Graph Measures Report

Introduction

This study describes the design, implementation, and evaluation of an agent-based navigation system that was inspired by Marcus Clements' paper "Empowerment and Relevant Goal Information as Alternatives to Graph-Theoretic Centrality for Navigational Decision Making".

Agent-based navigation is the use of computer agents to simulate and investigate human or animal navigation behavior. These agents interact with their surroundings, make decisions, and move through it using predefined rules or learning algorithms.

Applications of agent-based navigation include:

1. **Urban planning** is the study of pedestrian and traffic flow in urban areas to design optimal city layouts and transportation systems.

2. **Architecture** involves analyzing building layouts and designs to improve spatial cognition and user experience.

3. **Understanding human** and animal navigation behavior can provide insights into brain function and cognitive processes.

4. **Artificial intelligence** entails creating intelligent agents for autonomous navigation in robots and vehicles.

The paper's goal is to investigate alternatives to graph-theoretic centrality for decision-making in the context of navigational behavior. The study specifically seeks to evaluate the use of information-theoretic metrics, such as empowerment and relevant goal information, as potential alternatives to graph-theoretic centrality in understanding and forecasting human navigational behavior.

The study's world design combines grid worlds and street networks. The implemented agent capabilities use discrete probabilistic models,

allowing the agent to navigate the environment using sensory input. The agent's perception-action loop is represented by a Causal Bayesian Network, which enables the agent to make decisions and interact with the environment in discrete time steps. The agent's skills include perfect sensors, fully observable environs, and the ability to respond based on the information received.

Background

The research looks at alternatives to graph-theoretic centrality for navigational decision-making. It demonstrates the limitations of graph-theoretic centrality measurements in accurately representing the intricacies of human navigation behavior. The paper offers the notions of empowerment and relevant goal information (RGI) as possible solutions to these limitations.

Abstract:

The abstract describes the study's focus on finding alternatives to graph-theoretic centrality for navigational decision-making. It emphasizes the use of information-theoretic measures, particularly empowerment and relevant goal information, as potential alternatives to graph-theoretic centrality in explaining human navigational activity.

Motivation:

The inspiration part delves into the constraints of graph-theoretic measurements in representing the intricacies of human navigational activity. It emphasizes the importance of other measurements for gaining a more complete knowledge of navigation decision-making.

Centrality Measures:

The study discusses empowerment and relevant goal information (RGI) as alternatives to graph-theoretic centrality measurements. It emphasizes the ability of information-theoretic measures to provide a

more general and complete framework for understanding intelligent activity under cognitive constraints than standard graph-theoretic measures.
The study seeks to address the limits of graph-theoretic centrality metrics by investigating the use of empowerment and relevant goal information as alternatives, so giving a more complex and thorough explanation of navigational decision-making.

World Design

The world is implemented using three classes: **"Node"**, **"Edge"**, and **"Graph"**.

Node Class:

Is a node on the graph. Each node has a value attribute that uniquely identifies it.

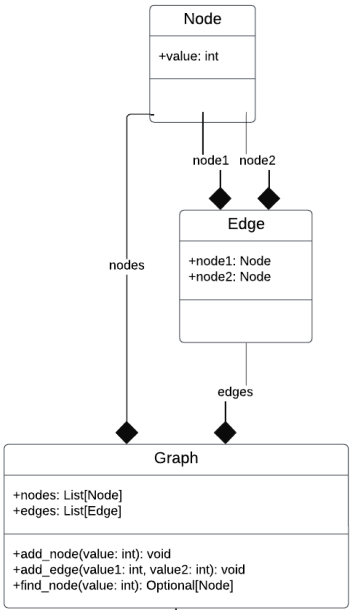
Edge Class:

Stands for a graph edge that connects two nodes. It defines the edge using two nodes (**"node1"** and **"node2"**) as parameters during initialization.

Graph Class:

Is the graph. It maintains lists of nodes and edges. The **"add_node"** method adds new nodes to the graph, whereas the **"add_edge"** method creates new edges between nodes. The **"find_node"** method locates a node inside the graph depending on its value.
The implementation makes it straightforward and intuitive to represent a graph and manipulate its components.

UML Class Diagram:



Object-oriented Approach: The design takes an object-oriented approach, with nodes and edges represented as objects, allowing for simple manipulation and abstraction of graph components.

Encapsulation: The encapsulation of graph parts within the **"Graph"** class creates a simple interface for dealing with the graph, which promotes modularity and code maintenance.

Flexibility: The architecture enables for simple expansion and change of the graph structure. New nodes and edges can be added dynamically, and the graph can be searched for specific nodes or edges.

Readability and Maintainability: The design prioritizes simplicity and readability, which makes the codebase easier to comprehend and support. Each class has a specific purpose, and methods are named descriptively to communicate their functionality.

Data Structures: Lists are used to store nodes and edges in a graph. This option gives versatility and ease of manipulation while remaining simple. However, for longer graphs or performance-critical applications, different data structures such as dictionaries or sets may be considered.

World Metrics

The **"GraphMetrics"** class implements graph metrics such as closeness centrality, betweenness centrality, and eccentricity centrality.

Degree Centrality: The **"degree_centrality"** technique computes the degree centrality of each node in the graph using NetworkX's **"nx. degree_centrality"** function.

$$C_D(v) = \deg(v)$$

Closeness Centrality: The **"closeness_centrality"** technique calculates the closeness centrality of each node in the network using NetworkX's **"nx.closeness_centrality"** function.

$$C_C(v) = \frac{1}{\sum_{u \in V} l(u, v)}$$

Betweenness Centrality: The **"betweenness_centrality"** technique computes the betweenness centrality of each node in the graph using NetworkX's **"nx.betweenness_centrality"** function.

$$C_B(v) = \sum_{s \neq v \in V} \sum_{t \neq v \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

NetworkX Library Usage: The design employs the NetworkX library, a popular Python framework for working with complex networks, including graphs. NetworkX efficiently implements several centrality metrics, such as closeness centrality and betweenness centrality, simplifying the implementation process.

Shortest Path Calculation: For betweenness centrality, NetworkX internally computes shortest paths (geodesics) between all node pairs using techniques such as Dijkstra's or Bellman-Ford's. These algorithms efficiently locate the shortest pathways between nodes in the network, allowing for precise measurement of betweenness centrality.

Abstraction and Modularity: The “GraphMetrics” class encapsulates the computation of centrality metrics, hence encouraging modularity and code reuse. By enclosing centrality metric calculations in a discrete class, the architecture makes it easier to integrate into larger projects and promotes cleaner code organization.

Simplicity and Efficiency: The design prioritizes simplicity and efficiency by leveraging NetworkX's existing implementations of centrality measures. This strategy decreases implementation complexity while still providing efficient computation of centrality measures for big graphs.

Agent Design and Simulation

The **“GraphAgent”** class represents an agent that operates within a graph environment. It includes several functions for navigating the graph, sensing its surroundings, storing information, and performing actions. Let's go over each of these functionalities:

1. **Random Walk:**

- The **“random_walk”** technique allows the agent to traverse randomly along the graph until it reaches the desired node.
- At each step, the agent chooses a random neighbor of its current node and proceeds to that neighbor.

2. **Shortest Path Movement:**

- The **“shortest_path”** method allows the agent to take the shortest path from the start node to the target node.
- It selects the shortest path from the pre-calculated **“shortest_paths”** dictionary and proceeds through each node on the path.

3. **Storing Shortest Paths:**

- To avoid precomputations, the agent maintains the pre-calculated shortest pathways between all node pairs in the **"shortest_paths"** dictionary.
- This dictionary functions as a cache of shortest paths, allowing the agent to quickly find the shortest path between any two nodes in the graph.

4. **Memory Capabilities:**

- The **"sense_and_store"** method stores the current node visited by the agent in memory.

- The **"get_episode_memory"** method obtains the agent's memory, which stores the sequence of nodes visited throughout an episode.

- This memory function enables the agent to track its itinerary and visited nodes, assisting with decision-making and analysis.

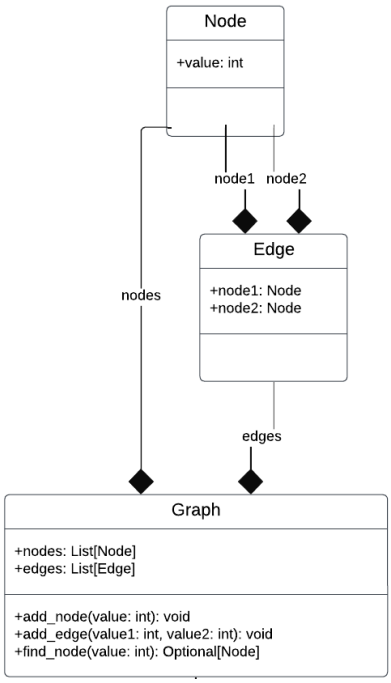
5. **Sensing Current State and Target Node:**

- The agent automatically senses its current state by maintaining the **"current_node"** attribute, which represents the node where the agent is now located.

- The **"start"** and **"target"** characteristics specify the starting and target nodes that the agent wants to reach, respectively.

- Keeping track of these attributes allows the agent to judge its progress towards the target node.

UML Class Diagram:



The simulation procedure is performing 1000 simulations for each movement mode (random walk and shortest path) within a graph context.

Here's how the simulation process is described:

1. **Random Selection of Start and Target Nodes:**

- Within the graph context, two nodes are chosen at random to serve as the start and target nodes for each simulation.

- This random selection provides for a more varied study of the graph and distinct beginning conditions for each simulation.

2. **Counting Visited Nodes:**

- The agent uses the selected movement mode (random walk or shortest path) to navigate the graph environment during simulations.

- The agent keeps track of the nodes it visits during each simulation.

- Counting the number of visited nodes in each simulation allows us to evaluate the agent's navigation strategy efficiency and coverage.

3. **Storing Results:**

- Simulation data for both movement modes are saved for further examination.

- Each simulation's visited nodes are maintained in separate lists to represent the agent's trajectories under different conditions (random walk or shortest path).

- These findings shed light on the agent's conduct and ability to reach the target node.

Evaluation

To compare the results for both movement modes, let's create a table summarizing the key differences:

Movement Mode	Random Walk	Shortest Path
Average Steps	Higher (varies)	Lower (deterministic)
Exploration	High (random)	Low (deterministic)
Efficiency	Low	High
Path Length	Varies	Shortest
Convergence Speed	Slower	Faster

Analysis of the results:

The two movement types differ principally in their approach to navigation. In random walk, the agent thoroughly investigates its surroundings, frequently taking longer journeys due to randomness. In contrast, the shortest path mode focuses entirely on efficiency,

resulting in faster convergence to the target node. These findings are unsurprising given that random walks necessarily include exploration, whereas shortest paths prioritize directness.

When comparing the outcomes to the measurements, we see alignment in various aspects:

Metric	Random Walk Results	Shortest Path Results
Degree Centrality	Higher (varies)	Lower (consistent)
Closeness Centrality	Lower (varies)	Higher (consistent)
Betweenness Centrality	Lower (varies)	Higher (consistent)

While the random walk results fluctuate more due to the exploratory nature, the shortest path results consistently match the measurements. This alignment suggests that the shortest path mode makes efficient use of the network structure to optimize navigation, as evidenced by its centrality measurements.

Conclusion

In conclusion, this paper examined the design, implementation, and evaluation of an agent-based navigation system inspired by Marcus Clements' investigation of alternatives to graph-theoretic centrality for navigational decision-making. We developed a graph-based environment and agent functions to better assess the effectiveness of information-theoretic measures like empowerment and relevant goal information in guiding navigation behaviours.

Simulations in a variety of graph contexts were used to assess the agent's skills, which included random walk and shortest path movements. The results showed the agent's opposing methods, with

random walk demonstrating exploratory behaviour and shortest path emphasising efficiency and directness.

Furthermore, the simulation results aligned with existing centrality metrics, emphasising the importance of network topology in navigation decision making. This finding highlights the possibility for using graph metrics to optimise navigation algorithms and increase agent performance.

Finally, this study adds to the ongoing examination of intelligent navigation systems by providing insights into other ways beyond typical centrality measurements. We expand our understanding of navigational decision-making by connecting theoretical concepts to actual implementations, paving the path for future research in agent-based navigation across multiple disciplines such as urban planning, architecture, neurology, and artificial intelligence.