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

CẤU TRÚC RỜI RẠC

Bài tập lớn

BÀI TOÁN NGƯỜI BÁN HÀNG

Sinh viên : Nguyễn Hoàng Long 2311906

THÀNH PHỐ HỒ CHÍ MINH, THÁNG 6 2024

Mục lục

1 Giới thiệu

1.1 Đồ thị

Cấu trúc này bao gồm các đỉnh và các cạnh nối các đỉnh này lại với nhau. Đồ thị có nhiều dạng khác nhau, tùy thuộc vào các đặc điểm cụ thể:

- Đồ thị có hướng và đồ thị vô hướng: Đồ thị có hướng biểu thị các cạnh một chiều trong khi đồ thị vô hướng biểu thị các cạnh hai chiều.
- Đa đồ thị: Đồ thị cho phép nhiều cạnh kết nối với cùng một cặp đỉnh.
- Khuyên: Một số đồ thị cho phép các đỉnh tự kết nối với chính nó.

1.2 Ứng dụng

Đồ thị phục vụ biểu diễn những mô hình trong nhiều lĩnh vực khác nhau. Dưới đây là một số ứng dụng:

- Điều hướng đường đi: Dùng để xác định xem có thể di chuyển qua các con đường trong một thành phố tối ưu về các mặt.
- Tô màu bản đồ: Giúp tìm ra số lượng màu cần thiết để tô màu trên các vùng của bản đồ sao cho các vùng kề nhau không cùng màu với nhau.
- Thiết kế vi mạch: Đánh giá một bảng mạch được thiết kế trên mặt phẳng.
- Cấu trúc hóa học: Biểu diễn các cấu trúc hóa học khác nhau của các chất có cùng công thức hóa học.
- Kết nối mạng: Kiểm tra xem liệu giữa hai máy tính có được kết nối qua một liên kết không.
- Lịch trình và phân bố kênh: Hỗ trợ trong việc lên lịch trình và phân bố các kênh cho đài truyền hình.

Sự đa dạng của lý thuyết đồ thị làm cho nó trở thành công cụ quan trọng trong việc giải quyết và phân tích vấn đề.

2 Bài toán người bán hàng

Bài toán người bán hàng (được gọi tắt là TSP) đặt ra một vấn đề: "Có một người bán hàng cần đi qua n thành phố. Anh ta xuất phát từ một thành phố nào đó, đi qua các thành phố khác rồi quay trở lại thành phố ban đầu. Mỗi thành phố chỉ đến một lần, và đã biết trước khoảng cách giữa các thành phố. Đâu là đường đi ngắn nhất thỏa mãn yêu cầu trên?". Đây là một bài toán thuộc lớp NP-khó trong tối ưu hóa tổ hợp, quan trọng trong lý thuyết khoa học máy tính và nghiên cứu hoạt động.

3 Giải quyết bài toán người bán hàng bằng phương pháp quy hoạch động

Có nhiều giải pháp được đưa ra để giải quyết vấn đề trên và cho ra được các kết quả tối ưu dựa trên những đồ thị cho trước. Muốn tối ưu về mặt thời gian xử lí hay bộ nhớ cho chương trình, người thực hiện cần phải tìm ra các giải pháp phù hợp cho chương trình. Ví dụ như các thuật toán cho ra các kết quả chính xác nhưng có thời gian chạy phức tạp (với n là số đỉnh):

- Thuật toán vét cạn
- Thuật toán nhánh cận
- Quy hoạch động

Hoặc các giải thuật heuristic (tìm các lời giải tốt với thời gian thực hiện nhanh chóng):

- Thuật toán láng giềng gần nhất
- Thuật toán tìm cây khung tối tiểu (sử dụng thuật toán Christofides cho ra giá trị với sai số không quá 3/2)
- Thuật toán di truyền

Ở bài báo cáo này đã tìm hiểu về cách giải bài toán TSP bằng cách tiếp cận quy hoạch động (Dynamic Programming) bởi vì số đỉnh của đồ thị đề cho không quá 20 đỉnh, điều này dẫn đến độ phức tạp của thuật toán là $O(n^2 * 2^n)$), trong khi các thuật toán tìm kiếm chính xác khác như vét cạn là $O((n-1)/2!)$ với đồ thị sở hữu chu trình Hamilton.

3.1 Mã nguồn

Listing 3.1: Khai báo cấu trúc cho mảng ghi nhớ

```
struct node{
    int weight;
    bool visited;
    int pre_vertex;

    bool* visitedNode;

    node() {
        weight = INT_MAX;
        visited = false;
        pre_vertex = -1;
        visitedNode = NULL;
    }
    node(int size){
        weight = INT_MAX;
        visited = false;
        pre_vertex = -1;
        visitedNode = new bool[size];
        for (int i = 0; i < size; i++) visitedNode[i] = false;
    }
};
```

`weight` lưu trọng số của một cạnh.
`visited` lưu trạng thái của đỉnh được xét với `true` là đỉnh đã được đi qua và `false` cho trạng thái ngược lại.
`pre_vertex` lưu đỉnh trước đó.
`visitedNode` là mảng lưu các node với các giá trị đã cho ở trên.
Bên dưới là các hàm khởi tạo với các giá trị tương ứng cho node:

- Khởi đầu với giá trị trọng số là lớn nhất
- Đặt các giá trị đã đến khởi đầu là `false`
- Giá trị của đỉnh trước đó là -1.
- Cấp phát động cho mảng lưu các đỉnh đã thăm với kích thước tương ứng.

Khai báo class TSP để giải quyết bài toán:

Listing 3.2: Các biến private

```
private:
int graphSize;
int start;
int graph[20][20];
int* path;
bool done;
int min_cost = INT_MAX;

node** saveStatus;
```

Các biến trên lần lượt là: số đỉnh của đồ thị được cho bởi ma trận trọng số, giá trị đỉnh bắt đầu, ma trận đã cho trước, mảng lưu đường đi tối ưu, biến xác nhận đã hoàn thành giải thuật (tránh chạy lại giải thuật), giá trị nhỏ nhất được thiết lập là `INT_MAX`, và mảng hai chiều để ghi nhớ dành cho lập trình quy hoạch động cùng đệ quy.

Hàm khởi tạo và hàm hủy đối tượng được hiện thực, với mảng hai chiều `saveStatus` thuộc kiểu dữ liệu `node` như sau:

```
saveStatus = new node*[graphSize];
for (int i = 0; i < graphSize; i++){
    int k = 1;
    for (int j = 0; j < graphSize; j++) k = k * 2;
    saveStatus[i] = new node[k];
    for (int j = 0; j < k; j++) saveStatus[i][j] =
        node(graphSize);
}
```

Hàm `execute();` để thực hiện các thiết lập cho mảng lưu các đỉnh cũng như gọi hàm đệ quy `recursionTSP();`

```
void execute() {
    bool* visited = new bool[graphSize];
    for (int i = 0; i < graphSize; ++i) visited[i] = false;
    visited[start] = true;
```

```
5           min_cost = recursionTSP(start, visited);
6
7           int i = start, t = 0;
8           while(true) {
9               path[t++] = i;
10              int next_i = saveStatus[i][convertBit(visited)].pre_vertex;
11              if(next_i == -1) break;
12              visited[next_i] = true;
13              i = next_i;
14          }
15          path[t] = start;
16          done = true;
17          delete[] visited;
18      }
```

Khởi tạo mảng `visited` với số lượng phần tử bằng với số đỉnh của đồ thị và giá trị ban đầu là `false`. Đặt giá trị tại đỉnh xuất phát là `true`.
Hàm đệ quy trả về giá trị nhỏ nhất của hành trình sẽ được cụ thể ở phần sau.
Tạo biến `i` gán là đỉnh bắt đầu, một biến chạy trong vòng lặp `while` là `t = 0`. Khi vào trong hàm `while`, mảng `path` sẽ lưu các đỉnh được đưa vào, đồng thời biến đếm sẽ cộng lên 1 để lưu đỉnh tiếp theo.
Tạo biến `next_i` để tính đỉnh kế tiếp bằng cách gọi từ mảng lưu với `[i]` là giá trị đỉnh hiện tại đang xét.
`convertBit(visited)` dùng để chuyển đổi giá trị đúng hoặc sai của mảng đánh dấu đã thăm thành một số nguyên sử dụng bitmask (để sử dụng các thao tác với chuỗi bit trong thuật toán quy hoạch động).

```
1       int convertBit(bool* visited) {
2           int pos = 0;
3           int k = 1;
4           for (int i = 0; i < graphSize; ++i) {
5               if (visited[i]) pos = pos + k;
6               k = k * 2;
7           }
8           return pos;
9       }
```

Ta có hai biến `pos` và `k` được gán lần lượt là 0 và 1, tạo một vòng lặp để duyệt tất cả các phần tử trong mảng `visited`, nếu phần tử đang duyệt trả về giá trị đúng, thì biến `k` được thêm vào `pos`, và ngược lại thì `k` sẽ không được thêm vào, rồi tăng giá trị biến `k = k * 2;` tương ứng với các giá trị bit trong hệ nhị phân được dịch sang trái:

- k = 1 tương ứng 2^0 tương ứng ...0001
- k = 2 tương ứng 2^1 tương ứng ...0010
- k = 4 tương ứng 2^2 tương ứng ...0100
- k = 8 tương ứng 2^3 tương ứng ...1000
- k = ...

Ta có thể lưu được giá trị các đỉnh thông qua biểu diễn chuỗi bit, mỗi một vị trí `pos` (đã đưa cách biểu diễn về số nguyên) thể hiện các trạng thái chưa hoặc đã thăm của đỉnh, và việc truy cập lại các trạng thái đó có quy tắc hơn. Sau đó `pos` trả về biểu diễn số nguyên của mảng `visited`. Sau khi tìm được nơi lưu dành cho đỉnh `i` trong mảng `saveStatus`, ta gọi `pre_vertex` để lấy được giá trị đỉnh kế tiếp của đỉnh `i` cho việc gán lại đỉnh `i`, và vòng lặp sẽ kết thúc khi không còn đỉnh nào để thăm `next_i == -1`. Quá trình trên là quá trình xây dựng đường đi tối ưu. Và cuối cùng gán cho phần tử cuối cùng của mảng `path` là đỉnh xuất phát để hoàn thành chu trình.

```
1       int recursionTSP(int vertex, bool* visited) {
2           int pos = convertBit(visited);
3           if (checkVisited(visited) == true) return graph[vertex][start];
4           if (saveStatus[vertex][pos].visited == true) return
                saveStatus[vertex][pos].weight;
5
6           int min = INT_MAX;
7           int index = -1;
8           for (int j = 0; j < graphSize; j++){
9               if (visited[j] == true) continue;
10              visited[j] = true;
11              int temp = graph[vertex][j] + recursionTSP(j, visited);
12              visited[j] = false;
13              if (temp < min){
14                  min = temp;
```

```
15                  index = j;
16              }
17          }
```

Bên trong hàm xử lí đã gọi ra hàm đệ quy trên để tìm ra đường đi tốt nhất cho bài toán. Hàm trên đưa dữ liệu đầu vào là đỉnh được gọi và mảng lưu trạng thái đã ghé thăm.
Đầu tiên lấy giá trị `pos` sau khi đã chuyển từ mảng `visited` bằng hàm `convertBit()`. Hàm `checkVisited()` kiểm tra tất cả các phần tử đã được đi qua hết hay chưa để trả về giá trị trọng số đỉnh đang xét và đỉnh bắt đầu để hoàn thành chu trình (điều kiện dừng của đệ quy), hoặc mảng lưu của đỉnh xét đến vị trí `pos` mà đã được thăm (nghĩa là đã trải qua tính toán) thì trả về giá trị trọng số đã tính. Nếu đỉnh chưa được thăm thì ta duyệt tất cả các phần tử trong mảng `visited`, nếu đỉnh nào đã được thăm thì bỏ qua, ngược lại, đánh dấu điểm đó đã được thăm, cho biến `temp` là tổng trọng số với giá trị nhỏ nhất của các biến chưa được thăm và gọi lại hàm đệ quy với đỉnh tiếp theo được xét nằm trong vòng lặp.
Sau khi kết thúc đệ quy, thiết lập lại trạng thái đỉnh `j` đang xét trong vòng lặp thành `false` để sử dụng lại cho các đỉnh `vertex` kế tiếp.
Cập nhật chi phí nhỏ nhất khi giá trị `temp < min`.
Đặt lại các trạng thái cho các mảng ghi nhớ quy hoạch động:

- `saveStatus[vertex][pos].visited = true;` để chỉ ra tại đỉnh `vertex` xét đến vị trí pos đã được tính toán.
- `saveStatus[vertex][pos].weight = min;` biểu diễn giá trị nhỏ nhất của chu trình từ đỉnh hiện tại (`vertex`) về đỉnh ban đầu.
- `saveStatus[vertex][pos].pre_vertex = index;` lưu lại đỉnh trước đó đi đến đihr hiện tại với `index = j` là đỉnh có được trọng số nhỏ hơn qua tính toán.

Cuối cùng đưa tất cả giá trị của mảng vừa tìm được vào mảng lưu giá trị đã đến `visitedNode` của mảng `visited` để đặt lại tất cả các phần tử của `visited` về `false` ban đầu. Trả về giá trị `min` để tính toán đệ quy.

```
1       int* getPath() {
2           if(done == false) execute();
3           return path;
4       }
```

Sau khi tính toán xong, `path` đã chứa toàn bộ chu trình của bài toán dưới dạng số nguyên với đỉnh `A` tương ứng với `0`.
Khai báo một đối tượng thuộc `class` `TSP` và gọi ra hàm `getPath()`, sử dụng vòng lặp `for` với số lần lặp bằng số đỉnh của đồ thị thêm 1 để trả về chuỗi `string` theo yêu cầu đề bài.

3.2 Vấn đề khi thực hiện

Ở mã nguồn trên, về việc đồ thị đề cho, nếu từ một đỉnh này đến một đỉnh khác mà không tồn tại đường đi, sẽ được biểu thị là 0. Từ đó dẫn đến việc xét trạng thái giá trị nhỏ nhất gặp khó, nếu thêm lệnh bỏ qua thì sẽ có khả năng đường đi thiếu các đỉnh của đồ thị. Bởi vì thế, khi lấy dữ liệu từ đồ thị cho sẵn:

```
TSP(int weight_matrix[20][20], int size, char start_char) :
    graphSize(size) {
    start = start_char - 'A';
    done = false;
    for (int i = 0; i < graphSize; i++){
        for (int j = 0; j < graphSize; j++) {
            graph[i][j] = weight_matrix[i][j];
            if(graph[i][j] == 0) graph[i][j] = 9999999;
            }
    }
    saveStatus = new node*[graphSize];
    for (int i = 0; i < graphSize; i++){
        int k = 1;
        for (int j = 0; j < graphSize; j++) k = k * 2;
        saveStatus[i] = new node[k];
        for (int j = 0; j < k; j++) saveStatus[i][j] =
            node(graphSize);
    }
    path = new int[graphSize + 1];
}
```

Nếu giá trị trọng số là 0, thì sửa đổi thành giá trị rất lớn để giá trị min có thể duyệt qua mà không cần phải bỏ đỉnh (điều này hạn chế với các đồ thị có trọng số lớn tương đương).

Về cấu trúc `struct node` làm cho việc thực hiện chương trình tốn khá nhiều bộ nhớ với khởi tạo mảng hai chiều, và bên trong mỗi phần tử lại chứa các mảng cấp phát động khác.

Tài liệu tham khảo

[1] David R. Hill Bernard Kolman. *INTRODUCTORY LINEAR ALGEBRA*. Person Prentice Hall, 8 edition.

[2] William Fiset. Algorithms. tham khảo.

[3] Stackoverflow Geeksforgeeks, Github. Dynamic programming. Technical report.

[4] Quang Nhat Nguyen. Travelling salesman problem and bellman-held-karp algorithm. Technical report, Nagoya University, 2020.

[5] Hoàng Kiếm Nguyễn Thanh Hùng. Áp dụng giải thuật di truyền với thông tinthoongs kê xác xuất giải quyết bài toán tìm chu trình hamilton. *Tạp chí phát triển KH-CN*, 7(12):5–11, 2004.