# BÁO CÁO VỀ THUẬT TOÁN DFS/BFS/UCS CHO SOKOBAN

**Sinh viên: Hà Huy Hoàng - MSSV: 22520460**

Ngày 17 tháng 3 năm 2024

## 1. Mô tả Sokoban đã được mô hình hóa ra sao?

**Trò chơi Sokoban** được xây dựng dựa trên hình tượng một nhân vật tên **Sokoban** với nhiệm vụ là đẩy các hộp vào các vị trí đích, tất nhiên là bao gồm việc không để các hộp hay nhân vật bị kẹt vào góc không thể di chuyển được. Trò chơi gồm 16 màn (**levels**) được biểu diễn bởi ma trận các kí tự, với **'#'** tượng trưng cho bức tường, **'.'** là đích của các hộp cần đến, **'X'** là vị trí hộp đã tới đích, **' '** là không gian trống, **'B'** và **'&'** lần lượt là vị trí các hộp và nhân vật. Tương ứng với ma trận trên thì hàm **transferToGameState2()** trong **solver.py** đã chuyển kí tự tương ứng với số (**0** là không gian trống, **1** là bức tường, **2** là vị trí nhân vật, **3** là vị trí hộp, **4** là vị trí đích các hộp cần đến, **5** là vị trí các hộp đã tới đích).

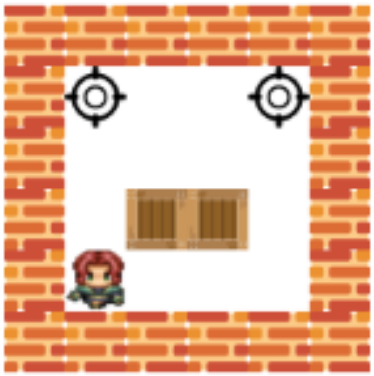
**Trò chơi Sokoban** có 8 cách di chuyển: 4 cách di chuyển không tương tác với hộp (**'r'** là di chuyển nhân vật sang phải 1 ô , **'l'** là di chuyển nhân vật sang trái 1 ô, **'u'** là di chuyển nhân vật lên 1 ô, **'d'** là di chuyển nhân vật xuống 1 ô) và 4 cách di chuyển tương tác với hộp (**'R'** là di chuyển hộp và nhân vật sang phải 1 ô, **'L'** là di chuyển hộp và nhân vật sang trái 1 ô, **'U'** di chuyển hộp và nhân vật lên 1 ô, **'D'** là di chuyển hộp và nhân vật xuống 1 ô)

### 1.1.1. Trạng thái khởi đầu

**Trạng thái khởi đầu** là trạng thái các hộp và nhân vật lúc bắt đầu màn chơi được lấy từ địa chỉ **.../assets/sokobanLevels/**. Trong thuật toán thì nó sẽ biểu diễn dưới dạng 1 tuple. Phần tử đầu tiên của tuple sẽ lưu vị trí bắt đầu của nhân vật, phần tử thứ hai thì sẽ lưu vị trí ban đầu của các hộp. Lấy ví dụ về màn 1 thì trạng thái khởi đầu của nó là: **((4, 1), ((3, 2), (3, 3)))**

```
# # # # # #
# .       . #
#           #
#   B B     #
# &         #
# # # # # #
```

**Hình 1.1.1.** Màn 1 dưới dạng text



**Hình 1.1.2.** Màn 1 dưới dạng ảnh

```
[[1. 1. 1. 1. 1. 1.]
 [1. 0. 0. 0. 0. 1.]
 [1. 0. 3. 3. 0. 1.]
 [1. 2. 0. 0. 0. 1.]
 [1. 1. 1. 1. 1. 1.]]
```

**Hình 1.1.3.** Màn 1 dưới dạng số

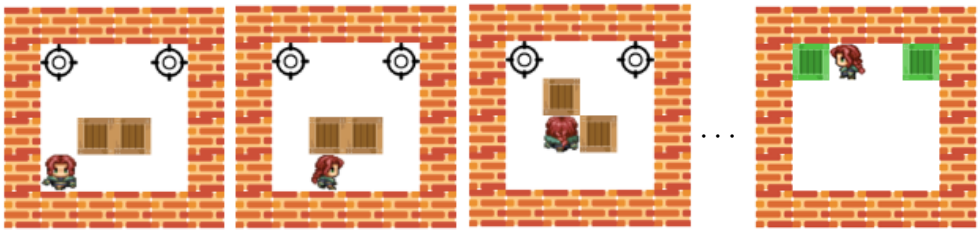
## 1.2. Trạng thái kết thúc

**Trạng thái kết thúc** (hay trạng thái chiến thắng) là trạng thái tất cả các hộp đều tới đích. Trong thuật toán thì nó sẽ biểu diễn dưới dạng 1 tuple. Hai phần tử sẽ lưu vị trí cuối cùng khi kết thúc màn của nhân vật và các hộp. Việc kiểm tra xem khi nào trạng thái kết thúc sẽ thông qua hàm **isEndState()** để kiểm tra các vị trí hộp hiện tại nếu vị trí tất cả các hộp (posBox) và tất cả các đích (posGoals) đều giống nhau thì khi ấy sẽ là trạng thái kết thúc. Lấy vị dụ màn 1 với thuật toán BFS thì trạng thái kết thúc sẽ có giá trị là: **((1, 2), ((1, 4), (1, 1)))**

**Hình 1.2.** Màn 1 khi kết thúc (chiến thắng) dùng BFS

## 1.3. Không gian trạng thái

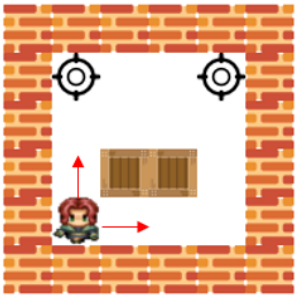
**Không gian trạng thái** là không gian lưu trữ tất cả trạng thái của nhân vật và các hộp (kể cả trạng thái bắt đầu lẫn trạng thái kết thúc). Trong thuật toán thì nó cũng lưu dưới dạng tuple chứa nhiều trạng thái khác nhau. Trong màn 1 khi dùng BFS thì không gian trạng thái sẽ là **[((4, 1), ((3, 2), (3, 3))), ((4, 2), ((3, 2), (3, 3))), ((3, 2), ((3, 3), (2, 2))), ((2, 2), ((3, 3), (1, 2))), ((3, 2), ((3, 3), (1, 2))), ((3, 3), ((1, 2), (3, 4))), ((4, 3), ((1, 2), (3, 4))), ((4, 4), ((1, 2), (3, 4))), ((3, 4), ((1, 2), (2, 4))), ((2, 4), ((1, 2), (1, 4))), ((2, 3), ((1, 2), (1, 4))), ((1, 3), ((1, 2), (1, 4))), ((1, 2), ((1, 4), (1, 1)))]**



**Hình 1.3.** Không gian trạng thái của màn 1 khi dùng BFS

## 1.4. Các hành động hợp lệ

**Các hành động hợp lệ** là các hành động có thể thực hiện trong một trạng thái nào đó bao gồm di chuyển tương tác với hộp và di chuyển không tương tác với hộp. Ví dụ khi màn 1 bắt đầu thì nhân vật chỉ có hai hướng di chuyển: 1 là đi sang phải 1 ô, 2 là đi lên trên 1 ô. Các hành động này được lưu trong một tuple dựa trên vị trí của nhân vật và các hộp tại trạng thái hiện tại thông qua hàm **legalActions()**. Lúc màn 1 bắt đầu thì sẽ trả về: **((-1, 0, 'u'), (0, 1, 'r'))**



**Hình 1.4.** Các hành động hợp lệ khi màn 1 bắt đầu

## 1.5. Hàm tiến triển (successor function) là gì?

Hàm tiến triển trong trường hợp này là hàm **updateState()**, có tác dụng dùng để cập nhật lại trạng thái của nhân vật hiện tượng sau khi thực hiện một hành động.

# 2. DFS/BFS/UCS cho Sokoban

| Màn | 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 |

**Bảng 1.** Bảng thống kê số đường đi

| Màn | DFS | BFS | UCS |
|---|---|---|---|
| 1 | 0.14s | 0.26s | 0.23s |
| 2 | 0.01s | 0.03s | 0.04s |
| 3 | 0.44s | 0.33s | 0.27s |
| 4 | 0.01s | 0.03s | 0.03s |
| 5 | - | 279.39s | 225.16s |
| 6 | 0.07s | 0.06s | 0.08s |
| 7 | 1.23s | 1.42s | 1.18s |
| 8 | 0.16s | 0.38s | 0.42s |
| 9 | 0.48s | 0.05s | 0.06s |
| 10 | 0.06s | 0.07s | 0.04s |
| 11 | 0.09s | 0.07s | 0.08s |
| 12 | 0.26s | 0.24s | 0.21s |
| 13 | 0.33s | 0.27s | 0.43s |
| 14 | 6.16s | 4.26s | 5.89s |
| 15 | 0.32s | 0.5s | 0.6s |
| 16 | - | 34.84s | 34.89s |

**Bảng 2.** Bảng thống kê thời gian giải

**Ghi chú:** Màn 5 và màn 16 em không chạy được ạ (đã ngồi đợi hơn 1 tiếng)

## 2.1 DFS

Ta có thể nhận thấy rằng thuật toán DFS thường hoàn thành màn chơi với số bước đi lớn hơn đáng kể so với hai thuật toán còn lại, tuy nhiên thời gian thực hiện không chênh lệch nhiều, thậm chí có thể nhỉnh hơn hai thuật toán còn lại. NHiện tượng này là do DFS tập trung chủ yếu vào việc tìm ra lời giải mà không quan tâm đến số lượng bước đi, chỉ cần khi duyệt qua các nhánh một cách sâu nhất có thể và tìm lời giải.

## 2.2 BFS

Thuật toán này có lời giải đường đi ít bước ngang với UCS nhưng tốn nhiều bộ nhớ nhất để lưu trữ các trạng thái được mở rộng. Nhưng nhìn chung đây là một thuật toán hiệu quả đối với trò chơi Sokoban khi mà mỗi bước đi chỉ được đi 1 bước.

## 2.3 UCS

Thuật toán này là sự kết hợp của dfs và bfs khi mà nó mở rộng trạng thái bằng cách ưu tiên xét những đường đi (số bước đi mà không di chuyển hộp) có chi phí bé nhất. Nên lời giải của bài toán này sẽ được những bước đi mà không di chuyển hộp ngắn hơn nhiều so với DFS.

Về cơ bản trong trường hợp của bài toán Sokoban này, UCS và BFS đều sẽ là sự lựa chọn tốt hơn so với DFS. Thường trong các trường hợp của bài toán tìm đường thì UCS sẽ tối ưu hơn BFS vì chi phí mỗi bước đi sẽ khác nhau nhưng trong bài này chi phí mỗi bước đi là như nhau nên bài này hiệu suất của BFS và UCS là gần như tương đương nhau.

Đối với bài toán này khi sử dụng DFS, BFS, UCS thì màn 15 là khó nhất vì số bước đi trong khi dùng BFS hoặc UCS (không tính DFS vì nó không tối ưu được số đường đi). Thời gian màn 5 có thể chạy lâu nhất (DFS còn không chạy nổi) nhưng chỉ bởi vì không gian trạng thái nó là lớn nhất trong tất cả các màn (kích thước rộng và ít tường ngăn) chứ không phải vì nó khó giải và phức tạp về đường đi.