Link Video:

https://youtu.be/go1CI25xJDo

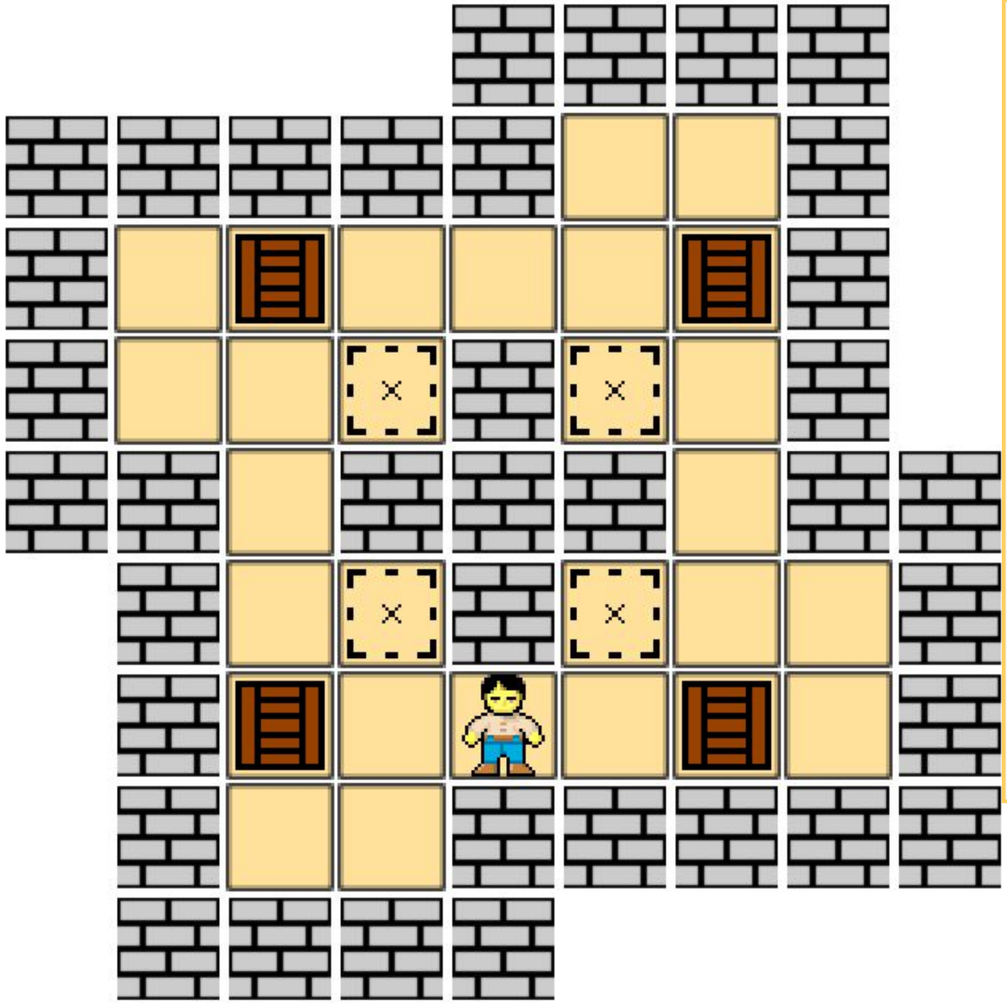
SOKOBAN GAME

Lớp: L02

Lê Gia Huy - 1910202

Nguyễn Duy Khang - 1910238

GIỚI THIỆU CHUNG VỀ BÀI TOÁN



- **Sokoban** là một trò chơi điện tử của Nhật Bản thuộc loại câu đố về sự di chuyển (transport puzzle).
- Giải thuật triển khai:
  - Breadth first search (BFS).
  - A Star search (A*).

CÁC VẤN ĐỀ ĐẶT RA CỦA BÀI TOÁN

INPUT CỦA BÀI TOÁN

Wall: #

Player: @

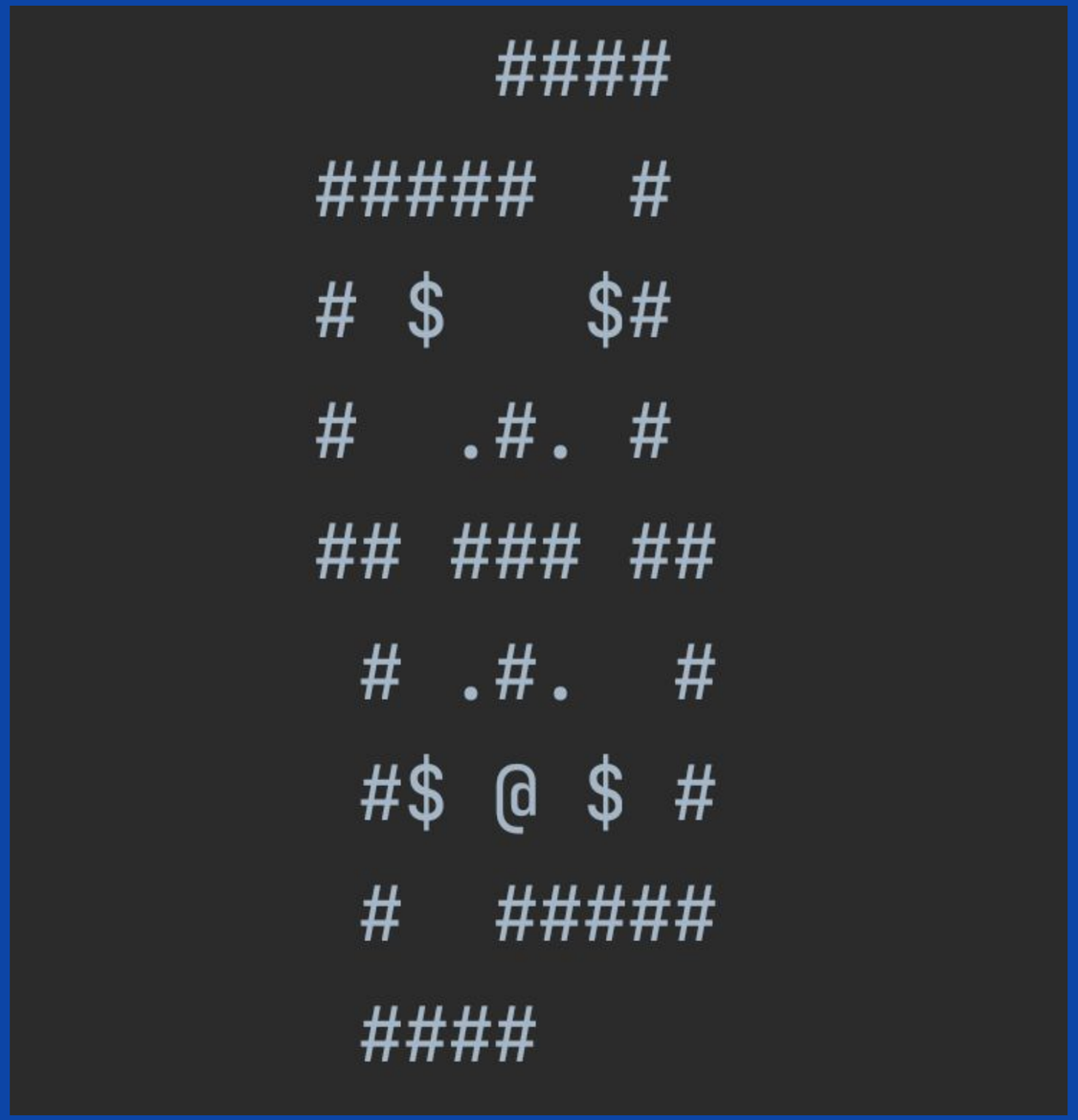
Player on goal square: +

Box $

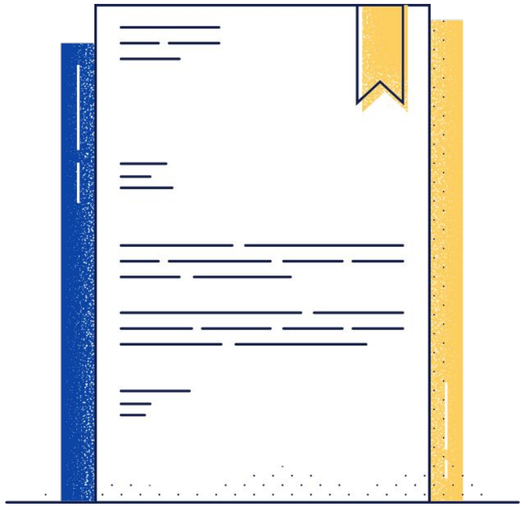
Box on goal square: *

Goal square: .

Floor: (Space)



TRẠNG THÁI (STATE) CỦA BÀI TOÁN



- Theo dõi vị trí của player và các boxes để thực hiện các bước chuyển trạng thái.
- Kiểm tra xem đã đến trạng thái cuối cùng hay chưa.

- Tạo ra một đối tượng (object) để lưu trữ vị trí hiện tại của player và các boxes.

VẤN ĐỀ VỀ CÁC BƯỚC DI CHUYỂN HỢP LỆ

Một box không thể được đẩy vào ô gây ra trạng thái **deadlock.**

Player chỉ được cho phép di chuyển một box tại một thời điểm, không thể di chuyển hai hay nhiều boxes một cách đồng thời.

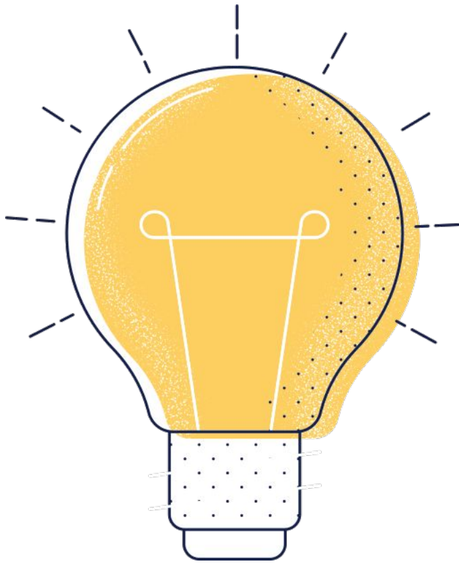
Phải đảm bảo đủ không gian để di chuyển các boxes.

OUTPUT CỦA BÀI TOÁN

- Là một danh sách các nước đi có dạng ‘L', ‘R', ‘U', ‘D' tương ứng với các nước di chuyển qua trái, phải, lên, xuống.

Ngoài ra, ở đây nhóm còn cho ra các output sau:

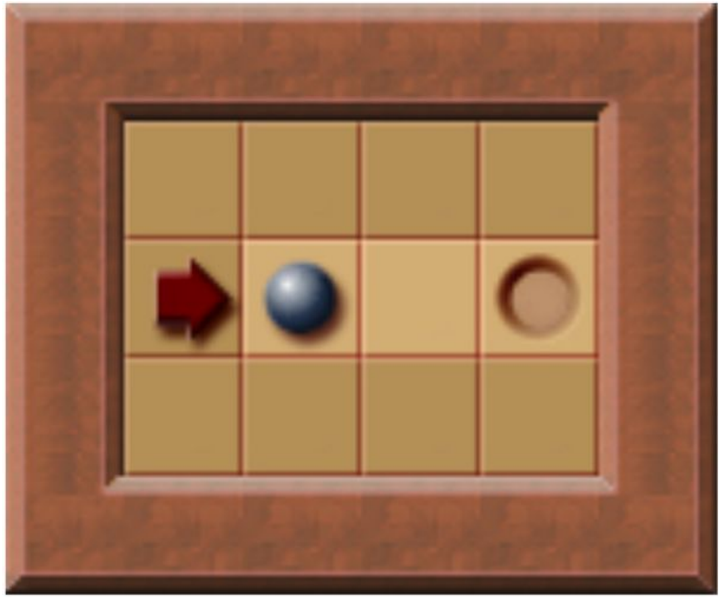
- Output về tổng số node được tìm thấy.
- Output về số node được lấy ra khỏi hàng đợi để thực hiện tìm kiếm.



GIẢI QUYẾT DEADLOCK

Khi rơi vào trạng thái **deadlock** thì mọi trạng thái lúc sau của giải thuật tìm kiếm được sinh ra (expanded) từ trạng thái đó đều trở nên vô nghĩa do chắc chắn từ nó không bao giờ đi được đến trạng thái kết thúc (goal state).

SIMPLE DEADLOCK



- Là tình trạng khi một box được đẩy (push) vào một ô bất kỳ thì không thể nào di chuyển box đó đến mục tiêu được nữa.
- Vị trí của các ô gây ra simple deadlock không bao giờ thay đổi trong suốt quá trình chơi.

GIẢI QUYẾT SIMPLE DEADLOCK

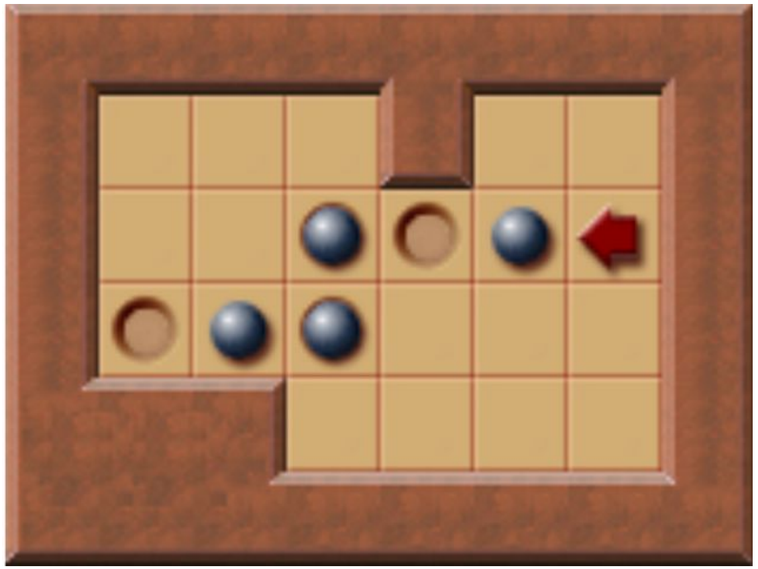
- Tạo một ma trận tương tự với ma trận đầu vào, đánh dấu tất cả là True.
- Với mỗi vị trí mục tiêu (goal square) ta thực hiện các bước sau:
  - Xóa tất cả các boxes của ma trận (Xem như không tồn tại các boxes trong ma trận đầu vào).
  - Đặt 1 box tại vị trí mục tiêu đó.
  - Từ vị trí đó, ta kéo (Pull) box đó đến tất cả các vị trí có thể được của mê cung và đánh dấu lại các vị trí đó là False (không có simple deadlock)

GIẢI QUYẾT SIMPLE DEADLOCK

- **Sau khi bước trên đã được hiện thực cho tất cả các mục tiêu, ta biết được rằng các ô đã được đánh dấu là False thì từ ô đó, ta chắc chắn có thể đẩy được các box đến một trong các mục tiêu.**

Hiện thực của giải thuật phát hiện deadlock trên được hiện thực qua hàm **has_simple_deadlock()** trong source code.

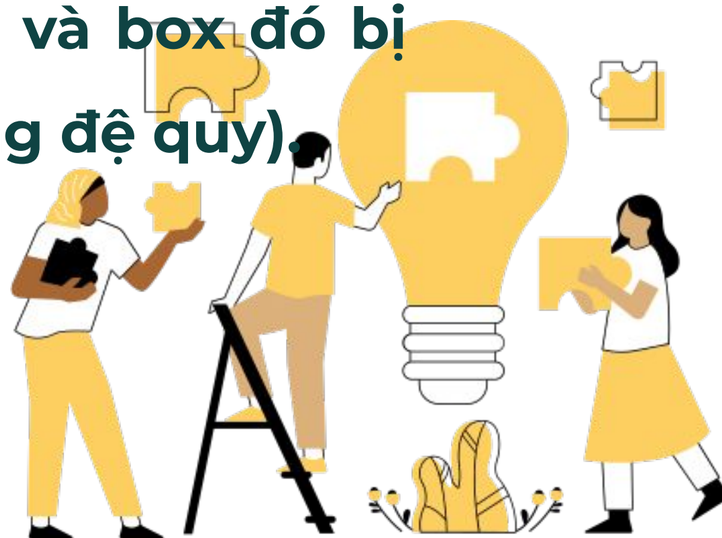
FREEZE DEADLOCK



- Là tình trạng mà các boxes không thể di chuyển được nữa => các box đó bị block
- Nếu một box không thể di chuyển được nữa mà đang trong một ô không phải là vị trí mục tiêu => cả trạng thái hiện tại bị “đóng băng" (freeze deadlock)

GIẢI QUYẾT FREEZE DEADLOCK

- Box bị block nếu một trong các điều kiện sau thỏa:
  - Box bị block theo chiều ngang khi một trong các điều kiện sau thỏa mãn:
    - Có một tường chắn bên trái hoặc bên phải box --> box bị block
    - Tồn tại simple deadlock ở cả hai phía (trái và phải) của box → box bị block
    - Nếu có một box ở trái hoặc phải của box hiện tại và box đó bị block → box hiện tại bị block (kiểm tra điều này bằng đệ quy).
  - Kiểm tra trường hợp box bị block theo chiều dọc tương tự như trên.



GIẢI QUYẾT FREEZE DEADLOCK

- **Nếu box bị block và tồn tại một box bị block trên mê cung hiện tại không ở vị trí mục tiêu → Cả trạng thái hiện tại bị freeze deadlock.**

  - Hiện thực của giải thuật phát hiện deadlock trên được hiện thực qua hàm **has_freeze_deadlock()** trong source code.

NHẬN XÉT

Giải quyết deadlock cải tiến được thời gian thực thi và số node duyệt qua các giải thuật tìm kiếm gấp nhiều lần.(đối với cả hai thuật toán).

Có những level ban đầu không thể giải được (do tìm kiếm rất lâu) thì sau khi giải quyết deadlock thì thời gian thực thi giảm xuống còn khoảng 5 giây.

HIỆN THỰC CÁC GIẢI THUẬT

PHƯƠNG THỨC HIỆN THỰC CÁC GIẢI THUẬT

Cả hai giải thuật đều được hiện thực dựa trên giải thuật **Graph Search**, nghĩa là ta sẽ không thực hiện duyệt lại các trạng thái đã được phát hiện (explored).

Dùng Graph Search vì việc di chuyển trong mê cung sẽ có rất nhiều lần ta có thể quay lại trạng thái đã khám phá ra

MỘT SỐ THUẬT NGỮ ĐƯỢC DÙNG TRONG CÁC GIẢI THUẬT VÀ HIỆN THỰC PYTHON

state: Một đối tượng gồm:

+ Một tuple chứa vị trí hiện tại của player

+ Một set của các tuple là các vị trí hiện tại của các boxes.

frontier: Một hàng đợi queue (có thể là FIFO queue hoặc Priority queue) được dùng để lưu các trạng thái trong quá trình search.

expand: là quá trình tìm ra các trạng thái có thể sinh ra từ trạng thái hiện tại và đưa vào hàng đợi frontier.

Closed set: chứa tất cả các trạng thái đã được explored (gồm tất cả các trạng thái còn trong frontier và các trạng thái đã được dequeue khỏi frontier).

explored node: Tất cả các trạng thái đã được tìm thấy trong quá trình search bao gồm trong hàng đợi và đã được dequeue ra khỏi hàng đợi.

expanded node: Tất cả các trạng thái đã được dequeue khỏi hàng đợi để thực hiện expand.

PSEUDOCODE CỦA CÁC GIẢI THUẬT ĐƯỢC SỬ DỤNG

```
Input: layout, initial_state        Output: the result path to finish this level map
function BFS(layout, initial_state):
    frontier ← empty FIFO queue
    Push initial_state to frontier
    closed_set ← empty set
    Add intial_state to closed_set
    while frontier is not empty:
        current_state ← head of frontier
        if current state is goal state:
            return solution
        Pop head node from frontier
        For each neighbor of current_state:
            if this neighbor is not in closed_set:
                Add this neighbor to closed set
                Push this neighbor to frontier
    return solution not found
```

THUẬT TOÁN BFS

Quá trình lặp qua các neighbor của trạng thái hiện tại hiện tại là quá trình expand.

THUẬT TOÁN A*

Đi tìm hàm lượng giá (Evaluation function):

Trong không gian hai chiều, cho hai điểm A(x1, y1) và B(x2, y2) thì **manhattan distance** giữa hai điểm được định nghĩa là:

$$d = |x1 - x2| + |y1 - y2|$$

```
Input: node, goal    Output: Manhattan distance of node and goal
function manhattan(node, goal):
    return abs(node.x - goal.x) + abs(node.y - goal.y)
```

THUẬT TOÁN A*

Hàm heuristic **h(state)**: ta sẽ áp dụng Manhattan distance vào heuristic function.

```
Input: state    Output: evaluated cost from current state to goal state
function h(state):
    sum ← 0
    for box in all box positions:
        sum = sum + minimum value of manhattan(box, goal) for goal in all goal positions
    return sum
```

THUẬT TOÁN A*

Hàm **g(state)**: trả về giá trị g-value là cost đi từ trạng thái ban đầu đến trạng thái hiện tại.

```
Input: state    Output: cost from initial state to current state
function g(state):
    return g(ancestor_state) + 1
```

THUẬT TOÁN A*

Hàm lượng giá **f(state)**: trả về giá trị f-value dùng để đánh giá độ ưu tiên cho các trạng thái lấy ra từ priority queue. Giá trị f-value càng nhỏ thì độ ưu tiên của trạng thái càng cao.

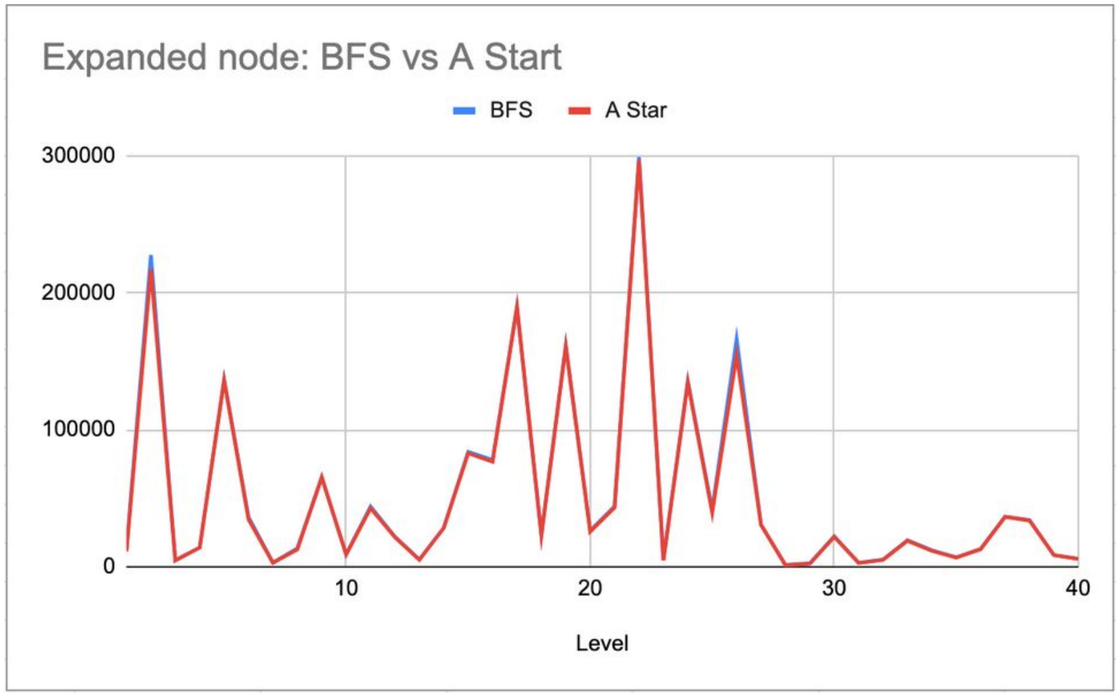
```
Input: state    Output: f-value
function f(state):
    return g(state) + h(state)
```

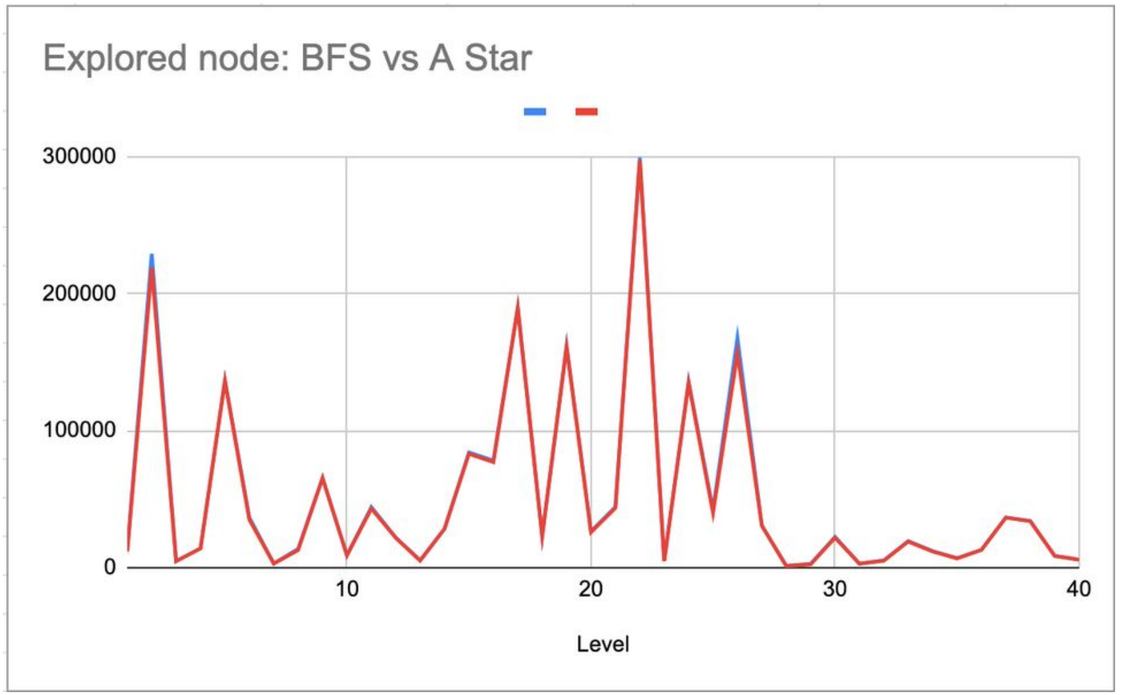
```
Input: layout, initial_state        Output: the result path to finish this level map
function AStar(layout, initial_state):
    frontier ← empty Priority queue
    Push initial_state to frontier
    closed_set ← empty set
    Add intial_state to closed_set
    while frontier is not empty:
        current_state ← node with highest priority (lowest f-value)
        if current state is goal state:
            return solution
        Pop node with highest priority node from frontier
        For each neighbor of current_state:
            if this neighbor is not in closed_set:
                Add this neighbor to closed set
                Push this neighbor to frontier
            else if this new g-value less than current g-value of this neighbor:
                Update g-value of this neighbor
                Update f-value of this neighbor
                Update parent of this neighbor
                If this neighbor is not in frontier:
                    Push this neighbor to frontier
    return solution not found
```

THUẬT TOÁN TÌM KIẾM A*

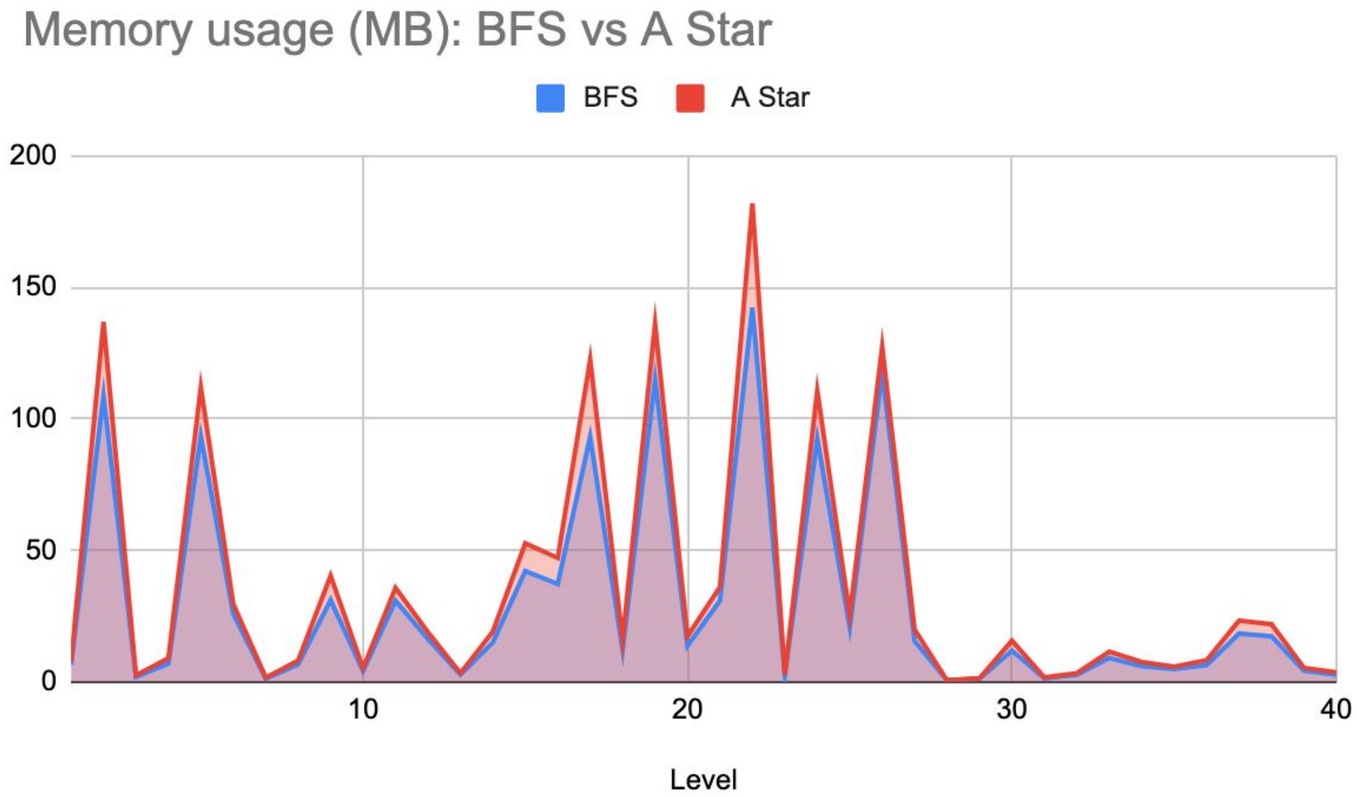
ĐÁNH GIÁ HAI THUẬT TOÁN

Thuật toán BFS có số expanded node và explored node đều lớn hơn so với thuật toán A* do không sử dụng hàm định giá.

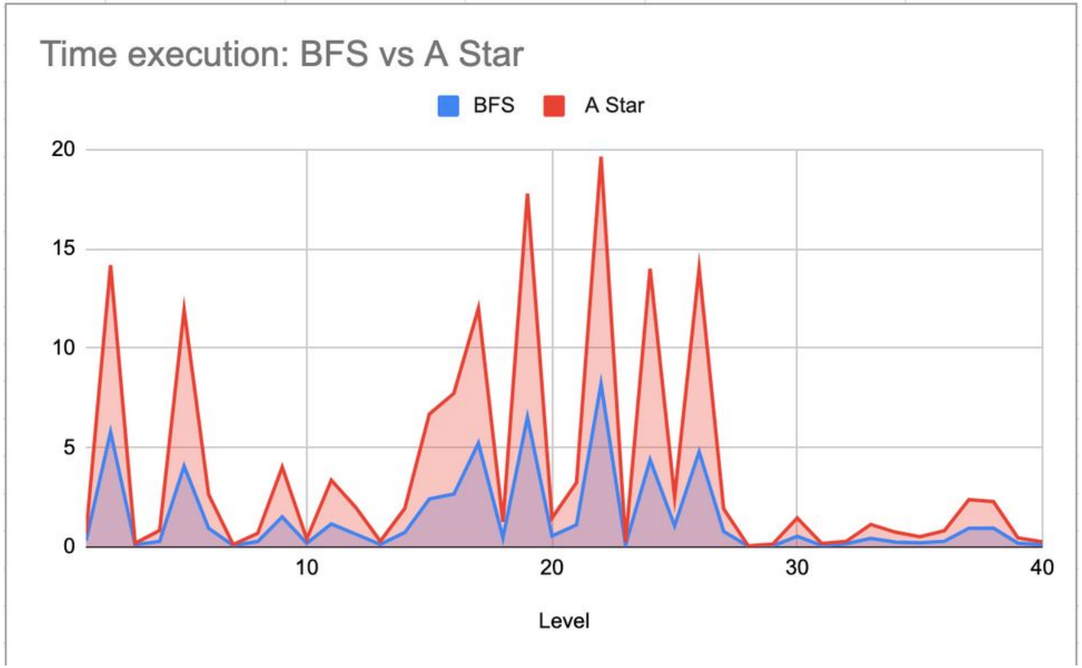




Memory usage của A Star nhiều hơn so với BFS



Thời gian của A* có phần chậm hơn BFS



- So sự chênh lệch giữa số lượng expanded node và explored node không quá lớn.
- A* sử dụng Priority queue có độ phức tạp của các operation push và pop là O(logn) => Số lượng node chênh lệch ít sẽ không bù đắp được các chi phí thực thi chương trình.
- Heuristic function được thiết kế không được tốt lắm => số lượng node chênh lệch ít.

Tuy nhiên, cả hai giải thuật đều cho một optimal solution (con đường ngắn nhất để đi đến trạng thái kết thúc) với cùng độ dài, điều này là đúng với lý thuyết.

Thank You