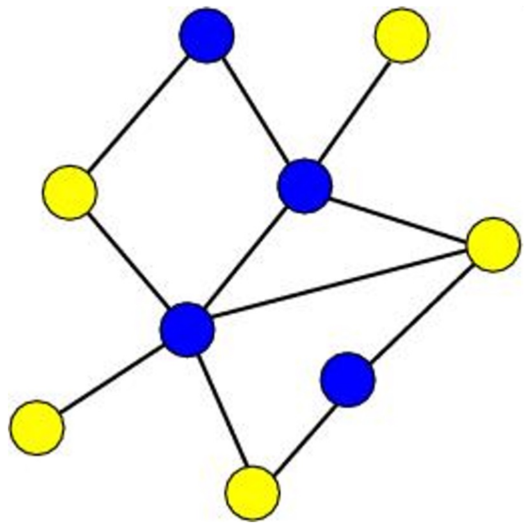
Graph Algorithms

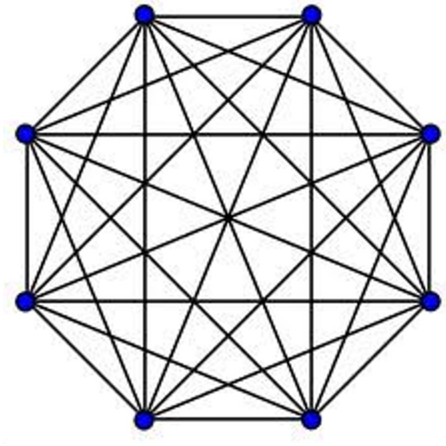
Today: Traversals

Representation



Sparse graphs

- Represented using neighbor lists
- Lacks structure, and has less work
- Harder to get load balance in parallel algorithms



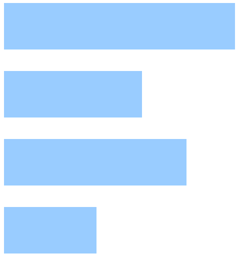
Dense graphs

- Represented using an adjacency matrix
- More work → Easier to parallelize
- Algorithms often resemble matrix algorithms

Sparse Graphs

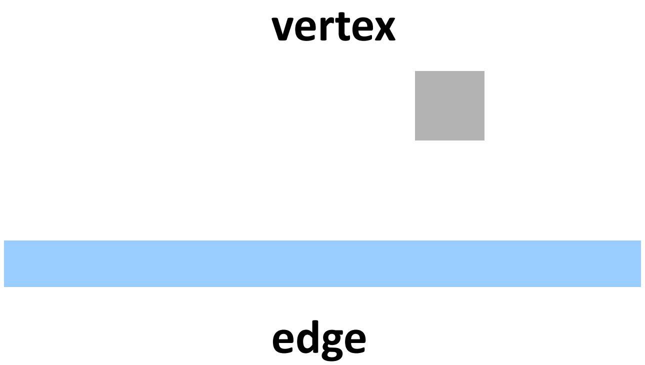
Individual lists

- One array for vertices
- Individual lists of neighbors
- Neighbor lists might not be consecutive in memory!
- Relatively easy to add or remove edges and vertices



Compressed lists

- Edges are consecutive
- Use extra dummy pointer at the end
- Points to last element **+ 1**
- Use integer values in both arrays
- Well suited for static graphs



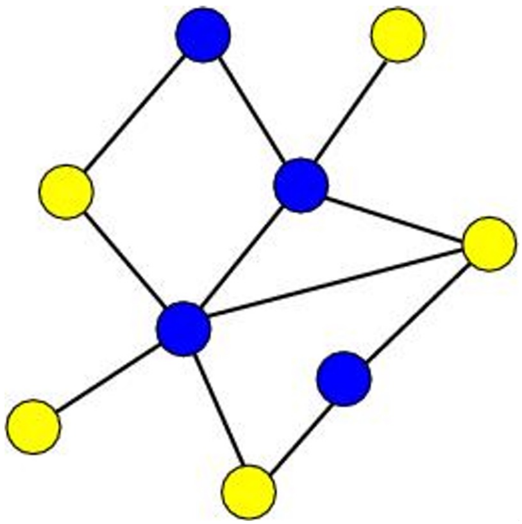
Traverse graph (**v** = vertex index, **e** = edge index):

for **v = 1 to n**
  for **e = vertex[v] to vertex[v+1]-1**
    process **edge[e]**

Parallel Graph Algorithms
(shared memory)

Need operations that can be performed in parallel

- Iterate through neighbor list of each vertex
- Work on several vertices simultaneously



Traverse graph (**v** = vertex index, **e** = edge index):

```
#pragma parallel for private(e)
for v = 1 to n
#pragma parallel for
   for e = vertex[v] to vertex[v+1]-1
      process edge[e]
```

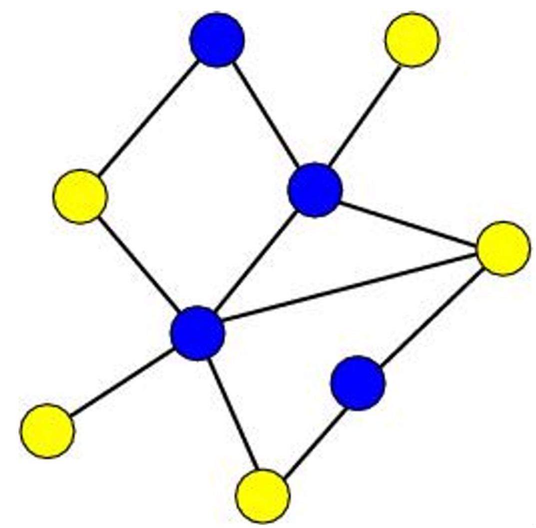
Which one to chose depends on the structure of the current problem.

Examples:

- Find heaviest incident neighbor of every vertex
- Compute independent set of vertices

Parallel Graph Algorithms
(shared memory)

Find heaviest incident neighbor of every vertex



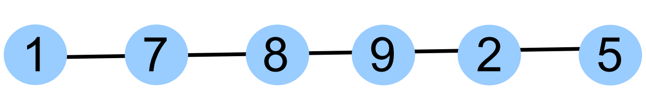
Traverse graph (**v** = vertex index, **e** = edge index):

```
#pragma parallel for private(e,heaviest)
for v = 1 to n
  heaviest = weight[vertex[v]]
  for e = vertex[v]+1 to vertex[v+1]-1
    if weight[edge[e]] > heaviest
      heaviest = weight[edge[e]]
```

Compute independent set (IS) of vertices:

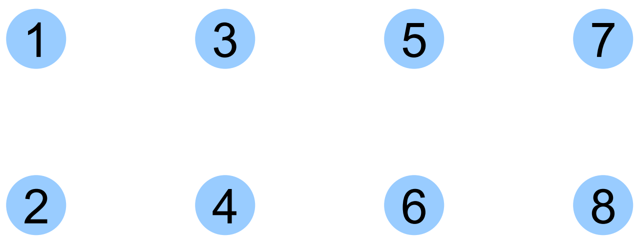
Luby´s algorithm:

- Generate random number for each vertex
- If a vertex is heavier than all its unmarked neighbors, put it in the IS, otherwise mark all neighbors as out of IS
- Test each vertex until either in IS or “out of IS”

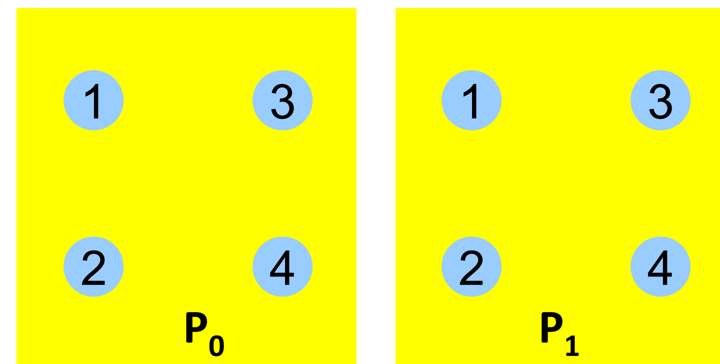


IS = {5,7,9}

Parallel Graph Structures
(for distributed memory)



Global graph



Partitioned graph

Local vertex #3

- Needs global ID
- List of local neighbors
- List of off-processor neighbors

Vertex 3 has off-processor neighbor 5

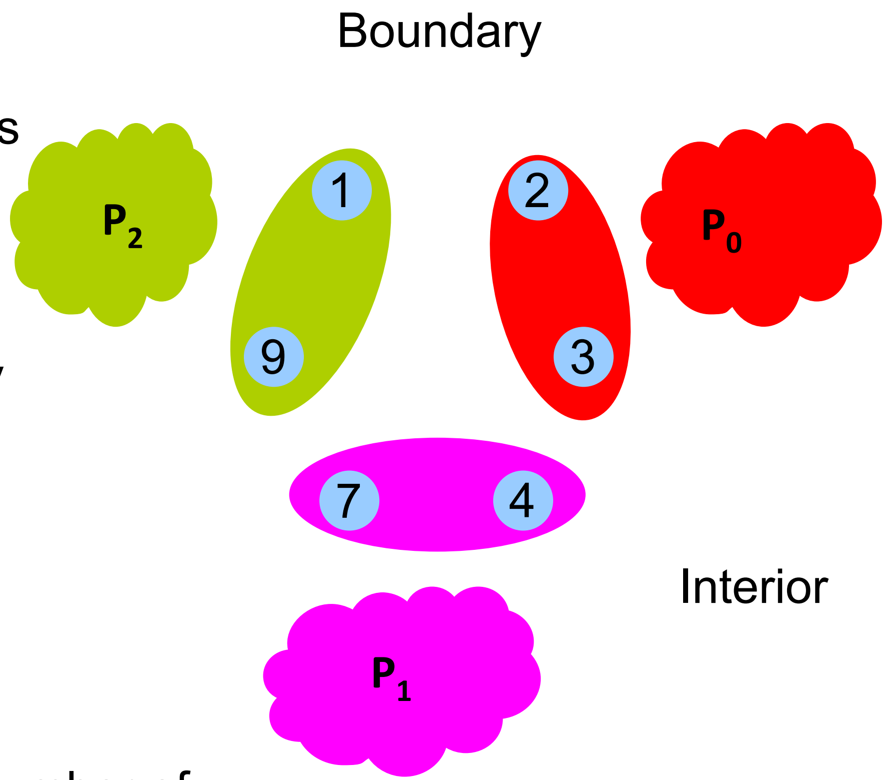
- Needs processor (=1) on which 5 is stored
- Needs local ID (=1) on remote processor

Graph Partitioning and Parallel Graph Algorithms

Graph algorithms may work on non-adjacent vertex sets simultaneously!

Parallel strategy:

1. Partition the graph into **p** parts and allocate one part to each processor
2. Partition each part into interior and boundary vertices
3. Solve problem in parallel on interior vertices using sequential algorithm
4. Use parallel algorithm to solve problem on boundary vertices



Examples of problems:

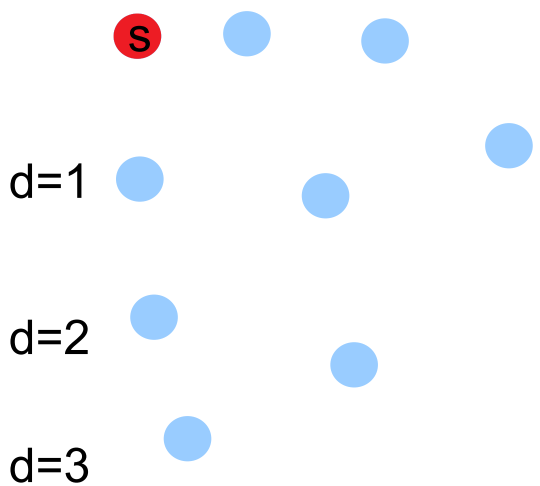
- Connected components
- Graph coloring ('few' but not necessarily minimum number of colors)
- Does not work for e.g. the shortest path problem

Graph Traversal: BFS

Idea: Visit each node of G by distance from starting node s

BFS(G,s)

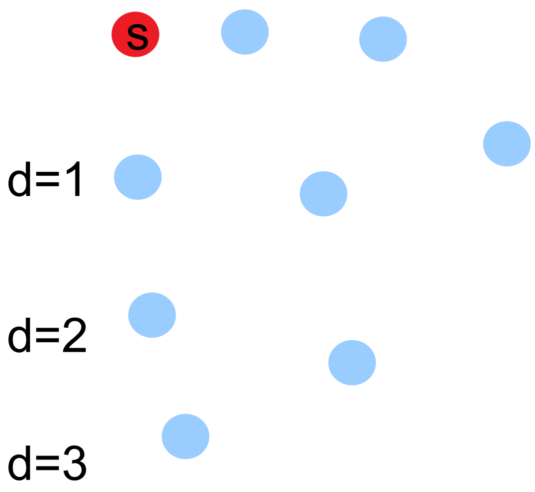
```
For each v in G:
   distance[v] = -1
   parent[v] = undefined

Q = Ø    // Queue
distance[s] = 0
parent[s] = s
Q.add(s)

While Q not empty
    v = Q.pop()
    For each w in N(v):
        If distance[w] == -1
          distance[w] = distance[v] + 1
          parent[w] = v
          Q.add[w]
```



Parallel BFS

Observations:

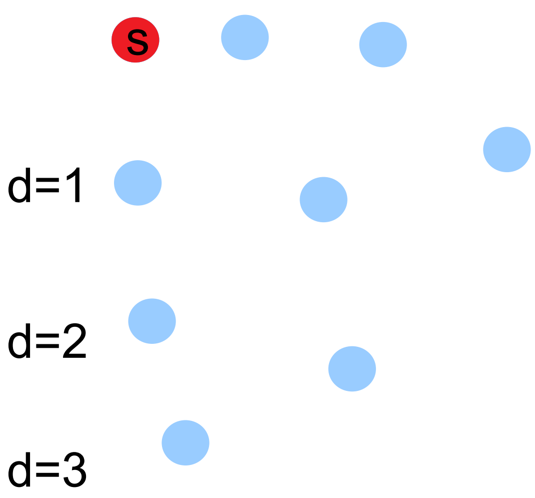
- Must visit all vertices at distance *i* before visiting any vertex at distance *i+1*
- Can explore from all vertices at distance *i* in parallel
- Does not matter which vertex in previous layer is set as parent
- Hard to parallelize work over Q: Can use two lists, one for layer *i* and one for layer *i+1*



Graph Traversal: BFS

Code using two arrays S and T

```
BFS(G,s)
For each v in G:
   distance[v] = -1
   parent[v] = undefined

distance[s] = 0
parent[s] = s
S[0] = s
num_r = 1
num_w = 0

While num_r > 0
    for(i=0;i<num_r;i++)
       v = S[i]
       For each w in N(v):
          If distance[w] == -1
            distance[w] = distance[v] + 1
            parent[w] = v
            T[num_w++] = w
    Swap S and T
    num_r = num_w; num_w = 0
```



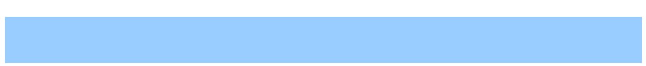
Parallel BFS

Observations:

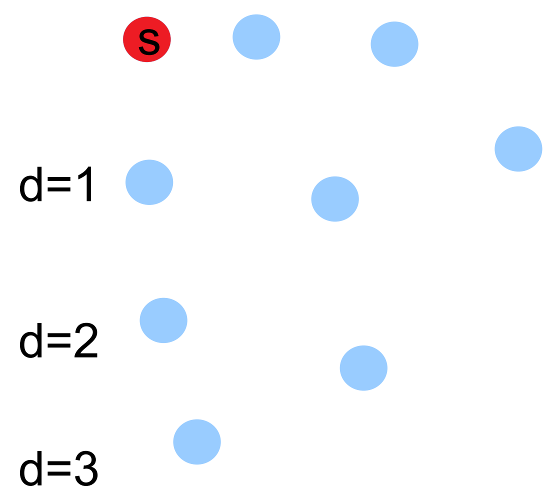
- Reading from one list S and writing to another T
- Anyone can write to distance and parent arrays

Issues:

- Distribute vertices in S
- Race conditions when updating distance and parent arrays
- How to organize writing to T?

S 

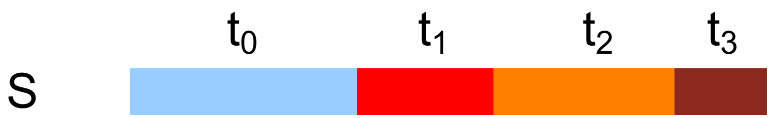
T



Parallel BFS

Possible solutions:

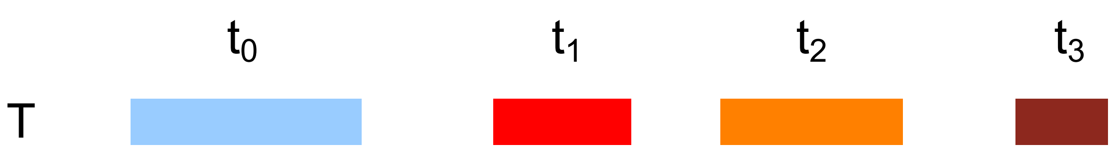
- Distribute vertices in S using “parallel for”, try different load balancing strategies



Note, work is proportional to vertex degree

- One common T list: Must protect counter for where to put next discovered vertex

- Individual T lists for each thread:



Combine lists or continue in a distributed fashion?

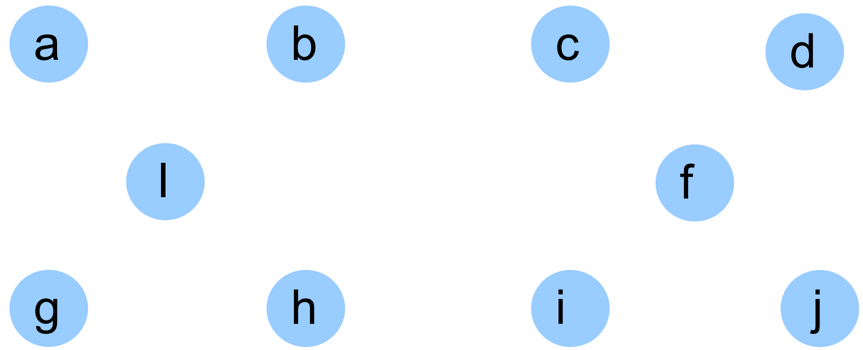
- Can a vertex be discovered and entered in T more than once?

T v v

Guard against (how?) or ignore?

Traversal of DAGs

DAG: Directed Acyclic Graph



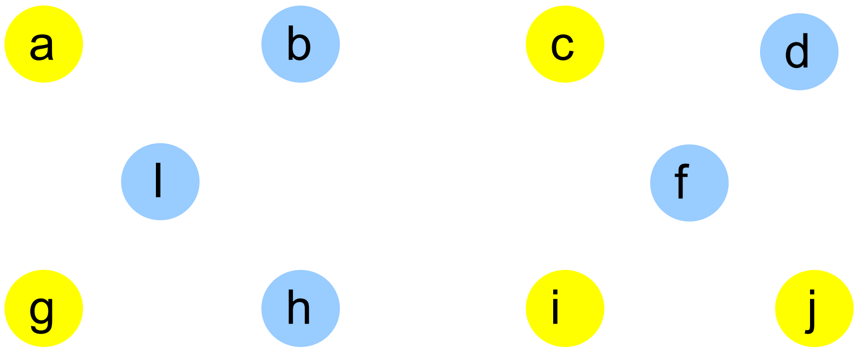
Topological ordering: Linear ordering of vertices where every edge goes from “left” to “right”

a, b, g, l, c, h, i, f, j, d

Outline of algorithm:
R = Queue with all vertices of indegree 0
While R is not empty
  v = pop(R)
  Add v to end of TopOrdering
  For each edge (v,w):
    Remove (v,w) from G
    If w has indegree == 0
    R = R U {w}

Can be parallelized similarly to BFS

Maximal Independent Set on DAGs



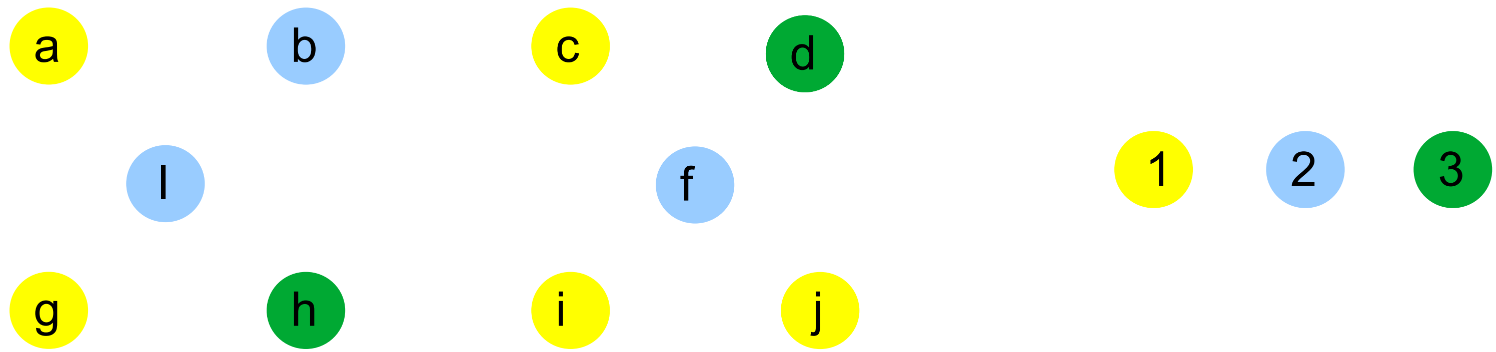
Independent set: Set of vertices such that two neighbors are not in the set

Maximal: A set where one cannot add a new vertex and still have an independent set

Outline of algorithm:
For each vertex v:
    IS(v) = true

Process each edge (v,w) according to TopOrd:
    IS(w) = (IS(w) and (not IS(v)))

Graph Coloring on DAGs



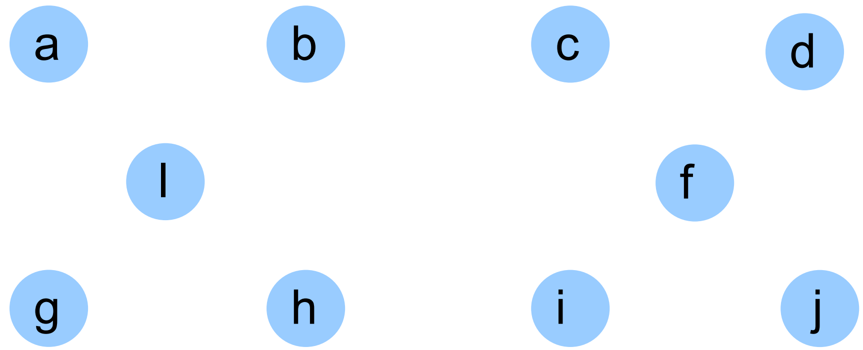
Graph coloring: Assignment of colors to vertices such that two incident vertices receives different colors

Outline of algorithm:
For each vertex v:
Maintain list of forbidden colors for each vertex

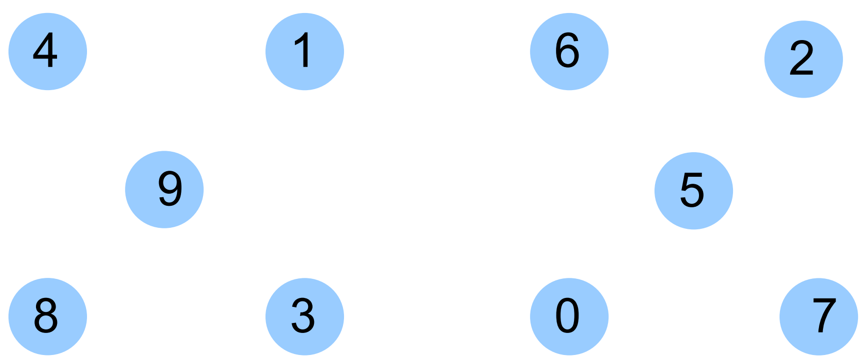
Process each vertex v according to TopOrd:
Assign lowest available color to v

Process edges (v,w):
Add color(v) to list of forbidden colors for w

Turning Graphs into DAGs



Assign random number to each vertex
Direct edges from lower numbered vertex to higher numbered one



Every path follows increasing numbered vertices:
 Cannot contain a cycle!