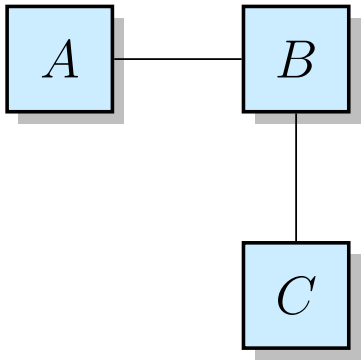
Graphs

Sujit Chakrabarti

In several of the next videos we are going to learn about graphs, arguably the most important and versatile abstract data types.

1 Graphs in Real Life

We often like to represent the presence of a relation or the lack thereof between two entities in the following manner:



The general common-sense interpretation of this picture is that A is related to B, B is related to C. But there seems to exist no relation between A and C. In this Internet age, the best example we can quote of such a scenario is FaceBook, or LinkedIn. A, B and C represent the users, and the existence of a line between any two elements of this type indicates that they are 'friends' of each other. There are n number of cases where entities and relations between them are shown pictorially using something like the above, e.g. flight/train/bus routes (transportation engineers), electric circuits (electrical engineers), class diagrams, flowcharts (computer scientists), finite state machines add more examples etc .

All these pictures are examples of graphs. Graphs are mathematical objects with *nodes* (representing things) and *edges* (representing relations). Nodes are often called *vertices*; edges are also called *arcs*.

2 Taxonomy of Graphs

Graphs can be of various types depending on what they are being used to represent. For example, consider the 'Friendship' graph in FaceBook. Here, two users are either friends of each other, or they are not. If A is B's friend, it implies that B is A's friend. That is, the *friendship* relation is symmetric. Graphs used to represent such relations would typically be drawn with edges being line segments. Such graphs are called undirected graphs.

On the other hand, consider a flowchart representing an algorithm. Here, an edge from A (representing an instruction in the algorithm) to B (representing another instruction) means that it is possible for control to flow directly from A to B in at least one of the executions of the algorithm. Here, an edge (called a control flow edge)from A to B defines some sort of successor relation, wherein B is A's successor. Edges in such graphs are typically arrow to indicate the directionality of the relation. These relations (like successor) are assymmetric. Therefore, an arrow flowing from B to A would mean exactly the opposite (i.e. A is successor of B). Such graphs are called *directed graphs* or *digraphs*.

There are graphs which are used to represent multiple types of relations between entities, some of them symmetric and some assymmetric. Take, for example, a graph that represents friendship between two Facebook users using an undirected edge, and 'follows' relation using an arrow or directed edge. Such graphs, with both directed and undirected edges are called *mixed graphs*.

Graphs may represent relations which are such that two entities may have only one relation, e.g. Facebook friendship. Such a graph will always have at the most one edge between two nodes. Also, a person can't be his own friend, i.e., there can't be an edge starting and ending on the same node. Such graphs are called *simple* graphs. On the contrary, there are graphs which represent relations multiple instances of which may exist between two entities. Again take the example of a flowchart, in which multiple control flow edges may exist between the same pair of nodes. Similarly, there may be edges starting and ending on the same node. Such graphs are often called *multigraphs*.

For the purpose of this class, we will often be dealing with undirected simple graphs. When we wish to deal with other types of graphs, we will make it explicit.

Two nodes are called neighbours if there exists an edge between them. Mathematically, predicate $neighbour(A, B)$ holds true if node A and node B are neighbours. A path is the sequence of nodes such that each pair of consecutive nodes in it are neighbours in the graph.

Example

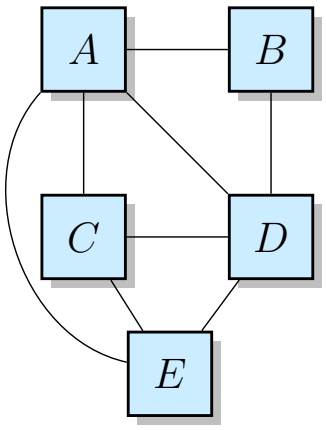


Figure 1: An undirected graph

Fig. 1 shows an undirected graph. Nodes are A, B, C, D and E. A and B are neighbours as there's an edge between them. So are B and D. ABD is a path since A and B are neighbours and B and D are neighbours. In contrast, ABC is not a path because B and C are not neighbours.

3 Graph ADT

4 Traversals

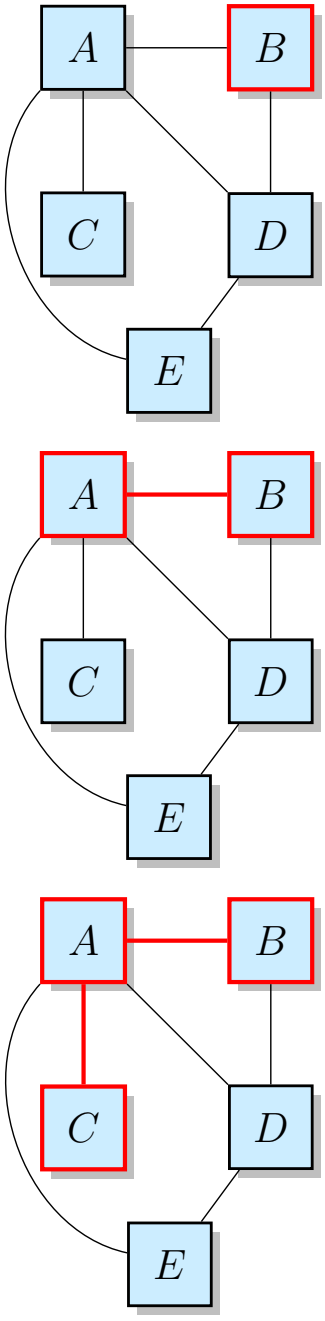
Most algorithms which work with graphs involves visiting the nodes in certain order and doing 'something'. For instance, you may want to print the graph, find the shortest path between two nodes, check whether one graph is same as the other etc. etc.

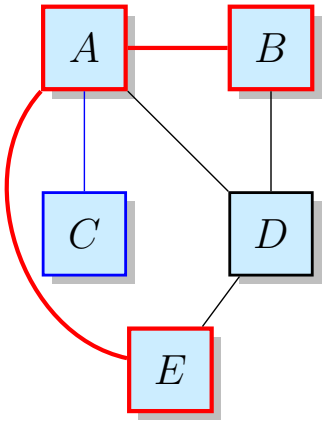
Most – if not all – of these algorithms are built on top of or are variations of two prototypical traversal techniques: the depth first search and breadth first search. Studying DFS and BFS also gives us a very good way of understanding how the graph abstract data type can be used to implement algorithms.

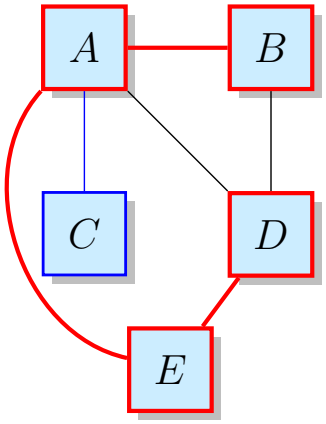
We now look at these two search/traversal techniques.

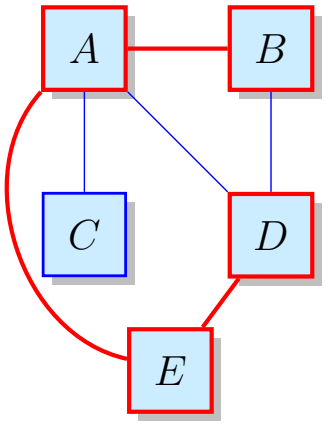
4.1 DFS

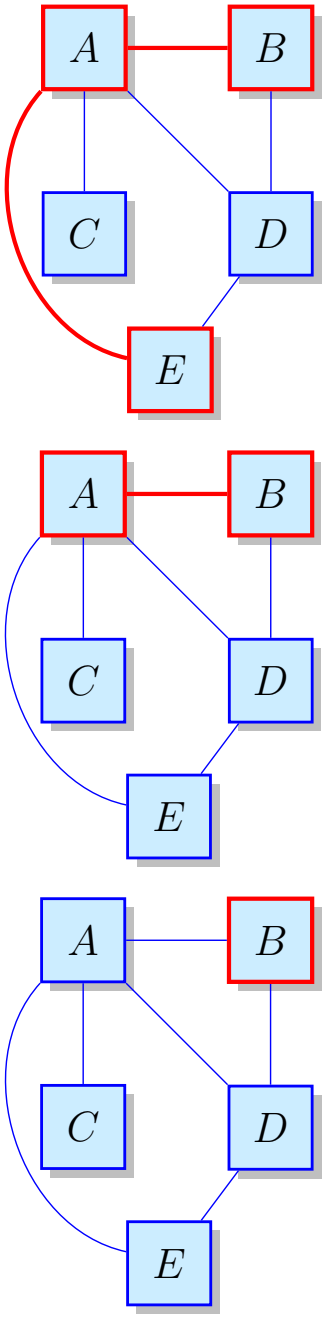
4.1.1 Example

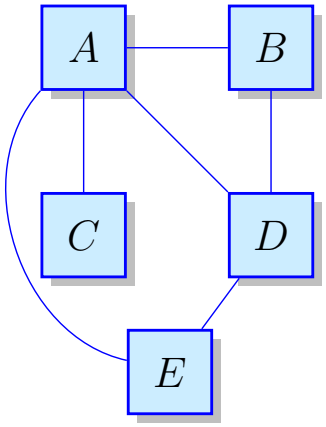



A
B
C
D
E


A
B
C
D
E


A
B
C
D
E


A
B
C
D
E
A
B
C
D
E
A
B
C
D
E



4.1.2 Algorithm

Algorithm 1 Depth first search algorithm

```
procedure dfs(node)
    for all n' ∈ neighbours(n) do
        if n' ∈ visited then
            return
        else
            DFS(n')
        end if
    end for
end procedure
```

4.1.3 Implementation

Sets and Maps

We have learned about a few abstract data types like lists, stacks and queues, and currently we are learning about graphs. We have been mentioning that they are drawn from a fairly large collection of ADTs which represent various mathematical abstractions of collection types. By collection types, we mean data-types which can be used to store multiple objects, which may either be atomic or collections themselves. List is a collection, since it can store any number of elements. Same is the case with stacks. They differ in the way they put or don't put a restriction in the order of insertion and removal of those elements.

Before we proceed to see the implementation of DFS as Java code, I would like to introduce to two more important ADTs. The main reason to

introduce them is because they useful, and fundamental and you will get to use them in a variety of scenarios. Another reason is that we are going to use them in a limited way, in the implementations of all the data-structures and algorithms that we come across in this and future videos.

The two ADTs are *sets* and *maps*.

Sets are unordered collections of elements with no duplicates allowed. So, the kind of operations you might expect from sets would be: adding an element or removing it; checking if an element is there in the set; checking if it's empty or not etc. Further, we can iterate upon a set, i.e. I can run a for-loop on a set where I do some computation with each of its element. What's not allowed is to access elements by their index: viz. for a set s and integer i, s.get(i) mayn't be defined. Where can I use sets? Well, you will see them being used in many of the programs that you will see shortly. But, in general, a good way of judge where they can be used is: look for lists where you are not interested in the ordering, or positioning of the elements, and you wish to prevent multiple instances of the same element from getting into the collection. And that's how we are going to use sets in our programs. Of course, if you wish set ADT to represent the mathematical notion of sets in a more complete way, you might want to define additional operations on them like union, intersection, complimentation and set minus. But as of now, we aren't concerned with those capabilities of sets.

You could use lists to represent sets. In that case you might have to ensure that certain restrictions, e.g. the non-duplication of elements is maintained. You could also use lists to implement a class to represent sets. And I would encourage you to do so as an exercise.

Maps, at least in the world of programming, provide a connection from members of a source set of elements to those of a target set of elements. The source set is often called the domain, and the target set the range. The domain is called the key set and the values in it are called keys. The range is called the value set, and the values in it are called values. So, a map can be used, for example, to represent a price-list, where commodities are mapped to their prices, a word-dictionary where a word is mapped to its meaning. You could use it to map authors to the set of books they have authored, and so on. What kind of operations would you expect maps to have? The most common operation is to be able to get a value in the range set by the argument in the domain. So, `price_list.get(potato)` will give you the running rates of potatos. `authors.get(tolstoy)` would give you a set of books Tolstoy has written, namely {"Anna Karenina", "War and Peace" }. The other operation is to add/update a mapping. So, `price_list.put(onion, 35)` will create a new entry for onion with price 35 in case it was not there. Otherwise, it will update the existing mapping. In any case, the result will be that hereafter

`price_list.get(onion)` will fetch you 35.

Just as an interesting addition, you might look at lists as restricted maps, where the values in the domain are always non-negative integers. And that should give you a hint as to where you could use a map. Look for cases where you feel like using a list which is indexed by things other than non-negative integers. That's a good place to use maps.

Maps can be represented using lists of key-value pairs. The first element of each of these pairs would be an element of the domain, and the second element would be the value it maps to. You could also implement a map ADT built upon lists. And again, I encourage you to do so as an exercise.

In our programming demonstration, you will see sets and maps being used at many places. We will go ahead use Java standart library classes for these ADTs. The reason why we don't implement our own sets and maps here is partly because we don't want to spend our video time in discussing their internal details while we wish to discuss something else. Another reason is that these library implementations are optimised and would provide much better performance as compared to any list based implementation that we may create.

4.1.4 Demo Examples

1. Pre-order DFS numbering (`DFS1.java`)
2. Set of reachable nodes from a given node (`DFS2.java`)

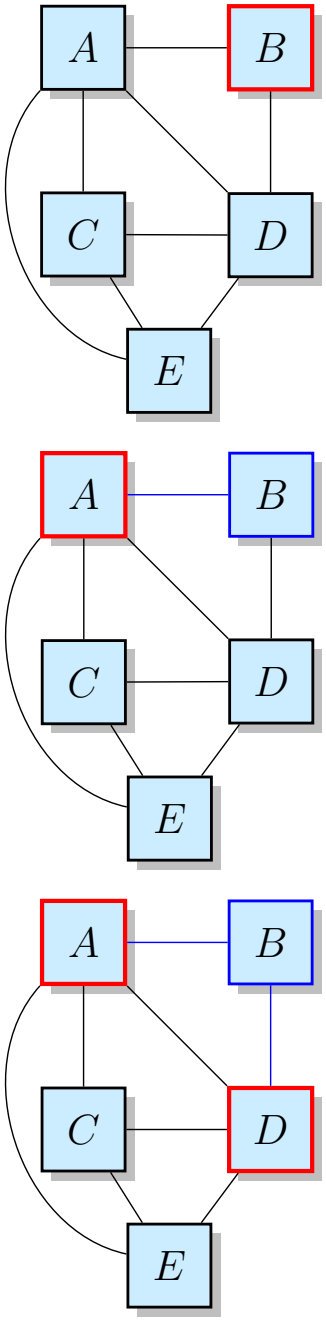
4.1.5 Programming Problems

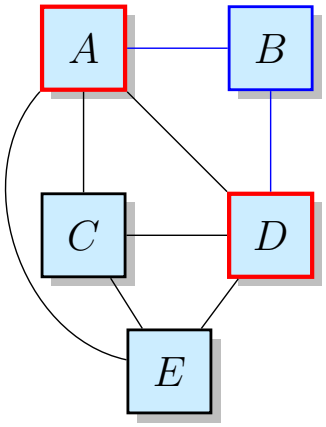
Use DFS to:

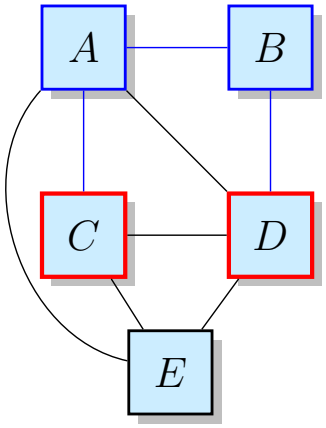
1. Find if a graph is connected (i.e. there exists a path between all pairs of nodes).
2. Find if a graph has a cycle (i.e. there exists a node which has a path to itself).

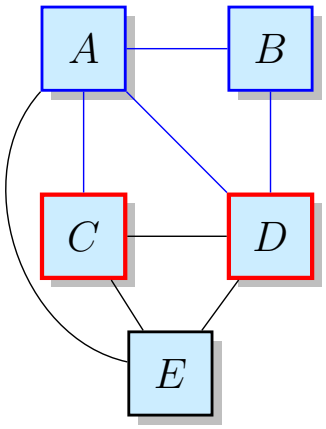
4.2 BFS

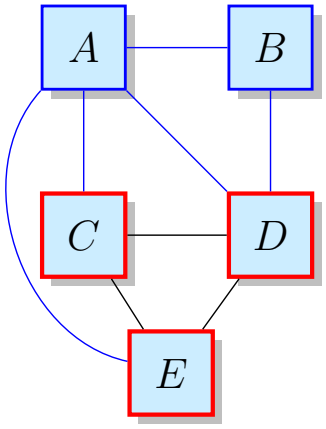
4.2.1 Example

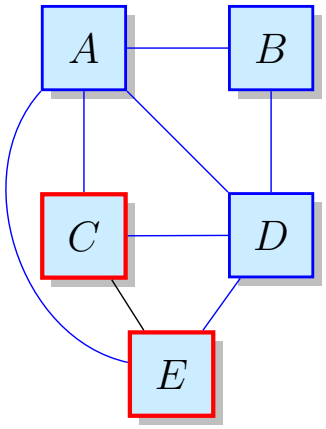


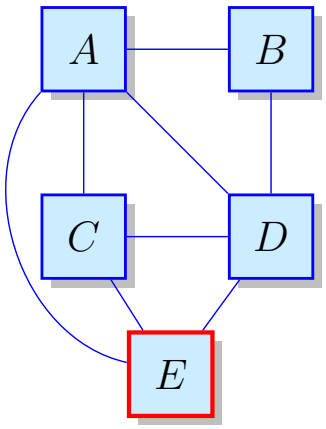

A
B
C
D
E

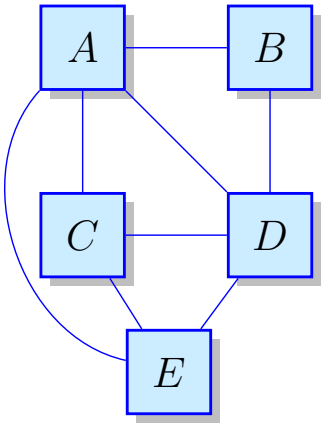

A
B
C
D
E


A
B
C
D
E


A
B
C
D
E


A
B
C
D
E


A
B
C
D
E



4.2.2 Algorithm

Algorithm 2 Breadth first search algorithm

```
procedure bfs(node)
    queue ← new Queue
    visited ← {}
    ENQUEUE(Q, node)
    add n to visited set
    while Q is not empty do
        n ← DEQUEUE(Q)
        for all n′ ∈ neighbours(n) do
            if n′ ∉ visited then
                ENQUEUE(Q, n′)
            else
            end if
        end for
    end while
end procedure
```

5 Implementations

5.1 Edge List

5.2 Adjacency Matrix

5.3 Adjacency List

5.4 Comparison

6 Directed Graph

6.1 Dijkstra

7 Directed Acyclic Graphs

7.1 Topological Sort