

ỨNG DỤNG CÁC THUẬT TOÁN BEST-FIRST SEARCH TRONG TRÒ CHƠI SOKOBAN



Thành Viên

Phạm Thị Lệ Quyên
B22DCAT241

Đặng Quân Bảo
B22DCCN060

Lại Duy Đông
B22DCCN216

Bùi Trung Hiếu
B22DCAT115

Hoàng Văn Hướng
B22DCAT156

Đặng Đức Tài
B22DCAT251

Đỗ Chí Tùng
B22DCAT273

Đoàn Minh Yến
B22DCAT320



Nội Dung

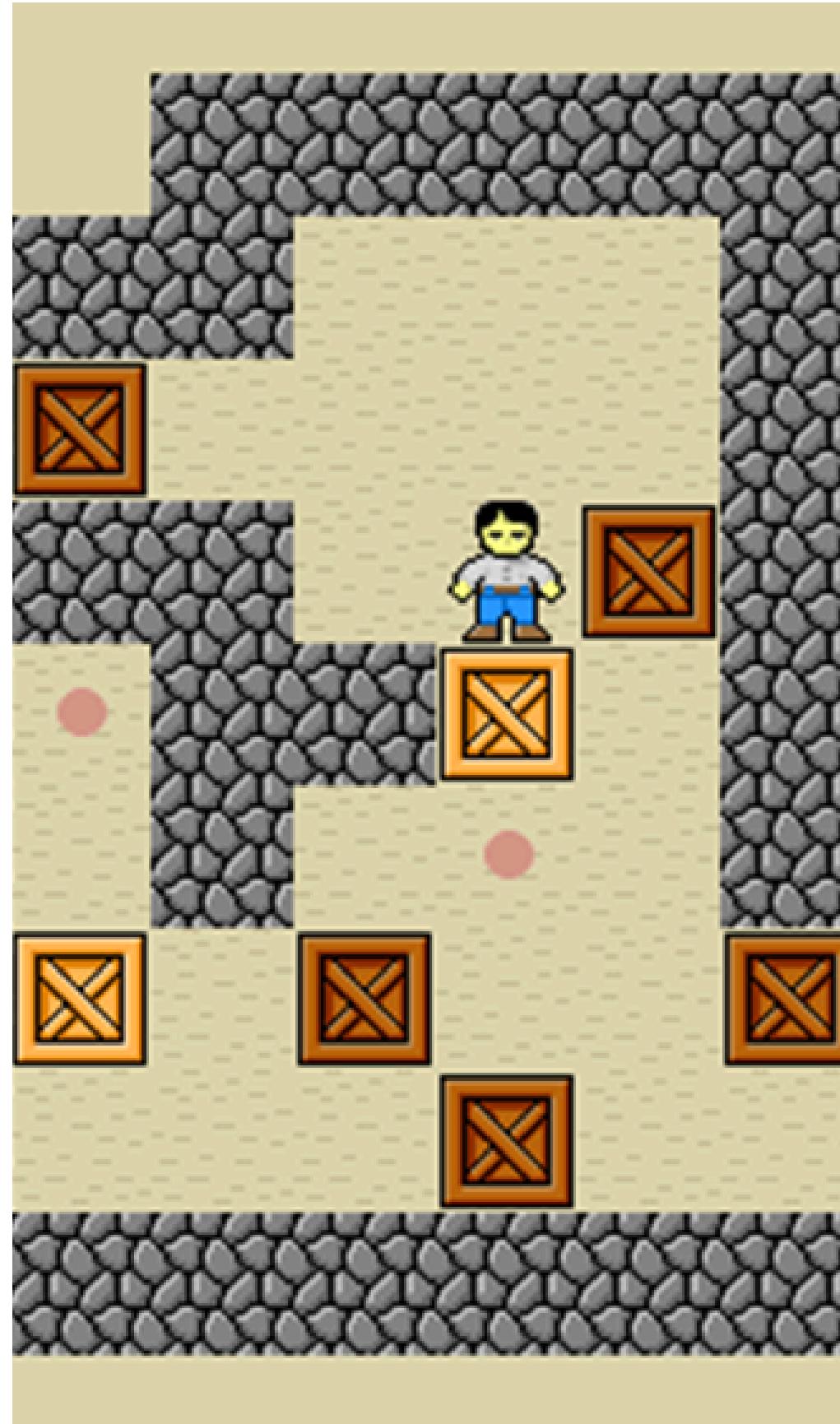


- 1 Giới thiệu về Sokoban
- 2 Tìm hiểu các thuật toán
BEST-FIRST SEARCH
- 3 Phân tích ứng dụng
BFS trong Sokoban
- 4 Triển khai & Đánh giá



1.GIỚI THIỆU.



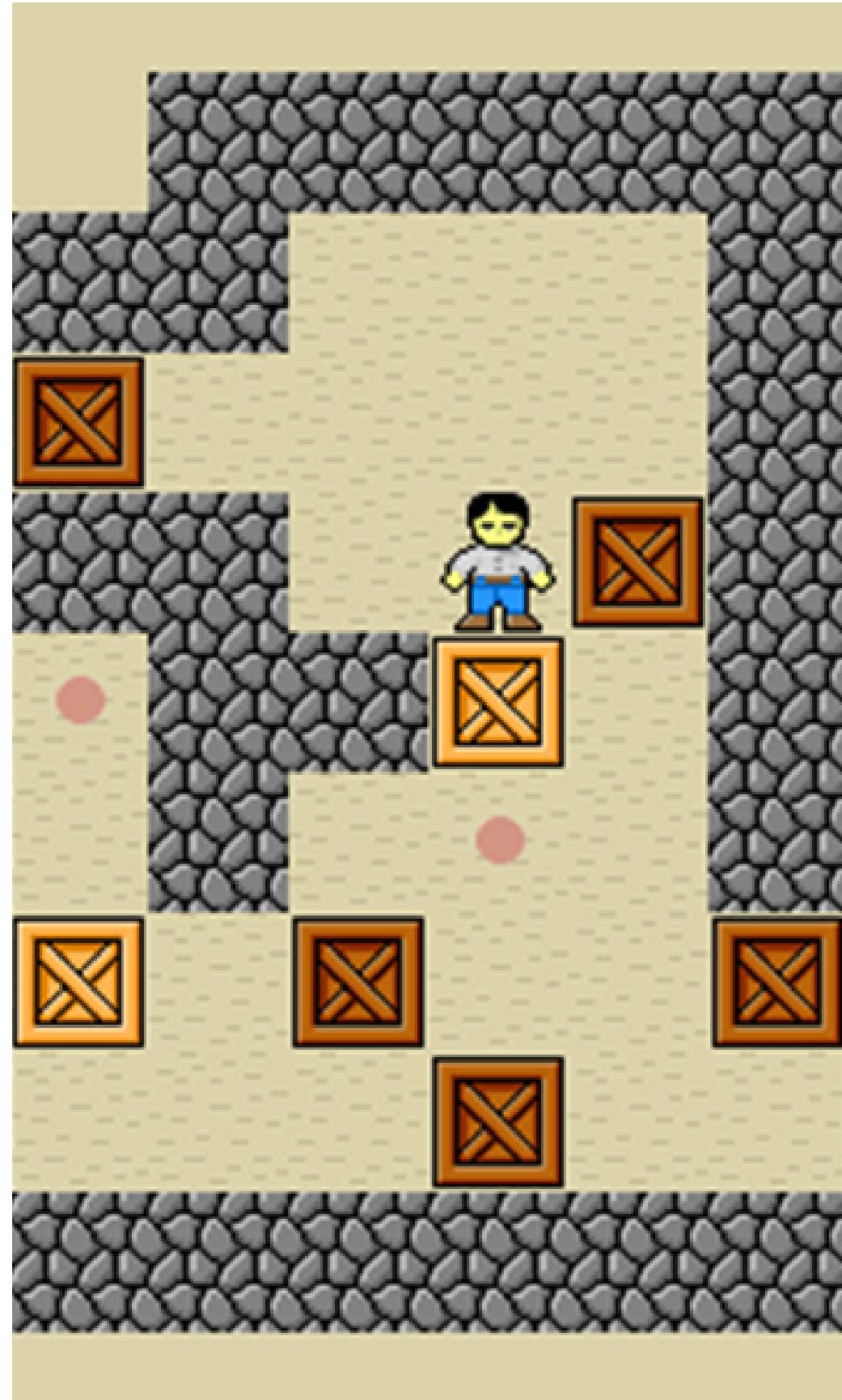


Tổng quan

- Sokoban ra đời năm 1982 tại Nhật Bản, do Hiroyuki Imabayashi phát triển cho Thinking Rabbit.
- Phát hành chính thức trên NEC PC-8801 năm 1984, sau đó lan rộng sang châu Âu và Mỹ.

Bản chất của trò chơi

- Mô phỏng quản lý kho: người chơi là nhân viên kho, đẩy thùng hàng vào vị trí quy định
- Đồ họa 2D lưới ô đơn giản nhưng đòi hỏi tư duy logic và chiến lược cao
- Ứng dụng trong nghiên cứu AI, tối ưu hóa thuật toán và điều khiển robot tự hành



Cách chơi & Luật chơi

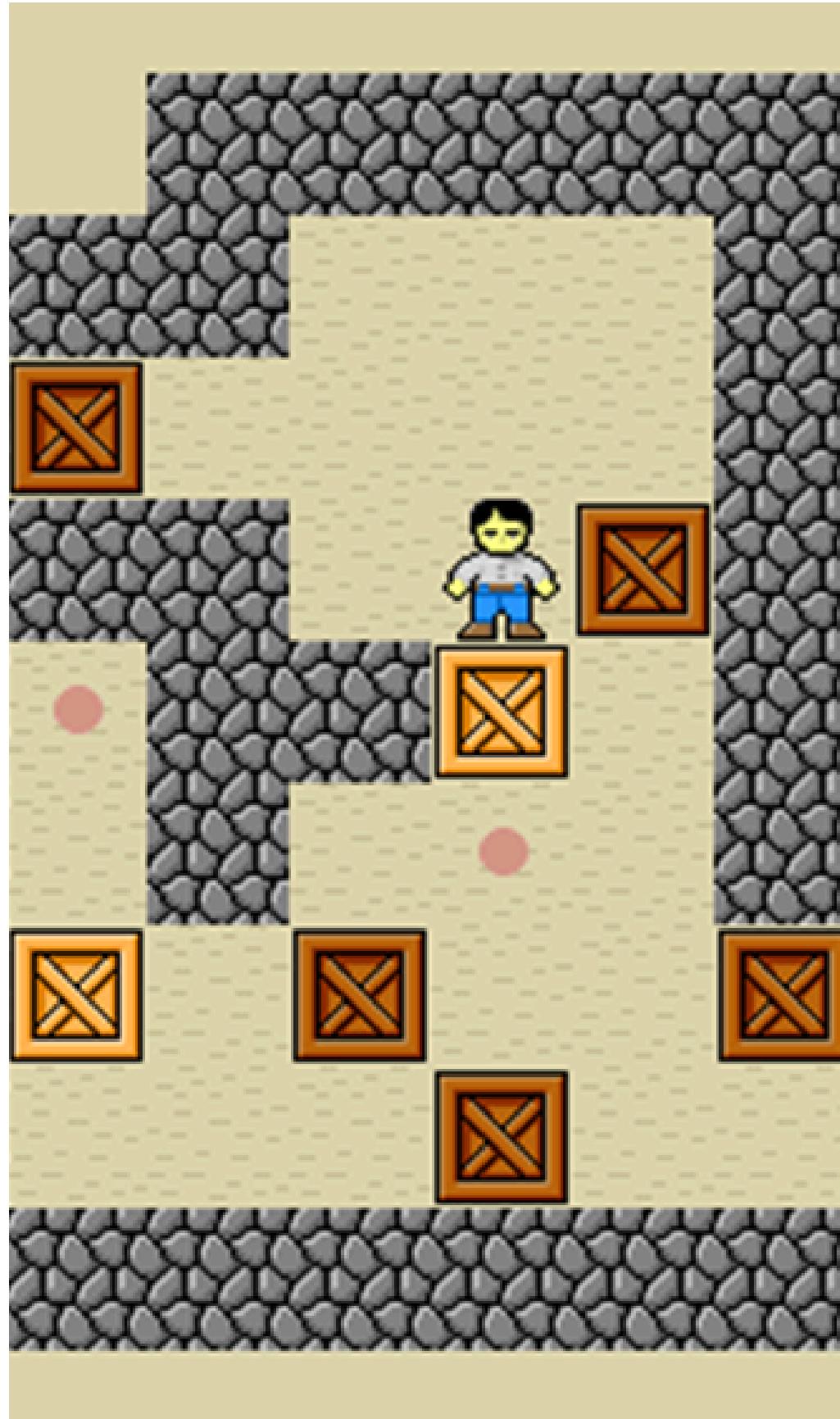
Mục tiêu: Đẩy tất cả hộp (Box) vào các ô đích (Goal Square) trong kho bãі có tường chắn (Wall).

Thành phần

- Người chơi (Sokoban) – nhân vật đẩy hộp.
- Hộp (Box) – vật thể cần di chuyển.
- Tường (Wall) – chướng ngại không thể vượt qua.
- Ô đích (Goal Square) & Sàn trống (Empty Floor).

Luật chơi cơ bản

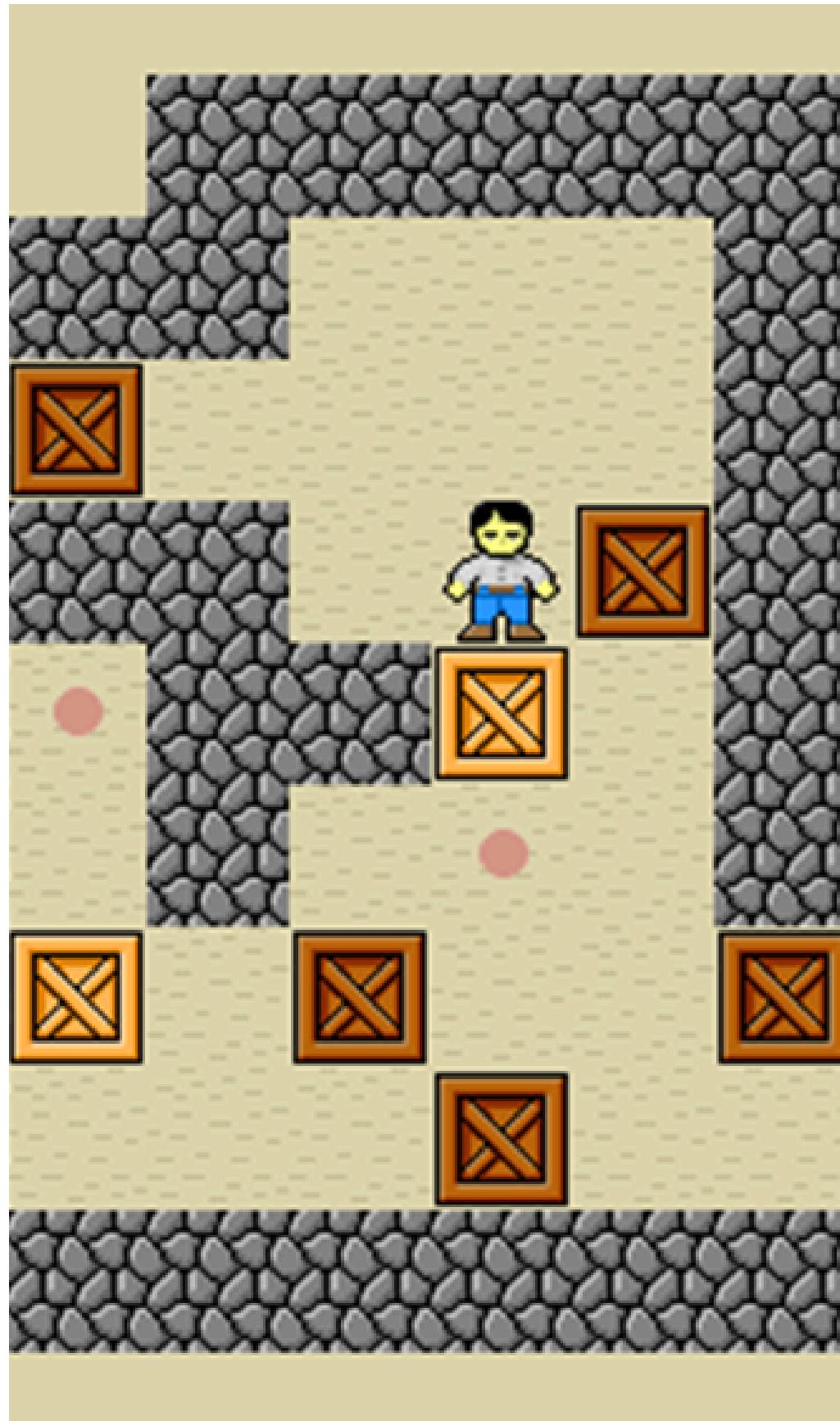
1. Chỉ được đẩy, không kéo hộp.
2. Mỗi lần chỉ đẩy một hộp.
3. Hộp không thể xuyên tường hoặc chồng lên nhau.
4. Trò chơi kết thúc khi tất cả hộp vào ô đích.



Cách chơi & Luật chơi

Yếu tố chiến lược

- Tránh đẩy thùng vào góc: nếu một thùng bị đẩy vào góc (không có mục tiêu ở đó), nó sẽ không thể di chuyển lại – đồng nghĩa thất bại.
- Lên kế hoạch từ xa: đôi khi phải mở đường trước khi đưa thùng đến đích.
- Quản lý không gian: không gian di chuyển là giới hạn, và người chơi có thể tự "nhốt" mình.



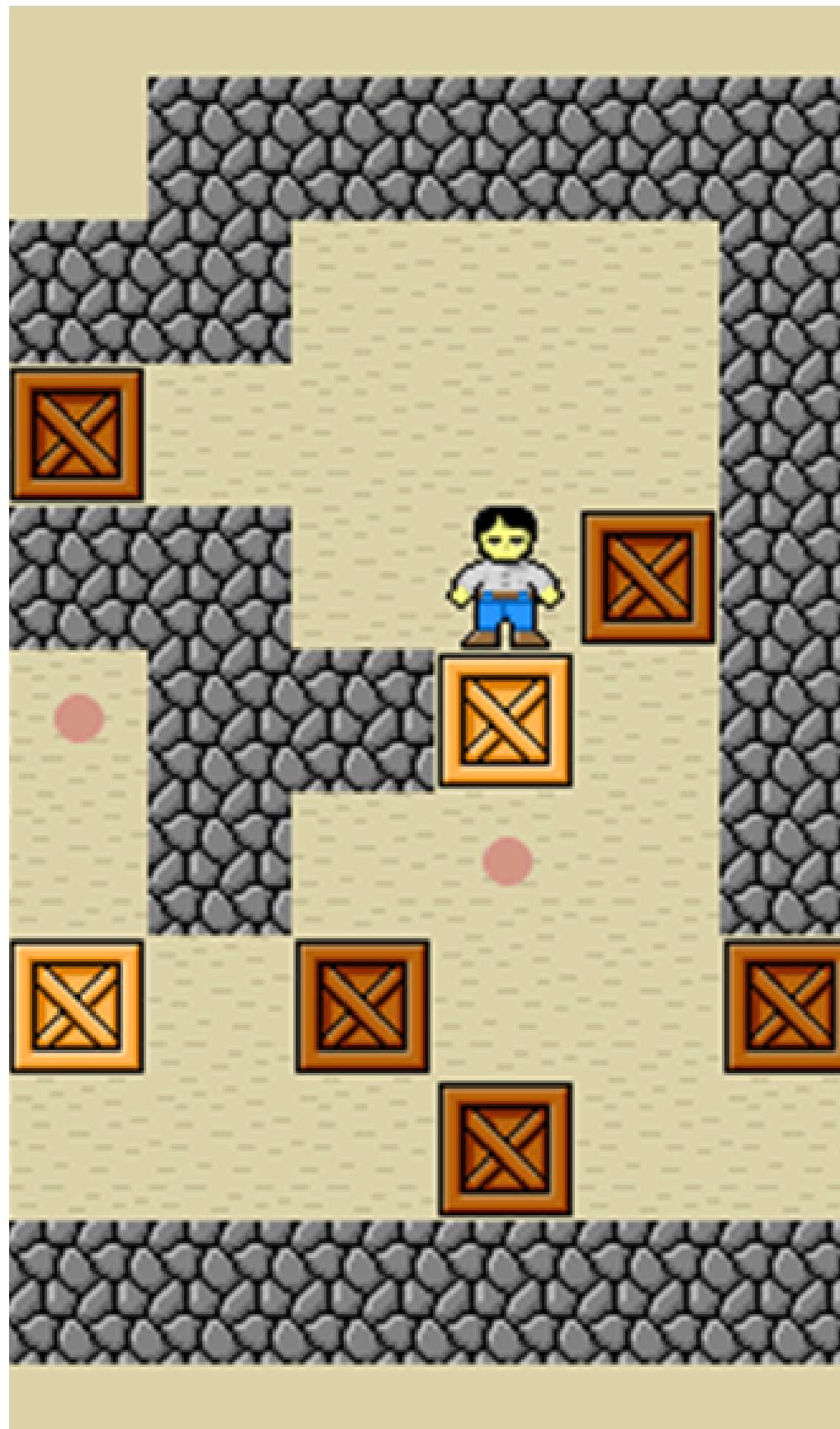
Độ phức tạp và độ khó của Sokoban

Về mặt tính toán

- PSPACE-complete: không gian trạng thái tăng theo cấp số mũ theo kích thước màn chơi
- Giải thuật đòi hỏi nhiều bộ nhớ và thời gian, gần như bất khả thi với bản đồ lớn

Về mặt người chơi

- Số lượng trạng thái khổng lồ: chỉ với ~4 thùng hàng, có thể lên đến hàng trăm nghìn cấu hình
- “Tư duy ngược”: phải lập kế hoạch lùi, tính toán chuỗi bước di chuyển như cờ vua
- Bẫy deadlock: một nhầm lẫn có thể vô hiệu hóa cả màn chơi
- Yêu cầu ghi nhớ chuỗi hành động, kiên nhẫn và tập trung cao độ



Ứng dụng học thuật và công nghệ

Huấn luyện AI & tìm đường

- Môi trường benchmark cho A*, IDDFS, BFS/DFS với heuristic tùy biến
- Thử nghiệm khả năng mở rộng và tối ưu hóa thuật toán

Mô phỏng robot & tự động hóa kho

- Kiểm thử chiến lược di chuyển, sắp xếp hàng hóa
- Áp dụng cho logistics, warehouse automation

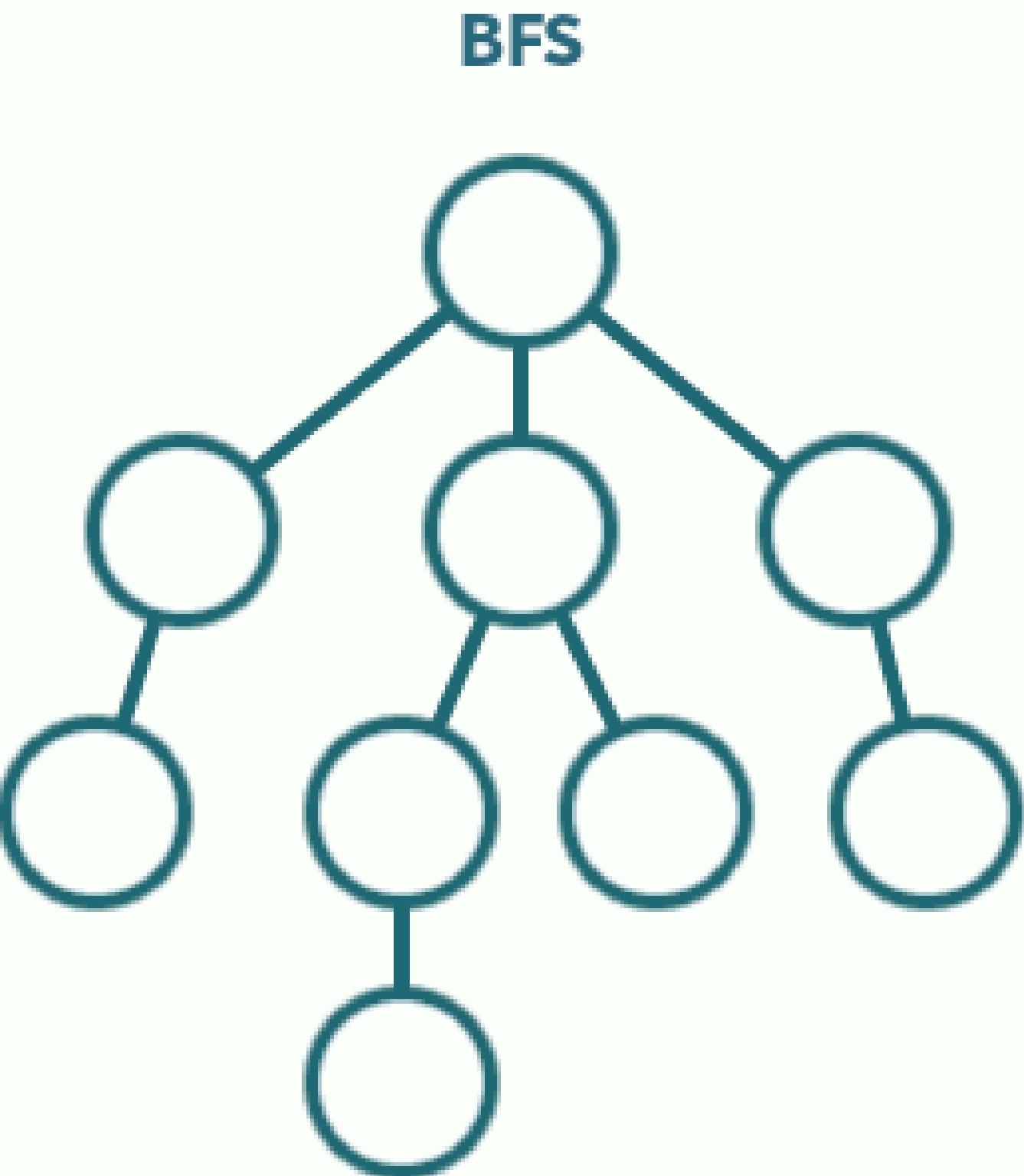
Nghiên cứu tối ưu hóa

- Khám phá heuristic, pattern databases, pruning techniques
- Thử nghiệm hiệu suất và quản lý deadlock

2. TÌM HIỂU VỀ CÁC THUẬT TOÁN BEST-FIRST SEARCH

Nguyên tắc Best-First Search (BFS)

- Sử dụng thông tin heuristic từ bài toán để định hướng, ưu tiên mở rộng các trạng thái “gần” đích hơn.
- Áp dụng hàm đánh giá $f(n) = h(n)$ ước lượng chi phí còn lại từ nút n đến đích
- Dùng Priority Queue lấy luôn nút có $f(n)$ nhỏ nhất; chất lượng và tính admissible của heuristic quyết định hiệu quả tìm kiếm.





Các thuật toán BFS

01

Greedy Best-First Search

- Phương pháp: Mỗi bước chọn nút có $h(n)$ nhỏ nhất
- Ưu điểm: Nhanh nếu heuristic tốt
- Nhược điểm:
 - Không đảm bảo tìm đường ngắn nhất
 - Dễ lạc vào deadlock

02

Thuật toán A*

- Phương pháp: Kết hợp chi phí đường đi $g(n)$ và heuristic $h(n)$
- Tính chất:
 - Đầy đủ và tối ưu nếu $h(n)$ chấp nhận được (admissible)
 - Yêu cầu bộ nhớ lớn

03

IDA*

- Ý tưởng: Lắp DFS với ngưỡng tăng dần theo $f(n)$
- Ưu/nhược:
 - Tiết kiệm bộ nhớ
 - Tốn thời gian lắp lại nhiều lần



3. PHÂN TÍCH ỨNG DỤNG BFS TRONG SOKOBAN

01

Sơ lược giải thuật đối với Sokoban

02

Kỹ thuật tối ưu hóa

03

Một số hạn chế khi ứng dụng BFS



Sơ lược giải thuật đối với Sokoban

Biểu diễn trạng thái

- State = (Vị trí người chơi (x,y) , Tập vị trí hộp $\{(x_i, y_i)\}$).
- Tường & map: ma trận 2D hoặc danh sách ô cấm.
- Không gian trạng thái tăng theo cấp số mũ với số hộp \rightarrow Uninformed search thường không hiệu quả

Mô hình tìm kiếm

- Initial state: vị trí người + hộp ban đầu
- Actions: di chuyển hoặc đẩy hộp (hợp lệ)
- Transition: sinh state mới từ state + action
- Goal: tất cả hộp trên vị trí đích
- Cost per action = 1

Chiến lược tìm kiếm

- Uninformed: DFS/BFS – dễ cài nhưng có thể lặp vô hạn, không tối ưu
- Informed (heuristic):
 - Greedy BFS: chọn n có $f(n)=h(n)$ nhỏ nhất (không tối ưu, có thể lặp)
 - A*: $f(n)=g(n)+h(n)$, đầy đủ & tối ưu nếu h admissible (bộ nhớ lớn)
 - IDA*: DFS lặp với ngưỡng f tăng dần, space tuyến tính, tính toán lặp lại



Sơ lược giải thuật đối với Sokoban

Thiết kế và yêu cầu Heuristic

- Mục đích: Ước lượng chi phí $h(n)$ từ state hiện tại đến goal để hướng tìm kiếm
- Các loại heuristic phổ biến:

1. Tổng khoảng cách Manhattan:

$$h(n) = \sum_i \min_j |box_i.x - goal_j.x| + |box_i.y - goal_j.y|$$

2. Minimum Cost Matching: Gán hộp-đích sao cho tổng chi phí (Manhattan hoặc số đếm) nhỏ nhất (Hungarian algorithm).

3. Pattern Databases (PDBs): Tra cứu chi phí tối ưu cho các mẫu con đã lưu trước.

4. Deadlock detection: Nếu phát hiện deadlock (góc tường, ngõ cụt), gán $h(n) = \infty$

- Yêu cầu với $h(n)$
 - Admissible: không bao giờ over-estimate ($h(n) \leq h^*(n)$) $\rightarrow A^*$ tối ưu
 - Consistent: $h(n) \leq cost(n,a,n') + h(n') \rightarrow$ đảm bảo mở rộng node tối ưu

Kỹ thuật tối ưu hóa

Phát hiện Deadlock

- Tầm quan trọng: Loại bỏ sớm các trạng thái không thể dẫn đến giải pháp để tiết kiệm thời gian và bộ nhớ.
- Các kiểu deadlock chính:

1. Corner: hộp vào góc hai bức tường:

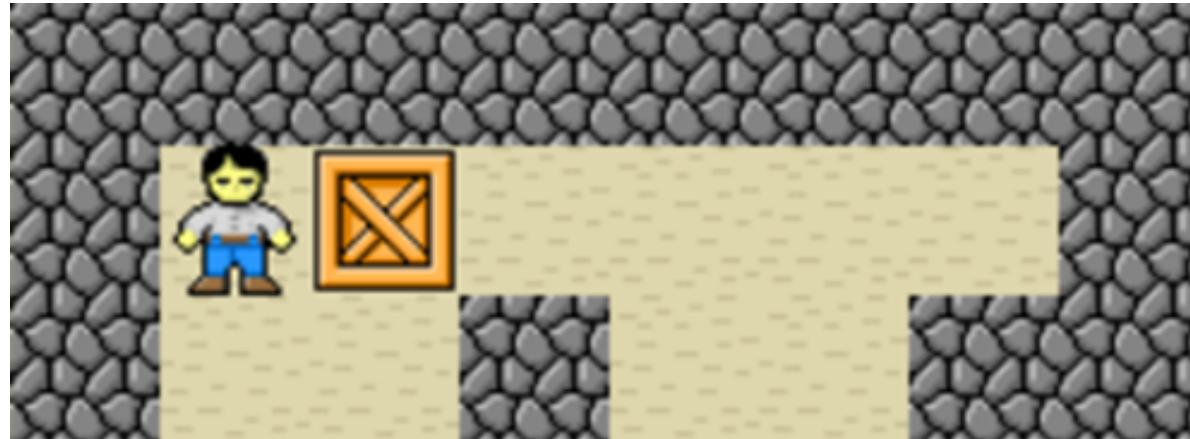


2. Freeze: nhiều hộp kẹt vào nhau, không hộp nào di chuyển được

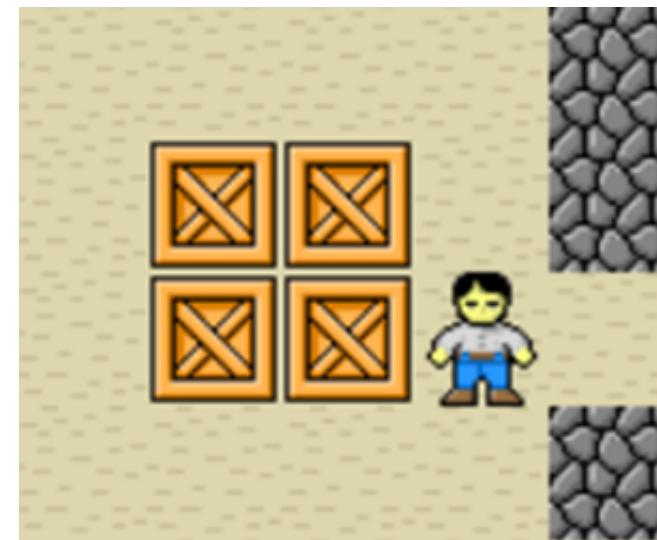


Kỹ thuật tối ưu hóa

3. Wall: hộp dọc tường không có đích
dọc theo đoạn tường đó

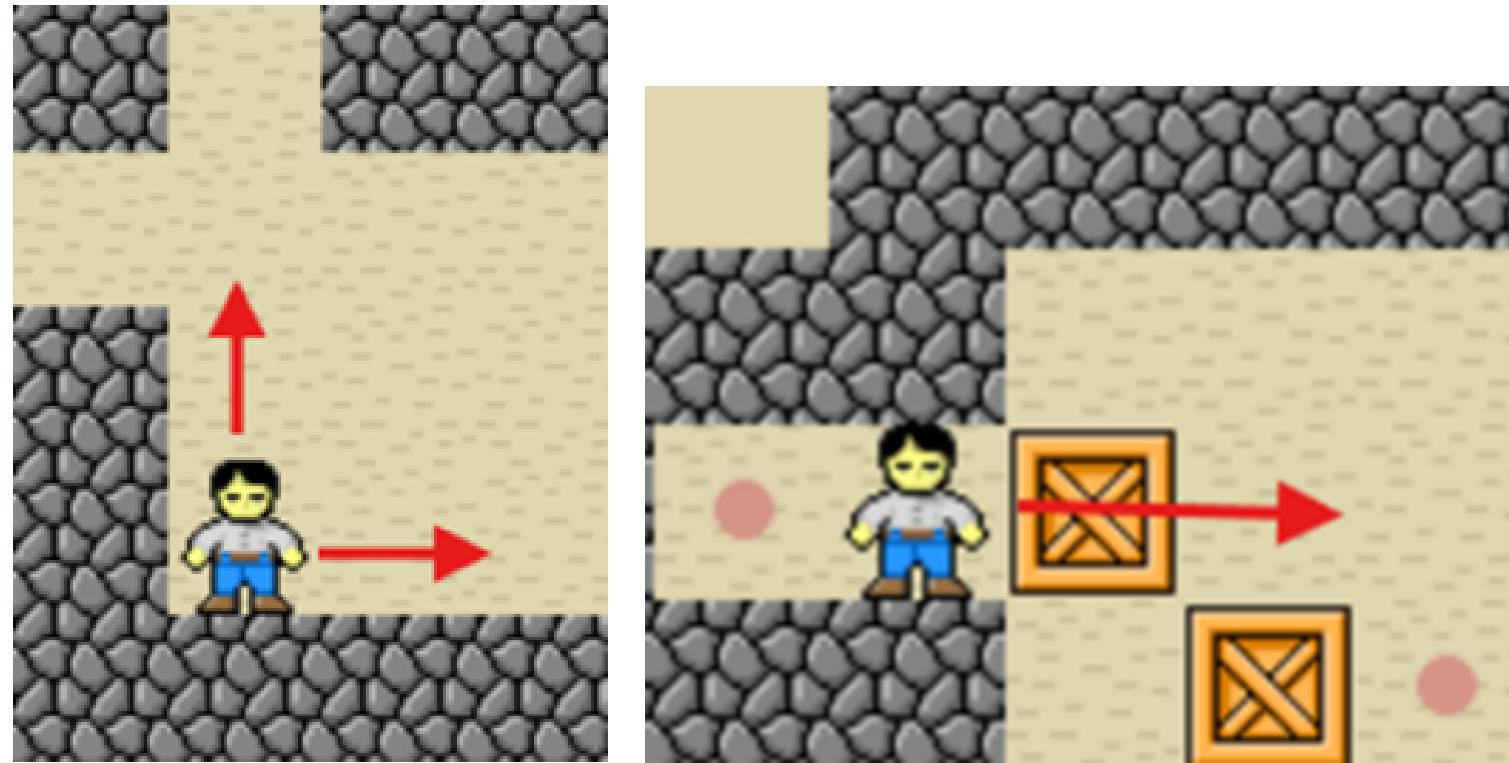


4. 2x2 Block: bốn hộp tạo khối vuông,
không thể tách ra



- Các kỹ thuật phát hiện deadlock có thể được phân loại như sau:
 - Pattern-based: kiểm tra mẫu deadlock đơn giản khi sinh trạng thái.
 - Static analysis: đánh dấu ô “nguy hiểm” (góc vô đích) ngay từ đầu.
 - Dynamic checking: mỗi lần đẩy hộp, xác định xem có kẹt không.

Sinh trạng thái kế tiếp hợp lệ



1. Người chơi di chuyển đến một ô trống bên cạnh: Ô kế bên phải là sàn trống → người chơi vào ô đó
2. Người chơi đẩy một hộp vào một ô trống phía sau hộp đó: Ô kế bên chứa hộp và ô phía sau hộp là sàn trống → đẩy hộp, người chơi chiếm ô vừa đẩy

Các trường hợp không sinh ra trạng thái kế tiếp hợp lệ:

- Chạm tường hoặc chạm hộp rồi đẩy vào tường/hộp khác
- Đẩy đồng thời hai hộp



Một số hạn chế khi ứng dụng BFS

- **Không gian trạng thái khổng lồ**
 - Vị trí người & nhiều hộp → nổ trạng thái theo cấp số mũ.
 - Dễ tràn bộ nhớ và chậm thời gian tìm kiếm.
- **Khó thiết kế heuristic tốt**
 - Heuristic kém → lạc hướng, chậm hoặc không tìm được giải.
 - Heuristic mạnh → tốn chi phí tính toán, giảm lợi ích.
- **Deadlock thiếu cơ chế xử lý**
 - Nếu không detect deadlock, thuật toán sẽ khám phá nhánh vô ích.
 - Gây lãng phí tài nguyên.



4. Triển khai và Đánh giá

Thuật toán A* (khi mode == 1) sử dụng $f(n) = g(n) + h(n)$ để tìm đường đi ngắn nhất từ trạng thái ban đầu đến trạng thái thắng.

```
def AStar_Search(board, list_check_point):
    start_time = time.time()
    ''' A* SEARCH SOLUTION '''
    if spf.check_win(board, list_check_point):
        print("Found win")
        return [board]
    start_state = spf.state(board, None, list_check_point, 1)
    list_state = [start_state]

    heuristic_queue = PriorityQueue()
    heuristic_queue.put(start_state)

    while not heuristic_queue.empty():
        now_state = heuristic_queue.get()
        cur_pos = spf.find_position_player(now_state.board)
        list_can_move = spf.get_next_pos(now_state.board, cur_pos)
        for next_pos in list_can_move:
            new_board = spf.move(now_state.board, next_pos, cur_pos, list_check_point)
            if spf.is_board_exist(new_board, list_state):
                continue
            if spf.is_board_can_not_win(new_board, list_check_point):
                continue
            if spf.is_all_boxes_stuck(new_board, list_check_point):
                continue
            new_state = spf.state(new_board, now_state, list_check_point, 1)
            if spf.check_win(new_board, list_check_point):
                print("Found win")
                print(time.time() - start_time)
                return (new_state.get_line(), len(list_state))
            list_state.append(new_state)
            heuristic_queue.put(new_state)
        end_time = time.time()
        if end_time - start_time > spf.TIME_OUT:
            return []
        end_time = time.time()
        if end_time - start_time > spf.TIME_OUT:
            return []
    print("Not Found")
    return []
```



Khởi tạo

- Tạo *start_state* từ *board* + *list_check_point* ($g=0$ cho A*, mode).
- Dùng *PriorityQueue* ưu tiên theo *f* nhỏ nhất.

Vòng lặp tìm kiếm

- Lấy trạng thái *now_state* có *f* nhỏ nhất.
- Xác định vị trí người chơi & các nước đi hợp lệ (*get_next_pos*).
- Với mỗi nước đi:
 - Sinh *new_board* = *move(now_state, next_pos)*.
 - Lọc bỏ nếu: đã duyệt (*is_board_exist*), không thể thắng (*is_board_can_not_win*), hoặc deadlock (*is_all_boxes_stuck*).
- Nếu *check_win(new_board)* → in thời gian, trả về chuỗi hành động.

Kết thúc

- Quá *TIME_OUT* hoặc queue rỗng → trả về []



Thuật toán Greedy BFS (khi mode != 1) chỉ sử dụng $h(n)$ để ưu tiên các trạng thái gần mục tiêu hơn. Tương tự A*, chỉ khác ở cách tạo tham số mode khi tạo đối tượng `spf.state`

```
def Best_First_Search(board, list_check_point):
    start_time = time.time()
    ''' Best First SEARCH SOLUTION '''
    if spf.check_win(board, list_check_point):
        print("Found win")
        return [board]
    start_state = spf.state(board, None, list_check_point, 0)
```



Đánh giá

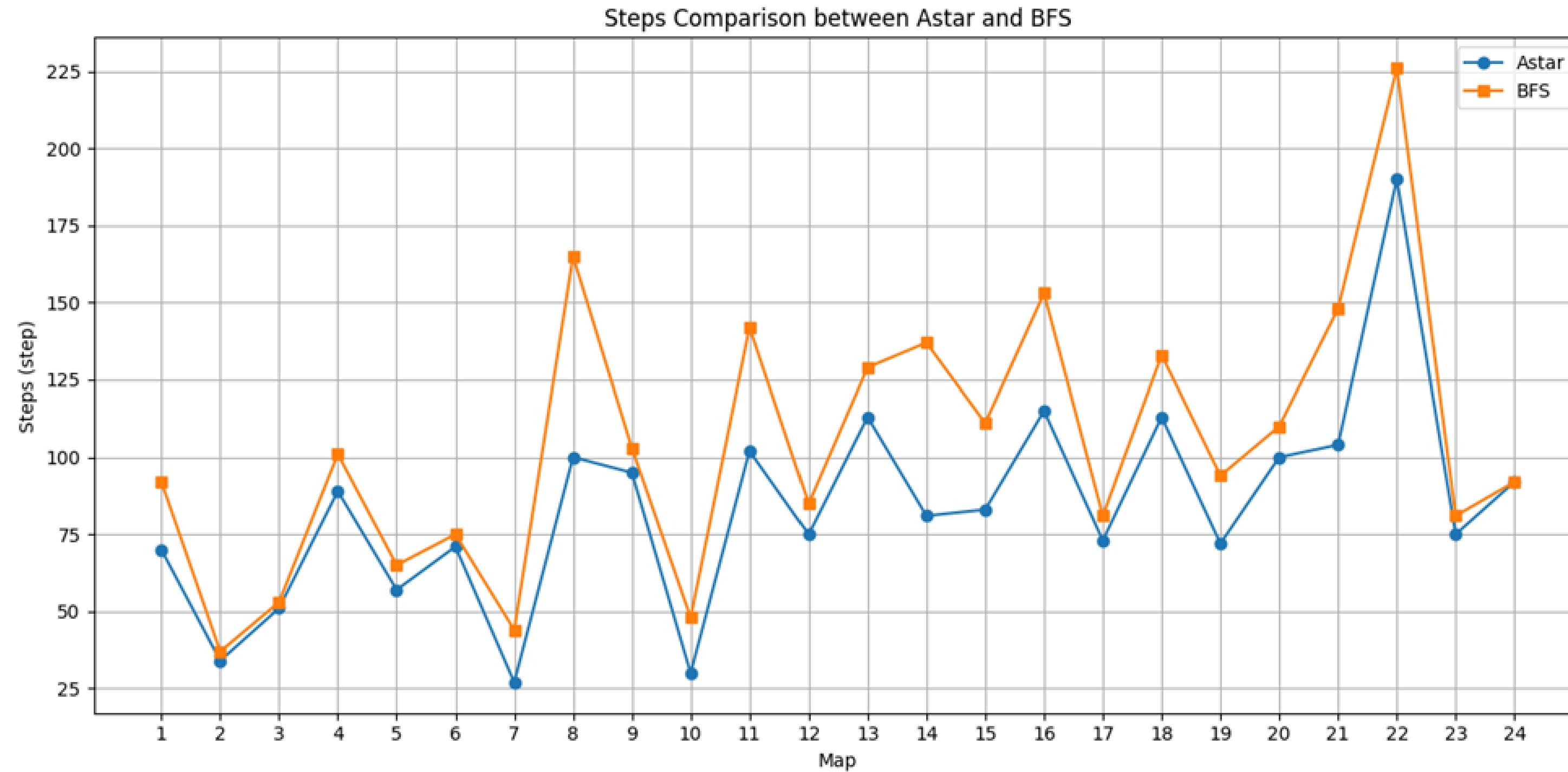
- Về số bước giải:
 - A* tối ưu hơn Greedy Best First Search về số bước giải ở tất cả các map
- Về thời gian và số trạng thái đã khám phá (States)
 - Greedy Best First Search tối ưu hơn A* ở hầu hết các map



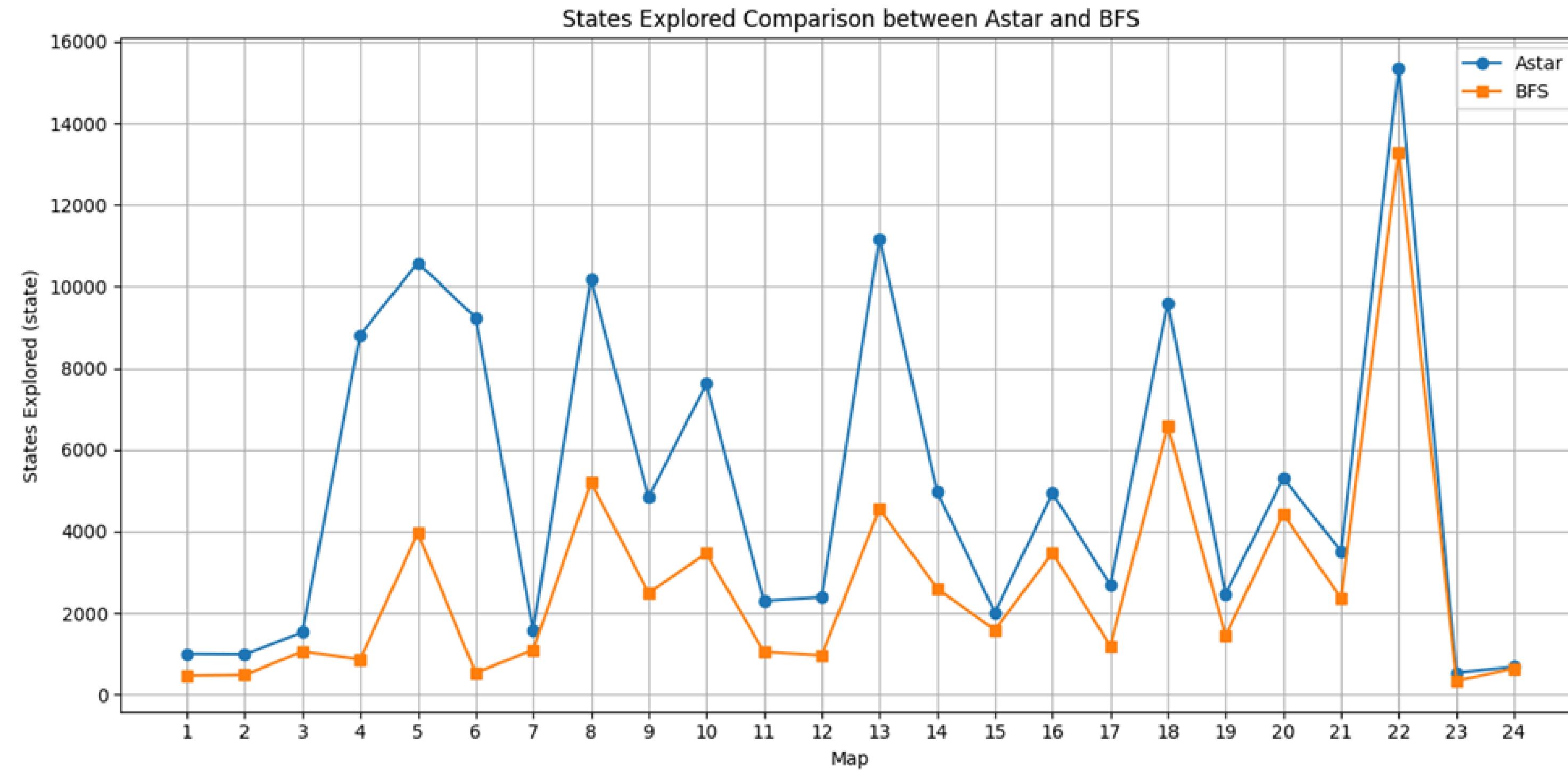
Bảng các tham số của từng map

Map	Timeout: 1800s					
	BFS			A*		
	Steps	StatesExplored	ExecutionTime	Steps	StatesExplored	ExecutionTime
1	92	472	0.556	70	998	1.596
2	37	488	0.566	34	989	1.063
3	53	1057	1.776	51	1534	2.673
4	101	866	2.519	89	8811	174.92
5	65	3972	49.221	57	10581	160.521
6	75	538	0.976	71	9237	151.107
7	44	1098	2.575	27	1589	3.289
8	165	5204	102.185	100	10182	200.025
9	103	2495	28.791	95	4840	41.343
10	48	3466	45.392	30	7618	90.703
11	142	1050	4.551	102	2303	8.685
12	85	962	4.758	75	2393	10.038
13	129	4550	85.674	113	11172	225.961
14	137	2603	34.224	81	4977	58.887
15	111	1587	12.535	83	2015	8.154
16	153	3477	35.071	115	4936	45.328
17	81	1194	5.36	73	2691	14.125
18	133	6576	141.056	113	9590	177.952
19	94	1443	6.368	72	2467	10.224
20	110	4422	56.455	100	5301	52.506
21	148	2359	18.202	104	3517	23.578
22	226	13294	405.757	190	15333	523.672
23	81	344	0.359	75	543	0.632
24	92	647	1.208	92	688	0.86

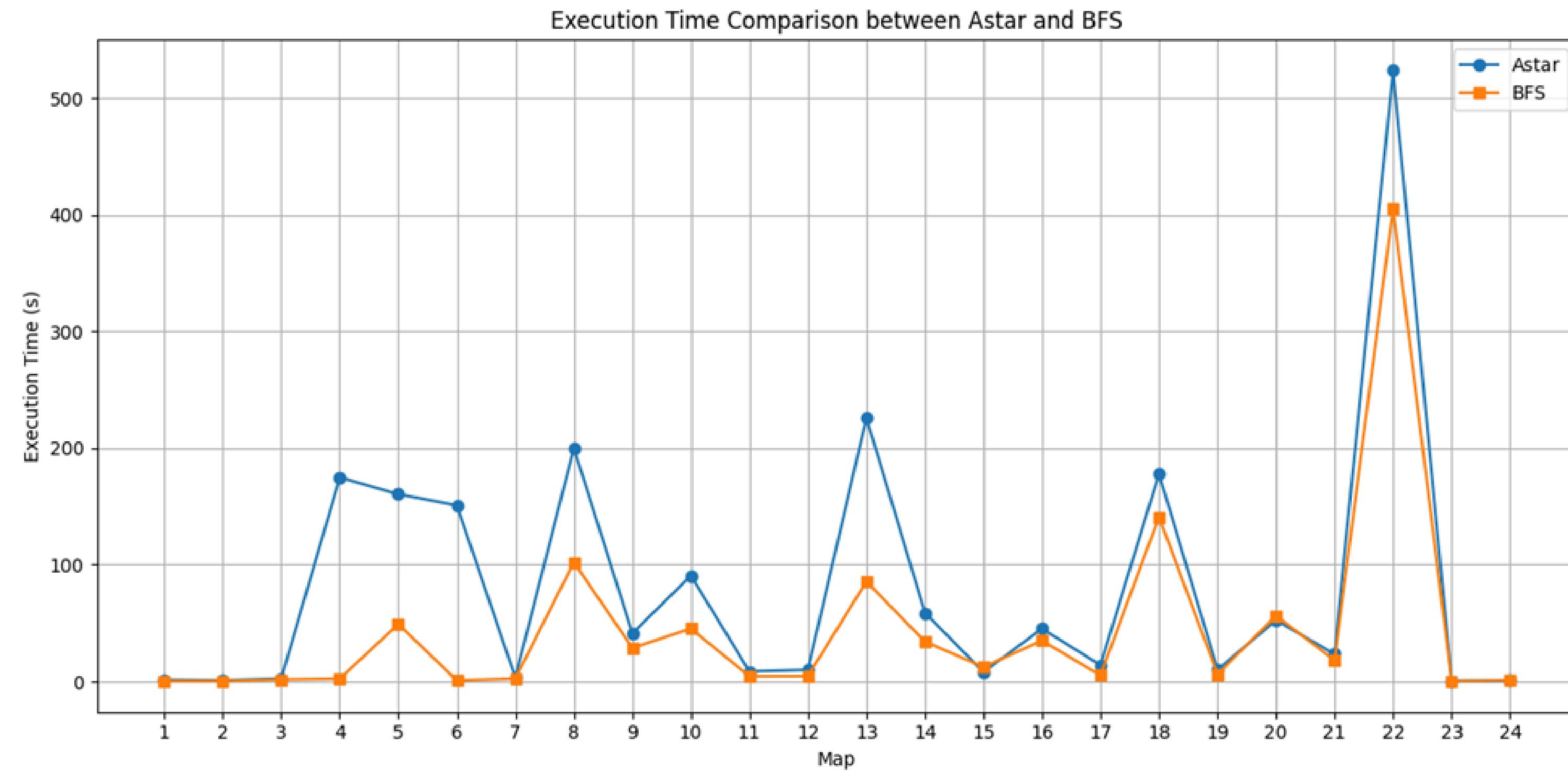
Biểu đồ so sánh giữa số bước giải giữa hai giải thuật:



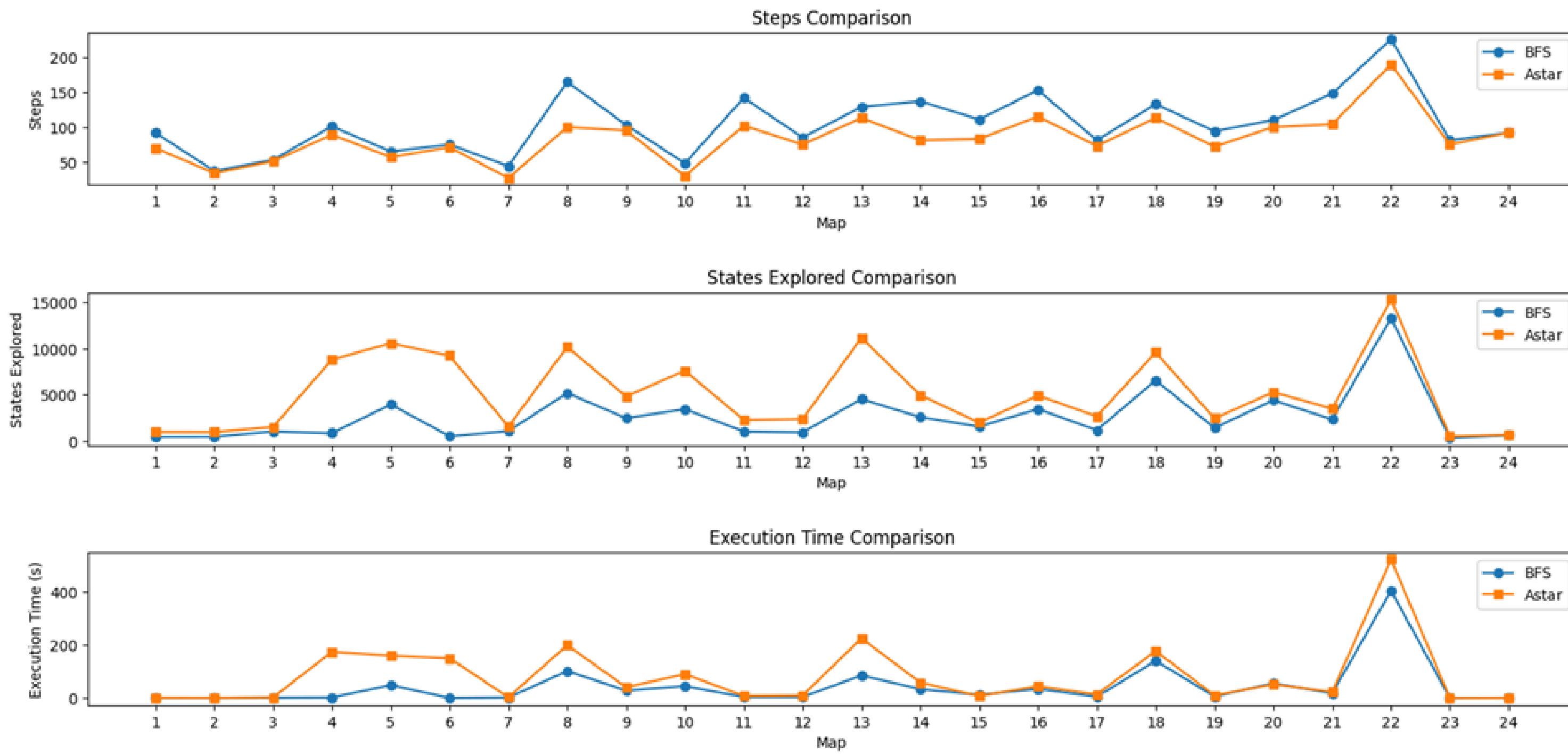
Biểu đồ so sánh giữa số lượng trạng thái cần duyệt giữa hai giải thuật



Biểu đồ so sánh thời gian chạy giữa hai giải thuật:



Biểu đồ tổng quan:





Thank You