BỘ GIÁO DỤC VÀ ĐÀO TẠO

ĐẠI HỌC KINH TẾ TP. HỒ CHÍ MINH (UEH)

TRƯỜNG CÔNG NGHỆ VÀ THIẾT KẾ

UEH UNIVERSITY

BÁO CÁO DỰ ÁN CUỐI KỲ

MÔN HỌC: LẬP TRÌNH PHÂN TÍCH DỮ LIỆU

THUẬT TOÁN DIJKSTRA TÌM ĐƯỜNG ĐI NGẮN NHẤT GIỮA HAI ĐỈNH BẤT KỲ TRÊN ĐỒ THỊ VÔ HƯỚNG

Giảng viên	:	TS. Nguyễn An Tế
Mã học phần	:	25C1INF50907001
Lớp	:	CT5 – B2.101
Thành viên nhóm	:	Thái Hoài An
		Dương Phương Anh
		Hoàng Thụy Hồng Ân
		Nguyễn Đức Tuấn Anh
		Nguyễn Thị Minh Anh

TP. Hồ Chí Minh - Tháng 10 năm 2025

Lời cảm ơn

Nhóm xin gửi lời cảm ơn chân thành và sâu sắc nhất đến **Thầy Nguyễn An Tế** – giảng viên phụ trách môn học Lập trình phân tích dữ liệu ngành Khoa học Dữ liệu. Trong suốt quá trình học tập thầy đã tận tình hướng dẫn, định hướng tư duy và truyền đạt cho nhóm những kiến thức về lập trình, xử lý dữ liệu và các phương pháp phân tích trong lĩnh vực Khoa học Dữ liệu.

Sự chỉ bảo tận tâm và nghiêm túc của thầy không chỉ giúp nhóm hiểu rõ hơn về lý thuyết và kỹ thuật lập trình, mà còn khơi gợi tinh thần nghiên cứu, khả năng tư duy phản biện. Thông qua môn học, nhóm đã có cơ hội vận dụng kiến thức vào thực tiễn, củng cố kỹ năng làm việc nhóm, cũng như nâng cao năng lực tư duy logic và kỹ năng giải quyết vấn đề trên nền tảng dữ liệu.

Cuối cùng, nhóm mong rằng thầy Nguyễn An Tế luôn dồi dào sức khỏe, thành công trong sự nghiệp giảng dạy và tiếp tục truyền cảm hứng học tập, nghiên cứu đến các thế hệ sinh viên sau này.

Nhóm sinh viên thực hiện

Danh mục bảng biểu

Danh mục hình vẽ

Mục lục

Chương 1

Giới thiệu đề tài

1.1 Sơ lược đề tài

Hiện nay, các bài toán liên quan đến tối ưu hóa đường đi ngày càng trở nên quan trọng. Ở mức độ lý thuyết, những bài toán này được mô hình hóa bằng đồ thị (graph). Việc tìm đường đi ngắn nhất giữa hai điểm bất kỳ trên đồ thị là một bài toán kinh điển, có nhiều ứng dụng thực tiễn trong nhiều lĩnh vực như:

- Mạng máy tính: giúp tối ưu hóa đường truyền dữ liệu giữa các nút mạng.
- Giao thông và quy hoạch đô thị: hỗ trợ tìm tuyến đường di chuyển ngắn nhất giữa hai địa điểm.
- Logistics: giúp giảm chi phí và thời gian vận chuyển hàng hóa.
- Định tuyến trong hệ thống bản đồ số: cung cấp lộ trình tối ưu cho người dùng.

Trong số các thuật toán giải quyết bài toán tìm đường đi ngắn nhất, thuật toán **Dijkstra** do Dijkstra (1959) đề xuất là một trong những phương pháp hiệu quả và phổ biến nhất để giải quyết những bài toán trên. Thuật toán hoạt động dựa trên nguyên tắc chọn dần các đỉnh có chi phí nhỏ nhất để mở rộng đường đi, đảm bảo tìm được lời giải tối ưu trong đồ thị có trọng số không âm.

Đề tài **"Thuật toán Dijkstra tìm đường đi ngắn nhất giữa hai đỉnh bất kỳ trên đồ thị vô hướng"** được thực hiện nhằm mô phỏng quá trình hoạt động của thuật toán Dijkstra trong việc xác định lộ trình ngắn nhất, qua đó thể hiện ứng dụng của lý thuyết đồ thị trong các bài toán tối ưu hóa đường đi.

1.2 Mục tiêu của đề tài

Mục tiêu của đề tài là xây dựng một chương trình Python nhằm minh họa cho thuật toán Dijkstra, giúp tìm đường đi ngắn nhất giữa hai đỉnh bất kỳ trên một đồ thị vô hướng, với dữ liệu đầu vào được đọc từ tệp CSV. Đề tài tập trung vào các nội dung chính sau:

1. Cài đặt thuật toán Dijkstra theo hướng đối tượng.
2. Xác định và hiển thị đường đi ngắn nhất giữa hai đỉnh bất kỳ trong đồ thị.
3. Tạo minh họa trực quan cách thuật toán Dijkstra hoạt động.
4. So sánh hiệu quả của thuật toán Dijkstra với các thuật toán tìm đường cơ bản khác như Astar, DFS, BFS.

1.3 Phạm vi nghiên cứu

Phạm vi nghiên cứu của đề tài là sử dụng thuật toán Dijkstra để tìm đường đi ngắn nhất giữa hai đỉnh bất kỳ trong đồ thị vô hướng có trọng số. Thuật toán này đảm bảo tìm được đường đi tối ưu trong trường hợp tất cả các trọng số cạnh đều không âm, đồng thời có độ phức tạp phù hợp cho các đồ thị có quy mô vừa và nhỏ.

Đề tài giới hạn trong phạm vi đồ thị vô hướng, tức là các cạnh không có hướng. Mỗi cạnh đều có trọng số biểu thị chi phí hoặc độ dài đường đi giữa hai đỉnh. Dữ liệu đầu vào của chương trình được đọc từ tệp `Graph.csv`, trong đó mỗi hàng biểu diễn một cạnh dưới dạng bộ ba `(v_from, v_to, weight)`.

Bên cạnh đó, phạm vi của đề tài không xem xét các đồ thị có hướng hoặc đồ thị có trọng số âm, vì thuật toán Dijkstra không đảm bảo tính tối ưu trong các trường hợp này. Không tập trung vào tối ưu hóa hiệu năng cho các đồ thị có quy mô rất lớn, mà chỉ hướng đến việc minh họa thuật toán và trực quan hóa kết quả. Hơn nữa là không đề cập đến việc triển khai trên môi trường đa luồng hoặc song song hóa thuật toán.

Chương 2

Cơ sở lý thuyết

2.1 Lý thuyết đồ thị

2.1.1 Đỉnh và Cạnh

Lý thuyết đồ thị là một nhánh của Toán học rời rạc nghiên cứu các mối quan hệ giữa các đối tượng thông qua cấu trúc đồ thị. Theo Bondy (2008), một đồ thị có thể được định nghĩa là một cặp có thứ tự G = (V, E), trong đó V là tập hợp hữu hạn và không rỗng các đỉnh, còn E là tập hợp các cạnh biểu diễn mối liên kết giữa các đỉnh trong V. Mỗi cạnh e E được xác định bởi hai phần tử của tập V, gọi là hai đầu mút của cạnh.

Nói cách khác, cạnh thể hiện mối quan hệ hoặc sự kết nối giữa hai đỉnh. Ví dụ, xét đồ thị G với tập đỉnh V = a, b, c và tập cạnh E = a, b, b, c. Khi đó, đồ thị G có hai cạnh, nối đỉnh a với b và nối đỉnh b với c. Cấu trúc này có thể được biểu diễn trực quan thông qua sơ đồ các điểm và đường nối, trong đó điểm đại diện cho đỉnh và đường nối biểu diễn cho cạnh.

Hình 2.1: Đồ thị G với các đỉnh a, b, c và các cạnh ab, bc.

2.1.2 Đồ thị có hướng và Đồ thị vô hướng

West (2001) phân loại đồ thị thành hai dạng cơ bản là đồ thị có hướng và đồ thị vô hướng. Đồ thị có hướng (directed graph) là đồ thị trong đó mỗi cạnh được gán một hướng cụ thể. Mỗi cạnh có thể được biểu diễn bằng một cặp có thứ tự (u, v), trong đó u là đỉnh nguồn và v là đỉnh đích. Cạnh (u, v) thể hiện một quan hệ một chiều, nghĩa là chỉ có thể di chuyển từ u đến v. Các cạnh trong loại đồ thị này thường được minh họa bằng mũi tên chỉ hướng. Đồ thị có hướng thường được sử dụng để mô tả các mối quan hệ không đối xứng, chẳng hạn trong cây gia phả, nếu A là cha của B thì không có nghĩa B là cha của A. Loại đồ thị này cũng được ứng dụng trong biểu diễn luồng dữ liệu, sơ đồ quy trình hoặc mạng thông tin, trong đó hướng của cạnh thể hiện chiều truyền thông tin hoặc sự phụ thuộc giữa các phần tử.

Ngược lại, đồ thị vô hướng (undirected graph) là đồ thị trong đó các cạnh không mang hướng. Mỗi cạnh là một cặp không có thứ tự u, v, nghĩa là nếu có cạnh nối giữa u và v thì việc di chuyển từ v về u cũng được coi là tương đương. Do đó, các quan hệ trong đồ thị vô hướng mang tính đối xứng. Đồ thị vô hướng thường được sử dụng để mô hình hóa các quan hệ hai chiều hoặc tương hỗ, ví dụ như trong mạng xã hội, nếu A là bạn của B thì B cũng là bạn của A. Một số ví dụ điển hình khác của đồ thị vô hướng là mạng máy tính với kết nối hai chiều giữa các thiết bị, hoặc các mô hình giao thông trong đó đường đi giữa hai địa điểm có thể được sử dụng theo cả hai chiều.

2.1.3 Cạnh có trọng số

Trong một đồ thị trọng số, mỗi cạnh e được gán một giá trị số gọi là trọng số. Theo Bondy (2008), trọng số của cạnh biểu thị một đại lượng định lượng liên quan đến mối quan hệ giữa hai đỉnh, có thể là độ dài, chi phí, thời gian hoặc năng lượng cần thiết để di chuyển giữa hai đỉnh đó. Thông thường, các trọng số được giả định là các số thực không âm, đặc biệt trong các bài toán tối ưu hóa đường đi ngắn nhất.

Cormen et al. (2009) cho rằng trọng số của cạnh đóng vai trò như chi phí của phép di chuyển dọc theo cạnh đó, và tổng trọng số của một đường đi được định nghĩa là tổng của tất cả các trọng số của các cạnh thuộc đường đi đó. Do đó, độ dài của một đường đi trong đồ thị trọng số chính là tổng chi phí di chuyển qua các cạnh của nó.

Các thuật toán tìm đường đi ngắn nhất, như thuật toán Dijkstra, hoạt động dựa trên nguyên tắc tối thiểu hóa tổng trọng số trên đường đi. Thuật toán Dijkstra đặc biệt thích hợp cho các đồ thị có trọng số không âm, nơi nó xác định được đường đi có tổng chi phí nhỏ nhất giữa hai đỉnh (West, 2001). Nhờ đặc tính này, đồ thị có trọng số

và các thuật toán đi kèm được ứng dụng rộng rãi trong nhiều lĩnh vực thực tiễn như lập kế hoạch lộ trình trong mạng lưới giao thông, tối ưu hóa luồng dữ liệu trong mạng máy tính, và mô hình hóa chi phí trong các hệ thống logistics.

2.1.4 Đường đi và Độ dài đường đi

Theo Bondy (2008), một đường đi trong đồ thị được định nghĩa là một dãy hữu hạn các đỉnh được nối liền bởi các cạnh. Cụ thể, một đường đi P có thể được biểu diễn dưới dạng $P = (u_0, u_1, u_2, \ldots, u_k)$, trong đó mỗi cặp đỉnh liên tiếp (u_i, u_{i+1}) thuộc tập cạnh E. Các đỉnh u_0 và u_k lần lượt được gọi là đỉnh bắt đầu và đỉnh kết thúc của đường đi. Trong hầu hết các trường hợp, người ta thường xem xét các đường đi đơn (simple path), nghĩa là không có đỉnh nào được lặp lại nhằm tránh hình thành các chu trình không cần thiết.

Nếu đỉnh đầu và đỉnh cuối của đường đi trùng nhau, tức $u_0 = u_k$ và $k \geq 3$, thì đường đi đó được gọi là một chu trình (cycle) hay vòng khép kín. Khái niệm này có vai trò quan trọng trong việc xác định cấu trúc và tính liên thông của đồ thị.

Về độ dài đường đi, Cormen et al. (2009) định nghĩa rằng trong đồ thị không trọng số, độ dài của một đường đi chính là số lượng cạnh mà nó bao gồm, hay tương đương với số bước di chuyển cần thiết để đi từ đỉnh đầu đến đỉnh cuối. Ngược lại, trong đồ thị có trọng số, độ dài của một đường đi được tính bằng tổng các trọng số của các cạnh thuộc đường đi đó.

Đường đi ngắn nhất giữa hai đỉnh là đường đi có tổng trọng số hoặc số cạnh nhỏ nhất trong tất cả các đường đi có thể giữa hai đỉnh đó. Các thuật toán như Dijkstra hoặc Bellman–Ford được thiết kế nhằm xác định đường đi ngắn nhất theo tiêu chí này, tùy thuộc vào tính chất của trọng số cạnh trong đồ thị.

2.2 Bài toán tìm đường đi ngắn nhất một nguồn

2.2.1 Tổng quan

Bài toán tìm đường đi ngắn nhất là một trong những vấn đề trung tâm của lý thuyết đồ thị và khoa học máy tính hiện đại. Theo Cormen et al. (2009), mục tiêu của bài toán là xác định đường đi có tổng trọng số nhỏ nhất giữa hai đỉnh bất kỳ trong một đồ thị, nơi mỗi cạnh được gán một giá trị biểu thị chi phí, khoảng cách hoặc thời gian di chuyển.

Vấn đề này xuất hiện trong nhiều bối cảnh thực tế, bao gồm lập kế hoạch lộ trình trong hệ thống giao thông (với các nút là giao điểm và trọng số là khoảng cách hoặc

thời gian di chuyển), tối ưu hóa truyền dẫn dữ liệu trong mạng máy tính, và lập lịch các nhiệm vụ trong quản lý dự án. Trong các bài toán thực tế này, việc xác định đường đi tối ưu giúp giảm thiểu chi phí, thời gian hoặc tài nguyên sử dụng, qua đó nâng cao hiệu quả vận hành của toàn hệ thống.

Một đồ thị có thể là có hướng hoặc vô hướng, có trọng số hoặc không trọng số. Trong trường hợp không trọng số, đường đi ngắn nhất giữa hai đỉnh là đường có số lượng cạnh ít nhất. Ngược lại, khi đồ thị có trọng số, đặc biệt là khi có trọng số âm, bài toán trở nên phức tạp hơn do yêu cầu phải kiểm soát các chu trình âm.

2.2.2 Định nghĩa bài toán

Cho một đồ thị $G = (V, E)$, trong đó V là tập các đỉnh và E là tập các cạnh. Mỗi cạnh $(u, v) \in E$ được gán một trọng số $w(u, v)$, là một số thực không âm biểu thị chi phí hoặc khoảng cách giữa hai đỉnh u và v.

Bài toán tìm đường đi ngắn nhất từ một nguồn (Single Source Shortest Path – SSSP) được phát biểu như sau: Cho một đỉnh nguồn $s \in V$, xác định độ dài đường đi ngắn nhất từ s đến mọi đỉnh $v \in V$. Độ dài của một đường đi $P = (v_0, v_1, \ldots, v_k)$ từ $s = v_0$ đến $t = v_k$ được xác định theo công thức:

$$\delta(s, t) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Mục tiêu là tìm đường đi sao cho $\delta(s, t)$ đạt giá trị nhỏ nhất cho mọi đỉnh $t \in V$.

2.2.3 Giả thuyết và thách thức chính

Theo Cormen et al. (2009), bài toán SSSP chỉ có lời giải hữu hạn khi đồ thị không chứa chu trình âm. Nếu tồn tại một chu trình có tổng trọng số âm và có thể đi đến được từ đỉnh nguồn s, thì không tồn tại đường đi ngắn nhất hữu hạn, bởi việc lặp lại chu trình đó vô hạn lần sẽ liên tục làm giảm tổng chi phí.

Trong các đồ thị có trọng số không âm, việc tính toán trở nên đơn giản hơn và cho phép áp dụng các thuật toán hiệu quả như thuật toán Dijkstra. Tuy nhiên, nếu đồ thị có trọng số âm, cần áp dụng các thuật toán đặc biệt như Bellman–Ford, có khả năng phát hiện chu trình âm và vẫn tìm được kết quả chính xác khi tồn tại các cạnh có trọng số âm nhưng không có chu trình âm.

2.3 Các thuật toán cốt lõi

2.3.1 Thuật toán Dijkstra (1959, Edsger W. Dijkstra)

Thuật toán Dijkstra, được đề xuất bởi Edsger W. Dijkstra vào năm 1959, là một trong những phương pháp hiệu quả nhất để giải quyết bài toán tìm đường đi ngắn nhất trong đồ thị có trọng số không âm Dijkstra (1959). Ý tưởng của thuật toán dựa trên chiến lược tham lam (greedy strategy), trong đó ở mỗi bước, thuật toán lựa chọn đỉnh có khoảng cách tạm thời nhỏ nhất từ nguồn và cố định giá trị khoảng cách này.

Thuật toán duy trì một hàng đợi ưu tiên (priority queue) chứa các đỉnh chưa được xử lý, được sắp xếp theo giá trị khoảng cách ngắn nhất tạm thời từ đỉnh nguồn. Ở mỗi bước, đỉnh có khoảng cách nhỏ nhất được lấy ra khỏi hàng đợi, và tất cả các đỉnh kề được cập nhật thông qua phép thư giãn cạnh (edge relaxation). Phép thư giãn có dạng:

$$d(v) > d(u) + w(u, v) \;\Rightarrow\; d(v) \leftarrow d(u) + w(u, v)$$

Khi một đỉnh được xử lý xong, khoảng cách của nó là cố định và không cần được cập nhật thêm. Tính đúng đắn của thuật toán được chứng minh dựa trên "tính chất tham lam tối ưu" (optimal substructure property): mọi đường con của một đường đi ngắn nhất cũng là đường đi ngắn nhất giữa hai đỉnh tương ứng Cormen et al. (2009).

Thuật toán Dijkstra hoạt động hiệu quả trên các đồ thị có trọng số không âm, với độ phức tạp thời gian $O((|V| + |E|) \log |V|)$ khi cài đặt bằng hàng đợi ưu tiên dạng heap. Tuy nhiên như đã đề cập ở trên, nó không áp dụng được cho đồ thị có trọng số âm, bởi giả định cơ bản là tổng khoảng cách chỉ tăng dần khi thuật toán tiến hành.

2.3.2 Thuật toán Bellman–Ford (1958, Richard Bellman và Lester Ford)

Thuật toán Bellman–Ford được phát triển gần như song song bởi Bellman (1958) và Ford (1956). Khác với Dijkstra, thuật toán này có thể xử lý các đồ thị có cạnh mang trọng số âm, miễn là không tồn tại chu trình âm có thể đi tới từ đỉnh nguồn.

Nguyên lý hoạt động của thuật toán dựa trên việc lặp lại quá trình thư giãn toàn bộ các cạnh của đồ thị $|V| - 1$ lần, với $|V|$ là số đỉnh. Mỗi phép thư giãn kiểm tra xem việc đi qua một cạnh có thể cải thiện khoảng cách ngắn nhất hiện tại hay không. Cụ thể, với mỗi cạnh $(u, v) \in E$, nếu thỏa điều kiện:

$$d[v] > d[u] + w(u, v)$$

thì cập nhật $d[v] = d[u] + w(u, v)$. Sau $|V| - 1$ lần lặp, tất cả các đường đi ngắn nhất (với tối đa $|V| - 1$ cạnh) đều được xác định. Đây là kết quả quy nạp theo độ dài đường đi (Bellman, 1958).

Để phát hiện chu trình âm, thuật toán thực hiện thêm một lần thư giãn toàn bộ cạnh. Nếu trong bước này vẫn tồn tại cạnh (u, v) thỏa mãn điều kiện thư giãn, tức là $d[v] > d[u] + w(u, v)$, thì kết luận rằng đồ thị chứa chu trình âm.

Mặc dù có thể xử lý trọng số âm, thuật toán Bellman–Ford có độ phức tạp cao hơn so với Dijkstra, cụ thể là $O(|V| \cdot |E|)$, do phải lặp lại toàn bộ các cạnh nhiều lần. Tuy nhiên, ưu điểm của nó nằm ở tính tổng quát và khả năng áp dụng trong các trường hợp mà Dijkstra không khả thi.

2.4 Thuật toán Dijkstra

2.4.1 Tổng quan về Dijkstra

Thuật toán Dijkstra, lần đầu được trình bày trong một ghi chú ngắn năm 1959 bởi Edsger W. Dijkstra, giải quyết các vấn đề nền tảng trong lý thuyết đồ thị, cụ thể là việc xây dựng cây liên thông có tổng độ dài nhỏ nhất và tìm đường đi ngắn nhất trong mạng lưới có khoảng cách không âm.

Bài báo được công bố trong tạp chí *Numerische Mathematik*, trình bày một phương pháp cho bài toán đường đi ngắn nhất mà không cần liệt kê toàn bộ khả năng, nhấn mạnh hiệu quả tính toán thủ công vào thời điểm đó. Theo cách hiểu hiện đại, nó giải quyết bài toán đường đi ngắn nhất từ một nguồn (*Single Source Shortest Path – SSSP*) trên đồ thị có hướng có trọng số $G = (V, E)$ với hàm trọng số $w : E \to [0, \infty)$, tính toán $\delta(s, v)$ – trọng số nhỏ nhất của mọi đường đi từ đỉnh nguồn s đến mỗi đỉnh $v \in V$.

Yêu cầu trọng số không âm là điều kiện quan trọng, vì cạnh âm có thể phá vỡ tính đúng đắn của giải thuật tham lam, đòi hỏi sử dụng các thuật toán phức tạp hơn như Bellman-Ford.

2.4.2 Nguyên lý hoạt động của thuật toán Dijkstra

Cơ chế của thuật toán dựa trên giải thuật tham lam (*greedy*), tương tự như thuật toán Prim dùng cho cây khung nhỏ nhất, trong đó tập S các đỉnh đã được xác định sẽ dần mở rộng bằng cách thêm vào đỉnh gần nhất chưa được xử lý.

Bước khởi tạo thiết lập $d[s] = 0$ và $d[v] = \infty$ cho mọi $v \neq s$, với $\pi[v] = NIL$ làm đỉnh tiền nhiệm. Một hàng đợi ưu tiên nhỏ nhất (*min-priority queue*) Q chứa tất cả các đỉnh, được sắp xếp theo giá trị $d[v]$.

Trong mỗi vòng lặp, thuật toán chọn

$$u = \arg\min_{u \in Q} d[u],$$

thêm u vào S, và thư giãn (*relax*) từng cạnh (u, v) bằng cách kiểm tra điều kiện

$$d[v] > d[u] + w(u, v).$$

Nếu đúng, cập nhật $d[v]$ và $\pi[v]$, đồng thời thực hiện thao tác *decrease-key* trên Q. Quá trình này lặp lại $|V|$ lần, với tổng cộng $|E|$ lần thư giãn.

Tính đúng đắn dựa trên hai định lý cơ bản: bất đẳng thức tam giác đảm bảo không có đường đi tốt hơn đi qua đỉnh đã xét, và tính tối ưu của các đường con (*subpath optimality*) giữ cho các đoạn của đường đi ngắn nhất cũng là ngắn nhất. Chứng minh quy nạp cho thấy khi một đỉnh u được thêm vào S, thì $d[u] = \delta(s, u)$, vì mọi đường đi ngắn nhất đến u đều phải đi qua ranh giới giữa S và phần còn lại của đồ thị, và điều kiện trọng số không âm đảm bảo không thể giảm giá trị $d[u]$ về sau.

2.4.3 Cấu trúc dữ liệu chính

Trọng tâm của việc triển khai thuật toán là hàng đợi ưu tiên nhỏ nhất Q, quản lý các thao tác *extract-min* (thực hiện $|V|$ lần) và *decrease-key* (tối đa $|E|$ lần).

Cấu trúc đồ thị thường được biểu diễn bằng danh sách kề để truy cập nhanh các đỉnh lân cận. Mỗi đỉnh lưu hai thuộc tính: d (khoảng cách tạm thời) và π (tiền nhiệm) để tạo cây đường đi ngắn nhất G_π.

Các biến thể của hàng đợi gồm:

- Mảng (*array*): cho phép *extract-min* $O(V)$ và *decrease-key* $O(1)$, tổng thời gian $O(V^2)$.

- Heap nhị phân (*binary heap*): cho các thao tác $O(\log V)$, tổng thời gian $O((V + E) \log V)$.
- Heap Fibonacci: *extract-min* $O(\log V)$ và *decrease-key* $O(1)$ trung bình, đạt $O(E + V \log V)$.

Trong thực tế, heap nhị phân thường được ưa chuộng hơn nhờ hằng số nhỏ và hiệu suất cao.

2.4.4 Độ phức tạp thời gian

Phân tích độ phức tạp thời gian trong các tài liệu lý thuyết nhấn mạnh sự phụ thuộc vào cách cài đặt. Với đồ thị dày đặc ($E \approx V^2$), $O(V^2)$ là hợp lý; còn với đồ thị thưa ($E < V^2$), các yếu tố logarit chiếm ưu thế.

Các nghiên cứu về hiệu năng kỳ vọng với độ dài cạnh ngẫu nhiên chỉ ra rằng số lần *decrease-key* kỳ vọng là $O(n \log(1 + m/n))$ với xác suất cao, tinh chỉnh thời gian thực tế của heap nhị phân xuống còn $O(m + n \log n \log(1 + m/n))$. Điều này giải thích tại sao heap Fibonacci hiếm khi vượt trội trong thực tế, vì số lần *decrease-key* thường nhỏ hơn tuyến tính so với m và các hằng số trong cài đặt đơn giản hơn.

Các nghiên cứu thực nghiệm về biến thể *Generic Dijkstra* – dùng cho mạng có giới hạn dung lượng – cho thấy độ phức tạp theo số đỉnh tăng bậc hai ($R^2 \approx 0{,}99$) và theo số cạnh tăng theo logarit, đạt cực đại ở mức sử dụng khoảng 0,25. So với mô hình *Filtered Graphs* ($O(SV \log V)$), *Generic Dijkstra* nhanh hơn khoảng 2–4 lần trong phần lớn trường hợp, khẳng định độ phức tạp thực tế thấp hơn.

2.4.5 Ứng dụng thuật toán Dijkstra

1. **Định tuyến mạng máy tính.**
   Thuật toán Dijkstra được sử dụng rộng rãi trong các giao thức định tuyến mạng như OSPF (*Open Shortest Path First*) để tìm đường đi ngắn nhất giữa hai thiết bị mạng. Điều này giúp tối ưu hóa việc truyền tải dữ liệu, giảm thiểu độ trễ và tăng cường hiệu suất của mạng. Trong các hệ thống lớn như mạng Internet, việc sử dụng thuật toán này đảm bảo dữ liệu luôn đi qua các tuyến đường nhanh và hiệu quả nhất.

2. **Ứng dụng bản đồ và hệ thống định vị.**
   Các ứng dụng bản đồ và định vị hiện đại như Google Maps hay Apple Maps đều

sử dụng thuật toán Dijkstra (hoặc các biến thể của nó) để tính toán lộ trình tối ưu giữa hai điểm. Thuật toán cho phép xác định tuyến đường ngắn nhất hoặc nhanh nhất tùy theo điều kiện giao thông thực tế, đồng thời giúp người dùng tránh các khu vực ùn tắc, công trình đang thi công hoặc những tuyến đường không khả dụng. Nhờ đó, trải nghiệm điều hướng của người dùng trở nên hiệu quả, thuận tiện và đáng tin cậy hơn.

3. **Lập kế hoạch sản xuất và quản lý chuỗi cung ứng.**
   Trong lĩnh vực công nghiệp và logistics, Dijkstra được ứng dụng để tối ưu hóa quy trình vận chuyển và lập lịch sản xuất. Các doanh nghiệp có thể sử dụng thuật toán này để tìm tuyến đường ngắn nhất khi vận chuyển nguyên liệu giữa các kho bãi, nhà máy hoặc điểm phân phối. Nhờ vậy, chi phí vận hành được cắt giảm, thời gian giao hàng được rút ngắn, đồng thời năng suất chuỗi cung ứng được cải thiện đáng kể. Ngoài ra, trong quản lý sản xuất, thuật toán còn giúp xác định trình tự di chuyển hợp lý giữa các công đoạn, giảm thiểu thời gian chờ và tăng hiệu quả sử dụng nguồn lực.

4. **Tối ưu hóa vận chuyển và logistics.**
   Trong ngành logistics và giao nhận hàng hóa, Dijkstra giúp xác định các tuyến đường ngắn nhất cho việc giao hàng hoặc thu gom sản phẩm. Điều này giúp các doanh nghiệp giảm chi phí vận hành, tiết kiệm nhiên liệu, và tối đa hóa hiệu quả vận chuyển.

Nhờ tính linh hoạt và hiệu quả cao, thuật toán tìm đường đi ngắn nhất Dijkstra đã được ứng dụng rộng rãi trong nhiều lĩnh vực, điển hình như xây dựng tuyến đường giao thông (Hongbo Wu et al., 2019), logistics và phân phối hàng hóa (Boli Huang, 2020).

Tại Việt Nam, thuật toán Dijkstra cũng được ứng dụng trong các bài nghiên cứu như tối ưu quãng đường mobile robot (Nguyễn Ngọc Kiên và cộng sự, 2023) hay hệ thống robot thông minh vận chuyển hàng trong kho (Bùi Quang Vương và cộng sự, 2017).

Chương 3

Thiết kế và cài đặt chương trình

3.1 Tổng quan thiết kế

Chương trình nhóm xây dựng có chức năng tạo đồ thị từ tập tin csv đầu vào, thực hiện thuật toán Dijkstra. Ngoài ra còn mô phỏng và so sánh hiệu năng của Dijkstra so với các thuật toán tìm đường đi khác, gồm A*, BFS và DFS.

Hệ thống được thiết kế theo hướng mô-đun, gồm ba phần chính: sinh dữ liệu đồ thị, biểu diễn và trực quan hoá cấu trúc đồ thị, cùng với phần triển khai các thuật toán tìm đường. Mục tiêu là giúp chương trình dễ mở rộng, dễ kiểm thử và minh bạch về mặt thuật toán.

Quy trình thực thi bắt đầu từ việc sinh ngẫu nhiên đồ thị vô hướng có trọng số không âm, sau đó khởi tạo đối tượng đồ thị từ tệp CSV, lần lượt chạy bốn thuật toán và cuối cùng trực quan hoá kết quả bằng biểu đồ so sánh.

3.2 Thiết kế mô hình dữ liệu

Dữ liệu đầu vào là tập các cạnh, mỗi cạnh chứa thông tin: đỉnh đầu (`v_from`), đỉnh cuối (`v_to`) và trọng số (`weight`). Dữ liệu được sinh ngẫu nhiên và lưu vào tệp CSV với định dạng:

```
v_from, v_to, weight
A, B, 10
A, C, 5
B, D, 8
...
```

Sau đó, hàm `import_graph_from_edges_csv()` đọc dữ liệu và xây dựng đối tượng đồ thị. Dưới đây là đoạn mã minh họa:

```
def import_graph_from_edges_csv(csv_path):
    edges_list = []
    with open(csv_path, "r", encoding="utf-8") as f:
        reader = csv.reader(f)
        header = next(reader)
        for row in reader:
            v_from, v_to, weight = row[0], row[1], float(row[2])
            edges_list.append((v_from, v_to, weight))
    g = Graph(edges_list)
    return g, edges_list
```

3.3 Cấu trúc lớp Graph

Lớp `Graph` đóng vai trò cốt lõi trong chương trình, chịu trách nhiệm biểu diễn đồ thị vô hướng có trọng số bằng danh sách kề, đồng thời hỗ trợ vẽ sơ đồ trực quan các đỉnh và cạnh. Mỗi đỉnh được lưu dưới dạng khóa trong từ điển `adj`, ánh xạ tới danh sách các đỉnh kề kèm trọng số.

Hàm khởi tạo nhận danh sách cạnh, sau đó tự động gọi hàm `_build_graph()` để thêm cạnh và gán tọa độ hiển thị cho từng đỉnh. Các đỉnh được bố trí theo kiểu "neural net", trong đó đỉnh bắt đầu (Start) nằm ở cột đầu tiên, đỉnh đích (Goal) nằm ở cột cuối, các đỉnh trung gian được phân bố đều giữa các lớp.

Ví dụ đoạn khởi tạo lớp:

```
def __init__(self, edges):
    self.adj = defaultdict(list)
    self.positions = {}
    self.edge_list = list(edges)
    self._build_graph(edges)
```

Phương thức `add_edge()` đảm bảo đồ thị vô hướng bằng cách thêm cả hai chiều của cạnh:

```
def add_edge(self, v_from, v_to, weight):
    self.adj[v_from].append((v_to, weight))
    self.adj[v_to].append((v_from, weight))
```

3.4 Hàm `dijkstra`

3.4.1 Mục tiêu

Hàm `dijkstra()` được triển khai nhằm tìm đường đi ngắn nhất (với tổng trọng số nhỏ nhất) từ một đỉnh xuất phát đến một đỉnh đích trên đồ thị có trọng số không âm. Đây là một trong những thuật toán nền tảng trong lý thuyết đồ thị, đảm bảo tính tối ưu về chi phí khi không tồn tại cạnh âm.

Các tham số đầu vào bao gồm:

- **graph (Graph)**: Đối tượng đồ thị có trọng số không âm.
- **start (Hashable)**: Đỉnh xuất phát.
- **goal (Hashable)**: Đỉnh đích cần tìm đường đi.

Hàm trả về một cấu trúc `dictionary` chứa toàn bộ kết quả tìm đường như sau:

```
{
  'name'    : 'Dijkstra',     # Tên thuật toán
  'path'    : List[node],     # Danh sách đỉnh trên đường đi tối ưu
  'cost'    : float,          # Tổng chi phí đường đi tối ưu
  'expanded': int,            # Số đỉnh được mở rộng (pop khỏi heap)
  'time_ms' : float           # Thời gian chạy (milliseconds)
}
```

3.4.2 Thuật toán Dijkstra

```
def dijkstra(graph, start, goal):
    frontier = [(0, start)]
    dist = {start: 0}
    prev = {start: None}
    expansions = 0
    t0 = time.time()

    while frontier:
        cost, node = heapq.heappop(frontier)
        if cost > dist.get(node, float('inf')):
            continue
```

```
        expansions += 1
        if node == goal:
            break
        for neigh, w in graph.neighbors(node):
            new_cost = cost + w
            if new_cost < dist.get(neigh, float('inf')):
                dist[neigh] = new_cost
                prev[neigh] = node
                heapq.heappush(frontier, (new_cost, neigh))

    if dist.get(goal, float('inf')) == float('inf'):
        return {'name': 'Dijkstra',
                'path': [],
                'cost': float('inf'),
                'expanded': expansions,
                'time_ms': (time.time() - t0) * 1000}

    path = []
    node = goal
    while node is not None:
        path.append(node)
        node = prev.get(node)
    path.reverse()

    return {'name': 'Dijkstra',
            'path': path,
            'cost': dist.get(goal, float('inf')),
            'expanded': expansions,
            'time_ms': (time.time() - t0) * 1000}
```

Thuật toán hoạt động theo nguyên tắc tham lam (*greedy*), sử dụng hàng đợi ưu tiên (*min-heap*) để luôn chọn đỉnh có chi phí nhỏ nhất tại mỗi bước mở rộng.

Bước 1: Khởi tạo

Trong giai đoạn khởi tạo, thuật toán tạo một heap chứa cặp $(cost, node)$, với chi phí khởi đầu là 0 cho đỉnh xuất phát. Bảng `dist` lưu chi phí ngắn nhất từ đỉnh xuất phát đến từng đỉnh khác; ban đầu, tất cả giá trị đều là vô cực trừ đỉnh xuất phát. Bảng

`prev` lưu thông tin truy vết, chỉ ra đỉnh cha của mỗi đỉnh trong đường đi tối ưu. Biến `expansions` đếm số đỉnh đã được lấy ra khỏi heap, đồng thời thời điểm bắt đầu được ghi lại để tính thời gian thực thi.

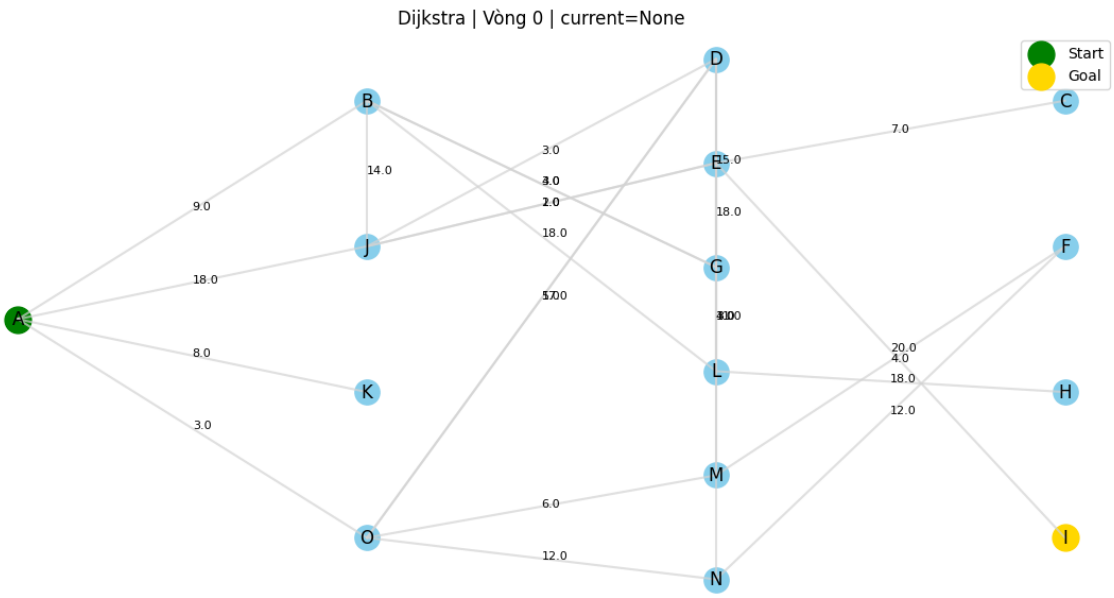
Bước 2: Vòng lặp chính

Sau khi khởi tạo, thuật toán bắt đầu đi vào vòng lặp chính. Ở mỗi vòng, đỉnh có chi phí nhỏ nhất trong hàng đợi ưu tiên (heap) sẽ được lấy ra để xử lý — ban đầu là đỉnh **A**. Từ đỉnh này, thuật toán lần lượt duyệt qua tất cả các đỉnh láng giềng $(neigh, w)$ và tính toán chi phí tích lũy mới:

$$new\_cost = dist[node] + w$$

Nếu chi phí mới `new_cost` nhỏ hơn giá trị đang lưu trong bảng `dist`, thuật toán cập nhật lại chi phí ngắn nhất (Dist) và ghi nhận đỉnh cha (Prev) tương ứng. Đồng thời, cặp $(new\_cost, neigh)$ được đưa vào heap để xét trong các vòng kế tiếp. Quá trình này đảm bảo rằng mọi đỉnh trong đồ thị đều được mở rộng theo thứ tự tăng dần của chi phí.

Hình 3.1 dưới đây minh họa trạng thái khởi tạo của thuật toán tại Vòng 0, khi chỉ có duy nhất đỉnh xuất phát **A** được biết đến với chi phí bằng 0. Tất cả các đỉnh khác đều chưa được khám phá.



Hình 3.1: Trạng thái khởi tạo của thuật toán Dijkstra (Vòng 0)

Ở thời điểm này, bảng Dist/Prev (Bảng 3.1) chỉ chứa thông tin của đỉnh **A**: khoảng cách nhỏ nhất (Dist) bằng 0 và không có đỉnh cha (Prev = None). Hàng đợi ưu tiên (heap) — thể hiện trong Bảng 3.2 — chỉ gồm một phần tử duy nhất là cặp $(0, A)$.

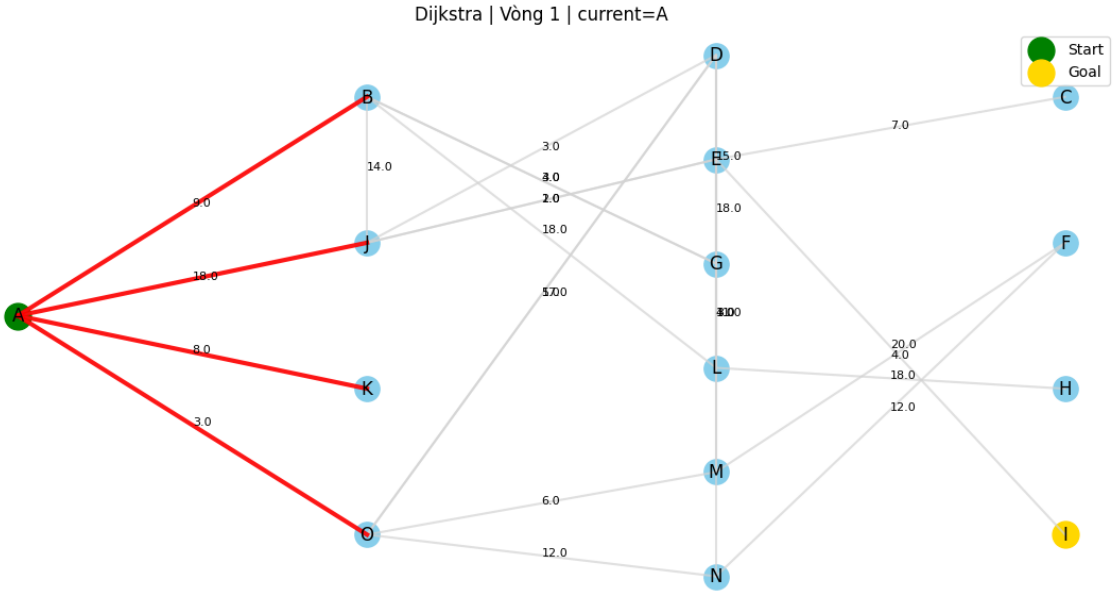
Node	Dist	Prev
A	0	None

Bảng 3.1: Bảng Dist/Prev tại thời điểm khởi tạo (Vòng 0).

Cost	Node
0	A

Bảng 3.2: Cấu trúc hàng đợi ưu tiên (Frontier heap) tại Vòng 0.

Khi mở rộng các đỉnh láng giềng của **A**, thuật toán tính được các chi phí mới và cập nhật bảng Dist/Prev tương ứng. Kết quả sau khi mở rộng **A** được thể hiện ở Bảng 3.3. Tại đây, các đỉnh **B**, **J**, **K**, và **O** đã được khám phá với chi phí lần lượt là 9, 18, 8 và 3. Trong đó, đỉnh **O** có chi phí nhỏ nhất nên sẽ được mở rộng tiếp theo ở vòng kế tiếp.



Hình 3.2: Trạng thái thuật toán Dijkstra tại vòng lặp đầu tiên khi mở rộng đỉnh

Node	Dist	Prev
A	0	None
B	9	A
J	18	A
K	8	A
O	3	A

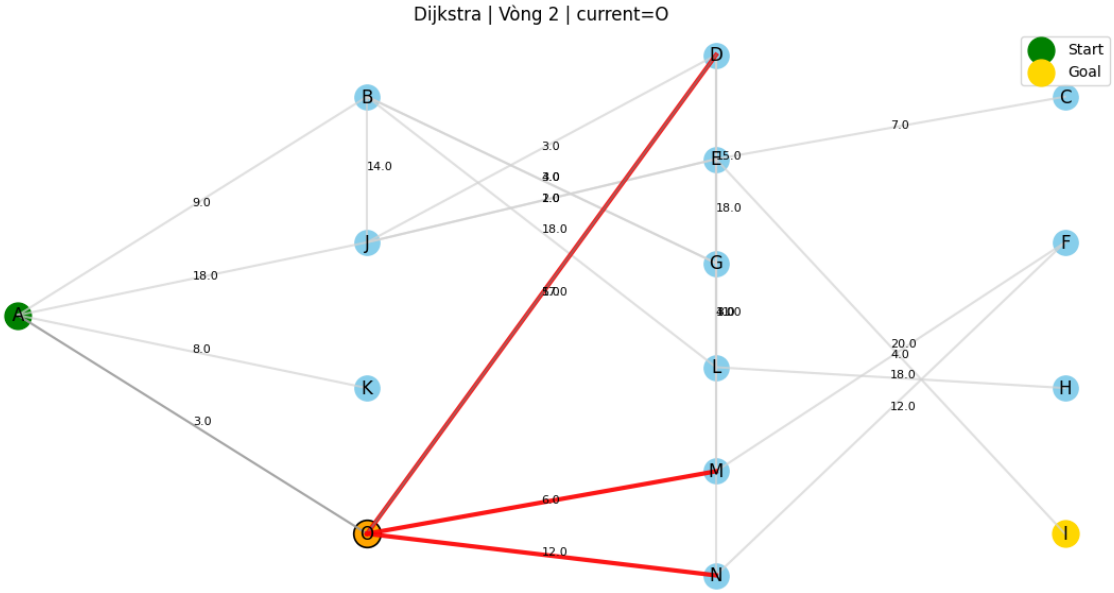
Bảng 3.3: Bảng Dist/Prev sau khi mở rộng đỉnh A (Vòng 1).

Tại thời điểm này, heap đã được cập nhật với các cặp $(new\_cost, node)$ tương ứng, sắp xếp theo thứ tự tăng dần của chi phí (Bảng 3.4). Đỉnh **O** với chi phí nhỏ nhất (3) sẽ được lấy ra để mở rộng ở vòng tiếp theo.

Cost	Node
3	O
8	K
9	B
18	J

Bảng 3.4: Trạng thái heap sau khi mở rộng đỉnh A (Vòng 1).

Sau khi mở rộng **O**, các đỉnh mới được cập nhật là **D**, **M** và **N**, với chi phí ngắn nhất lần lượt là 8, 9 và 15. Bảng 3.5 dưới đây cho thấy trạng thái của thuật toán tại Vòng 2, khi heap tiếp tục được mở rộng để tìm ra đường đi ngắn nhất đến các đỉnh còn lại.



Hình 3.3: Trạng thái của Dijkstra ở vòng lặp thứ hai

Node	Dist	Prev
D	8	O
K	8	A
B	9	A
M	9	O
N	15	O
J	18	A

Bảng 3.5: Bảng Dist/Prev sau khi mở rộng đỉnh O (Vòng 2).

Thuật toán sẽ tiếp tục quá trình này, lần lượt mở rộng các đỉnh có chi phí nhỏ nhất trong heap, cho đến khi tìm thấy đỉnh đích hoặc khi toàn bộ đồ thị đã được khám phá (heap rỗng).

Bước 3: Lần vết đường đi

Sau khi vòng lặp kết thúc, thuật toán xây dựng lại đường đi bằng cách lần ngược từ đỉnh đích (`goal`) về đỉnh xuất phát (`start`) thông qua bảng `prev`.

Tại vòng lặp cuối cùng (vòng 14), kết quả đạt được như sau:

Node	Dist	Prev
A	0	None
B	9.0	A
C	19.0	E
D	8.0	O
E	12.0	M
F	13.0	M
G	12.0	B
H	41.0	L
I	32.0	E
J	11.0	D
K	8.0	A
L	23.0	G
M	9.0	O
N	12.0	D
O	3.0	A

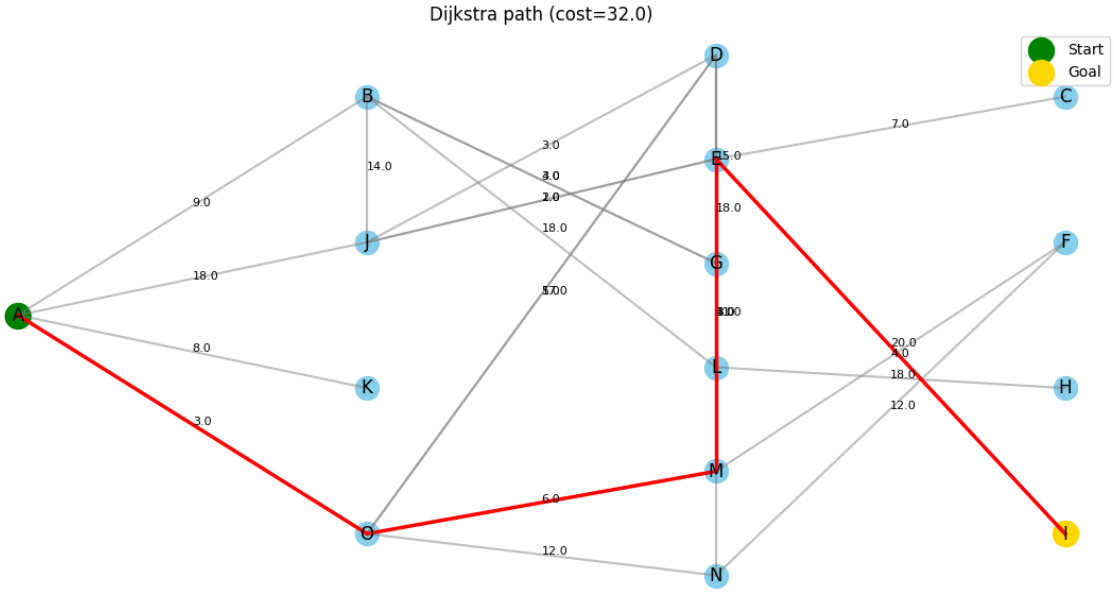
Bảng 3.6: Bảng Dist/Prev cuối cùng.

Đỉnh	Khoảng cách (Dist)	Truy vết (Prev)
A	0	None
O	3	$\leftarrow$ A
M	9	$\leftarrow$ O
E	12	$\leftarrow$ M
I	32	$\leftarrow$ E

Như vậy, đường đi ngắn nhất từ đỉnh xuất phát A đến đỉnh đích I là:

$$A \to O \to M \to E \to I$$

với tổng chi phí là 32.



Hình 3.4: Kết quả cuối cùng của thuật toán Dijkstra

3.4.3 Yêu cầu của hàm

Đối tượng `graph` phải triển khai phương thức `neighbors()` để trả về danh sách các đỉnh kề cùng trọng số của các cạnh tương ứng. Tất cả các trọng số trong đồ thị phải không âm, vì sự tồn tại của cạnh âm sẽ khiến thuật toán Dijkstra mất tính đúng đắn.

3.5 Cài đặt các thuật toán tìm đường để so sánh với Dijkstra

3.5.1 Thuật toán A* (A-star)

Thuật toán A* là một cải tiến của Dijkstra, bổ sung thêm một hàm heuristic để ước lượng khoảng cách từ một đỉnh đến đỉnh đích. Hàm này giúp thuật toán ưu tiên mở rộng các đỉnh có khả năng đi tới đích nhanh nhất, giảm thiểu số đỉnh cần phải kiểm tra.

A* sử dụng công thức tổng chi phí $f(n) = g(n) + h(n)$, trong đó:

- $g(n)$: Chi phí từ đỉnh xuất phát đến đỉnh n.
- $h(n)$: Ước lượng chi phí từ đỉnh n đến đích (sử dụng hàm heuristic).

Hàm A* được cài đặt dưới đây:

```python
def astar(graph, start, goal):
    frontier = [(0, start)]  # Min-heap lưu (f-score, node)
    g = {start: 0}  # Chi phí từ start đến các node
    f = {start: heuristic(start, goal, graph.positions)}  # f = g + h
    prev = {start: None}

    while frontier:
        _, node = heapq.heappop(frontier)
        if node == goal:
            break
        for neigh, w in graph.neighbors(node):
            new_cost = g[node] + w
            if new_cost < g.get(neigh, float('inf')):
                g[neigh] = new_cost
                f[neigh] = new_cost + heuristic(neigh, goal,
                                                graph.positions)
                prev[neigh] = node
                heapq.heappush(frontier, (f[neigh], neigh))

    # Xây dựng đường đi từ goal về start
    path = []
    node = goal
    while node is not None:
        path.append(node)
        node = prev.get(node)
    path.reverse()

    return {'name': 'A*', 'path': path, 'cost': g.get(goal, float('inf')),
            'expanded': len(f), 'time_ms': (time.time() - t0) * 1000}
```

A* có ưu điểm là thường mở rộng ít đỉnh hơn Dijkstra nhờ vào heuristic, giúp tăng tốc quá trình tìm kiếm.

3.5.2 Thuật toán BFS (Breadth-First Search)

BFS là thuật toán tìm kiếm theo chiều rộng, duyệt qua tất cả các đỉnh ở độ sâu hiện tại trước khi chuyển sang độ sâu tiếp theo. Đối với đồ thị không trọng số, BFS sẽ tìm

được đường đi ngắn nhất giữa hai đỉnh, nhưng không thể tối ưu chi phí nếu đồ thị có trọng số.

Hàm BFS được cài đặt như sau:

```python
def bfs(graph, start, goal):
    queue = deque([start])
    prev = {start: None}
    visited = {start}
    expansions = 0

    while queue:
        node = queue.popleft()
        expansions += 1
        if node == goal:
            break
        for neigh, _ in graph.neighbors(node):
            if neigh not in visited:
                visited.add(neigh)
                prev[neigh] = node
                queue.append(neigh)

    # Xây dựng đường đi từ goal về start
    path = []
    node = goal
    while node is not None:
        path.append(node)
        node = prev.get(node)
    path.reverse()

    return {'name': 'BFS', 'path': path, 'cost': len(path) - 1,
            'expanded': expansions, 'time_ms': (time.time() - t0) * 1000}
```

Mặc dù BFS có thể tìm được đường đi ngắn nhất trong đồ thị không trọng số, nhưng không tối ưu chi phí trong đồ thị có trọng số.

3.5.3 Thuật toán DFS (Depth-First Search)

DFS là thuật toán tìm kiếm theo chiều sâu, duyệt xuống các đỉnh sâu nhất trong đồ thị trước khi quay lại. Thuật toán này có thể tìm ra một đường đi từ đỉnh xuất phát đến đỉnh đích, nhưng không đảm bảo tính tối ưu về chi phí hay số bước.

Hàm DFS được cài đặt như sau:

```python
def dfs(graph, start, goal):
    stack = [start]
    prev = {start: None}
    visited = {start}
    expansions = 0

    while stack:
        node = stack.pop()
        expansions += 1
        if node == goal:
            break
        for neigh, _ in graph.neighbors(node):
            if neigh not in visited:
                visited.add(neigh)
                prev[neigh] = node
                stack.append(neigh)

    # Xây dựng đường đi từ goal về start
    path = []
    node = goal
    while node is not None:
        path.append(node)
        node = prev.get(node)
    path.reverse()

    return {'name': 'DFS', 'path': path, 'cost': len(path) - 1,
            'expanded': expansions, 'time_ms': (time.time() - t0) * 1000}
```

DFS có thể rất nhanh trong một số trường hợp, nhưng vì không tối ưu chi phí hay độ dài đường đi, nên nó không phải là lựa chọn tốt nhất cho đồ thị có trọng số.

3.6 Chương trình chính và trực quan hóa kết quả

Phần cuối chương trình khởi tạo đồ thị từ tệp `edges.csv`, chọn đỉnh bắt đầu là "A" và đỉnh kết thúc là đỉnh cuối trong danh sách. Bốn thuật toán được gọi lần lượt để tìm đường đi, sau đó kết quả được in ra và trực quan hóa qua biểu đồ so sánh.

```
# khởi tạo đồ thị từ tệp edges.csv
g, edges_list_in = import_graph_from_edges_csv("edges.csv")

nodes = list(g.adj.keys())
start = "A"
goal = nodes[-1]

results=[]
for algo in [dijkstra,astar,bfs,dfs]:
    res=algo(g,start,goal)
    results.append(res)
    print(f"{res['name']:8s} |
            cost={res['cost']:>5} |
            expanded={res['expanded']:>4} |
            time={res['time_ms']:.2f} ms |
            path={res['path']}")

# Vẽ đường đi của Dijkstra
best=results[0]
g.plot(best['path'],start,goal,title=f"Dijkstra path (cost={best['cost']})")

# Chuẩn bị dữ liệu cho các biểu đồ so sánh
labels=[r['name'] for r in results]
costs=[r['cost'] for r in results]
expanded=[r['expanded'] for r in results]
times=[r['time_ms'] for r in results]

plt.figure();
plt.bar(labels,costs); plt.title("So sánh tổng chi phí"); plt.show()

plt.figure();
plt.bar(labels,expanded); plt.title("So sánh số node mở rộng"); plt.show()
```

```
33
34 plt.figure();
35 plt.bar(labels,times); plt.title("So sánh thời gian (ms)");
36
37 plt.show()
```

Chương 4

Đánh giá và so sánh

4.1 Thiết kế thí nghiệm

Để đánh giá hiệu quả của các thuật toán tìm đường, bao gồm Dijkstra, A*, BFS và DFS, nhóm tiến hành ba thí nghiệm riêng biệt trên các đồ thị ngẫu nhiên có số lượng đỉnh lần lượt là 15, 20 và 24. Mỗi đồ thị được sinh ngẫu nhiên với số cạnh và trọng số khác nhau, đảm bảo tính đa dạng của cấu trúc để quan sát hành vi của các thuật toán trong nhiều tình huống.

Các đồ thị được tạo từ tệp `.csv` sinh ngẫu nhiên với các tham số như sau:

- Số đỉnh (NUM_NODES): 15, 20 và 24 đỉnh.
- Số cạnh (EDGE_COUNT): Từ 20 đến 30 cạnh, được chọn ngẫu nhiên sao cho đồ thị liên thông.
- Trọng số cạnh (weight): Sinh ngẫu nhiên trong khoảng từ 1 đến 20, đại diện cho chi phí di chuyển giữa hai đỉnh.

Mã sinh dữ liệu sử dụng Python được thể hiện như sau:

```
NUMS_NODE = 15
NODES = [chr(ord('A') + i) for i in range(NUMS_NODE)]
EDGE_COUNT = random.randint(20, 25)
edges = set()

while len(edges) < EDGE_COUNT:
    v_from, v_to = random.sample(NODES, 2)
    if (v_from, v_to) not in edges and (v_to, v_from) not in edges:
```

```
        weight = random.randint(1, 20)
        edges.add((v_from, v_to, weight))

with open("edges.csv", "w", encoding="utf-8", newline="") as f:
    writer = csv.writer(f, delimiter=",")
    writer.writerow(["v_from", "v_to", "weight"])
    for v_from, v_to, weight in edges:
        writer.writerow([v_from, v_to, weight])
```

Mỗi thuật toán được thực thi để tìm đường đi ngắn nhất từ một đỉnh xuất phát (`start`) đến một đỉnh đích (`goal`). Nhóm đo ba chỉ số chính:

- Chi phí tối ưu (Optimal Cost): Tổng trọng số của đường đi ngắn nhất mà thuật toán tìm được.
- Số đỉnh mở rộng (Expanded Nodes): Số lượng đỉnh đã được duyệt trong quá trình tìm kiếm, thể hiện mức độ hiệu quả của thuật toán.
- Thời gian thực thi (Execution Time): Thời gian (ms) mà thuật toán cần để hoàn tất tìm kiếm.

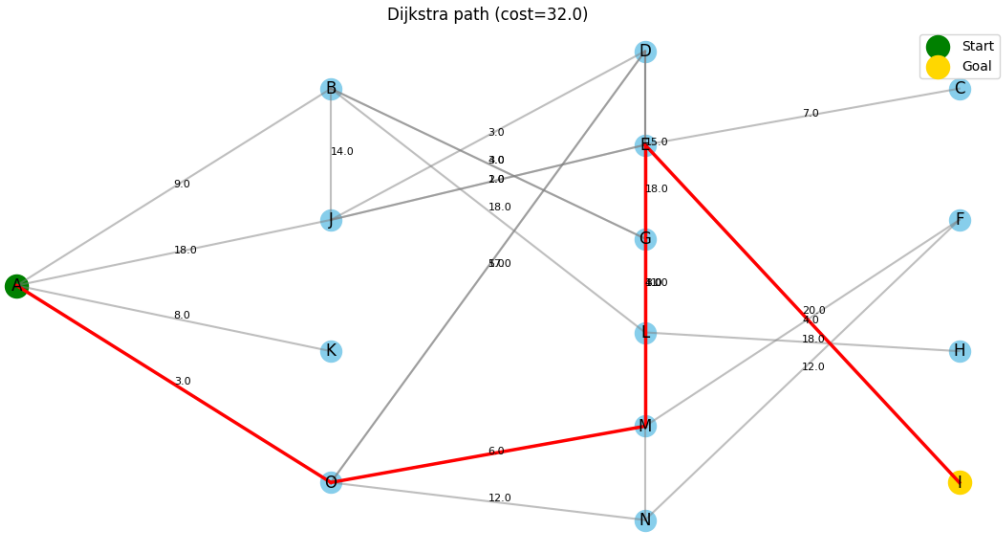
Thí nghiệm được thực hiện trong môi trường Google Colab (CPU), với cùng điều kiện phần cứng để đảm bảo tính khách quan của kết quả.

4.2 Kết quả và nhận xét

Kết quả của ba thí nghiệm với số lượng đỉnh tăng dần được trình bày trong các bảng và hình ảnh đồ thị dưới đây.

Bảng 4.1: Kết quả thí nghiệm 1 – 15 đỉnh

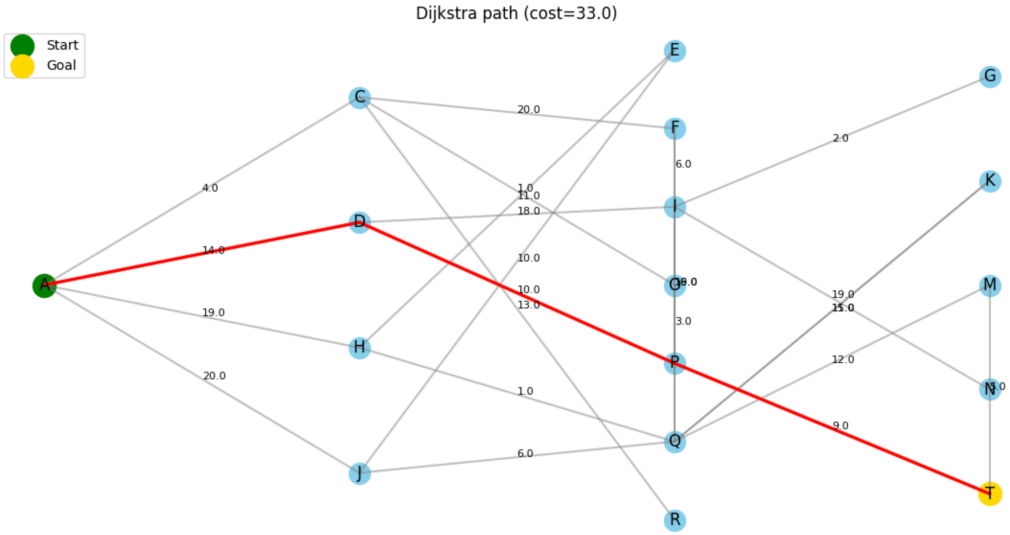
Thuật toán	**Chi phí**	**Số đỉnh mở rộng**	**Thời gian (ms)**
Dijkstra	19	12	0.08
A*	19	14	0.07
BFS	28	14	0.03
DFS	19	11	0.02



Hình 4.1: Kết quả thí nghiệm 1 - 15 đỉnh

Bảng 4.2: Kết quả thí nghiệm 2 – 20 đỉnh

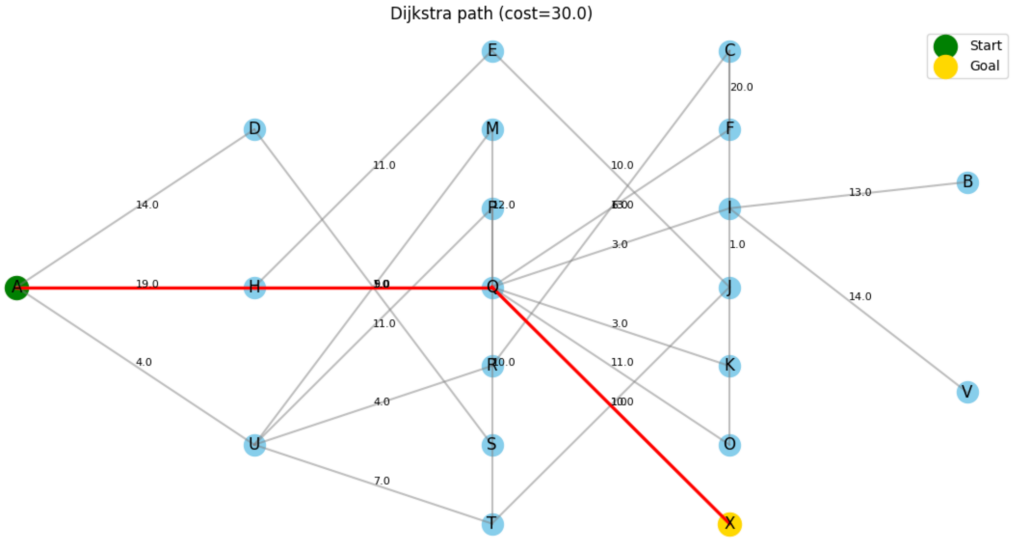
Thuật toán	**Chi phí**	**Số đỉnh mở rộng**	**Thời gian (ms)**
Dijkstra	33	16	0.06
A*	33	16	0.06
BFS	33	17	0.02
DFS	33	4	0.01



Hình 4.2: Kết quả thí nghiệm 2 - 20 đỉnh

Bảng 4.3: Kết quả thí nghiệm 3 – 24 đỉnh

Thuật toán	Chi phí	Số đỉnh mở rộng	Thời gian (ms)
Dijkstra	30	18	0.07
A*	30	19	0.11
BFS	30	15	0.05
DFS	43	10	0.02



Hình 4.3: Kết quả thí nghiệm 3 - 24 đỉnh

4.2.1 Phân tích kết quả

Dijkstra luôn đảm bảo tìm được đường đi có chi phí tối ưu trong cả ba thí nghiệm. Tuy nhiên, số lượng đỉnh mở rộng tăng dần khi kích thước đồ thị lớn hơn, cho thấy chi phí tính toán của thuật toán tỷ lệ thuận với số lượng đỉnh và cạnh.

A* cho kết quả chi phí giống hệt Dijkstra nhưng có thời gian chạy ngắn hơn hoặc tương đương, nhờ vào việc sử dụng hàm heuristic để định hướng quá trình tìm kiếm. Trong thí nghiệm thứ ba (24 đỉnh), A* vẫn duy trì độ chính xác nhưng mất thêm thời gian do heuristic phải được tính cho nhiều trạng thái hơn.

BFS chỉ tối ưu về số lượng bước đi chứ không tối ưu về chi phí trong đồ thị có trọng số. Ở các thí nghiệm 1 và 3, chi phí của BFS cao hơn đáng kể so với Dijkstra và A*, mặc dù thời gian thực thi nhanh nhất do không cần xử lý trọng số.

DFS có thời gian chạy ngắn nhất và số đỉnh mở rộng ít, nhưng kết quả không ổn định. Trong một số trường hợp (thí nghiệm 3), chi phí tìm được cao hơn đáng kể (43 so

với 30). Điều này phản ánh đặc trưng của DFS: chỉ tìm một đường đi khả dĩ, không đảm bảo tối ưu toàn cục.

4.2.2 Nhận xét tổng quan

Từ ba thí nghiệm, có thể nhận thấy rằng:

- Dijkstra và A* cho kết quả tối ưu và ổn định nhất về chi phí.
- A* tỏ ra hiệu quả hơn trong các đồ thị lớn nhờ sử dụng heuristic để giảm số lượng đỉnh mở rộng.
- BFS phù hợp cho đồ thị không trọng số hoặc khi yêu cầu đường đi ngắn nhất theo số bước.
- DFS nhanh nhất nhưng không đảm bảo tối ưu, chỉ thích hợp khi yêu cầu kiểm tra tính liên thông hoặc thăm dò nhanh.

Như vậy, trong bài toán tìm đường trên đồ thị có trọng số, **A*** vẫn là lựa chọn cân bằng nhất giữa tốc độ và độ chính xác khi heuristic được thiết kế phù hợp.

4.2.3 Kết luận

Dựa trên kết quả so sánh, thuật toán A* là lựa chọn tối ưu nhất khi cần tối ưu chi phí và thời gian thực thi nhờ vào sự kết hợp giữa Dijkstra và heuristic. Dijkstra, mặc dù cho kết quả chính xác về chi phí, nhưng lại tốn nhiều thời gian và tài nguyên hơn so với A*. Trong khi đó, BFS và DFS có thể là lựa chọn hợp lý trong những tình huống không yêu cầu tối ưu chi phí, đặc biệt khi cần kết quả nhanh chóng hoặc đơn giản trong các đồ thị không trọng số. Tuy nhiên, đối với đồ thị có trọng số và yêu cầu hiệu suất tốt, A* vẫn là sự lựa chọn ưu tiên.

Chương 5

Tổng kết đề tài

5.1 Kết quả của thuật toán Dijkstra

Thuật toán Dijkstra đóng vai trò nền tảng trong việc giải quyết bài toán tìm đường đi ngắn nhất trên đồ thị có trọng số không âm. Cơ chế hoạt động của thuật toán dựa trên nguyên tắc tham lam, lựa chọn dần các đỉnh có chi phí nhỏ nhất tính từ đỉnh nguồn, sau đó mở rộng sang các đỉnh kề chưa được xử lý. Quá trình này đảm bảo rằng mỗi đỉnh khi được chọn đều mang giá trị chi phí tối ưu tại thời điểm đó.

Với tính chính xác, ổn định và khả năng mở rộng tốt, thuật toán Dijkstra được xem là một trong những phương pháp cốt lõi của lĩnh vực tối ưu hóa đường đi và là cơ sở cho nhiều thuật toán tiên tiến khác, chẳng hạn như thuật toán A* (A-star). Kết quả thực nghiệm trong đề án cho thấy, khi so sánh với các thuật toán khác như A*, BFS và DFS, Dijkstra luôn tìm được đường đi có tổng chi phí nhỏ nhất và đạt hiệu năng tốt nhờ việc sử dụng cấu trúc hàng đợi ưu tiên (min-heap).

Thông qua mô phỏng trực quan, người dùng có thể quan sát tiến trình tìm đường, sự khác biệt giữa các chiến lược tìm kiếm, và hiểu rõ hơn về ưu thế của Dijkstra trong việc tối ưu hóa chi phí di chuyển trên đồ thị.

5.2 Những đóng góp của đề án

Về phương diện lý thuyết, đề án đã trình bày chi tiết nguyên lý hoạt động của thuật toán Dijkstra, bao gồm cơ chế lựa chọn và mở rộng các đỉnh có chi phí tạm thời nhỏ nhất, cùng với cách thức cập nhật bảng khoảng cách và bảng truy vết để đảm bảo tìm được đường đi tối ưu. Việc đối chiếu giữa Dijkstra và các thuật toán tìm kiếm khác cũng giúp làm rõ mối quan hệ giữa tìm kiếm theo chi phí (cost-based search) và tìm

kiếm theo cấu trúc đồ thị (structure-based search), qua đó củng cố hiểu biết về bản chất của các chiến lược tìm kiếm trong đồ thị.

Về mặt ứng dụng, đề án đã chứng minh khả năng triển khai thực tế của thuật toán Dijkstra trong các bài toán tối ưu hóa chi phí, tiêu biểu như tìm đường trong hệ thống bản đồ, định tuyến mạng, hoặc tối ưu hóa vận tải logistics. Kết quả mô phỏng trực quan không chỉ minh họa rõ ràng tiến trình của thuật toán mà còn thể hiện giá trị thực tiễn của nó, qua đó khẳng định vai trò quan trọng của Dijkstra trong khoa học máy tính và kỹ thuật phần mềm.

5.3 Hướng phát triển

Đề án có thể được mở rộng trong các hướng nghiên cứu tiếp theo như sau:

1. Kết hợp thuật toán Dijkstra với các hàm heuristic thực tế nhằm cải thiện hiệu năng và mở rộng sang thuật toán A* trong các ứng dụng yêu cầu tìm kiếm nhanh hơn.

2. Mở rộng mô phỏng cho các đồ thị có hướng và có trọng số âm, từ đó so sánh kết quả với các thuật toán khác như Bellman–Ford hoặc Floyd–Warshall.

3. Ứng dụng mô hình vào dữ liệu bản đồ thực tế, cho phép người dùng nhập tọa độ địa lý và quan sát trực quan đường đi ngắn nhất theo chi phí di chuyển thực tế.

Tổng kết lại, kết quả của đề án không chỉ góp phần củng cố cơ sở lý thuyết về thuật toán Dijkstra mà còn minh chứng cho khả năng ứng dụng mạnh mẽ của nó trong các lĩnh vực có yếu tố tối ưu hóa, đặc biệt là trong các bài toán liên quan đến giao thông, mạng máy tính và hệ thống thông minh.

Tài liệu tham khảo

Bellman, R. (1958). On a Routing Problem. *Quarterly of Applied Mathematics*, *16*, 87–90.

Bondy, U. S. R., J. A. Murty. (2008). *Graph Theory*. Springer.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd). MIT Press.

Dijkstra, E. W. (1959). *A Note on Two Problems in Connexion with Graphs* (Vol. 1). Springer.

Ford, L. R. (1956). *Network Flow Theory* (tech. rep.). RAND Corporation.

West, D. B. (2001). *Introduction to Graph Theory*. Prentice Hall.

Phụ lục

1. Thừa nhận sử dụng AI

Nhóm xác nhận có sử dụng ChatGPT và các công cụ AI khác trong quá trình biên soạn báo cáo, nhằm cải thiện tính mạch lạc, cấu trúc diễn đạt và hạn chế lỗi ngữ pháp. Việc sử dụng công cụ này chỉ nhằm mục đích hỗ trợ trình bày và đảm bảo tính rõ ràng của nội dung, hoàn toàn không can thiệp hay thay đổi các kết quả, số liệu, hay nội dung chuyên môn trong báo cáo.

Toàn bộ quá trình xây dựng mô hình, thiết kế thuật toán, lập trình, thu thập và xử lý dữ liệu, phân tích kết quả cũng như viết phần nội dung kỹ thuật đều được thực hiện trực tiếp bởi các thành viên trong nhóm.

2. Mã nguồn dự án và file kết quả các thí nghiệm

Toàn bộ mã nguồn của đồ án, bao gồm các tệp chương trình, dữ liệu thử nghiệm và kết quả mô phỏng, được nhóm lưu trữ công khai tại kho GitHub sau:

https://github.com/hoaianthai345/LTPTDL25_TENA.git

3. Phân công công việc

Thành viên	**Nội dung báo cáo**	**Code/Thực thi**
Thái Hoài An	Tổng hợp, định dạng báo cáo	Thuật toán Dijkstra, minh họa kết quả
Hoàng Thụy Hồng Ân	Phần 3.1, 3.2, 4.1, 4.2	Thiết kế thí nghiệm
Dương Phương Anh	Chương 1, Phần 3.3, 3.4	Thiết kế Class Graph

Nguyễn Đức Tuấn Anh	Chương 2: Cơ sở lý thuyết	Thiết kế các thuật toán khác
Nguyễn Thị Minh Anh	Chương 5	Thiết kế các hàm tính năng khác

4. Kiểm tra đạo văn

Kết quả kiểm tra đạo văn của báo cáo được tải lên kho Github ở trên.