جامعة الأخوين

ⵜⴰⵙⴷⴰⵡⵉⵜ ⵏ ⵓⵅⴰⵡⴰⵢⵏ

AL AKHAWAYN UNIVERSITY

School of Science & Engineering

A Knowledge Graph-Based Framework for Job-Oriented Learning Path Generation

Capstone Design Final Report

Spring 2025

Yassine Jebbouri

Supervised by:

Dr. Violetta Cavalli-Sforza

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

Abbreviation	Definition
A*	A-Star Search
AUC	Area Under the Receiver Operating Characteristic Curve
ACL	Association for Computational Linguistics
AI	Artificial Intelligence
API	Application Programming Interface
AP	Average Precision
CSV	Comma–Separated Values
CPR	Concept Prerequisite Relations
DAG	Directed Acyclic Graph
DP	Dynamic Programming
EMNLP	Empirical Methods in Natural Language Processing
FPR	False Positive Rate
GAT	Graph Attention Network
GNN	Graph Neural Network
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
LLM	Large Language Model
MLOps	Machine Learning Operations
NLP	Natural Language Processing
OWL	Web Ontology Language
PCA	Principal Component Analysis
PR	Precision–Recall
ReLU	Rectified Linear Unit
RDF	Resource Description Framework
ROC	Receiver Operating Characteristic
SBERT	Sentence-BERT
SCC	Strongly Connected Component
SRS	Software Requirements Specification
UI	User Interface
UX	User Experience

ABSTRACT (ENGLISH)

This capstone project presents a knowledge graph-based framework designed to generate personalized learning paths for individuals aiming to transition into AI and data science careers. The system takes as input the user's existing skill set and a target job title. It then constructs a customized learning plan using a graph of prerequisite relationships extracted from academic datasets (CPR and LectureBank) and real-world job requirements scraped from hiring platforms. The graph integrates concept dependencies, job-skill mappings, and skill definitions. To ensure coherence, the system applies order embeddings for edge scoring, predicts missing prerequisite links, and removes cycles. A layered schedule is finally generated, guiding learners from their current skills to the required competencies. This approach provides a scalable and explainable solution to job-oriented skill development.

Keywords: Knowledge Graph, Learning Path, Order Embedding, AI Careers, Skill Recommendation, Graph Neural Networks, Neo4j

RESUMÉ (FRENCH)

Ce projet de fin d'études propose un cadre basé sur les graphes de connaissances pour générer des parcours d'apprentissage personnalisés destinés aux personnes souhaitant intégrer les métiers de l'intelligence artificielle et de la science des données. Le système prend en entrée les compétences actuelles de l'utilisateur ainsi que le poste ciblé, puis élabore un plan de formation sur mesure en s'appuyant sur un graphe de relations de prérequis, construit à partir de jeux de données académiques (CPR et LectureBank) et d'exigences professionnelles issues d'annonces d'emploi récentes. Le graphe intègre les dépendances entre concepts, les liens emploi-compétence et les définitions de chaque nœud. Le système applique des embeddings d'ordre pour évaluer les relations, prédit les liens manquants, supprime les cycles, et génère un calendrier hiérarchisé de progression. Cette approche fournit une solution évolutive et explicable pour guider le développement de compétences en fonction des métiers visés.

Mots clés: Graphe de connaissances, Parcours d'apprentissage, Embedding d'ordre, Métiers de l'IA, Recommandation de compétences, Réseaux de neurones graphiques, Neo4j

1 Introduction

In recent years, the rapid growth of artificial intelligence (AI) and data science has led to a significant transformation in the job market. These fields now require professionals with specialized technical skills and a strong foundation in computational thinking. However, many learners: students, recent graduates, or professionals from other domains, struggle to navigate the complex landscape of prerequisite knowledge needed to access these careers. Standard educational platforms and career counseling tools often fail to offer personalized guidance aligned with industry requirements.

This project addresses this gap by proposing a structured and explainable approach to skill acquisition tailored to a user's background and career goal. We develop a knowledge graph-based system that maps out learning paths between a user's current competencies and the requirements of a target AI or data science job. Unlike traditional recommendation systems, our method incorporates both academic concept hierarchies and real-world job data to provide context-aware guidance.

At the core of the system is a directed concept graph built from curated datasets (CPR and LectureBank) and enriched with up-to-date job descriptions scraped from a specialized platform. Each node represents a concept or skill, and each edge captures a prerequisite relationship. To ensure structural soundness and accuracy, the graph undergoes refinement through order embedding-based scoring, cycle pruning, and link prediction. The result is a directed acyclic graph (DAG) that enables path computation and personalized curriculum generation.

This report details the methodology, implementation, and potential applications of the system. It demonstrates how graph-based machine learning can be leveraged to support career transitions in a dynamic field by providing learners with clear, actionable learning trajectories.

2 Problem Statement

Despite the increasing availability of online educational resources, learners aiming for careers in AI and data science face three key obstacles: (1) fragmented access to reliable information about industry expectations, (2) limited systems that connect personal skill profiles to job-specific learning paths, and (3) a lack of clear guidance on the order in which to acquire technical skills.

Most existing solutions either offer generic course recommendations or focus solely on job matching without explaining how to close the skill gap. They fail to capture the hierarchical nature of technical knowledge, where some skills serve as foundations for others and rarely align learning resources with the nuanced requirements of AI-related roles.

This capstone addresses these shortcomings by developing a system that generates personalized, dependency-aware learning paths. The core challenge is to integrate structured academic knowledge and real-time labor market data into a single, traversable knowledge graph. The solution must also ensure correctness (by preserving prerequisite logic), flexibility (to adapt to diverse learner profiles), and explainability (so users understand each recommended step).

3 Project Specification

This project develops a knowledge-graph framework that integrates academic prerequisite data and real-world job postings to produce personalized learning paths for aspiring AI and data science professionals. By extracting and linking concepts, hard skills, technologies, and job titles in Neo4j, we refine the graph through cycle removal and order-embedding–based link prediction. The resulting directed acyclic graph is then traversed with score-weighted algorithms to generate tailored curricula. Work proceeded over a 16-week timeline: data collection and preprocessing (Weeks 1–4), graph construction and enrichment (Weeks 5–8), algorithm development (Weeks 9–12), user interface integration (Weeks 13–14), and final testing and documentation (Weeks 15–16).

Key deliverables include the Neo4j knowledge graph, trained order-embedding and pathfinding models, a web or notebook interface for skill input and curriculum output, and comprehensive documentation covering system design and usage. We leverage an ordinary workstation, open-source tooling (Neo4j, Python, PyTorch), and API access for concept definitions. Principal risks include data quality, computational scalability, and user adoption which can be addressed via continuous data validation, performance profiling, and iterative UX refinements.

3.1 Software Engineering Approach

We follow an Agile methodology emphasizing incremental development and regular supervisor feedback. Our iterative process evolved through:

- **Increment 1 (Weeks 1–4)**: Gathered data and requirements; received supervisor feedback through weekly meetings to refine requirements and adjust priorities.
- **Increment 2 (Weeks 5–8)**: Built functioning graph construction with cycle detection; demonstrated to supervisor, incorporated feedback, and made adjustments to improve functionality.
- **Increment 3 (Weeks 9–14)**: Enhanced product with pathfinding algorithms and link-prediction; supervisor review led to refinements in algorithm implementation and evaluation metrics.
- **Increment 4 (Weeks 12–16)**: Integrated all components into web application; conducted final review sessions with supervisor to optimize performance and finalize documentation.

Each increment concluded with a supervisor review meeting where working functionality was demonstrated, feedback was collected, and adjustments were planned for the next iteration.

3.2 Requirements

3.2.1 User Stories

1. As a user, I want to create and manage my account so I can access personalized features
2. As a job seeker, I want to manage my skills profile so I can track my capabilities
3. As a job seeker, I want to explore AI/data science opportunities so I can find relevant positions

4. As a learner, I want personalized learning paths so I can efficiently acquire job-relevant skills

5. As a user, I want interactive skill visualizations so I can understand prerequisite relationships

6. As a user, I want intuitive navigation so I can easily access all system features

3.2.2 Non-Functional Requirements

1. **Performance**: Page loads under 3 seconds, API responses under 2 seconds

2. **Security**: Firebase authentication, secure data handling

3. **Usability**: Responsive design, consistent UI across pages

4. **Browser Compatibility**: Works on modern browsers (Chrome, Firefox, Safari, Edge)

4 STEEPLE Analysis

4.1 Social Factors

The project promotes equitable access to high-demand careers in artificial intelligence and data science by providing personalized learning paths. This can help reduce social inequality by enabling individuals from diverse academic or socio-economic backgrounds to pursue technical roles, even if they lack traditional university credentials.

4.2 Technological Factors

The project leverages modern technologies such as knowledge graphs (Neo4j), natural language processing (SkillNER), and machine learning models (order embeddings) to generate accurate and explainable learning paths. It aligns with current trends in AI-driven personalization and graph-based knowledge representation.

4.3 Economic Factors

By helping individuals acquire job-relevant skills more efficiently, the system can contribute to lowering unemployment in technical fields and reducing the cost and time associated with career transitions. It may also inform training institutions on which skill sets are most in demand, improving curriculum design.

4.4 Environmental Factors

The project has minimal direct environmental impact (it uses a reasonable amount of energy for computations), as it is a software-based system. However, it encourages digital and remote learning, which can reduce the need for physical infrastructure and travel associated with traditional education pathways.

4.5 Political Factors

The system supports national and international initiatives that promote digital transformation, lifelong learning, and upskilling. It aligns with educational goals in many countries to foster AI talent and increase competitiveness in the digital economy.

4.6 Legal Factors

The project ensures compliance with data protection and intellectual property regulations by using publicly available datasets and scraping only publicly posted job information. Any use of third-party APIs, such as OpenAI, adheres to their respective terms of service.

4.7 Ethical Factors

Ethically, the project supports inclusive access to education and professional growth. Care has been taken to prevent bias in learning path recommendations by grounding them in structured prerequisite logic and current job requirements rather than opaque algorithmic decisions.

5 Engineering Standards

This project adheres to a set of engineering standards that guide the development of software systems involving artificial intelligence, data structures, and user-facing applications. These standards ensure that the project maintains quality, interoperability, and ethical compliance throughout its lifecycle.

5.1 IEEE 830-1998 – Software Requirements Specification

This standard provides guidelines for writing software requirements specifications (SRS). It ensures that functional and non-functional requirements for the system such as learning path accuracy, user input validation, and graph consistency are clearly defined and verifiable. It influenced the early planning and scoping stages of this capstone.

5.2 ISO/IEC/IEEE 12207 – Software Life Cycle Processes

The project follows this international standard to structure the software development process. It provides a framework for planning, development, testing, and maintenance. Even within the constrained timeline of a capstone project, this standard supports disciplined iteration from requirement definition to final evaluation.

5.3 W3C RDF and OWL – Semantic Web Standards

Although Neo4j uses the labeled property graph model, the project draws conceptual inspiration from the W3C's RDF (Resource Description Framework) and OWL (Web Ontology Language) standards. These standards are central to structuring and querying knowledge representations in a way that supports semantic reasoning and extensibility goals aligned with our use of prerequisite-based learning graphs.

5.4 ISO/IEC 25010 – System and Software Quality Models

This standard helps evaluate software quality across dimensions such as usability, reliability, performance efficiency, and maintainability. It informs both internal testing and final evaluation, especially as the system must be intuitive for non-expert users who rely on the generated learning paths for career planning.

5.5 ISO/IEC 2382 – Information Technology Vocabulary

To ensure terminological consistency across the graph's definitions and job-related concepts, this standard supports the use of controlled vocabularies in information systems. It provides a reference when labeling concepts, definitions, and skill types within the knowledge graph.

5.6 IEEE 7000 Series – Ethical Considerations in AI Systems

As this project involves automated recommendations affecting educational and career outcomes, ethical standards must be upheld. The IEEE 7000 series, particularly IEEE 7001 (Transparency of Autonomous Systems) and IEEE 7003 (Algorithmic Bias Considerations),

are relevant to ensuring that the system is transparent, explainable, and designed with fairness in mind.

6 Logic Model Framework

6.1 Target Population

The primary target population of this project includes university students, early-career professionals, and individuals seeking a career transition into AI and data science fields. These users typically possess partial technical backgrounds or foundational knowledge in related domains but lack a clear roadmap to acquire the full set of competencies required for specific job roles.

6.2 Underlying Assumptions

This project is based on the following key assumptions:

- Users can accurately self-report their existing skill sets.
- Prerequisite relationships among concepts are meaningful and consistent across academic and industry contexts.
- Job descriptions provide a reasonably accurate reflection of real-world skill requirements.
- The graph structure can be refined through machine learning to reflect actual learning dependencies.
- Users benefit from structured and explainable guidance in acquiring new skills.

6.3 Resources/Challenges

Resources:

- Publicly available academic datasets (e.g., CPR, LectureBank).
- Web scraping tools (Selenium, BeautifulSoup).
- Open-source libraries (Neo4j, SKILLNER, PyTorch).
- Access to OpenAI API for concept definition generation.

Challenges:

- Ensuring accuracy and consistency in scraped job descriptions.
- Modeling complex and sometimes ambiguous skill dependencies.
- Balancing explainability with model performance.
- Managing computational complexity for large-scale graph operations.

6.4 Activities

- Collecting and preprocessing prerequisite relationship data from academic sources.
- Scraping real-world job postings and extracting required skills.
- Constructing and embedding a knowledge graph using Neo4j.
- Training order embedding models to score edges and detect inconsistencies.
- Predicting missing links for hard skills and removing cycles in the graph.
- Generating personalized learning paths for users.

6.5 Outputs of the Project

- A cleaned and structured knowledge graph of AI and data science concepts.
- A set of representative AI-related job roles with mapped hard skill requirements.
- An embedding-based scoring model to evaluate prerequisite consistency (explained in section: 8.4.2.
- A web-based or notebook-based interface to generate learning paths.
- Documentation and guidelines for further enhancement and deployment.

6.6 Outcomes

Short-term outcomes:

- Increased clarity for users about the skills they need to acquire.
- Personalized and explainable learning plans that reduce time-to-employment.

Long-term outcomes:

- Broader accessibility to technical career pathways in AI and data science.
- Data-driven approaches to educational planning and curriculum development.
- A foundation for scalable, adaptive learning technologies based on knowledge graphs.

7 Literature Review

7.1 Graph-Based Learning-Path Generation

Early efforts such as LectureBank [12] and ConceptNet [9] established the value of representing curricula and common-sense knowledge as graphs, where edges denote "prerequisite" or "related" links. These systems excel at visualizing concept hierarchies but stop short of producing user-specific sequences: they ask learners to explore rather than guide them step by step. More recent platforms have layered on collaborative-filtering or popularity-based recommendations [7], yet these lack both interpretability and an explicit model of skill dependency. In contrast, our framework unites academic prerequisites with real-world job requirements to deliver tailored, explainable learning trajectories.

7.2 Embedding and Loss Methods for Directed Graphs

Embedding a directed graph demands not only that connected nodes be close but also that the directionality be respected. General-purpose approaches like TransE [3], DistMult , GraphSAGE [5] optimize margin-based or cross-entropy losses to reconstruct observed links, yet they treat every edge symmetrically. To capture the one-way nature of prerequisites, researchers have proposed asymmetric scoring functions and margin-based penalties on reversed edges. We build on this insight by using a coordinate-wise hinge loss that explicitly drives true prerequisites to dominate in the embedding space.

7.3 Order-Preserving Embeddings

Order embeddings [11] focus squarely on partial orders: each node lives in the non-negative orthant of Euclidean space, and a valid edge requires a coordinate-wise greater-than-or-equal-to relation. This design elegantly captures hierarchies where deeper or more advanced concepts naturally sit below their prerequisites.

Order embeddings are specifically designed to represent partial order structures within vector spaces. In the Euclidean implementation, each concept is embedded in a positive orthant (the region where all coordinates are non-negative) of $R^d_{\geq 0}$, and order is enforced by ensuring that for each prerequisite pair (u, v), all dimensions of u are greater than or equal to those of v. The violation is measured using a coordinate-wise hinge function:

$$d_{\text{order}}(u, v) = \sum_{k=1}^{d} \max\big(0,\, v_k - u_k\big).$$

This approach has been used successfully for encoding hierarchical relationships in lexical ontologies, knowledge bases, and course graphs.

Recent research extends this idea into hyperbolic space. Nickel and Kiela [10] propose Poincaré embeddings, which place nodes inside the unit ball so that distances grow exponentially toward the boundary, better matching tree-like hierarchies. In the Poincaré ball, order can be modeled via geodesic cones and loss functions adapted to the hyperbolic metric, preserving differentiability and stability. While hyperbolic extensions [10] promise even tighter fits for tree-like data, they introduce numerical and optimization complexities. For our purposes where we are balancing implementation robustness with representational fidelity, the Euclidean order embedding remains a good choice.

7.4 Algorithmic Pathfinding: Dynamic Programming vs. A*

Once a directed, weighted DAG is in place, the task becomes: how do we traverse it? Dynamic Programming (DP) leverages the acyclicity to compute a globally optimal cost-to-reach every node in a single topological sweep [2]. This guarantees that the assembled sequence is optimal under the chosen scoring criterion and naturally aligns with progressive learning. By contrast, A* search [6] incrementally expands paths guided by a heuristic (often semantic similarity or estimated remaining cost) which can be faster for single-goal queries but may miss globally coherent routes if the heuristic is not admissible.

8 Methodology and Capstone Design

8.1 Graph Structure

Our system is built upon a directed knowledge graph stored in Neo4j. Each node in the graph represents a concept, job title, or skill, and each directed edge represents a dependency between them, most commonly a prerequisite relationship.

The main node types and their labels include:

- **Concept** (`:Concept`): Academic or theoretical concepts (e.g., "Linear Algebra", "Neural Networks") with associated properties such as `name` and `definition`.
- **HardSkill** and **Technology** (`:HardSkill`, `:Technology`): Applied or technical skills often required by job descriptions.
- **Job** (`:Job`): Representative job roles grouped into clusters.
- **SoftSkill** (`:SoftSkill`): Non-technical competencies (e.g., "communication").

Each node typically includes two key properties: `name` (or `title` for jobs) and `definition`, both stored as strings. Figure 8.1 shows a screenshot of this schema as extracted from Neo4j.

	labels	props
1	["Concept"]	[["name:", "String"], ["definition:", "String"]]
2	["Concept", "HardSkill"]	[["name:", "String"], ["definition:", "String"]]
3	["Concept", "HardSkill", "Technology"]	[["name:", "String"], ["definition:", "String"]]
4	["Concept", "Technology"]	[["name:", "String"], ["definition:", "String"]]
5	["Job"]	[["title:", "String"], ["definition:", "String"]]
6	["SoftSkill"]	[["name:", "String"], ["definition:", "String"]]

Figure 8.1: Node Labels and Properties in the Graph

The main relationship type is:

- `(:Concept)-[:REQUIRES]->(:Concept)`: Indicates that one concept requires another as a prerequisite.

The REQUIRES relationship forms the conceptual backbone of the learning path graph. An example subgraph is shown in Figure 8.2.

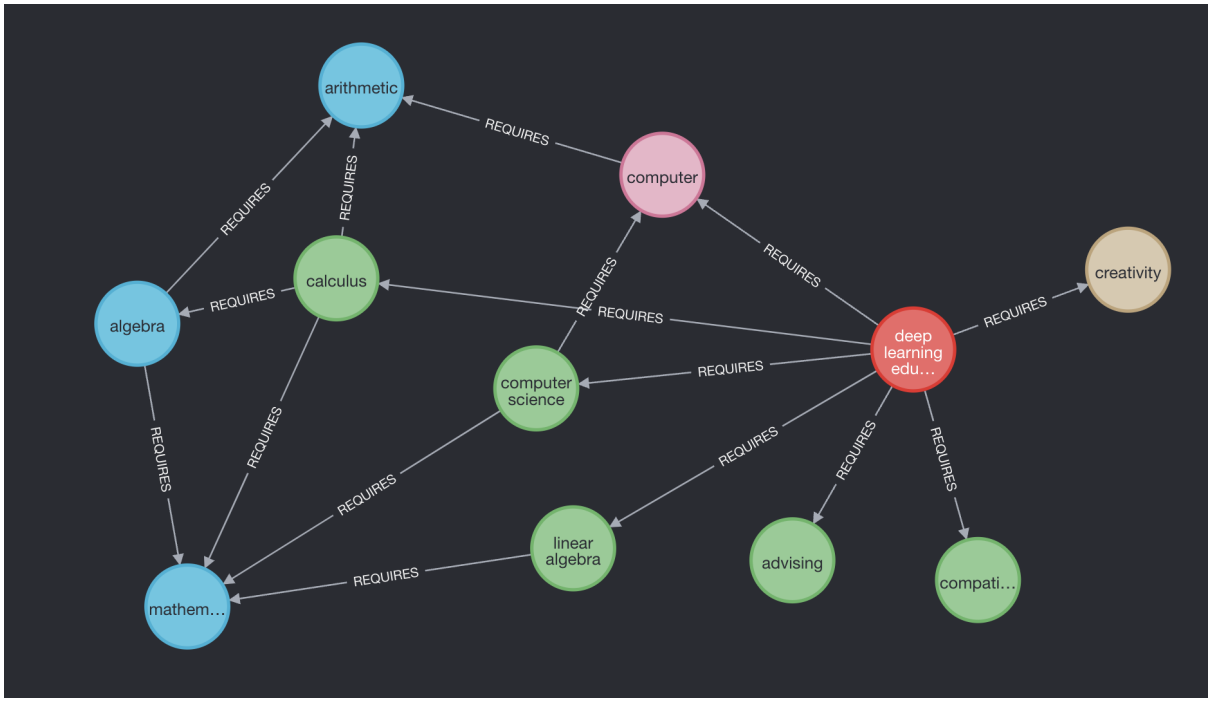


Figure 8.2: Example of Concept Dependencies (REQUIRES relationships)

8.2 Cycle Detection

8.2.1 Cycle Detection Motivation

The goal of the graph is to allow traversal from a learner's current skills to their target job through valid, ordered learning steps. However, the presence of **cycles** in the REQUIRES relationships violates this assumption and makes path generation infeasible. For example, if Concept A requires B and B requires A, the graph contains a logical contradiction.

These cycles must be removed to produce a valid Directed Acyclic Graph (DAG) suitable for learning path generation.

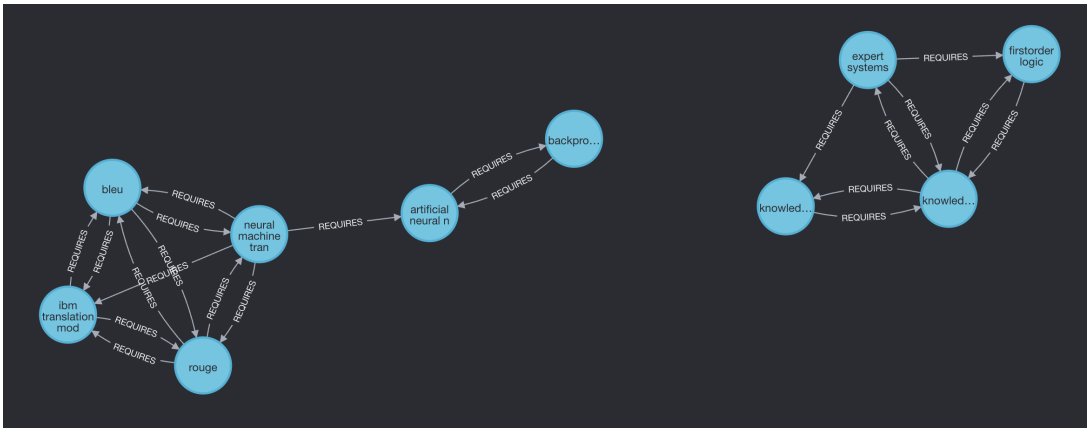


Figure 8.3: Detected Cycles in the Graph via Cypher Query

8.2.2 Cycle Detection via Strongly Connected Components (SCC)

To identify cycles efficiently, we compute Strongly Connected Components (SCCs). In a directed graph, each SCC is a maximal set of nodes such that every node is reachable from every other node in the set.

We exported the edge list from Neo4j and used NetworkX in Python to compute SCCs. Any SCC with more than one node corresponds to a cycle in the graph.

8.2.3 Definition of Strongly Connected Components (SCCs)

In a directed graph $G = (V, E)$, a **strongly connected component (SCC)** is a maximal subgraph $G' = (V', E')$ such that for every pair of vertices $u, v \in V'$, there exists a directed path from u to v and a directed path from v to u. In other words, all nodes in an SCC are mutually reachable. SCCs are particularly useful for detecting cycles: if an SCC contains more than one node, then those nodes form a cyclic structure. Any REQUIRES relationship inside such a component violates the prerequisite hierarchy needed for learning path generation.

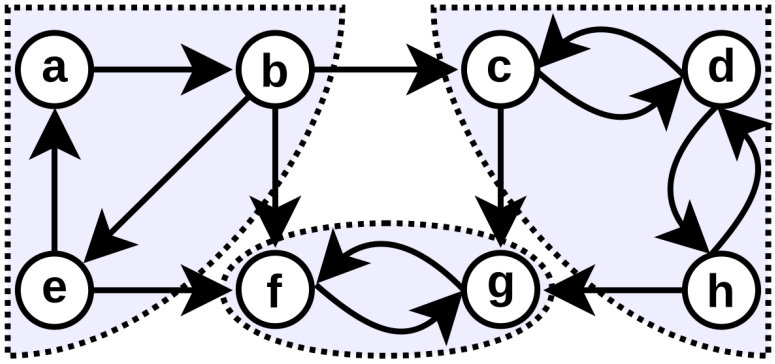


Figure 8.4: Example of Strongly Connected Components Detected in a Graph

8.3 Cycle Removal Strategy

To break all cycles while removing as few edges as possible, one would need to solve the minimum feedback-arc set problem, which is about finding the smallest set of edges whose removal makes the graph acyclic. Unfortunately, this is NP-hard, it naïvely checks every subset of edges, which takes $O(2^{|E|})$ time, so we instead employ a simple, degree-based greedy heuristic that removes one edge per cycle. For each edge $(u \to v)$ in a detected cycle, compute the *combined degree*: $deg(u) + \deg(v)$. We then remove the edge with the *smallest* combined degree, under the intuition that less-connected links are less central to the overall graph structure.

Although this heuristic does not guarantee a minimum-size feedback set, it runs in polynomial time and typically breaks all cycles with only a small fraction of edges removed.

```
G_full: 1050 SCCs, 6 multi-node (max size 80)
INFO: Removed 32 edges; edges 3667 → 3635; graph is now acyclic
```

This log shows our heuristic breaks all cycles by pruning under 1 % of edges.

8.4 Order Embeddings for Prerequisite Hierarchies

8.4.1 Definition of Order Embeddings

To capture the directional nature of prerequisite relationships, we embed each concept node in a non-negative vector where valid edges correspond to a coordinate-wise dominance relation. Concretely, let

$$f : R^d_{\geq 0} \longrightarrow R^h_{\geq 0}$$

be a learnable projection (a linear map plus ReLU) that takes an initial non-negative representation of a node to its h-dimensional order embedding. For any candidate edge $(u \to v)$, we then define the order violation to be:

$$d_{\text{order}}\big(f(x_u),\, f(x_v)\big) \;=\; \sum_{k=1}^{h} \max\big(0,\; f_k(x_v) \;-\; f_k(x_u)\big) \;\geq 0.$$

Here, k refers to the k-th coordinate of the embedding.

When $d_{\text{order}} = 0$, $f(x_u)$ dominates $f(x_v)$ in every coordinate, indicating perfect adherence to the prerequisite direction. Larger positive values means higher violation of the prerequisite relationship.

8.4.2 Role of Order Embeddings

Once the graph is acyclic (Section 8.3), we leverage order embeddings for two key tasks:

1. **Link prediction for loose skills:** For nodes that have no incoming prerequisites but are

required by jobs, we compute the *order-violation* metric

$$d_{\text{order}}(u, v) = \sum_{i=1}^{h} \max\big(0,\ v_i - u_i\big)$$

on their embeddings $\mathbf{u} = f(x_u)$, $\mathbf{v} = f(x_v)$, to rank and suggest missing prerequisite edges.

2. **Score-weighted pathfinding:** We treat $d_{\text{order}}(u, v)$ as an edge weight in our traversal algorithms, finding learning paths that minimize the total violation cost and thus ensure pedagogical coherence.

8.5 Link Prediction for Loose Skills

8.5.1 Node Embedding Initialization

To embed each concept node in a vector space that reflects its semantic meaning, we first concatenate its *name* and *definition* into a single text string. These texts are fed through a pre-trained Sentence-BERT model ("all-MiniLM-L6-v2"), a variant of BERT fine-tuned to produce semantically meaningful sentence-level embeddings, to obtain high-dimensional sentence embeddings. Because downstream order-embedding training is both more efficient and numerically stable in a lower-dimensional space, we do the following:

1. Apply Principal Component Analysis (PCA) to reduce from D down to the target dimension d (e.g. $d = 128$):

$$x_v^{\text{PCA}} \ = \ \text{PCA}_{D \to d}\big(x_v^{\text{SBERT}}\big).$$

PCA identifies the directions along which the embeddings vary most, and projects each vector

onto the top d components. This preserves variance in a compact form.

2. Clamp negative values to zero via a ReLU:

$$x_v \;=\; \max\bigl(0,\; x_v^{\mathrm{PCA}}\bigr) \;\in\; R^d_{\geq 0}.$$

8.5.2 Training Pair Generation

Order embeddings are learned contrastively on positive and negative node pairs.

For **positive pairs:** Every existing directed edge $(u \to v)$ in the pruned DAG is considered a positive example, since it reflects a true prerequisite relationship.

For **negative pairs:** To avoid trivial negatives, for example totally unrelated nodes, we first estimate a layer index using the following algorithm:

Algorithm 1 Estimate Node Layers via Topological Sort

Require: DAG $G = (V, E)$
Ensure: Map $\ell : V \to N$ giving each node's layer
Phase 1: Topological sort (Kahn)
compute in-degree $d[v]$ for all $v \in V$
$Q \leftarrow \{\, v \mid d[v] = 0\}$, topo $\leftarrow [\,]$
while $Q \neq \emptyset$ **do**
  $u \leftarrow \mathsf{pop}(Q)$; append u to topo
  for all $(u \to w) \in E$ **do**
    $d[w] \leftarrow d[w] - 1$
    if $d[w] = 0$ **then** push w into Q
    end if
  end for
end while
Phase 2: Longest-predecessor layer assignment
for all v in topo **do**
  $\ell[v] \leftarrow 1 + \max\{\ell[p] \mid (p \to v) \in E\}$ {0 if no predecessors}
end for
return ℓ =0

Rather than selecting entirely random non-edges as negatives, we sample *plausible* negatives that are more likely to be prerequisites, thereby forcing the model to learn subtle distinctions.

For each true edge $(u \to v)$ we identify all nodes w such that $\ell(w) < \ell(u)$ in the topological layering and $(u, w) \notin E$. These w lie before u in the learning hierarchy, so the edge $u \to w$ would be directionally plausible yet is incorrect. Then, we randomly pick k such nodes (e.g. $k = 5$) as negative examples for u.

8.5.3 Train/Validation split

We combine all (u, v, y) triples with labels $y \in \{0, 1\}$, y=1 of there is an edge between u and v ,y=0 otherwise. We perform a split of 80/20 and we monitor generalization via held-out validation AUC.

8.5.4 Contrastive Margin-Based Loss

We optimize the following loss over positive and negative node pairs, with margin $\gamma = 1$:

$$\ell(u, v) \;=\; y\, d_{\text{order}}(f(x_u), f(x_v)) \;+\; (1 - y)\, \max\big(0,\; \gamma \;-\; d_{\text{order}}(f(x_u), f(x_v))\big),$$

where $y = 1$ for true prerequisite edges and $y = 0$ for sampled negatives.

- If $(y = 1)$: $\ell = d_{\text{order}}$. We drive the violation toward zero, enforcing $f(x_u)$ to dominate $f(x_v)$.
- If $(y = 0)$: $\ell = \max(0, \gamma - d_{\text{order}})$. Any negative whose violation is below the margin γ incurs a penalty, pushing it to satisfy $d_{\text{order}} \geq \gamma$.

The margin γ defines a minimum gap between positives and negatives. True edges are encouraged to have zero violation, while false edges must exceed γ in order-violation to be considered safely non-prerequisite.

8.5.5 Optimisation Routine

Our goal is to minimise the mini-batch version of the contrastive objective $\mathcal{L} = \frac{1}{B} \sum_{(u,v,y) \in \mathcal{B}} \ell(u, v, y)$ with respect to the projection parameters $\theta = (W, b)$. For every batch $\mathcal{B}$ of 64 pairs we:

1) **Forward pass.** Compute the projected embeddings $z_u = f(x_u)$ and $z_v = f(x_v)$, evaluate the order-violation scores $d_{\text{order}}(z_u, z_v)$, and accumulate the batch loss $\mathcal{L}$.

2) **Back-propagation.** Differentiate $\mathcal{L}$ with respect to θ; the ReLU keeps the gradient flow sparse but stable.

3) **Parameter update.** Apply Adam with learning-rate $\alpha = 10^{-3}$ (default β_1=0.9, β_2=0.999). Adam maintains first- and second-moment estimates of the gradient, rescales them ($\hat{m}_t, \hat{v}_t$), and performs the adaptive step $\theta \leftarrow \theta - \alpha\, \hat{m}_t/(\sqrt{\hat{v}_t} + \varepsilon)$.

After each epoch we score the validation set with $-d_{\text{order}}$ and compute the ROC–AUC. The model with the highest AUC is retained; training stops when the metric fails to improve for 10 consecutive epochs. This early-stopping scheme prevents over-fitting and yields a well-calibrated projection that we later reuse for link prediction and path scoring.

8.5.6 Identification of Loose Skills

A loose skill is a hard-skill or technology node w that has no incoming prerequisites in the concept graph, $\deg^-(w) = 0$, and is required by at least one job node. These nodes form the set

$$L = \big\{\, w \mid \deg^-(w) = 0,\ \exists\, j \in \text{Job} :\ j \to w \big\},$$

and are the primary targets for link prediction.

8.5.7 Link Prediction for Loose Skills

For each $\ell \in L$, let ℓ_{layer} be its topological layer (see Algorithm 1). We consider as candidates all concept nodes u with $\text{layer}(u) < \text{layer}(\ell)$. Each candidate is scored by

$$\hat{s}(u, \ell) \ = \ - \, d_{\text{order}}\big(f(x_u),\ f(x_\ell)\big),$$

so that larger $\hat{s}$ values indicate more plausible prerequisite relationships. Since $d_{\text{order}} \in [0, \infty)$, it follows that

$$\hat{s}(u, \ell) \ \in \ (-\infty,\ 0],$$

with $\hat{s} = 0$ corresponding to a perfect coordinate-wise ordering. To ensure acyclicity, we temporarily add $(u \to \ell)$, test for new cycles, and discard any u that creates one. Finally, we sort the surviving candidates by $\hat{s}$, and propose the top k whose scores exceed a threshold $\tau = -0.3$. Once the top-k candidates have been selected, each suggested edge $(u \to \ell)$ is immediately inserted into the graph with two additional properties:

- `predicted = true`, marking it as algorithmically inferred rather than originating from the source data,
- `score` $= \hat{s}(u, \ell)$, storing the computed link plausibility (in $(-\infty, 0]$) for later use.

These flags and scores are retained on every edge and form the basis for subsequent unified scoring and score-weighted pathfinding.

8.6 Learning Path Generation

8.6.1 Problem Formulation

After constructing an acyclic knowledge graph with meaningful prerequisite relationships, the core challenge is to find optimal learning paths from a learner's current knowledge to their target skills. Formally, given:

- $G = (V, E)$: a directed acyclic graph where nodes represent concepts and skills
- $U \subset V$: the set of nodes representing skills the user already knows
- $T \subset V$: the set of target skills required for a job

we seek to find, for each target $t \in T$, an optimal path $P_t = (v_1, v_2, \ldots, v_n)$ where:

- $v_1 \in U$ (path starts from user knowledge)
- $v_n = t$ (path ends at the target skill)
- $(v_i, v_{i+1}) \in E$ for all $i \in \{1, 2, \ldots, n-1\}$ (path follows valid edges)
- P_t optimizes a scoring function that considers multiple factors

The quality of a learning path depends on several factors, including the strength of prerequisite relationships, semantic coherence of transitions, and the overall path length. We define a unified edge scoring function that combines these factors:

$$s(u, v) = \Delta - \alpha \cdot s_{\text{norm}}(u, v) - \beta \cdot \text{sem\_pen}(u, v) \tag{8.1}$$

Where:

- $\Delta = 0.3$ is a base gain constant that provides a positive incentive to extend paths, counterbalancing the negative terms. Higher values favor longer paths that include more concepts.
- $s_{\text{norm}}(u, v) = \frac{\text{raw\_score}(u,v)}{s_{\text{max}}}$ is the normalized edge score from our prerequisite graph.
- $\text{sem\_pen}(u, v) = 1 - \cos(f(x_u), f(x_v))$ is the semantic dissimilarity between concepts, measuring how different they are in meaning space.
- $\alpha = 0.9$ controls the importance of prerequisite strength from the graph structure. Higher α prioritizes well-established prerequisites and penalizes weaker connections.
- $\beta = 0.1$ governs how much semantic similarity influences path selection. Higher β values force paths to follow more coherent conceptual progressions.
- $\lambda = 0.7$ penalizes algorithmically inferred edges, making the system more conservative about using predicted relationships versus explicit ones.
- $\lambda_{\text{len}} = 0.08$ prevents excessive path length by imposing a cumulative cost as paths grow, helping balance depth versus directness.

We determined these specific values through iterative testing on representative learning scenarios. The relative magnitudes ($\alpha > \beta$) reflect our decision to prioritize explicit prerequisite structure over semantic similarity. This makes intuitive sense: while ”linear algebra” and ”calculus” may be semantically similar mathematical concepts, one is not necessarily the best prerequisite for the other.

For predicted edges (from Section 8.4.2), we add a prediction penalty:

$$\text{raw\_score}(u, v) = |s(u, v)| + \lambda \cdot 1_{\text{predicted}}(u, v) \tag{8.2}$$

where $\lambda = 0.7$ and $1_{\text{predicted}}$ is an indicator function that equals 1 for predicted edges.

We implement and compare two different approaches for finding optimal learning paths: Dynamic Programming and A* Search.

8.6.2 Dynamic Programming Approach

Dynamic Programming (DP) leverages the acyclic structure of the knowledge graph to find globally optimal paths. The key insight is that optimal paths to a node can be constructed from optimal paths to its predecessors.

Algorithm Overview

We define a value function $dp(v)$ that represents the score of the best path ending at node v:

$$dp(v) \;=\; \begin{cases} 0, & v \in U, \\ \max\limits_{(u,v)\in E} \big\{\, dp(u) \;+\; s(u,v) \;-\; \lambda_{\text{len}} \;\cdot\; \text{pathlen}(u) \big\}, & \text{otherwise.} \end{cases} \tag{8.3}$$

Where pathlen(u) is the length of the optimal path to u, and λ_{len} is a length penalty factor (set to 0.08) that discourages excessively long paths.

Path Construction

After computing optimal paths, we simply follow the $prev$ pointers backwards from the target node, then reverse the resulting sequence. This simple backtracking process reconstructs the full path from a user skill to the target skill. If no path exists (i.e., $prev[t] = \text{null}$), we fall back to our orphan skill handling strategy (Section 8.6.4).

The DP approach has several advantages:

- Guarantees optimal paths according to our scoring function
- Naturally handles the topological structure of knowledge acquisition
- Processes the entire graph in a single pass
- Explicitly balances multiple path quality factors

Algorithm 2 Dynamic Programming for Learning Paths

Require: DAG $G = (V, E)$, user skills $U \subset V$, target skill $t \in V$
Ensure: Optimal learning path from U to t

```
dp[v] ← −∞ for all v ∈ V {Initialize scores}
prev[v] ← null for all v ∈ V {Store predecessors}
pathlen[v] ← 0 for all v ∈ V {Path lengths}
for all u ∈ U do
    dp[u] ← 0 {Initialize user skills with zero cost}
end for
for all v in topological sort of G do
    if dp[v] = −∞ then continue
    end if
    for all (v, w) ∈ E do
        length_penalty ← λ_len · pathlen[v]
        gain ← s(v, w) − length_penalty
        new_score ← dp[v] + gain
        if new_score > dp[w] and pathlen[v] + 1 ≤ MAX_PATH_LENGTH then
            dp[w] ← new_score
            prev[w] ← v
            pathlen[w] ← pathlen[v] + 1
        end if
    end for
end for
return constructPath(prev, t) =0
```

8.6.3 A* Search Approach

While Dynamic Programming provides a global optimization across the entire graph, the A* search algorithm offers a more focused, targeted approach for finding paths between specific pairs of nodes.

A* Search Formulation

As an alternative approach, we implement A* search, which uses a heuristic function to guide exploration toward promising paths:

$$f(v) = g(v) + h(v) \tag{8.4}$$

Where $g(v)$ is the cost of the path from a starting node to v, and $h(v)$ is a heuristic estimate of the cost from v to the target. We define $h(v) = 1 - \cos(f(x_v), f(x_t))$, using semantic similarity to estimate conceptual distance to the target node.

The core A* algorithm maintains a priority queue of paths ordered by their f-scores, always expanding the most promising path. When a path reaches the target node, it is immediately returned as the solution. For users with multiple skills, we run A* from each skill and select the best resulting path.

Algorithm 3 A* Search for Learning Paths

Require: DAG $G = (V, E)$, user skill $u \in U$, target skill $t \in V$
Ensure: Shortest path from u to t guided by semantic similarity
0: $open \leftarrow$ priority queue with $(f(u) = h(u), g(u) = 0, u, [u])$
0: $closed \leftarrow \emptyset$
0: **while** $open$ is not empty **do**
0: $(f_{curr}, g_{curr}, curr, path) \leftarrow open$.pop() {Get node with minimum f-score}
0: **if** $curr = t$ **then return** $path$
0: **end if**
0: **if** $curr \in closed$ **or** $|path| > MAX\_PATH\_LENGTH$ **then continue**
0: **end if**
0: Add $curr$ to $closed$
0: **for all** $(curr, v) \in E$ **do**
0: **if** $v \in closed$ **then continue**
0: **end if**
0: $g_v \leftarrow g_{curr} + 1$
0: $f_v \leftarrow g_v + (1 - \cos(emb[v], emb[t]))$
0: Add $(f_v, g_v, v, path + [v])$ to $open$
0: **end for**
0: **end while**
0: **return** null {No path found} =0

The A* approach has several potential advantages:

- More efficient for finding paths between specific pairs of nodes
- Semantic-guided exploration may find more intuitive paths
- Can terminate early when a valid path is found
- May produce shorter paths in terms of steps

8.6.4 Orphan Skill Handling and Comparison Methodology

For target skills where no path can be found (orphan skills), we implement a fallback strategy with two phases:

1. Semantic Connection: We search for user skills that are semantically similar to the orphan skill, suggesting potential conceptual bridges when direct prerequisites don't exist.

2. Dependent Concept Recommendation: When semantic connections fail, we identify concepts that depend on the orphan skill and suggest these as next steps after learning the orphan skill directly.

We evaluate both pathfinding approaches on multiple dimensions:

- **Completeness**: Percentage of target skills for which a valid path is found
- **Path Length**: Average number of steps in the generated paths
- **Semantic Coherence**: Average semantic similarity between consecutive concepts
- **Educational Validity**: Subjective assessment of path quality by domain experts
- **Computational Efficiency**: Time required to generate complete learning plans

By implementing both DP and A* approaches and comparing their effectiveness on real learning scenarios, we aim to determine which method produces more useful and educationally valid learning paths.

8.7 Web App Development

8.7.1 System Architecture

The Career Path Navigator implements a distributed multi-tier architecture designed to efficiently process user interactions and generate personalized learning paths through interconnected components.

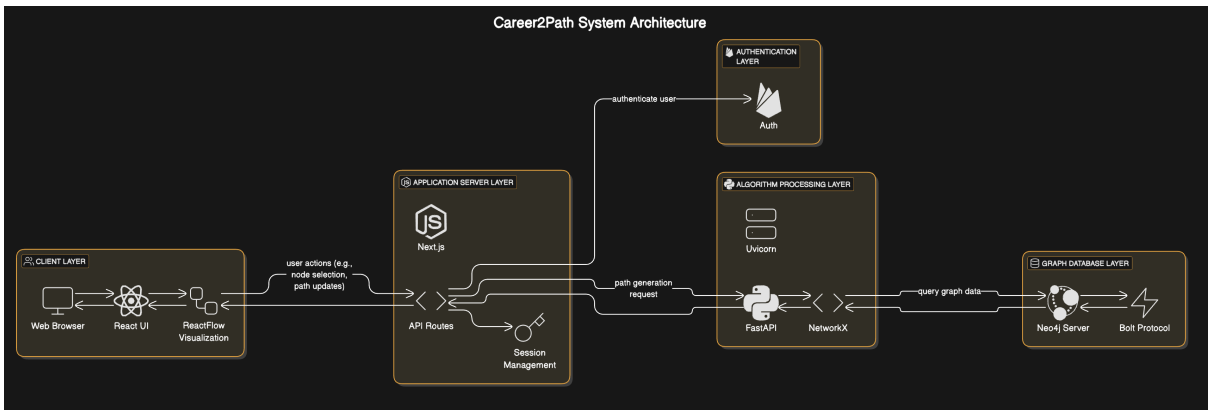


Figure 8.5: System Architecture for Concept2Career

The architecture demonstrates critical system interactions: the Client Layer handles user input through React UI and ReactFlow visualization, while the Application Server Layer manages API routes and session control. Authentication flows directly to Firebase, ensuring secure user access. When path generation is requested, the API Routes communicate with FastAPI through Uvicorn on the Algorithm Processing Layer, which utilizes NetworkX for graph algorithms. This layer queries the Neo4j Graph Database via Bolt Protocol for skill and relationship data, creating a circular flow that returns optimized learning paths to the user interface.

8.7.2 Technology Stack Overview

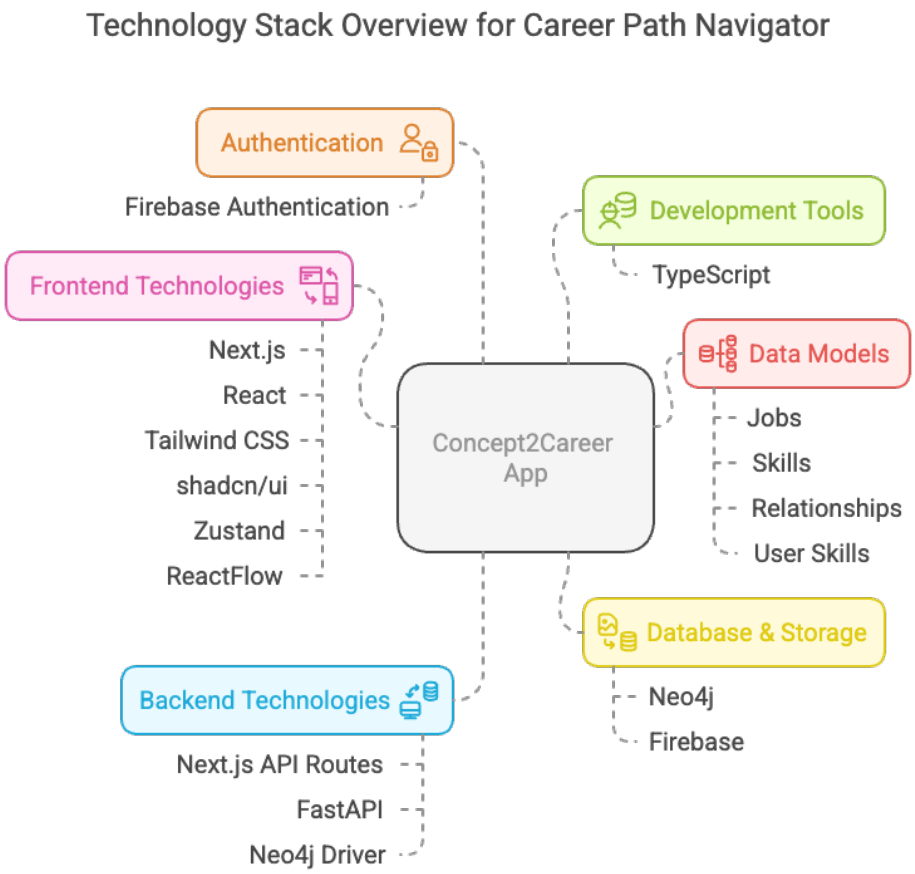


Figure 8.6: Technology Stack Overview for Career Path Navigator

The technology stack is organized into five functional categories that support the application's core features. Frontend technologies provide responsive UI with Next.js for server-side rendering and React for interactive components, while Tailwind CSS and shadcn/ui ensure

```python
# Set up logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.StreamHandler(sys.stdout)
    ]
)
logger = logging.getLogger(__name__)

app = FastAPI()

# Add CORS middleware
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # Adjust in production
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Figure 8.7: FastAPI Initialization

consistent, accessible design. TypeScript enhances development with static typing across the stack. The backend layer combines Next.js API Routes for request handling with FastAPI for computationally intensive algorithm processing. Neo4j serves as the graph database for skill-relationship modeling, complemented by Firebase for secure authentication. This architecture balances performance requirements with development efficiency, allowing each technology to operate within its area of expertise.

8.7.3 Backend Implementation

The backend leverages FastAPI, a modern Python web framework, running on Uvicorn ASGI server to deliver high-performance API endpoints for learning path generation.

FastAPI Service Architecture

FastAPI was selected for its asynchronous capabilities, automatic request validation, and high performance characteristics. The service implements a layered architecture as follows:

Database Integration

The system implements a custom Neo4j driver wrapper for efficient connection management:

```python
class Neo4jDriver:
    def __init__(self):
        self._driver = None

    def get_driver(self):
        if self._driver is None:
            try:
                logger.info(f"Initializing Neo4j driver with URI: {NEO4J_URI}")
                self._driver = GraphDatabase.driver(
                    NEO4J_URI, auth=(NEO4J_USER, NEO4J_PASSWORD)
                )
                # Test the connection
                with self._driver.session() as session:
                    result = session.run("RETURN 1 as test")
                    record = result.single()
                    assert record["test"] == 1
                logger.info(f"Successfully connected to Neo4j at {NEO4J_URI}")
            except Exception as e:
                logger.error(f"Failed to connect to Neo4j: {e}")
                logger.error(traceback.format_exc())
                raise
        return self._driver

    def close(self):
        if self._driver is not None:
            self._driver.close()
            self._driver = None

neo4j_driver = Neo4jDriver()
```

Figure 8.8: Neo4j Driver Initialization

Request Handling

The service employs Pydantic models for request validation, ensuring type safety and automatic documentation:

```python
class PathRequest(BaseModel):
    jobId: str
    userSkills: Optional[List[str]] = []

@app.post("/generate-path")
async def generate_path(request: PathRequest):
    job_id = request.jobId
    user_skills = request.userSkills or []

    # Process request and generate learning paths
    skills = get_job_skills(job_id)
    nodes, label_of, edges = get_all_graph_data()
    prev = build_dp_paths(nodes, label_of, edges)
```

Figure 8.9: Code for Request Handling

Graph Query Integration

The backend extensively uses Cypher queries to retrieve skill relationships and prerequisite chains from Neo4j:

```python
def get_skill_prerequisites(skill_id):
    """Retrieve prerequisite tree for a specific skill using Cypher"""
    query = """
    MATCH path = (s1:Skill {id: $skillId})-[:REQUIRES*1..5]->(s2:Skill)
    RETURN path,
           relationships(path) as edges,
           [node in nodes(path) | node.id] as node_ids,
           length(path) as depth
    ORDER BY depth DESC
    """

    with neo4j_driver.get_driver().session() as session:
        result = session.run(query, skillId=skill_id)

        prerequisite_paths = []
        for record in result:
            path_data = {
                "node_ids": record["node_ids"],
                "depth": record["depth"],
                "prerequisite_edges": [
                    {
                        "source": rel.start_node.get("id"),
                        "target": rel.end_node.get("id"),
                        "score": rel.get("score", 0.7),
                        "predicted": rel.get("predicted", False)
                    } for rel in record["edges"]
                ]
            }
            prerequisite_paths.append(path_data)

    return prerequisite_paths
```

Figure 8.10: Cypher Queries Integration

This Cypher query efficiently traverses the skill graph up to 5 levels deep, retrieving all prerequisite relationships and their properties for path analysis.

Error Handling and Response Management

The system implements comprehensive error handling to ensure robust operation:

```python
@app.exception_handler(RequestValidationError)
async def validation_exception_handler(request, exc):
    logger.error(f"Validation error: {exc}")
    return JSONResponse(
        status_code=422,
        content={"status": "error",
                 "message": "Invalid request data",
                 "detail": str(exc)}
    )


@app.exception_handler(Exception)
async def general_exception_handler(request, exc):
    logger.error(f"Unhandled exception: {exc}")
    return JSONResponse(
        status_code=500,
        content={"status": "error",
                 "message": "Internal server error",
                 "detail": str(exc)}
    )
```

Figure 8.11: Error and Response Handling

The FastAPI backend efficiently handles complex graph operations, providing a scalable foundation for the learning path generation engine while maintaining high performance through asynchronous processing and optimized database interactions.

9 Data Presentation

This section provides an overview of the datasets used in the project, the steps taken to collect and prepare the data, and the challenges encountered during this process. Our system integrates data from both academic sources (for concept prerequisite modeling) and industry sources (for job requirements and skill extraction).

9.1 Academic Concept Data: Prerequisite Graphs

To model conceptual dependencies between technical topics, we used two well-established datasets that contain validated prerequisite relations between computer science concepts.

CPR Dataset (Concept Prerequisite Relations)

- Source: `https://github.com/harrylclc/eaai17-cpr-recover`
- Contents: Prerequisite pairs, course descriptions, and manual annotations
- Structure: A CSV file where each line (A, B) means "B is a prerequisite for A"

We used the `cs_preqs.csv` file containing concept pairs [8] file containing concept pairs generated through expert majority voting. These pairs form the backbone of the initial `REQUIRES` relationships in the graph.

LectureBank Dataset

- Source: `https://github.com/Yale-LILY/LectureBank`

- Contents: Concept relations in NLP and Computer Vision
- Structure: XML and CSV-formatted files with directed prerequisite edges

This dataset expanded our graph beyond traditional CS concepts and allowed us to incorporate advanced AI topics (e.g., "Word Embeddings," "Neural Machine Translation")[4].

9.2 Job Description Data and Market Requirements

To identify skills required by real-world AI and data science jobs, we collected data from job postings. Several approaches were tested before settling on the final solution.

Initial Attempts and Challenges

- **LinkedIn and Indeed**: These platforms were explored first due to their relevance and job variety. However, both rely on dynamic content and have robust anti-bot mechanisms (e.g., CAPTCHA, JavaScript rendering) that made reliable scraping infeasible.
- **Public Datasets (Kaggle, OpenML)**: Many open datasets were examined. However, most were outdated (collected between 2015–2019), lacked AI-related roles, or emphasized salary/location rather than technical requirements.

Final Data Source: HiringCafe.com

We identified `https://hiringcafe.com` as a modern job board with recent postings focused on machine learning and AI roles. The site offered consistent HTML formatting, limited anti-scraping barriers, and detailed descriptions.

We used **Selenium** and **BeautifulSoup** in Python to scrape over **800 job postings**, each with a job title, detailed description, and required skills/technologies.

9.3 Skill Extraction and Clustering

SkillNER for Named Entity Recognition

To extract both hard and soft skills from raw job descriptions, we used the **SkillNER** library [1]:

- Detects both technical (hard) skills and behavioral (soft) skills
- Outputs structured lists of relevant skills and tools

This process allowed us to construct bipartite links:

$$\texttt{(:Job)} \rightarrow \texttt{(:Skill)} \rightarrow \texttt{(:Concept)}$$

Job Title Clustering

Since many job titles were similar or redundant (e.g., "ML Engineer" vs "Machine Learning Specialist"), we applied a clustering pipeline:

1. Tokenize titles based on role keywords (e.g., engineer, architect, researcher)
2. Encode them using keyword vectors (based on LLMs, CV, MLOps, etc.)
3. Cluster into **50 representative job groups**
4. Assign the most common job title per cluster as the **canonical title**

5. Aggregate skills per cluster to define required skill sets

This allowed us to reduce noise and ensure that each job node in the graph corresponded to a generalizable and well-defined role.

9.3.1 Mathematical Formulation of Job Title Clustering

The goal of this clustering step was to group semantically similar job titles into clusters using soft feature encoding based on relevant technical skills and job roles. The process is mathematically structured as follows:

Step 1: Feature Vector Construction

Let $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ be the set of cleaned job titles. Each title t_i is mapped to a feature vector $x_i \in R^d$:

$$x_i = [\text{SkillVec}(t_i) \,|\, \text{RoleVec}(t_i)]$$

Here:

- SkillVec(t_i) encodes matches to m domain-specific skill categories
- RoleVec(t_i) encodes matches to r job role categories

Each match is optionally weighted using hierarchy depth. Let w_{ij} be the weighted match score for title t_i and skill category j:

$$\text{SkillVec}(t_i)[j] = \frac{w_{ij}}{\sum_{k=1}^{m} w_{ik}}, \quad \text{where } w_{ij} = \text{match count} \times (\text{depth}(j) + 1)$$

Step 2: Feature Normalization

All vectors are normalized using z-score normalization:

$$\tilde{x}_i = \frac{x_i - \mu}{\sigma}$$

Where μ and σ are the mean and standard deviation vectors over all x_i.

Step 3: Distance Matrix via Cosine Distance

We compute a cosine-based distance matrix:

$$D_{ij} = 1 - \frac{\tilde{x}_i \cdot \tilde{x}_j}{\|\tilde{x}_i\| \cdot \|\tilde{x}_j\|}$$

Step 4: Hierarchical Clustering

Agglomerative clustering is performed using average linkage:

$$\text{dist}(C_a, C_b) = \frac{1}{|C_a||C_b|} \sum_{u \in C_a} \sum_{v \in C_b} D_{uv}$$

Clusters are merged until a full dendrogram is produced. A dendrogram is a tree-structured diagram that shows how individual items (here, job titles) are agglomeratively merged into clusters. Each *leaf* at the bottom corresponds to one original item. Vertical lines connect these leaves up to *merge points* (nodes), which represent the distance at which two clusters join. The *height* of a horizontal line equals the linkage distance θ between clusters; a low merge point shows that the two clusters were very similar, while a high merge point indicates they were

more dissimilar. By scanning from left to right, you can see which items first group together and then how those groups successively combine, all the way up to a single root.

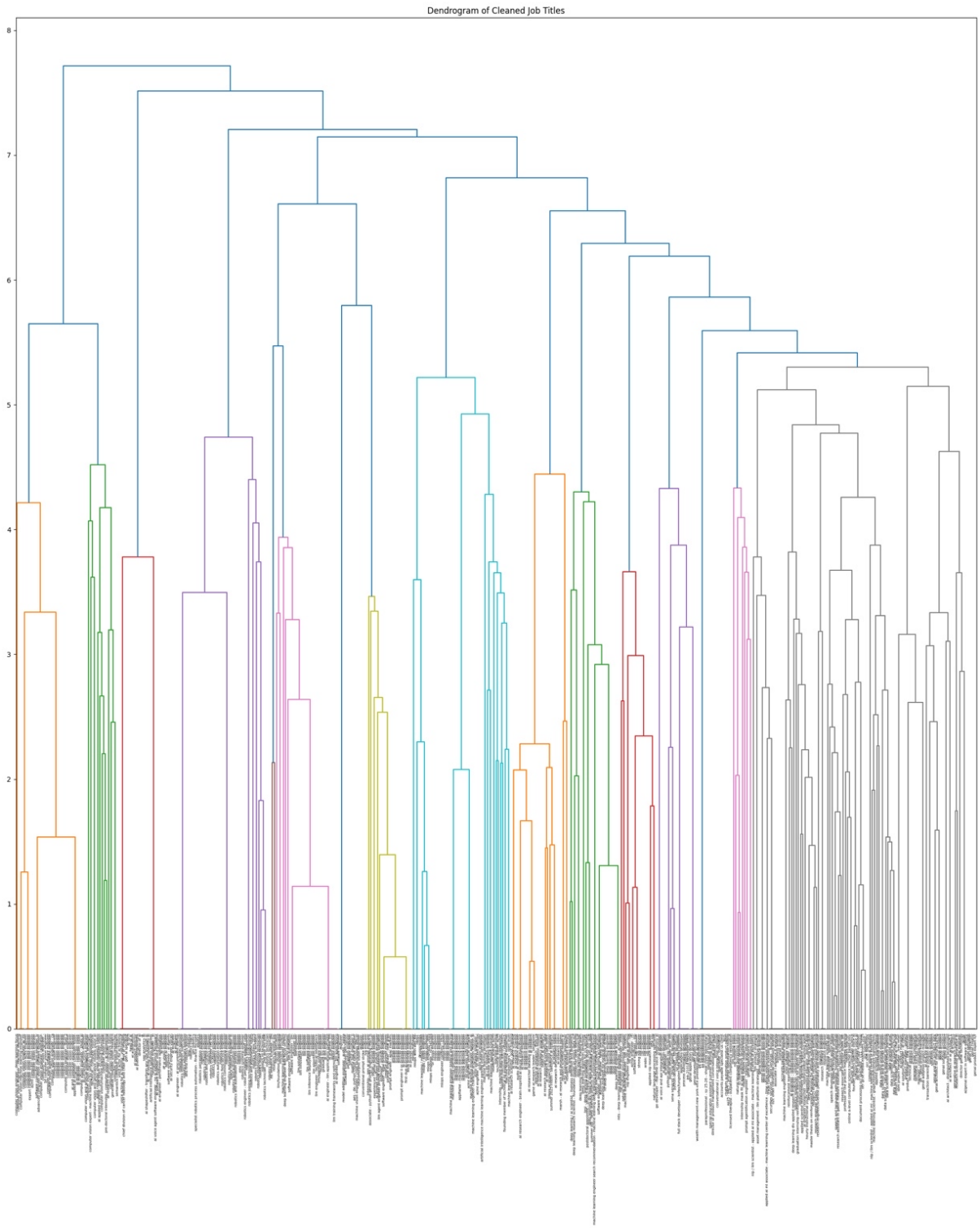


Figure 9.1: Dendrogram of Job Titles Based on Feature Similarity

Step 5: Threshold Selection via Elbow Method

We plot the number of clusters $k(\theta)$ obtained by cutting the dendrogram at various thresholds θ:

$$k(\theta) = \text{number of clusters when cut at distance } \theta$$

The elbow of this curve (where diminishing returns begin) is used to select the final clustering threshold (in our case, approximately $\theta = 3.15$).

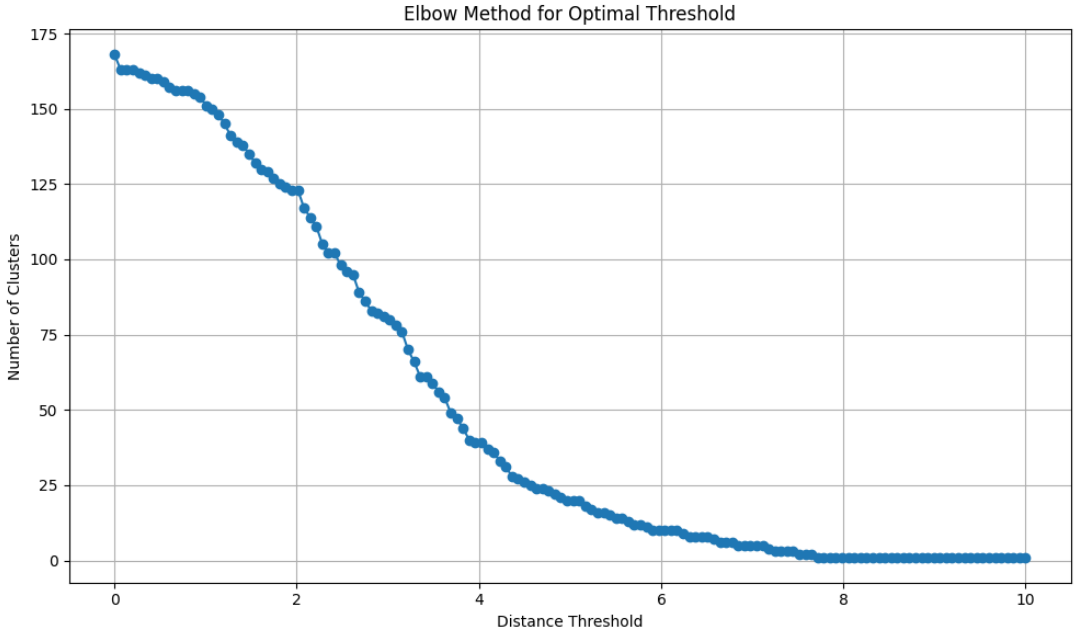


Figure 9.2: Elbow Method: Optimal Threshold for Job Title Clustering

Step 6: Cluster Label Assignment

Each cluster's average vector $\bar{x}_i$ is computed and the dominant skill and role dimensions are extracted:

$$\bar{x}_i = \frac{1}{|C_i|} \sum_{x_j \in C_i} x_j$$

This gives a representative title such as “NLP Scientist” or “MLOps Engineer.”

9.3.2 Skill Hierarchy Tree Used for Soft Encoding

To incorporate semantic depth into skill vectorization, we manually constructed a hierarchical tree of AI-related concepts. This structure was used to propagate weights downward during feature vector creation.

This hierarchy enables us to assign more weight to deeper categories during vector generation, ensuring that “prompt engineer” is recognized as a specialization within LLMs, which in turn falls under deep learning and machine learning. Semantic depth is crucial because incorporating hierarchical information into embeddings rather than treating all concepts as flat has been shown to boost link-prediction accuracy by over 8 % on standard benchmarks (e.g. the HAKE model) compared to non-hierarchical baselines [13].

9.3.3 Aggregation of Cluster-Based Job Requirements

To determine the most representative skills and technologies for each clustered job title, we performed a frequency-based aggregation across all job descriptions belonging to the same cluster.

Let each cluster be associated with a canonical `job_title`. For every such cluster, we:

- Counted the occurrences of each extracted hard skill, soft skill, and technology across all its job descriptions.

- Selected the top 15 hard skills, top 10 soft skills, and top 10 technologies.

9.3.4 Generating the Bipartite Graph in Neo4j

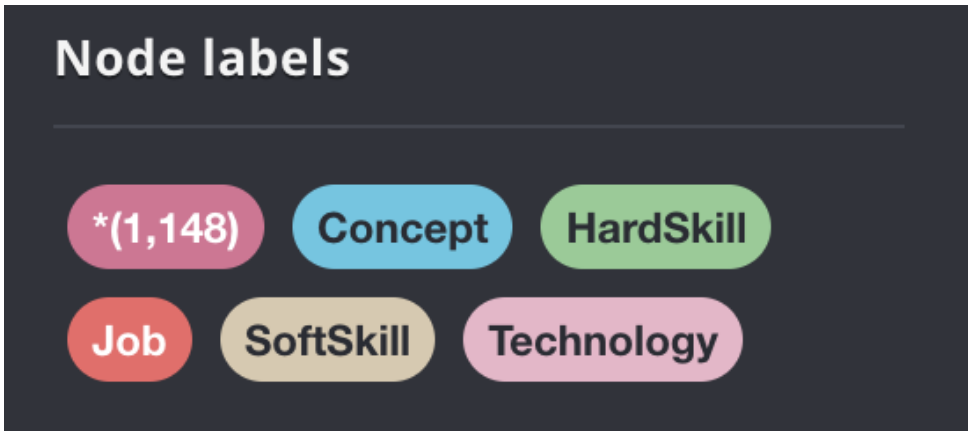
Each representative `:Job` node was linked in the graph to the top skills and technologies via a `REQUIRES` relationship.

The result is a bipartite subgraph with the structure:

$$(:Job) \xrightarrow{\texttt{REQUIRES}} (:HardSkill) \quad \text{and} \quad (:Job) \xrightarrow{\texttt{REQUIRES}} (:Technology)$$

9.3.5 Visualization in Neo4j

To visualize this bipartite structure, we use Cypher, the standard query language for Neo4j databases:



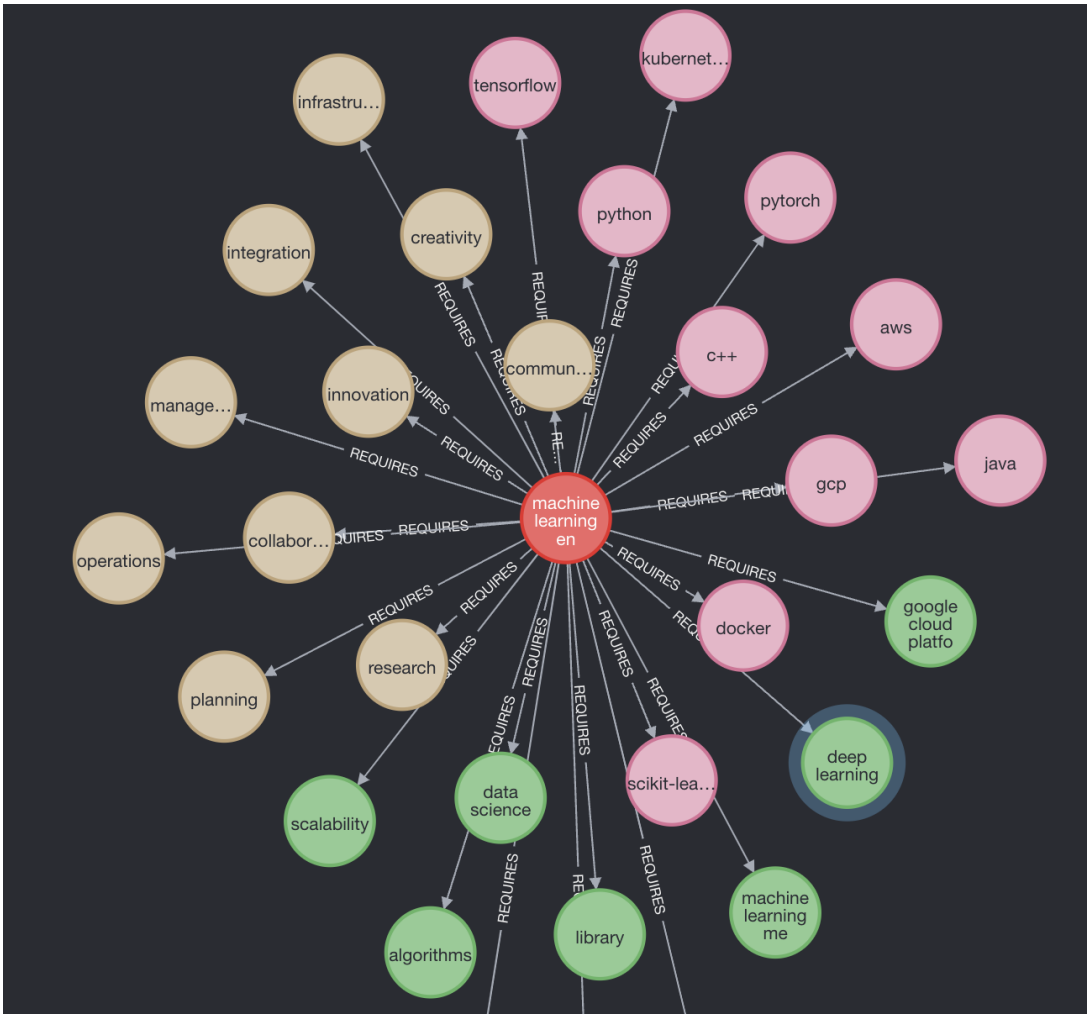


Figure 9.3: Bipartite Graph of Representative Job Titles and Their Required Skills and Technologies

10 Results and Discussion

10.1 Results and Analysis of Link Prediction

10.1.1 Evaluation Metric: AUC

To quantify how well our link-prediction model separates true prerequisite edges from spurious ones, we use the *Area Under the Receiver Operating Characteristic Curve* (AUC-ROC). Before presenting our results, we briefly recall the key definitions.

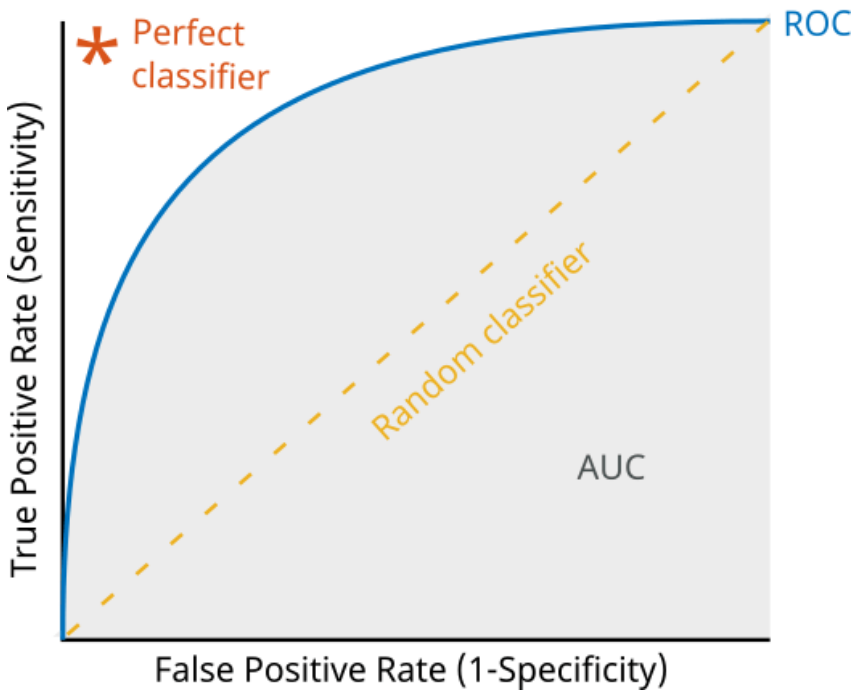


Figure 10.1: AUC-ROC Curve example

True Positive Rate (TPR) also called *sensitivity*, is the fraction of actual positives correctly identified:

$$\text{TPR} \;=\; \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}.$$

False Positive Rate (FPR) is the fraction of negatives that are mistakenly classified as positives:

$$\text{FPR} \;=\; \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}}.$$

The ROC curve plots TPR against FPR as the decision threshold on the model's score is varied. The *AUC* value, which is the area under this curve, measures the probability that a randomly chosen true edge ranks above a randomly chosen non-edge:

$$\text{AUC} = \int_0^1 \text{TPR}(t)\,\mathrm{d}\big(\text{FPR}(t)\big).$$

An AUC of 1.0 indicates perfect separation, 0.5 indicates random guessing, and any value below 0.5 denotes performance worse than chance.

Precision–Recall (PR) and Average Precision (AP) focus on the positive class: the PR curve plots precision versus recall, and the AP summarizes its area. High AP signals that most top-ranked suggestions are indeed valid prerequisites.

10.1.2 Training Dynamics

Beyond final scores, we inspect how the model learned over time. Figure 10.2 shows (a) the training vs. validation loss curves and (b) the validation AUC across epochs. The training loss steadily declines, while the validation loss follows suit with only a small gap, indicating min-

imal overfitting. The validation AUC rises rapidly in the first 20–30 epochs and then plateaus around 0.98. Early stopping (patience=10) halted training once no further AUC gains were observed, ensuring we selected the model with optimal generalization.

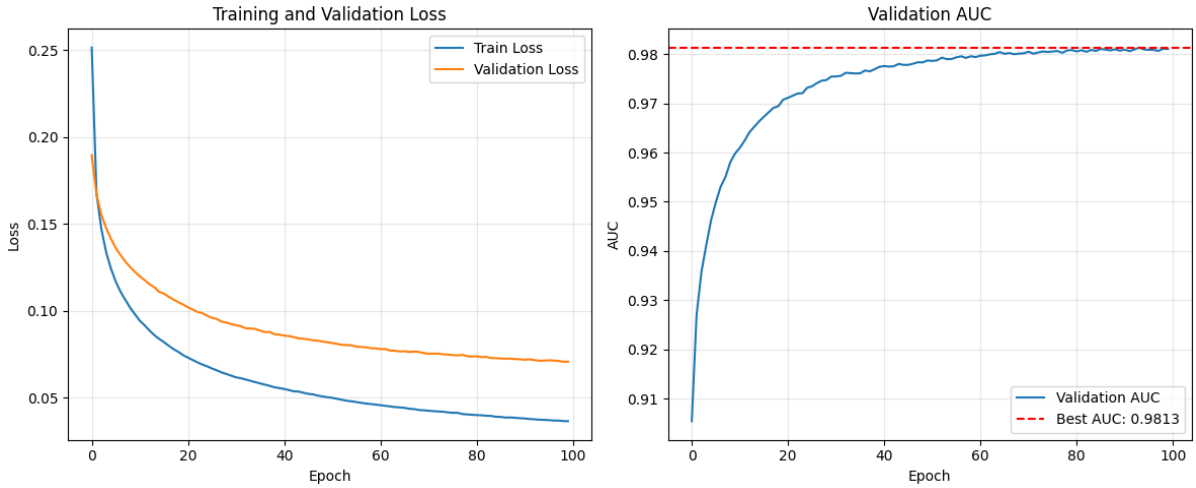


Figure 10.2: Graph of Training Metrics

10.1.3 Prediction Evaluation

From these plots, we can see that the the order-embedding projection converges smoothly and generalizes well.

In our experiments, we withheld 20% of true and sampled negative edges for validation. Figure 10.3 displays the resulting ROC curve, and the PR curve.

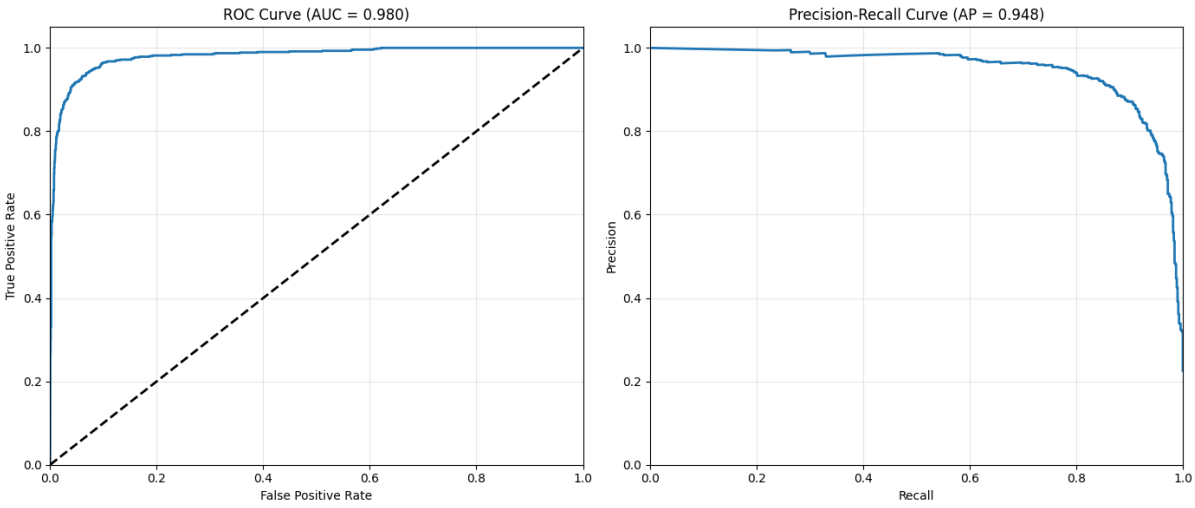


Figure 10.3: (a) ROC-AUC Curve (b) Precision-Recall Curve

The ROC curve in Figure 10.3 demonstrates that our model achieves very high discrimination between true prerequisite edges and non-edges. In particular, the fact that the true positive rate (TPR) exceeds 90 % at a false positive rate (FPR) below 10 % indicates that we can recover the vast majority of genuine prerequisites while introducing only a small number of spurious links. An AUC of 0.9803 means that if we randomly sample one real edge and one non-edge, the model will rank the real edge higher than the non-edge roughly 98 % of the time. This level of performance gives us confidence that the highest-ranked candidates for link prediction are overwhelmingly correct.

Equally important is the Precision–Recall curve in Figure 10.3, where an average precision (AP) of 0.9478 shows that nearly 95 % of the top-scoring suggestions are true prerequisites. In practical terms, if we present the user with the top 50 predicted links for manual review, we can expect fewer than 3 of them to be false positives. This high precision is crucial in our application, where every proposed edge directly affects the learner's curriculum, so we must minimize noise to preserve trust and usability.

10.2 Results and Analysis of Pathfinding Algorithms

10.2.1 Comparing Dynamic Programming and A* Search

We implemented both the Dynamic Programming (DP) and A* Search approaches across various job roles including Computer Vision Engineers, AI Architects, and NLP Specialists. As we evaluated the resulting learning paths, we quickly noticed that Dynamic Programming produced educationally superior paths, especially for skills requiring deep prerequisite knowledge. The NLP Engineer role offers a particularly clear illustration of these differences. Starting with knowledge of graph theory, both algorithms found paths to basic skills like computer science and natural language processing.

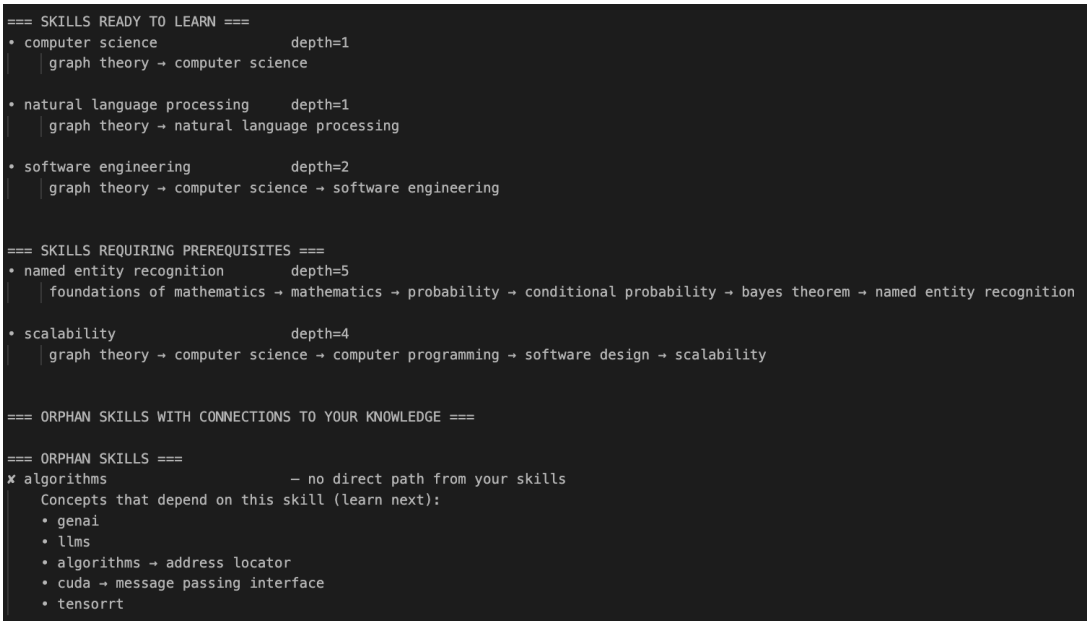


Figure 10.4: DP results for the Job Tile ”NLP Engineer”

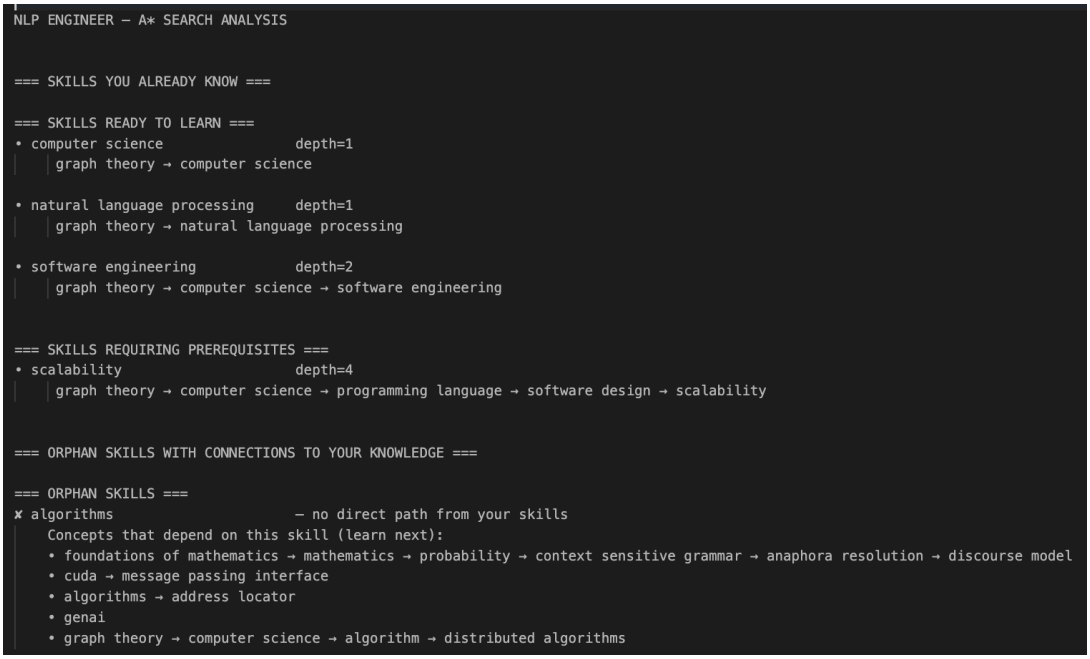


Figure 10.5: A* results for the Job Tile "NLP Engineer"

In Figures 10.4 and 10.5, $A \rightarrow B$ means "A is a prerequisite of B."

Let us look at: named entity recognition, a fundamental NLP skill. Dynamic Programming uncovered a coherent path through mathematical foundations:

> *foundations of mathematics $\rightarrow$ mathematics $\rightarrow$ probability $\rightarrow$ conditional probability $\rightarrow$ Bayes theorem $\rightarrow$ named entity recognition*

A* Search, despite its sophistication, couldn't find any path to this skill. This wasn't an isolated case; we observed similar patterns across multiple domains where DP discovered important paths that A* missed entirely.

Even when both algorithms found paths to the same skill, DP's paths better reflected how knowledge truly builds. For scalability, DP suggested following computer science with "computer programming" before software design, while A* recommended "programming language", a subtle but meaningful distinction that better aligns with typical educational progression.

10.2.2 Why Dynamic Programming Works Better

Five key factors explain DP's advantage in educational contexts:

First, DP's global perspective allows it to discover non-obvious paths that A* misses. By considering the entire knowledge graph simultaneously rather than searching directionally, DP uncovers connections that may initially seem counterintuitive but prove educationally sound, like the mathematical foundations needed for advanced NLP skills.

Second, A*'s semantic similarity heuristic can be misleading in educational pathfinding. Just because two concepts share terminology doesn't mean one leads naturally to the other. Named entity recognition and graph theory both deal with entities and relationships, but the learning journey between them requires mathematical foundations that semantic similarity doesn't capture.

Third, DP's topological processing naturally mirrors how education typically progresses from foundations to applications. This structural alignment gives DP an inherent advantage in creating paths that feel natural to educators and learners alike.

Our A* implementation uses a semantic-similarity–based heuristic to estimate the cost from any intermediate concept to the target. However, this heuristic is neither strictly admissible (it can overestimate the true remaining cost) nor guaranteed consistent across the DAG. As a result, A* may prematurely prune otherwise valid partial paths, skipping sequences that actually yield the optimal curriculum and, lacking reliable cost bounds, often explores nearly as many nodes as the full DP sweep. In practice, this undermines both the correctness guarantees and the efficiency gains that A* typically offers.

Finally, our DP implementation balances multiple educational factors rather than focusing

solely on path length. Learning isn't just about finding the shortest route, it's about finding the right route that builds knowledge properly.

Target Skill: Named Entity Recognition
Dynamic Programming Path:
foundations of mathematics → mathematics → probability → conditional probability → Bayes theorem → named entity recognition

A* Search Path:
No path found

Figure 10.6: Comparing algorithm results for a key NLP skill

10.2.3 Addressing Orphan Skills

Both algorithms identified similar orphan skills. Specialized technologies like Python, PyTorch, and TensorFlow that lacked clear learning paths from theoretical concepts. This consistent finding highlighted a legitimate gap in our knowledge graph: while theoretical hierarchies were well-developed, their connections to implementation technologies were insufficient.

While the identification of orphans was similar, DP's recommendations for what to learn after mastering these orphans showed better domain relevance. For "machine learning methods," DP suggested dependent concepts that maintained stronger thematic coherence within the target domain.

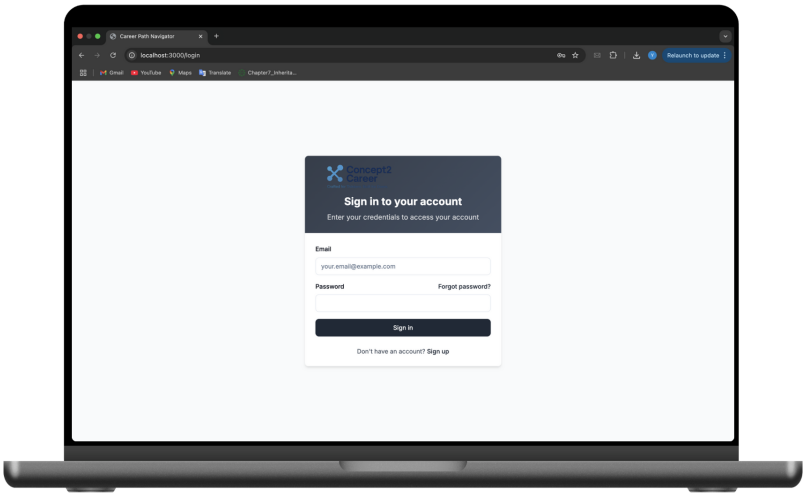
11 Web App Part

11.1 Frontend Layer Implementation

The Career Path Navigator's frontend is built using Next.js and React, providing a responsive and intuitive user interface that seamlessly integrates all system functionalities. The application follows a modern, component-based architecture with server-side rendering for improved performance. TypeScript ensures type safety throughout the codebase, while Tailwind CSS and shadcn/ui components deliver a consistent, accessible design system. The interface features ReactFlow for interactive skill graph visualizations, enabling users to explore learning paths through an intuitive node-based display. State management is handled efficiently through Zustand, maintaining persistent authentication states and user preferences across sessions.

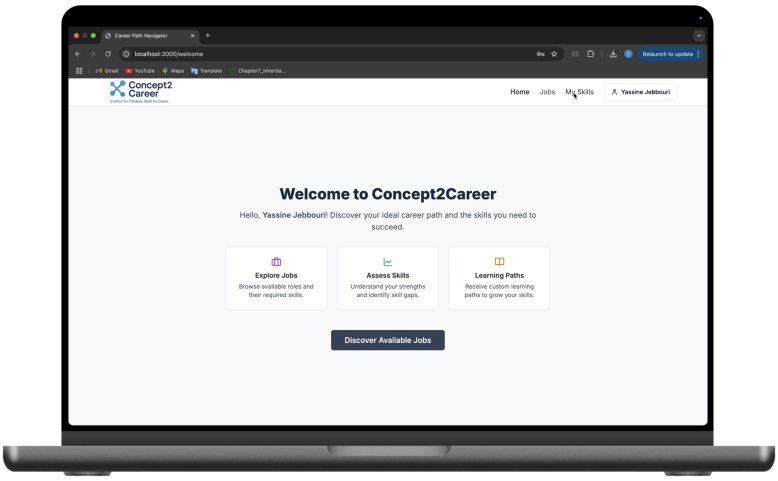
11.2 Application Interface Walkthrough

11.2.1 Login Page



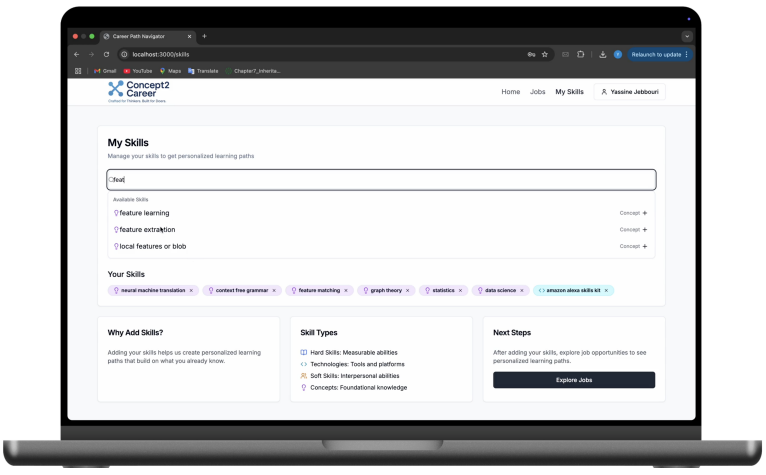
The login page provides a clean and secure entry point to the Career Path Navigator platform. Users can authenticate using their email and password credentials through the Firebase Authentication system. The interface includes form validation, a ”Forgot password?” link for account recovery, and a ”Sign up” option for new users. The Concept2Career branding is prominently displayed, along with clear instructions to guide users through the authentication process.

11.2.2 Welcome Page / Dashboard



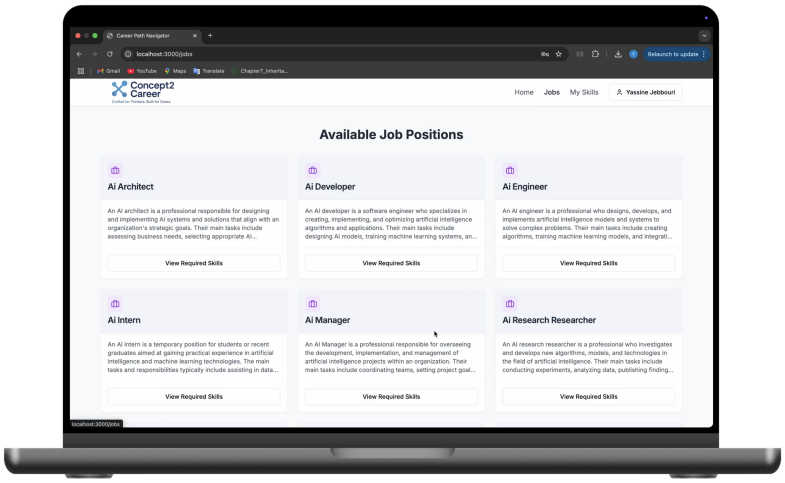
The welcome dashboard serves as the main navigation hub, greeting users by name and presenting three key action cards: ”Explore Jobs” for browsing available positions, ”Assess Skills” for managing personal skill inventories, and ”Learning Paths” for accessing customized training routes. The top navigation bar provides quick access to Home, Jobs, My Skills, and user profile options including logout functionality. This centralized design ensures users can efficiently access all platform features from a single entry point.

11.2.3 Skills Management Page



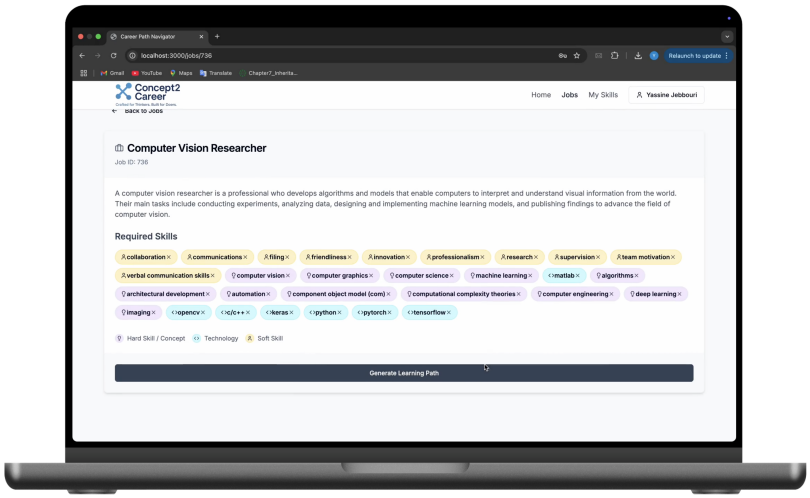
The My Skills page features a searchable interface where users can manage their skill inventory. Users can search for skills in the Available Skills section and add them to their personal collection through color-coded tags. Each skill is categorized by type (Hard Skills, Technologies, Soft Skills, Concepts) and includes remove functionality. The interface provides educational information about different skill types and guides users on how adding skills helps create personalized learning paths.

11.2.4 Job Listings



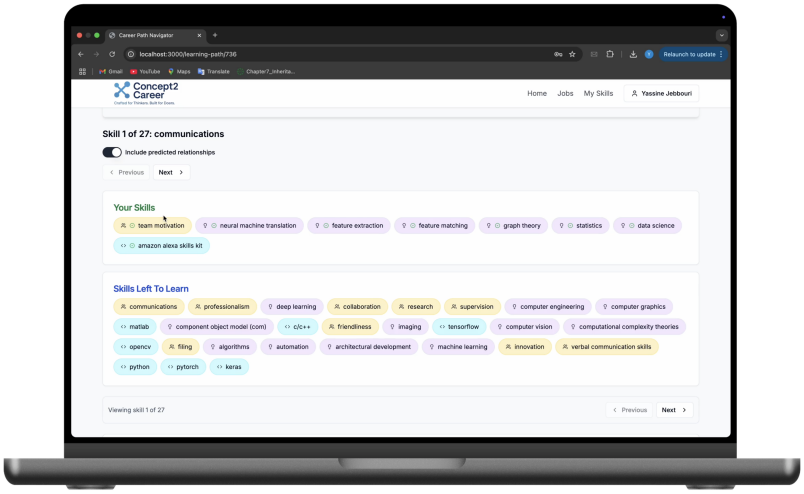
The Available Job Positions page displays a grid of career opportunities with clear descriptions. Each job card includes the title, detailed definition, and a ”View Required Skills” button. The page currently features AI-focused positions such as AI Architect, AI Developer, AI Engineer, AI Intern, AI Manager, and AI Research Researcher, demonstrating the platform’s specialization in emerging technology careers. The consistent layout allows users to quickly compare different positions and their requirements.

11.2.5 Job Details



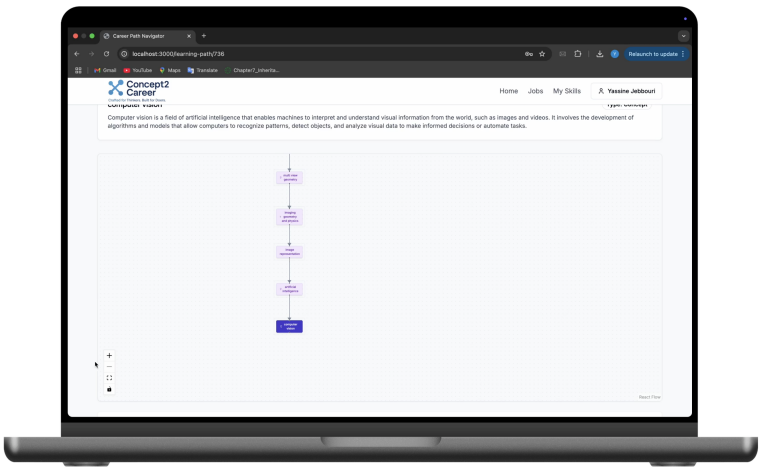
The job details page provides comprehensive information about selected positions, including the full job description and categorized required skills. Skills are displayed with color-coded tags distinguishing between communications (beige), technologies (blue), and domain knowledge (purple). The ”Generate Learning Path” button enables users to create personalized development plans. The page includes a back navigation option and clear categorization of skill types to help users understand the competencies needed for each role.

11.2.6 Learning Path Overview



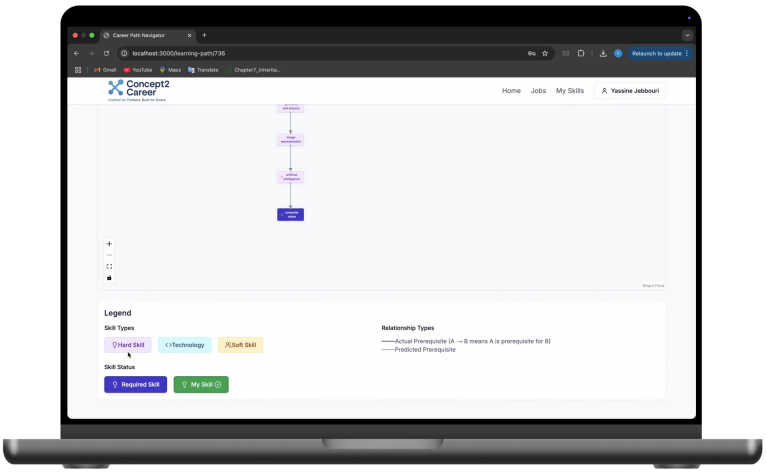
This interface shows the comparison between the user's current skills and required skills for the selected job. "Your Skills" displays existing competencies with color coding, while "Skills Left To Learn" highlights the gaps that need to be addressed. Users can navigate between different required skills using the Previous/Next controls. The "Include predicted relationships" toggle allows users to see AI-suggested skill connections beyond direct prerequisites.

11.2.7 Interactive Learning Path Visualization



The learning path visualization presents an interactive graph showing the prerequisite relationships between skills. Using ReactFlow, the interface displays nodes in different colors representing skill types (purple for hard skills, green for technologies). Arrows indicate prerequisite dependencies, creating a clear visual roadmap for skill acquisition. The interface includes zoom controls, pan functionality, and interactive elements that allow users to explore the learning path in detail.

11.2.8 Graph Legend



The legend provides clear explanations for the graph visualization elements. It defines skill types (Hard Skill, Technology, Soft Skill), skill status indicators (Required Skill vs My Skill), and relationship types (Actual Prerequisite vs Predicted Prerequisite). This comprehensive guide ensures users can properly interpret the learning path visualization and understand the logic behind skill progression recommendations.

Note: The complete source code for the Career2Path application is available at: `https://github.com/yassinejebbouri/career-path-navigator`

12 Conclusion

This project developed a structured framework for generating personalized learning paths by combining knowledge graphs, order embeddings, and score-driven pathfinding. Starting from curated prerequisite data and enriched with real-world job descriptions, we built an explainable system that helps learners chart clear, coherent trajectories toward AI and data science careers. This theoretical foundation was materialized into Career Path Navigator, a full-stack web application that brings intelligent career guidance to users through an intuitive, interactive platform.

The strengths of the approach are evident at both the algorithmic and application levels. The knowledge graph was cleaned and expanded effectively, removing less than 1% of edges while preserving essential structure. The trained order embedding model achieved high validation performance (AUC ≈ 0.98), showing it could accurately rank prerequisite relationships. The dynamic programming-based pathfinding produced layered, logically ordered learning paths that align with how skills typically build on one another. These components were successfully integrated into a modern web application architecture combining Next.js, React, FastAPI, and Neo4j, delivering a responsive and scalable platform for career development.

The Career Path Navigator application demonstrates the practical value of this research by providing users with interactive learning path visualizations, personalized skill gap analysis, and job-matching capabilities. Through its clean, type-safe codebase and robust backend infrastructure, the platform serves as a proof-of-concept for how graph-based learning systems can

be deployed in real-world career guidance scenarios.

While the project successfully demonstrates a complete learning path generation system, several limitations and opportunities for improvement are possible.

13 Limitations

The initial knowledge graph exhibited significant bias toward academic concepts, with fewer connections to applied tools and technologies, occasionally leaving practical skill gaps. Although link prediction helped address this imbalance, the system remains sensitive to missing or noisy data in concept definitions. The relatively small dataset constrains the comprehensiveness of our skill graph, limiting coverage across diverse technical domains.

Additionally, the prerequisite relationships between skills often encode vague hierarchies that may not accurately represent learning dependencies. Concepts that are hierarchically related (where concept A might be a subtopic of B) are sometimes incorrectly encoded as prerequisites, creating confusion in the learning path structure. For instance, ”machine learning” might be incorrectly positioned as a prerequisite for ”neural networks” when in reality it's better understood as a parent topic encompassing it.

The reliance on static snapshots of job markets presents another limitation: skills and technologies evolve rapidly, and a truly adaptive system would need continuous updates. Furthermore, while semantic similarity proved useful, it does not always capture the deeper structure of learning dependencies, particularly for complex technical relationships.

14 Future Work

One clear direction is the development of a **prerequisite annotation tool**. High-quality prerequisite data remains scarce and inconsistent across sources. Building a lightweight platform where domain experts can manually define or validate prerequisite relationships would not only expand the graph but also increase its reliability. This tool could integrate directly with the Neo4j database, allowing for continuous validation and refinement of skill hierarchies.

Enhancing the **link prediction models** presents another opportunity. While order embeddings provided a strong foundation, incorporating more sophisticated graph neural networks (such as GraphSAGE or GAT) could better capture local and global graph structures, particularly for skills at the periphery of our current graph.

Improving the integration between conceptual and applied knowledge remains a priority. While the current system links many theoretical concepts, the bridge to practical technologies such as frameworks, libraries, and tools needs strengthening. Expanding the data collection process to include curated technical documentation, project repositories, and course syllabi could help populate these missing connections.

Addressing prerequisite ambiguity requires developing more nuanced relationship types between skills. Rather than simple prerequisite links, the system could distinguish between true prerequisites, subtopics, related concepts, and co-requisites, providing clearer guidance for learners.

Finally, **dynamic graph updating** represents an important long-term goal. Instead of relying on static datasets, future versions could implement real-time ingestion of job postings and industry trends, continuously evolving the skill graph to reflect current demands. This would transform the Career Path Navigator from a static recommender into a truly adaptive career guidance platform that responds to the changing landscape of technology careers.

In summary, while the Career Path Navigator establishes a solid foundation for explainable learning path generation through its combination of algorithms and web application, its full potential will emerge from ongoing enrichment of the knowledge graph, improved relationship modeling, and continuous adaptation to evolving industry needs.

References

[1] Anas Aitouazi, Mohamed Amor, and Laila Benhlima. Skillner: Github repository. https://github.com/AnasAito/SkillNER, 2021. Accessed: 2025-04-20.

[2] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[3] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Durán, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Proceedings of NIPS*, 2013.

[4] X. Chen and Y. Wang. Lecturebank dataset for nlp and cv concept graphs. https://github.com/Yale-LILY/LectureBank, 2018. Accessed: 2025-04-20.

[5] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of NIPS*, 2017.

[6] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. In *IEEE Transactions on Systems Science and Cybernetics*, 1968.

[7] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[8] H. Liu and Y. Song. Concept prerequisite dataset. https://github.com/harrylclc/eaai17-cpr-recover, 2017. Accessed: 2025-04-20.

[9] Henry Liu and Push Singh. Conceptnet: A practical commonsense reasoning toolkit. In *BT Technology Journal*, 2004.

[10] Maximilian Nickel and Douwe Kiela. Poincaré embeddings for learning hierarchical representations. *Advances in Neural Information Processing Systems*, 30, 2017.

[11] Ivan Vendrov, Ryan Kiros, Sanja Fidler, and Raquel Urtasun. Order-embeddings of images and language. In *International Conference on Learning Representations (ICLR)*, 2016.

[12] Bowen Yuan, Thanh Dao, and Andrew McCallum. Lecturebank: A dataset of lecture slides for learning concept prerequisites. In *Proceedings of EMNLP*, 2018.

[13] Zhanqiu Zhang, Jianyu Cai, Yongdong Zhang, and Jie Wang. Learning hierarchy-aware knowledge graph embeddings for link prediction. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3744–3751, 2019.

A Appendix: Code Snapshots

This appendix contains screenshots of code relevant to the implementation of the system described in the report. Each figure corresponds to a specific component or function used in the project pipeline.

```
MATCH (j:Job {title: "machine learning engineer"})-[:REQUIRES]→(s)
WHERE s:HardSkill OR s:SoftSkill OR s:Technology
RETURN j, s
LIMIT 30
```

Figure A.1: Cypher Query to get skills required for Machine Learning Engineer

jobscraper5.py > ...

```python
import ssl
ssl._create_default_https_context = ssl._create_unverified_context

import undetected_chromedriver as uc
from selenium.webdriver.common.by import By
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.keys import Keys

import time
import pandas as pd
from urllib.parse import quote

# Set up headless browser
options = uc.ChromeOptions()
options.add_argument("--headless")
driver = uc.Chrome(version_main=134, options=options)

# Keywords to search for
search_keywords = [
    "data scientist", "AI research", "Reinforcement learning", "robotics"
]

results = []

# Loop through each keyword
for keyword in search_keywords:
    print(f"[SEARCHING] {keyword}")
    encoded_keyword = quote(keyword)
    url = f"https://hiring.cafe/?searchState=%7B%22defaultToUserLocation%22%3Afalse%2C%22searchQuery%22%3A%22{encoded_keyword}%22%7D"
    driver.get(url)
    time.sleep(3)

    # Scroll until 50 job cards are loaded or max scrolls reached
    scroll_attempts = 0
    max_scrolls = 10

    while scroll_attempts < max_scrolls:
        elems = driver.find_elements(By.CSS_SELECTOR, "span.font-bold.text-start.mt-1.line-clamp-2")
        if len(elems) >= 50:
            break
        driver.execute_script("window.scrollTo(0, document.body.scrollHeight);")
        time.sleep(4)
        scroll_attempts += 1
```

jobscraper5.py > ...

```python
    print(f"[INFO] Found {len(elems)} job cards for keyword '{keyword}'")

    cards = driver.find_elements(By.CSS_SELECTOR, "div.relative.bg-white.rounded-xl.border.border-gray-200.shadow")
    actions = ActionChains(driver)

    for idx, card in enumerate(cards[:50]):
        driver.execute_script("arguments[0].scrollIntoView(true);", card)
        time.sleep(2)

        try:
            title_elem = card.find_element(By.CSS_SELECTOR, "span.font-bold.text-start.mt-1.line-clamp-2")
            job_title = title_elem.text.strip()
        except:
            job_title = "N/A"

        try:
            tech_elem = card.find_element(By.CSS_SELECTOR, "span.line-clamp-2.font-light")
            tech_text = tech_elem.text.strip().strip('"')
        except:
            tech_text = "N/A"

        actions.move_to_element(card).move_by_offset(30, 30).click().perform()

        try:
            WebDriverWait(driver, 8).until(
                EC.presence_of_element_located((By.CSS_SELECTOR, "article.prose"))
            )
        except:
            pass

        try:
            article_elem = WebDriverWait(driver, 10).until(
                EC.visibility_of_element_located((By.CSS_SELECTOR, "article.prose"))
            )
            job_description = article_elem.text.strip()
        except:
            job_description = "N/A"

        results.append({
            "keyword_searched": keyword,
            "job_title": job_title,
            "technologies": tech_text,
            "description": job_description
        })
```


Figure A.2: Scraper code

```python
# === Cleaning functions ===
def is_english(text):
    try: return detect(text) == "en"
    except: return False

def remove_locations(text):
    doc = nlp(text)
    for ent in doc.ents:
        if ent.label_ == "GPE":
            text = text.replace(ent.text, "")
    return text

def remove_durations(text):
    return re.sub(r'\b\d{1,2}(\s?(to|-)?\s?\d{1,2})?\s?(month|months|year|years|yrs|yr)\b', '', text, flags=re.IGNORECASE)

def clean_job_title(title):
    title = re.sub(r"&amp;", "and", title)
    title = re.sub(r"\(.*?\)", "", title)
    title = re.sub(r"\b(senior|sr\.?|junior|jr\.?|lead|principal|staff|freelancer|contractor|vp|avp|consultant)\b", "", title, flags=re.IGNORECASE)
    title = remove_durations(title)
    title = re.sub(r"\b(remote|full[- ]?time|part[- ]?time|freelance|contract|temporary|hybrid|onsite)\b", "", title, flags=re.IGNORECASE)
    title = remove_locations(title)
    title = re.sub(r"[^a-zA-Z0-9\s\-/]", "", title)
    return re.sub(r"\s+", " ", title).strip().lower()

def get_skill_depth(skill):
    depth = 0
    while skill_hierarchy.get(skill):
        skill = skill_hierarchy[skill][0]
        depth += 1
    return depth

def get_soft_vector(text, categories, use_hierarchy=False):
    vec = np.zeros(len(categories))
    for i, (cat, terms) in enumerate(categories.items()):
        match_score = sum(1 for term in terms if term in text)
        if match_score:
            weight = match_score * (get_skill_depth(cat)+1 if use_hierarchy else 1)
            vec[i] = weight
    return vec / vec.sum() if vec.sum() else vec

# === Run pipeline ===
skill_list = list(skill_hierarchy.keys())
job_titles = df["job_title"].dropna().unique()
job_titles_en = [t for t in job_titles if is_english(t)]

cleaned_titles = []
feature_vectors = []

for t in job_titles_en:
    cleaned = clean_job_title(t)
    skill_vec = get_soft_vector(cleaned, skill_groups, use_hierarchy=True)
    role_vec = get_soft_vector(cleaned, job_roles, use_hierarchy=False)
    final_vec = np.hstack([skill_vec, role_vec])
    cleaned_titles.append(cleaned)
    feature_vectors.append(final_vec)

X = np.vstack(feature_vectors)
X_scaled = StandardScaler().fit_transform(X)
dist_matrix = 1 - cosine_similarity(X_scaled)
linked = linkage(dist_matrix, method='average')
```

Figure A.3: Code for clustering logic

```python
class OrderEmbeddingModel(nn.Module):
    """Order embedding model for learning prerequisite relationships"""
    def __init__(self, input_dim, hidden_dim):
        super(OrderEmbeddingModel, self).__init__()
        self.projection = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU()
        )

    def forward(self, x):
        return self.projection(x)

class PrerequisiteDataset(Dataset):
    """Dataset for training order embeddings"""
    def __init__(self, pairs, node_embeddings):
        self.pairs = pairs  # list of (u, v, label) triples
        self.node_embeddings = node_embeddings

    def __len__(self):
        return len(self.pairs)

    def __getitem__(self, idx):
        u, v, label = self.pairs[idx]
        return self.node_embeddings[u], self.node_embeddings[v], torch.tensor(label, dtype=torch.float32)

def order_violation(u_emb, v_emb):
    """Compute order embedding violation: d_order(u, v) = sum(max(0, v_k - u_k))"""
    return torch.sum(torch.clamp(v_emb - u_emb, min=0))
```

Figure A.4: Code for Order Embedding Calculation

```python
def train_order_embeddings(train_pairs, val_pairs, node_embeddings,
                           input_dim=128, hidden_dim=64, margin=1.0,
                           learning_rate=0.001, epochs=100, batch_size=64,
                           early_stopping_patience=10):
    """Train order embedding model on positive and negative pairs with validation"""
    # Create datasets and dataloaders
    train_dataset = PrerequisiteDataset(train_pairs, node_embeddings)
    val_dataset = PrerequisiteDataset(val_pairs, node_embeddings)

    train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
    val_dataloader = DataLoader(val_dataset, batch_size=batch_size)

    # Create model and optimizer
    model = OrderEmbeddingModel(input_dim, hidden_dim)
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    # Metrics tracking
    train_losses = []
    val_losses = []
    val_aucs = []

    # Early stopping
    best_val_auc = 0
    best_model_state = None
    patience_counter = 0

    logging.info(f"Training order embeddings for {epochs} epochs...")

    for epoch in tqdm(range(epochs), desc="Training"):
        # Training
        model.train()
        epoch_train_loss = 0

        for u_emb, v_emb, labels in train_dataloader:
            # Forward pass
            u_proj = model(u_emb)
            v_proj = model(v_emb)

            # Compute loss
            violations = torch.stack([order_violation(u, v) for u, v in zip(u_proj, v_proj)])
            positive_loss = labels * violations
            negative_loss = (1 - labels) * torch.clamp(margin - violations, min=0)
            loss = torch.mean(positive_loss + negative_loss)

            # Backward pass and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
```

```python
    avg_train_loss = epoch_train_loss / len(train_dataloader)
    train_losses.append(avg_train_loss)

    # Validation
    model.eval()
    epoch_val_loss = 0
    val_preds = []
    val_true = []

    with torch.no_grad():
        for u_emb, v_emb, labels in val_dataloader:
            # Forward pass
            u_proj = model(u_emb)
            v_proj = model(v_emb)

            # Compute loss
            violations = torch.stack([order_violation(u, v) for u, v in zip(u_proj, v_proj)])
            positive_loss = labels * violations
            negative_loss = (1 - labels) * torch.clamp(margin - violations, min=0)
            loss = torch.mean(positive_loss + negative_loss)

            epoch_val_loss += loss.item()

            # Store predictions (negative violations as scores) and true labels
            val_preds.extend((-violations).tolist())
            val_true.extend(labels.tolist())

    avg_val_loss = epoch_val_loss / len(val_dataloader)
    val_losses.append(avg_val_loss)

    # Compute validation AUC
    val_auc = roc_auc_score(val_true, val_preds)
    val_aucs.append(val_auc)

    # Early stopping check
    if val_auc > best_val_auc:
        best_val_auc = val_auc
        best_model_state = model.state_dict()
        patience_counter = 0
    else:
        patience_counter += 1

    if patience_counter >= early_stopping_patience:
        logging.info(f"Early stopping triggered at epoch {epoch+1}")
        break

    if (epoch + 1) % 10 == 0 or epoch == 0:
        logging.info(f"Epoch {epoch+1}/{epochs}, Train Loss: {avg_train_loss:.4f}, Val Loss: {avg_val_loss:.4f}, Val AUC: {val_auc:.4f}")

# Load best model
if best_model_state is not None:
    model.load_state_dict(best_model_state)
```

Figure A.5: Training code

```python
def predict_links_for_loose_skills(G, model, node_embeddings, loose_skills, node_id_to_properties,
                                   threshold=-0.3, top_k=7):
    """Predict links for loose skills using order embeddings"""
    if not loose_skills:
        logging.info("No loose skills found, skipping link prediction")
        return pd.DataFrame()

    # Estimate node layers
    node_layers = estimate_node_layers(G)

    # Find job nodes to exclude from candidates
    job_nodes = [n for n, attrs in G.nodes(data=True) if "Job" in attrs.get("labels", [])]

    # Predict links for loose skills
    predicted_links = []
    model.eval()

    for loose_skill in loose_skills:
        skill_layer = node_layers.get(loose_skill, 0)
        candidates = []

        # Find candidate concepts/skills with lower layer than the loose skill
        for concept in G.nodes():
            if concept != loose_skill and concept not in job_nodes:
                concept_layer = node_layers.get(concept, 0)
                if concept_layer < skill_layer:
                    candidates.append(concept)

        if not candidates:
            continue

        # Score each candidate
        candidate_scores = []

        with torch.no_grad():
            loose_skill_emb = node_embeddings[loose_skill]
            loose_skill_proj = model(loose_skill_emb.unsqueeze(0)).squeeze(0)

            for concept in candidates:
                # Get concept embedding
                concept_emb = node_embeddings[concept]
                concept_proj = model(concept_emb.unsqueeze(0)).squeeze(0)

                # Calculate order embedding score (negative violation = more plausible)
                score = -order_violation(concept_proj, loose_skill_proj).item()

                # Check if adding this edge would create a cycle
                G.add_edge(concept, loose_skill)
                has_cycle = nx.has_path(G, loose_skill, concept)
                G.remove_edge(concept, loose_skill)

                if not has_cycle:
                    candidate_scores.append((concept, score))

        # Sort candidates by score and take top-k
        candidate_scores.sort(key=lambda x: x[1], reverse=True)
        top_candidates = [
            (concept, score) for concept, score in candidate_scores[:top_k]
            if score >= threshold
        ]

        # Add to predicted links
        for concept, score in top_candidates:
            predicted_links.append({
                'loose_skill_id': loose_skill,
                'loose_skill_name': node_id_to_properties[loose_skill]['name'],
                'concept_id': concept,
                'concept_name': node_id_to_properties[concept]['name'],
                'score': score
            })
```

Figure A.6: Code for Link Prediction

```python
# ──────────────────────────────── DYNAMIC PROGRAMMING CELL ────────────────────────────────
# Run the common code cell first before executing this cell

# 3) DP: longest + best-score path to every node with improved path quality
dp = {n: -math.inf for n in H}
prev = {}
path_lens = {n: 0 for n in H}  # Keep track of path lengths

# seed roots: any Concept/HardSkill with zero prerequisites
for n in H:
    if H.in_degree(n) == 0 and (CONCEPT in label_of[n] or HARDSKILL in label_of[n]):
        dp[n] = 0.0
        path_lens[n] = 1

# Improved DP prioritizing path quality and reasonable length
for u in nx.topological_sort(H):
    if dp[u] < -1e8:
        continue
    for _, v, d in H.out_edges(u, data=True):
        s_norm = d["raw"] / Smax if Smax else 0.0
        sem_pen = (1 - cosine(emb[u], emb[v])) if (u in emb and v in emb) else 1.0

        # Add length penalty to discourage excessively long paths
        length_factor = LENGTH_PENALTY * path_lens[u]

        gain = Δ - α * s_norm - β * sem_pen - length_factor
        cand = dp[u] + gain

        # Only consider if path length is reasonable
        new_path_len = path_lens[u] + 1
        if new_path_len <= MAX_PATH_LENGTH and cand > dp[v]:
            dp[v], prev[v] = cand, u
            path_lens[v] = new_path_len

# back-track chain builder with length consideration
def build_chain_dp(n):
    if dp.get(n, -math.inf) < -1e8:
        return None
    path = [n]
    while path[-1] in prev:
        path.append(prev[path[-1]])
        if len(path) > MAX_PATH_LENGTH:
            break  # Enforce maximum path length
    return list(reversed(path))

# Function to find orphan skills' dependents (for skills without paths)
def find_orphan_dependents(sid):
    deps = edges.loc[edges[":END_ID"] == sid, ":START_ID"].astype(int)
    cands = [d for d in deps if (CONCEPT in label_of.get(d,()) or HARDSKILL in label_of.get(d,()))]

    scored = []
    for d in cands:
        r = edges[(edges[":START_ID"]==d)&(edges[":END_ID"]==sid)].iloc[0]
        sc = abs(float(r["score:float"])) + (PRED if r["predicted:boolean"] else 0.0)
        scored.append((d, sc))
    scored.sort(key=lambda x: x[1])

    return [d for d, _ in scored[:SUGGEST_K]]
```

Figure A.7: Code for Pathfinding using Dynamic Programming