Reflection :

The shortest path problem finds the minimum distance between two nodes in a graph. Its solution depends on whether the edges are weighted.

For unweighted graphs, the shortest path is measured by the number of edges. Breadth-first search (BFS) solves this problem efficiently, with a running time proportional to the number of nodes and edges.

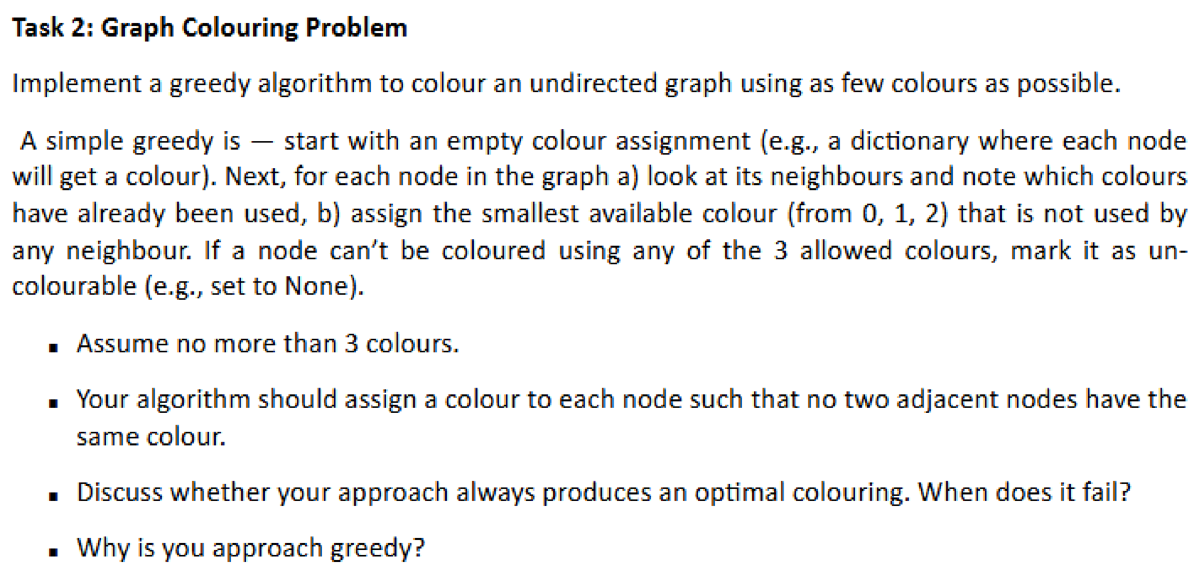
For weighted graphs, the shortest path is defined as the path with the smallest total weight.

Dijkstra's algorithm finds the shortest path from a source to all other nodes. It runs efficiently even for non-negative weights. Its basic form runs in $O(n^2)$, and can be improved using a heap. Dijkstra's algorithm cannot handle negative weights; any change in edge weights requires rerunning the algorithm.

The Bellman-Ford algorithm can also compute single-source shortest paths, but it can handle negative weights as long as there are no negative cycles. However, because it repeatedly relaxes all edges, it is slower, running in $O(n \times m)$.

Both methods embody the principle of dynamic programming, which solves problems by combining solutions to smaller subproblems.

For the all-pairs shortest path problem, the Floyd-Warshall algorithm is commonly used. This algorithm checks for each pair of nodes whether adding an intermediate node improves the path. Its runtime is $O(n^3)$. It is simpler to implement than running other algorithms multiple times and can handle negative weights, but not negative cycles.



Task 2: Graph Colouring Problem

Implement a greedy algorithm to colour an undirected graph using as few colours as possible.

A simple greedy is — start with an empty colour assignment (e.g., a dictionary where each node will get a colour). Next, for each node in the graph a) look at its neighbours and note which colours have already been used, b) assign the smallest available colour (from 0, 1, 2) that is not used by any neighbour. If a node can't be coloured using any of the 3 allowed colours, mark it as un-colourable (e.g., set to None).

- Assume no more than 3 colours.
- Your algorithm should assign a colour to each node such that no two adjacent nodes have the same colour.
- Discuss whether your approach always produces an optimal colouring. When does it fail?
- Why is you approach greedy?

A :

The general implementation of this method is as follows:

1. First, check which colors are already used by the points next to it.

2. Between 0, 1, and 2, pick the smallest color that its neighbors do not already use and assign it to that point.

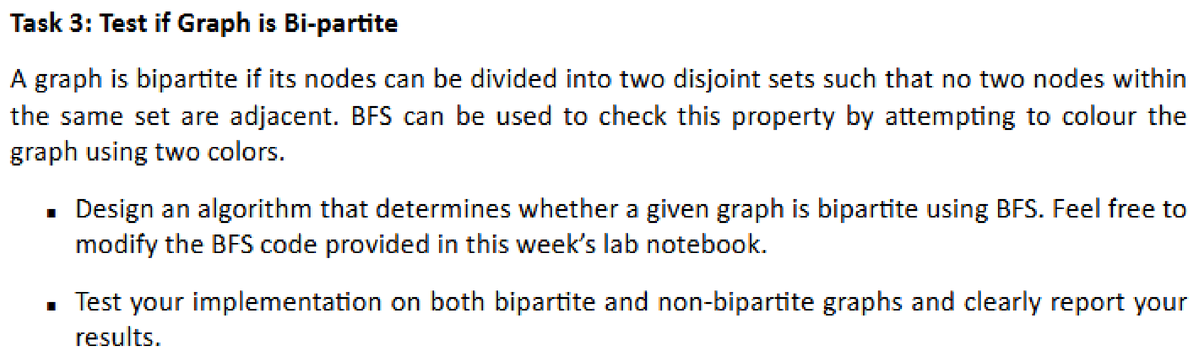
3. If its neighbors already use all three colors, mark the point as "uncolored."

Always produces an optimal coloring? When does it fail?

Not always. Changing the order in which points are checked can lead to completely different results. Sometimes, the entire image could be painted with just three colors, but a poor initial choice prevents further painting.

Why "greedy"?

Because each step focuses solely on the immediate present, it chooses the smallest available color for the current point and doesn't care if this will lead to errors later on.



Task 3: Test if Graph is Bi-partite

A graph is bipartite if its nodes can be divided into two disjoint sets such that no two nodes within the same set are adjacent. BFS can be used to check this property by attempting to colour the graph using two colors.

- Design an algorithm that determines whether a given graph is bipartite using BFS. Feel free to modify the BFS code provided in this week's lab notebook.
- Test your implementation on both bipartite and non-bipartite graphs and clearly report your results.

A:

Results

Testing on a bipartite graph:

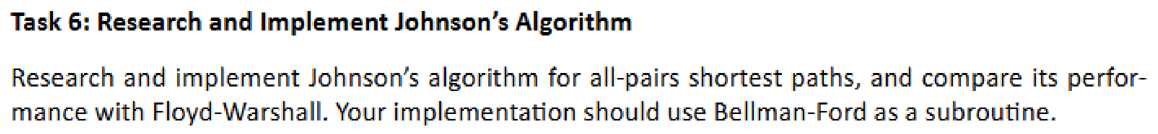
The algorithm successfully completes the coloring and determines that the graph is bipartite.

Example: The four-node chain structure 1-2-3-4 can be split into two sets: {1,3} and {2,4}.

Testing on a non-bipartite graph:

The algorithm detects a conflict and correctly reports that the graph is not bipartite.

Example: The three-node cycle 1-2-3-1 cannot be split into two sets.



Task 6: Research and Implement Johnson's Algorithm

Research and implement Johnson's algorithm for all-pairs shortest paths, and compare its performance with Floyd-Warshall. Your implementation should use Bellman-Ford as a subroutine.

A:

The Johnson algorithm is an all-source shortest path algorithm suitable for cases with negative-weight edges but no negative-weight cycles. Its main ideas are:

1. Introducing an auxiliary node: Add a new source node q to the graph, connecting it to all other nodes with an edge weight of 0;

2. Running the Bellman-Ford algorithm: Using q as the source, calculate the "potential energy" h(v) of each node, which is used to reweight the edges;

3. Reweighting edges: Modify the weight of each edge (u, v)

$w'(u, v) = \omega(u, v) + h(u) - h(v)$

Performance Comparison (with Floyd-Warshall)

Johnson's Algorithm

It has a complexity of approximately $O(V^2 \log V + VE)$ and performs better on sparse graphs ($E << V^2$). It uses Dijkstra's algorithm multiple times and is suitable for both sparse and large graphs.

The Floyd-Warshall Algorithm

It has a complexity of $O(V^3)$ and is suitable for small dense graphs. It is simple and straightforward, but less efficient on large graphs.

Resources:

1. https://medium.com/algorithm-solving/johnsons-algorithm-c461506fccae ; Aaron; Apr 26 2020; Johnson’s Algorithm
2. https://blog.csdn.net/myRealization/article/details/124190455 ; memcpy0; 15,04,2022;