

ĐẠI HỌC QUỐC GIA TP.HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN



BÁO CÁO THỰC HÀNH

MÔN C106.M21: TRÍ TUỆ NHÂN TẠO

LAB: Assignment 1 - BFS/DFS/UCS for Sokoban

Giảng viên hướng dẫn: TS. Lương Ngọc Hoàng

Sinh viên thực hiện:

Họ và tên MSSV

1. Trương Văn Khải 21520274

TP. Hồ Chí Minh, tháng 3, năm 2023

I. Sokoban Game

Game Sokoban là một trò chơi giải đố được tạo ra vào năm 1981. Trong trò chơi này, người chơi điều khiển “quản kho” và cố gắng đẩy các hộp đến vị trí dự kiến. Luật chơi của trò chơi này khá dễ hiểu, tuy nhiên, ở một số “level” khó hơn, nó khó hơn cho chúng ta để có thể giải quyết màn chơi với số lượng phép tính lớn.

Ở bài Report này, em sẽ cố gắng cài đặt các thuật toán tìm kiếm để giải quyết các màn chơi. Việc sử dụng AI để cố gắng phát triển một phương pháp để giải quyết trò chơi Sokoban với các thuật toán DFS, BFS và UCS.

1. Mô hình game Sokoban

1.1 Resource Level

- Các level nằm trong thư mục assets/sokobanLevels được biểu diễn bằng các file .txt và tổng cộng có 18 level.

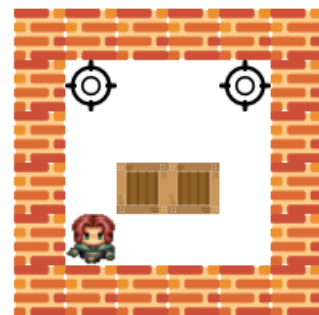
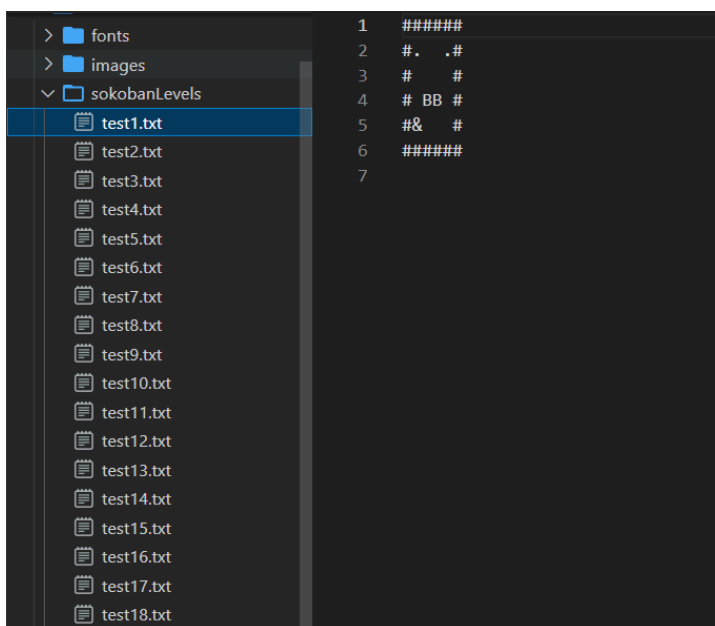


Fig1. Danh sách level game Sokoban và bản đồ level 1

- Các đối tượng trong bản đồ:
 - Đối tượng Player được biểu diễn bằng “&”
 - Đối tượng Space được biểu diễn bằng “.”
 - Đối tượng Target được biểu diễn bằng “.”
 - Đối tượng Invalid Box được biểu diễn bằng “B”
 - Đối tượng Valid Box được biểu diễn bằng “X”
 - Đối tượng Wall được biểu diễn bằng “#”

1.2 Hàm *get_move*

- Input: layout, player_pos, method
 - o Layout: có được từ level.structure được tạo nên từ class Level trong level.py (đọc các file .txt tương ứng).
 - o Player_pos: có được từ position_player được tạo nên từ class Level trong level.py.
 - o Method: Có ba phương án là dfs,bfs,ucs tương ứng với Depth First Search, Breadth First Search và Uniform Cost Search.

1.3 Hàm *auto_move*

- Lựa chọn các thuật toán: Depth First Search (dfs), Breadth First Search (bfs) và Uniform Cost Search (ucs).

1.4 File *Sokoban.py*

- Thực thi để load resources và level từ game thông qua class Game trong game.py hàm load_textures và load_level.
- Phần UX thì sẽ liên quan đến các hàm render và update screen nên bài Report này sẽ không bàn kĩ, chỉ nói kĩ đến phần logic và algorithms.

2. *Trạng thái khởi đầu – kết thúc.*

- Các trạng thái khởi đầu được load từ class Level bao gồm container list chứa dữ liệu được trích xuất từ các file ‘assets/.txt’
- 2 biến global vì chúng cố định theo thời gian thể hiện:
 - + posWalls là kết quả của hàm PosofWalls dãy các tuple giá trị mà tại đó là Wall ứng với *constant* = 1 trong constant.py
 - + posGoals là kết quả của hàm PosofGoals dãy các tuple giá trị mà tại đó là vị trí các goal ứng với *constant* = 3 hay *constant* = 5 trong constant.py
- 2 biến local vì chúng thay đổi theo thời gian thể hiện:
 - + beginBox là kết quả của hàm PosofBoxes là các giá trị ban đầu tuple vị trí của các box
 - + beginPlayer là kết quả của hàm PosofPlayers là giá trị ban đầu tuple vị trí của player
- isEndState là hàm được thiết kế để dùng để kiểm tra xem vị trí các box đã khớp với vị trí các target hay chưa bằng cách sort các vector rồi so sánh chúng với nhau

2.1 *Trạng thái khởi đầu*

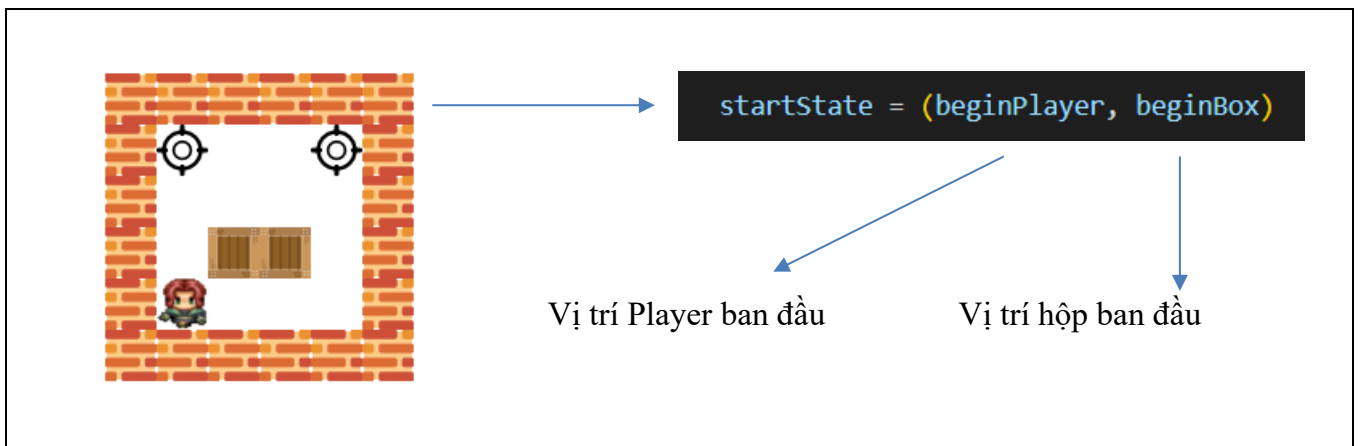


Fig2. Vị trí mở đầu của game

2.2 Trạng thái kết thúc

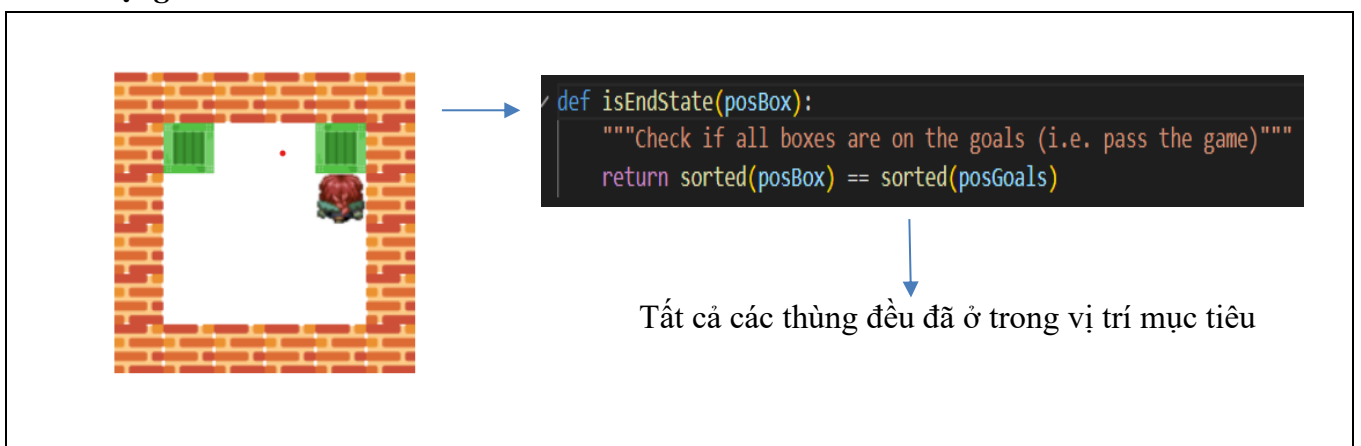


Fig3. Vị trí kết thúc của game

3. Không gian trạng thái

- Không gian trạng thái được quản lí bởi hàm `legaActions`.
- List các hành động `allActions` chứa các cách đi $[x, y, c1, c2]$
 - + x, y là offset so với vị trí hiện tại
 - + $c1, c2$ là l, r, u, d hay L, R, U, D
- $c1, c2$ viết thường tương ứng là việc đi bình thường, viết hoa tương ứng với việc đẩy 1 box.

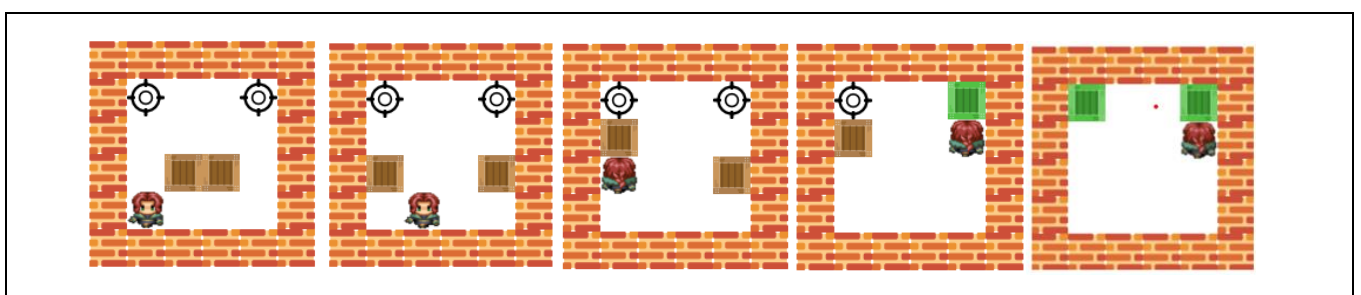


Fig4. Không gian trạng thái của game

4. Các hành động hợp lệ

- Các hành động hợp lệ được xác định bởi hàm `isLegalActions`
- Để xét một hành động được coi hợp lệ thì cần 2 yếu tố:
 - Khi di chuyển: vị trí tiếp theo của nhân vật theo hành động đó không phải là vị trí của tường.
 - Khi đẩy thùng: vị trí tiếp theo của thùng khi đã được đẩy không phải là vị trí của tường hoặc vị trí của một cái thùng khác.

5. Hàm tiền triển (Successor function)

- Một hàm với đầu vào là trạng thái thực hiện và một hành động sau đó tính toán chi phí thực hiện hành động đó cũng như đề xuất ra trạng thái kế tiếp.
- Trong DFS và BFS, khi đưa vào một trạng thái, cả 2 phương pháp tìm kiếm này đều duyệt không gian tìm kiếm một cách ngẫu nhiên với dựa vào trạng thái state đó đã thăm hay chưa và sự hợp lệ của hành động:
 - State đã thăm
 - Sự hợp lệ của hành động (Hành động hợp lệ thỏa mãn hàm hợp lệ)
- BFS, DFS cài đặt theo cơ chế FIFO sử dụng Queue:

```
while frontier:  
    # do BFS theo cơ chế FIFO, sử dụng popleft() để lấy node có giá trị đầu hàng đợi  
    node = frontier.popleft()  
    node_action = actions.pop()
```

- Khác với BFS và DFS, UCS kế thừa từ thuật toán Dijkstra, sử dụng Priority Queue (Min Heap) để đưa vào lựa chọn state thích hợp từ đó từ không gian {Up, Down, Left, Right} đề xuất ra state kế tiếp.

```
def cost(actions):  
    return len([x for x in actions if x.islower()])
```

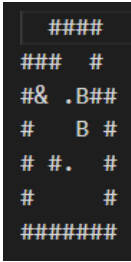
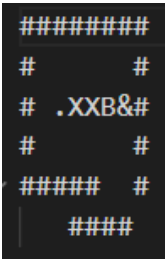
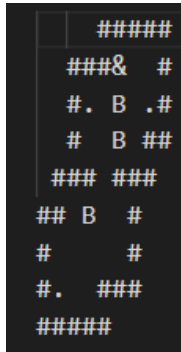
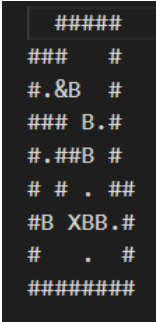
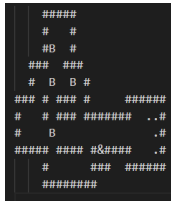
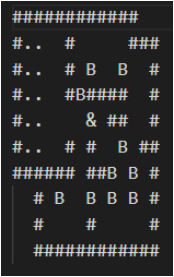
- Hàm `cost` là quãng đường (số actions thực hiện) trong actions (Đi không đẩy thùng). Một action có `cost = 1`.

II. Thống kê kết quả và Nhận xét

1. Thống kê kết quả:

Cấu hình máy:	+ Microsoft Windows 11 + Ram: 8.00 GB + 11th Intel(R) Core(TM) i7-11800U CPU @ 2.30GHz 2304 Mhz, 8 Core(s)
---------------	--

Bản đồ - Trạng thái khởi đầu		Level 1	Level 2	Level 3	Level 4	Level 5	Level 6
DFS	Time	0.05s	0.00s	0.22s	0.00s	Memory Error	0.01s
	Cost	79	24	403	27		55
BFS	Time	0.09s	0.01s	0.16s	0.00s	179.27s	0.01s
	Cost	12	9	15	7	20	19
UCS	Time	0.06s	0.00s	0.08s	0.01s	71.08s	0.01s
	Cost	12	9	15	7	20	19
Bản đồ - Trạng thái khởi đầu		Level 7	Level 8	Level 9	Level 10	Level 11	Level 12
DFS	Time	0.53s	0.08s	0.31s	0.01s	0.02s	0.12s
	Cost	707	323	74	37	36	109
BFS	Time	0.99s	0.18s	0.01s	0.02s	0.01s	0.09s
	Cost	21	97	8	33	34	23
UCS	Time	0.52s	0.20s	0.01s	0.01s	0.02s	0.09s
	Cost	21	97	8	33	34	23

Bản đồ - Trạng thái khởi đầu		Level 13	Level 14	Level 15	Level 16	Level 17	Level 18
							
DFS	Time	0.22s	4.16s	0.19s	Không thể Giải	Không thể Giải	Không thể Giải
	Cost	185	865	291			
BFS	Time	0.14s	3.04s	0.25s	16.57s	Không thể Giải	Không thể Giải
	Cost	31	23	105	34		
UCS	Time	0.18s	2.97s	0.29s	15.94s	Không thể Giải	Không thể Giải
	Cost	31	23	105	34		

2. Nhận xét:

- Ta sẽ so sánh 3 thuật toán DFS, BFS và UCS ở 2 khía cạnh là Thời gian (Time) và Độ dài đường đi (Cost)

Thuật toán	DFS	BFS	UCS
Nhận xét	<p>- Thuật toán DFS cho ra kết quả nhanh chóng trong những màn chơi mức độ dễ, tuy nhiên độ hiệu quả chưa cao, cần phải thực hiện quá nhiều bước đi để đạt tới trạng thái cuối. Như trong chương trình, level 7 DFS tìm ra kết quả tận 707 bước đi trong khi chỉ cần 21 bước ở BFS. Điều này là do DFS tìm ra kết quả theo chiều sâu nên các kết quả được tìm ra nằm ở tầng đáy của cây đồ thị trạng thái, khi này càng sâu thì phải càng thực hiện nhiều bước.</p>	<p>- Lời giải của thuật toán BFS là lời giải tối ưu cho bài toán này vì BFS sẽ tìm kiếm kết quả theo chiều rộng và kết quả đó chắc chắn là kết quả tối ưu bởi vì ở đây, cost của mỗi action luôn luôn là 1.</p> <p>- Thời gian và độ dài đường đi đã được cải thiện đáng kể so với DFS.</p>	<p>- Lời giải của thuật toán UCS là lời giải tối ưu cho bài toán này vì UCS sẽ tìm kiếm kết quả theo thuật toán Dijkstra, và kết quả đó chắc chắn là kết quả tối ưu bởi vì cây tìm kiếm có đều có trọng số = 1 (≥ 0).</p> <p>- Độ dài đường đi của UCS tìm ra tương tự với BFS bởi vì mọi hành động di chuyển có chi phí bằng nhau. Nhưng thời gian của phần lớn level nhỏ hơn BFS.</p>

Đánh giá	<ul style="list-style-type: none"> - Qua thực nghiệm cho thấy UCS cho kết quả tốt hơn DFS và BFS. Lời giải của UCS luôn tối ưu và thời gian thực thi nhanh hơn so với 2 thuật toán kia. - Việc đánh giá thuật toán nào tốt hơn mang tính rất khách quan vì điều đó phụ thuộc rất là nhiều thứ. Nhưng nếu để nói về optimal results thì BFS sẽ chắc chắn tối ưu hơn UCS bởi vì lượng node được push vào priority queue = lượng node được push vào queue vì cost của mỗi action đều = 1. - Nhưng trong bài report này, nếu so sánh dựa trên thời gian và độ dài đường đi thì UCS là thuật toán tốt nhất so với DFS và BFS. Và DFS là tệ nhất.
-----------------	--

3. Bàn luận về những map không giải được:

- Ở map thứ 5:

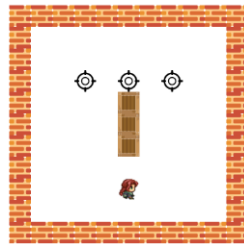


Fig5. Bản đồ trạng thái map 5

- Cả ba thuật toán đều tốn rất nhiều thời gian để giải: DFS (Memory Error), BFS (179.27 s), UCS (71.08 s)
- Không gian trạng thái ở bản đồ level là khá lớn, tức là có rất nhiều ô trống. Điều này dẫn đến các thuật toán sẽ phải cần rất nhiều thời gian để khám phá đường đi.

- Ở map thứ 16, 17, 18:

- Map 16: Chỉ có thuật toán BFS (tốn 16.57 s) và UCS (tốn 0.53 s) là giải được map này. Còn lại, thuật toán DFS không thể giải quyết được màn này. Có lẽ, lí do là bởi không gian trạng thái map 16 khá giống map 5 là có rất nhiều ô trống (tức không gian trạng thái khá lớn), duyệt cây DFS sẽ tìm kiếm lần lượt qua các nhánh và sẽ sinh ra rất nhiều các trạng thái. Vì vậy, DFS không thể bao quát nổi không gian trạng thái này, tức không thể có đủ bộ nhớ (có thể đối với máy em) để giải quyết màn này.

- Vậy ta cần phải tìm một hàm để có thể tối ưu thuật toán DFS

- Map 17, 18: Không có kết quả hay nói cách khác là không tồn tại trạng thái goal state. Em đã cố thử hai map này bằng DFS, BFS, UCS nhưng cả ba giải thuật đều trả về kết quả không tìm thấy đường đi. Nên có lẽ 2 map này sẽ là 2 map khó nhất.