Applications of Computer Science
Personalized Museum Tour System

By Mor Taboh and Amit Ben Hemo

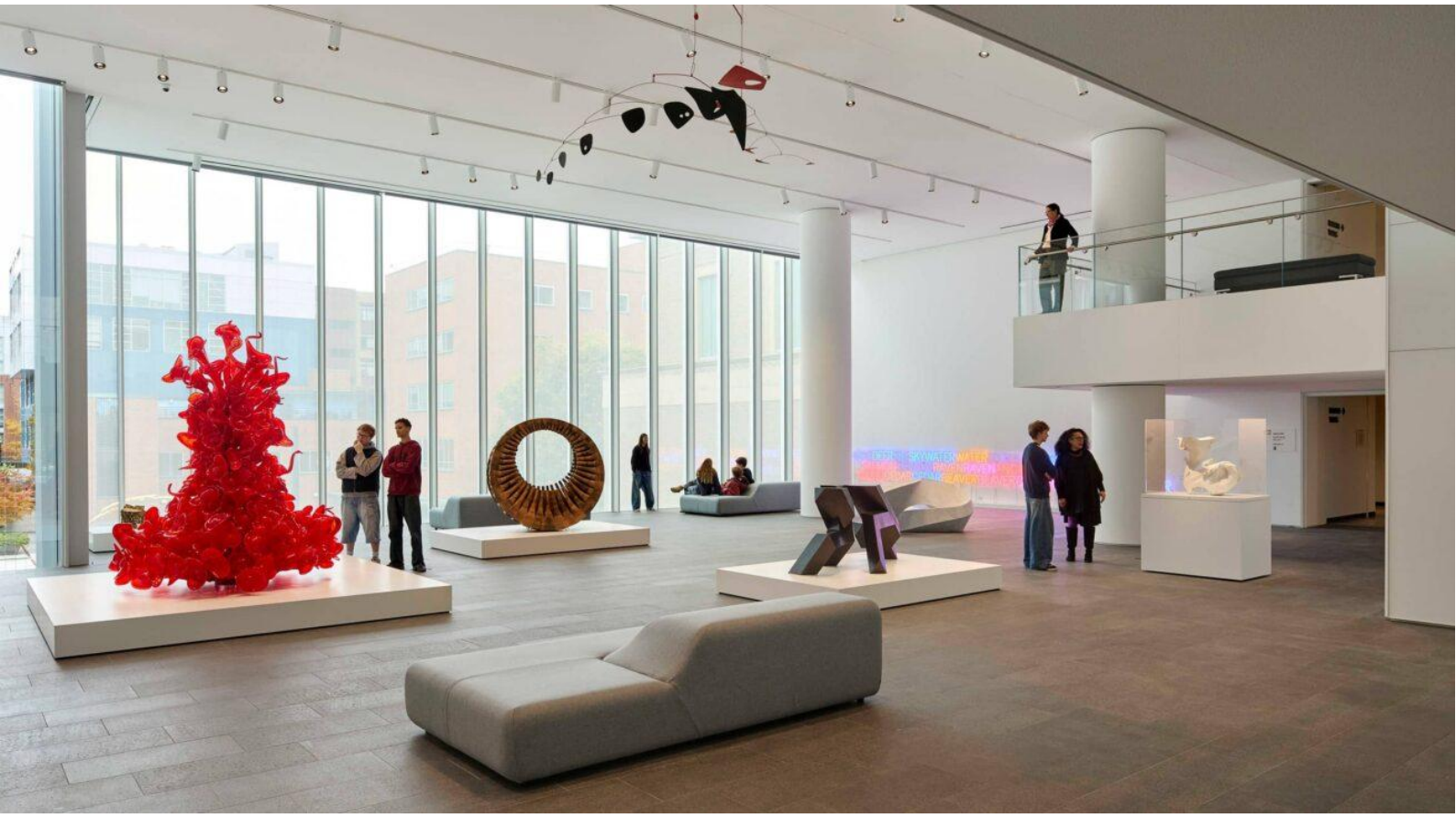


Table Of Contents

Introduction

Modern museums contain vast collections that often overwhelm visitors due to time limitations. Furthermore, visitors typically have different interests and preferences for certain types of exhibitions, such as paintings or sculptures. Designing a system that assists visitors in navigating such complex spaces while maximizing their overall experience is therefore a non-trivial computational problem. This project addresses the challenge of planning personalized museum routes that maximize visitor satisfaction within a strict time budget.

Problem Definition

The core problem addressed in this project is the computation of an efficient and personalized visitation route through a museum, under limited time and spatial constraints. The museum is modeled as a multi-floor environment composed of rooms, each containing multiple exhibits. Each room is characterized by its spatial location and a category name ("painting", "sculpture"), while each exhibit maintains a rating score and average viewing time derived from visitor feedback. Exhibit-level attributes are aggregated to form a room-level utility used for route planning.

Given a visitor's input: such as a total time budget, preferred exhibition type, mandatory rooms to include, and rooms to avoid - the system must determine an ordered sequence of rooms that can be feasibly visited within the available time. The route must account not only for walking distances, but also for the layout topology, including transitions between floors, which introduce additional movement costs. The objective is to maximize the visitors' overall satisfaction while respecting all constraints.

Formally, this problem can be viewed as a variant of the Orienteering Problem, a constrained optimization problem in which the goal is to select a path that maximizes the total score collected from visited nodes, subject to a time budget. This problem is known to be NP-hard, due to the combinatorial nature of possible sequences and the complexity of the spatial layout.
As a result, exact optimization approaches are impractical for real time applications, and heuristic methods are required to generate high quality solutions efficiently.

This project focuses on designing a practical and extensible solution that balances optimality and computational efficiency, making it suitable for interactive use in real world museum settings.

System Design

1. System Analysis

Before detailing the algorithmic solution, we define the core requirements that guided the system's design. These are divided into functional requirements (what the system must do) and non-functional requirements (system properties and constraints).

Functional Requirements

The system is designed to provide a personalized museum tour experience. The core functional requirements include:

- **User Profiling & Preferences:** The system must allow users to define their interests by assigning weights to different exhibit categories (e.g. sculpture, painting).
- **Constraint-Based Route Generation:** The system must generate an ordered list of rooms (a route) that:
    - Maximizes the total expected interest score.
    - Strictly adheres to the user's defined time budget.
    - Respects specific user constraints: Must-see rooms (mandatory inclusion), excluded rooms (blacklist), and maximal number of rooms to include.
- **Dynamic Feedback Collection:** The system must provide an interface for users to rate visited rooms (1-5 stars) and record the actual visit duration.
- **Visual Route Presentation:** The output must be presented clearly to the user, displaying the sequence of rooms and the estimated total time.

Non-Functional Requirements

The choice of algorithms and architecture was heavily influenced by the following performance and quality requirements:

- **Real-Time Responsiveness (Latency)**: The route calculation must be performed in sub-second time, even for complex museum layouts. This requirement precludes the use of exponential-time exact algorithms (like brute force) and necessitates a heuristic approach (Greedy Algorithm).
- **Constraint Adherence (Validity)**: The system must guarantee that the generated route never exceeds the user's time budget. A route that is "optimal" in score but requires more time than allocated is considered invalid.
- **Dynamic Adaptability**: The system must not be static. It requires a mechanism to update room scores and estimated dwell times based on crowd-sourced data, ensuring recommendations stay relevant and reflect real-time crowd conditions.
- **Scalability:** The routing algorithm should maintain performance stability $O(N^2)$ complexity as the number of rooms (N) increases

2. System Logic & Algorithm

Greedy Algorithm

A Greedy Algorithm is an algorithmic approach that builds up a solution piece by piece, always choosing the next piece that offers the most immediate benefit.
At each step, the algorithm makes the locally optimal choice.
While greedy algorithms don't guarantee an optimal solution for every problem, they are widely used as heuristics to approximate optimal solutions within a reasonable amount of time.

We chose this approach for the museum routing system due to the inherent complexity of the problem. Since finding the absolute perfect path in an Orienteering Problem is computationally expensive (NP-Hard), using exact algorithms would create unacceptable latency for a web application where users expect instant results.

In contrast, the Greedy approach allows for real time responsiveness.
By evaluating the next potential steps based on a specific cost function, calculating the ratio of a room's potential interest score against the time cost to reach it, the algorithm generates a logical and high-quality route efficiently, balancing visitor satisfaction with strict time constraints.

The Museum Routing Greedy Heuristic

In our implementation, in each step the algorithm selects a room to visit that maximizes the following cost function:

$$H(u,v) = \frac{score(v) \times Weight_{pref}}{Time(u,v) + Penalty(u,v) + Time_{dwell}(v)}$$

This function scores the decision to come visiting room v, from the current room (or entrance) u. The idea is that we want a good ratio such that the room's score is higher for the visitor, in less time to arrive there. Let us describe the function components:

- $score(v)$ - The average ratings of exhibits within room v derived from historical feedback.
- $Weight_{pref}$ - A personalization multiplier (e.g. 1.2) representing the visitor's interest in the category of the room v. This is determined by the user's preference vector. If a specific category preference is missing, a default value of 1 is applied
- $Time(u,v)$ – The estimated walking time between rooms u and v, based on the Euclidean distance.
- $Penalty(u,v)$ - A time penalty is added if u and v are on different floors, representing the additional time required to change floor.
- $Time_{dwell}(v)$ - The expected duration of the visit inside the room. This ensures the algorithm accounts for the total time consumption of the visit, not just the walking time.

We require the denominator to be less or equal to the visitor's remaining time budget, so that the choice would not take longer than the visitor's remaining time to tour the museum.

The Algorithm Flow

Input: Museum's layout – rooms and exhibits and a visitor with time budget and vector of preferences weights.

The Steps:

1. Initialize a route at the entrance as the current room denoted u.
2. Initialize a list of the rooms, maintaining the rooms that are not visited yet.
3. For each room in the list:
    a. Calculate total time cost
    $$Time(u,v) + Penalty(u,v) + Time_{dwell}(v)$$
    if it is larger than the time budget, skip this room.
    b. Calculate $H(u,v)$
4. Select the room with highest $H(u,v)$ value, remove it from the list
5. Subtract from time budget the time cost of the selected room and set current room to be the selected room.
6. Repeat the process until there are no available rooms.
7. **Time Complexity:** Since the algorithm iterates through the remaining unvisited rooms at each step, and performs constant time operations for each room, the time complexity is $O(N^2)$, ensuring immediate response time for typical museum sizes.

Handling User Constraints

In addition to the algorithm steps we described in previous section, our code implementation supports specific user-defined constraints to further tailor the generated route:

- **Rooms to avoid** - The users can name rooms they are not interested in visiting. When starting the route planning, these rooms are filtered out of the rooms list.
- **Must-see rooms** - The users can name rooms that must be included in their route. In our implementation we maintain a prioritized list of these rooms. Mandatory rooms are prioritized, provided they can be feasibly visited within the remaining time budget.
- **Max rooms** - The users can limit the numbers of rooms to include in their route. We implemented a stop condition that terminates the search once the route reaches this specified count, or earlier if the time budget is exceeded.

Dynamic Scoring and Feedback System

To ensure route recommendations remain accurate and resilient to sparse data, the system implements advanced statistical methods for updating room scores and estimated dwell times based on visitor feedback.

- **Bayesian Average for Room Scores:** Instead of a simple arithmetic mean, which can be misleading for rooms with few ratings, we employ a Bayesian Average. This method weighs the observed ratings against a "prior" belief. This ensures that a room with a single 5-star rating does not artificially outrank a popular room with a solid 4.8 average from hundreds of visitors. The formula used is:

$$score = \frac{W \times S_{prior} + \Sigma ratings}{W + N}$$

Where:

- W - The weight assigned to the prior score, e.g. W=10 equivalent to 10 dummy votes.
- S_{prior} – Default initial score.
- $\Sigma ratings$ – Sum of ratings given by the visitors
- N – The total number of visitors' rating.

- **Exponential Moving Average (EMA) for Dwell Time:** For estimating the time visitors spend in a room, we use an Exponential Moving Average. Unlike the score (which represents quality), dwell time is dynamic and affected by current crowd levels. EMA gives more weight to recent visits, allowing the system to adapt quickly to changes in visitor flow.
  Consider user feedback for a room with a view time of t_{new} and dwell time of the room t_{old}, the new dwell time is:

$$New\ Dwell\ Time = \alpha t_{new} + (1 - \alpha) t_{old}$$

  Where α is a smoothing factor between 0 and 1.

Implementation Description

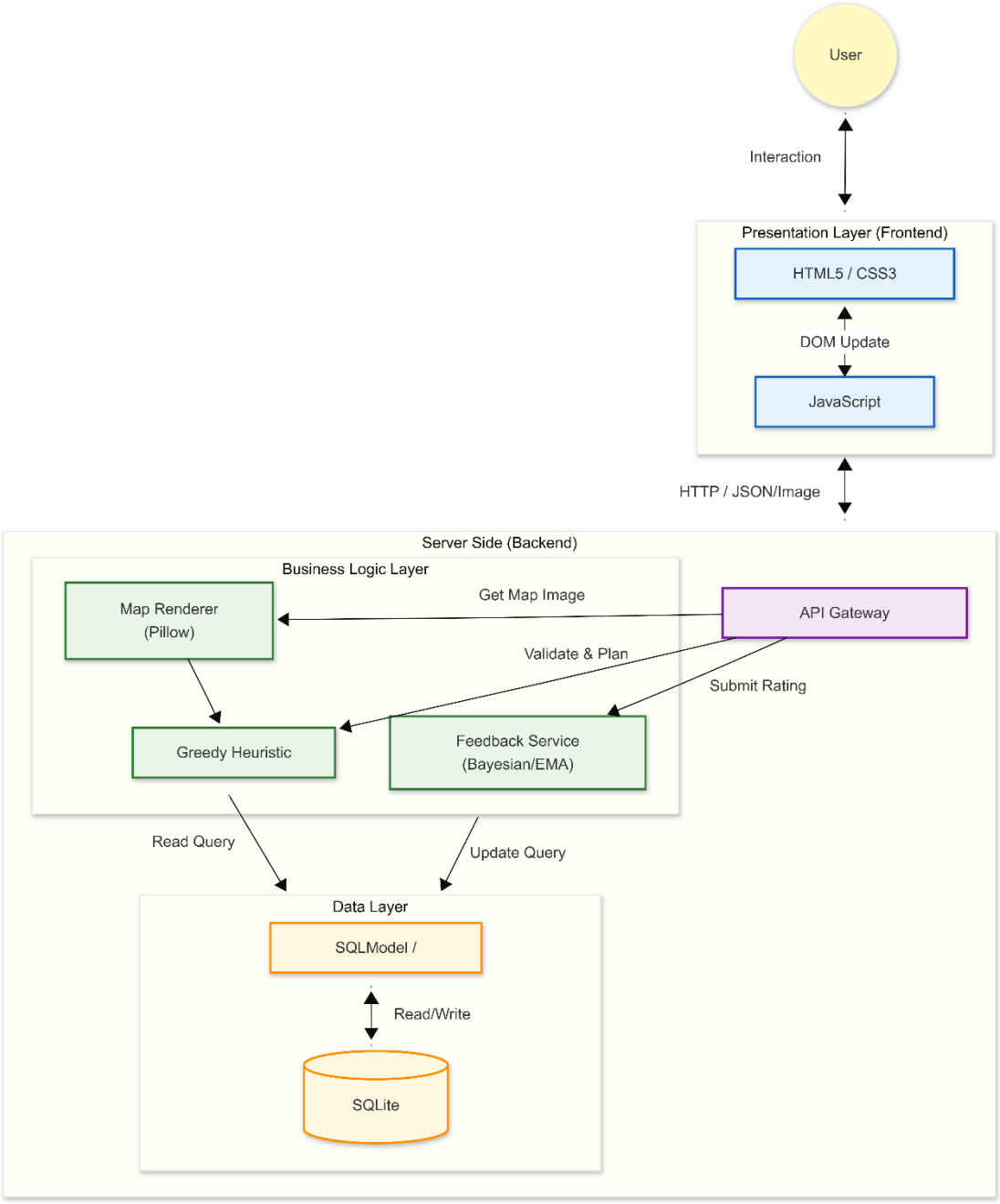
This chapter details the technological stack, system architecture, and data models used to build the museum routing system. The system was developed with a focus on high performance and ease of integration, following an API-First strategy.

1. Technology Stack

Component	Technology	Description
Language	Python 3.10+	Chosen for its rich ecosystem in data structures and rapid prototyping capabilities.
Web Framework	FastAPI	An asynchronous web framework that provides high performance and automatic documentation (Swagger UI).
Database	SQLite	A serverless, self-contained SQL database engine, ideal for rapid development.
ORM	SQLModel	A library that combines SQLAlchemy and Pydantic, ensuring strict type validation and intuitive database interactions.
Frontend Structure	HTML5 & CSS3	Used to build the lightweight dashboard interface, providing semantic structure and responsive styling.
Client Scripting	JavaScript (ES6+)	Used for asynchronous API communication (Fetch API) and dynamic DOM manipulation (e.g., feedback forms).
Visualization	Pillow (PIL)	Python imaging library used for server-side rendering of the route directly onto the museum floor plans.

2. System Architecture

The system follows a classic layered client-server architecture, designed to ensure separation of concerns and modularity. The application is divided into four main logical layers: the Presentation Layer (Client), the API Layer (Controller), the Business Logic Layer (Service), and the Data Persistence Layer.



1. Presentation Layer (Client)

- **Role:** Handles user interaction and visualization.
- **Implementation:** A lightweight web dashboard built with HTML5, CSS3 and JavaScript.
- **Functionality:**
    - Collects user inputs (time budget, preferences, constraints).
    - Communicates with the backend via asynchronous HTTP requests (Fetch API).

- Dynamically updates the DOM to display the itinerary and the generated route map image.
- Captures user feedback (ratings/dwell time) and sends it to the server.

2. API Layer (Controller)

- **Role:** Acts as the entry point for the backend, managing request handling and validation.
- **Implementation:** FastAPI.
- **Functionality:**
  - Defines RESTful endpoints (e.g., POST /route/plan-rooms, POST /feedback).
  - Uses Pydantic schemas to strictly validate incoming JSON payloads and outgoing response structures.
  - Routes valid requests to the appropriate services in the Logic Layer.

3. Business Logic Layer

- **Role:** Contains the core intelligence and algorithms of the system.
- **Implementation:** Python modules.
- **Key Components:**
  - **Routing Engine:** Implements the Greedy Heuristic algorithm. It calculates weights based on user preferences and computes the optimal path under time constraints.
  - **Feedback Service:** Handles the mathematical updates of room scores (Bayesian Average) and dwell times (Exponential Moving Average).
  - **Map Renderer:** A dedicated service using Pillow (PIL) that draws the calculated polyline route and markers directly onto the museum's floor plan images, returning a static image to the client.

4. Data Persistence Layer

- **Role:** Manages the storage and retrieval of persistent data.
- **Implementation:** SQLite wrapped with SQLModel (ORM).
- **Functionality:**
  - Stores static graph data (Rooms, Exhibits, Coordinates).
  - Persists dynamic data (User sessions, collected feedback logs).
  - The ORM abstracts SQL queries, allowing the logic layer to interact with Python objects (Room, Exhibit) rather than raw database rows.

3. Database Scheme

The database is designated to support efficient querying of spatial data and exhibit metadata. The core entities are:

1. **Room:** Represents a physical space in the museum and acts as a node in the navigation graph.
    - **Spatial Attributes:** Position (x, y, floor) define the room's centroid in the coordinate system.
    - **Navigation Attributes:** dwell time is a derived value representing the average time required to visit the room, aggregated from its exhibits.
2. **Exhibit:** Represents a specific artwork or display. This table holds the dynamic statistical data used by the routing system.
    - **Static Data:** name, type (painting/sculpture), and local position (x, y) (offset within the room).
    - **Dynamic Stats (Bayesian & EMA):**
        - score: The current quality score (1-5), calculated via the Bayesian Average formula.
        - prior / prior weight: Constants used to stabilize the score for new exhibits (Cold Start handling).
        - average view time : The Exponential Moving Average (EMA) of dwell time.
3. **Feedback:** A transaction log storing raw user interactions.
    - **Purpose:** Enables the recalculation of exhibit scores.
    - **Fields:** Stores the rating (1-5), actual view time measured, and a timestamp for time-series analysis.
4. **Route Log:** A logging table used for system analytics and debugging.
    - **Purpose:** Records every route generation request to analyze user behavior and algorithm performance.
    - **Serialization:** Complex structures such as user preferences and the resulting path are stored as serialized JSON strings to maintain compatibility with SQLite.

4. Project Structure

The project is organized into a modular directory structure that separates the backend logic, database models, and frontend assets. This organization allows easy maintenance.

Directory Tree

```
Museum_project/
├── src/
|   ├── api/              # Backend Core (FastAPI)
|   |   ├── __init__.py
|   |   ├── main.py         # Application entry point & configuration
|   |   ├── routes.py       # API endpoints & Greedy Algorithm logic
|   |   ├── models.py       # SQLModel database schemas (Room, Exhibit, Feedback)
|   |   ├── database.py     # Database connection & session management
|   |   └── services.py     # Business logic (Bayesian Score, EMA)
|   |
|   └── web/              # Frontend Presentation
|       ├── map.html        # The Client-Side Dashboard (JS/HTML/CSS)
|       ├── floor0.png      # Ground floor plan image
|       └── floor1.png      # First floor plan image
|
├── scripts/              # Utility Scripts
|   └── seed.py           # Data seeding script (JSON -> SQLite)
|
├── instances/
|   └── floor_two_rooms_extended.json            # Initial dataset (Rooms/Exhibits)
├── app.db              # SQLite Database file (generated)
└── requirements.txt        # Python dependencies
```

Module Description

- **src/api/**: The heart of the application.
    - **routes.py**: Contains the RESTful controllers. This file also hosts the greedy heuristic engine, which processes the routing requests, and the map renderer logic that uses Pillow to draw the path.
    - **models.py**: Defines the data structure using SQLModel. It serves as the single source of truth for both the database schema and the API response shapes.
    - **services.py**: Encapsulates isolated logic, such as the apply_feedback() function that updates the Bayesian scores and dwell time averages.
    - **main.py**: Initializes the FastAPI app, mounts the static files directory, and configures the database on startup.
- **src/web/**: Contains the presentation layer.
    - **map.html**: A standalone single-page interface. It communicates with the API via fetch() requests to send user preferences and display the returned route map.
    - **Floor Plans (.png)**: Static assets used by the server-side renderer to generate the route overlays.
- **scripts/**:
    - **seed.py**: A dedicated automation script used to populate the database with initial room and exhibit data from floor_two_rooms_extended.json.

Demonstration

This chapter demonstrates the system's capabilities through core scenarios, ranging from basic routing to complex constraint handling and real time adaptation.

Experimental Setup

The demonstration runs on a simulated dataset representing a two floors museum layout containing:

- **12 Rooms:** Distributed across two floors (Floor 0 and Floor 1).
- **37 Exhibits:** Categorized into "Painting" and "Sculpture".

Database State: Below is a representative sample of the data stored in the rooms and exhibits tables, showing the diversity of locations and categories:

Rooms:

	id	name	theme	pos_x	pos_y	pos_z	dwell_min
1	R101	Impressionists	painting	10	4	0	4.533333333333334
2	R102	Marble Hall	sculpture	5	12	0	4.1
3	R103	Modern Shapes	sculpture	2	6	0	3.2138666666666666
4	R104	Renaissance Court	painting	12	3	0	6.133333333333333
5	R105	Ancient Pottery	sculpture	4	2	0	3
6	R106	Landscape Hall	painting	15	5	0	5
7	R201	Blue Gallery	painting	14	9	1	4.733333333333333
8	R202	Bronze Atrium	sculpture	6	15	1	4.3
9	R203	Abstract Wing	painting	3	10	1	3.6933333333333334
10	R204	Photography Nook	painting	11	13	1	4.266666666666667
11	R205	Pop Art Corner	painting	1	12	1	3.692
12	R206	Glasswork Gallery	sculpture	9	17	1	4.633333333333334

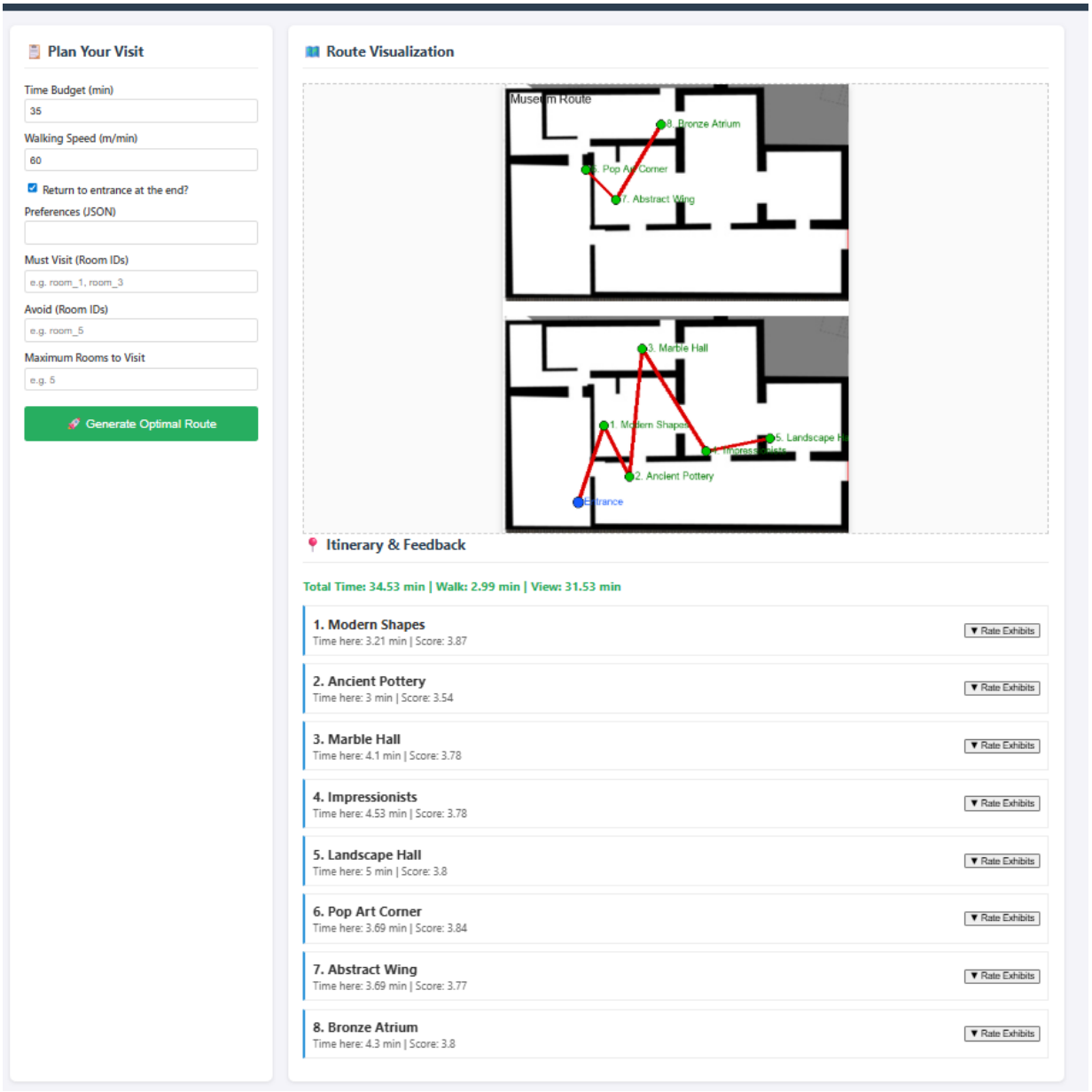
Exhibits:

	name	type	local_x	local_y	prior	prior_weight	rating_sum	rating_count	score	avg_view_time...	view_time_count
R201	Blue Room	painting	0	0	3.8	20	0	0	3.8	5	0
R102	Marble Hero	sculpture	0	0	3.8	20	2	1	3.71428571...	4.4	1
R101	Sunset	painting	0	0	3.8	20	4	1	3.80952380...	4.4	1
R101	Water Lilies	painting	0	0	3.8	20	4	1	3.80952380...	5.200000000000001	1
R101	Paris Street	painting	0	0	3.8	20	2	1	3.71428571...	4	1
R102	Roman Athlete	sculpture	0	0	3.8	20	0	0	3.8	4	0
R102	Venus Fragment	sculpture	0	0	3.8	20	4	1	3.80952380...	3.2	1
R102	Marble Lion	sculpture	0	0	3.8	20	0	0	3.8	4.8	0
R103	Wire Form No. 2	sculpture	0	0	3.8	20	15	3	3.95652173...	2.9216	3
R103	Stone Curve	sculpture	0	0	3.8	20	5	1	3.85714285...	3.760000000000000...	1
R103	Minimal Totem	sculpture	0	0	3.8	20	4	1	3.80952380...	2.960000000000000...	1
R104	Madonna & Child	painting	0	0	3.8	20	0	0	3.8	6.5	0
R104	Courtly Hunt	painting	0	0	3.8	20	0	0	3.8	5.7	0
R104	Golden Triptych	painting	0	0	3.8	20	0	0	3.8	6.2	0
R201	Blue Horizon	painting	0	0	3.8	20	0	0	3.8	4.3	0
R201	Cobalt Forest	painting	0	0	3.8	20	0	0	3.8	4.9	0
R202	Bronze Runner	sculpture	0	0	3.8	20	0	0	3.8	4.4	0
R202	Sun Mask	sculpture	0	0	3.8	20	0	0	3.8	3.9	0
R202	Armored Figure	sculpture	0	0	3.8	20	0	0	3.8	4.6	0
R203	Red on Grey	painting	0	0	3.8	20	2	1	3.71428571...	3.679999999999999...	1

Scenario A: Constraint-Based Routing

In this scenario, we simulate a general visitor with a strict time budget of 35 minutes and no specific preferences. The goal is to maximize the score within the time limit.

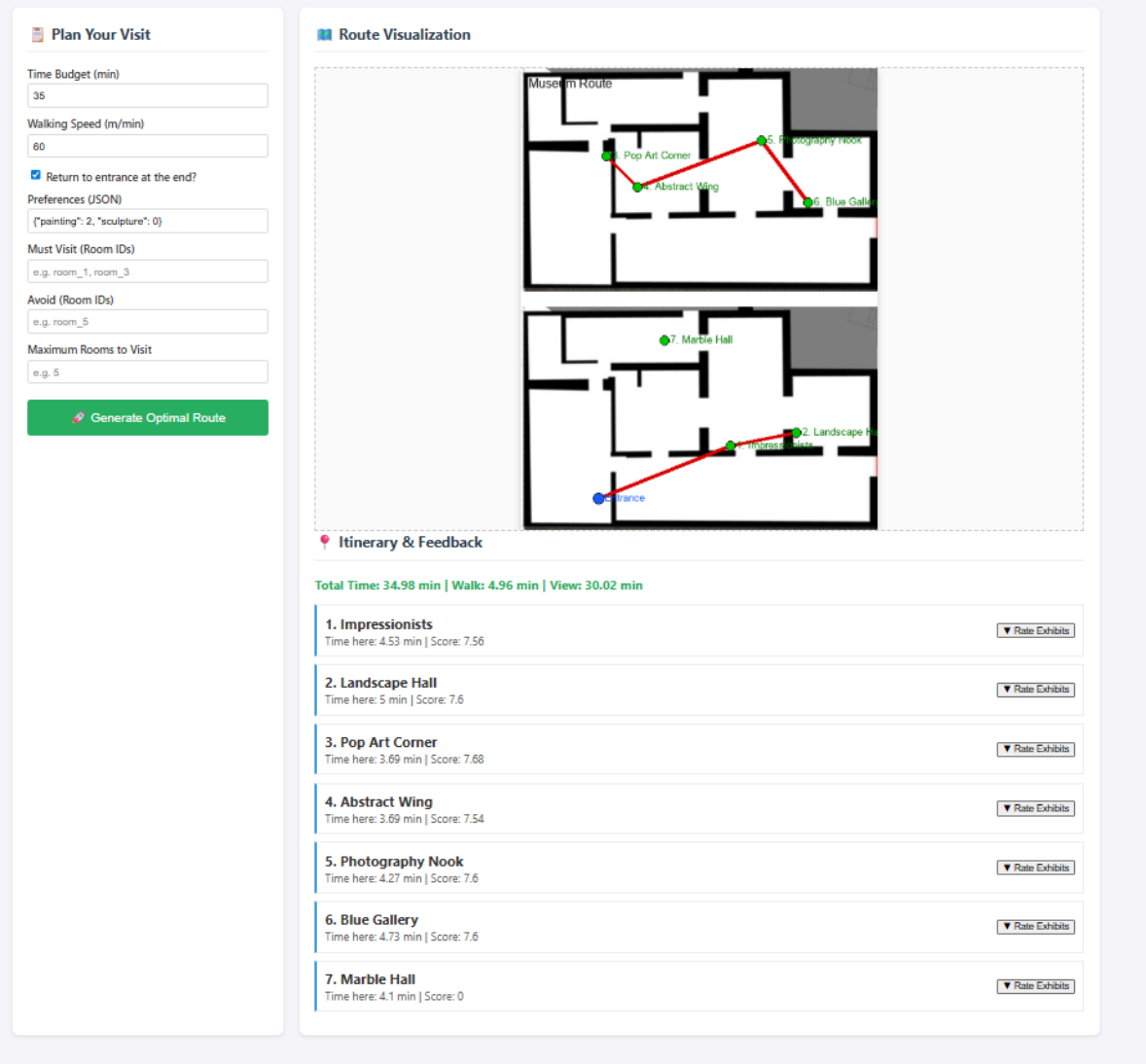
- **Input:** Budget time of 35 minutes
- **Result:** The algorithm executes the greedy heuristic, selecting high score rooms with minimized travel costs.



Scenario B: Personalized Experience

In this scenario, we demonstrate the weighting mechanism. We simulate an "Art Enthusiast" with a strong preference for Paintings (weight=2) and zero interest in Sculpture (weight=0).

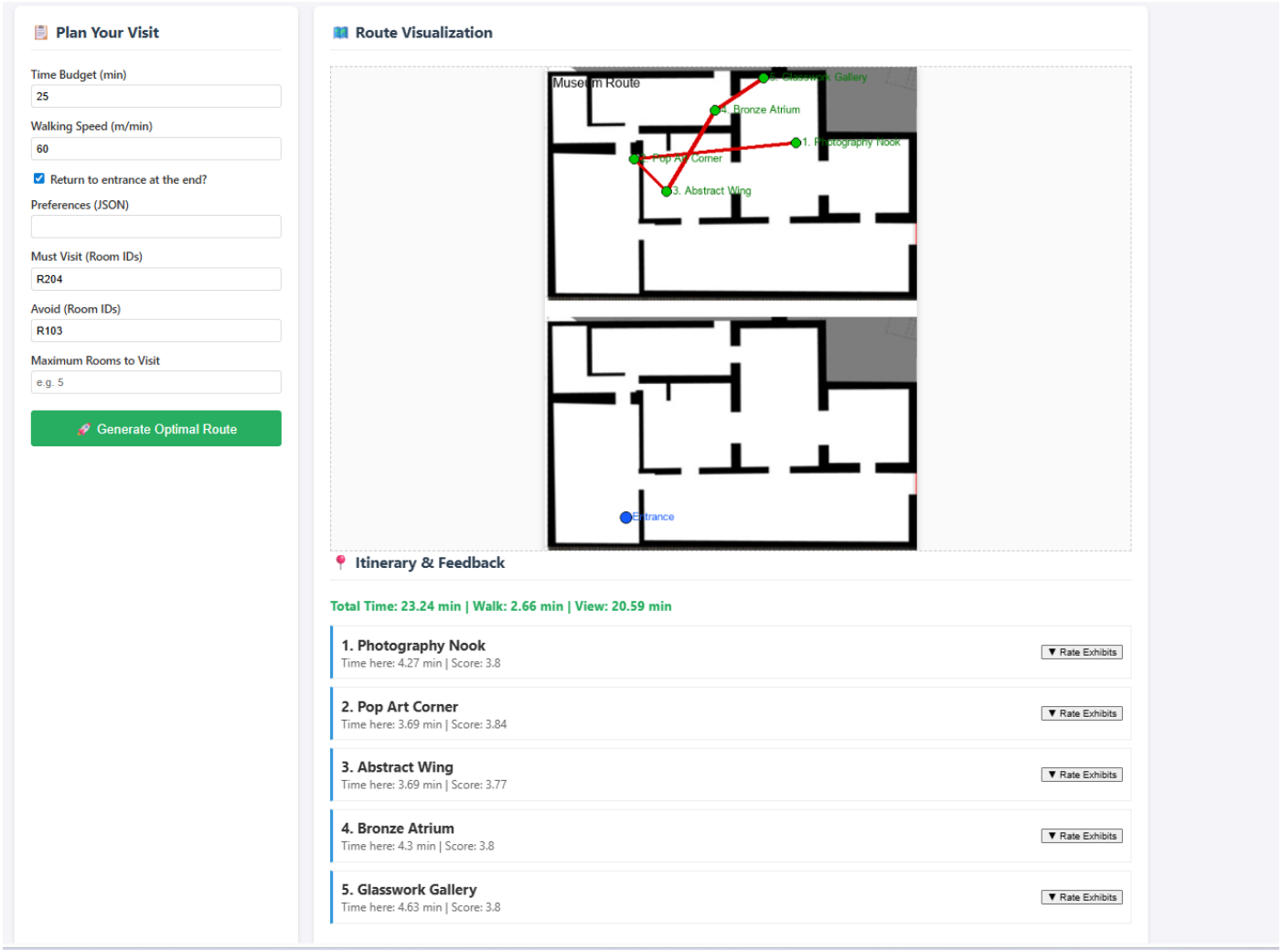
- **Input:** Budget time of 35 minutes, prefs={"painting": 2.0, "sculpture": 0.0}
- **Result:** The route trajectory changes significantly. The algorithm bypasses nearby sculptures rooms to reach distant painting galleries, proving that the weighting mechanism successfully overrides simple proximity.



Scenario C: Complex Constraints

This scenario tests the system's robustness by applying strict inclusion and exclusion rules. The user demands to visit a specific remote room ("Must Visit") while blocking a popular central room ("Avoid").

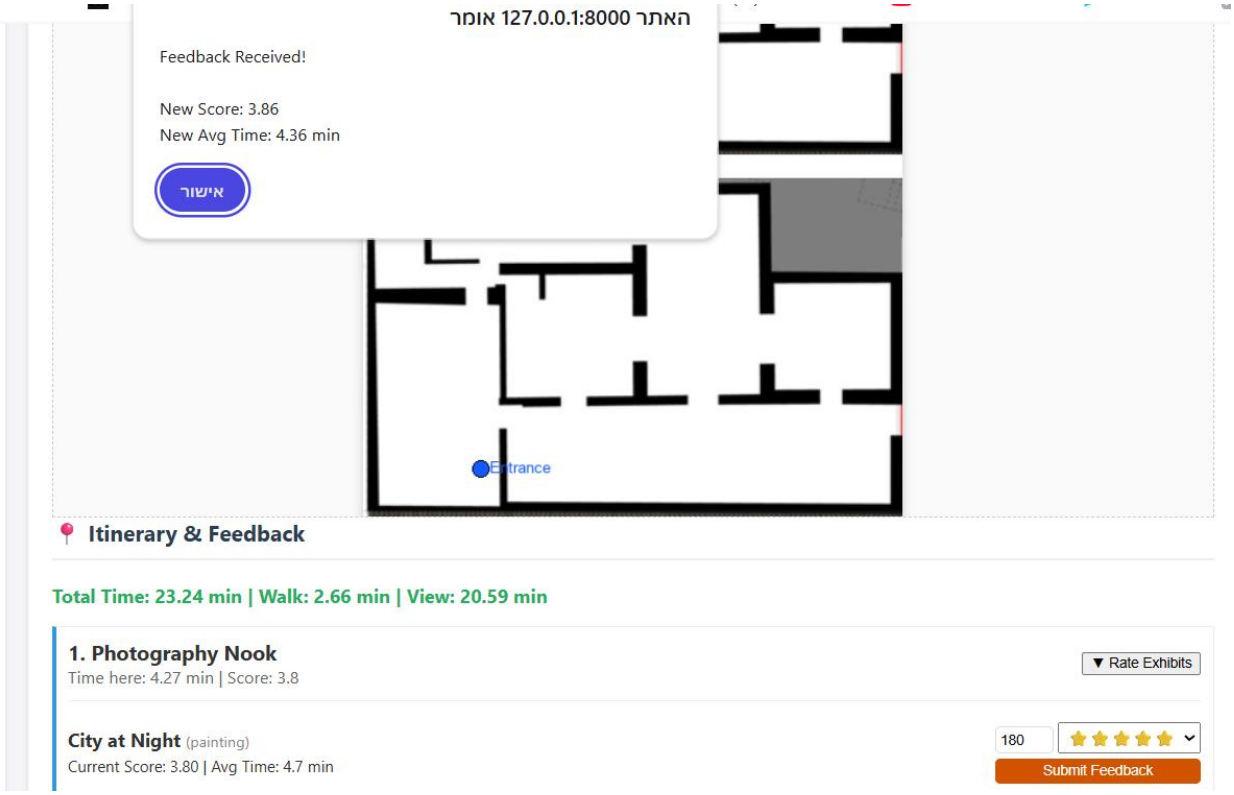
- **Input:**
  - Time Budget of 25 minutes
  - Must Visit=['R204'] (A distant Photography Nook on Floor 1).
  - Avoid=['R103'] (The nearby Modern Shapes).
- **Result:** As shown below, the algorithm forces a path to R204 despite the travel cost, ensuring the "Must" constraint is satisfied first. Additionally, R103 is completely skipped, demonstrating the filtering capability.



Scenario D: Feedback Submission

Finally, we demonstrate the system's ability to learn from crowd-sourced data.

- **Action:** User rates an exhibit (Current Score: 3.80) with 5 stars, and 180 seconds view time
- **Result:** The system processes the POST request, updates the database, and returns the new Bayesian score (3.86) and new average view time (4.36 min). The updated data will immediately influence the route generation for the next user.



Summary and Conclusions

This project dealt with the problem of museum touring under constraints by developing a smart, personalized navigation system. We successfully implemented a full-stack solution that combines a Python based algorithmic engine with a lightweight web interface. The system meets all defined objectives: it generates feasible routes under strict time constraints, respects user preferences (must-visit/avoid), and employs a self-correcting feedback loop using Bayesian statistics to refine recommendations over time.

Key Conclusions

- **Efficiency of Heuristics:** The decision to use a greedy heuristic approach proved effective for this use case. While it does not guarantee a mathematically perfect solution (global optimum), it provides a "near-optimal" route in negligible time, which is critical for a real time web application.
- **The Power of Feedback:** Implementing the Bayesian Average and Exponential Moving Average (EMA) transformed the system from a static map into a dynamic platform. This ensures that the system remains relevant by adapting to changing trends and visitor behaviors without manual intervention.
- **Server-Side Rendering:** Offloading the map generation to the server (using Pillow) simplified the client-side logic significantly, ensuring consistent visualization across different devices and reducing frontend battery consumption.

Limitations

While functional, the current system has inherent limitations:

- **Local Optimization:** Due to the greedy nature of the algorithm, the system might choose a locally optimal step (e.g., a nearby room) that prevents visiting a higher-value room later in the path due to exhaustion of time.
- **Simplified Navigation:** The current movement model assumes direct paths between room centroids. It does not fully account for physical obstacles (walls, crowds, location of stairs and elevators) or complex navigation within large halls.
- **Single-User Focus:** The routing engine optimizes for the individual user without considering global crowd congestion. If all users request a route simultaneously, the system might send everyone to the same popular room, creating bottlenecks.

Installation and Running

Follow these steps to set up the environment, populate the database, and launch the application.

Prerequisites

- **Python 3.10** or higher installed
- **pip** (Python package manager).

Installation

1. Clone the Repository:
   git clone https://github.com/amit220105/computer_sience_project.git
   cd computer_sience_project
2. Setup virtual environment:
   python -m venv .venv
   .\.venv\Scripts\activate
3. Install the required dependencies:
   pip install -r requirements.txt
4. Run the app:
   uvicorn src.api.main:app
   The server starts at address such as: " http://127.0.0.1:8000"
5. Open on browser:
   http://127.0.0.1:8000/map