ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH

CẤU TRÚC RỜI RẠC (CO1007)

Giải thuật tìm đường đi ngắn nhất - Học kỳ 241

GVHD: Trần Tuấn Anh
SV thực hiện: Phạm Công Võ - 2313946

Tp. Hồ Chí Minh, Tháng 11/2024

Mục lục

1 Bài Toán Người Bán Hàng Du Lịch (Traveling Salesman Problem - TSP)

Bài toán Nhân viên bán hàng du lịch (hay Traveling Salesman Problem - TSP) là một trong những bài toán tối ưu hóa nổi tiếng và lâu đời nhất trong các lĩnh vực toán học, khoa học máy tính và nghiên cứu hoạt động. Được hình thành từ những năm 1930, bài toán này đã thu hút sự quan tâm của rất nhiều nhà nghiên cứu và tiếp tục là một chủ đề trọng tâm trong các nghiên cứu khoa học. Tuy nhiên, những nguyên lý đầu tiên của bài toán có thể được tìm thấy trong các công trình của các nhà toán học vĩ đại như W.R. Hamilton và Thomas Kirkman từ thế kỷ 19.

Bài toán TSP yêu cầu tìm kiếm tuyến đường ngắn nhất mà một nhân viên bán hàng phải di chuyển qua để thăm một tập hợp các thành phố. Nhân viên bán hàng phải thăm mỗi thành phố một lần duy nhất và quay lại thành phố xuất phát, sao cho tổng quãng đường hoặc chi phí là thấp nhất. Toán học mô tả bài toán này dưới dạng một đồ thị, trong đó các thành phố là các đỉnh (nodes) và các con đường nối giữa các thành phố là các cạnh (edges), với trọng số các cạnh có thể là khoảng cách, chi phí hay thời gian di chuyển.

Điều đặc biệt về TSP là bài toán này thuộc loại NP-hard, nghĩa là không có một thuật toán nào có thể giải quyết bài toán này một cách hiệu quả trong thời gian đa thức cho mọi trường hợp. Độ khó của bài toán tăng rất nhanh khi số lượng thành phố tăng lên, làm cho việc tìm kiếm giải pháp trở nên phức tạp và đòi hỏi một lượng tài nguyên tính toán khổng lồ. Tuy nhiên, mặc dù có tính chất lý thuyết phức tạp, bài toán TSP lại có những ứng dụng thực tế sâu rộng trong nhiều lĩnh vực khác nhau, từ hậu cần, lập kế hoạch, đến sản xuất vi mạch và thậm chí là trong nghiên cứu sinh học như giải trình tự DNA.

1.1 Các Ứng Dụng Thực Tế của TSP

TSP, dù là một vấn đề lý thuyết tối ưu hóa, lại có nhiều ứng dụng quan trọng trong thực tế, đặc biệt là trong các lĩnh vực yêu cầu tối ưu hóa các tuyến đường hoặc thứ tự công việc. Một số ứng dụng nổi bật của TSP bao gồm:

- **Hậu cần và vận tải:** Bài toán TSP có thể được áp dụng để tối ưu hóa các tuyến đường vận chuyển hàng hóa, giúp giảm thiểu chi phí và thời gian di chuyển của các phương tiện giao hàng, mang lại hiệu quả kinh tế cao cho các công ty vận tải.
- **Lập kế hoạch sản xuất:** Trong các ngành sản xuất, TSP có thể giúp tối ưu hóa thứ tự thực hiện các công đoạn hoặc các bước sản xuất trong một dây chuyền, nhằm giảm thiểu thời gian chết và tối đa hóa hiệu quả sử dụng tài nguyên.
- **Sản xuất vi mạch:** Trong ngành công nghiệp vi mạch, TSP có thể được sử dụng để tối ưu hóa quá trình di chuyển của các dụng cụ trong khi sản xuất các vi mạch, giúp tiết kiệm chi phí và thời gian xử lý.
- **Giải trình tự DNA:** Một ứng dụng quan trọng khác của TSP là trong sinh học, nơi thuật toán này được sử dụng để xác định trật tự các chuỗi ADN, với mục tiêu tối ưu hóa các bước xử lý và phân tích dữ liệu gen.

TSP không chỉ là một bài toán lý thuyết mà còn đóng vai trò quan trọng trong việc giải quyết các vấn đề thực tiễn phức tạp. Chính vì vậy, TSP tiếp tục là một chủ đề nghiên cứu sôi động, thu hút sự quan tâm của các nhà khoa học và các nhà phát triển thuật toán tối ưu hóa.

1.2 Các Phương Pháp Giải Quyết Bài Toán TSP

Dù có tính chất khó giải quyết trong các trường hợp lớn, nhưng có nhiều phương pháp đã được phát triển để giải quyết TSP, bao gồm cả những thuật toán chính xác và thuật toán xấp xỉ.

- **Phương pháp vũ phu (Brute Force):** Phương pháp này thử tất cả các khả năng có thể để tìm ra giải pháp tốt nhất, tuy nhiên, với số lượng thành phố lớn, phương pháp này trở nên không khả thi vì số lượng phép thử quá lớn.

- **Thuật toán nhánh và cắt (Branch and Bound):** Đây là một phương pháp tối ưu dựa trên cách tiếp cận đệ quy, cắt bỏ những nhánh không khả thi và chỉ tiếp tục duyệt qua những nhánh có thể mang lại giải pháp tối ưu. Mặc dù có thể giúp giảm không gian tìm kiếm, nhưng thuật toán này vẫn gặp khó khăn khi giải quyết bài toán với số lượng thành phố lớn.
- **Thuật toán di truyền (Genetic Algorithms) và Mô phỏng annealing (Simulated Annealing):** Đây là những phương pháp xấp xỉ được áp dụng khi số lượng thành phố lớn. Các thuật toán này không đảm bảo tìm ra giải pháp tối ưu, nhưng có thể tìm ra các giải pháp gần tối ưu trong một thời gian hợp lý.
- **Thuật toán tối ưu hóa đàn kiến (Ant Colony Optimization - ACO):** Thuật toán này dựa trên hành vi tìm kiếm thức ăn của các đàn kiến, giúp tìm ra giải pháp tối ưu thông qua quá trình mô phỏng hành vi tự nhiên.

Bài toán TSP không chỉ là một thách thức lý thuyết về tối ưu hóa mà còn mang lại giá trị nghiên cứu sâu sắc trong nhiều lĩnh vực. Từ các ứng dụng trong hậu cần, sản xuất, cho đến nghiên cứu sinh học, việc giải quyết TSP không chỉ giúp tiết kiệm chi phí mà còn cải thiện hiệu quả công việc trong các ngành công nghiệp quan trọng.

Tuy nhiên, bài toán này cũng đặt ra nhiều thách thức đáng kể. Sự gia tăng nhanh chóng về độ phức tạp của bài toán khi số lượng thành phố tăng lên khiến cho việc tìm kiếm giải pháp tối ưu trở nên rất khó khăn, đặc biệt trong các trường hợp có quy mô lớn. Việc xây dựng các thuật toán hiệu quả cho TSP, đặc biệt trong các trường hợp phức tạp và quy mô lớn, vẫn là một vấn đề cần được giải quyết trong nghiên cứu và ứng dụng thực tế.

2 Thuật Toán Branch and Bound

2.1 Tổng Quan về Thuật Toán Branch and Bound trong Nghiên Cứu

Thuật toán Branch and Bound (B&B) là một phương pháp tối ưu hóa tổ hợp, được sử dụng để giải quyết các bài toán ra quyết định và tối ưu hóa. Đây là một kỹ thuật tìm kiếm theo cây, trong đó không gian tìm kiếm được đại diện dưới dạng một cây, với mỗi nút trong cây đại diện cho một phần không gian giải pháp. Mục tiêu của thuật toán là tìm ra giải pháp tối ưu hoặc gần tối ưu mà không phải kiểm tra toàn bộ không gian tìm kiếm.

**** Cách Hoạt Động của Thuật Toán:***

- ***Khám Phá Không Gian Giải Pháp:*** *Thuật toán B&B chia nhỏ không gian giải pháp của một bài toán thành các bài toán con, gọi là các nhánh. Mỗi nhánh là một phần không gian cần được tiếp tục tìm kiếm.*
- ***Giới Hạn Giải Pháp:*** *Mỗi bước của thuật toán sẽ tính toán một giới hạn (bound) trên giá trị của giải pháp tối ưu có thể đạt được. Các giới hạn này giúp loại bỏ những nhánh không thể dẫn đến một giải pháp tốt hơn so với những giải pháp đã tìm được. Đây chính là bước cắt tỉa (pruning) trong thuật toán.*
- ***Chiến Lược Nhánh:*** *Thuật toán sử dụng một chiến lược nhánh để tạo ra các bài toán con. Điều này thường liên quan đến việc chọn một biến hoặc điểm quyết định, sau đó tạo các nhánh dựa trên các giá trị hoặc quyết định khác nhau ở điểm đó.*
- ***Khám Phá và Cắt Tỉa:*** *Thuật toán tiếp tục khám phá các nhánh trong không gian giải pháp thông qua tìm kiếm theo chiều sâu (depth-first) hoặc theo chiều rộng (breadth-first). Trong quá trình này, thuật toán liên tục cập nhật các giới hạn và loại bỏ các nhánh không thể mang lại kết quả tốt hơn.*
- ***Giải Pháp Tối Ưu:*** *Thuật toán tiếp tục quá trình khám phá và cắt tỉa cho đến khi tìm ra giải pháp tối ưu hoặc hết không gian tìm kiếm. Khi không gian tìm kiếm có giới hạn và bài toán có các ràng buộc rõ ràng, thuật toán B&B đảm bảo sẽ tìm được giải pháp tối ưu.*

Thuật toán Branch and Bound được ứng dụng rộng rãi trong các bài toán tối ưu hóa tổ hợp, chẳng hạn như Bài toán người bán hàng du lịch (Traveling Salesman Problem - TSP), Bài toán ba lô (Knapsack Problem), và Bài toán Clique cực đại (Maximum Clique Problem).

Ngoài ra, thuật toán B&B cũng được áp dụng trong các bài toán phân bổ tài nguyên, lập trình số nguyên, tối ưu hóa toàn cục, thiết kế mạng, và cả trong học máy. B&B cung cấp một phương pháp tiếp cận hệ thống để tìm kiếm và giải quyết các bài toán tối ưu hóa phức tạp trong nhiều lĩnh vực nghiên cứu.

2.2 Thuật Toán Branch and Bound trong Bài Toán Người Bán Hàng Du Lịch (TSP)

Bài toán TSP yêu cầu tìm ra một tuyến đường mà người bán hàng phải đi qua tất cả các thành phố chính xác một lần và quay trở lại thành phố xuất phát sao cho tổng chi phí hoặc khoảng cách di chuyển là tối thiểu. Thuật toán Branch and Bound có thể được sử dụng để giải quyết bài toán này một cách hiệu quả.

**** Cách Thực Hiện Thuật Toán Branch and Bound cho Bài Toán TSP:***

- ***Khởi Tạo:*** *Bắt đầu từ thành phố gốc (ví dụ, thành phố 0). Xây dựng một cây tìm kiếm, trong đó mỗi nút đại diện cho một tập hợp các thành phố đã thăm và những thành phố chưa thăm. Cây tìm kiếm sẽ mở rộng ra từ thành phố này, với mỗi nhánh thể hiện một quyết định tiếp theo về thành phố cần thăm.*
- ***Giới Hạn:*** *Tại mỗi bước, tính toán một giới hạn (bound) cho chi phí tối thiểu có thể đạt được từ một nút trong cây tìm kiếm. Giới hạn này có thể được tính bằng cách sử dụng phương pháp giới hạn dưới, trong đó ta tính toán chi phí tối thiểu của việc di chuyển từ các thành phố hiện tại tới các thành phố chưa thăm, cộng với chi phí tối thiểu để hoàn thành chuyến đi (từ các thành phố còn lại quay lại thành phố xuất phát). Giới hạn giúp giảm số lượng các nhánh cần phải kiểm tra, từ đó giảm độ phức tạp tính toán.*
- ***Nhánh:*** *Từ mỗi nút hiện tại, tạo các nhánh con tương ứng với các thành phố tiếp theo cần thăm. Mỗi nhánh sẽ có một giới hạn riêng, được tính toán dựa trên chi phí đã di chuyển và chi phí ước tính còn lại để hoàn thành chuyến đi.*
- ***Chọn Nhánh Tốt Nhất:*** *Lựa chọn nhánh có giới hạn nhỏ nhất để tiếp tục phát triển. Điều này giúp thuật toán tập trung vào những nhánh có khả năng mang lại một chuyến đi với chi phí nhỏ nhất.*
- ***Cắt Tỉa (Pruning):*** *Nếu giới hạn của một nhánh lớn hơn chi phí của chuyến đi tốt nhất đã tìm được (nếu có), loại bỏ nhánh đó khỏi cây tìm kiếm. Nếu một chuyến đi hoàn chỉnh (thăm tất cả các thành phố và quay lại thành phố gốc) được tìm thấy và chi phí của nó nhỏ hơn chuyến đi tốt nhất hiện tại, cập nhật lại chuyến đi tốt nhất.*
- ***Lặp Lại:*** *Tiếp tục các bước nhánh, lựa chọn nhánh tốt nhất và cắt tỉa cho đến khi tất cả các nhánh đều được xem xét hoặc loại bỏ. Khi thuật toán kết thúc, chuyến đi có chi phí thấp nhất sẽ là giải pháp tối ưu.*

3 Chi tiết hiện thực hàm Traveling

Kriển khai hàm Traveling để giải quyết bài toán TSP bằng cách áp dụng phương pháp nhánh cận cắt tỉa, nhằm tối ưu hóa lộ trình và giảm thiểu chi phí tổng thể.

3.1 calculateFirstEdgeCost()

Mục đích: Hàm này được thiết kế để tìm cạnh có trọng số nhỏ nhất từ một đỉnh cụ thể trong đồ thị, với điều kiện cạnh đó phải hợp lệ (không phải cạnh nối tới chính nó và có trọng số lớn hơn 0). Đây là bước quan trọng để xác định cận dưới (bound) trong thuật toán nhánh cận.

1 *Đầu vào:*

- *Ma trận trọng số của đồ thị G (kích thước tối đa là* $30x30$*).*
- *Chỉ số đỉnh đang xét thứ i.*
- *Tổng số đỉnh trong đồ thị numVertices.*

2 Thuật toán:

- *Tìm giá trị nhỏ nhất khác 0 từ đỉnh i đến các đỉnh khác.*

3 Cách thức thực hiện chi tiết:

- *Khởi tạo giá trị nhỏ nhất (min) bằng INT_MAX để đảm bảo bất kỳ giá trị cạnh hợp lệ nào cũng sẽ nhỏ hơn.*
- *Duyệt qua tất cả các đỉnh kết nối với đỉnh i.*
  - *Nếu cạnh nối là hợp lệ (k != i và G[i][k] != 0): Kiểm tra xem giá trị trọng số có nhỏ hơn min không. Nếu đúng, cập nhật min.*
- *Nếu không tìm thấy cạnh hợp lệ nào (giá trị của min vẫn là INT_MAX), trả về 0. Nếu không, trả về giá trị min.*

4 Trả về:

- *Giá trị trọng số nhỏ nhất từ đỉnh i đến một đỉnh khác nếu tồn tại.*
- *Trả về 0 nếu không tìm thấy cạnh hợp lệ.*

3.2 computeSecondEdgeCost()

Mục đích: Hàm này tìm trọng số của cạnh nhỏ thứ hai từ một đỉnh cụ thể trong đồ thị. Điều này hỗ trợ việc tính toán cận dưới chính xác hơn, giúp cải thiện hiệu quả cắt tỉa các nhánh không cần thiết.

1 Đầu vào:

- *Ma trận trọng số của đồ thị G (kích thước tối đa là 30 x 30).*
- *Chỉ số đỉnh đang xét thứ i.*
- *Tổng số đỉnh trong đồ thị numVertices.*

2 Thuật toán:

- *Theo dõi hai giá trị nhỏ nhất khác 0*
- *Ưu tiên cập nhật cạnh nhỏ nhất trước*

3 Cách thức thực hiện chi tiết:

- *Khởi tạo hai biến one và two để lưu giá trị nhỏ nhất và nhỏ thứ hai, cả hai đều được đặt bằng INT_MAX.*
- *Duyệt qua tất cả các đỉnh nối với đỉnh i.*
  - *Nếu cạnh không hợp lệ (i == j hoặc G[i][j] == 0), bỏ qua.*
  - *Nếu trọng số của cạnh nhỏ hơn one:*
    - *Cập nhật two bằng giá trị của one.*
    - *Cập nhật one bằng trọng số của cạnh hiện tại.*
  - *Nếu trọng số của cạnh lớn hơn one nhưng nhỏ hơn hoặc bằng two và không bằng one:*
    - *Cập nhật two.*
- *Nếu không tìm thấy giá trị nhỏ thứ hai (two == INT_MAX), trả về 0. Nếu không, trả về giá trị của two.*

4 Trả về:

- *Trọng số cạnh nhỏ thứ hai từ đỉnh i nếu tồn tại.*
- *0 nếu không tồn tại cạnh hợp lệ.*

3.3 processTSPBranch()

Mục đích: Thực hiện thuật toán nhánh và cận để tìm đường đi ngắn nhất trong bài toán TSP. Hàm này duyệt đệ quy qua tất cả các nhánh có khả năng và loại bỏ (cắt tỉa) những nhánh không hứa hẹn dựa trên cận dưới.

1 *Đầu vào:*

- *Ma trận trọng số của đồ thị G (30 x 30).*
- ***currentBound:*** *Biên hiện tại (Giá trị cận dưới của nhánh hiện tại).*
- ***currWeight:*** *Tổng trọng số của các cạnh trong chu trình đang được xây dựng.*
- ***level:*** *Độ sâu hiện tại trong cây nhánh cận (số đỉnh đã đi qua trong chu trình).*
- ***numVertices:*** *Tổng số đỉnh trong đồ thị.*
- *Các tham số để theo dõi đường đi và trạng thái:*
  - ***minimumCost:*** *Tổng trọng số nhỏ nhất của chu trình đã tìm thấy.*
  - ***routePath[]:*** *Mảng lưu trữ đường đi tạm thời tại nhánh hiện tại.*
  - ***nodeVisited[]:*** *Mảng trạng thái để kiểm tra đỉnh nào đã được duyệt.*
  - ***optimalPath[]:*** *Mảng lưu trữ đường đi tối ưu tương ứng với minimumCost.*

2 *Thuật toán:*

- *Duyệt đệ quy các nhánh có khả năng*
- *Cắt tỉa các nhánh không hứa hẹn*
- *Cập nhật đường đi tối ưu*
- *Sử dụng cận để loại bỏ các nhánh không hiệu quả*
- *Quay lui (backtracking) để khám phá các khả năng*

3 *Cách thức thực hiện chi tiết:*

a *Kiểm tra điều kiện dừng:*

- *Mục đích: Kiểm tra xem chu trình Hamilton đã hoàn thành chưa (tất cả các đỉnh đã được duyệt).*
- *Quy trình:*
  - *Bước 1: Nếu tất cả các đỉnh đã được đi qua (level == numVertices) thì đến bước 2.*
  - *Bước 2: Kiểm tra xem có cạnh từ đỉnh cuối cùng quay về đỉnh đầu tiên hay không*
    $$(G[routePath[level-1]][routePath[0]] ! = 0).$$
    Nếu có:
  - *Tính tổng trọng số của chu trình:*
    $$currRes = currWeight + G[routePath[level-1]][routePath[0]].$$
  - *So sánh với minimumCost: Nếu nhỏ hơn, cập nhật minimumCost và lưu đường đi tối ưu vào optimalPath.*
  - *Bước 3: Dừng nhánh bằng lệnh return.*

b *Duyệt qua các nhánh tiềm năng:*

- *Mục đích: Thử tất cả các đỉnh chưa được duyệt, kiểm tra xem có thể thêm vào chu trình không.*
- *Cách thực hiện:Thử từng đỉnh chưa được duyệt:*
  - *Tính cận dưới mới dựa trên trọng số cạnh và các chi phí cận (gọi hàm calculateFirstEdgeCost và computeSecondEdgeCost).*
  - *Nếu cận dưới và trọng số hiện tại nhỏ hơn minimumCost, tiếp tục mở rộng chu trình bằng đệ quy.*

c *Quay lui (Backtracking):*

- *Hoàn tác các thay đổi (trọng số, cận dưới, trạng thái đánh dấu) để thử nhánh khác.*

d *Cắt tỉa nhánh (Pruning):*

- *Bỏ qua các nhánh có cận dưới và trọng số lớn hơn hoặc bằng minimumCost.*

3.4 Hàm Traveling

Mục đích: Hàm chính giải quyết TSP

1 *Đầu vào:*

- *Ma trận đồ thị G*
- *Số lượng đỉnh numVertices*
- *Đỉnh bắt đầu start*

2 *Các bước chi tiết:*

- *Tiền xử lý ma trận: Chuyển tất cả các giá trị 0 (ngoại trừ trên đường chéo chính) thành INT_MAX để biểu thị cạnh không tồn tại.*
- *Tính toán biên ban đầu:*
  - *Dựa trên tổng các giá trị calculateFirstEdgeCost và computeSecondEdgeCost cho mỗi đỉnh.*
  - *Chia đôi tổng này để lấy giá trị cận dưới.*
- *Khởi tạo trạng thái (thiết lập điểm bắt đầu):*
  - *Đánh dấu đỉnh bắt đầu là đã thăm.*
  - *Đặt giá trị biên (currentBound) và trọng số hiện tại (currWeight) bằng 0.*
- *Gọi đệ quy: Duyệt qua tất cả các nhánh và tìm đường đi tối ưu bằng cách gọi hàm processTSPBranch.*
- *Chuyển đổi kết quả (mảng optimalPath) thành chuỗi các đỉnh theo thứ tự.*

3 *Trả về:*

- *Chuỗi các đỉnh thể hiện chu trình Hamilton tối ưu.*

4 Mã nguồn C++ sử dụng đệ quy và phân nhánh và thuật toán liên kết để giải bài toán Nhân viên bán hàng du lịch (TSP)

Phần 1: Mã C++

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <sstream>
#include <limits.h>
#include <cstring>

#using namespace std;

const int INT_MAX = 2147483647;

int calculateFirstEdgeCost(int G[30][30], int i, int numVertices) {
    int min = INT_MAX;
    for (int k = 0; k < numVertices; k++){
        if (k != i && G[i][k] != 0 && G[i][k] < min)
        {
```

```
                min = G[i][k];
        }
    }
    if(min == INT_MAX) return 0;
    return min;
}

int computeSecondEdgeCost(int G[30][30], int i, int numVertices)
{
    int one = INT_MAX;
    int two = INT_MAX;
    for (int j = 0; j < numVertices; j++) {
        if (i == j || G[i][j] == 0)
        {
             continue;
        }

        if (G[i][j] <= one)
        {
            two = one;
            one = G[i][j];
        }
        else if (G[i][j] <= two && G[i][j] != one)
        {
            two = G[i][j];
        }

    }
    return two == INT_MAX ? 0 : two;
}

void processTSPBranch(int G[30][30], int currentBound, int currWeight, int level,
            int routePath[], bool nodeVisited[], int &minimumCost, int optimalPath[], int
                numVertices) {
    if (level == numVertices)
    {
        if (G[routePath[level - 1]][routePath[0]] != 0)
        {
            int currRes = currWeight + G[routePath[level - 1]][routePath[0]];
            if (currRes < minimumCost)
            {
                for (int i = 0; i < numVertices; i++)
                {
                    optimalPath[i] = routePath[i];
                }

                optimalPath[numVertices] = routePath[0];
                minimumCost = currRes;
            }
        }
        return;
    }


    for (int i = 0; i < numVertices; i++)
    {
        if (G[routePath[level - 1]][i] != 0 && !nodeVisited[i])
        {
            int temp = currentBound;
            currWeight += G[routePath[level - 1]][i];

            if (level == 1)
```

```
            currentBound -= ((calculateFirstEdgeCost(G, routePath[level - 1], numVertices) +
                             calculateFirstEdgeCost(G, i, numVertices)) / 2);
        else
            currentBound -= ((computeSecondEdgeCost(G, routePath[level - 1], numVertices) +
                             calculateFirstEdgeCost(G, i, numVertices)) / 2);

        if (currentBound + currWeight < minimumCost) {
            routePath[level] = i;
            nodeVisited[i] = true;

            processTSPBranch(G, currentBound, currWeight, level + 1,
                   routePath, nodeVisited, minimumCost, optimalPath, numVertices);

            nodeVisited[i] = false;
        }
        currWeight -= G[routePath[level - 1]][i];
        currentBound = temp;
    }
  }
}

string Traveling(int G[30][30], int numVertices, char start) {
    int routePath[31];
    int optimalPath[31];
    bool nodeVisited[30] = {false};
    string routeResult;

    for (int i = 0; i < numVertices; i++)
    {
        for (int j = 0; j < numVertices; j++)
        {
            if (G[i][j] == 0 && i != j)
            {
                G[i][j] = INT_MAX;
            }
        }
    }

    int currentBound = 0;
    for (int i = 0; i < numVertices; i++)
        currentBound += (calculateFirstEdgeCost(G, i, numVertices) +
                    computeSecondEdgeCost(G, i, numVertices));

    currentBound = (currentBound & 1) ? currentBound / 2 + 1 : currentBound / 2;

    nodeVisited[start - 'A'] = true;
    routePath[0] = start - 'A';

    int minimumCost = INT_MAX;

    processTSPBranch(G, currentBound, 0, 1, routePath, nodeVisited, minimumCost, optimalPath,
         numVertices);


    for (int i = 0; i <= numVertices; i++)
    {
        routeResult += (char)(optimalPath[i] + 'A');
        if (i < numVertices) routeResult += " ";
    }

    return routeResult;
}
```

Phần 2: Hàm main() C++

```
    #define MAX_VERTICES 30

int main() {
    int graph[MAX_VERTICES][MAX_VERTICES] = {
    {0, 12, 29, 22, 13, 27, 18, 25, 40, 36, 50, 28},
    {12, 0, 15, 35, 19, 23, 29, 41, 17, 24, 38, 20},
    {29, 15, 0, 30, 31, 10, 21, 26, 45, 33, 25, 39},
    {22, 35, 30, 0, 18, 12, 31, 50, 20, 17, 44, 32},
    {13, 19, 31, 18, 0, 29, 16, 34, 25, 27, 14, 24},
    {27, 23, 10, 12, 29, 0, 40, 22, 35, 20, 28, 30},
    {18, 29, 21, 31, 16, 40, 0, 28, 39, 15, 24, 19},
    {25, 41, 26, 50, 34, 22, 28, 0, 31, 26, 45, 35},
    {40, 17, 45, 20, 25, 35, 39, 31, 0, 12, 18, 22},
    {36, 24, 33, 17, 27, 20, 15, 26, 12, 0, 19, 21},
    {50, 38, 25, 44, 14, 28, 24, 45, 18, 19, 0, 37},
    {28, 20, 39, 32, 24, 30, 19, 35, 22, 21, 37, 0}
};
    int numVertices = 12;
    char start = 'A';
    char goal= 'I';
    string path = BF_Path(graph, numVertices, start, goal);
    // string path = Traveling(graph, numVertices, start);

    cout << "BF path: " << path << endl;

    return 0;
}
```

Tài liệu

https://en.wikipedia.org/wiki/Travelling_salesman_problem