Adjacency matrix - Operations
- addEdge(i,j): matrix[i][j] = 1
- removeEdge(i,j): matrix[i][j] = 0
- Not very good for inserting/removing vertices: requires shifting elements of matrix.
- Requires space $O(n^2)$

Lists vs Matrices
- Adjacency lists are better if:
  - You frequently need to add/remove vertices
  - The graph has few edges
  - Need to traverse the graph
- Adjacency matrices are better if:
  - You frequently need to add/remove edges, but NOT vertices
  - Check for the presence/absence of an edge between i,j
  - Matrix is small enough to fit in memory

Graph Traversal
Depth-First Search
Breadth-First Search

Graph traversal - Idea
- Problem:
  - you visit each node in a graph, but all you have to start with is:
    - One vertex A
    - A method getNeighbors(vertex v) that returns the set of vertices adjacent to v

Graph traversal - Motivations
- Applications
  - Exploration of graph not known in advance, or too big to be stored:
    - Web crawling
    - Exploration of a maze
  - Graph may be computed as you go. Example: game strategy:
    - Vertices = set of all configurations of a Rubik’s cube
    - Edges connect pairs of configuration that are one rotation away.

Depth-First Search
- Idea: Go Deep!
  - Intuition: Adventurous web browsing: always click the first unvisited link available. Click “back” when you hit a deadend.
  - Start at some vertex v
  - Let w be the first neighbor of v that is not yet visited. Move to w.
  - If no such unvisited neighbor exists, move back to the vertex that lead to v
11/10/04 10:20 Depth-First Search 7
Example
Ex.

11/10/04 10:20 Depth-First Search 8
Example (cont.)
Ex.

11/10/04 10:20 Depth-First Search 9
DFS Algorithm
Algorithm DFS(G, v)
Input: graph G with no parallel edges and a start vertex v of G
Output: Visits each vertex once (as long as G is connected)
print v // or do some kind of processing on v
v.setLabel(VISITED)
for all u ∈ v.getNeighbors()
if (u.getLabel() != VISITED) then DFS(G, u)

11/10/04 10:20 Depth-First Search 10
DFS and Maze Traversal
- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)

11/10/04 10:20 Depth-First Search 11
DFS and Rubik’s cube
- Rubik’s cube game can be represented as a graph:
  - Vertices: Set of all possible configurations of the cube
  - Edges: Connect configurations that are just one rotation away from each other
- Given a starting configuration S, find a path to the “perfect” configuration P.
- Depth-first search could in principle be used:
  - start at S and making rotations until P is reached, avoiding configurations already visited
- Problem: The graph is huge: 43,252,003,274,489,856,000 vertices

11/10/04 10:20 Depth-First Search 12
Running time of DFS
- DFS(G, v) is called once for every vertex v (if G is connected)
- When visiting node v, the number of iterations of the for loop is deg(v).
- Conclusion: The total number of iterations of all for loops is: Σ deg(v) = ?
- Thus, the total running time is O(|E|)
Applications of variants of DFS

- DFS can be used to:
  - Determine if a graph is connected
  - Determine if a graph contains cycles
  - Solve games single-player games like Rubik’s cube

Iterative DFS

- Use a stack to remember your path so far

**Algorithm iterativeDFS(G, v)**

- **Input** graph $G$ with no parallel edges and a start vertex $v$ of $G$
- **Output** Visits each vertex once (as long as $G$ is connected)

1. $s \leftarrow$ new Stack()
2. $v$ setLabel(VISITED)
3. $s.push(v)$
4. while (! $s.empty()$) do
   - $w \leftarrow s.pop()$
   - print $w$
   - for all $u \in w.getNeighbors()$ do
     - if ($u.getLabel() \neq VISITED$) then
       - $u$ setLabel(VISITED)
       - $s.push(u)$

Breadth-First Search

- **Idea:**
  - Explore graph layers by layers
  - Start at some vertex $v$
  - Then explore all the neighbors of $v$
  - Then explore all the unvisited neighbors of the neighbors of $v$
  - Then explore all the unvisited neighbors of the neighbors of the neighbors of $v$
  - until no more unvisited vertices remain

Example

- $L_0$
- $L_1$
- $L_2$
- $L_3$
- $L_4$
- $L_5$

Example (cont.)

- $L_0$
- $L_1$
- $L_2$
- $L_3$
- $L_4$
- $L_5$

Example (cont.)

- $L_0$
- $L_1$
- $L_2$
- $L_3$
- $L_4$
- $L_5$
Iterative BFS
• Idea: use a queue to remember the set of vertices on the frontier

```
Algorithm iterativeBFS(G, v)
    Input graph G with no parallel edges and a start vertex v of G.
    Output Visits each vertex once (as long as G is connected).
    q ← new Queue()
    v.setLabel(VISITED)
    q.enqueue(v)
    while (! q.empty()) do
        w ← q.dequeue() // or do some kind of processing on w
        print w
        for all u ∈ w.getNeighbors() do
            if (u.getLabel() != VISITED) then
                u.setLabel(VISITED)
                q.enqueue(u)
```

Notice: Code is identical to DFS, but with a queue instead of a stack

Running time and applications
• Running time of BFS: Same as DFS, O(|E|)
• BFS can be used to:
  - Find a shortest path between two vertices
  - Rubik's cube's fastest solution
  - Determine if a graph is connected
  - Determine if a graph contains cycles
  - Get out of an infinite maze...