Graph Management Exercises Report

Breadth-First Search, Depth-First Search, and Graph Coloring

Sara Sherif Daoud Saad

September 29, 2025

Contents

1 Breadth-First Search and Bipartite Graphs

1.1 Exercise 1: BFS Distance Computation

Problem: Using graph traversal algorithms, propose an algorithm that computes the number of edges between a given vertex and all other vertices.

Solution: I adjusted the BFS algorithm to find the shortest paths in unweighted graphs by exploring vertices level by level.

```python
# See exercise1_bfs_distances.py for complete implementation
def compute_distances_from_vertex(graph, start_vertex):
    distances = {}
    for vertex in graph:
        distances[vertex] = float('inf')

    distances[start_vertex] = 0
    queue = deque([start_vertex])

    while queue:
        current_vertex = queue.popleft()
        for neighbor in graph[current_vertex]:
            if distances[neighbor] == float('inf'):
                distances[neighbor] = distances[current_vertex] + 1
                queue.append(neighbor)

    return distances
```

Listing 1: BFS Distance Computation

Time Complexity: O(V + E) where V is the number of vertices and E is the number of edges.

Implementation: See `exercise1_bfs_distances.py` for complete implementation with custom Queue class.

1.2 Exercise 2: Odd Cycles Analysis

Problem: Given the following cycles with even and odd length (with the distances or depths from the grey vertex), what do you think about the case of graphs with an odd cycle (in number of edges)? Is this a characteristic property? State the general case.

Solution:

Observation from the graphs:

- A graph with only even cycles can be bipartite
- A graph with an odd cycle cannot be bipartite

Characteristic property:

- A graph is bipartite $\Leftrightarrow$ it contains no odd-length cycles

General case:

- Odd cycles break bipartiteness; even cycles do not

Mathematical Foundation:

- In bipartite graphs, all cycles must have even length
- If a graph contains an odd cycle, it cannot be bipartite
- This provides a necessary and sufficient condition for bipartiteness
- The presence of an odd cycle is a definitive proof that a graph is not bipartite

1.3 Exercise 3: Odd Cycle Detection Algorithm

Problem: Propose an algorithm that determines if a graph contains an odd cycle.

Solution: Use BFS with vertex coloring to detect odd cycles. If we find a back edge connecting two vertices of the same color, we have found an odd cycle.

```python
# See exercise3and5_odd_cycle_detection_bipartite.py for complete
    implementation
def has_odd_cycle(graph):
    color = {}
    for node in graph:
        if node not in color:
            queue = deque([node])
            color[node] = 0

            while queue:
                u = queue.popleft()
                for v in graph[u]:
                    if v not in color:
                        color[v] = 1 - color[u]
                        queue.append(v)
                    elif color[v] == color[u]:
                        return True
    return False
```

Listing 2: Odd Cycle Detection

Time Complexity: O(V + E) **Implementation:** See `exercise3and5_odd_cycle_detection_bip` for complete implementation with detailed comments and test cases.

1.4 Exercise 4: Bipartite Graphs and Odd Cycles

Problem: In a bipartite graph, can there be a cycle with an odd number of edges? Is this a characteristic property? Justify your answer.

Answer: No, bipartite graphs cannot contain odd cycles. This is indeed a characteristic property.

Property:

- Bipartite graphs cannot have odd cycles

Justification: Bipartite graphs divide vertices into two sets, all edges go across sets. If there were an odd cycle, you would need an edge connecting vertices in the same set, which is impossible.

Mathematical Foundation:

1. In a bipartite graph, vertices can be divided into two disjoint sets U and V
2. Every edge connects a vertex in U to a vertex in V
3. Any cycle must alternate between sets U and V
4. To return to the starting vertex, we need an even number of steps
5. Therefore, all cycles in bipartite graphs have even length

1.5 Exercise 5: Bipartite Graph Detection

Problem: Propose an algorithm that allows to determine if a graph is bipartite. Test your algorithm on the following graph. Is it bipartite? Justify your answer.

Solution: BFS-based bipartite detection algorithm.

This is the same BFS coloring algorithm. If you can color the graph with 2 colors without conflict, it's bipartite.

Result: If `has_odd_cycle` returns `True`, the graph is not bipartite. If `False`, it is bipartite.

Algorithm: The same BFS coloring algorithm from Exercise 3 can be used to detect bipartiteness.

2 Depth-First Search and 2-Colorable Graphs

2.1 Exercise 1: Link with Previous Exercise

Problem: What is the link with the previous exercise? Justify your answer.

Answer:

1) Link with the previous exercise

In the previous exercise, we studied BFS, odd cycles, and bipartite graphs. A graph is bipartite $\Leftrightarrow$ 2-colorable $\Leftrightarrow$ contains no odd cycle. So the question of 2-colorability is exactly the same as asking whether the graph is bipartite.

Justification:

- If a graph is 2-colorable, then all cycles are even (so no odd cycle)
- If there's an odd cycle, 2-coloring fails
- Hence, the link is that checking bipartiteness (with BFS in the previous exercise) is equivalent to checking 2-colorability (with DFS here)

Implementation: See `dfs_exercise2_2colorability_algorithm.py` for detailed analysis.

2.2 Exercise 2: 2-Colorability Algorithm

Problem: We want to write an algorithm, inspired by DFS search which takes as input a graph G(V, E) and which returns a pair (result, color) where result is true if the graph is colorable, false otherwise and color is a dictionary associating a color 0 or 1 to each vertex. This algorithm should stop as soon as possible when the graph is not 2-colorable. Propose an iterative version or a recursive version.

Solution:
We need an algorithm that:

- Uses DFS traversal
- Assigns alternating colors (0 and 1)
- Stops immediately when a conflict (same color on adjacent vertices) is found
- Returns (result, color)

Recursive DFS implementation:

```python
def is_2_colorable_dfs(graph):
    color = {}

    def dfs(node, c):
        color[node] = c
        for neighbor in graph[node]:
            if neighbor not in color:
                if not dfs(neighbor, 1 - c):  # alternate color
                    return False
            elif color[neighbor] == color[node]:
                return False  # conflict: same color for adjacent
                    nodes
        return True

    # handle disconnected graphs
    for v in graph:
        if v not in color:
            if not dfs(v, 0):
                return False, color
    return True, color
```

Listing 3: Recursive DFS 2-Colorability Detection

Iterative DFS implementation (stack-based):

```python
def is_2_colorable_dfs_iterative(graph):
    color = {}

    for start in graph:
        if start not in color:
            stack = [(start, 0)]

            while stack:
                node, c = stack.pop()

                if node in color:
                    if color[node] != c:
                        return False, color  # conflict
                    continue

                color[node] = c
```

```

                    for neighbor in graph[node]:
                        stack.append((neighbor, 1 - c))

    return True, color
```

Listing 4: Iterative DFS 2-Colorability Detection

Example usage:

```
graph = {
    0: [1, 2],
    1: [0, 3],
    2: [0, 3],
    3: [1, 2]
}

print(is_2_colorable_dfs(graph))
# Output: (True, {0:0, 1:1, 2:1, 3:0})     the graph is 2-colorable
    .
```

Listing 5: Example Usage

If there's an odd cycle, the function stops early and returns (False, color).
Features:

- Early termination when conflict is found
- Returns (result, color) pair as specified
- Both iterative and recursive versions
- Time complexity: O(V + E)
- Handles disconnected graphs

3 Dijkstra's Algorithm Results

3.1 Final Results: Shortest Paths from Node A

Destination	Distance	Path
a	0	a
g	1	$a \to g$
f	2	$a \to g \to f$
e	3	$a \to g \to e$
c	4	$a \to g \to e \to c$
b	5	$a \to g \to e \to b$
d	6	$a \to g \to e \to c \to d$

Table 1: Shortest paths from node a to all other nodes

4 Code References

- `exercise1_bfs_distances.py` - BFS distance computation
- `exercise3and5_odd_cycle_detection_bipartite.py` - Odd cycle detection and bipartite graphs
- `dfs_exercise2_2colorability_algorithm.py` - 2-colorability algorithm