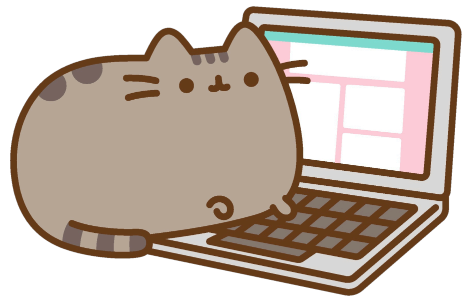
CSE 6230:
HPC Tools and Applications

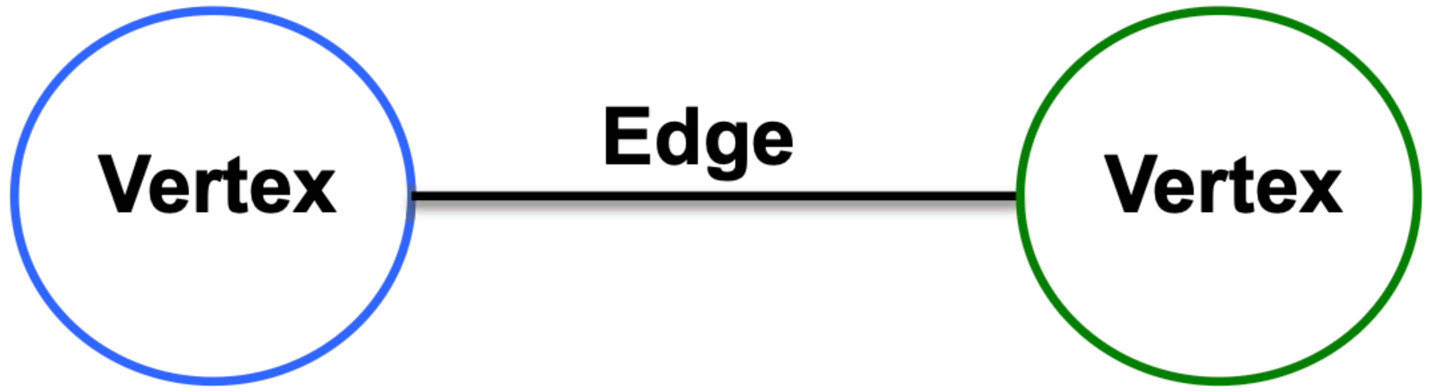

+


Lecture 16: Graph Optimization

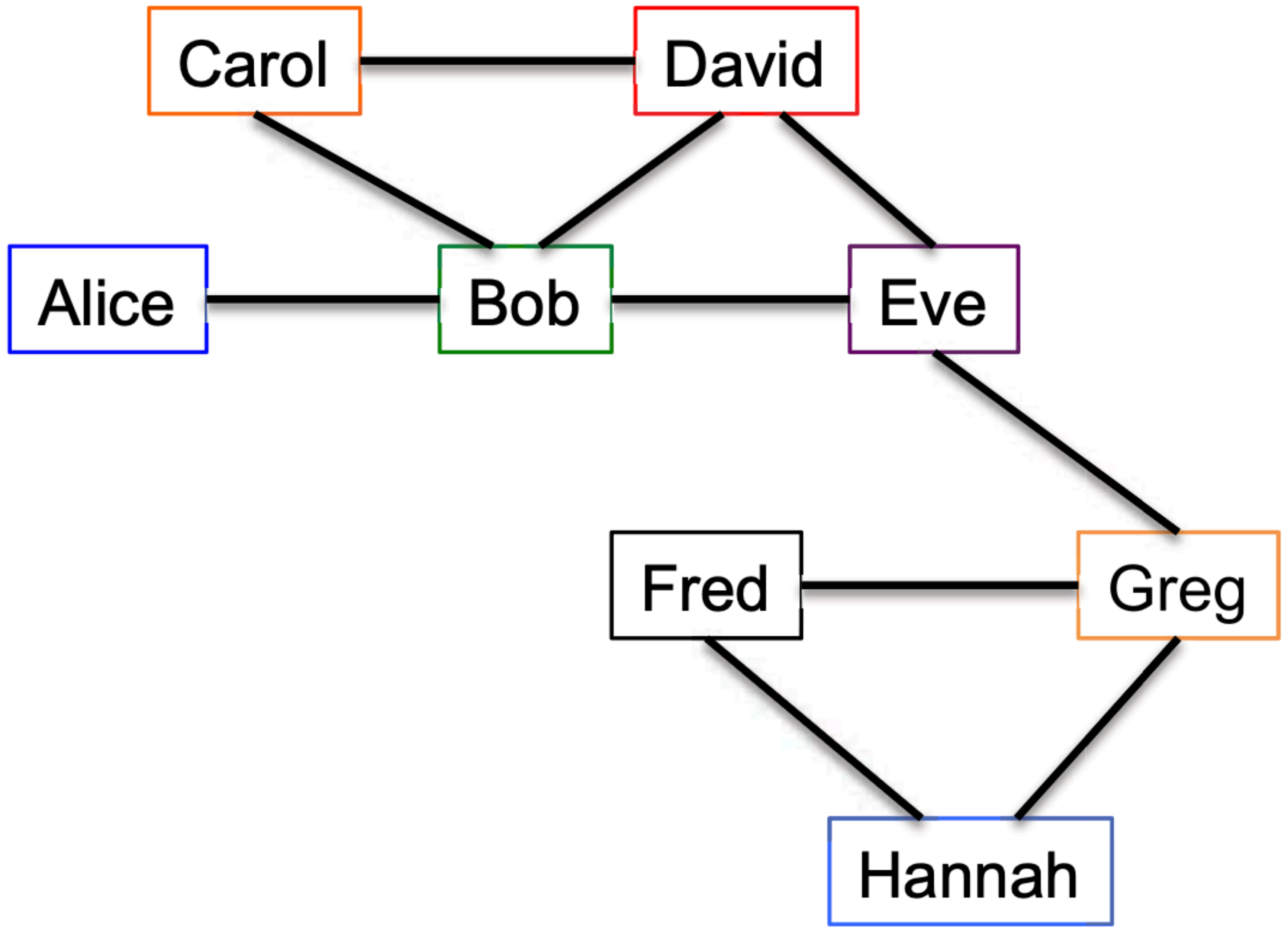
Helen Xu
hxu615@gatech.edu

Georgia Tech College of Computing
School of Computational Science and Engineering

What is a graph?



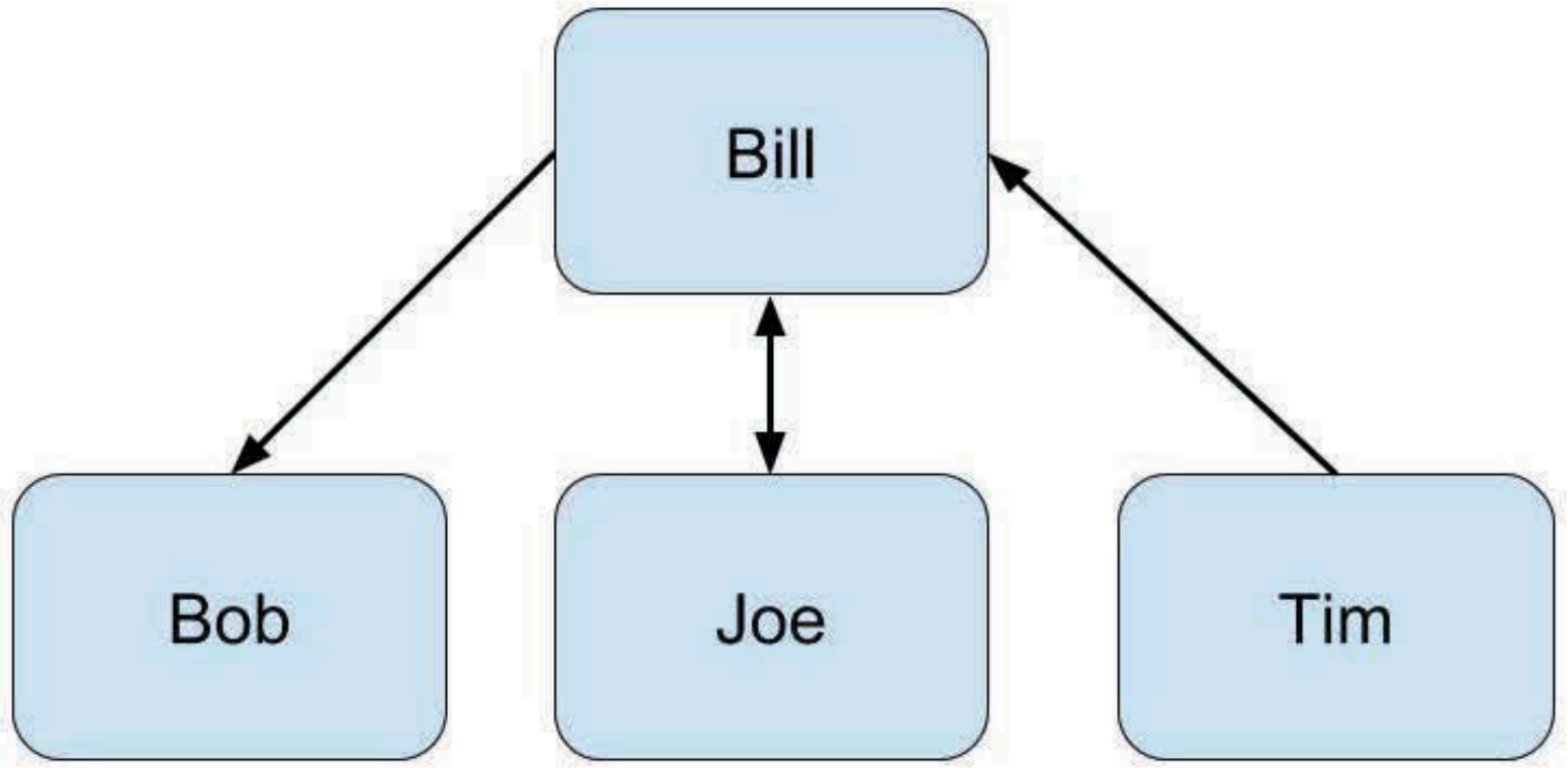
Vertices model objects, edges model **relationship between objects**



What is a graph?

Edges can be **directed**

- Relationship can go one way or both ways

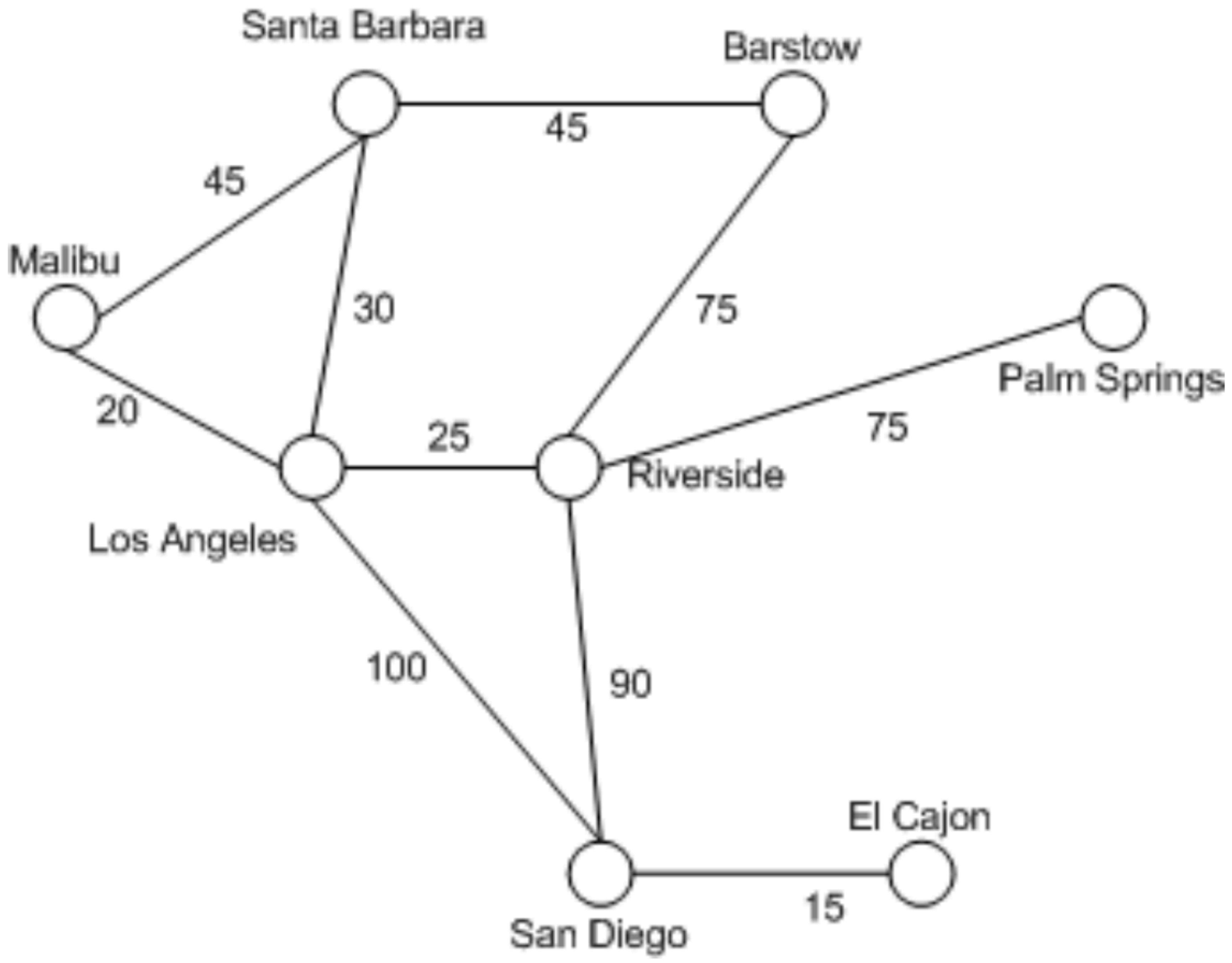


What is a graph?

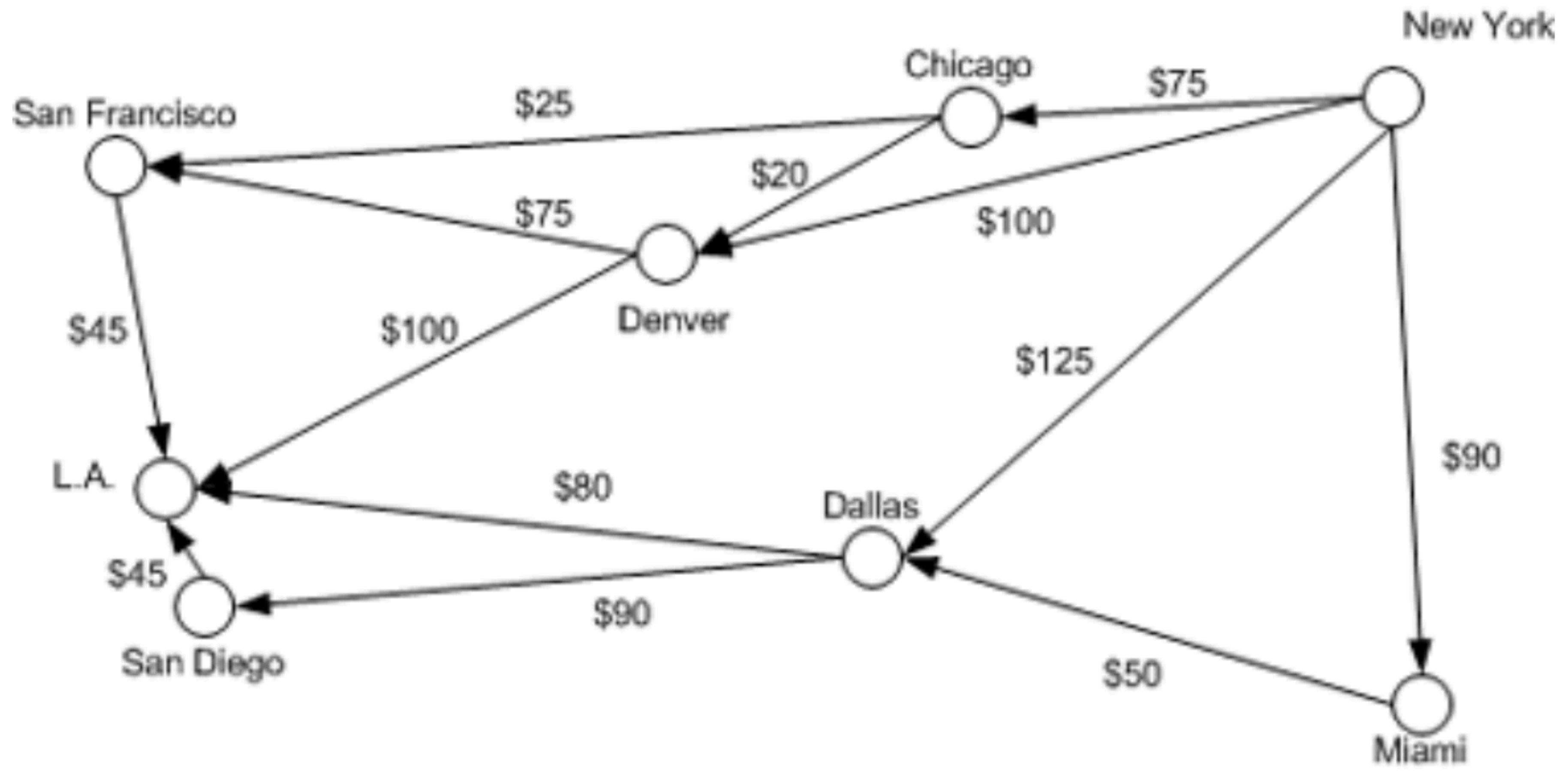
Edges can be **weighted**

- Denotes “strength”, distance, etc.

Distance between cities



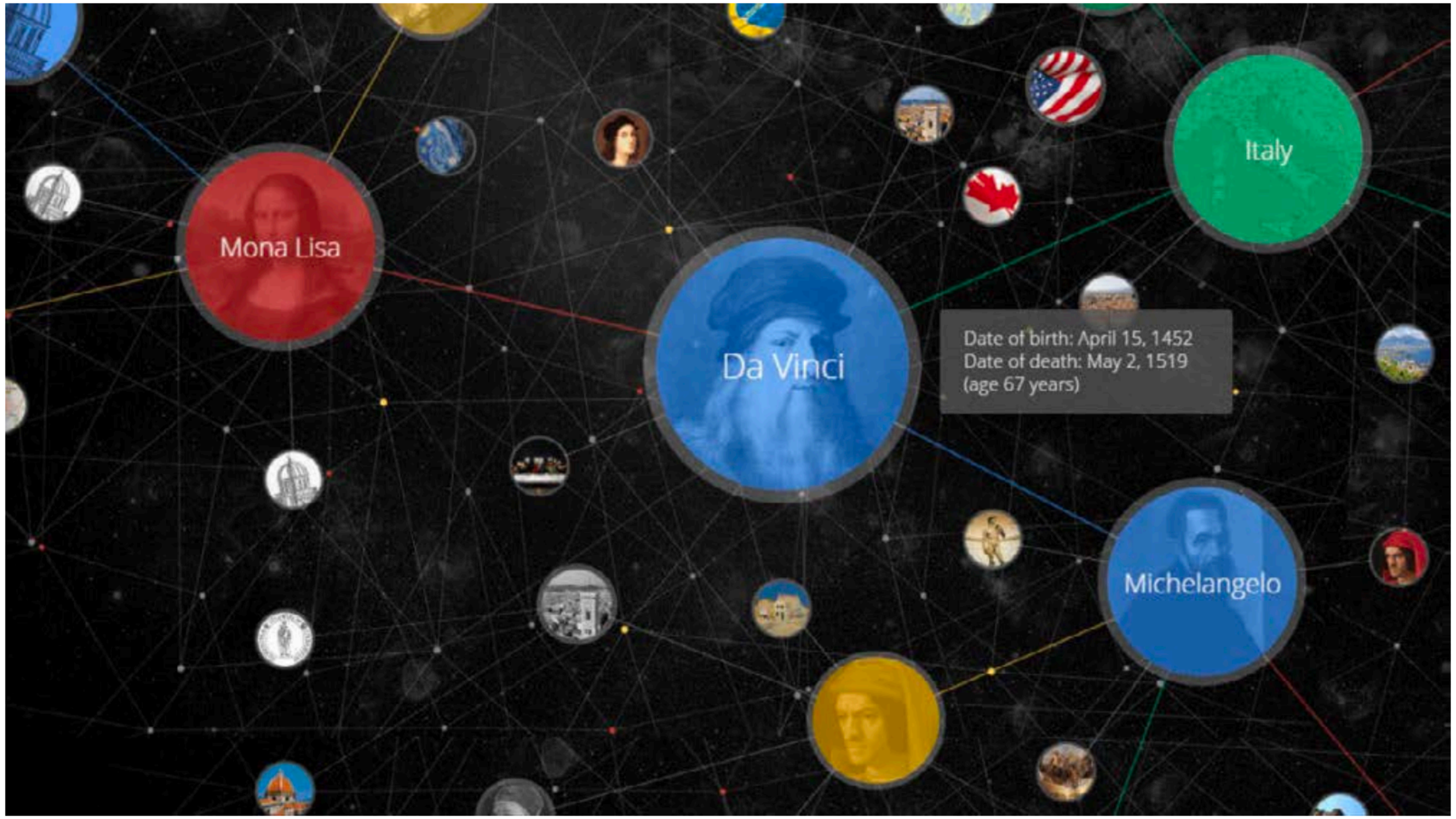
Flight costs



What is a graph?

Vertices and edges can have **types and metadata**

Google Knowledge Graph



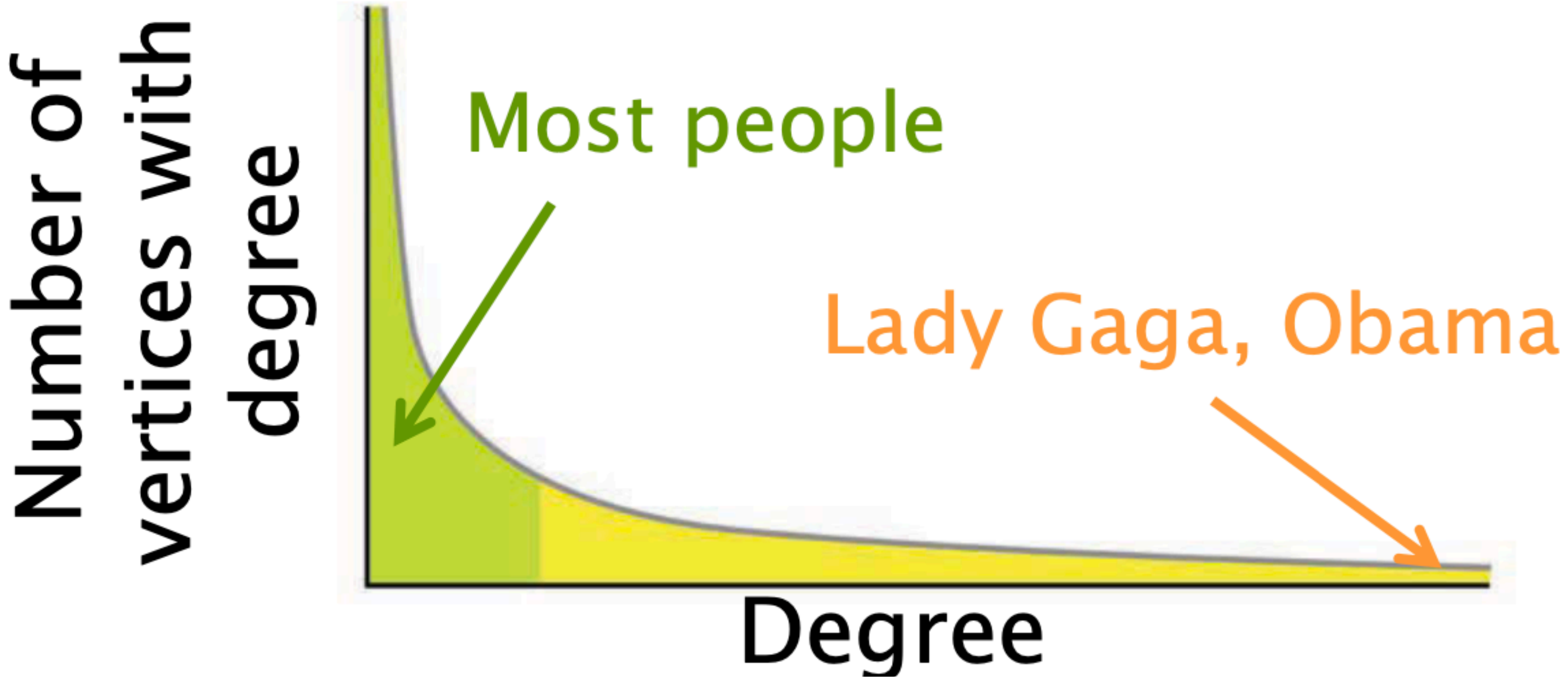
Properties of Real-World Graphs

They can be big (but not too big)

Social network	Web graph	Web graph
41 million vertices	1.4 billion vertices	3.5 billion vertices
1.5 billion edges	6.6 billion edges	128 billion edges
(6.3 GB)	(38 GB)	(540 GB)

Sparse (number of edges is much less than n^2)

Degrees can be highly **skewed**



Studies have shown that many real-world graphs have a power law degree distribution

#vertices with deg. $d \approx a \times d^{-p}$ $(2 < p < 3)$

Graph Applications

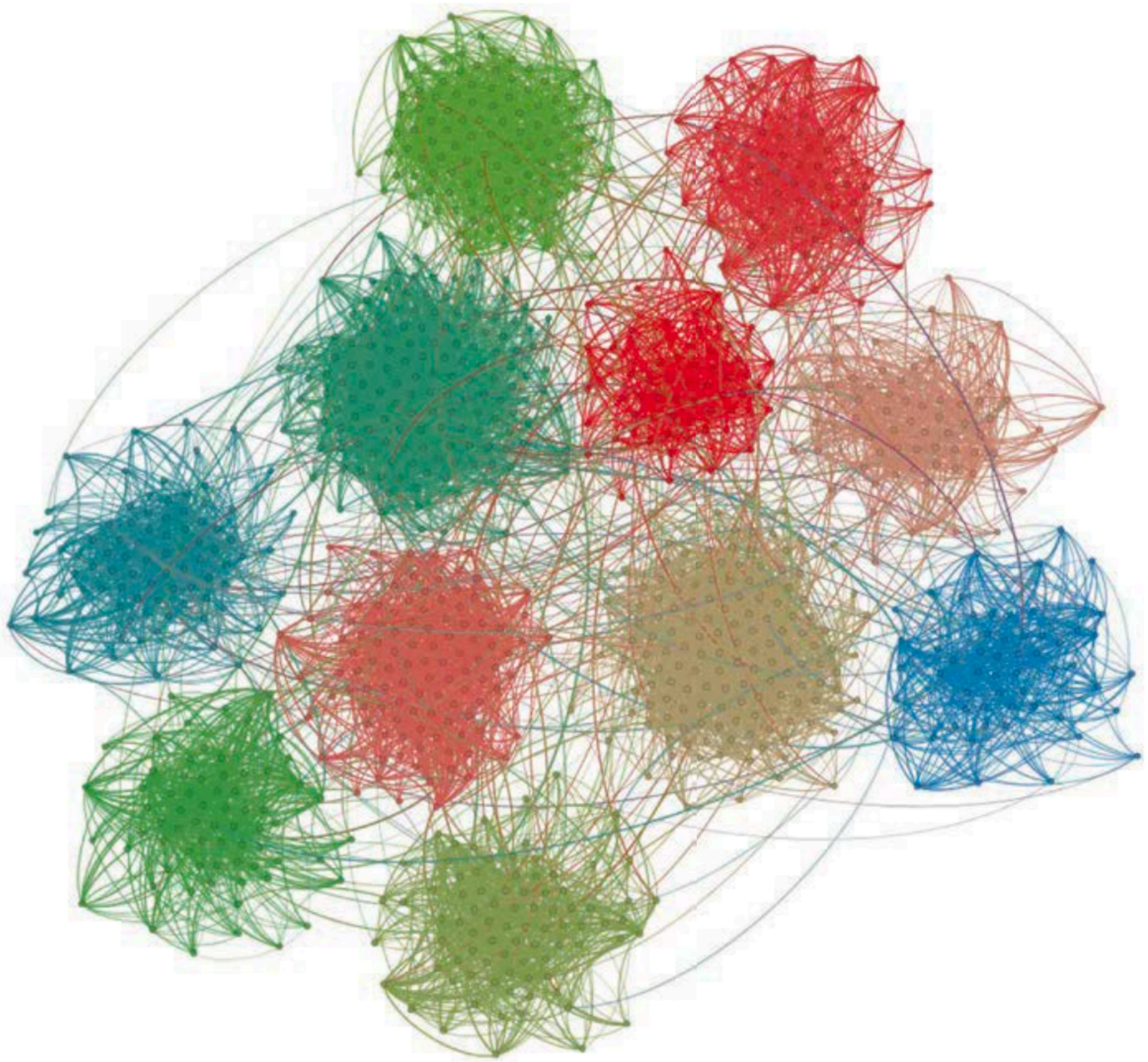
Social network queries

Examples:

- Finding all your friends who went to the same high school as you
- Finding common friends with someone
- Social networks recommending people whom you might know
- Product recommendation

Finding good clusters

Finding **groups of vertices** that are “well-connected” internally and “poorly-connected” externally



Some applications

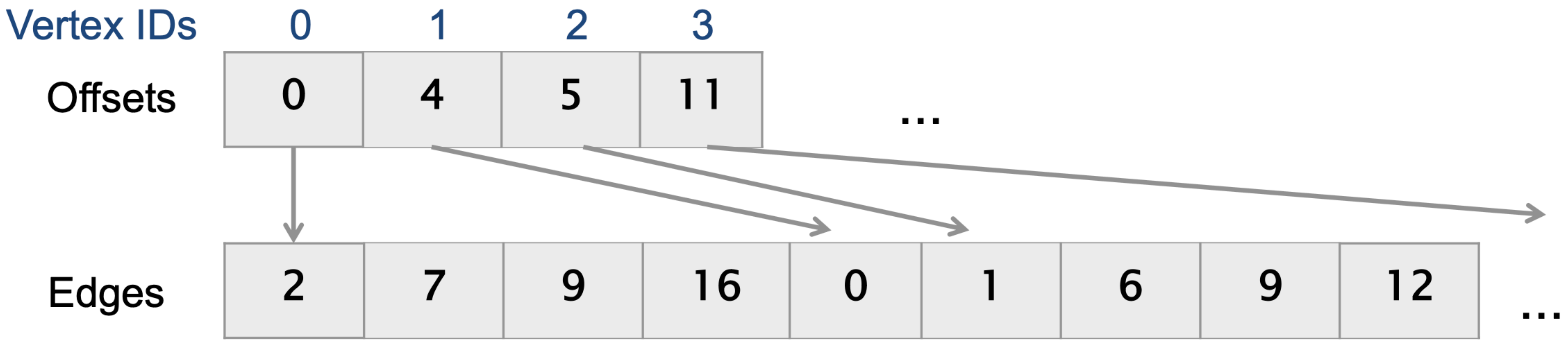
- Finding people with similar interests
- Detecting fraudulent websites
- Document clustering
- Unsupervised learning

Graph Representations
(short, we will have a full lecture on this later)

CSR is the default representation for static graphs

The algorithms we will discuss today are best implemented with **compressed sparse row** (CSR) format

- Sparse graphs
- Static algorithms-no updates to graph
- Need to scan over neighbors of a given set of vertices



From MIT 6.172

Implementing a Graph Algorithm: Breadth-First Search

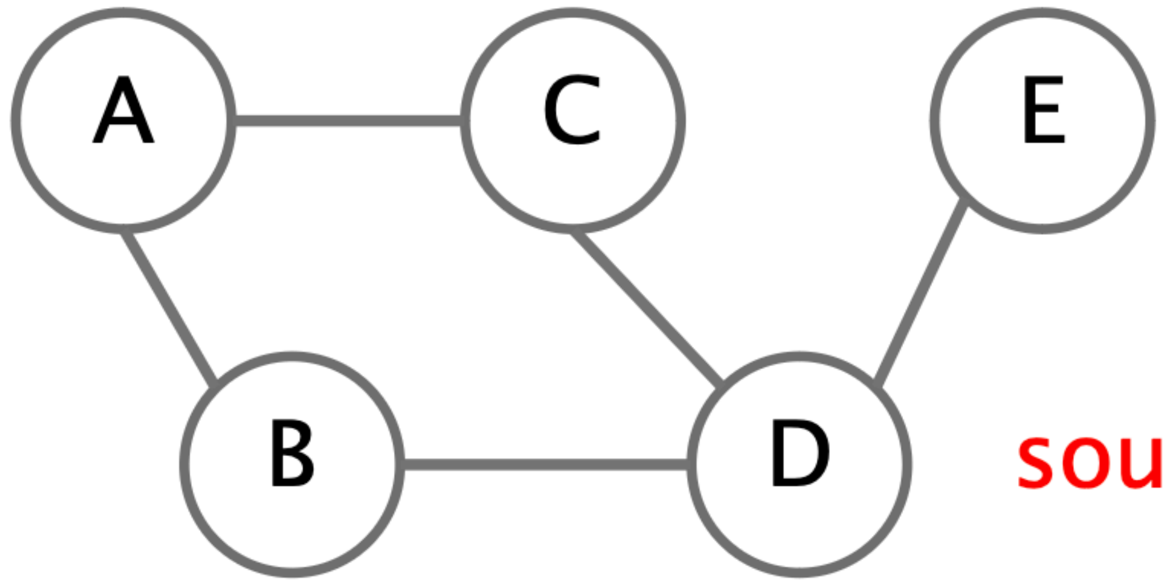
Breadth-First Search (BFS)

- Given a source vertex *s*, visit the vertices in order of distance from *s*
- Possible outputs:
  - Vertices in the order they were visited
    - D, B, C, E, A
  - The distance from each vertex to *s*

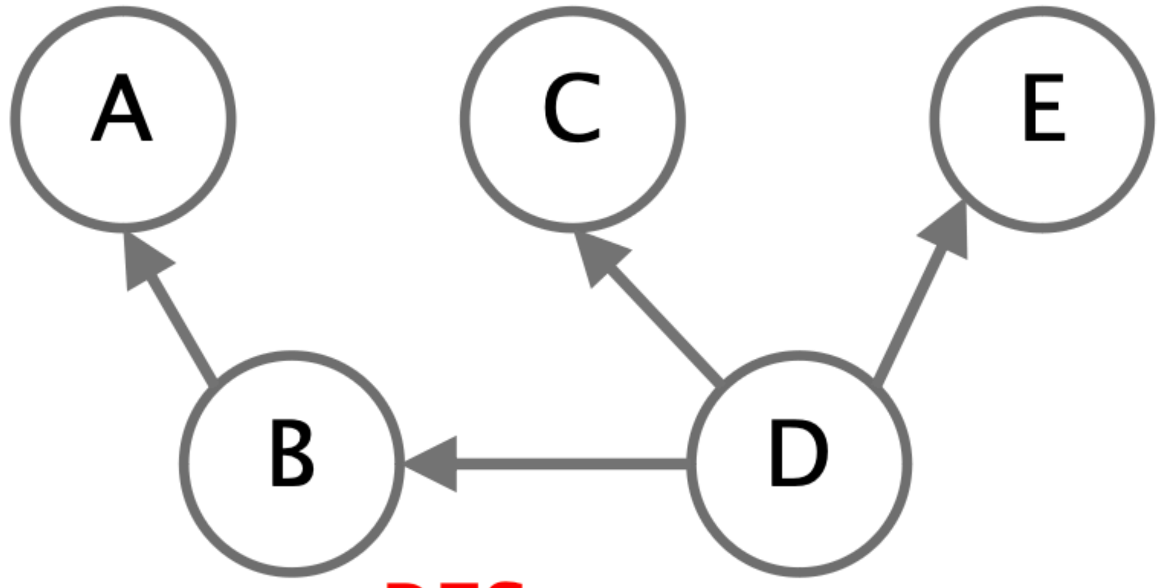
A	B	C	D	E
2	1	1	0	1

  - A BFS tree, where each vertex has a parent to a neighbor in the previous level

Applications
Betweenness centrality
Eccentricity estimation
Maximum flow
Web crawlers
Network broadcasting
Cycle detection
…



source = D



BFS tree

Serial BFS Algorithm Initialization

Suppose that we will compute the parents array (BFS tree)

Output

Nodes to visit next

Init queue with source

```
int* parent =
 (int*) malloc(sizeof(int)*n);
int* queue =
 (int*) malloc(sizeof(int)*n);

for(int i=0; i<n; i++) {
   parent[i] = -1;
}

queue[0] = source;
parent[source] = source;

int q_front = 0, q_back = 1;
```

https://en.wikipedia.org/wiki/Breadth-first_search

Serial BFS Algorithm

Assume the graph is in CSR: offsets and edges array
We have n vertices and m edges

```
//while queue not empty
while(q_front != q_back) {
    int current = queue[q_front++]; //dequeue
    int degree =
         Offsets[current+1]-Offsets[current];
    for(int i=0;i<degree; i++) {
         int ngh = Edges[Offsets[current]+i];
         //check if neighbor has been visited
         if(parent[ngh] == -1) {
             parent[ngh] = current;
             //enqueue neighbor
             queue[q_back++] = ngh;
         }
    }
}
```

Total of m random accesses

Remember: random access costs more than sequential access

What is the most expensive part of the code?

Analyzing the program

```
int* parent =
 (int*) malloc(sizeof(int)*n);
int* queue =
 (int*) malloc(sizeof(int)*n);

for(int i=0; i<n; i++) {
   parent[i] = -1;
}

queue[0] = source;
parent[source] = source;

int q_front = 0; q_back = 1;
```

```
//while queue not empty
while(q_front != q_back) {
   int current = queue[q_front++]; //dequeue
   int degree =
        Offsets[current+1]-Offsets[current];
   for(int i=0;i<degree; i++) {
        int ngh = Edges[Offsets[current]+i];
        //check if neighbor has been visited
        if(parent[ngh] == -1) {
            parent[ngh] = current;
            //enqueue neighbor
            queue[q_back++] = ngh;
        }
   }
}
```

(Approx.) analyze number of cache misses (cold cache; cache size << n; 64 byte cache line size; 4 byte int)

- n/16 for initialization
- n/16 for dequeueing
- n for accessing Offsets array
- ≤ 2n + m/16 for accessing Edges array
- m for accessing parent array
- n/16 for enqueueing

Total ≤ (51/16)n + (17/16)m

How can we reduce cache misses?

Analyzing the program

```
int* parent =
 (int*) malloc(sizeof(int)*n);
int* queue =
 (int*) malloc(sizeof(int)*n);

for(int i=0; i<n; i++) {
   parent[i] = -1;
}

queue[0] = source;
parent[source] = source;

int q_front = 0; q_back = 1;
```

```
//while queue not empty
while(q_front != q_back) {
   int current = queue[q_front++]; //dequeue
   int degree =
        Offsets[current+1]-Offsets[current];
   for(int i=0;i<degree; i++) {
        int ngh = Edges[Offsets[current]+i];
        //check if neighbor has been visited
        if(parent[ngh] == -1) {
            parent[ngh] = current;
            //enqueue neighbor
            queue[q_back++] = ngh;
        }
   }
}
```

Check bitvector first before accessing parent array

n cache misses instead of m

- What if we can fit a bitvector of size n in cache?
  - Might reduce the number of cache misses
  - More computation to do bit manipulation

BFS with bitvector

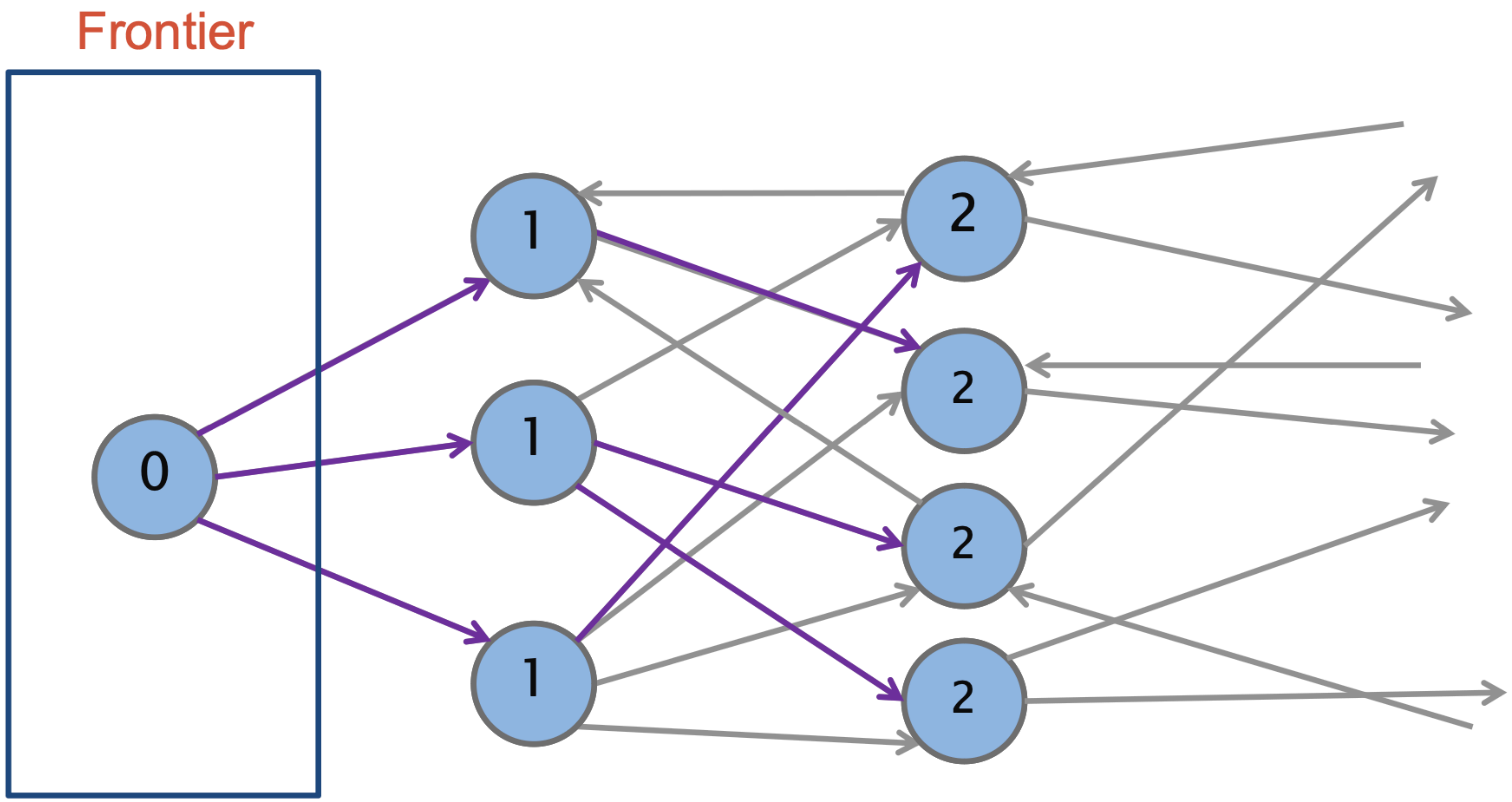
```
int* parent =
 (int*) malloc(sizeof(int)*n);
int* queue =
 (int*) malloc(sizeof(int)*n);
int nv = 1+n/32;
int* visited =
 (int*) malloc(sizeof(int)*nv);

for(int i=0; i<n; i++) {
   parent[i] = -1;
}

for(int i=0; i<nv; i++) {
   visited[i] = 0;
}

queue[0] = source;
parent[source] = source;
visited[source/32]
   = (1 << (source % 32));

int q_front = 0; q_back = 1;
```

```
//while queue not empty
while(q_front != q_back) {
   int current = queue[q_front++]; //dequeue
   int degree =
        Offsets[current+1]-Offsets[current];
   for(int i=0;i<degree; i++) {
      int ngh = Edges[Offsets[current]+i];
      //check if neighbor has been visited
      if(!((1 << ngh%32) & visited[ngh/32])){
         visited[ngh/32] |= (1 << (ngh%32));
         parent[ngh] = current;
         //enqueue neighbor
         queue[q_back++] = ngh;
      }
   }
}
```

- Bitvector version is faster for large enough values of m

Parallelizing Breadth-First Search

Parallel BFS Algorithm



- Can process each frontier in parallel
  - Parallelize over both the vertices and their outgoing edges
- Races, load balancing

Parallel BFS Code - Initialization

Instead of a queue, we have arrays for frontier, frontierNext, degrees

```
BFS(Offsets, Edges, source) {
  parent, frontier, frontierNext, and degrees are arrays
  parallel_for(int i=0; i<n; i++) parent[i] = -1;
  frontier[0] = source, frontierSize = 1, parent[source] = source;

  …
```

Parallel BFS: Overview

While the frontier is not empty:

In parallel, for all vertices v in the frontier:

Copy all neighbors of v into frontierNext (for the next iteration) - only if they have not yet been visited

Set v as the parent of all ngh(v) in the parents array - if ngh(v) does not yet have a parent in the parents array

Set frontierNext to frontier

Problem: How do we know where to copy into?

Problem: What if multiple vertices in the frontier have the same neighbor?

Parallel BFS: Overview

While the frontier is not empty:

In parallel, for all vertices v in the frontier:

Copy all neighbors of v into frontierNext (for the next iteration) - only if they have not yet been visited

Set v as the parent of all ngh(v) in the parents array - if ngh(v) does not yet have a parent in

Otherwise, do not add to frontierNext

Set frontierNext to frontier

Parallel BFS Code - Degree Setup

Problem: How do we know **where to copy the neighbors** for each vertex in the frontier to?

Answer: **Prefix sum** on the degrees

```
…

while(frontierSize > 0) {
  parallel_for(int i=0; i<frontierSize; i++)
    degrees[i] = Offsets[frontier[i]+1] – Offsets[frontier[i]];

  perform prefix sum on degrees array

  …
}
```

For all vertices in frontier, get their degrees

Exclusive scan to get starting point for each vertex

Example: Degrees:

2	4	3	1	3

Exclusive scan →

0	2	6	9	10

Parallel BFS Code

```
…

while(frontierSize > 0) {
  // SETUP DEGREES AS ON PREVIOUS SLIDE

  parallel_for(int i=0; i<frontierSize; i++) {
    v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
    for(int j=0; j<d; j++) { //can be parallel
      ngh = Edges[Offsets[v]+j];
      if(parent[ngh] == -1 && compare-and-swap(&parent[ngh], -1, v)) {
        frontierNext[index+j] = ngh;
      } else { frontierNext[index+j] = -1; }
    }
  }
  filter out "-1" from frontierNext, store in frontier, and update
  frontierSize to be the size of frontier (all done using prefix sum)
}
```

Iterate over vertices in frontier

Parallel BFS Code

```
…

while(frontierSize > 0) {
  // SETUP DEGREES AS ON PREVIOUS SLIDE

  parallel_for(int i=0; i<frontierSize; i++) {
    v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
    for(int j=0; j<d; j++) { //can be parallel
      ngh = Edges[Offsets[v]+j];
      if(parent[ngh] == -1 && compare-and-swap(&parent[ngh], -1, v)) {
        frontierNext[index+j] = ngh;
      } else { frontierNext[index+j] = -1; }
    }
  }
  filter out "-1" from frontierNext, store in frontier, and update
  frontierSize to be the size of frontier (all done using prefix sum)
}
```

Iterate over vertices in frontier

Copy in using starting points computed previously

Parallel BFS Code

```
…

while(frontierSize > 0) {
  // SETUP DEGREES AS ON PREVIOUS SLIDE

  parallel_for(int i=0; i<frontierSize; i++) {
    v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
    for(int j=0; j<d; j++) { //can be parallel
      ngh = Edges[Offsets[v]+j];
      if(parent[ngh] == -1 && compare-and-swap(&parent[ngh], -1, v)) {
        frontierNext[index+j] = ngh;
      } else { frontierNext[index+j] = -1; }
    }
  }
  filter out "-1" from frontierNext, store in frontier, and update
  frontierSize to be the size of frontier (all done using prefix sum)
}
```

Iterate over vertices in frontier

Copy in using starting points computed previously

If this neighbor hasn't been explored yet

Parallel BFS Code

```
…

while(frontierSize > 0) {
  // SETUP DEGREES AS ON PREVIOUS SLIDE

  parallel_for(int i=0; i<frontierSize; i++) {
    v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
    for(int j=0; j<d; j++) { //can be parallel
      ngh = Edges[Offsets[v]+j];
      if(parent[ngh] == -1 && compare-and-swap(&parent[ngh], -1, v)) {
        frontierNext[index+j] = ngh;
      } else { frontierNext[index+j] = -1; }
    }
  }
  filter out "-1" from frontierNext, store in frontier, and update
  frontierSize to be the size of frontier (all done using prefix sum)
}
```

Iterate over vertices in frontier

Copy in using starting points computed previously

If this neighbor hasn't been explored yet

Other vertices in the frontier may also have ngh as their neighbor. Only one should add it.

Parallel BFS Code

```
…

while(frontierSize > 0) {
  // SETUP DEGREES AS ON PREVIOUS SLIDE

  parallel_for(int i=0; i<frontierSize; i++) {
    v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
    for(int j=0; j<d; j++) { //can be parallel
      ngh = Edges[Offsets[v]+j];
      if(parent[ngh] == -1 && compare-and-swap(&parent[ngh], -1, v)) {
        frontierNext[index+j] = ngh;
      } else { frontierNext[index+j] = -1; }
    }
  }
  filter out "-1" from frontierNext, store in frontier, and update
  frontierSize to be the size of frontier (all done using prefix sum)
}
```

Iterate over vertices in frontier

Copy in using starting points computed previously

If this neighbor hasn't been explored yet

Other vertices in the frontier may also have ngh as their neighbor. Only one should add it.

Otherwise, do not add to frontierNext

Parallel BFS Code

```
…
while(frontierSize > 0) {
  // SETUP DEGREES AS ON PREVIOUS SLIDE

  parallel_for(int i=0; i<frontierSize; i++) {
    v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
    for(int j=0; j<d; j++) { //can be parallel
      ngh = Edges[Offsets[v]+j];
      if(parent[ngh] == -1 && compare-and-swap(&parent[ngh], -1, v)) {
        frontierNext[index+j] = ngh;
      } else { frontierNext[index+j] = -1; }
    }
  }
  filter out "-1" from frontierNext, store in frontier, and update
  frontierSize to be the size of frontier (all done using prefix sum)
}
```

Iterate over vertices in frontier

Copy in using starting points computed previously

If this neighbor hasn't been explored yet

Other vertices in the frontier may also have ngh as their neighbor. Only one should add it.

Otherwise, do not add to frontierNext

Question: How would you do this?

Filter: Filling in next frontier with prefix sum

Problem: We have frontierNext, which has some -1 (empty) and some valid vertices (>=0). How do we pack them to the front of frontierNext?

Answer: Parallel filter with prefix sum

Example:

frontierNext:	-1	4	8	-1	-1	2	1	-1	9	-1
flags:	0	1	1	0	0	1	1	0	1	0
exclusive_scan(flags):	0	0	1	1	1	2	3	3	4	4

Pink values are dest locations of vertices in frontier

```
parallel_for i from 0 to len(frontierNext):
  if flags[i] == 1:
    frontier[result_of_flag_scan[i]] = frontierNext[i]
```

Compare and swap

Compare-and-swap (CAS) is an **atomic** instruction that compares the contents of a memory location with a given (old) value and, only if they are the same, modifies the contents of the location to a new given value.

CAS is used to implemented mutexes, as well as lock-free and wait-free algorithms.

```
function cas(p: pointer to int, old: int, new: int)
    if *p ≠ old
        return false

    *p ← new

    return true
```

BFS Span Analysis

Longest path in graph

Number of iterations <= **diameter** D of graph

Each iteration takes $\Theta(\log(m))$ span for parallel for loops, prefix sum, and filter (assuming inner loop is parallelized)

Span = $\Theta(D\log(m))$

BFS Work Analysis

Sum of frontier sizes = n

Each edge traversed once -> m total visits

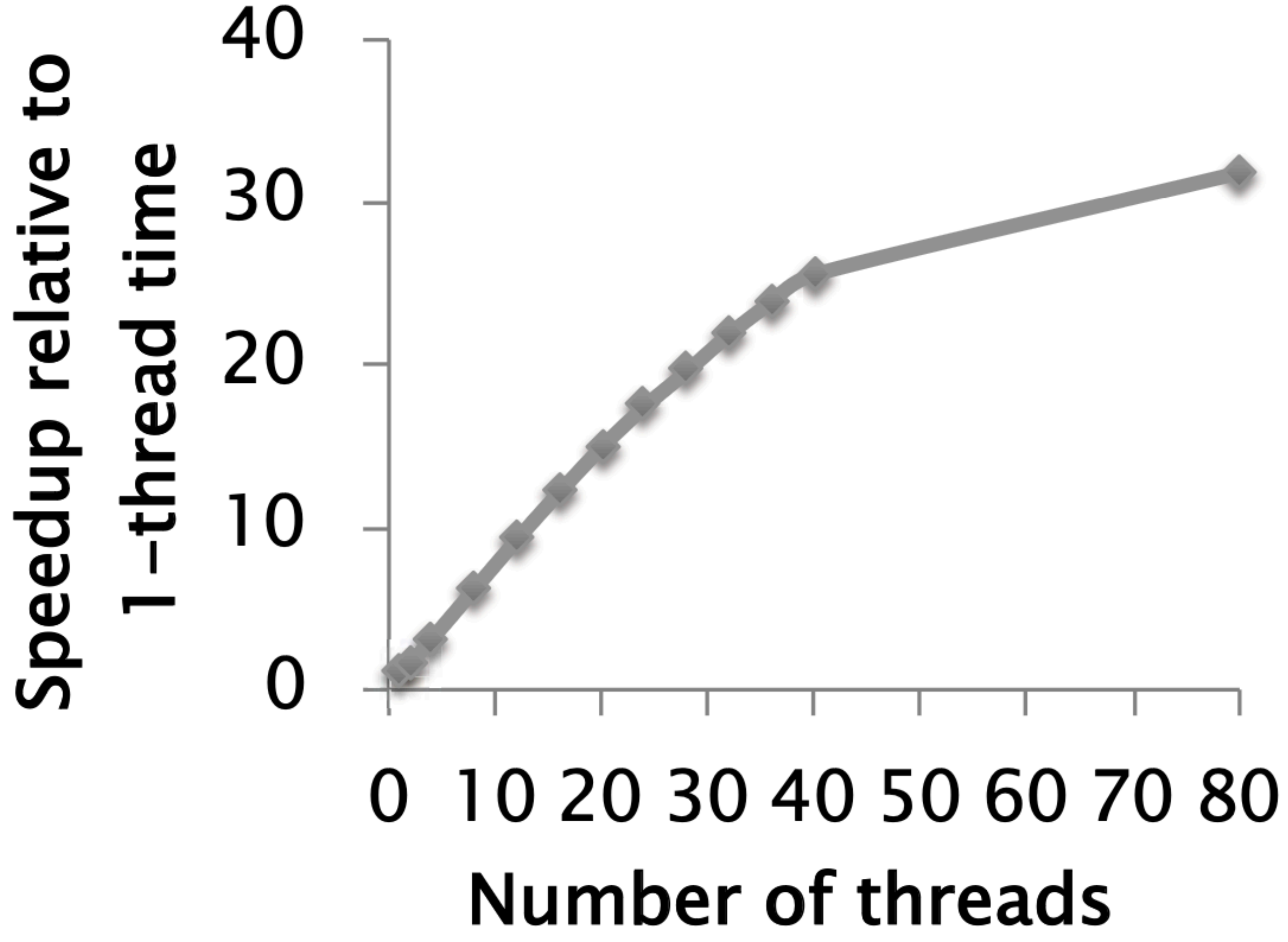
Work of prefix sum on each iteration is proportional to frontier size -> $\Theta(n)$ total

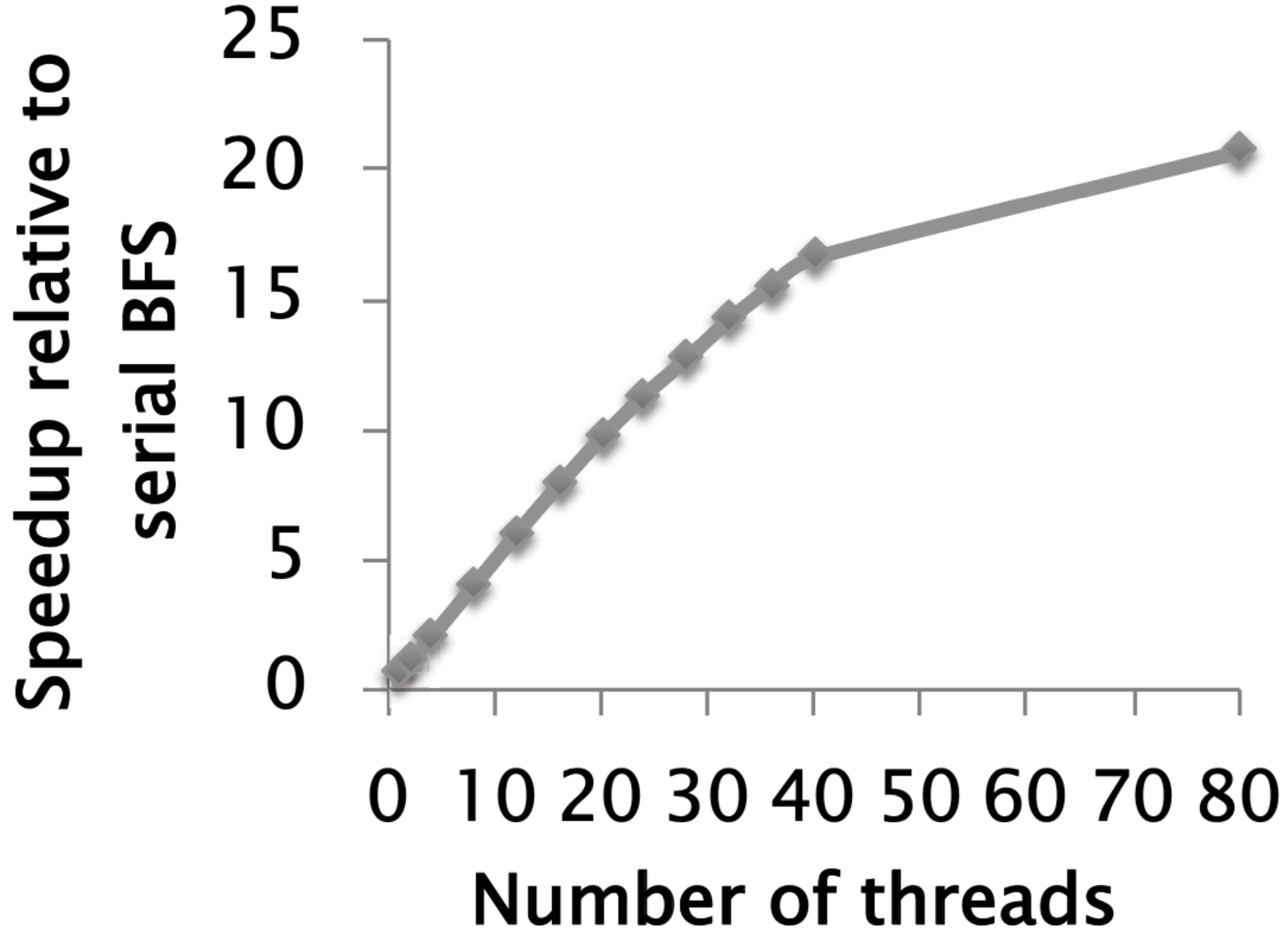
Work of filter on each iteration is proportional to number of edges traversed -> $\Theta(m)$ total

Work = $\Theta(m + n)$

Performance of Parallel BFS

- Random graph with $n=10^7$ and $m=10^8$
  - 10 edges per vertex
- 40-core machine with 2-way hyperthreading





- 31.8x speedup on 40 cores with hyperthreading
- Serial BFS is 54% faster than parallel BFS on 1 thread

Dealing with nondeterminism

```
…

while(frontierSize > 0) {
  // SETUP DEGREES AS ON PREVIOUS SLIDE

  parallel_for(int i=0; i<frontierSize; i++) {
    v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
    for(int j=0; j<d; j++) { //can be parallel
      ngh = Edges[Offsets[v]+j];
      if(parent[ngh] == -1 && compare-and-swap(&parent[ngh], -1, v)) {
        frontierNext[index+j] = ngh;
      } else { frontierNext[index+j] = -1; }
    }
  }
  …
}
```

Nondeterministic

Nondeterministic parallel programs are hard to debug. Can we substitute a **deterministic alternative**?

Deterministic Parallel BFS

```
writeMin(addr, newval):
  oldval = *addr
  while(newval < oldval):
    if(CAS(addr, oldval, newval)) return
    else: oldval = addr*
```

```
parallel(int i=0; i<frontierSize; i++) { //phase 1
  v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
  for(int j=0; j<d; j++) { //can be parallel
    ngh = Edges[Offsets[v]+j];
    writeMin(&parent[ngh], v); }
}
parallel_for(int i=0; i<frontierSize; i++) { //phase 2
  v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
  for(int j=0; j<d; j++) { //can be parallel
    ngh = Edges[Offsets[v]+j];
    if(parent[ngh] == v) {
      parent[ngh] = -v; //to avoid revisiting
      frontierNext[index+j] = ngh; }
    else { frontierNext[index+j] = -1; }}
  }
  filter out "-1" from frontierNext, store in frontier, and update frontierSize
}}
```

Smallest value gets written

Check if v "won"

Deterministic Parallel BFS

```
writeMin(addr, newval):
  oldval = *addr
  while(newval < oldval):
    if(CAS(addr, oldval, newval)) return
    else: oldval = addr*
```

```
parallel(int i=0; i<frontierSize; i++) { //phase 1
  v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
  for(int j=0; j<d; j++) { //can be parallel
    ngh = Edges[Offsets[v]+j];
    writeMin(&parent[ngh], v); }
}
parallel_for(int i=0; i<frontierSize; i++) { //phase 2
  v = frontier[i], index = degrees[i], d = Offsets[v+1]-Offsets[v];
  for(int j=0; j<d; j++) { //can be parallel
    ngh = Edges[Offsets[v]+j];
    if(parent[ngh] == v) {
      parent[ngh] = -v; //to avoid revisiting
      frontierNext[index+j] = ngh; }
    else { frontierNext[index+j] = -1; }}
  }
  filter out "-1" from frontierNext, store in
}}
```

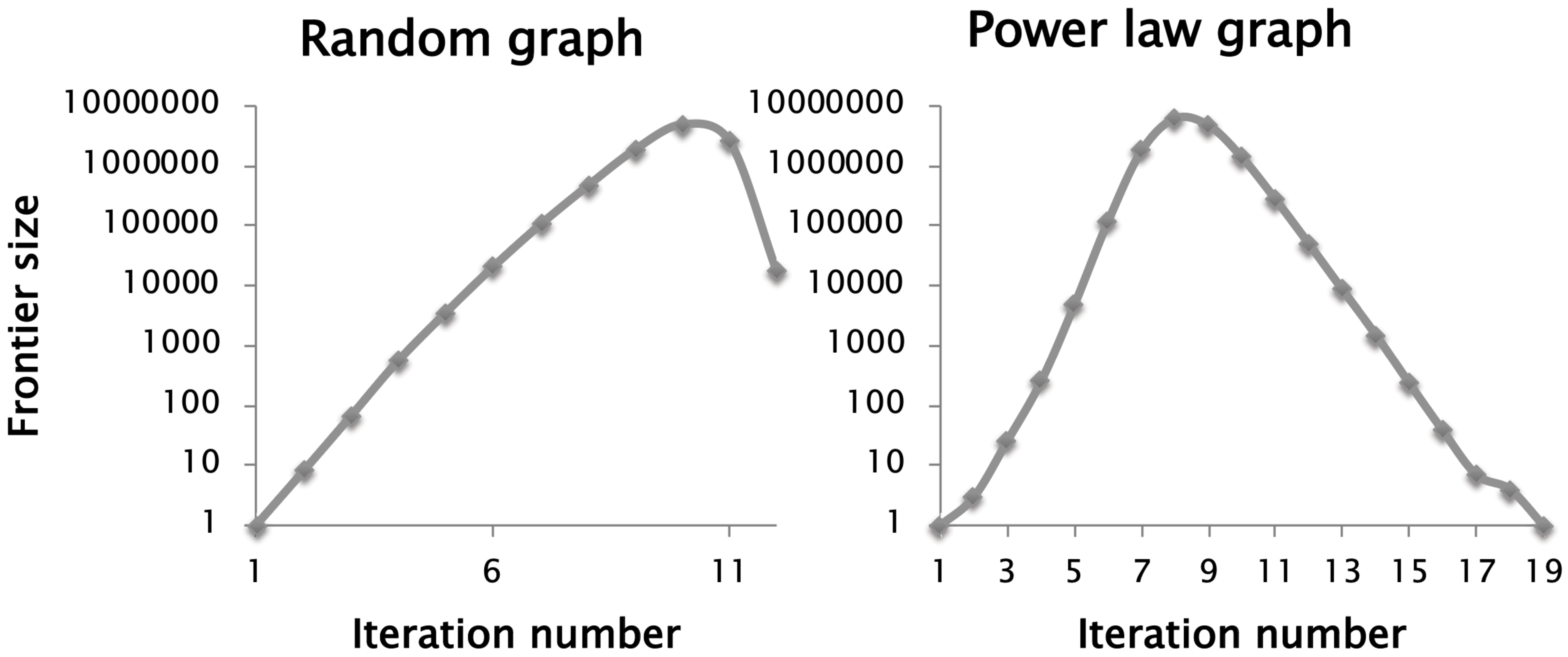
Smallest value gets written

Check if v "won"

On 32 cores, (an optimized version of) deterministic BFS is 5-20% slower than nondeterministic BFS

Direction-Optimizing Breadth-First Search

Growth of Frontiers



- For many graphs, frontier grows rapidly and then shrinks
- Most of the work done with frontier (and sum of out-degrees) is large

Top-Down BFS

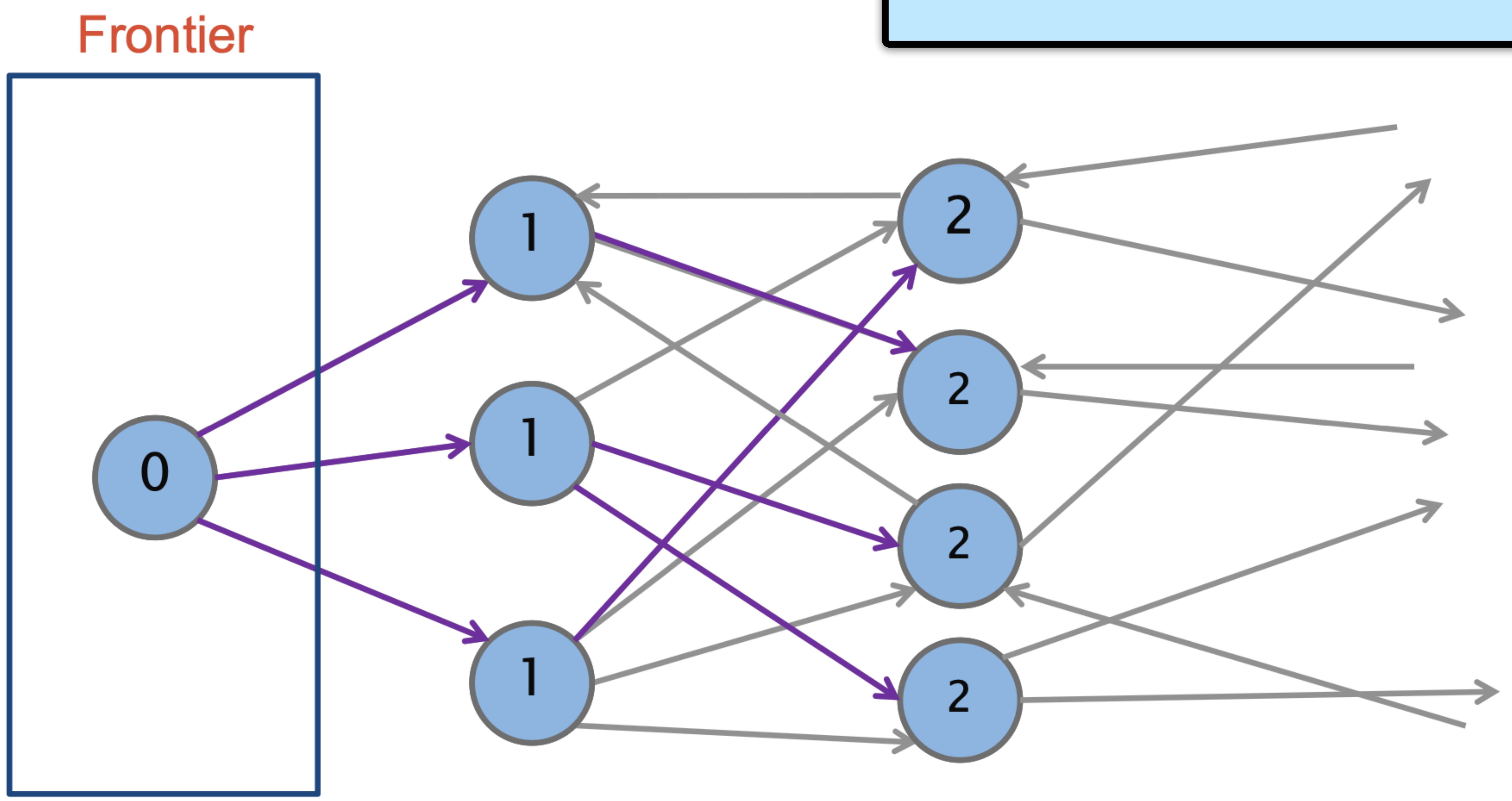
Most of the work is checking if the endpoint has been visited.

Loop through frontier vertices and explore unvisited neighbors

Efficient for small frontiers

Updates to parent array is **atomic**

If the frontier is large, there are many wasted attempts because only one can update the parents array



Bottom-Up BFS

```
for all vertices v in parallel:
  if parent[v] == -1:
    for all neighbors ngh of v:
      if ngh on frontier:
        parent[v] = ngh;
        place v on frontierNext;
        break;
```

Iterate over all vertices

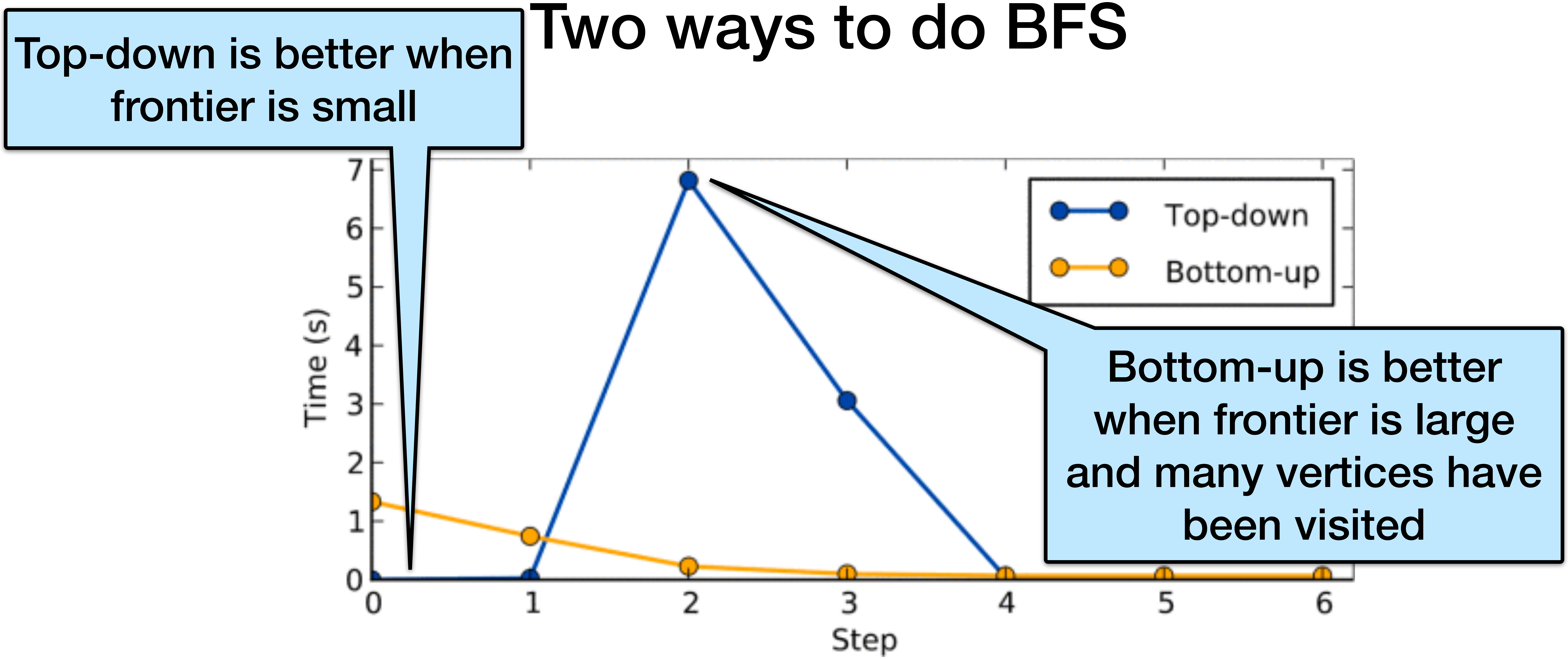
If vertex has not been visited

If ngh is on the frontier, set it as v's parent and put v on the next frontier

Efficient for **large frontiers**

Update to parent array **need not be atomic**

Two ways to do BFS



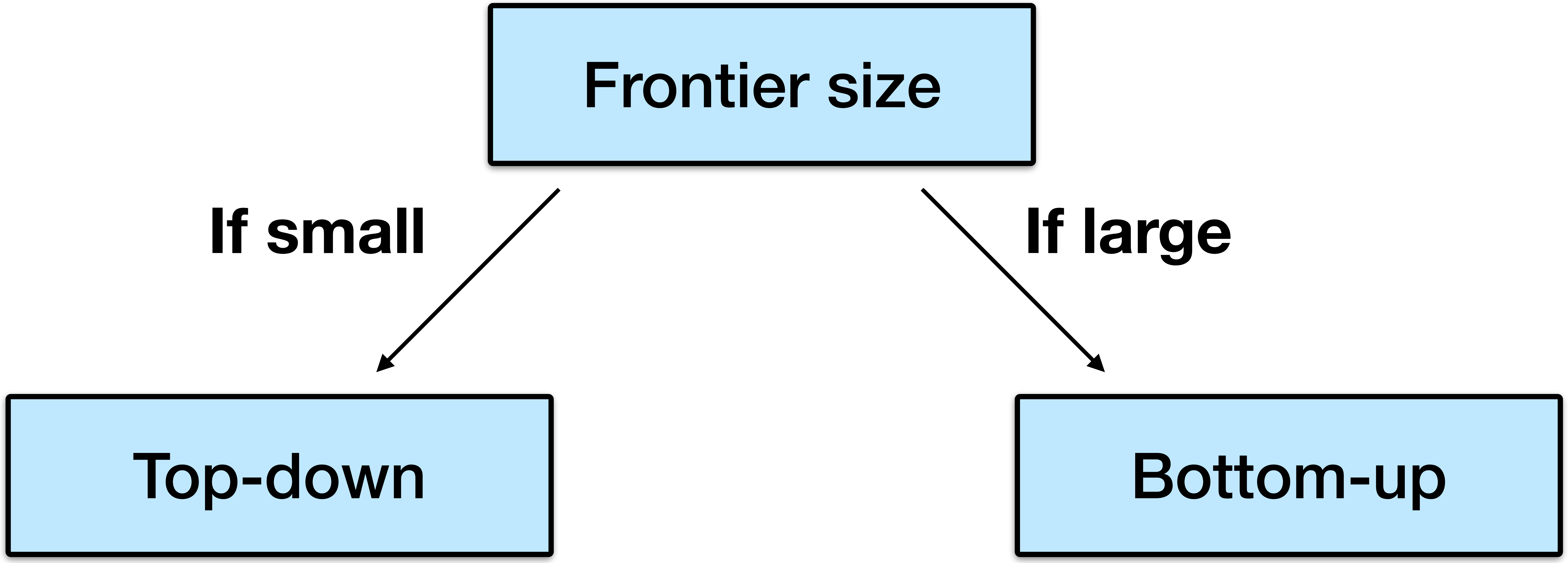
Sample search on kron27 (Kronecker 128M vertices with 2B undirected edges) on a 16-core system.

Which variant (top-down or bottom-up) to use?

From MIT 6.172 "Direction-optimizing Breadth-First Search," Beamer, Asanovic, and Patterson. Supercomputing 2012.

Direction-optimizing BFS

Idea: Choose **based on frontier size** (Beamer, Asanovic, and Patterson in SC 2012)



Threshold of frontier size > n/20 works well in practice

- Can also consider sum of out-degrees

"Direction-optimizing Breadth-First Search," Beamer, Asanovic, and Patterson. Supercomputing 2012.

Representing the frontier

Sparse integer array

- For example, [1, 4, 7]

> Used for top-down

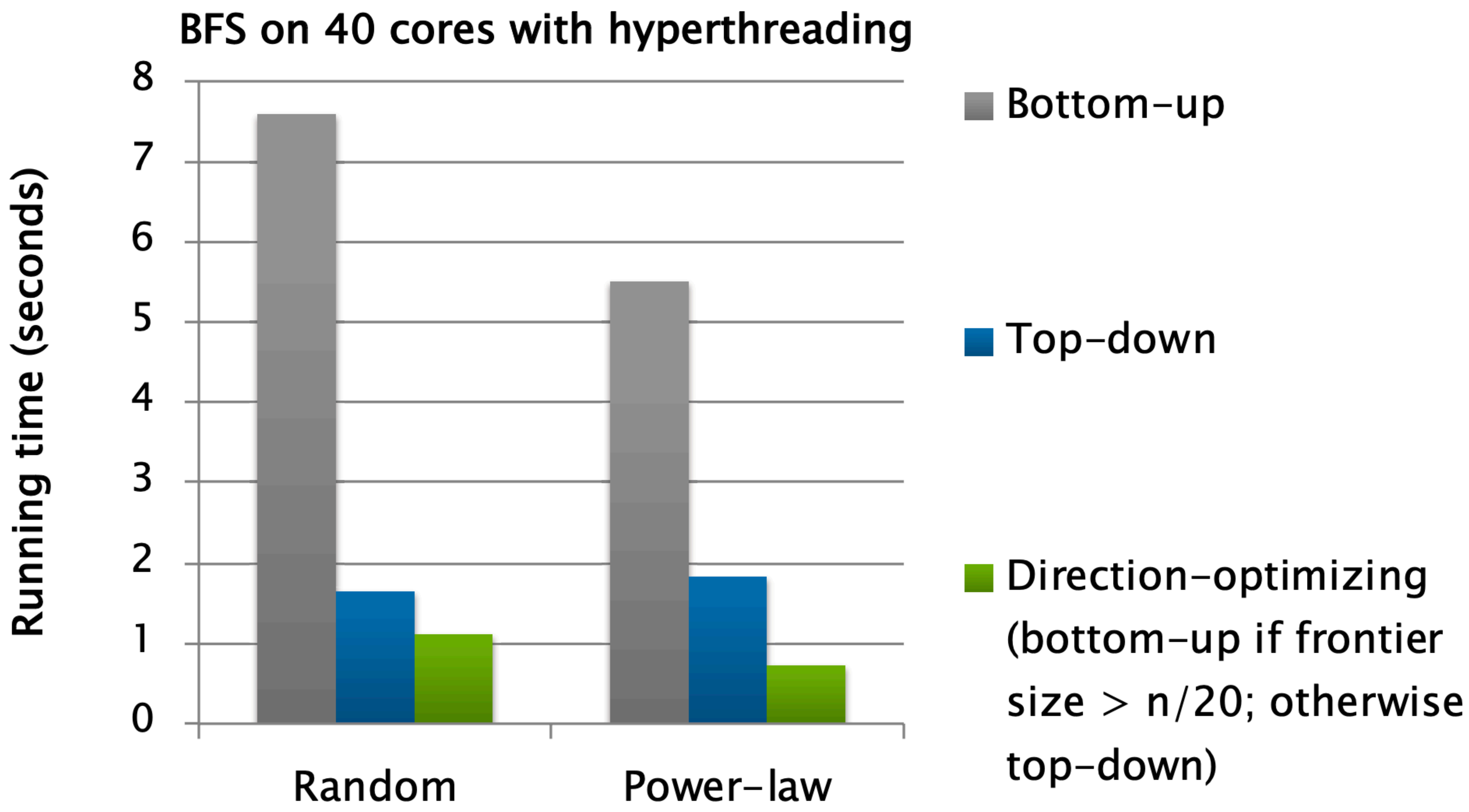
Dense byte array

- For example, [0, 1, 0, 0, 1, 0, 0, 1] (n = 8)
- Can further compress this by using 1 bit per vertex and using bit-level operations to access it

> Used for bottom-up

Need to **convert between representations** when switching methods

Direction-Optimizing BFS Performance



- Benefits highly dependent on graph
- No benefits if frontier is always small (e.g., on a grid graph or road network)

Ligra Graph Framework

Update function for vertex

Condition to add to next frontier

```
procedure EDGEMAP(G, frontier, Update, Cond):
  if(size(frontier) + sum of out-degrees > threshold) then:
    return EDGEMAP_DENSE(G, frontier, Update, Cond);
  else:
    return EDGEMAP_SPARSE(G, frontier, Update, Cond);
```

More general than BFS!

Ligra framework generalizes direction optimization to many other problems

- e.g., betweenness centrality, connected components, sparse PageRank, shortest paths, eccentricity estimation, graph clustering, k-core decomposition, set cover, etc.

Julian Shun and Guy Blelloch. "Ligra : A Lightweight Graph Processing Framework for Shared Memory," PPoPP 2013

Ligra Example - BFS

Update function for vertex

Condition to add to next frontier

```
procedure EDGEMAP(G, frontier, Update, Cond):
  if(size(frontier) + sum of out-degrees > threshold) then:
    return EDGEMAP_DENSE(G, frontier, Update, Cond);
  else:
    return EDGEMAP_SPARSE(G, frontier, Update, Cond);
```

```
bool Update(int s, int d) {
  if(parents[d] == -1) {
    parents[d] = s; return 1;
  }
  else return 0;
}
```

if unvisited, set parents

otherwise, just return false

```
bool cond(int d) {
  return (parents[d] == -1);
}
```

unvisited

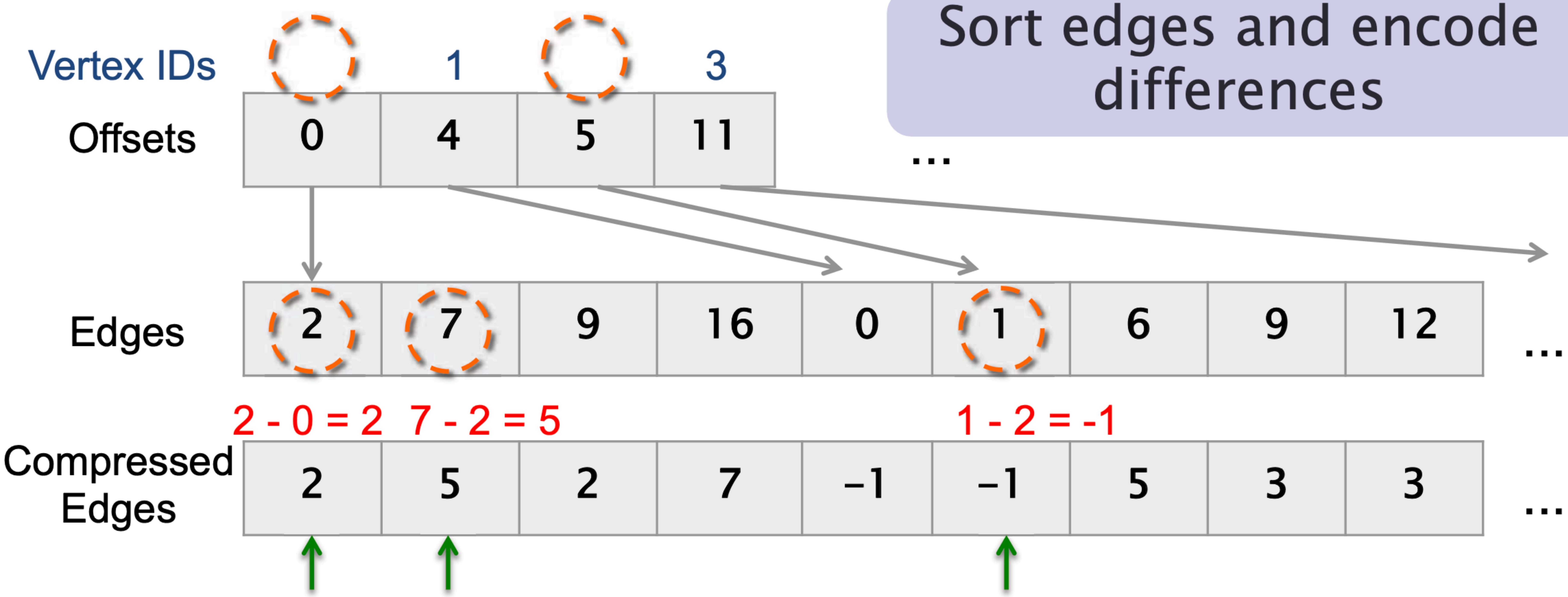
https://github.com/jshun/ligra/blob/master/apps/BFS.C

Julian Shun and Guy Blelloch. "Ligra : A Lightweight Graph Processing Framework for Shared Memory," PPoPP 2013

Graph Compression and Reordering

Graph Compression on CSR

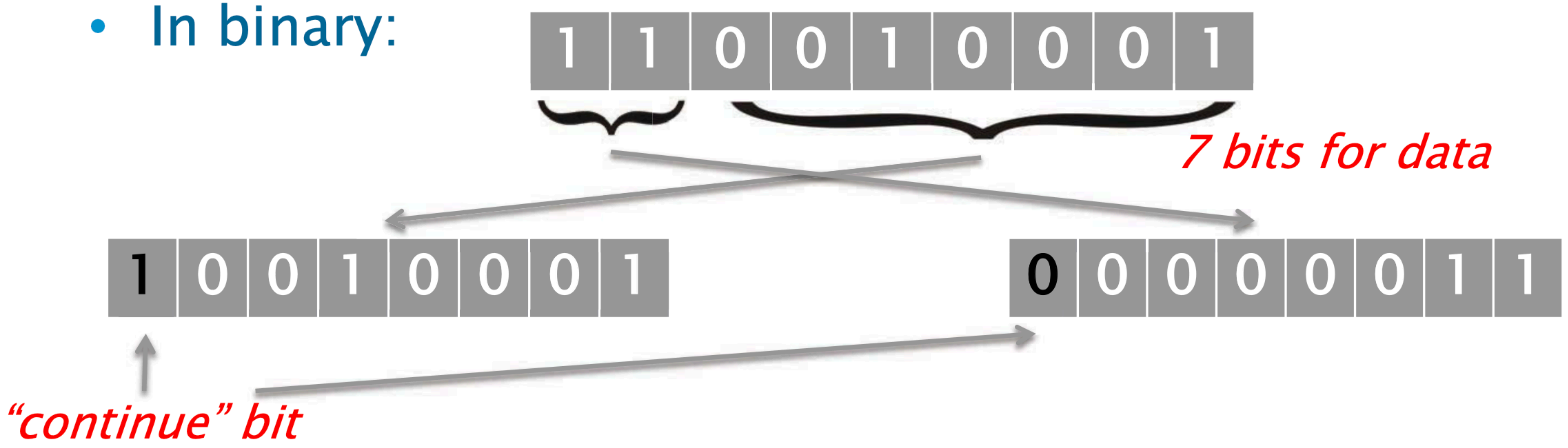
Sort edges and encode differences



- For each vertex v:
  - First edge: difference is Edges[Offsets[v]]−v
  - i'th edge (i>1): difference is Edges[Offsets[v]+i]−Edges[Offsets[v]+i−1]
- Want to use fewer than 32 or 64 bits to store each value

Variable-length codes

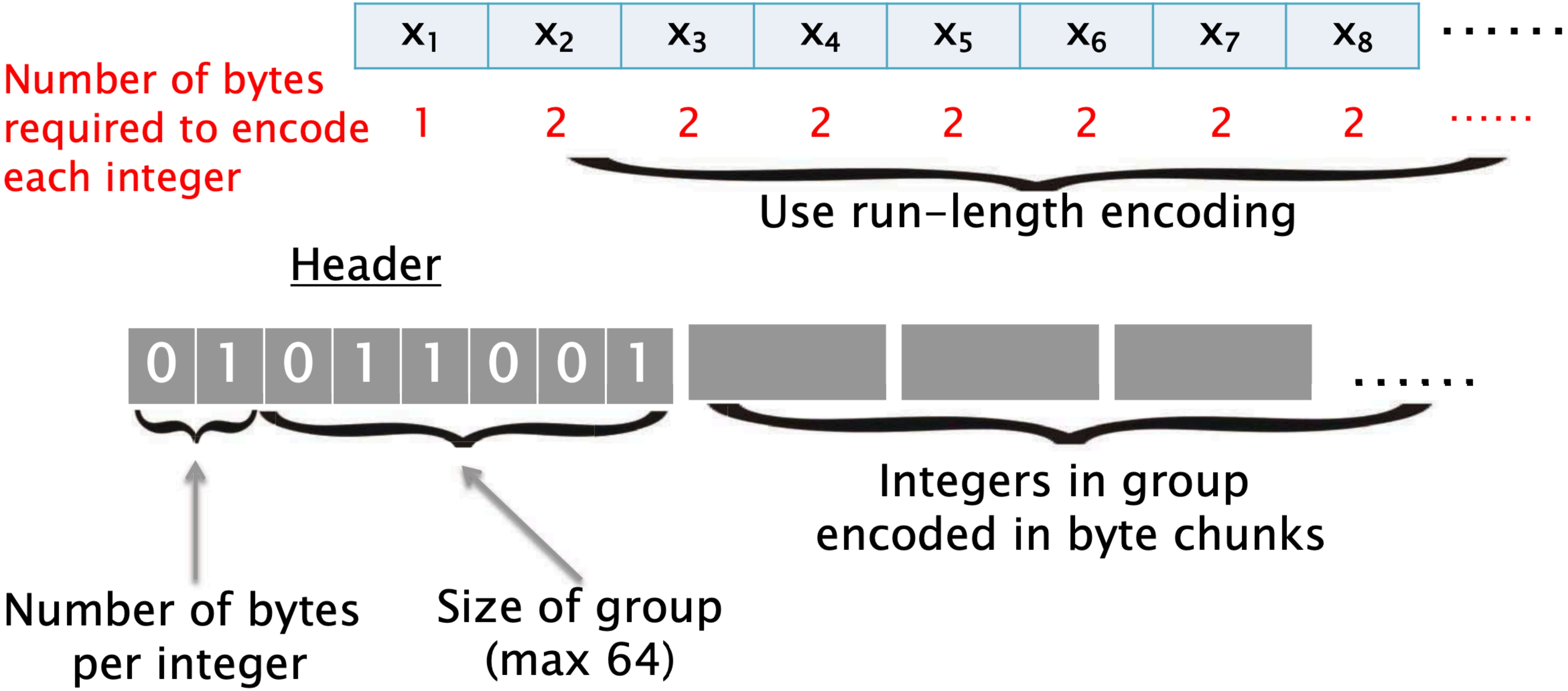
- k-bit (variable-length) codes
  - Encode value in chunks of k bits
  - Use k–1 bits for data, and 1 bit as the "continue" bit
- Example: encode "401" using 8-bit (byte) codes
- In binary:



- Decoding is just encoding "backwards"
  - Read chunks until finding a chunk with a "0" continue bit
  - Shift data values left accordingly and sum together
- Branch mispredictions from checking continue bit

Encoding optimization

- Another idea: get rid of "continue" bits

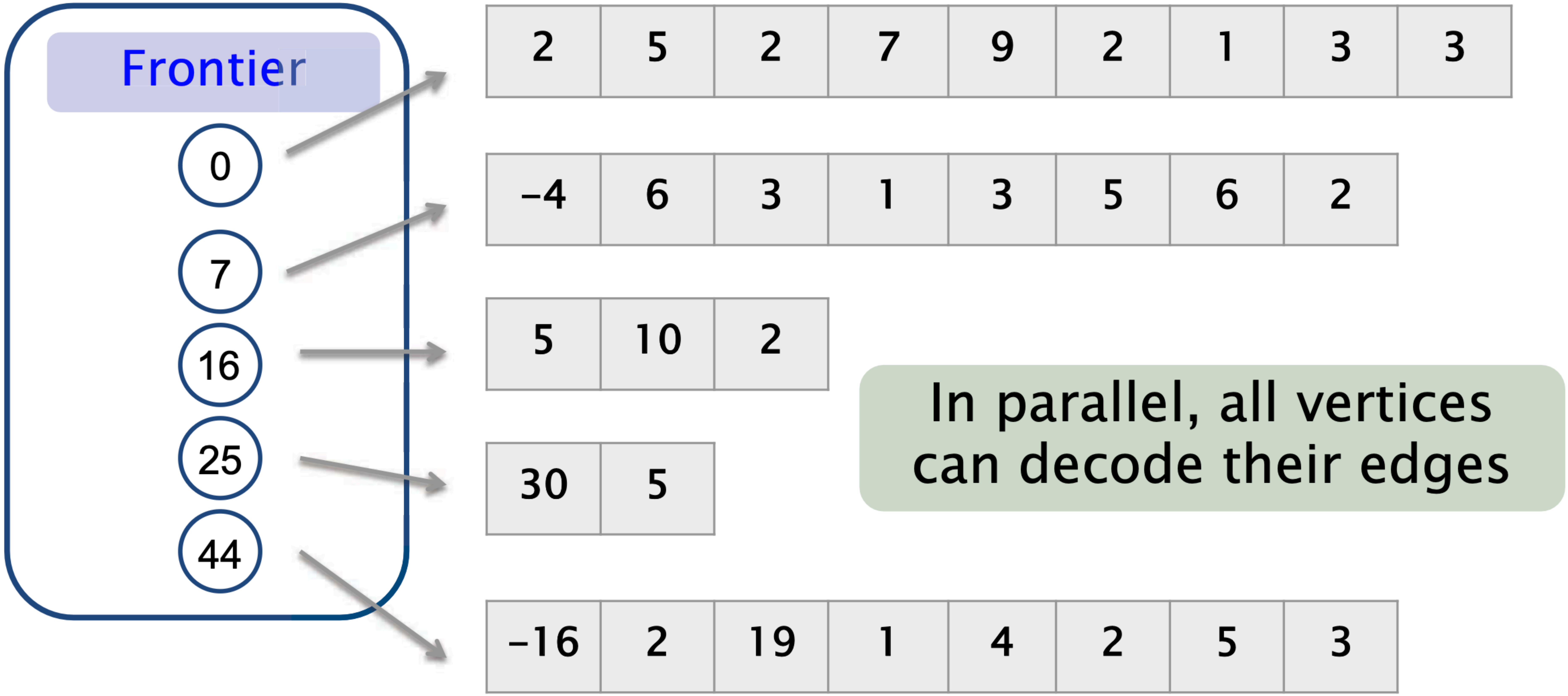


- Increases space, but makes decoding cheaper (no branch misprediction from checking "continue" bit)

Julian Shun, Laxman Dhulipala and Guy Blelloch. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra +, DCC 2015

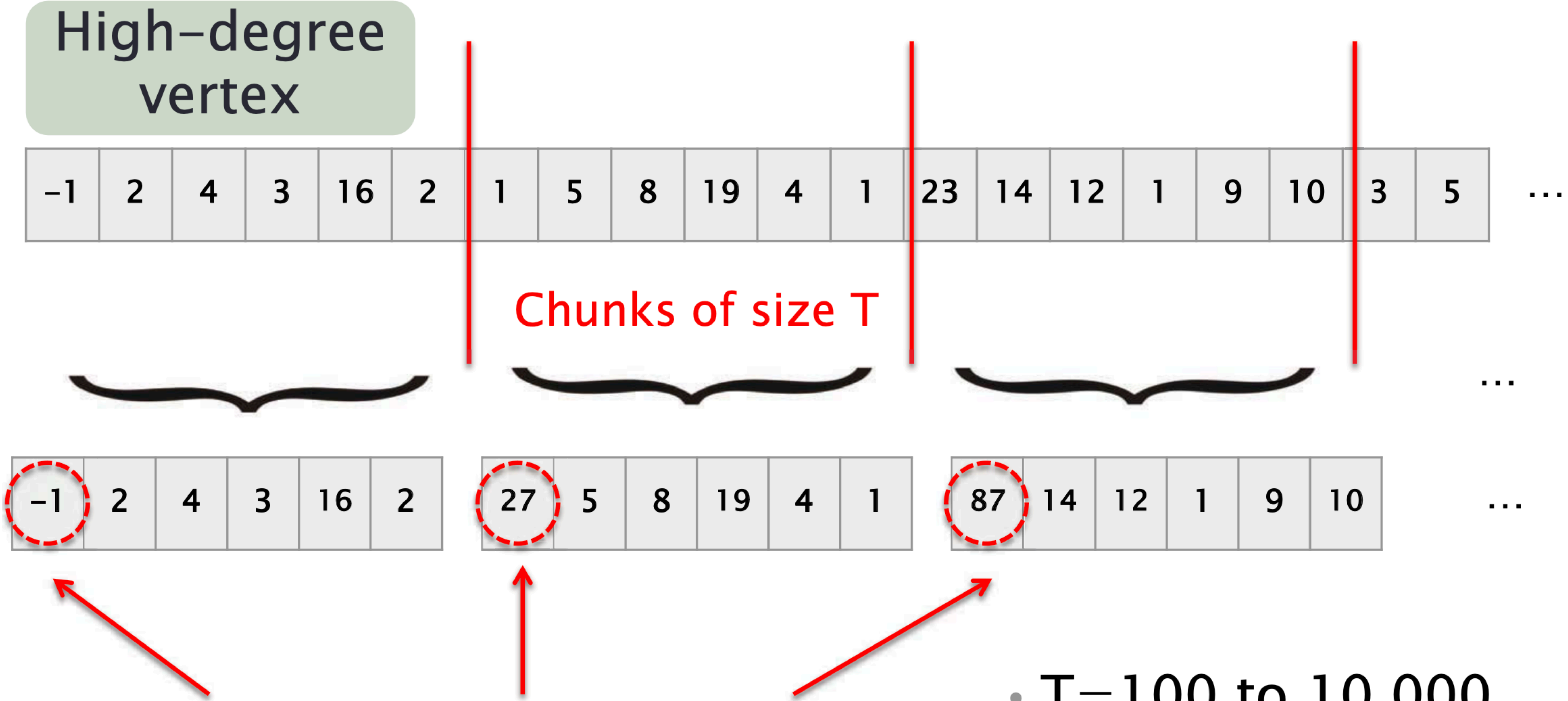
Decoding on-the-fly

- Need to decode during the algorithm
  - If we decoded everything at the beginning we would not save any space!



- Each vertex decodes its edges sequentially
  - What about high degree vertices?

Parallel decoding



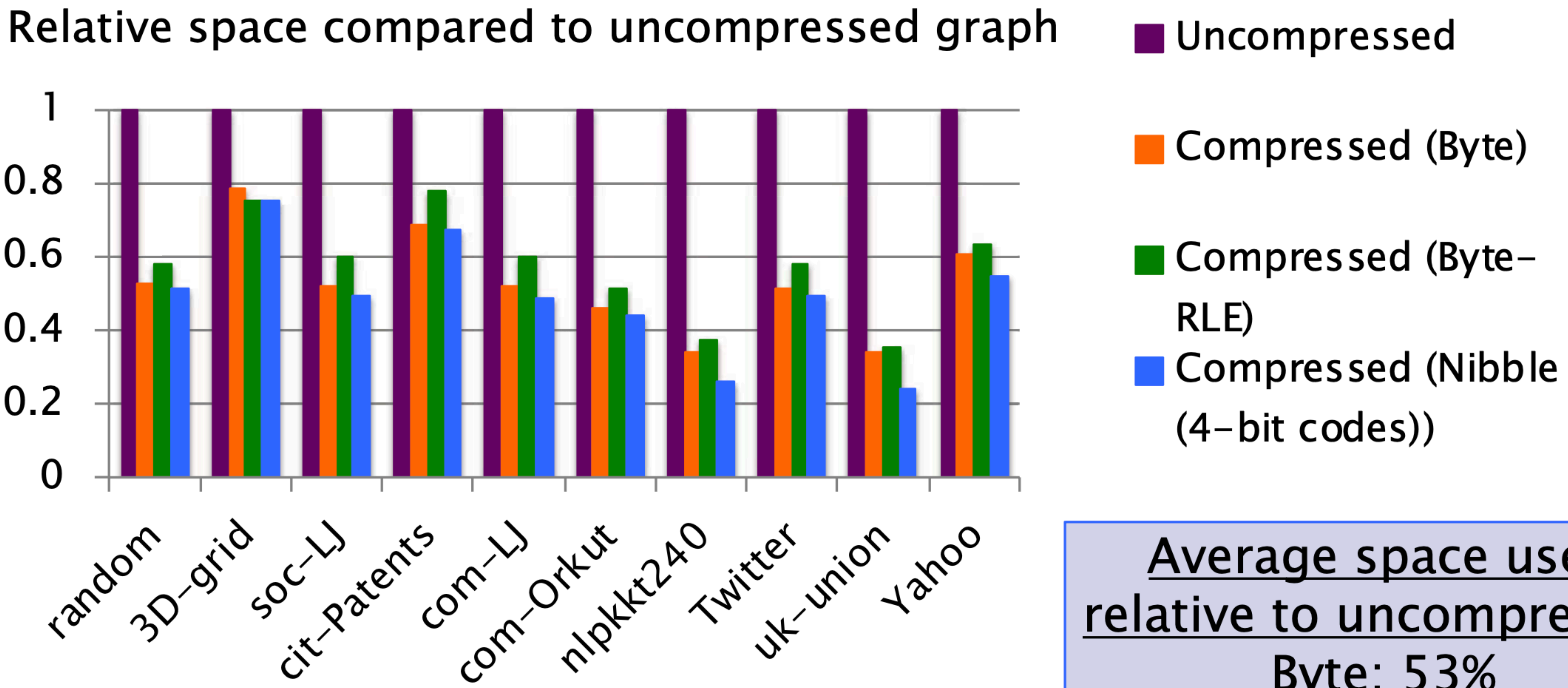
Encode first entry relative to source vertex

All chunks can be decoded in parallel!

- T=100 to 10,000 works well in practice

Good compression for most graphs

- Space to store graph, which dominates the actual space usage for most graphs

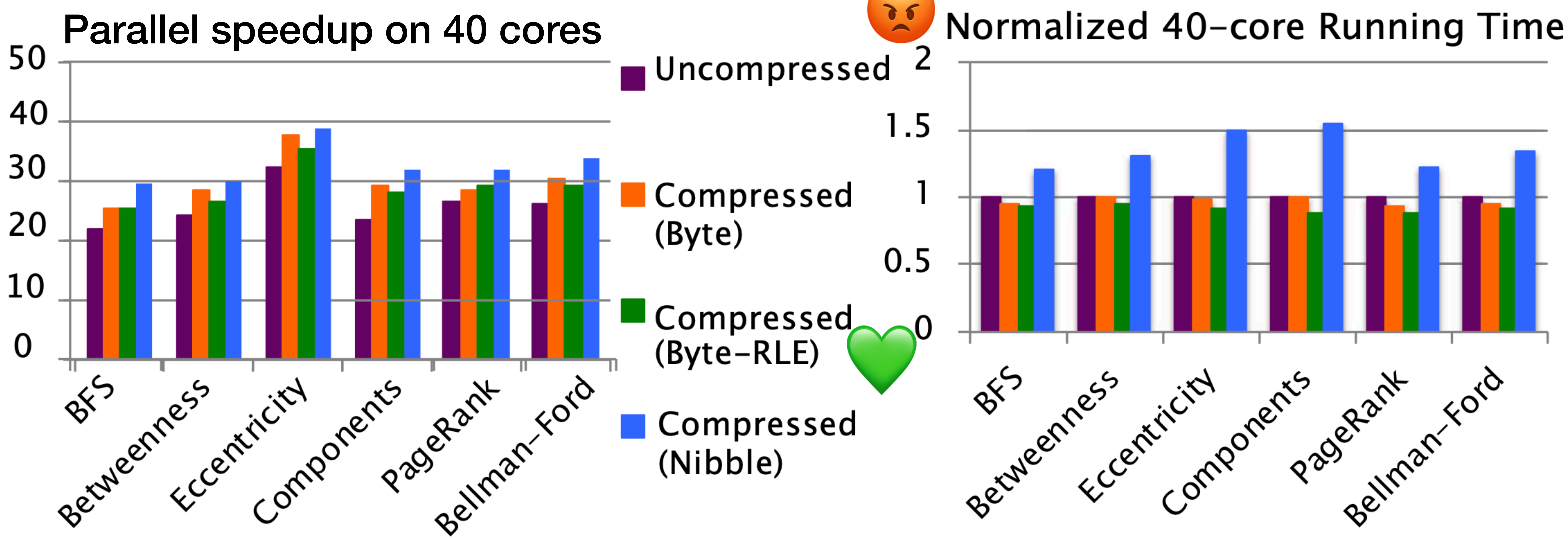


Average space used relative to uncompressed
Byte: 53%
Byte-RLE: 56%
Nibble: 49%

- Can further reduce space but need to ensure decoding is fast

Julian Shun, Laxman Dhulipala and Guy Blelloch. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra +, DCC 2015

What is the cost of decoding on-th-fly?



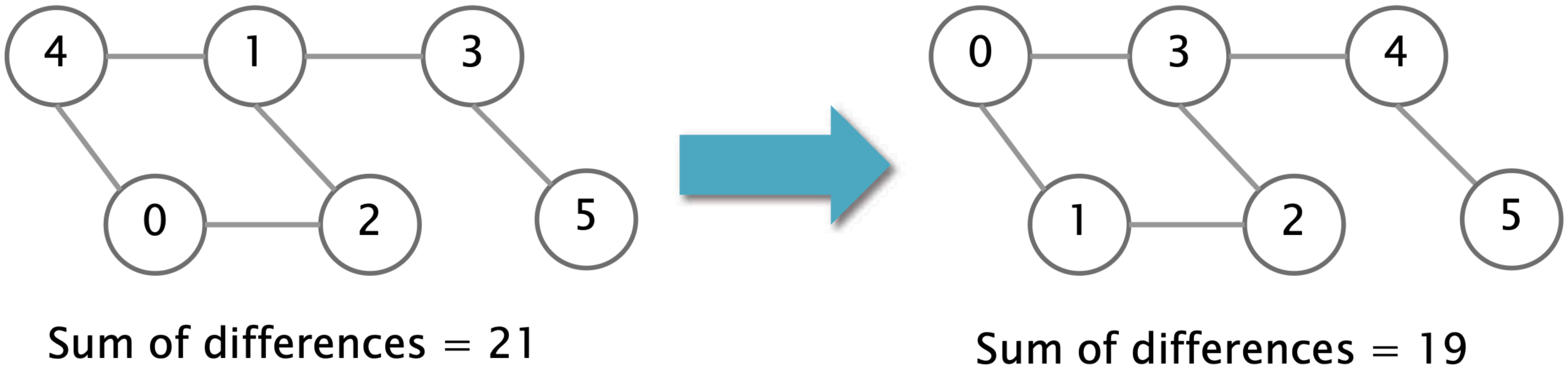
- In parallel, compressed can outperform uncompressed
  - These graph algorithms are memory-bound and memory subsystem is a bottleneck in parallel (contention for resources)
  - Spends less time on memory operations, but has to decode
- Decoding has good speedup so overall speedup is higher
- All techniques integrated into Ligra framework

Julian Shun, Laxman Dhulipala and Guy Blelloch. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra +, DCC 2015

Graph Reordering

Reassign IDs to vertices to improve locality

- Goal: **Make vertex IDs close to their neighbors' IDs** and neighbors' IDs close to each other



- Can improve **compression rate** due to smaller "differences"
- Can improve **performance** due to higher cache hit rate
- Various methods: BFS, DFS, METIS, by degree, etc.

From MIT 6.172

Summary

Real-world graphs are **large and sparse**

Many graphs algorithms are **irregular** and involve many **memory accesses**

Improve performance with **algorithmic optimizations** and by **creating/exploiting locality**

Optimizations may work for some graphs, but not others

BACKUP

Vertices labeled from 0 to n-1

Graph representations

	0	1	2	3	4
0	0	1	0	0	0
1	1	0	0	1	1
2	0	0	0	1	0
3	0	1	1	0	0
4	0	1	0	0	0

Adjacency matrix
("1" if edge exists,
"0" otherwise)

(0,1)
(1,0)
(1,3)
(1,4)
(2,3)
(3,1)
(3,2)
(4,1)

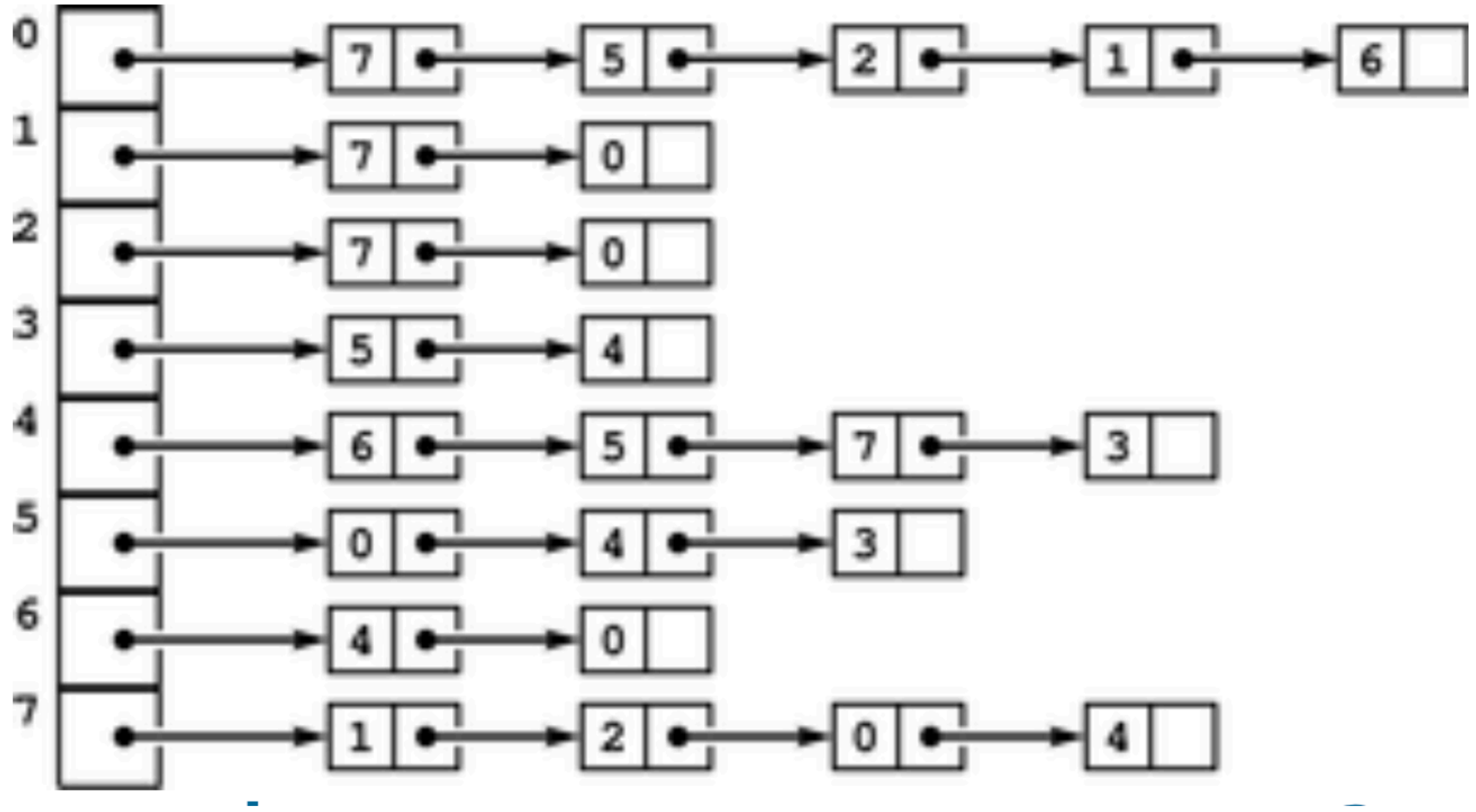
Edge list

What is the space requirement for each in terms of number of edges (m) and number of vertices (n)?

Graph representations

Adjacency list

- Array of pointers (one per vertex)
- Each vertex has an unordered list of its edges
- Can substitute linked lists with arrays for better cache performance (at the cost of updatability)



What is the space requirement?