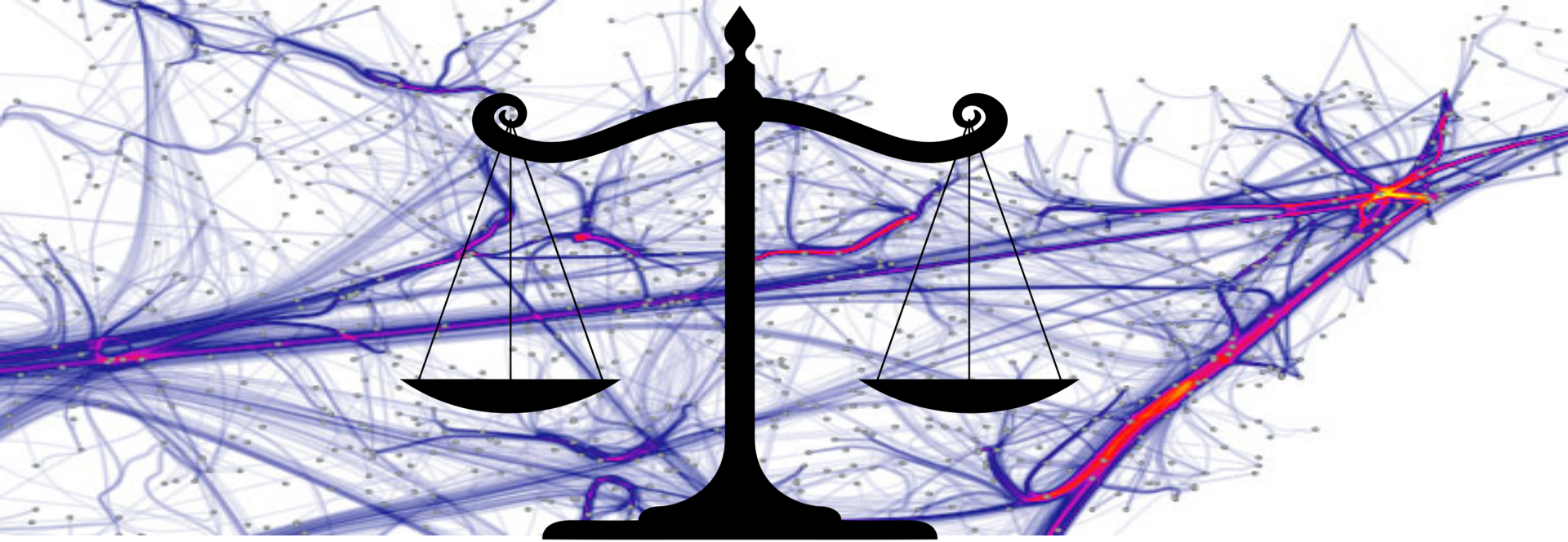


TRADE-OFFs IN LARGE GRAPH PROCESSING: REPRESENTATIONS, STORAGE, SYSTEMS AND ALGORITHMS

Deepak Ajwani, Alessandra Sala, Marcel Karnstedt and Pat Nicholson

19th May 2015

Bell Labs
Alcatel·Lucent

Constantly Interacting on Different Domain …

SOCIAL

BIOLOGICAL

PHYSICAL

Bell Labs
Alcatel·Lucent

The Network of the Future … Essentially a Graph

Building an Integrated knowledge Network

- Mining integrated knowledge graph to derive **Insights**
  - Building from heterogeneous data sources
    1. People interactions
    2. Objects network, i.e. physical things
    3. Information network from digital documents

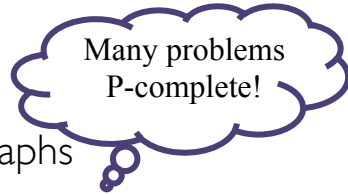
  - Navigate data with richer context
    - Connecting physical and digital worlds with semantic meaning

By-product: even bigger graphs with million of attributes to manage

Bell Labs
Alcatel·Lucent

The "Theoretical" Limits in Manipulating Large-Scale Graphs

- Random accesses in graph computations manifest in multiple dimensions:
  - I/O Overhead: Mismatch of logical graph structure with physical storage
  - Distribution and communication bottleneck: no good partitions for many real-world graphs
  - Parallelism constraints: even basic traversal problem do not have work efficient PRAM algorithms

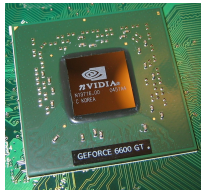


- Need to support variety of graph problems with different storage access pattern and latency requirements
  - E.g., pattern matching, traversal, batch/iterative computations, etc.

- Different domain graphs presents fundamentally different topological structures
  - Customized approaches required to satisfy real-time application constraints

No one size fits all !!!

Plethora of design choices: Architecture, Graph Systems, Algorithms!



ST
xXL

Pregel
GraphLab

Galois

Spark

GRAPH
500

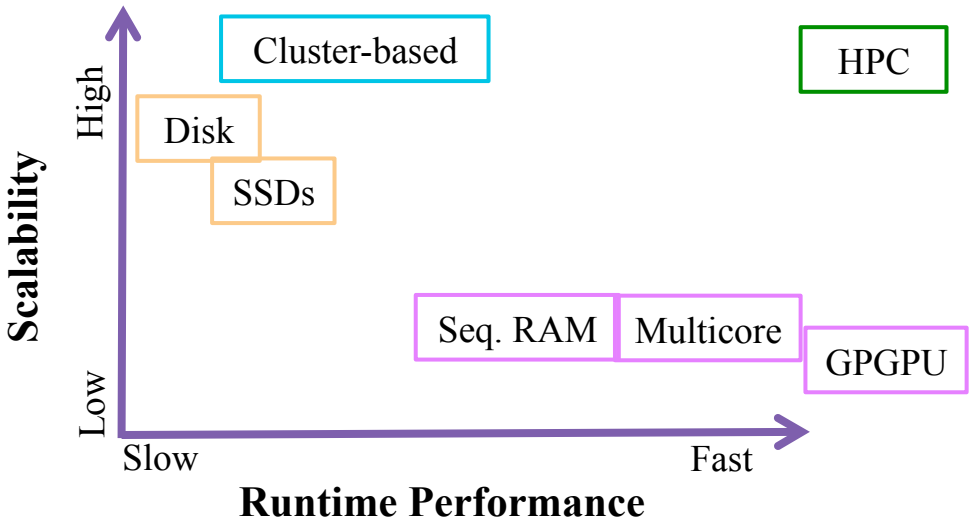
Focus Of This Tutorial

Trade-offs to be aware of, when you design your graph processing solution: Architecture, System, API, Algorithms and Data Structure

- Scalability vs. running time for Architecture/System
- Expressivity vs. development time for System/API
- Parallelism vs. workload for Algorithm/Implementation
- Parallelism vs. robustness for Algorithm/Implementation
- Space vs. accuracy vs. time for Data Structure/Storage

Random Memory Accesses Do Not Scale

- Ideally: graph systems and architectures that would support
  - Very fast generic access patterns for sub-graph mining, traversal queries, etc.
  - Unlimited scalability: no restriction to the graph scale

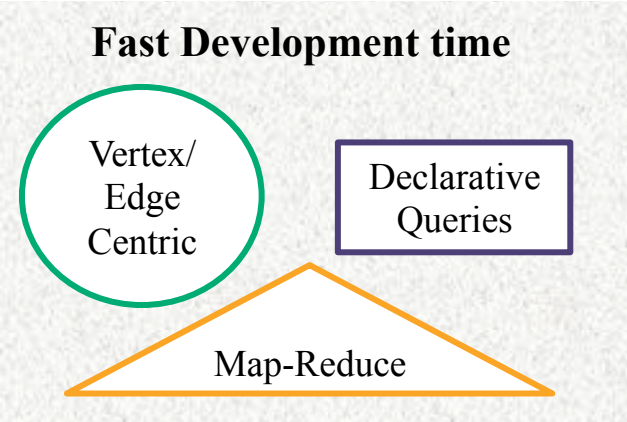


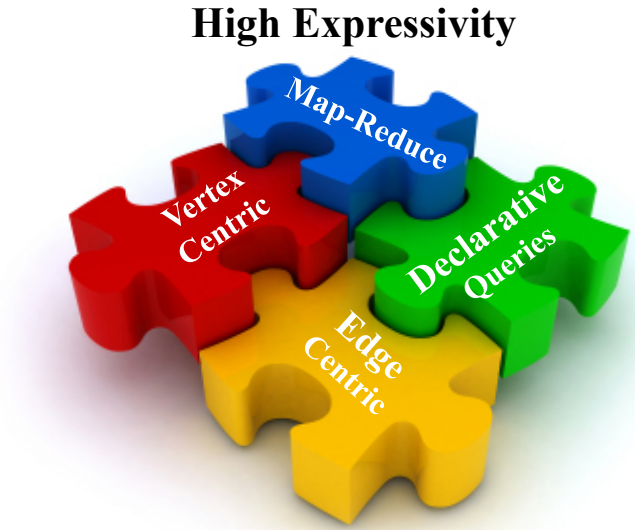
Trade-off:
Scalability vs Running Time

High Development Effort For Complex Problems

- Ideally: as simple as writing sequential code and the system will automatically distribute, parallelize and externalize it
  - Support arbitrarily complex graph algorithms and data-structures
  - Little development and maintenance effort

Trade-off:
Expressivity vs Development Time



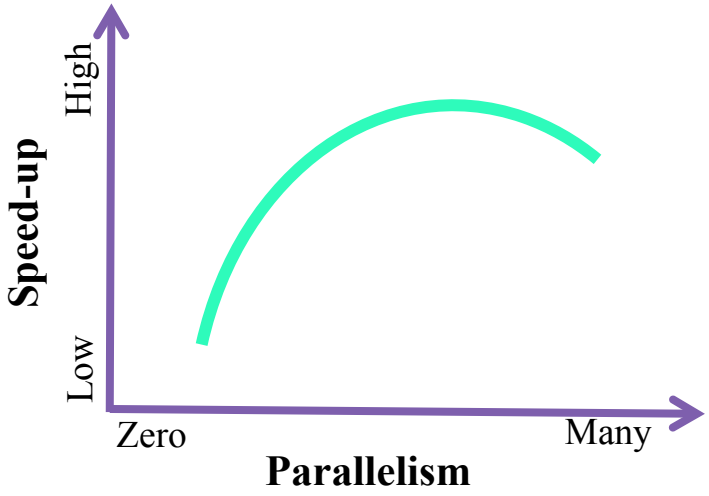


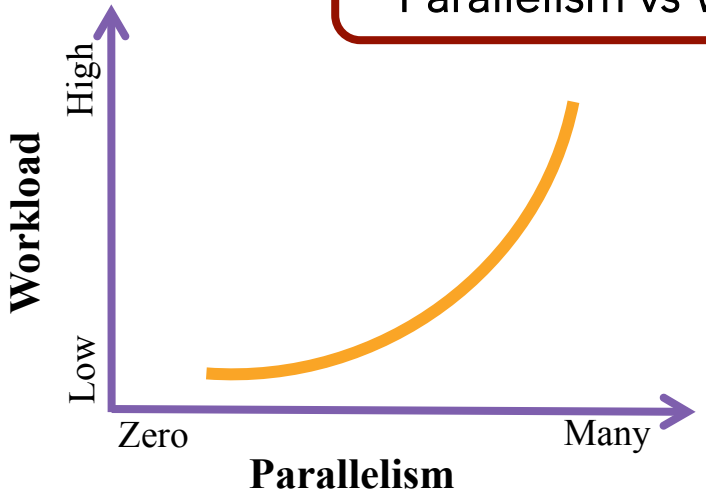
Bell Labs
Alcatel·Lucent

More Parallelism Can Increase Workload

- Ideally: the design and the implementation of graph algorithms should
  1. Leverage as much parallelism as architecture offers
  2. Perform as little work as the fastest sequential algorithm
  3. No restriction on the scale of the graph

Trade-off:
Parallelism vs Workload





Parallelism Increases Non-Determinism

- Ideally: no result variability across parallel executions due to system and architectural features
  - Timing of private cache flushes and relative speed of different processors should not impact application results
  - Exploit parallelism with speedup comparable to fastest parallel implementation

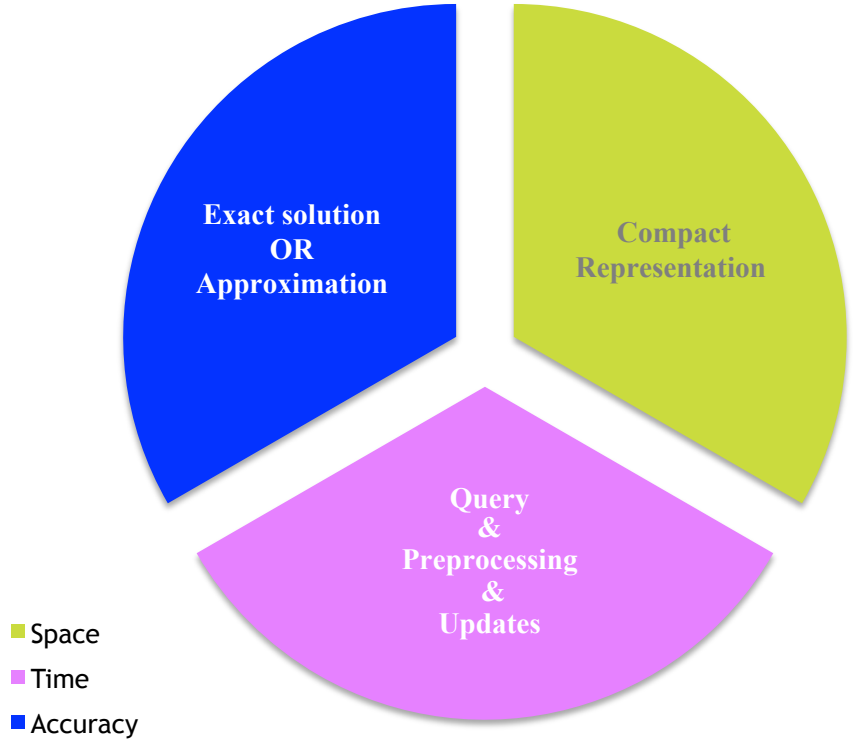
Trade-off:
Parallelism vs Robustness

Conquer sub-computations with:
- No aggregation
- Commutative aggregation

Accuracy & Speed Require Spacious Memory

- Ideally: graph representations should support
  - Space: compress raw graph and low space indexes
  - Time: fast pre-processing, queries and update time
  - Accuracy: highly accurate results

Trade-off:
Space vs Time vs Accuracy



Bell Labs
Alcatel·Lucent

This Tutorial Is Not About …

- Survey of graph processing systems
  - Tutorial: "Systems for Big Graphs," by Khan and Elniteky, VLDB 2014
  - "A Survey of Parallel Graph Processing Frameworks," by Doekemeijer and Varbanescu, PDS report
- In-depth tour of any one graph processing system/database
  - Tutorial: "Managing and mining Large Graphs: Systems and Implementations," by Wang, Shao and Xiao, SIGMOD 2012
- Graph Databases
  - Tutorial: "Database Techniques for Linked Data Management," by Harth, Hose and Schenkel, SIGMOD 2012
- Distributed SPARQL Engines and RDF-Stores
  - Tutorial: "Graph Data Management Systems for New Application Domains," by Cudré-Mauroux and Elnikety, VLDB 2011
  - Tutorial: "Cloud-based RDF data management," by Kaoudi and Manolescu, SIGMOD 2014
- Other NoSQL Systems
  - Tutorial: *"Modern Database Systems," by* Mohan, VLDB 2013
- HPC Systems for Graphs
  - Web: http://www.graph500.org
  - Survey: "Graph Analysis with High- Performance Computing," by Hendrickson and Berry, in "Combinatorics in Computing," 2008

This Tutorial Is Not About …

- Streaming Graph Algorithms
  - Survey: "Graph Stream Algorithms: A Survey," by McGregor, in ACM SIGMOD Record, 2014
- Dynamic Graph Algorithms
  - Survey: "Dynamic Graph Algorithms," by Demetrescu, Eppstein, Galil and Italiano, chapter in Book "Algorithms and theory of computation handbook," Chapman & Hall/CRC, 2010
- I/O-efficient Graph Algorithms
  - Survey: "Design and Engineering of External Memory Traversal Algorithms for General Graphs," by Ajwani and Meyer, chapter in "Algorithmics of Large and Complex Networks", Springer, 2009
  - Book: "Algorithms for Memory Hierarchies,", Meyer, Sanders and Sibeyn (eds.), Springer, 2003
- Parallel Graph Algorithms
  - Book: "Parallel Graph Algorithms," Bader (ed.), Chapman and Hall/CRC, 2015
  - Tutorial: "Mining Billion-Scale Graphs: Patterns and Algorithms," by Faloutsos and Kang, SIGMOD 2012
- Graph Mining and Management Algorithms
  - Book: "Managing and Mining Graph Data," Aggarwal and Wang (eds.), Springer, 2010
- Data Structures for Shortest Path Queries
  - Survey: "Shortest Path Queries in Static Networks", Christian Sommer, 2013

Outline

PART I

Scalability vs. Running Time - Architecture and System

Expressivity vs. Development Time - System and API

PART II

Parallelism vs. Workload – Algorithm and Implementation

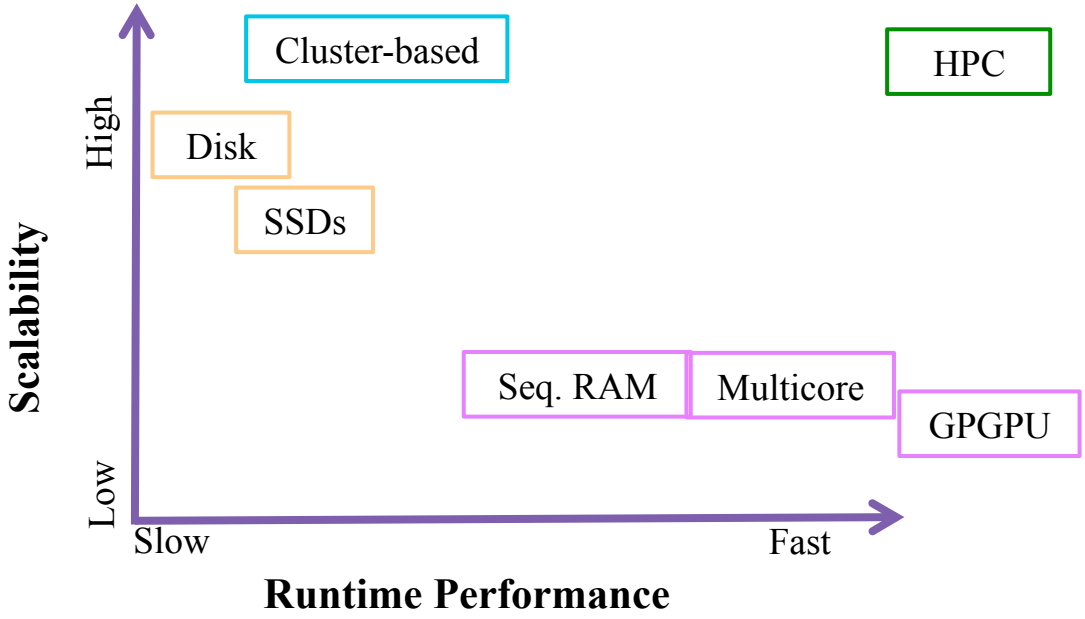
Parallelism vs. Robustness – Algorithm and Implementation

PART III

Space vs. Query Time vs. Approximation Quality – Storage and Data Structures

Hands-on Session: Neo4J and Galois

Diversity Of Architecture For Graph Processing

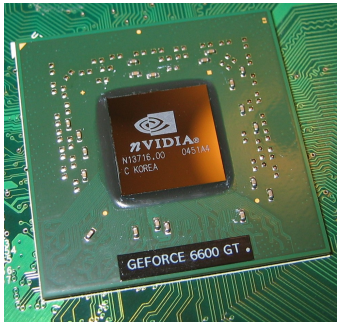


- In-Memory
- On Disk
- Distributed
- Highly Customized SuperComputer

Bell Labs
Alcatel·Lucent

GPGPU: Gunrock

- GPGPUs offer thousands of parallel processing units,
  - Not many graph algorithms can leverage its fine-grained parallelism
  - Requires considerable expertise to write efficient code for processing graphs with GPGPUs
  - Library approach as opposed to systems approach
  - Ready-to-use graph algorithms already implemented on GPGPU (Gunrock from UC Davis)



- Scalability:
  - Subgraphs needs to fit in GPU memory which is much less than CPU memory
    - Moving the graph data between CPU and GPU memory slow

- Running time:
  - Unless embarassingly parallel algorithm, thousands of processing units typically speed-up the algorithm by a factor between one and hundred

Multicore: Ligra, Galois

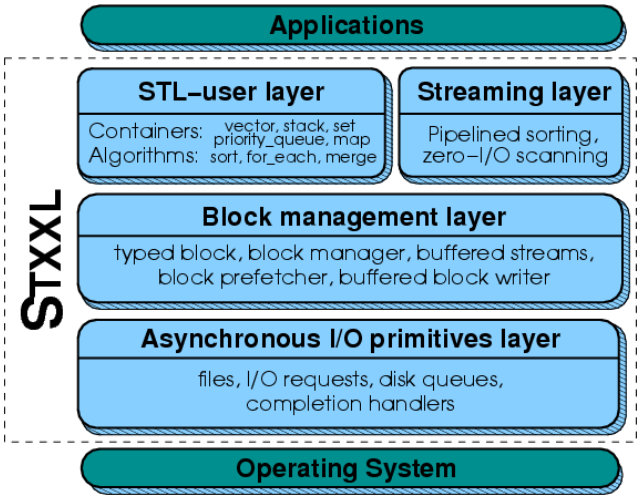
- Multi-cores with coarse-grained parallelism and shared memory are better suited to parallelize graph algorithms
  - Graphs are irregular and graph algorithms have a lot of branching instructions and random accesses
  - Systems such as Ligra and Galois define low-level primitives to support graph algorithms for multicores

- Scalability:
  - Graph needs to fit in the main memory restricting the size of graphs

- Running time:
  - Results in very fast parallel implementations
  - Typical multi-core architectures have 2-64 cores, but the number is increasing

Disk: STXXL, GraphChi

- Large body of literature to design and implement I/O-efficient algorithms and data structures
  - Replace random accesses by structured accesses to alleviate I/O-bottlenecks
  - STXXL: Library for I/O-efficient data structures
  - GraphChi: System for quickly deploying vertex-centric algorithms I/O-efficiently



- Scalability:
  - Can process graphs as long as graph and its intermediate copies can fit on disk
- Running time:
  - Web-graph with many billions of edges can be processed in less than an hour on a PC

SSDs: PrefEdge

- SSDs provide faster random reads and provide I/O parallelism
  - PrefEdge: A prefetcher for graph algorithms that prefetches requests to derive maximum throughput from SSDs
  - Write bandwidth starts matching read bandwidth, but write latencies are still significantly more than read latencies
  - Graph algorithms often involve updating the state of nodes/edges: Random write accesses
- Scalability:
  - SSDs with capacity in TBs readily available

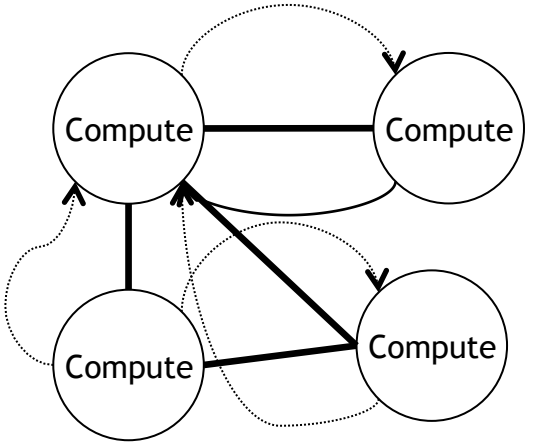
- Running time:
  - On a Twitter graph with around 1.6 billion edges, Dijkstra's SSSP algorithm with PrefEdge runs within a factor 5 of the in-memory, despite using only 15-20% of the main memory required
  - I/Os still remain the bottleneck

Distributed In-Memory Systems:

Pregel GraphLab

APACHE GIRAPH

- Restricted but intuitive computation model: bulk synchronous processing
  - "MapReduce for graphs", batch style
  - Typically vertex-centric, data exchanged along graph edges
  - Synchronous and asynchronous variants
- Scalability:
  - Designed to support massive scale for iterative processing
  - Replication for fault-tolerance
- Running time:
  - Good performance for restricted computation model
  - Inter-machine communication over network becomes bottleneck
    - Performance strongly influenced by graph partitioning



Bell Labs
Alcatel·Lucent

Distributed In-Memory Persistence: Trinity, GraphX, Horton+, …

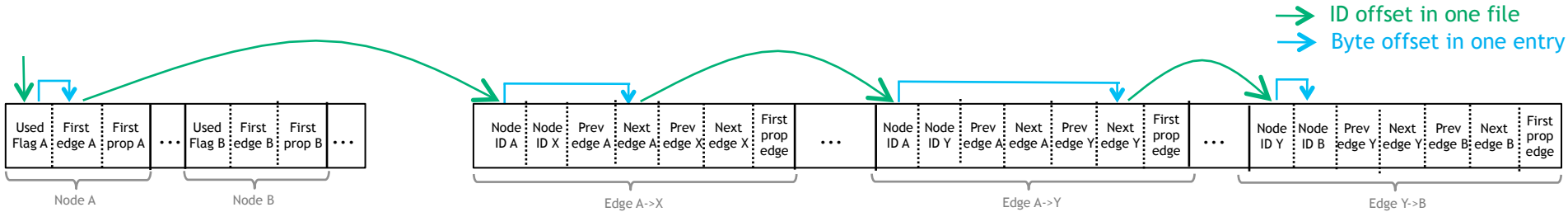
- Similar approach to distributed batch-style systems, but **keep** graph in memory
  - Supports interactive & online graph querying
  - Trinity: distributed "shared" memory cloud
  - GraphX: graph support on top of SPARK's resilient distributed data structures
  - Horton+, Green-Marl, Grace: in-memory graph systems (more or less) influenced by DBMS

- Scalability:
  - Same as distributed batch-style systems

- Running time:
  - Avoid reloading of graph from disk
  - Good performance also for database-like querying

Graph Databases:

- Persistent storage solutions with native graph support or independent graph layer
  - Users think and query directly in graphs, no translation to tables or similar
  - Follow core principles from "traditional" DBMS, but end in different **trade-offs**
  - Strong support for persistency & one-shot queries, but typically limited in graph mining
- Scalability:
  - Often limited in distribution, some support NoSQL backends like Cassandra, HBase, etc.
- Running time:
  - Optimised for concurrency & (*local*) traversals that require several joins in an RDBMS



Node file: direct access via node ID

Edge file: direct access via edge ID, doubly linked list

Bell Labs
Alcatel·Lucent

High Performance Computing (HPC)

- HPC systems used for many scientific applications requiring fast and scalable graph processing
  - Specialized expensive hardware
  - Difficult to develop and optimize code, non-trivial maintenance of code-base and the system
  - Many implementations of BFS and SSSP on different supercomputers
    - BFS and SSSP on Cray MTA-2, Streaming graph analytics on Cray XMT
- Scalability:
  - HPC systems typically have very large distributed or "shared" memory
- Running time:
  - Graph500 Benchmark evaluates the suitability of different supercomputers for basic graph algorithms:
    - IBM BlueGene/Q with more than 1.5 million cores can traverse more than 23 trillion edges per second on a benchmark graph with more than 2 trillion vertices and 32 trillion edges

Summary: Scalability vs. Running Time

- For fast graph traversal, one can leverage shared-memory parallelism
  - Multicores with shared-memory
  - Massively multi-threaded shared memory supercomputers

- For scalable graph computing, one has to go beyond the main memory constraint
  - I/O-efficiency on HDDs and SSDs
  - Distributed In-memory (with persistence)
  - Graph Databases

- HPC architectures provide both scalability and fast traversal queries, but they are costlier and it is difficult to maintain and optimize code for them

Outline

PART I

Scalability vs. Running Time - Architecture and System

Expressivity vs. Development Time - System and API

PART II

Parallelism vs. Workload – Algorithm and Implementation

Parallelism vs. Robustness – Algorithm and Implementation

PART III

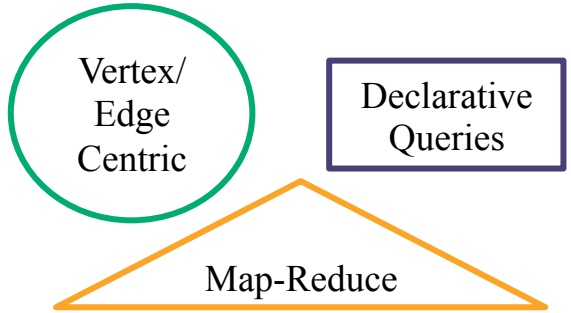
Space vs. Query Time vs. Approximation Quality – Storage and Data Structures

Hands-on Session: Neo4J and Galois

Restricting Computations to Reduce Development Time

- Many graph systems restrict the set of supported graph algorithms
  - Vertex-centric computation, Edge-centric computation, Gather-apply-scatter, Bulk Synchronous Parallel, Map-reduce
- Many Database APIs restrict the set of queries that can be expressed

- Results in significant reduction in development time
  - Only define a simple update function for vertex-centric computation without worrying about load balancing among processors, graph partitioning, scheduling of tasks on processors, fault tolerance etc.



Expressivity vs. Development Time

- Restricting computation model also reduces the flexibility of algorithms, limiting the design choices
  - Can lead to significant loss in efficiency as the fastest algorithms can't be deployed and sub-optimal algorithms have to be used
- APIs that provide fast and simple access to data often restricted to declarative queries, while more flexible APIs require more time to express the queries

- **Walk through with three examples:**
  1. SSSP using GraphLab vs. Galois
  2. I/O-efficient BFS using GraphChi vs. STXXL
  3. Pattern matching, traversal and pagerank using Cypher vs. Gremlin vs Java API

Example 1: Single Source Shortest Path (SSSP)

Problem: Given a vertex s, compute length of the shortest path from s to any vertex

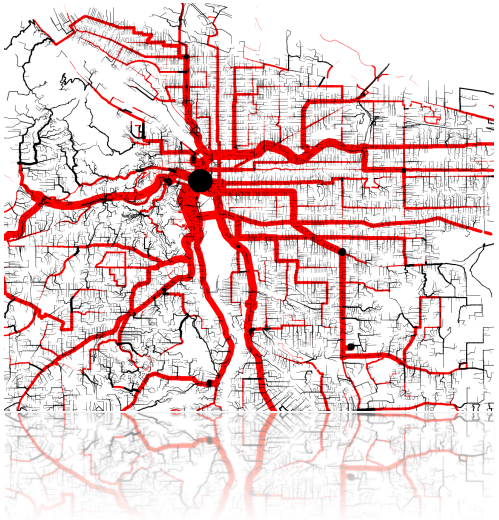
- Sequential algorithm: Dijkstra's algorithm
  - *Complexity*: O((n + m) log n) or O(n log n + m) with a Fibonacci heap

- Vertex-centric algorithm: based on Bellman-Ford's algorithm
  - *Complexity*: O((n+m)L/p) time for p < n

    where L is the maximum number of edges in any shortest path: can be O(n)

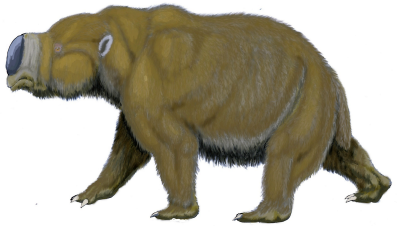
- Parallel algorithm: Delta-stepping (described later)
  - *Complexity*: O((n + m)/p) time* for p < (n + m)/$\log^2$ n

    *Assuming a random graph with edge probability (avg. degree)/n, and random edge weights



Example 1: Single Source Shortest Path with GraphLab

GraphLab: A system for parallel/distributed graph computation that is mostly restricted to vertex-centric computation



- Expressivity:
  - Vertex-centric single source shortest path
    - Each vertex v stores a distance, dist(v): initially dist(s) = 0, and infinity otherwise
    - Iterate till convergence:

      For each vertex v in the graph:

      For each neighbour u of v where edge (v,u) has length l

      If dist(u) + l is smaller than dist(v) then set dist(v) = dist(u) + l
  - Terminates after O(L) rounds, where L is the max number of edges in any shortest path
- Development time:
  - Easy to write vertex-centric algorithms
  - Implementing data-structures and algorithms that do not adhere to the basic vertex-centric model require significant development time

Example 1: Single Source Shortest Path with Galois

Galois: A graph system that defines low-level primitives to parallelize iterating through vertices and edges in the code

- Expressivity:
  - No restriction on the set of algorithms that can be used
    - Can implement the fast Delta-Stepping algorithm for single source shortest path (described later)
  - Results in parallel implementations that are among the fastest on shared-memory multicore systems

- Development time:
  - If an app is not already available, significant development time
  - Build from scratch from low-level primitives, including data structure accesses

Example 2: Breadth-First Search with GraphChi

GraphChi: Graph system for I/O-efficiency that uses a parallel sliding window method to execute many graph mining and machine learning algorithms on just a PC

- Expressivity:
  - Restricts to vertex-centric computation
    - Graph algorithms that require global data structures and control over the graph's disk layout are not supported
  - Good for small-diameter networks arising in social networks and machine learning applications, but results in poor performance for moderate-to-large-diameter graphs
    - Vertex-centric breadth-first search requires as many iterations as the graph diameter

- Development time:
  - Very fast, ideal for quick prototyping
  - No need to worry about file-system, I/O-layer, creating balanced shards etc.

Example 2: Breadth-First Search with STXXL

STXXL: Library that provides I/O-efficient data structures such as stacks, vectors, matrices, priority queues with a STL interface

- Design and implement I/O-efficient algorithms using these data-structures; structured memory accesses to replace random accesses

- Expressivity:
  - Arbitrary graph algorithms can be expressed as it provides complete control over the storage volumes
  - Can implement I/O-efficient breadth-first search algorithm for large diameter graphs that requires its own disk layout of graph

- Development time
  - Not as simple as replacing the priority queue in Dijkstra's algorithm by I/O-efficient priority queue
  - I/O-efficient algorithms need to be carefully engineered
  - Many months to implement a pipelined Breadth-First Search implementation

Example 3: Matching and Traversal Queries via different APIs

- Cypher: declarative query language of Neo4j graph database
  - Like SQL, user does not (have to) influence at all **how** data is computed
  - Extremely fast entry into data, no code required at all
  - Usage of graph algorithms restricted to provided functions

- Gremlin: domain-specific language for traversing graphs
  - User describes *how* nodes and edges are traversed
  - Needs more training, but provides more control
  - Available in several systems that support Blueprints property graph data model

- Java API: A native API for Neo4J
  - Rich expressivity in specifying queries
  - Need to develop Java code for each query

Example 3: Simple pattern matching

- Cypher: match – where – return
  - Makes use of cost-based optimiser (latest Neo4j version) & indexes
  - Hands on! Later in this tutorial

```
match (n:users {name: 'Tom'})-[r:KNOWS]->v where r.date =~ '2015.*' return v
```

- Gremlin: nodes – edges - filter

```
g.idx(users)[[name:'Tom']]
  .outE('KNOWS').filter{it.date.matches('2015.*')}.inV.map
```

- API-based: supports methods for "everything manually"

```
Node n = graphDb.findNodes(usersLabel, "name", "Tom");
Iterable<Relationship> it = n.getRelationships(knowsType);
it.next().getProperty("date")…;
```

Example 3: Stack Overflow Discussion

41

19

I'm starting to develop with Neo4j using the REST API. I saw that there are two options for performing complex queries - Cypher (Neo4j's query language) and Gremlin (the general purpose graph query/traversal language).

Here's what I want to know - is there any query or operation that can be done by using Gremlin and can't be done with Cypher? or vice versa?

Cypher seems much more clear to me than Gremlin, and in general it seems that the guys in Neo4j are going with Cypher. But - if Cypher is limited compared to Gremlin - I would really like to know that in advance.

neo4j graph-databases cypher gremlin

share improve this question

edited Oct 30 '13 at 16:40
Ricardo
274 ● 4 ● 16

asked Dec 11 '12 at 17:04
Rubinsh
1,907 ● 2 ● 17 ● 28

add a comment

5 Answers

active oldest votes

27

For general querying, Cypher is enough and is probably faster. The advantage of Gremlin over Cypher is when you get into high level traversing - in Gremlin, you can better define the exact traversal pattern (or your own algorithms) whereas in Cypher the engine tries to find the best traversing solution itself.

Example 3: Path Traversal: BFS, DFS, Shortest Paths

- Cypher: possible, but little control – can result in poor performance

```
match n-[]-x where … return allShortestPaths(n,x)
match p = n-[*..6]-x where … return p
```

- Gremlin: navigational character allows more fine-grained control

```
g.v(1).as('x').outE.gather.scatter.inV.loop('x'){it.loops < 7}{true}.path
g.v(1).as('x').outE.inV.loop('x'){it.loops < 7}{true}.path
val shortestPath = tmpList.sortBy(_.distance).head
```

- API-based: full flexibility in how (order, weights, …)
  - Neo4j additionally offers very flexible Traversal API

Bell Labs
Alcatel·Lucent

Example 3: PageRank

- Cypher: not possible
  - One-shot query character contradicts with global graph algorithms like PageRank

- Gremlin:
  - Possible in 12 lines of code, see slide 55
    http://www.slideshare.net/slidarko/gremlin-a-graphbased-programming-language-3876581

- API-based: many ways of doing it, full flexibility - and burden

Summary: Expressivity vs. Development time

- Some graph systems restrict the computation paradigm
  - This is often a severe restriction, resulting in poorer design choice and a very significant loss in efficiency
  - But it makes the code deployment significantly faster and the APIs easier to use

- Alternative graph systems that allow for arbitrary graph algorithms
  - Support graph implementations that have better running time
  - Requires considerable time for
    - Developing code from low-level primitives
    - Debugging the code (as it is far bulkier than the comparable code in restricted systems)
    - Maintaining the code

Outline

PART I

Scalability vs. Running Time - Architecture and System
Expressivity vs. Development Time - System and API

PART II

Parallelism vs. Workload – Algorithm and Implementation
Parallelism vs. Robustness – Algorithm and Implementation

PART III

Space vs. Query Time vs. Approximation Quality – Storage and Data Structures

Hands-on Session: Neo4J and Galois

Graph Algorithms Are Notoriously Hard To Parallelize Work-efficiently

- Many graph problems are not amenable to parallelism:
  - Depth-First search is inherently sequential
    - Lexicographic DFS is P-complete

- Even for many graph problems that can be parallelized,
  - Work-efficient parallel algorithms are not known, despite decades of research
    - Includes fundamental problems like single-source shortest paths, some flow problems, delaunay triangulation of mesh etc.
  - Achieving high parallelism often involves speculative computation, significantly increasing the total work
  - Higher levels of parallelism result in diminishing speed-up and in extreme cases, the algorithm can become even slower than a sequential algorithm

- Simple Illustration: Single-Source Shortest Path

Degree of Parallelism in Single-Source Shortest Path Algorithms

- Dijkstra Algorithm: Sequential approach with O(n log n + m) work
  - Relies on global priority queue data structure

- Bellman-Ford Algorithm: Easy to parallelize, but O(m n) work
  - Requires a very large number of parallel processing units to make it faster than Dijkstra's algorithm

- Δ-stepping algorithm for *tunable* trade-off between parallelism and work-load
  - A generic algorithm that results in Bellman-Ford when **Δ** > max{w(e)} and in Dijkstra when **Δ** < min{w(e)}

△-Stepping Algorithm for Single Source Shortest Paths

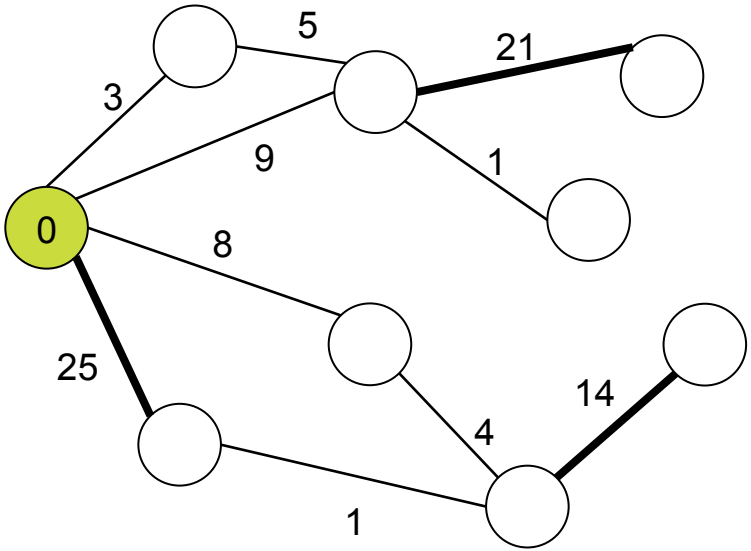
- Dijkstra's algorithm: Label-setting, settles the distance of one vertex per iteration
- Bellman-Ford: Label-correcting, relaxes edges from unsettled vertices also, resulting in speculative work

- **Δ**-stepping: Label-correcting, relaxes edges from "buckets" of unsettled vertices concurrently
  - Bucket i consists of vertices with tentative distance between **Δ**(i-1) and **Δ**i – 1
  - A vertex may have to be re-inserted if a shorter path to it is found
  - Higher the **Δ**, more the parallelism and higher speculative work

- For each vertex v in the graph, classify its edges as either "heavy" or "light"
  - Light edges have weight less than **Δ**
  - Observation: Only light edges affect the tentative distance of vertices in the current bucket

Δ - Stepping Algorithm: Illustration

Δ = 10



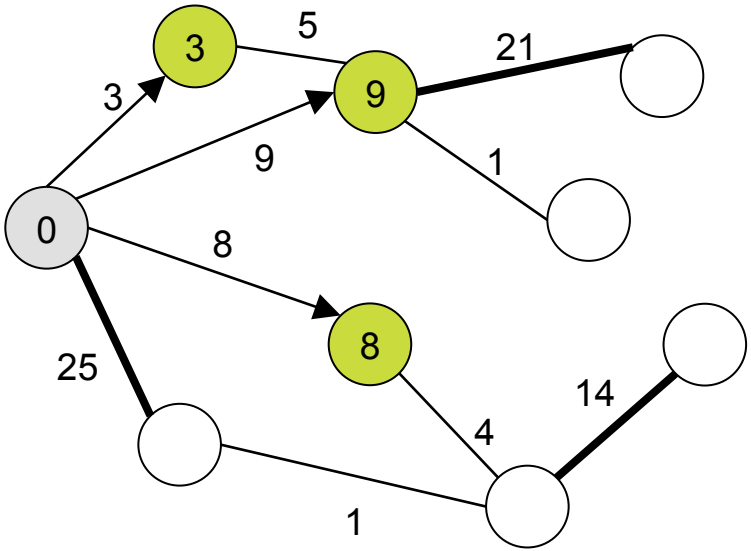
while this bucket is non-empty:
relax along light edges
relax along heavy edges
go to next bucket

Bell Labs
Alcatel·Lucent

Δ - Stepping Algorithm: Illustration

Δ = 10

while this bucket is non-empty:
relax along light edges
relax along heavy edges
go to next bucket



Processed

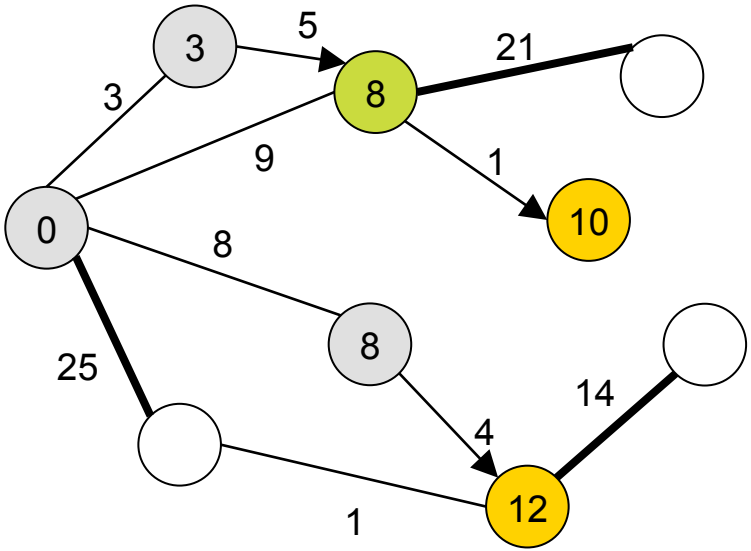
Estimated distance 0-9

Estimated distance 10-19

Estimated distance 20-29

Δ - Stepping Algorithm: Illustration

Δ = 10



while this bucket is non-empty:
relax along light edges
relax along heavy edges
go to next bucket

Processed

Estimated distance 0-9

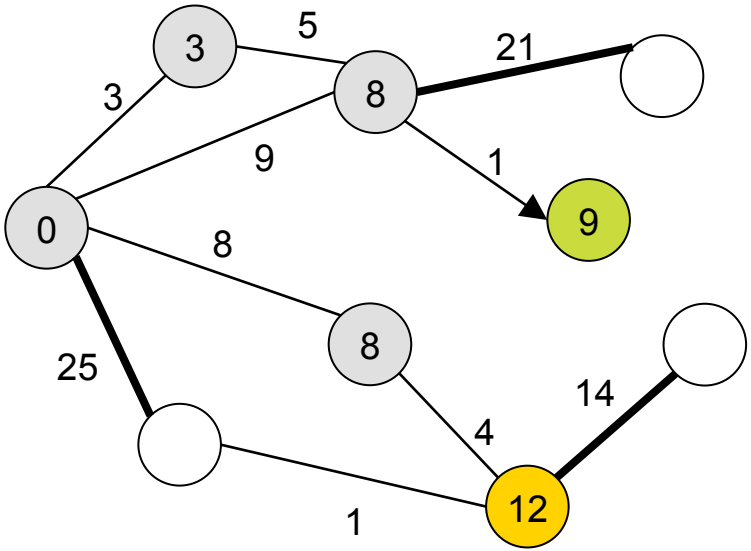
Estimated distance 10-19

Estimated distance 20-29

Bell Labs
Alcatel·Lucent

Δ - Stepping Algorithm: Illustration

Δ = 10



while this bucket is non-empty:
relax along light edges
relax along heavy edges
go to next bucket

Processed

Estimated distance 0-9

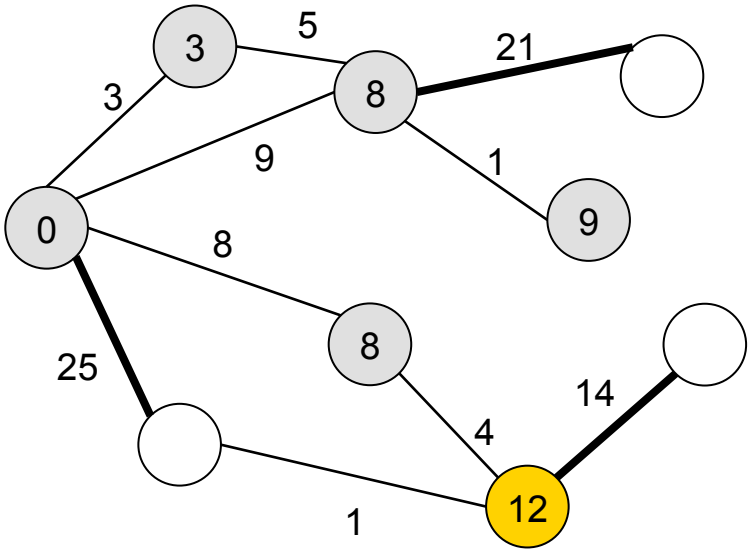
Estimated distance 10-19

Estimated distance 20-29

Bell Labs
Alcatel·Lucent

Δ - Stepping Algorithm: Illustration

Δ = 10



while this bucket is non-empty:
relax along light edges
relax along heavy edges
go to next bucket

Processed

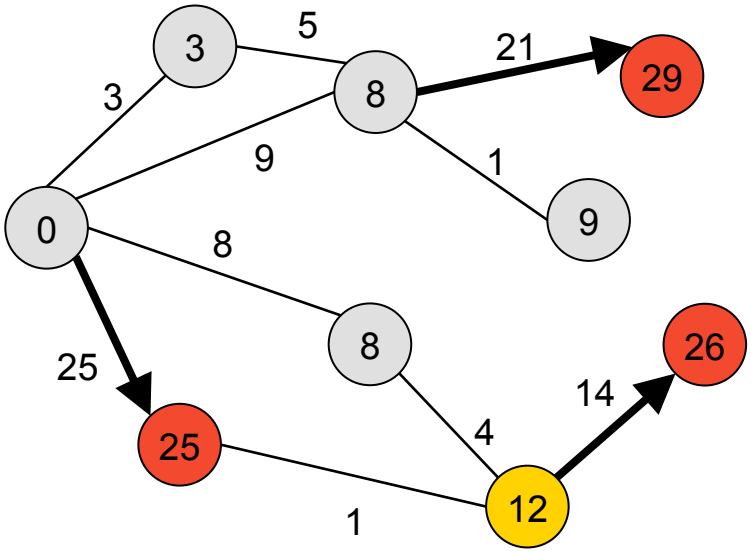
Estimated distance 0-9

Estimated distance 10-19

Estimated distance 20-29

Δ - Stepping Algorithm: Illustration

Δ = 10



while this bucket is non-empty:
relax along light edges
relax along heavy edges
go to next bucket

Processed

Estimated distance 0-9

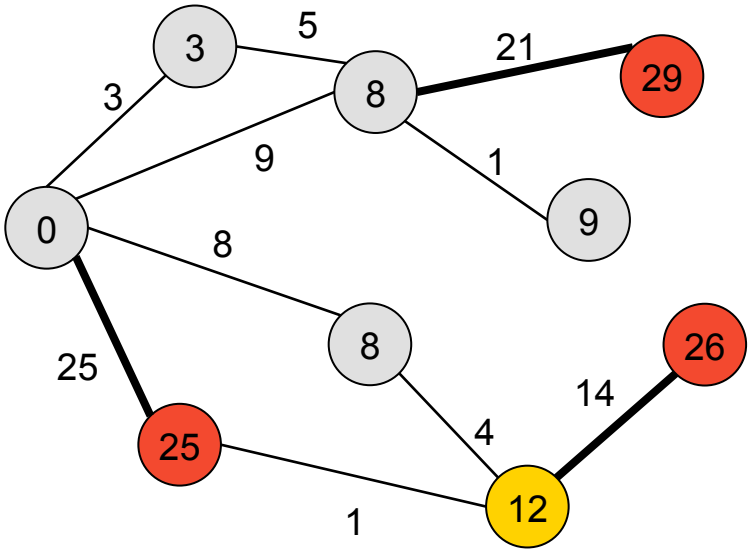
Estimated distance 10-19

Estimated distance 20-29

Bell Labs
Alcatel·Lucent

Δ - Stepping Algorithm: Illustration

Δ = 10



while this bucket is non-empty:
relax along light edges
relax along heavy edges
go to next bucket

Processed

Estimated distance 0-9

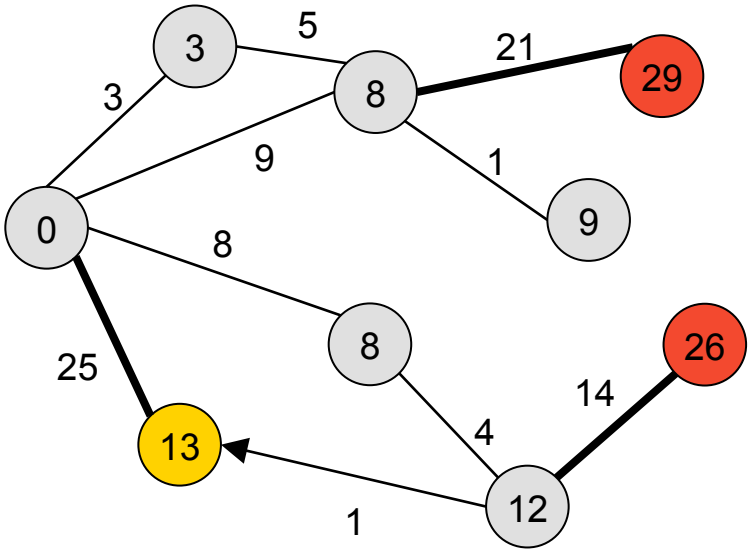
Estimated distance 10-19

Estimated distance 20-29

Bell Labs
Alcatel·Lucent

Δ - Stepping Algorithm: Illustration

Δ = 10



while this bucket is non-empty:
relax along light edges
relax along heavy edges
go to next bucket

Processed

Estimated distance 0-9

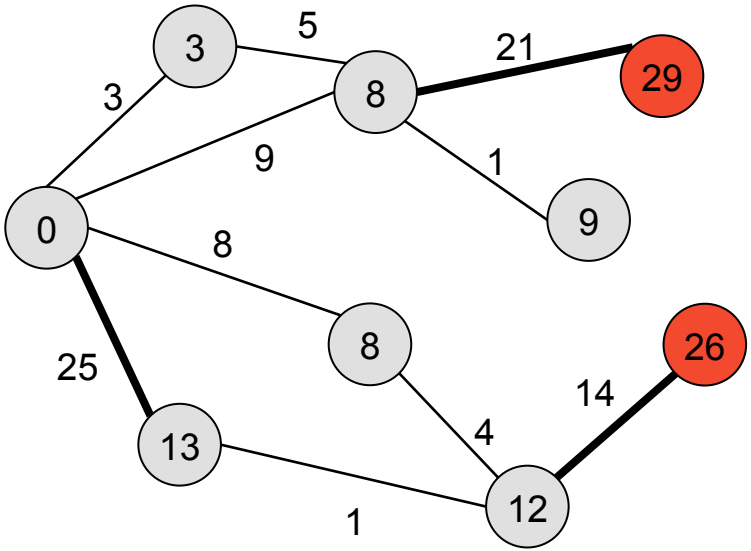
Estimated distance 10-19

Estimated distance 20-29

Δ - Stepping Algorithm: Illustration

Δ = 10

while this bucket is non-empty:
    relax along light edges
relax along heavy edges
go to next bucket



Processed

Estimated distance 0-9

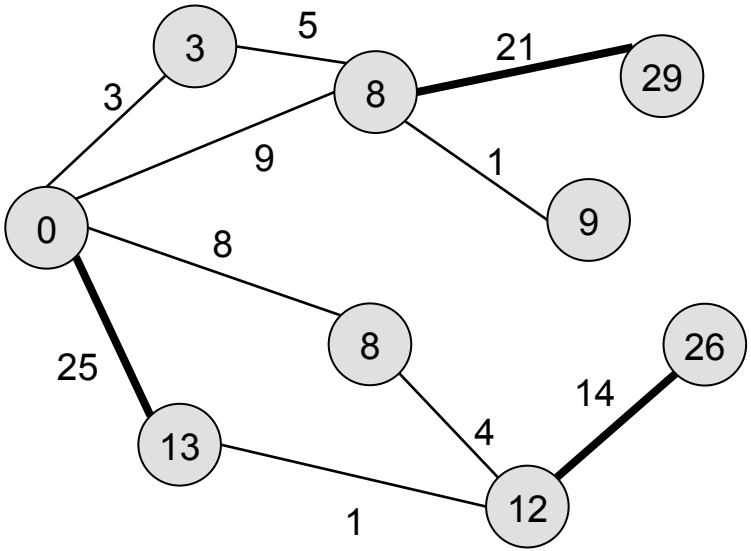
Estimated distance 10-19

Estimated distance 20-29

Bell Labs
Alcatel·Lucent

Δ - Stepping Algorithm: Illustration

Δ = 10



while this bucket is non-empty:
    relax along light edges
relax along heavy edges
go to next bucket

Processed

Estimated distance 0-9

Estimated distance 10-19

Estimated distance 20-29

Bell Labs
Alcatel·Lucent

Summary: Parallelism vs. Workload

- Graph algorithms amenable to parallelism to different degrees

- Carefully decide whether to use an algorithm with more work but parallelizable or one with less work but inherently sequential

- For some problems like Single Source Shortest Paths, parameterized algorithms exist to provide finer control on the trade-off between parallelism and workload
  - Delta-Stepping has been used for many different systems/architectures:
    - Galois for multi-cores
    - Parallel Boost Graph Library for distributed architecture
    - Gunrock for GPGPUs
    - Implementations known for Cray XMT supercomputer

Outline

PART I

Scalability vs. Running Time - Architecture and System
Expressivity vs. Development Time - System and API

PART II

Parallelism vs. Workload – Algorithm and Implementation
Parallelism vs. Robustness – Algorithm and Implementation

PART III

Space vs. Query Time vs. Approximation Quality – Storage and Data Structures

Hands-on Session: Neo4J and Galois

Parallelism vs. Robustness

- For many batched iterative graph computations, parallelization leads to a loss of control over the order in which vertices are processed
  - The order of processing vertices depends on architecture and system issues such as the speed of processors, other workloads on the processors, when the private caches flush (for multi-cores) and what cache-coherency protocols are used
- The vertex processing order can significantly impact the quality of the results and the running time

- Ideally, we want the results to be robust, i.e., independent of architecture and system issues
- Trade-off between parallel speed-up and robustness

- An example to illustrate this: Label propagation for community finding

Label Propagation for Community Detection

- Algorithm:

  Initialize Label(u) = Id(u) for all vertices u

  Repeat till convergence:

    For all vertices u:

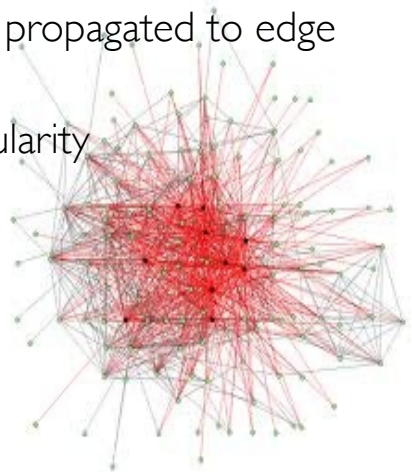
      Label(u) = Most frequent label among its neighbors and itself

  Vertices with same label form a community

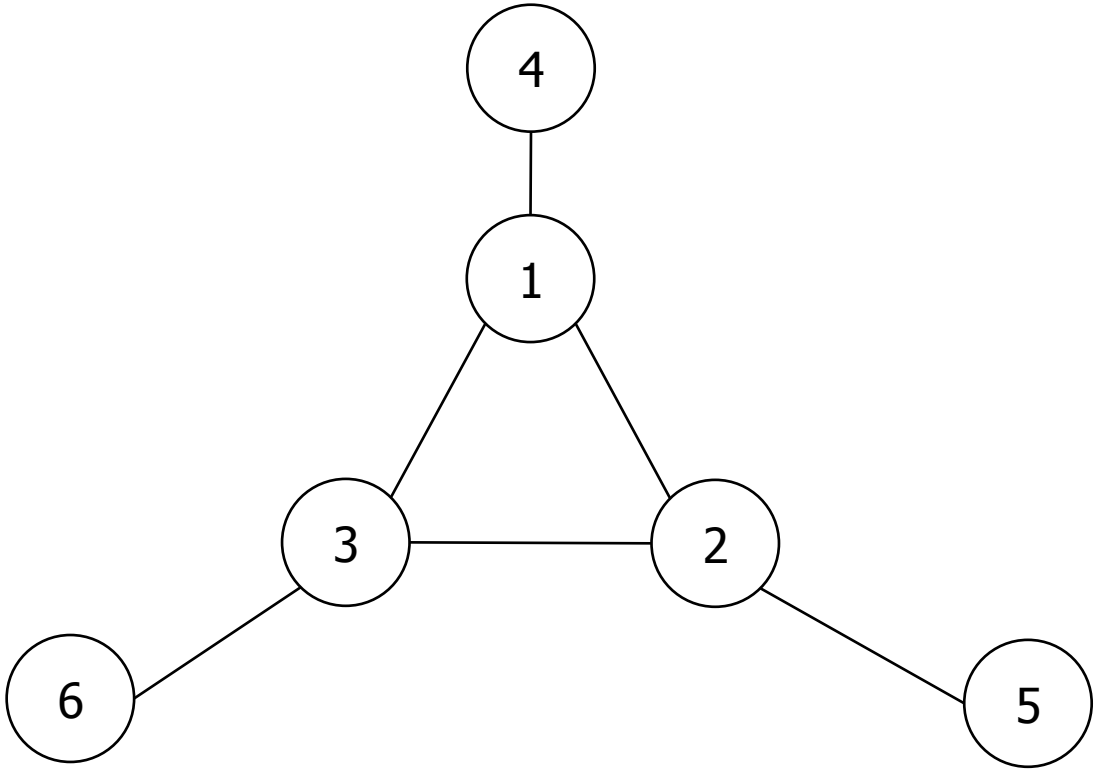
- Ties can be resolved arbitrarily or in favour of smaller label or higher label
- Labels of nodes are used as soon as they become available

Effect of Vertex Update Ordering

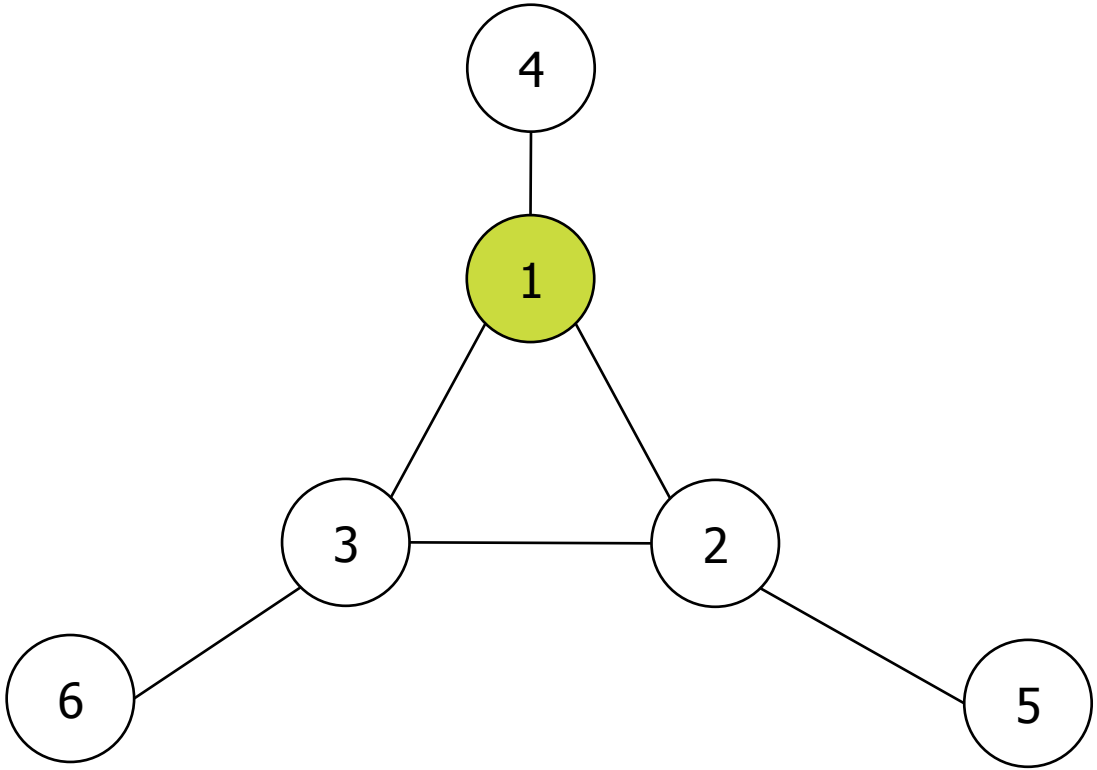
- Vertex update ordering has a big influence on the quality of communities as well as on the running time
  - Convergence depends on how fast labels mix and that depends on vertex update order
- Vertex update ordering in decreasing order of degree
  - Start from the core of network and move to the edge
  - Vertices at the core of the network form a giant community, which is then propagated to edge vertices
  - Very fast convergence, but results in one giant community with poor modularity
- Vertex update ordering in increasing order of degree
  - Start from the edge of network and move to the core
  - Vertices at the edge of the network get partitioned first and they percolate the communities inwards to the core
  - Slow convergence, but better communities



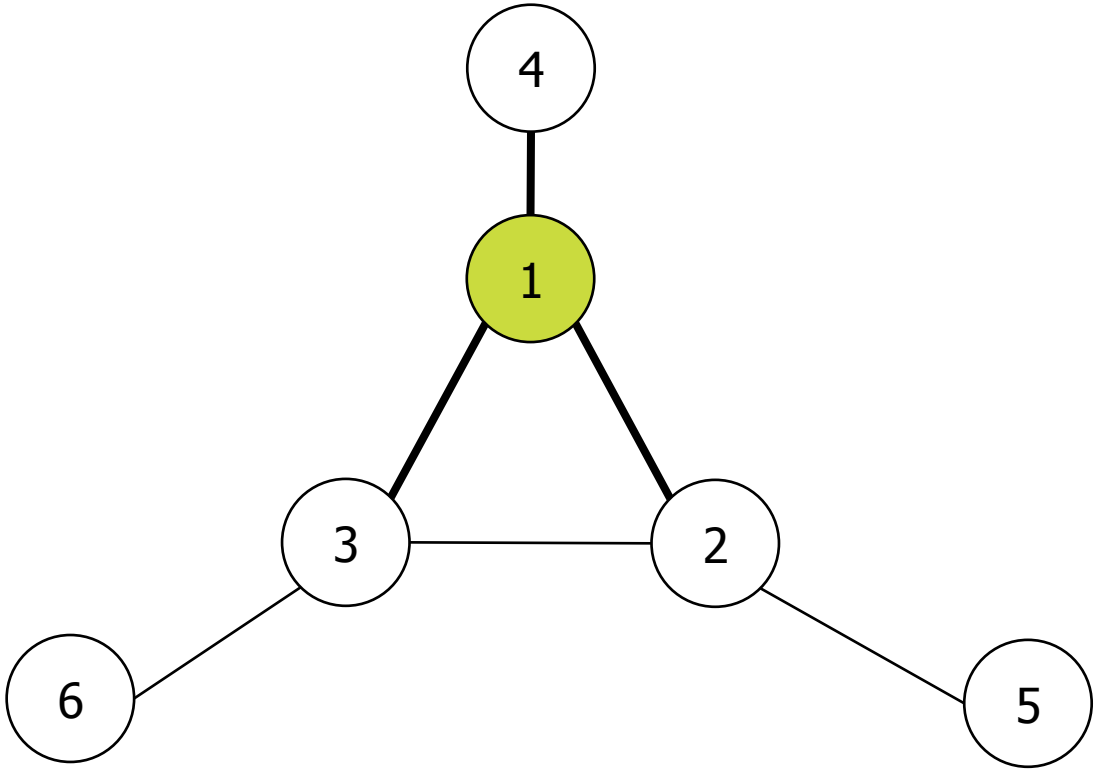
Label Propagation: Ordering from High to Low-degree Nodes



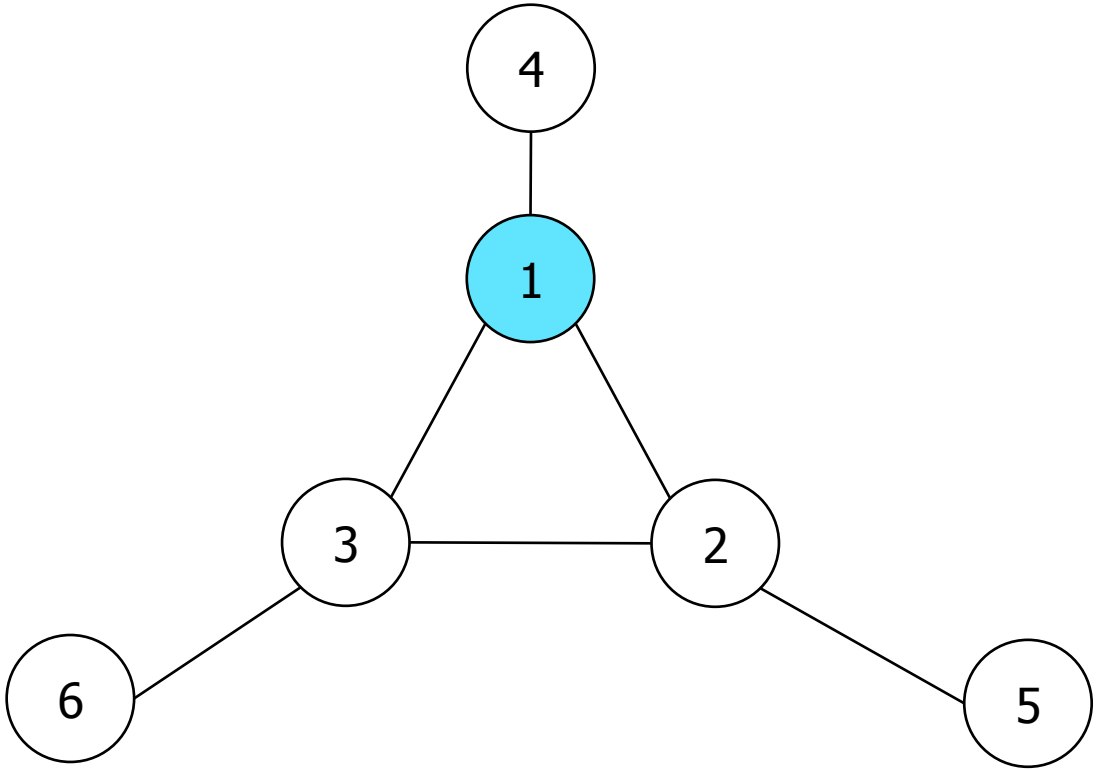
Label Propagation: Ordering from High to Low-degree Nodes



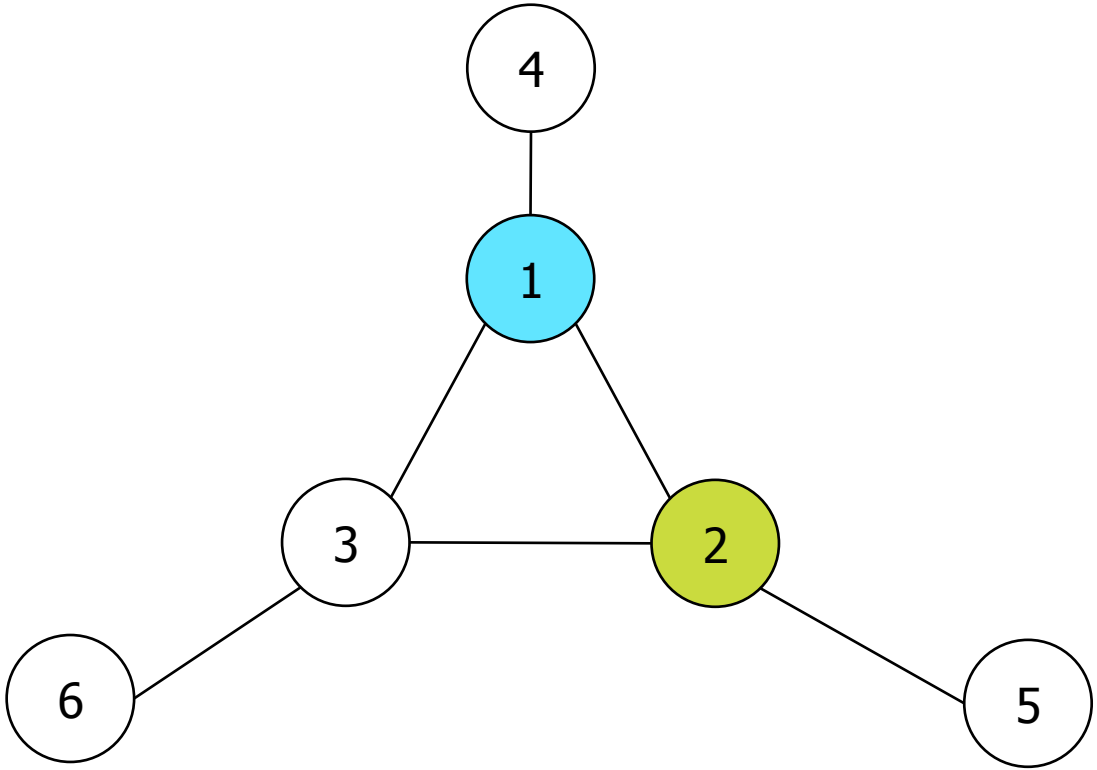
Label Propagation: Ordering from High to Low-degree Nodes



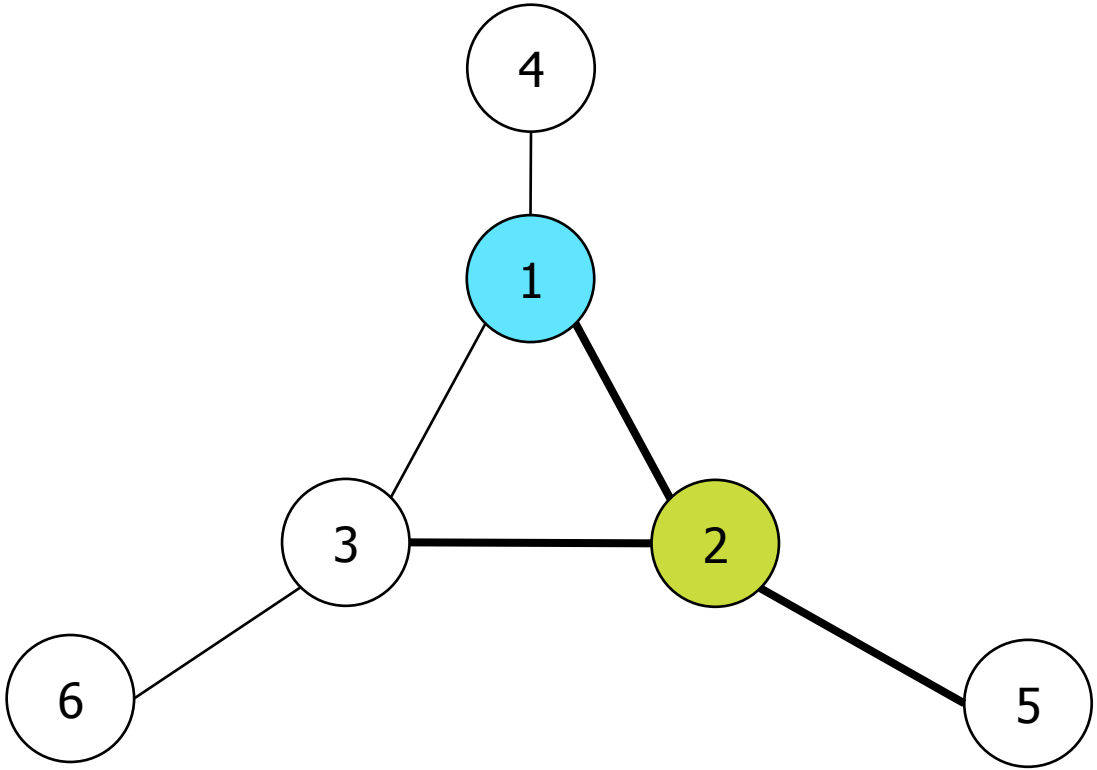
Label Propagation: Ordering from High to Low-degree Nodes



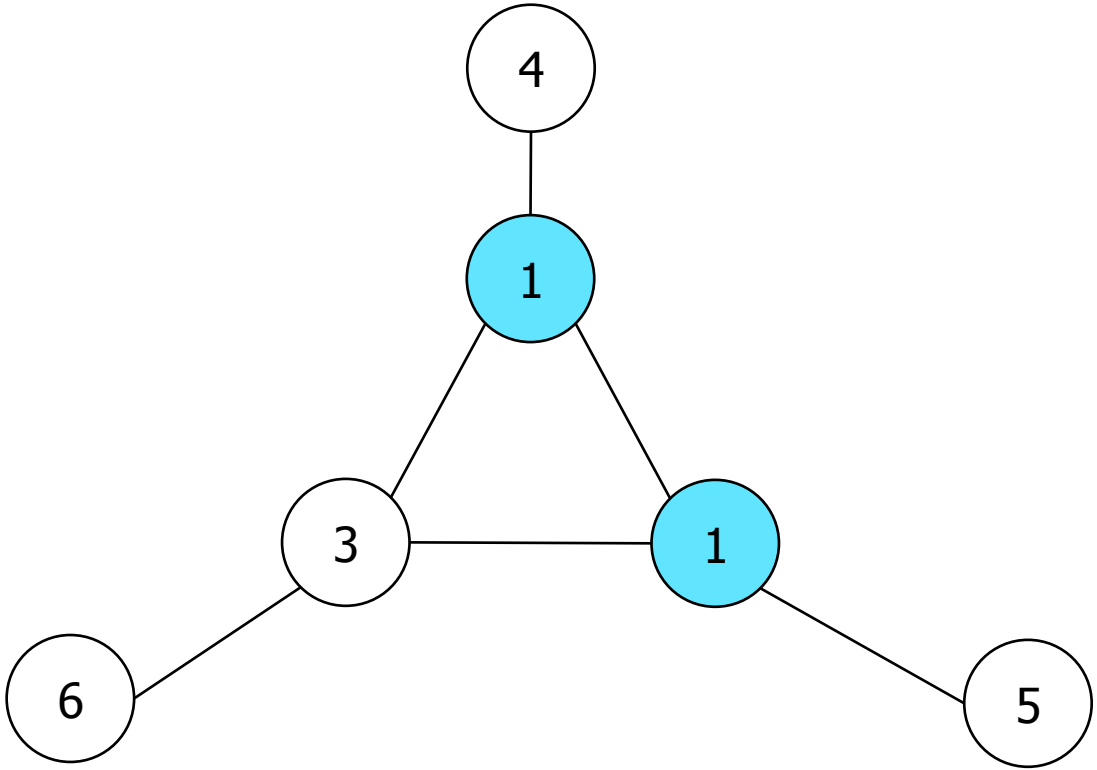
Label Propagation: Ordering from High to Low-degree Nodes



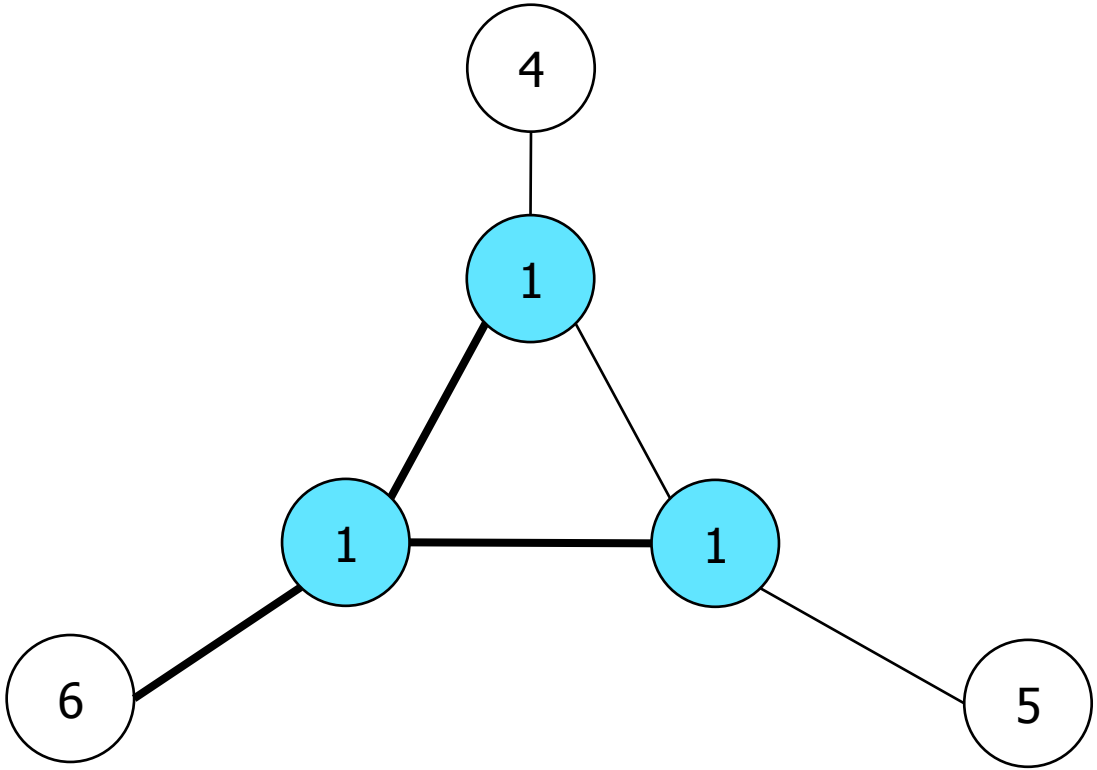
Label Propagation: Ordering from High to Low-degree Nodes



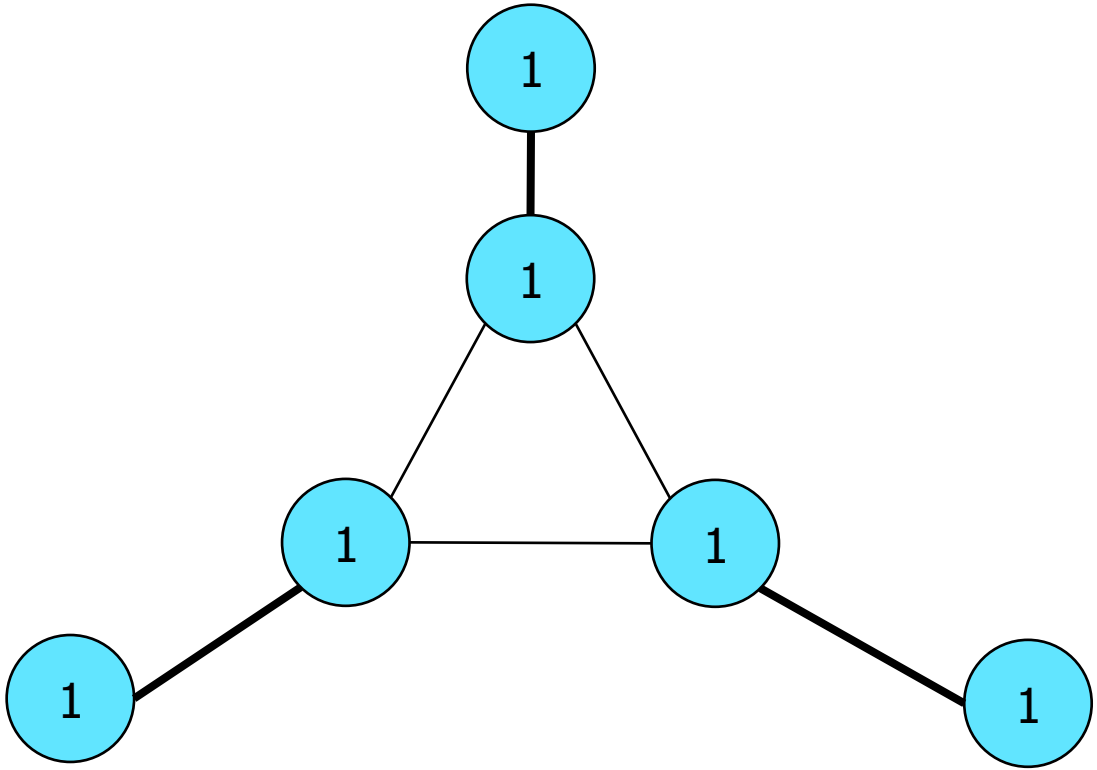
Label Propagation: Ordering from High to Low-degree Nodes



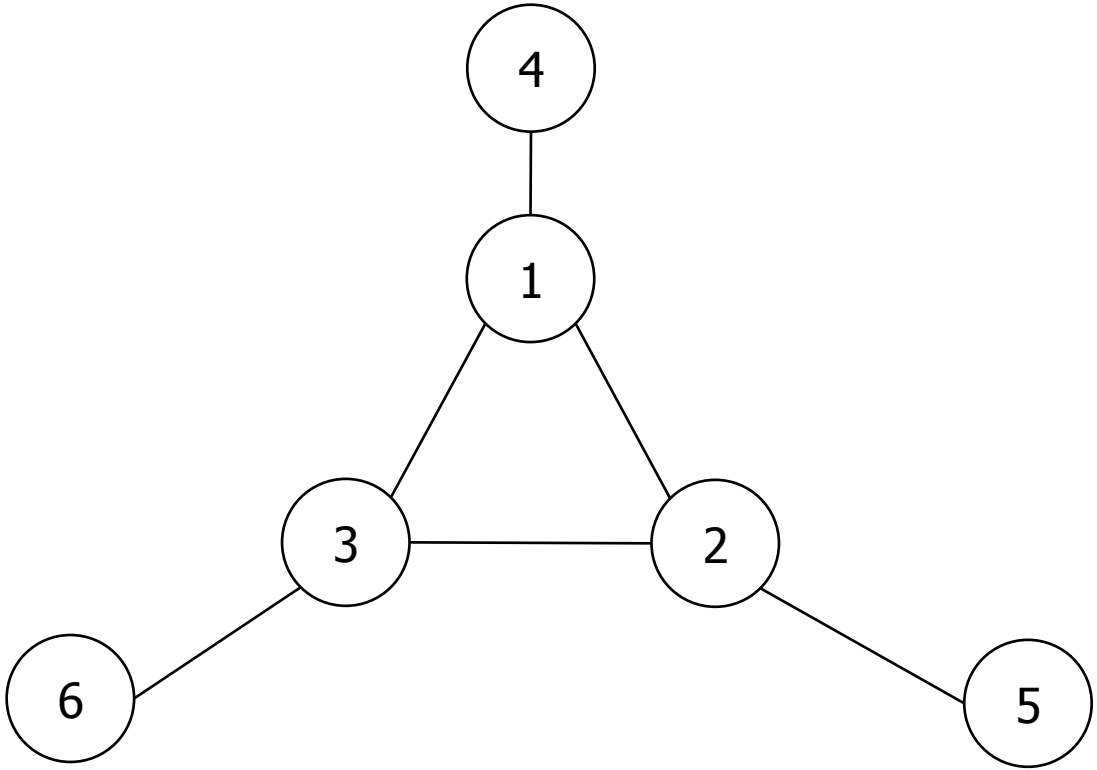
Label Propagation: Ordering from High to Low-degree Nodes



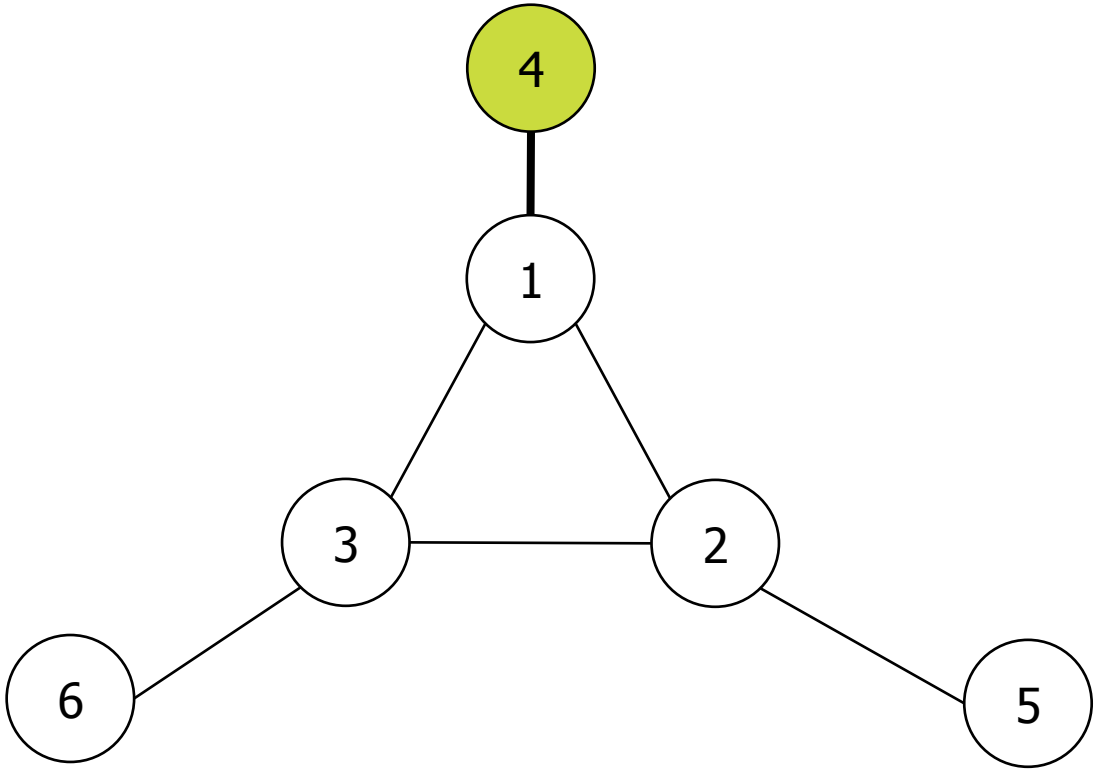
Label Propagation: Ordering from High to Low-degree Nodes



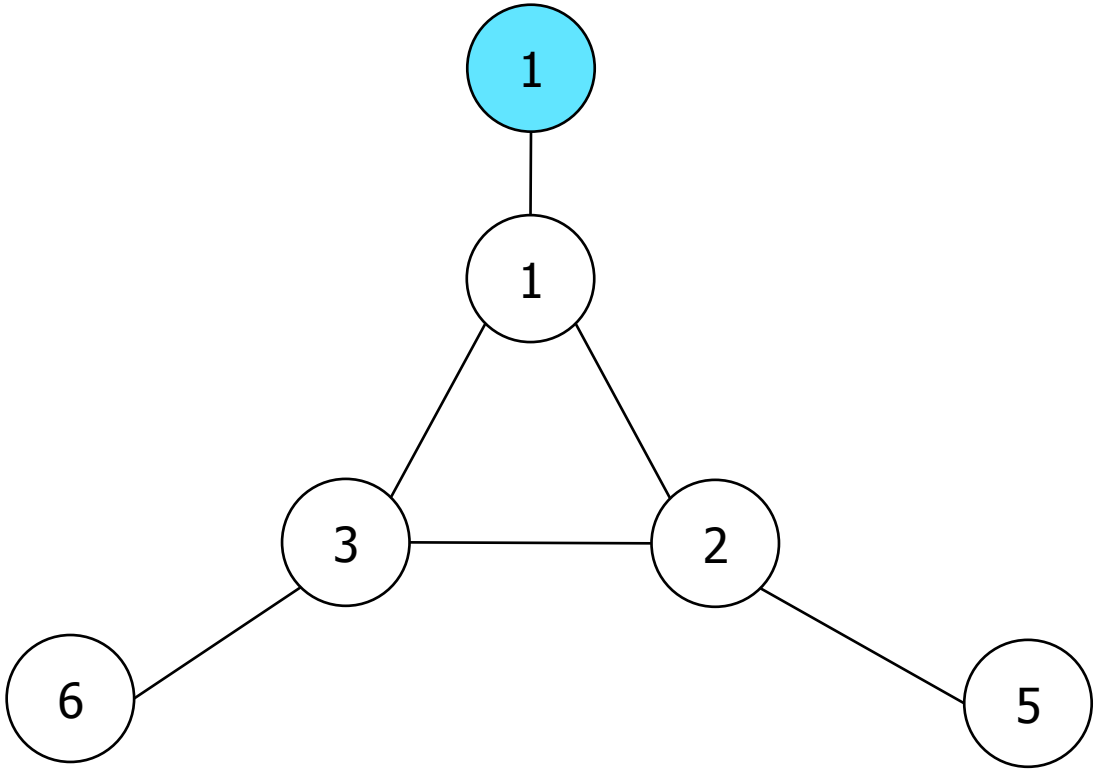
Label Propagation: Ordering from Low to High-degree Nodes



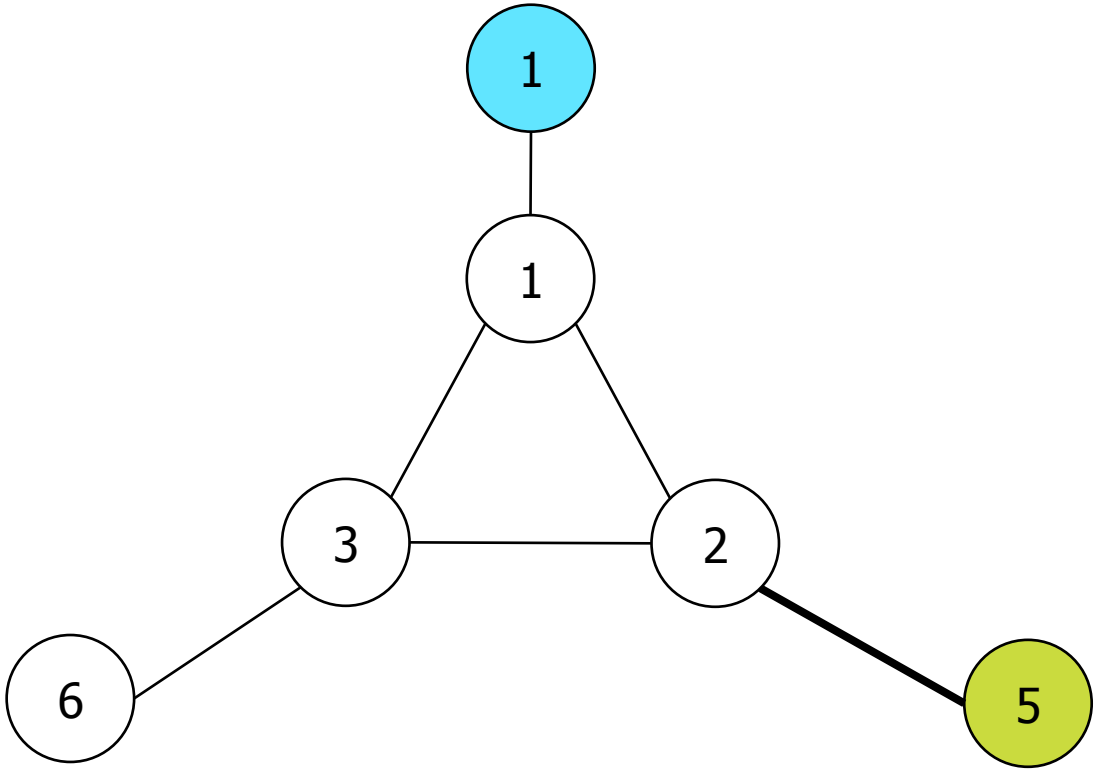
Label Propagation: Ordering from Low to High-degree Nodes



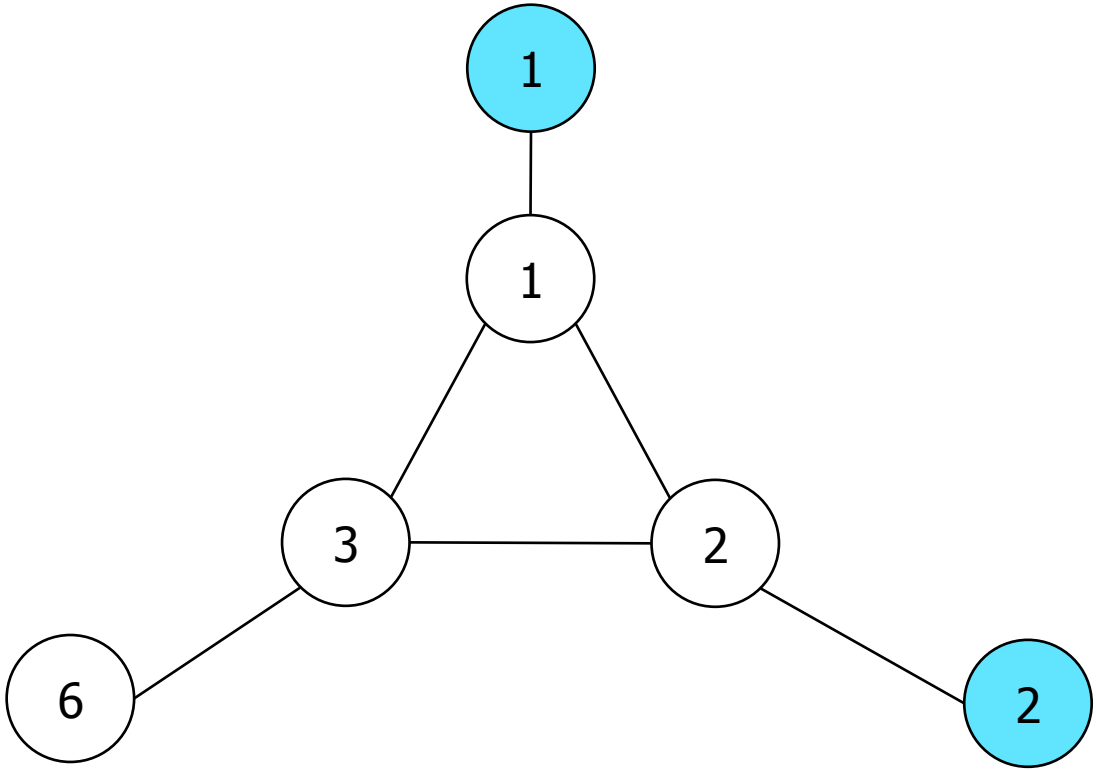
Label Propagation: Ordering from Low to High-degree Nodes



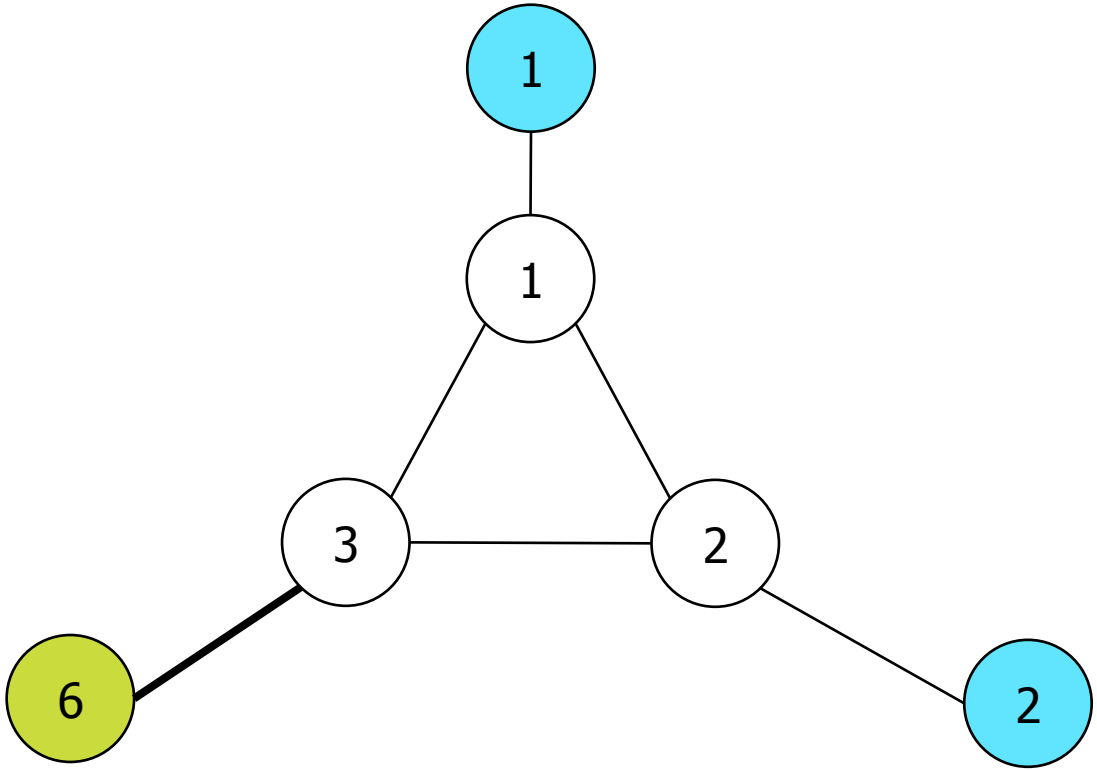
Label Propagation: Ordering from Low to High-degree Nodes



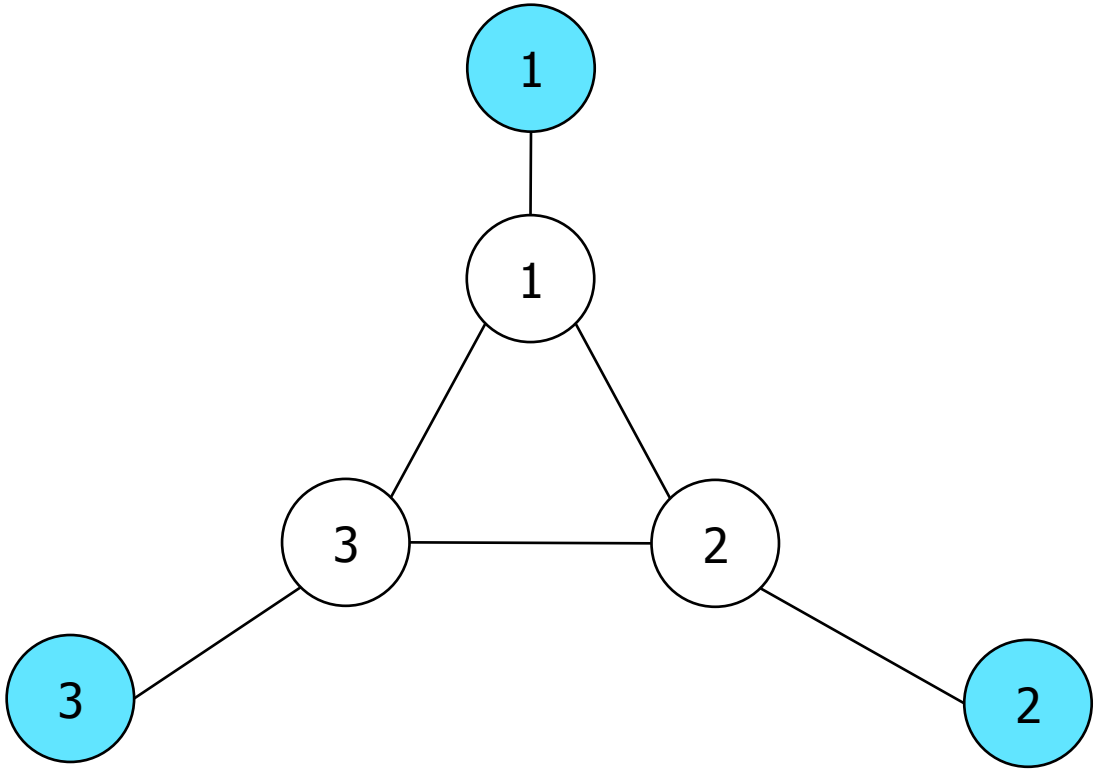
Label Propagation: Ordering from Low to High-degree Nodes



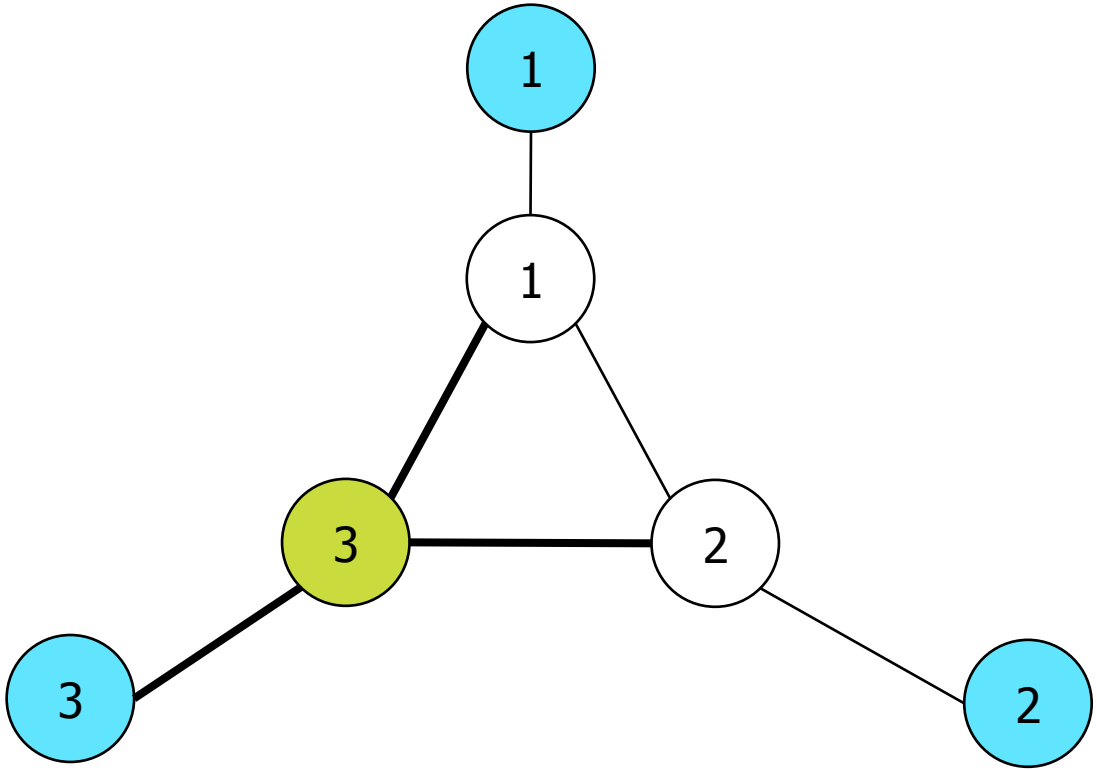
Label Propagation: Ordering from High to Low-degree Nodes



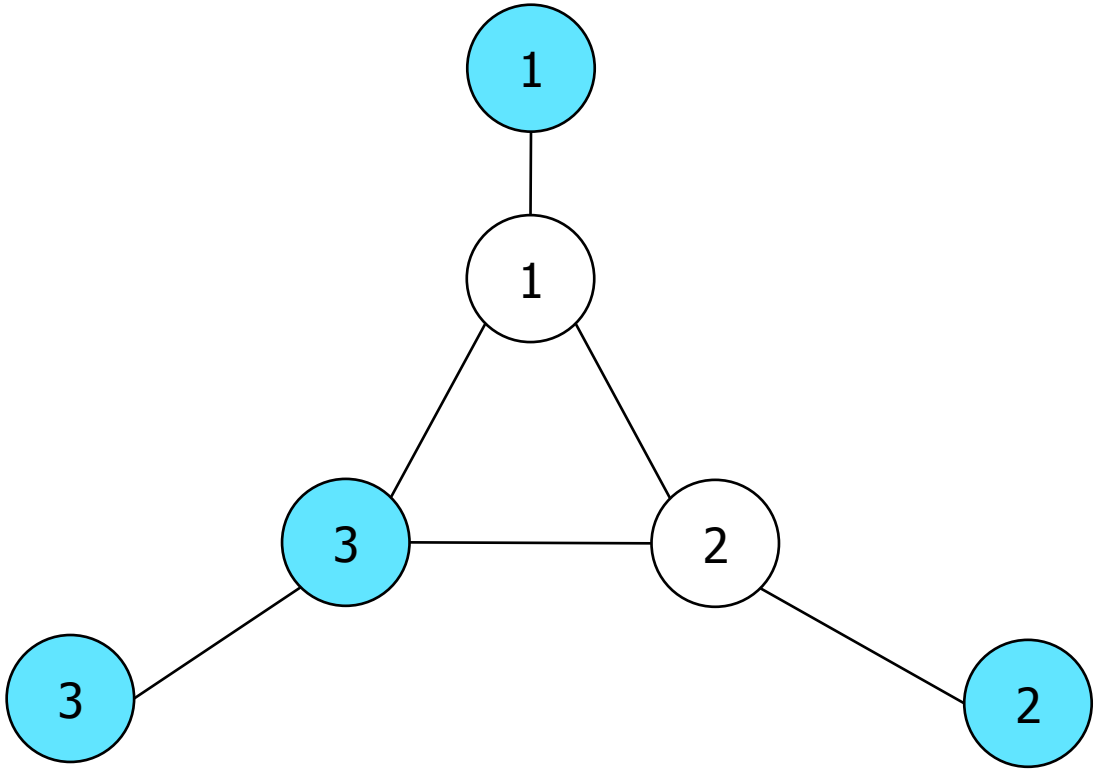
Label Propagation: Ordering from Low to High-degree Nodes



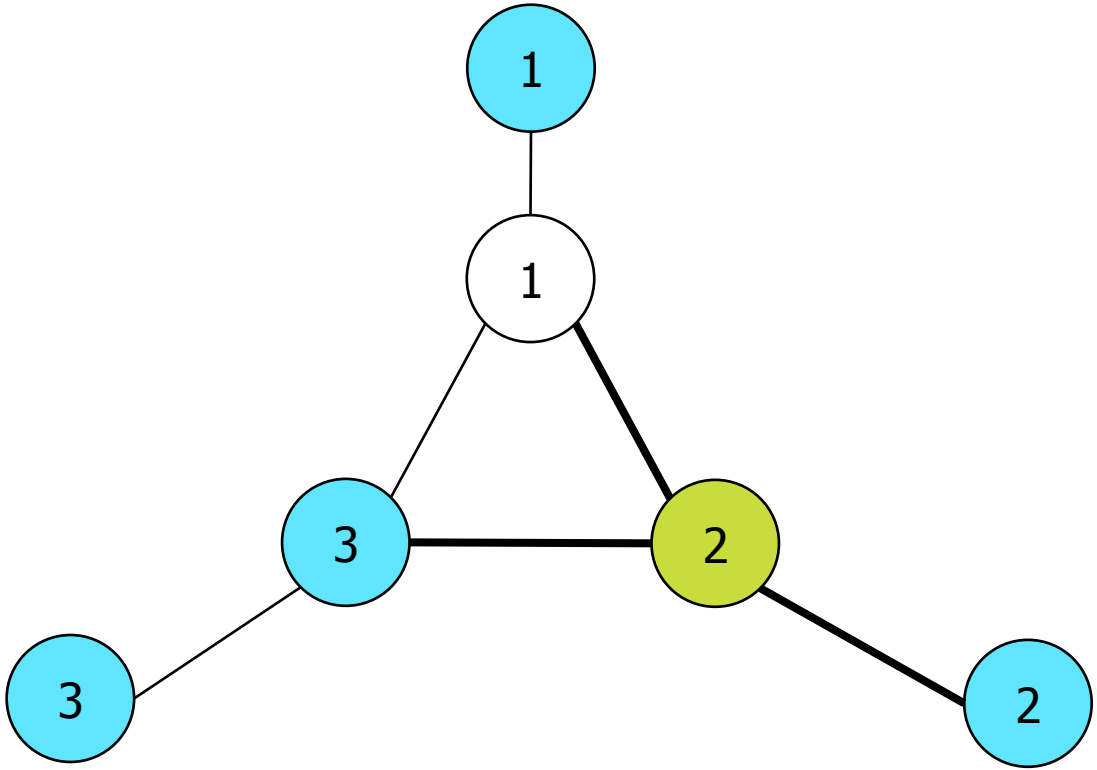
Label Propagation: Ordering from Low to High-degree Nodes



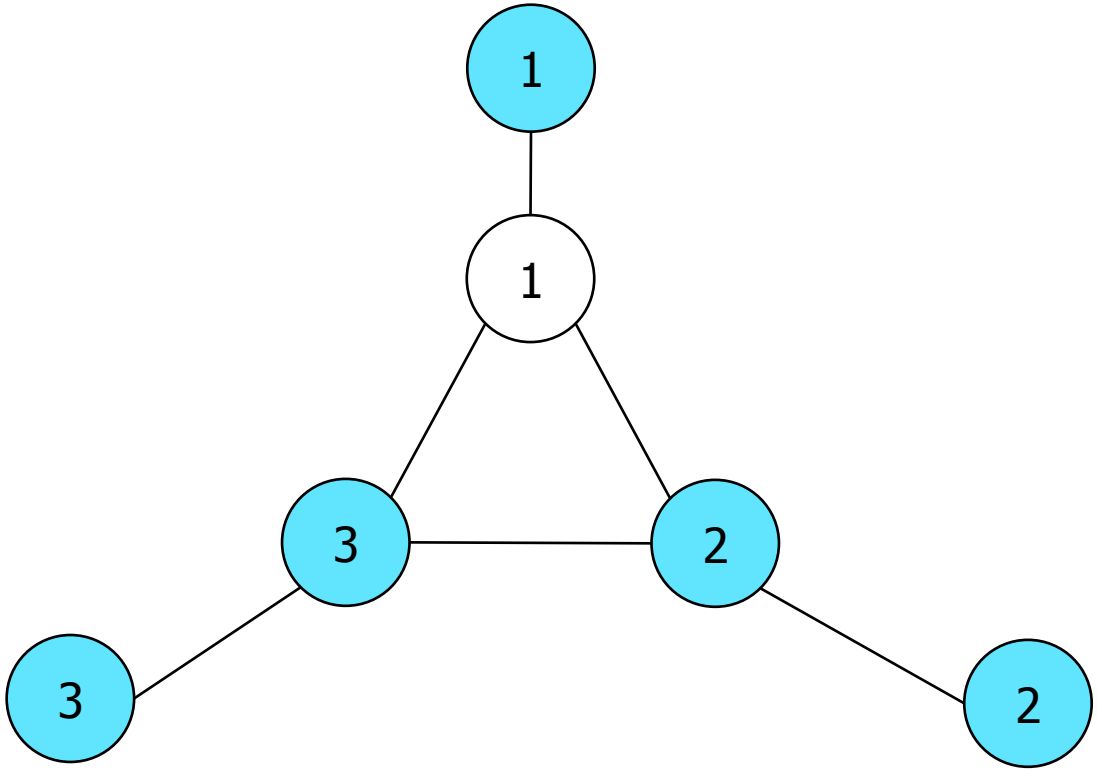
Label Propagation: Ordering from Low to High-degree Nodes



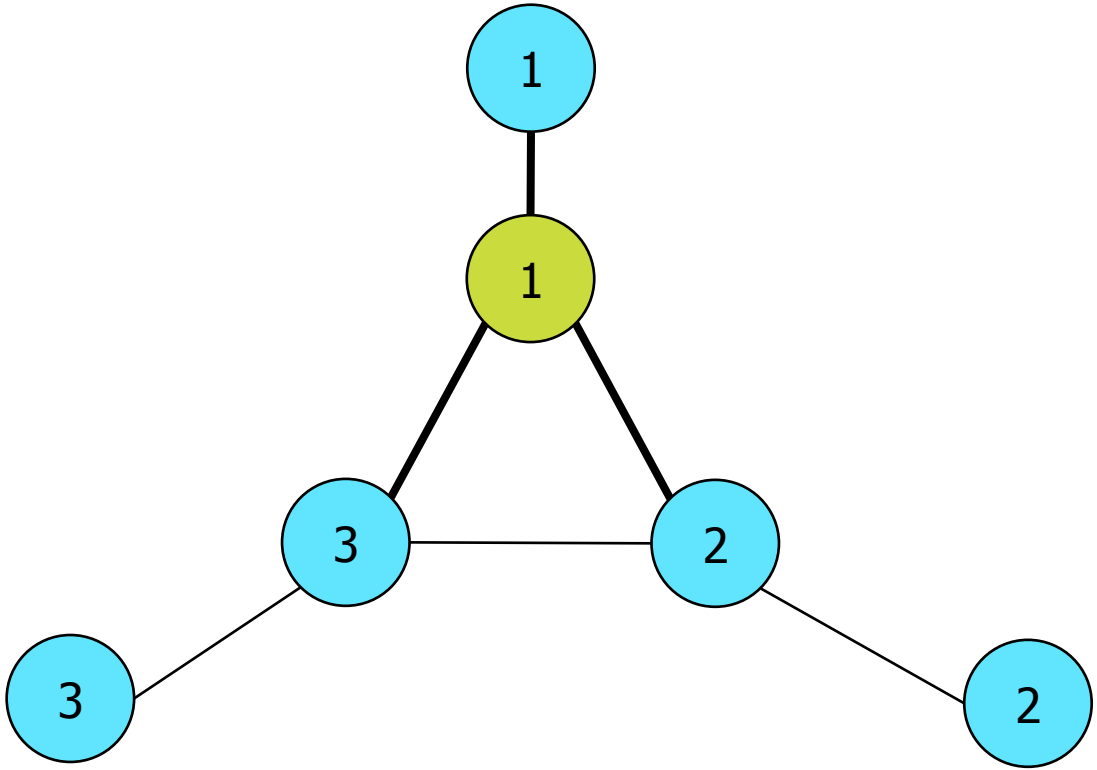
Label Propagation: Ordering from Low to High-degree Nodes



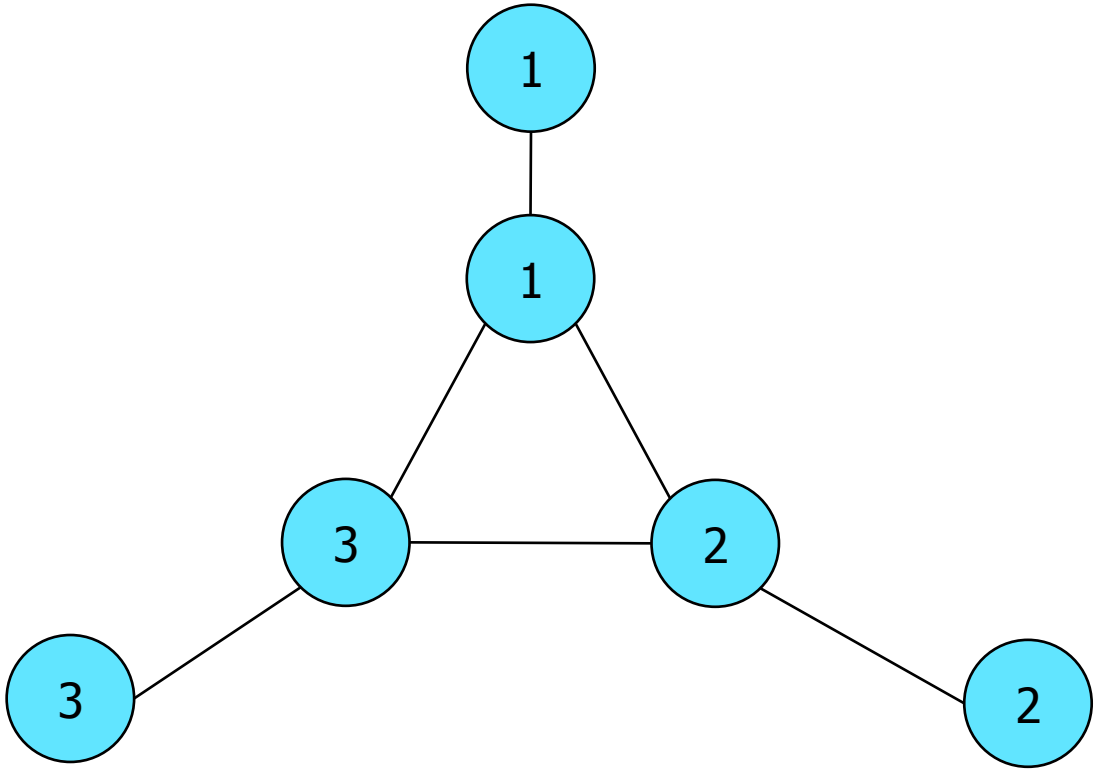
Label Propagation: Ordering from Low to High-degree Nodes



Label Propagation: Ordering from Low to High-degree Nodes



Label Propagation: Ordering from Low to High-degree Nodes



Parallel Community Detection: Asynchronous Execution

- Approach: All vertices processed in parallel
  - Use the most recent label to allow for fast mixing

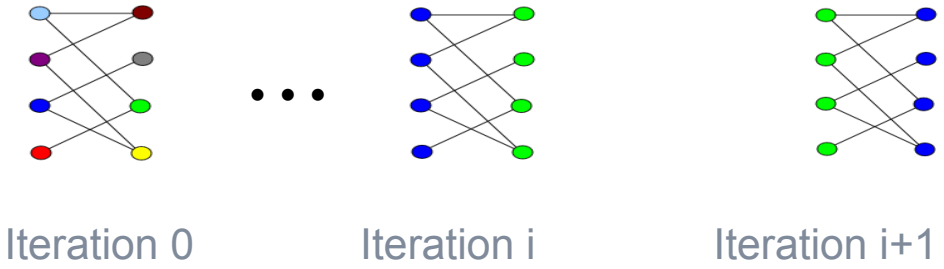
- Parallel running time:
  - Faster convergence owing to faster label mixing

- Robustness:
  - The control over vertex ordering is lost leading to inconsistent results
  - Identified communities may be affected by architecture and system level issues
    - Racing
    - Cache coherency issues such as when the private caches flush their content

Parallel Community Detection: Synchronous Execution

- Approach: Use labels from previous iteration to update labels in the current iteration

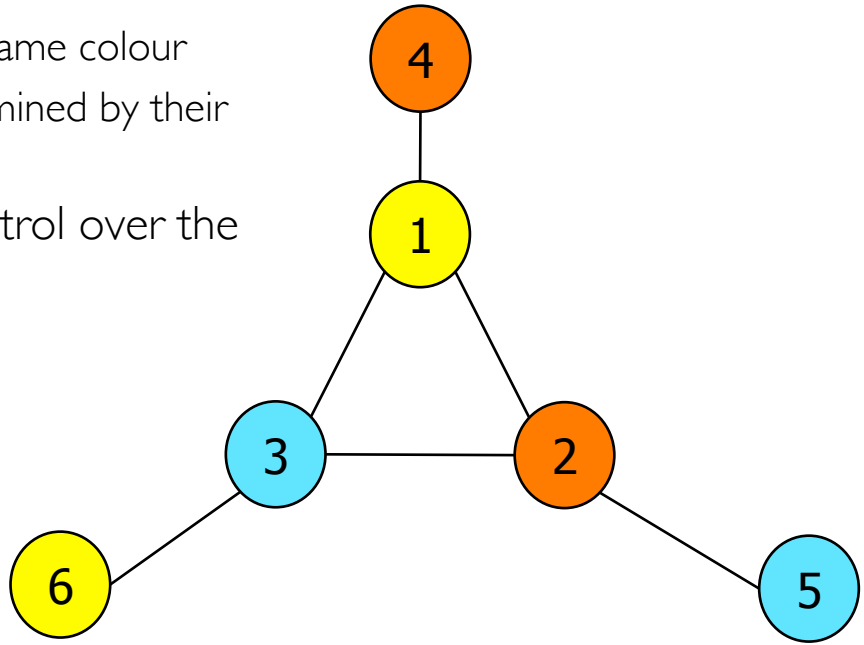
- Parallel Running time:
  - Slow label mixing
  - Can lead to label oscillation problem



- Robustness:
  - Vertex ordering is not important anymore
  - Racing is eliminated
  - Results are independent of the underlying system and architecture

Parallel Community Detection: Semi-Synchronous Execution

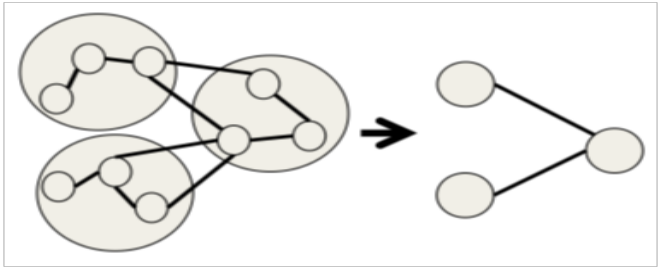
- Approach consists of two steps:
  - Graph colouring: no two adjacent vertices share the same colour
  - Community detection step: order of vertices is determined by their colour
- Coloring strategy provides a certain degree of control over the vertex ordering and parallelism
- Parallel running time:
  - Faster label mixing than synchronous execution
  - Graph coloring only takes a small fraction of time
- Robustness:
  - No racing
  - Results independent of underlying system



Making Semi-synchronous Execution Faster

- Careful selection of graph colouring algorithms
  - Minimise the number of iterations
  - Minimise the load imbalance per iteration

- Work at a cluster level, rather than at individual vertex level



- Other speed-up tricks
  - Only consider that part of a graph where community structure is still changing
  - Stop the iterative framework if too few vertices were updated

Semi-synchronous Community Detection: Experimental Evaluation

- Large-scale real-world graphs

Network	# nodes	# edges	Type
UK 2002	18,520,486	298,113,762	web graph
Orkut	3,072,441	117,185,083	social network
Live Journal	4,847,571	42,851,237	social network
Pokec	1,632,803	22,301,964	social network

- Range of shared-memory architectures

Name	# cores	Processor type	RAM
Config A	2 x 10	Intel Xeon E5-2470 v2 2.4GHz	128GB
Config B	2 x 16	2.6GHz AMD Opteron 6282 SE	96GB
Config C	4	3.6GHz Intel(R) Core i7-3820	64GB
Config D	6	2.4GHz Intel(R) Xeon E7450	128GB

Robustness of Parallel Semi-synchronous Approach

- Running time:
  - A carefully engineered semi-synchronous execution can be almost as fast as the asynchronous execution

- Parallelism:
  - Semi-synchronous: Compared to sequential semi-synchronous execution, running time improves by a factor of ~4 with 8 cores, across all networks

- Robustness in quality:
  - Semi-synchronous: Relatively high modularity of communities for all social networks [0.48,0.55]
  - Asynchronous: Highly variable modularity of communities [0.24,0.51]

Summary: Parallelism vs. Robustness

- Parallelizing iterative batched algorithms can result in loss of control over the ordering in which vertices are processed

- To ensure that quality and running time of results are independent of architecture and systems issues, a slower algorithm may have to be implemented

- For label propagation to find communities,
  - Asynchronous parallelization: Fast, but results vary depending on architecture and system issues
  - Synchronous parallelization: Robust, but poor label mixing resulting in slow convergence
  - Semi-synchronous parallelization: Robust, almost as good a parallel running time as asynchronous approach
    - Colouring strategy provides a bit of control over node ordering as well as parallelism

Outline

PART I

Scalability vs. Running Time - Architecture and System
Expressivity vs. Development Time - System and API

PART II

Parallelism vs. Workload – Algorithm and Implementation
Parallelism vs. Robustness – Algorithm and Implementation

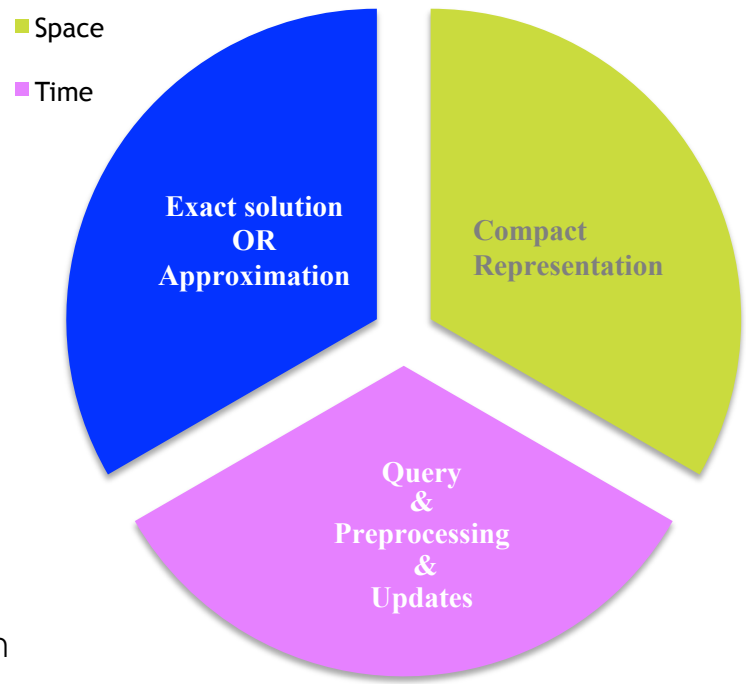
PART III

Space vs. Query Time vs. Approximation Quality – Storage and Data Structures

Hands-on Session: Neo4J and Galois

Data structures: graph representation

- For graph data structures, there is often a trade-off between space, access time and approximation
  - Variety of access queries: adjacency, connectivity, distances, paths, flows etc.
  - Access time can be dominated by I/Os or inter-processor communication
  - Better understanding of the trade-offs because of decades of research
- Two examples to illustrate these trade-offs:
  - Partitioned graph representations for space vs. distributed access
  - Distance oracles for distance and path queries in real-world graphs with trade-off between space, time and approximation



Bell Labs
Alcatel·Lucent

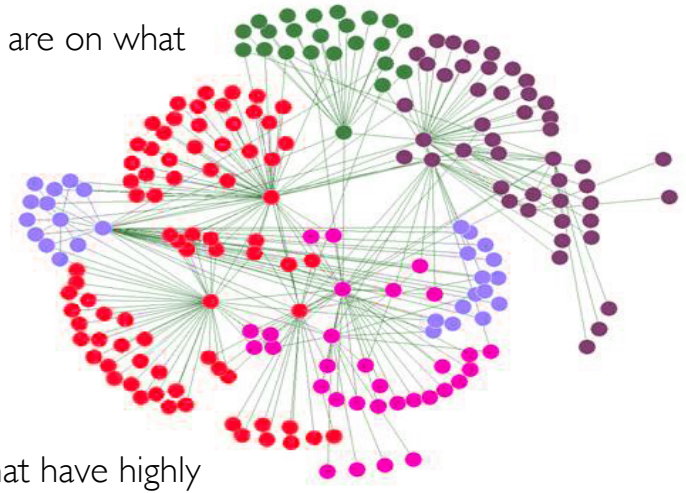
Example 1: Distributed Graph Representations

- Each processor stores the entire graph ("full replication")
  - Space:
    - Waste of space, limits scalability
  - Access time:
    - Enables concurrent pattern-matching queries, but not good for batched/global computation

- Each processor stores n/p vertices and all adjacencies out of these vertices
  - Space:
    - No replication
  - Access time:
    - Simpler access to adjacencies, simpler pre-processing
    - Different vertices have different degrees, so it can cause extreme load imbalance, resulting in slow batched/global computations

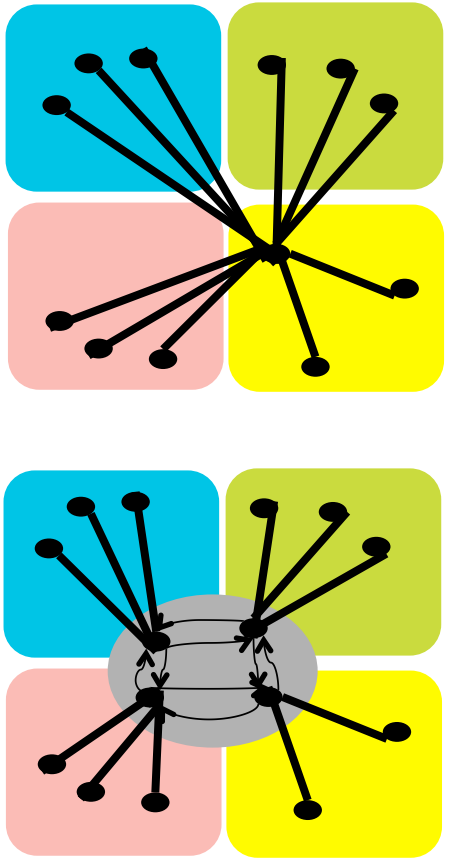
Example 1: Distributed Graph Representations

- More complex partitioning techniques (e.g., multilevel partitioning)
  - Space:
    - No replication
    - Access to edges may require additional indices to determine what edges are on what compute node
  - Access time:
    - Slightly more access time because of a level of indirection
    - Slightly more pre-processing time
    - Significantly better load balance
    - Inter-processor communication can be very high for real-world graphs that have highly skewed power-law degree distribution as no good graph partitioning exists for these graphs



Example 1: Distributed Graph Representations

- Copy the celebrity vertices (delegates) and keep their state consistent (e.g., PowerGraph)
  - Space:
    - Increased space due to replication
    - Considerable replication for fault-tolerance anyway!
    - Increased space can limit scalability
  - Access time:
    - Significantly reduced inter-processor communication for global computations



Example 2: Distance Oracles

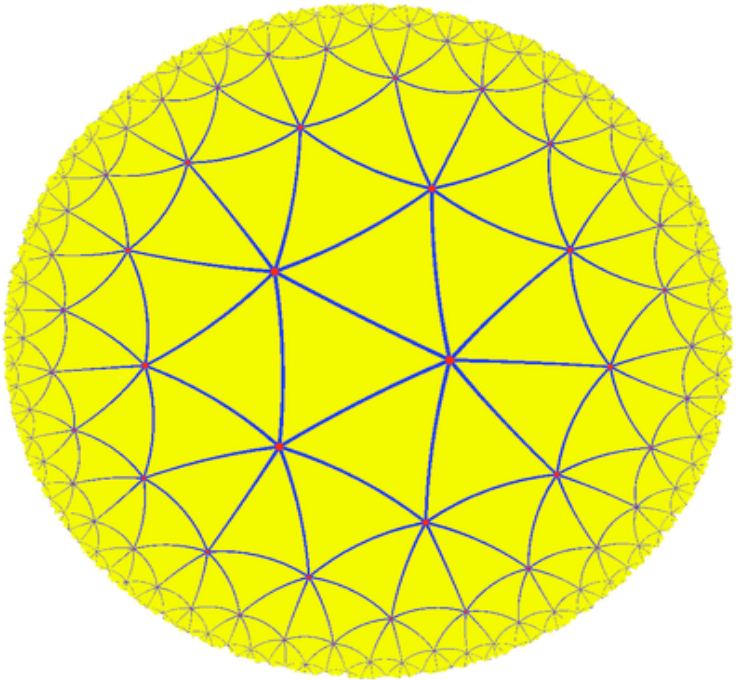
- A distance oracle is a data structure for efficient distance queries in graphs
  - Considerable literature on distance oracles
- Not possible to achieve exact distance oracle with O(n) space
- Even for approximation, theoretical lower bound for general graphs:
  - Not possible to get better than 2k-1 approximation with $O(k\, n^{1+1/k})$ space for any $k >= 2$
- Not very useful trade-off for real-world graphs with O(log n) diameter

Example 2: Leveraging specific graph properties

- Can we get better trade-offs for graphs from specific domains?

- Simple indexing schemes that achieves surprisingly good trade-off by leveraging specific properties of graphs
  - For road networks, various techniques based on goal-directed search, highway hierarchies, transit node routing, small natural cuts have proved successful
  - For real-world graphs, we utilize graph hyperbolicity and correlation between high degree nodes and high centrality nodes

Example 2: Graph Hyperbolicity



Basic Intuition: Shortest paths between most node pairs pass through a small set of “core” nodes

Bell Labs
Alcatel·Lucent

Example 2: Leveraging Graph Hyperbolicity

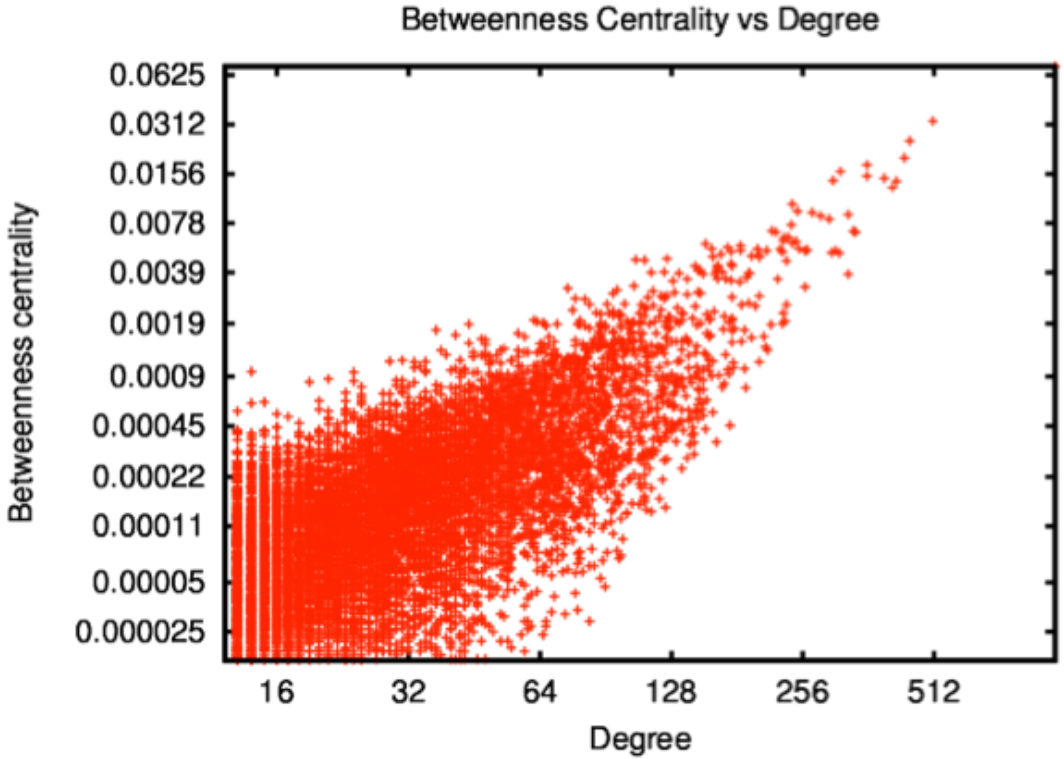
- Start from a high centrality vertex and grow a shortest path tree around it

- For vertices with same distance from the root, prioritize vertex expansion based on centrality
  - Expand the vertices in a BFS level in decreasing order of betweenness centrality

- To answer query for distance between two vertices, return the distance in the tree

- Centrality is computationally expensive

Example 2: Betweenness Centrality and Degree



Bell Labs
Alcatel·Lucent

Example 2: Leveraging Correlation between Centrality and Degree

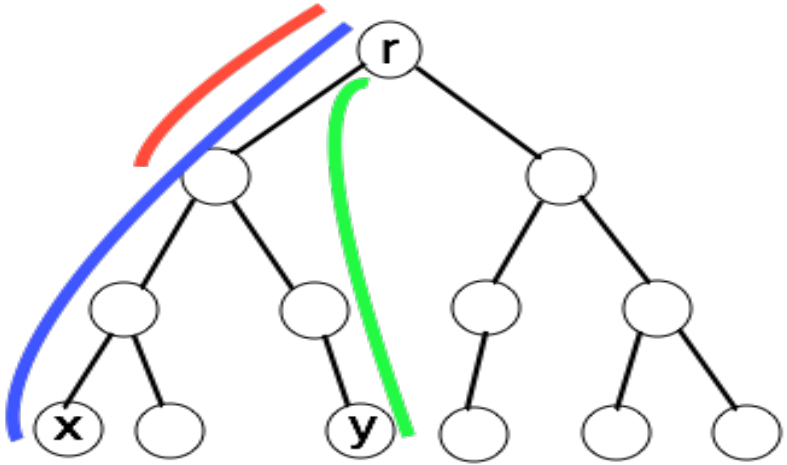
- Start from a high degree node and grow a shortest path tree around it

- For vertices with same distance from the root, prioritize vertex expansion based on degree
  - Expand the vertices in a BFS level in decreasing order of degree

- To improve the accuracy, take a number of trees and return the minimum distance over them
  - Taking n different trees results in an exact oracle, but a very small number suffices to give high accuracy

- Number of trees help to navigate the trade-off between space and accuracy

Example 2: A Missing Detail: Distance Queries in Trees



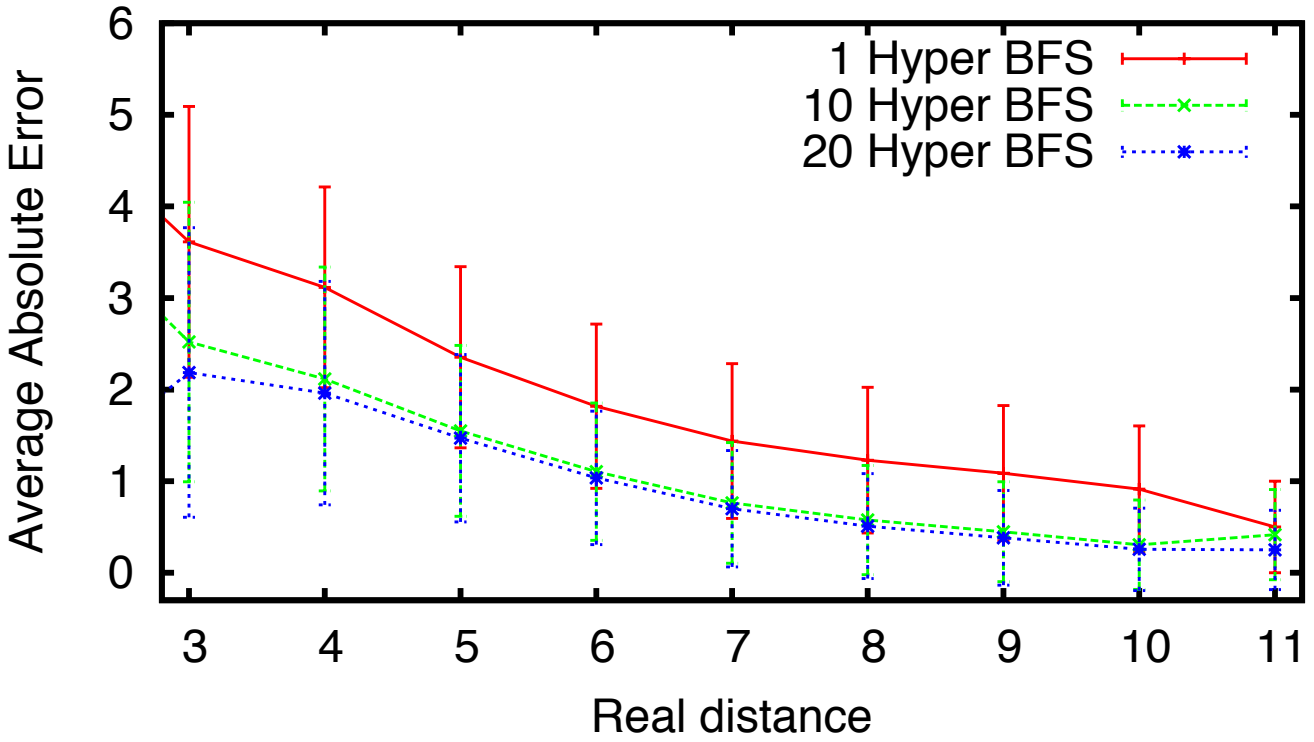
$$d(x,r) + d(y,r) - 2\,d(lca(x,y),r)$$

- With each vertex, we store its distance to root
- Additional structure for efficient LCA queries
  - O(log n + h) bits per vertex that supports O(1) query time

Example 2: Storage Space, Preprocessing Time, Query Time

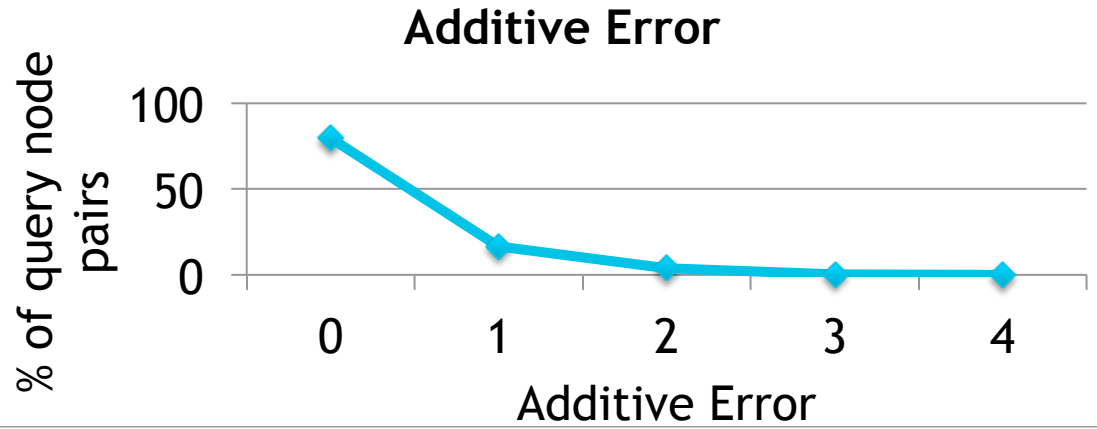
- Just a few bytes of storage space per vertex!
- Less than 15 minutes to preprocess for a call graph with 50 million vertices and 325 million edges using a single core
- Preprocessing can even be parallelised!
- Query time in microseconds

Example 2: Accuracy vs. Space on a Large Call Graph



Example 2: Distance Oracle in External Memory

- For a web-graph with 50 million vertices and 1.8 billion edges that doesn't fit in main memory,
  - Preprocessing time: < 3 hours
  - Index size: Less than the graph size
  - Query time: Microseconds with parallel SSDs
  - Update time: Milliseconds per edge



Bell Labs
Alcatel·Lucent

Summary: Graph Representation

- Graph representations often involve trade-off between space, access time and approximation

- For distance and path queries on real-world graphs, a simple *tunable* data structure
  - Highly accurate distance estimates
  - Query time of microseconds in internal memory
  - Few bytes of storage space per node
  - Sequential preprocessing time in minutes for graphs with hundreds of millions of edges
  - Can be externalized to deal with graphs containing billions of edges
    - Preprocessing time in hours, query time in microseconds and update time in milliseconds

Conclusion

- No single best way to deal with different graph applications

- Many different trade-offs to consider when selecting the right combination of architecture, system, API, algorithms, representation, storage for a graph application
  - To select the right architecture/system, consider the scalability vs. runtime trade-off
  - To select the right system/API, consider the expressivity vs. development time trade-off
  - To select the right graph algorithms, consider the parallelism vs. workload and parallelism vs. robustness trade-off
  - To select the right graph representation, consider the approximation vs. space vs. query time trade-off

- Parameterized algorithms and data structures exist for many problems that enable a fine-grained tuning of the various trade-offs

References - 1

- Tutorial: "Systems for Big Graphs," by Khan and Elniteky, VLDB 2014
- "A Survey of Parallel Graph Processing Frameworks," by Doekemeijer and Varbanescu, PDS report
- Tutorial: "Managing and mining Large Graphs: Systems and Implementations," by Wang, Shao and Xiao, SIGMOD 2012
- Tutorial: "Database Techniques for Linked Data Management," by Harth, Hose and Schenkel, SIGMOD 2012
- Tutorial: "Graph Data Management Systems for New Application Domains," by Cudré-Mauroux and Elnikety, VLDB 2011
- Tutorial: "Cloud-based RDF data management," by Kaoudi and Manolescu, SIGMOD 2014
- Tutorial: "Modern Database Systems," by Mohan, VLDB 2013
- "Graph Analysis with High- Performance Computing," by Hendrickson and Berry, in "Combinatorics in Computing," 2008
- "Graph Stream Algorithms: A Survey," by McGregor, in ACM SIGMOD Record, 2014
- "Dynamic Graph Algorithms," by Demetrescu, Eppstein, Galil and Italiano, chapter in Book "Algorithms and theory of computation handbook," Chapman & Hall/CRC, 2010
- "Algorithms for Memory Hierarchies,", Meyer, Sanders and Sibeyn (eds.), Springer, 2003
- "Parallel Graph Algorithms," Bader (ed.), Chapman and Hall/CRC, 2015
- Tutorial: "Mining Billion-Scale Graphs: Patterns and Algorithms," by Faloutsos and Kang, SIGMOD 2012

References - 2

- E. Duriakova, N. Hurley, D.Ajwani, A. Sala: Analysis of the semi-synchronous approach to large-scale parallel community finding. COSN 2014: 51-62
- U. Meyer, P. Sanders: [Delta]-stepping: a parallelizable shortest path algorithm. J. Algorithms 49(1): 114-152 (2003)
- D. Ajwani, W. S. Kennedy, A. Sala, I. Saniee: A Geometric Distance Oracle for Large Real-World Graphs. CoRR abs/1404.5002 (2014)
- D. Ajwani, U. Meyer, D.Veith: An I/O-efficient Distance Oracle for Evolving Real-World Graphs. ALENEX 2015: 159-172
- D. Ajwani, U. Meyer: Design and Engineering of External Memory Traversal Algorithms for General Graphs. Algorithmics of Large and Complex Networks 2009: 1-33
- D. Ajwani, I. Malinger, U. Meyer, S. Toledo: Characterizing the Performance of Flash Memory Storage Devices and Its Impact on Algorithm Design. WEA 2008: 208-219
- D. Ajwani, A. Beckmann, R. Jacob, U. Meyer, G. Moruz: On Computational Models for Flash Memory Devices. SEA 2009: 16-27
- K. Nilakant, V. Dalibard, A. Roy, E. Yoneki: PrefEdge: SSD Prefetcher for Large-Scale Graph Traversal. SYSTOR 2014:4:1-4:12
- D. Ajwani, U. Meyer, V. Osipov: Improved External Memory BFS Implementation. ALENEX 2007
- D. Ajwani, R. Dementiev, U. Meyer: A computational study of external-memory BFS algorithms. SODA 2006: 601-610
- K. Mehlhorn, U. Meyer: External-Memory Breadth-First Search with Sublinear I/O. ESA 2002: 723-735

References - 3

- K. Munagala, A. G. Ranade: I/O-Complexity of Graph Algorithms. SODA 1999: 687-694
- E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In OSDI, 2012.
- W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC. In KDD, 2013.
- S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In ASPLOS, 2012.
- S. Hong, S. Salihoglu, J. Widom, and K. Olukotun. Simplifying Scalable Graph Processing with a Domain-Specific Language. In CGO, 2014.
- A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale Graph Computation on Just a PC. In OSDI, 2012.
- A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry. Challenges in Parallel Graph Processing. Parallel Processing Letters, 17(1):5–20, 2007.
- G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-scale Graph Processing. In SIGMOD, 2010.
- K. Munagala and A. Ranade. I/O-complexity of Graph Algorithms. In SODA, 1999.
-] S. Salihoglu and J. Widom. Optimizing Graph Algorithms on Pregel-like Systems. In VLDB, 2014.
- A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In SOSP, 2013.

References - 4

- M. Sarwat, S. Elnikety, Y. He, and M. F. Mokbel. Horton+: A Distributed System for Processing Declarative Reachability Queries over Partitioned Graphs. 2013.
- B. Shao, H. Wang, and Y. Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In SIGMOD, 2013.
- J. Shun and G. E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In PPoPP, 2013.
- Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "Think Like a Vertex" to "Think Like a Graph". In VLDB, 2013.
- K. D. Underwood, M. Vance, J. W. Berry, and B. Hendrickson. Analyzing the Scalability of Graph Algorithms on Eldorado. In IPDPS, 2007.
- L. G. Valiant. A Bridging Model for Parallel Computation. Commun. ACM, 33(8), 1990.
- G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous Large-Scale Graph Processing Made Easy. In CIDR, 2013.
- R. S. Xin, D. Crankshaw, A. Dave, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: Unifying Data-Parallel and Graph-Parallel Analytics. CoRR, abs/1402.2394, 2014.
- S. Yang, X. Yan, B. Zong, and A. Khan. Towards Effective Partition Management for Large Graphs. In SIGMOD, 2012.
- A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In SC, 2005.
-] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In HotCloud, 2010.

References - 5

- M. Patrascu, L. Roditty: Distance Oracles beyond the Thorup-Zwick Bound. SIAM J. Comput. 43(1): 300-311 (2014)
- M. Thorup, U. Zwick: Approximate distance oracles. J. ACM 52(1): 1-24 (2005)
- Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J. M. Hellerstein: Distributed GraphLab: A Framework for Machine Learning in the Cloud. PVLDB 5(8): 716-727 (2012)
- J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin: PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. OSDI 2012: 17-30
- Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, J. D. Owens: Gunrock: a high-performance graph processing library on the GPU. PPOPP 2015: 265-266
- D. Nguyen, A. Lenharth, K. Pingali: Deterministic galois: on-demand, portable and parameterless. ASPLOS 2014: 499-512
- K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, X. Sui: The tao of parallelism in algorithms. PLDI 2011: 12-25
- R. Dementiev, L. Kettner, P. Sanders: STXXL: standard template library for XXL data sets. Softw., Pract. Exper. 38(6): 589-637 (2008)
- L. G. Valiant, A bridging model for parallel computation, Communications of the ACM, Volume 33 Issue 8, Aug. 1990
- Titan. http://thinkaurelius.github.com/titan/

References - 6

- D. Nicoara, S. Kamali, K. Daudjee, L. Chen: Hermes: Dynamic Partitioning for Distributed Social Network Graph Databases. EDBT 2015: 25-36
- D. A. Bader, K. Madduri: Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2. ICPP 2006: 523-530
- D. Ediger, D. A. Bader: Investigating Graph Algorithms in the BSP Model on the Cray XMT. IPDPS Workshops 2013: 1638-1645
- F. Holzschuher, R. Peinl: Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j. EDBT/ICDT Workshops 2013: 195-204
- Gremlin: http://www.slideshare.net/slidarko/gremlin-a-graphbased-programming-language-3876581
- J. H. Reif: Depth-First Search is Inherently Sequential. Inf. Process. Letters 20(5) 1985: 229-234
- D. Gregor , A. Lumsdaine: The parallel BGL: A generic library for distributed graph computations, POOSC 2005
- W. S. Kennedy, O. Narayan, I. Saniee: On the Hyperbolicity of Large-Scale Networks. CoRR abs/1307.0031 (2013)
- "Managing and Mining Graph Data," Aggarwal and Wang (eds.), Springer, 2010
- C. Sommer: "Shortest Path Queries in Static Networks", 2013

Outline

PART I

Scalability vs. Running Time - Architecture and System
Expressivity vs. Development Time - System and API

PART II

Parallelism vs. Workload – Algorithm and Implementation
Parallelism vs. Robustness – Algorithm and Implementation

PART III

Space vs. Query Time vs. Approximation Quality – Storage and Data Structures

Hands-on Session: Neo4J and Galois

Hands on!
Neo4j and Galois in Action

Neo4j: Setting Up

- Extract the archive neo4j-community-…-unix/windows to your local disk:
  - Contains Neo4j stable version 2.2, Community Edition
- Copy DBpedia data directory `graph.db` into Neo4j's `data/` directory
- Starting/stopping the instance: `bin/neo4j start/stop`
  - `In Windows, double-click bin\Neo4j to start and close the shell window to stop`
- Access: via Neo4j shell `bin/neo4j-shell` (double-click bin\Neo4JShell in Windows) or browser interface `localhost:7474/` (default config)
- Configuring the instance: `conf/` directory
  - Might need to set port and IP address in `neo4j-server.properties`
  - Other important files: `neo4j.properties` (instance) & `neo4-wrapper.conf` **(java config) - no change needed today**
- Logs are written to `data/log/console.log` (general) & `data/graph.db/messages.log` (DB specific)

DBpedia Graph Data

- Contains only the DBpedia 2014 data from 'Mapping-based Properties (Cleaned)'
- Only node properties: uri for resource nodes, value for literals
- More node properties require careful analysis of the RDF data and own design choices
  - Multiple RDF predicates of same subject (e.g., international `names` of movies) cannot be modelled as node properties
- Data has schema indexes on: `:Resource(uri)`, `:Literal(value)`
  - For exact match only - for efficient fuzzy match one has to create legacy (i.e. Lucene full-text) indexes (as of now)

Neo4j: Loading Data on Your Own

- Several alternatives:
  1. Use an import tool from here https://github.com/jexp/neo4j-shell-tools
  2. Use a tailored tool, e.g. https://github.com/knutwalker/dbpedia-neo4j
  3. Fetch an existing Neo4j dump, e.g. from https://github.com/kbastani/neo4j-dbpedia-importer
- Choice should depend on the graph you eventually want (structure, properties, etc.), aspired performance & flexibility
- Simple own solution: Neo4j BatchInserter

  No indexes built during batch insert, thus:
  1. Scan all triples, create Neo4j nodes and collect mapping URI->node ID **manually**
  2. Use mapping to create Neo4j relationships between **existing** node IDs
     - For each triple, decide what is modelled as relationship, what as property

Neo4j: Loading Data on Your Own - Fairly Simple

```
Map<String, Long> neo4jids = new HashMap<String, Long>();
Label resourceLabel = DynamicLabel.label("Resource");
Label literalLabel = DynamicLabel.label("Literal");
BatchInserter inserter = BatchInserters.inserter(dbName, config );
inserter.createDeferredSchemaIndex(resourceLabel).on("uri").create();
...
Map<String, Object> subjProperties = new HashMap<String, Object>();
nodeProperties.put("uri", tripleSubj); // props can still be extended later in the batch run
if (!neo4jids.contains(tripleSubj)) {
 neo4jids.put(tripleSubj, inserter.createNode(subjProperties, resourceLabel));
}
...
relProperties.put("uri", triplePred);
inserter.createRelationship(neo4jids.get(tripleSubj), neo4jids.get(tripleObj),
                            DynamicRelationshipType.withName(triplePred), relProperties);
...
nodeInserter.shutdown();
```

Neo4j Cypher: Relatedness & Pattern Matching

- Relatedness Larry Page and Sergey Brin: all shortest paths

```
match
      (x:Resource {uri:'Sergey_Brin'}),
      (y:Resource {uri:'Larry_Page'})
      return allShortestPaths(x-[*]-y);
```

- All people who have been awarded a Turing Award from Italy

```
match
    p=(x:Resource
{uri:'Turing_Award'})-[:award]-(y:Resource)-[:birthPlace]-(z:Resource {uri:'Italy'})
        return p
```

- 2-hop neighborhood around Silvio Micali

```
match
p=(x:Resource {uri:'Silvio_Micali'})-[*0..2]-()
return p limit 100
```

Neo4j Cypher: Local to Global Pattern Matching

- All persons born & died in Florence

```
match
    (x:Resource)-[b:birthPlace]->(f:Resource
{uri:'Florence'})<-[d:deathPlace]-(x) return b,d
```

- Assume we want to find out how Florence's relationship to Italy is modeled:

```
match (f:Resource {uri:'Florence'})-[r]-(x:Resource)
   where x.uri =~ '.*Italy.*' return r;
```

- Now let's see all persons born and died in an Italian city:

```
match
  (c:Resource)-[:country]->(italy:Resource {uri:'Italy'}),
  (x:Resource)-[b:birthPlace]->(c)<-[d:deathPlace]-(x) return b,d Limit 50;
```

- Example for 'global pattern', rather than 'local' pattern: all persons born & died in the same city:

```
match (x)-[b:birthPlace]->(f:Resource)<-[d:deathPlace]-(x) return b,d limit 50;
```


Bell Labs
Alcatel·Lucent

Neo4j Cypher: Not all queries are fast

- Relatedness WWW conference and Florence: all shortest paths – Takes very long!

```
match
(x:Resource {uri:'International_World_Wide_Web_Conference'}),
(y:Resource {uri:'Florence'})
return allShortestPaths(x-[*]-y);
```

- Relatedness WWW conference and Florence: all paths up to length 6 – expensive!

```
match
p=(x:Resource {uri:'International_World_Wide_Web_Conference'})-[*..6]-(y:Resource {uri:'Florence'})
return p;
```


Bell Labs
Alcatel·Lucent

Neo4j API: Pattern Matching

- Using Neo4j's standard JAVA API for pattern matching is fairly simple:

```
graphDb = new GraphDatabaseFactory().newEmbeddedDatabase( DB_PATH );
registerShutdownHook( graphDb );
try ( Transaction tx = graphDb.beginTx() )
{
        Node n = graphDb.findNodes(usersLabel, "name", "Tom");
        Iterable<Relationship> it = n.getRelationships(knowsType);
        it.next().getProperty("date")…;
        tx.success();
}
graphDb.shutdown();
```

- …but not necessarily efficient!
  - Indexes etc. can be used, but (cost-based) optimisation is left to developer

Neo4j API: Traversals

```
TraversalDescription TRAVERSAL = traversal().uniqueness(Uniqueness.NODE_PATH)
          .order(new MySelectorFactory( forMultiplePaths, costEvaluator ) ) // use own, or
//        .order(BranchOrderingPolicies.PREORDER_BREADTH_FIRST ) // use one of the default options
          .evaluator( Evaluators.pruneWhereEndNodeIs(endNode));

abstract class MySelectorFactory<P extends Comparable<P>, D> implements BranchOrderingPolicy {
          static class Visit<P extends Comparable<P>> implements Comparable<P> {...}
          static class PriorityQueue<P> {...}
          // that's the true magic bit! …skips a lot of important intelligence here
          public TraversalBranch next( TraversalContext metadata ) {return queue.pop().getEntity(); }
}

public class ProbabilityWeightEvaluator implements CostEvaluator<Double> {
  public Double getCost( Relationship relationship, Direction direction ) {
          if(relationship.hasProperty("probability"))
                    return 1.0/(Double)relationship.getProperty("probability");
          return 1.0;
}
```

Installing Galois Dependencies: cmake

```
cp  -r /path/to/USB/Galois-dependencies /your/choice/of/directory
cd /your/choice/of/directory/Galois-dependencies/cmake
gunzip cmake-3.2.2.tar.gz
tar -xvf cmake-3.2.2.tar
cd cmake-3.2.2
./bootstrap
make
make install
```

This step may require sudo!

Installing Galois Dependencies: boost

```
cd /your/choice/of/directory/Galois-dependencies/boost
unzip boost_1_58_0.zip
cd boost_1_58_0
./boostrap.sh
./b2
./b2 install
```

This step might require sudo!

Galois: Setup

- Compiling Galois is straightforward once dependencies installed

```
wget http://iss.ices.utexas.edu/projects/galois/downloads/
Galois-2.2.1.tar.gz > /dev/null
tar xzf Galois-2.1.8.tar.gz
cd Galois-2.1.8/build
mkdir default; cd default
cmake ../.. > /dev/null
make > /dev/null
```

- To run a SSSP benchmark on a Galois graph file use:

```
apps/sssp/sssp -algo=async -t=[# threads] [path to input]
```

Galois: Compiling and Running the Demo

- We have written a simple shell program to play with the dbpedia graph in Galois
  - The demo is found in the directory *demo-www*
    - To run the demo, first ensure that Galois-2.2.1 has been compiled (boost and cmake required)
      - If Galois-2.2.1 has been copied to /directory/structure/Galois-2.2.1 and compiled there, then ensure that demo-www is copied to /directory/structure/Galois-dbpedia-demo-www as well
    - Inside the Galois-dbpedia-demo-www directory execute make to compile the demo
    - Run the demo using make run (Note: Number of threads used by Galois can be modified in the Makefile)
- When the demo is run, it loads a subset of the dbpedia graph dumped from Neo4j
  - Each node corresponds to a **Resource** where its label is its truncated URI
    - For instance the Resource "http://dbpedia.org/resource/Turing_Award" is stored with label "Turing_Award".
    - The DBpedia graph is stored in the binary file Galois-dbpedia-demo-www/data/dbpedia.g
    - The mapping between node IDs and URI labels is stored in the file Galois-dbpedia-demo-www/data/URI_labels.txt
  - Edges are stored based on their type and there are about 500 different types
    - Initially, all edges have weight 1, though our shell code enables you to change these weights.
    - Mapping between edge type IDs and their values can be found in Galois-dbpedia-demo-www/data/edge_types.txt

Galois: Loading Data on Your Own

- We can extract the graph topology from Neo4j and load it into Galois.
  - Java snippet using embedded Neo4j to dump the topology:

```
GraphDatabaseService db = new GraphDatabaseFactory().newEmbeddedDatabase( DB_PATH );
try ( Transaction ignored = db.beginTx();
      Result result = db.execute( "match (n)-[r]-(m) return id(n), id(m), type(r)" ) ) {
  while ( result.hasNext() ) {
    Map<String,Object> row = result.next();
    System.out.println(row.get("id(n)")+" "+row.get("id(m)")+"    + row.get("id(r)"));
  }
}
db.shutdown();
```

- Galois API includes tools like Galois::Graph::FileGraphWriter which can be used to convert a plain text dump of the topology into a Galois binary format graph file.

Galois: Demo Description and Query Example

- When the graph is loaded into memory, a Galois single source shortest path (SSSP) algorithm (modified from *Galois-2.2.1/apps/sssp/SSSP.cpp*) is run to determine the SSSPs from the node "International_World_Wide_Web_Conference"
- After running the SSSP algorithm, can execute several commands:
  - To find a shortest path from the source to "Florence" we can execute the command:

    PATH Florence

    >(International_World_Wide_Web_Conference [dist: 0]) -[award 1]-> (Torsten_Suel [dist: 1]) -[residence 1]-> (Germany [dist: 2]) -[deathPlace 1]-> (Scipione_Piattoli [dist: 3]) -[birthPlace 1]-> (Florence [dist: 4])

    - The path has the form > `(NODE-URI [dist: DISTANCE-FROM-SOURCE]) -[EDGE-TYPE EDGE-WEIGHT]->` …
    - Note that your path may be different since the SSSP algorithm contains asynchronous parallelism
  - To change the weight of the edge type "residence" to 100000 we execute:

    REWEIGHT residence 1000000

    - This will cause the SSSP algorithm to be re-run.
    - When the query "`PATH Florence`" is executed again, the path will be different! Try it out.

Galois: More Query Examples

- We can also run the SSSP algorithm with a different source node:
  - To set the source node to "Italy" use the command:

    SSSP Italy
  - We can then find shortest paths starting from Italy:

    PATH Turing_Award

    ```
    >(Italy [dist: 0]) -[birthPlace 1]-> (Silvio_Micali [dist: 1]) -[award 1]-> (Turing_Award [dist: 2])
    ```
  - Finally, to explore the types of edges associated with a node, we can use the command ADJ:

    ADJ Florence

    > Florence(221782) ---[country]--- Italy

    …

    Format for an edge (A,B) is URI_A(node ID) –[edge_type]– URI_B

Galois API: Code Snippet for Distance Initialization

- Galois abstracts parallelization in the form of iterators:

```
struct NodeData {
  unsigned int dist;
};
typedef Galois::Graph::LC_CSR_Graph<NodeData, unsigned int> Graph;
typedef Graph::GraphNode GGNode;
Graph gr; Galois::Graph::readGraph(gr, PATH_TO_GR_FILE);
struct Init {
    Graph& g;
    Init(Graph& g): g(g) {}
    void operator()(GGNode& n, Galois::UserContext<DataNode>& ctx) const {
      g.getData(n).dist = DIST_INFINITY;
    }
  };
Galois::for_each(gr.begin(), gr.end(), Init(gr)); // Apply Init's operator on all nodes
```

Structure containing fields for graph nodes

Loop API: can add requests

Galois requires specific function signatures be passed to the iterators

Galois API: More Code Snippets for Delta-stepping

- For the Galois delta-stepping based algorithms a built-in parallel priority queue is used:

```
using namespace Galois::WorkList;
typedef dChunkedFIFO<64> Chunk;
typedef OrderedByIntegerMetric<UpdateRequestIndexer<UpdateRequest>, Chunk, 10> OBIM;
```

- When edges are relaxed, their destination nodes are inserted into the queue

```
template<typename UpdateRequest>
struct UpdateRequestIndexer: public std::unary_function<UpdateRequest, unsigned int> {
    unsigned int operator() (const UpdateRequest& val) const {
    unsigned int t = val.w >> stepShift;
    return t;
}};
```

Requests are inserted with priority = shifted relaxed weight

- These **`UpdateRequests`** are user-defined structures:
  - The user specifies what is stored: in this case, a node id and weight
- See the SSSP app source code for the full details.