TRƯỜNG ĐẠI HỌC BÁCH KHOA - ĐẠI HỌC QUỐC GIA TP.HCM
KHOA KHOA HỌC VÀ KĨ THUẬT MÁY TÍNH

NHẬP MÔN TRÍ TUỆ NHÂN TẠO (CO3061)

Bài Tập Lớn 1

BÀI TOÁN SOKOBAN

Lớp: L01
Giảng viên hướng dẫn: Vương Bá Thịnh

Sinh viên thực hiện: Nguyễn Đăng Dũng - 2210581
Nguyễn Quốc Thắng - 2213205
Trần Anh Khoa - 2211644
Trịnh Duy Nghiêm - 2312256
Văn Thái Quân - 2312861

Thành phố Hồ Chí Minh, 1/11/2025

Mục lục

1 Giới thiệu về bài toán

Sokoban (tiếng Nhật có nghĩa là "người giữ kho") là một trò chơi giải đố trong đó người chơi phải đẩy các thùng (`$`) trong một nhà kho đến các vị trí lưu trữ đã định trước (`.`).

Luật chơi: *Sokoban* được chơi trên một bản đồ dạng lưới. Trạng thái của trò chơi được xác định bởi vị trí của người chơi (`@`), các bức tường (`#`), các thùng (`$`) và các vị trí đích (`.`).

1. Người chơi có thể di chuyển theo bốn hướng: Lên, Xuống, Trái, Phải vào một ô trống.
2. Người chơi có thể **đẩy** một thùng, với điều kiện ô phía sau thùng đó phải trống (không phải là tường hoặc một thùng khác).
3. Người chơi **không thể kéo** thùng.
4. Người chơi **không thể đẩy** hai hay nhiều thùng cùng một lúc.
5. Trò chơi kết thúc (thắng) khi tất cả các thùng đều nằm trên các vị trí lưu trữ.

2 Giải thuật Blind search

Theo yêu cầu đề bài, nhóm hiện thực một giải thuật Blind search (DFS hoặc BrFS). Nhóm đã chọn hiện thực giải thuật **Breadth-First Search (BFS)**.

2.1 Giới thiệu về giải thuật BFS

Breadth-First Search (BFS) là một thuật toán tìm kiếm duyệt theo chiều rộng. Nó bắt đầu từ một trạng thái gốc và khám phá tất cả các trạng thái kề ở mức hiện tại trước khi chuyển sang các trạng thái ở mức tiếp theo. BFS sử dụng một hàng đợi (Queue) theo cơ chế FIFO (First-In, First-Out) và được đảm bảo tìm thấy đường đi ngắn nhất (về số bước) nếu lời giải tồn tại.

2.2 Sử dụng BFS cho bài toán Sokoban

- **Bước 1: Khởi tạo**
  Khởi tạo một hàng đợi `q` (trong code là `deque`) và thêm trạng thái ban đầu (vị trí ban đầu của người chơi và các thùng) vào `q`.
  Khởi tạo một tập `visited` để lưu các trạng thái đã thăm (một trạng thái được định nghĩa bởi vị trí người chơi và vị trí của tất cả các thùng). Thêm trạng thái ban đầu vào `visited`.
- **Bước 2: Lặp và Tìm kiếm**
  Trong khi hàng đợi `q` còn phần tử:
  - Lấy (dequeue) trạng thái `curr` từ đầu hàng đợi.
  - Kiểm tra `curr` có phải là trạng thái đích (goal) không (hàm `curr.reached_goal()`). Nếu đúng, trả về chuỗi hành động (`curr.get_move()`) và kết thúc.
  - Nếu không, tạo ra tất cả các trạng thái kề (neighbor) `e` từ `curr` (hàm `curr.get_neighbors()`).
  - Với mỗi trạng thái `e`, nếu `e` chưa có trong `visited`:
    * Thêm `e` vào `visited`.
    * Thêm (enqueue) `e` vào cuối hàng đợi `q`.

- **Bước 3: Kết thúc**
  Nếu hàng đợi `q` rỗng mà chưa tìm thấy trạng thái đích, kết luận bài toán không có lời giải và trả về danh sách rỗng.

2.3 Giải thuật (Pseudocode)

```
procedure BFS(start_state)
    q ← new Queue()
    visited ← new Set()
    q.enqueue(start_state)
    visited.add(start_state)
    while q is not empty do
        curr_state ← q.dequeue()
        if IsGoal(curr_state) then
            return curr_state.get_move_path()
        end if
        for neighbor_state in GetNeighbors(curr_state) do
            if neighbor_state not in visited then
                visited.add(neighbor_state)
                q.enqueue(neighbor_state)
            end if
        end for
    end while
    return NO_SOLUTION
end procedure
```

3 Giải thuật Heuristic Search

Nhóm hiện thực bài toán bằng giải thuật **A* Search**.

3.1 Giới thiệu về giải thuật A*

A* Search là một thuật toán tìm kiếm có thông tin, kết hợp giữa tìm kiếm theo chi phí nhỏ nhất (Uniform Cost Search) và tìm kiếm tham lam (Greedy Best-First Search). Thuật toán sử dụng hàm đánh giá $f(n) = g(n) + h(n)$ để quyết định mở rộng nút nào trước, trong đó:

- $g(n)$ là chi phí thực tế (số bước) đã thực hiện từ trạng thái ban đầu đến trạng thái n.
- $h(n)$ là hàm heuristic, ước lượng chi phí từ trạng thái n đến trạng thái đích.

A* sử dụng một hàng đợi ưu tiên (Priority Queue) và ưu tiên mở rộng trạng thái có $f(n)$ nhỏ nhất.

3.2 Sử dụng A* cho bài toán Sokoban

- **Bước 1: Khởi tạo**
  Khởi tạo một hàng đợi ưu tiên `pq` (trong code là `heapq`).
  Khởi tạo một từ điển `best_g` để lưu chi phí $g(n)$ tốt nhất tìm được cho mỗi trạng thái.
  Đặt `best_g[start_state] = 0`.

- **Bước 2: Định nghĩa Hàm $g(n)$ và $h(n)$**
  Hàm $g(n)$: Là chi phí thực tế, trong bài toán này chính là số bước di chuyển. Được tính bằng `g = len(curr.get_move())`.
  Hàm $h(n)$ (Heuristic): Nhóm đã hiện thực 2 hàm heuristic: `euclidean` và `manhatten`. Hàm heuristic chính được sử dụng là **Manhattan Distance**.
  Hàm heuristic $h(n)$ được tính bằng **tổng khoảng cách Manhattan** (tổng $|x_1 - x_2| + |y_1 - y_2|$) từ mỗi thùng (box) đến vị trí lưu trữ (storage) gần nhất mà chưa có thùng nào khác. Đây là một heuristic *admissible* (không đánh giá quá cao chi phí) vì nó bỏ qua vật cản là tường và các thùng khác.

- **Bước 3: Áp dụng A* để tìm kiếm lời giải**
  Duy trì hàng đợi ưu tiên `pq` sắp xếp theo $f(n)$.
  - Lấy trạng thái `curr` có $f(n)$ nhỏ nhất ra khỏi `pq`.
  - Nếu `curr` là trạng thái đích, trả về lời giải.
  - Với mỗi trạng thái kề `e` của `curr`:
  - Tính chi phí mới `g2 = g + 1`.
  - Nếu `e` chưa được thăm hoặc tìm thấy đường đi tốt hơn (`g2 < best_g[e]`):
    * Cập nhật `best_g[e] = g2`.
    * Tính $f = g2 + h(e)$.
    * Thêm `e` vào `pq` với độ ưu tiên `f`.

- **Bước 4: Kết thúc**
  Nếu `pq` rỗng mà chưa tìm được lời giải, kết luận bài toán không có đáp án.

3.3 Giải thuật (Pseudocode)

```
1:  procedure AStar(start_state)
2:      pq ← new PriorityQueue()                          ▷ Sắp xếp theo f(n)
3:      best_g ← new Map()                                ▷ Lưu g(n) tốt nhất cho mỗi state
4:      best_g.set(start_state, 0)
5:      f_start ← 0 + Heuristic(start_state)
6:      pq.push(start_state, f_start)
7:      while pq is not empty do
8:          curr_state ← pq.pop_lowest_f()
9:          if IsGoal(curr_state) then
10:             return curr_state.get_move_path()
11:         end if
12:         g_curr ← best_g.get(curr_state)
13:         for neighbor_state in GetNeighbors(curr_state) do
14:             g_new ← g_curr + 1                        ▷ Chi phí di chuyển là 1
15:             if (neighbor_state not in best_g) or (g_new < best_g.get(neighbor_state)) then
16:                 best_g.set(neighbor_state, g_new)
17:                 f_new ← g_new + Heuristic(neighbor_state)
18:                 pq.push(neighbor_state, f_new)
19:             end if
20:         end for
21:     end while
22:     return NO_SOLUTION
23: end procedure
```

4 Độ phức tạp của BFS và A* trong Sokoban

Giả sử b là "nhân tố rẽ nhánh"(branching factor - số lượng hành động hợp lệ trung bình từ một trạng thái) và d là độ sâu của lời giải (số bước để đến đích).

4.1 Độ phức tạp của BFS trong Sokoban

Breadth-First Search (BFS) phải duyệt tất cả các trạng thái ở độ sâu $d-1$ trước khi tìm thấy lời giải ở độ sâu d.

- **Độ phức tạp (Thời gian):** $O(b^d)$.
- **Tiêu tốn bộ nhớ:** $O(b^d)$. BFS phải lưu trữ tất cả các nút lá của cây tìm kiếm trong hàng đợi và tập `visited`.

4.2 Độ phức tạp của A* trong Sokoban

A* sử dụng hàm heuristic để ưu tiên tìm kiếm, giúp cắt tỉa đáng kể không gian trạng thái.

- **Hàm đánh giá:** $f(n) = g(n) + h(n)$
- **Độ phức tạp (Thời gian):**
  - **Trường hợp xấu nhất:** (với heuristic $h(n) = 0$), độ phức tạp tương tự BFS, là $O(b^d)$.
  - **Trường hợp thực tế (với heuristic tốt):** A* hiệu quả hơn BFS rất nhiều, $O(b^d)$ nhưng với b nhỏ hơn đáng kể.
- **Tiêu tốn bộ nhớ:** $O(b^d)$, A* vẫn phải lưu trữ rất nhiều trạng thái trong hàng đợi ưu tiên, đây là nhược điểm chính của A*.

4.3 So sánh tổng quan

Thuật toán	Độ phức tạp (Thời gian)	Độ phức TẠP (Bộ nhớ)	Tính tối ưu?
BFS	$O(b^d)$	$O(b^d)$	Có (Tìm ra lời giải ngắn nhất)
A*	$O(b^d)$ (Nhanh hơn BFS trong thực tế)	$O(b^d)$ (Thường tốn nhiều bộ nhớ)	Có (Nếu $h(n)$ là admissible)

Bảng 1: So sánh độ phức tạp của BFS và A*

Kết luận: A* với heuristic Manhattan nhanh hơn đáng kể so với BFS trong việc giải Sokoban, vì nó tránh được việc khám phá các trạng thái "vô vọng"(thùng bị đẩy vào góc xa). BFS, mặc dù cũng tìm ra lời giải tối ưu, nhưng lại tốn thời gian và bộ nhớ hơn rất nhiều do duyệt mù quáng.

5 Bảng số liệu thực nghiệm

Test Case	Thuật toán	Thời gian (ms)	Số bước
Test 1 (Map Dễ)	BFS	12.70	10
	A* (Manhattan)	3.48	10
Test 2 (Map TB)	BFS	161.74	27
	A* (Manhattan)	48.31	27
Test 3 (Map Khó)	BFS	2099.29	39
	A* (Manhattan)	550.90	39

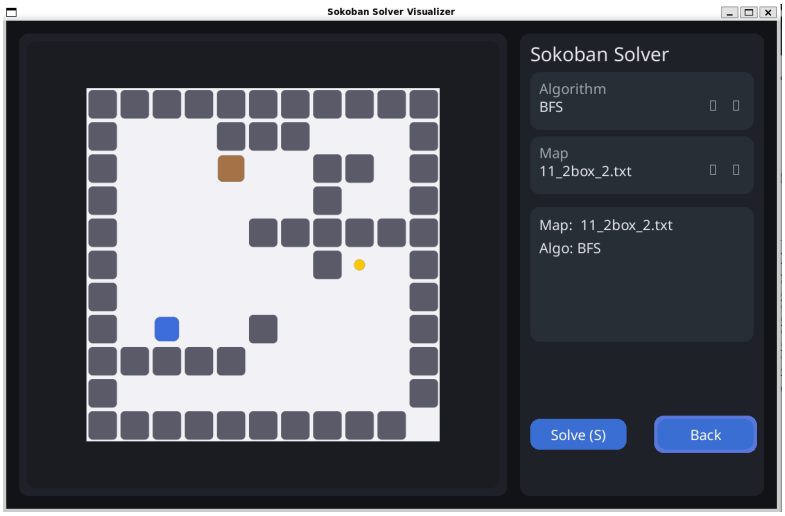
Bảng 2: So sánh hiệu năng BFS và A* trên các test case

5.1 Giải thích bảng số liệu

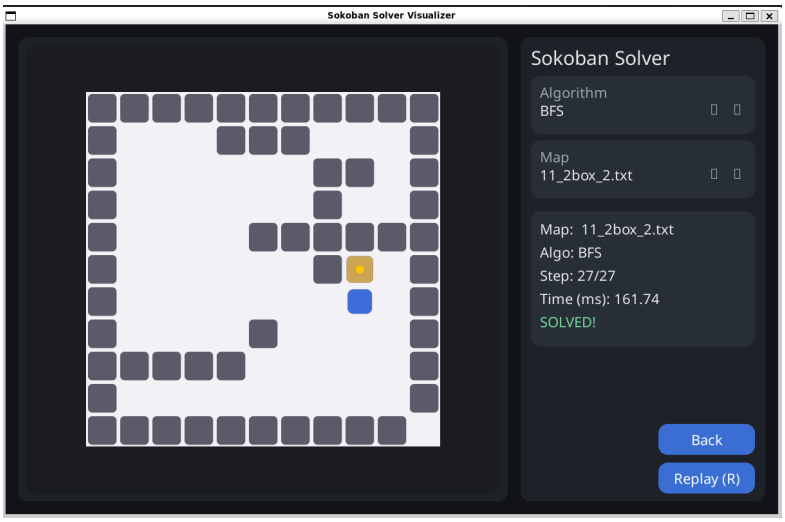
Dựa trên kết quả thực nghiệm, ta có thể rút ra những nhận xét rõ ràng về hiệu năng của hai thuật toán:

1. **Cả hai thuật toán đều tối ưu:** Trong cả ba test case, cột **Độ dài lời giải** là như nhau cho cả BFS và A* (cùng 10, 27 và 39 bước). Điều này chứng minh rằng cả hai thuật toán đều tìm ra được lời giải tối ưu (ngắn nhất), đúng với lý thuyết (BFS luôn tìm ra đường đi ngắn nhất, và A* với heuristic *admissible* như Manhattan cũng vậy).
2. **A* hiệu quả hơn BFS đáng kể:** Mặc dù tìm ra cùng một kết quả, A* Search với heuristic Manhattan luôn cho **Thời gian (ms)** nhanh hơn nhiều so với BFS.
   - Ở Test 1 (10 bước, Dễ), A* (3.48 ms) nhanh hơn BFS (12.70 ms) khoảng **3.65 lần**.
   - Ở Test 2 (27 bước, TB), A* (48.31 ms) nhanh hơn BFS (161.74 ms) khoảng **3.35 lần**.
   - Ở Test 3 (39 bước, Khó), A* (550.90 ms) nhanh hơn BFS (2099.29 ms) khoảng **3.81 lần**.
3. **Nguyên nhân của sự khác biệt (Độ phức tạp thuật toán):** Sự chênh lệch về thời gian này được giải thích bằng chính bản chất của hai thuật toán:
   - **BFS** là một giải thuật "mù"(Blind Search). Để tìm được lời giải ở độ sâu $d = 39$, nó *phải* duyệt qua *tất cả* các trạng thái ở độ sâu $d = 38$ trước. Điều này lãng phí cực kỳ nhiều thời gian và bộ nhớ để khám phá hàng triệu trạng thái "vô vọng"(ví dụ: đẩy thùng vào góc chết).
   - **A*** là một giải thuật "có thông tin"(Heuristic Search). Bằng cách sử dụng hàm heuristic (Manhattan distance), nó có khả năng "biết đường"và ưu tiên khám phá các trạng thái *có vẻ* tốt hơn (các trạng thái mà thùng ở gần đích hơn). Nó không lãng phí thời gian vào các nhánh tìm kiếm xấu, giúp "cắt tỉa"(prune) phần lớn không gian trạng thái.
4. **Kết luận về xu hướng (Bùng nổ tổ hợp):** Khi độ khó của bài toán tăng tuyến tính (từ 10 lên 27, rồi 39 bước), chênh lệch tuyệt đối về thời gian chạy tăng **gần như theo hàm mũ** (từ 9ms, lên 113ms, rồi vọt lên 1548ms). Điều này cho thấy rõ hiện tượng "bùng nổ tổ hợp"($O(b^d)$) của BFS. Với các màn chơi khó hơn nữa, BFS sẽ trở nên hoàn toàn không khả thi, trong khi A* vẫn tìm ra lời giải một cách hiệu quả.

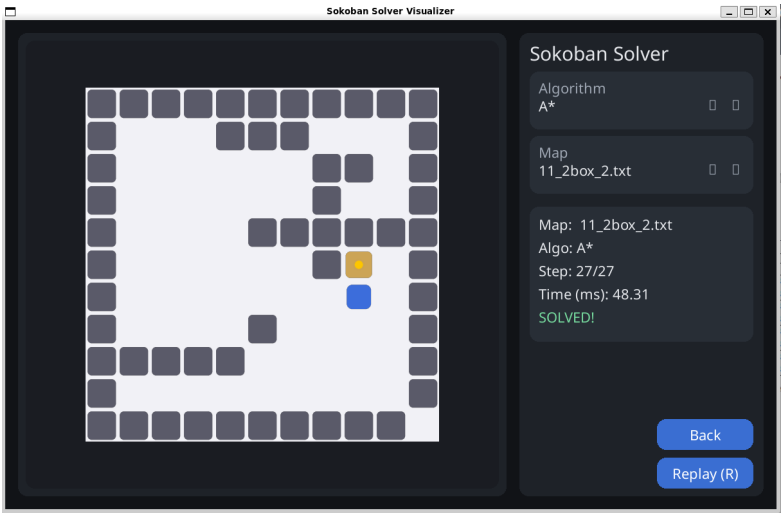
6 Hình ảnh Demo



Hình 1: Trạng thái ban đầu của màn chơi (Test 2 - Map TB)



Hình 2: Kết quả giải màn chơi bằng thuật toán BFS (Test 2)



Hình 3: Kết quả giải màn chơi bằng thuật toán A* (Test 2)