Final Project Part 03 README

Project Summary:
A Java library for parsing, manipulating, and searching graphs. This project provides functionality to work with directed graphs, including parsing DOT format files, adding/removing nodes and edges, and performing various graph search algorithms (BFS, DFS, and Random Walk).

Core Functionality:
- Parse graph representations from DOT files
- Add and remove nodes and edges
- Output graphs in DOT format
- Render graphs as PNG or SVG images using Graphviz

Search Algorithms:
- Breadth-First Search (BFS): Finds the shortest path between two nodes
- Depth-First Search (DFS): Explores as far as possible along branches before backtracking
- Random Walk: Uses a probabilistic approach to find paths, with backtracking capability

Prerequisites:
- Java 17 or higher
- Maven

Building Project:
git clone https://github.com/shiven01/CSE-464-2025-sshekar9
cd "Project Part 1 + 2"
mvn clean install

Note:
Running “mvn package” WILL compile the code, but it won’t display all of the tests and in proper formatted output.

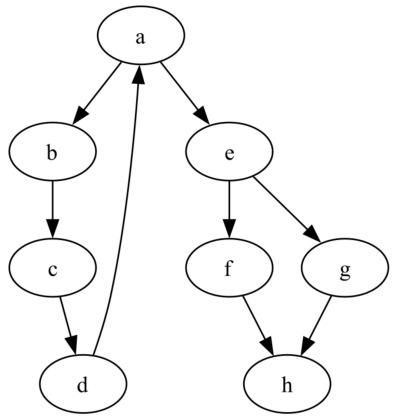
Running BFS, DFS, Random Walk

Steps to run BFS, DFS, and Random Walk all at the same time:
cd "Project Part 1 + 2"
mvn compile exec:java -Dexec.mainClass="com.shivenshekar.graphparser.demo.SearchDemo"

Output:

```
=== BFS Demonstration (Scheme A) ===

visiting Path{nodes=[Node{a}]}
visiting Path{nodes=[Node{a}, Node{b}]}
visiting Path{nodes=[Node{a}, Node{e}]}
Path{nodes=[Node{a}, Node{b}, Node{c}]}

=== DFS Demonstration (Scheme A) ===

visiting Path{nodes=[Node{a}]}
visiting Path{nodes=[Node{a}, Node{b}]}
visiting Path{nodes=[Node{a}, Node{b}, Node{c}]}
Path{nodes=[Node{a}, Node{b}, Node{c}]}

=== Random Walk Demonstration ===

Random Walk from a to h:
Attempt 1: a->b->c->d->a->e->f->h (target node!)
Attempt 2: a->b->c->d->a->e->f->h (target node!)
Attempt 3: a->b->c->d->a->e->f->h (target node!)
Attempt 4: a->b->c->d->a->e->g->h (target node!)
Attempt 5: a->b->c->d->a->e->f->h (target node!)
```

```
Random Walk Summary:
Found 2 different successful paths out of at least 5 attempts.
1: a->b->c->d->a->e->f->h
2: a->b->c->d->a->e->g->h
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  27.182 s
[INFO] Finished at: 2025-05-05T17:08:30-07:00
[INFO] ------------------------------------------------------------------------
shivenshekar@Shiven-Shekar Project Part 1 + 2 %
```



Project Part 03 Feature Request Commit Links:

Five Refactorings:

1. Link: Additional Refactoring: Extracted parsing logic parseDotContent, parseEdge, parseNode, GraphContent to separate file to promote single responsibility design principle
2. Link: Additional Refactoring: Package Reorganization. Established separation of concerns for project files, enhanced discoverability, enhanced maintainability, scalability, etc.
3. Link: Additional Refactoring: Extracted Graph Renderer Interface from core Graph class. Modularized code and further promoted Single Responsibility principle.
4. Link: Additional Refactoring: Moved the graph search methods from the Graph class to the GraphSearchService class to handle all of the path-finding operations. This makes the codebase more modular and promotes the single responsibility principle.
5. Link: Additional Refactoring: Moved all of the path formatting methods to the Path Formatting utility class and updated the search demo to use those methods, therefore promoting modularity and single responsibility principle

Template Pattern Refactoring:

- Link: When I apply the template pattern to BFS and DFS, I first identify the algorithmic skeleton they share, which includes initialization, traversal logic, and node processing. I create an abstract GraphSearch class containing a template method that defines this common structure while declaring abstract methods for the varying parts. For the implementation, I create concrete subclasses: BFSSearch uses a queue to visit nodes level by level, while DFSSearch uses a stack to explore branches deeply before backtracking. Each subclass implements the abstract methods according to its specific traversal strategy - for example, BFSSearch.getNextVertex() dequeues from the front (FIFO), while DFSSearch.getNextVertex() pops from the top (LIFO). This approach lets me maintain the invariant parts in one place while allowing each algorithm to define its own node-ordering behavior, making the code more maintainable and clearly expressing the relationship between these related algorithms.
- Commit Notes:
    - Extracted common search functionality into an abstract GraphSearchAlgorithm class
    - Implemented the Template Method pattern to define the skeleton of the search algorithm
    - Created concrete BFSAlgorithm and DFSAlgorithm classes that extend the base class
    - Modified Graph class to use the new algorithm classes
    - Maintained backward compatibility with existing code

Strategy Pattern Refactoring:

- Link: When I apply the Strategy pattern to BFS and DFS, I focus on extracting the algorithmic behaviors into separate strategy classes while maintaining a unified

interface. First, I define a SearchStrategy interface that declares the common method signature for all search algorithms. Then, I implement concrete strategy classes like BFSStrategy and DFSStrategy, each containing their specific traversal logic - BFS using a queue for breadth-first exploration and DFS using a stack or recursion for depth-first exploration. I then create a SearchContext class that maintains a reference to the current strategy and provides a method to switch between strategies at runtime. When the client code needs to perform a graph search, it simply configures the context with the desired strategy and calls the context's execution method, which delegates the actual search operation to the current strategy object. This approach eliminates complex conditional logic in the client code, makes adding new search algorithms straightforward, and allows algorithms to vary independently from the code that uses them.

- Commit Notes:
    - Created SearchStrategy interface for different search algorithms
    - Adapted Template Pattern classes to implement the Strategy interface
    - Created SearchContext class to manage algorithm selection
    - Modified Graph class to use the Strategy Pattern
    - Maintained backward compatibility with existing code

Random Walk Search Implementation:

- Link: When I implemented the Random Walk algorithm as part of the Strategy pattern for graph searching, I focused on creating a solution that would explore graphs in a fundamentally different way than BFS or DFS. In my RandomWalkStrategy class, I built a search method that doesn't deterministically follow a specific traversal pattern but instead makes probabilistic decisions at each step. I start by placing the source node in a set of visited nodes and then repeatedly select a random neighbor of the current node to visit next. If the current node has no unvisited neighbors, I implement a "teleportation" mechanism that randomly jumps to another visited node that still has unexplored neighbors. What makes my implementation special is that it incorporates an element of randomness that can be useful for certain graph applications like exploring social networks or simulating particle movements. I maintain a path tracking structure that records the nodes visited, which allows the algorithm to return a valid path if it happens to find the destination node during its random exploration. The beauty of using the Strategy pattern here is that I can simply plug this new RandomWalk algorithm into the existing SearchContext without modifying the client code at all - clients can now choose between BFS, DFS, or RandomWalk simply by passing the desired algorithm to the context object.
- Commit Notes:
    - Added RANDOM to Algorithm enum
    - Implemented RandomWalkAlgorithm class following Template and Strategy patterns
    - Updated SearchContext to support the new algorithm
    - Added unit tests for Random Walk algorithm
    - Demonstrated randomness through multiple search attempts