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 CHO KHOA HỌC MÁY TÍNH

Giải quyết bài toán

Traveling Saleman Problem

GVHD: Trần Hồng Tài
SV: Võ Tiến Nam - 2312205

TP. HỒ CHÍ MINH, THÁNG 6/2024

Mục lục

1 Giới thiệu về bài toán Traveling Salesman Problem

1.1 Giới thiệu

- Bài toán Traveling Salesman Problem (TSP), hay còn được biết với cái tên là bài toán tìm đường đi của người bán hàng, là một bài toán tối ưu hóa kinh điển trong lý thuyết đồ thị và lập trình. Bài toán đặt ra câu hỏi: một người bán hàng đi du lịch muốn đi qua một loạt các thành phố, mỗi thành phố chỉ được đi qua một lần, sau đó trở về thành phố ban đầu. Mục tiêu là tìm ra hành trình với tổng chi phí di chuyển nhỏ nhất.
- TSP là một bài toán được đánh giá là NP-Hard (Non-deterministic Polynomial-time hard), nghĩa là không có một thuật toán đa thức chính xác nào đã biết đến giải được bài toán. Tuy nhiên, ta vẫn có thể giải quyết vấn đề trên bằng những thuật toán xấp xỉ và Heuristic, qua đó ta có thể tìm ra lời giải gần đúng cho các bài toán tối ưu hóa, đặc biệt là bài toán mà việc tìm lời giải chính xác là không thực tế vì mất quá nhiều thời gian nhưng vẫn giải quyết vấn đề một cách hiệu quả trong thực tế.

1.2 Ứng dụng

- TSP có rất nhiều ứng dụng trong thực tế, từ việc lập lịch giao hàng cho các hãng vận chuyển, quản lý hệ thống giao thông đô thị, tới lập kế hoạch du lịch và đi lại.
- Ứng dụng trong lập trình và nghiên cứu: TSP là một bài toán kinh điển trong lĩnh vực lý thuyết đồ thị và tối ưu hóa, làm nền tảng cho nhiều nghiên cứu trong lĩnh vực này.
- Tối ưu hóa tài nguyên: Giải quyết TSP giúp tối ưu hóa sử dụng tài nguyên như thời gian và chi phí, từ đó giúp tăng hiệu quả và giảm chi phí.
- Ứng dụng trong công nghệ: TSP cũng có ứng dụng trong các lĩnh vực công nghệ như mạng lưới cảm biến không dây, viễn thông, và Machine Learning

2 Cách giải quyết bài toán Traveling Salesman Problem

2.1 Phương pháp

- Vì mục tiêu của TSP là tìm ra hành trình với tổng chi phí nhỏ nhất, đồng thời mỗi thành phố chỉ được ghé thăm đúng một lần, vì thế chúng ta sẽ áp dụng chu trình Hamilton để giải quyết bài toán này
- **Chu trình Hamilton:** Chu trình Hamilton là một đường đi qua tất cả các đỉnh của đồ thị mà mỗi đỉnh đều được ghé qua một lần duy nhất và trở về đỉnh xuất phát. Trong ngữ cảnh của bài toán TSP, một chu trình Hamilton sẽ tương ứng với một hành trình du lịch mà mỗi thành phố được ghé qua đúng một lần.

2.2 Hướng tiếp cận vấn đề

- Ta sẽ xây dựng một chu trình Hamilton, tức là một chuỗi đỉnh trong đồ thị sao cho mỗi đỉnh chỉ được ghé thăm một lần và cuối cùng quay trở lại đỉnh xuất phát. - Với chu trình Hamilton, mục tiêu được đặt ra là tìm đường đi có tổng chi phí nhỏ nhất đi qua tất cả các đỉnh trong đồ thị, với kết quả chính xác nhất có thể.
+ **Ưu điểm:** Lời giải bằng chu trình Hamilton sẽ tối ưu bởi vì mỗi đỉnh trong đồ thị chỉ được

ghé thăm một lần. Thêm vào đó nó dễ hiểu và dễ dàng thực thi, đặc biệt với những đồ thị nhỏ.
+ **Nhược điểm:** Với những đồ thị lớn với nhiều đỉnh, việc sử dụng chu trình Hamilton có thể sẽ khó khăn hơn, thời gian tính toán sẽ tăng lên theo số lượng đỉnh. Ngoài ra với những đồ thị không đầy đủ cũng sẽ gây khó khăn cho quá trình giải quyết
- Với hiểu biết về ưu, nhược điểm chu trình Hamilton, chúng ta có thể áp dụng nó một cách hiệu quả để giải quyết bài toán Traveling Saleman Problem.

2.3 Thuật toán

- Trong bài tập lớn này, em đã sử dụng chu trình Hamilton với cách tiếp cận Backtracking (Thuật toán quay lui). Đây là một thuật toán có độ phức tạp thời gian là **O(n!)**, bởi vì nó thử hết tất cả các hoán vị có thể có của đỉnh, có n! cách để sắp xếp n đỉnh. Tuy nhiên, trong thực tế, thuật toán có thể cắt bớt một số nhánh không cần thiết trong không gian tìm kiếm, do đó thời gian thực tế có thể ít hơn nhiều so với trường hợp tệ nhất là O(n!).
- **Cách cài đặt:** Tất cả các đỉnh ban đầu trong đồ thị ban đầu sẽ được đánh dấu là "chưa được ghé thăm", sau đó xét hết tất cả các đỉnh trong đồ thị. Nếu đỉnh chưa được ghé thăm, và chi phí để di chuyển từ đỉnh hiện tại đến đỉnh mới không phải là không xác định, thì những điều sau sẽ được thực hiện:

- Đỉnh đó sẽ được đánh dấu là "đã ghé thăm"
- Đỉnh hiện tại sẽ được cập nhật (là đỉnh mới được đánh dấu "đã ghé thăm")
- Giá trị chi phí sẽ được lưu lại và cộng vào tổng chi phí của đường đi hiện tại

+ Sau khi đã thăm hết tất cả các đỉnh, ta sẽ kiểm tra xem có thể trở về đỉnh bắt đầu được hay không.

- Nếu có, cập nhật tổng chi phí bằng cách cộng thêm chi phí từ đỉnh kết thúc về đỉnh bắt đầu. Sau đó thực hiện so sánh để tìm ra hành trình có chi phí nhỏ nhất (điều này có nghĩa là mỗi khi tìm được một con đường mới, nó sẽ được so sánh với con đường cũ có chi phí thấp nhất).
- Nếu không, thuật toán sẽ quay lại bước trước đó (backtracking) để thử các đỉnh khác.

Quá trình này sẽ được lặp lại cho đến khi tất cả các khả năng đều đã được xem xét và hành trình có chi phí nhỏ nhất được tìm thấy.

3 Chương trình giải bài toán:

- Dưới đây là đoạn mã minh họa chi tiết cách cài đặt thuật toán:

```
#include "tsp.h"

vector<int> bestPath;

void TSP(int i, int num_vertex, vector<int> &X, vector<int> &visited,

vector<vector<int>> &cost, int &dist, int &result, int min_cost, int start_vertex)
{
    for (int j = 1; j <= num_vertex; j++)
        // If not visited and can move to vertex j
        if (!visited[j] && cost[X[i - 1]][j] != INT_MAX)
        {
```

```
            visited[j] = 1; // Mark as visited
            X[i] = j;      // Store vertex j
            int temp_dist = dist;
            if (cost[X[i - 1]][X[i]] != INT_MAX)
            {
                temp_dist += cost[X[i - 1]][X[i]]; // Update distance
            }
            if (i == num_vertex) // If all vertices are visited
            {
                if (cost[X[num_vertex]][start_vertex] != INT_MAX)
                {
                    temp_dist += cost[X[num_vertex]][start_vertex]; // Update distance
                    if (temp_dist < result)
                    {
                        result = temp_dist;                     // Update

                        shortest distance
                        // Store the shortest path
                        bestPath = vector<int>(X.begin(), X.end());
                    }
                }
            }
            // If current distance is less than the shortest distance
            else if (temp_dist + (num_vertex - i + 1) * min_cost < result)
            {
                dist = temp_dist;
                TSP(i + 1, num_vertex, X, visited, cost, dist, result,

                min_cost, start_vertex); // Recursive, continue searching
                dist -= cost[X[i - 1]][X[i]]; // Backtrack
            }
            visited[j] = 0; // Unmark
        }
}

string Traveling(int Graph[20][20], int num_vertex, char start)
{
    vector<vector<int>> cost(num_vertex + 1, vector<int>(num_vertex + 1, INT_MAX));
    for (int i = 0; i < num_vertex; i++)
        for (int j = 0; j < num_vertex; j++)
            if (Graph[i][j] != 0)
                cost[i + 1][j + 1] = Graph[i][j]; // Store adjacency matrix
    int start_vertex = start - 'A' + 1;
     // If unable to return to the start vertex
    if (Graph[num_vertex - 1][start_vertex - 1] == 0)
        cost[num_vertex][start_vertex] = INT_MAX; // Assign INT_MAX
    vector<int> visited(num_vertex + 1, 0);
    vector<int> X(num_vertex + 1, 0);
    int dist = 0;
    int result = INT_MAX;
    visited[start_vertex] = 1;
    X[1] = start_vertex;
    int min_cost = INT_MAX;
    for (int i = 1; i <= num_vertex; i++) // Traverse all vertices
        for (int j = 1; j <= num_vertex; j++)
            if (i != j && cost[i][j] < min_cost) // Find the minimum value
                min_cost = cost[i][j];
    // Find the next vertex
    TSP(2, num_vertex, X, visited, cost, dist, result, min_cost, start_vertex);
    string res = "";
    if (!bestPath.empty() && bestPath.size() >= num_vertex)
    {
```

```
        if (cost[bestPath[num_vertex]][start_vertex] == INT_MAX)
        {
            cost[bestPath[num_vertex]][start_vertex] = 0;
            bestPath.push_back(start_vertex);
        }
        for (int i = 1; i <= num_vertex; i++)
        {
            res += char(bestPath[i] + 'A' - 1);
            res += " ";
        }
        res += char(bestPath[1] + 'A' - 1);
        res += " ";
        res += to_string(result);
    }
    return res;
}
```

Traveling Salesman Problem in C++

4 Kết luận

Trong báo cáo này, chúng ta đã tìm hiểu và giải quyết bài toán Traveling Salesman Problem (TSP), một bài toán tối ưu hóa kinh điển và khó khăn trong lý thuyết đồ thị và lập trình. Chúng tôi đã trình bày các phương pháp giải quyết TSP, bao gồm thuật toán chu trình Hamilton, và cài đặt một giải pháp cụ thể bằng C++.

Kết quả cho thấy rằng phương pháp áp dụng có thể tìm ra hành trình với tổng chi phí nhỏ nhất trong một số trường hợp cụ thể. Mặc dù thuật toán này đơn giản và dễ hiểu, nhưng nó gặp hạn chế về hiệu suất khi số lượng đỉnh tăng lên do tính chất NP-Hard của bài toán. Điều này dẫn đến thời gian tính toán tăng lên rất nhanh khi số lượng thành phố tăng, làm cho thuật toán không thực tế cho các bài toán có kích thước lớn.

Để nâng cao hiệu quả giải quyết TSP trong tương lai, các nghiên cứu có thể tập trung vào việc phát triển các thuật toán xấp xỉ và Heuristic mới. Các phương pháp này có thể cung cấp lời giải gần đúng trong thời gian hợp lý, phù hợp với các ứng dụng thực tế hơn.

Tài liệu

[1] Lê Minh Hoàng, *Lý thuyết đồ thị, phần Chu trình Hamilton, đường đi Hamilton, đồ thị Hamilton.*

[2] Bộ môn Cấu trúc rời rạc cho Khoa học Máy tính, *Bài giảng Connectivity, mục Hamilton Paths and Circuits*, Trường Đại học Bách khoa TP.HCM.

[3] *P, NP, CoNP, NP hard and NP complete | Complexity Classes*: https://www.geeksforgeeks.org/types-of-complexity-classes-p-np-conp-np-hard-and-np-complete/