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

BÁO CÁO BÀI TẬP LỚN

CẤU TRÚC RỜI RẠC CHO KHOA HỌC MÁY TÍNH

ĐỀ TÀI: PHÂN TÍCH HƯỚNG TIẾP CẬN VÀ GIẢI THUẬT CHO BÀI TOÁN TRAVELING SALESMAN

Giảng viên hướng dẫn: MAI XUÂN TOÀN
Sinh viên thực hiện:

Họ và tên	MSSV	Mã lớp
Lê Trọng Tín	2313452	Ł05

TP Hồ Chí Minh, 2024

Mục lục

1 Giới thiệu bài toán

Bài toán người du lịch (Traveling Salesman Problem) là được biết đến là một trong những bài toán kinh điển và phức tạp nhất của lí thuyết đồ thị. Bài toán có yêu cầu như sau: "Giả sử rằng cho trước một danh sách các thành phố và khoảng cách giữa mỗi cặp thành phố, một người muốn đi đến tất cả các thành phố này, yêu cầu đặt ra là hãy xác định một lộ trình với tổng quãng đường đi ngắn nhất sao cho người này chỉ đi qua mỗi thành phố đúng một lần và quay trở về thành phố nơi người này chọn làm điểm xuất phát".

Bài toán này không chỉ có giá trị về mặt lí thuyết mà còn có nhiều ứng dụng thực tiễn trong các lĩnh vực như Logistic, thiết kế các lịch trình đường đi và thiết kế mạch điện. Việc tìm ra lộ trình tối ưu giúp tiết kiệm chi phí và tài nguyên, đồng thời cải thiện hiệu suất hoạt động của các hệ thống phức tạp. Mặc dù có nhiều phương pháp tiếp cận để giải quyết bài toán này, từ các thuật toán đơn giản như tìm kiếm vét cạn, sử dụng kỹ thuật nhánh cận và quay lui đến các kỹ thuật nâng cao như sử dụng trí tuệ nhân tạo và học máy, việc tìm ra lời giải tối ưu vẫn là một thách thức lớn đối với các nhà nghiên cứu và kỹ sư.

2 Cơ sở lí thuyết

Bài toán này dựa trên các kiến thức nền tẳng về lí thuyết đồ thị, đồng thời yêu cầu kĩ năng lập trình cơ bản và cần trang bị thêm một vài giải thuật cần thiết để có thể giải quyết được bài toán này, đảm bảo rằng sẽ cho ra được kết quả chính xác đồng thời phải tối ưu hóa được thời gian chạy của chương trình.

Trong bối cảnh của bài toán Traveling Salesman (TSP), mỗi đỉnh đại diện cho một thành phố và mỗi cạnh đại diện cho khoảng cách hoặc chi phí di chuyển giữa hai thành phố. Từ đó, bài toán TSP yêu cầu tìm một chu trình Hamiltonian tối thiểu trong đồ thị hoàn chỉnh có trọng số, tức là một chu trình đi qua mỗi đỉnh đúng một lần và quay lại đỉnh ban đầu, sao cho tổng trọng số của các cạnh trong chu trình là nhỏ nhất.

Ta cần ôn lại một só khái niệm như sau:

- **Đồ thị hoàn chỉnh**: Là đồ thị mà mỗi cặp đỉnh đều được nối với nhau bởi một cạnh. Trong bài toán TSP, đồ thị thường được giả định là hoàn chỉnh.
- **Chu trình Hamiltonian**: Là một chu trình đi qua mỗi đỉnh của đồ thị đúng một lần và quay lại đỉnh ban đầu.

3 Phân tích hướng tiếp cận và giải thuật

Như đã đề cập ở trên, TSP có thể được giải quyết bằng nhiều hướng, trong phạm vi của bài tập lớn lần này, em xin được trình bày cách giải của mình theo hướng nhánh và cận (Branch and Bound), kỹ thuật này sẽ dựa vào giá trị giới hạn cận trên và cận dưới của khoảng cách mà từ đó sẽ loại bỏ đi những nhánh không khả thi sớm nhất có thể. Phương pháp này sẽ duyệt qua tất cả các lộ trình có thể

và tính toán chi phí tạm thời. Nếu chi phí tạm thời lớn hơn khoảng cách tối ưu đã tính hiện tại, nhành này sẽ được loại bỏ.

Kết hợp với phương pháp nhánh và cận, đoạn code còn sử dụng kỹ thuật quy hoạch động để lưu trữ các kết quả trung gian, giảm thiểu việc tính toán lặp lại và tăng tốc quá trình tìm kiếm. Quy hoạch động giúp tối ưu hóa bài toán bằng cách chia nhỏ bài toán lớn thành các bài toán con và giải quyết từng bài toán con một cách tối ưu. Sau đó, các kết quả của bài toán con được kết hợp để tìm ra lời giải cho bài toán lớn ban đầu.

Giả sử rằng đầu vào của bài toán là một ma trận G kiểu `int` có kích thước tối đa là 20×20, biểu diễn các khoảng cách giữa các thành phố. Mỗi phần tử $G[i][j]$ của ma trận đại diện cho khoảng cách từ thành phố i đến thành phố j, trong đó giá trị 0 có thể được sử dụng để biểu thị rằng không có đường trực tiếp giữa hai thành phố. Đỉnh bắt đầu là một ký tự kiểu `char` biểu diễn một trong các thành phố trong ma trận. Ký tự này sẽ được chuyển đổi sang chỉ số tương ứng trong ma trận G để sử dụng trong quá trình tính toán.

Đây là protoype của hàm Traveling:

```
string Traveling(const int G[20][20], int n, char startVertex);
```

3.1 Hàm calculatePathLength

Hàm `calculatePathLength` có chức năng tính tổng chi phí của một lộ trình dựa trên ma trận khoảng cách và lộ trình đầu vào dưới dạng chuỗi.

```
int calculatePathLength(const int G[20][20], int n, const string &
    path) {
    stringstream ss(path);
    char vertex;
    int vertices[20];
    int num_vertex_path = 0;

    // chuyen doi cac ki tu trong chuoi thanh cac index
    // tuong ung trong ma tran G
    while (ss >> vertex) {
        vertices[num_vertex_path++] = vertex - 'A';
    }

    int total_cost = 0;
    for (int i = 0; i < num_vertex_path - 1; i++) {
        // Tinh toan tong chi phi cua lo trinh bang cach
        // cong don cac trong so giua cac dinh lien tiep trong
        // lo trinh
        total_cost += G[vertices[i]][vertices[i + 1]];
    }
    return total_cost;
}
```

Trong hàm này đã sử dụng `stringstream` để phân tích chuỗi đầu vào `path` thành các đỉnh tương ứng. Sau đó, cộng dồn các trọng số cạnh giữa các đỉnh liên tiếp trong lộ trình. Cuối cùng, hàm trả về giá trị là tổng chi phí của lộ trình `path`.

3.2 Hàm constructPathString

Hàm `constructPathString` có chức năng xây dựng chuỗi lộ trình tối ưu từ các đỉnh đã được tìm thấy.

```
string constructPathString(int n)
{
    string result = "";
    for (int i = 0; i < n; ++i)
    {
        result += string(1, path_toi_uu[i] + 'A') + " ";
    }
    return result + string(1, path_toi_uu[0] + 'A');
}
```

3.3 Hàm find_next

Hàm `find_next` được sử dụng để tìm đỉnh kế tiếp trong lộ trình hiện tại và tính toán chi phí tương ứng.

```
// Tim dinh ke tiep trong lo trinh
void find_next(const int G[20][20], int n, int currentCost, int
    level)
{
    for (int i = 0; i < n; i++)
    {
        // Kiem tra xem dinh i da duoc tham hay chua va
        // chi phi di tu dinh hien tai den dinh i khac 0
        if (!visited[i] && G[curent_travel[level - 1]][i] != 0)
        {
            // Danh dau dinh i la da tham
            visited[i] = true;
            // Luu dinh i vao lo trinh hien tai
            curent_travel[level] = i;
            // Tinh chi phi tam thoi cho lo trinh
            int tempCost = currentCost + G[curent_travel[level -
                1]][i];
            // Neu chi phi tam thoi cong them chi phi uoc luong
            // con lai nho hon chi phi nho nhat hien tai
            if (tempCost + (n - level) * min_trong_so < min_cost)
            {
                // Giai TSP voi dinh ke tiep
                solveTSP(G, n, tempCost, level + 1);
            }
            // Bo danh dau dinh i
            visited[i] = false;
        }
    }
}
```

Giải thuật của hàm find_next:
- Vòng lặp qua tất cả các đỉnh (`for (int i = 0; i < n; i++)`).
- Kiểm tra xem đỉnh chưa được thăm và có đường đi

- Đánh dấu đỉnh đã được thăm (`visited[i] = true`).
- Cập nhật lộ trình hiện tại (`curent_travel[level] = i`).
- Tính toán chi phí tạm thời
(`int tempCost = currentCost + G[curent_travel[level - 1]][i]`).
- Nếu chi phí tạm thời nhỏ hơn chi phí nhỏ nhất hiện tại cộng với ước lượng chi phí còn lại, tiếp tục tìm kiếm
(`if (tempCost + (n - level) * min_trong_so < min_cost)`):
- Gọi hàm `solveTSP` để tiếp tục tìm kiếm với chi phí tạm thời và mức tiếp theo (`solveTSP(G, n, tempCost, level + 1)`).
- Đánh dấu lại đỉnh chưa được thăm (`visited[i] = false`).

3.4 Hàm solveTSP

Hàm `solveTSP` có chức năng tìm kiếm lời giải tối ưu cho bài toán TSP, đây là hàm cốt lõi của thuật toán nhánh và cận, được dùng để tìm đường đi tối ưu nhất trong TSP.

```
void solveTSP(const int G[20][20], int n, int currentCost, int
    level)
{
    if (level == n)
    {
        if (G[curent_travel[level - 1]][curent_travel[0]] == 0)
        {
            return;
        }
        int total_cost = currentCost + G[curent_travel[level -
           1]][curent_travel[0]];
        if (total_cost < min_cost)
        {
            min_cost = total_cost;
            memcpy(path_toi_uu, curent_travel, n * sizeof(int));
        }
        return;
    }
    find_next(G, n, currentCost, level);
}
```

Trong hàm `solveTSP`, kiểm tra nếu đã duyệt hết tất cả các đỉnh (`level == n`), và nếu lộ trình kết thúc tại đỉnh xuất phát thì sẽ cập nhật lộ trình tối ưu. Nếu chưa, thì tiếp tục tìm các đỉnh tiếp theo có thể đi từ đỉnh hiện tại bằng cách gọi hàm `find_next`.

3.5 Hàm Traveling

Hàm `Traveling` là hàm chính của chương trình, có chức năng tìm lộ trình tối ưu nhất cho bài toán TSP.

```
string Traveling(const int G[20][20], int n, char startVertex) {
    memset(visited, false, sizeof(visited));
    int startIndex = startVertex - 'A';
```

```
    curent_travel[0] = startIndex;
    visited[startIndex] = true;

    min_cost = 1e9;
    min_trong_so = 1e9;

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            // phai dam bao 2 o khong bi trung nhau va
            // cost < min_cost
            if (G[i][j] != 0 && G[i][j] < min_trong_so)
            {
                min_trong_so = G[i][j];
            }
        }
    }

    solveTSP(G, n, 0, 1);
    return constructPathString(n);
}
```

Trong các hàm trên, đã áp dụng giải thuật nhánh và cận để tìm lộ trình tối ưu nhất cho bài toán TSP.

Ngoài ra, chương trình còn sử dụng các biến toàn cục:

```
// Mang luu tru cac dinh trong lo trinh hien tai va lo trinh toi
   uu
int curent_travel[20], path_toi_uu[20];

// Mang danh dau cac dinh da duoc tham
bool visited[20];

// Bien luu tru chi phi nho nhat va trong so nho nhat
int min_cost, min_trong_so;
```

Các biến toàn cục này bao gồm:
`curent_travel`: Mảng lưu trữ các đỉnh trong lộ trình hiện tại.
`path_toi_uu`: Mảng lưu trữ lộ trình tối ưu tìm được.
`visited`: Mảng đánh dấu các đỉnh đã được thăm trong quá trình tìm kiếm.
`min_cost`: Biến lưu trữ chi phí nhỏ nhất tìm được.
`min_trong_so`: Biến lưu trữ trọng số nhỏ nhất giữa các đỉnh.

4 Đánh giá mức độ tối ưu của giải pháp

Đánh giá mức độ tối ưu của giải pháp nhánh và cận (Branch and Bound) kết hợp với các kỹ thuật tối ưu khác được sử dụng trong đoạn mã để giải quyết bài toán Traveling Salesman (TSP):

4.1 Kết quả trả về

- **Độ chính xác**: Chương trình này đã này đảm bảo tìm được lộ trình ngắn nhất bằng cách duyệt qua tất cả các khả năng có thể.

4.2 Độ phức tạp thời gian

- **Theo lí thuyết**: độ phức tạp thời gian của thuật toán nhánh và cận trong trường hợp xấu nhất vẫn là $O(n!)$, với n là số lượng thành phố. Tuy nhiên, kỹ thuật nhánh và cận đã loại bỏ đi những lộ trình không khả thi, giúp giảm đáng kể số lượng lộ trình cần kiểm tra so với việc kiểm tra toàn bộ $n!$ lộ trình.

4.3 Hiệu quả thực tế

- **Sử dụng chi phí ước lượng (lower bound)**: Việc sử dụng chi phí ước lượng nhỏ nhất (`min_trong_so`) giúp cắt bớt nhiều nhánh không cần thiết, do đó, tăng tốc độ tìm kiếm giải pháp.
- **Đánh dấu đỉnh đã thăm**: Việc sử dụng mảng `visited` để đánh dấu các đỉnh đã thăm giúp tránh quay lại thăm các đỉnh đó lần nữa, giảm số lượng tính toán không cần thiết.

4.4 Sử dụng bộ nhớ

- **Biến toàn cục**: Sử dụng các biến toàn cục như `curent_travel`, `path_toi_uu`, `visited`, `min_cost`, và `min_trong_so` giúp lưu trữ trạng thái và giảm chi phí truyền tham số, tuy nhiên, điều này cũng có thể gây ra những hạn chế về tính linh hoạt của chương trình.

4.5 Khả năng mở rộng

- **Giới hạn kích thước ma trận**: Giải pháp này được thiết kế cho ma trận tối đa 20x20, điều này có nghĩa là nó sẽ không hoạt động hiệu quả cho các bài toán TSP có kích thước lớn hơn mà không cần thay đổi đáng kể thuật toán.

4.6 Đánh giá tổng thể

Phương pháp nhánh và cận kết hợp với các kỹ thuật tối ưu khác được sử dụng trong đoạn mã là một giải pháp mạnh mẽ và chính xác cho bài toán TSP với quy mô nhỏ và trung bình. Mặc dù có thể không phải là giải pháp tối ưu nhất cho các bài toán TSP rất lớn, nhưng nó cung cấp một cách tiếp cận thực tiễn và hiệu quả. Với các cải tiến như kỹ thuật heuristic hoặc giải pháp xấp xỉ, phương pháp này có thể được mở rộng và tối ưu hóa hơn nữa để xử lý các bài toán lớn hơn.

5 Lời kết

Thông qua những phân tích về hướng tiếp cận với bài toán Traveling Salesman đã trình bày ở trên, em xin cảm ơn các quý thầy đã dành thời gian ra để đọc bài báo cáo. Bài tập lớn lần này đã giúp củng cố kiến thức của môn học và giúp em có cơ hội được tiếp cận với một mảng kiến thức mới trong lập trình là quy hoạch động, đồng thời cung cấp các giải thuật quan trọng là nền tảng của ngành học trong tương lai.

Và xin được gửi lời cảm ơn chân thành nhất đến thầy Mai Xuân Toàn, là người đã trực tiếp đứng lớp đồng thời là người giải đáp các thắc mắc trong suốt quá trình học cũng như của bài tập lớn trong quá trình học tập tại lớp.

Trong quá trình thực hiện bài tập lớn, không tránh khỏi những sai sót và chưa đảm bảo tối ưu nhất mã nguồn chương trình khi chạy, em xin ghi nhận mọi đóng góp của các thây để những bài báo cáo các môn học trong tương lai được hoàn thiện hơn.

Cuối lời, xin kính chúc các thầy có thật nhiều sức khỏe và thành công.

References

[Rosen] Rosen, *Discrete Mathematics And Its Applications*

[Cormen] Comen, *Introduction to Algorithms*
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest và Clifford Stein.