**BỘ GIÁO DỤC VÀ ĐÀO TẠO**

**TRƯỜNG ĐẠI HỌC PHENIKAA**

**BÀI TẬP LỚN**

**TÊN HỌC PHẦN: NHẬP MÔN TRÍ TUỆ NHÂN TẠO**

**LỚP: N01**

**HỌC KỲ I NĂM HỌC 2023 - 2024**

**ĐỀ TÀI: Bài toán Ta Canh**

|  |  |  |
| --- | --- | --- |
| Mã Sinh Viên | Họ và Tên | Ngày Sinh |
| 22010168 | Hoàng Vinh Quân | 05/08/2004 |
| 22014063 | Lã Thị Linh | 05/07/2004 |

### 

**Hà Nội, năm 2023**

**MỤC LỤC**

[CHƯƠNG 1. GIỚI THIỆU 5](#_Toc150376427)

[CHƯƠNG 2. MÔ TẢ THUẬT TOÁN 6](#_Toc150376428)

[2.1. Chiến lược tìm kiếm theo bề rộng 6](#_Toc150376429)

[2.2. Chiến lược tìm kiếm theo chiều sâu 7](#_Toc150376430)

[2.3. Chiến lược tìm kiếm với độ sâu lặp 8](#_Toc150376431)

[2.4. Chiến lược tìm kiếm tốt nhất đầu tiên 9](#_Toc150376432)

[2.5. Chiến lược tìm kiếm leo đồi 10](#_Toc150376433)

[2.6. Chiến lược tìm kiếm beam 11](#_Toc150376434)

[CHƯƠNG 3. PHÂN TÍCH HIỆU QUẢ 13](#_Toc150376435)

[3.1 Phương pháp đánh giá 13](#_Toc150376436)

[3.1.1 Breadth-First Search 13](#_Toc150376437)

[3.1.2 Depth-First Search 15](#_Toc150376438)

[3.1.3 Deepening depth-first search 17](#_Toc150376439)

[3.1.4 Best – First – Search 19](#_Toc150376440)

[3.1.5 Hill-climbing search 22](#_Toc150376441)

[3.1.6 Beam Search 24](#_Toc150376442)

[3.2 So sánh hiệu quả 28](#_Toc150376443)

[CHƯƠNG 4. KẾT LUẬN 29](#_Toc150376444)

**MỤC LỤC HÌNH ẢNH**

[Hình 1: Bài toán ta-canh 4x4 7](#_Toc149313241)

**MỤC LỤC BẢNG**

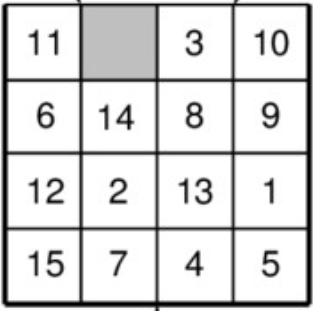
[Bảng 1: So sánh hiệu quả của các thuật toán 25](#_Toc149313479)

**BẢNG CÁC TỪ VIẾT TẮT**

**(Nếu có)**

|  |  |  |
| --- | --- | --- |
| **STT** | **TỪ VIẾT TẮT** | **VIẾT ĐẦY ĐỦ** |
| **1** | **BFS** | **Breadth-First Search** |
| **2** | **DFS** | **Depth-First Search** |
| **3** | **DLS** | **Depth-Limited Search** |

# CHƯƠNG 1. GIỚI THIỆU

Bài toán Ta-canh đã từng là một trò chơi khá phổ biến. Trò chơi bao gồm một hình vuông kích thước 4x4 ô. Có 15 ô có số, mỗi ô có một số từ 1 đến 15. Một ô còn trống. Mỗi lần di chuyển chỉ được di chuyển một ô nằm cạnh ô trống về phía ô trống.

Hình 1: Bài toán ta-canh 4x4

Mục đích: từ một trạng thái ban đầu bất kỳ, làm sao đưa được về trạng thái cuối là trạng thái mà các ô được sắp lần lượt từ 1 đến 15 theo thứ tự từ trái sang phải, từ trên xuống dưới, ô cuối dùng là ô trống.

Cho đến nay, vẫn chưa tìm được một thuật toán chính xác, tối ưu để giải bài toán này. Trong báo cáo này nhóm em sẽ ứng dụng các thuật toán đã học để nghiên cứu bài toán Ta-canh.

Các thuật toán được sử dụng để giải quyết bài toán:

* Chiến lược tìm kiếm theo bề rộng (Breadth-First- Search)
* Chiến lược tìm kiếm theo chiều sâu (Depth-First – Search)
* Chiến lược tìm kiếm với độ sâu lặp ( Depth-Deepening- Search)
* Chiến lược tìm kiếm tốt nhất – đầu tiên ( Best – First – Search)
* Chiến lược tìm kiếm leo đồi (Hill – Climbing – Search)
* Chiến lược tìm kiếm beam( Beam search)

# CHƯƠNG 2. MÔ TẢ THUẬT TOÁN

## 2.1. Chiến lược tìm kiếm theo bề rộng

Chiến lược tìm kiếm theo bề rộng (Breadth-First- Search) sẽ khảo sát không gian theo từng mức. Chỉ đến khi trong một mức cho trước không còn một trạng thái nào để khảo sát thì thuật toán mới chuyển sang mức tiếp theo.

Sử dụng hàng đợi open (có cấu trúc FIFO) để lưu giữ các trạng thái được sinh ra nhưng chưa được khảo sát.

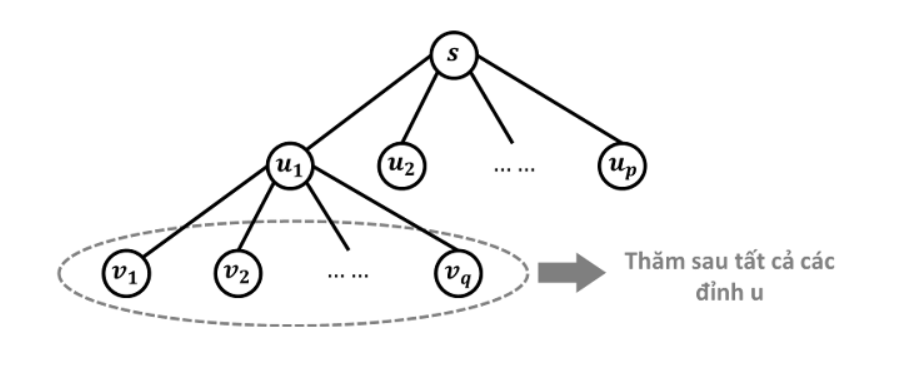
Hàm father(X) dùng để lưu lại cha của đỉnh X trên đường đi.

X là đỉnh Y

Xác định trạng thái bắt đầu S

Xác định tập các trạng thái kết thúc GD

Xác định các toán tử chuyển trạng thái



Hình 2: Breadth-First- Search

**BREADTH-FIRST-SEARCH**

- Đưa trạng thái bắt đầu S vào open;

- loop do

● Nếu open = ∅ thì thông báo Không tìm thấy kết quả. Kết thúc

thuật toán.

● Lấy 1 trạng thái từ hàng đợi open đưa vào biến trạng thái X, sau đó loại bỏ trạng thái đó khỏi hàng đợi.

● Nếu X ∈ GDthì thông báo tìm kiếm thành công. Kết thúc thuật toán.

● Nếu X ∉ GD

- Với mỗi trạng thái Yi được sinh ra bởi trạng thái X thông qua các toán tử

● Đưa trạng thái Yi vào hàng đợi open

● father(Yi) ← X

- end loop

## 2.2. Chiến lược tìm kiếm theo chiều sâu

Chiến lược tìm kiếm theo chiều sâu (Depth-First Search) là khi một trạng thái được xem xét, tất cả các con của nó được xét đến rồi đến các thế hệ sau của các con đó được xem xét ưu tiên trước bất kỳ một trạng thái anh em nào của nó. Tìm kiếm theo chiều sâu sẽ tiến sâu hơn vào trong không gian tìm kiếm khi nào còn có thể. Chỉ khi nào không tìm được các con cháu xa hơn của trạng thái thì mới xem xét đến các trạng thái anh em của nó.

Ta sử dụng stack open (có cấu trúc LIFO) để lưu giữ các trạng thái được sinh ra nhưng chưa được khảo sát.

Hàm father(X) dùng để lưu lại cha của đỉnh X trên đường đi.

X là đỉnh Y

Xác định trạng thái bắt đầu S

Xác định tập các trạng thái kết thúc GD

Xác định các toán tử chuyển trạng thái

* **DEPTH-FIRST-SEARCH**

- Đưa trạng thái bắt đầu S vào open;

- loop do

● Nếu open = ∅ thì thông báo Không tìm thấy kết quả. Kết thúc thuật toán.

● Lấy 1 trạng thái từ stack open đưa vào biến trạng thái X, sau đó loại bỏ trạng thái đó khỏi stack.

● Nếu X ∈ GD thì thông báo tìm kiếm thành công. Kết thúc thuật toán.

● Nếu X ∉ GD

- Với mỗi trạng thái Yi được sinh ra bởi trạng thái X thông qua các toán tử

● Đưa trạng thái Yi vào stack open

● father(Yi) ← X

- end loop

## 2.3. Chiến lược tìm kiếm với độ sâu lặp

* Chiến lược tìm kiếm với độ sâu lặp sử dụng chiến lược tìm kiếm độ sâu giới hạn đến một độ sâu d nào đó. Nếu không tìm ra kết quả, chiến lược tiếp tục tăng độ sâu lên d+1 rồi lại tiếp tục tìm kiếm với độ sâu giới hạn d+1. Quá trình trên được lặp lại cho đến độ sâu max nào đó.
* Xác định độ sâu tối đa MAX

**● DEPTH-DEEPENING-SEARCH**

For d ←0 to MAX do

● DEPTH-LIMITED-SEARCH(d)

● Nếu tìm kiếm thành công thì Kết thúc thuật toán

## 2.4. Chiến lược tìm kiếm tốt nhất đầu tiên

Chiến lược tìm kiếm tốt nhất - đầu tiên (Best-first search) là chiến lược tìm kiếm theo bề rộng và được hướng dẫn bởi hàm đánh giá. Cụ thể hơn, chiến lược này chọn trạng thái để phát triển là trạng thái tốt nhất được xác định bởi hàm đánh giá, trạng thái này có thể nằm ở mức hiện tại hoặc ở mức trên.

● Ta sử dụng danh sách open để lưu giữ các trạng thái được sinh ra nhưng chưa được khảo sát.

● Hàm father(X) dùng để lưu lại cha của đỉnh X trên đường đi.

● Xác định trạng thái bắt đầu S

● Xác định tập các trạng thái kết thúc GD

● Xác định các toán tử chuyển trạng thái

**● BEST-FIRST-SEARCH**

- Đưa trạng thái bắt đầu S vào open;

- loop do

● Nếu open = ∅ thì thông báo Không tìm thấy kết quả. Kết thúc thuật

toán.

● Lấy 1 trạng thái ở đầu danh sách open rồi đưa vào biến trạng thái X,

sau đó loại bỏ trạng thái đó khỏi danh sách.

● Nếu X ∈GD thì thông báo tìm kiếm thành công. Kết thúc thuật toán.

● Nếu X ∉GD

- Với mỗi trạng thái Yi được sinh ra bởi trạng thái X thông qua các toán tử

● Đưa trạng thái Yi vào danh sách open rồi sắp xếp các trạng thái trong danh

sách theo hàm đánh giá từ tốt nhất đến xấu nhất

● father(Yi) ← X

- end loop

## 2.5. Chiến lược tìm kiếm leo đồi

● Chiến lược tìm kiếm leo đồi (Hill-climbing search) là chiến lược tìm kiếm theo chiều sâu và được hướng dẫn bởi hàm đánh giá.

- Cụ thể hơn, trong chiến lược này, khi phát triển một đỉnh u, thì bước tiếp theo ta chọn trong các trạng thái được sinh ra bởi u, trạng thái nào được coi là tốt nhất thì trạng thái đó sẽ được lựa chọn để phát triển tiếp.

● Việc đánh giá các trạng thái được thực hiện thông qua hàm đánh giá.

● Ta sử dụng danh sách open để lưu giữ các trạng thái được sinh ra nhưng chưa được khảo sát.

● Danh sách temp được dùng để lưu giữ các trạng thái được sinh ra bởi một trạng thái nào đó thông qua các toán tử.

● Hàm father(X) dùng để lưu lại cha của đỉnh X trên đường đi.

- Ví dụ: Cú pháp father(X) ← Y tức là đỉnh cha của đỉnh X là đỉnh Y

● Xác định trạng thái bắt đầu S

● Xác định tập các trạng thái kết thúc GD

● Xác định các toán tử chuyển trạng thái

**● HILL-CLIMBING-SEARCH**

- Đưa trạng thái bắt đầu S vào open;

- loop do

● Nếu open = ∅ thì thông báo Không tìm thấy kết quả. Kết thúc thuật

toán.

● Lấy 1 trạng thái ở đầu danh sách open rồi đưa vào biến trạng thái X,

sau đó loại bỏ trạng thái đó khỏi danh sách.

● Nếu X ∈ GD thì thông báo tìm kiếm thành công. Kết thúc thuật toán.

● Nếu X ∉ GD

- Với mỗi trạng thái Yi được sinh ra bởi trạng thái X thông qua các toán tử

● Đưa trạng thái Yi vào danh sách temp rồi sắp xếp các trạng thái trong danh

sách temp theo hàm đánh giá từ tốt nhất đến xấu nhất

● Đưa toàn bộ danh sách temp vào đầu danh sách open

● father(Yi) ← X

- end loop

## 2.6. Chiến lược tìm kiếm beam

● Chiến lược tìm kiếm beam (Beam search) là chiến lược tìm kiếm theo bề rộng và chỉ phát triển k đỉnh tốt nhất ở mỗi mức (các đỉnh tốt nhất này được xác định bởi hàm đánh giá).

- Như vậy, ở bất kỳ mức nào trong tìm kiếm beam cũng chỉ có k đỉnh được phát triển.

● Ta sử dụng danh sách open để lưu giữ các trạng thái được sinh ra nhưng chưa được khảo sát.

● Hàm father(X) dùng để lưu lại cha của đỉnh X trên đường đi.

- Ví dụ: Cú pháp father(X) ← Y tức là đỉnh cha của đỉnh X là đỉnh Y

● Xác định trạng thái bắt đầu S

● Xác định tập các trạng thái kết thúc GD

● Xác định các toán tử chuyển trạng thái

**● BEAM-SEARCH(k)**

- Đưa trạng thái bắt đầu S vào open;

- loop do

● Nếu open = ∅ thì thông báo Không tìm thấy kết quả. Kết thúc thuật toán.

● Lấy min[k, size(open)] trạng thái ở đầu danh sách open rồi đưa vào các

biến trạng thái Xk, sau đó loại bỏ tất cả các trạng thái bên trong danh sách

open để danh sách trở thành rỗng.

● Với mỗi trạng thái Xk

- Nếu Xk ∈ GD thì thông báo tìm kiếm thành công. Kết thúc thuật toán.

- Nếu Xk ∉ GD

● Với tất cả các trạng thái Yi được sinh ra bởi các trạng thái Xk thông qua các toán tử

● Đưa các trạng thái Yi vào danh sách open rồi sắp xếp các trạng thái trong danh

sách open theo hàm đánh giá từ tốt nhất đến xấu nhất

● Xác định father(Yi) ← Xk tương ứng

- end loop

# CHƯƠNG 3. PHÂN TÍCH HIỆU QUẢ

## 3.1 Phương pháp đánh giá

### 3.1.1 Breadth-First Search

**a)Mã giả**

- Bắt đầu chương trình bằng việc bao gồm các thư viện cần thiết và định nghĩa hằng số, cấu trúc dữ liệu, và biến toàn cục.

- Mở tệp input và output để đọc và ghi dữ liệu.

- Định nghĩa các hàm in/output cho bảng trò chơi và đọc dữ liệu đầu vào.

- Hàm "Find" tìm vị trí của ô trống trong bảng trò chơi.

- Định nghĩa vector "Move" chứa các phép di chuyển hợp lệ (lên, xuống, trái, phải).

- Hàm "isSafe" kiểm tra xem một phép di chuyển có hợp lệ không.

- Trong hàm main:

+ Đọc dữ liệu đầu vào và dữ liệu kết thúc từ tệp.

+ Khai báo các biến và cấu trúc dữ liệu cần thiết cho thuật toán tìm kiếm (open, visited, father, Path).

+ Bắt đầu đo thời gian chạy thuật toán.

+ Thêm trạng thái ban đầu vào danh sách "open".

+ Thực hiện tìm kiếm theo chiều rộng (BFS):

- Lặp cho đến khi danh sách "open" trống hoặc tìm thấy trạng thái kết thúc:

+ Lấy trạng thái đầu tiên từ danh sách "open".

+ Tìm vị trí của ô trống trong trạng thái này.

+ Thử tất cả các phép di chuyển hợp lệ và thêm các trạng thái mới vào danh sách "open".

+ Kết thúc đo thời gian chạy thuật toán.

+ Tạo danh sách trạng thái từ trạng thái kết thúc đến trạng thái ban đầu (được lưu trong "father").

+ In ra số bước di chuyển cần thiết để đạt được trạng thái kết thúc và thời gian chạy của thuật toán.

+ In ra các trạng thái trên đường đi từ trạng thái kết thúc đến trạng thái ban đầu

- Kết thúc chương trình.

**b) Nhận xét**

- Nếu tồn tại đường đi từ trạng thái đầu tới trạng thái kết thúc thì thuật toán luôn thực hiện thành công.

- Không gian lưu trữ hết sức tốn kém là vấn đề lớn nhất đối với phương pháp tìm kiếm theo chiều rộng.

**d) Đánh giá**

- Tính đầy đủ: Lời giải bài toán luôn tìm được nếu b (độ phân nhánh của cây) là hữu hạn.

- Độ phức tạp về thời gian:

1+b+b2+...+bd = O(bd)

- Độ phức tạp về không gian lưu trữ: O(bd)

- Tính tối ưu: Thuật toán BFS luôn tìm ra lời giải với ít trạng thái trung gian nhất (đường đi từ trạng thái bắt đầu đến trạng thái kết thúc).

### 3.1.2 Depth-First Search

**a)Mã giả**

- Bắt đầu chương trình bằng việc bao gồm các thư viện cần thiết và định nghĩa hằng số, cấu trúc dữ liệu và biến toàn cục.

- Mở tệp input và output để đọc và ghi dữ liệu.

- Định nghĩa các hàm in/output để làm việc với bảng trò chơi và đọc dữ liệu đầu vào.

- Hàm "Find" tìm vị trí của ô trống trong bảng trò chơi.

- Định nghĩa vector "Move" chứa các phép di chuyển hợp lệ (lên, xuống, trái, phải).

- Hàm "isSafe" kiểm tra xem một phép di chuyển có hợp lệ không.

- Trong hàm main:

+ Đọc dữ liệu đầu vào và dữ liệu kết thúc từ tệp.

+ Khai báo các biến và cấu trúc dữ liệu cần thiết cho thuật toán tìm kiếm (open, visited, father, Path).

+ Bắt đầu đo thời gian chạy thuật toán.

+ Thêm trạng thái ban đầu vào danh sách "open".

+ Thực hiện tìm kiếm theo chiều rộng (BFS):

- Lặp cho đến khi danh sách "open" trống hoặc tìm thấy trạng thái kết thúc:

+ Lấy trạng thái cuối cùng từ danh sách "open" và loại bỏ nó.

+ Kiểm tra xem trạng thái này có phải là trạng thái kết thúc không. Nếu có, thông báo "Tim kiem thanh cong" và kết thúc tìm kiếm.

+ Tìm vị trí của ô trống trong trạng thái này.

+ Thử tất cả các phép di chuyển hợp lệ và thêm các trạng thái mới vào danh sách "open" nếu chưa được duyệt qua.

+ Kết thúc đo thời gian chạy thuật toán.

+ Tạo danh sách trạng thái từ trạng thái kết thúc đến trạng thái ban đầu (được lưu trong "father").

+ In ra màn hình số bước di chuyển cần thiết để đạt được trạng thái kết thúc và thời gian chạy của thuật toán.

+ In ra các trạng thái trên đường đi từ trạng thái kết thúc đến trạng thái ban đầu.

- Kết thúc chương trình.

**b) Nhận xét**

- Nếu tồn tại đường đi từ trạng thái bắt đầu tới trạng thái kết thúc thì thuật toán luôn thực hiện thành công đối với không gian tìm kiếm theo chiều sâu là hữu hạn.

- Lý do, trong trường hợp không gian trạng thái vô hạn thì thuật toán có thể luôn đi sâu xuống mà không đi đúng nhánh chứa trạng thái kết thúc.

**c) Đánh giá**

- Tính đầy đủ: Lời giải bài toán chưa chắc đã tìm được trong trường hợp không gian bài toán là vô hạn.

- Độ phức tạp về thời gian: O(bm)

- Độ phức tạp về không gian lưu trữ: O(b.m)

- Tính tối ưu: Không cho lời giải tối ưu do chưa chắc đã chọn được trạng thái kết thúc gần nhất với trạng thái bắt đầu.

### 3.1.3 Deepening depth-first search

**a)Mã giả**

- Bắt đầu chương trình bằng việc bao gồm các thư viện cần thiết và định nghĩa hằng số, cấu trúc dữ liệu và biến toàn cục.

- Mở tệp input và output để đọc và ghi dữ liệu.

- Định nghĩa các hàm in/output để làm việc với bảng trò chơi và đọc dữ liệu đầu vào.

- Hàm "Find" tìm vị trí của ô trống trong bảng trò chơi.

- Định nghĩa vector "Move" chứa các phép di chuyển hợp lệ (lên, xuống, trái, phải).

- Hàm "isSafe" kiểm tra xem một phép di chuyển có hợp lệ không.

- Hàm "DLS" (Depth-Limited Search) thực hiện tìm kiếm theo chiều sâu với giới hạn độ sâu k:

+ Khai báo các biến và cấu trúc dữ liệu cần thiết cho tìm kiếm.

+ Thêm trạng thái ban đầu vào danh sách "open" và gán độ sâu bằng 0.

+ Lặp cho đến khi danh sách "open" trống:

- Lấy trạng thái cuối cùng từ danh sách "open" và loại bỏ nó.

- Kiểm tra xem độ sâu của trạng thái này có vượt quá giới hạn k không. Nếu có, bỏ qua và tiếp tục với trạng thái tiếp theo.

- Kiểm tra xem trạng thái này có phải là trạng thái kết thúc không. Nếu có, thông báo "Tim kiem thanh cong" và kết thúc tìm kiếm.

- Tìm vị trí của ô trống trong trạng thái này.

- Thử tất cả các phép di chuyển hợp lệ và thêm các trạng thái mới vào danh sách "open" nếu chưa được duyệt qua.

- Trong hàm main:

+ Đọc dữ liệu đầu vào và dữ liệu kết thúc từ tệp.

+ Bắt đầu đo thời gian chạy thuật toán.

+ Duyệt qua các giới hạn độ sâu từ 0 đến MAX và thực hiện tìm kiếm bằng hàm DLS cho mỗi giới hạn.

+ Nếu không tìm thấy giải pháp, thông báo "Tim kiem khong thanh cong".

- Kết thúc chương trình.

**b)Nhận xét**

- Chiến lược này kết hợp được các ưu điểm của các chiến lược

tìm kiếm theo bề rộng và tìm kiếm theo chiều sâu.

● Cũng như tìm kiếm theo bề rộng, tìm kiếm sâu lặp luôn tìm ra kết quả

(nếu bài toán có kết quả), miễn là ta chọn độ sâu MAX đủ lớn.

● Tìm kiếm sâu lặp chỉ cần không gian nhớ như tìm kiếm theo độ sâu.

● Trong tìm kiếm sâu lặp, ta phải phát triển lặp lại nhiều lần cùng một

trạng thái. Điều đó làm cho ta có cảm giác rằng, tìm kiếm sâu lặp lãng

phí nhiều thời gian. Thực ra thời gian tiêu tốn cho phát triển lặp lại các

trạng thái là không đáng kể so với thời gian tìm kiếm theo bề rộng.

**c)** **Đánh giá**

- Tính đầy đủ: Lời giải bài toán luôn tìm được nếu b (độ phân nhánh củ cây) là hữu hạn và ta chọn độ sâu MAX đủ lớn.

- Độ phức tạp về thời gian:

● (d+1)b0 + db1 + (d-1)b2 + …. + bd = O(bd)

- Độ phức tạp về không gian lưu trữ: O(b.d)

- Tính tối ưu: Thuật toán luôn tìm ra lời giải với ít trạng thái trung gian nhất (đường đi từ trạng thái bắt đầu đến trạng thái kết thúc).

### 3.1.4 Best – First – Search

**a)Mã giả**

- Bắt đầu chương trình bằng việc bao gồm các thư viện cần thiết và định nghĩa hằng số, cấu trúc dữ liệu và biến toàn cục.

- Mở tệp input và output để đọc và ghi dữ liệu.

- Định nghĩa các hàm in/output để làm việc với bảng trò chơi và đọc dữ liệu đầu vào.

- Hàm "Find" tìm vị trí của một số u trong bảng trò chơi.

- Định nghĩa vector "Move" chứa các phép di chuyển hợp lệ (lên, xuống, trái, phải).

- Hàm "isSafe" kiểm tra xem một phép di chuyển có hợp lệ không.

- Hàm "h" tính giá trị hàm heuristic (sự sai khác ước tính) giữa trạng thái hiện tại và trạng thái kết thúc.

- Trong hàm main:

+ Đọc dữ liệu đầu vào và dữ liệu kết thúc từ tệp.

+ Khai báo các biến và cấu trúc dữ liệu cần thiết cho thuật toán tìm kiếm (priority\_queue, Path, visited, father).

+ Bắt đầu đo thời gian chạy thuật toán.

+ Thêm trạng thái ban đầu vào hàng đợi ưu tiên "open" với ước tính heuristic h(initial)

+ Thực hiện tìm kiếm theo thuật toán A\*:

- Lặp cho đến khi hàng đợi "open" trống hoặc tìm thấy trạng thái kết thúc:

+ Lấy trạng thái có giá trị hàm heuristic nhỏ nhất từ hàng đợi "open".

+ Kiểm tra xem trạng thái này có phải là trạng thái kết thúc không. Nếu có, thông báo "Tim kiem thanh cong" và kết thúc tìm kiếm.

+ Tìm vị trí của ô trống trong trạng thái này (sử dụng Find với số u là 0).

+ Thử tất cả các phép di chuyển hợp lệ và thêm các trạng thái mới vào hàng đợi ưu tiên "open" nếu chưa được duyệt qua. Giá trị heuristic của các trạng thái mới được tính.

+ Kết thúc đo thời gian chạy thuật toán

+ Tạo danh sách trạng thái từ trạng thái kết thúc đến trạng thái ban đầu (được lưu trong "father").

+ In ra màn hình số bước di chuyển cần thiết để đạt được trạng thái kết thúc và thời gian chạy của thuật toán.

+ In ra các trạng thái trên đường đi từ trạng thái kết thúc đến trạng thái ban đầu.

- Kết thúc chương trình.

**b)Nhận xét:**

- Best-First Search là một thuật toán tìm kiếm thông minh sử dụng heuristics để ước tính khoảng cách từ trạng thái hiện tại đến trạng thái kết thúc. Nó thường hiệu quả hơn tìm kiếm theo chiều rộng truyền thống.

- Nếu tồn tại đường đi từ trạng thái đầu tới trạng thái kết thúc, BFS sẽ tìm ra lời giải. Tuy nhiên, nếu không tồn tại đường đi, thuật toán sẽ không thể tìm ra lời giải.

- BFS sử dụng không gian lưu trữ rất tốn kém, đặc biệt khi cây tìm kiếm có nhiều trạng thái. Điều này có thể gây ra vấn đề lớn về không gian lưu trữ, đặc biệt khi giới hạn tài nguyên của hệ thống hạn chế.

**c) Đánh giá:**

- Tính đầy đủ: BFS luôn tìm ra lời giải nếu tồn tại một đường đi từ trạng thái ban đầu đến trạng thái kết thúc, với điều kiện rằng độ phân nhánh (b) là hữu hạn.

- Độ phức tạp về thời gian: Độ phức tạp thời gian của BFS phụ thuộc vào số lượng trạng thái trong không gian tìm kiếm và độ phân nhánh của cây tìm kiếm. Nếu b là độ phân nhánh và d là độ sâu của trạng thái kết thúc, thì độ phức tạp thời gian là O(b^d). BFS có thể trở nên chậm khi cây tìm kiếm rất sâu.

- Độ phức tạp về không gian lưu trữ: Độ phức tạp về không gian lưu trữ của BFS cũng phụ thuộc vào độ phân nhánh b và độ sâu d. Khi không gian lưu trữ là vấn đề, BFS có thể gặp khó khăn và cần sử dụng nhiều tài nguyên bộ nhớ.

- Tính tối ưu: BFS tìm ra lời giải với ít trạng thái trung gian nhất, tức là tìm đường đi từ trạng thái ban đầu đến trạng thái kết thúc có ít bước nhất. Tuy nhiên, nó không nhất thiết tạo ra lời giải tối ưu nếu hàm heuristic không đảm bảo tính tối ưu.

### 3.1.5 Hill-climbing search

a)Mã giả

- Bắt đầu chương trình bằng việc bao gồm các thư viện cần thiết và định nghĩa hằng số, cấu trúc dữ liệu và biến toàn cục.

- Mở tệp input và output để đọc và ghi dữ liệu.

- Định nghĩa các hàm in/output để làm việc với bảng trò chơi và đọc dữ liệu đầu vào.

- Hàm "Find" tìm vị trí của một số u trong bảng trò chơi.

- Định nghĩa vector "Move" chứa các phép di chuyển hợp lệ (lên, xuống, trái, phải).

- Hàm "isSafe" kiểm tra xem một phép di chuyển có hợp lệ không.

- Hàm "h" tính giá trị hàm heuristic (sự sai khác ước tính) giữa trạng thái hiện tại và trạng thái kết thúc.

- Trong hàm main:

+ Đọc dữ liệu đầu vào và dữ liệu kết thúc từ tệp.

+ Khai báo các biến và cấu trúc dữ liệu cần thiết cho thuật toán tìm kiếm (current\_state, neighbor\_state, current\_h, neighbor\_h).

+ Bắt đầu đo thời gian chạy thuật toán.

+ Lặp cho đến khi không còn cách nào để cải thiện:

- Lưu trạng thái hiện tại vào current\_state và tính giá trị heuristic cho nó (current\_h).

- Khai báo biến boolean "found\_better" và gán giá trị ban đầu là false.

- Lặp qua tất cả các ô trong bảng trò chơi:

+ Thử tất cả các phép di chuyển hợp lệ và tính giá trị heuristic cho trạng thái mới (neighbor\_h).

+ So sánh neighbor\_h với current\_h. Nếu neighbor\_h tốt hơn (nhỏ hơn) current\_h, gán current\_state bằng neighbor\_state, current\_h bằng neighbor\_h, và đặt "found\_better" thành true.

- Nếu không tìm thấy cách nào cải thiện, thoát khỏi vòng lặp

+ Kết thúc đo thời gian chạy thuật toán.

+ In ra màn hình trạng thái cuối cùng (là trạng thái tốt nhất mà thuật toán tìm được) và giá trị heuristic của nó.

+ In ra thời gian chạy của thuật toán.

- Kết thúc chương trình.

b)Nhận xét

- Thành công nếu tồn tại đường đi từ trạng thái đầu tới trạng thái kết thúc: Hill-Climbing Search là một thuật toán tìm kiếm tốc độ cao, và nó có thể thành công nhanh chóng nếu có một giải pháp gần đó từ trạng thái đầu.

- Hạn chế với không gian lưu trữ: Hill-Climbing Search có thể bị kẹt trong cục bộ tối ưu nếu không có cách nào để cải thiện trạng thái hiện tại. Nó có thể dễ dàng rơi vào cực tiểu cục bộ.

c) Đánh giá

- Tính đầy đủ: Hill-Climbing Search không đảm bảo tìm kiếm toàn diện. Nó có thể không tìm ra giải pháp tối ưu nếu bị kẹt trong cực tiểu cục bộ.

- Độ phức tạp về thời gian: Hill-Climbing Search có độ phức tạp thời gian tương đối thấp khi so sánh với nhiều thuật toán tìm kiếm khác. Độ phức tạp thời gian là tuyến tính với số lần di chuyển và số lượng ô trong bảng trò chơi.

- Độ phức tạp về không gian lưu trữ:Hill-Climbing Search yêu cầu ít không gian lưu trữ do nó chỉ duyệt qua một số trạng thái trung gian tương đối nhỏ.

- Tính tối ưu: Hill-Climbing Search không đảm bảo giải pháp tối ưu vì nó có thể kết thúc tại cực tiểu cục bộ mà không khám phá được toàn bộ không gian trạng thái.

### 3.1.6 Beam Search

- Bắt đầu chương trình bằng việc bao gồm các thư viện cần thiết và định nghĩa hằng số, cấu trúc dữ liệu và biến toàn cục.

- Mở tệp input và output để đọc và ghi dữ liệu.

- Định nghĩa các hàm in/output để làm việc với bảng trò chơi và đọc dữ liệu đầu vào.

- Hàm "Find" tìm vị trí của một số u trong bảng trò chơi.

- Định nghĩa vector "Move" chứa các phép di chuyển hợp lệ (lên, xuống, trái, phải).

- Hàm "isSafe" kiểm tra xem một phép di chuyển có hợp lệ không.

- Hàm "h" tính giá trị hàm heuristic (sự sai khác ước tính) giữa trạng thái hiện tại và trạng thái kết thúc.

- Trong hàm main:

+ Đọc dữ liệu đầu vào và dữ liệu kết thúc từ tệp.

+ Khai báo các biến và cấu trúc dữ liệu cần thiết cho thuật toán tìm kiếm (current\_state, neighbor\_states, current\_h, neighbor\_h).

+ Bắt đầu đo thời gian chạy thuật toán.

+ Khởi tạo một hàng đợi ưu tiên (priority\_queue) "open" để lưu trạng thái ban đầu với giá trị heuristic.

+ Lặp cho đến khi tìm kiếm thành công hoặc không còn cách nào để cải thiện:

- Khai báo vector "Xk" để lưu trữ k trạng thái tốt nhất trong hàng đợi "open".

- Lấy ra k trạng thái tốt nhất từ "open" và lưu vào "Xk".

- Làm trống "open".

- Lặp qua tất cả trạng thái trong "Xk":

+ Nếu trạng thái hiện tại là trạng thái kết thúc, đánh dấu thành công và thoát khỏi vòng lặp.

+ Lấy vị trí của ô số 0 (blank) trong trạng thái hiện tại.

+ Lặp qua tất cả các phép di chuyển hợp lệ và tính giá trị heuristic cho các trạng thái mới.

+ So sánh heuristic của trạng thái mới với heuristic của trạng thái hiện tại. Nếu tốt hơn, thêm trạng thái mới vào "open", đánh dấu trạng thái cha và tương ứng với nó.

- Nếu đã tìm kiếm thành công, thoát khỏi vòng lặp chính.

+ Kết thúc đo thời gian chạy thuật toán.

+ In ra màn hình trạng thái cuối cùng (là trạng thái tốt nhất mà thuật toán tìm được) và giá trị heuristic của nó.

+ In ra thời gian chạy của thuật toán.

- Kết thúc chương trình.

b)Nhận xét:

- Beam Search là một biến thể của thuật toán tìm kiếm theo chiều rộng (BFS) với sự giới hạn về không gian lưu trữ.

- Nếu tồn tại đường đi từ trạng thái ban đầu tới trạng thái kết thúc và không gian lưu trữ đủ lớn để chứa toàn bộ trạng thái, thuật toán Beam Search sẽ thực hiện thành công.

- Không gian lưu trữ có thể là vấn đề lớn nhất với phương pháp này, vì việc duyệt nhiều trạng thái cùng một lúc (k trạng thái) có thể tạo áp lực lên bộ nhớ.

- Beam Search không đảm bảo tính tối ưu và có thể bỏ sót các lời giải tối ưu nếu không duyệt đủ trạng thái.

c)Đánh giá:

- Tính đầy đủ: Beam Search sẽ tìm được lời giải nếu có đường đi từ trạng thái ban đầu tới trạng thái kết thúc và không gian lưu trữ đủ lớn để chứa toàn bộ trạng thái. Tuy nhiên, nó không đảm bảo tính đầy đủ nếu không gian lưu trữ bị giới hạn (với k trạng thái tốt nhất)

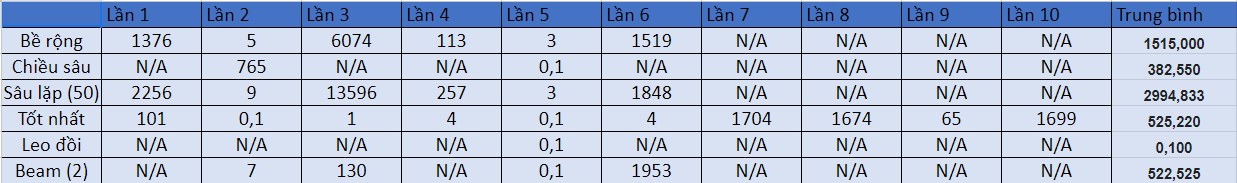
- Độ phức tạp về thời gian: Độ phức tạp thời gian của Beam Search phụ thuộc vào số lượng trạng thái tối ưu (b) và độ sâu của cây tìm kiếm (d). Trong trường hợp tốt nhất, nó có thể tìm kiếm một lời giải trong thời gian tương đối ngắn. Tuy nhiên, trong trường hợp xấu nhất, độ phức tạp có thể là O(b^d).

- Độ phức tạp về không gian lưu trữ: Beam Search yêu cầu lưu trữ k trạng thái tốt nhất tại mỗi bước. Do đó, không gian lưu trữ cần thiết là O(k), với k là số lượng trạng thái được duyệt qua tại mỗi bước. Tổng không gian lưu trữ sẽ phụ thuộc vào độ sâu tối đa của cây tìm kiếm (d).

- Tính tối ưu: Beam Search không đảm bảo tính tối ưu vì nó có thể bỏ sót các lời giải tối ưu nếu không duyệt đủ trạng thái. Điều này là do nó giới hạn không gian lưu trữ và tập trạng thái được duyệt.

## 3.2 So sánh hiệu quả

Bảng 1: So sánh hiệu quả của các thuật toán



# CHƯƠNG 4. KẾT LUẬN

Như vậy, mỗi một phương pháp đều có ưu và nhược điểm riêng. Với các dạng

bài toán tìm kiếm trên không gian trạng thái, có hai trường hợp cần tới heuristic:

• Những vấn đề không thể có nghiệm chính xác do các mệnh đề không phát biểu

chặt chẽ hay do thiếu dữ liệu để khẳng định kết quả.

• Những vấn đề có nghiệm chính xác nhưng chi phí tính toán để tìm ra nghiệm là

quá lớn (dẫn đến bùng nổ tổ hợp).

-Heuristic giúp ta tìm kiếm đạt kết quả với chi phí thấp hơn. Và như trong bài

Ta Canh này thì thuật toán Best – First – Search là giải quyết được yêu cầu bài

toán hiệu quả nhất.

**DANH MỤC TÀI LIỆU THAM KHẢO**

1. Trần Đức Minh, *Giáo trình nhập môn trí tuệ nhân tạo*, Đại học Phenikaa.