Graph Theory - Project Report

Romane DIYAR AZIZ & Chih-Chin CHANG

December 12, 2024

Contents

1 Introduction

This report provides an overview of a Python graph editor and dynamic maze game. The editor allows users to perform various graph operations such as adding/removing nodes and edges, finding paths, and visualizing graphs. The maze game uses the graph to represent a maze, where the player begins from a start node to an end node, using powers such as teleportation.

2 Graph Editor and Analysis Tools

The graph is represented as an adjacency list. It includes a set of operations for adding nodes, adding edges, saving and loading graphs, and performing traversal algorithms.

2.1 Graph Operations

2.1.1 Add Node Function

Listing 1: Add Node Function

```
def add_node(self, node):
    # Add a node to the graph.
    if node not in self.graph:
        self.graph[node] = []
        print(f"Node '{node}' added.")
    else:
        print(f"Node '{node}' already exists.")
```

2.1.2 Add Edge Function

Listing 2: Add Edge Function

```
def add_edge(self, node1, node2, weight=1):
    if node2 not in self.graph[node1]:  # Prevent duplicate edges
        self.graph[node1].append(node2)
        self.weights[(node1, node2)] = weight
        if not self.directed:
            self.graph[node2].append(node1)
            self.weights[(node2, node1)] = weight
        print(f"Edge added: {node1} -- {node2} (weight={weight})")
```

2.1.3 Remove Edge Function

Listing 3: Remove Edge Function

```
def remove_edge(self, node1, node2):
    if node2 in self.graph[node1]:  # Only proceed if the edge exists
        self.graph[node1].remove(node2)
        del self.weights[(node1, node2)]
        if not self.directed:
            self.graph[node2].remove(node1)
            del self.weights[(node2, node1)]
        print(f"Edge removed: {node1} —— {node2}")
```

2.1.4 Save Graph

Listing 4: Save Graph Function

```
def save_graph(self, filename):
    # Save the graph to a file.
    with open(filename, 'w') as file:
        for node, neighbors in self.graph.items():
            for neighbor in neighbors:
                weight = self.weights.get((node, neighbor), 1)
                file.write(f"{node} {neighbor} {weight}\n")
    print(f"Graph saved to {filename}")
```

2.1.5 Load Graph

Listing 5: Load Graph Function

```
def load_graph(self, filename):
    # Load a graph from a file.
    with open(filename, 'r') as file:
        for line in file:
            node1, node2, weight = line.strip().split()
            self.add_edge(node1, node2, int(weight))
    print(f"Graph loaded from {filename}")
```

2.2 Graph Algorithms

2.2.1 Graph Density

The density of a graph is a measure of how many edges are present relative to the number of nodes. It is calculated as the ratio of the number of edges to the maximum possible edges.

Listing 6: Graph Density Function

```
def graph_density(self):
    # Calculate the density of undirected graph.
    num_edges = sum(
        len(neighbors) for neighbors in self.graph.values()) // 2
    num_nodes = len(self.graph)
    if num_nodes > 1:
        max_edges = num_nodes * (num_nodes - 1) / 2
        return num_edges / max_edges
    else:
        return 0  # No edges if there's only one node
```

2.2.2 Node Degree

The degree of a node is the number of edges connected to it.

Listing 7: Node Degree Function

```
def node_degree(self, node):
    # Return the degree of a given node.
    if node in self.graph:
        return len(self.graph[node])
    else:
        return 0  # Node does not exist
```

2.2.3 Shortest Path (Dijkstra's Algorithm)

This function calculates the shortest path between two nodes using Dijkstra's algorithm.

Listing 8: Shortest Path Function

```
def find_shortest_path(self, start, end):
    # Find the shortest path between two nodes (Dijkstra's algorithm).
    distances = {node: float('inf') for node in self.graph}
    distances[start] = 0
    previous_nodes = {node: None for node in self.graph}
    unvisited = set(self.graph)

    while unvisited:
        current = min(unvisited, key=lambda node: distances[node])
        unvisited.remove(current)

        if current == end:
            break

        for neighbor in self.graph[current]:
            weight = self.weights[(current, neighbor)]
            alternative_route = distances[current] + weight
            if alternative_route < distances[neighbor]:
                distances[neighbor] = alternative_route
```

```
                    previous_nodes[neighbor] = current

    # Reconstruct path
    path, current = [], end
    while current:
        path.insert(0, current)
        current = previous_nodes[current]
    return path, distances[end]
```

2.2.4 All Paths

This function finds all paths between two nodes, not just the shortest.

Listing 9: All Paths Function

```
def all_paths(self, start, end, path=None):
    # Find all paths from 'start' to 'end'.
    if path is None:
        path = [start]
    if start == end:
        return [path]
    paths = []
    for neighbor in self.graph[start]:
        if neighbor not in path:
            new_paths = self.all_paths(neighbor, end, path + [neighbor])
            paths.extend(new_paths)
    return paths
```

2.2.5 Strongly Connected Components (SCC)

This function finds the strongly connected components in a directed graph using Tarjan's algorithm.

Listing 10: Strongly Connected Components Function

```
def tarjan_scc(self):
    # Find strongly connected components using Tarjan's algorithm.
    for node in self.graph:
        if node not in self.indices:
            self._strongconnect(node)
    return self.sccs

def _strongconnect(self, node):
        # Helper function for DFS in Tarjan's Algorithm.
        # Set the discovery time and low-link value
        self.indices[node] = self.low_link[node] = self.index
        self.index += 1
        self.stack.append(node)
        self.on_stack.add(node)

        # Explore each neighbor
        for neighbor in self.graph[node]:
            # If the neighbor hasn't been visited
            if neighbor not in self.indices:
                self._strongconnect(neighbor)
                # After the recursive call, we update the low-link value
                self.low_link[node] = min(
                    self.low_link[node], self.low_link[neighbor])
            # If the neighbor is in the current stack
            elif neighbor in self.on_stack:
                # Means the neighbor is part of the current SCC
                self.low_link[node] = min(
                    self.low_link[node], self.indices[neighbor])

        # If node is a root (start point of an SCC)
        if self.low_link[node] == self.indices[node]:
            scc = []
            while True:
                w = self.stack.pop()
                self.on_stack.remove(w)
                scc.append(w)
                if w == node:
                    break
            self.sccs.append(scc)
```

2.2.6 Connected Components

This function finds all connected components in an undirected graph using DFS.

Listing 11: Connected Components Function

```
def connected_components(self):
    # Find connected components using DFS.
    visited = set()
    components = []

    def dfs_helper(node, visited):
        # DFS to collect all nodes in the current connected component.
        visited.add(node)
        component = [node]
        for neighbor in self.graph[node]:
            if neighbor not in visited:
                # Recursive call for unvisited neighbors
                component += dfs_helper(neighbor, visited)
        return component

    for node in self.graph:
        if node not in visited:
            component = dfs_helper(node, visited)
            components.append(component)  # Add the component to the list

    return components
```

2.2.7 Graph Coloring

This function assigns colors to nodes so that no two adjacent nodes share the same color.

Listing 12: Graph Coloring Function

```
def graph_coloring(self):
    # Assign colors such that no two adjacent nodes share the same color.
    color = {}
    for node in self.graph:
        available_colors = {0, 1, 2, 3}  # For simplicity, just 4 colors
        for neighbor in self.graph[node]:
            if neighbor in color:
                available_colors.discard(color[neighbor])
        # Assign the smallest available color
        color[node] = min(available_colors)
    return color
```

3 Maze Game Functions

In the maze game, players navigate through the graph from a start node to an end node, using different powers like teleportation and blocking graph dynamics.

3.1 Apply Dynamics

This function modifies the graph each turn by adding and removing edges, ensuring the graph remains connected.

Listing 13: Apply Dynamics Function

```
def apply_dynamics(self, block_dynamics=False):
    # Ensure at least one edge is added and one edge is removed per turn.
    if block_dynamics:
        print("Graph dynamics blocked this turn!")
        return

    nodes = list(self.graph.graph.keys())

    # Ensure adding a new edge
    if len(nodes) > 1:
        added = False
        for _ in range(10):  # Try up to 10 times to find a valid pair
            node1, node2 = random.sample(nodes, 2)
            # Only add if no edge exists
            if node2 not in self.graph.graph[node1]:
                self.graph.add_edge(node1, node2, random.randint(1, 10))
                added = True
                break
        if not added:
            print("No valid edge to add this turn.")

    # Ensure removing an edge while keeping the graph connected
    if len(nodes) > 1:
        removed = False
        for _ in range(10):  # Try up to 10 times to find a valid edge
            node1, node2 = random.sample(nodes, 2)
            # Only remove if edge exists
            if node2 in self.graph.graph[node1]:
                self.graph.remove_edge(node1, node2)

                # Check if the graph is still connected
                components = self.graph.connected_components()
                if len(components) == 1:  # Graph is still connected
                    removed = True
                    break
                else:  # Graph would be disconnected, re-add the edge
                    self.graph.add_edge(node1, node2, random.randint(1, 10))

        if not removed:
            print("No valid edge to remove this turn.")
```

3.2 Use Power

This function allows the player to use a power such as viewing the graph or teleporting.

Listing 14: Use Power Function

```
def use_power(self):
    # Allow the player to use a power.
    print("Available powers:")
    print(f"1. View graph structure for the next 3 turns (
        {self.powers['view_graph']} left)")
    print(f"2. Block graph dynamics for one turn (
        {self.powers['block_dynamics']} left)")
    print(f"3. Teleport to an unconnected node (
        {self.powers['teleport']} left)")
    choice = input("Choose a power (1/2/3 or press Enter to skip): ")

    if choice == '1' and self.powers['view_graph'] > 0:
        self.show_graph_turns = 3
        self.powers['view_graph'] -= 1
        print("You can now view the graph structure for the next 3 turns!")
        return 'view_graph'
    elif choice == '2' and self.powers['block_dynamics'] > 0:
        self.powers['block_dynamics'] -= 1
        print("You have blocked graph dynamics for the next turn!")
        return 'block_dynamics'
    elif choice == '3' and self.powers['teleport'] > 0:
        self.powers['teleport'] -= 1
        return 'teleport'
    elif choice == '':
        print("You chose to skip using a power.")
    else:
        print("Invalid choice or no powers left for this option!")
    return None
```

3.3 Play Game

This function handles the core logic of the maze game, where the player moves from the start to the end node, using powers and interacting with the graph.

Listing 15: Play Game Function

```
def play_game(self):
    # Play the maze game.
    current_position = self.start
    print(f"Start: {self.start}, Goal: {self.end}")

    while current_position != self.end:

        print(f"Current position: {current_position}")

        # Allow the player to use a power
        power_action = self.use_power()

        # Handle teleportation
        if power_action == 'teleport':
            current_position = self.teleport(current_position)
            print(f"After teleporting, current position is:
                {current_position}")

            if current_position == self.end:
                print("You reached the goal!")
                break

        self.apply_dynamics(block_dynamics=(
            power_action == 'block_dynamics'))

        # Show the updated graph structure after dynamics (if viewing
            power is still active)
        if self.show_graph_turns > 0 and self.powers['view_graph'] > 0:
            self.graph.display_graph()
            self.show_graph_turns -= 1

        # Check if player reached the goal after the move
        if current_position == self.end:
            print("You reached the goal!")
            break

        # Player chooses a move
        move = input(f"Choose your next node from
            {self.graph.graph[current_position]}: ")
        if move in self.graph.graph[current_position]:
            current_position = move
            print(f"Moved to {current_position}")
        else:
            print("Invalid move!")

    print("Game finished!")  # Message after reaching the goal
```

4 Testing and Results

4.1 Graph Structure

```
A: ['B', 'C']
B: ['A', 'D', 'E']
C: ['A']
D: ['B', 'F']
E: ['B', 'G']
F: ['C', 'D', 'H']
G: ['E', 'I']
H: ['F', 'I']
I: ['G', 'H']
```

4.2 Basic Graph Metrics Results

```
Degree of A: 2
Graph Density: 0.2777777777777778
```

4.3 Pathfinding Results

```
Shortest Path from A to I: (['A', 'C', 'F', 'H', 'I'], 11)
All Paths from A to I: [['A', 'B', 'D', 'F', 'H', 'I'], ['A', 'B', 'E', 'G', 'I'],
    ['A', 'C', 'F', 'D', 'B', 'E', 'G', 'I'], ['A', 'C', 'F', 'H', 'I']]
```

4.4 Graph Traversal Results

```
BFS starting from A to I: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']
DFS starting from A to I: ['A', 'B', 'D', 'F', 'H', 'I']
```

4.5 Connected Components Results

```
Connected Components: [['A', 'B', 'D', 'F', 'C', 'H', 'I', 'G', 'E']]
```

4.6 Strongly Connected Components Results

```
Strongly Connected Components: [['D'], ['C', 'B', 'A']]
```

4.7 Graph Coloring Results

```
Graph Coloring: {'A': 0, 'B': 1, 'C': 1, 'D': 0,
    'E': 0, 'F': 2, 'G': 1, 'H': 0, 'I': 2}
```

4.8 Maze Game Results

```
You reached the goal!
```

5 Conclusion

This project successfully integrates graph theory algorithms with an interactive maze game. The implementation allows users to add nodes, edges, and visualize graph structures while playing a game with various dynamic features such as teleportation and blocking graph changes.