

Chuyên đề A3: BFS biến đổi trạng thái

Ban Chuyên môn Tin học – Tổ chức The Gifted Battlefield nhiệm kỳ 2025-2026

Xuất bản vào Ngày 10 tháng 2 năm 2026

Tóm tắt nội dung

BFS (Breadth First Search – Tìm kiếm theo chiều rộng) là một trong những thuật toán cơ bản và quan trọng nhất trong lập trình thi đấu. Thuật toán này thường được sử dụng để duyệt hoặc tìm đường trong đồ thị, lưới ô vuông, bản đồ đô thị,... cũng như để giải quyết nhiều bài toán thực tế như tìm đường ngắn nhất, phân tích mạng lưới, hay mô phỏng quá trình lan truyền.

Chuyên đề này sẽ giúp hiểu nguyên lý để giải quyết các bài toán liên quan đến BFS biến đổi trạng thái. Trước khi đi vào phần này, ta sẽ ôn lại về BFS và sau đó ta sẽ đi sâu vào phần lý thuyết liên quan đến BFS biến đổi trạng thái. Sau phần lý thuyết sẽ có một số bài tập giúp ta luyện tập, cũng như một số bài cho ta luyện tập thêm.

Mục lục

1 Ôn lại khái niệm BFS (Tìm kiếm theo chiều rộng)	2
1.1 Cách hoạt động của thuật toán BFS	2
1.2 Mô hình thuật toán	2
2 BFS biến đổi theo trạng thái	4
2.1 BFS biến đổi trạng thái là gì?	4
2.2 Mô hình thuật toán	4
3 Bài tập ví dụ	7
3.1 Bài tập: Biến đổi số	7
3.2 Bài tập: Bội số “bit” nhỏ nhất của n	9
3.3 Bài tập: Biến đổi dãy	11
4 Bài tập thêm	13
4.1 Inversion Sort	13
4.2 Yet Another Multiple Problem	13
4.3 8 Puzzle	13
5 Tài liệu tham khảo	14
6 Lời kết	15

1 Ôn lại khái niệm BFS (Tìm kiếm theo chiều rộng)

1.1 Cách hoạt động của thuật toán BFS

Để hiểu hơn về cách hoạt động của BFS, ta xem qua bài toán đơn giản dưới đây.

Ví dụ 1.1

Cho 1 mảng 2 chiều kích thước 5×5 với ô $(3, 3)$ được đánh số là 1 và những ô còn lại được đánh số là 0. Hãy thực hiện loang từ ô $(3, 3)$ sang những ô kề cạnh đến khi tất cả các ô được đánh số 1.

Ban đầu, ta có mảng hai chiều như sau, trong đó ô $(3,3)$ (tô màu) là vị trí xuất phát và được gán giá trị 1.

4	3	2	3	4
3	2	1	2	3
2	1	0	1	2
3	2	1	2	3
4	3	2	3	4

Ta bắt đầu thực hiện BFS từ ô ở giữa (ô có ghi số 1) đến 4 ô xung quanh, đồng thời ta sẽ sử dụng mảng **visited** để đánh dấu những ô đi đến thành giá trị 1. Ở đây, ta sẽ BFS theo 4 hướng chính là trên, dưới, phải và trái.

Dưới đây là mô phỏng của thuật toán qua các bước:

Bước 1					Bước 2					Bước 3					Bước 4				
4	3	2	3	4	4	3	2	3	4	4	3	2	3	4	4	3	2	3	4
3	2	1	2	3	3	2	1	2	3	3	2	1	2	3	3	2	1	2	3
2	1	0	1	2	2	1	0	1	2	2	1	0	1	2	2	1	0	1	2
3	2	1	2	3	3	2	1	2	3	3	2	1	2	3	3	2	1	2	3
4	3	2	3	4	4	3	2	3	4	4	3	2	3	4	4	3	2	3	4

Sau khi đã có được ý tưởng chạy của thuật toán, bây giờ ta cần thực hiện nó. Thông thường, BFS sẽ sử dụng cấu trúc dữ liệu hàng đợi (queue) để lưu, nhưng ở những bài toán đặc biệt, ta có thể linh động thay đổi.

Về tổng quát, chúng ta có thể hình dung của thuật toán như thế này:

1.2 Mô hình thuật toán

- Khởi tạo hàng đợi Q rỗng.
- Khởi tạo mảng đánh dấu **visited**[n] [m] với tất cả các giá trị là **false**.
- Đưa tọa độ xuất phát $(start_u, start_v)$ vào Q .
- Đánh dấu ô xuất phát đã được thăm: **visited**[start_u] [start_v] = **true**.
- Trong khi Q chưa rỗng:
 - Lấy phần tử ở đầu Q hiện tại ra (u, v) .

- Nếu (u, v) là trạng thái đích $\rightarrow$ dừng (hoặc lưu kết quả).
- Duyệt 4 ô kề (new_u, new_v) tương ứng với các hướng di chuyển: $(u + 1, v)$, $(u - 1, v)$, $(u, v + 1)$, $(u, v - 1)$.
- Với mỗi ô kề (new_u, new_v) tiếp theo:
 - ◊ Kiểm tra trạng thái thăm: Kiểm tra xem ô này đã được thăm chưa bằng cách xét giá trị `visited[new_u][new_v]`.
 - ◊ Kiểm tra ô (new_u, new_v) có hợp lệ không (ví dụ: có nằm trong biên hay không, có phải vật cản hay không...).
 - ◊ Nếu chưa gặp (`visited[new_u][new_v] == false`) và ô (new_u, new_v) hợp lệ:
 - ▷ Đánh dấu đã gặp: `visited[new_u][new_v] = true`.
 - ▷ Cập nhật thông tin (ví dụ: khoảng cách, đường đi...).
 - ▷ Đẩy trạng thái mới (new_u, new_v) vào Q (thao tác `push`).

Áp dụng mô hình thuật toán, ta có đoạn cài đặt minh hoạ như sau:

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 // Mảng hằng số định hướng di chuyển (Lên, Phải, Xuống, Trái)
5 const int DX[] = {-1, 0, 1, 0};
6 const int DY[] = {0, 1, 0, -1};
7
8 // Hàm BFS duyệt lưới
9 int bfs(int n, int startX, int startY) {
10     vector<vector<bool>> visited(n + 1, vector<bool>(n + 1, false));
11     queue<pair<int, int>> q;
12
13     // 1. Đưa vị trí ban đầu vào hàng đợi
14     q.push({startX, startY});
15
16     // 2. Đánh dấu ngay khi push để tránh lặp
17     visited[startX][startY] = true;
18
19     while (!q.empty()) {
20         pair<int, int> currentPos = q.front();
21         q.pop();
22
23         // Duyệt 4 hướng xung quanh
24         for (int i = 0; i < 4; i++) {
25             int nextX = currentPos.first + DX[i];
26             int nextY = currentPos.second + DY[i];
27
28             // 3. Kiểm tra điều kiện biên
29             // Lưu ý: Giả sử lưới đánh số từ 1 đến n
30             if (nextX >= 1 && nextX <= n && nextY >= 1 && nextY <= n) {
31                 // 4. Nếu ô này chưa được thăm
32                 if (!visited[nextX][nextY]) {
33                     visited[nextX][nextY] = true; // Đánh dấu
34                     q.push({nextX, nextY});        // Đẩy vào hàng đợi
35                 }
36             }
37         }
38     }
39 }

```

```

40
41 int main() {
42     bfs(5, 3, 3);
43 }

```

2 BFS biến đổi theo trạng thái

2.1 BFS biến đổi trạng thái là gì?

Lưu ý

Trên thực tế, BFS **không chỉ được sử dụng trên ma trận** mà còn được ứng dụng được cho nhiều trường hợp và bài toán khác.

- **Trạng thái:** Hãy tưởng tượng trạng thái giống như một “ô” trong ma trận. Tuy nhiên, thay vì được định danh bằng tọa độ (i, j) , một trạng thái có thể là một con số, một dãy số, hoặc một cách sắp xếp cụ thể nào đó.
- **Chuyển trạng thái:** Thay vì chỉ đơn thuần di chuyển sang các ô kề bên (trên, dưới, trái, phải), ta chuyển từ trạng thái này sang trạng thái khác thông qua **các quy luật hoặc phép xử lý** mà đề bài cho phép (ví dụ: cộng trừ số, hoán đổi vị trí, xoay hình,...).

So với BFS thông thường, BFS biến đổi trạng thái sẽ có ý tưởng và cơ chế hoạt động **giống nhau** nhưng sẽ được thực hiện trên những trạng thái của một hệ thống nào đó.

Thay vì duyệt qua các vị trí liên kề trên một ma trận thì những vị trí, hay trạng thái liên kề sẽ được sinh ra từ một hệ thống, quy luật nào đó.

Để hiểu rõ hơn, ta sẽ thử lấy lại ví dụ đã nói ở trên.

Trong ví dụ mảng 2 chiều 5×5 như trên, mỗi trạng thái là các ô (i, j) , từ một trạng thái chính (ô ở giữa có đánh số là 1), ta chạy BFS để sinh ra các trạng thái con xung quanh.

Ví dụ thử cách chạy, ta có ở đây trạng thái ban đầu là ô $(3, 3)$, sau khi chạy BFS 1 lần ta chuyển từ trạng thái ban đầu ở ô $(3, 3)$ sang 4 trạng thái con là các ô $(2, 3)$, $(3, 2)$, $(4, 3)$, $(3, 4)$.

Tuy nhiên, như đã đề cập, các bài toán BFS có thể không đơn thuần được cho trên ma trận có sẵn như bài toán trên. Ta cần tự thiết kế một định nghĩa trạng thái và các phép biến đổi trạng thái tương ứng dựa trên những dữ kiện được cho, và đó cũng chính là điểm khác biệt chính của BFS thường và BFS chuyển đổi trạng thái.

2.2 Mô hình thuật toán

- Khởi tạo hàng đợi Q rỗng.
- Khởi tạo mảng đánh dấu $dist$ (hoặc $visited$) với giá trị khởi tạo (ví dụ -1 hoặc $false$).
- Đưa trạng thái xuất phát S_{start} vào Q .
- Đánh dấu trạng thái xuất phát đã được thăm: $dist[S_{start}] = 0$ (hoặc $visited[S_{start}] = true$).
- Trong khi Q chưa rỗng:
 - Lấy trạng thái ở đầu Q hiện tại ra (S_{cur}).
 - Nếu S_{cur} là trạng thái đích $\rightarrow$ dừng (hoặc lưu kết quả).

- Sinh ra các trạng thái mới S_{new} từ S_{cur} bằng cách áp dụng các quy tắc biến đổi của đề bài (ví dụ: cộng, trừ, nhân, xoay, tráo đổi...).
- Với mỗi trạng thái S_{new} vừa được sinh ra:
 - ◊ Kiểm tra tính hợp lệ: Kiểm tra xem S_{new} có thỏa mãn điều kiện bài toán không (ví dụ: nằm trong giới hạn giá trị, không vi phạm quy tắc cấm...).
 - ◊ Kiểm tra trạng thái thăm: Kiểm tra xem S_{new} đã được duyệt trước đó hay chưa (dựa vào mảng `dist` hoặc `visited`).
 - ◊ Nếu chưa gặp (ví dụ `dist[S_new] == -1`) và trạng thái S_{new} hợp lệ:
 - ▷ Đánh dấu đã gặp: `dist[S_new] = dist[S_cur] + 1`.
 - ▷ Cập nhật thông tin khác (nếu cần thiết như truy vết đường đi).
 - ▷ Đẩy trạng thái mới S_{new} vào Q (thao tác `push`).

Để hình dung rõ hơn, chúng ta đến với 1 trong những kiểu bài đặc trưng của BFS biến đổi trạng thái.

Ví dụ 2.1

Cho 2 số nguyên dương n và m ($n, m \leq 10^5$). Ta có 3 phép toán có thể thực hiện lên n là:

- Tăng n lên 1 đơn vị.
- Giảm n đi 1 đơn vị.
- Nhân n cho 2.

Tìm số bước ít nhất để $n = m$.

■ Hướng giải:

Một chiến lược “tham lam” (greedy) có thể sẽ thất bại. Ví dụ, để đi từ $n = 4$ đến $m = 11$, chiến lược luôn ưu tiên nhân 2 nếu $n < m$ sẽ cho đường đi $4 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11$ (mất 4 bước). Rõ ràng đây không phải là tối ưu, vì BFS sẽ tìm ra đường đi $4 \rightarrow 5 \rightarrow 10 \rightarrow 11$ (chỉ mất 3 bước).

Nhận xét: Với mỗi thao tác, ta sẽ có các số mới, giống như việc ta di chuyển trên 1 cạnh hay phép biến đổi từ số này sang số mới, ta có định nghĩa sau:

Định nghĩa 2.1 (Lý thuyết cơ bản về đồ thị)

Một **đồ thị** là một cặp $G = (V, E)$, trong đó:

- V là **tập đỉnh** (vertices), biểu diễn các đối tượng.
- $E \subseteq V \times V$ là **tập cạnh** (edges), biểu diễn các mối quan hệ giữa các đỉnh trong V .
- Nếu các cạnh có **hướng**, mỗi cạnh là một cặp có thứ tự (u, v) , và ta gọi G là **đồ thị có hướng**.
- Nếu các cạnh **không có hướng**, mỗi cạnh là một cặp không có thứ tự $\{u, v\}$ với $u \neq v$, và ta gọi G là **đồ thị vô hướng**.

Ý tưởng: Ta nhìn theo hướng đồ thị. Ta sẽ có 1 đồ thị $G = (V, E)$ với:

- G là đồ thị có hướng (có cạnh $n \rightarrow 2 \times n$ nhưng không có cạnh $2 \times n \rightarrow n$)
- V là tập hợp các con số (trạng thái) có thể đạt được.

- E là tập hợp các cạnh liên kết 2 số, biểu diễn 3 phép biến đổi:
 - Cạnh $u \rightarrow u - 1$ (giảm 1).
 - Cạnh $u \rightarrow u + 1$ (tăng 1).
 - Cạnh $u \rightarrow u \times 2$ (nhân 2).

⇒ Bài toán trở thành tìm đường đi ngắn nhất (ít phép biến đổi nhất) từ đỉnh n đến đỉnh m trên đồ thị này. Ta duyệt BFS trạng thái, loang ra theo 3 cách biến đổi. Nếu đang duyệt mà gặp m thì in ra số bước. Chúng ta có 3 phép biến đổi (tương ứng với các cạnh của đồ thị) để đi từ trạng thái này (một con số) sang trạng thái khác.

Do đó, ta phải dùng BFS để duyệt song song tất cả các khả năng, đảm bảo tìm được đường đi ngắn nhất (số bước ít nhất).

Ta có thể chứng minh cận trên của một trạng thái cần xét là khoảng $2m$. Gọi $\text{dist}[u]$ là số bước di chuyển ít nhất từ trạng thái ban đầu n đến trạng thái u .

- Lý do là nếu ta đang ở một trạng thái $u > m$, thì thao tác tối ưu nhất chắc chắn là giảm dần u về m (trừ 1), vì việc nhân 2 sẽ làm u xa m hơn.
- Do đó, đáp án từ u sẽ là $\text{dist}[u] + (u - m)$. Việc đi quá xa khỏi m rồi quay lại là không cần thiết.
- Vì vậy, ta chỉ cần khai báo mảng đánh dấu kích thước khoảng $2m$ (với $m \leq 10^5$ thì mảng khoảng $2 \cdot 10^5$ phần tử) là đủ để code chạy nhanh và tiết kiệm bộ nhớ.

■ Mô phỏng ý tưởng

- Khởi tạo hàng đợi Q rỗng.
- Khởi tạo mảng dist với kích thước khoảng $2m$ (theo nhận xét ở trên), gán tất cả giá trị là -1 (chưa thăm).
- Đưa trạng thái ban đầu n vào Q .
- Đánh dấu trạng thái xuất phát đã được thăm: $\text{dist}[n] = 0$.
- Trong khi Q chưa rỗng:
 - Lấy trạng thái ở đầu Q hiện tại ra x .
 - Nếu $x = m$ là trạng thái đích → dừng và trả về kết quả $\text{dist}[x]$.
 - Sinh ra các trạng thái kế nx từ x tương ứng với 3 phép biến đổi: $x - 1, x + 1, x \times 2$.
 - Với mỗi trạng thái kế nx tiếp theo:
 - ◊ Kiểm tra tính hợp lệ: $nx \geq 0$ và $nx < 2m$.
 - ◊ Kiểm tra trạng thái thăm: xét xem $\text{dist}[nx]$ có bằng -1 hay không.
 - ◊ Nếu chưa gặp ($\text{dist}[nx] == -1$) và trạng thái nx hợp lệ:
 - ▷ Đánh dấu đã gặp và cập nhật khoảng cách: $\text{dist}[nx] = \text{dist}[x] + 1$.
 - ▷ Đẩy trạng thái mới nx vào Q .

Lưu ý

Để ý rằng sẽ luôn luôn có cách để biến đổi n thành m . Do trường hợp tệ nhất thì luôn có thể sử dụng $|n - m|$ phép biến đổi $n + 1$ hay $n - 1$.

Code hàm xử lý mẫu (Độ phức tạp: $\mathcal{O}(m)$):

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 const int MAX = 2e5 + 5; // Với m <= 10^5, trạng thái nx có thể lên đến 2 * 10^5.
5 vector<int> dist(MAX, -1);
6
7 int main() {
8     int n, m;
9     cin >> n >> m;
10
11     queue<int> q;
12     q.push(n);
13     dist[n] = 0;
14
15     while (!q.empty()) {
16         int x = q.front(); q.pop();
17
18         // Tìm thấy đích
19         if (x == m) {
20             cout << dist[x];
21             return 0;
22         }
23
24         // Mảng chứa 3 trạng thái kề
25         int next_states[3] = {x - 1, x + 1, x * 2};
26
27         // Kiểm tra trạng thái mới có hợp lệ (trong biên) và chưa được thăm
28         for (int nx : next_states) {
29             if (nx >= 0 && nx < MAX && dist[nx] == -1) {
30                 dist[nx] = dist[x] + 1; // Cập nhật số bước
31                 q.push(nx);
32             }
33         }
34     }
35     return 0;
36 }
```

3 Bài tập ví dụ

3.1 Bài tập: Biến đổi số

■ Tóm tắt đề bài

Cho 2 số nguyên dương n và m ($n, m \leq 10^5$), ta được phép biến đổi số n theo 2 cách sau:

- Cộng 3 vào n .
- Nhân n cho 2.

Hãy tìm số lần biến đổi ít nhất sao cho $n = m$. Nếu không thể thì in ra “IMPOSSIBLE”.

■ Gợi ý

Bài toán này có cấu trúc cốt lõi tương tự như ví dụ đã phân tích ở trên (biến đổi $n \rightarrow m$ với các phép toán $-1, +1, \times 2$).

■ Hướng giải

Ý tưởng: Ta nhìn theo hướng đồ thị. Ta sẽ có 1 đồ thị $G = (V, E)$ với:

- G là đồ thị có hướng (có cạnh $n \rightarrow 2 \times n$ nhưng không có cạnh $2 \times n \rightarrow n$)
- V là tập hợp các con số từ phép biến đổi n
- E là tập hợp các cạnh liên kết 2 số có 2 dạng là cộng 3 và nhân 2

⇒ Đây chính là bài toán tìm đường đi ngắn nhất. Ta duyệt BFS trạng thái, loang ra theo 2 cách biến đổi. Nếu đang duyệt mà gặp m thì in ra giá trị. Khi duyệt xong mà vẫn không có m thì in ra IMPOSSIBLE.

Lưu ý

Chỉ cần xét tới các số $x \leq m$ vì càng lớn hơn m thì càng xa dẫn tới bất lợi.

Độ phức tạp: $\mathcal{O}(m)$

■ Cài đặt

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 const int MAX = 2e5 + 5; // Với m <= 10^5, trạng thái nx có thể lên đến 2 * 10^5.
5 vector<int> dist(MAX, -1);
6
7 int main() {
8     int n, m;
9     cin >> n >> m;
10
11     queue<int> q;
12     q.push(n);
13     dist[n] = 0;
14
15     while (!q.empty()) {
16         int x = q.front(); q.pop();
17
18         // Tìm thấy đích
19         if (x == m) {
20             cout << dist[x];
21             return 0;
22         }
23         // Do chỉ có phép cộng và nhân, nếu x > m thì không có cách nào để tiến về m từ x được
24         if (x > m) {
25             continue;
26         }
27     }
```

```

28 // Mảng chứa 2 trạng thái kế
29 int next_states[2] = {x + 3, x * 2};
30
31 // Kiểm tra trạng thái mới có hợp lệ (trong biên) và chưa được thăm
32 for (int nx : next_states) {
33     if (nx < MAX && dist[nx] == -1) {
34         dist[nx] = dist[x] + 1; // Cập nhật số bước
35         q.push(nx);
36     }
37 }
38 }
39 cout << "IMPOSSIBLE";
40 return 0;
41 }

```

3.2 Bài tập: Bội số “bit” nhỏ nhất của n

Tóm tắt đề bài

Cho số nguyên dương n ($n \leq 20,000$). Tìm số nguyên dương X nhỏ nhất là bội số của n mà X chỉ chứa toàn các chữ số 0 và 1, đồng thời bắt đầu bằng chữ số 1.

Gợi ý

Ta hình dung quá trình duyệt như việc “xây dựng” một số nguyên lớn từ trái sang phải, ta có 2 phép biến đổi là thêm chữ số 0 hoặc 1 vào vị trí tận cùng bên phải (chữ số sau khi thêm trở thành hàng đơn vị).

Hướng giải

Ý tưởng:

- Ta có đồ thị $G = (V, E)$ với V là các dãy số gồm 0 và 1
- E là tập hợp 2 phép biến đổi thêm 0 và 1 liên kết 2 số trước và sau

⇒ Ta duyệt BFS trạng thái với trạng thái đầu tiên là số 1 (yêu cầu của đề). Gọi trạng thái hiện tại là số nguyên dương s , ta loang ra thêm 2 trạng thái mới là $s \times 10$ (thêm số 0) và $s \times 10 + 1$ (thêm số 1). Với điều kiện tìm bội số bé nhất, ta ưu tiên thêm số 0 để có số là bé nhất có thể.

Lưu ý

Nếu $n = 1$ thì nên in ra 1 luôn tránh chương trình bị lỗi.

Độ phức tạp: $\mathcal{O}(n)$

Cài đặt

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 const int MAX_N = 20005;
5

```

```

6 // Struct lưu thông tin để truy vết đường đi
7 struct TraceInfo {
8     int prevRem; // Số dư của trạng thái trước đó
9     char digit; // Chữ số ('0' hoặc '1') đã thêm vào
10 };
11
12 int n;
13 bool visited[MAX_N];
14 TraceInfo trace[MAX_N]; // Mảng lưu vết
15
16 int main() {
17     ios_base::sync_with_stdio(false);
18     cin.tie(NULL);
19
20     cin >> n;
21
22     // Xử lý trường hợp đặc biệt n = 1
23     if (n == 1) {
24         cout << "1";
25         return 0;
26     }
27
28     queue<int> q;
29
30     // Trạng thái ban đầu: Số 1
31     q.push(1);
32     visited[1] = true;
33
34     // Đánh dấu điểm bắt đầu: prevRem = -1 để làm điểm dừng khi truy vết
35     trace[1] = {-1, '1'};
36
37     while (!q.empty()) {
38         int currRem = q.front(); // Là số dư hiện tại
39         q.pop();
40
41         // Nếu tìm thấy số dư 0 (tức là số đã tạo chia hết cho n)
42         if (currRem == 0) {
43             string res = "";
44             int temp = 0; // Bắt đầu từ trạng thái đích (dư 0)
45
46             // Lăn ngược lại theo vết cho đến khi gặp -1
47             while (temp != -1) {
48                 res.push_back(trace[temp].digit);
49                 temp = trace[temp].prevRem;
50             }
51
52             // Đảo ngược chuỗi để có thứ tự đúng (từ trái sang phải)
53             reverse(res.begin(), res.end());
54
55             cout << res;
56             return 0;
57         }
58
59         // Thử thêm '0' và '1' vào cuối số hiện tại
60         char digits[] = {'0', '1'};
61

```

```

62     for (char d : digits) {
63         int val = d - '0';
64         // số dư mới = (số dư cũ * 10 + chữ số thêm) int nextRem = (currRem * 10 + val)
65         // Nếu số dư này chưa từng gặp
66         if (!visited[nextRem]) {
67             visited[nextRem] = true;
68             trace[nextRem] = {currRem, d}; // Lưu vết: từ currRem thêm d ra nextRem
69             q.push(nextRem);
70         }
71     }
72 }
73
74 return 0;
75 }

```

3.3 Bài tập: Biến đổi dãy

■ Tóm tắt đề bài

Cho một dãy gồm các số nguyên không âm $a_1, a_2, \dots, a_n$ ($4 \leq n \leq 8, 0 \leq a_i \leq 10$). Một *phép biến đổi* được định nghĩa như sau: chọn một đoạn con liên tiếp gồm 4 phần tử (a, b, c, d) và thay thế chúng bởi:

$$(|a - b|, |b - c|, |c - d|, |d - a|)$$

Tìm số phép biến đổi ít nhất cần thực hiện để toàn bộ dãy trở thành dãy toàn 0.

■ Hướng giải

Trước hết, ta nhận thấy rằng các giới hạn n, a_i khá nhỏ nên ta có thể biểu diễn trạng thái bằng một **vector** gồm n phần tử, sau đó sinh các trạng thái kế bằng cách thử mọi đoạn con 4 phần tử liên tiếp rồi tiếp tục tìm kiếm trên các trạng thái này. Quá trình tìm kiếm kết thúc khi ta gặp trạng thái chỉ gồm số 0.

Để quản lý các trạng thái đã duyệt qua, ta sử dụng cấu trúc dữ liệu **map**.

■ Cài đặt

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      ios_base::sync_with_stdio(false);
6      cin.tie(NULL);
7
8      int n;
9      cin >> n;
10
11     vector<int> startSeq(n);
12     for (int i = 0; i < n; ++i) {
13         cin >> startSeq[i];
14     }
15

```

```

16 // Trạng thái đích: toàn bộ dãy là số 0
17 vector<int> targetSeq(n, 0);
18
19 // Hàng đợi lưu các trạng thái của dãy số
20 queue<vector<int>>> q;
21
22 // Map dùng để lưu số bước đi ngắn nhất đến mỗi trạng thái
23 // Đồng thời đóng vai trò kiểm tra trạng thái đã thăm
24 map<vector<int>, int> dist;
25
26 // Đẩy trạng thái đầu vào hàng đợi
27 q.push(startSeq);
28 dist[startSeq] = 0;
29
30 while (!q.empty()) {
31     vector<int> currentSeq = q.front();
32     q.pop();
33
34     int currentDist = dist[currentSeq];
35
36     // Kiểm tra nếu đã đạt trạng thái đích
37     if (currentSeq == targetSeq) {
38         cout << currentDist;
39         return 0;
40     }
41
42     // Duyệt qua tất cả các vị trí có thể áp dụng phép biến đổi
43     // Cửa sổ 4 phần tử bắt đầu từ i: [i, i+1, i+2, i+3]
44     for (int i = 0; i < n - 3; ++i) {
45         vector<int> nextSeq = currentSeq;
46
47         // Lấy giá trị của 4 phần tử liên tiếp
48         int a = currentSeq[i];
49         int b = currentSeq[i + 1];
50         int c = currentSeq[i + 2];
51         int d = currentSeq[i + 3];
52
53         // Áp dụng công thức biến đổi của đề bài
54         // Thay thế bằng (|a-b|, |b-c|, |c-d|, |d-a|)
55         nextSeq[i] = abs(a - b);
56         nextSeq[i + 1] = abs(b - c);
57         nextSeq[i + 2] = abs(c - d);
58         nextSeq[i + 3] = abs(d - a);
59
60         // Nếu chưa có trong map nghĩa là chưa được thăm
61         if (!dist.count(nextSeq)) {
62             dist[nextSeq] = currentDist + 1;
63             q.push(nextSeq);
64         }
65     }
66 }
67
68 return 0;
69 }

```

4 Bài tập thêm

4.1 Inversion Sort

- Link đề gốc: <https://www.spoj.com/problems/INVESORT/>

■ Tóm tắt đề bài

Có nhiều bộ test, với mỗi bộ test gồm: hai chuỗi C và D , mỗi chuỗi có độ dài 10 là một hoán vị của các chữ cái từ a đến j.

Một thao tác *inversion* là chọn một đoạn liên tiếp trong chuỗi và đảo ngược thứ tự các ký tự trong đoạn đó.

Tìm số lượng thao tác đảo ngược ít nhất cần thiết để biến chuỗi C thành chuỗi D .

■ Hướng giải

Từ chuỗi C , ta dùng BFS trên không gian là mọi hoán vị có thể. Tại mỗi trạng thái, ta sẽ đảo ngược mọi đoạn con rồi đưa vào hàng đợi để xử lý tiếp. Số bước tối thiểu từ C đến D chính là số lần đảo ngược cần thiết.

4.2 Yet Another Multiple Problem

- Link đề gốc: <https://www.spoj.com/problems/MULTII/>

■ Tóm tắt đề bài

Cho n ($1 \leq n \leq 10^4$) và m số nguyên dương. Tìm bội nhỏ nhất của n sao cho trong bội đó không có các chữ số trong m số được cho trước. Nếu không thể tìm được, in ra -1 .

■ Hướng giải

Dùng BFS trên các bội số của n , biểu diễn mỗi trạng thái (bội số) bằng phần dư khi chia cho n :

- Mỗi trạng thái là một phần dư modulo n và chuỗi số hiện tại.
- Từ mỗi trạng thái, thêm một chữ số hợp lệ (không bị cấm) vào cuối và tính phần dư mới.
- BFS đảm bảo rằng lần đầu gặp phần dư 0 sẽ là bội nhỏ nhất thỏa điều kiện.

Vì $n \leq 10000$ nên ta có phần dư modulo n của mọi trạng thái luôn nhỏ hơn 10000. Do đó, ta có thể đánh dấu phần dư đó vào một mảng đánh dấu.

4.3 8 Puzzle

- Link đề gốc: judge.u-aizu.ac.jp/onlinejudge/description.jsp

■ Tóm tắt đề bài

“Bài toán 8 số” là một bài toán trượt ô, được giải quyết bằng cách di chuyển các ô số trong một khung hình. Khung hình được chia thành 9 ô, xếp thành 3 hàng và 3 cột, với 8 ô chứa số từ 1 đến 8 và một ô trống (ô 0). Mục tiêu của trò chơi là sắp xếp lại các số về đúng thứ tự bằng cách trượt các ô số vào ô trống.

Trong bài toán này, trạng thái ban đầu của khung hình được cho trước. Hãy viết một chương trình tìm số lần di chuyển **tối thiểu** để đạt được trạng thái đích (trạng thái hoàn thành) như sau:

$$\text{Trạng thái đích: } \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$$

■ Hướng giải

Biểu diễn mỗi trạng thái là một chuỗi hoặc mảng 9 phần tử, trong đó ô trống là 0. Ta thực hiện BFS như sau:

- Từ trạng thái hiện tại, sinh tất cả các trạng thái kế bằng cách trượt một ô số vào ô trống, nghĩa là từ ô trống (ô có giá trị 0), ta xét việc các ô kề với nó được trượt vào ô trống đó và đưa vào hàng đợi.
- Khi lần đầu gặp trạng thái đích, số bước đi tới chính là số bước di chuyển tối thiểu do BFS đảm bảo tìm được số bước ít nhất.

■ Mở rộng: CSES - Swap Game

Đây là một biến thể thú vị có ý tưởng tương tự bài 8 Puzzle, các bạn có thể luyện tập thêm để nắm vững kỹ thuật BFS trên lưới.

- Link đề gốc: <https://cses.fi/problemset/task/1670>

Bài toán: Cho một lưới 3×3 chứa đầy đủ các số từ 1 đến 9. Khác với 8 Puzzle (chỉ có 1 ô trống để trượt), ở bài này, tại mỗi bước bạn được phép **hoán đổi vị trí của hai ô bất kỳ kề nhau** (theo chiều dọc hoặc ngang). Mục tiêu vẫn là đưa lưới về trạng thái các số được sắp xếp tăng dần từ 1 đến 9 với số bước ít nhất.

Gợi ý: Ta vẫn sử dụng BFS để tìm đường đi ngắn nhất tương tự như 8 Puzzle.

- *Trạng thái:* Một hoán vị của 9 chữ số (có thể lưu dạng chuỗi hoặc mảng).
- *Chuyển trạng thái:* Thay vì chỉ xét các ô kề với ô trống, ta xét **tất cả** các cặp ô kề nhau trong lưới hiện tại. Nếu hoán đổi chúng tạo ra một trạng thái chưa thăm, ta đẩy vào hàng đợi.
- *Độ phức tạp:* Không gian trạng thái là $9! = 362,880$, hoàn toàn phù hợp để duyệt toàn bộ bằng BFS.

5 Tài liệu tham khảo

Tài liệu

- [1] VNOI Wiki. (n.d.). *BFS (Breadth-first search)*. Retrieved January 16, 2026, from <https://wiki.vnoi.info/algo/graph-theory/breadth-first-search.md>

- [2] CP-Algorithms. (n.d.). *Breadth first search*. Retrieved January 16, 2026, from <https://cp-algorithms.com/graph/breadth-first-search.html>
- [3] GeeksforGeeks. (n.d.). *Breadth first search or BFS for a graph*. Retrieved January 16, 2026, from <https://www.geeksforgeeks.org/dsa/breadth-first-search-or-bfs-for-a-graph/>

6 Lời kết

Sau những bài học thú vị về thuật toán BFS biến đổi trạng thái, mình tin rằng bạn đã hiểu rõ hơn về nó trong lập trình thi đấu và thực, cũng như những ứng dụng thực tế của nó trong các bài toán đồ thị, như Edsger Dijkstra từng nói:

“Thuật toán là trái tim của lập trình, và hiểu một thuật toán chính là hiểu cách tư duy.”

Hãy tiếp tục khám phá và rèn luyện thêm với các dạng bài toán về BFS, bởi phía trước vẫn còn rất nhiều điều thú vị đang chờ bạn chinh phục!