Assignment 1 - DFS/BFS/UCS for Sokoban

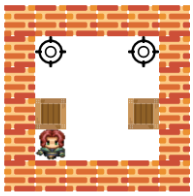
CS106.N21.KHCL

Lê Gia Khang – 21522189

I. Giới thiệu

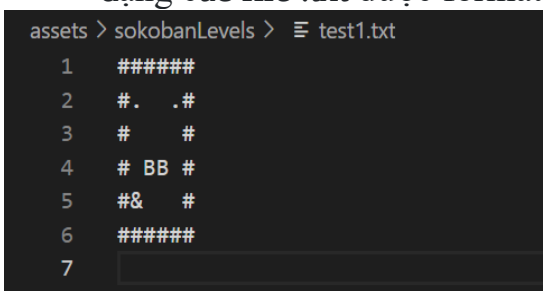
- Theo wikipedia, Sokoban là trò chơi dạng câu đố trong đó người chơi phải đẩy một số khối vuông vượt qua chướng ngại vật để đến đích. Trò chơi đã được thiết kế vào năm 1981 bởi Hiroyuki Imabayashi và được ra mắt lần đầu vào tháng 12 năm 1982.
- Trong assigment này, chúng ta sẽ dùng những thuật toán Uninformed Search như DFS, BFS và UCS để chơi Sokoban với tổng thảy 18 level.

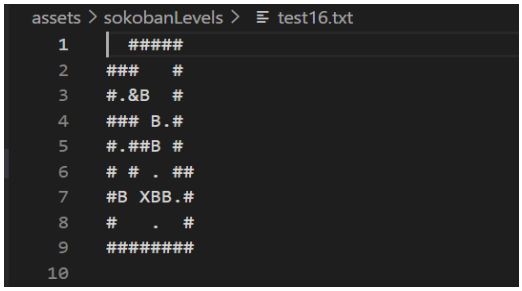
II. Mô hình hóa Sokoban



1. Biểu diễn bài toán

- Khi chạy chương trình, game sẽ được hiển thị dưới dạng đồ họa 2D.
- Nhưng khi đưa input vào, những level sẽ được lưu dưới dạng các file .txt được format như sau:





- Sau đó chúng ta convert những kí tự đó sang một ma trận 2 chiều với các số nguyên tương ứng.
- Với các kí hiệu:
  - "#" là bức tường được biểu diễn là 1.
  - "B" là các hộp cần được đẩy được biểu diễn là 3.
  - "X" là các hộp đã nằm trên goal được biểu diễn là 5.
  - "." chính là các goal mà ta cần phải đẩy hộp vào được biểu diễn là 4.
  - "&" chính là player được biểu diễn là 2.
  - " " chính là các không gian trống được biểu diễn là 0

```
for irow in range(len(layout)):
    for icol in range(len(layout[irow])):
        if layout[irow][icol] == ' ': layout[irow][icol] = 0   # free space
        elif layout[irow][icol] == '#': layout[irow][icol] = 1 # wall
        elif layout[irow][icol] == '&': layout[irow][icol] = 2 # player
        elif layout[irow][icol] == 'B': layout[irow][icol] = 3 # box
        elif layout[irow][icol] == '.': layout[irow][icol] = 4 # goal
        elif layout[irow][icol] == 'X': layout[irow][icol] = 5 # box on goal
    colsNum = len(layout[irow])
```

2. Bài toán tìm kiếm

- Một bài toán tìm kiếm (Search Problems) bao gồm các yếu tố như state space, successor function, goal state, etc..
- Một trạng thái bao gồm thông tin về vị trí hiện tại của player và những chiếc hộp.
- Initial State: vị trí khởi đầu của player và những chiếc hộp được cho sẵn trong file .txt.
- Actions: player có thể di chuyển 4 hướng (Up, Down, Left, Right). Bên cạnh đó, player có thể đẩy hộp (chỉ 1 ô). Ta sinh ra các hành động bằng hàm LegalActions() dưới đây. Kết quả trả về là 1 tuple với 3 thành phần: 2 thành phần đầu là tọa độ x,y của player và một kí tự để xem bước đi đó có phải là đẩy hộp không.

```
def legalActions(posPlayer, posBox):
    """Return all legal actions for the agent in the current game state"""
    allActions = [[-1,0,'u','U'],[1,0,'d','D'],[0,-1,'l','L'],[0,1,'r','R']]
    xPlayer, yPlayer = posPlayer
    legalActions = []
    for action in allActions:
        x1, y1 = xPlayer + action[0], yPlayer + action[1]
        if (x1, y1) in posBox: # the move was a push
            action.pop(2) # drop the little letter
        else:
            action.pop(3) # drop the upper letter
        if isLegalAction(action, posPlayer, posBox):
            legalActions.append(action)
        else:
            continue
    return tuple(tuple(x) for x in legalActions) # e.g. ((0, -1, 'l'), (0, 1, 'R'))
```

- Trong các hành động có những cái hợp lệ và những cái không – như di chuyển xuyên tường, đẩy nhiều hộp chồng chất kế tiếp nhau,… Ta có thể kiểm tra tính hợp lệ của hành động thông qua hàm isLegalAction().

```
def legalActions(posPlayer, posBox):
    """Return all legal actions for the agent in the current game state"""
    allActions = [[-1,0,'u','U'],[1,0,'d','D'],[0,-1,'l','L'],[0,1,'r','R']]
    xPlayer, yPlayer = posPlayer
    legalActions = []
    for action in allActions:
        x1, y1 = xPlayer + action[0], yPlayer + action[1]
        if (x1, y1) in posBox: # the move was a push
            action.pop(2) # drop the little letter
        else:
            action.pop(3) # drop the upper letter
        if isLegalAction(action, posPlayer, posBox):
            legalActions.append(action)
        else:
            continue
    return tuple(tuple(x) for x in legalActions) # e.g. ((0, -1, 'l'), (0, 1, 'R'))
```

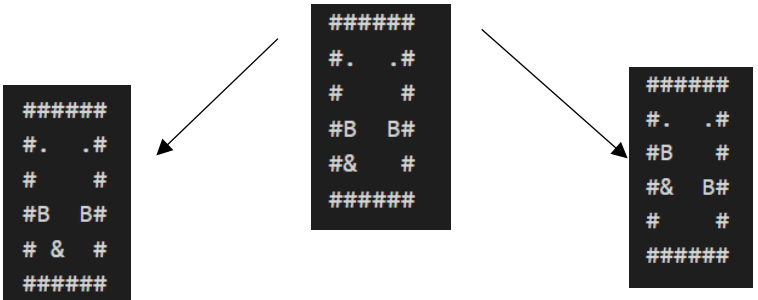
- Cost: được định nghĩa theo hàm Cost() như dưới đây. Cost của một hành động được tính là 1 nếu đó là di chuyển và 0 nếu đó là đẩy hộp.

```
def cost(actions):
    """A cost function"""
    return len([x for x in actions if x.islower()])
# UCS cost is the number of step moving from 1 state to other states, excluding steps where we push boxes
```

- Successor function (Hàm tiến triển): cập nhật lại vị trí của player và những chiếc hộp sau mỗi action thông qua hàm updateState().

```
def updateState(posPlayer, posBox, action):
    """Return updated game state after an action is taken"""
    xPlayer, yPlayer = posPlayer # the previous position of player
    newPosPlayer = [xPlayer + action[0], yPlayer + action[1]] # the current position of player
    posBox = [list(x) for x in posBox]
    if action[-1].isupper(): # if pushing, update the position of box
        posBox.remove(newPosPlayer)
        posBox.append([xPlayer + 2 * action[0], yPlayer + 2 * action[1]])
    posBox = tuple(tuple(x) for x in posBox)
    newPosPlayer = tuple(newPosPlayer)
    return newPosPlayer, posBox
```

- Từ một state hiện tại, ta có thể sinh ra tối đa 4 state tiếp theo – phụ thuộc vào những hành động hợp lệ có thể được tạo ra.



- Từ state cha ta có thể sinh ra các state con lần lượt và từ các state con sẽ sinh ra các state kế tiếp. Dần dần sẽ có cấu trúc như một cây, với các node là những state của bài toán và root note là initial state.
- Ta có thể sử dụng các thuật toán tìm kiếm trên cây như DFS/ BFS/ UCS để tìm đường đi từ state ban đầu tới goal state.

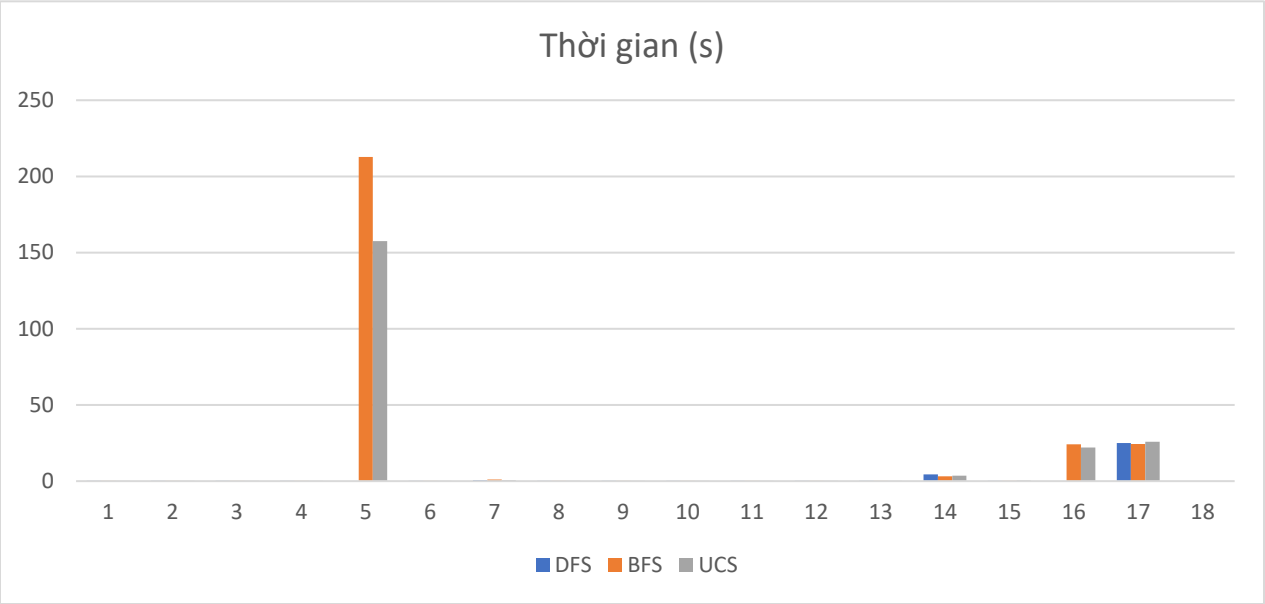
III. Các thuật toán tìm kiếm

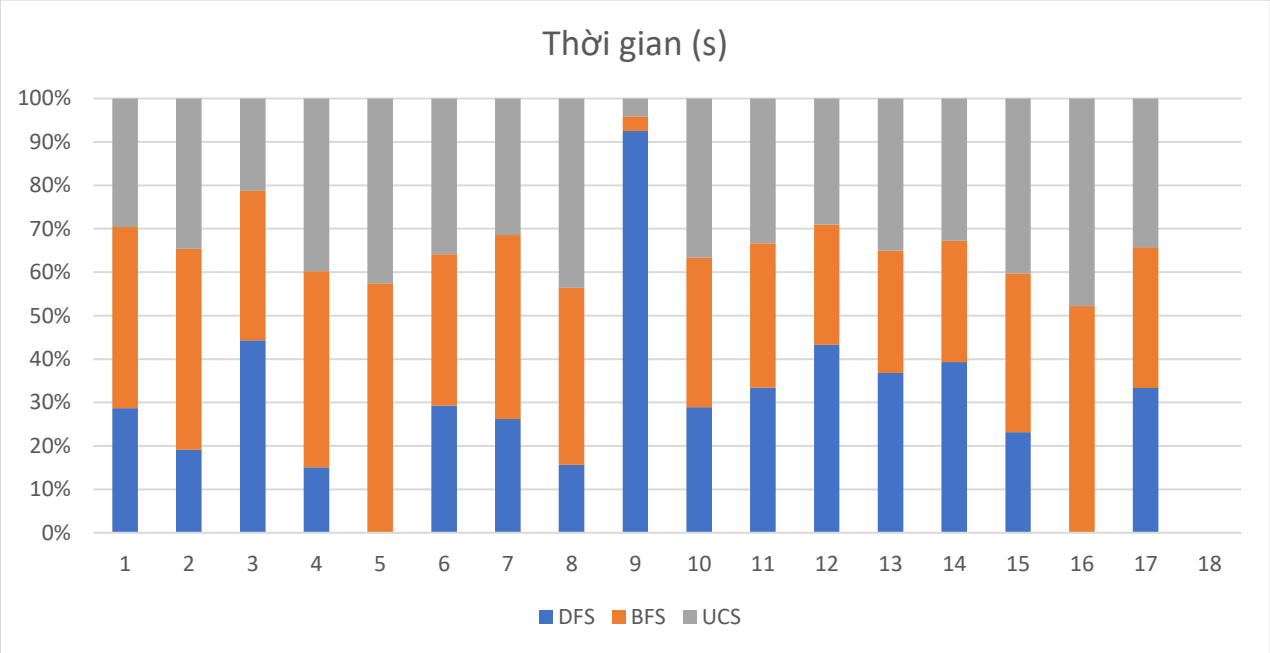
1. Thống kê độ hiệu quả

- Đây là bảng thống kê thời gian mỗi thuật toán chạy tương ứng với từng level, kèm với biểu đồ.

Level	DFS	BFS	UCS
1	0.069615602	0.101242065	0.071926355
2	0.003890753	0.009401321	0.007040501
3	0.253443003	0.197012186	0.121503115
4	0.002997398	0.008998394	0.007911205
5	*	212.7240245	157.5026531
6	0.013032913	0.015529156	0.015944004
7	0.594836712	0.964583397	0.714101315
8	0.08434701	0.219148636	0.234815121
9	0.301252604	0.010903835	0.013406754
10	0.015935659	0.018935204	0.020202875
11	0.020026445	0.019904375	0.019971609
12	0.158441305	0.101223946	0.106099367
13	0.217540026	0.1661551	0.207343102
14	4.335035086	3.079730511	3.61292696
15	0.188528299	0.298265696	0.328128815
16	*	24.1237483	22.08950424
17	25.10556	24.40730286	25.74931264
18	*	*	*

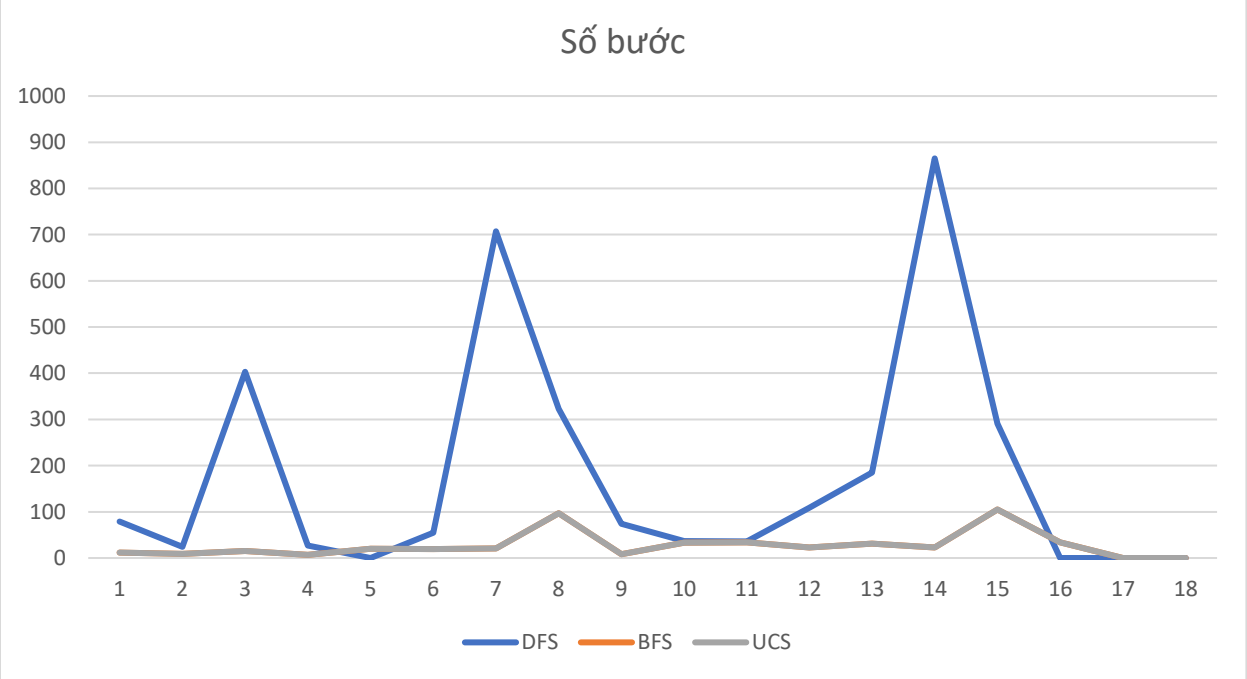
- Note: "*" là các trường hợp không thể chạy ra, vấn đề là bị tràn RAM như ở level 5 của DFS hoặc chờ đợi kết quả quá lâu như là level 18.

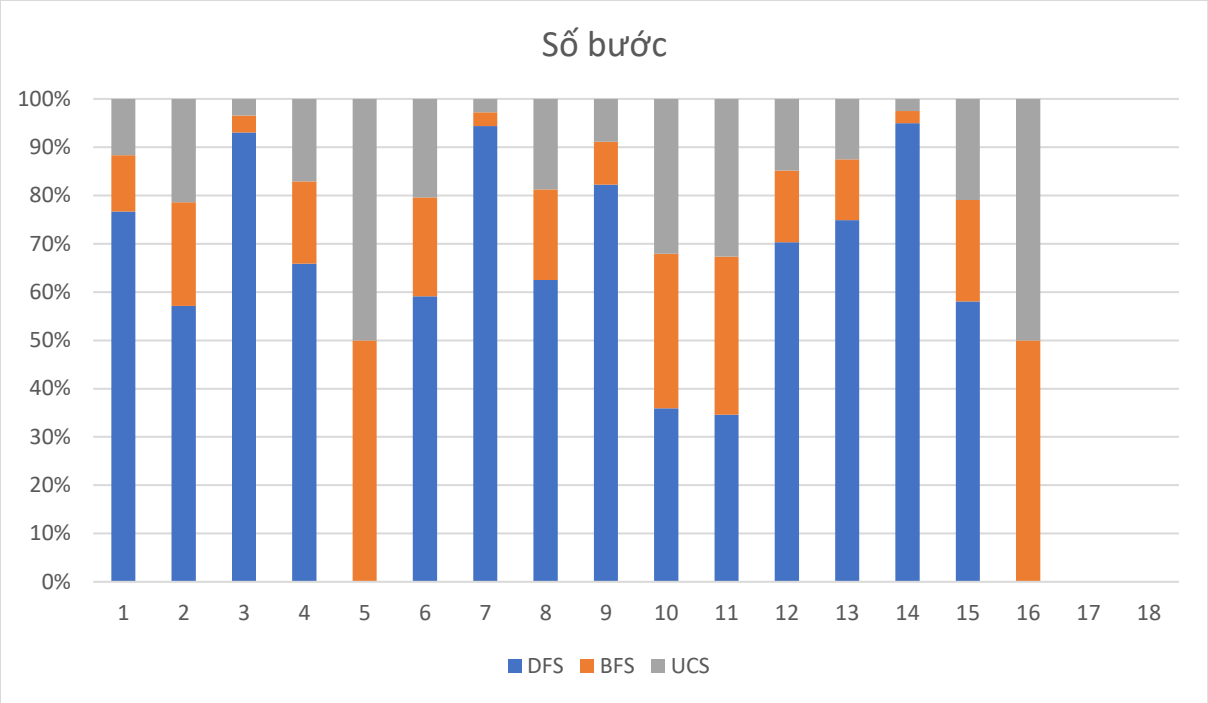




- Đây là bảng thống kê số bước chạy đối với mỗi thuật toán tương ứng với từng level, kèm theo biểu đồ.

Level	DFS	BFS	UCS
1	79	12	12
2	24	9	9
3	403	15	15
4	27	7	7
5	0	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	0	34	34
17	0	0	0
18	0	0	0





2. Nhận xét

- Về các thuật toán:
  - DFS: kết quả mà thuật toán giải ra không thật sự tối ưu và hoàn thiện. Đường đi từ state gốc tới goal state luôn dài hơn cần thiết (như ở level 3, DFS cho ra tận 403 bước đi trong khi so sánh với UCS và BFS là 15 bước). Khi so về thời gian chạy, ở một số trường hợp DFS cho ra kết quả nhanh hơn 2 thuật toán kia. Nhưng DFS có thể chạy lặp vĩnh viễn ở 1 nhánh mặc dù cho kết quả nằm ở nhánh kế bên, điều đó dẫn đến DFS có thể gây tràn bộ nhớ.
  - BFS: thuật toán trả về kết quả tương đồng với UCS (tối ưu). Số hành động của 2 thuật toán là như nhau, nhưng bởi vì time complexity của BFS là $O(b^d)$ nên giải thuật có thể xử lý chậm hơn.
  - UCS: tương tự như BFS ở số bước xử lý. Vì BFS sẽ tìm con đường có số bước phải đi ít nhất, trong khi UCS xem chi phí ở mỗi action tương ứng với 1 lần di chuyển và sẽ mở con đường có chi phí thấp nhất (nếu ta thay đổi hàm cost, phạt nặng hơn ở mỗi lần không phải đẩy hộp thì có thể kết quả sẽ khác). Chúng ta có thể thấy thời gian chạy của UCS tương đối thấp hơn BFS một chút, điều này có thể do cách ta cài đặt cấu trúc dữ liệu Priority Queue.

⇨ **Thuật toán hiệu quả nhất là UCS khi xét đến thời gian xử lý và số bước thực hiện.**

- Về các level đáng chú ý:
  - Level 5: khiến cho DFS gây tràn ram và thời gian xử lý của 2 thuật toán còn lại cũng lâu nhất trong khi thoạt nhìn qua thì mô hình bài toán của level 5 rất đơn giản. Bởi vì level 5 có rất nhiều legal actions, nên số node rẽ nhánh ra và được xét rất lớn. Khi DFS xử lý đã lẩn quẩn lặp vô tận trong 1 nhánh quá sâu khiến cho bộ nhớ bị tràn, trong khi BFS và UCS chỉ mở đến tầng thứ 20 là đã giải được.
  - Level 17: ở level này không hề có kết quả. Tuy nhiên cả 3 thuật toán đều chạy được.
  - Level 18: có lẽ đây là level khó nhất, mặc dù không thể hiện bất kì dấu hiệu gì về tràn bộ nhớ nhưng em đã chạy UCS trên level 18 hơn 30 phút vẫn không ra được kết quả.

IV. Tổng kết

- Với các số liệu như trên, ta có thể nói UCS là thuật toán Uniformed Search tối ưu nhất. Tuy nhiên vẫn có level mà UCS chưa giải được, bên cạnh đó thời gian xử lý cũng chưa hẳn là tối ưu.
- Có thể với các thuật toán tìm kiếm khác như A* sẽ tối ưu hơn cả về thời gian xử lý và số bước thực hiện.