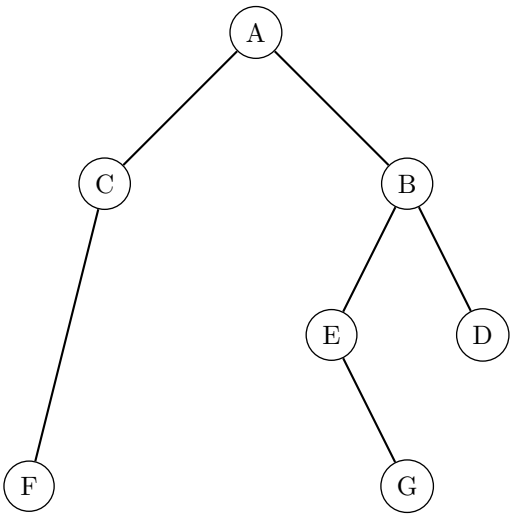
CAB203
Tutorial 8

1 Trees

1. For each rooted tree with root A, give determine the ancestors, descendants, parent, and children of the vertices C and D. For each tree give the set of leaves.

   (a)

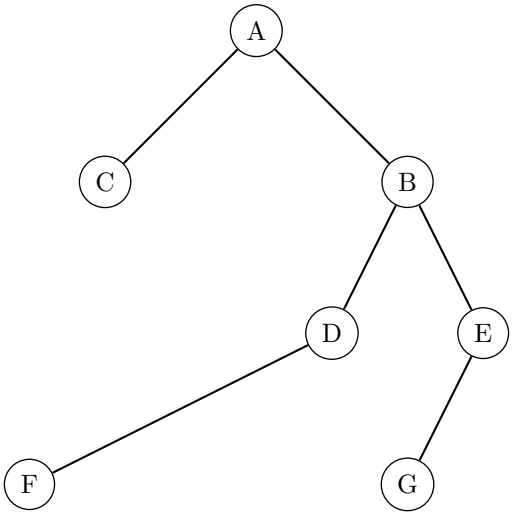


Solution:

	Parent	Ancestors	Children	Descendents
C	A	$\{A\}$	$\{F\}$	$\{F\}$
D	B	$\{A, B\}$	$\emptyset$	$\emptyset$

Leaves are $\{F, G, D\}$.

   (b)



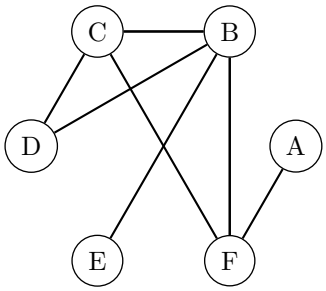
Solution:

	Parent	Ancestors	Children	Descendents
C	A	$\{A\}$	$\emptyset$	$\emptyset$
D	B	$\{A, B\}$	$\{F\}$	$\{F\}$

Leaves are $\{C, F, G\}$.

2. For the following graphs, give a spanning tree with root A such that all paths from A to another vertex u in the tree are the minimal length.

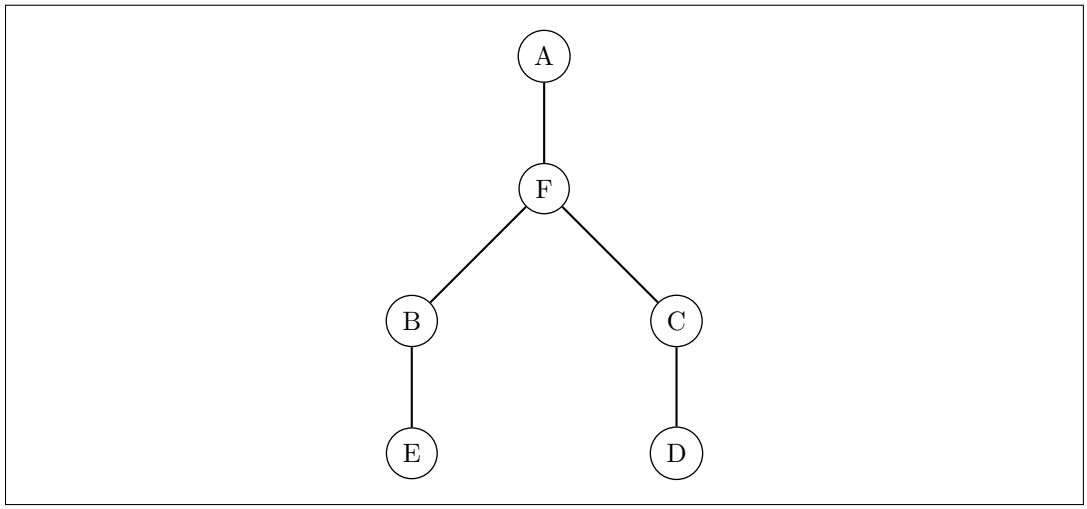
   (a)



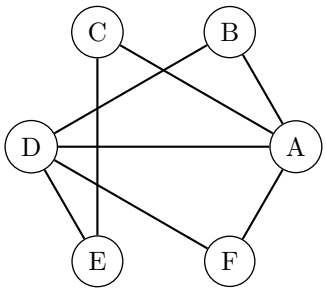
Solution:

$$\begin{aligned} G_0 &= \{A\} \\ G_1 &= \{F\} \\ G_2 &= \{B, C\} \\ G_3 &= \{D, E\} \end{aligned}$$

There is more than one tree that satisfies the conditions. One is



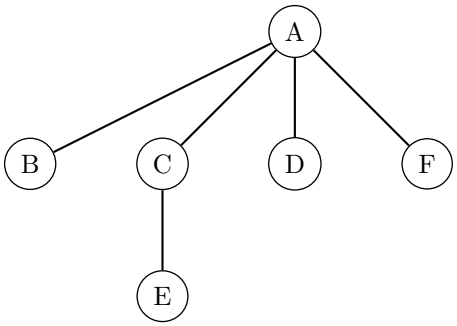
(b)



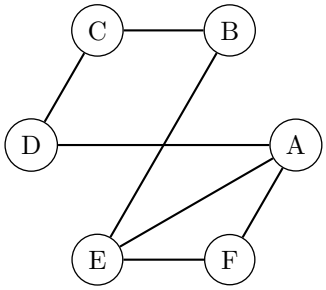
Solution:

$$\begin{aligned} G_0 &= \{A\} \\ G_1 &= \{B, C, D, F\} \\ G_2 &= \{E\} \end{aligned}$$

There is more than one tree that satisfies the conditions. One is



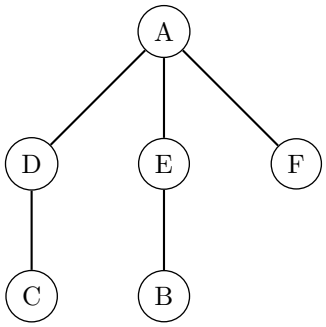
(c)



Solution:

$$\begin{aligned} G_0 &= \{A\} \\ G_1 &= \{D, E, F\} \\ G_2 &= \{C, B\} \end{aligned}$$

There is more than one tree that satisfies the conditions. One is



3. A power company needs to connect up several towns with power. The company needs to apply for permission for every power line that it installs (one permission per line connecting two towns). It would be impractical to construct a direct line between some towns due to challenging geography. The company wants to design a network that requires the least number of permissions, connects all the towns together, and avoids the impractical lines.

Solution: The vertices are the towns. Two towns are adjacent if it is practical to build a power line between them. Finding a practical connected network requiring the fewest permissions corresponds to finding a spanning tree. This could be done with either a depth-first or breadth-first traversal.

2 Case study: OSPF

Open Shortest Path First is a protocol used in networking for dynamically generating routing tables on medium sized networks with many routers. (By contrast, BPG is the protocol used between routers at the ISP level.) We will explore a simplified version of the problem that OSPF solves, which we will call the *routing table problem* (RTP).

Routers are assumed to already know about other routers that they are directly connected to, i.e. their neighbours. Each router broadcasts its list of neighbours to all other routers. Then each router reconstructs the graph of the entire network and constructs a *routing table* for itself, which tells the router, for each possible destination router, the first step on a shortest path to that destination.

1. Routers receive information from each other as lists of neighbours. Such a collection of lists is known as an *adjacency list* and it is common for graphs to be given in such a format (which is more efficient for certain applications.) How should the adjacency list for all routers be modelled:

   (a) using mathematical notation?

   Solution: An adjacency list gives a relationship between a vertex and its neighbours. The neighbours (i.e. the neighbourhood) are most reasonably represented as a set of vertices. It is reasonable to model this as a relation $A_G \subseteq V \times P(V)$ (where $P(V)$ is the power set of V). The pairs in A_G would be (u, N_u) where N_u is the neighbourhood of u.

   As an aside, since each vertex has exactly one neighbourhood, this relation is in fact a function, and the function is just $N_G(u)$ which we have studied before!

   (b) in Python?

   Solution: The adjacency list forms, as noted above, a function. In Python it is most reasonable to represent this as a dictionary, which are often used as containers for these kinds of functions composed of discrete information. The keys are vertices and the values are neighbourhoods. An example would be:

```
neighbours = {
    "A": { "B", "D" },
    "B": { "A", "C", "D" },
    "C": { "B", "D", "E" },
    "D": { "A", "B", "C" },
    "E": { "C" }
}
```

2. Given an adjacency list in the format given above, how can we find the vertex and edge sets of the graph?

   (a) using mathematical notation?

   Solution: Given the set of pairs $A_G = \{(u, N_u) : u \in V\}$ we can calculate the vertex set as:

$$V = \{u : (u, N_u) \in A\}$$

   The edge set is a bit trickier, but we essentially want to put all the contents of each neighbourhood into it. We can do this using a big union.

$$E = \bigcup_{(u,N_u)\in A} \{(u, v) : v \in N_u\}$$

   Another way is to use a set comprehension where we include an edge (u, v) if we can find (w, N) in the adjacency list where $u = w$ and $v \in N$.

$$E = \{(u, v) \in V^2 : \exists (w, N) \in A_G \; w = u \wedge v \in N\}$$

   (b) using Python?

   Solution: For the vertex set we can easily get the set of keys from the dictionary. We need to convert it to a set, though, because Python actually gives an iterator. For the edge set Python set comprehensions are a bit easier to work with than the big union. The Python combination of set comprehensions with replacements makes it a bit easier than using strict mathematical notation.

```
def adjacencyListToGraph(adjacencyList):
    V = set(adjacencyList.keys())
    E = { (u, v) for u, Nu in adjacencyList.items() for v in Nu}
    return V, E
```

3. Write a short Python function that takes a graph (V, E) and vertices u and v and returns the *first* vertex (after u) on a shortest path from u to v in the graph. We will call this vertex the *first hop*. You may make calls to previous functions in the unit. What should you do if $u = v$?

Solution: We can solve this by first solving an SPP (using solveSpp from the lecture). Once we have the path, we take the second vertex from it, i.e. the first vertex that comes after the start vertex. If the start is the end, then there is nothing to do! We can just return `None` to indicate that no routing is required.

```
def firstHop(V, E, start, end):
    path = solveSpp(V, E, start, end)
    return path[1] if len(path) > 1 else None
```

4. Write a short Python function that takes a graph (V, E) and starting vertex u returns a routing table which gives the first vertex for each destination vertex v. What form should the return take?

Solution: The routing table associates a destination vertex with a first hop. This could be a list of pairs, but we are most likely to want to look up by destination vertex, so we can code this in Python as a dictionary. A simple dictionary comprehension works. What about the vertex u? We could either not add it to the dictionary, or add it with `None` as the first hop to indicate that no routing is necessary. The latter choice is closer to what be used in an actual routing table, which would give an interface to route to for local traffic rather than a next hop. It's also easier to program!

```
def routingTable(V, E, u):
    return {v: firstHop(V, E, u, v) for v in V }
```

5. Combine the above to create a function which solves the RTP: given an adjacency list and a starting vertex, give a routing table that gives the first hop for any destination vertex.

Solution: Here is the complete code, which relies on `graphs.py` which is available on Blackboard.

```
def solveRTP(adjacencyList, u):
    V, E = adjacencyListToGraph(adjacencyList)
    return routingTable(V, E, u)
```

Solution:

```
from graphs import *

def adjacencyListToGraph(adjacencyList):
```

```
    V = set(adjacencyList.keys())
    E = { (u, v) for u, Nu in adjacencyList.items() for v in Nu}
    return V, E

def firstHop(V, E, start, end):
    path = solveSpp(V, E, start, end)
    return path[1] if len(path) > 1 else None

def routingTable(V, E, u):
    return {v: firstHop(V, E, u, v) for v in V }

def solveRTP(adjacencyList, u):
    V, E = adjacencyListToGraph(adjacencyList)
    return routingTable(V, E, u)

if __name__ == "__main__":
    neighbours = {
        "A": { "B", "D" },
        "B": { "A", "C", "D" },
        "C": { "B", "D", "E" },
        "D": { "A", "B", "C" },
        "E": { "C" }
    }
    print(solveRTP(neighbours, "A"))
```