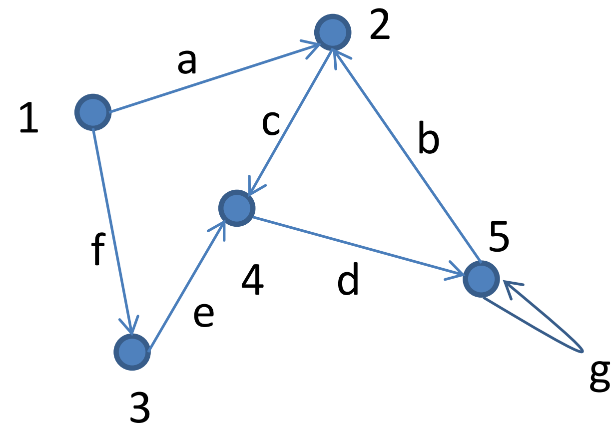
Graph Algorithms

Overview

- Graphs are very general data structures
  - G = (V,E) where V is set of nodes, E is set of edges $\subseteq$ VxV
  - data structures such as dense and sparse matrices, sets, multi-sets, etc. can be viewed as representations of graphs
- Algorithms on matrices/sets/etc. can usually be interpreted as graph algorithms
  - but it may or may not be useful to do this
  - sparse matrix algorithms can be usefully viewed as graph algorithms
- Some graph algorithms can be interpreted as matrix algorithms
  - but it may or may not be useful to do this
  - may be useful if graph structure is fixed as in graph analytics applications:
    - topology-driven algorithms can often be formulated in terms of a generalized sparse MVM

Graph-matrix duality

- Graph (V,E) as a matrix
  - Choose an ordering of vertices
  - Number them sequentially
  - Fill in |V|x|V| matrix
    - A(i,j) is w if graph has edge from node i to node j with label w
  - Called *adjacency matrix* of graph
  - Edge (u → v):
    - v is *out-neighbor* of u
    - u is in-neighbor of v
- Observations:
  - Diagonal entries: weights on self-loops
  - Symmetric matrix ←→ undirected graph
  - Lower triangular matrix ←→ no edges from lower numbered nodes to higher numbered nodes
  - Dense matrix ←→ clique (edge between every pair of nodes)

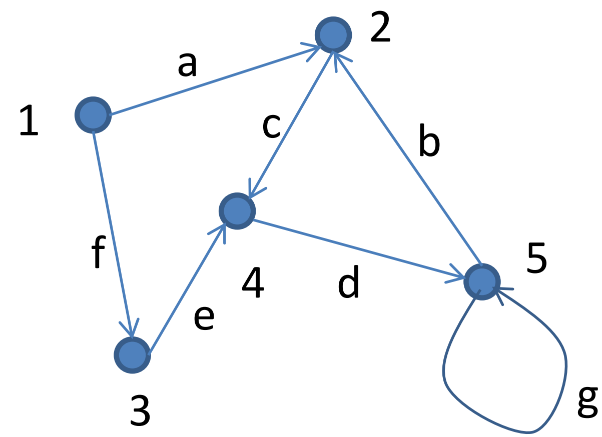


to / from

$$\begin{array}{c|c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} 0 & a & f & 0 & 0 \\ 0 & 0 & 0 & c & 0 \\ 0 & 0 & 0 & e & 0 \\ 0 & 0 & 0 & 0 & d \\ 0 & b & 0 & 0 & g \end{bmatrix} \end{array}$$

Matrix-vector multiplication

- Matrix computation: $\underline{y} = A\underline{x}$
- Graph interpretation:
  - Each node i has two values (labels) x(i) and y(i)
  - Each node i updates its label y using the x value from each out-neighbor j, scaled by the label on edge (i,j)
  - Topology-driven, unordered algorithm
- Observation:
  - Graph perspective shows dense MVM is special case of sparse MVM
  - What is the interpretation of $\underline{y} = A^T\underline{x}$?



$$
\begin{array}{c c}
 & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} 0 & a & f & 0 & 0 \\ 0 & 0 & 0 & c & 0 \\ 0 & 0 & 0 & e & 0 \\ 0 & 0 & 0 & 0 & d \\ 0 & b & 0 & 0 & g \end{bmatrix}
\end{array}
\quad
\begin{bmatrix} x1 \\ x2 \\ x3 \\ x4 \\ x5 \end{bmatrix}
$$

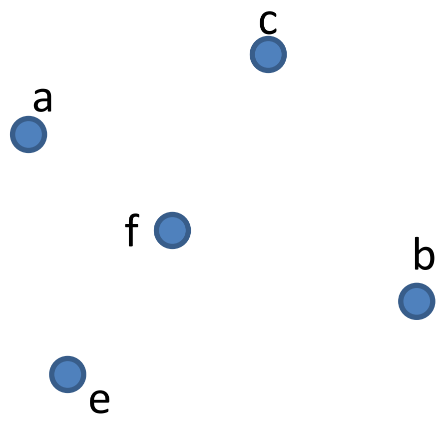
A x

Graph set/multiset duality

- Set/multiset is isomorphic to a graph
  - labeled nodes
  - no edges
- “Opposite” of clique
- Algorithms on sets/multisets can be viewed as graph algorithms
- Usually no particular advantage to doing this but it shows generality of graph algorithms

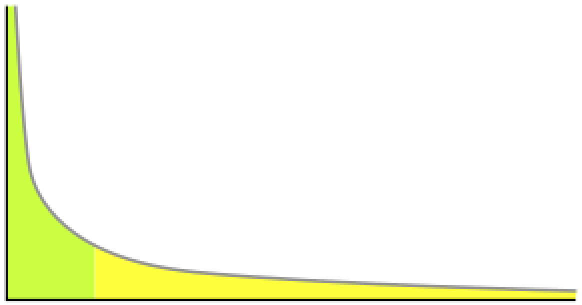
{a,c,f,e,b}

Set



Graph

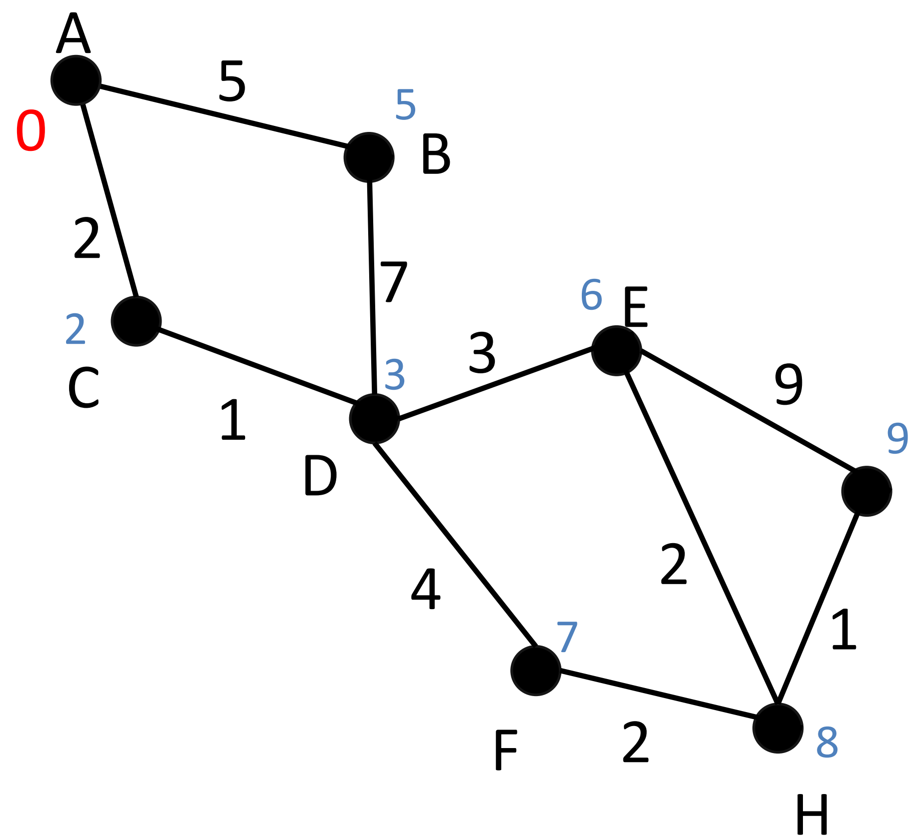
Sparse graph types



- Power-law graphs
  - small number of very high degree nodes (hubs)
  - low diameter:
    - get from a node to any other node in O(1) hops
    - "six degrees of separation" (Karinthy 1929, Milgram 1967), on Facebook, it is 4.74
  - typical of social network graphs like the Internet graph or the Facebook graph
- Uniform-degree graphs
  - nodes have roughly same degree
  - high diameter
  - road networks, IC circuits, finite-element meshes
- Random (Erdös-Rènyi) graphs
  - constructed by random insertion of edges
  - mathematically interesting but few real-life examples

Graph problem:SSSP

- Problem: single-source shortest-path (SSSP) computation
- Formulation:
  - Given an undirected graph with positive weights on edges, and a node called the source
  - Compute the shortest distance from source to every other node
- Variations:
  - Negative edge weights but no negative weight cycles
  - All-pairs shortest paths
  - Breadth-first search: all edge weights are 1
- Applications:
  - GPS devices for driving directions
  - social network analyses: centrality metrics



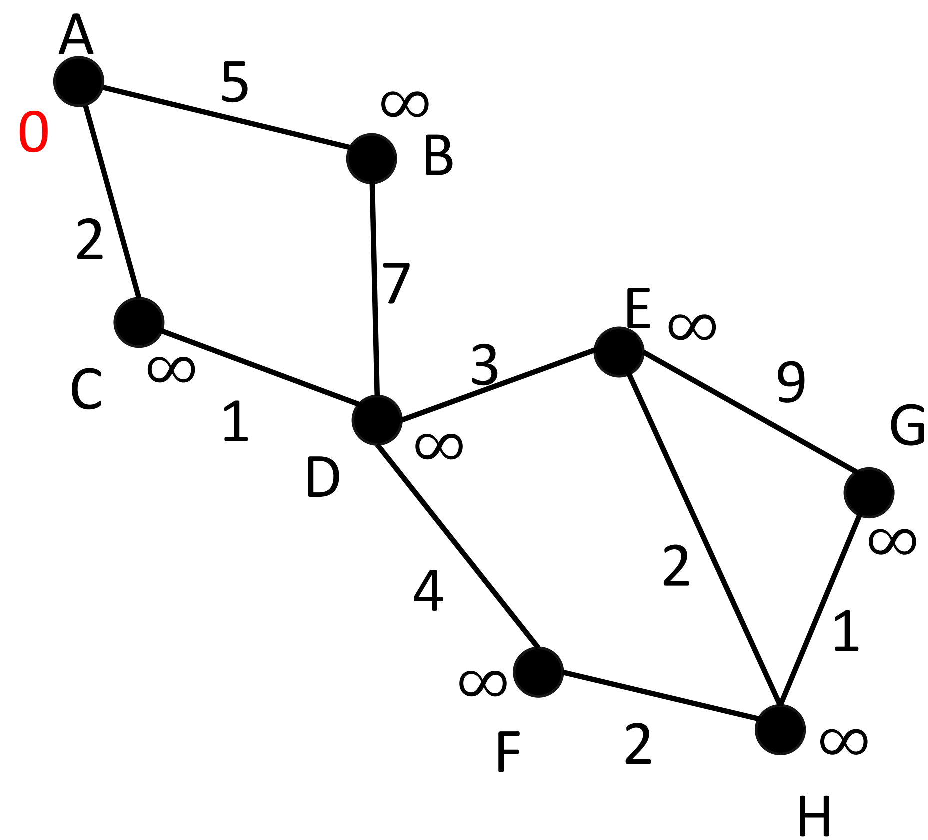
Node A is the source

SSSP Problem

- Many algorithms
  - Dijkstra (1959)
  - Bellman-Ford (1957)
  - Chaotic relaxation (1969)
  - Delta-stepping (1998)
- Common structure:
  - Each node has a label d that is updated repeatedly
    - initialized to 0 for source and ∞ for all other nodes
    - during algorithm: shortest known distance to that node from source
    - termination: shortest distance from source
  - All of them use the same *operator*:

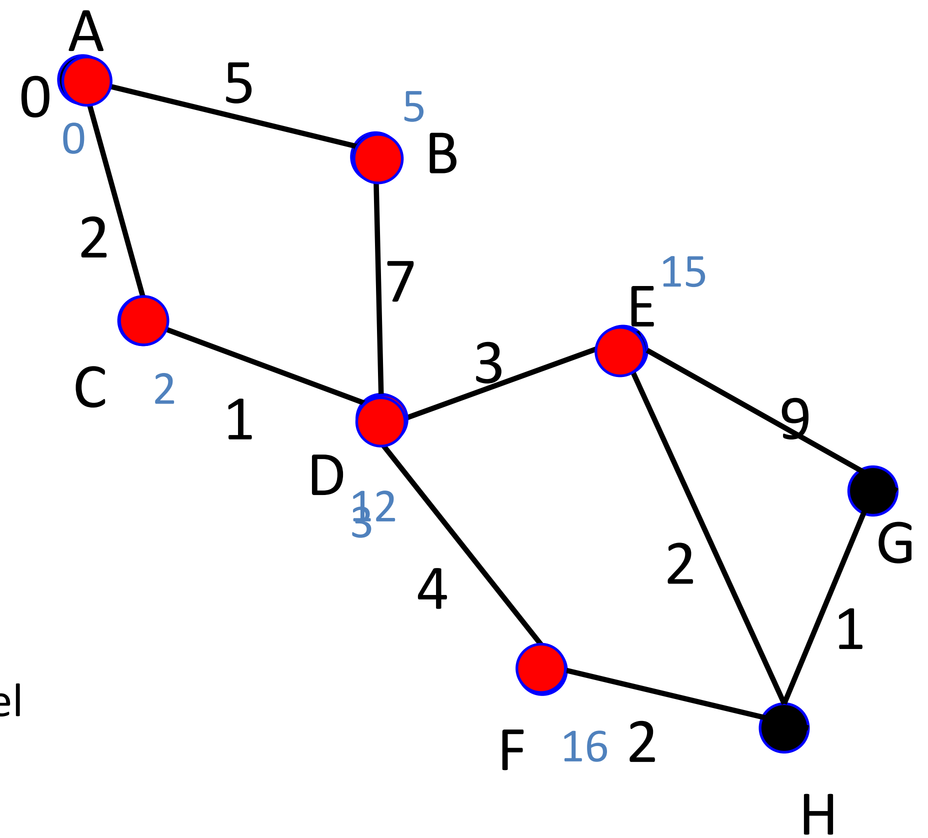
relax-edge(u,v):
  if d[v] > d[u]+w(u,v)
 then d[v] ← d[u]+w(u,v)

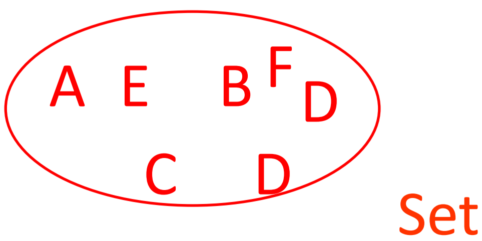
relax-node(u):
  relax all edges connected to u



Chaotic relaxation (1969)

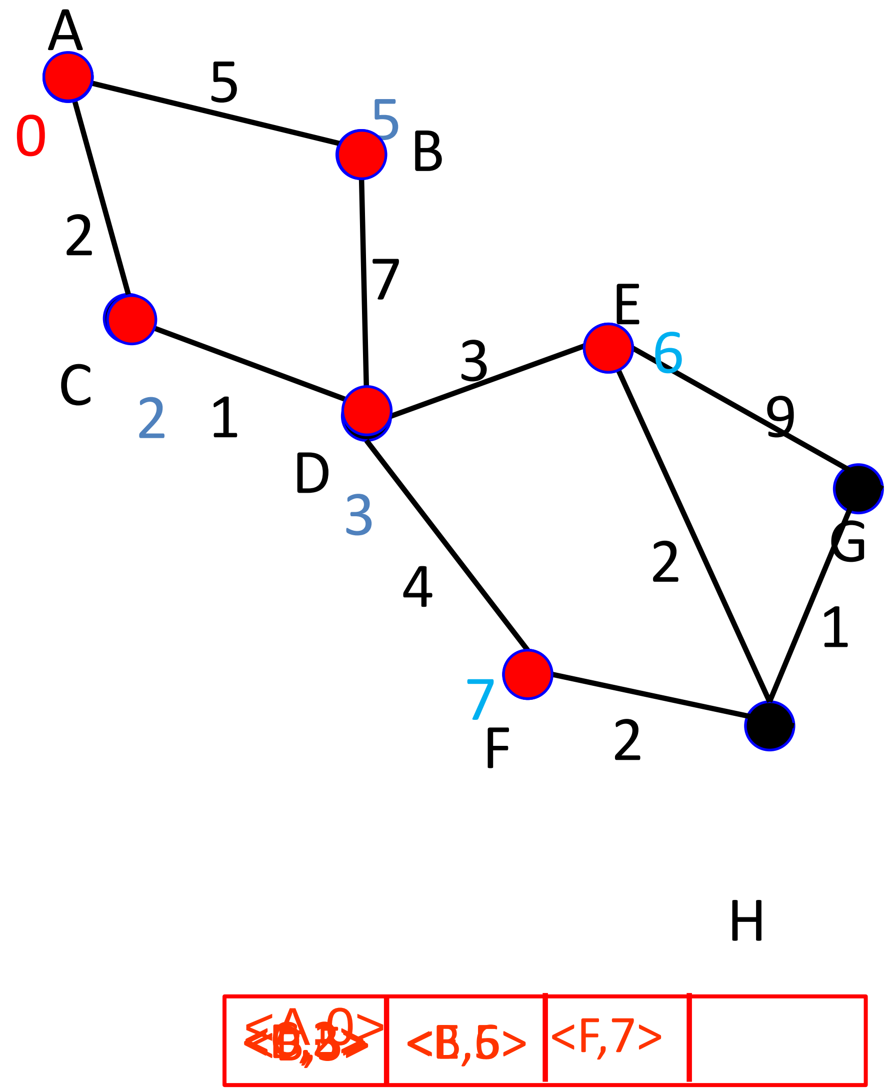
- Active node:
  - node whose label has been updated
  - initially, only source is active
- Schedule for processing nodes
  - pick active nodes at random
- Implementation
  - use a (work)set or multiset to track active nodes
- TAO classification:
  - unstructured graph, data-driven, unordered, local computation
  - compare/contrast with DMR
- Parallelization:
  - process multiple work-set nodes in parallel
  - conflict: two activities may try to update label of the same node
    - eg., B and C may try to update D
  - amorphous data-parallelism
- Main inefficiency: number of node relaxations depends on the schedule
  - can be exponential in the size of graph





Dijkstra's algorithm (1959)

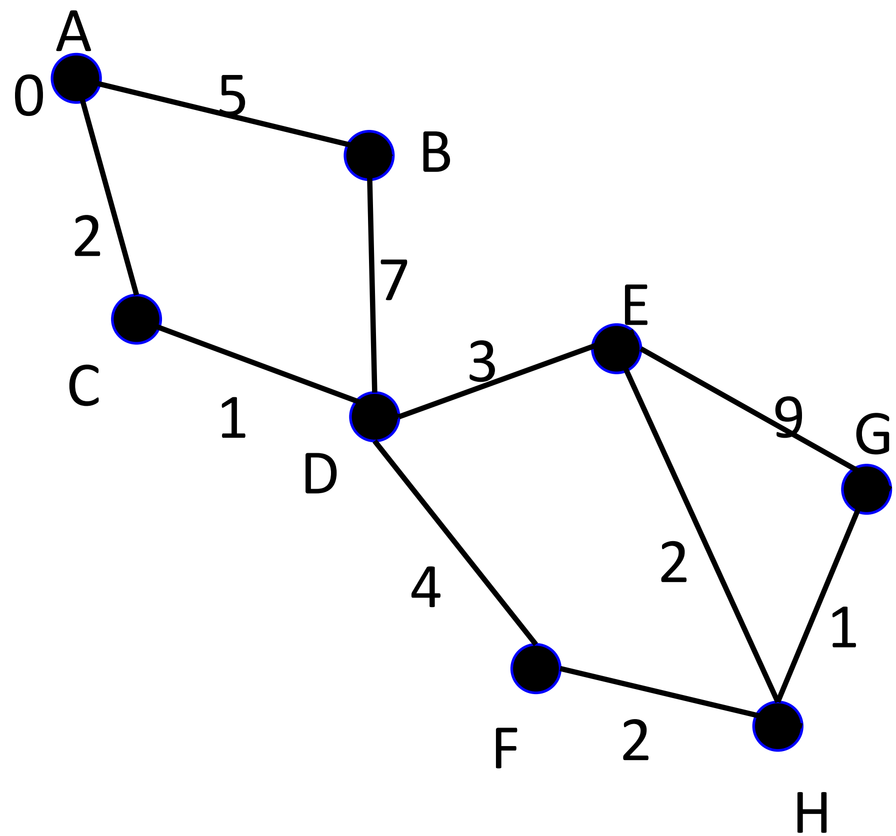
- Active nodes:
  - node whose label has been updated
  - initially, only source is active
- Schedule for processing nodes:
  - prefer nodes with smaller labels since they are more likely to have reached final values
- Implementation of work-set:
  - priority queue of nodes, ordered by label
- Work-efficient ordered algorithm
  - node is relaxed just once
  - O(|E|*lg(|V|))
- TAO classification:
  - unstructured graph, data-driven, ordered, local computation
  - compare with tree summation
- Parallelism
  - nodes with minimal labels can be done in parallel if they don't conflict
  - "level-by-level" parallelization
  - limited parallelism for most graphs
- Main inefficiency:
  - little parallelism in sparse graphs

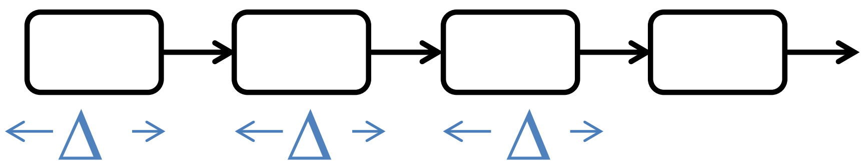


Priority queue

Delta-stepping (1998)

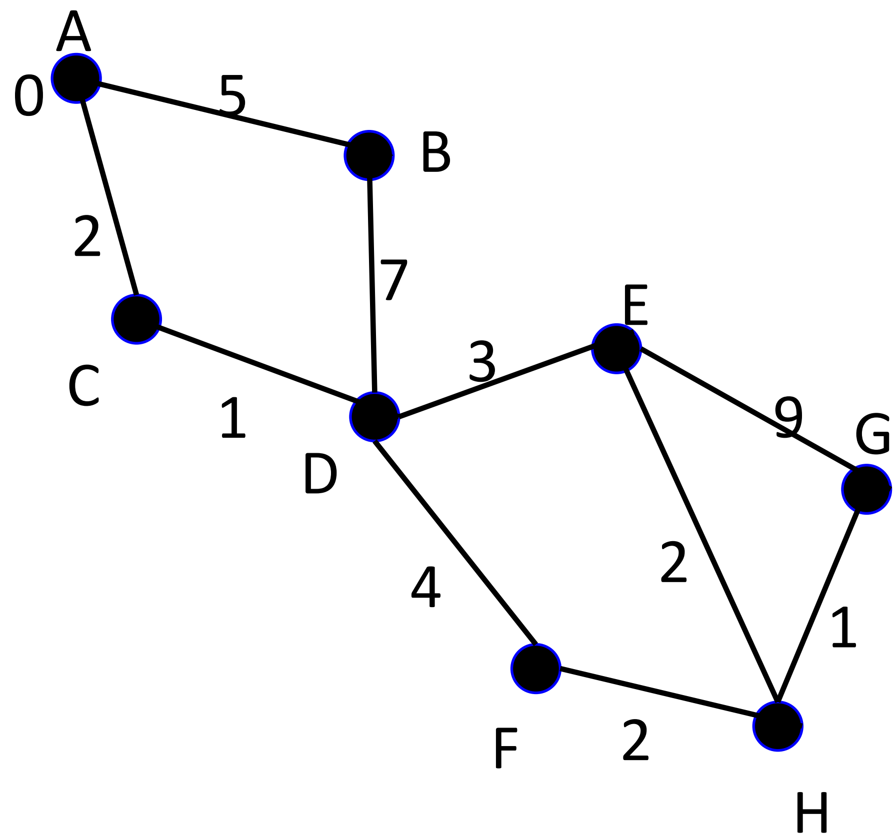
- Controlled chaotic relaxation
  - Exploit the fact that SSSP is robust to priority inversions
  - “soft” priorities
- Implementation of work-set:
  - parameter: Δ
  - sequence of sets
  - nodes whose current distance is between $n\Delta$ and $(n+1)\Delta$ are put in the n^{th} set
  - nodes in each set are processed in parallel
  - nodes in set n are completed before processing of nodes in set (n+1) are started
    - implementation requires barrier synchronization: no worker can proceed past barrier until all workers are at the barrier
- $\Delta = 1$: Dijkstra
- $\Delta = \infty$: Chaotic relaxation
- Picking an optimal Δ :
  - depends on graph and machine
  - high-diameter graph → large Δ
  - find experimentally

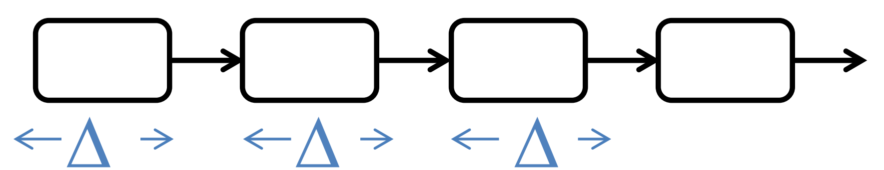




Delta-stepping (II)

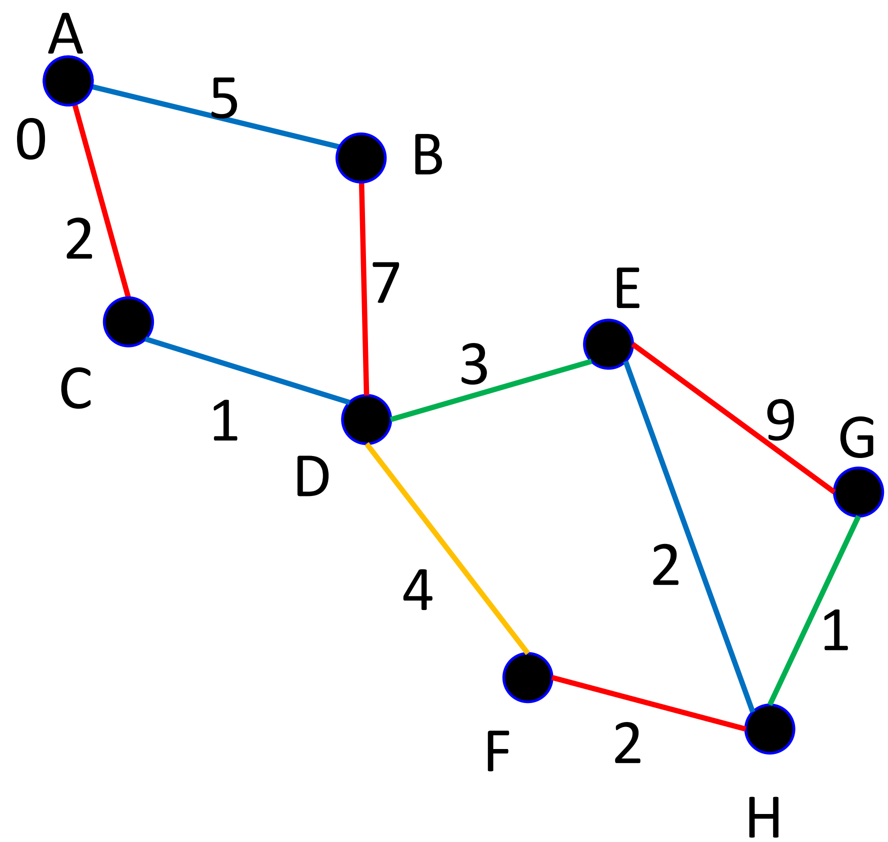
- Standard implementation of work-set:
  - nodes in set n are completed before processing of nodes in set (n+1) are started
  - barrier synchronization between processing of successive sets
- Strict barrier synchronization is not actually needed
  - once set n is empty, some threads can begin executing active nodes from set (n+1) without waiting for all threads to finish executing nodes from set n





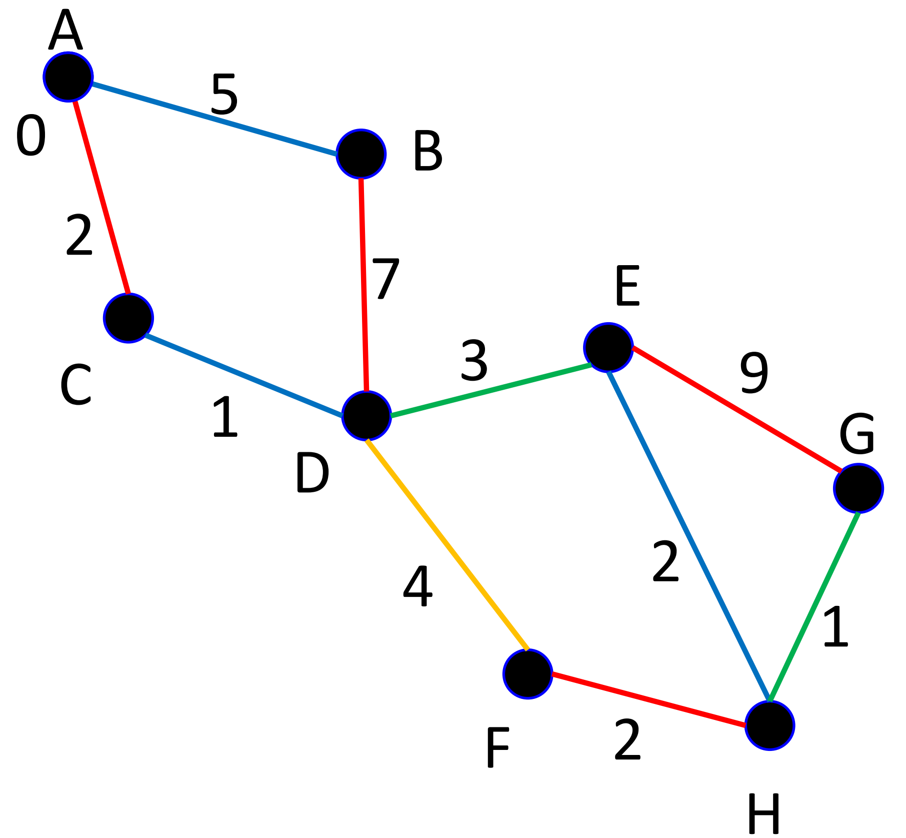
Bellman-Ford (1957)

- Bellman-Ford (1957):
  - Iterate over all edges of graph in any order, relaxing each edge
  - Do this |V| times
  - O(|E|*|V|)
- TAO classification:
  - unstructured graph, topology-driven, unordered, local computation
- Parallelism
  - one approach: optimistic parallelization
  - repeat until no node label changes
    - put all edges into workset
    - workers get edges and apply relaxation operator if they can mark both nodes of edge, until workset is empty
  - can we do better?
    - since we may have to make O(|V|) sweeps over graph, it may be better to preprocess edges to avoid conflicts
    - overhead of preprocessing can be amortized over the multiple sweeps over the graph



Matching

- Given a graph G = (V,E), a matching is a subset of edges such that no edges in the subset have a node in common
  - (eg) {(A,B),(C,D),(E,H)}
  - Not a matching: {(A,B),(A,C)}
- Maximal matching: a matching to which no new edge can be added without destroying matching property
  - (eg) {(A,B),(C,D),(E,H)}
  - (eg) {(A,C),(B,D)(E,G),(F,H)}
  - Can be computed in O(|E|) time using a simple greedy algorithm
- Preprocessing strategy:
  - partition edges into matchings
  - many possible partitions, some better than others



Edges partitioned into matchings

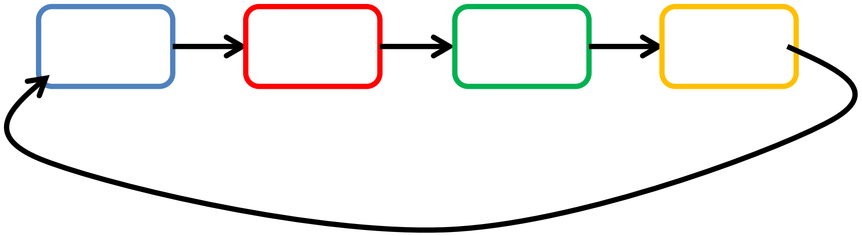
1. {(A,B),(C,D),(E,H)},
2. {(A,C),(B,D),(E,G),(F,H)},
3. {(D,E),(G,H)}
4. {(D,F)}

Execution strategy

- Round-based execution
  - in each round, edges in one matching are processed in parallel w/o neighborhood marking (data parallelism)
  - barrier synchronization between rounds
- Disadvantage of round-based execution
  - all workers must wait at the barrier even if there is just one straggler
  - if we have 2 workers, round-based execution takes 6 steps
- Question: at a high level, there is some similarity to Δ-stepping:
  - sequence of buckets
  - finish one bucket before moving on to next
  - what are the key differences in the implementations?

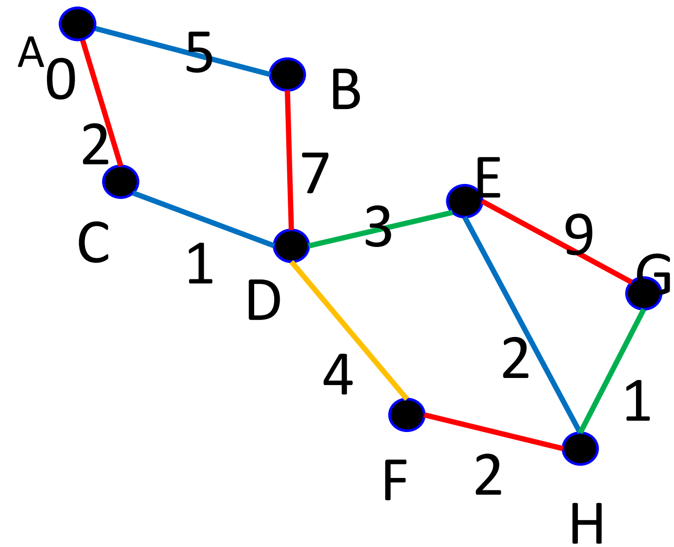
Round-based execution

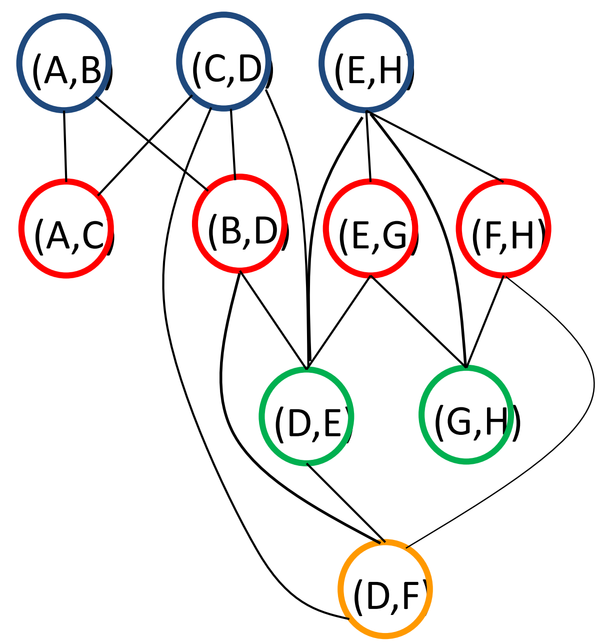
1. {(A,B),(C,D),(E,H)}
2. {(A,C),(B,D),(E,G),(F,H)}
3. {(D,E),(G,H)}
4. {(D,F)}



Another approach: interference graph

- Build interference graph (IG)
  - nodes are activities (SSSP graph edges)
  - edges represent conflicts between activities
    - for our problem, SSSP edges have a node in common
- For our problem
  - each SSSP graph edge represents a task
  - edge between task i and task j in IG if edges corresponding to tasks i and j in SSSP graph have a node in common



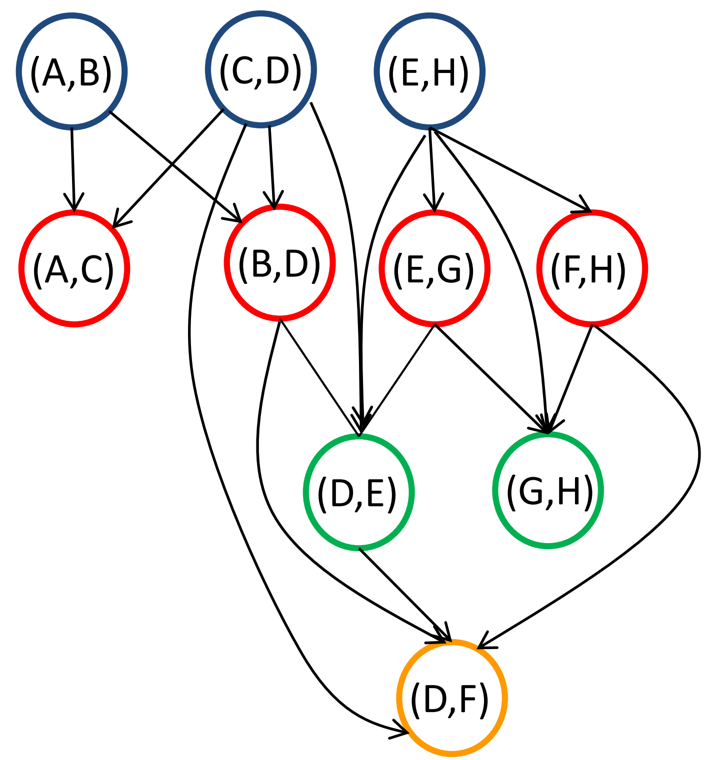


Interference graph

Interference graph → Dependence graph

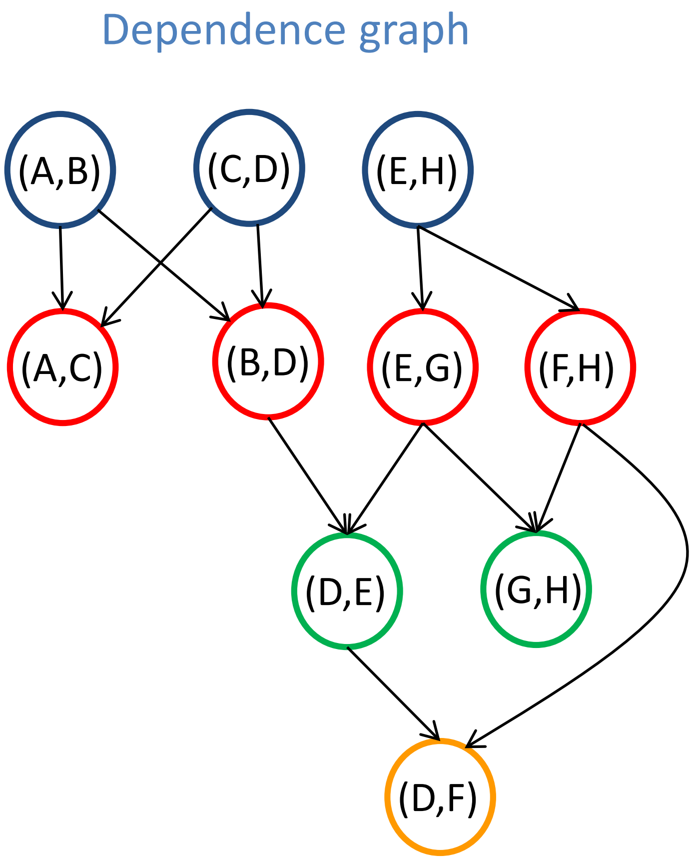
- Generate dependence graph
  - change edges in IG to directed edges (precedence edges)
  - make sure there are no cycles
  - simple approach:
    - number all nodes in interference graph and direct edges from lower numbered nodes to higher numbered nodes
    - many other choices
  - simplification: remove transitive edges

Interference graph



Dependence graph

- Execution using dependence graph
  - each node has counter with number of incoming edges
  - any node with no incoming edges can be executed by a worker
  - when task is done, counters at out-neighbors are decremented, potentially making some of them sources
    - requires marks to ensure correct execution
  - execution terminates when all tasks have been completed
- Fewer ordering constraints between tasks than execution strategy based on matchings and rounds



Inspector-executor

- When is inspector-executor parallelization possible?
  - when active nodes and neighborhoods are known as soon as input is given but before actual computation is executed
    - contrast:
      - static parallelization: active nodes and neighborhoods known at compile-time modulo problem size (example: Jacobi)
      - optimistic parallelization: active nodes and neighborhoods known only after program has been executed in parallel (example: DMR)
  - binding time analysis: when do we know some information regarding program behavior? Example: types
- When is inspector-executor parallelization useful?
  - when overhead of inspector can be amortized over many executions
    - works for Bellman-Ford because we make $O(|V|)$ sweeps over graph
  - when overhead of inspector is small compared to executor
    - sparse Cholesky factorization: inspector is called symbolic factorization, executor is called numerical factorization

Summary of SSSP Algorithms

- Chaotic relaxation
  - parallelism but amount of work depends on execution order of active nodes
  - unordered, data-driven algorithm: use sets/multisets
- Dijkstra's algorithm
  - work-efficient but difficult to extract parallelism
    - level-by-level parallelism
  - ordered, data-driven algorithm: use priority queues
- Delta-stepping
  - controlled chaotic relaxation: parameter Δ
  - Δ permits trade-off between parallelism and extra work
- Bellman-Ford algorithm
  - Inspector-executor parallelization:
    - inspector: use matchings or dependence graph to find parallelism after input is given

Machine learning

- Many machine learning algorithms are sparse graph algorithms
- Examples:
  - Page rank: used to rank webpages to answer Internet search queries
  - Recommender systems: used to make recommendations to users in Netflix, Amazon, Facebook etc.

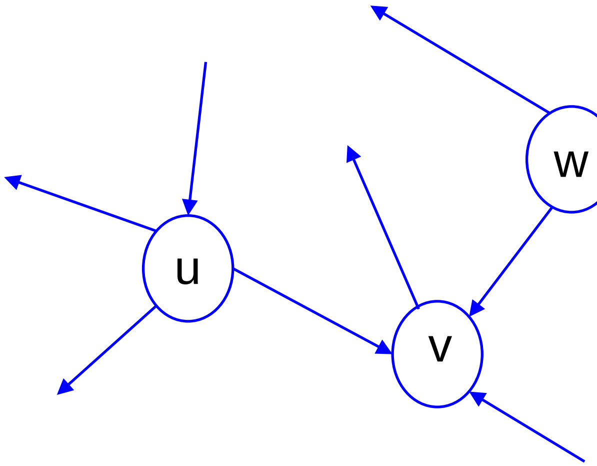
Web search

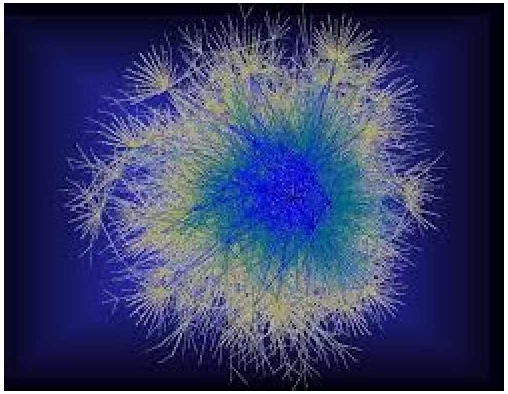
- When you type a set of keywords to do an Internet search, which web-pages should be returned and in what order?
- Basic idea:
  - offline:
    - crawl the web and gather webpages into data center
    - build an index from keywords to webpages
  - online:
    - when user types keywords, use index to find all pages containing the keywords
  - key problem:
    - usually you end up with tens of thousands of pages
    - how do you rank these pages for the user?

Ranking pages

- Manual ranking
  - Yahoo did something like this initially, but this solution does not scale
- Word counts
  - order webpages by how many times keywords occur in webpages
  - problem: easy to mess with ranking by having lots of meaningless occurrences of keyword
- Citations
  - analogy with citations to articles
  - if lots of webpages point to a webpage, rank it higher
  - problem: easy to mess with ranking by creating lots of useless pages that point to your webpage
- PageRank
  - extension of citations idea
  - weight link from webpage A to webpage B by “importance” of A
  - if A has few links to it, its links are not very “valuable”
  - how do we make this into an algorithm?

Web graph



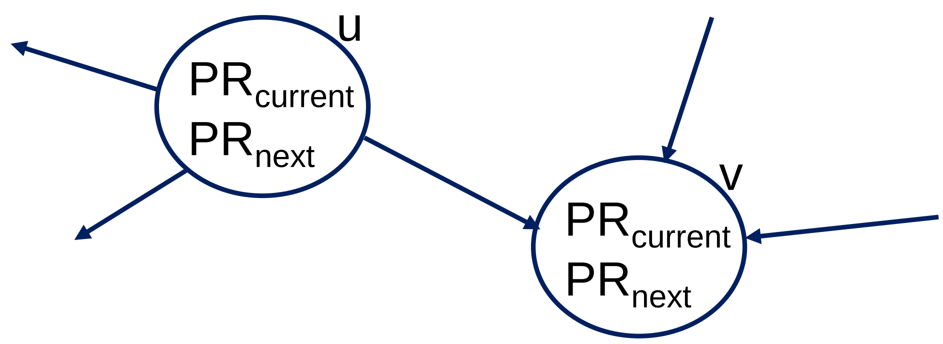


Webgraph from commoncrawl.org

- Directed graph: nodes represent webpages, edges represent links
  - edge from u to v represents a link in page u to page v
- Size of graph: commoncrawl.org (2012)
  - 3.5 billion nodes
  - 128 billion links
- Intuitive idea of pageRank algorithm:
  - each node in graph has a weight (pageRank) that represents its importance
  - assume all edges out of a node are equally important
  - importance of edge is scaled by the pageRank of source node

PageRank (simple version)

Graph G = (V,E)
|V| = N



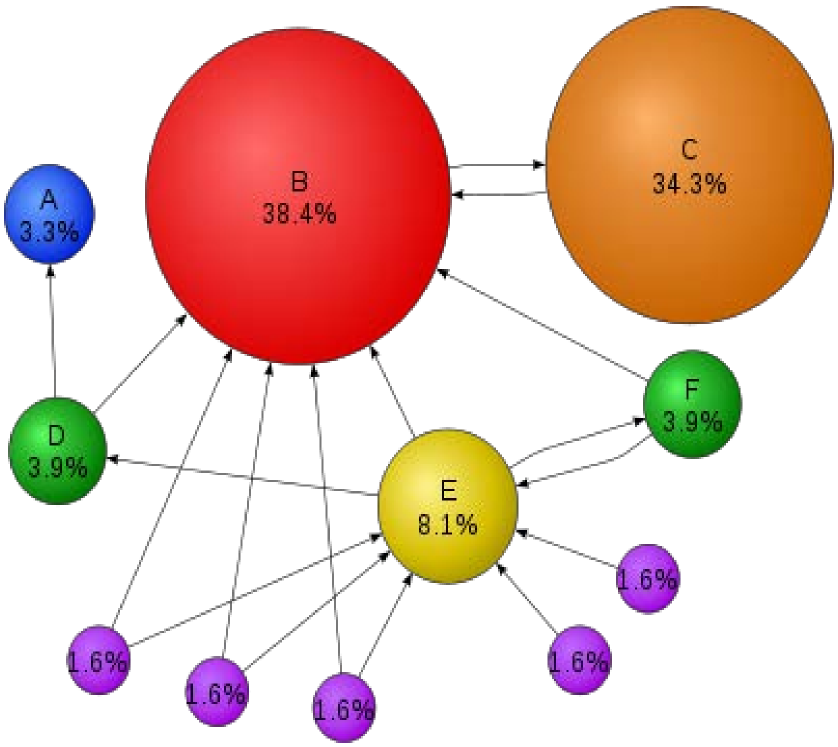
- Iterative algorithm:
  - compute a series PR_0, PR_1, PR_2, ... of node labels
- Iterative formula:
  - $\forall v \in V.\ PR_0(v) = 1/N$
  - $\forall v \in V.\ PR_{i+1}(v) = \sum_{u \in in-neighbors(v)} \frac{PR_i(u)}{out-degree(u)}$
- Implement with two fields $PR_{current}$ and PR_{next} in each node

Page Rank (contd.)

- Small twist needed to handle nodes with no outgoing edges
- Damping factor: d
  - small constant: 0.85
  - assume each node may also contribute its pageRank to a randomly selected node with probability (1-d)
- Iterative formula
  - $\forall v \in V.\ PR_0(v) = \frac{1}{N}$
  - $\forall v \in V.\ PR_{i+1}(v) = \frac{1-d}{N} + d * \sum_{u \in in-neighbors(v)} \frac{PR_i(u)}{out-degree(u)}$

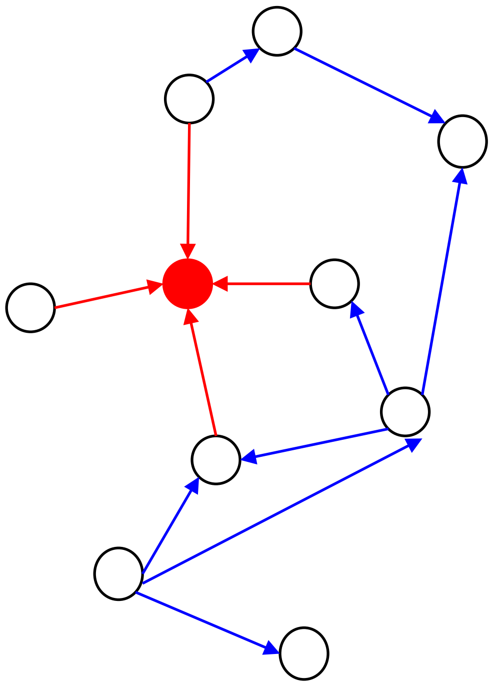
PageRank example

- Nice example from Wikipedia
- Note
  - B and E have many in-edges but pageRank of B is much greater
  - C has only one in-edge but high pageRank because its in-edge is very valuable
- Caveat:
  - search engines use many criteria in addition to pageRank to rank webpages



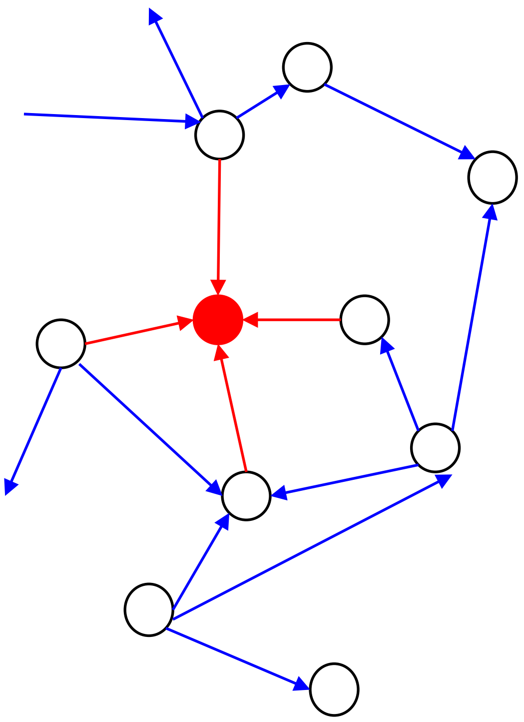
Parallelization of pageRank

- TAO classification
  - topology: unstructured graph
  - active nodes:
    - topology-driven, unordered
  - operator: local computation
- PageRank$_{next}$ values at all nodes can be computed in parallel
- Which algorithm does this remind you of?
  - Jacobi iteration with 5-point stencil
  - main difference: topology
    - 5-point stencil: regular grid, uniform degree graph
    - web-graph: power-law graph
    - this has a major impact on implementation, as we will see later



PageRank discussion

- Vertex program (GraphLab):
  - value at node is updated using values at immediate neighbors
  - very limited notion of neighborhood but adequate for pageRank and some ML algorithms
- CombBlas: combinatorial BLAS
  - generalized sparse MVM: + and * in MVM are generalized to other operations like $\vee$ and $\wedge$
  - adequate for pageRank
- Interesting application of TAO
  - standard pageRank is topology-driven
  - can you think of a data-driven version of pageRank?

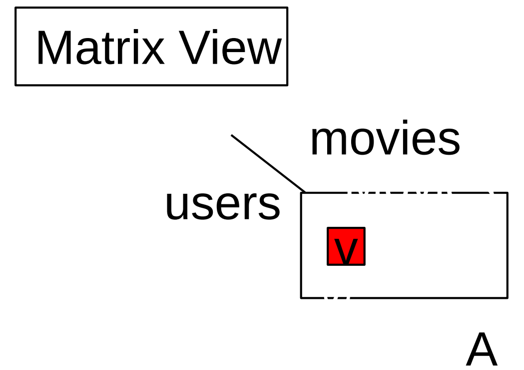


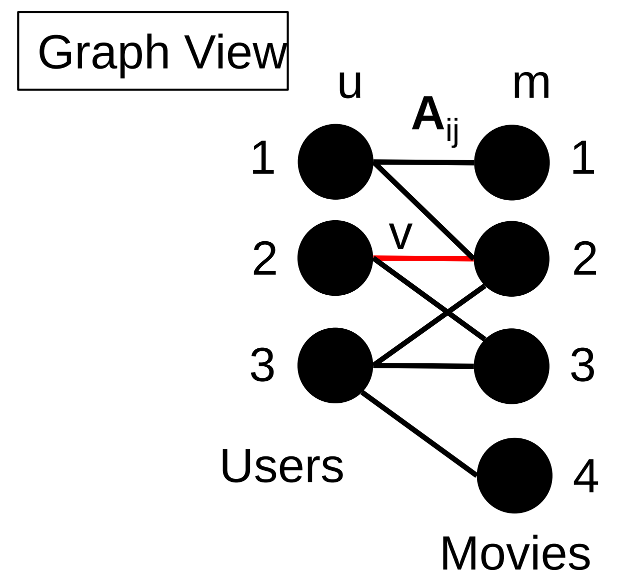
Recommender system

- Problem
  - given a database of users, items, and ratings given by each user to some of the items
  - predict ratings that user might give to items he has not rated yet (usually, we are interested only in the top few items in this set)
- Netflix challenge
  - in 2006, Netflix released a subset of their database and offered $1 million prize to anyone who improved their algorithm by 10%
  - triggered a lot of interest in recommender systems
  - prize finally given to BellKor's Pragmatic Chaos team in 2009

Data structure for database

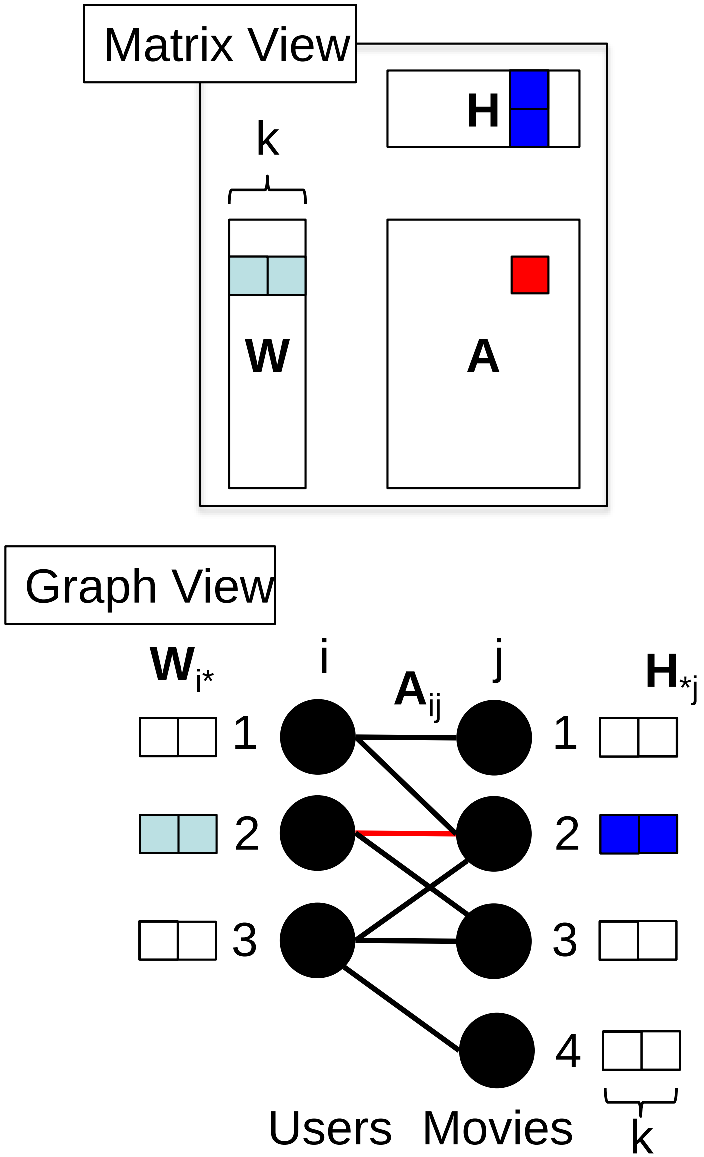
- Sparse matrix view:
  - rows are users
  - columns are movies
  - A(u,m) = v is user u has given rating v to movie m
- Graph view:
  - bipartite graph
  - two sets of nodes, one for users, one for movies
  - edge (u,m) with label v
- Recommendation problem:
  - predict missing entries in sparse matrix
  - predict labels of missing edges in bipartite graph





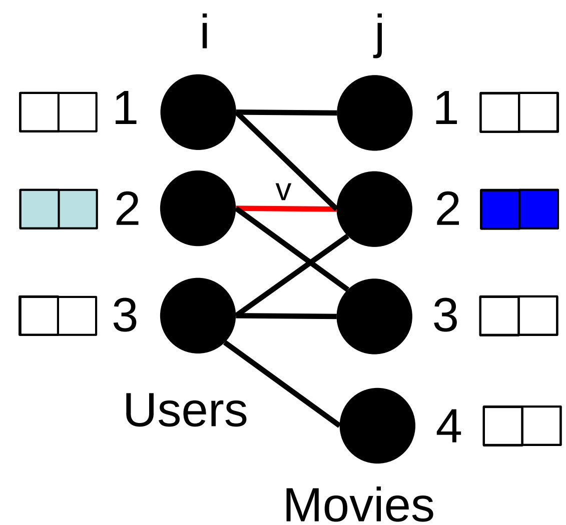
One approach: matrix completion

- Optimization problem
  - Find m×k matrix **W** and k×n matrix **H** (k << min(m,n)) such that **A ≈ WH**
  - Low-rank approximation
  - **H** and **W** are dense so all missing values are predicted
- Graph view
  - Label of user nodes i is vector corresponding to row W_{i*}
  - Label of movie node j is vector corresponding to column H_{*j}
  - If graph has edge (u,m), inner product of labels on u and m must be approximately equal to label on edge



One algorithm:SGD

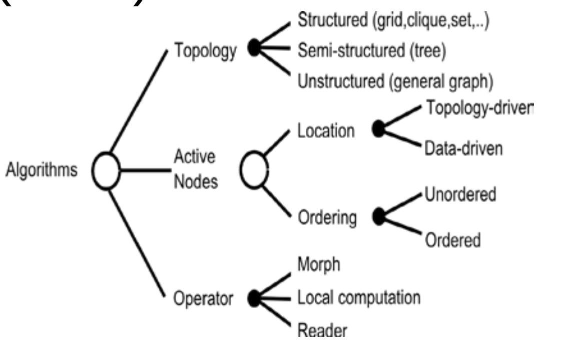
- Stochastic gradient descent (SGD)
- Iterative algorithm:
  - initialize all node labels to some arbitrary values
  - iterate until convergence
    - visit all edges (u,m) in some order and update node labels at u and m based on the residual
- TAO analysis:
  - topology: unstructured (power-law) graph
  - active edges: topology-driven, unordered
  - operator: local computation
- Parallelism in SGD:
  - edges that form a matching can be processed in parallel
- What algorithm does this remind you of?
  - Bellman-Ford



Summary of discussion of algorithms

What we have learned (I)

- Data-centric view of algorithms
- TAO classification
- Unordered algorithms
  - lots of parallelism for large problem sizes
  - soft priorities need in many/most algorithms (e.g. chaotic SSSP)
  - don't-care non-determinism in some algorithms (DMR)
- Topology-driven algorithms
  - iterate over data structure
  - no explicit work-list needed



- Data-driven algorithms
  - need efficient parallel work-list (put/get)
  - may need to support soft priorities (e.g. chaotic SSSP)
- Some problems
  - have both ordered and unordered algorithms (e.g. SSSP)
  - have both topology-driven and data-driven algorithms (e.g. SSSP, pageRank)
  - data-driven algorithm may be more work-efficient than topology-driven one

What we have learned (II)

- Amorphous data-parallelism
  - data-parallelism is special case
- Parallelization strategies:
  - key question: when do you know the active nodes and neighborhoods?
    - static: known at compile-time (modulo problem size)
    - inspector-executor: after input is given but before program is executed
    - optimistic: after program has finished execution
- Implementation concepts:
  - edge matchings in graphs
  - synchronization
    - barrier synchronization: coarse-grain
    - marking graph elements, get/put on work-lists: mutual exclusion, fine-grain

What we will study (I)

- Parallel architectures: workers can be heterogeneous and may be organized in different ways
  - vector architectures, GPUs, FPGAs
  - shared and distributed-memory architectures
- Synchronization: coordination between workers
  - coarse-grain synchronization: barriers
  - fine-grain synchronization: locks, lock-free instructions
    - application: marking of graph elements, work-lists, mutual exclusion
- Scheduling activities on workers
  - locality: temporal, spatial, network
  - load-balancing
  - minimize conflicts between concurrent activities (optimistic parallelization)
- Concurrent data structure implementations
  - graphs/sparse matrices
  - work sets/multisets
    - soft priorities
  - priority queues

What we will study (II)

- Programming language issues
  - how do we express information about parallelism, locality, scheduling, data structure implementations?
  - how do we simplify parallel programming so most application programmers can benefit from parallelism without having to write parallel code?
- Parallel notations and libraries
  - shared-memory: pThreads, MPI, Galois
  - distributed-memory: MPI

Lock-step execution

- With 2 workers, we can execute all tasks in 5 steps with the right schedule
  - (A,B), (E,H)
  - (C,D), (E,G)
  - (B,D), (F,H)
  - (D,E), (G,H)
  - (A,C), (D,F)

step →

worker ↓

	0	1	2	3	4
P0	(A,B)	(C,D)	(B,D)	(D,E)	(A,C)
P1	(E,H)	(E,G)	(F,H)	(G,H)	(D,F)

- Two implementations:
  - synchronous execution: assign work to workers as specified above (no need for work-lists) and use barrier synchronization between steps
  - autonomous execution:
    - threads proceed independently of each other
    - each node of dependence graph has an integer counter initialized to the number of in-edges
    - free thread grabs any node with zero counter, executes it, and then updates counters at out-neighbors in the DAG
    - updating counters needs fine-grain synchronization: must mark nodes before updating their counters