TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP. HỒ CHÍ MINH

KHOA CÔNG NGHỆ THÔNG TIN

TIỂU LUẬN CUỐI KỲ

Môn học : TRÍ TUỆ NHÂN TẠO

Tên tiểu luận:

XÂY DỰNG CHƯƠNG TRÌNH TÌM KIẾM LỜI GIẢI CHO TRÒ CHƠI SOKOBAN SỬ DỤNG MỘT SỐ THUẬT TOÁN TRONG AI

Nhóm lớp: ARIN330585_23_1_02CLC

Giảng viên hướng dẫn: PGS.TS. Hoàng Văn Dũng

Danh sách sinh viên thực hiện

Mã số SV	Họ và tên	Mức độ đóng góp (%)
21110254	Nguyễn Hoàng Phương Ngân	100%
20161347	Nguyễn Bùi Minh Nhật	100%
21110858	Nguyễn Minh Trí	100%

TP. Hồ Chí Minh, tháng 12 năm 2023

Nhận xét của giáo viên

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Ngày ... tháng 12 năm 2023

Giáo viên chấm điểm

KẾ HOẠCH PHÂN CÔNG NHIỆM VỤ THỰC HIỆN ĐỀ TÀI CUỐI KỲ MÔN TRÍ TUỆ NHÂN TẠO HỌC KỲ I NĂM HỌC 2023-2024

1. Mã lớp môn học: ARIN330585_23_1_02CLC

2. Giảng viên hướng dẫn: PGS.TS. Hoàng Văn Dũng

3. Tên đề tài: Xây dựng chương trình tìm kiếm lời giải cho trò chơi Sokoban sử dụng một số thuật toán trong AI

4. Bảng phân công nhiệm vụ:

Nội dung hoàn thành	**Sinh viên hoàn thành**	**Mức độ hoàn thành**
VIẾT CODE PYTHON CHO ĐỀ TÀI		
1. Thiết kế giao diện game Sokoban trên Python	Nguyễn Hoàng Phương Ngân Nguyễn Bùi Minh Nhật	100 %
2. Viết code cho thuật toán DFS, Depth Limited Search	Nguyễn Bùi Minh Nhật	100%
3. Viết code cho thuật toán BFS	Nguyễn Minh Trí	100%
4. Viết code cho thuật toán A*, UCS, Greedy Best First Search	Nguyễn Hoàng Phương Ngân Nguyễn Minh Trí	100%
5. Tinh chỉnh lại code	Nguyễn Bùi Minh Nhật	100%
VIẾT BÁO CÁO CHO ĐỀ TÀI		
PHẦN 1. MỞ ĐẦU		
1.1. Lời nói đầu	Nguyễn Minh Trí Nguyễn Hoàng Phương Ngân	100%
1.2. Lý do chọn đề tài		
1.3. Phát biểu bài toán		
1.4. Mục tiêu của đề tài		
1.5. Đối tượng nghiên cứu		
1.6. Phạm vi nghiên cứu		

PHẦN 2. CƠ SỞ LÝ THUYẾT		
2.1. Tổng quan về trò chơi Sokoban và đề tài	Nguyễn Minh Trí	100%
2.2. Môi trường lập trình		
2.3. Thư viện hỗ trợ		
2.4. Các thuật toán áp dụng vào game Sokoban	Nguyễn Minh Trí Nguyễn Hoàng Phương Ngân	100%
PHẦN 3. PHÂN TÍCH, THIẾT KẾ GIẢI PHÁP CHO GAME SOKOBAN		
3.1. Sơ đồ khối giải pháp thực hiện bài toán	Nguyễn Hoàng Phương Ngân	100%
3.2. Chi tiết về các thuật toán chính	Nguyễn Hoàng Phương Ngân Nguyễn Minh Trí Nguyễn Bùi Minh Nhật	100%
PHẦN 4. THỰC NGHIỆM, ĐÁNH GIÁ, VÀ PHÂN TÍCH KẾT QUẢ		
4.1. Thiết kế tổng quan giao diện	Nguyễn Hoàng Phương Ngân	100%
4.2. Xây dựng giao diện trên Python	Nguyễn Hoàng Phương Ngân Nguyễn Minh Trí	100%
4.3. Hướng dẫn thực thi phần mềm	Nguyễn Hoàng Phương Ngân	100%
4.4. Trình bày các kết quả thử nghiệm	Nguyễn Hoàng Phương Ngân Nguyễn Bùi Minh Nhật	100%
PHẦN 5. KẾT LUẬN		
5.1 Đánh giá kết quả thực hiện	Nguyễn Minh Trí	100%
5.2 Định hướng phát triển		

DANH MỤC CÁC CỤM TỪ VIẾT TẮT

BFS	Breadth First Search
DFS	Depth First Seach
A*	A Star
UCS	Uniform cost search
DLS	Depth Limited Search
GBFS	Greedy Best-First Search

DANH MỤC HÌNH ẢNH

MỤC LỤC

PHẦN 1. MỞ ĐẦU

1.1. Lời nói đầu

Trong thời đại ngày nay, sự bùng nổ của công nghệ thông tin và trí tuệ nhân tạo đã mở ra những cánh cửa mới đầy tiềm năng trong nhiều lĩnh vực khác nhau. Trí tuệ nhân tạo đang đóng một vai trò quan trọng trong việc giải quyết các thách thức trong cuộc sống hàng ngày. Việc này không chỉ giới hạn trong lĩnh vực công nghiệp và tự động hóa, mà còn mở ra những khả năng mới trong việc giải trí điển hình là áp dụng trí tuệ nhân tạo vào các trò chơi. Trò chơi Sokoban, một trong những trò chơi logic kinh điển, được chúng em chọn làm đối tượng nghiên cứu để minh họa cách trí tuệ nhân tạo có thể làm thay đổi cách chúng ta tận hưởng thế giới giải trí.

Trong bài báo cáo này, chúng em sẽ tập trung đề cập đến việc ứng dụng trí tuệ nhân tạo vào trò chơi Sokoban, nơi mà sự kết hợp giữa thuật toán và giao diện trực quan giúp tạo ra trải nghiệm độc đáo và hấp dẫn cho người chơi.

Chúng em sẽ trình bày cụ thể về cách chúng em kết hợp thuật toán và giao diện trực quan để tạo ra một trải nghiệm Sokoban độc đáo, không chỉ thách thức người chơi mà còn thể hiện sức mạnh của trí tuệ nhân tạo trong việc giải quyết vấn đề và tối ưu hóa quy trình giải quyết bài toán đã đề ra.

Vì thời gian có hạn và trình độ kiến thức của em chúng em chưa được hoàn hảo nên việc trò chơi xuất hiện những thiếu sót là khó tránh khỏi. Vì vậy chúng em mong nhận được sự đóng góp của thầy cô và bạn bè để đề tài của nhóm em có thể được hoàn thiện hơn.

1.2. Lý do chọn đề tài

Trò chơi Sokoban được chọn làm đối tượng nghiên cứu vì nó là một trò chơi logic kinh điển, đòi hỏi người chơi phải sử dụng tư duy chiến lược và kỹ năng giải quyết vấn đề để hoàn thành mỗi cấp độ. Hơn nữa, Sokoban cung cấp một môi trường lý tưởng để áp dụng các thuật toán trí tuệ nhân tạo, vì việc tìm kiếm đường đi tối ưu trong một không gian giới hạn là một thách thức lớn mà trí tuệ nhân tạo có thể giải quyết một cách hiệu quả.

1.3. Phát biểu bài toán

Bài toán đặc ra là xây dựng hệ thống trí tuệ nhân tạo để phân tích và đánh giá kết quả thời gian, số bước, số trạng thái duyệt và bộ nhớ trong việc nhân vật di chuyển trong map để đẩy hộp về đích bằng các giải thuật tìm kiếm đường đi trong AI mà chúng em đã được học.

1.4. Mục tiêu của đề tài

Mục tiêu chính của đề tài này là khám phá cách trí tuệ nhân tạo có thể được áp dụng vào trò chơi Sokoban để tạo ra một trải nghiệm chơi game mới mẻ và thú vị. Chúng em mong muốn không chỉ tạo ra một trò chơi giải trí, mà còn tạo ra một công cụ học tập giúp người chơi hiểu rõ hơn về cách trí tuệ nhân tạo hoạt động. Ngoài ra, chúng em cũng hy vọng rằng công việc này sẽ mở ra cánh cửa cho những nghiên cứu và ứng dụng trí tuệ nhân tạo tiếp theo trong lĩnh vực giải trí.

1.5. Đối tượng nghiên cứu

Đối tượng nghiên cứu của chúng em trong đề tài này là trò chơi Sokoban và các thuật toán trí tuệ nhân tạo có thể được áp dụng để giải quyết các bài toán trong trò chơi. Chúng em sẽ tập trung vào việc phân tích cách mà các thuật toán trí tuệ nhân tạo có thể giúp người chơi giải quyết những thách thức theo mong muốn.

1.6. Phạm vi nghiên cứu

Phạm vi nghiên cứu của chúng em bao gồm việc phân tích và hiểu rõ cấu trúc của trò chơi Sokoban, cũng như việc nghiên cứu và áp dụng các thuật toán trí tuệ nhân tạo phù hợp để giải quyết các bài toán trong trò chơi. Chúng em sẽ không đi sâu vào việc phát triển phần mềm hoặc giao diện người dùng, mà tập trung vào việc tìm hiểu cách trí tuệ nhân tạo có thể được áp dụng để cải thiện trải nghiệm chơi game. Chúng em cũng sẽ không đi sâu vào việc nghiên cứu các thuật toán trí tuệ nhân tạo nâng cao mà chỉ tập trung vào những thuật toán cơ bản đã được học trong khóa học.

PHẦN 2. CƠ SỞ LÝ THUYẾT

2.1. Tổng quan về trò chơi Sokoban và đề tài

2.1.1. Giới thiệu trò chơi

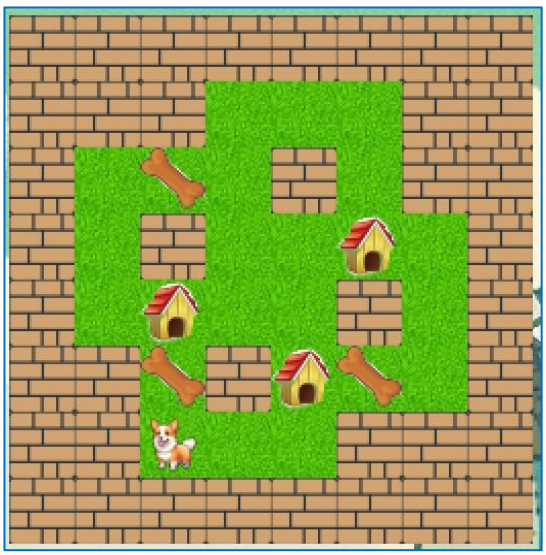
Trò chơi Sokoban là một trò chơi logic nổi tiếng xuất phát từ Nhật Bản. Nó được tạo ra bởi Hiroyuki Imabayashi và được phát hành lần đầu tiên vào năm 1982 tại Nhật Bản. Trò chơi có dạng bảng ô vuông. Có một số khối vuông được đẩy đến đích (số ô đích đúng bằng số khối vuông). Chỉ có thể đẩy từng khối vuông một, và không thể kéo, cũng như không thể đẩy một dãy hai hay nhiều khối.

Mục tiêu: Trong Sokoban, người chơi có mục tiêu là điều khiển một nhân vật để đẩy các hộp về các vị trí đích trên bản đồ.

Thành phần chính trong trò chơi:

- Nhân vật (Player): Người chơi điều khiển nhân vật di chuyển để đưa các hộp về vị trí đích.
- Khối vuông (Box): Đây là đối tượng mà bạn cần đẩy vào các ô đích.
- Ô đích (Goal): Ô đích là nơi bạn cần đẩy các hộp tới đề hoàn thành trò chơi.

Trong bài làm của nhóm trò chơi Sokoban đã có một giao diện hoàn toàn mới. Dựa theo cảm hứng từ cuộc sống hằng ngày của một chú chó corgi, chúng em đã thay thế nhân vật thành một chú corgi dễ thương đang tìm cách để đưa thức ăn (thay thế cho chiếc hộp/thùng) về ngôi nhà của mình (thay thế cho ô đích).



Hình 1: Một số hình ảnh về trò chơi

2.1.2. Áp dụng trí tuệ nhân tạo trong trò chơi Sokoban

Trí tuệ nhân tạo đã đánh dấu sự tiến bộ lớn trong việc tạo ra các trình điều khiển tự động, việc áp dụng các trình điều khiển tự động này vào trò chơi là không ngoại lệ. Ở trò chơi Sokoban thay vì chỉ dựa vào người chơi là con người, chúng ta có thể sử dụng các thuật toán để trò chơi có thể tự động tìm lời giải.

Sự kết hợp giữa lập trình và trí tuệ nhân tạo đã cho phép chúng ta tạo ra các "người máy" có khả năng tư duy logic và giải quyết những vấn đề trong trò chơi Sokoban một cách hiệu quả.

Những ứng dụng của trí tuệ nhân tạo trong trò chơi Sokoban:

- Tạo trình điều khiển tự động để giúp người chơi giải quyết những level mà người chơi gặp khó khăn.
- Áp dụng những thuật toán tối ưu để có thể tìm ra cách di chuyển những khối vuông về dích trong thời gian ngắn nhất.
- Hiểu rõ hơn cách máy tính áp dụng kiến thức trong quá trình học máy để cải thiện hiệu suất chơi trò chơi Sokoban.

2.2. Môi trường lập trình

Với đồ án cho môn học này chúng em sử dụng Python làm ngôn chính để viết chương trình cùng với biên dịch là Visual Studio Code.

2.3. Thư viện hỗ trợ

- NumPy: Thư viện NumPy được sử dụng cho các phép toán số học và thao tác trên mảng đa chiều.
- Colorama: Colorama là một thư viện Python giúp làm cho việc in màu trên terminal trở nên dễ dàng hơn. Fore được sử dụng để thay đổi màu chữ, và Style được sử dụng để điều chỉnh kiểu chữ.
- Copy: Import deepcopy từ thư viện copy. deepcopy được sử dụng để tạo ra một bản sao sâu (copy sâu) của đối tượng, giúp tránh vấn đề liên quan đến tham chiếu đối tượng.
- Pygame: Pygame là một thư viện cho việc phát triển trò chơi và ứng dụng đa phương tiện sử dụng ngôn ngữ lập trình Python.

2.4. Các thuật toán áp dụng vào game Sokoban

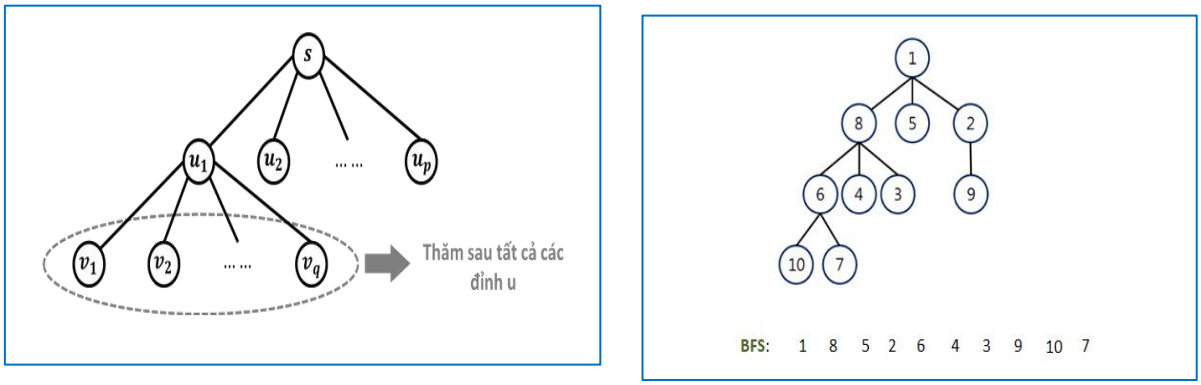
2.4.1. Giải thuật BFS (Breadth First Search)

Thuật toán BFS là thuật toán tìm kiếm không có trọng số được sử dụng để duyệt và tìm kiếm trạng thái hoặc nút trong một đồ thị hoặc cây. BFS bắt đầu từ nút gốc (nút ban đầu) và duyệt toàn bộ đồ thị theo chiều rộng, tức là duyệt qua tất cả các nút cùng cấp trước khi di chuyển xuống cấp tiếp theo. Thuật toán này thường được sử dụng để tìm đường đi ngắn nhất giữa hai nút trong đồ thị hoặc cây không có trọng số. Dưới đây là mô tả cụ thể của thuật toán BFS:

1. Khởi tạo: Bắt đầu khởi tạo từ nút gốc (nút ban đầu) và đặt nó vào hàng đợi.
2. Duyệt và mở rộng: Lặp qua các bước sau đây cho đến khi hàng đợi trống:
   - Lấy nút đầu tiên ra khỏi hàng đợi.
   - Kiểm tra xem nút này có phải là mục tiêu (nút kết thúc) hay không. Nếu có, quá trình kết thúc và bạn đã tìm thấy lời giải (đường đi) nếu mục tiêu là mục đích của tìm kiếm.
   - Nếu nút không phải là mục tiêu, mở rộng nút này bằng cách thêm tất cả các nút con của nó (các nút kề) vào hàng đợi.
3. Lặp lại: Quay lại bước 2 để tiếp tục duyệt và mở rộng các nút khác trong hàng đợi.
4. Kết thúc: Khi hàng đợi trống, nếu không tìm thấy mục tiêu, thuật toán đã hoàn thành và thông báo rằng không có đường đi nào tới mục tiêu.

BFS đảm bảo tìm đường đi ngắn nhất trong đồ thị vô hướng không có trọng số. Nó cũng thường được sử dụng trong các bài toán tìm kiếm đường đi, kiểm tra kết nối giữa các đối tượng, và phân tích cấu trúc đồ thị.

Một lưu ý quan trọng là BFS tiêu tốn nhiều bộ nhớ khi tìm kiếm trong các đồ thị lớn với số lượng lớn nút con.



Hình 2: Mô tả cách thuật toán Breadth First Search hoạt động

2.4.2. Giải thuật DFS (Depth First Search)

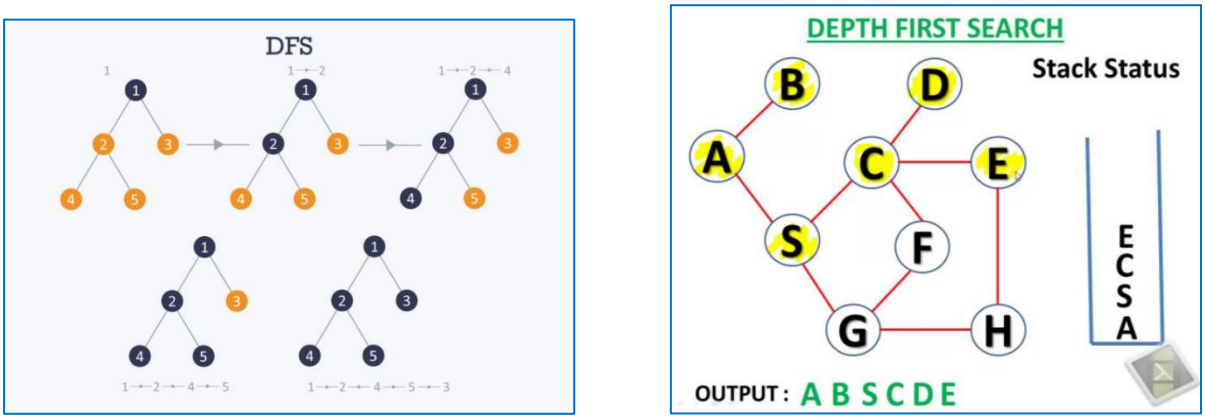
Thuật toán DFS, hay còn gọi là tìm kiếm theo chiều sâu, là một thuật toán tìm kiếm không có trọng số được sử dụng để duyệt và tìm kiếm trạng thái hoặc nút trong một đồ thị hoặc cây. DFS bắt đầu từ một nút gốc (nút ban đầu) và liên tục di chuyển xuống theo một nhánh cụ thể cho đến khi không thể di chuyển nữa, sau đó quay lại và thử các nhánh khác. Thuật toán này thường được sử dụng để kiểm tra kết nối giữa các đối tượng trong đồ thị, tìm kiếm các thành phần liên thông, và giải quyết các bài toán liên quan đến cấu trúc cây hoặc đồ thị. Dưới đây là mô tả cụ thể của thuật toán DFS:

1. Khởi tạo: Bắt đầu từ nút gốc và đặt nó vào ngăn xếp.
2. Duyệt và mở rộng: Lặp qua các bước sau cho đến khi ngăn xếp trống:

- Lấy nút trên đỉnh ngăn xếp.
- Kiểm tra xem nút này có phải là mục tiêu hay không. Nếu có, quá trình kết thúc và bạn đã tìm thấy lời giải nếu mục tiêu là mục đích của tìm kiếm.
- Nếu nút không phải là mục tiêu, mở rộng nút này bằng cách thêm tất cả các nút con của nó vào đỉnh ngăn xếp.

3. Lặp lại: Quay lại bước 2 để tiếp tục duyệt và mở rộng các nút khác trong ngăn xếp.
4. Kết thúc: Khi ngăn xếp trống, nếu không tìm thấy mục tiêu, thuật toán đã hoàn thành và thông báo rằng không có đường đi nào tới mục tiêu.

DFS không đảm bảo tìm kiếm đường đi ngắn nhất, nhưng nó thường được sử dụng trong các tình huống cần kiểm tra cấu trúc đồ thị hoặc cây. Một điểm quan trọng là DFS có thể tiết kiệm bộ nhớ hơn so với BFS khi tìm kiếm trong các đồ thị lớn với số lượng lớn nút con.



Hình 3: Mô tả cách thuật toán Depth First Search hoạt động

2.4.3. Giải thuật Depth Limited Search (DLS)

Thuật toán Depth Limited Search (DLS) là một biến thể của thuật toán tìm kiếm theo chiều sâu (DFS), nhưng với một giới hạn độ sâu cố định.

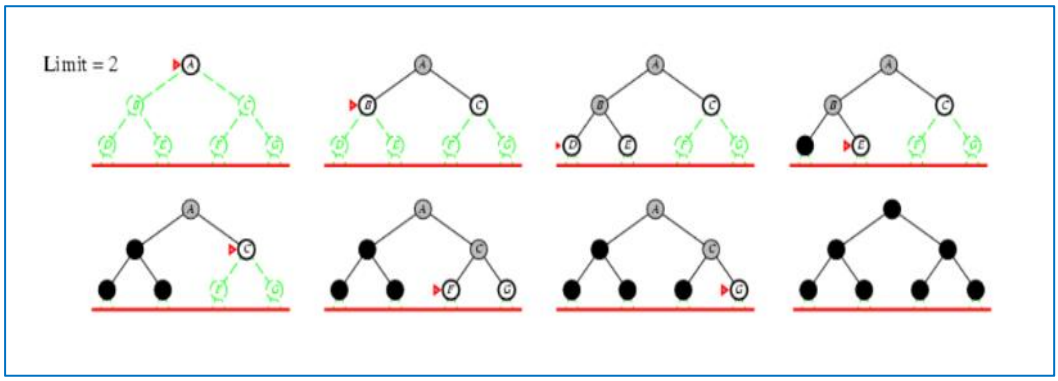
Dưới đây là cách thuật toán hoạt động:

1. Khởi tạo: Thuật toán bắt đầu bằng cách khởi tạo một ngăn xếp (stack) và một tập hợp đã duyệt (visited set). Trạng thái ban đầu được đẩy vào ngăn xếp.
2. Duyệt: Thuật toán tiếp tục duyệt qua các trạng thái bằng cách lấy ra (pop) trạng thái ở đỉnh ngăn xếp. Nếu trạng thái này chưa được duyệt và chưa vượt quá độ sâu tối đa, thuật toán sẽ tạo ra các trạng thái con của nó và đẩy chúng vào ngăn xếp.
3. Kiểm tra: Mỗi khi một trạng thái mới được tạo ra, thuật toán sẽ kiểm tra xem nó có phải là trạng thái mục tiêu (goal state) hay không. Nếu đúng, thuật toán sẽ dừng lại và trả về đường dẫn từ trạng thái ban đầu đến trạng thái mục tiêu.

4. Giới hạn độ sâu: Nếu độ sâu của một trạng thái con vượt quá độ sâu tối đa, trạng thái đó sẽ không được thêm vào ngăn xếp. Điều này ngăn thuật toán đi quá sâu vào không gian trạng thái, giúp giảm thiểu thời gian tính toán và sử dụng tài nguyên.

5. Lặp lại: Quá trình duyệt và kiểm tra này sẽ lặp lại cho đến khi tìm thấy trạng thái mục tiêu hoặc ngăn xếp trống.

Lưu ý rằng DLS không nhất thiết phải tìm thấy giải pháp tối ưu nhất, nhưng nó có thể tìm thấy giải pháp nhanh hơn so với DFS thông thường nếu độ sâu tối đa được chọn một cách hợp lý. Nếu không tìm thấy giải pháp trong độ sâu tối đa, bạn có thể tăng độ sâu và thử lại.



Hình 4: Mô tả cách thuật toán Depth Limited Search hoạt động(giới hạn là 2)

2.4.4. Giải thuật Uniform Cost Search (UCS)

Thuật toán Uniform Cost Search (UCS) là một thuật toán tìm kiếm không gian trạng thái dựa trên chi phí. Dưới đây là cách UCS hoạt động:

1. Khởi tạo: Bắt đầu bằng cách đặt nút gốc vào hàng đợi ưu tiên. Hàng đợi này được sắp xếp dựa trên chi phí đường đi từ nút gốc đến mỗi nút, với nút có chi phí thấp nhất được xem xét đầu tiên.

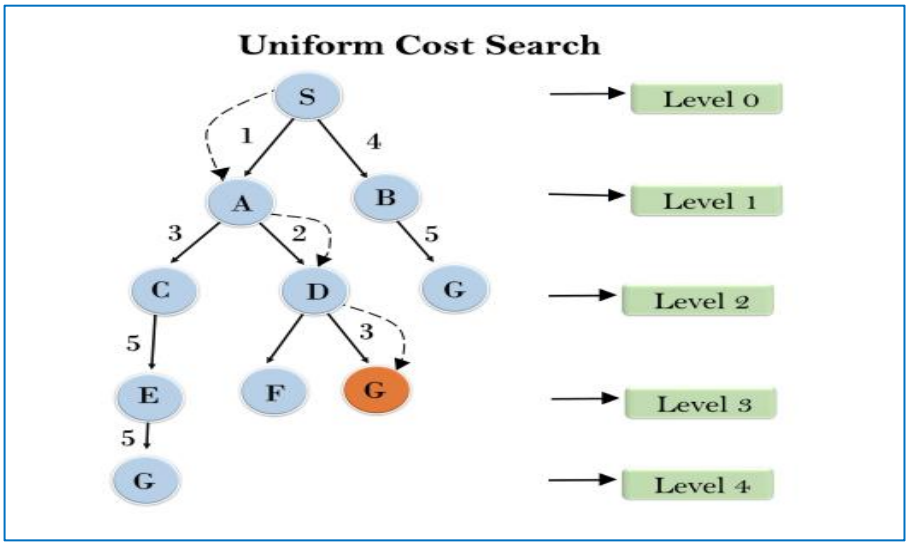
2. Lặp lại cho đến khi tìm thấy giải pháp hoặc không còn nút nào để kiểm tra: Nếu hàng đợi trống, thuật toán sẽ kết thúc và thông báo rằng không tìm thấy giải pháp. Nếu nút đầu tiên trong hàng đợi là nút mục tiêu, thuật toán sẽ kết thúc và trả về nút đó như là giải pháp.

3. Mở rộng nút: Nếu không tìm thấy giải pháp, thuật toán sẽ loại bỏ nút đầu tiên khỏi hàng đợi và thêm tất cả các nút con của nó vào hàng đợi. Các nút con này được sắp xếp trong hàng đợi dựa trên chi phí đường đi từ nút gốc đến chúng.

4. Quay lại bước 2: Thuật toán tiếp tục lặp lại quá trình này cho đến khi tìm thấy giải pháp hoặc không còn nút nào để kiểm tra.

Điều quan trọng là chi phí đường đi từ nút gốc đến mỗi nút phải được tính toán một cách chính xác. Nếu không, UCS có thể không tìm thấy đường đi tối ưu. Ngoài ra, nếu hai nút có cùng chi phí, UCS sẽ chọn nút được thêm vào hàng đợi đầu tiên.

Điều này có nghĩa là thứ tự mà các nút được thêm vào hàng đợi có thể ảnh hưởng đến kết quả của thuật toán.



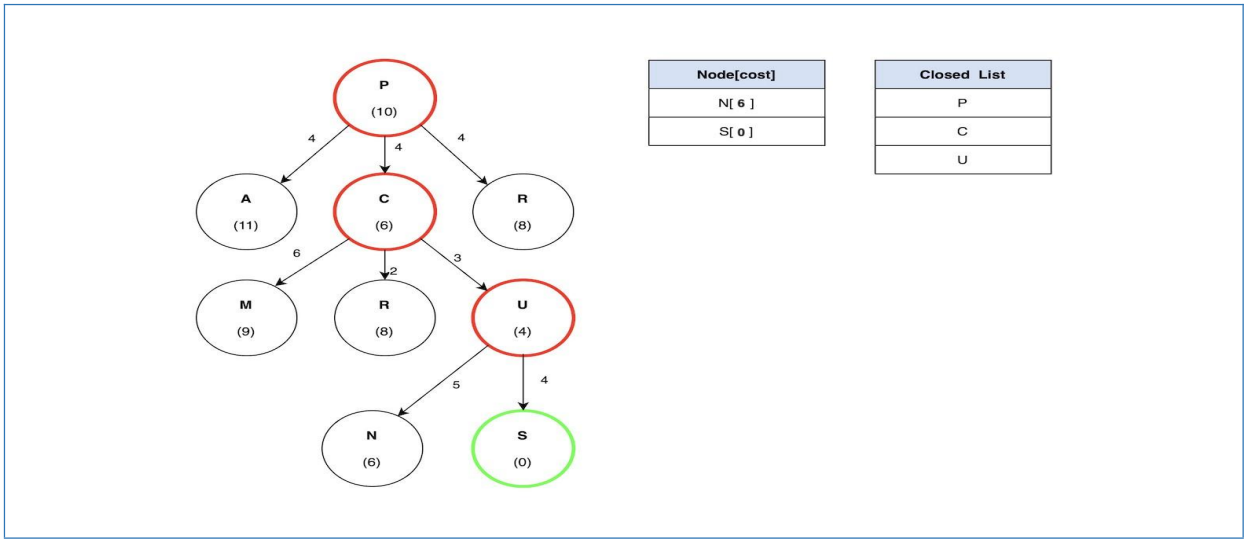
Hình 5: Mô tả cách thuật toán Uniform Cost Search hoạt động

2.4.5. Giải thuật Greedy Best-First Search

Thuật toán Greedy Best-First Search là một thuật toán tìm kiếm thông minh sử dụng hàm đánh giá. Hàm đánh giá này đưa ra ước lượng về chất lượng của mỗi nút từ nút hiện tại đến nút mục tiêu. Dưới đây là mô tả cách giải thuật này hoạt động:

1. Khởi tạo: Bắt đầu bằng cách đặt nút gốc vào hàng đợi ưu tiên. Hàng đợi này được sắp xếp dựa trên giá trị của hàm đánh giá, với nút có giá trị thấp nhất được xem xét đầu tiên.
2. Lặp lại cho đến khi tìm thấy giải pháp hoặc không còn nút nào để kiểm tra: Nếu hàng đợi trống, thuật toán sẽ kết thúc và thông báo rằng không tìm thấy giải pháp. Nếu nút đầu tiên trong hàng đợi là nút mục tiêu, thuật toán sẽ kết thúc và trả về nút đó như là giải pháp.
3. Mở rộng nút: Nếu không tìm thấy giải pháp, thuật toán sẽ loại bỏ nút đầu tiên khỏi hàng đợi và thêm tất cả các nút con của nó vào hàng đợi. Các nút con này được sắp xếp trong hàng đợi dựa trên giá trị của hàm đánh giá.
4. Quay lại bước 2: Thuật toán tiếp tục lặp lại quá trình này cho đến khi tìm thấy giải pháp hoặc không còn nút nào để kiểm tra.

Điều quan trọng là hàm đánh giá được sử dụng trong Best-First Search phải được chọn cẩn thận vì nó có ảnh hưởng đến kết quả của thật toán. Ngoài ra, nếu hai nút có cùng giá trị hàm đánh giá, Best-First Search sẽ chọn nút được thêm vào hàng đợi đầu tiên. Điều này có nghĩa là thứ tự mà các nút được thêm vào hàng đợi cũng có thể ảnh hưởng đến kết quả của thuật toán.



Hình 6: Mô tả cách thuật toán Best First Search hoạt động (Hình b).
A là nút gốc, tìm đường đi tốt nhất đến B

Greedy best-first search có tối ưu không?

Từ những kiến thức về Greedy best first search, ta có thể rút ra được những nhận xét:

- GBFS không phải lúc nào cũng tối ưu. Điều này là do GBFS chỉ tập trung vào việc tìm đường đi có tiềm năng nhất, không phải là đường đi ngắn nhất. Do đó, GBFS không đảm bảo rằng đường đi tìm được sẽ là đường đi ngắn nhất từ điểm bắt đầu đến mục tiêu.
- Tuy nhiên, GBFS có thể tối ưu trong một số trường hợp. Nếu hàm heuristic được sử dụng trong GBFS được ước lượng tốt, GBFS có thể tìm ra giải pháp nhanh chóng, ngay cả trong không gian tìm kiếm lớn.
- Thế nhưng, GBFS cũng có nhược điểm. GBFS có thể bị mắc kẹt trong điểm tối ưu cục bộ, nghĩa là đường đi được chọn có thể không phải là đường đi tốt nhất có thể.
- GBFS cũng đòi hỏi một hàm heuristic để hoạt động, điều này thêm phức tạp vào thuật toán. Nếu hàm heuristic không hiệu quả nếu không gian tìm kiếm quá phức tạp thì thuật toán có thể bị mắc kẹt trong vòng lặp và không tìm ra được lời giải cho bài toán.

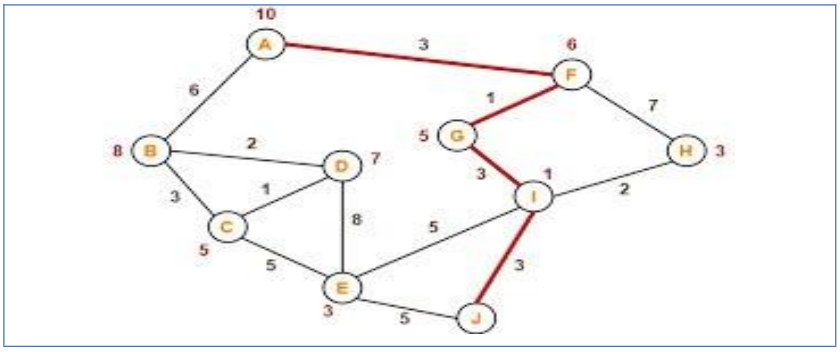
2.4.6. Giải thuật A Star

Thuật toán A* là một thuật toán tìm kiếm thông minh (intelligent search) hay tìm kiếm có thông tin (informed search algorithm) được sử dụng để tìm đường đi tối ưu từ một trạng thái ban đầu đến trạng thái kết thúc trong một không gian trạng thái có cấu

trúc. Thuật toán này thường được áp dụng trong các bài toán tìm kiếm đường đi trong trò chơi, lập trình robot tự lái, quá trình quyết định trong trí tuệ nhân tạo, và nhiều lĩnh vực khác. A* sử dụng một hàm heuristic để ước tính chi phí tối ưu từ trạng thái hiện tại đến trạng thái kết thúc, và nó duyệt qua các trạng thái tiềm năng theo thứ tự ước tính chi phí thấp nhất đầu tiên. Dưới đây là mô tả cụ thể về cách thuật toán A* hoạt động:

1. Khởi tạo: Bắt đầu từ trạng thái ban đầu, tạo một danh sách có trạng thái ban đầu và gán chi phí gốc (g) cho nó là 0.
2. Danh Sách Mở: Tạo một danh sách mở để lưu trữ các trạng thái chưa được duyệt. Đưa trạng thái ban đầu vào danh sách này.
3. Danh Sách Đóng: Tạo một danh sách đóng để lưu trữ các trạng thái đã được duyệt.
4. Lặp cho đến khi danh sách mở trống:
    - Chọn trạng thái (nút) có ước tính chi phí thấp nhất từ danh sách mở.
    - Di chuyển trạng thái này từ danh sách mở sang danh sách đóng.
    - Kiểm tra nếu trạng thái này là trạng thái kết thúc. Nếu có, dừng thuật toán và trả về đường đi tối ưu.
    - Duyệt qua các trạng thái con (nếu có) của trạng thái hiện tại:
        - Tính toán chi phí g từ trạng thái ban đầu đến trạng thái con thông qua trạng thái hiện tại.
        - Tính toán chi phí ước tính h từ trạng thái con đến trạng thái kết thúc bằng cách sử dụng hàm heuristic.
        - Tính toán chi phí tổng f = g + h.
        - Nếu trạng thái con không nằm trong danh sách mở hoặc chi phí f từ trạng thái ban đầu đến trạng thái con thấp hơn chi phí đã biết, cập nhật chi phí và thêm trạng thái con vào danh sách mở.
5. Nếu danh sách mở trống mà không tìm thấy đường đi đến trạng thái kết thúc, thì không có đường đi tới trạng thái đó.

Thuật toán A* thường được sử dụng trong trò chơi Sokoban và nhiều ứng dụng khác để tìm đường đi tối ưu và đảm bảo hiệu suất tối ưu. Hàm heuristic trong thuật toán A* là một yếu tố quan trọng, và nó cần được thiết kế sao cho ước tính chi phí từ trạng thái hiện tại đến trạng thái kết thúc là thực tế và không quá lạc hậu.



Hình 7: Mô tả cách thuật toán A hoạt động (đỉnh bắt đầu A – đỉnh kết thúc K)*

Về đặc điểm của A*:

❖ Nếu không gian các trạng thái là hữu hạn và có giải pháp để tránh việc xét (lặp) lại các trạng thái, thì giải thuật A* là hoàn chỉnh (tìm được lời giải) – nhưng không đảm bảo là tối ưu.

❖ Nếu không gian các trạng thái là hữu hạn và không có giải pháp để tránh việc xét (lặp) lại các trạng thái, thì giải thuật A* là không hoàn chỉnh.

❖ Nếu không gian các trạng thái là vô hạn, thì giải thuật A* là không hoàn chỉnh.

- Thuật toán A* (A-star) là một thuật toán tìm đường tốt nhất trong một đồ thị có trọng số.
- Thuật toán A* được xem là một thuật toán tối ưu vì nó luôn tìm ra đường ngắn nhất từ điểm bắt đầu đến điểm đích, nếu điều kiện đặt ra đúng (tức là nếu heuristic được đặt ra một cách hiệu quả).
- Tuy nhiên, thuật toán A* có thể không tối ưu trong một số trường hợp. Một trong những trường hợp là khi không có hàm ước lượng hợp lý được sử dụng, hoặc khi hàm ước lượng không đảm bảo rằng nó sẽ không bao giờ đánh giá một điểm mới có chi phí lớn hơn so với điểm hiện tại. Trong trường hợp này, thuật toán A* có thể mở rộng điểm không cần thiết, dẫn đến việc tăng tổng chi phí cần xét.

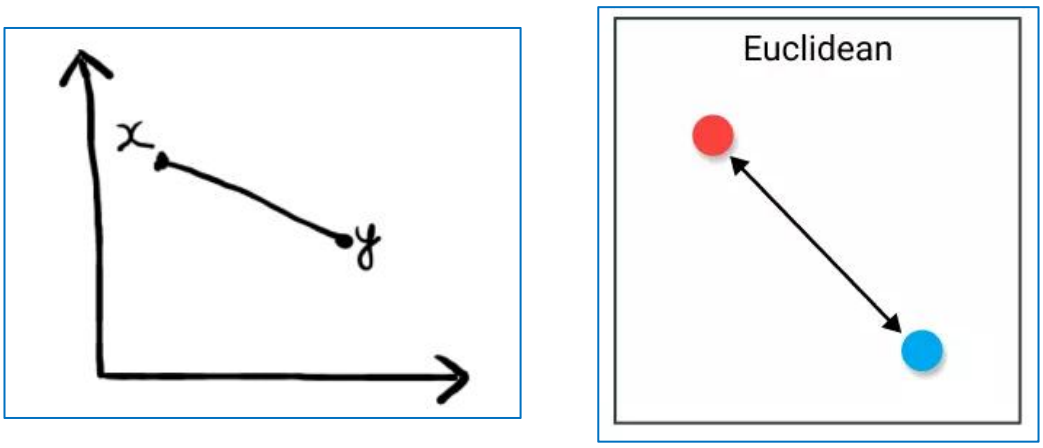
Tính heuristic bằng công thức tính khoảng cách Euclidean và Manhattan

a) Euclidean Distance

- Trong toán học, khoảng cách Euclid (tiếng Anh: Euclidean distance) giữa hai điểm trong không gian Euclid là độ dài của đoạn thẳng nối hai điểm đó.
- Euclidean Distance còn được biết đến với cái tên L_2 distance.

$$D(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

Hình 8: Công thức tính khoảng cách Euclidean



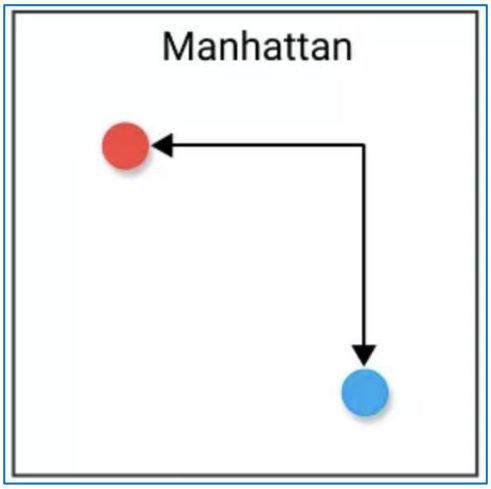
Hình 9: Mô tả khoảng cách Euclidean

b) Manhattan Distance

- Khoảng cách Manhattan, còn được gọi là khoảng cách L1 hay khoảng cách trong thành phố, là một dạng khoảng cách giữa hai điểm trong không gian Euclid với hệ tọa độ Descartes. Đại lượng này được tính bằng tổng chiều dài của hình chiếu của đường thẳng nối hai điểm này trong hệ trục tọa độ Descartes.

$$D(x,y) = \sum_{i=1}^{k} |x_i - y_i|$$

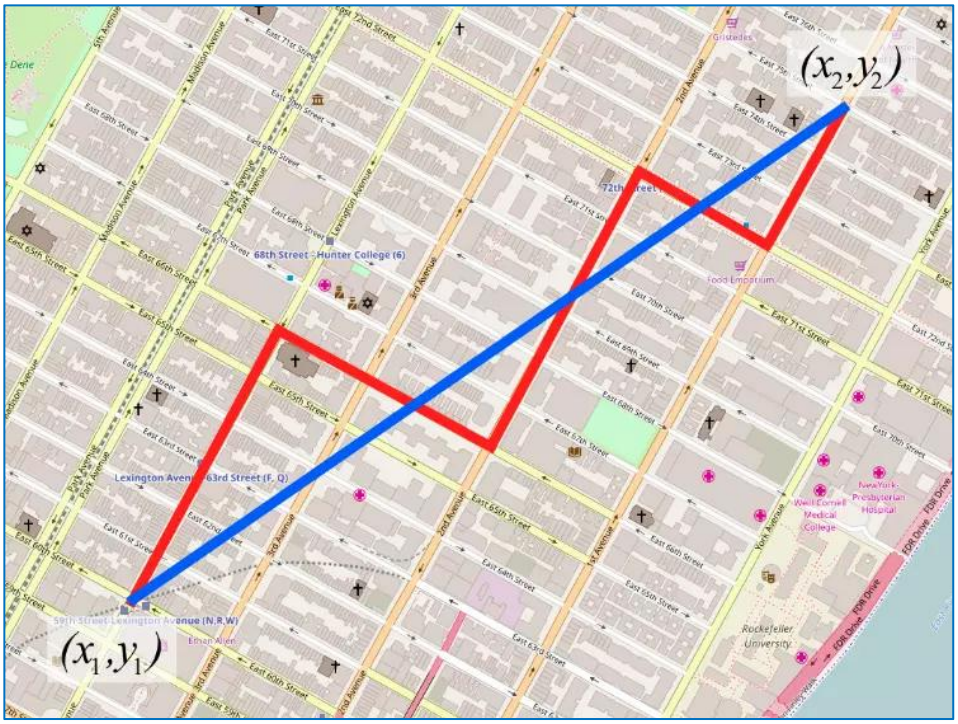
Hình 10: Công thức tính khoảng cách Manhattan



Hình 11: Mô tả khoảng cách Manhattan

Khi nào thì dùng khoảng cách Euclidean cho heuristc tốt hơn? Khi nào thì dùng khoảng cách Manhattan cho heuristc tốt hơn?

Để so sánh Manhattan Distance với Euclide Distance thì ta có thể nhìn bản đồ bên dưới: Euclide Distance là đường chim bay (màu xanh dương) còn Manhattan là đường bộ (màu đỏ) theo các block nhà.



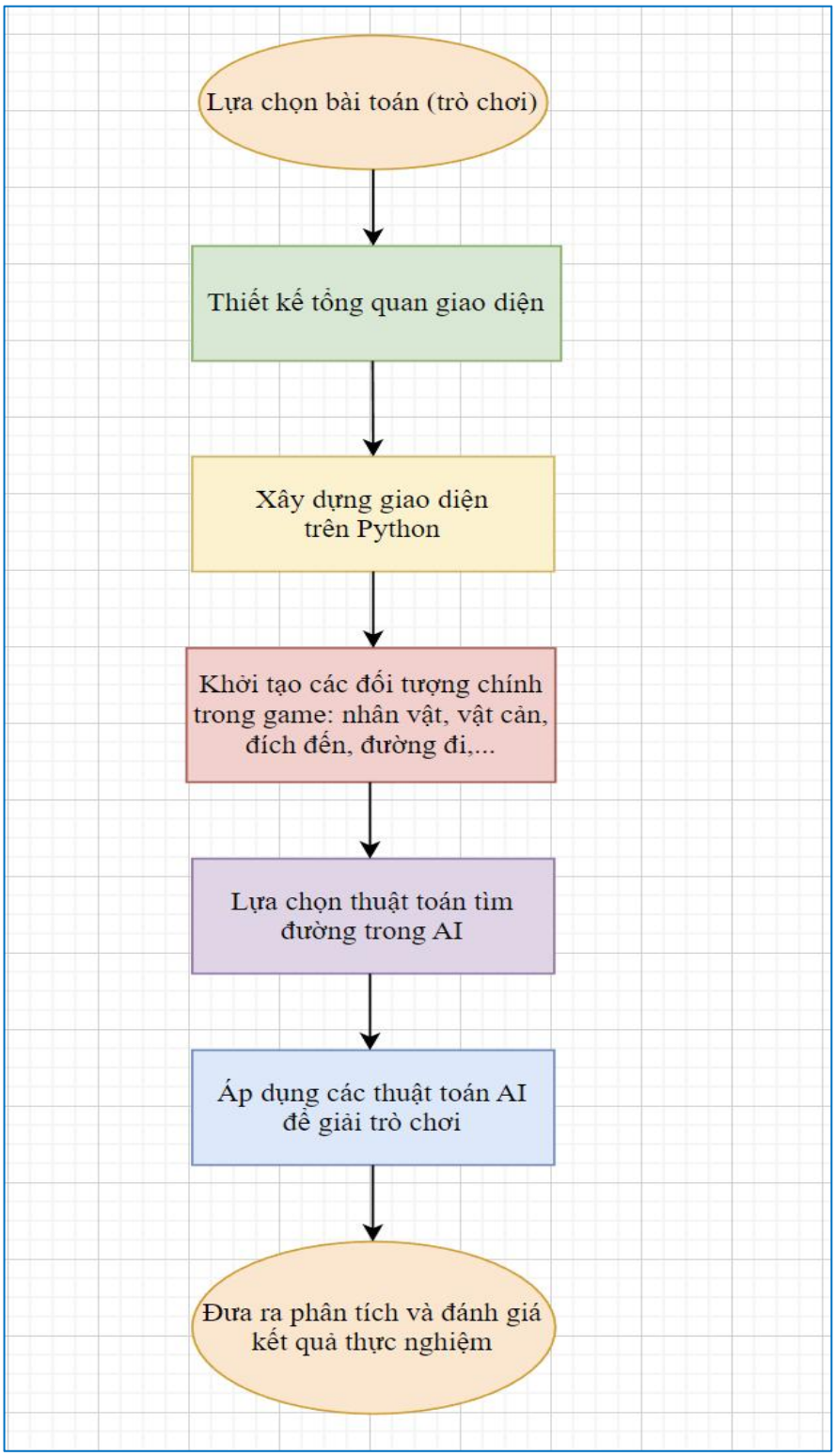
Hình 12: Khoảng cách Euclidean (màu xanh dương) và Manhattan (màu đỏ) trên bản đồ

Từ những lý thuyết về tính khoảng cách Euclidean và khoảng cách Manhattan ta có thể rút ra những nhận xét:

- Khoảng cách Euclidean thường được sử dụng khi đang làm việc với một bản đồ cho phép di chuyển theo bất kỳ hướng nào. Điều này là do khoảng cách Euclidean đảm bảo rằng nó sẽ đánh giá đúng giá trị chi phí từ một đỉnh này đến đỉnh khác.
- Khoảng cách Manhattan thường được sử dụng khi đang làm việc với một bản đồ vuông cho phép di chuyển theo 4 hướng. Điều này là do khoảng cách Manhattan không bao giờ đánh giá quá cao giá trị chi phí giữa các đỉnh. Manhattan đảm bảo rằng nó sẽ đánh giá đúng giá trị chi phí từ một đỉnh này đến đỉnh khác.

PHẦN 3. PHÂN TÍCH, THIẾT KẾ GIẢI PHÁP CHO GAME SOKOBAN

3.1. Sơ đồ khối giải pháp thực hiện bài toán



Hình 13: Sơ đồ khối giải pháp thực hiện bài toán

3.2. Chi tiết về các thuật toán chính

3.2.1. Giải thích module support_function

❖ **Class Result dùng để chứa kết quả trả về**

```
class Result:
    def __init__(self):
        self.approved_states = None
        self.memory = None
        self.map_level = None
        self.time = None
        self.list_board = None
        self.countFindBox = 0
        self.algorithmName = None
        self.countMove = 0
```

Hình 14: Class Result

Giải thích:

- approved_states: Số lượng trạng thái đã duyệt.
- memory: Bộ nhớ sử dụng (MB).
- map_level: Mức độ của bản đồ.
- time: Thời gian thực hiện thuật toán.
- list_board: Danh sách bảng trạng thái và số trạng thái đã duyệt.
- countFindBox: Số lần tìm thấy trạng thái chiến thắng.
- algorithmName: Tên thuật toán.
- countMove: Số lần di chuyển.

❖ **Hàm kiểm tra các điểm kiểm tra đã được che phủ**

```
def check_win(board, list_check_point):
    '''trả về True nếu tất cả các điểm kiểm tra được che phủ bởi các hộp''
    for p in list_check_point:
        if board[p[0]][p[1]] != '$':
            return False
    return True
```

Hình 15: Hàm check_win

Giải thích:

- check_win: Kiểm tra xem tất cả các điểm kiểm tra trong danh sách đã được che phủ bởi các hộp trên bảng hay chưa.

❖ **Hàm tạo bảng mới giống mảng hiện tại**

```
def assign_matrix(board):
    '''trả về bảng giống như bảng đầu vào'''
    return [[board[x][y] for y in range(len(board[0]))] for x in range(len(board))]
```

Hình 16: Hàm assign_matrix

Giải thích:

- assign_matrix : Sẽ dùng vòng lặp giúp copy các giá trị của mạng hiện tại và trả về một bảng mới giống hệt . Hàm này giúp hỗ trợ trong việc tạo trạng thái mới

❖ **Hàm tìm vị trí nhân vật**

```
def find_position_player(board):
    '''trả về vị trí của người chơi trong bảng'''
    for x in range(len(board)):
        for y in range(len(board[0])):
            if board[x][y] == '@':
                return (x, y)
    return (-1, -1)  # bảng lỗi
```

Hình 17: Hàm find_position_player

Giải thích:

- find_position_player: Tìm và trả về vị trí của người chơi ('@') trên bảng. Nếu không tìm thấy, trả về (-1, -1) để chỉ ra lỗi.

❖ **Hàm so sách 2 ma trận**

```
def compare_matrix(board_A, board_B):
    '''trả về True nếu bảng A giống bằng bảng B'''
    if len(board_A) != len(board_B) or len(board_A[0]) != len(board_B[0]):
        return False
    for i in range(len(board_A)):
        for j in range(len(board_A[0])):
            if board_A[i][j] != board_B[i][j]:
                return False
    return True
```

Hình 18: Hàm compare_matrix

Giải thích:

- compare_matrix: Trả về True nếu bảng A giống bằng bảng B. Dùng so sánh từng phần tử của hai ma trận.

❖ **Hàm kiểm tra có hộp nằm trên điểm kiểm tra**

```
def is_box_on_check_point(box, list_check_point):
    for check_point in list_check_point:
        if box[0] == check_point[0] and box[1] == check_point[1]:
            return True
    return False
```

Hình 19: Hàm is_box_on_check_point

Giải thích:

- is_box_on_check_point: Kiểm tra xem một hộp cụ thể (box) có đặt trên một điểm kiểm tra trong danh sách (list_check_point) hay không..

❖ **Hàm kiểm tra có ít nhất một hộp kẹt cạn không**

```
def check_in_corner(board, x, y, list_check_point):
    '''trả về True nếu board[x][y] ở góc'''
    if board[x - 1][y - 1] == '#':
        if board[x - 1][y] == '#' and board[x][y - 1] == '#':
            if not is_box_on_check_point((x, y), list_check_point):
                return True
    if board[x + 1][y - 1] == '#':
        if board[x + 1][y] == '#' and board[x][y - 1] == '#':
            if not is_box_on_check_point((x, y), list_check_point):
                return True
    if board[x - 1][y + 1] == '#':
        if board[x - 1][y] == '#' and board[x][y + 1] == '#':
            if not is_box_on_check_point((x, y), list_check_point):
                return True
    if board[x + 1][y + 1] == '#':
        if board[x + 1][y] == '#' and board[x][y + 1] == '#':
            if not is_box_on_check_point((x, y), list_check_point):
                return True
    return False
```

Hình 20: Hàm check_in_corner

Giải thích:

- check_in_corner: Kiểm tra xem ô tại vị trí (x, y) trên bảng có thuộc góc không.

❖ **Hàm tìm vị trí hộp trên bảng**

```
def find_boxes_position(board):
    result = []
    for i in range(len(board)):
        for j in range(len(board[0])):
            if board[i][j] == '$':
                result.append((i, j))
    return result
```

Hình 21: Hàm find_boxes_position

Giải thích:

- find_boxes_position: Tìm và trả về danh sách vị trí của tất cả các hộp ('$') trên bảng.

❖ **Hàm kiểm tra box có thể di chuyển không**

```
def is_box_can_be_moved(board, box_position):
    left_move = (box_position[0], box_position[1] - 1)
    right_move = (box_position[0], box_position[1] + 1)
    up_move = (box_position[0] - 1, box_position[1])
    down_move = (box_position[0] + 1, box_position[1])
    if (board[left_move[0]][left_move[1]] == ' ' or board[left_move[0]][left_move[1]] == '%' or board[left_move[0]][
        left_move[1]] == '@') and board[right_move[0]][right_move[1]] != '#' and board[right_move[0]][
        right_move[1]] != '$':
        return True
    if (board[right_move[0]][right_move[1]] == ' ' or board[right_move[0]][right_move[1]] == '%' or
        board[right_move[0]][right_move[1]] == '@') and board[left_move[0]][left_move[1]] != '#' and \
            board[left_move[0]][left_move[1]] != '$':
        return True
    if (board[up_move[0]][up_move[1]] == ' ' or board[up_move[0]][up_move[1]] == '%' or board[up_move[0]][
        up_move[1]] == '@') and board[down_move[0]][down_move[1]] != '#' and board[down_move[0]][down_move[1]] != '$':
        return True
    if (board[down_move[0]][down_move[1]] == ' ' or board[down_move[0]][down_move[1]] == '%' or board[down_move[0]][
        down_move[1]] == '@') and board[up_move[0]][up_move[1]] != '#' and board[up_move[0]][up_move[1]] != '$':
        return True
    return False
```

Hình 22: Hàm is_box_can_be_moved

Giải thích:

- is_box_can_be_moved: Kiểm tra xem hộp tại vị trí box_position có thể được di chuyển không.

❖ **Hàm kiểm tra box kẹt cạn**

```
def is_all_boxes_stuck(board, list_check_point):
    box_positions = find_boxes_position(board)
    result = True
    for box_position in box_positions:
        if is_box_on_check_point(box_position, list_check_point):
            return False
        if is_box_can_be_moved(board, box_position):
            result = False
    return result
```

Hình 23: Hàm is_all_boxed_stuck

Giải thích:

- is_all_boxes_stuck: Kiểm tra xem tất cả các hộp trên bảng có bị kẹt hay không.

❖ **Hàm kiểm tra bảng không thể thắng**

```
def is_board_can_not_win(board, list_check_point):
    '''trả về True nếu hộp nằm ở góc của bức tường -> không thể thắng'''
    for x in range(len(board)):
        for y in range(len(board[0])):
            if board[x][y] == '$':
                if check_in_corner(board, x, y, list_check_point):
                    return True
    return False
```

Hình 24: Hàm is_board_can_not_win

Giải thích:

- is_board_can_not_win: Kiểm tra xem bảng có chứa hộp nằm ở góc của bức tường hay không.

❖ **Hàm danh sách có thể di chuyển**

```
def get_next_pos(board, cur_pos):
    '''trả về danh sách vị trí mà người chơi có thể di chuyển đến từ vị trí hiện tại'''
    x, y = cur_pos[0], cur_pos[1]
    list_can_move = []
    # DI CHUYỂN LÊN (x - 1, y)
    if 0 <= x - 1 < len(board):
        value = board[x - 1][y]
        if value == ' ' or value == '%':
            list_can_move.append((x - 1, y))
        elif value == '$' and 0 <= x - 2 < len(board):
            next_pos_box = board[x - 2][y]
            if next_pos_box != '#' and next_pos_box != '$':
                list_can_move.append((x - 1, y))
    # DI CHUYỂN XUỐNG (x + 1, y)
    if 0 <= x + 1 < len(board):
        value = board[x + 1][y]
        if value == ' ' or value == '%':
            list_can_move.append((x + 1, y))
        elif value == '$' and 0 <= x + 2 < len(board):
            next_pos_box = board[x + 2][y]
            if next_pos_box != '#' and next_pos_box != '$':
                list_can_move.append((x + 1, y))
    # DI CHUYỂN TRÁI (x, y - 1)
    if 0 <= y - 1 < len(board[0]):
        value = board[x][y - 1]
```

```
    # DI CHUYỂN TRÁI (x, y - 1)
    if 0 <= y - 1 < len(board[0]):
        value = board[x][y - 1]
        if value == ' ' or value == '%':
            list_can_move.append((x, y - 1))
        elif value == '$' and 0 <= y - 2 < len(board[0]):
            next_pos_box = board[x][y - 2]
            if next_pos_box != '#' and next_pos_box != '$':
                list_can_move.append((x, y - 1))
    # DI CHUYỂN PHẢI (x, y + 1)
    if 0 <= y + 1 < len(board[0]):
        value = board[x][y + 1]
        if value == ' ' or value == '%':
            list_can_move.append((x, y + 1))
        elif value == '$' and 0 <= y + 2 < len(board[0]):
            next_pos_box = board[x][y + 2]
            if next_pos_box != '#' and next_pos_box != '$':
                list_can_move.append((x, y + 1))
    return list_can_move
```

Hình 25: Hàm get_next_pos

Giải thích:

- get_next_pos: Trả về danh sách vị trí mà người chơi có thể di chuyển đến từ vị trí hiện tại (cur_pos) trên bảng (board).

❖ **Hàm di chuyển bảng**

```
def move_with_cost(board, next_pos, cur_pos, list_check_point):
    '''trả về một bảng mới sau khi di chuyển'''
    # TẠO BẢNG MỚI NHƯ BẢNG HIỆN TẠI
    new_board = assign_matrix(board)
    # TẠO BIẾN COST
    new_cost = 0
    # TÌM VỊ TRÍ TIẾP THEO NẾU DI CHUYỂN ĐẾN HỘP
    if new_board[next_pos[0]][next_pos[1]] == '$':
        x = 2 * next_pos[0] - cur_pos[0]
        y = 2 * next_pos[1] - cur_pos[1]
        new_board[x][y] = '$'
        new_cost += 2  # Di chuyển kèm theo hộp, cost + 2
    # DI CHUYỂN NGƯỜI CHƠI ĐẾN VỊ TRÍ MỚI
    new_board[next_pos[0]][next_pos[1]] = '@'
    new_board[cur_pos[0]][cur_pos[1]] = ' '
    # KIỂM TRA NẾU TẠI VỊ TRÍ ĐIỂM KIỂM TRA KHÔNG CÓ HỘP HAY NHÂN VẬT THÌ CẬP NHẬT % NHƯ CŨ
    for p in list_check_point:
        if new_board[p[0]][p[1]] == ' ':
            new_board[p[0]][p[1]] = '%'
    new_cost += 1
    return new_board ,new_cost
```

Hình 26: Hàm move_with_cost

Giải thích:

- move_with_cost: Trả về một bảng mới sau khi di chuyển và chi phí của di chuyển.

❖ **Hàm tìm danh sách checkpoint**

```
def find_list_check_point(board):
    '''trả về danh sách điểm kiểm tra từ bảng
        nếu không có bất kỳ điểm kiểm tra nào, trả về danh sách trống
        nó sẽ kiểm tra số hộp, nếu số hộp < số điểm kiểm tra
            trả về danh sách [(-1, -1)]'''
    list_check_point = []
    num_of_box = 0
    '''KIỂM TRA TOÀN BỘ BẢNG ĐỂ TÌM ĐIỂM KIỂM TRA VÀ SỐ HỘP'''
    for x in range(len(board)):
        for y in range(len(board[0])):
            if board[x][y] == '$':
                num_of_box += 1
            elif board[x][y] == '%':
                list_check_point.append((x, y))
    '''KIỂM TRA NẾU SỐ HỘP < SỐ LƯỢNG ĐIỂM KIỂM TRA'''
    if num_of_box < len(list_check_point):
        return [(-1, -1)]
    return list_check_point
```

Hình 27: Hàm find_list_check_point

Giải thích:

- find_list_check_point: Trả về danh sách các điểm kiểm tra từ bảng..

❖ **Hàm di chuyển khi tự chơi**

```
        elif key == 'DOWN':
            next_pos = (cur_pos[0] + 1, cur_pos[1])

        # Kiểm tra xem nước đi tiếp theo có hợp lệ hay không
        if 0 <= next_pos[0] < len(board) and 0 <= next_pos[1] < len(board[0]) and board[next_pos[0]][
            next_pos[1]] != '#':
            if board[next_pos[0]][next_pos[1]] == '$':
                # Kiểm tra xem hộp có thể được đẩy hay không
                if key == 'LEFT':
                    next_box_pos = (next_pos[0], next_pos[1] - 1)
                elif key == 'RIGHT':
                    next_box_pos = (next_pos[0], next_pos[1] + 1)
                elif key == 'UP':
                    next_box_pos = (next_pos[0] - 1, next_pos[1])
                elif key == 'DOWN':
                    next_box_pos = (next_pos[0] + 1, next_pos[1])
                if 0 <= next_box_pos[0] < len(board) and 0 <= next_box_pos[1] < len(board[0]) and (
                        board[next_box_pos[0]][next_box_pos[1]] == ' ' or board[next_box_pos[0]][
                    next_box_pos[1]] == '%'):
                    new_board = move(board, next_pos, cur_pos, list_check_point)
                    return new_board
            else:
                new_board = move(board, next_pos, cur_pos, list_check_point)
                return new_board
    return board  # Trả về bảng hiện tại nếu nước đi không hợp lệ
```

```
            else:
                new_board = move(board, next_pos, cur_pos, list_check_point)
                return new_board
    return board  # Trả về bảng hiện tại nếu nước đi không hợp lệ
    if board[next_pos[0]][next_pos[1]] == '$':
        # Kiểm tra xem hộp có thể được đẩy hay không
        if key == 'LEFT':
            next_box_pos = (next_pos[0], next_pos[1] - 1)
        elif key == 'RIGHT':
            next_box_pos = (next_pos[0], next_pos[1] + 1)
        elif key == 'UP':
            next_box_pos = (next_pos[0] - 1, next_pos[1])
        elif key == 'DOWN':
            next_box_pos = (next_pos[0] + 1, next_pos[1])
        if 0 <= next_box_pos[0] < len(board) and 0 <= next_box_pos[1] < len(board[0]) and (board[next_box_pos[0]][next_
            new_board = move(board, next_pos, cur_pos, list_check_point)
            return new_board
    else:
        new_board = move(board, next_pos, cur_pos, list_check_point)
        return new_board

    return board  # Trả về bảng hiện tại nếu nước đi không hợp lệ
```

Hình 28: Hàm move_board_by_key

Giải thích:

- move_board_by_key: Di chuyển bảng theo hành động người dùng.

3.2.2 Áp dụng các thuật toán trong AI

Các giải thuật tìm kiếm đường đi được sử dụng trong đề tài là:

- Breadth First Search (BFS)
- Depth First Search (DFS)
- Depth Limited Search (DLS)
- Uniform Cost Search (UCS)
- Greedy Best First Search (GBFS)
- A Star (A*)

Trong đó:

- GBFS sử dụng hàm đánh giá heuristic bằng công thức tính khoảng cách Euclidean
- Mỗi file AStar.py sẽ sử dụng các hàm đánh giá heuristic khác nhau bằng công thức tính khoảng cách Euclidean và Manhattan.

Trong source code của chúng em, tất cả các file có định dạng ten_thuat_toan.py đều được xây dựng bằng hai phần chính, gồm class state và def ten_thuat_toan. Bên dưới chúng em sẽ chỉ đưa vào một vài phân đoạn code của một số thuật toán để minh họa cho quá trình thực hiện.

❖ **Class state**

Class state của các thuật toán BFS, DFS và DLS về cơ bản sẽ giống nhau.

```
class state:
    def __init__(self, board, state_parent, list_check_point):
        '''lưu trạng thái hiện tại và trạng thái cha của trạng thái này'''
        self.board = board
        self.state_parent = state_parent
        self.check_points = deepcopy(list_check_point)
    '''HÀM ĐỆ QUY ĐỂ TRUY VẾT ĐẾN TRẠNG THÁI ĐẦU TIÊN NẾU TRẠNG THÁI HIỆN TẠI LÀ MỤC TIÊU'''
    def get_line(self):
        '''sử dụng vòng lặp để tìm danh sách trạng thái từ đầu đến trạng thái hiện tại'''
        if self.state_parent is None:
            return [self.board]
        return (self.state_parent).get_line() + [self.board]
```

Hình 29 : Code class state của ba thuật toán BFS, DFS và DLS

Giải thích:

- Phương thức init nhằm khởi tạo một thể hiện của lớp state. Phương thức này được gọi khi tạo một đối tượng mới từ lớp state. Nó nhận vào một board, state_parent, và list_check_point sau đó khởi tạo các thuộc tính tương ứng của đối tượng như board , state_parent và check_points.
- Phương thức get_line để truy vết một chuỗi các trạng thái từ trạng thái hiện tại về trạng thái ban đầu. Nó làm điều này bằng cách gọi đệ quy phương thức get_line trên trạng thái cha cho đến khi đạt đến trạng thái không có trạng thái cha (tức là trạng thái ban đầu).

--> Tóm lại, lớp state được sử dụng để quản lý và truy vết các trạng thái trong trò chơi Sokoban. Đây là một phần thiết yếu của chương trình xây dựng game đẩy hộp.

Class state của các thuật toán UCS, GBFS và A* sẽ có một số điểm khác biệt:

```python
class state:
    def __init__(self, board, state_parent, list_check_point , cost):
        self.board = board
        self.state_parent = state_parent
        self.cost = cost
        self.check_points = deepcopy(list_check_point)

    def get_line(self):
        if self.state_parent is None:
            return [self.board]
        return (self.state_parent).get_line() + [self.board]

    def __gt__(self, other):
        return self.cost > other.cost

    def __lt__(self, other):
        return self.cost < other.cost
```

Hình 30: Code class state của thuật toán UCS

```python
class state:
    def __init__(self, board, state_parent, list_check_point):
        self.board = board
        self.state_parent = state_parent
        self.heuristic = 0 #h(n), không cộng g(n)
        self.check_points = deepcopy(list_check_point)

    def get_line(self):
        if self.state_parent is None:
            return [self.board]
        return (self.state_parent).get_line() + [self.board]

    def compute_euclidean_heuristic_for_best_first_search(self):
        list_boxes = spf.find_boxes_position(self.board)
        if self.heuristic == 0:
            total_distance = 0
            for i in range(len(list_boxes)):
                box = list_boxes[i]
                checkpoint = self.check_points[i]
                distance = math.sqrt((box[0] - checkpoint[0])**2 + (box[1] - checkpoint[1])**2)
                total_distance += distance
            self.heuristic = total_distance
        return self.heuristic

    def __gt__(self, other):
        if self.compute_euclidean_heuristic_for_best_first_search() > other.compute_euclidean_heuristic_for_best_first_search():
            return True
        else:
            return False      javier, 2 weeks ago • feat: delete duplicate support_function and creat…

    def __lt__(self, other):
        if self.compute_euclidean_heuristic_for_best_first_search() < other.compute_euclidean_heuristic_for_best_first_search():
            return True
        else:
            return False
```

Hình 31: Code class state của thuật toán GBFS

```python
class State:
    def __init__(self, board, state_parent, list_check_point):
        self.board = board
        self.state_parent = state_parent
        self.cost = 1
        self.heuristic = 0
        self.check_points = deepcopy(list_check_point)

    def compute_heuristic(self):
        if self.heuristic == 0:
            list_boxes = spf.find_boxes_position(self.board)
            self.heuristic = self.cost + abs(sum(list_boxes[i][0] + list_boxes[i][1] - self.check_points[i][0] - self.check_points[i][1] for i in range(len(list_boxes))))
        return self.heuristic

    def get_line(self):
        if self.state_parent is None:
            return [self.board]
        return self.state_parent.get_line() + [self.board]

    def __gt__(self, other):
        return self.compute_heuristic() > other.compute_heuristic()

    def __lt__(self, other):
        return self.compute_heuristic() < other.compute_heuristic()
```

*Hình 32: Code class state của thuật toán A**

Giải thích:

- Nhìn chung class state của UCS, GBFS và A* đều có các phương thức init, get_line và cách giải thích cũng tương tự như phần giải thích class state của BFS, DFS và DLS.
- Tuy nhiên, vẫn có các điểm khác biệt sau:

  + Cả ba thuật toán đều có thêm phương thức __gt__ và __lt__ tức là greater than (lớn hơn) và less than (nhỏ hơn). Cả hai phương thức này dùng để so sánh các đối tượng state dựa trên giá trị của cost (g(n)) hoặc giá trị của heurisic (h(n)).

  + Ở giải thuật GBFS và A* sẽ có thêm hàm đánh giá heuristic: def compute_euclidean_heuristic_for_best_first_search (GBFS); def compute_heuristic (cho A* tính h(n) bằng Manhattan); def compute_euclidean_heuristic (cho A* tính h(n) bằng Euclidean). Mục đích của các hàm này là dùng để tính toán giá trị heuristic cho hai thuật toán GBFS và A* nhằm tìm ra đường đi có chi phí tốt nhất.

❖ **Def ten_thuat_toan**

Ở phần này chúng em sẽ chỉ lấy hai thuật toán trong 6 thuật toán để minh họa cách làm, hai thuật toán đó là BFS và A*(tính heuristic bằng công thức tính khoảng cách Manhattan).

■ def BFS_search

```python
def BFS_search(board, list_check_point):
    start_time = time.time()
    box_push_count = 0

    if spf.check_win(board, list_check_point):
        print("Found Win")
        return [board]

    ''' KHỞI TẠO TRẠNG THÁI BẮT ĐẦU '''
    start_state = state(board, None, list_check_point)

    ''' KHỞI TẠO 2 DANH SÁCH ĐƯỢC SỬ DỤNG CHO TÌM KIẾM BFS '''
    queue = deque([start_state])
    visited = set()

    while queue:
        now_state = queue.popleft()
        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)
            new_board, move_cost = spf.move_with_cost(now_state.board, next_pos, cur_pos, list_check_point)
            if move_cost > 1:
                box_push_count += 1
            board_tuple = tuple(map(tuple, new_board))  # convert board to tuple of tuples so it can be added to a set
            if board_tuple in visited:
                continue
            if spf.is_board_can_not_win(new_board, list_check_point):      javier, last week • feat: done normal playing
                continue
            if spf.is_all_boxes_stuck(new_board, list_check_point):
                continue
            new_state = state(new_board, now_state, list_check_point)

            if spf.check_win(new_board, list_check_point):
                print("\nBreadth First Search")
                print("Found Win")
                print("  Số trạng thái đã duyệt : {} ".format(len(visited)))
                print("  Số lần đẩy hộp : {} ".format(box_push_count))
                process = psutil.Process(os.getpid())
                memory_usage = process.memory_info().rss / (1024**2)
                result = spf.Result()
                result.countFindBox = box_push_count
                result.approved_states = len(visited)
                result.memory = memory_usage
                result.time = time.time()
                result.list_board = (new_state.get_line(), len(visited))
                result.algorithmName = "Breadth First Search"
                return result

            visited.add(board_tuple)
            queue.append(new_state)
            process = psutil.Process(os.getpid())
            memory_usage = process.memory_info().rss / (1024**2)

            end_time = time.time()
            if end_time - start_time > TIME_OUT:
                return result
```

Hình 33: Code def BFS_search của thuật toán BFS

■ def AStar_manhattan_Search

```python
def AStar_manhattan_Search(board, list_check_point):
    start_time = time.time()
    result = spf.Result()
    box_push_count = 0

    if spf.check_win(board, list_check_point):
        print("Found Win")
        return [board]

    start_state = State(board, None, list_check_point)
    list_state = set()

    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, move_cost = spf.move_with_cost(now_state.board, next_pos, cur_pos, list_check_point)
            if move_cost > 1:
                box_push_count += 1
            board_tuple = tuple(map(tuple, new_board))

            if board_tuple in list_state:
                continue

            if spf.is_board_can_not_win(new_board, list_check_point) or spf.is_all_boxes_stuck(new_board, list_check_point):
                continue

            new_state = State(new_board, now_state, list_check_point)

            if spf.check_win(new_board, list_check_point):
                print("\nA*(Manhattan)")
                print("Found Win")
                print("  Số trạng thái đã duyệt : {} ".format(len(list_state)))
                process = psutil.Process(os.getpid())
                memory_usage = process.memory_info().rss / (1024**2)

                result = spf.Result()
                result.countFindBox = box_push_count
                result.approved_states = len(list_state)
                result.memory = memory_usage
                result.time = time.time()
                result.list_board = (new_state.get_line(), len(list_state))
                result.algorithmName = "A*(Manhattan)"

                return result

            list_state.add(board_tuple)
            heuristic_queue.put(new_state)

            end_time = time.time()
            if end_time - start_time > spf.TIME_OUT:
                return result

    print("Not Found")
    return result
```

Hình 34: Code def AStar_manhattan_Search của thuật toán A (Manhattan)*

Giải thích

- Giống nhau

+ Tất cả các hàm có định dạng def ten_thuat_toan đều khởi tạo một trạng thái bắt đầu và kiểm tra xem trạng thái bắt đầu có thắng không. Nếu có, các hàm này sẽ trả về trạng thái đó. Còn nếu chưa phải là trạng thái chiến thắng thì sẽ khởi tạo trạng thái bắt đầu.

+ Các hàm đều sử dụng một vòng lặp while để duyệt qua tất cả các trạng thái có thể đạt được từ trạng thái hiện tại và kiểm tra xem trạng thái mới có thắng không. Nếu có, hàm sẽ trả về kết quả tìm kiếm và thông tin về trạng thái.

- Khác nhau:

+ Trong BFS_search, cấu trúc dữ liệu này là một queue.

+ Trong DFS_search và DLS_Search thì dùng cấu trúc dữ liệu stack.

+ Trong UCS_Search, Best_First_Search AStar_Search và AStar_Search1 sẽ dùng PriorityQueue.

- Mục đích của các hàm này là đều dùng để tìm kiếm trạng thái chiến thắng trong trò chơi Sokoban.

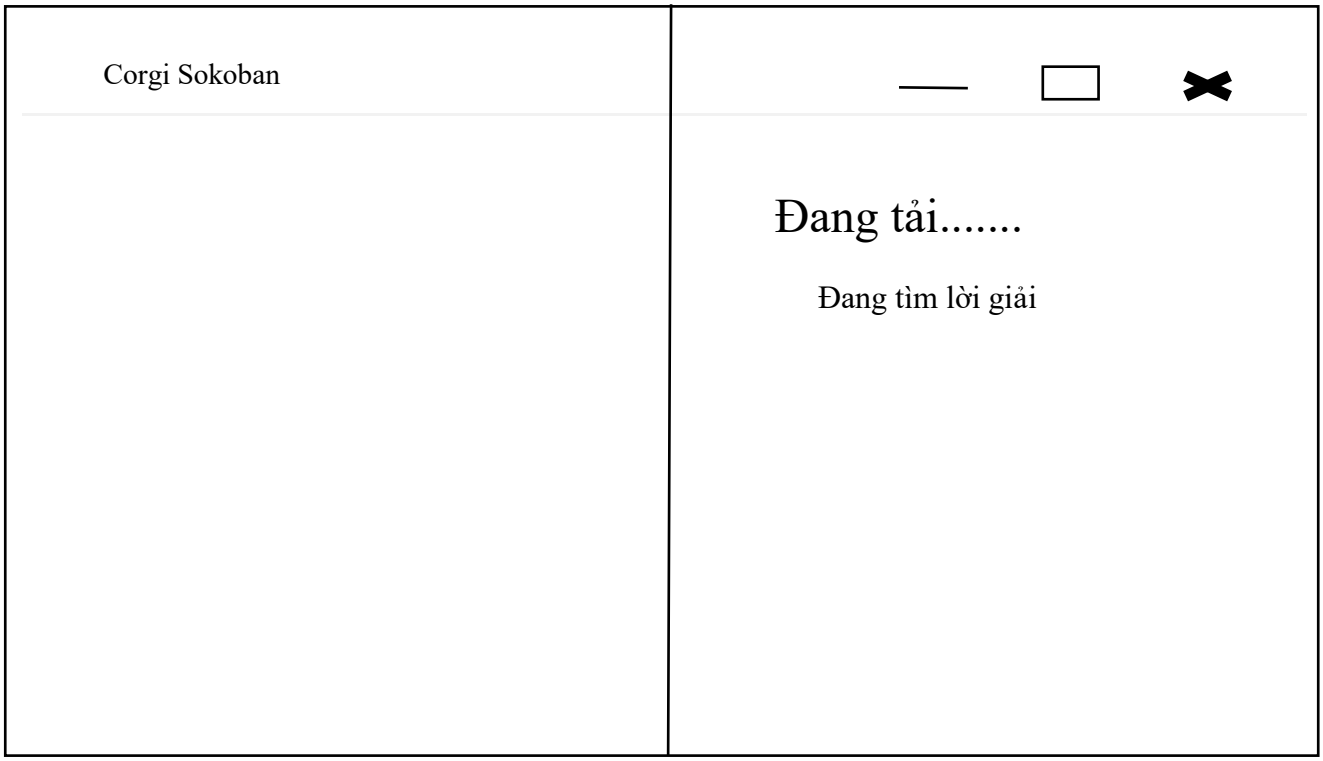
PHẦN 4. THỰC NGHIỆM, ĐÁNH GIÁ, VÀ PHÂN TÍCH KẾT QUẢ

4.1 Thiết kế tổng quan giao diện

4.1.1. Giao diện khởi tạo game

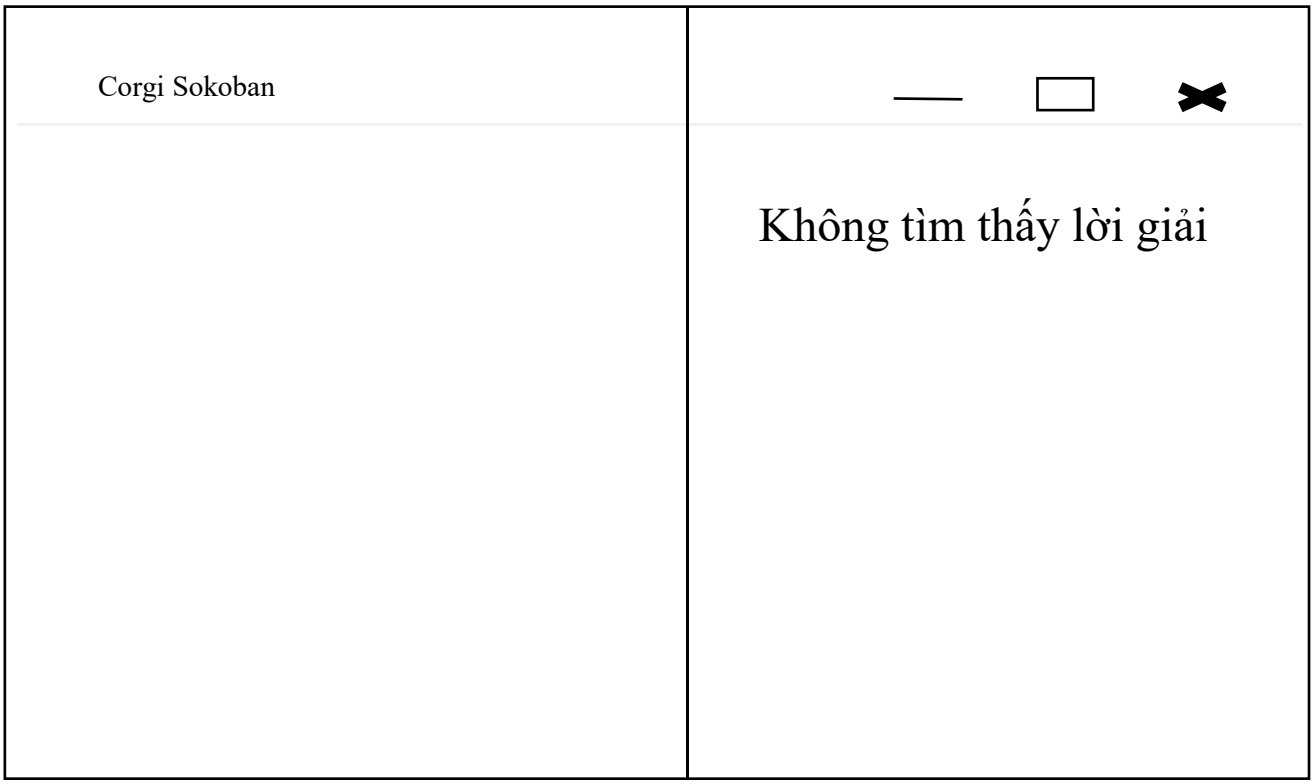
Hình 35: Giao diện tổng quan khởi tạo game

4.1.2. Giao diện loading game



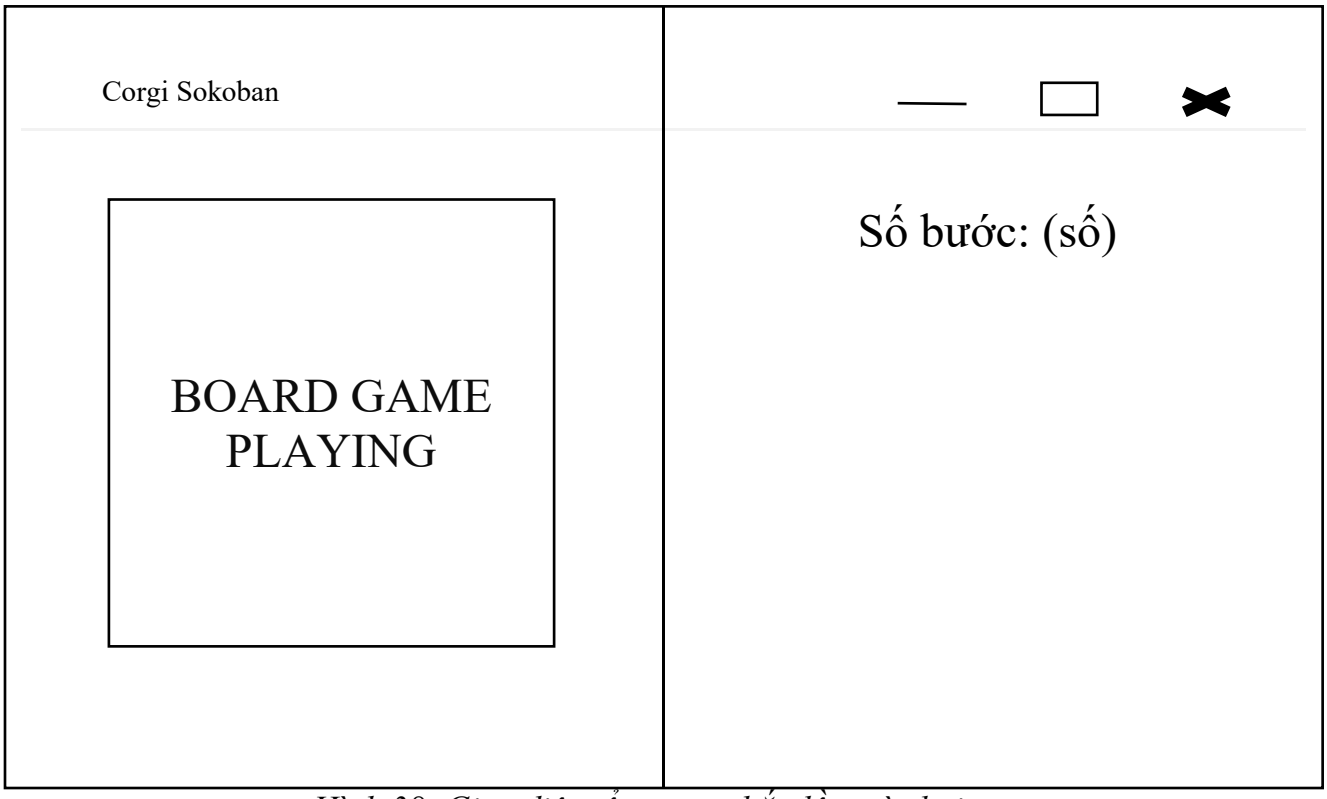
Hình 36: Giao diện tổng quan loading game

4.1.3. Giao diện không tìm thấy lời giải



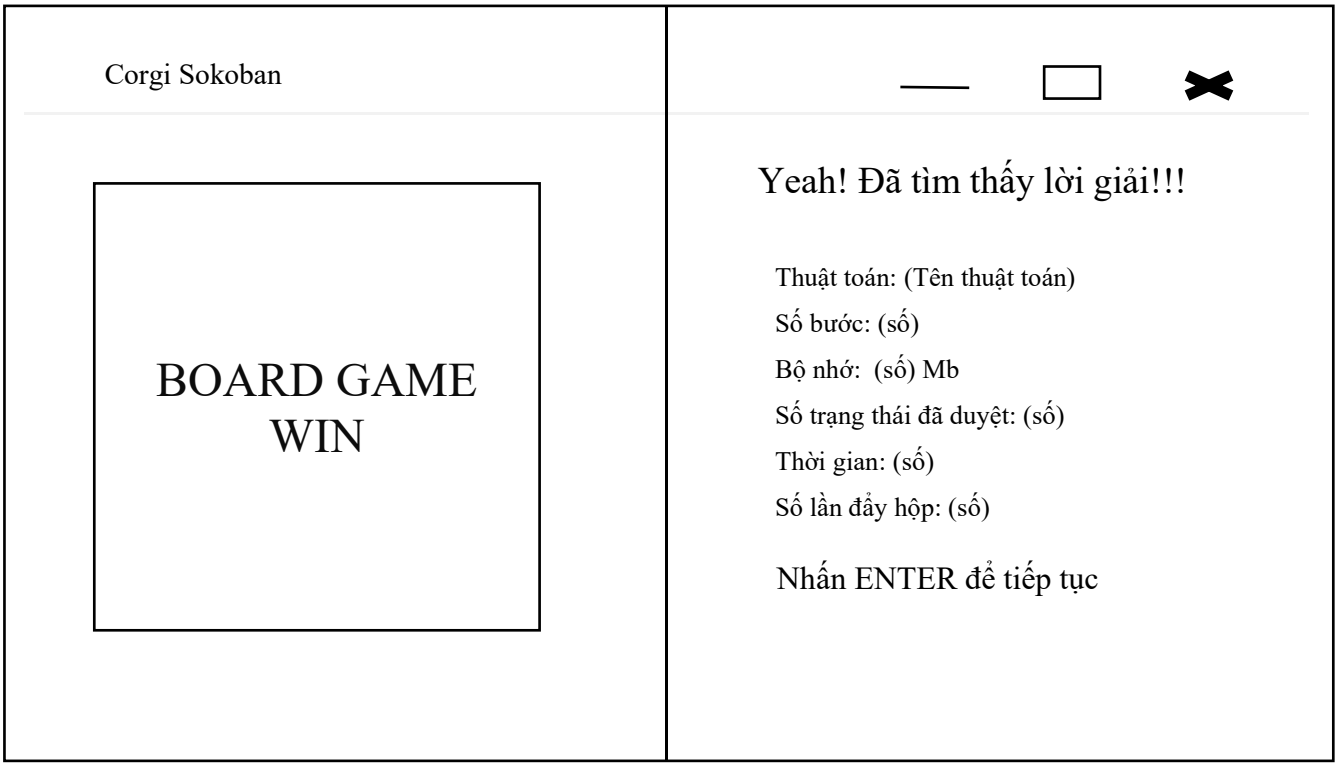
Hình 37: Giao diện tổng quan không tìm thấy lời giải

4.1.4 Giao diện bắt đầu trò chơi



Hình 38: Giao diện tổng quan bắt đầu trò chơi

4.1.5. Giao diện khi tìm được kết quả



Hình 39: Giao diện tổng quan khi tìm được kết quả

4.2. Xây dựng giao diện trên Python

4.2.1 Giao diện khởi tạo trò chơi Sokoban

Khi chạy file main.py màn hình sẽ hiện ra giao diện khởi tạo.

Mô tả giao diện khởi tạo:

- Kích thước màn hình game là 1024x640
- Trong giao diện khởi tạo sẽ có:
  + Tên trò chơi
  + Hướng dẫn cách chọn map
  + Level của map
  + Board game
  + Hướng dẫn đổi thuật toán
  + Tên chế độ chơi: Normal: người chơi; Tên các thuật toán: Máy chơi

Trò chơi có tất cả là 30 map, các map là cố định và không thể thay đổi, độ khó của trò chơi sẽ được nâng cao qua từng map, giúp người tạo ra cảm giác muốn khám phá và sử dụng hết khả năng của mình để phá đảo trò chơi.

Hình 40: Giao diện khởi tạo game

Phân bố độ khó của trò chơi thông qua map:

- Mức dễ: Từ Map 1 đến Map 10
- Mức trung bình: Từ Map 11 đến Map 20
- Mức khó: Từ Map 21 đến Map 30

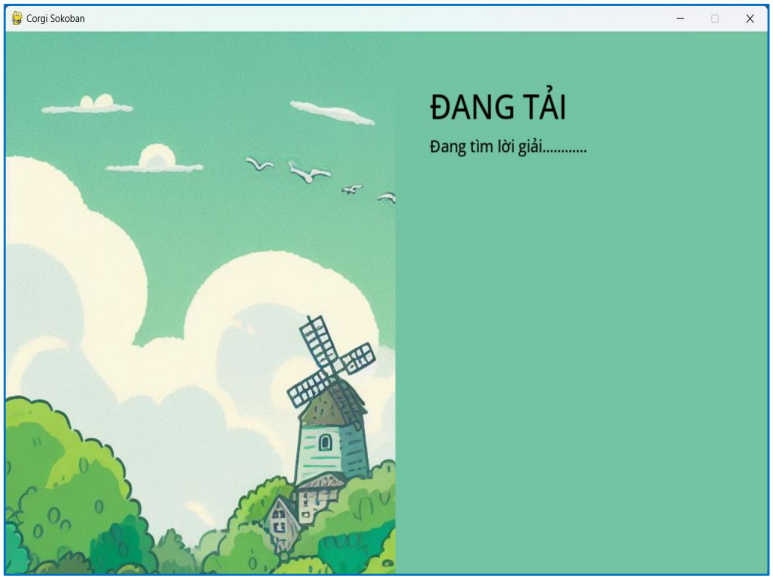
Thông tin các chức năng:

THÔNG TIN CÁC CHỨC NĂNG		
Số thứ tự	**Tên chức năng**	**Ý nghĩa**
1.	Map	Nhấn phím mũi tên < > trên bàn phím để chọn map
2.	Tên chế độ chơi (Normal, BFS, DFS, Greedy Best First Search, A*(Euclidean), A*(Manhattan), Uniform Cost Search)	Nhấn phím Space trên bàn phím để chọn các giải thuật

Bảng thông tin các chức năng

4.2.2. Giao diện khi loading trò chơi

Khi người dùng chọn các thuật toán để hệ thống tự tìm lời giải cho trò chơi thì giao diện game khởi tạo sẽ chuyển sang giao diện loading game như bên dưới:



Hình 41: Giao diện loading game

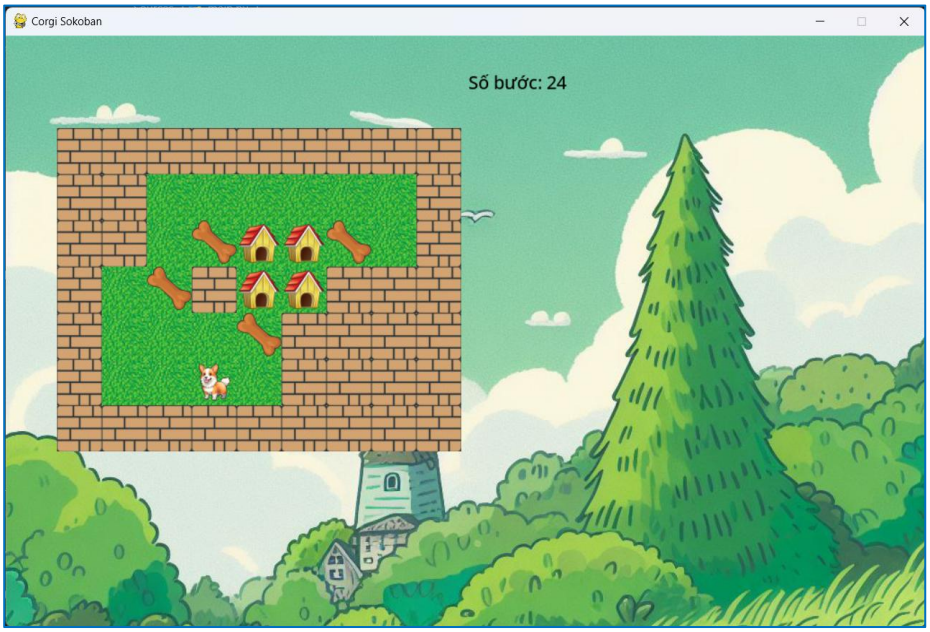
4.2.3 Giao diện không tìm thấy lời giải

Sau một khoảng thời gian hệ thống tìm kiếm lời giải cho một map nào đó, nếu thời gian bị time out, tức là không tìm thấy phương hướng giải quyết cho trò chơi ở thuật toán và map đang được chọn thì màn hình sẽ hiển thị:

Hình 42: Giao diện không tìm thấy lời giải

4.2.4. Giao diện bắt đầu trò chơi

Sau khi đã tìm ra lời giải cho một map nào đó, hệ thống sẽ tự động chuyển sang giao diện bắt đầu trò chơi, ở giao diện này sẽ hiện ra số bước di chuyển của chú chó. Tại đây chú cún corgi sẽ di chuyển trên map để di chuyển tất cả cục xương về những ngôi nhà.



Hình 43: Giao diện bắt đầu trò chơi

4.2.5. Giao diện khi tìm được kết quả

Sau khi chú chó đưa được hết các cục xương về nhà, tức là đã hoàn thành game thì giao diện sẽ chuyển sang:

Hình 44: Giao diện khi tìm được kết quả

Sau khi thắng game thì màn hình sẽ hiển thị các thông tin về:

- Tên thuật toán mình lựa chọn đề chạy game
- Số bước đi của chú chó
- Bộ nhớ đã dùng
- Số trạng thái đã duyệt
- Thời gian tìm thấy lời giải
- Số lần đẩy hộp

4.3. Hướng dẫn thực thi phần mềm

Bước 1: Người dùng sẽ chạy file main.py để khởi chạy game sokoban

Bước 2: Chọn map bằng cách nhấn vào phím < > trên keyboard

Bước 3: Lựa chọn chế độ chơi bằng cách nhấn phím space trên keyboard. Nếu chọn NORMAL sẽ là chế độ người chơi. Nếu chọn tên các thuật toán sẽ là chế độ máy tự chơi.

Bước 4: Nếu lựa chọn chế độ NORMAL thì màn hình sẽ chuyển san giao diện chơi game. Lúc này người dùng sẽ nhấn các phím (← → ↑ ↓) trên keyboard để di chuyển chú chó đẩy xương về nhà.

Bước 5: Nếu người dùng muốn tiếp tục chơi game thì nhấn ENTER để thay đổi map, chế độ chơi.

4.4. Trình bày các kết quả thử nghiệm

4.4.1. So sánh kết quả các thuật toán sau khi chạy ngẫu nhiên một số map

❖ **Map 1:**

	Algorithm	Approved States	Memory (MB)	Map Level	Time (s)	Count Push Box	Count Move
2	Euclidean Distance Heuristic	664	63.82421875	1	0.280838728	134	82
3	Manhattan Distance Heuristic	472	64.34765625	1	0.04499507	92	92
4	Breadth First Search	1032	64.80078125	1	0.11493969	199	70
5	Depth First Search	318	64.8046875	1	0.035627127	50	82
6	Greedy Best First Search	664	64.75390625	1	0.146894932	134	82
7	Depth Limited Search	318	64.796875	1	0.030981064	50	82
8	Uniform Cost Search	1103	64.88671875	1	0.099149227	210	70

Hình 45: So sánh các thuật toán trên Map 1

Nhận xét kết quả: Dựa vào những thông số của hệ thống hiện thị lên màn hình tìm ra kết quả của các thuật toán, nhóm em rút ra được những phân tích sau:

- Số bước:

+ A* Manhattan > (GBFS = A* Euclidean = DFS = DLS) > (UCS = BFS)

- Số trạng thái đã duyệt:

+ (DFS = DLS) > A* Manhattan > (GBFS = A* Euclidean) > BFS > UCS

- Thời gian tìm ra lời giải:

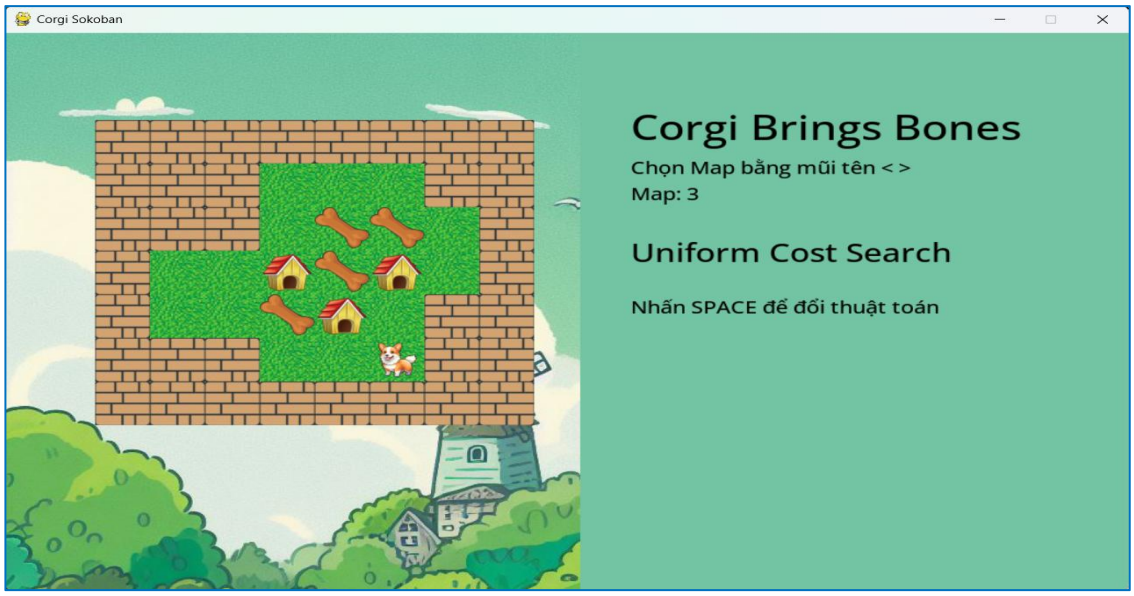
+ DLS < DFS < A* Manhattan < UCS < BFS < GBFS < A* Euclidean

Nhìn vào đặc điểm của map ta có thể thấy map này có nhiều trướng ngại vật hơn vì vậy thuật toán A* Manhattan sẽ mang lại kết quả khá tối ưu cho map này, mặc dù map này DLS và DFS lại trả về kết quả nhanh nhất nhưng có thể là do trạng thái đích nằm đúng vào nhánh duyệt sâu nên 2 thuật toán này được phát huy nổi trội .

Tiếp đến là 2 thuật toán UCS và BFS , mặc dù UCS sẽ ưu tiên trọng số ít hơn , tức là hạn chế việc đẩy thùng nhất có thể nhưng ở trường hợp này UCS lại duyệt nhiều hơn và đẩy thùng nhiều hơn , có thể lý do nằm ở việc bản đồ này có khá nhiều chướng ngại vật dẫn đến việc hạn chế đẩy thùng sẽ làm khó tìm đến lời giải hơn

Cuối cùng là 2 thuật toán GBFS và A* Euclidean , do bản đồ có nhiều chướng ngại vật, dẫn đến việc thuật toán này phải quay lui khi không tìm được lời giải nhiều hơn , dẫn đến không tối ưu

❖ **Map 3:**



	Algorithm	Approved States	Memory (MB)	Map Level	Time (s)	Count Push Box	Count Move
2	Euclidean Distance Heuristic	1084	71.28125	3	0.251799822	349	30
3	Manhattan Distance Heuristic	3466	71.2890625	3	0.472440958	1399	48
4	Breadth First Search	8231	75.0234375	3	0.873985052	3311	30
5	Depth First Search	9666	75.05859375	3	0.954888344	4094	162
6	Greedy Best First Search	1084	75.046875	3	0.241261482	349	30
7	Depth Limited Search	9664	75	3	0.682301998	4093	162
8	Uniform Cost Search	7325	75.0625	3	0.661652327	2972	32

Hình 46: So sánh các thuật toán trên Map 3

Nhận xét kết quả: Dựa vào những thông số của hệ thống hiện thị lên màn hình tìm ra kết quả của các thuật toán, nhóm em rút ra được những phân tích sau:

- Số bước:

  + (GBFS = A* Euclidean = BFS) > UCS > A* Manhattan > (DFS = DLS)

- Số trạng thái đã duyệt:

  + (GBFS = A*Euclidean) > A* Manhattan > UCS > BFS > DLS > DFS

- Thời gian tìm ra lời giải:

  + GBFS < A*Euclidean < A* Manhattan < UCS < DLS < BFS < DFS

Ở map trên ta thấy được đặc điểm là map hầu như không có chướng ngại vật , vì vậy thuật toán GBFS và A* Euclidean phát huy được ưu điểm của mình và chạy rất nhau , kế đến là thuật toán A* Manhattan .

Thuật toán UCS thì sẽ nhanh hơn thuật toán BFS do việc ưu tiên chi phí ít hơn lúc chọn và duyệt trên đồ thị có trọng số , trọng số ở đây nhóm em chọn là việc đẩy thùng sẽ làm trọng số cao hơn nên khi chạy thuật toán UCS sẽ mong đợi số lượt đẩy thùng ít hơn số lượt đẩy thùng của BFS .

Thuật toán DLS sẽ hoạt động giống như thuật toán DFS nhưng nó sẽ có giá trị limited (ở đây nhóm em chọn mặc định là 300) nó sẽ giúp ta tạo giới hạn duyệt sâu của hàm DFS và sẽ tối ưu hơn DFS nếu chọn giá trị limited hợp lý. Ở trường hợp trên thì nó đã giúp ta giảm được vài trạng thái cần duyệt dẫn đến chạy nhanh hơn DFS

Kết luận:

Ở trên chúng em chỉ chọn ra 2 map ngẫu nhiên để phân tích các kết quả thu được của mỗi thuật toán, và 28 map còn lại sẽ cho ra những kết quả khác nhau cũng như ta sẽ thấy những điểm khác biệt về các thông số mà thuật toán mang lại. Điều đó có nghĩa là không phải lúc nào A* cũng là thuật toán tối ưu nhất, cũng như không phải lúc nào DFS, BFS, UCS,... không tốt bằng A*. Chúng ta cần đánh giá dựa vào bản đồ (mức độ) trò chơi khó hay dễ, rộng hay hẹp, hàm ước lượng có tốt hay không, đồ thị đó có trọng số hay không có trọng số và có nhiều chướng ngại vật hay không.

PHẦN 5. KẾT LUẬN

5.1 Đánh giá kết quả thực hiện

Về mặt giải thuật: Đã triển khai và thực hiện thành công các thuật toán Breadth First Search, Depth First Seach, A Star, Uniform cost search, Depth Limited Search, Best First Search tích hợp chúng vào chương trình. Đồng thời, làm rõ sự khác biệt giữa các thuật toán và thực hiện so sánh một cách trực quan thông qua chương trình đã xây dựng. Tuy đã áp dụng công thức tính khoảng cách của Manhattan và Euclidean cho thuật toán A* nhưng về mặt thời gian tìm ra lời giải vẫn chưa đủ nhanh, chưa đủ thuyết phục, chúng em thừa nhận đây là một thiếu sót mà chúng em gặp phải trong quá trình thực hiện đề tài. Chúng em sẽ cố gắng tìm hiểu lại vấn đề này để giúp cho thuật toán tối ưu hơn

Về mặt giao diện: Còn nhiều ý tưởng đã được lên kế hoạch nhưng không đủ thời gian thực hiện, tuy nhiên về mặt tổng quan, nhóm đã xây dựng hầu hết những yêu cầu cần có cho game Sokoban.

5.2 Định hướng phát triển

Nhóm em định hướng sẽ phát triển game thành một trò chơi hoàn chỉnh có các tính năng phù hợp với nhu cầu của người chơi trong xã hội ngày nay.

- Thêm Map Mới: Để tăng thêm độ thú vị và khám phá cho người chơi, người chơi có thể thêm vào một loạt các map mới với các cấp độ khó khác nhau. Các map này có thể bao gồm các thử thách độc đáo như chướng ngại vật di chuyển, thời gian giới hạn….

- Tạo Map Bằng Tay: Tính năng này cho phép người chơi tự tạo ra các map của riêng mình, tạo ra một không gian sáng tạo không giới hạn. Người chơi có thể chia sẻ các map do mình tạo ra với cộng đồng, tạo ra một nguồn tài nguyên map phong phú và đa dạng.

- Bảng Xếp Hạng: Bảng xếp hạng dựa trên thời gian hoàn thành map sẽ tạo ra một môi trường cạnh tranh lành mạnh giữa người chơi. Người chơi sẽ được thách thức để hoàn thành map nhanh nhất có thể và so sánh kết quả của mình với người chơi khác.

- Tùy Chỉnh Nhân Vật: Tính năng này cho phép người chơi tùy chỉnh nhân vật của mình theo sở thích cá nhân. Người chơi có thể thay đổi hình dạng, màu sắc, và các phụ kiện cho nhân vật của mình, tạo ra một trải nghiệm chơi game cá nhân hóa.

PHẦN TÀI LIỆU THAM KHẢO

1. Tham khảo thuật toán A*:

Iostream(08/08/2020), SHICHIKI LÊ, truy cập ngày 22/11/2023, đường dẫn: https://www.iostream.co/article/thuat-giai-a-DVnHj

2. Tham khảo thuật toán Best First Search:

Viblo(13/01/2019), Thơ Trần, truy cập ngày 22/11/2023, đường dẫn: https://viblo.asia/p/cac-thuat-toan-co-ban-trong-ai-phan-biet-best-first-search-va-uniform-cost-search-ucs-Eb85omLWZ2G

3. Tham khảo thuật toán Depth First Search:

Viettuts(2016), ngày truy cập 22/11/2023, đường dẫn: https://viettuts.vn/cau-truc-du-lieu-va-giai-thuat/giai-thuat-tim-kiem-theo-chieu-sau-depth-first-search

Tài liệu môn học Trí Tuệ Nhân Tạo

4. Lê Thanh Hương - Viện Công nghệ thông tin và Truyền thông Trường Đại Học Bách Khoa Hà Nội, *Trí Tuệ Nhân Tạo - Chương 3: Tìm kiếm Heuristic* (File PDF)

5. vi.wikipedia.org(2023), Khoảng cách Euclid, truy cập ngày 2/12/2023, truy cập tại link: https://vi.wikipedia.org/wiki/Khoảng_cách_Euclid

6. @linhdb (2021), Distance Measure trong Machine learning, truy cập ngày: 2/12/2023, truy cập tại link: https://viblo.asia/p/distance-measure-trong-machine-learning-ByEZkopYZQ0

7. vi.wikipedia.org(2023), Khoảng cách Manhattan, truy cập ngày 2/12/2023, truy cập tại link: https://vi.wikipedia.org/wiki/Khoảng_cách_Manhattan