

M269 Tutorial: Practice Block Three

Dr James Burton, March 2023

Graphs and traversals.

The following exercises are to give you some practice working with *graphs* and in particular with implementing various *traversals*. My solutions are in the `solutions` branch of the repository at <https://github.com/jimburtont/M269>.

TODO: REFERENCES TO THE BOOK.

The implementation of the graph ADT in `graph.py` stores a list of *nodes* (also called *vertices*), which can be values of any type, and a list of *edges*, which are pairs of nodes representing connections in the graph. It is a *directed* graph, in that an edge (`n1`, `n2`) is a connection between `n1` to `n2` and does not mean there is necessarily any connection between `n2` and `n1`. In the latter example we say that `n1` is the *source* of the edge and `n2` is its *target*.

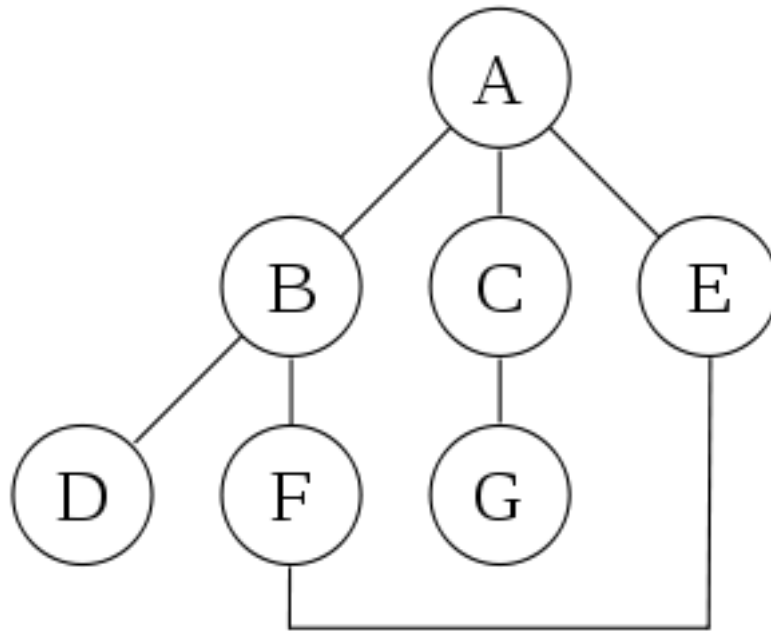


Figure 1: Figure 1: Visualisation of a graph

Figure 1 shows a graph with seven nodes labelled 'A' to 'G' and seven edges: (A,B), (A,C) and so on. Because no direction is shown along the edges we can assume this is an undirected graph, so there are also edges (B,A), (C,A) and so on. Because all nodes are reachable by some path, whichever node we start from, this graph is said to be *connected*.

Depth-first traversal

A depth-first traversal begins at a given node, **n**, then selects one of **n**'s neighbours and continues in this way until it reaches a node with no neighbours, **p**. It then checks whether the node visited prior to **p**, **o** has any neighbours: if it does, it selects one of those and once again continues until it reaches a dead end. If **o** has no neighbours it carries on reversing along the path it took until it finds a previously visited node that has neighbours.

Depth-first traversal can be implemented *recursively* or *iteratively* (ie with a loop). Iterative implementation commonly use a *stack*.

1. Pseudocode for recursive implementation

```
PROCEDURE DFT_REC(G, v): list
  discovered := empty list
  add v to result
  FOR all directed edges from v to w in G DO
    IF vertex w is not in discovered THEN
      add DFT_REC(G, w) to discovered
    END IF
  END FOR
  RETURN discovered
END PROCEDURE
```

2. Pseudocode for iterative implementation

```
PROCEDURE DFT_ITER(G, v): list
  discovered := empty list
  stack := empty stack
  stack.push(v)
  WHILE stack is not empty DO
    v = stack.pop()
    IF v is not in discovered THEN
      add v to discovered
    END IF
    FOR all edges from v to w in G DO
      push w onto stack
    END FOR
  END WHILE
END PROCEDURE
```

Breadth-first traversal

A breadth-first traversal begins at a given node, **n** then visits each of **n**'s neighbours. It then selects one of **n**'s neighbours and continues in this way until it has visited all reachable nodes.

Breadth-first traversal is most often implemented *iteratively* using a *queue*.

1. Pseudocode

```

PROCEDURE BFS(G, v): list
  discovered := empty list
  queue := empty queue
  add v to discovered
  enqueue v
  WHILE queue is not empty DO
    w := dequeue from queue
    FOR all edges from w to x in G DO
      IF x is not in discovered THEN
        add x to discovered
        enqueue x
      END IF
    END FOR
  END WHILE
END PROCEDURE

```

Exercises

Implement the following methods in the file `graph.py`.

- `disconnected(self)` -> `set`: Collect the set of disconnected nodes (those which are not part of any edge) in the graph.
- `elem(self, n)` -> `bool`: Returns true if `n` is a node in this graph, otherwise false.
- `neighbours_out(self, n)` -> `set`: Collect the set of nodes that are connected to `n` by an edge, where `n` is the source of that edge. Throw a `RuntimeError` if `n` is not in the graph.
- `neighbours_in(self, n)` -> `set`: Collect the set of nodes that are connected to `n` by an edge, where `n` is the target of that edge. Throw a `RuntimeError` if `n` is not in the graph.
- `traverse_df_rec(self, n)` -> `list~`: Implement a recursive depth-first traversal of the graph starting at `n` using the pseudocode above. Return the node labels in a list. Throw a `RuntimeError` if `n` is not in the graph.
- `traverse_df_iter(self, n)` -> `list~`: Implement an iterative depth-first traversal of the graph starting at `n` using the pseudocode above. Return the node labels in a list. Throw a `RuntimeError` if `n` is not in the graph.
- `traverse_bf(self, n)` -> `list`: Implement an iterative breadth-first traversal of the graph starting at `n`, and returning the node labels as a list. Throw a `RuntimeError` if `n` is not in the graph.

Extension: Breadth-first search

With a minor extension this algorithm can be adapted to search for a particular node, forming a breadth-first *search* of the graph. The pseudocode below gives the algorithm for finding *every* path from `source` to `target`

```

PROCEDURE BFS(G, source, target): <type of labels in G>
  discovered := empty list
  queue := empty queue
  add source to discovered
  enqueue source
  WHILE queue is not empty DO
    v := dequeue from queue
    IF v == target THEN
      RETURN v
    FOR all edges from v to w in G DO
      IF w is not in discovered THEN
        add w to discovered
        enqueue w
      END IF
    END FOR
  END WHILE
END PROCEDURE

```

We then need to find the *shortest* path from source to target.