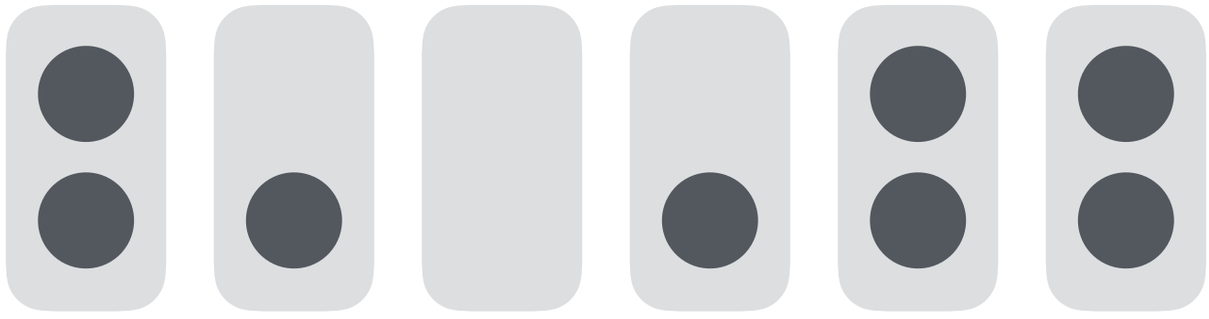
Insert vertices in bucket structure by degree

Work-efficient Peeling



Insert vertices in bucket structure by degree

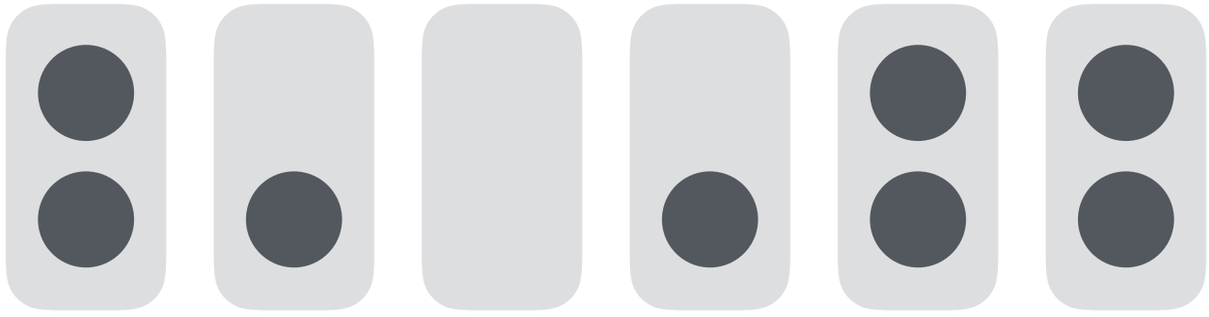
Work-efficient Peeling



Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

Work-efficient Peeling

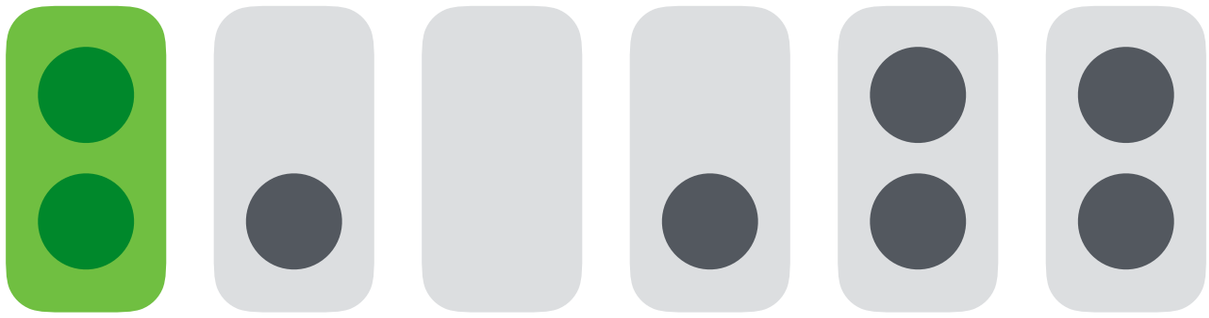


Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next bucket, set core numbers

Work-efficient Peeling

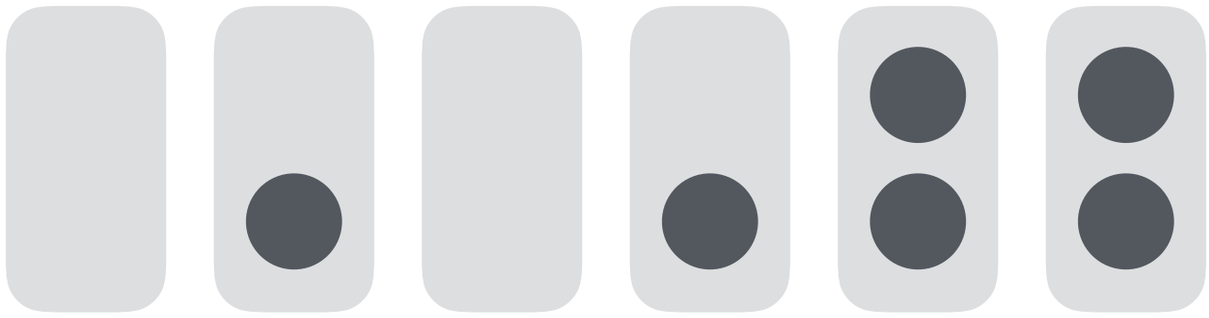


Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next bucket, set core numbers

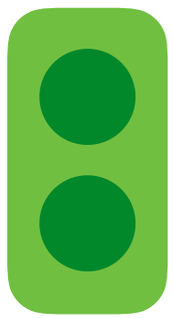
Work-efficient Peeling



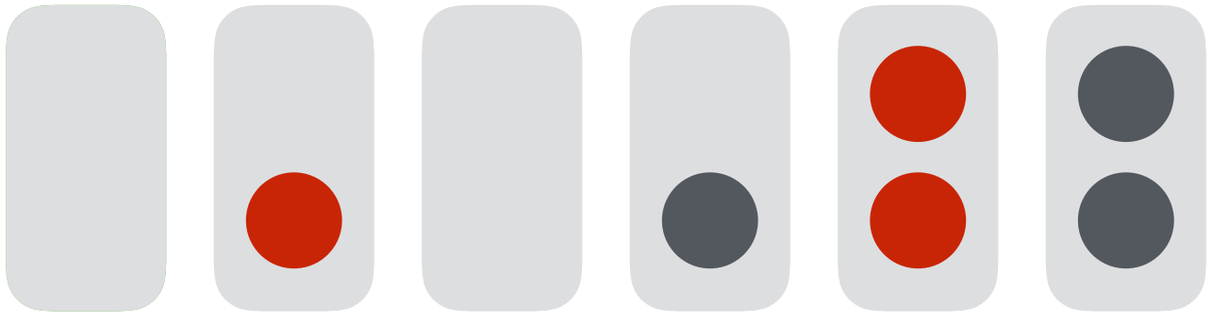
Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next bucket, set core numbers



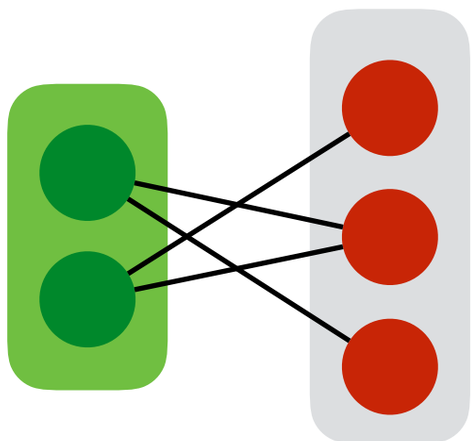
Work-efficient Peeling



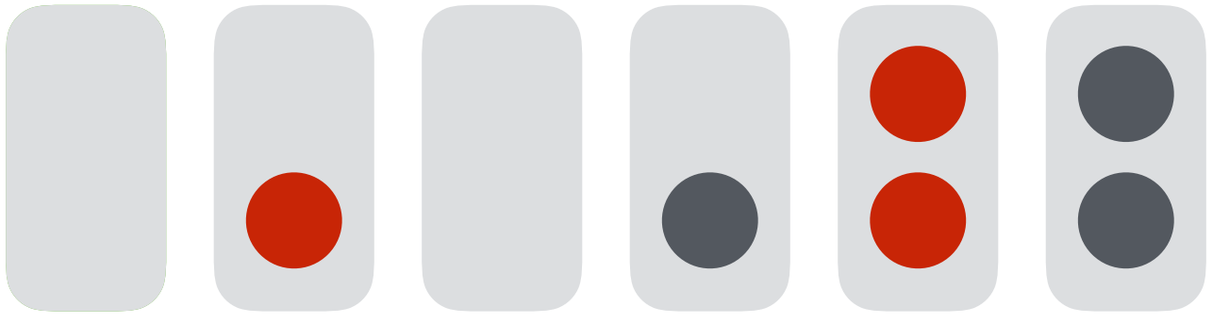
Insert vertices in bucket structure by degree

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



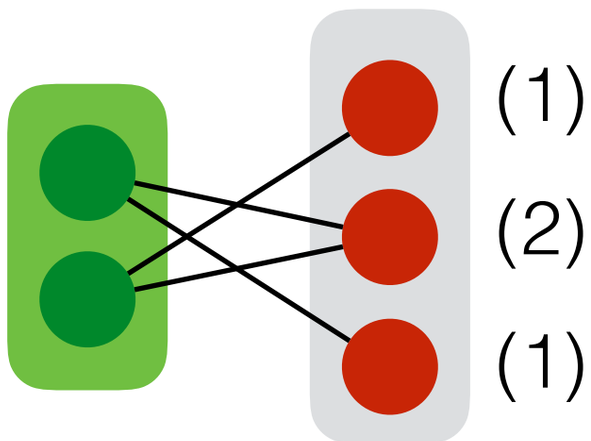
Work-efficient Peeling



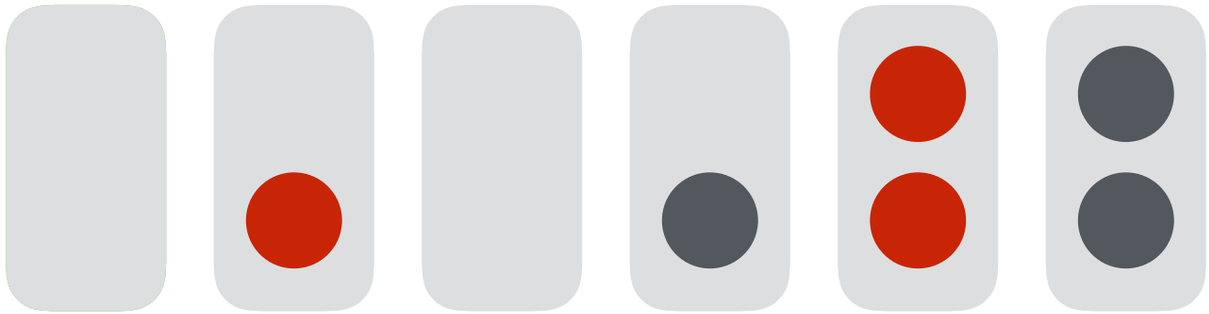
Insert vertices in bucket structure by degree

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



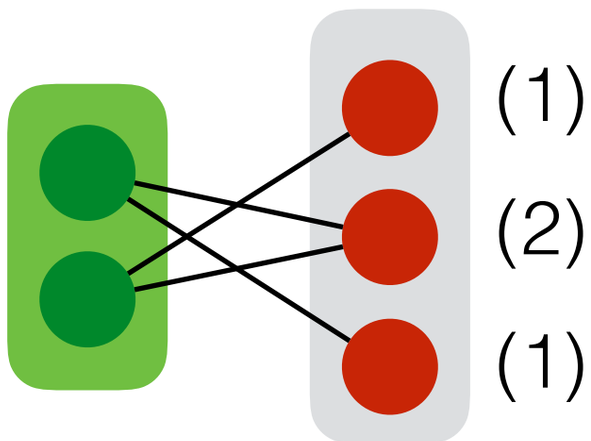
Work-efficient Peeling



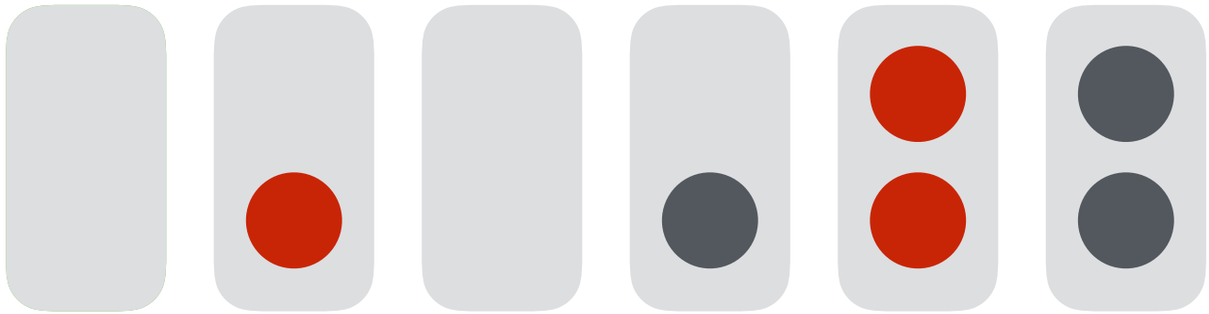
Insert vertices in bucket structure by degree

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



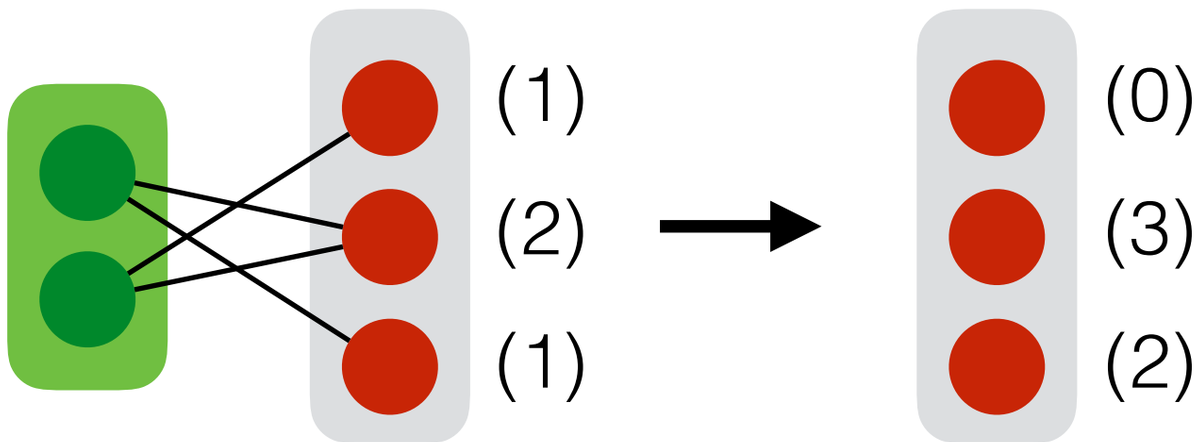
Work-efficient Peeling



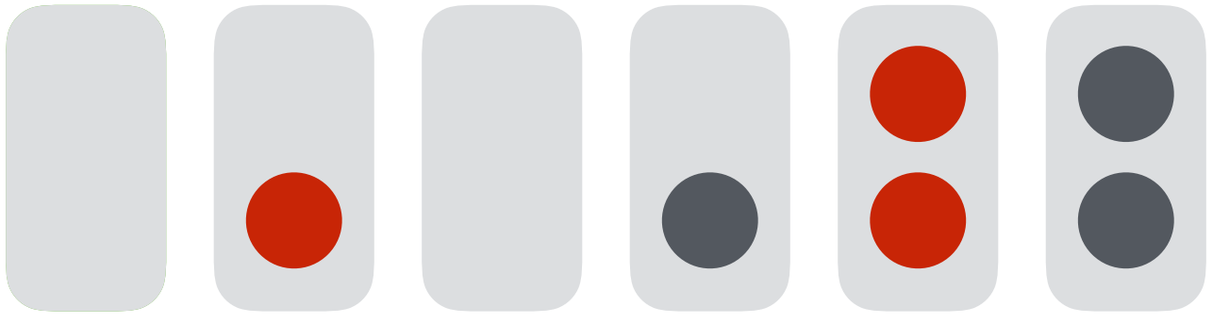
Insert vertices in bucket structure by degree

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



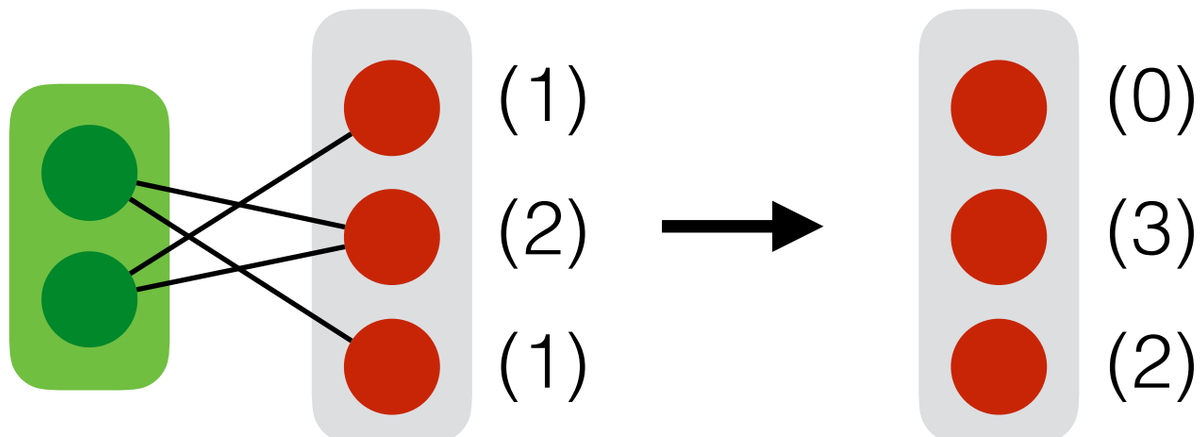
Work-efficient Peeling



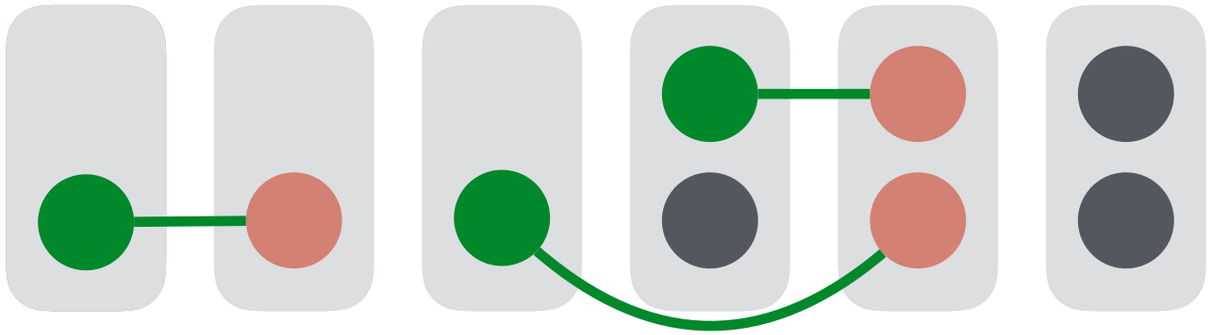
Insert vertices in bucket structure by degree

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)



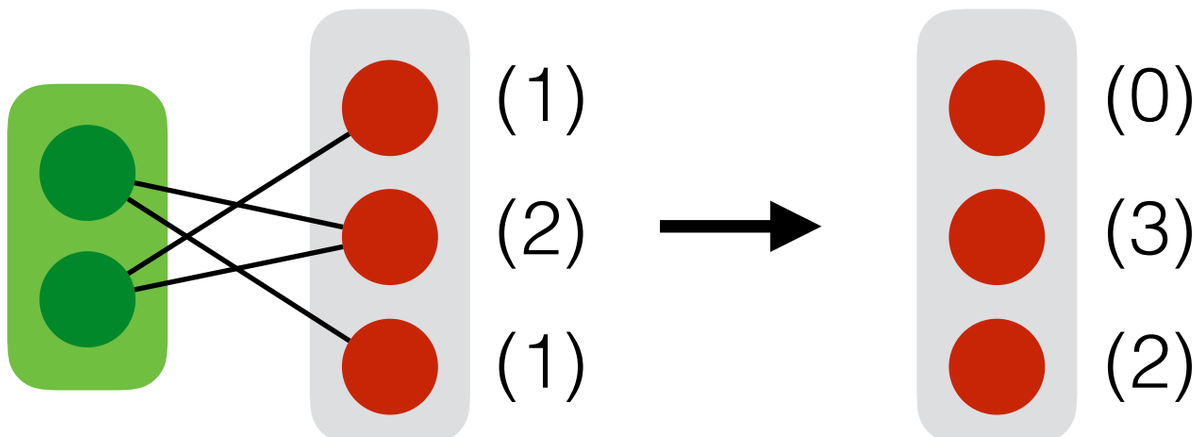
Work-efficient Peeling



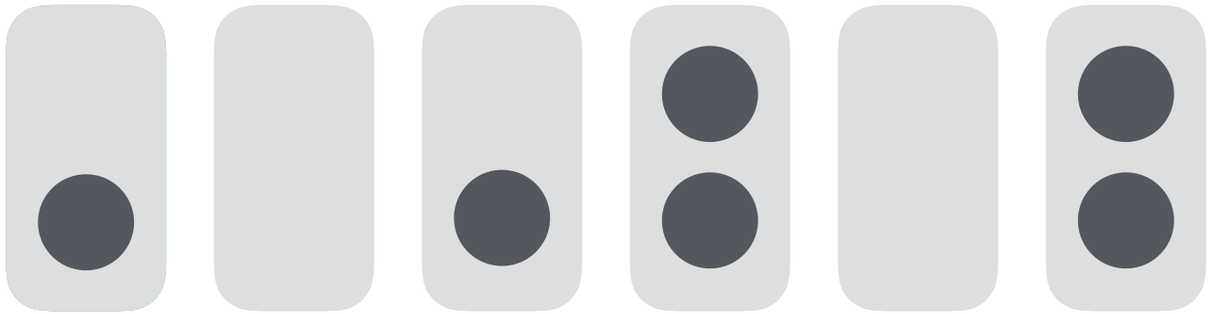
Insert vertices in bucket structure by degree

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)



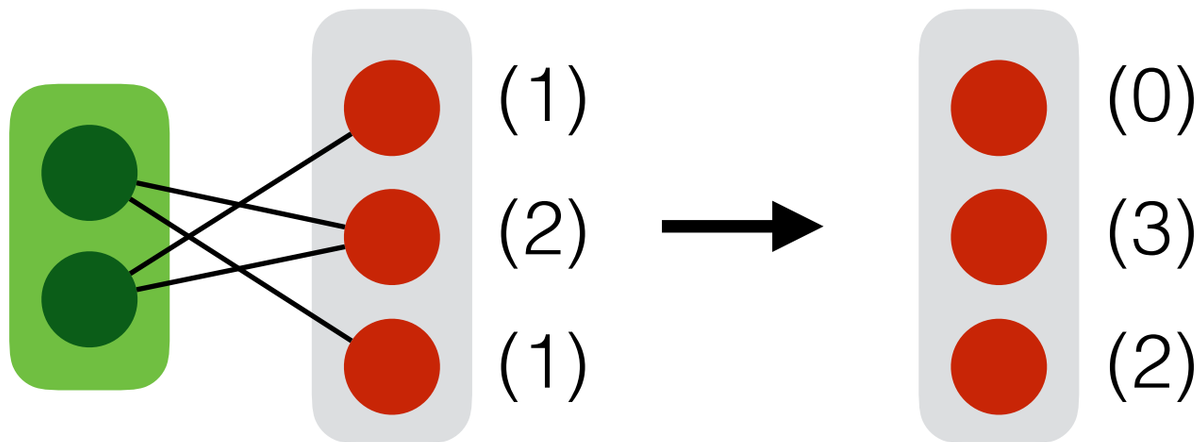
Work-efficient Peeling



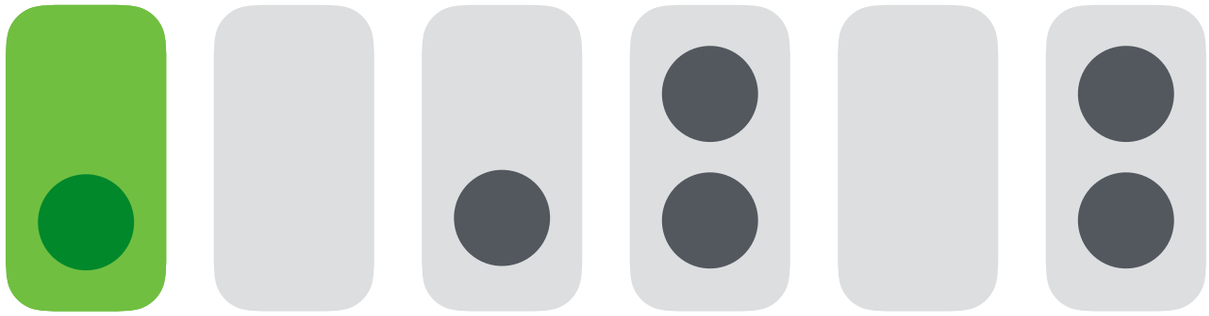
Insert vertices in bucket structure by degree

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)



Work-efficient Peeling



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)

Work-efficient Peeling

Work-efficient Peeling

We process each edge at most once in each direction:

Work-efficient Peeling

We process each edge at most once in each direction:

updates $= O(|E|)$

Work-efficient Peeling

We process each edge at most once in each direction:

updates $= O(|E|)$

buckets $\leq |V|$

Work-efficient Peeling

We process each edge at most once in each direction:

updates $= O(|E|)$

buckets $\leq |V|$

calls to NextBucket $= \rho$

Work-efficient Peeling

We process each edge at most once in each direction:

updates $= O(|E|)$

buckets $\leq |V|$

calls to NextBucket $= \rho$

calls to UpdateBuckets $= \rho$

Work-efficient Peeling

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.

Work-efficient Peeling

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, $\rho = 130,728$

Work-efficient Peeling

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

Work-efficient Peeling

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-inefficient algorithm does not terminate within 3 hours

Efficient peeling using Julienne

Summary of results

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

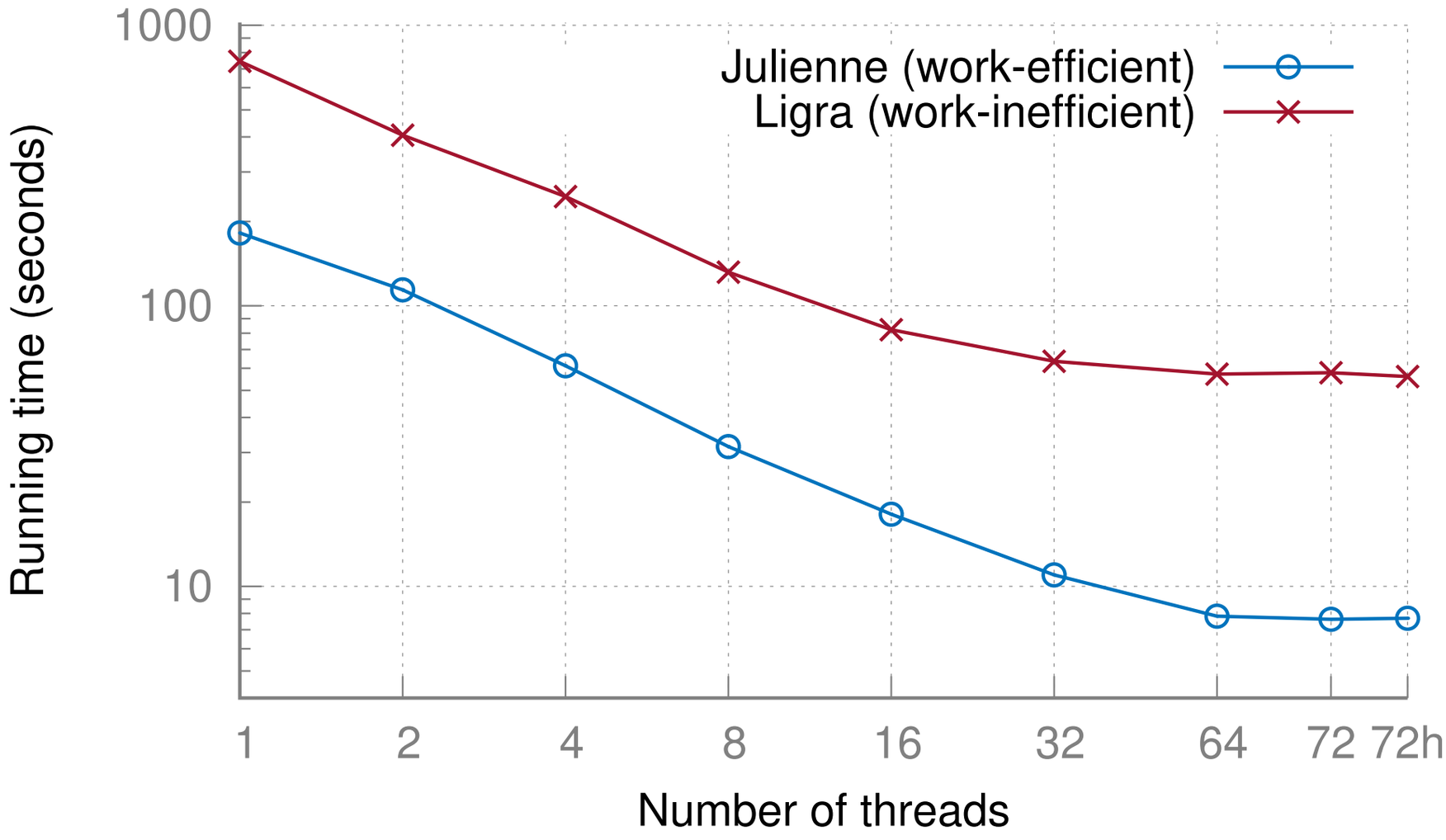
w_Δ, d_Δ : work and number of rounds of the delta-stepping algorithm

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

Experiments: k-core



Friendster

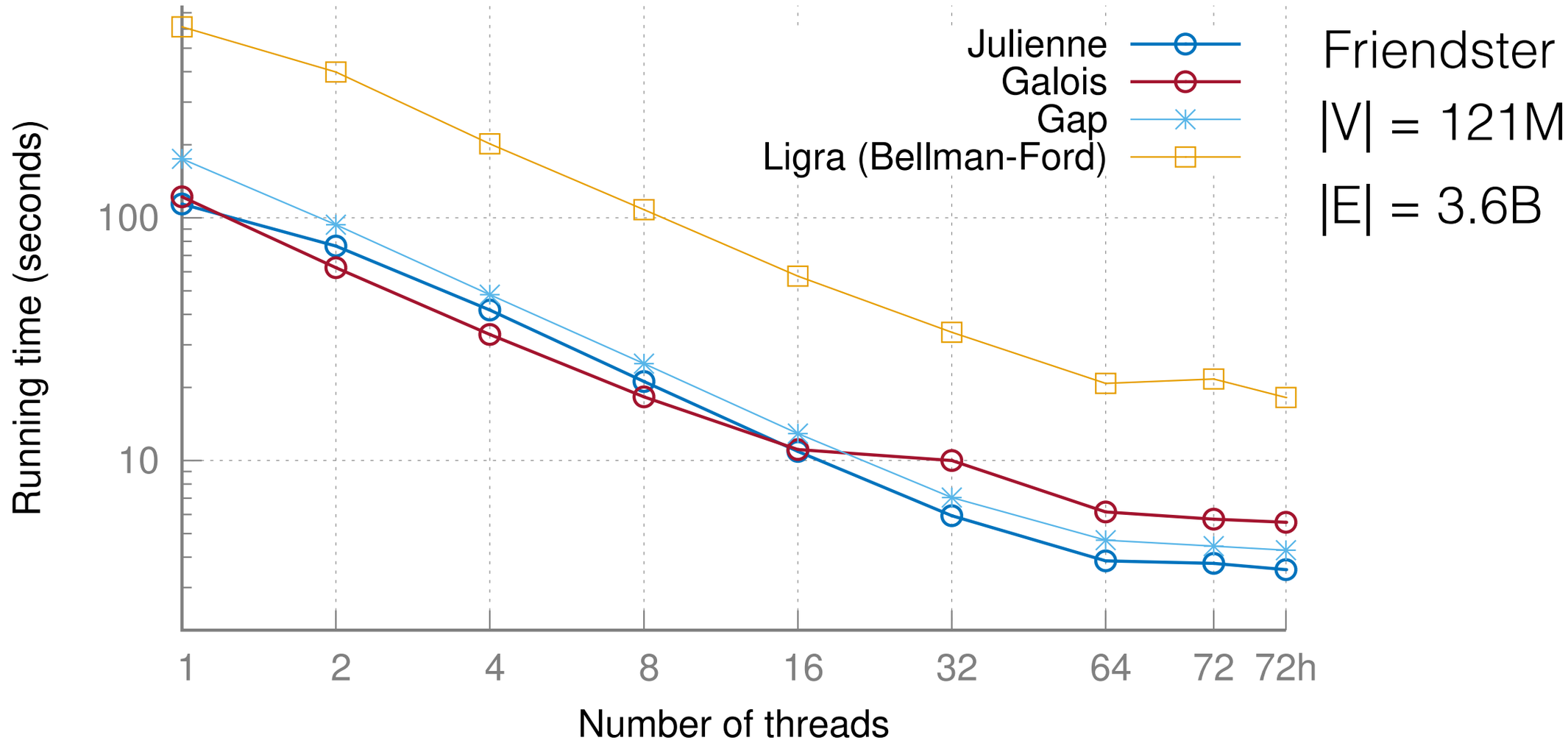
|V| = 121M

|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

Experiments: Delta-stepping



Across all inputs:

- 18-32x self-relative speedup, 17-30x speedup over DIMACS solver
- 1.1-1.7x faster than best existing implementation of Delta-Stepping
- 1.8-5.2x faster than (work-inefficient) Bellman-Ford

Experiments: 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 ~2TB 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

Experiments: Hyperlink Graphs

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

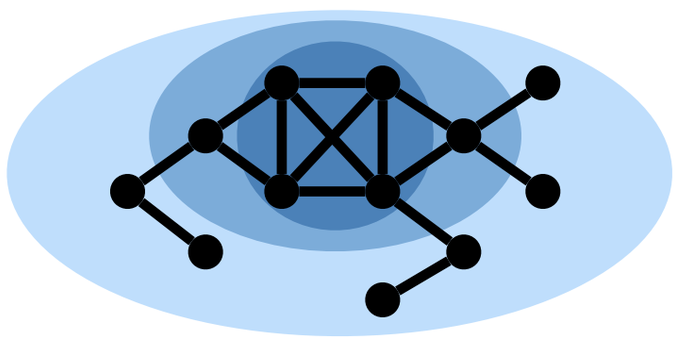
Running time in seconds on 72 cores with hyperthreading

- Able to process in main-memory of 1TB machine by compressing
- 23-43x speedup across applications
- Compression is crucial
  - Julienne/Ligra codes run without any modifications
  - Can’t run other codes on these graphs without significant effort

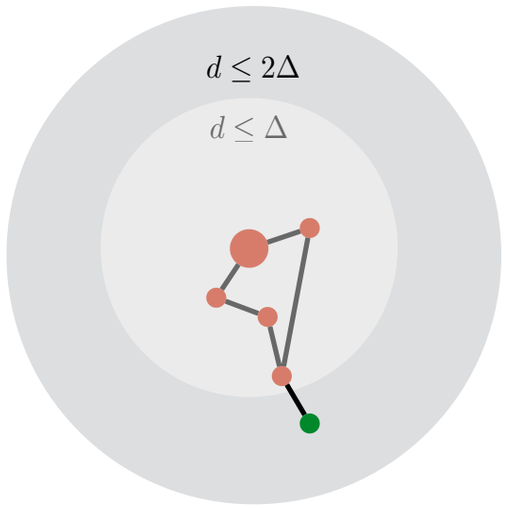
Conclusion

Julienne: framework for *bucketing-based algorithms*

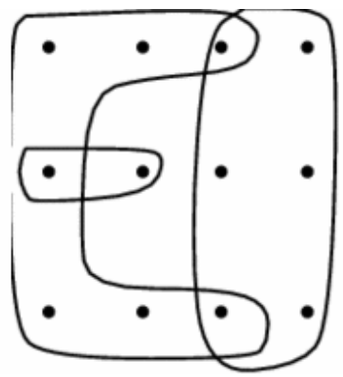
k-core



Delta-stepping wBFS



Parallel Approximate Set Cover

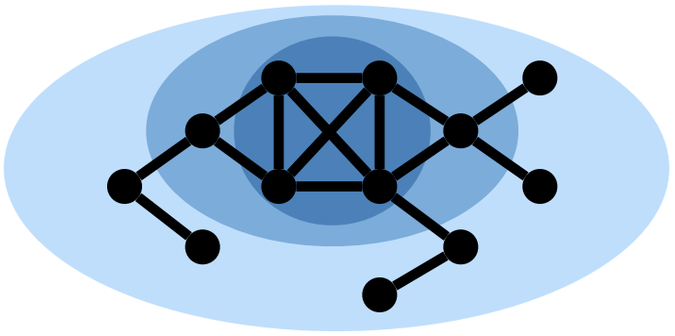


Conclusion

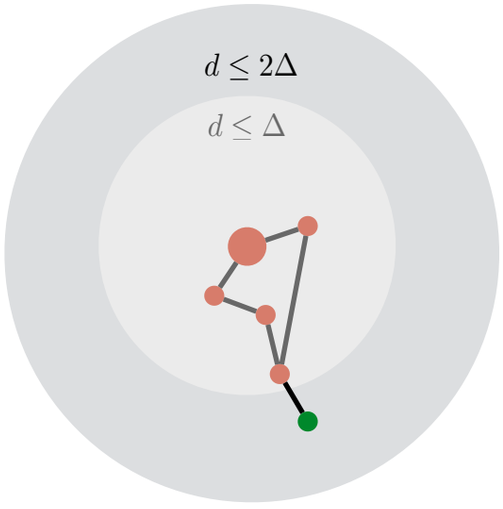
Julienne: framework for *bucketing-based algorithms*

- Codes:
  - Simple (< 100 lines each)
  - Theoretically efficient
  - Good performance in practice
  - Code will be included as part of github.com/jshun/ligra
- Future work: Trusses, Nucleus Decomposition, Densest Subgraph

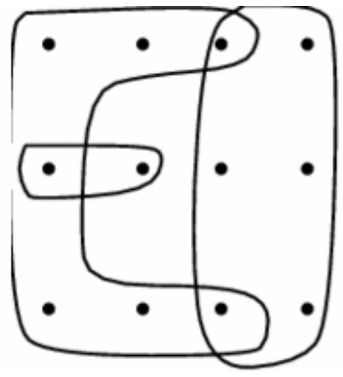
k-core



Delta-stepping wBFS



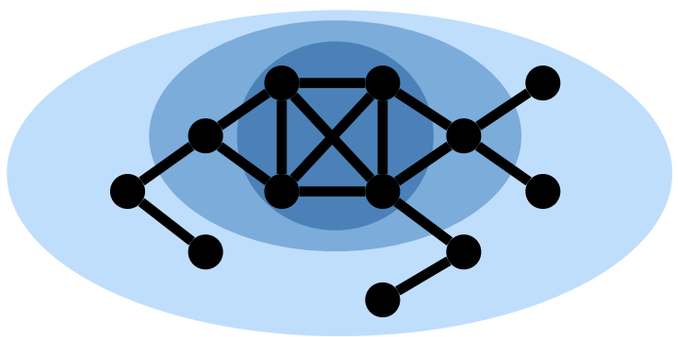
Parallel Approximate Set Cover



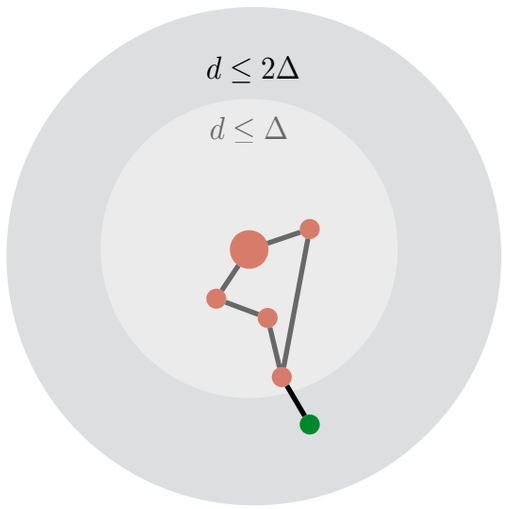
Thank you!

Please feel free to reach out to ldhulipa@cs.cmu.edu

k-core



Delta-stepping wBFS



Parallel Approximate Set Cover