Assignment 7

Course: COMPSCI 2C03
Date: 11/21/2025
Author: Sulagna Nandi
MacID: nandis
Student Number: 400575118
Professor: Dr. Vincent Maccio

P1.1. Every strongly connected graph is automatically semi-connected because strong connectivity guarantees that for every pair of nodes m and n, both $m \to n$ and $n \to m$ exist. Semi-connectedness only requires that at least one of the two paths exists. Since strong connectivity already provides both directions, it trivially satisfies the semi-connectedness condition.

P1.2. A connected graph that is not semi-connected can be constructed as follows. Consider nodes a, b, and c with directed edges $a \to b$ and $c \to b$. Ignoring the directions, the underlying undirected graph is connected, forming a simple path $a - b - c$. However, there is no path from a to c and no path from c to a. This violates semi-connectedness, which requires every pair to be comparable in at least one direction. Therefore, this graph is connected but not semi-connected.

P1.3. Algorithm for DAGs in $O(|N| + |E|)$

For a directed acyclic graph, we use the topological order property. A DAG is semi-connected if and only if every consecutive pair in some topological ordering is connected by a directed edge. Here is the explicit algorithm:

Algorithm 1 IsSemiConnectedDAG(G)

Require: A DAG $G = (N, E)$ represented as adjacency lists
Ensure: Returns TRUE if G is semi-connected, FALSE otherwise

```
Compute a topological ordering L = [v_1, v_2, ..., v_n] of G using Kahn's algorithm or DFS
for i = 1 to n − 1 do
    if v_{i+1} is not in the adjacency list of v_i then
        return FALSE
    end if
end for
return TRUE
```

This algorithm is correct because, in a topological order, a path from a later node back to an earlier node cannot exist. Therefore, for semi-connectedness, each consecutive pair must be reachable forward. Checking adjacency lists guarantees linear runtime $O(|N| + |E|)$.

P1.4. Algorithm for general directed graphs in $O(|N| + |E|)$

For a general directed graph, we first reduce the problem to the DAG case using strongly connected components. The graph is semi-connected if and only if its condensation DAG (with each SCC as a node) is semi-connected. The algorithm is:

Algorithm 2 IsSemiConnectedGeneral(G)

Require: A directed graph $G = (N, E)$ represented as adjacency lists
Ensure: Returns TRUE if G is semi-connected, FALSE otherwise

```
Compute the strongly connected components (SCCs) of G using Tarjan's or Kosaraju's algorithm
Build the condensation DAG G' = (N', E'), where each node represents one SCC and an edge exists from SCC C_i to C_j if any edge in G goes from a node in C_i to a node in C_j
Return IsSemiConnectedDAG(G') {Use algorithm from P1.3}
```

Correctness follows because nodes within each SCC are mutually reachable, so the only possible violations of semi-connectedness occur between SCCs. Each step runs in linear time with adjacency

lists: computing SCCs is $O(|N|+|E|)$, building the condensation DAG is $O(|N|+|E|)$, and running the DAG semi-connectedness test is $O(|N|+|E|)$.

P1.5. Representation and complexity considerations

All algorithms above assume adjacency-list representation, which allows iterating over outgoing edges in time proportional to the out-degree. This ensures $O(|N|+|E|)$ runtime for topological sort, SCC computation, and the semi-connectedness checks. If an adjacency matrix were used instead, enumerating neighbors of a node requires scanning a full row of length $|N|$, increasing the running time to $O(|N|^2)$, which is suboptimal for sparse graphs.

P2.1. Modeling the problem as a graph

We can represent the $m \times n$ game board as a directed graph $G = (V, E)$. Each cell of the board corresponds to a node in V, so there are $m \cdot n$ nodes. An edge $(u, v) \in E$ exists if a player can move from cell u to cell v in one turn according to the rules: the distance of a move is determined by the value in the starting cell, and the move can be in any of the four cardinal directions (up, down, left, right), wrapping around the edges of the board if necessary. All edges are unweighted because each move counts as one turn.

The problem we want to answer on this graph is: for a given player A starting at a particular cell s_A and the opponent starting at s_O, does there exist a strategy for A that guarantees reaching s_O in fewer moves than O can reach s_A, regardless of how O moves? In graph terms, this reduces to computing the minimum number of moves from s_A to all other nodes (a shortest-path problem in terms of number of moves) and comparing it to the opponent's shortest-path distances from s_O.

P2.2. Algorithm to determine a strong starting position

We can solve this using a Breadth-First Search (BFS) approach. Since each move has equal cost (one turn), BFS naturally computes the minimum number of moves from the starting cell to every other cell. We perform BFS for both players, then compare the distance to each relevant cell to determine whether player A can guarantee a win.

Algorithm 3 StrongStartingPosition(Board, s_A, s_O)

Require: Game board values $Board[1..m][1..n]$, starting cell for player A s_A, opponent starting cell s_O

Ensure: TRUE if player A has a strong starting position, FALSE otherwise

1: $dist_A \leftarrow$ BFS distances from s_A on the graph induced by Board
2: $dist_O \leftarrow$ BFS distances from s_O on the graph induced by Board
3: **if** $dist_A[s_O] < dist_O[s_A]$ **then**
4: **return** TRUE
5: **else**
6: **return** FALSE
7: **end if**

The BFS procedure constructs the adjacency information implicitly: for each cell, generate up to four edges according to its value and the board wrap-around rules. BFS ensures that the minimum number of moves to each reachable cell is computed efficiently.

P2.3. Correctness and efficiency

The algorithm is correct because BFS computes the minimum number of moves from a source to all reachable nodes in an unweighted graph. Since each move is equivalent to one turn, BFS

distances correspond exactly to the number of rounds required to reach any cell. Comparing $dist_A[s_O]$ to $dist_O[s_A]$ captures the definition of a strong starting position: A has a guaranteed win if and only if A can reach O's initial cell in fewer moves than O can reach A's initial cell.

The efficiency comes from the fact that BFS runs in $O(|V| + |E|)$ time. Here, $|V| = m \cdot n$, and $|E| \leq 4 \cdot m \cdot n$ because each cell has at most four outgoing edges. Therefore, the algorithm runs in $O(mn)$ time for any $m \times n$ board.

P2.4. Graph representation and complexity

We used an adjacency list representation implicitly in our BFS: for each cell, we compute its up to four neighbors dynamically instead of storing all edges explicitly. This keeps the space usage linear in the number of nodes. If we had used an adjacency matrix representation, we would have required $O((mn)^2)$ space and each BFS step would have needed $O(mn)$ time per node to scan potential neighbors, leading to a total time complexity of $O((mn)^2)$. Thus, adjacency lists (or generating neighbors on-the-fly) are far more efficient for this problem.