TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

------o0o------

MÔN HỌC

CƠ SỞ TRÍ TUỆ NHÂN TẠO

LAB 1:

SEARCHING

Giảng viên hướng dẫn: Bùi Duy Đăng

Nguyễn Thanh Tình

Tháng 11 năm 2024

MỤC LỤC

MỤC LỤC BẢNG

MỤC LỤC HÌNH

THÔNG TIN NHÓM & BẢNG PHÂN CÔNG CÔNG VIỆC

1. Thông tin thành viên

STT	MSSV	Họ và tên	Email
1	20120594	Nguyễn Duy Tiến	20120594@student.hcmus.edu.vn
2	21120539	Trần Minh Quang	21120539@student.hcmus.edu.vn
3	22120034	Lương Thị Kim Chi	22120034@student.hcmus.edu.vn
4	22120094	Lê Bảo Hồng Hạnh	22120094@student.hcmus.edu.vn

2. Bảng phân công công việc

STT	Nội dung công việc	Người thực hiện	Mức độ hoàn thành yêu cầu
1	Triển khai thuật toán BFS	Trần Minh Quang	100%
2	Triển khai thuật toán DFS	Trần Minh Quang	100%
3	Triển khai thuật toán UCS	Lê Bảo Hồng Hạnh	100%
4	Triển khai thuật toán A*	Lương Thị Kim Chi	100%
5	Viết 10 test cases (tệp input)	Lê Bảo Hồng Hạnh	100%
6	Thiết kế GUI	Nguyễn Duy Tiến	100%
7	Quay video demo	Nguyễn Duy Tiến	100%
8	Viết báo cáo	Lương Thị Kim Chi Lê Bảo Hồng Hạnh	100%

BẢNG ĐÁNH GIÁ YÊU CẦU DỰ ÁN

STT	Yêu cầu	Mức độ hoàn thành yêu cầu
1	Triển khai chính xác thuật toán BFS	100%
2	Triển khai chính xác thuật toán DFS	100%
3	Triển khai chính xác thuật toán UCS	100%
4	Triển khai chính xác thuật toán A*	100%
5	Viết 10 test cases (tệp input) với thuộc tính khác nhau	100%
6	Kết quả (tệp output và GUI)	100%
7	Quay video trình bày các thuật toán với một số test case	100%
8	Báo cáo thuật toán và thử nghiệm	100%

I. Giới thiệu sơ lược về bài toán đẩy đá trong mê cung

Bài toán **đẩy đá trong mê cung** là một bài toán thuộc dạng bài toán lập kế hoạch di chuyển (motion planning), trong đó một nhân vật **(Ares)** phải di chuyển các viên đá đến những vị trí đích **(công tắc)**. Mê cung bao gồm các bức tường, các viên đá và các công tắc đích, với các quy tắc di chuyển hạn chế, khiến cho bài toán trở nên phức tạp và thách thức.

Mô tả bài toán

1. **Mê cung**:

   Mê cung là một lưới hình chữ nhật kích thước *n x m* gồm nhiều ô vuông.
   Mỗi ô có thể là:
   - **Tường "#":** Không thể đi qua.
   - **Ares "@"**
   - **Ô trống " ":** Ares có thể đứng hoặc di chuyển qua.
   - **Đá "$":** Đối tượng mà Ares cần đẩy đến đích (công tắc).
   - **Công tắc ".":** Vị trí mà Ares cần đẩy đá đến.
   - **Đá trên công tắc "*"**
   - **Ares trên công tắc "+"**
2. **Ares**:
   - Ares có thể di chuyển tự do theo 4 hướng (lên, xuống, trái, phải) trong mê cung nếu các ô đó không phải là tường hoặc đá (trừ khi đá có thể đẩy).
   - Ares chỉ có thể **đẩy đá** chứ không thể kéo đá. Để đẩy, viên đá phải ở ngay cạnh Ares và phía sau của đá (theo hướng đẩy) phải là ô trống hoặc là công tắc. Không thể đẩy nhiều viên đá cùng lúc.
3. **Viên đá**:
   - Các viên đá nằm rải rác trong mê cung.
   - Nhiệm vụ của Ares là đẩy các viên đá vào các ô công tắc đích.
   - Một viên đá chỉ có thể được đẩy nếu phía sau nó (theo hướng đẩy) là một ô trống hoặc công tắc.
4. **Mục tiêu**:
   - Đẩy tất cả các viên đá đến vị trí của các công tắc đích.
   - Khi tất cả các viên đá đều ở trên các công tắc đích, bài toán được xem là hoàn thành.

Các phương pháp giải quyết

Bài toán này có thể được giải bằng các thuật toán tìm kiếm, như:

- **BFS (Tìm kiếm theo chiều rộng)**: Đảm bảo tìm ra giải pháp nếu có, nhưng không phải lúc nào cũng tối ưu và có thể tốn nhiều bộ nhớ.
- **DFS (Tìm kiếm theo chiều sâu)**: Có thể tìm ra giải pháp nhanh chóng nhưng dễ bị rơi vào ngõ cụt và không đảm bảo tìm giải pháp tối ưu.
- **UCS**: Mở rộng nút (trạng thái) có chi phí nhỏ nhất trước. Nếu có nhiều đường đi đến cùng một trạng thái, UCS chỉ giữ lại đường đi có chi phí thấp nhất.
- ***A (A-star):** Sử dụng heuristic để ước lượng chi phí còn lại, giúp dẫn dắt tìm kiếm hiệu quả hơn và có thể tìm giải pháp tối ưu.

Ứng dụng

Bài toán đẩy đá trong mê cung không chỉ là một bài toán giải trí mà còn mang tính ứng dụng cao trong các lĩnh vực như:

- **Robot học**: Lập kế hoạch di chuyển cho robot trong môi trường phức tạp.
- **Trí tuệ nhân tạo**: Bài toán điển hình cho việc tìm kiếm đường đi và tối ưu hóa.
- **Thiết kế trò chơi**: Sokoban là nền tảng của nhiều trò chơi giải đố, nơi người chơi phải tìm cách đẩy các đối tượng trong môi trường phức tạp.

II. Các thuật toán triển khai

2.1. Thuật toán BFS (Breadth-First Search)

2.1.1. Mô tả thuật toán

BFS là thuật toán duyệt đồ thị theo lớp, đảm bảo rằng tất cả các vị trí có thể đạt được từ điểm bắt đầu sẽ được thăm theo khoảng cách ngắn nhất. Thuật toán này sử dụng một hàng đợi FIFO để lưu trữ các trạng thái, lần lượt duyệt qua các trạng thái lân cận cho đến khi đạt được mục tiêu.

2.1.2. Triển khai giải thuật

a. Khởi tạo

- **Biến queue**: Sử dụng hàng đợi deque để lưu trữ các trạng thái bao gồm vị trí của Ares, vị trí các viên đá, đường đi hiện tại, tổng trọng lượng đã đẩy và số bước đã thực hiện.
- **Biến visited**: Sử dụng tập visited để lưu các trạng thái đã thăm (tránh lặp lại trạng thái).

b. Vòng lặp chính

Trong mỗi vòng lặp, thuật toán sẽ thực hiện các bước sau:

1. **Lấy trạng thái hiện tại từ hàng đợi**.
2. **Tìm các nước đi hợp lệ** từ vị trí hiện tại của Ares, bao gồm cả di chuyển và đẩy đá.
3. **Cập nhật hàng đợi** với các trạng thái mới chưa thăm, bao gồm vị trí mới của Ares, các viên đá, hành động thực hiện, trọng lượng, và số bước đi.

Mô tả chi tiết từng bước:

- **Di chuyển**: Nếu ô kế tiếp là trống hoặc là công tắc, Ares có thể di chuyển vào ô đó.
- **Đẩy đá**: Nếu gặp đá, kiểm tra ô kế tiếp có trống hay không hoặc không có đá khác. Nếu thỏa mãn, cập nhật vị trí mới của đá và tăng trọng lượng tổng.

c. Điều kiện dừng

- Sau mỗi lần cập nhật trạng thái, nếu tất cả các viên đá đã vào vị trí công tắc (switches), BFS dừng lại và trả về kết quả.
- Nếu không tìm được kết quả, thuật toán cũng kết thúc và trả về `None`.

2.1.3. Hàm get_possible_moves

Hàm **get_possible_moves** chịu trách nhiệm xác định các hành động hợp lệ từ trạng thái hiện tại, bao gồm:

- **Kiểm tra các ô kế bên vị trí Ares:** để xem có thể di chuyển hoặc đẩy đá hay không.
- **Xác định hành động**: Nếu Ares có thể di chuyển, thêm vào hàng đợi trạng thái mới với ký hiệu di chuyển. Nếu đẩy đá, thêm trạng thái mới với ký tự đẩy (kèm trọng lượng đá).

Mô tả các bước:

- **Di chuyển**: Các bước r, d, l, u (phải, xuống, trái, lên) mô tả việc Ares di chuyển giữa các ô.
- **Đẩy đá**: Khi đẩy đá, trạng thái mới lưu trữ vị trí mới của đá cùng với hành động đẩy tương ứng (R, D, L, U).

2.1.4. Kết quả và đánh giá

1. Kết quả trả về

Bao gồm:

- **Số bước đi**: Tổng số bước đã thực hiện.
- **Trọng lượng tổng**: Tổng chi phí (di chuyển và trọng lượng đá đã đẩy).
- **Số nút:** Số lượng nút được tạo ra bởi thuật toán
- **Thời gian**: Thời gian cần thiết để hoàn tất tìm kiếm.
- **Bộ nhớ sử dụng**: Tính toán tổng bộ nhớ sử dụng bởi visited và queue.
- **Đường đi**: Chuỗi ký tự thể hiện đường đi từ vị trí bắt đầu đến khi tất cả các viên đá đã vào công tắc.

2. Đánh giá hiệu quả

- **Thời gian thực hiện**: BFS có hiệu quả cao cho bài toán tìm đường đi ngắn nhất.
- **Sử dụng bộ nhớ**: Lưu trữ trạng thái thăm viếng giúp giảm thiểu việc lặp trạng thái, tối ưu bộ nhớ.

2.2. Thuật toán DFS (Depth-First Search)

2.2.1. Mô tả thuật toán

DFS là một thuật toán tìm kiếm theo chiều sâu, hoạt động dựa trên việc khám phá càng sâu càng tốt trên mỗi nhánh của cây tìm kiếm trước khi quay trở lại và khám phá các nhánh khác.

Cách hoạt động của DFS trong bài toán này:

1. Bắt đầu từ vị trí của **Ares** (ký hiệu `@`), DFS sẽ chọn một hướng để di chuyển.
2. Nếu gặp một viên đá `$`, DFS kiểm tra xem có thể đẩy đá đến ô trống hoặc công tắc `.` hay không. Nếu có, Ares sẽ đẩy viên đá và tiếp tục khám phá vị trí mới.
3. Khi gặp đường cụt (không thể di chuyển), DFS sẽ quay lại và khám phá các nhánh khác chưa được thăm.
4. DFS sẽ tiếp tục cho đến khi tìm thấy giải pháp (tất cả các viên đá đã được đặt trên công tắc) hoặc không còn đường nào để đi.

2.2.2. Triển khai giải thuật

a. Khởi tạo:

- **Ngăn xếp (stack)** được khởi tạo với trạng thái bắt đầu: vị trí của **Ares**, vị trí của các viên đá, hành động trống, và trọng lượng ban đầu là 0.
- **Tập hợp visited** lưu trữ các trạng thái đã duyệt để tránh vòng lặp vô tận.

b. Vòng lặp chính:

1. **Lấy trạng thái hiện tại từ ngăn xếp.** Trạng thái này bao gồm:
    a. **current_pos**: Vị trí hiện tại của Ares.
    b. **current_rocks**: Vị trí của các viên đá.
    c. **path**: Chuỗi hành động (các di chuyển và đẩy đá).
    d. **total_weight**: Tổng trọng lượng của các lần di chuyển và đẩy đá.
2. **Kiểm tra điều kiện thành công**: Nếu tất cả các viên đá đã được đặt lên các công tắc, thuật toán dừng lại và trả về đường đi.
   Duyệt các bước di chuyển hợp lệ: Sử dụng hàm get_possible_moves() để lấy danh sách các bước di chuyển hợp lệ từ vị trí hiện tại của Ares.
    a. Kiểm tra các vị trí kế bên Ares xem có thể di chuyển được không.
    b. Nếu có thể đẩy đá, cập nhật vị trí của các viên đá và thêm trạng thái mới vào ngăn xếp.

3. **Cập nhật trạng thái mới**: Mỗi khi Ares di chuyển hoặc đẩy đá thành công, trạng thái mới được thêm vào ngăn xếp để tiếp tục khám phá.
4. **Tránh lặp lại**: Nếu trạng thái mới đã được thăm, nó sẽ bị bỏ qua để tránh lặp lại vòng lặp vô tận.

c. Kết thúc:

- Sau mỗi lần cập nhật trạng thái, nếu tất cả các viên đá đã vào vị trí công tắc (switches), DFS dừng lại và trả về kết quả.
- Nếu không tìm được giải pháp, thuật toán trả về `None`.

2.2.3. Đánh giá

- **Không tối ưu**: DFS không đảm bảo tìm được giải pháp tối ưu (ví dụ, giải pháp ngắn nhất) vì nó có thể tìm ra một giải pháp ở độ sâu lớn hơn mặc dù có thể có giải pháp ngắn hơn ở nhánh khác.
- **Không hoàn chỉnh**: Trong trường hợp không có giới hạn về độ sâu, DFS có thể không bao giờ kết thúc nếu nó đi sâu vào một nhánh vô hạn.
- **Không khả thi với trạng thái lớn**: DFS có thể yêu cầu rất nhiều bộ nhớ trong trường hợp không giới hạn về độ sâu.

2.3. Thuật toán UCS (Uniform Cost Search)

2.3.1. Mô tả thuật toán

Thuật toán UCS là một thuật toán duyệt, tìm kiếm trên một cấu trúc cây, hoặc đồ thị có trọng số (chi phí), trong đó mỗi hành động có một chi phí và thuật toán luôn ưu tiên các hành động có chi phí thấp nhất. UCS sử dụng hàng đợi ưu tiên (priority queue) để duyệt qua các trạng thái và sẽ dừng lại khi đạt được trạng thái đích với chi phí tối ưu.

2.3.2. Triển khai giải thuật

a. Khởi tạo

- **Hàng đợi ưu tiên (priority queue)**: Dùng để lưu các trạng thái và đảm bảo trạng thái có chi phí thấp nhất được duyệt trước.
- **Biến priority_queue**: Mỗi phần tử trong hàng đợi là một tuple gồm tổng trọng lượng, vị trí hiện tại của Ares, vị trí các viên đá, chuỗi hành động, bước đi.
- **Biến visited**: Sử dụng một tập visited để lưu các trạng thái đã thăm (tránh lặp lại các trạng thái).

b. Các bước thực hiện

1. **Khởi tạo hàng đợi ưu tiên** với trạng thái ban đầu.
2. **Lặp qua hàng đợi ưu tiên:** mỗi lần lấy ra trạng thái có chi phí thấp nhất.
3. **Kiểm tra điều kiện dừng:** Nếu tất cả các viên đá đã vào vị trí công tắc, thuật toán trả về đường đi tối ưu và kết thúc.
4. **Duyệt qua các hành động hợp lệ** từ trạng thái hiện tại, bao gồm di chuyển và đẩy đá.

Mô tả các hành động:

- **Di chuyển**: Nếu Ares có thể di chuyển đến một ô trống hoặc công tắc sẽ thêm trạng thái mới vào hàng đợi ưu tiên mà không thay đổi tổng trọng lượng.
- **Đẩy đá**: Nếu Ares gặp đá, kiểm tra ô phía sau đá có trống hoặc là công tắc hay không. Nếu có thể đẩy, Ares đẩy viên đá vào ô đó, cập nhật trọng lượng tổng và thêm trạng thái mới vào hàng đợi ưu tiên.

Tối ưu hoá:

Trạng thái mới chỉ được thêm vào hàng đợi nếu chưa có trong visited, giúp giảm bớt số lượng trạng thái duyệt lặp lại.

c. Điều kiện dừng

- Sau mỗi lần cập nhật trạng thái, nếu tất cả các viên đá đã vào vị trí công tắc (switches), UCS dừng lại và trả về kết quả.
- Nếu không tìm được kết quả, thuật toán cũng kết thúc và trả về `None`.

2.3.3. Kết quả và đánh giá

1. Kết quả trả về

Bao gồm:

- **Số bước đi**: Tổng số bước đã thực hiện.
- **Trọng lượng tổng**: Tổng chi phí (di chuyển và trọng lượng đá đã đẩy).
- **Số nút:** Số lượng nút được tạo ra bởi thuật toán
- **Thời gian**: Thời gian cần thiết để hoàn tất tìm kiếm.
- **Bộ nhớ sử dụng**: Tính toán tổng bộ nhớ sử dụng bởi visited và queue.
- **Đường đi**: Chuỗi ký tự thể hiện đường đi từ vị trí bắt đầu đến khi tất cả các viên đá đã vào công tắc.

2. Đánh giá hiệu quả

- **Tính chính xác**: UCS đảm bảo tìm ra đường đi tối ưu nhất về mặt tổng chi phí đẩy đá.
- **Hiệu suất**: Tùy thuộc vào cấu trúc mê cung và vị trí các viên đá, UCS có thể tạo ra nhiều trạng thái và tiêu tốn bộ nhớ nếu trạng thái có nhiều khả năng mở rộng.
- **Bộ nhớ**: Lượng bộ nhớ sử dụng phụ thuộc vào số lượng trạng thái duyệt qua và khả năng lưu trữ trong hàng đợi ưu tiên.

2.4. Thuật toán A*

2.4.1. Mô tả thuật toán

Thuật toán A* là một trong những thuật toán tìm kiếm đường đi tốt nhất khi xét đến cả chi phí di chuyển và ước lượng khoảng cách đến đích. A* sử dụng hàm đánh giá tổng quát f(n) = g(n) + h(n) trong đó:

- g(n) là chi phí đã đi từ điểm bắt đầu đến điểm n.
- h(n) là ước lượng chi phí từ điểm n đến đích (hàm heuristic).
- f(n) là tổng chi phí dự kiến khi đi qua n.

2.4.2. Triển khai giải thuật

Bước 1: Khởi tạo các cấu trúc dữ liệu

- **Open list**: Đây là hàng đợi ưu tiên lưu trữ các trạng thái cần mở rộng, các trạng thái đã được sinh ra nhưng chưa được xét đến. Trong đó, phần tử có độ ưu tiên cao nhất là phần tử tốt nhất, mức độ ưu tiên dựa trên giá trị của hàm f (với f = g + h)
- **Closed list**: Danh sách các trạng thái đã được xét đến. Cần lưu trữ các trạng thái này để tránh kiểm tra lại các trạng thái đã xét.
- Khởi tạo tọa độ của Ares, vị trí các viên đá và các công tắc trong mê cung.

Bước 2: Khởi tạo trạng thái đầu

Ban đầu, thuật toán sẽ:

- Đặt trạng thái của Ares tại vị trí bắt đầu.
- Khởi tạo vị trí của các viên đá.
- Đặt trọng lượng ban đầu là 0.
- Đưa trạng thái ban đầu vào **open list** với chi phí dự tính

Bước 3: Chọn trạng thái có f(n) nhỏ nhất

- Sử dụng hàng đợi ưu tiên (min-heap) để chọn trạng thái với giá trị f(n) nhỏ nhất.
- Lấy trạng thái này ra để xử lý và đưa vào **closed list**.

Bước 4: Tính toán các bước di chuyển hợp lệ

Tại mỗi trạng thái:

- Thuật toán sẽ kiểm tra các hướng di chuyển của Ares (lên, xuống, trái, phải).

- Nếu gặp đá, kiểm tra xem đá có thể đẩy được không (có khoảng trống hoặc công tắc phía sau đá).
- Các trạng thái di chuyển hoặc đẩy đá hợp lệ sẽ được tính toán và thêm vào **open list**.

Bước 5: Cập nhật hàm g và h

- Cập nhật g(n) – chi phí đã đi từ điểm bắt đầu đến điểm hiện tại (bao gồm cả trọng lượng đã đẩy đá).
- Sử dụng một hàm heuristic h(n) để ước lượng khoảng cách từ điểm hiện tại đến đích.

Bước 6: Kiểm tra trạng thái đích

- Mỗi khi một trạng thái mới được thêm vào, kiểm tra xem tất cả các viên đá đã được đặt trên công tắc hay chưa.
- Nếu điều kiện thỏa mãn, thuật toán dừng lại và trả về đường đi.

Bước 7: Ghi kết quả và báo cáo thống kê

- Khi tìm được lời giải, thuật toán sẽ ghi lại các thông số như số bước đã thực hiện, số nút đã mở rộng, tổng thời gian chạy, và trọng lượng cuối cùng.
- Lưu kết quả vào tệp đầu ra.

Cải thiện Heuristic và Tối ưu hiệu suất

- **Hàm Heuristic**: Một hàm heuristic tốt sẽ giúp thuật toán tìm được đường đi nhanh hơn. Ở đây, sử dụng khoảng cách Manhattan làm hàm Heuristic h(n) để ước tính chi phí từ 1 trạng thái đến trạng thái đích (đến công tắc)
- **Input đầu vào**:
  - `posAres`: Tọa độ hiện tại của Ares.
  - `posStone`: Danh sách tọa độ của các viên đá.
  - `posSwitch`: Danh sách tọa độ của các công tắc.
  - `weights`: Danh sách trọng lượng tương ứng của các viên đá.
- **Tính khoảng cách từ Ares đến các viên đá**:
  - Đầu tiên, hàm này tính tổng khoảng cách Manhattan từ vị trí hiện tại của Ares đến tất cả các viên đá.
  - Để tối ưu hơn, các viên đá được sắp xếp theo trọng lượng của chúng
  - Tổng khoảng cách này sẽ được cộng vào chi phí heuristic.
- **Tính khoảng cách từ các viên đá đến công tắc**:

- Với mỗi viên đá, hàm tính khoảng cách ngắn nhất từ viên đá đó đến một trong các công tắc.
- Khoảng cách này được nhân với trọng lượng của viên đá để phản ánh rằng những viên đá nặng hơn có thể tốn nhiều công sức hơn để di chuyển.
- Chi phí cho mỗi viên đá (khoảng cách đến công tắc * trọng lượng) được cộng vào tổng chi phí.

2.4.3. Đánh giá

Nếu heuristic được chọn là **chấp nhận được** (admissible), A* sẽ luôn tìm ra đường đi ngắn nhất.

A* thường nhanh hơn nhiều so với các thuật toán tìm kiếm mù (uninformed search) vì heuristic hướng dẫn quá trình tìm kiếm.

- **Ưu điểm:**
  Tìm kiếm hiệu quả với chi phí thấp nhất (nếu heuristic tốt).
  Áp dụng tốt cho các bài toán tìm đường, lập kế hoạch di chuyển trong không gian hai/ba chiều.
- **Nhược điểm**:
  Bộ nhớ yêu cầu cao, đặc biệt với không gian trạng thái lớn vì phải lưu trữ tất cả các trạng thái đã duyệt.

III. Thử nghiệm và đánh giá thuật toán

1. Thử nghiệm các test case

❖ Test case 01

- **Mô tả:** Mê cung này khá đơn giản (gồm 1 viên đá, 1 công tắc và Ares được bao quanh bởi khung tường hình chữ nhật), chủ yếu kiểm tra khả năng xác định vị trí đá và công tắc, đồng thời tìm ra được đường đi của thuật toán.
  - **Ares Position**: (1, 2) — Ares nằm ở hàng 1, cột 2.
  - **Rock Position**: (2, 3) — Viên đá nằm ở hàng 2, cột 3.
  - **Switch Position**: (4, 3) — Công tắc nằm ở hàng 4, cột 3
- **Mê cung:**

```
2
##########
# @      #
#  $     #
#        #
#  .     #
##########
```

Bảng 1. Mê cung test case 01

- **Kết quả:**

```
BFS
Steps: 3, Weight: 5, Node: 12, Time (ms): 0.00, Memory (MB): 0.00
DFS
Steps: 23, Weight: 29, Node: 235, Time (ms): 1.09, Memory (MB): 0.01
UCS
Steps: 3, Weight: 5, Node: 67, Time (ms): 0.00, Memory (MB): 0.00
A*
Steps: 3, Weight: 5, Node: 12, Time (ms): 0.00, Memory (MB): 0.00
```

Bảng 2. Kết quả test case 01

❖ **Test case 02**

- **Mô tả:** Số lượng đá tăng lên nhưng không đáng kể, vị trí công tắc cũng khá gần đá. Vậy nên, thuật toán không mất quá nhiều thời gian để tìm đường đi. Nhưng thuật toán cần phải xác định được nên đẩy viên đá nào vào công tắc nào để tối ưu nhất.
  - **Ares Position**: (1, 2) — Ares nằm ở hàng 1, cột 2.
  - **Rock Positions**: (2, 2), (2, 3) — Có hai viên đá, một ở hàng 2, cột 2 và một ở hàng 2, cột 3.
  - **Switch Position**: (2,4) và (3,3) — Có hai công tắc, một ở hàng 2, cột 4 và một ở hàng 3, cột 3.
- **Mê cung:**

```
3 6
######
# @  #
# $$.#
#  . #
#    #
######
```

Bảng 3. Mê cung test case 02

- **Kết quả:**

```
BFS
Steps: 6, Weight: 15, Node: 126, Time (ms): 1.00, Memory (MB): 0.01
DFS
Steps: 8, Weight: 17, Node: 938, Time (ms): 4.00, Memory (MB): 0.03
UCS
Steps: 6, Weight: 15, Node: 227, Time (ms): 1.00, Memory (MB): 0.01
A*
Steps: 8, Weight: 17, Node: 84, Time (ms): 0.33, Memory (MB): 0.01
```

Bảng 4. Kết quả test case 02

❖ **Test case 03**

- **Mô tả:** Tương tự như test case 2 nhưng vị trí công tắc đã xa hơn, mê cung cũng phức tạp hơn. Điều này khiến cho mỗi thuật toán lại tìm ra một đường đi khác nhau.
  - **Ares Position**: (3, 3) — Ares nằm ở hàng 3, cột 3.
  - **Rock Positions**: (2, 5), (2, 7) — Có hai viên đá, một ở hàng 2, cột 5 và một ở hàng 2, cột 7.
  - **Switch Positions**: (4, 3) và (4, 9) — Có hai công tắc, một ở hàng 4, cột 3 và một ở hàng 4, cột 9.
- **Mê cung:**

```
2 4
############
#          #
#  # $#$   #
#  @  #    #
#  .      .#
############
```

Bảng 5. Mê cung test case 03

- **Kết quả:**

```
BFS
Steps: 24, Weight: 48, Node: 3866, Time (ms): 12.25, Memory (MB): 0.13
DFS
Steps: 112, Weight: 138, Node: 3753, Time (ms): 7.47, Memory (MB): 0.13
UCS
Steps: 24, Weight: 48, Node: 7865, Time (ms): 19.55, Memory (MB): 0.50
A*
Steps: 24, Weight: 48, Node: 1476, Time (ms): 6.35, Memory (MB): 0.13
```

Bảng 6. Kết quả test case 03

❖ **Test case 04**

- **Mô tả:** Mê cung với địa hình phức tạp hơn, không chỉ xung quanh mà chính giữa mê cung cũng rải rác tường không thể đi qua.
  - **Ares Position**: (1, 2) — Ares nằm ở hàng 1, cột 2.
  - **Rock Positions**: (1, 4), (3, 2), (3, 7) — Có ba viên đá, một ở hàng 1, cột 4, một ở hàng 3, cột 2 và một ở hàng 3, cột 7.
  - **Switch Positions**: (1, 7), (3, 6) và (4, 2) — Có ba công tắc, một ở hàng 1, cột 7, một ở hàng 3, cột 6 và một ở hàng 4, cột 2.
- **Mê cung:**

```
7 11 2
 ##########
##@ $  .###
#    #######
# $   .$  ##
##.       ##
 ##########
```

Bảng 7. Mê cung test case 04

- **Kết quả:**

```
BFS
Steps: 19, Weight: 48, Node: 2961, Time (ms): 7.02, Memory (MB): 0.13
DFS
Steps: 34, Weight: 87, Node: 1969, Time (ms): 5.00, Memory (MB): 0.13
UCS
Steps: 19, Weight: 48, Node: 6042, Time (ms): 17.15, Memory (MB): 0.51
A*
Steps: 22, Weight: 51, Node: 288, Time (ms): 1.01, Memory (MB): 0.01
```

Bảng 8. Kết quả test case 04

❖ **Test case 05**

- **Mô tả:** Mê cung có hình dáng phức tạp hơn và nhiều viên đá với khối lượng khác nhau đòi hỏi thuật toán phải xác định được điều này để di chuyển đá mà tốn ít sức lực. Có các viên đá nằm cạnh nhau nên các thuật toán phải có khả năng xác định rằng không thể đẩy 2 viên đá cùng lúc.
  - **Ares Position**: (2, 1) — Ares nằm ở hàng 2, cột 1.
  - **Rock Positions**:
    - (2, 5) — Viên đá 1.
    - (2, 6) — Viên đá 2.
    - (3, 2) — Viên đá 3.
    - (5, 4) — Viên đá 4.
  - **Switch Positions**:
    - (3, 6) — Công tắc 1.
    - (4, 4) — Công tắc 2.
    - (5, 2) — Công tắc 3.
    - (6, 6) — Công tắc 4.
- **Mê cung:**

```
3 5 1 9
########
#      #
#@   $$##
# $# #. #
#   .  ##
# . $  ##
#     . ##
##########
```

Bảng 9. Mê cung test case 05

- **Kết quả:**

```
BFS
Steps: 24, Weight: 58, Node: 91878, Time (ms): 203.36, Memory (MB): 4.17
DFS
Steps: 72, Weight: 126, Node: 158368, Time (ms): 375.00, Memory (MB): 8.00
UCS
Steps: 26, Weight: 52, Node: 211007, Time (ms): 887.94, Memory (MB): 8.24
A*
Steps: 27, Weight: 53, Node: 11791, Time (ms): 85.44, Memory (MB): 0.53
```

Bảng 10. Kết quả test case 05

❖ **Test case 06**

- **Mô tả:** Test case này nhằm kiểm tra thuật toán có xác định được vị trí của Ares đang đứng trên công tắc hay không. Tường nằm rải rác ở giữa mê cung nên việc xác định đường đi khó khăn hơn.
  - **Ares Position**: (7, 5) — Ares nằm ở hàng 7, cột 5.
  - **Rock Positions**:
    - (2, 2) — Viên đá 1.
    - (3, 5) — Viên đá 2.
    - (6, 5) — Viên đá 3.
  - **Switch Positions**:
    - (3, 4) — Công tắc 1.
    - (4, 5) — Công tắc 2.
    - (7, 5) — Công tắc 3.
- **Mê cung:**

```
2 10 4
##########
#   #    #
#  $     #
#   . $ ##
# ###.# #
#       ##
#     $  #
#    +   #
##########
```

Bảng 11. Mê cung test case 06

- **Kết quả:**

```
BFS
Steps: 26, Weight: 52, Node: 196937, Time (ms): 427.44, Memory (MB): 8.35
DFS
Steps: 1165, Weight: 2041, Node: 1783851, Time (ms): 6080.66, Memory (MB): 64.01
UCS
Steps: 26, Weight: 52, Node: 145537, Time (ms): 482.75, Memory (MB): 4.13
A*
Steps: 26, Weight: 52, Node: 6999, Time (ms): 43.60, Memory (MB): 0.52
```

Bảng 12. Kết quả test case 06

❖ **Test case 07**

- **Mô tả:** Ở mê cung này, không chỉ Ares mà còn có một viên đá đã nằm sẵn trên công tắc nên thuật toán cần xác định được điều này. Mỗi thuật toán có một tiêu chí khác nhau nên cách xác định hướng giải quyết cũng khác nhau. Vì vậy test case kiểm tra được sự tối ưu của từng thuật toán.
  - **Ares Position**: (6, 3) — Ares nằm ở hàng 6, cột 3.
  - **Rock Positions**:
    - (2, 3) — Viên đá 1.
    - (5, 1) — Viên đá 2.
    - (5, 4) — Viên đá 3.
  - **Switch Positions**:
    - (5, 4) — Công tắc 1.
    - (6, 1) — Công tắc 2.
    - (6, 3) — Công tắc 3.
- **Mê cung:**

```
3 2 1
#######
#     #
#  $  ##
# ###  #
#      #
#$  *  #
#. +   #
########
```

Bảng 13. Mê cung test case 07

- **Kết quả:**

```
BFS
Steps: 21, Weight: 38, Node: 10468, Time (ms): 25.31, Memory (MB): 0.52
DFS
Steps: 333, Weight: 366, Node: 16513, Time (ms): 33.19, Memory (MB): 0.50
UCS
Steps: 25, Weight: 38, Node: 33260, Time (ms): 93.90, Memory (MB): 2.03
A*
Steps: 21, Weight: 38, Node: 5207, Time (ms): 27.68, Memory (MB): 0.51
```

Bảng 14. Kết quả test case 07

❖ **Test case 08**

- **Mô tả:** Những viên đá có khối lượng chênh lệch, một viên đá đã nằm sẵn trên công tắc. Điều này đòi hỏi thuật toán phải xác định được khối lượng của những viên đá để dời đi mà tốn ít công sức nhất. Ngoài ra, thuật toán phải xác định có cần đẩy viên đá đã nằm trên công tắc đến công tắc khác để tìm được đường đi tối ưu nhất hay không.
  - **Ares Position**: (1, 5) — Ares nằm ở hàng 1, cột 5.
  - **Rock Positions**:
    - (2, 6) — Viên đá 1.
    - (3, 6) — Viên đá 2.
    - (5, 4) — Viên đá 3.
  - **Switch Positions**:
    - (3, 3) — Công tắc 1.
    - (3, 6) — Công tắc 2.
    - (5, 2) — Công tắc 3.
- **Mê cung:**

```
9 1 5
 ########
##   @  ##
#     $  #
#  .  *  #
#        #
# . $    #
##      ##
 ########
```

Bảng 15. Mê cung test case 08

- **Kết quả:**

```
BFS
Steps: 16, Weight: 56, Node: 43146, Time (ms): 82.03, Memory (MB): 2.13
DFS
Steps: 3288, Weight: 5272, Node: 1338375, Time (ms): 4287.69, Memory (MB): 64.02
UCS
Steps: 20, Weight: 36, Node: 113812, Time (ms): 380.66, Memory (MB): 4.17
A*
Steps: 22, Weight: 38, Node: 3715, Time (ms): 18.78, Memory (MB): 0.14
```

Bảng 16. Kết quả test case 08

❖ **Test case 09**

- **Mô tả:** Mê cung phức tạp hơn, số lượng viên đá nhiều hơn, khối lượng cũng chênh lệch lớn và một viên đá cũng đã có sẵn trên công tắc. Thuật toán cần tính toán nhiều hơn để xác định đường đi tối ưu nhất.
  - **Ares Position**: (5, 2) — Ares nằm ở hàng 5, cột 2.
  - **Rock Positions**:
    - (2, 3) — Viên đá 1.
    - (3, 2) — Viên đá 2.
    - (4, 7) — Viên đá 3.
    - (5, 3) — Viên đá 4.
  - **Switch Positions**:
    - (1, 7) — Công tắc 1.
    - (3, 2) — Công tắc 2.
    - (4, 3) — Công tắc 3.
    - (5, 8) — Công tắc 4.
- **Mê cung:**

```
11 2 15 3
###########
#       . #
#  $ #  ##
# *  ### #
#  .   $ ##
# @$    . #
############
```

Bảng 17. Mê cung test case 09

- **Kết quả:**

```
BFS
Steps: 30, Weight: 146, Node: 423361, Time (ms): 1147.44, Memory (MB): 16.48
DFS
Steps: 230, Weight: 549, Node: 513834, Time (ms): 1304.88, Memory (MB): 16.00
UCS
Steps: 37, Weight: 130, Node: 703172, Time (ms): 3281.22, Memory (MB): 32.21
A*
Steps: 37, Weight: 130, Node: 184842, Time (ms): 1743.44, Memory (MB): 8.15
```

Bảng 18. Kết quả test case 09

❖ **Test case 10**

- **Mô tả:** Ở mê cung này, đa số các viên đá nằm khá sát tường, test case này kiểm tra khả năng xác định đường đi của Ares của từng thuật toán vì hướng để đẩy viên đá đến một công tắc không nhiều.
    - **Ares Position**: (4, 2) — Ares nằm ở hàng 4, cột 2.
    - **Rock Positions**:
        - (2, 4) — Viên đá 1.
        - (2, 7) — Viên đá 2.
        - (4, 7) — Viên đá 3.
        - (6, 2) — Viên đá 4.
    - **Switch Positions**:
        - (3, 3) — Công tắc 1.
        - (4, 7) — Công tắc 2.
        - (5, 6) — Công tắc 3.
        - (6, 4) — Công tắc 4.
- **Mê cung:**

```
4 5 3 2
 ########
##      ##
#   $  $ #
#  . ##  #
# @    * #
#     .  #
# $ .    #
##########
```

Bảng 19. Mê cung test case 10

- **Kết quả:**

```
BFS
Steps: 39, Weight: 59, Node: 2570118, Time (ms): 10535.32, Memory (MB): 130.86
DFS
Steps: 30334, Weight: 44941, Node: 8746457, Time (ms): 111871.82, Memory (MB): 256.14
UCS
Steps: 39, Weight: 59, Node: 1095382, Time (ms): 15214.07, Memory (MB): 32.86
A*
Steps: 40, Weight: 60, Node: 61619, Time (ms): 1302.72, Memory (MB): 2.15
```

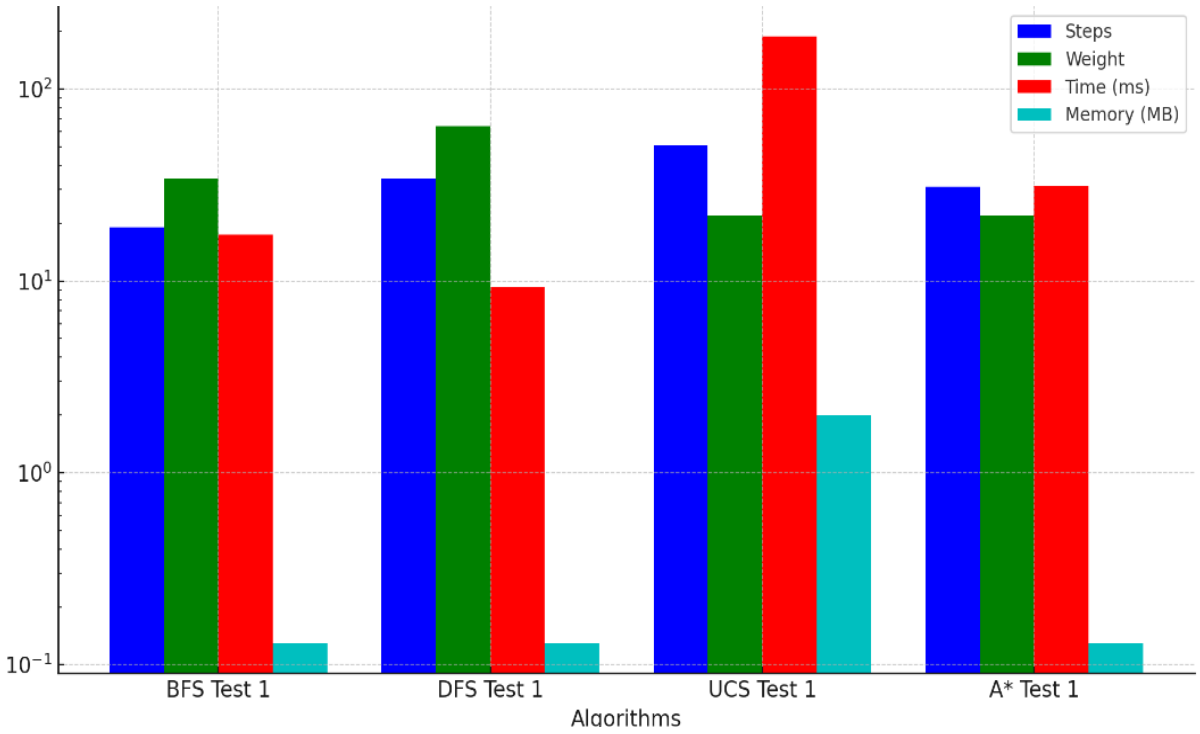
Bảng 20. Kết quả test case 10

Ý nghĩa chung của các test case:

1. **Khả năng di chuyển**: Kiểm tra xem Ares có thể di chuyển vào các ô trống hay không. Ares có thể di chuyển lên, xuống, trái hoặc phải nhưng không thể đi qua tường hay không.
2. **Đẩy viên đá**: Xác định xem Ares có thể đẩy các viên đá vào các vị trí thích hợp hay không. Ares có đẩy hai viên đá cùng một lúc hay không.
3. **Công tắc**: Kiểm tra xem viên đá có thể được đẩy vào vị trí công tắc phù hợp hay không.
4. **Quản lý không gian**: Đảm bảo rằng Ares không bị kẹt giữa các viên đá và tường khi cố gắng đẩy viên đá.
5. **Giải quyết mê cung**: Đảm bảo rằng Ares có thể tìm ra cách hoàn thành nhiệm vụ, đẩy tất cả các viên đá vào các công tắc mà không bị kẹt.

Link google drive video demo: AI - Google Drive

2. Đánh giá và so sánh các thuật toán:



Hình 1. Biểu đồ so sánh các thuật toán theo từng tiêu chí (test case 04)

2.1. BFS (Breadth-First Search)

- **Ưu điểm**:
  - **Đảm bảo tìm được giải pháp tối ưu**: BFS tìm kiếm theo mức độ, nên nó luôn tìm được giải pháp ngắn nhất (ít bước di chuyển nhất) nếu có.
  - **Khả năng khám phá đồng đều**: BFS khám phá tất cả các trạng thái ở một mức độ trước khi chuyển sang mức độ tiếp theo, giúp phát hiện ra các giải pháp gần hơn.
- **Khuyết điểm**:
  - **Tiêu tốn bộ nhớ**: BFS cần phải lưu trữ tất cả các trạng thái đã khám phá, điều này có thể dẫn đến việc tiêu tốn rất nhiều bộ nhớ, đặc biệt trong các mê cung lớn.
  - **Thời gian tìm kiếm cao**: Trong trường hợp mê cung lớn, thời gian tìm kiếm có thể rất lâu do cần phải khám phá tất cả các nhánh.

2.2. DFS (Depth-First Search)

- **Ưu điểm**:
  - **Tiết kiệm bộ nhớ**: DFS chỉ lưu trữ một nhánh duy nhất tại một thời điểm, do đó nó có yêu cầu bộ nhớ thấp hơn so với BFS.
  - **Dễ dàng triển khai**: DFS có thể được triển khai dễ dàng với đệ quy, không cần cấu trúc dữ liệu phức tạp.
- **Khuyết điểm**:
  - **Không đảm bảo giải pháp tối ưu**: DFS có thể tìm ra một giải pháp nhanh chóng nhưng không nhất thiết phải là giải pháp ngắn nhất.
  - **Nguy cơ bị kẹt trong vòng lặp**: Nếu không quản lý tốt các trạng thái đã khám phá, DFS có thể quay trở lại các trạng thái trước đó và không bao giờ tìm thấy giải pháp.

2.3. UCS (Uniform Cost Search)

- **Ưu điểm**:
  - **Tìm giải pháp tối ưu**: UCS đảm bảo tìm được giải pháp tối ưu cho bài toán tìm kiếm, bởi vì nó luôn mở rộng nhánh có chi phí thấp nhất.
  - **Chú trọng vào chi phí**: UCS rất hiệu quả trong các bài toán mà chi phí di chuyển giữa các trạng thái không đồng đều.
- **Khuyết điểm**:
  - **Tiêu tốn bộ nhớ**: Giống như BFS, UCS cũng yêu cầu lưu trữ tất cả các trạng thái đã khám phá, do đó tiêu tốn nhiều bộ nhớ.
  - **Thời gian tìm kiếm cao**: Việc tính toán chi phí cho từng trạng thái có thể làm tăng thời gian tìm kiếm, đặc biệt trong các mê cung lớn.

2.4. A* (A-Star)

- **Ưu điểm**:
  - **Tìm giải pháp tối ưu**: A* đảm bảo tìm được giải pháp tối ưu như UCS nhưng sử dụng hàm heuristic để ưu tiên các nhánh khả thi hơn.
  - **Tiết kiệm thời gian**: Nhờ vào việc sử dụng heuristic, A* có thể giảm thiểu số lượng trạng thái cần khám phá, giúp tiết kiệm thời gian tìm kiếm.
  - **Chọn lựa thông minh**: A* có thể điều chỉnh theo cách mà nó khám phá không gian tìm kiếm, giúp nó hiệu quả hơn trong nhiều bài toán cụ thể.

- **Khuyết điểm**:
    - **Cần thiết lập hàm heuristic tốt**: Hiệu suất của A* phụ thuộc rất nhiều vào chất lượng của hàm heuristic. Nếu heuristic không chính xác, A* có thể chậm hơn so với các thuật toán khác.
    - **Tiêu tốn bộ nhớ**: Mặc dù A* thường tốt hơn so với BFS và UCS, nhưng nó vẫn yêu cầu bộ nhớ cao do lưu trữ trạng thái và chi phí của chúng.

TÀI LIỆU KHAM KHẢO

https://vi.wikipedia.org/wiki/Sokoban

https://wiki.vnoi.info/algo/graph-theory/breadth-first-search.md

https://viblo.asia/p/data-structure-algorithm-graph-algorithms-depth-first-search-dfs-qPoL7zyXJvk

https://labs.flinters.vn/algorithm/algorithm-cac-thuat-toan-tim-kiem-trong-ai/

https://vi.wikipedia.org/wiki/Gi%E1%BA%A3i_thu%E1%BA%ADt_t%C3%ACm_ki%E1%BA%BFm_A*