TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

KHOA KHOA HỌC MÁY TÍNH

BÀI TẬP VỀ NHÀ NHÓM 7

MÔN: PHÂN TÍCH VÀ THIẾT KẾ THUẬT TOÁN

Nhóm 9

Sinh viên thực hiện:

Họ và tên	MSSV
Bùi Ngọc Thiên Thanh	23521436
Nguyễn Thái Sơn	23521356

Thành phố Hồ Chí Minh, 2024

Mục lục

1 Bài toán Set Cover

Bài toán Set Cover là một bài toán trong lý thuyết đồ thị và tối ưu hóa tổ hợp. Mục tiêu của bài toán là tìm ra một tập con tối thiểu của các tập con từ S sao cho tất cả các phần tử trong U đều được bao phủ, tức là mỗi phần tử của U phải xuất hiện ít nhất một lần trong các tập con được chọn.

1.1 Mô tả bài toán

Cho:

- $U = \{u_1, u_2, \ldots, u_n\}$ là tập hợp các phần tử cần được bao phủ.
- $S = \{S_1, S_2, \ldots, S_m\}$ là tập hợp các tập con của U, mỗi tập con $S_i \subseteq U$.

Mục tiêu của bài toán là chọn một số ít các tập con từ S sao cho mỗi phần tử trong U xuất hiện ít nhất một lần trong các tập con được chọn, đồng thời tối thiểu hóa số lượng tập con được chọn. Điều này có thể diễn đạt bằng công thức sau:

$$\min |S'| \quad \text{với} \quad \bigcup_{S_i \in S'} S_i = U$$

Trong đó $S' \subseteq S$ là tập con của S cần tìm.

1.2 Giải thuật tham lam (Greedy Algorithm)

Bài toán Set Cover là một bài toán NP-Complete, tức là không có thuật toán giải quyết tối ưu nhanh chóng cho các bài toán có quy mô lớn. Tuy nhiên, thuật toán tham lam có thể đưa ra một giải pháp gần tối ưu. Cách tiếp cận của thuật toán tham lam là lựa chọn các tập con sao cho mỗi lần chọn sẽ bao phủ càng nhiều phần tử chưa được bao phủ càng tốt.

1.2.1 Các bước của thuật toán

1. Khởi tạo một tập rỗng $S' = \{\}$ và một tập $U' = U$ chứa tất cả các phần tử của U.
2. Lặp cho đến khi tất cả các phần tử trong U đều được bao phủ:
   - Chọn tập con $S_i \in S$ sao cho S_i bao phủ được nhiều phần tử trong U' nhất.
   - Cập nhật U' bằng cách loại bỏ các phần tử đã được bao phủ bởi S_i.
   - Thêm S_i vào S'.

1.2.2 Mã nguồn Python

Dưới đây là mã Python cho thuật toán tham lam để giải bài toán Set Cover:

Listing 1: Mã Python ví dụ

```
def set_cover(U, S):
    uncovered = set(U)
    selected_sets = []

    while uncovered:
        best_set = None
        covered_elements = set()

        for s in S:
            uncovered_in_s = uncovered.intersection(s)
            if len(uncovered_in_s) > len(covered_elements):
                best_set = s
                covered_elements = uncovered_in_s

        uncovered -= covered_elements
        selected_sets.append(best_set)

    return selected_sets

# Example
```

```
U = {1, 2, 3, 4, 5, 6}
S = [{1, 2, 3}, {2, 4}, {3, 5}, {4, 6}, {5, 6}]

result = set_cover(U, S)
print("Cac tap con duoc chon:", result)
```

1.3 Đánh giá độ phức tạp

Độ phức tạp của thuật toán tham lam là $O(m \cdot n)$, trong đó:

- m là số lượng tập con trong S.
- n là số phần tử trong tập U.

Trong mỗi vòng lặp, thuật toán cần tính toán giao của tất cả các tập con với tập chưa bao phủ. Do đó, độ phức tạp tính toán của thuật toán là khá lớn khi m và n tăng.

2 Bài toán Travelling Salesman Problem

Bài toán TSP (Bài toán người du lịch) là một bài toán tối ưu hóa cổ điển trong lý thuyết đồ thị và tổ hợp. Mục tiêu của bài toán này là tìm một chu trình Hamilton trong đồ thị sao cho:

- Mỗi đỉnh được thăm đúng một lần.
- Tổng chi phí (hoặc tổng quãng đường) đi qua tất cả các đỉnh là nhỏ nhất.

2.1 Định nghĩa chính thức

Cho một đồ thị $G = (V, E)$, trong đó:

- $V = \{v_1, v_2, \ldots, v_n\}$ là tập hợp các đỉnh, mỗi đỉnh tương ứng với một thành phố.
- E là tập hợp các cạnh nối các đỉnh.
- Mỗi cạnh $(v_i, v_j) \in E$ có trọng số $c(v_i, v_j)$, có thể là chi phí, thời gian hoặc khoảng cách di chuyển giữa các thành phố v_i và v_j.

Mục tiêu là tìm một chu trình Hamiltonian sao cho tổng trọng số của các cạnh trong chu trình này là nhỏ nhất.

2.2 Phương pháp giải quyết

Bài toán TSP là một bài toán NP-Hard, nên không thể giải quyết được chính xác với thời gian ngắn cho các đồ thị có nhiều đỉnh. Tuy nhiên, chúng ta có thể sử dụng các thuật toán gần đúng để tìm ra các nghiệm gần tối ưu.

2.2.1 Giải thuật gần đúng 1: Thuật toán gần nhất (Nearest Neighbor)

Thuật toán gần nhất là một thuật toán tham lam, trong đó mỗi lần di chuyển đến đỉnh gần nhất chưa được thăm.

Mô tả thuật toán:

- Bắt đầu từ một thành phố bất kỳ.
- Tìm thành phố gần nhất chưa được thăm và di chuyển đến đó.
- Lặp lại cho đến khi thăm tất cả các thành phố, sau đó quay lại thành phố ban đầu.

2.2.2 Mã C++ thuật toán Nearest Neighbor:

Listing 2: Thuật toán Nearest Neighbor

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

int nearestNeighbor(vector<vector<int>>& dist, int n) {
    vector<bool> visited(n, false);
    int totalCost = 0;
    int currentCity = 0;
    visited[currentCity] = true;
```

```
    int visitedCount = 1;

    while (visitedCount < n) {
        int nextCity = -1;
        int minDist = INT_MAX;


        for (int i = 0; i < n; i++) {
            if (!visited[i] && dist[currentCity][i] < minDist) {
                minDist = dist[currentCity][i];
                nextCity = i;
            }
        }

        visited[nextCity] = true;
        totalCost += minDist;
        currentCity = nextCity;
        visitedCount++;
    }


    totalCost += dist[currentCity][0];

    return totalCost;
}

int main() {
    int n = 4;
    vector<vector<int>> dist = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };

    int result = nearestNeighbor(dist, n);
    cout << "Chi phi cua duong di: " << result << endl;

```

```
49    return 0;
50}
```

2.2.3 Giải thuật gần đúng 2: Thuật toán 2-opt

Thuật toán 2-opt là một thuật toán cải tiến cho thuật toán gần nhất. Nó thực hiện việc đảo ngược một đoạn của đường đi để giảm tổng chi phí.

Mô tả thuật toán:

- Chạy thuật toán nearest neighbor để có một giải pháp ban đầu.
- Sau đó, kiểm tra tất cả các cặp đoạn (i, j) trong chu trình và thực hiện đảo ngược đoạn giữa chúng nếu điều này làm giảm tổng chi phí.
- Lặp lại quá trình này cho đến khi không thể cải thiện thêm nữa.

2.2.4 Mã C++ thuật toán 2-opt:

Listing 3: Thuật toán 2-opt

```
1#include <iostream>
2#include <vector>
3#include <cmath>
4#include <climits>
5using namespace std;
6
7int calculateCost(const vector<vector<int>>& dist, const
    vector<int>& tour) {
8    int totalCost = 0;
9    for (int i = 0; i < tour.size() - 1; i++) {
10       totalCost += dist[tour[i]][tour[i+1]];
11   }
12   totalCost += dist[tour[tour.size()-1]][tour[0]];
13   return totalCost;
14}
15
```

```cpp
void twoOptSwap(vector<int>& tour, int i, int j) {
    while (i < j) {
        swap(tour[i], tour[j]);
        i++;
        j--;
    }
}

int twoOpt(vector<vector<int>>& dist, int n) {
    vector<int> tour(n);
    for (int i = 0; i < n; i++) {
        tour[i] = i;
    }

    int bestCost = calculateCost(dist, tour);

    bool improved = true;
    while (improved) {
        improved = false;
        for (int i = 1; i < n - 2; i++) {
            for (int j = i + 1; j < n - 1; j++) {
                vector<int> newTour = tour;
                twoOptSwap(newTour, i, j);
                int newCost = calculateCost(dist, newTour);
                if (newCost < bestCost) {
                    tour = newTour;
                    bestCost = newCost;
                    improved = true;
                }
            }
        }
    }
    return bestCost;
}

int main() {
    int n = 4;
    vector<vector<int>> dist = {
```

```
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };

    int result = twoOpt(dist, n);
    cout << "Chi phi sau khi 2-opt: " << result << endl;

    return 0;
}
```

2.3 Kết luận

Trong hai phương pháp gần đúng trên, thuật toán Nearest Neighbor đơn giản và nhanh nhưng có thể không cho kết quả tối ưu. Thuật toán 2-opt cải tiến giải pháp ban đầu và có thể giảm được chi phí tổng, tuy nhiên thời gian chạy lâu hơn vì cần phải thử nhiều cặp đoạn khác nhau.