ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

KHOA: KHOA HỌC MÁY TÍNH

MÔN: TRÍ TUỆ NHÂN TẠO - CS106.O21.KHTN

# BT1 - DFS/BFS/UCS for Sokoban

*Sinh viên thực hiện:*
Trương Thanh Minh - 21520064

# Mục lục

# 1 Giới thiệu tổng quan

## 1.1 Yêu cầu

- Cài đặt 2 hàm breadthFirstSearch(gameState) và uniformCostSearch(gameState) trong file solver.py.
- Để thay đổi thuật toán chạy thì cần comment/uncomment các dòng lệnh tương ứng trong hàm auto_move() trong file game.py.

## 1.2 Source code

Link github: https://github.com/trthminh/CS106.O21.KHTN/tree/main/sokoban

# 2 Mô hình hóa Sokoban

Sokoban là một trò chơi được tạo bởi Hiroyuki Imabayashi, trong đó người chơi phải điều khiển một agent để đẩy các hộp vào vị trí đích trên một bản đồ có các ô vuông. Bản đồ có các vật cản như tường và hộp. Mục tiêu là phải đẩy tất cả các hộp vào vị trí đích mà không bị kẹt hoặc không còn đường đi nào. Gọi $x, y$ là giá trị tọa độ trong bản đồ, các đối tượng trong trò chơi được tham số hóa như sau:

- **Agent**: posOfPlayer $= (x, y)$.
- **Hộp**: posOfBox $= (x, y)$. Tập hợp các hộp là posOfBoxes = set(posOfBox)
- **Tường**: posOfWall $= (x, y)$. Tập hợp các bức tường là posOfWalls = set(posOfWall)
- **Vị trí đích**: posOfGoal $= (x, y)$. Tập hợp các vị trí đích posOfGoals = set(posOfGoal)

Bài toán Sokuban được mô hình hóa như sau:

- **Trạng thái khởi đầu**: Vị trí ban đầu của **agent** và các **hộp** được đặt trên bản đồ hay nói cách khác [(posOfPlayer, posOfBoxes)].
- **Trạng thái kết thúc**: Một trạng thái mà tất cả các hộp đã được đẩy vào các ô đích trên bản đồ hay sorted(posOfBox) = sorted(posOfGoals)
- **Không gian trạng thái**: $\forall$ [(posOfPlayer, posOfBoxes)]
- **Các hành động hợp lệ**: Người chơi có thể thực hiện các hành động để di chuyển agent trong bản đồ. Các hành động bao gồm: *Up, Down, Left, Right*. Tuy nhiên, người chơi chỉ có thể di chuyển nếu ô đích tiếp theo không phải là tường và nếu có thể đẩy một hộp mà không gây kẹt.
- **Hàm tiến triển (successor function)**: Tất cả các trạng thái tiếp theo có thể đạt được từ một trạng thái hiện tại bằng cách thực hiện một hành động hợp lệ (bao gồm việc di chuyển agent hoặc đẩy 1 hộp).

# 3 Thống kê độ dài đường đi của các thuật toán

| Màn chơi | DFS | BFS | UCS |
|---|---|---|---|
| **1** | 79 | 12 | 12 |
| **2** | 24 | 9 | 9 |
| **3** | 403 | 15 | 15 |
| **4** | 27 | 7 | 7 |
| **5** | × | 20 | 20 |
| **6** | 55 | 19 | 19 |
| **7** | 707 | 21 | 21 |
| **8** | 323 | 97 | 97 |
| **9** | 74 | 8 | 8 |
| **10** | 37 | 33 | 33 |
| **11** | 36 | 34 | 34 |
| **12** | 109 | 23 | 23 |
| **13** | 185 | 31 | 31 |
| **14** | 865 | 23 | 23 |
| **15** | 291 | 105 | 105 |
| **16** | × | 34 | 34 |
| **17** | × | × | × |

Bảng 1: Bảng thống kê về độ dài đường đi

Trong đó, DFS chạy không ra trong màn 5 và màn 16. Bên cạnh đó, bằng mắt thường ta có thể thấy màn 17 không có lời giải, và sau khi chạy 3 thuật toán thì nó cũng không cung cấp cho ta lời giải.

# 4 Nhận xét

Từ bảng 1, ta có một số nhận xét về lời giải (đường đi) được tìm ra ở mỗi thuật toán:

- **DFS**: Ta thấy rằng thuật toán luôn tìm được lời giải (nếu có, vì có tập *exploredSet* nhằm mục đích: những đỉnh đã đi rồi thì không đi lại nữa) cho các màn chơi, tuy nhiên lời giải tìm được không đảm bảo là lời giải tối ưu. Dễ dàng thấy trong màn 5 và 16, không gian trạng thái khá lớn, mà DFS là thuật toán đi sâu vào một nhánh do đó, tốn thời gian và bộ nhớ khi đi vào các nhánh không có lời giải.

- **BFS**: Thuật toán luôn tìm được lời giải tối ưu (nếu có lời giải, ở đây BFS tối ưu là do chi phí của mỗi đường đi là 1). Tuy nhiên nếu không gian trạng thái lớn thì sẽ rất tốn thời gian và bộ nhớ (thậm chí là tốn bộ nhớ hơn DFS) vì mỗi lần BFS sẽ chọn node nông nhất để mở rộng hay nói cách khác, nó sẽ mở rộng dần cho các node có cùng cấp.

- **UCS**: Thuật toán luôn tìm được lời giải tối ưu về mặt chi phí (nếu tồn tại lời giải), nhưng thời gian tìm kiếm lời giải **thường** tốt hơn so với BFS. Sự tối ưu về mặt thời gian này chủ yếu là do cấu trúc dữ liệu heap, giúp mỗi lần nó có thể lấy ra node có

chi phí nhỏ nhất. Ngoài ra, sự tối ưu này cũng đến từ cách thiết kế hàm cost tốt, giúp ước lượng hiệu quả chi phí thực sự.

Nhìn chung, trong bài toán Sokoban, nếu mục tiêu của chúng ta chỉ đơn giản là tìm đường đi tới trạng thái đích, không quan tâm về thời gian và bộ nhớ thì ta có thể dùng 1 trong 3 thuật toán DFS, BFS và UCS. Vì cả ba thuật toán đều cho ta một đường đi tới trạng thái đích (nếu có đường đi).

Tuy vậy, trên thực tế, việc tìm lời giải tối ưu là rất cần thiết. Trong trường hợp đó, DFS tệ hơn 2 thuật toán BFS và UCS khi xét về thời gian chạy và độ dài đường đi. Còn khi xét giữa BFS và UCS, ta thấy không có thuật toán nào tốt hơn trong mọi hoàn cảnh. Cụ thể trong trò chơi này, ta thấy **phần lớn** các màn chơi UCS thường chạy nhanh hơn BFS, nhưng vẫn có một số màn chơi BFS chạy nhanh hơn UCS.

Trong các bản đồ trong game này, màn 5 là màn khó nhất vì khi chạy, các thuật toán đều gặp khó khăn ở màn 5. Điều này được thể hiện rõ thông qua số trạng thái của bản đồ 5:

$$\begin{aligned}\text{totalStates} &= \text{numberPlayerStates} \times \prod_{i=1}^{3} \text{numberBoxStates}\,[i] \\ &= 81 \times 80 \times 79 \times 78 = 39,929,760(\text{ trạng thái })\end{aligned}$$

Nếu xét về số trạng thái thì tại sao bản đồ 16 có số lượng trạng thái lớn hơn rất nhiều (235,989,936,000 trạng thái) nhưng lại không khó bằng bản đồ 5? Đó là bởi vì bản đồ 5 có diện tích 81 lớn hơn bản đồ 16 (diện tích là 31), số lượng box của màn 16 nhiều hơn so với bản đồ 5. Ngoài ra, bản đồ 16 lằng nhằng hơn, chứ không trống trải, đơn giản như màn 5. Từ đó, bằng mắt có thể thấy được màn 16 tuy nhiều trạng thái, nhưng cũng có nhiều trạng thái dẫn tới thất bại. Do đó, theo em màn 5 là màn khó nhất.