Team Name and Members

Project Name: Suggestion Path Mapping

Group 152 : Yepeth Berhie, Carrie Ruble, Adam Schwartz, Kevin Yu

GitHub Repo: https://github.com/CouldBeYourMom/suggestion-path-mapping

Video Demo: https://youtu.be/D8Phex495pc

Extended and Refined Proposal

Problem: What problem are we trying to solve?

Modern content platforms like YouTube rely heavily on algorithmic recommendation systems to maximize user engagement. While these systems are highly effective at keeping users on the platform, they can unintentionally lead viewers down paths that diverge from their original intent—particularly in cases involving children. Starting from safe or age-appropriate content, a user may be nudged toward videos that are increasingly inappropriate, extreme, or otherwise misaligned with parental or educational expectations.

Our project aims to simulate and analyze this phenomenon by representing YouTube's recommendation structure as a graph. We model how a user might traverse this graph from a child-safe starting point, following algorithmically suggested videos. By applying different search algorithms, Dijkstra's, A*, and Random Walk, we investigate how quickly and through what kinds of pathways a user could encounter flagged or unsuitable content. This analysis allows us to better understand the structure and risk patterns embedded within the recommendation system.

Motivation: Why is this a problem?

Understanding the structure and behavior of recommendation pathways is crucial for ensuring transparency, safety, and ethical content delivery on digital platforms. YouTube's recommendation algorithm can unintentionally direct users, especially children, toward content that is misaligned with their age, values, or expectations. This becomes particularly concerning when pathways lead to videos involving violence, misinformation, or psychologically manipulative content.

By modeling the recommendation network as a graph and simulating traversal using different search strategies, we can explore how quickly a user might drift into problematic content spaces. This type of analysis provides valuable insight into the architecture of algorithmic suggestions and their unintended consequences.

Our findings can inform the design of safer algorithms, the implementation of more effective parental controls, and the development of tools for auditing and regulating content recommendations. Ultimately, this work contributes to the broader goal of holding platforms accountable for the user experiences they shape, especially for vulnerable audiences.

Features implemented

Our project focuses on evaluating how different traversal strategies navigate a stat-weighted suggestion graph. To enable meaningful comparison, we implemented a robust pipeline that includes:

- **Graph construction** from both real and synthetic YouTube recommendation data, allowing us to control for content scope and relationship density.
- **Three traversal strategies**—Dijkstra's, A*, and Random Walk—each designed to simulate different user behaviors and algorithmic priorities within the same graph structure.
- **Stat-based edge weighting**, using raw counts of flags, likes, views, or comments. A custom transformation function (veraVerto) ensures that higher values are favored by shortest-path algorithms without altering their internal logic.
- **Cycle detection and edge deduplication**, ensuring that repeated recommendations across multiple parent videos are preserved in the graph structure without inflating traversal paths. For the real-world dataset, this maintained the naturally cyclical nature of recommendation behavior. For the synthetic dataset, we manually introduced edge overlap to simulate cycles and control for uniformity
- **Node visit logging**, which records the timestamp of each node as it is visited. This supports both traversal path analysis and dynamic visualization.
- **Export capabilities**, including CSV and JSON output of traversal logs, which allow for both structured comparison and external rendering.
- A **prototype visualization tool** that highlights traversal paths interactively. While not the focus of the project, this tool serves as a valuable aid for interpreting algorithm behavior and verifying results visually.

Description of data

This project uses two primary datasets to support both exploratory testing and algorithmic comparison:

- **Real-world dataset**: Over 2,000 unique YouTube videos were manually collected through seed-based crawling of the platform's recommendation system. Starting from child-safe content, we recorded recommended video links across multiple layers of suggestions. Each video in the dataset includes its unique ID, title, and four core statistics: flag count, like count, view count, and comment count.
- **Synthetic dataset**: To enable performance benchmarking and large-scale stress testing, we supplemented the real-world data with a synthetic graph comprising 100,000 tuples. This dataset is based on structural properties from the publicly available com-Youtube graph provided by Stanford's SNAP project. The synthetic data mimics realistic connectivity patterns, allowing us to evaluate traversal scalability in a controlled environment.

Using both datasets allowed us to test our algorithms under real-world conditions and at larger theoretical scales. This dual approach was particularly valuable for comparing traversal strategies—such as A*, Dijkstra, and Random Walk—under conditions that mirrored both the depth of actual user paths and the breadth of large-scale recommendation systems.

Tools/Languages/APIs/Libraries used

Our project utilized a combination of programming languages, libraries, and external tools to manage data collection, processing, traversal, and visualization:

- **C++** was used to implement all core logic and traversal algorithms (Dijkstra's, A*, and Random Walk), as well as graph construction, stat-based weight transformation, and traversal export for analysis and visualization.
- **Python** was used for data preprocessing and collection, including script-based access to the YouTube Data API. It also facilitated file I/O, database interactions, and controlled the full compile-and-run workflow for convenience.
- **SQLite** served as the project's local database, providing lightweight yet structured storage for all collected video metadata, relationships, and traversal logs.
- **YouTube Data API v3** (part of the Google API suite) was used to collect video metadata, transcripts, comments, and recommendation links. Playlists were manually constructed to simulate algorithmic suggestions and avoid the limitations of automated tracking.
- **JavaScript (Three.js + 3d-force-graph)** was used in a prototype visualization tool that renders traversal paths interactively for debugging and analysis. While not the focus of the project, this tool proved valuable for verifying search behavior.
- **GitHub** was used for version control, collaboration, and staging updates across multiple branches. Development was managed across PowerShell, VSCode, Clion, and MSYS2 UCRT64 terminals for maximum compatibility.

Algorithms implemented

We implemented three primary search algorithms, each chosen to reflect a different approach to navigating recommendation graphs:

- **Dijkstra's Algorithm** – A classic shortest-path algorithm that explores all possible paths while favoring lower-cost edges. Ideal for finding globally optimal routes across large graphs.
- **A* Search** – A heuristic-guided variation of Dijkstra's that prioritizes paths that appear more promising, reducing search overhead when the heuristic is well-aligned with the graph structure.

- **Random Walk** – A probabilistic approach that mimics how a naive user might traverse a suggestion network without clear direction, selecting each next node based on stat-derived weighted probabilities.

We also implemented **Depth-First Search (DFS)** and **Breadth-First Search (BFS)** to support testing, comparison, and visualization scaffolding. These baseline strategies helped us validate traversal accuracy and visualize node ordering early in development.

Additional Data Structures/Algorithms used

To support the above traversal strategies and maintain performance across datasets, we used a range of built-in C++ data structures:

Structure	**Usage Context**
Adjacency List	Main graph representation (video → suggested videos)
Hash Map (unordered_map)	Used for: parents, distances, fValues, and videoStatsMap
Hash Set (unordered_set)	Used for: visitedNodes, visitedPairs in DFS/BFS/random walk
Priority Queue (std::priority_queue)	Min-heap behavior for A* and Dijkstra
Vector (std::vector)	For storing neighbors, visit logs, and path outputs
Pair (std::pair)	Representing (video_id, weight) and (video_id, timestamp)

Distribution of Responsibility and Roles: Who did what?

This project was a collaborative effort between four team members, with responsibilities distributed to reflect each member's strengths and interests. All major components were built with coordination across the team, and each member maintained ownership of their individual contributions through GitHub workflows.

Yepeth Berhie [@Y-Berhie] implemented Dijkstra's algorithm and contributed to the early design of the graph structure and stat-weight integration. He helped validate traversal output, test comparison logic, and supported refinement of edge cases during integration.

Carrie Ruble [@CouldBeYourMom] led the development of the unified export and visualization pipeline, managed the SQLite database and data collection scripts, and implemented the transformation function to support stat-based weighting. She coordinated repository structure, version control, documentation, and final report editing.

Adam Schwartz [@schwartza-afs] implemented the Random Walk traversal, including probability-based neighbor selection. He also documented the algorithm's theoretical underpinnings and collaborated on traversal output formatting and testing.

Kevin Yu [@kevinyuQ] developed the A* search algorithm and designed the associated heuristic to align with stat-based edge weights. He worked closely on debugging traversal behavior and integrating the A* output into the export and visualization system.

Each team member also contributed to presentation preparation, peer testing, and discussions around complexity trade-offs, algorithm behavior, and ethical considerations.

Analysis

Any changes the group made after the proposal? The rationale behind the changes.

After submitting our proposal, we added a fourth group member to assist with implementation and documentation. Additionally, we introduced a third traversal algorithm, Random Walk, to broaden our comparison set. Random Walk was a valuable addition because it represents a stochastic, memoryless navigation method that contrasts meaningfully with the more deterministic and optimization-focused behaviors of A* and Dijkstra's algorithms. Given the unpredictable and exploratory nature of YouTube's recommendation graph, Random Walk serves as an effective baseline for evaluating how different algorithms uncover pathways through complex, user-influenced suggestion networks.

Big O worst case time complexity analysis of the major functions/features you implemented

Our project focused on three primary traversal strategies, Dijkstra's Algorithm, A* Search, and a weighted Random Walk, with DFS and BFS retained for comparison and baseline testing.

Dijkstra's Algorithm has a worst-case time complexity of $O(E + V \log V)$, where E is the number of edges and V is the number of vertices. The algorithm relaxes every edge and updates node priorities using a min-heap (priority queue), which enables efficient distance updates throughout the graph. Since we used a priority queue and adjacency list structure, our implementation adheres to this complexity.

A* Search has a worst-case time complexity of $O(EV \times \log V)$, where E is the number of edges and V is the number of nodes in the graph. In the worst case, the heuristic function must consider up to V neighbors in each iteration, contributing to the higher theoretical complexity. The algorithm also relaxes every edge and updates the priority queue during each iteration, which compounds the overall runtime. While the heuristic can improve efficiency in well-structured graphs, in the worst case it behaves similarly to Dijkstra's algorithm but with additional overhead.

Weighted Random Walk has a worst-case time complexity of $O(N^2)$, where N is the number of nodes. At each step, the algorithm performs a weighted random selection of neighbors based on a stat-derived probability distribution. While this selection is efficient in practice, the theoretical worst case assumes every node is revisited through many iterations, leading to quadratic behavior in sparse graphs.

To ensure that shortest-path algorithms like Dijkstra and A* would favor higher values (e.g., more likes or more flags), we applied a preprocessing transformation to all edge weights using a custom function in our graph.cpp file: *transformed = log(raw + 1) / (raw + 1)*. This transformation is designed to be monotonically decreasing, so that larger raw values yield smaller transformed weights. It ensures all edge weights remain positive, preventing any undefined behavior in graph traversal. The function also preserves a meaningful separation between different magnitudes, allowing algorithms to still differentiate between, for example, 100 likes and 10,000 likes, while compressing the scale. By applying the transformation before passing any data to the traversal algorithms allowed each to operate effectively without modifying its internal logic.

Algorithm Comparison

Using a test dataset of 100,000 tuples simulating a connected suggestion graph, we compared four graph traversal algorithms: Depth-First Search (DFS), Dijkstra's algorithm, A* Search, and Random Walk. Our goal was to evaluate how efficiently each search type reached highly flagged content, simulating a user's path through potentially harmful or inappropriate recommendations.

All algorithms were able to reach the most flagged video in the dataset (flag_count = 500), but their efficiency varied significantly:

- **Dijkstra** reached the top-flagged video fastest (63.77 ms), leveraging its exhaustive cost-based strategy.
- **DFS** followed at 102.69 ms, benefitting from its deep path-first structure.
- **A*** reached the same flag level at 348.86 ms, guided by a simple weight-based heuristic.
- **Random Walk** was the slowest, reaching the top-flagged video in 4080.17 ms, illustrating the unpredictable nature of unguided navigation.

To evaluate real-world traversal behavior, we ran each of the four search algorithms on our hand-collected YouTube dataset, starting from a known child-safe video. The goal was to simulate how quickly and efficiently each strategy could reach the most heavily flagged content—representing potentially inappropriate or harmful videos.

Each algorithm successfully reached the most flagged video (flag_count = 441), but with notable variation in performance:

- **Dijkstra** reached the flagged video in 28.92 ms, demonstrating the strength of a cost-based approach in tightly connected graphs.
- **DFS** followed closely at 46.13 ms, showing that even unguided depth-first exploration can be effective in low-depth recommendation trees.
- **A*** took 96.45 ms, reflecting the influence of its stat-weighted heuristic even in a relatively small graph.
- **Random Walk** required 1177.71 ms, underscoring the inefficiency of stochastic movement when no prioritization is applied.

While the absolute times are smaller than in the synthetic dataset (due to graph size), the relative behavior remains consistent: Dijkstra and DFS perform efficiently, A* varies with heuristic quality, and Random Walk lags significantly behind.

Additionally, one key distinction between the datasets is structural: the real-world YouTube graph naturally includes cycles, as many videos suggest the same downstream content. In contrast, cycles were artificially introduced in the synthetic dataset by manually adding relationships between distant nodes. As a result, the traversal behavior in the real graph more accurately reflects the non-linear, user-influenced patterns of real recommendation systems, while synthetic cycles may appear more uniform or random.

Reflection

As a group, how was the overall experience for the project?

The project was collaborative and intellectually rewarding. Each team member took ownership of a core component, from traversal algorithms to visualization and database handling, and we all benefited from each other's perspectives. Throughout the process, we gained experience working with GitHub workflows, resolving merge conflicts, and coordinating across branches. The project challenged us to think critically about algorithm behavior in real-world systems, and our regular code reviews and testing sessions helped reinforce good communication and shared accountability. By the end, we had not only built a technically robust system, but also developed a deeper appreciation for what strong collaboration looks like in a development environment.

Did you have any challenges? If so, describe.

One of the most complex aspects of the project was merging traversal outputs from multiple algorithms into a shared export and visualization system. Each algorithm produced slightly different structures, which required careful standardization to ensure consistent formatting for both analysis and rendering. Additionally, managing stat-based edge weights introduced its own layer of complexity. To ensure that algorithms like Dijkstra and A* interpreted weights correctly, we had to design a custom transformation function and rigorously test it across all search types. Another ongoing challenge was maintaining database integrity while preserving repeated relationships across nodes, especially in the real-world dataset where cycles and multi-parent links were common. Coordinating these technical details across multiple branches and teammates while avoiding merge conflicts also tested our GitHub workflow discipline, but ultimately strengthened our ability to work collaboratively on shared systems. We overcame these challenges through clear communication, well-organized code ownership, and a strong commitment to group testing and documentation

If you were to start once again as a group, any changes you would make to the project and/or workflow?

If we were to start this project again, we would invest earlier in standardizing input/output formats across algorithms to reduce integration friction later in development. We would also establish clearer branch conventions and implement shared test scripts earlier in the process to avoid merge conflicts and manual verification steps. In terms of project planning, we might prioritize early visualization scaffolding to catch data formatting mismatches sooner, and spend more time aligning each algorithm's traversal signature with the export pipeline. While our workflow evolved into a strong system by the end, some of the integration and cleanup tasks could have been simplified with more upfront structure.

Comment on what each of the members learned through this process.

Adam –
"During this project my biggest takeaway was the effect of adding and implementing weights to a random walk. The addition of weights in the walk drastically reduced the runtime of the program while simultaneously making the walk more in line with a real-world application."

Carrie –
"One of the most valuable things I learned was how to manage integration across multiple contributors and algorithms. Coordinating traversal logic, file exports, and visualization in a shared system helped me develop stronger debugging and collaboration skills. I also gained a deeper understanding of how real-world recommendation structures behave—especially the importance of thoughtful graph weighting when using traditional search algorithms."

Kevin –
"I gained a lot more hands on experience with collaborating with group members on a large project. This helped me learn a lot about how to coordinate with group members and use GitHub to collectively organize our work. Another big takeaway for me was learning to take a complex real-world task and break it down into smaller, more manageable tasks that I can discuss with teammates."

Yepeth –

"My biggest takeaway from this project was the need for documentation when integrating the work of multiple programmers for large projects. I had used GitHub before on smaller projects, but the use and organization of GitHub for this project was much larger than any project I have had before. Getting more hands on experience was very valuable to me."