Graphs (Undirected)

Defn. **Undirected Graph**

As in Piff " Discrete Mathematics" we take a graph to mean a 'simple' graph, i.e. the graph has at most one edge between vertices and there are no loops.
A graph G = (V,E) consists of a set V of vertices and a set E of edges. Each edge is an unordered pair, a 2-element set.

e.g. **V = {1,2,3,4,5,6}** **G = { {1,2}, { 2,3}, {4,5} }**

A graph may or may not be connected, i.e. a graph may consist of more than one connected component.

Adjacency Matrix:

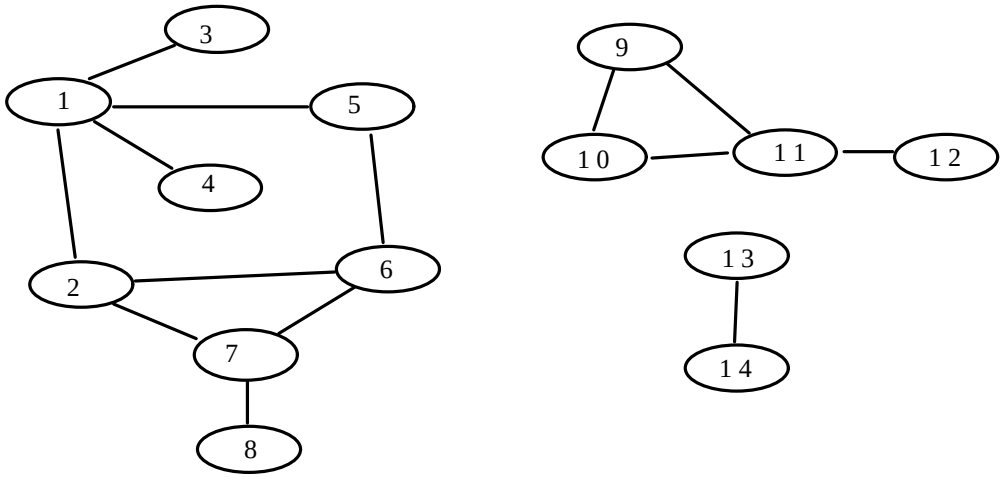
We can implement a graph as a matrix,
G : ARRAY2[BOOLEAN],
where we have 2 entries G(i,j) and G(j,i) are set to true for the edge {i,j},
i.e. G.item(i,j) $\wedge$ G.item(j,i) $\equiv$ there is an edge from i to j.

Graph Traversal

Depth First:
We start with a node and traverse along its descendants before going to next sibling.

Breadth First:
We traverse the graph 'level by level':
i.e. the parent, the children, the grandchildren etc.

Consider, as an example, the (disconnected) graph



A possible Depth First traversal would output: 1, 2, 6, 5, 7, 8, 3, 4, 9, 10, 11, 12, 13, 14

A Breadth First traversal could be: 1, 5, 4, 3, 2, 6, 7, 8, 9, 10, 11, 12, 13, 14

The traversals are not unique as the 'neighbours' of a nodes may be in any order.

Depth_First algorithm

Assume, at first, that the graph G is connected. In traversing a graph we 'visit' each node and, in general, we may then 'process' that node. For example, we may be traversing the graph looking for a node with some property. In our case, 'processing' a node just prints the 'information' at the node which is just the label in our examples.

```
Visit(j:Vertex) is—visit j and its descendants
        do
                'visit' / 'process' vertex j
                Mark j as visited
                from
                until
                        no more children of j
                loop
                        choose a (next) child k of j
                        if not visited(k) then
                                Visit(k)
                        -- recursively visit k & descendants
                        end
                end
        end—Visit
```

In Eiffel:

```
DFT_Visit(j:INTEGER) is
        local
                k :INTEGER
        do
                io.put_integer(j) -- print vertex
                V.put(True,j)
                from
                        k :=1
                until
                        k > G.height
                loop
                         if G.item(j,k) and not v.item(k) then
                                        DFT_Visit(k)
                        end
                         k := k+1
                end
        end—DFT_Visit
```

The graph G is represented as an Adjacency Matrix.

This routine assumes the graph is connected. If the graph is not connected we use Visit on each connected component.

```
Depth_First is
        local
                i : INTEGER
        do
                !!v.make(1,13)
                io.put_string("%N Depth First traversal is: %N")
                from
                        i := 1
                until
                        i > v.count
                loop
                        if not V.item(i) then
                                DFT_Visit(i)
                        end
                        i := i+1
                end
        end—Depth_first
```

Getting graph from Input:

The graph can be represented as edge pairs, one pair is sufficient for 'double-entry' into matrix.

1 2 1 3 1 4 1 5
2 6 2 7
5 6
6 7
7 8
etc

Input Routine—Read_Graph

```
Read_Graph is
        local
          i,j : INTEGER
        do
          from
                !!G.make(13,13) --default value, False
                io.read_integer
          until
                        io.last_integer = 0 -- using 0 for end of input
          loop
                        i := io.last_integer
                        io.read_integer
                        j := io.last_integer
                        G.put(True,i,j)
                        G.put(True,j,i)
                        io.read_integer
          end
        end—Read_Graph
```

Printing out Graph:

Sample output:

Adjacent nodes of 1
2 3 4 5
Adjacent nodes of 2
1 6 7
etc.

```
Print_Graph is
        local
                i,j : INTEGER
        do
                from
                        i := 1
                until
                        i > G.height
                loop
                        io.put_string("%N Adjacent nodes of  ")
                        io.put_integer(i)
                        io.new_line
                        from
                                j := 1
                        until
                                j > G.width
                        loop
                                if G.item(i,j) then
                                        io.put_integer(j)
                                        io.putchar(' ')
                                end
                                j := j+1
                        end
                        i := i+1
                end
        end—Print_Graph
```

Connected Components of a Graph

The Depth_First procedure above can be adapted to find or count the connected components in a graph.

```
Components is
        local
                i, k: INTEGER
        do
                !!v.make(1,13)
                io.put_string("%NConnected Components: %N")
                from
                        i := 1
                until
                        i > v.count
                loop
                        if not V.item(i) then
                                k := k+1
                                io.put_string("%N Component Num: ")
                                io.put_integer(k)
                                io.put_string("consists of")
                                Visit(i)
                                io.new_line
                        end
                        i := i+1
                end
        end—Components
```

Breadth First Traversal

In Depth First Traversal we used an implicit Stack in the recursion to stack a 'child' of a vertex and the descendants of a 'child' were considered before a 'sibling' or next 'child' was considered.

In Breadth First Traversal an explicit Queue is used so that each child's descendants are queued and due to the FIFO—First In First Out—action of a queue, the children are processed before the descendants.

The class QUEUE has among it features: (See DISPENSER cluster in ISE Eiffel)

```
put(x:G) -- put x to the end of the queue
item : G  -- the item at the front of the queue
remove     -- Remove  (front) item from queue
count : INTEGER -- #items in the queue
empty : BOOLEAN—count = 0
```

Recursive Breadth First Traverse.

To BFT (Breadth First Traverse) from a vertex V, we process V and then queue the 'children' of V so that they in turn can be BFT'd
If the graph is connected we have the pseudo-code

```
BFT (v : VERTEX) is
do
    Remove v from front of Q and process it
    For each 'child' of v
        Mark 'child' as visited (if not already)
        Add 'child' to Q
    end
    For each item, it, in Q
        BFT(it)
    end
end—BFT
```

Initially, the Q will contain the 'first' item in the graph.
In effect, the Q will contain, in order, the vertex v, the 'children' of v, the 'granchildren'etc
The graph G is stored as an adjacency Matrix.
As in Depth First, we BFT_Visit each connected component in turn.

The main routine, Breadth_First, calls BFT_Visit for each connected component. Also the queue, Q, is initialised in Breadth_First, which BFT_Visits each connected component.

```
    Breadth_First is
        local
            i : INTEGER
        do
            io.put_string("%N Breadth First traversal is: %N")
            !!v.make(1,size)
            !!Q
            from
                i := 1
            until
                i > size
            loop
                if not v.item(i) then
                    v.put(True,i)  -- mark vertex
                    Q.put(i)
                    BFT_Visit(i)
                end
                i := i+1
            end
            io.new_line
        end—Breadth_First
-----------------------------------------------------------------
    BFT_Visit(j:INTEGER) is
        require
            Non_Empty_Q:    not Q.Empty
        local
            k :INTEGER
        do
            Q.remove  -- remove j from Q
            io.put_integer(j)  -- process vertex
            io.putchar(' ')
            from
                k := 1
            until
                k > size
            loop
                if G.item(j,k) and not v.item(k) then
                    v.put(True,k)     -- mark vertex
                    Q.put(k)    -- Add child of j to Q
                end
                k := k+1
            end
            from
            until
                Q.empty
            loop
                BFT_Visit(Q.item)
            end
        end—BFT_Visit
```

Non-Recursive (Iterative) Breadth First Traverse

In the iterative version, the main routine, Breadth_First, is almost exactly as in the Recursive case, except that BFT_Visit has no argument. BFT_Visit, in effect, has Q as an argument.

```
Breadth_First is
        local
                i : INTEGER
        do
          io.put_string("%N Breadth First traversal is: %N")
          !!v.make(1,size)
          !!Q
          from
                i := 1
          until
                i > size
          loop
                if not v.item(i) then
                        v.put(True,i)  -- mark vertex
                        Q.put(i)
                        BFT_Visit
                end
                i := i+1
          end
          io.new_line
        end—Breadth_First

BFT_Visit is
        require
                Non_Empty_Q: not Q.empty
        local
                i,j :INTEGER
        do
                from
                until
                        Q.empty
                loop
                        i := Q.item
                        io.put_integer(i)
                        io.putchar(' ')
                        Q.remove
                        from
                                j := 1
                        until
                                j > size
                         loop
                                if G.item(i,j) and not v.item(j) then
                                        v.put(True,j)  -- mark vertex
                                        Q.put(j)
                                end
                                j := j+1
                        end—inner
                end  -- outer
        end—BFT_Visit
```

```
class ITERBRTH  -- Iterative Breadth First Traverse
creation
        make
feature
        G : ARRAY2[BOOLEAN]
        V : ARRAY[BOOLEAN]
        Q : QUEUE[INTEGER]
        size : INTEGER

        make is
        do
                Read_Graph;
                Print_Graph
                Breadth_First
        end—make

        Read_Graph is
                local
                        i,j : INTEGER
                do
                        from
                                size := 14
                                !!G.make(size, size)
                                io.read_integer
                        until
                                io.last_integer = 0
                        loop
                                i := io.last_integer
                                io.read_integer;
                                j := io.last_integer
                                G.put(True,i,j) ;
                                G.put(True,j,i)
                                io.read_integer
                        end
                end—Read_Graph
```

```
Breadth_First is
        local
                i : INTEGER
        do
          io.put_string("%N Breadth First traversal is: %N")
          !!v.make(1,size)
          !!Q.make
          from
                i := 1
          until
                i > size
          loop
                if not v.item(i) then
                        v.put(True,i)  -- mark vertex
                        Q.add(i)
                        BFT_Visit
                end
                i := i+1
          end
          io.new_line
        end—Breadth_First

BFT_Visit is
        require
                Non_Empty_Q: not Q.empty
        local
                i,j :INTEGER
        do
                from
                until
                        Q.empty
                loop
                         i := Q.item
                         io.put_int(i)
                         io.put_char(' ')
                         Q.remove
                         from
                                 j := 1
                        until
                                j > size
                        loop
                                if G.item(i,j) and not v.item(j) then
                                        v.put(True,j)  -- mark vertex
                                        Q.add(j)
                                end
                                 j := j+1
                        end—inner
                end  -- outer
end—BFT_Visit
```

```
Print_Graph is
        local
                i,j : INTEGER
        do
                from
                        i := 1
                until
                        i > G.height
                 loop
                        io.put_string("%N Adjacent nodes of  ")
                        io.put_integer(i)
                        io.new_line
                        from
                                j := 1
                        until
                                j > G.width
                        loop
                                if G.item(i,j) then
                                        io.put_integer(j)
                                        io.put_character(' ')
                                end
                                j := j+1
                        end
                        i := i+1
                end
        end—Print_Graph

end—ITERBRTH
```