BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC NHA TRANG
KHOA CÔNG NGHỆ THÔNG TIN

BÀI BÁO CÁO
TỔNG QUAN VỀ KỸ THUẬT NHÁNH CẬN VÀ QUY HOẠCH ĐỘNG

Giảng viên hướng dẫn: ThS. Nguyễn Hải Triều
Sinh viên thực hiện: Võ Văn Thành
Mã số sinh viên: 64132201
Lớp học phần: 63.CNTT-2

Khánh Hòa, tháng 4 năm 2024

BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC NHA TRANG
KHOA CÔNG NGHỆ THÔNG TIN

BÀI BÁO CÁO
TỔNG QUAN VỀ KỸ THUẬT NHÁNH CẬN VÀ QUY HOẠCH ĐỘNG

Giảng viên hướng dẫn: ThS. Nguyễn Hải Triều
Sinh viên thực hiện: Võ Văn Thành
Mã số sinh viên: 64132201
Lớp học phần: 63.CNTT-2

Khánh Hòa, tháng 4 năm 2024

Mục lục

Chương I

Phần mở đầu

Từ trước đến nay, công nghệ thông tin vẫn luôn là lĩnh vực được rất nhiều người quan tâm. Trong lĩnh vực công nghệ thông tin thì bất cứ ngành nghề nào, từ phát triển ứng dụng, quản trị mạng hay nổi tiếng gần đây là trí tuệ nhân tạo thì cũng đều yêu cầu kiến thức chuyên môn rất cao. Và trước khi tiếp cận với những kiến thức chuyên môn hóa như thế thì chắc chắn ai cũng phải học qua những kiến thức được gọi là nền tảng. Và "Kỹ thuật lập trình" là một trong số các môn nền tảng ấy. Môn học này sẽ giúp chúng ta giải quyết các bài toán một cách tối ưu, từ đó có thể rút ngắn thời gian thực hiện chương trình,...

Trong bài báo cáo này, ta sẽ tìm hiểu về hai kỹ thuật cơ bản là nhánh cận và quy hoạch động cũng như đưa ra ví dụ cho từng kỹ thuật. Cả hai đều là các kỹ thuật được dùng với mục tiêu giải quyết các bài toán một cách tối ưu hơn so với bình thường. Qua đó, bằng việc phân tích ưu điểm và nhược điểm của từng kỹ thuật và các ví dụ, chúng ta sẽ có cái nhìn rõ hơn cũng như sự khác biệt và khả năng áp dụng của hai kỹ thuật trên.

Bài báo cáo sẽ có 3 phần chính:

- Tổng quan lý thuyết, ý tưởng, ưu và nhược điểm của hai 2 kỹ thuật và so sánh chúng.
- Phân tích các bài toán sử dụng hai kỹ thuật trên.
- Tổng kết báo cáo.

Tất cả các đoạn code cũng như chương trình đều sử dụng ngôn ngữ lập trình C++ vì đây là ngôn ngữ tương đối phổ biến cũng như thể hiện rõ được hai kỹ thuật cần xét.

Chương II

Tổng quan vấn đề

1 Tổng quan lý thuyết

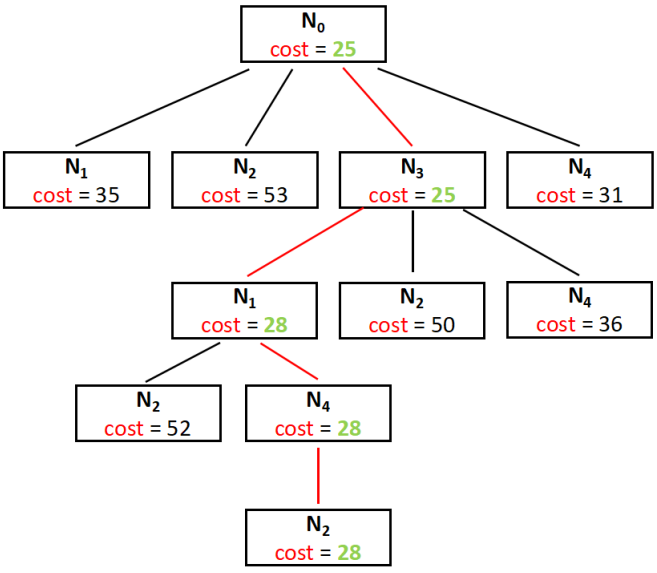
1.1 Kỹ thuật nhánh cận (Branch and Bound)

1.1.1 Lý thuyết

Trong Tin học, chúng ta sẽ rất thường gặp những bài toán yêu cầu phải tìm ra một kết quả thỏa mãn tất cả các yêu cầu của bài toán sao cho kết quả đó là tốt nhất dựa trên một tiêu chí nào đó. Có rất nhiều bài toán như vậy và ta gọi chúng là những ***bài toán tối ưu***.

Phương pháp ***nhánh cận (Branch and Bound)*** là một dạng cải tiến của phương pháp ***quay lui (Backtracking)*** được dùng để tìm nghiệm của những bài toán tối ưu.

Bản chất của kỹ thuật này vẫn là sử dụng phương pháp quay lui nhưng hạn chế không gian duyệt. Thay vì phải kiểm tra hết tất cả các trường hợp thì chúng ta sẽ tận dụng những thông tin đã tìm được để *loại bỏ sớm những phương án chắc chắn không phải là phương án tối ưu.*



Hình II.1: Ứng dụng của kỹ thuật nhánh cận để giải bài toán TSP (Travelling Salesman Problem)

1.1.2 Ý tưởng

Bước 1. Chúng ta sẽ xây dựng nghiệm của bài toán dưới dạng vector $X = (x_1, x_2, \cdots, x_n)$, trong đó mỗi thành phần x_i được chọn ra từ tập các ứng cử viên S_i tương tự như ở phương pháp quay lui.
Bước 2. Vẫn sử dụng phương pháp quay lui nhưng thay vì tuần tự chọn các ứng viên cho từng thành phần của vector nghiệm thì chúng ta sẽ đi đánh giá "độ tốt" của vector nghiệm X bằng một hàm $f(X)$. Vì đây là bài toán tối ưu nên mục tiêu của chúng ta là đi tìm nghiệm X sao cho $f(X)$ tốt nhất (thường là ***lớn nhất*** hoặc ***nhỏ nhất***).
Bước 3. Xây dựng nghiệm của bài toán. Giả sử chúng ta đã xác định được thành phần x_i của nghiệm và chuẩn bị tìm thành phần x_{i+1}. Nếu như kết quả cho ra "tốt hơn" lời giải hiện tại thì ta sẽ chấp nhận thành phần x_{i+1} và tiếp tục tìm các thành phần tiếp theo; trong trường hợp ngược lại, nếu không có trường hợp nào chấp nhận được x_{i+1} hoặc kết quả xấu hơn thì ta sẽ lùi lại bước trước để xác định lại thành phần x_i.

1.1.3 Ưu điểm và nhược điểm

Ưu điểm:
- Giảm không gian tìm kiếm, đặc biệt là đối với các bài toán có không gian tìm kiếm lớn vì không cần phải duyệt qua tất cả các khả năng của bài toán.
- Giảm thời gian tìm kiếm vì đã loại bỏ đi những phương án chắc chắn không thể là phương án tốt nhất.
- Giảm bớt thời gian tính toán.
- Phù hợp với các bài toán tìm kiếm hoặc liên quan đến tối ưu hóa.

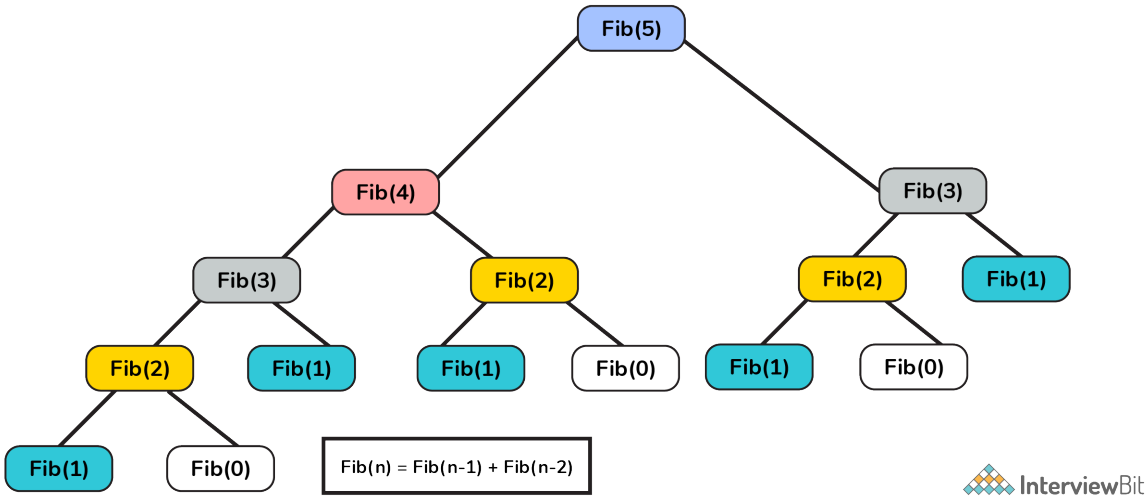
Nhược điểm:
- Phụ thuộc rất nhiều vào việc chọn nhánh. Nếu chọn nhánh không đúng thì có thể làm tăng thời gian thực hiện.
- Việc tính toán cho từng nhánh có thể lâu nếu không gian tìm kiếm quá lớn hoặc các phép tính quá phức tạp.
- Trong một số trường hợp thì chỉ có thể tìm được phương án gần tối ưu nhất.
- Việc xác định được "độ tốt" của nghiệm không phải là chuyện dễ dàng.

1.2 Kỹ thuật quy hoạch động (Dynamic programming)

1.2.1 Lý thuyết

Phương pháp ***quy hoạch động (Dynamic programming)*** là quá trình tiếp cận thuật toán theo kiểu bottom-up, thường được dùng để giải những bài toán tối ưu có bản chất đệ quy. Đây là phương pháp làm giảm thời gian chạy của các thuật toán có các ***bài toán con trùng lắp (overlapping subproblem)*** và ***cấu trúc con tối ưu (optimal substructure)***.
Tương tự như phương pháp ***chia để trị (Devide and Conquer)***, việc tìm nghiệm tối ưu của bài toán đã cho được thực hiện dựa trên việc tìm nghiệm tối ưu của các bài toán

con. Khi đó kết quả của các bài toán con sẽ được ghi nhận lại để phục vụ cho giải quyết các bài toán lớn hơn, từ đó giải được bài toán yêu cầu.
Tuy nhiên, thay vì một bài toán con có thể được giải nhiều lần trong quá trình phân chia bài toán cần giải thành các bài toán con nhỏ hơn như ở kỹ thuật chia để trị, thì ở kỹ thuật quy hoạch động, chúng ta sẽ giải bài toán con một lần và lời giải của các bài toán con đó sẽ được lưu lại nhằm khỏi phải giải lại các bài toán con ấy nếu gặp lại.



Hình II.2: Ứng dụng của kỹ thuật quy hoạch động để tìm số Fibonacci thứ n (cụ thể với $n = 5$)

1.2.2 Ý tưởng

Bước 1. Phân rã bài toán đã cho thành các bài toán con và tìm không gian nhớ $\mathbb{S}$ phù hợp để lưu trữ kết quả của các bài toán con.
Bước 2. Giải các bài toán con nhỏ nhất (hay còn gọi là các bài toán cơ sở) lưu vào $\mathbb{S}$.
Bước 3. Kết hợp kết quả của các bài toán cơ sở để đưa ra lời giải của các bài toán lớn hơn.
Bước 4. Căn cứ vào không gian $\mathbb{S}$ để đưa ra lời giải cho bài toán tổng quát (truy vết của quy hoạch động).

1.2.3 Ưu điểm và nhược điểm

Ưu điểm:
- Tối ưu hóa thời gian tính toán bằng cách lưu trữ kết quả và sử dụng lại chúng khi cần thiết.
- Hoạt động hiệu quả với các bài toán có thể chia nhỏ thành các bài toán con có cấu trúc giống nhau.
- Áp dụng được cho đa số loại bài toán, bao gồm cả các bài toán tối ưu hóa hay tìm kiếm.

Nhược điểm:

• Vì phải lưu trữ lại kết quả của các bài toán con nên không gian lưu trữ phải đủ lớn. Đây là một vấn đề tương đối nan giải đối với các bài toán có dữ liệu vào lớn.

• Khó xác định được cũng như xây dựng được lời giải cho bài toán, từ đó dễ tạo ra một lời giải không tối ưu và hiệu quả.

• Để giải được bài toán đã cho một cách tối ưu thì các bài toán con phải có cấu trúc tối ưu.

• Số lượng các bài toán con không được quá lớn. Về mặt lý thuyết, rất nhiều bài toán có thể giải được bằng kỹ thuật quy hoạch động, tuy nhiên cách làm này là không hiệu quả bởi vì số lượng các bài toán con tăng theo hàm mũ. Một đòi hỏi quan trọng đối với kỹ thuật quy hoạch động là tổng số các bài toán con không được quá nhiều, cùng lắm phải bị chặn bởi một đa thức của kích thước dữ liệu vào.

1.3 So sánh giữa 2 kỹ thuật

Kỹ thuật / **Nội dung so sánh**	**Nhánh cận**	**Quy hoạch động**
Kiểu tiếp cận	Top-down	Bottom-up
Cách tính toán	Dùng đệ quy	Dùng bảng để lưu trữ
Cách phân rã	Nhánh	Các bài toán con
Độ phức tạp thời gian	Lớn nếu lựa chọn nhánh không hiệu quả	Nhỏ vì sử dụng bảng để lưu kết quả
Bài toán thích hợp	Tối ưu hóa	Tối ưu hóa hoặc cấu trúc con

Chương III

Bài toán áp dụng

1 Bài toán người du lịch (TSP - Travelling Salesman Problem)

1.1 Phát biểu bài toán

Cho n thành phố được đánh số từ 1 đến n và các tuyến đường giao thông hai chiều giữa chúng. Mạng giao thông này được cho bởi mảng hai chiều C, trong đó $C_{i,j}$ là chi phí đi trên đoạn đường nối giữa thành phố i và thành phố j. Hiển nhiên $C_{i,j} = C_{j,i}$ và $C_{i,i} = 0$. Một người đi du lịch xuất phát từ thành phố 1, muốn đi thăm tất cả các thành phố còn lại sao cho mỗi thành phố chỉ đi đúng một lần và cuối cùng quay trở về thành phố xuất phát. Hãy chỉ ra hành trình đi của người du lịch với ***chi phí ít nhất***.

Dữ liệu vào:

- Dòng đầu tiên chứa số lượng thành phố N.
- N dòng tiếp theo chứa ma trận chi phí $C[i][j]$.

Dữ liệu ra: In ra đường đi với chi phí thấp nhất.

Ví dụ:

INPUT	OUTPUT
4	117
0 20 35 10	$1 \to 2 \to 4 \to 3 \to 1$
20 0 90 50	
35 90 0 12	
10 50 12 0	

1.2 Giải bằng kỹ thuật nhánh cận

1.2.1 Phân tích ý tưởng

Vector nghiệm của bài toán có dạng $X = (x_1, x_2, \cdots, x_n, x_{n+1})$, trong đó $x_1 = x_{n+1} = 1$ và phải có đường đi trực tiếp giữa hai thành phố x_i và x_{i+1}. Ngoài ra, mỗi thành phố chỉ được phép xuất hiện đúng 1 lần trừ thành phố 1 lặp lại 2 lần nên ta có thể thấy dãy $(x_2, x_3, \cdots, x_n)$ là một hoán vị của $(2, 3, \cdots, n)$.

Ý tưởng dùng kỹ thuật quay lui như sau: Khi đã xây dựng được $(x_1, x_2, \cdots, x_i)$ thì x_{i+1} có thể chọn một trong các thành phố có đường nối trực tiếp với x_i và chưa được chọn. Vì

bài toán sẽ có lượng không gian tìm kiếm lớn khi số lượng thành phố tăng lên, nếu duyệt toàn bộ không gian tìm kiếm sẽ rất phức tạp nên ở đây ta sẽ áp dụng nhánh cận để giảm độ phức tạp như sau:

- Gọi chi phí của đường đi tốt nhất hiện tại là *bestCost*.
- Với mỗi bước chọn x_i, ta kiểm tra xem chi phí đường đi tính tới lúc đó có lớn hơn bằng chi phí tốt nhất hiện tại hay không. Nếu có thì loại bỏ hướng đi này và chọn giá trị khác cho x_i; ngược lại thì ta cố định x_i và kiểm tra tiếp x_{i+1}.
- Tới khi chọn giá trị cho x_n thì cần kiểm tra xem chi phí từ x_n cộng thêm chi phí từ x_n về thành phố 1 có nhỏ hơn chi phí tốt nhất hiện tại hay không. Nếu có thì cập nhật lại chi phí tốt nhất và cách đi tốt nhất; ngược lại thì ta tiếp tục chọn giá trị khác cho x_n.

1.2.2 Các bước thực hiện

- **Bước 1.** Khai báo thư viện, định nghĩa hằng $maxN$ là số lượng thành phố tối đa và $inf = 10^9 + 7$ để thay thế cho $+\infty$. Tiếp đến, ta khai báo các biến toàn cục gồm n là số lượng thành phố, *currentCost* là chi phí của đường đi hiện tại, *bestCost* là chi phí của đường đi tốt nhất; mảng $x[maxN]$ chứa các thành phố, $xBest[maxN]$ chứa các thành phố của đường đi tốt nhất, $visited[maxN]$ dùng để đánh dấu các thành phố để tránh đi lại các thành phố đã đi qua và ma trận $cost[maxN][maxN]$ dùng để lưu chi phí giữa các thành phố.

```
#include <iostream>
#define inf 1e9+7
using namespace std;
const int maxN = 21;
int n, currentCost, bestCost;
int x[maxN], xBest[maxN], visited[maxN], cost[maxN][maxN];
```

- **Bước 2.** Viết hàm *input* dùng để nhập số lượng thành phố, chi phí đi lại giữa các thành phố. Vì xuất phát từ thành phố 1 nên ta đặt $x[1] = 1$ và $visited[1] = 1$ để xác nhận rằng đã đi qua thành phố 1. Tiếp đến, giả sử phương án hiện tại đang rất tệ nên ta sẽ đặt $bestCost = inf$. Vì chưa đi tới thành phố nào nên ta khởi tạo $currentCost = 0$.

```
void input(){
    cin >> n;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            cin >> cost[i][j];
    x[1] = 1;
    visited[1] = 1;
    bestCost = inf;
    currentCost = 0;
}
```

- **Bước 3.** Viết hàm *updateBestSolution* để cập nhật cấu hình tốt nhất. Nếu chi phí hiện tại cộng với chi phí đi từ thành phố x_n đến thành phố 1 nhỏ hơn chi phí tốt nhất, tức là $currentCost + cost[x[n]][1] < bestCost$ thì ta sẽ cập nhật lại chi phí tốt nhất *bestCost*

và đường đi tốt nhất $xBest[i] = x[i]$ với $\forall i = \overline{1, n}$.

```
void updateBestSolution(int currentCost){
    if (currentCost + cost[x[n]][1] < bestCost){
        bestCost = currentCost + cost[x[n]][1];
        for (int i = 1; i <= n; i++)
            xBest[i] = x[i];
    }
}
```

- **Bước 4.** Viết hàm *printBestSolution* để in ra chi phí tốt nhất và đường đi tốt nhất.

```
void printBestSolution(){
    cout << bestCost << endl;
    for (int i = 1; i <= n; i++)
        cout << xBest[i] << "->";
    cout << 1;
}
```

- **Bước 5.** Viết hàm *solution* để kiểm tra giá trị cho thành phố thứ i. Rõ ràng, nếu $curentCost >= bestCost$ thì ta sẽ dừng luôn vì nếu đi tiếp thì giá trị của *currentCost* sẽ ngày càng tăng. Trong trường hợp ngược lại, ta sẽ duyệt các thành phố $j \in [2, n]$. Nếu thành phốj chưa được chọn, tức là $visited[j] = 0$ thì ta sẽ gán $visited[j] = 1$ để đánh dấu rằng thành phố j đang được kiểm tra. Vì đang kiểm tra thành phố j nên ta sẽ gán $x[i] = j$ và cộng thêm vào *currentCost* chi phí đi giữa thành phố $x[i-1]$ và j. Nếu đã sinh xong một cấu hình, tức $i = n$ thì ta sẽ cập nhật chi phí tốt nhất bằng cách gọi hàm *updateBestSolution*; ngược lại, nếu chưa sinh xong thì tiếp tục sinh thành phần tiếp theo. Sau khi kiểm tra xong thành phố j thì ta sẽ bỏ đánh dấu bằng cách gán $visited[j] = 0$ và giảm giá trị của *currentCost* bằng chi phí đi giữa thành phố $x[i-1]$ và j.

```
void solution(int i){
    if (currentCost >= bestCost)
        return ;
    for (int j = 2; j <= n; j++){
        if (!visited[j]){
            visited[j] = 1;
            x[i] = j;
            currentCost += cost[x[i - 1]][j];
            if (i == n)
                updateBestSolution(currentCost);
            else solution(i + 1);
            visited[j] = 0;
            currentCost -= cost[x[i - 1]][j];
        }
    }
}
```

- **Bước 6.** Hàm *main* dùng để gọi tất cả các hàm trên.

```
int main(){
    input();
```

```
    solution(2);
    printBestSolution();
    return 0;
}
```

1.2.3 Chương trình hoàn chỉnh

```
#include <iostream>
#define inf 1e9+7
using namespace std;
const int maxN = 21;
int n, currentCost, bestCost;
int x[maxN], xBest[maxN], visited[maxN], cost[maxN][maxN];
void input(){
    cin >> n;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            cin >> cost[i][j];
    x[1] = 1;
    visited[1] = 1;
    bestCost = inf;
    currentCost = 0;
}
void updateBestSolution(int currentCost){
    if (currentCost + cost[x[n]][1] < bestCost){
        bestCost = currentCost + cost[x[n]][1];
        for (int i = 1; i <= n; i++)
            xBest[i] = x[i];
    }
}
void printBestSolution(){
    cout << bestCost << endl;
    for (int i = 1; i <= n; i++)
        cout << xBest[i] << "->";
    cout << 1;
}
void solution(int i){
    if (currentCost >= bestCost)
        return ;
    for (int j = 2; j <= n; j++){
        if (!visited[j]){
            visited[j] = 1;
            x[i] = j;
            currentCost += cost[x[i - 1]][j];
            if (i == n)
                updateBestSolution(currentCost);
            else solution(i + 1);
            visited[j] = 0;
            currentCost -= cost[x[i - 1]][j];
        }
    }
}
```

```
int main(){
    input();
    solution(2);
    printBestSolution();
    return 0;
}
```

2 Bài toán xếp ba lô 1 (Knapsack)

2.1 Phát biểu bài toán

Trong cửa hàng có N vật, vật thứ i có trọng lượng là W_i và giá trị là V_i. Một tên trộm đột nhập vào cửa hàng, tên trộm mang theo một cái túi có trọng lượng tối đa là M. Hỏi tên trộm sẽ lấy đi những vật nào để có tổng giá trị là lớn nhất, biết rằng mỗi vật chỉ lấy đúng 1 lần.

Dữ liệu vào:

- Dòng đầu tiên chứa giá trị của n và M.
- Dòng thứ 2 chứa giá trị của các đồ vật $v_1, v_2, \cdots, v_n$.
- Dòng thứ 3 chứa trọng lượng của các đồ vật $w_1, w_2, \cdots, w_n$.

Dữ liệu ra:

- Dòng đầu tiên in ra tổng giá trị lớn nhất của các vật lấy được.
- Dòng thứ hai in ra chỉ số của các vật lấy được theo thứ tự giảm dần.

Ví dụ:

INPUT	OUTPUT
4 9 4 7 10 2 5 3 6 4	17 3 2

2.2 Giải bằng kỹ thuật quy hoạch động

2.2.1 Phân tích ý tưởng

Lý do sử dụng quy hoạch động ở đây là vì bài toán có thể được chia thành các bài toán con nhỏ hơn có cấu trúc tối ưu hóa.

Mục tiêu của bài toán là chọn ra các vật sao cho tổng giá trị của chúng là lớn nhất trong khi tổng trọng lượng không được vượt quá trọng lượng của cái túi. Khi đó kỹ thuật quy hoạch động sẽ giúp tối ưu việc này bằng cách lưu trữ và sử dụng lại các kết quả tối ưu của các bài toán con.

Ý tưởng dùng quy hoạch động như sau:

- Ta sẽ xét đồ vật thứ i và giới hạn trọng lượng là j, nếu vật thứ i không được chọn vào phương án tối ưu thì kết quả tối ưu sẽ là $i-1$ vật trước đó và giới hạn trọng lượng vẫn là j.

• Trong trường hợp ngược lại, trọng lượng còn lại là $j - w_i$ cho $i - 1$ vật trước đó và ta tăng thêm giá trị v_i của vật i - trường hợp này chỉ xét khi giới hạn trọng lượng vẫn lớn hơn trọng lượng của vật hiện tại.

2.2.2 Các bước thực hiện

Vì bài toán này không cần đệ quy nên ta sẽ viết tất cả vào hàm *main* của chương trình.
• **Bước 1.** Khai báo và nhập các biến N, W. Tiếp đến ta khai báo và viết vòng lặp để nhập giá trị cho các mảng một chiều $w[N+1]$, $v[N+1]$.

```
int N, W;
cin >> N >> W;
int v[N + 1], w[N + 1];
for (int i = 1; i <= N; i++)
    cin >> v[i];
for (int i = 1; i <= N; i++)
    cin >> w[i];
```

• **Bước 2.** Ta sẽ khai báo một mảng hai chiều $dp[i][j]$ với $0 \leq i \leq N$, $0 \leq j \leq W$ là tổng giá trị lớn nhất của các vật lấy được khi lấy các vật từ 1 tới i và giới hạn tổng trọng lượng là j. Các phần tử của mảng này được xác định như sau:
- Hiển nhiên $dp[0][j] = 0$, $\forall j = \overline{0, W}$ vì giá trị lớn nhất có thể chọn được trong số 0 vật là 0. Tương tự thì ta cũng có $dp[i][0] = 0$, $\forall i = \overline{0, N}$.
- Xét tại vật thứ i và giới hạn trọng lượng hiện tại là j, nếu vật thứ i không được chọn vào phương án tối ưu thì kết quả tối ưu sẽ được chọn trong $i-1$ vật trước đó với giới hạn trọng lượng vẫn là j, tức ta có $dp[i][j] = dp[i-1][j]$. Trong trường hợp ngược lại, nếu vật thứ i được chọn thì trọng lượng tối đa lúc này là $j - w[i]$ cho $i - 1$ vật trước đó, và ta thêm giá trị $v[i]$ của vật thứ i. Dĩ nhiên trường hợp này chỉ xét khi $j \geq w[i]$.
Từ đây ta thấy rằng, nếu $j < w[i]$ thì ta chỉ có một cách lựa chọn là $dp[i-1][j]$, còn nếu $j \geq w[i]$ thì ta có thể chọn giữa $dp[i-1][j]$ hoặc $dp[i-1][j - w[i]] + v[i]$. Vậy ta có công thức quy hoạch động như sau:

$$dp[i][j] = \begin{cases} 0 & \text{nếu } i = 0 \text{ hoặc } j = 0 \\ dp[i-1][j] & \text{nếu } j < w[i] \\ \max\left(dp[i-1][j], dp[i-1][j - w[i]] + v[i]\right) & \text{nếu } j \geq w[i] \end{cases}$$

```
int dp[N + 1][W + 1];
memset(dp, 0, sizeof(dp));
for (int i = 1; i <= N; ++i){
    for (int j = 1; j <= W; ++j){
        dp[i][j] = dp[i - 1][j];
        if (j >= w[i])
            dp[i][j] = max(dp[i][j], dp[i - 1][j - w[i]] + v[i]);
    }
}
```

- **Bước 3.** Kết quả bài toán chính là $dp[N][W]$. Để in ra các vật được chọn, ta sẽ truy ngược lại từ $dp[N][W]$. Nếu $dp[N][W] = dp[N-1][W]$, tức là vật thứ n không được chọn thì ta lùi về $dp[N-1][W]$. Ngược lại, nếu $dp[N][W] \neq dp[N-1][W]$, tức là vật thứ n được chọn thì ta in ra N và lùi về $dp[N-1][W-w[N]]$. Ta tiếp tục thực hiện quá trình trên cho đến khi lùi về $dp[0][j]$, $\forall 0 \leq j \leq W$ thì dừng lại.

```
cout << dp[N][W] << endl;
int i = N, j = W;
while (i > 0 && j > 0){
    if (dp[i][j] != dp[i - 1][j]){
        cout << i << " ";
         j -= w[i];
    }
    i--;
}
```

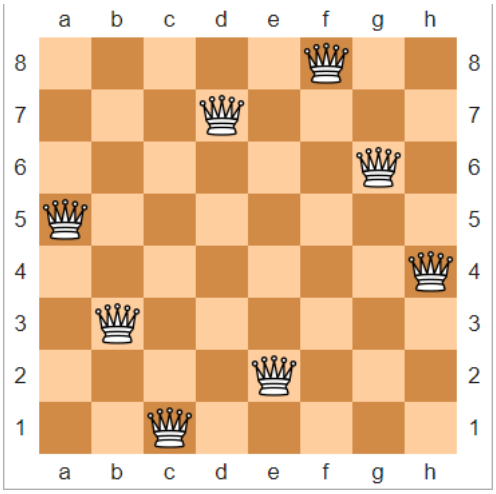
2.2.3 Chương trình hoàn chỉnh

```
#include <iostream>
#include <cstring>
using namespace std;
int main(){
    int N, W;
    cin >> N >> W;
    int v[N + 1], w[N + 1];
    for (int i = 1; i <= N; i++)
        cin >> v[i];
    for (int i = 1; i <= N; i++)
        cin >> w[i];
    int dp[N + 1][W + 1];
    memset(dp, 0, sizeof(dp));
    for (int i = 1; i <= N; ++i){
        for (int j = 1; j <= W; ++j){
            dp[i][j] = dp[i - 1][j];
            if (j >= w[i])
                dp[i][j] = max(dp[i][j], dp[i - 1][j - w[i]] + v[i]);
        }
    }
    cout << dp[N][W] << endl;
    int i = N, j = W;
    while (i > 0 && j > 0){
        if (dp[i][j] != dp[i - 1][j]){
            cout << i << " ";
             j -= w[i];
        }
        i--;
    }
}
```

3 Bài toán N-Queens

3.1 Phát biểu bài toán

Cho bàn cờ vua có kích thước $N \times N$ và N quân hậu. Hãy in ra một cách sắp xếp N quân hậu trên sao cho không có hai quân hậu nào ăn được nhau (trên mỗi hàng, mỗi cột và mỗi đường chéo chỉ có tối đa 1 quân hậu).



Hình III.1: Ví dụ 1 cách xếp của bài toán N-Queens (với $n = 8$)

Dữ liệu vào: Một dòng duy nhất chứa số nguyên dương N nhập từ bàn phím.
Dữ liệu ra: In ra ma trận có kích thước $N \times N$ dùng để chỉ vị trí của các quân hậu. Vị trí nào có quân hậu thì in ra số 1, không có thì in ra số 0.
Ví dụ:

INPUT	OUTPUT
4	0 1 0 0 0 0 0 1 1 0 0 0 0 0 1 0

3.2 Giải bằng kỹ thuật nhánh cận

3.2.1 Phân tích ý tưởng

Ý tưởng dựa trên kỹ thuật quay lui: Ta sẽ đặt từng quân hậu vào các cột khác nhau, bắt đầu từ cột đầu tiên. Khi đặt mỗi quân hậu vào một cột, ta sẽ kiểm tra xem nó có nằm trên đường đi của các quân hậu đã đặt hay không. Ở cột hiện tại, nếu ta tìm thấy một ô mà không nằm trên đường đi của các quân hậu đã đặt, ta sẽ đánh dấu nó như một phần của kết quả. Nếu không tìm thấy ô như vậy thì ta sẽ quay lại bước trước đó.
Nếu như ở thuật toán quay lui, chúng ta tạo ra một mảng hai chiều kiểu đúng/sai có kích thước $N \times N$ để đánh dấu các vị trí có thể đặt và không thể đặt rồi cập nhật nó mỗi khi ta đặt một quân hậu. Tuy nhiên, cách này sẽ tương đối lâu vì ta phải kiểm tra những ô

có thể đặt. Thay vào đó, chúng ta sẽ sử dụng hai mảng 2 chiều có kích thước $N \times N$ để đánh dấu các ô nằm trên đường chéo chính và đường chéo phụ (định nghĩa về đường chéo chính và đường chéo phụ trong mảng 2 chiều này tương tự như ở trong ma trận vuông) và điền giá trị vào hai mảng 2 chiều sao cho nếu 2 quân hậu nằm chung trên một đường chéo thì nó sẽ có cùng giá trị.
Khi đặt quân hậu thứ i vào hàng j, ta sẽ kiểm tra xem nó có nằm trên hàng, đường chéo chính hay đường chéo phụ đã được đánh dấu hay chưa. Nếu chưa thì ta đánh dấu hàng vừa đặt quân hậu thứ i cũng như hai đường chéo chứa quân hậu thứ i rồi chuyển sang quân hậu thứ $i+1$. Ngược lại, nếu có thì trước khi tìm vị trí mới cho quân hậu thứ i, ta cần phải bỏ đánh dấu các hàng, đường chéo chính và đường chéo phụ.

3.2.2 Các bước thực hiện

• **Bước 1.** Khai báo các thư viện và số nguyên dương N. Vì N cố định nên ta có thể khởi tạo giá trị luôn cho N (ở đây lấy $n = 8$).

```
#include <iostream>
#include <cstring>
#define N 8
using namespace std;
```

• **Bước 2.** Viết hàm *solveNQueens* đặt ở cuối cùng (chỉ đứng trước hàm *main* vì còn thêm lệnh) chỉ để khai báo và khởi tạo giá trị cho các mảng 2 chiều chứa các số nguyên gồm *board* có kích thước $N \times N$ để lưu vị trí của các quân hậu, mD có kích thước $N \times N$ để đánh số cho các ô trên đường chéo chính, sD để đánh số các ô trên đường chéo phụ và các mảng 1 chiều có kiểu dữ liệu *bool* gồm $rowCheck$ có kích thước N để kiểm tra xem hàng nào không thể đặt, $mDCheck$ và $sDCheck$ có cùng kích thước $2N-1$ để kiểm tra xem đường chéo chính và đường chéo phụ nào không thể đặt. Tiếp đến, ta sẽ khởi tạo giá trị cho các mảng như sau: $board = \{0\}$, $rowCheck = \{false\}$, $mDCheck = \{false\}$, $sDCheck = \{false\}$. Cuối cùng, ta khởi tạo giá trị cho mD và sD như sau

$$\begin{cases} mD[i][j] = i + j \\ sD[i][j] = i - j + N - 1 \end{cases}$$

Lấy $N-1$ ở đây là để tránh có số âm.

```
void solveNQueens(){
    int board[N][N], mD[N][N], sD[N][N];
    bool rowCheck[N] = {false};
    bool mDCheck[2 * N - 1] = {false};
    bool sDCheck[2 * N - 1] = {false};
    memset(board, 0, sizeof(board));
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            mD[i][j] = i + j;
            sD[i][j] = i - j + N - 1;
        }
    }
```

```
}
```

- **Bước 3.** Viết hàm *printSolution* để in ra ma trận vị trí các quân hậu.

```
void printSolution(int board[N][N]){
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++)
            cout << board[i][j] << " ";
        cout << endl;
    }
}
```

- **Bước 4.** Viết hàm $isSafe$ kiểm tra xem tại $board[i][j]$ có thể đặt quân hậu được không.

```
bool isSafe(int i, int j, int mD[N][N], int sD[N][N], bool rowCheck[], bool
    mDCheck[], bool sDCheck[]){
    if (mDCheck[mD[i][j]] || sDCheck[sD[i][j]] || rowCheck[i])
        return false;
    return true;
}
```

- **Bước 5.** Viết hàm *solve* bằng kỹ thuật nhánh cận với tham trị là cột đang xét hiện tại . Nếu như N quân hậu đều đã được đặt, tức $col \geq N$ thì ta trả về *true*. Ngược lại, ta sẽ đi kiểm tra tại cột thứ j và thử đặt từng quân hậu vào từng ô trong cột đó bằng cách đặt quân hậu vào ô đang xét rồi dùng hàm $isSafe$ để kiểm tra. Nếu được thì ta sẽ chuyển các giá trị thành 1 và *true* rồi tiếp tục với kiểm tra với cột thứ $j + 1$ bằng hàm *solve*. Nếu được thì trả về *true*, ngược lại thì ta sẽ quay về cột trước đó để kiểm tra rồi trả các giá trị về 0 và $false$.

```
bool solve(int j, int board[N][N], int mD[N][N], int sD[N][N], bool
    rowCheck[], bool mDCheck[], bool sDCheck[]){
    if (j >= N)
        return true;
    for (int i = 0; i < N; i++){
        if (isSafe(i, j, mD, sD, rowCheck, mDCheck, sDCheck)){
            board[i][j] = 1;
            rowCheck[i] = true;
            mDCheck[mD[i][j]] = true;
            sDCheck[sD[i][j]] = true;
            if (solve(j + 1, board, mD, sD, rowCheck, mDCheck, sDCheck))
                return true;
            board[i][j] = 0;
            rowCheck[i] = false;
            mDCheck[mD[i][j]] = false;
            sDCheck[sD[i][j]] = false;
        }
    }
    return false;
}
```

- **Bước 6.** Ta sẽ viết vào hàm *solveNQueens* câu lệnh sau để in ra kết quả bài toán.

```
if (solve(0, board, mD, sD, rowCheck, mDCheck, sDCheck))
    printSolution(board);
else cout << "Khong co cach sap xep!" << endl;
```

- **Bước 7.** Hàm *main* để gọi hàm *solveNQueens*.

```
int main(){
    solveNQueens();
    return 0;
}
```

3.2.3 Chương trình hoàn chỉnh

```
#include <iostream>
#include <cstring>
#define N 8
using namespace std;
void printSolution(int board[N][N]){
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++)
            cout << board[i][j] << " ";
        cout << endl;
    }
}
bool isSafe(int i, int j, int mD[N][N], int sD[N][N], bool rowCheck[], bool
    mDCheck[], bool sDCheck[]){
    if (mDCheck[mD[i][j]] || sDCheck[sD[i][j]] || rowCheck[i])
        return false;
    return true;
}
bool solve(int j, int board[N][N], int mD[N][N], int sD[N][N], bool
    rowCheck[], bool mDCheck[], bool sDCheck[]){
    if (j >= N)
        return true;
    for (int i = 0; i < N; i++){
        if (isSafe(i, j, mD, sD, rowCheck, mDCheck, sDCheck)){
            board[i][j] = 1;
            rowCheck[i] = true;
            mDCheck[mD[i][j]] = true;
            sDCheck[sD[i][j]] = true;
            if (solve(j + 1, board, mD, sD, rowCheck, mDCheck, sDCheck))
                return true;
            board[i][j] = 0;
            rowCheck[i] = false;
            mDCheck[mD[i][j]] = false;
            sDCheck[sD[i][j]] = false;
        }
    }
    return false;
}
```

```
void solveNQueens(){
    int board[N][N], mD[N][N], sD[N][N];
    bool rowCheck[N] = {false};
    bool mDCheck[2 * N - 1] = {false};
    bool sDCheck[2 * N - 1] = {false};
    memset(board, 0, sizeof(board));
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            mD[i][j] = i + j;
            sD[i][j] = i - j + N - 1;
        }
    }
    if (solve(0, board, mD, sD, rowCheck, mDCheck, sDCheck))
        printSolution(board);
    else cout << "Khong co cach sap xep!" << endl;
}
int main(){
    solveNQueens();
    return 0;
}
```

Chương IV

Kết luận

1 Tổng kết vấn đề

Nhìn chung, cả nhánh cận và quy hoạch động đều là những kỹ thuật giúp đưa ra được lời giải tối ưu, từ đó có thể tiết kiệm được thời gian thực hiện chương trình hay dung lượng bộ nhớ,...
Thông qua ưu, nhược điểm của từng kỹ thuật cũng như các ví dụ, chúng ta nên sử dụng từng kỹ thuật trong từng trường hợp cụ thể. Tuy nhiên không phải sử dụng những kỹ thuật này là đủ. Chúng ta cần phải kết hợp từng kỹ thuật với các kỹ thuật và cấu trúc dữ liệu khác để có thể đưa ra lời giải tốt nhất.

2 Tài liệu tham khảo

1. Phạm Thị Kim Ngoan, slide bài giảng *Kỹ thuật lập trình*, Trường Đại học Nha Trang.
2. Nguyễn Đức Thuần, *Giáo trình Kỹ thuật lập trình*, Trường Đại học Nha Trang.
3. `https://www.geeksforgeeks.org`
4. `https://viblo.asia`