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 2

CÀI ĐẶT THUẬT TOÁN A STAR SEARCH 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. CÀI ĐẶT HÀM HEURISTIC

Heuristic là một hàm ước lượng chi phí để đi từ trạng thái hiện tại đến trạng thái đích. Nó chính là **"thông tin"** trong các thuật toán tìm kiếm có thông tin như Greedy Search, A Star Search…

1. Heuristic là tổng khoảng cách Manhattan giữa các thùng và các vị trí đích (hàm sẵn có trong code mẫu)

Khoảng cách Manhattan giữa một thùng và một vị trí đích được xác định theo công thức:

$$d\left(box,\ goal\right) = \left|box.x - goal.x\right| + \left|box.y - goal.y\right|$$

```
def heuristic(posPlayer, posBox):
    # print(posPlayer, posBox)
    """A heuristic function to calculate the overall distance between the else boxes and the else goals"""
    distance = 0
    completes = set(posGoals) & set(posBox)
    sortposBox = list(set(posBox).difference(completes))
    sortposGoals = list(set(posGoals).difference(completes))
    for i in range(len(sortposBox)):
        distance += (abs(sortposBox[i][0] - sortposGoals[i][0])) + (abs(sortposBox[i][1] - sortposGoals[i][1]))
    return distance
```

Hàm heuristic này sẽ xác định các thùng đã nằm trên đích (set(posGoals) & set(posBox)) rồi loại bỏ các thùng này và vị trí đích tương ứng. Sau đó tính toán và trả về **tổng khoảng cách Manhattan giữa các thùng chưa được đặt vào đích và các vị trí đích chưa có thùng** (mỗi thùng được tính khoảng cách với một vị trí đích).

2. Heuristic là tổng khoảng cách Euclid giữa các thùng và các vị trí đích

Khoảng cách Euclid giữa một thùng và một vị trí đích được xác định theo công thức:

$$d\left(box,\ goal\right) = \sqrt{(box.x - goal.x)^2 + (box.y - goal.y)^2}$$

```
def heuristic(posPlayer, posBox):
    # print(posPlayer, posBox)
    """A heuristic function to calculate the overall distance between the else boxes and the else goals"""
    distance = 0
    completes = set(posGoals) & set(posBox)
    sortposBox = list(set(posBox).difference(completes))
    sortposGoals = list(set(posGoals).difference(completes))
    for i in range(len(sortposBox)):
        distance += ((sortposBox[i][0] - sortposGoals[i][0])**2 + (sortposBox[i][1] - sortposGoals[i][1])**2)**0.5
    return distance
```

Hàm heuristic này sẽ tính toán và trả về **tổng khoảng cách Euclid giữa các thùng chưa được đặt vào đích và các vị trí đích chưa có thùng** (mỗi thùng sẽ tính khoảng cách với một vị trí đích).

II. CÀI ĐẶT THUẬT TOÁN A STAR SEARCH

A Star Search (A*) là thuật toán tìm kiếm có thông tin (Informed Search).

Các thuật Uninformed Search (DFS, BFS, UCF) khám phá không gian tìm kiếm một cách có hệ thống nhưng mù quáng mà không xem xét chi phí để đạt được mục tiêu hoặc khả năng tìm ra giải pháp (*UCS có xét đến chi phí nhưng lại là chi phí từ trạng thái khởi đầu đến trạng thái hiện tại, không phải chi phí để đạt được trạng thái đích*). Ngược lại, các thuật toán Informed Search có **thông tin về mục tiêu** giúp tìm kiếm hiệu quả hơn. Thông tin này có được bằng một **hàm ước lượng mức độ "gần" trạng thái đích của một trạng thái – hàm heuristic**.

Tương tự như DFS, BFS và UCS, trong A Star, 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 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 các hàng đợi ưu tiên **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.

A Star có cách cài đặt như UCS, chỉ khác trong A Star sử dụng **độ ưu tiên** trong các hàng đợi **frontiers và actions** là **tổng chi phí đường đi của node và heuristic của trạng thái cuối cùng trong node đó**, thay vì chỉ là chi phí đường đi trong UCS.

Tại mỗi lần lặp, một node được lấy ra khỏi **frontier**, nếu trạng thái cuối cùng trong node đó là *trạng thái đích* 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.

```python
def aStarSearch(gameState):
    """Implement aStarSearch approach"""
    # start =  time.time()
    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)
    temp = [] # List sẽ chứa đường đi cần tìm từ start_state đến goalState
    start_state = (beginPlayer, beginBox)  # Tạo trạng thái khởi đầu gồm vị trí khởi đầu của người chơi và các thùng
    # Gọi f(n) là tổng giữa chi phí đường đi từ start_state đến trạng thái cuối cùng trong node n
                        # và chi phí ước lượng (bằng hàm heuristic) từ trạng thái cuối cùng này đến goalState

    frontier = PriorityQueue() # Khởi tạo hàng đợi ưu tiên sẽ chứa các NODE với độ ưu tiên là f của các node đó
    # Thêm node đầu tiên chỉ gồm start_state vào frontier với độ ưu tiên = heurictis(start_state) + 0
    frontier.push([start_state], heuristic(beginPlayer, beginBox))
    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
                        # Độ ưu tiên 1 đường đi của actions cũng chính là độ ưu tiên của node tương ứng với nó trong frontier
    actions.push([0], heuristic(beginPlayer, start_state[1])) # Thêm vào actions đường đi đầu tiên (chưa có hành động nào) ứng với node đầu tiên frontier
    while len(frontier.Heap) > 0: # Lặp qua hàng đợi ưu tiên frontier để mở rộng các node
        node = frontier.pop() # Lấy ra khỏi frontier node có độ ưu tiên cao nhất (f 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 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
```

```python
        ### CONTINUE YOUR CODE FROM HERE
        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

                # Cập nhật vị trí mới của người chơi và các thùng dựa trên TRẠNG THÁI HIỆN TẠI và action ta được TRẠNG THÁI MỚI
                newPosPlayer, newPosBox = updateState(node[-1][0], node[-1][1], action)
                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
                # Thêm action[-1] (e.g 'l') vào cuối đường đi node_action để được ĐƯỜNG ĐI MỚI từ start_state đến TRẠNG THÁI MỚI
                new_action = node_action + [action[-1]]
                new_f =  heuristic(newPosPlayer, newPosBox) + cost(new_action[1:]) # Tính f của NODE MỚI chứa TRẠNG THÁI MỚI trên
                # Thêm TRẠNG THÁI MỚI vào cuối node đang xét để được NODE MỚI rồi thêm NODE MỚI này vào frontier, độ ưu tiên là new_f
                frontier.push(node + [(newPosPlayer, newPosBox)], new_f)
                actions.push(new_action, new_f) # Thêm ĐƯỜNG ĐI MỚI vào actions với độ ưu tiên là new_f

    # end =  time.time()

    return temp # Trả về đường đi từ start_state đến goalState, hoặc trả về mảng rỗng nếu không thể tìm thấy đường đi nào
```

Cài đặt thuật toán A Star Search

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

1. Bảng thống kê kết quả

** Chạy và thống kê các thuật toán bằng laptop **Dell G15 5515 R5 5600H/8GB**.*

Kí hiệu:

+ A*(M): A Star có heuristic là tổng khoảng cách Manhattan giữa các thùng và các vị trí đích.

+ A*(E): A Star có heuristic là tổng khoảng cách Euclid giữa các thùng và các vị trí đích.

Màn	Thời gian (s)			Số nút đã mở		
	UCS	A*(M)	A*(E)	UCS	A*(M)	A*(E)
1	0.05	0.01	0.02	720	122	223
2	0.00	0.00	0.00	64	39	39
3	0.08	0.01	0.01	509	54	49
4	0.00	0.00	0.00	55	29	29
5	74.09	0.08	0.06	357203	485	349
6	0.01	0.01	0.01	250	208	219
7	0.51	0.06	0.12	6046	715	1097
8	0.20	0.21	0.23	2383	2352	2365
9	0.01	0.00	0.00	74	42	42
10	0.02	0.02	0.02	218	198	198
11	0.02	0.02	0.02	296	284	284
12	0.09	0.04	0.05	1225	563	628
13	0.17	0.13	0.16	2342	1699	1820
14	2.86	0.89	1.25	26352	8108	10057
15	0.27	0.26	0.30	2505	2183	2261
16	16.68	0.28	0.46	57275	1286	1927
17	25.12	25.90	27.09	71595	71595	71595
18	∞	∞	∞	-	-	-

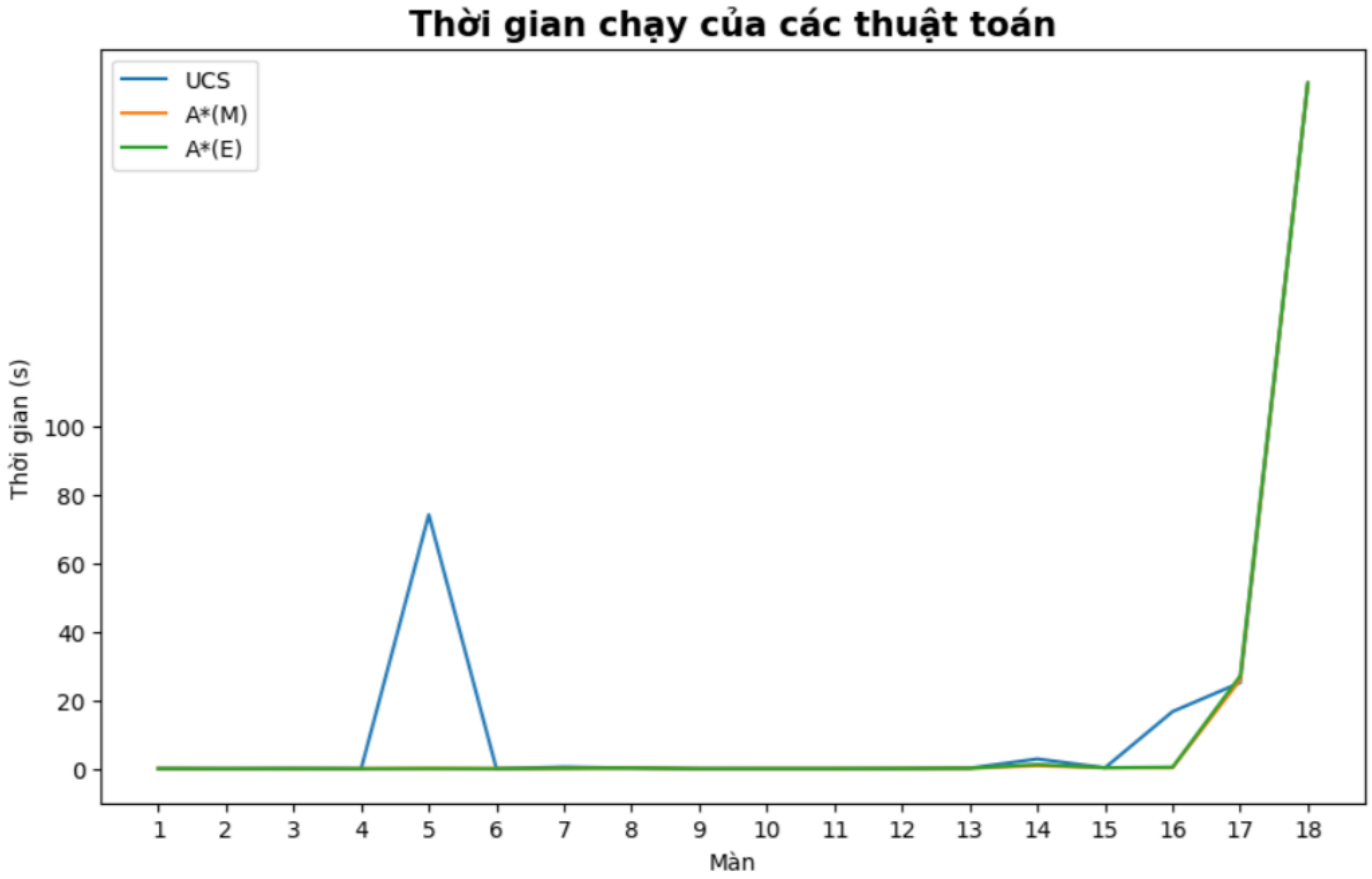
**** Thời gian biểu diễn bằng ∞ ở các trường hợp mà để máy chạy thuật toán 6 - 7 tiếng nhưng chưa ra kết quả.***

Màn	Số bước đi			Lời giải tối ưu?		
	UCS	A*(M)	A*(E)	UCS	A*(M)	A*(E)
1	12	13	12	Yes	**No**	Yes
2	9	9	9	Yes	Yes	Yes
3	15	15	15	Yes	Yes	Yes
4	7	7	7	Yes	Yes	Yes
5	20	22	20	Yes	**No**	Yes
6	19	19	19	Yes	Yes	Yes
7	21	21	21	Yes	Yes	Yes
8	97	97	97	Yes	Yes	Yes
9	8	8	8	Yes	Yes	Yes
10	33	33	33	Yes	Yes	Yes
11	34	34	34	Yes	Yes	Yes
12	23	23	23	Yes	Yes	Yes
13	31	31	31	Yes	Yes	Yes
14	23	23	23	Yes	Yes	Yes
15	105	105	105	Yes	Yes	Yes
16	34	42	36	Yes	**No**	**No**
17	0	0	0	Yes	Yes	Yes
18	-	-	-	-	-	-

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

2. Nhận xét

- **Về thời gian:**

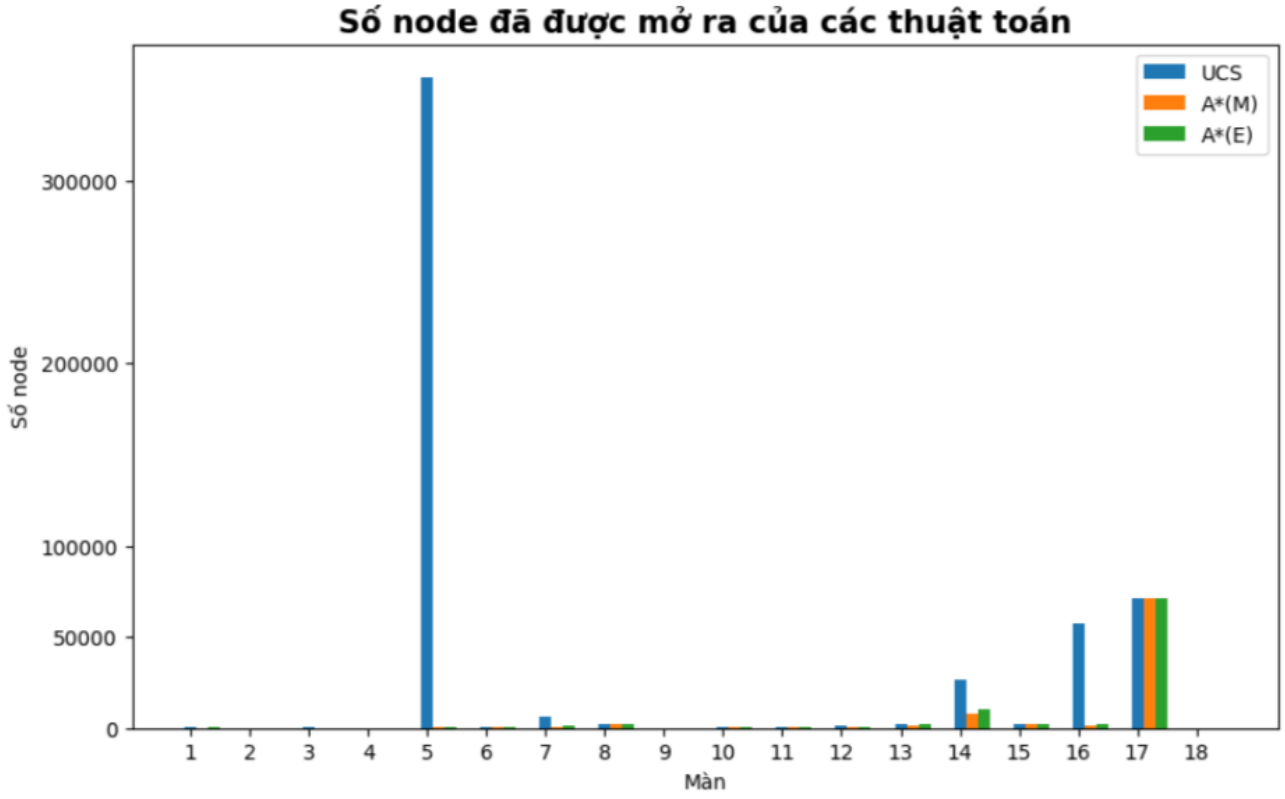


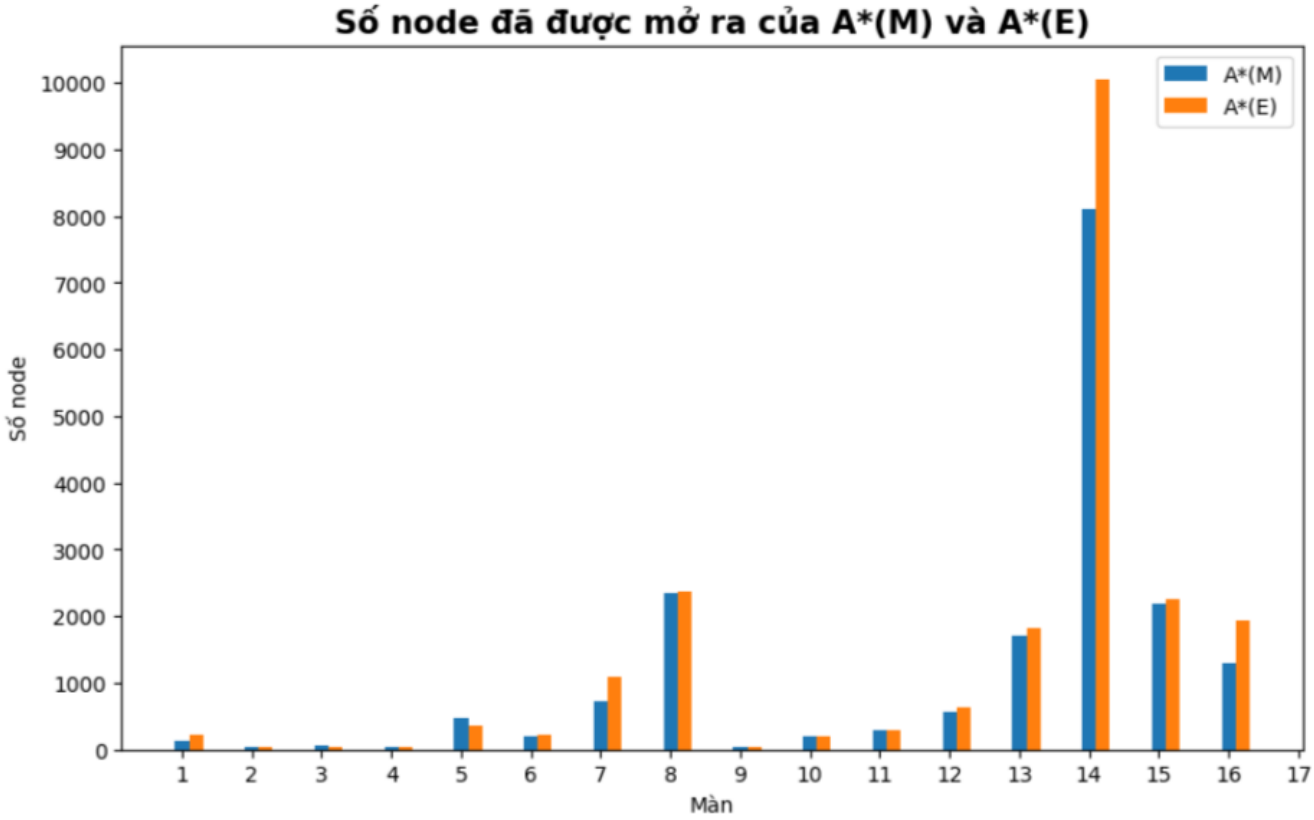
Dễ thấy các thuật toán A* đều có thời gian chạy nhanh hơn so với UCS ở hầu hết các màn chơi. Đặc biệt là ở màn 5 và màn 16, A* vượt trội hơn UCS rất nhiều. Khi UCS cần đến 74.09s để chạy màn 5 thì A*(E) chỉ cần 0.06s (nhanh hơn gấp 1234 lần), hay A*(M) chậm hơn cũng chỉ cần 0.08s (nhanh hơn gấp 926 lần UCS). Ở màn 16, A* chạy dưới 0.5s thì UCS chạy tới 16.68s.

A*(M) có thời gian chạy tương đối nhanh so với A*(E) ở nhiều màn nhưng sự chênh lệch không đáng kể.

Thuật toán A* đã cải thiện rất nhiều về tốc độ so với UCS nhưng **vẫn không chạy nổi màn 18**.

- **Về số node đã được mở ra:**



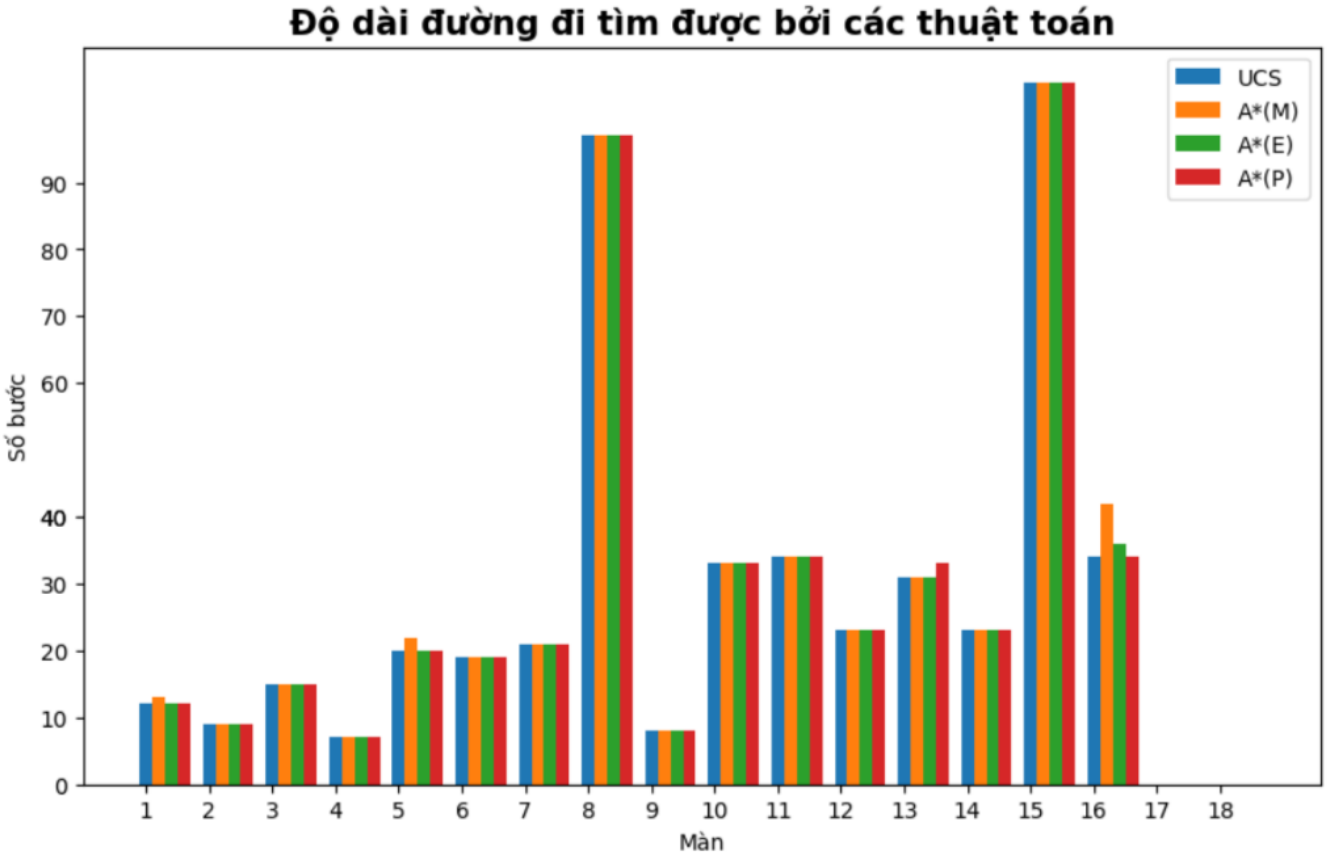


Có thể thấy UCS chạy chậm hơn so với A* là do UCS phải mở một lượng rất lớn các node mới tìm được lời giải. Tất cả các màn từ 1 đến 16, số lượng node được mở bởi A* nhỏ hơn UCS rất rất nhiều. Màn 5, UCS A*(M) chỉ cần mở 485 node, trong khi UCS mở tới 357203 node (nhiều gấp 736 lần).

A*(M) mở ít node hơn so với A*(E) ở hầu hết các màn (đây là một phần lí do giúp A*(M) chạy nhanh hơn A*(E), ngoài ra còn do việc tính khoảng cách Manhattan đơn giản hơn tính khoảng cách Euclid)

Màn 17 không có lời giải nên tất cả các thuật toán phải mở hết tất cả các node có thể có là 71595 (USC ở màn này nhanh hơn A* do không phải tính toán heuristic).

- **Về độ dài đường đi:**



Đường đi tìm được bởi UCS ở tất cả các màn từ 1 đến 17 đều là lời giải tối ưu.

Đường đi của A*(M) ở 3 màn 1, 5, 16 không phải lời giải tối ưu trong khi A*(E) chỉ có màn 16 không phải lời giải tối ưu.

3. Kết luận

Từ màn 1 đến 17, UCS tuy luôn cho lời giải tối ưu nhưng do số node nó phải mở để tìm được lời giải là rất lớn nên thời gian chạy UCS khá chậm.

Nhờ **thông tin ước lượng chi phí đến trạng thái đích của hàm heuristic kết hợp với chi phí đường đi từ trạng thái khởi đầu**, các thuật toán A* tìm được lời giải mà chỉ cần mở rất ít node (so với UCS) dẫn đến thời gian chạy nhanh và lời giải ở đa số các màn đều là tối ưu.

Thuật toán A* phụ thuộc nhiều vào hàm heuristic. Do đó, điều quan trọng của A* là phải thiết kế được hàm heuristic ước lượng tốt. Có sự đánh đổi giữa lời giải tối ưu và thời gian chạy khi thiết kế các hàm heuristic. **Nhưng nhìn chung A*(E) sẽ tốt hơn A*(M) khi chơi trò chơi này.**