

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA KHOA HỌC MÁY TÍNH



Assignment 1

DFS/BFS/UCS for Sokoban

Môn: Trí tuệ nhân tạo
Lớp: CS106.N21.KHCL
Giảng viên hướng dẫn: Lương Ngọc Hoàng
Sinh viên: Bùi Mạnh Hùng - 21522110

Ngày 28 tháng 3 năm 2023

Mục lục

1	Bài toán Sokoban	3
1.1	Mô hình hóa Sokoban	3
1.1.1	Tài nguyên	3
1.1.2	File Level.py	3
1.1.3	Function 'transferToGameState2()'	4
1.1.4	Function 'get_move()'	4
1.1.5	Function 'auto_move()'	4
1.2	Trạng thái khởi đầu & kết thúc	4
1.2.1	Trạng thái khởi đầu	4
1.2.2	Trạng thái kết thúc	5
1.3	Không gian trạng thái	5
1.4	Các hành động hợp lệ	5
1.5	Hàm tiến triển(successor function)	6
1.5.1	Đối với Depth First Search & Breadth First Search Algorithm	6
1.5.2	Đối với Uniform Cost Search Algorithm	6
2	Thống kê	6
3	Nhận xét	6

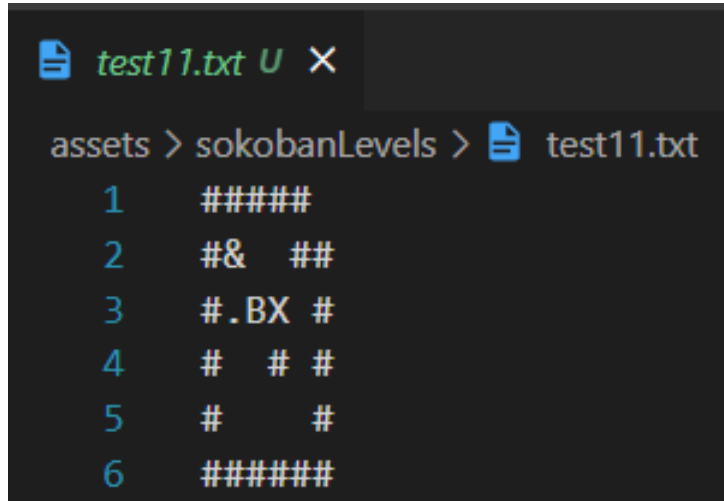
1 Bài toán Sokoban

1.1 Mô hình hóa Sokoban

Sokoban được mô hình hóa dưới dạng bài toán tìm kiếm trạng thái

1.1.1 Tài nguyên

Có 18 levels nằm trong thư mục "../assets/sokobanLevels", biểu diễn dưới dạng file .txt



```
test11.txt U X
assets > sokobanLevels > test11.txt
1 #####
2 #& ##
3 #.BX #
4 # # #
5 # #
6 #####
```

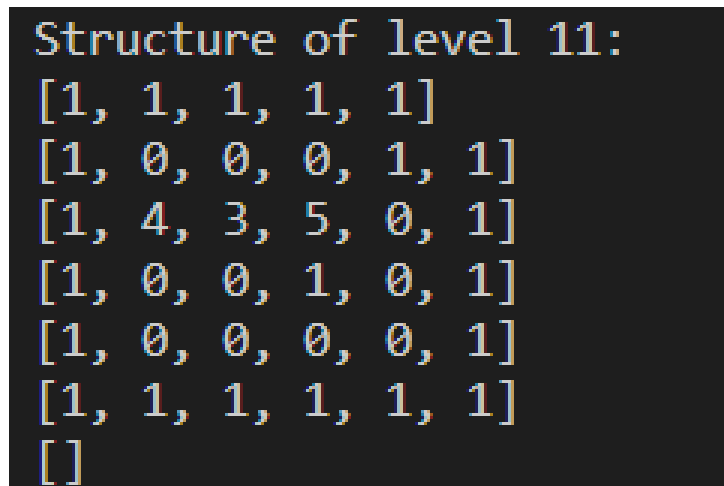
Hình 1: Map of level 11

Chú thích

- ' ': Free space
- '#': Wall
- '&': Player
- 'B': Box
- '.': Goal
- 'X': Box on goal

1.1.2 File Level.py

Dùng để convert những ký tự chữ trong file .txt(Hình 1) thành những ký tự số được định nghĩa trong file constants.py



```
Structure of level 11:
[1, 1, 1, 1, 1]
[1, 0, 0, 0, 1, 1]
[1, 4, 3, 5, 0, 1]
[1, 0, 0, 1, 0, 1]
[1, 0, 0, 0, 0, 1]
[1, 1, 1, 1, 1, 1]
[]
```

Hình 2: Structure of level 11

Chú thích

- 0 : AIR(Free space)
- 1 : WALL
- 2 : PLAYER
- 3 : BOX
- 4 : TARGET(Goal)
- 5 : TARGET_FILLED(Box on goal)

1.1.3 Function 'transferToGameState2()'

Input:

- layout: Lấy từ structure của file level.py
- player_pos: Lấy từ position_player của file level.py

Output: gameState kiểu numpy.ndarray bao gồm layout và vị trí người chơi

1.1.4 Function 'get_move()'

Input:

- layout: Lấy từ structure của file level.py
- player_pos: Lấy từ position_player của file level.py
- method: Thuật toán tìm kiếm (dfs/bfs/ucs/..)

Output: Result & runtime

1.1.5 Function 'auto_move()'

- Lựa chọn thuật toán tìm kiếm để thực thi
- Lưu kết quả chạy vào file .txt

1.2 Trạng thái khởi đầu & kết thúc

1.2.1 Trạng thái khởi đầu

Trạng thái khởi đầu được load lên từ Class Level bao gồm list dữ liệu được trích từ các file .txt tương ứng với mỗi level. => Sau đó nạp vào hàm transferToGameState2() trả về gameState

Lưu trữ vị trí

Hai biến global posWalls & posGoals: Cố định trong thời gian thực thi

- posWalls: giá trị trả về của hàm PosofWalls() có kiểu dữ liệu tuple lưu vị trí của các WALL trong game
- posGoals: giá trị trả về của hàm PosOfGoals() có kiểu dữ liệu tuple lưu vị trí của các GOAL trong game

Hai biến local beginBox & beginPlayer: Thay đổi trong thời gian thực thi

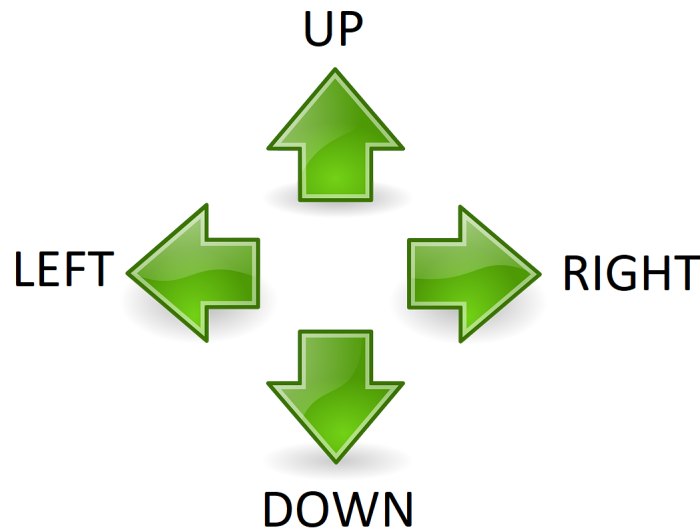
- beginBox: giá trị trả về của hàm PosOfBoxes có kiểu dữ liệu tuple lưu vị trí của các BOX trong game
- beginPlayer: giá trị trả về của hàm PosOfPlayer có kiểu dữ liệu tuple lưu vị trí của Player trong game

1.2.2 Trạng thái kết thúc

Hàm `isEndState()`: kiểm tra vị trí các BOX đã trùng với vị trí các GOAL hay chưa ?
=> Sắp xếp các vector của tập `posBox` và `posGoals` rồi so sánh với nhau

1.3 Không gian trạng thái

Hàm `legalActions`: quản lý không gian trạng thái, player có 4 hướng di chuyển:



Hình 3: 4 directions

Danh sách các hành động có thể thực hiện của Player được lưu trữ trong list `allActions[]` theo dạng `[x,y,a,A]`, trong đó:

- `x,y`: offset so với vị trí hiện tại, có 4 kiểu $\{(-1,0), (1,0), (0,-1), (0,1)\}$
- `a,A`: Hướng đi của player $a \in u, d, l, r$ & $A \in U, D, L, R$
 - `a`: di chuyển bình thường qua vị trí 'Free Space'
 - `A`: di chuyển vào vị trí Box và đẩy Box theo hướng di chuyển 1 offset

Lập các action trong list `AllActions` với vị trí tương ứng (`xPlayer + action[0]`, `yPlayer + action[1]`) với (`xPlayer`, `yPlayer`) là vị trí hiện tại

=> Kiểm tra tính hợp lệ của action trong hàm `isLegalAction()`, dựa vào kết quả trả về để thêm/không thêm action vào list `legalActions[]`

Sau đó, hàm `legalActions()` trả về tập các action hợp lệ.

1.4 Các hành động hợp lệ

Các hành động hợp lệ được xác định bởi hàm `isLegalActions()`

Vị trí xét (`xPlayer + x`, `yPlayer + y`) trong đó (`xPlayer`, `yPlayer`) là vị trí Player hiện tại; (`x,y`) offset của action đang xét. Hành động được gọi là hợp lệ khi vị trí action không đi vào tường hoặc ở trong tường hay *"pos_action not in posBox + posWalls"*

1.5 Hàm tiến triển(successor function)

Successor function là hàm đánh giá ước lượng chi phí còn lại để đạt được trạng thái đích từ trạng thái hiện tại.

1.5.1 Đối với Depth First Search & Breadth First Search Algorithm

Hai thuật toán tìm kiếm này hàm tiến triển không được sử dụng vì chúng không cần đánh giá ước lượng chi phí còn lại. DFS, BFS tìm kiếm đường đi từ trạng thái khởi đầu đến trạng thái đích bằng cách duyệt qua tất cả trạng thái có thể đạt được từ trạng thái khởi đầu.

1.5.2 Đối với Uniform Cost Search Algorithm

Hàm tiến triển giúp tối ưu hóa việc tìm kiếm đường đi bằng cách cập nhật độ ưu tiên của các trạng thái dựa trên tổng chi phí đường đi đã đi qua và ước lượng chi phí còn lại đến trạng thái đích.

Hàm cost trong UCS: tổng quãng đường (số action thực hiện) trong actions(đi không đẩy thùng)

2 Thống kê

Số liệu thống kê các bước thực hiện, runtime: được lưu tại ../sokobanSolver/..

Level	1	2	3	4	5	6	7	8	9
DFS	79	24	403	27	X	55	707	323	74
BFS	12	9	15	7	20	19	21	97	8
UCS	12	9	15	7	20	19	21	97	8

Bảng 1: Số bước chạy của các thuật toán với Level: 1->9

Level	10	11	12	13	14	15	16	17	18
DFS	37	36	109	185	865	291	X	X	X
BFS	33	34	23	31	23	105	34	X	X
UCS	33	34	23	31	23	105	34	X	X

Bảng 2: Số bước chạy của các thuật toán với Level: 10->18

Chú thích:

X: Thuật toán không tìm ra đường đi

3 Nhận xét

Trong game Sokoban, chi phí cho mỗi action đều bằng 1

Lời giải của các thuật toán

Đối với thuật toán Depth First Search

Kết quả thuật toán chưa tối ưu vì DFS tìm kiếm theo độ sâu và dừng lại khi tìm được một lời giải của bài toán nên có thể bỏ qua lời giải tối ưu ở các nhánh khác

Đối với thuật toán Breadth First Search

Kết quả thuật toán là tối ưu vì BFS đảm bảo tìm được đường đi ngắn nhất nếu tồn tại đường đi từ trạng thái hiện tại đến trạng thái đích. Tuy nhiên, thuật toán này có thể tốn nhiều bộ nhớ để lưu trữ tất cả các trạng thái đã duyệt qua.

Đối với thuật toán Uniform Cost Search

Kết quả thuật toán là tối ưu vì nó đảm bảo tìm được đường đi ngắn nhất từ vị trí hiện tại đến vị trí đích nếu tồn tại đường đi. Tuy nhiên trong trò chơi Sokoban, UCS có thể gặp vấn đề độ phức tạp tính toán và thời gian thực thi.

Thuật toán tối ưu nhất

Việc đánh giá thuật toán nào tốt hơn mang tính khách quan vì phụ thuộc vào nhiều yếu tố: thời gian thực thi, không gian lưu trữ ,... Nhưng xét trong phạm vi trò chơi Sokoban với 18 levels và chi phí mỗi action đều bằng 1 thì BFS(Breadth First Search) sẽ tối ưu hơn so với UCS(Uniform Cost Search) vì:

- Số lượng node được push vào hàng đợi $\text{PriorityQueue(UCS)} = \text{Queue(BFS)}$
- Độ phức tạp:
 - BFS: $O((E+V)\log V)$
 - UCS: $O(E+V)$

Với V là số đỉnh, E là số cạnh của đồ thị.

Các bản đồ của bài toán

-Map 17,18,19 không có kết quả dù thử cả 3 thuật toán BFS,DFS,UCS

-Map 5,16 thì thuật toán DFS thực hiện quá lâu, nguyên nhân do:

- Bản đồ lớn và số lượng ô trống nhiều dẫn đến cây tìm kiếm có nhiều trạng thái.
- Thuật toán DFS tìm kiếm theo độ sâu và duyệt hết lần lượt tất cả các nhánh.