ENGSCI 233 - Combinatorics
Lecture 1

Colin Simpson

Semester One, 2022

Department of Engineering Science
University of Auckland
(Beamer Theme: Metropolis)

Introduction

Combinatorics is a branch of mathematics concerned with sets of objects, usually conforming to certain conditions.

It can be used in computational techniques to develop algorithms for solving many different problems. In this course we cover:

- Searching a tree data structure
- Sorting algorithms
- Dijkstra's shortest path algorithm

Revision of Data and Objects

Existing Data Types

Once data is available in a computer program (e.g. read in from file, or generated during a simulation), it is often convenient to store it as a particular data type. Some already familiar types:

- Integer
- Float
- NumPy Array
- Python List
- String

Here is a handy reference list of some common data types.

Custom Data Structures

Week 2 introduced the idea of an object belonging to a class. We used this to construct a linked list and network class.

A linked list object contains a sequence of connected nodes. The nodes themselves belonged to another class that we wrote.

The linked list serves as a distinct data type from a built-in Python list or NumPy array, with its own pros and cons.

Networks

A network can be thought of as a set of nodes connected to each other by arcs (which will have a source node, a destination node, and possibly a weight).

A linked list, where nodes have a single outward connection to one other node, could be considered a special case of a network.

Consider a railway network as a realistic example:

- An individual train station can be represented as a node, with its name as an attribute.
- Stations are connected by train lines, which can be represented by an arc and may have a weight.

Defining a Node Class

```
class Node(object):
    def __init__(self):
        self.name = None
        self.value = None
        self.arcs_in = []
        self.arcs_out = []
```

Each node will have a name (i.e. string), a value, and a list of arc objects that either start at this node (`arcs_out`), or end at this node (`arcs_in`). This is a little different from our linked lists.

Defining an Arc Class

```
class Arc(object):
    def __init__(self):
        self.weight = None
        self.from_node = None
        self.to_node = None
```

Each arc will have a weight value, a node object as its origin, and a node object as a destination.

Defining a Network Class

```
class Network(object):
    def __init__(self):
        self.nodes = []
        self.arcs = []
```

The only attributes required for the Network class is a list of all node and arc objects. The node and arc objects themselves contain all the information about the network structure. We can also write methods to perform actions on the objects within the network.

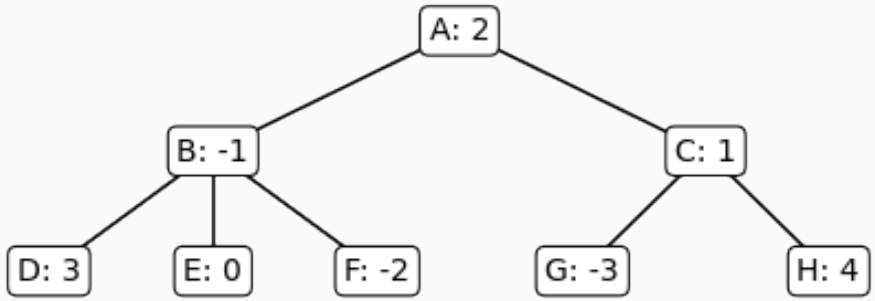
Networks have many useful applications in computational techniques.

Searching a Tree Data Structure

Tree Data Structure

We will consider a special data structure called a tree:

- The node connections resemble a family tree.
- The first generation contains only the *head node*.
- Otherwise, one parent and any number of children nodes.



The head node has a name of ‘A’ and a value of 2. The head node has two children, the nodes with names ‘B’ and ‘C’, and so on...

Tree Class

In Python, we can define a tree as a derived class of the `class Network` seen previously. This inherits all attributes and methods, which can then optionally be overloaded.

This `class Tree(Network)` includes:

- Two attributes: list of nodes and arcs.
- Method `build(...)`: constructs the network from a tuple, rather than reading from a file.
- Method `add_daughter(...)`: joins a node to its parent node and recursively add its children nodes.

Searching a Tree Data Structure

Consider the task of finding a specific node value within a tree data structure. There are two main ways to approach this search:

- Breadth-first search: check all nodes in a generation, before moving onto the next generation.
- Depth-first search: explore down a single "family line" before moving onto the next one.

In each case, we start at the head node, and end either when the value is found or all nodes have been checked.

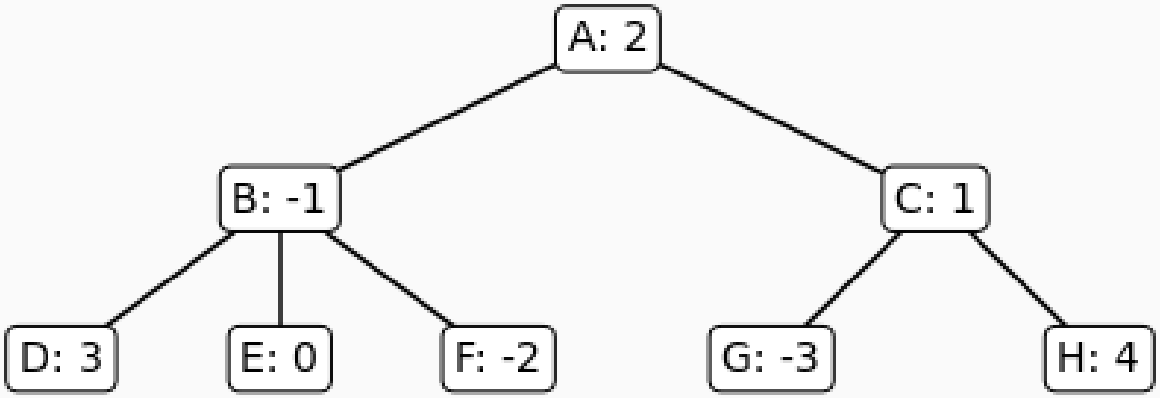
Breadth-First Search

Create a Linked List object which contains just the head node of the tree data structure – this linked list can be thought of as a queue.

Iteratively perform the following operations:

- Pop (i.e. access and remove) the next node from the queue.
- Query the node value to determine if the search can end.
- If not, append any children node(s) to the queue.

Breadth-First Search: Example



Depth-First Search

There are two ways to implement a depth-first search:

1. Call a recursive function on the head node. If node value matches the search, return that node.

   Otherwise, call this same function on any children node(s). If no children, return `None`.
2. Use a linked list as a queue, similar to breadth-first search. Except this time, pop nodes from end of queue.

Depth-First Search: Example