## TRƯỜNG ĐẠI HỌC NHA TRANG KHOA CÔNG NGHỆ THÔNG TIN

------



# BÁO CÁO MÔN HỌC TOÁN RỜI RẠC

## NỘI DUNG:

**3 BÀI TOÁN TRONG TOÁN RỜI RẠC**

**GVHD: ThS. Nguyễn Hải Triều SVTH: Cao Nguyễn Quốc Lâm**

**MSSV: 63132195**

**Lớp: 63.CNTT-4**

**I. Kỹ thuật quy hoạch động (Dynamic Programming):**

a. Khái niệm: Kỹ thuật này giải quyết bài toán bằng cách chia bài toán ban đầu thành các bài toán nhỏ hơn, giải quyết và lưu trữ kết quả của các bài toán nhỏ này để tái sử dụng khi cần. Phương pháp này thích hợp cho các bài toán con của bài toán gốc có tính chất chồng chéo (overlapping subproblems), nghĩa là các bài toán con có thể chia sẻ các kết quả lẫn nhau.

b. Ưu điểm:

- Hiệu quả với các bài toán có cấu trúc lặp lại, tránh tính toán lại các bài toán con đã giải quyết.

- Tối ưu hóa thời gian thực thi bài toán, giảm độ phức tạp của bài toán gốc.

c. Nhược điểm:

- Tiêu tốn nhiều bộ nhớ để lưu trữ kết quả các bài toán con.

- Phải xác định các bài toán con và quy luật tái sử dụng kết quả.

**II. Kỹ thuật nhánh cận (Branch and Bound):**

a. Khái niệm: Kỹ thuật này giải quyết bằng cách tạo ra các nhánh con cho các lựa chọn khả thi và kiểm tra những giới hạn (bounds) của từng nhánh. Các giới hạn này được sử dụng để cắt tỉa các nhánh không cần thiết, giúp tăng tốc độ tìm kiếm và giảm không gian tìm kiếm giải pháp.

b. Ưu điểm:

- Giúp tìm kiếm giải pháp tốt nhất theo chiều sâu, có thể dừng khi đã tìm thấy giải pháp tối ưu.

- Hiệu quả với những bài toán khác nhau với không gian tìm kiếm khổng lồ và có thể sắp xếp giá trị giới hạn (bounds).

c. Nhược điểm:

- Tính chất cắt tỉa và giới hạn phụ thuộc vào cách xây dựng và tính chất của bài toán, không áp dụng được cho mọi bài toán.

- Cần xác định hàm chi phí và giá trị giới hạn chính xác, tránh dẫn đến sai sót trong quá trình tính toán giới hạn.

**III. So sánh hai kỹ thuật nhánh cận và quy hoạch động:**

**1. Ưu điểm:**

- Hiệu quả với bài toán có tính con tối ưu và con chồng chất.

- Tối ưu hóa hiệu năng bằng cách lưu trữ kết quả tính toán và tái sử dụng chúng.

- Xử lý các bài toán tối ưu hóa có nhiều kết quả phụ thuộc vào các bài toán con.

- Hiệu quả với dạng toán có không gian tìm kiếm vô cùng lớn.

- Tìm được giải pháp gần đúng trong thời gian tương đối ngắn.

- Loại bỏ các nhánh không tiềm năng từ đầu, giảm số lượng giải pháp không cần thiết.

**2. Nhược điểm:**

*a. Nhược điểm của quy hoạch động:*

- Yêu cầu của bài toán có tính chất con tối ưu và con chồng chất để sử dụng quy hoạch động.

- Sử dụng không gian bộ nhớ phụ thuộc vào số lượng bài toán con cần phải lưu trữ, có thể tăng đáng kể nếu bài toán có quá nhiều bài toán con.

*b. Nhược điểm của nhánh cận:*

- Cần xây dựng hàm đánh giá trước để đánh giá chất lượng của giải pháp gần đúng.

- Khó tìm được giải pháp tối ưu chính xác.

- Khi không có hàm đánh giá tốt, kết quả có thể không chính xác và chưa đáng tin cậy.

**IV. Bài toán “Người bán hàng”:**

**1. Lý thuyết:**

Bài toán yêu cầu tìm chu trình Hamilton ngắn nhất trên đồ thị đầy đủ n đỉnh, trong đó mỗi đỉnh phải được đi qua đúng một lần và cuối cùng quay lại đỉnh xuất phát. Mỗi cạnh trong đồ thị có một trọng số và mục tiêu của bài toán là tìm chu trình Hamilton có tổng trọng số là nhỏ nhất.

**2. Bài toán “Người bán hàng” áp dụng kỹ thuật quy hoạch động:**

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

const int INF = 10000000;

// Hàm quy hoạch động TSP

int tsp(vector<vector<int>>& graph, vector<vector<int>>& dp, int mask, int pos) {

    int n = graph.size();

int i;

    if (mask == (1 << n) - 1) {

        return graph[pos][0];

    }

    if (dp[mask][pos] != -1) {

        return dp[mask][pos];

    }

    int ans = INF;

    for (i = 0; i < n; ++i) {

        if ((mask & (1 << i)) == 0) {

            ans = min(ans, graph[pos][i] + tsp(graph, dp, mask | (1 << i), i));

        }

    }

    return dp[mask][pos] = ans;

}

int GiaiTSP(vector<vector<int>>& graph) {

    int n = graph.size();

    vector<vector<int>> dp(1 << n, vector<int>(n, -1));

    return tsp(graph, dp, 1, 0);

}

int main() {

        int n;

    cout << "Nhap so luong thanh pho: ";

    cin >> n;

    if (n <= 0) {

        cout << "So luong thanh pho phai lon hon 0." << endl;

        return 1;

    }

    vector<vector<int>> graph(n, vector<int>(n, 0));

    cout << "Nhap khoang cach (nhap " << INF << " neu khong co duong di):" << endl;

    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < n; ++j) {

            cin >> graph[i][j];

        }

    }

    cout << "Khoang cach min can di qua all thanh pho: " << GiaiTSP(graph) << endl;

    return 0;}

- Hàm tsp: Đây là hàm quy hoạch động thực hiện tính toán đường ngắn nhất từ một thành phố pos đến all thành phố còn lại qua một bitmask mask. Bitmask mask dùng để đánh dấu những thành phố đã được thăm qua và tránh việc lặp lại trong quá trình tính toán. Nếu mask bằng (1 << n) - 1, tức là đã đi qua all thành phố, ta sẽ quay trở về thành phố xuất phát và trả về khoảng cách từ thành phố hiện tại tới thành phố xuất phát.

- Hàm GiaiTSP: Ta khởi tạo một bảng dp với giá trị ban đầu là -1 để lưu trữ kết quả tính toán của các bài toán con. Hàm này sẽ gọi hàm tsp để tính toán đường đi ngắn nhất từ thành phố xuất phát ban đầu tới all thành phố còn lại.

- Trong hàm main, người dùng nhập số lượng thành phố n. Nếu n <= 0, sẽ in ra thông báo lỗi. Sau đó, người dùng nhập khoảng cách giữa các thành phố và lưu vào ma trận graph. Cuối cùng, chương trình gọi hàm GiaiTSP để tìm khoảng cách ngắn nhất cần đi qua all thành phố, in kết quả.

**3. Bài toán “Người bán hàng” áp dụng kỹ thuật nhánh cận:**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

const int INF = 1000000000; // Giá trị vô cùng (đại diện cho khoảng cách vô hạn)

// Hàm tính toán chi phí của tuyến đường từ thành phố hiện tại tới thành phố tiếp theo

int TTCP(vector<vector<int>>& graph, int src, int dest) {

    if (src >= 0 && src < graph.size() && dest >= 0 && dest < graph.size()) {

        return graph[src][dest];

    }

    return INF;

}

// Hàm nhánh cận giải bài toán người bán hàng

void NCBH(vector<vector<int>>& graph, vector<int>& path, vector<bool>& visited, int currentCost, int& minCost, int currentVertex) {

int i;

    int n = graph.size();

    if (currentVertex == n) {

        // Nếu đã đi qua all thành phố, cập nhật giá trị chi phí nhỏ nhất

        minCost = min(minCost, currentCost + TTCP(graph, path[currentVertex - 1], path[0]));

        return;

    }

    for (i = 0; i < n; ++i) {

        if (!visited[i]) {

            int temp = currentCost + TTCP(graph, path[currentVertex - 1], i);

            if (temp < minCost) {

                path[currentVertex] = i;

                visited[i] = true;

                NCBH(graph, path, visited, temp, minCost, currentVertex + 1);

                visited[i] = false;

            }

        }

    }

}

// Hàm giải bài toán người bán hàng bằng nhánh cận

int TSB(vector<vector<int>>& graph) {

    int n = graph.size();

    vector<int> path(n);

    vector<bool> visited(n, false);

    int currentCost = 0;

    int minCost = INF;

    // Bắt đầu từ thành phố 0

    path[0] = 0;

    visited[0] = true;

    NCBH(graph, path, visited, currentCost, minCost, 1);

    return minCost;

}

int main() {

    int n;

int i;

    cout << "Nhap so luong thanh pho: ";

    cin >> n;

    if (n <= 0) {

        cout << "So luong thanh pho phai lon hon 0." << endl;

        return 1;

    }

    vector<vector<int>> graph(n, vector<int>(n, 0));

    cout << "Nhap khoang cach (nhap " << INF << " neu khong co duong di):" << endl;

    for (i = 0; i < n; ++i) {

        for (int j = 0; j < n; ++j) {

            cin >> graph[i][j];

        }

    }

    cout << "Khoang cach ngan nhat di qua all thanh pho: " << TSB(graph) << endl;

    return 0;

}

- Hàm TTCP: tính tổng chi phí của tuyến đường đi qua các thành phố được lưu trong một vector path. Khoảng cách hai thành phố được lưu trong ma trận graph.

- Hàm NCBH: Hàm này dùng phương pháp đệ quy để thử tất cả các tuyến đường đi qua các thành phố chưa đi qua, loại bỏ các tuyến đường không tiềm năng để cải thiện hiệu năng. Nó duyệt qua các thành phố chưa đi qua và thêm các thành phố này vào tuyến đường đi. Nếu tuyến đường đã đi qua all thành phố, nó tính tổng chi phí và cập nhật minCost nếu tìm thấy tuyến đường tốt hơn. Hàm này sử dụng nhánh cận bằng cách kiểm tra nếu tổng chi phí hiện tại vượt quá minCost thì không tiếp tục thực hiện nhánh này.

- Hàm TSB: Hàm này khởi tạo các biến cần thiết và gọi hàm NCBH để tìm tuyến đường ngắn nhất.

- Trong hàm main, người dùng nhập số lượng thành phố n. Nếu n <= 0, sẽ in ra thông báo lỗi. Sau đó, người dùng nhập khoảng cách giữa các thành phố và lưu vào ma trận graph. Cuối cùng, chương trình gọi hàm TSB để tìm khoảng cách ngắn nhất cần đi qua all thành phố và in kết quả.

**V. Bài toán “xếp ba lô 1, 2”:**

**1. Lý thuyết:**

Bài toán “xếp ba lô” là một bài toán trong đó người dùng cần chọn một số vật phẩm từ một tập hợp các vật phẩm có sẵn để xếp vào ba lô. Đích đến của bài toán là tối ưu hóa giá trị tổng của các vật phẩm được chọn, trong khi đảm bảo tổng trọng lượng của ba lô không vượt quá trọng lượng tối đa.

**2. Bài toán “xếp ba lô” áp dụng quy hoạch động:**

#include <iostream>

#include <vector>

using namespace std;

int GiaiBLQH(int W, vector<int>& cannang, vector<int>& values) {

    int n = cannang.size();

    vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));

    for (int i = 1; i <= n; ++i) {

        for (int w = 1; w <= W; ++w) {

            if (cannang[i - 1] <= w) {

                dp[i][w] = max(dp[i - 1][w], values[i - 1] + dp[i - 1][w - cannang[i - 1]]);

            } else {

                dp[i][w] = dp[i - 1][w];

            }

        }

    }

    return dp[n][W];

}

int main() {

    int W, n;

    cout << "Nhap suc chua cua ba lo: ";

    cin >> W;

    cout << "Nhap so luong vat pham: ";

    cin >> n;

    vector<int> cannang(n);

    vector<int> values(n);

    cout << "Nhap trong luong va gia tri cua cac vat pham:" << endl;

    for (int i = 0; i < n; ++i) {

        cin >> cannang[i] >> values[i];

    }

    cout << "Gia tri max: " << GiaiBLQH(W, cannang, values) << endl;

    return 0;

}

- Hàm GiaiBLQH: Ta sử dụng một ma trận dp có kích thước (n + 1) x (W + 1) để lưu trữ kết quả tính toán. Với mỗi vật phẩm thứ i và trọng lượng w, ta tính dp[i][w] là giá trị lớn nhất có thể đạt được bằng cách chọn i vật phẩm trong số n vật phẩm có sẵn và trọng lượng tối đa là w.

- Hàm main: Trong hàm main, người dùng nhập trọng lượng tối đa của ba lô (W) và số lượng vật phẩm (n). Sau đó, người dùng nhập trọng lượng và giá trị của từng vật phẩm. Cuối cùng, gọi hàm GiaiBLQH để giải bài toán và in ra màn hình.

**3. Bài toán “xếp ba lô” vận dụng kỹ thuật nhánh cận:**

\* Code bài toán “xếp ba lô” bằng phương pháp nhánh cận:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Item {

    int weight;

    int value;

};

bool compareItems(Item a, Item b) {

    return (a.value / a.weight) > (b.value / b.weight);

}

// Hàm tính

int TT(vector<Item>& selectedItems) {

    int totalValue = 0;

    for (const auto& item : selectedItems) {

        totalValue += item.value;

    }

    return totalValue;

}

void NCBL(int W, vector<Item>& items, int currentValue, int currentWeight, int& maxValue, vector<Item>& selectedItems, int index) {

    if (index >= items.size()) {

        maxValue = max(maxValue, currentValue);

        return;

    }

    if (currentWeight + items[index].weight <= W) {

        selectedItems.push\_back(items[index]);

        NCBL(W, items, currentValue + items[index].value, currentWeight + items[index].weight, maxValue, selectedItems, index + 1);

        selectedItems.pop\_back();

    }

    NCBL(W, items, currentValue, currentWeight, maxValue, selectedItems, index + 1);

}

// Hàm giải bài toán xếp ba lô bằng phương pháp nhánh cận

int GiaiBLNC(int W, vector<Item>& items) {

    int maxValue = 0;

    vector<Item> selectedItems;

    NCBL(W, items, 0, 0, maxValue, selectedItems, 0);

    return maxValue;

}

int main() {

    int W, n;

    cout << "Nhap suc chua cua ba lo: ";

    cin >> W;

    cout << "Nhap so luong vat pham: ";

    cin >> n;

    vector<Item> items(n);

    cout << "Nhap trong luong va gia tri cua cac vat pham:" << endl;

    for (int i = 0; i < n; ++i) {

        cin >> items[i].weight >> items[i].value;

    }

    sort(items.begin(), items.end(), compareItems);

    cout << "Gia tri max: " << GiaiBLNC(W, items) << endl;

    return 0;

}

- Hàm compareItems: Đây là một hàm so sánh được sử dụng trong hàm main để sắp xếp các vật phẩm theo giá trị trên đơn vị trọng lượng giảm dần.

- Hàm TT: Hàm này tính tổng giá trị của các vật phẩm đã được chọn trong ba lô.

- Hàm NCBL: Hàm này sử dụng đệ quy để thử tất cả các trường hợp có thể có. Chọn từng vật phẩm và xem xét xem có chọn vật phẩm này hay không, sau đó đi tiếp đến vật phẩm tiếp theo. Trong quá trình duyệt, ta kiểm tra xem giá trị của vật phẩm hiện tại có lớn hơn giá trị lớn nhất đã tìm thấy hay không, nếu có, thì cập nhật giá trị lớn nhất.

- Hàm GiaiBLNC: Hàm này ta sắp xếp các vật phẩm theo giá trị trên đơn vị trọng lượng giảm dần, sau đó gọi hàm NCBL để tìm giá trị max có thể đạt được.

- Hàm main: Hàm main, người dùng nhập trọng lượng tối đa của ba lô (W) và số lượng vật phẩm (n). Sau đó, người dùng nhập trọng lượng và giá trị của từng vật phẩm. Tiếp theo, các vật phẩm được sắp xếp theo giá trị trên đơn vị trọng lượng giảm dần và chương trình gọi hàm GiaiBLNC để giải bài toán và xuất ra màn hình kết quả.

**VI. “Lập lịch gia công chi tiết (Scheduling Problem)”:**

**1. Lý thuyết:**

“Lập lịch gia công chi tiết” là một bài toán tối ưu ta cần lập lịch thời gian gia công các chi tiết trên các máy móc để tối ưu hóa mục tiêu nào đó, ví dụ như lợi nhuận, hoặc tối đa hiệu quả sử dụng máy móc của nhân công.

**2. Mã code bài toán “lập lịch gia công chi tiết” áp dụng quy hoạch động:**

#include <iostream>

#include <vector>

using namespace std;

int GiaiLLQH(vector<vector<int>>& tg) {

    int n = tg.size(); // Số lượng chi tiết

    int m = tg[0].size(); // Số lượng máy móc

    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));

    for (int i = 1; i <= n; ++i) {

        for (int j = 1; j <= m; ++j) {

            dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]) + tg[i - 1][j - 1];

        }

    }

    return dp[n][m];

}

int main() {

    int n, m;

    cout << "Nhap so luong chi tiet: ";

    cin >> n;

    cout << "Nhap so luong may moc: ";

    cin >> m;

    vector<vector<int>> tg(n, vector<int>(m));

    cout << "Nhap time gia cong cho tung chi tiet tren moi may moc:" << endl;

    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < m; ++j) {

            cin >> tg[i][j];

        }

    }

    cout << "Thoi gian hoan thanh toi uu: " << GiaiLLQH(tg) << endl;

    return 0;

}

- Hàm GiaiLLQH: Hàm này ta sử dụng 1 ma trận dp có kích thước (n + 1) x (m + 1) để lưu trữ kết quả tính toán. Với mỗi chi tiết thứ i và máy móc thứ j, ta tính dp[i][j] là thời gian hoàn thành tối ưu của việc lập lịch cho i chi tiết đầu tiên trên j máy móc đầu tiên.

- Hàm main: Trong hàm main, người dùng nhập số lượng chi tiết (n) và số lượng máy móc (m). Sau đó, người dùng nhập thời gian gia công cho từng chi tiết trên mỗi máy móc. Cuối cùng, gọi hàm GiaiLLQH để chạy chương trình và in kết quả.

**3. Mã code bài toán “lập lịch gia công chi tiết” áp dụng phương pháp nhánh cận:**

#include <iostream>

#include <vector>

#include <algorithm>

#include <limits>

using namespace std;

struct Task {

    int tg;

    int machineIndex;

};

bool compareTasks(Task a, Task b) {

    return a.tg > b.tg;

}

// Hàm tính thời gian hoàn thành tối ưu

int TinhTime(vector<Task>& scheduledTasks, vector<vector<int>>& tg) {

int i;

    int n = scheduledTasks.size(); // Số lượng chi tiết

    int m = tg[0].size(); // Số lượng máy móc

    vector<int> machineTimes(m, 0); // Lưu trữ thời gian hoàn thành trên từng máy móc

    for (i = 0; i < n; ++i) {

        int taskIndex = scheduledTasks[i].machineIndex;

        int machineTime = machineTimes[taskIndex];

        int processingTime = scheduledTasks[i].tg;

        machineTimes[taskIndex] = max(machineTime, machineTime + processingTime);

    }

    return \*max\_element(machineTimes.begin(), machineTimes.end());

}

// Hàm nhánh cận

void LLNC(vector<Task>& tasks, int currentValue, int& minValue, vector<Task>& scheduledTasks, vector<vector<int>>& processingTimes) {

    if (tasks.empty()) {

        int completionTime = TinhTime(scheduledTasks, processingTimes);

        minValue = min(minValue, completionTime);

        return;

    }

    for (int i = 0; i < tasks.size(); ++i) {

        int taskProcessingTime = tasks[i].tg;

        int taskMachineIndex = tasks[i].machineIndex;

        if (currentValue + taskProcessingTime >= minValue) {

            continue;

        }

        scheduledTasks.push\_back(tasks[i]);

        tasks.erase(tasks.begin() + i);

        LLNC(tasks, currentValue + taskProcessingTime, minValue, scheduledTasks, processingTimes);

        tasks.insert(tasks.begin() + i, scheduledTasks.back());

        scheduledTasks.pop\_back();

    }

}

int GiaiLLNC(vector<Task>& tasks, vector<vector<int>>& processingTimes) {

    int minValue = numeric\_limits<int>::max();

    vector<Task> scheduledTasks;

    sort(tasks.begin(), tasks.end(), compareTasks);

    LLNC(tasks, 0, minValue, scheduledTasks, processingTimes);

    return minValue;

}

int main() {

    int n, m;

    cout << "Nhap so luong chi tiet: ";

    cin >> n;

    cout << "Nhap so luong may moc: ";

    cin >> m;

    vector<vector<int>> processingTimes(n, vector<int>(m));

    vector<Task> tasks;

    cout << "Nhap time gia cong cho tung chi tiet tren moi may moc:" << endl;

    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < m; ++j) {

            cin >> processingTimes[i][j];

            tasks.push\_back({ processingTimes[i][j], j });

        }

    }

    cout << "Thoi gian hoan thanh toi uu: " << GiaiLLNC(tasks, processingTimes) << endl;

    return 0;

}

- Hàm compareTasks: Đây là một hàm so sánh được sử dụng trong hàm main để quản lý việc sắp xếp công việc theo thời gian gia công giảm dần.

- Hàm TinhTime: Hàm này tính thời gian hoàn thành tối ưu của lịch trình lập công việc trên các máy móc.

- Hàm LLNC: Đây là hàm thực hiện thuật toán nhánh cận. Hàm này dùng đệ quy để thử all trường hợp có thể có. Chọn từng công việc và xem xét xem có chọn công việc này hay không, sau đó đi tiếp đến công việc tiếp theo. Trong quá trình duyệt, ta kiểm tra xem thời gian hiện tại có nhỏ hơn thời gian nhỏ nhất đã tìm thấy hay không, nếu có, thì cập nhật thời gian nhỏ nhất.

- Hàm GiaiLLNC: Hàm này sắp xếp các công việc theo thời gian gia công giảm dần và chương trình gọi hàm LLNC để giải bài toán và tìm ra thời gian hoàn thành tối ưu.

**VII. Tổng kết:**

Quy hoạch động và nhánh cận là hai kỹ thuật quan trọng để giải quyết các bài toán tối ưu. Kỹ thuật quy hoạch động thích hợp cho các bài toán TSP có số lượng đỉnh nhỏ, nơi đảm bảo tính chính xác của giải pháp tối ưu là quan trọng. Trong khi đó, kỹ thuật Nhánh cận phù hợp với bài toán TSP có số lượng đỉnh lớn, nơi ta cần tìm giải pháp gần đúng trong thời gian hợp lý có sai số nhất định.