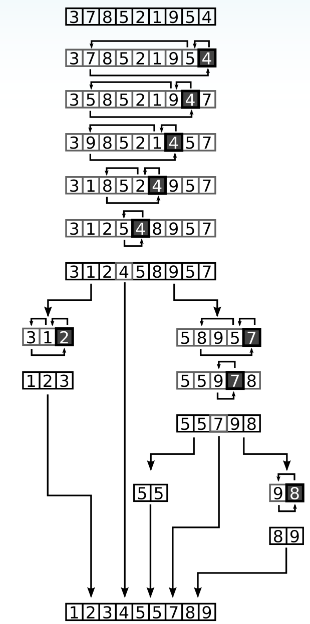
6.827:
Algorithm Engineering



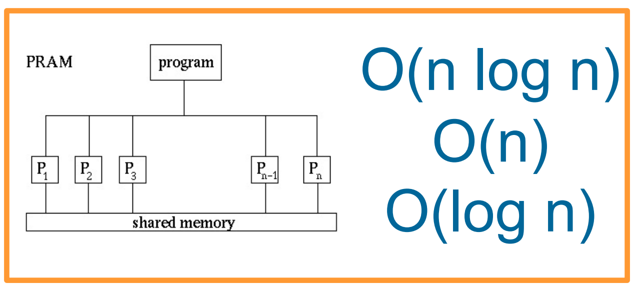
LECTURE 1
Introduction

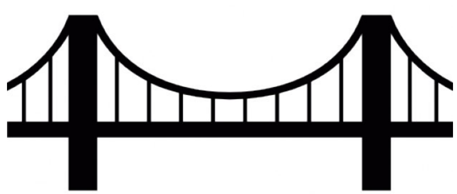
Julian Shun

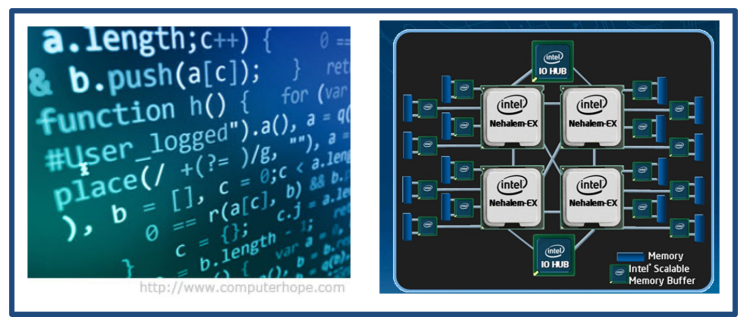
February 1, 2022

What is Algorithm Engineering?

- Algorithm design
- Algorithm analysis
- Algorithm implementation
- Optimization
- Profiling
- Experimental evaluation

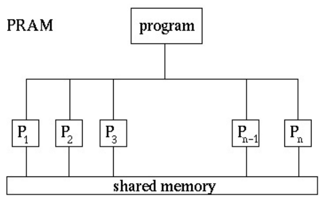




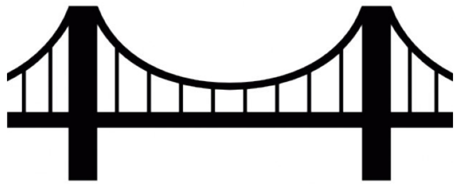


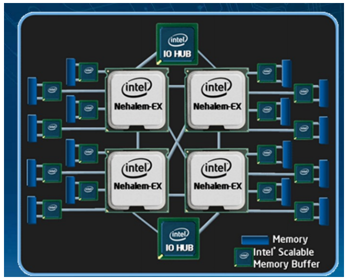
Theory ***Practice***

Bridging Theory and Practice



O(n log n)
O(n)
O(log n)





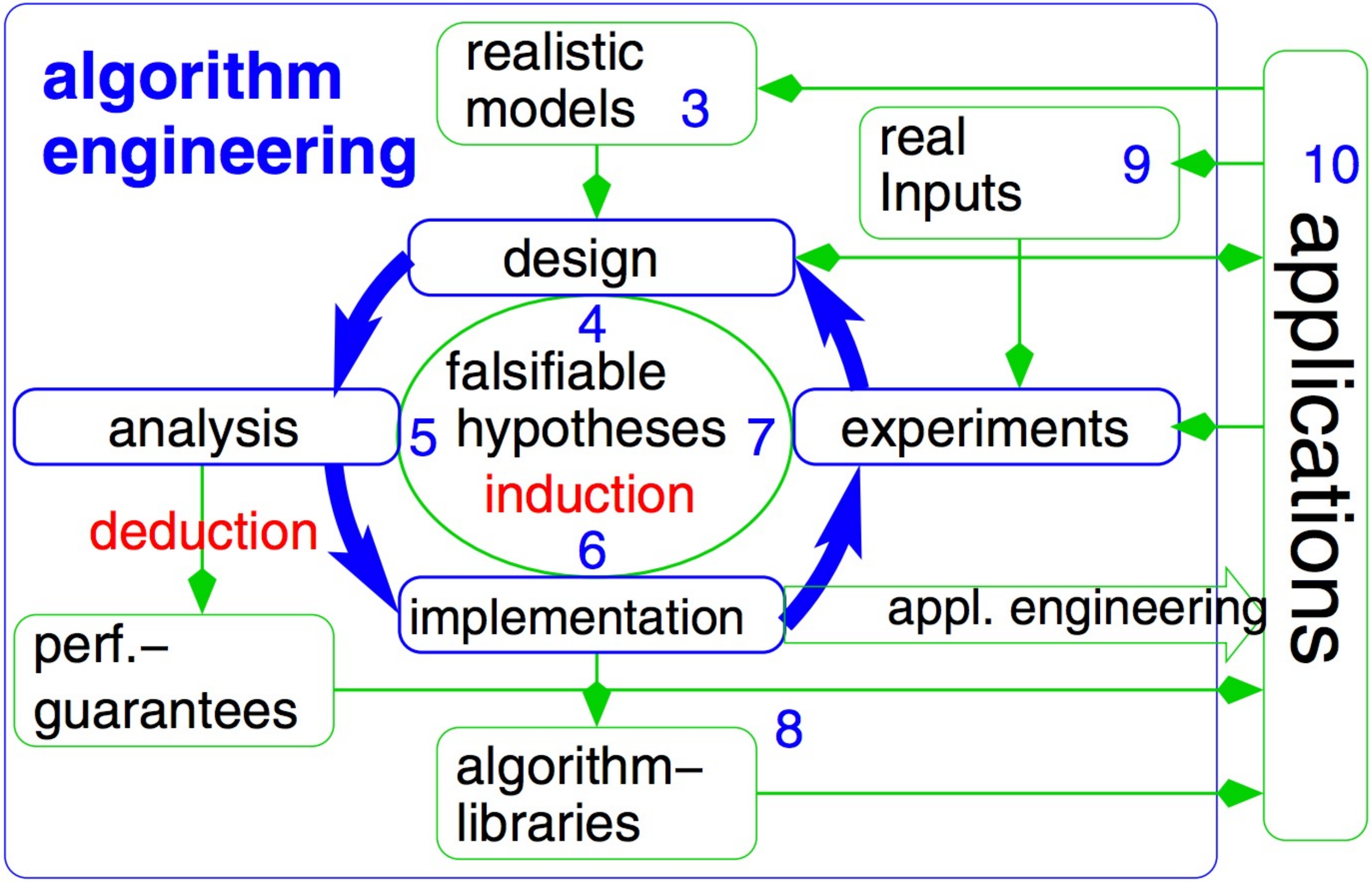
- Good empirical performance
- Confidence that algorithms will perform well in many different settings
- Ability to predict performance (e.g., in real-time applications)
- Important to develop theoretical models to capture properties of technologies

Use theory to inform practice and practice to inform theory.

Brief History

- In early days, implementing algorithms designed was standard practice
- 1970s–1980s: Algorithm theory is a subdiscipline in CS mostly devoted to ”paper and pencil” work
- Late 1980s–1990s: Researchers began noticing gaps between theory and practice
- 1997: First Workshop on Algorithm Engineering (WAE) by P. Italiano (now part of ESA)
- 1998: Meeting on Algorithm Engineering & Experiments (ALENEX)
- 2003: annual Workshop on Experimental Algorithms (WEA), now Symposium on Experimental Algorithms (SEA)
- Nowadays many conferences have papers on algorithm engineering

What is Algorithm Engineering?



Source: “Algorithm Engineering – An Attempt at a Definition”, Peter Sanders

Models of Computation

- Random-Access Machine (RAM)
  - Infinite memory
  - Arithmetic operations, logical operations, and memory accesses take O(1) time
  - Most sequential algorithms are designed using this model (6.006/6.046)
- Nowadays computers are much more complex
  - Deep cache hierarchies
  - Instruction level parallelism
  - Multiple cores
  - Disk if input doesn't fit in memory
  - Asymmetric read-write costs in non-volatile memory

Algorithm Design & Analysis

	Algorithm 1	Algorithm 2
Complexity	$N \log_2 N$	$1000\ N$

- Constant factors matter!
- Avoid unnecessary computations
- Simplicity improves applicability and can lead to better performance
- Think about locality and parallelism
- Think both about worst-case and real-world inputs
- Use theory as a guide to find practical algorithms
- Time vs. space tradeoffs
- Work vs. parallelism tradeoffs

Implementation

- Write clean, modular code
  - Easier to experiment with different methods, and can save a lot of development time
- Write correctness checkers
  - Especially important in numerical and geometric applications due to floating-point arithmetic, possibly leading to different results
- Save previous versions of your code!
  - Version control helps with this

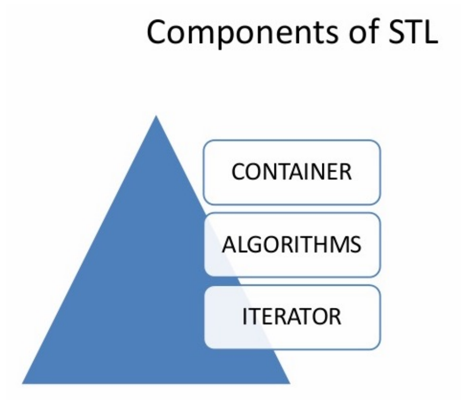
Experimentation

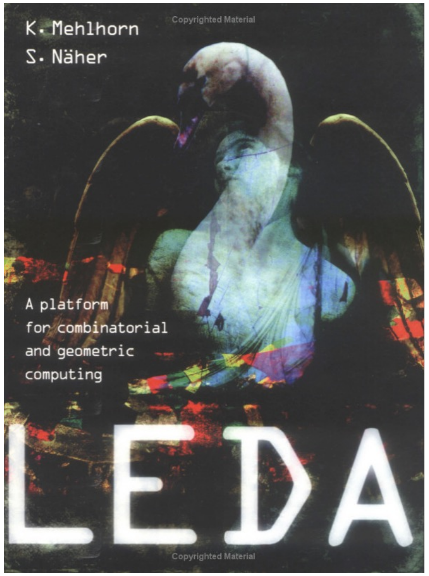
- Instrument code with timers and use performance profilers (e.g., perf, gprof, valgrind)
- Use large variety of inputs (both real-world and synthetic)
  - Use different sizes
  - Use worst-case inputs to identify correctness or performance issues
- Reproducibility
  - Document environmental setup
  - Fix random seeds if needed
- Run multiple times to deal with variance

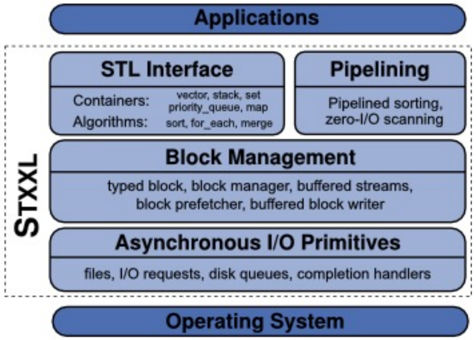
Experimentation II

- For parallel code, test on varying number of processors to study scalability
- Compare with best serial code for problem
- For reproducibility, write deterministic parallel code if possible
  - Or make it easy to turn off non-determinism
- Use numactl to control NUMA effects on multi-socket machines
- Useful tools: Cilkscale, Cilksan

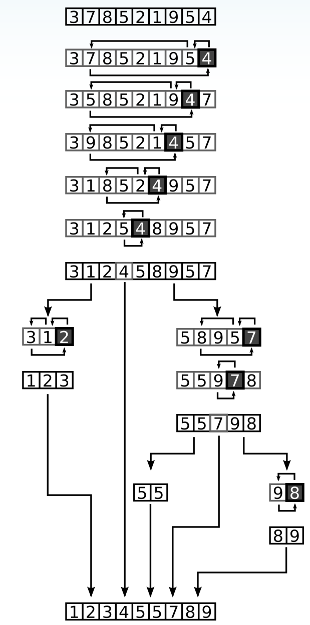
Libraries and Frameworks







- Use efficient building blocks from existing libraries/frameworks when appropriate
- Contribute to existing libraries/frameworks or develop your own to help others and improve applicability



Course information

Course Information

- Graduate-level class
  - Undergraduates who have taken 6.046 and 6.172 are welcome
- Lectures: Tuesday/Thursday 4-5:30pm ET in 36-156
- Instructor: Julian Shun (jshun@mit.edu)
- TA: Tom Tseng (tomtseng@mit.edu)
- Units: 3-0-9
- We will use Piazza for communication
- Office hours by appointment
- This course will cover various ideas in algorithm engineering, with an emphasis on parallelism and graph problems

Course Website

https://people.csail.mit.edu/jshun/6827-s22/

Date	Topic	Speaker	Required Reading	Optional Reading
Tuesday 2/1	Course Introduction	Julian Shun	Algorithm Engineering - An Attempt at a Definition A Theoretician's Guide to the Experimental Analysis of Algorithms	Algorithm Engineering: Bridging the Gap Between Algorithm Theory and Practice A Guide to Experimental Algorithmics Algorithm engineering: an attempt at a definition using sorting as an example Algorithm Engineering for Parallel Computation Distributed Algorithm Engineering Experimental algorithmics Programming Pearls Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time
Thursday 2/3	Parallel Algorithms	Julian Shun	Parallel Algorithms Thinking in Parallel: Some Basic Data-Parallel Algorithms and Techniques (Chapters 4-8) CLRS Chapter 27	Prefix Sums and Their Applications Algorithm Design: Parallel and Sequential Introduction to Parallel Algorithms Scheduling Multithreaded Computations by Work Stealing Thread Scheduling for Multiprogrammed Multiprocessors Problem Based Benchmark Suite
Tuesday 2/8	Parallel Graph Traversal		Direction-Optimizing Breadth-First Search* A Faster Algorithm for Betweenness Centrality The More the Merrier: Efficient Multi-Source Graph Traversal*	A Work-Efficient Parallel Breadth-First Search Algorithm (or How to Cope with the Nondeterminism of Reducers) Internally Deterministic Parallel Algorithms Can Be Fast SlimSell: A Vectorizable Graph Representation for Breadth-First Search Chapter 3.6 of Networks, Crowds, and Markets (describes Betweenness Centrality with an example) Better Approximation of Betweenness Centrality ABRA: Approximating Betweenness Centrality in Static and Dynamic Graphs with Rademacher Averages KADABRA is an ADaptive Algorithm for Betweenness via Random Approximation

Grading

Grading Breakdown	
Paper Reviews	15%
Problem Set	10%
Paper Presentations	20%
Research Project	45%
Class Participation	10%

You must complete all assignments to pass the class.

Paper Presentations

- This is a research-oriented course
- Cover content from 2-3 research papers each lecture
- 25-30 minute student presentation per paper
  - Discuss motivation for the problem solved
  - Key technical ideas
  - Theoretical/experimental results
  - Related work
  - Strengths/weaknesses
  - Directions for future work
  - Include several questions for discussion
  - Presentation should cover necessary background to understand paper (you may have to read related papers)
  - Make slides but may use the whiteboard for theory
- Sign up for presentations today in Google doc
- Would be helpful to sign up even if listening

Paper Reviews

- Submit one paper review for each lecture
  - Starting next week
  - Cover motivation, key ideas, results, novelty, strengths/weaknesses, your ideas for improving the techniques or evaluation, any open problems or directions for further work
  - Submit on Canvas by 12pm ET on the day of each lecture (before we cover the papers)

Problem Set

- Complete a problem set on parallel algorithms
  - To be released this week and due on 2/28

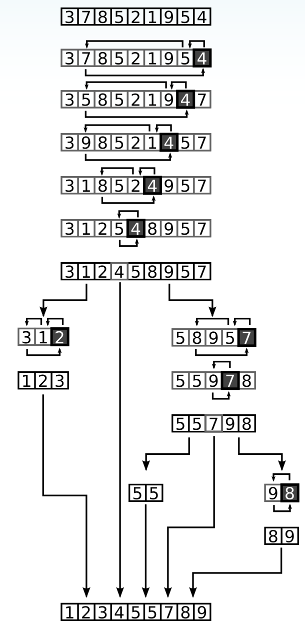
Research Project

- Open-ended research project to be done in groups of 1-3 people
- Some ideas
  - Implementation of non-trivial algorithms
  - Analyzing/optimizing performance of existing algorithms
  - Designing new theoretically and/or practically efficient algorithms
  - Applying algorithms in the context of larger applications
  - Improving or designing new algorithm frameworks or libraries
  - Any topic may involve parallelism, cache-efficiency, I/O-efficiency, and memory-efficiency
- Must contain an implementation component
- Can be related to research that you are doing

Project Timeline

Assignment	Due Date
Pre-proposal meeting	3/3
Proposal	3/8
Weekly progress reports	3/11, 3/18, 4/1, 4/8, 4/15, 4/22, 4/29, 5/6
Mid-term report	4/12
Project presentations	5/10
Final report	5/10

- Pre-proposal meeting
  - 15-minute meeting to run ideas by instructors
- Computing resources for the project
  - Sign up for AWS Educate or Google Cloud Platform for free cloud computing credits
  - Talk to instructor if you need additional credits



PARALLELISM

Parallelism

Data is becoming very large!

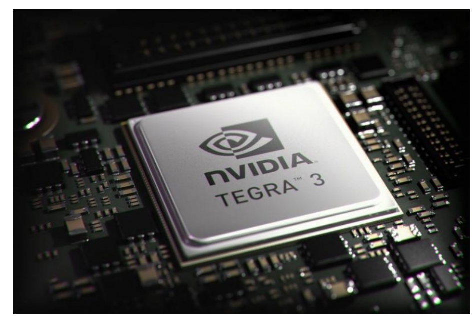
41 million vertices
1.5 billion edges
(6.3 GB)

1.4 billion vertices
6.6 billion edges
(38 GB)

Common Crawl

3.5 billion vertices
128 billion edges
(540 GB)

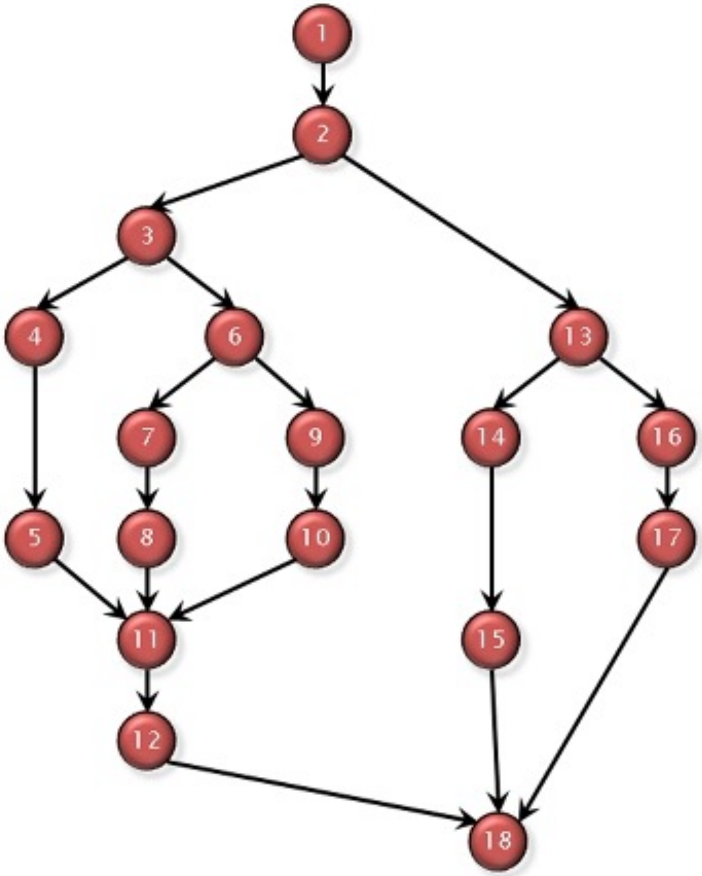
Parallel machines are everywhere!



Can rent machines on AWS with 72 cores (144 hyper-threads) and 4TB of RAM

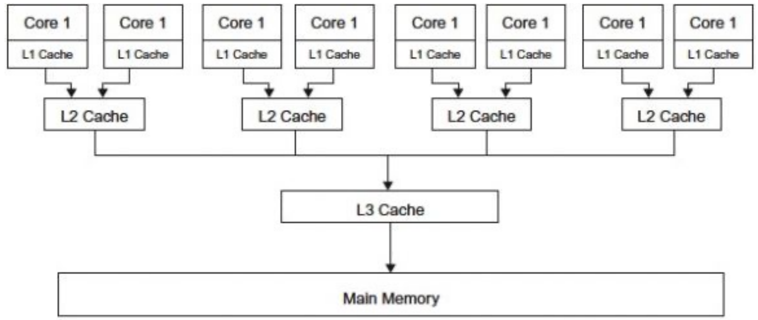
Parallelism Models

Computation graph

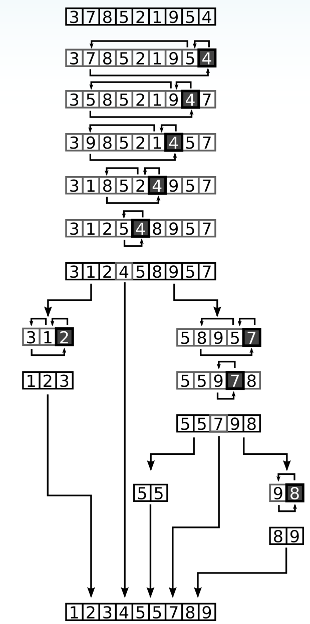


- Work = number of vertices in graph (number of operations)
- Depth (Span) = longest directed path in graph (dependence length)
- Running time ≤ (Work/#processors) + O(Depth)
- A **work-efficient** parallel algorithm has work that asymptotically matches that of the best sequential algorithm for the problem

Goal 1: work-efficient and low (polylogarithmic) depth algorithms

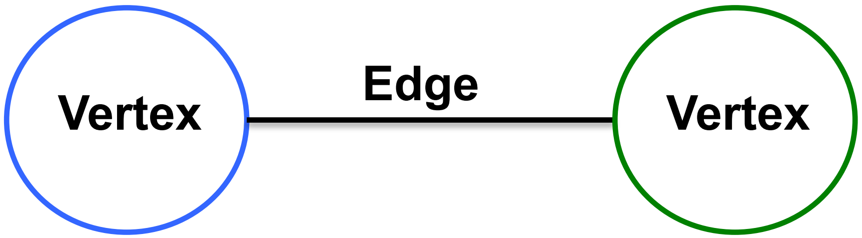


Goal 2: simple, practical, and cache-friendly

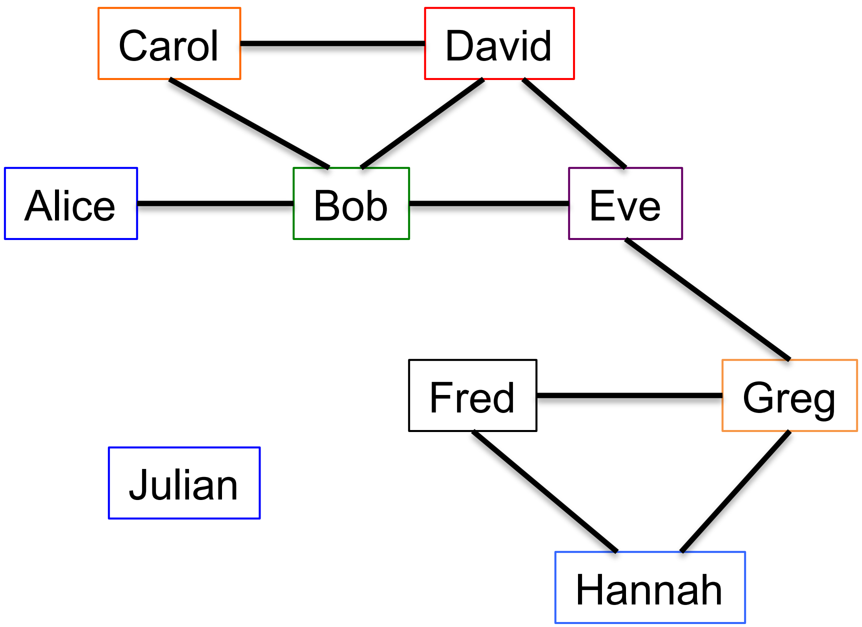

3 7 8 5 2 1 9 5 4
3 7 8 5 2 1 9 5 4
3 5 8 5 2 1 9 4 7
3 9 8 5 2 1 4 5 7
3 1 8 5 2 4 9 5 7
3 1 2 5 4 8 9 5 7
3 1 2 4 5 8 9 5 7
3 1 2
5 8 9 5 7
1 2 3
5 5 9 7 8
5 5 7 9 8
5 5
9 8
8 9
1 2 3 4 5 5 7 8 9

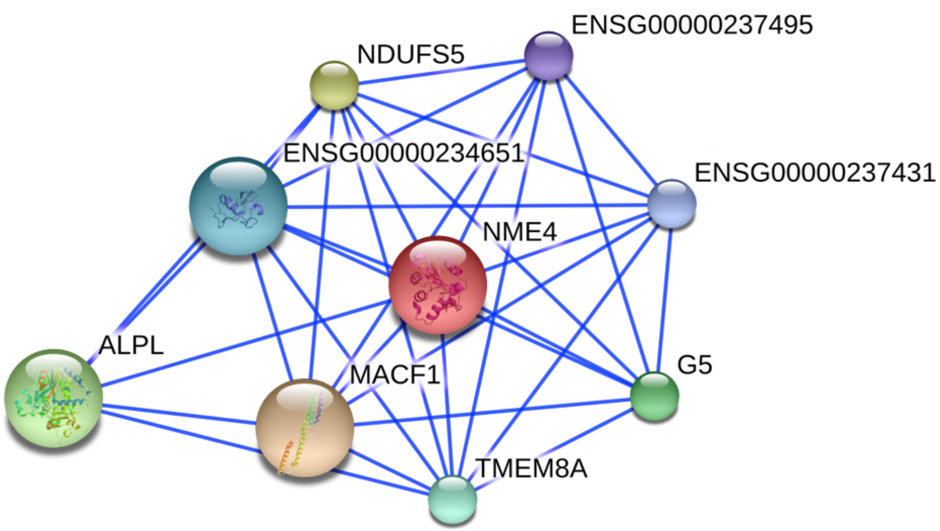
GRAPHS

What is a graph?



- **Vertices** model objects
- **Edges** model relationships between objects





https://commons.wikimedia.org/wiki/File:Protein_Interaction_Network_for_TMEM8A.png

Graph Representations

- Vertices labeled from 0 to n–1

	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)

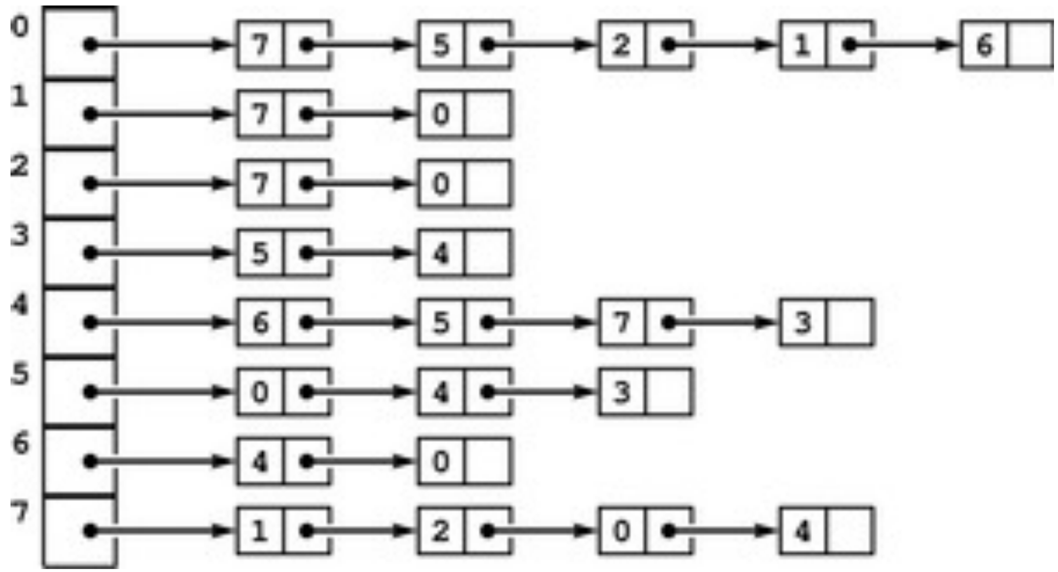
(0,1)
(1,0)
(1,3)
(1,4)
(2,3)
(3,1)
(3,2)
(4,1)

Edge list

- $O(n^2)$ space for adjacency matrix
- $O(m)$ space for edge list

Graph Representations

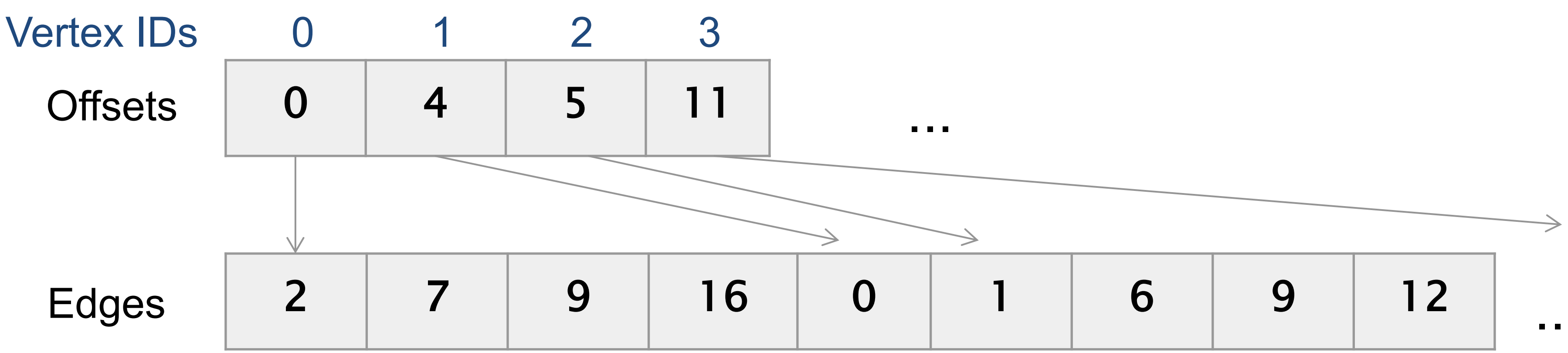
- Adjacency list
  - Array of pointers (one per vertex)
  - Each vertex has an unordered list of its edges



- Space requirement is O(n+m)
- Can substitute linked lists with arrays for better cache performance
  - Tradeoff: more expensive to update graph

Graph Representations

- Compressed sparse row (CSR)
  - Two arrays: Offsets and Edges
  - Offsets[i] stores the offset of where vertex i's edges start in Edges



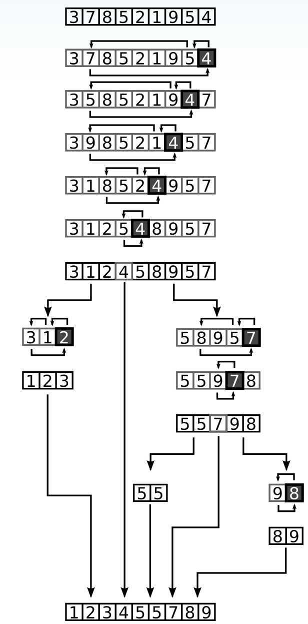
- How do we know the degree of a vertex?
- Space usage is O(n+m)
- Can also store values on the edges with an additional array or interleaved with Edges

Tradeoffs in Graph Representations

- What is the cost of different operations?

	Adjacency matrix	Edge list	Adjacency list (linked list)	Compressed sparse row
Storage cost / scanning whole graph	$O(n^2)$	$O(m)$	$O(m+n)$	$O(m+n)$
Add edge	$O(1)$	$O(1)$	$O(1)$	$O(m+n)$
Delete edge from vertex v	$O(1)$	$O(m)$	$O(deg(v))$	$O(m+n)$
Finding all neighbors of a vertex v	$O(n)$	$O(m)$	$O(deg(v))$	$O(deg(v))$
Finding if w is a neighbor of v	$O(1)$	$O(m)$	$O(deg(v))$	$O(deg(v))$

- There are variants/combinations of these representations



Breadth-first search

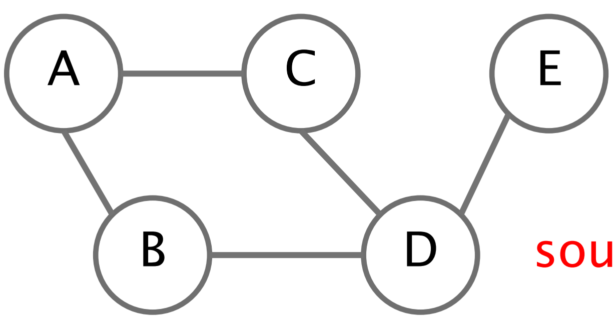
Breadth-First Search (BFS)

- 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*

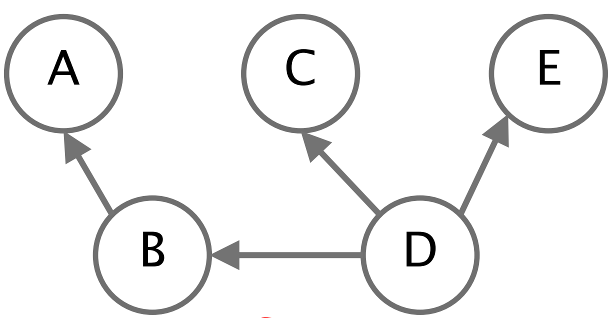
A	B	C	D	E
2	1	1	0	1

  - 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

Sequential BFS Algorithm

```
procedure BFS(G, root) is
    let Q be a queue
    label root as explored
    Q.enqueue(root)
    while Q is not empty do
        v := Q.dequeue()
        if v is the goal then
            return v
        for all edges from v to w in G.adjacentEdges(v) do
            if w is not labeled as explored then
                label w as explored
                Q.enqueue(w)
```

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

- What is the running time of BFS?
  - Each node is enqueued and dequeued once: O(n)
  - Each edge is visited once in each direction: O(m)
- Total: O(n+m)

Sequential BFS Algorithm

- Assume graph is given in compressed sparse row format
  - Two arrays: Offsets and Edges
  - n vertices and m edges (assume Offsets[n] = m)

```
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;
         }
    }
}
```

Total of m random accesses

- What is the most expensive part of the code?
  - Random accesses cost more than sequential accesses

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

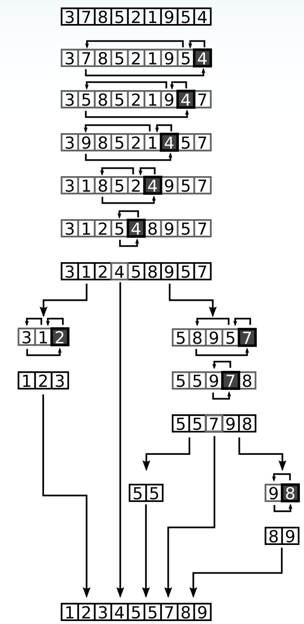
- What if we can fit a bitvector of size n in cache?
  - Might reduce the number of cache misses
  - More computation to do bit manipulation

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

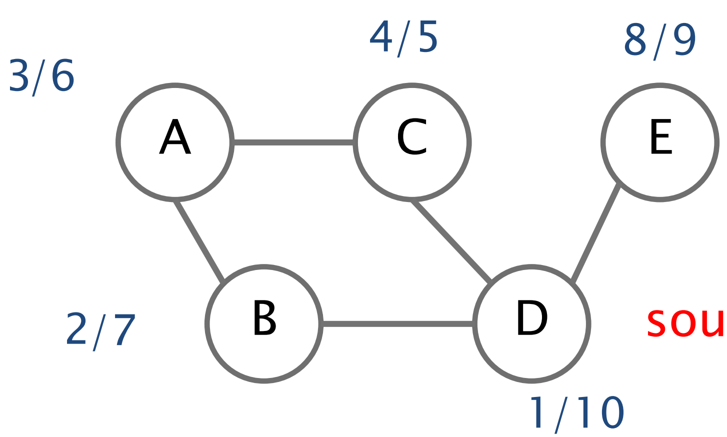


DEPTH-FIRST SEARCH

Depth-First Search (DFS)

- Explores edges out of the most recently discovered vertex
- Possible outputs:
  - Depth-first forest
  - Vertices in the order they were first visited (preordering)
  - Vertices in the order they were last visited (postordering)
  - Reverse postordering

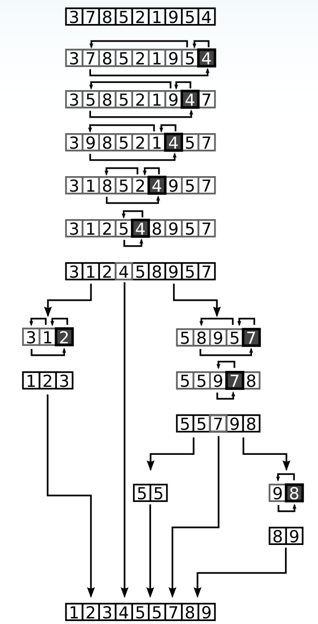
Applications
Topological sort
Solving mazes
Biconnected components
Strongly connected components
Cycle detection
...



source = D

Preorder: D, B, A, C, E
Postorder: C, A, B, E, D
Reverse postorder: D, E, B, A, C

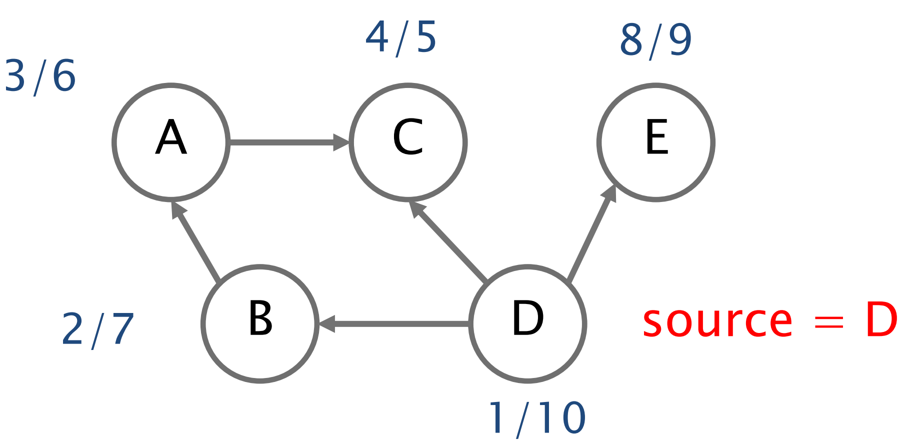
DFS requires O(n+m) work on n vertices and m edges



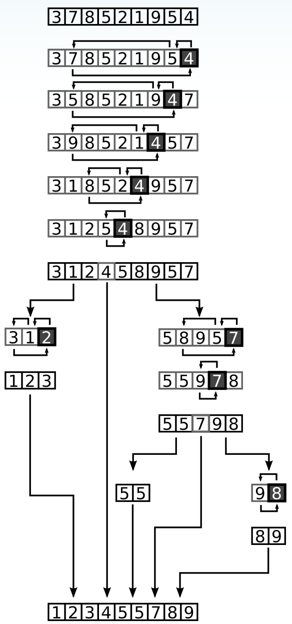
TOPOLOGICAL SORT

Topological Sort

- Given a directed acyclic graph, output the vertices in an order such that all predecessors of a vertex appear before it
  - Application: scheduling tasks with dependencies (e.g., parallel computing, Makefile)
- Solution: output vertices in reverse postorder in DFS



Reverse postorder: D, E, B, A, C



SHORTEST PATHS

Single-Source Shortest Paths

- Given a weighted graph and a source vertex, output the distance from the source vertex to every vertex

- Non-negative weights
  - Dijkstra's algorithm
  - O(m + n log n) work using Fibonacci heap

- General weights
  - Bellman-Ford algorithm
  - O(mn) work

Dijkstra's Algorithm

```
function Dijkstra(Graph, source):
    dist[source] ← 0                                        // Initialization

    create vertex set Q

```

- O((m+n)log n) work using normal heap
- O(m + n log n) work using Fibonacci heap
  - Extract-min takes O(log n) work but decreasing priority only takes O(1) work (amortized)

Bellman-Ford Algorithm

```
Bellman-Ford(G, source):
    ShortestPaths = {∞, ∞, …, ∞}    //size n; stores shortest path distances
    ShortestPaths[source] = 0
    for i=1 to n:
        for each vertex v in G:
            for each w in neighbors(v):
                if(ShortestPaths[v] + weight(v,w) < ShortestPaths[w]):
                    ShortestPaths[w] = ShortestPaths[v] + weight(v,w)
        if no shortest paths changed:
            return ShortestPaths
    report "negative cycle"
```

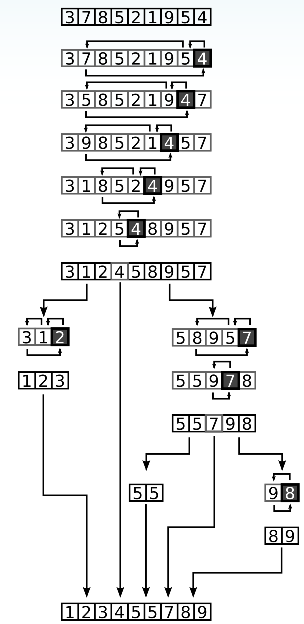
- At most O(n) rounds, each doing O(n+m) work
- Total work = O(mn)

More Graph Algorithms

- We will study algorithms for particular problems
  - Parallelism, cache-efficiency, I/O-efficiency, dynamic updates

Breadth-first search	Betweenness centrality
PageRank	Clustering
Low-diameter decomposition	SSSP
Connected components	Maximal independent set
K-core decomposition	Multi-BFS
Minimum spanning forest	Spanning forest
Maximal matching	Graph coloring
Subgraph matching	Dense subgraph discovery

GRAPH PROCESSING FRAMEWORKS



Graph Processing Frameworks

- Provides high-level primitives for graph algorithms
- Reduce programming effort of writing efficient parallel graph programs

Graph processing frameworks/libraries

Pregel, Giraph, GPS, GraphLab, PowerGraph, PRISM, Pegasus, Knowledge Discovery Toolbox, CombBLAS, GraphChi, GraphX, Galois, X-Stream, Gunrock, GraphMat, Ringo, TurboGraph, TurboGraph++, FlashGraph, Grace, PathGraph, Polymer, GPSA, GoFFish, Blogel, LightGraph, MapGraph, PowerLyra, PowerSwitch, Imitator, XDGP, Signal/Collect, PrefEdge, EmptyHeaded, Gemini, Wukong, Parallel BGL, KLA, Grappa, Chronos, Green-Marl, GraphHP, P++, LLAMA, Venus, Cyclops, Medusa, NScale, Neo4J, Trinity, GBase, HyperGraphDB, Horton, GSPARQL, Titan, ZipG, Cagra, Milk, Ligra, Ligra+, Julienne, GraphPad, Mosaic, BigSparse, Graphene, Mizan, Green-Marl, PGX, PGX.D, Wukong+S, Stinger, cuStinger, Distinger, Hornet, GraphIn, Tornado, Bagel, KickStarter, Naiad, Kineograph, GraphMap, Presto, Cube, Giraph++, Photon, TuX2, GRAPE, GraM, Congra, MTGL, GridGraph, NXgraph, Chaos, Mmap, Clip, Floe, GraphGrind, DualSim, ScaleMine, Arabesque, GraMi, SAHAD, Facebook TAO, Weaver, G-SQL, G-SPARQL, gStore, Horton+, S2RDF, Quegel, EAGRE, Shape, RDF-3X, CuSha, Garaph, Totem, GTS, Frog, GBTL-CUDA, Graphulo, Zorro, Coral, GraphTau, Wonderland, GraphP, GraphIt, GraPu, GraphJet, ImmortalGraph, LA3, CellIQ, AsyncStripe, Cgraph, GraphD, GraphH, ASAP, RStream, and many others…

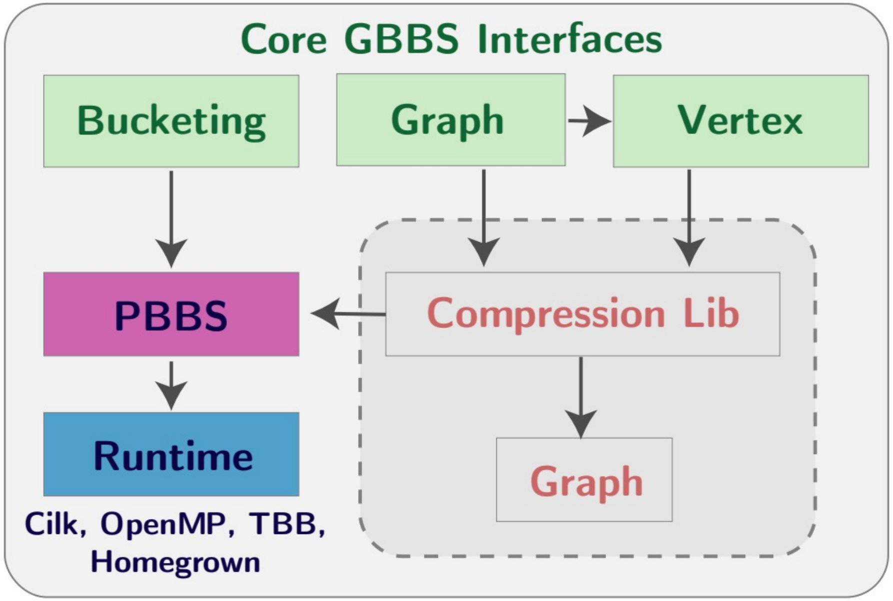
Graph Based Benchmark Suite (GBBS)

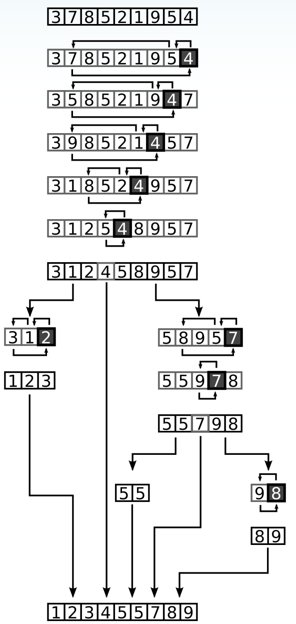
- Benchmark suite containing fast multicore implementations for over 20 graph problems
  - Fast in both theory and practice
  - Scalable to the largest publicly-available graphs

- High-level graph processing interface

- Compressed graph representations

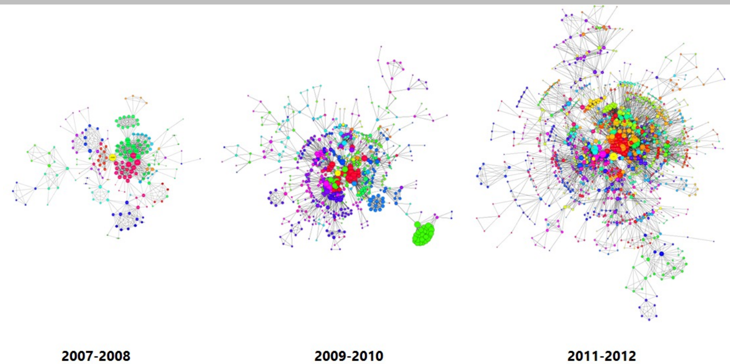
- Python wrapper

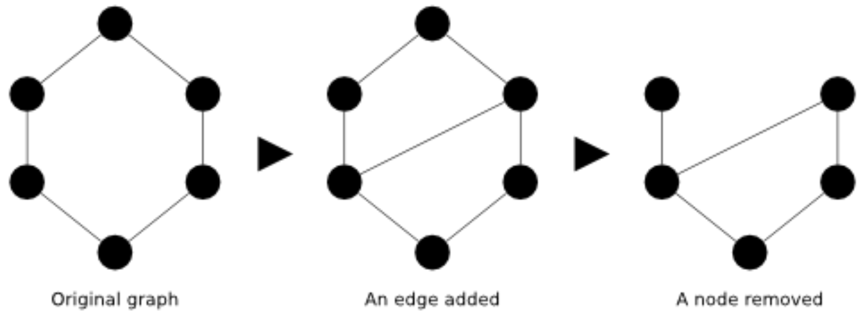




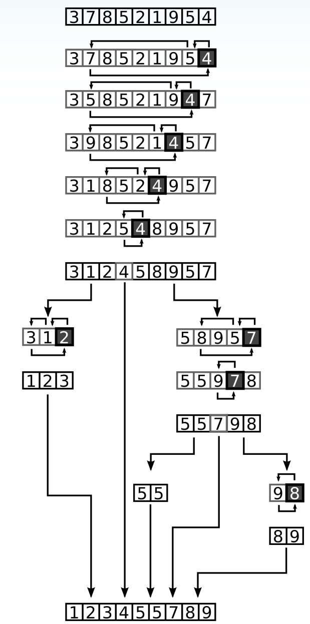
DYNAMIC GRAPHS

Dynamic Graphs



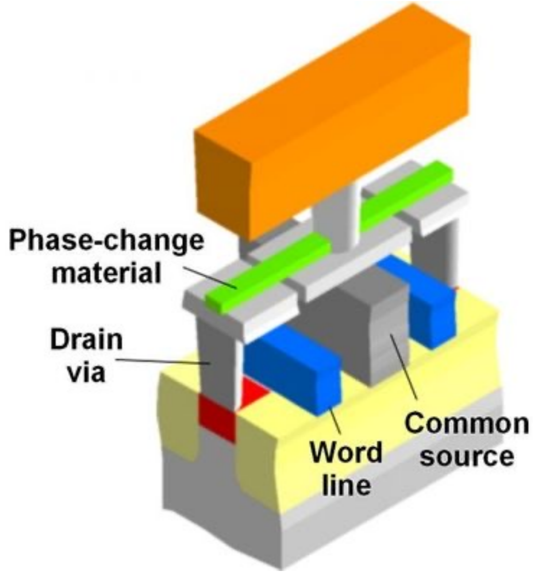


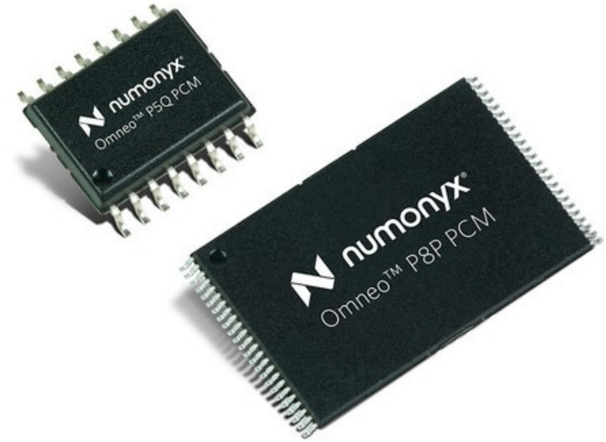
- Many graphs are changing over time
  - Adding/deleting connections on social networks
  - Traffic conditions changing
  - Communication networks (email, IMs)
  - World Wide Web
  - Content sharing (Youtube, Flickr, Pinterest)
- Need graph data structures that allow for efficient updates (in parallel)
- Need (parallel) algorithms that respond to changes without re-computing from scratch



WRITE-EFFICIENT GRAPH ALGORITHMS

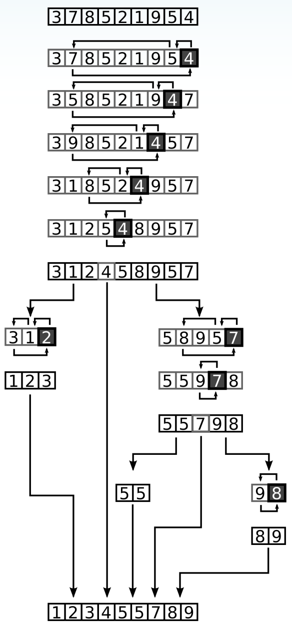
Non-Volatile Memory



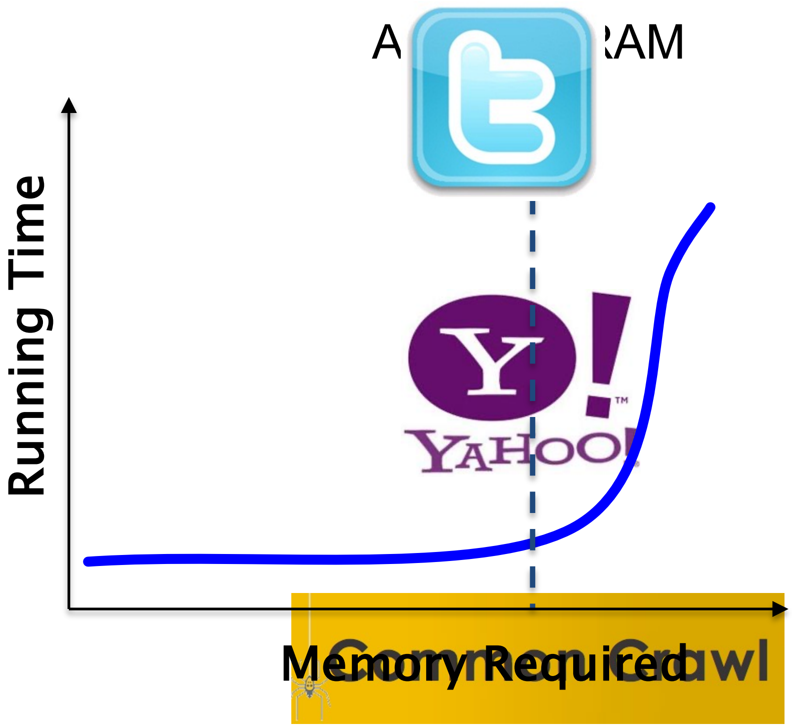


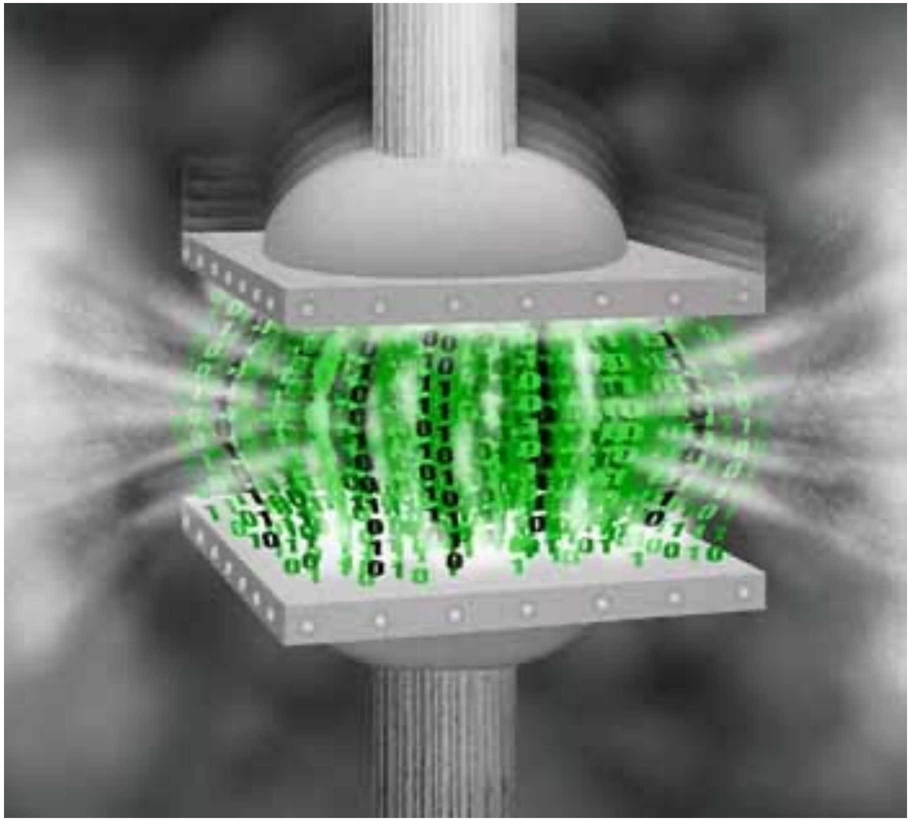
- Non-volatile memories projected to become a dominant form of main memory
- Significant gap in cost for reads vs. writes (energy and latency)
- Need to design models and algorithms (for graphs) that take read-write asymmetry into account

Compression



Large Graphs



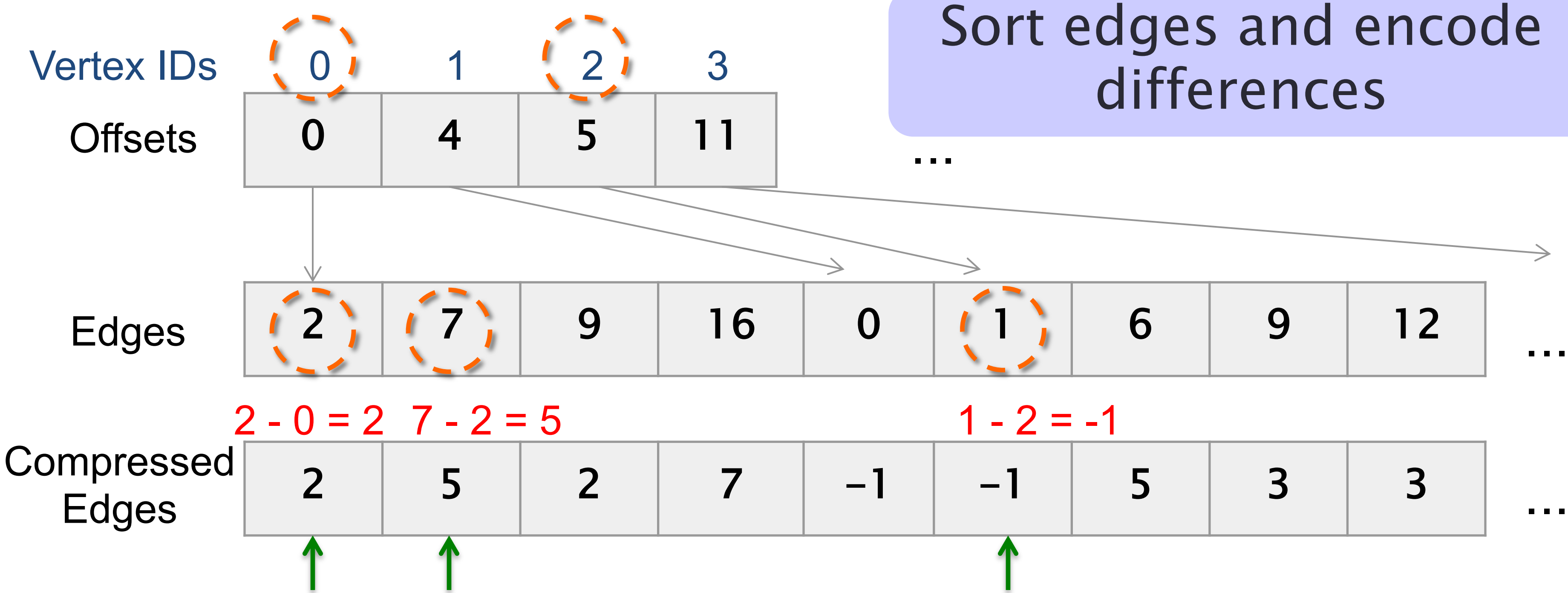


- What if you cannot fit a graph on your machine?
- Cost of machines increases with memory size

Graph Compression

Graph Compression on CSR

Sort edges and encode differences



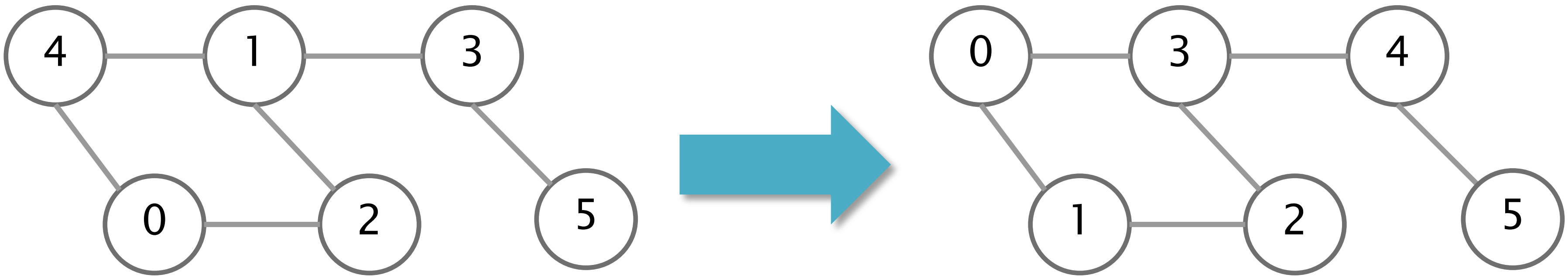
- For each vertex v:
  - First edge: difference is Edges[Offsets[v]]−v
  - i'th edge (i>1): difference is Edges[Offsets[v]+i]−Edges[Offsets[v]+i−1]
- Want to use fewer than 32 or 64 bits per value
- Compression can improve running time

Fast Compression Schemes

- Study speed and space tradeoffs in compression schemes for integer sequences
- Also useful in storing inverted lists for information retrieval

Graph Reordering

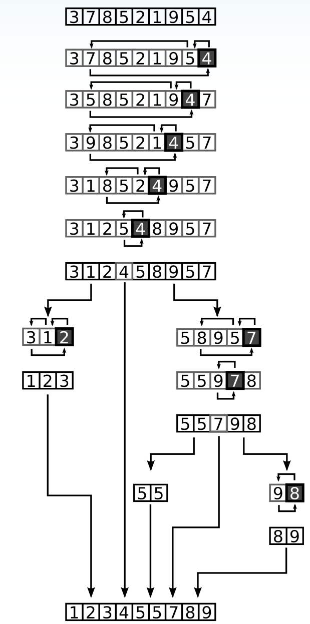
- Reassign IDs to vertices to improve locality
  - Goal: Make vertex IDs close to their neighbors' IDs and neighbors' IDs close to each other



Sum of differences = 23

Sum of differences = 20

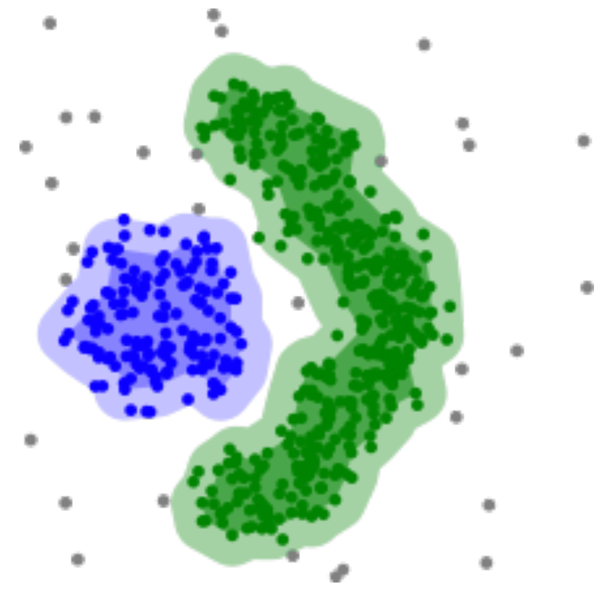
- Can improve compression rate due to smaller "differences"
- Can improve performance due to higher cache hit rate
- Various methods: BFS, DFS, METIS, degree, etc.

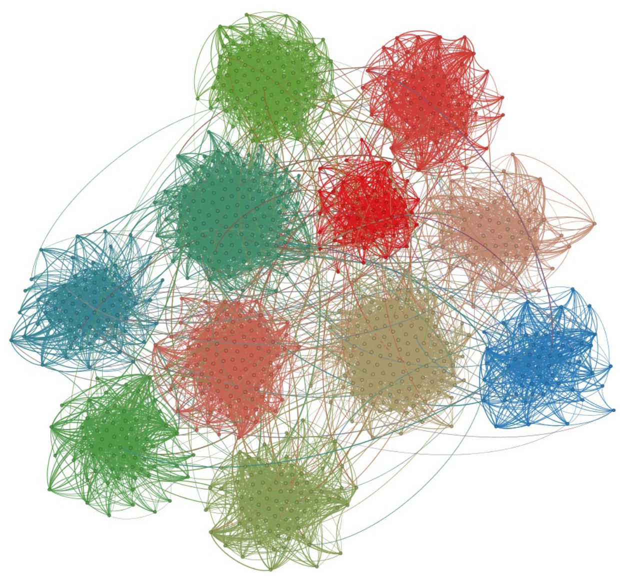

3 7 8 5 2 1 9 5 4
3 7 8 5 2 1 9 5 4
3 5 8 5 2 1 9 4 7
3 9 8 5 2 1 4 5 7
3 1 8 5 2 4 9 5 7
3 1 2 5 4 8 9 5 7
3 1 2 4 5 8 9 5 7
3 1 2
5 8 9 5 7
1 2 3
5 5 9 7 8
5 5 7 9 8
5 5
9 8
8 9
1 2 3 4 5 5 7 8 9

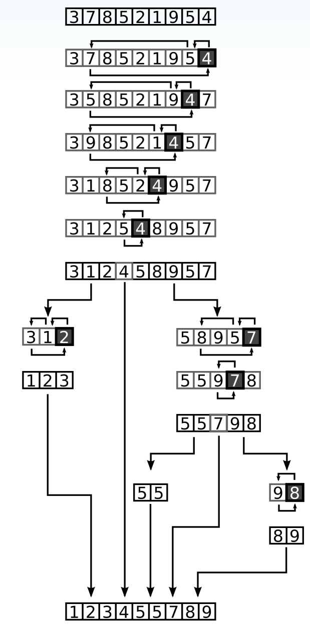
CLUSTERING

Clustering

- Group "similar" objects together, and separate "dissimilar" objects
- Can be applied to spatial data and graph data
- Applications: Community detection, bioinformatics, parallel/distributed processing, visualization, image segmentation, anomaly detection, document analysis, machine learning, etc.

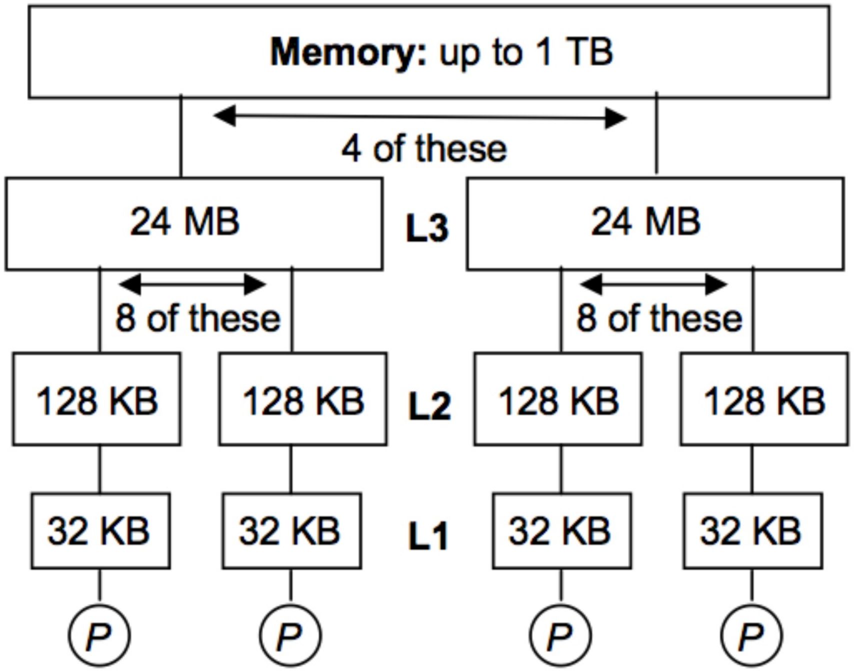






Caching and Non-uniform memory access

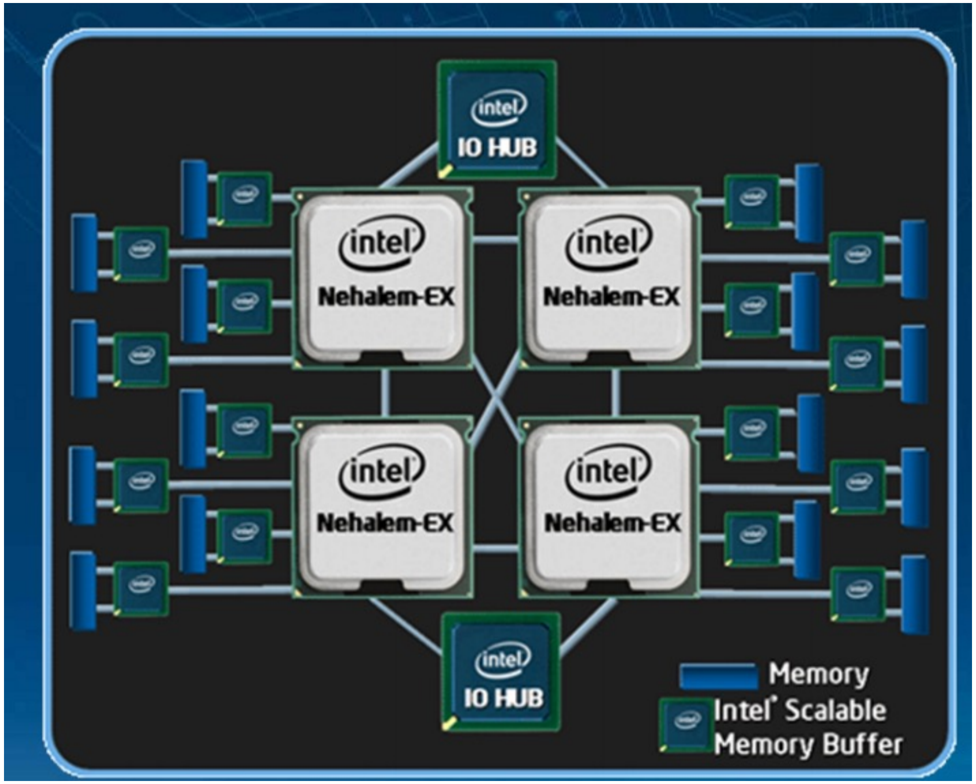
Cache Hierarchies



Design cache-efficient and cache-oblivious algorithms to improve locality

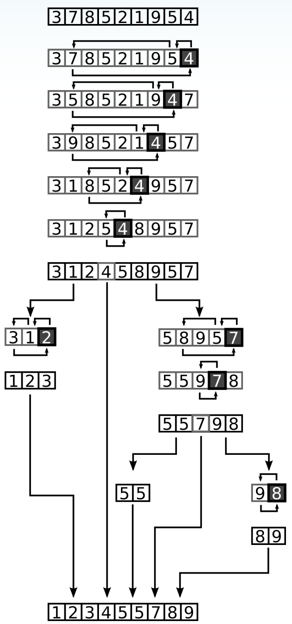
Memory level	Approx latency
L1 Cache	1–2ns
L2 Cache	3–5ns
L3 cache	12–40ns
DRAM	60–100ns

Non-uniform Memory Access (NUMA)



Design NUMA-aware algorithms to improve locality

- Accessing remote memory is more expensive than accessing local memory of a socket
  - Latency depends on the number of hops



I/O Efficiency

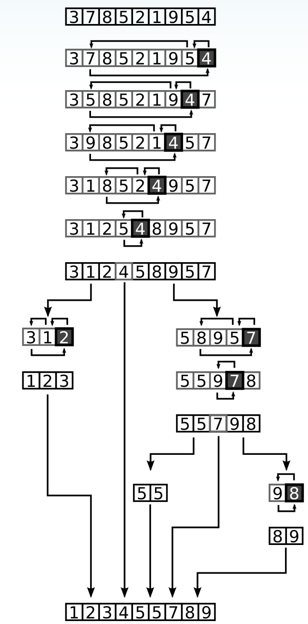
I/O Efficiency

- Need to read input from disk at least once
- Need to read many more times if input doesn't fit in memory

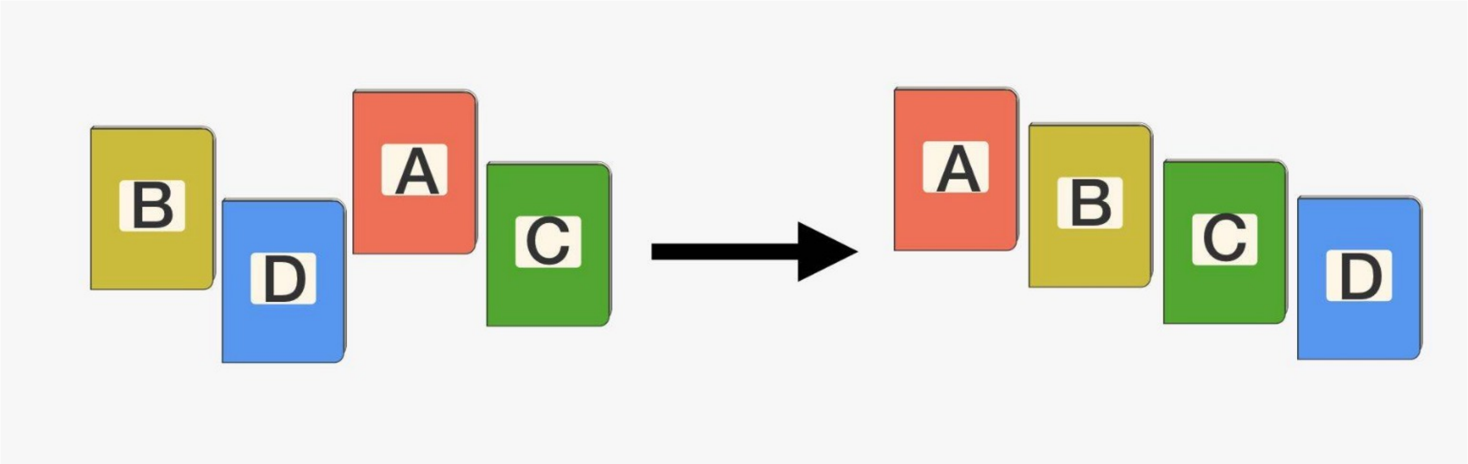
Memory	Latency	Throughput
DRAM	60-100 ns	Tens of GB/s
SSD	Tens of μs	500 MB-2 GB/s (seq), 50-200 MB/s (rand)
HDD	Tens of ms	200 MB/s (seq), 1 MB/s (rand)

Source: https://www.pcgamer.com/hard-drive-vs-ssd-performance/2/

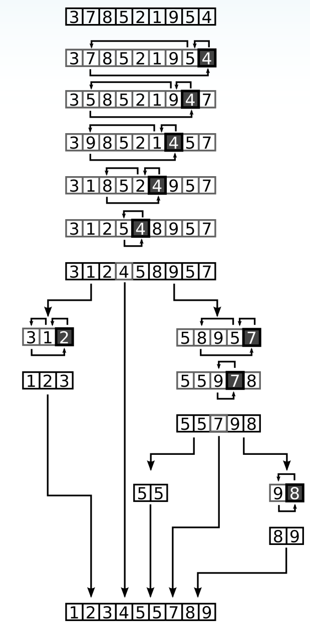
Sorting Algorithms



Sorting

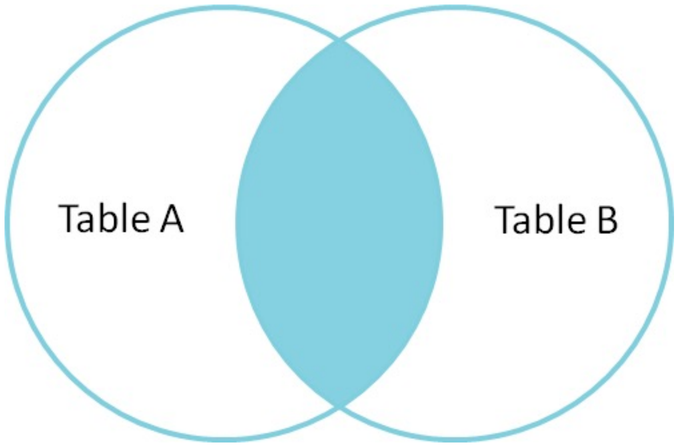


- Lots of research on engineering sorting algorithms
- Will study parallel comparison sorting and radix sorting algorithms
- http://sortbenchmark.org/

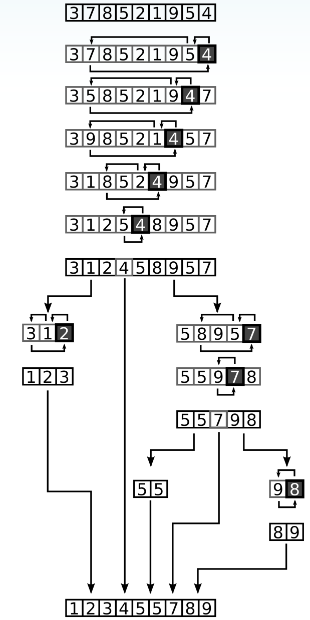


JOINS AND AGGREGATION

Joins and Aggregation

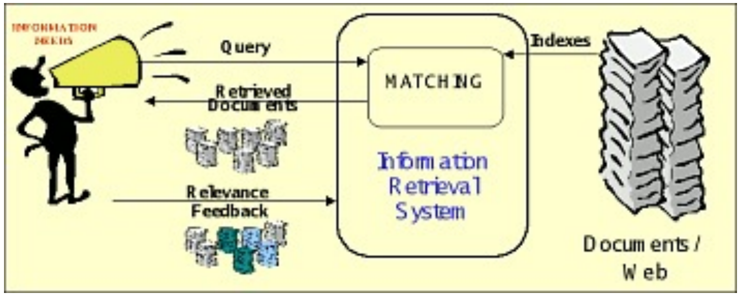


- JOIN and GROUPBY are two of the most expensive operations in database systems
- We will study algorithms and optimizations for these operations (in main-memory)



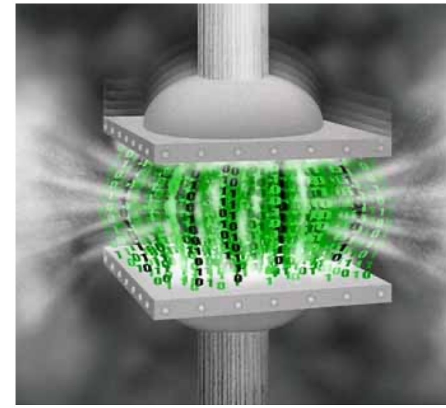
STRING ALGORITHMS

String Algorithms

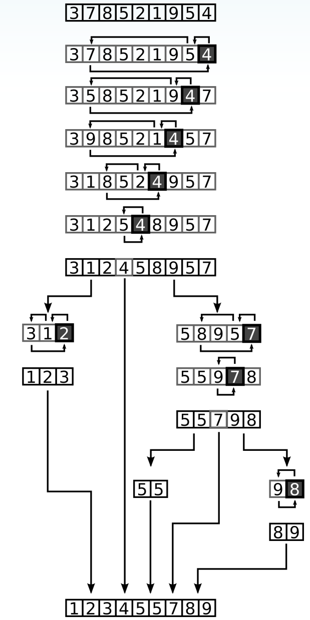


	A	B	C	D	E	F	G	H	I	J	K	L	M
ROT13	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕
	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

	H	E	L	L	O
ROT13	↕	↕	↕	↕	↕
	U	R	Y	Y	B



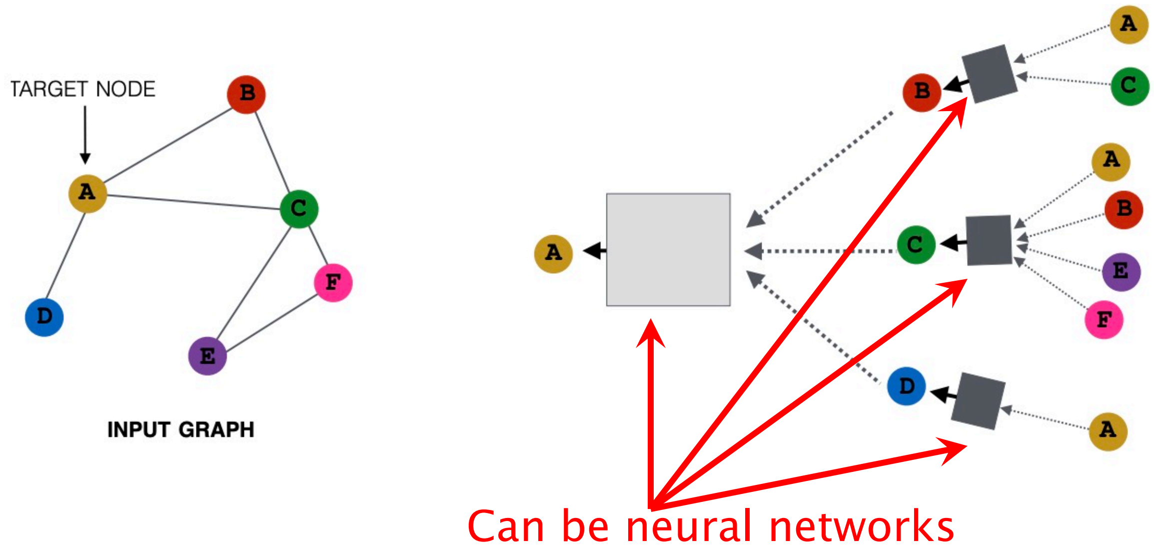
- We will study algorithms for efficiently constructing suffix arrays and suffix trees
- Many other interesting problems (edit distance, Lempel-Ziv compression, approximate string matching, alignment, etc.)



GRAPH NEURAL NETWORKS

Graph Neural Networks (GNNs)

Source: https://snap-stanford.github.io/cs224w-notes/machine-learning-with-networks/graph-neural-networks

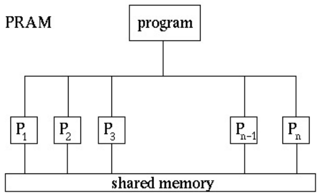


- Traditional neural networks have a fixed topology, but in GNNs the topology is the graph
  - Repeatedly pass messages to neighbors, and aggregate messages received to update node
  - Each node has a different computation graph!
  - Many different graph neural networks, based on how they pass and aggregate messages
  - We will study some high-performance GNN systems

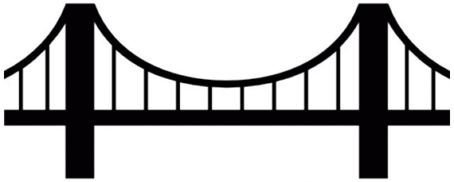
Relevant Topics Not Covered

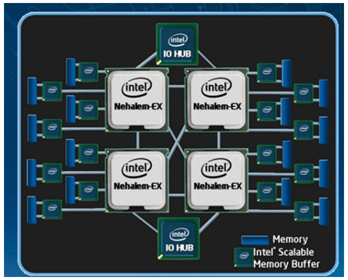
- GPUs, other accelerators, and special-purpose hardware
- Computer networking
- Linear and integer programming
- Optimizing NP-hard problems
- Succinct data structures
- Computational geometry
- Transactional memory
- Performance of different programming languages

Summary



O(n log n)
O(n)
O(log n)





- Lots of exciting research going on in algorithm engineering!
- Take this course to learn about latest results and try out research in the area