Graphs

Elena Dimitrova

March 2020

Definitions

Graph
- a set of nodes (vertices) connected by edges.

(In/Out)-degree of a node
- the number of edges (entering/exiting) a node.

Node u is “adjacent” to node v
- iff (v, u) is an edge.

Path
- a sequence $v_1, v_2, \ldots, v_n$, so that (v_i, v_{i+1}) is an edge for every i.

Cycle
- a path which begins and ends in the same node.

Examples: cities connected by roads, networks (including social)…

Types of graphs

Weighted/unweighted: if edges have weights, the graph is weighted

Directed/undirected: if for every edge (u, v), (v, u) is also an edge, the graph is undirected

Cyclic/acyclic: if there is a cycle in a graph, it is cyclic

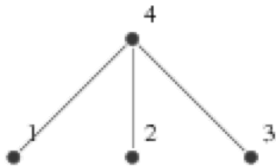
Connected/un-connected: if there is a vertex v, from which we can reach every other vertex, the graph is connected

Multi-graph(hypergraph): if there can be more than one edge between two vertices

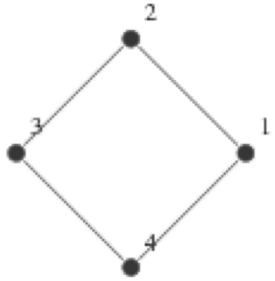
Directed acyclic graph (DAG): a lot like a tree, but not quite

Representations

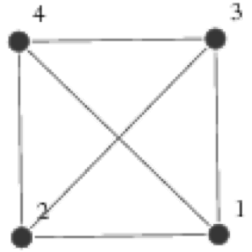
Adjacency matrix



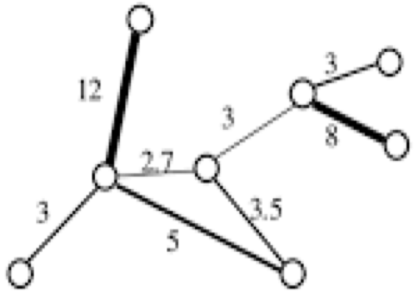
$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$



$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$



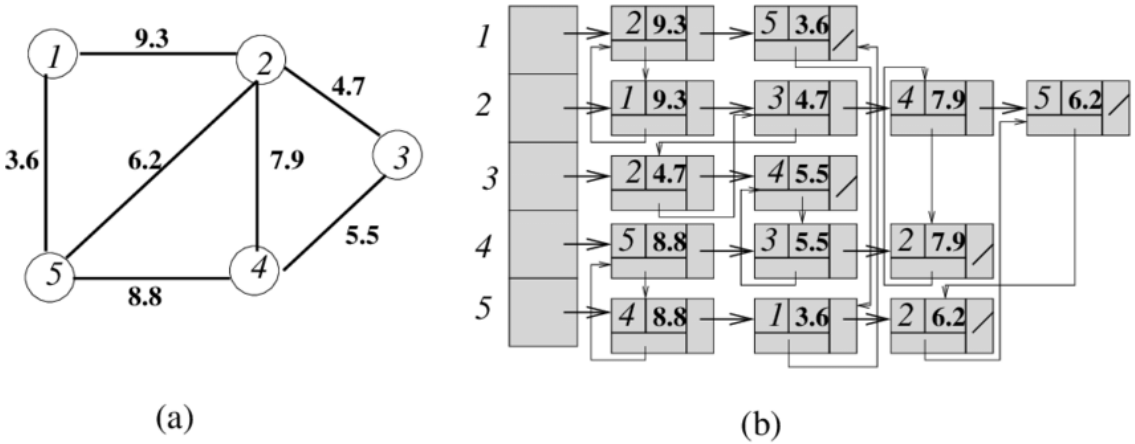
$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

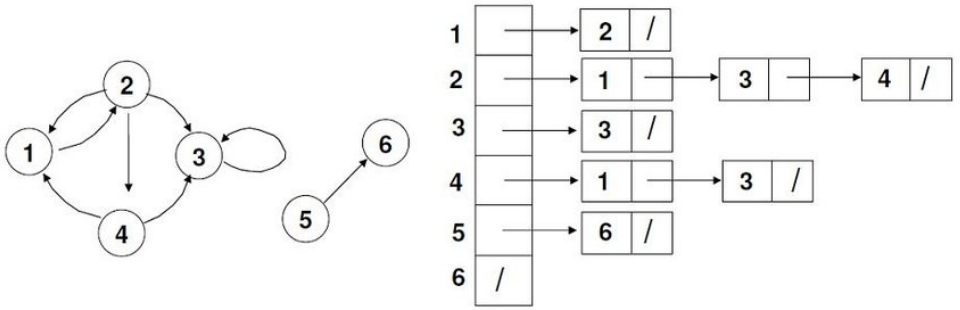


	1	2	3	4	5	6	7	8
1		3	0	0	0	0	0	0
2	3		12	2.7	5	0	0	0
3	0	12		0	0	0	0	0
4	0	2.7	0		3.5	3	0	0
5	0	5	0	3.5		0	0	0
6	0	0	0	3	0		3	8
7	0	0	0	0	0	3		0
8	0	0	0	0	0	8	0	

Representations

Adjacency lists





Representations

Comparison

	Adjacency list	Adjacency matrix
Add node	O(1)	$O(\|V\|^2)$
Remove node	O(1)	$O(\|V\|^2)$
Add edge	O(1)	O(1)
Remove edge	O(|V|)	O(1)
Check if adjacent	O(|V|)	O(1)
Memory	O(|V|+|E|)	$O(\|V\|^2)$

Algorithms, part 1: Traversal

Task: Check if there is a path from node A to node B.

Breath-first search (BFS)

```
choose start vertex
add to queue
while queue is not empty:
   remove the top element
   mark it as visited
   get list of all adjacent, unvisited nodes
   add them to queue
```

Complexity: O(|V|+|E|)

Side effects:
Is the graph connected?
Is the graph cyclic?

Depth-first search (DFS)

```
choose start vertex
add to stack
while stack is not empty:
    remove top element
    if top element is not visited:
       get list of adjacent, unvisited nodes
       add them to stack
       mark node as visited
```

Complexity: O(|V| + |E|)

Algorithms, part 2: Shortest path

Find shortest path from A to everywhere else in weighted graph

Dijkstra's algorithm

- but also, Bellman-Ford, which we won't cover

```
set distance to infinity for all nodes
set all nodes as unvisited
(optional): set all nodes as having no parent
distance[start_node] = 0
current_node = start_node
while unvisited nodes left:
   mark current_node as visited
   for all nodes adjacent to current_node:
      if distance[current] + weight(current, adj) < distance[adj]:
         distance[adj] = distance[current] + weight(current, adj)
         (optional): parent[adj] = current
   current_node = node with the shortest current distance
```

Restrictions?

Task 1

Implement the following interface for a weighted graph.
Which methods will change if the graph is a multi-graph?

```
class Graph {
   bool hasEdgeTo(int v1, int v2);
   bool hasEdgeFrom(int v1, int v2);
   int[] getNeighbors(int v);
   int[] getParents(int v);
   // finds all nodes at distance n
   // edges from a given node v
   int[] distanceN(int v, int n);

   // checks if the graph is directed
   bool directed();
   // checks if the graph is connected
   bool connected();
   // count the number of connected components
   int connectedCount();
   // checks if graph is cyclic
   bool cyclic();
   // finds the length of the longest path
   // do not factor in weights.
   int diameter();
   // find the length of the shortest path,
   // factoring in edge weights
   float shortest(int start, int end)

   int addNode(); // returns index of the node
   void removeNode(); // and all edges from it
   void addEdge(int v1, int v2, float w = 1.0);
   void removeEdge(int v1, int v2);

}
```

Task 2

Games

Let's write a simple game.

The level is represented by a matrix NxM (N, M ≥ 2), filled with '#'s and '.'s. '#'s represent impassable walls, and '.'s represent passable floor tiles.

There are two points of interest in the level – the player's position, which is always on a floor tile, marked with '@'; and the location of a monster, marked with 'M' (also, always on a floor tile). Assuming the player is caught in a trap for the next five monster steps, can the monster reach the player? Neither the player nor the monster can move beyond the level.

Bonus: Implement warp-around: if the player or the monster reaches the end of the level, they can reach the same tile on the opposite side of the level.

Sample:

```
##### N = 5          ..@.. N = 5
#.@.# M = 5          ..... M = 5
#...# Result: True   ##### Result: False
#.M.#                .....
#####                ..M..
```

Task 3

Games, continued

Let’s generate the level from the previous task.

A very simple way to do this is to fill the level with ”walls” (‘#’s) and ”tunnel”, by starting from a random tile, and choosing a random, adjacent tile which to convert into passable floor on every step. If there are no suitable adjacent tiles, then go back to the previous tile and ”tunnel” from there.

Adjacent cells are ‘suitable’, if converting them into a floor will create a floor tile with at most 4 adjacent floor tiles.

Sound familiar?

Task 4

Can X ”do this” with that?

Let’s look at a system which has a hierarchy of objects, and a hierarchy of principals. Each item in a hierarchy can contain any number of other items from the hierarchy, and can be contained in any number of items (think of files and folders). There are no circular dependencies. The hierarchy of principals functions in the same way: it contains ”users” and “groups”. Each user can belong to one or more groups, and each group can contain zero or more users or groups.

A set of strings, representing privileges is given. A permission is the tuple (object, principal, [privileges…]). Permissions are propagated down both hierarchies: if a group has a set of privileges on an object, then all principals in the group have those privileges on that object; and if a user has a set of privileges on an object, the user has those privileges on all objects contained in the object. Permissions for specific users take precedence over permissions for groups, and permissions granted specifically on an object take precedence over permissions granted on parent objects. Sets of privileges propagated from multiple “parents” are united.

Given the two hierarchies, the privileges, and a set of permissions, determine if a principal has a specific privilege on an object.

Task 4, example

Can X ”do this” with that?

Objects: folder0, folder1, folder2, file1, file2; folder0 contains folder2 and file1; folder1 contains file1

Users: group0, group1, user0, user1; group0 contains group1; group1 contains user0.

Privileges: Read, Write, Execute

Explicit permission: group0 has “Read” on folder0 and “Write” and folder1; user0 has “Execute” on folder2

Implicit permissions: user0 has “Read” and ”Write” on file1: “Read” because it is a member of group0, which has “Read” on folder0, which contains file1, and “Write”, because it is a member of group0, which has “Write” on folder1, which contains file1.

user0 does not have “Read” on folder2, because it is explicitly given “Execute” on folder2.

Literature

https://hurna.io/academy/algorithms/maze_generator/dfs.html

https://web.stanford.edu/class/cs97si/06-basic-graph-algorithms.pdf

https://visualgo.net/en

https://neo4j.com/graphgists/


ANY
QUESTIONS
?

Thank You