Tên: Châu Tấn

MSSV: 20520926

Lớp: CS106.M21 (Môn: Trí tuệ nhân tạo)

GVHD: TS. Lương Ngọc Hoàng

BÁO CÁO BÀI TẬP 1:

CÀI ĐẶT THUẬT TOÁN CHƠI SOKOBAN

Yêu cầu: Trình bày lại việc áp dụng các thuật toán tìm kiếm DFS, BFS, UCS cho Sokoban như thế nào. Sokoban đã được mô hình hóa ra sao? Trạng thái khởi đầu, trạng thái kết thúc, không gian trạng thái, các hành động hợp lệ, hàm tiến triển (successor function) là gì?

Thống kê về độ dài đường đi tìm được bởi 3 thuật toán DFS, BFS, UCS tại tất cả các bản đồ có sẵn. Các em có nhận xét thế nào về lời giải tìm ra bởi mỗi thuật toán? Thuật toán nào là tốt hơn cả? Trong các bản đồ thì bản đồ nào khó giải nhất, tại sao?

LỜI MỞ ĐẦU

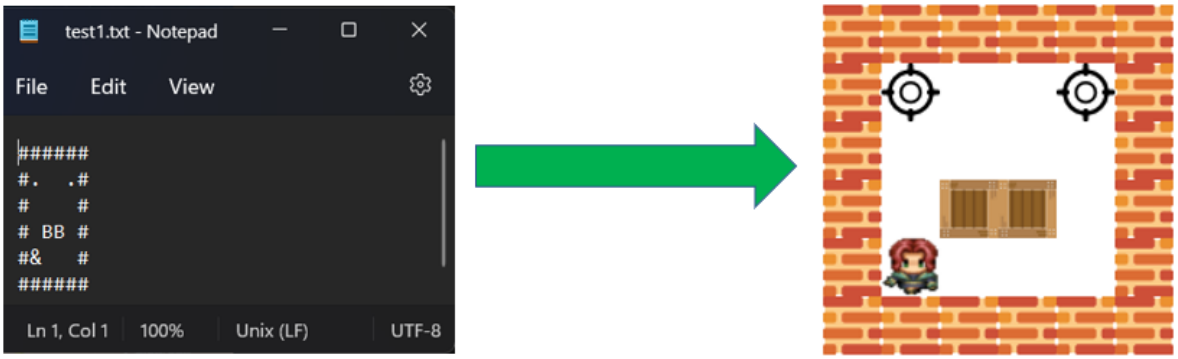
Sokoban là một trò chơi thuộc thể loại giải đố. Nhiệm vụ của người chơi là phải đẩy hết các thùng về vị trí đích.

Bài báo cáo này sẽ trình bày lại việc áp dụng 3 thuật toán tìm kiếm không có thông tin (Uninformed Search Algorithms) là DFS, BFS và UCS trên trò chơi Sokoban. Qua đó, chúng ta có thể thể rút ra được một vài nhận xét về 3 thuật toán trên. Ngoài ra, bài báo cáo còn trả lời được các câu hỏi như: *"Sokoban đã được mô hình hóa ra sao?"* và *"Trạng thái khởi đầu, trạng thái kết thúc, không gian trạng thái, các hành động hợp lệ, hàm tiến triển (successor function) là gì?"*

I. MÔ HÌNH HÓA GAME SOKOBAN VÀ TỔNG QUAN VỀ CÁC NGUYÊN TẮC CỦA TRÒ CHƠI.

1. Mô hình hóa của Sokoban

- Các màn chơi của sokoban ban đầu sẽ được lưu trong file txt với các ký tự "#" → Tường chắn, "&" → Nhân vật, "B" → Thùng, "." → Đích.



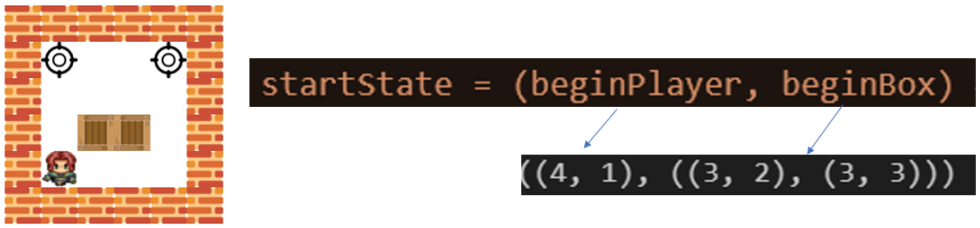
(Hình 1: demo level 1 của trò chơi Sokoban)

- Trong các thuật toán được áp dụng, Sokoban được mô hình hóa là một cây gồm các node. Một node bao gồm vị trí của người chơi và vị trí của các thùng trên bản

đồ. Mỗi node có thể mở ra các node con từ các hành động hợp lệ mà chúng có thể sinh ra.

- Frontier được tạo ra để lưu trữ các node và actions được tạo ra để lưu trữ các hành động của các node đang nằm trong frontier.
- Ứng với mỗi thuật toán thì cấu trúc dữ liệu của frontier và actions có thể khác nhau.
- Thuật toán sẽ kết thúc khi một node được lấy ra khỏi frontier và được xác định là trạng thái kết thúc hoặc là frontier rỗng (tức là không tìm được đường đi tới đích)

2. **Trạng thái khởi đầu**
   Trạng thái khởi đầu là vị trí ban đầu của nhân vật và các thùng.



(Hình 2: demo tại level 1 của game)

3. **Trạng thái kết thúc**
   Trạng thái kết thúc là trạng thái mà các thùng đã được đặt vào đúng vị trí của game.

(Hình 3: demo tại level 1 của game)

4. **Các hành động hợp lệ**

   Các hành động hợp lệ là các hành động không lặp lại một trạng thái đã được ghi nhận và vị trí player không trùng vị trí box, vị trí wall hoặc vị trí box không được trùng vị trí box khác, vị trí wall.

5. **Không gian trạng thái**
   Không gian trạng thái là không gian chứa những vị trí của hộp và những vi trí của người chơi có thể đi trên bản đồ.
6. **Hàm tiến triển**
   Hàm tiến triển là hàm để thực thi các bước đi tiếp theo để giải quyết bài toán.

II. CÁC HÀM TIẾN TRIỂN

Cả 3 hàm tiến triển trong bài báo cáo này đề cập đến đều thuộc loại thuật toán tìm kiếm không có thông tin (Uniform Search) hay còn được biết đến là thuật toán tìm kiếm mù. Ba hàm tiến triển được đề cập đến trong bài này sẽ là DFS, BFS và UCS.

1. **DEPTH-FIRST SEARCH (DFS)**

- Depth-first Search (Tìm kiếm theo chiều sâu) là một thuật toán tìm kiếm không có thông tin. DFS hoạt động dựa theo cấu trúc dữ liệu ngăn xếp (Stack). Tức là cấu trúc dữ liệu của frontier và actions là stack.
- Stack tuân theo phương pháp LIFO (Last in First Out) tức là phần tử đầu tiên vào cũng là phần tử đầu tiên được xử lý. Do đó, DFS sẽ duyệt xa nhất giữa các nhánh, khi nhánh đã duyệt hết thì lùi đến nhánh sâu gần nhất để tiếp tục duyệt đến khi tìm được lời giải.
- Đối với thuật toán DFS thì các node được mở rộng theo chiều sâu, liên tục lấy các node đầu tiên trong frontier (node có độ sâu lớn nhất) để mở rộng.
- Điểm yếu của thuật toán này là sẽ phải tốn rất nhiều bước để thực hiện thuật toán. Điều này dễ nhận thấy khi ta chạy Sokoban bằng phương pháp dfs khi nó chạy rất nhiều bước đi thừa.

```
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
```

2. BREADTH-FIRST SEARCH (BFS)

- Breadth-first Search (Tìm kiếm theo chiều rộng) là một thuật toán tìm kiếm không có thông tin. BFS hoạt động dựa theo cấu trúc dữ liệu hàng đợi (Queue). Tức là cấu trúc dữ liệu của frontier và actions là queue.
- Queue tuân theo phương pháp FIFO (First in First Out) tức là phần tử đầu tiên vào cũng là phần tử đầu tiên được xử lý. Do đó, BFS sẽ duyệt rộng nhất

giữa các nhánh, khi đã duyệt xong thì sẽ tìm nhánh nông nhất để duyệt cho đến khi tìm ra lời giải cho bài toán.

- Đối với thuật toán BFS thì các node được mở rộng theo chiều rộng, liên tục lấy các node cuối cùng trong frontier (node có độ sâu bé nhất) để mở rộng.
- Điểm yếu của thuật toán này là sẽ phải tốn khá nhiều bộ nhớ để lưu trữ hàng đợi. Tuy nhiên, BFS có thể giải bài toán với số bước đi khá tối ưu.

```
def breadthFirstSearch(gameState):
    """Implement breadthFirstSearch approach"""
    beginBox = PosOfBoxes(gameState)
    #Initial the coordinates of the box on the screen
    beginPlayer = PosOfPlayer(gameState)
    #Initial the coordinates of the player on the screen
    # The data structure of frontier and actions is double queue (The queue which we can
pop or append from both the left side and the right side)
    startState = (beginPlayer, beginBox)
    # e.g. ((2, 2), ((2, 3), (3, 4), (4, 4), (6, 1), (6, 4), (6, 5)))
    frontier = collections.deque([[startState]]) # A queue which stores states
    actions = collections.deque([[0]]) # A queue which stores actions
    exploredSet = set() # Close set
    temp = [] # The list that store the path from begining to solution
    ### Implement breadthFirstSearch here
    while frontier: # Iterate through the queue that stores the states
        node = frontier.popleft() # Pop the leftside node from the frontier
        node_action = actions.popleft() # Pop the leftside node from the actions
        if isEndState(node[-1][-1]): # Check if node is the end state or not
            temp += node_action[1:]
            # if the node is end state, temp will save the path from beginning to solution
            print("Cost of bfs", states(temp)) # print cost of the way to solution
            break # Exit the frontier
        if node[-1] not in exploredSet: #check if the node is explored or not
            exploredSet.add(node[-1]) # if not, add it to explored set
            for action in legalActions(node[-1][0], node[-1][1]):
            # Loop through the set of legal action
                newPosPlayer, newPosBox = updateState(node[-1][0], node[-1][-1], action)
                #Update the new Player and Box Position corresponding with the action
                if isFailed(newPosBox):
                #check if the new position of box is pottentially failed or not
                    continue # if yes, skip this loop and go to another loop
                frontier.append(node + [(newPosPlayer, newPosBox)])
                # add new position to the frontier
                actions.append(node_action + [action[-1]]) # add new action to the actions
    return temp # return the path to solution
```

3. UNIFORMED-COST SEARCH (UCS)

- Uninform-Cost Search (Tìm kiếm tối ưu chi phí) là một thuật toán tìm kiếm không có thông tin. UCS hoạt động dựa theo cấu trúc dữ liệu hàng đợi ưu tiên (Priority queue). Tức là cấu trúc dữ liệu của frontier và actions cũng đều là priority queue.

- Priority queue là thuật toán xếp hàng đợi theo độ ưu tiên. Áp dụng priority queue trong trường hợp này thì chúng ta có thể hiểu rằng đây là một hàng đợi nhưng sẽ ưu tiên những node có chi phí thấp nhất để giải ra bài toán.
- Đối với thuật toán UCS thì các node được mở rộng theo độ ưu tiên, cụ thể hơn UCS sẽ chọn các node mà có chi phí thấp nhất để mở rộng.
- Giống với BFS, UCS có thể giải bài toán với số bước đi khá tối ưu.

```
def cost(actions):
    return len([x for x in actions if x.islower()])
def uniformCostSearch(gameState):
    """Implement uniformCostSearch approach"""
    beginBox = PosOfBoxes(gameState) #Initial the coordinates of the box on the screen
    beginPlayer = PosOfPlayer(gameState)
    #Initial the coordinates of the player on the screen
    # The data structure of frontier and actions is priority queue.
    startState = (beginPlayer, beginBox) #Initial the startState
    frontier = PriorityQueue() #Initial frontier as Priority queue
    frontier.push([startState], 0) #Push the startState to Frontier with Priority = 0
    exploredSet = set() # Close Set
    actions = PriorityQueue() # Initial actions as Priority queue
    actions.push([0], 0) #Push the first item to actions with Priority = 0
    temp = [] # A set that store the path to solution
    ### Implement uniform cost search here
    while frontier: #Iterating through the frontier
        node = frontier.pop() # Pop the node from the frontier
        node_action = actions.pop() # Pop the actions from the frontier
        if isEndState(node[-1][-1]): # Check if node is the end state or not
            temp += node_action[1:]
            # if the node is end state, temp will save the path from beginning to solution
            print("Cost of ucs", cost(temp)) # print cost of the way to solution
            break # Exit the frontier
        if node[-1] not in exploredSet: #check if the node is explored or not
            exploredSet.add(node[-1]) # if not, add it to explored set
            Cost_Action = cost(node_action[1:]) # calculate the cost of node_action
            for action in legalActions(node[-1][0], node[-1][1]):
            # Loop through the set of legal action
                newPosPlayer, newPosBox = updateState(node[-1][0], node[-1][1], action)
                #Update the new Player and Box Position corresponding with the action
                if isFailed(newPosBox):
                #check if the new position of box is pottentially failed or not
                    continue # if yes, skip this loop and go to another loop
                frontier.push(node + [(newPosPlayer, newPosBox)],Cost_Action)
                # push the new node to the frontier with the priority of cost action
                actions.push(node_action + [action[-1]],Cost_Action)
                # push the new node to the actions with the priority of cost action
    return temp # return the path to solution
```

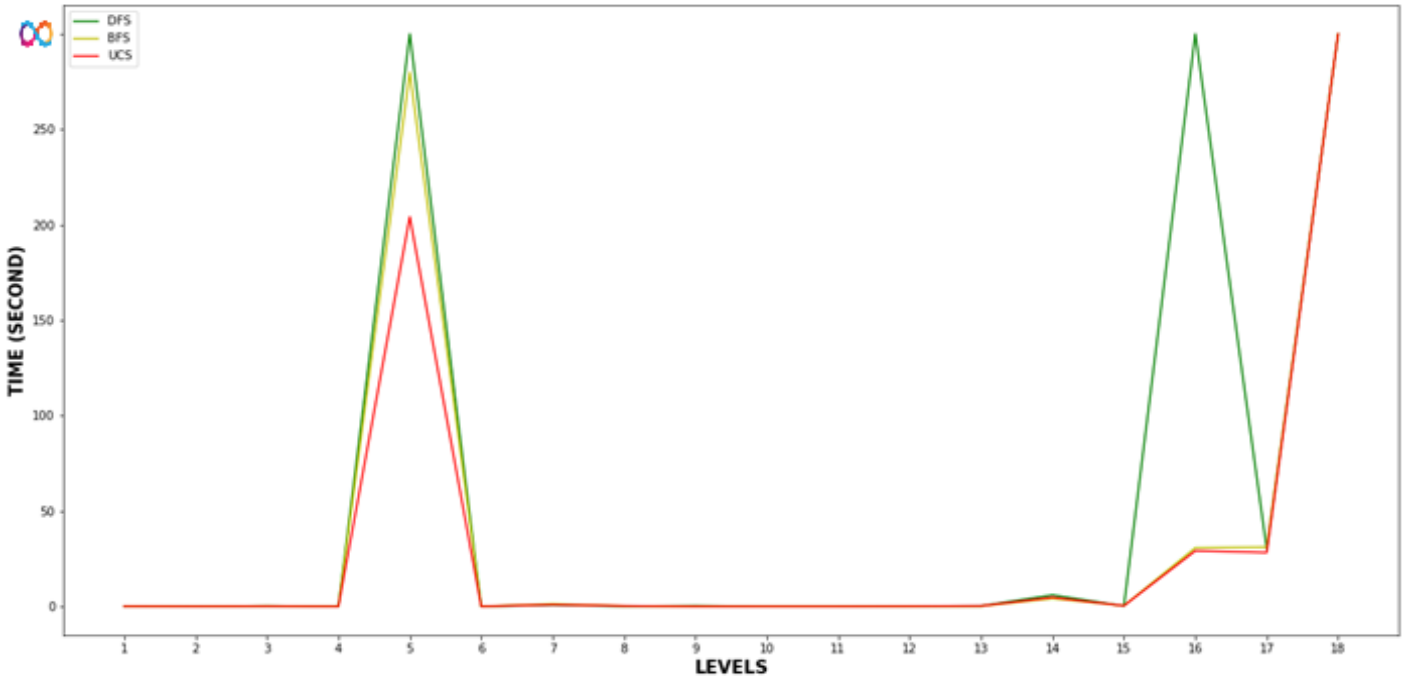
III. THỐNG KÊ KẾT QUẢ VÀ RÚT RA KẾT LUẬN

Sau đây là một vài kết luận của em sau khi đã chạy cả ba thuật toán DFS, BFS, UCS trong game Sokoban.

Sau đây là bảng thống kê về thời gian và chi phí của các thuật toán trong game Sokoban. (Đối với những trường hợp infinity là những trường hợp đã để máy chạy thuật toán hơn 20 phút mà chưa ra kết quả)

LEVELS	Time *(seconds)*			Cost		
	BFS	UCS	DFS	BFS	UCS	DFS
1	0.16	0.1	0.09	12	6	79
2	0.01	0.04	0.01	9	5	24
3	0.28	0.13	0.3	15	9	403
4	0.01	0.01	0.01	7	4	27
5	279.89	204.15	infinity	20	10	0
6	0.04	0.02	0.02	19	14	55
7	1.31	0.94	0.79	21	10	707
8	0.27	0.27	0.1	97	65	323
9	0.01	0.02	0.37	8	5	74
10	0.02	0.02	0.01	33	25	37
11	0.02	0.02	0.02	34	27	36
12	0.11	0.12	0.16	23	16	109
13	0.18	0.29	0.28	31	24	185
14	4.14	4.76	6	23	16	865
15	0.4	0.36	0.24	105	62	291
16	30.73	29.15	infinity	34	22	0
17	31.12	28.23	29.99	0	0	0
18	infinity	infinity	infinity	0	0	0

Để nhìn rõ hơn về sự khác biệt của thời gian chạy và chi phí của 3 thuật toán thì việc visualization ra dạng biểu đồ là cần thiết. Sau đây là biểu đồ thể hiện thời gian chạy của 3 thuật toán dưới dạng line-chart.

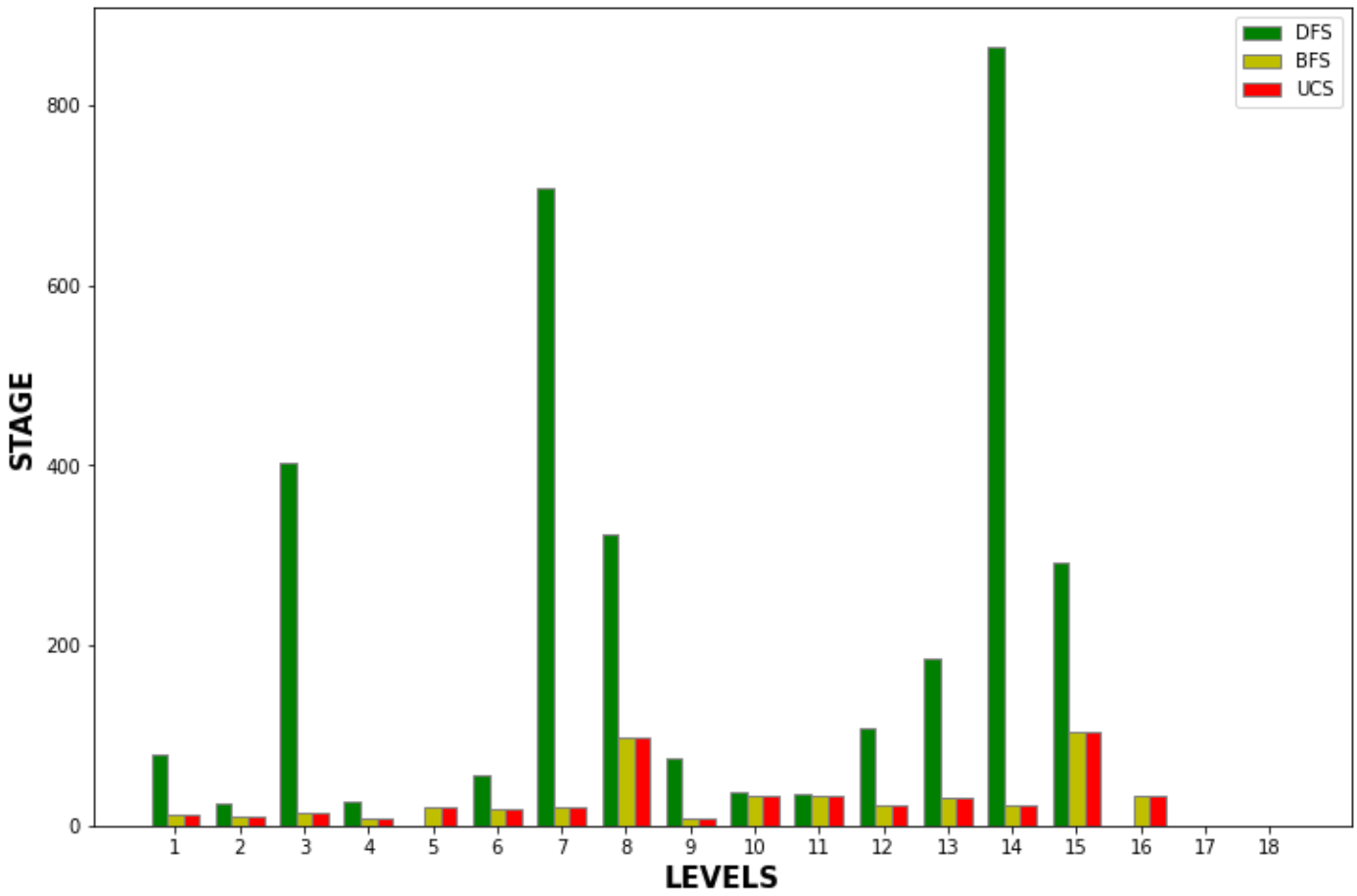


(Hình 4.1: Biểu đồ được plot ra từ số liệu Time)

Sau khi nhìn bảng trên và biểu đồ trên thì em rút ra một số kết luận về thời gian chạy.

- ✓ Trong hầu hết mọi màn chơi thì UCS là thuật toán có thời gian chạy tối ưu nhất và DFS là có thời gian chạy lâu nhất. Trong một số màn thì BFS cũng có thể vượt qua được UCS. Tuy nhiên khi xét về tổng thể thì UCS vẫn tối ưu nhất
- ✓ Màn 5 là màn có thời gian chạy lâu nhất để có thể đưa ra kết quả đối với cả 3 thuật toán (thậm chí là vô cùng đối với DFS). Lý do là do cái 3 thùng trồng lên nhau và không có một bức tường chướng ngại vật nào (ngoại trừ 4 bức tường bao quanh màn chơi) do đó số legal move sẽ được sinh ra nhiều hơn từ đó dẫn đến số đường đi được sinh ra nhiều hơn do đó nó sẽ tốn khá nhiều thời gian để chạy ra được.
- ✓ Màn 16 thì UCS và BFS giải ra được trong lần lượt 29.15 và 30.73 giây trong khi DFS phải tốn rất nhiều thời gian vẫn chưa giải ra được.
- ✓ Màn 18 là màn "kinh khủng" nhất khi có tới 10 hộp cần phải đẩy vào 10 vị trí đích. Độ phức tạp của màn chơi là khá lớn (màn này em để 45 phút với mỗi thuật toán) nhưng mà vẫn không thể chạy ra được kết quả. Cả 3 thuật toán đều không thể giải ra màn này (mặc dù em có thể tự giải ra màn này sau 3 phút). Cho nên đối với 3 thuật toán tìm kiếm được áp dụng thì đây là màn khó nhất. Theo ý kiến của bản thân, các thuật toán tìm kiếm có thông tin sẽ giải được màn này nhanh chóng.

Sau đây là biểu đồ thể hiện số bước đi của cả 3 thuật toán

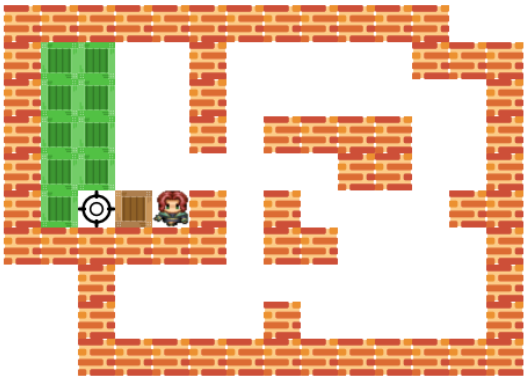


(Hình 4.2: Biểu đồ được plot ra từ số liệu stage)

- ✓ Có thể thấy trong 3 biểu đồ DFS luôn có số bước đi vượt trội hoàn toàn so với hai thuật toán còn lại trong khi đó BFS và UCS có số bước đi bằng nhau (Do mỗi bước đi có chi

phí bằng 1). Tuy nhiên, khi tính cost thì UCS bỏ qua các bước đẩy thùng do đó bảng thống kê trên thì chi phí của UCS ít hơn so với chi phí của BFS.

- ✓ Ở level 16, do DFS có thời gian chạy là vô cùng cho nên không thể xác định được số bước đi.
- ✓ Ở level 17, do màn này không có lời giải cho nên số đường đi của cả 3 thuật toán đều trả về 0.
- ✓ Ở level 18, do cả 3 thuật toán đều "bó tay" trước độ phức tạp (chứ không phải màn này không có lời giải) của màn này nên cũng không thể xác định được số bước đi của màn này.



(Hình 5: "AI chạy bằng cơm" sắp giải ra màn 18)

Qua những điều rút ra của 2 khía cạnh ở trên thì ta hoàn toàn dễ dàng nhận thấy được:

- ➢ Màn chơi khó nhất chính là **level 18**. Cả 3 thuật toán đều phải "bó tay" trước độ phức tạp "khủng khiếp" của nó.
- ➢ Thuật toán tối ưu nhất trong 3 thuật toán là **UCS**.

KẾT THÚC