ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC BÁCH KHOA

BÁO CÁO BÀI TẬP LỚN SỐ 1
MÔN HỌC: NHẬP MÔN TRÍ TUỆ NHÂN TẠO
HK251 - Lớp: L02
(CO3061)

Đề tài: Giải Bài Toán Sokoban

GVHD: ThS. Vương Bá Thịnh

Thành viên:	Nguyễn Thành Phát	–	2312593
	Nguyễn Thành Công	–	2310373
	Nguyễn Anh Kiệt	–	2211758
	Nguyễn Văn Nhật Tân	–	2213064
	Phạm Xuân Vũ	–	2313966

Thành Phố Hồ Chí Minh, tháng 10 năm 2025

Contents

Thành Viên Nhóm

Stt	Họ và Tên	MSSV	Nội dung thực hiện	Ký tên xác nhận
1	Nguyễn Thành Phát	2312593	Code BrFS Phân tích bài toán, đánh giá giải thuật	
2	Nguyễn Thành Công	2310373	Code A* Trình bày giải thuật A*	
3	Nguyễn Anh Kiệt	2211758	Code demo pygame Kết luận và mở rộng	
4	Nguyễn Văn Nhật Tân	2213064	Code thu thập, load các testcase Phân tích bài toán, trình bày giải thuật BrFS	
5	Phạm Xuân Vũ	2313966	Code demo pygame Giới thiệu bài toán	

I Giới Thiệu

1 Đề bài

Bài toán **Sokoban** là một trò chơi logic cổ điển, thường được sử dụng để minh họa và kiểm chứng các kỹ thuật tìm kiếm trong lĩnh vực Trí tuệ Nhân tạo (AI). Người chơi điều khiển một nhân vật trong kho hàng với nhiệm vụ đẩy các thùng (*boxes*) vào các vị trí đích (*goals*) đã định, tuân theo quy tắc: nhân vật chỉ có thể đẩy chứ không thể kéo, và không được xuyên tường. Thách thức của bài toán nằm ở việc xác định chuỗi hành động hợp lệ với chi phí di chuyển tối thiểu, trong khi không gian trạng thái tăng lên theo cấp số nhân khi số thùng và kích thước bản đồ tăng.

2 Mục tiêu

Đồ án *"Sokoban Solver"* được thực hiện với các mục tiêu sau:

- Xây dựng chương trình có khả năng tự động tìm lời giải Sokoban bằng hai giải thuật tìm kiếm: **Breadth-First Search (BrFS)** và **A* Search**;
- Ứng dụng **kỹ thuật phát hiện deadlock** để cắt tỉa không gian tìm kiếm, từ đó tối ưu hiệu năng;
- Đo lường và so sánh hiệu năng giữa hai giải thuật dựa trên thời gian chạy, số lượng node mở rộng và bộ nhớ sử dụng;
- Trực quan hóa quá trình tìm kiếm và kết quả thông qua **giao diện hiển thị Pygame**.

3 Phạm vi nghiên cứu

Nghiên cứu tập trung vào các bản đồ Sokoban có độ khó từ dễ đến trung bình, được lấy từ hai bộ bản đồ `Mini Cosmos` và `Micro Cosmos` tại `https://ksokoban.online`. Chương trình không xét đến các yếu tố mở rộng như nhiều người chơi, môi trường động hay học tăng cường, mà chỉ tập trung vào việc đánh giá và tối ưu hai thuật toán tìm kiếm cổ điển.

Hai thuật toán được hiện thực theo định hướng sau:

- **BrFS:** Tìm kiếm toàn diện theo lớp, đảm bảo tìm được lời giải tối ưu nhưng tiêu tốn nhiều bộ nhớ và thời gian khi không gian trạng thái lớn;
- **A*:** Áp dụng hàm heuristic dựa trên **Hungarian Algorithm** để ước lượng chi phí di chuyển tối thiểu giữa các hộp và vị trí đích.

4 Công nghệ sử dụng

Ngôn ngữ lập trình: Python 3.10

Thư viện và công cụ hỗ trợ:

- `pygame`: Hiển thị bản đồ và mô phỏng trực quan quá trình tìm kiếm;
- `scipy.optimize.linear_sum_assignment`: Cài đặt Hungarian Algorithm cho hàm heuristic của A*;
- `tracemalloc`: Đo lường mức sử dụng bộ nhớ trong quá trình thực thi;
- `numpy`: Xử lý dữ liệu và ma trận bản đồ.

II Phân Tích Bài Toán

1 Định nghĩa trạng thái

Trong hiện thực của nhóm, mỗi trạng thái được biểu diễn bằng một đối tượng (*object*) thuộc lớp `SokobanState`.

Các thuộc tính quan trọng của `SokobanState`:

- `self.map`: Bản đồ tổng thể bài toán
- `self.player`: Tọa độ (x, y) của người chơi.
- `self.boxes`: Một danh sách (list) chứa tọa độ (x, y) của tất cả các thùng.
- `self.path`: Một danh sách (list) chứa các hành động ('up', 'down', 'left', 'right') đã thực hiện để đi từ trạng thái ban đầu đến trạng thái này.

Trong đó `self.map` là một ma trận 2 chiều có quy định các ký tự ở bảng 1

Kí tự	Ý nghĩa	Hình ảnh
#	tường	
.	ô trống	
?	đích	
x	hộp	
+	hộp ở đích	
@	người chơi	
-	người chơi ở đích	

Bảng 1: Bảng quy định ký tự

Để thuật toán hoạt động chính xác, chúng ta phải định nghĩa cách so sánh và băm các trạng thái. Hai trạng thái được coi là giống hệt nhau nếu chúng có cùng vị trí người chơi và cùng một tập hợp vị trí các thùng (không phân biệt thứ tự).

Điều này được hiện thực qua phương thức `__hash__`:

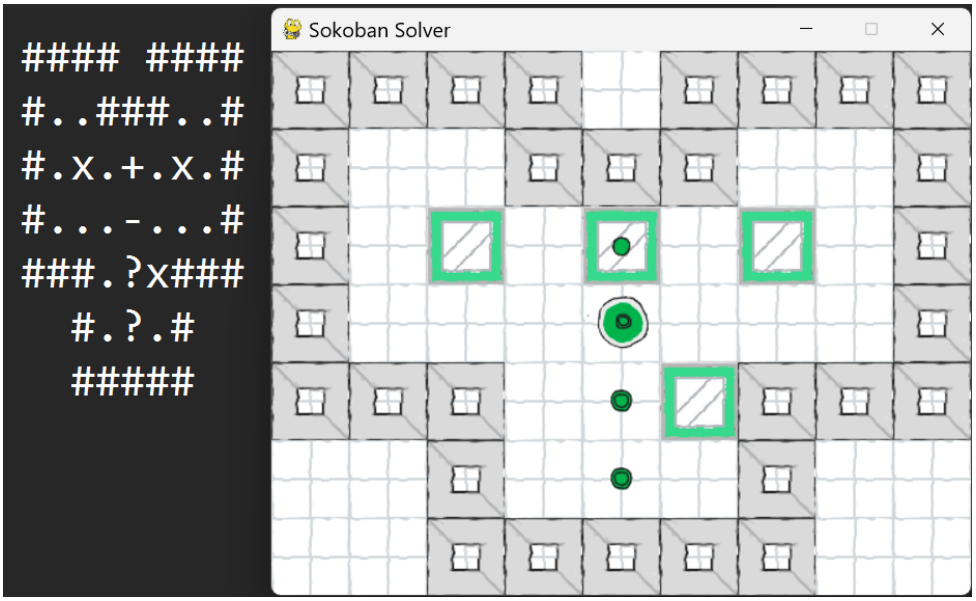
```
def __hash__(self):
    # Sort boxes so order doesn't matter
    return hash((self.player, tuple(sorted(self.boxes))))
```

Listing 1: Phương thức `__hash__` trong class `SokobanState`

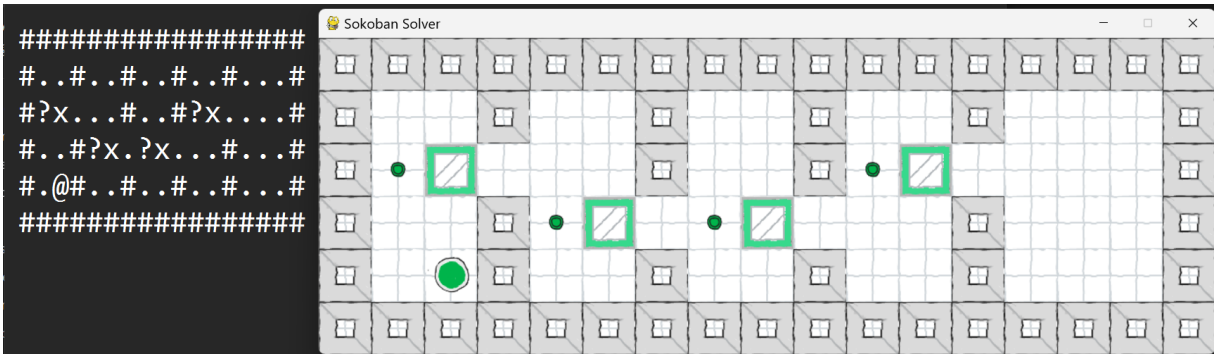
2 Trạng thái khởi đầu

Trạng thái khởi đầu là ma trận 2 chiều được đọc từ file **testcase<*id*>.txt**.

Ví dụ:



Hình 1: Trạng thái khởi đầu testcase 1

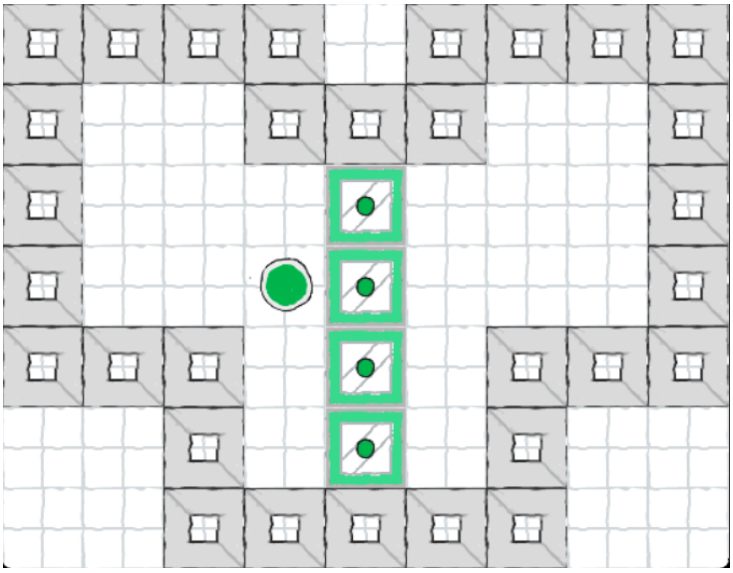


Hình 2: Trạng thái khởi đầu testcase 2

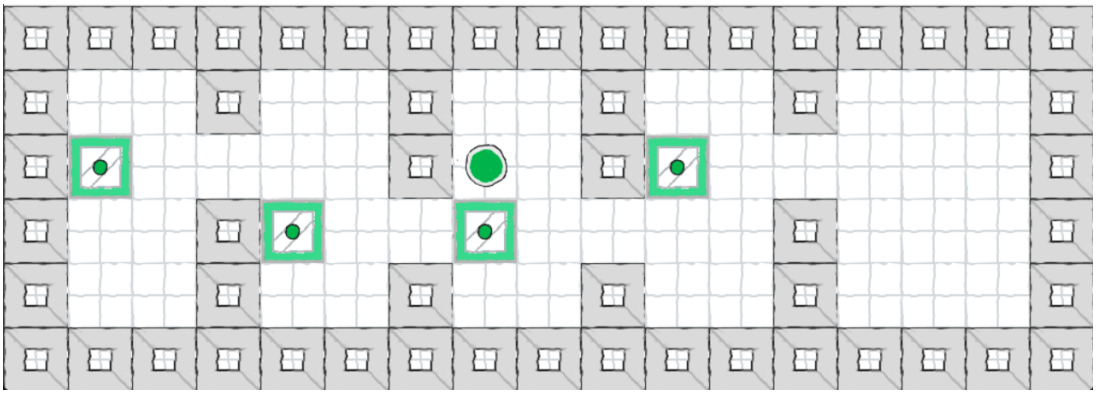
3 Trạng thái đích

$$\text{Boxes} \subseteq \text{Goals} \tag{1}$$

Là trạng thái mà tất cả các hộp được đặt ở vị trí đích
Ví dụ:



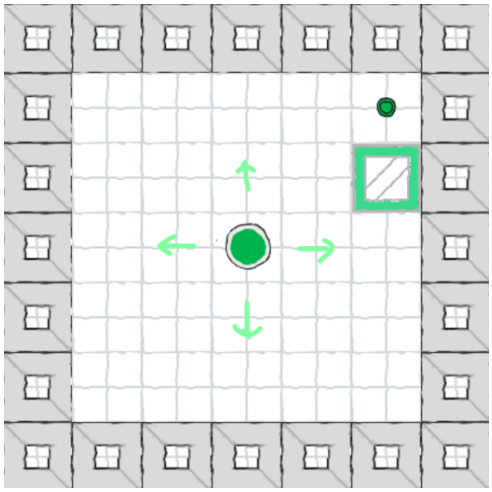
Hình 3: Trạng thái đích của testcase 1



Hình 4: Trạng thái đích của testcase 2

4 Nước đi hợp lệ

Từ một trạng thái bất kỳ được định nghĩa như trên có thể sinh ra **Tối Đa** 4 trạng thái con tương ứng với di chuyển nhân vật theo 4 hướng (lên, xuống, trái phải).

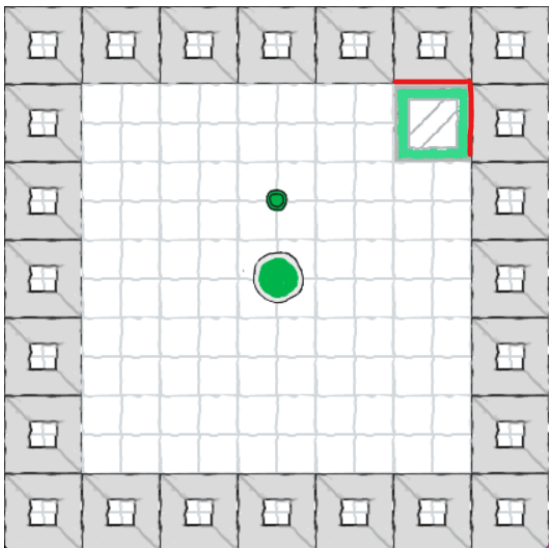


Hình 5: Di chuyển cơ bản

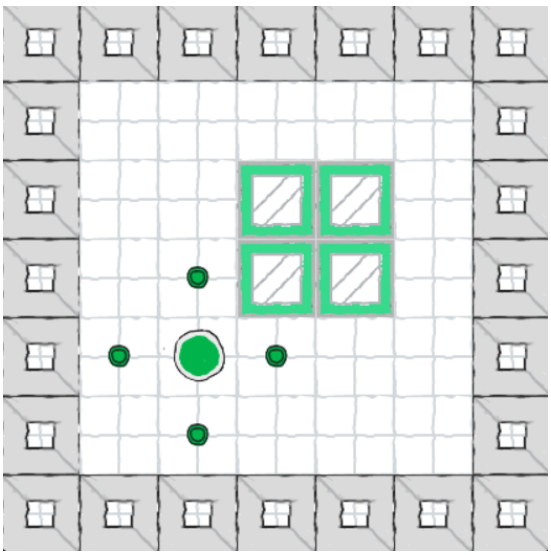
Một cải tiến quan trọng trong hiện thực của nhóm là khả năng phát hiện "deadlock" – các trạng thái mà từ đó không thể nào đi đến chiến thắng (ví dụ: thùng bị đẩy vào góc).

Bằng cách phát hiện sớm và ngừng khám phá các nhánh này, chúng ta có thể thu hẹp đáng kể không gian tìm kiếm. Hàm này kiểm tra 3 loại deadlock cơ bản:

- **Corner Deadlock**: là trạng thái tồn tại một hộp không ở đích bị *bao quanh bởi ít nhất 2 bức tường*. Ví dụ ở hình 6a
- **2x2 block deadlock**: là trạng thái mà các hộp không ở đích bị *xếp thành một khối lớn 2x2* dẫn đến không thể di chuyển thêm bất kỳ khối nào. Ví dụ ở hình 6b
- **Edge deadlock**: là trạng thái tồn tại một hộp ở vị trí sát tường, chỉ có thể duy chuyển trên một *trục* mà không tồn tại bất kỳ đích nào trong *trục* đó. Ví dụ ở hình 7

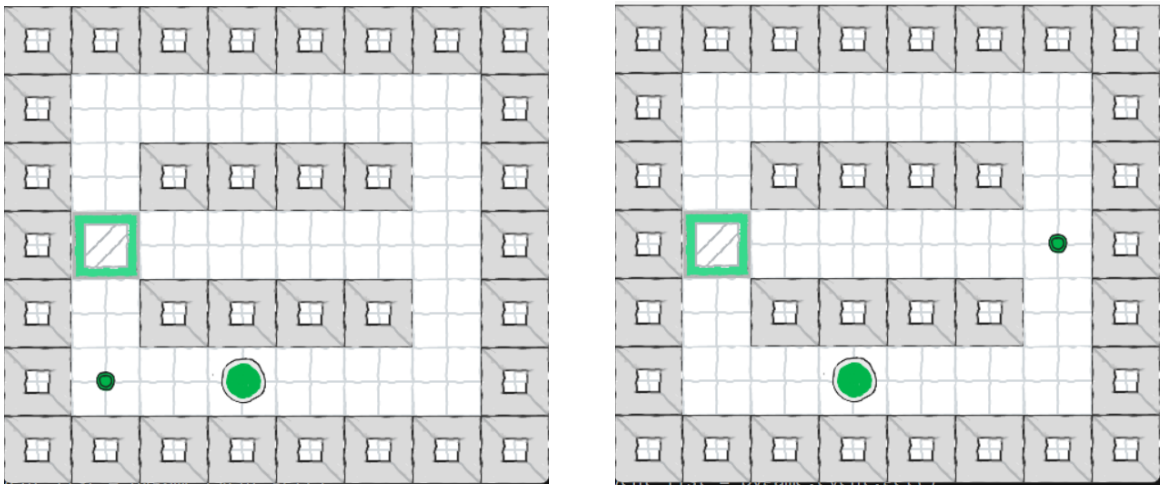


(a) Corner Deadlock



(b) 2x2 Block Deadlock

Hình 6: Ví dụ cho Corner và Block deadlock



(a) Không bị Edge deadlock (b) Bị Edge deadlock

Hình 7: Ví dụ cho Edge deadlock

Tóm lại: một nước đi được xem là hợp lệ khi đó là một nước đi sinh ra trạng thái mới và trạng thái mới đó không bị deadlock.

III Giải thuật

1 Breath First Search

1.1 Giới thiệu Giải thuật BrFS

Tìm kiếm theo chiều rộng (Breadth-First Search - BrFS) là một giải thuật tìm kiếm cơ bản, thuộc nhóm **tìm kiếm mù** (uninformed search). Đặc điểm cốt lõi của giải thuật này là nó không sử dụng bất kỳ thông tin ước lượng (heuristic) nào để định hướng về phía trạng thái đích.

Thay vào đó, BrFS khám phá không gian trạng thái một cách có hệ thống theo từng lớp (level-by-level).

Nguyên lý hoạt động: BrFS bắt đầu từ trạng thái gốc (trạng thái ban đầu của bản đồ) và khám phá tất cả các trạng thái láng giềng (các trạng thái có thể đạt được bằng một bước di chuyển) trước khi chuyển sang khám phá các trạng thái ở độ sâu tiếp theo.

Cơ chế này đảm bảo rằng BrFS sẽ tìm thấy đường đi ngắn nhất (về số lượng bước di chuyển) từ trạng thái đầu đến trạng thái đích, nếu một đường đi như vậy tồn tại.

1.2 Hiện thực BrFS cho Bài toán Sokoban

1.2.1 Cấu trúc dữ liệu của BrFS

BrFS sử dụng hai cấu trúc dữ liệu cơ bản để quản lý quá trình tìm kiếm:

1. **Hàng đợi (Queue):** Để đảm bảo nguyên lý "khám phá theo từng lớp", BrFS sử dụng hàng đợi theo cơ chế "Vào trước - Ra trước" (FIFO). Trong code, nhóm sử dụng một danh sách (list) của Python cho mục đích này:
   - `queue = [init_state]`: Khởi tạo hàng đợi với trạng thái ban đầu.
   - `current_state = queue.pop(0)`: Lấy trạng thái ở *đầu* hàng đợi (trạng thái cũ nhất) để khám phá.
   - `queue.append(state)`: Thêm các trạng thái láng giềng mới vào *cuối* hàng đợi.
2. **Tập Đã Thăm (Visited Set):** Để tránh lãng phí thời gian khám phá lại các trạng thái đã thăm và tránh các chu trình vô hạn, nhóm sử dụng `set` để ghi lại.
   - `visited = set()`: Khởi tạo một tập hợp rỗng.
   - `visited.add(state)`: Thêm một đối tượng `SokobanState` vào tập hợp. (Điều này khả thi nhờ phương thức `__hash__` đã định nghĩa).
   - `if state in visited:`: Kiểm tra xem trạng thái đã tồn tại trong `visited` hay chưa với độ phức tạp $O(1)$.

1.2.2 Cắt tỉa (Pruning) - Phát hiện Deadlock

Hàm `state.is_deadlock()` được gọi trước khi thêm một trạng thái mới vào hàng đợi

```
for state in explored:
    generated_node += 1
    if state.is_deadlock():
        continue
    if state in visited:
        continue

    #...
    queue.append(state)
    visited.add(state)
```

Listing 2: Kiểm tra deadlock trong hàm `BrFS`

1.3 Mã giả giải thuật BrFS

Khởi tạo:

1. Khởi tạo hàng đợi: `queue = [s0]` (với `s0` là `SokobanState` ban đầu)
2. Khởi tạo tập đã thăm: `visited = set()`
3. Thêm trạng thái ban đầu: `visited.add(s0)`

Vòng lặp chính:

4. **While** `queue` không rỗng:
   (a) Lấy trạng thái đầu hàng đợi: `scurr = queue.pop(0)`
   (b) **For each** hành động `a` ∈ {`UP, DOWN, LEFT, RIGHT`}:
      i. Tạo trạng thái mới: `snext = scurr.move(a)`
      ii. **If** `snext` là `None` (di chuyển không hợp lệ): **Continue** (bỏ qua)
      iii. **If** `snext.is_deadlock()`: **Continue** (bỏ qua nhánh deadlock)
      iv. **If** `snext` **not in** `visited`:
         A. **If** `snext.is_win(goal)`: **Return** `snext.path` (tìm thấy lời giải)
         B. Thêm vào đã thăm: `visited.add(snext)`
         C. Thêm vào hàng đợi: `queue.append(snext)`
5. **Return** `None` (không tìm thấy lời giải)

1.4 Phân tích ưu/nhược điểm

Để phân tích độ phức tạp, nhóm sử dụng hai biến số quan trọng đặc trưng cho không gian tìm kiếm:

- b (Branching Factor): **Hệ số phân nhánh**. Đây là số lượng trạng thái con hợp lệ trung bình có thể sinh ra từ một trạng thái. Trong bài toán Sokoban, $b \leq 4$ (tương ứng với 4 hướng di chuyển).
- d (Depth): **Độ sâu của lời giải**. Đây là số bước di chuyển trong đường đi ngắn nhất (tối ưu) từ trạng thái đầu đến trạng thái đích.

Ưu điểm

1. **Tính đầy đủ:** BrFS đảm bảo luôn tìm thấy lời giải nếu lời giải tồn tại.
2. **Tính tối ưu:** BrFS đảm bảo luôn tìm thấy lời giải *tối ưu* (ngắn nhất về số bước di chuyển), vì nó luôn khám phá các trạng thái ở độ sâu d trước khi khám phá các trạng thái ở độ sâu $d+1$.

Nhược điểm

1. **Độ phức tạp không gian (Space Complexity): $O(b^d)$**
   Đây là nhược điểm lớn nhất. BrFS phải lưu trữ tất cả các trạng thái ở biên giới tìm kiếm (`queue`) và tất cả các trạng thái đã thăm (`visited`). Trong các bản đồ phức tạp (khi d lớn), bộ nhớ sẽ bị tiêu thụ theo cấp số nhân và nhanh chóng gây tràn bộ nhớ (dẫn tới Out of Memory).
2. **Độ phức tạp thời gian (Time Complexity): $O(b^d)$**
   Do không có thông tin định hướng (heuristic), BrFS phải duyệt qua tất cả b^d trạng thái (trong trường hợp xấu nhất) trước khi tìm thấy lời giải. Mặc dù việc phát hiện deadlock giúp cải thiện hiệu suất thực tế bằng cách cắt tỉa các nhánh vô dụng, độ phức tạp trong trường hợp xấu nhất vẫn không thay đổi.

2 A Star

2.1 Giới thiệu Giải thuật A* (A-Star)

Giải thuật **A* (A-Star)** là một thuật toán tìm kiếm có thông tin (*informed search*), khác biệt với BrFS - một thuật toán tìm kiếm mù (*uninformed search*). Trong khi BrFS duyệt các trạng thái theo thứ tự phát sinh (FIFO), A* sử dụng **tri thức heuristic** để định hướng quá trình tìm kiếm về phía trạng thái đích, giảm thiểu số lượng trạng thái cần khám phá.

Hàm đánh giá:

Mỗi trạng thái n trong A* được đánh giá bởi hàm chi phí tổng hợp:

$$f(n) = g(n) + h(n) \tag{2}$$

Trong đó:

- $g(n)$: Chi phí thực tế từ trạng thái ban đầu đến trạng thái n (số bước đã di chuyển);
- $h(n)$: Chi phí ước lượng (*heuristic*) từ trạng thái n đến trạng thái đích (số bước dự kiến còn lại);
- $f(n)$: Tổng chi phí ước lượng của đường đi hoàn chỉnh đi qua trạng thái n.

Nguyên lý hoạt động: A* luôn chọn trạng thái có giá trị $f(n)$ nhỏ nhất để mở rộng tiếp theo. Chiến lược này cho phép thuật toán tập trung vào các đường đi có khả năng dẫn đến lời giải tối ưu, loại bỏ các nhánh không hiệu quả trong không gian tìm kiếm.

2.2 Hiện thực A* cho Bài toán Sokoban

2.2.1 Hàm chi phí $g(n)$

Trong hiện thực của chúng tôi, $g(n)$ được định nghĩa là **số bước di chuyển** (độ dài của đường đi) từ trạng thái ban đầu đến trạng thái n. Cụ thể, với mỗi trạng thái `current_state`, giá trị $g(n)$ chính là:

$$g(n) = \text{len}(\texttt{current\_state.path}) \tag{3}$$

Mỗi khi người chơi thực hiện một bước di chuyển hợp lệ, $g(n)$ tăng lên 1 đơn vị.

2.2.2 Hàm Heuristic $h(n)$ - Hungarian Heuristic

Vấn đề:

Hàm heuristic $h(n)$ cần ước lượng chi phí còn lại để đưa *tất cả các thùng* từ vị trí hiện tại về các vị trí đích. Với k thùng và k đích, tồn tại $k!$ cách ghép cặp (thùng-đích) khác nhau, và việc xác định cách ghép nào cho chi phí tối thiểu là bài toán tổ hợp phức tạp.

Phương pháp giải quyết - Hungarian Heuristic:

Chúng tôi chuyển đổi bài toán thành **Bài toán phân công tối ưu** (*Optimal Assignment Problem*) và áp dụng **Giải thuật Hungarian** để tìm lời giải hiệu quả.

Bước 1: Xây dựng ma trận chi phí

Với mỗi cặp (thùng i, đích j), chi phí được ước lượng bằng **khoảng cách Manhattan**:

$$\text{cost}[i][j] = |x_{\text{box}_i} - x_{\text{goal}_j}| + |y_{\text{box}_i} - y_{\text{goal}_j}| \tag{4}$$

Khoảng cách Manhattan cung cấp ước lượng cận dưới về số bước tối thiểu để di chuyển thùng i đến đích j trong môi trường không có vật cản.

Bước 2: Giải bài toán phân công

Ta có ma trận chi phí $\text{cost}[k \times k]$. Mục tiêu là tìm một phép ghép cặp (bijection) giữa các thùng và các đích sao cho:

$$h(n) = \min_{\sigma \in S_k} \sum_{i=1}^{k} \text{cost}[i][\sigma(i)] \tag{5}$$

Trong đó S_k là tập tất cả các hoán vị của k phần tử.

Bước 3: Áp dụng Hungarian Algorithm

Giải thuật Hungarian có độ phức tạp $O(k^3)$ giúp tìm ra phép ghép tối ưu một cách hiệu quả. Trong code, chúng tôi sử dụng thư viện `scipy.optimize.linear_sum_assignment`:

```python
def A_star_h(boxes, goal, walls):
    Hungarian heuristic: Optimal assignment between boxes and goals
    if not boxes or not goal:
        return 0

    # Build cost matrix (Manhattan distance)
    cost_matrix = []
    for box in boxes:
        row = []
        for goal_pos in goal:
            manhattan_dist = abs(box[0] - goal_pos[0]) +
                             abs(box[1] - goal_pos[1])
            row.append(manhattan_dist)
        cost_matrix.append(row)

    # Apply Hungarian algorithm
    row_ind, col_ind = linear_sum_assignment(cost_matrix)

    # Sum up optimal assignment costs
    total_cost = sum(cost_matrix[i][j]
                     for i, j in zip(row_ind, col_ind))

    return total_cost
```

Listing 3: Hàm heuristic A* với Hungarian Algorithm

Tính chấp nhận được (Admissibility):

Một heuristic được gọi là *chấp nhận được* (admissible) nếu nó không bao giờ đánh giá quá cao chi phí thực tế:

$$h(n) \leq h^*(n) \quad \forall n \tag{6}$$

Trong đó $h^*(n)$ là chi phí thực tế từ n đến đích.

Chứng minh heuristic của chúng tôi là admissible:

1. Khoảng cách Manhattan giữa 2 điểm luôn $\leq$ khoảng cách đường đi thực tế (do phải đi vòng tường);
2. Chi phí phân công tối ưu (Hungarian) cho ta cận dưới chặt nhất có thể với thông tin hiện có;
3. Do đó: $h(n) \leq h^*(n)$ luôn đúng.

Hệ quả: A* với heuristic chấp nhận được đảm bảo tìm ra đường đi có độ dài tối ưu nếu lời giải tồn tại.

2.2.3 Cấu trúc dữ liệu riêng của A*

1. Hàng đợi ưu tiên (Priority Queue):

Khác với BrFS sử dụng `queue` (FIFO), A* *bắt buộc* phải dùng **hàng đợi ưu tiên** để luôn lấy ra trạng thái có $f(n)$ nhỏ nhất.

Trong Python, chúng tôi sử dụng module `heapq` để hiện thực min-heap:

```python
import heapq

# Initialize priority queue
open_list = []
heapq.heappush(open_list, (f_score, state_hash, initial_state))

# Always get state with minimum f(n)
while open_list:
    current_f, current_hash, current_state = heapq.heappop(open_list)
    # Process state...
```

Mỗi phần tử trong heap có dạng `(f_score, state_hash, state)`, trong đó:

– `f_score`: Giá trị $f(n) = g(n) + h(n)$ làm khóa sắp xếp;

– `state_hash`: Mã băm của trạng thái để tránh trùng lặp;

– `state`: Đối tượng trạng thái đầy đủ.

Độ phức tạp:

– `heappush`: $O(\log n)$

– `heappop`: $O(\log n)$

2. Tập VISITED với cập nhật chi phí:
Khác với BrFS chỉ lưu các trạng thái đã thăm dưới dạng `set`, A* cần lưu thêm **chi phí** $g(n)$ **tốt nhất** để đến mỗi trạng thái.

```
# Dictionary: state_hash -> best g(n) found so far
visited = {}

# When exploring a state
if state_hash not in visited or g_score < visited[state_hash]:
    visited[state_hash] = g_score
    # Add to open list for further exploration
    heapq.heappush(open_list, (f_score, state_hash, new_state))
```

Lý do: Trong A*, một trạng thái n có thể được khám phá nhiều lần thông qua các đường đi khác nhau. Nếu phát hiện đường đi mới với giá trị $g(n)$ nhỏ hơn, thuật toán cần *cập nhật* chi phí và đưa trạng thái đó vào hàng đợi lại.

Cơ chế này đảm bảo A* luôn duy trì đường đi tốt nhất đến mỗi trạng thái, từ đó đảm bảo tính tối ưu của lời giải cuối cùng.

2.3 Mã giả giải thuật A*

Khởi tạo:

1. Khởi tạo $g(s_0) = 0$, $h(s_0) = \text{A\_star\_h}(s_0)$, $f(s_0) = g(s_0) + h(s_0)$
2. Khởi tạo hàng đợi ưu tiên: open_list $= \{(f(s_0), \text{hash}(s_0), s_0)\}$
3. Khởi tạo tập đã thăm: visited $= \{\text{hash}(s_0) : g(s_0)\}$

Vòng lặp chính:

4. **While** open_list $\neq \emptyset$:
   (a) Lấy trạng thái có f nhỏ nhất: $(f_{\text{curr}}, h_{\text{curr}}, s_{\text{curr}}) = \text{heappop(open\_list)}$
   (b) **If** s_{curr} là trạng thái đích: **Return** đường đi
   (c) **If** $g(s_{\text{curr}}) > \text{visited}[h_{\text{curr}}]$: **Continue** (đã có đường tốt hơn)
   (d) **For each** hành động $a \in \{\text{UP, DOWN, LEFT, RIGHT}\}$:
      i. Tạo trạng thái mới: $s_{\text{next}} = s_{\text{curr}}.\text{move}(a)$
      ii. **If** $s_{\text{next}} = \text{None}$ hoặc s_{next} là deadlock: **Continue**
      iii. Tính chi phí:
         – $g_{\text{next}} = g(s_{\text{curr}}) + 1$
         – $h_{\text{next}} = \text{A\_star\_h}(s_{\text{next}})$
         – $f_{\text{next}} = g_{\text{next}} + h_{\text{next}}$
      iv. $h_{\text{next}} = \text{hash}(s_{\text{next}})$
      v. **If** $h_{\text{next}} \notin$ visited hoặc $g_{\text{next}} < \text{visited}[h_{\text{next}}]$:
         – Cập nhật: $\text{visited}[h_{\text{next}}] = g_{\text{next}}$
         – Thêm vào hàng đợi: $\text{heappush}(\text{open\_list}, (f_{\text{next}}, h_{\text{next}}, s_{\text{next}}))$
5. **Return None** (không tìm thấy lời giải)

2.4 Phân tích độ phức tạp

Độ phức tạp thời gian:

Trong trường hợp xấu nhất, A* có thể duyệt qua tất cả các trạng thái hợp lệ:

$$O(b^d \cdot \log(b^d)) = O(b^d \cdot d \log b) \tag{7}$$

Trong đó:

- b: Branching factor (số trạng thái con trung bình, trong Sokoban ≤ 4);
- d: Độ sâu của lời giải (số bước trong đường đi tối ưu);
- $\log(b^d)$: Chi phí cho các thao tác heap.

Tuy nhiên, nhờ heuristic tốt, A* thực tế chỉ duyệt một phần nhỏ không gian trạng thái:

$$O(b^{d \cdot \epsilon}) \tag{8}$$

Trong đó $\epsilon < 1$ phụ thuộc vào chất lượng heuristic ($\epsilon \to 0$ khi $h(n) \to h^*(n)$).

Độ phức tạp không gian:

A* cần lưu trữ:

- `open_list`: Các trạng thái đang chờ duyệt;
- `visited`: Tất cả các trạng thái đã khám phá cùng chi phí $g(n)$.

Trong trường hợp xấu nhất:

$$O(b^d) \tag{9}$$

Đây là hạn chế chính của A* so với các thuật toán tiết kiệm bộ nhớ như IDA*.

IV Đánh giá

1 Tiêu chí đánh giá

- **Thời gian (time taken)**: dựa trên số phép tính, quy mô bài toán, hoặc đo trực tiếp thời gian chạy qua nhiều lần thử
- **Không gian (memory used)**: không gian bộ nhớ cần dùng (số phần tử, số node, mảng,...)
- **Độ chính xác, hiệu quả (node expaneded, node generated, node revisited, efficiency)**: trong bài tập lớn này sử đo đạt hiệu quả bằng việc đếm số lần các node mở rộng, các node được sinh ra, các node sinh ra và trùng với một node cũ (đã đi qua) so với kết quả cuối cùng

  node expanded: số node trạng thái được duyệt qua để sinh các node mới trong cấu trúc dữ liệu.

  node generated: tổng số node được sinh ra từ các node trạng thái đã duyệt.

  node revisited: số node được sinh ra trùng với node trạng thái cũ

  $efficiency = result\ path\ length / expanded\ node$: tỉ lệ sinh ra một node trạng thái đúng.

2 Tiếp cận hướng deadlock

Việc thêm trạng thái deadlock giúp cắt giảm rất nhiều các nhánh trong cây trạng thái. Có thể thấy từ số liệu thu thập khi chạy testcase 1, 2 khi có và không có kiểm tra deadlock trong bảng 2 hiệu năng hoàn toàn khác biệt, thời gian, không gian tiêu tốn và hiệu quả tìm kiếm tốt hơn rất nhiều. Như trong bảng 3 việc phát hiện deadlock nâng cao hiệu năng giải thuật hơn $>$**1000%**.

A*	**Không kiểm tra deadlock**			**Kiểm tra deadlock**		
	Thời gian (s)	Không gian (MB)	Hiệu quả tìm kiếm (%)	Thời gian (s)	Không gian (MB)	Hiệu quả tìm kiếm (%)
Testcase 1	47.510	163.253	0.033	9.594	13.929	0.385
Testcase 2	2221.583	9332.252	0.004	189.898	392.464	0.097

Bảng 2: Số liệu giải testcase 1,2 theo giải thuật A*

A*	**Độ dài kết quả**	**Tỉ số (lần)**		
Testcase 1	49	4.952	11.720	11.750
Testcase 2	211	11.699	23.779	24.923
Trung bình		8.325510158	17.74949338	18.33653846

Bảng 3: So sánh kết quả kiểm tra deadlock

3 So sánh hai giải thuật

Xét lý thuyết A* sẽ giải quyết bài toán tốt hơn về mặt không gian do các quyết định được hàm heuristic hỗ trợ và đạt hiệu suất tốt hơn so với BrFS nếu quy mô bài toán lớn. Còn về thời gian bị ảnh hưởng bởi nhiều yếu tố, độ phức tạp của hàm heuristic, cấu trúc dữ liệu, cài đặt tối ưu...

Testcase	**BrFS_time (s)**	**BrFS_memory (MB)**	**BrFS_eff**	**A*_time (s)**	**A*_memory (MB)**	**A*_eff**
1	8.91853	15.626	0.003377	9.593721	13.929	0.003854
2	161.982764	418.836	0.000918	189.897973	392.464	0.000972
3	3.404521	8.056	0.022841	3.544431	7.622	0.023094
4	9.593736	17.907	0.007012	10.611476	17.842	0.007066
5	207.961415	404.175	0.000408	239.884976	404.264	0.00041
6	30.597758	54.352	0.001399	36.554209	52.128	0.001491
7	1.97446	4.679	0.033629	2.140787	4.673	0.033784
8	10.221651	18.122	0.006006	11.665228	16.798	0.00641
9	45.35375	115.339	0.003135	47.952101	115.882	0.003135
10	5.147473	13.155	0.012937	5.924235	12.942	0.012941

Bảng 4: Số liệu thu thập khi chạy bằng 2 giải thuật

Theo bảng 4, có thể thấy giải thuật **A*** nhờ hàm heuristic mà trong đa phần các trường hợp sẽ tiết kiệm được không gian bộ nhớ và cải thiện hiệu suất tìm kiếm (**≈5%**). Tuy nhiên cũng vì độ phức tạp của hàm heuristic cùng với triển khai, cài đặt phức tạp hơn so với **Tìm kiếm theo chiều rộng (BrFS)** mà thời gian tìm kiếm có phần lâu hơn trung bình (**≈12%**).

V Kết Luận

1 Tổng kết

Dự án **Sokoban Solver** đã hiện thực thành công hai thuật toán **Breadth-First Search (BrFS)** và **A* Search**, đồng thời tích hợp **bộ phát hiện deadlock cải tiến** để tối ưu không gian tìm kiếm. Hệ thống có khả năng tự động đọc bản đồ, giải bài toán, ghi log thống kê hiệu năng và hiển thị kết quả trực quan bằng **Pygame**. Kết quả thực nghiệm cho thấy:

- **BrFS** luôn tìm được lời giải tối ưu, song tốc độ giảm đáng kể khi bản đồ có độ phức tạp cao;
- **A*** với **Hungarian heuristic** giúp giảm đáng kể số node duyệt và thời gian thực thi so với BrFS;
- Việc áp dụng **deadlock detection** giúp tiết kiệm đáng kể số node duyệt trên các test case trung bình.

2 Đánh giá

Sự kết hợp giữa cơ chế phát hiện deadlock và heuristic đã giúp thuật toán A* đạt hiệu năng vượt trội mà vẫn đảm bảo tính chính xác của lời giải. Bên cạnh đó, giao diện Pygame cho phép người dùng trực quan hóa quá trình tìm kiếm, qua đó hỗ trợ đánh giá và phân tích thuật toán một cách sinh động và dễ hiểu hơn.

3 Hạn chế

- Chưa hỗ trợ xử lý đa luồng hoặc chạy song song;
- Hiệu suất vẫn suy giảm đáng kể khi bản đồ có nhiều hộp hoặc không gian mở rộng lớn;
- Hàm heuristic hiện mới dừng ở mức Hungarian cơ bản, chưa xét đến yếu tố chướng ngại vật hay chi phí di chuyển thực tế.

4 Hướng phát triển

Trong tương lai, nhóm hướng tới:

- Nâng cấp heuristic của A* bằng **pattern database** hoặc heuristic có trọng số (*weighted heuristic*);
- Tối ưu bộ nhớ bằng các kỹ thuật tìm kiếm giới hạn hoặc nén trạng thái như **IDDFS** hay **IDA***;
- TỐi ưu tốc độ xử lý bằng phương pháp đa luồng.
- Phát triển giao diện Pygame thân thiện hơn, cho phép người dùng tạo bản đồ ngẫu nhiên và tùy chỉnh tham số thuật toán.
- Nghiên cứu và hiện thực thêm một số thuật toán tìm kiếm khác như **DFS**, **Hill Climbing**.

5 Kết luận

Dự án đã hoàn thành các mục tiêu đề ra, minh chứng cho hiệu quả của việc kết hợp giữa heuristic mạnh và kỹ thuật cắt tỉa thông minh trong các bài toán tìm kiếm trạng thái lớn. Thông qua quá trình triển khai, thử nghiệm và đánh giá, nhóm đã hiểu sâu hơn về cơ chế hoạt động của các giải thuật tìm kiếm, đồng thời tích lũy kinh nghiệm trong việc áp dụng chúng vào một bài toán thực tế đầy thách thức như Sokoban.

Phụ lục

[1] Testcase được chạy trên cấu hình: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz (1.80 GHz), 12GB RAM

[2] Toàn bộ số liệu chạy testcase trong file: comparison.xlsx

[3] Github triển khai của nhóm: ZanissNguyen/Sokoban

[4] Assets lấy từ repo: dangarfield/Sokoban-solver

[5] Drive video báo cáo ở Đây

[6] Link Video youtube ở Đây

Tài Liệu Tham Khảo

[1] Tài liệu học tập của môn học

[2] http://en.wikipedia.org/wiki/Breadth-first_search

[3] https://en.wikipedia.org/wiki/A*_search_algorithm

[4] https://ksokoban.online/