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

BÁO CÁO ĐỒ ÁN

MÔN HỌC: TRÍ TUỆ NHÂN TẠO

ĐỀ TÀI:

SỬ DỤNG THUẬT TOÁN BFS VÀ UCS TRONG TRÒ CHƠI SOKOBAN

Lớp: CS106.L21.KHCL

Giảng viên hướng dẫn:

TS. Lương Ngọc Hoàng

Sinh viên thực hiện:

Phan Nguyễn Thành Nhân 19521943

Thành phố Hồ Chí Minh, ngày 28 tháng 03 năm 2021

MỤC LỤC

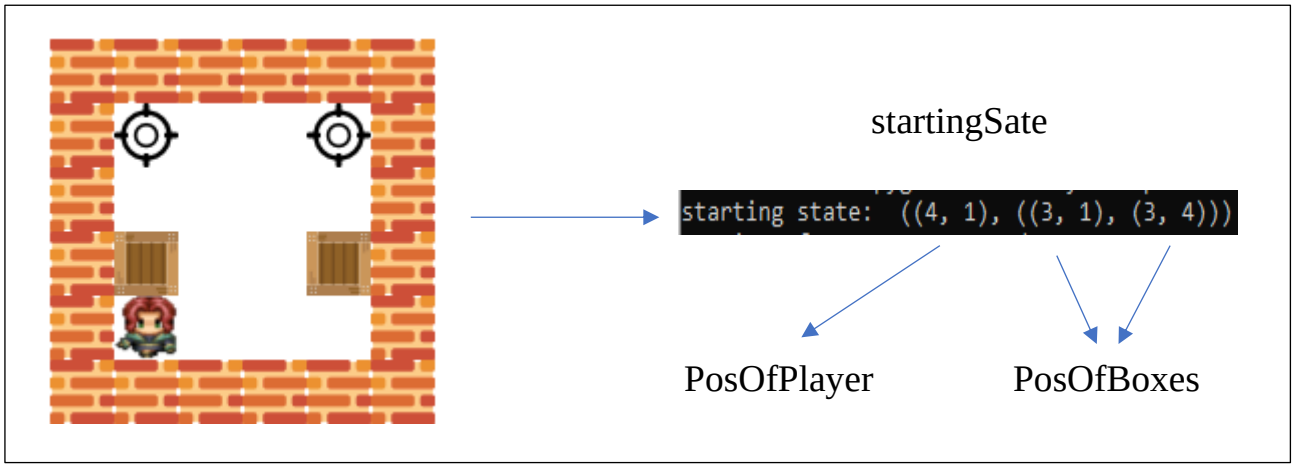
I. Mô hình hóa trong game Sokoban 1
1. Trạng thái mở đầu 1
2. Trạng thái kết thúc 2
3. Không gian trạng thái 2
4. Các hành động hợp lệ 3
II. Các hàm cần thiết 3
III. Hàm tiến triển 4
1. Depth First Search (DFS) 4
2. Breadth First Search (BFS) 6
3. Uniform Cost Search (UCS) 7
IV. So sánh và nhận xét 8

I. Mô hình hóa trong game Sokoban

- Sokoban là một trò chơi giải đố, trong đó người chơi phải đẩy tất cả các thùng vào tất cả các vị trí đích.
- Trò chơi có dạng bảng ô vuông; số tất cả các thùng bằng với số tất cả các đích; người chơi chỉ có thể đẩy từng cái thùng một, không thể đẩy một lúc một dãy hai hay nhiều thùng, và cũng như không thể kéo.
- Dính tường là một trường hợp mà người chơi phải cần tránh. Thùng được coi là dính tường khi nó bị đẩy sát vào góc tường, nếu vị trí đó không phải là đích thì người chơi sẽ bị thua cuộc.

1. Trạng thái mở đầu

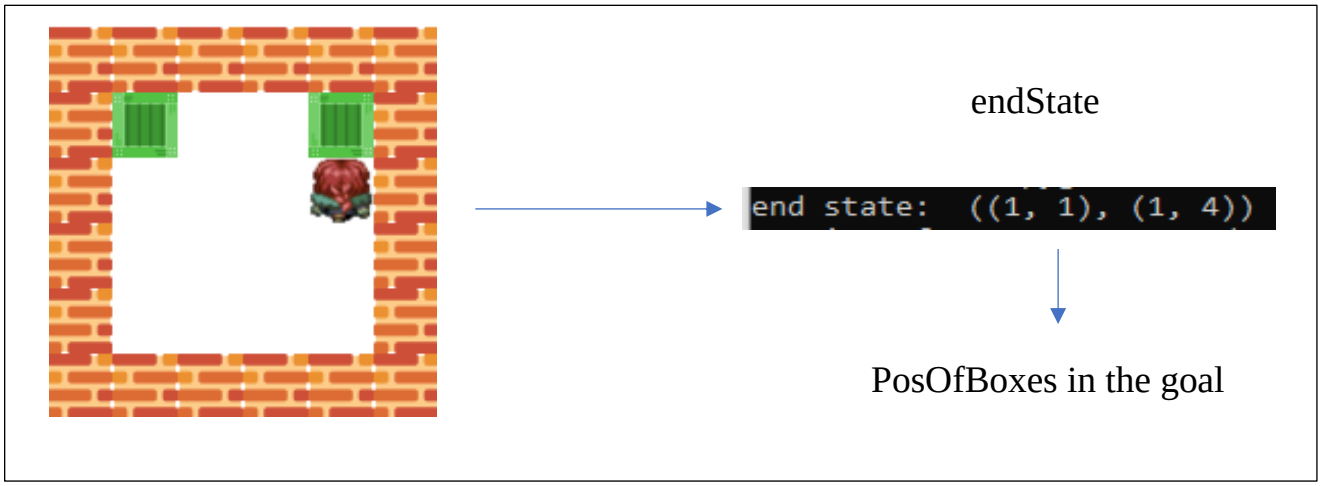
- Trạng thái mở đầu của game là những vị trí đầu tiên của nhân vật và các thùng (như trong Hình 1).



Hình 1: Trạng thái mở đầu của game

2. Trạng thái kết thúc

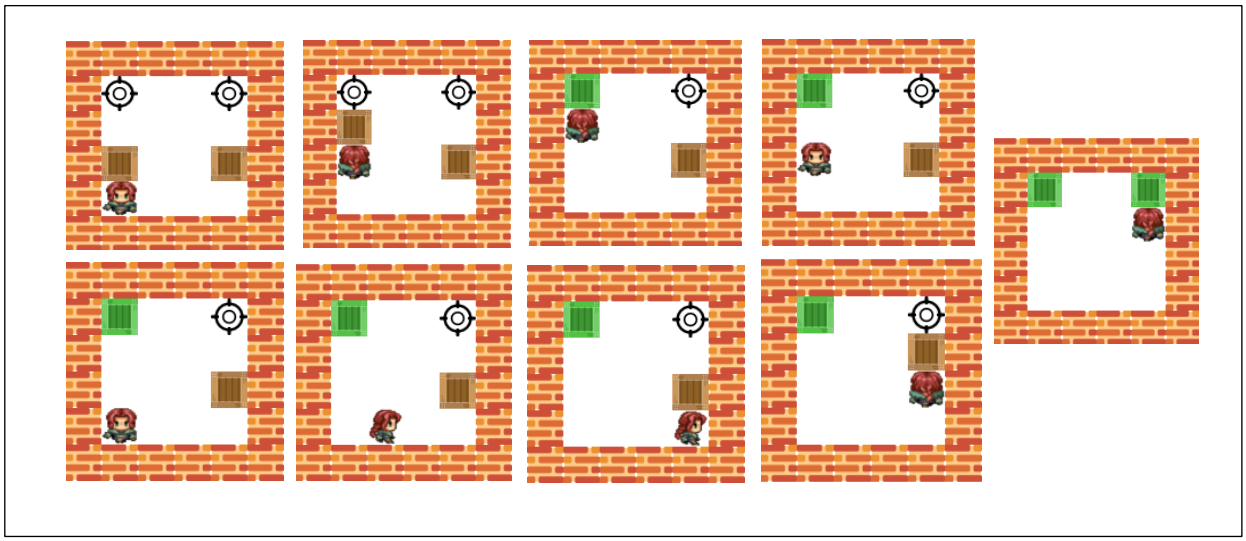
- Trạng thái kết thúc của game là vị trí của các thùng khi đã vào vị trí mục tiêu (goal) (như trong Hình 2).



Hình 2: Trạng thái kết thúc của game

3. Không gian trạng thái

- Không gian trạng thái của game là tất cả các vị trí của các hộp và các vị trí mà nhân vật có thể đi trên bản đồ (như trong Hình 3).



Hình 3: Không gian trạng thái của game

4. Các hành động hợp lệ

- 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.

II. Các hàm cần thiết

- PosOfPlayer(gameState): nhận đầu vào là bố cục của bản đồ từng level, trả về vị trí của nhân vật.
- PosOfBoxes(gameState): nhận đầu vào là bố cục của bản đồ từng level, trả về vị trí của các thùng.
- PosOfWalls(gameState): nhận đầu vào là bố cục của bản đồ từng level, trả về vị trí của các tường.
- PosOfGoals(gameState): nhận đầu vào là bố cục của bản đồ từng level, trả về vị trí của các đích.
- isEndState(gameState): nhận đầu vào là vị trí hiện tại của các thùng, trả về true nếu các thùng đã ở vị trí của các đích và ngược lại.
- isLegalAction(action, posPlayer, posBox): nhận đầu vào là trạng thái tiếp theo của nhân vật, vị trí hiện tại của nhân vật và các thùng; trả về true nếu trạng thái tiếp theo của nhân vật là hợp lệ.
- legalActions(posPlayer, posBox): nhận đầu vào là vị trí hiện tại của nhân vật và các thùng, trả về tất cả các trạng thái hợp lệ.
- updateState(posPlayer, posBox, action): nhận đầu vào là vị trí hiện tại của nhân vật và các thùng, trạng thái tiếp theo của nhân vật; trả về vị trí mới của nhân vật và các thùng.

- isFailed(posBox): nhận đầu vào là vị trí mới của các thùng, trả về True nếu vị trí mới hợp lệ và ngược lại.
- cost(actions): nhận đầu vào các hành động của nhân vật, trả về số bước đi của nhân vật từ trạng thái ban đầu đến trạng thái tiếp theo.

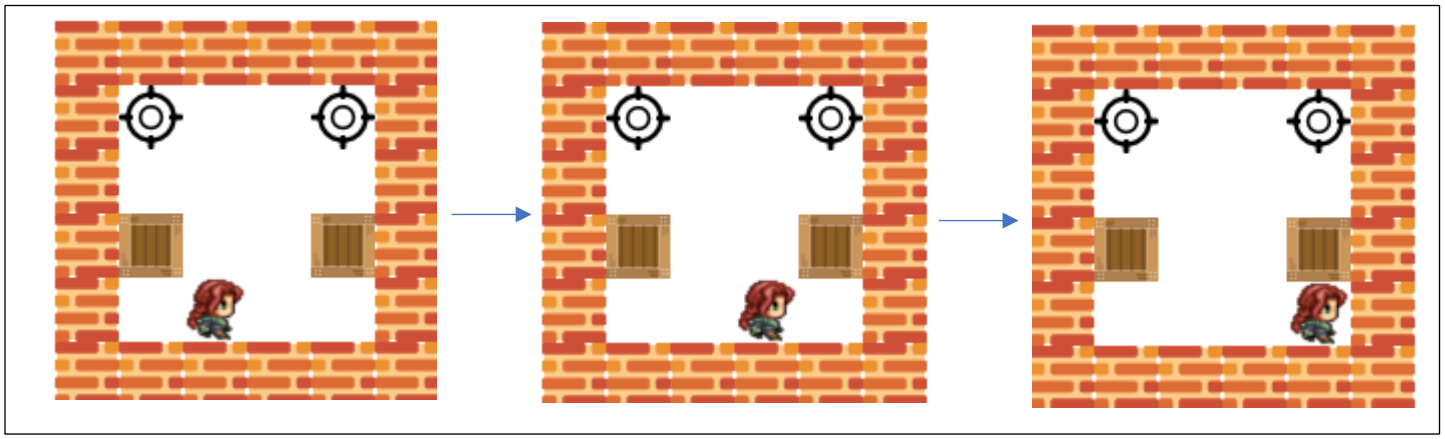
III. Hàm tiến triển

1. Depth First Search (DFS)

- Thuật toán Depth First Search (DFS – Tìm kiếm theo chiều sâu) là hàm tiến triển theo chiến lược tìm kiếm mù (tìm kiếm không có định hướng, không có thông tin, giá trị được duyệt)
- DFS hoạt động dựa trên cấu trúc stack (LIFO), duyệt đi xa nhất theo từng nhánh, khi nhánh đã duyệt hết lùi về từng đỉnh để duyệt những nhánh tiếp theo, và thuật toán chỉ dừng lại khi tìm thấy đỉnh cần tìm.

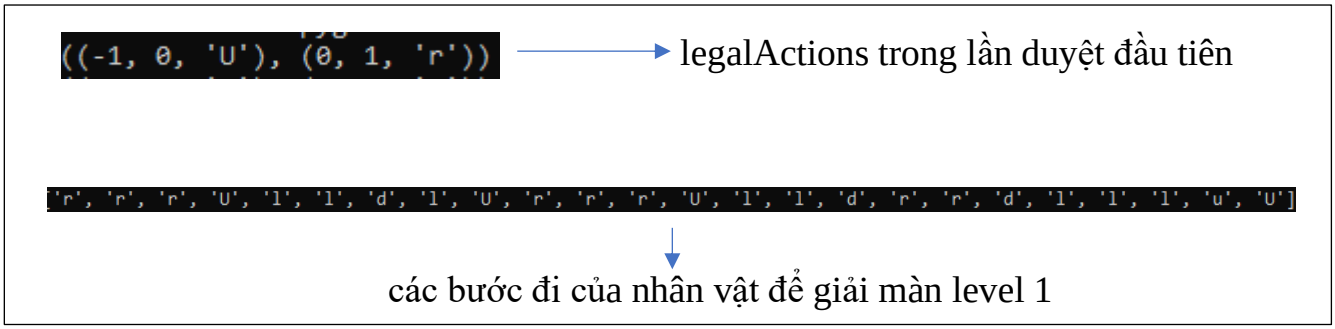
```
def depthFirstSearch(gameState):
    """Implement depthFirstSearch approach"""
    beginBox = PosOfBoxes(gameState)
    beginPlayer = PosOfPlayer(gameState)
    startState = (beginPlayer, beginBox)
    frontier = collections.deque([[startState]])

    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
```

- DFS hoạt động dựa trên cấu trúc stack, nên khi bắt đầu duyệt DFS sẽ lấy ra state ngoài cùng (cuối cùng) của frontier để duyệt.
- Điểm yếu của thuật toán DFS là tốn rất nhiều bước đi của nhân vật để có thể giải được bài toán.
- Như trong Hình 4, thay vì đẩy thùng gần nhất nhân vật trong game lại chọn đẩy thùng xa hơn vào goal trước rồi sau đó mới đẩy thùng còn lại.



Hình 4: Hình mô phỏng tự động chơi màn level 2 của thuật toán DFS

- Nguyên nhân chính là khi lần duyệt đầu tiên, những hành động hợp lệ của nhân vật được trả về là (U,r), do DFS hoạt động dựa trên cấu trúc stack nên hành động ‘r’ được thực hiện đầu tiên (như Hình 5).



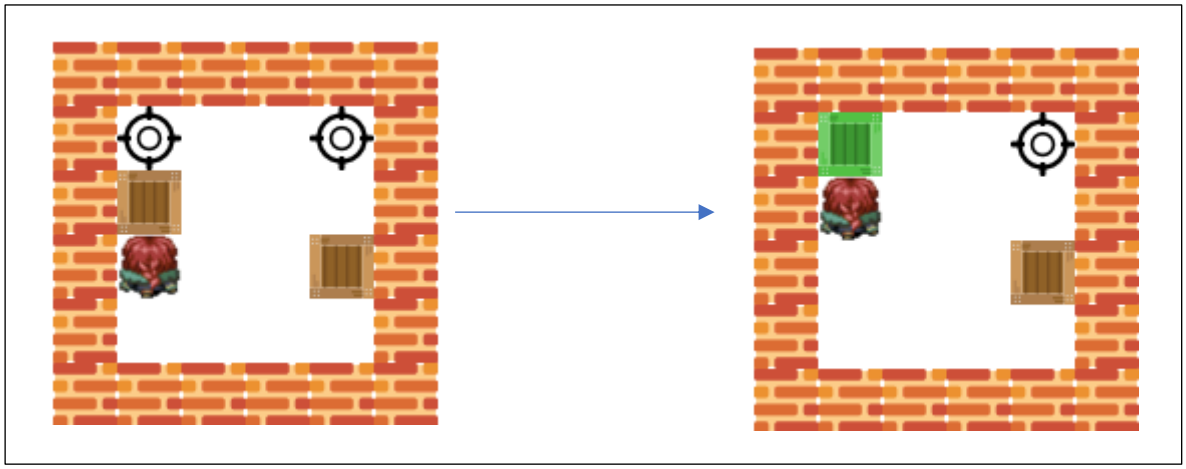
Hình 5: Hình mô phỏng legalActions trong lần duyệt đầu tiên và các bước đi của nhân vật trong màn level 1

2. Breadth First Search (BFS)

- Cũng tương tự như DFS, thuật toán Breadth First Search (BFS – Tìm kiếm theo chiều rộng) cũng là một hàm tiến triển theo chiến lược tìm kiếm mù.
- Khác với DFS, BFS hoạt động dựa trên cấu trúc queue (FIFO). Khởi đầu với nút gốc, BFS sẽ lần lượt duyệt các đỉnh kề xung quanh đỉnh gốc vừa xét, thuật toán sẽ dừng lại khi tìm thấy đỉnh cần tìm.

```
def breadthFirstSearch(gameState):
    """Implement breadthFirstSearch approach"""
    beginBox = PosOfBoxes(gameState) #lấy vị trí của những cái box
    beginPlayer = PosOfPlayer(gameState) #lấy vị trí của nhân vật

    startState = (beginPlayer, beginBox) # e.g. ((2, 2), ((2, 3), (3, 4), (4, 4), (6, 1), (6, 4), (6, 5)))
    frontier = collections.deque([[startState]]) # store states
    actions = collections.deque([[0]]) # store actions
    exploredSet = set()
    temp = []
    ### Implement breadthFirstSearch here

    while frontier:
        node = frontier.popleft() #lấy vị trí hiện tại của nhân vật và box
        node_action = actions.popleft() #lấy hành động hiện tại của nhân vật
        if isEndState(node[-1][-1]): #kiểm tra các box đã vô goal chưa, nếu vô rồi thì kết thúc game, nếu chưa vô thì tiếp tục game
            temp += node_action[1:]
            break
        if node[-1] not in exploredSet: #kiểm tra trạng thái của nhân vật đã được duyệt hay chưa
            exploredSet.add(node[-1]) #nếu chưa duyệt thì thêm vô exploreSet
            for action in legalActions(node[-1][0], node[-1][1]): #lấy ra từng trạng thái hợp lệ để duyệt
                newPosPlayer, newPosBox = updateState(node[-1][0], node[-1][1], action) #cập nhập lại vị trí mới của nhật vật và của thùng theo vị trí hợp l
                if isFailed(newPosBox): #kiểm tra vị trí mới của thùng có hợp lệ hay không
                    continue #nếu không hợp lệ thì bỏ qua vị trí mới của thùng
                frontier.append(node + [(newPosPlayer, newPosBox)]) #nếu vị trí mới của thùng hợp lệ thì thêm vị trí mới của nhân vật và thùng vào frontier
                actions.append(node_action + [action[-1]]) #nếu vị trí mới của thùng hợp lệ thì thêm hành động tiếp theo của nhân vật vào actions
    return temp
```

- BFS hoạt động dựa trên cấu trúc queue, nên khi bắt đầu duyệt BFS sẽ lấy ra state đầu tiên của frontier để duyệt.
- Khác với DFS, BFS tốn rất ít bước đi của nhân vật để có thể giải quyết được bài toán.



Hình 6: Hình mô phỏng tự động chơi màn level 2 của thuật toán BFS

- Vì BFS hoạt động theo cấu trúc queue, nên sau khi lần duyệt đầu tiên hành động ‘U’ được thực hiện đầu tiên.

3. Uniform Cost Search (UCS)

- Thuật toán Uniform Cost Search (UCS) cũng là một hàm tiến triển theo chiến lược tìm kiếm mù.
- UCS hoạt động dựa trên cấu trúc priority queue. Khởi đầu với nút gốc, UCS sẽ duyệt lần lượt duyệt các đỉnh có chi phí thấp nhất, thuật toán sẽ dừng lại khi tìm thấy đỉnh cần tìm.

```
def uniformCostSearch(gameState):
    """Implement uniformCostSearch approach"""
    beginBox = PosOfBoxes(gameState) #Lấy vị trí của những cái thùng
    beginPlayer = PosOfPlayer(gameState) #Lấy vị trí của nhân vật

    startState = (beginPlayer, beginBox)
    frontier = PriorityQueue()
    frontier.push([startState], 0) #Lưu trữ vị trí của nhân vật
    exploredSet = set()
    actions = PriorityQueue()
    actions.push([0], 0) #Lưu trữ trạng thái của nhân vật
    temp = []
    ### Implement uniform cost search here

    while frontier:
        node = frontier.pop() #Lấy vị trí hiện tại của nhân vật và thùng
        node_action = actions.pop() #Lấy hành động hiện tại của nhân vật

        if isEndState(node[-1][-1]): #kiểm tra các box đã vô goal chưa, nếu vô rồi thì kết thúc game, nếu chưa vô thì tiếp
            temp += node_action[1:]
            break
        if node[-1] not in exploredSet: #kiểm tra trạng thái của nhân vật đã được duyệt hay chưa
            exploredSet.add(node[-1]) #nếu chưa duyệt thì thêm vô exploreSet
            for action in legalActions(node[-1][0], node[-1][1]): #Lấy ra từng trạng thái hợp lệ để duyệt
                newPosPlayer, newPosBox = updateState(node[-1][0], node[-1][1], action) #cập nhập lại vị trí mới của nhật v
                if isFailed(newPosBox): #kiểm tra vị trí mới của thùng có hợp lệ hay không
                    continue #nếu không hợp lệ thì bỏ qua vị trí mới của thùng
                new_action = node_action + [action[-1]]
                frontier.push(node + [(newPosPlayer, newPosBox)],cost(new_action[1:])) #nếu vị trí mới của thùng hợp lệ thì
                actions.push(node_action + [action[-1]],cost(new_action[1:])) #nếu vị trí mới của thùng hợp lệ thì thêm hàn
    return temp
```

- Chi phí của 1 trạng thái trong trò chơi sokoban được tính bằng số bước đi của nhân vật từ trạng thái ban đầu đến trạng thái tiếp theo.
- UCS hoạt động dựa trên cấu trúc priority queue, nên khi bắt đầu duyệt UCS sẽ lấy trạng thái có chi phí thấp nhất để duyệt.
- Cũng tương tự như BFS, UCS tốn rất ít bước đi để giải quyết bài toán.

IV. So sánh và nhận xét

- Khi xét ở màn level 2:
    + DFS tốn 24 actions mới có thể giải quyết được bài toán.
    + BFS và UCS chỉ tốn 9 actions đã có thể giải quyết được bài toán.

⇨ Qua đó chúng ta thấy được cả 2 thuật toán UCS và BFS đều cho ra kết quả tối ưu hơn là DFS. Vì vậy trong lần so sánh tiếp theo chúng ta sẽ không đề cập tới DFS

- Khi xét ở màn level 5: cả BFS và UCS đều tốn mất 20 actions mới có thể giải quyết được bài toán, tuy nhiên BFS phải tốn thời gian gấp 3 lần UCS mới có thể tìm được actions.
- Nguyên nhân chủ yếu là do mỗi chi phí của tất cả trạng thái trong BFS là như nhau, nên khi không gian trạng thái quá lớn BFS sẽ tốn nhiều thời gian hơn trong việc xét các trạng thái; còn khi xét các trạng thái UCS chỉ xét những trạng thái có chi phí thấp nên sẽ tốn ít thời gian hơn so với UCS.

⇨ **Kết luận:** trong 3 thuật toán tìm kiếm, UCS là thuật toán tối ưu nhất.