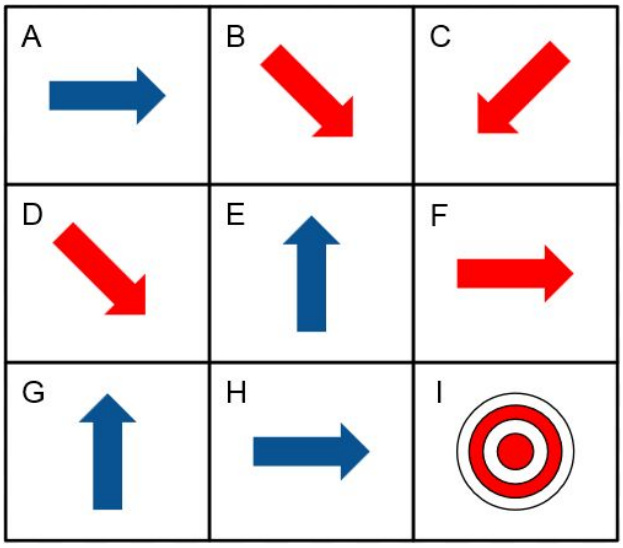
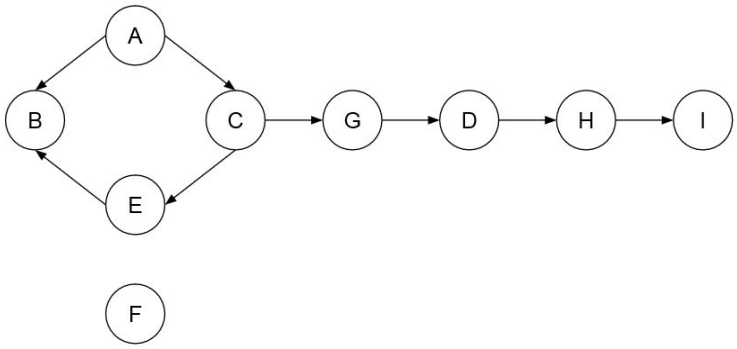
John Gangemi
U6871-4612
COT4400 Analysis of Algorithms
Project 2 Report

Modeling The Problem

For the “Apollo and Diana” maze problem I find it appropriate to model the problem input (detailed in Section 3.1 of the project guidelines) with a directed graph. Upon reviewing a provided example problem it was apparent that vertices are arrows/bullseye, edges are valid connections between different colored arrows, and paths are a series of edges. Disconnection in the graph can occur when an arrow has no adjacent neighbor, consider the following maze...



Let us assume a vertex list would consist of {A, B, C, D, E, F, G, H, I} and an edge list would consist of {(A-B), (A-C), (C-E), (C-G), (D-H), (E-B), (G-D), (H-I)}. The graph representation of the maze would resemble the following...



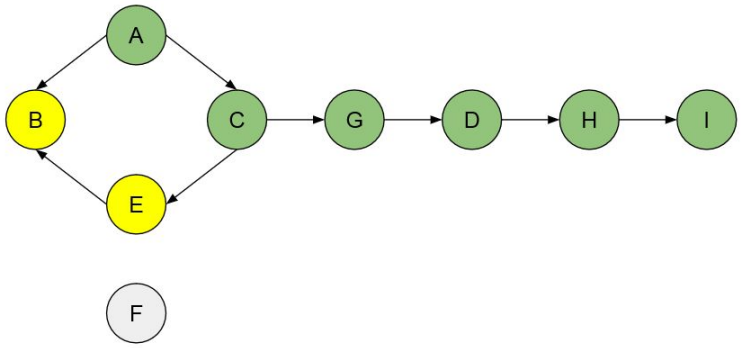
Therefore, one can clearly notice the disconnection of vertex F from the rest of the graph due to its non-adjacency.

To solve the problem, represented as a graph, I have chosen the depth-first search algorithm to perform graph traversal. However, it has been slightly modified to adhere to the specific conditions required of the “Apollo and Diana” maze such that if a solution exists it will always be a path from the first vertex to the last vertex. Also, it should be noted that an iterative approach to DFS was preferred over a recursive implementation.

Input: Graph *digraph*
Output: Stack *path*
Algorithm: ModDFS
vertexList = get list of all vertices from *digraph*

```
set first vertex in vertexList as discovered
set first vertex as current vertex
add first vertex to path

while (path is not empty)
{
  set flag discoveredVertex to false
  neighbors = get list of all neighbors for current vertex
  iterate through neighbors
    if (neighbor is undiscovered)
    {
      set neighbor as current vertex
      set current vertex as discovered
      add current vertex to path
      set flag discoveredVertex  to true
      stop iterating through neighbors
    {

  if (discoveredVertex is false)
  {
    pop current vertex from path
    set top vertex of path as current vertex
  }

  if (current vertex is the last vertex in vertexList)
    exit while loop
}

return path
```

To prove correctness of the modified DFS algorithm it must be re-stated that the solution to the "Apollo and Diana" maze will always be a **distinct path** from the **first** vertex to the **last** vertex of a graph. Therefore, it's safe to assume the first vertex will always have one or more K adjacent vertices. Thus, any K + 1 vertex, that is a vertex on the path, will itself have one or more adjacent vertices by definition of the "Apollo and Diana" maze. The K + n vertex represents the last vertex of the path and signifies the solution has been found.
If we re-examine the graph...



We see that vertex A is K, vertex C is K + 1, vertex G is K + 2, vertex D is K + 3, vertex H is K + 4, and vertex I is K + n.

Furthermore, the inclusion of a *discoveredVertex* flag avoids keeping vertices with no neighbors or already discovered neighbors (would form cycle between directed vertices) in the stack. The is evident by the exclusion of vertex B and E from the solution path. When vertex E is first examined its only neighbor has already been discovered, therefore *discoveredVertex* evaluates to false and vertex E is removed from the stack.

Analysing the time complexity requires a procedural deconstruction of the pseudo-code into blocks. Let us assume that getting a list of all vertices from the graph is O(n) where n is the number of vertices. We can safely proclaim the next three lines as taking constant time, or O(1). We will skip the while loop condition for now and examine the inner structures. Setting the *discoveredVertex* flag is O(1), getting a list of all neighbors is O(degree(V)) where degree(V) is the number of vertices adjacent to V. Iterating through all neighbors is O(degree(V)), as the inner instructions are O(1) and thus add no complexity. All code remaining in the while loop can be assumed to take O(1) time.

As stated earlier, the while loop allows for all connected vertices to be visited therefore in a graph of all connected vertices the total number of iterations is equal to the total number of vertices plus the degree of each vertex.

The final time complexity equation for the modified depth-first search algorithm is **O(n * degree(V))**. The contributions to the final time complexity can be attributed to the while loop and its nested for loop.