

Sử dụng các giải thuật tìm kiếm để giải trò chơi Sokoban

Nhập môn trí tuệ nhân tạo

GVHD: Vương Bá Thịnh

Link github: <https://github.com/nh0znoisung/sokoban>

Link video trình bày: <https://www.youtube.com/watch?v=oT5ag8KVyHA>





Giới thiệu thành viên

1914637 Tô Thanh Phong

1914405 Võ Anh Nguyên

1910663 Quách Minh Tuấn

Nội dung

1

Giới thiệu về trò chơi

2

Phân tích trò chơi

3

Thực hiện các giải thuật tìm kiếm

4

**Một số giải pháp nhóm đề xuất để
tối ưu hóa lời giải**

5

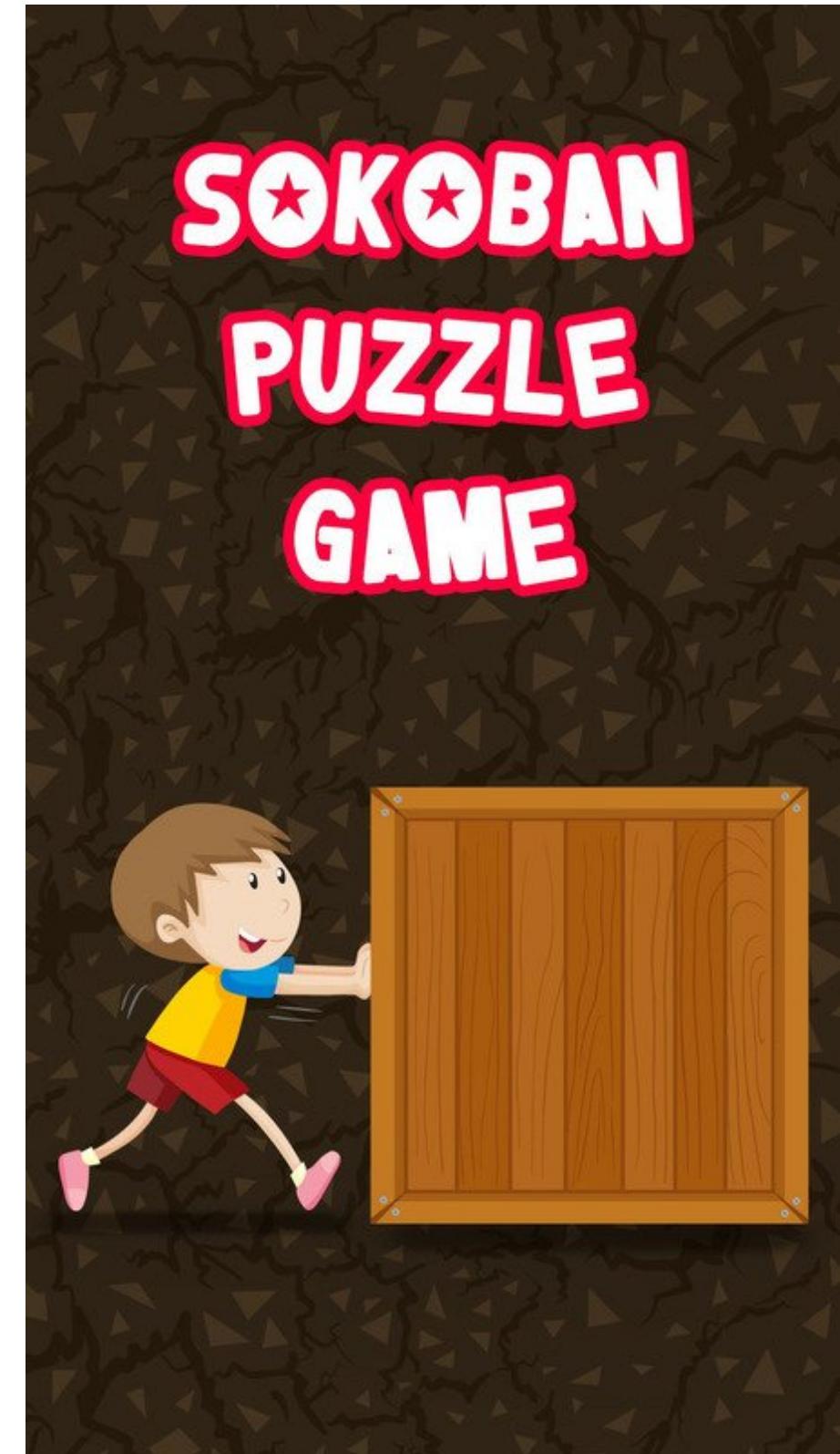
Nhận xét kết quả



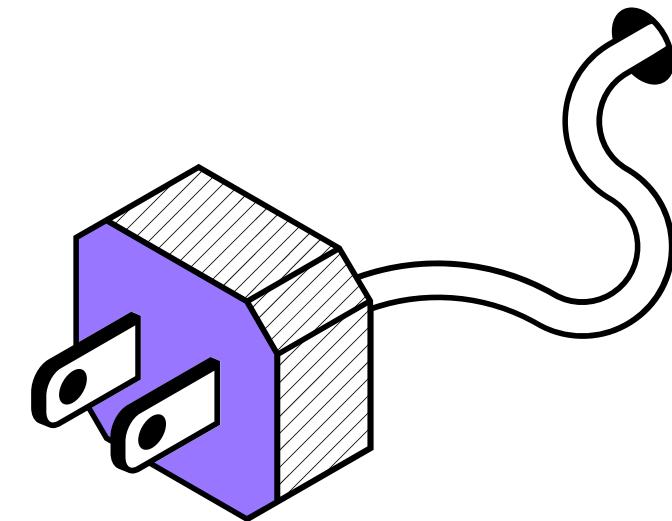
Giới thiệu về trò chơi

Giới thiệu

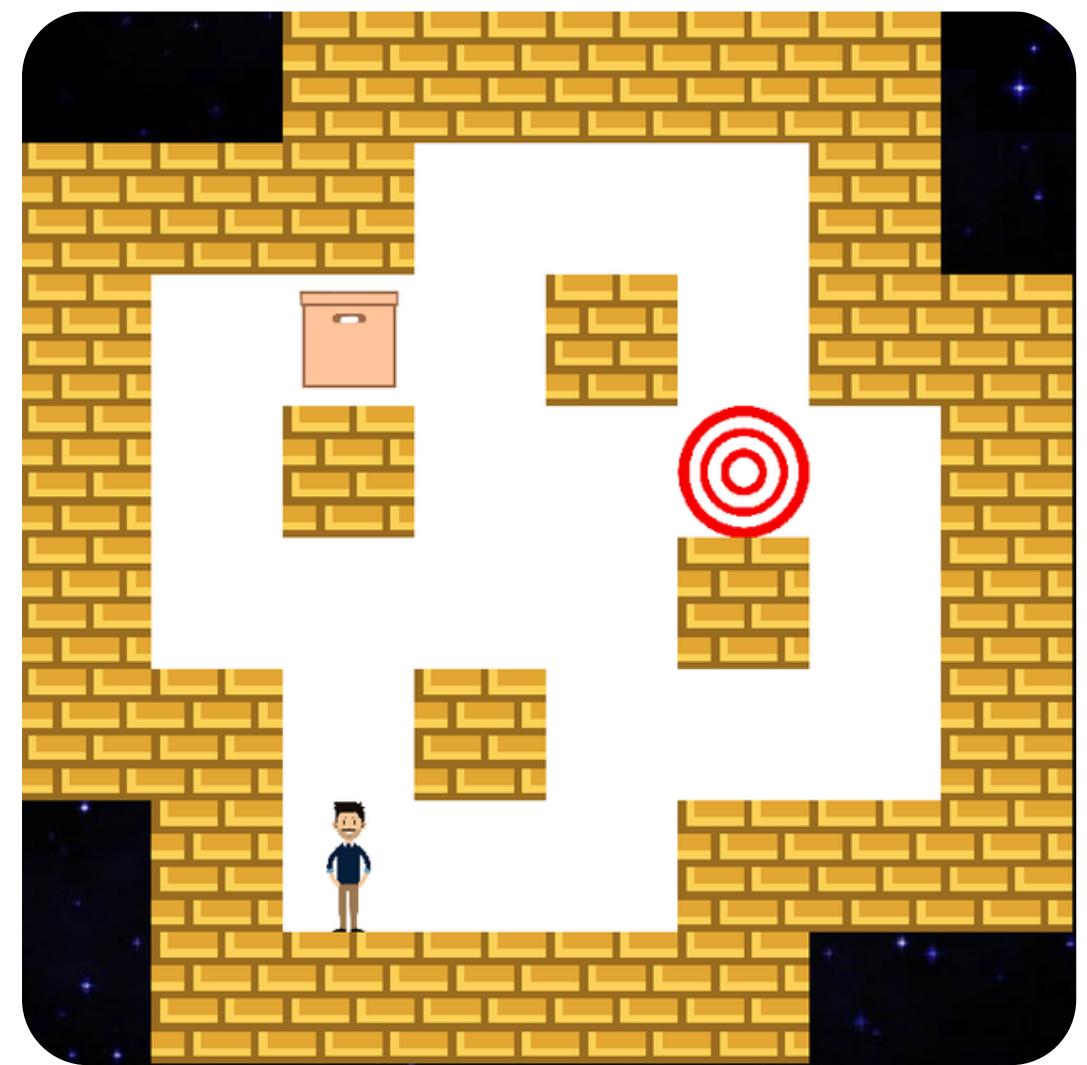
Sokoban là một trò chơi điện tử mang tính giải đố (puzzling game) có nguồn gốc từ Nhật Bản. Trong tiếng Nhật, "sokoban" có nghĩa là người quản lí kho hàng. Trò chơi này được phát triển bởi Hiroyuki Imai vào năm 1981 và được phát hành 1 năm sau đó.



Luật chơi

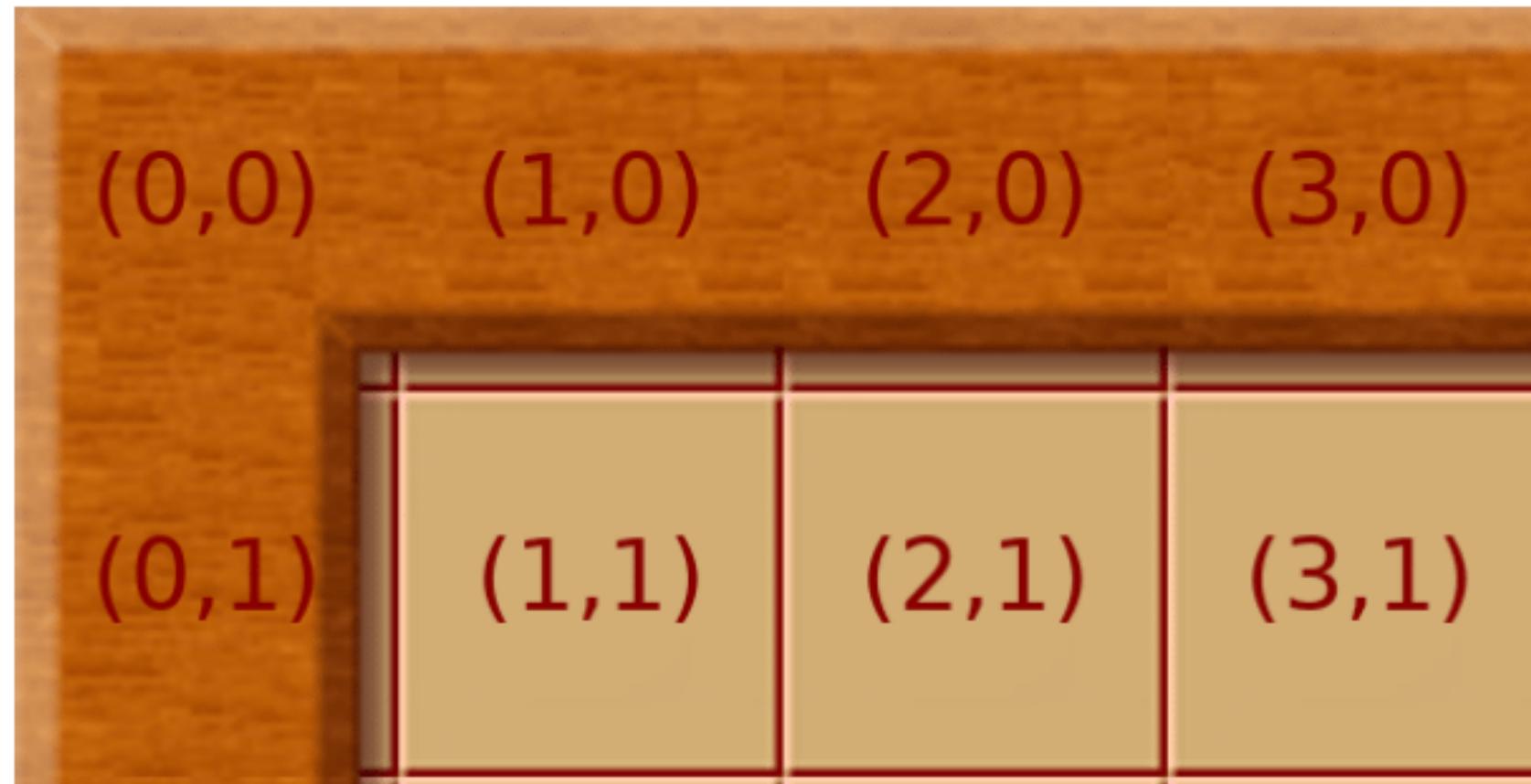


Mỗi "cấp độ" trong trò chơi mô phỏng một nhà kho, trong đó các chiếc thùng được đặt một cách ngẫu nhiên. Nhiệm vụ của "người quản lí kho hàng" này là đẩy các thùng này đến vị trí đích được sắp xếp ngẫu nhiên trong kho



Phân tích trò chơi

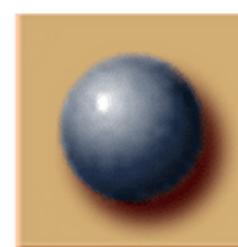
Một số quy ước



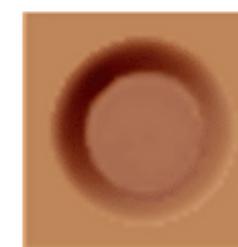
Mỗi cấp độ bao gồm một ma trận hai chiều hình chữ nhật tạo thành "nhà kho". Các ô vuông được lập chỉ mục bắt đầu từ trên cùng bên trái với tọa độ (0; 0). Nếu một hình vuông không chứa gì nó được gọi là sàn (floor). Nếu không, nó sẽ bị chiếm giữ bởi một trong các thực thể sau:



a) Tường



b) Hộp



c) Đích

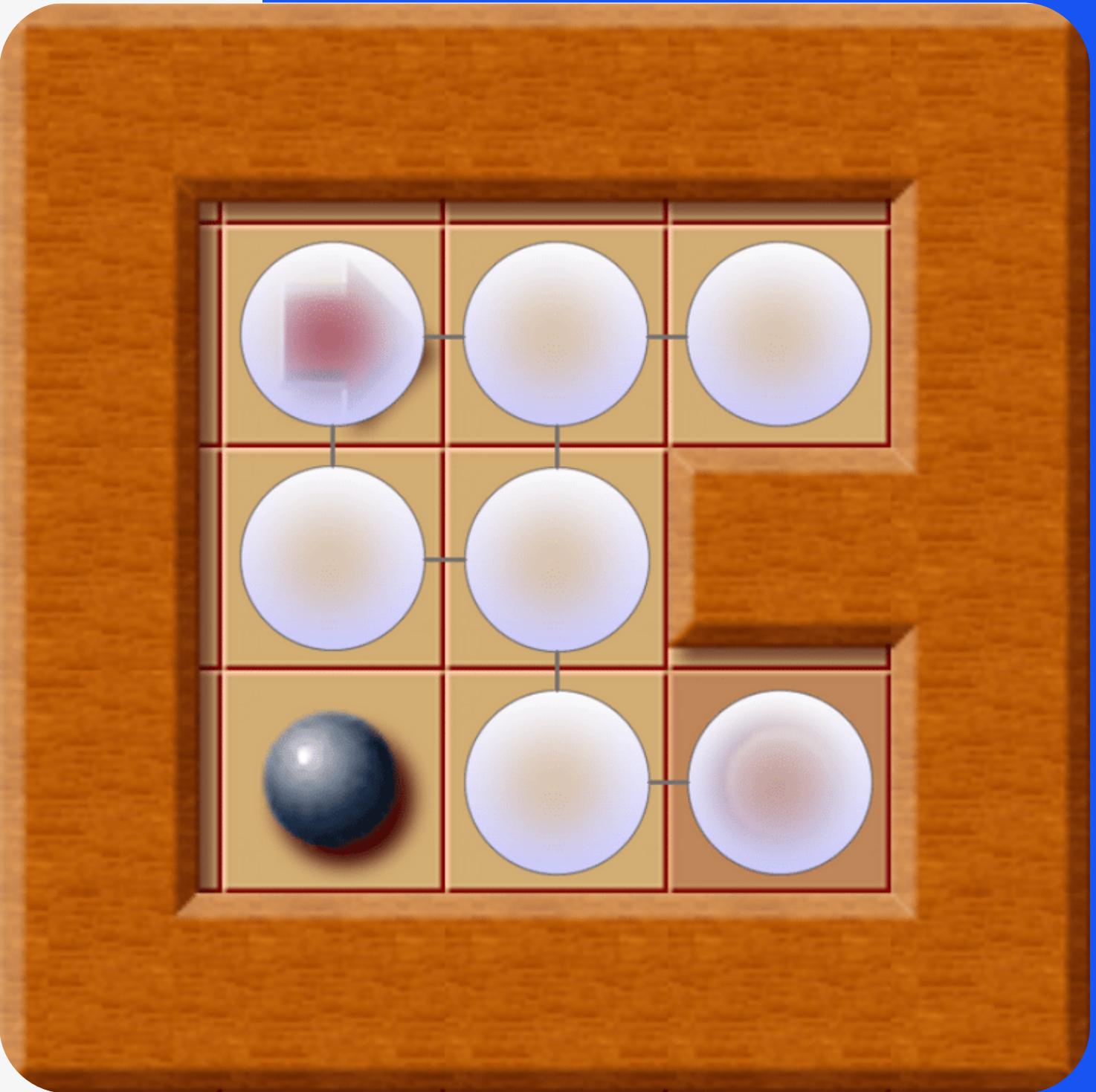


d) Người chơi

Không gian trò chơi

Không gian trò chơi là một đồ thị mà các đỉnh được tạo thành từ các hình vuông mà người chơi có thể di chuyển được và hai đỉnh chỉ được nối với nhau bằng một cạnh khi người chơi có thể chuyển đổi từ vị trí này sang vị trí khác trong một lần di chuyển. Vì người chơi luôn có thể di chuyển trở lại nên đồ thị là vô hướng.

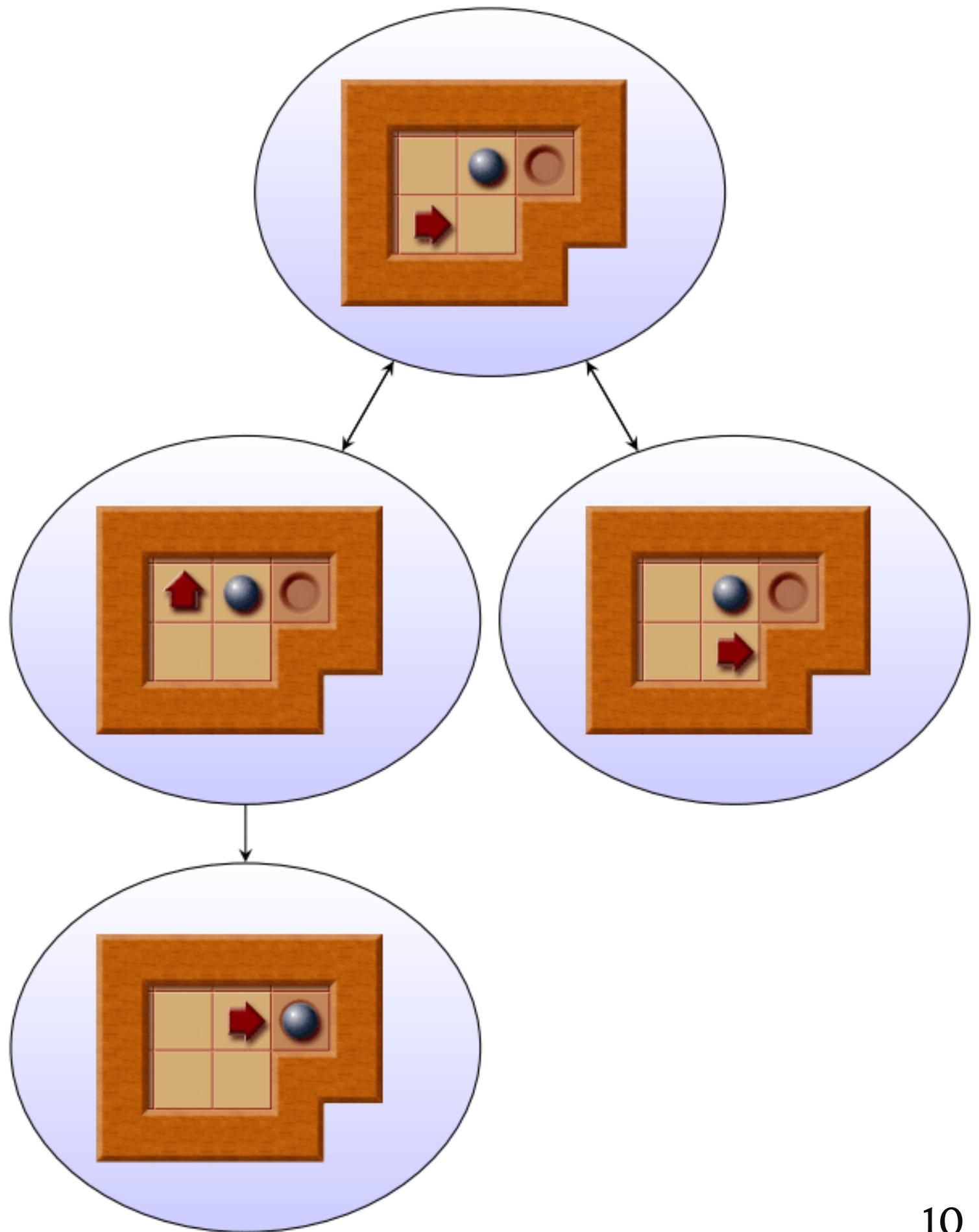
Đồ thị của không gian trò chơi thay đổi mỗi khi một trong các hộp được di chuyển.



Không gian trạng thái

Mỗi trạng thái được xác định bằng vị trí của người chơi và vị trí của các hộp. Khi chơi trò chơi có thể được coi là chuyển đổi từ trạng thái này sang trạng thái khác, tạo ra một đồ thị trong đó các đỉnh là các trạng thái có thể tiếp cận và một cạnh biểu thị sự chuyển đổi hợp pháp từ trạng thái này sang trạng thái khác. Đồ thị có hướng vì quá trình chuyển đổi có khả năng không thể đảo ngược.

Tập hợp các bước di chuyển khác nhau có thể dẫn đến cùng một trạng thái, do đó đồ thị có thể chứa các chu trình.

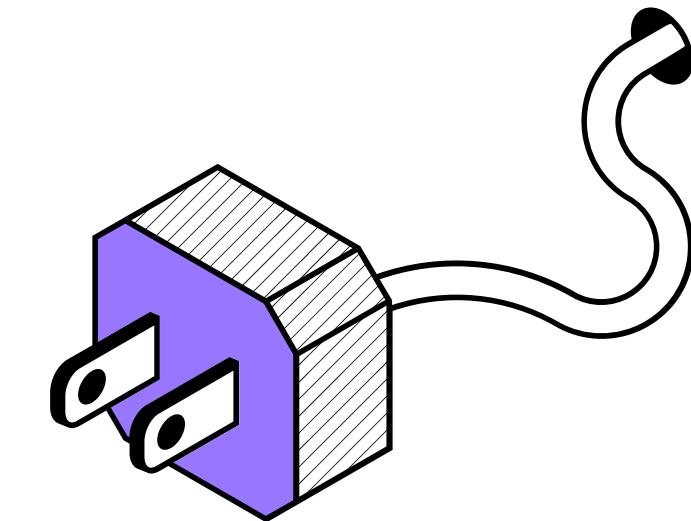


Định nghĩa các yếu tố

- State: Vị trí hiện tại của người chơi và vị trí hiện tại của các hộp
- Initial state: Vị trí ban đầu của người chơi và các hộp
- Goal state: Tất cả vị trí của các hộp trùng với vị trí của các đích
- Legal moves: Giả sử vị trí hiện tại của người chơi là (x,y) , người chơi có thể đến được các vị trí sau
 - Lên trên: $(x,y) \rightarrow (x, y-1)$ nếu
 - + Vị trí $(x, y-1)$ không có hộp và không có tường
 - + Vị trí $(x, y-1)$ có hộp thì xét $(x, y-2)$ phải không có hộp và không có tường, vị trí hộp khi đó là $(x,y-1) \rightarrow (x,y-2)$
 - Xuống dưới: $(x,y) \rightarrow (x, y+1)$ nếu
 - + Vị trí $(x, y+1)$ không có hộp và không có tường
 - + Vị trí $(x, y+1)$ có hộp thì xét $(x,y+2)$ phải không có hộp và không có tường, vị trí hộp khi đó là $(x,y+1) \rightarrow (x,y+2)$
 - Qua trái: $(x, y) \rightarrow (x-1, y)$ nếu
 - + Vị trí $(x-1, y)$ không có hộp và không có tường
 - + Vị trí $(x-1, y)$ có hộp thì xét $(x-1, y)$ phải không có hộp và không có tường, vị trí hộp khi đó là $(x,y-1) \rightarrow (x,y-2)$
 - Qua phải: $(x,y) \rightarrow (x+1, y)$ nếu
 - + Vị trí $(x+1, y)$ không có hộp và không có tường
 - + Vị trí $(x+1, y)$ có hộp thì xét $(x+1, y)$ phải không có hộp và không có tường, vị trí hộp khi đó là $(x,y-1) \rightarrow (x,y-2)$

Thực hiện các giải thuật tìm kiếm

Breadth First Search

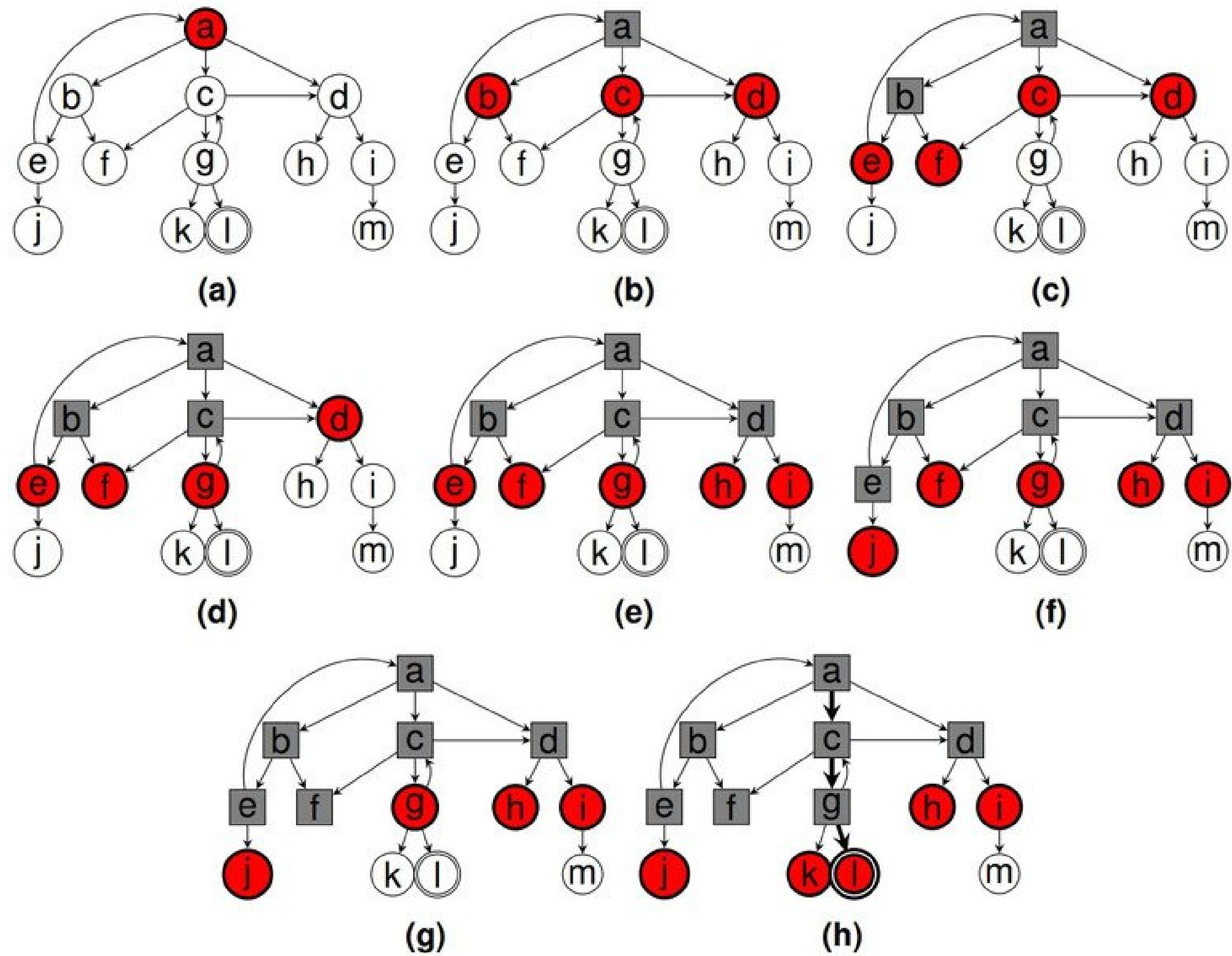


Giới thiệu giải thuật

Sử dụng hàng đợi (queue). Bắt đầu với đỉnh gốc, chèn tất cả những đỉnh kề của nó vào cuối hàng đợi và sau đó tiếp tục với đỉnh đầu tiên trong hàng đợi.

Giả sử hệ số nhánh (branching factor) không đổi là b và độ sâu của lời giải (solution dept) là d thì không gian lưu trữ cần sử dụng là b^d (b lũy thừa d)

Đối với không gian tìm kiếm lớn, nơi hầu hết các đỉnh có nhiều hơn một đỉnh con, điều này sẽ dẫn đến vấn đề bộ nhớ.



Chú thích

- "a" là đỉnh gốc và "l" là đỉnh nghiệm duy nhất.
- Các đỉnh màu trắng là đỉnh chưa được duyệt
- Các đỉnh màu xám là các đỉnh đã được duyệt.
- Các đỉnh màu đỏ là các đỉnh đang nằm trong hàng đợi

Mã giả

Input: rootVertex

Result: Path to a solution vertex

Begin

 visited $\leftarrow \emptyset$ // Set of visited vertices

 visited.add(rootVertex)

 queue.push(rootVertex) // Queue of vertices with rootVertex as the only element

while queue $\neq \emptyset$ **do**

 vertex = queue.pop()

foreach successor of vertex **do**

if isSolution(successor) **then**

 return pathTo(successor)

if not visited.contains(successor) **then**

 queue.push(successor)

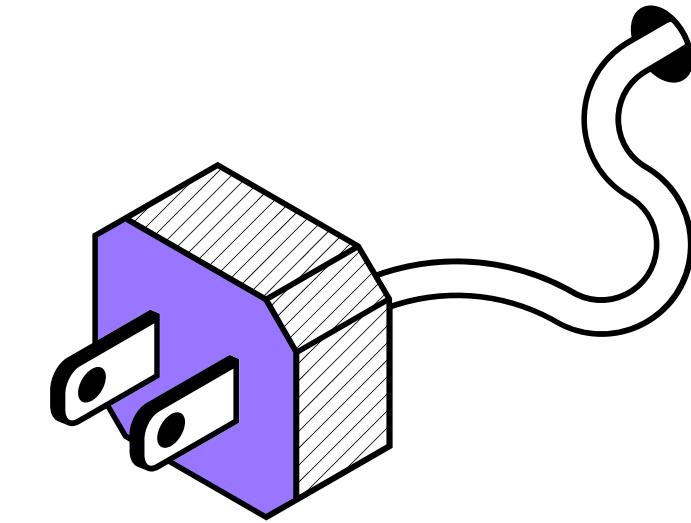
 visited.add(successor)

 return no solution found.

End



A*



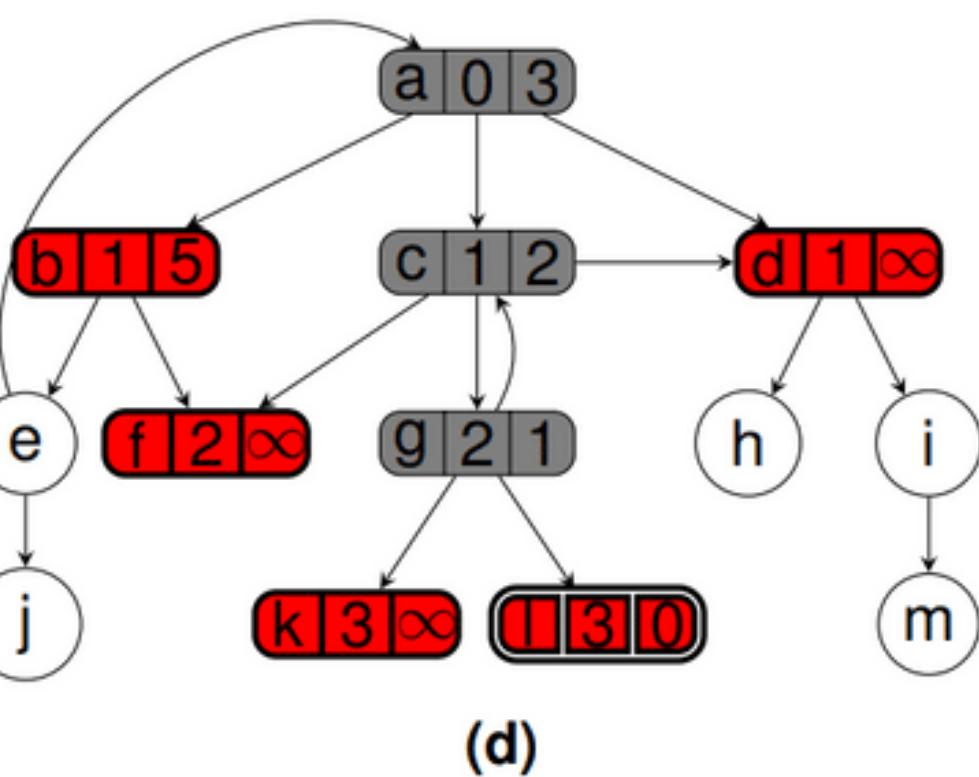
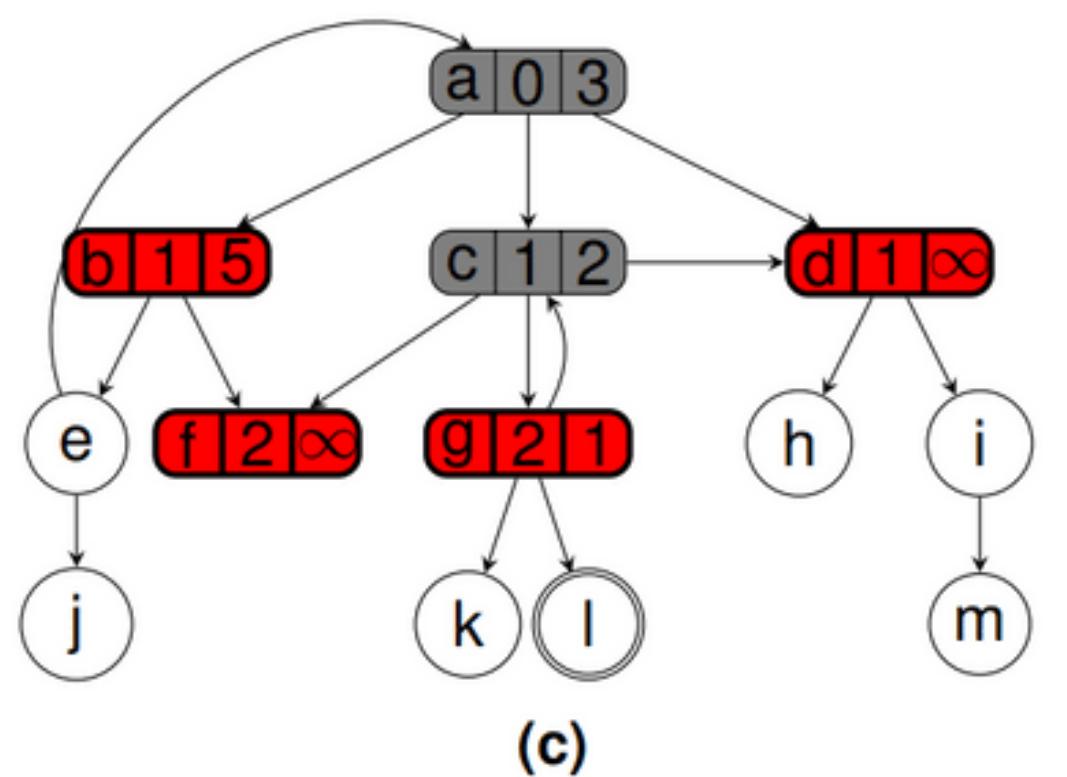
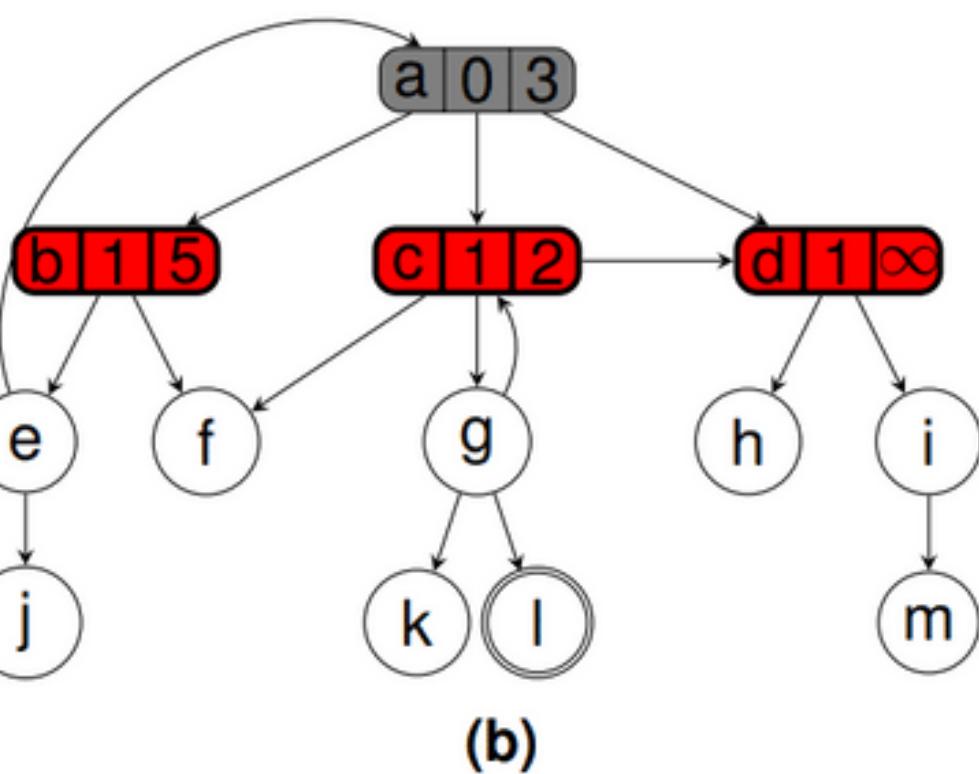
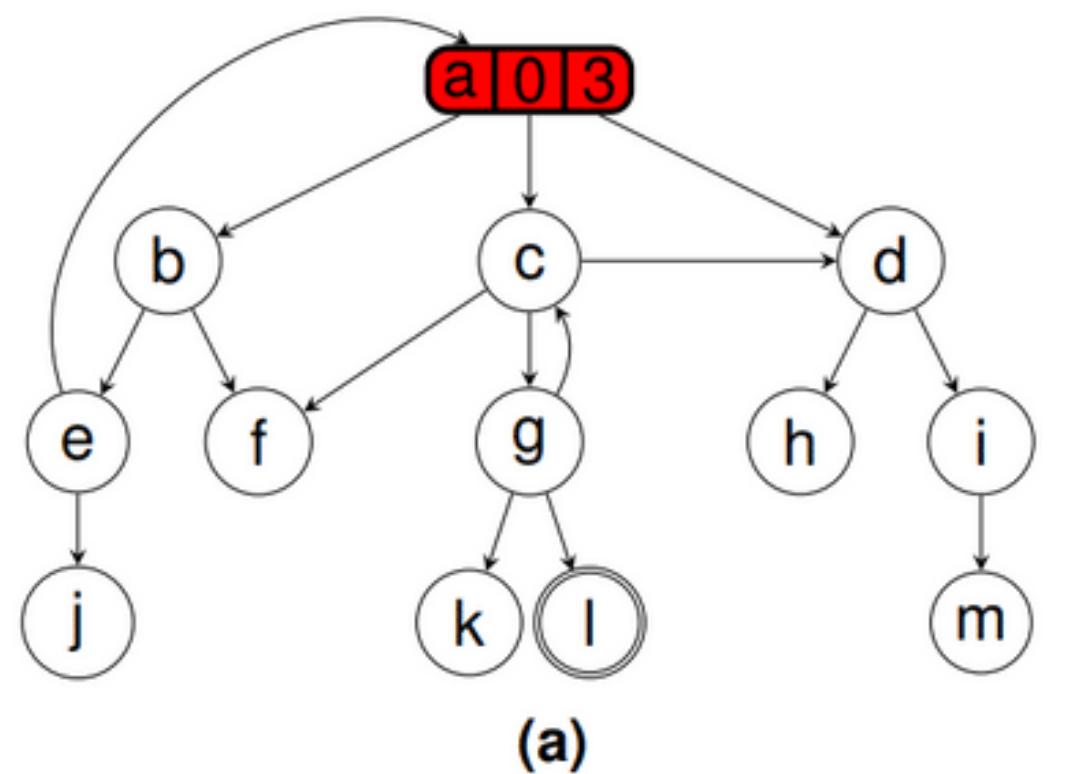
Giới thiệu giải thuật

A* là một trong các thuật toán Informed Search (tìm kiếm theo kinh nghiệm). Nó xây dựng một cây gồm các đường dẫn từng phần bắt đầu từ đỉnh gốc cho đến khi một trong các đường dẫn kết thúc ở đỉnh nghiệm. Đỉnh cuối cùng của mỗi đường dẫn từng phần được chèn vào hàng đợi ưu tiên (priority queue) được sắp xếp theo chi phí của đường dẫn đến đỉnh đó cộng với khoảng cách còn lại tới nghiệm như ước tính của phương pháp heuristic:

$$f(n) = g(n) + h(n)$$

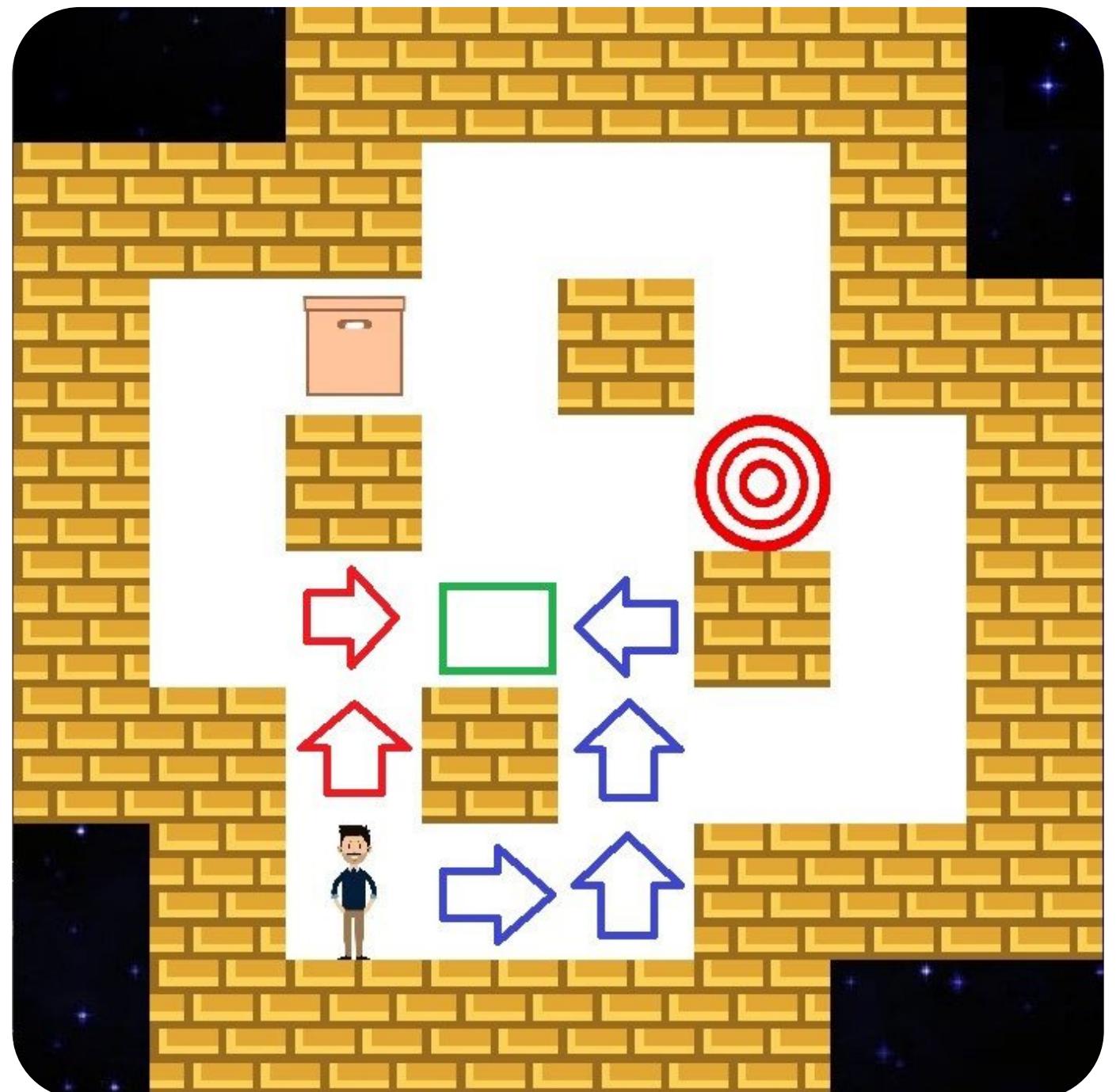
Chú thích

- "a" là đỉnh gốc và "l" là đỉnh nghiệm duy nhất.
- Các đỉnh màu trắng là đỉnh chưa được duyệt
- Các đỉnh màu xám là các đỉnh đã được duyệt.
- Các đỉnh màu đỏ là các đỉnh đang nằm trong hàng đợi
- Giá trị thứ 2 của đỉnh là khoảng cách từ đỉnh gốc và giá trị thứ 3 là khoảng cách tới đích được ước tính bằng heuristic



Transposition Table

Trong đồ thị không gian trạng thái, nhiều đường đi có thể dẫn đến cùng một đỉnh. Do đó, chúng ta cần một cách để nhận ra các trạng thái đã được duyệt để ngăn chặn việc tính toán không cần thiết. Điều này được thực hiện bằng cách sử dụng một bảng chuyển vị (Transposition Table). Một cách phổ biến để thực hiện nó là sử dụng bảng băm các trạng thái. Ở đây nhóm chúng em sử dụng cấu trúc set trong python, vì set sử dụng bảng băm khi cần tìm kiếm giá trị và độ phức tạp khi tìm kiếm là $O(1)$ (nếu không xảy ra đụng độ (collision))

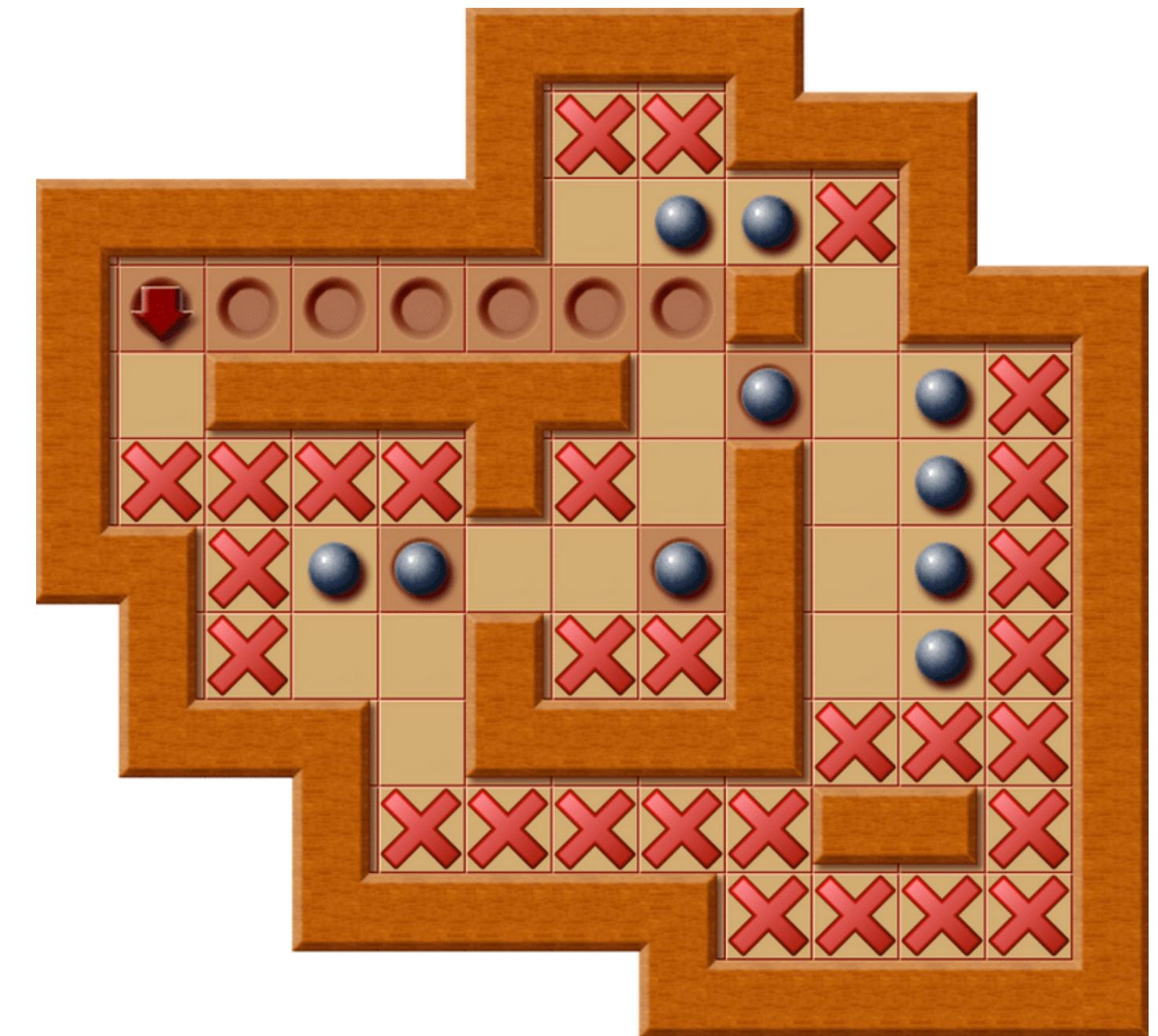


Tính toán khoảng cách đến các đích và phát hiện Deadlock

Một hình vuông được gọi là "dead" nếu một hộp được đặt trên nó không bao giờ có thể được đẩy đến bất cứ một đích nào, đặt hộp trên hình vuông đó sẽ dẫn đến "deadlock".

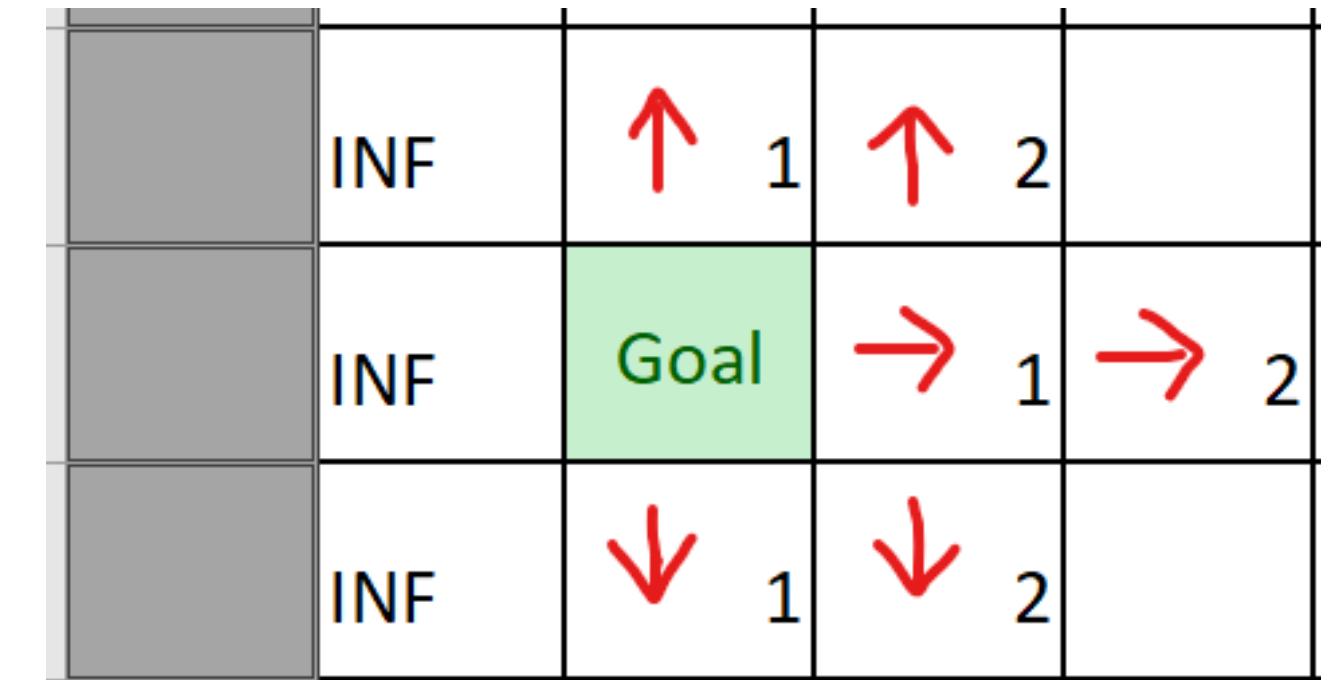
Trong bài toán này, nhóm chúng em chỉ sử dụng những dữ liệu tĩnh như là vị trí của đích và tường để tính toán Deadlock.

Nhóm chúng em sử dụng phương pháp là "Goal pull distance" để tính toán khoảng cách từ mọi vị trí đến mọi đích. Nếu vị trí nào có khoảng cách đến mọi đích đều là vô cùng thì vị trí đó chính là deadlock.



Phương pháp tính khoảng cách: Goal pull Distance

Phương pháp này yêu cầu các bước tiền xử lý, chúng em thực hiện điều này bằng cách sử dụng thuật toán **breath first search** để "kéo" một hộp từ đích, tức là xuất phát từ một đích bất kì, kiểm tra 4 vị trí liền kề (trên, dưới, trái, phải) xem một hộp được đặt ở vị trí đó thì nó có thể được đẩy tới đích hay không. Các ô vuông này sau đó được đánh dấu khoảng cách đến đích hiện tại và thuật toán tiếp tục bằng cách kiểm tra các ô vuông liền kề của chúng. Điều này được lặp lại cho tất cả các đích.



Mã giả

Result: distance from all positions to all goals

begin

 distanceToGoal [Goals][Positions] $\leftarrow \infty$

foreach goal **do**

 distanceToGoal [goal][goal .position] = 0

 queue.push(goal .position) // *FIFO queue with the position of the goal as the only element*

while queue $\neq \emptyset$ **do**

 position = queue.pop()

foreach direction **do**

 boxPosition = position + direction

 playerPosition = position + 2 · direction

if distanceToGoal [goal][boxPosition] = ∞ **then**

if not wallAtPosition (boxPosition) and

 not wallAtPosition (playerPosition) **then**

 distanceToGoal [goal][boxPosition] =

 distanceToGoal [goal][position]+1

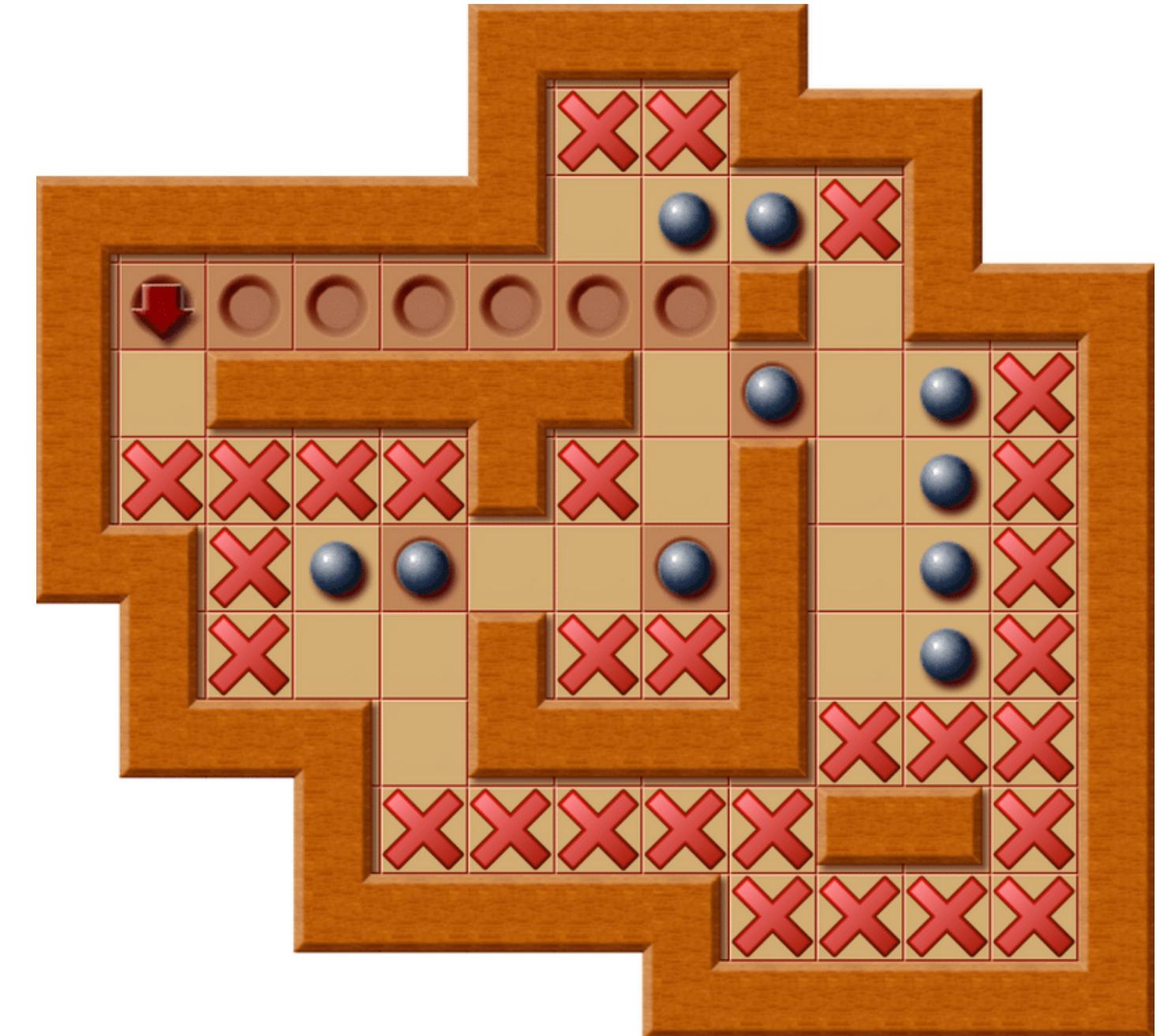
 queue.push(boxPosition)

INF	↑	1	↑	2	
INF	Goal	→	1	→	2
INF	↓	1	↓	2	

Các Deadlock là những vị trí có khoảng cách đến tất cả các đích bằng vô cùng.

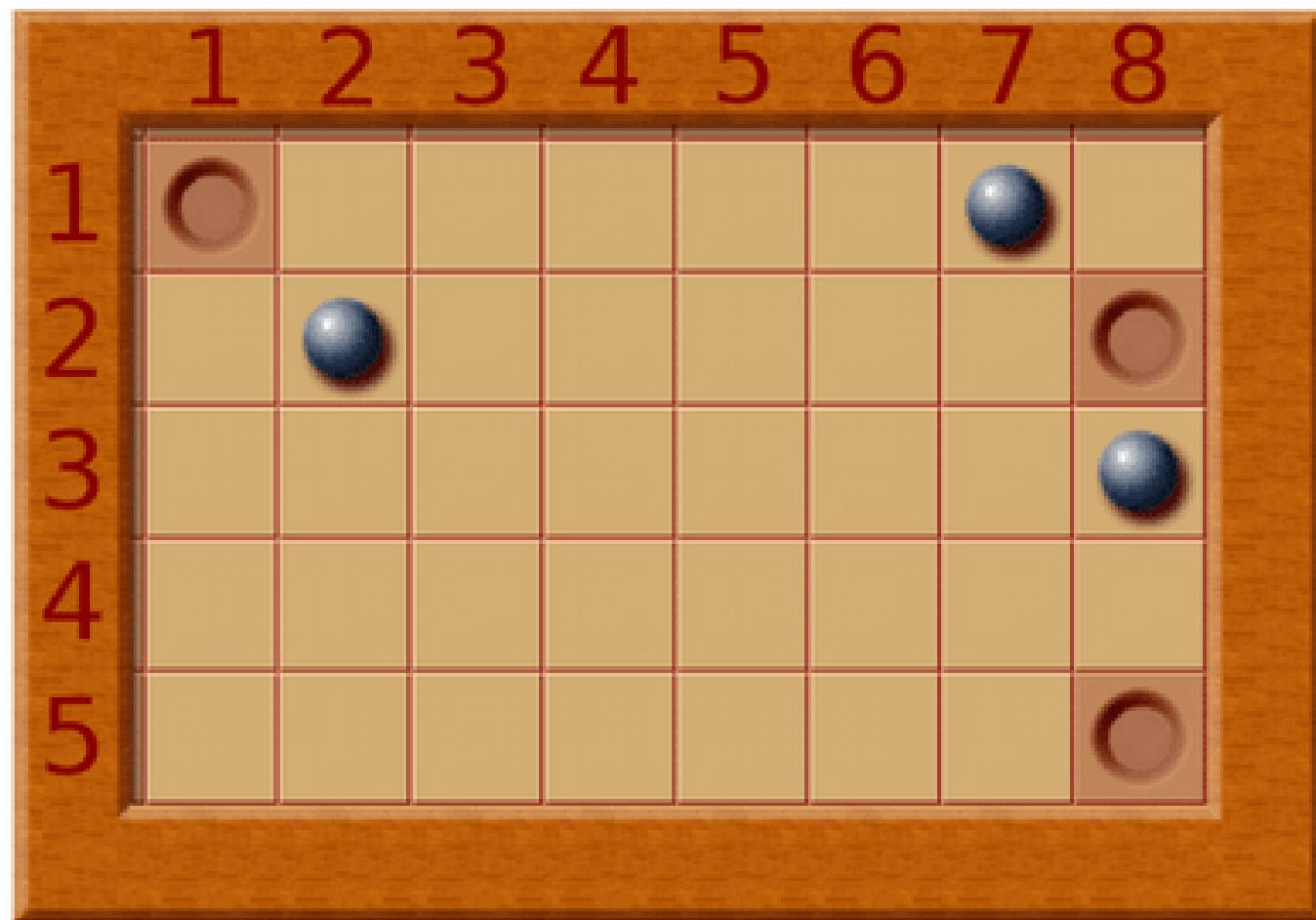
Sau khi tính được Deadlock thì không gian trạng thái sẽ được giảm đi đáng kể, sẽ rất hiệu quả khi xài các giải thuật tìm kiếm.

Phương pháp này có lợi thế so với các phương pháp tính khoảng cách như Manhattan hay Pytago vì nó giúp chúng ta nhận biết được các Deadlock trong khi các phương pháp tính khoảng cách đó không làm được.

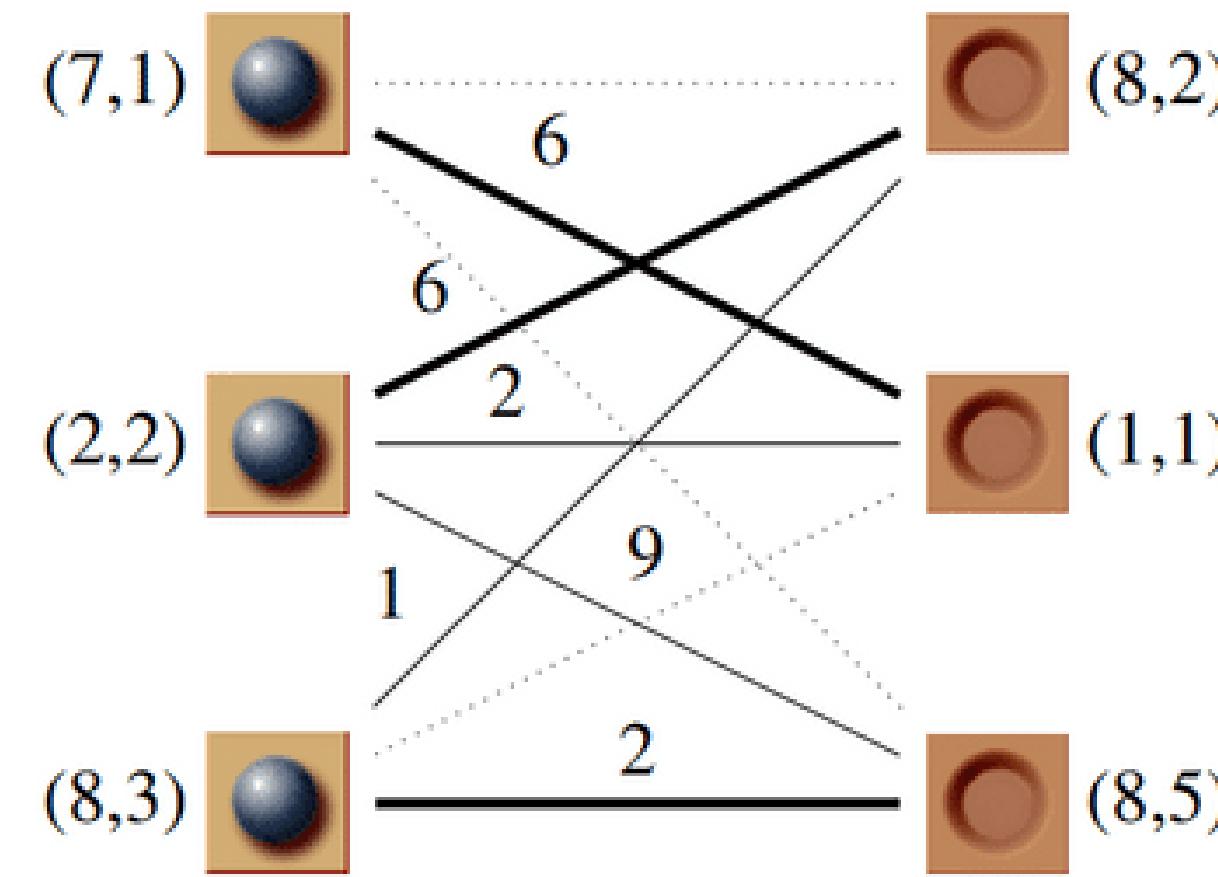


Thuật toán gán (Assignment algorithm)

Giả sử, ta đã sử dụng phương pháp "goal pull distance" đã được giới thiệu ở các slide trước và tính được các khoảng cách từ các hộp đến các đích như sau (trong đó, đường nét đứt biểu thị khoảng cách là vô cùng):

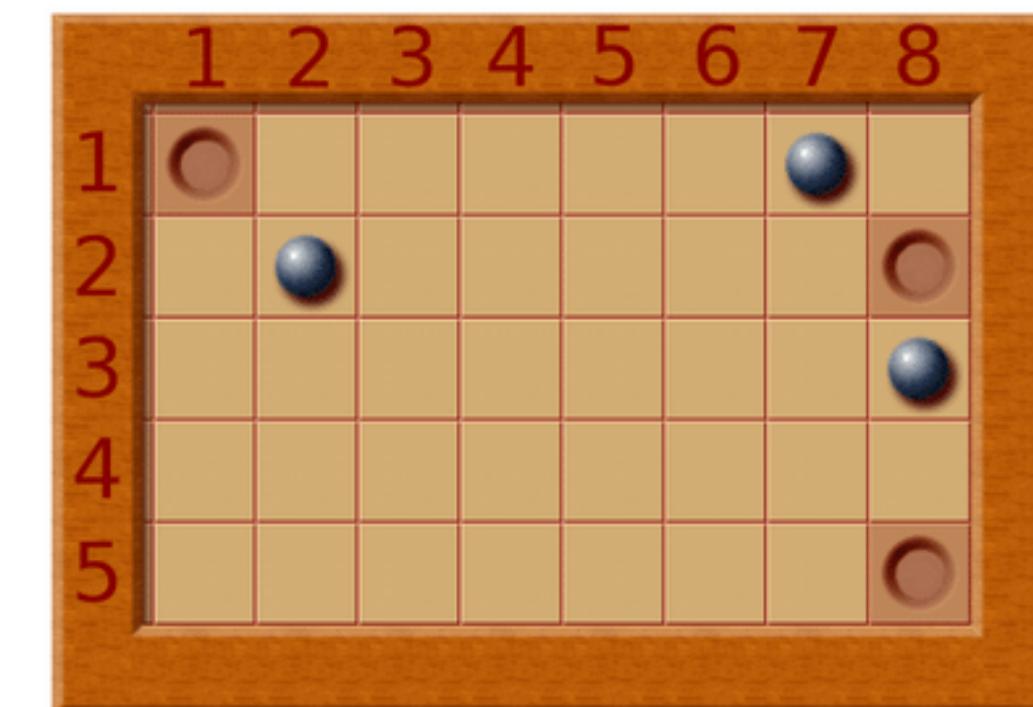
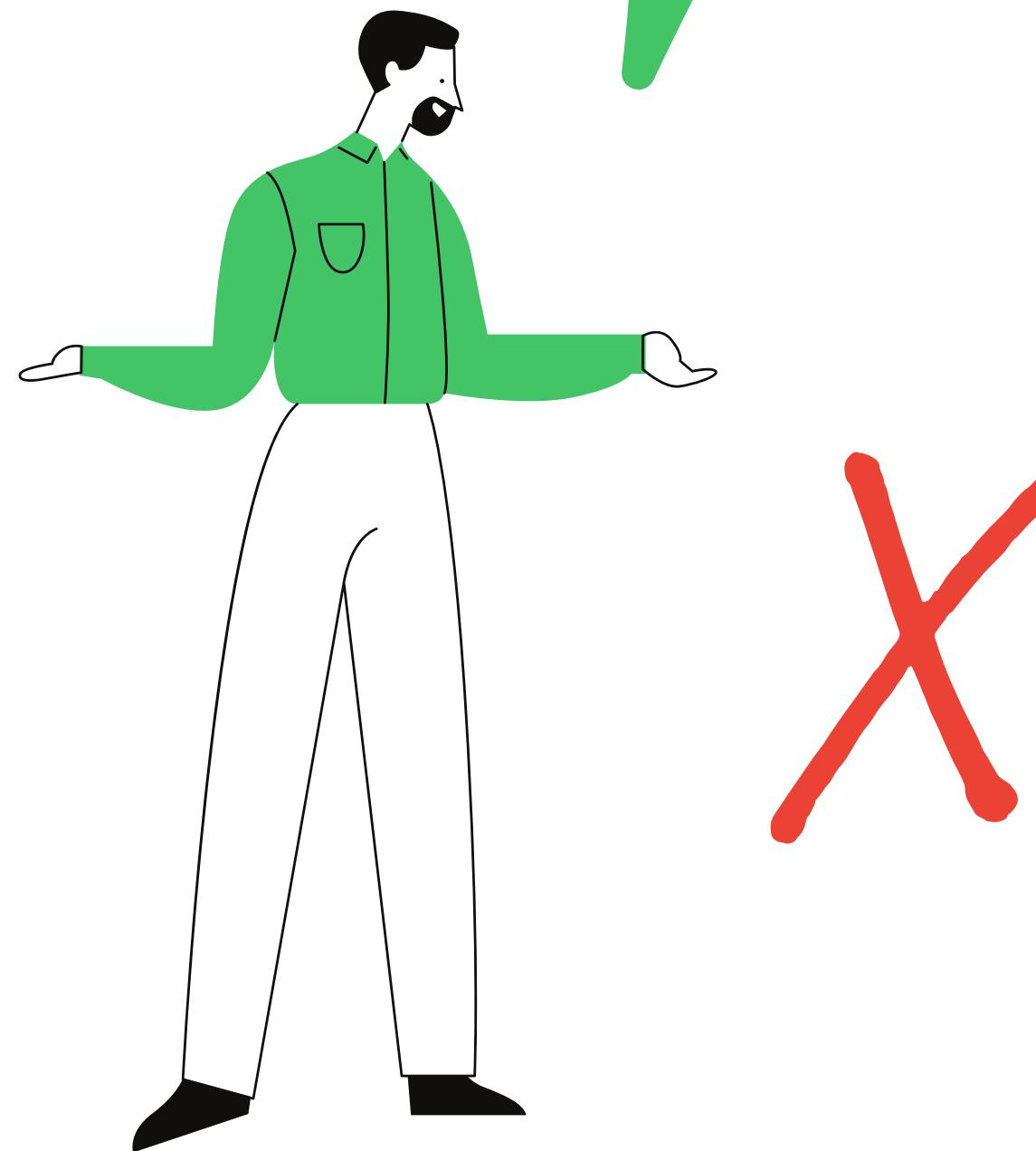


(a)

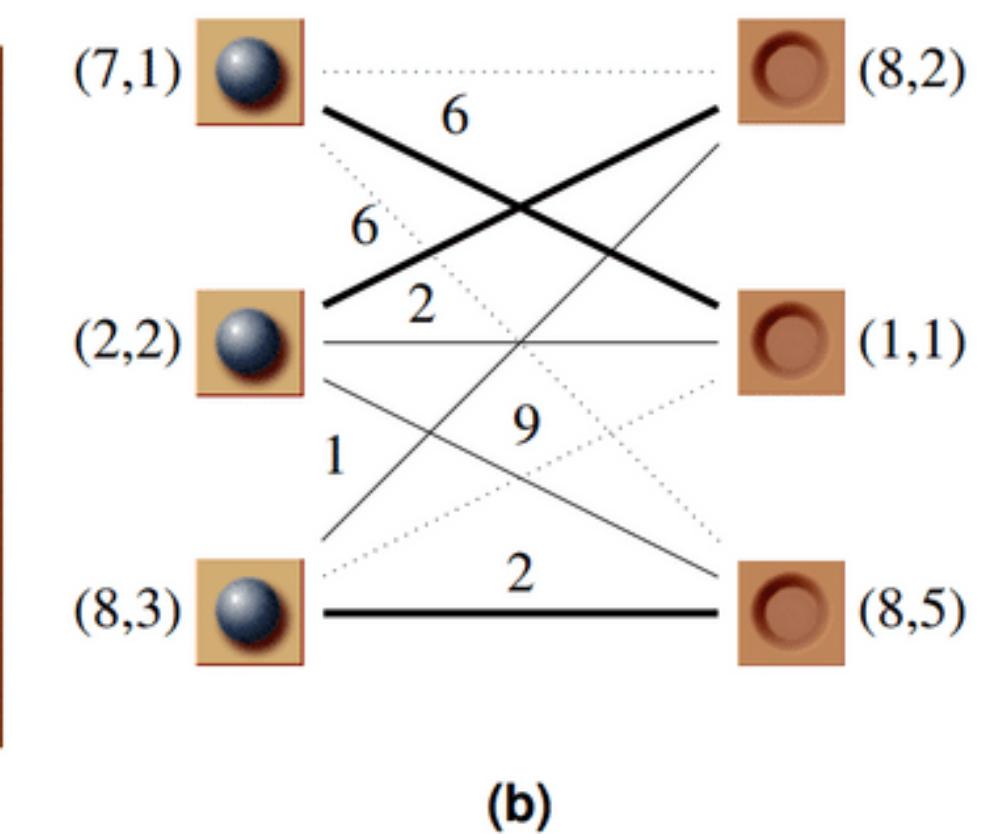


(b)

Một đích chỉ được chứa một hộp và một hộp
chỉ nằm trong 1 đích, làm sao để gán các đích
vào các hộp một cách tối ưu nhất?

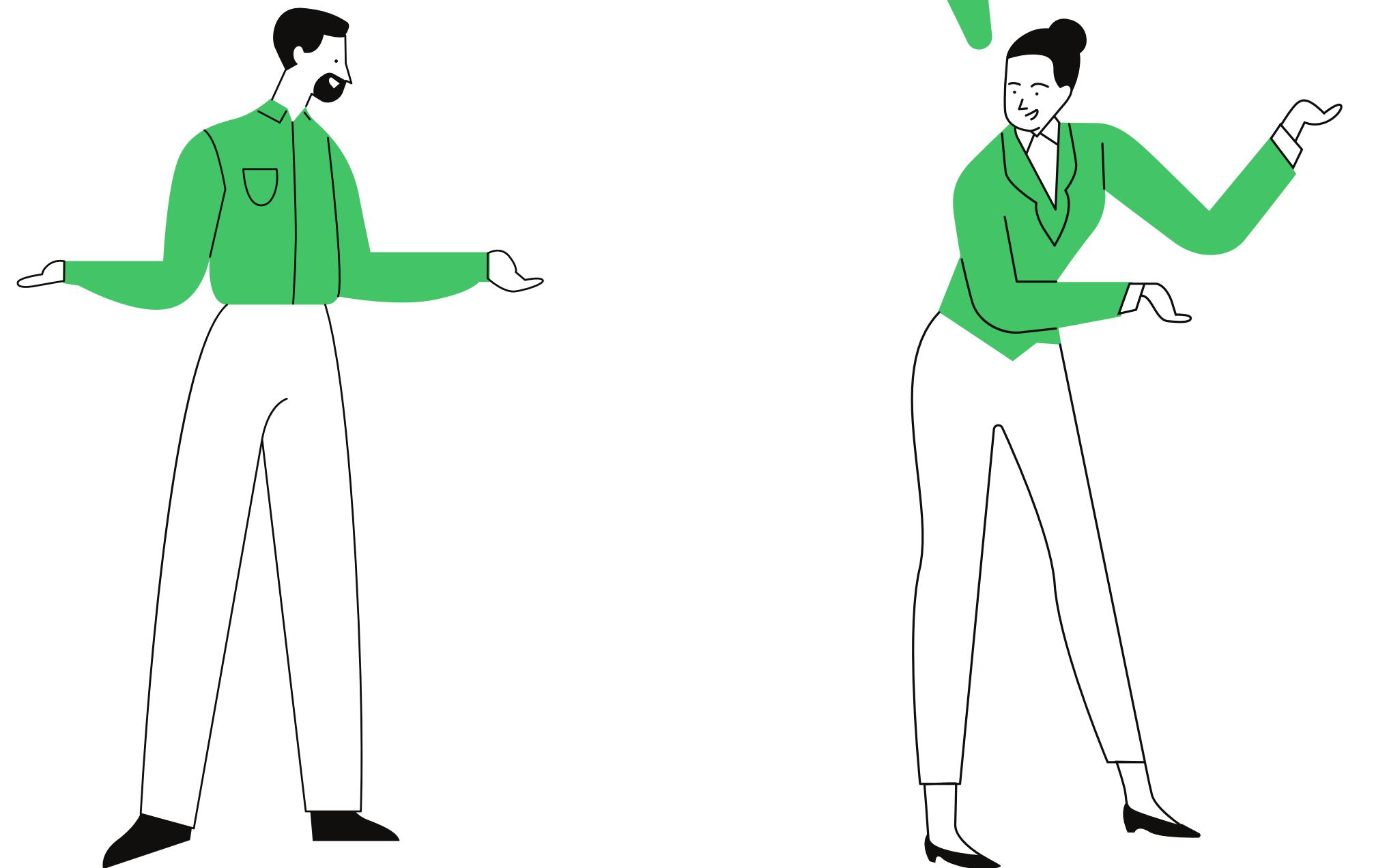


(a)

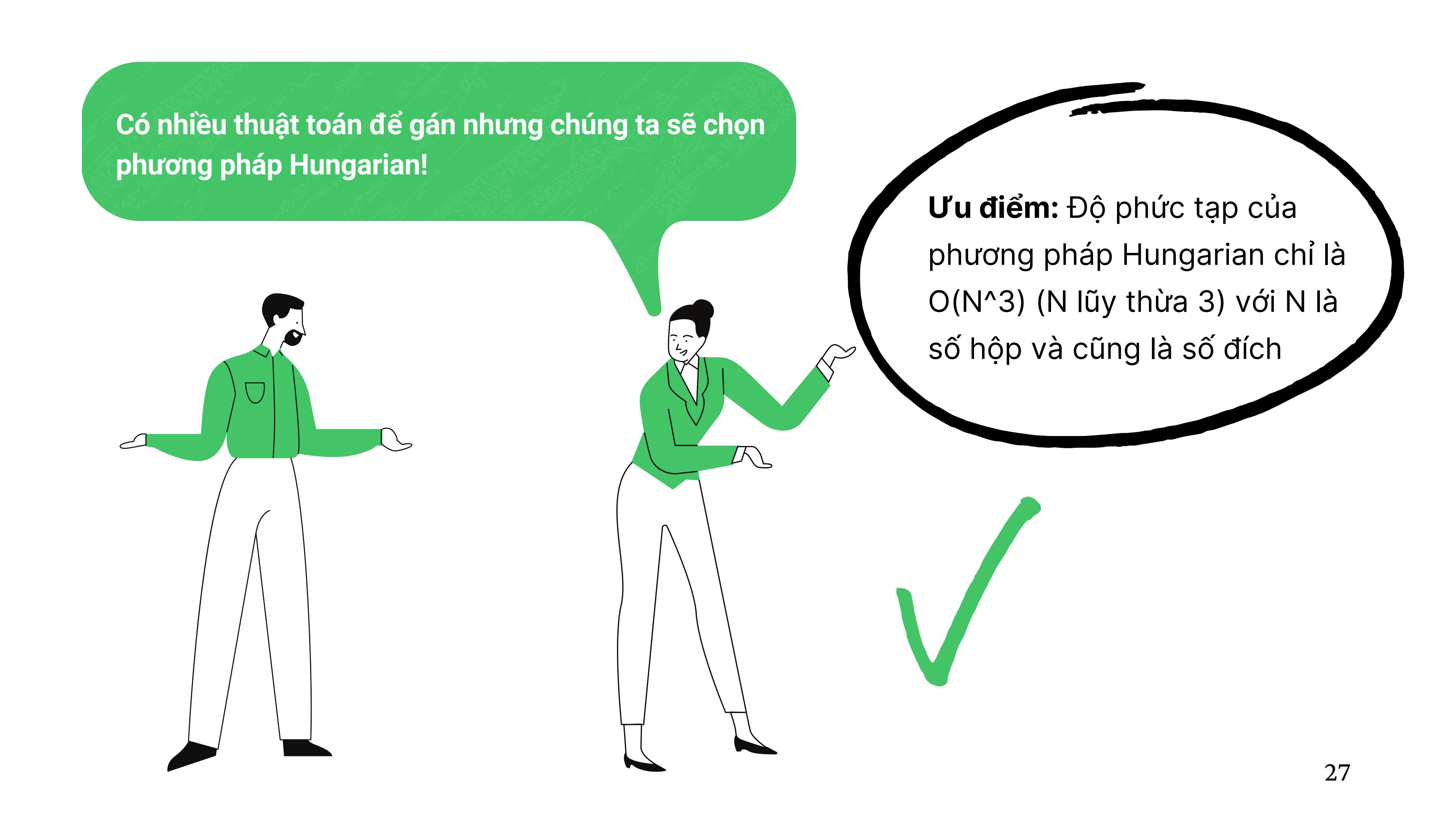


(b)

**Chúng ta sẽ gán các hộp vào các đích sao cho tổng
khoảng cách từ các hộp đến các đích là nhỏ nhất!**







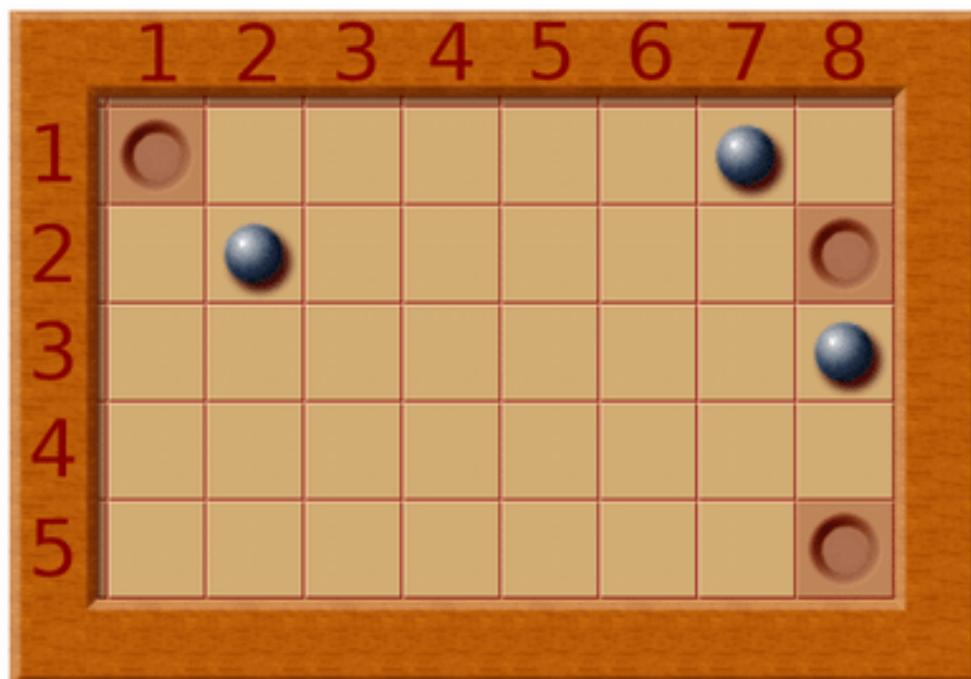
Có nhiều thuật toán để gán nhưng chúng ta sẽ chọn phương pháp Hungarian!

Ưu điểm: Độ phức tạp của phương pháp Hungarian chỉ là $O(N^3)$ (N lũy thừa 3) với N là số hộp và cũng là số đích

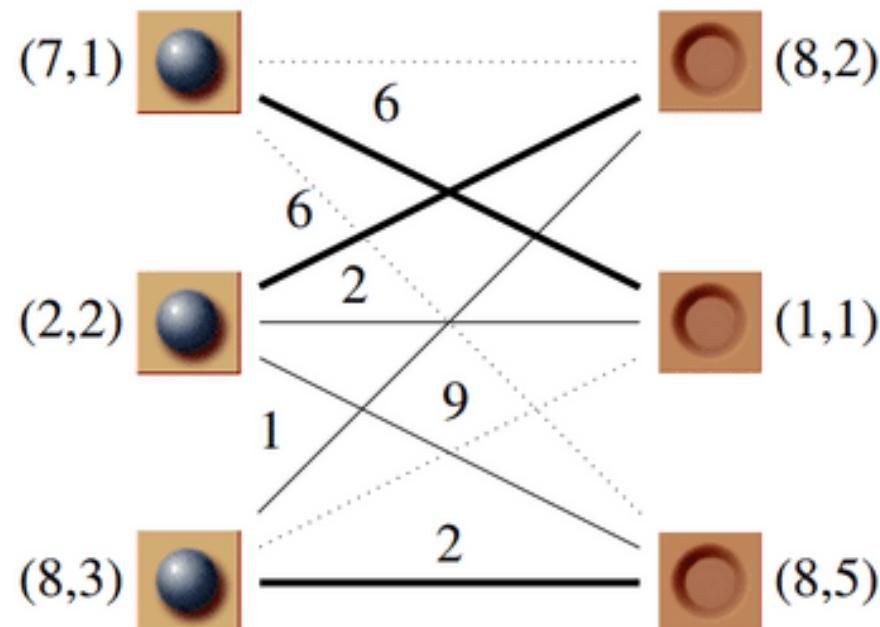


Thuật toán Hungarian

	Đích (8,2)	Đích (1,1)	Đích (8,5)
Hộp (7,1)	INF	6	INF
Hộp (2,2)	6	2	9
Hộp (8,3)	1	INF	2



(a)



(b)

```
>>> import numpy as np
>>> import math
>>> from scipy.optimize import linear_sum_assignment
>>> cost = np.array([[math.inf, 6, math.inf], [6, 2, 9], [1, math.inf, 2]])
>>> row_ind, col_ind = linear_sum_assignment(cost)
>>> col_ind
array([1, 0, 2], dtype=int64)
>>> cost[row_ind, col_ind].sum()
14.0
```

Chúng ta có thể dễ dàng
sử dụng phương pháp
Hungarian từ thư viện
scipy của python

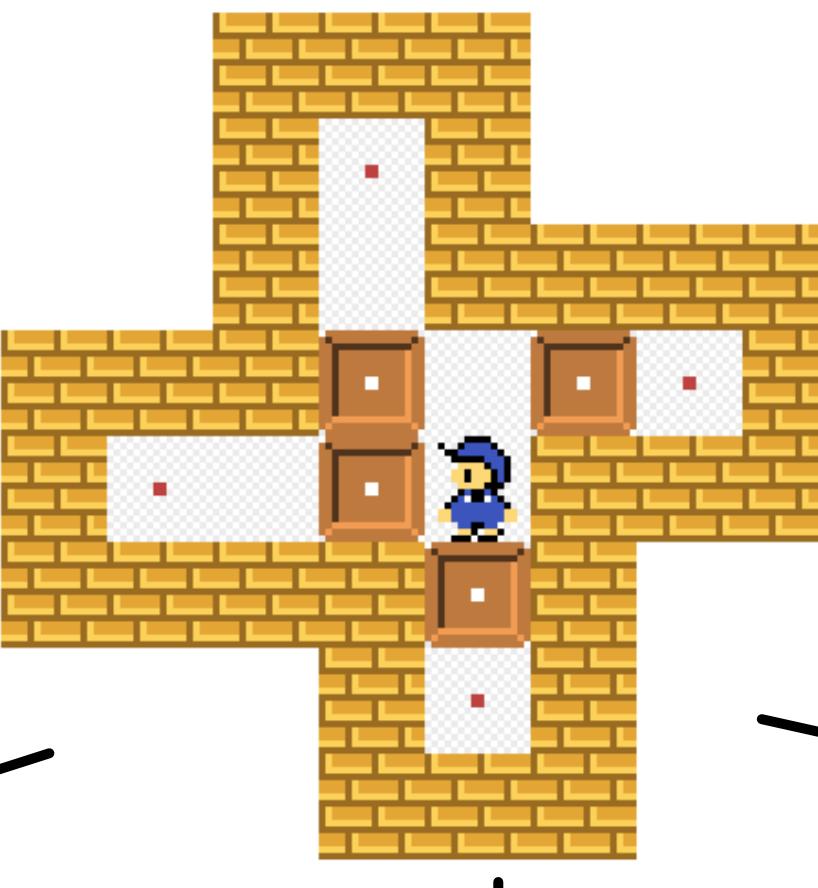
Xây dựng hàm h và g cho thuật toán A*

Trong bài toán này, chúng em xây dựng hàm **g** chính là số bước mà người chơi đã đi được kể từ trạng thái khởi đầu và hàm **h** chính là tổng khoảng cách nhỏ nhất từ các hộp đến các đích (sử dụng phương pháp "Hungarian"). Khi đó, hàm chi phí sẽ là:

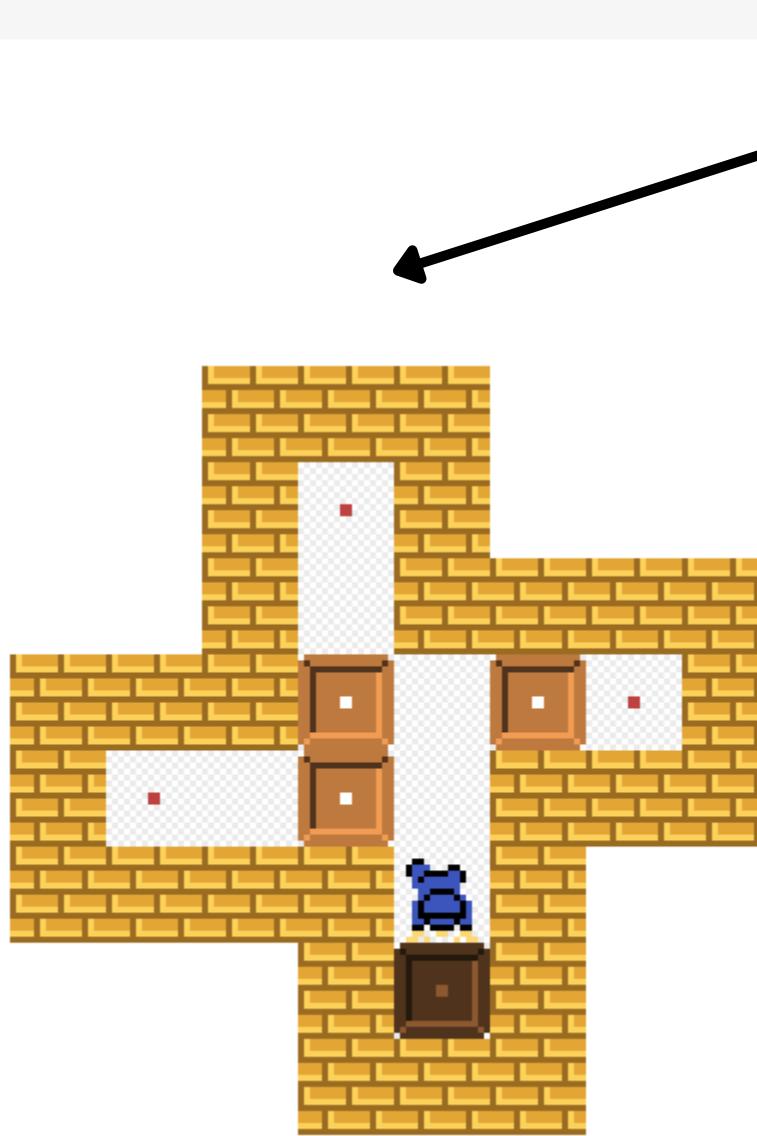
$$f = g + h.$$

Chúng em sẽ lựa chọn bước đi tiếp theo dựa vào hàm chi phí này, nếu trạng thái nào có giá trị hàm chi phí thấp nhất thì sẽ được ưu tiên lựa chọn để duyệt trước. Nếu có nhiều hơn một trạng thái có chi phí thấp nhất thì thứ tự ưu tiên sẽ dựa vào thứ tự đến Priority queue sớm hơn. Ở đây, từ một trạng thái ban đầu (đỉnh ban đầu) sẽ có tối đa 4 trạng thái kế tiếp (đỉnh con) tương ứng với 4 hướng di chuyển của người chơi. Giá trị hàm **h** chỉ thay đổi khi vị trí của các hộp có sự thay đổi.

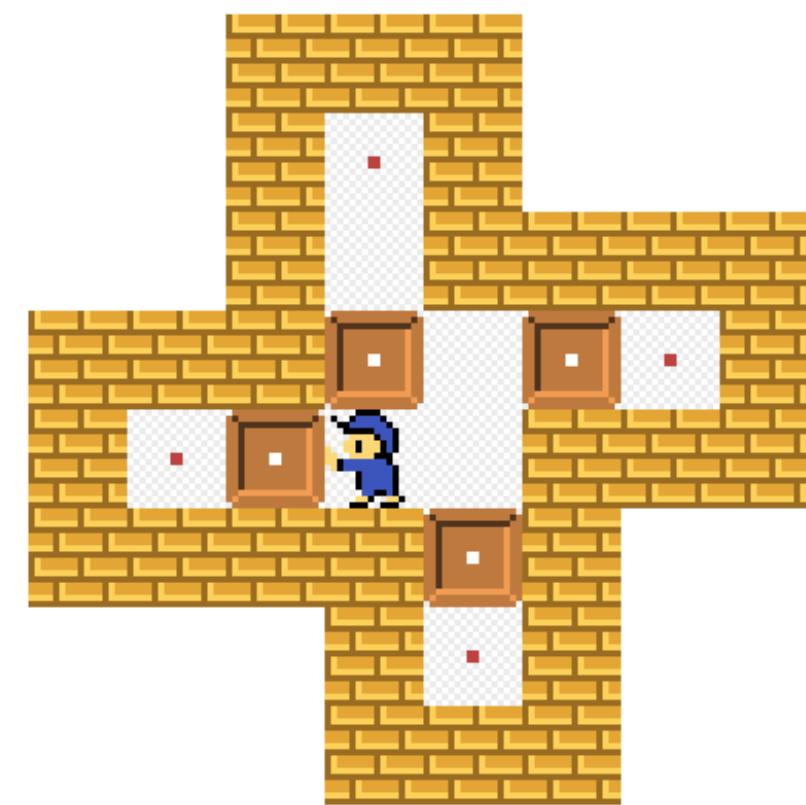
Mỗi lần tính hàm **h** sẽ mất khoảng thời gian là $O(n^3)$ (n là số hộp). Do đó, thời gian để thực hiện 1 bước di chuyển tiếp theo từ trạng thái hiện tại sẽ là $O(n^3)$



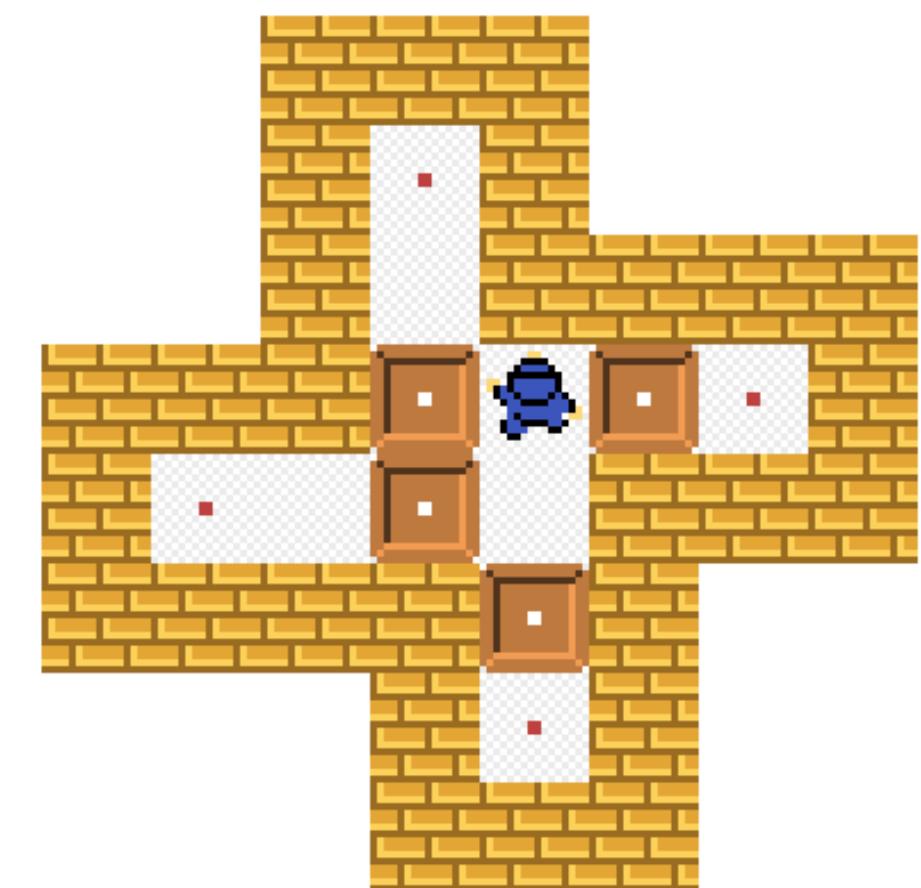
trạng thái bắt đầu:
 $g = 0, h = 6, f = 6$



$g = 1, h = 5, f = 6$

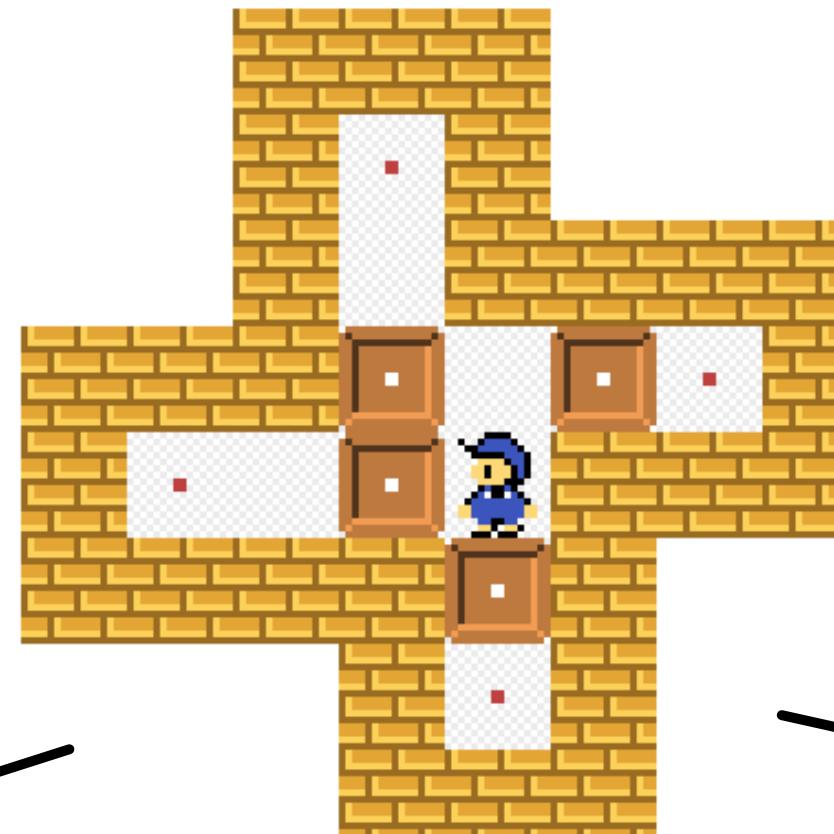


$g = 1, h = 5, f = 6$

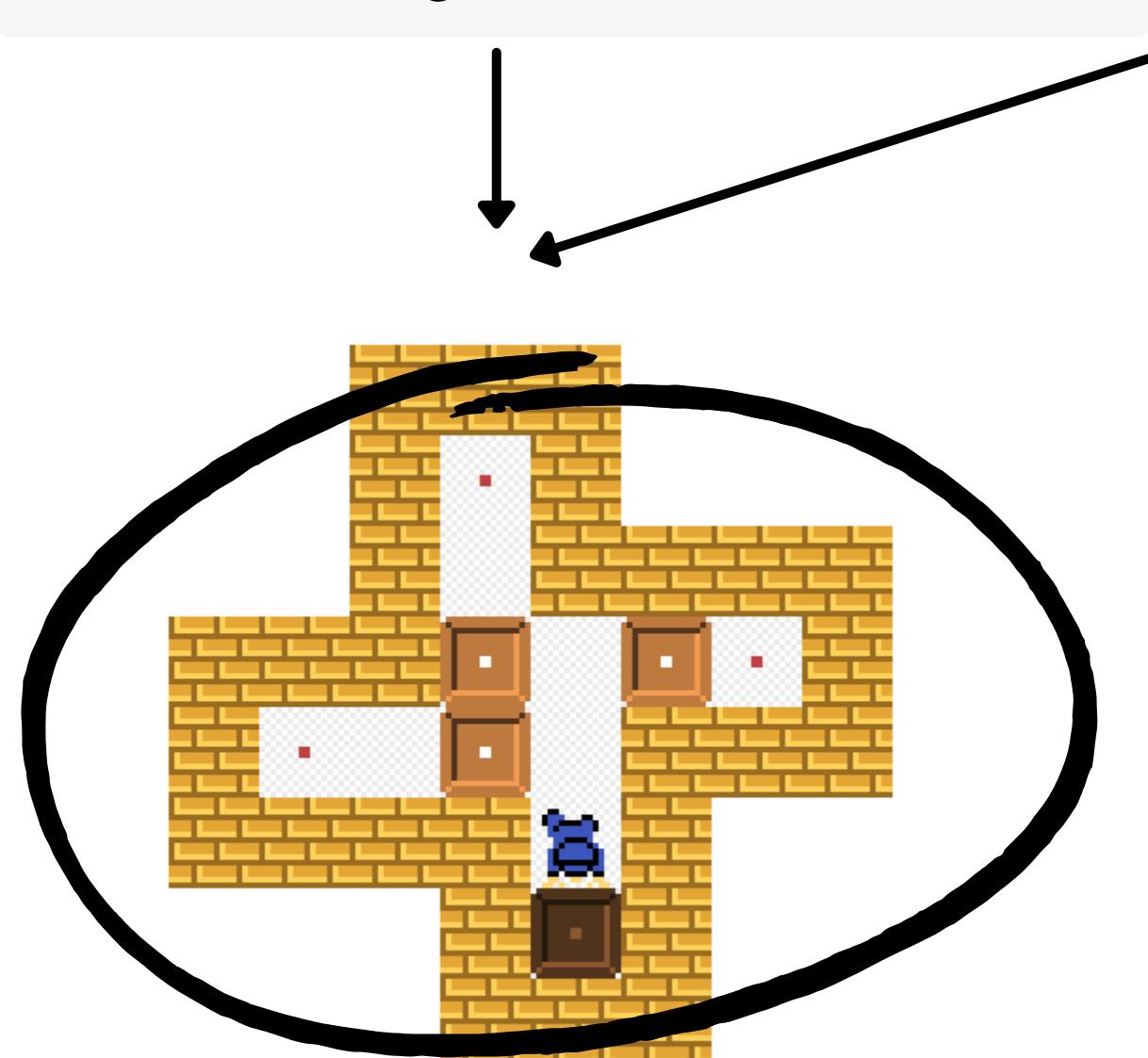


$g = 1, h = 6, f = 7$

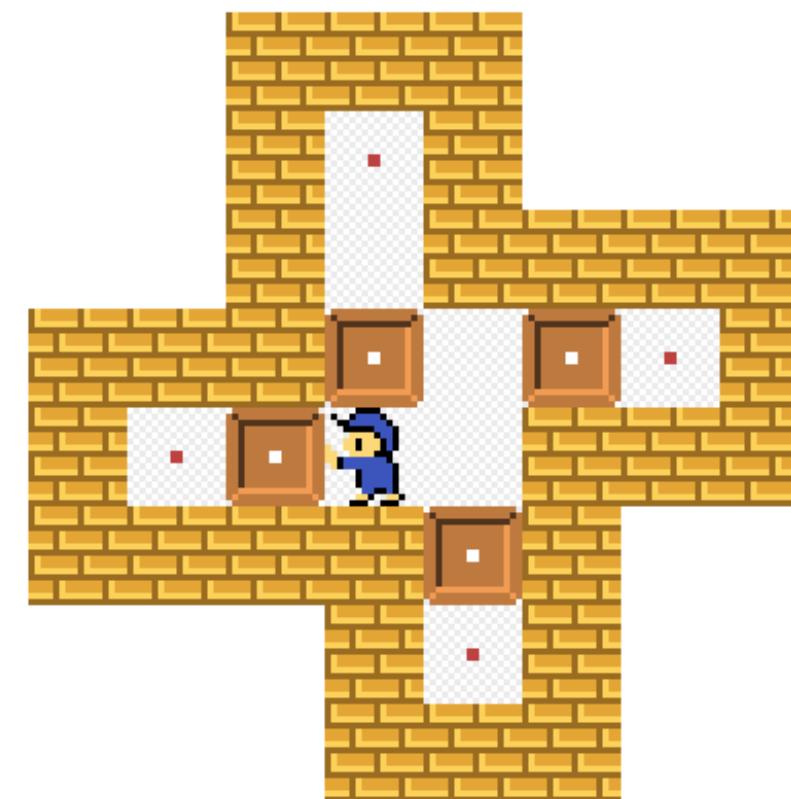
Giả sử người chơi đi xuống dưới trước, khi đó, trạng thái tiếp theo sẽ là trạng thái dưới đây vì nó là một trong các trạng thái có hàm chi phí (f) nhỏ nhất và đi vào hàng đợi ưu tiên trước



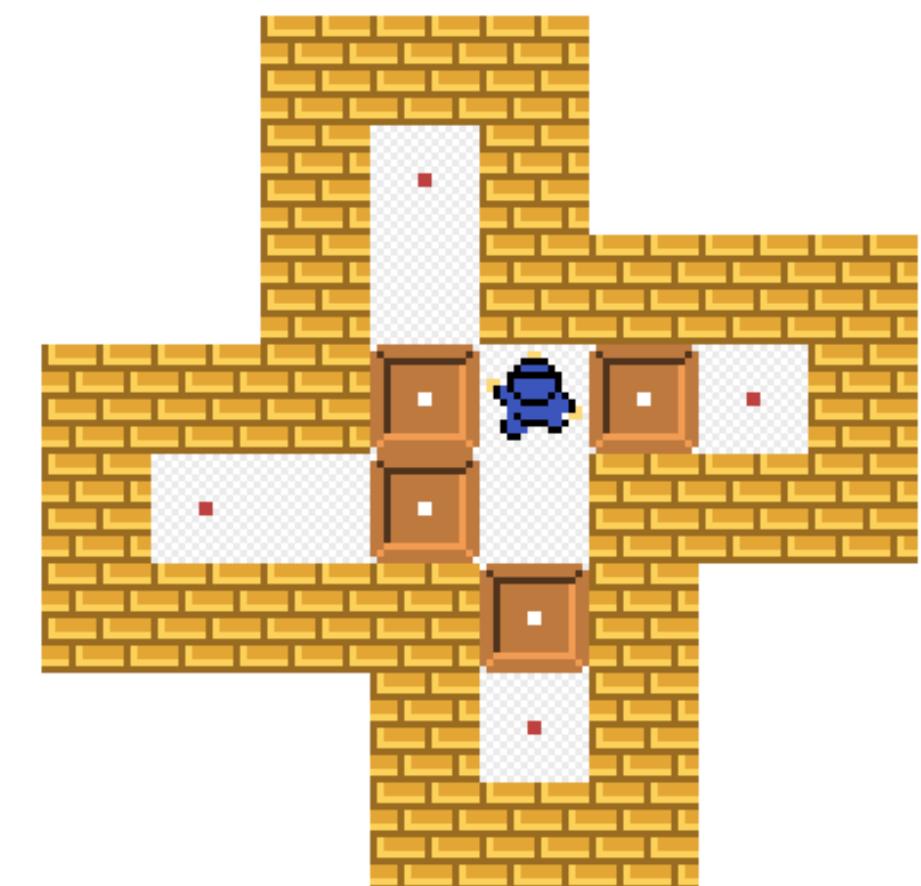
trạng thái bắt đầu:
 $g = 0, h = 6, f = 6$



$g = 1, h = 5, f = 6$



$g = 1, h = 5, f = 6$



$g = 1, h = 6, f = 7$

Mã giả

Input: rootVertex

Result: Path to a solution vertex

Begin

```
visited ← Ø // Set of visited vertices
rootVertex.g = 0
visited.add(rootVertex)
priority_queue.push(rootVertex) // Priority queue of value f of vertices with rootVertex as the only element
while priority_queue ≠ Ø do
    vertex = priority_queue.pop() // Pop the element having minimum f
    foreach successor of vertex do
        if isSolution(successor) then
            return pathTo(successor)
        if not visited.contains(successor) then
            successor.g = vertex.g + 1
            successor.h = Minimum distance from boxes to goals using Hungarian method
            successor.f = successor.g + successor.h
            priority_queue.push(successor)
            visited.add(successor)
    return no solution found.
```

End

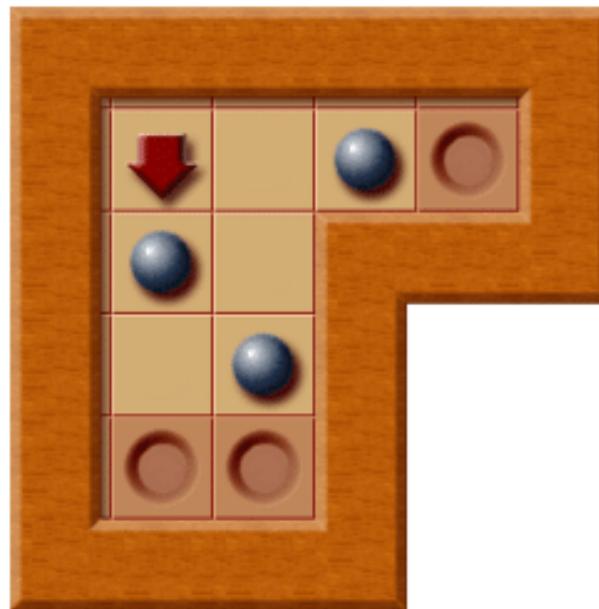


Một số giải pháp nhóm đề xuất để tối ưu hóa lời giải

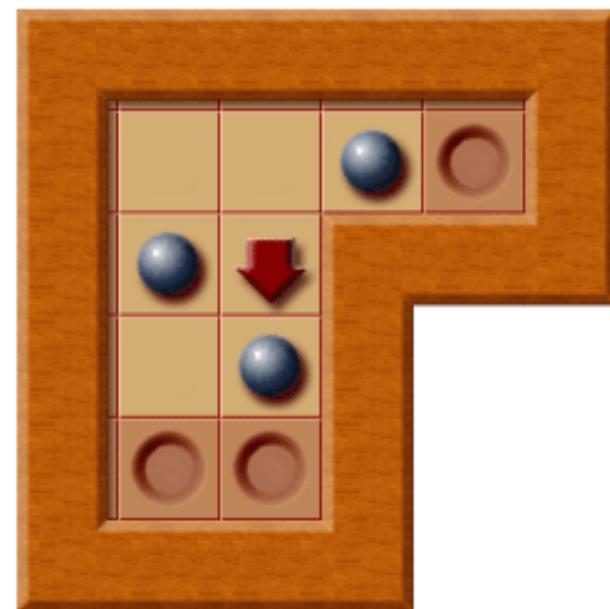
Tối ưu hóa không gian trạng thái

Để giảm kích thước của không gian trạng thái, trước tiên, chúng ta sẽ nới lỏng định nghĩa về một trạng thái ở Sokoban. Hai trạng thái được xem là giống nhau, nếu các vị trí của các hộp giống nhau và người chơi có thể di chuyển từ vị trí mà người chơi đang đứng ở trạng thái này sang vị trí mà người chơi đang đứng ở trạng thái khác mà không cần di chuyển bất cứ hộp nào.

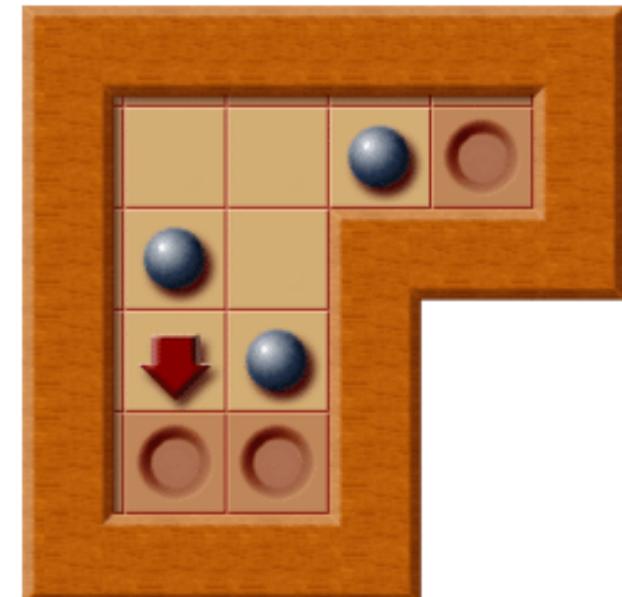
Việc giải Sokoban lúc này là tìm đường đi từ trạng thái đầu đến trạng thái cuối cùng bằng cách di chuyển các hộp, kèm theo một số ràng buộc. Mỗi lần hộp bị di chuyển thì chúng ta luôn xác định được vị trí của người chơi để thực hiện hành động di chuyển đó. Sau khi có được một chuỗi các trạng thái di chuyển các hộp để dẫn đến kết quả, ta sẽ chèn người chơi ở giữa mỗi 2 trạng thái và áp dụng thuật toán tìm đường đi như là Dijkstra để tìm ra được các bước di chuyển của người chơi để đạt được trạng thái mong muốn.



(a)

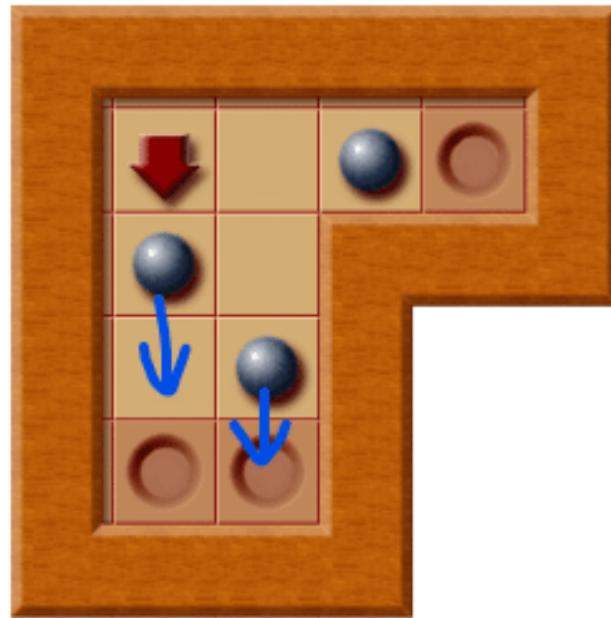


(b)

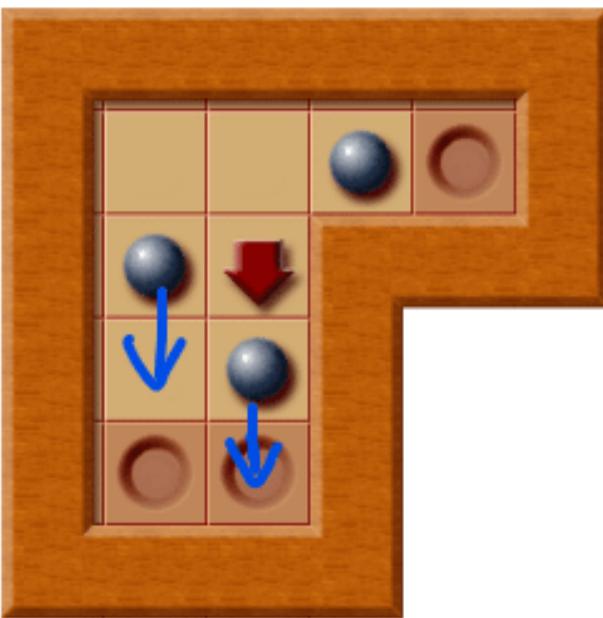


(c)

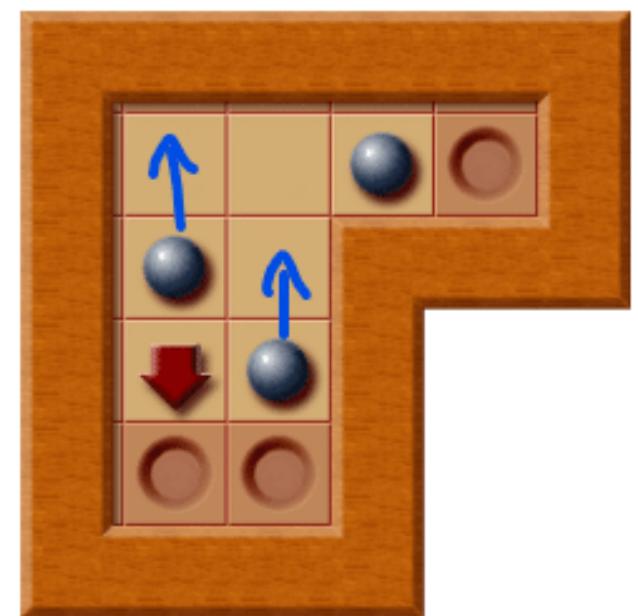
Hình (a) và (b) cho thấy cùng một trạng thái. Người chơi có thể di chuyển từ vị trí của mình trong hình (a) đến vị trí của mình trong hình (b) mà không cần di chuyển bất cứ hộp nào.
Hình (c) mô tả một trạng thái khác.



(a)

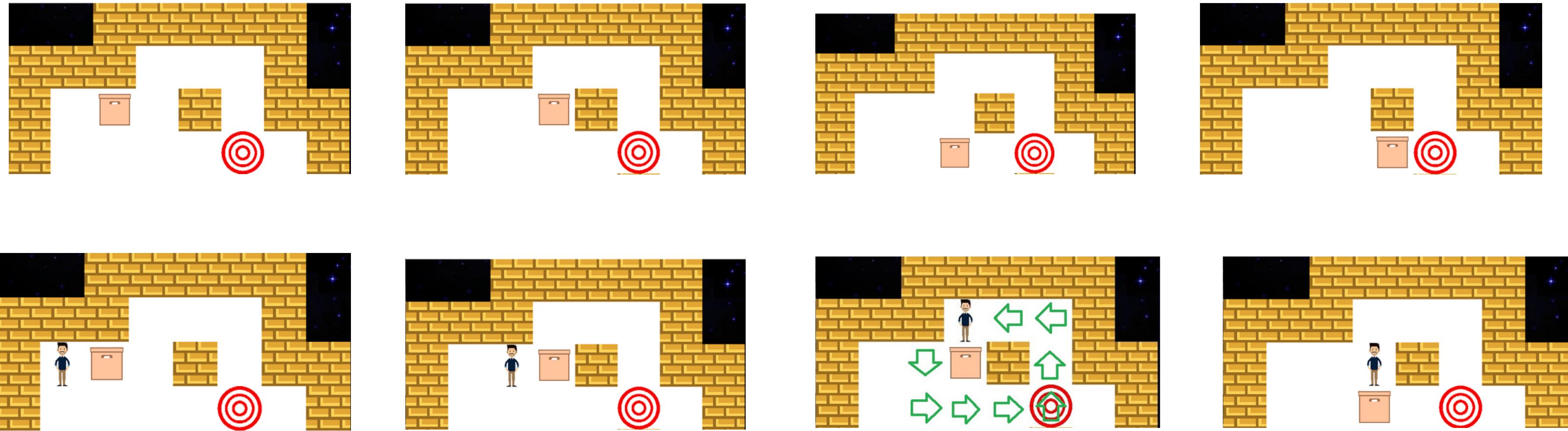


(b)



(c)

Ràng buộc: Giả sử 2 chiếc hộp nằm bên dưới chia không gian trò chơi thành 2 phần, nếu người chơi đang ở phần không gian bên trên (trong hình (a) và (b)) thì chỉ có thể đẩy một trong 2 hộp đó xuống dưới. Còn người chơi đang ở phần không gian bên dưới thì chỉ có thể đẩy một trong 2 hộp đó lên trên.



Sau khi có được chuỗi di chuyển của các hộp thì ta dễ dàng suy ra được các bước di chuyển của người chơi

Nhận xét kết quả

Thông số

Thông số máy chạy: Windows 10, 1.6 GHz, 4 cores, 8 Logical Processors, 8GB RAM.

Kết quả chạy được lưu ở file excel dưới đường link sau:

https://github.com/nh0znoisung/Sokoban/blob/main/all_test.xlsx

KẾT QUẢ CHẠY TRÊN MINI COSMOS

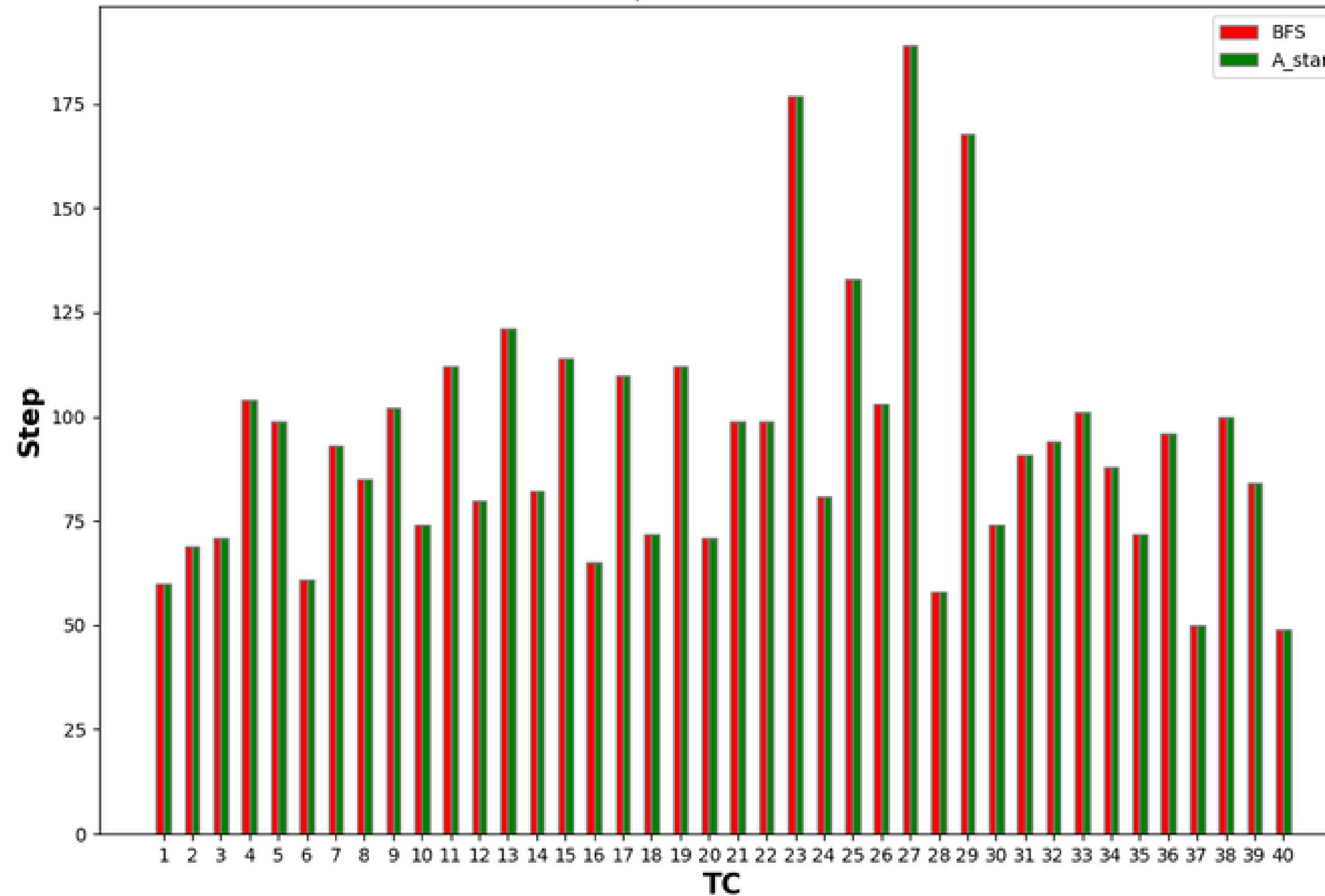
Map	Level	Boxes	Paths	BFS				A*			
				Node generated	Step	Time (s)	Memory (MB)	Node generated	Step	Time (s)	Memory (MB)
MINI COSMOS	1	1	24	378	37	0.000999	0.003906	367	37	0.007675	0.125
MINI COSMOS	2	2	24	1445	60	0.004987	0.128906	1383	60	0.013999	0.296875
MINI COSMOS	3	3	24	2658	69	0.011967	0.402344	2428	69	0.025967	0.421875
MINI COSMOS	4	1	28	893	71	0.00299	0.402344	893	71	0.00802	0.425781
MINI COSMOS	5	2	28	5830	104	0.021941	0.683594	5618	104	0.052204	0.824219
MINI COSMOS	6	3	28	17258	99	0.074799	2.070312	16432	99	0.153555	2.160156
MINI COSMOS	7	2	25	4520	61	0.016953	2.074219	4411	61	0.041814	2.222656
MINI COSMOS	8	3	25	13791	93	0.054891	2.113281	12498	93	0.13133	2.355469
MINI COSMOS	9	2	23	2321	85	0.008951	2.113281	2291	85	0.020944	1.726562
MINI COSMOS	10	3	23	6555	102	0.032913	2.113281	6476	102	0.06682	1.742188
MINI COSMOS	11	2	26	5321	74	0.018951	2.128906	5318	74	0.0483	1.75
MINI COSMOS	12	3	26	28398	112	0.123736	3.136719	28239	112	0.326848	3.1875
MINI COSMOS	13	2	29	13616	80	0.061835	3.160156	13219	80	0.132242	3.25
MINI COSMOS	14	3	29	92405	121	0.384435	10.632812	91166	121	1.03096	10.535156
MINI COSMOS	15	2	24	4945	82	0.019947	8.628906	4939	82	0.04501	7.699219
MINI COSMOS	16	3	24	13018	114	0.061074	8.628906	12398	114	0.124102	8.269531
MINI COSMOS	17	2	28	3381	65	0.018949	8.628906	3033	65	0.02497	8.269531
MINI COSMOS	18	3	28	10859	110	0.048869	8.628906	10802	110	0.096777	8.269531
MINI COSMOS	19	2	27	5722	72	0.039892	8.628906	5702	72	0.04691	8.269531
MINI COSMOS	20	3	27	17955	112	0.162563	8.628906	17847	112	0.177927	8.363281
MINI COSMOS	21	2	25	6192	71	0.059839	8.628906	6146	71	0.053888	8.363281
MINI COSMOS	22	3	25	14535	99	0.135641	8.628906	14291	99	0.138219	8.363281
MINI COSMOS	23	2	35	21941	99	0.082816	8.628906	21904	99	0.19847	8.363281
MINI COSMOS	24	3	35	166311	177	0.718547	15.359375	166240	177	1.871009	16.128906
MINI COSMOS	25	2	29	5917	81	0.022938	13.515625	5465	81	0.04618	14.128906
MINI COSMOS	26	3	29	24340	133	0.096815	13.960938	23664	133	0.236723	14.59375
MINI COSMOS	27	2	28	7338	103	0.025499	13.960938	7139	103	0.066156	14.734375
MINI COSMOS	28	3	28	28449	189	0.120418	13.960938	27957	189	0.265072	14.742188
MINI COSMOS	29	2	30	4976	58	0.019374	13.960938	4530	58	0.039892	14.742188
MINI COSMOS	30	3	30	40421	168	0.143022	13.988281	39150	168	0.507598	14.816406
MINI COSMOS	31	2	21	1239	74	0.006201	13.988281	1234	74	0.011464	14.816406
MINI COSMOS	32	2	19	1342	91	0.00698	13.988281	1342	91	0.012006	14.816406
MINI COSMOS	33	3	23	7444	94	0.028918	13.988281	5382	94	0.050899	14.816406
MINI COSMOS	34	3	23	10390	101	0.05489	13.988281	8860	101	0.083762	14.816406
MINI COSMOS	35	3	20	2119	88	0.004197	13.988281	2046	88	0.020977	14.816406
MINI COSMOS	36	3	21	4823	72	0.018951	13.988281	4470	72	0.045847	14.816406
MINI COSMOS	37	3	22	4914	96	0.020943	13.988281	4885	96	0.04991	14.816406
MINI COSMOS	38	3	21	5819	50	0.021982	13.988281	5244	50	0.046912	14.816406
MINI COSMOS	39	3	26	12557	100	0.047902	13.988281	11633	100	0.107843	14.816406
MINI COSMOS	40	3	21	4625	84	0.021507	13.738281	4290	84	0.0444	14.816406

KẾT QUẢ CHẠY TRÊN MICRO COSMOS

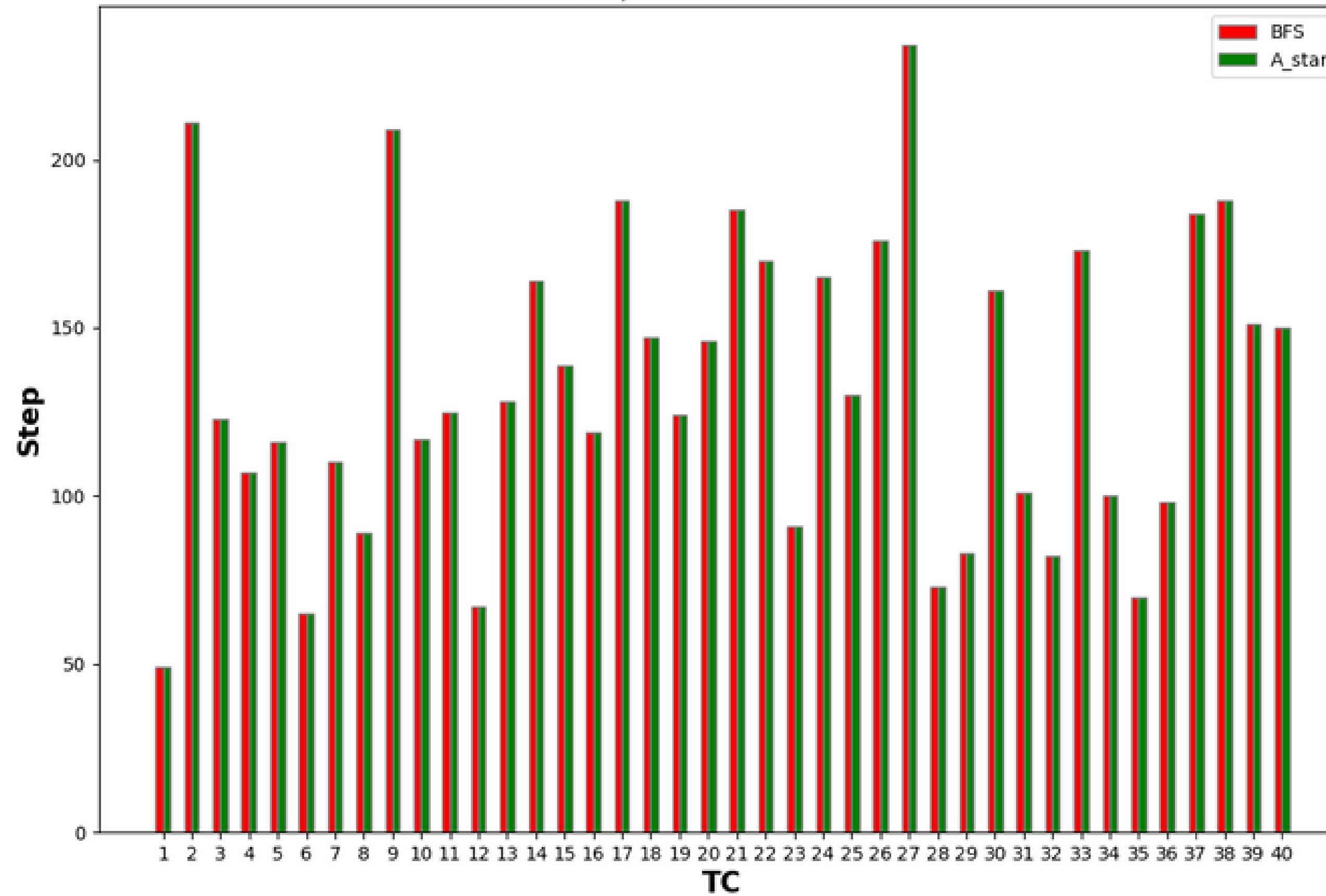
Map	Level	Boxes	Paths	BFS				A*			
				Node generated	Step	Time (s)	Memory (MB)	Node generated	Step	Time (s)	Memory (MB)
MICRO COSMOS	1	4	24	54490	49	0.218553	15.742188	43862	49	0.486557	14.832031
MICRO COSMOS	2	4	48	778499	211	3.688451	74.066406	717155	211	8.747407	75.375
MICRO COSMOS	3	3	28	16620	123	0.076793	62.308594	16161	123	0.178017	58.765625
MICRO COSMOS	4	4	21	84236	107	0.349231	59.15625	82775	107	0.962076	55.0625
MICRO COSMOS	5	6	33	1041482	116	5.517955	116.421875	1030515	116	14.47195	116.542969
MICRO COSMOS	6	5	25	163865	65	0.752206	96.21875	140618	65	1.916837	96.902344
MICRO COSMOS	7	3	23	9852	110	0.036937	70.554688	9711	110	0.097775	74.441406
MICRO COSMOS	8	4	24	53322	89	0.284922	53.628906	46879	89	0.577475	57.835938
MICRO COSMOS	9	4	28	287450	209	1.292639	50.519531	287401	209	3.190408	56.390625
MICRO COSMOS	10	3	28	28837	117	0.116411	33.035156	28826	117	0.26591	35.675781
MICRO COSMOS	11	5	26	156675	125	0.665126	33.316406	147113	125	2.102102	33.859375
MICRO COSMOS	12	6	24	108386	67	0.458727	31.570312	99863	67	1.42072	28.105469
MICRO COSMOS	13	4	24	19855	128	0.077828	27.242188	19050	128	0.20319	26.042969
MICRO COSMOS	14	4	29	153686	164	0.636049	29.039062	153646	164	1.734298	26.195312
MICRO COSMOS	15	4	33	361288	139	1.492759	37.070312	349713	139	4.873847	37.3125
MICRO COSMOS	16	4	30	366858	119	1.622701	36.957031	345740	119	4.580398	38.242188
MICRO COSMOS	17	4	35	900689	188	4.014299	94.542969	899554	188	10.24127	94.390625
MICRO COSMOS	18	4	28	113525	147	0.495136	81.109375	112656	147	1.20235	81.328125
MICRO COSMOS	19	6	30	927057	124	4.49333	101.695312	920444	124	12.44422	102.6875
MICRO COSMOS	20	4	34	106522	146	0.491814	86.324219	97802	146	1.147076	88.109375
MICRO COSMOS	21	5	29	145805	185	0.669202	81.472656	137704	185	1.744711	84.09375
MICRO COSMOS	22	4	36	1412851	170	6.725378	146.324219	1349365	170	16.48716	146.296875
MICRO COSMOS	23	4	22	17929	91	0.074839	122.957031	16108	91	0.206144	115.328125
MICRO COSMOS	24	5	34	580177	165	3.049147	92.390625	540534	165	7.946045	102.582031
MICRO COSMOS	25	4	28	201502	130	0.789764	68.632812	178895	130	2.435369	63.675781
MICRO COSMOS	26	5	38	663258	176	3.135278	71.289062	606099	176	7.550926	70.410156
MICRO COSMOS	27	4	30	116258	234	0.489151	61.402344	109952	234	1.445153	58.117188
MICRO COSMOS	28	3	20	4907	73	0.017603	57.652344	4392	73	0.042187	56.144531
MICRO COSMOS	29	4	21	8138	83	0.037901	55.042969	7880	83	0.076837	55.894531
MICRO COSMOS	30	4	31	91038	161	0.345324	53.109375	77155	161	0.925666	57.941406
MICRO COSMOS	31	4	27	12878	101	0.062834	45.378906	11795	101	0.125375	53.1875
MICRO COSMOS	32	4	27	20189	82	0.098735	36.878906	19184	82	0.218049	49.6875
MICRO COSMOS	33	4	26	77061	173	0.338044	34.136719	75388	173	0.945706	45.941406
MICRO COSMOS	34	4	25	57606	100	0.254102	32.136719	28059	100	0.308311	42.207031
MICRO COSMOS	35	5	22	34148	70	0.116529	26.632812	11556	70	0.121806	30.261719
MICRO COSMOS	36	4	23	59507	98	0.216192	26.636719	58902	98	0.579211	30.824219
MICRO COSMOS	37	4	30	181267	184	0.752281	25.898438	177564	184	2.21454	29.46875
MICRO COSMOS	38	4	31	171579	188	0.736604	24.421875	163826	188	1.90125	27.160156
MICRO COSMOS	39	4	24	32836	151	0.123392	21.167969	32532	151	0.333429	24.554688
MICRO COSMOS	40	3	26	19363	150	0.091801	19.667969	19123	150	0.183254	22.066406

BIỂU ĐỒ

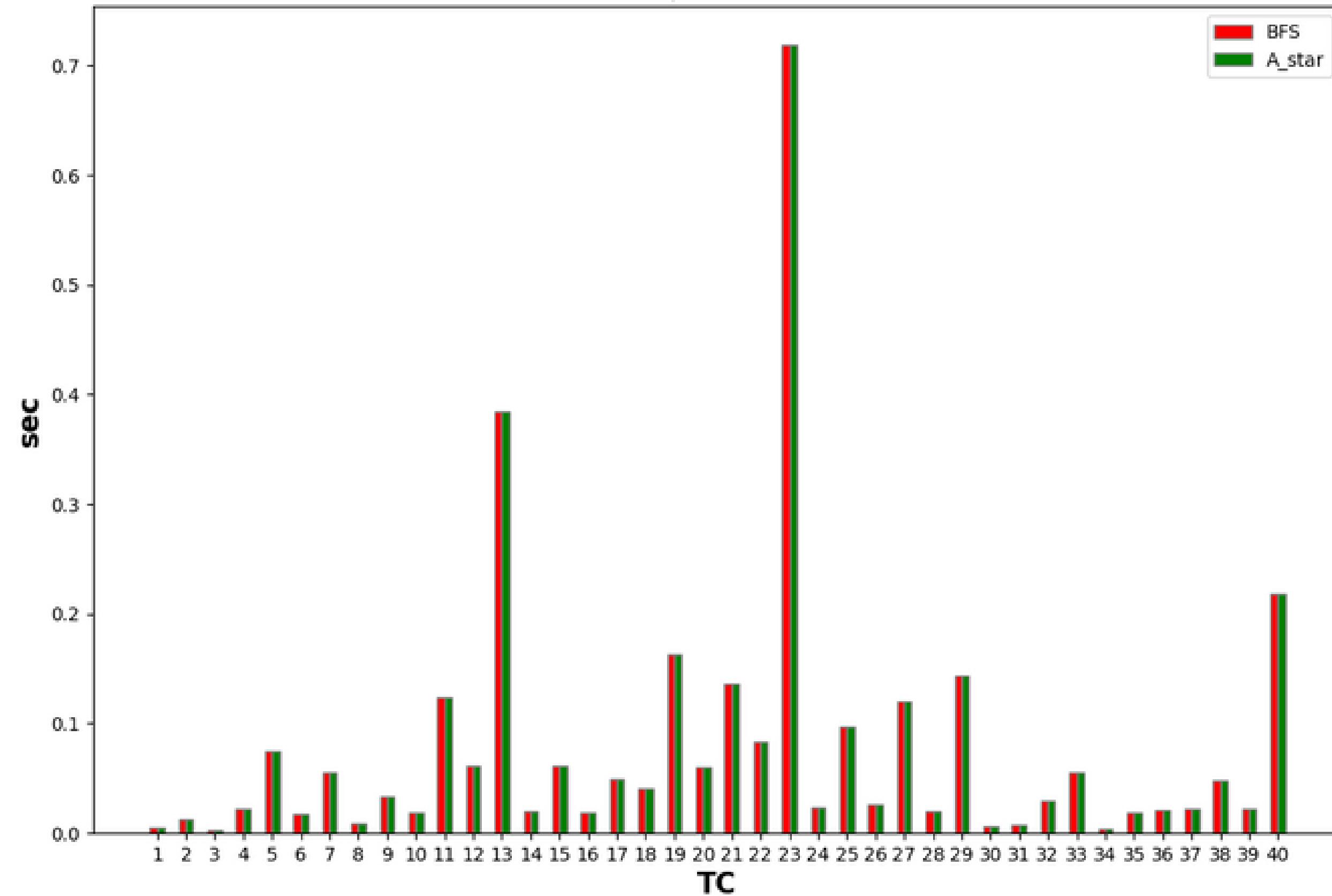
The number of Step taken in MINI COSMOS Testcases



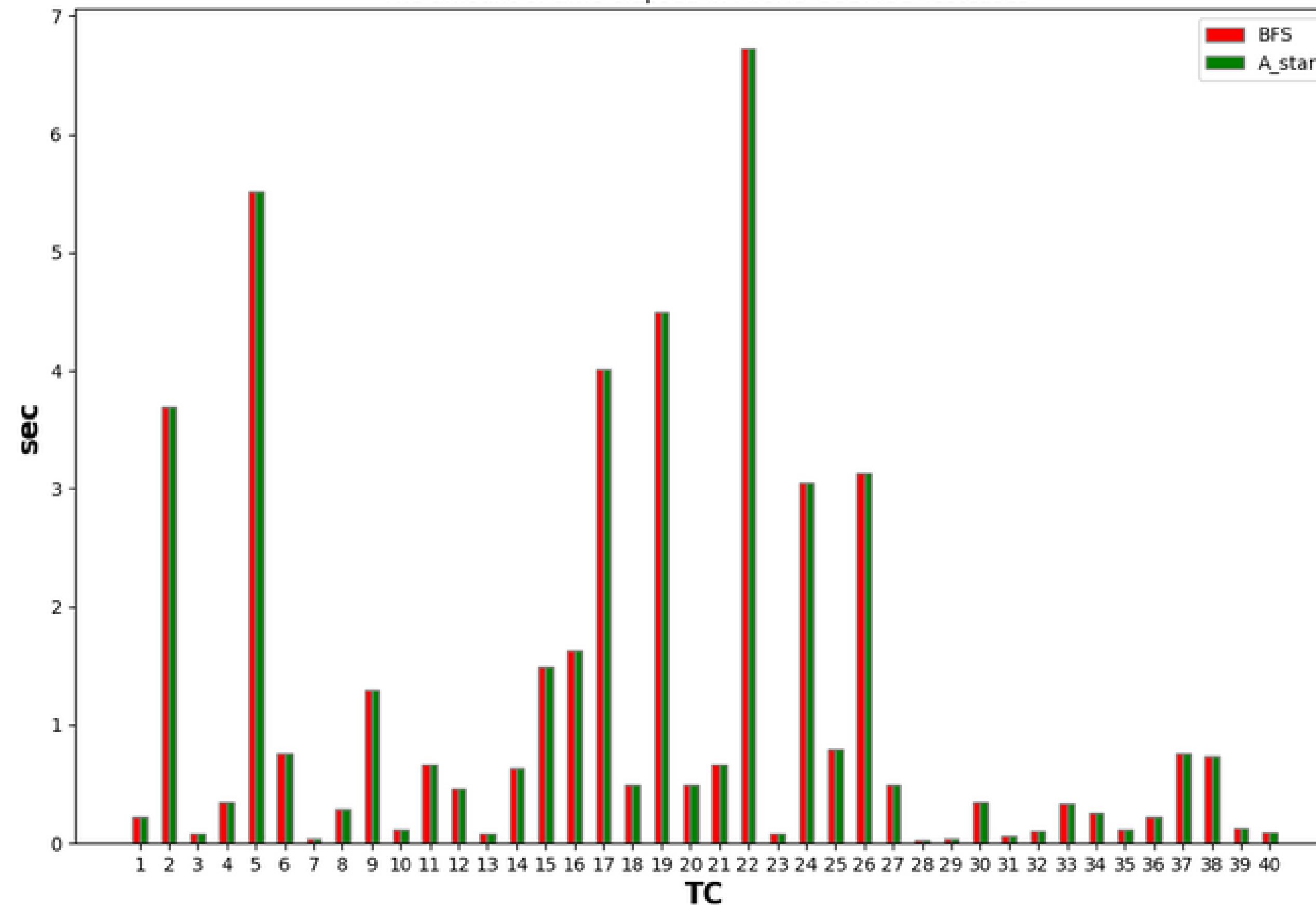
The number of Step taken in MICRO COSMOS Testcases



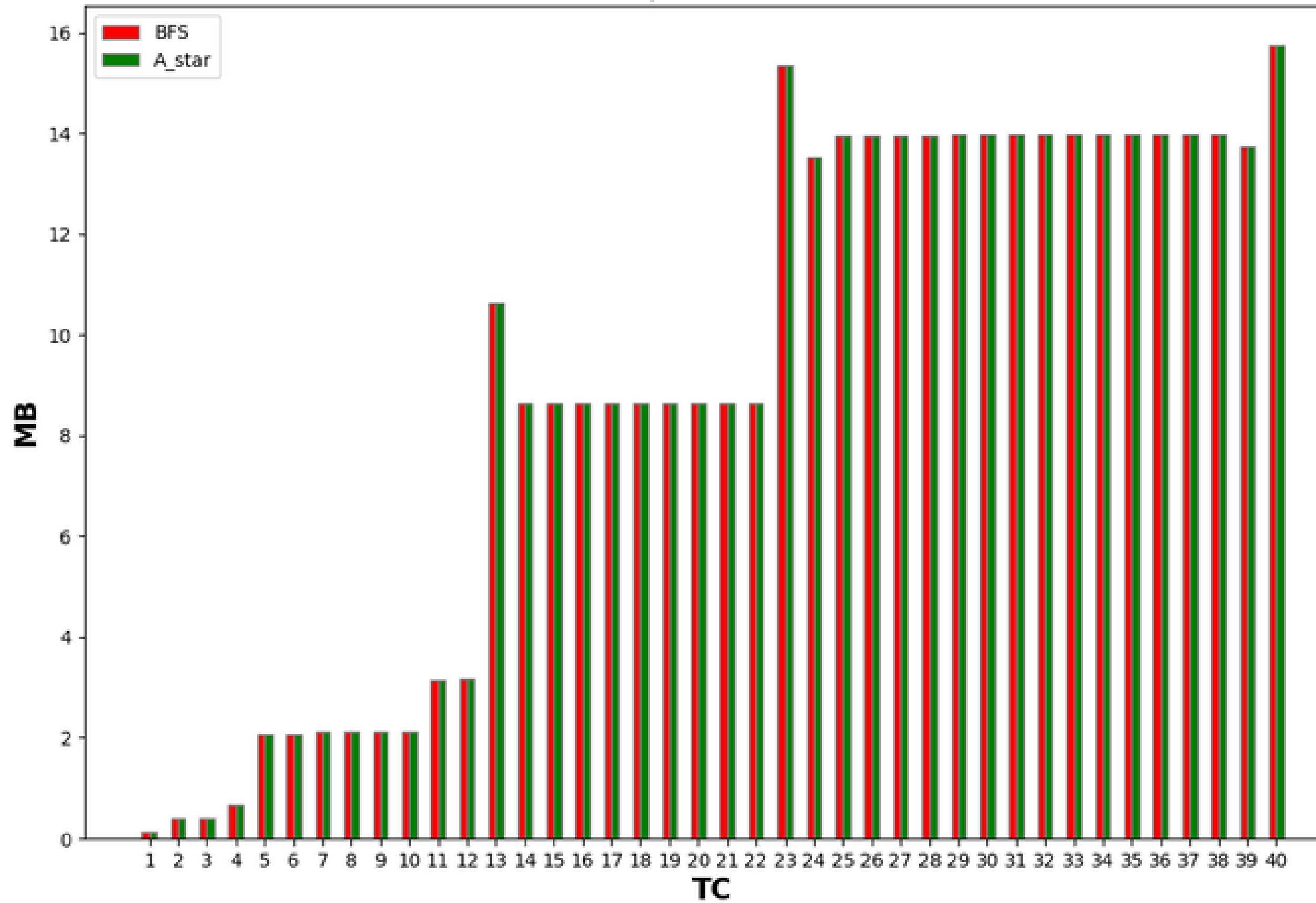
The amount of time elapsed in MINI COSMOS Testcases



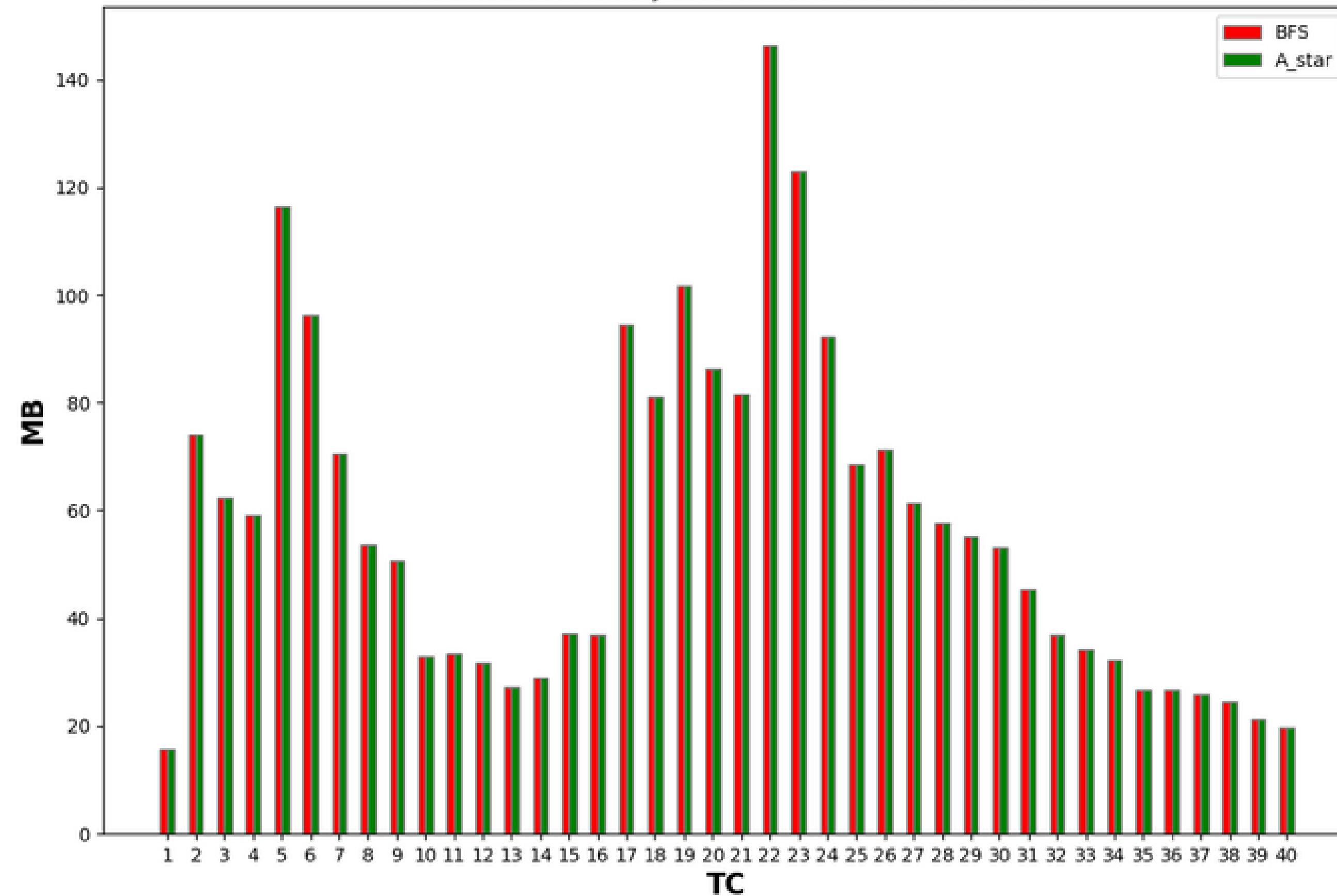
The amount of time elapsed in MICRO COSMOS Testcases



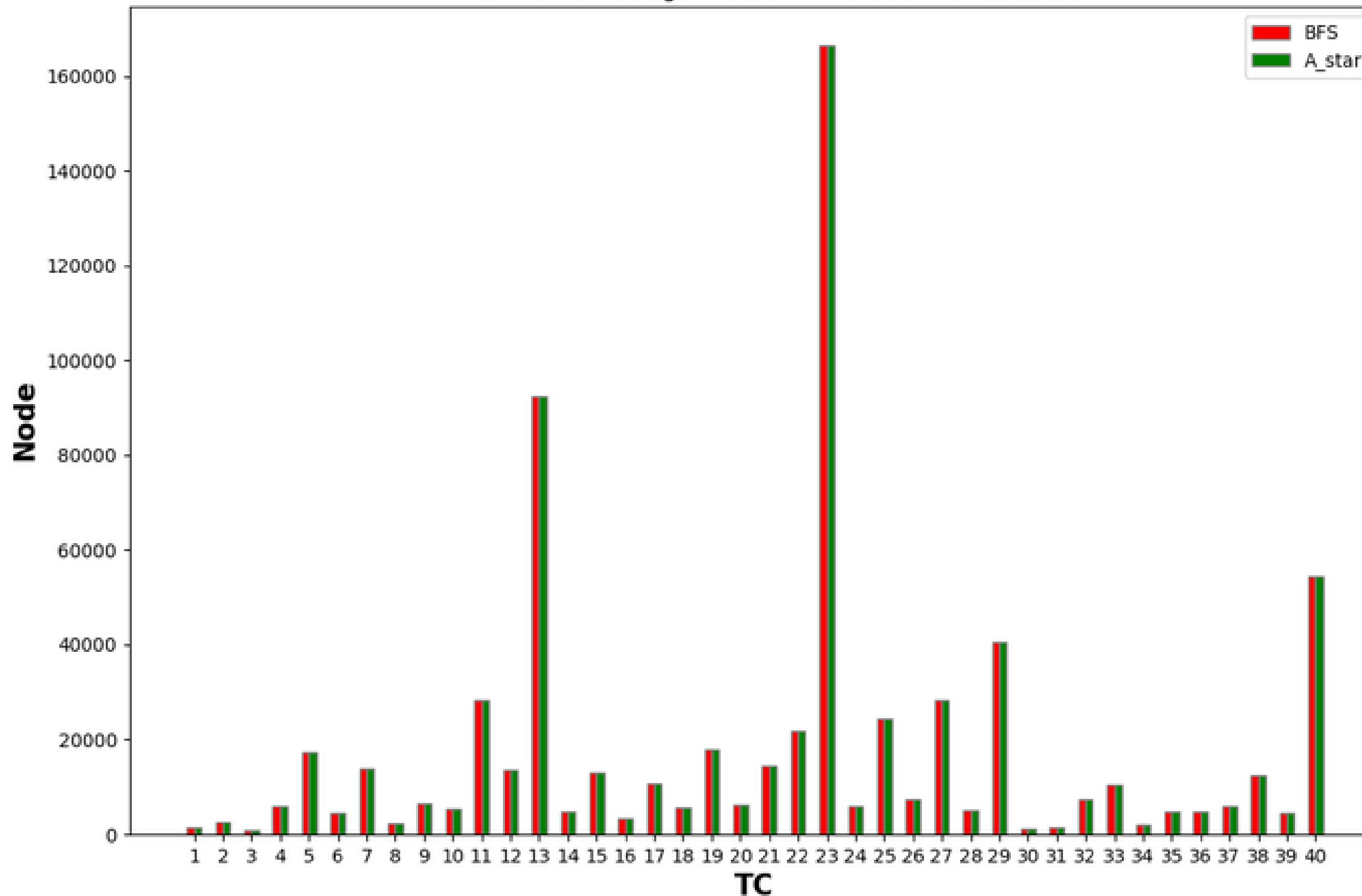
The amount of memory used in MINI COSMOS Testcases

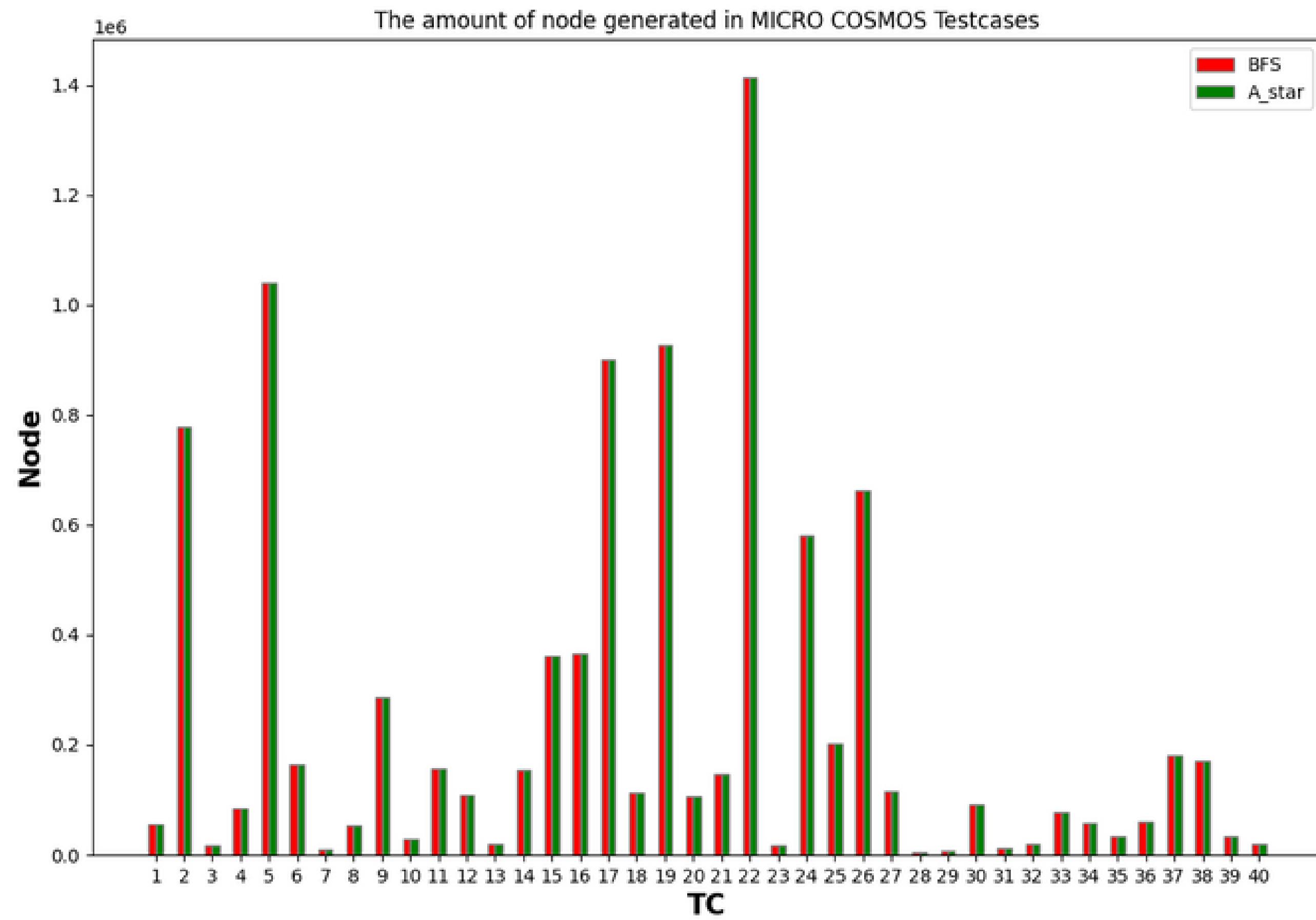


The amount of memory used in MICRO COSMOS Testcases



The amount of node generated in MINI COSMOS Testcases





PHÂN TÍCH KẾT QUẢ

Về mặt lí thuyết:

- **BFS** là thuật toán **vết cạn theo chiều ngang**. Điều này đảm bảo **kết quả tìm kiếm** được là đường đi **ngắn nhất** nhưng **số lượng Node sinh ra** sẽ **nhiều hơn**.
- Giải thuật **A*** tùy thuộc vào hàm **heuristic** nên **số lượng Node** sinh ra sẽ **ít hơn** nhưng **kết quả tìm kiếm** được là **khá tốt** không phải luôn luôn là **tốt nhất**.

PHÂN TÍCH KẾT QUẢ

Về mặt thực nghiệm:

- Sau khi thực thi 80 Testcase, **BFS** và **A*** đều cho ra **kết quả tìm kiếm** là như nhau và ngắn nhất.
- **Số lượng Node sinh ra** của **A*** ít hơn tương đối so với **BFS**.
- **Không gian bộ nhớ** được sử dụng giữa **BFS** và **A*** là chênh lệch không đáng kể.
- **Thời gian tìm kiếm trung bình** của **A*** gấp 3 lần so với **BFS**. (*)

=> **Nhìn chung**, giải thuật **BFS** hiệu quả hơn **A***.

	BFS	A*
Average node generated	128355.65	122335.3625
Average time	0.596687375	1.538857738
Average memory	33.01142571	34.01557611

PHÂN TÍCH KẾT QUẢ

Tại sao thời gian tìm kiếm trung bình của A* gấp 3 lần so với BFS?

- Thứ nhất, A* phải tính Cost của State hiện tại sau mỗi lần tạo ra Node mới. Tác vụ này sử dụng giải thuật Hungarian để matching các Boxes và các Goals với độ phức tạp về thời gian cho công việc này là $O(n^3)$ với n là số lượng boxes.
- Thứ hai, A* sử dụng Priority Queue (min-heap theo Cost của State) với độ phức tạp về thời gian cho các tác vụ thêm và xóa mất đến $O(\log(n))$ với n là số Node hiện tại trong Priority Queue. Còn với BFS chỉ sử dụng Queue với độ phức tạp về thời gian các tác vụ thêm và xóa chỉ mất $O(1)$.

PHÂN TÍCH KẾT QUẢ

Có tất cả **10/80 Testcase** mà cả 2 giải thuật **BFS** và **A*** đều giải **trên 1s**. Các **Testcase** này có đặc điểm chung là:

- Số **box** và **goal** cao (**Từ 4 đến 6 box**).
- **Path** (số lượng các ô có thể đi) **tương đối cao**. (**Từ 28 đến 48 ô**)

=> Hai yếu tố này làm cho **không gian trạng thái** phải xét tăng lên rất lớn.

TÀI LIỆU THAM KHẢO

1. <https://verificationglasses.wordpress.com/2021/01/17/a-star-sokoban-planning/>
2. <https://www.sokobanonline.com/play/web-archive/howard-abed/first-set>
3. <https://ksokoban.online/>
4. <https://github.com/janecakemaster/sokoban>
5. https://github.com/cyndr/blog/blob/master/sokoban_astar/main.py
6. <https://baldur.iti.kit.edu/theses/SokobanPortfolio.pdf>