ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

HỌC PHẦN: TRÍ TUỆ NHÂN TẠO

BÁO CÁO BÀI TẬP 1

XÂY DỰNG THUẬT TOÁN DFS/UFS/UCS CHO TRÒ CHƠI SOKOBAN

Giảng viên hướng dẫn: TS. Lương Ngọc Hoàng

Lớp: CS106.O22

Sinh viên thực hiện: Cao Huyền My – 22520896

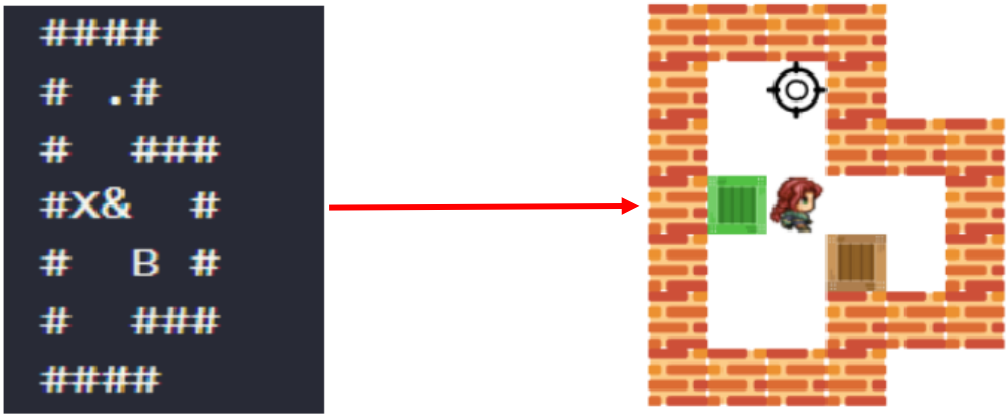
TP. Hồ Chí Minh, tháng 3, năm 2024

I. MÔ HÌNH HÓA TRÒ CHƠI SOKOBAN

1. Các bản đồ (màn chơi) trong Sokoban

Các bản đồ trong Sokoban được lưu vào file .txt và sẽ được biểu diễn bằng các kí tự (mỗi kí tự là một ô có tọa độ) như sau:

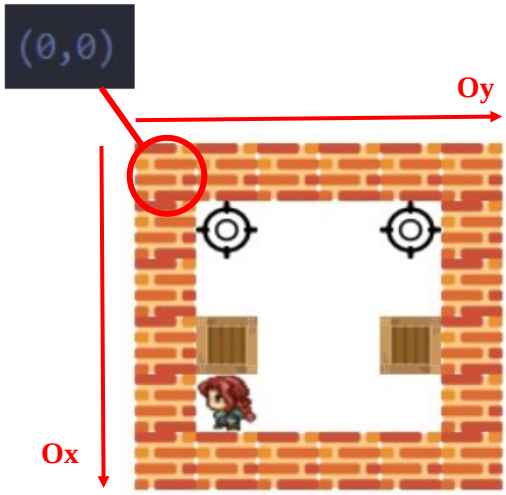
- '**#**': tường chắn (wall)
- '**&**': người chơi (player)
- '**B**': thùng (box)
- '**.**': đích (goal)
- '**X**': thùng nằm trên đích (box on goal)
- ' ': không gian trống



Màn 10

2. Trạng thái khởi đầu

Trạng thái khởi đầu là vị trí ban đầu của người chơi và các thùng.

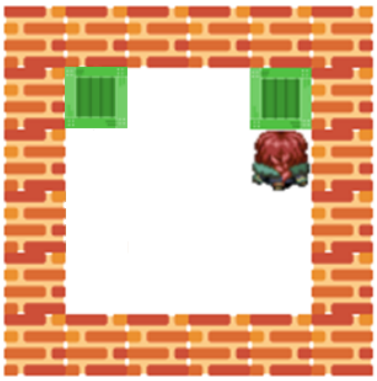


Màn 2

```
startState = (beginPlayer, beginBox)
((4,1) , ((3,1), (3,4)))
```

3. Trạng thái kết thúc

Trạng thái kết thúc là trạng thái mà tất cả các hộp đã nằm trên đích (không cần quan tâm đến vị trí của người chơi).



Vị trí các thùng khi đã nằm trên đích

```
((1,1), (1,4))
```

Màn 2

4. Không gian trạng thái

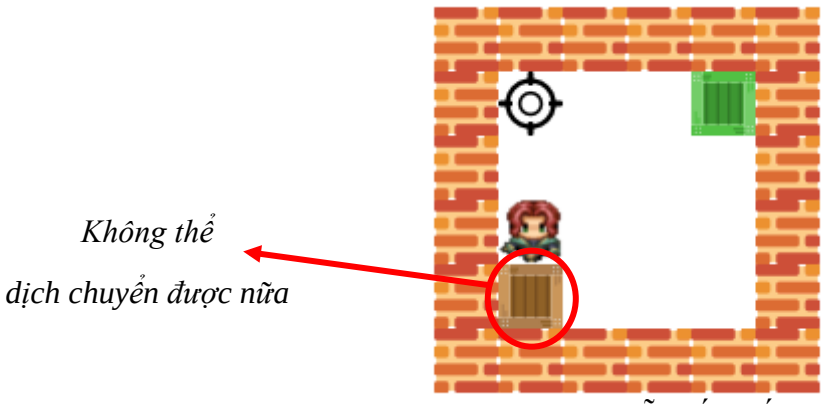
Không gian trạng thái là tập hợp các trạng thái mà mỗi trạng thái bao gồm vị trí người chơi và vị trí của các thùng có thể có trên bản đồ.

5. Các hành động hợp lệ

Hành động hợp lệ là các hành động đưa người chơi *đi lên* (‘u’, ‘U’), *đi xuống* (‘d’, ‘D’), *sang trái* (‘l’, ‘L’), *sang phải* (‘r’, ‘R’) một ô sao cho không di chuyển người chơi đến ô chứa tường chắn, hoặc không đẩy thùng tới ô chứa tường chắn hoặc thùng khác. Hành động có dịch chuyển thùng sẽ được biểu diễn bằng kí tự in hoa, ngược lại sẽ là kí tự thường.

6. Trạng thái dẫn đến thất bại

Trạng thái dẫn đến thất bại là trạng thái có ít nhất một thùng không nằm trên đích nhưng không thể dịch chuyển được nữa dù có thực hiện bao nhiêu hành động.



Một trạng thái dẫn đến thất bại

7. Hàm tiến triển (Successor function)

Hàm tiến triển dùng để xác định các trạng thái tiếp theo có thể đạt được từ một trạng thái hiện tại trong quá trình tìm kiếm.

Các hàm tiến triển trong bài tập này đều là thuật toán tìm kiếm không có thông tin (Uninformed Search) hay thuật toán tìm kiếm điểm mù, gồm: **Depth-First Search, Breadth-First Search, Uniform-Cost Search.**

Trong các thuật toán trên, Sokoban được mô hình hóa là một cây gồm các node. Mỗi node là một dãy các trạng thái liên tiếp từ *trạng thái khởi đầu* đến một trạng thái nào đó (dãy các trạng thái liên tiếp là dãy mà một trạng thái bất kì có thể chuyển qua trạng thái liền sau nó bằng một hành động hợp lệ). Node gốc là chỉ gồm *trạng thái khởi đầu*.

Mỗi node sẽ mở ra các node con từ các hành động hợp lệ mà node đó sinh ra miễn là các hành động đó không tạo ra *trạng thái dẫn đến thất bại*.

Triển khai các thuật toán bằng ***Graph Search***, tức là không xét lại các trạng thái đã xét trước đó (không xét một trạng thái quá 1 lần).

Sử dụng **frontier** để lưu trữ các node, **actions** để lưu trữ các hành động (đường đi) của các node đang nằm trong **frontier**, và **exploredSet** là tập hợp các trạng thái đã xét. Ứng với mỗi thuật toán thì cấu trúc dữ liệu của **frontier** và **actions** sẽ khác nhau.

Tại mỗi lần lặp, một node được lấy ra khỏi **frontier**, nếu trạng thái cuối cùng trong node đó là *trạng thái kết thúc* thì kết thúc thuật toán. Ngược lại, ta sẽ thêm các node con của nó vào **frontier** và đường đi tương ứng vào **actions**. Thuật toán cũng kết thúc khi **frontier** rỗng (tức là không tìm được đường đi tới đích).

Hàm **isFailed()** sẽ được sử dụng để loại bỏ các hành động tạo ra *trạng thái dẫn đến thất bại* trong quá trình duyệt node.

7.1. Depth-First Search (DFS) - Tìm kiếm theo chiều sâu

Trong DFS, cây sẽ được mở rộng theo chiều sâu (duyệt hết nhánh này mới tới nhánh khác), **frontier** và **actions** là các **stack**. Stack là cấu trúc dữ liệu hoạt động theo nguyên tắc "Last In First Out" (LIFO), do đó **node vào sau cùng (node có độ sâu lớn nhất) sẽ được duyệt trước**.

```
def depthFirstSearch(gameState):
    """Implement depthFirstSearch approach"""
    beginBox = PosOfBoxes(gameState)
    beginPlayer = PosOfPlayer(gameState)

    startState = (beginPlayer, beginBox)
    frontier = collections.deque([[startState]])
    exploredSet = set()
    actions = [[0]]
    temp = []
    while frontier:
        node = frontier.pop()
        node_action = actions.pop()
        if isEndState(node[-1][-1]):
            temp += node_action[1:]
            break
        if node[-1] not in exploredSet:
            exploredSet.add(node[-1])
            for action in legalActions(node[-1][0], node[-1][1]):
                newPosPlayer, newPosBox = updateState(node[-1][0], node[-1][1], action)
                if isFailed(newPosBox):
                    continue
                frontier.append(node + [(newPosPlayer, newPosBox)])
                actions.append(node_action + [action[-1]])
    return temp
```

Code thuật toán DFS

7.2. Breadth-First Search (BFS) – Tìm kiếm theo chiều rộng

Trong BFS, cây sẽ được mở rộng theo chiều rộng (duyệt hết bậc này mới tới bậc khác), **frontier** và **actions** là các **queue**. Queue là cấu trúc dữ liệu hoạt động theo nguyên tắc "First In First Out" (FIFO), do đó **node vào sớm nhất (node có độ sâu nhỏ nhất) sẽ được duyệt trước.**

```
def breadthFirstSearch(gameState):
    """Implement breadthFirstSearch approach"""
    beginBox = PosOfBoxes(gameState) # Lấy vị trí khởi đầu của các thùng e.g. ((3,1), (3,4))
    beginPlayer = PosOfPlayer(gameState) # Lấy vị trí khởi đầu của người chơi e.g. (4,1)

    startState = (beginPlayer, beginBox) # Tạo trạng thái khởi đầu gồm vị trí ban đầu của người chơi và các thùng e.g. ((4,1) , ((3,1), (3,4))
    frontier = collections.deque([[startState]]) # Khởi tạo hàng đợi frontier sẽ chứa các NODE với NODE đầu tiên chỉ gồm startState
    exploredSet = set() # Khởi tạo tập hợp chứa các trạng thái đã được xét
    actions = collections.deque([[0]]) # Khởi tạo hàng đợi mà MỖI PHẦN TỬ LÀ MỘT ĐƯỜNG ĐI (dãy các hành động) ứng với một NODE trong frontier theo thứ tự
                                       # e.g. ([0, 'l', 'r'], [0, 'R', l])
    temp = [] # List sẽ chứa đường đi cần tìm từ startState đến goalState (trạng thái kết thúc)
    ### CODING FROM HERE ###
    while frontier: # Lặp qua hàng đợi frontier để mở rộng các node
        node = frontier.popleft() # Lấy ra khỏi frontier node trái cùng (phần tử vào hàng đợi sớm nhất)
        # Ta sẽ xét trạng thái cuối cùng trong node này: node[-1] - gọi trạng thái này là TRẠNG THÁI HIỆN TẠI
        node_action = actions.popleft() # Lấy ra khỏi actions phần tử trái cùng (phần tử vào hàng đợi sớm nhất)
                                        # là đường đi tương ứng với node ở trên, e.g. [0, 'l', 'r']
        if isEndState(node[-1][-1]): # Kiểm tra xem TRẠNG THÁI HIỆN TẠI đó có phải là goalState hay không
            temp += node_action[1:] # Nếu đúng, ta thêm đường đi tương ứng với node đang xét vào temp và dừng vòng lặp
            break
        if node[-1] not in exploredSet: # Ngược lại, kiểm tra TRẠNG THÁI HIỆN TẠI đã được xét chưa
            exploredSet.add(node[-1]) # Nếu chưa được xét, thêm TRẠNG THÁI HIỆN TẠI vào tập exploredSet
            for action in legalActions(node[-1][0], node[-1][1]): # Duyệt qua tất cả các hành động (action) hợp lệ từ TRẠNG THÁI HIỆN TẠI
                                                                  # action có dạng (0, -1, 'l')
                newPosPlayer, newPosBox = updateState(node[-1][0], node[-1][1], action)
    # Cập nhật vị trí mới của người chơi và các thùng dựa trên TRẠNG THÁI HIỆN TẠI và action được TRẠNG THÁI MỚI (gồm vị trí mới của người chơi và thùng)

                if isFailed(newPosBox): # Kiểm tra TRẠNG THÁI MỚI có dẫn tới thất bại không
                    continue # Nếu đúng, bỏ qua vòng lặp hiện tại và tới vòng lặp tiếp theo
                frontier.append(node + [(newPosPlayer, newPosBox)])
                # Ngược lại, thêm TRẠNG THÁI MỚI vào cuối node đang xét để được node mới rồi thêm node mới này vào frontier
                actions.append(node_action + [action[-1]])
    # Thêm action[-1] (e.g 'l') vào cuối đường đi node_action để được đường đi từ startState đến TRẠNG THÁI MỚI rồi thêm đường đi này vào actions
    return temp # Trả về đường đi từ startState đến goalState e.g. ['l', 'r', 'R','u','U'], hoặc trả về mảng rỗng nếu không thể tìm thấy đường đi nào
```

Code thuật toán BFS

7.3. Uniform-Cost Search (UCS) – Tìm kiếm tối ưu chi phí

Chi phí của một đường đi là số bước đi không dịch chuyển thùng.

Khác với DFS và BFS, UCS có xét thêm yếu tố chi phí đường đi, nó sẽ tìm kiếm đường đi có chi phí thấp nhất.

Trong UCS, **frontier** và **actions** là các **priority queue**. Priority queue một cấu trúc dữ liệu hoạt động dựa trên nguyên tắc "ưu tiên cao ra trước" (heap priority). Nó như một hàng đợi nơi các phần tử được sắp xếp theo thứ tự ưu tiên, với phần tử có mức độ ưu tiên cao nhất được lấy ra đầu tiên. UCS sẽ sử dụng chi phí đường đi của node làm độ ưu tiên cho node đó trong **frontier** và đường đi tương ứng với nó trong **actions**. Do đó, **node mà đường đi có chi phí thấp nhất sẽ được duyệt trước.**

```
def uniformCostSearch(gameState):
    """Implement uniformCostSearch approach"""
    beginBox = PosOfBoxes(gameState) # Lấy vị trí khởi đầu của các thùng e.g. ((3,1), (3,4))
    beginPlayer = PosOfPlayer(gameState) # Lấy vị trí khởi đầu của người chơi e.g. (4,1)
    startState = (beginPlayer, beginBox) # Tạo trạng thái khởi đầu gồm vị trí ban đầu của người chơi và các thùng
    frontier = PriorityQueue() # Khởi tạo hàng đợi ưu tiên sẽ chứa các NODE
                               # với độ ưu tiên là chi phí đường đi từ stateStart đến trạng thái cuối cùng trong NODE
    frontier.push([startState], 0) # Thêm node đầu tiên chỉ gồm startState vào frontier với chi phí là 0
    exploredSet = set() # Khởi tạo tập hợp chứa các trạng thái đã được xét
    actions = PriorityQueue() # Khởi tạo hàng đợi ưu tiên mà MỖI PHẦN TỬ LÀ MỘT ĐƯỜNG ĐI ứng với một node trong frontier theo thứ tự
                              # với độ ưu tiên là chi phí đường đi đó
    actions.push([0], 0)  # Thêm vào actions đường đi đầu tiên (chưa có hành động nào) với chi phí là 0
    temp = [] # List sẽ chứa đường đi cần tìm từ startState đến goalState
    ### CODING FROM HERE ###
    while not frontier.isEmpty(): # Lặp qua hàng đợi ưu tiênfrontier để mở rộng các node
        node = frontier.pop() # Lấy ra khỏi frontier node có độ ưu tiên cao nhất (chi phí thấp nhất)
        # Ta sẽ xét trạng thái cuối cùng trong node này: node[-1] - gọi trạng thái này là TRẠNG THÁI HIỆN TẠI
        node_action = actions.pop() # Lấy ra khỏi actions đường đi có độ ưu tiên cao nhất (chi phí thấp nhất) ứng với node ở trên
        if isEndState(node[-1][-1]): # Kiểm tra xem TRẠNG THÁI HIỆN TẠI đó có phải là goalState hay không
            temp += node_action[1:] # Nếu đúng, ta thêm đường đi tương ứng với node đang xét vào temp và dừng vòng lặp
            break
        if node[-1] not in exploredSet: # Ngược lại, kiểm tra TRẠNG THÁI HIỆN TẠI đã được xét chưa.
            exploredSet.add(node[-1]) # Nếu chưa được xét, thêm TRẠNG THÁI HIỆN TẠI vào tập exploredSet
            for action in legalActions(node[-1][0], node[-1][1]): # Duyệt qua tất cả các hành động (action) hợp lệ từ TRẠNG THÁI HIỆN TẠI
                newPosPlayer, newPosBox = updateState(node[-1][0], node[-1][1], action)
            # Cập nhật vị trí mới của người chơi và các hộp dựa trên TRẠNG THÁI HIỆN TẠI và action ta được TRẠNG THÁI MỚI
                if isFailed(newPosBox): # Kiểm tra TRẠNG THÁI MỚI có dẫn tới thất bại không
                    continue # Nếu đúng, bỏ qua vòng lặp hiện tại và tới vòng lặp tiếp theo
                # Ngược lại
                new_action = node_action + [action[-1]]
                # Thêm action[-1] (e.g 'l') vào cuối đường đi node_action để được ĐƯỜNG ĐI MỚI từ startState đến TRẠNG THÁI MỚI
                new_cost = cost(new_action[1:]) # Tính chi phí của ĐƯỜNG ĐI MỚI ở trên
                frontier.push(node + [(newPosPlayer, newPosBox)], new_cost)
                # thêm TRẠNG THÁI MỚI vào cuối node đang xét để được node mới rồi thêm node mới này vào frontier, độ ưu tiên là chi phí ĐƯỜNG ĐI M
                actions.push(new_action, new_cost) #Thêm ĐƯỜNG ĐI MỚI vào actions với độ ưu tiên là chi phí của nó
    return temp # Trả về đường đi có chi phí thấp nhất từ startState đến goalState, hoặc trả về mảng rỗng nếu không thể tìm thấy đường đi nào
```

Code thuật toán UCS

II. BẢNG THỐNG KÊ KẾT QUẢ, NHẬN XÉT VÀ KẾT LUẬN

1. Bảng thống kê kết quả

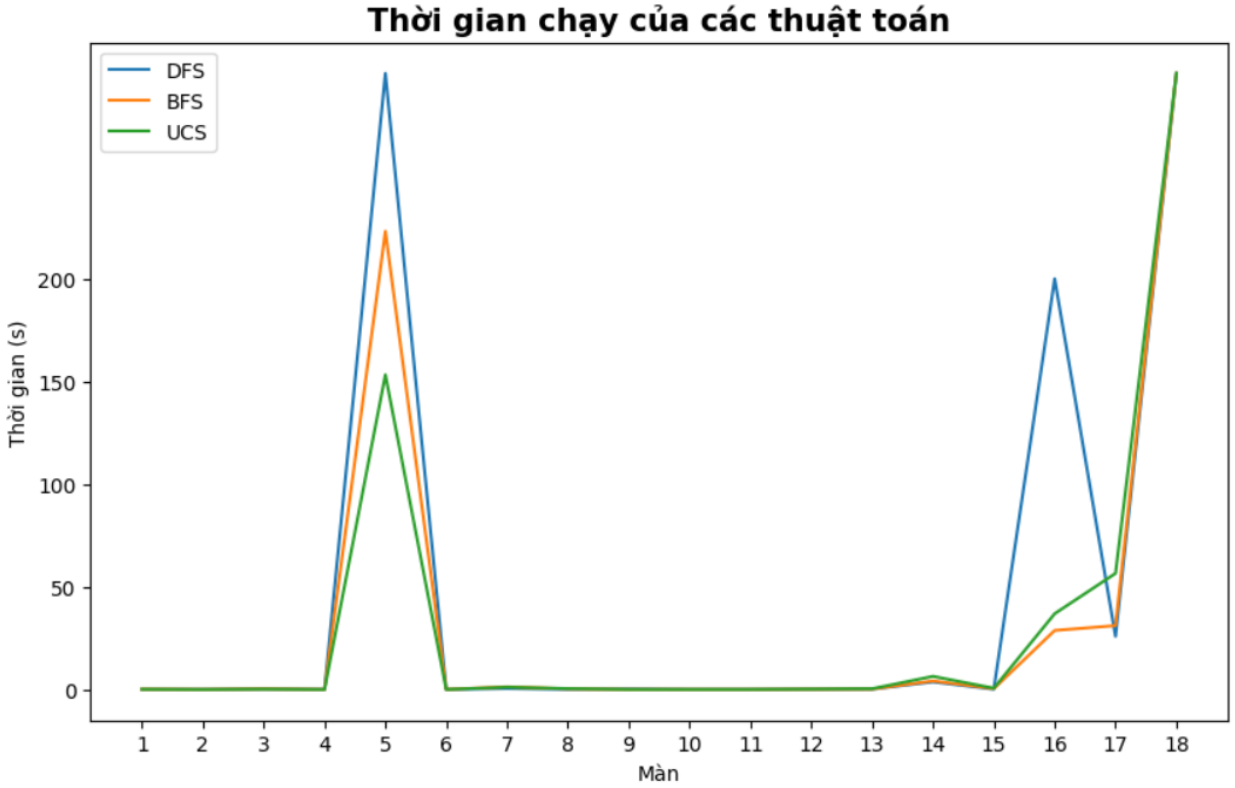
Màn	Thời gian (s)			Số bước đi			Chi phí		
	DFS	BFS	UCS	DFS	BFS	UCS	DFS	BFS	UCS
1	0.05	0.12	0.11	79	12	12	65	6	6
2	0.00	0.01	0.01	24	9	9	20	5	5
3	0.21	0.27	0.19	403	15	15	349	9	9
4	0.01	0.01	0.01	27	7	7	22	4	4
5	∞	223.10	153.26	-	20	20	-	10	10
6	0.01	0.01	0.02	55	19	19	46	14	14
7	0.51	1.28	1.15	707	21	21	638	10	10
8	0.07	0.30	0.45	323	97	97	275	65	65
9	0.25	0.01	0.02	74	8	8	59	5	5
10	0.02	0.02	0.03	37	33	33	29	25	25
11	0.02	0.02	0.04	36	34	34	29	27	27
12	0.13	0.12	0.19	109	23	23	96	16	16
13	0.18	0.20	0.38	185	31	31	164	24	24
14	3.70	4.16	6.42	865	23	23	726	16	16
15	0.15	0.41	0.63	291	105	105	232	62	62
16	∞	28.75	36.91	-	34	34	-	22	22
17	25.88	31.10	56.55	0	0	0	0	0	0
18	∞	∞	∞	-	-	-	-	-	-

** Thời gian biểu diễn bằng ∞ ở các trường hợp mà để máy chạy thuật toán đến khi chương trình báo lỗi và buộc phải restart máy.*

Màn 17 không có lời giải nên cả 3 thuật toán đều trả về mảng rỗng (không có đường đi).

2. Nhận xét

- **Về thời gian:**

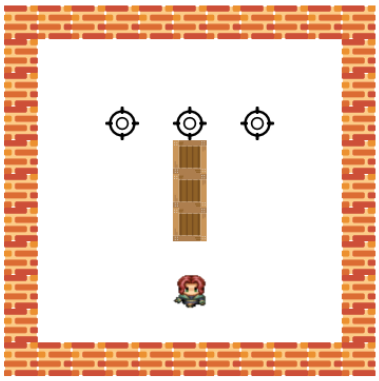


Hầu hết ở các màn chơi mà DFS có thể chạy được thì nó có thời gian chạy nhanh nhất. UCS có chạy nhanh hơn BFS ở 1 số màn đầu nhưng chậm hơn ở các màn sau (do tốn thời gian cho việc tính chi phí).

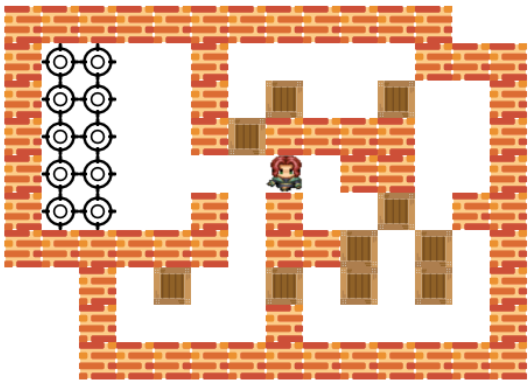
Màn 5 là một màn dễ nhưng lại tốn nhiều thời gian nhất để tìm ra đường đi cho cả 3 thuật toán, đặc biệt là DFS có thể mất thời gian vô hạn. Màn này có cấu trúc đặc biệt với 3 thùng xếp chồng lên nhau và không có tường chắn nào ngoài 4 bức tường bao quanh màn chơi làm cho số lượng hành động hợp lệ lớn hơn rất nhiều so với các màn khác, tạo ra số đường đi khổng lồ. Do đó, các thuật toán cần phải tính toán và đánh giá nhiều khả năng hơn, dẫn đến thời gian thực thi lâu hơn.

Màn 16, UCS và BFS đều tìm được đường đi với thời gian tương đối nhanh (28.88s và 36.91s) thì DFS có thể tốn rất rất nhiều thời gian vẫn chưa giải ra được.

Cả 3 thuật toán đều không thể chạy được **màn 18**. Màn này có tới 10 thùng cần đẩy tới đích, có cấu trúc cực kì phức tạp với nhiều chướng ngại vật (tường chắn) và các thùng được đặt ở vị trí khó di chuyển. Việc di chuyển các thùng qua các chướng ngại vật và sắp xếp chúng vào đích đòi hỏi tính toán rất nhiều và tốn nhiều thời gian. Do đó, đây là màn chơi khó nhất.



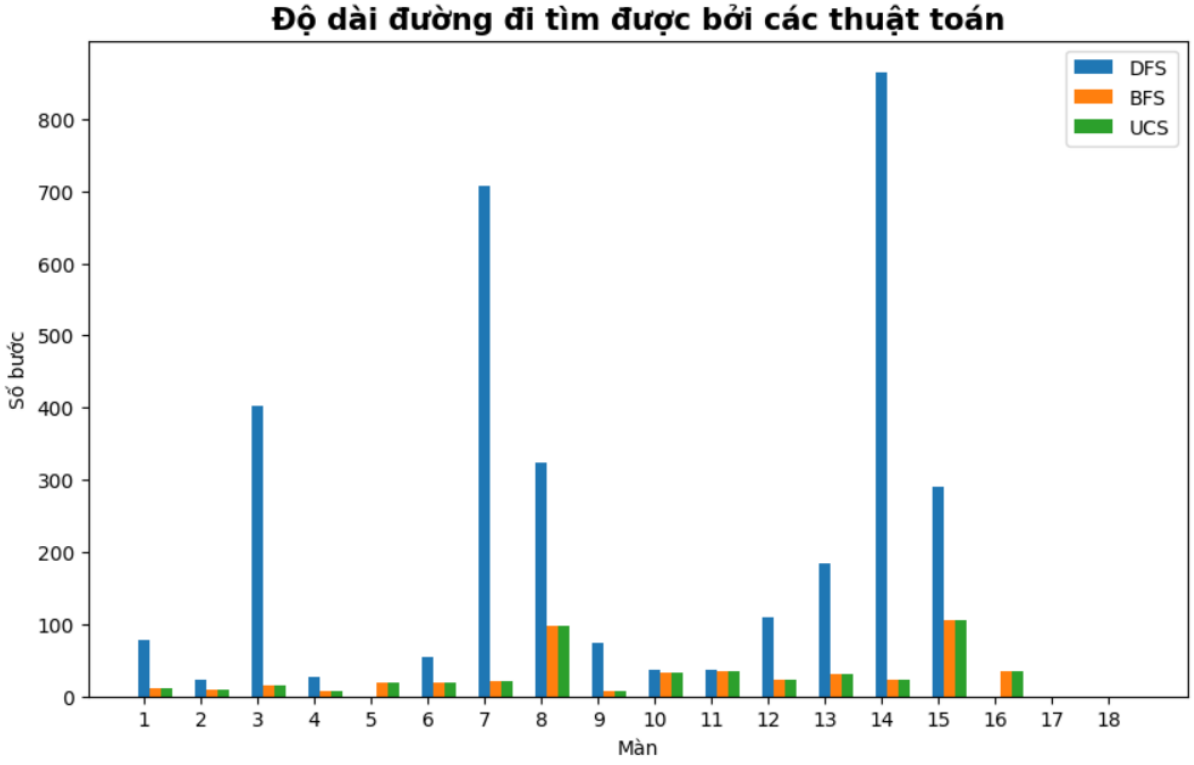
Màn 5



Màn 18

Như vậy, DFS tuy có thời gian chạy nhanh cho nhiều màn nhưng lại không chạy được 3 màn (5, 6 và 18), trong khi hai thuật toán còn lại chỉ không chạy được **màn 18**. BFS và UCS có thời gian chạy ở đa số màn chơi không chênh lệch nhiều. Tuy nhiên, ở **màn 5** (màn có thời gian lâu nhất) thì UCS lại cho thời gian vượt trội hơn (153.26s) so với BFS (223.10s).

- **Về độ dài đường đi:**



Độ dài đường đi (số bước đi) và chi phí tìm được bởi thuật toán DFS lớn hơn rất nhiều lần so với BFS và UCS.

Độ dài đường đi và chi phí đường đi của BFS và UCS là như nhau.

3. Kết luận

Màn 18 là màn khó nhất khi không thuật toán nào chạy được.

UCS là thuật toán tốt hơn cả.