ENGSCI 233 - Lab Worksheet

Combinatorics

Date: April 13, 2022

Lab Introduction

Your objective in this lab is to implement two different combinatoric algorithms, a tree search algorithm and Dijkstra's shortest path algorithm. You will then use your shortest path algorithm to investigate travel through a network of connected Pacific Islands, simulating the migration of early Polynesian explorers from Southeast Asia to Aotearoa.

The following files have been made available to you:

- *functions_comblab.py*
- *test_comblab.py*
- *waka_voyage_comblab.py*
- Network files in *simple_network.txt* and *waka_voyage_network.txt*.

Any classes or functions that you write/complete should include a suitable docstring. Note that test cases, like those in *test_comblab.py*, will generally be self-documenting (i.e. the test name explains its purpose) or can be documented very briefly.

Provided Code

A lot more pre-completed code is provided for this lab than in your previous labs. The reason for this is that we working on network-related algorithms. It would not be practical to have you write a fully working set of classes for representations of linked lists, tree data structures and networks. It is also an important skill to learn to work with and build off of pre-existing code (even if that code is sub-optimal!).

Below is an overview of the already complete code that has been provided to you. These classes should **not** be modified during your lab attempt:

1. `class LinkedListNode`

   Implementation for nodes that can be used in a linked list data structure. The nodes in a linked list behave more simply than those required for a more general network, so have therefore been defined using their own class.

Includes two attributes, `self.value` and `self.pointer`. The node value is stored in `self.value`. The outward connection of the node is stored in `self.pointer`. If equal to `None`, the node must be at the end of the linked list. Otherwise, it is equal to the node object that comes next in the linked list.

Includes only a single method: `def next`. This simply returns the node object that this one points to i.e. the next node in the linked list.

2. `class LinkedList`

   Implementation of a linked list data structure.

   Includes a single attribute, `self.head`. If the linked list is empty, this attribute is equal to `None`. Otherwise, the attribute is equal to the head node of the linked list.

   Includes the methods `def append`, `def insert`, `def pop`, `def delete`, `def get_length`, `def get_node` and `def get_value`. Basic docstrings for each method are provided, though the method names are somewhat self-documenting.

3. `class Node`

   Implementation of nodes that can be used in a network. The nodes defined in this class behave distinctly from those defined for use in the linked list data structure.

   Includes four attributes. The node value is stored in `self.value`. The node name (i.e. unique ID) is stored in `self.name`. Lists of inward and outward directed arcs are stored in `self.arcs_in` and `self.arcs_out`, respectively.

   Includes no methods other than `def __init__` and `def __repr__`.

4. `class Arc`

   Implementation of single-directed arcs (i.e. connection between nodes) that can be used in a network.

   Includes three attributes. The arc weight, if not equal to `None`, is stored in `self.weight`. The node from which this arc originates is stored in `self.from_node`.

The node at which this arc terminates is stored in `self.to_node`.

Includes no methods other than `def __init__` and `def __repr__`.

5. `class Network`

   Implementation of a generalised network structure, which includes a set of nodes connected by single-directed arcs.

   Includes two attributes. The list of all nodes in the network is stored in `self.nodes`. The list of all arcs in the network is stored in `self.arcs`.

   Includes the methods `def add_node`, `def join_nodes`, `def read_network` and `def get_node`. These are used to read data from an external data file and construct the network by initialising the relevant nodes and arcs. The method names are somewhat self-documenting.

6. `class NetworkError`

   A simple class used to raise an error from within the network class.

7. `class Tree`

   Implementation of a tree data structure. It is a *derived class* based on `class Network` i.e. it inherits the attributes and methods of that class. While it does **not** overload any methods or attributes, it does include some additional attributes and methods (detailed below).

   Includes one additional attribute, `self.head`. This is equal to the node that is at the top of the tree data structure.

   Includes additional methods `def build`, `def add_daughter`, `def assign_values` and `def show`. These are used to construct and visualise the tree data structure, which is performed differently to the more general network class.

Task 1: Searching a Tree Network

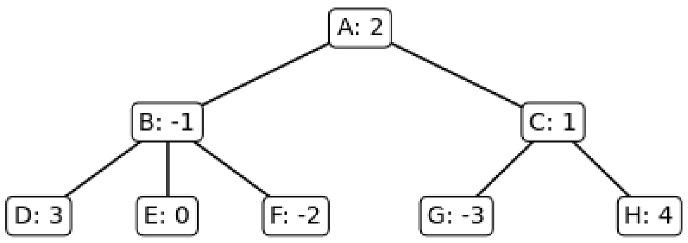
Background

You will be implementing either a depth-first or breadth-first (your choice) search algorithm to be applied to a tree data structure. This task will make use of the following classes described previously: `class Tree`, `class LinkedListNode` and `class LinkedList`.

Exercises

Complete the following exercises for this task:

1. Write some pseudocode and perform a few steps of your search by hand for the tree data structure discussed in lecture:

   Complete the test function `def test_search_value_negative_three` in *test_comblab.py*. Optionally, you may want to write an additional test for a search value of 5, to check if your function behaves as expected when the search value is not present in the tree.

2. Complete the function `def search` in *functions_comblab.py*.

   The function has two inputs: `tree`, which is an object belonging to `class Tree`, and `search_value`, which is an integer value to be searched for in the network.

   The function has one return, which is the name of the node as a string (**not** the node object itself) that contains the search value. If the search value is not found in the tree, the function should return `None`.

   You may optionally write additional helper functions that are called from within `def search`, but do not modify that functions name or its arguments.

   It is recommended that you use a linked list object to act as a queue of nodes to search. Within each iteration, you should pop a node from the linked list and check for a match between the node value and the search value. If a match is found, you should return the node name, otherwise you should append the children nodes to your linked list. If all nodes have been searched and the search value has not been found, you should return `None` i.e. indicating that no node contains the search value.

Task 2: Dijkstra's Algorithm

Background

You will be implementing Dijkstra's shortest path algorithm on a generalised network that assumes singly-directed arcs between nodes. This task will make use of the following classes described previously: `class Node`, `class Arc` and `class Network`.

Exercises

Complete the following exercises for this task:

1. View the simple network defined in *simple_network.txt*. Consider the first line:

   ```
   A,B;2,C;4
   ```

   This can be interpreted as the node 'A' having outward connections to:

   - Node 'B' with arc weight 2
   - Node 'C' with arc weight 4

   Consider now the final line of the file:

   ```
   F
   ```

   This can be interpreted as the node 'F' having no outward connections. Based on this information, sketch the network provided in *simple_network.txt*. Perform Dijkstra's algorithm by hand for this test network, under assumption of a source node 'A' and destination node 'F'.

   Complete the test function `def test_shortest_path_simple_A_to_F`, which already includes commands to parse the network file and construct the network object. Note that this test therefore has a dependency with `class Network` i.e. it relies on that class in order to be tested in a meaningful way. This strictly makes our test an *integration test* rather than a *unit test*.

2. Complete the function `def shortest_path` in *functions_comblab.py*.

   The function has three inputs: `network` is an object belonging to `class Network`, `source_name` is the string name of the source node, and `destination_name` is the string name of the destination node.

   The function has two returns: `distance` is the distance associated with the shortest path, and `path` is a Python list containing the node names (**not** the node objects themselves) of the shortest path, which should be ordered from source node as the first item to destination node name as the final item. For example, if a shortest path from source 'A' to destination 'C' is 'A' to 'B' to 'C', then the return would be `path = ['A', 'B', 'C']`. If no shortest path is found, then both returns can be set to `None`.

   This is a fairly complicated algorithm that involves multiple operations. You may optionally write additional helper functions that are called from within `def shortest_path`. Your final algorithm will need to include operations such as:

   - Initialise the value of each node in the network. The source node will have a value of `[0, None]` (i.e. a list containing the initial distance and predecessor node), all other nodes will have a value of `[float("Inf"), None]`.

   - Initialise an unvisited set, using a Python `set`, that contains all nodes in the network. Note that you may want to store the node names in this set, rather than the node objects themselves. The method `network.get_node` can be used to easily retrieve the node object itself if its name is known.

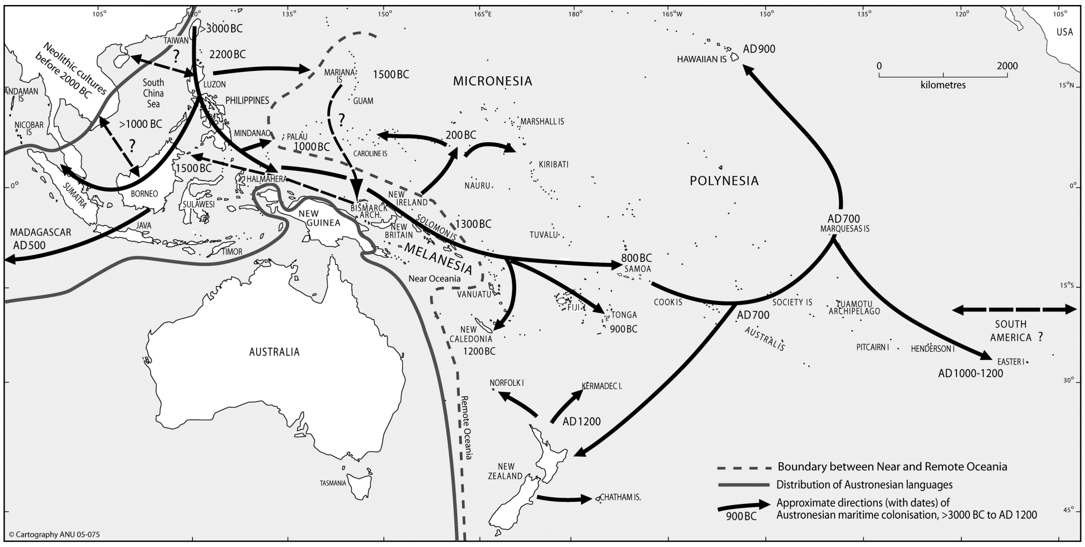
   - Within each iteration, find the node in the unvisited set with the smallest distance. This node will need to be removed from the unvisited set.

   - Within each iteration, calculate prospective new distances to outward connected nodes from our node just removed from the unvisited set. If the new distance is smaller than that previously stored for a node, update the node value to be the new distance and predecessor node.

Task 3: Great Pacific Migration

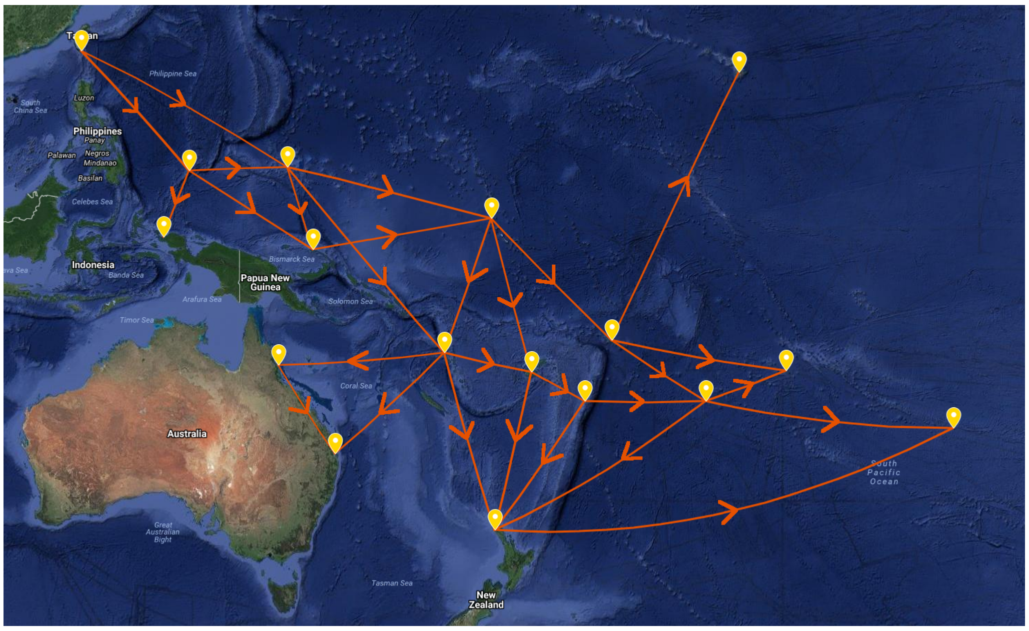
Māori oral tradition tells the story of Kupe, a great fisherman, who chased a giant Wheke (octopus) from his homeland of Hawaiki across the Pacific Ocean, eventually discovering Aotearoa. From archaeological evidence, we know that Māori had settled in New Zealand by 1300 AD, but from where and how did they arrive? By tracing the evolution of culture and language throughout Polynesia, it is now believed that the settlement of New Zealand occurred during the late stages of a much larger wave of Pacific migration. Highly capable seafarers originating from West Micronesia travelled between islands in sophisticated Waka ama (outrigger canoes), using weather systems and the stars for navigation. These people carried with them plants and animals to cultivate new settlements, and they are thought to have maintained regular trading voyages between islands.

While it is difficult to identify the exact route and dates of the migration, a partial picture can be constructed. This would appear to place the location of Hawaiki in Eastern Polynesia (the Society Islands, Cook Islands or French Polynesia):



GK Chambers(2013).

This exercise imagines a restaging of the Great Pacific Migration using modern-day sailing craft and navigational equipment. In keeping with the early explorers' desire to discover and settle new lands, we shall restrict individual sailing legs to discrete "hops" (arcs) between adjacent islands (nodes). Instead of assigning each hop a distance, we shall instead imagine a *travel time* (arc weight), which incorporates both the distance between islands as well as difficulty of passage due to prevailing currents or weather. Below is our reconstruction of the voyage:



You have been provided with a network file, *waka_voyage_network.txt*, that represents the nodes and arcs associated with our modern restaging of the Great Pacific Migration.

Exercises

Complete the following exercises for this task:

1. Within *waka_voyage_comblab.py*, use your `def shortest_path` to determine the route and travel time of the shortest path from *Taiwan* to *Hokianga*, where Kupe first arrived in Aotearoa aboard his great Waka, Matahourua.

2. Within *waka_voyage_comblab.py*, use your `def shortest_path` to determine which **pair of islands** in the network are separated by the **greatest travel time**. One way of doing this is to iterate across each possible pair of nodes as source and destination nodes, looking for the smallest distance returned by your Dijkstra's algorithm. Travelling from a node to itself, with a shortest path distance of zero, is not a valid answer to this task.

3. Answer the following questions relating to this task:

   - What is the shortest path (include both the travel time **and** route) from Taiwan to Hokianga?

   - Which pair of locations in the network are the furthest apart? Briefly describe how you arrived at this answer.

Submission Instructions

For this lab, you should submit the following:

- *functions_comblab.py*
- *test_comblab.py*
- *waka_voyage_comblab.py*
- A PDF or Word file containing your answers to the Task 3 questions

You do **not** need to submit any other files. Do **not** modify the file names, but don't worry if Canvas appends a `-1` or `-2` to the end.

Remember, all submissions are compared against each other and those from previous years. Copying someone else's code and changing the variable names constitutes academic misconduct by *both* parties.