

TRƯỜNG ĐẠI HỌC PHENIKAA
KHOA CÔNG NGHỆ THÔNG TIN



Nhập môn Trí tuệ nhân tạo (N01)

Bài tập lớn

Chương trình tự giải Sokoban

Giảng viên hướng dẫn: TS. Trần Đức Minh

Sinh viên: Hà Tài Thanh (MSV: 22010392)

Nguyễn Dương Tuấn Nguyễn (MSV: 22010091)

HA NOI CITY, NOVEMBER 2023

Mục lục

1	Giới thiệu chung	4
1.1	Sản phẩm	4
1.1.1	Thư mục thành phần	4
1.1.2	Thư viện	4
1.1.3	Chạy chương trình	4
1.1.4	Thao tác	4
1.2	Bài toán	5
1.2.1	Mục tiêu	5
1.2.2	Thành phần	6
1.2.3	Toán tử	7
1.2.4	Trạng thái thất bại	9
1.3	Thuật toán	9
1.3.1	Tìm kiếm mù	9
1.3.2	Tìm kiếm Heuristic	9
2	Thiết kế chương trình	10
2.1	Cấu trúc dữ liệu	10
2.2	Toán tử	11
2.3	Thuật toán	12
2.3.1	Breadth First Search	12
2.3.2	A*	14
2.3.3	Best First Search	17
2.4	Hàm đánh giá	17
2.4.1	Cơ sở lý thuyết	17
2.4.2	Code	19
2.5	Mẫu thử	20
2.6	Giao diện	21
3	Thống kê	22
3.1	Thời gian	23
3.2	Bộ nhớ	25
3.3	Trạng thái	27
3.3.1	Trạng thái cần duyệt	27
3.3.2	Trạng thái sử dụng - Số bước di chuyển	29

Danh sách hình vẽ

1.1	Mục tiêu bài toán	5
1.2	Sang trái	7
1.3	Sang phải	7
1.4	Đi lên	8
1.5	Đi xuống	8
1.6	Minh họa các trạng thái thất bại	9
3.1	Biểu đồ Thời gian chạy thuật toán	24
3.2	Biểu đồ dung lượng tiêu tốn	26
3.3	Biểu đồ Số trạng thái cần duyệt	28
3.4	Biểu đồ Số bước nhân vật di chuyển	30

Danh sách bảng

1.1	Danh sách đối tượng thành phần	6
2.1	Danh sách các biến và cấu trúc dữ liệu quan trọng	10
2.2	Định dạng ký tự đại diện vật thể	20
3.1	Thời gian chạy thuật toán	23
3.2	Dung lượng bộ nhớ tiêu tốn	25
3.3	Số trạng thái cần duyệt	27
3.4	Số bước nhân vật di chuyển	29

1 Giới thiệu chung

1.1 Sản phẩm

Toàn bộ sản phẩm của nhóm được lưu trữ tại link github: https://github.com/HTT5104/Teemo_Sokoban.git

1.1.1 Thư mục thành phần

Sources: Lưu trữ file code Python xây dựng chương trình tự giải bài toán Sokoban.

Testcases: Lưu trữ file txt mô hình bản đồ của 20 màn chơi.

Checkpoints: Lưu trữ file txt tọa độ của các điểm đích của 20 màn chơi.

Assets: Lưu trữ các thành phần cho hiển thị ứng dụng.

1.1.2 Thư viện

Để chạy được chương trình, ta cần cài thêm một số thư viện ngoài thư viện mặc định của Python bằng cách chạy các lệnh dưới đây trong terminal:

- **Numpy:** pip install numpy
- **Pygame:** pip install pygame
- **Memory_profiler:** pip install memory_profiler

1.1.3 Chạy chương trình

- **Sử dụng IDE:**

Mở thư mục ../Teemo_Sokoban/Sokoban_with_Teemo/Sources

Run file main.py

- **Sử dụng terminal:**

cd ../Teemo_Sokoban/Sokoban_with_Teemo/Sources

python main.py

1.1.4 Thao tác

Sau khi chạy chương trình, ta có thể đổi level và đổi thuật toán chạy:

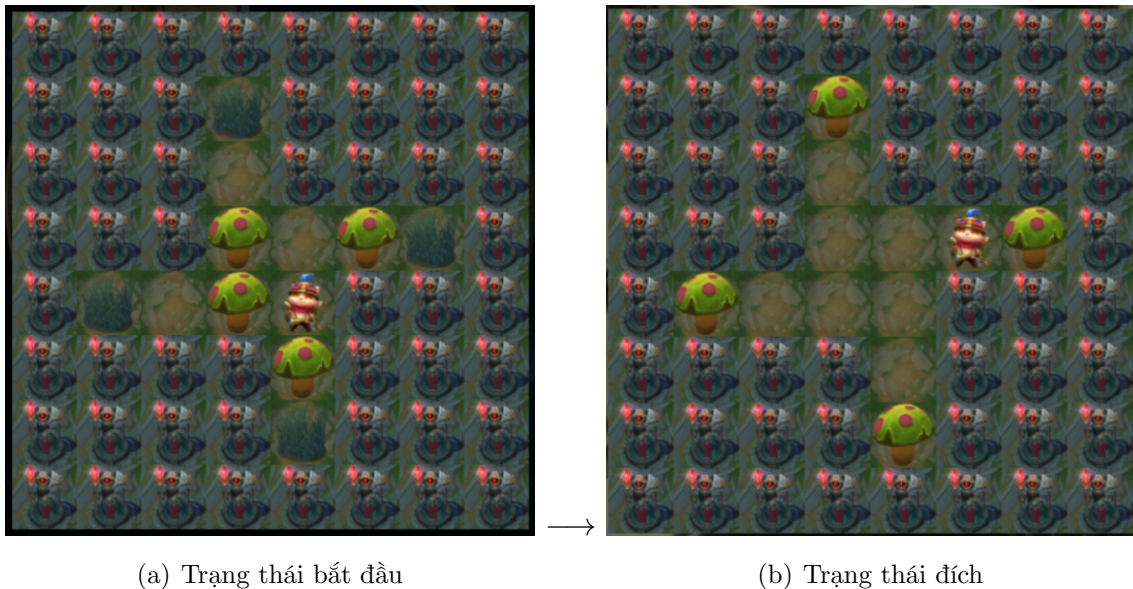
- Đổi level: Sử dụng nút Trái hoặc Phải để thay đổi level giảm hoặc tăng. Các level được sắp xếp theo thứ tự từ 1 đến 20.

- Đổi thuật toán: Sử dụng nút Space để thay đổi thuật toán. Thứ tự các thuật toán là Breadth First Search, A* và Best First Search.
- Sau khi chọn level và thuật toán, nhấn Enter để chạy thuật toán. Thời gian chạy được giới hạn trong 30 phút. Sau khi có kết quả, màn hình sẽ hiển thị từng bước giải.

1.2 Bài toán



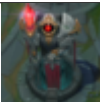

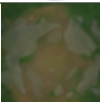
1.2.1 Mục tiêu

Trạng thái chiến thắng (Trạng thái đích) của trò chơi Sokoban là khi điều khiển nhân vật để đẩy các thùng (Cây nấm) đến vị trí đích (Bụi cỏ).



Hình 1.1: Mục tiêu bài toán

1.2.2 Thành phần

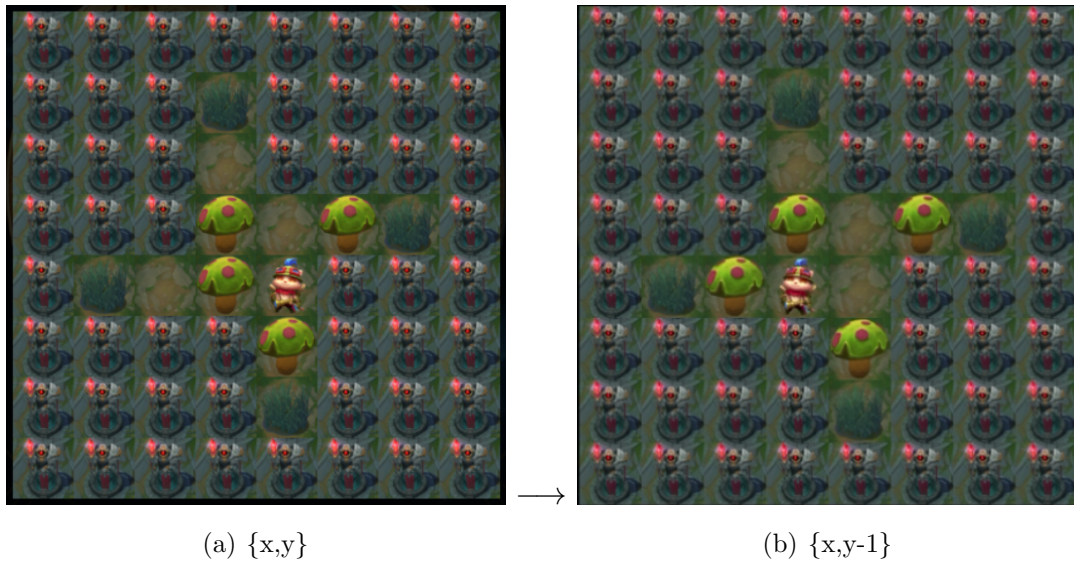
STT	Đối tượng	Hình ảnh	Mô tả
1	Nhân vật		Hình tượng nhân vật Teemo: Chủ thể để người chơi điều khiển thông qua phím điều hướng của bàn phím.
2	Thùng		Hình tượng Cây nấm độc của Teemo: Vật thể được nhân vật tác động để thay đổi vị trí. Cần đẩy tất cả vật thể đến vị trí đích để chiến thắng.
3	Tường		Hình tượng Turret: Vật cản của bản đồ, nhân vật và thùng không thể ở trên tường.
4	Đích		Hình tượng Bụi cỏ: Vị trí mà thùng cần được đẩy tới.
5	Đường đi		Nhân vật và thùng có thể di chuyển trên các ô này.

Bảng 1.1: Danh sách đối tượng thành phần

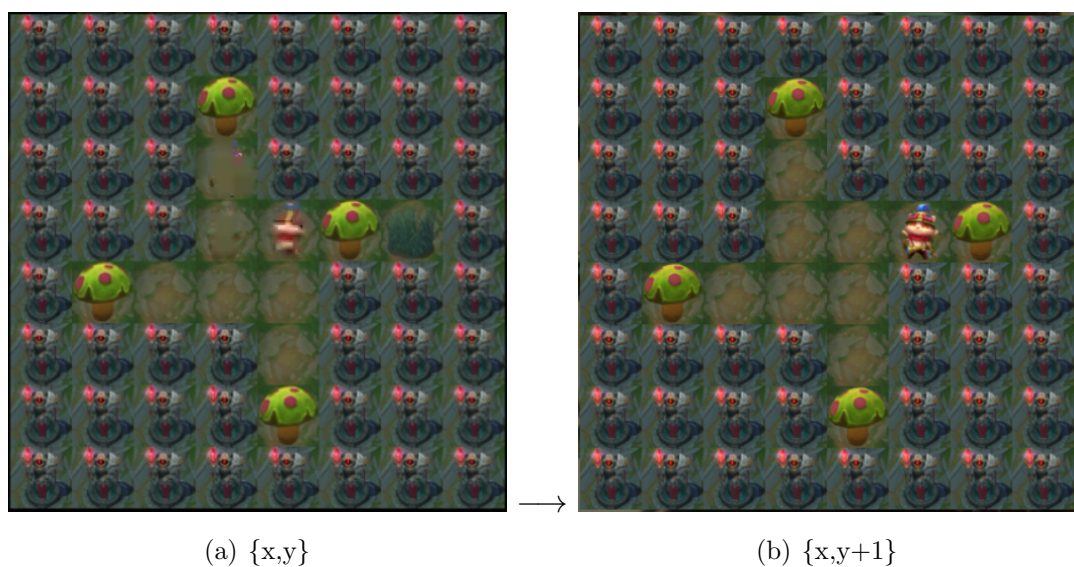
1.2.3 Toán tử

Bài toán có 4 toán tử làm thay đổi vị trí của nhân vật, từ đó làm thay đổi vị trí của các đối tượng khác trên bản đồ.

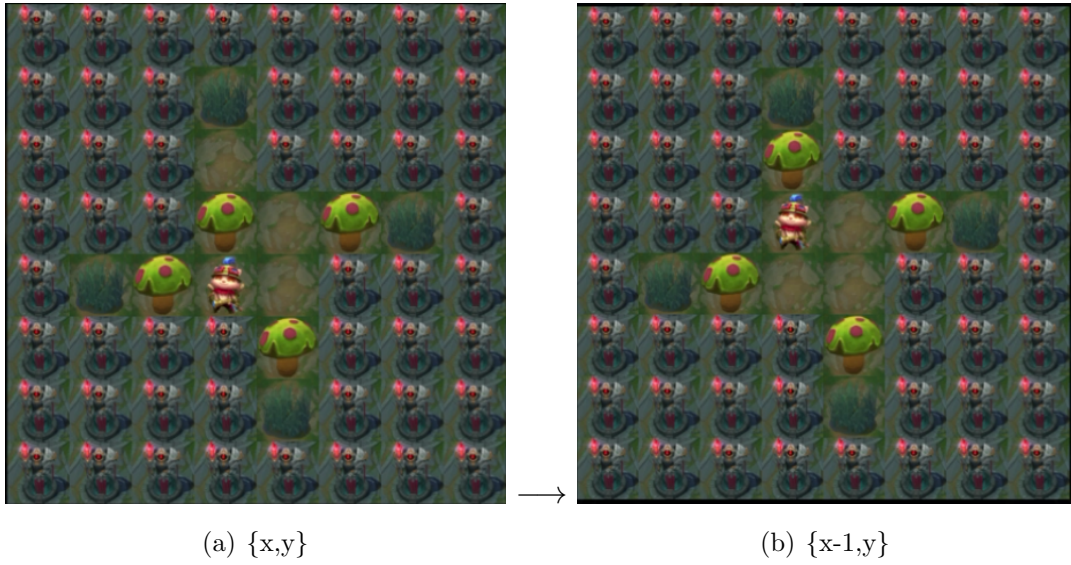
Bản đồ là một ma trận $p \times q$.



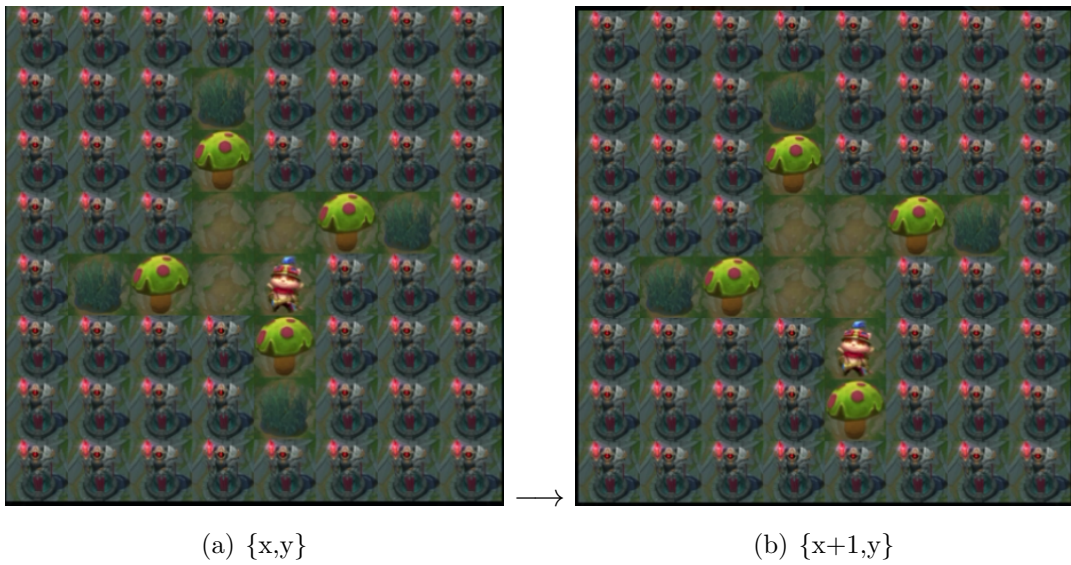
Hình 1.2: Sang trái



Hình 1.3: Sang phải



Hình 1.4: Đi lên



Hình 1.5: Đi xuống

1.2.4 Trạng thái thất bại

Bài toán sẽ không thể giải tiếp và dẫn tới thất bại khi và chỉ khi nhân vật đẩy một thùng vào vị trí không phải vị trí đích mà có ít nhất hai cạnh liền kề của thùng đó bị chặn.



(a) Trạng thái bắt đầu

(b) Thùng bị kẹt góc

(c) Hai thùng chặn nhau

Hình 1.6: Minh họa các trạng thái thất bại

1.3 Thuật toán

1.3.1 Tìm kiếm mù

Breadth First Search: Thuật toán duyệt các trạng thái theo chiều rộng của không gian trạng thái. Thứ tự duyệt là tuần tự theo thứ tự trạng thái bắt gặp trên không gian nên độ ưu tiên của chúng là như nhau.

1.3.2 Tìm kiếm Heuristic

- Best First Search: Thuật toán duyệt các trạng thái theo thứ tự ưu tiên. Thứ tự này được xác định thông qua một hàm đánh giá chi phí dự kiến từ trạng thái hiện tại tới trạng thái đích.
- A*: Bản chất là thuật toán Best First Search nhưng phức tạp hơn khi A* không chỉ xét chi phí dự kiến từ trạng thái hiện tại tới trạng thái đích, mà còn xét chi phí thực tế từ trạng thái bắt đầu tới trạng thái hiện tại.

2 Thiết kế chương trình

2.1 Cấu trúc dữ liệu

STT	Tên	Kiểu dữ liệu	Cấu trúc
1	board	list[list]	Mà trận
2	new_board	list[list]	Mà trận
3	start_state	state	Class tự tạo
4	now_state	state	Class tự tạo
5	new_state	state	Class tự tạo
6	state_parent	state	Class tự tạo
7	box_positions	list	List
8	list_can_move	list	List
9	list_check_point	list	List
10	num_of_box	int	Số nguyên
11	list_state	list[state]	Mà trận dữ liệu
12	list_visit	list[state]	Mà trận dữ liệu
13	cur_pos	tuple[int][int]	Vector hai chiều
14	next_pos	list[int][int]	Vector hai chiều
15	heuristic_queue	PriorityQueue	Min Heap
16	list_board	list tuple	Mà trận dữ liệu

Bảng 2.1: Danh sách các biến và cấu trúc dữ liệu quan trọng

2.2 Toán tử

Như đã giới thiệu trước đó, Sokoban có 4 toán tử là Sang trái, Sang phải, Đi lên và Đi xuống. Thêm vào đó, bản đồ của chương trình được mô hình bằng một ma trận $p \times q$ nên tọa độ được lưu chính là chỉ số hàng và chỉ số cột của ma trận.

Do đó, việc xác định vị trí tiếp theo nhân vật có thể di chuyển sẽ tuân theo nguyên tắc tọa độ của ma trận (Vị trí gốc (0,0) ở trên cùng bên trái, đi xuống và bên phải các chỉ số tăng) chứ không phải trục tọa độ (Vị trí gốc (0,0) ở giữa, đi lên và sang phải chỉ số tăng, đi xuống và sang trái chỉ số giảm). Điều này đã được thể hiện rõ ở hàm sau:

```
1 def get_next_pos(board, cur_pos):
```

Trong đó, x và y gán lần lượt là tọa độ hàng và tọa độ cột hiện tại của nhân vật.

```
1 x, y = cur_pos[0], cur_pos[1]
```

Nếu thỏa mãn các điều kiện ràng buộc gồm:

- Vị trí đi tới không phải tường.
- Nếu vị trí tới là thùng, Thì vị trí sau nữa khi tiếp tục sử dụng cùng toán tử phải là Đường đi hoặc Điểm đích.

Thì toán tử sẽ được xác nhận bằng việc cho thêm tọa độ nhân vật sau khi thực hiện toán tử vào danh sách chờ thực hiện:

- Đi lên

```
1 list_can_move.append((x - 1, y))
```

- Đi xuống

```
1 list_can_move.append((x + 1, y))
```

- Sang trái

```
1 list_can_move.append((x, y - 1))
```

- Sang phải

```
1 list_can_move.append((x, y + 1))
```

2.3 Thuật toán

Chương trình giải Sokoban này ban đầu được xây dựng dựa trên hai thuật toán là Breath First Search và A*. Từ đó, dựa trên A*, nhóm phát triển thêm thuật toán Best First Search.

2.3.1 Breadth First Search

Ý tưởng thuật toán:

Thành phần:

- Hàng đợi open: Lưu trữ các trạng thái được sinh ra nhưng chưa được duyệt.
- Hàm father(X): Lưu trữ cha của X.
- Trạng thái S: Một trạng thái bắt đầu của bài toán.
- Trạng thái GD: Một hoặc nhiều trạng thái đích của bài toán.

Triển khai:

Cho S vào open.

Lặp {

Nếu open rỗng \implies Không tìm thấy kết quả \implies Kết thúc thuật toán.

Gán X = Trạng thái đầu tiên trong open \implies Loại bỏ trạng thái đó khỏi open.

Nếu X \in GD \implies Tìm kiếm thành công.

Nếu X \notin GD \implies Cho các trạng thái Y sinh ra bởi X vào cuối open
 \implies father(Y) = X.

}

Ánh xạ sang code, ta có:

Thành phần:

- Hàng đợi open \iff List list_visit.
- Hàm father(X) \iff Biến State state_parent \implies Hàm X.get_line(): Mở rộng hơn của hàm father(X) khi X.get_line tra ra danh sách các trạng thái từ trạng thái xuất phát tới trạng thái X.

```
1 def get_line(self):  
2     '''use loop to find list board from start to this  
    state'''
```

```
3     if self.state_parent is None:
4         return [self.board]
5     return (self.state_parent).get_line() + [self.board]
```

- Trạng thái $S \iff$ Biến State `start_state`.
- Trạng thái GD \iff Các trạng thái mà tất cả các điều đang nằm trên điểm đích.
- Mở rộng:
 - List `list_state` dùng để lưu trữ các trạng thái đã được duyệt.
 - Trạng thái đang xét $X \iff$ Biến State `now_state`.
 - Trạng thái Y sinh ra bởi $X \iff$ Biến `new_board` ứng với mỗi biến `next_pos` (thể hiện vị trí của nhân vật) \in List `list_can_move`.

Triển khai:

Hàm `BFS_search`.

```
1 def BFS_search(board, list_check_point):
```

Khởi tạo biến `start_state`.

```
1 start_state = spf.state(board, None, list_check_point)
```

Thêm `start_state` vào `list_state` (do `start_state` chắc chắn được duyệt ở bước sau) và `list_visit`.

```
1 list_state = [start_state]
2 list_visit = [start_state]
```

Lặp khi `list_visit` có phần tử.

```
1 while len(list_visit) != 0:
```

Khởi tạo biến `now_state` (trạng thái X đang xét) là phần tử đầu tiên của `list_visit` và loại bỏ phần tử khỏi `list_visit`.

```
1 now_state = list_visit.pop(0)
```

Tạo List `list_can_move` để lưu trữ các tọa độ của nhân vật trong trạng thái Y được sinh ra bởi `now_state`.

```
1 list_can_move = spf.get_next_pos(now_state.board, cur_pos)
```

Với mỗi phần tử của `list_can_move`, khởi tạo một ma trận `new_board` tương ứng với bản đồ của trạng thái Y,

```
1 for next_pos in list_can_move:
2     ''' MAKE NEW BOARD '''
3     new_board = spf.move(now_state.board, next_pos, cur_pos,
        list_check_point)
```

Với mỗi `new_board` không đưa vào toán vào trạng thái thất bại, khởi tạo trạng thái `new_state`.

```
1 new_state = spf.state(new_board, now_state, list_check_point)
```

Nếu `new_state` là trạng thái đích, hàm `BFS_search` trả về danh sách các bản của các trạng thái được chọn và số lượng trạng thái đã duyệt.

```
1 if spf.check_win(new_board, list_check_point):
2     print("Found win")
3     print("The number of state visited:", len(list_state))
4     print("The number of step:", len(new_state.get_line())-1)
5     return (new_state.get_line(), len(list_state))
```

Ngược lại, thêm `new_state` vào cuối `list_state` và `list_visit` rồi lặp đến khi tìm được trạng thái đích hoặc timeout.

```
1 list_state.append(new_state)
2 list_visit.append(new_state)
```

2.3.2 A*

Ý tưởng thuật toán:

Thành phần:

- Hàng đợi open: Lưu trữ các trạng thái được sinh ra nhưng chưa được duyệt.
- Hàm `father(X)`: Lưu trữ cha của X.
- Trạng thái S: Một trạng thái bắt đầu của bài toán.
- Trạng thái GD: Một hoặc nhiều trạng thái đích của bài toán.
- Hàm `g(X)`: Chi phí thực tế đi từ trạng thái S đến trạng thái X.

- Hàm $k(X, Y)$: Chi phí thực tế đi từ trạng thái X đến trạng thái Y .
- Hàm $h(X)$: Chi phí ước tính đi từ trạng thái X đến trạng thái GD.
- Hàm $f(X) = g(X) + h(X)$.

Triển khai:

Cho S vào open.

Lặp {

Nếu open rỗng \implies Không tìm thấy kết quả \implies Kết thúc thuật toán.

Gán $X =$ Trạng thái đầu tiên trong open \implies Loại bỏ trạng thái đó khỏi open.

Nếu $X \in \text{GD}$ \implies Tìm kiếm thành công.

Nếu $X \notin \text{GD}$ \implies Xét các trạng thái Y sinh ra bởi X :

$g(Y) = g(X) + k(X, Y)$.

$f(Y) = g(Y) + h(Y)$.

Cho các trạng thái Y vào open.

$\text{father}(Y) = X$.

Sắp xếp open theo hàm $f()$.

}

Ánh xạ sang code, ta có:

Thành phần:

- Hàng đợi open \iff PriorityQueue heuristic_queue.
- Hàm $\text{father}(X)$ \iff Biến State state_parent \implies Hàm $X.\text{get_line}()$: Mở rộng hơn của hàm $\text{father}(X)$ khi $X.\text{get_line}$ tra ra danh sách các trạng thái từ trạng thái xuất phát tới trạng thái X .

```
1 def get_line(self):
2     '''use loop to find list board from start to this
   state'''
3     if self.state_parent is None:
4         return [self.board]
5     return (self.state_parent).get_line() + [self.board]
```

- Trạng thái S \iff Biến State start_state .
- Trạng thái GD \iff Các trạng thái mà tất cả các điều đang nằm trên điểm đích.

- Hàm $k(X, Y) \iff$ Biến `int cost = 1`.
- Hàm $g(X)$, $h(X)$, $f(X)$ tương ứng với các thành phần của hàm đánh giá `compute_heuristic` (chi tiết tại mục Hàm đánh giá).
- Mở rộng:
 - List `list_state` dùng để lưu trữ các trạng thái đã được duyệt.
 - Trạng thái đang xét $X \iff$ Biến `State now_state`.
 - Trạng thái Y sinh ra bởi $X \iff$ Biến `new_board` ứng với mỗi biến `next_pos` (thể hiện vị trí của nhân vật) \in List `list_can_move`.

Triển khai:

Hàm `AStar_Search`.

```
1 def AStar_Search(board, list_check_point):
```

Khởi tạo biến `start_state`.

```
1 start_state = spf.state(board, None, list_check_point)
```

Thêm `start_state` vào `list_state` (do `start_state` chắc chắn được duyệt ở bước sau) và hàng đợi ưu tiên `heuristic_queue`.

```
1 list_state = [start_state]
2 ''' INITIALIZE PRIORITY QUEUE '''
3 heuristic_queue = PriorityQueue()
4 heuristic_queue.put(start_state)
```

Lặp khi `heuristic_queue` có phần tử.

```
1 while not heuristic_queue.empty():
```

Khởi tạo biến `now_state` (trạng thái X đang xét) là phần tử đầu tiên của `heuristic_queue` và loại bỏ phần tử khỏi `heuristic_queue`.

```
1 now_state = heuristic_queue.get()
```

Tạo List `list_can_move` để lưu trữ các tọa độ của nhân vật trong trạng thái Y được sinh ra bởi `now_state`.

```
1 list_can_move = spf.get_next_pos(now_state.board, cur_pos)
```

Với mỗi phần tử của `list_can_move`, khởi tạo một ma trận `new_board` tương ứng với bản đồ của trạng thái Y,

```
1 for next_pos in list_can_move:
2     ''' MAKE NEW BOARD '''
3     new_board = spf.move(now_state.board, next_pos, cur_pos,
        list_check_point)
```

Với mỗi `new_board` không đưa vào toán vào trạng thái thất bại, khởi tạo trạng thái `new_state`.

```
1 new_state = spf.state(new_board, now_state, list_check_point)
```

Nếu `new_state` là trạng thái đích, hàm `AStar_search` trả về danh sách các bản của các trạng thái được chọn và số lượng trạng thái đã duyệt.

```
1 if spf.check_win(new_board, list_check_point):
2     print("Found win")
3     print("The number of state visited:", len(list_state))
4     print("The number of step:", len(new_state.get_line())-1)
5     return (new_state.get_line(), len(list_state))
```

Ngược lại, thêm `new_state` vào cuối `list_state` và `heuristic_queue` rồi lặp đến khi tìm được trạng thái đích hoặc timeout.

```
1 list_state.append(new_state)
2 heuristic_queue.put(new_state)
```

2.3.3 Best First Search

Như đã trình bày ở phần trước, thuật toán A^* chính là Best First Search với một hàm đánh giá phức tạp hơn. Do đó, phần mô tả thuật toán và ánh xạ sang code của Best First Search cũng giống như A^* .

2.4 Hàm đánh giá

2.4.1 Cơ sở lý thuyết

Trước hết, chúng ta cần xác định rõ mục tiêu của bài toán là đẩy tất cả thùng vào vị trí đích thông qua việc di chuyển nhân vật. Do đó, giá trị cần được đánh giá cho trạng thái ở đây là chi phí để đẩy từng thùng tới từng điểm đích. Vì chi phí cho mỗi di chuyển

(toán tử) là 1 đơn vị nên chi phí cần đánh giá trên chính là khoảng cách từ nhân vật tới thùng và từ thùng tới điểm đích.

Dựa vào tọa độ của các vật thể, ta có thể công thức hóa mô tả trên thành hàm f :

$$\begin{aligned} f &= \left| \sum_{i=1}^n \sum_{j=1}^2 ((pos_j - box_{ij}) + (box_{ij} - point_{ij})) \right| \\ &= \left| \sum_{i=1}^n \sum_{j=1}^2 (pos_j - point_{ij}) \right| \end{aligned}$$

Trong đó:

- pos_j : Tọa độ của nhân vật.
- box_{ij} : Tọa độ thùng thứ i .
- $point_{ij}$: Tọa độ điểm đích thứ i .
- $j = 1$ là tọa độ hàng; $j = 2$ là tọa độ cột.

Nhìn vào công thức trên, có thể thấy yếu tố vị trí của thùng đã bị triệt tiêu. Điều này phủ nhận đi lập luận ban đầu trong mô tả. Do đó công thức này là không phù hợp.

Vì chi phí cho mỗi nước đi là 1 đơn vị, là đơn vị nhỏ nhất trong bài toán nên việc thêm bớt các thành phần của hàm đánh giá có thể sẽ không gây ảnh hưởng quá lớn đối với kết quả. Ta tạm thời bỏ qua khoảng cách từ nhân vật tới thùng mà hãy tập trung vào vấn đề lớn nhất của bài toán là đường đi từ thùng tới điểm đích.

Để tối ưu giá trị hàm đánh giá, ta cần chọn đường đi ngắn nhất từ thùng tới một điểm đích bằng cách lấy MIN của các công thức dưới đây với các $point_{ij}$ được sắp xếp thứ tự các điểm đích khác nhau:

$$f = \min \left(\sum_{i=1}^n \sum_{j=1}^2 |box_{ij} - point_{ij}| \right)_k \Bigg|_{k=1}^{n!}$$

Hàm đánh giá trên có vẻ đã tối ưu về mặt tính toán. Tuy nhiên hàm đánh giá này đang gặp phải hai vấn đề:

- Để triển khai hàm, ta cần xét tới $n!$ trường hợp của $point_{ij}$. Đây là một con số không cố định và biên độ thay đổi quá lớn đối với mỗi giá trị n khác nhau. Vậy nên hàm sẽ rất khó để triển khai.

- Hàm tìm ra tổng đường đi ngắn nhất từ từng thùng tới từng điểm đích. Do đó, giá trị MIN luôn là trường hợp mà các thùng được đẩy tới vị trí gần nhất còn trống. Nhưng trong một số bài toán cụ thể nhất định, để giải được, ta cần đẩy thùng tới vị trí rất xa so với nó. Điều này khiến hàm sẽ có định hướng giải sai và chỉ có thể đổi hướng giải khi gặp phải trạng thái thất bại.

Từ các lập luận trên, ta cần đưa ra một hàm đánh giá tổng quát cho mọi trường hợp và lấy đường đi từ thùng đến điểm đích làm cốt lõi:

$$f = \left| \sum_{i=1}^n \sum_{j=1}^2 (box_{ij} - point_{ij}) \right|$$

2.4.2 Code

Từ công thức trên của hàm đánh giá, ta có thể mô hình thành đoạn code sau:

```
1 self.heuristic = abs(sum(list_boxes[i][0] + list_boxes[i][1]
    - self.check_points[i][0] - self.check_points[i][1] for i
    in range(len(list_boxes))))
```

Trong đó:

- self.heuristic \iff f.
- list_boxes[i] \iff box_i.
- self.checkpoints[i] \iff point_i.
- len(list_boxes) \iff n.

Đây cũng chính là mô hình hàm đánh giá cho Best First Search bởi thuật toán này xét tới chi phí dự kiến để đi từ trạng thái hiện tại đến trạng thái đích.

Đối với A*, thuật toán này còn xét tới chi phí để đi từ trạng thái bắt đầu đến trạng thái hiện tại. Do đó ta cần cộng thêm một thành phần nữa là số bước đi đã thực hiện trong trường hợp cụ thể \iff n = len(self.getline()):

```
1 self.heuristic = len(self.get_line()) + abs(sum(list_boxes[i]
    [0] + list_boxes[i][1] - self.check_points[i][0] - self.
    check_points[i][1] for i in range(len(list_boxes))))
```

Hàm đánh giá này là hàm compute_heuristic của file support_function.py

Mở rộng:

Ta có thể cộng thêm một yếu tố nữa vào hàm đánh giá, đó là số lượng thùng chưa được đẩy tới đích. Trên lý thuyết, hàm mở rộng sẽ tốt hơn, tuy nhiên nhóm đã triển khai chạy hàm này với 20 mẫu thử và kết quả không khác so với hàm gốc. Do đó, để tối ưu dung lượng bộ nhớ và số bước tính toán, nhóm quyết định chọn hàm gốc làm hàm đánh giá cho bài toán.

2.5 Mẫu thử

Mỗi mẫu thử bao gồm hai file thành phần:

- Bản đồ: File txt chứa ma trận biểu thị bản đồ; nằm trong thư mục Testcases.
- Điểm đích: File txt chứa ma trận tọa độ các điểm đích; nằm trong thư mục Checkpoints.

Các file này được đặt tên theo thứ tự từ 1 đến 20 trong hệ nhị phân để khi duyệt thư mục python sẽ sắp xếp thứ tự level theo đúng thứ tự bản đồ trong thư mục gốc.

Bản đồ được mô hình theo từng ký tự. Mỗi ký tự ứng với một vị trí, ngăn cách nhau bởi dấu ','. Các vật thể được mô hình theo quy tắc dưới đây:

STT	Vật thể	Ký tự đầu vào	Ký tự biên dịch
1	Tường	'1'	'#'
2	Thùng	'b'	'\$'
3	Điểm đích	'c'	'%'
4	Nhân vật	'p'	'@'
5	Đường đi	' '	' '

Bảng 2.2: Định dạng ký tự đại diện vật thể

Chương trình không biên dịch trực tiếp từ ký tự đầu vào mà cần phải có thêm bước chuyển thành các ký tự đặc biệt trong cột Ký tự biên dịch là do các ký tự này là quy chuẩn cho bài toán Sokoban. Do đó, khi dùng bộ mẫu theo chuẩn hay bộ mẫu mà nhóm tìm được tương ứng với cột Ký tự đầu vào thì chương trình cũng đều có thể nhận biết thành phần của bài toán.

2.6 Giao diện

Giao diện của chương trình được cấu thành từ file .png gắn vào từng đối tượng.

Toàn bộ thành phần của giao diện được lưu trữ trong thư mục Assets.

Công cụ xây dựng giao diện: Thư viện Pygame.

Ý tưởng:

Thay vì làm một trò chơi Sokoban thông thường với các đối tượng là người, thùng, tường và checkpoint, nhóm đã biến tấu giao diện với cảm hứng từ vị tướng Teemo trong tựa game League of Legends (LoL).

Vị tướng này có kỹ năng R là ném bẫy nấm độc. Do đó nhóm đã chuyển đổi bối cảnh cho trò chơi thành Teemo đẩy Nấm độc vào Bụi cỏ (Vị trí mà kẻ địch không thể nhìn thấy). Bên cạnh đó, Hình ảnh Tường cũng được thay thế bằng Turret (Tháp canh của đối thủ mà khi đi vào nhân vật sẽ mất máu).

3 Thống kê

Toàn bộ thống kê dưới đây đều được thu thập qua quá trình chạy chương trình trong cùng một điều kiện về phần cứng là cùng chạy trên laptop LENOVO:

- Hệ điều hành: Windows 11 Home Single Language 64-bit.
- CPU: Intel Core i5-1135G7 2.4 GHz (8 cores).
- RAM: 8192 MB \approx 8GB.
- SSD: 256 GB
- GPU: Intel Iris Xe Graphics.

Giới hạn thời gian cho mỗi lần chạy chương trình là 30 phút (1800 giây). Nếu quá khoảng thời gian này mà chương trình vẫn chưa tìm được lời giải thì coi như chương trình không thể giải.

Lưu ý: Những trường hợp timeout sẽ được tô đỏ trong bảng số liệu và biểu thị bằng số âm trong biểu đồ.

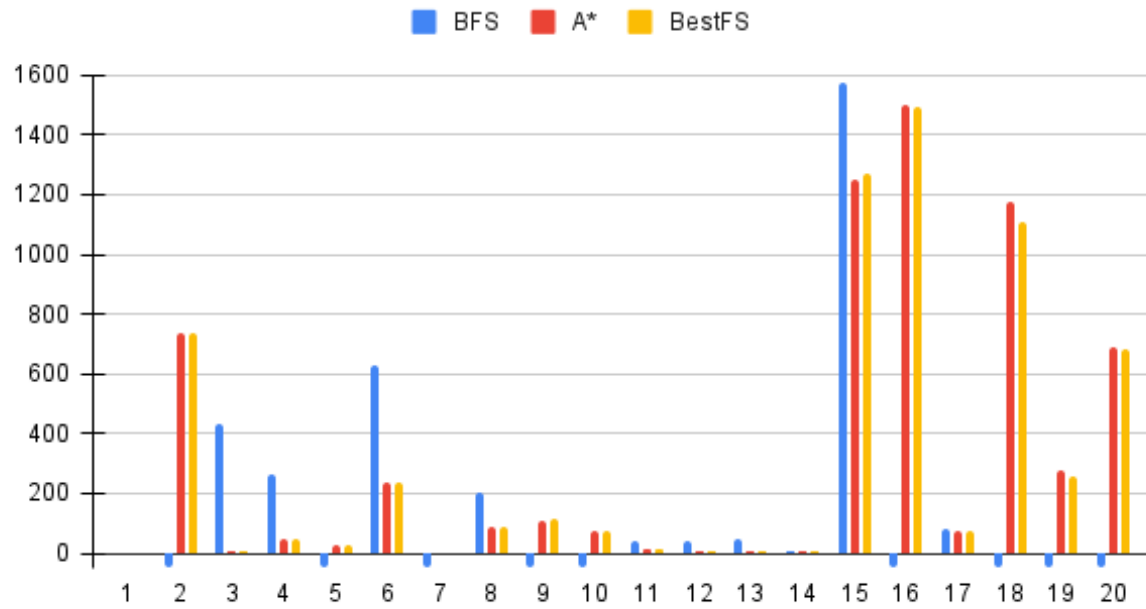
3.1 Thời gian

Level	BFS	A*	BestFS
1	2,736	0,654	0,641
2	-50	735	738,366
3	430	5	5
4	261	45,342	46,37
5	-50	26,652	27,015
6	631,687	235,781	237,892
7	-50	1,5	1,5
8	205,94	90,578	90,332
9	-50	111	112
10	-50	76,34	77
11	42,559	11,456	11,29
12	40,043	5,9	5,9
13	45,62	8	8,102
14	9,047	6,731	6,776
15	1578	1248	1273
16	-50	1500	1492
17	81	75	77
18	-50	1173	1110
19	-50	276,686	257
20	-50	692,92	681,782

Bảng 3.1: Thời gian chạy thuật toán

Đơn vị: Giây

Thời gian chạy thuật toán (Giây)



Hình 3.1: Biểu đồ Thời gian chạy thuật toán

3.2 Bộ nhớ

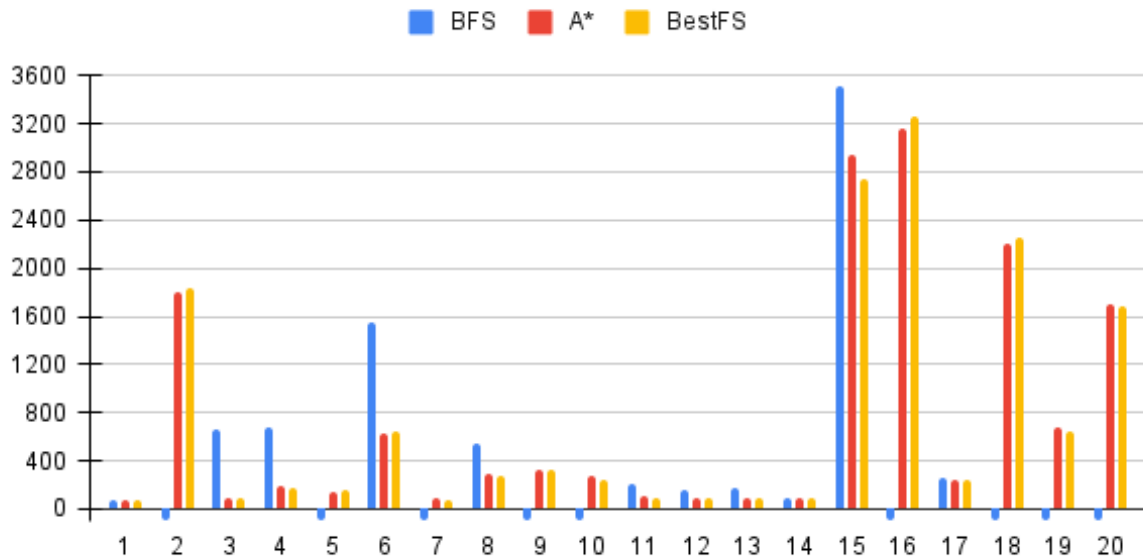
Level	BFS	A*	BestFS
1	81	79	78,4
2	-100	1809	1832
3	655	87	88
4	681	187	180
5	-100	147,3	153,6
6	1555,4	630,3	641,4
7	-100	83,7	82,3
8	541	290	282,8
9	-100	325	321
10	-100	274	249,2
11	204	102	99,5
12	167	95,5	96,5
13	170	95	93,3
14	93	92	95,5
15	3513	2936	2740
16	-100	3157,5	3256
17	264	241	245
18	-100	2213	2260,7
19	-100	684,6	651
20	-100	1703	1688

Bảng 3.2: Dung lượng bộ nhớ tiêu tốn

Đơn vị: MiB (Mebibyte)

Dung lượng tiêu tốn (Mebibyte)

1 MiB = 1048576 B = 1.048576 MB



Hình 3.2: Biểu đồ dung lượng tiêu tốn

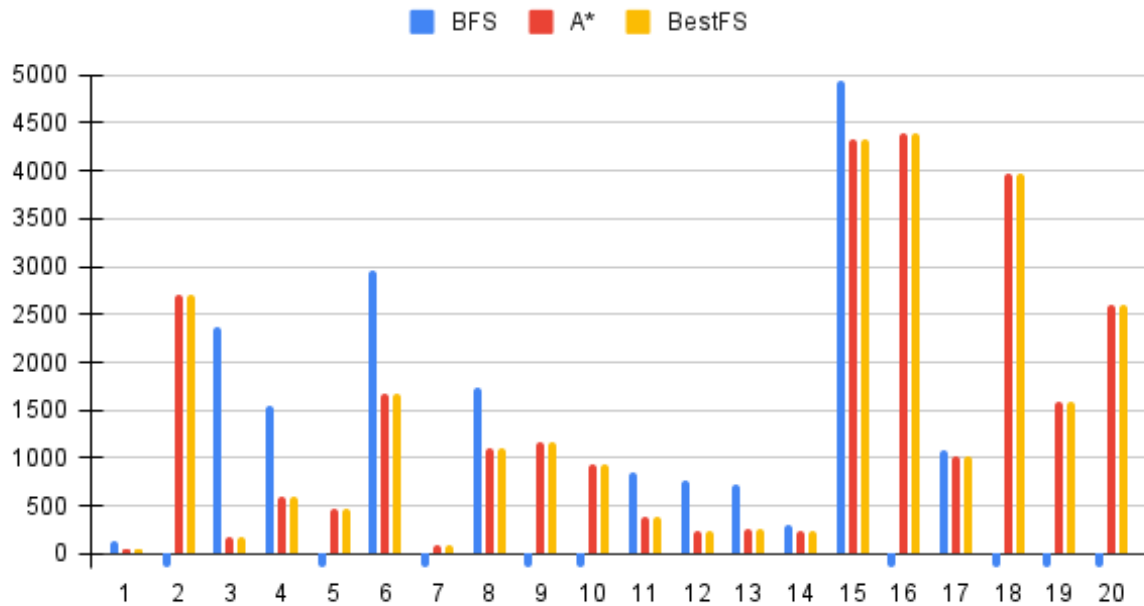
3.3 Trạng thái

3.3.1 Trạng thái cần duyệt

Level	BFS	A*	BestFS
1	131	47	47
2	-150	2716	2716
3	2379	183	183
4	1550	595	595
5	-150	479	479
6	2970	1674	1678
7	-150	81	81
8	1739	1095	1095
9	-150	1170	1170
10	-150	926	923
11	857	378	378
12	763	233	233
13	724	260	260
14	293	234	234
15	4937	4321	4321
16	-150	4400	4400
17	1090	1013	1013
18	-150	3971	3971
19	-150	1587	1587
20	-150	2603	2603

Bảng 3.3: Số trạng thái cần duyệt

Số trạng thái cần duyệt



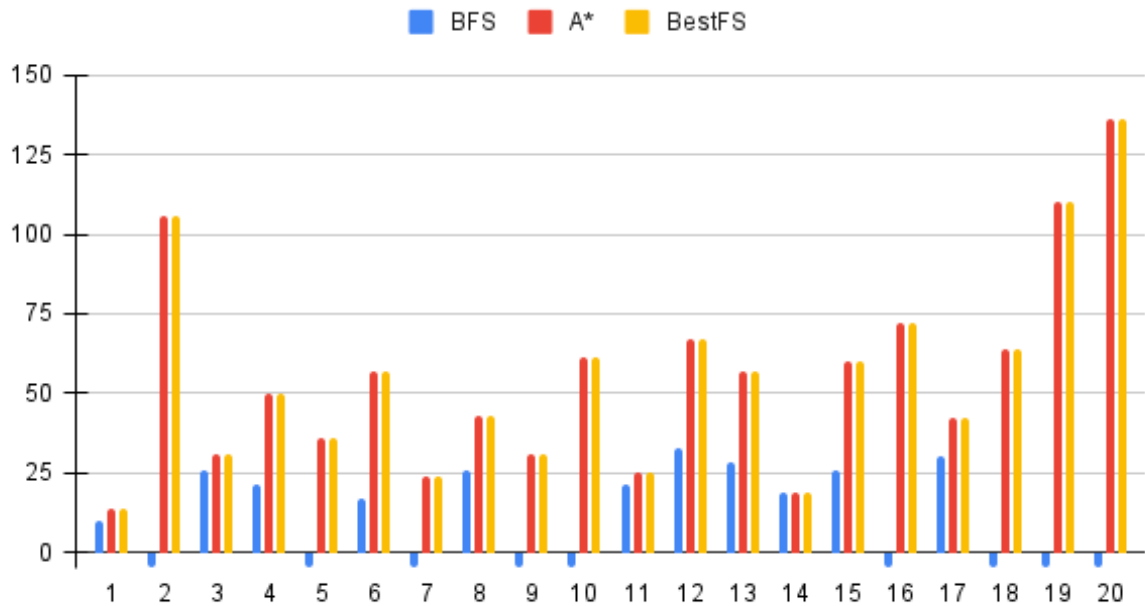
Hình 3.3: Biểu đồ Số trạng thái cần duyệt

3.3.2 Trạng thái sử dụng - Số bước di chuyển

Level	BFS	A*	BestFS
1	10	10	14
2	-5	106	106
3	26	31	31
4	21	50	50
5	-5	36	36
6	17	57	57
7	-5	24	24
8	26	43	43
9	-5	31	31
10	-5	61	61
11	21	25	25
12	33	67	67
13	28	57	57
14	19	19	19
15	26	60	60
16	-5	72	72
17	30	42	42
18	-5	64	64
19	-5	110	110
20	-5	136	136

Bảng 3.4: Số bước nhân vật di chuyển

Số bước đi được chọn



Hình 3.4: Biểu đồ Số bước nhân vật di chuyển

Tài liệu

- [1] Petr Jarusek và Radek Pelánek. Difficulty rating of sokoban puzzle. In *STAIRS 2010 - Proceedings of the Fifth Starting AI Researchers*. IOS Press, 2010.
- [2] Dorit Dor và Uri Zwick. Sokoban and other motion planning problems. Technical report, Phòng Khoa học máy tính, Trường Toán học và Raymond and Beverly Sackler Khoa Khoa học định lượng, Đại học Tel Aviv, November 1998.
- [3] Tim Allan Wheeler. Basic search algorithms on sokoban. January 2022.