LAB 10

Agenda

- Announcements
- Different representation formats for Graphs
- DFS, a working example, how to implement it
- SQL

Stacks

Stacks is a structure to organize. This data structure can be both built up and whittled down.

Imagine you have a bunch of homework assignments piling up. In one scenario, you have a pile of papers on your desk. As you complete homework assignments, you pull the top paper off of the stack. Unfortunately, your professor can also add homework to the top of the pile. To complete your pile of homework, you would need to pull sheets off the pile one by one until you reach the bottom, and the pile is empty. This scenario is analagous to the stack. It follows a LIFO (Last In First Out) protocol, and is characterized by data being both accessed and added to the front (or in this case top) of the stack. With the stack, to add an element, you push, and to access the element you pop.

Graphs

For Adjacency list and matrix representation: Make sure that you explain the concept well and then show an example to explain it further.

A graph consists of a set of **vertices** and a set of **edges** that connect those vertices.

- Edges
  - Can be directed or undirected (**show the image in graphs.png**), weighted or unweighted.
- Vertices
  - Must be unique.

Why are graphs useful?

- Intuitive way to represent physical entities and the connections between them
- Great for pathfinding/maps/circuits/social networks.
- A common example of how graphs are used is to think about the Google Maps application on your phone, towns or cities can be the nodes and the different roads/state highways that connect can be thought of as edges. When you search a roadmap to go from a given city to another, it's like doing graph traversal to find the path from source city to your destination city.

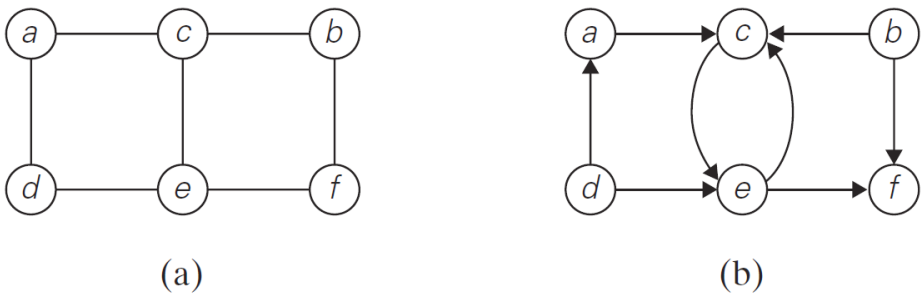

FIGURE 1.6 (a) Undirected graph. (b) Digraph.

Common Graph Representations

- **Adjacency Matrix**
    - A two-dimensional matrix, in which the rows of the matrix represent source vertices and columns represent the destination vertices.
    - As you can see that in the matrix representation, if we have `n` nodes then we have to store at-least n*n pieces of information because, `n` nodes means `n` rows and `n` columns. We will see in the next section that there is infact a way to store the graph by using less memory, which is called as **Adjacency List Representation**.

    -**Show this with an example**: Imagine having a 2D array or a list of lists where you can set the value for a specific element in the list to be 1 if there is an edge connection between those nodes.

    - In this case, if you want - you can also store the cost of the edge for a given position (this will be useful for weighted graphs).

Graph A

```
    A B C D E F
A [ 0 0 1 1 0 0 ]
B [ 0 0 1 0 0 1 ]
C [ 1 1 0 0 1 0 ]
D [ 1 0 0 0 1 0 ]
E [ 0 0 1 1 0 1 ]
F [ 0 1 0 0 1 0 ]
```

Graph B

```
      A   B   C   D   E   F
A [   0   0   1   0   0   0   ]
B [   0   0   1   0   0   1   ]
C [   0   0   0   0   1   0   ]
D [   1   0   0   0   1   0   ]
E [   0   0   1   0   0   1   ]
F [   0   0   0   0   0   0   ]
```

- **Adjacency list**
    - Collection of linked lists (or arrays), one for each vertex, that contain all vertices adjacent to the given vertex.
    - Advantages of using a adjacency list is that it's cheaper to store for sparsely connected graphs. As you saw that for a matrix, we need at-least $n*n$ pieces for information but if the graph is not connected very well (by which we mean that if there aren't many edges in the graph) then we can actually save memory by storing the graph as a list rather than a matrix.

Graph A

```
[a] -> c, d
[b] -> c, f
[c] -> a, b, e
[d] -> a, e
[e] -> c, d, f
[f] -> b, e
```

Graph B

```
[a] -> c
[b] -> c, f
[c] -> e
[d] -> a, e
[e] -> c, f
[f] -> None
```

- **Graph Traversal (Search) Algorithms:** Searching a graph is a quite common and useful operation to answer questions like, how many hops between any two points in a graph, what is the length of a path between given two vertices in a graph etc.
- Among the many graph search algorithms, the two fundamental ones are BFS (also known as Breadth First Search) and DFS (also known as Depth First Search). We are doing DFS for this lab.
- We will explain these algorithms to you with an example and then you will implement them as part of the code demonstration. For helping you, we have given the starter code for functions that will be used by DFS, but you have to implement the algorithms by making use of the starter code.
- In particular, we have given ready-made implementations of stack and queue that you can use to implement these algorithms.
- **Note:** DFS only work on unweighted (or uniformly weighted) graphs. Revise the concept of what it means to be unweighted or uniformly weighted.
- **Do the Depth First Search Algorithm** with an example.
- Depth-First Search (this is how you can do it procedurally i.e. a computer program for it)
    - Add a node to an empty stack
    - Add all neighbors of that node to the stack
    - Remove nodes from the stack one by one, every time adding all of its neighbors to the stack if that node has not been visited already
    - Repeat until stack is empty

SQL

SQL is short for "Structured Query Language". SQL was introduced in the lectures on Monday or Wednesday. We will build off of that material.

create a database, **create** tables for a database, **insert** into tables, and **query** to get data from a table.

We want to show interaction with a tool that is also used in industry. Databases are used every day and getting a chance to see it allows you to start to understand more aspects of computer science.

We will go through `ourSQL.py` together, solving the problems described in the file itself.
Order things should be done (with notes):

- Explain we give the data already
    - Top of file with table name
- do `createDatabase`
- do `createTable`
- do `insertValues`
- do `queryStatement`
- in the `if __name__ == "__main__":`
    - Where it says "cement", we need to add those (in solution)
        - As commiting forces the changes to the database
        - You will do this several times
            - The rest of the code is already completed for running the functions we wrote
    - We will write 3 queries
        - Fill in each `queryStatement` call
        - The last one you will need to talk about `distinct`

Note: The purpose of the first four functions is to abstract out the `sqlite3` functions common to all SQL queries.

File Locations should be:

- `Laboratory/Lab10/DFS.py`
- `Laboratory/Lab10/graph_test.py`
- `Laboratory/Lab10/Graph.py`
- `Laboratory/Lab10/Stack.py`
- `Laboratory/Lab10/ourSQL.py`