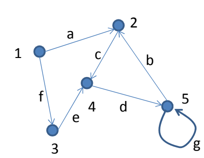
Graph Algorithms

Overview

- Graphs are very general data structures
  - 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:
    - many of these applications can be formulated as sparse matrix-vector product

Graph-matrix duality

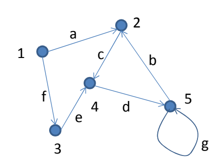
- Graph (V,E) as a matrix
  - Choose an ordering of vertices
  - Number them sequentially
  - Fill in |V|x|V| matrix
  - Called "adjacency matrix" of graph
- Observations:
  - Diagonal entries: weights on self-loops
  - Symmetric matrix ←→ undirected graph
  - Lower triangular matrix ←→ no edges from lower numbered nodes to higher numbered nodes
  - Dense matrix ←→ clique (edge between every pair of nodes)



	1	2	3	4	5
1	0	a	f	0	0
2	0	0	0	c	0
3	0	0	0	e	0
4	0	0	0	0	d
5	0	b	0	0	g

Matrix-vector multiplication

- Matrix computation: y = Ax
- 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 of its neighbors j, scaled by the label on edge (i,j)
- Observation:
  - Graph perspective shows dense MVM is just a special case of sparse MVM

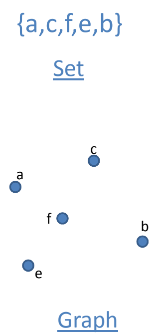


	1	2	3	4	5	x
1	0	a	f	0	0	$x1$
2	0	0	0	c	0	$x2$
3	0	0	0	e	0	$x3$
4	0	0	0	0	d	$x4$
5	0	b	0	0	g	$x5$

A x

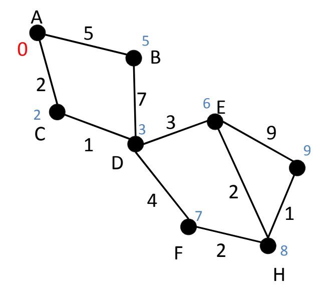
Graph set/multiset duality

- 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



Graph algorithm examples

- 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
- Applications:
  - GPS devices for driving directions
  - social network analyses: centrality metrics



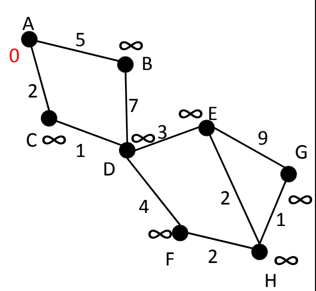
Node A is the source

SSSP Problem

- Many algorithms
  - Dijkstra (1959)
  - Bellman-Ford (1957)
  - Chaotic relaxation (1969)
  - Delta-stepping (1998)
- Common structure:
  - Each node has a label d containing shortest known distance to that node from source
    - Initialized to 0 for source and infinity for all other nodes
  - Key operations:

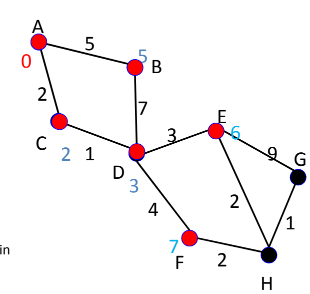
    relax-edge(u,v):
    if d[v] > d[u]+w(u,v)
    then d[v] ← d[u]+w(u,v)

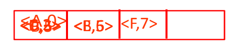
    relax-node(u):
    relax all edges connected to u



SSSP algorithms (I)

- Dijkstra's algorithm (1959):
  - priority queue of nodes, ordered by shortest distance known to node
  - iterate over nodes in priority order
  - node is relaxed just once
  - work-efficient: O(|E|*lg(|V|))
- Active nodes:
  - nodes in PQ: level has been lowered but node has not yet been relaxed
- Key data structures:
  - Graph
  - Work set/multiset: ordered
    - Priority queue
- Parallelism in algorithm
  - Edges connected to node can be relaxed in parallel
  - Difficult to relax multiple nodes from priority queue in parallel
  - Little parallelism for sparse graphs
- Ordered algorithm

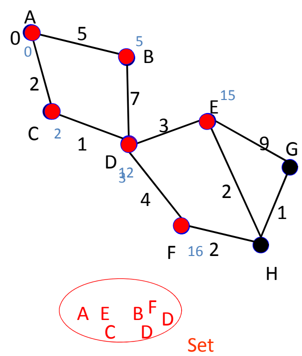




Priority queue

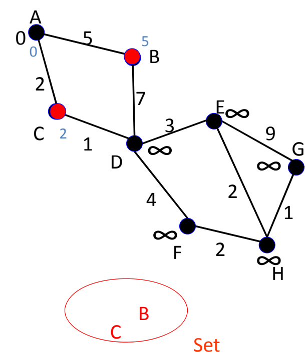
SSSP algorithms (II)

- Chaotic relaxation (1969):
  - use set to track active nodes
  - iterate over nodes in any order
  - nodes can be relaxed many times
    - may do more work than Dijkstra
- Key data structures:
  - Graph
  - Work set/multiset: unordered
- Parallelization:
  - process multiple work-set nodes
  - need concurrent data structures
    - concurrent set/multiset: elements are added/removed correctly
    - concurrent graph: simultaneous updates to node happen correctly
- Unordered algorithm



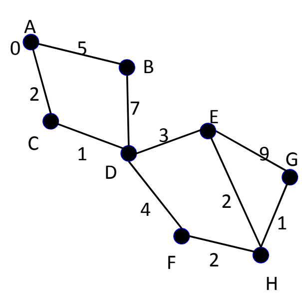
SSSP algorithms (II contd.)

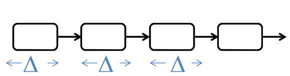
- Need for synchronization at graph nodes
  - Suppose nodes B and C are relaxed simultaneously
  - Both relaxations may update value at D
    - Value at D is infinity
    - Relax-C operation reads this value and wants to update it to 3.
    - At the same time, Relax-D operation reads D's value and wants to update it to 12
    - If the two updates are not sequenced properly, final value at D after both relaxations may be 12, which is incorrect
  - One solution: ensure that the "read-modify-write" in edge relaxation is "atomic" – no other thread can read or write that location while the read-modify-write is happening
- Also need synchronization at node being relaxed to ensure its value is not changed by some other core when the node relaxation is going on



SSSP algorithms (III)

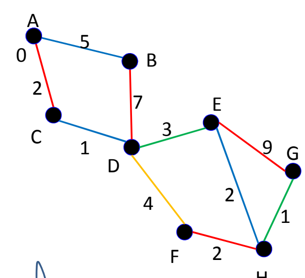
- Delta-stepping (1998)
  - variation of chaotic relaxation
  - active nodes currently closer to source are more likely to be chosen for processing from set
- Work-set/multiset:
  - Parameter: Δ
  - Sequence of sets
  - Nodes whose current distance is between $n\Delta$ and $(n+1)\Delta$ are put in the 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
- Δ = 1: Dijkstra
- $\Delta = \infty$: Chaotic relaxation
- Picking an optimal Δ :
  - depends on graph and machine
  - Do experimentally





SSSP algorithms (IV)

- Bellman-Ford (1957):
  - Iterate over all edges of graph in any order, relaxing each edge
  - Do this |V| times
  - O(|E|*|V|)
- Parallelization
  - Iterate over set of edges
  - Inspector-executor: use graph matching to generate a conflict-free schedule of edge relaxations after input graph is given
  - Edges in a matching do not have nodes in common so they can be relaxed without synchronization
  - Barrier synchronization between successive stages in schedule

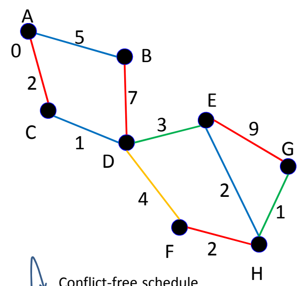


Conflict-free schedule

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

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
- Maximum matching: matching that contains the largest number of edges
  - (eg) {(A,C),(B,D)(E,G),(F,H)}
  - Can be computed in time O(sqrt(|V|)|E|)



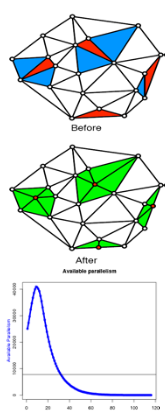
Conflict-free schedule
1. {(A,B),(C,D),(E,H)},
2. {(A,C),(B,D),(E,G),(F,H)},
3. {(D,E),(G,H)}
4. {(D,F)}

Summary of SSSP Algorithms

- Ordered algorithms: use priority queues
  - Dijkstra's algorithm
    - Work-efficient but difficult to extract parallelism
- Unordered algorithms: use sets/multisets
  - Chaotic relaxation
    - Parallelism but amount of work depends on schedule
    - Fine-grain synchronization
- Ordered outer/unordered inner
  - Delta-stepping
    - Controlled chaotic relaxation: parameter Δ
    - Δ permits trade-off between parallelism and extra work
    - Both fine-grain and coarse-grain synchronization
  - Bellman-Ford algorithm
    - Inspector: use matching to find contention-free schedule for inner loop
    - Executor: perform relaxations using barriers
    - Only coarse-grain synchronization

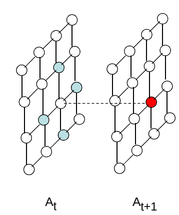
Delaunay Mesh Refinement

```
Mesh m = /* read in mesh */
WorkList wl;
wl.add(m.badTriangles());
while (true) {
    if ( wl.empty() ) break;
        Element e = wl.get();
        if (e no longer in mesh) continue;
        Cavity c = new  Cavity(e);//determine new cavity
        c.expand();
        c.retriangulate();
        m.update(c);//update mesh
        wl.add(c.badTriangles());
}
```



Stencil computation: Jacobi iteration

- Finite-difference method for solving pde's
  - discrete representation of domain: grid
- Values at interior points are updated using values at neighbors
  - values at boundary points are fixed
- Data structure:
  - dense arrays
- Parallelism:
  - values at next time step can be computed simultaneously
  - parallelism is not dependent on runtime values
- Compiler can find the parallelism
  - spatial loops are DO-ALL loops

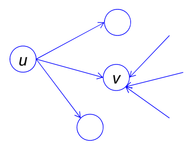


Jacobi iteration, 5-point stencil

```
//Jacobi iteration with 5-point stencil
//initialize array A
for time = 1, nsteps
   for <i,j> in [2,n-1]x[2,n-1]
       temp(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
   for <i,j> in [2,n-1]x[2,n-1]:
       A(i,j) = temp(i,j)
```

Page Rank

- Used to determine relative importance of webpages by examining links between pages
- Abstraction:
  - graph in which nodes are webpages and edges are links
  - nodes have weights [0,1]
    - initialized to 1/N (N is number of nodes)
    - when algorithm terminates, weight is heuristic measure of importance
- Core algorithm: iterative step repeated a few times
  - each node u contributes an equal fraction of its own current weight to its immediate neighbors in the graph

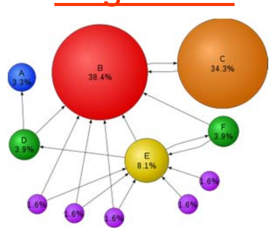


$PR_{i+1}(v) = \sum_{u \in Neighbors(v)} \frac{PR_i(u)}{Degree(u)}$

Page Rank (contd.)

- Intuition behind page rank:
  - if you do a random walk on the graph, how likely is it that you end up at various nodes in the graph in the limit?
- Small twist needed to handle nodes with no outgoing edges
- Damping factor: d
  - Small constant: 0.85
  - Assume that each node, you may also take a random jump to any other node with probability (1-d)

PageRank



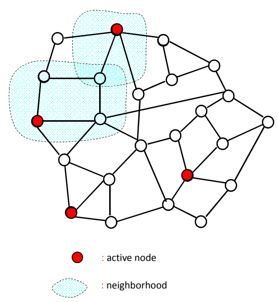
- $PR_{i+1}(v) = \frac{(1-d)}{N} + d * \sum_{u \in Neighbors(v)} \frac{PR_i(u)}{Degree(u)}$
  - N is the number of nodes in graph
  - *Neighbors(v)*: set of nodes *u* with edge *(u,v)*
  - *Degree(u)*: number of outgoing edges

Questions

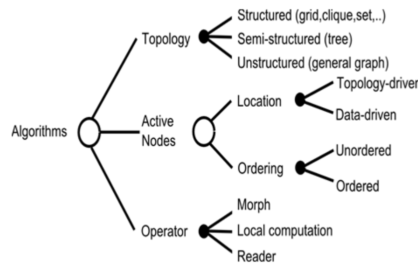
- We have seen several algorithms from a number of problem domains
  - Networks: Dijkstra SSSP, chaotic-relaxation SSSP, delta-stepping SSSP, Bellman-Ford
  - Graphics: Delaunay mesh refinement
  - Finite-differences: Stencil computations
  - Big data: Page rank
- What are the right abstractions for seeing commonalities and differences between these algorithms?

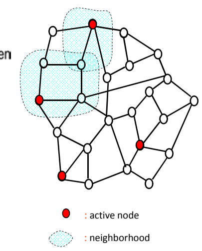
Abstraction of algorithms

- Operator formulation
  - Active elements: nodes or edges where there is work to be done
  - Operator: computation at active element
    - Activity: application of operator to active element
    - Neighborhood: graph elements read or written by activity
  - Ordering: order in which active elements must appear to have been processed
    - Unordered algorithms: any order is fine (eg. chaotic relaxation, Jacobi, PageRank)
    - Ordered algorithms: algorithm-specific order (eg. Dijkstra)



TAO analysis: algorithm abstraction





Dijkstra SSSP: general graph, data-driven, ordered, local computation
Chaotic relaxation SSSP: general graph, data-driven, unordered, local computation
Bellman-Ford SSSP: general graph, topology-driven, unordered, local computation
Delta-stepping SSSP: general graph, data-driven, ordered, local computation
Delaunay mesh refinement: general graph, data-driven, unordered, morph
Jacobi: grid, topology-driven, unordered, local computation

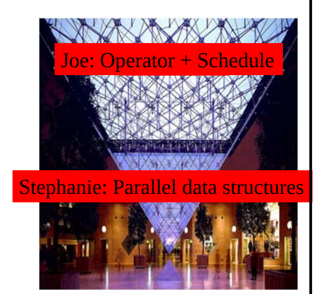
22

Infrastructure for graph algorithms

- Concurrent data structures:
  - Concurrent graph data structure
  - Concurrent set/bag, priority queue
  - Can be very complex to implement
- One software architecture:
  - Exploit Wirth's equation:
    - Program = Algorithm + Data Structure
    - Parallel program = Parallel algorithm + Parallel data structure
      = Operator + Schedule + Parallel data structure
  - Provide a library of concurrent data structures
  - Programmer specifies
    - operator
    - schedule for applying operator at different active elements
- This is the approach we use in the Galois project

Two-level infrastructure: Galois

- Small number of expert programmers must support a large number of application programmers
  - cf. SQL
- Galois project:
  - Program = Algorithm + Data structure (Wirth)
  - Library of concurrent data structures and runtime system written by expert programmers
  - Application programmers code in sequential C++
    - All concurrency control is in data structure library and runtime system



Parallel program = Operator + Schedule + Parallel data structures

Summary

- Graph algorithms can be very complex
  - Work may be created dynamically
  - Different orders of doing work may result in different amounts of work
  - Parallelism may not be known until runtime
  - Underlying graph structure may change dynamically
- SSSP algorithms illustrate most of this complexity so the SSSP problem is a good model problem for the study of parallel graph algorithms
- Operator formulation and TAO analysis are useful abstractions for understanding parallelism in algorithms
- Galois project: software architecture is based on these ideas
  - Library of concurrent data structures written by expert programmers
  - Joe programmer writes C++ code to specify operator and schedule