# Assignment 2

Madiba Hudson-Quansah

# Part A

## Question 1

A network topology specifies how computers, printers, and other devices are connected over a network. You are given a boolean matrix $A\left[0 \dots n-1, 0 \dots n-1\right]$, where $n > 3$, which is supposed to be the adjacency matrix of a graph modeling a network with one of these topologies.
Your task is to determine which of these three topologies, if any, the matrix represents. Design a brute-force algorithm for this task and indicate its time efficiency class.

***Solution:***

**Algorithm 1** DetermineTopology $(A, n)$

▷

▷ Determine if the network topology of a given adjacency matrix $A$ is or isn't any of the three mentioned topologies
▷ Input: An boolean adjacency matrix $A$ of size $n \times n$
▷ Output: 1 if star, 2 if mesh, 3 if ring and -1 if none

```
isStar ← T
isRing ← T
isMesh ← T
mI ← −1
allTCount ← 0
oneTCount ← 0

for i ← 0 to n − 1 do
    tCount ← 0
    for j ← 0 to n − 1 do
        if i == j then continue
        end if

        if A_ij == T then
            tCount ← tCount + 1
        end if
    end for

    if tCount == 1 then
        oneTCount ← oneTCount + 1
    end if

    if tCount ≠ 2 then
        isRing ← F
    end if

    if tCount == n − 1 then
        if mI == −1 then
            mI ← i
        else
            isStar ← F
        end if
        allTCount ← allTCount + 1
    end if
end for
```

**Algorithm 2** DetermineTopology (Continued) $(A, n)$

```
if allTCount ≠ m then
    isMesh ← F
end if

if mI == −1 then
    isStar ← F
end if

if oneTCount ≠ m − 1 then
    isStar ← F
end if

if isStar then
    return 1
else if isMesh then
    return 2
else if isRing then
    return 3
else
    return -1
end if
```

**Question 2**

"Word find" (or "Word search") puzzles ask the player to find each of a given set of words in a square table filled with single letters. A word can read horizontally (left or right), vertically (up or down), or along a 45 degree diagonal (in any of the four directions) formed by consecutively adjacent cells of the table; it may wrap around the table's boundaries, but it must read in the same direction with no zigzagging. The same cell of the table may be used in different words, but, in a given word, the same cell may be used no more than once. Design a brute-force algorithm for solving this puzzle.

***Solution:***

**Algorithm 3** WordFind $(A, n, \text{targetWords}, m)$

▷

▷ Finds the indices and directions of a set of words in a word find puzzle

▷ Input: The word square $A$ with dimensions $n \times n$, A set of target words of length $m$

▷ Output: A list containing the word, the start index and the found direction of each of the found words in the word square in the format [word, start, direction]

```
function checkHR( word, w, A, i, j)        ▷ Function to check if a word can be found horizontally right
    ▷ word - the word to find, w - the length of the word, A - the word square, i - the row index, j - the column index
    idx ← j
    count ← 0
    while count < w do
        if word[count] ≠ A[i][idx] then
            return F
        end if
        idx ← (idx + 1) mod n
        count ← count + 1
    end while
    return T
end function

function checkHL( word, w, A, i, j)        ▷ Function to check if a word can be found horizontally left
    ▷ word - the word to find, w - the length of the word, A - the word square, i - the row index, j - the column index
    idx ← j
    count ← 0
    while count < w do
        if word[count] ≠ A[i][idx] then
            return F
        end if
        idx ← (idx − 1) mod n
        count ← count + 1
    end while
    return T
end function
```

**Algorithm 3** WordFind (Continued) ($A, n$, targetWords, $m$)

```
function checkVD( word, w, A, i, j)        ▷ Function to check if a word can be found vertically down
    ▷ word - the word to find, w - the length of the word, A - the word square, i - the row index, j - the column index
    idx ← i
    count ← 0
    while count < w do
        if word[count] ≠ A[idx][j] then
            return F
        end if
        idx ← (idx + 1) mod n
        count ← count + 1
    end while
    return T
end function

function checkVU( word, w, A, i, j)        ▷ Function to check if a word can be found vertically up
    ▷ word - the word to find, w - the length of the word, A - the word square, i - the row index, j - the column index
    idx ← j
    count ← 0
    while count < w do
        if word[count] ≠ A[idx][j] then
            return F
        end if
        idx ← (idx − 1) mod n
        count ← count + 1
    end while
    return T
end function

function checkDTR( word, w, A, i, j)        ▷ Function to check if a word can be found diagonally top right
    ▷ word - the word to find, w - the length of the word, A - the word square, i - the row index, j - the column index
    idxR ← i
    idxC ← j
    count ← 0
    while count < w do
        if word[count] ≠ A[idxR][idxC] then
            return F
        end if
        idxR ← (idxR − 1) mod n
        idxC ← (idxC + 1) mod n
        count ← count + 1
    end while
    return T
end function
```

**Algorithm 4** WordFind (Continued) $(A, n, \text{targetWords}, m)$

```
function checkDTL( word, w, A, i, j)    ▹ Function to check if a word can be found diagonally top left
    ▹ word - the word to find, w - the length of the word, A - the word square, i - the row index, j - the column index
    idxR ← i
    idxC ← j
    count ← 0
    while count < w do
        if word[count] ≠ A[idxR][idxC] then
            return F
        end if
        idxR ← (idxR − 1) mod n    ▹ Move up
        idxC ← (idxC − 1) mod n    ▹ Move left
        count ← count + 1
    end while
    return T
end function

function checkDBL( word, w, A, i, j)    ▹ Function to check if a word can be found diagonally bottom left
    ▹ word - the word to find, w - the length of the word, A - the word square, i - the row index, j - the column index
    idxR ← i
    idxC ← j
    count ← 0
    while count < w do
        if word[count] ≠ A[idxR][idxC] then
            return F
        end if
        idxR ← (idxR + 1) mod n    ▹ Move down
        idxC ← (idxC − 1) mod n    ▹ Move left
        count ← count + 1
    end while
    return T
end function

function checkDBR( word, w, A, i, j)    ▹ Function to check if a word can be found diagonally bottom right
    ▹ word - the word to find, w - the length of the word, A - the word square, i - the row index, j - the column index
    idxR ← i
    idxC ← j
    count ← 0
    while count < w do
        if word[count] ≠ A[idxR][idxC] then
            return F
        end if
        idxR ← (idxR + 1) mod n    ▹ Move down
        idxC ← (idxC + 1) mod n    ▹ Move right
        count ← count + 1
    end while
    return T
end function

foundIds ← []
wIdx ← 0
found ← F
```

**Algorithm 5** WordFind (Continued) ($A, n$, targetWords, $m$)

```
for idx ← 0 to m − 1 do
    word ← targetWords[idx]
    w ← length of word
    found ← F
    for i ← 0 to n − 1 do
        if found then            ▷ Assuming a word only appears once we break out of the word loop once it is found
            break
        end if
        for j ← 0 to n − 1 do
            if checkHR( word, w, A, i, j) then
                foundIds[wIdx] ← [word, [i, j], "HR"]
                wIdx ← wIdx + 1
                found ← T
                break
            else if checkHL( word, w, A, i, j) then
                foundIds[wIdx] ← [word, [i, j], "HL"]
                wIdx ← wIdx + 1
                found ← T
                break
            else if checkVD( word, w, A, i, j) then
                foundIds[wIdx] ← [word, [i, j], "VD"]
                wIdx ← wIdx + 1
                found ← T
                break
            else if checkVU( word, w, A, i, j) then
                foundIds[wIdx] ← [word, [i, j], "VU"]
                wIdx ← wIdx + 1
                found ← T
                break
            else if checkDTR( word, w, A, i, j) then
                foundIds[wIdx] ← [word, [i, j], "TR"]
                wIdx ← wIdx + 1
                found ← T
                break
            else if checkDBR( word, w, A, i, j) then
                foundIds[wIdx] ← [word, [i, j], "BR"]
                wIdx ← wIdx + 1
                found ← T
                break
            else if checkDTL( word, w, A, i, j) then
                foundIds[wIdx] ← [word, [i, j], "TL"]
                wIdx ← wIdx + 1
                found ← T
                break
            else if checkDBL( word, w, A, i, j) then
                foundIds[wIdx] ← [word, [i, j], "BL"]
                wIdx ← wIdx + 1
                found ← T
                break
            end if
        end for
    end for
end for

return foundIds
```

**Question 3**

1. Apply the DFS-based algorithm to solve the topological sorting problem for the following directed acyclic graph.
2. Apply the source-removal algorithm to solve the topological sorting problem for the following directed acyclic graph.

***Solution:***

1. The DFS-based algorithm involves performing a DFS on all the nodes of a graph and reversing the order each node was visited to obtain a topological sort. This can be done by observing the stack frames made by the recursive algorithm and assigning numbers to each of the nodes based on the order they where pushed on to the stack and popped of the stack. Assuming the nodes are arranged in alphabetical order and prioritizing alphabetical order, the stack order is:

$$a\,1,6$$
$$b\,2,4$$
$$e\,3,1$$
$$g\,4,3$$
$$f\,5,2$$
$$c\,6,5$$
$$d\,7,7$$

   And the resulting topological sort is:

$$d, a, c, b, g, f, e$$

2. Using the source-removal algorithm, we remove nodes with an in-degree of 0 and add them to the beginning of the topological sort. But since this graph has a cycle from $e$ to $e$, after the first source removal of $f$, there are no other nodes with an in-degree of 0. Thus the graph has no topological sort and a partial topological sort is of just $f$.

**Question 4**

A digraph is called strongly connected if for any pair of two distinct vertices $u$ and $v$ there exists a directed path from $u$ to $v$ and a directed path from $v$ to $u$. In general, a digraph's vertices can be partitioned into disjoint maximal subsets of vertices that are mutually accessible via directed paths; these subsets are called strongly connected components of the digraph. There are two DFS-based algorithms for identifying strongly connected components. Here is the simpler (but somewhat less efficient) one of the two:

**Step 1** Perform a DFS traversal of the digraph given and number its vertices in the order they become dead ends.
**Step 2** Reverse the directions of all the edges of the digraph.
**Step 3** Perform a DFS traversal of the new digraph by starting (and, if necessary, restarting) the traversal at the highest numbered vertex among still unvisited vertices.

The strongly connected components are exactly the vertices of the DFS trees obtained during the last traversal.

1. Apply this algorithm to the following digraph to determine its strongly connected components.
2. What is the time efficiency class of this algorithm? Give separate answers for the adjacency matrix representation and adjacency list representation of an input digraph.
3. How many strongly connected components does a dag have? Justify your answer.

***Solution:***

1. Applying this algorithm first I start with the DFS traversal of the graph and number the vertices in the order they

become dead ends. The order is:

$$\begin{aligned} a&\ 4\\ b&\ 3\\ g&\ 2\\ f&\ 1\\ c&\ 8\\ d&\ 5\\ e&\ 7\\ h&\ 6 \end{aligned}$$

Next I reverse the directions of the edges and perform a DFS traversal starting at the highest numbered node $c$, the DFS trees obtained are:

$$\begin{aligned} c \rightarrow h \rightarrow e\\ d\\ a \rightarrow f \rightarrow g \rightarrow b \end{aligned}$$

Therefore the strongly connected components are: $\{c, h, e\}$, $\{d\}$ and $\{a, f, g, b\}$.

2. **Adjacency Martix** - The time efficiency class of this algorithm for the adjacency matrix representation is $O\left(n^2\right)$, where $n$ is the number of vertices in the graph.

   This is beacause the first step of the algorithm involves performing a DFS traversal of the graph which takes $O\left(n^2\right)$ to check all the possible edges in the graph.

   The second step involves reversing the directions of all the edges which also takes $O\left(n^2\right)$ to check and reverse all existing edges.

   The third step involves another a DFS traversal of the new graph which also takes $O\left(n^2\right)$.

   In total this is $O\left(n^2\right) + O\left(n^2\right) + O\left(n^2\right)$ which is $O\left(n^2\right)$.

   **Adjacency List** - The time efficiency class of this algorithm for the adjacency list representation is $O(n+m)$, where $n$ is the number of vertices in the graph and $m$ is the number of edges in the graph.

   This is because the first step of the algorithm does a DFS traversal of the graph which in a adjacency list represented graph takes $O(n+m)$ as it only check each edge once using this representation.

   The second step involves reversing the directions of all the edges which takes $O(m)$ time where $m$ is the number of edges in the graph as the algorithm just needs to iterate through the list of edges and reverse them.

   Finally the third step invloves another DFS traversal of the new graph which also takes $O(n+m)$ time.

   In total this is $O(n+m) + O(m) + O(n+m)$ which is $O(n+m)$.

3. A DAG can only have exactly $n$ strongly connected components. This is because a DAG has no cycles thus it is impossible to have a directed path from $v$ to $u$ and another from $u$ to $v$, i.e. a cycle. Thus each strongly connected component is a single node and there are $n$ nodes in the graph.