Making Graph Computations Fast, Simple, and Portable

Yunming Zhang and Collaborators

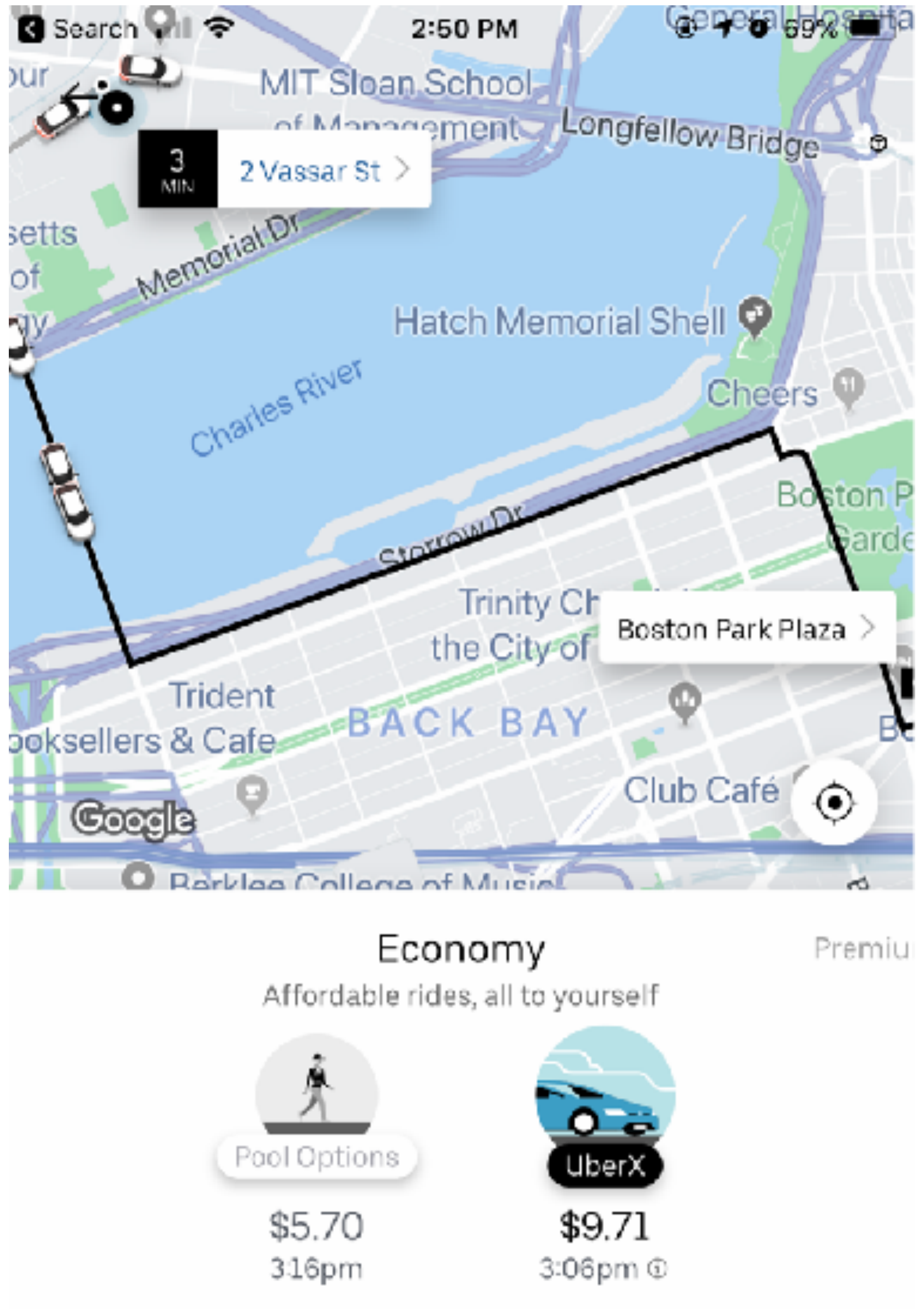
CARNEGIE MELLON UNIVERSITY PITTSBURGH PENNSYLVANIA 1900

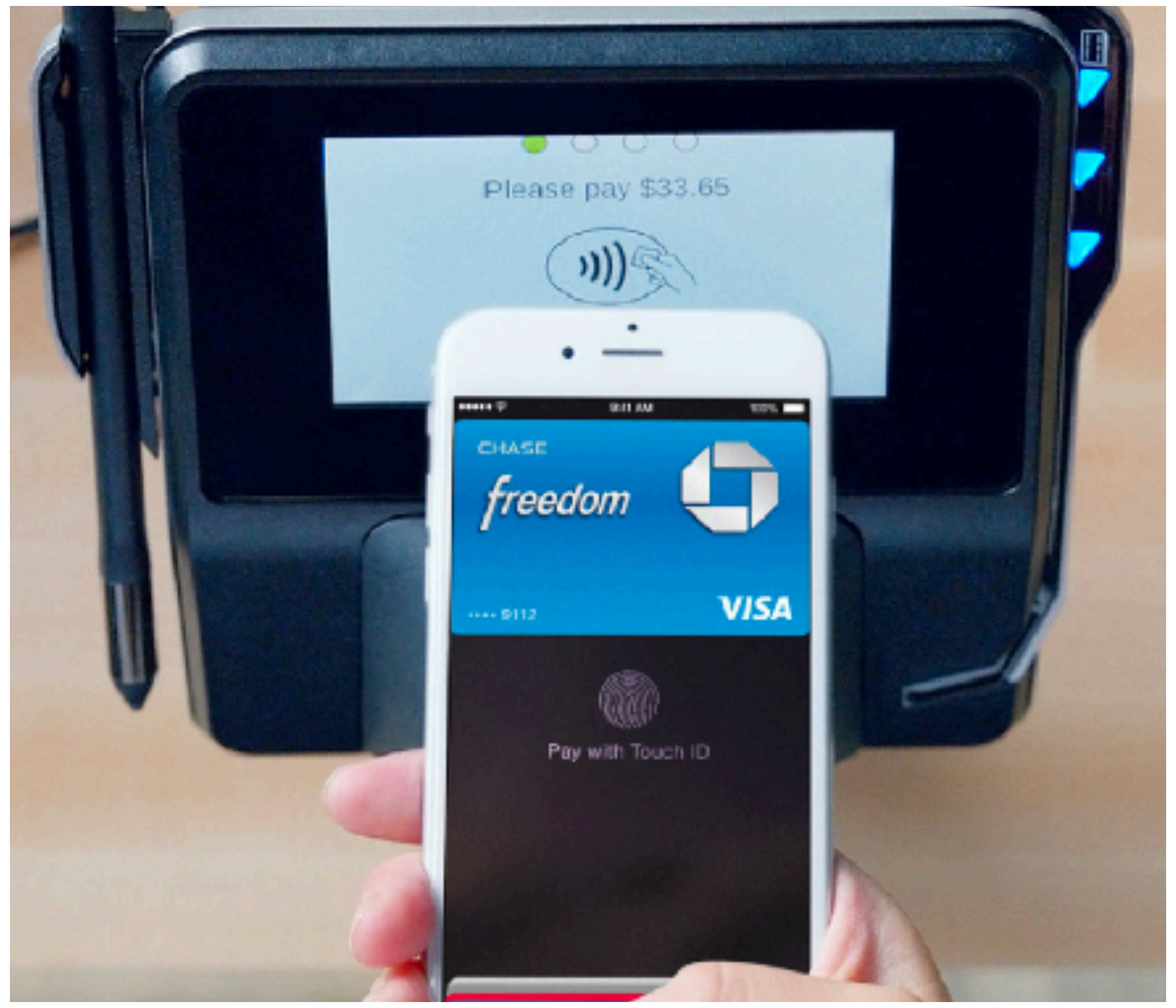
Massachusetts
Institute of
Technology

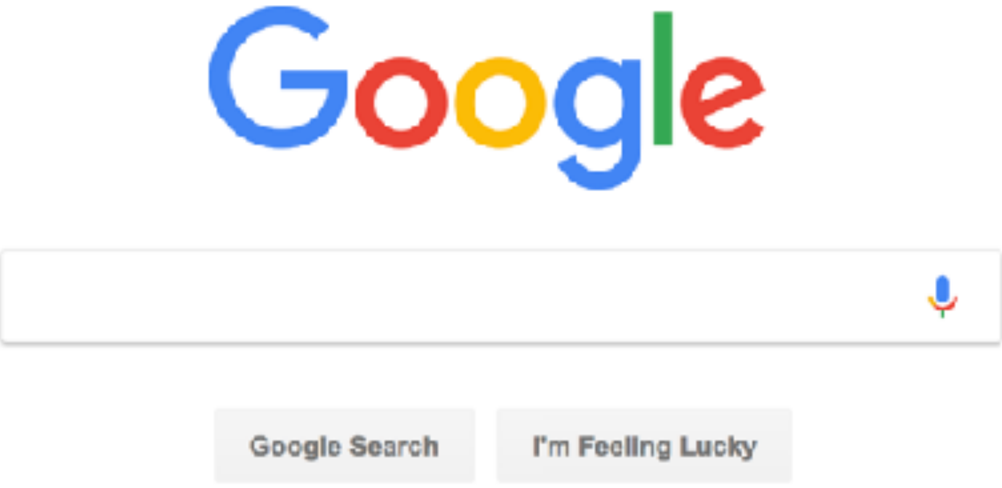
CSAIL
MIT COMPUTER SCIENCE AND ARTII

Adobe Research

Graphs Are Everywhere



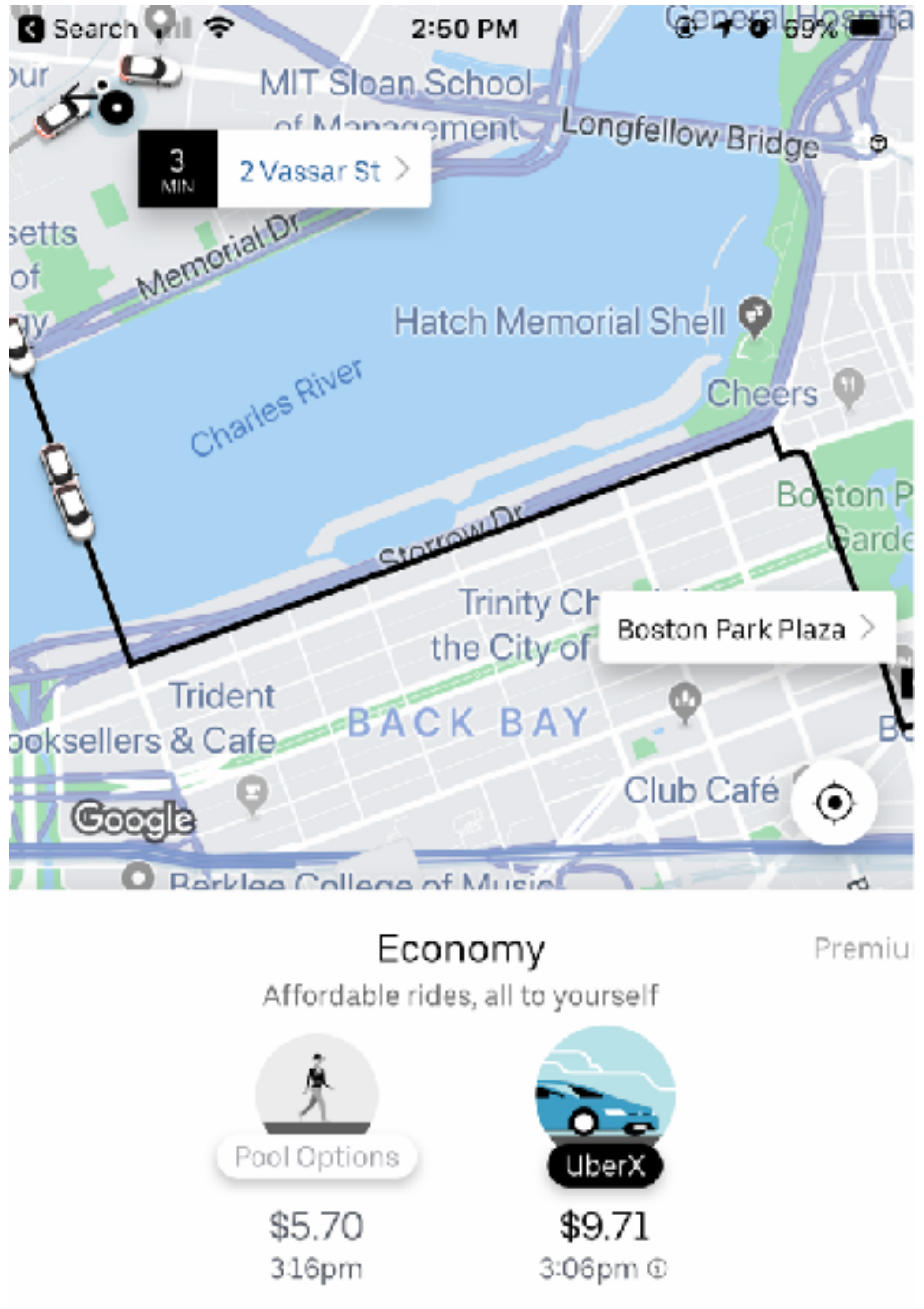


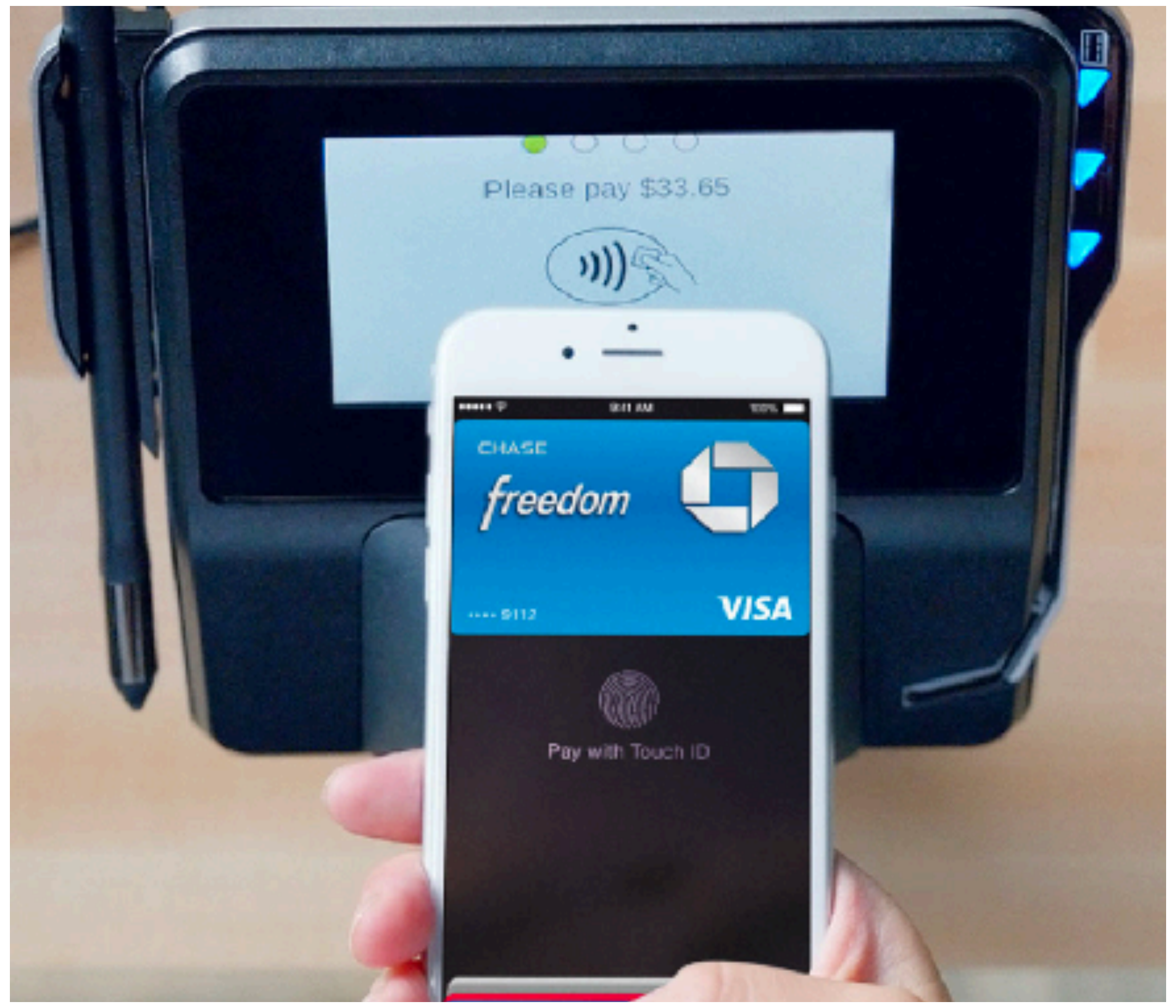


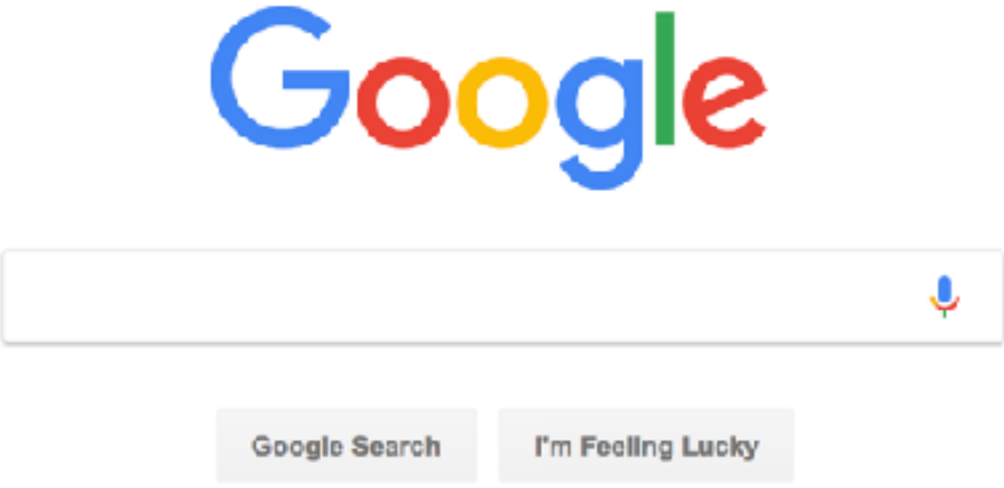
Recommendations for You, Yunming

http://google.com . https://en.wikipedia.org/wiki/Polygon_mesh#/media/File:Dolphin_triangle_mesh.png, https://www.bankinfosecurity.com/webinars/customer-awareness-what-works-in-fraud-detection-prevention-w-423 http://amazon.com https://makeawebsitehub.com/social-media-sites/

Performance is Important







Recommendations for You, Yunming

http://google.com . https://en.wikipedia.org/wiki/Polygon_mesh#/media/File:Dolphin_triangle_mesh.png, https://www.bankinfosecurity.com/webinars/customer-awareness-what-works-in-fraud-detection-prevention-w-423 http://amazon.com https://makeawebsitehub.com/social-media-sites/

World is Built for Dense

- Hardware Utilization

- Programming System

World is Built for Dense

- Hardware Utilization
  - Peak Performance (GEMM)
    - 70-80% of CPU
    - 80-90% of GPU
  - Optimizations
    - Prefetching, Branch Predictions, TLB, cache, ..

- Programming System

World is Built for Dense

- Hardware Utilization
  - Peak Performance (GEMM)
    - 70-80% of CPU
    - 80-90% of GPU
  - Optimizations
    - Prefetching, Branch Predictions, TLB, cache, ..

- Programming System
  - Abstractions that Work across Different Algorithms (Dense Linear Algebra, Image Processing, Deep Learning, ..)
    - BLAS, Halide, TensorFlow, …
    - Optimizing Compilers
      - Tiling, Vectorization, Unrolling, ..

Not Ready for Graphs

- Hardware Utilization

- Programming System

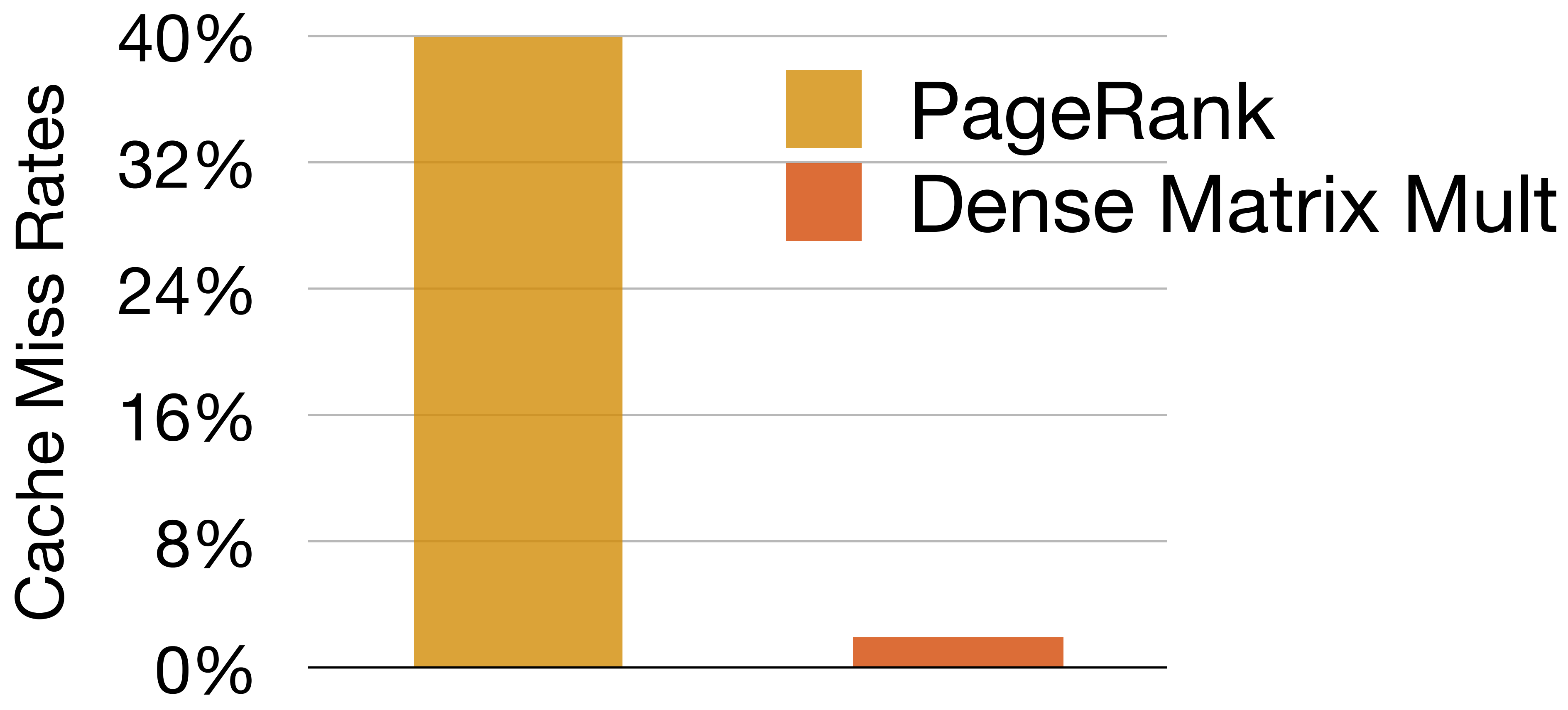
Not Ready for Graphs

- Hardware Utilization
  - Peak Performance (PageRank, SpMv)
    - < 10% Peak of CPU and GPU

- Programming System

Not Ready for Graphs

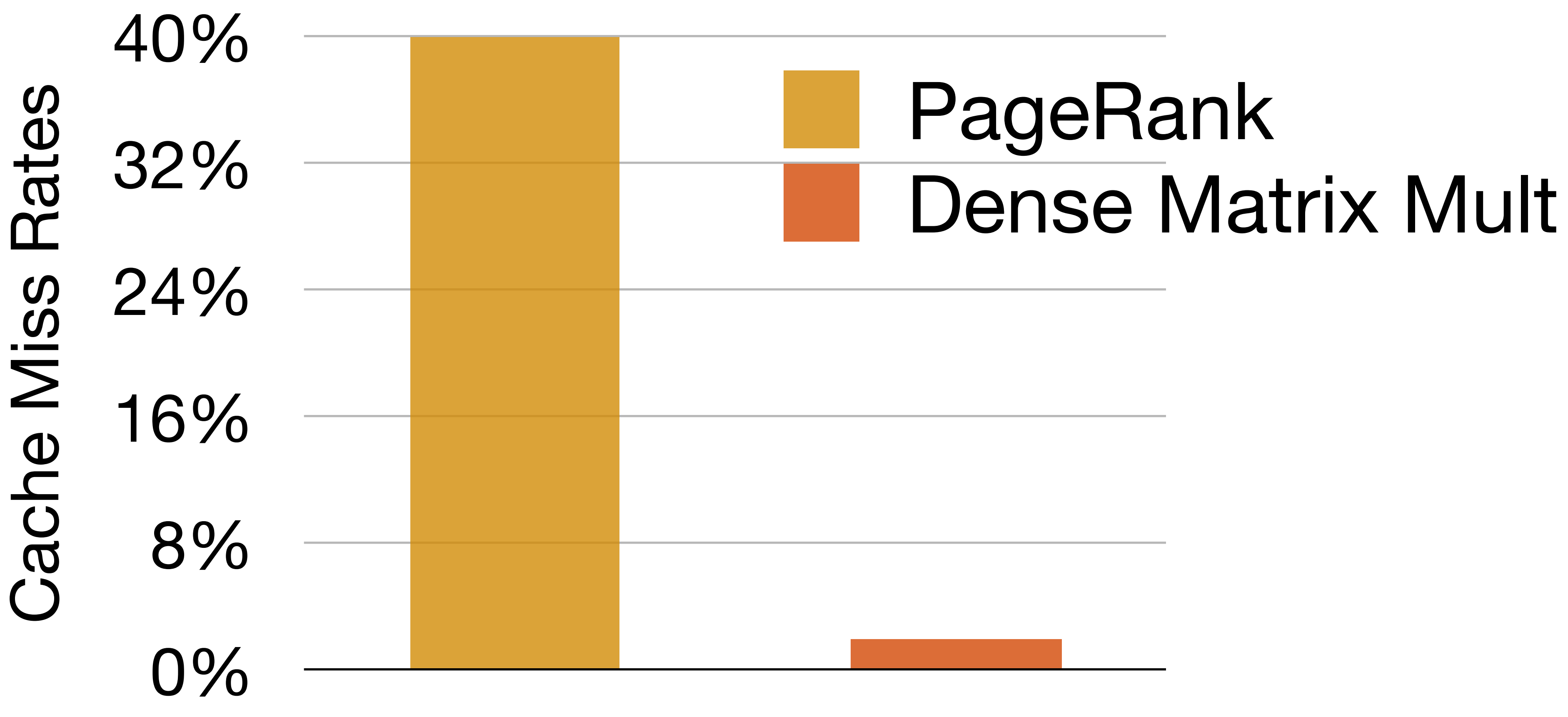
- Hardware Utilization
  - Peak Performance (PageRank, SpMv)
    - < 10% Peak of CPU and GPU



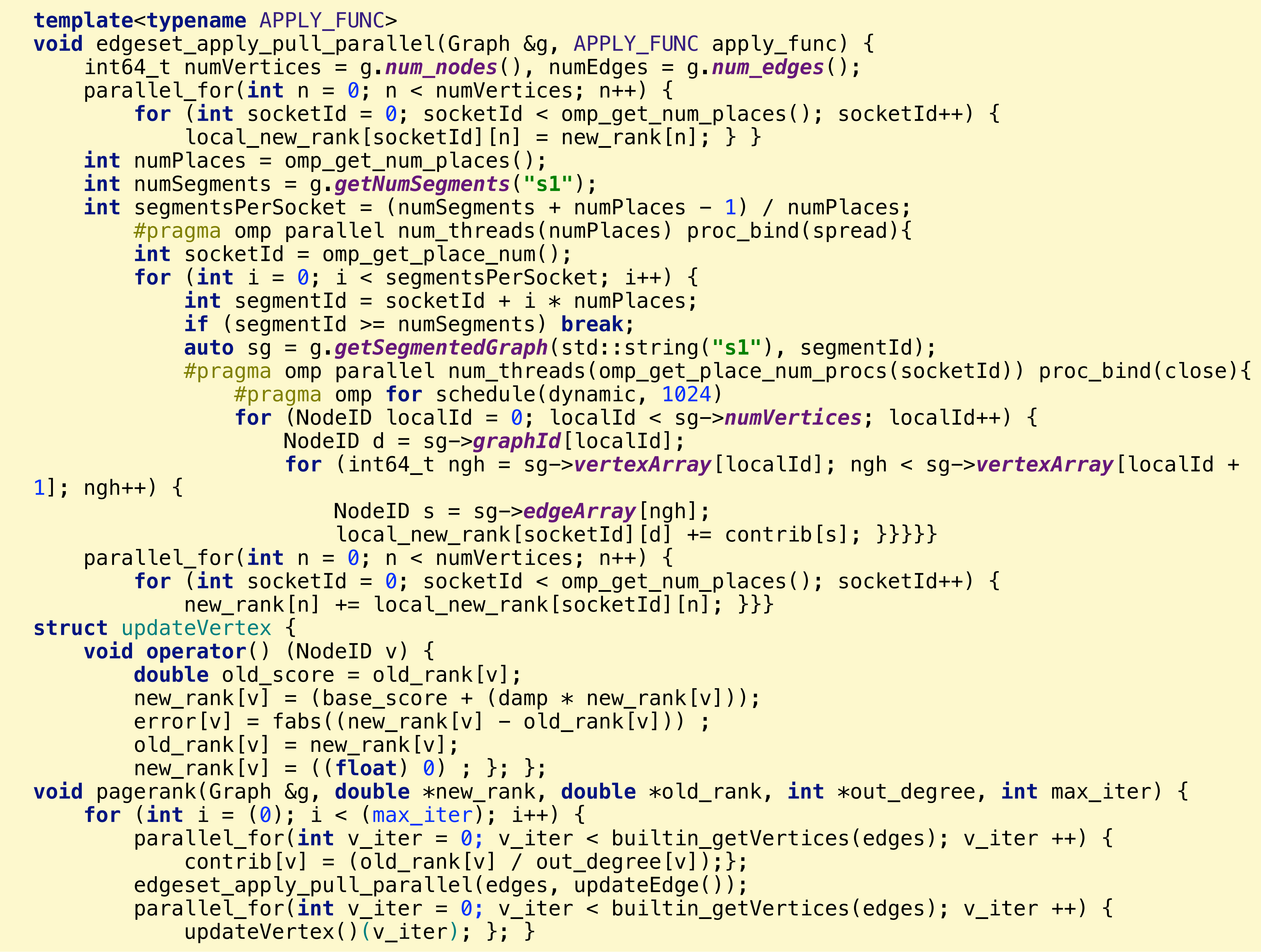
- Programming System

Not Ready for Graphs

- Hardware Utilization
  - Peak Performance (PageRank, SpMv)
    - < 10% Peak of CPU and GPU



- Programming System



```
template<typename APPLY_FUNC>
void edgeset_apply_pull_parallel(Graph &g, APPLY_FUNC apply_func) {
    int64_t numVertices = g.num_nodes(), numEdges = g.num_edges();
    parallel_for(int n = 0; n < numVertices; n++) {
        for (int socketId = 0; socketId < omp_get_num_places(); socketId++) {
            local_new_rank[socketId][n] = new_rank[n]; } }
    int numPlaces = omp_get_num_places();
    int numSegments = g.getNumSegments("s1");
    int segmentsPerSocket = (numSegments + numPlaces - 1) / numPlaces;
        #pragma omp parallel num_threads(numPlaces) proc_bind(spread){
        int socketId = omp_get_place_num();
        for (int i = 0; i < segmentsPerSocket; i++) {
            int segmentId = socketId + i * numPlaces;
            if (segmentId >= numSegments) break;
            auto sg = g.getSegmentedGraph(std::string("s1"), segmentId);
            #pragma omp parallel num_threads(omp_get_place_num_procs(socketId)) proc_bind(close){
                #pragma omp for schedule(dynamic, 1024)
                for (NodeID localId = 0; localId < sg->numVertices; localId++) {
                    NodeID d = sg->graphId[localId];
                    for (int64_t ngh = sg->vertexArray[localId]; ngh < sg->vertexArray[localId +
1]; ngh++) {
                        NodeID s = sg->edgeArray[ngh];
                        local_new_rank[socketId][d] += contrib[s]; }}}}}
    parallel_for(int n = 0; n < numVertices; n++) {
        for (int socketId = 0; socketId < omp_get_num_places(); socketId++) {
            new_rank[n] += local_new_rank[socketId][n]; }}}
struct updateVertex {
    void operator() (NodeID v) {
        double old_score = old_rank[v];
        new_rank[v] = (base_score + (damp * new_rank[v]));
        error[v] = fabs((new_rank[v] - old_rank[v])) ;
        old_rank[v] = new_rank[v];
        new_rank[v] = ((float) 0) ; }; };
void pagerank(Graph &g, double *new_rank, double *old_rank, int *out_degree, int max_iter) {
    for (int i = (0); i < (max_iter); i++) {
        parallel_for(int v_iter = 0; v_iter < builtin_getVertices(edges); v_iter ++) {
            contrib[v] = (old_rank[v] / out_degree[v]);};
        edgeset_apply_pull_parallel(edges, updateEdge());
        parallel_for(int v_iter = 0; v_iter < builtin_getVertices(edges); v_iter ++) {
            updateVertex()(v_iter); }; }
```

Optimized PageRank for Multi-Core CPU

Graph Computations have Variety

Data

Social Networks, Web Graphs, Road Networks, Engineering Meshes, Transaction Graphs, Network Traffic Graphs, Email Networks, Similarity Graphs, …

×

Algorithms

Breadth-first search, betweenness centrality, Bellman-Ford, Delta-stepping, collaborative filtering, Page Rank, Page Rank Delta, connected components, k-core decomposition, triangle counting, local clustering, structural clustering minimum spanning forest, eccentricity estimation, graph coloring, k-truss decomposition, nuclei decomposition, biconnectivity, set cover, maximum flow, butterfly counting, strongly connected components, graph partitioning, RDF queries, random walks, point-to-point shortest paths, A* search, low-diameter decomposition, densest subgraph, multi-source BFS, maximal independent set, maximal matching, etc…

×

Hardware

CPU, GPU, KNL, Distributed Environment, FPGA, HammerBlade, Symphony,…

Outline

Hardware Utilization

Making Caches Work for Graph Analytics (BigData17)
Zhang, et al.

Programming System to Handle Variety in Data and Algorithms

GraphIt: a High-Performance Graph DSL (OOPSLA18)
Zhang, et al.

Optimizing Ordered Graph Algorithms with GraphIt (CGO2020)
Zhang, et al.

Variety in Hardware

Universal Graph Framework (Under Submission)
Brahmakshatriya, Zhang, et al.

Outline

Hardware Utilization

Programming System to Handle Variety in Data and Algorithms

Variety in Hardware

Making Caches Work for Graph Analytics
(BigData17)
Zhang, et al.

GraphIt: a High-Performance Graph DSL
(OOPSLA18)
Zhang, et al.

Optimizing Ordered Graph Algorithms with GraphIt
(CGO2020)
Zhang, et al.

Universal Graph Framework
(Under Submission)
Brahmakshatriya, Zhang, et al.

- **Frequency-based Reordering**
- **Cache-aware Partitioning**

Outline

Hardware Utilization

Programming System to Handle Variety in Data and Algorithms

Variety in Hardware

Making Caches Work for Graph Analytics
(BigData17)
Zhang, et al.

GraphIt: a High-Performance Graph DSL
(OOPSLA18)
Zhang, et al.

Optimizing Ordered Graph Algorithms with GraphIt
(CGO2020)
Zhang, et al.

Universal Graph Framework
(Under Submission)
Brahmakshatriya, Zhang, et al.

- **Frequency-based Reordering**
- **Cache-aware Partitioning**

***GraphIt* Compiler and DSL that Decouples**
- **Algorithm**
- **Optimization**
- **Hardware**

for Graph Applications

Outline

Hardware Utilization

Programming System to Handle Variety in Data and Algorithms

Variety in Hardware

Making Caches Work for Graph Analytics (BigData17)
Zhang, et al.

GraphIt: a High-Performance Graph DSL (OOPSLA18)
Zhang, et al.

Optimizing Ordered Graph Algorithms with GraphIt (CGO2020)
Zhang, et al.

Universal Graph Framework (Under Submission)
Brahmakshatriya, Zhang, et al.

- **Frequency-based Reordering**
- **Cache-aware Partitioning**

***GraphIt* Compiler and DSL that Decouples**

- **Algorithm**
- **Optimization**
- **Hardware**

for Graph Applications

PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

PageRank

```
while …
     for node : graph.vertices
         for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
         newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

PageRank

```
while …
     for node : graph.vertices
         for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
         newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

PageRank

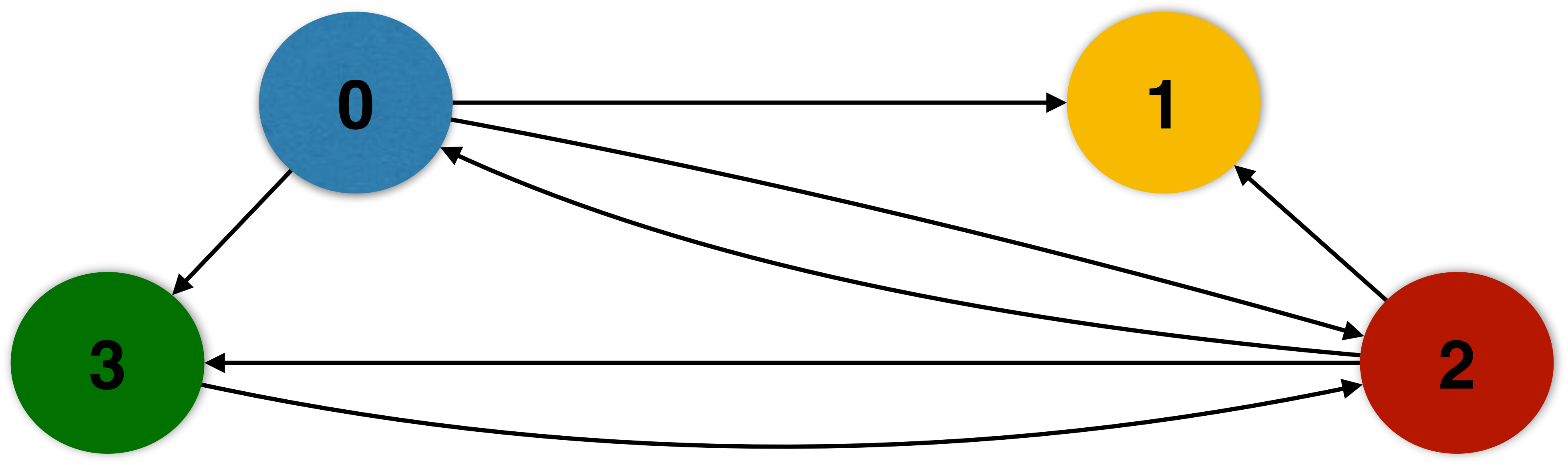
```
while …
     for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

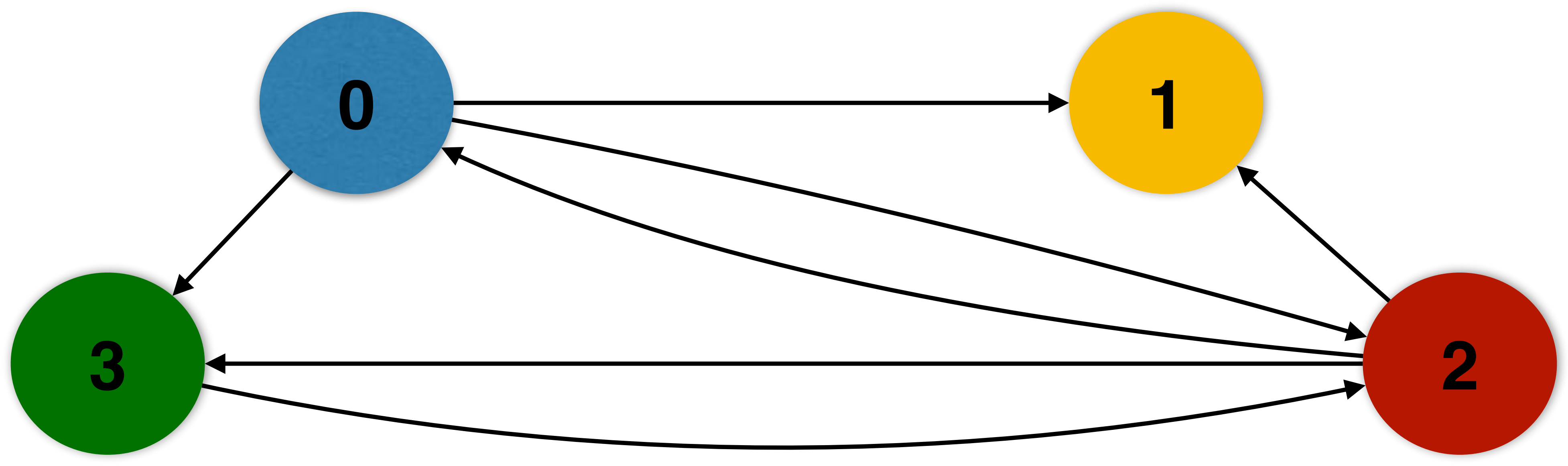


PageRank

```
while …
     for node : graph.vertices
         for ngh : graph.getInNeighbors(node)
             newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
         newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Compressed Sparse Row (CSR)

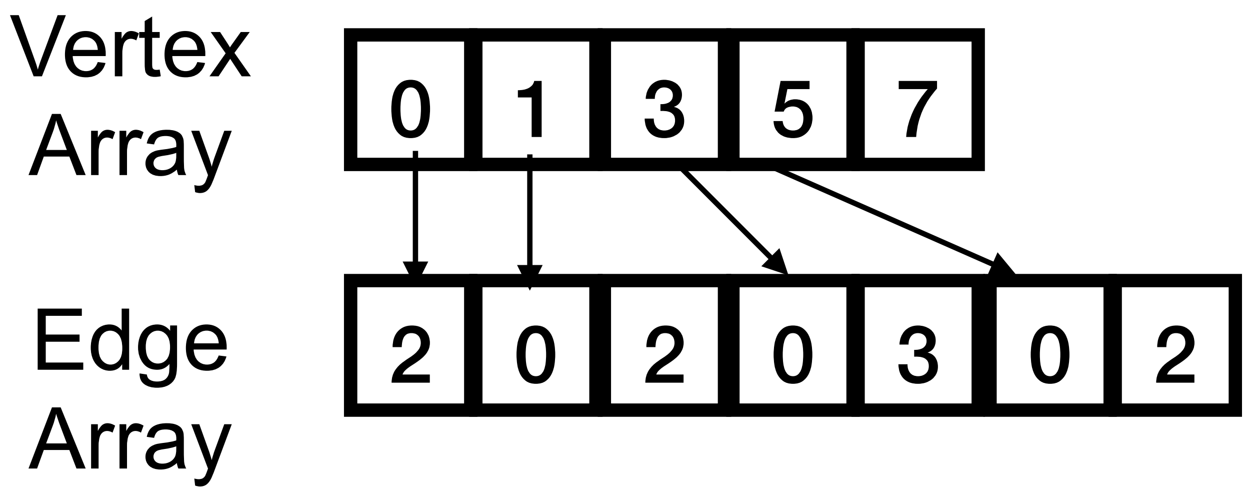
Vertex Array	0	1	3	5	7		
Edge Array	2	0	2	0	3	0	2



PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

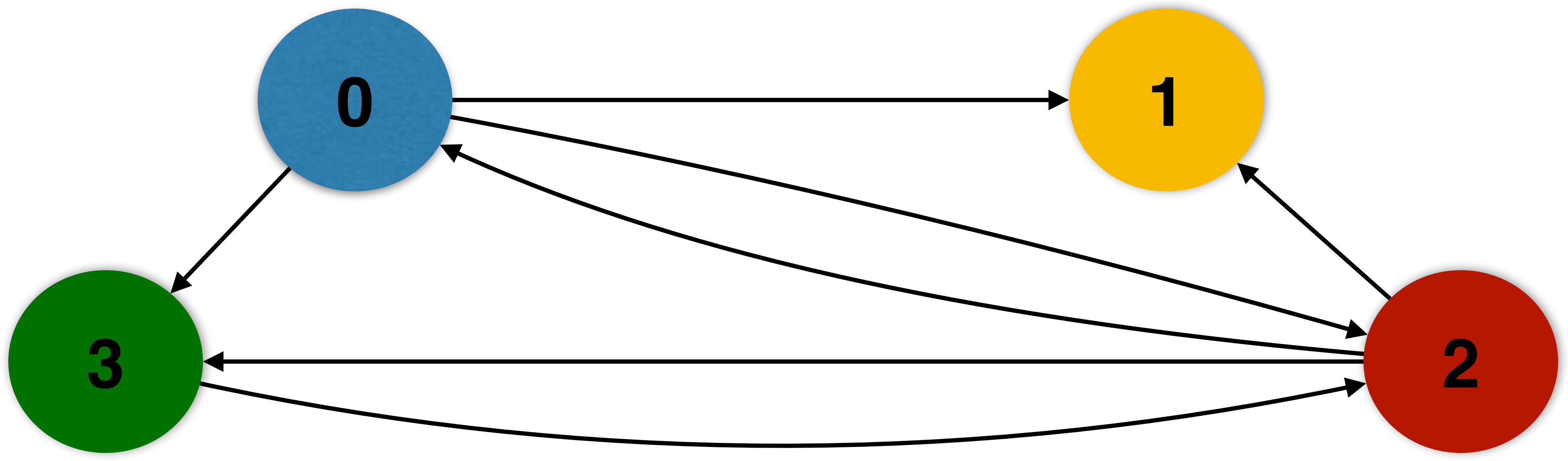
Compressed Sparse Row (CSR)



Vertex Array: 0 1 3 5 7

Edge Array: 2 0 2 0 3 0 2

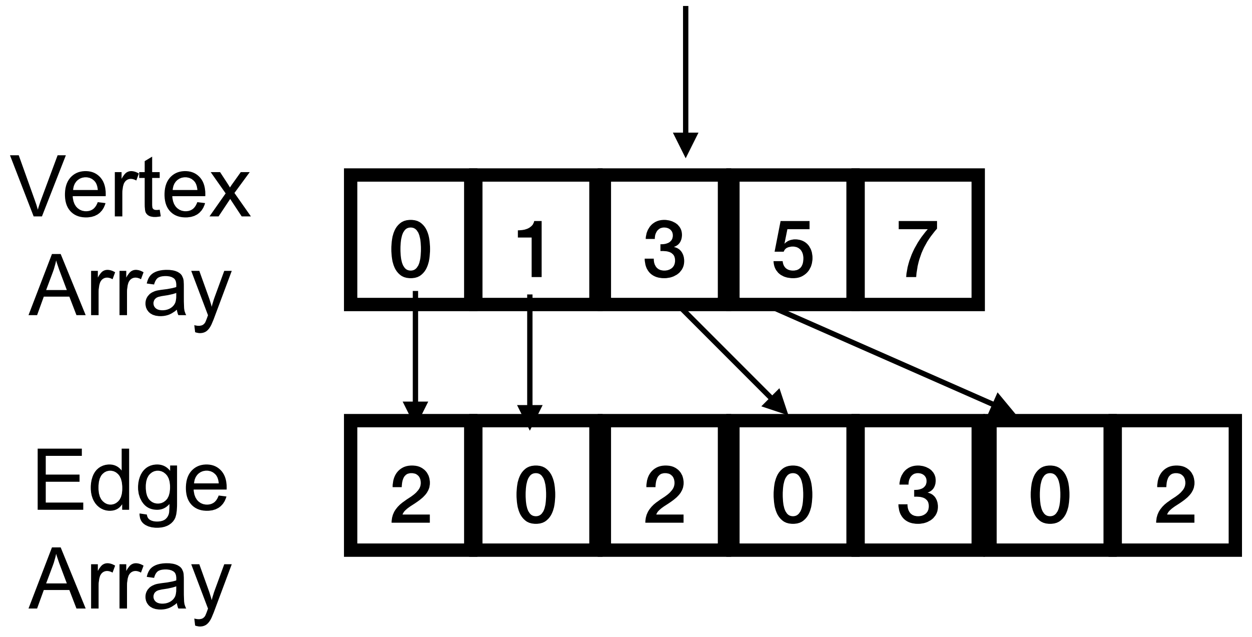
Vertex Array stores indices into the Edge Array. Edge Array stores neighbors' ID in the CSR



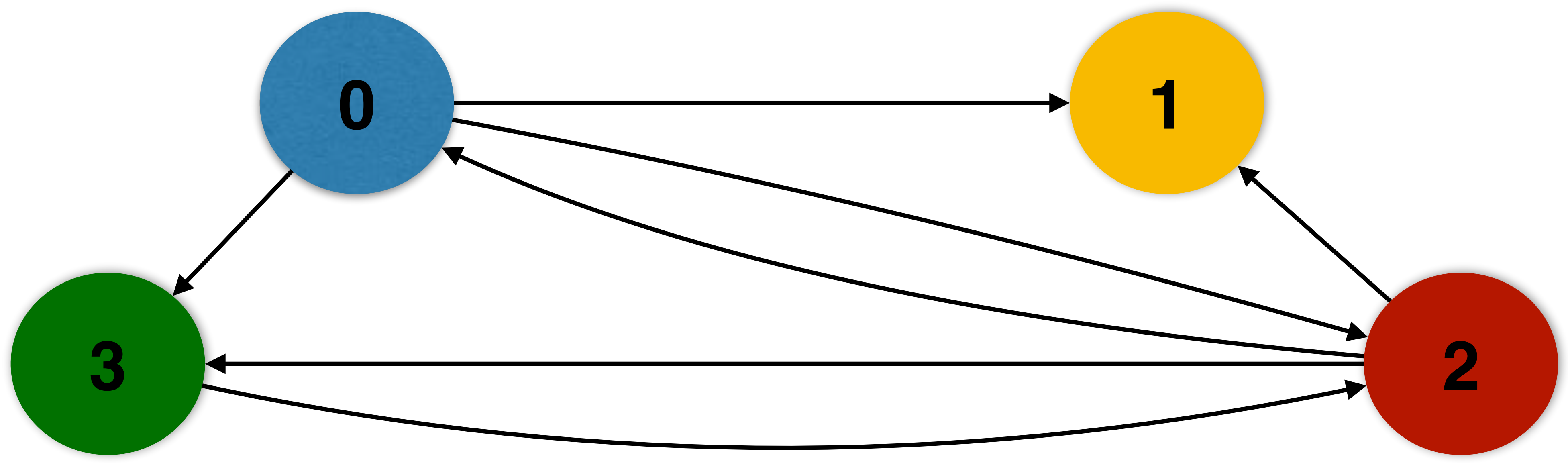
PageRank

```
while …
     for node : graph.vertices
         for ngh : graph.getInNeighbors(node)
             newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Sequential access on node when scanning through vertex array

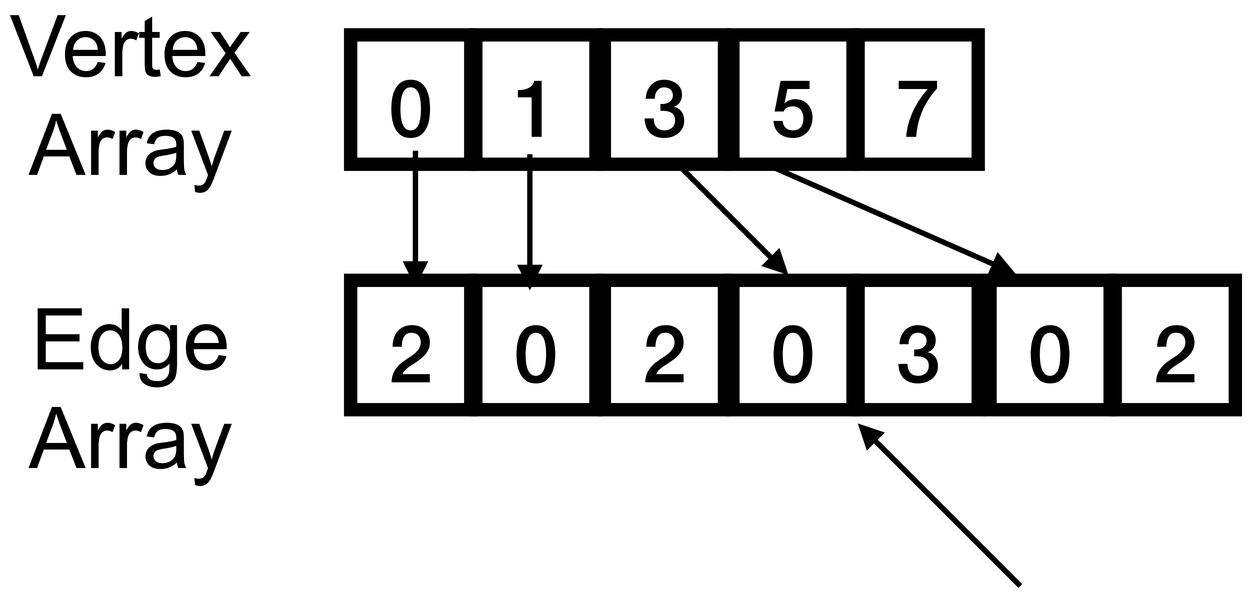


Vertex Array	0	1	3	5	7		
Edge Array	2	0	2	0	3	0	2

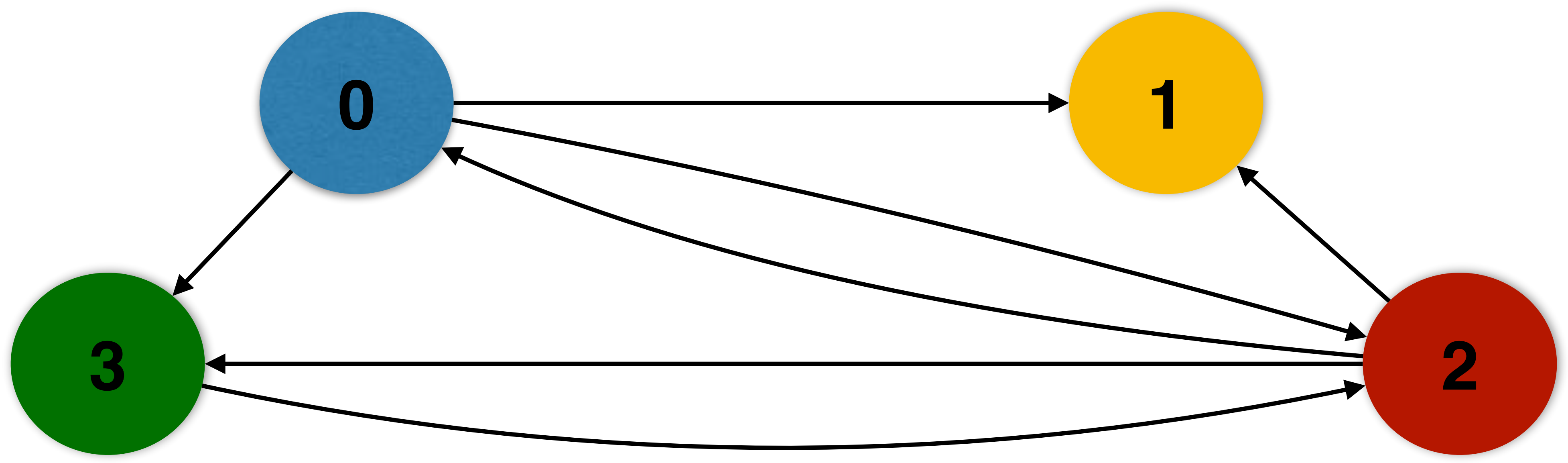


PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



Irregular access on ngh's rank and outDegree data when scanning through the edge array



PageRank

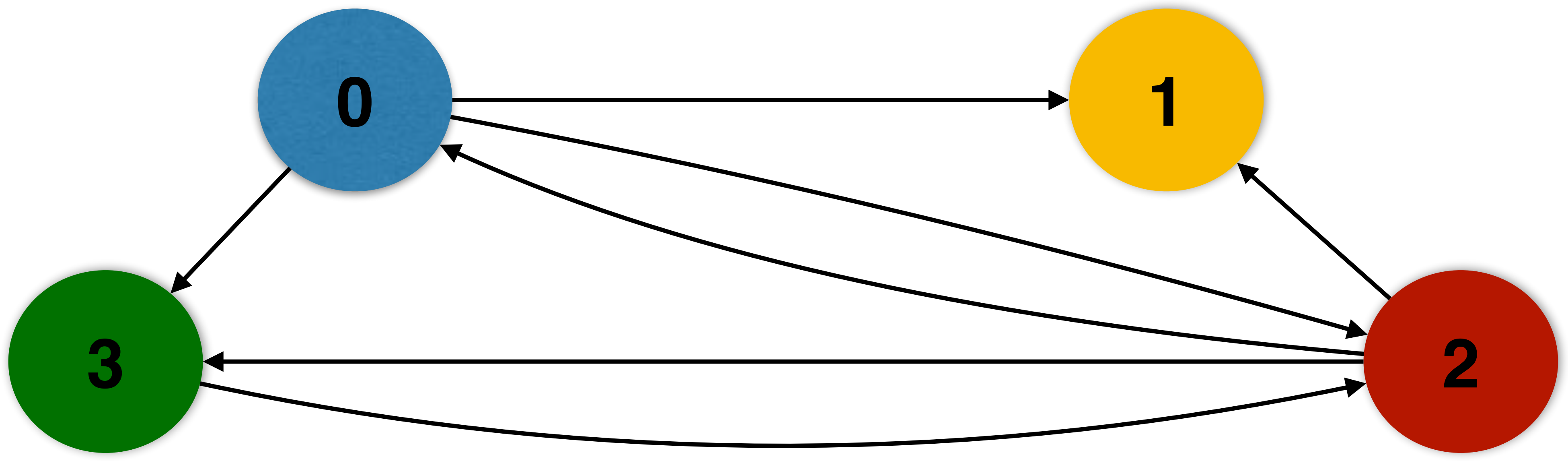
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache

#hits: 0

#misses: 0

0	1	2	3



PageRank

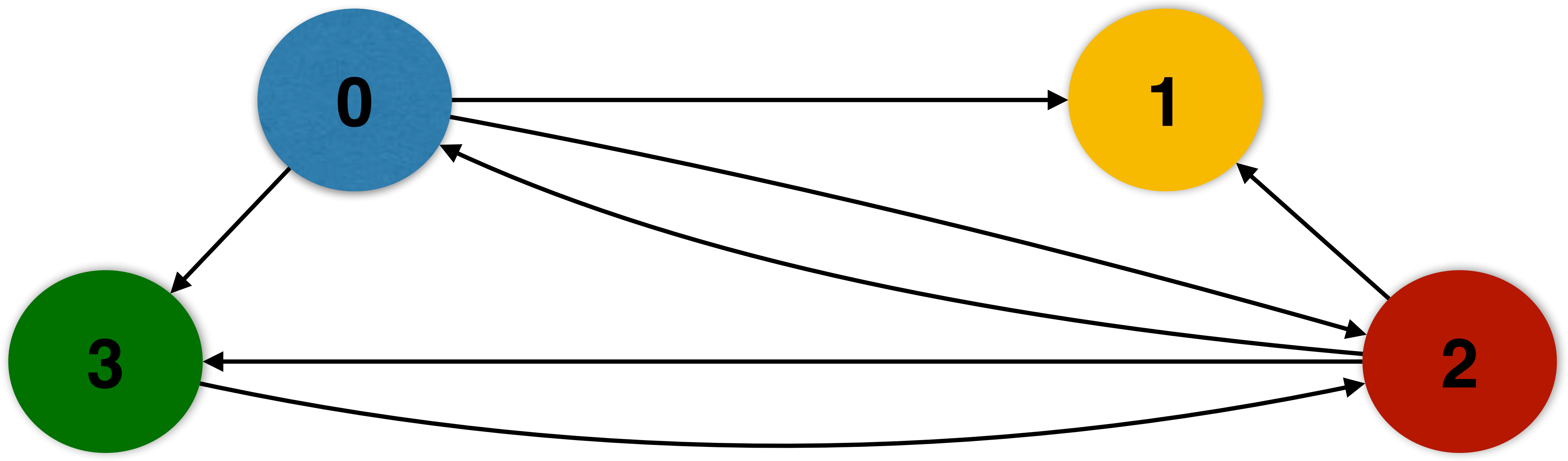
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Focus on the random memory accesses on ranks array

Cache

#hits: 0
#misses: 0

0	1	2	3



PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

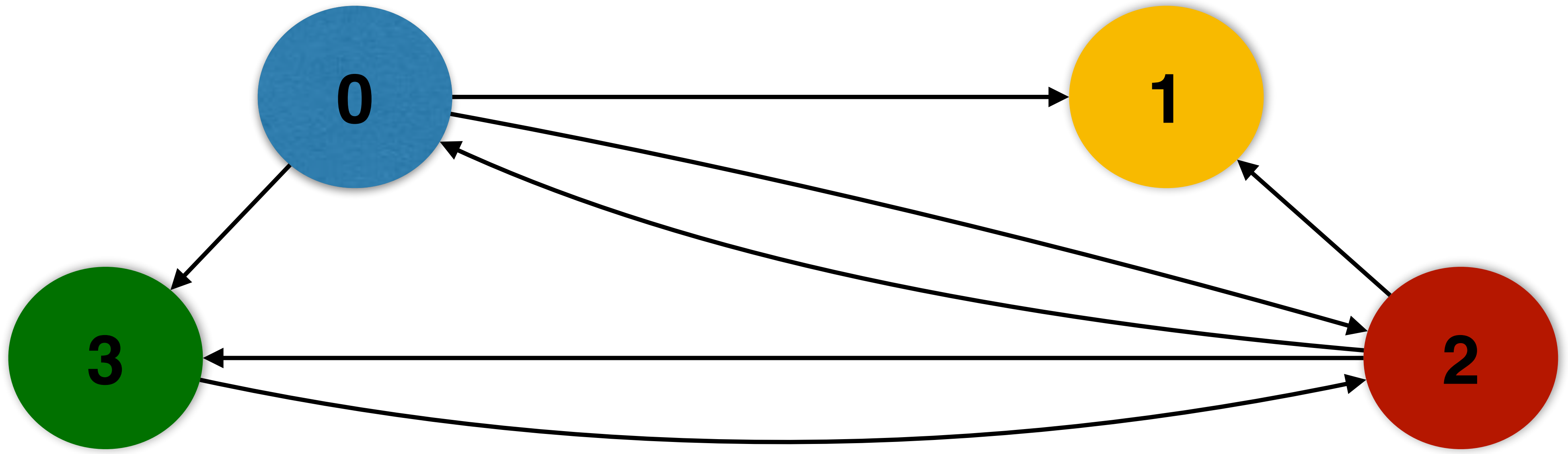
Focus on the random memory accesses on ranks array

Cache ← **holds one cache line**

#hits: 0

#misses: 0

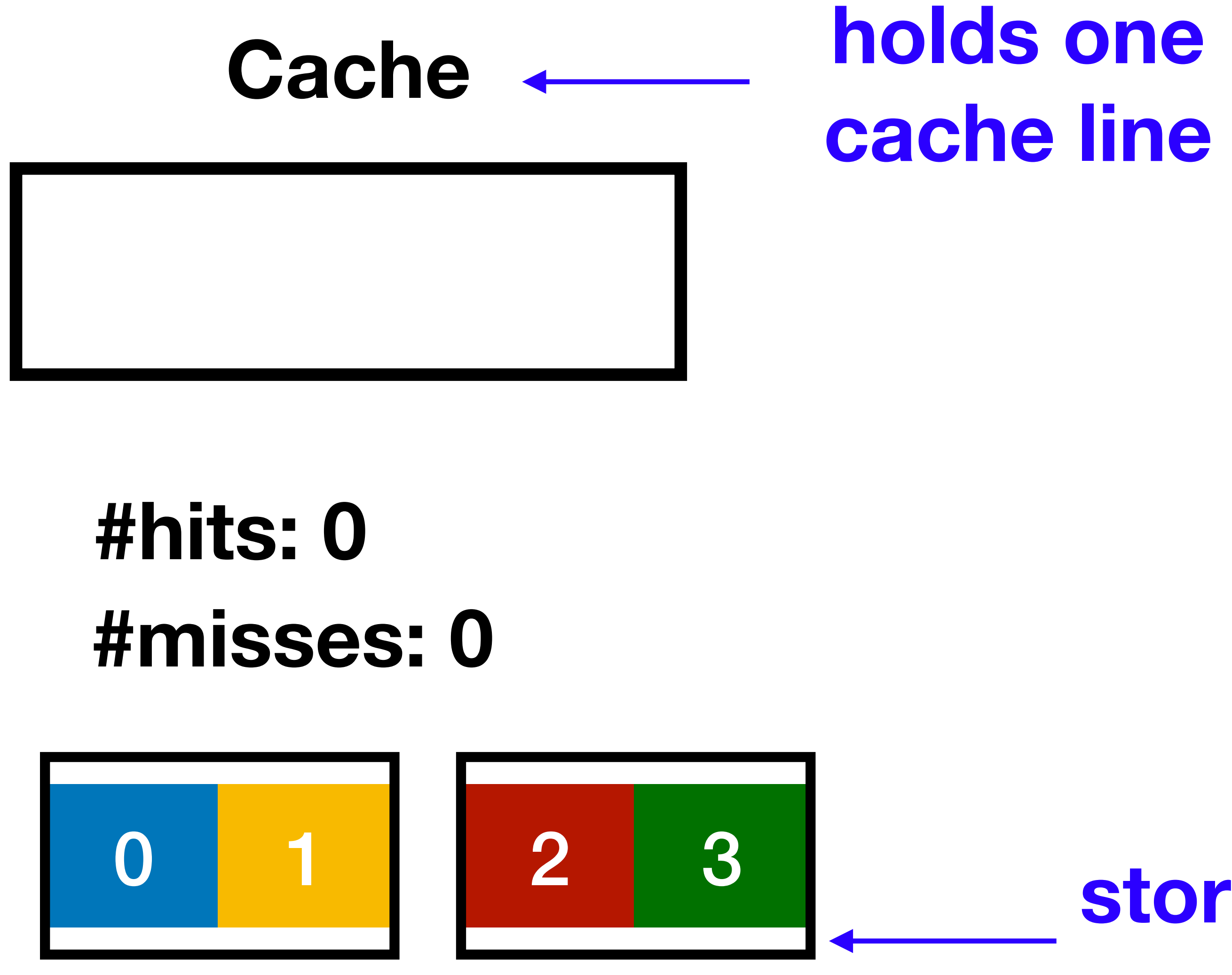
0	1	2	3



PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Focus on the random memory accesses on ranks array

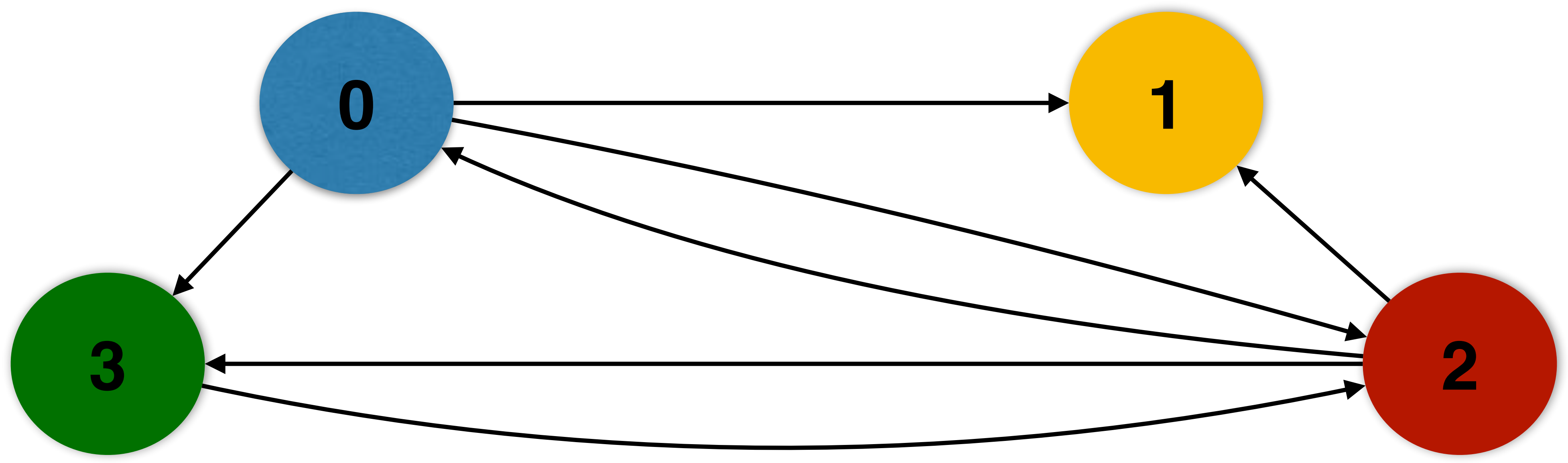


Cache

holds one cache line

#hits: 0
#misses: 0

stored in two cache lines



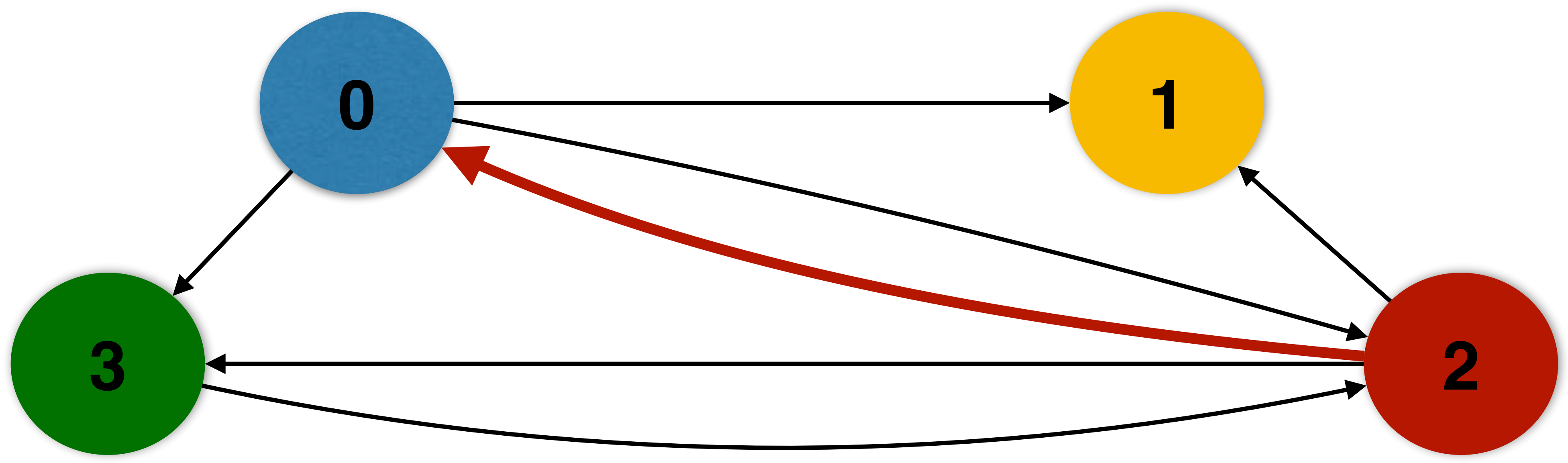
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache

#hits: 0
#misses: 0

0	1	2	3



PageRank

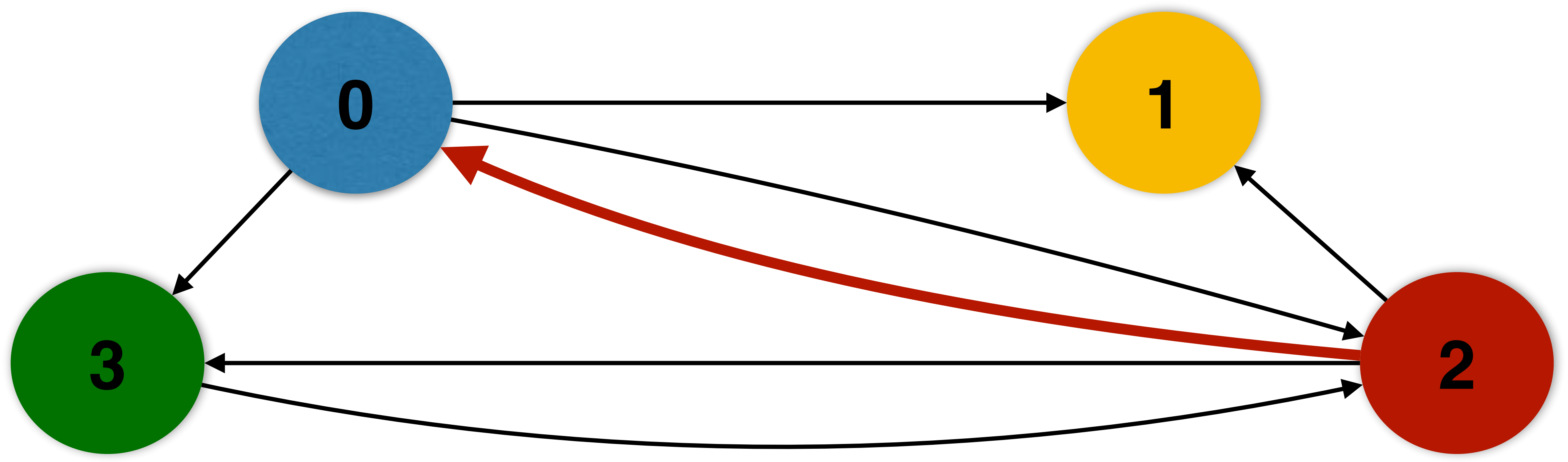
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache

#hits: 0

#misses: 0

0	1	2	3



PageRank

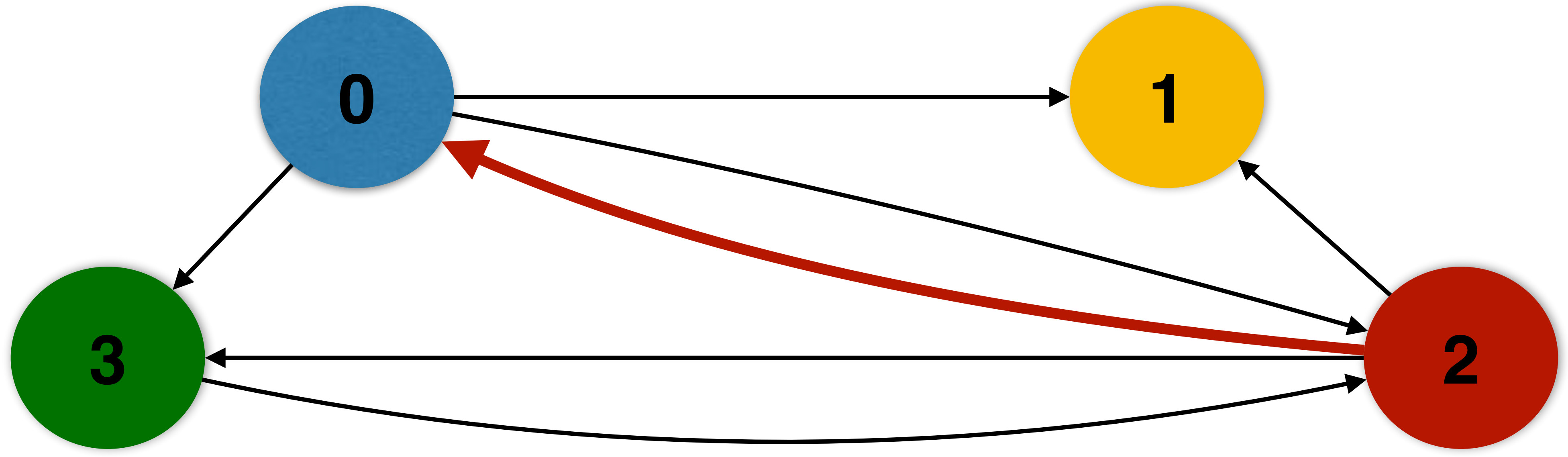
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache

2	3

#hits: 0
#misses: 1

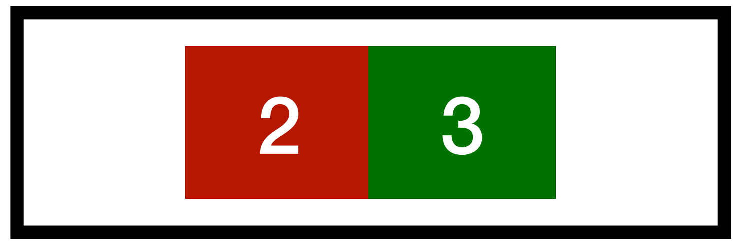
0	1	2	3



PageRank

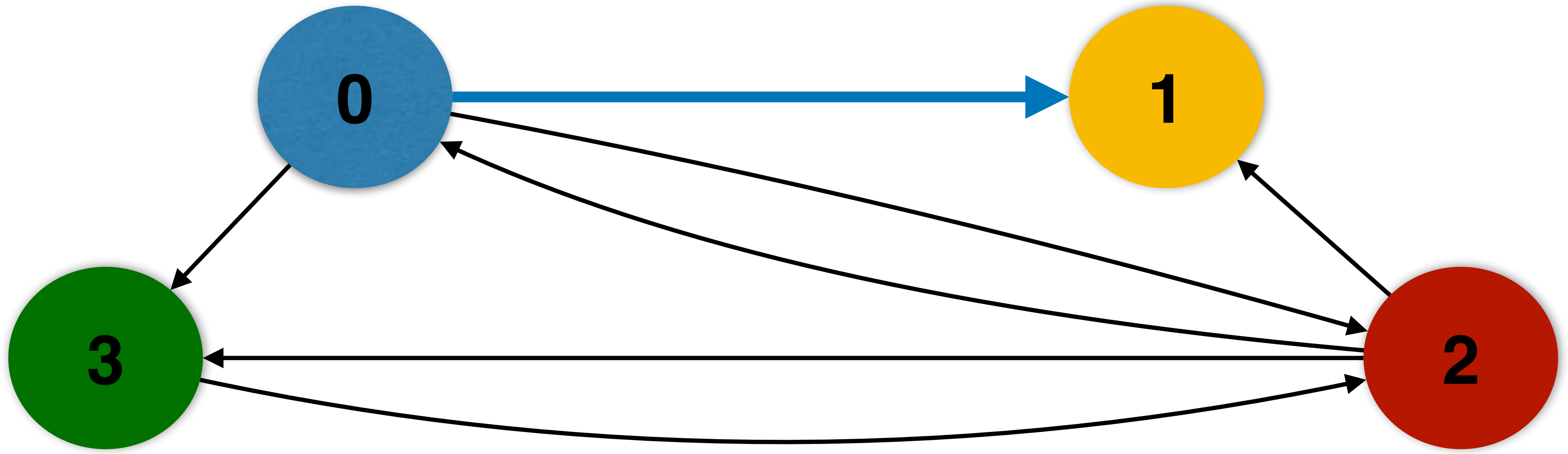
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache



#hits: 0
#misses: 1

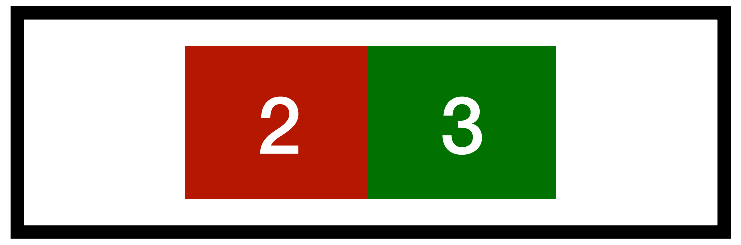
0	1	2	3



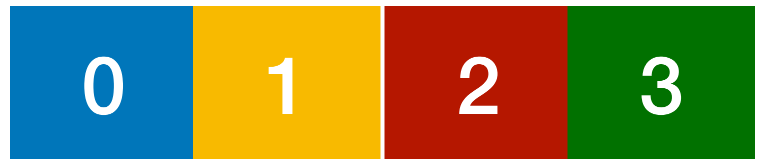
PageRank

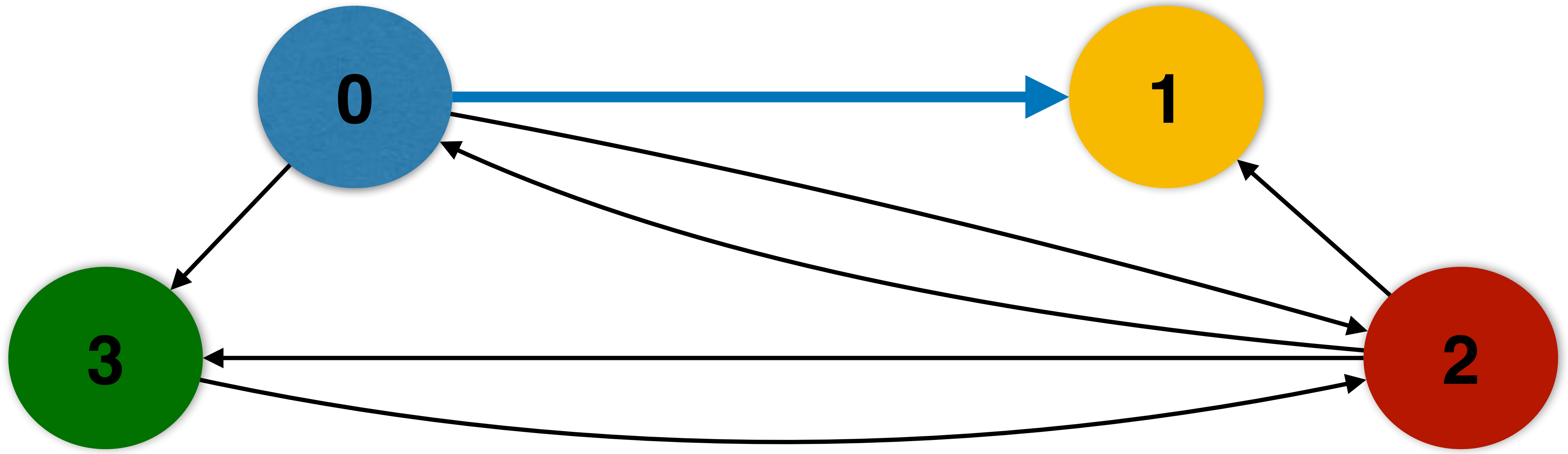
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache



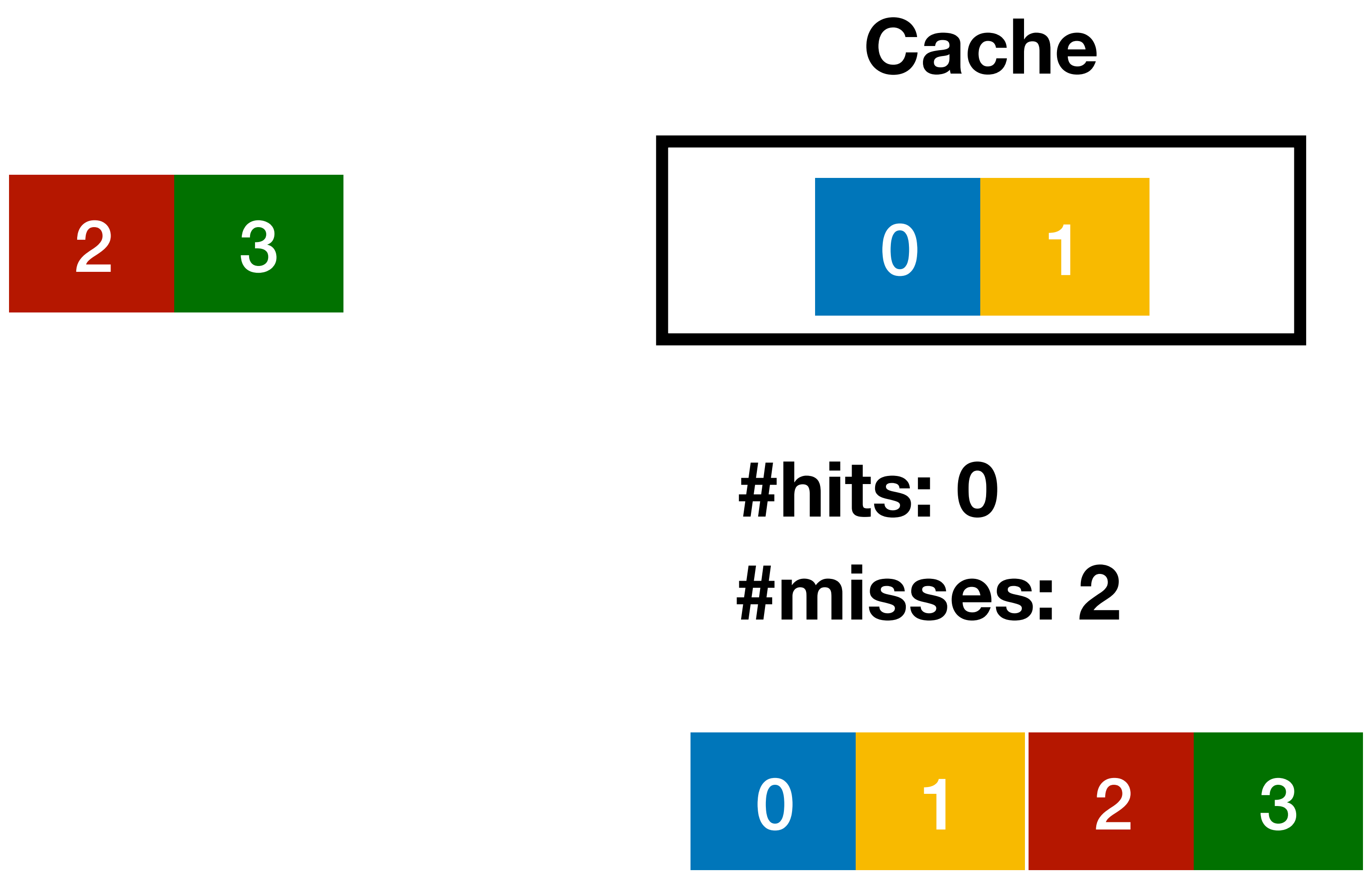
#hits: 0
#misses: 1





PageRank

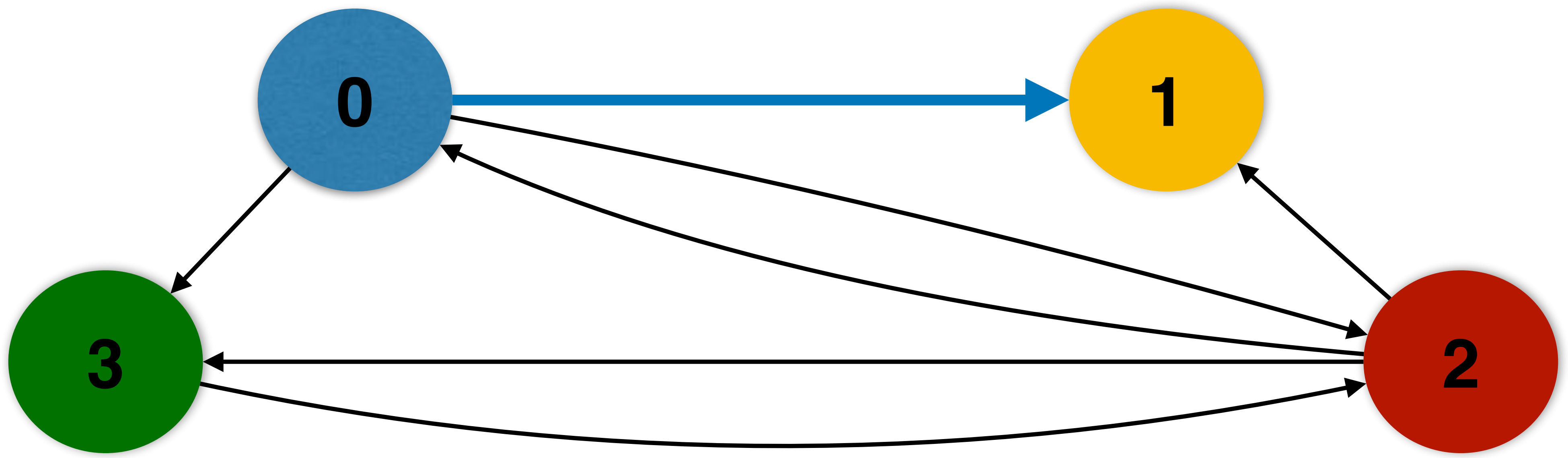
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



Cache

#hits: 0

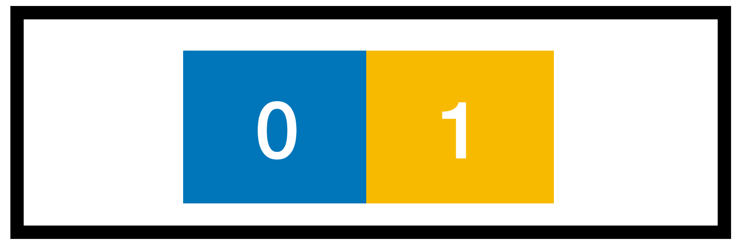
#misses: 2



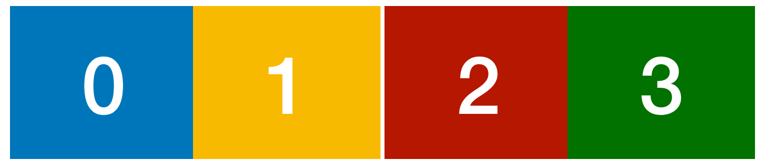
PageRank

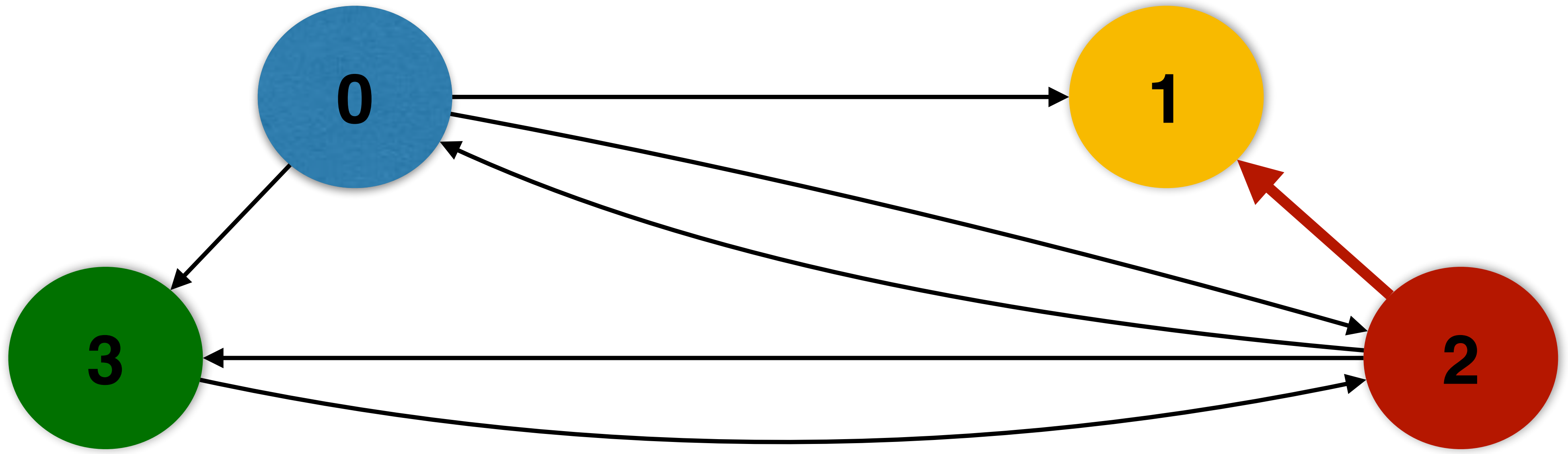
```
while …
     for node : graph.vertices
         for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[ngh]/outDegree[ngh];
     for node : graph.vertices
          newRanks[node] = baseScore + damping*newRanks[node];
     swap ranks and newRanks
```

Cache



#hits: 0
#misses: 2

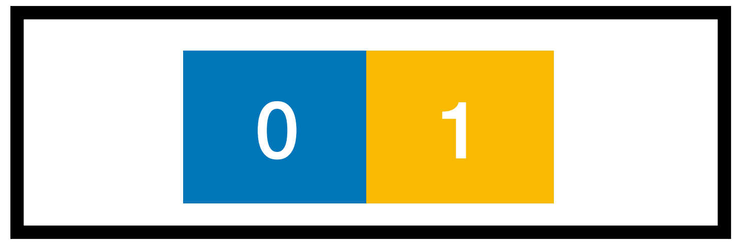




PageRank

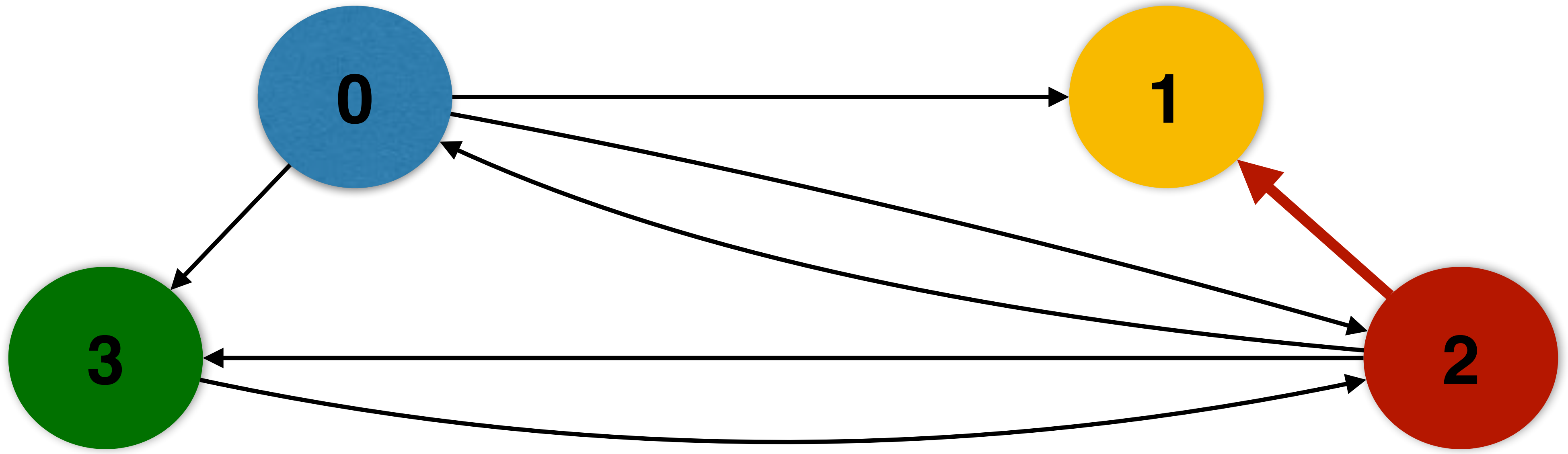
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache



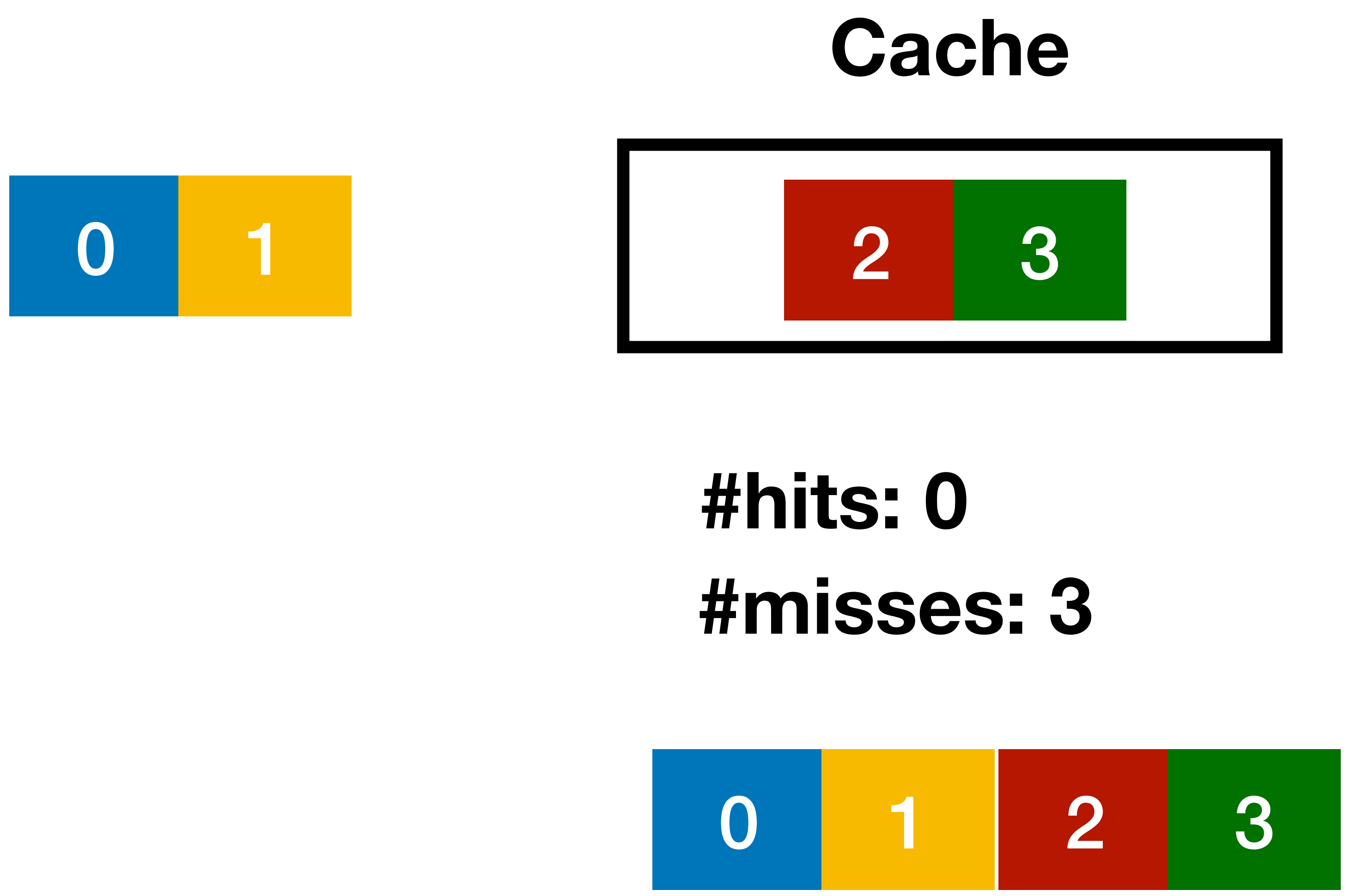
#hits: 0
#misses: 2

0	1	2	3



PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



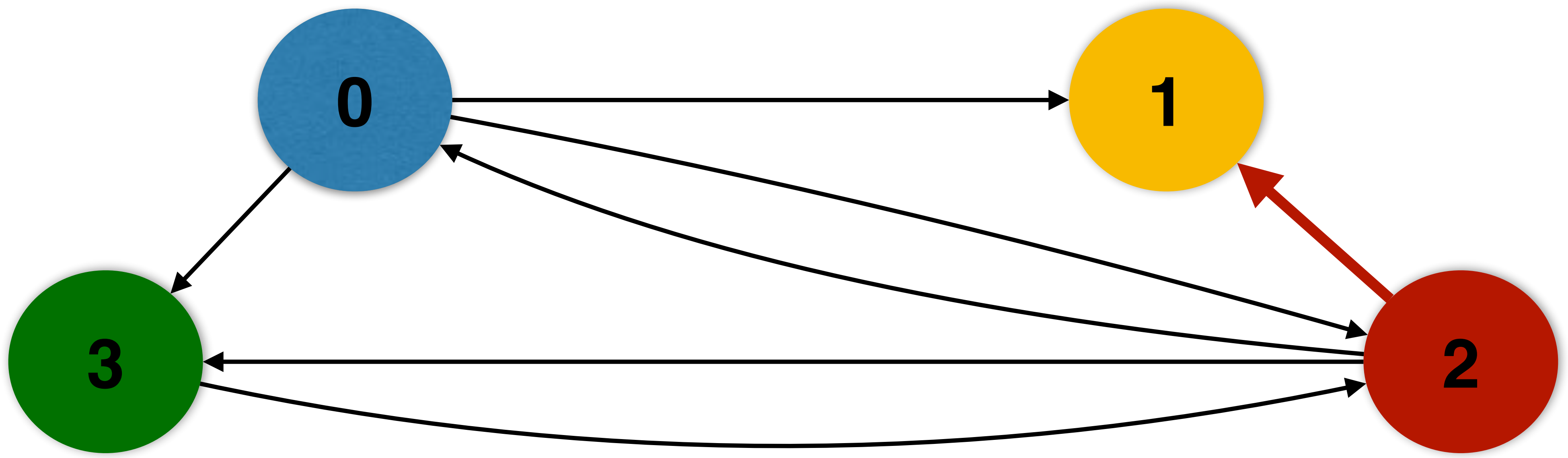
Cache

0	1

2	3

#hits: 0
#misses: 3

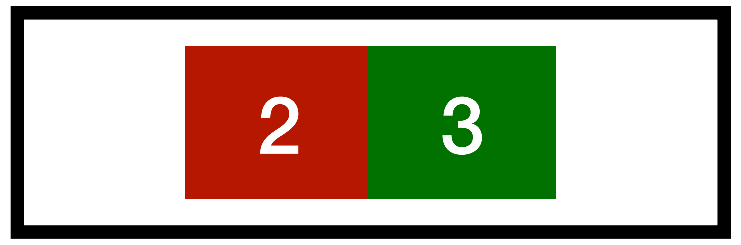
0	1	2	3



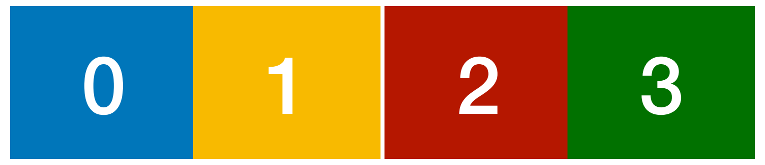
PageRank

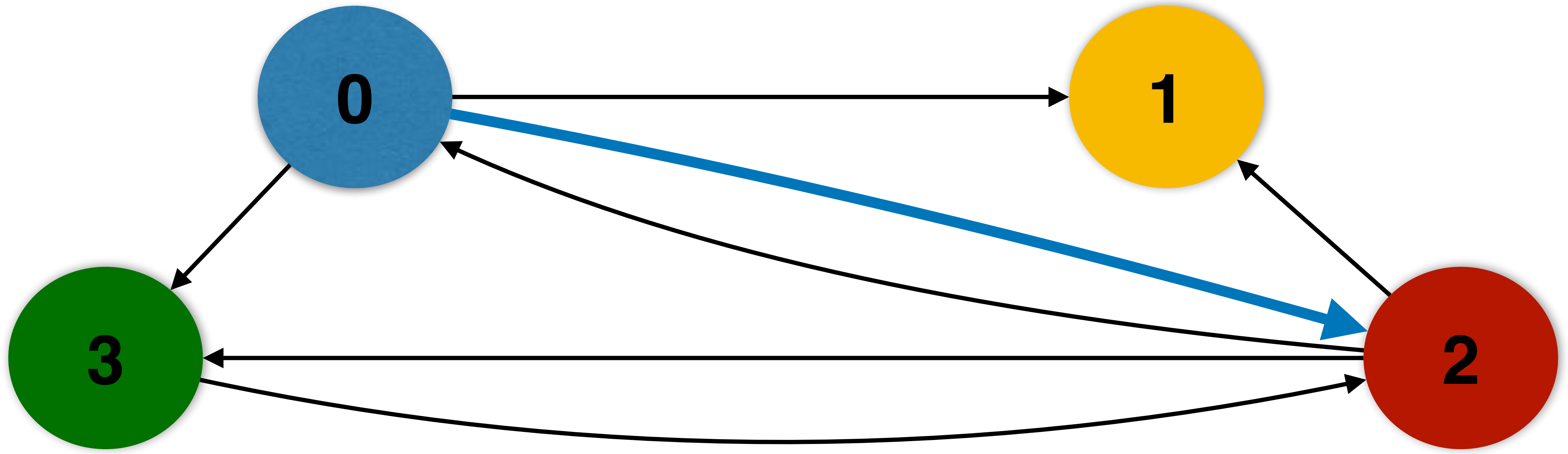
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache



#hits: 0
#misses: 3





PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

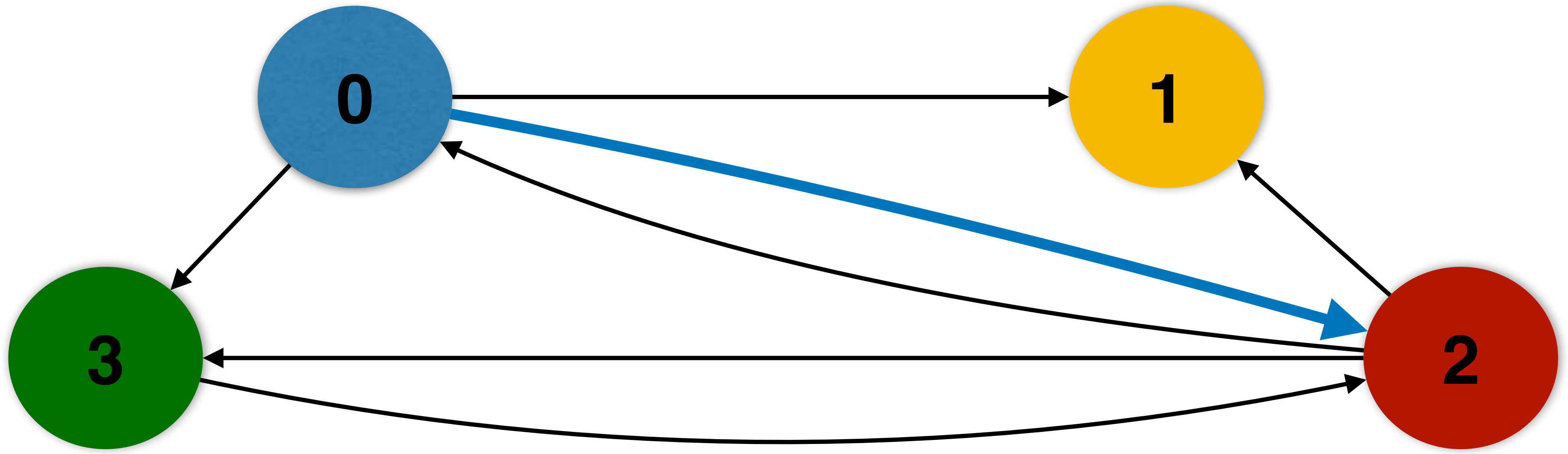
2	3

Cache

0	1

#hits: 0
#misses: 4

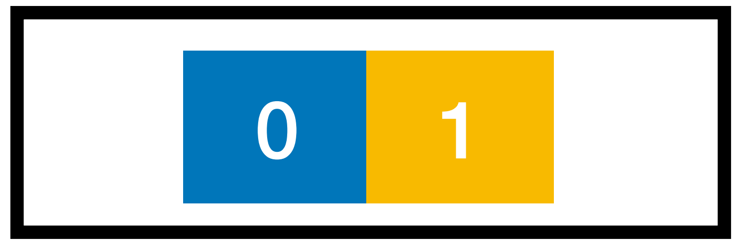
0	1	2	3



PageRank

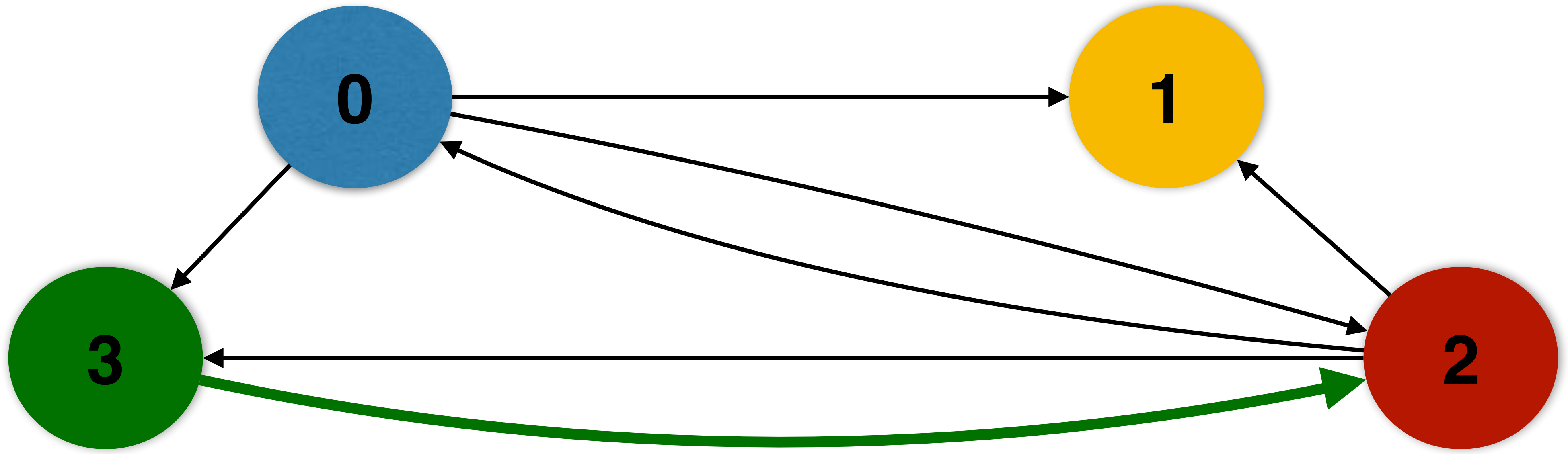
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache



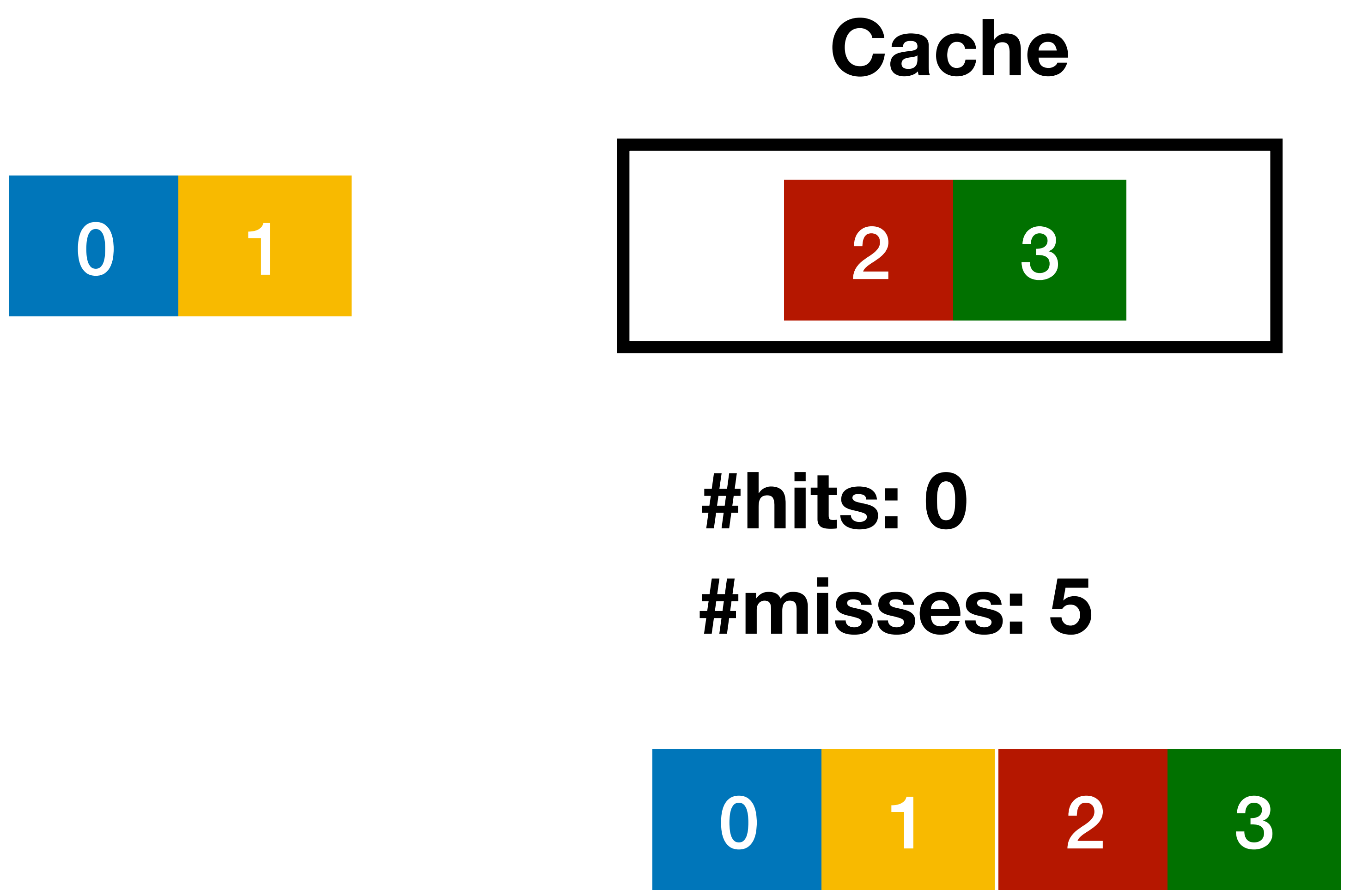
#hits: 0
#misses: 4

0	1	2	3



PageRank

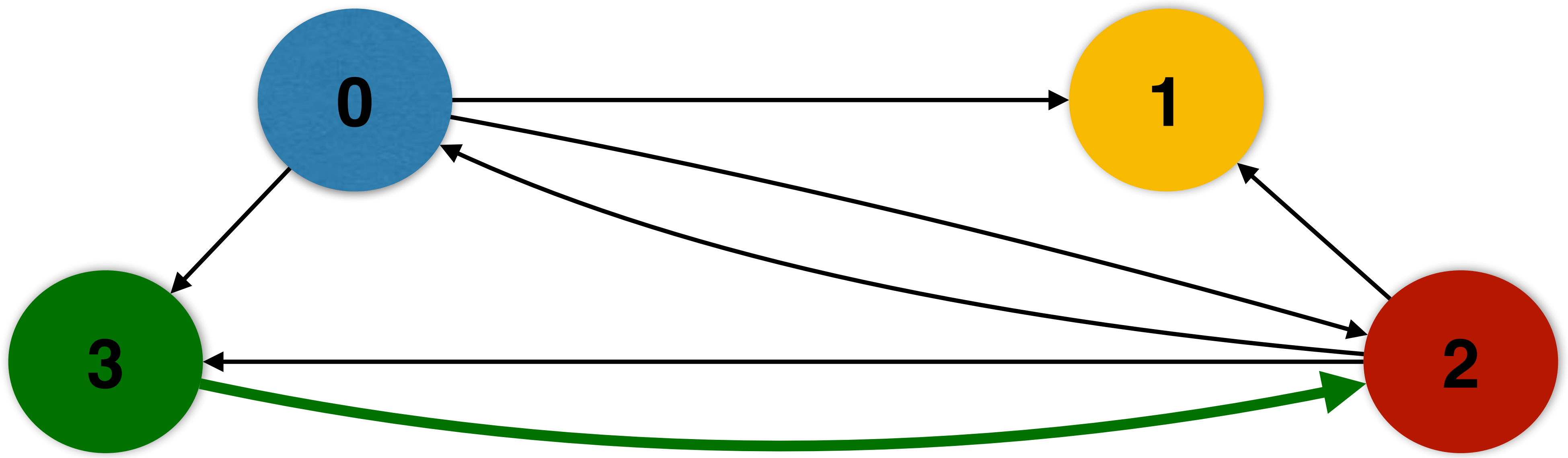
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



Cache

#hits: 0

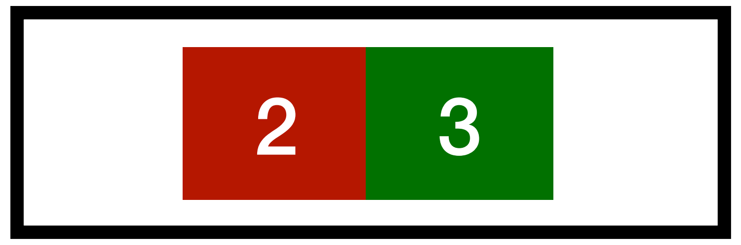
#misses: 5



PageRank

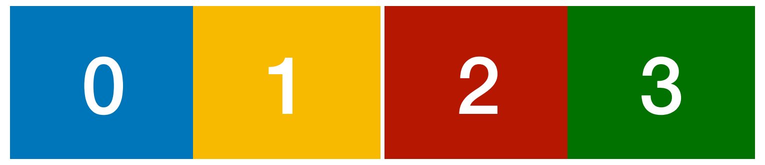
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

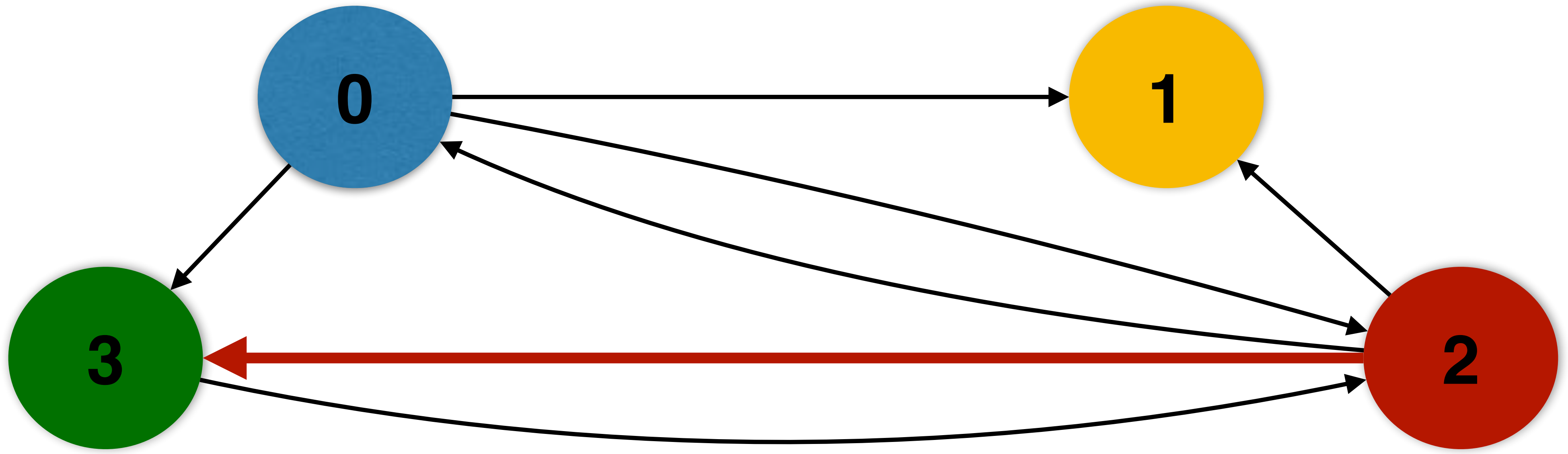
Cache



#hits: 0

#misses: 5





PageRank

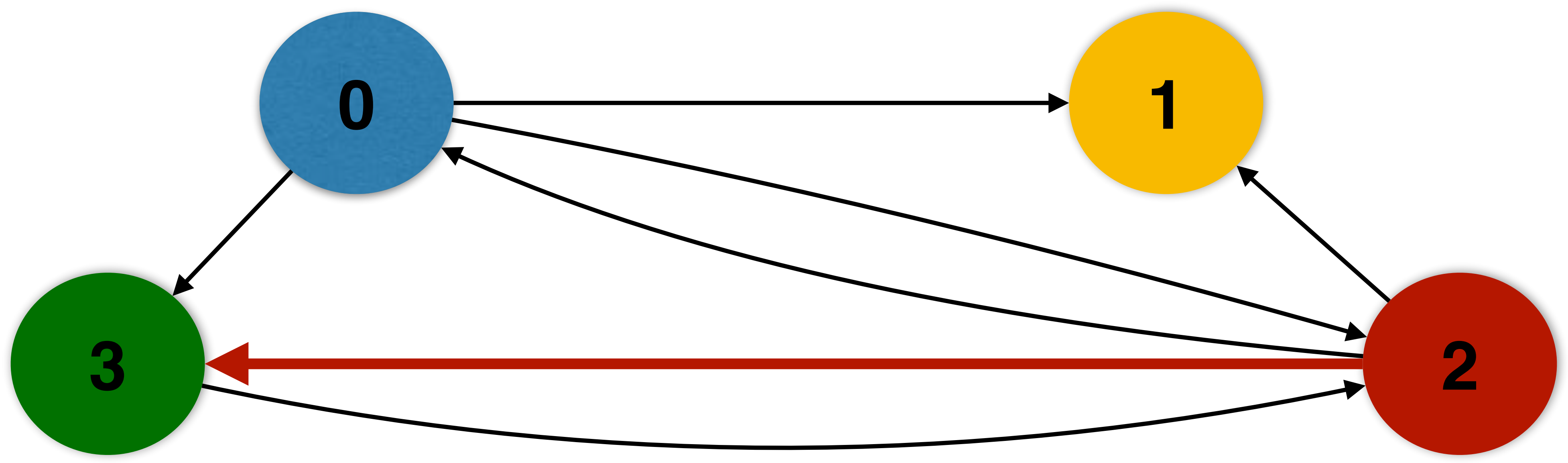
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache

2	3

#hits: 1
#misses: 5

0	1	2	3



PageRank

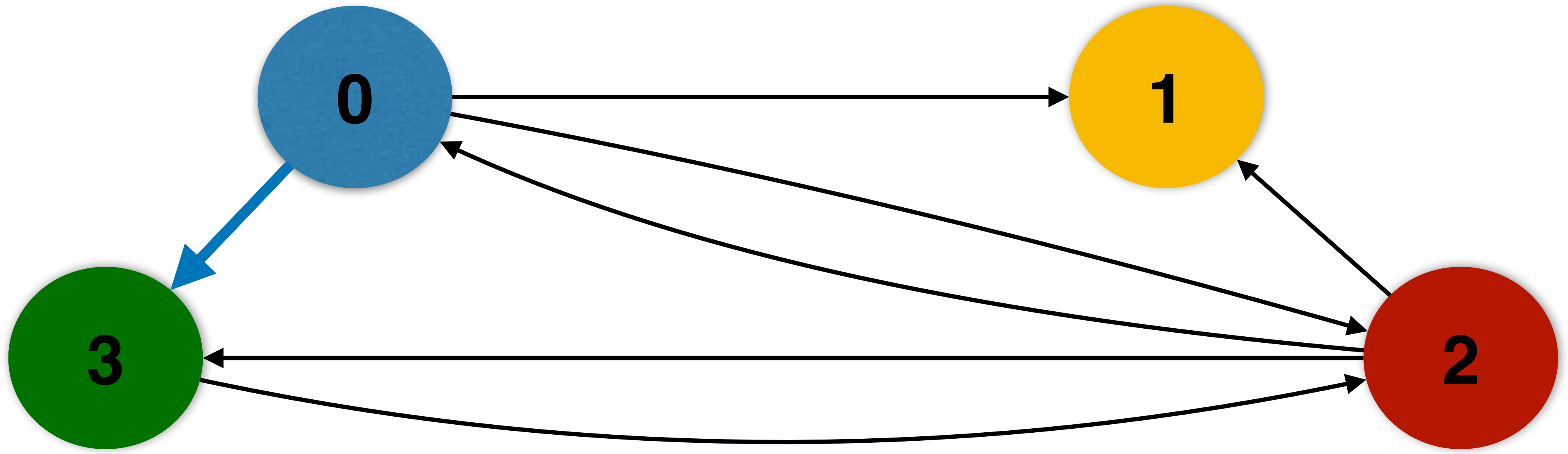
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache

2	3

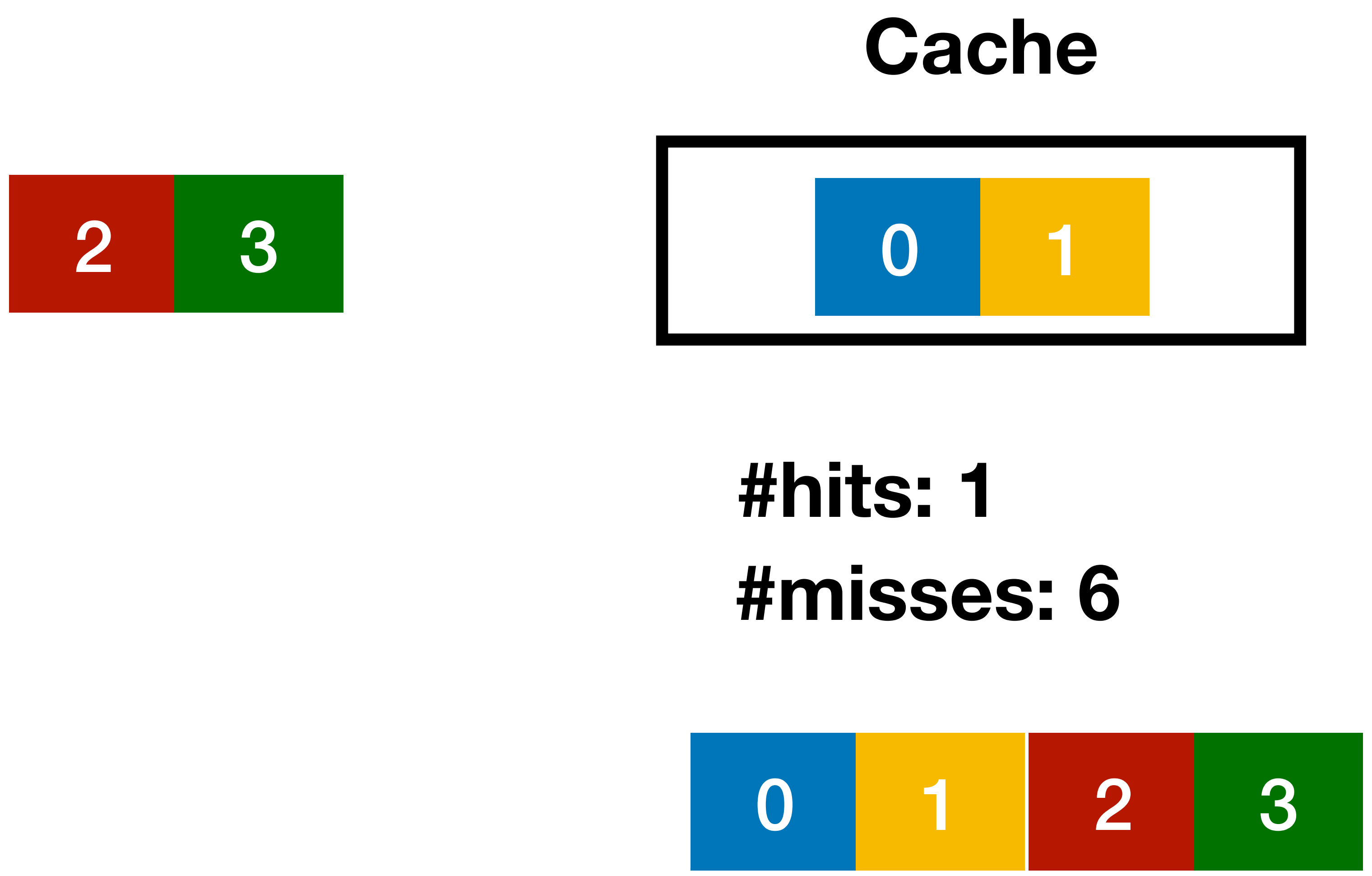
#hits: 1
#misses: 5

0	1	2	3

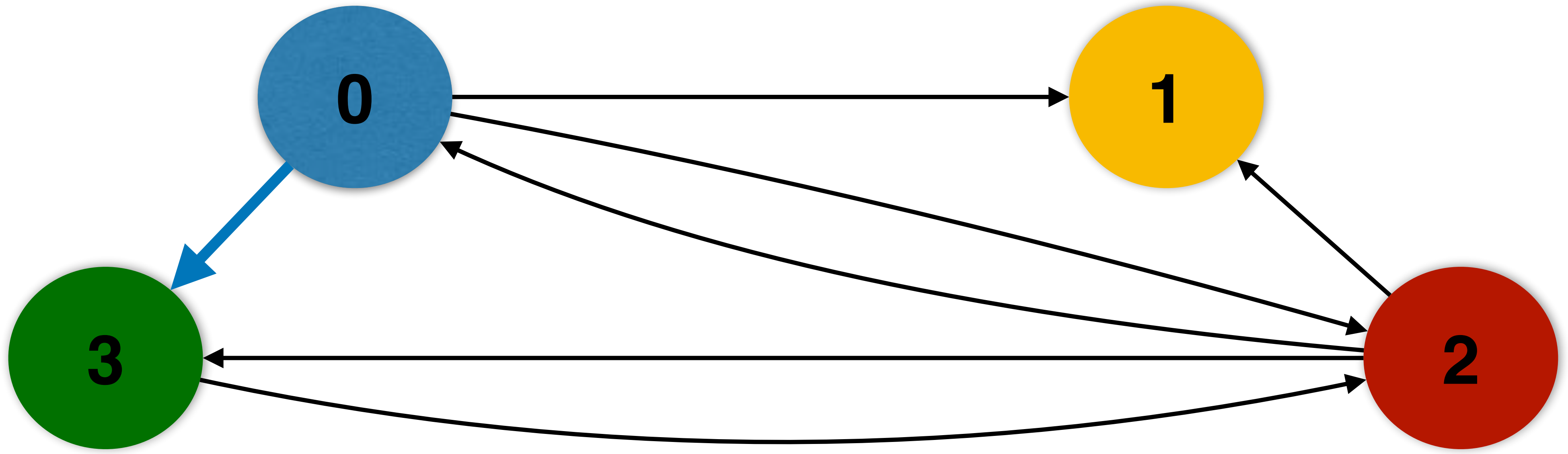


PageRank

```
while …
   for node : graph.vertices
      for ngh : graph.getInNeighbors(node)
         newRanks[node] += ranks[ngh]/outDegree[ngh];
   for node : graph.vertices
      newRanks[node] = baseScore + damping*newRanks[node];
   swap ranks and newRanks
```


Cache

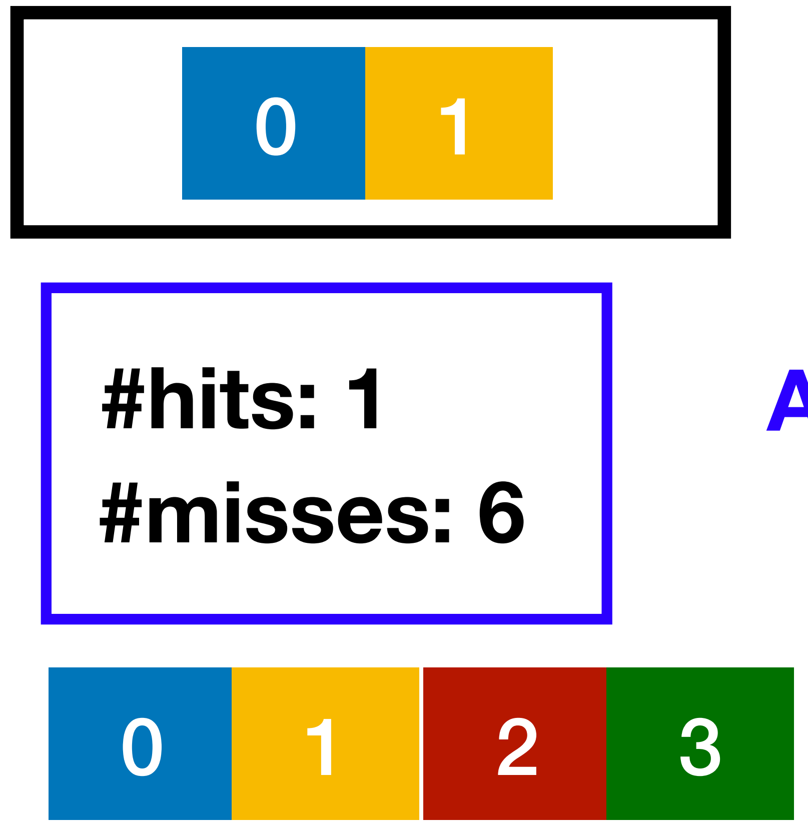
#hits: 1
#misses: 6



PageRank

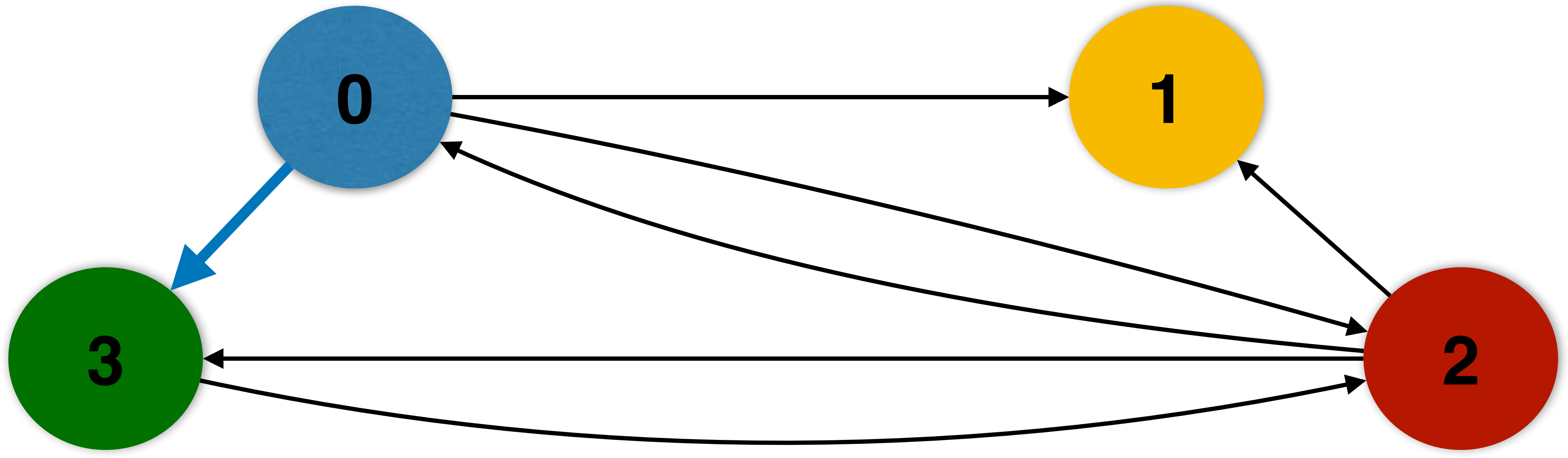
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache



#hits: 1
#misses: 6

A very high miss rate



PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache

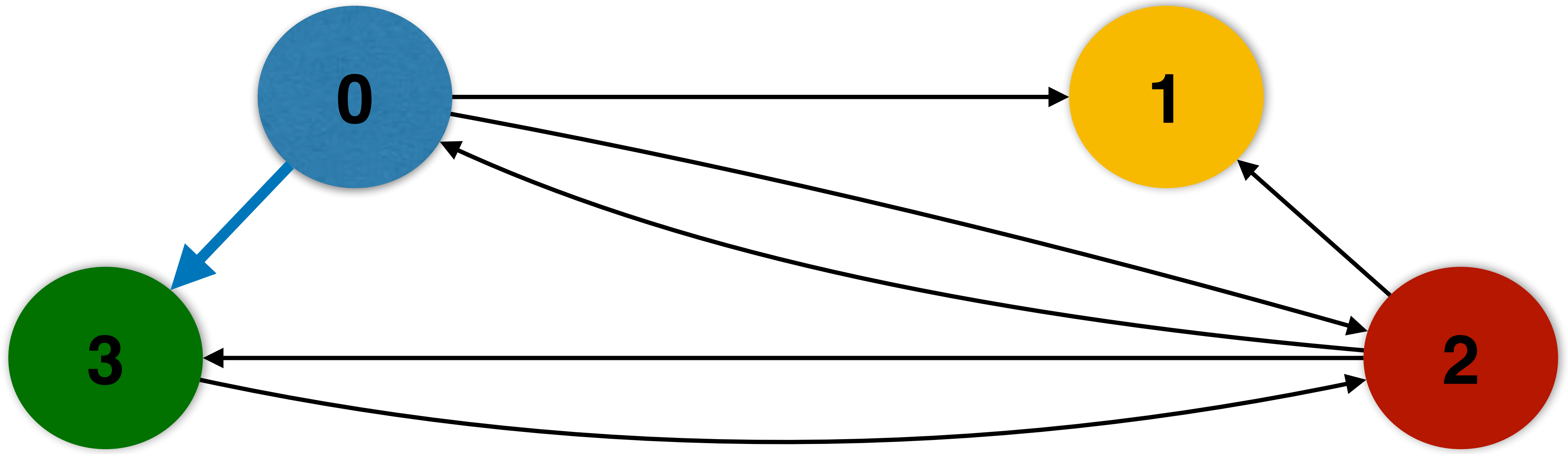
0	1

#hits: 1

#misses: 6

Working set larger than cache

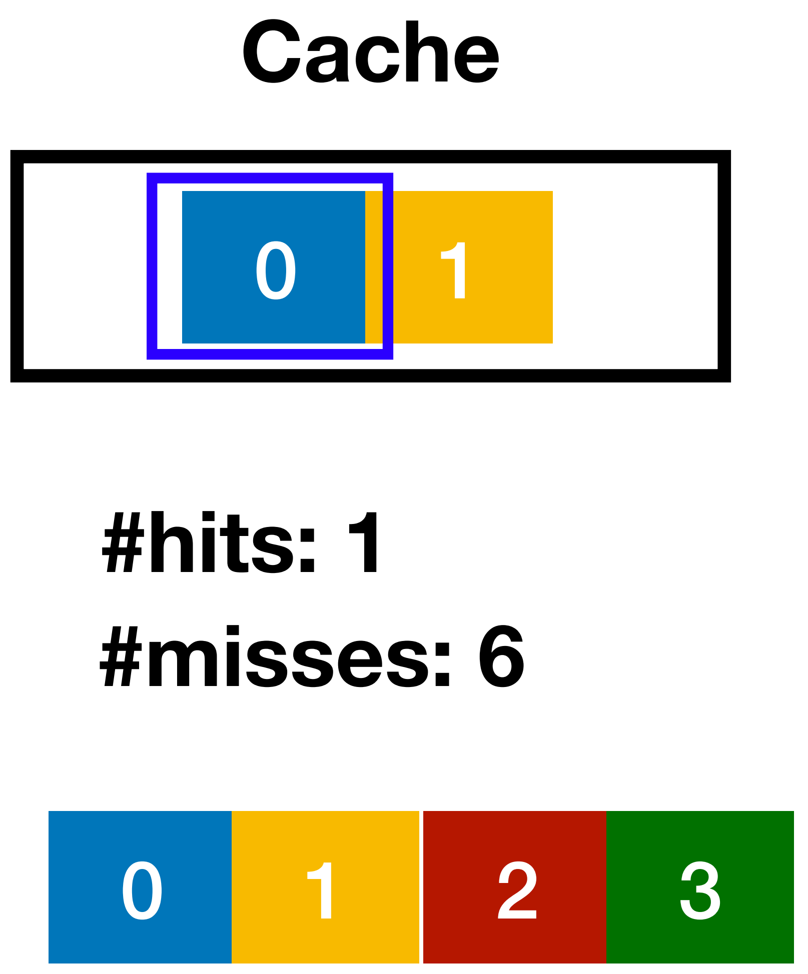
0	1	2	3



PageRank

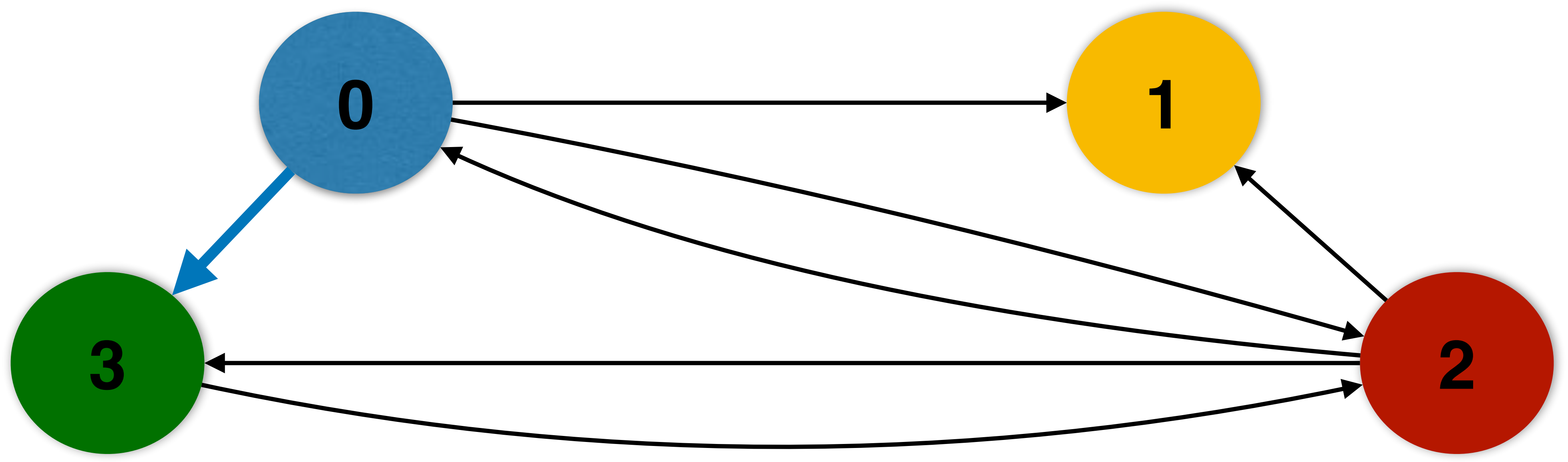
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Often only use part of the cache line



#hits: 1

#misses: 6



Performance Bottleneck

- Working set much larger than cache size
- Access pattern is irregular
  - Often uses part of the cache line
  - Hard to benefit from hardware prefetching
  - TLB miss, DRAM row miss (hundreds of cycles)

Performance Bottleneck

Real-world graphs often have working set 10-200x larger than cache size

- Working set much larger than cache size
- Access pattern is irregular
  - Often uses part of the cache line
  - Hard to benefit from hardware prefetching
  - TLB miss, DRAM row miss (hundreds of cycles)

Performance Bottleneck

- Working set much larger than cache size
- Access pattern is irregular
    - Often uses part of the cache line
    - Hard to benefit from hardware prefetching
    - TLB miss, DRAM row miss (hundreds of cycles)

Performance Bottleneck

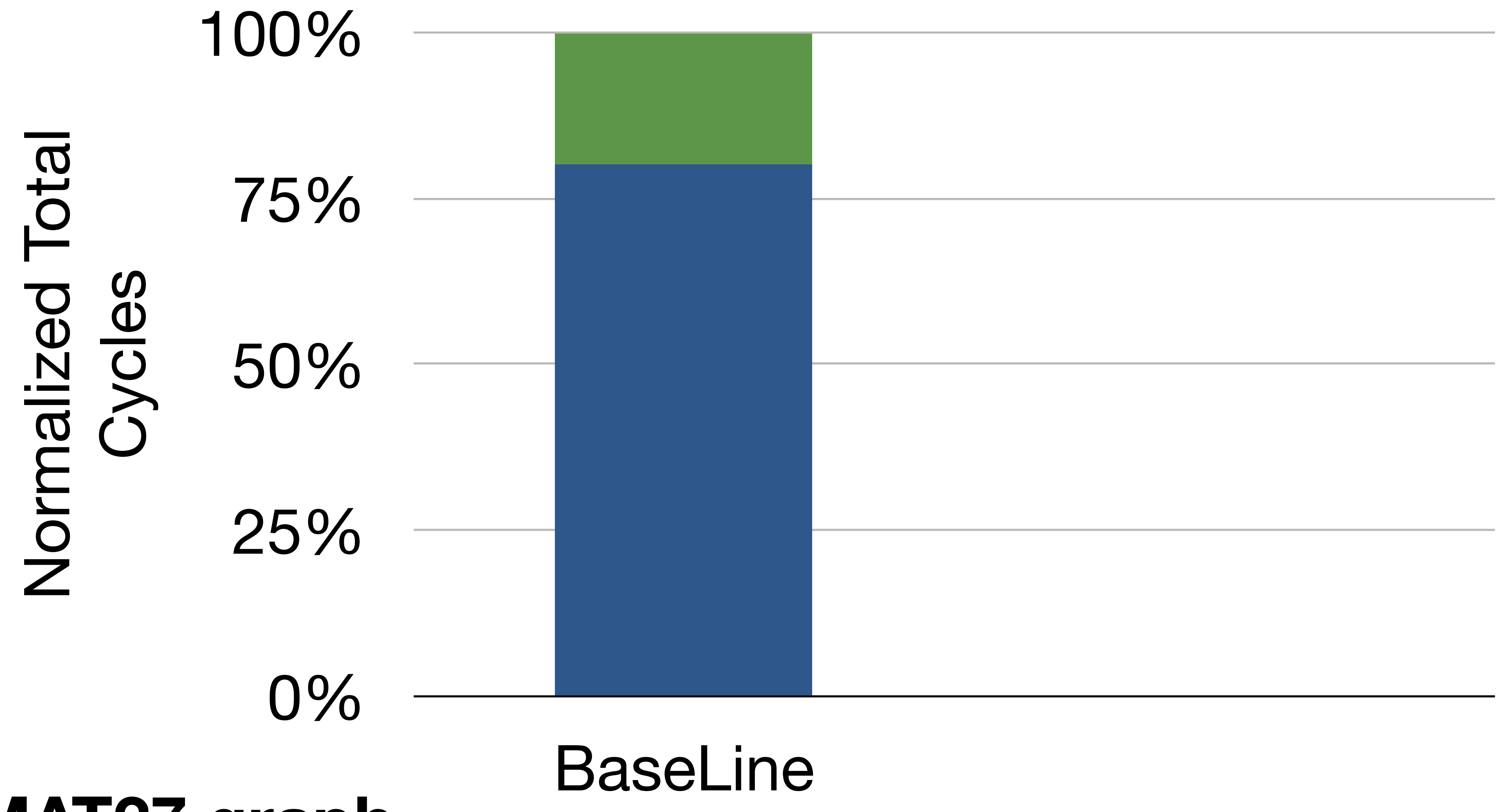
- Working set much larger than cache size
- Access pattern is irregular

Often only use 1/16 - 1/8 of a cache line in modern hardware

  - Often uses part of the cache line
  - Hard to benefit from hardware prefetching
  - TLB miss, DRAM row miss (hundreds of cycles)

PageRank

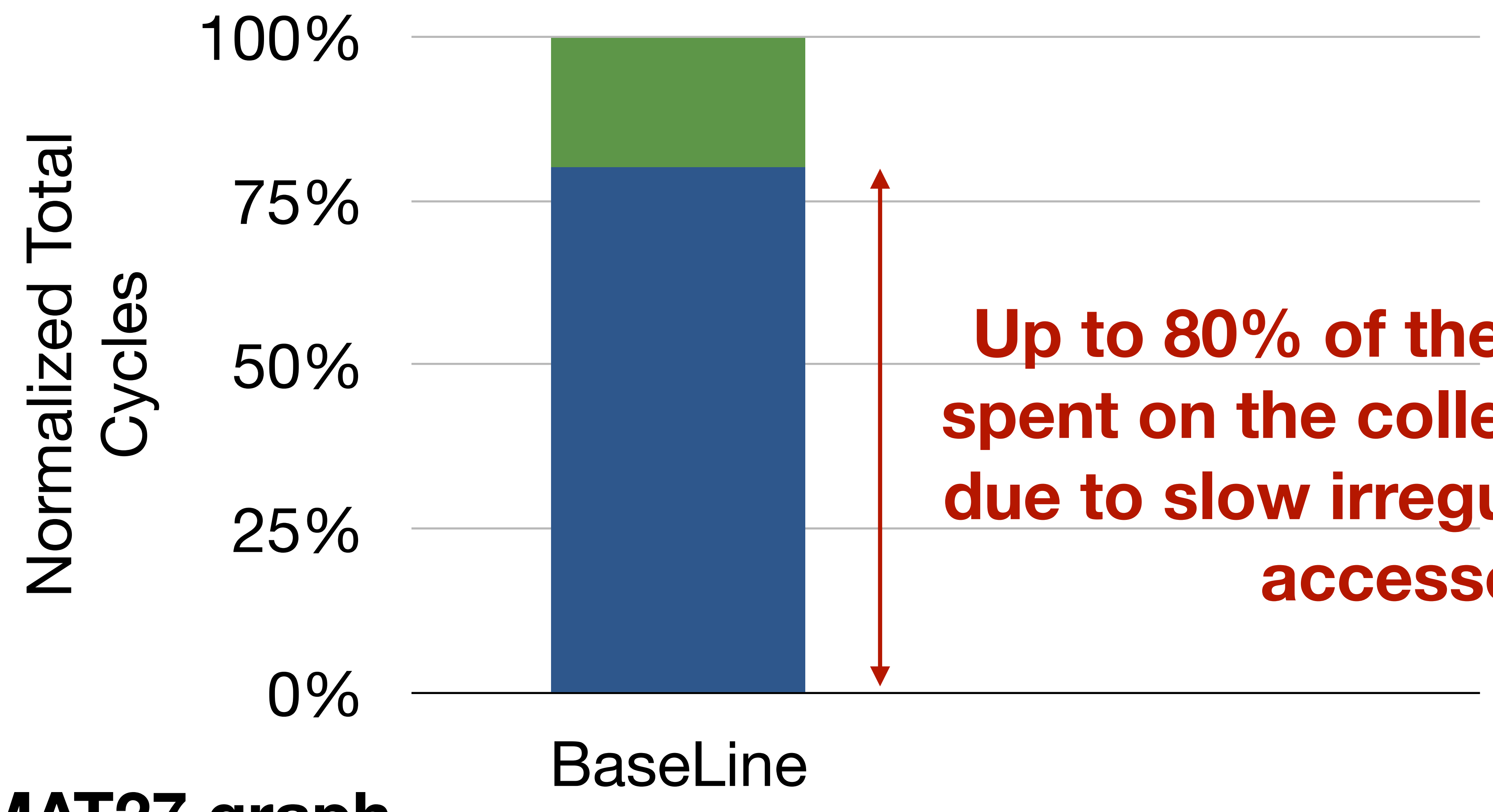
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



on RMAT27 graph

PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



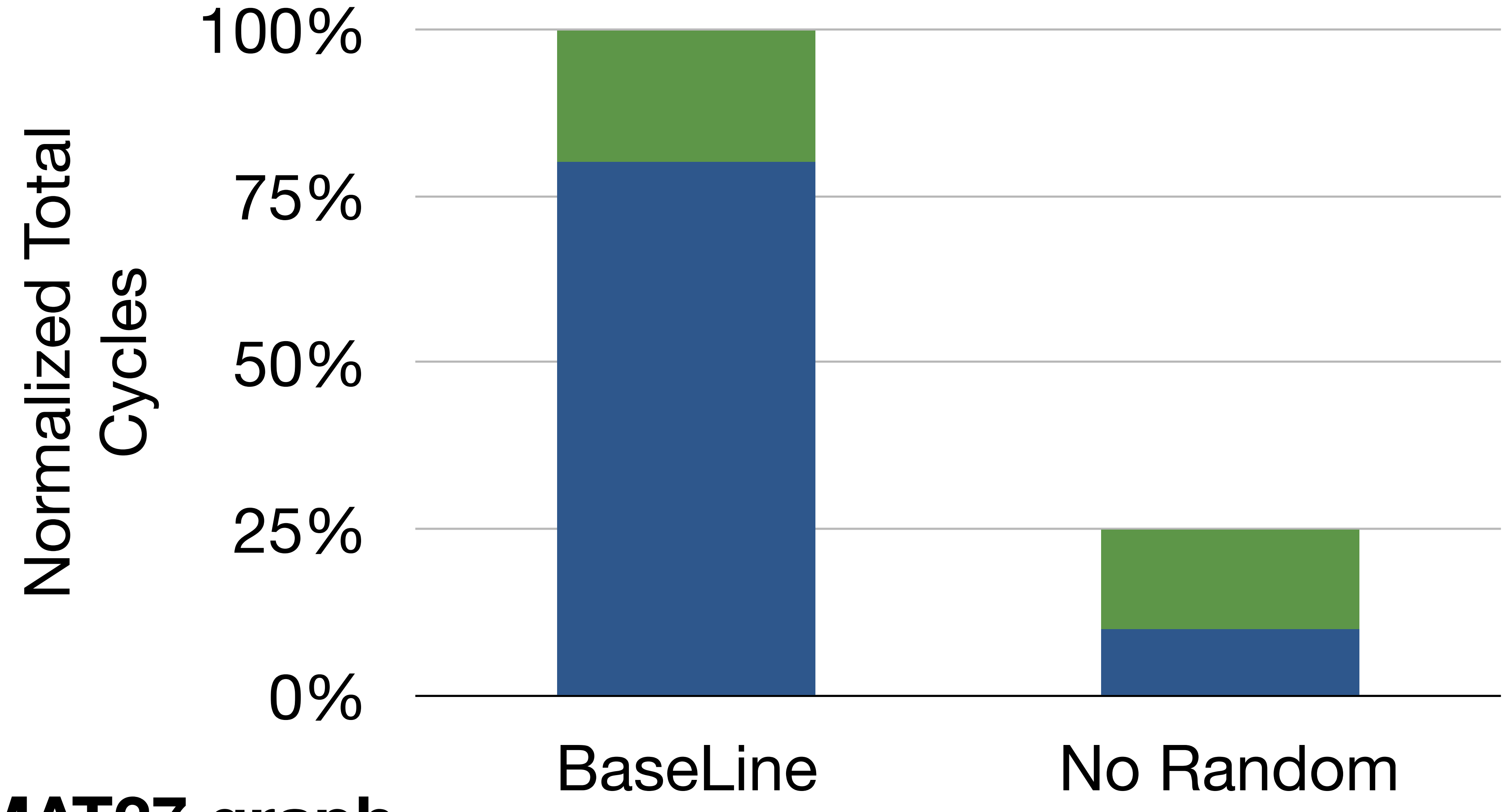
Up to 80% of the cycles are spent on the collection phase due to slow irregular memory accesses

on RMAT27 graph

PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[0]/outDegree[0];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Removing Random Accesses (Incorrect)

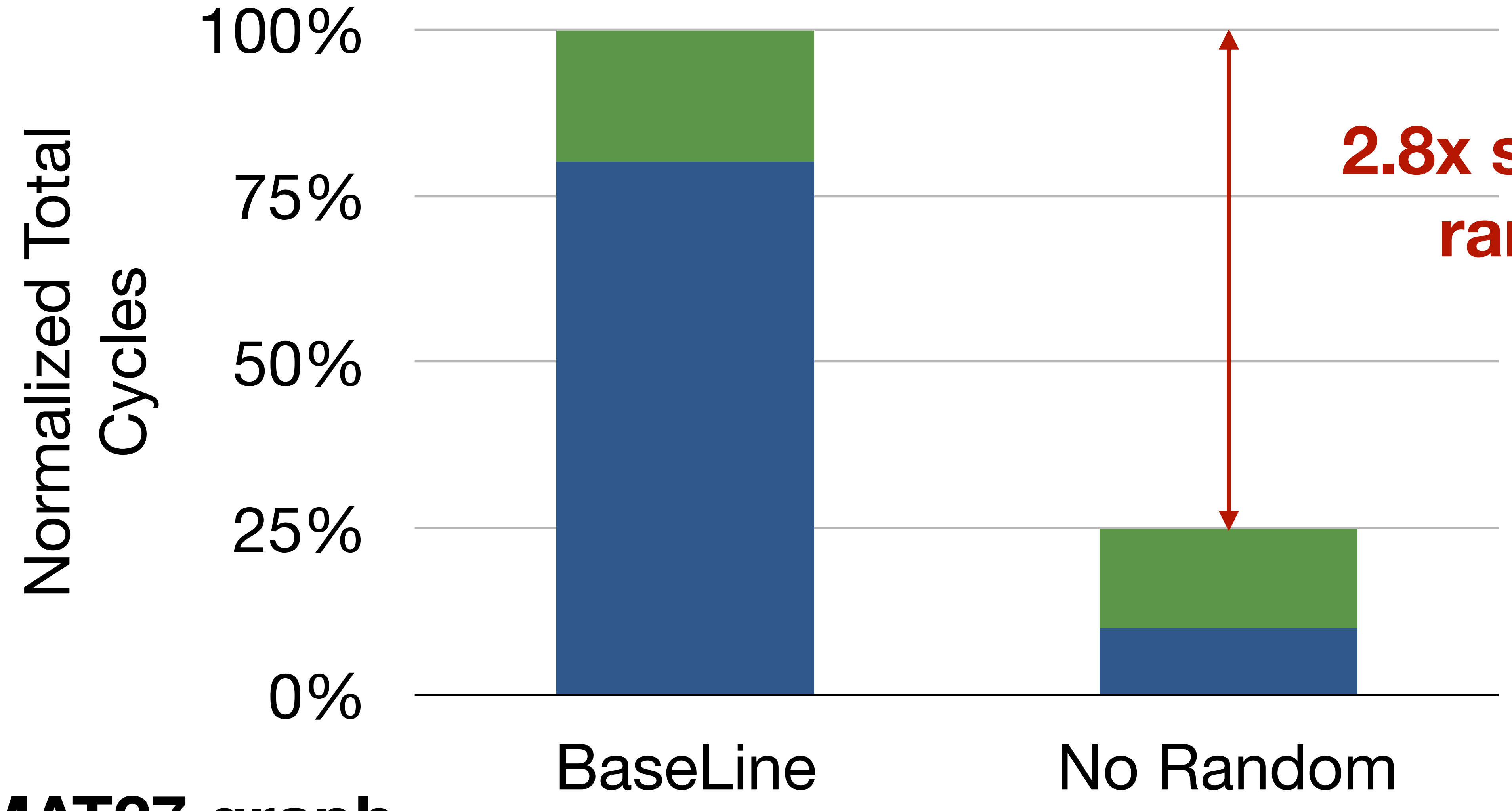


on RMAT27 graph

PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[0]/outDegree[0];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Removing Random Accesses (Incorrect)

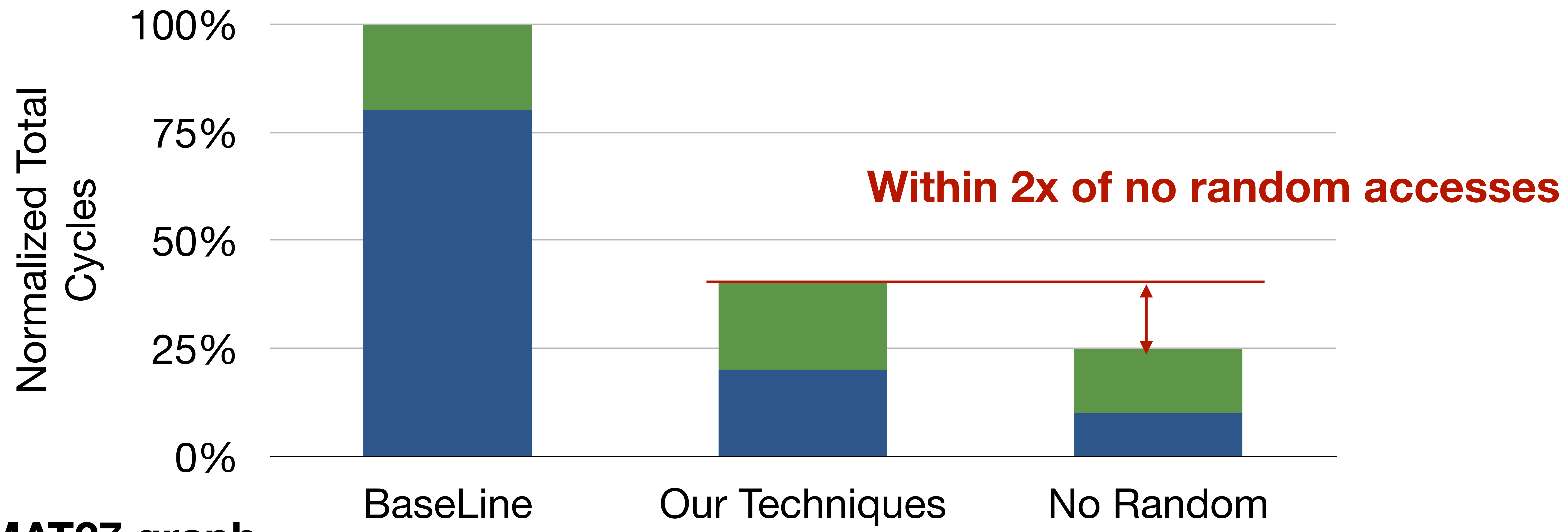


2.8x speedup if we can eliminate random memory accesses

on RMAT27 graph

PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



Within 2x of no random accesses

on RMAT27 graph

Frequency-based Vertex Reordering

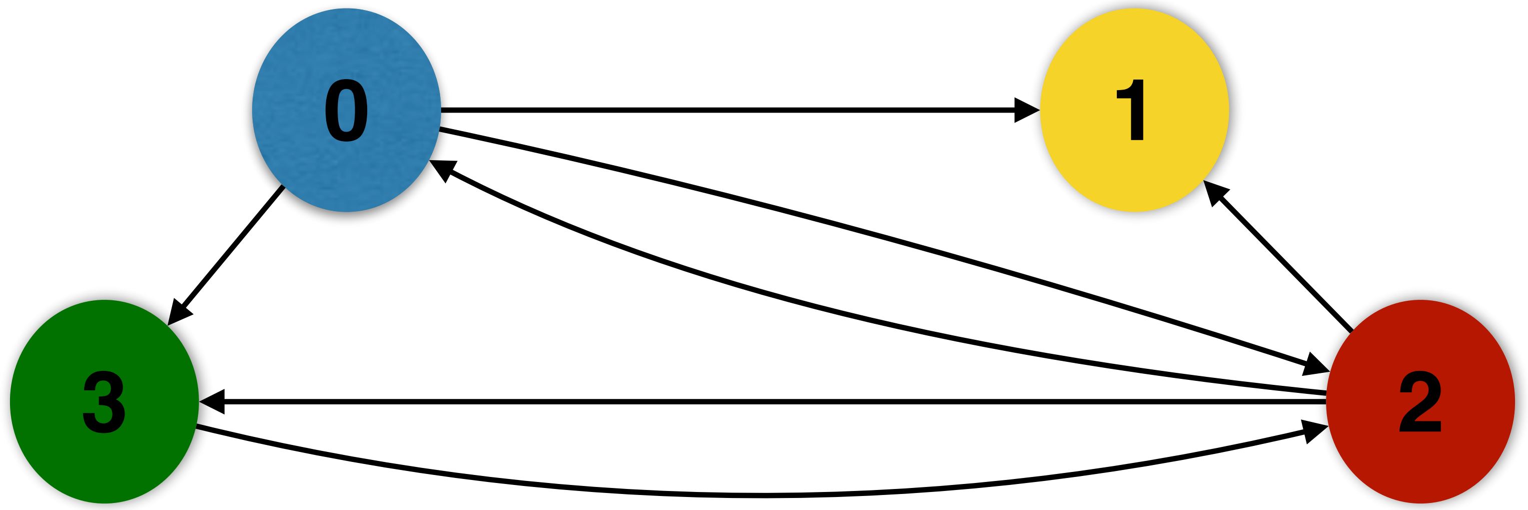
- Key Observations
    - Cache lines are underutilized
    - Certain vertices are much more likely to be accessed than other vertices

Frequency-based Vertex Reordering

- Key Observations
  - Cache lines are underutilized
  - Certain vertices are much more likely to be accessed than other vertices
- Design
  - Group together the frequently accessed nodes
  - Keep the ordering of average degree nodes

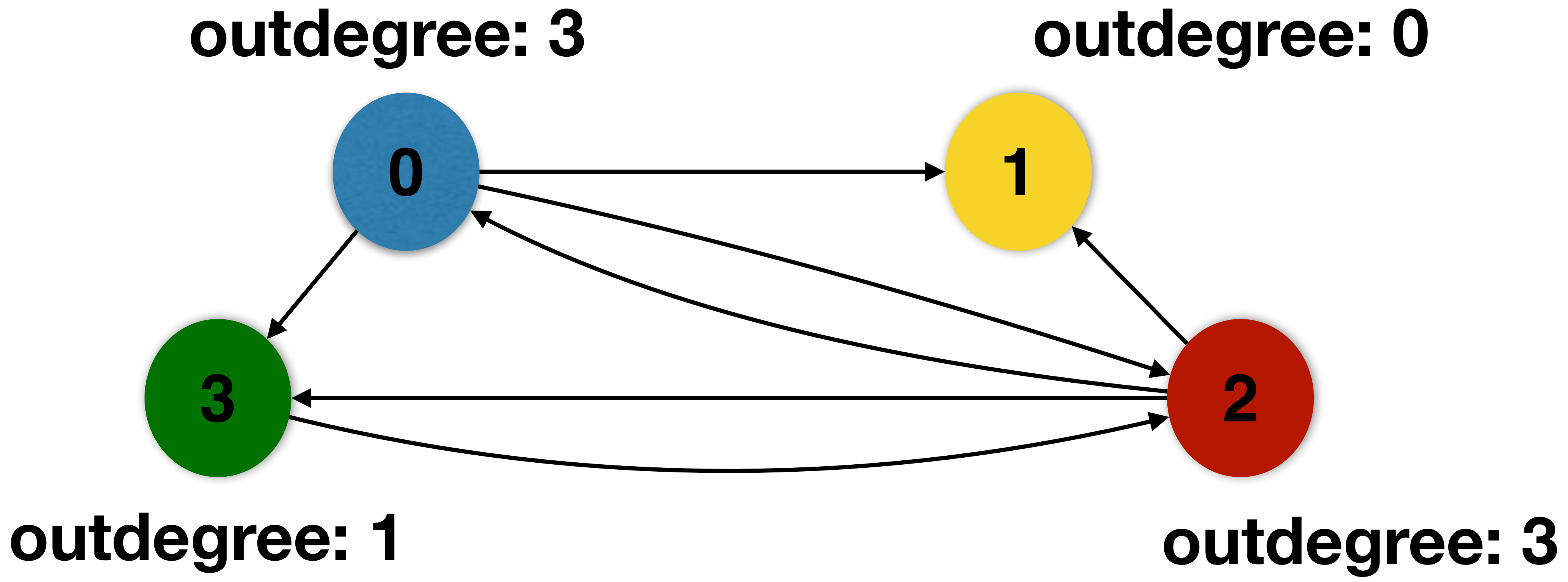
Frequency-based Vertex Reordering

0	1	2	3



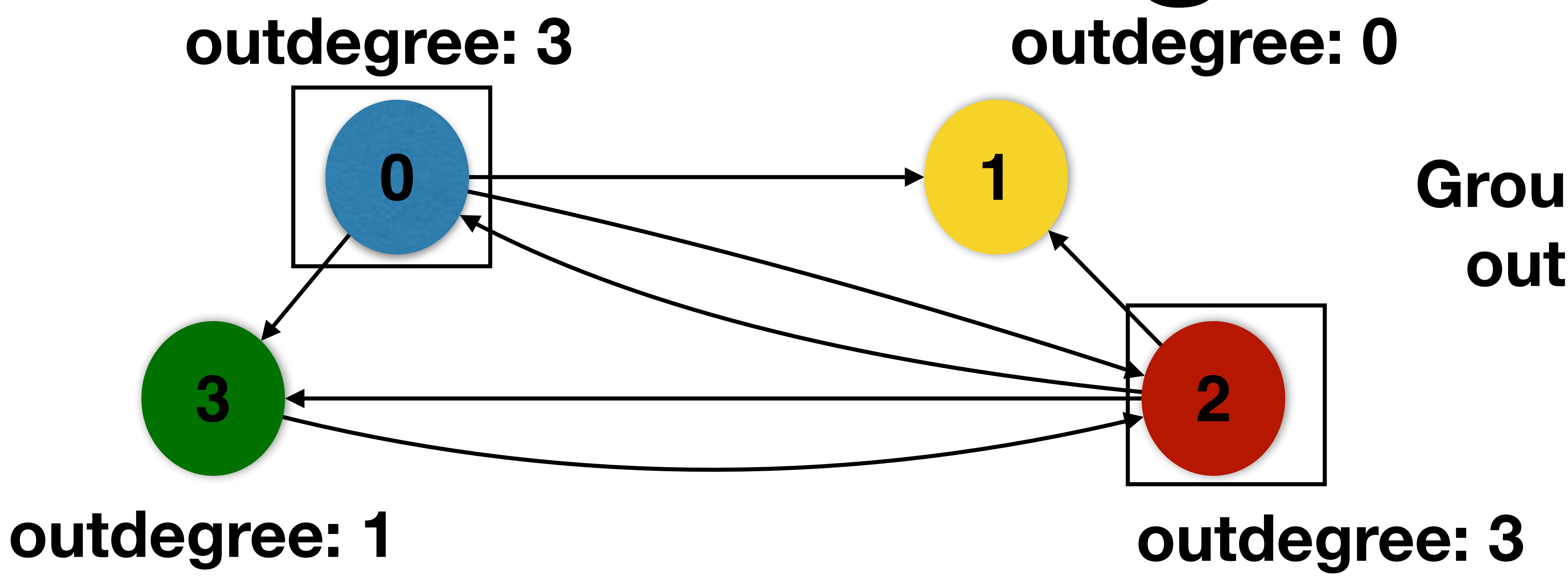
Frequency-based Vertex Reordering

0	1	2	3



Frequency-based Vertex Reordering

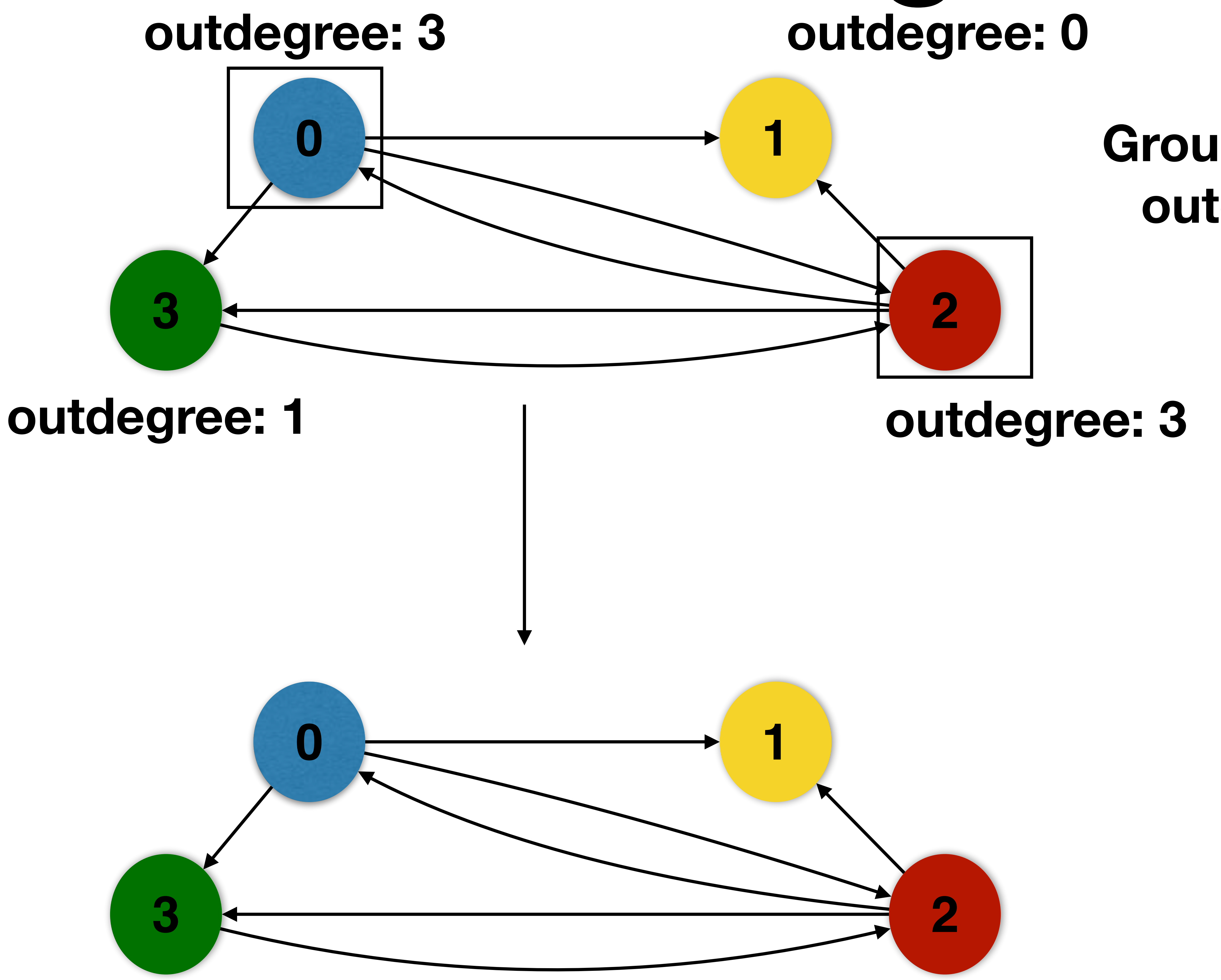
0	1	2	3



Group together high outdegree nodes

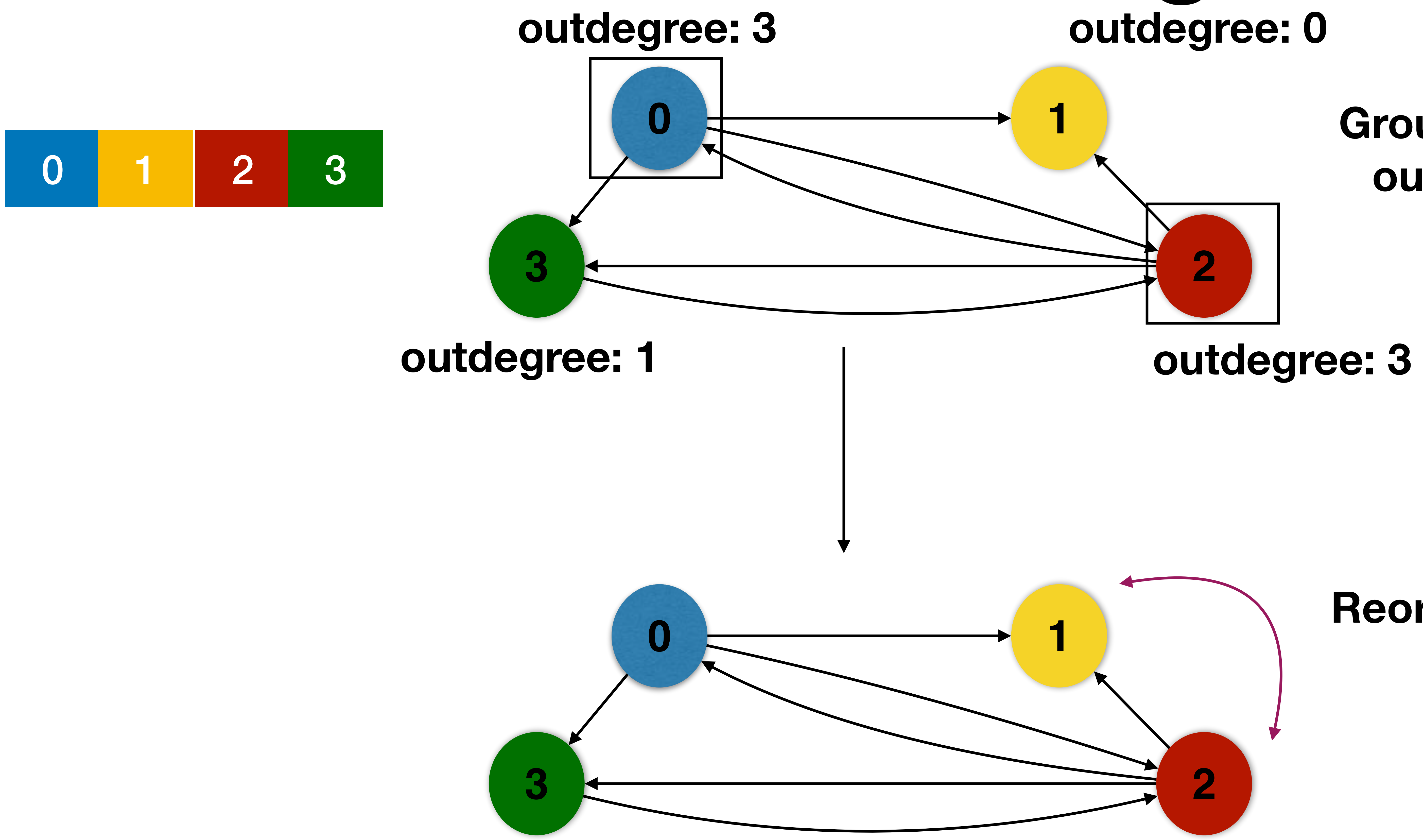
Frequency-based Vertex Reordering

0	1	2	3



Group together high outdegree nodes

Frequency-based Vertex Reordering



outdegree: 3

outdegree: 0

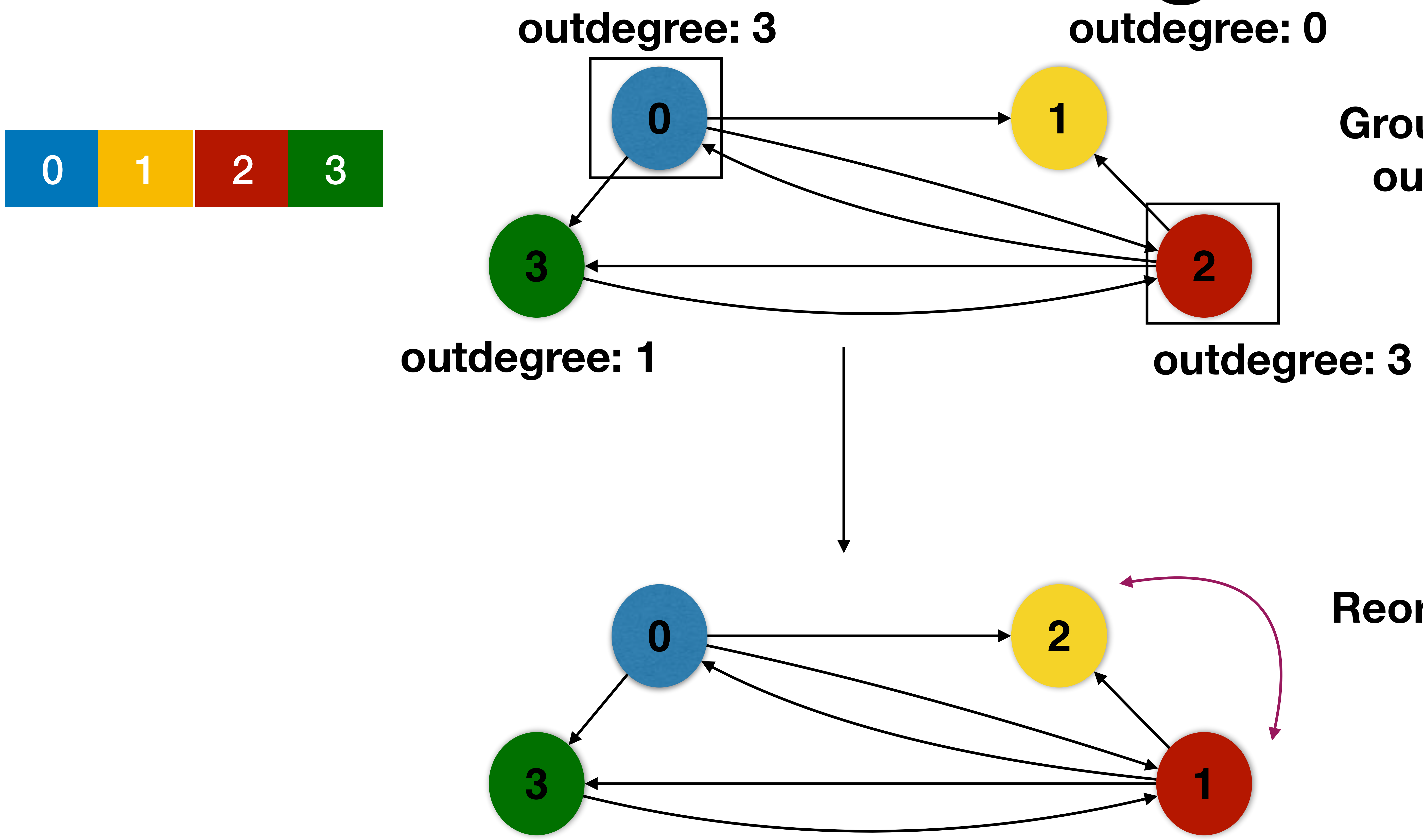
outdegree: 1

outdegree: 3

Group together high outdegree nodes

Reorder nodes 1 and 2

Frequency-based Vertex Reordering



outdegree: 3

outdegree: 0

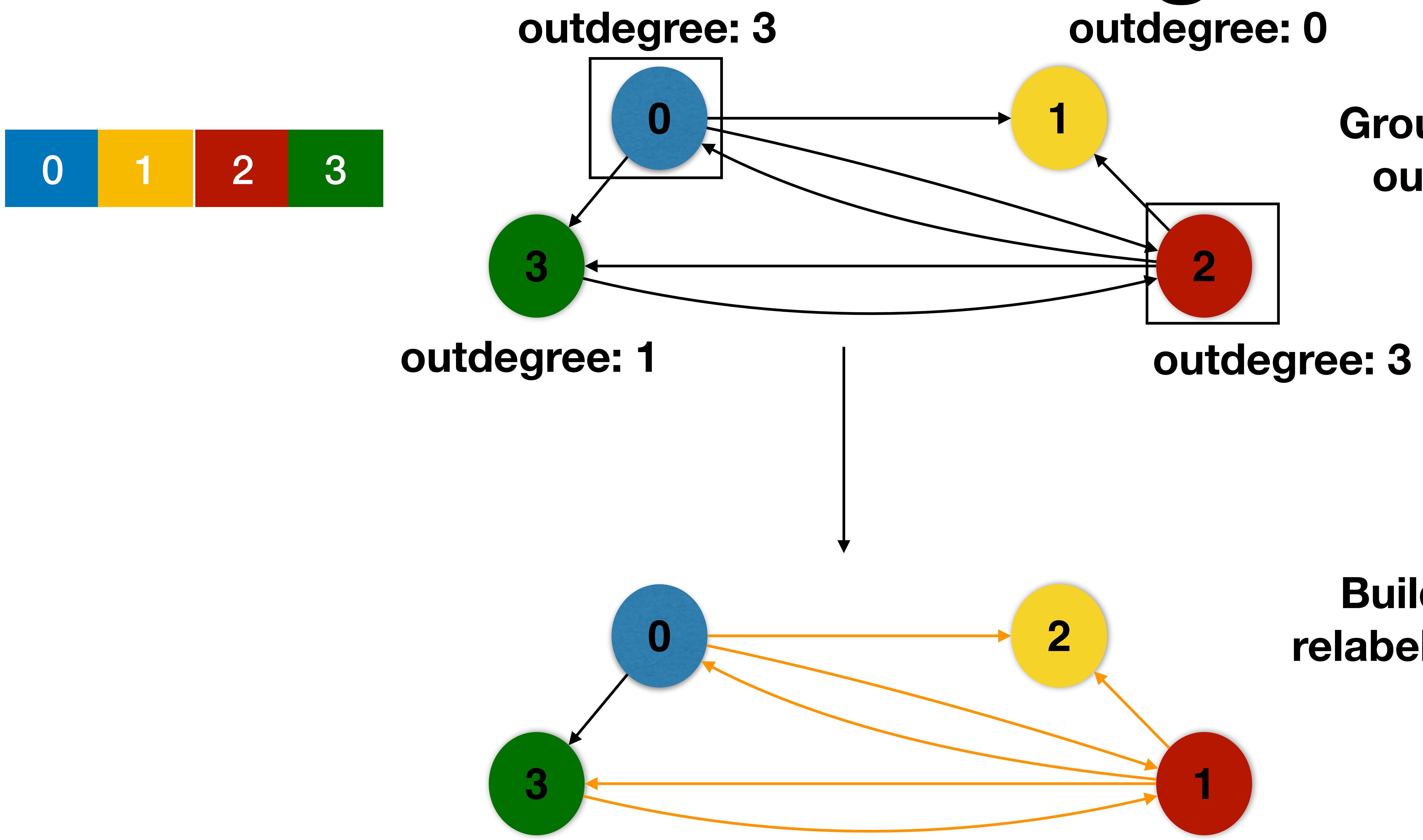
outdegree: 1

outdegree: 3

Group together high outdegree nodes

Reorder nodes 1 and 2

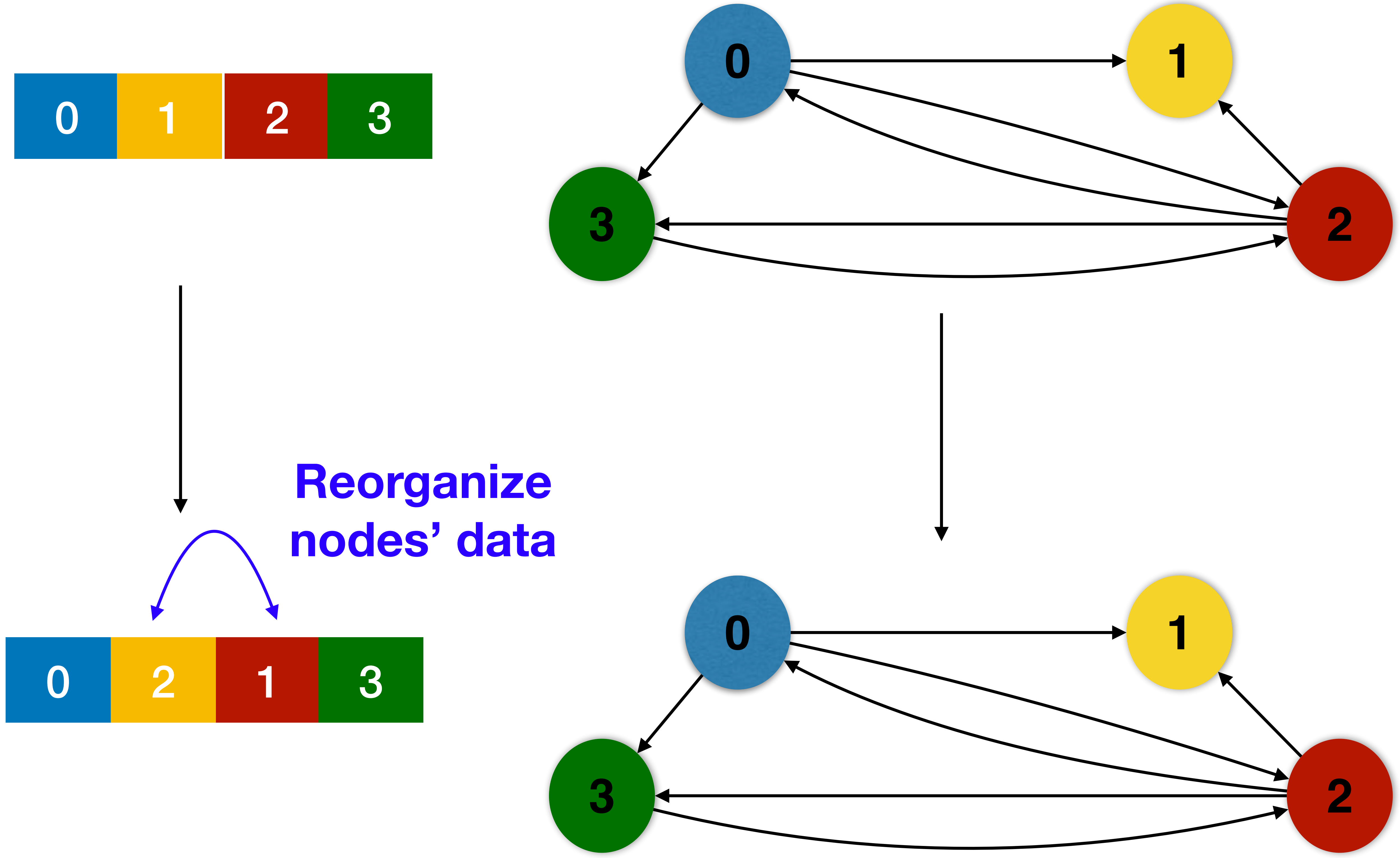
Frequency-based Vertex Reordering



Group together high outdegree nodes

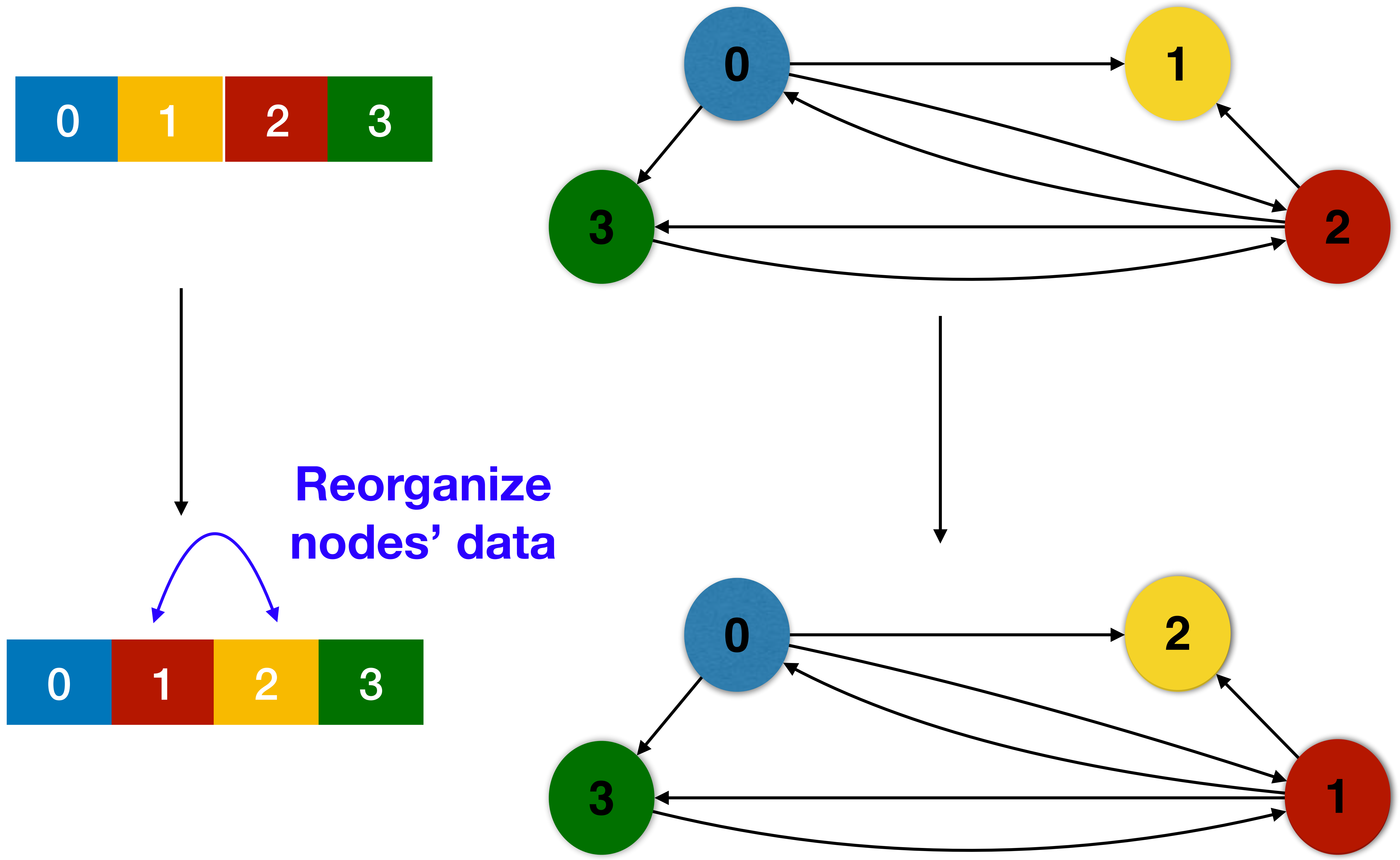
Build a new CSR, relabel corresponding edges

Frequency-based Vertex Reordering



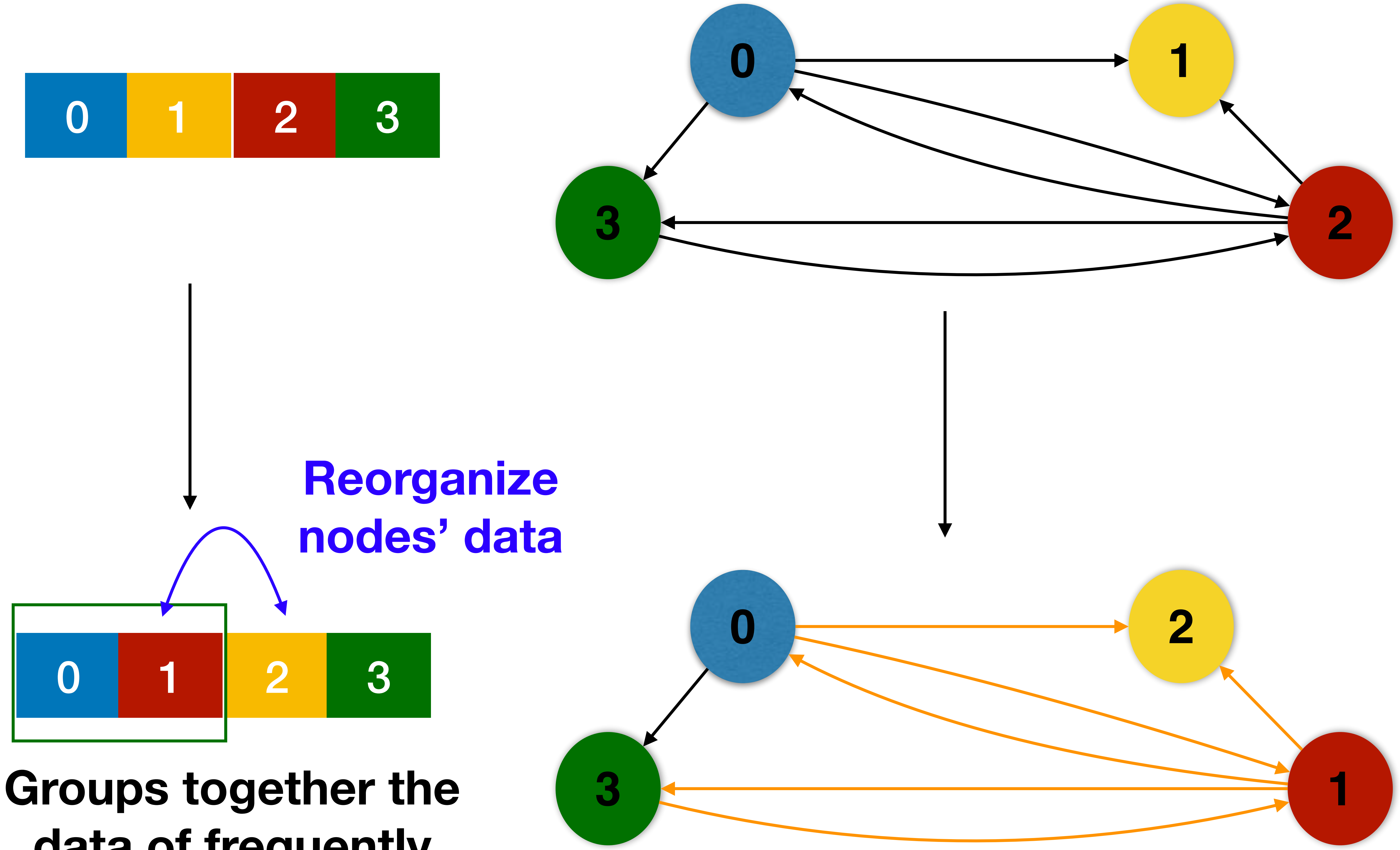
Reorganize nodes’ data

Frequency-based Vertex Reordering



Reorganize nodes' data

Frequency-based Vertex Reordering



Reorganize nodes' data

Groups together the data of frequently accessed nodes in one cache line

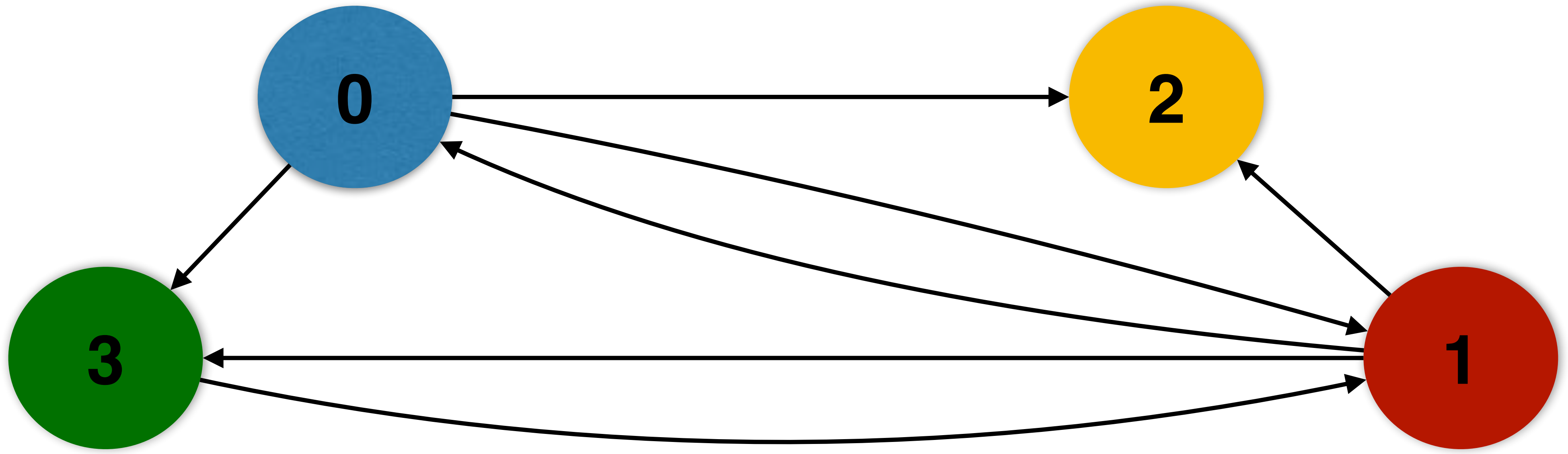
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache

#hits: 0
#misses: 0

0	1	2	3



PageRank

Focus on the random memory accesses on ranks array

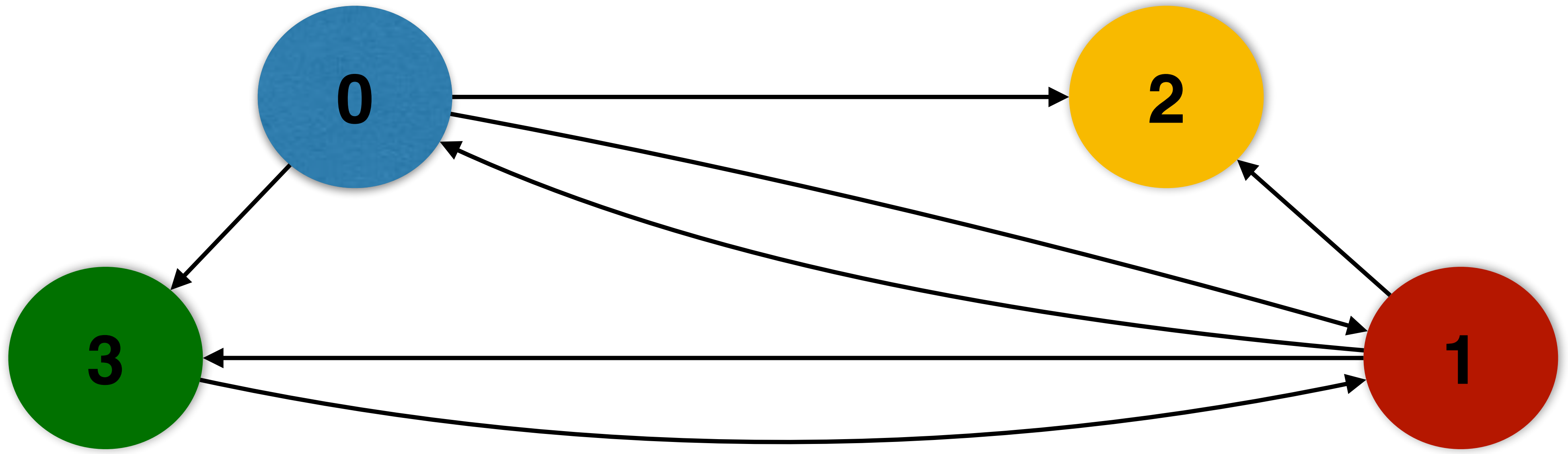
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache

#hits: 0

#misses: 0

0	1	2	3



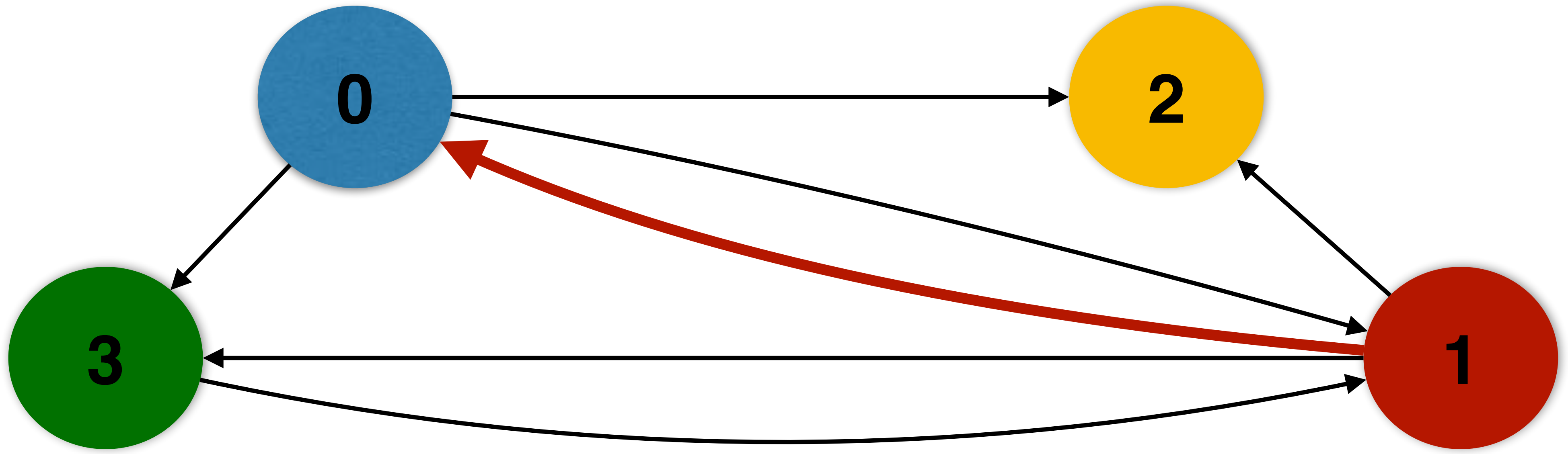
PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache

#hits: 0
#misses: 0

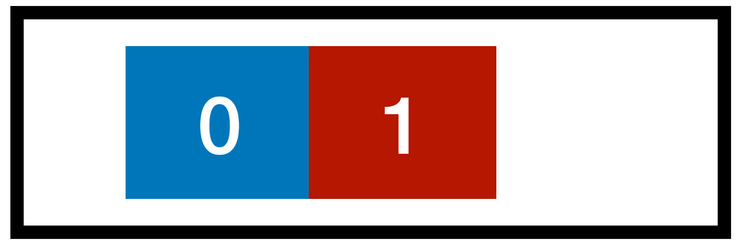
0	1	2	3



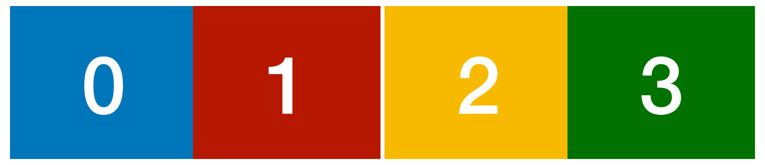
PageRank

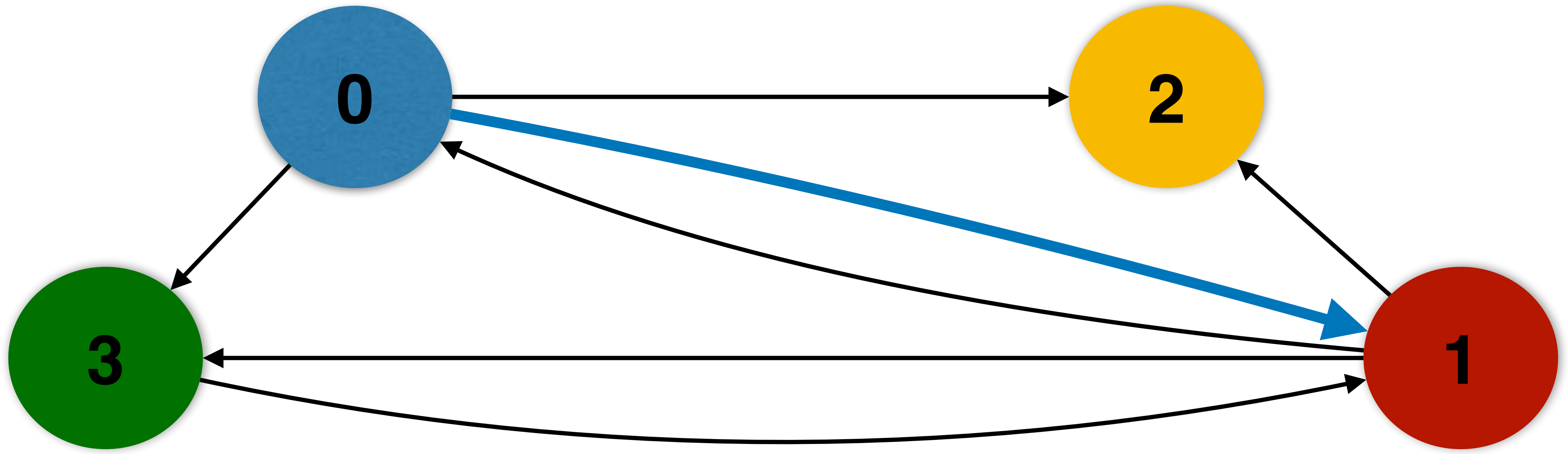
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache



#hits: 0
#misses: 1

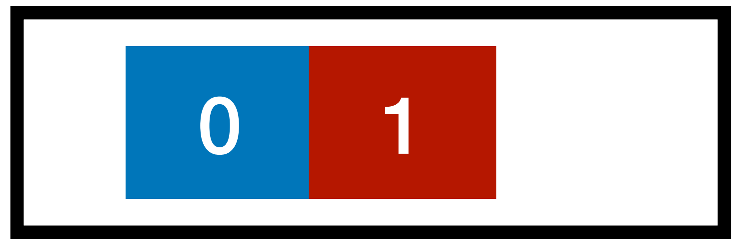




PageRank

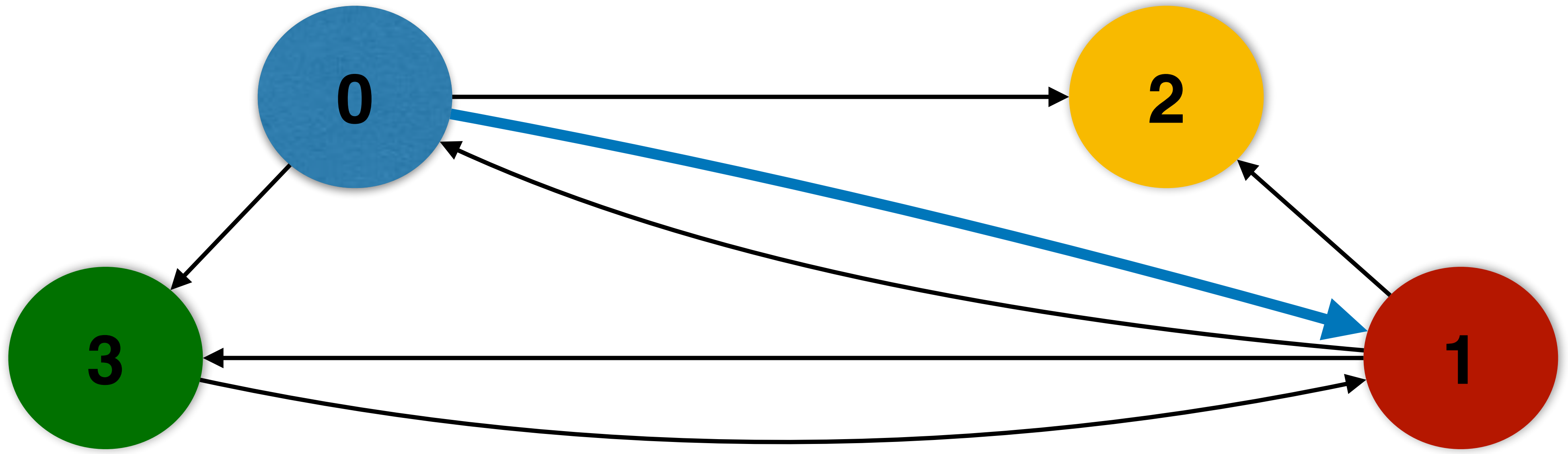
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

Cache



#hits: 1

#misses: 1



PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

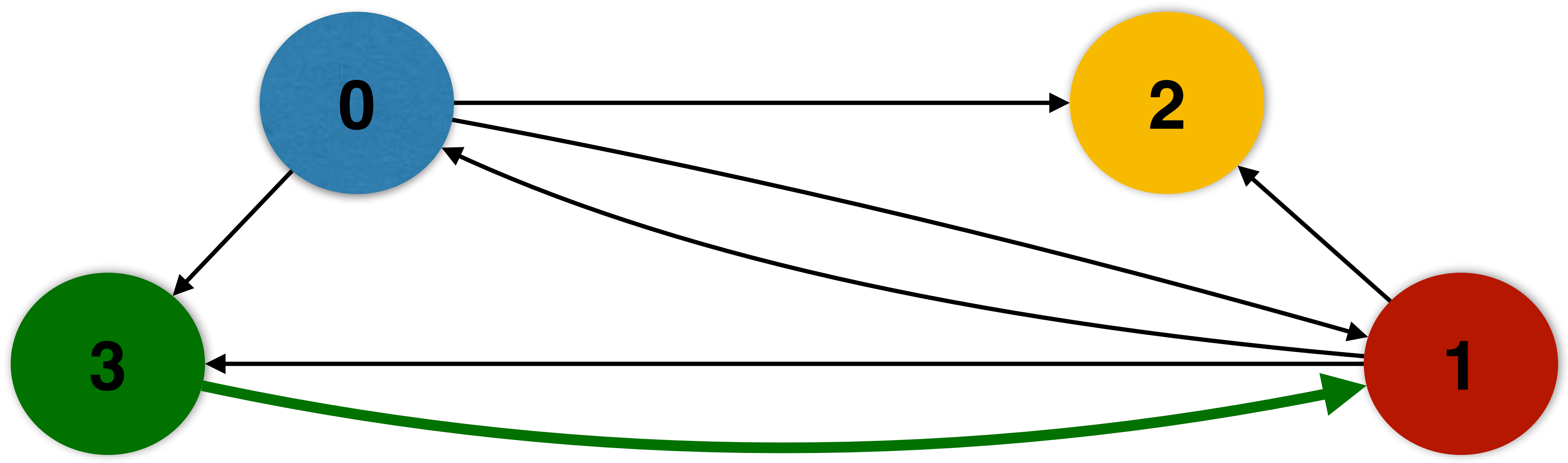
Cache

2	3

#hits: 1

#misses: 2

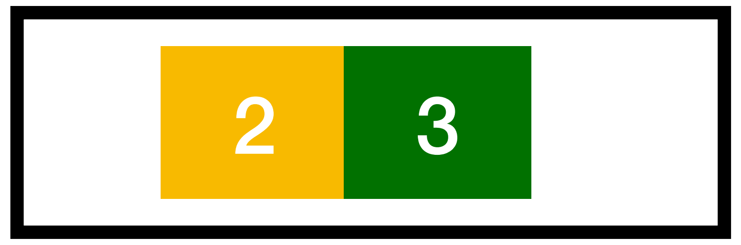
0	1	2	3



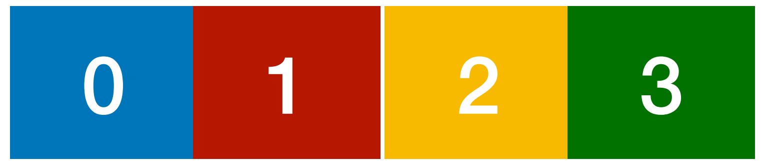
PageRank

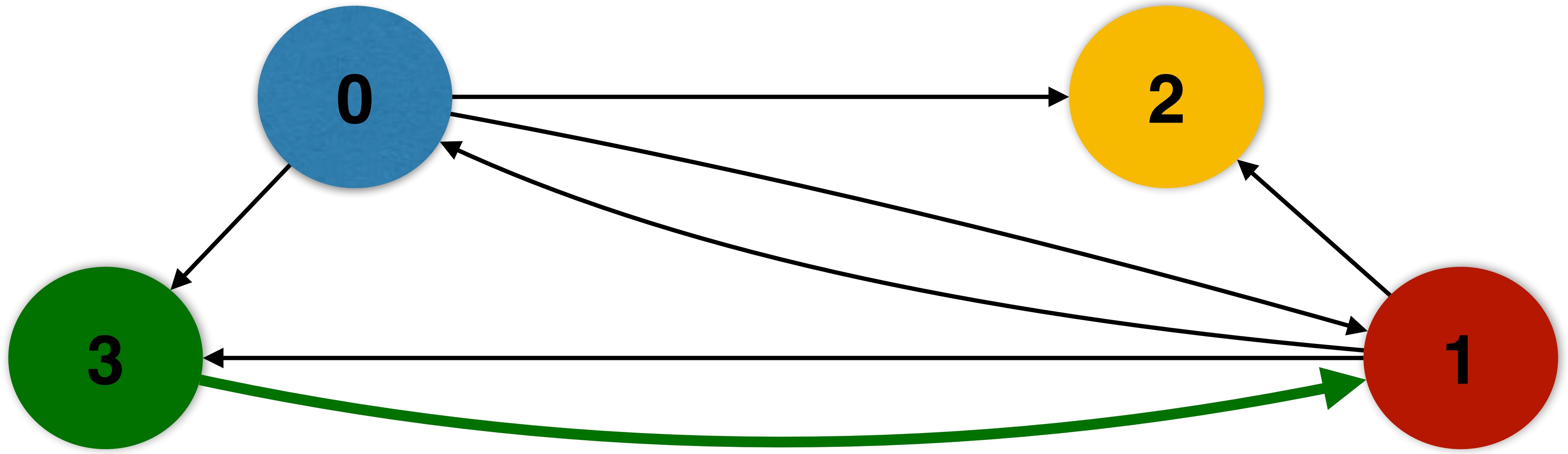
```
while …
     for node : graph.vertices
         for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[ngh]/outDegree[ngh];
     for node : graph.vertices
         newRanks[node] = baseScore + damping*newRanks[node];
     swap ranks and newRanks
```

Cache



#hits: 1
#misses: 2





PageRank

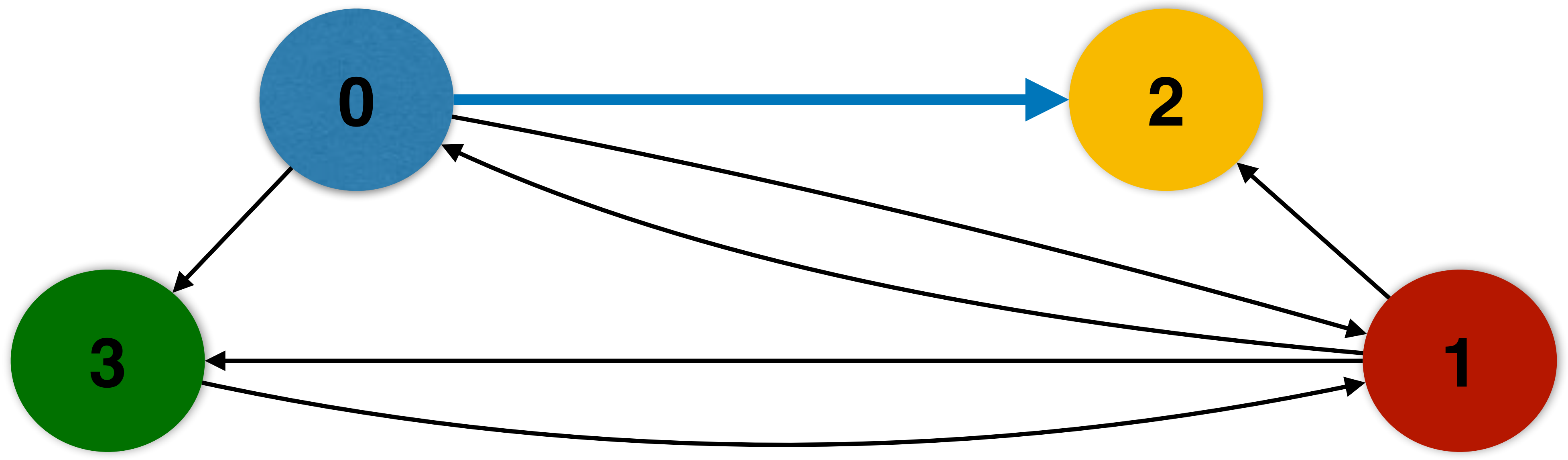
```
while …
     for node : graph.vertices
         for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[ngh]/outDegree[ngh];
     for node : graph.vertices
         newRanks[node] = baseScore + damping*newRanks[node];
     swap ranks and newRanks
```

Cache

0	1	

#hits: 1
#misses: 3

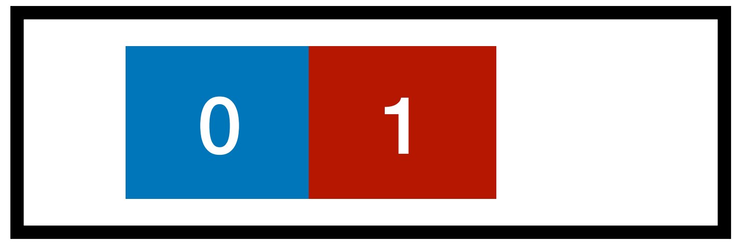
0	1	2	3



PageRank

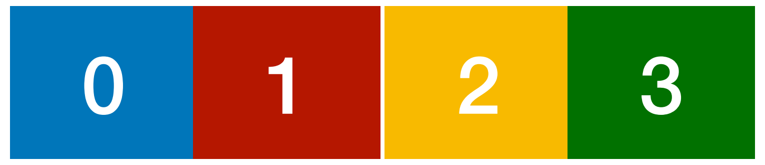
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

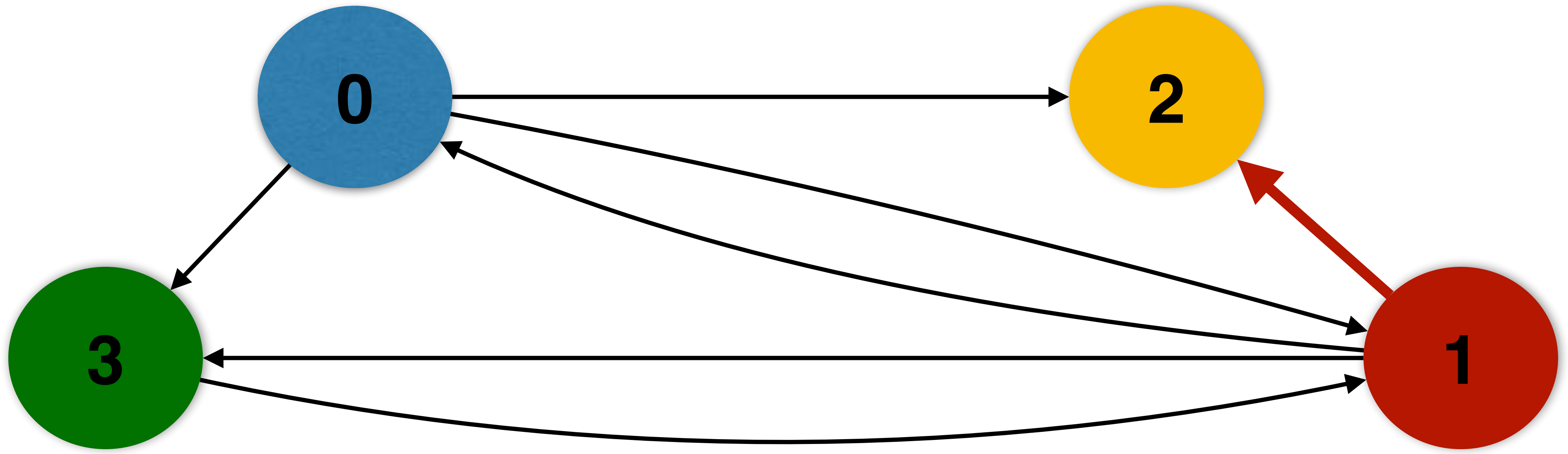
Cache



#hits: 2

#misses: 3





PageRank

```
while …
     for node : graph.vertices
         for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
         newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```

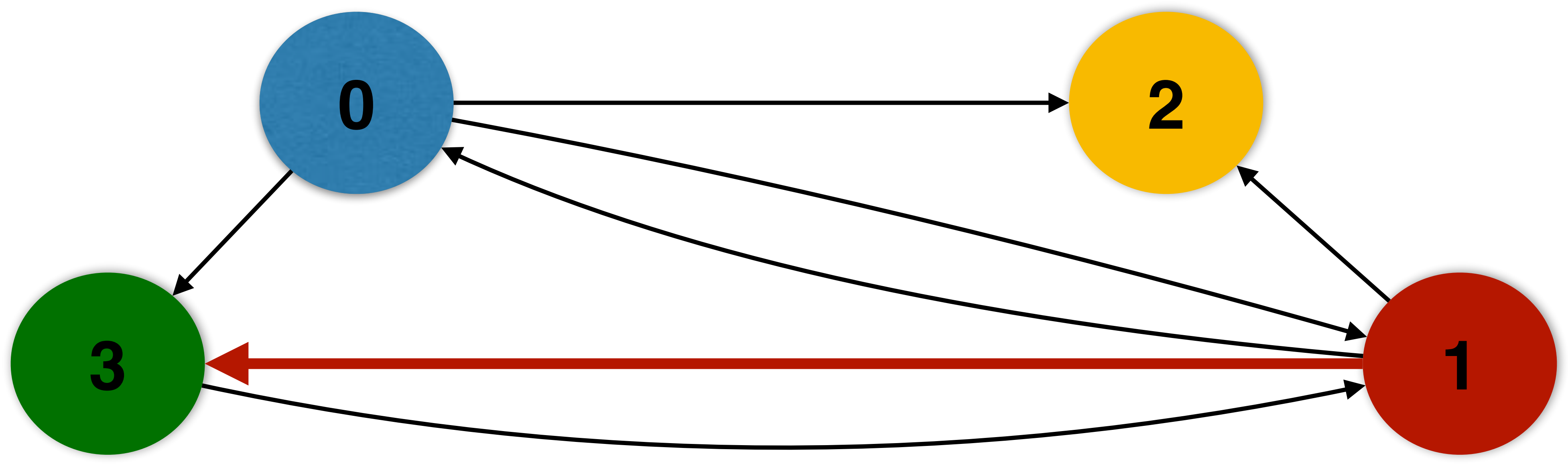
Cache

0	1	

#hits: 3

#misses: 3

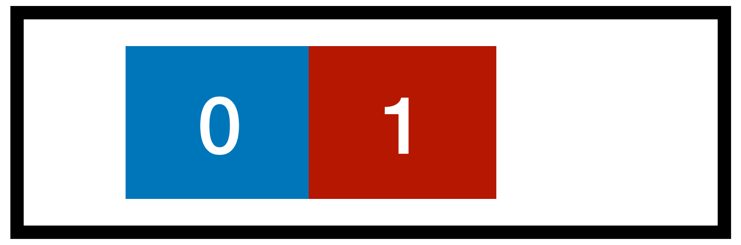
0	1	2	3



PageRank

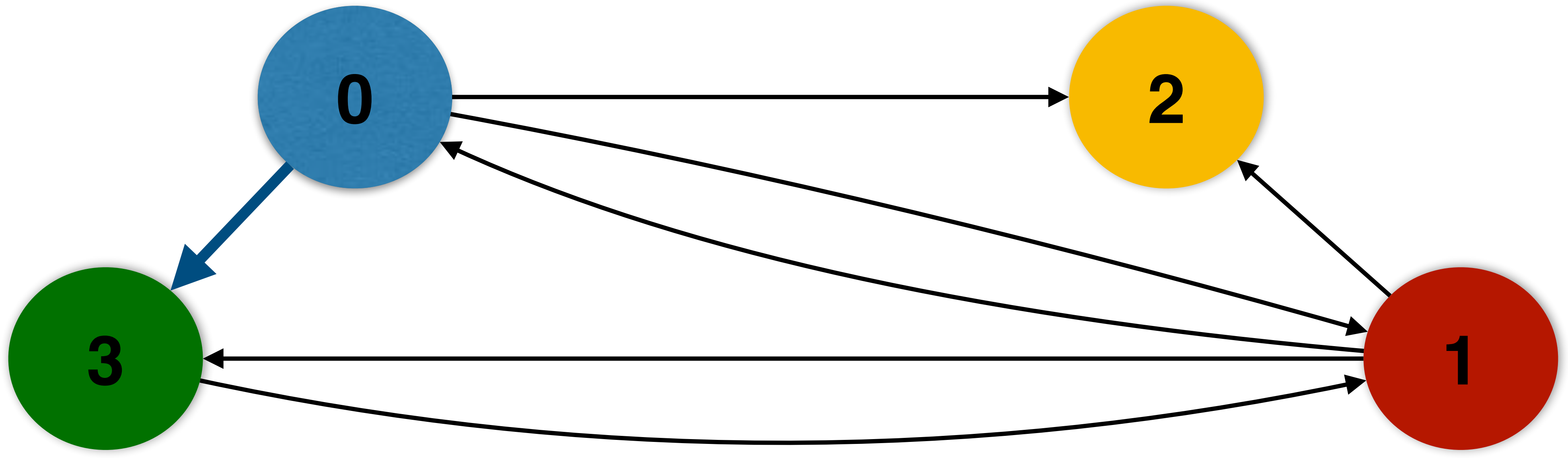
```
while …
     for node : graph.vertices
         for ngh : graph.getInNeighbors(node)
              newRanks[node] += ranks[ngh]/outDegree[ngh];
     for node : graph.vertices
         newRanks[node] = baseScore + damping*newRanks[node];
     swap ranks and newRanks
```

Cache



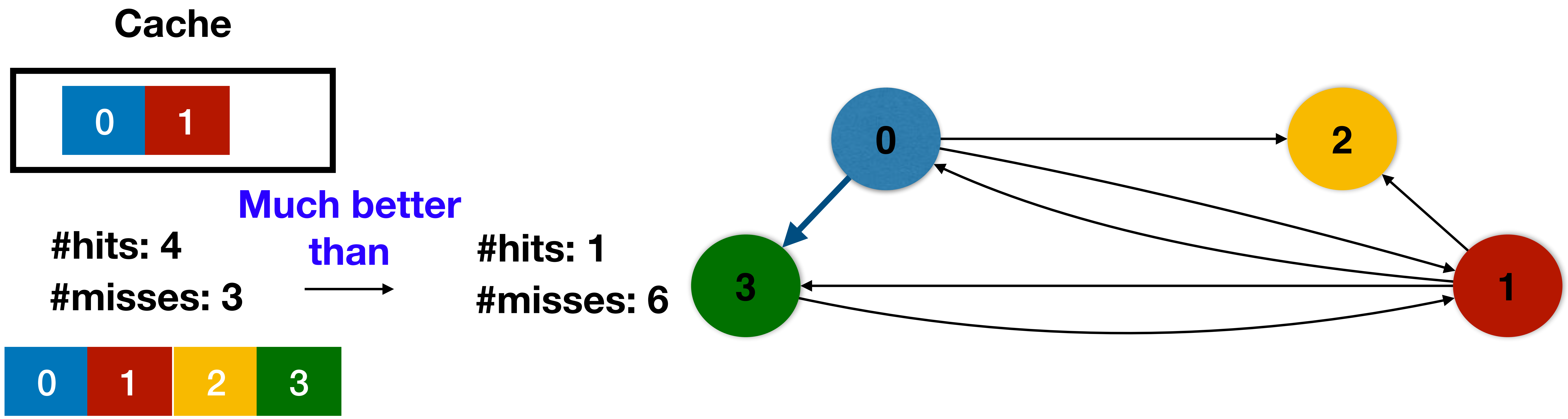
#hits: 4

#misses: 3



PageRank

```
while …
  for node : graph.vertices
    for ngh : graph.getInNeighbors(node)
      newRanks[node] += ranks[ngh]/outDegree[ngh];
  for node : graph.vertices
    newRanks[node] = baseScore + damping*newRanks[node];
  swap ranks and newRanks
```



Cache

0	1

#hits: 4
#misses: 3

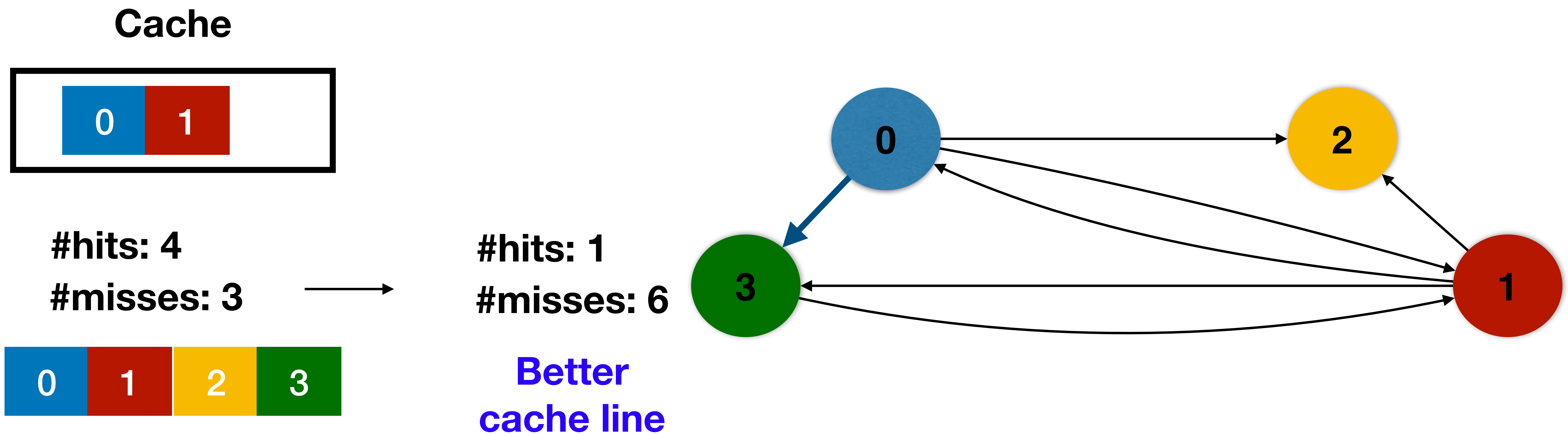
Much better than →

#hits: 1
#misses: 6

0	1	2	3

PageRank

```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



#hits: 4
#misses: 3

→

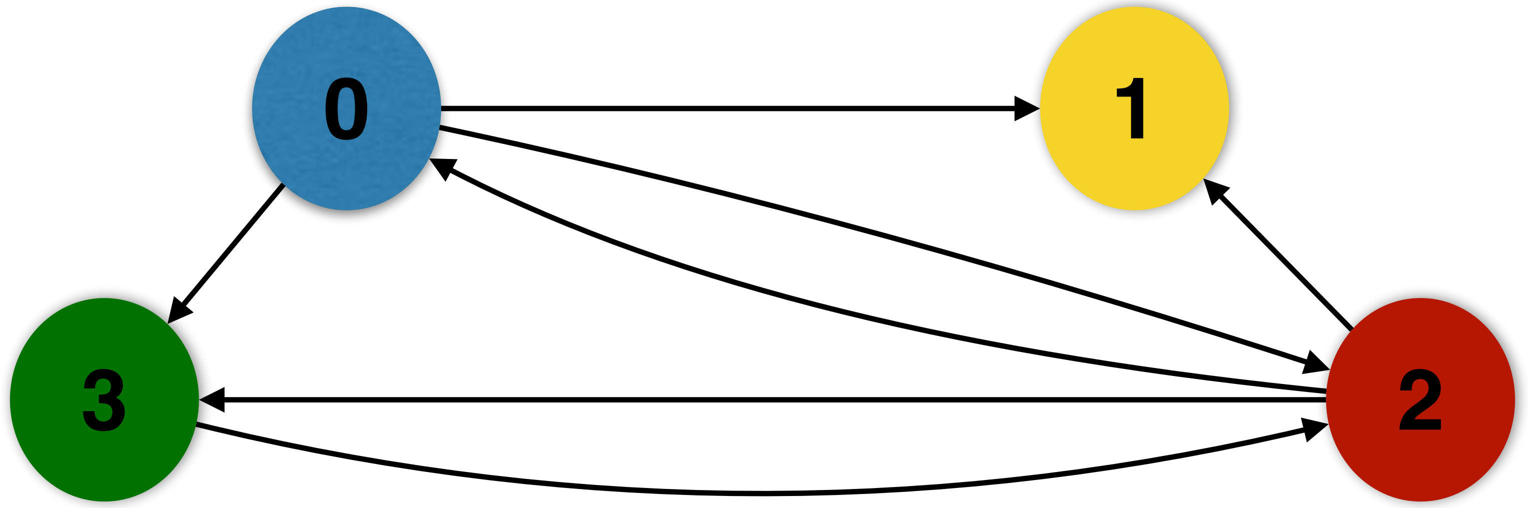
#hits: 1
#misses: 6

Better cache line utilization

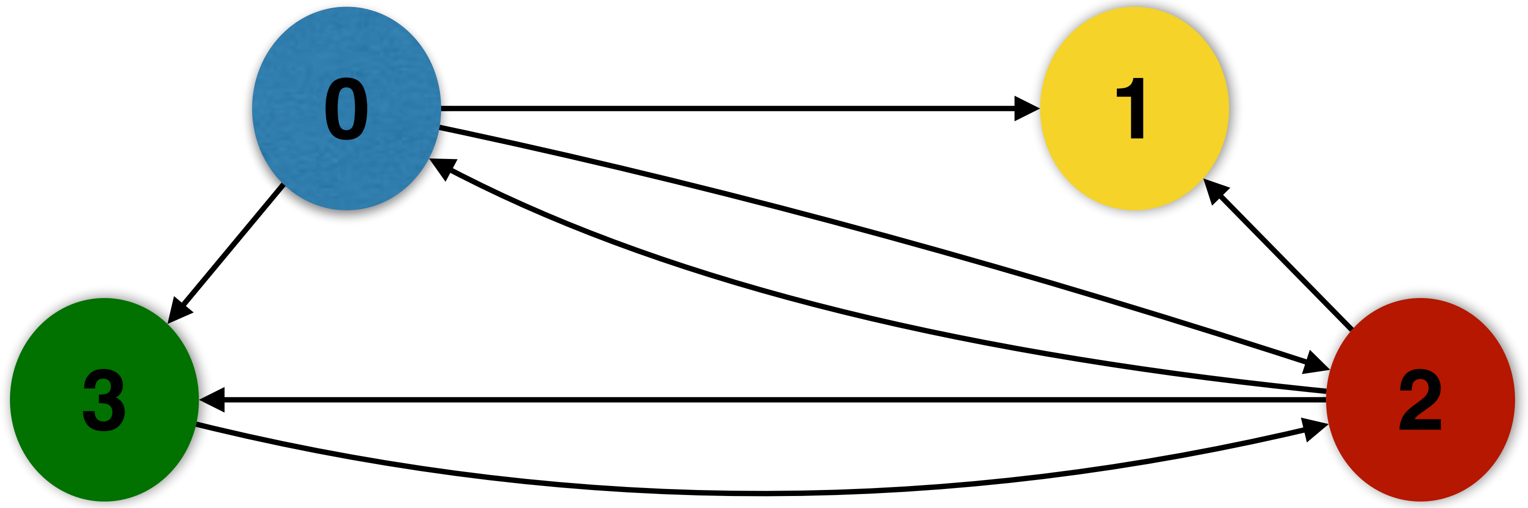
Cache-aware Partitioning

- Design
    - Partition the graph into subgraphs where the random access are limited to LLC
    - Process each partition sequentially and merge rank contributions for each partition

Graph Partitioning

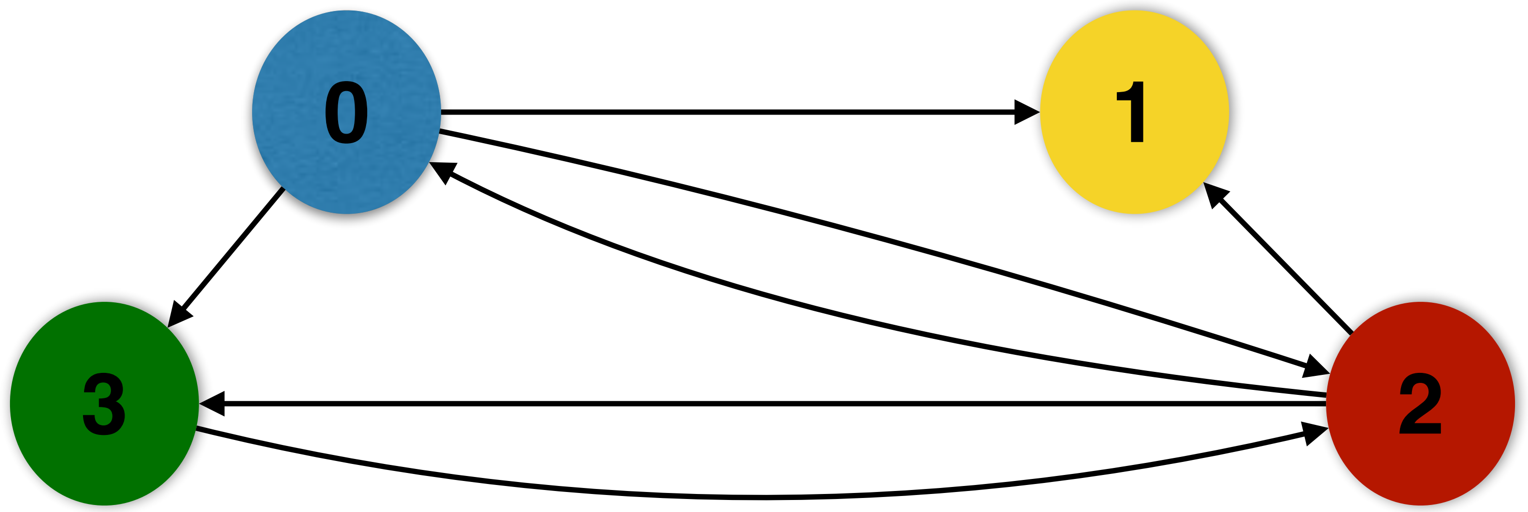


Graph Partitioning



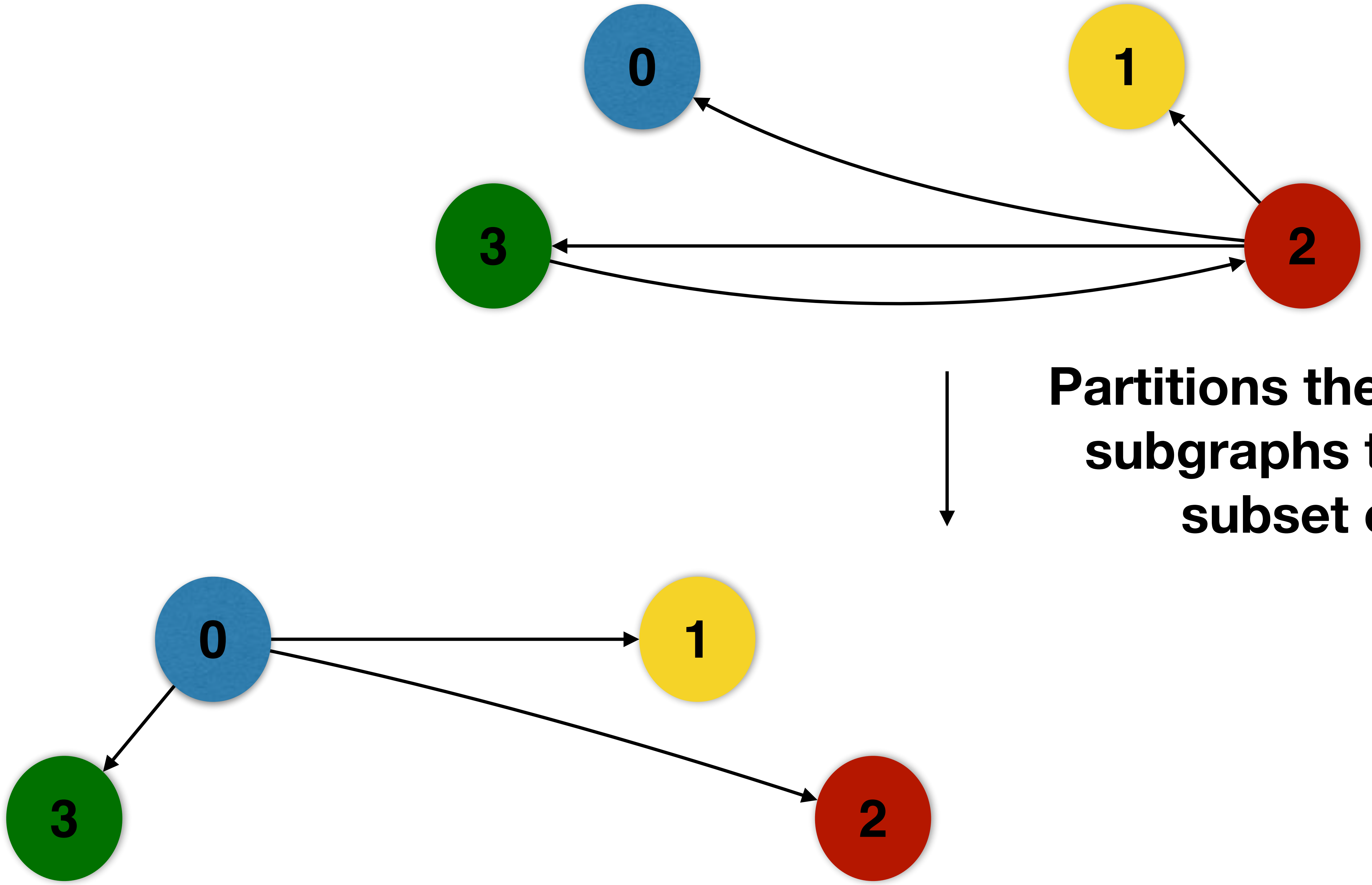
Partitions the original graph into subgraphs that only access a subset of nodes' data

Graph Partitioning



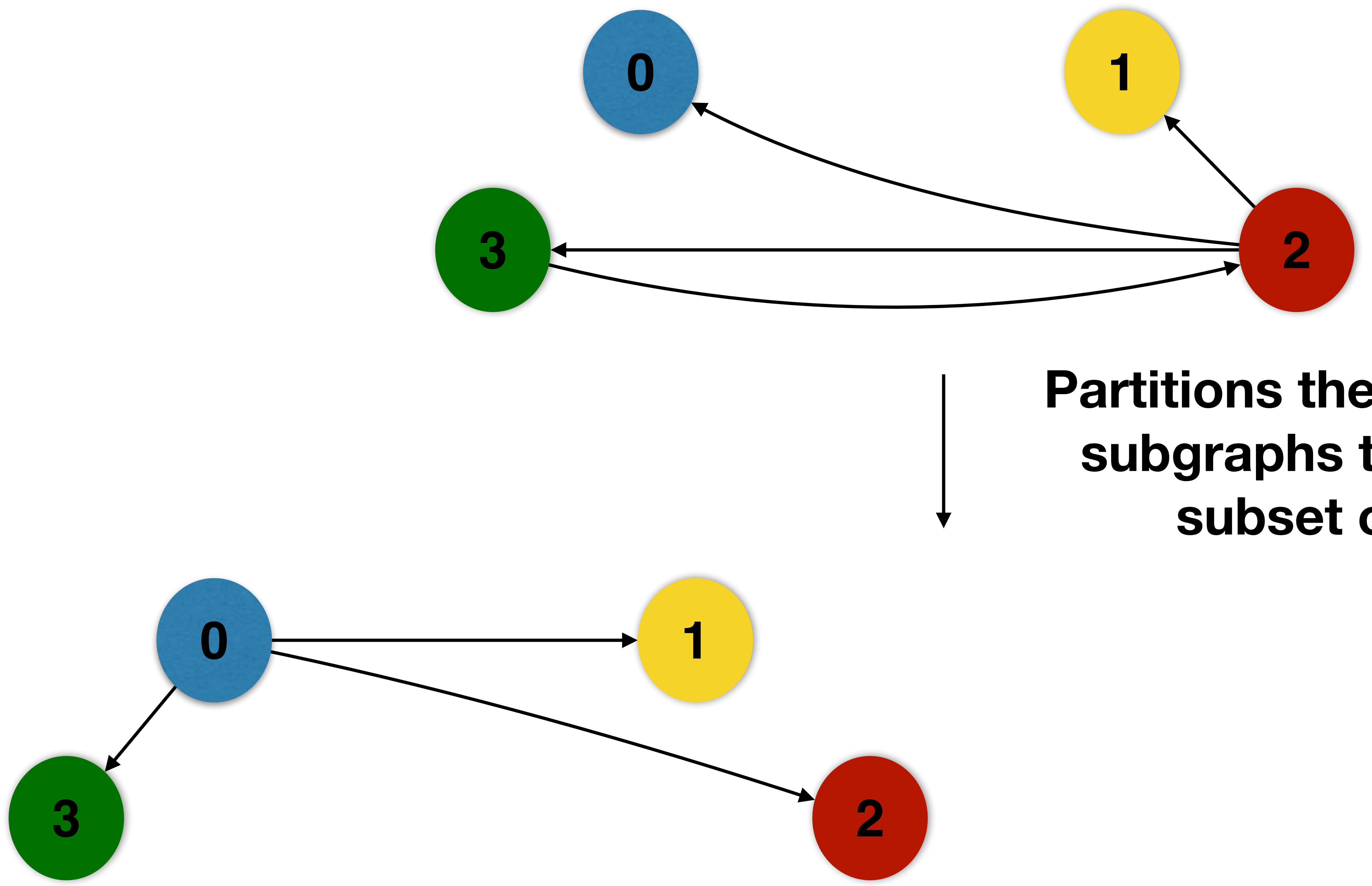
Partitions the original graph into subgraphs that only access a subset of nodes’ data

Graph Partitioning



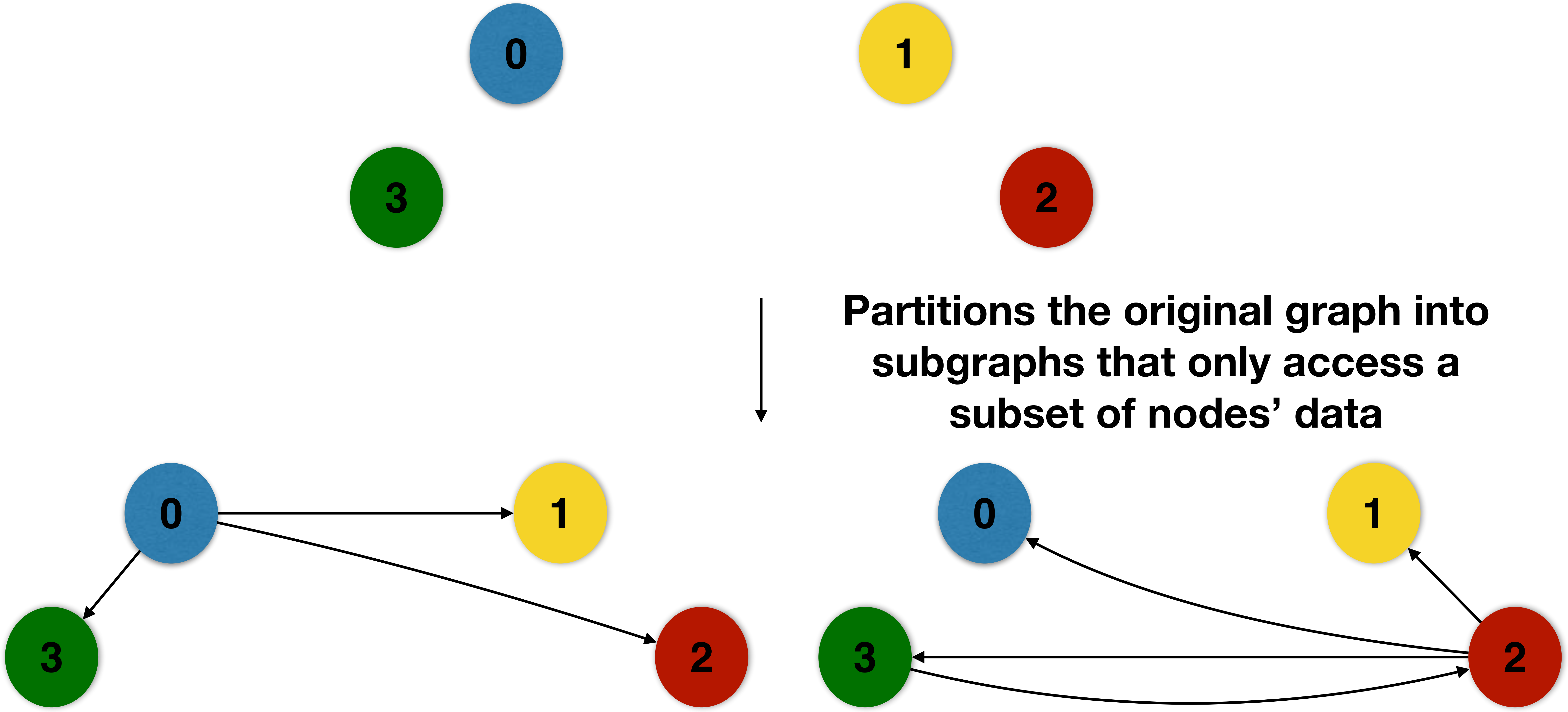
Partitions the original graph into subgraphs that only access a subset of nodes' data

Graph Partitioning



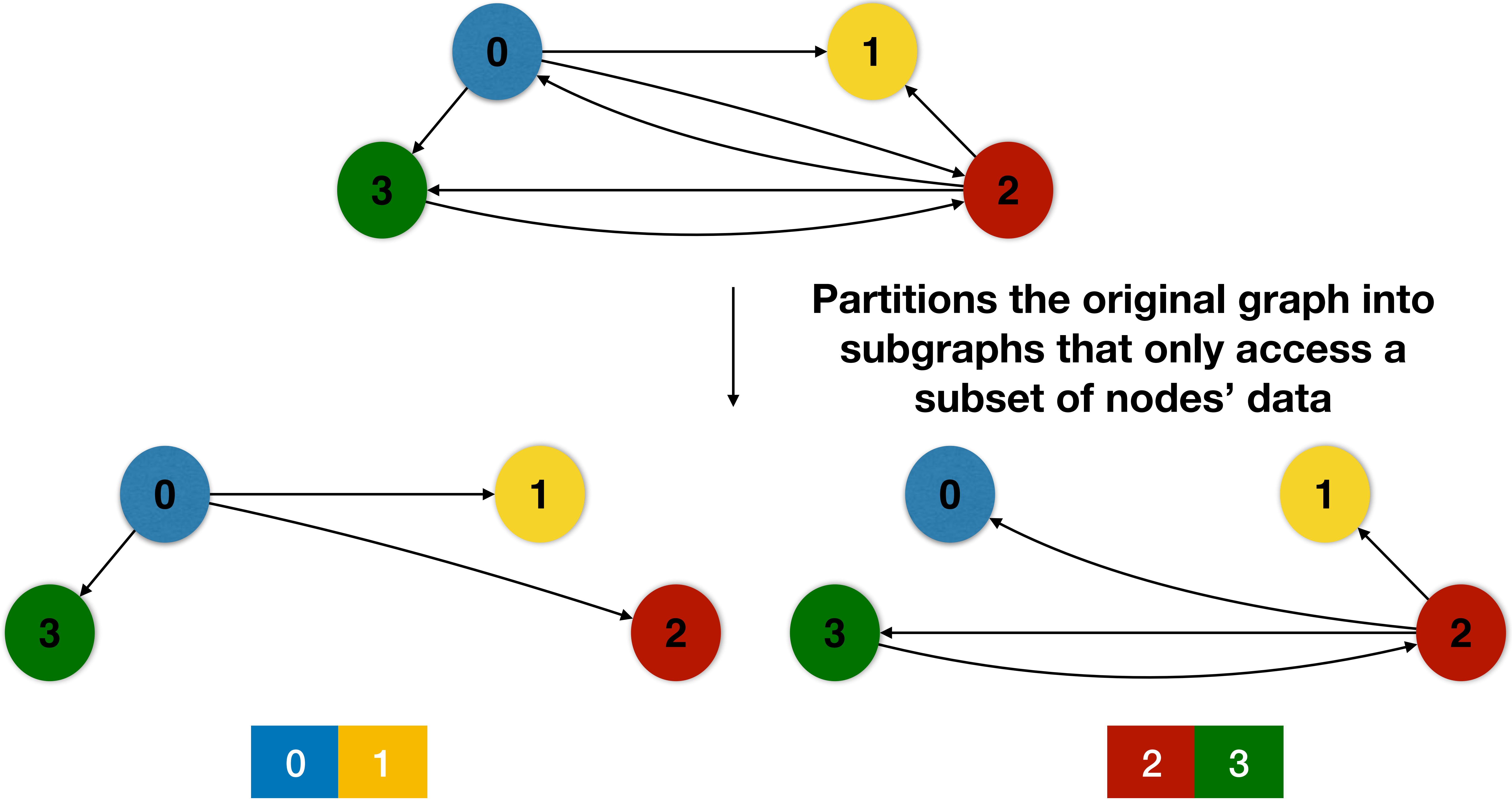
Partitions the original graph into subgraphs that only access a subset of nodes' data

Graph Partitioning



Partitions the original graph into subgraphs that only access a subset of nodes' data

Graph Partitioning

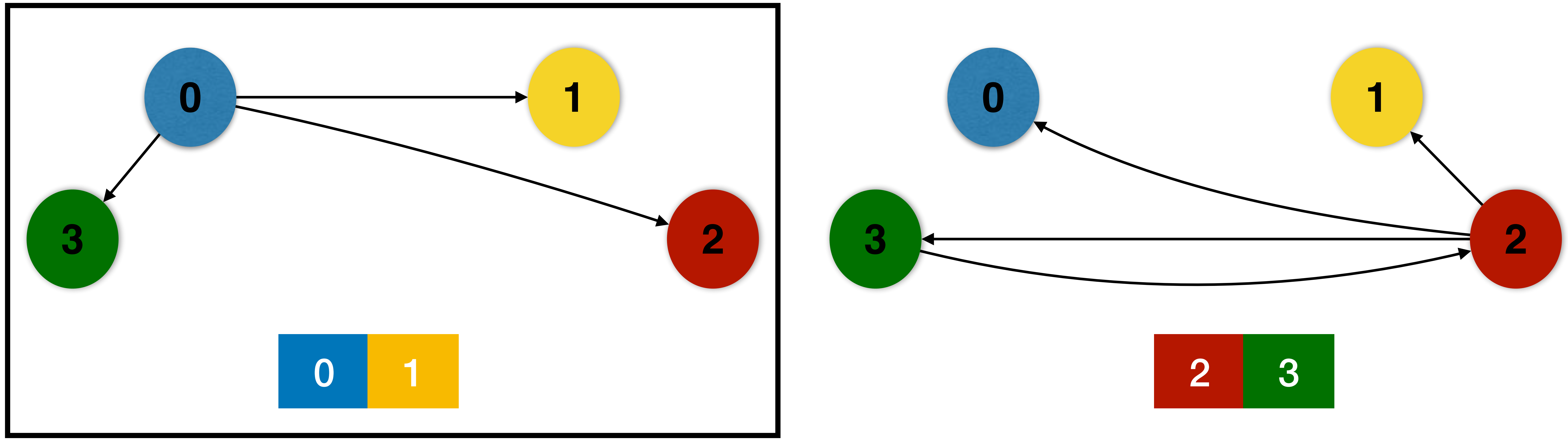


Partitions the original graph into subgraphs that only access a subset of nodes' data

Graph Processing

Cache

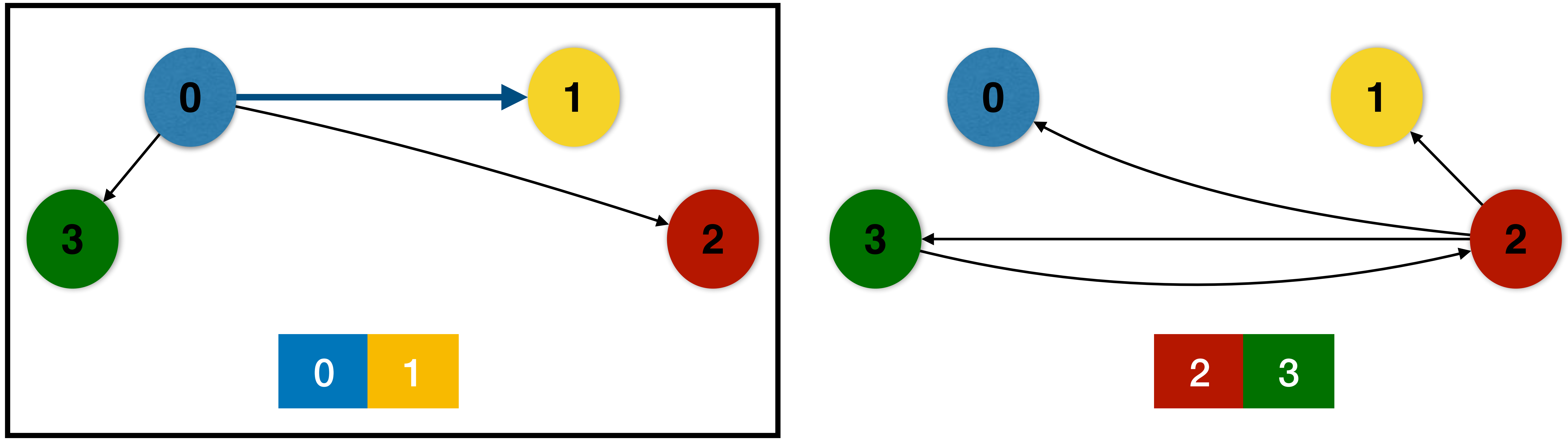
#hits: 0
#misses: 0



Graph Processing

Cache

#hits: 0
#misses: 0

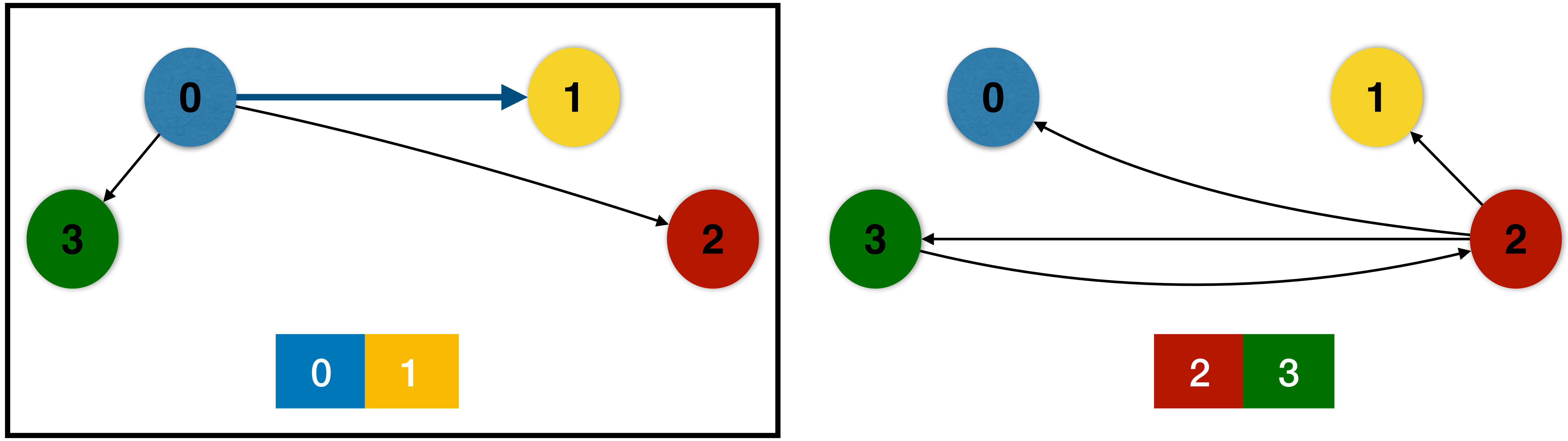


Graph Processing

Cache

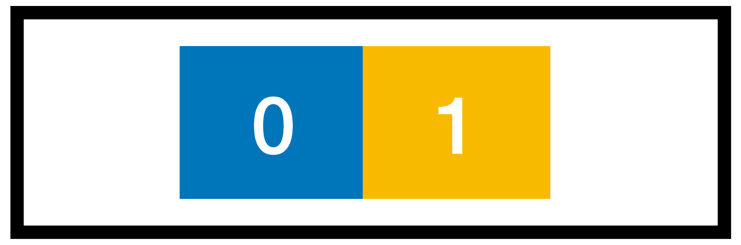
#hits: 0

#misses: 1

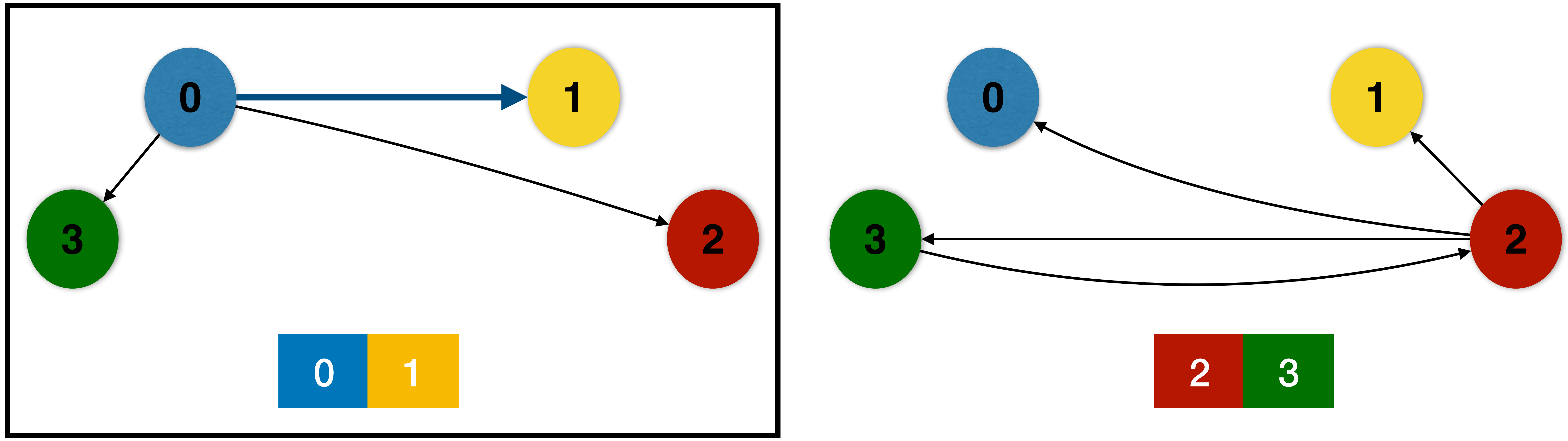


Graph Processing

Cache

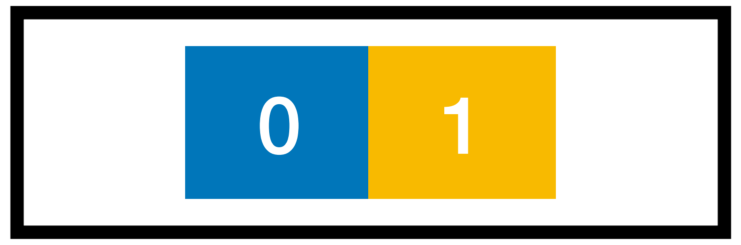


#hits: 0
#misses: 1

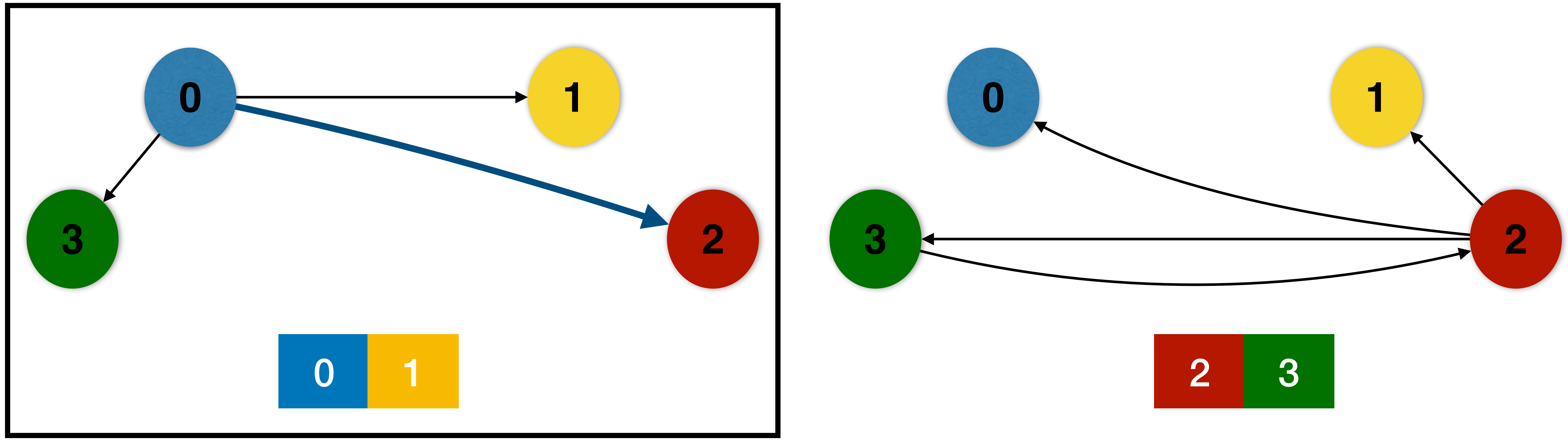


Graph Processing

Cache

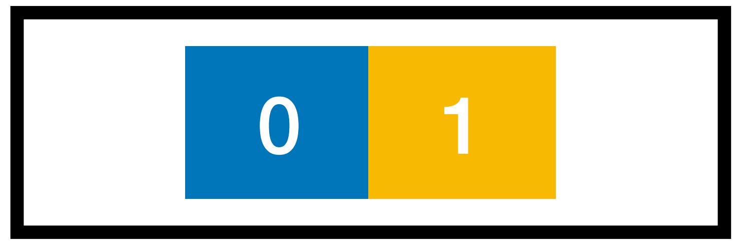


#hits: 1
#misses: 1

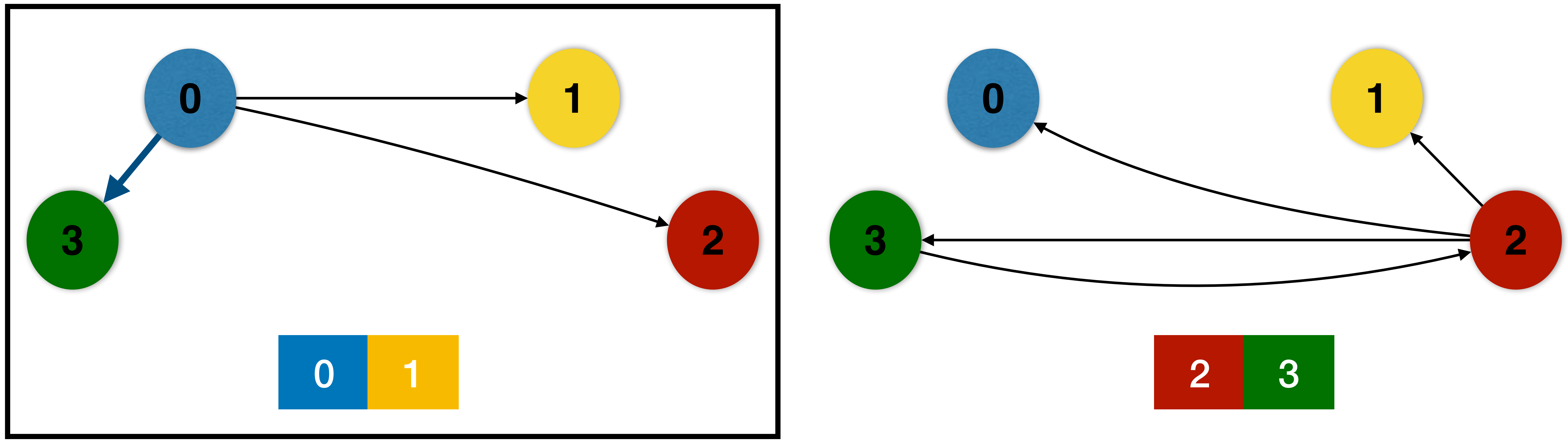


Graph Processing

Cache

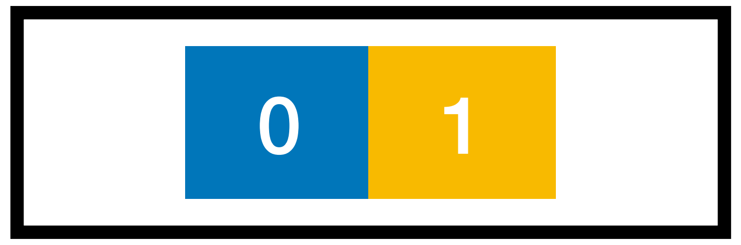


#hits: 2
#misses: 1



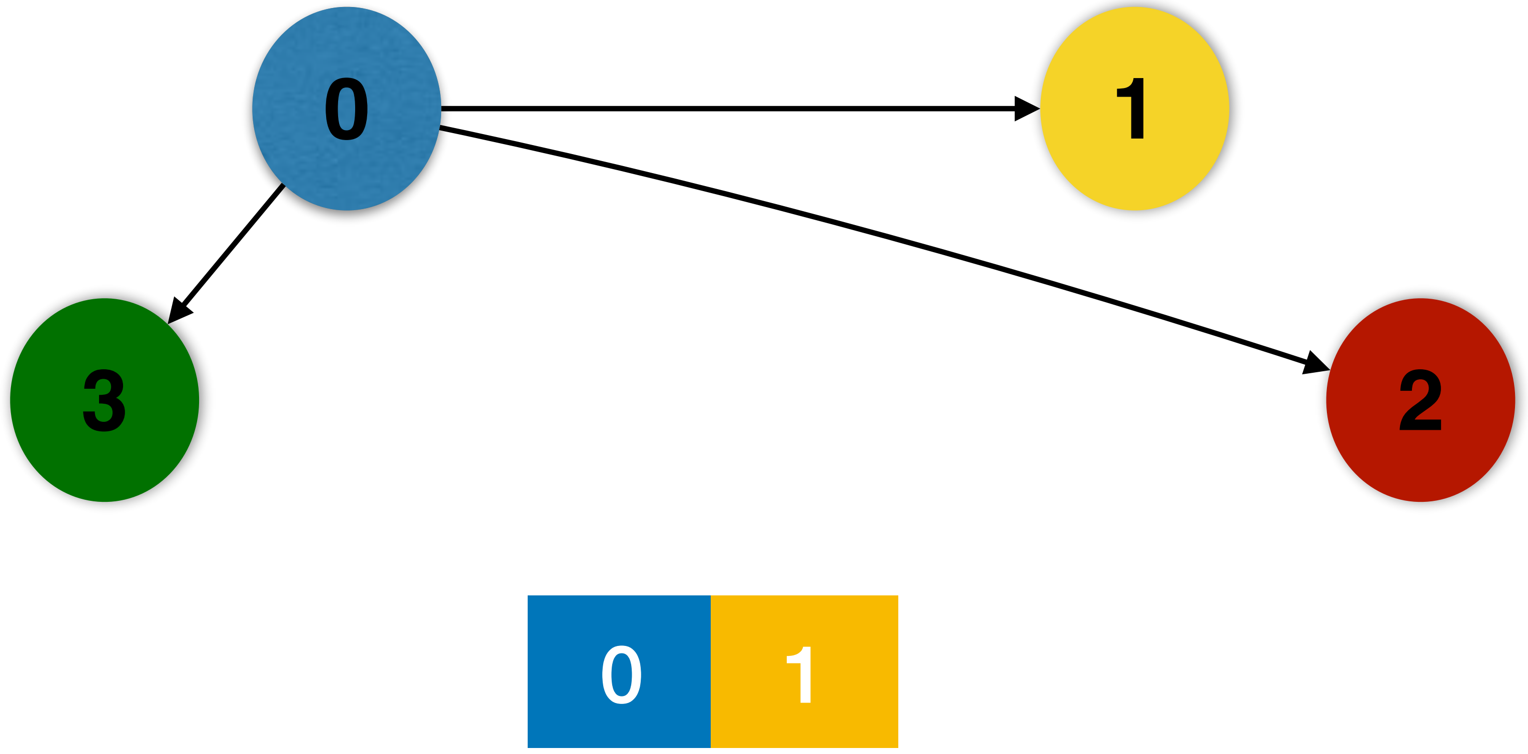
Graph Processing

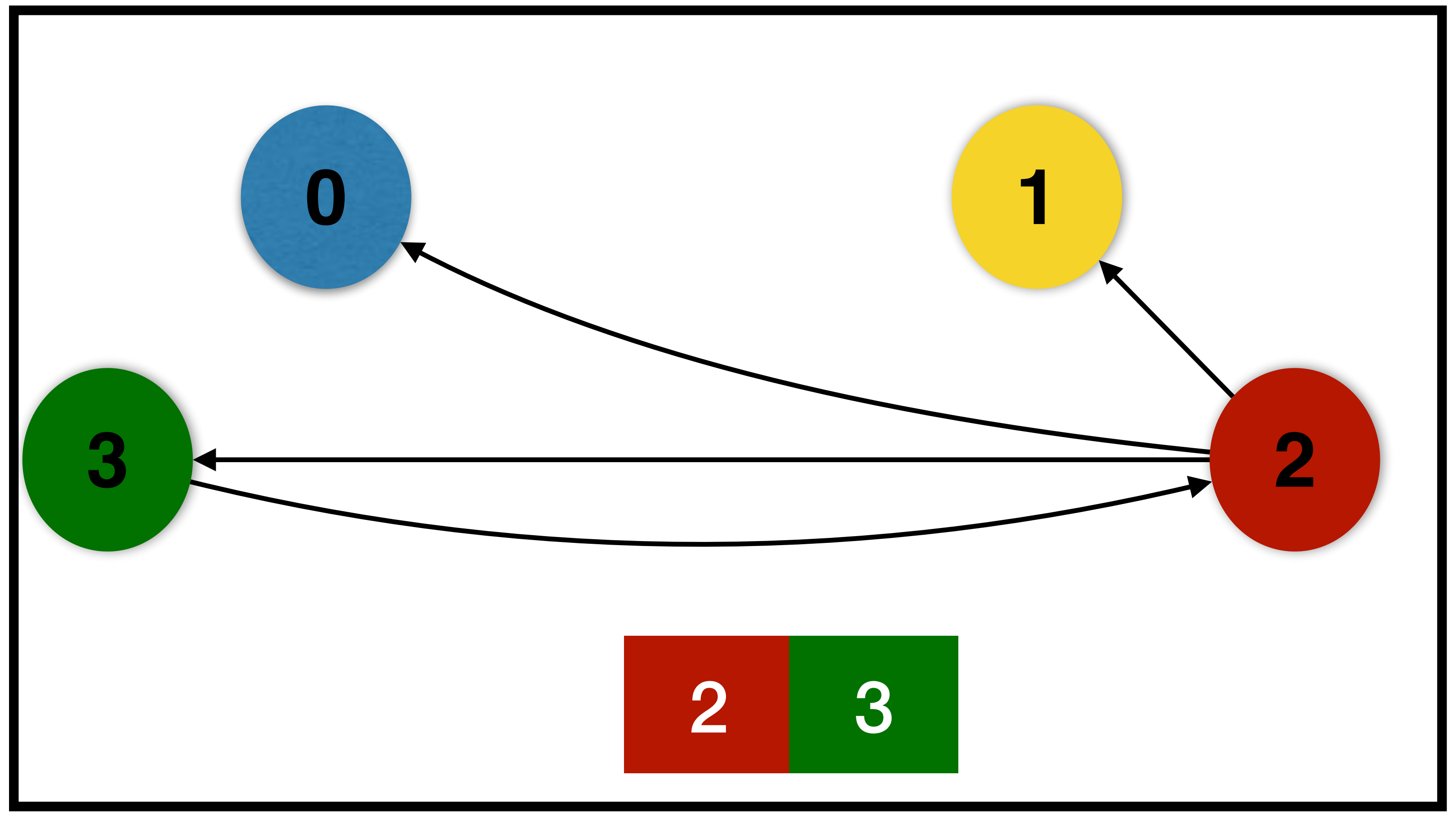
Cache



#hits: 2

#misses: 1





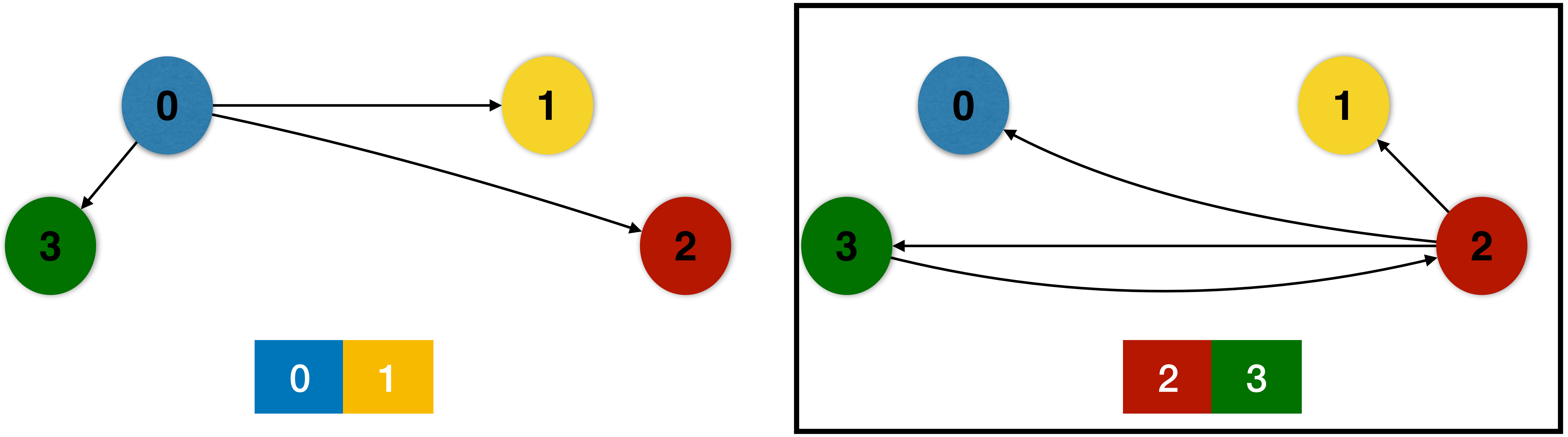
Graph Processing

Cache

0	1

#hits: 2

#misses: 1



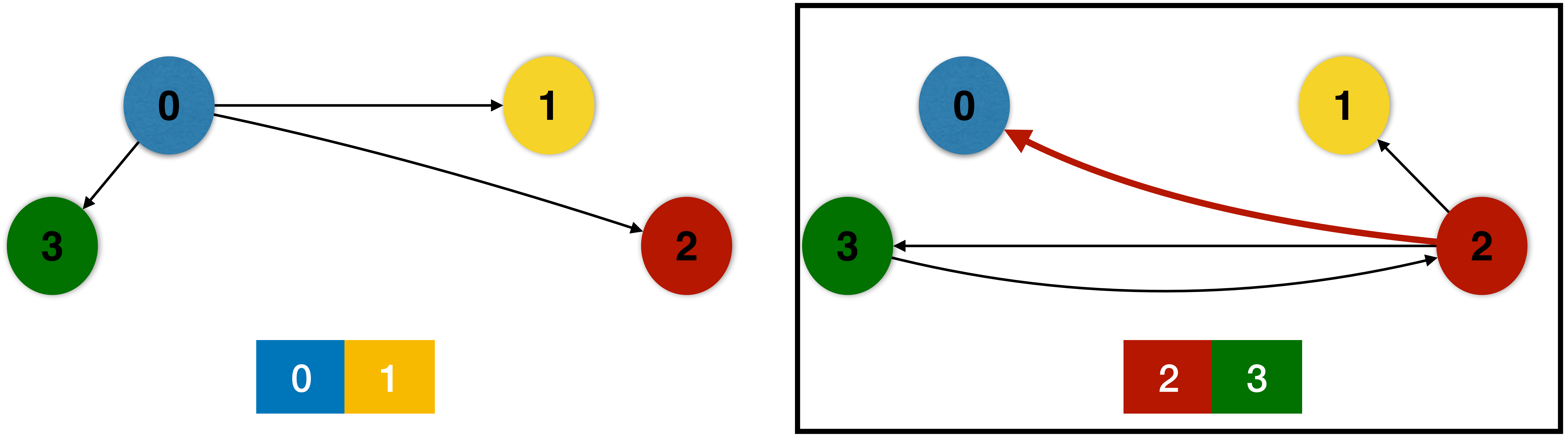
Graph Processing

Cache

0	1

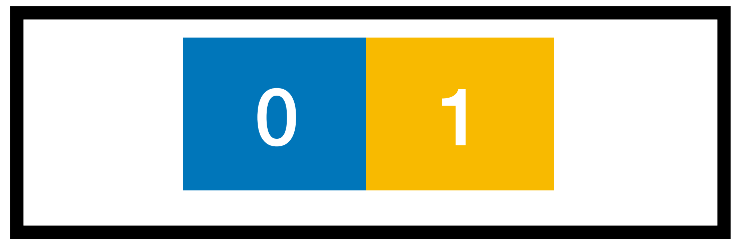
#hits: 2

#misses: 1



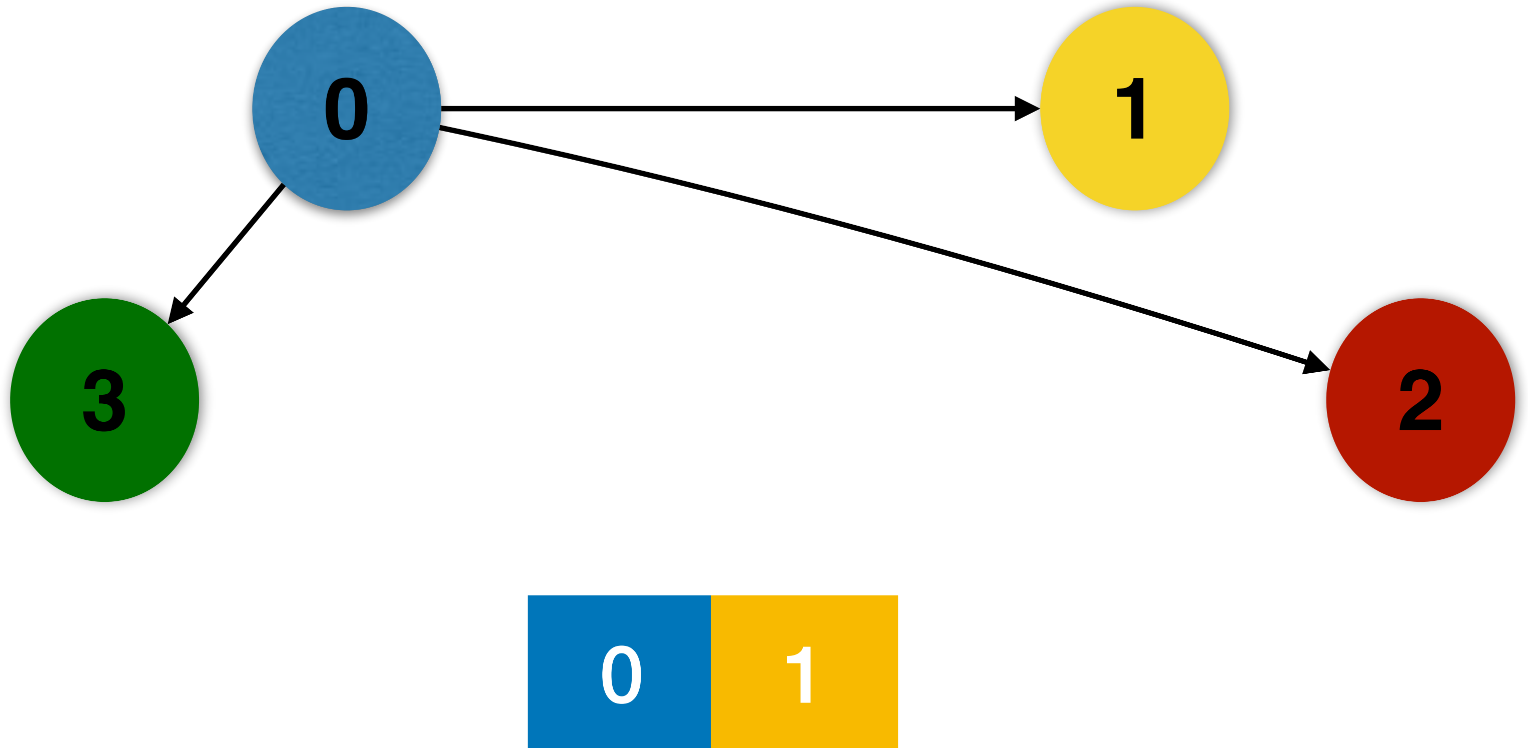
Graph Processing

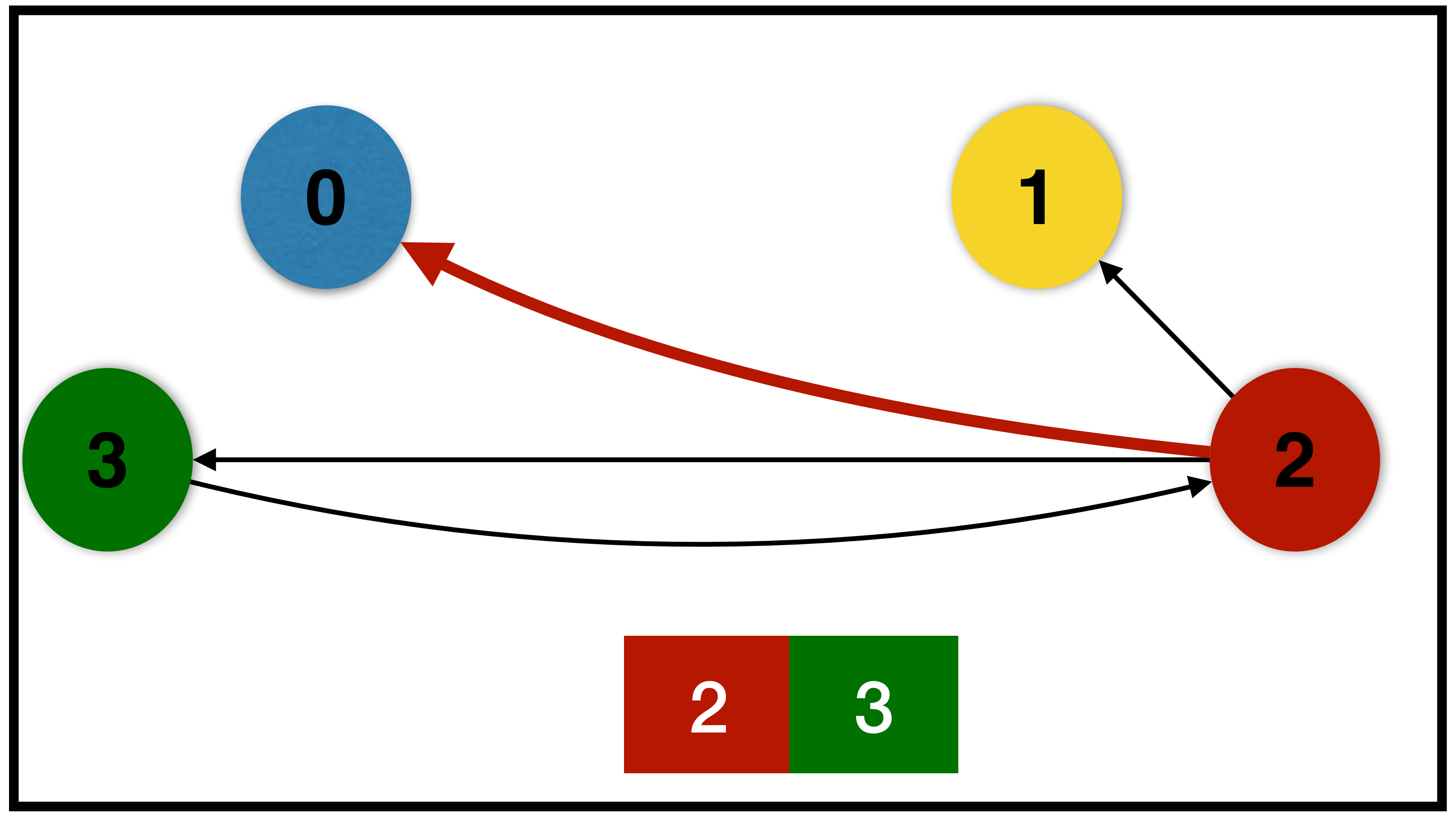
Cache



#hits: 2

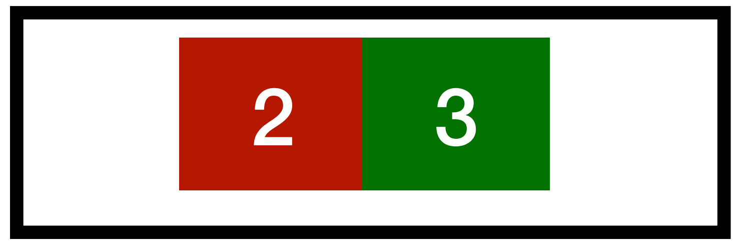
#misses: 1





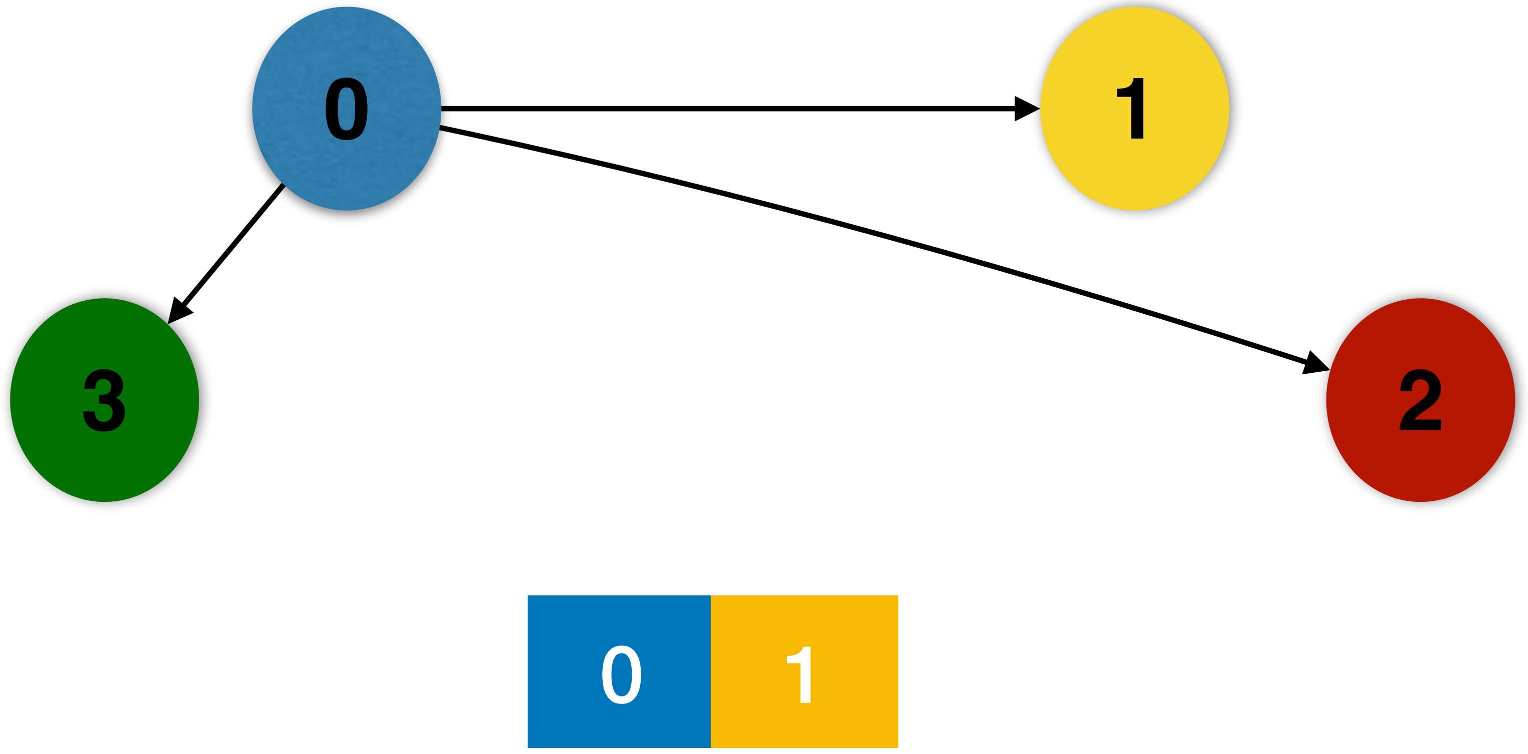
Graph Processing

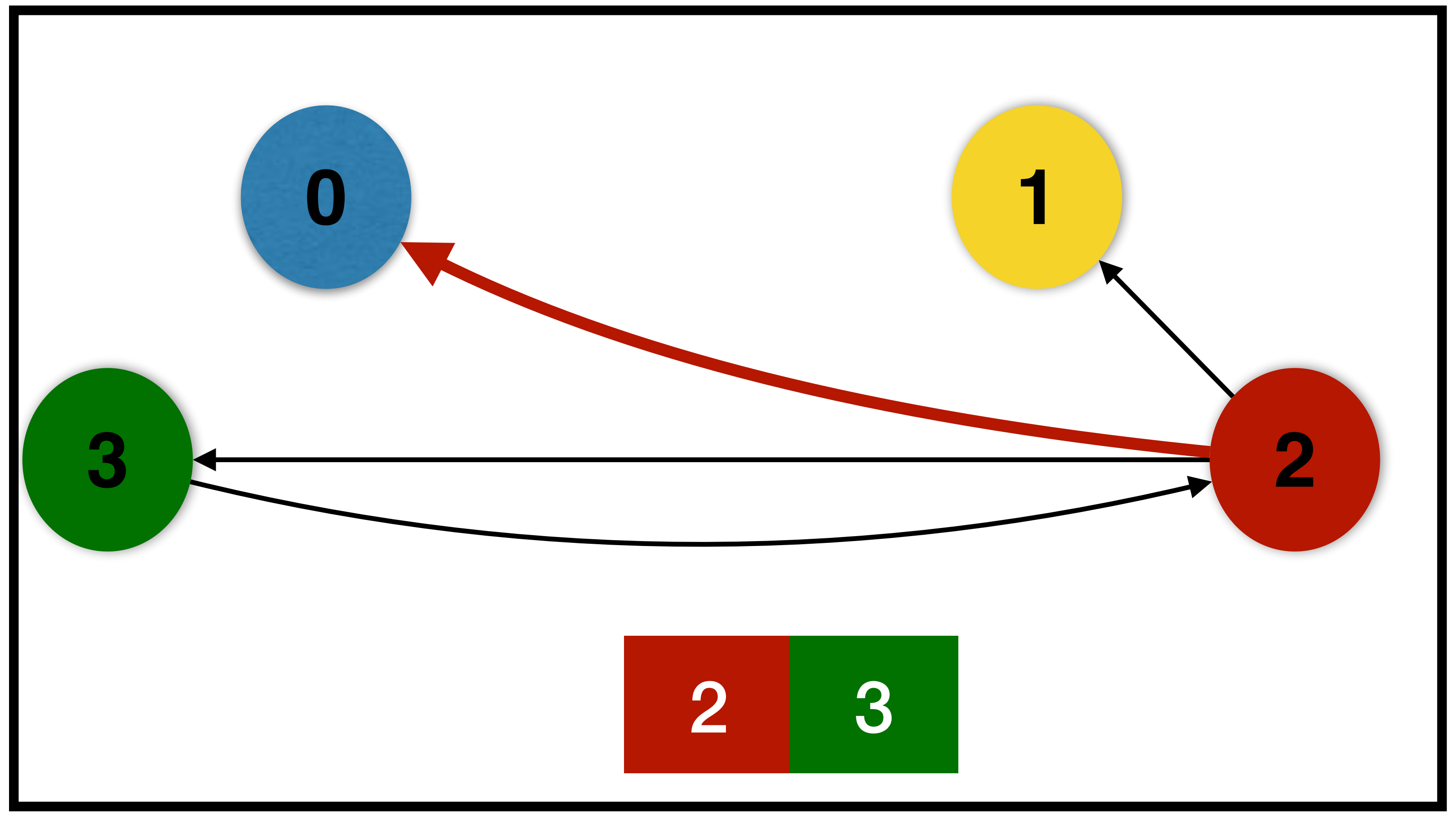
Cache



#hits: 2

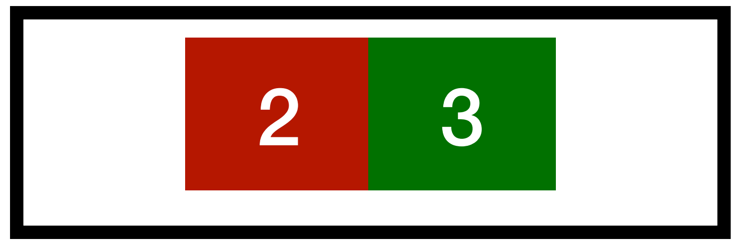
#misses: 2





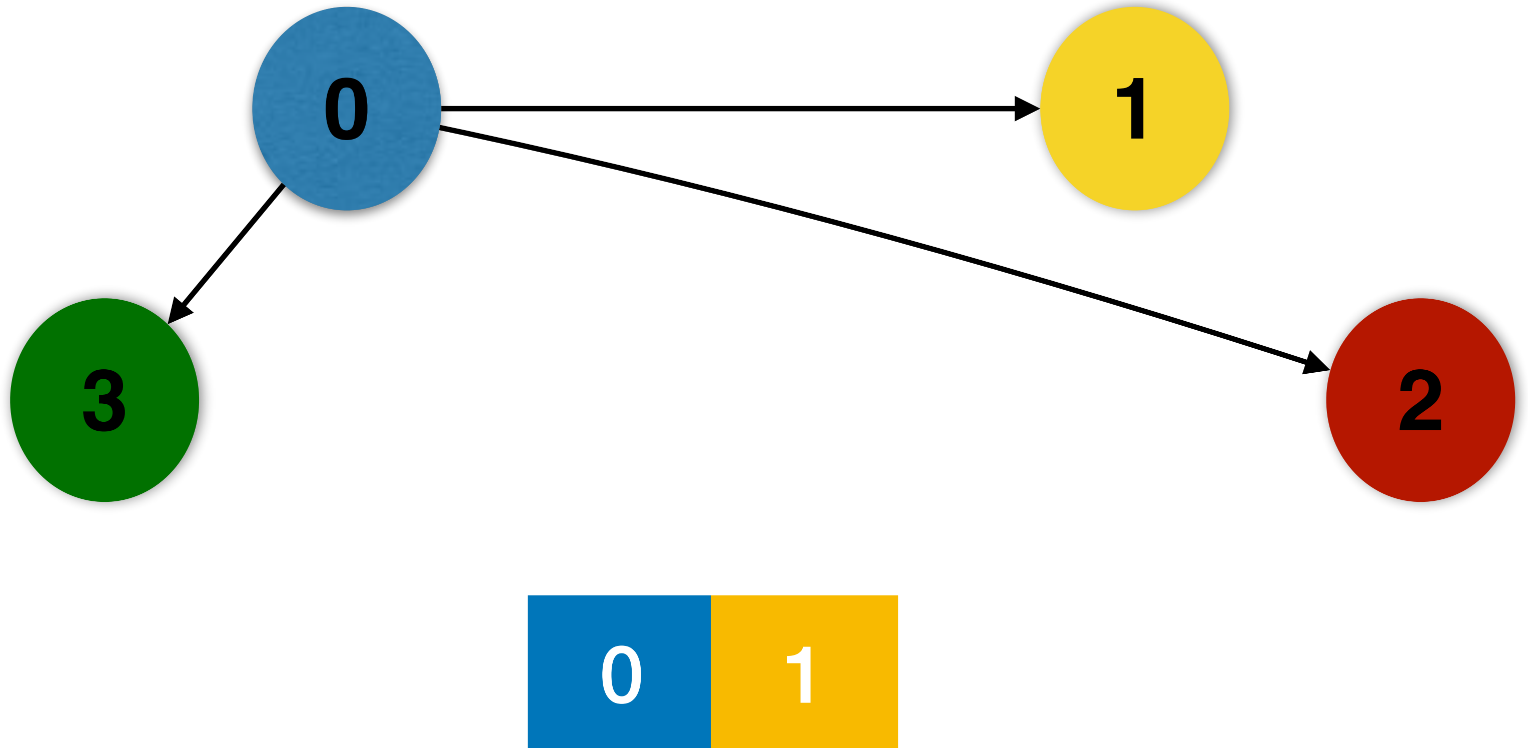
Graph Processing

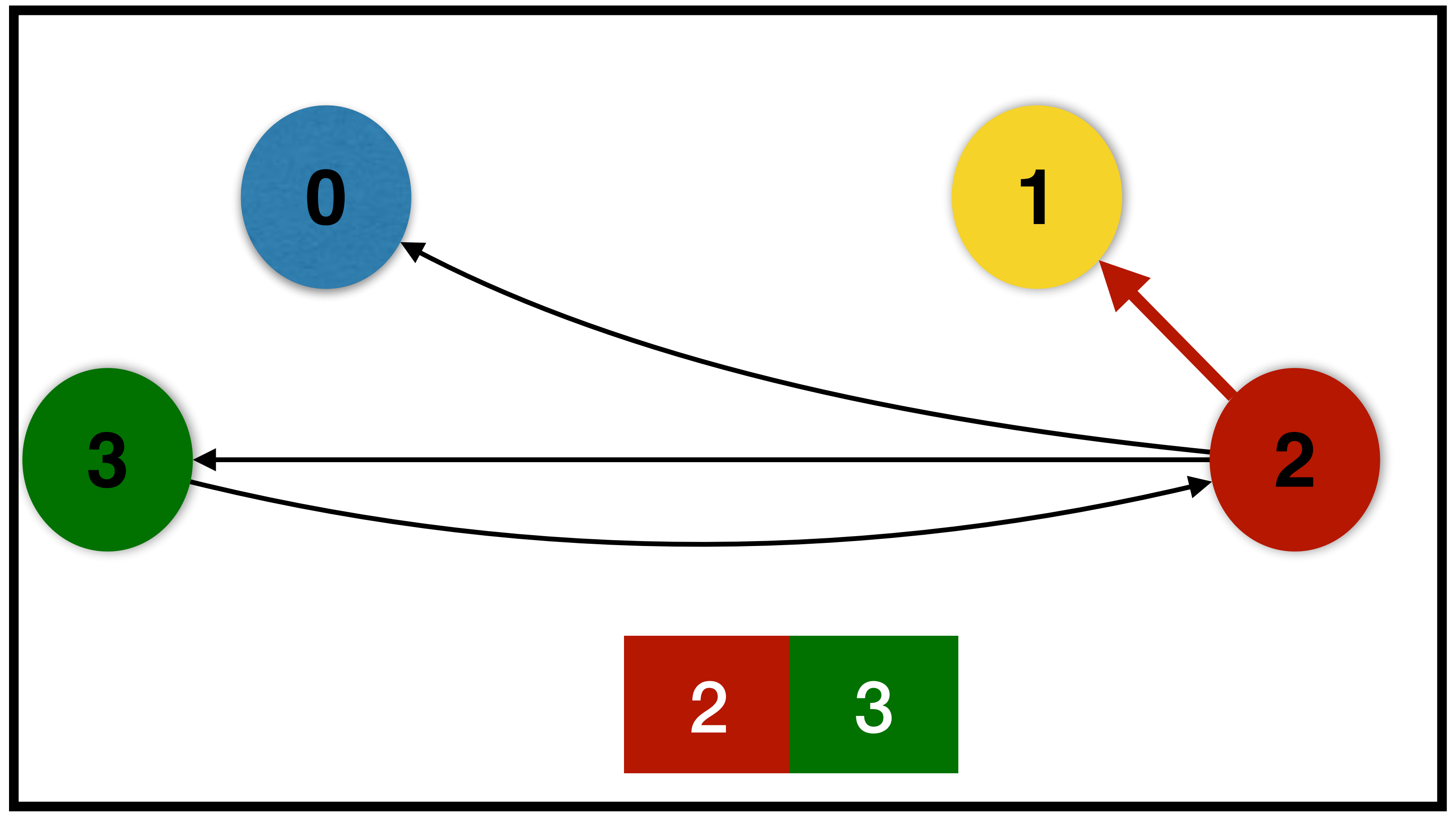
Cache



#hits: 3

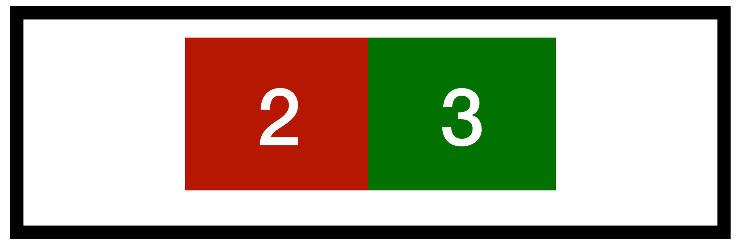
#misses: 2





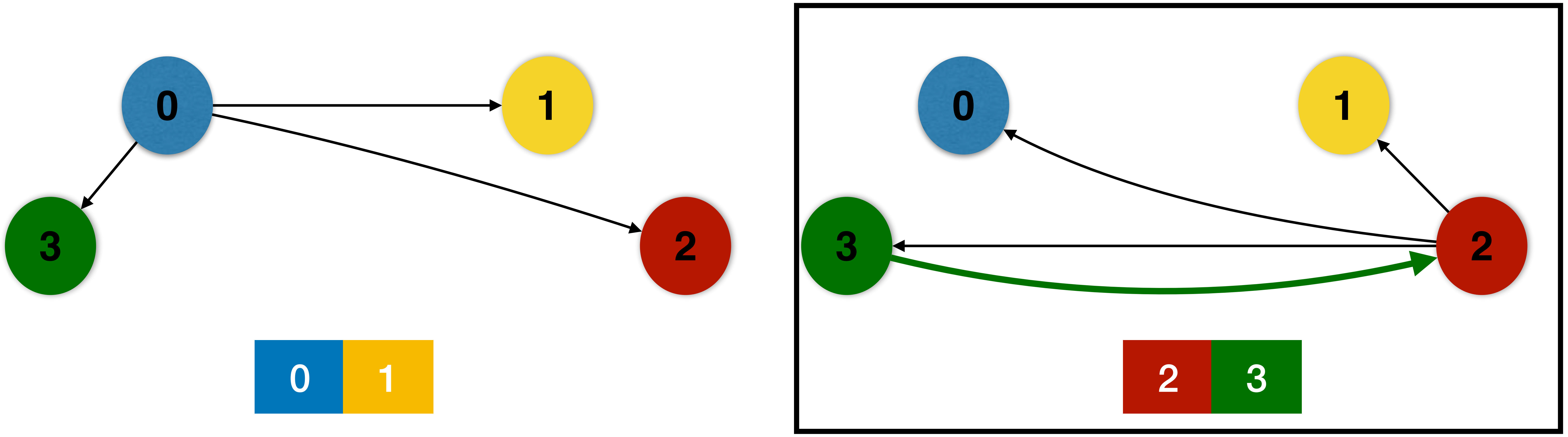
Graph Processing

Cache



#hits: 4

#misses: 2



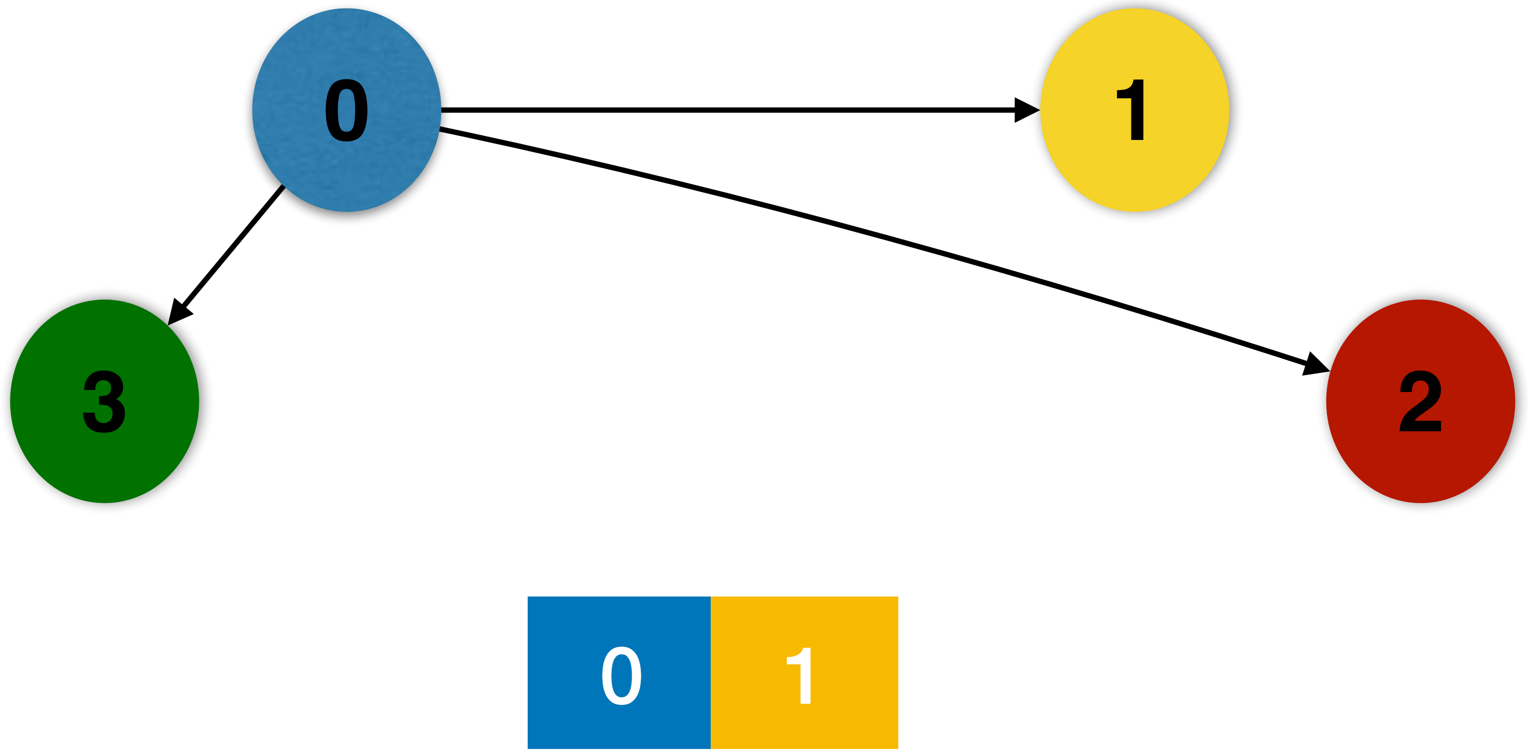
Graph Processing

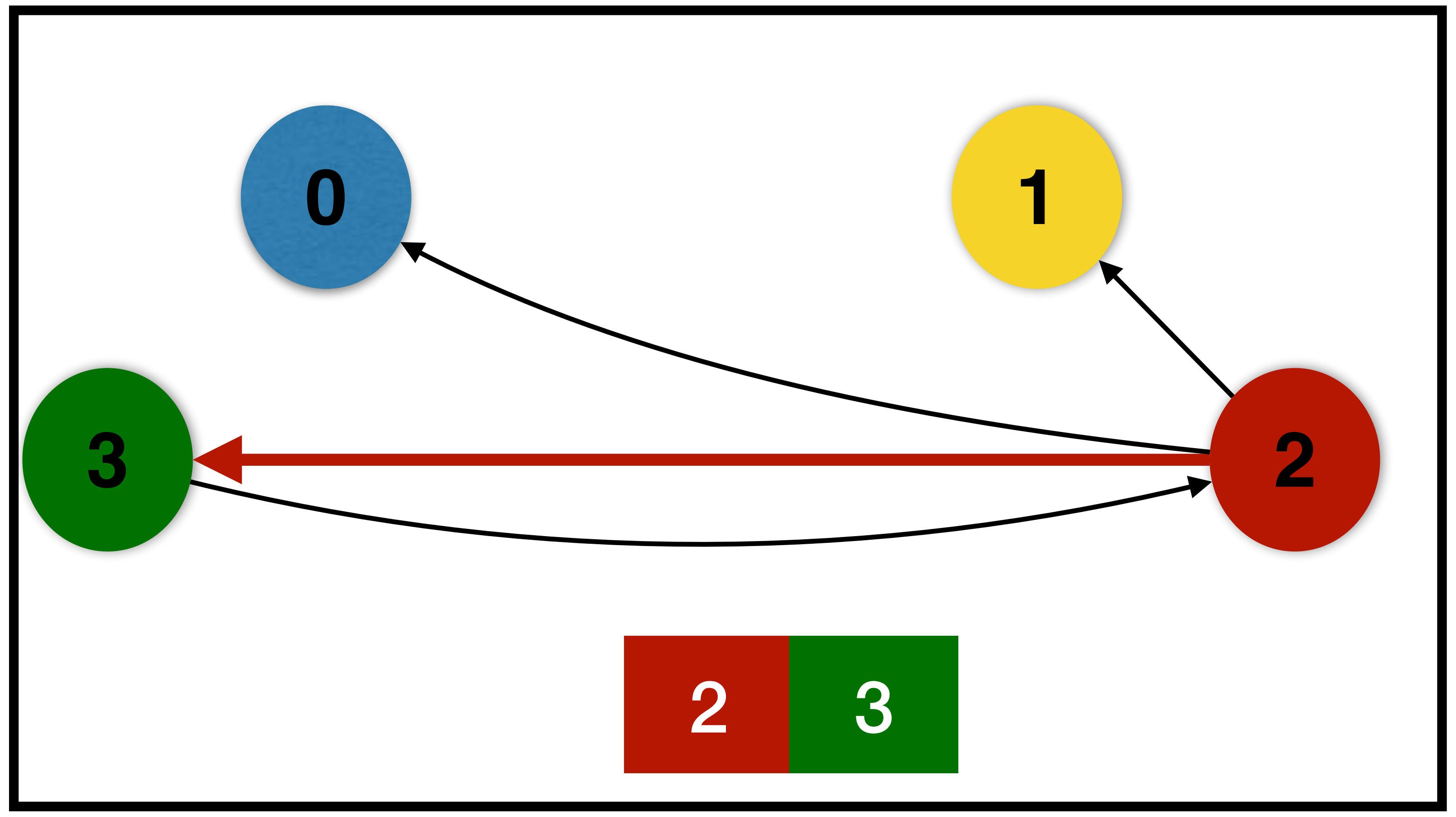
Cache

2	3

#hits: 5

#misses: 2





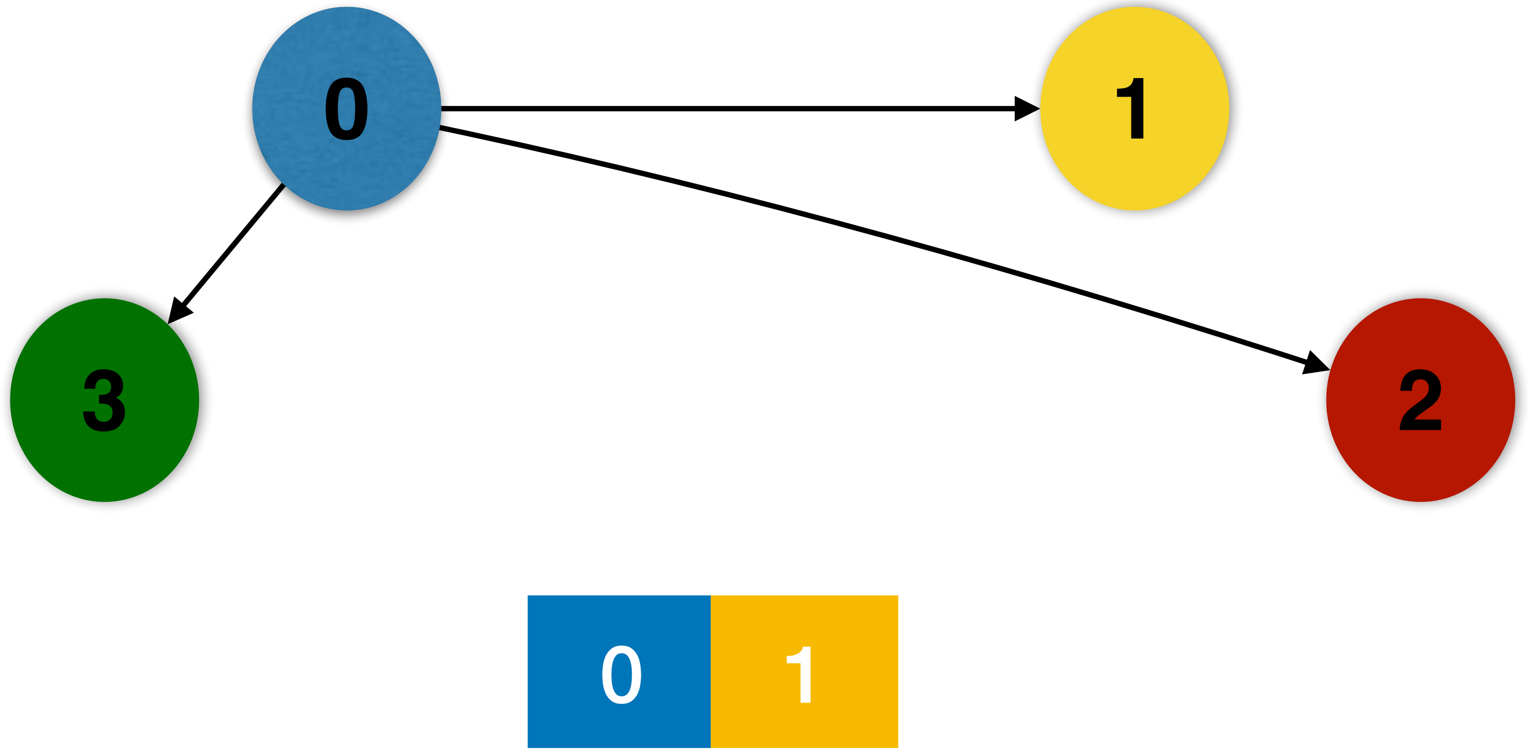
Graph Processing

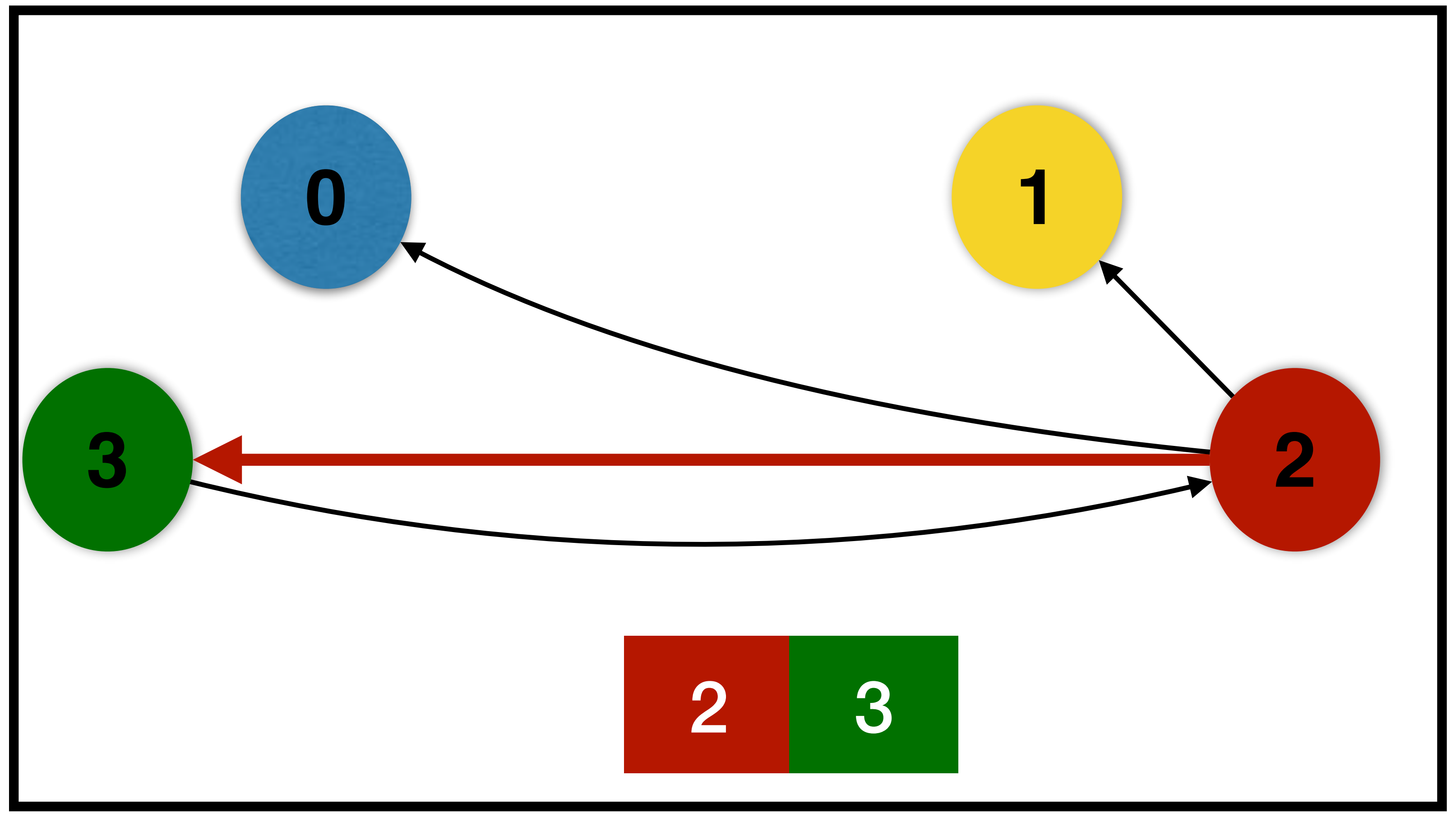
Cache

2	3

Only have 2 misses

#hits: 5
#misses: 2





Graph Processing

Cache

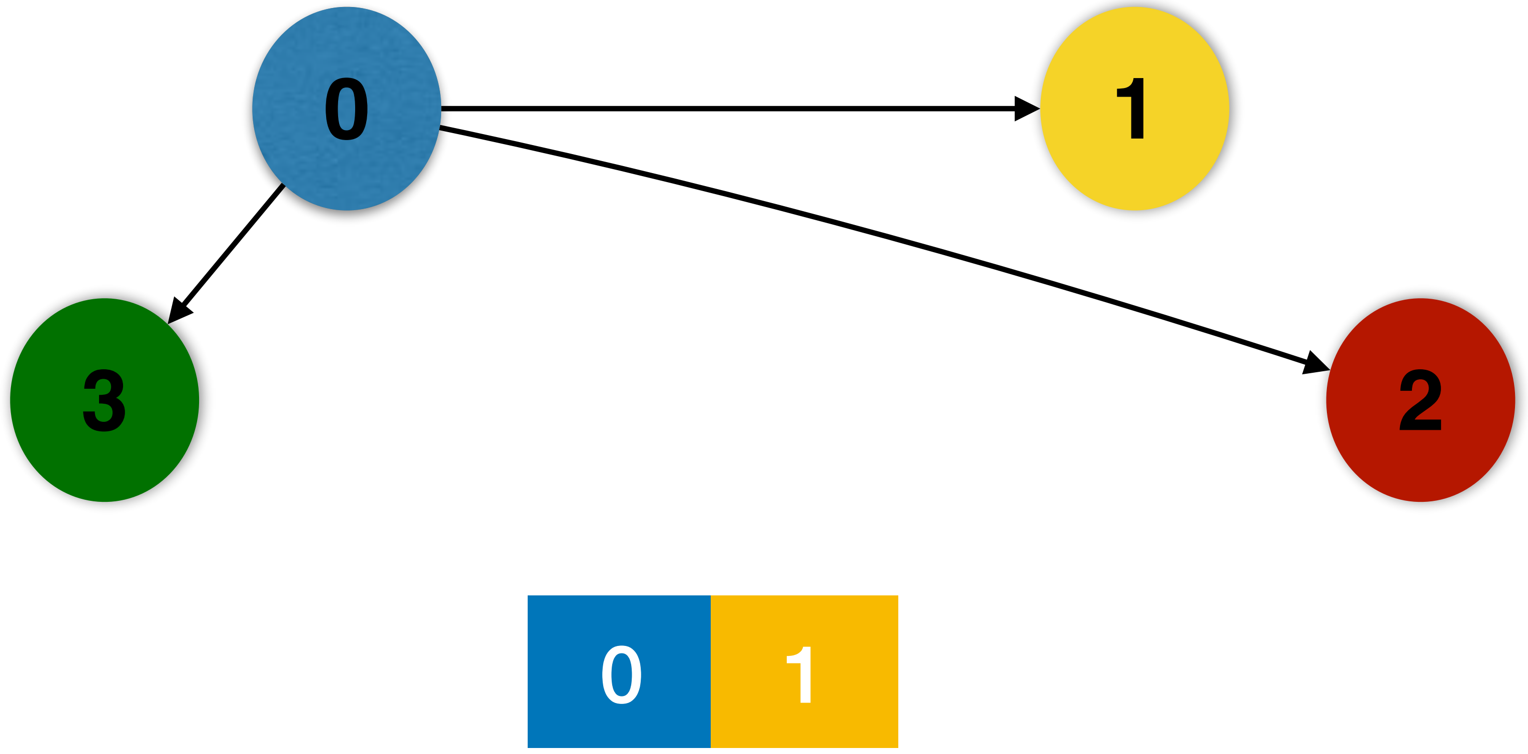
2	3

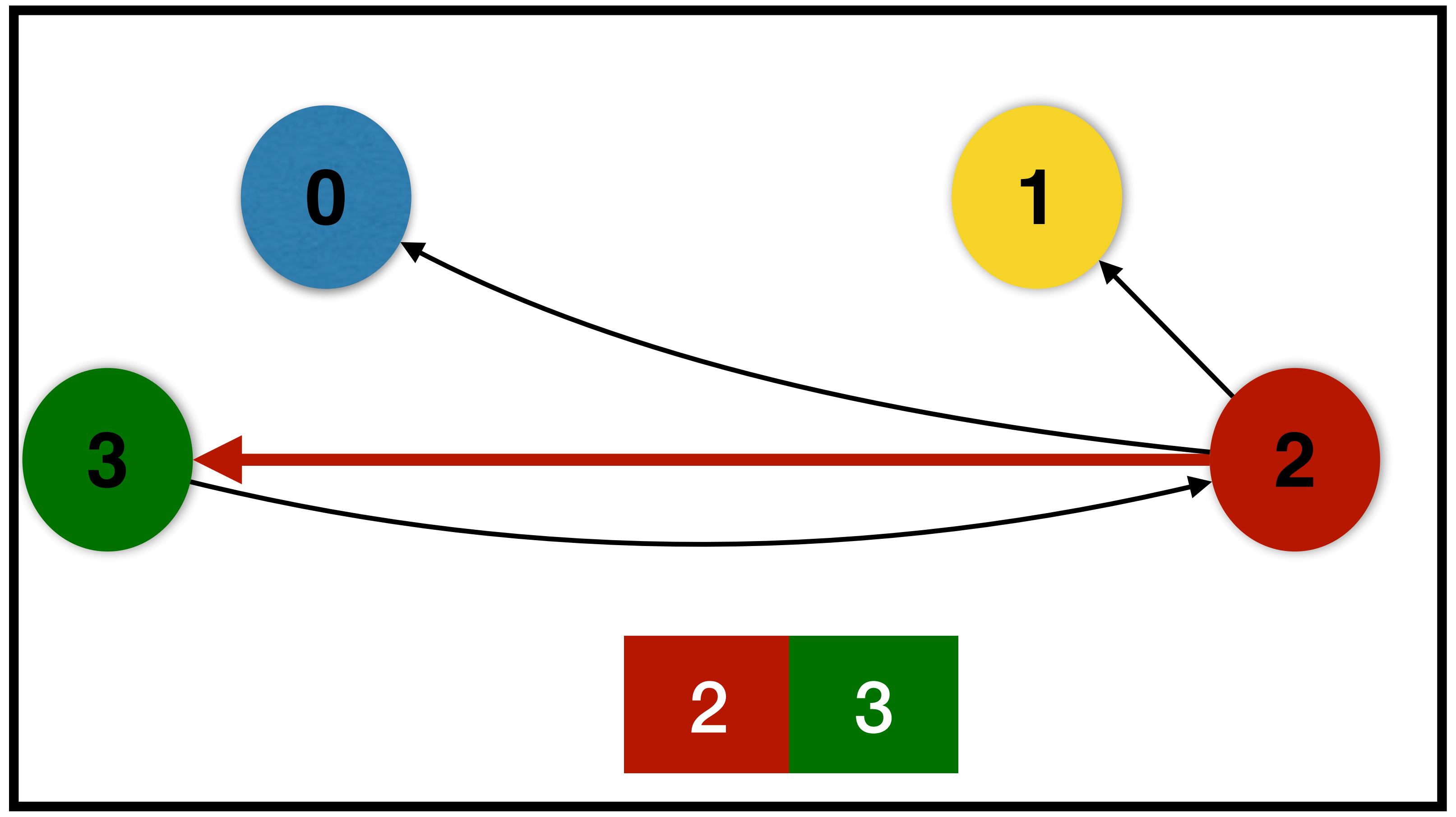
Better than Frequency-based Reordering

#hits: 4
#misses: 3

←

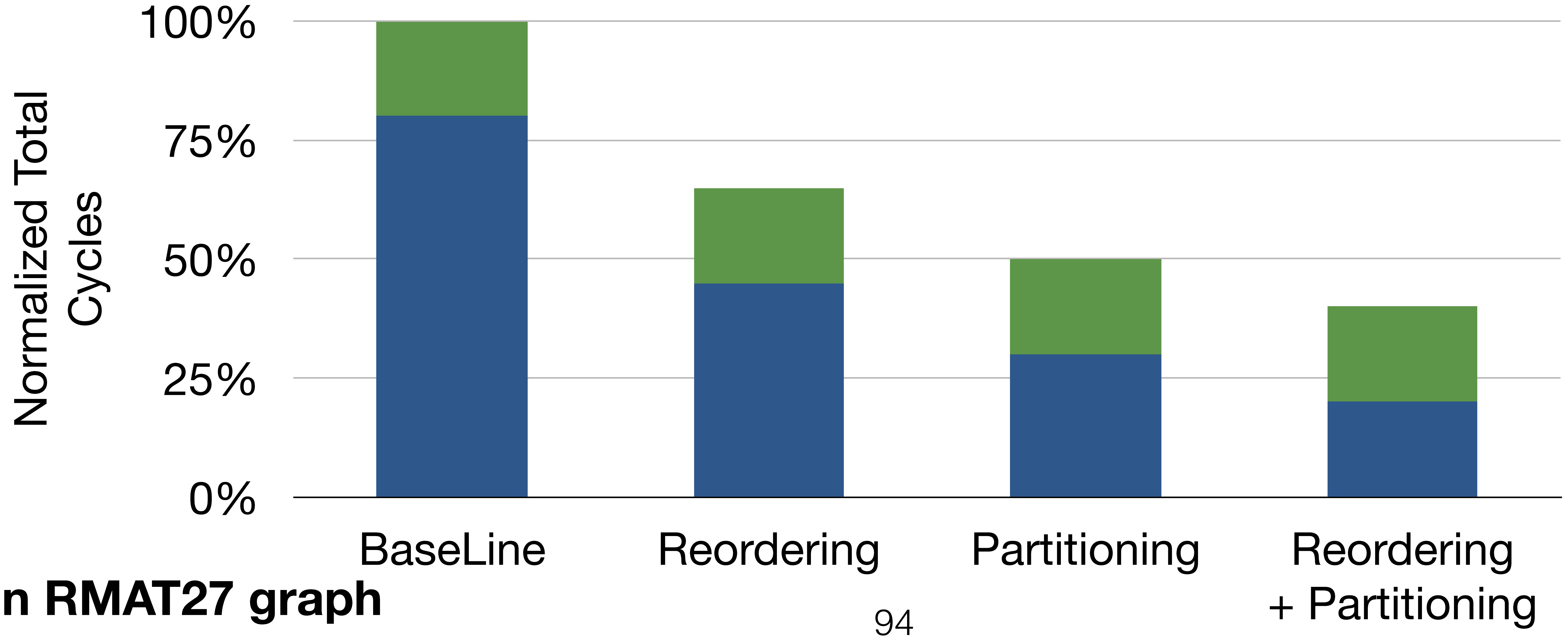
#hits: 5
#misses: 2





PageRank

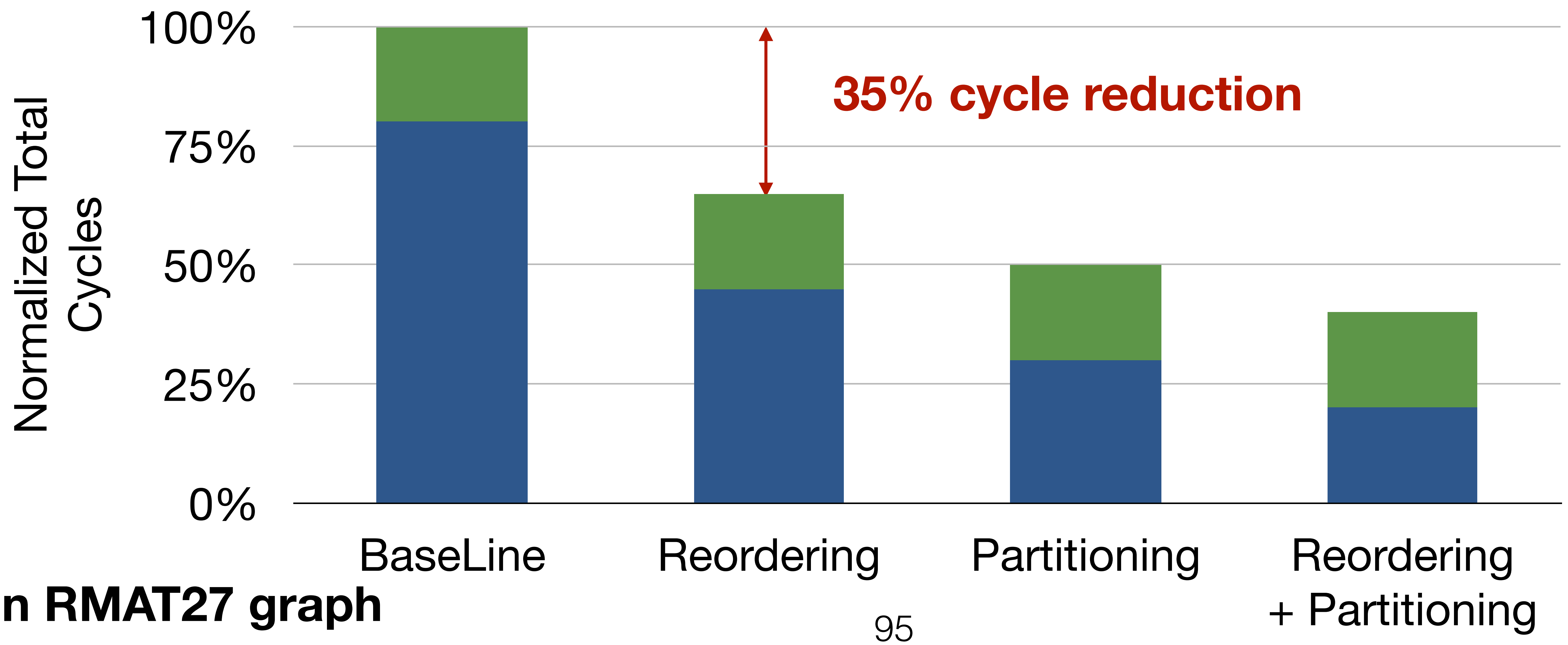
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



on RMAT27 graph

PageRank

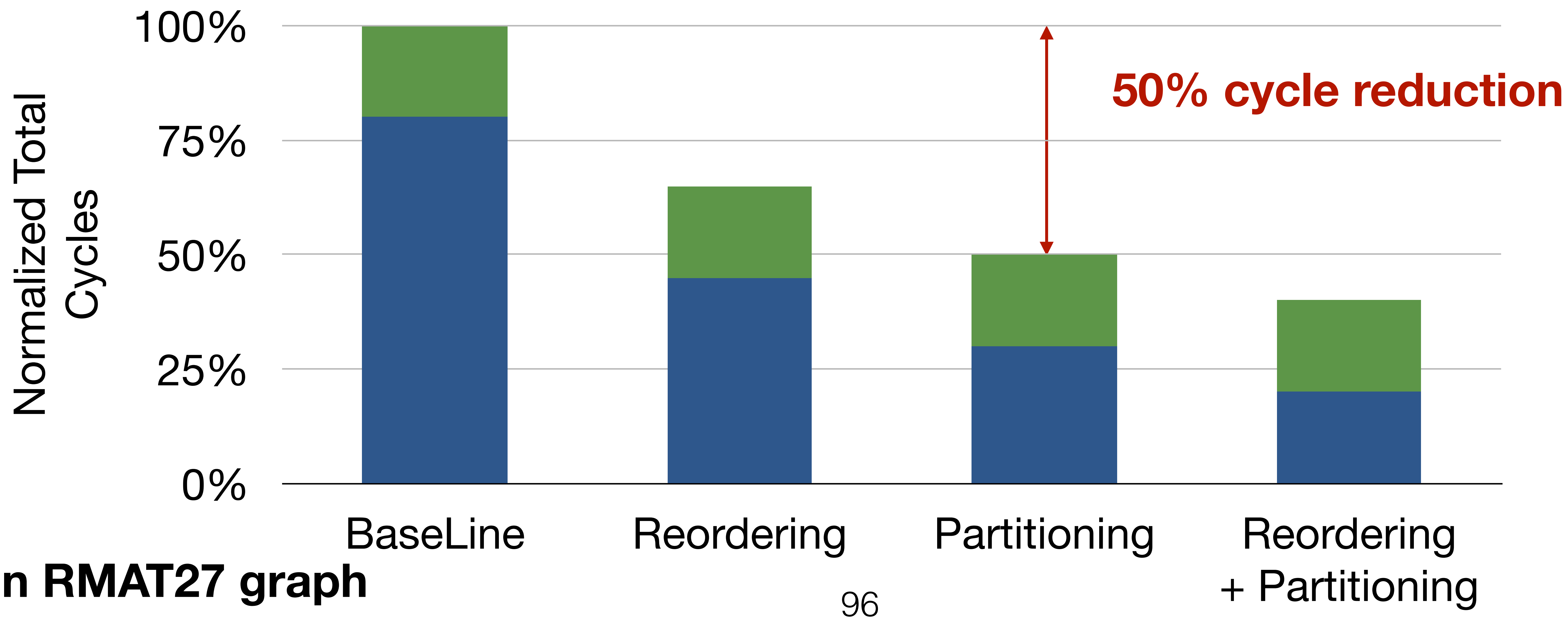
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
   for node : graph.vertices
       newRanks[node] = baseScore + damping*newRanks[node];
   swap ranks and newRanks
```



on RMAT27 graph

PageRank

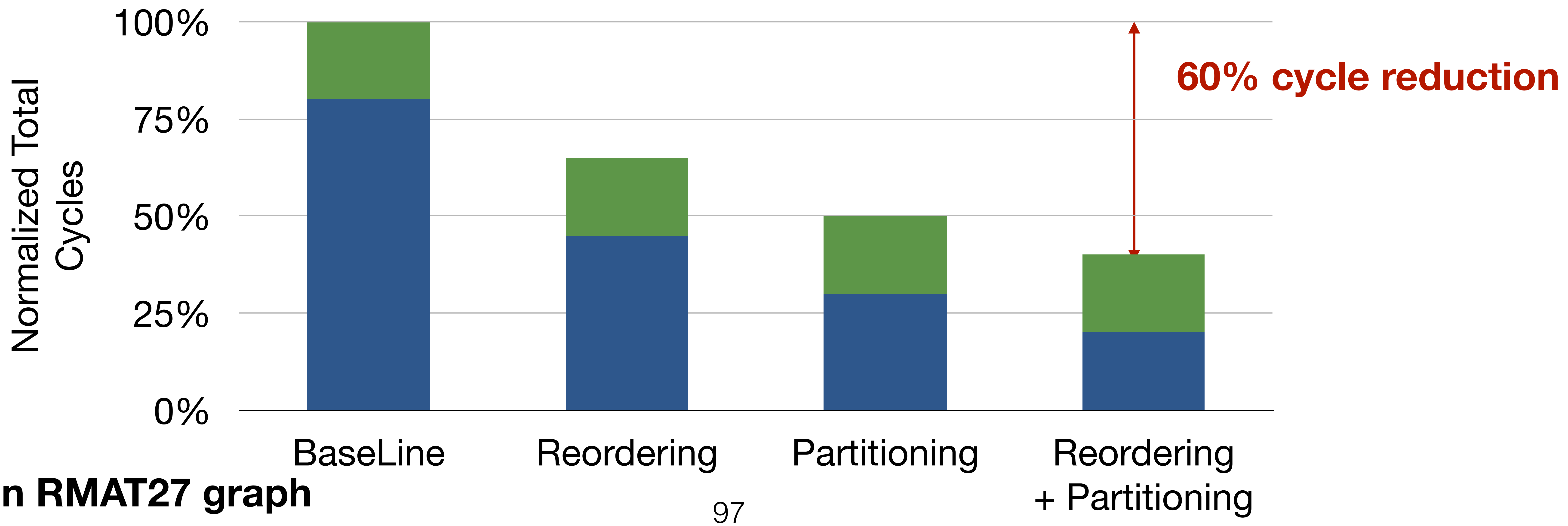
```
while …
  for node : graph.vertices
     for ngh : graph.getInNeighbors(node)
         newRanks[node] += ranks[ngh]/outDegree[ngh];
 for node : graph.vertices
     newRanks[node] = baseScore + damping*newRanks[node];
 swap ranks and newRanks
```



on RMAT27 graph

PageRank

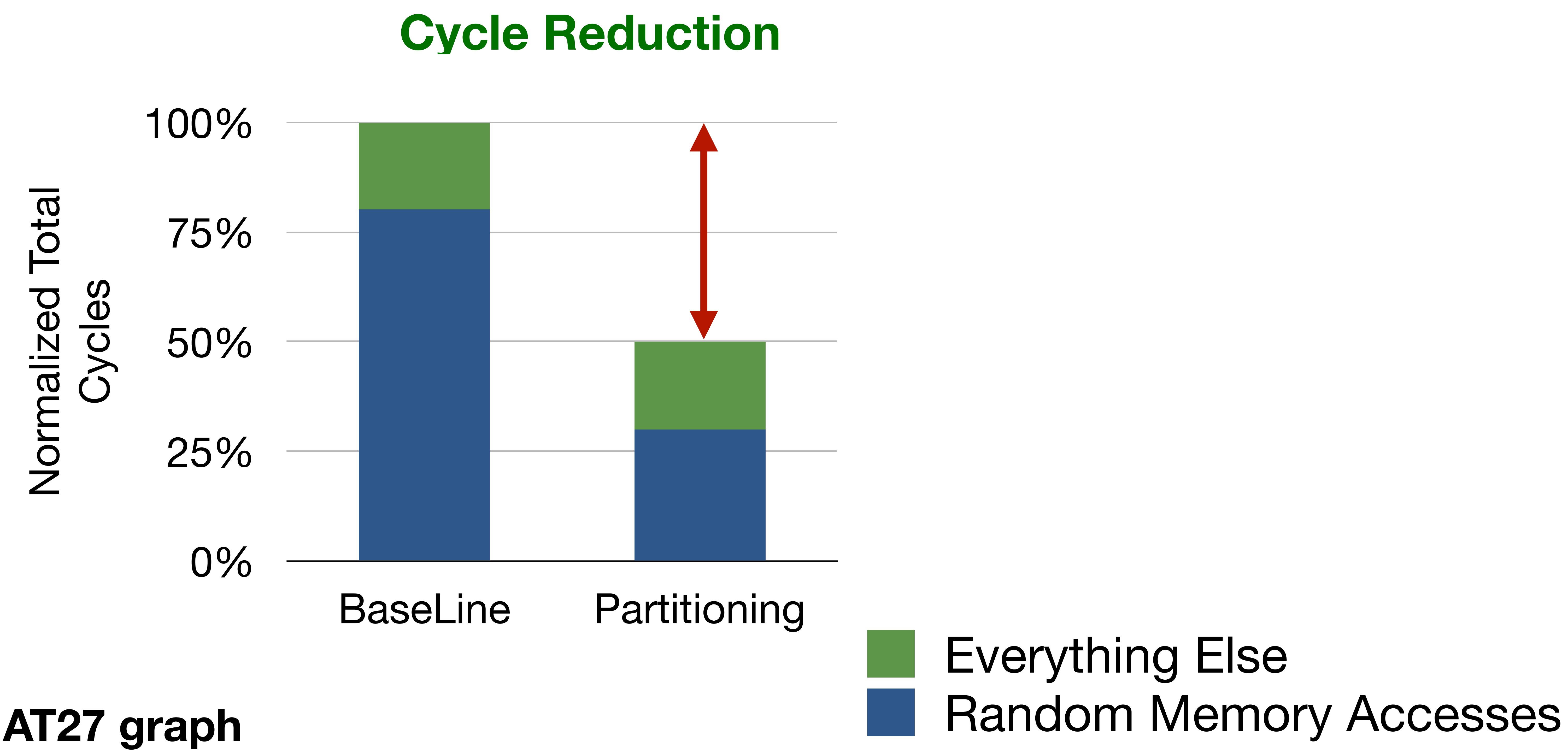
```
while …
    for node : graph.vertices
        for ngh : graph.getInNeighbors(node)
            newRanks[node] += ranks[ngh]/outDegree[ngh];
    for node : graph.vertices
        newRanks[node] = baseScore + damping*newRanks[node];
    swap ranks and newRanks
```



on RMAT27 graph

Impact of Partitioning

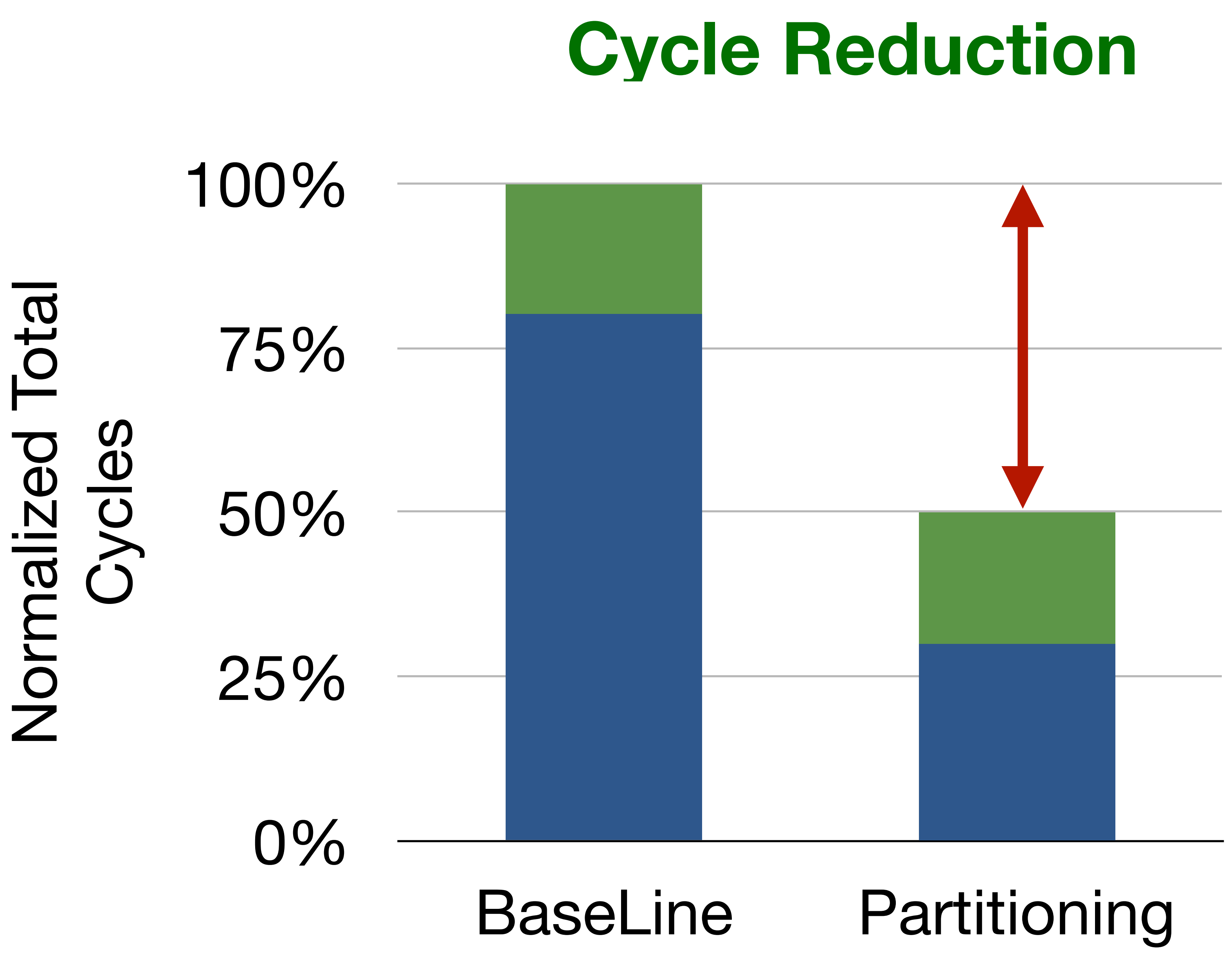
PageRank



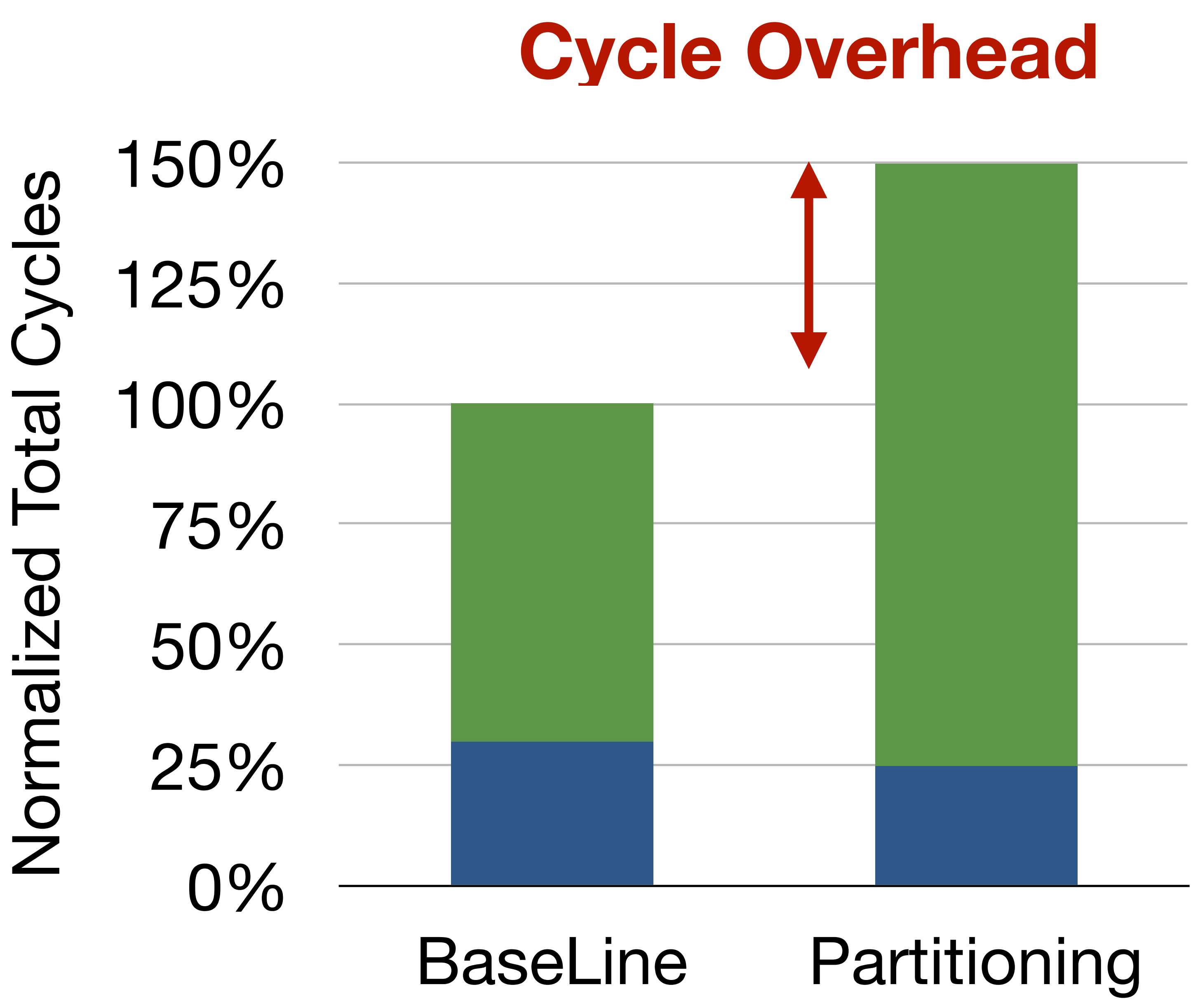
on RMAT27 graph

Impact of Partitioning

PageRank



Breadth First Search



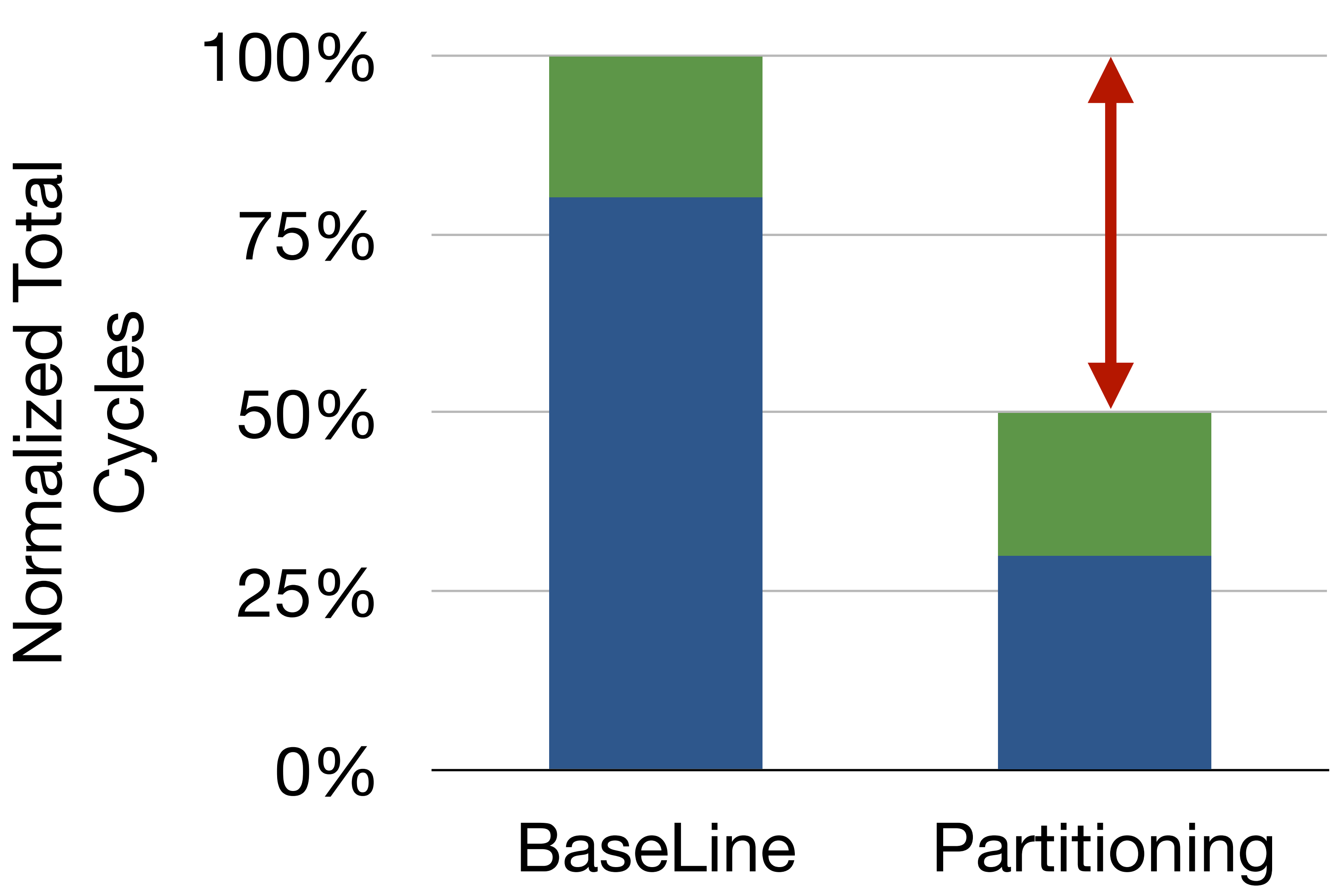
🟩 Everything Else
🟦 Random Memory Accesses

on RMAT27 graph

Impact of Partitioning

PageRank

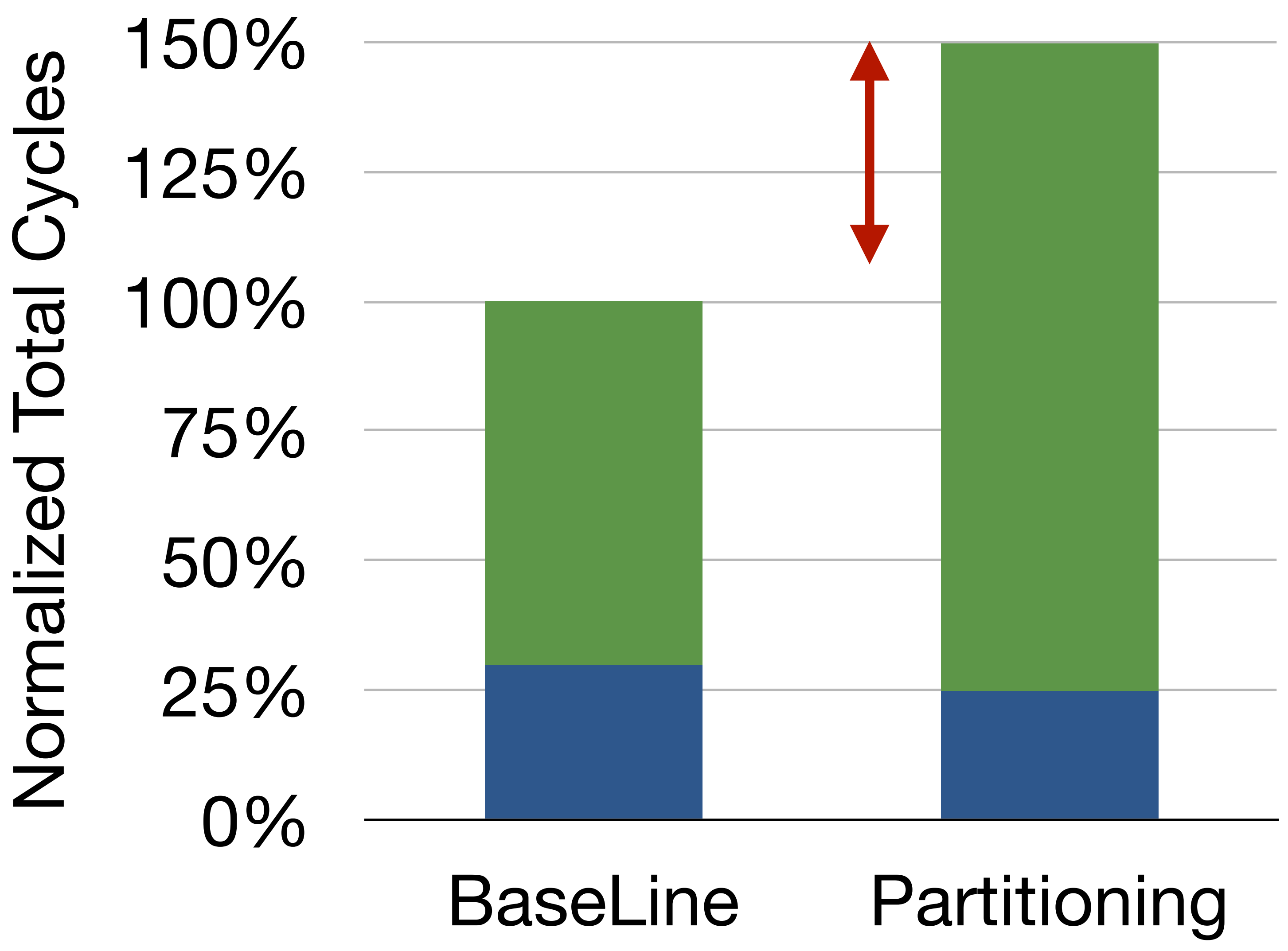
Cycle Reduction



Optimizations Don’t Always Work across Different Algorithms and Data

Breadth First Search

Cycle Overhead



🟩 Everything Else

🟦 Random Memory Accesses

on RMAT27 graph

Graph Computations have Variety

Data

Social Networks, Web Graphs, Road Networks, Engineering Meshes, Transaction Graphs, Network Traffic Graphs, Email Networks, Similarity Graphs, …

×

Algorithms

Breadth-first search, betweenness centrality, Bellman-Ford, Delta-stepping, collaborative filtering, Page Rank, Page Rank Delta, connected components, k-core decomposition, triangle counting, local clustering, structural clustering minimum spanning forest, eccentricity estimation, graph coloring, k-truss decomposition, nuclei decomposition, biconnectivity, set cover, maximum flow, butterfly counting, strongly connected components, graph partitioning, RDF queries, random walks, point-to-point shortest paths, A* search, low-diameter decomposition, densest subgraph, multi-source BFS, maximal independent set, maximal matching, etc…

×

Hardware

CPU, GPU, KNL, Distributed Environment, FPGA, HammerBlade, Symphony,…

Outline

Hardware Utilization

Programming System to Handle Variety in Data and Algorithms

Variety in Hardware

Making Caches Work for Graph Analytics (BigData17)
Zhang, et al.

GraphIt: a High-Performance Graph DSL (OOPSLA18)
Zhang, et al.

Optimizing Ordered Graph Algorithms with GraphIt (CGO2020)
Zhang, et al.

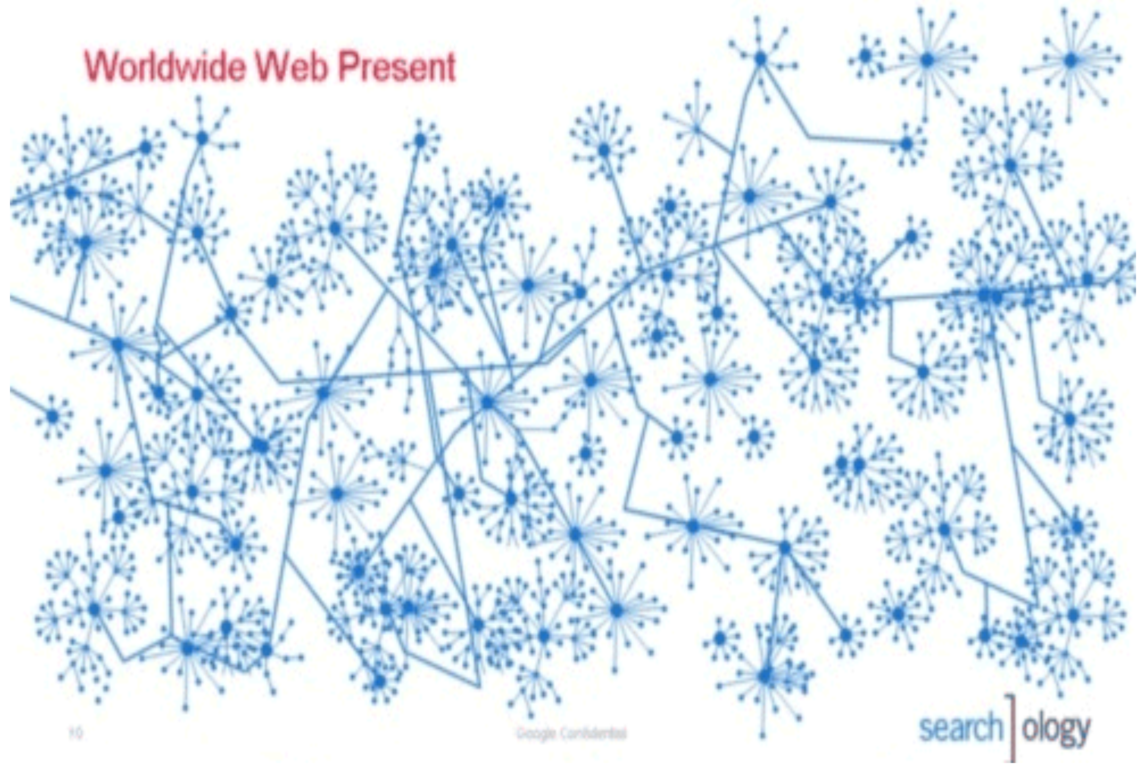
Universal Graph Framework (Under Submission)
Brahmakshatriya, Zhang, et al.

- **Frequency-based Reordering**
- **Cache-aware Partitioning**

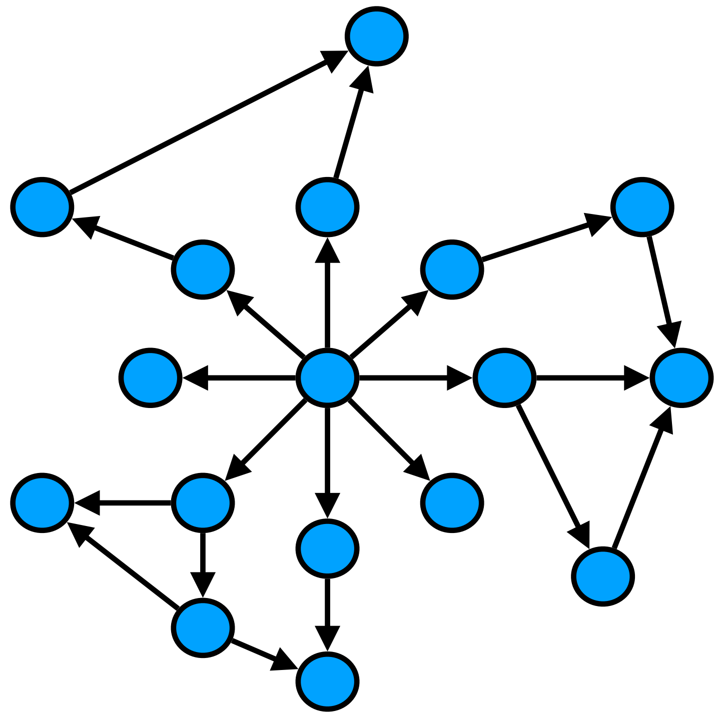
***GraphIt* Compiler and DSL that Decouples**
- **Algorithm**
- **Optimization**
- **Hardware**

for Graph Applications

Power-Law Graphs

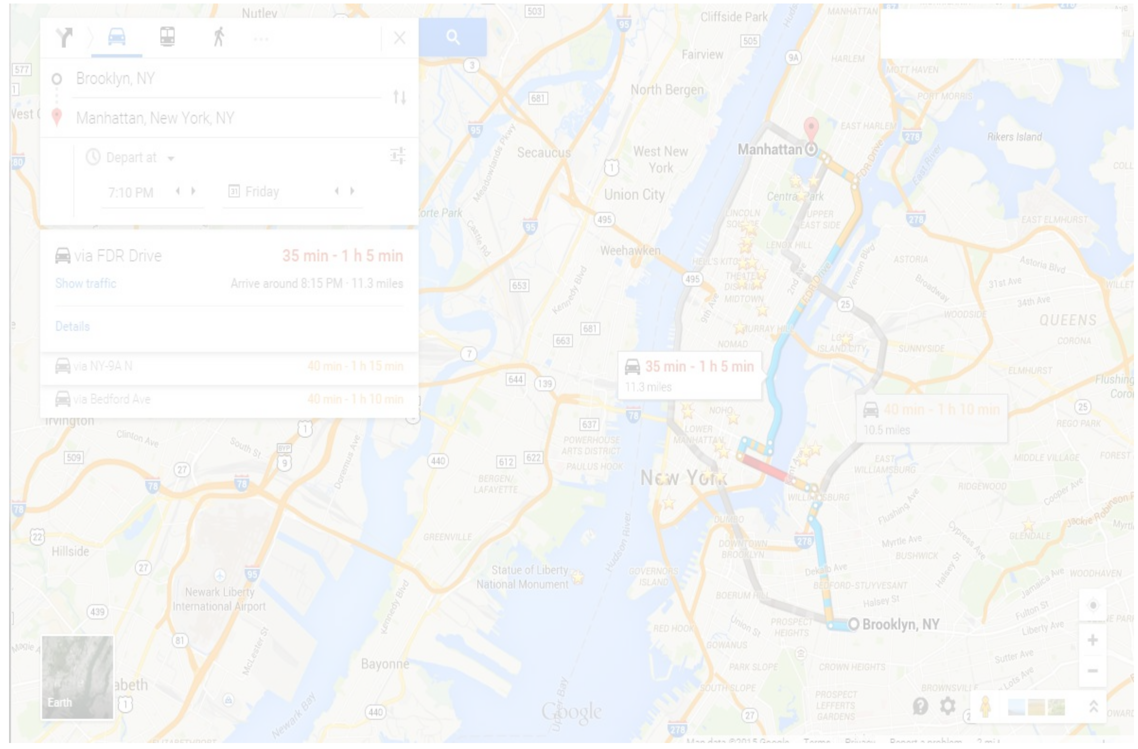


World Wide Web

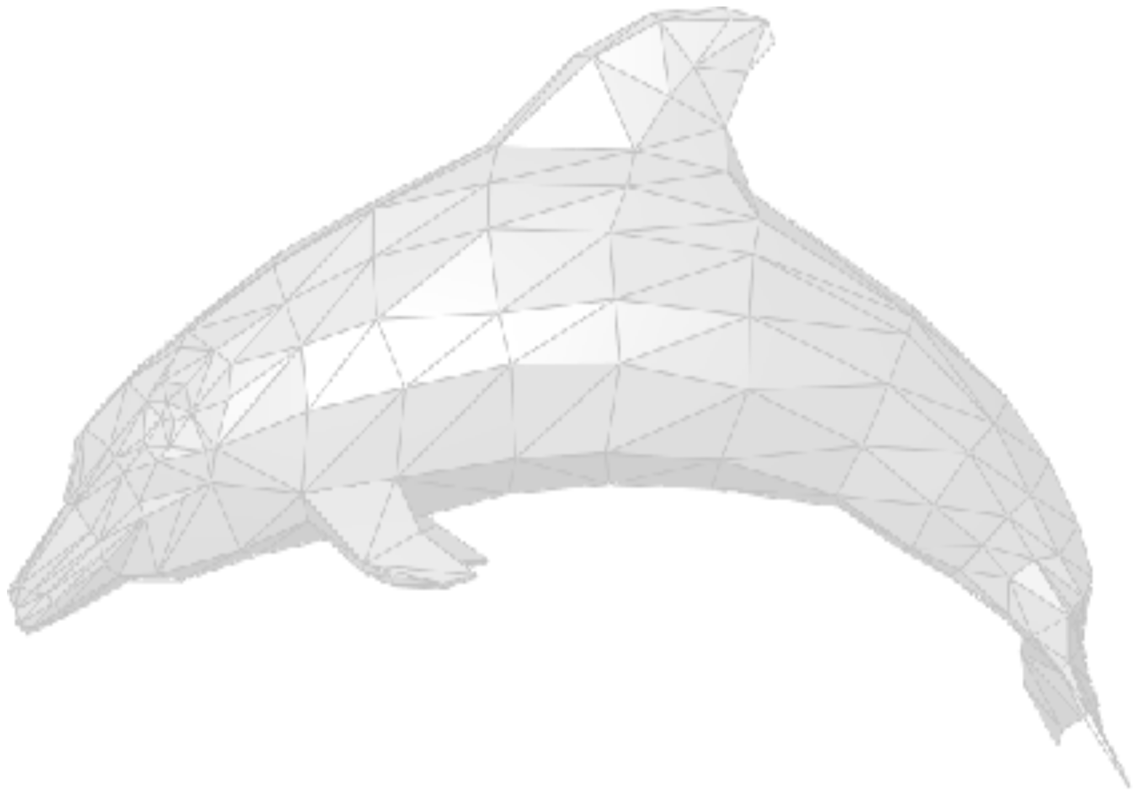


Power-Law Degree Distribution, Small Diameter, Poor Locality

Social Networks



Maps

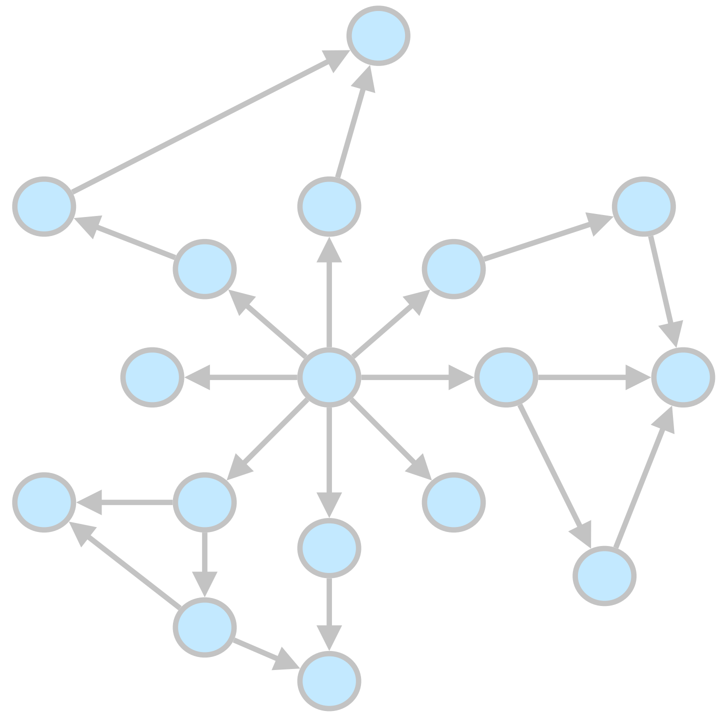


Engineering Meshes

1. http://googlesystem.blogspot.com/2007/05/world-wide-web-as-seen-by-google.html 2. http://www.facebookfever.com/introducing-facebook-new-graph-api-explorer-features/ 3. http://maps.google.com **4. https://en.wikipedia.org/wiki/Polygon_mesh#/media/File:Dolphin_triangle_mesh.png**

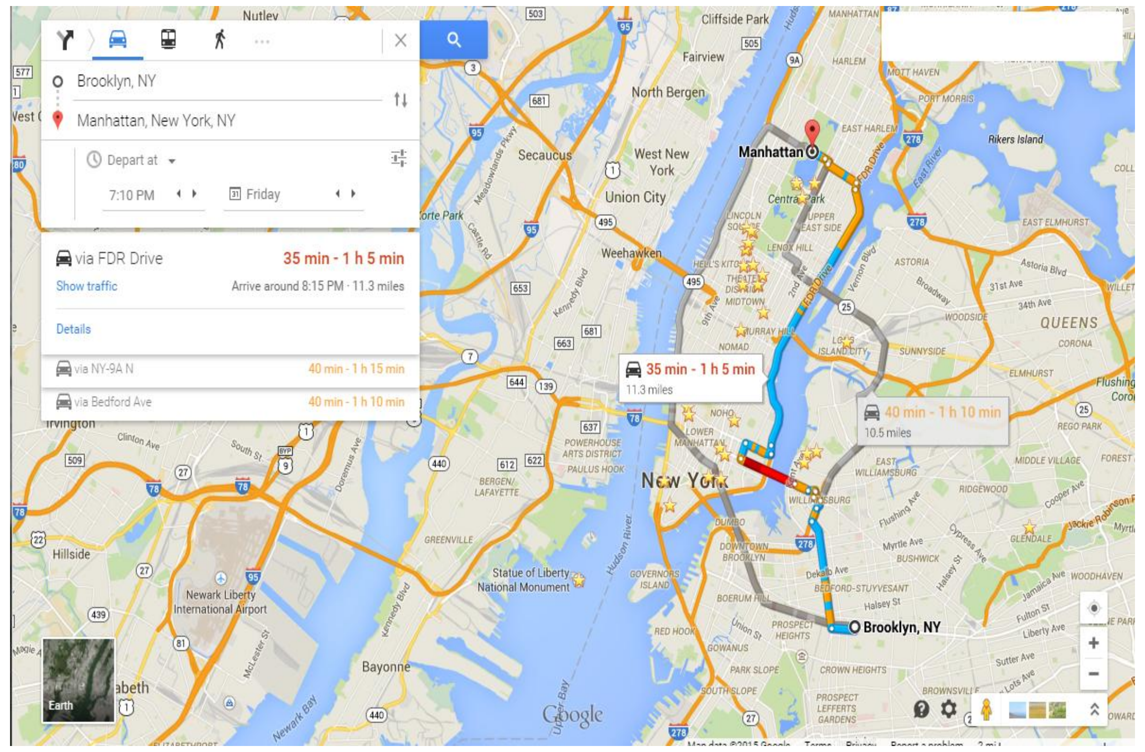
Bounded-Degree Graphs

World Wide Web

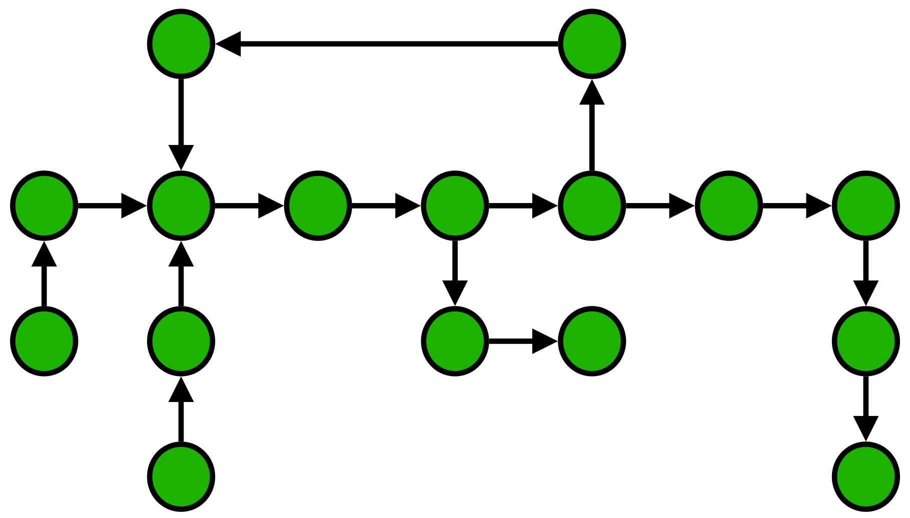


Power-Law Degree Distribution, Small Diameter, Poor Locality

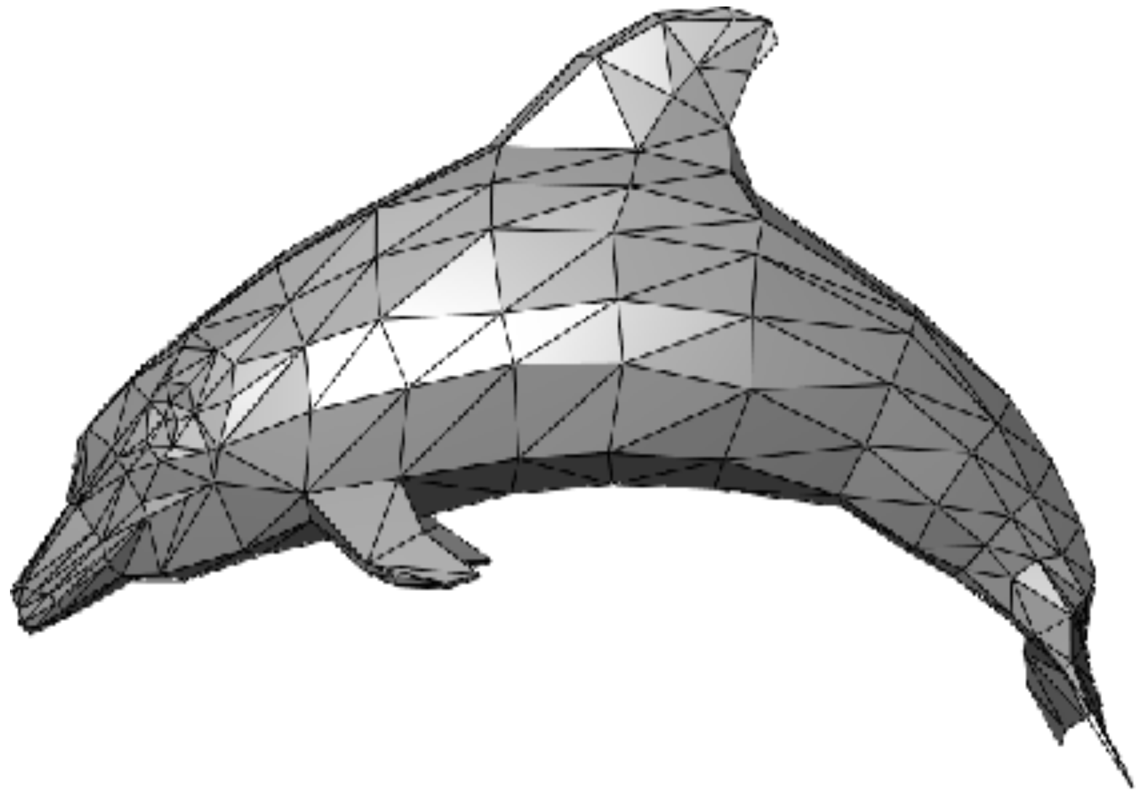
Social Networks



Maps



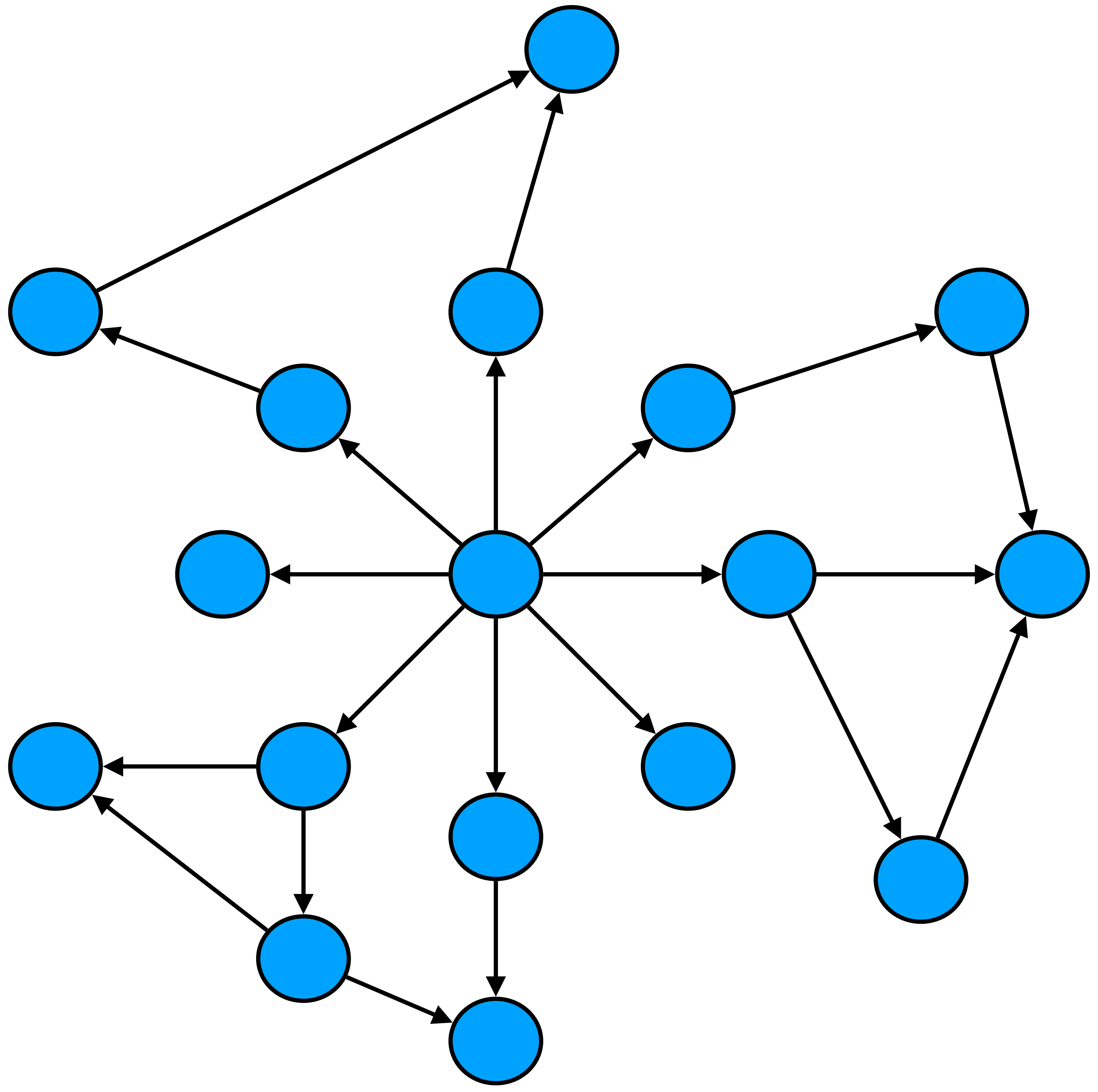
Bounded Degree Distribution
Large Diameter, Excellent Locality



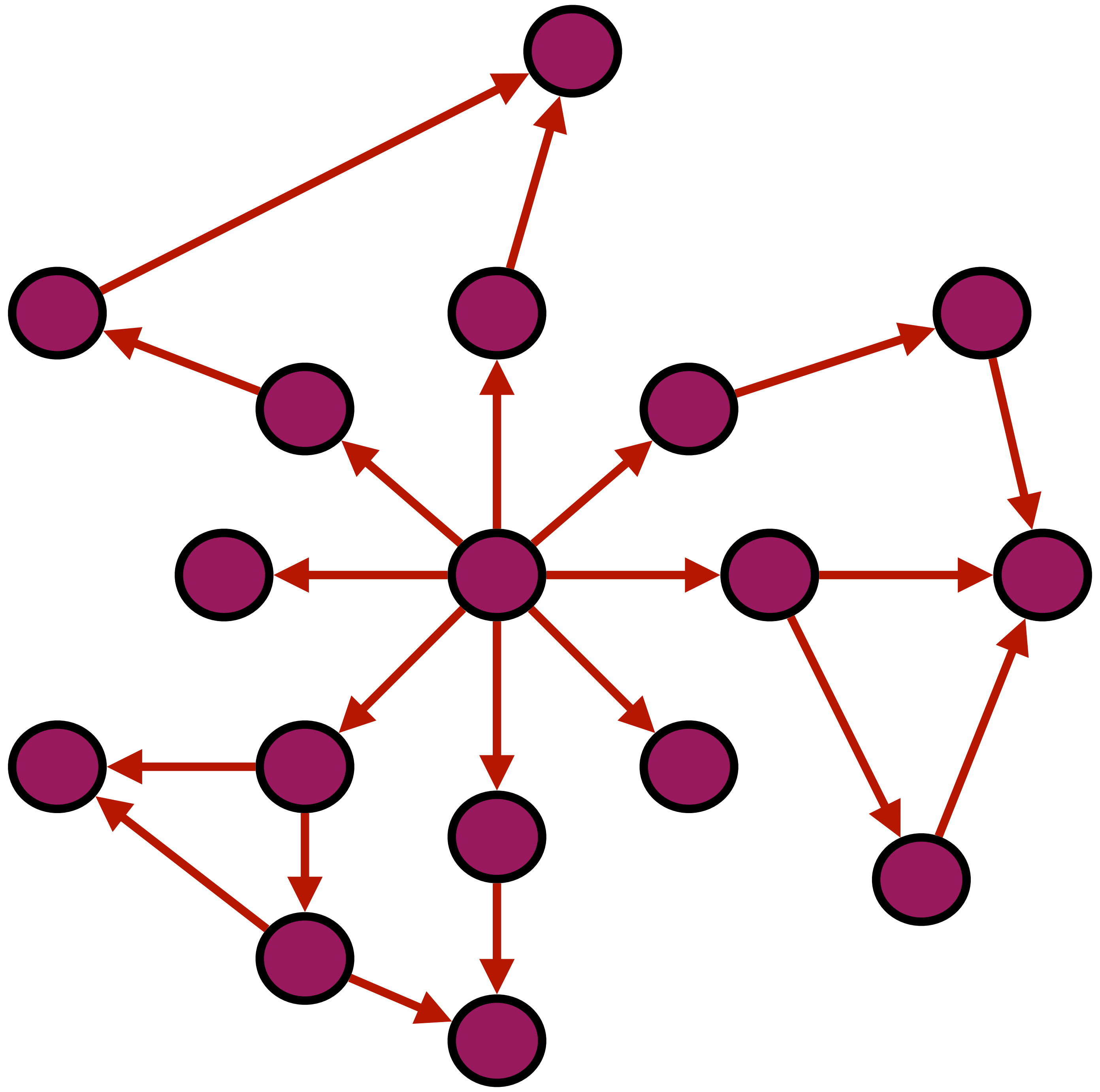
Engineering Meshes

1. http://googlesystem.blogspot.com/2007/05/world-wide-web-as-seen-by-google.html 2. http://www.facebookfever.com/introducing-facebook-new-graph-api-explorer-features/ 3. http://maps.google.com **4. https://en.wikipedia.org/wiki/Polygon_mesh#/media/File:Dolphin_triangle_mesh.png**

Graph Algorithms

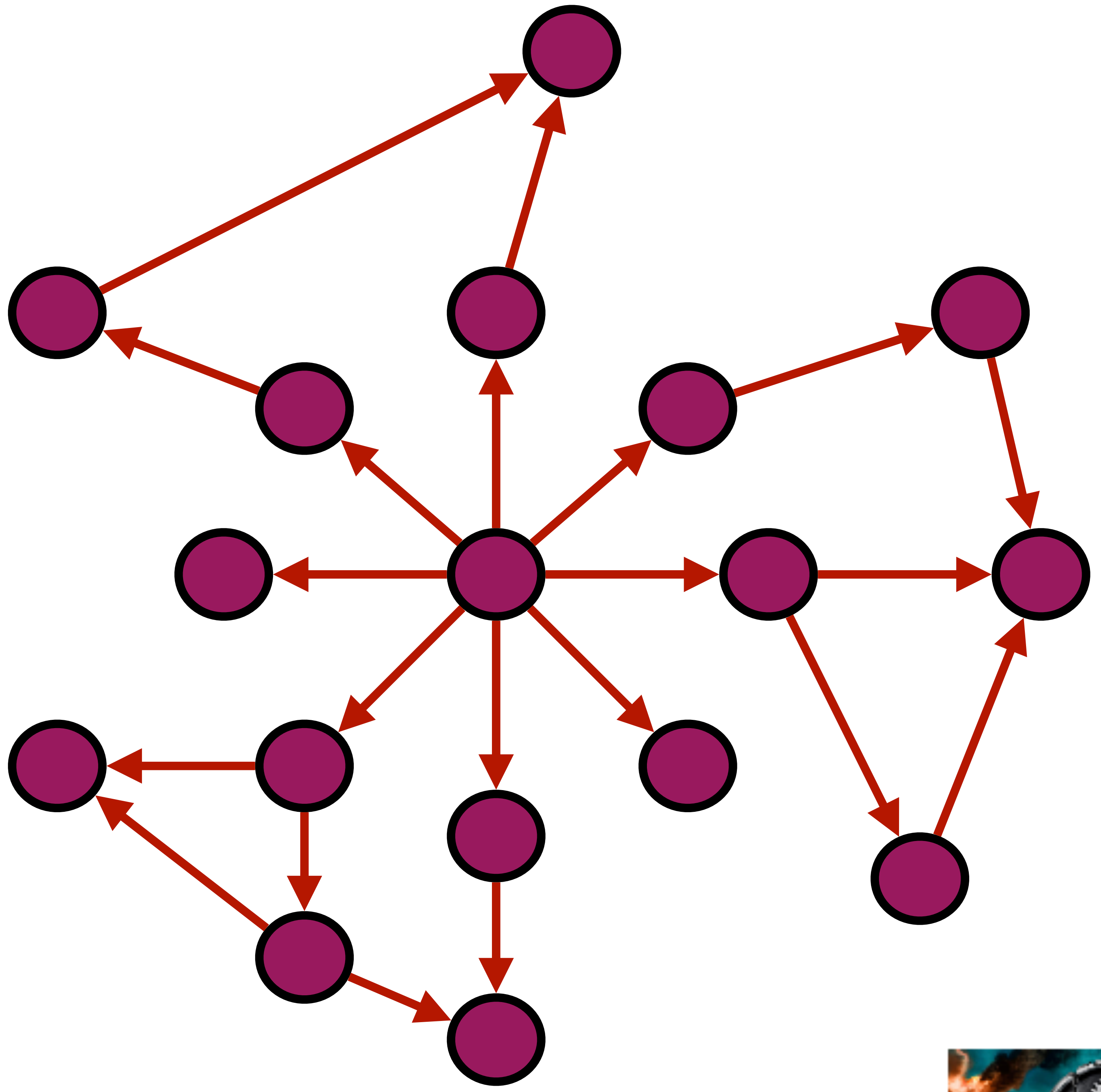


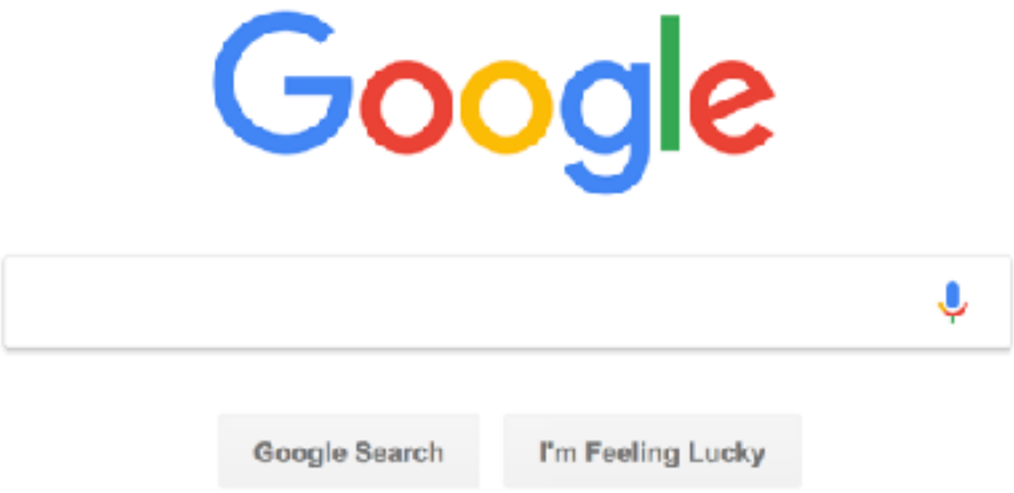
Topology-Driven Algorithms

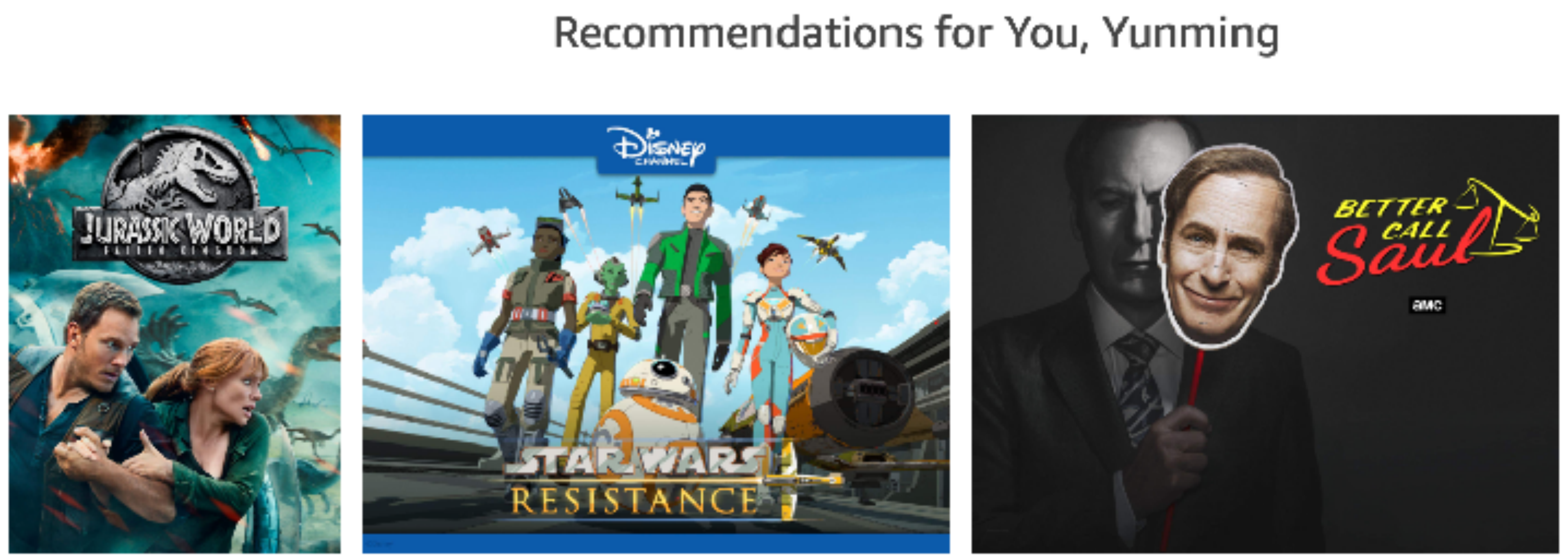


Work on All Edges and Vertices

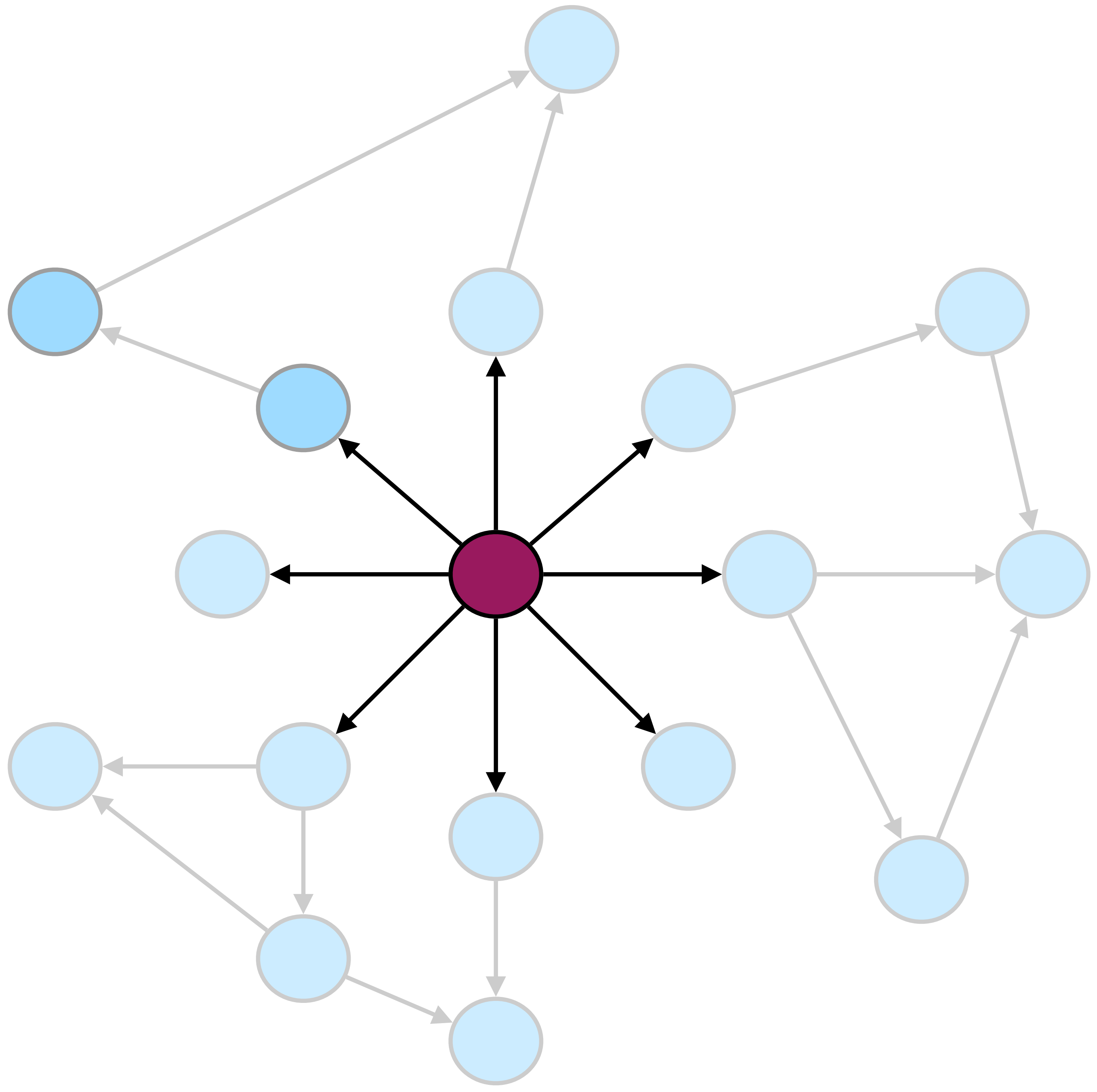
Topology-Driven Algorithms



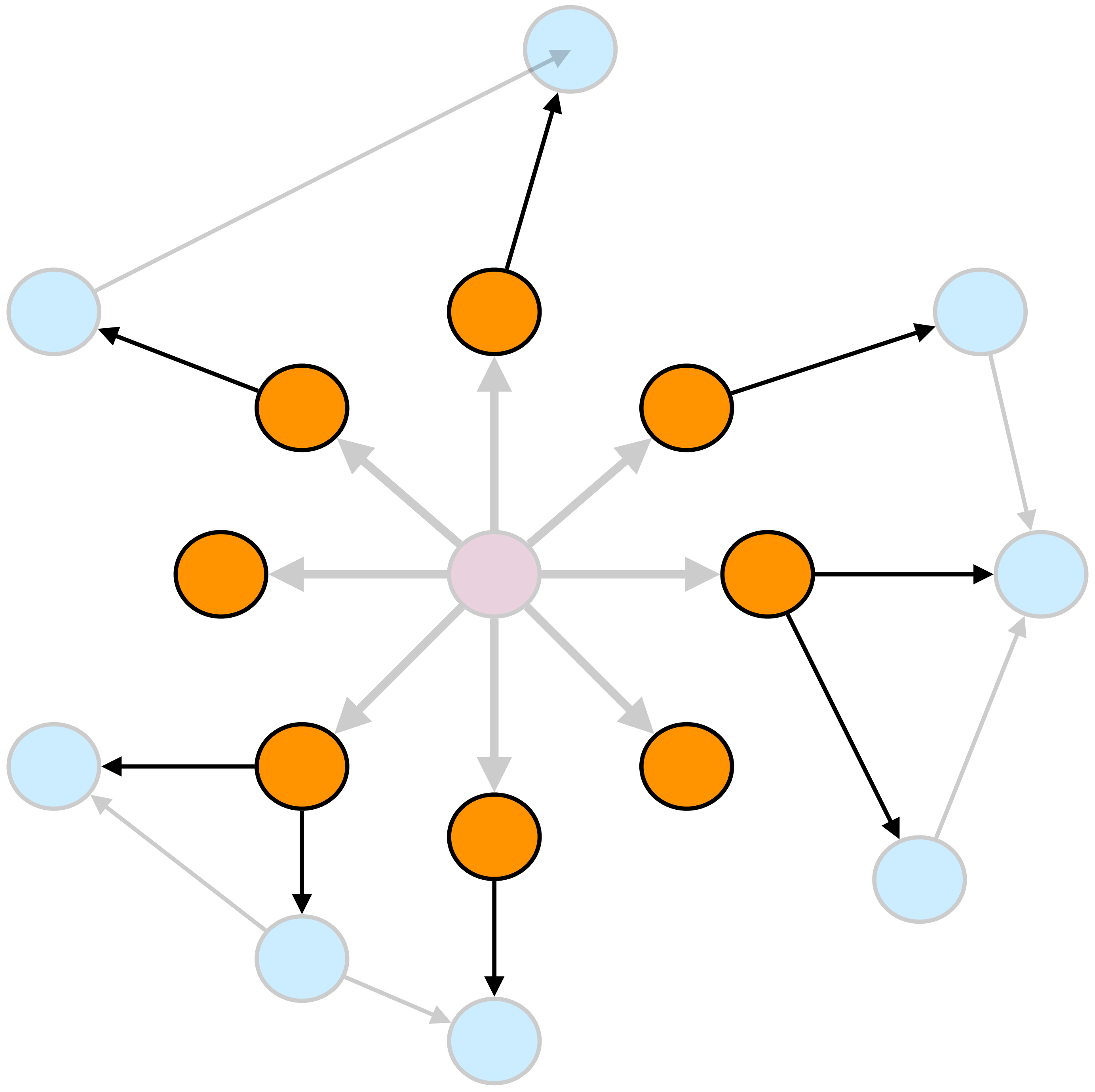



Recommendations for You, Yunming

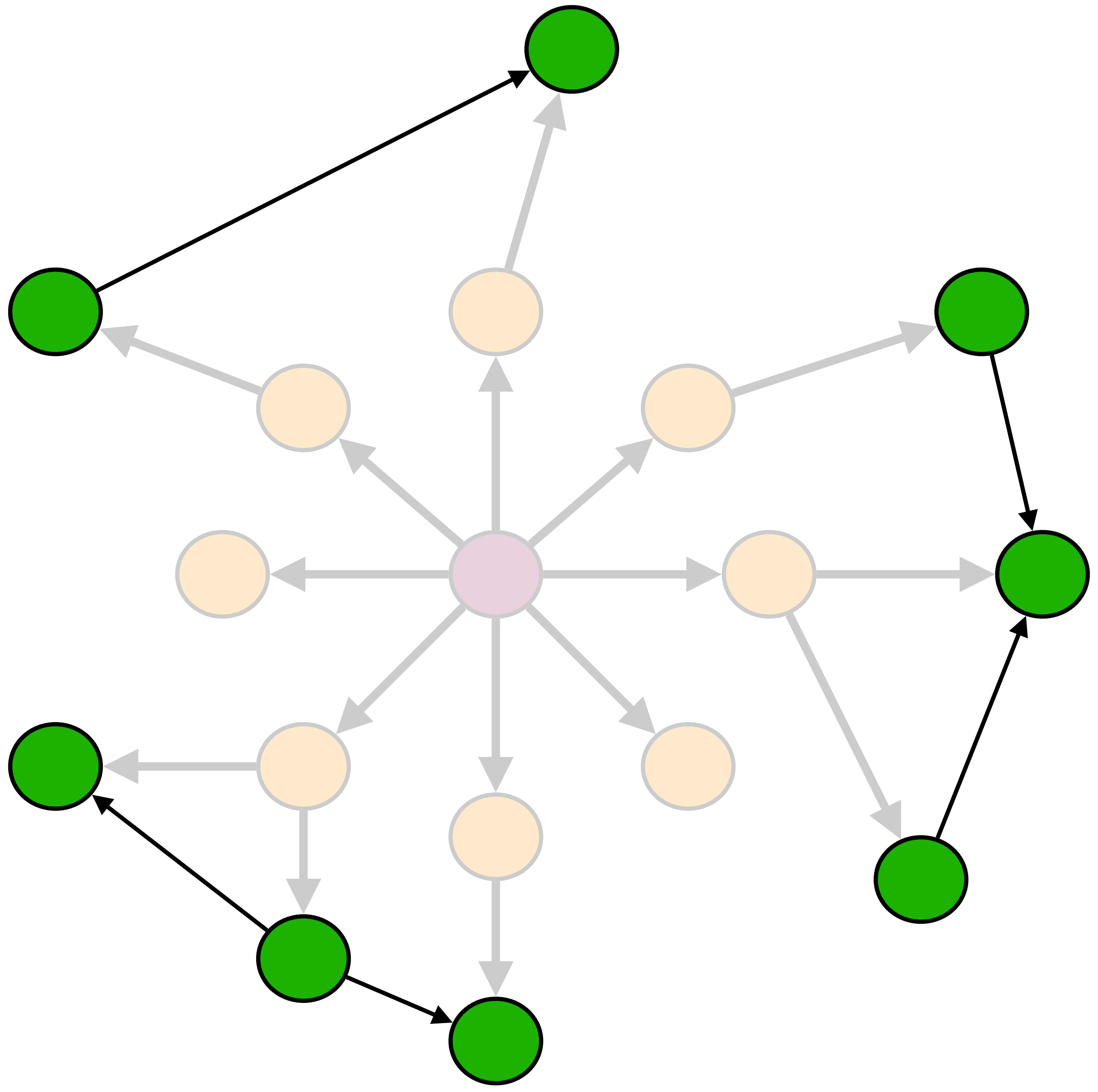
Data-Driven Algorithms



Data-Driven Algorithms

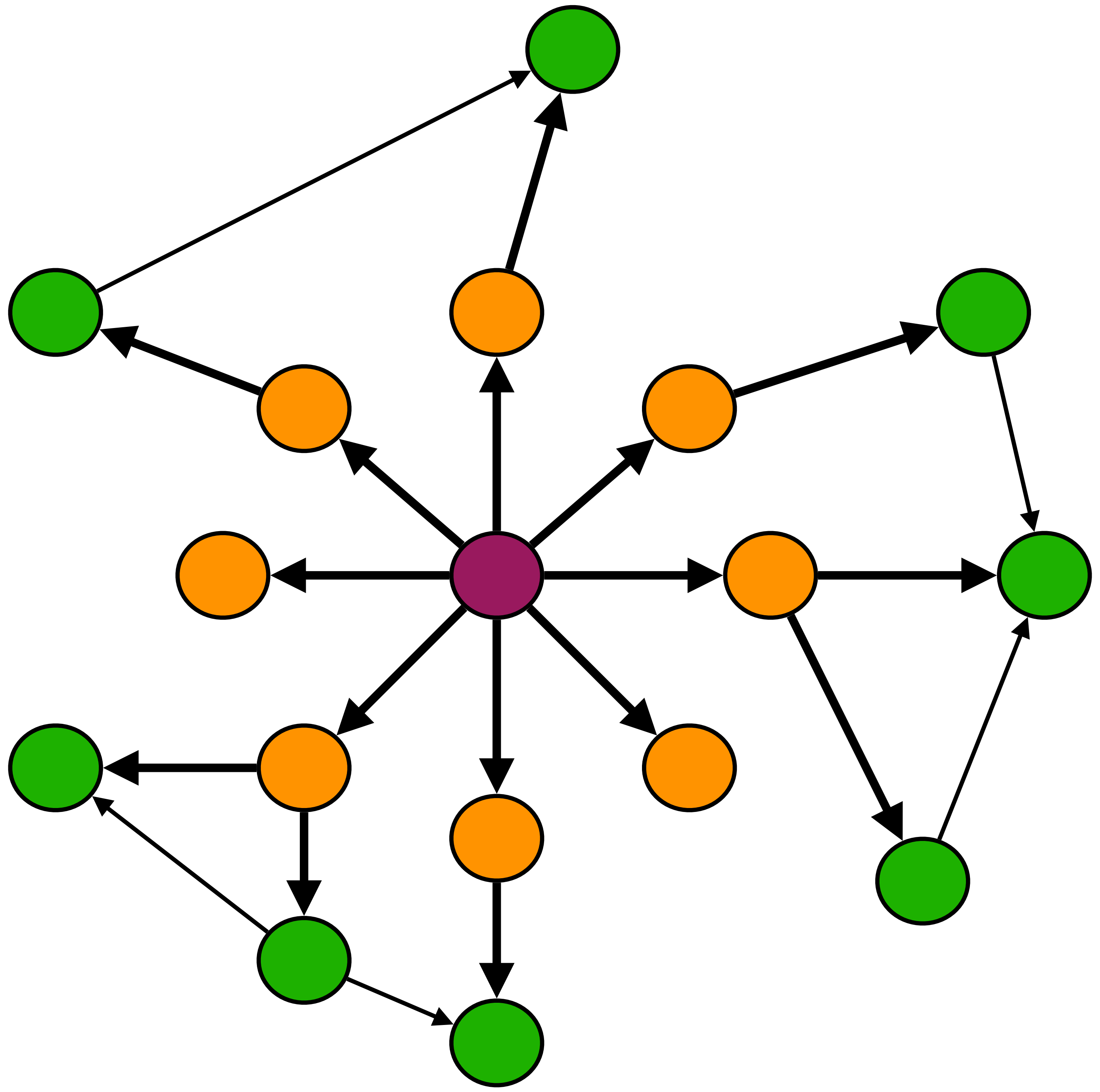


Data-Driven Algorithms



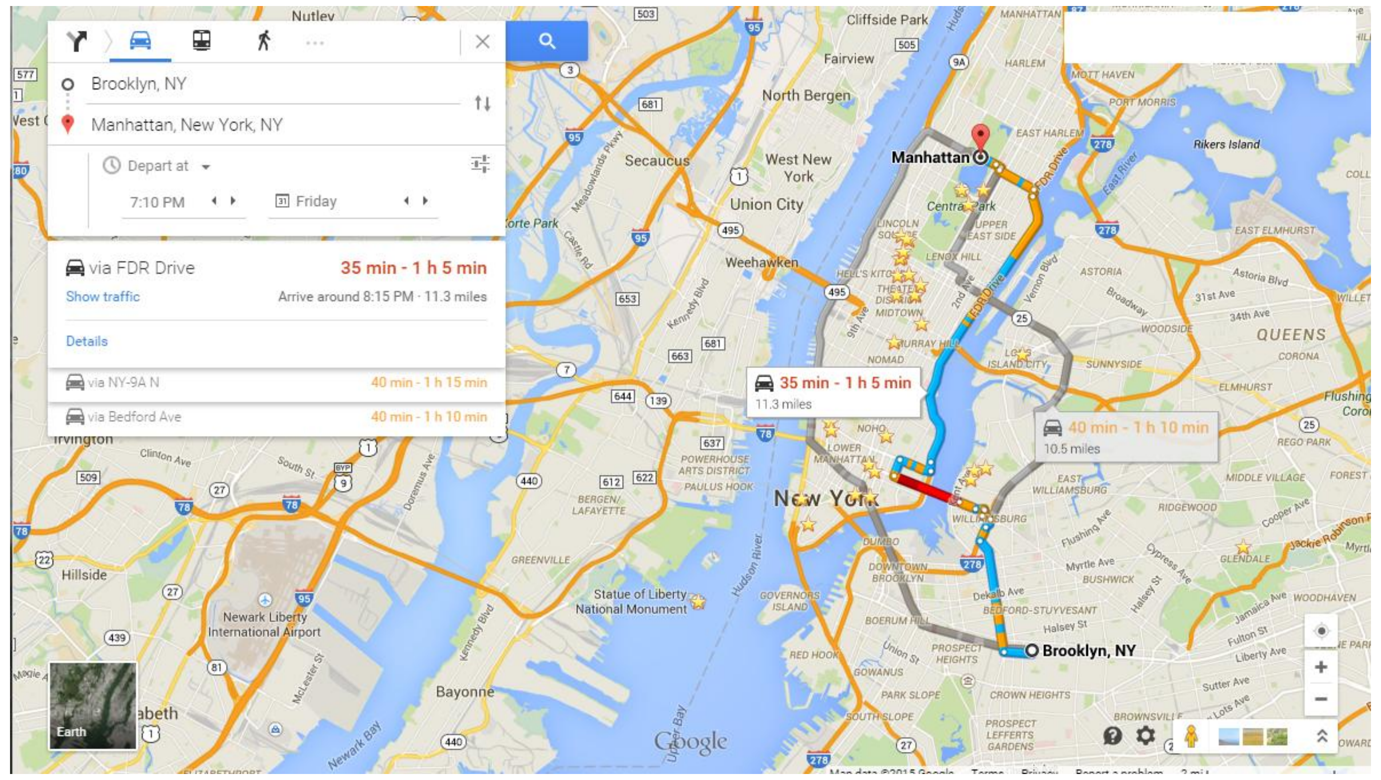
Work on a subset of vertices and edges
(Data-Driven)

Data-Driven Algorithms



Work on a subset of vertices and edges (Data-Driven)

Data-Driven Algorithms



Brooklyn, NY

Manhattan, New York, NY

Depart at

7:10 PM

Friday

via FDR Drive 35 min - 1 h 5 min

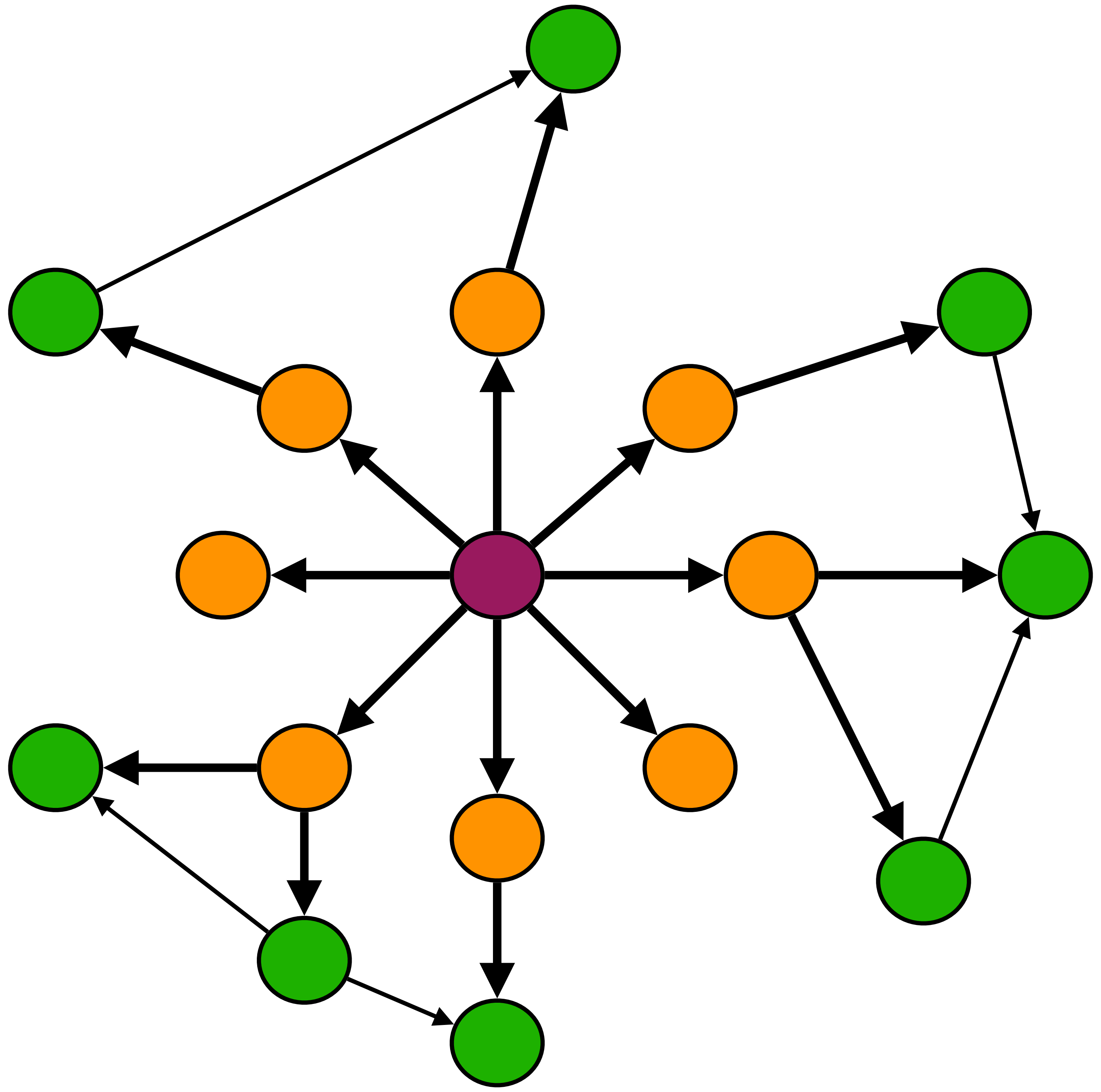
Show traffic

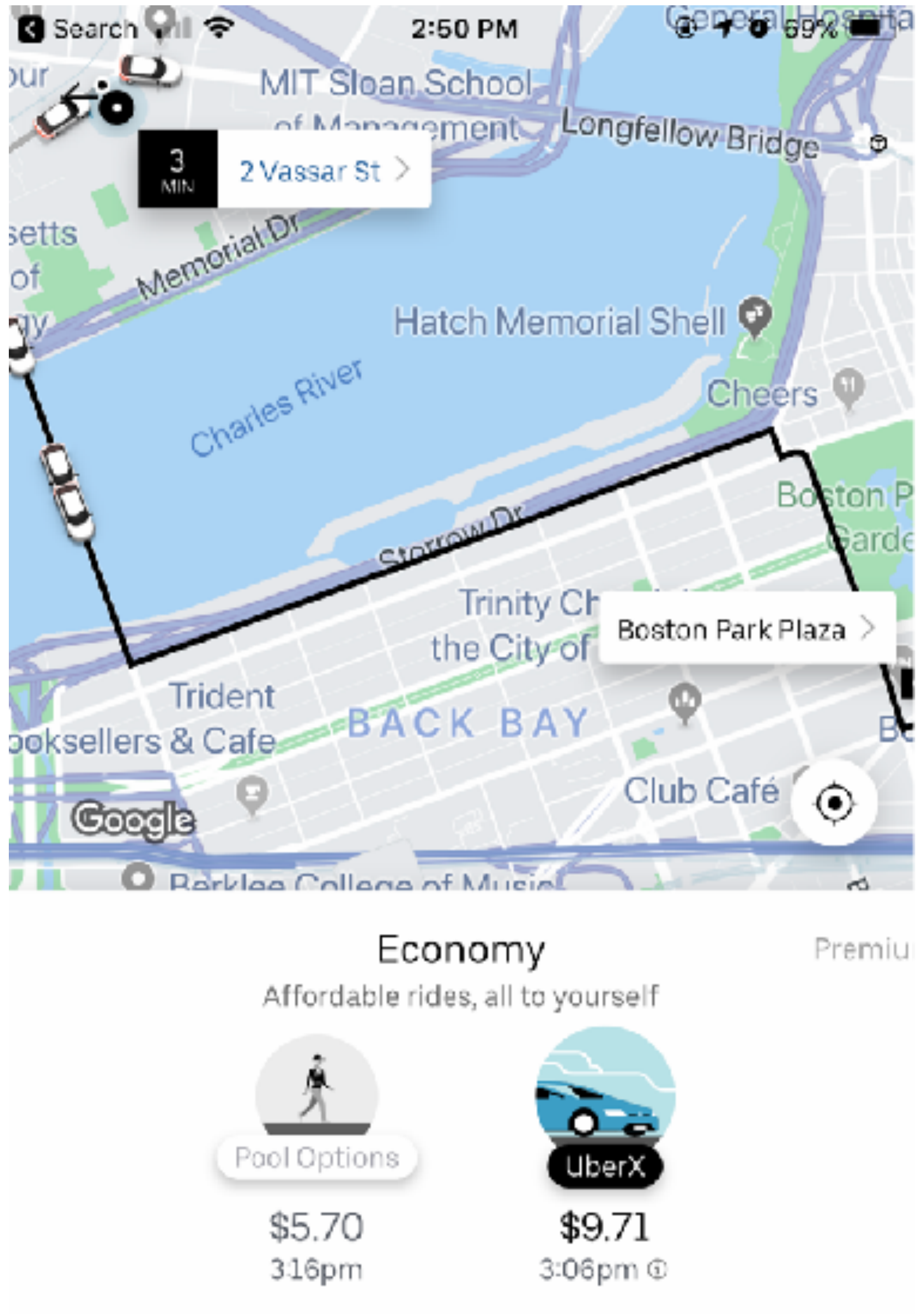
Arrive around 8:15 PM · 11.3 miles

Details

via NY-9A N 40 min - 1 h 15 min

via Bedford Ave 40 min - 1 h 10 min





3 MIN 2 Vassar St

Boston Park Plaza

Economy

Affordable rides, all to yourself

Pool Options $5.70 316pm

UberX $9.71 3:06pm

Graph Execution Hardware

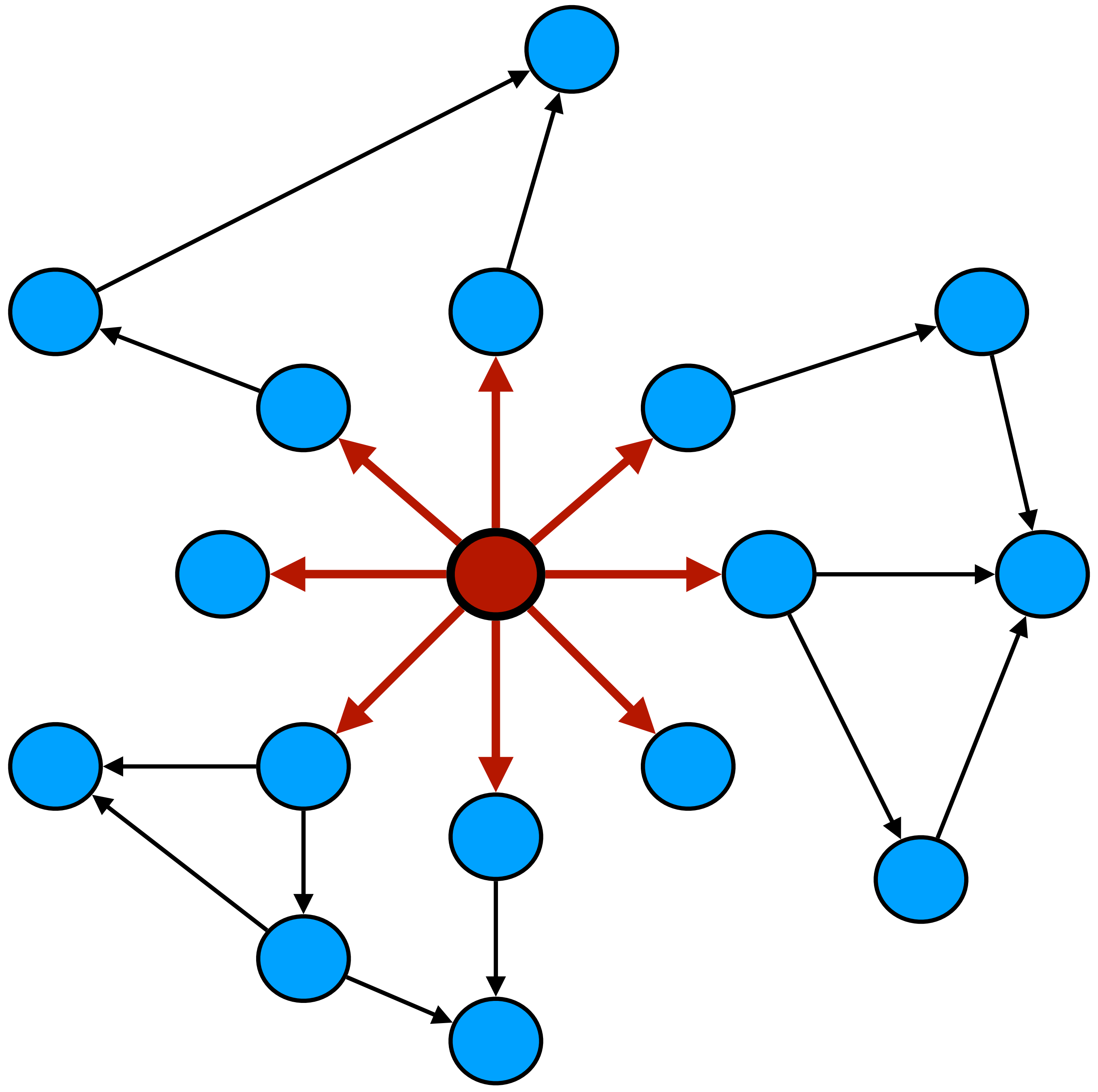
CPU

GPU

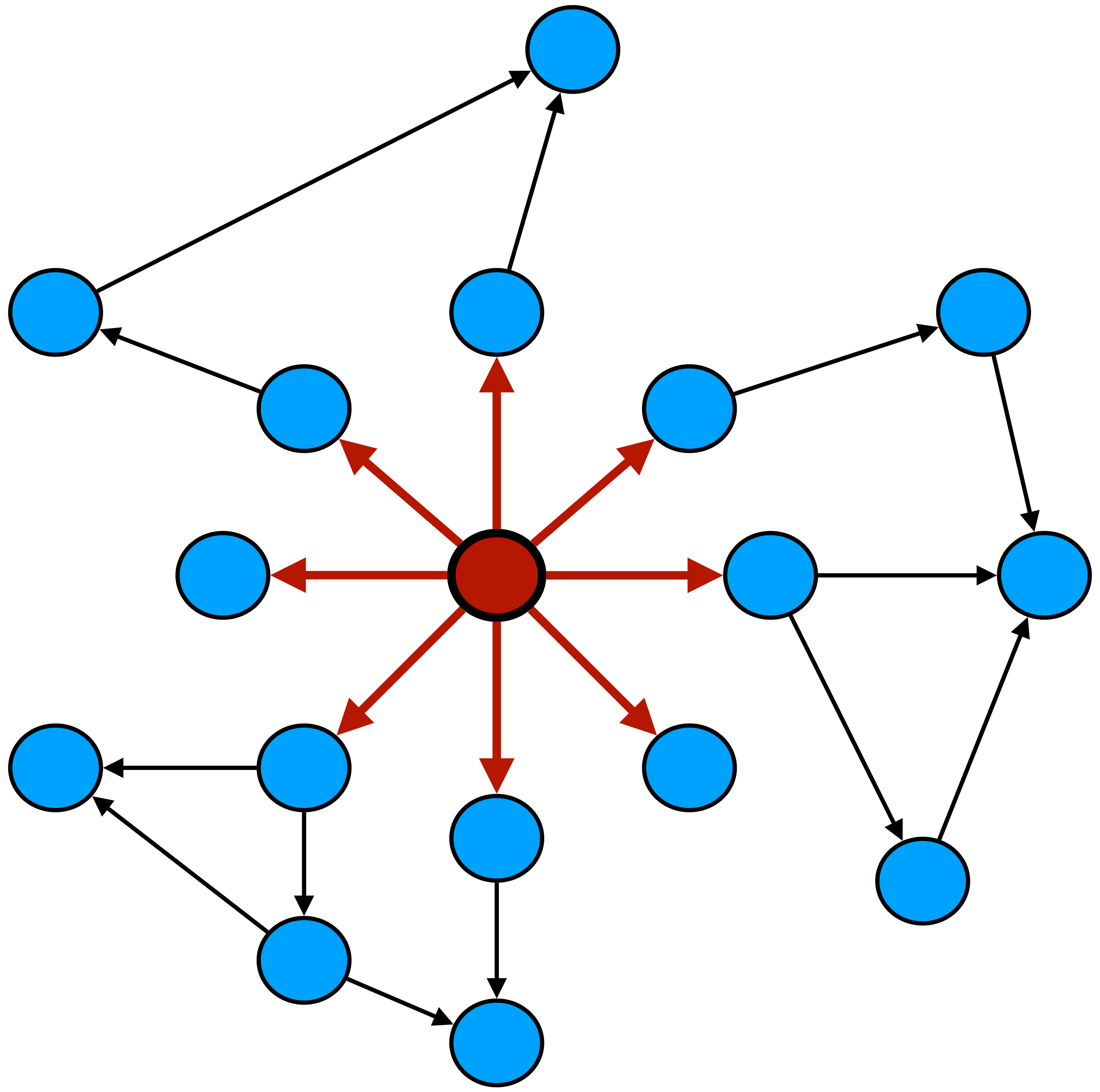
Xeon Phi

Distributed Cluster

Push Traversal



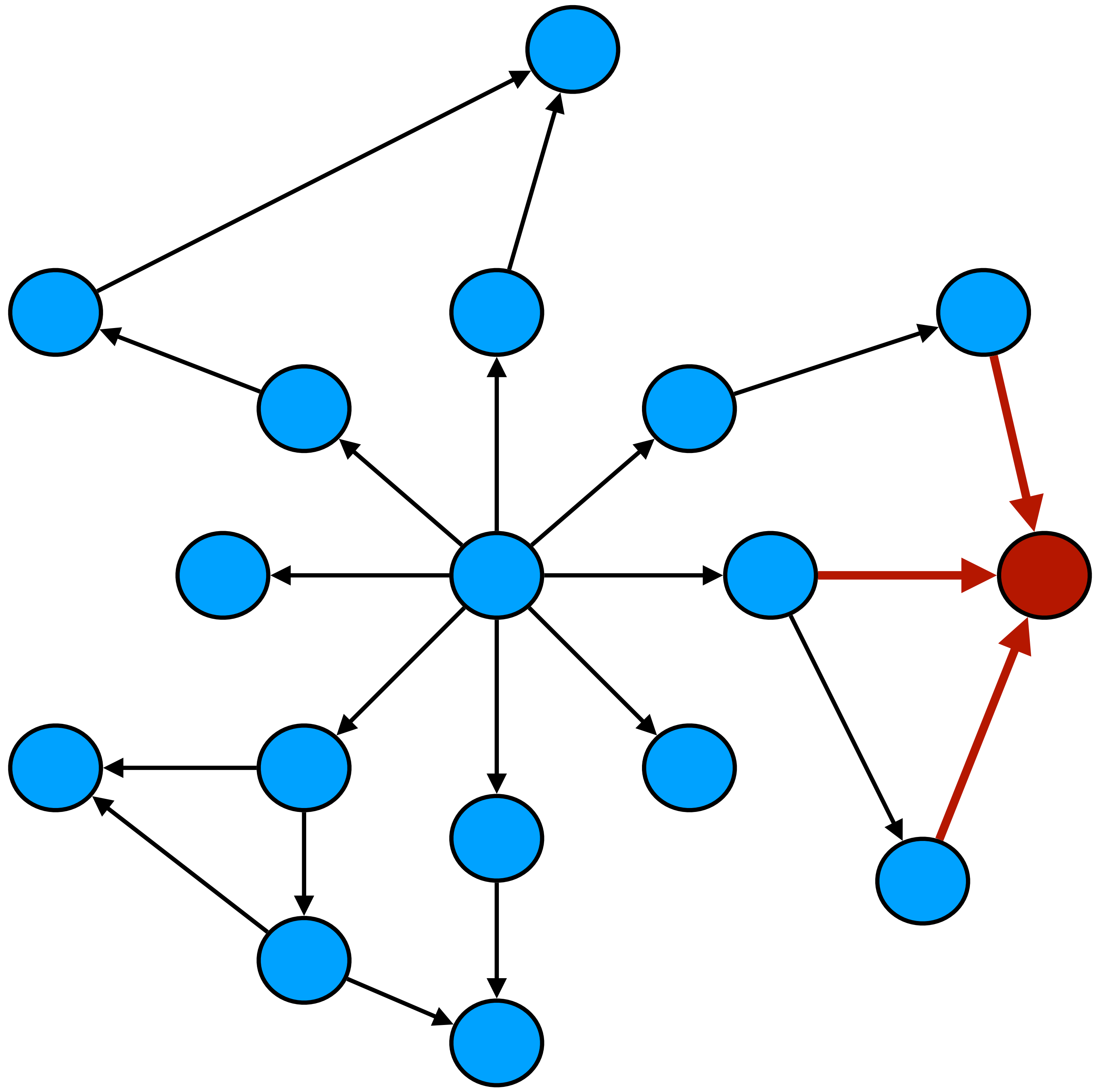
Push Traversal



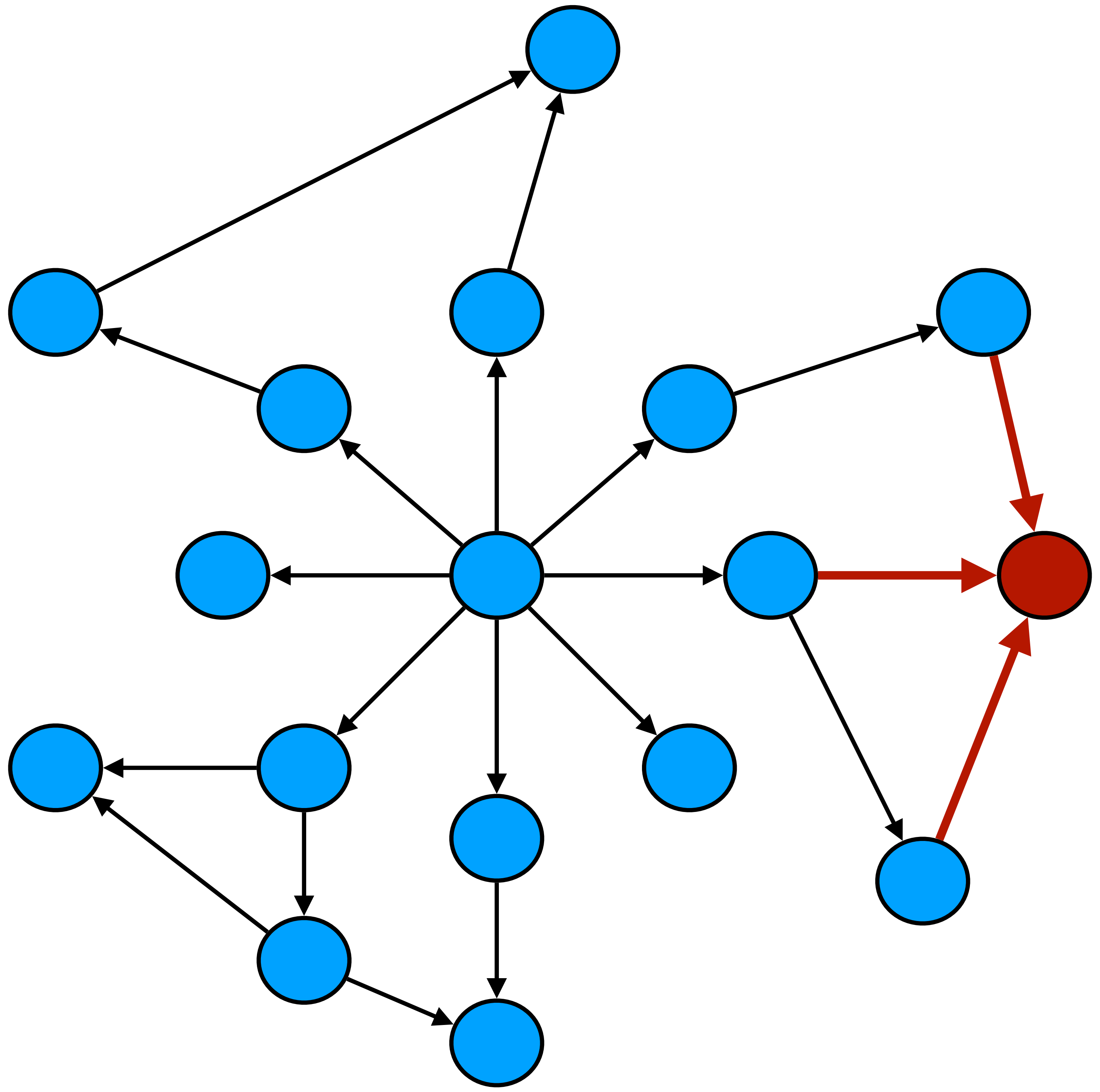
Incurs overhead with atomics

Traverses no extra edges

Pull Traversal



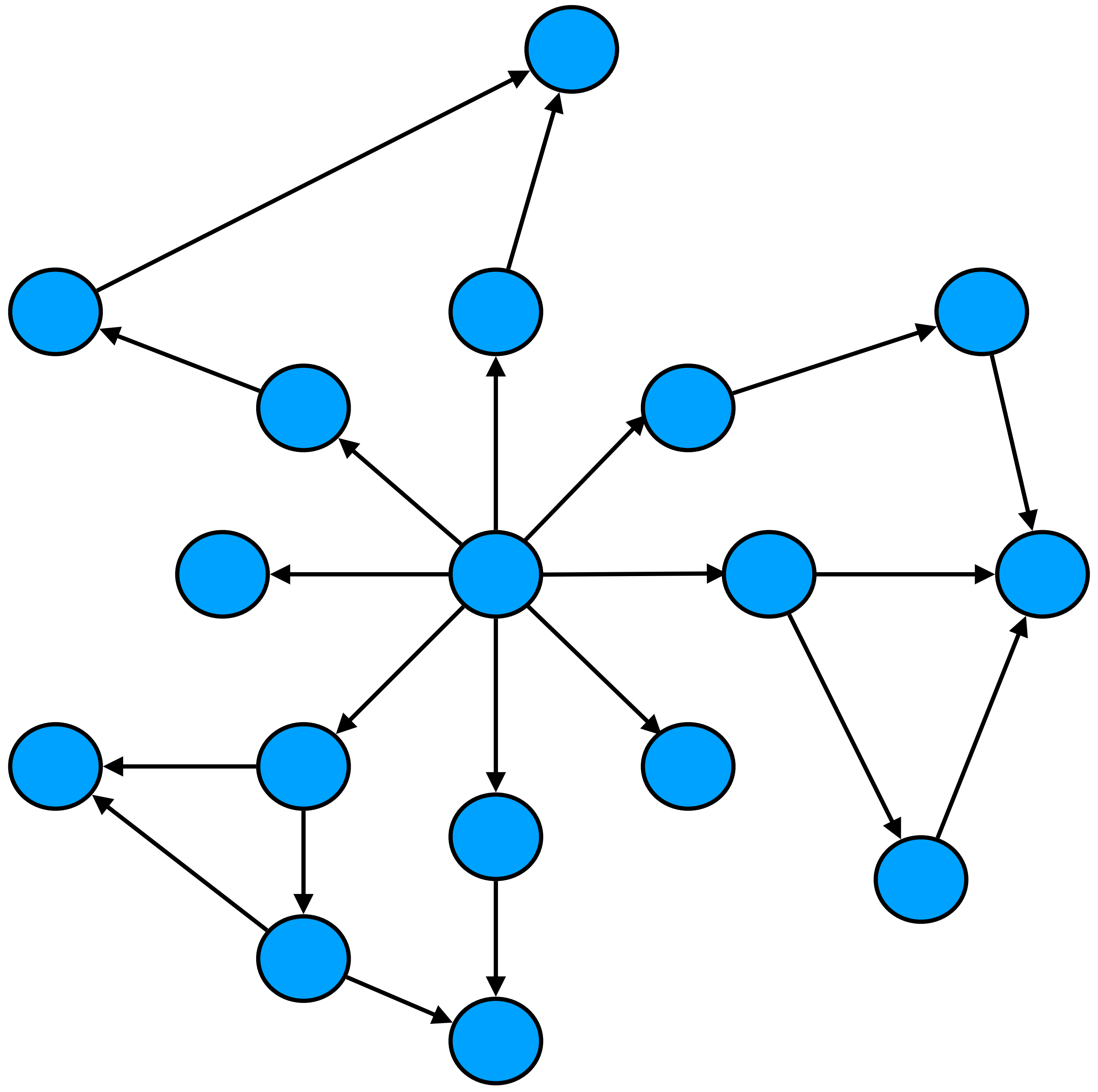
Pull Traversal



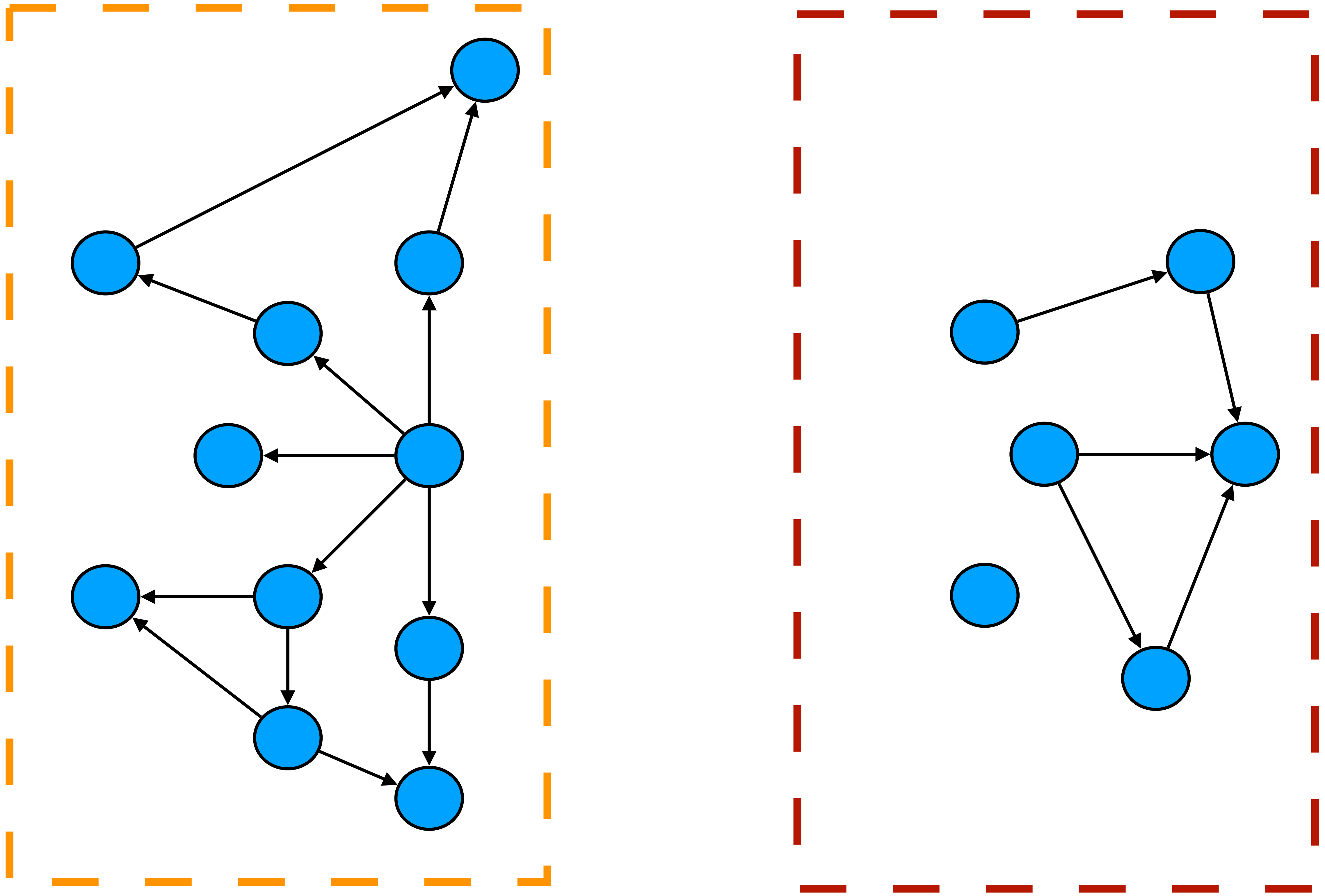
Incurs no overhead from atomics

Traverses extra edges

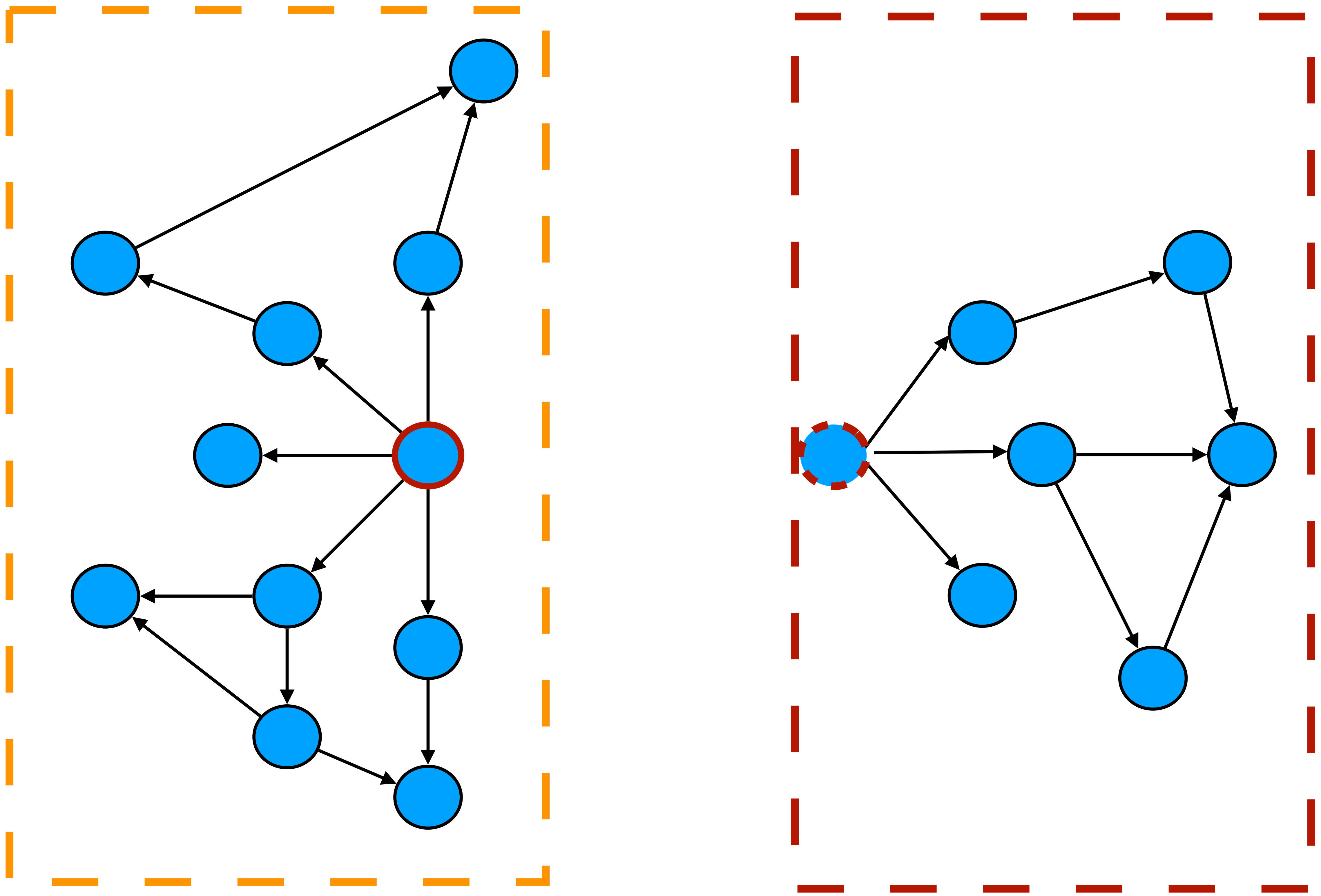
Partitioning



Partitioning



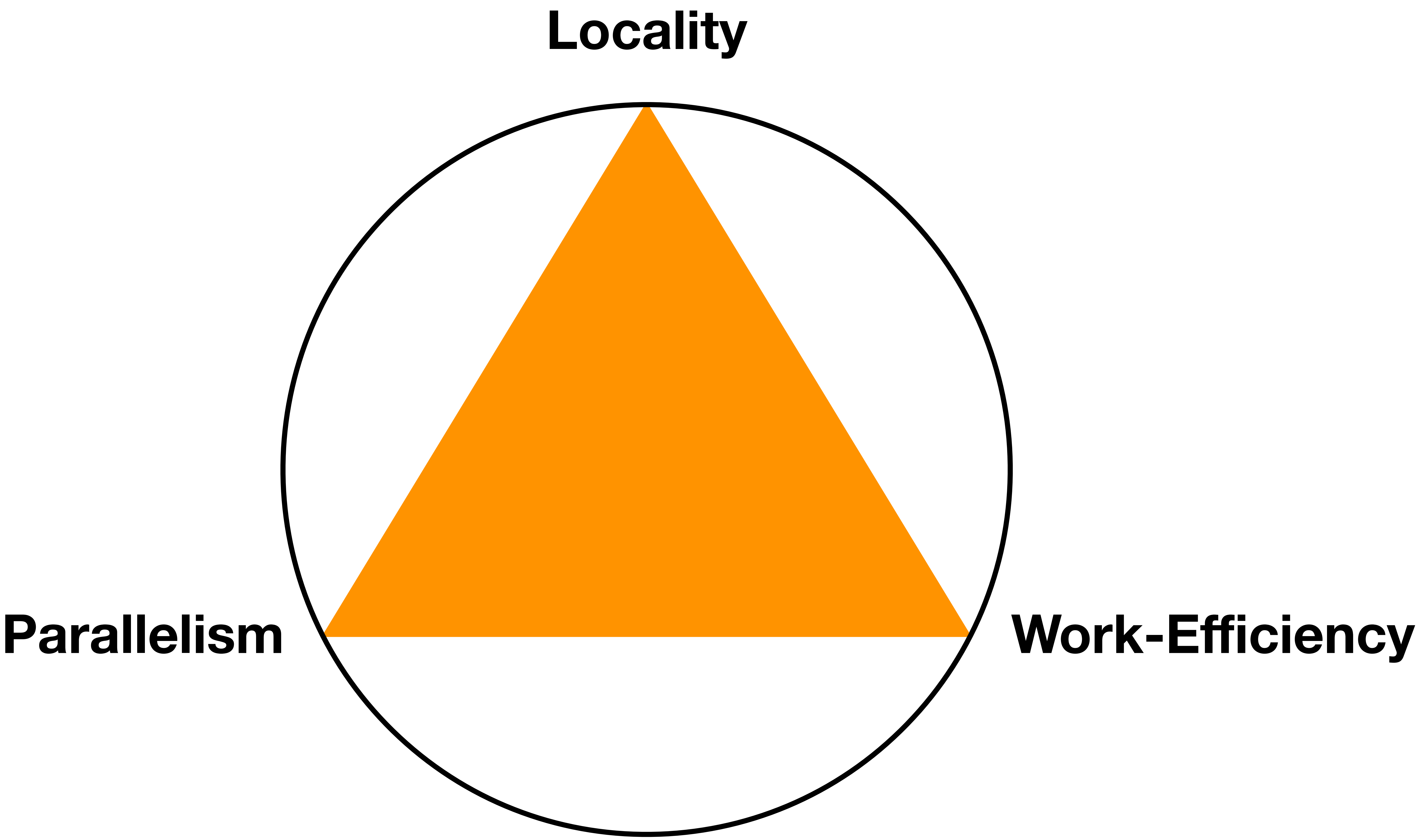
Partitioning



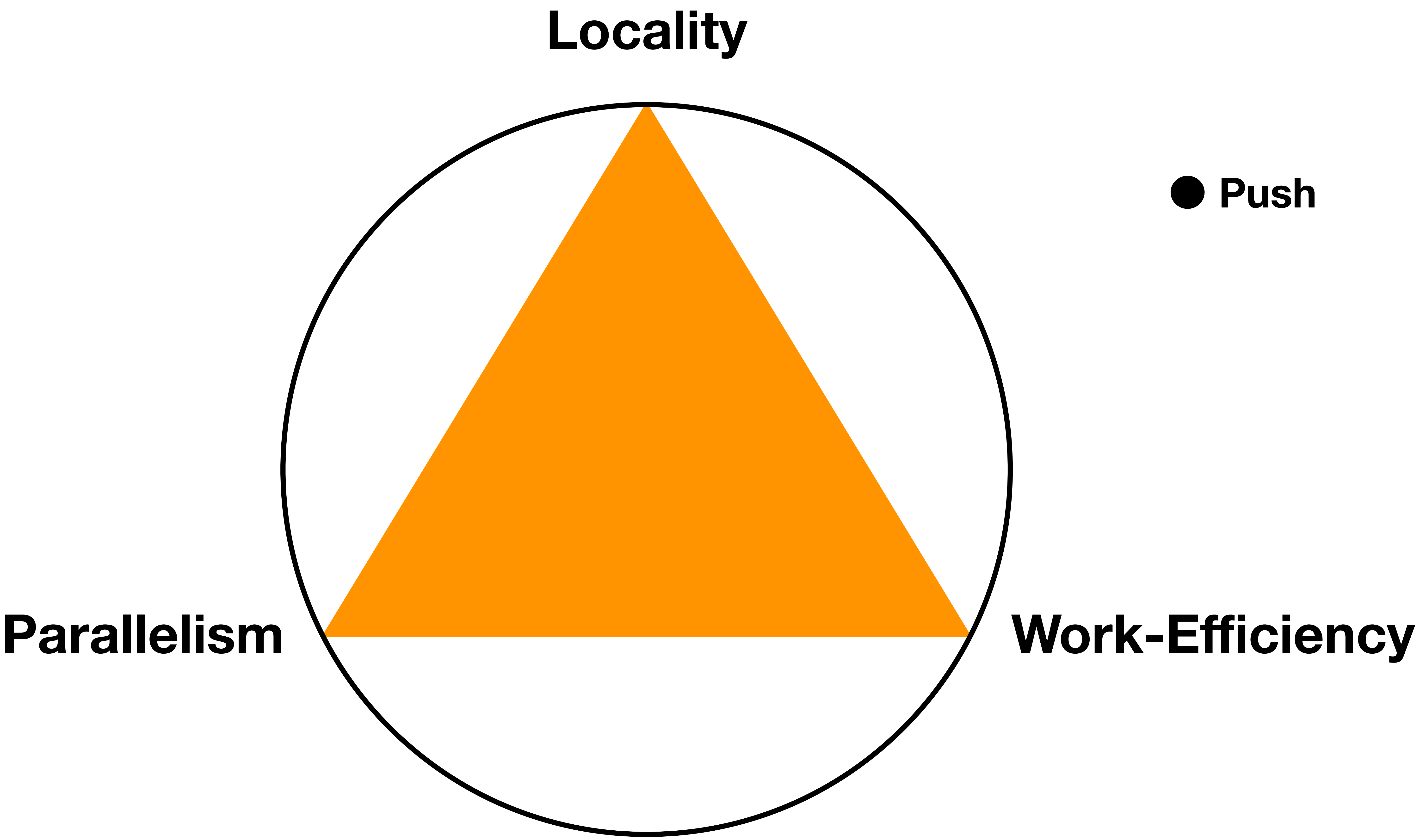
Improves locality

Needs extra instructions to traverse two graphs

Optimization Tradeoff Space

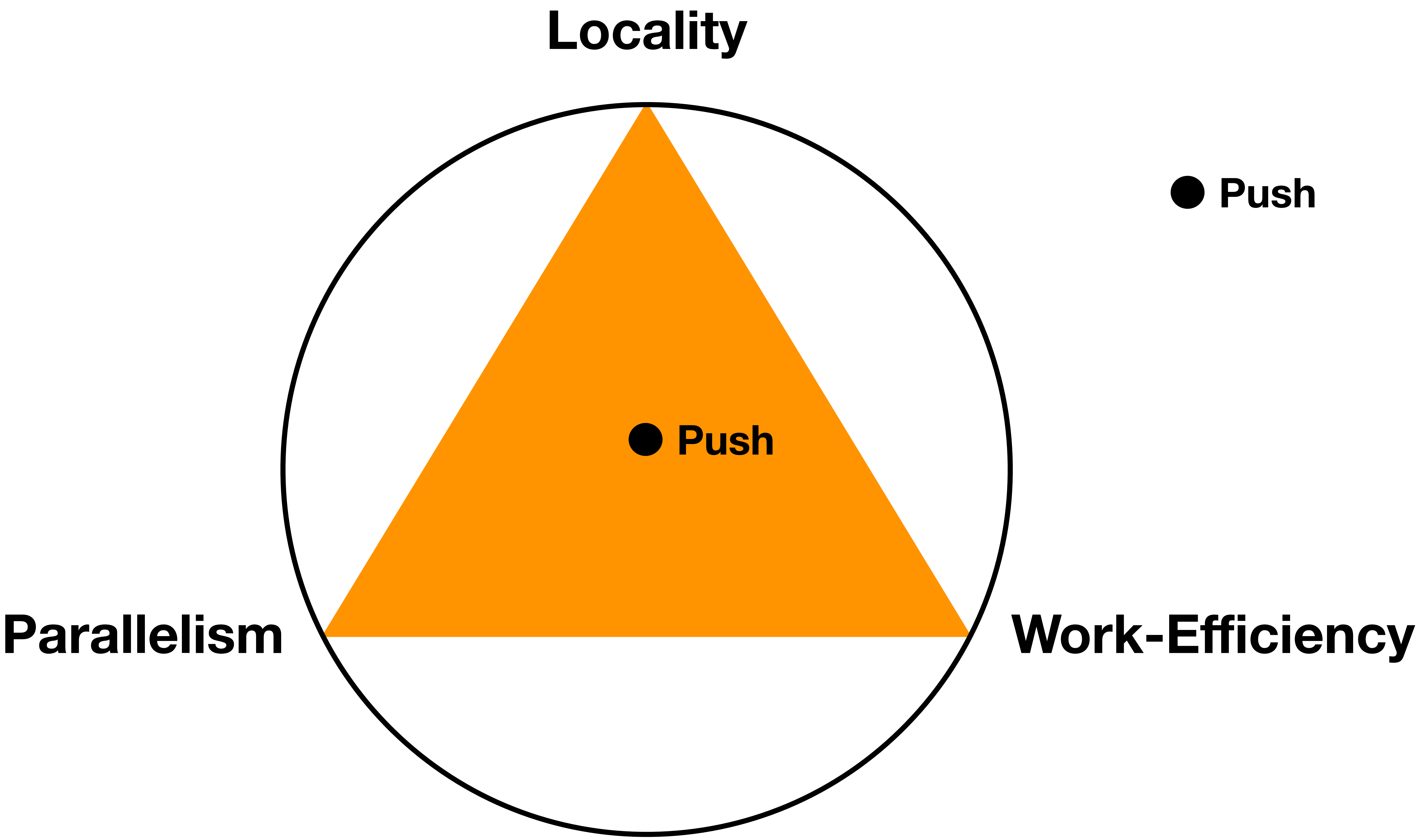


Optimization Tradeoff Space

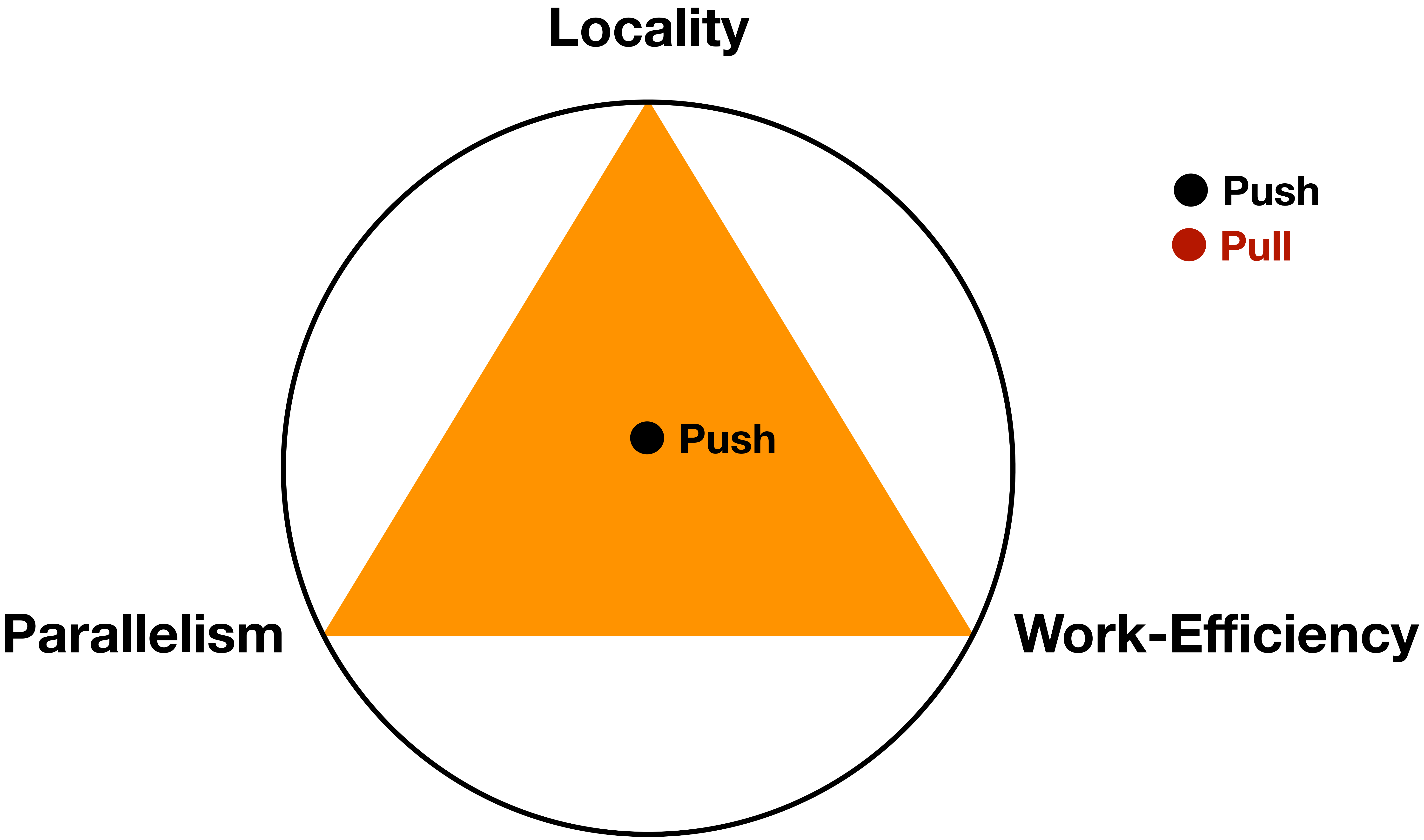


● Push

Optimization Tradeoff Space

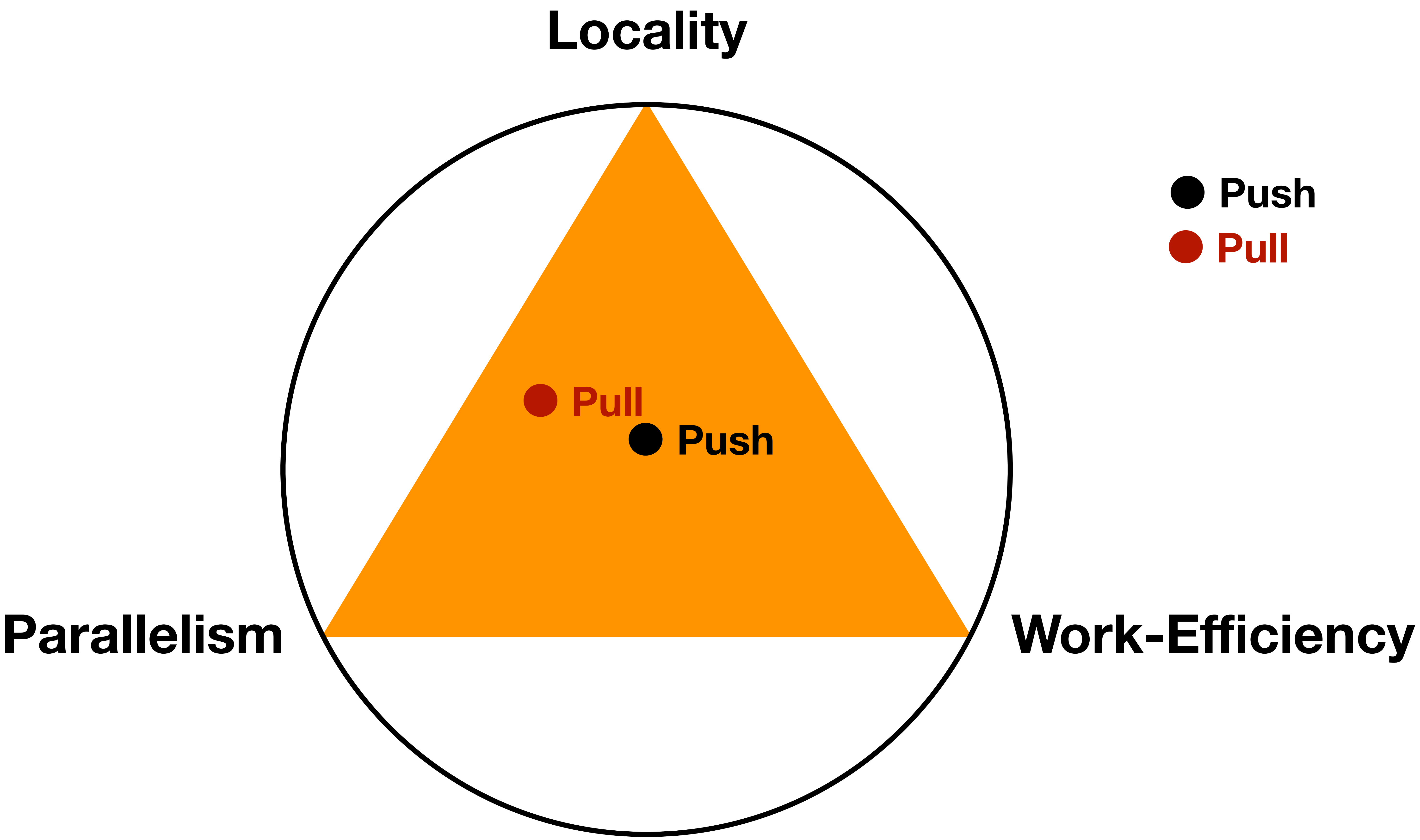


Optimization Tradeoff Space



● Push
● Pull

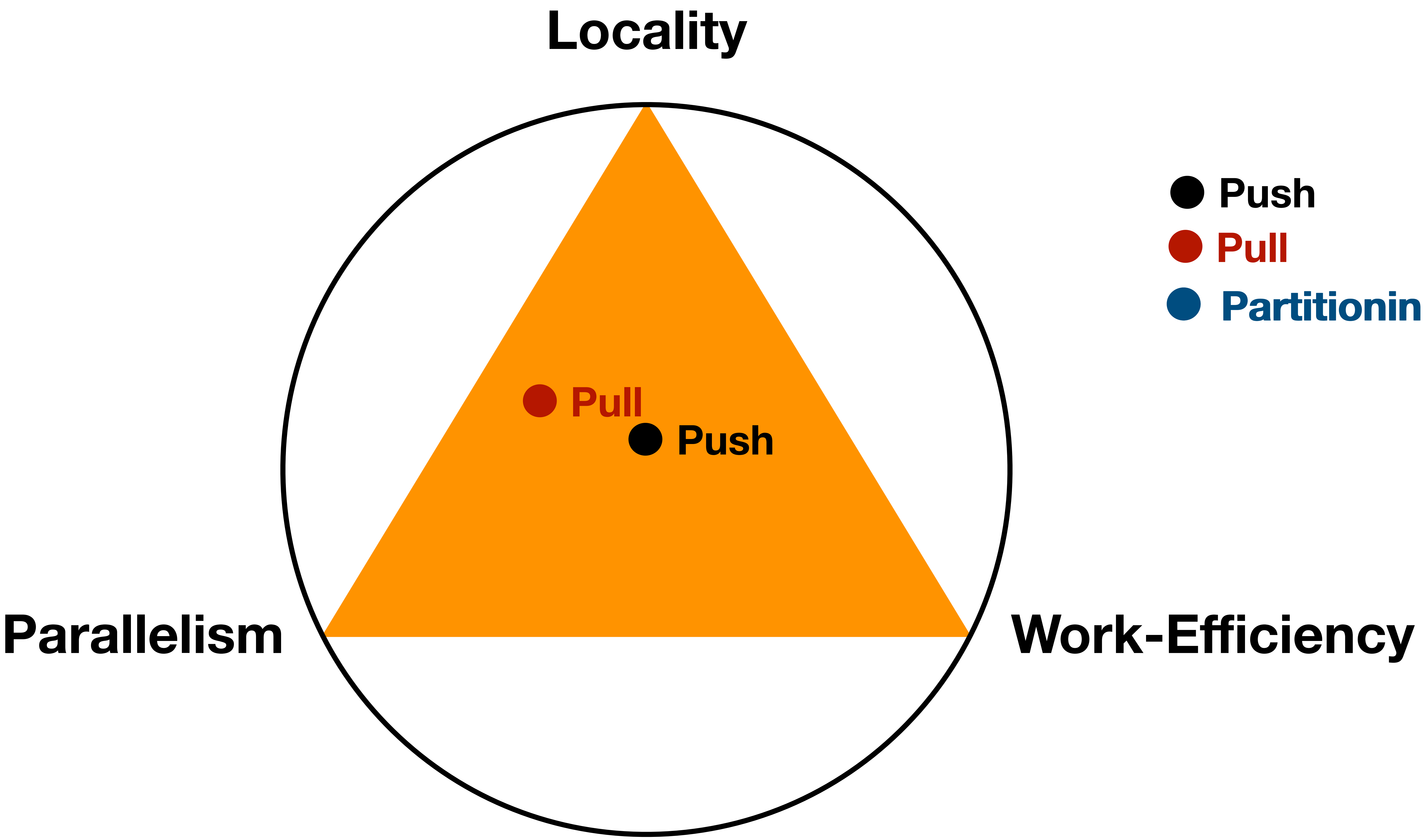
Optimization Tradeoff Space



● Push

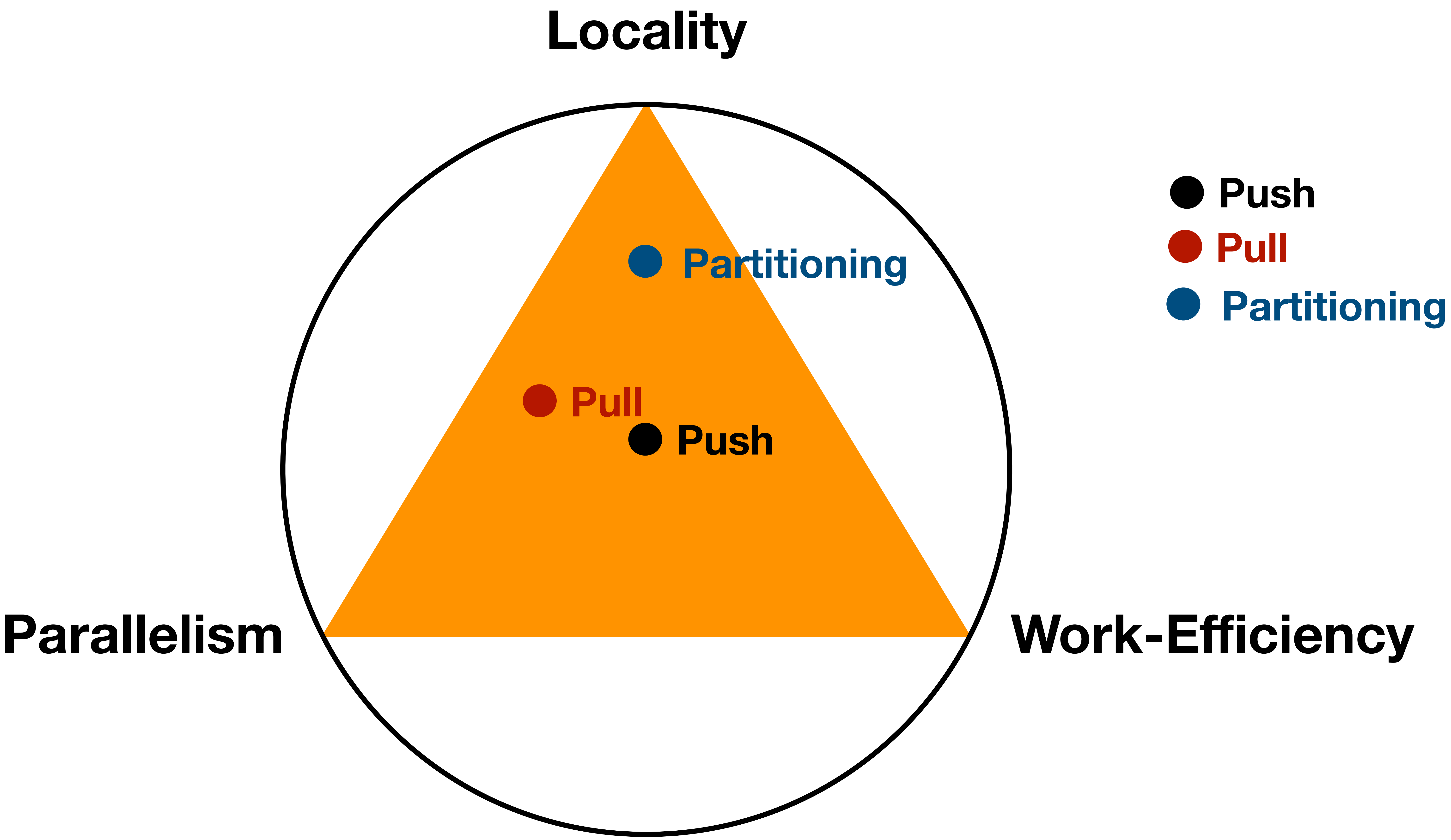
● Pull

Optimization Tradeoff Space

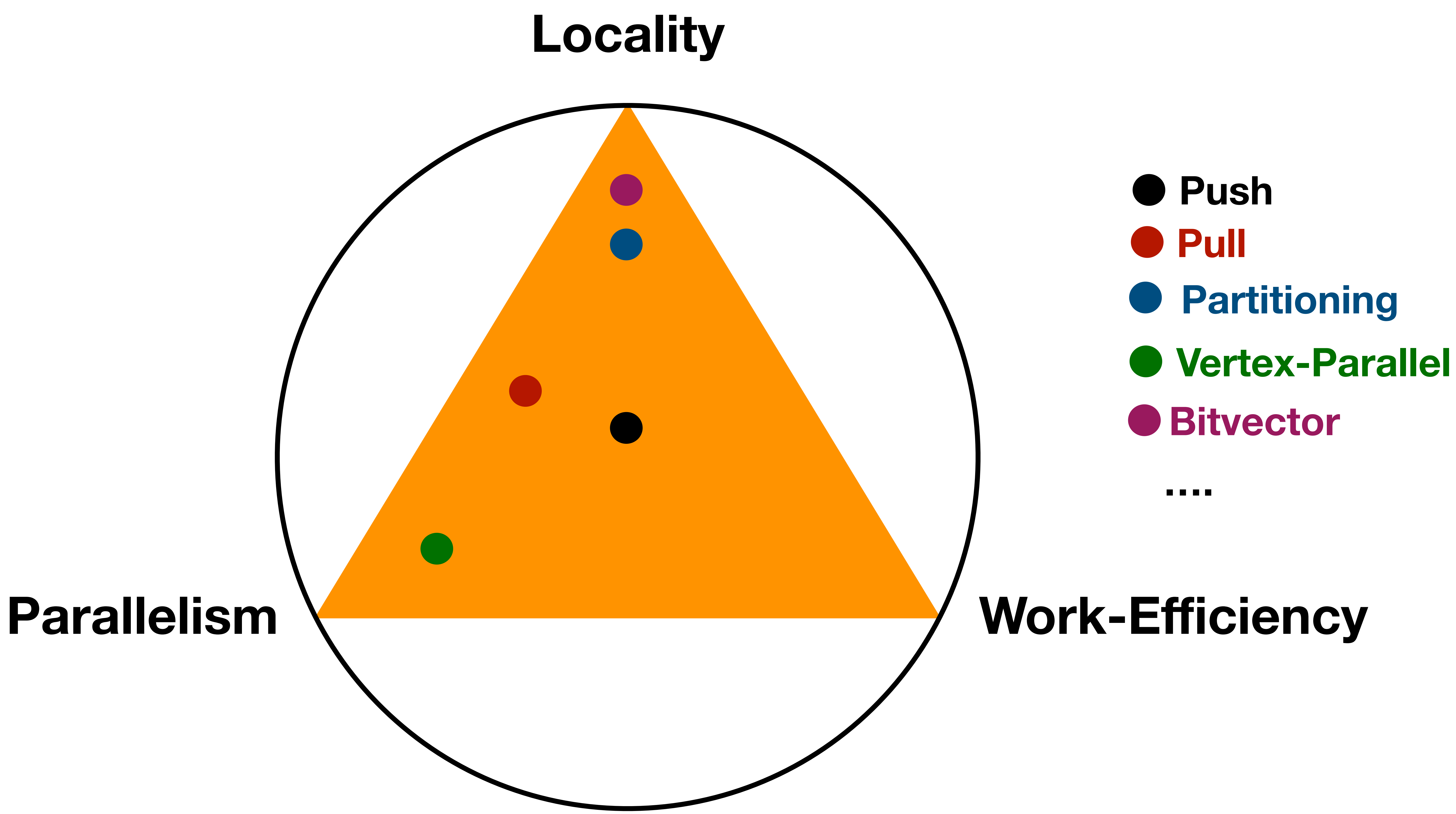


- ● Push
- ● Pull
- ● Partitioning

Optimization Tradeoff Space



Optimization Tradeoff Space



Locality

Parallelism

Work-Efficiency

● Push

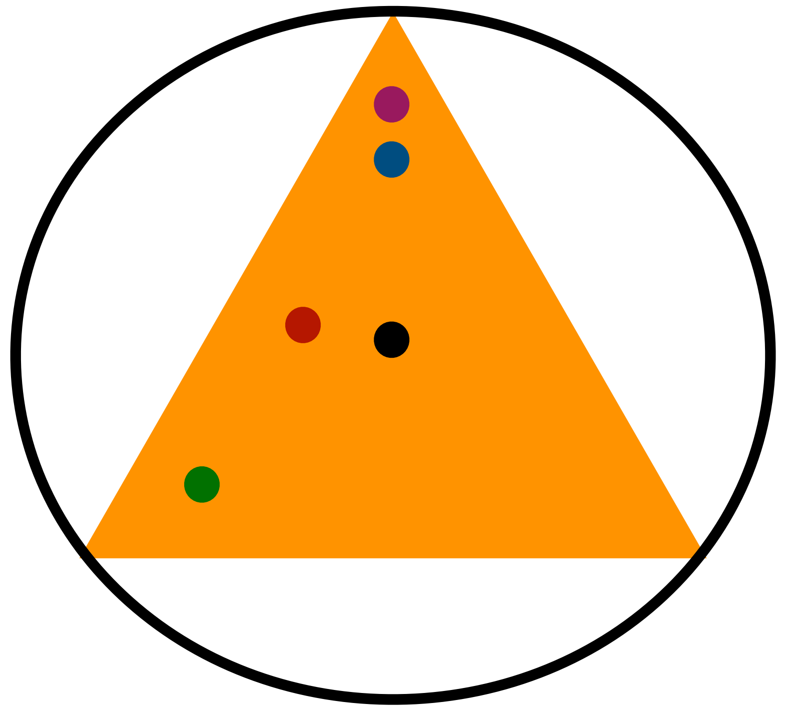
● Pull

● Partitioning

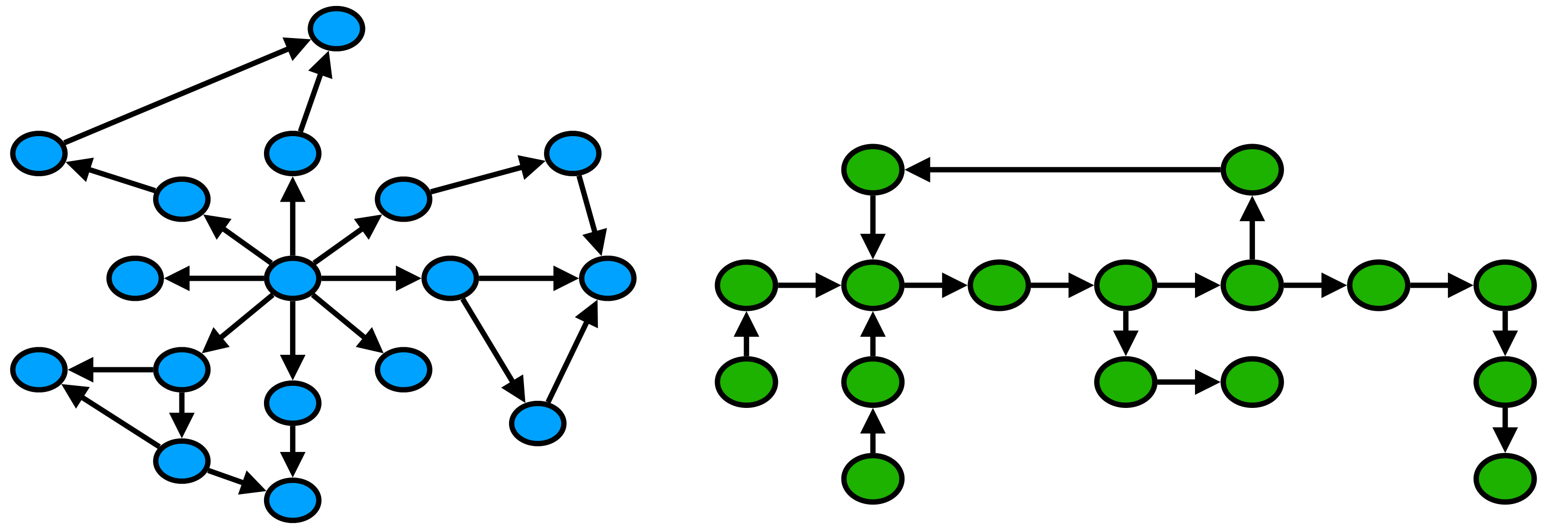
● Vertex-Parallel

● Bitvector

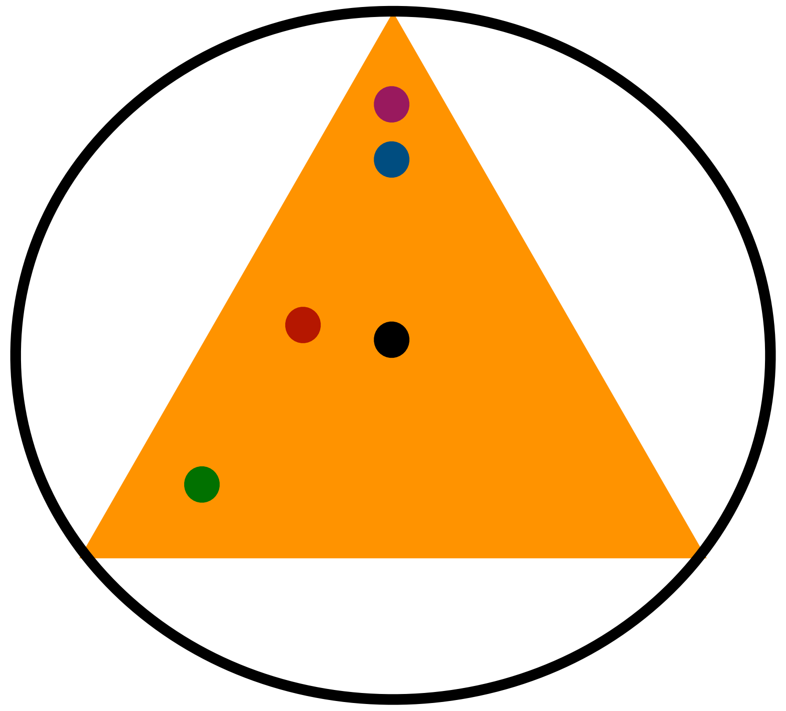
....



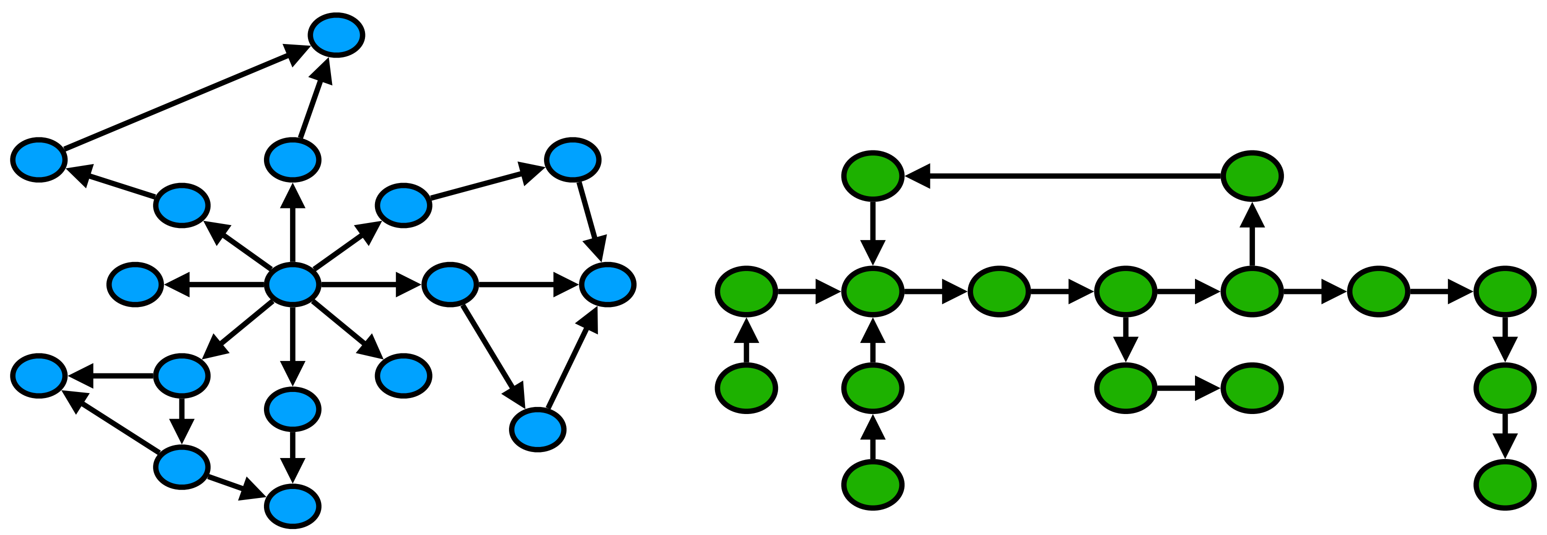
Optimizations



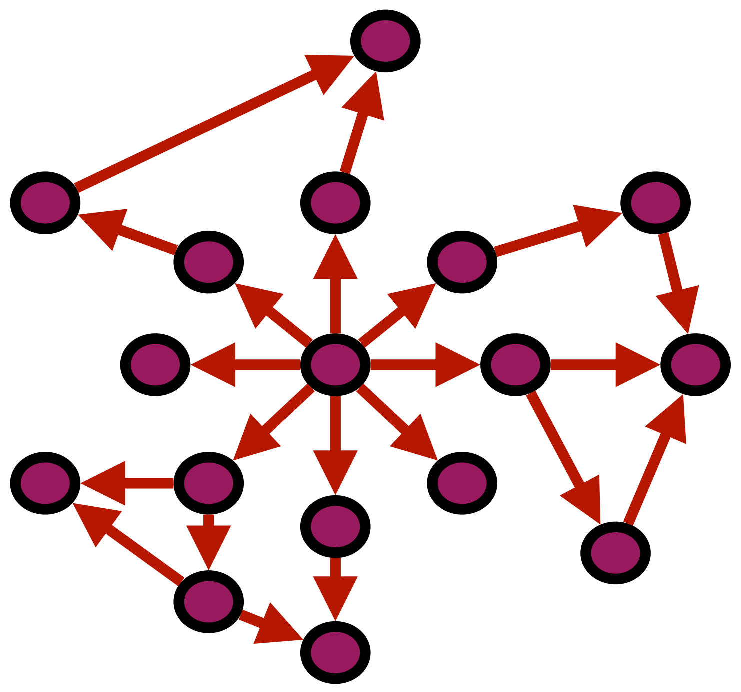
Graphs



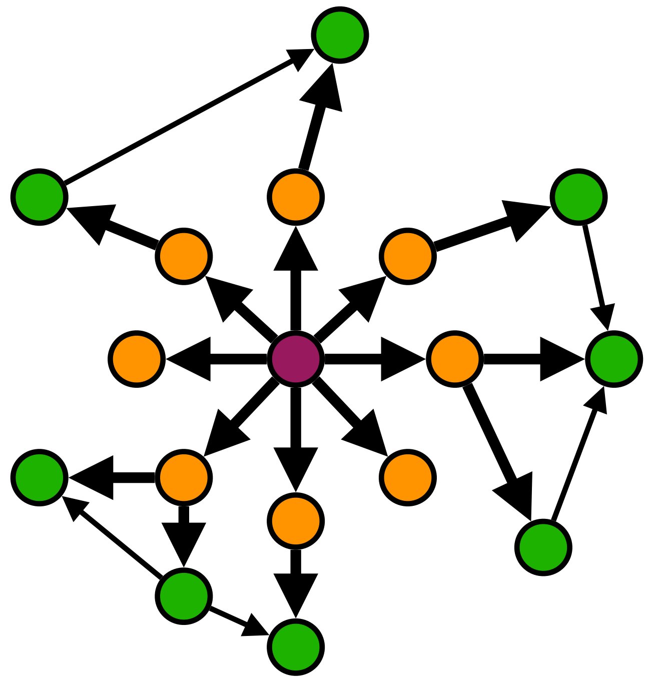
Optimizations

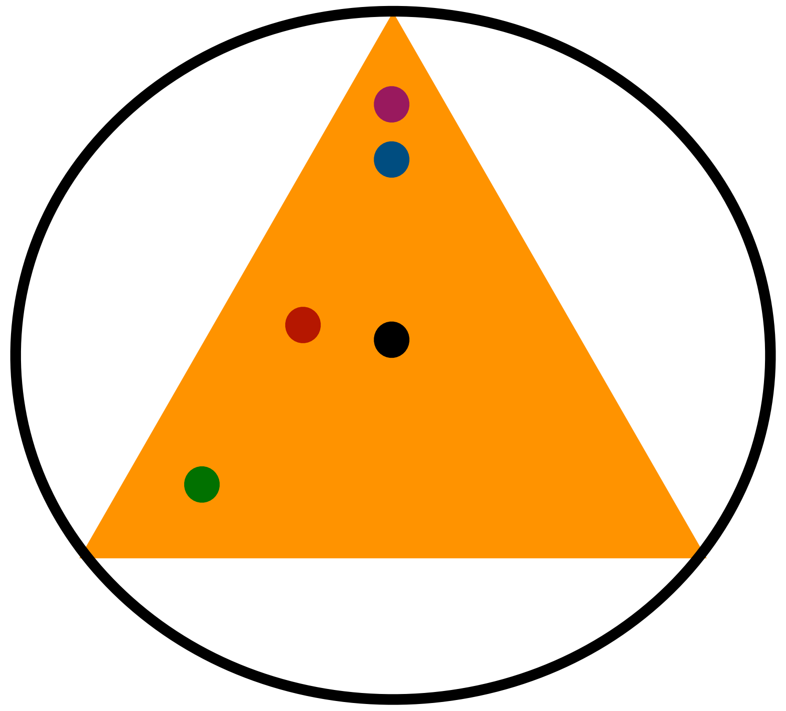


Graphs

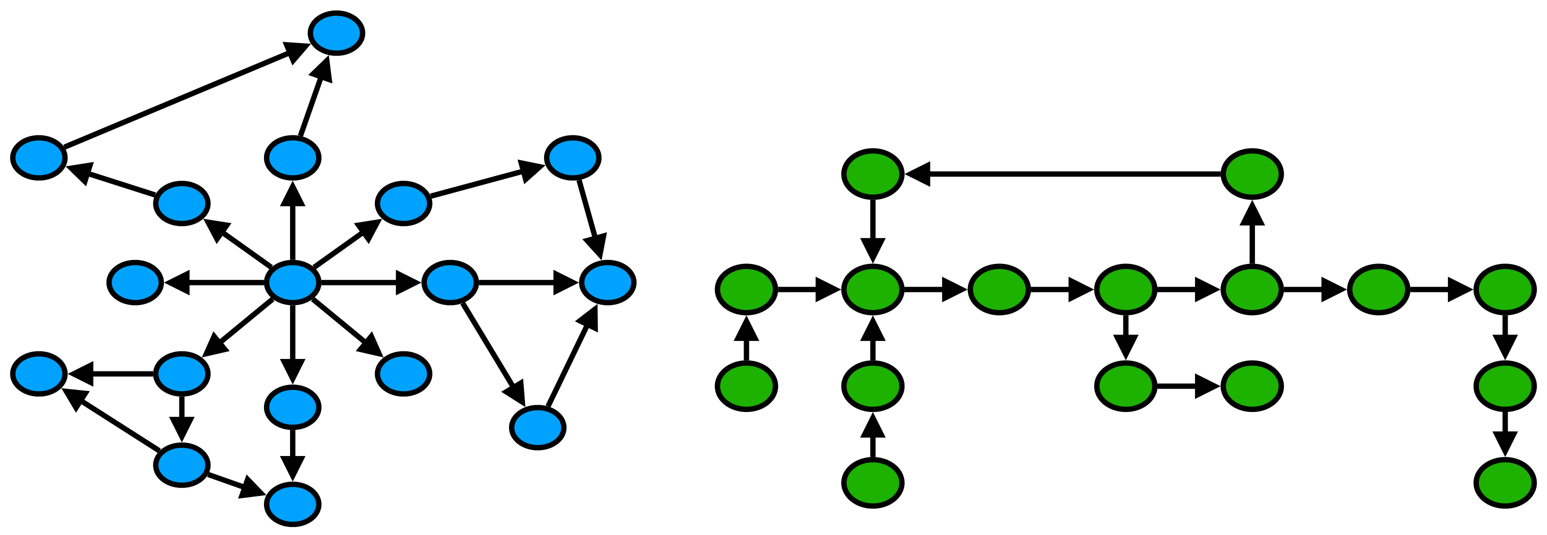


Algorithms

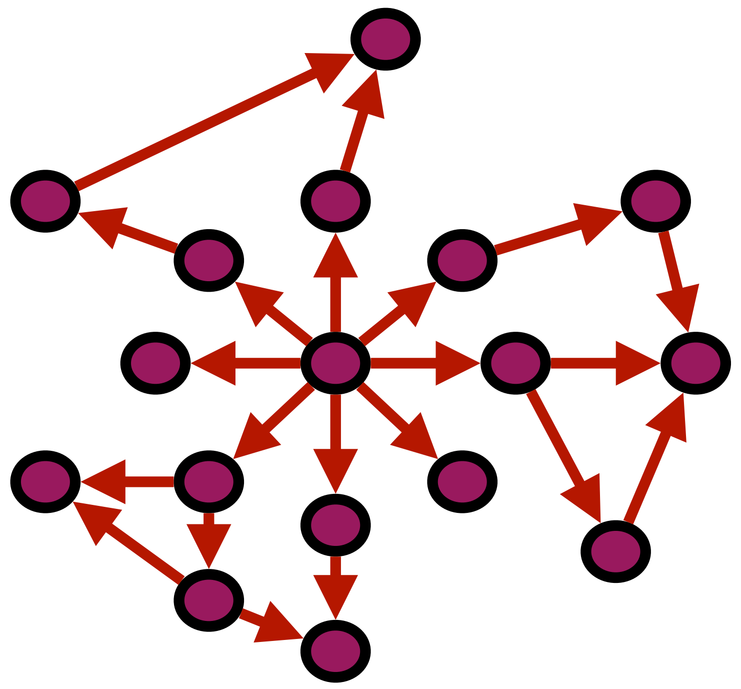




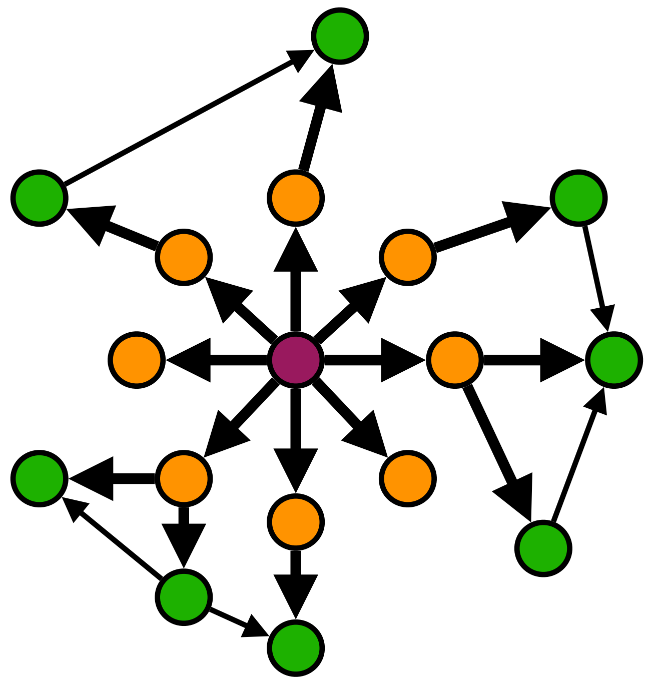
Optimizations

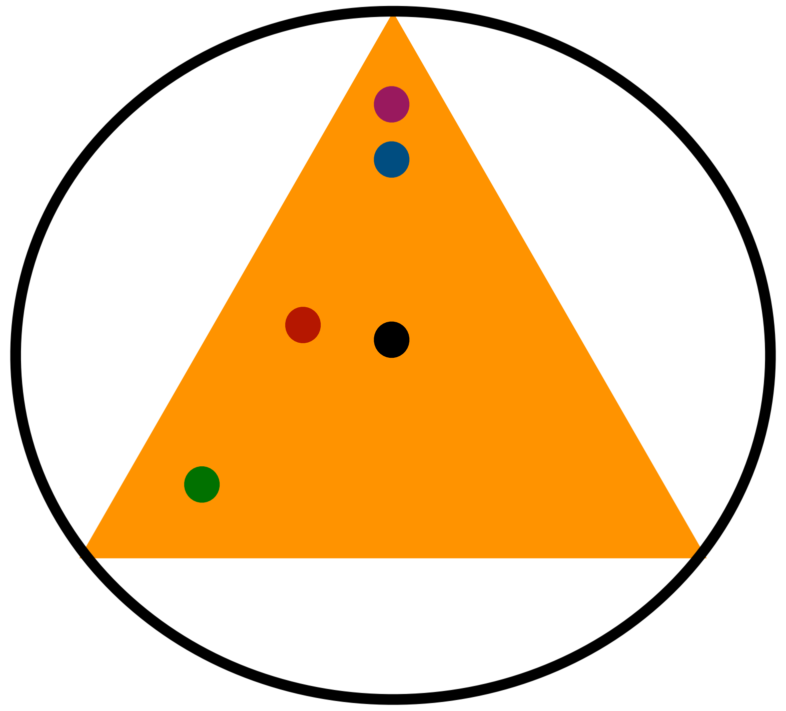


Graphs



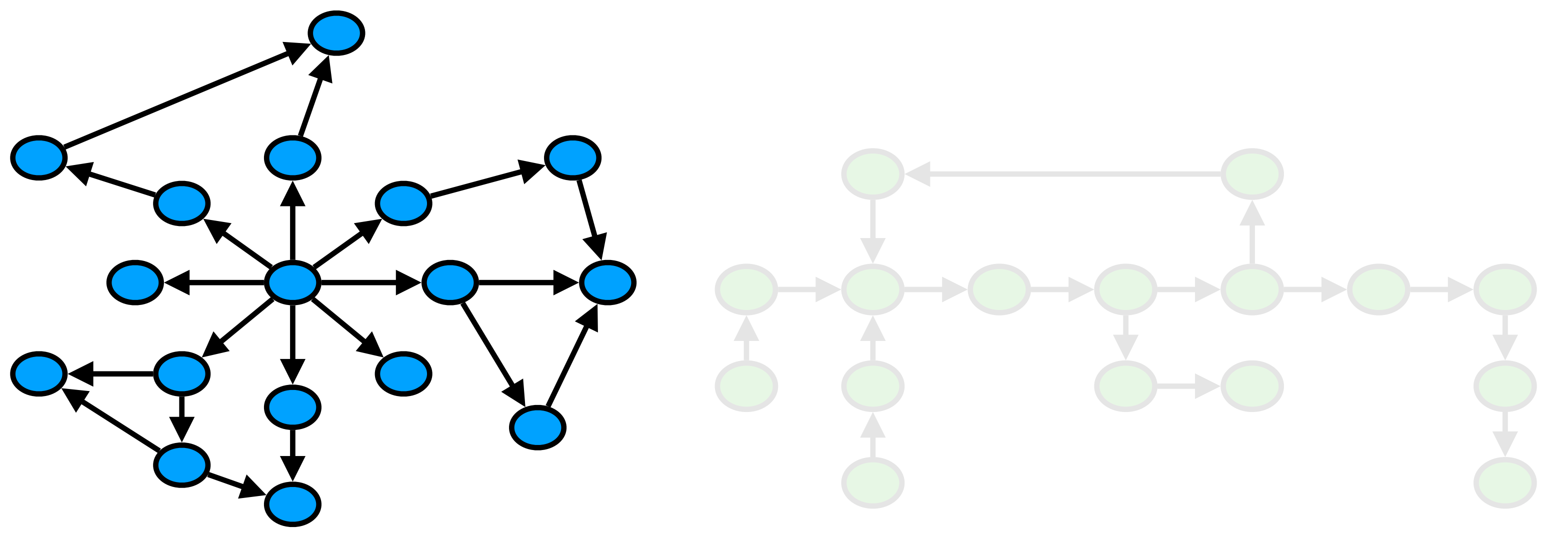
Algorithms





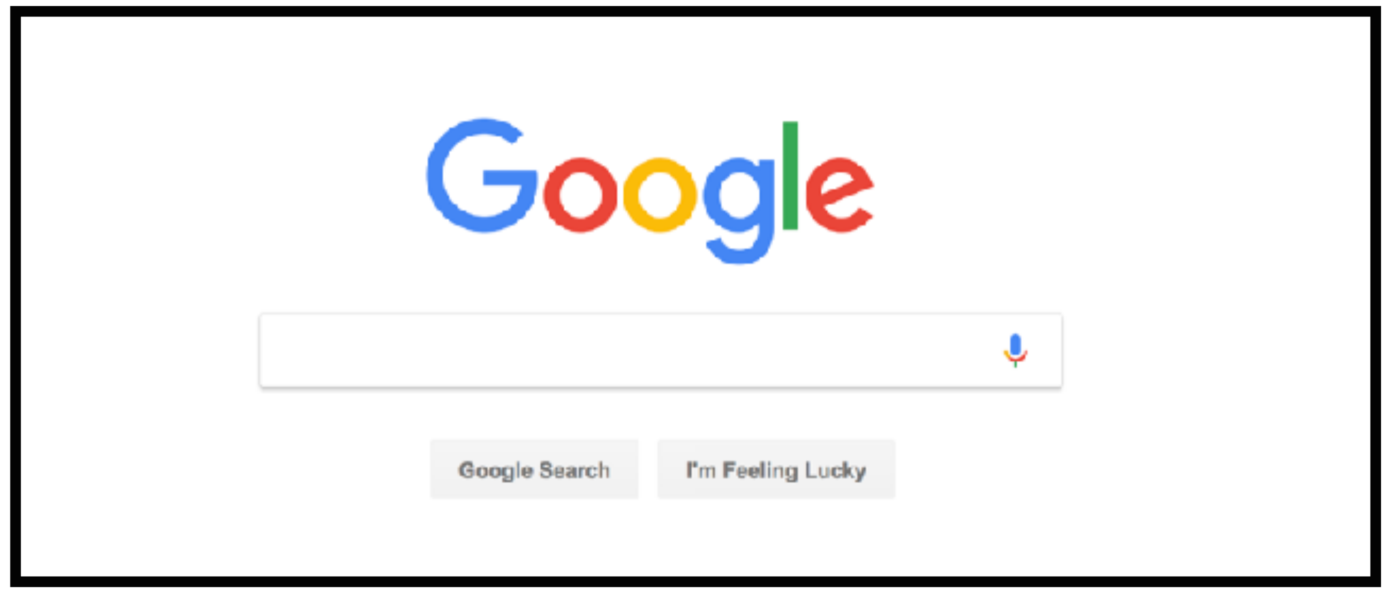
Optimizations

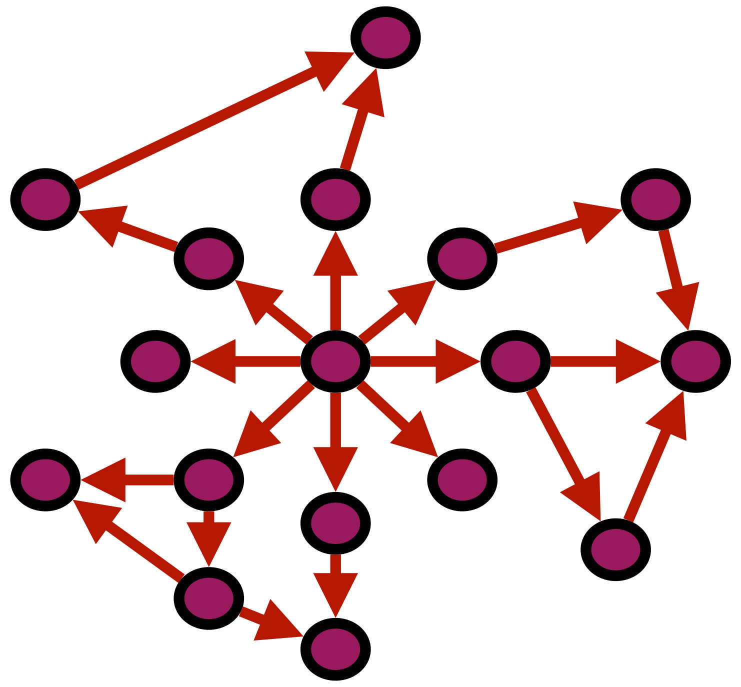
Hardware



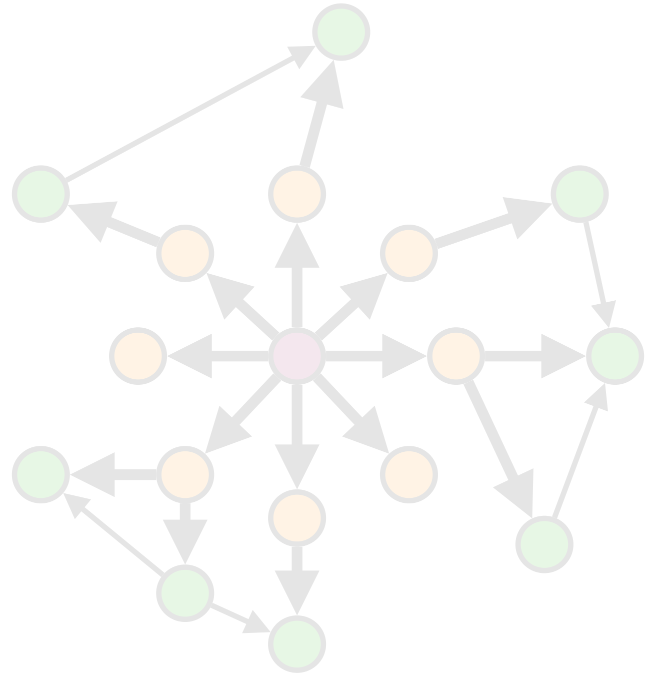
Graphs

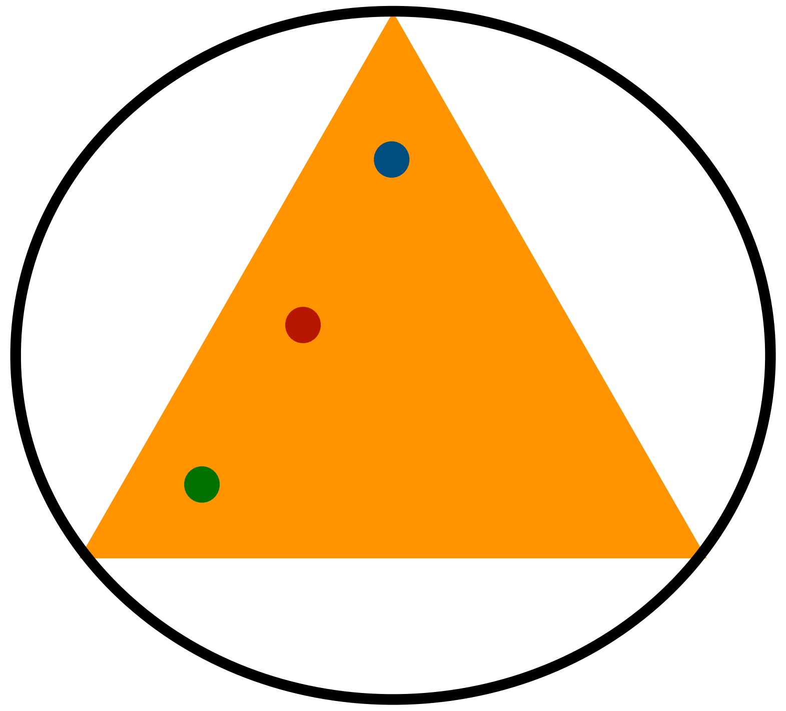
PageRank





Algorithms





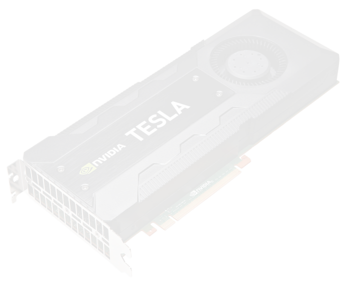
Optimizations

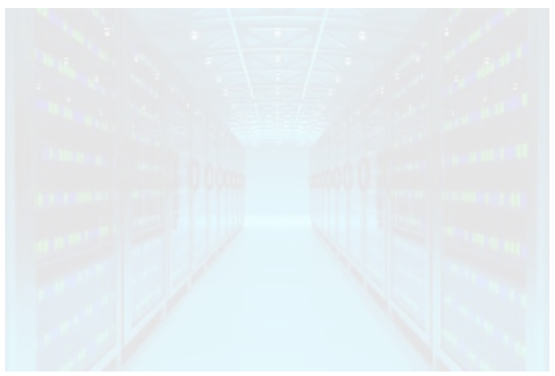
Pull

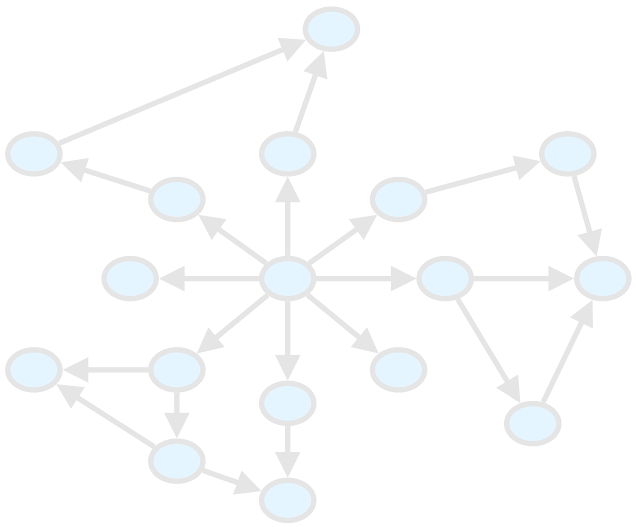
Partitioning

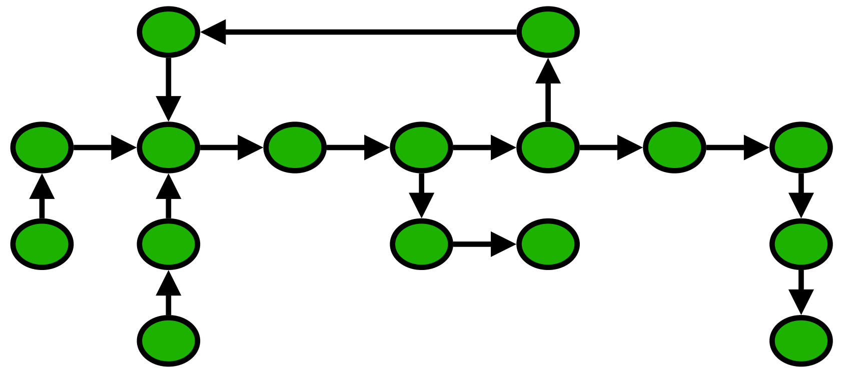
Vertex-Parallel

Hardware



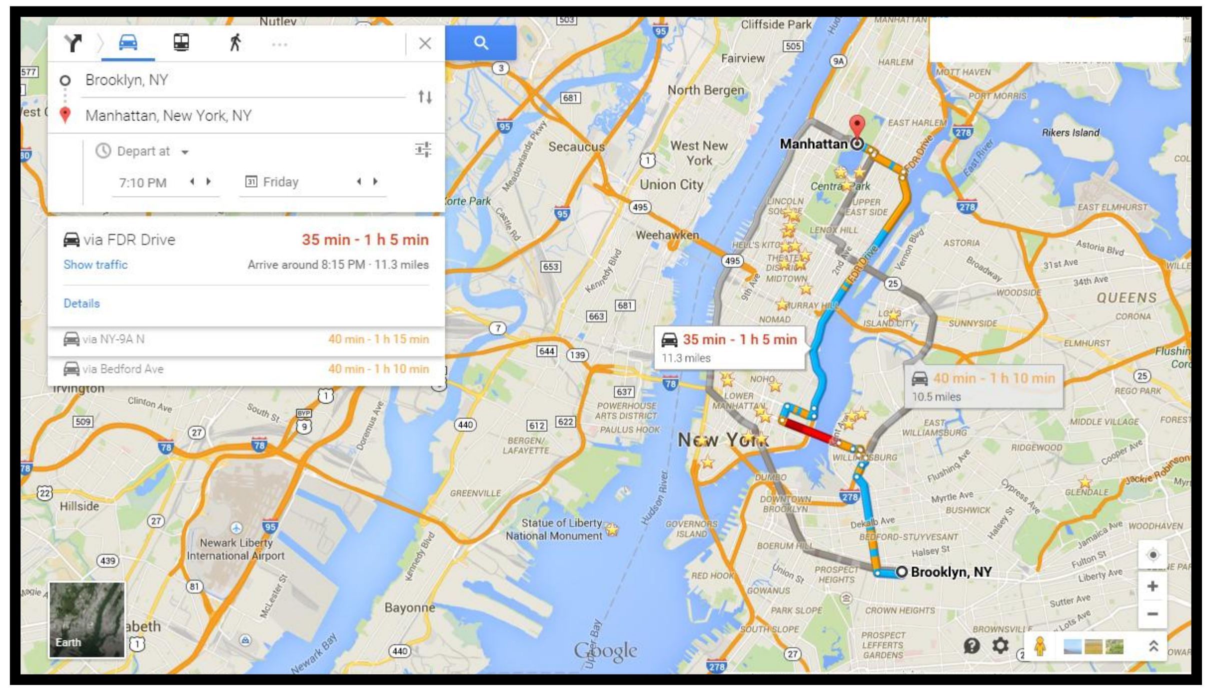


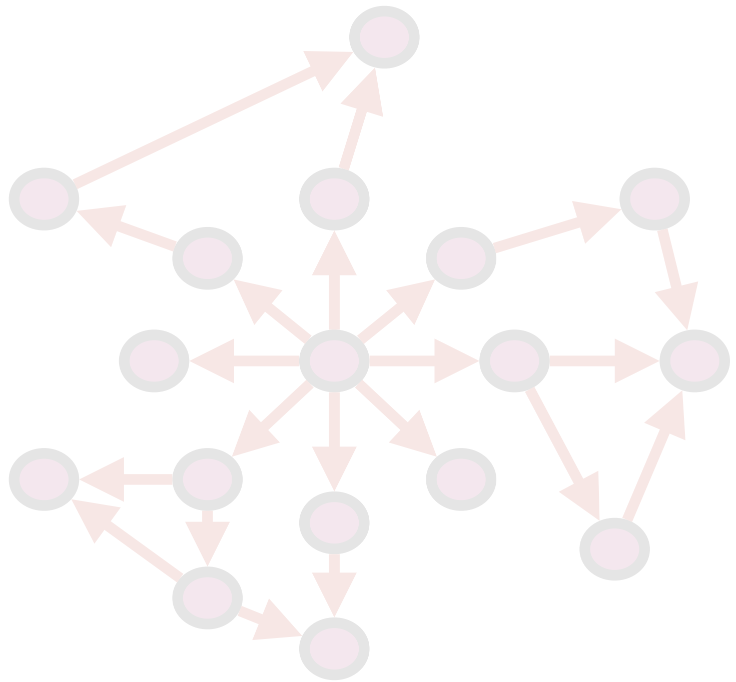




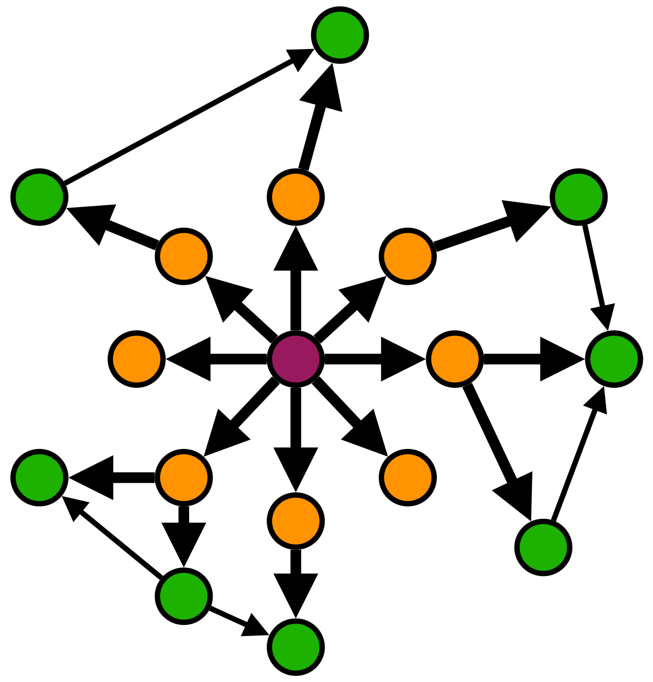
Graphs

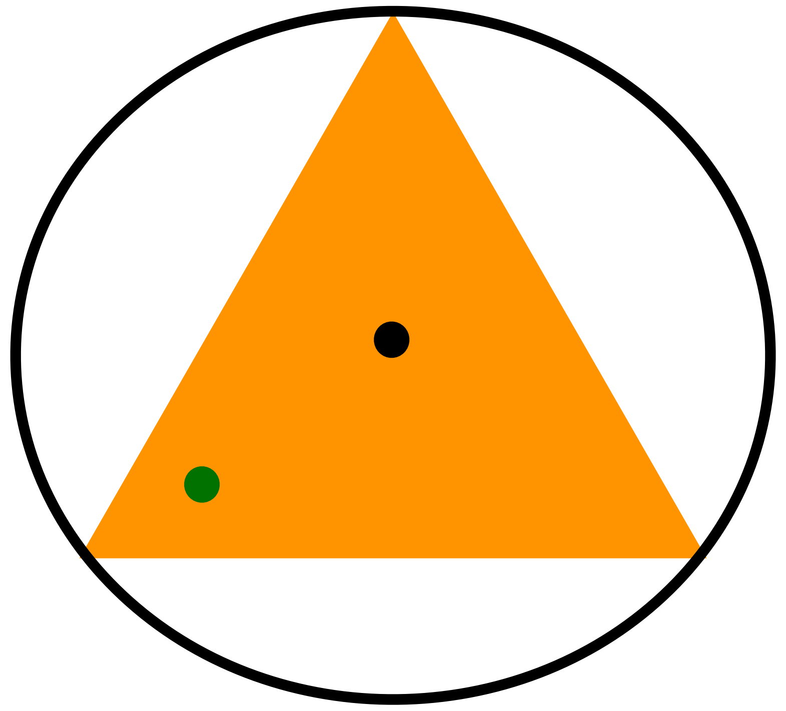
SSSP





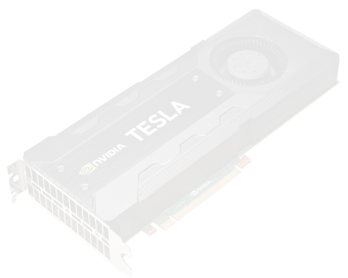
Algorithms

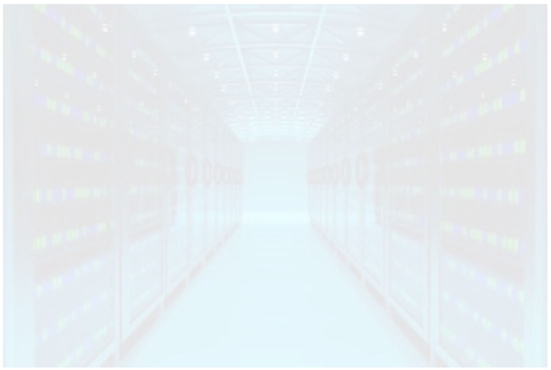




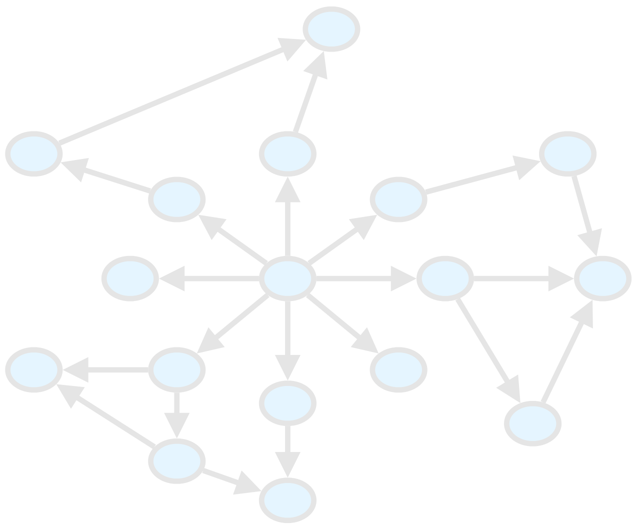
Optimizations

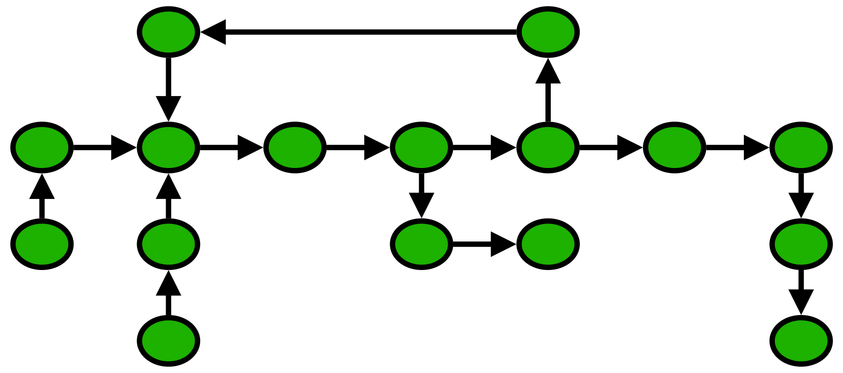
Hardware





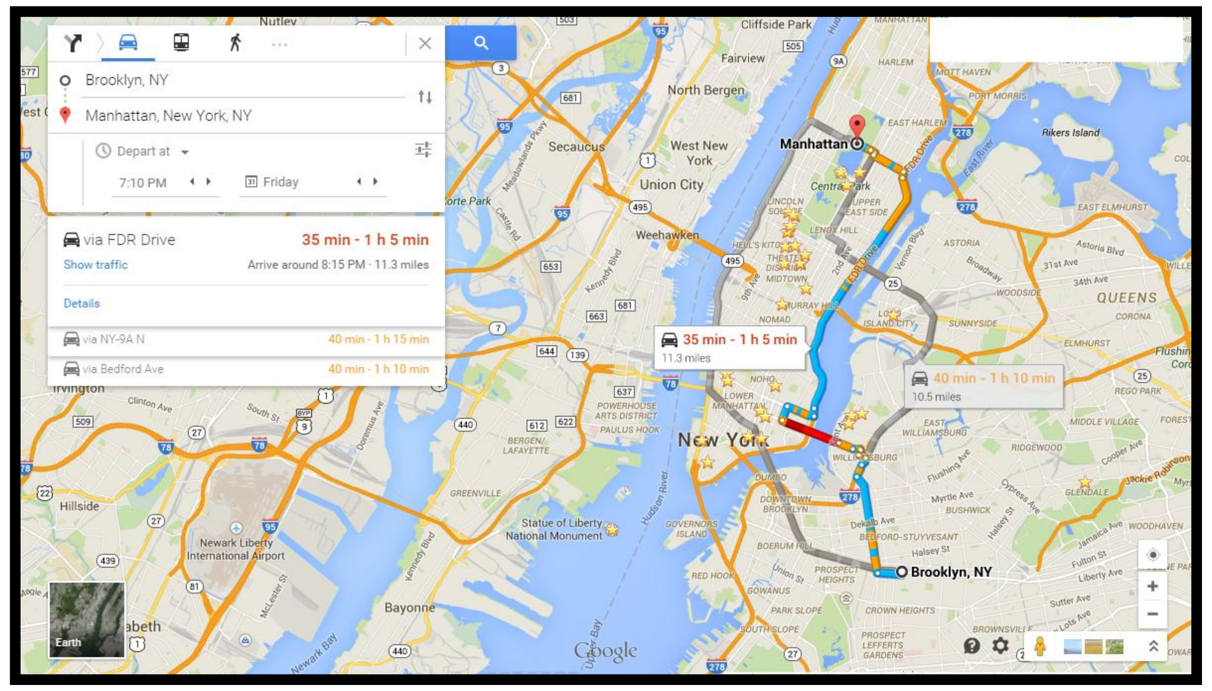
Push
Vertex-Parallel

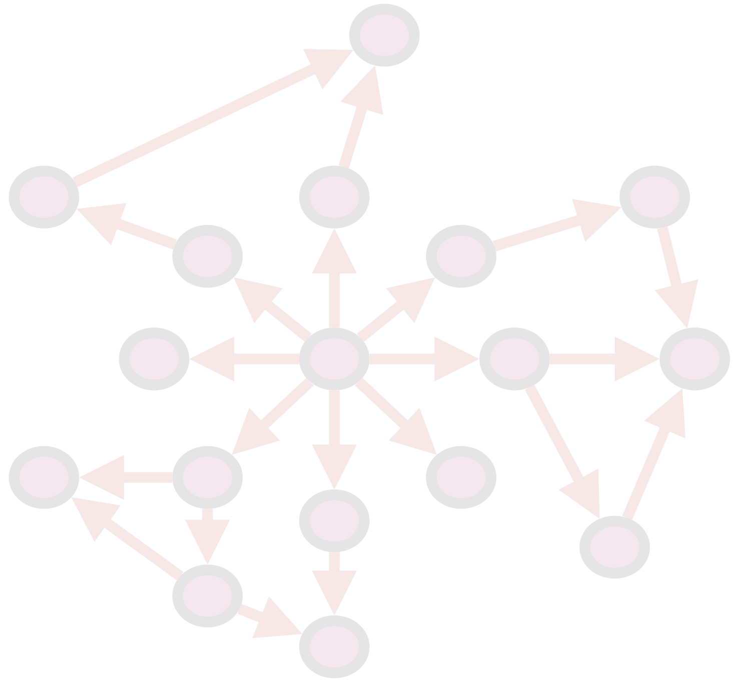




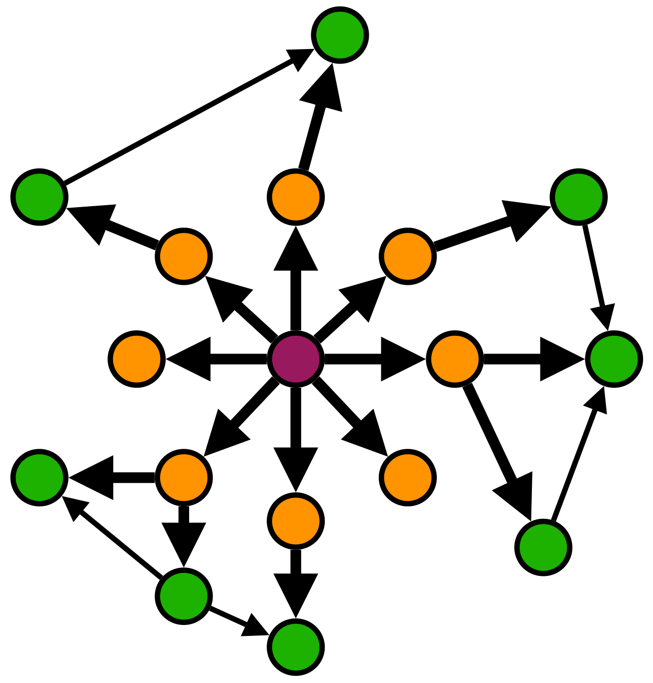
Graphs

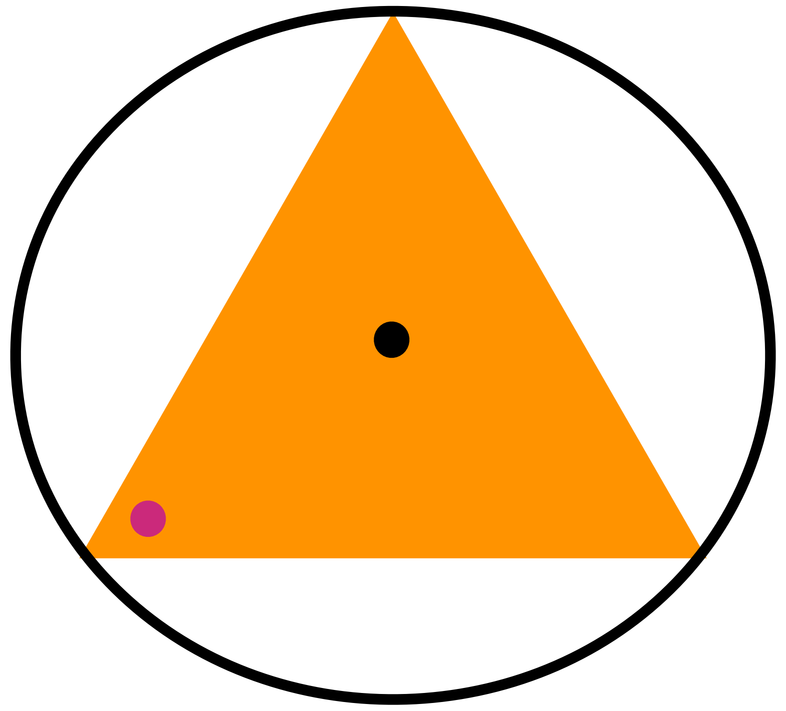
SSSP





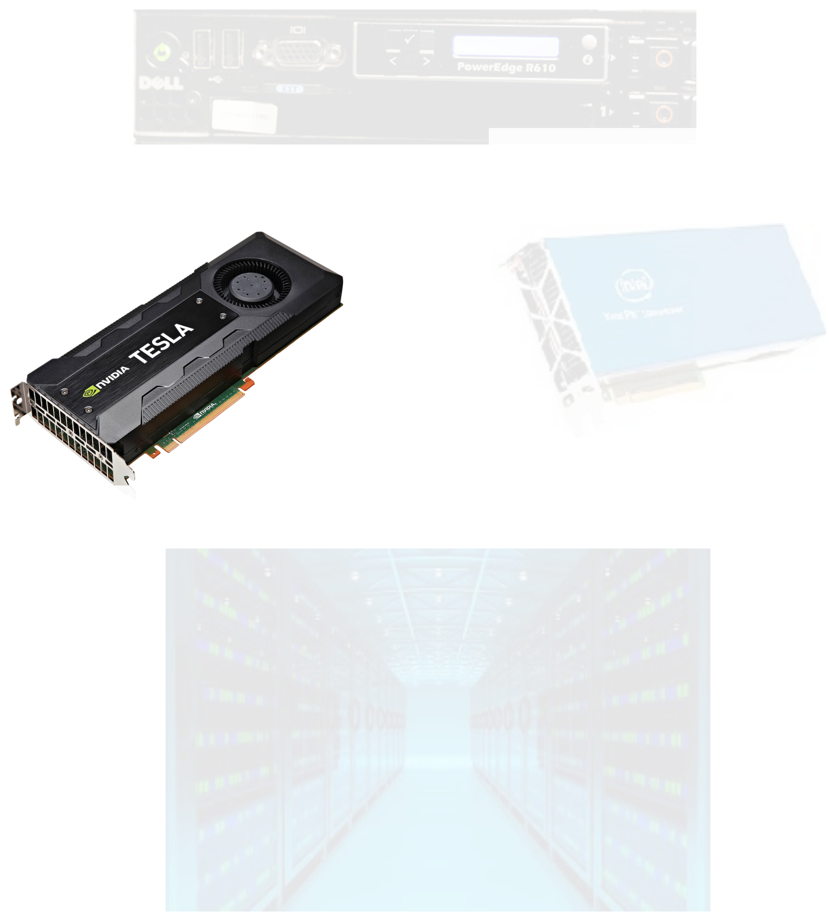
Algorithms





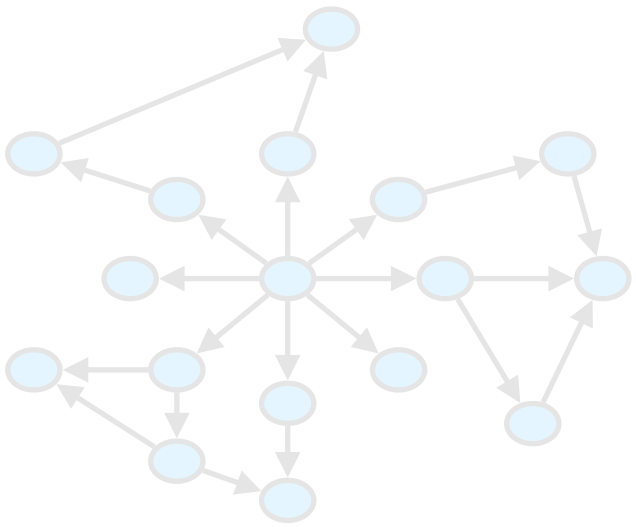
Optimizations

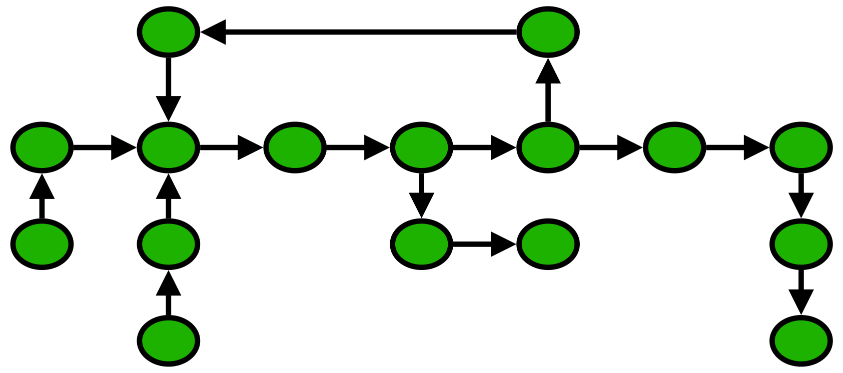
Hardware



Push

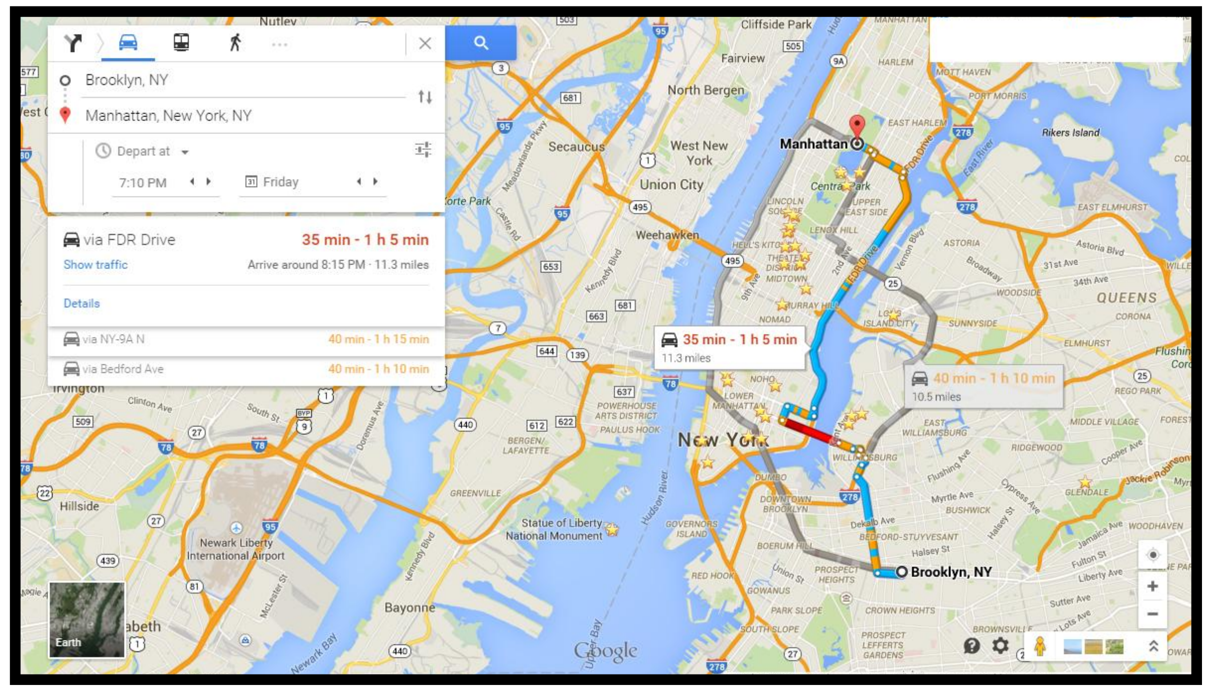
Edge-Parallel





Graphs

SSSP



Brooklyn, NY

Manhattan, New York, NY

Depart at

7:10 PM

Friday

via FDR Drive 35 min - 1 h 5 min

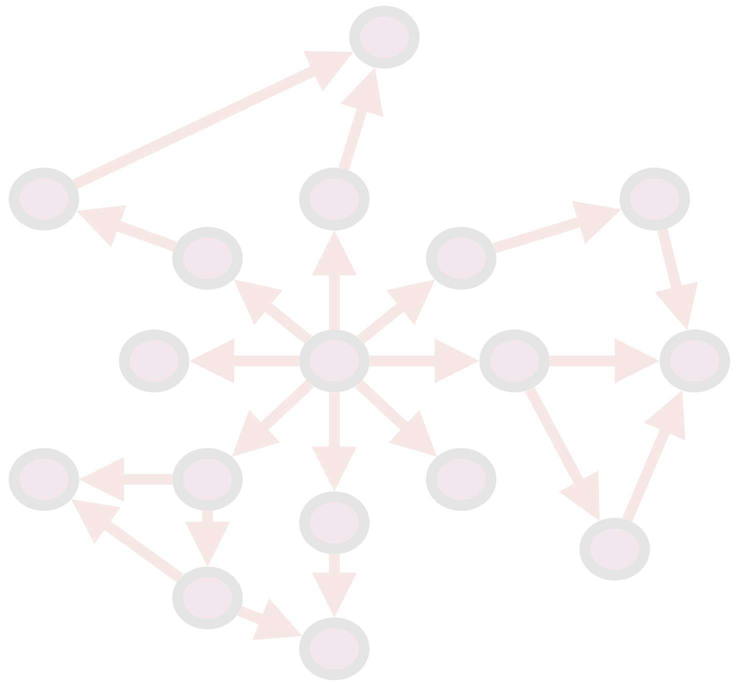
Show traffic

Arrive around 8:15 PM · 11.3 miles

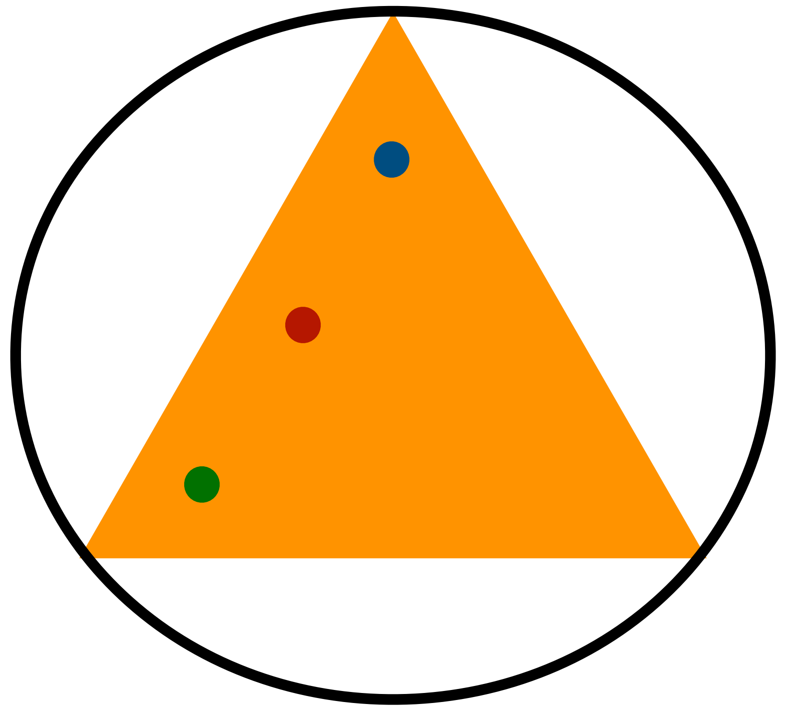
Details

via NY-9A N 40 min - 1 h 15 min

via Bedford Ave 40 min - 1 h 10 min

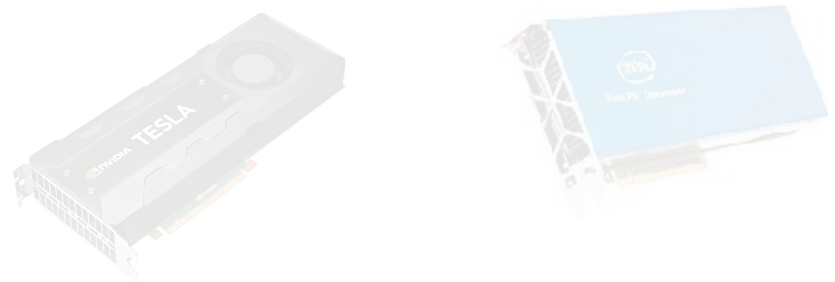


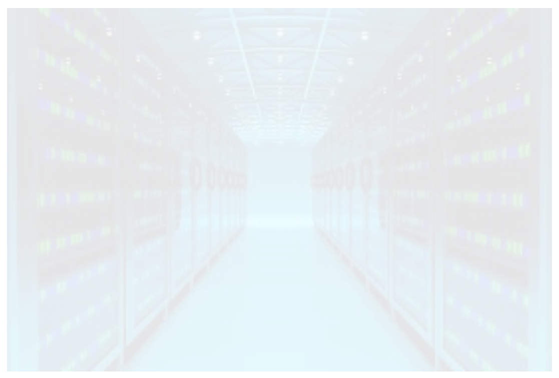
Algorithms

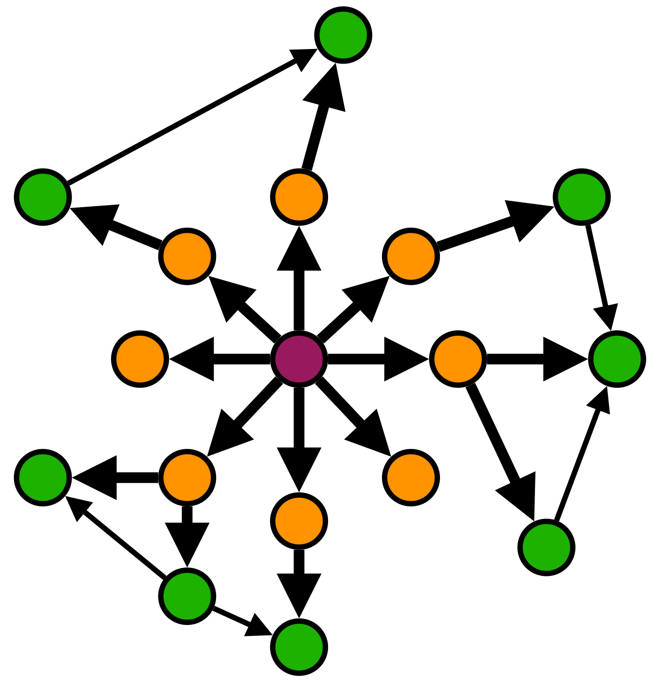


Optimizations

Hardware







Bad optimizations (schedules) can be > 100x slower

Pull

Partitioning

Vertex-Parallel

GraphIt

A Domain-Specific Language for Graph Applications

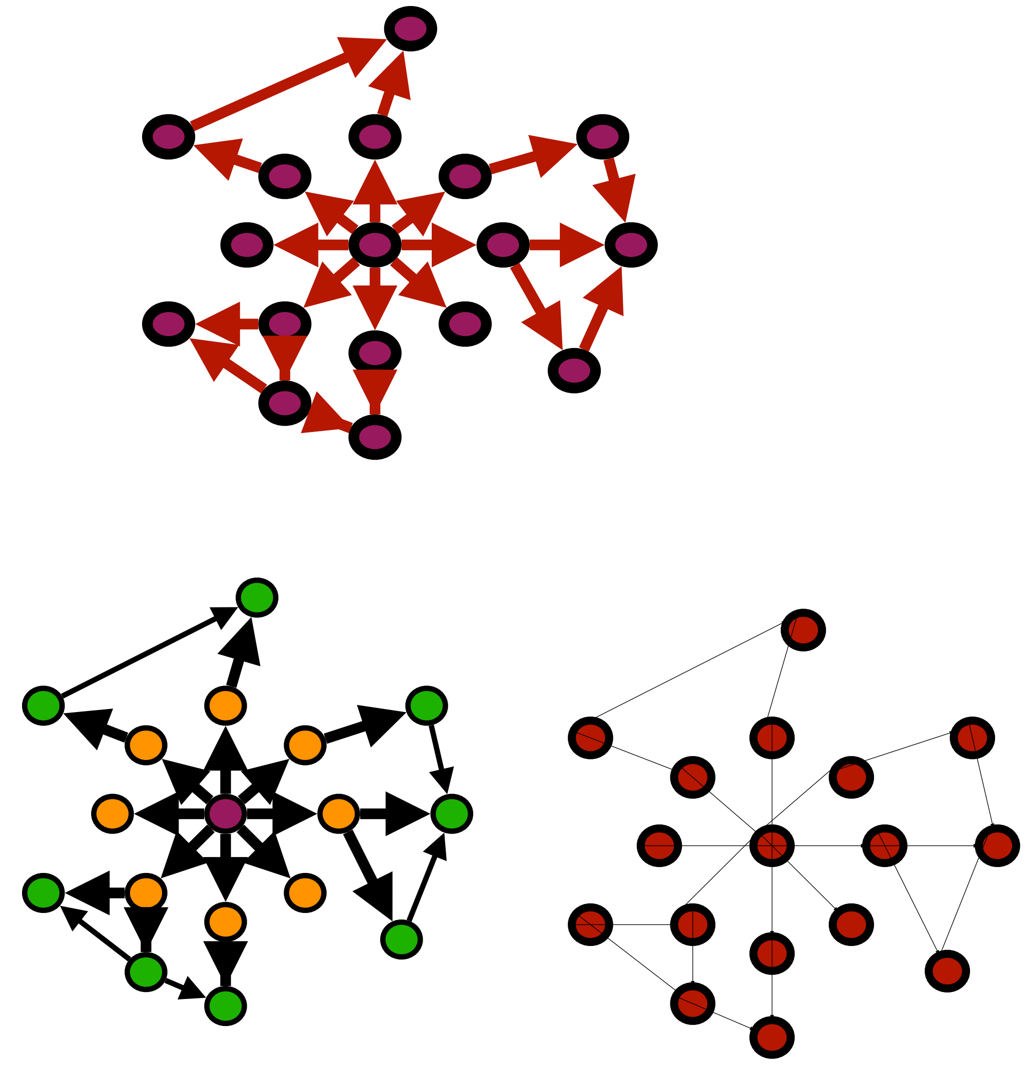
- **Decouple algorithm from optimization for graph applications**
  - **Algorithm**: What to Compute
  - **Optimization (schedule)**: How to Compute

GraphIt

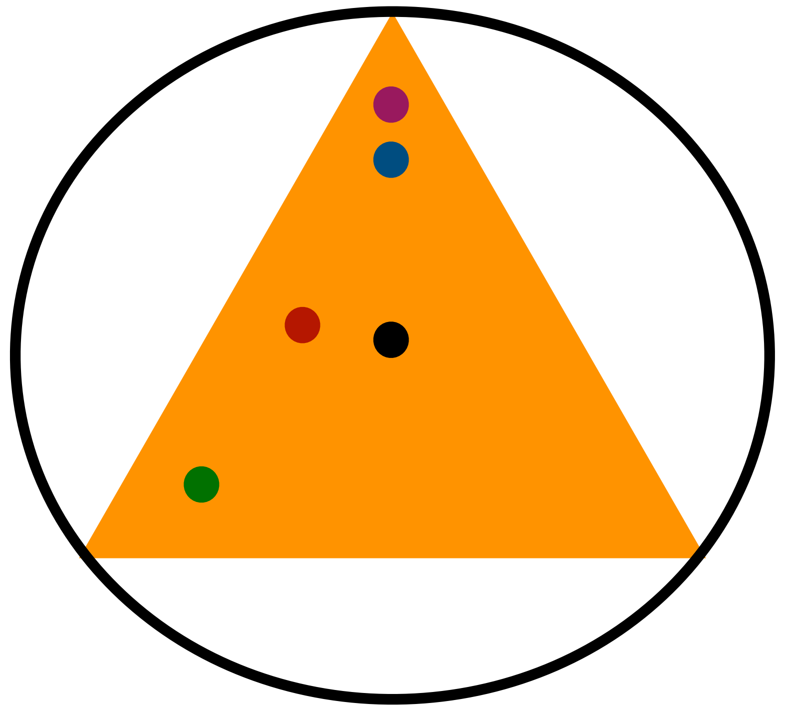
A Domain-Specific Language for Graph Applications

- **Decouple algorithm from optimization for graph applications**
  - **Algorithm**: What to Compute
  - **Optimization (schedule)**: How to Compute
- **DSL and Compiler**
  - **Ease-of-Use:** Improve Productivity for Data Scientists and Library Developers
  - **Performance:** Beat hand-optimized libraries by up to 4.8x on CPU and GPU
  - **Portability:** Working with NVIDIA, UW, Cornell to develop new backends for GPU and Domain-Specific Accelerators

GraphIt DSL

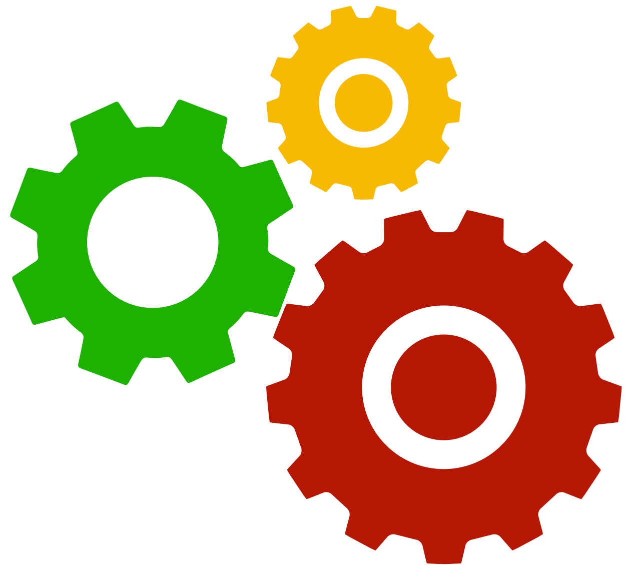


Algorithm Representation
(Algorithm Language)



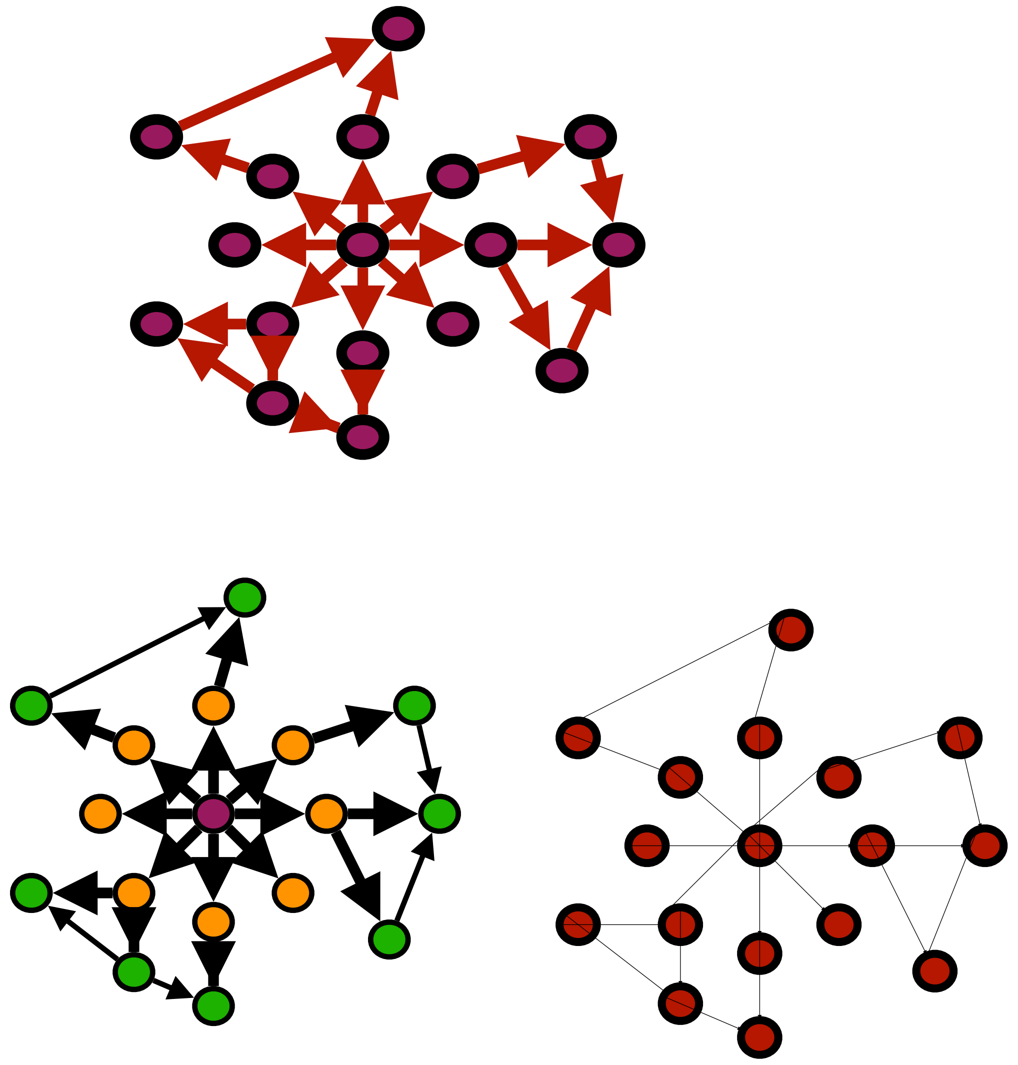
Optimization Representation

- **Scheduling Language**
- **Schedule Representation (e.g. Graph Iteration Space)**

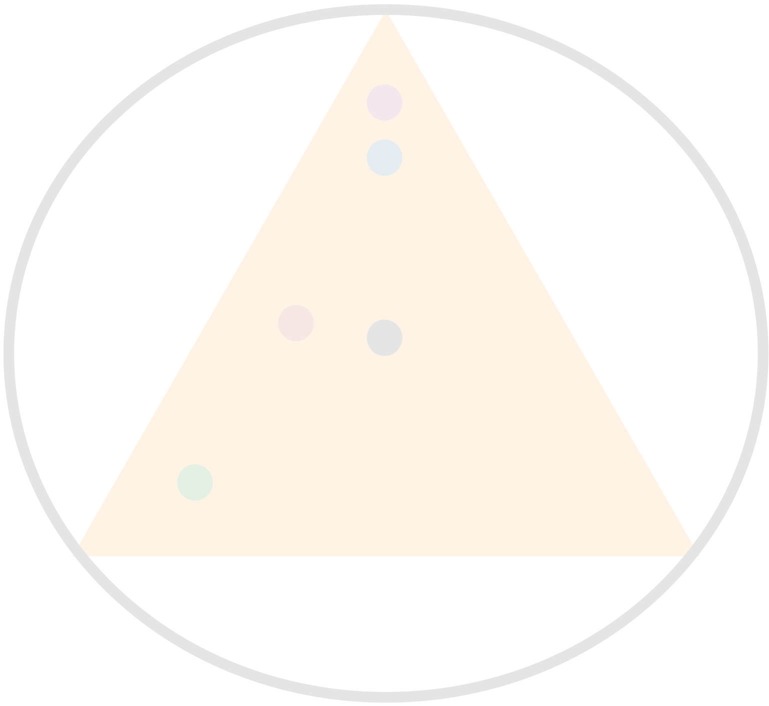


Autotuner

GraphIt DSL

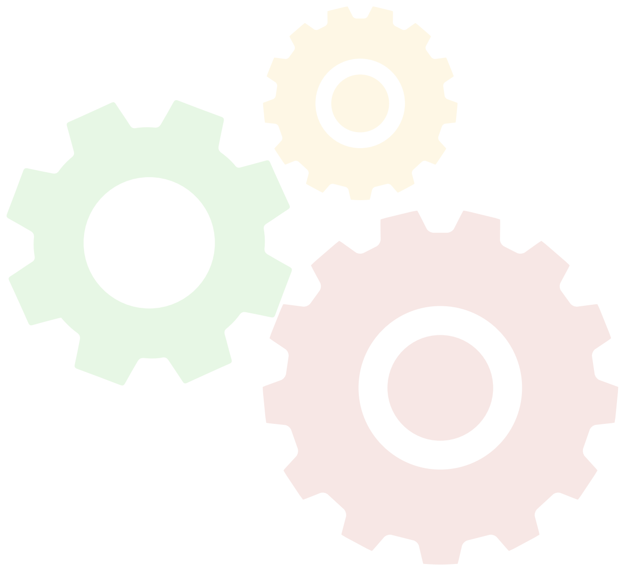


Algorithm Representation (Algorithm Language)



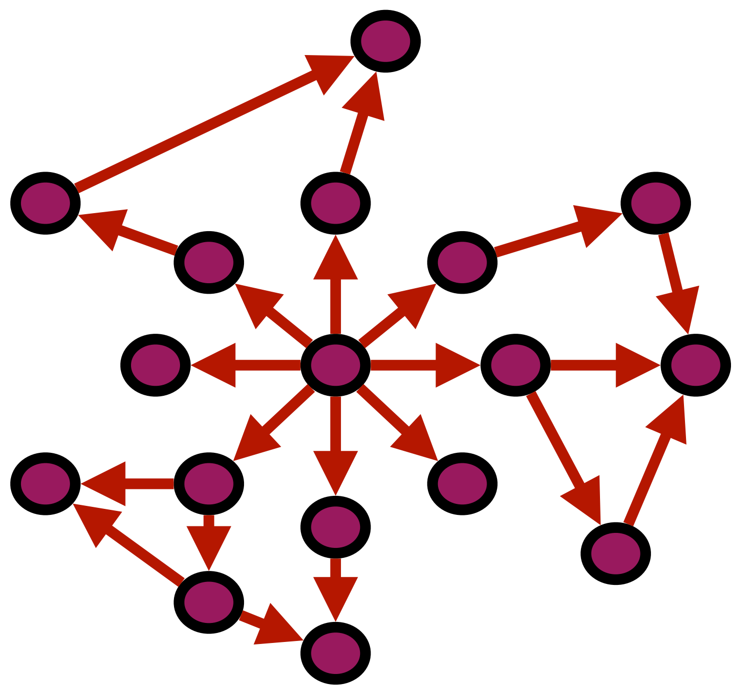
Optimization Representation

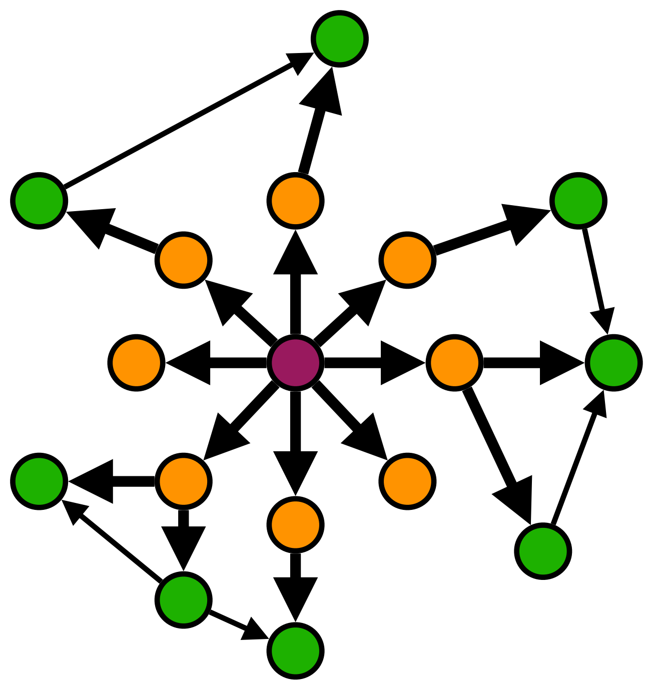
- Scheduling Language
- Schedule Representation (e.g. Graph Iteration Space)

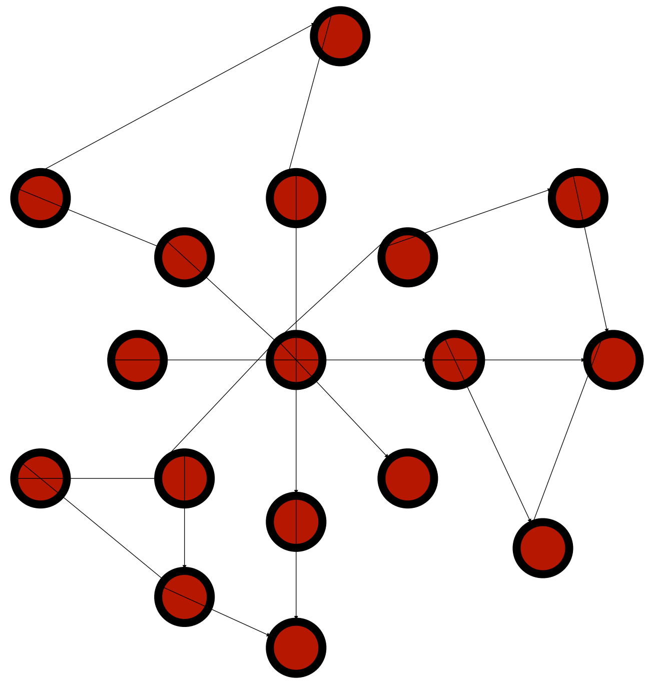


Autotuner

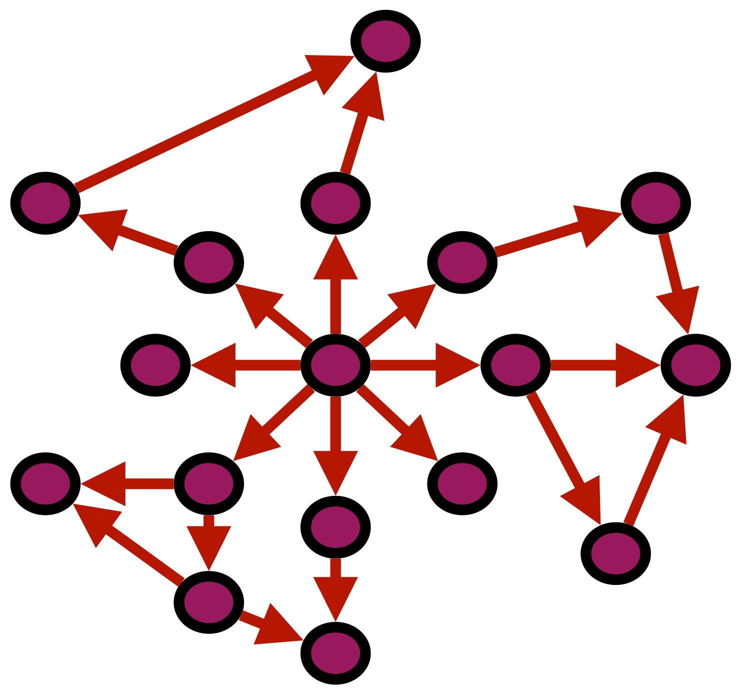
Algorithm Language



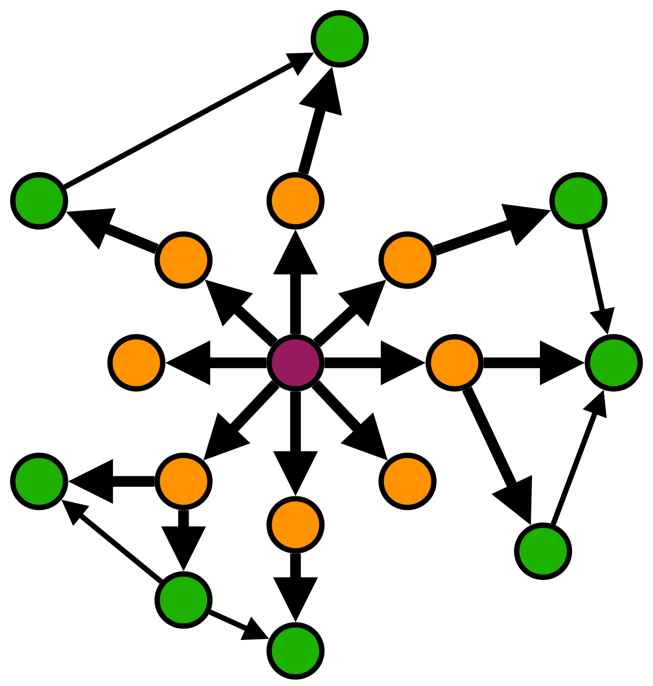


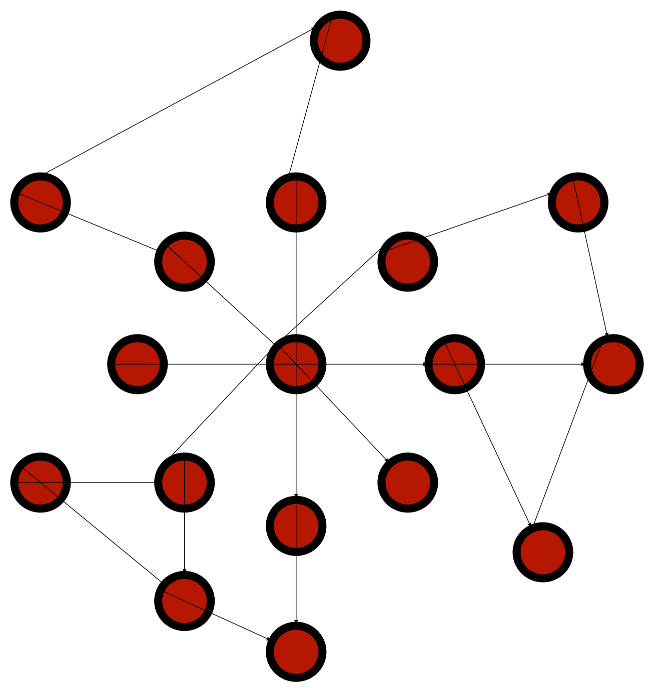


Algorithm Language

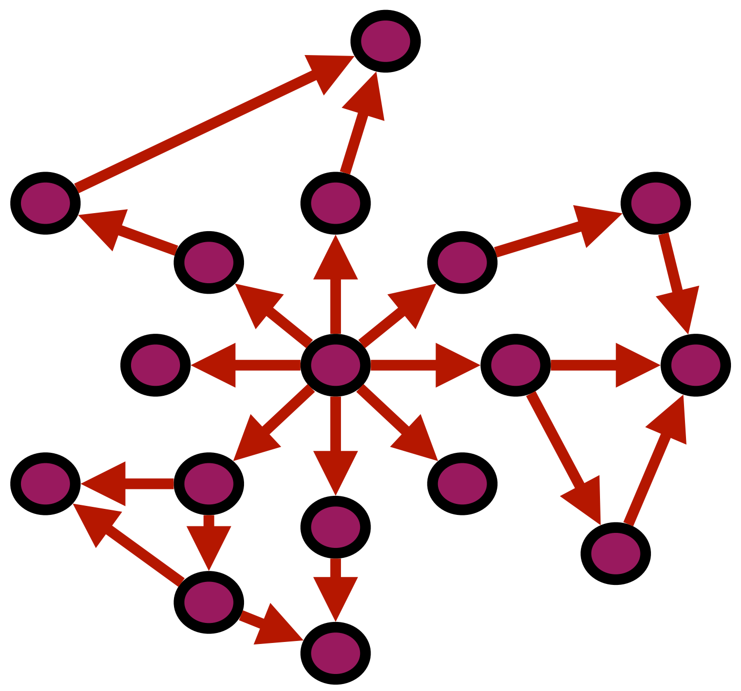


edges.apply(func)

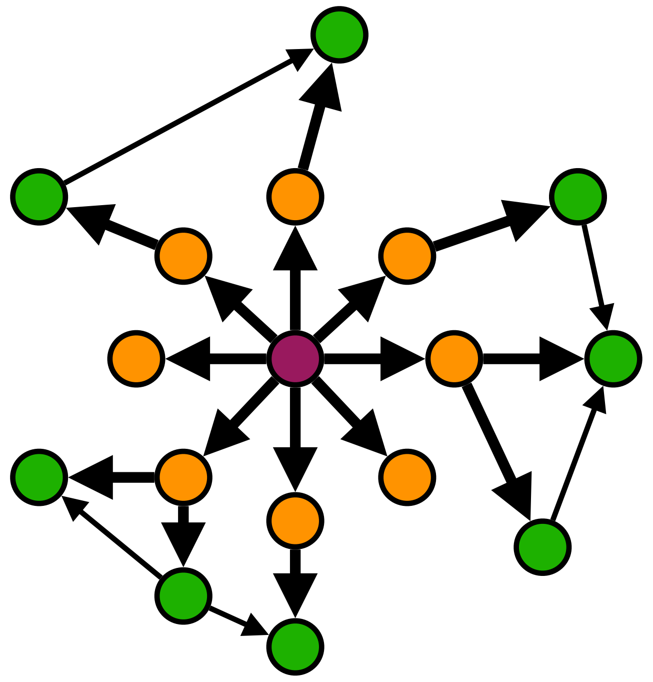




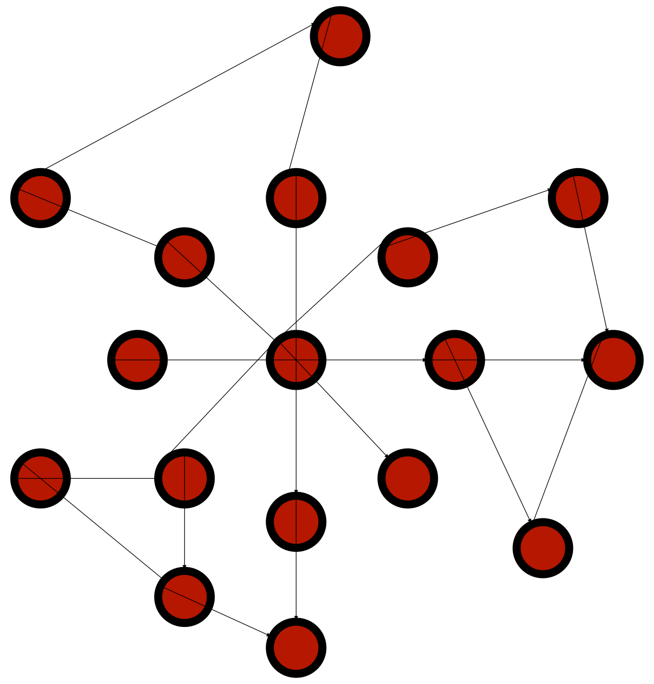
Algorithm Language



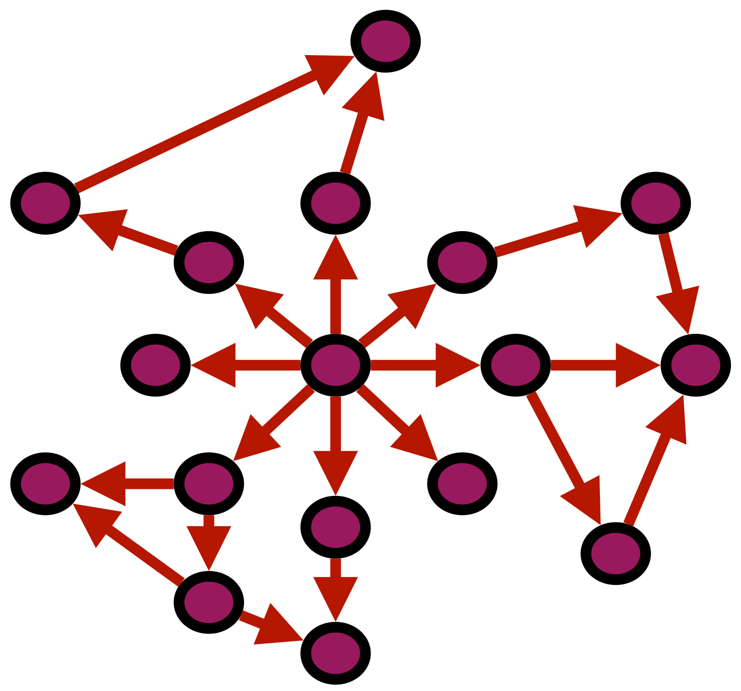
edges.apply(func)



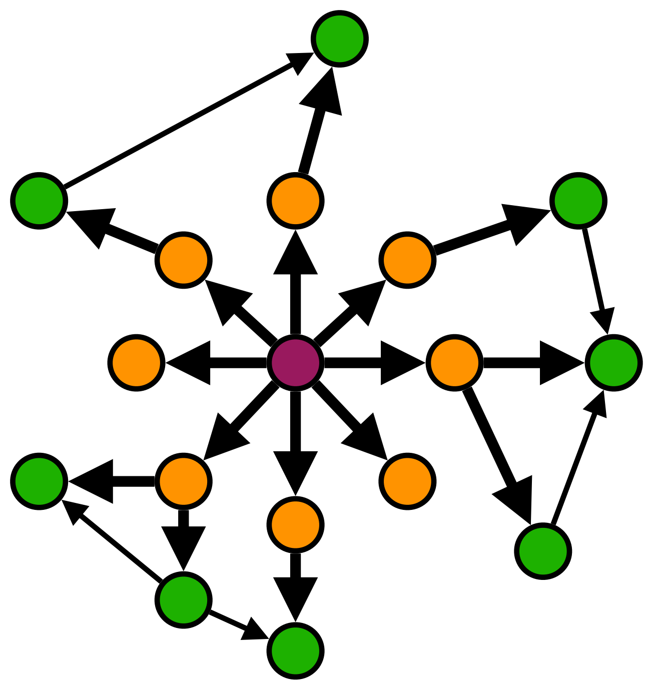
```
edges.from(vertexset)
     .to(vertexset)
     .srcFilter(func)
     .dstFilter(func)
     .apply(func)
```



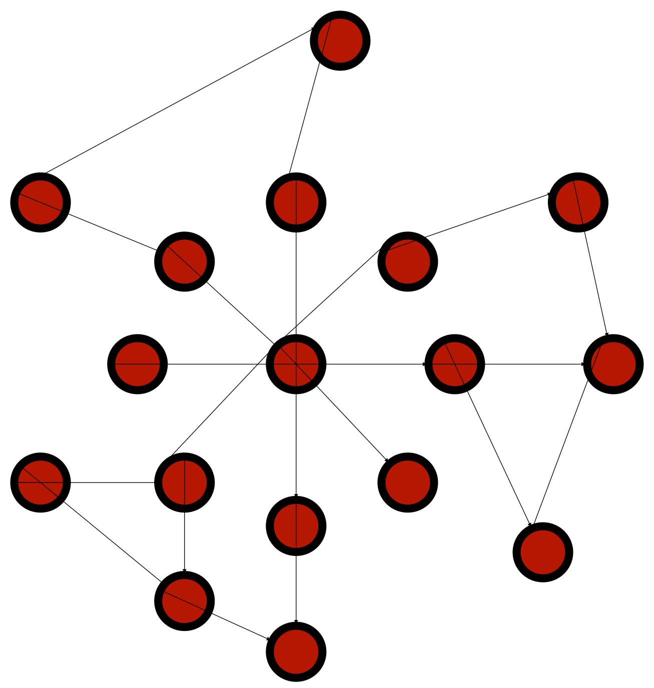
Algorithm Language



edges.apply(func)

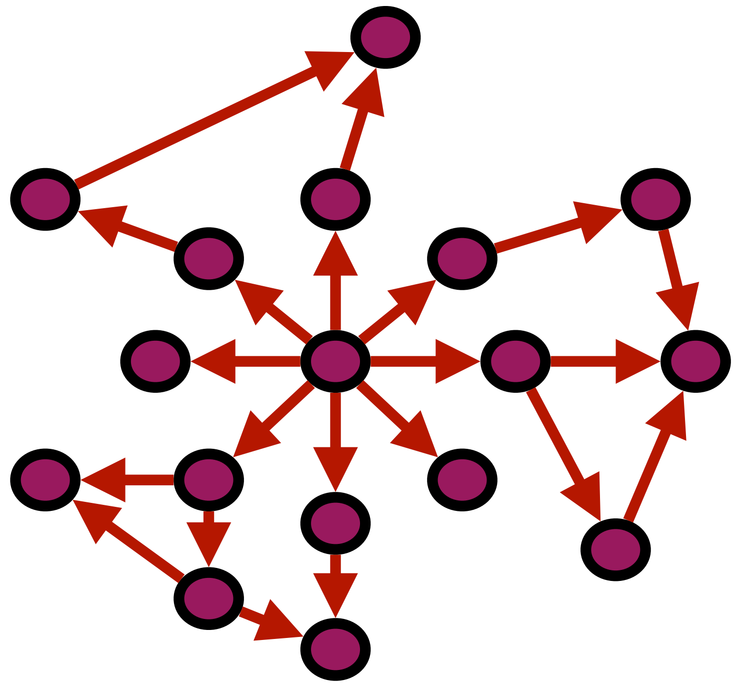


edges.from(vertexset)
.to(vertexset)
.srcFilter(func)
.dstFilter(func)
.apply(func)



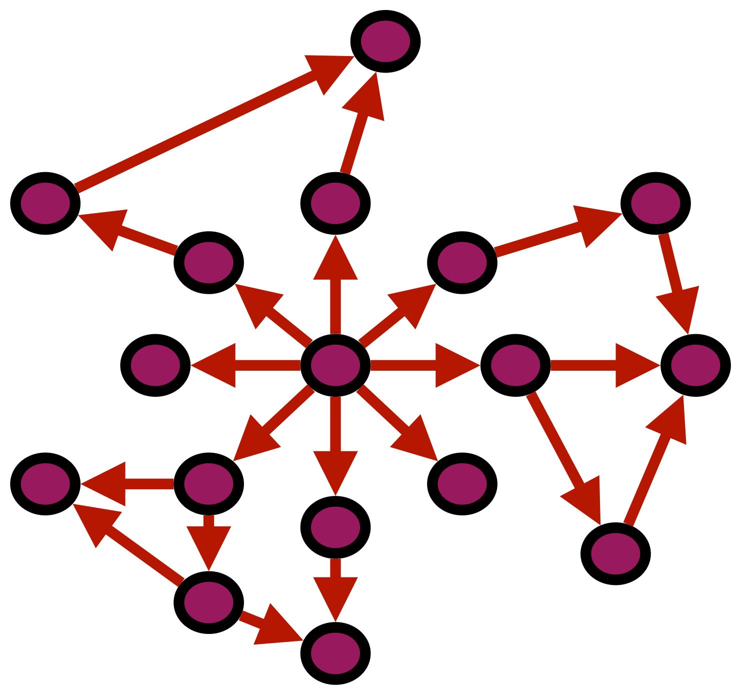
vertices.apply(func)

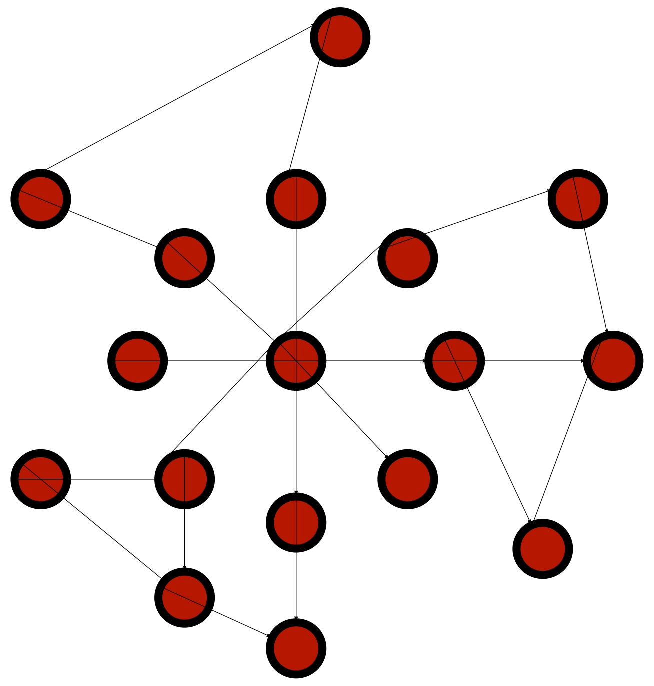
PageRank Example



```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end
```

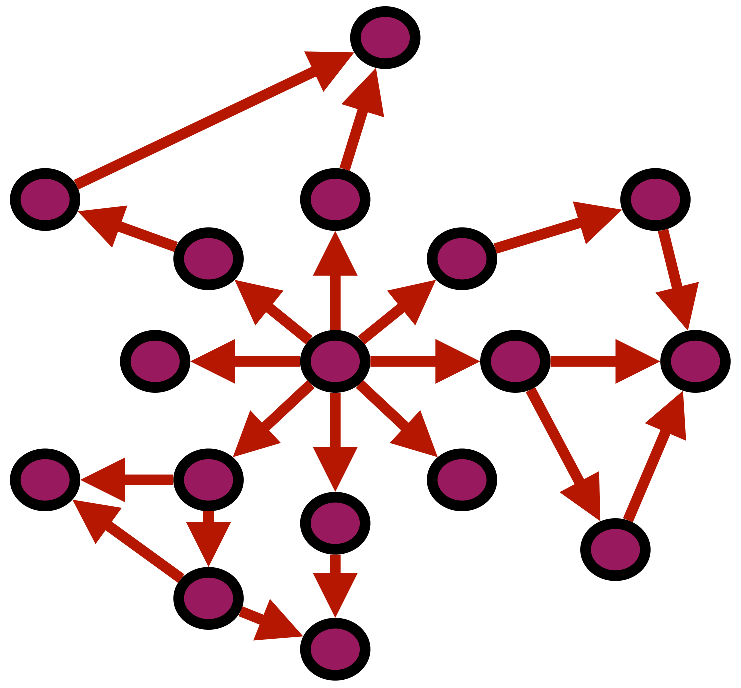
PageRank Example

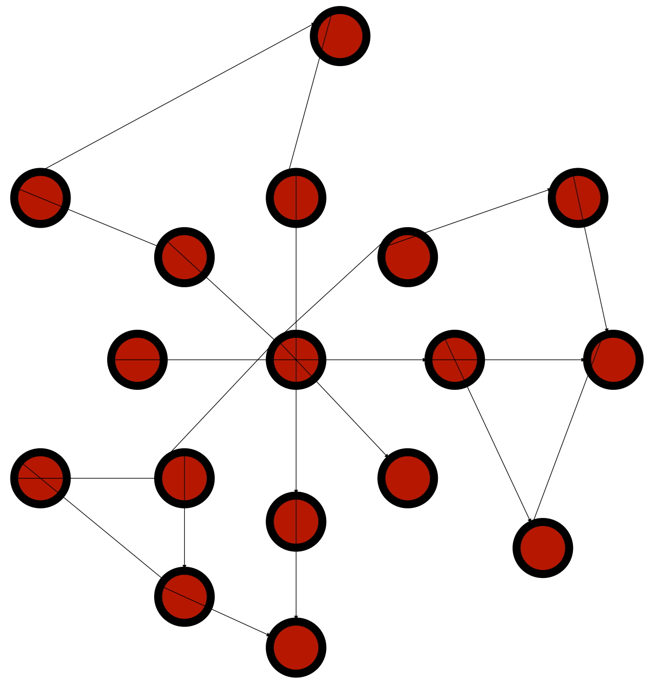




```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end


func updateVertex (v: Vertex)
   new_rank[v] = base_score + 0.85*new_rank[v];
   old_rank[v] = new_rank[v];
   new_rank[v] = 0;
end
```

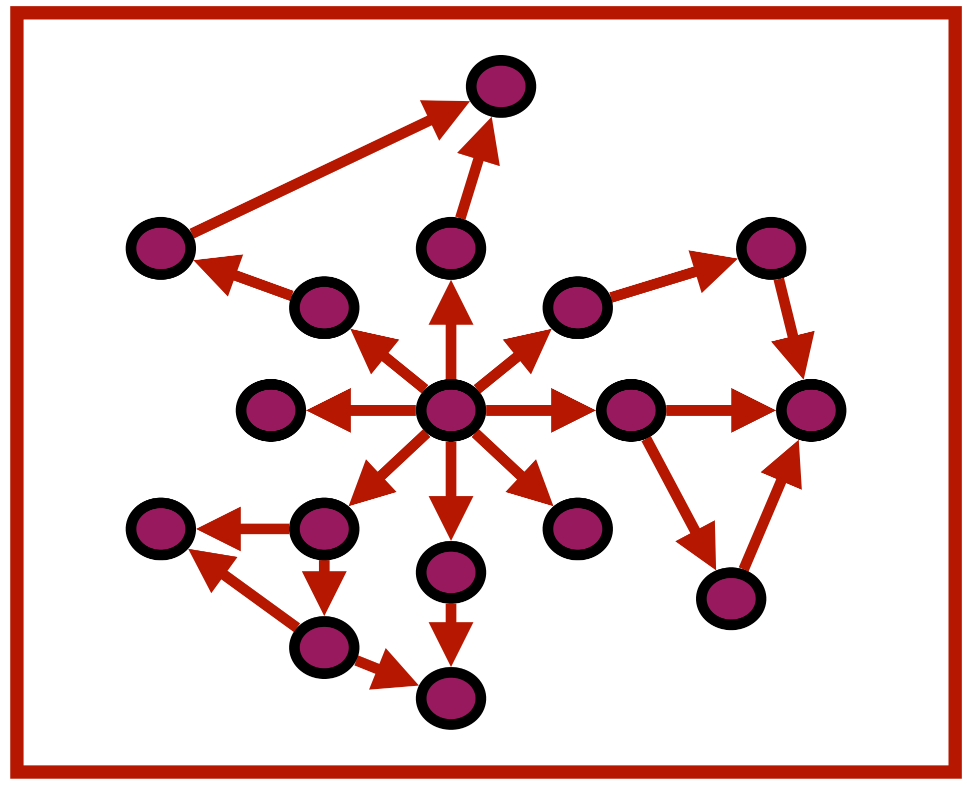
PageRank Example

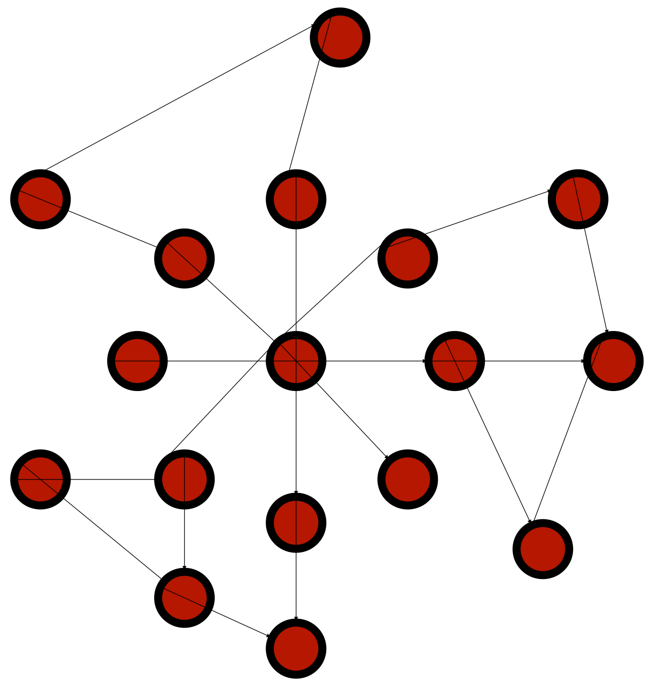




```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end


func updateVertex (v: Vertex)
    new_rank[v] = base_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end

 func main()
     for i in 1:max_iter
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
    end
 end
```

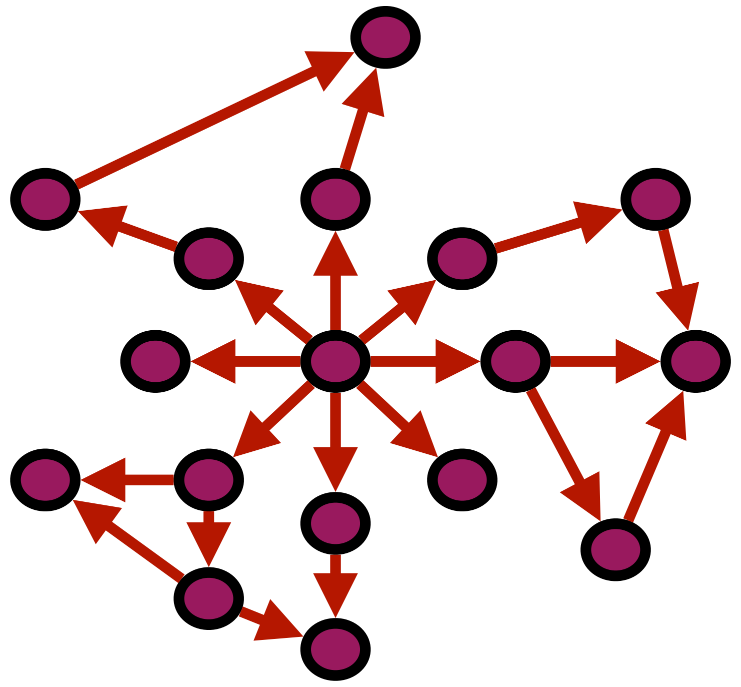
PageRank Example

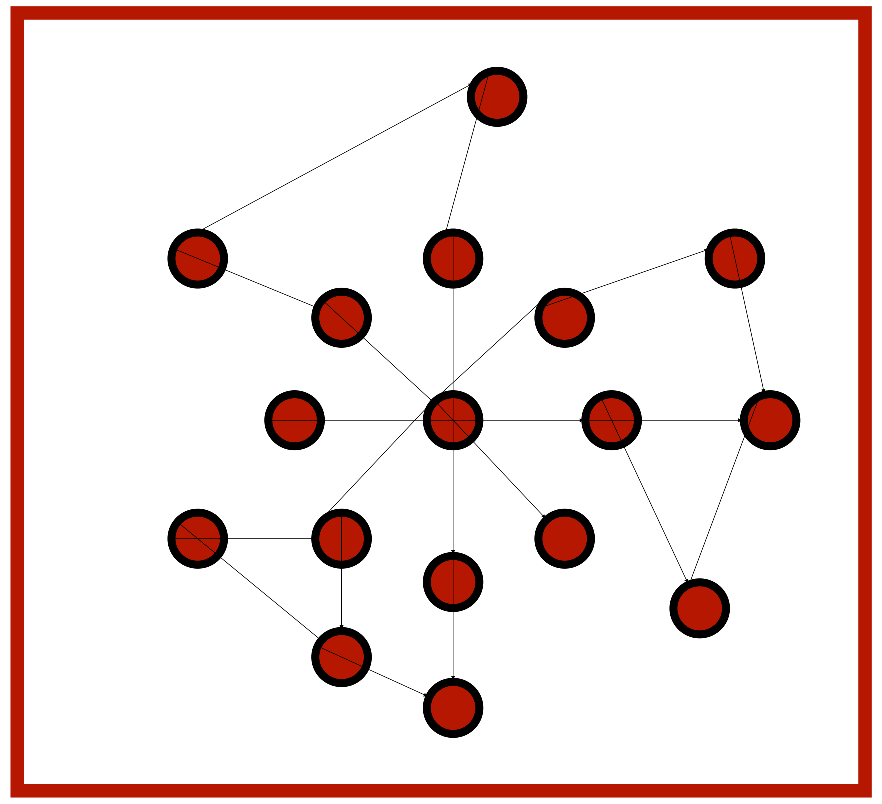




```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end


func updateVertex (v: Vertex)
    new_rank[v] = base_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end

 func main()
     for i in 1:max_iter
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
    end
 end
```

PageRank Example



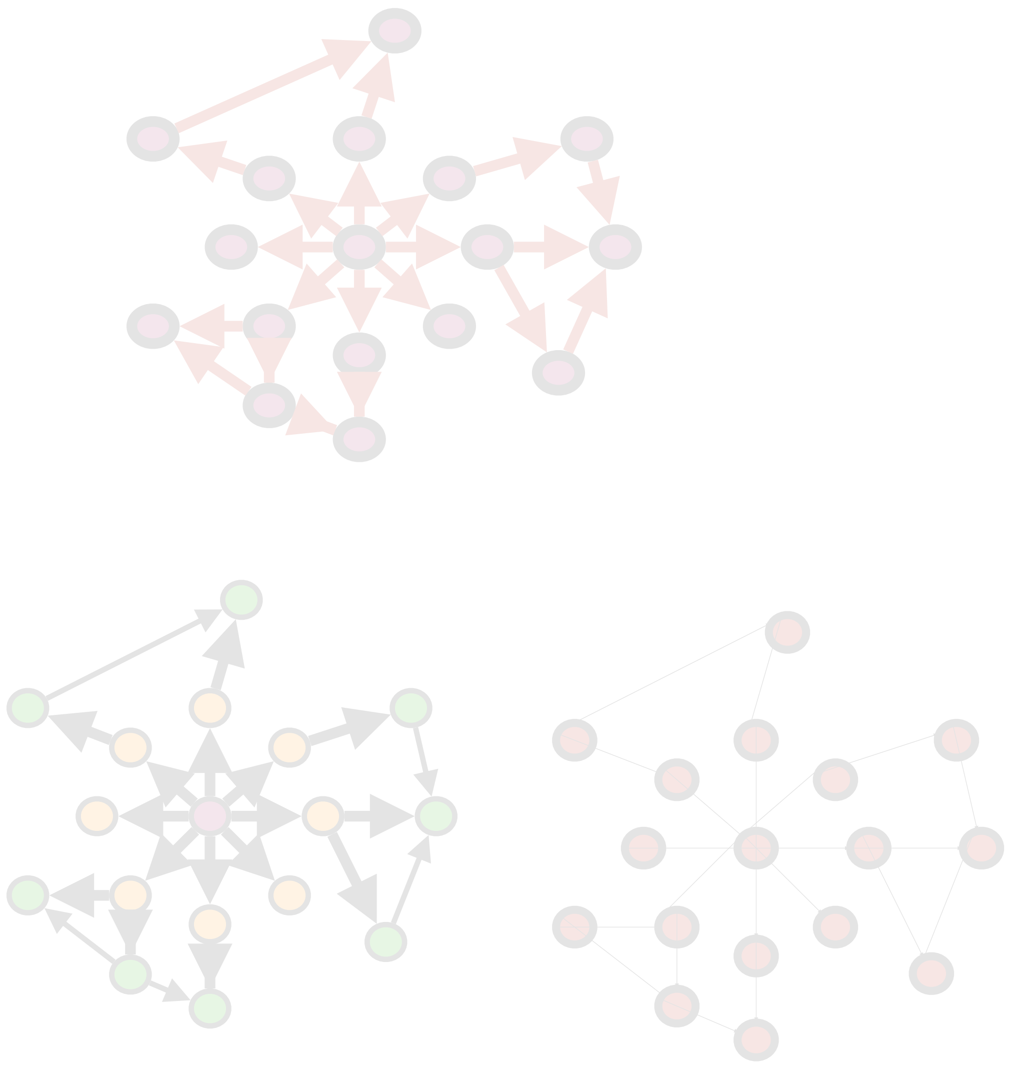


```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end
```

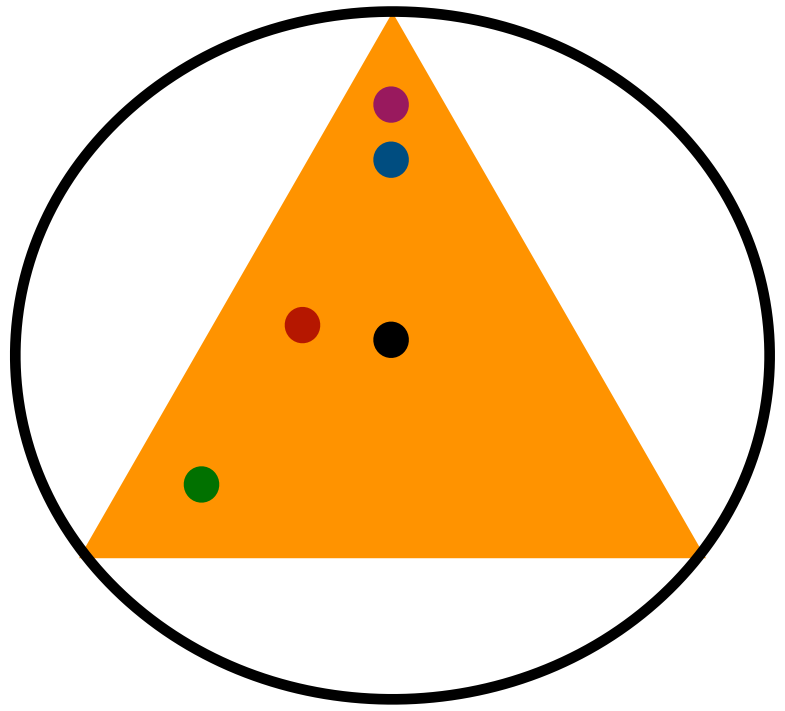
```
func updateVertex (v: Vertex)
    new_rank[v] = base_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end
```

```
func main()
    for i in 1:max_iter
      #s1# edges.apply(updateEdge);
      vertices.apply(updateVertex);
    end
end
```

GraphIt DSL

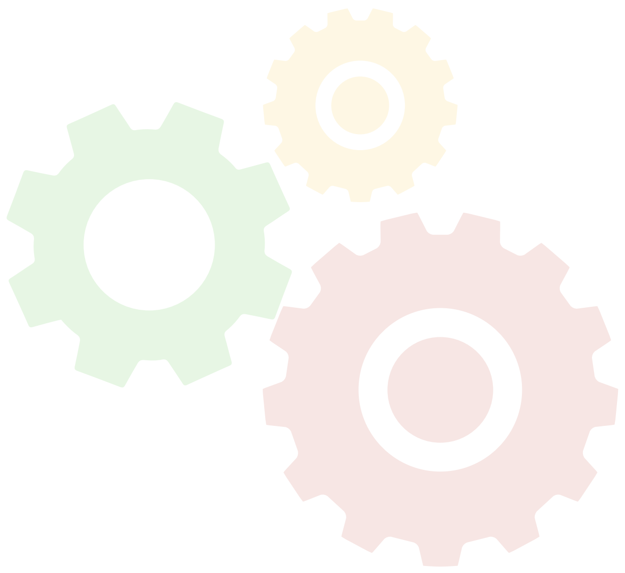


Algorithm Representation
(Algorithm Language)



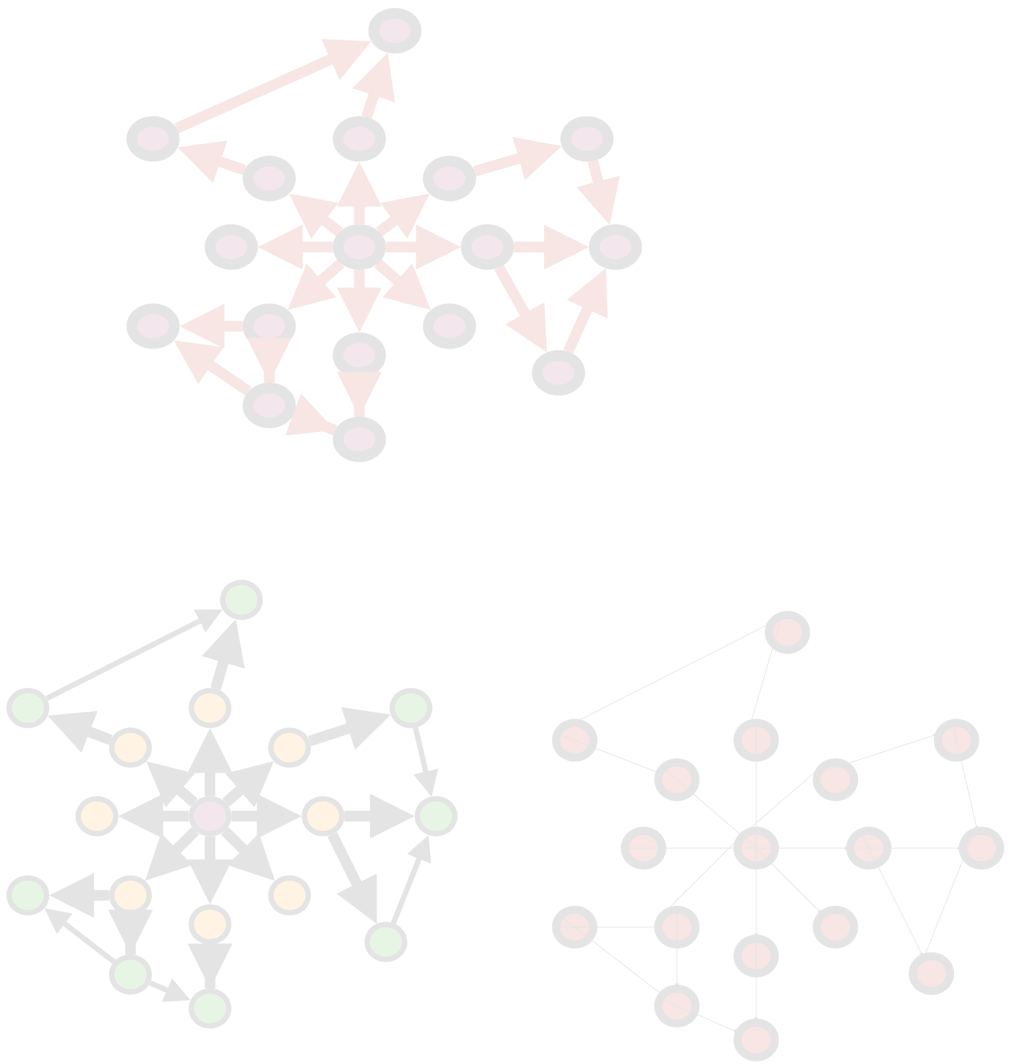
Optimization Representation

- **Scheduling Language**
- **Schedule Representation (e.g. Graph Iteration Space)**

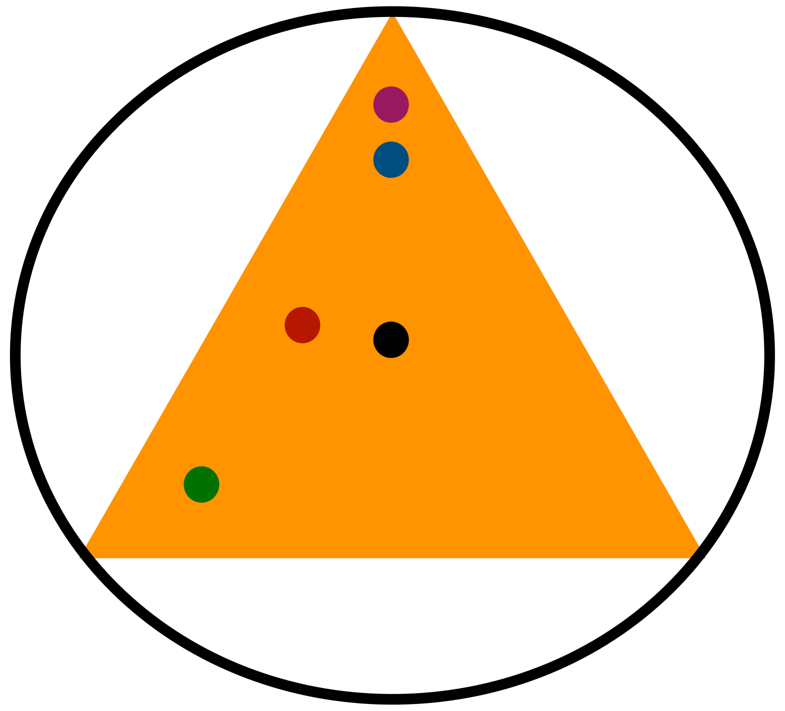


Autotuner

GraphIt DSL

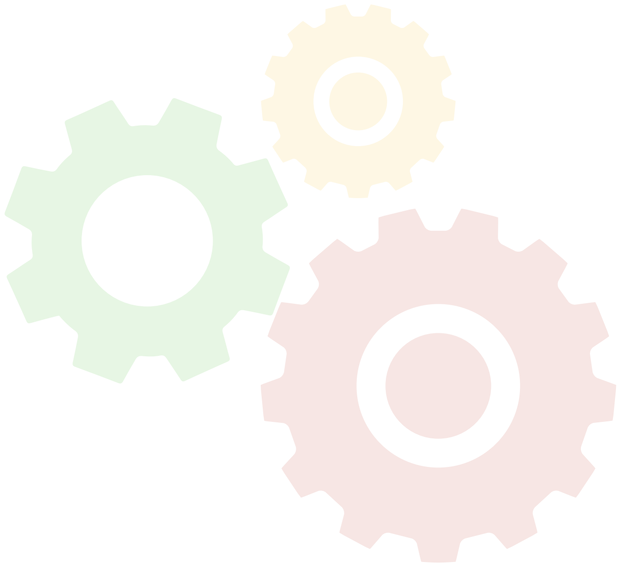


Algorithm Representation
(Algorithm Language)



Optimization Representation

- **Scheduling Language**
- **Schedule Representation (e.g. Graph Iteration Space)**



Autotuner

Scheduling Language

```
func updateEdge (src: Vertex, dst: Vertex)
    new_rank[dst] += old_rank[src] / out_degree[src]
end


func updateVertex (v: Vertex)
    new_rank[v] = base_score + 0.85*new_rank[v];
    old_rank[v] = new_rank[v];
    new_rank[v] = 0;
end

 func main()
     for i in 1:max_iter
        #s1# edges.apply(updateEdge);
        vertices.apply(updateVertex);
    end
 end
```

Scheduling Language

```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
   new_rank[v] = base_score + 0.85*new_rank[v];
   old_rank[v] = new_rank[v];
   new_rank[v] = 0;
end

 func main()
    for i in 1:max_iter
      #s1# edges.apply(updateEdge);
      vertices.apply(updateVertex);
   end
 end
```

Scheduling Language

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
   new_rank[v] = base_score + 0.85*new_rank[v];
   old_rank[v] = new_rank[v];
   new_rank[v] = 0;
end

 func main()
    for i in 1:max_iter
       #s1# edges.apply(updateEdge);
       vertices.apply(updateVertex);
    end
 end
```

Schedule 1

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
   new_rank[v] = base_score + 0.85*new_rank[v];
   old_rank[v] = new_rank[v];
   new_rank[v] = 0;
end

 func main()
    for i in 1:max_iter
       #s1# edges.apply(updateEdge);
       vertices.apply(updateVertex);
    end
 end
```

Scheduling Functions

```
schedule:
    program->configApplyDirection(“s1”, “SparsePush”);
```

Schedule 1

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
   new_rank[v] = base_score + 0.85*new_rank[v];
   old_rank[v] = new_rank[v];
   new_rank[v] = 0;
end

 func main()
    for i in 1:max_iter
      #s1# edges.apply(updateEdge);
      vertices.apply(updateVertex);
    end
 end
```

Scheduling Functions

```
schedule:
    program->configApplyDirection(“s1”, “SparsePush”);
```

Schedule 1

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
   new_rank[v] = base_score + 0.85*new_rank[v];
   old_rank[v] = new_rank[v];
   new_rank[v] = 0;
end

func main()
   for i in 1:max_iter
      #s1# edges.apply(updateEdge);
      vertices.apply(updateVertex);
   end
end
```

Pseudo Generated Code

```
double * new_rank = new double[num_verts];
double * old_rank = new double[num_verts];
int * out_degree = new int[num_verts];

…

for (NodeID src : vertices) {
  for(NodeID dst : G.getOutNgh(src)){
     new_rank[dst]  +=  old_rank[src] / out_degree[src];
   }
}

….
```

Scheduling Functions

```
schedule:
    program->configApplyDirection(“s1”, “SparsePush”);
```

Schedule 2

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
   new_rank[v] = base_score + 0.85*new_rank[v];
   old_rank[v] = new_rank[v];
   new_rank[v] = 0;
end

 func main()
    for i in 1:max_iter
      #s1# edges.apply(updateEdge);
      vertices.apply(updateVertex);
    end
 end
```

Pseudo Generated Code

```
double * new_rank = new double[num_verts];
double * old_rank = new double[num_verts];
int * out_degree = new int[num_verts];

…

parallel_for (NodeID src : vertices) {
  for(NodeID dst : G.getOutNgh(src)){
      atomic_add (&new_rank[dst],
                          old_rank[src] / out_degree[src] );
   }
}

….
```

Scheduling Functions

```
schedule:
    program->configApplyDirection("s1", "SparsePush");
    program->configApplyParallelization("s1", "dynamic-vertex-parallel");
```

Schedule 3

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
   new_rank[v] = base_score + 0.85*new_rank[v];
   old_rank[v] = new_rank[v];
   new_rank[v] = 0;
end

 func main()
    for i in 1:max_iter
       #s1# edges.apply(updateEdge);
       vertices.apply(updateVertex);
    end
 end
```

Pseudo Generated Code

```
double * new_rank = new double[num_verts];
double * old_rank = new double[num_verts];
int * out_degree = new int[num_verts];

…

parallel_for (NodeID dst : vertices) {
  for(NodeID src : G.getInNgh(dst)){
     new_rank[dst]  += old_rank[src] / out_degree[src];
   }
}

….
```

Scheduling Functions

```
schedule:
    program->configApplyDirection(“s1”, “DensePull”);
    program->configApplyParallelization(“s1”, “dynamic-vertex-parallel”);
```

Schedule 4

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
   new_rank[v] = base_score + 0.85*new_rank[v];
   old_rank[v] = new_rank[v];
   new_rank[v] = 0;
end

 func main()
    for i in 1:max_iter
       #s1# edges.apply(updateEdge);
       vertices.apply(updateVertex);
    end
 end
```

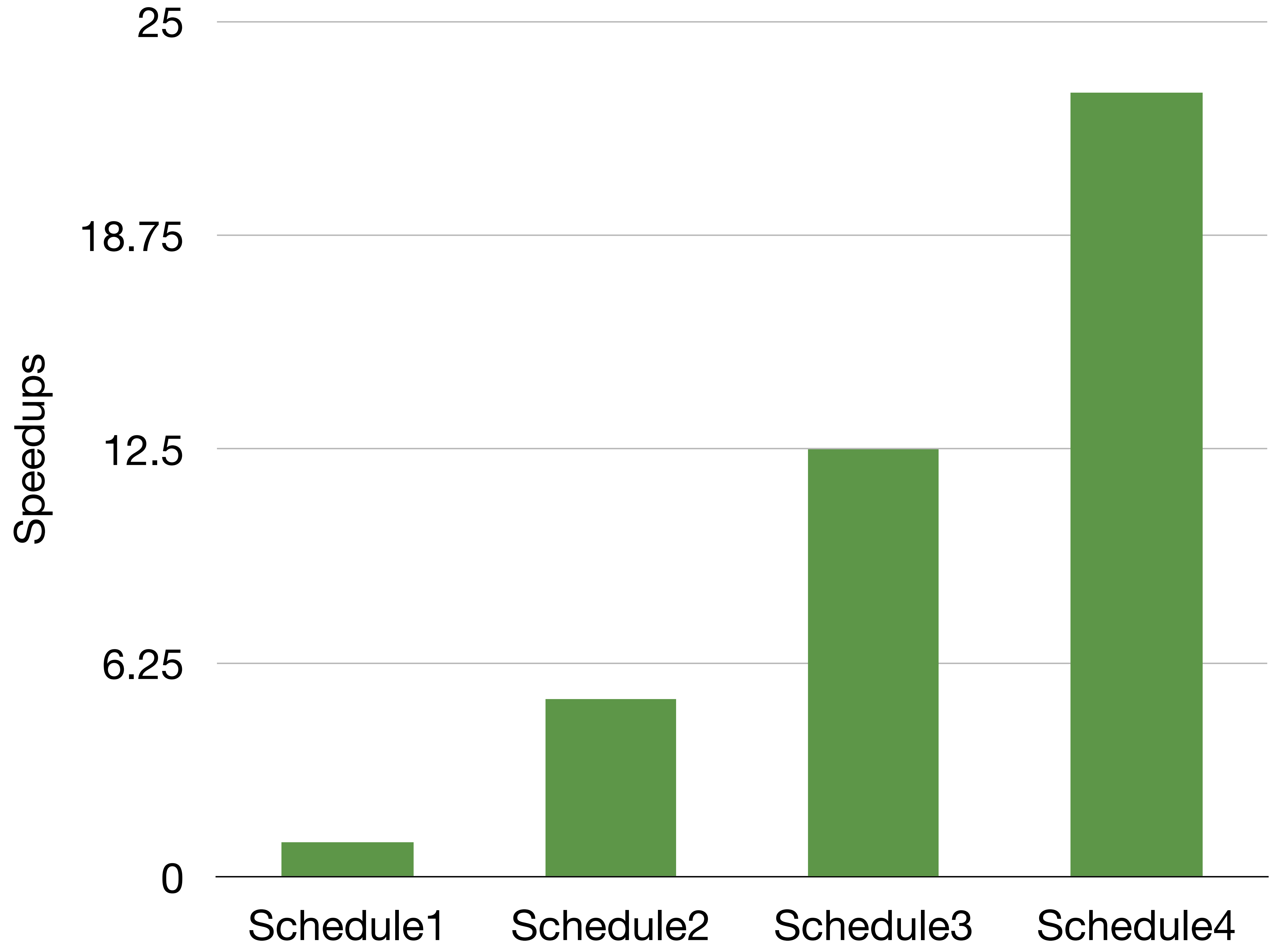
Pseudo Generated Code

```
double * new_rank = new double[num_verts];
double * old_rank = new double[num_verts];
int * out_degree = new int[num_verts];

…
for (Subgraph sg : G.subgraphs) {
  parallel_for (NodeID dst : verticesa) {
    for(NodeID src : G.getInNgh(dst)){
     new_rank[dst]  += old_rank[src] / out_degree[src];
    }
  }
}
….
```

Scheduling Functions

```
schedule:
    program->configApplyDirection(“s1”, “DensePull”);
    program->configApplyParallelization(“s1”, “dynamic-vertex-parallel”);
    program->configApplyNumSSG(“s1”, “fixed-vertex-count”, 10);
```

Speedups of Schedules

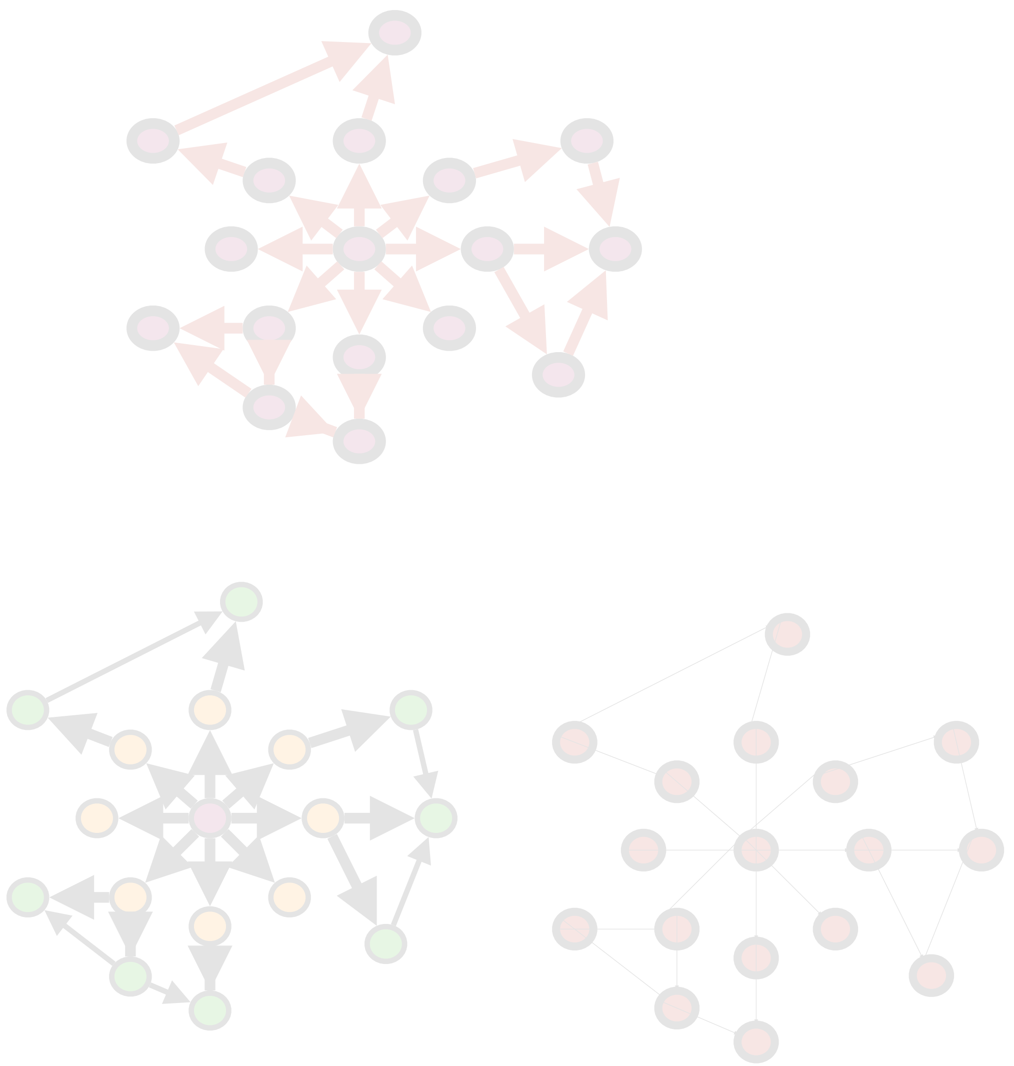


Performance of PageRank on Twitter Graph on Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores and 48 hyper-threads.

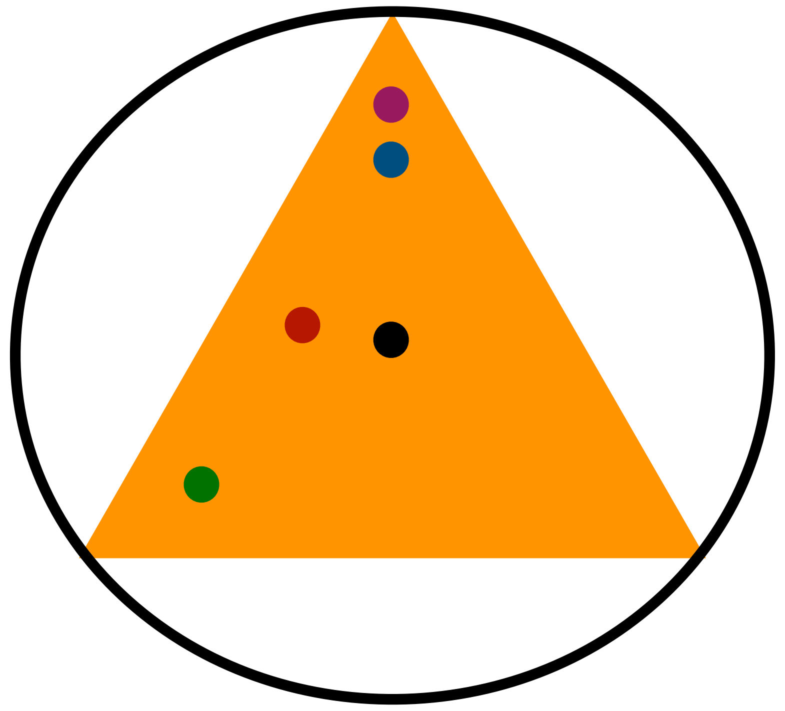
Many More Optimizations

- Direction optimizations (configApplyDirection),
  - SparsePush, DensePush, DensePull, DensePull-SparsePush, DensePush-SparsePush
- Parallelization strategies (configApplyParallelization)
  - serial, dynamic-vertex-parallel, static-vertexparallel, edge-aware-dynamic-vertex-parallel, edge-parallel
- Cache (configApplyNumSSG)
  - fixed-vertex-count, edge-aware-vertexcount
- NUMA (configApplyNUMA)
  - serial, static-parallel, dynamic-parallel
- AoS, SoA (fuseFields)
- Vertexset data layout (configApplyDenseVertexSet)
  - bitvector, boolean array

GraphIt DSL

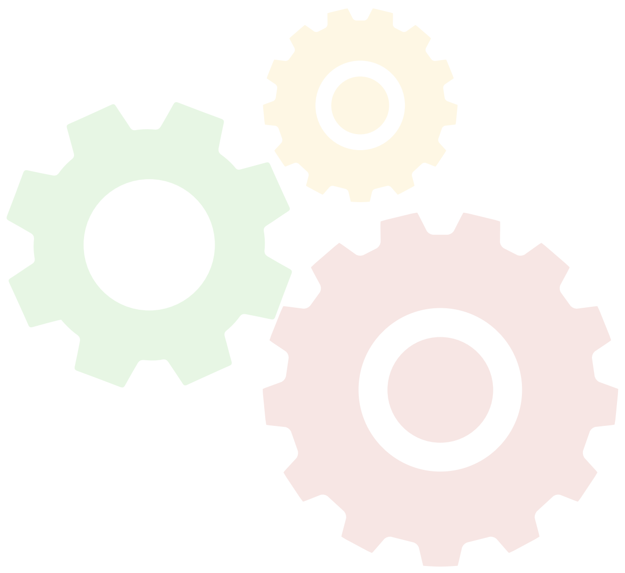


Algorithm Representation
(Algorithm Language)



Optimization Representation

- **Scheduling Language**
- **Schedule Representation (e.g. Graph Iteration Space)**



Autotuner

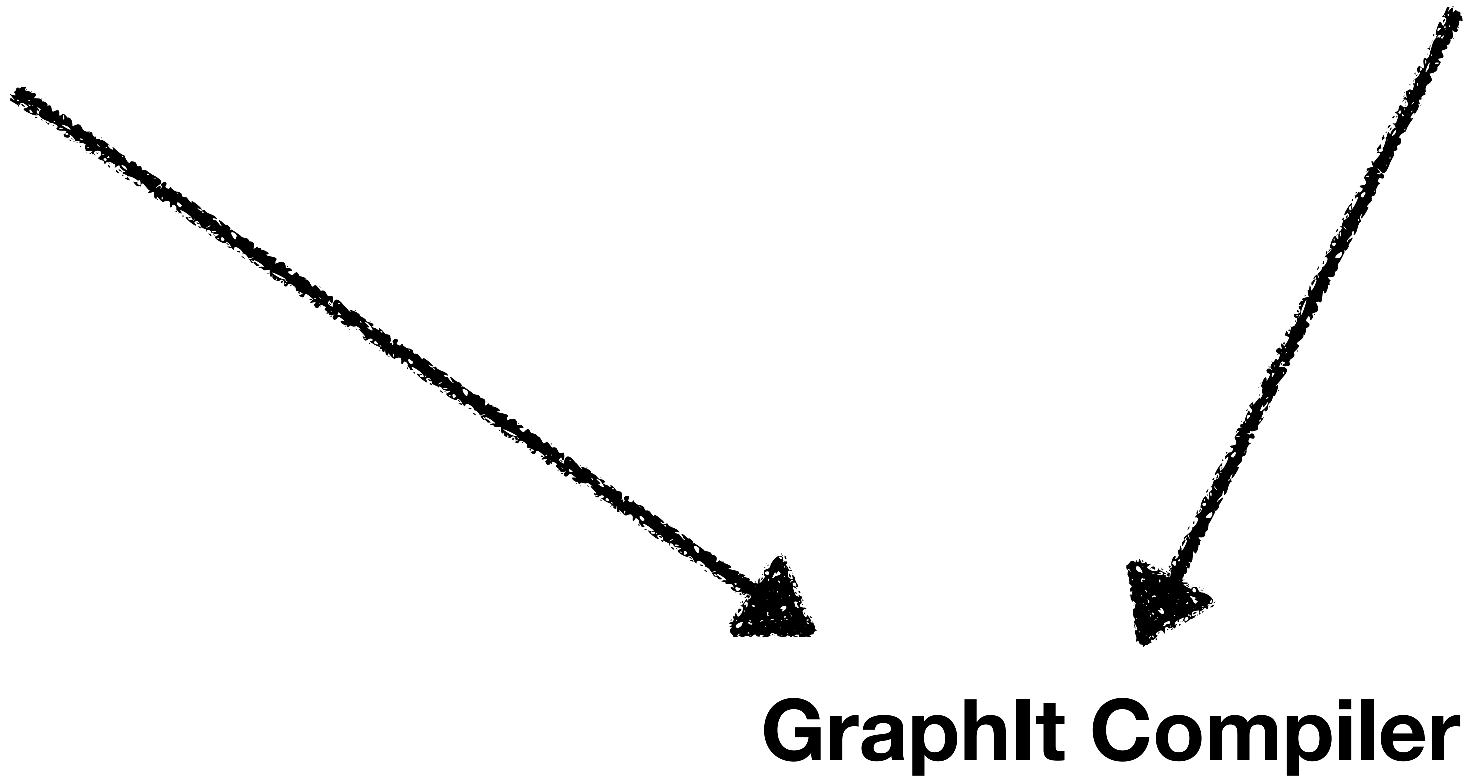
Schedule Representation

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end


func updateVertex (v: Vertex)
   new_rank[v] = base_score + 0.85*new_rank[v];
   old_rank[v] = new_rank[v];
   new_rank[v] = 0;
end

 func main()
    for i in 1:11
       #s1# edges.apply(updateEdge);
       vertices.apply(updateVertex);
    end
 end
```

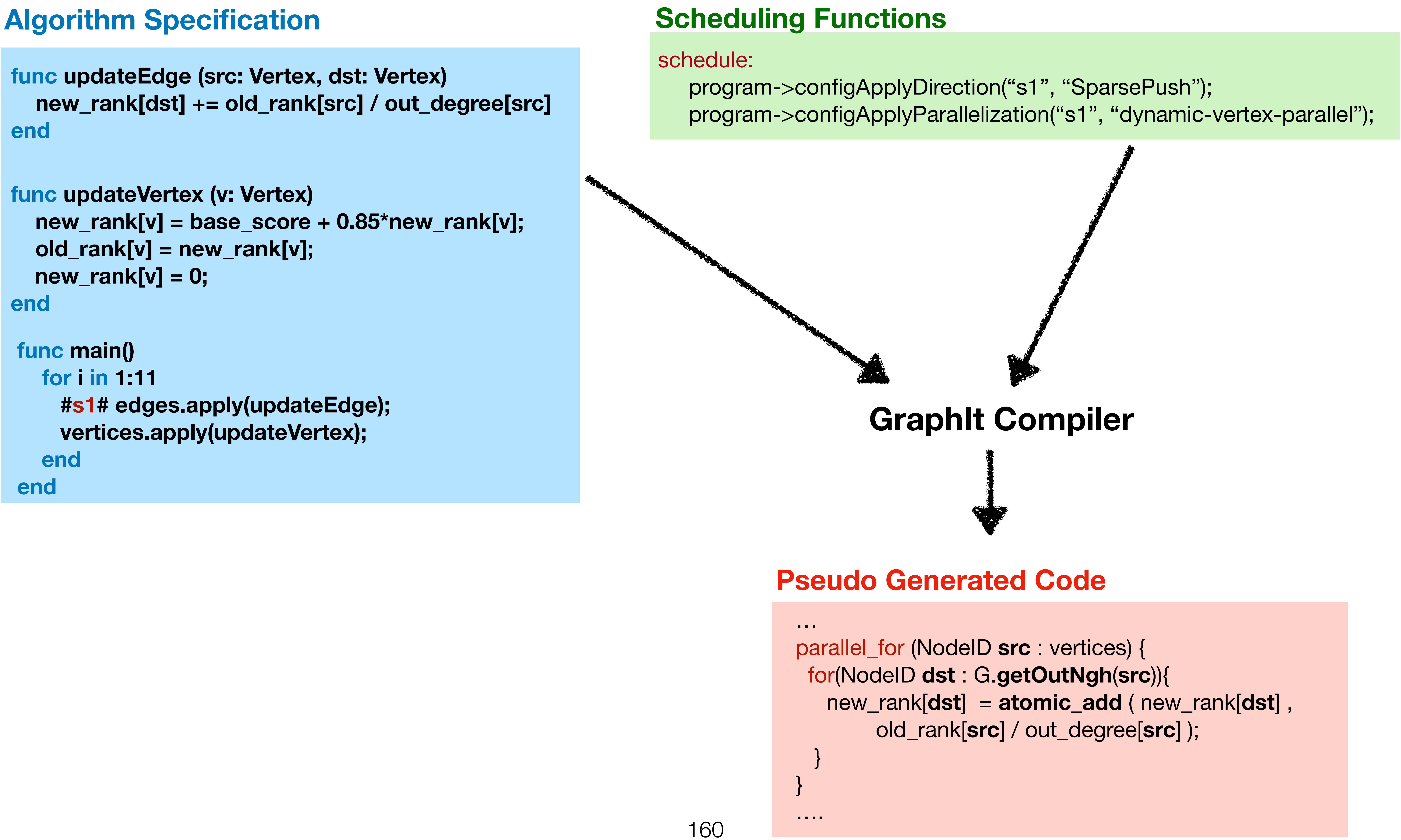
Scheduling Functions

```
schedule:
    program->configApplyDirection(“s1”, “SparsePush”);
    program->configApplyParallelization(“s1”, “dynamic-vertex-parallel”);
```



GraphIt Compiler

Schedule Representation



Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end


func updateVertex (v: Vertex)
   new_rank[v] = base_score + 0.85*new_rank[v];
   old_rank[v] = new_rank[v];
   new_rank[v] = 0;
end

 func main()
    for i in 1:11
       #s1# edges.apply(updateEdge);
       vertices.apply(updateVertex);
    end
 end
```

Scheduling Functions

```
schedule:
    program->configApplyDirection(“s1”, “SparsePush”);
    program->configApplyParallelization(“s1”, “dynamic-vertex-parallel”);
```

GraphIt Compiler

Pseudo Generated Code

```
 …
parallel_for (NodeID src : vertices) {
  for(NodeID dst : G.getOutNgh(src)){
     new_rank[dst]  = atomic_add ( new_rank[dst] ,
             old_rank[src] / out_degree[src] );
   }
}
….
```

Schedule Representation

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
   new_rank[v] = base_score + 0.85*new_rank[v];
   old_rank[v] = new_rank[v];
   new_rank[v] = 0;
end

func main()
   for i in 1:11
      #s1# edges.apply(updateEdge);
      vertices.apply(updateVertex);
   end
end
```

Scheduling Functions

```
schedule:
    program->configApplyDirection(“s1”, “SparsePush”);
    program->configApplyParallelization(“s1”, “dynamic-vertex-parallel”);
```

Internal Schedule Representation

Pseudo Generated Code

```
...
parallel_for (NodeID src : vertices) {
  for(NodeID dst : G.getOutNgh(src)){
     new_rank[dst]  = atomic_add ( new_rank[dst] ,
            old_rank[src] / out_degree[src] );
   }
}
....
```

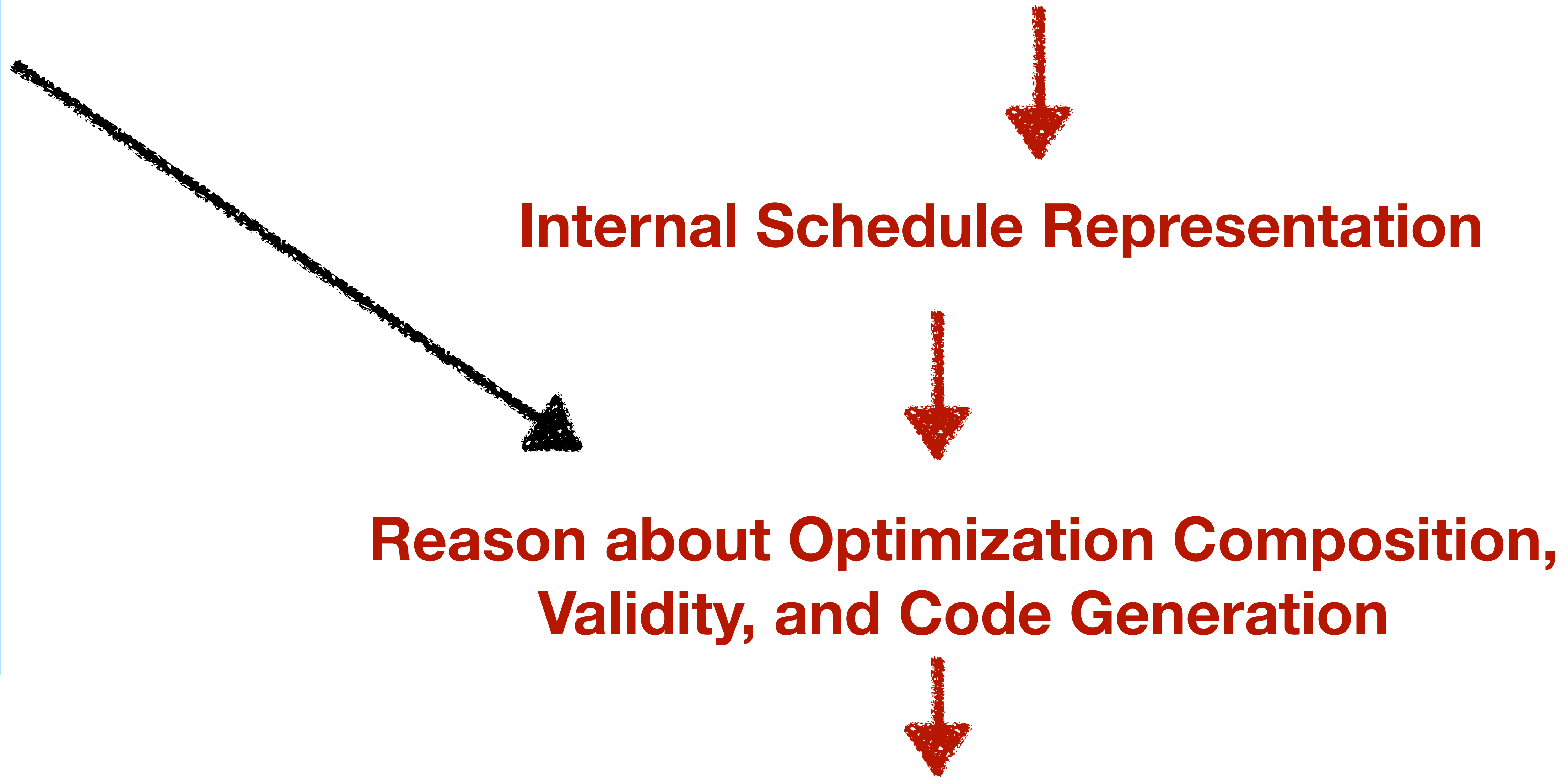
Schedule Representation

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
   new_rank[v] = base_score + 0.85*new_rank[v];
   old_rank[v] = new_rank[v];
   new_rank[v] = 0;
end

 func main()
    for i in 1:11
       #s1# edges.apply(updateEdge);
       vertices.apply(updateVertex);
    end
 end
```

Scheduling Functions

```
schedule:
    program->configApplyDirection(“s1”, “SparsePush”);
    program->configApplyParallelization(“s1”, “dynamic-vertex-parallel”);
```



Internal Schedule Representation

Reason about Optimization Composition, Validity, and Code Generation

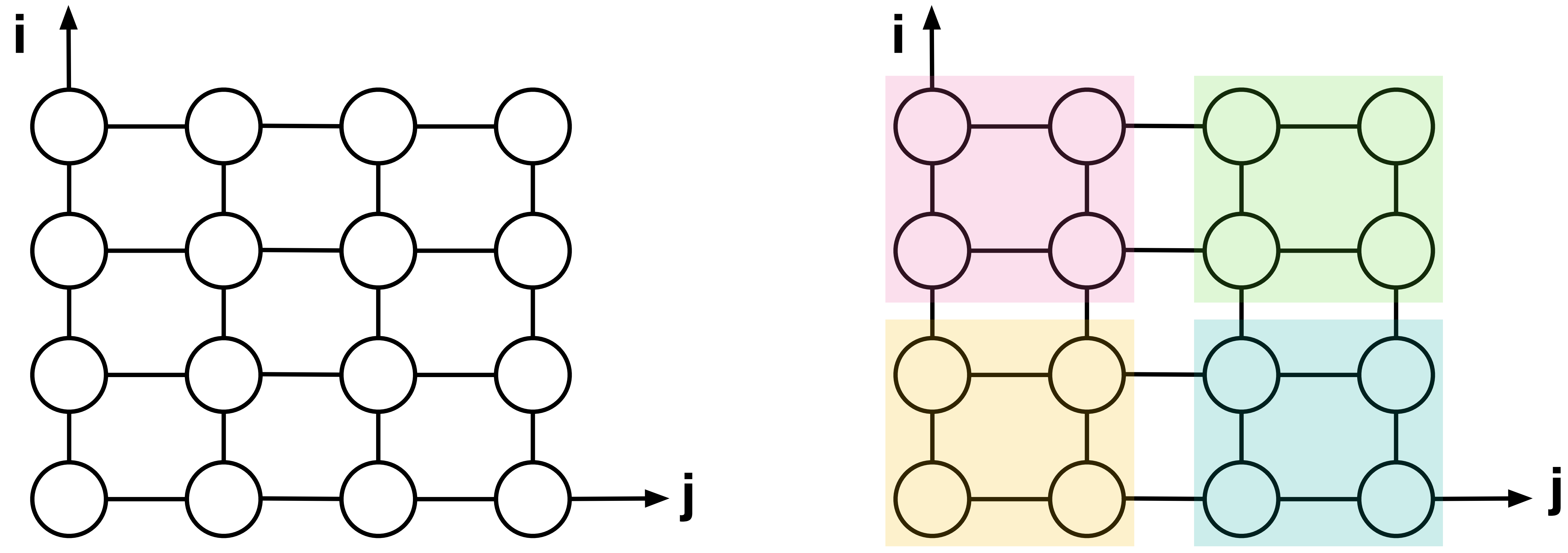
Pseudo Generated Code

```
...
parallel_for (NodeID src : vertices) {
  for(NodeID dst : G.getOutNgh(src)){
     new_rank[dst]  = atomic_add ( new_rank[dst] ,
             old_rank[src] / out_degree[src] );
   }
}
....
```

Iteration Spaces

- Very versatile representation of dense loops and arrays
- Used for:
  - Program analyses
  - Composition of complex loop transformations
  - Framework for code generation

Iteration Space Representation



Iteration Space Vectors

<i, j>

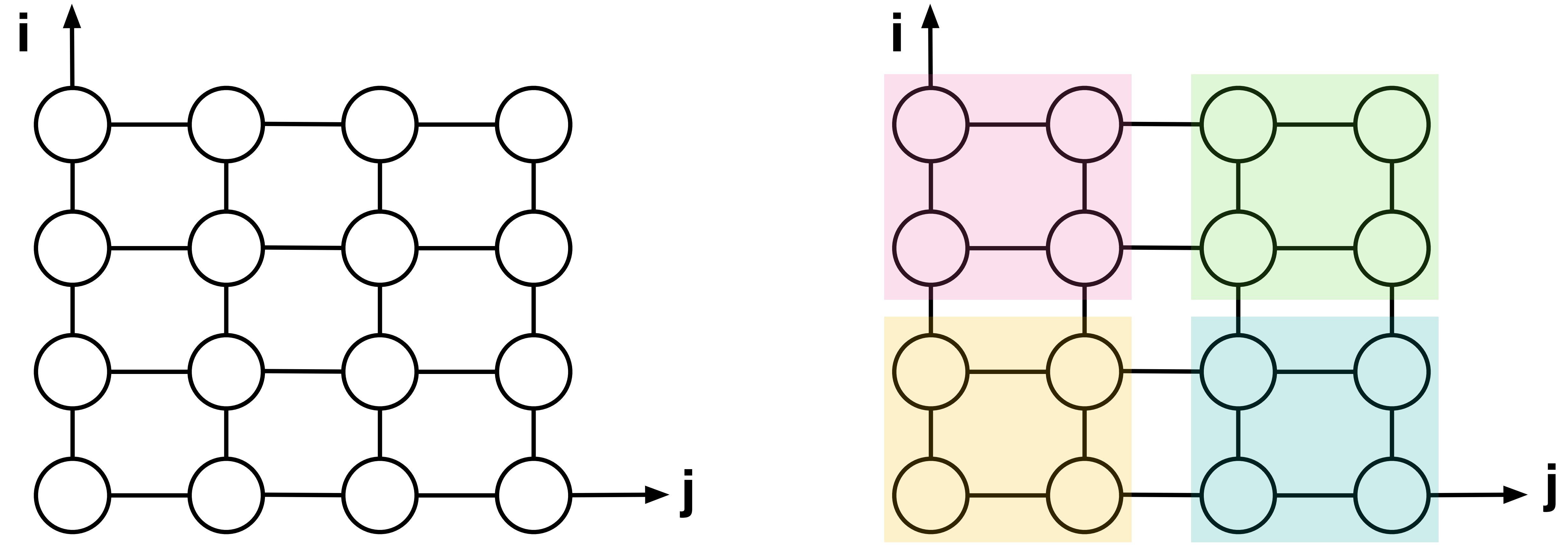
$<i_1, j_1, i_2, j_2>$

Iteration Spaces

- Very versatile representation of dense loops and arrays
- Used for:
  - Program analyses
  - Composition of complex loop transformations
  - Framework for code generation

We extend it to <u>sparse</u> loops

Iteration Space Representation



Iteration Space Vectors

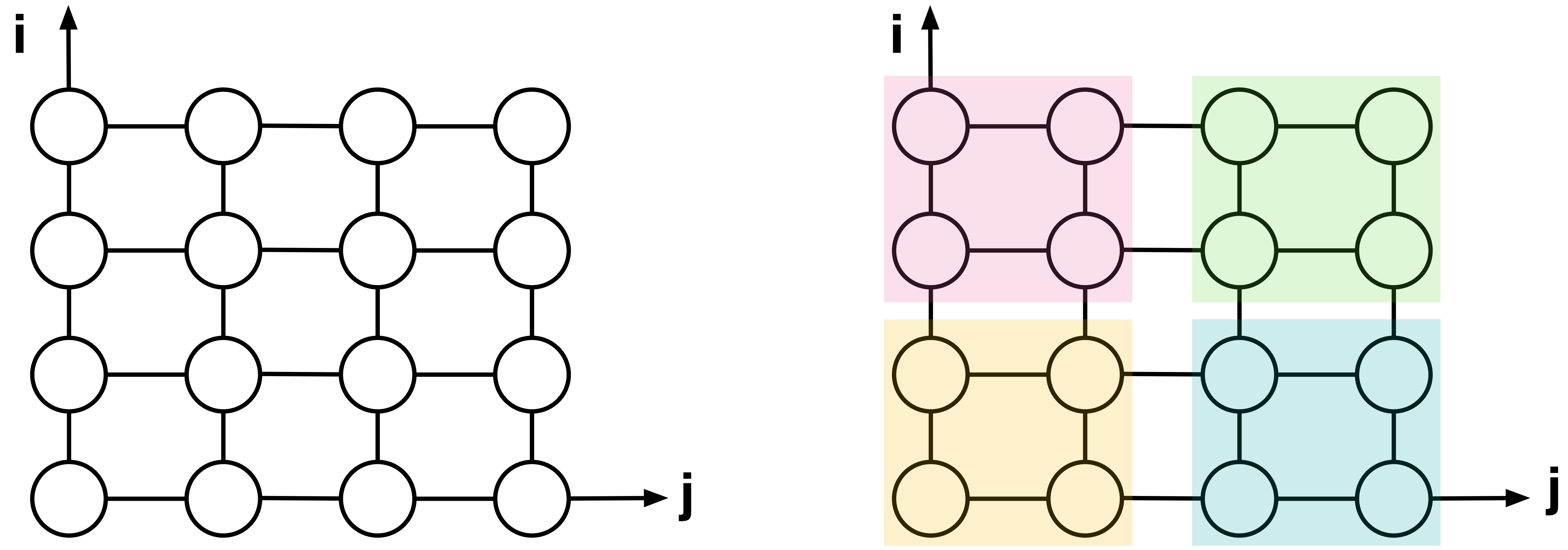
$\langle i, j\rangle$

$\langle i_1, j_1, i_2, j_2\rangle$

Graph Iteration Space

- Very versatile representation of dense loops and arrays
- Used for:
  - Program analyses
  - Composition of complex loop transformations
  - Framework for code generation

Iteration Space Representation



Iteration Space Vectors

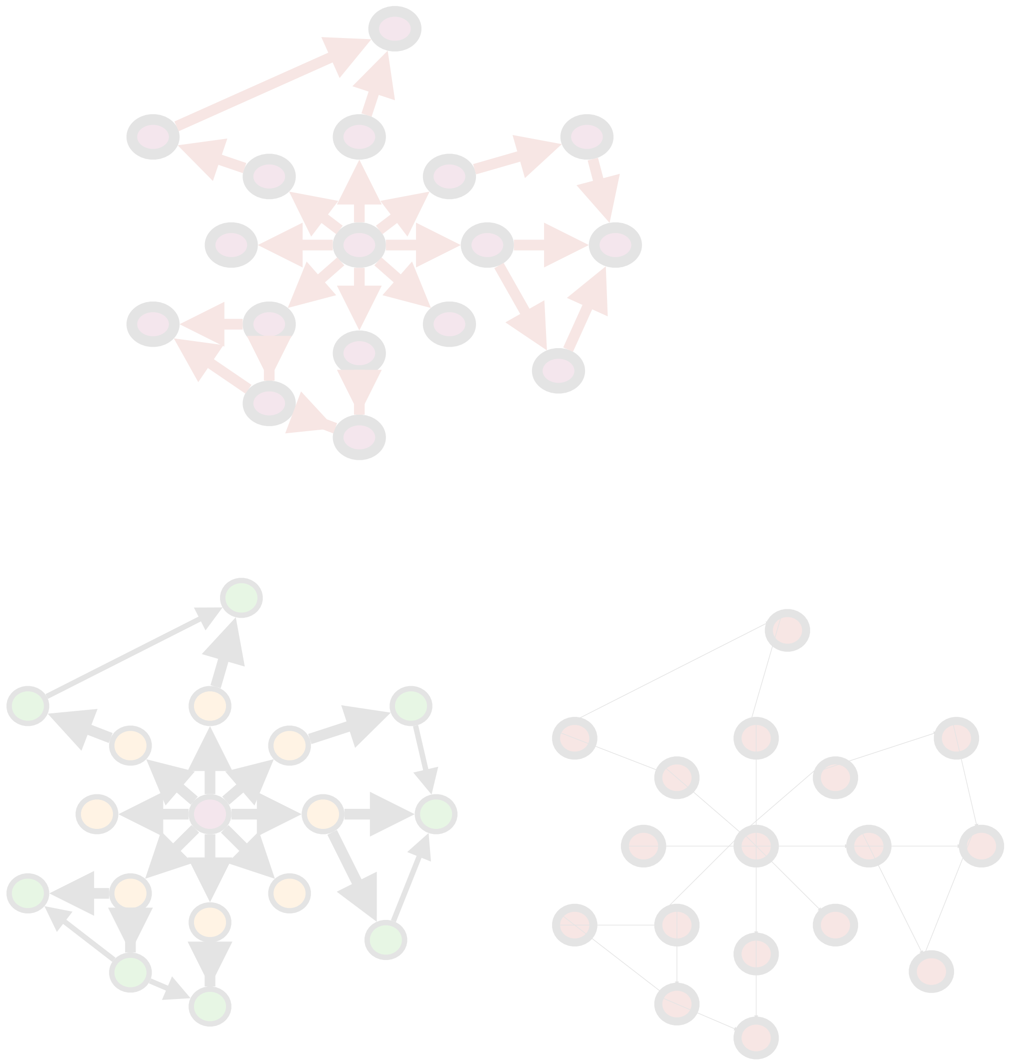
<i, j> **$<i_1, j_1, i_2, j_2>$**

We extend it to <u>sparse</u> loops

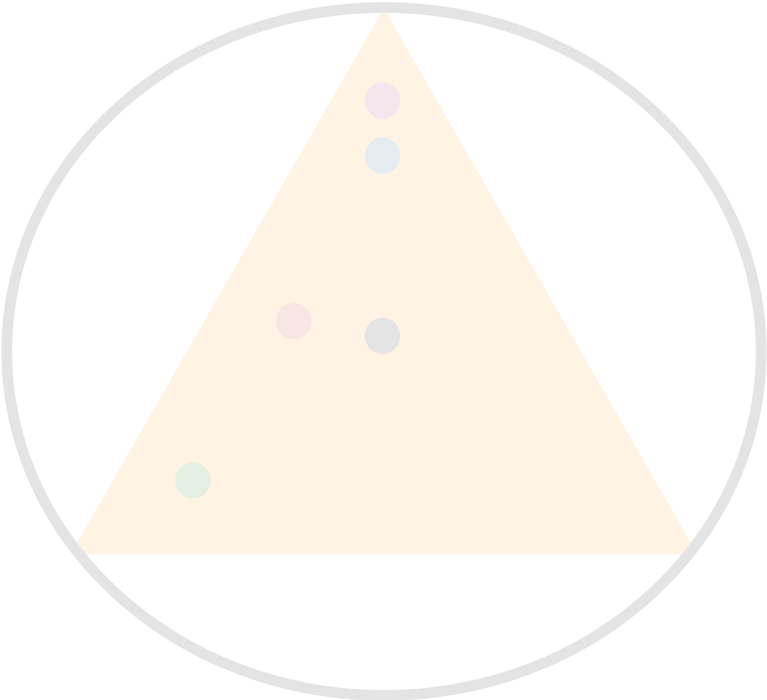
< SSG **[tags]**, BSG **[tags]**, OuterIter [dst, **tags**], InnerIter [src, **tags**] >

Augmented with Parallelization, Partitioning, Data Layout Tags

GraphIt DSL

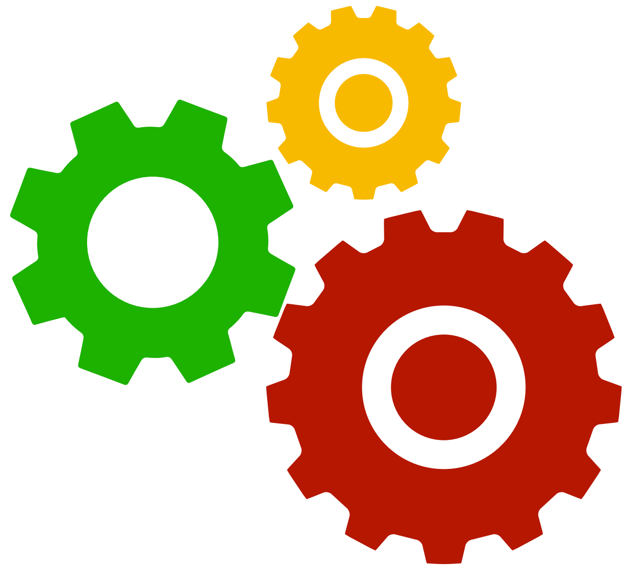


Algorithm Representation (Algorithm Language)



Optimization Representation

- Scheduling Language
- Schedule Representation (e.g. Graph Iteration Space)



Autotuner

Schedule 4

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
   new_rank[v] = base_score + 0.85*new_rank[v];
   old_rank[v] = new_rank[v];
   new_rank[v] = 0;
end

 func main()
    for i in 1:11
      #s1# edges.apply(updateEdge);
      vertices.apply(updateVertex);
    end
 end
```

Scheduling Functions

```
schedule:
    program->configApplyDirection("s1", "DensePull");
    program->configApplyParallelization("s1", "dynamic-vertex-parallel");
    program->configApplyNumSSG("s1", "fixed-vertex-count", 10);
```

Schedule 4

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
   new_rank[v] = base_score + 0.85*new_rank[v];
   old_rank[v] = new_rank[v];
   new_rank[v] = 0;
end

 func main()
    for i in 1:11
       #s1# edges.apply(updateEdge);
       vertices.apply(updateVertex);
    end
 end
```

Scheduling Functions

```
schedule:
    program->configApplyDirection(“s1”, “DensePull”);
    program->configApplyParallelization(“s1”, “dynamic-vertex-parallel”);
    program->configApplyNumSSG(“s1”, “fixed-vertex-count”, 10);
```

Finding the best schedule can be hard for non-experts.

Goal

Algorithm Specification

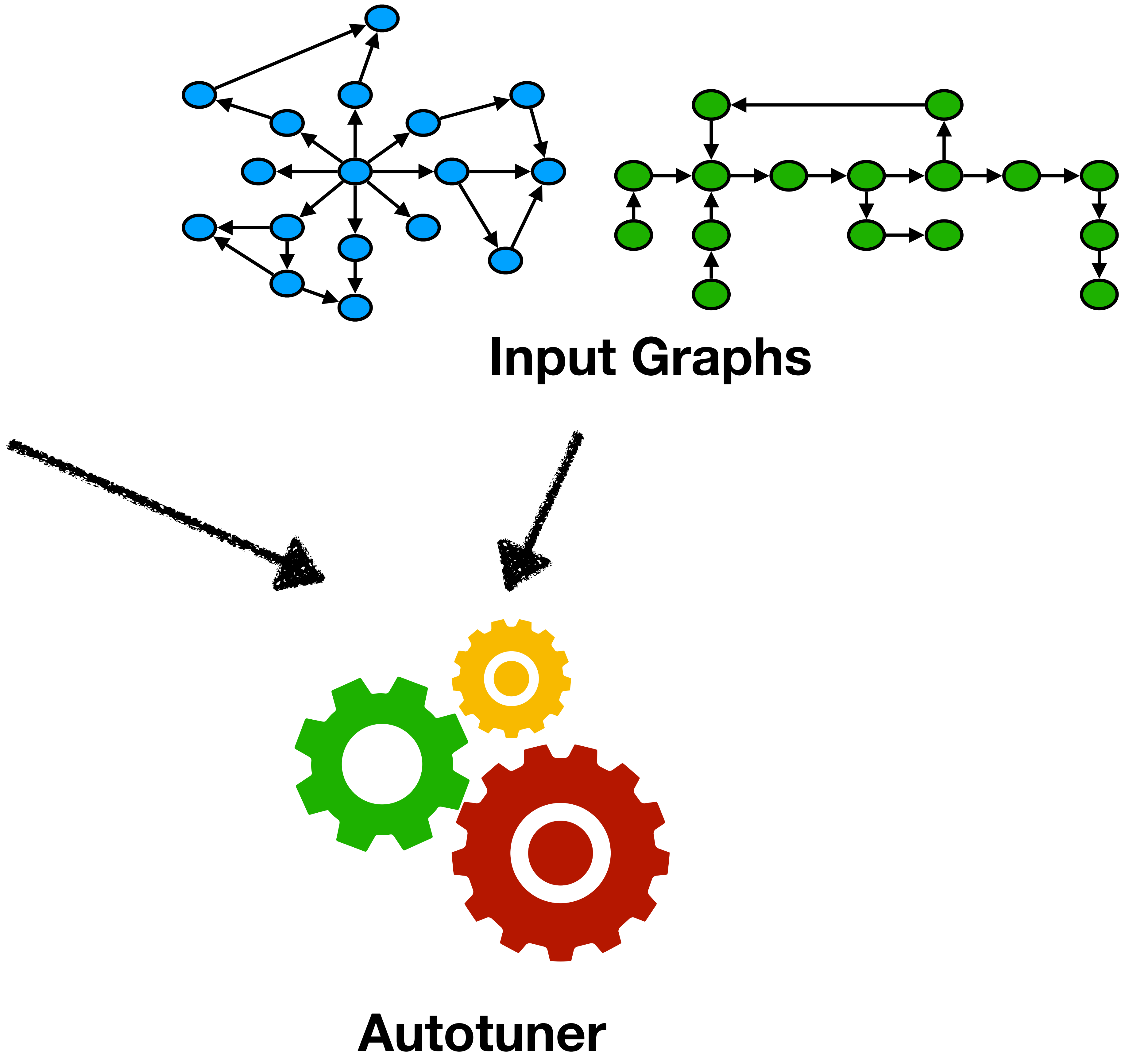
```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
   new_rank[v] = base_score + 0.85*new_rank[v];
   old_rank[v] = new_rank[v];
   new_rank[v] = 0;
end

 func main()
    for i in 1:11
      #s1# edges.apply(updateEdge);
      vertices.apply(updateVertex);
    end
 end
```

Ideally, the user only need to write the algorithm

Autotuner

Algorithm Specification

```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
   new_rank[v] = base_score + 0.85*new_rank[v];
   old_rank[v] = new_rank[v];
   new_rank[v] = 0;
end

 func main()
    for i in 1:11
       #s1# edges.apply(updateEdge);
       vertices.apply(updateVertex);
    end
 end
```



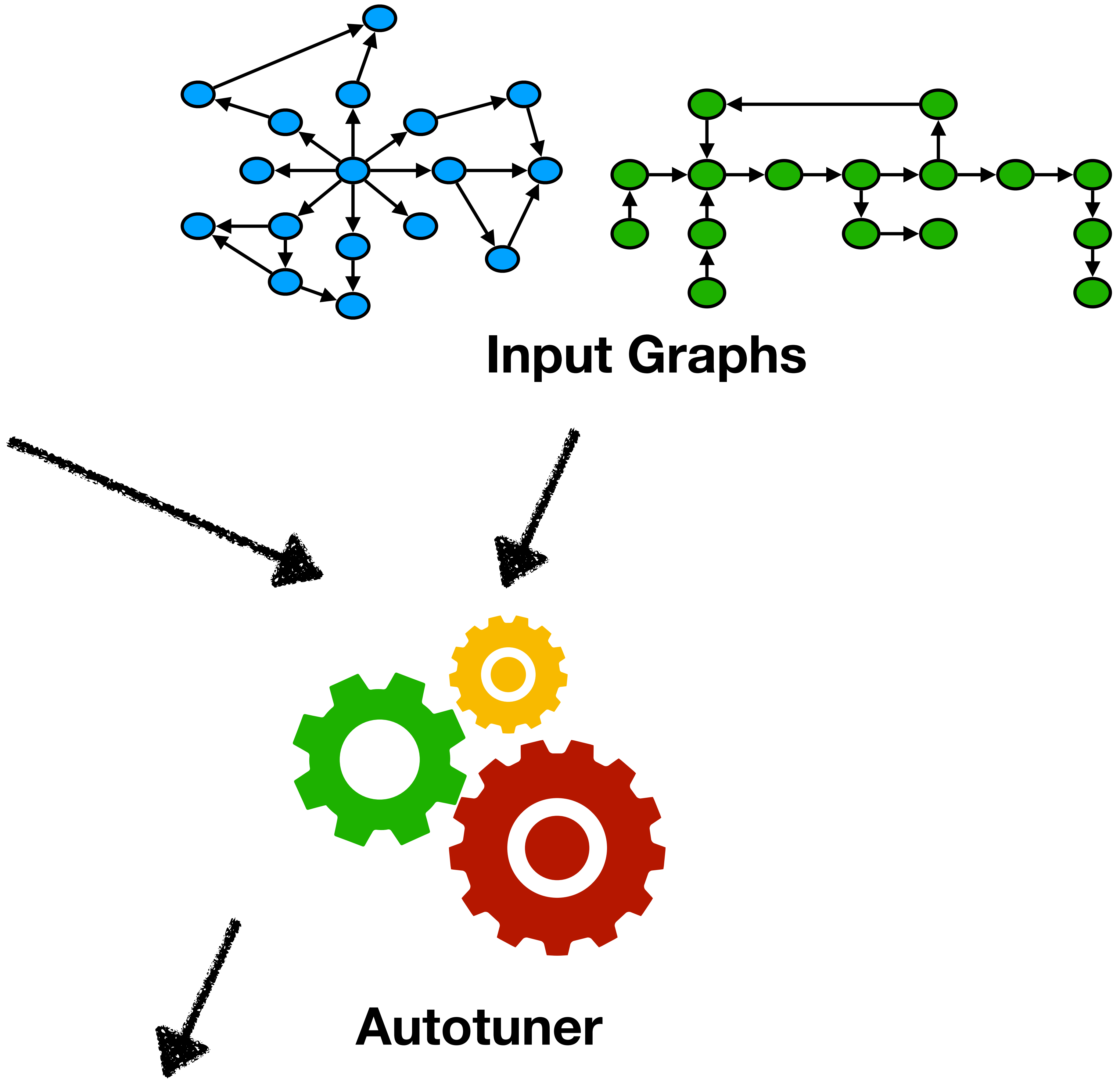
Input Graphs

Autotuner

Autotuner

Algorithm Specification

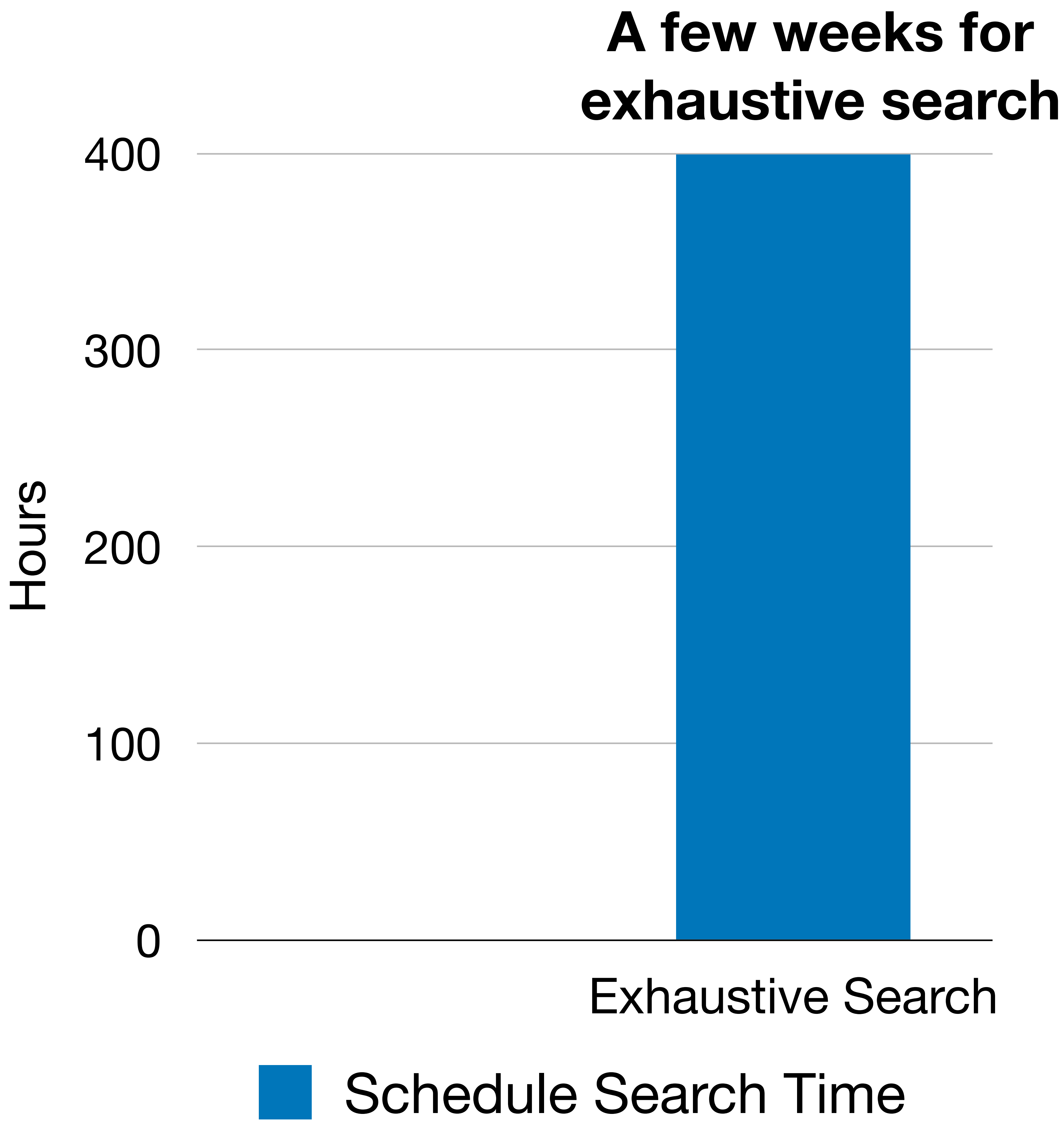
```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
   new_rank[v] = base_score + 0.85*new_rank[v];
   old_rank[v] = new_rank[v];
   new_rank[v] = 0;
end

 func main()
    for i in 1:11
       #s1# edges.apply(updateEdge);
       vertices.apply(updateVertex);
    end
 end
```



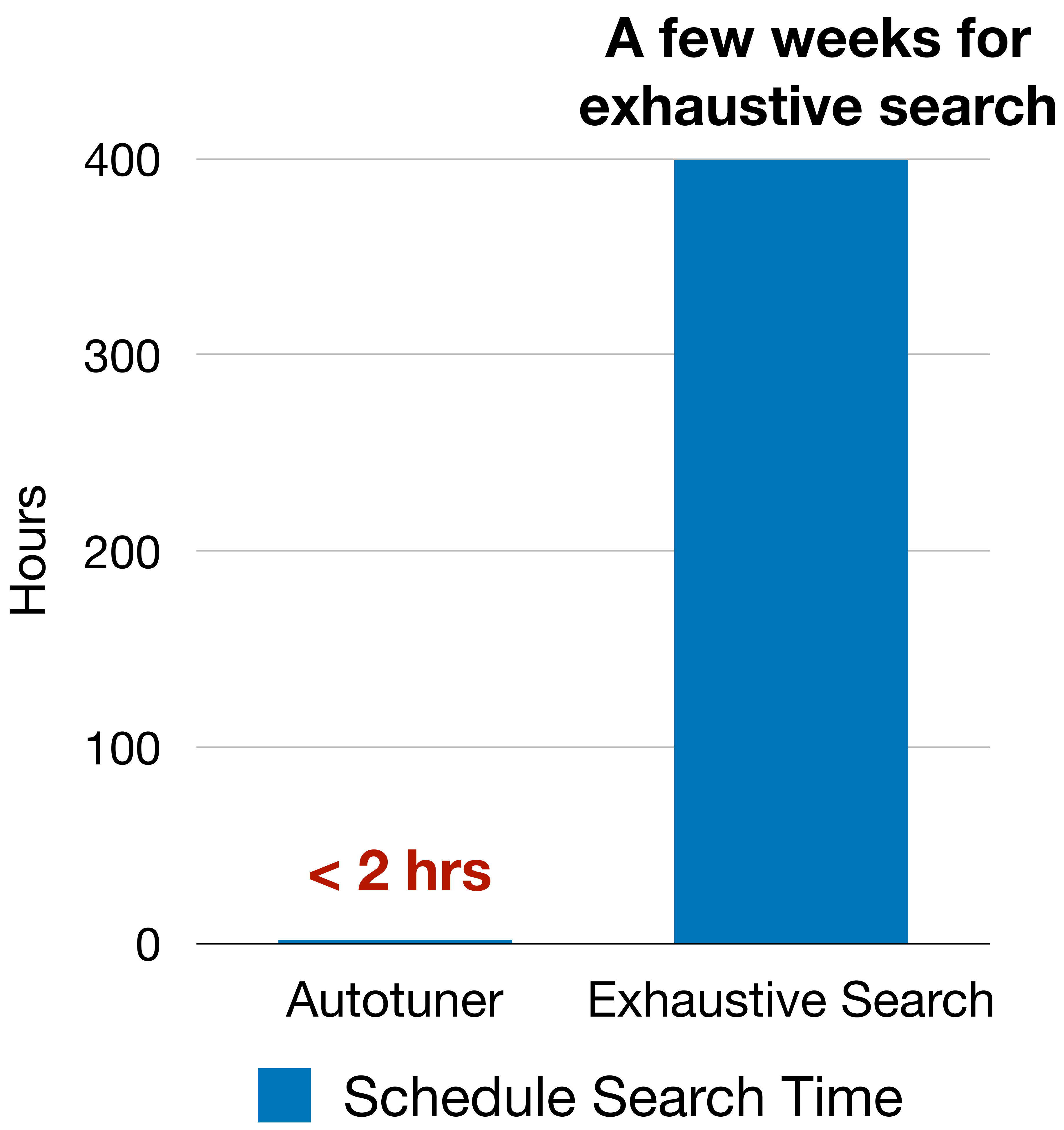
Scheduling Functions

```
schedule:
    program->configApplyDirection(“s1”, “DensePull”);
    program->configApplyParallelization(“s1”, “dynamic-vertex-parallel”);
    program->configApplyNumSSG(“s1”, “fixed-vertex-count”, 10);
```

Autotuner

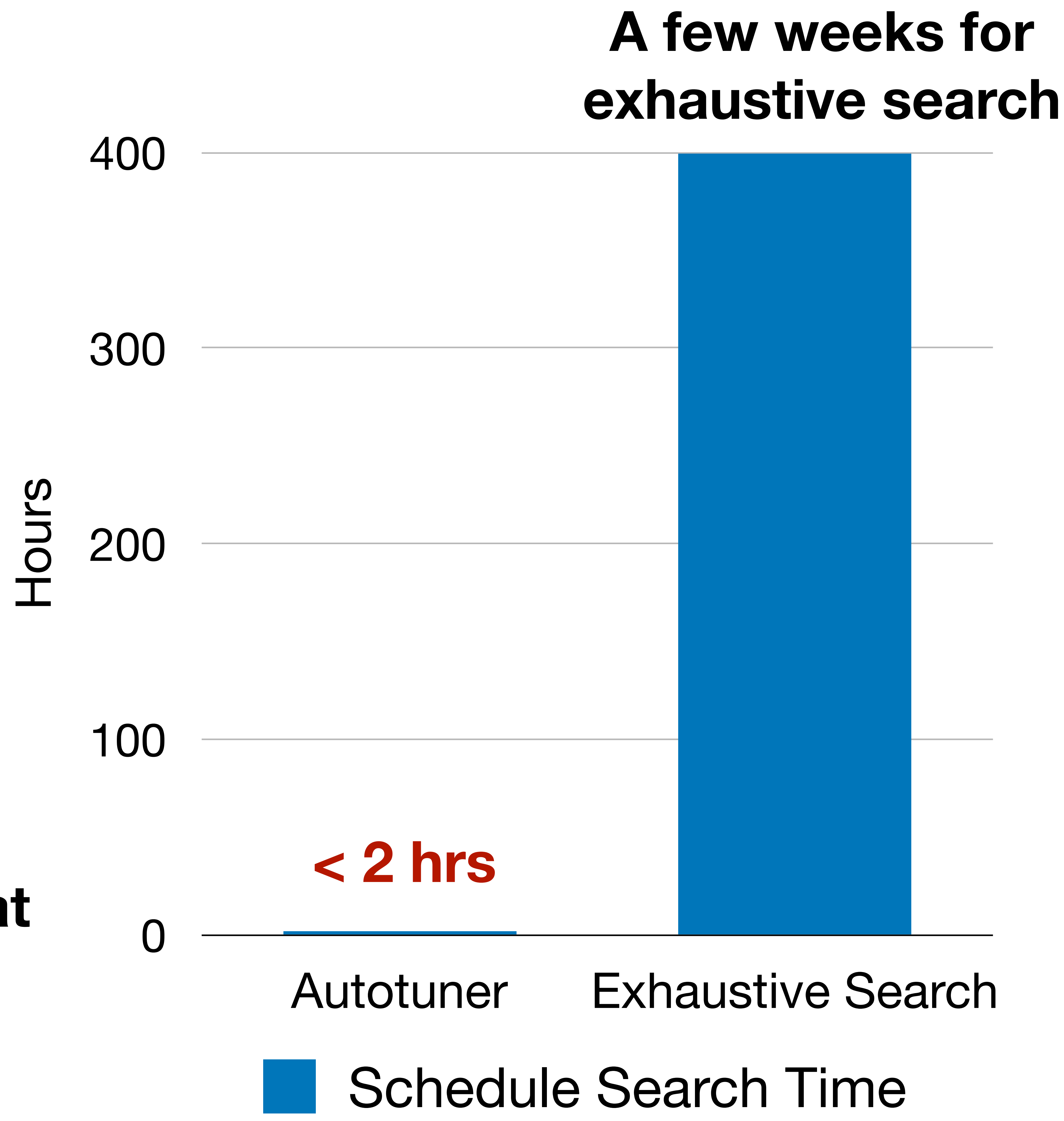

A few weeks for exhaustive search
400
300
200
100
0
Hours
Exhaustive Search
Schedule Search Time

Autotuner



Uses an ensemble of search methods. Build on top of OpenTuner [PACT14]

Autotuner



Finds a few schedules that outperform hand-tuned schedules

State of the Art and GraphIt

	PR	BFS	CC	SSSP
LJ	3.48	1	1	1
TW	5.63	1.13	3.12	1.14
WB	4.15	1.42	2.96	1.13
RD	2.69	4.81	2.16	4.57
FT	6.17	1.38	4.94	2.77

Ligra
(PPoPP13)

	PR	BFS	CC	SSSP
LJ	1.64	3.7	5.98	1.86
TW	2.34	9.4	11	1.62
WB	2.14	7.44	9.13	2.98
RD	1.61	9.06	7.04	151
FT				

GraphMat
(VLDB15)

	PR	BFS	CC	SSSP
LJ	1.51	1.83	3.06	1.82
TW	2.42	6.03	5.78	1.41
WB	2.59	2.84	5.96	2.54
RD	1.26	2.45	8.99	328
FT				

GreenMarl
(ASPLOS12)

	PR	BFS	CC	SSSP
LJ	8.15	1.41	2.05	1.78
TW	3.53	4.49	5.68	1.43
WB	2.82	1.83	8.07	1.36
RD	13	1.02	1.05	3.25
FT	3.61	7.02	7.05	1.08

Galois
(SOSP13)

	PR	BFS	CC	SSSP
LJ	1.26	2.22	2.46	1.57
TW	1.26	1.64	4.33	1
WB	1	1.52	4.93	1.67
RD	1.49	48.8	7.08	26.1
FT	1.37	1.49	5.24	1.43

Gemini
(OSDI16)

	PR	BFS	CC	SSSP
LJ	1.08	1.93	1.38	
TW	1.8	1.17	1.94	
WB	1.26	1.28	1.64	
RD	1	8.26	1	
FT	1.67	1.04	2.24	

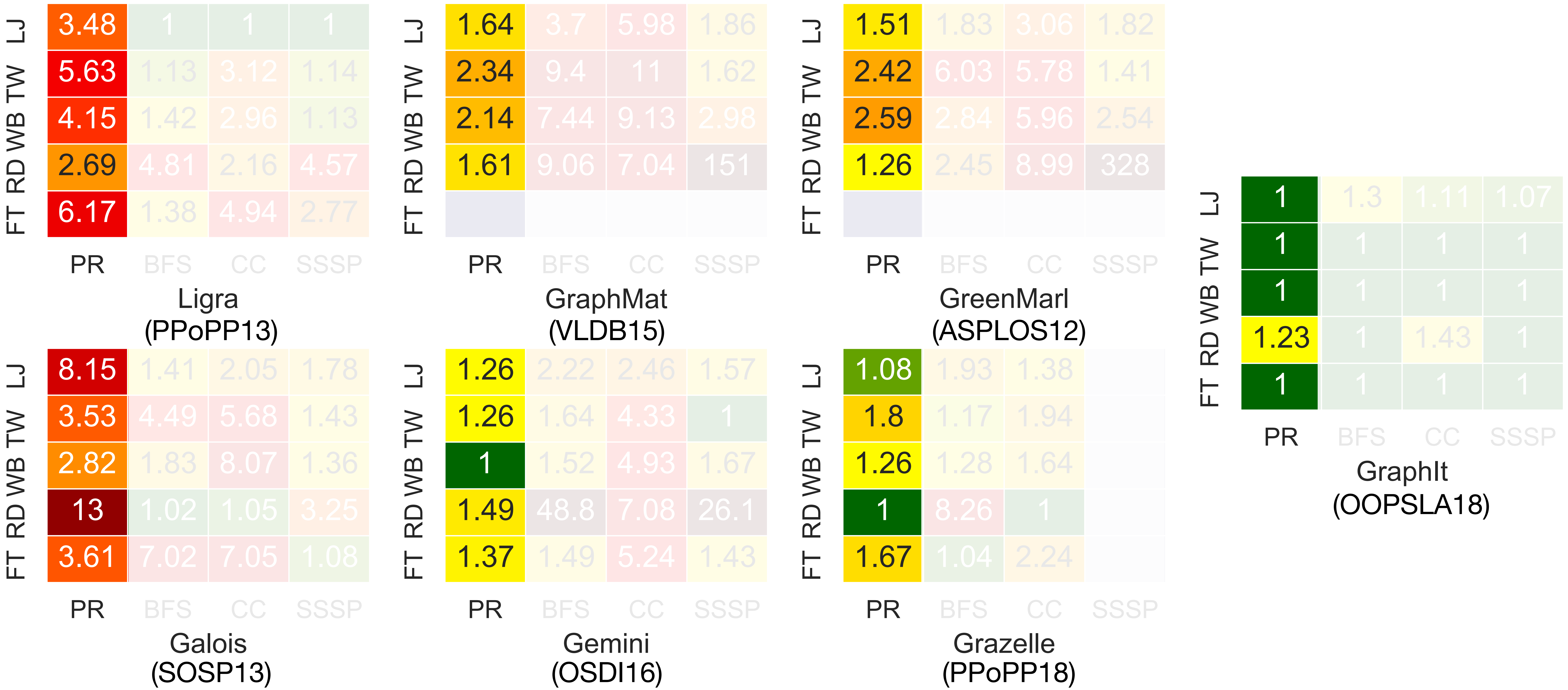
Grazelle
(PPoPP18)

	PR	BFS	CC	SSSP
LJ	1	1.3	1.11	1.07
TW	1	1	1	1
WB	1	1	1	1
RD	1.23	1	1.43	1
FT	1	1	1	1

GraphIt
(OOPSLA18)

Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores and 48 hyper-threads.

State of the Art and GraphIt



	PR	BFS	CC	SSSP
LJ	3.48	1	1	1
TW	5.63	1.13	3.12	1.14
WB	4.15	1.42	2.96	1.13
RD	2.69	4.81	2.16	4.57
FT	6.17	1.38	4.94	2.77

Ligra
(PPoPP13)

	PR	BFS	CC	SSSP
LJ	1.64	3.7	5.98	1.86
TW	2.34	9.4	11	1.62
WB	2.14	7.44	9.13	2.98
RD	1.61	9.06	7.04	151
FT				

GraphMat
(VLDB15)

	PR	BFS	CC	SSSP
LJ	1.51	1.83	3.06	1.82
TW	2.42	6.03	5.78	1.41
WB	2.59	2.84	5.96	2.54
RD	1.26	2.45	8.99	328
FT				

GreenMarl
(ASPLOS12)

	PR	BFS	CC	SSSP
LJ	8.15	1.41	2.05	1.78
TW	3.53	4.49	5.68	1.43
WB	2.82	1.83	8.07	1.36
RD	13	1.02	1.05	3.25
FT	3.61	7.02	7.05	1.08

Galois
(SOSP13)

	PR	BFS	CC	SSSP
LJ	1.26	2.22	2.46	1.57
TW	1.26	1.64	4.33	1
WB	1	1.52	4.93	1.67
RD	1.49	48.8	7.08	26.1
FT	1.37	1.49	5.24	1.43

Gemini
(OSDI16)

	PR	BFS	CC	SSSP
LJ	1.08	1.93	1.38	
TW	1.8	1.17	1.94	
WB	1.26	1.28	1.64	
RD	1	8.26	1	
FT	1.67	1.04	2.24	

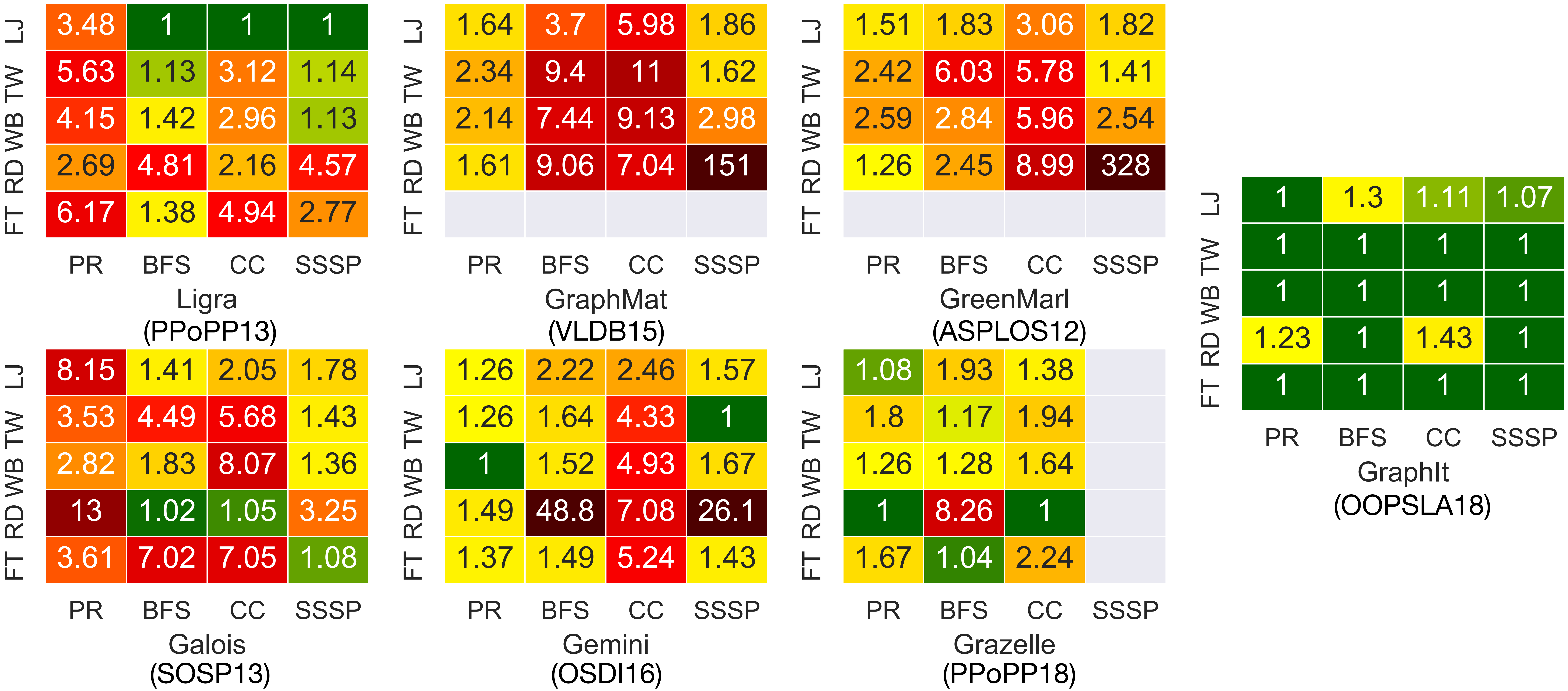
Grazelle
(PPoPP18)

	PR	BFS	CC	SSSP
LJ	1	1.3	1.11	1.07
TW	1	1	1	1
WB	1	1	1	1
RD	1.23	1	1.43	1
FT	1	1	1	1

GraphIt
(OOPSLA18)

Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores and 48 hyper-threads.

State of the Art and GraphIt



Ligra (PPoPP13)

	PR	BFS	CC	SSSP
LJ	3.48	1	1	1
TW	5.63	1.13	3.12	1.14
WB	4.15	1.42	2.96	1.13
RD	2.69	4.81	2.16	4.57
FT	6.17	1.38	4.94	2.77

GraphMat (VLDB15)

	PR	BFS	CC	SSSP
LJ	1.64	3.7	5.98	1.86
TW	2.34	9.4	11	1.62
WB	2.14	7.44	9.13	2.98
RD	1.61	9.06	7.04	151
FT				

GreenMarl (ASPLOS12)

	PR	BFS	CC	SSSP
LJ	1.51	1.83	3.06	1.82
TW	2.42	6.03	5.78	1.41
WB	2.59	2.84	5.96	2.54
RD	1.26	2.45	8.99	328
FT				

GraphIt (OOPSLA18)

	PR	BFS	CC	SSSP
LJ	1	1.3	1.11	1.07
TW	1	1	1	1
WB	1	1	1	1
RD	1.23	1	1.43	1
FT	1	1	1	1

Galois (SOSP13)

	PR	BFS	CC	SSSP
LJ	8.15	1.41	2.05	1.78
TW	3.53	4.49	5.68	1.43
WB	2.82	1.83	8.07	1.36
RD	13	1.02	1.05	3.25
FT	3.61	7.02	7.05	1.08

Gemini (OSDI16)

	PR	BFS	CC	SSSP
LJ	1.26	2.22	2.46	1.57
TW	1.26	1.64	4.33	1
WB	1	1.52	4.93	1.67
RD	1.49	48.8	7.08	26.1
FT	1.37	1.49	5.24	1.43

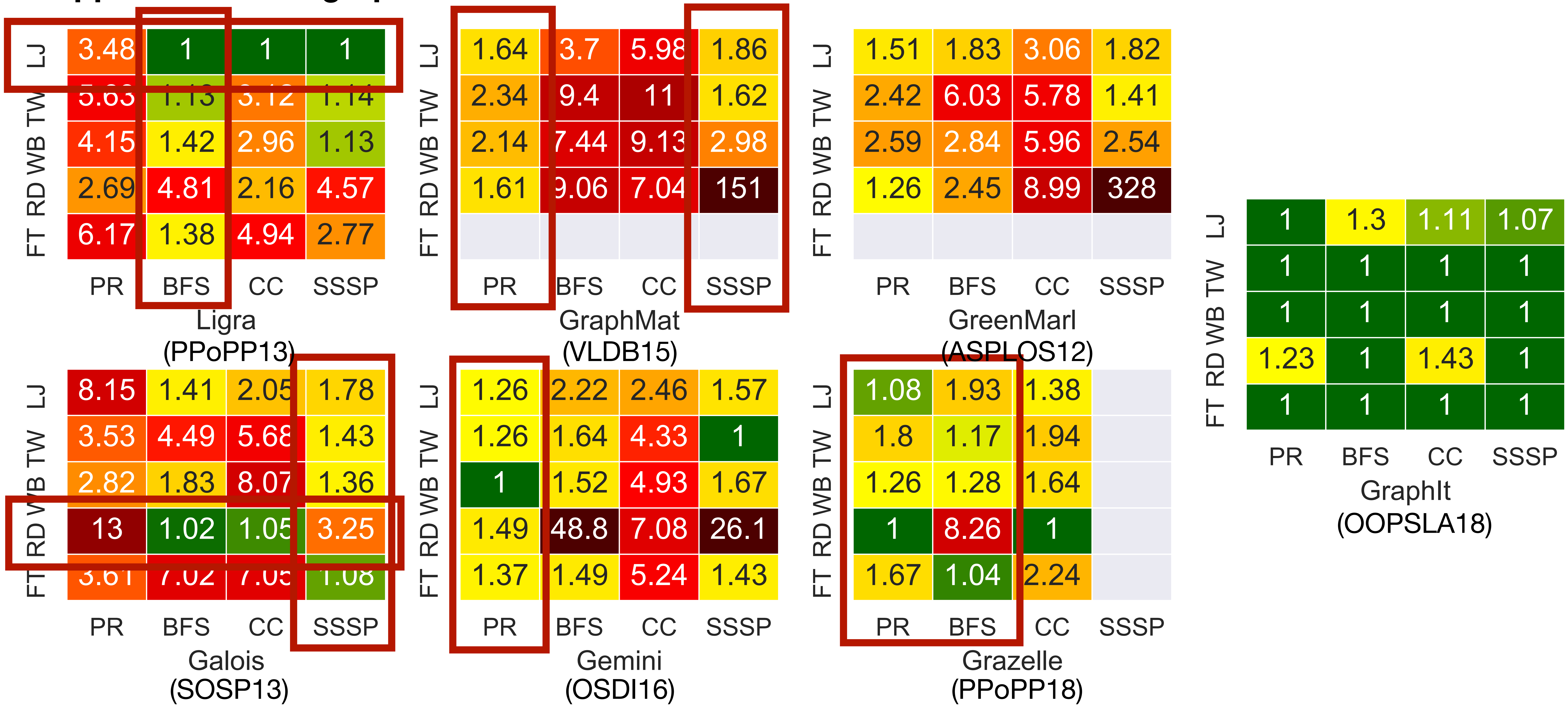
Grazelle (PPoPP18)

	PR	BFS	CC	SSSP
LJ	1.08	1.93	1.38	
TW	1.8	1.17	1.94	
WB	1.26	1.28	1.64	
RD	1	8.26	1	
FT	1.67	1.04	2.24	

Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores and 48 hyper-threads.

State of the Art and GraphIt

Most frameworks are good at certain applications and graphs



	PR	BFS	CC	SSSP
LJ	3.48	1	1	1
TW	5.63	1.13	3.12	1.14
WB	4.15	1.42	2.96	1.13
RD	2.69	4.81	2.16	4.57
FT	6.17	1.38	4.94	2.77

Ligra (PPoPP13)

	PR	BFS	CC	SSSP
LJ	1.64	3.7	5.98	1.86
TW	2.34	9.4	11	1.62
WB	2.14	7.44	9.13	2.98
RD	1.61	9.06	7.04	151
FT				

GraphMat (VLDB15)

	PR	BFS	CC	SSSP
LJ	1.51	1.83	3.06	1.82
TW	2.42	6.03	5.78	1.41
WB	2.59	2.84	5.96	2.54
RD	1.26	2.45	8.99	328
FT				

GreenMarl (ASPLOS12)

	PR	BFS	CC	SSSP
LJ	1	1.3	1.11	1.07
TW	1	1	1	1
WB	1	1	1	1
RD	1.23	1	1.43	1
FT	1	1	1	1

GraphIt (OOPSLA18)

	PR	BFS	CC	SSSP
LJ	8.15	1.41	2.05	1.78
TW	3.53	4.49	5.68	1.43
WB	2.82	1.83	8.07	1.36
RD	13	1.02	1.05	3.25
FT	3.61	7.02	7.05	1.08

Galois (SOSP13)

	PR	BFS	CC	SSSP
LJ	1.26	2.22	2.46	1.57
TW	1.26	1.64	4.33	1
WB	1	1.52	4.93	1.67
RD	1.49	48.8	7.08	26.1
FT	1.37	1.49	5.24	1.43

Gemini (OSDI16)

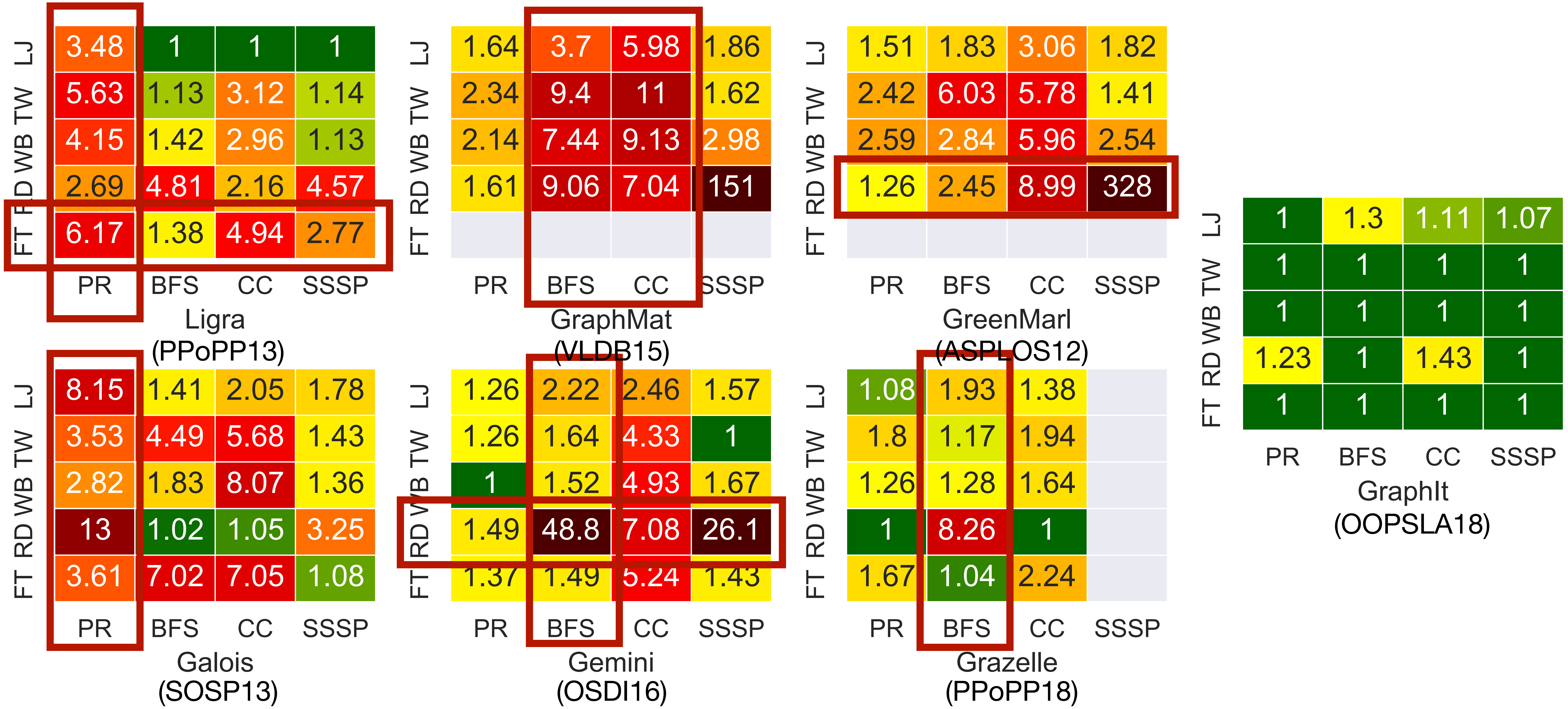
	PR	BFS	CC	SSSP
LJ	1.08	1.93	1.38	
TW	1.8	1.17	1.94	
WB	1.26	1.28	1.64	
RD	1	8.26	1	
FT	1.67	1.04	2.24	

Grazelle (PPoPP18)

Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores and 48 hyper-threads.

State of the Art and GraphIt

Most frameworks are bad at certain applications and graphs



	PR	BFS	CC	SSSP
LJ	3.48	1	1	1
TW	5.63	1.13	3.12	1.14
WB	4.15	1.42	2.96	1.13
RD	2.69	4.81	2.16	4.57
FT	6.17	1.38	4.94	2.77

Ligra (PPoPP13)

	PR	BFS	CC	SSSP
LJ	1.64	3.7	5.98	1.86
TW	2.34	9.4	11	1.62
WB	2.14	7.44	9.13	2.98
RD	1.61	9.06	7.04	151
FT				

GraphMat (VLDB15)

	PR	BFS	CC	SSSP
LJ	1.51	1.83	3.06	1.82
TW	2.42	6.03	5.78	1.41
WB	2.59	2.84	5.96	2.54
RD	1.26	2.45	8.99	328
FT				

GreenMarl (ASPLOS12)

	PR	BFS	CC	SSSP
LJ	1	1.3	1.11	1.07
TW	1	1	1	1
WB	1	1	1	1
RD	1.23	1	1.43	1
FT	1	1	1	1

GraphIt (OOPSLA18)

	PR	BFS	CC	SSSP
LJ	8.15	1.41	2.05	1.78
TW	3.53	4.49	5.68	1.43
WB	2.82	1.83	8.07	1.36
RD	13	1.02	1.05	3.25
FT	3.61	7.02	7.05	1.08

Galois (SOSP13)

	PR	BFS	CC	SSSP
LJ	1.26	2.22	2.46	1.57
TW	1.26	1.64	4.33	1
WB	1	1.52	4.93	1.67
RD	1.49	48.8	7.08	26.1
FT	1.37	1.49	5.24	1.43

Gemini (OSDI16)

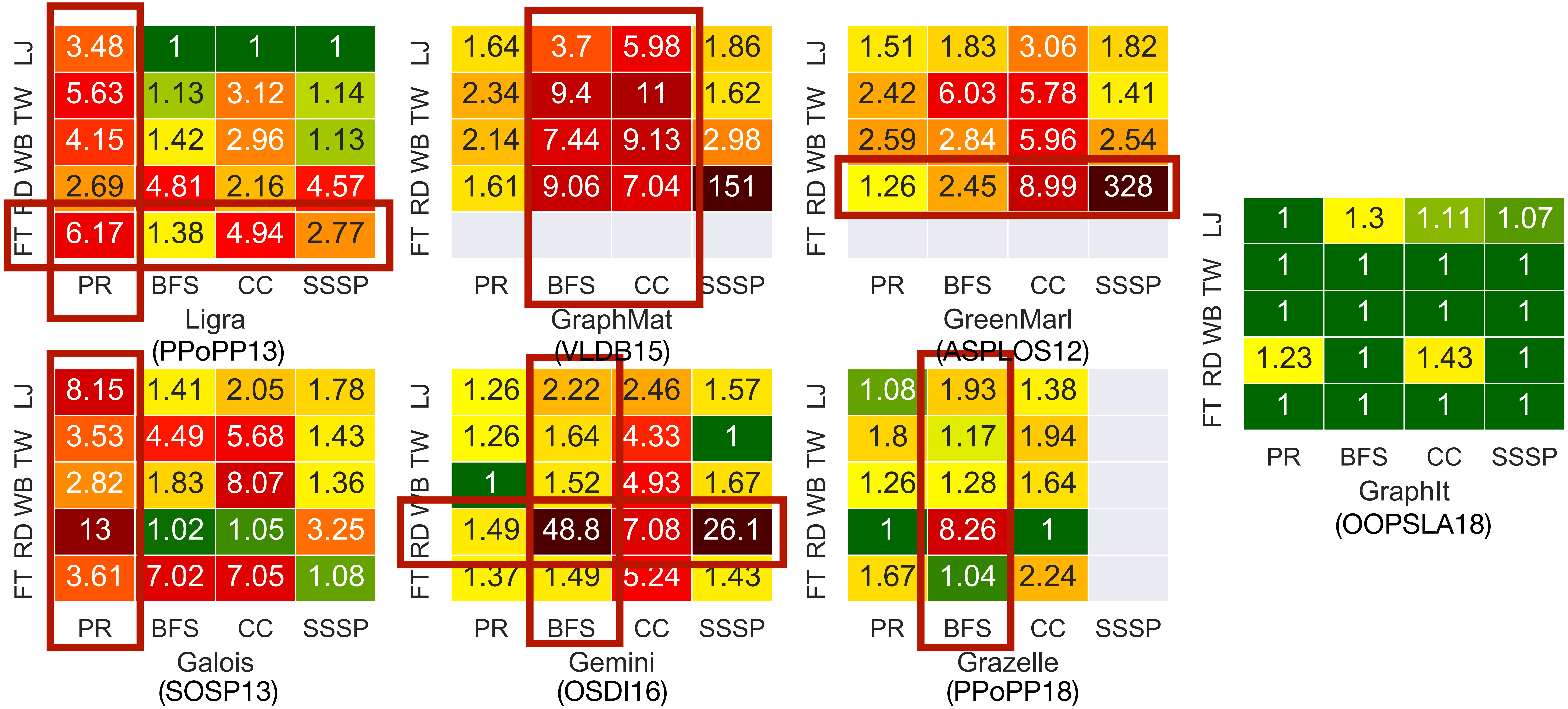
	PR	BFS	CC	SSSP
LJ	1.08	1.93	1.38	
TW	1.8	1.17	1.94	
WB	1.26	1.28	1.64	
RD	1	8.26	1	
FT	1.67	1.04	2.24	

Grazelle (PPoPP18)

Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores and 48 hyper-threads.

State of the Art and GraphIt

Previous work support a subset of optimizations



	PR	BFS	CC	SSSP
LJ	3.48	1	1	1
TW	5.63	1.13	3.12	1.14
WB	4.15	1.42	2.96	1.13
RD	2.69	4.81	2.16	4.57
FT	6.17	1.38	4.94	2.77

Ligra (PPoPP13)

	PR	BFS	CC	SSSP
LJ	1.64	3.7	5.98	1.86
TW	2.34	9.4	11	1.62
WB	2.14	7.44	9.13	2.98
RD	1.61	9.06	7.04	151
FT				

GraphMat (VLDB15)

	PR	BFS	CC	SSSP
LJ	1.51	1.83	3.06	1.82
TW	2.42	6.03	5.78	1.41
WB	2.59	2.84	5.96	2.54
RD	1.26	2.45	8.99	328
FT				

GreenMarl (ASPLOS12)

	PR	BFS	CC	SSSP
LJ	1	1.3	1.11	1.07
TW	1	1	1	1
WB	1	1	1	1
RD	1.23	1	1.43	1
FT	1	1	1	1

GraphIt (OOPSLA18)

	PR	BFS	CC	SSSP
LJ	8.15	1.41	2.05	1.78
TW	3.53	4.49	5.68	1.43
WB	2.82	1.83	8.07	1.36
RD	13	1.02	1.05	3.25
FT	3.61	7.02	7.05	1.08

Galois (SOSP13)

	PR	BFS	CC	SSSP
LJ	1.26	2.22	2.46	1.57
TW	1.26	1.64	4.33	1
WB	1	1.52	4.93	1.67
RD	1.49	48.8	7.08	26.1
FT	1.37	1.49	5.24	1.43

Gemini (OSDI16)

	PR	BFS	CC	SSSP
LJ	1.08	1.93	1.38	
TW	1.8	1.17	1.94	
WB	1.26	1.28	1.64	
RD	1	8.26	1	
FT	1.67	1.04	2.24	

Grazelle (PPoPP18)

Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores and 48 hyper-threads.

Consistent High-Performance

	PR	BFS	CC	SSSP
LJ	3.48	1	1	1
TW	5.63	1.13	3.12	1.14
WB	4.15	1.42	2.96	1.13
RD	2.69	4.81	2.16	4.57
FT	6.17	1.38	4.94	2.77

Ligra
(PPoPP13)

	PR	BFS	CC	SSSP
LJ	1.64	3.7	5.98	1.86
TW	2.34	9.4	11	1.62
WB	2.14	7.44	9.13	2.98
RD	1.61	9.06	7.04	151
FT				

GraphMat
(VLDB15)

	PR	BFS	CC	SSSP
LJ	1.51	1.83	3.06	1.82
TW	2.42	6.03	5.78	1.41
WB	2.59	2.84	5.96	2.54
RD	1.26	2.45	8.99	328
FT				

GreenMarl
(ASPLOS12)

Good across different applications and graphs

	PR	BFS	CC	SSSP
LJ	1	1.3	1.11	1.07
TW	1	1	1	1
WB	1	1	1	1
RD	1.23	1	1.43	1
FT	1	1	1	1

GraphIt
(OOPSLA18)

	PR	BFS	CC	SSSP
LJ	8.15	1.41	2.05	1.78
TW	3.53	4.49	5.68	1.43
WB	2.82	1.83	8.07	1.36
RD	13	1.02	1.05	3.25
FT	3.61	7.02	7.05	1.08

Galois
(SOSP13)

	PR	BFS	CC	SSSP
LJ	1.26	2.22	2.46	1.57
TW	1.26	1.64	4.33	1
WB	1	1.52	4.93	1.67
RD	1.49	48.8	7.08	26.1
FT	1.37	1.49	5.24	1.43

Gemini
(OSDI16)

	PR	BFS	CC	SSSP
LJ	1.08	1.93	1.38	
TW	1.8	1.17	1.94	
WB	1.26	1.28	1.64	
RD	1	8.26	1	
FT	1.67	1.04	2.24	

Grazelle
(PPoPP18)

Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores and 48 hyper-threads.

Speedup over State of the Art

	PR	BFS	CC	SSSP
LJ	3.48	1	1	1
TW	5.63	1.13	3.12	1.14
WB	4.15	1.42	2.96	1.13
RD	2.69	4.81	2.16	4.57
FT	6.17	1.38	4.94	2.77

Ligra
(PPoPP13)

	PR	BFS	CC	SSSP
LJ	1.64	3.7	5.98	1.86
TW	2.34	9.4	11	1.62
WB	2.14	7.44	9.13	2.98
RD	1.61	9.06	7.04	151
FT				

GraphMat
(VLDB15)

	PR	BFS	CC	SSSP
LJ	1.51	1.83	3.06	1.82
TW	2.42	6.03	5.78	1.41
WB	2.59	2.84	5.96	2.54
RD	1.26	2.45	8.99	328
FT				

GreenMarl
(ASPLOS12)

Finds previously unexplored combinations of optimizations

	PR	BFS	CC	SSSP
LJ	1	1.3	1.11	1.07
TW	1	1	1	1
WB	1	1	1	1
RD	1.23	1	1.43	1
FT	1	1	1	1

GraphIt
(OOPSLA18)

	PR	BFS	CC	SSSP
LJ	8.15	1.41	2.05	1.78
TW	3.53	4.49	5.68	1.43
WB	2.82	1.83	8.07	1.36
RD	13	1.02	1.05	3.25
FT	3.61	7.02	7.05	1.08

Galois
(SOSP13)

	PR	BFS	CC	SSSP
LJ	1.26	2.22	2.46	1.57
TW	1.26	1.64	4.33	1
WB	1	1.52	4.93	1.67
RD	1.49	48.8	7.08	26.1
FT	1.37	1.49	5.24	1.43

Gemini
(OSDI16)

	PR	BFS	CC	SSSP
LJ	1.08	1.93	1.38	
TW	1.8	1.17	1.94	
WB	1.26	1.28	1.64	
RD	1	8.26	1	
FT	1.67	1.04	2.24	

Grazelle
(PPoPP18)

Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores and 48 hyper-threads.

Ease-of-Use

Ligra (PPoPP13)

	PR	BFS	CC	SSSP
LJ	3.48	1	1	1
TW	5.63	1.13	3.12	1.14
WB	4.15	1.42	2.96	1.13
RD	2.69	4.81	2.16	4.57
FT	6.17	1.38	4.94	2.77

GraphMat (VLDB15)

	PR	BFS	CC	SSSP
LJ	1.64	3.7	5.98	1.86
TW	2.34	9.4	11	1.62
WB	2.14	7.44	9.13	2.98
RD	1.61	9.06	7.04	151
FT				

GreenMarl (ASPLOS12)

	PR	BFS	CC	SSSP
LJ	1.51	1.83	3.06	1.82
TW	2.42	6.03	5.78	1.41
WB	2.59	2.84	5.96	2.54
RD	1.26	2.45	8.99	328
FT				

Reduces the lines of code by an order of magnitude compare to the next fastest framework

GraphIt (OOPSLA18)

	PR	BFS	CC	SSSP
LJ	1	1.3	1.11	1.07
TW	1	1	1	1
WB	1	1	1	1
RD	1.23	1	1.43	1
FT	1	1	1	1

Galois (SOSP13)

	PR	BFS	CC	SSSP
LJ	8.15	1.41	2.05	1.78
TW	3.53	4.49	5.68	1.43
WB	2.82	1.83	8.07	1.36
RD	13	1.02	1.05	3.25
FT	3.61	7.02	7.05	1.08

Gemini (OSDI16)

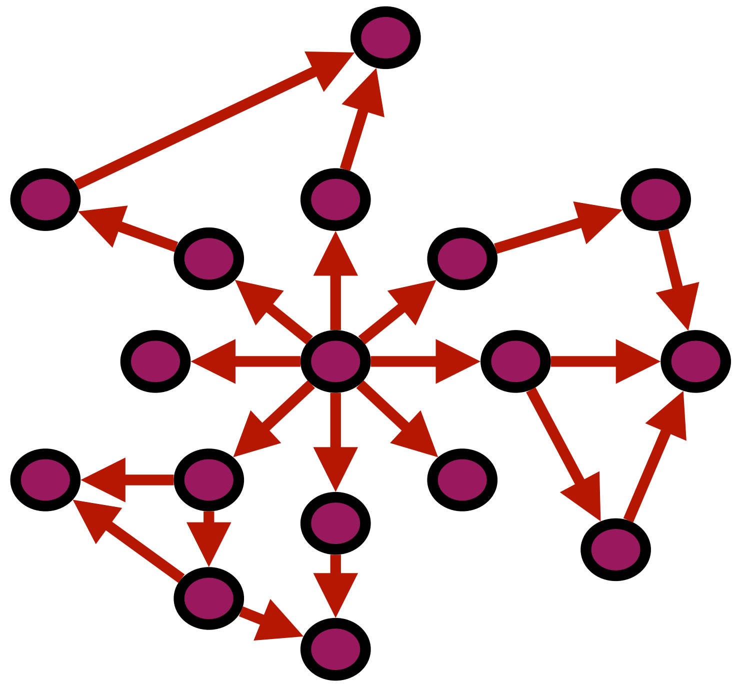
	PR	BFS	CC	SSSP
LJ	1.26	2.22	2.46	1.57
TW	1.26	1.64	4.33	1
WB	1	1.52	4.93	1.67
RD	1.49	48.8	7.08	26.1
FT	1.37	1.49	5.24	1.43

Grazelle (PPoPP18)

	PR	BFS	CC	SSSP
LJ	1.08	1.93	1.38	
TW	1.8	1.17	1.94	
WB	1.26	1.28	1.64	
RD	1	8.26	1	
FT	1.67	1.04	2.24	

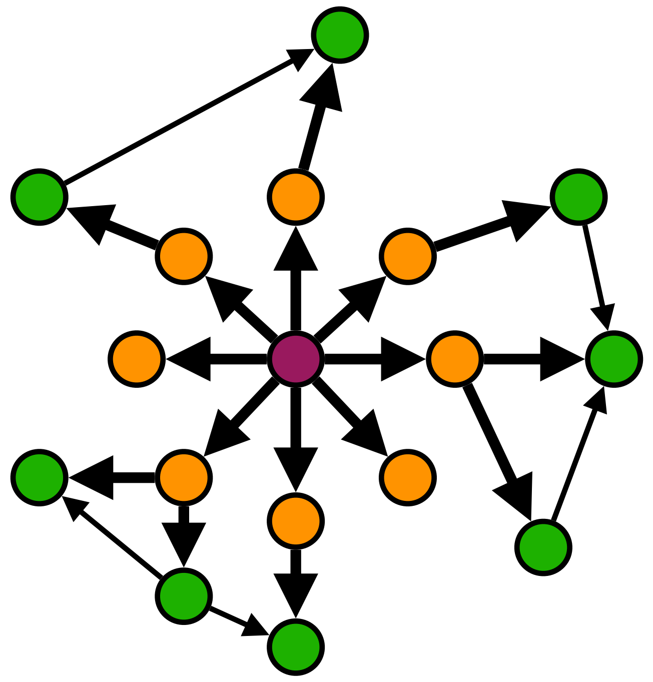
Intel Xeon E5-2695 v3 CPUs with 12 cores each for a total of 24 cores and 48 hyper-threads.

Topology-Driven

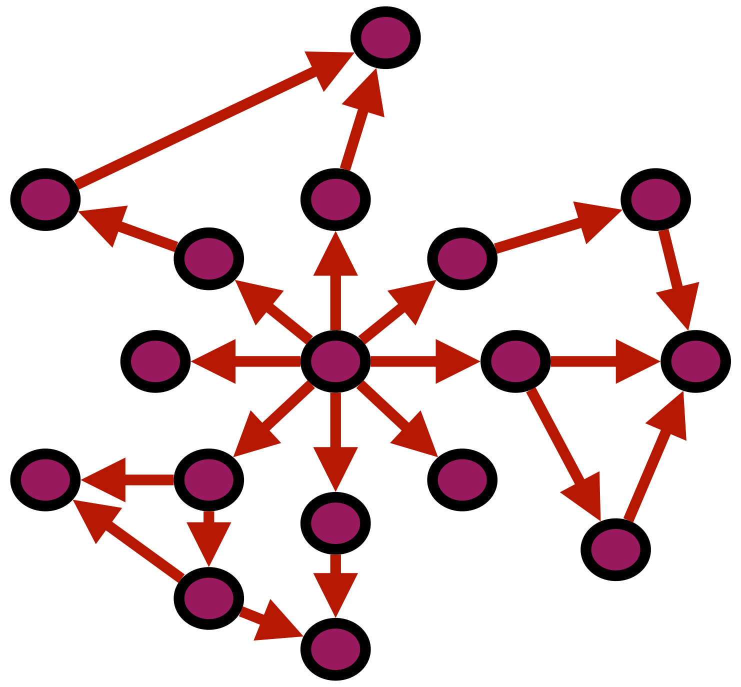


Algorithms

Data-Driven

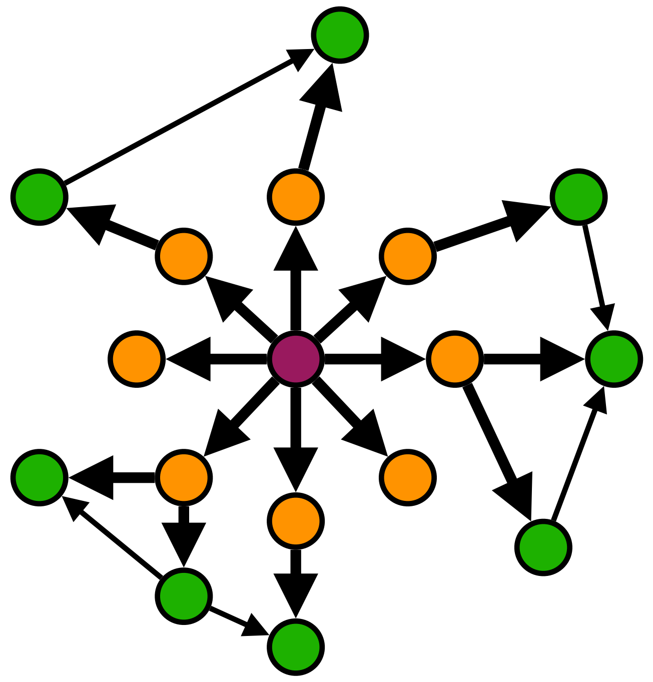


Topology-Driven

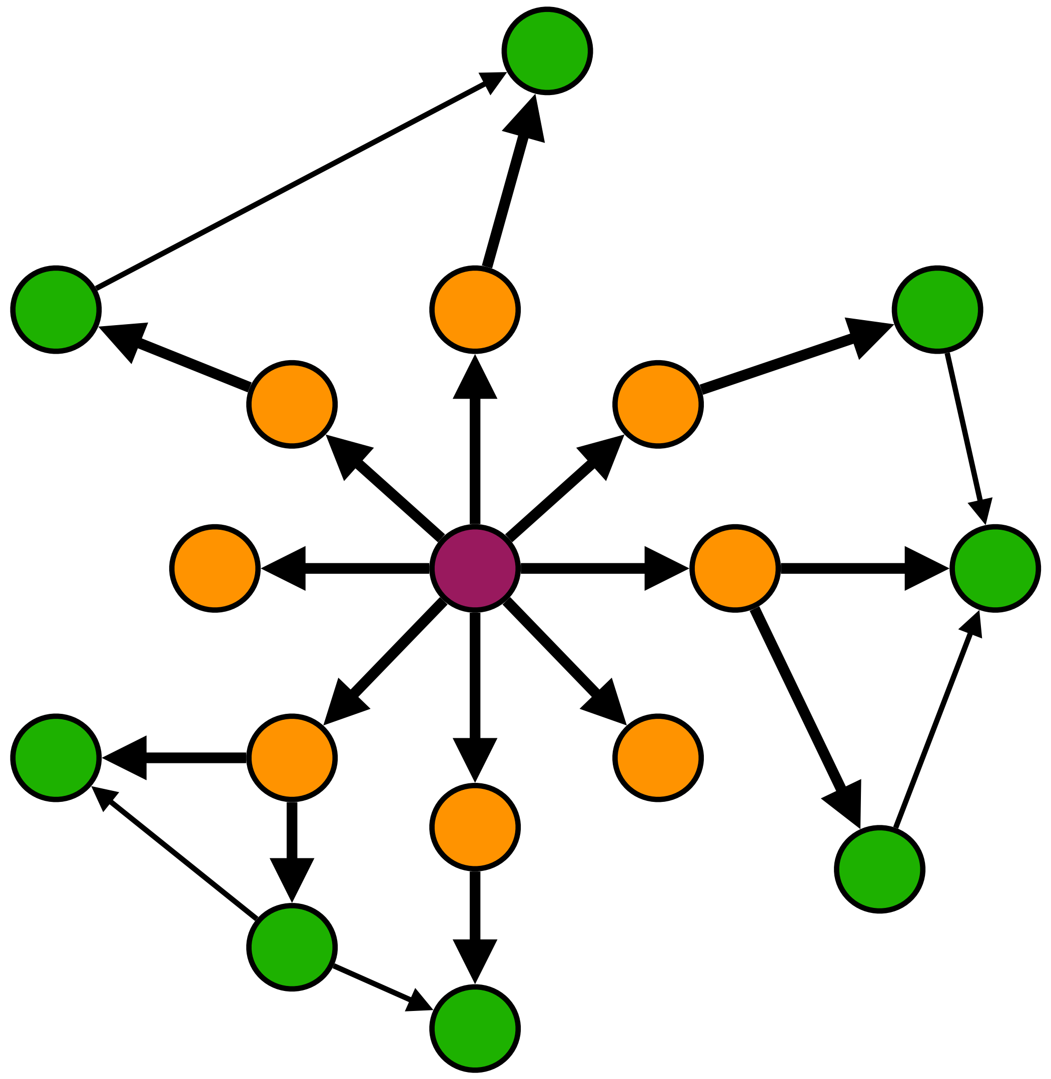


Algorithms

Data-Driven

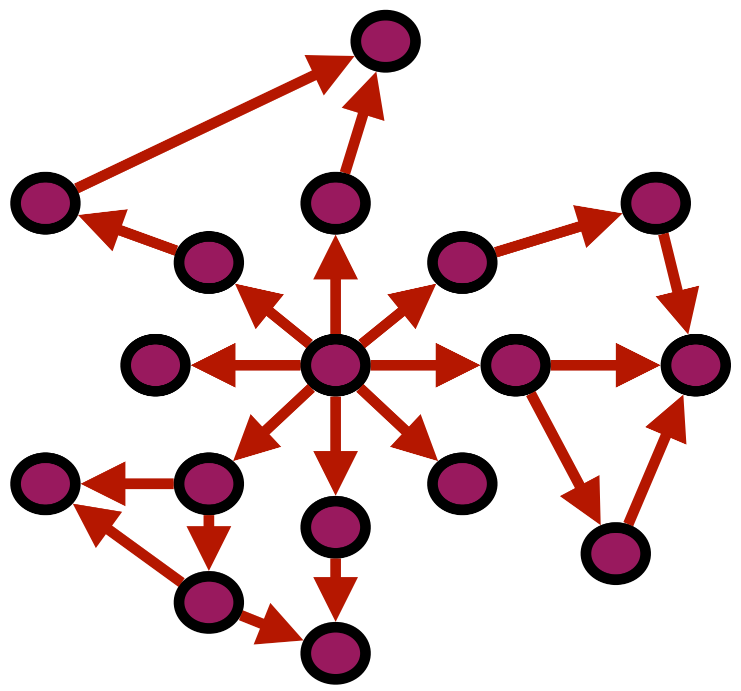


Unordered

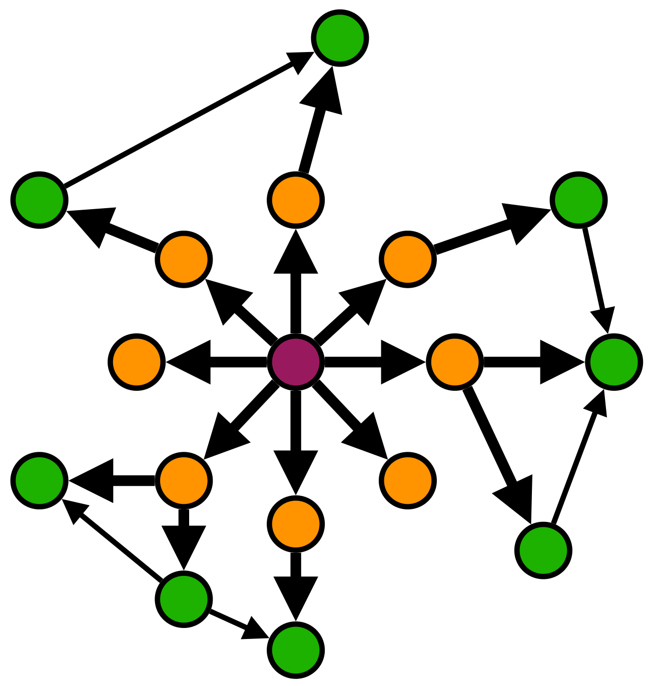


Active Vertices can be processed in parallel in arbitrary order

Topology-Driven

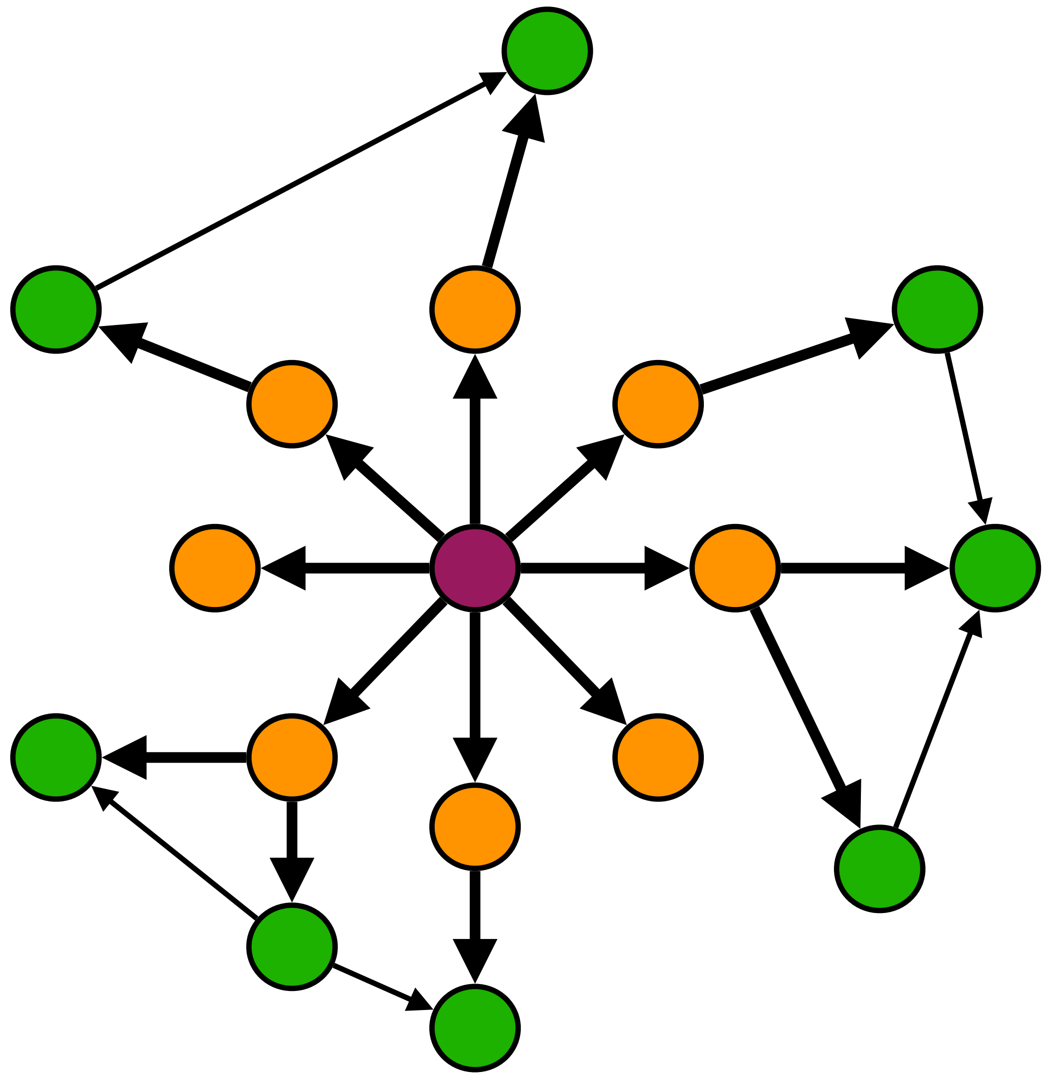


Data-Driven



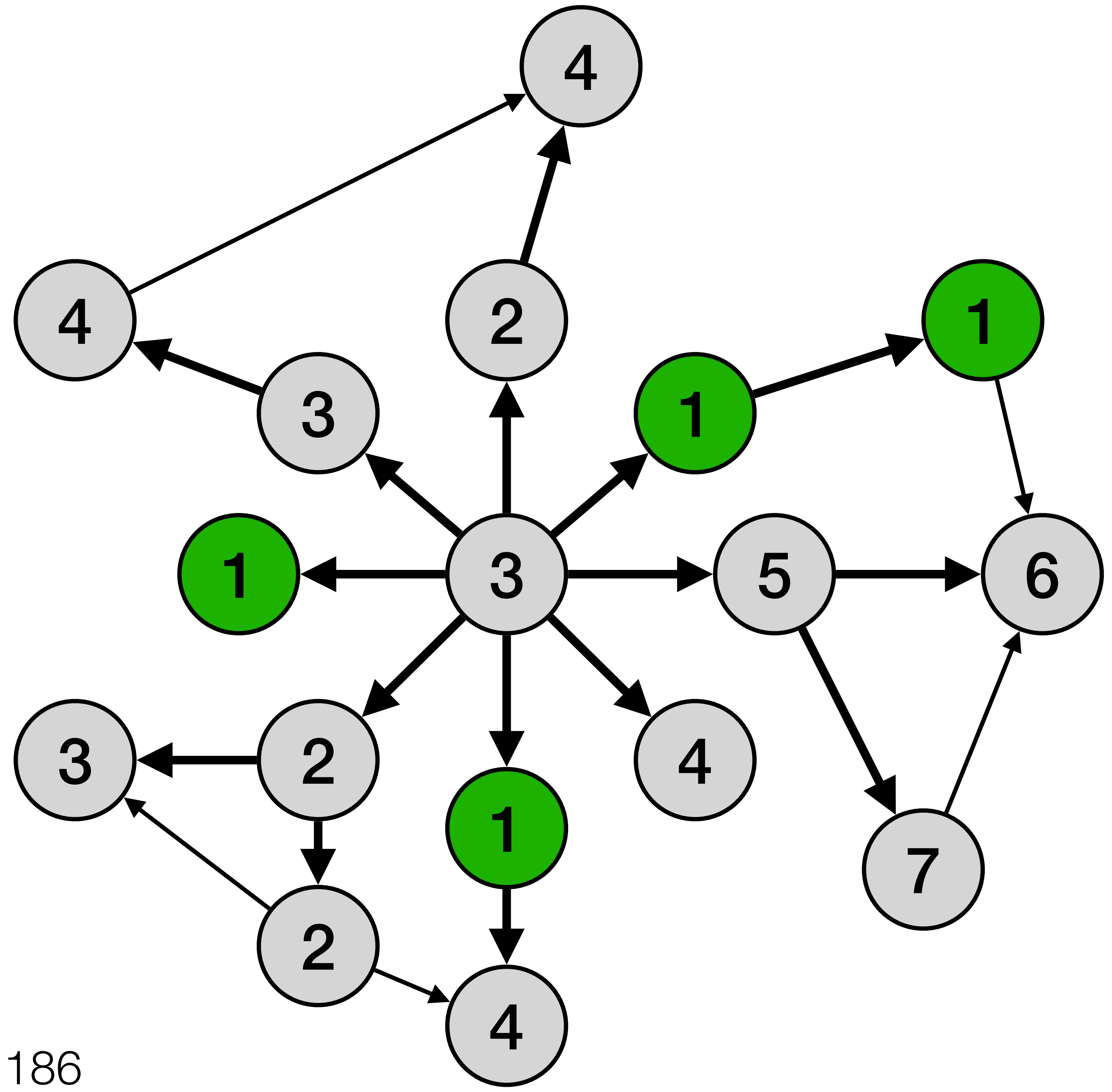
Algorithms

Unordered



Active Vertices can be processed in parallel in arbitrary order

Ordered

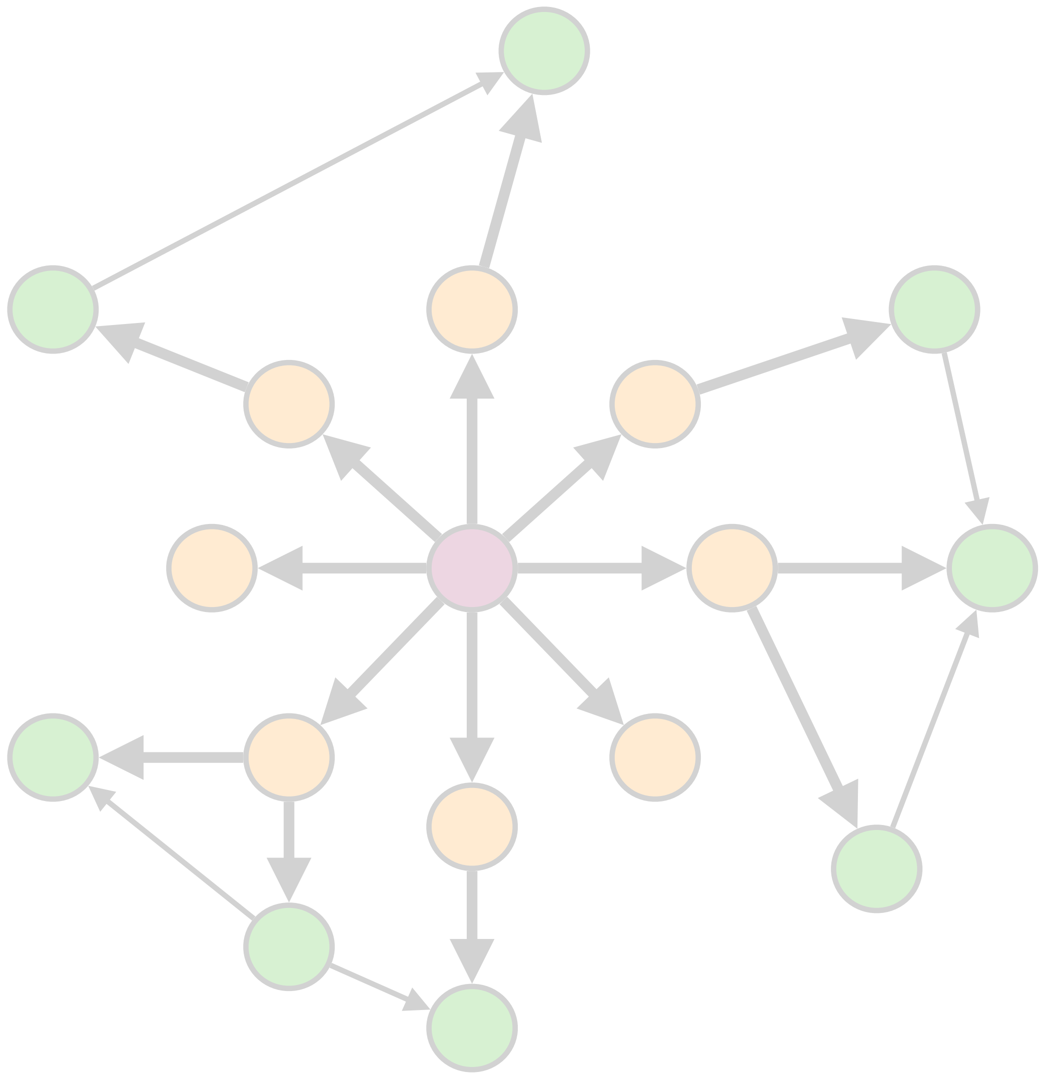


Vertices are processed according to priorities

Priorities can change dynamically

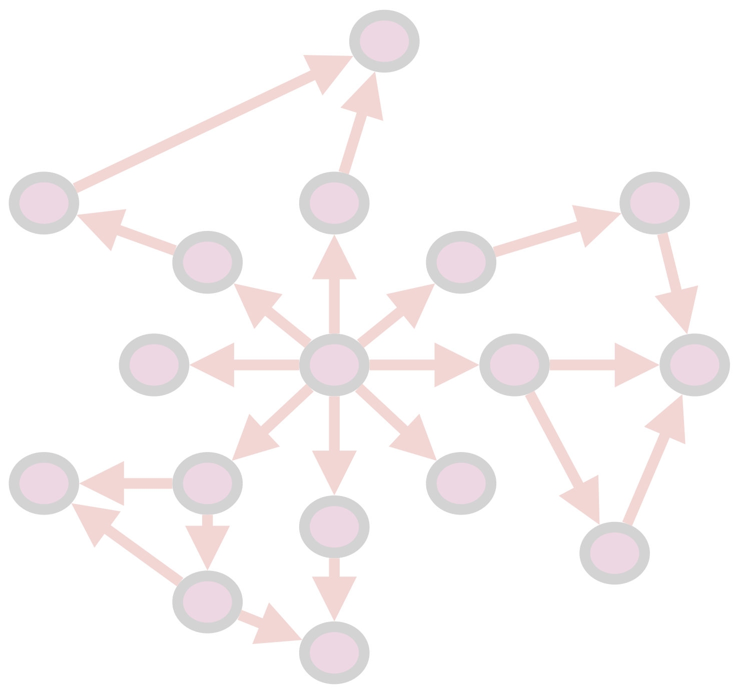
Vertices of the same priority are processed in parallel

Unordered



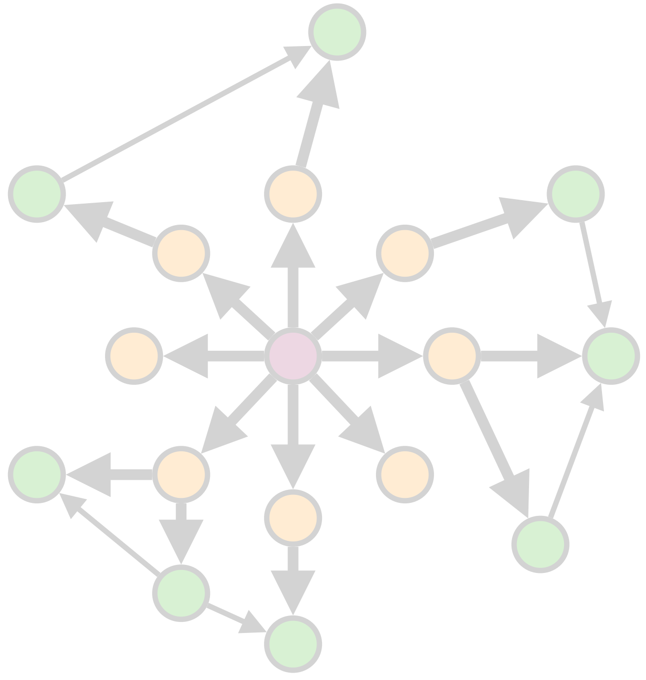
Active Vertices can be processed in parallel in arbitrary order

Topology-Driven

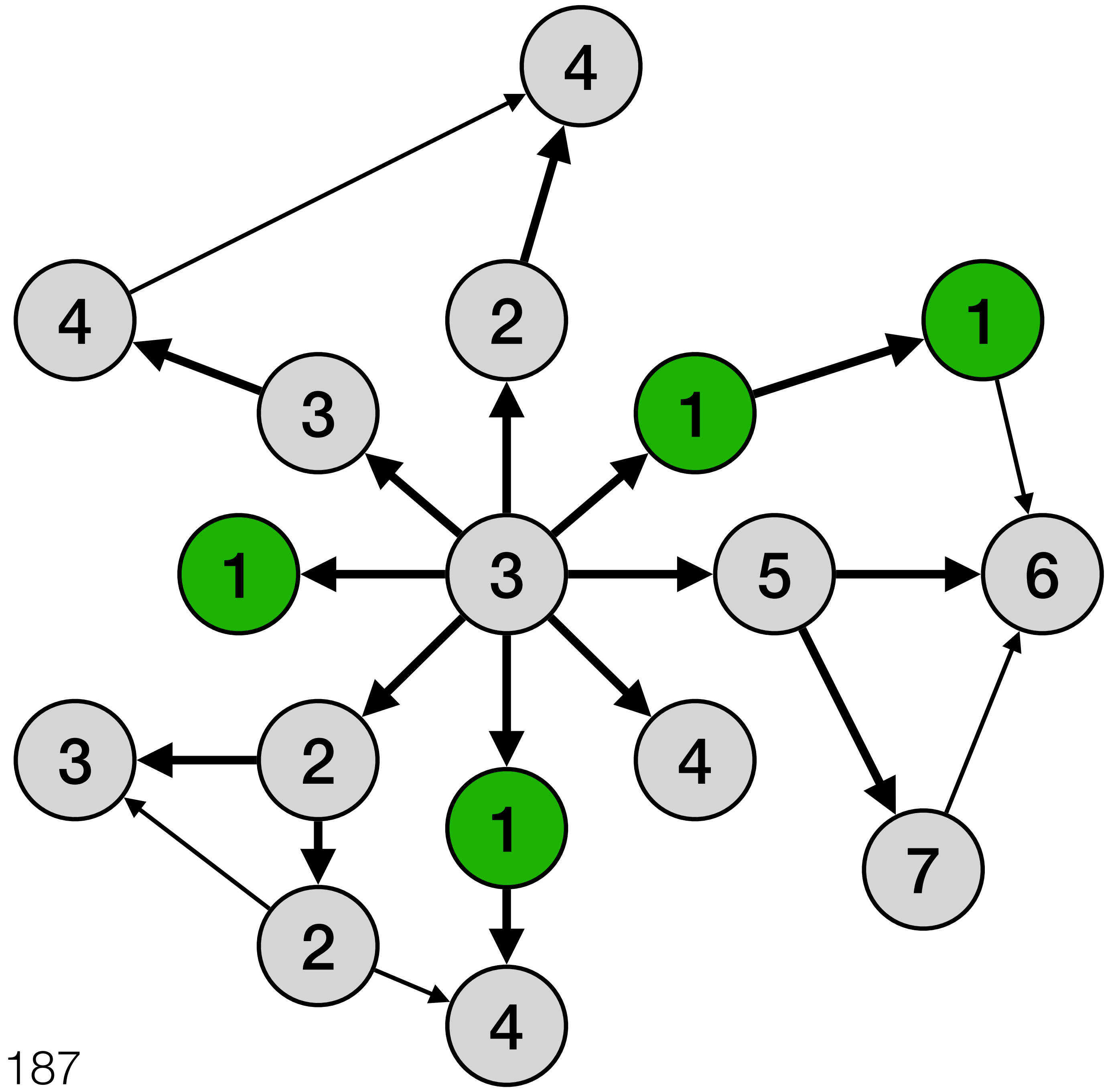


Algorithms

Data-Driven



Ordered



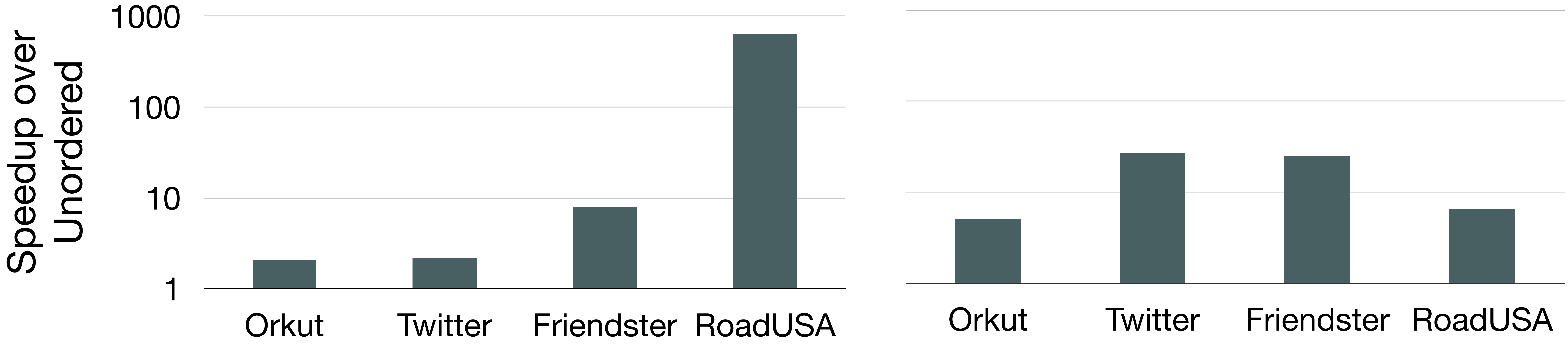
Vertices are processed according to priorities

Priorities can change dynamically

Vertices of the same priority are processed in parallel

Ordered vs Unordered

Ordered algorithms can often achieve 2x to 640x speedup over unordered counterparts

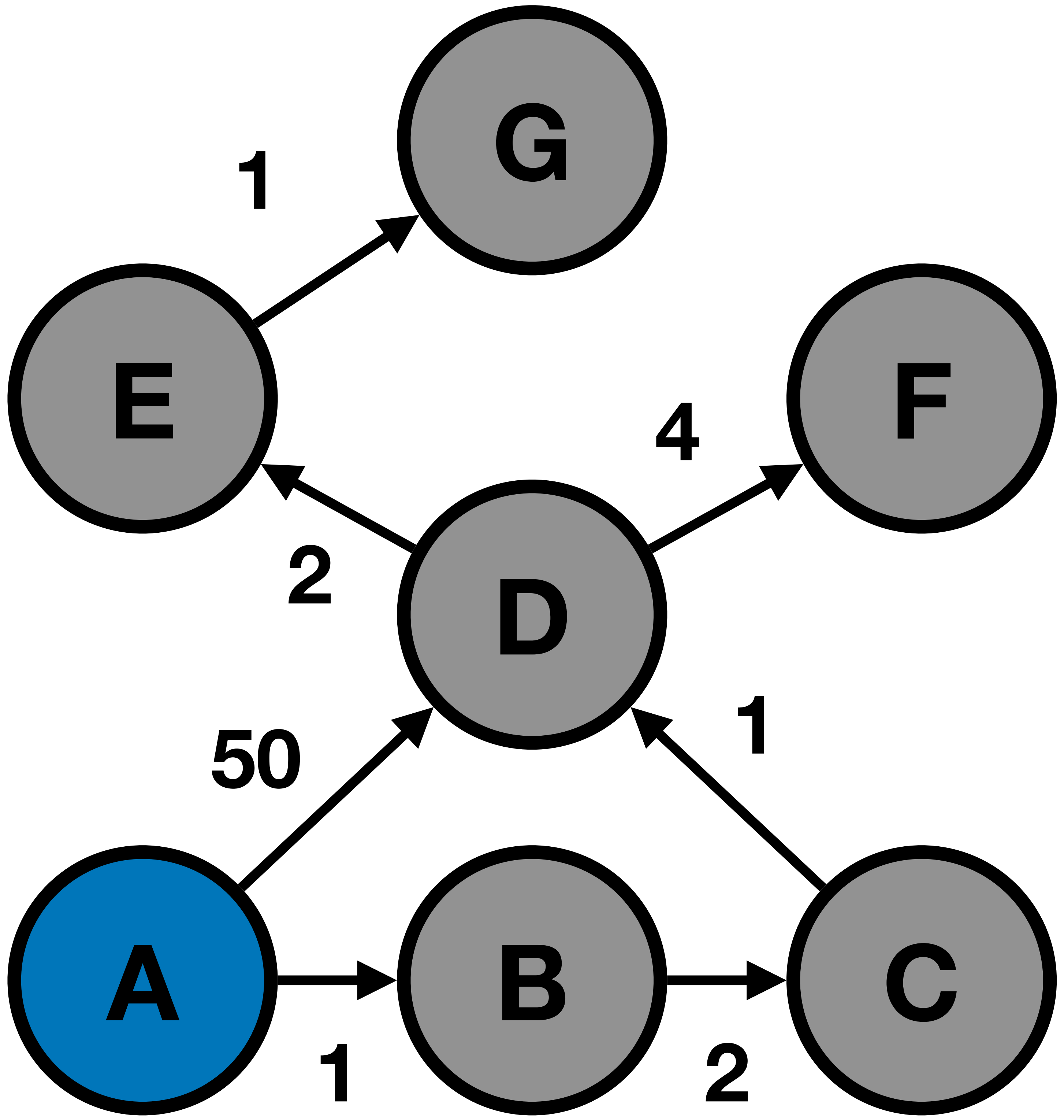


Single Source Shortest Paths

KCore

Speedup of Ordered vs Unordered on a 24-core CPU

Bellman-Ford (Unordered SSSP)



🔵 Start Vertex

⚫ Unreached Vertex

🟢 Vertex with Shortest Distance

🔴 Vertex with Suboptimal Distance

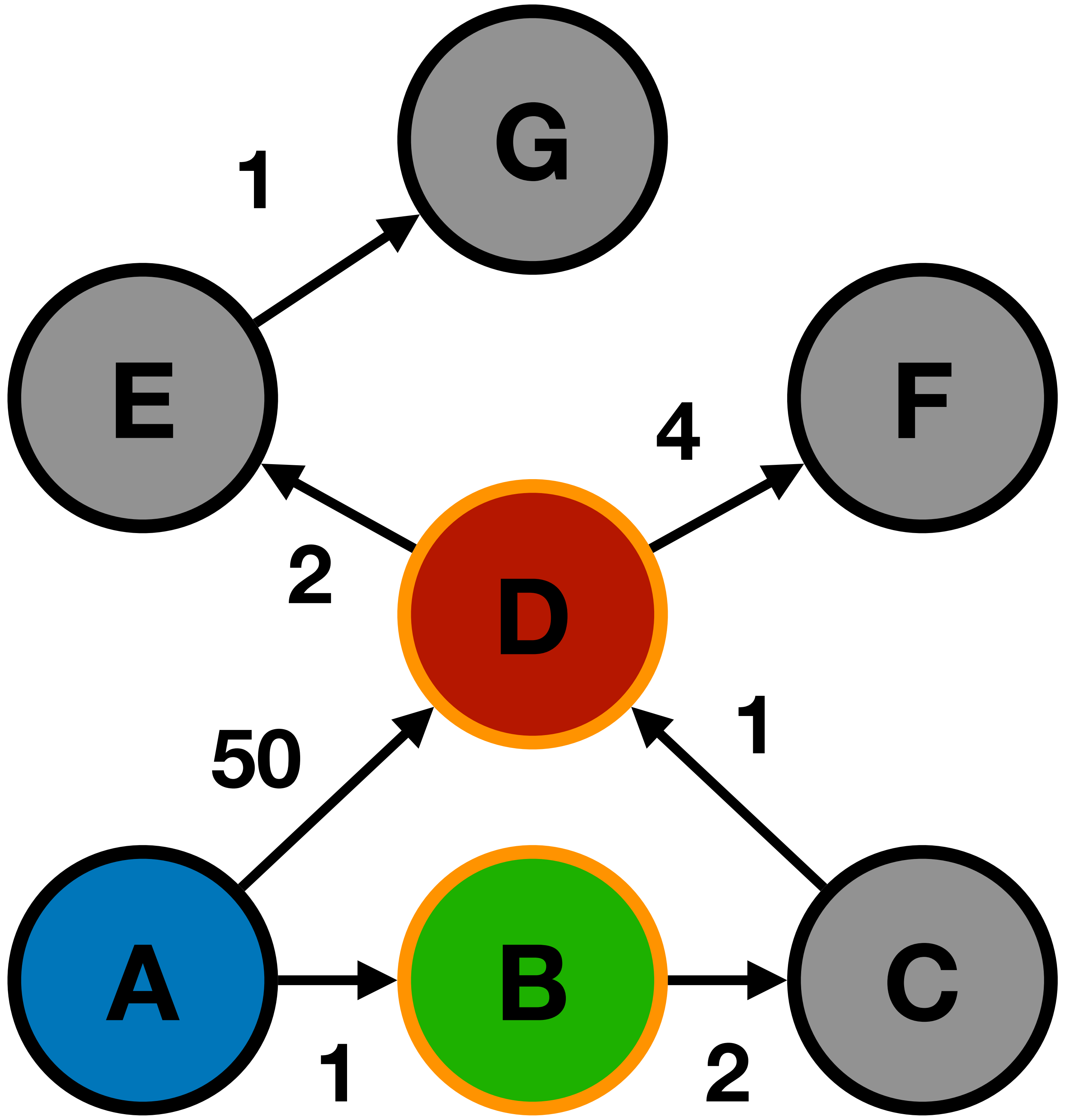
○ Active Vertex

# Rounds: 0

# Updates: 0

A	B	C	D	E	F	G
0	∞	∞	∞	∞	∞	∞

Bellman-Ford (Unordered SSSP)



🔵 Start Vertex

⚫ Unreached Vertex

🟢 Vertex with Shortest Distance

🔴 Vertex with Suboptimal Distance

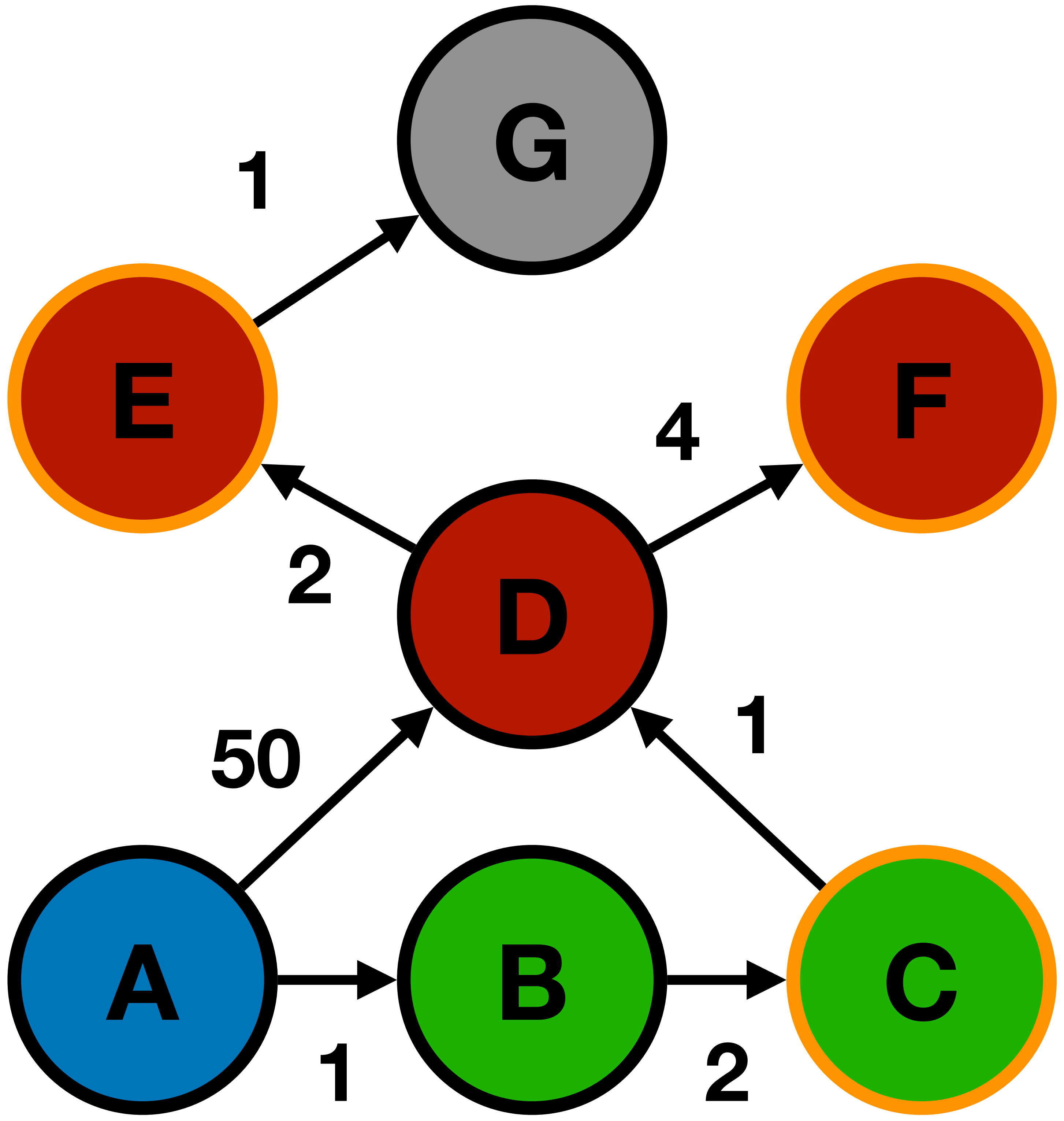
○ Active Vertex

# Rounds: 1

# Updates: 2

A	B	C	D	E	F	G
0	1	∞	50	∞	∞	∞

Bellman-Ford (Unordered SSSP)



🔵 Start Vertex

⚫ Unreached Vertex

🟢 Vertex with Shortest Distance

🔴 Vertex with Suboptimal Distance

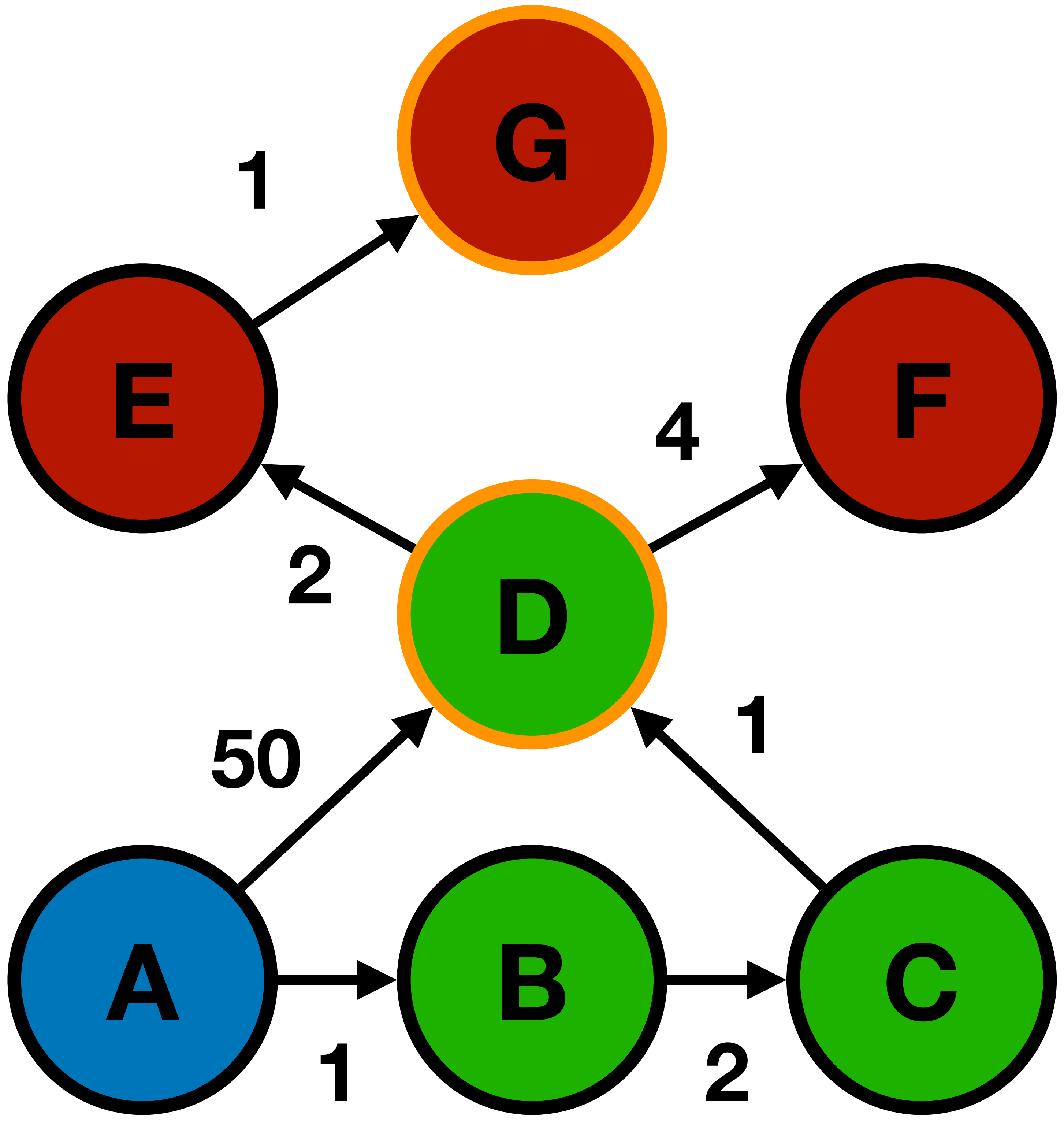
○ Active Vertex

# Rounds: 2

# Updates: 5

A	B	C	D	E	F	G
0	1	3	50	52	54	∞

Bellman-Ford (Unordered SSSP)



🔵 Start Vertex

⚫ Unreached Vertex

🟢 Vertex with Shortest Distance

🔴 Vertex with Suboptimal Distance

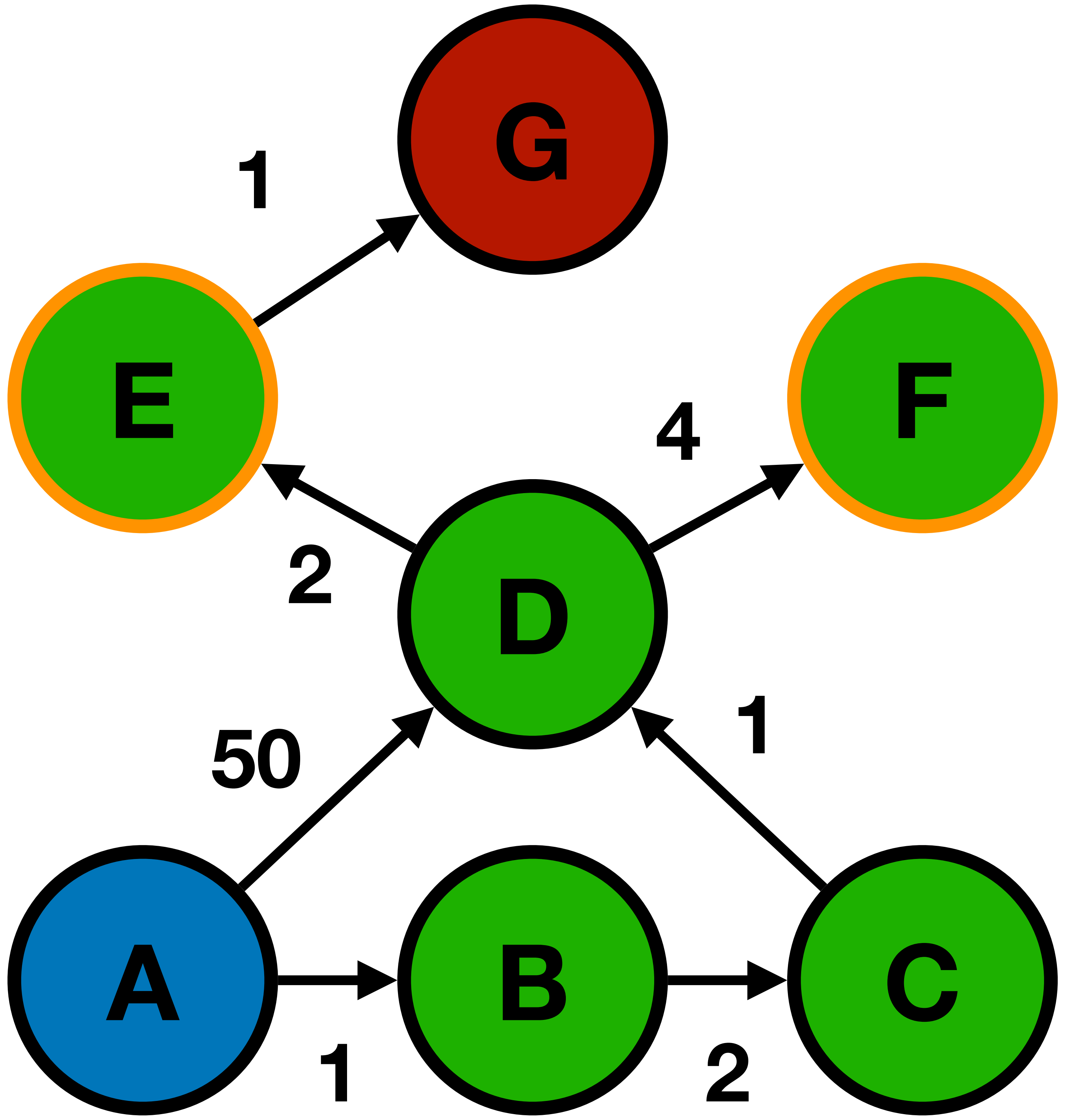
○ Active Vertex

# Rounds: 3

# Updates: 7

A	B	C	D	E	F	G
0	1	3	4	52	54	53

Bellman-Ford (Unordered SSSP)



🔵 Start Vertex

⚫ Unreached Vertex

🟢 Vertex with Shortest Distance

🔴 Vertex with Suboptimal Distance

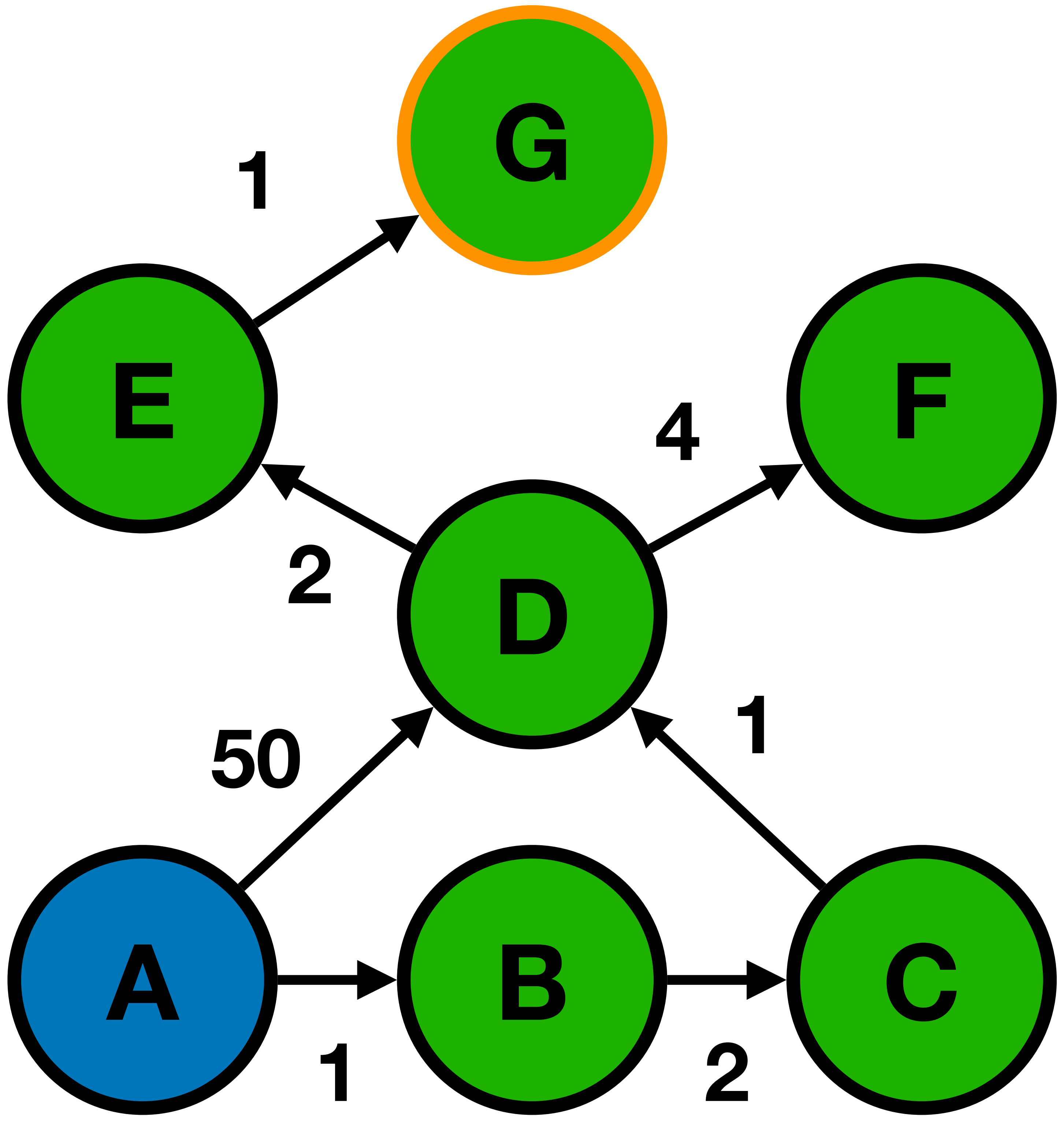
○ Active Vertex

A	B	C	D	E	F	G
0	1	3	4	6	8	53

# Rounds: 4

# Updates: 9

Bellman-Ford (Unordered SSSP)



🔵 Start Vertex

⚫ Unreached Vertex

🟢 Vertex with Shortest Distance

🔴 Vertex with Suboptimal Distance

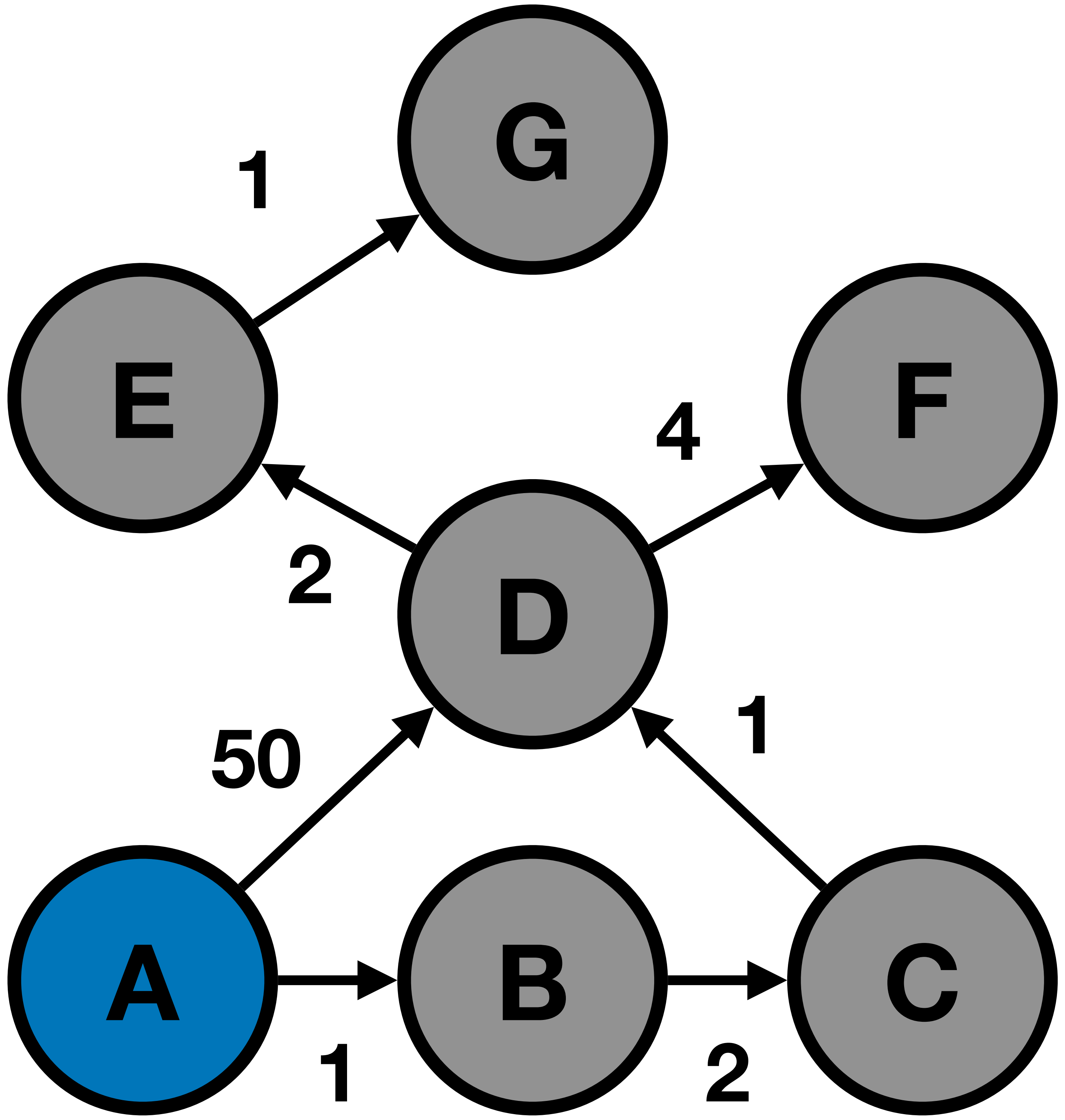
○ Active Vertex

# Rounds: 5

# Updates:10

A	B	C	D	E	F	G
0	1	3	4	6	8	7

Delta-Stepping (Ordered SSSP)



🔵 Start Vertex

⚫ Unreached Vertex

🟢 Vertex with Shortest Distance

🔴 Vertex with Suboptimal Distance

○ Active Vertex

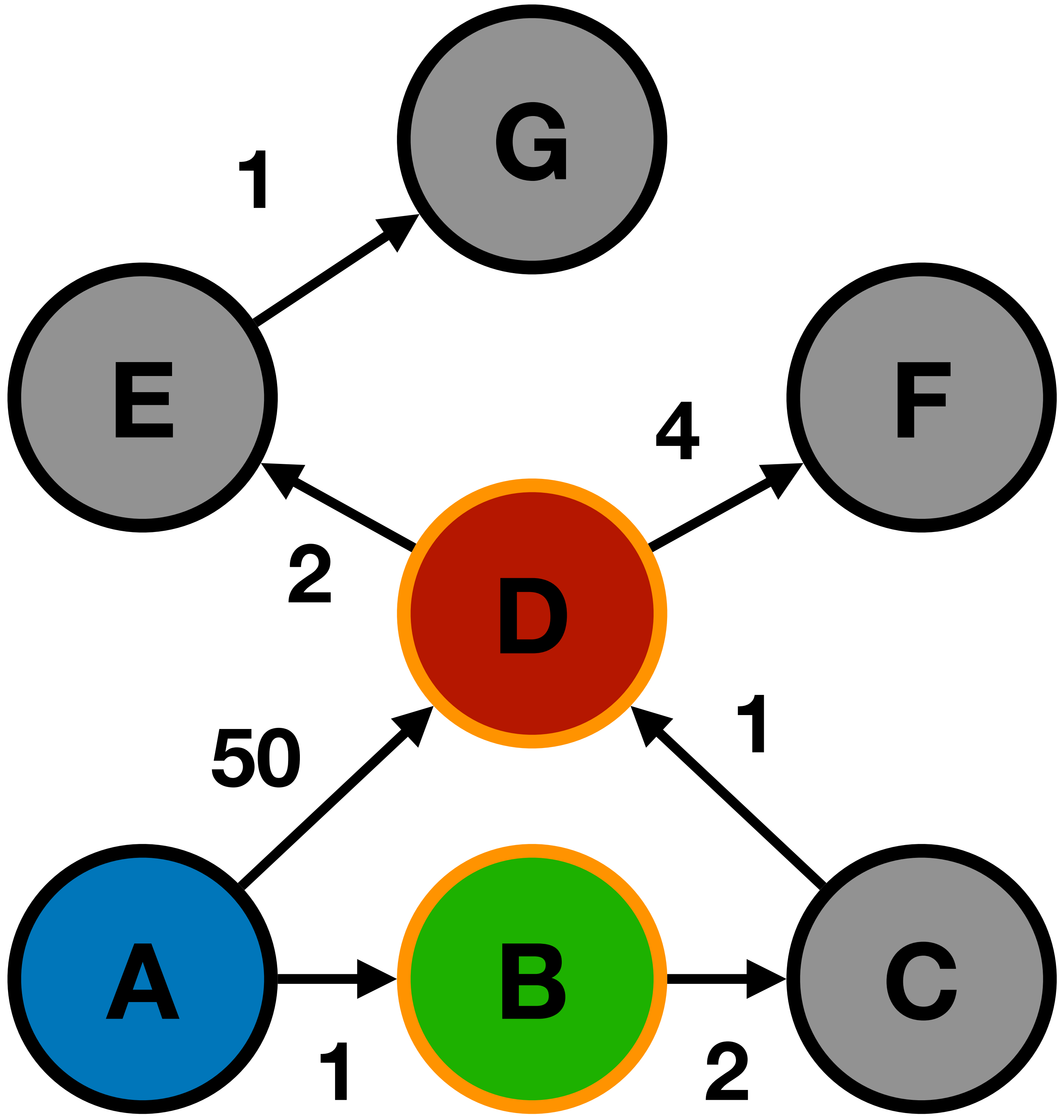
A	B	C	D	E	F	G
0	∞	∞	∞	∞	∞	∞

# Rounds: 0

# Updates: 0

Delta: 10

Delta-Stepping (Ordered SSSP)



🔵 Start Vertex

⚫ Unreached Vertex

🟢 Vertex with Shortest Distance

🔴 Vertex with Suboptimal Distance

⭕ Active Vertex

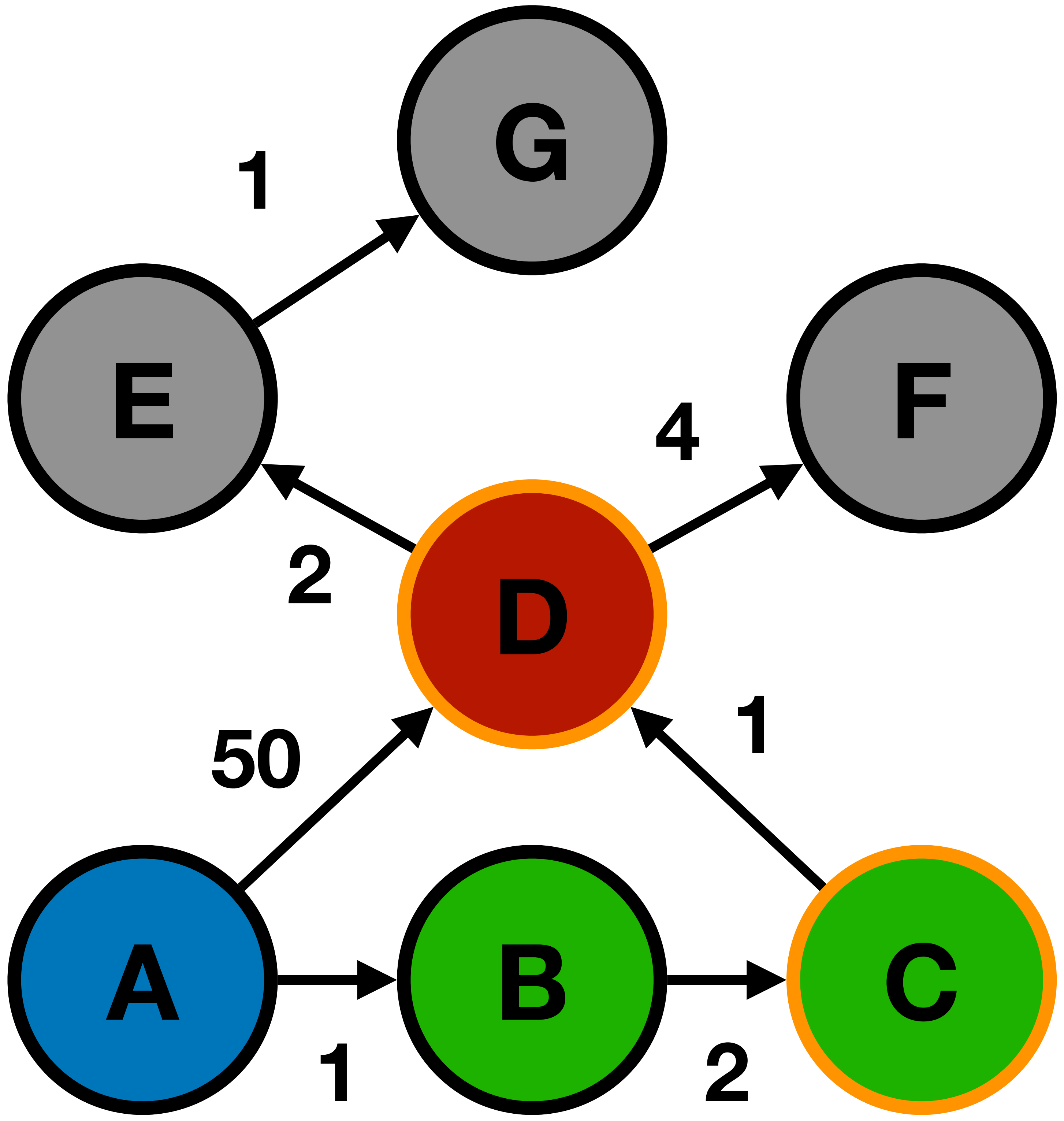
A	B	C	D	E	F	G
0	1	∞	50	∞	∞	∞

# Rounds: 1

# Updates: 2

Delta: 10

Delta-Stepping (Ordered SSSP)



🔵 Start Vertex

⚫ Unreached Vertex

🟢 Vertex with Shortest Distance

🔴 Vertex with Suboptimal Distance

○ Active Vertex

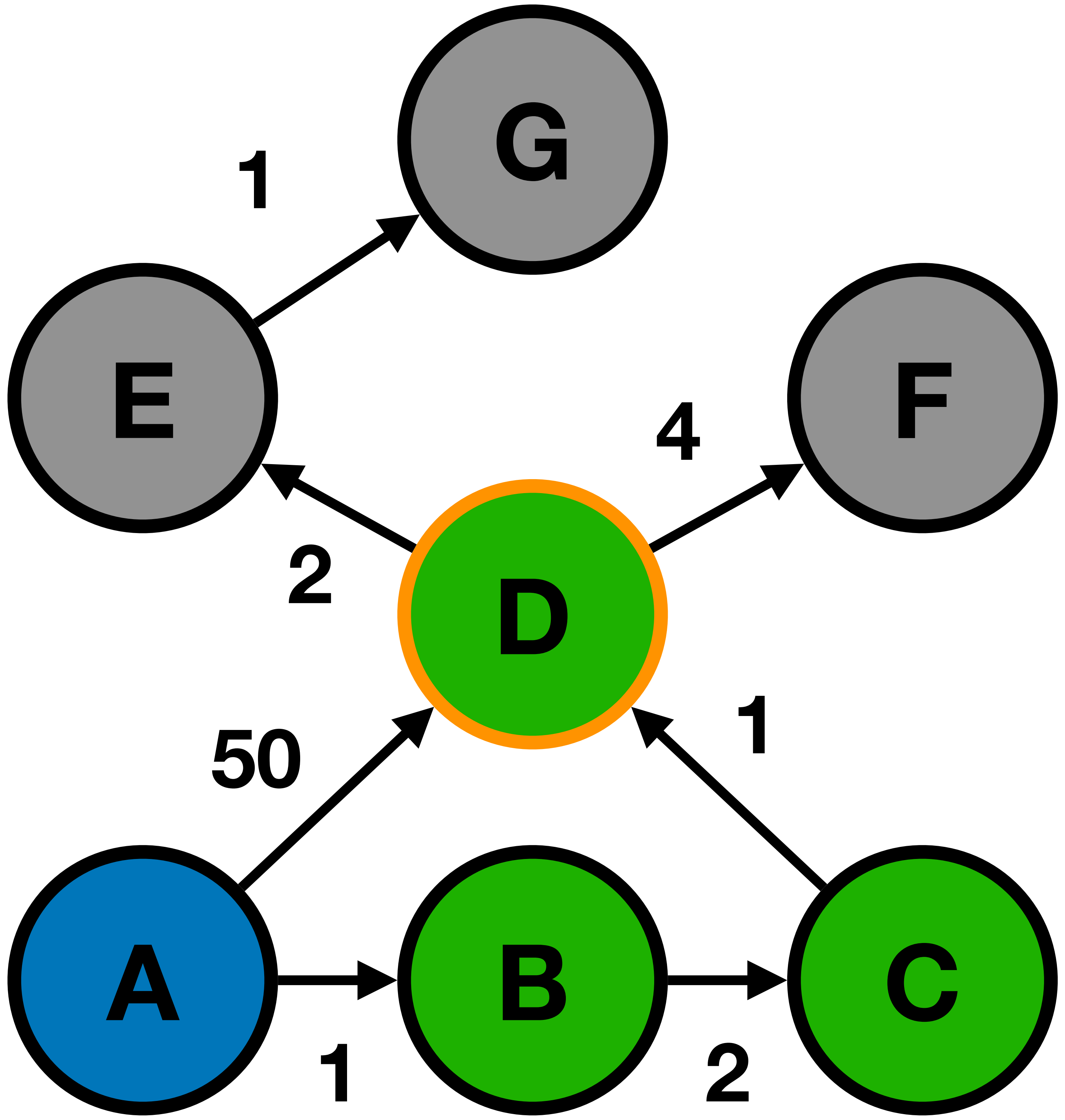
A	B	C	D	E	F	G
0	1	3	50	∞	∞	∞

# Rounds: 2

# Updates: 3

Delta: 10

Delta-Stepping (Ordered SSSP)



🔵 Start Vertex

⚫ Unreached Vertex

🟢 Vertex with Shortest Distance

🔴 Vertex with Suboptimal Distance

○ Active Vertex

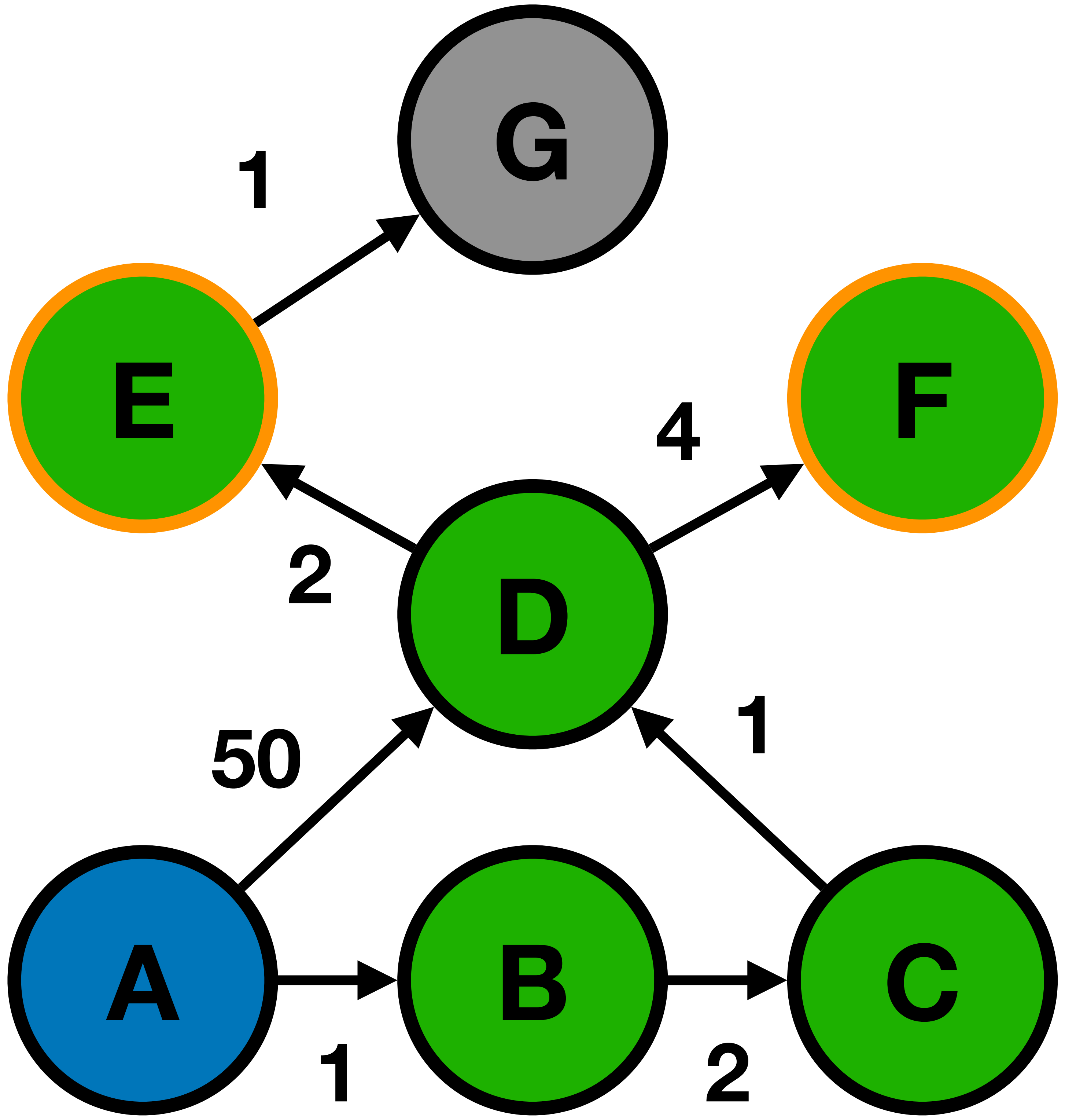
A	B	C	D	E	F	G
0	1	3	4	∞	∞	∞

# Rounds: 3

# Updates: 4

Delta: 10

Delta-Stepping (Ordered SSSP)



🔵 Start Vertex

● Unreached Vertex

🟢 Vertex with Shortest Distance

🔴 Vertex with Suboptimal Distance

○ Active Vertex

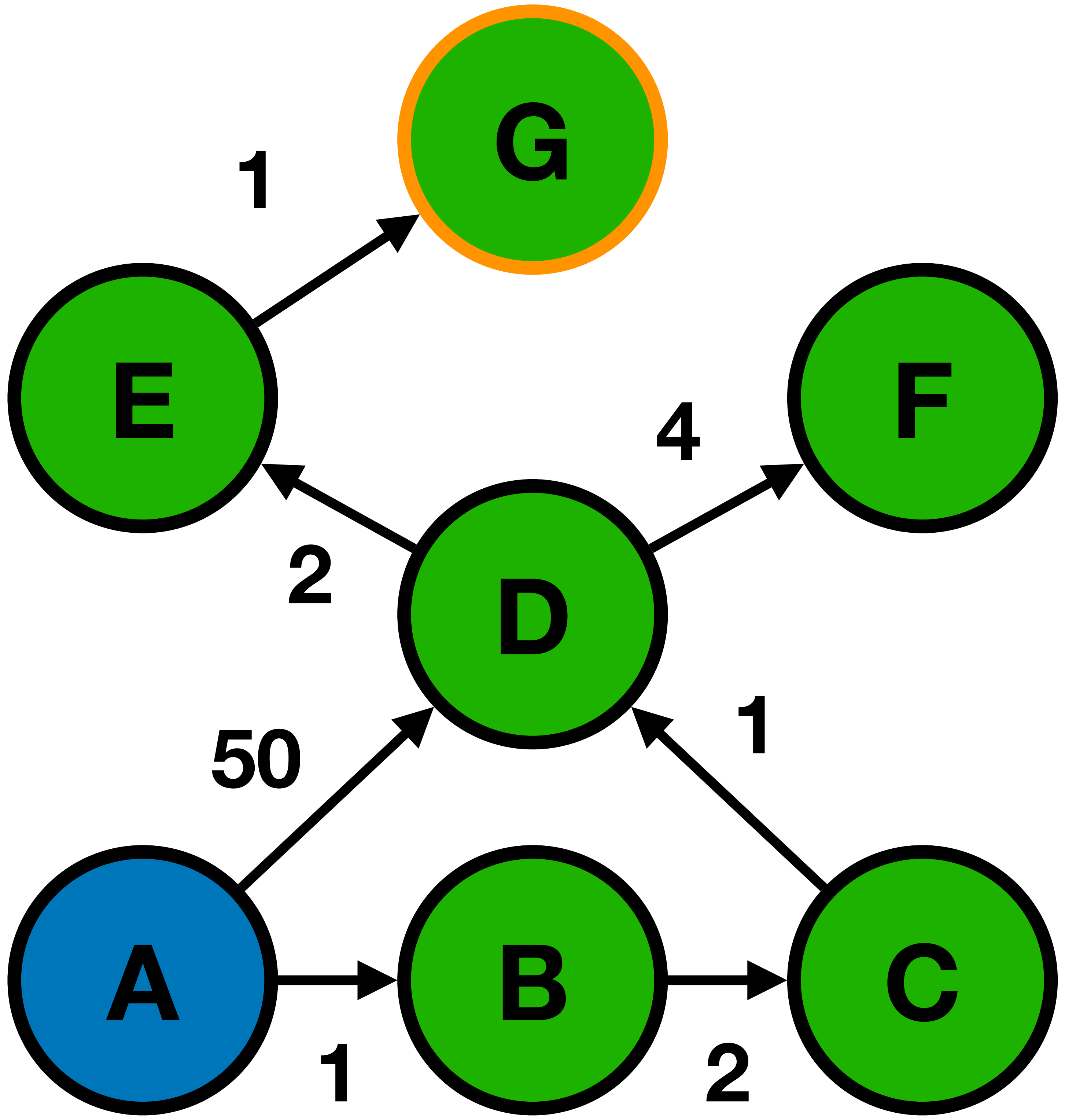
A	B	C	D	E	F	G
0	1	3	4	6	8	∞

# Rounds: 4

# Updates: 6

Delta: 10

Delta-Stepping (Ordered SSSP)



🔵 Start Vertex

● Unreached Vertex

🟢 Vertex with Shortest Distance

🔴 Vertex with Suboptimal Distance

○ Active Vertex

A	B	C	D	E	F	G
0	1	3	4	6	8	7

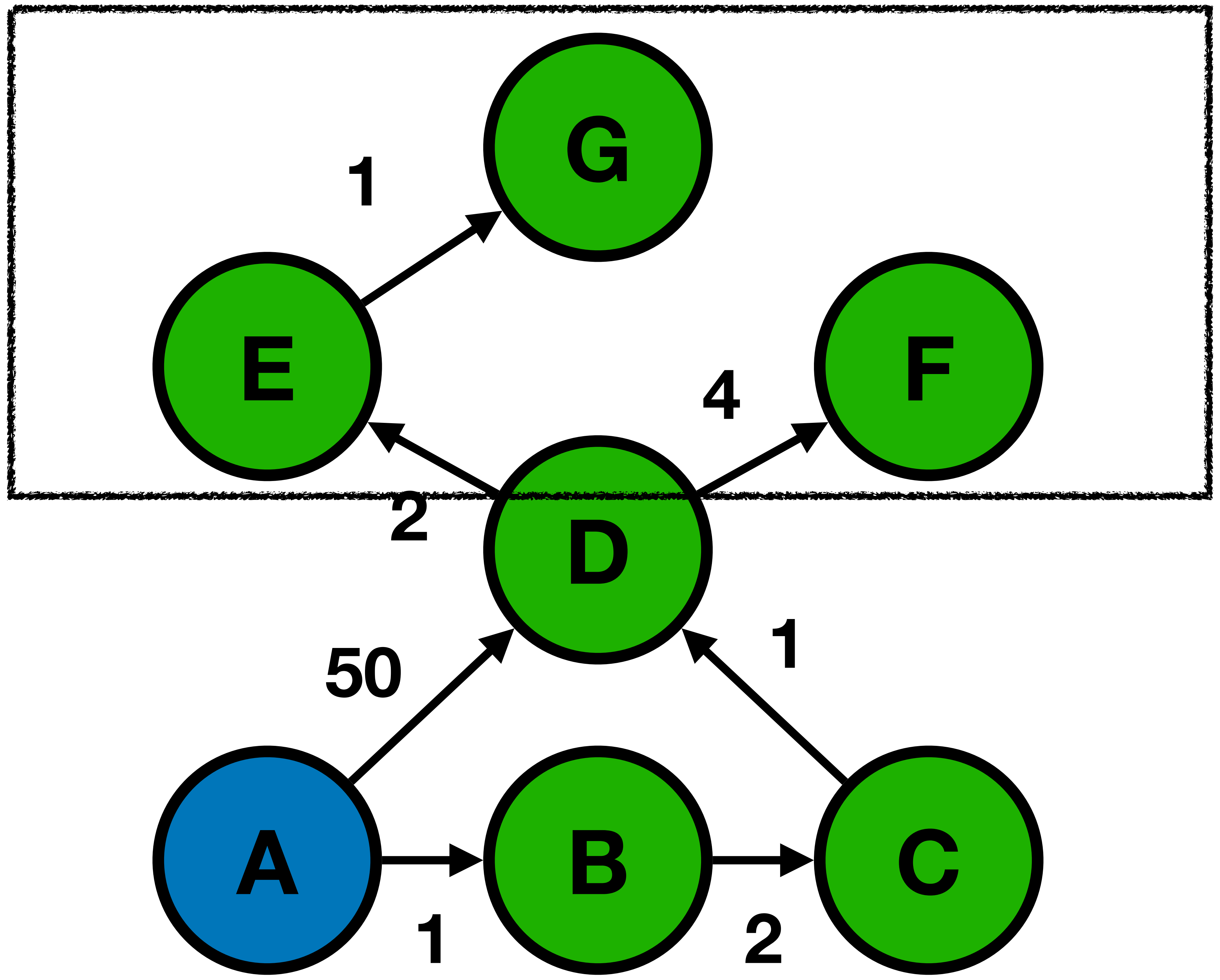
# Rounds: 5

# Updates: 7

Delta: 10

Delta-Stepping (Ordered SSSP)

No Redundant Updates for E, F, and G



A	B	C	D	E	F	G
0	1	3	4	6	8	7

Rounds: 5

Updates:10

Rounds: 5

Updates: 7

Delta: 10

🔵 Start Vertex

⚫ Unreached Vertex

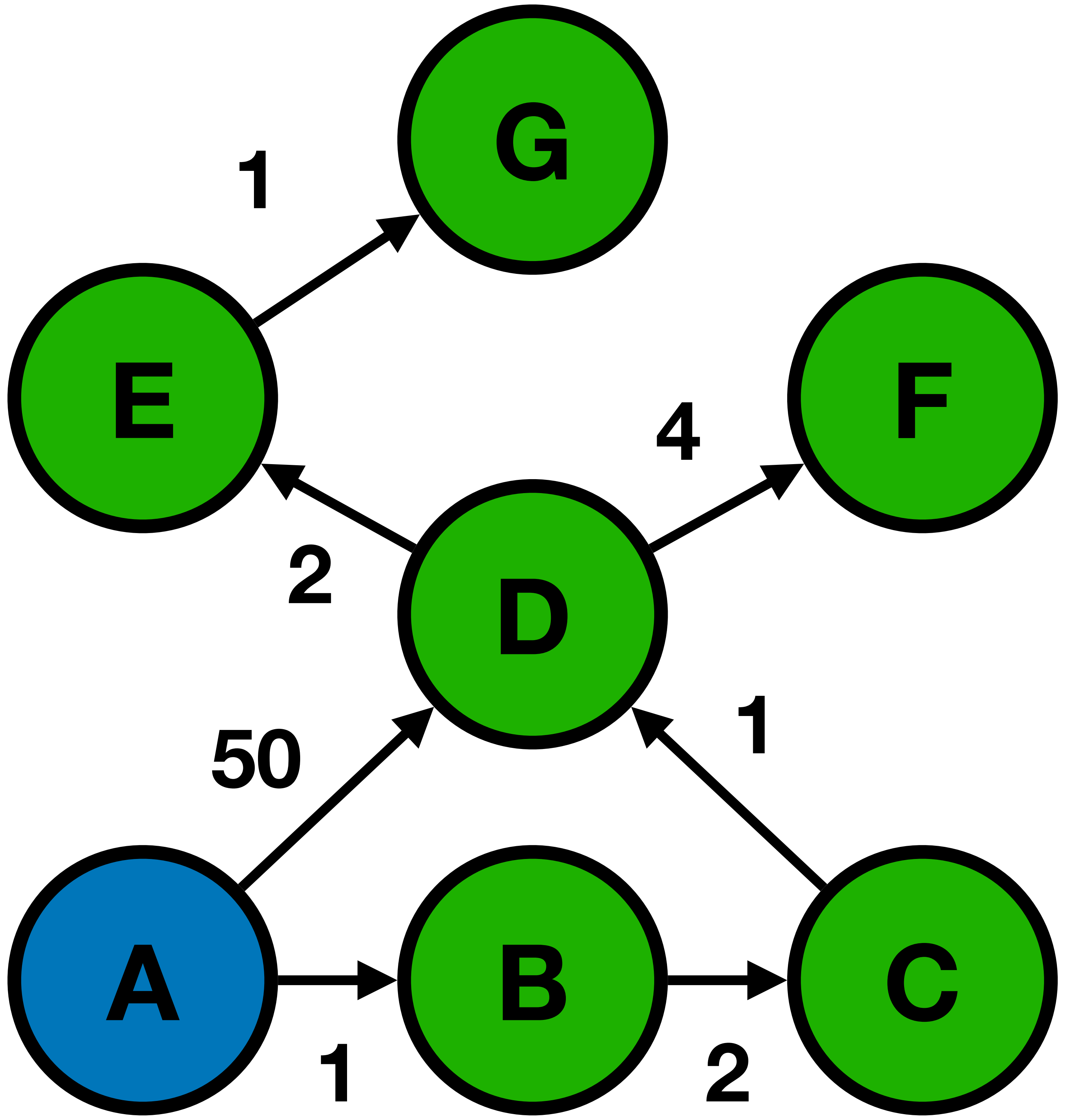
🟢 Vertex with Shortest Distance

🔴 Vertex with Suboptimal Distance

○ Active Vertex

Delta-Stepping (Ordered SSSP)

Algorithmic Tradeoff Between Parallelism and Work-Efficiency



🔵 **Start Vertex**

⚫ **Unreached Vertex**

🟢 **Vertex with Shortest Distance**

🔴 **Vertex with Suboptimal Distance**

○ **Active Vertex**

A	B	C	D	E	F	G
0	1	3	4	6	8	7

# Rounds: 5

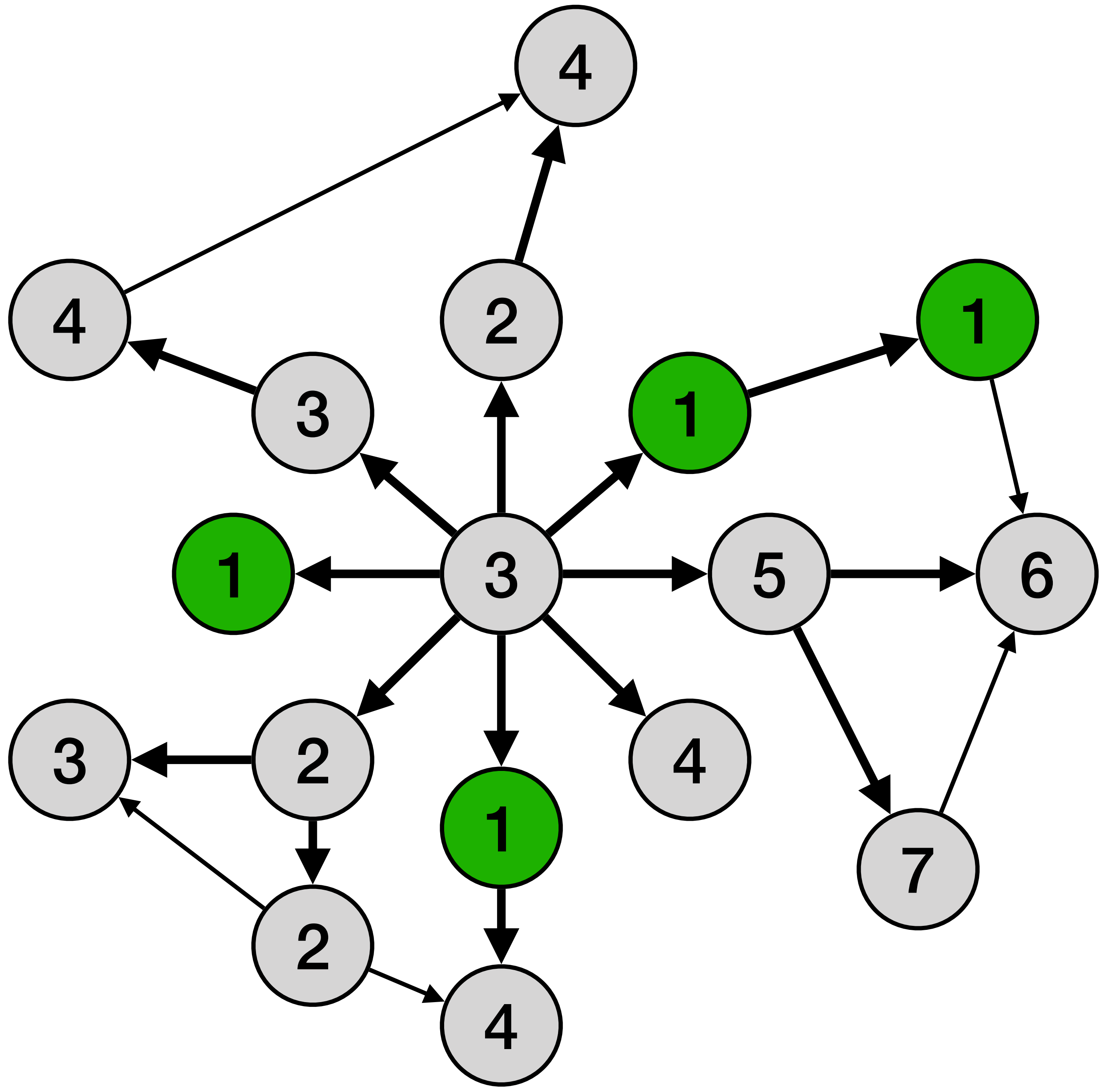
# Updates: 7

Delta: 10

Priority-based Extensions to GraphIt

- **Decouple Algorithm from Optimization for Ordered Graph Algorithms**
    - Priority-based Algorithm Language Operators
    - Optimizations for Ordered Parallelism
- **Language and Compiler Extensions Achieve**
    - Ease-of-Use
    - Consistent High-Performance

Priority-Based Extensions



- PriorityQueue
  - dequeueReadySet()
  - getCurrentPriority()
  - finished(), finishedNode()
  - updatePriorityMin, updatePrioritySum, ..

Delta-Stepping

```
const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
   pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end



func main ( )
  var start_vertex : int = 0;
  SP[start_vertex] = 0;
  pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
  while (!pq.finished())
      var frontier: vertexset{Vertex} = pq.dequeueReadySet();
      #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
      delete frontier;
  end
end
```

Delta-Stepping

```
 const pq: priority_queue{Vertex}(int);

 func updateEdge(src : Vertex, dst : Vertex, weight : int)
    pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
 end



func main ( )
  var start_vertex : int = 0;
  SP[start_vertex] = 0;
  pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
  while (!pq.finished())
      var frontier: vertexset{Vertex} = pq.dequeueReadySet();
      #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
      delete frontier;
 end
end
```

Delta-Stepping

```
 const pq: priority_queue{Vertex}(int);

 func updateEdge(src : Vertex, dst : Vertex, weight : int)
     pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
 end


func main ( )
   var start_vertex : int = 0;
   SP[start_vertex] = 0;
   pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
   while (!pq.finished())
        var frontier: vertexset{Vertex} = pq.dequeueReadySet();
        #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
        delete frontier;
  end
end
```

Delta-Stepping

```
const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
    pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end



func main ( )
   var start_vertex : int = 0;
   SP[start_vertex] = 0;
   pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
   while (!pq.finished())
        var frontier: vertexset{Vertex} = pq.dequeueReadySet();
        #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
        delete frontier;
  end
end
```

Delta-Stepping

Hides Physical Implementation for PriorityQueue

```
const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
    pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end


func main ( )
   var start_vertex : int = 0;
   SP[start_vertex] = 0;
   pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
   while (!pq.finished())
        var frontier: vertexset{Vertex} = pq.dequeueReadySet();
        #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
        delete frontier;
   end
end
```

Delta-Stepping

```
const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
    pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end



func main ( )
   var start_vertex : int = 0;
   SP[start_vertex] = 0;
   pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
   while (!pq.finished())
        var frontier: vertexset{Vertex} = pq.dequeueReadySet();
        #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
        delete frontier;
  end
end
```

Delta-Stepping

```
 const pq: priority_queue{Vertex}(int);

 func updateEdge(src : Vertex, dst : Vertex, weight : int)
    pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
 end



func main ( )
  var start_vertex : int = 0;
  SP[start_vertex] = 0;
  pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
  while (!pq.finished())
      var frontier: vertexset{Vertex} = pq.dequeueReadySet();
      #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
      delete frontier;
  end
end
```

Delta-Stepping

```
 const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
   pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end


func main ( )
  var start_vertex : int = 0;
  SP[start_vertex] = 0;
  pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
  while (!pq.finished())
      var frontier: vertexset{Vertex} = pq.dequeueReadySet();
      #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
      delete frontier;
 end
end
```

Delta-Stepping

```
 const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
   pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end



func main ( )
  var start_vertex : int = 0;
  SP[start_vertex] = 0;
  pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
  while (!pq.finished())
      var frontier: vertexset{Vertex} = pq.dequeueReadySet();
      #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
      delete frontier;
 end
end
```

Schedule:

```
program
->configApplyPriorityUpdate(“s1”, “lazy”)
->configApplyPriorityUpdateDelta(“s1”, 4)
->configApplyDirection(“s1“, SparsePush”)
->configApplyParallelization(“s1”, “dynamic-
vertex-parallel”)
```

Delta-Stepping

```
const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
   pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end


func main ( )
   var start_vertex : int = 0;
   SP[start_vertex] = 0;
   pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
   while (!pq.finished())
      var frontier: vertexset{Vertex} = pq.dequeueReadySet();
      #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
      delete frontier;
   end
end
```

Schedule:

```
program
->configApplyPriorityUpdate(“s1”, “lazy”)
->configApplyPriorityUpdateDelta(“s1”, 4)
->configApplyDirection(“s1“, SparsePush”)
->configApplyParallelization(“s1”, “dynamic-
vertex-parallel”)
```

Delta-Stepping

```
const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
    pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end


func main ( )
   var start_vertex : int = 0;
   SP[start_vertex] = 0;
   pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
   while (!pq.finished())
        var frontier: vertexset{Vertex} = pq.dequeueReadySet();
        #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
        delete frontier;
   end
end
```

Schedule:

```
program
->configApplyPriorityUpdate(“s1”, “lazy”)
->configApplyPriorityUpdateDelta(“s1”, 4)
->configApplyDirection(“s1“, SparsePush”)
->configApplyParallelization(“s1”, “dynamic-vertex-parallel”)
```

→

```
int * dist = new int[num_verts];
LazyPriorityQueue* pq;
int delta = 4;
WGraph* G = loadGraph(argv[1]);

//simplified snippets of the generated main function
…
dist[start_vertex] = 0;
pq = new LazyPriorityQueue(true, “lower”, dist, delta);
While (pq.finished()){
  VertexSubset *  frontier = getNextBucket(pq);
  uint* outEdges = setupOutputBuffer(g, frontier);
  uint* offsets = setupOutputBufferOffsets(g, frontier);
  parallel_for (uint s : frontier.vert_array) {
    int j = 0;
    uint offset = offsets[i];
    for(WNode d : G.getOutNgh(s)){
      bool tracking_var = false;
      int new_dist = dist[s.v] + d.weight;
      tracking_var = atomicWriteMin(&dist[d.v], new_dist);
      If (tracking_var && CAS(dedup_flags[d.v],0,1)){
         outEdges[offset + j] = d.v;
      } else { outEdges[offset + j] = UINT_MAX; }
      j++;
  }}
  VertexSubset* nextFrontier = setupFrontier(outEdges);
  updateBuckets(nextFrontier, pq, delta);
  …
}
…
```

Delta-Stepping

```
 const pq: priority_queue{Vertex}(int);

func updateEdge(src : Vertex, dst : Vertex, weight : int)
    pq.updatePriorityMin(dst, SP[dst], SP[src] + weight);
end


func main ( )
   var start_vertex : int = 0;
   SP[start_vertex] = 0;
   pq = new priority_queue{Vertex}(int)(true, "lower_first", SP, start_vertex);
   while (!pq.finished())
        var frontier: vertexset{Vertex} = pq.dequeueReadySet();
        #s1# edges.from(frontier).applyUpdatePriority(updateEdge);
        delete frontier;
   end
end
```

Schedule:

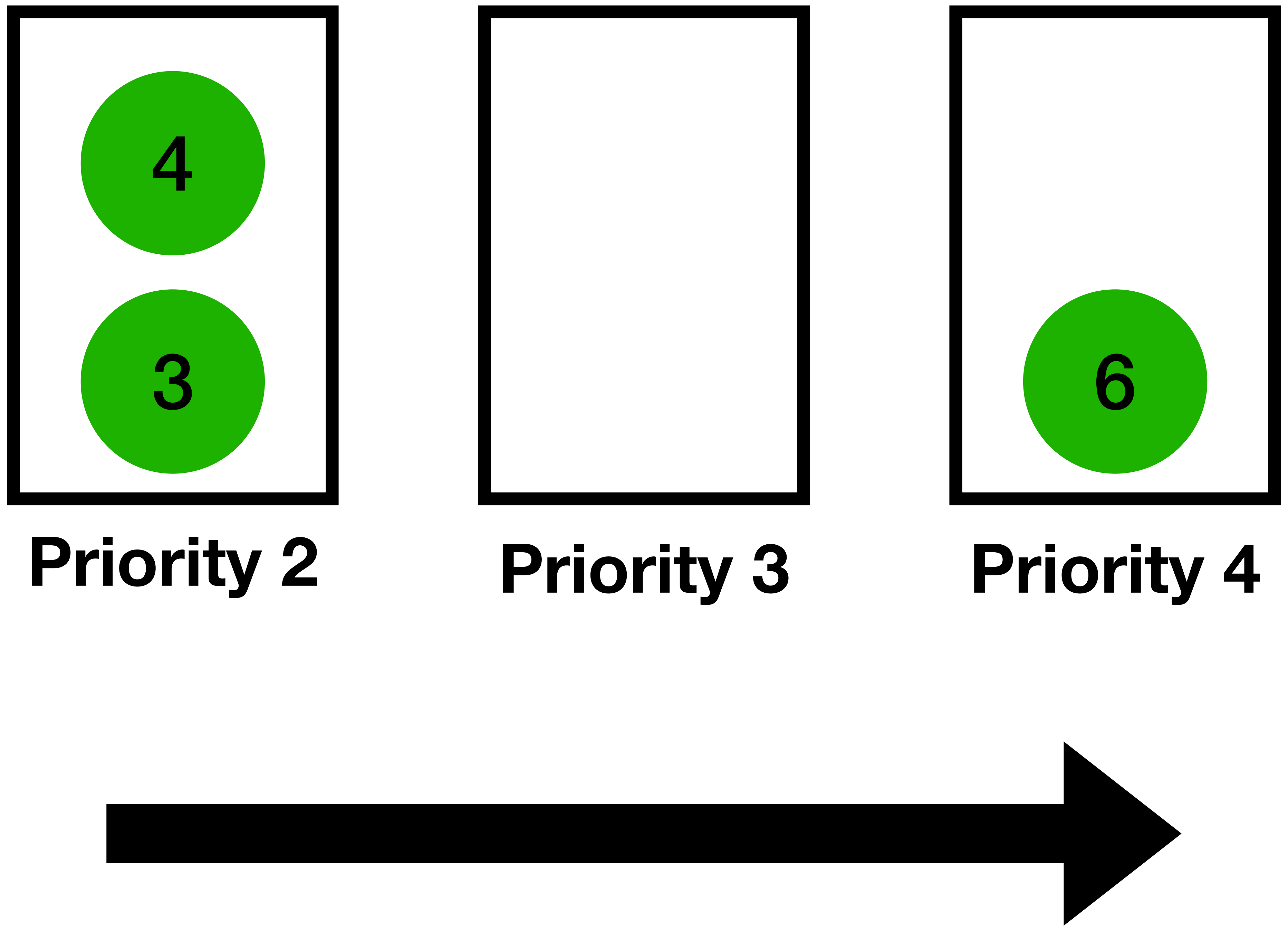
```
program
->configApplyPriorityUpdate(“s1”, “eager”)
->configApplyPriorityUpdateDelta(“s1”, 4)
->configApplyDirection(“s1“, SparsePush”)
->configApplyParallelization(“s1”, “dynamic-vertex-parallel”)
```

→

```
int * dist = new int[num_verts];
EagerPriorityQueue* pq;
int delta = 4;
WGraph* G = loadGraph(argv[1]);

//simplified snippets of the generated main function
…
dist[start_vertex] = 0;
frontier[0] = start_vertex;
pq = new EagerPriorityQueue(true, “lower”, dist, delta);
uint* frontier =  new uint[G.num_edges()];
#pragma omp parallel
{   vector<vector<uint> > local_bins(0);
    while (pq.finished()) {
      #pragma omp for nowait schedule(dynamic, 64)
      for (size_t i = 0; i < frontier.size(); i++) {
        uint s = frontier[i];
        for (WNode d : G.getOutNgh(s)) {
          int new_dist = dist[s] + d.weight;
          bool changed = atomicWriteMin(&dist[d.v],new_dist);
          if (changed == false) {break;}}
          if (changed) {
            size_t dest_bin = new_dist/delta;
            if (dest_bin >= local_bins.size()) {
              local_bins.resize(dest_bin+1);}
            local_bins[dest_bin].push_back(d.v);
          }}}// end of for frontier for loop
    … //omitted:find next bucket
   #pragma omp barrier
    … //omitted:copy local buckets to global bucket
   #pragma omp barrier } // end of parallel region
…
```

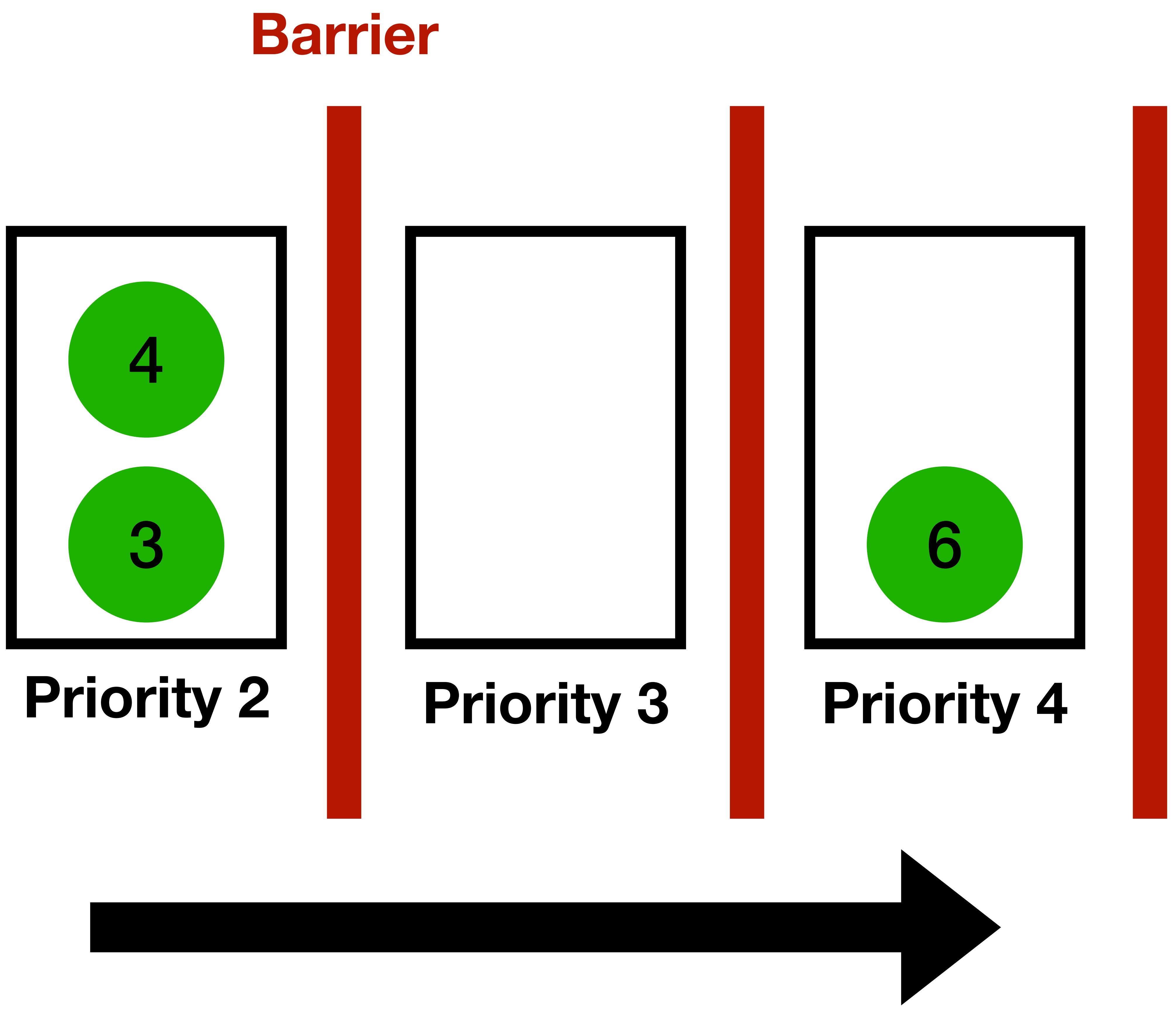
PriorityQueue with Bucketing

Vertices are stored in buckets according to their priority, and are processed in order



PriorityQueue with Bucketing

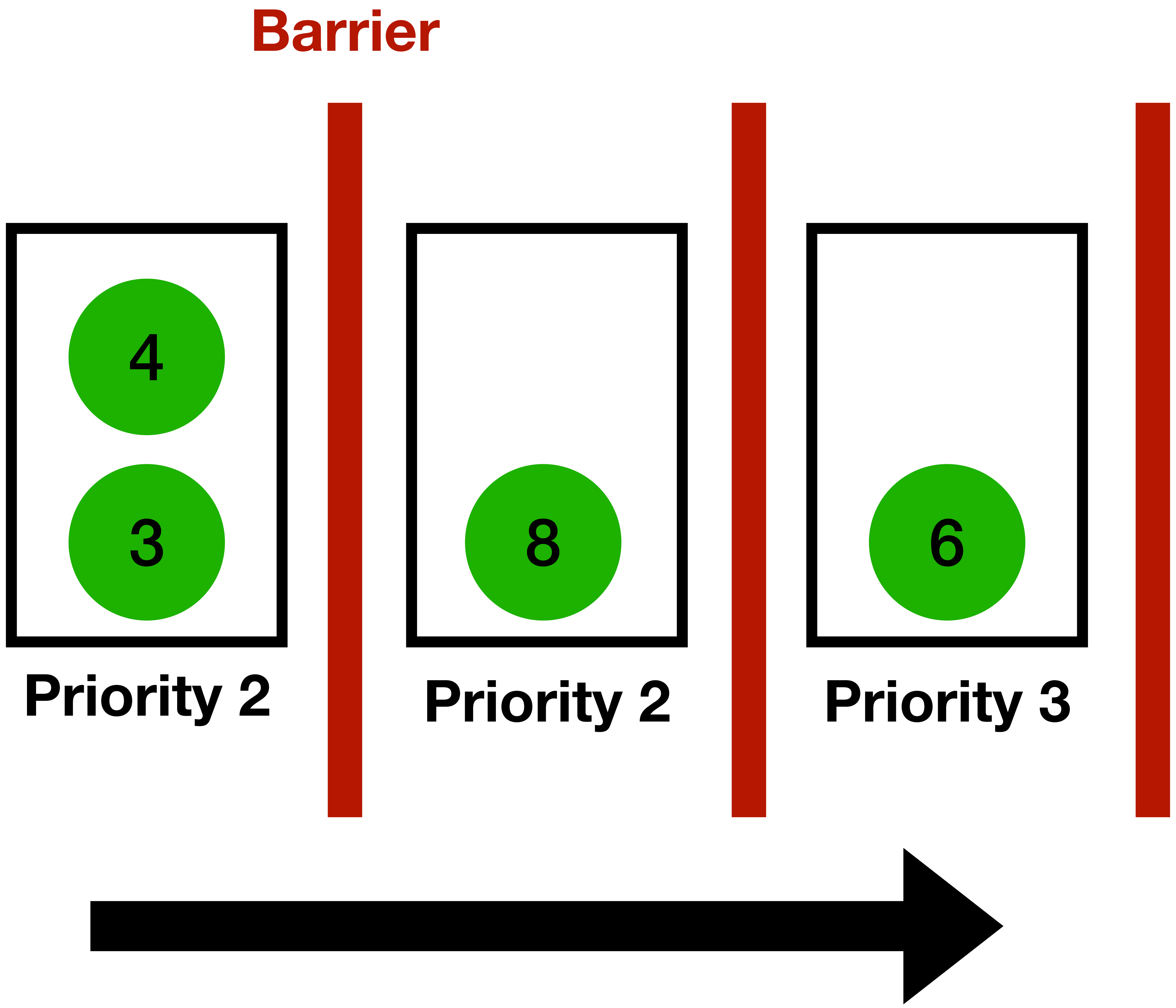
Global synchronization after each bucket (barrier)



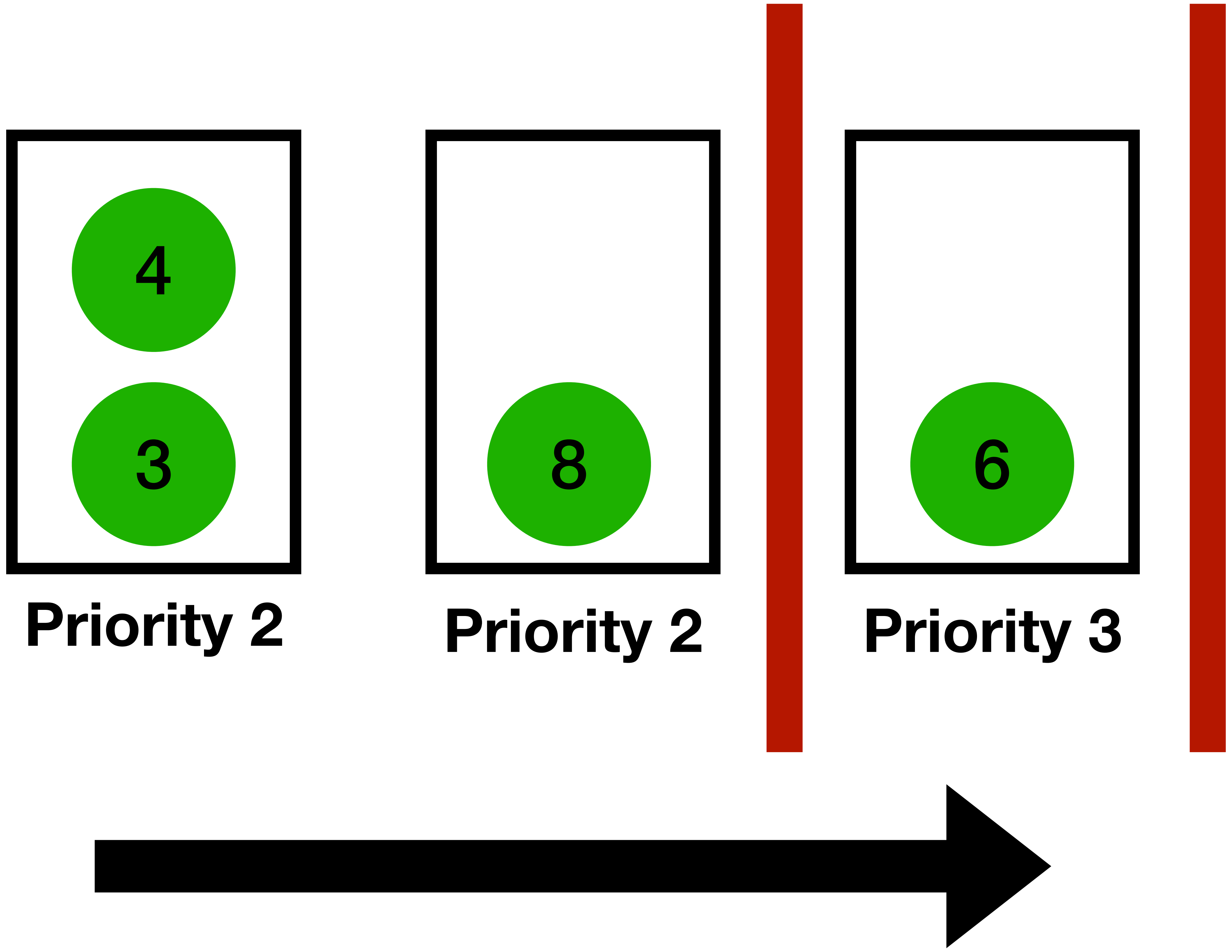
Eager vs Lazy

- Eager
  - Update Buckets Immediately after Priority Changes
  - Faster when Synchronization is the Bottleneck
- Lazy
  - Buffer and Reduce Priority Changes before Bucket Updates
  - Faster when Redundant Updates are the Bottleneck

Bucket Fusion



Bucket Fusion



Scheduling Space Extensions

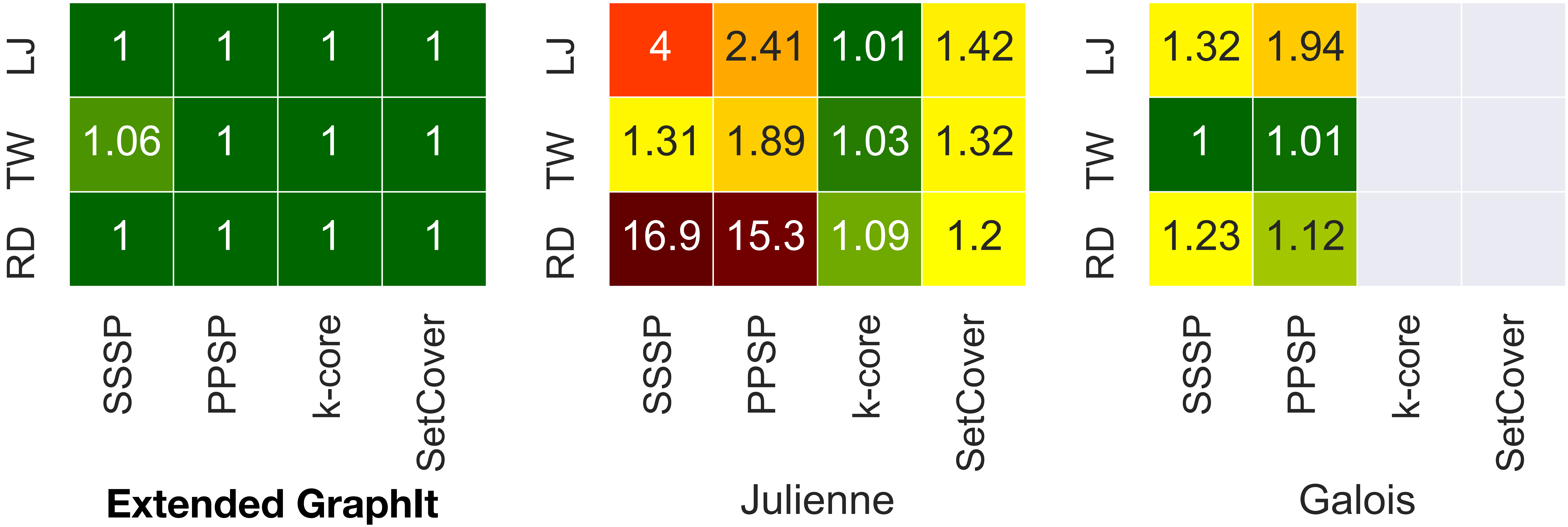
- **configApplyPriorityUpdate**
  - lazy, lazy_const_sum, eager_with_fusion, eager_no_fusion
- **configApplyPriorityUpdateDelta**
  - delta parameter for priority coarsening
- **configBucketFusionThreshold**
- **configNumBuckets** (number of materialized buckets)

Scheduling Space Extensions

- **configApplyPriorityUpdate**
  - lazy, lazy_const_sum, eager_with_fusion, eager_no_fusion
- **configApplyPriorityUpdateDelta**
  - delta parameter for priority coarsening
- **configBucketFusionThreshold**
- **configNumBuckets** (number of materialized buckets)

Compatible with existing GraphIt schedules

Comparisons with Ordered Frameworks



Extended GraphIt

	SSSP	PPSP	k-core	SetCover
LJ	1	1	1	1
TW	1.06	1	1	1
RD	1	1	1	1

Julienne

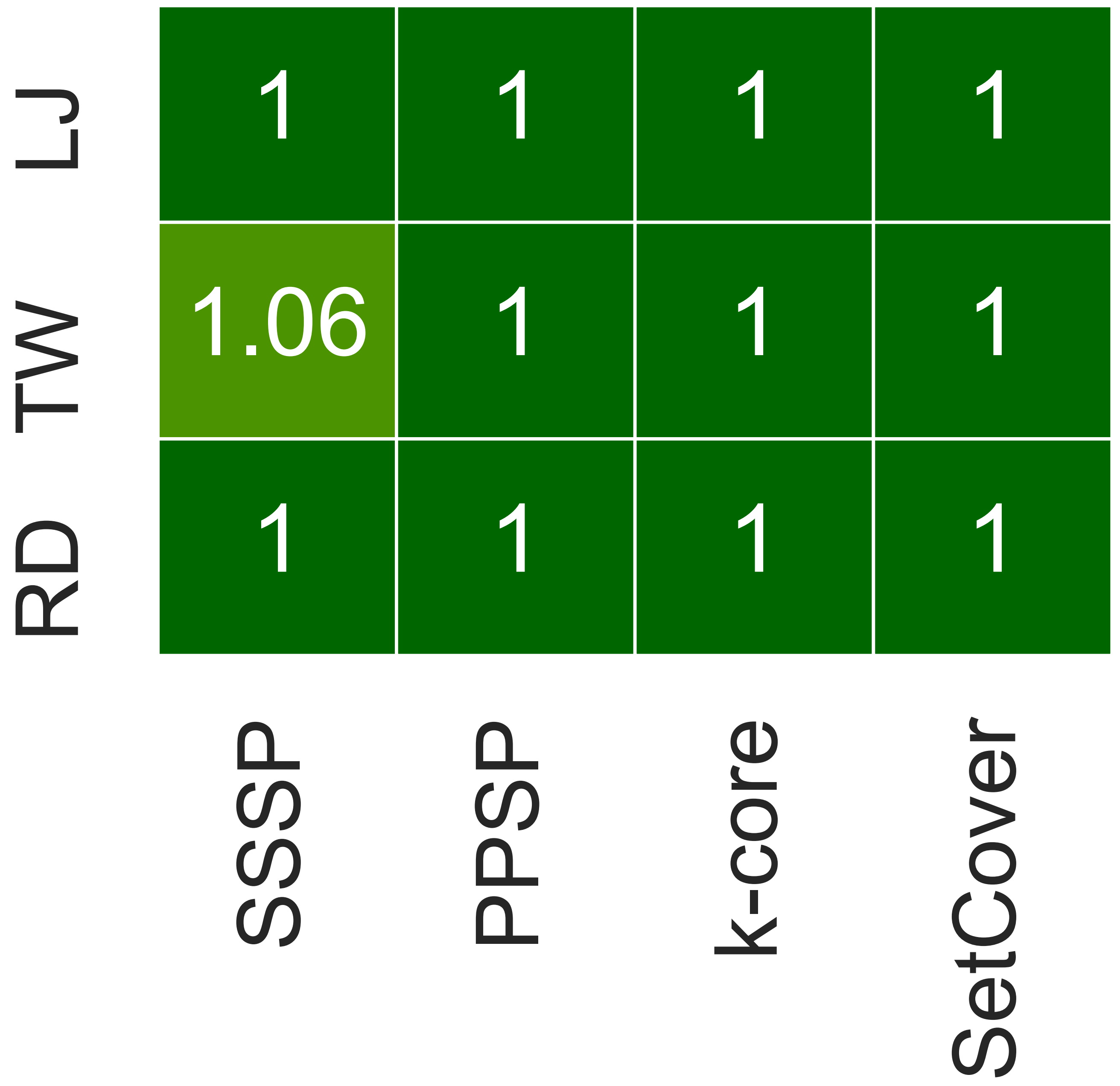
	SSSP	PPSP	k-core	SetCover
LJ	4	2.41	1.01	1.42
TW	1.31	1.89	1.03	1.32
RD	16.9	15.3	1.09	1.2

Galois

	SSSP	PPSP	k-core	SetCover
LJ	1.32	1.94		
TW	1	1.01		
RD	1.23	1.12		

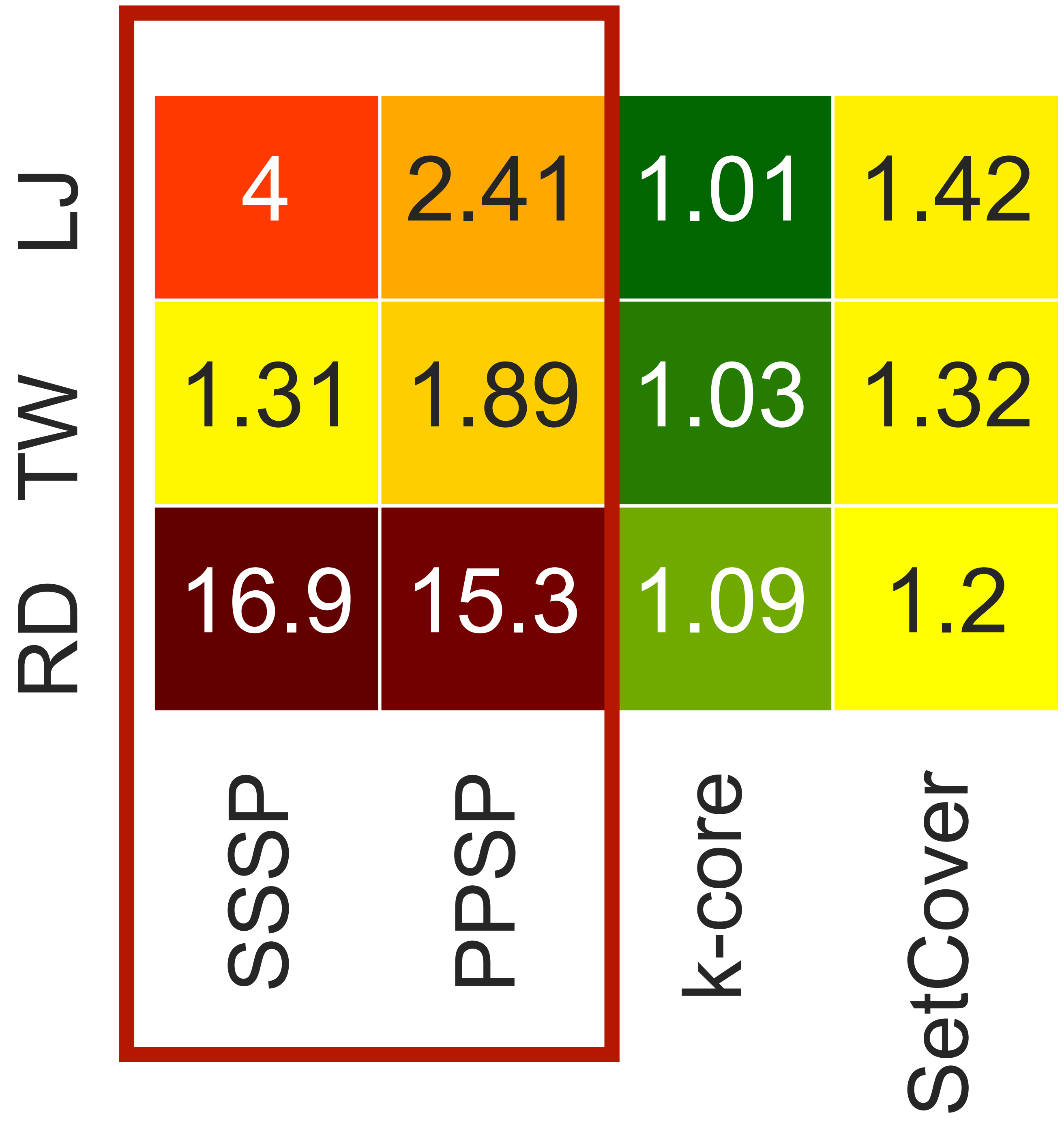
Results for more graphs and algorithms are in the paper (AStar Search, weighted BFS)

Comparisons with Ordered Frameworks



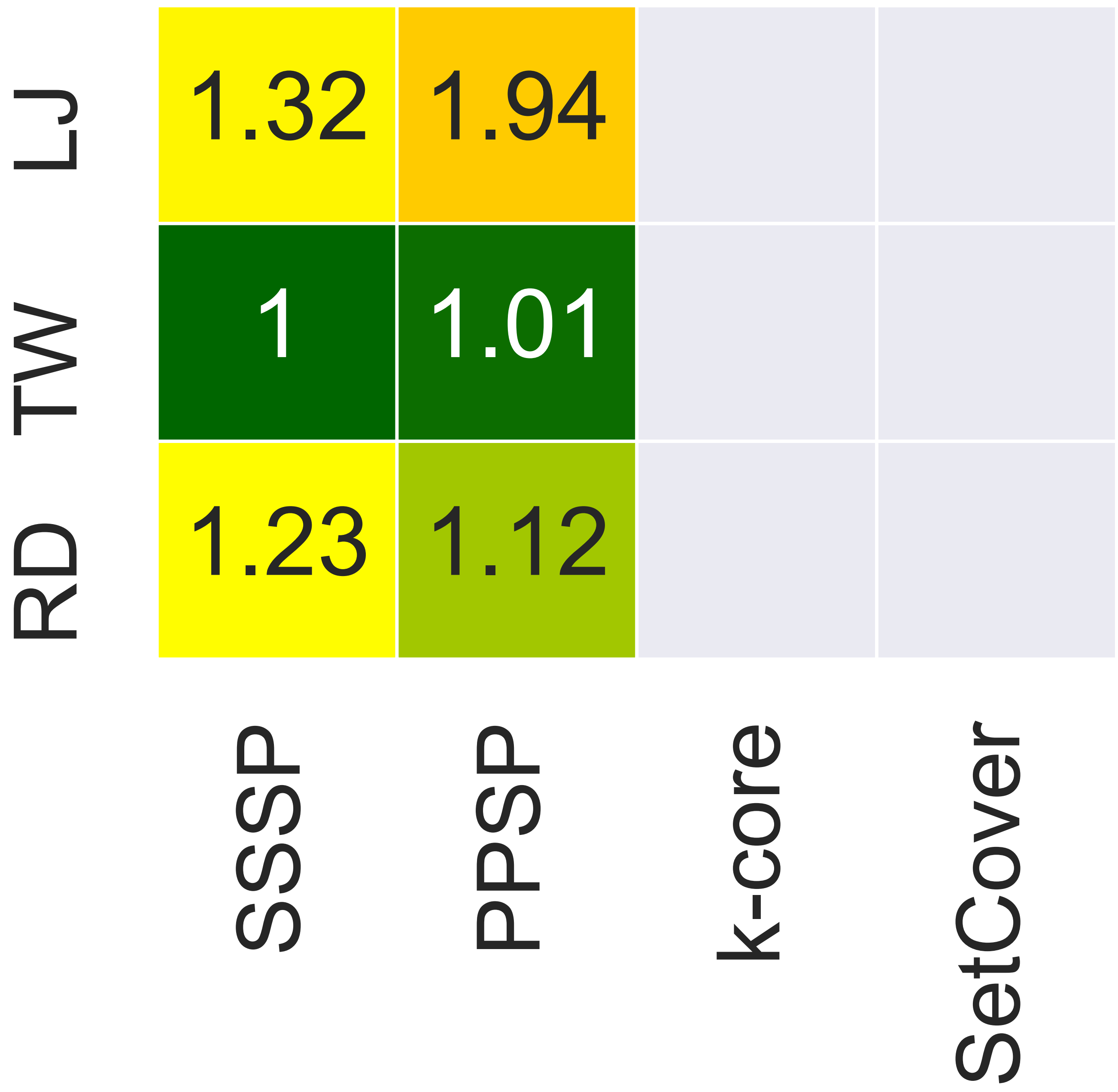
	SSSP	PPSP	k-core	SetCover
LJ	1	1	1	1
TW	1.06	1	1	1
RD	1	1	1	1

Extended GraphIt



	SSSP	PPSP	k-core	SetCover
LJ	4	2.41	1.01	1.42
TW	1.31	1.89	1.03	1.32
RD	16.9	15.3	1.09	1.2

Julienne

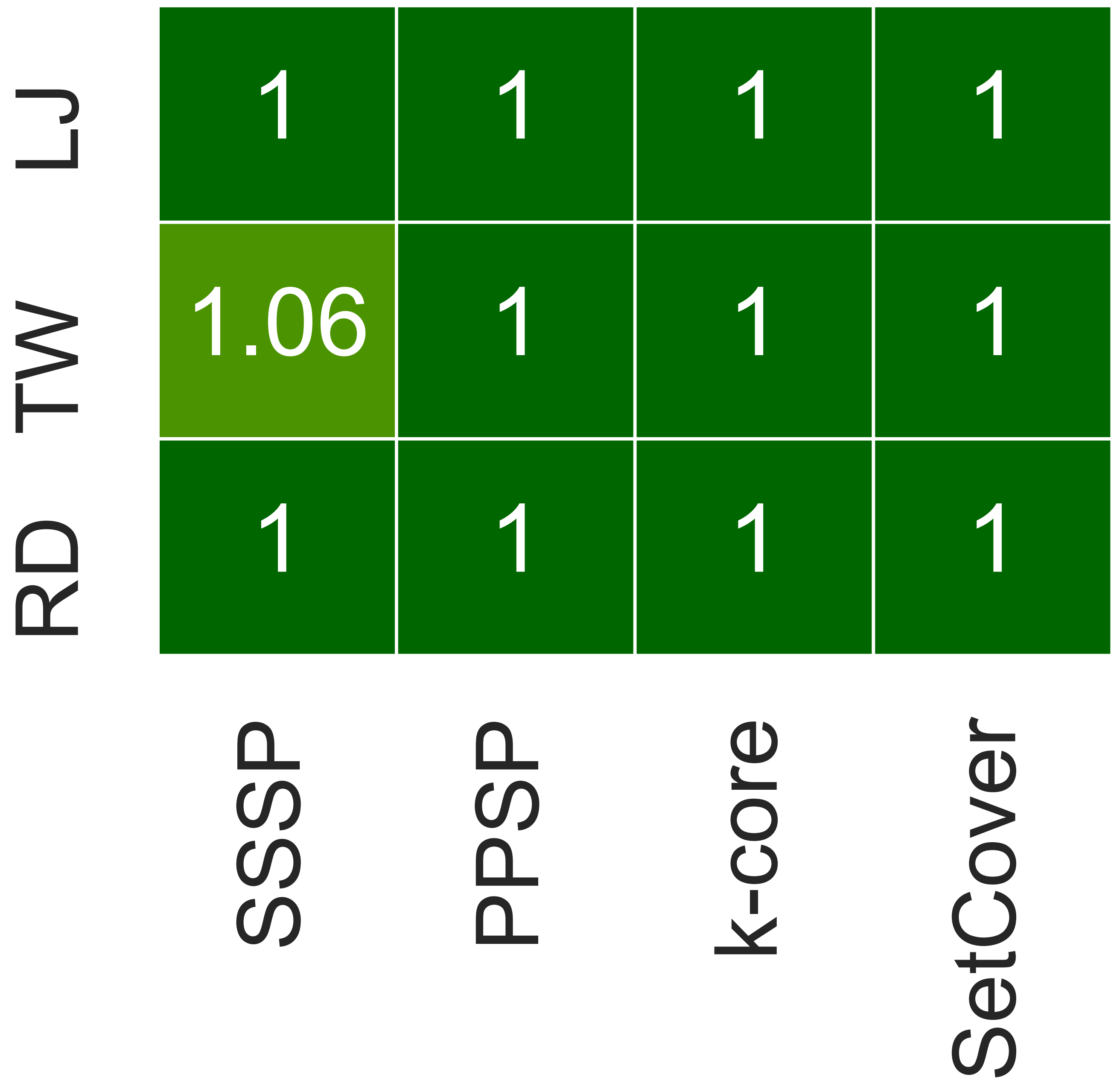


	SSSP	PPSP	k-core	SetCover
LJ	1.32	1.94		
TW	1	1.01		
RD	1.23	1.12		

Galois

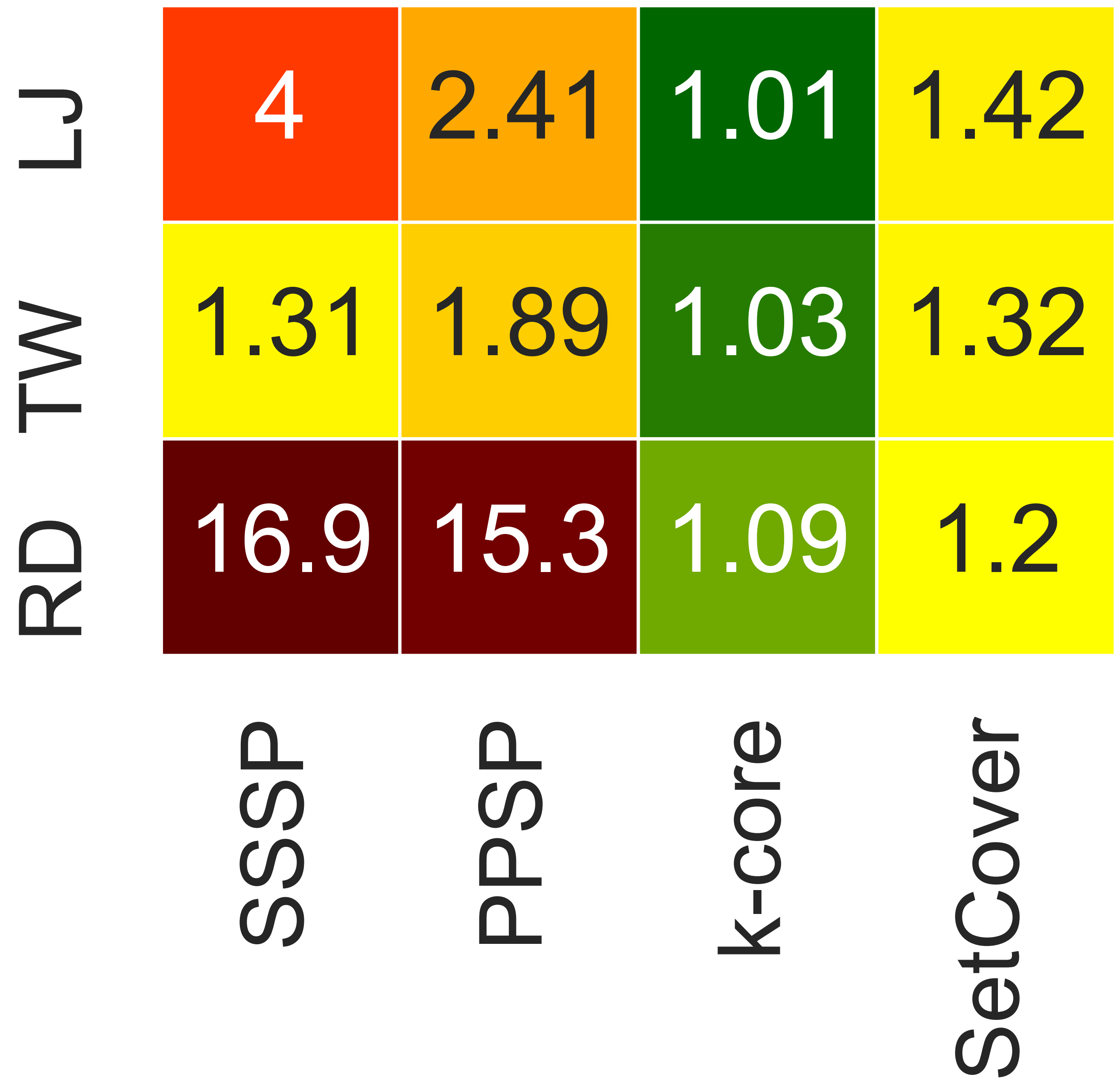
No Support for Eager Bucket Update

Comparisons with Ordered Frameworks



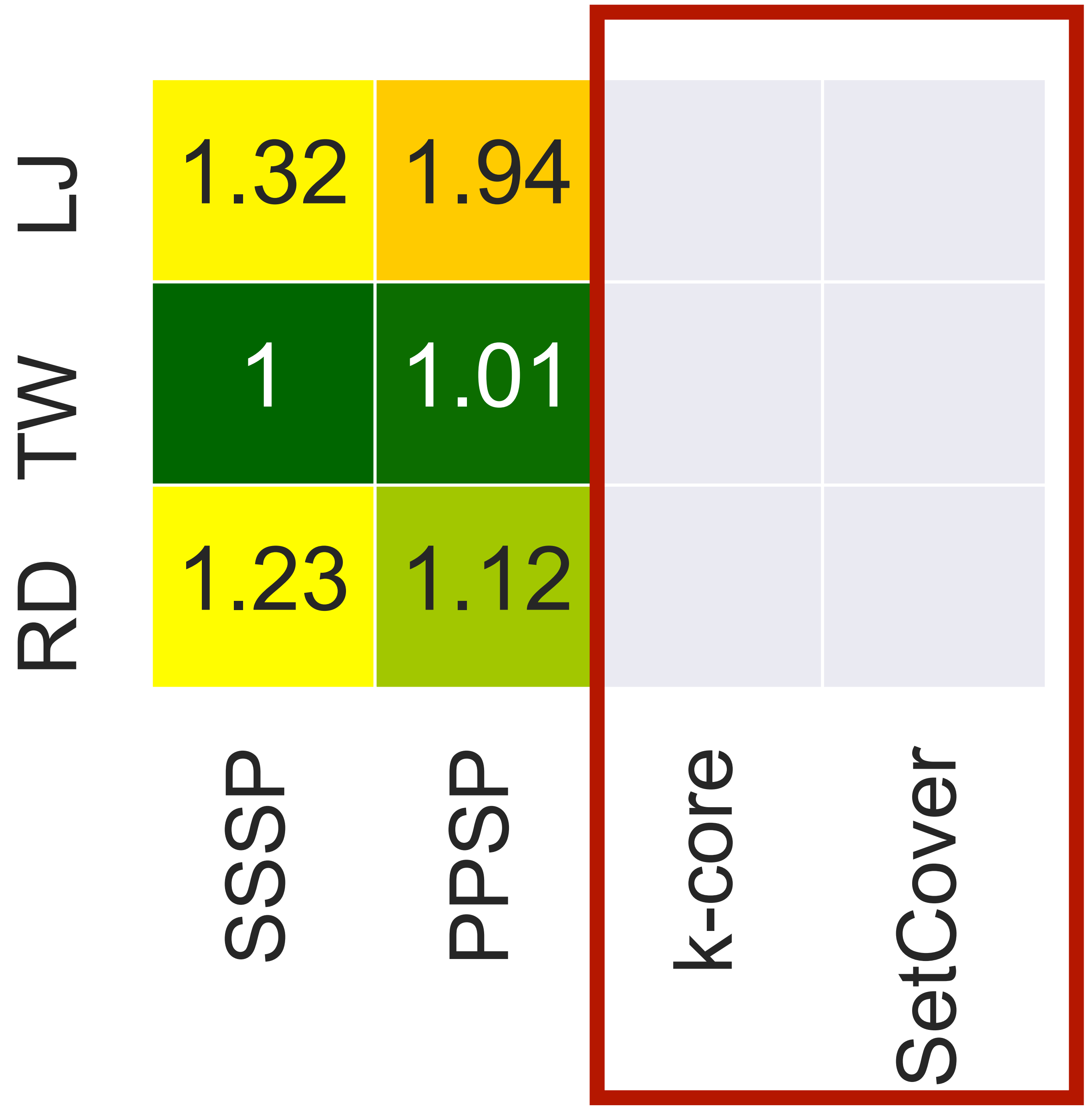
	SSSP	PPSP	k-core	SetCover
LJ	1	1	1	1
TW	1.06	1	1	1
RD	1	1	1	1

Extended GraphIt



	SSSP	PPSP	k-core	SetCover
LJ	4	2.41	1.01	1.42
TW	1.31	1.89	1.03	1.32
RD	16.9	15.3	1.09	1.2

Julienne

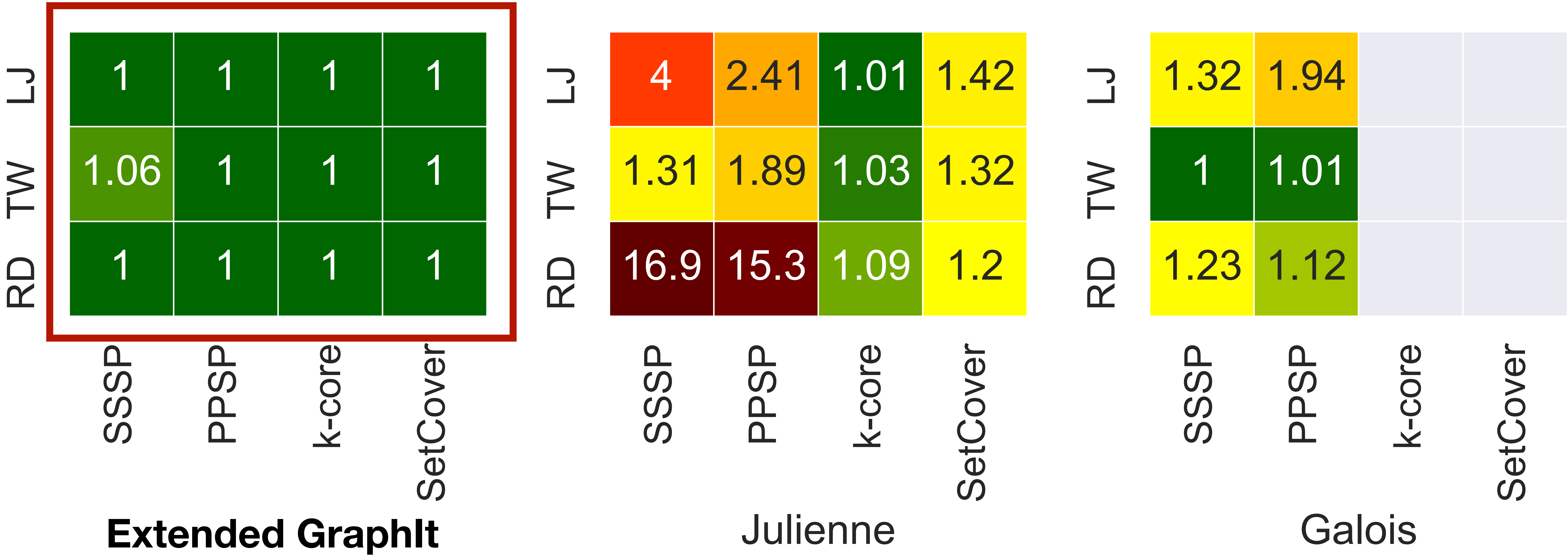


	SSSP	PPSP	k-core	SetCover
LJ	1.32	1.94		
TW	1	1.01		
RD	1.23	1.12		

Galois

No Support for Strict Priority Ordering Needed for Correctness

Comparisons with Ordered Frameworks



Extended GraphIt

	SSSP	PPSP	k-core	SetCover
LJ	1	1	1	1
TW	1.06	1	1	1
RD	1	1	1	1

Julienne

	SSSP	PPSP	k-core	SetCover
LJ	4	2.41	1.01	1.42
TW	1.31	1.89	1.03	1.32
RD	16.9	15.3	1.09	1.2

Galois

	SSSP	PPSP	k-core	SetCover
LJ	1.32	1.94		
TW	1	1.01		
RD	1.23	1.12		

Achieves consistent high-performance across algorithms and graphs

Outline

Hardware Utilization

Programming System to Handle Variety in Data and Algorithms

Variety in Hardware

Making Caches Work for Graph Analytics (BigData17)
Zhang, et al.

GraphIt: a High-Performance Graph DSL (OOPSLA18)
Zhang, et al.

Optimizing Ordered Graph Algorithms with GraphIt (CGO2020)
Zhang, et al.

Universal Graph Framework (Under Submission)
Brahmakshatriya, Zhang, et al.

- **Frequency-based Reordering**
- **Cache-aware Partitioning**

***GraphIt* Compiler and DSL that Decouples**
- **Algorithm**
- **Optimization**
- **Hardware**

for Graph Applications

GPU vs CPU

GPU (Volta)

- Memory Bandwidth: ~900 GB/s
- Compute Power: ~10 TFLOPs
- Cache Size: ~3 MB Shared Cache, ~96KB L1 Cache per SM
- Memory Size: 32 GB
- Less Powerful Cores

CPU (Skylake)

- Memory Bandwidth: ~100 GB/s
- Compute Power: ~1 TFLOPs
- Cache Size: ~64MB Shared Cache, 256KB Private Cache per Core
- Memory Size: Up to 3 TB
- More Powerful Cores with Out-of-Order Execution, Branch predictor, …

No Best Hardware

GPU (Volta)

CPU (Skylake)

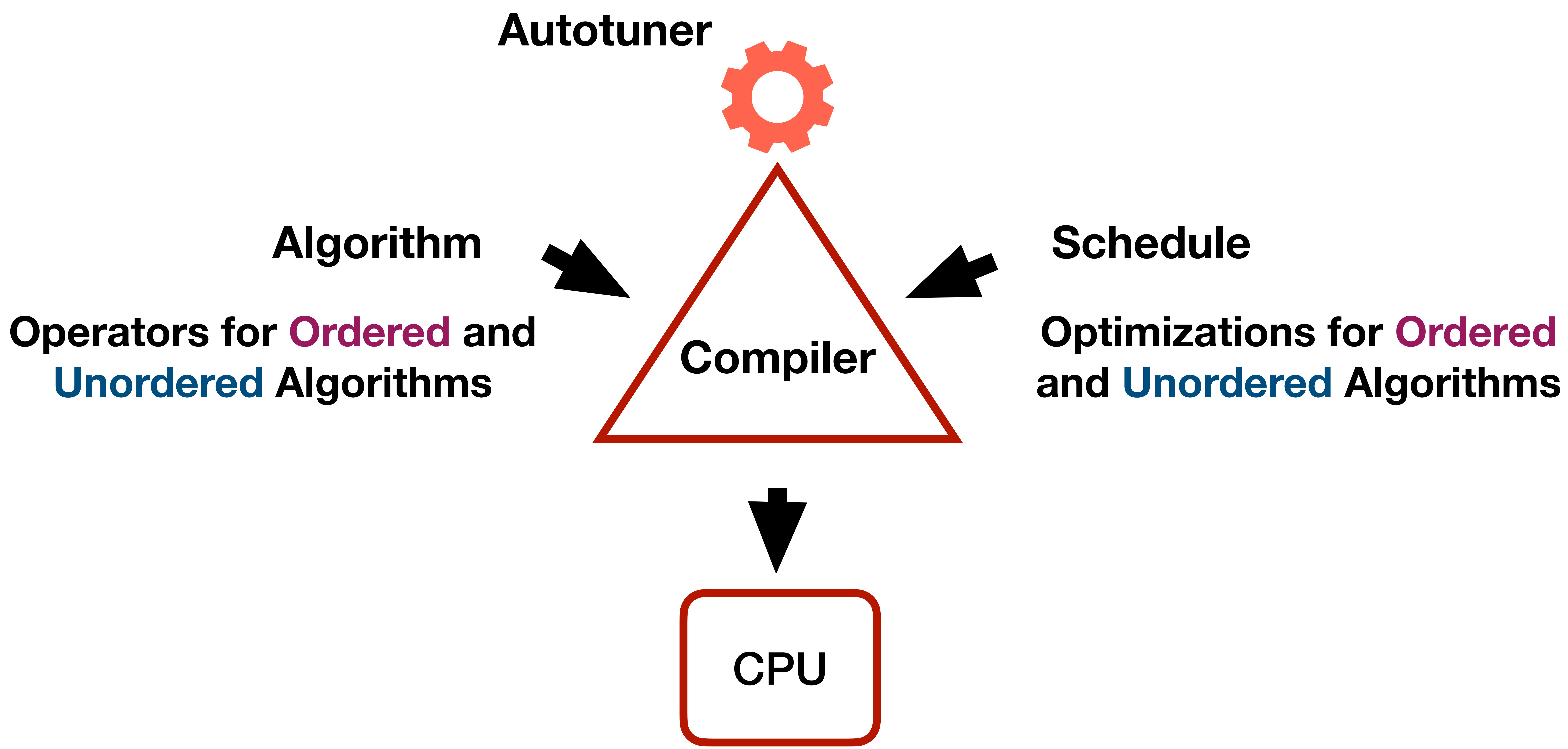
- Throughput-Oriented, Abundant Parallelism, Little Control Flow, Medium-Sized Graphs
  - PageRank, Connected Components, Label Propagation on Social and Web Graphs …

- Latency-Sensitive, Limited parallelism, or Large Graphs
  - SSSP, PPSP, AStar on Road Networks and Smaller social networks, …

Key Optimizations for GPU

- Direction Optimization
- Cache Optimization
- Load Balance
- Kernel Fusion across Iterations
- Active Vertexset Creation
- Active Vertexset Deduplication
- Active Vertexset Processing Ordering (Ordered Algorithms)

GraphIt



Autotuner

Algorithm

Operators for Ordered and Unordered Algorithms

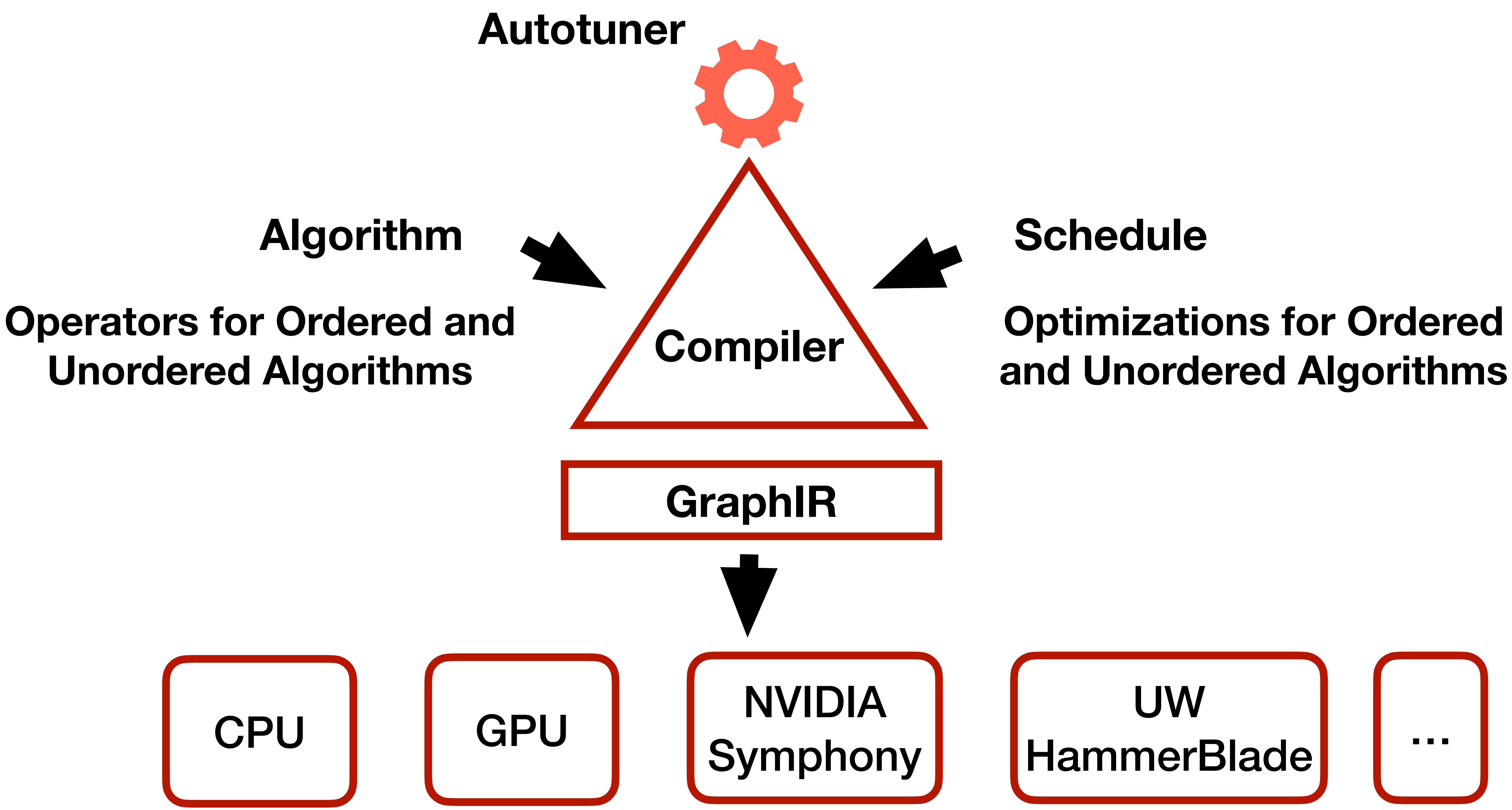
Compiler

Schedule

Optimizations for Ordered and Unordered Algorithms

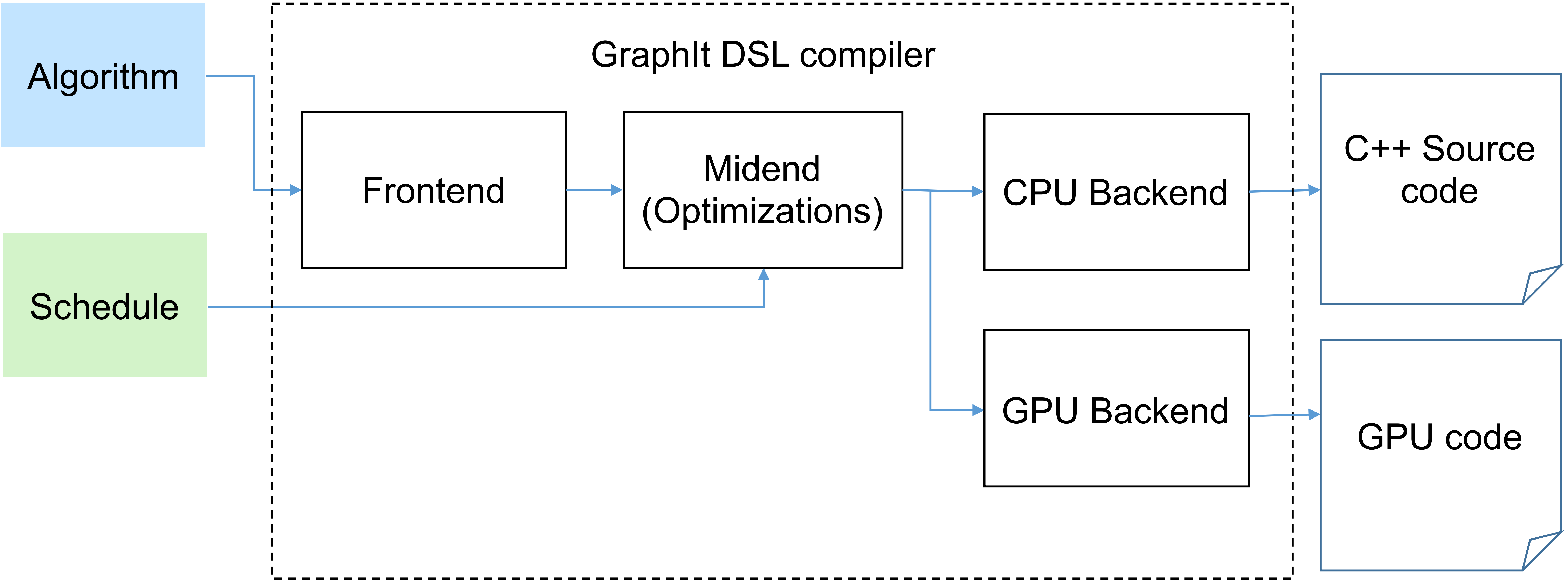
CPU

Portability

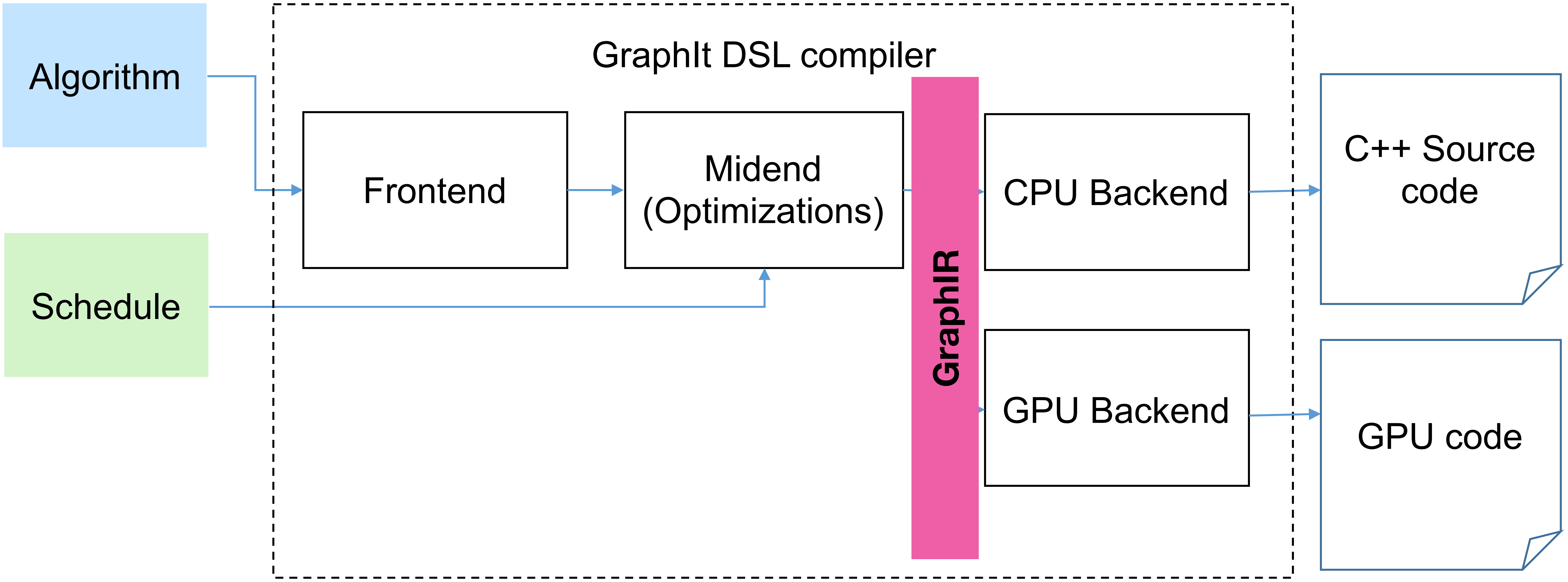

Autotuner
Algorithm
Operators for Ordered and Unordered Algorithms
Compiler
Schedule
Optimizations for Ordered and Unordered Algorithms
GraphIR
CPU
GPU
NVIDIA Symphony
UW HammerBlade
...

Work In Progress Brahmakshatriya, Zhang, Hong, et al.

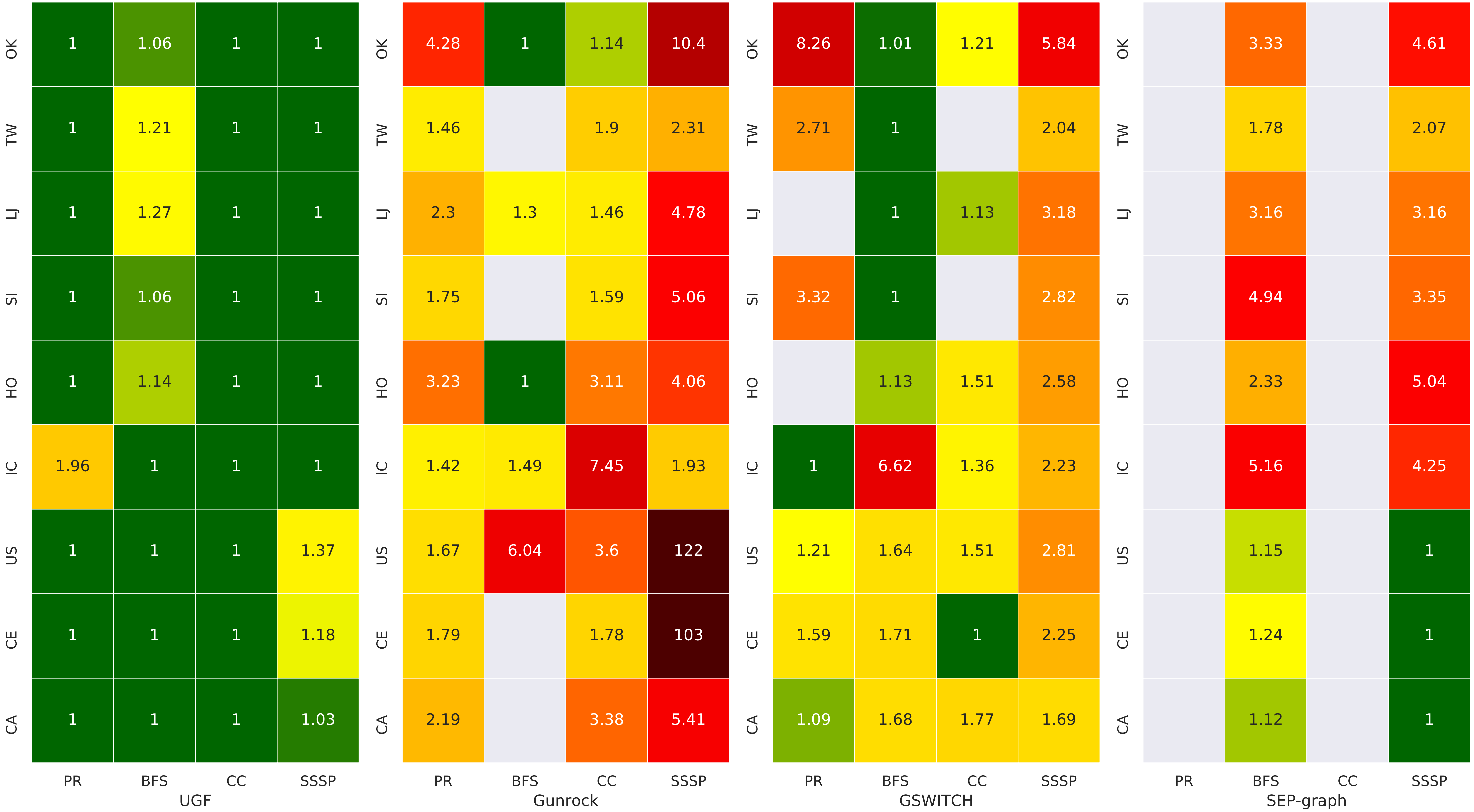
GraphIR


Algorithm
Schedule
GraphIt DSL compiler
Frontend
Midend (Optimizations)
CPU Backend
GPU Backend
C++ Source code
GPU code

GraphIR


Algorithm
Schedule
GraphIt DSL compiler
Frontend
Midend
(Optimizations)
GraphIR
CPU Backend
GPU Backend
C++ Source
code
GPU code

GPU Evaluation



	PR	BFS	CC	SSSP
OK	1	1.06	1	1
TW	1	1.21	1	1
LJ	1	1.27	1	1
SI	1	1.06	1	1
HO	1	1.14	1	1
IC	1.96	1	1	1
US	1	1	1	1.37
CE	1	1	1	1.18
CA	1	1	1	1.03

UGF

	PR	BFS	CC	SSSP
OK	4.28	1	1.14	10.4
TW	1.46		1.9	2.31
LJ	2.3	1.3	1.46	4.78
SI	1.75		1.59	5.06
HO	3.23	1	3.11	4.06
IC	1.42	1.49	7.45	1.93
US	1.67	6.04	3.6	122
CE	1.79		1.78	103
CA	2.19		3.38	5.41

Gunrock

	PR	BFS	CC	SSSP
OK	8.26	1.01	1.21	5.84
TW	2.71	1		2.04
LJ		1	1.13	3.18
SI	3.32	1		2.82
HO		1.13	1.51	2.58
IC	1	6.62	1.36	2.23
US	1.21	1.64	1.51	2.81
CE	1.59	1.71	1	2.25
CA	1.09	1.68	1.77	1.69

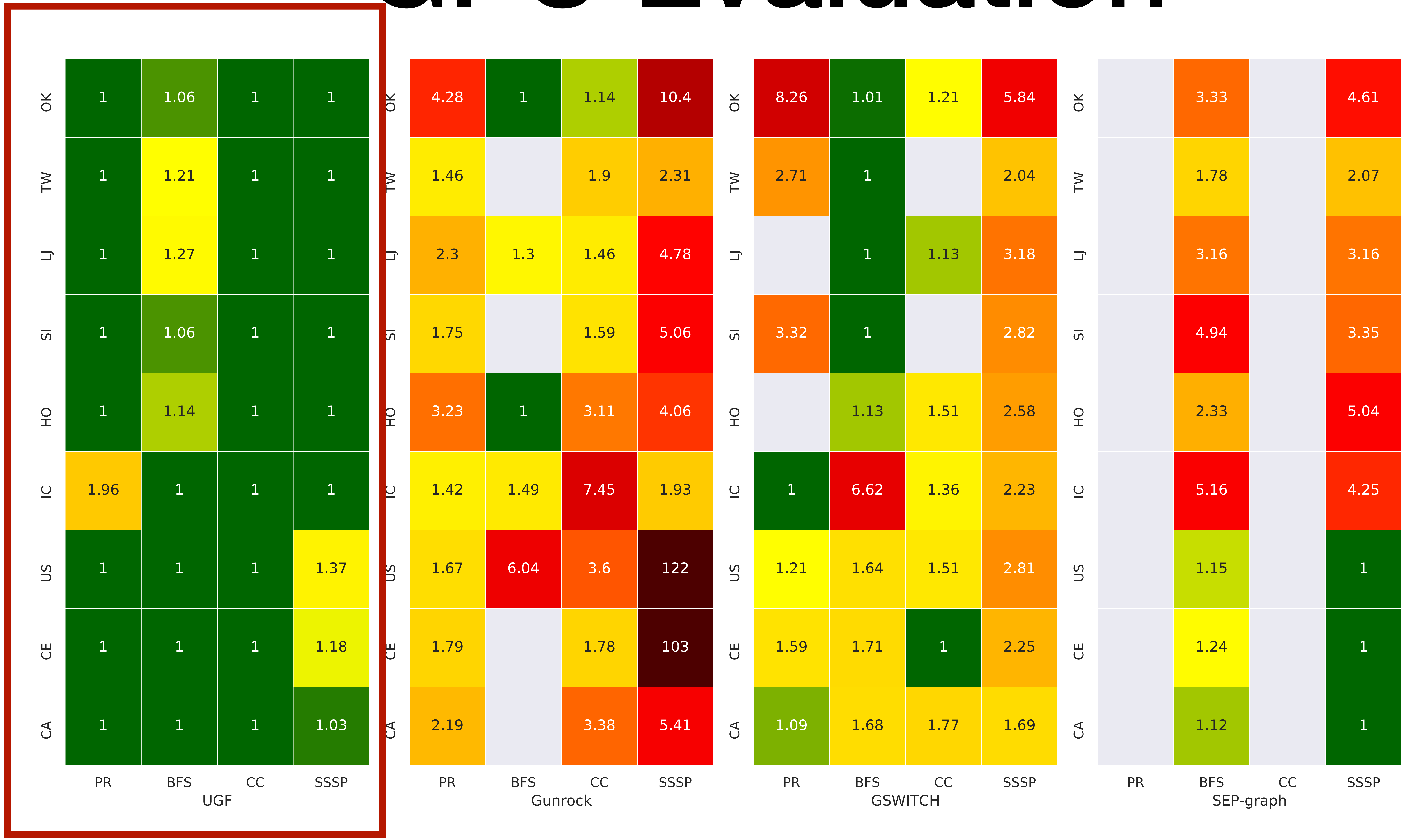
GSWITCH

	PR	BFS	CC	SSSP
OK		3.33		4.61
TW		1.78		2.07
LJ		3.16		3.16
SI		4.94		3.35
HO		2.33		5.04
IC		5.16		4.25
US		1.15		1
CE		1.24		1
CA		1.12		1

SEP-graph

Achieves Consistent High Performance over Different Algorithms and Graphs on GPU

GPU Evaluation



	PR	BFS	CC	SSSP
OK	1	1.06	1	1
TW	1	1.21	1	1
LJ	1	1.27	1	1
SI	1	1.06	1	1
HO	1	1.14	1	1
IC	1.96	1	1	1
US	1	1	1	1.37
CE	1	1	1	1.18
CA	1	1	1	1.03

UGF

	PR	BFS	CC	SSSP
OK	4.28	1	1.14	10.4
TW	1.46		1.9	2.31
LJ	2.3	1.3	1.46	4.78
SI	1.75		1.59	5.06
HO	3.23	1	3.11	4.06
IC	1.42	1.49	7.45	1.93
US	1.67	6.04	3.6	122
CE	1.79		1.78	103
CA	2.19		3.38	5.41

Gunrock

	PR	BFS	CC	SSSP
OK	8.26	1.01	1.21	5.84
TW	2.71	1		2.04
LJ		1	1.13	3.18
SI	3.32	1		2.82
HO		1.13	1.51	2.58
IC	1	6.62	1.36	2.23
US	1.21	1.64	1.51	2.81
CE	1.59	1.71	1	2.25
CA	1.09	1.68	1.77	1.69

GSWITCH

	PR	BFS	CC	SSSP
OK		3.33		4.61
TW		1.78		2.07
LJ		3.16		3.16
SI		4.94		3.35
HO		2.33		5.04
IC		5.16		4.25
US		1.15		1
CE		1.24		1
CA		1.12		1

SEP-graph

Related Work

DSLs that separates algorithm from schedules and data layouts:

Halide [PLDI13], Tiramisu [CGO19], Taco [CGO19], TVM [OSDI18], Taichi [SIGGRAPH Asia19] …

Graph frameworks, optimizations, and architectures that support unordered parallelism:

Ligra [PPoPP13], Galois [SOSP13], GraphBLAS[HPEC 2016], GunRock[PPoPP16], Propagation Blocking [IPDPS17], Cagra [BigData17], Grazelle [PPoPP18], GSwitch[PPoPP19], Sep-Graph[PPoPP19], PHI [MICRO19] …

Graph frameworks, optimizations, and architectures that support ordered parallelism

Julienne [SPAA17], Galois [SOSP13, PPoPP11], GAPBS [arxiv, IISWC15], Swarm [MICRO15, MICRO16, ISCA17, MICRO18]

Graph DSLs:

GreenMarl [ASPLOS12], EmptyHeaded [SIGMOD16], Elixir [OOPSLA12], Gluon [PLDI18], Abelian [EuroPar18]…

Related Work

Focus on the Graph Domain

DSLs that separates algorithm from schedules and data layouts:

Halide [PLDI13], Tiramisu [CGO19], Taco [CGO19], TVM [OSDI18], Taichi [SIGGRAPH Asia19] …

Graph frameworks, optimizations, and architectures that support unordered parallelism:

Ligra [PPoPP13], Galois [SOSP13], GraphBLAS[HPEC 2016], GunRock[PPoPP16], Propagation Blocking [IPDPS17], Cagra [BigData17], Grazelle [PPoPP18], GSwitch[PPoPP19], Sep-Graph[PPoPP19], PHI [MICRO19] …

Focus on Cache and Composing Optimizations

Graph frameworks, optimizations, and architectures that support ordered parallelism

Julienne [SPAA17], Galois [SOSP13, PPoPP11], GAPBS [arxiv, IISWC15], Swarm [MICRO15, MICRO16, ISCA17, MICRO18]

Graph DSLs:

GreenMarl [ASPLOS12], EmptyHeaded [SIGMOD16], Elixir [OOPSLA12], Gluon [PLDI18], Abelian [EuroPar18]…

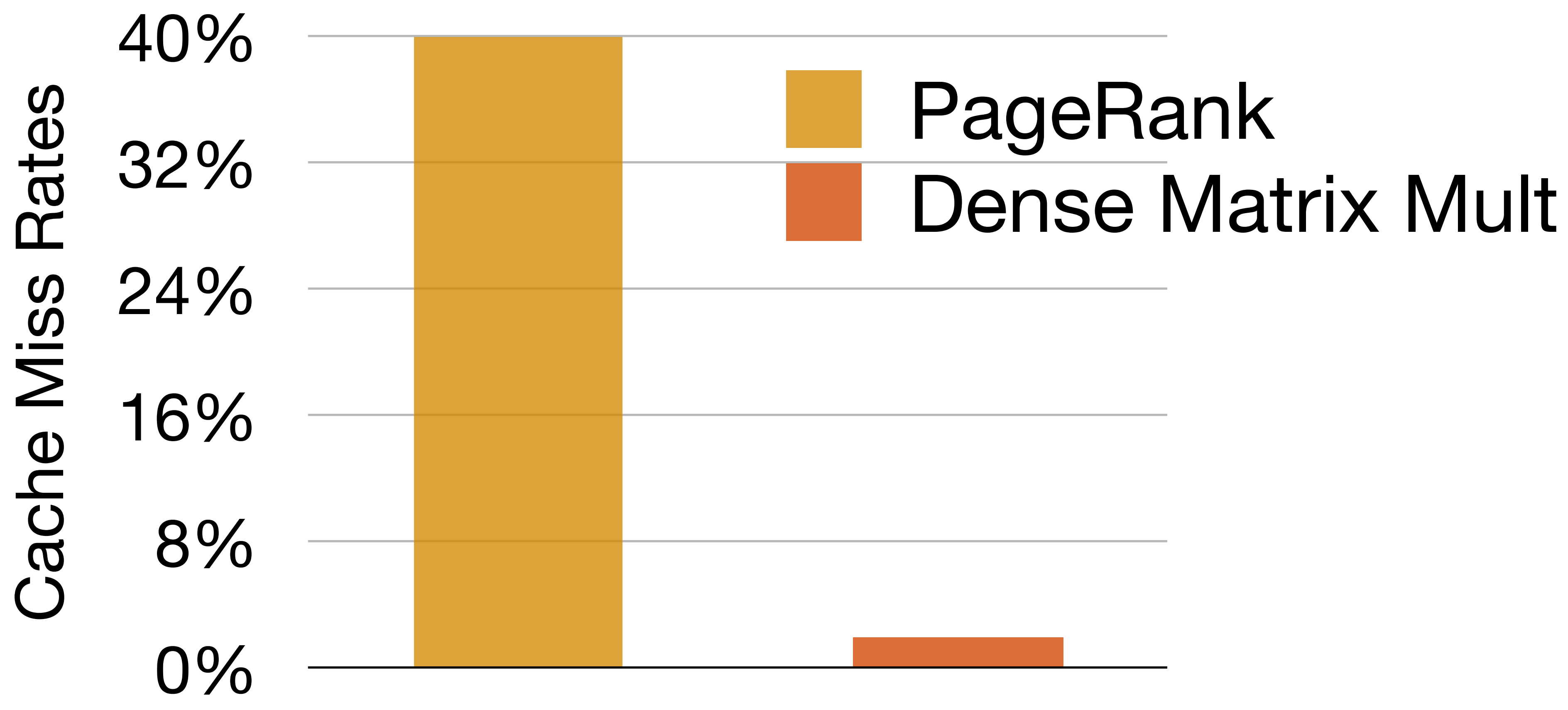
Focus on Searching through Optimization Space

Lessons Learned

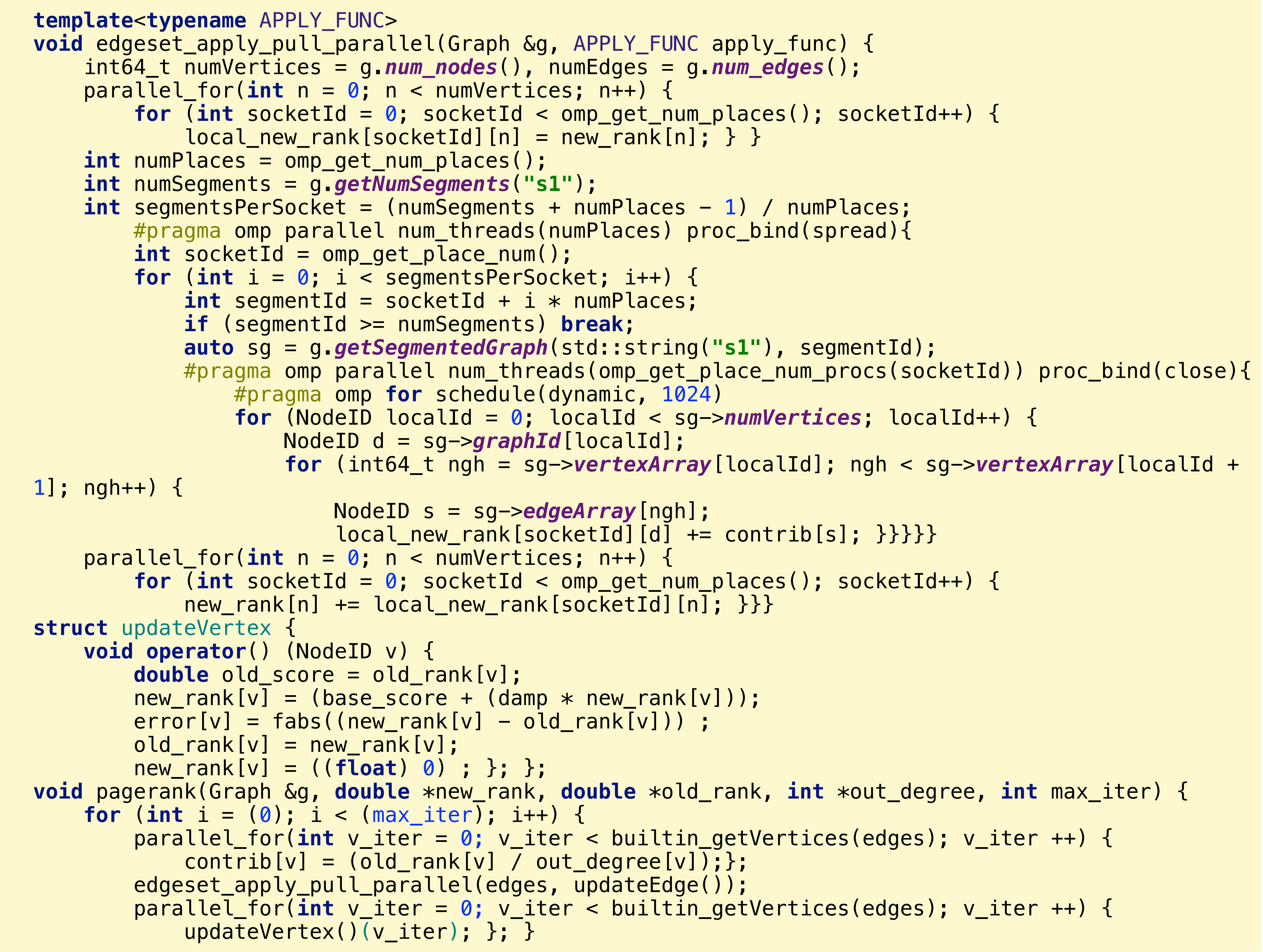
- No set of optimizations fits all graph applications well
- Select the best algorithm, optimization strategy, and hardware platform for each application
- Need software optimizations to fully utilize the underlying hardware (CPU or GPU)

Sparse Graph Computations

- Hardware Utilization
  - Peak Performance (PageRank, SpMv)
    - < 10% Peak of CPU and GPU



- Programming System

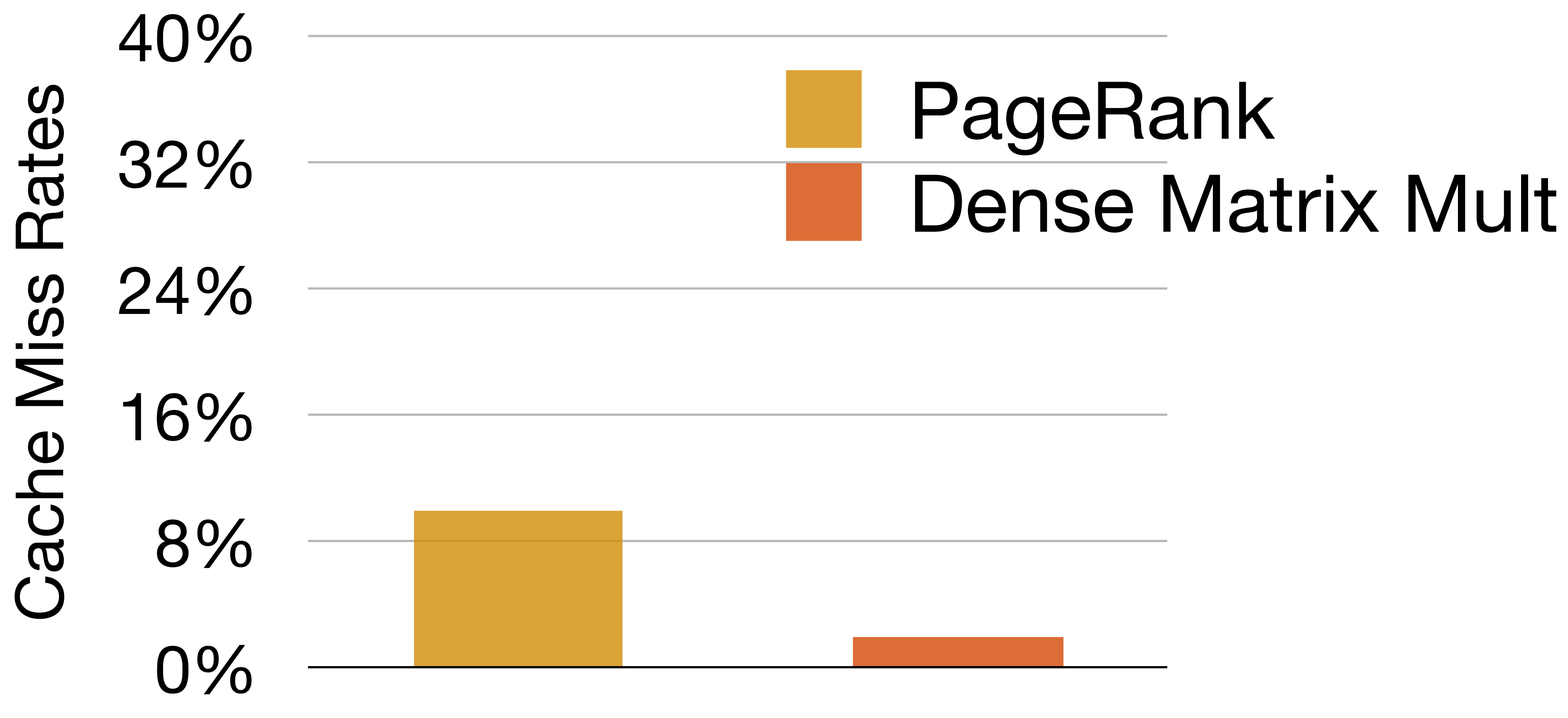


```
template<typename APPLY_FUNC>
void edgeset_apply_pull_parallel(Graph &g, APPLY_FUNC apply_func) {
    int64_t numVertices = g.num_nodes(), numEdges = g.num_edges();
    parallel_for(int n = 0; n < numVertices; n++) {
        for (int socketId = 0; socketId < omp_get_num_places(); socketId++) {
            local_new_rank[socketId][n] = new_rank[n]; } }
    int numPlaces = omp_get_num_places();
    int numSegments = g.getNumSegments("s1");
    int segmentsPerSocket = (numSegments + numPlaces − 1) / numPlaces;
        #pragma omp parallel num_threads(numPlaces) proc_bind(spread){
        int socketId = omp_get_place_num();
        for (int i = 0; i < segmentsPerSocket; i++) {
            int segmentId = socketId + i ∗ numPlaces;
            if (segmentId >= numSegments) break;
            auto sg = g.getSegmentedGraph(std::string("s1"), segmentId);
            #pragma omp parallel num_threads(omp_get_place_num_procs(socketId)) proc_bind(close){
                #pragma omp for schedule(dynamic, 1024)
                for (NodeID localId = 0; localId < sg->numVertices; localId++) {
                    NodeID d = sg->graphId[localId];
                    for (int64_t ngh = sg->vertexArray[localId]; ngh < sg->vertexArray[localId +
1]; ngh++) {
                        NodeID s = sg->edgeArray[ngh];
                        local_new_rank[socketId][d] += contrib[s]; }}}}}
    parallel_for(int n = 0; n < numVertices; n++) {
        for (int socketId = 0; socketId < omp_get_num_places(); socketId++) {
            new_rank[n] += local_new_rank[socketId][n]; }}}
struct updateVertex {
    void operator() (NodeID v) {
        double old_score = old_rank[v];
        new_rank[v] = (base_score + (damp ∗ new_rank[v]));
        error[v] = fabs((new_rank[v] − old_rank[v])) ;
        old_rank[v] = new_rank[v];
        new_rank[v] = ((float) 0) ; }; };
void pagerank(Graph &g, double ∗new_rank, double ∗old_rank, int ∗out_degree, int max_iter) {
    for (int i = (0); i < (max_iter); i++) {
        parallel_for(int v_iter = 0; v_iter < builtin_getVertices(edges); v_iter ++) {
            contrib[v] = (old_rank[v] / out_degree[v]);};
        edgeset_apply_pull_parallel(edges, updateEdge());
        parallel_for(int v_iter = 0; v_iter < builtin_getVertices(edges); v_iter ++) {
            updateVertex()(v_iter); }; }
```

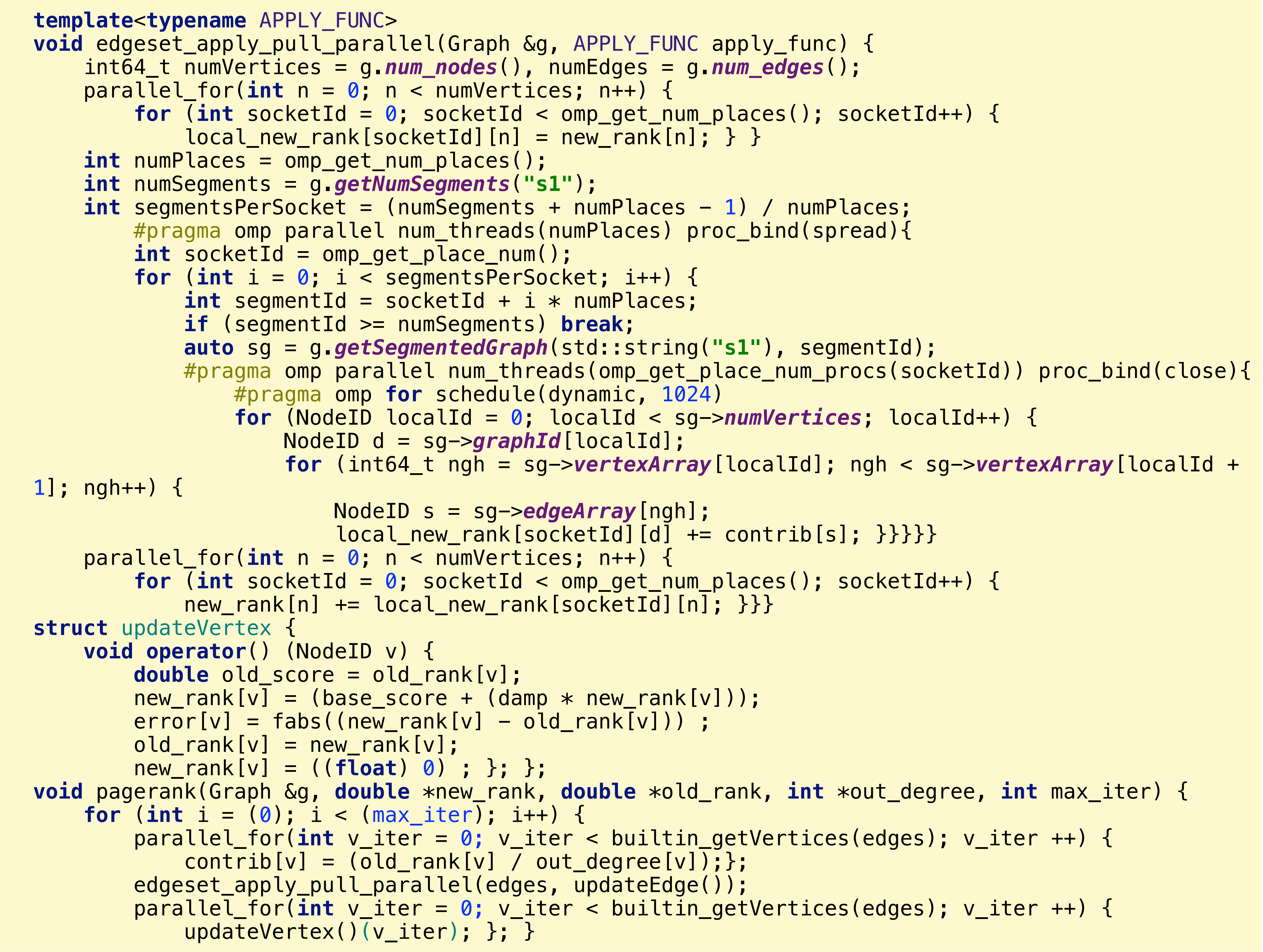
Optimized PageRank for Multi-Core CPU

Sparse Graph Computations

- Hardware Utilization
  - Peak Performance (PageRank, SpMv)
    - ~ 20% Peak of CPU and GPU with cache and other optimizations



- Programming System

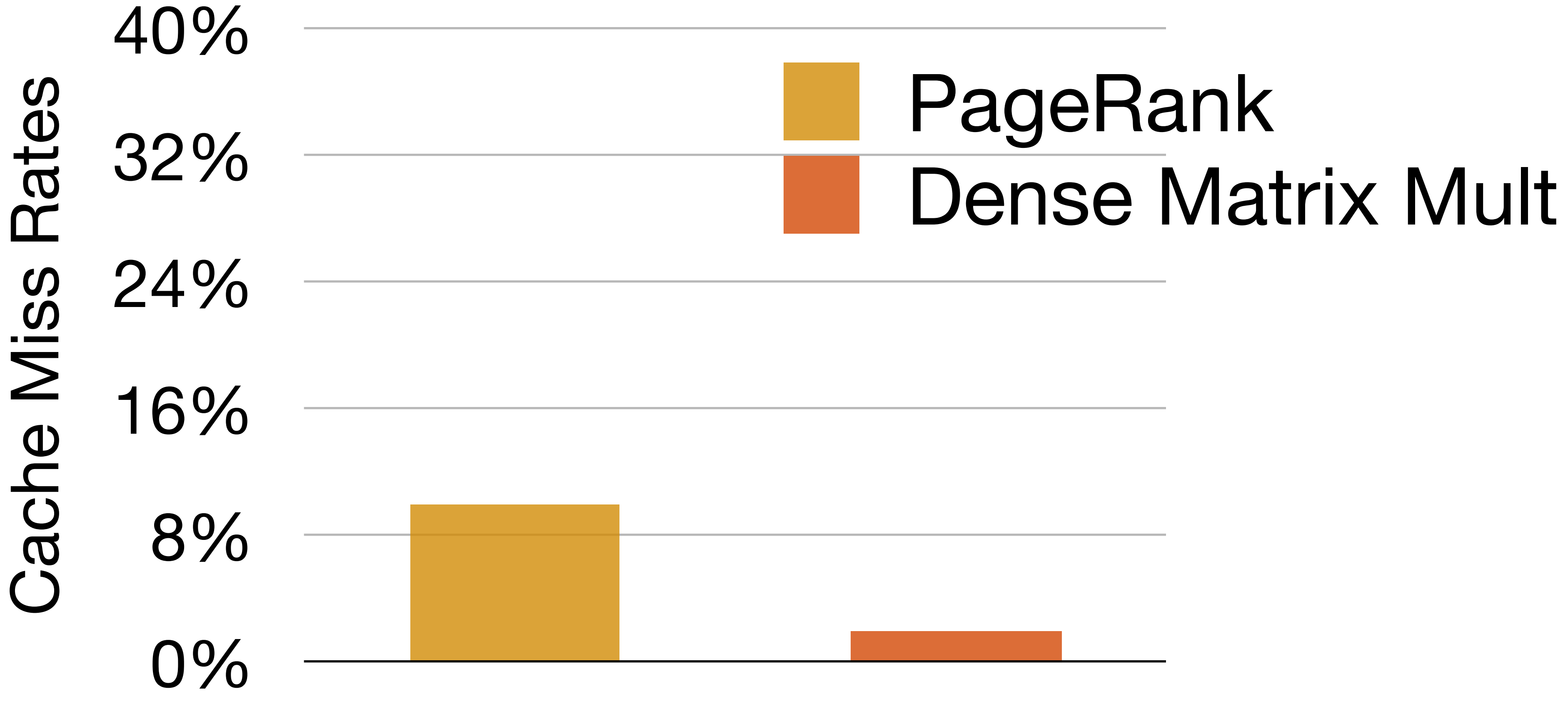


```
template<typename APPLY_FUNC>
void edgeset_apply_pull_parallel(Graph &g, APPLY_FUNC apply_func) {
    int64_t numVertices = g.num_nodes(), numEdges = g.num_edges();
    parallel_for(int n = 0; n < numVertices; n++) {
        for (int socketId = 0; socketId < omp_get_num_places(); socketId++) {
            local_new_rank[socketId][n] = new_rank[n]; } }
    int numPlaces = omp_get_num_places();
    int numSegments = g.getNumSegments("s1");
    int segmentsPerSocket = (numSegments + numPlaces - 1) / numPlaces;
        #pragma omp parallel num_threads(numPlaces) proc_bind(spread){
        int socketId = omp_get_place_num();
        for (int i = 0; i < segmentsPerSocket; i++) {
            int segmentId = socketId + i * numPlaces;
            if (segmentId >= numSegments) break;
            auto sg = g.getSegmentedGraph(std::string("s1"), segmentId);
            #pragma omp parallel num_threads(omp_get_place_num_procs(socketId)) proc_bind(close){
                #pragma omp for schedule(dynamic, 1024)
                for (NodeID localId = 0; localId < sg->numVertices; localId++) {
                    NodeID d = sg->graphId[localId];
                    for (int64_t ngh = sg->vertexArray[localId]; ngh < sg->vertexArray[localId +
1]; ngh++) {
                        NodeID s = sg->edgeArray[ngh];
                        local_new_rank[socketId][d] += contrib[s]; }}}}}
    parallel_for(int n = 0; n < numVertices; n++) {
        for (int socketId = 0; socketId < omp_get_num_places(); socketId++) {
            new_rank[n] += local_new_rank[socketId][n]; }}}
struct updateVertex {
    void operator() (NodeID v) {
        double old_score = old_rank[v];
        new_rank[v] = (base_score + (damp * new_rank[v]));
        error[v] = fabs((new_rank[v] - old_rank[v])) ;
        old_rank[v] = new_rank[v];
        new_rank[v] = ((float) 0) ; }; };
void pagerank(Graph &g, double *new_rank, double *old_rank, int *out_degree, int max_iter) {
    for (int i = (0); i < (max_iter); i++) {
        parallel_for(int v_iter = 0; v_iter < builtin_getVertices(edges); v_iter ++) {
            contrib[v] = (old_rank[v] / out_degree[v]);};
        edgeset_apply_pull_parallel(edges, updateEdge());
        parallel_for(int v_iter = 0; v_iter < builtin_getVertices(edges); v_iter ++) {
            updateVertex()(v_iter); }; }
```

Optimized PageRank for Multi-Core CPU with C++

Sparse Graph Computations

- Hardware Utilization
  - Peak Performance (PageRank, SpMv)
    - ~ 20% Peak of CPU and GPU with cache and other optimizations



- Programming System

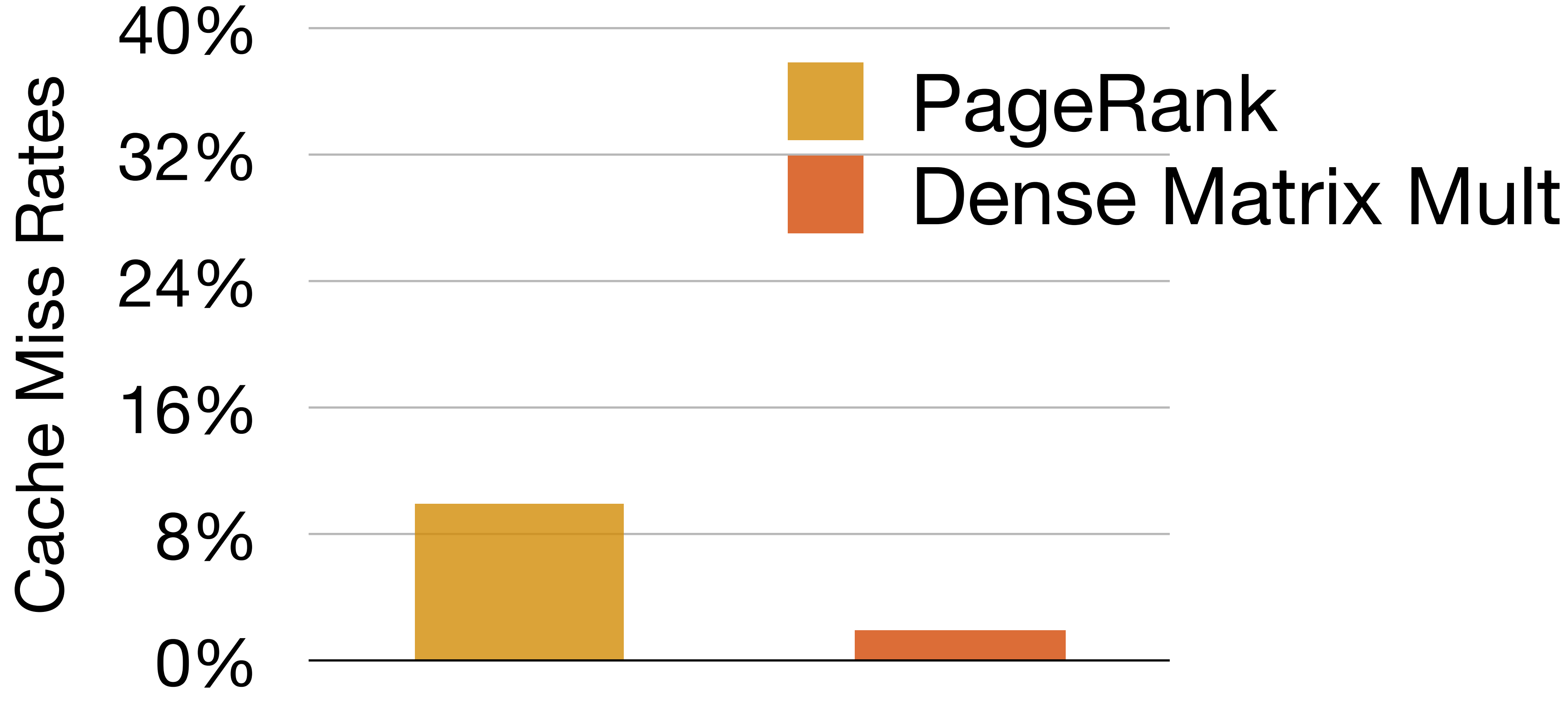
```
func updateEdge (src: Vertex, dst: Vertex)
   new_rank[dst] += old_rank[src] / out_degree[src]
end

func updateVertex (v: Vertex)
   new_rank[v] = base_score + 0.85*new_rank[v];
   old_rank[v] = new_rank[v];
   new_rank[v] = 0;
end

 func main()
    for i in 1:11
       #s1# edges.apply(updateEdge);
       vertices.apply(updateVertex);
    end
 end
```

```
schedule:
    program->configApplyDirection(“s1”, “DensePull”);
    program->configApplyParallelization(“s1”, “dynamic-vertex-parallel”);
    program->configApplyNumSSG(“s1”, “fixed-vertex-count”, 10);
```

- Open source (**graphit-lang.org**).

Sparse Graph Computations

- Hardware Utilization
  - Peak Performance (PageRank, SpMv)
    - ~ 20% Peak of CPU and GPU with cache and other optimizations



- Programming System
  - Easy-to-Use
  - High-Performance across Different Algorithms and Data
  - Portable across Architectures
  - Open source (**graphit-lang.org**).

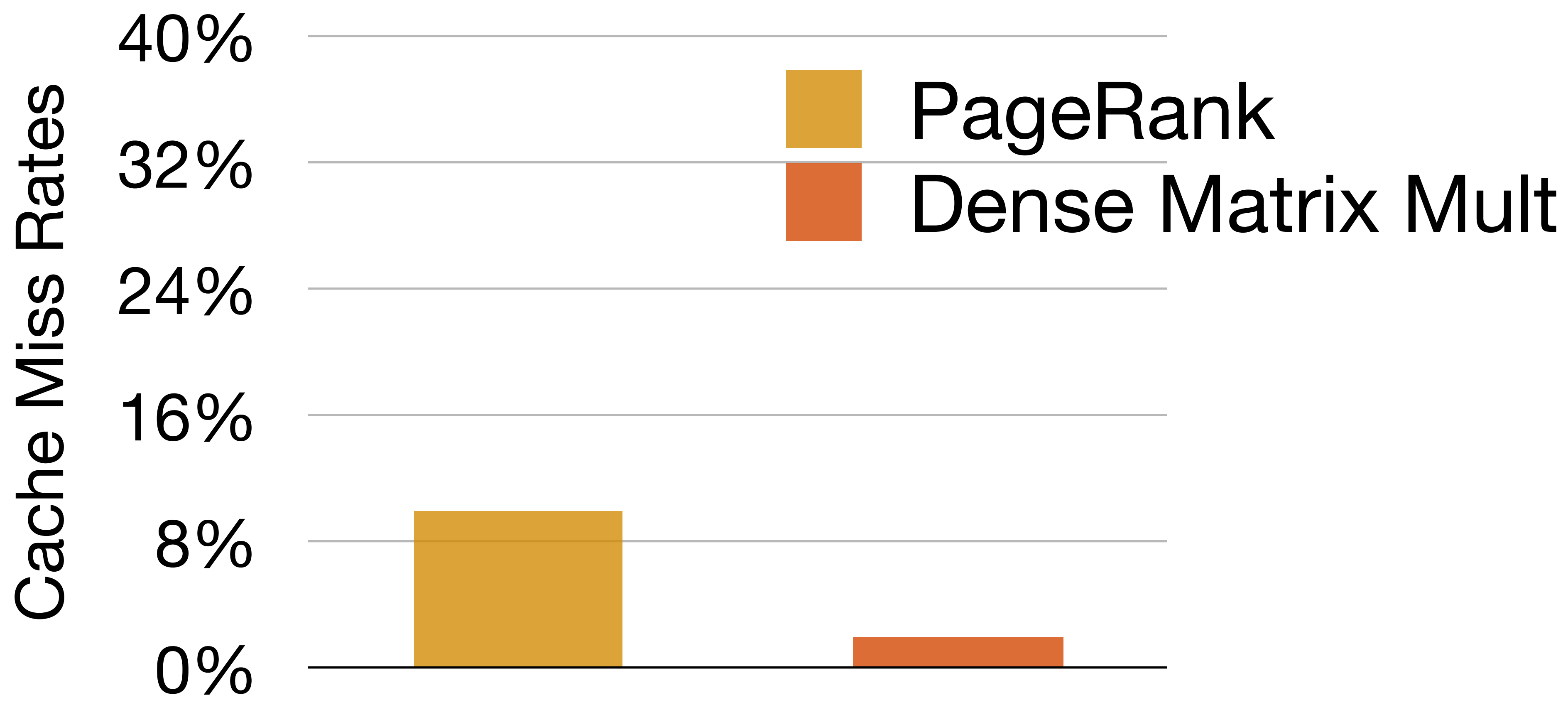
Acknowledgements

- Professors and Postdocs
  - Saman Amarasinghe, Julian Shun, Shoaib Kamil, Matei Zaharia, Riyadh Baghdadi, Changwan Hong
- PhD Students
  - Ajay Brahmakshatriya, Vladimir Kiriansky, Laxman Dhulipala, Charith Mendis
- Master and Undergraduate Students
  - Mengjiao (Sherry) Yang, Tugsbayasgalan Manlaibaatar, Xinyi Chen, Claire Hsu, Haokuan Luo, Kenny Yang

This Work Supported By: DARPA TOYOTA RESEARCH INSTITUTE NSF DEPARTMENT OF ENERGY UNITED STATES OF AMERICA SRC® Semiconductor Research Corporation

Summary

- Hardware Utilization
  - Peak Performance (PageRank, SpMv)
    - ~ 20% Peak of CPU and GPU with cache and other optimizations



- Programming System
  - Easy-to-Use
  - High-Performance across Different Algorithms and Data
  - Portable across Architectures
  - Open source (**graphit-lang.org**).