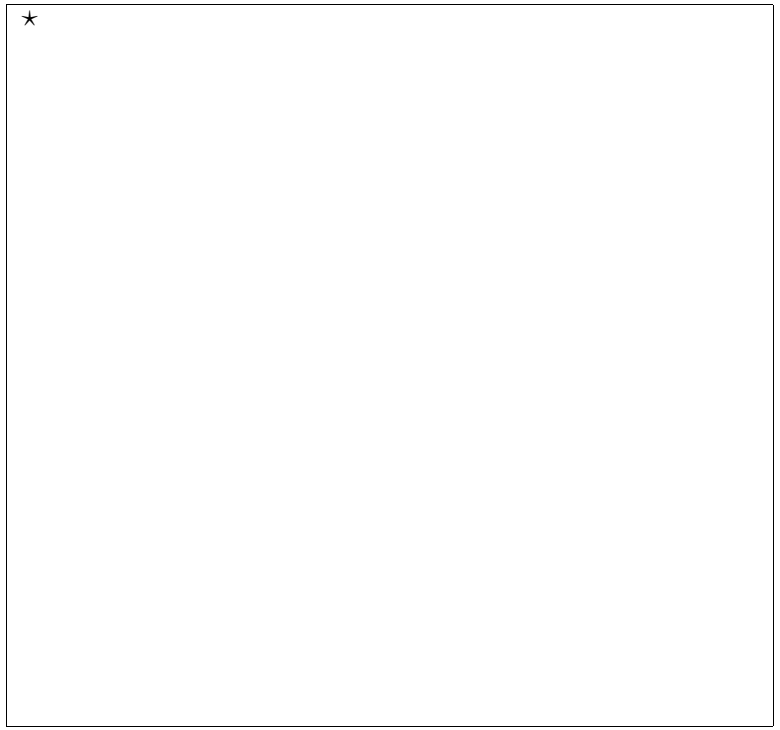
.

Introduction to Graphs

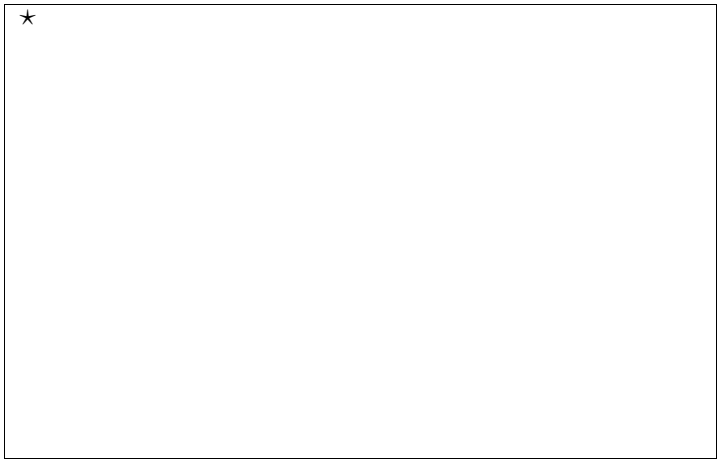
An **undirected graph** is a tuple $G = (V, E)$ where V is the set of **nodes** or **vertices**, and E is the set of **edges**, each of which is of the form $\{i, j\}$, where i, j are nodes in the set V. We use the following notation: $|V| = n$ and $|E| = m$.



Graphs are very useful in modeling computational problems in many areas. In some applications, the edges of the graph are weighted. In others, the edges of the graph are directed, represented graphically by drawing edges as arrows pointing in the direction of the edge.

Here are just a few examples of situations that can be modeled by graphs.

- **Computer networks**. Here, nodes represent computers or gateways, and edges represent links between nodes. Network reliability problems involve testing if the network is connected even when certain edges fail.
- **Probabilistic graphical models**. These are large, sparse reasoning systems driving research in AI, speech and computer vision.



- **Scheduling**. We already saw an example in which nodes represent courses that students want to take, and an edge between two nodes indicates that those two courses can't be scheduled at the same time. A coloring of the nodes represents a schedule of the courses, where each color represents a distinct time slot. A natural problem is to assign a color to each node in such a way that no two adjacent nodes have the same color; the goal is to minimize the number of colors used. Another example is that of scheduling jobs on a parallel computer so as to minimize the total time until all jobs are completed.

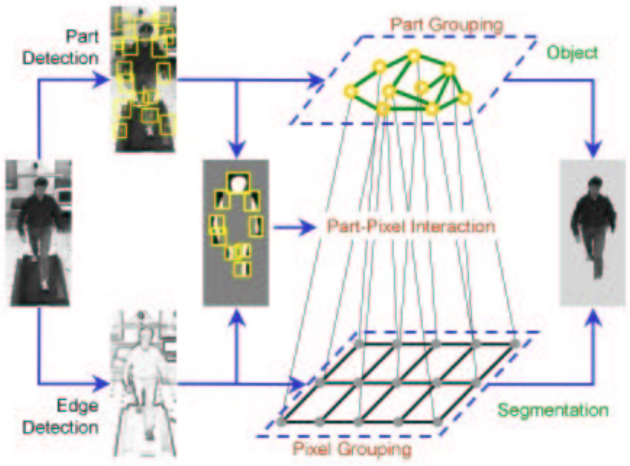
- **VLSI designs** are typically represented as graphs. A basic "layout" problem is to arrange the nodes of the graph in a limited area so as to ensure that no edges cross.

- In many applications, the nodes of the graph represent

places and the edge weights represent the **distance between the places** or the cost of getting from one place to another.

- In **DNA sequencing** software, short DNA strands may be represented by nodes and the degree of similarity of two strands may be represented by the weight of the edge connecting the strands.

- In **computer vision**, undirected graphs are used to model image constraints and segmentation.



Central computational problems that arise in manipulating graphs for such applications include:

- **Reachability**: is node B of the graph reachable from node A? Is the graph connected? Is it biconnected, i.e. are there at least two disjoint paths between any pair of nodes? Graph traversal algorithms such as depth first search and breadth first search can be used to solve such problems.
- **Shortest path**: in a weighted graph, what is the shortest path from node A to node B? To try out software that solves the shortest path problem, go to a mapping website `www.mapquest.com` or `www.mapblast.com`.
- **Min cost spanning tree**: We will define this problem precisely later, but for now think of a min cost spanning tree as providing the cheapest possible way to broadcast information across all edges of a weighted graph.

- **Matching**: the goal is to "match" the nodes of the graph into pairs, where a pair of nodes can be matched if they are connected by an edge. Each node can be matched to at most one other node.
- **Flow**: Suppose that the weights on the edges of a directed graph represent capacities. One version of the flow problem is to determine what is the maximum "flow" that can be transported from a given source node to a given sink node via the edges, where the amount of flow across a given edge is bounded by its capacity.
- **Coloring**: we saw the coloring problem in practice homework 1 (tutorial 2).
- **Traveling Salesman**: This is a classical problem on weighted graphs. What is the cheapest way to start at some node, visit all other nodes exactly once, and then return to the start?

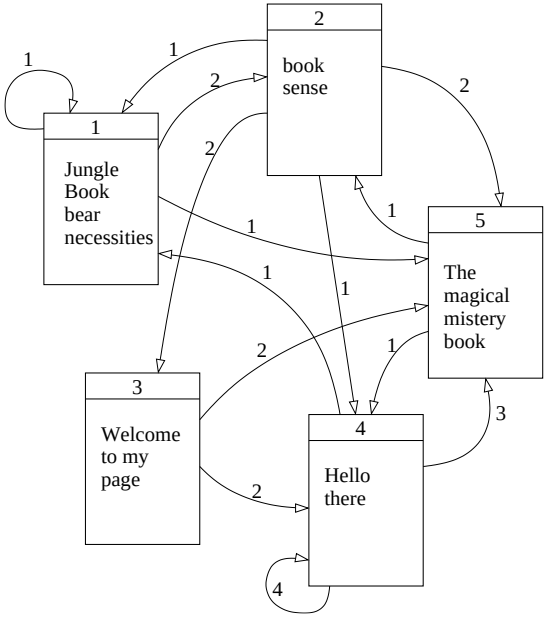
In the next lectures, we'll see beautiful algorithms for the min cost spanning tree and shortest path problems. Efficient algorithms are also known for matching and flow problems (but we will not cover them). Unfortunately, no-one knows of efficient algorithms for the coloring and traveling salesman problems. At the end of the course we'll return to these.

Graph Representation

Before discussing algorithms, we need to discuss data structures for representing graphs. There are two commonly-used methods:

1. *Adjacency Matrix Representation:* A graph is represented by a $n \times n$ matrix A, where $A(i, j) = 1$ iff there is an edge between the ith and jth nodes, and is 0 otherwise.

Given the following web graph:



⋆ Construct the graph transition matrix.

We can compute Google's rank vector using the PageRank algorithm:

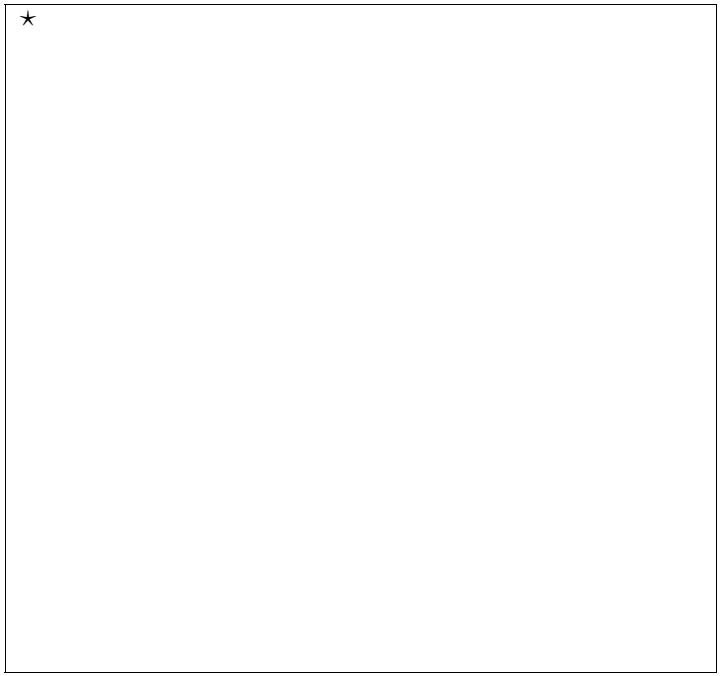
```
G = [1/4 2/4 0 0 1/4
     1/6 0 2/6 1/6 2/6
     0 0 0 2/4 2/4
     1/8 0 0 4/8 3/8
     0 1/2 0 1/2 0];

x=[.5 .3 .1 .1 0];    % Arbitrary initial vector.
R = x * ( G^(1000) )  % Rank vector after 1000 iterations.
```

The final rank is:

$$R = [\ 0.11 \quad 0.18 \quad 0.06 \quad 0.39 \quad 0.26\ ]$$

⋆ The answer to the query " *book*" is:

2. *Adjacency List Representation:* For each node, maintain a list of nodes to which it is connected by an edge.



In this representation it is a little harder to get information about an edge. For example, if you want to know about edge (3,4), you know that you start with 3. Then you have to traverse the list to determine if the

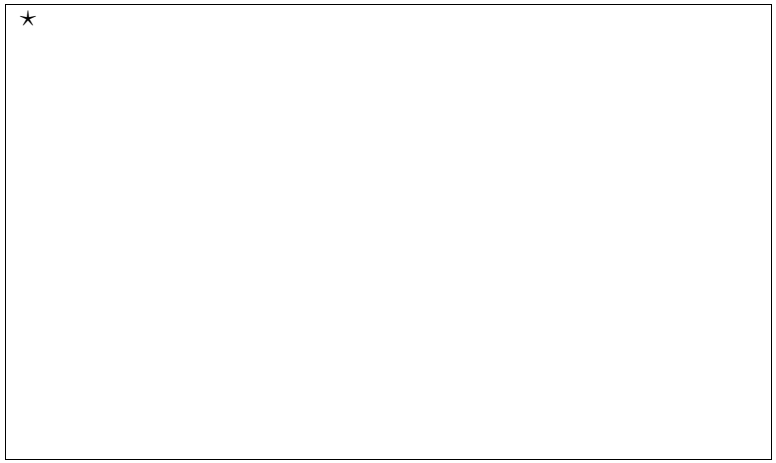
edge between 3 and 4 exits. A benefit occurs in saving space. The space used is proportional to the number of nodes plus the number of edges. Because we have array of size n and then we have two records for each edge that's in the graph, assuming space per record in lists is constant. Hence, the space used is $O(m+n)$. In an application, we may expand node and edge records to include additional information such as names(names of cities), weighted edges (ie distances between cities), etc.

Graph Traversal Algorithms

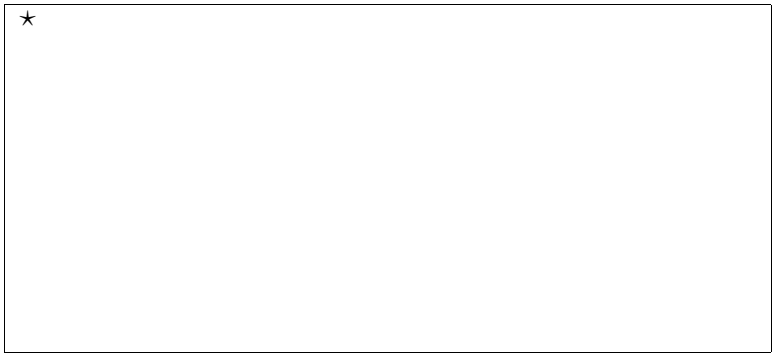
Graph traversal algorithms are useful in answering reachability questions such as: Is a graph connected? Is the node i reachable from node j? There are two standard methods of searching through a graph.

Depth First Search: Visit all of the nodes in a connected graph by expanding the search from the currently visited node. Initially, we set visited(i) = 0 for all nodes i. After visiting each node, it is marked as visited. The algorithm is initially called from any initial node of the graph.

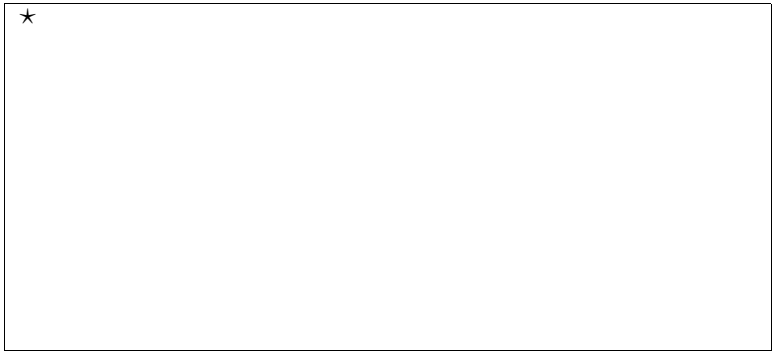
```
DFS(s)
  Visited (s) = 1
  for each w adjacent to s do
     if Visited (w) = 0 then DFS(w)
```



Other possible traversal orderings of the nodes would also be consistent with this algorithm. What is the running time of this algorithm?

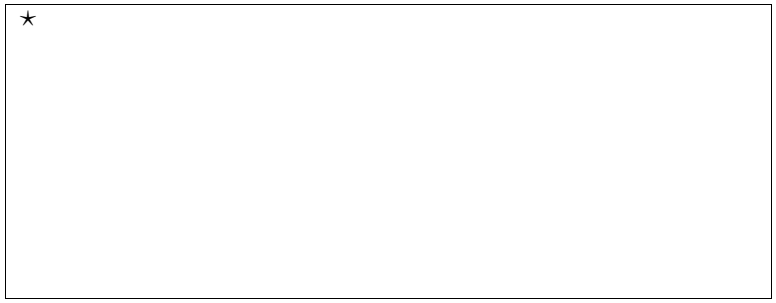


Breadth First Search: Always visit an unvisited node that is closest to the starting vertex. For example,



The following algorithm uses a queue Q to keep track of the order in which nodes are visited. Initially, Q is empty and no node has been visited.

BFS(s)
    put node s in the queue Q
    while Q is not empty
        remove node v from the top of the queue
        visited(v) $\leftarrow 1$
        for all w adjacent to v
            if visited(w) $= 0$ and w is not in Q, then
            add w to Q (and mark that it is in Q).



On a graph that is simply a linear list, or a graph consisting of a "root" node v that is connected to all other nodes, but such that no other edges are in the graph, breadth first and depth first traversals can visit the nodes of the graph in the same order. On other graphs, however, an ordering that is possible with depth first search is not possible with breadth first search and vice versa. Can you construct an example of such a graph?