

DẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH  
TRƯỜNG ĐẠI HỌC BÁCH KHOA  
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH



## NHẬP MÔN TRÍ TUỆ NHÂN TẠO (CO3061)

Bài tập lớn 1 (học kì: 251)

# "BÀI TOÁN SOKOBAN"

Instructor(s): GV. Vương Bá Thịnh

**Student(s):** Trần Thị Ngọc Huyền 2211311 (*Lớp L02 - Nhóm x*)  
Trịnh Thị Anh Thư 2213412 (*Lớp L02 - Nhóm x*)  
Nguyễn Mạnh Dũng 2210583 (*Lớp L01 - Nhóm x*)  
Phạm Thị Tố Như 2212478 (*Lớp L01 - Nhóm x*)



## Contents

<b>1 Thành viên và phân chia công việc</b>	<b>3</b>
<b>2 Giới thiệu đề tài - Bài toán SOKOBAN</b>	<b>4</b>
2.1 Giới thiệu bài toán . . . . .	4
2.2 Cơ chế chơi . . . . .	4
<b>3 Cơ sở lý thuyết</b>	<b>5</b>
3.1 Thuật toán Blind Search - BFS . . . . .	5
3.1.1 Giới thiệu . . . . .	5
3.1.2 Ý tưởng áp dụng BFS để giải bài toán Sokoban . . . . .	5
3.1.3 Ưu điểm của BFS . . . . .	6
3.1.4 Nhược điểm của BFS . . . . .	6
3.2 Thuật toán Heuristic . . . . .	6
3.2.1 Giới thiệu . . . . .	6
3.2.2 Ý tưởng áp dụng A* search để giải bài toán Sokoban . . . . .	6
3.2.3 So sánh độ hiệu quả với BFS . . . . .	7
<b>4 Dữ liệu sử dụng</b>	<b>8</b>
4.1 Giới thiệu nguồn dữ liệu . . . . .	8
4.2 Định dạng dữ liệu . . . . .	8
4.3 Giải thích các ký hiệu . . . . .	9
4.4 Cấu trúc logic của bản đồ . . . . .	9
4.5 Mục đích sử dụng . . . . .	9
<b>5 Hiện thực</b>	<b>10</b>
5.1 Mã nguồn . . . . .	10
5.1.1 Giải thuật Blind Search . . . . .	10
5.1.2 Giải thuật Heuristic . . . . .	15
5.2 Kiểm thử: . . . . .	16
<b>6 Giao diện kiểm thử</b>	<b>22</b>
6.1 Mã nguồn . . . . .	22
6.2 Các màn hình . . . . .	22
<b>7 Tham khảo và phụ lục</b>	<b>24</b>



## 1 Thành viên và phân chia công việc

STT	Họ và tên	MSSV	Công việc	Đóng góp
1	Nguyễn Mạnh Dũng	2210583	3.1 Cơ sở lý thuyết - Blind Search và 5.1.1 Hiện thực - Blind Search	100%
2	Phạm Thị Tố Như	2212478	3.2 Cơ sở lý thuyết - Heuristic và 5.1.1 Hiện thực - Heuristic	100%
3	Trịnh Thị Anh Thư	2213412	4. Dữ liệu sử dụng	100%
4	Trần Thị Ngọc Huyền	2211311	6. Giao diện kiểm thử	100%

Table 1: Phân chia công việc

Thời gian	Nhiệm vụ
GD 1: 07/10 - 13/10	1. Tạo map từ Sokoban Mini cosmos 2. Tìm hiểu giải thuật và setup dự án
GD 2: 14/10 - 22/10	1. Hiện thực giải thuật 2. Xây dựng giao diện ban đầu
GD 3: 23/10 - 27/10	1. Hoàn thiện giao diện 2. Đo bộ nhớ và thời gian
GD 4: 28/10 - 01/11	1. Quay video 2. Viết báo cáo

Table 2: Công việc nhóm thực hiện trong các giai đoạn

## 2 Giới thiệu đề tài - Bài toán SOKOBAN

### 2.1 Giới thiệu bài toán

Sokoban là trò chơi dạng câu đố trong đó người chơi phải đẩy một số khối vuông vượt qua chướng ngại vật để đến đích. Trò chơi đã được thiết kế vào năm 1981 bởi Hiroyuki Imabayashi và được ra mắt lần đầu vào tháng 12 năm 1982 bởi Thinking Rabbit, một công ty phần mềm có trụ sở tại Takarazuka, Nhật Bản.

Mặc dù cơ chế chơi rất đơn giản, Sokoban đã trở thành một biểu tượng của thể loại puzzle hay logic và được phát hành trên nhiều nền tảng. Ngoài ra, trong khoa học máy tính và trí tuệ nhân tạo, Sokoban được sử dụng làm bài toán mô hình cho việc lập kế hoạch, tìm kiếm và đánh giá heuristics vì mang đặc trưng “vừa đơn giản về luật, nhưng khó về giải thuật”.

### 2.2 Cơ chế chơi

- **Mục tiêu:** của trò chơi là đẩy tất cả các khối hộp vào đúng những vị trí được đánh dấu đích (thường là ô tròn hoặc màu khác).

- **Thao tác:**

- Người chơi điều khiển một nhân vật thủ kho trong mê cung.
- Nhân vật có thể di chuyển 4 hướng: Lên, Xuống, Trái và Phải.
- Nhân vật chỉ có thể đẩy hộp (không thể kéo) và mỗi lần chỉ đẩy được một hộp.
- Hộp được đẩy theo hướng Nhân vật di chuyển trong điều kiện không vướng vật cản, VD: Khi đẩy hộp về phía bên trái, nhân vật phải nằm ở cạnh bên phải hộp và thực hiện thao tác di chuyển về bên trái.

- **Trò chơi kết thúc:** Khi tất cả các hộp đều nằm đúng trên các vị trí đích.

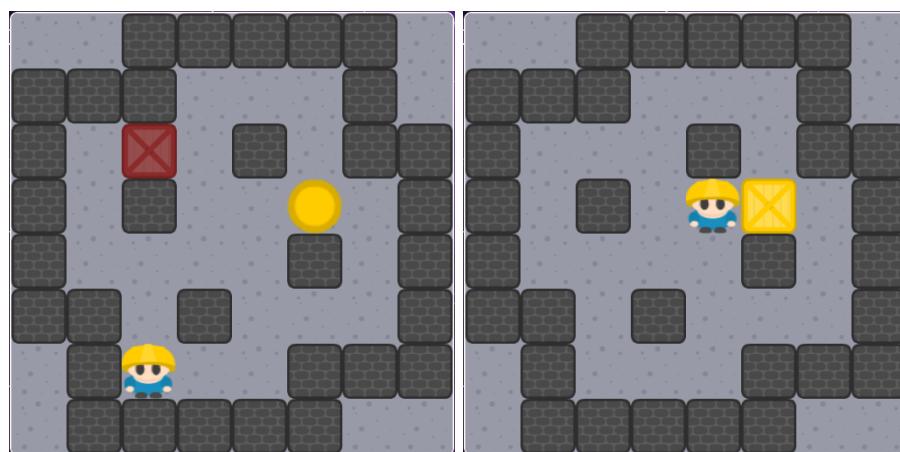


Figure 1: Trạng thái khởi đầu và kết thúc của một màn Sokoban



### 3 Cơ sở lý thuyết

#### 3.1 Thuật toán Blind Search - BFS

##### 3.1.1 Giới thiệu

Thuật toán BFS (Breadth-First Search) được sử dụng để giải bài toán Sokoban dựa trên việc mô hình hoá trò chơi thành một không gian trạng thái. Mỗi trạng thái được xác định bởi vị trí của người chơi, vị trí của các hộp và vị trí của ô đích trên bản đồ. Mục tiêu cuối cùng là đưa tất cả hộp vào các ô đích được quy định trước. Tại mỗi bước, người chơi có thể di chuyển lên, xuống, trái hoặc phải nếu ô phía trước không phải tường hay hộp chắn; trong trường hợp ô phía trước là một hộp và ô tiếp theo sau hộp trống, người chơi có thể đẩy hộp theo hướng di chuyển. Bằng cách áp dụng BFS, thuật toán sẽ mở rộng tất cả các trạng thái có thể đạt được trong một số bước nhất định trước khi tiếp tục đến những trạng thái xa hơn.

##### 3.1.2 Ý tưởng áp dụng BFS để giải bài toán Sokoban

Để áp dụng BFS vào Sokoban, ta tiến hành mô hình hoá trò chơi như một bài toán tìm kiếm trong không gian trạng thái như sau:

**Trạng thái (state):** Một trạng thái được biểu diễn bởi cặp (vị trí người chơi, tập vị trí các hộp). Hai trạng thái được xem là giống nhau nếu người chơi và tất cả các hộp cùng chiếm các ô tương ứng.

**Trạng thái đầu (start):** Trạng thái xuất phát được lấy từ cấu hình ban đầu của bản đồ Sokoban, nơi người chơi và các hộp được đặt theo đề bài.

**Trạng thái đích (goal test):** Một trạng thái được xem là mục tiêu khi *tất cả* các hộp đều được đặt trên các ô đích quy định.

**Hàm sinh láng giềng (neighbors):** Từ một trạng thái, ta có thể sinh ra các trạng thái kề bắc các hành động hợp lệ:

- Người chơi di chuyển sang một ô trống kề bên.
- Nếu người chơi di chuyển vào một ô có hộp và ô phía sau hộp trống thì người chơi có thể đẩy hộp sang ô đó.

##### Cấu trúc dữ liệu:

- **Hàng đợi (queue):** dùng để duyệt các trạng thái theo lớp (mức độ sâu tăng dần).
- **Tập đã thăm (visited):** lưu lại các trạng thái đã xuất hiện nhằm tránh lặp vô hạn và giảm không gian tìm kiếm.

**Khởi tạo:** Ta đưa trạng thái ban đầu vào hàng đợi, đánh dấu nó đã được thăm trong visited. Đồng thời, ta lưu lại thông tin cha của trạng thái này để phục vụ việc truy vết lời giải về sau.

**Vòng lặp duyệt theo lớp:** Trong khi hàng đợi chưa rỗng, ta lấy ra trạng thái đầu hàng để xét. Nếu trạng thái này thoả mãn điều kiện đích thì thuật toán dừng lại và dựng lại đường đi từ thông tin cha. Nếu không, ta sinh tất cả các trạng thái kề hợp lệ và với mỗi trạng thái mới chưa từng được thăm, ta đánh dấu, lưu cha và đưa vào hàng đợi.

**Kết thúc:** Nếu hàng đợi rỗng mà vẫn chưa tìm thấy trạng thái đích, kết luận bài toán không có lời giải. Nếu tìm thấy trạng thái đích, lời giải mà BFS trả về chính là lời giải ngắn nhất tính theo số bước di chuyển, do BFS duyệt theo lớp và mọi bước đều có chi phí bằng nhau.



### 3.1.3 Ưu điểm của BFS

BFS được xếp vào nhóm thuật toán blind search vì nó không sử dụng bất kỳ thông tin ước lượng hay heuristic nào để đánh giá xem trạng thái nào có tiềm năng tốt hơn. Nó duyệt theo chiều rộng toàn bộ không gian trạng thái theo từng lớp, từ gần đến xa. Trong quá trình tìm kiếm, thuật toán phải lưu lại những trạng thái đã từng ghé qua để tránh lặp vô hạn và giảm kích thước không gian tìm kiếm. Nhờ tính chất duyệt theo mức độ, BFS rất hữu ích cho Sokoban khi yêu cầu tối ưu về số bước di chuyển.

BFS sẽ lần lượt mở rộng toàn bộ không gian trạng thái có thể đạt được từ trạng thái ban đầu. Nếu sau khi duyệt hết tất cả các trạng thái hợp lệ mà vẫn không gặp trạng thái mục tiêu (goal state), hàng đợi (queue) sẽ rỗng. Khi đó, BFS kết luận không có lời giải. Trong Sokoban, mỗi bước di đều có chi phí như nhau. BFS mở rộng trạng thái theo mức độ, nên lời giải đầu tiên mà BFS tìm thấy luôn là lời giải tối ưu nhất (số bước đi ít nhất).

### 3.1.4 Nhược điểm của BFS

Không gian trạng thái của Sokoban rất lớn do sự kết hợp giữa các vị trí hộp và đường di chuyển của người chơi. Điều này khiến thuật toán dễ gặp tình trạng bùng nổ trạng thái (state explosion) và tiêu tốn nhiều bộ nhớ.

## 3.2 Thuật toán Heuristic

### 3.2.1 Giới thiệu

A\* search (Astar search) thuộc nhóm các thuật toán tìm kiếm tốt nhất trước (Best-first Search)<sup>1</sup>. Thuật toán này có gắng mở rộng nút (node) mà nó dự đoán là gần mục tiêu nhất dựa trên hàm đánh giá  $f(n) = g(n) + h(n)$ . Trong đó  $g(n)$  là chi phí di từ nút đầu đến nút  $n$ ,  $h(n)$  là chi phí ước lượng của đường ngắn nhất từ nút  $n$  đến mục tiêu.

Tìm kiếm Tốt nhất Trước (Best-First Search) kết hợp các ưu điểm của biến thể tìm kiếm theo chiều sâu (depth-first) và theo chiều rộng (breadth-first) bằng cách cho phép một trong hai kiểu mẫu mở rộng tùy thuộc vào thông tin heuristic. Theo đó, tìm kiếm tốt nhất trước có thể tạm thời thăm dò sâu vào một nhánh trước khi xác định rằng nhánh khám phá hiện tại kém tiềm năng hơn một nút chưa được mở rộng có độ ưu tiên cao hơn nằm ở cấp độ nông hơn trong cây. Thuật toán có thể quay lại nhánh đã tạm dừng sau đó, nếu các trạng thái khác được đánh giá là không còn hứa hẹn.

### 3.2.2 Ý tưởng áp dụng A\* search để giải bài toán Sokoban

Ở bước mô hình hóa bài toán, bài toán này được mô hình hóa tương tự như quá trình mô hình hóa để giải quyết bài toán Sokoban bằng thuật toán BrFS được trình bày ở trên.

#### Cấu trúc dữ liệu:

- Hàng đợi ưu tiên (priority queue): dùng để duyệt các trạng thái theo thứ tự tăng dần của giá trị hàm đánh giá của mỗi trạng thái
- Dictionary: lưu trữ chi phí thực tế thấp nhất của các trạng thái đã xuất hiện

**Khởi tạo:** Thuật toán bắt đầu bằng việc đọc dữ liệu đầu vào, xác định vị trí ban đầu của người chơi, hộp và các ô mục tiêu. Trạng thái ban đầu được đưa vào hàng đợi ưu tiên (priority

<sup>1</sup>Russell, S. and Norvig, P. (2010) Artificial Intelligence: A Modern Approach. 3rd Edition, Prentice-Hall, Upper Saddle River.



queue), trong đó giá trị  $f(n)=g(n)+h(n)$  được tính dựa trên chi phí đường đi thực tế  $g(n)$  và giá trị heuristic  $h(n)$  ước lượng khoảng cách còn lại đến mục tiêu. Đồng thời, ta lưu lại thông tin cha của mỗi trạng thái để phục vụ việc truy vết lời giải.

**Vòng lặp tìm kiếm:** Trong khi hàng đợi chưa rỗng, thuật toán lấy ra trạng thái có giá trị  $f(n)$  nhỏ nhất để xét. Nếu trạng thái này thoả mãn điều kiện đích, thuật toán dừng lại và dựng lại đường đi lời giải thông qua thông tin cha. Nếu chưa đạt đích, thuật toán sinh tất cả các trạng thái kề hợp lệ, tính toán lại các giá trị  $f$  tương ứng. Với mỗi trạng thái mới tốt hơn (có  $g$  nhỏ hơn), ta cập nhật thông tin và đưa vào hàng đợi ưu tiên.

**Kết thúc:** Thuật toán kết thúc khi tìm được trạng thái đích (trả về chuỗi các bước di chuyển tương ứng) hoặc khi hàng đợi rỗng mà chưa đạt đích (kết luận bài toán không có lời giải).

### 3.2.3 So sánh độ hiệu quả với BFS

Khác với BFS duyệt toàn bộ các trạng thái theo lớp mà không ưu tiên hướng đi, A\* sử dụng hàm đánh giá  $f(n)=g(n)+h(n)$  để tập trung mở rộng các trạng thái tiềm năng nhất. Nhờ đó, A\* thường tìm được lời giải nhanh hơn và tiết kiệm bộ nhớ hơn so với BFS, đặc biệt khi hàm heuristic được thiết kế tốt và gần với chi phí thực tế.

## 4 Dữ liệu sử dụng

### 4.1 Giới thiệu nguồn dữ liệu

Bộ dữ liệu được sử dụng trong đề tài được lấy từ **Sokoban — Mini Cosmos**, một tập hợp các màn chơi (*maps*) Sokoban trực tuyến tại địa chỉ:

<https://ksokoban.online/Mini%20Cosmos>

Bộ dữ liệu bao gồm **20 bản đồ đầu tiên**, mỗi bản đồ đại diện cho một **testcase đầu vào** mô tả trạng thái ban đầu của trò chơi Sokoban: vị trí người chơi, hộp, tường và các ô đích.

### 4.2 Định dạng dữ liệu

Mỗi bản đồ trong bộ dữ liệu được thể hiện trực quan dưới dạng lưới ô vuông bao gồm các tường, hộp, vị trí người chơi và các ô đích. Hình 2 minh họa ví dụ **Level 1 – Mini Cosmos** trong trò chơi Sokoban.



Figure 2: Minh họa bản đồ Level 1 trong bộ dữ liệu Sokoban – Mini Cosmos

Từ bản đồ trực quan như trên, dữ liệu được trích xuất và lưu trữ dưới dạng **ma trận ký tự hai chiều (2D array)**, trong đó mỗi ký hiệu đại diện cho một loại ô trong lưới. Ví dụ, bản đồ **Mini Cosmos – Map 1** được mã hóa thành:

```
[  
  [" ", "#", "#", "#", "#", "#", "#", " "],  
  ["#", "#", "#", " ", " ", "#", " "],  
  ["#", " ", "$", " ", "#", " ", "#", "#"],  
  ["#", " ", "#", " ", " .", " ", "#"],  
  ["#", " ", " ", " ", "#", " ", "#"],  
  ["#", "#", " ", "#", " ", " ", "#"],  
  [" ", "#", "@", " ", " ", "#", "#"],
```



[ " ", "#", "#", "#", "#", "#", " ", " " ]

### 4.3 Giải thích các ký hiệu

Ký hiệu	Tên chuẩn (English)	Vai trò / Mô tả
#	Wall	Tường – ô cố định, không thể di chuyển hoặc đi qua
.	Goal / Storage	Ô đích – nơi cần đẩy hộp vào
\$	Box / Crate	Hộp – vật thể có thể được người chơi đẩy
*	Box on Goal	Hộp đang nằm đúng vị trí đích
@	Player	Người chơi (điều khiển chính)
+	Player on Goal	Người chơi đang đứng trên ô đích
(space)	Floor / Empty	Ô trống – có thể di chuyển qua

Table 3: Bảng mô tả ý nghĩa các ký hiệu trong bản đồ Sokoban

### 4.4 Cấu trúc logic của bản đồ

Một bản đồ Sokoban thường gồm ba thành phần chính:

- **Tường (Wall):** bao quanh hoặc chia khu vực, xác định không gian di chuyển hợp lệ.
- **Đối tượng di chuyển (Box, Player):** nằm trong các ô trống, được phép đẩy nhưng không thể kéo.
- **Mục tiêu (Goal):** điều kiện thắng khi tất cả hộp được đặt lên các ô đích.

### 4.5 Mục đích sử dụng

Các bản đồ trong bộ dữ liệu được sử dụng làm testcases để kiểm thử và đánh giá hiệu quả của hai giải thuật được triển khai trong đề tài. Cụ thể, việc sử dụng bộ dữ liệu nhằm:

- **Kiểm thử độ chính xác:** xác định liệu giải thuật có thể tìm được lời giải hợp lệ cho từng bản đồ Sokoban hay không.
- **Đánh giá hiệu suất:** đo lường thời gian thực thi, số bước di chuyển và lượng bộ nhớ sử dụng của từng giải thuật.
- **So sánh kết quả:** đối chiếu hiệu quả hoạt động giữa hai giải thuật **Blind Search** và **Heuristic Search** để phân tích ưu, nhược điểm của từng hướng tiếp cận.



## 5 Hiện thực

### 5.1 Mã nguồn

#### 5.1.1 Giải thuật Blind Search

```
1 # 4-neighborhood movement: Up, Down, Left, Right
2 DIRS = {
3     'U': (-1, 0), # move up: row decreases, column unchanged
4     'D': ( 1, 0), # move down
5     'L': ( 0,-1), # move left
6     'R': ( 0, 1), # move right
7 }
```

**Listing 1:** Imports and direction vectors

Khởi tạo các import tiêu chuẩn và từ điển DIRS ánh xạ nhãn hướng  $\{U, D, L, R\}$  sang véc-tơ dịch chuyển  $(\Delta r, \Delta c)$  trên lưới. Cấu trúc này được tái sử dụng để sinh láng giềng, kiểm tra tiếp cận, v.v

```
1 def parse_board(testcase):
2     """
3         Read a Sokoban board from a test file and return its components.
4
5     Args:
6         testcase (str): filename of the test case (e.g., "mini_cosmos_1.txt")
7
8     Returns:
9         H (int): number of rows
10        W (int): number of columns
11        walls (set[tuple[int,int]]): set of wall coordinates
12        goals (set[tuple[int,int]]): set of target coordinates
13        boxes (frozenset[tuple[int,int]]): set of box coordinates (hashable)
14        player (tuple[int,int]): player coordinate (row, col)
15
16    Symbols in the board:
17        '#': wall
18        '.': target
19        '$': box
20        '@': player
21        '*': box on target (both a goal and a box)
22        '+': player on target (both a goal and the player)
23        ' ': empty floor
24
25    os.chdir(os.path.abspath(os.path.join(os.path.dirname(__file__), '..', '..')))
26    file_path = os.path.join('SokobanMap', testcase)
27
28    with open(file_path, 'r') as f:
29        board = ast.literal_eval(f.read()) # expects a Python literal (e.g., list of
30                                         strings)
31
32    H = len(board)
```



```
32     W = len(board[0])
33
34     walls, goals, boxes = set(), set(), set()
35     player = None
36
37     for r in range(H):
38         for c in range(W):
39             ch = board[r][c]
40             if ch == '#':
41                 walls.add((r, c))
42             elif ch == '.':
43                 goals.add((r, c))
44             elif ch == '$':
45                 boxes.add((r, c))
46             elif ch == '@':
47                 player = (r, c)
48             elif ch == '*': # box on a goal
49                 goals.add((r, c))
50                 boxes.add((r, c))
51             elif ch == '+': # player on a goal
52                 goals.add((r, c))
53                 player = (r, c)
54
55     return H, W, walls, goals, frozenset(boxes), player
```

**Listing 2:** parse board: read and parse a Sokoban board

Hàm đọc bản đồ từ tệp test (định dạng literal Python) và trích xuất các tập: `walls` (tường), `goals` (dích), `boxes` (hộp), cùng vị trí `player`. Việc trả `boxes` dưới dạng `frozenset` giúp trạng thái (`player, boxes`) trở nên *hashable* để dùng trong `visited`.

```
1 def is_inside(r, c, H, W):
2     """
3     Return True if (r, c) is within the board of size H x W.
4     """
5     return 0 <= r < H and 0 <= c < W
```

**Listing 3:** is inside: boundary check

Kiểm tra biên: một tọa độ  $(r, c)$  có nằm trong lưới kích thước  $H \times W$  hay không. Đây là điều kiện nền để tránh truy cập ngoài biên.

```
1 def is_free(cell, walls, boxes, H, W):
2     """
3     Return True if 'cell' is in-bounds and not occupied by a wall or a box.
4
5     Args:
6         cell (tuple[int,int]): (row, col)
7     """
8     return is_inside(*cell, H, W) and (cell not in walls) and (cell not in boxes)
```

**Listing 4:** is free: emptiness check



Xác định một ô có “trống” để người có thể đi vào hoặc hộp có thể được đẩy đến: phải trong biên, không là tường, không bị hộp chiếm.

```
1 def goal_test(boxes, goals):
2     """
3         Return True if every box is on a goal position.
4     """
5     for box in boxes:
6         if box not in goals:
7             return False
8     return True
```

**Listing 5:** goal test: all boxes on targets

Điều kiện đích của Sokoban: mọi hộp đều nằm trên các ô đích. Hàm này trả về True nếu  $\text{boxes} \subseteq \text{goals}$ .

```
1 def neighbors(state, walls, goals, H, W):
2     """
3         Generate adjacent states from 'state' along with the move character ('U','D','L','R').
4
5     Rules:
6         - If the front cell is free => player makes a normal step.
7         - If the front cell has a box AND the cell beyond is free => push the box.
8
9     Returns:
10        list[ ((player, boxes'), move_char) ]
11    """
12    player, boxes = state
13    res = []
14    for m, (dr, dc) in DIRS.items():
15        nr, nc = player[0] + dr, player[1] + dc
16        nxt = (nr, nc)
17
18        # Blocked by walls or outside the board
19        if not is_inside(nr, nc, H, W) or nxt in walls:
20            continue
21
22        if nxt in boxes:
23            # Attempt to push the box: the beyond cell must be free
24            br, bc = nr + dr, nc + dc
25            beyond = (br, bc)
26            if is_free(beyond, walls, boxes, H, W):
27                new_boxes = set(boxes)
28                new_boxes.remove(nxt)
29                new_boxes.add(beyond)
30                res.append(((nxt, frozenset(new_boxes)), m))
31            else:
32                # Normal step (no box in front)
33                res.append(((nxt, boxes), m))
34
35    return res
```



**Listing 6:** neighbors: generate successor states

Sinh các trạng thái kế từ trạng thái hiện tại: xét bốn hướng; nếu trước mặt là ô trống thì người bước thường, nếu là hộp và ô sau hộp trống thì tạo nước *dẩy* (cập nhật tập hộp). Mỗi trạng thái sinh kèm ký tự tự bước đi để phục hồi lời giải về sau.

```
1 def bfs(testcase):
2     """
3         Run BFS on the given test case and return a move sequence (e.g., 'ULLRU'),
4         or None if no solution exists.
5     """
6     H, W, walls, goals, boxes0, player0 = parse_board(testcase)
7     start = (player0, boxes0)
8     if goal_test(boxes0, goals):
9         return ""
10
11    q = deque([start])
12    visited = set([(player0, boxes0)])
13    parent = {start: (None, None)} # child_state -> (parent_state, move_char)
14
15    while q:
16        cur = q.popleft()
17        player, boxes = cur
18
19        for nxt, move_ch in neighbors(cur, walls, goals, H, W):
20            if nxt in visited:
21                continue
22            visited.add(nxt)
23            parent[nxt] = (cur, move_ch)
24
25            _, boxes_n = nxt
26            if goal_test(boxes_n, goals):
27                # Reconstruct path by following parent pointers
28                path_moves = []
29                s = nxt
30                while parent[s][0] is not None:
31                    prev, ch = parent[s]
32                    path_moves.append(ch)
33                    s = prev
34                return "".join(reversed(path_moves))
35            q.append(nxt)
36
37    return None
```

**Listing 7:** bfs: breadth-first search over (player, boxes)

BFS trên trạng thái tổng hợp (*player, boxes*): khởi tạo với trạng thái xuất phát, nếu đã là đích thì trả chuỗi rỗng. Vòng lặp: lấy đầu hàng đợi, sinh láng giềng, bỏ qua trạng thái đã thăm, lưu cha và kiểm tra đích; khi gặp đích, lần theo parent để dựng chuỗi hành động. Nếu duyệt hết mà không gặp đích, trả *None*. BFS đảm bảo lời giải ngắn nhất theo số bước khi mỗi bước có chi phí bằng nhau.



```
1 def build_solutions():
2     """
3         Run BFS on a batch of test files and write each solution string to 'solutions/
4             testcase_i.txt'.
5         """
6
7     current_dir = os.path.dirname(os.path.abspath(__file__))
8     solutions_dir = os.path.join(current_dir, "solutions")
9     os.makedirs(solutions_dir, exist_ok=True)
10
11    for i in range(1, 21):
12        tc_file = f"mini_cosmos_{i}.txt"
13        result_file_path = os.path.join(solutions_dir, f"testcase_{i}.txt")
14
15        with open(result_file_path, "w") as file:
16            result = bfs(tc_file) # either a move string or None
17            file.write(result if result is not None else "UNSOLVABLE")
```

**Listing 8:** build solutions: batch-run BFS and save results

Hàm tiện ích để chạy hàng loạt test và ghi đáp án: tạo thư mục `solutions` nếu chưa có, chạy `bfs` cho từng tệp `mini_cosmos_i.txt` và ghi chuỗi lời giải hoặc `UNSLVABLE`. Có thể thêm ký tự xuống dòng cuối file để tiện đọc.



### 5.1.2 Giải thuật Heuristic

Các hàm hỗ trợ như parse\_board, is\_inside, is\_free, goal\_test, neighbors được tái sử dụng từ phần triển khai thuật toán Blind Search.

Hàm compute\_heuristic tính toán khoảng cách Manhattan tối thiểu để di chuyển tất cả các hộp đến các vị trí mục tiêu. (mỗi hộp ứng với một vị trí khác nhau)

```
1 def compute_heuristic(boxes, goals):
2     """
3         Heuristic function: sum of Manhattan distances from each goal to the nearest box
4             which has not been computed yet
5     """
6
7     total_distance = 0
8     goals_copy = list(goals.copy())
9
10    for box in boxes:
11        min_dist = float('inf')
12        nearest_goal = goals_copy[0]
13        for goal in goals_copy:
14            dist = abs(box[0] - goal[0]) + abs(box[1] - goal[1])
15            if dist < min_dist:
16                min_dist = dist
17                nearest_goal = goal
18        goals_copy.remove(nearest_goal)
19        total_distance += min_dist
20
21    return total_distance
```

**Listing 9:** Hàm tính giá trị heuristic  $h(n)$

Biến toàn cục heuristic\_cache và hàm cached\_heuristic được sử dụng để tăng tốc độ thuật toán A\* bằng cách áp dụng kỹ thuật lưu trữ đệm (caching), từ đó tránh việc tính toán lặp lại giá trị heuristic cho cùng một cấu hình vị trí hộp.

```
1 heuristic_cache = {}
2 def cached_heuristic(boxes, goals):
3     key = tuple(sorted(boxes))
4     if key not in heuristic_cache:
5         heuristic_cache[key] = utils.compute_heuristic(boxes, goals)
6     return heuristic_cache[key]
```

**Listing 10:** Hàm lưu trữ "bộ nhớ đệm" heuristic

Hàm triển khai giải thuật A\* để giải quyết bài toán Sokoban. Khác với BFS, A\* sử dụng hàng đợi ưu tiên để luôn mở rộng trạng thái có chi phí ước tính thấp nhất. Đồng thời, chỉ những trạng thái mà đường đi mới đến nó cho chi phí thực tế  $g(n)$  tốt hơn so với chi phí  $g(n)$  đã được ghi nhận trước đó mới được cập nhật và đưa vào hàng đợi.

```
1 def astar(testcase):
2     """
3         A* Search
4         Final function for the heuristic solution
5         input:
6             Testcase file name, eg: "mini_cosmos_1.txt"
```



```
7     output:
8         solutions: The sequence of movements, eg: UULR
9             or None if there is no solution
10        ...
11    h, w, walls, goals, boxes0, player0 = utils.parse_board(testcase)
12    # If the initial state is already the goal state
13    if utils.goal_test(boxes0, goals):
14        return ""
15    start = (player0, boxes0)
16    parent = {start: (None, None)} # state -> (prev_state, move_char)
17    g_val={start:0} # keep track of g(n) values for each state
18    queue=[]
19    g_start=0
20    f_start=g_start+cached_heuristic(boxes0, goals)
21    heapq.heappush(queue, (f_start, g_start, start)) # push the initial state into the
22    priority queue with f(0)=h(0)
23
24    while not queue:
25        _, g_current, state_current = heapq.heappop(queue) # get the state with the
26        lowest f and remove it from the queue
27        if g_current>g_val[state_current]:
28            continue # if this state has a higher f(n) value than the one in the open
29            list, skip it
30            # If the current state is the goal state
31            _, boxes_current = state_current
32            if utils.goal_test(boxes_current, goals):
33                # reconstruct path
34                path_moves = []
35                s = state_current
36                while parent[s][0] is not None:
37                    prev, ch = parent[s]
38                    path_moves.append(ch)
39                    s = prev
40                return "".join(reversed(path_moves))
41            # second, get all next possible states
42            for state_neighbor, move_char in utils.neighbors(state_current, walls, h, w):
43                g_neighbor=g_current+1 # cost from current state to the neighbor state
44                if state_neighbor not in g_val or g_neighbor<g_val[state_neighbor]:
45                    g_val[state_neighbor]=g_neighbor
46                    parent[state_neighbor]=(state_current, move_char)
47                    _, boxes_neighbor = state_neighbor
48                    f_neighbor=g_neighbor+cached_heuristic(boxes_neighbor, goals)
49                    heapq.heappush(queue, (f_neighbor, g_neighbor, state_neighbor)) # push the
50                    next state into the priority queue with f(n)=g(n)+h(n)
51
52    return None
```

Listing 11: Hàm Astar giải bài toán Sokoban

## 5.2 Kiểm thử:

Testcase:



```
1 # Board literal for testing (list of rows; each cell is a single-character string)
2 BOARD = [
3     [" ", "#", "#", "#", "#", "#", "#", " "],
4     ["#", "#", "#", " ", " ", " ", "#", " "],
5     ["#", " ", "$", " ", "#", " ", "#", "#"],
6     ["#", " ", "#", " ", " ", ".", " ", "#"],
7     ["#", " ", " ", " ", " ", "#", " ", "#"],
8     ["#", " ", "#", " ", " ", " ", " ", "#"],
9     [" ", "#", "@", " ", " ", "#", "#", "#"],
10    [" ", "#", "#", "#", "#", "#", " ", " "]
```

**Listing 12:** Bản đồ Sokoban

Kết quả giải thuật BFS và A\* search

**BFS:** UULUURLDDRRURRUULLDDULLDDRRRLDDRRUULUR

**A\* search:** UULUURLDDRRURRUULLDDULLDDRRRLDDRRUULUR



Table 4: Comparison of Blind Search and Heuristic Search for 21 test cases

Test Case	Algorithm	Solution Length	Runtime (s)	Peak Memory (KB)
1	Blind Search	37	0.0021	125.06
	Heuristic	37	0.0023	124.72
2	Blind Search	66	0.0096	479.11
	Heuristic	66	0.0111	351.13
3	Blind Search	70	0.0313	1886.94
	Heuristic	70	0.0447	1696.85
4	Blind Search	71	0.0030	130.98
	Heuristic	71	0.0040	130.98
5	Blind Search	95	0.0403	2158.66
	Heuristic	95	0.0580	1989.85
6	Blind Search	86	0.2723	14410.02
	Heuristic	86	0.3832	14874.43
7	Blind Search	60	0.0171	955.74
	Heuristic	60	0.0290	983.27
8	Blind Search	88	0.1016	6741.60
	Heuristic	88	0.1487	5044.34
9	Blind Search	66	0.0100	523.59
	Heuristic	66	0.0151	470.63
10	Blind Search	85	0.0467	2533.06
	Heuristic	85	0.0700	2294.71
11	Blind Search	70	0.0404	1876.42
	Heuristic	70	0.0520	1685.13
12	Blind Search	88	0.2242	11189.74
	Heuristic	88	0.3376	10604.96
13	Blind Search	70	0.0350	2109.89
	Heuristic	70	0.0526	2183.87
14	Blind Search	90	0.2892	14952.47
	Heuristic	90	0.4460	15611.49
15	Blind Search	63	0.0247	1585.85
	Heuristic	63	0.0311	1093.16
16	Blind Search	88	0.0858	4130.62
	Heuristic	88	0.1249	4247.77
17	Blind Search	60	0.0412	2111.03
	Heuristic	60	0.0557	1925.24
18	Blind Search	90	0.3455	16726.96
	Heuristic	90	0.5380	17571.90
19	Blind Search	66	0.0380	2171.42
	Heuristic	66	0.0586	2001.31
20	Blind Search	96	0.4587	15853.77
	Heuristic	96	0.4587	15853.77
21	Blind Search	2000	40.0688	1775751.89
	Heuristic	2000	66.8997	1341177.08

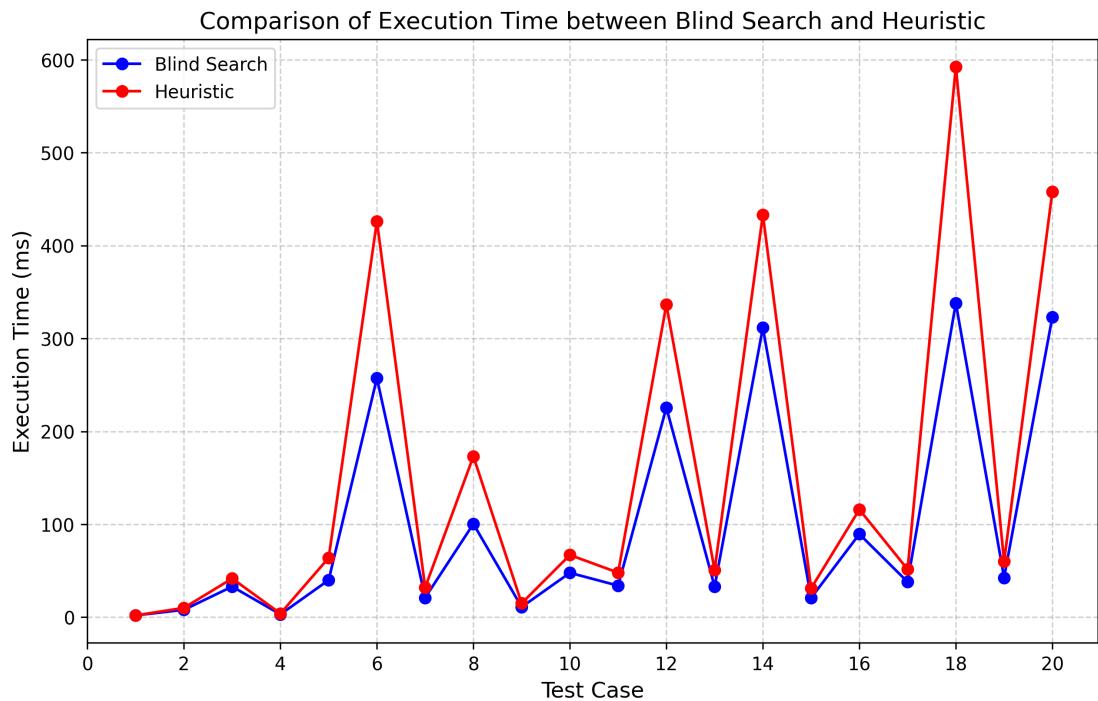


Figure 3: So sánh thời gian thực thi (ms) giữa thuật toán BrFS và A\* Search.

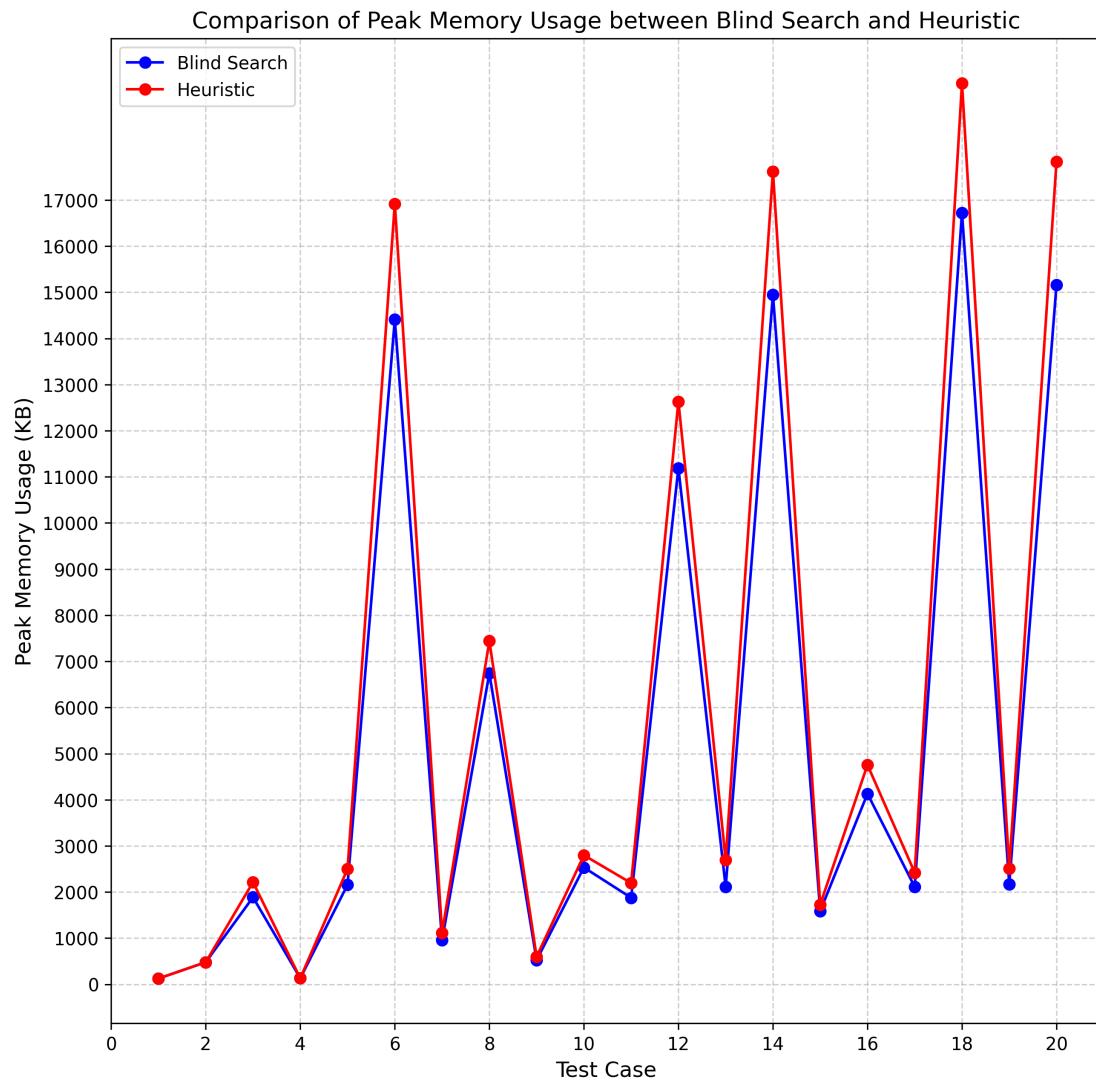


Figure 4: So sánh dung lượng bộ nhớ tiêu tốn (KB) giữa thuật toán BrFS và A\* Search.



Trong một số testcase, A\* tiêu tốn bộ nhớ và thời gian hơn so với Breadth-First Search (BFS) do phải lưu danh sách trạng thái mở (open list) và danh sách trạng thái đã thăm (closed list) để truy vết đường đi tối ưu, đồng thời tính toán và lưu trữ giá trị heuristic. Điều này làm tăng khối lượng tính toán và bộ nhớ cần sử dụng, đặc biệt với các bài toán nhỏ hoặc không gian trạng thái hạn chế, khi BFS vẫn có thể tìm lời giải nhanh mà không cần heuristic.

Ngược lại, A\* vượt trội BFS trong các bài toán có không gian trạng thái lớn (testcase 21) và khi có heuristic tốt và có thể được tính toán một cách đơn giản (ước lượng gần đúng chi phí còn lại đến đích). Nhờ ưu tiên mở rộng những trạng thái có khả năng dẫn đến giải pháp trước, A\* giảm đáng kể số trạng thái cần duyệt, tiết kiệm thời gian và bộ nhớ, đồng thời đảm bảo tìm được đường đi tối ưu theo chi phí.

## 6 Giao diện kiểm thử

Địa chỉ Web Demo: [https://mangtre503.github.io/ASSIGNMENT\\_1\\_AI/](https://mangtre503.github.io/ASSIGNMENT_1_AI/)

### 6.1 Mã nguồn

```
1 index.html # main entry file for deployment
2 Frontend/
3 |-- style.css # styles for UI components
4 |-- game.js # common logic functions
5 \-- screens/
6   |-- Map.js # handles player-related functions
7   \-- Menu.js # allows map selection and navigation to the correct level
```

Listing 13: Thư mục dự án

- Ngôn ngữ: javascript, html, css.
- Cách chạy: Mở index.html chế độ with live server.

### 6.2 Các màn hình

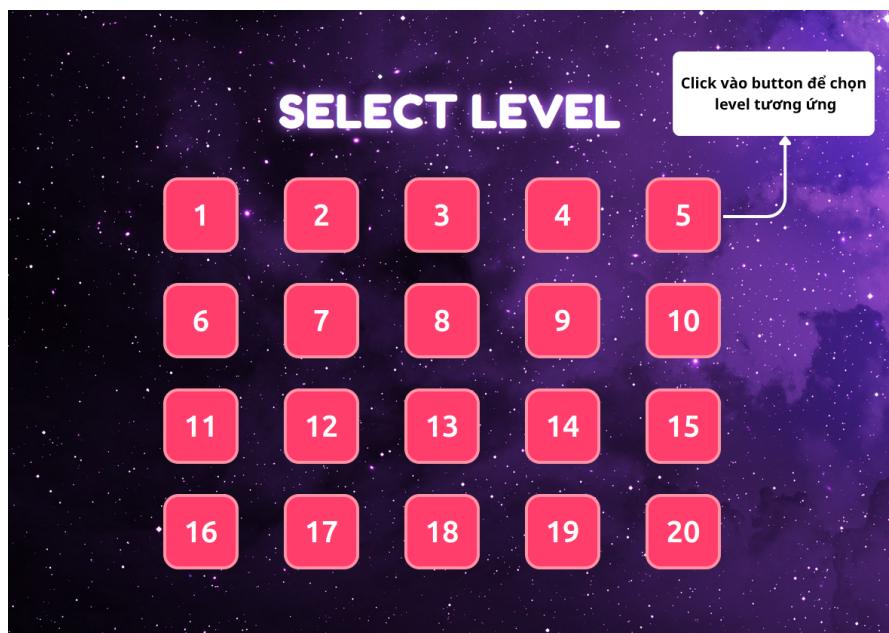


Figure 5: Screen 1: Màn hình Menu lựa chọn Level

Người chơi Click vào các nút để lựa chọn cấp độ (Level) tương ứng, những nút này sẽ điều hướng người chơi đến màn hình lời giải - chơi trò chơi (Screen 2).

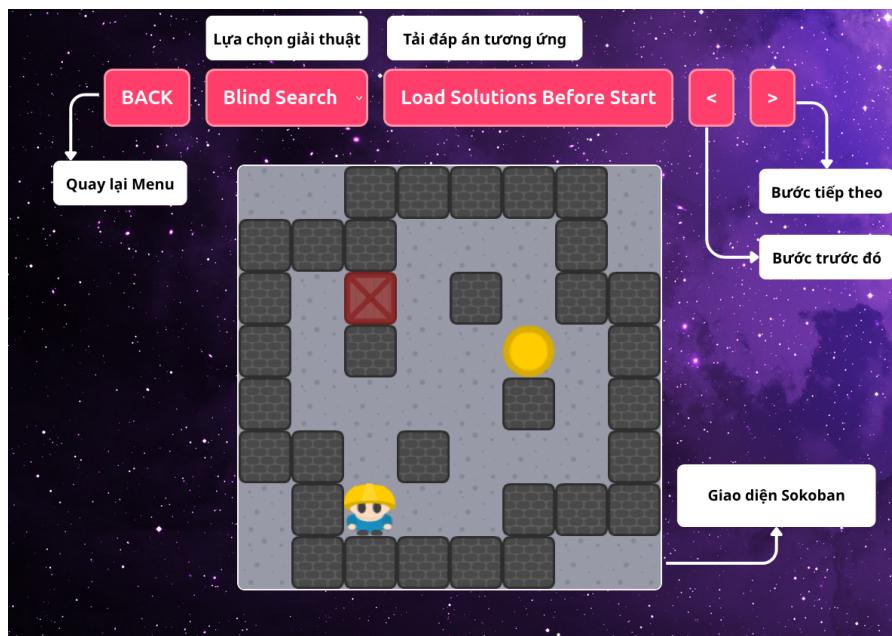


Figure 6: Screen2: Màn hình xem lời giải từ giải thuật Blind search hoặc Heuristic

Screen 2 bao gồm:

- Button 'Back': Nhấn để quay lại Screen 1.
- Select: Lựa chọn giải thuật.
- Button 'Load Solutions Before Start': Nhấn để tải đáp án trước khi bắt đầu.
- Button '<' và '>': Người chơi di chuyển bước tiếp theo dựa trên đáp án từ mô hình hoặc quay lại trạng thái trước đó.
- Giao diện Sokoban: Giao diện trực quan hóa các bước di chuyển của nhân vật dựa trên lời giải.



## 7 Tham khảo và phụ lục

### Tài liệu tham khảo

- [1] Wikipedia contributors. *Sokoban*. Wikipedia, the free encyclopedia. Available at: <https://en.wikipedia.org/wiki/Sokoban>. Accessed: 2025-10-29.
- [2] GeeksforGeeks. *Python for Breadth First Search for a Graph*. Available at: <https://www.geeksforgeeks.org/python-program-for-breadth-first-search-or-bfs-for-a-graph/>. Accessed: 2025-10-29.
- [3] Tim Allan Wheeler. *Basic Search Algorithms on Sokoban*. 2022. Available at: <https://timallanwheeler.com/blog/2022/01/19/basic-search-algorithms-on-sokoban/>. Accessed: 2025-10-29.

### Phụ lục

- [1] Toàn bộ mã nguồn: [https://github.com/Mangtre503/ASSIGNMENT\\_1\\_AI](https://github.com/Mangtre503/ASSIGNMENT_1_AI).