Parallel Graph Algorithms

Design and Analysis of Parallel Algorithms

5DV050 Spring 2012

Part I

Introduction

Overview

- Graphs—definitions, properties, representation
- Minimal spanning tree
  - Prim's algorithm
- Shortest paths (1-to-all)
  - Dijkstra's algorithm
- Shortest paths (all-to-all)
  - Algorithm based on matrix multiplication
  - Dijkstra's algorithm
    - Source partitioned
    - Source parallel
  - Floyd's algorithm
- Transitive closure
- Connected components

Graphs: Definitions

Graphs

- $G = (V, E)$: V is the set of *vertices* and E is the set of *edges*

Undirected graphs

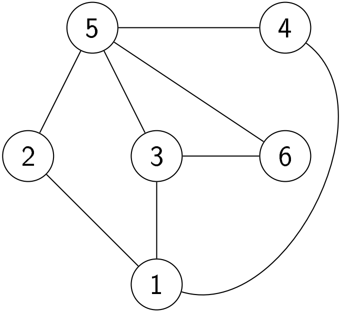
- An *edge* $e \in E$ is an unordered pair $\{u, v\}$ where $u, v \in V$

Directed graphs

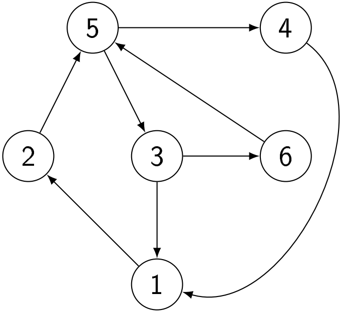
- An *arc* $e \in E$ is an ordered pair (u, v) directed from u to v
- A *path* from u to v is a sequence $u, \ldots, v$ of vertices where consecutive vertices correspond to an arc
    - *Simple path*: all vertices are distinct
    - *Cycle*: $u = v$
    - *Acyclic*: contains no cycles

Examples of graphs

Undirected



Directed



Graphs: Properties

- A graph is *connected* if it exists a path between every pair of vertices
- A graph is *complete* if it exists an edge between every pair of vertices
- $G' = (V', E')$ is a *sub-graph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$
- A *tree* is a connected acyclic graph
- A *forest* consists of several trees
- A graph $G = (V, E)$ is *sparse* if $|E|$ is much smaller than $|V|^2$

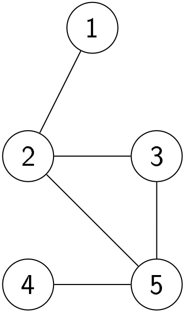
Weighted graphs:

- $G = (V, E, w)$, where w is a real-valued function defined on E (every edge/arc has a value)
- The weigh of a graph is the sum of the weights of its edges

Matrix representation of graphs

Non-weighted graphs:

$$a_{i,j} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$



Weighted graphs:

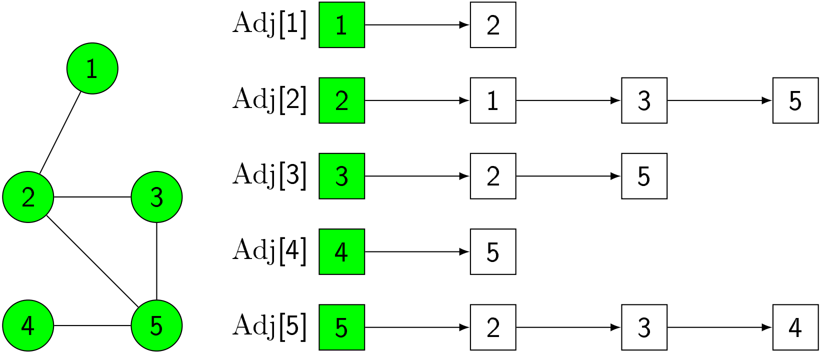
$$a_{i,j} = \begin{cases} w(v_i, w_j) & \text{if } (v_i, v_j) \in E, \\ 0 & \text{if } i = j, \\ \infty & \text{otherwise.} \end{cases}$$

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Suitable for dense graphs

List representation of graphs

- $G = (V, E)$ is represented by the list $\mathrm{Adj}[1 \ldots |V|]$ of lists
- For each $v \in V$, $\mathrm{Adj}[v]$ is a linked list of all vertices that has an edge in common with v



Suitable for sparse graphs

Part II

Minimum Spanning Tree (MST)

Minimum Spanning Tree (MST)

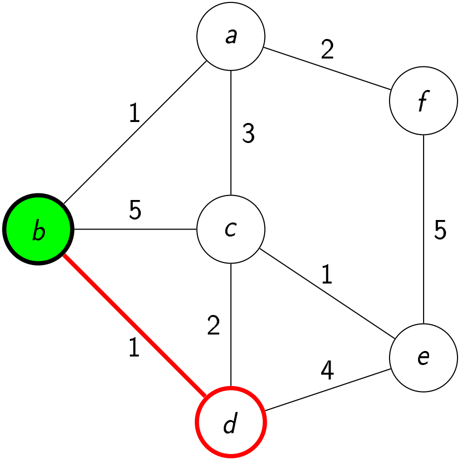
- A *spanning tree* of a graph $G = (V, E)$ is a sub-graph of G that is a tree and contains all vertices of G
- MST for a weighted graph is a spanning tree with minimum weight
- If $G = (V, E)$ is not connected, then it cannot have an MST but instead has a minimum spanning forest
- Assume that G is connected, otherwise we find connected components and find an MST Of each component

Prim's algorithm

- Greedy algorithm: Add one edge at a time to the MST
- Select a vertex at random
- Choose an edge with minimum weight between the selected set and the unselected set (break ties arbitrarily)
- Select the unselected vertex
- Repeat until a spanning tree has been created
- The constructed tree is an MST

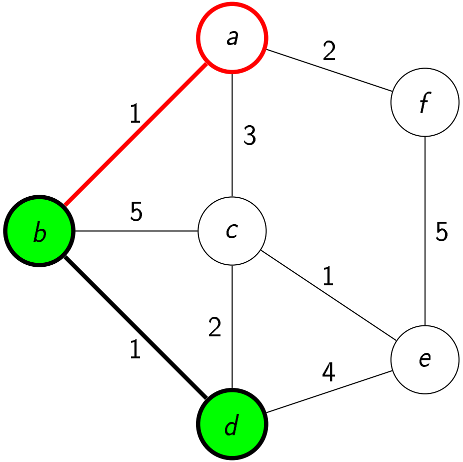
Prim's algorithm: Example (1/6)

	a	b	c	d	e	f
$d[]$	1	0	5	1	∞	∞



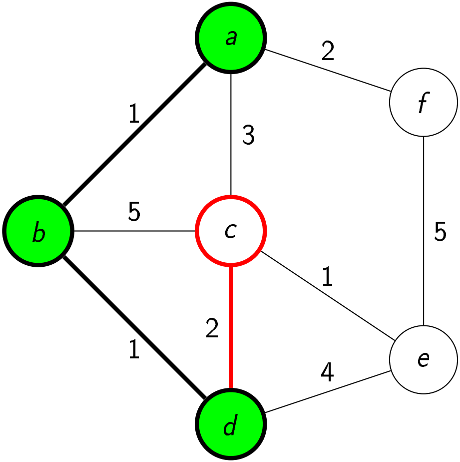
Prim's algorithm: Example (2/6)

	a	b	c	d	e	f
$d[]$	1	0	2	1	4	∞



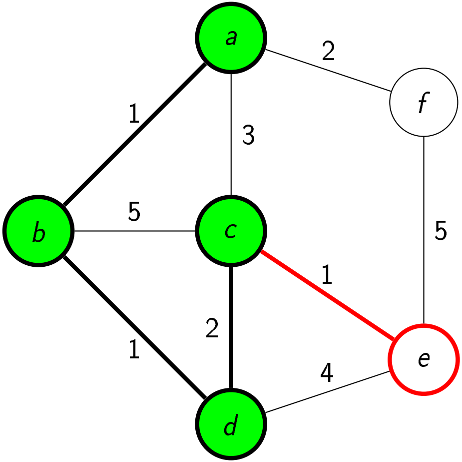
Prim's algorithm: Example (3/6)

	a	b	c	d	e	f
$d[]$	1	0	2	1	4	2



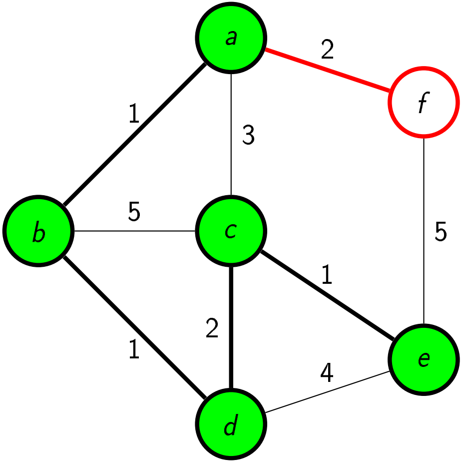
Prim's algorithm: Example (4/6)

	a	b	c	d	e	f
$d[]$	1	0	2	1	1	2



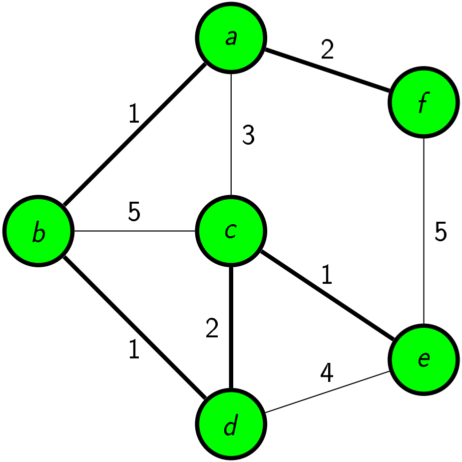
Prim's algorithm: Example (5/6)

	a	b	c	d	e	f
$d[]$	1	0	2	1	1	2



Prim's algorithm: Example (6/6)

	a	b	c	d	e	f
$d[]$	1	0	2	1	1	2



Prim's algorithm

PRIM(V, E, w, r)
$V_T := \{r\}$
$d[r] := 0$
for all $v \in (V - V_T)$ **do**
  $d[v] := w(r, v)$ if $(r, v) \in E$ else $d[v] := \infty$
end for
while $V_T \neq V$ **do**
  Find vertex $u \in (V - V_T)$ such that $d[u] = \min\{d[v] \mid v \in (V - V_T)\}$
  $V_T := V_T \cup \{u\}$
  for all $v \in (V - V_T)$ **do**
    $d[v] := \min\{d[v], w(u, v)\}$
  end for
end while

Parallelizing Prim's algorithm

- $d[v]$ is updated for all v
  - Cannot choose two vertices in parallel
  - Cannot parallelize outer `while` loop
  - Instead we parallelize the inner `for` loop
- Every process holds a block column of adjacency matrix A:

  $$A = \begin{bmatrix} A_1 & A_2 & \cdots & A_p \end{bmatrix}$$

  and corresponding part of vector d
- Process P_i holds vertex subset V_i
- Owner computes: Process P_i responsible for updating its part of d
- Find global minimum (line 8) with all-reduce
- Update d in parallel (line 10)

Analysis of the parallel Prim's algorithm

Per iteration:

- Computation (line 10): $\Theta(n/p)$
- All-reduce (line 8): $\Theta(\log_2 p)$

Total for n iterations:

- $T_P(n, p) = \Theta(n^2/p) + \Theta(n \log_2 p)$
- $T_S(n) = \Theta(n^2)$
- Iso-efficiency condition: $n = \Omega(p \log_2 p)$

Part III

Shortest paths (1-to-all)

Dijkstra's algorithm

Dijkstra's algorithm:

- Essentially identical to Prim's algorithm, except
  - Instead of $d[u]$ store $\ell[u]$, which is the total weight from r to u

Parallel Dijkstra's algorithm:

- Identical to the parallel Prim's algorithm (with the change above)
- Analysis identical

Part IV

Shortest paths (all-to-all)

Shortest paths (all-to-all)

- Goal is to find the weight of the shortest path between all pairs of vertices
- The result is a square matrix $D = (d_{i,j})$ where $d_{i,j}$ is the weight of the shortest path from v_i to v_j

Algorithm based on matrix multiplication

- Let $G = (V, E, w)$ be represented by the matrix A
- Let $d_{i,j}^{(k)}$ represent the weight of the shortest path from v_i to v_j that contains *a maximum of k* edges
- (Thus, $D_{i,j}^{(1)} = A$)
- Let v_m be a vertex in that path
- Then $d_{i,j}^{(k)} = \min_m \{d_{i,m}^{(k-1)} + w(v_m, v_j)\}$

Matrix multiplication-based algorithm (continued)

- $D^{(k)}$ computed from $D^{(k-1)}$ using modified matrix multiplication:
  $$c_{i,j} := \min_k a_{i,k} + b_{k,j}$$
  (Find k that minimizes $a_{i,k} + b_{k,j}$)
- $D^{(k)} = \underbrace{AA \cdots A}_{k \text{ factors}}$
- But we *only need* $D^{(n-1)}$
- Compute $D^{(n-1)}$ using repeated squaring:
  $$D^{(1)} \mapsto D^{(2)} \mapsto D^{(4)} \mapsto D^{(8)} \mapsto \cdots \mapsto D^{(n-1)}$$
- Complexity for matrix multiplication: $\Theta(n^3)$
- Number of steps: $\Theta(\log_2 n)$
- Total complexity: $T_P(n, p) = \Theta(n^3 \log_2 n)$

Dijkstra's algorithm applied to all-to-all shortest paths

Source-partitioned approach:

- Distribute the sources
- Run *sequential* 1-vertex Dijkstra on all processors in parallel
- Complexity: $\Theta(n^2) \cdot \Theta(n/p) = \Theta(n^3/p)$
- Perfectly parallel
- Degree of concurrency limited to $p = \mathcal{O}(n)$
- Each processor must have access to the entire graph

Source-parallel approach:

- Partition processors into groups (e.g., of size $\sqrt{p}$)
- Distributed sources over the groups
- Run *parallel* 1-vertex Dijkstra on all processor *groups* in parallel
- Complexity: $\left[\Theta(n^2/\sqrt{p}) + \Theta(n \log_2 \sqrt{p})\right] n/\sqrt{p} = \Theta(n^3/p) + \Theta((n^2/\sqrt{p}) \log_2 p)$
- Degree of concurrency $p = \mathcal{O}(n^2)$ (much better)
- Each processor needs access only to a sub-graph

Floyd's algorithm

- Given $G = (V, E, w)$
- Let $V_k := \{v_1, \ldots, v_k\}$ (first k vertices of G)
- For any pair $v_i, v_j \in V$, consider all paths whose intermediate vertices belong to the subset V_k
- Let $p_{i,j}^{(k)}$ be the shortest such path and let $d_{i,j}^{(k)}$ be the corresponding weight
- **If the vertex v_k is *not* in the path**, then $p_{i,j}^{(k)} = p_{i,j}^{(k-1)}$
- **If the vertex v_k *is* in the path**, then it can be split into two paths: One from v_i to v_k and one from v_k to v_j where both paths uses vertices only from V_{k-1}
- In that case, the weight of the path is $d_{i,j}^{(k)} := d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$

Floyd's algorithm

FLOYD(A)
$D^{(0)} := A$
for $k = 1$ to n **do**
    for $i = 1$ to n **do**
        for $j = 1$ to n **do**
            $d_{i,j}^{(k)} := \min\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\}$
        end for
    end for
end for

- Similar in structure to matrix multiplication
- Parallelized using SUMMA-style algorithm

Part V

Transitive closure

Transitive closure

- If $G = (V, E)$ is a graph, then its *transitive closure* is the graph $G^* = (V, E^*)$, where

$$E^* := \{(v_i, v_j) \mid \text{exists path from } v_i \text{ to } v_j \text{ in } G\}$$

- Computes the connectivity matrix A^* such that $a_{i,j} = 1$ if $i = j$ or a path from v_i to v_j exists in G and $a_{i,j} = \infty$ otherwise

Method 1:

- Set the weights in G to 1 and compute all-pairs shortest paths followed by re-interpreting the output

Method 2:

- Modify Floyd's algorithm by replacing min with *logical or* and $+$ with *logical and*:

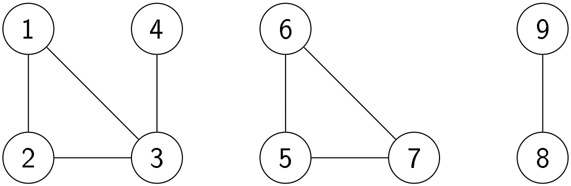
$$d_{i,j}^{(k)} := d_{i,j}^{(k-1)} \quad \texttt{or} \quad (d_{i,k}^{(k-1)} \quad \texttt{and} \quad d_{k,j}^{(k-1)})$$

Part VI

Connected components

Connected components

- Partition V into maximal disjoint subsets $C_1, C_2, \ldots, C_r$ such that $V = C_1 \cup C_2 \cup \cdots \cup C_r$ and $u, v \in C_i$ if and only if u is reachable from v and vice versa
- The graph below has three connected components



Depth-first search based algorithm

- Perform depth-first traversal of the graph to generate a spanning forest
- Each tree in the forest defines a connected component

Parallel formulation:

- Give sub-graph $G_i = (V, E_i)$ to process P_i
- Perform sequential algorithm on each sub-graph in parallel
- Merge forests pair-wise using $\log_2 p$ steps

Forest merging

- `find`(x): Tree to which x belongs
- `union`(u, v): Merge trees to which u and v belong
- Merge forest A into forest B:
  - Send A to processor holding B
  - For each edge (u, v) (there are at most $n - 1$) in A:
    - `find`(u) in B
    - `find`(v) in B
    - Same tree? Do nothing
    - Different trees? `union`(u, v) in B
  - Discard A, continue with B
- Using appropriate set data structure and algorithms, `find`(x) and `union`(u, v) have expected constant time complexity
- Complexity (1D block partitioning):

$$T_P(n, p) = \Theta(n^2/p) + \Theta(n \log_2 p)$$

Part VII

Johnson's algorithm

Johnson's algorithm (1-to-all shortest paths)

- Dijkstra's algorithm
  - Find unprocessed u such that

$$d[u] = \min\{d[v] \mid v \text{ unprocessed}\}$$

  - Update for all unprocessed vertices v:

$$d[v] := \min\{d[v], d[u] + w(u, v)\}$$

- For *sparse graph*, store unprocessed vertices in a *priority queue* based on $d[v]$ (smallest first)
- Take minimum weight vertex from the priority queue
- Update adjacent vertices

Johnson's algorithm

```
JOHNSON(V, E, r)
Q := V
for all v ∈ Q do
    d[v] := ∞
end for
d[r] := 0
while Q ≠ ∅ do
    u := ExtractMin(Q)
    for each v adjacent to u do
        if v ∈ Q and d[u] + w(u, v) < d[v] then
            d[v] := d[u] + w(u, v)
        end if
    end for
end while
```

Parallel Johnson's (centralized queue)

- Maintain Q at a centralized location
- Processors compute new values and request updates of Q
- Major bottleneck
- No asymptotic speedup, since $\mathcal{O}(|E|)$ updates that are serialized and each take time $\mathcal{O}(\log_2 n)$ leading to the same complexity as the sequential formulation
- Moreover, only $|E|/|V|$ vertices can be updated in parallel in each iteration, and this number is small since the graph is assumed sparse

Parallel Johnson's (distributed queue)

Distributed queue:

- Manage Q using distributed algorithm
- Requires a machine with very low latency to be practical
- Even if the update complexity is reduced from $\mathcal{O}(\log_2 n)$ to $\mathcal{O}(1)$, we can expect no more than a $\mathcal{O}(\log_2 n)$ speedup since the updates are applied sequentially

Safe vertices:

- Let u be a vertex with minimal $d[u]$
- All vertices v with $d[v] = d[u]$ can be processed in parallel
- If we know that the minimum edge weight is m, then we can relax this to all vertices v with

$$d[v] \leq d[u] + m$$

Parallel Johnson's (unsafe vertices)

- Process also unsafe vertices in parallel
- Leads to an algorithm that is no longer equivalent to the sequential algorithm
- Process p vertices at the top of Q in parallel
- Each process maintains its own priority queue
- The distances might no longer correspond to the shortest paths
- Detect instances of wrongly computed distances and re-process the corresponding vertices

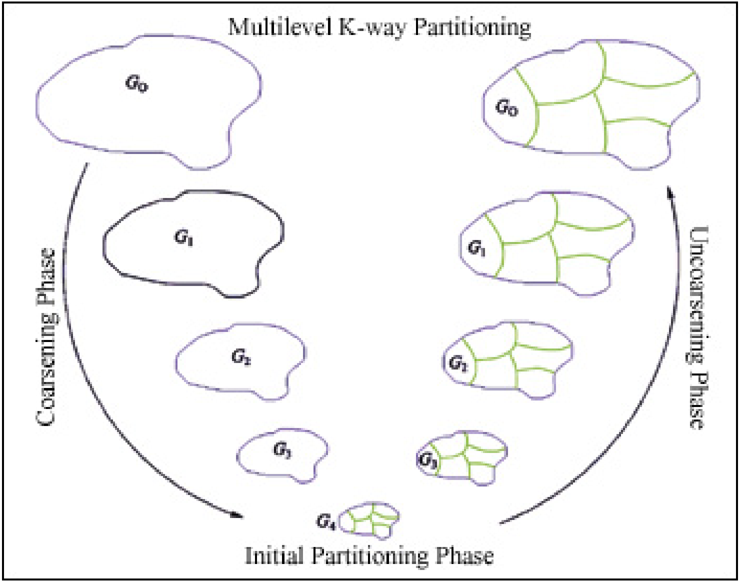
Part VIII

Weighted matchings

Weighted matchings

- A *matching* $M(G)$ of a graph $G = (V, E)$ is any sub-graph of G where each vertex is incident to at most one edge
- Let $w(e)$ be the weight of an edge e
- We define the weight $w(G)$ of a graph G as the sum of all edge weights
- A *maximum weighted matching* $M^*(G)$ of G is a matching whose weight $w(M^*(G))$ is maximum among all matchings of G

Matchings and parallel graph partitioning



Sequential greedy weighted matching algorithm

```
MATCHING(V, E)
M(G) := ∅
while E ≠ ∅ do
    Pick locally heaviest edge e from E
    Add e to M(G)
    Remove e = {u, v} and all edges incident to u and v from E
end while
```

Approximates a maximal matching within a factor 2

Parallel greedy weighted matching algorithm

For each vertex v in parallel:

PMATCHING(V, E)
$R := \emptyset$
Initialize N to the neighborhood of v (all vertices adjacent to v)
Let the *candidate* c be the vertex connected to v by the *locally heaviest edge* in N
if $c \neq \perp$ **then**
    Send `req` to c
end if
while $N \neq \emptyset$ **do**
    Receive message m from vertex u
    If $m = \texttt{req}$, then $R := R \cup \{u\}$
    If $m = \texttt{drop}$, then $N := N - \{u\}$ and update c if $u = c$ and if $c \neq \perp$ send `req` to c
    If $c \neq \perp$ and $c \in R$, send `drop` to all vertices in N except c
end while