

ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

—o0o—



BÁO CÁO
BÀI TOÁN AHC11 - A. SLIDING TREE PUZZLE

Giảng viên hướng dẫn: **TS. Nghiêm Nguyễn Việt Dũng**

Lớp môn học: **2324II_INT3103_1 - Tối ưu hóa**

Sinh viên thực hiện: **Huỳnh Tiến Dũng** **21020007**

Vũ Quốc Tuấn **21020033**

Ngô Hán Quang Ngọc **21020550**

HÀ NỘI, 2024

Mục lục

1	Bài toán	2
1.1	Tóm tắt	2
1.2	Đầu vào	2
1.3	Đầu ra	3
1.4	Các ràng buộc và đánh giá	3
2	Giải quyết bài toán	4
2.1	Xem xét bài toán	4
2.2	Cơ sở lý thuyết	4
2.2.1	Tìm kiếm kinh nghiệm	4
2.2.1.1	A* Search	4
2.2.1.2	Greedy Best-first Search	5
2.2.1.3	Beam Search	6
2.2.2	Zobrist Hash	6
2.3	Áp dụng và đánh giá kết quả	7
2.3.1	Greedy Best-first search (9742316/50000000)	7
2.3.1.1	Trạng thái	7
2.3.1.2	Hàm mục tiêu	8
2.3.1.3	Hàm heuristic	8
2.3.1.4	Truy vết đáp án	8
2.3.1.5	Áp dụng	8
2.3.2	Beam Search + Greedy BFS (15491533/50000000)	9
2.3.2.1	Trạng thái, hàm mục tiêu, hàm heuristic và truy vết	9
2.3.2.2	Áp dụng	9
2.3.3	Beam Search + A* (17702996/50000000)	10
2.3.3.1	Trạng thái, hàm mục tiêu, hàm heuristic và truy vết	10
2.3.3.2	Áp dụng	10
2.4	Kết luận và hướng phát triển	11
3	Case Study	11
3.1	starpentagon (Rank #87 - 33497001/50000000)	11
3.1.1	Tổng quan thuật toán	11
3.1.2	Bước 1: Tạo bảng trạng thái cuối cùng	12
3.1.2.1	Simulated Annealing	12
3.1.2.2	Áp dụng	12
3.1.3	Bước 2: Tìm lời giải để tạo ra bảng trạng thái cuối cùng	15
















1 Bài toán

1.1 Tóm tắt

- Đường dẫn bài toán: [Atcoder Heuristic Contest 011 - A. Sliding Tree Puzzle](#)
- Tóm tắt:

Cho số nguyên N ($N \in \mathbb{Z}; 6 \leq N \leq 10$).

Có $N^2 - 1$ ô vuông trên bảng có kích thước $N \times N$ và một ô rỗng, mỗi ô ngoại trừ ô rỗng chứa một hình vẽ với các đường thẳng từ tâm hướng ra một hoặc nhiều trong bốn hướng: lên, xuống, trái và phải. Các ô được biểu diễn dưới dạng ký tự như sau:

Tile																
Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Hex	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f

Hình 1: Các ô vuông được biểu diễn dưới dạng ký tự

Mục tiêu của bài toán là liên tục di chuyển ô trống bốn hướng xung quanh sao cho hình thành được một cây (thành phần liên thông không có chu trình) có kích thước lớn nhất ở trên bảng.

1.2 Đầu vào

Đầu vào của bài toán có dạng:

```

N T
t0,0 ... t0,N-1
⋮
tN-1,0 ... tN-1,N-1

```

Hình 2: Dạng đầu vào của bài toán

Với N là độ dài cạnh của bảng, T là số lần di chuyển tối đa có thể thực hiện. Với tất cả các test case, ta có $T = 2 \times N^3$.

Mảng ký tự hai chiều t thể hiện trạng thái ban đầu của bảng, từng phần tử trong mảng là một ký tự hexa biểu diễn cho từng ô trên bảng (xem Hình 1).

Ví dụ. Ta có input:

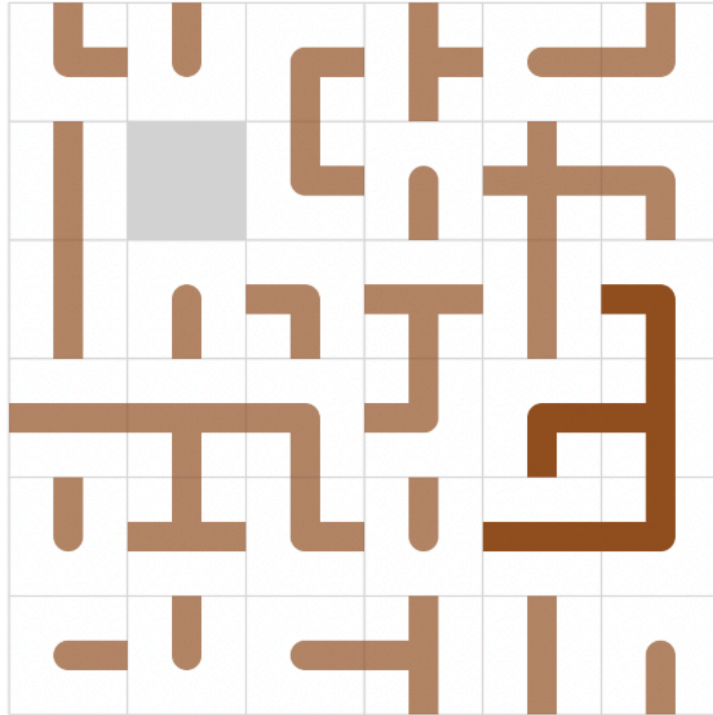
```

6 432
62ce43
a068f9
a89da9
5d93cb
276253
424ba8

```

Hình 3: Ví dụ cụ thể một đầu vào

Được thể hiện bằng hình ảnh như sau:



Hình 4: Biểu diễn trạng thái của bảng của ví dụ đầu vào

1.3 Đầu ra

Đầu ra của bài toán có dạng là một chuỗi ký tự có độ dài không cố định gồm bốn chữ: R, L, U, D thể hiện bốn hướng đi: phải, trái, xuống, lên mà người chơi có thể thực hiện. Đầu ra bài toán thể hiện các bước di chuyển của ô rỗng từ trạng thái ban đầu của bảng do đầu vào cung cấp. Ví dụ:

RRRDLUULDDDDL UUUR

Hình 5: Ví dụ cụ thể một đầu ra

1.4 Các ràng buộc và đánh giá

- **Các ràng buộc:**
 - $N \in \mathbb{Z}; 6 \leq N \leq 10$
 - $T = 2 \times N^3$
 - Giới hạn thời gian: 3 giây
 - Giới hạn bộ nhớ: 1024 MB
 - Đảm bảo rằng với giới hạn không quá T thao tác, luôn có phương án tạo được cây có kích thước $N^2 - 1$ với ô trống ở vị trí $(N - 1, N - 1)$, tức góc phải dưới của bảng.
- **Đánh giá:** Gọi K là số thao tác, S là kích thước cây lớn nhất tìm được, điểm của một test được cho như sau:

- Nếu $S < N^2 - 1$, điểm của test = $\lfloor 500000 * \frac{S}{N^2-1} \rfloor$.
- Nếu $S = N^2 - 1$, điểm của test = $\lfloor 500000 * (2 - \frac{K}{T}) \rfloor$.
- Nếu $K > T$ hoặc thao tác không hợp lệ (không phải R, L, U, D hoặc ô trống ra khỏi bảng), kết quả trả ra *Wrong Answer*.

2 Giải quyết bài toán

2.1 Xem xét bài toán

Sau khi xem xét bài toán, nhóm em đã rút ra được các kết luận chung như sau:

- Bài toán có không gian trạng thái lớn: $|\text{States}| \approx N^2 \times 15^{N^2}$.
- Bài toán có độ phức tạp cao.
- Bài toán giới hạn tài nguyên bộ nhớ và thời gian.
- Bài toán có thể chưa có hoặc không có lời giải tối ưu nhất.

Các kết luận trên đưa nhóm em đến quyết định sử dụng các chiến lược tìm kiếm kinh nghiệm (hay heuristic search) để giải quyết bài toán. Sau đây, nhóm em sẽ trình bày về các cơ sở lý thuyết, đồng thời giải thích chi tiết thuật toán của nhóm em trong việc giải bài toán tối ưu.

2.2 Cơ sở lý thuyết

2.2.1 Tìm kiếm kinh nghiệm

- Tìm kiếm kinh nghiệm là một kỹ thuật giải quyết vấn đề sử dụng kiến thức và kinh nghiệm để đưa ra quyết định trong quá trình tìm kiếm. Kỹ thuật này sử dụng hàm đánh giá (heuristic function) để ước lượng mức độ tiềm năng của một trạng thái dẫn đến giải pháp tối ưu.
- Các chiến lược tìm kiếm nói chung sử dụng hàng đợi ưu tiên để lưu trữ và duyệt các trạng thái, với chỉ số ưu tiên được tính toán sử dụng hàm đánh giá. Các chiến lược khác nhau có các cách thức sắp xếp truy cập hàng đợi ưu tiên, cách thức sử dụng hàm đánh giá khác nhau. Các chiến lược tìm kiếm kinh nghiệm điển hình:

2.2.1.1 A* Search

A Search* có thể nói là chiến lược tìm kiếm kinh nghiệm tốt nhất khi tận dụng cả đường đi thực tế từ trạng thái ban đầu tới trạng thái hiện tại và ước lượng đánh giá từ trạng thái hiện tại tới trạng thái đích. Cụ thể, giả sử trạng thái n là trạng thái hiện tại:

- $f(n)$ là chỉ số ưu tiên của hàng đợi ưu tiên (càng thấp càng có độ ưu tiên cao).
- $g(n)$ là đường đi thực tế từ trạng thái ban đầu tới trạng thái n .
- $h(n)$ là giá trị của hàm đánh giá từ trạng thái n đến trạng thái đích.

Chỉ số hàng đợi ưu tiên được tính theo công thức: $f(n) = g(n) + h(n)$. Để rõ hơn, ta có mã giả như sau:

```
function A_star(trạng_thái_xuất_phát, đích, hàm_đánh_giá):
    var visited := tập rỗng
    var pq := hàng_đội_ưu_tiên(trạng_thái, chỉ_số_ưu_tiên)
    pq.add(trạng_thái_xuất_phát, 0 + hàm_đánh_giá(trạng_thái_xuất_phát))
    while q không phải tập rỗng:
        var trạng_thái_hiện_tại := pq.pop() // lấy phần tử đầu tiên
        if trạng_thái_hiện_tại in visited:
            continue
        if trạng_thái_hiện_tại = đích
            return đường_từ_điểm_xuất_phát_tới_trạng_thái_hiện_tại
        visited.add(trạng_thái_hiện_tại)
        foreach lân_cận in lấy_các_lân_cận(trạng_thái_hiện_tại):
            chỉ_số_ưu_tiên = đường_đi_từ_xuất_phát(lân_cận) +
                           hàm_đánh_giá(lân_cận)
            pq.add(lân_cận, chỉ_số_ưu_tiên)
    return failure
```

2.2.1.2 Greedy Best-first Search

Greedy Best-first Search (hay *Greedy BFS*) là một phương pháp tìm kiếm kinh nghiệm có cách thức sử dụng hàng đợi ưu tiên tương tự như A*, tuy nhiên chiến lược này bỏ qua giá trị đường đi thực từ trạng thái ban đầu mà chỉ sử dụng hàm đánh giá tại trạng thái hiện tại làm chỉ số ưu tiên của từng trạng thái trong hàng đợi.

Chính vì vậy, *Greedy BFS* có thể tìm ra lời giải nhanh hơn so với A*, tuy nhiên lời giải tìm ra không đảm bảo tối ưu.

Ta có mã giả:

```
function greedy(trạng_thái_xuất_phát, đích, hàm_đánh_giá):
    var visited := tập rỗng
    var pq := hàng_đội_ưu_tiên(trạng_thái, chỉ_số_ưu_tiên)
    pq.add(trạng_thái_xuất_phát, 0 + hàm_đánh_giá(trạng_thái_xuất_phát))
    while q không phải tập rỗng:
        var trạng_thái_hiện_tại := pq.pop() // lấy phần tử đầu tiên
        if trạng_thái_hiện_tại in visited:
            continue
        if trạng_thái_hiện_tại = đích
            return đường_từ_điểm_xuất_phát_tới_trạng_thái_hiện_tại
        visited.add(trạng_thái_hiện_tại)
        foreach lân_cận in lấy_các_lân_cận(trạng_thái_hiện_tại):
            chỉ_số_ưu_tiên = hàm_đánh_giá(lân_cận)
            pq.add(lân_cận, chỉ_số_ưu_tiên)
    return failure
```

2.2.1.3 Beam Search

Chiến lược *Beam Search* gần tương tự như *Greedy* về cách thức sử dụng hàm đánh giá và hàng đợi ưu tiên. Điểm khác biệt của *Beam* là hàng đợi ưu tiên chỉ được giới hạn một số lượng trạng thái cố định. Sau khi thêm các trạng thái mới vào hàng đợi, nếu số lượng vượt qua giới hạn, những trạng thái được cho là kém hữu dụng nhất (có chỉ số ưu tiên cao nhất) sẽ bị loại khỏi hàng đợi. Nhờ cơ chế này, các trạng thái được cho là vô dụng sẽ được bỏ qua, giúp bộ nhớ tiêu dùng khi sử dụng Beam được giới hạn và kiểm soát, đồng thời tăng tốc quá trình tìm ra điểm tối ưu. Tuy nhiên trong một số trường hợp, lời giải có thể không được tìm ra do các trạng thái có ích đã bị loại bỏ. Ta có mã giả:

```
function beam(trạng_thái_xuất_phát, đích, hàm_đánh_giá, giới_hạn_hàng_đợi):
    var visited:= tập_rỗng
    var pq:= hàng_đợi_ưu_tiên(trạng_thái, chỉ_số_ưu_tiên, giới_hạn_hàng_đợi)
    pq.add(trạng_thái_xuất_phát, 0 + hàm_đánh_giá(trạng_thái_xuất_phát))
    while q không phải tập_rỗng:
        var trạng_thái_hiện_tại:= pq.pop() // lấy phần tử đầu tiên
        if trạng_thái_hiện_tại in visited:
            continue
        if trạng_thái_hiện_tại == đích:
            return đường_từ_điểm_xuất_phát_tới_trạng_thái_hiện_tại
        visited.add(trạng_thái_hiện_tại)
        foreach lân_cận in lấy_các_lân_cận(trạng_thái_hiện_tại):
            chỉ_số_ưu_tiên = hàm_đánh_giá(lân_cận)
            qState.add(lân_cận, chỉ_số_ưu_tiên)
    return failure
```

2.2.2 Zobrist Hash

Zobrist Hashing là một cấu trúc hàm băm được sử dụng trong các chương trình máy tính chơi các trò chơi trên bảng trừu tượng như cờ vua và cờ Go để triển khai các bảng chuyển vị và tránh phân tích cùng một vị trí nhiều lần.

Zobrist Hashing bắt đầu bằng cách ngẫu nhiên tạo chuỗi bit cho mỗi phần tử có thể của trò chơi trên bảng, tức là cho mỗi kết hợp của một loại ô và một vị trí (trong bài này, đó là 16 loại ô $\times N \times N$ vị trí trên bảng). Bây giờ bất kỳ cấu hình bảng nào cũng có thể được chia thành các thành phần độc lập của loại ô/vị trí, được ánh xạ vào các chuỗi bit ngẫu nhiên đã tạo trước đó. Giá trị băm *Zobrist* cuối cùng được tính bằng cách kết hợp các chuỗi bit đó bằng cách sử dụng XOR bitwise.

Mã giả của Zobrist Hashing:

```

function init_zobrist():
    # fill a table of random numbers/bitstrings
    table := a 3-d array of size N×N×16
    for i from 1 to N:
        for j from 1 to N:      # loop over the board positions
            for k from 1 to 16:  # loop over the tiles
                table[i][j][k] := random_bitstring()

function hash(board):
    h := 0
    for i from 1 to N:
        for j from 1 to N:      # loop over the board positions
            h := h XOR table[i][j][board[i][j]]
    return h

# Recompute the hash when swapping cells (i, j) and (x, y),
# with h being the current hash. (Before actual swapping)
function change_hash(h, i, j, x, y):
    new_h := h

    new_h := new_h XOR table[i][j][board[i][j]]
    new_h := new_h XOR table[x][y][board[x][y]]

    new_h := new_h XOR table[i][j][board[x][y]]
    new_h := new_h XOR table[x][y][board[i][j]]

    return new_h

```

2.3 Áp dụng và đánh giá kết quả

2.3.1 Greedy Best-first search (9742316/50000000)

2.3.1.1 Trạng thái

- Trạng thái gồm:
 - adj[N][N]: Trạng thái của $N \times N$ ô.
 - Sử dụng encoding theo đề bài: 1 ô sẽ lưu 1 số nhị phân gồm 4 bit. Xét từ trái sang phải:
 - Bit thứ 1: Nếu là bit 1 nghĩa là ô có trở xuống dưới. Ngược lại là không có.
 - Bit thứ 2: Nếu là bit 1 nghĩa là ô có trở qua phải. Ngược lại là không có.
 - Bit thứ 3: Nếu là bit 1 nghĩa là ô có trở lên trên. Ngược lại là không có.
 - Bit thứ 4: Nếu là bit 1 nghĩa là ô có trở qua trái. Ngược lại là không có.
 - zobristHash: Giá trị Hash của trạng thái sử dụng Zobrist Hashing
 - emptyx, emptyy: Vị trí của ô trống trong bảng.
 - posInTrace: ID của vị trí tương ứng với trạng thái nằm trong mảng truy vết trace.
 - pathLength: Độ dài của đường đi đến trạng thái hiện tại.

2.3.1.2 Hàm mục tiêu

- Hàm mục tiêu là số lượng đỉnh lớn nhất của cây trong bảng.
- Tính toán:
 - Sử dụng thuật toán DFS để kiểm tra mỗi thành phần liên thông. Nếu thành phần liên thông đó là một cây (hay không có chu trình) thì cập nhật vào biến số lượng đỉnh lớn nhất.
 - Di chuyển trên đồ thị sử dụng mảng `adj[N][N]`. Giả sử nếu ở ô (x, y) và muốn di chuyển xuống ô $(x + 1, y)$ thì ta sẽ xét nếu `adj[x][y]` có bit thứ 1 là bit 1 và `adj[x + 1][y]` có bit thứ 3 là bit 1.

2.3.1.3 Hàm heuristic

Hàm heuristic chính là hàm mục tiêu

2.3.1.4 Truy vết đáp án


- Lưu một mảng động truy vết `trace`. Mỗi phần tử của `trace` là một cặp số nguyên (`parentPosInTrace, direction`).
- Ví dụ: `trace[10] = (2, L)` nghĩa là trạng thái có giá trị `posInTrace = 10` sẽ có trạng thái cha là trạng thái có `posInTrace = 2` và từ trạng thái 2 sang trạng thái 10 sử dụng đi qua trái (L).
- Sau đó khi truy vết, chỉ cần nhảy từ trạng thái tốt nhất về lại trạng thái ban đầu và xây dựng xâu đáp án tương ứng.

2.3.1.5 Áp dụng


- Xem xét phương pháp đánh giá của bài toán, nhóm em nhận thấy yêu cầu giảm thiểu số bước di chuyển chỉ là yêu cầu thứ yếu ($\leq T$) so với yêu cầu chủ yếu là xây dựng được cây có độ dài lớn nhất. Chính vì vậy, việc xem xét số bước di chuyển là không cần thiết, để bài toán có tốc độ hội tụ nhanh hơn, nhóm em ban đầu quyết định bỏ qua chiến lược A^* và sử dụng *Greedy BFS*.
- Link submission: [Greedy BFS submission](#)

Sau khi áp dụng *Greedy BFS*, nhóm em nhận được kết quả như sau:

Submission Info

Submission Time	2024-03-04 18:12:34
Task	A - Sliding Tree Puzzle
User	HynDuf 
Language	C++ 20 (gcc 12.2)
Score	9742316
Code Size	5767 Byte
Status	
Exec Time	2784 ms
Memory	124056 KB

Judge Result

Set Name	test_ALL
Score / Max Score	9742316 / 50000000
Status	 × 50

Hình 6: Kết quả khi sử dụng Greedy BFS

Sau khi sử dụng *Greedy BFS*, số điểm đạt 9742316 điểm

2.3.2 Beam Search + Greedy BFS (15491533/50000000)

2.3.2.1 Trạng thái, hàm mục tiêu, hàm heuristic và truy vết



Tương tự như với *Greedy BFS*.

2.3.2.2 Áp dụng


- Sau khi áp dụng chiến lược *Greedy BFS*, nhóm em nhận thấy kết quả không được thực sự tốt. Vấn đề của *Greedy BFS* là chúng lưu quá nhiều những trạng thái được đánh giá là thừa hay kém hữu dụng vào trong hàng đợi ưu tiên, dẫn đến tốn kém bộ nhớ và mất thời gian trong việc sử dụng hàng đợi, đồng thời khiến quá trình tìm kiếm bị “rộng” hơn, dẫn đến hội tụ lâu hơn. Chính vì vậy nhóm em lựa chọn *Beam Search* để giải quyết vấn đề đó.
- Như đã đề cập ở phần Cơ sở lý thuyết, nhược điểm chính của *Beam Search* là chúng có thể bỏ qua các trạng thái có thể dẫn tới trạng thái đích, dẫn đến có thể tìm được lời giải. Tuy nhiên sau khi nhìn nhận bài toán, nhóm em thấy bài toán gần như không có lời giải tối ưu tuyệt đối, vậy nên áp dụng *Beam Search* cho bài toán này là hoàn toàn hợp lý hơn so với *Greedy BFS*.
- Thay đổi duy nhất so với *Greedy BFS* là thêm việc giới hạn số lượng trạng thái ở trong hàng đợi ưu tiên sau mỗi vòng lặp.
 - Số lượng trạng thái trong hàng đợi ưu tiên = $N \times 100$. (Có thay đổi thành $N \times 50$ và $N \times 150$ nhưng điểm số tệ đi).
- Link submission: [Beam Search + Greedy BFS submission](#)

Kết quả sau khi sử dụng chiến lược *Beam Search + Greedy BFS*:

Submission Info

Submission Time	2024-03-25 16:01:01
Task	A - Sliding Tree Puzzle
User	HynDuf 
Language	C++ 20 (gcc 12.2)
Score	15491533
Code Size	7453 Byte
Status	
Exec Time	2944 ms
Memory	34804 KB

Judge Result

Set Name	test_ALL
Score / Max Score	15491533 / 50000000
Status	 × 50

Hình 7: Kết quả khi sử dụng Beam

Như vậy, sau khi áp dụng ý tưởng *Beam Search* kết hợp với *Greedy BFS*, số điểm tăng lên từ 9742316 điểm lên 15491533 điểm. Đồng thời, bộ nhớ sử dụng cũng tiết kiệm hơn tương đối (từ ≈ 124 MB xuống ≈ 29 MB).

2.3.3 Beam Search + A* (17702996/50000000)

2.3.3.1 Trạng thái, hàm mục tiêu, hàm heuristic và truy vết


Tương tự như với *Greedy BFS*.

2.3.3.2 Áp dụng

- Sau khi áp dụng Beam Search và nhận được kết quả tương đối tốt, nhóm em nhận thấy tốc độ hội tụ đã tăng đáng kể, dựa vào đó có thể đưa ra giả thuyết rằng đã có một số trường hợp thuật toán đã tìm ra được trạng thái tối ưu tuyệt đối (khi độ dài của cây đạt tối đa có thể). Điều đó giúp việc xem xét thêm yếu tố tối thiểu hóa số bước di chuyển trở nên hoàn toàn khả thi. Chính vì vậy, nhóm em quyết định sử dụng A* hay (chính xác hơn là Weighted A*) để xem xét giá trị đường đi thực tế bên cạnh hàm đánh giá.
- Ở đây, nhóm em cài đặt một trọng số (nhóm em chọn 20 (đã thử những tham số khác nhưng điểm cho tệ hơn)) cho giá trị hàm đánh giá để giúp quá trình tìm kiếm hướng về phía tối ưu nhanh hơn. Cụ thể, chỉ số hàng đợi ưu tiên được tính theo công thức: $f(n) = -g(n) + 20 \times h(n)$.
 - $f(n)$ là chỉ số ưu tiên của hàng đợi ưu tiên.
 - $g(n)$ là đường đi thực tế từ trạng thái ban đầu tới trạng thái n.
 - $h(n)$ là giá trị của hàm đánh giá từ trạng thái n đến trạng thái đích.
- Số lượng trạng thái trong hàng đợi ưu tiên $= N \times 100$.
- Link submission: [Beam Search + A* submission](#)

Kết quả sau khi sử dụng chiến lược *Beam Search* + A^* :

Submission Info

Submission Time	2024-03-30 17:11:17
Task	A - Sliding Tree Puzzle
User	HynDuf 
Language	C++ 20 (gcc 12.2)
Score	17702996
Code Size	7623 Byte
Status	AC
Exec Time	2985 ms
Memory	22740 KB

Judge Result

Set Name	test_ALL
Score / Max Score	17702996 / 50000000
Status	AC x 50

Hình 8: Kết quả khi sử dụng Beam + A^*

Sau khi sử dụng *Beam Search* kết hợp với A^* , số điểm đạt 17702996 điểm, tăng hơn 2 triệu điểm so với khi sử dụng *Beam Search* kết hợp với *Greedy BFS*.

2.4 Kết luận và hướng phát triển

Như vậy, sau khi áp dụng các chiến lược tìm kiếm kinh nghiệm để giải quyết bài toán Sliding Tree Puzzle, ta có được kết quả tối đa $\approx 17702996/50000000$. Có thể nói kết quả đã tạm chấp nhận được, tuy nhiên, việc sử dụng chiến lược tìm kiếm kinh nghiệm hoàn toàn có thể hướng đến nhiều cách phát triển trong tương lai:

- Cải thiện hàm đánh giá: Trong lời giải này, nhóm em sử dụng hàm mục tiêu làm hàm đánh giá. Cách thức này tương đối đơn giản và dễ cài đặt. Tuy nhiên, để có được hiệu suất tốt hơn, hàm đánh giá nên được cải thiện nhằm đánh giá tốt hơn một trạng thái, đồng thời tránh hiện tượng tối ưu cục bộ.
- Cài đặt *Hill-Climbing Search*: *Hill-Climbing Search* là một sự kết hợp của phương pháp tìm kiếm theo chiều sâu (DFS) và ứng dụng của hàm đánh giá. Điểm yếu lớn nhất của *Hill-Climbing* chính là việc nó gây hiện tượng tối ưu cục bộ. Tuy nhiên với một hàm đánh giá đã được cải thiện, *Hill-Climbing*, với tốc độ hội tụ cao, hoàn toàn là một phương pháp tìm kiếm nên được cân nhắc.

3 Case Study

3.1 starpentagon (Rank #87 - 33497001/50000000)

3.1.1 Tổng quan thuật toán

Gồm hai bước chính:

- Bước 1: Tạo bảng trạng thái cuối cùng từ bảng ban đầu sử dụng giải thuật *Simulated Annealing*.
- Bước 2: Tìm lời giải sử dụng ít thao tác nhất từ bảng trạng thái ban đầu đến bảng trạng thái tìm được ở **Bước 1**.

Link submission: [Starpentagon's submission](#)

3.1.2 Bước 1: Tạo bảng trạng thái cuối cùng

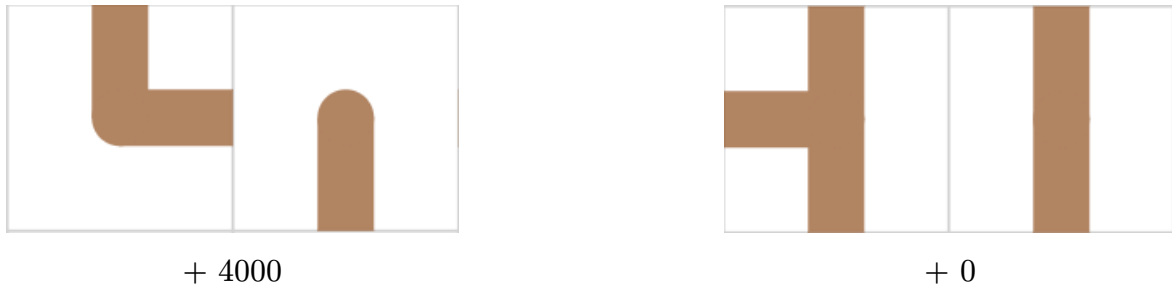
3.1.2.1 Simulated Annealing

- *Simulated Annealing* là sự kết hợp của hai kỹ thuật *Hill-Climbing* và *Pure Random Walk*. Kỹ thuật hill climbing giúp tìm giá trị cực trị toàn cục, còn kỹ thuật pure random walk giúp tăng hiệu quả tìm kiếm giá trị tối ưu.
- Về cơ bản, thay vì chỉ luôn cập nhật trạng thái lân cận được đánh giá nhất hơn như *Hill-Climbing*, *Simulated Annealing* cài đặt một tham số “nhiệt độ”, là một xác suất cho phép chọn những trạng thái được đánh giá thấp hơn, “nhiệt độ” sẽ được giảm dần theo tốc độ tùy chọn trong quá trình tìm kiếm. Yếu tố ngẫu nhiên này khiến *Simulated Annealing* có thể giải quyết vấn đề mắc kẹt tối ưu cục bộ của *Hill-Climbing*.
- Giải thuật của *Simulated Annealing*:
 1. Cài đặt một tham số ‘nhiệt độ’,
 2. Chọn một trạng thái ngẫu nhiên trong không gian trạng thái, đặt là trạng thái hiện tại.
 3. Lấy các trạng thái lân cận của trạng thái hiện tại.
 4. Với từng trạng thái lân cận:
 - Nếu có đánh giá tốt hơn trạng thái hiện tại, cập nhật trạng thái đó thành trạng thái hiện tại.
 - Nếu có đánh giá kém hơn trạng thái hiện tại, tùy thuộc tham số “nhiệt độ”, trạng thái đó vẫn có xác suất cập nhật thành trạng thái hiện tại.
 5. Giảm “nhiệt độ”.
 6. Lặp lại các bước 3-5 cho đến khi đạt đến điều kiện dừng.
 7. Trả ra trạng thái hiện tại.

3.1.2.2 Áp dụng

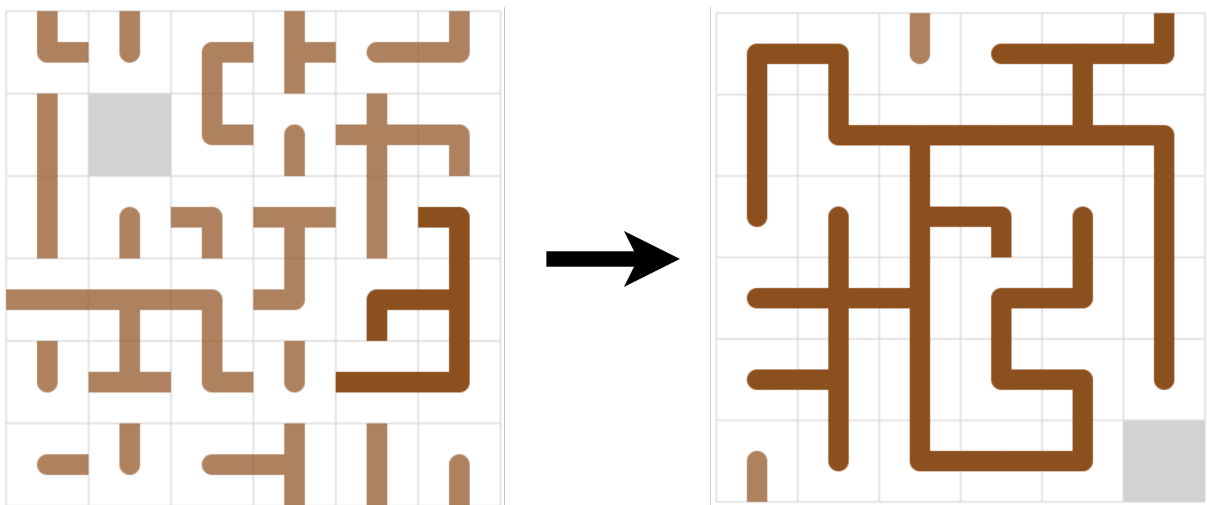
- Đầu tiên, đưa ô trống về góc phải dưới của bảng để dễ xử lý.
- Tại mỗi vòng lặp của *Simulated Annealing*, thuật toán sẽ tìm 2 ô ngẫu nhiên để thực hiện trao đổi sao cho sau khi trao đổi thì 2 ô là 2 ô “tốt”.
 - Một ô (x, y) được gọi là một ô “tốt” nếu không nằm ở viền bảng hoặc nếu nằm ở viền bảng thì không có cạnh dẫn ra ngoài bảng. Ví dụ ô $(0, 1)$, nằm ở viền trái và có cạnh nối qua bên trái, là một ô không “tốt”, do cạnh nối qua bên trái đã ra ngoài bảng và không có cách nào để nối vào đó.

- Sau khi trao đổi, thuật toán sẽ tính “*chi phí*” của cây mới và dựa vào “*chi phí*” của cây trước đó và cây sau khi trao đổi để thực hiện cập nhật.
 - “*Chi phí*” của một cây được tính bằng cách xét mọi cặp ô chung cạnh của bảng. Nếu 1 trong 2 ô có 1 ô hướng vào ô còn lại nhưng ô còn lại thì không hướng vào ô đó thì chi phí tăng thêm 4000 (xem Hình 9). Ngoài ra, với mỗi ô sẽ cộng thêm khoảng cách Manhattan của một ô cùng loại gần ô đó nhất. Điều này đảm bảo, chi phí sẽ ưu tiên bảng gần giống bảng ban đầu nhất.



Hình 9: Ví dụ về cặp ô bị tính chi phí +4000 (bên trái) và cặp ô không bị tính chi phí (bên phải)

- $\Delta = \text{chi_phí_của_cây_trước_khi_tráo} - \text{chi_phí_của_cây_sau_khi_tráo}$. Sử dụng giá trị Δ để cập nhật sao cho phù hợp sử dụng *Simulated Annealing*.
- Trong quá trình chạy các vòng lặp, sẽ đồng thời lưu lại bảng có điểm số lớn nhất (hay kích thước của cây lớn nhất).
- Các tham số:
 - $\text{MAX_TEMPERATURE} = 1000$
 - $\text{MIN_TEMPERATURE} = 500$
 - Số vòng lặp khi chạy *Simulated Annealing*: $\text{SA_ITERATION} = 300000$
- Chúng em có lập trình lại ý tưởng thuật toán này tại: [simulated_annealing.cpp](#)



Hình 10: Từ bảng trạng thái ban đầu tìm ra bảng trạng thái tối ưu

- Mã nguồn của hàm chính của thuật toán *Simulated Annealing*:

```
pair<vector<vector<int>>, int> simulatedAnnealing(vector<vector<int>> adj)
{
    buildManhattanDist(adj);
    // bestAdj with bestScore
    vector<vector<int>> bestAdj;
    int bestScore = getScore(T, adj);
    int maxTemp = 1000;
    int minTemp = 500;

    // cost is for the annealing process
    int curCost = getCost(adj);
    for (int it = 0; it < SA_ITERATION; ++it) {
        double progress = (double) (it + 1) / SA_ITERATION;
        double temp = maxTemp + (minTemp - maxTemp) * progress;

        // Tìm một cặp ô ngẫu nhiên sao cho sau khi trao đổi là các ô "tốt"
        auto [p, q] = getGoodPairOfCells();
        swap(adj[p.fi][p.se], adj[q.fi][q.se]);

        // Tính toán "chi phí" của cây sau khi đổi
        int nxtCost = getCost(adj);
        int delta = curCost - nxtCost;
        bool update = false;
        if (delta < 0) {
            // Công thức của Simulated Annealing
            double probab = exp(delta / temp);
            double rnd = rdReal();
            if (rnd < probab) {
                update = true;
            }
        } else {
            update = true;
        }
        if (!update) {
            swap(adj[p.fi][p.se], adj[q.fi][q.se]);
            continue;
        }

        // Lưu bảng có điểm số tốt nhất (kích thước cây lớn nhất)
        int nxtScore = getScore(T, adj);
        if (bestScore < nxtScore) {
            bestScore = nxtScore;
            bestAdj = adj;
            if (bestScore == 500000) {
                break;
            }
        }
    }
    return {bestAdj, bestScore};
}
```

3.1.3 Bước 2: Tìm lời giải để tạo ra bảng trạng thái cuối cùng

- Sau khi tìm được bảng trạng thái cuối cùng tốt nhất có thể, ta sẽ chuyển từng ô sao cho đúng, bắt đầu từ ô trái trên của hàng đầu tiên đến ô tiếp theo của hàng đầu tiên. Sau khi xong một hàng thì xuống hàng tiếp theo. Khi đó, ô trống sẽ có đường chuyển động như những vòng tròn và khi chuyển một ô về đúng vị trí của nó sẽ không làm ảnh hưởng đến những ô đã được về đúng vị trí (những ô ở hàng trên hoặc cùng hàng nhưng ở bên trái).
- Tuy nhiên, một bất lợi của cách này là số lượng thao tác sử dụng sẽ rất lớn nên tác giả có thực hiện một số tối ưu như quy hoạch động với mỗi hàng có thể thực hiện từ trái sang hoặc từ phải sang. Ngoài ra, ở **Bước 1**, tác giả cũng tối ưu *Simulated Annealing* bằng cách cập nhật nhanh điểm số và chi phí của bảng sau khi trao đổi mà không cần phải đi qua toàn bộ bảng để tính lại.
- Tác giả đạt 33497001/50000000 điểm trên bộ 50 test preliminary và đạt 2105121037/3000000000 điểm trên bộ 3000 test hệ thống, giành thứ hạng #87 trên tổng cộng 926 thí sinh.